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

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка web-застосунку для управління задачами та візуалізацією робочих процесів»

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

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

_____ Ілля М'ЯКОТА
(підпис)

Виконав: здобувач вищої освіти групи ПД-44

Ілля М'ЯКОТА

Керівник: Ірина ЗАМРІЙ
д-р техн. наук, проф.

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

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

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

М'якоті Іллі Андрійовичу

1. Тема кваліфікаційної роботи: «Розробка web-застосунку для управління задачами та візуалізацією робочих процесів».
керівник кваліфікаційної роботи д-р техн. наук, професор Ірина ЗАМРІЙ,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року № 51.

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

3. Вихідні дані до кваліфікаційної роботи: матеріали предметної області управління задачами, документація технологій Flask, SQLAlchemy, PostgreSQL, Docker, Redis, Flask-SocketIO, APScheduler, OWASP, програмний код web-застосунку MiniTrello.

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

1. Огляд та аналіз предметної області та аналогів;
2. Обґрунтування засобів реалізації;
3. Проєктування архітектури, моделі даних і сценаріїв використання;
4. Програмна реалізація MiniTrello;
5. Тестування, безпека даних і демонстрація результатів.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації;
4. Діаграма варіантів використання.
5. Архітектура проєкту;
6. Граф діалогів;
7. Діаграма діяльності життєвого циклу задачі;
8. Екранні форми;
9. Апробація результатів дослідження;

6. Дата видачі завдання«23» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	23.02-28.02.2026	
2	Аналіз предметної області та існуючих аналогів	01.03-07.03.2026	
3	Визначення вимог до MiniTrello	08.03-14.03.2026	
4	Проєктування архітектури, моделі даних та сценаріїв	15.03-28.03.2026	
5	Програмна реалізація web-застосунку	29.03-26.04.2026	
6	Тестування функціональності та механізмів безпеки	27.04-03.05.2026	
7	Оформлення вступу, розділів, висновків і реферату	04.05-10.05.2026	
8	Розробка демонстраційних матеріалів	11.05-17.05.2026	
9	Попередній захист роботи	11.05-22.05.2026	

Здобувач вищої освіти _____ Ілля М’ЯКОТА
(підпис)

Керівник
кваліфікаційної роботи _____ Ірина ЗАМРІЙ
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 88 стор., 18 табл., 24 рис., 32 джерел.

Мета роботи – покращення процесу управління задачами та візуалізацією робочих процесів шляхом розробки web-застосунку.

Об'єкт дослідження – процес управління задачами та візуалізацією робочих процесів у командній роботі.

Предмет дослідження – методи та програмні засоби розробки web-застосунку для управління задачами та візуалізації процесів.

Короткий зміст роботи: у кваліфікаційній роботі розглянуто предметну область управління задачами, принципи Kanban-візуалізації, проблеми планування, контролю строків, призначення відповідальних осіб, комунікації та збереження робочого контексту. Окремо досліджено Trello, Jira, Asana, Notion і ClickUp, визначено їхні обмеження для навчального та локального використання. Сформовано функціональні й нефункціональні вимоги до MiniTrello, обґрунтовано вибір Flask, Jinja2, JavaScript, PostgreSQL, SQLAlchemy, Alembic, Redis, Flask-SocketIO, APScheduler, Docker Compose, pytest і GitHub Actions. Спроєктовано архітектуру web-застосунку, логічну модель даних, сценарії роботи з дошками, задачами, учасниками, ролями, коментарями, вкладеннями, сповіщеннями, спринтами, автоматизаціями та інтеграціями. Описано програмну реалізацію основних модулів, API, механізмів захисту даних, експорту, імпорту та тестування.

Сферою використання застосунку є організація командної роботи, ведення навчальних проєктів і робота малих груп.

КЛЮЧОВІ СЛОВА: WEB-ЗАСТОСУНОК, УПРАВЛІННЯ ЗАДАЧАМИ, KANBAN-ВІЗУАЛІЗАЦІЯ, КОМАНДНА ВЗАЄМОДІЯ, FLASK, POSTGRESQL, DOCKER, REDIS, WEBSOCKET, АВТОМАТИЗАЦІЯ, ТЕСТУВАННЯ.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface

REST – Representational State Transfer

HTTP – Hypertext Transfer Protocol

HTML – HyperText Markup Language

CSS – Cascading Style Sheets

JS – JavaScript

JSON – JavaScript Object Notation

ORM – Object-Relational Mapping

SQL – Structured Query Language

СУБД – система управління базами даних

RBAC – Role-Based Access Control

CSRF – Cross-Site Request Forgery

TOTP – Time-based One-Time Password

OAuth – Open Authorization

CI – Continuous Integration

WebSocket – протокол двосторонньої взаємодії клієнта і сервера

KPI – Key Performance Indicator

CPM – Critical Path Method

ВСТУП

Актуальність: сучасні команди розробників працюють у середовищі, де швидкість обміну інформацією та узгодженість дій безпосередньо впливають на результат виконання проєкту. Команді потрібно бачити статуси, строки, відповідальних осіб, коментарі, прикріплені матеріали й повідомлення, що виникають під час виконання задач. Коли ці відомості розосереджені між таблицями, чатами, локальними файлами та окремими нотатками, поступово втрачається загальна картина проєкту. Учасники витрачають час на пошук контексту, дублюють уже виконану роботу, пропускають дедлайни або передають задачі без потрібних уточнень, через це виникає потреба у web-застосунку, який об'єднує у собі головні якості: планування, візуалізація процесів, комунікацію й контроль виконання в одному робочому середовищі.

