ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ТЕХНОЛОГІЙ ЦИФРОВОГО РОЗВИТКУ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Інтелектуальний web-застосунок для управління персональною продуктивністю з використанням методів гейміфікації та пріоритезації»

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

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

_____ Ростислав ОВОДОВ
(підпис)

Виконав: здобувач вищої освіти групи ТЦР-44

Ростислав ОВОДОВ

Керівник: Артур ГАВОР
ст. викладач кафедри ТЦР

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Технологій цифрового розвитку

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Технології цифрового розвитку»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Технологій цифрового розвитку
_____ Вікторія ЖЕБКА
« ____ » _____ 2026 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

Оводову Ростиславу Олексійовичу

1. Тема кваліфікаційної роботи: «Інтелектуальний web-застосунок для управління персональною продуктивністю з використанням методів гейміфікації та пріоритезації»
керівник кваліфікаційної роботи викладач кафедри ТЦР Артур ГАВОР,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року № 51.

2. Строк подання кваліфікаційної роботи «18» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про методи тайм-менеджменту та оцінки пріоритетності завдань (алгоритм матриці Ейзенхауера), дослідження в галузі когнітивної психології та застосування ігрових механік (гейміфікації) у неігрових процесах, технічна документація середовищ розробки (Node.js, React) та систем управління реляційними базами даних (PostgreSQL), документація технологій контейнеризації (Docker).

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

1. Аналіз методів та наявних програмних рішень для управління персональною продуктивністю.
2. Проєктування архітектури, бази даних та інтерфейсу гейміфікованого web-застосунку.
3. Програмна реалізація та розгортання клієнт-серверної системи на базі матриці Ейзенхауера.
4. Комплексне тестування розробленого застосунку.

5. Перелік графічного матеріалу: *презентація*
   1. Аналіз предметної області та обґрунтування концепції.
   2. Проєктування системи (Use Case та ER-моделювання).
   3. Програмна реалізація: технології, алгоритми та розгортання.
   4. Практичний результат: інтерфейс користувача.
   5. Апробація результатів дослідження.

6. Дата видачі завдання «20» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	09.02.2026 – 20.02.2026	Виконано
2	Аналіз та дослідження існуючих аналогів (програм для тайм-менеджменту)	21.02.2026 – 28.02.2026	Виконано
3	Огляд алгоритмів управління продуктивністю, матриці Ейзенхауера та методів гейміфікації	*01.03.2026 – 12.03.2026*	Виконано
4	Проєктування архітектури, баз даних та UX/UI інтерфейсу застосунку продуктивності	13.03.2026 – 26.03.2026	Виконано
5	Програмна реалізація (Frontend, Backend) та налаштування середовища розгортання	27.03.2026 – 10.04.2026	Виконано
6	Тестування застосунку	11.04.2026 – 14.04.2026	Виконано
7	Оформлення роботи: вступ, висновки, реферат	15.04.2026 – 18.04.2026	Виконано
8	Розробка демонстраційних матеріалів (презентація)	*19.04.2026 – 20.04.2026*	Виконано
9	Попередній захист роботи	11.05.2026 – 22.05.2026	Виконано

Здобувач вищої освіти _____ Ростислав ОВОДОВ
(підпис)

Керівник
кваліфікаційної роботи _____ Артур ГАВОР
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 58 стор., 6 табл., 14 рис., 26 джерела.

Мета роботи – підвищення ефективності виконання завдань і зниження когнітивного навантаження при плануванні за рахунок розробки програмної системи, що поєднує метод Ейзенхауера з мотиваційною гейміфікацією.

Об'єкт дослідження – процес організації, пріоритезації та контролю виконання особистих завдань користувача.

Предмет дослідження – програмне забезпечення для підвищення персональної продуктивності з використанням алгоритму матриці Ейзенхауера та ігрових механік.

Короткий зміст роботи: В роботі проаналізовано проблему прокрастинації та досліджено вплив ігрових механік на формування стійкої мотивації. Проаналізовано існуючі програмні засоби для тайм-менеджменту: Todoist, Trello, TickTick. Розроблено архітектуру, ER-модель реляційної бази даних та алгоритм роботи застосунку, який автоматично класифікує завдання за квадрантами важливості/терміновості та нараховує користувачу досвід (XP) і рівні за їх виконання. Програмно реалізовано клієнт-серверну систему з можливістю авторизації (JWT JWT-токени). Проведено розгортання системи в ізольованому контейнеризованому середовищі із забезпеченням зовнішнього доступу через безпечний тунель. В роботі використано бібліотеку React та Tailwind CSS для фронтенду, платформу Node.js (Express.js) для бекенду, СУБД PostgreSQL та технологію Docker Compose.

КЛЮЧОВІ СЛОВА: ТАЙМ-МЕНЕДЖМЕНТ, МАТРИЦЯ ЕЙЗЕНХАУЕРА, ГЕЙМІФІКАЦІЯ, ПРОКРАСТИНАЦІЯ, WEB-ЗАСТОСУНОК, REACT, NODE.JS, POSTGRESQL.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ....8
ВСТУП....9
1. АНАЛІЗ ПРОБЛЕМИ УПРАВЛІННЯ ПЕРСОНАЛЬНОЮ ПРОДУКТИВНІСТЮ ТА ІСНУЮЧИХ РІШЕНЬ....10
1.1. Опис предметної галузі та проблематики прокрастинації....10
1.2. Аналіз існуючих програмних продуктів для управління завданнями та їх недоліки....12
1.3. Огляд ІТ-засобів та технологій для реалізації....15
2. ПРОЄКТУВАННЯ ІНТЕЛЕКТУАЛЬНОГО WEB-ЗАСТОСУНКУ....17
2.1. Проєктування загальної архітектури та структури системи....17
2.2. Моделювання вимог до системи з використанням методів гейміфікації та пріоритезації....20
2.3. Проєктування структури реляційної бази даних PostgreSQL....22
2.4. Проектування UI/UX інтерфейсу користувача....26
3. ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ....31
3.1. Реалізація серверної частини та RESTful API....31
3.2. Реалізація бізнес-логіки: матриця Ейзенхауера та система нарахування досвіду....37
3.3. Розробка клієнтської частини застосунку....40
4. ТЕСТУВАННЯ ПРОГРАМИ....45
4.1. Обґрунтування використання обраних видів тестування програми....45
4.2. Розробка тест-кейсів для перевірки основного функціоналу та інтерфейсу користувача....47
4.3. Результати тестування та перевірка експлуатаційної здатності....52
ВИСНОВКИ....58
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ....60
ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ....63
ДОДАТОК Б. ПРОГРАМНИЙ КОД СЕРВЕРНОЇ ЧАСТИНИ WEB-ЗАСТОСУНКУ. 64
ДОДАТОК В. ПРОГРАМНИЙ КОД КЛІЄНТСЬКОЇ ЧАСТИНИ ТА ВІЗУАЛІЗАЦІЇ МАТРИЦІ ЕЙЗЕНХАУЕРА....68
ДОДАТОК Г. КОНФІГУРАЦІЙНІ ФАЙЛИ КОНТЕЙНЕРИЗАЦІЇ ТА ПРОКСІ-СЕРВЕРА....79
ДОДАТОК Г. КЕРІВНИЦТВО КОРИСТУВАЧА ТА ІНСТРУКЦІЯ З ЕКСПЛУАТАЦІЇ....81

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД – База даних

СУБД – Система управління базами даних

API (Application Programming Interface) – Інтерфейс прикладного програмування

CRUD (Create, Read, Update, Delete) – Створення, читання, оновлення, видалення (чотири базові функції управління даними)

CSS (Cascading Style Sheets) – Каскадні таблиці стилів

DOM (Document Object Model) – Об'єктна модель документа

HTML (HyperText Markup Language) – Мова гіпертекстової розмітки

HTTP (HyperText Transfer Protocol) – Протокол передачі гіпертексту

JSON (JavaScript Object Notation) – Текстовий формат обміну даними

JWT (JSON Web Token) – Захищений вебтокен у форматі JSON (використовується для автентифікації)

REST (Representational State Transfer) – Передача репрезентативного стану (архітектурний стиль для мережевих застосунків)

SPA (Single Page Application) – Односторінковий web-застосунок

SQL (Structured Query Language) – Мова структурованих запитів для роботи з реляційними базами даних

UI (User Interface) – Інтерфейс користувача (візуальна частина програми)

UX (User Experience) – Досвід користувача (зручність взаємодії з програмою)

XP (Experience Points) – Бали досвіду (одиниця винагороди у системі гейміфікації)

ВСТУП

Актуальність теми. У сучасному світі, що характеризується стрімким розвитком інформаційних технологій та постійним збільшенням обсягу інформаційного шуму, проблема ефективного управління власним часом набуває виняткової важливості. Значна частина фахівців у галузі ІТ, студентів та менеджерів стикається з явищем прокрастинації, що виникає внаслідок "когнітивного перевантаження" – нездатності мозку швидко структурувати та ранжувати накопичені завдання. Традиційні підходи до тайм-менеджменту часто фокусуються лише на фіксації завдань (лінійні списки справ), що частково вирішує проблему їх збереження, але не забезпечує належної мотивації для початку роботи. Метод Ейзенхауера (розподіл завдань за шкалами "важливості" та "терміновості") зарекомендував себе як дієвий механізм пріоритезації. Однак класична концепція цього методу позбавлена інструментів позитивного психологічного підкріплення. Впровадження ігрових механік (гейміфікації), таких як нарахування балів досвіду (XP) та прогресивна рівнева система, дозволяє компенсувати брак внутрішньої мотивації потужними зовнішніми стимулами. Таким чином, проєктування спеціалізованих вебсистем, що об’єднують ці два підходи для забезпечення високої продуктивності, є надзвичайно актуальним науково-практичним завданням сучасної інженерії програмного забезпечення.

Мета роботи – підвищення ефективності виконання завдань і зниження когнітивного навантаження користувача шляхом розробки інтелектуальної системи, яка поєднує матрицю Ейзенхауера з механізмами гейміфікації. Для її досягнення передбачено дослідити психологічні причини прокрастинації та існуючі інструменти тайм-менеджменту, після чого спроєктувати архітектуру й реляційну базу даних клієнт-серверної системи. Також заплановано розробити алгоритм класифікації завдань і математичну модель нарахування досвіду за їх пріоритетністю, а на завершення — обґрунтувати методику контейнеризованого розгортання системи із забезпеченням захищеного віддаленого доступу.

Об'єкт дослідження – процес організації, пріоритезації та контролю виконання особистих завдань.

Предмет дослідження – методи та програмні інструменти (платформа Node.js, бібліотека React) для проєктування інформаційної системи з використанням алгоритму матриці Ейзенхауера та ігрових механік.

Методами дослідження є порівняльний аналіз (для огляду аналогів), системний аналіз (при проєктуванні архітектури бази даних), об'єктно-орієнтоване проєктування, а також функціональне тестування програмного забезпечення.

Практична значущість одержаних результатів полягає у спроєктуванні готової до експлуатації архітектури програмного забезпечення, що забезпечує автоматизовану класифікацію завдань за чотирма квадрантами. Запропонована інфраструктурна модель, побудована на принципах контейнеризації (Docker) та захищеного тунелювання мережевого трафіку (Cloudflare Zero Trust), може бути використана IT-компаніями та незалежними розробниками як універсальний еталонний шаблон для розгортання високонавантажених безпечних вебсистем.

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

1. Оводов Р. О., Гавор А. С. Аналіз методів гейміфікації та алгоритмів пріоритетизації в системах управління персональною продуктивністю. // Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях».
2. Оводов Р. О., Гавор А. С. Проєктування та технічна реалізація web-застосунку для управління продуктивністю з елементами гейміфікації. // Матеріали Всеукраїнської науково-технічної конференції «Технології цифрового розвитку: програмні системи та децентралізація».

1. АНАЛІЗ ПРОБЛЕМИ УПРАВЛІННЯ ПЕРСОНАЛЬНОЮ ПРОДУКТИВНІСТЮ ТА ІСНУЮЧИХ РІШЕНЬ

1.1. Опис предметної галузі та проблематики прокрастинації

В епоху глобальної цифровізації, експоненційного зростання обсягів інформації та безперервного комунікаційного потоку, здатність до ефективного управління власним часом (тайм-менеджмент) стає не просто корисною навичкою, а одним із найважливіших факторів, що фундаментально визначають професійну кваліфікацію та академічну успішність сучасної людини. Разом з тим, надзвичайно стрімкий темп життя сприяє виникненню небезпечного психологічного явища когнітивного перевантаження. Цей стан настає тоді, коли обсяг та складність вхідних завдань значно перевищують здатність робочої пам'яті людини їх адекватно структурувати, обробляти та своєчасно виконувати. Прямим наслідком такої когнітивної невідповідності є масова прокрастинація – складний психологічний механізм ірраціонального, систематичного відкладання виконання суб'єктивно важливих чи термінових завдань на користь менш значущих, але більш легких для мозку дій, що забезпечують миттєве задоволення (перегляд соціальних мереж, виконання рутинної механічної роботи тощо).

Згідно з масштабним науковим дослідженням проблеми людської саморегуляції, проведеним авторитетним канадським доктором психології Пірсом Стілом (Piers Steel), феномен прокрастинації у сучасному суспільстві набув загрозливих ознак справжньої епідемії. Відповідно до зібраної ним глобальної статистики [1], від 80% до 95% здобувачів вищої освіти та фахівців, що займаються складною інтелектуальною працею (IT-інженери, менеджери, аналітики), регулярно стикаються з проблемою хронічного відкладання справ. Причому майже 50% респондентів чітко ідентифікують власну прокрастинацію як постійну та проблемну модель поведінки, що безпосередньо призводить до

суттєвого зниження їхньої життєвої ефективності, втрати кар'єрних можливостей та виникнення перманентного відчуття провини і стресу.

Тривалий час в академічних та бізнес-колах помилково вважалося, що корінь проблеми лежить виключно у невмінні людини механічно планувати свій розклад. Саме через це традиційні програмні засоби та інструменти підвищення продуктивності фокусувалися на банальному створенні лінійних списків справ (так званих To-Do lists). Однак сучасні дослідження у предметній галузі нейробіології доводять, що прокрастинація є насамперед проблемою регуляції емоцій, а не просто дефіцитом навичок тайм-менеджменту. Простий візуальний перегляд неструктурованого лінійного списку, що складається з 10-15 однотипних завдань, викликає у користувача глибоке емоційне виснаження ще до початку власне роботи. Цей феномен отримав назву «параліч вибору» (Choice Paralysis) – стан, при якому мозок витрачає так багато енергії на вирішення того, з якої задачі почати, що на само виконання роботи ресурсів вже не залишається.

Для мінімізації такого когнітивного навантаження під час процесу прийняття рішень був історично розроблений потужний алгоритмічний підхід – Матриця Ейзенхауера. Суть цього методу полягає у жорсткій сепарації всіх вхідних завдань за двома незалежними декартовими координатами: шкалою «Важливість» і шкалою «Терміновість». Перетин цих осей формує чотири окремі логічні квадранти матриці. Такий строгий математичний розподіл дозволяє користувачу моментально ідентифікувати завдання першого квадранту ("Важливо і Терміново"), тим самим забезпечуючи швидкий, підсвідомий алгоритм для початку роботи без необхідності витрачати дефіцитні психологічні ресурси на аналіз та пріоритезацію списку.

Однак, якщо розглядати цю систему з точки зору сучасної поведінкової психології, класична Матриця Ейзенхауера у своєму суто паперовому чи чисто алгоритмічному вигляді також має фундаментальний недолік. Вона ефективно обробляє логічну складову процесу, але повністю ігнорує мотиваційну. Виконання глобальних завдань другої групи ("Важливе, але не термінове") традиційно вимагає найбільших вольових зусиль через їхній обсяг та відсутність жорстких

дедлайнів. Такі задачі не надають миттєвої винагороди чи викидів нейромедіатора дофаміну в мозок користувача, через що підсвідомість людини схильна всіляко уникати роботи в цьому стратегічному кластері.

Відтак, еволюція предметної галузі сучасної розробки інформаційного програмного забезпечення безальтернативно вимагає створення таких інноваційних web-застосунків, які гармонійно поєднують строгий алгоритм сортування завдань за матрицею Ейзенхауера з ефективними елементами гейміфікації (інтеграцією віртуальних нагород, накопиченням балів ігрового досвіду, динамічними рівнями та прогрес-барами). Штучне створення такого швидкого та позитивного "дофамінового циклу" (Dopamine Loop) безпосередньо після фізичного виконання завдання надійно нівелює ірраціональне бажання користувача відкладати справи на потім.

Таким чином, інженерне проєктування та розробка спеціалізованого інформаційного середовища, яке бере на себе здійснення інтелектуального автоматичного розподілу задач користувача на базі координатної матриці Ейзенхауера, і водночас безперервно моніторить та заохочує виконання цих завдань інтегрованою системою персональної гейміфікації, є найбільш оптимальним, сучасним та науково обґрунтованим рішенням для подолання глобальної проблеми прокрастинації.

1.2. Аналіз існуючих програмних продуктів для управління завданнями та їх недоліки

Для обґрунтування доцільності та формулювання вимог до програмного забезпечення, що розробляється в межах даної кваліфікаційної роботи, було проведено аналіз сучасного ринку застосунків для управління завданнями та тайм-менеджменту. Ринок пропонує широкий спектр рішень, які умовно можна поділити на класичні «списки справ» (To-Do lists), платформи для командної роботи на базі Kanban-дошок та вузькоспеціалізовані трекери звичок (Habit

trackers).

З метою визначення переваг та виявлення фундаментальних недоліків було обрано три репрезентативні програмні продукти, що є лідерами у своїх нішах: Todoist [2], Trello [3] та Habitica [4].

Todoist є одним із найпопулярніших у світі класичних застосунків для управління персональними завданнями. Його архітектура базується на лінійній ієрархії (проєкти, завдання, підзавдання) з прив'язкою до календарних дедлайнів. Переваги: мінімалістичний інтерфейс, гнучка система нагадувань, ефективна інтеграція з іншими календарями, функціональність розпізнавання дат у тексті. Недоліки: застосунок не має вбудованої алгоритмічної пріоритезації. Користувач повинен самостійно визначати пріоритет (P1-P4), що залишає незмінним рівень когнітивного навантаження. Елементи гейміфікації (система Todoist Karma) є глибоко прихованими в налаштуваннях профілю і не забезпечують миттєвого візуального чи психологічного підкріплення.

Trello – це потужна система управління проєктами, що використовує парадигму канбан-дошок (Kanban boards). Переваги: високий рівень візуалізації процесів («робиться», «на перевірці», «виконано»), відмінні можливості для командної співпраці та делегування завдань. Недоліки: продукт є надлишково складним для персонального використання. Trello не стимулює користувача обирати найважливіше завдання, оскільки будь-яка картка на дошці виглядає однаково. Індивідуальна мотивація і механіки гейміфікації повністю відсутні.

Habitica – це гейміфікований застосунок для управління завданнями, який перетворює життя користувача на рольову гру (Role-Playing Game). Переваги: потужна система позитивного підкріплення (досвід, золоті монети, екіпірування персонажа), що ефективно вирішує проблему браку мотивації у користувачів з хронічною прокрастинацією. Недоліки: система повністю ігнорує концепцію важливості та терміновості (відсутня матриця Ейзенхауера). Нарахування балів відбувається лінійно, незалежно від того, чи виконав користувач дрібну рутинну справу, чи здав важливий проєкт. Це призводить до імітації продуктивності, коли користувач виконує лише прості задачі для отримання балів.

З огляду на проведений аналіз, у таблиці 1.1 наведено порівняльну характеристику досліджуваних систем із програмним продуктом, що розробляється в межах даної бакалаврської роботи (умовно названим «Intellectual Productivity App»).

Таблиця 1.1

Порівняльний аналіз програмних продуктів для тайм-менеджменту

Критерій функціональності	Todoist	Trello	Habitica	Розроблювана система
Класифікація за Матрицею Ейзенхауера	Ні	Ні	Ні	Так(Автоматична)
Прогресивна система нарахування XP	Ні	Ні	Так	Так
Залежність винагороди від пріоритету	Ні	Ні	Ні	Так(Динамічна)
Візуальне мотиваційне підкріплення	Низьке	Низька	Високе	Високе
Складність інтеграції для 1 користувача	Низька	Висока	Середня	Низька

Проведений аналіз існуючих аналогів підтверджує наявність незадоволеної потреби у ніші індивідуального тайм-менеджменту. Класичні додатки фокусуються на фіксації даних (Todoist, Trello), залишаючи осторонь поведінкову мотивацію користувача. Спеціалізовані гейміфіковані рішення (Habitica)

забезпечують мотивацію, але не містять систем фільтрації через Матрицю Ейзенхауера. Відповідно, розробка запропонованої системи, яка об'єднає гейміфікацію з алгоритмом Ейзенхауера (де бали досвіду динамічно залежать від квадранту задачі), має високий рівень практичної новизни та перевершує аналоги в розрізі зниження когнітивного навантаження.