Об’єктом дослідження є процес управління задачами та візуалізацією робочих процесів у командній роботі.

Предметом дослідження є методи та програмні засоби розробки web-застосунку для управління задачами та візуалізації процесів.

Метою роботи є покращення процесу управління задачами та візуалізацією робочих процесів шляхом розробки web-застосунку.

Методи дослідження: для виконання роботи застосовано системний аналіз предметної області, за допомогою якого визначено основні вимоги до управління задачами та командної взаємодії. Наявні програмні рішення порівнювалися за функціональністю, зручністю роботи з Kanban-дошками, підтримкою спринтів, аналітики, автоматизацій та інтеграцій. Проєктування MiniTrello виконувалося з урахуванням об'єктно-орієнтованого підходу, модульної архітектури та реляційної моделі даних. Практична частина охопила розробку серверної логіки, інтерфейсу, API, механізмів авторизації, сповіщень і взаємодії з додатковими сервісами. Окремі компоненти перевірено функціональними тестами. Вибір технологій обґрунтовано на основі фахової літератури з інженерії програмного забезпечення, архітектури,

управління проєктами, якості ПЗ та офіційної документації Flask, SQLAlchemy, PostgreSQL, Docker і пов’язаних інструментів програмного стеку [1–11; 24-29].

Для досягнення поставленої мети в бакалаврській роботі виконано такі задачі:

1. Проаналізовано предметну область управління задачами.
2. Проведено порівняльний аналіз аналогів і наявних вебсистем.
3. Визначено функціональні та нефункціональні вимоги до MiniTrello.
4. Обґрунтовано вибір технологічного стеку (Python, Flask, PostgreSQL).
5. Спроектовано архітектуру, модель даних і основні сценарії взаємодії користувачів.
6. Реалізовано вебзастосунок із використанням обраних технологій.
7. Проведено тестування розробленого рішення.

Наукова новизна роботи полягає в поєднанні в одному web-застосунку Kanban-візуалізації, планування спринтів, аналітичних показників і додаткових засобів контролю, зокрема теплової карти активності та методу критичного шляху. У такій конфігурації застосунок розглядається не лише як цифрова дошка задач, а як середовище, що підтримує повніший цикл командної роботи - від створення доручення до аналізу виконання.

Практична значущість результатів полягає в можливості використання розробленого застосунку для організації процесів розробки, ведення навчальних проєктів і моніторингу активності учасників. Окрему практичну цінність мають механізми авторизації, API, сповіщення й автоматизації рутинних дій. Ці компоненти можуть слугувати прикладом побудови web-систем, у яких робочі процеси пов'язані з ролями користувачів, даними задач, інтеграціями та вимогами до захисту інформації.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ СИСТЕМ УПРАВЛІННЯ ЗАДАЧАМИ

1.1 Особливості організації задач у командній роботі

Ефективна робота над програмним або навчальним проєктом неможлива без прозорої системи обліку задач. Для команди недостатньо мати простий перелік доручень, оскільки кожна задача існує не ізольовано, а в межах певного робочого потоку. Вона пов’язана зі строками виконання, відповідальними учасниками, поточним статусом, супровідними матеріалами, історією обговорень і пріоритетами. Якщо для індивідуальної роботи ще може вистачати звичайного списку справ, то в командному середовищі потрібна єдина операційна площина, у якій усі учасники бачать актуальний стан проєкту та розуміють, хто, що і на якому етапі виконує.

На практиці невеликі команди часто починають із найпростіших інструментів: електронних таблиць, нотаток, групових чатів або окремих файлів. Такий підхід працює лише доти, доки кількість задач, виконавців і паралельних обговорень залишається мінімальною. Після масштабування проєкту швидко виникає інформаційний шум: частина рішень губиться в месенджерах, вкладення зберігаються без єдиної структури, дедлайни не синхронізуються зі статусами, а відповідальність за виконання розмивається. У результаті команда втрачає керованість процесу, складніше відстежує завантаження учасників, частіше дублює дії та пізніше реагує на ризики зриву строків.

Основою такого процесу є задача. У професійній системі управління вона має розглядатися не як короткий запис у списку, а як повноцінний робочий артефакт. До її структури доцільно включати назву, деталізований опис очікуваного результату, статус, строк виконання, рівень пріоритету, відповідального виконавця, коментарі, вкладення, мітки, а за потреби – залежності від інших задач.

Саме така модель дозволяє пов’язати окреме доручення з ширшим контекстом проєкту: етапом розробки, поточним навантаженням команди, блокерами та попередніми рішеннями.

Окремої уваги потребує життєвий цикл задачі. У типовому командному процесі він охоплює створення, первинне уточнення вимог, декомпозицію за потреби, призначення виконавця, виконання, внутрішню перевірку, доопрацювання та закриття. На кожному з цих етапів змінюється не лише статус, а й фактичний зміст задачі: уточнюються критерії готовності, додаються матеріали, фіксуються зауваження або змінюється відповідальний учасник. Якщо такі зміни не логуються, команда втрачає трасованість рішень і не може точно визначити, на якому саме етапі виникла затримка або помилка в комунікації.

Командна організація задач також передбачає чітке розмежування ролей і прав доступу. Адміністратор або власник робочого простору керує дошкою, структурою процесу, учасниками та налаштуваннями доступу. Виконавець відповідає за реалізацію конкретних задач і оновлення їхнього стану. Переглядач отримує доступ до інформації без права втручання в критичні елементи проєкту. Такий поділ важливий не лише з погляду зручності, а й з позиції безпеки: користувачі не повинні мати однакові повноваження на редагування, видалення, зміну налаштувань або керування складом команди.

Додаткові елементи задачі – коментарі, вкладення, сповіщення та історія змін – формують робочий контекст, без якого командна взаємодія швидко фрагментується. Коментарі дають змогу зберігати обговорення безпосередньо поруч із відповідною задачею, вкладення фіксують необхідні матеріали або проміжні результати, а сповіщення підтримують оперативну реакцію на зміни. Завдяки цьому команда не розносить роботу між різними каналами, а зберігає основну інформацію в одному середовищі. Це особливо важливо для задач, які проходять кілька ітерацій уточнення, перевірки та доопрацювання, оскільки саме в таких випадках втрачений контекст найчастіше призводить до помилок і повторної роботи.

1.2 Метод Kanban як підхід до візуалізації робочих процесів

Kanban є методом організації потоку робіт, у якому основний акцент робиться на візуалізації станів, керуванні пропускною здатністю процесу та своєчасному виявленні вузьких місць. У межах такого підходу робочий процес подається у вигляді дошки з колонками, кожна з яких відповідає певному етапу виконання: заплановано, у роботі, на перевірці або завершено. Завдяки цьому команда бачить не окремий набір доручень, а цілісну картину руху задач через операційний цикл.

Ключова перевага Kanban полягає у високій прозорості процесу. Учасникам не потрібно вручну зіставляти інформацію з таблиць, чатів, звітів або окремих файлів, щоб зрозуміти поточний стан роботи. Картка задачі виконує роль компактного робочого артефакту, який містить основні атрибути: назву, відповідального виконавця, пріоритет, строк виконання, мітки та поточний статус. Деталізований контекст, зокрема опис, коментарі, вкладення, історія змін і службові події, доцільно виносити в окрему форму перегляду або редагування, щоб не перевантажувати саму дошку другорядною інформацією.

Як визначено в методології Kanban, робочі елементи розглядаються як окремі одиниці роботи, що проходять через визначений потік створення цінності. Особлива увага в цьому підході приділяється контролю незавершеної роботи, тобто Work in Progress. Якщо на одному етапі одночасно накопичується надмірна кількість відкритих задач, процес втрачає передбачуваність: знижується точність оцінювання строків, ускладнюється пріоритизація, а перевантаження окремих виконавців або етапів стає помітним надто пізно [12; 13; 31; 32]. Тому Kanban-дошка має не лише відображати картки у статичному вигляді, а й показувати їх фактичний рух між етапами процесу.

Одним із базових механізмів організації потоку роботи є WIP-обмеження. Воно визначає максимальну кількість задач, які можуть одночасно перебувати на певному етапі або в роботі загалом. Таке обмеження дисциплінує потік, не дозволяє

накопичувати надмірний обсяг незавершених задач і зміщує фокус команди з постійного відкриття нових робіт на доведення вже розпочатих задач до завершення. На практиці це зменшує контекстні перемикання, скорочує черги очікування та допомагає швидше виявляти етапи, де процес фактично блокується. Велика кількість одночасно відкритих задач часто створює лише ілюзію високої продуктивності, тоді як реальна цінність для користувача або замовника не зростає, поки задачі не проходять повний цикл виконання.

Kanban варто відрізняти від Scrum, оскільки ці підходи по-різному організовують роботу команди. Scrum спирається на фіксовані ітерації, планування спринтів, попередньо визначений обсяг робіт і регулярні церемонії. Kanban, навпаки, краще підходить для неперервного потоку задач, де навантаження може змінюватися нерівномірно, а пріоритети потребують гнучкого коригування без очікування завершення чергового спринту. Водночас у межах розробленого програмного рішення ці підходи можуть поєднуватися: Kanban-дошка забезпечує візуальний контроль потоку, а елементи спринтового планування дають змогу групувати задачі за ітераціями та аналізувати виконання протягом установленого часу [12; 14]. Порівняльна характеристика цих підходів наведена у таблиці 1.1.

У web-застосунку реалізація Kanban не обмежується простим відображенням колонок і карток. Кожна дія користувача з карткою, зокрема створення, редагування, зміна статусу або переміщення між колонками, запускає відповідний ланцюг системних операцій. До нього входять перевірка прав доступу, валідація вхідних даних, обробка запиту через API, оновлення записів у базі даних, фіксація активності та, за потреби, надсилання сповіщень іншим учасникам. Отже, інтерфейс дошки виступає не декоративним елементом, а прикладним шаром, через який користувач взаємодіє з бізнес-логікою системи.

Таким чином, Kanban-дошка в контексті розробленого застосунку є не лише способом групування задач за колонками, а повноцінним інструментом операційного контролю. Її функціональність може доповнюватися ролями учасників, журналом активності, системою сповіщень, аналітичними показниками, CPM-розрахунками, heatmap-візуалізацією та KPI-метриками. Завдяки цьому

дошка використовується для контролю строків, відповідальних осіб, завантаження команди, динаміки виконання задач і загального стану робочого процесу, а не залишається лише візуальним компонентом інтерфейсу.