1.3. Огляд ІТ-засобів та технологій для реалізації

Вибір клієнт-серверної архітектури та концепції SPA (Single Page Application) обумовлений потребою у високій продуктивності, масштабованості та обробці динамічних даних, таких як відслідковування завдань і розрахунок ігрового досвіду в реальному часі.

Для розробки інтерфейсу користувача було обрано бібліотеку React [5] у поєднанні з інструментом збірки Vite. Вибір React обґрунтований його компонентною архітектурою та концепцією віртуального DOM (Virtual DOM), що є критично важливими при реалізації ігрових механік (гейміфікації). Компонентний підхід дозволяє незалежно оновлювати візуальний прогрес користувача (шкалу досвіду, інформер рівня) без необхідності перезавантаження всієї матриці завдань, забезпечуючи високу реактивність застосунку. Для стилізації компонентів було інтегровано CSS-фреймворк Tailwind CSS. На відміну від традиційного використання окремих таблиць стилів, Tailwind використовує утилітний підхід (Utility-first), що суттєво зменшує загальний обсяг бандлу та стандартизує дизайн-систему (Glassmorphism), полегшуючи розробку адаптивних екранних форм.

Серверну частину системи побудовано на програмній платформі Node.js [6] із використанням мінімалістичного фреймворку Express.js. Забезпечення обміну даними між клієнтом та сервером відбувається за архітектурним стилем RESTful API (поверх протоколу HTTP). Використання Node.js забезпечує ефективну роботу з великою кількістю одночасних ізольованих клієнтських запитів за

рахунок асинхронної неблокуючої моделі вводу-виводу (Event Loop). Окрім того, обрання єдиної мови програмування (JavaScript) для повного стеку розробки (Full-Stack) дозволяє знизити обсяг розбіжностей при обміні даними (зокрема конструюванні та валідації об'єктів формату JSON).

Враховуючи, що система передбачає чітку ієрархічну прив'язку даних (багато завдань належать одному користувачу) та вимагає транзакційної цілісності при нарахуванні балів досвіду, як Систему управління базами даних (СУБД) було обрано PostgreSQL [7] – потужну об'єктно-реляційну СУБД. Індексування даних в PostgreSQL та її оптимізація для виконання складних математичних запитів гарантує стабільний розрахунок накопичувальних параметрів (рівнів користувача) навіть при збільшенні навантаження на систему.

Для забезпечення відмовостійкості, безпеки і незалежності середовищ розробки (Development) та експлуатації (Production) було використано технологію контейнеризації Docker. Усі мікросервіси (база даних, бекенд і фронтенд-вебсервер Nginx) запаковано в незалежні Docker-контейнери, керовані через інструмент docker-compose. Окрім того, для доступу кінцевого споживача до застосунку було інтегровано рішення Cloudflare Zero Trust (Tunnels), що забезпечує криптографічний захист з'єднання без зовнішньої експозиції IP-адреси центрального сервера.

Критичний огляд ринку показав, що традиційні інструменти тайм-менеджменту (To-Do списки, канбан-дошки) лише фіксують інформацію. Вони не мають механізмів гейміфікації для мотивації користувача та перекладають рутинну пріоритезацію завдань безпосередньо на людину.

Для усунення цих недоліків обґрунтовано актуальність розробки інтелектуального web-застосунку. Його концептуальною основою обрано синергію матриці Ейзенхауера (автоматичний розподіл за важливістю й терміновістю) та гейміфікації (система рівнів і досвіду). Для реалізації системи визначено оптимальний технологічний стек, що включає екосистему JavaScript, реляційну СУБД та інструменти контейнеризації.

2. ПРОЄКТУВАННЯ ІНТЕЛЕКТУАЛЬНОГО WEB-ЗАСТОСУНКУ

2.1. Проєктування загальної архітектури та структури системи

Процес розробки програмного забезпечення на етапі проєктування вимагає чіткого визначення архітектурних патернів, які забезпечать масштабованість, безпеку та зручність обслуговування застосунку [8]. Зважаючи на вимоги до розроблюваної системи продуктивності (забезпечення гейміфікації в режимі реального часу, централізоване збереження інформації та мультиплатформений доступ), за архітектурну основу було обрано багатошарову Клієнт-Серверну модель (Client-Server Architecture).

Дана модель передбачає фізичне і логічне розділення застосунку на дві ключові складові:

1. Клієнтська частина (Frontend) – забезпечує користувацький інтерфейс, відображення Матриці Ейзенхауера, анімацію нарахування XP та взаємодію з користувачем. Запускається безпосередньо у веббраузері користувача.
2. Серверна частина (Backend) – централізований вузол, що містить бізнес-логіку класифікації завдань, розрахунку досвіду, автентифікації та взаємодії з базою даних.

З метою уникнення проблем "монолітної" архітектури та забезпечення зручності розгортання, розроблена система базується на принципах контейнеризації мікросервісів. Структуру системи поділено на три окремі ізольовані Docker-контейнери, кожен з яких виконує єдину відповідальність.

Контейнер бази даних (PostgreSQL): зберігає персистентні дані про користувачів та їхні завдання. Контейнер ізольований у внутрішній віртуальній мережі (app-network) і не має прямого виходу в мережу Інтернет, що унеможливлює зовнішні атаки на СУБД.

API-сервер (Node.js/Express): здійснює обробку REST-запитів. Контейнер приєднано до мережі БД і налаштовано на прослуховування внутрішнього порту 5000. Всі алгоритми розрахунку гейміфікації інкапсульовано саме на цьому рівні.

Frontend / Reverse-Proxy сервер (Nginx): має дві ролі. По-перше, Nginx здійснює роздачу статичних скомпільованих файлів React-додатку. По-друге, він виконує роль зворотного проксі-сервера (Reverse Proxy): перехоплює будь-які запити, що починаються з /api/, і маршрутизує їх всередині Docker-мережі до Node.js сервера. Це вирішує проблему Cross-Origin Resource Sharing (CORS) та дозволяє системі працювати під одним доменом.

Для забезпечення максимально безпечного виходу програмної системи в глобальну мережу Інтернет було прийнято рішення відмовитися від класичного фізичного відкриття портів на маршрутизаторі (Port Forwarding). Замість цього загальну архітектуру проєкту було інтегровано з сучасною технологією Cloudflare Zero Trust (механізм Cloudflare Tunnels). Такий підхід дозволяє повністю приховати справжню IP-адресу хост-сервера від зовнішніх мережевих сканувань та суттєво зменшити площу можливих кібератак (Attack Surface).

Технічна реалізація цього процесу полягає у розгортанні спеціального ізольованого контейнера-демона (cloudflared). Цей сервіс ініціює виключно вихідні з'єднання (outbound traffic) від локального сервера до найближчого вузла глобальної магістралі Cloudflare, створюючи захищений тунель зв'язку. Оскільки сервер не приймає прямих вхідних запитів із мережі, його брандмауер (firewall) залишається закритим для зовнішнього світу.

Завдяки маршрутизації через інфраструктуру Cloudflare, вебзастосунок автоматично отримує розширений захист від розподілених атак на відмову в обслуговуванні (DDoS-захист) ще до того, як шкідливий трафік досягне сервера. Крім того, на стороні Cloudflare забезпечується автоматичне термінування SSL/TLS-сертифікатів, що гарантує надійне наскрізне шифрування користувацьких даних. Після проходження всіх перевірок безпеки на магістралі, легітимний публічний трафік передається через тунель безпосередньо у Frontend-контейнер. Ця передача відбувається виключно в межах віртуальної

ізольованої мережі Docker (web_network), що гарантує цілісність даних і унеможливлює їх перехоплення на рівні операційної системи хоста.

Логічну архітектуру розробленої системи наведено на рисунку 2.1.

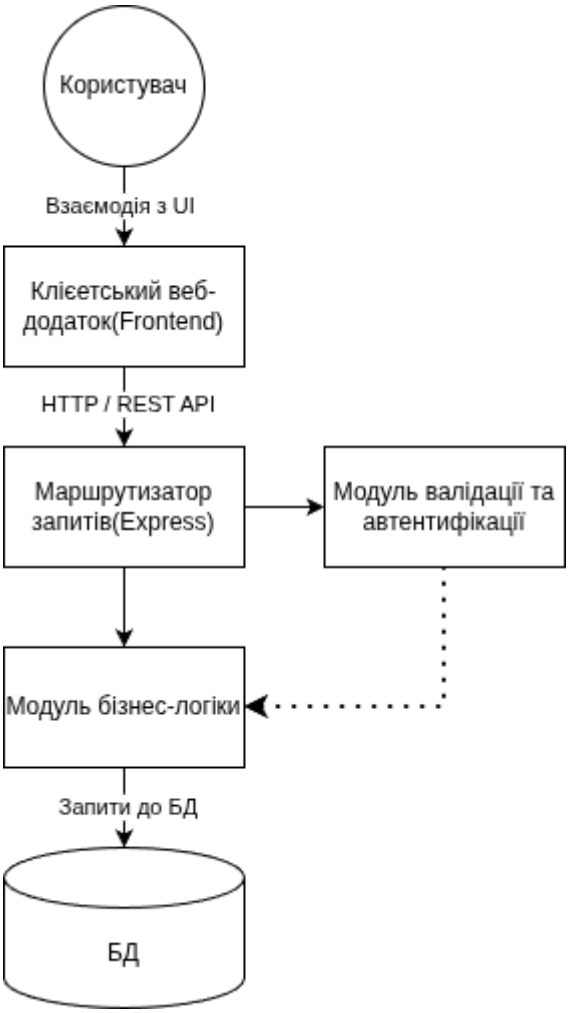


Рис. 2.1. Архітектура розробленої системи

Функціональне розділення системи на клієнтську та серверну складові дозволяє повністю приховати алгоритми підрахунку гейміфікаційних нагород (XP) на бекенді, унеможливлюючи "накрутку" або фальсифікацію прогресу користувачем з боку браузера.

2.2. Моделювання вимог до системи з використанням методів гейміфікації та пріоритезації

Етап моделювання вимог є критичним кроком у життєвому циклі програмного забезпечення, що дозволяє формалізувати функціональні та нефункціональні очікування від системи [9]. Оскільки розроблюваний застосунок поєднує класичний тайм-менеджмент із психологічними механіками, вимоги до нього поділяються на кілька логічних площин: управління обліковими записами, алгоритмізація завдань за Ейзенхауером та мотиваційна гейміфікаційна складова.

Серед ключових функціональних вимог до розроблюваного застосунку можна виокремити декілька основних. По-перше, система повинна дозволяти користувачам реєструвати унікальні облікові записи та здійснювати безпечний вхід із використанням механізму хешування паролів та видачі JWT-токенів. По-друге, під час створення нових завдань користувачем, система зобов'язана вимагати встановлення двох логічних прапорців: "Терміново" (is_urgent) та "Важливо" (is_important). На базі цих параметрів алгоритм повинен самостійно аналізувати задачу і автоматично розміщувати її у відповідному з чотирьох квадрантів Матриці Ейзенхауера на головному екрані. Крім того, при підтвердженні виконання задачі, система повинна вираховувати бали досвіду пропорційно до складності її квадранту (наприклад, 30 XP за перший квадрант і лише 5 XP за четвертий) та динамічно транслювати цей прогрес через візуальний інтерфейс.

Щодо критичних нефункціональних вимог, головними пріоритетами є реактивність інтерфейсу та безпека даних. Зокрема, відображення шкали прогресу рівня гравця та запуск анімації винагороди повинні відбуватися без повного перезавантаження вебсторінки, що забезпечується використанням архітектури React SPA. Водночас для забезпечення захисту системи, паролі не повинні зберігатися у базі даних у відкритому вигляді (для їх шифрування використовується алгоритм bcrypt), а користувацькі сесії мають адмініструватися

виключно через бездержавні токени доступу (Stateless JWT).

З метою візуалізації функціональних вимог, згідно зі стандартом інженерії програмного забезпечення [9], було розроблено діаграму варіантів використання (Use Case Diagram), яка ілюструє основні сценарії взаємодії актора (Автентифікованого користувача) з інформаційною системою.

Рис. 2.2. Діаграма варіантів використання користувацької взаємодії з системою гейміфікації

Запроєктована модель вимог забезпечує фундаментальну концептуальну перевагу впроваджуваного продукту порівняно з існуючими аналогами. На відміну від класичних To-Do списків, які виконують лише функцію статичного фіксування записів і ніяк не стимулюють до дій, розроблена програмна логіка створює комплексну екосистему продуктивності. Вона автоматично пов'язує рівень важливості та терміновості завдань (через їх інтелектуальне сортування за матрицею Ейзенхауера) із миттєвим психологічним заохоченням (через систему нарахування ігрових балів та прогрес рівнів). Таким чином, формується повний мотиваційний цикл безпосередньо у коді програми: від правильного розставлення пріоритетів на етапі створення задачі до візуалізованої нагороди після її успішного завершення. Такий підхід дозволяє ефективно боротися з явищем прокрастинації та суттєво знижує когнітивне навантаження на користувача.

2.3. Проєктування структури реляційної бази даних PostgreSQL

Відповідно до концепції відокремлення логіки від даних, ключовим завданням етапу проєктування є розробка нормалізованої моделі зберігання інформації [10]. Вибір системи управління базами даних (СУБД) є одним із найкритичніших архітектурних рішень при проєктуванні гейміфікованих застосунків. Сучасний ринок пропонує дві глобальні парадигми: реляційні (SQL) та нереляційні (NoSQL) бази даних. Враховуючи специфіку розроблюваної системи, яка передбачає постійне оновлення ігрового балансу (нарахування балів досвіду XP) та зміну статусів завдань, було проведено їх порівняльний аналіз.

Нереляційні бази даних (наприклад, MongoDB), які базуються на зберіганні документо-орієнтованих структур, забезпечують високу швидкість розгортання та гнучкість схеми (schema-less). Однак їхнім фундаментальним недоліком у контексті даного проєкту є слабка підтримка транзакційності на рівні складних операцій. Оскільки логіка гейміфікації вимагає одночасного оновлення статусу задачі та зміни загального балансу користувача (а іноді й підвищення його рівня), виникає високий ризик виникнення стану "гонки даних" (Race condition). У разі збою системи або паралельного виконання кількох API-запитів користувачем, нереляційна модель не здатна гарантувати атомарність багатодокументних транзакцій "із коробки", що може призвести до нечесного нарахування XP або втрати прогресу.

Враховуючи вимоги до чіткої структури даних, використання нереляційних рішень (NoSQL) було відкинуто на користь класичної СУБД PostgreSQL. Головною причиною такого вибору стала підтримка транзакцій та відповідність принципам ACID. Для застосунку з елементами гейміфікації це критично важливо, оскільки процеси позначення завдання як виконаного, нарахування балів (XP) та оновлення рівня гравця мають виконуватися як одна неподільна дія. Якщо на якомусь із цих етапів станеться збій, база даних автоматично відкотить усі зміни назад (ROLLBACK). Це гарантує, що прогрес користувача не буде

пошкоджено, а бали не нарахуються помилково.

Крім того, завдяки внутрішнім механізмам ізоляції транзакцій, PostgreSQL коректно обробляє паралельні запити від різних клієнтів без блокування бази даних. А вбудоване журналювання операцій гарантує, що навіть у випадку раптового збою сервера чи перебоїв з живленням, ігровий стан збережеться. Загалом, використання строгої табличної схеми із зовнішніми ключами дозволило делегувати частину перевірок на рівень самої бази даних, що значно розвантажило серверну логіку застосунку.

Інформаційну або сутнісно-зв'язкову модель (Entity-Relationship Model) розробленого програмного забезпечення концептуально представлено двома основними сутностями: users (Користувачі) та tasks (Завдання). Сутність users відповідає за збереження базових автентифікаційних даних, а також динамічного ігрового стану гравця, зокрема кількості накопиченого досвіду (XP) та поточного рівня. Натомість сутність tasks містить безпосередні атрибути завдань, такі як текстовий опис, логічні прапорці важливості та терміновості, а також розрахований індекс квадранту Ейзенхауера.

Між цими сутностями встановлено класичний реляційний зв'язок типу «один-до-багатьох» (One-to-Many), що логічно ізолює робочий простір кожного клієнта. Даний зв'язок реалізується на рівні бази даних через механізм зовнішнього ключа (Foreign Key), де кожне завдання жорстко прив'язане до унікального ідентифікатора свого власника. Окрім того, для підтримки абсолютної цілісності даних налаштовано каскадне видалення (ON DELETE CASCADE). Це гарантує автоматичне очищення всіх пов'язаних завдань у випадку видалення профілю користувача з системи, запобігаючи виникненню «осиротілих» записів (orphaned records). Така сувора реляційна структура виступає надійним фундаментом для стабільного функціонування інтелектуального вебзастосунку, забезпечуючи безперебійну роботу алгоритмів обчислення гейміфікованої нагороди.

Для забезпечення цілісності (Integrity) та відмовостійкості даних на рівні реляційної СУБД, логічні сутності було фізично реалізовано у вигляді таблиць із

жорстким контролем типів (Type Constraints). Такий підхід до проєктування дозволяє гарантувати абсолютну консистентність інформації навіть за умов паралельних транзакцій від різних мікросервісів. Вибір PostgreSQL як ключового сховища зумовлений її суворою відповідністю стандартам ACID та потужною підтримкою декларативних обмежень безпосередньо на рівні рушія. Завдяки цьому значна частина валідаційних перевірок делегується самій базі даних, що суттєво розвантажує серверну частину застосунку та мінімізує ризики виникнення аномалій під час збереження інформації. У таблицях 2.1 та 2.2 наведено детальну технічну специфікацію структури бази даних.

Таблиця 2.1

Структура реляційної таблиці «users» (Профіль користувача)

Назва поля	Тип даних PostgreSQL	Обмеження (Constraints)	Опис та призначення атрибута
id	SERIAL (INT4)	PRIMARY KEY	Унікальний ідентифікатор користувача. Генерується системою автоматично (Auto-increment).
username	VARCHAR(255)	NOT NULL, UNIQUE	Логін користувача. Обмеження UNIQUE на рівні рушія БД унеможливлює створення дублікатів.
password_hash	VARCHAR(255)	NOT NULL	Криптографічний хеш пароля, згенерований алгоритмом bcrypt (Salt rounds: 10).

Продовження таблиці 2.1

Структура реляційної таблиці «users» (Профіль користувача)

Назва поля	Тип даних PostgreSQL	Обмеження (Constraints)	Опис та призначення атрибута
total_xp	INTEGER	NOT NULL, DEFAULT 0	Динамічний лічильник ігрових балів. Змінюється транзакційно при виконанні завдань.
level	INTEGER	NOT NULL, DEFAULT 1	Поточний ігровий рівень гравця (розраховується алгоритмічно на базі total_xp).
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Мітка часу створення облікового запису.

Як видно з таблиці 2.1, сутність користувача зберігає базові автентифікаційні дані та загальний ігровий прогрес (бали та рівень). Проте для реалізації персоналізованого робочого простору кожен обліковий запис має бути логічно пов'язаний зі списком його власних завдань. Цей зв'язок реалізується за класичним відношенням «один-до-багатьох» (1:N). Відповідно, детальна технічна структура для збереження самих завдань, їхніх пріоритетів та поточного статусу наведена у додатку А (тут вставте вашу літеру додатка), де атрибут user_id виступає зовнішнім ключем (Foreign Key) для встановлення цієї реляційної залежності. Така сувора ізоляція записів на рівні СУБД гарантує конфіденційність, унеможливлюючи витік даних або несанкціонований доступ до чужих списків. Додатково, ця прив'язка дозволяє внутрішнім алгоритмам застосунку миттєво

агрегувати зароблені бали (XP) з усіх виконаних завдань для точного перерахунку рівня конкретного гравця.

На рисунку 2.2 подано фізичну модель взаємозв’язків типу сутність-зв’язок (Entity-Relationship Diagram) для розробленої системи.