Таблиця 1.1

Порівняння Kanban і Scrum

Критерій	Kanban	Scrum	Використання в MiniTrello
Організація	безперервний потік	ітерації	kanban + модель sprint
Планування	гнучке	перед спринтом	підтримано обидва підходи
Контроль	wip і статуси	velocity і backlog	kpi, overload, sprint analytics
Звітність	потік і прострочення	виконання спринту	cpm, gantt, heatmap
Доцільність	поточна робота	ітераційна розробка	навчальні та командні задачі

1.3 Проблеми планування, контролю та командної взаємодії

Планування задач у командному середовищі швидко втрачає керованість, якщо строки, відповідальні особи, залежності, супровідні матеріали та результати обговорень зберігаються в різних каналах. У такій ситуації керівник проєкту фактично працює не з цілісною моделлю розробки, а з набором розрізнених сигналів: повідомленнями в месенджерах, локальними файлами, особистими нотатками учасників або фрагментарними таблицями. Це ускладнює контроль виконання, погіршує прогнозування строків і підвищує ризик управлінських помилок.

Однією з найбільш типових проблем є неточна оцінка фактичного навантаження. Кількість призначених задач сама по собі не відображає зайнятості фахівця, оскільки кожна задача має різну трудомісткість, рівень пріоритету, технічну складність, залежності та ступінь невизначеності. Для коректного планування потрібно враховувати не лише сам факт призначення доручення, а й його вплив на загальний робочий потік. Наприклад, задача з високим пріоритетом, жорстким дедлайном або блокуючою залежністю потребує більшого фокусу, ніж стандартний поточний запит, навіть якщо формально обидві задачі відображаються як одна одиниця роботи.

Відсутність єдиного робочого простору часто призводить до дублювання задач і втрати актуального контексту. Коли учасники не мають доступу до спільного переліку активних доручень, одна й та сама робота може описуватися кілька разів у різних місцях, а розробники змушені витрачати час на уточнення її реального статусу. Єдина дошка з фільтрами, мітками, прив’язаними виконавцями та прозорою історією змін зменшує такі ризики, оскільки джерело правди щодо стану задачі переноситься з неформальних каналів у структуровану систему.

Якість управління проєктом напряму залежить від прозорості комунікацій. Навіть добре сформульована задача швидко втрачає прикладну цінність, якщо уточнення вимог, результати проміжних перевірок, зауваження або домовленості залишаються в особистих повідомленнях. У такому випадку команда бачить лише фінальний запис, але не розуміє логіку попередніх рішень. Використання коментарів, сповіщень і вкладень безпосередньо в картці задачі дає змогу зберігати робочий контекст поруч із відповідним елементом процесу та робить його доступним для всіх учасників, які мають відповідні права доступу.

Схожа проблема виникає під час роботи з супровідними матеріалами. Якщо технічні документи, макети, скриншоти, файли експорту або проміжні результати розміщені в різних сховищах і чатах, команда витрачає додатковий час на пошук актуальної версії. Крім того, зростає ризик використання застарілих матеріалів. Зберігання вкладень у межах структури задачі зменшує фрагментацію даних,

спрощує доступ до ресурсів і дозволяє пов’язати кожен файл із конкретним етапом роботи.

Окремим бар’єром для контролю є неузгодженість статусної моделі. Якщо учасники по-різному трактують стани «у роботі», «на перевірці» або «завершено», звітність перестає відображати реальний стан процесу. Тому логіка статусів має бути не умовною позначкою в інтерфейсі, а формалізованою моделлю проходження задачі через технологічний цикл. Кожен статус повинен відповідати конкретному етапу роботи, мати зрозуміле значення для всіх користувачів і не допускати подвійного трактування.

Для оптимізації повторюваних бізнес-процесів важливо передбачити автоматизацію рутинних операцій. Насамперед це стосується таких елементів:

- регулярні задачі;
- системні нагадування;
- реакції на наближення або порушення строків виконання.

Такі дії не повинні постійно виконуватися вручну, оскільки це створює зайве адміністративне навантаження і підвищує ймовірність пропусків. Використання шаблонів, правил автоматизації та планувальника подій дозволяє підтримувати стабільний робочий цикл, своєчасно створювати повторювані задачі, надсилати сповіщення та звільняти час команди для складніших інженерних завдань.

Звітність у системі управління задачами має спиратися не на суб’єктивні оцінки, а на фактичні операційні метрики, зокрема на:

- динаміку виконання спринтів;
- дотримання дедлайнів;
- аналіз критичних шляхів;
- завантаження виконавців;
- кількість прострочених або заблокованих задач.

Ручне формування звітів є малоефективним, оскільки такі дані швидко застарівають і часто містять помилки через людський фактор. Натомість аналітичні інструменти, зокрема діаграми Ганта, heatmap-візуалізація активності, KPI-показники та метод критичного шляху, дають змогу оцінювати не лише стан

окремих задач, а й загальну пропускну здатність команди, темп реалізації проєкту та потенційні точки затримки.

Окрему увагу на етапі проєктування потрібно приділити інформаційній безпеці. Система працює з персональними даними користувачів, токенами інтеграцій, файлами, історією активності та внутрішньою проєктною документацією. За відсутності належного контролю доступу виникають ризики несанкціонованого перегляду, зміни або видалення даних. Тому під час розробки доцільно орієнтуватися на практики, що відповідають рекомендаціям OWASP Top 10: надійну автентифікацію, рольове розмежування доступу, перевірку прав на рівні об’єктів, захист сесій, валідацію вхідних даних і криптографічний захист критичних інформаційних потоків [15–18]. Узагальнений перелік виявлених проблем командної взаємодії та відповідні інструментальні методи їх вирішення в межах MiniTrello представлено у таблиці 1.2.

Таблиця 1.2

Проблеми командної роботи та способи їх вирішення у web-застосунку

Проблема	Прояв	Рішення в MiniTrello
Навантаження	окремий виконавець має надмірну кількість задач або задачі з високим пріоритетом	team overload і kpi
Дедлайни	строки виконання губляться або не пов’язуються зі статусом задачі	due_date, reminders, calendar
Дублювання	одна й та сама робота може бути створена кілька разів	єдина дошка, фільтри та видимі статуси
Файли	матеріали зберігаються в чатах, локальних папках або сторонніх сервісах	taskattachment
Статуси	немає єдиного процесу виконання задач	kanban-колонки

Продовження таблиці 1.2

Проблеми командної роботи та способи їх вирішення у web-застосунку

Проблема	Прояв	Рішення в MiniTrello
Комунікація	рішення залишаються в приватних повідомленнях або окремих обговореннях	taskcomment і notification
Звітність	ручні звіти швидко застарівають і не відображають фактичний стан роботи	cpm, gantt, heatmap
Безпека	доступ сторонніх осіб до задач, файлів або токенів інтеграцій	rbac, csrf, tokens

1.4 Аналіз існуючих програмних аналогів

Для коректного формування вимог до вебзастосунку було виконано порівняльний аналіз поширених систем управління задачами та проєктною роботою: Trello, Jira, Asana, Notion і ClickUp. Ці рішення фактично займають різні ніші одного класу програмного забезпечення: від легких Kanban-дошок для невеликих команд до комплексних платформ із розвиненою моделлю процесів, автоматизацією, аналітикою, інтеграціями та розмежуванням доступу.

Порівняння виконувалося не лише за наявністю окремих функцій, а й за тим, наскільки ці функції придатні для реальної командної експлуатації. Зокрема, враховувалися такі критерії:

- підтримка Kanban-підходу та візуального керування потоком задач;
- гнучкість налаштування робочих процесів, статусів, міток і представлень;
- наявність механізмів автоматизації рутинних операцій;
- інструменти аналітики, звітності та контролю дедлайнів;
- рольова модель доступу та керування правами користувачів;
- підтримка інтеграцій зі сторонніми сервісами;
- зручність роботи з командною комунікацією, вкладеннями та історією змін.

Це дало змогу оцінити не лише сильні сторони наявних платформ, а й їхні практичні обмеження: надмірну складність конфігурації, залежність від платних тарифів, недостатню прозорість кастомізації або перевантаженість інтерфейсу для невеликих команд. Отримані результати були використані як орієнтир під час проєктування власної системи, зокрема під час визначення функціонального ядра, моделі доступу, логіки Kanban-дошок, автоматизацій, аналітичних модулів та інтеграційного шару.

1.4.1 Trello

Trello – один із найвідоміших SaaS-продуктів для візуального менеджменту. В основі лежать дошки, списки та картки, що максимально наближено до класичної методології Kanban. Користувач переміщує картки між вертикальними колонками. Колонки зазвичай символізують послідовні етапи робіт: «Черга задач», «В роботі», «На тестуванні», «Завершено». Таке візуальне подання дозволяє команді миттєво оцінити стан проєкту без складних текстових звітів, як показано на рисунку 1.1.

Кожна картка – це універсальний контейнер. Вона містить заголовок, розгорнутий опис задачі, чеклісти для декомпозиції великих завдань, терміни виконання, перелік відповідальних учасників і файли. Командна взаємодія відбувається через контекстні коментарі прямо в картці. Додаткову функціональність дають модулі Power-Ups: календар, поля для голосування, сторонні сервіси. Інструмент Butler на основі правил (triggers) автоматизує переміщення карток або створення типових завдань за розкладом.

Головні переваги сервісу:

- візуалізація та зрозумілість;
- інтуїтивно зрозумілий інтерфейс;
- гнучкість та універсальність;
- доступність (безкоштовна версія);
- кросплатформеність;
- автоматизація (butler).

Однак для складних інженерних проєктів з розгалуженою ієрархією Trello часто не вистачає.

Недоліки:

- складність при великих масштабах;
- обмежений функціонал безкоштовної версії [19];
- відсутність вбудованої аналітики;
- проста ієрархія;
- залежність від інтернету;
- немає вбудованого чату;
- безпека.