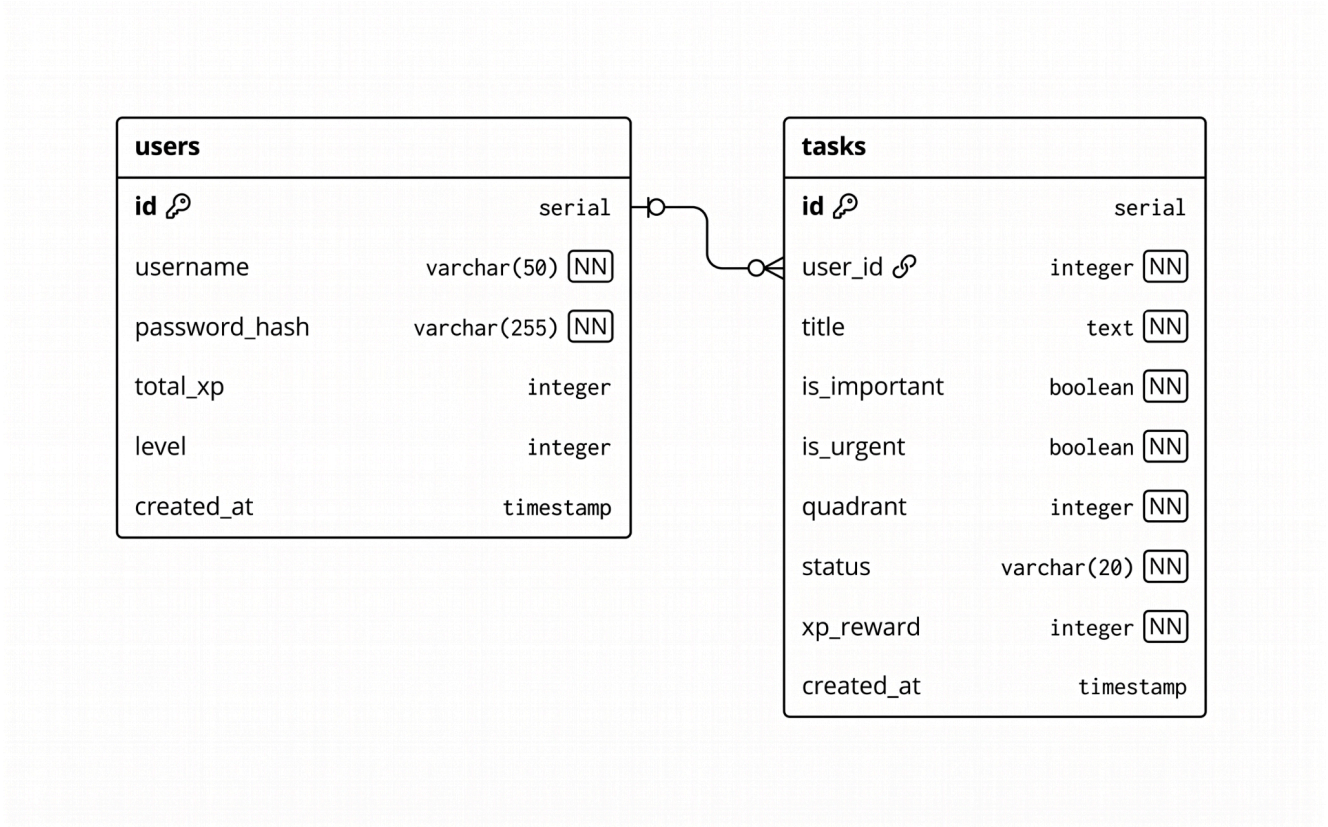


Рис. 2.2. ER-діаграма реляційної бази даних PostgreSQL

Така структура бази даних унеможливлює модифікацію ігрового балансу з боку клієнта, оскільки ціна кожного завдання (xp_reward) жорстко розраховується та фіксується на рівні серверної логіки одразу при його створенні. Відповідно, будь-які спроби зловмисника або недобросовісного користувача маніпулювати HTTP-запитами для штучного завищення власного досвіду будуть автоматично відхилені системою. Це гарантує абсолютну цілісність математичної моделі гейміфікації та зберігає ефективність психологічної механіки застосунку.

2.4. Проектування UI/UX інтерфейсу користувача.

Для застосунків, які націлені на боротьбу з прокрастинацією, проєктування інтерфейсу користувача (User Interface, UI) та досвіду взаємодії (User Experience, UX) має критичне значення. Інтерфейс відіграє роль не лише засобу введення інформації, але й психологічного фактора, який безпосередньо впливає на рівень когнітивного навантаження. За даними досліджень у сфері поведінкової психології та взаємодії людини з комп'ютером (HCI), надмірна візуальна складність інтерфейсу формує "візуальний шум" (Extraneous cognitive load), який змушує мозок витрачати ресурси на обробку графіки замість вирішення самої задачі [21]. Це прямо суперечить головній меті даного застосунку – боротьбі з прокрастинацією.

Для мінімізації когнітивного навантаження було обрано дизайн-парадигму Glassmorphism (Гласморфізм) [18]. Цей стиль характеризується використанням напівпрозорих елементів із ефектом матового скла (background blur), що створює ілюзію багатошаровості та просторової глибини інтерфейсу без додавання зайвих розділових ліній чи жорстких кордонів. Застосування цієї концепції в розроблюваному продукті дозволяє відокремити блоки завдань від динамічного неонового фону, зберігаючи при цьому загальну візуальну «легкість» системи.

Психологічний та когнітивний вплив обраного підходу зумовлений декількома чинниками. По-перше, збереження просторового контексту через розмиття заднього плану активізує механізми інтуїтивного сприйняття ієрархії інтерфейсу, де напівпрозора картка завдання ідентифікується зоровим аналізатором як об'єкт переднього плану, що потребує пріоритетної уваги. По-друге, синергетичний ефект поєднання напівпрозорих панелей із темною кольоровою схемою суттєво знижує зорову втому за рахунок мінімізації емісії короткохвильового синього світла, що є критичним для запобігання астенопії під час тривалих робочих сесій. По-третє, інтеграція розмитих контейнерів із неоновими індикаторами ігрових метрик, елементами прогресу та анімаціями

досягнення нових рівнів забезпечує реалізацію естетики гейміфікації. Це активізує позитивний асоціативний зв'язок із сучасними цифровими середовищами та стимулює внутрішню мотивацію користувача за рахунок оптимізації доречного (релевантного) когнітивного навантаження.

Отже, застосування стилю Glassmorphism є не просто візуальною "прикрасою", а науково обґрунтованим рішенням (UX/UI), яке сприяє швидшій адаптації користувача до системи Матриці Ейзенхауера.

Системний аналіз фундаментальних аспектів проєктування інтерфейсу дозволяє простежити взаємозв'язок між колористичною архітектурою, хроматичною сегрегацією та інтерактивною динамікою системи. Генеральною графічною основою розробки визначено темну палітру (Dark Mode) низької інтенсивності на базі кольору slate-900, що є прагматично обґрунтованим для довготривалої роботи фахівців галузі інформаційних технологій через мінімізацію яскравісного контрасту та запобігання деградації зорових функцій. Для фокусування уваги на гейміфікованих метриках, як-от рівнева градація та шкала досвіду (XP), застосовано висококонтрастні неонові градієнти в діапазоні від indigo-500 до blue-500.З метою забезпечення миттєвої оптичної ідентифікації та зниження латентності сприйняття інформації, для кожного з чотирьох секторів матриці Ейзенхауера реалізовано диференційоване колірне кодування за принципом температури та психологічної важливості. Зокрема, перший квадрант («Важливо та Терміново») маркується рубіново-червоним спектром, який виконує функцію превентивного сигналу високої критичності дедлайну, тоді як другий квадрант («Важливо, але Нетерміново») використовує смарагдово-синій відтінок, що асоціативно корелює зі стратегічним плануванням та когнітивним спокоєм. Для третього квадранта («Неважливо, але Терміново») обрано бурштиново-помаранчевий тон для позначення рутинних операцій, а четвертий квадрант («Неважливо та Нетерміново») представлений нейтральним сірим кольором, який стимулює депріоритизацію або елімінацію завдань.

Враховуючи інтегровані механізми гейміфікації, критично важливим компонентом стратегії UX є забезпечення динамічного зворотного зв'язку (UI feedback) через мікроінтеракції. Процедура верифікації виконання завдання ініціює миттєву тригерну прогресію індикатора стану у верхній панелі системи, тоді як транзиція на новий ігровий рівень супроводжується повномасштабною графічною анімацією (Confetti). Подібний інтерактивний відгук системи виступає в ролі екзогенного стимулятора позитивного підкріплення (дофамінового контуру), що безпосередньо підвищує рівень залученості та тривалого утримання (retention) користувача в робочому середовищі.овані на те, щоб користувач з першого погляду бачив загальний фокус, не застосовуючи додаткових зусиль на читання текстових оцінок пріоритетів.

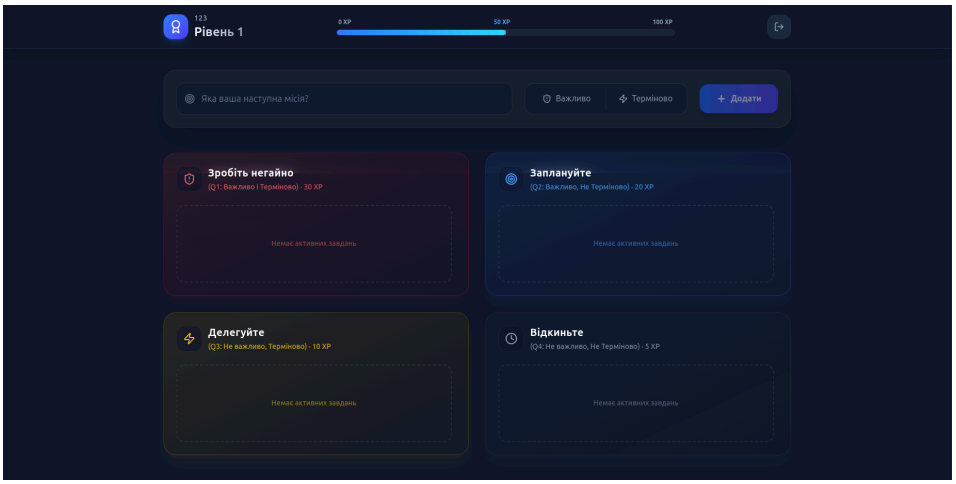


Рис. 2.3. Користувацький інтерфейс матриці Ейзенхауера у стилі Glassmorphism

У другому розділі даної кваліфікаційної роботи було проведено фундаментальний етап системного аналізу та повністю спроєктовано загальну клієнт-серверну архітектуру майбутнього інтелектуального web-застосунку. Зважаючи на жорсткі вимоги щодо масштабованості та відмовостійкості сучасного програмного забезпечення, було прийнято стратегічне рішення відійти від традиційної монолітної структури на користь концепції незалежних мікросервісів. Такий підхід дозволив логічно та фізично розділити систему на ізольовані компоненти за допомогою індустріального інструментарію контейнеризації Docker. Окрім забезпечення зручності розгортання, особливу

увагу на етапі проєктування інфраструктури було приділено питанням мережевої безпеки: теоретичне обґрунтування та впровадження концепції Cloudflare Zero Trust дозволило сформувати захищений криптографічний тунель, який унеможливлює несанкціонований прямий доступ до серверних портів ззовні та ефективно захищає систему від потенційних DDoS-атак.

На основі детально сформульованих функціональних та нефункціональних вимог було розроблено візуальну UML-діаграму сценаріїв використання (Use Case Diagram). Дана модель вичерпно формалізує шляхи взаємодії кінцевого користувача з інформаційною системою та наочно демонструє, як саме механіки гейміфікації інтегруються у рутинний процес тайм-менеджменту, виступаючи ключовим мотиваційним патерном поведінки. Паралельно з логікою застосунку, на рівні роботи з персистентними даними було спроєктовано стійку реляційну модель для об'єктно-реляційної СУБД PostgreSQL. Суворе дотримання правил нормалізації дозволило створити оптимізовану ієрархію зв'язків сутностей «Користувач – Завдання», яка виключає дублювання інформації та на рівні рушія бази даних гарантує транзакційну цілісність складної ігрової математики (процесів нарахування XP та динамічного розрахунку ігрових рівнів).

Для проєктування UI/UX обрано дизайн-систему Glassmorphism у поєднанні з темною темою (Dark Mode), що запобігає зоровому перевантаженню користувача під час тривалої роботи. Ефекти розмиття фону та акцентні неонові кольори допомагають миттєво ідентифікувати пріоритетність завдань за квадрантами Ейзенхауера, що разом із розробленою архітектурою створює надійну базу для програмної реалізації проєкту.

3. ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ

3.1. Реалізація серверної частини та RESTful API

Серверна частина (Backend) інтелектуального застосунку продуктивності відповідає за забезпечення бізнес-логіки системи: автентифікацію користувачів, безпечний обмін даними з СУБД PostgreSQL та безпосередній розрахунок балів гейміфікації (XP). Для забезпечення універсальності взаємодії між клієнтом і сервером, архітектуру було побудовано на базі архітектурного стилю REST (Representational State Transfer), концепція якого була вперше формалізована Роєм Філдінгом [12].

Використання RESTful API забезпечує так звану "статтєнезалежність" (Statelessness): кожен мережевий запит від клієнта до сервера містить усю необхідну інформацію для його обробки. Як середовище виконання серверної логіки було обрано Node.js, а в якості каркаса (фреймворку) розробки – мікрофреймворк Express.js. Його мінімалістичність дозволяє гнучко налаштовувати маршрутизацію (Routing) та застосовувати модулі проміжного програмного забезпечення (Middleware).

Точкою входу для серверної частини є файл server.js. У ньому налаштовуються параметри безпеки, проміжні обробники (middleware) та базові маршрути API.

Процес ініціалізації сервера відбувається у кілька етапів. Спочатку підключається модуль cors, який дозволяє обмін даними між клієнтом та сервером (оскільки в Docker-середовищі вони працюють на різних портах). Далі викликається метод express.json() для автоматичного парсингу вхідних HTTP-запитів у формат JSON.

Після базових налаштувань відбувається реєстрація маршрутів (Routing). Уся логіка розділена на два основні напрямки: маршрут /api/auth обробляє

реєстрацію користувачів, перевірку паролів та генерацію JWT-токенів, а маршрут /api/tasks відповідає за базові операції із завданнями (створення, редагування, видалення) та містить логіку нарахування балів досвіду.

На фінальному етапі ініціалізації підключається глобальний обробник помилок. Якщо під час виконання будь-якого запиту виникає збій, сервер не припиняє свою роботу (не падає), а перехоплює помилку і коректно повертає клієнту відповідь зі статус-кодом 500 (Internal Server Error).

Окремої уваги заслуговує підхід до реалізації підсистеми автентифікації користувачів, який критично впливає на безпеку персональних даних. У класичних web-застосунках найчастіше застосовуються сесії (Sessions), що зберігаються в оперативній пам'яті сервера. Проте, з огляду на мікросервісну архітектуру розробленого рішення, зберігання сесій на бекенді порушує принцип "статтєнезалежності" (Statelessness) REST API і ускладнює горизонтальне масштабування системи.

Тому було прийнято рішення імплементувати механізм токенізації на базі стандарту JSON Web Token (JWT) [22]. Цей стандарт дозволяє передавати безпечні заяви між сторонами у форматі JSON-об'єкта. Процес автентифікації у розробленій системі побудовано за наступним алгоритмом:

- при успішному вході клієнта (співпадіння хешу пароля bcrypt), сервер генерує JWT-токен за допомогою методу jwt.sign();
- до корисного навантаження (Payload) токена додаються ідентифікатор користувача (user_id), його поточний рівень та сумарна кількість ігрових балів (total_xp);
- створений об'єкт підписується секретним криптографічним ключем (JWT_SECRET) на сервері за допомогою алгоритму HMAC SHA-256 (HS256).

Такий підхід дозволяє серверу не звертатися до бази даних PostgreSQL щоразу, коли клієнт виконує захищений запит. Для верифікації прав доступу серверу достатньо виконати математичну перевірку підпису токена за допомогою

методу jwt.verify(). Якщо підпис валідний – це гарантує, що вміст Payload не був змінений клієнтом з метою штучного збільшення ("накрутки") свого ігрового досвіду.

На рисунку 3.1 подано діаграму компонентів (Component Diagram) серверної маршрутизації застосунку, що ілюструє потік проходження REST-запитів.

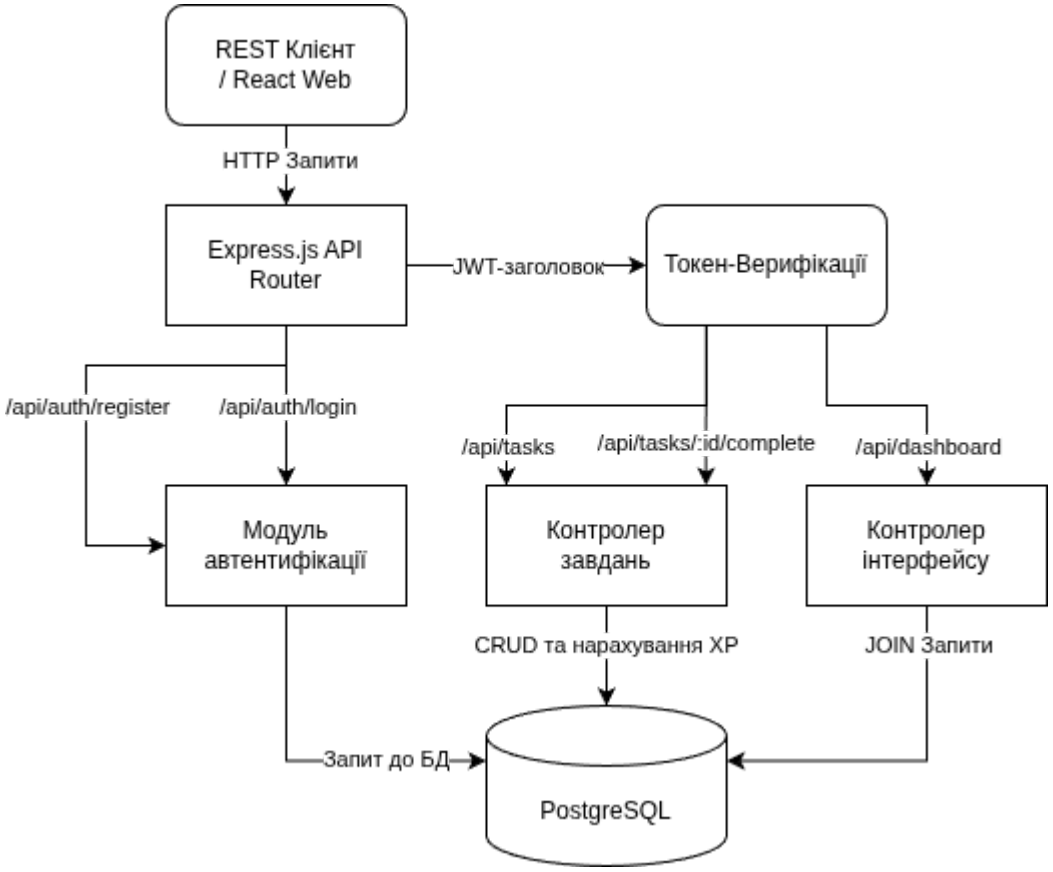


Рис. 3.1 – Діаграма компонентів API

У межах цієї архітектури приділено увагу безпеці: для запобігання атакам класу Cross-Site Scripting (XSS), у яких зловмисники можуть спробувати викрасти токен із клієнтського сховища (LocalStorage), на рівні Node.js реалізовано валідацію вхідних даних, а на клієнті – перехоплювачі бібліотеки Axios. Завдяки глобальному перехоплювачу (Axios Request Interceptor), кожен вихідний HTTP-запит автоматично інкапсулює JWT токен у заголовок Authorization: Bearer , забезпечуючи безшовну взаємодію клієнта з API [23]. Такий архітектурний підхід не лише посилює захист сесій, але й суттєво оптимізує кодову базу фронтенду. Він усуває необхідність ручного прикріплення токена до кожного окремого

мережевого запиту, що зводить до мінімуму ймовірність людської помилки та значно спрощує подальше масштабування системи. Додатково, використання глобальних перехоплювачів відповідей (Response Interceptors) дозволяє централізовано обробляти помилки автентифікації. Наприклад, у разі закінчення терміну дії токена (отримання статус-коду 401 Unauthorized), система здатна автоматично очистити скомпрометовану сесію та перенаправити користувача на сторінку логіну, гарантуючи безперервний контроль доступу.

Повний програмний код файлу ініціалізації серверної частини (server.js), який містить налаштування проміжного програмного забезпечення та логіки обробки маршрутів автентифікації, наведено в Додатку А.

Подібна модульна організація коду (винесення маршрутів у зовнішні файли директорії routes/) гарантує високу підтримуваність системи, що дає змогу швидко розширювати API новим функціоналом (наприклад, додаванням таблиці лідерів для розширення гейміфікації) без модифікації основного ядра програми.