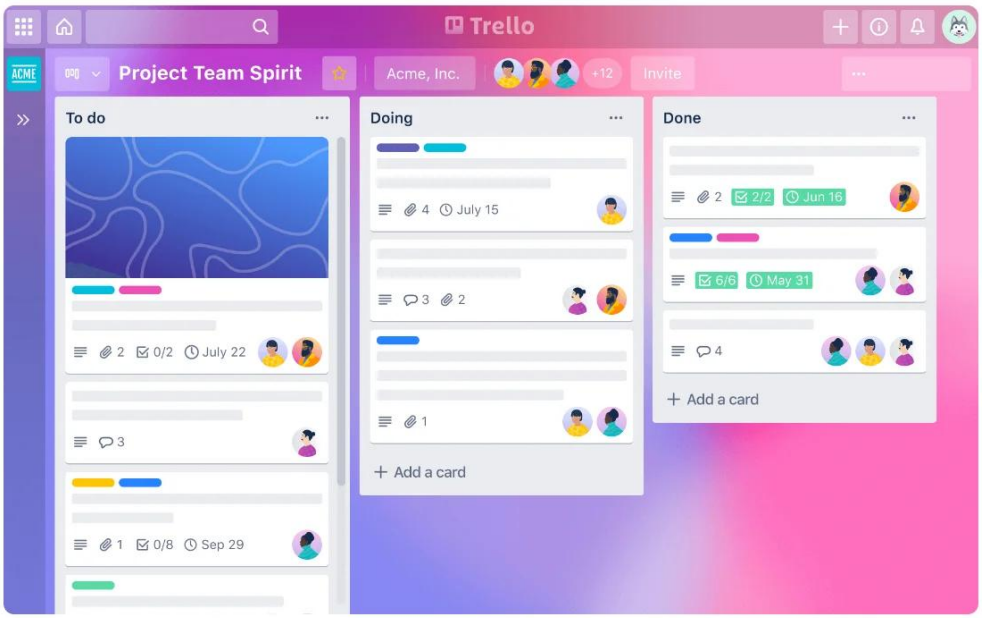


Рис. 1.1 Інтерфейс Trello

1.4.2 Jira

Jira компанії Atlassian належить до найбільш поширених корпоративних систем керування розробкою програмного забезпечення. Її основна сфера застосування – середні та великі команди, які працюють за Agile-підходами, зокрема Scrum і Kanban. На відміну від легких task-tracking інструментів, Jira підтримує розгалужену ієрархію робочих сутностей: Epic, User Story, Task, Bug, Sub-task. І це в свою змогу дає можливість декомпозувати вимоги від рівня великої

функціональної області до конкретних інженерних задач і контролювати виконання на різних рівнях деталізації.

На рисунку 1.2 зображено одну з ключових переваг Jira, а саме - гнучке налаштування workflow, формалізованого життєвого циклу задачі. Система дозволяє задавати власні статуси, переходи між ними, умови виконання переходів, валідатори, post-functions і permission scheme для різних ролей користувачів. На практиці це означає, що команда може не просто переміщувати задачу між колонками, а будувати повноцінну процесну модель із перевіркою обов'язкових полів, обмеженням доступу до окремих дій і автоматизованою зміною службових атрибутів під час переходу задачі між станами.

Аналітичний модуль Jira також орієнтований на професійне управління розробкою. Система підтримує burndown chart, velocity chart, cumulative flow diagram, sprint report та інші інструменти, які використовуються для оцінювання темпу команди, стабільності потоку задач, прогнозування строків релізу та виявлення bottleneck-ділянок у процесі. Для керівника проєкту або Scrum Master такі метрики мають практичну цінність, оскільки дозволяють аналізувати не лише факт виконання задач, а й динаміку накопичення незавершеної роботи, перевантаження окремих етапів і відхилення від планового ритму розробки [20].

Сильними сторонами сервісу є професійний інструментарій для ітераційної розробки, один із найкращих на ринку механізмів баг-трекінгу та глибока інтеграція з репозиторіями коду, такими як GitHub, GitLab або Bitbucket. Водночас системі притаманний ряд недоліків, зокрема високий поріг входження для нових користувачів, значна перевантаженість інтерфейсу другорядними елементами та зниження швидкодії при роботі з великими масивами даних. Для невеликих стартапів чи навчальних груп такий інструментарій часто надлишковий, а вартість ліцензій для великої кількості користувачів – економічно недоцільна.

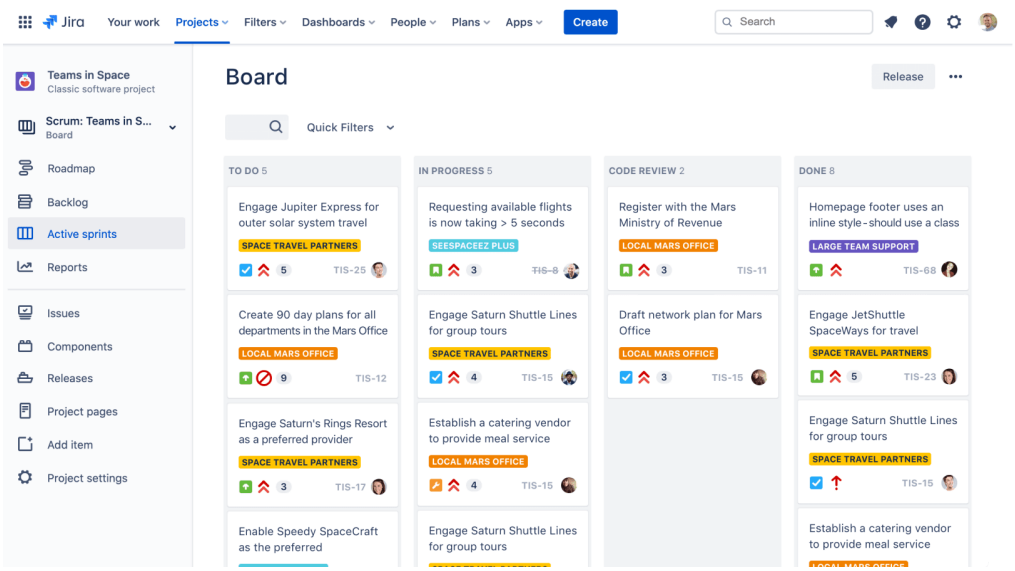


Рис. 1.2 Інтерфейс Jira

1.4.3 Asana

Asana є комплексною платформою для координації командної роботи, орієнтованою не лише на облік задач, а й на управління проєктними портфелями, залежностями, строками та завантаженням виконавців. На відміну від більш вузьких Kanban-орієнтованих сервісів, Asana пропонує користувачу кілька режимів представлення робочих даних, що дозволяє адаптувати інтерфейс під різні сценарії планування.