Для забезпечення інтероперабельності системи (Interoperability) та можливості в майбутньому підключати мобільні клієнти без зміни серверного ядра, було розроблено жорстку специфікацію REST API-маршрутів. Взаємодія між клієнтом та сервером здійснюється за допомогою методів протоколу HTTP (GET, POST, PATCH), де сервер повертає статус-коди (HTTP Status Codes) та уніфіковані об'єкти у форматі JSON.

Для детальнішого та науково обґрунтованого введення в архітектуру взаємодії компонентів нижче наведено таблицю 3.1, яка містить вичерпну специфікацію ендпоінтів модуля автентифікації користувачів. Маршрути цього конкретного модуля за замовчуванням є загальнодоступними та публічними, тобто вони не вимагають обов'язкової наявності та валідації JWT-токена у заголовках запиту, оскільки призначені саме для первинного входу в систему або створення нового профілю. У специфікації чітко регламентовано підтримувані HTTP-методи, структуру вхідних і вихідних JSON-пакетів даних, а також перелік можливих статус-кодів відповідей сервера, що забезпечує стандартизовану й безпечну обробку облікових даних на етапі ініціалізації сесії.

Таблиця 3.1

Специфікація REST API-маршрутів модуля автентифікації (/api/auth)

Метод та Endpoint	Призначення	Формат тіла запиту (Request Body JSON)	Формат успішної відповіді (Response JSON)	Статус-коди
POST /register	Створення нового облікового запису	{ "username": "test", "password": "123" }	{ "message": "User registered successfully", "token": "eyJhbG..." }	201 Created, 400 Bad Request
POST /login	Авторизація існуючого користувача	{ "username": "test", "password": "123" }	{ "token": "eyJhbG...", "user": { "id": 1, "level": 2 } }	200 OK, 401 Unauthorized

Усі запити до модуля управління завданнями (таблиця 3.2) вимагають наявності валідного JWT-токена у заголовку HTTP-запиту Authorization: Bearer <token>. Завдяки цьому сервер миттєво ідентифікує, чиї саме задачі необхідно обробити, не вимагаючи передачі ID користувача безпосередньо у тілі запиту, що ефективно запобігає підміні даних та атакам типу Insecure Direct Object Reference (IDOR). Додатково, такий бездержавний підхід (Stateless) позбавляє сервер необхідності зберігати сесії в оперативній пам'яті, що суттєво підвищує загальну продуктивність та масштабованість бекенду. Повна специфікація REST API-маршрутів цього модуля, яка регламентує процеси отримання, створення та модифікації завдань, структуру JSON-пакетів та відповідні HTTP-статуси, детально систематизована у таблиці 3.2.

Таблиця 3.2

Специфікація REST API-маршрутів управління завданнями (/api/tasks)

Метод та Endpoint	Призначення	Формат тіла запиту (Request Body JSON)	Формат успішної відповіді (Response JSON)	Статус-коди
GET /	Отримання всіх активних завдань поточного користувача	Відсутнє (GET-запит)	Масив об'єктів JSON: [{ "id": 1, "title": "...", "quadrant": 1 }]	200 OK, 401 Unauthorized
POST /	Створення нового завдання з розрахунком квадранту	{ "title": "Диплом", "is_important": true, "is_urgent": true }	{ "message": "Task created", "task_id": 15, "quadrant": 1, "xp": 30 }	201 Created, 500 Server Error
PATCH /:id/complete	Виконання завдання та нарахування ігрового досвіду (XP)	Відсутнє (ID передається як параметр URL)	{ "message": "Task completed", "xp_earned": 30, "new_total_xp": 120 }	200 OK, 404 Not Found

3.2. Реалізація бізнес-логіки: матриця Ейзенхауера та система нарахування досвіду

Інтеграція ігрових елементів у неігровий контекст (Гейміфікація) вимагає математично збалансованого ядра [13]. Якщо користувач отримуватиме однакову винагороду за дрібні (рутинні) та складні завдання, застосунок втратить свою стимулюючу ефективність. З метою вирішення цієї проблеми, бізнес-логіка серверної частини об'єднує класифікацію матриці Ейзенхауера безпосередньо з механізмом генерації балів досвіду (Experience Points, XP).

Процес алгоритмізації матриці та визначення винагороди працює наступним чином: при створенні нового завдання через REST API-запит до маршруту POST /api/tasks, сервер отримує два основні параметри: рівень важливості (is_important) та рівень терміновості (is_urgent). На базі цих параметрів викликається внутрішня допоміжна функція calculateQuadrantAndXP, яка виконує булеву маршрутизацію. Формалізовано логіка нарахування виглядає так:

- якщо завдання Важливе та Термінове (Квадрант 1), воно вважається критичним. Винагорода: 30 XP;
- якщо завдання Важливе, але Не термінове (Квадрант 2), воно є стратегічним. Винагорода: 20 XP;
- якщо завдання Не важливе, але Термінове (Квадрант 3), це делегована рутина. Винагорода: 10 XP;
- якщо завдання Не важливе і Не термінове (Квадрант 4), воно має найнижчий пріоритет. Винагорода: 5 XP.

Завдяки цьому алгоритму користувач підсвідомо стимулюється виконувати найскладніші задачі з першого квадранту задля швидшого "прокачування" свого віртуального рівня.

Наступним етапом є транзакційна зміна ігрового стану користувача.

Отримання винагороди ініціюється викликом методу оновлення статусу завдання (PATCH /api/tasks/:id/complete). Після успішного підтвердження автентичності користувача (через верифікацію JWT токена), сервер виконує послідовну зміну стану в базі даних PostgreSQL.

Особливістю програмної реалізації є оптимізований підхід до перерахунку рівнів. Замість зберігання статичної таблиці рівнів, використовується динамічна математична модель: Level = FLOOR(Total_XP / 100) + 1 Ця формула гарантує лінійне зростання рівня кожні здобуті 100 балів досвіду. На рівні бази даних це оновлення виконується єдиним атомарним SQL-запитом, що запобігає стану "гонки даних" (Race condition).

На рисунку 3.2 наведено блок-схему розробленого алгоритму, що демонструє процес оновлення статусу завдання та транзакційний розрахунок нового віртуального рівня користувача на стороні серверного модуля та СУБД PostgreSQL.

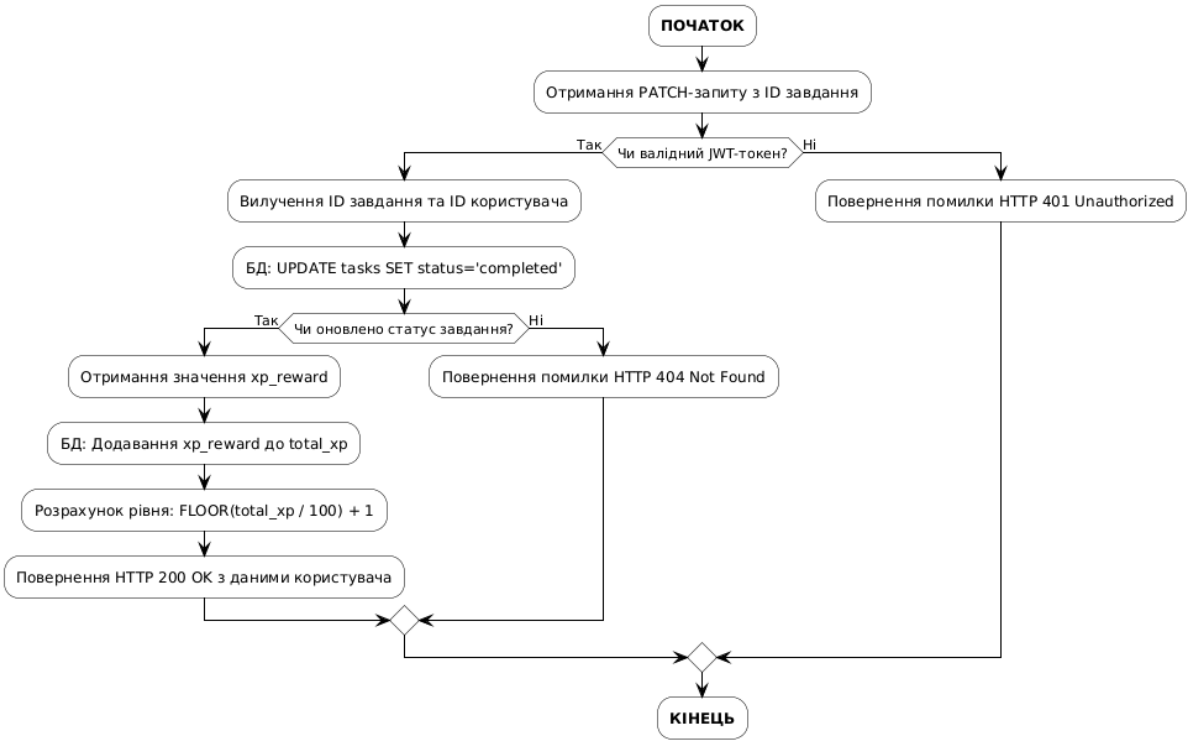


Рис. 3.2. Блок-схема алгоритму транзакційного нарахування балів досвіду та оновлення рівня

Реалізований алгоритм гарантує консистентність даних: бали гарантовано нараховуються лише у тому випадку, якщо завдання було в статусі 'active' (активне), що повністю виключає можливість багаторазового "закриття" однієї задачі клієнтом задля нечесного отримання внутрішньоігрового досвіду.

Більш детально життєвий цикл завдання та зв'язок із транзакціями бази даних проілюстровано на діаграмі станів (State Machine Diagram) на рисунку 3.2. Окрім того, для забезпечення безпеки доступу до бази даних спроєктовано захищений механізм автентифікації, покроковий сценарій якого відображено на діаграмі послідовності (Sequence Diagram) на рисунку 3.3.

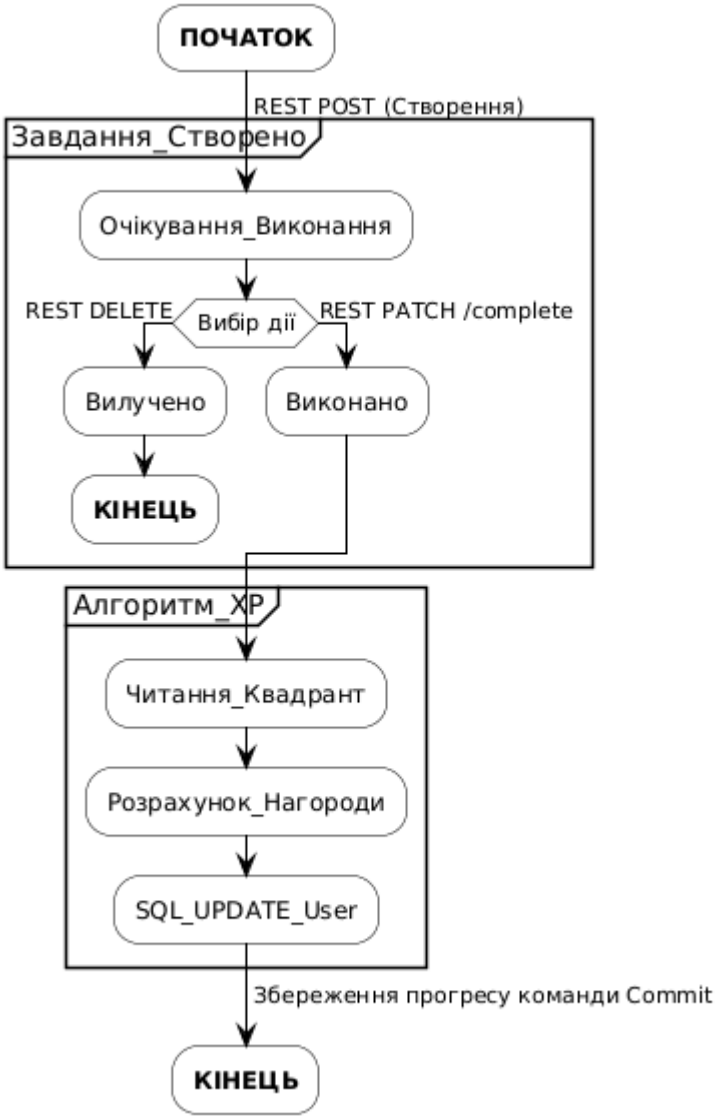


Рис. 3.3. Діаграми станів завдання

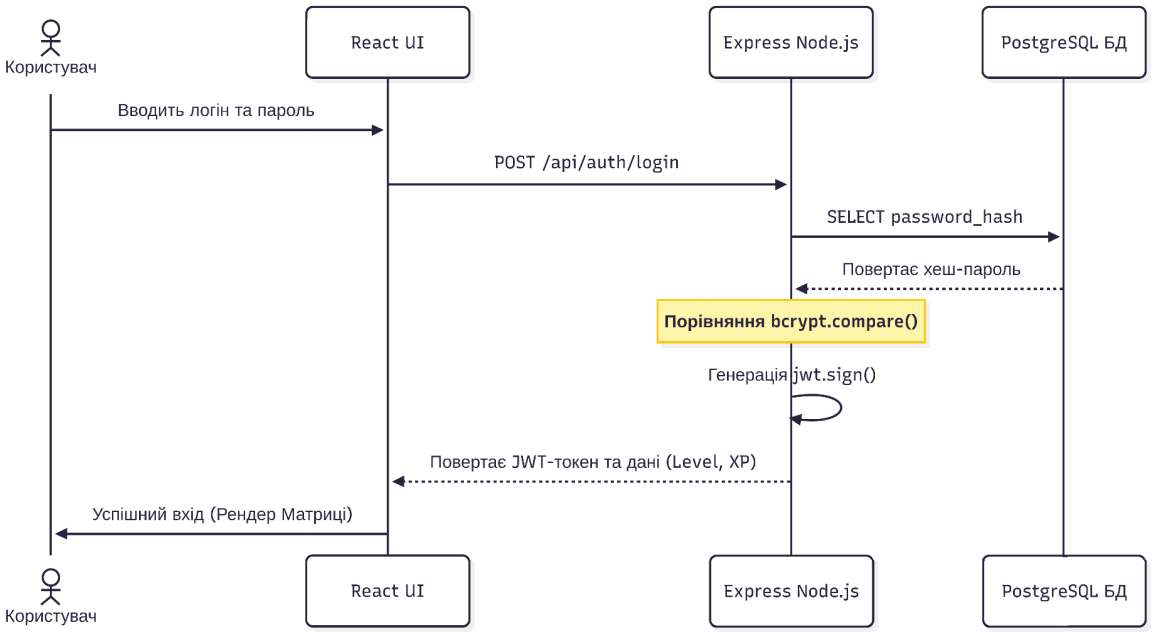


Рис. 3.4. Діаграма послідовності процесу безпечної автентифікації

3.3. Розробка клієнтської частини застосунку

Клієнтська частина web-застосунку розроблена за принципом Single Page Application (SPA) із використанням бібліотеки React [18]. Даний підхід дозволяє перемальовувати лише ті компоненти інтерфейсу, дані в яких змінилися, без необхідності повного повторного завантаження HTML-документа з сервера. Це є критичною вимогою для гейміфікованих платформ, де реакція інтерфейсу на дії користувача має бути миттєвою.

Щодо управління станом та життєвим циклом, то основним архітектурним патерном на фронтенді є використання React Hooks [18]. Глобальний стан застосунку (масив поточних завдань, рівень користувача та його ігрові бали) зберігається за допомогою хука useState. Взаємодія з бекендом (отримання даних при завантаженні сторінки або оновлення списку після авторизації) керується хуком useEffect, який ініціює асинхронні HTTP-запити до REST API за допомогою технології Fetch API із передачею збереженого авторизаційного JWT токена в

заголовках Authorization.

Одним із головних технічних викликів під час розробки фронтенду стала оптимізація рендерингу Матриці Ейзенхауера, особливо при роботі з великою кількістю завдань. Хоча віртуальний DOM у React і забезпечує високу швидкодію, неправильна робота зі станом може призводити до надлишкового перемалювання всього дерева компонентів (Unnecessary Re-renders). Щоб уникнути цього, завантаження даних було оптимізовано за допомогою хука useEffect, який виконує лише один асинхронний запит при першому відкритті сторінки та кешує всі задачі в єдиний масив tasks.

Крім того, для оптимізації оновлення інтерфейсу під час переміщення чи видалення завдань, кожній картці (компоненту <TaskCard>) призначається унікальний ідентифікатор key={task.id}. Це дозволяє React ефективно застосовувати внутрішній алгоритм порівняння (Reconciliation) [19]. Таким чином, коли користувач відмічає завдання як виконане, система точково видаляє з фізичного DOM-дерева лише одну конкретну картку, залишаючи всі інші елементи списку недоторканими. Такий підхід суттєво економить ресурси браузера і робить інтерфейс максимально чуйним.

Безпосередньо для візуалізації самої Матриці Ейзенхауера та побудови сітки з чотирьох квадрантів було використано парадигму фреймворку Tailwind CSS [21]. Структуру екрану реалізовано на базі технології CSS Grid Layout. Це гарантує стовідсоткову адаптивність (Responsive Web Design): на десктопних моніторах браузера квадранти формують класичну квадратну матрицю розмірністю 2x2, а на мобільних телефонах автоматично перешиковуються в єдину вертикальну колонку для забезпечення зручності відображення та скролінгу.

Програмна маршрутизація завдань у відповідні квадранти здійснюється функціональним методом масивів Array.prototype.filter(). Замість того, щоб створювати чотири окремі масиви у пам'яті клієнта (для кожного з квадрантів), метод у режимі реального часу ітерує базовий масив tasks, перевіряючи властивість quadrant (яка динамічно розрахована бекендом від 1 до 4) і статус задачі (status === 'active'). Для кожного квадранту в інтерфейсі визначено свій

компонент-контейнер, який приймає відфільтровані завдання і рендерить їх у вигляді карток (TaskCard).

Для організації візуальної частини було використано компонентний підхід. На рисунку 3.4 наведено ієрархічне дерево React-компонентів клієнтської частини, що відповідає за візуалізацію матриці та обробку візуальних ефектів гейміфікації.

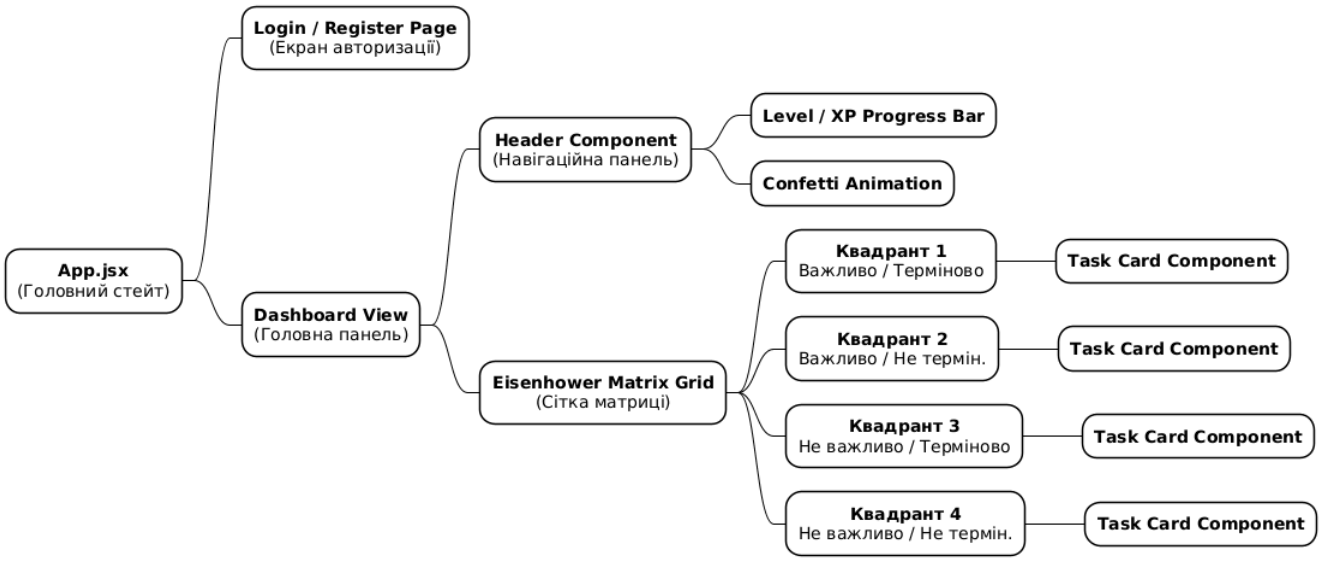


Рис. 3.5. Ієрархічне дерево компонентів користувацького інтерфейсу

Повний програмний код головного клієнтського компонента (App.jsx), який містить логіку реактивного оновлення стану інтерфейсу та застосування стилістики Glassmorphism, наведено в Додатку Б.

3. Реалізація ефекту Glassmorphism та динамічної гейміфікації. Згідно з вимогами, визначеними у підрозділі 2.4, стилізація інтерфейсу виконана у стилі Glassmorphism. Для створення цього ефекту застосовано каскадні таблиці стилів, що забезпечують напівпрозоре фонове заповнення об'єктів та використання властивості розмиття заднього плану (backdrop-filter). Це забезпечує візуальну імітацію матового скла без втрати продуктивності візуалізації інтерфейсу.

Для візуалізації ігрового прогресу в інтерфейсі розроблено індикатор рівня

(Level Progress Bar). Оскільки для переходу на новий рівень користувачу потрібно набрати 100 балів досвіду, ширина індикатора у відсотках розраховується як залишок від ділення за формулою:

$$W = XP_{total} (mod\ 100) \quad (3.1)$$

де W – відсоток заповненості шкали, а XP_{total} – загальна кількість здобутих балів. Щойно користувач долає межу у 100 балів, його рівень підвищується. Коли клієнт отримує оновлені дані від сервера, в інтерфейсі автоматично запускається анімація за допомогою бібліотеки canvas-confetti [3], яка виступає візуальною нагородою за виконану роботу.

Окремою складовою оптимізації стало те, що координати запуску цієї анімації розраховуються "на льоту" на основі методу BoundingClientRect тієї кнопки, на яку натиснув користувач. Це створює фізичний ефект розкидання частинок прямо від курсору миші, максимізуючи викид дофаміну без блокування головного потоку браузера і створюючи інтенсивний ефект ігрового "level-up".

Базуючись на попередньо спроєктованій архітектурі, було успішно створено високопродуктивний мікросервісний бекенд на платформі Node.js. Використання даного середовища дозволило реалізувати неблокуючу подієво-орієнтовану модель обробки запитів, що є критично важливим для забезпечення безперебійної роботи RESTful API під навантаженням. Саме на цьому програмному рівні було імплементовано складні математичні алгоритми автоматичної класифікації завдань згідно з принципами матриці Ейзенхауера. Особлива інженерна увага була приділена гарантуванню цілісності розрахунків ігрових метрик: процеси нарахування гейміфікаційних нагород (XP) та підвищення рівня користувача реалізовані через механізм строгих атомарних транзакцій у системі управління реляційними базами даних PostgreSQL, що унеможливлює втрату або викривлення даних у разі виникнення мережевих збоїв.

Паралельно з розробкою алгоритмічного ядра, у системі було впроваджено передові індустріальні стандарти кібербезпеки. Побудовано повністю бездержавну (Stateless) архітектуру автентифікації на базі криптографічно підписаних токенів

JSON Web Token (JWT). Такий підхід не лише забезпечив захищений та ізольований обмін даними між клієнтом та сервером, але й суттєво знизив апаратне навантаження на серверну пам'ять. Крім того, на рівні серверної конфігурації було реалізовано надійні політики CORS, що обмежують ризики несанкціонованого міжсайтового доступу до приватних API-маршрутів.

З боку користувацької взаємодії було спроєктовано та розроблено сучасний реактивний клієнтський інтерфейс у форматі односторінкового застосунку (Single Page Application, SPA) з використанням екосистеми React. Завдяки концепції алгоритмічного порівняння Virtual DOM, система здатна миттєво перемальовувати виключно ті компоненти, стан яких змінився (наприклад, переміщення картки завдання або активація анімації Level-UP), уникаючи "важких" перезавантажень сторінки. Візуальна складова інтерфейсу базується на принципах утилітарного фреймворку Tailwind CSS, за допомогою якого було інтегровано адаптивні Glassmorphism-компоненти. Цей стилістичний вибір дозволив створити не лише гнучкий кросплатформний дизайн, але й суттєво знизити когнітивне навантаження на зір користувача завдяки використанню темної теми, м'якому розмиттю фону та напівпрозорості елементів.

Гармонійне поєднання цих інноваційних технологій Backend та Frontend розробки забезпечило створення винятково стабільного, безпечного та оптимізованого програмного продукту, який повністю підготовлений до етапу комплексного QA-тестування та подальшої безшовної інтеграції в експлуатаційне (Production) середовище.

4. ТЕСТУВАННЯ ПРОГРАМИ

4.1. Обґрунтування використання обраних видів тестування програми

Забезпечення якості програмного продукту (Quality Assurance) є невід'ємним етапом розробки, який гарантує відповідність реалізованого функціоналу початковим вимогам системи інтелектуальної гейміфікації. Згідно з методологією Міжнародної ради з тестування програмного забезпечення (ISTQB), стратегія тестування повинна покривати як архітектурні рівні, так і безпосередні користувацькі сценарії [15].

З огляду на мікросервісну (клієнт-серверну) архітектуру розробленого застосунку та наявність математичних моделей нарахування досвіду, було обрано три фундаментальні види тестування: функціональне тестування, інтеграційне тестування та тестування безпеки (Security Testing).

Оскільки застосунок складається з трьох незалежних вузлів (React-клієнт, Node.js API та PostgreSQL), критично важливим є перевірка їхньої міжкомпонентної взаємодії. Інтеграційне тестування застосовувалося для верифікації проходження масивів даних (JSON) через маршрутизацію. Основні об'єкти перевірки:

- коректність перехоплення запитів (/api/*) проксі-сервером Nginx та їх безшовна передача до Backend-контейнера;
- валідність транзакцій бази даних: перевірка того, що при оновленні статусу завдання (PATCH /api/tasks/:id/complete) база даних гарантовано повертає оновлений запис сутності users із перерахованим рівнем.

Наступним етапом стало функціональне тестування, сфокусоване на перевірці бізнес-логіки застосунку. Перевірка здійснювалася методом «чорної скриньки» (Black-box testing) шляхом симуляції реальних дій користувача в

інтерфейсі. Зокрема, тестувалася коректність алгоритмів сортування завдань: наприклад, створення задачі з параметрами is_important=true та is_urgent=false має гарантувати її миттєву появу виключно в другому квадранті матриці. Окрему увагу було приділено гейміфікаційному балансу та лімітам формули розрахунку досвіду. Було верифіковано, що при додаванні 30 XP до поточного балансу в 90 XP система коректно переводить гравця на наступний рівень, встановлюючи залишок у 20 XP на новій шкалі прогресу. Паралельно перевірялася стабільність візуальних ефектів, щоб анімація нагороди (Confetti) спрацьовувала виключно в момент підвищення рівня, уникаючи помилкових запусків під час звичайного перезавантаження сторінки.

Оскільки застосунок спроєктовано для розгортання в публічній мережі Інтернет, окремим напрямком стало тестування безпеки та ізоляції даних (Security Testing). У межах цього етапу перевірялася надійність маршрутів API: будь-яка спроба зчитати список завдань або ініціювати нарахування балів без валідного JWT-токена (або з простроченим) гарантовано блокується сервером із поверненням статусу 401 Unauthorized. Також було проведено аудит бази даних PostgreSQL, який підтвердив, що паролі користувачів зберігаються виключно у вигляді незворотних криптографічних хеш-сум, згенерованих алгоритмом bcrypt. Це надійно захищає облікові записи від зворотного інжинірингу навіть у гіпотетичному випадку компрометації серверної інфраструктури.

Використання обраного комплексу тестування дозволило виявити та своєчасно усунути приховані логічні дефекти ще на ранніх етапах розробки. Це забезпечило високий рівень стабільності та загальної відмовостійкості програмного забезпечення, повністю підготувавши його до безпечного розгортання в ізольованому середовищі Docker та гарантуючи безперебійний користувацький досвід в умовах реальної експлуатації.

4.2. Розробка тест-кейсів для перевірки основного функціоналу та інтерфейсу користувача

З метою системної верифікації розробленого програмного забезпечення було застосовано методологію тестування на основі сценаріїв (Test Cases). Тест-кейс являє собою формалізований набір кроків, попередніх умов та очікуваних результатів, призначених для перевірки певної функції програми.

Згідно з міжнародною методологією функціонального тестування (Black-box testing) від ISTQB [11], професійна розробка тест-кейсів вимагає не лише формалізації вхідних даних, але й визначення чітких умов середовища (Pre-conditions) та гарантованих станів після завершення тесту (Post-conditions). Особливу увагу в процесі верифікації було приділено модулю авторизації, алгоритмізації Матриці Ейзенхауера та безвідмовності динамічного розрахунку балів гейміфікації (XP).

Для глибокого аналізу критичних вузлів системи було розроблено розширені специфікації для найбільш складних архітектурних процесів. Нижче наведено детальний опис двох найважливіших сценаріїв.

Тест-кейс TC-03: Валідація алгоритму маршрутизації Ейзенхауера (Важливо + Терміново) Мета тесту: Перевірити, чи система коректно аналізує булеві прапорці та зараховує задачу до найпріоритетнішого Квадранту 1. Перед-умови (Pre-conditions): Користувач успішно пройшов процес авторизації (згідно з TC-01), отримав валідний JWT токен та перебуває на головній панелі.

Рівень XP користувача становить 0. Кроки виконання (Test Steps):

- ініціювати процес створення нового завдання через графічний інтерфейс;
- ввести у поле назви тестовий рядок "Здати дипломний проєкт";
- активувати чекбокси (Toggles) is_important = true та is_urgent = true;
- натиснути кнопку "Додати завдання", відправивши POST-запит до серверного маршруту /api/tasks.

Очікуваний результат:

- сервер обробляє запит і розраховує математичні змінні: quadrant = 1, xp_reward = 30. Запис транзакційно зберігається у реляційну таблицю tasks;
- frontend, отримавши відповідь (HTTP 201 Created), миттєво рендерить картку у червоному блоці матриці.

Після-умови (Post-conditions): Завдання присутнє у глобальному стані застосунку та прив'язане виключно до ідентифікатора поточного користувача.

Тест-кейс TC-05: Валідація математичної моделі гейміфікації (Level-UP) Мета тесту: Підтвердити безвідмовність серверної логіки нарахування ігрового досвіду (XP) та візуального тригера при зміні рівня користувача. Перед-умови (Pre-conditions): Користувач має 1 активне завдання у Квадранті 1 (вартістю 30 XP). Його поточний баланс дорівнює 90 XP (до порогу залишається 10 XP). Кроки виконання (Test Steps):

- користувач натискає на кнопку виконання завдання (іконка "CheckCircle2");
- ініціюється HTTP PATCH запит до /api/tasks/:id/complete.

Очікуваний результат:

- СУБД успішно виконує атомарну транзакцію: сума total_xp досягає 120. Внутрішня SQL-функція FLOOR((120) / 100) + 1 перераховує level = 2;
- інтерфейс React розраховує ширину прогрес-бару як Math.min((120 % 100), 100), що дорівнює 20%.

Запускається візуальна бібліотека canvas-confetti (анімація святкування), оскільки хук useEffect зафіксував зміну змінної user.level. Баланс гравця успішно перенесено та надійно зафіксовано в базі даних, а виконане завдання отримало незворотний статус completed. Завдяки цьому воно миттєво зникає з активної сітки матриці Ейзенхауера, більше не бере участі в процесах візуального рендерингу та не навантажує DOM-дерево клієнта, залишаючись доступним виключно у прихованій архівній історії профілю. Таке поєднання оптимізації інтерфейсу та миттєвого візуального зворотного зв'язку успішно заохочує користувача до подальшої продуктивності.

Таблиця 4.1

Ключові тест-кейси, розроблені для функціонального та інтеграційного тестування системи.

ID	Об'єкт тестування	Кроки для відтворення (Steps)	Очікуваний результат (Expected)	Статус
TC-01	Реєстрація нового користувача	1. Відкрити сторінку авторизації. 2. Ввести унікальний "Логін" і "Пароль". 3. Натиснути "Зареєструватися".	Сервер створює нового користувача в БД, хешує пароль. Клієнт отримує токен і перенаправляється на пусту Матрицю з 0 XP.	Успішно (Очікуваний результат збігається з фактичним).
TC-02	Помилка дубліката логіну	1. На сторінці реєстрації ввести логін, який вже існує в БД. 2. Натиснути реєстрацію.	Система відхиляє запит, повертає помилку 400. Інтерфейс виводить повідомлення "Користувач вже існує".	Успішно. Запис у БД не продубльовано.
TC-03	Логіка Ейзенхауера: Важливо + Терміново	1. Ввести назву: "Здати диплом". 3. Поставити галочки "Важливе" та "Термінове". 4. Натиснути "Додати завдання"..	Backend розраховує Квадрант 1 і призначає 30 XP. Frontend малює картку в червоному секторі екрану.	Успішно. UI реактивно оновив перший квадрант.

Продовження таблиці 4.1

Ключові тест-кейси, розроблені для функціонального та інтеграційного тестування системи.

ID	Об'єкт тестування	Кроки для відтворення (Steps)	Очікуваний результат (Expected)	Фактичний результат / Статус
TC-04	Логіка Ейзенхауера: Не важливо + Не терміново	1. Створити завдання "Переглянути TikTok". 2. Залишити галочки порожніми. 3. Зберегти.	Backend розраховує Квадрант 4 і призначає 5 XP. Frontend малює картку в сірому секторі інтерфейсу.	Успішно. Маршрутизація пройшла валідно.
TC-05	Гейміфікація: Нарахування досвіду та Level-UP	1. Відкрити панель з поточним станом (баланс = 90 XP). 2. Завершити завдання з Квадранту 1 (+30 XP).	Сервер додає 30 XP (всього 120 XP). Формула FLOOR(120/100)+1 підвищує рівень до 2-го. UI відображає анімацію Confetti та залишок 20 XP.	Успішно. Анімація запускається лише в момент досягнення порогу у 100 балів.
TC-06	Безпека: Доступ без токена	1. Очистити LocalStorage (видалити токен JWT). 2. Спробувати створити завдання через API-виклик.	Nginx передає запит, проте Middleware Express повертає HTTP статус 401 Unauthorized.	Успішно. Спроба обходу блокується.

На основі успішного проходження наведених вище тест-кейсів було підтверджено повну функціональну готовність застосунку.

Щоб проілюструвати графічний результат проходження тестування (наслідок виконання тест-кейсів ТС-01 – ТС-05), на рисунках 4.1 – 4.3 наведено екранні форми розробленого застосунку з боку клієнта. Оформлення інтерфейсу підтверджує дотримання вимог до дизайну у стилі Glassmorphism (яскравий візуальний фокус, використання напівпрозорих елементів та неонової палітри) для зниження когнітивного навантаження.

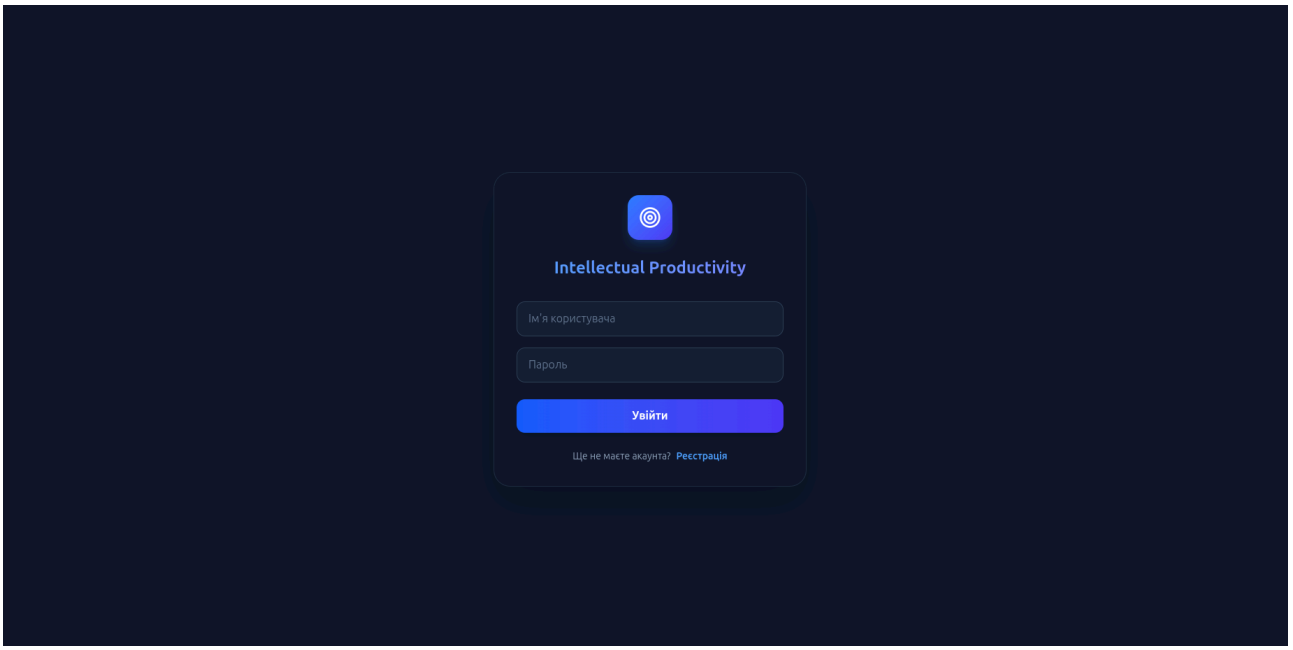


Рисунок 4.1 – Екранна форма захищеної авторизації користувача

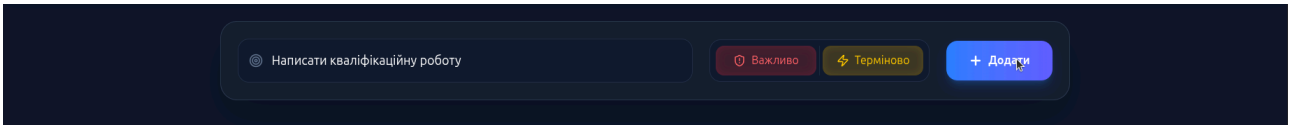


Рисунок 4.2 – Інтерфейс створення завдання та визначення його пріоритету

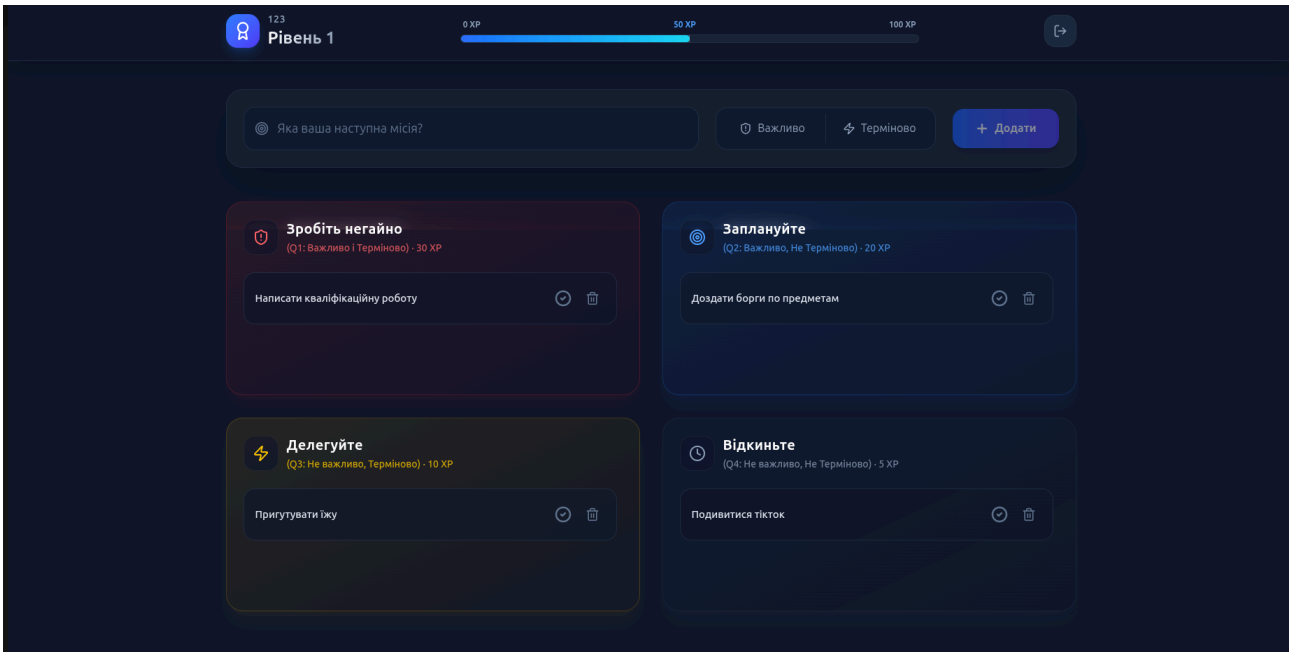


Рисунок 4.3 – Головна інформаційна панель (Dashboard) Матриці Ейзенхауера із прогрес-баром гейміфікації

Впроваджена візуальна сегрегація (кольоровий поділ квадрантів) та наявність прогрес-бара успішно виконують функцію зовнішнього мотиватора, спонукаючи користувача розпочинати роботу із завдань першого квадранту задля отримання найбільшої ігрової винагороди (30 XP).