До основних форматів подання, що наведені на рисунку 1.3, належать: класичний список задач, kanban-дошка, календарне представлення та timeline-подання з часовою шкалою. Таке поєднання режимів зручне для команд, у яких різні учасники працюють із проєктом на різних рівнях деталізації. Наприклад, виконавцю достатньо списку або дошки з поточними задачами, тоді як керівнику проєкту важливі строки, залежності, критичні етапи та загальна картина виконання.

Функціональне ядро Asana зміщене в бік тактичного й стратегічного планування. Інструмент Timeline фактично реалізує логіку спрощеної діаграми Ганта: користувач може візуально розміщувати задачі на часовій шкалі, задавати залежності між ними, контролювати послідовність виконання та бачити потенційні конфлікти за строками. Це особливо важливо для проєктів, де окремі роботи мають

жорстку технологічну послідовність, а запуск наступного етапу неможливий без завершення попереднього.

Окремої уваги заслуговує модуль Workload, який використовується для аналізу завантаження учасників команди. Він дає змогу оцінювати розподіл задач між виконавцями, виявляти перевантажених фахівців і своєчасно перерозподіляти роботу. Для менеджера це не просто зручна візуалізація, а інструмент операційного контролю, який допомагає зменшити ризик зриву дедлайнів через нерівномірне навантаження або приховані вузькі місця в команді [21].

Якщо говорити про основні переваги сервісу, то одразу виділяється приємний та зрозумілий інтерфейс, висока якість візуалізації через таймлайни й діаграми Ганта, а також зручне управління навантаженням команди (workload). Водночас платформі притаманні певні недоліки, зокрема висока ціна платних тарифів, суворі обмеження безкоштовної версії та неможливість призначення однієї задачі на кількох виконавців, що є частиною філософії сервісу. Крім того, для невеликих навчальних або стартап-проєктів платформа може бути функціонально надлишковою: велика кількість режимів, рівнів налаштування та управлінських сутностей ускладнює первинне освоєння і створює додаткове когнітивне навантаження для користувачів, яким потрібен простіший інструмент для швидкої організації задач.

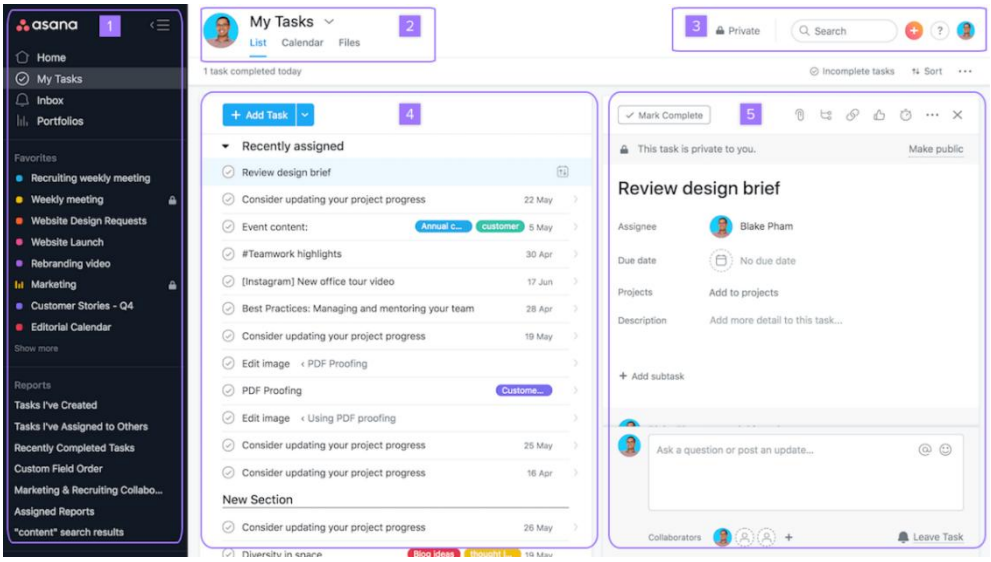


Рис. 1.3 Інтерфейс Asana

1.4.4 Notion

Notion можна розглядати як універсальне інформаційне середовище, що поєднує можливості текстового редактора, баз знань, реляційних таблиць і базових інструментів проєктного менеджменту. Його архітектурна особливість полягає у блочній моделі побудови сторінок: будь-який елемент документа, таблиці, списку, посилання або вкладеного об’єкта подається як окремий блок, який користувач може комбінувати відповідно до власної логіки роботи. Завдяки цьому Notion не нав’язує жорстку структуру процесу, а дає змогу самостійно формувати робочий простір.