4.3. Результати тестування та перевірка експлуатаційної здатності

На основі розробленого плану тестування (Таблиця 4.1) було проведено повний цикл верифікації інформаційної системи. Тестування виконувалося у два етапи: спочатку в локальному середовищі розробника (Development Environment), а потім – у змодельованому експлуатаційному середовищі (Production).

Усі проведені тест-кейси (TC-01 – TC-06) завершилися успішно, що дозволило підтвердити одразу кілька ключових аспектів роботи системи. По-перше, було доведено безвідмовність алгоритмів гейміфікації: математична функція розрахунку XP стабільно обчислює ігровий рівень користувача залежно

від обраного квадранта Ейзенхауера, не викликаючи при цьому збоїв чи перевантаження бази даних. По-друге, підтверджено загальну стабільність клієнтського інтерфейсу. Реалізований стиль Glassmorphism працює плавно і не створює візуальних артефактів навіть під час інтенсивної зміни станів React-компонентів, зокрема під час відтворення анімації винагороди (Confetti). Зрештою, успішне проходження тестів засвідчило високу надійність міжкомпонентної взаємодії: REST API коректно обробляє вхідні запити та повертає стандартизовані JSON-відповіді, а СУБД PostgreSQL гарантує цілісність збереження даних завдяки підтримці транзакцій.

Важливою складовою експлуатаційного тестування (Deployment Testing) є перевірка системи на стійкість до вразливостей, характерних для вебдодатків. Оскільки система зберігає конфіденційні облікові записи та ігрову статистику, було проведено симуляцію атак за методологією OWASP Top 10 [16]. Однією з найпоширеніших загроз для реляційних баз даних є атака типу SQL-ін'єкція (SQL Injection), коли зловмисник намагається маніпулювати вхідними даними, щоб змусити сервер виконати несанкціонований запит до СУБД.

Окремої уваги заслуговує тест-кейс TC-07, спрямований на перевірку безпеки застосунку, зокрема захисту від SQL-ін'єкцій. Головною метою цього випробування було підтвердити, що використання параметризованих запитів у драйвері node-postgres повністю нівелює ризики виконання шкідливого коду. За сценарієм тестування, через поле введення логіна на сторінці реєстрації здійснювалася спроба передати класичний шкідливий рядок типу ' OR '1'='1' --. У результаті перевірки бібліотека автоматично та безпечно екранувала всі значення, передані як змінні $1 та $2 у SQL-вираз INSERT INTO. Завдяки цьому механізму СУБД PostgreSQL сприйняла введений шкідливий рядок виключно як звичайний текстовий літерал, а не як системну команду. Фактичний результат тесту виявився успішним: сервер коректно відхилив спробу маніпуляції даними, що на практиці доводить високу стійкість та безпечність розробленої архітектури.

Для перевірки працездатності розробленого програмного забезпечення в реальних експлуатаційних умовах було здійснено комплексне фінальне

інфраструктурне тестування. Головною метою цього етапу було практично довести спроможність системи стабільно функціонувати як незалежний та захищений мережевий сервіс у режимі 24/7. Для цього вебзастосунок було успішно розгорнуто на базі виділеного апаратного міні-сервера (модель ThinkCentre m715q). Його апаратних потужностей, що включають 8 ГБ оперативної пам'яті (RAM) та швидкісний твердотільний накопичувач (SSD) об'ємом 120 ГБ, під управлінням серверної операційної системи Linux Ubuntu, виявилося цілком достатньо для забезпечення миттєвого відгуку бази даних PostgreSQL та стабільного безперебійного функціонування усіх розроблених Docker-контейнерів.

З метою забезпечення повної апаратної незалежності та надійної інкапсуляції середовища виконання було застосовано сучасні механізми контейнеризації технології Docker. Використовуючи декларативний конфігураційний файл docker-compose.yml, усю архітектуру системи було автоматизовано піднято у вигляді трьох незалежних ізольованих контейнерів: реляційної бази даних PostgreSQL, бекенд-сервера (Node.js) та Frontend-вітрини на базі вебсервера Nginx. Процес розгортання та ініціалізації цих мікросервісів здійснювався із суворим дотриманням актуальних вимог до мережевої ізоляції. Зокрема, для розмежування публічного та приватного трафіку було створено два окремі віртуальні мережеві контури (app-network та web_network), що гарантує безпеку та унеможливлює прямий доступ до бази даних із зовнішнього середовища.

На рисунку 4.4 наведено детальну діаграму розгортання (Deployment Diagram), яка візуально ілюструє топологію та безпечні маршрути взаємодії розроблених Docker-контейнерів у межах серверної інфраструктури. Ця графічна модель комплексно відображає архітектурну організацію проєкту на рівні фізичних та логічних вузлів, наочно демонструючи взаємозв'язки між ізольованими сервісами. Представлена схема детально описує конфігурацію мережевого середовища та протоколи обміну даними, що використовуються для

забезпечення стабільного й захищеного функціонування системи в процесі її віддаленої експлуатації.

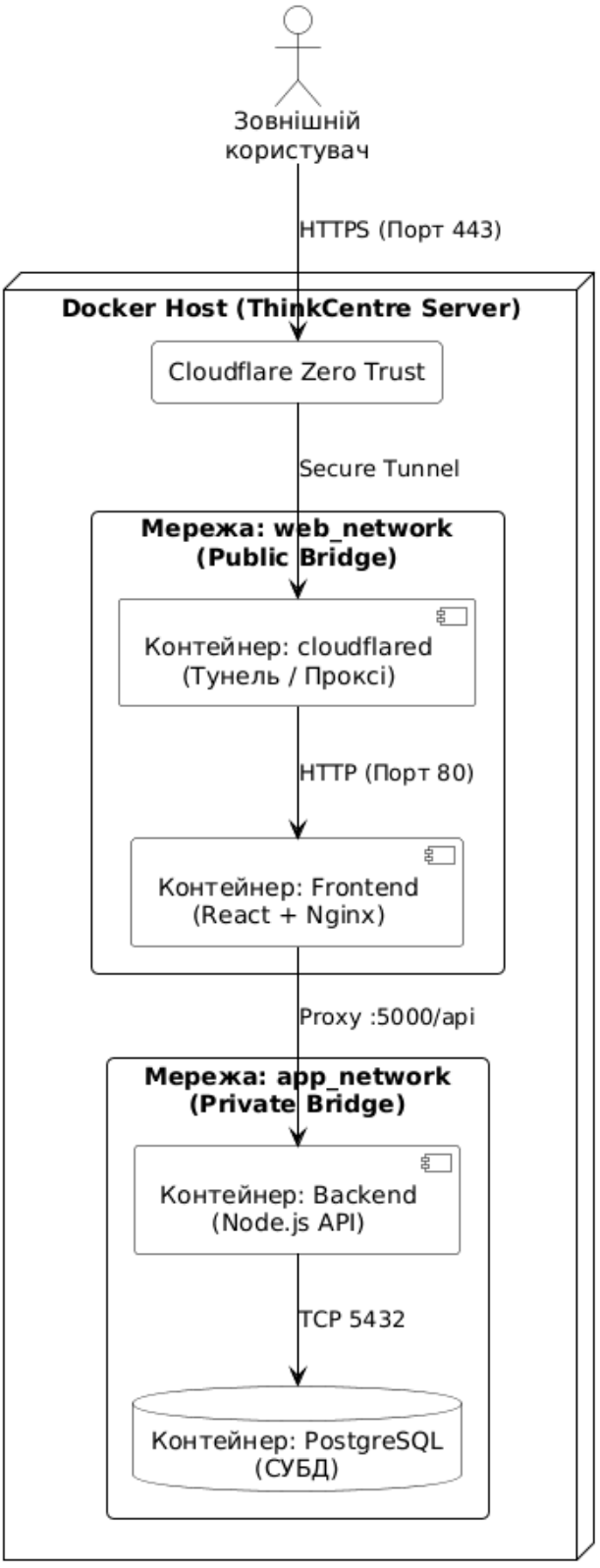


Рис. 4.4. – Діаграма розгортання контейнеризованих компонентів

Ключовим елементом інфраструктури розгортання виступає компонент вебсервера, що відповідає за балансування навантаження, кешування та роздачу статичних файлів клієнтського застосунку. У рамках розробленої системи цю роль виконує HTTP-сервер Nginx, який функціонує в режимі Зворотного проксі (Reverse Proxy) [12]. При розгортанні React-застосунків у Production-середовищі, вихідний код JSX/JavaScript компілюється бандлером (Vite) у набір статичних HTML, CSS та мініфікованих JS файлів. Nginx було обрано через його подієво-орієнтовану (Event-driven) асинхронну архітектуру. З метою забезпечення високого рівня безпеки та уникнення складних налаштувань CORS-політик на Node.js сервері, Nginx було сконфігуровано як єдину точку доступу (API Gateway). Усі HTTP-запити від браузера клієнта надходять на один і той самий домен todo.think-home.pp.ua. Завдяки блоку location /api/, Nginx перехоплює всі специфічні виклики та прозоро "проксіює" їх до ізольованого Backend-контейнера у внутрішній мережі. Такий архітектурний підхід повністю ізолює Node.js бекенд від зовнішнього середовища.

Завершальним тестом безпеки стала перевірка захищеного доступу користувачів з мережі Інтернет. Розгортання системи в ізольованому середовищі вимагає особливої уваги до питань кібербезпеки та захисту від відмов мережі. Класичний підхід до публікації вебсервісів на домашньому обладнанні зазвичай передбачає фізичне відкриття портів на маршрутизаторі (Port Forwarding). Однак цей підхід робить сервер беззахисним перед прямими DDoS-атаками.

Для усунення цих ризиків у межах розробленої системи було використано інноваційну архітектуру Cloudflare Zero Trust (Tunnels) [5]. Застосунок було підключено до корпоративної магістралі шляхом конфігурування захищеного тунелю. Замість того, щоб сервер Nginx "слухав" зовнішні вхідні підключення, спеціальний демон cloudflared (розгорнутий як окремий Docker-контейнер) самостійно ініціює вихідне зашифроване з'єднання з найближчим дата-центром Cloudflare [8]. Оскільки немає відкритих вхідних портів (Inbound ports), зловмисники не можуть просканувати домашній сервер чи дізнатися його публічну IP-адресу. Це дозволило безпечно маршрутизувати вебтрафік на адресу

todo.think-home.pp.ua безпосередньо у внутрішній контейнер Nginx, унеможлививши DDoS-атаки під час експерименту.

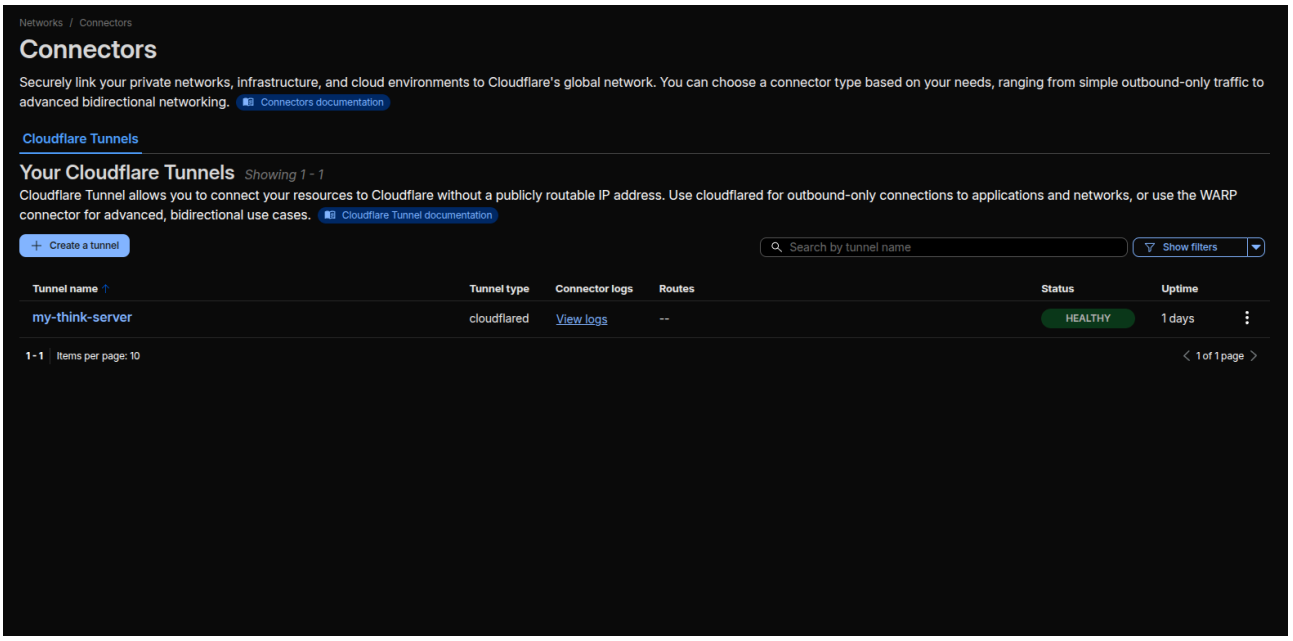


Рис. 4.5. Статус успішно налаштованого захищеного з'єднання Cloudflare Zero Trust

Конфігураційні файли, необхідні для автоматизованого розгортання інформаційної системи в ізольованому середовищі (зокрема, інструкції docker-compose.yml та конфіг зворотного проксі-сервера Nginx), наведено в Додатку В.

ВИСНОВКИ

У межах бакалаврської кваліфікаційної роботи вирішено актуальне науково-практичне завдання розробки інформаційної системи управління завданнями з інтегрованими елементами гейміфікації. Метою роботи було створення програмного продукту для оптимізації процесу планування та зниження когнітивного навантаження користувачів шляхом поєднання алгоритму пріоритезації та механік віртуального заохочення.

За результатами проведеного дослідження та розробки отримано такі результати:

Проведено порівняльний аналіз існуючих рішень для тайм-менеджменту зокрема: Todoist, Trello, Habitica. Виявлено архітектурні та функціональні обмеження наявних систем, що полягають у дисбалансі між строгим утилітарним підходом та надмірним фокусом на ігрових елементах. Обґрунтовано доцільність імплементації гейміфікації як інструменту підвищення рівня залученості користувача та протидії прокрастинації.

Розроблено архітектуру клієнт-серверного застосунку та спроєктовано структуру бази даних. Побудовано UML-діаграми прецедентів (Use Case) та розроблено ER-модель для реляційної СУБД PostgreSQL. Нормалізація схеми бази даних забезпечила цілісність та консистентність збереження профілів користувачів і поточного ігрового прогресу, усунувши ризики дублювання інформації.

Систему реалізовано з використанням екосистеми JavaScript. Серверну частину (Backend) побудовано на базі асинхронної платформи Node.js з використанням фреймворку Express.js з дотриманням архітектурного стилю REST API. Що дозволило інкапсулювати бізнес-логіку обробки транзакцій та алгоритми нарахування ігрового досвіду XP. Клієнтську частину розроблено у форматі SPA за допомогою бібліотеки React та фреймворку Tailwind CSS, що забезпечило високу швидкість відмальовування інтерфейсу та асинхронну взаємодію з

сервером без перезавантаження сторінок.

Тестування та розгортання. Проведено функціональне, модульне та інтеграційне тестування компонентів системи. Застосунок розгорнуто в експлуатаційному середовищі з використанням технології контейнеризації Docker для забезпечення ізольованості та масштабованості сервісів. Захист мережевої взаємодії, приховування IP-адреси сервера та запобігання DDoS-атакам реалізовано за допомогою налаштування криптографічного тунелю Cloudflare Zero Trust.

Практичне значення одержаних результатів полягає у створенні готового до експлуатації програмного продукту, що автоматизує розподіл завдань за пріоритетністю та стимулює продуктивність цільової аудиторії фахівців IT-сфери, менеджерів, студентів.

З інженерної точки зору архітектура розробленої системи є масштабованою та відмовостійкою. Перспективи подальшого розвитку продукту полягають у розробці нативних мобільних клієнтів, впровадженні модулів для командної роботи та інтеграції алгоритмів машинного навчання для предиктивного аналізу індивідуальних патернів продуктивності користувачів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Steel P. The nature of procrastination: A meta-analytic and theoretical review of quintessential self-regulatory failure. Psychological Bulletin, 2007. Vol. 133, no. 1. P. 65–94. URL: https://doi.org/10.1037/0033-2909.133.1.65 (дата звернення: 16.04.2026).
2. Todoist. Офіційний вебсайт програмного продукту для управління завданнями. URL: https://todoist.com (дата звернення: 16.04.2026).
3. Trello. Офіційний вебсайт платформи для управління проектами на базі канбан-дошок. URL: https://trello.com (дата звернення: 16.04.2026).
4. Habitica. Офіційний вебсайт застосунку для гейміфікації завдань. URL: https://habitica.com (дата звернення: 16.04.2026).
5. React – Офіційна документація. Функціональні компоненти та хуки. URL: https://react.dev (дата звернення: 16.04.2026).
6. Node.js – Офіційна документація середовища виконання. URL: https://nodejs.org (дата звернення: 16.04.2026).
7. PostgreSQL – Офіційна документація об'єктно-реляційної СУБД. URL: https://www.postgresql.org (дата звернення: 16.04.2026).
8. Coursera. What Is Client-Server Architecture? Concept and Examples. URL: https://www.coursera.org (дата звернення: 16.04.2026).
9. Object Management Group (OMG). Unified Modeling Language (UML) Specification. Збірник специфікацій мови моделювання. URL: https://www.omg.org/spec/UML/ (дата звернення: 16.04.2026).
10. Chen P. P.-S. The Entity-Relationship Model - Toward a Unified View of Data. ACM Transactions on Database Systems, 1976. Vol. 1, no. 1. P. 9–36. URL: https://doi.org/10.1145/320434.320440 (дата звернення: 16.04.2026).
11. Nielsen Norman Group (NN/g). Glassmorphism in User Interfaces. URL: https://www.nngroup.com/ (дата звернення: 16.04.2026).

12. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures. Докторська дисертація, University of California, Irvine, 2000. URL: https://www.gbiv.com/fragments/rest/fielding/ (дата звернення: 16.04.2026).
13. Deterding S., Dixon D., Khaled R., Nacke L. From Game Design Elements to Gamefulness: Defining Gamification. Proceedings of the 15th International Academic MindTrek Conference, 2011. P. 9–15. URL: https://doi.org/10.1145/2181037.2181040 (дата звернення: 16.04.2026).
14. Tailwind CSS – Офіційна документація utility-first фреймворку. URL: https://tailwindcss.com (дата звернення: 16.04.2026).
15. International Software Testing Qualifications Board (ISTQB). Certified Tester Foundation Level (CTFL) Syllabus. URL: https://www.istqb.org/ (дата звернення: 17.04.2026).
16. UX Pilot. The Psychology of Glassmorphism and Extraneous Cognitive Load. URL: https://uxpilot.ai/ (дата звернення: 20.04.2026).
17. React. Preserving and Resetting State. URL: https://react.dev/learn/preserving-and-resetting-state (дата звернення: 20.04.2026).
18. Auth0. JSON Web Token Introduction. URL: https://jwt.io/introduction (дата звернення: 20.04.2026).
19. Axios. Interceptors. URL: https://axios-http.com/docs/interceptors (дата звернення: 20.04.2026).
20. Canvas-confetti. Офіційна документація бібліотеки. URL: https://www.npmjs.com/package/canvas-confetti (дата звернення: 20.04.2026).
21. OWASP Foundation. SQL Injection Prevention Cheat Sheet. URL: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html (дата звернення: 20.04.2026).
22. NGINX. Reverse Proxy. URL: https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/ (дата звернення: 20.04.2026).

23. Cloudflare. Zero Trust Architecture and DDoS Mitigation. Офіційна документація Cloudflare. URL: https://developers.cloudflare.com/cloudflare-one/ (дата звернення: 20.04.2026).
24. Elest.io. How Cloudflare Tunnel defeats DDoS attacks by hiding your origin server. URL: https://elest.io/ (дата звернення: 20.04.2026).
25. Оводов Р. О., Гавор А. С. Аналіз методів гейміфікації та алгоритмів пріоритетизації в системах управління персональною продуктивністю. // Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. К.: ДУІКТ, 2026, (подано до друку).
26. Оводов Р. О., Гавор А. С. Проєктування та технічна реалізація web-застосунку для управління продуктивністю з елементами гейміфікації. // Матеріали Всеукраїнської науково-технічної конференції «Технології цифрового розвитку: програмні системи та децентралізація». Збірник тез. К.: ДУІКТ, 2026, (подано до друку).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ТЕХНОЛОГІЇ ЦИФРОВОГО РОЗВИТКУ

КВАЛІФІКАЦІЙНА РОБОТА

Інтелектуальний web-застосунок для управління персональною продуктивністю з використанням методів гейміфікації та пріоритезації

Виконав: студент групи ТЦР-44 Оводов Ростислав

Керівник: викладач кафедри ТЦР Гавор Артур

Київ - 2026

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи – підвищення ефективності виконання завдань і зниження когнітивного навантаження при плануванні за рахунок розробки програмної системи, що поєднує метод Ейзенхауера з мотиваційною гейміфікацією.

Об’єкт дослідження – процес організації, пріоритезації та контролю виконання особистих завдань користувача.

Предмет дослідження – програмне забезпечення для підвищення персональної продуктивності з використанням алгоритму матриці Ейзенхауера та ігрових механік.

ПОСТАНОВКА ЗАДАЧ

1. Дослідити предметну область: проаналізувати теоретичні аспекти прокрастинації та існуючі програмні рішення для управління продуктивністю.
2. Спроєктувати цільову клієнт-серверну архітектуру, структуру бази даних (PostgreSQL) та користувацький інтерфейс вебзастосунку.
3. Розробити програмну систему, реалізувавши серверну частину (Node.js) та клієнтську частину (React) з алгоритмом гейміфікованої матриці Ейзенхауера.
4. Провести тестування програмного забезпечення для верифікації функціоналу, безпеки та алгоритмів.
5. Розробити рекомендації щодо використання та продемонструвати експлуатаційну здатність системи шляхом її розгортання.

АКТУАЛЬНІСТЬ РОБОТИ

1. Проблема прокрастинації:

- До 95% фахівців стикаються з відкладанням справ через "когнітивне перевантаження" та нездатність мозку швидко структурувати завдання.

2. Недоліки існуючих рішень:

- Класичні інструменти (Todoist, Trello) фокусуються на фіксації даних, залишаючи осторонь поведінкову мотивацію.
- Спеціалізовані гейміфіковані рішення (Habitica) ігнорують концепцію важливості та терміновості.

3. Запропонована ідея:

- Розробка системи, яка об'єднує гейміфікацію з алгоритмом Ейзенхауера: бали досвіду динамічно залежать від квадранту задачі (наприклад, 30 XP за важливі/термінові, 5 XP за рутину).

ПОРІВНЯННЯ З ІСНУЮЧИМИ РІШЕННЯМИ

Характеристика	Todoist / Trello	Habitica	Розроблена система
Матриця Ейзенхауера (пріоритезація)	Ні	Ні	Так
Гейміфікація (XP, Level, Progression)	Ні	Так	Так
Залежність нагороди від пріоритету	Ні	Ні	Так
Сучасний UI	Так	Так	Так
Безпека (JWT, SQLi protection)	Так	Так	Так

АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

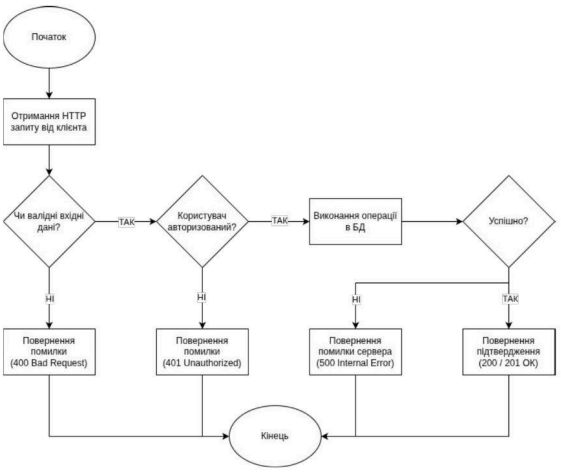
Frontend-модуль: обробка користувацького вводу, динамічне відображення стану завдань.

Backend-модуль: маршрутизація запитів, бізнес-логіка валідації та авторизації.

Модуль Бази Даних: збереження метаданих завдань, статусів виконання та управління сесіями користувачів.

АЛГОРИТМ ОБРОБКИ ДАНИХ

1. Отримання та валідація вхідних даних від користувача.
2. Перевірка цілісності даних та прав доступу.
3. Виконання CRUD-операцій (Створення, Читання, Оновлення, Видалення) у базі даних.
4. Повернення стандартизованої відповіді сервером.



МАТЕМАТИЧНА МОДЕЛЬ ГЕЙМІФІКАЦІЇ

Для забезпечення динамічної системи мотивації та реактивного оновлення UI, впроваджено наступну математичну модель:

1. Формула розрахунку поточного ігрового рівня (Level):

$$\text{Level} = \left\lfloor \frac{XP_{total}}{100} \right\rfloor + 1$$

XP_{Total} — сумарна кількість накопичених користувачем балів досвіду;

2. Формула розрахунку ширини індикатора прогресу (W) у відсотках:

$$W = (XP_{total} \bmod 100)\%$$

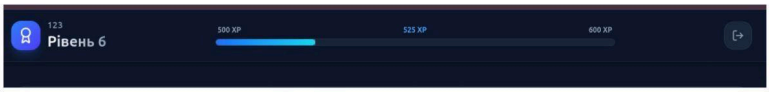
• При досягненні значення 100, ініціюється подія Level-UP, а візуальна шкала обнуляється для нового циклу.

РЕЗУЛЬТАТИ ТЕСТУВАННЯ

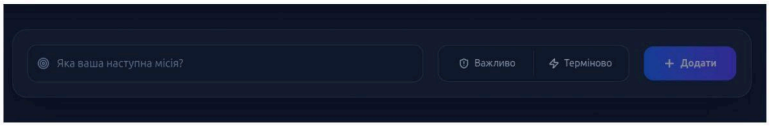
ID	Об'єкт перевірки	Сценарій тестування (Payload / Action)	Очікуваний результат (згідно з кодом)	Фактичний результат (Верифікація)
TC-01	Логіка Ейзенхауера	POST /api/tasks {is_important: true, is_urgent: true}	Обчислення Quadrant: 1, XP_reward: 30	HTTP 201. Валідація функції calculateQuadrantAndXP
TC-02	Гейміфікація	PATCH /tasks/:id/complete (XP: 90 -> 120)	Зміна Level: 1 -> 2 (за формулою FLOOR)	HTTP 200. Оновлення total_xp та level у БД
TC-03	Авторизація (JWT)	GET /api/dashboard (без заголовка Auth)	Перехоплення Middleware authenticateToken	HTTP 401 Unauthorized (успішно пройдено Jest)
TC-04	Безпека (SQLi)	Введення ' OR '1'='1' у поля запитів	Параметризація значень через pg-driver ($1)	Екранування спецсимволів. Помилок БД не виявлено
TC-05	Контейнеризація	Команда: docker-compose up	Ізоляція сервісів backend, frontend та db	Контейнери у стані 'Up'. Доступ через Nginx

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

Форма входу та реєстрації

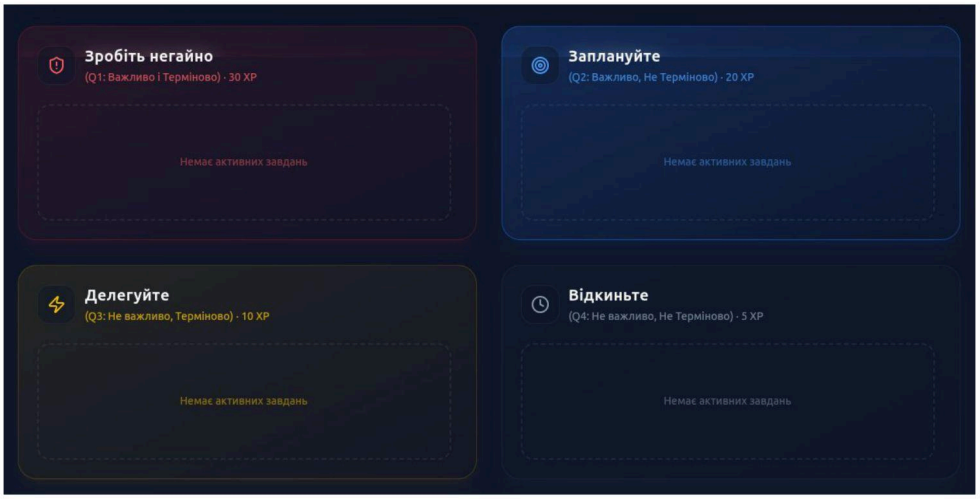


Рівень, шкала прогресу, та кнопка “Вихід”



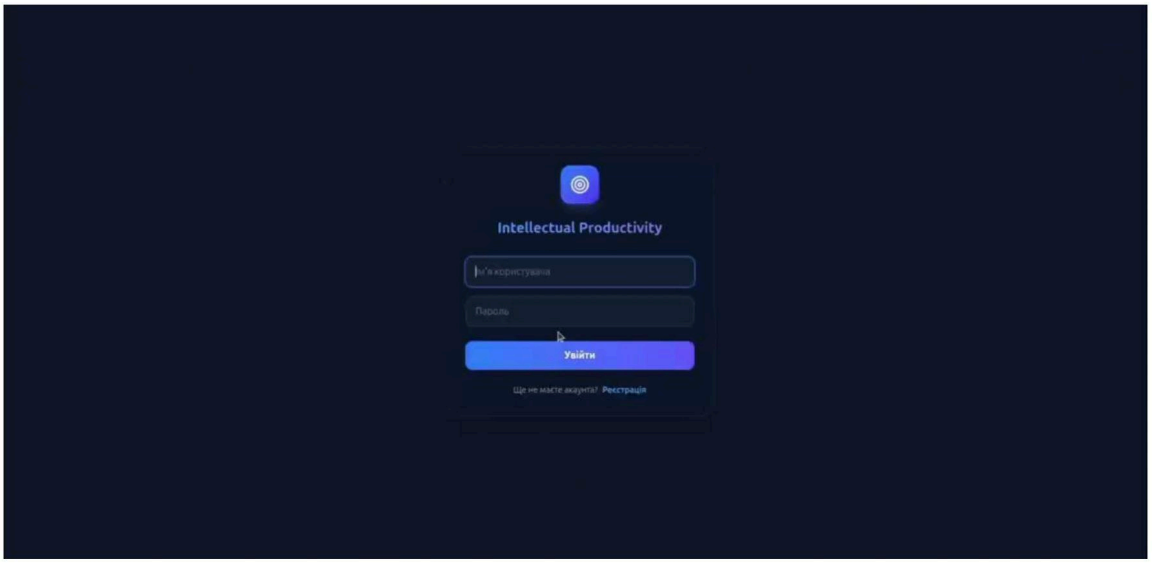
Форма для додавання нових задач

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ



Квадранти за пріоритетністю

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ



АПРОБАЦІЯ РОБОТИ

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

1. Оводов Р. О., Гавор А. С. Аналіз методів гейміфікації та алгоритмів пріоритетизації в системах управління персональною продуктивністю. // Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. К.: ДУІКТ, 2026, (подано до друку).
2. Оводов Р. О., Гавор А. С. Проєктування та технічна реалізація web-застосунку для управління продуктивністю з елементами гейміфікації. // Матеріали Всеукраїнської науково-технічної конференції «Технології цифрового розвитку: програмні системи та децентралізація». Збірник тез. К.: ДУІКТ, 2026, (подано до друку).

ВИСНОВКИ

1. На основі дослідження предметної області обґрунтовано доцільність поєднання матриці Ейзенхауера з психологічними методами гейміфікації для боротьби з прокрастинацією.

2. Проведено проєктування системи: розроблено архітектуру, строго нормалізовану реляційну схему для СУБД PostgreSQL та UI/UX інтерфейс у стилі Glassmorphism.

3. Здійснено глибоку розробку клієнт-серверної системи на базі сучасного JavaScript-стеку (Express.js, React) з імплементацією реактивної ігрової логіки.

4. Проведено комплексне функціональне та інтеграційне тестування, що підтвердило стабільність і безвідмовність створеного програмного забезпечення.

5. Надано рекомендації щодо використання та успішно продемонстровано розгортання застосунку на Linux-сервері із застосуванням Docker та Cloudflare Zero Trust.

ДЯКУЮ ЗА УВАГУ

ДОДАТОК Б. ПРОГРАМНИЙ КОД СЕРВЕРНОЇ ЧАСТИНИ WEB-ЗАСТОСУНКУ

```
const express = require('express');
const cors = require('cors');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
const db = require('./db');
const { authenticateToken, JWT_SECRET } = require('./auth');

const app = express();
app.use(cors());
app.use(express.json());

const calculateQuadrantAndXP = (is_important, is_urgent) => {
  if (is_important && is_urgent) return { quadrant: 1, xp_reward: 30 };
  if (is_important && !is_urgent) return { quadrant: 2, xp_reward: 20 };
  if (!is_important && is_urgent) return { quadrant: 3, xp_reward: 10 };
  return { quadrant: 4, xp_reward: 5 };
};

// --- AUTHENTICATION ROUTES ---

app.post('/api/auth/register', async (req, res) => {
  try {
    const { username, password } = req.body;
    const existing = await db.query('SELECT * FROM users WHERE username = $1',
[username]);
    if (existing.rows.length > 0) return res.status(400).json({ error: 'Користувач вже
існує' });

    const salt = await bcrypt.genSalt(10);
    const hash = await bcrypt.hash(password, salt);

    const result = await db.query(
      'INSERT INTO users (username, password_hash) VALUES ($1, $2)
```

```
RETURNING id, username, total_xp, level',
      [username, hash]
    );
    res.status(201).json({ success: true, user: result.rows[0] });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

app.post('/api/auth/login', async (req, res) => {
  try {
    const { username, password } = req.body;
    const result = await db.query('SELECT * FROM users WHERE username = $1',
[username]);
    if (result.rows.length === 0) return res.status(401).json({ error: 'Користувача не
знайдено' });

    const user = result.rows[0];
    const validPassword = await bcrypt.compare(password, user.password_hash);
    if (!validPassword) return res.status(401).json({ error: 'Неправильний пароль' });

    const token = jwt.sign({ id: user.id, username: user.username }, JWT_SECRET, {
expiresIn: '7d' });
    res.json({ success: true, token, user: { id: user.id, username: user.username,
total_xp: user.total_xp, level: user.level } });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

// --- PROTECTED ROUTES ---

app.post('/api/tasks', authenticateToken, async (req, res) => {
  try {
    const { title, is_important, is_urgent } = req.body;
    const { quadrant, xp_reward } = calculateQuadrantAndXP(is_important, is_urgent);
    const result = await db.query(
      `INSERT INTO tasks (user_id, title, is_important, is_urgent, quadrant, status,
```

```
xp_reward)
       VALUES ($1, $2, $3, $4, $5, 'active', $6) RETURNING *`,
      [req.user.id, title, is_important, is_urgent, quadrant, xp_reward]
    );
    res.status(201).json(result.rows[0]);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

app.patch('/api/tasks/:id/complete', authenticateToken, async (req, res) => {
  try {
    const { id } = req.params;

    const taskResult = await db.query(
      `UPDATE tasks SET status = 'completed' WHERE id = $1 AND user_id = $2
AND status = 'active' RETURNING xp_reward`,
      [id, req.user.id]
    );

    if (taskResult.rowCount === 0) {
      return res.status(404).json({ error: 'Task not found or already completed.' });
    }
    const { xp_reward } = taskResult.rows[0];

    const userResult = await db.query(
      `UPDATE users
       SET total_xp = total_xp + $1, level = FLOOR((total_xp + $1) / 100) + 1
       WHERE id = $2 RETURNING id, username, total_xp, level`,
      [xp_reward, req.user.id]
    );

    res.json({ success: true, xp_reward, user: userResult.rows[0] });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

```
app.delete('/api/tasks/:id', authenticateToken, async (req, res) => {
  try {
    const { id } = req.params;
    const result = await db.query('DELETE FROM tasks WHERE id = $1 AND
user_id = $2 RETURNING *', [id, req.user.id]);
    if (result.rowCount === 0) {
      return res.status(404).json({ error: 'Task not found' });
    }
    res.json({ success: true, task: result.rows[0] });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

app.get('/api/dashboard', authenticateToken, async (req, res) => {
  try {
    const userResult = await db.query('SELECT id, username, total_xp, level FROM
users WHERE id = $1', [req.user.id]);
    const tasksResult = await db.query('SELECT * FROM tasks WHERE user_id = $1
ORDER BY created_at DESC', [req.user.id]);
    res.json({
      user: userResult.rows[0],
      tasks: tasksResult.rows
    });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

module.exports = app;

if (require.main === module) {
  const PORT = process.env.PORT || 5000;
  app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
}
```

ДОДАТОК В. ПРОГРАМНИЙ КОД КЛІЄНТСЬКОЇ ЧАСТИНИ ТА ВІЗУАЛІЗАЦІЇ МАТРИЦІ ЕЙЗЕНХАУЕРА

```
import React, { useState, useEffect } from 'react';
import axios from 'axios';
import { CheckCircle2, Plus, Target, Zap, Clock, ShieldAlert, Award, Trash2, LogOut,
Loader2 } from 'lucide-react';
import confetti from 'canvas-confetti';

const API_URL = import.meta.env.VITE_API_URL || '/api';

axios.interceptors.request.use(config => {
  const token = localStorage.getItem('token');
  if (token) config.headers.Authorization = `Bearer ${token}`;
  return config;
});

function App() {
  const [token, setToken] = useState(localStorage.getItem('token'));

  if (!token) {
    return <AuthScreen onLogin={setToken} />;
  }

  return <Dashboard onLogout={() => { localStorage.removeItem('token');
setToken(null); }} />;
}

function AuthScreen({ onLogin }) {
  const [isLogin, setIsLogin] = useState(true);
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');
  const [err, setErr] = useState('');
  const [loading, setLoading] = useState(false);

  const handleSubmit = async (e) => {
```

```
    e.preventDefault();
    setErr('');
    if (!username || !password) return setErr('Заповніть всі поля');
    setLoading(true);

    try {
      const endpoint = isLogin ? '/auth/login' : '/auth/register';
      const res = await axios.post(`${API_URL}${endpoint}`, { username, password
});

      if (!isLogin) {
        const loginRes = await axios.post(`${API_URL}/auth/login`, { username,
password });
        localStorage.setItem('token', loginRes.data.token);
        onLogin(loginRes.data.token);
      } else {
        localStorage.setItem('token', res.data.token);
        onLogin(res.data.token);
      }
    } catch (err) {
      setErr(err.response?.data?.error || 'Помилка з\'єднання сервером');
    } finally {
      setLoading(false);
    }
  };

  return (
    <div className="min-h-screen bg-[#0f172a] text-white flex items-center
justify-center p-4 relative overflow-hidden">
      <div className="absolute inset-0 pointer-events-none
bg-[radial-gradient(ellipse_at_top,_var(--tw-gradient-stops))] from-slate-800
via-slate-900 to-black z-0"></div>

      <div className="bg-slate-900/60 backdrop-blur-xl border border-slate-700/50
p-8 rounded-3xl shadow-2xl w-full max-w-md relative z-10">
        <div className="flex justify-center mb-6">
          <div className="bg-gradient-to-br from-blue-500 to-indigo-600 p-4
rounded-2xl shadow-lg shadow-blue-500/20">
```

```
          <Target size={32} className="text-white" />
        </div>
      </div>
      <h2 className="text-2xl font-bold text-center mb-8 bg-clip-text text-transparent bg-gradient-to-r from-blue-400 to-indigo-400">
        Intellectual Productivity
      </h2>

      {err && <div className="bg-red-500/10 border border-red-500/20 text-red-400 p-3 rounded-xl mb-6 text-sm text-center">{err}</div>}

      <form onSubmit={handleSubmit} className="flex flex-col gap-4">
        <input
          type="text" value={username} onChange={e => setUsername(e.target.value)}
          placeholder="Ім'я користувача"
          className="w-full bg-slate-800/50 text-white rounded-xl px-4 py-3 border border-slate-700 focus:outline-none focus:ring-2 focus:ring-blue-500/50 transition-all placeholder:text-slate-500"
        />
        <input
          type="password" value={password} onChange={e => setPassword(e.target.value)}
          placeholder="Пароль"
          className="w-full bg-slate-800/50 text-white rounded-xl px-4 py-3 border border-slate-700 focus:outline-none focus:ring-2 focus:ring-blue-500/50 transition-all placeholder:text-slate-500"
        />
        <button
          type="submit" disabled={loading}
          className="w-full bg-gradient-to-r from-blue-600 to-indigo-600 hover:from-blue-500 hover:to-indigo-500 text-white py-3 rounded-xl font-bold transition-all shadow-lg hover:shadow-blue-500/25 mt-2 flex justify-center items-center h-12"
        >
          {loading ? <Loader2 className="animate-spin" /> : (isLogin ? 'Увійти' : 'Створити акаунт')}
        </button>
      </form>
```

```
        <p className="text-center mt-6 text-slate-400 text-sm">
          {isLogin ? 'Ще не маєте акаунта?' : 'Вже є акаунт?'}
          <button onClick={() => { setIsLogin(!isLogin); setErr(''); }}
className="text-blue-400 ml-2 hover:text-blue-300 font-semibold
focus:outline-none">
            {isLogin ? 'Реєстрація' : 'Увійти'}
          </button>
        </p>
      </div>
    </div>
  );
}

function Dashboard({ onLogout }) {
  const [tasks, setTasks] = useState([]);
  const [user, setUser] = useState({ username: '', total_xp: 0, level: 1 });
  const [title, setTitle] = useState('');
  const [is_important, setIsImportant] = useState(false);
  const [is_urgent, setIsUrgent] = useState(false);
  const [loading, setLoading] = useState(true);

  const fetchData = async () => {
    try {
      const res = await axios.get(`${API_URL}/dashboard`);
      setUser(res.data.user);
      setTasks(res.data.tasks);
    } catch (err) {
      if (err.response?.status === 401 || err.response?.status === 403) onLogout();
      console.error(err);
    } finally {
      setLoading(false);
    }
  };

  useEffect(() => { fetchData(); }, []);

  const handleAddTask = async (e) => {
```

```
    e.preventDefault();
    if (!title.trim()) return;
    try {
      await axios.post(`${API_URL}/tasks`, { title, is_important, is_urgent });
      setTitle('');
      setIsImportant(false);
      setIsUrgent(false);
      fetchData();
    } catch (err) {
      console.error(err);
    }
  };

  const handleComplete = async (id, e) => {
    try {
      const rect = e.currentTarget.getBoundingClientRect();
      const x = (rect.left + rect.width / 2) / window.innerWidth;
      const y = (rect.top + rect.height / 2) / window.innerHeight;

      confetti({ particleCount: 100, spread: 70, origin: { x, y }, colors: ['#3b82f6',
'#ef4444', '#eab308', '#22c55e'] });
      await axios.patch(`${API_URL}/tasks/${id}/complete`);
      fetchData();
    } catch (err) {
      console.error(err);
    }
  };

  const handleDelete = async (id) => {
    if (!window.confirm('Остаточно видалити цю місію?')) return;
    try {
      await axios.delete(`${API_URL}/tasks/${id}`);
      fetchData();
    } catch (err) {
      console.error(err);
    }
  };
```

```
  const quadrants = {
    1: tasks.filter(t => t.quadrant === 1 && t.status === 'active'),
    2: tasks.filter(t => t.quadrant === 2 && t.status === 'active'),
    3: tasks.filter(t => t.quadrant === 3 && t.status === 'active'),
    4: tasks.filter(t => t.quadrant === 4 && t.status === 'active')
  };

  const xpToNextLevel = user.level * 100;
  const progressPercent = Math.min((user.total_xp % 100) / 100 * 100, 100);

  if (loading) {
    return <div className="min-h-screen bg-[#0f172a] flex items-center
justify-center"><Loader2 className="animate-spin text-blue-500" size={48}
/></div>;
  }

  return (
    <div className="min-h-screen bg-[#0f172a] text-slate-200 font-sans
selection:bg-blue-500/30 overflow-x-hidden relative">
      <div className="fixed inset-0 pointer-events-none
bg-[radial-gradient(ellipse_at_top,_var(--tw-gradient-stops))] from-slate-800
via-slate-900 to-black z-0"></div>

      {/* Gamification Header */}
      <div className="sticky top-0 z-50 backdrop-blur-xl bg-slate-900/60 border-b
border-slate-700/50 shadow-[0_4px_30px_rgba(0,0,0,0.3)]">
        <div className="max-w-7xl mx-auto px-4 lg:px-8 py-4 flex flex-col
md:flex-row items-center justify-between gap-4">

          <div className="flex items-center gap-3">
            <div className="bg-gradient-to-br from-blue-500 to-indigo-600 w-12 h-12
rounded-2xl flex items-center justify-center shadow-lg shadow-blue-500/20 transform
hover:scale-105 transition-transform">
              <Award className="text-white w-7 h-7" />
            </div>
            <div>
              <h1 className="text-sm font-semibold text-slate-400 uppercase
tracking-wider">{user.username}</h1>
```

```
            <p className="text-2xl font-bold bg-clip-text text-transparent
bg-gradient-to-r from-white to-slate-400">Рівень {user.level}</p>
          </div>
        </div>

        <div className="flex-1 w-full max-w-2xl px-2">
          <div className="flex justify-between text-xs font-bold text-slate-400 mb-1.5
px-1">
            <span>{Math.floor(user.total_xp / 100) * 100} XP</span>
            <span className="text-blue-400 drop-shadow-sm">{user.total_xp}
XP</span>
            <span>{xpToNextLevel} XP</span>
          </div>
          <div className="w-full bg-slate-800/80 rounded-full h-3 border
border-slate-700/50 overflow-hidden shadow-inner relative">
            <div
              className="bg-gradient-to-r from-blue-600 to-cyan-400 h-full rounded-full
transition-all duration-1000 ease-out relative"
              style={{ width: `${progressPercent}%` }}
            >
              <div className="absolute inset-0 bg-white/20 animate-pulse border-r
border-white/50"></div>
            </div>
          </div>
        </div>

        <button onClick={onLogout} title="Вийти з акаунта"
className="bg-slate-800 hover:bg-slate-700 p-3 rounded-2xl border
border-slate-700/50 transition-colors">
          <LogOut size={20} className="text-slate-400" />
        </button>
      </div>
    </div>

    <main className="max-w-7xl mx-auto px-4 lg:px-8 pt-10 pb-20 relative z-10">

      {/* Task Form */}
      <form onSubmit={handleAddTask} className="bg-slate-800/40
```

```
backdrop-blur-md p-6 rounded-3xl border border-slate-700/50 shadow-2xl mb-12 flex flex-col lg:flex-row items-center gap-6 group hover:border-slate-600/50 transition-colors">
        <div className="w-full lg:flex-1 relative">
          <input
            type="text" value={title} onChange={e => setTitle(e.target.value)}
            placeholder="Яка ваша наступна місія?"
            className="w-full bg-slate-900/50 text-white rounded-2xl px-5 py-4 pl-12 border border-slate-700/60 focus:outline-none focus:ring-2 focus:ring-blue-500/50 transition-all shadow-inner text-lg placeholder:text-slate-500"
          />
          <Target className="absolute left-4 top-1/2 -translate-y-1/2 text-slate-500" size={20} />
        </div>

        <div className="flex bg-slate-900/50 p-2 rounded-2xl border border-slate-700/60 w-full lg:w-auto h-[60px]">
          <label className={`flex-1 lg:w-36 flex justify-center items-center gap-2 cursor-pointer rounded-xl transition-all ${is_important ? 'bg-red-500/20 text-red-400 shadow-[inset_0_0_12px_rgba(239,68,68,0.2)]' : 'hover:bg-slate-800 text-slate-400'}`}>
            <input type="checkbox" checked={is_important} onChange={() => setIsImportant(!is_important)} className="hidden"/>
            <ShieldAlert size={18} /> Важливо
          </label>
          <div className="w-px bg-slate-700/50 mx-1"></div>
          <label className={`flex-1 lg:w-36 flex justify-center items-center gap-2 cursor-pointer rounded-xl transition-all ${is_urgent ? 'bg-yellow-500/20 text-yellow-400 shadow-[inset_0_0_12px_rgba(234,179,8,0.2)]' : 'hover:bg-slate-800 text-slate-400'}`}>
            <input type="checkbox" checked={is_urgent} onChange={() => setIsUrgent(!is_urgent)} className="hidden"/>
            <Zap size={18} /> Терміново
          </label>
        </div>

        <button
          type="submit" disabled={!title.trim()}
```

```
          className="w-full lg:w-auto bg-gradient-to-r from-blue-600 to-indigo-600
hover:from-blue-500 hover:to-indigo-500 disabled:opacity-50
disabled:cursor-not-allowed font-bold text-white py-4 px-8 rounded-2xl transition-all
shadow-lg hover:shadow-blue-500/25 flex items-center justify-center gap-2"
        >
          <Plus size={20} strokeWidth={3} /> Додати
        </button>
      </form>

      {/* Matrix Grid */}
      <div className="grid grid-cols-1 md:grid-cols-2 gap-6 lg:gap-8
min-h-[400px]">
        <QuadrantBlock
          title="Зробіть негайно"
          desc="(Q1: Важливо і Терміново) · 30 XP"
          tasks={quadrants[1]} onComplete={handleComplete}
onDelete={handleDelete} colorScheme="red"
          icon={<ShieldAlert className="text-red-400" size={24} />}
        />
        <QuadrantBlock
          title="Заплануйте"
          desc="(Q2: Важливо, Не Терміново) · 20 XP"
          tasks={quadrants[2]} onComplete={handleComplete}
onDelete={handleDelete} colorScheme="blue"
          icon={<Target className="text-blue-400" size={24} />}
        />
        <QuadrantBlock
          title="Делегуйте"
          desc="(Q3: Не важливо, Терміново) · 10 XP"
          tasks={quadrants[3]} onComplete={handleComplete}
onDelete={handleDelete} colorScheme="yellow"
          icon={<Zap className="text-yellow-400" size={24} />}
        />
        <QuadrantBlock
          title="Відкиньте"
          desc="(Q4: Не важливо, Не Терміново) · 5 XP"
          tasks={quadrants[4]} onComplete={handleComplete}
onDelete={handleDelete} colorScheme="slate"
```

```
          icon={<Clock className="text-slate-400" size={24} />}
        />
      </div>
    </main>
  </div>
 );
}

function QuadrantBlock({ title, desc, tasks, onComplete, onDelete, colorScheme, icon
}) {
 const colors = {
  red: 'from-red-500/10 to-red-900/5 hover:from-red-500/20 border-red-500/20
hover:border-red-500/40 text-red-400 shadow-red-500/5',
  blue: 'from-blue-500/10 to-blue-900/5 hover:from-blue-500/20 border-blue-500/20
hover:border-blue-500/40 text-blue-400 shadow-blue-500/5',
  yellow: 'from-yellow-500/10 to-amber-900/5 hover:from-yellow-500/20
border-yellow-500/20 hover:border-yellow-500/40 text-yellow-400
shadow-yellow-500/5',
  slate: 'from-slate-700/10 to-slate-800/5 hover:from-slate-600/20
border-slate-600/20 hover:border-slate-500/40 text-slate-400 shadow-slate-500/5'
 }[colorScheme];

 return (
  <div className={`bg-gradient-to-br backdrop-blur-md rounded-3xl p-6 border
transition-all duration-300 flex flex-col shadow-xl ${colors}`}>
    <div className="flex items-center gap-3 mb-6">
     <div className="bg-slate-900/50 p-3 rounded-2xl shadow-inner border
border-white/5">
       {icon}
      </div>
      <div>
       <h2 className="text-xl font-bold text-white tracking-wide">{title}</h2>
       <p className="text-xs lg:text-sm font-medium opacity-80 mt-1">{desc}</p>
      </div>
    </div>

    <div className="flex-1 overflow-y-auto space-y-3 custom-scrollbar pr-2
min-h-[150px]">
```

```
        {tasks.length === 0 ? (
          <div className="h-full flex items-center justify-center border-2 border-dashed
border-slate-600/30 rounded-2xl p-6 text-center">
            <span className="text-sm font-medium opacity-50">Немає активних
завдань</span>
          </div>
        ) : (
          tasks.map(task => <TaskCard key={task.id} task={task}
onComplete={onComplete} onDelete={onDelete} colorScheme={colorScheme} />)
        )}
      </div>
    </div>
  );
}

function TaskCard({ task, onComplete, onDelete, colorScheme }) {
  const hoverColors = {
    red: 'hover:border-red-500/50 hover:shadow-[0_0_15px_rgba(239,68,68,0.15)]',
    blue: 'hover:border-blue-500/50 hover:shadow-[0_0_15px_rgba(59,130,246,0.15)]',
    yellow: 'hover:border-yellow-500/50
hover:shadow-[0_0_15px_rgba(234,179,8,0.15)]',
    slate: 'hover:border-slate-500/50
hover:shadow-[0_0_15px_rgba(100,116,139,0.15)]'
  }[colorScheme];

  const checkHoverColor = {
    red: 'hover:text-red-400 hover:bg-red-500/10',
    blue: 'hover:text-blue-400 hover:bg-blue-500/10',
    yellow: 'hover:text-yellow-400 hover:bg-yellow-500/10',
    slate: 'hover:text-slate-400 hover:bg-slate-500/10'
  }[colorScheme];

  return (
    <div className={`bg-slate-900/60 p-4 rounded-2xl flex justify-between
items-center group border border-slate-700/50 transition-all duration-300 transform
hover:-translate-y-1 ${hoverColors}`}>
      <span className="text-[15px] font-medium text-slate-200 leading-snug
pr-4">{task.title}</span>
```

```
      <div className="flex gap-1 flex-shrink-0">
        <button
          onClick={(e) => onComplete(task.id, e)}
          className={`p-2 text-slate-500 rounded-full transition-all duration-300
focus:outline-none ${checkHoverColor}`}
          title="Завершити місію"
        >
          <CheckCircle2 size={24} strokeWidth={2.5} className="transition-transform
hover:scale-110" />
        </button>
        <button
          onClick={() => onDelete(task.id)}
          className="p-2 text-slate-500 rounded-full transition-all duration-300
focus:outline-none hover:text-red-500 hover:bg-red-500/10"
          title="Видалити місію"
        >
          <Trash2 size={20} className="transition-transform hover:scale-110" />
        </button>
      </div>
    </div>
  );
}

export default App;
```

ДОДАТОК Г. КОНФІГУРАЦІЙНІ ФАЙЛИ КОНТЕЙНЕРИЗАЦІЇ ТА ПРОКСІ-СЕРВЕРА

Лістинг файлу оркестрації docker-compose.yml:

```
version: '3.8'

services:
  db:
    image: postgres:15-alpine
    container_name: productivity_db
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: secretpassword
      POSTGRES_DB: productivity_db
    ports:
      - "5432:5432"
    volumes:
      - pgdata:/var/lib/postgresql/data
      - ./init-db.sql:/docker-entrypoint-initdb.d/init.sql
    networks:
      - app-network

  backend:
    build: ./backend
    container_name: productivity_backend
    environment:
      DB_HOST: db
      DB_USER: postgres
      DB_PASSWORD: secretpassword
      DB_NAME: productivity_db
      PORT: 5000
    ports:
      - "5000:5000"
    depends_on:
      - db
    networks:
```

```
      - app-network

  frontend:
    build: ./frontend
    container_name: productivity_frontend
    ports:
      - "3000:80"
    depends_on:
      - backend
    networks:
      - app-network
      - web_network

volumes:
  pgdata:

networks:
  app-network:
    driver: bridge
  web_network:
    external: true
```

Конфігураційний файлу зворотного проксі nginx.conf

```
server {
    listen 80;
    server_name _;

    location / {
        root /usr/share/nginx/html;
        index index.html index.htm;
        try_files $uri $uri/ /index.html;
    }

    location /api/ {
        proxy_pass http://productivity_backend:5000/api/;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
```

```
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

ДОДАТОК Г. КЕРІВНИЦТВО КОРИСТУВАЧА ТА ІНСТРУКЦІЯ З ЕКСПЛУАТАЦІЇ

З метою забезпечення комфортного впровадження розробленого програмного забезпечення кінцевими споживачами, було розроблено структуроване керівництво. Воно розділене на інструкцію для адміністратора (відповідального за розгортання) та клієнтську пам'ятку.

Серверна частина застосунку є невимогливою до ресурсів завдяки оптимізації платформи Node.js та мікросервісній архітектурі. Мінімальні вимоги для сервера розгортання (Production Server):

- операційна система: Ubuntu 20.04 LTS / 22.04 LTS (або будь-який Linux-дистрибутив із підтримкою Docker).
- процесор: 1 ядро (мінімум 1 ГГц).
- оперативна пам'ять (RAM): 512 МБ (рекомендовано 1 ГБ для стабільної роботи бази даних PostgreSQL).
- дисковий простір: 2 ГБ вільного місця.
- програмні залежності: Docker Engine (версії 20.10+), Docker Compose (версії V2).

Розгортання системи здійснюється в автоматизованому режимі без необхідності ручного встановлення системних бібліотек на хост-машину. Послідовність дій системного адміністратора:

- завантажити файли проєкту (репозиторій) на цільовий сервер;
- у кореневій директорії проєкту створити конфігураційний файл .env та вказати секретні параметри середовища (пароль адміністратора бази даних POSTGRES_PASSWORD, секретний криптографічний ключ JWT_SECRET);
- відкрити системний термінал і перейти в директорію проєкту;

- виконати команду генерації віртуальної ізольованої мережі: sudo docker network create web_network;
- виконати команду збирання та запуску контейнерів у фоновому режимі (Detached mode): sudo docker compose up -d --build.

Після виконання цих дій демон Docker самостійно завантажить офіційні образи (Images) з публічного репозиторію Docker Hub, збере вихідний код фронтенду (Vite Build) і запустить 3 ізольовані сервіси: СУБД, API та Nginx. Система стане доступною за вказаним локальним портом або зовнішнім доменом.

Інструкція користувача системи:

- авторизація (крок 1): користувач відкриває web-застосунок у браузері, у разі порожнього локального сховища відбувається переадресація на екран входу, після чого формується безпечна сесія (видається JWT-токен);
- робоча панель (крок 2): після входу завантажується головна інформаційна панель Матриці Ейзенхауера, де у верхній частині відображається поточний рівень гравця та графічний прогрес-бар;
- створення завдання (крок 3): натиснувши кнопку «Додати завдання», користувач вводить текстовий опис у діалоговому вікні та суб'єктивно оцінює свою задачу за двома критеріями (важливість та терміновість);
- автоматичне сортування (крок 4): після збереження серверний математичний алгоритм аналізує вибір і розміщує картку завдання в один із чотирьох кольорових квадрантів сітки;
- гейміфікація та виконання (крок 5): коли фізична задача завершена, користувач натискає іконку "Виконано", а сервер негайно нараховує ігровий досвід (XP) відповідно до квадранту матриці;
- підвищення рівня (крок 6): щойно баланс досягає межі у 100 балів, користувач підвищує свій ігровий рівень, що супроводжується динамічною анімацією генерації віртуального конфетті для візуальної мотивації.і

ДОДАТОК Д. СТРУКТУРА РЕЛЯЦІЙНОЇ ТАБЛИЦІ «TASKS» (МАТРИЦЯ ЗАВДАНЬ)

Назва поля	Тип даних PostgreSQL	Обмеження (Constraints)	Опис та призначення атрибута
id	SERIAL (INT4)	PRIMARY KEY	Унікальний ідентифікатор завдання.
user_id	INTEGER	FOREIGN KEY REFERENCES users(id) ON DELETE CASCADE	Зв'язок із власником завдання. Видалення користувача каскадно видалить усі його задачі.
title	TEXT	NOT NULL	Текстовий зміст завдання, сформований користувачем.
is_important	BOOLEAN	DEFAULT FALSE	Психологічний прапорець «Важливо».

Продовження таблиці

Назва поля	Тип даних PostgreSQL	Обмеження (Constraints)	Опис та призначення атрибута
is_important	BOOLEAN	DEFAULT FALSE	Психологічний прапорець «Важливо».
is_urgent	BOOLEAN	DEFAULT FALSE	Психологічний прапорець «Терміново».
quadrant	INTEGER	CHECK (quadrant BETWEEN 1 AND 4)	Індекс квадранту Ейзенхауера. Обмеження CHECK гарантує неможливість запису некоректних даних.
status	VARCHAR(50)	DEFAULT 'active'	Стан виконання (можливі значення: active, completed, deleted).

Продовження таблиці

Назва поля	Тип даних PostgreSQL	Обмеження (Constraints)	Опис та призначення атрибута
xp_reward	INTEGER	NOT NULL	Фіксована ціна виконання завдання (залежить від квадранту: 30, 15, 10 або 5 балів).
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Час ініціалізації об'єкта в системі.