Управління задачами реалізується переважно через бази даних, тобто один і той самий набір записів може відображатися у вигляді таблиці, Kanban-дошки, календаря, галереї або списку. Такий механізм представлень дає змогу змінювати спосіб роботи з даними без дублювання самих задач. Як видно на рисунку 1.4, команда може вести загальний backlog у табличному вигляді, переглядати активні задачі на дошці, а дедлайни контролювати через календарне подання.

Суттєвою перевагою Notion є можливість безпосередньо поєднувати оперативні задачі з технічною документацією, специфікаціями, базою знань і результатами обговорень. Кожна задача фактично є окремою сторінкою, усередині якої можна розміщувати фрагменти коду, чеклісти, посилання на пов’язані документи, результати тестування, макети або службові нотатки. Такий підхід зручний для команд, яким важливо зберігати не лише статус задачі, а й увесь контекст її виконання в одному місці.

Окремо слід відзначити гнучкість атрибутної моделі. Користувачі можуть самостійно створювати властивості записів: статуси, пріоритети, відповідальних осіб, дати, зв’язки між базами, формули, теги та інші службові поля. Це дозволяє адаптувати Notion під нестандартні процеси без потреби в окремій розробці або складному адмініструванні [22].

Сервіс відзначається своєю універсальністю, що дозволяє поєднувати документи, бази даних і задачі в одному вікні, повна свобода в дизайні сторінок, а також ефективністю інструментів для структурування інформації. Серед недоліків

платформи слід виділити значні часові витрати на первинне налаштування системи під конкретні потреби, слабкі механізми автоматизації та сповіщень про дедлайни, а також певні труднощі при управлінні великою кількістю дрібних завдань. Разом з тим за відсутності чітких внутрішніх правил велика свобода налаштувань може призвести до хаотичної структури: різні команди починають по-різному оформлювати задачі, статуси, сторінки та зв'язки між ними. У великих проєктах це ускладнює супровід системи й знижує передбачуваність робочого процесу.

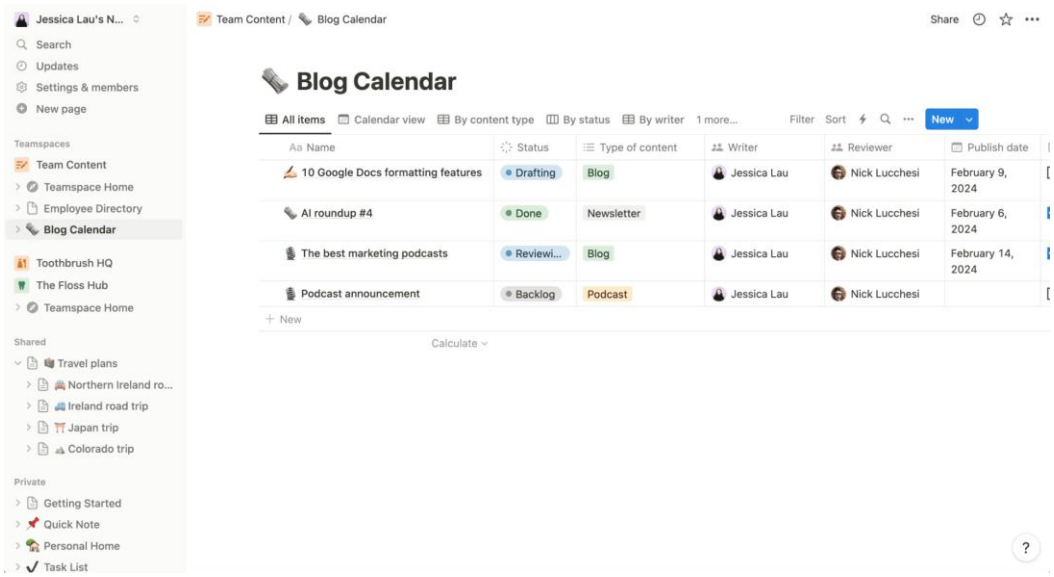


Рис. 1.4 Інтерфейс Notion

1.4.5 ClickUp

ClickUp позиціонується як комплексна платформа для організації командної роботи, що прагне замінити кілька окремих сервісів, як показано на рисунку 1.5. Вона поєднує task management, документи, цілі, чати, дашборди, календарі, автоматизації та інструменти звітності. На відміну від простих Kanban-рішень, ClickUp має багаторівневу ієрархію організації даних: Workspaces, Spaces, Folders, Lists і Tasks. Така структура дає змогу масштабувати систему під великі проєкти, кілька команд або різні напрями роботи в межах однієї організації.

Платформа підтримує значну кількість представлень даних. Серед них можна виділити Kanban-дошки, списки, календарі, діаграми Ганта, mind maps, workload-подання та дашборди для моніторингу KPI. Завдяки цьому один і той самий набір

задач можна аналізувати з різних управлінських позицій: як поточний backlog, як часовий план, як навантаження виконавців або як джерело метрик для керівника проєкту.

Сильною стороною ClickUp є модуль автоматизації, який дозволяє налаштовувати правила за принципом «подія – умова – дія» без написання коду. Приміром, після зміни статусу задачі система може автоматично призначити відповідального, створити пов’язану задачу в іншому списку, змінити пріоритет, надіслати сповіщення або оновити службове поле. Для команд із повторюваними процесами це дає змогу скоротити ручну операційну роботу й зменшити кількість помилок, пов’язаних із людським фактором [23].

Одними із головних переваг сервісу можна виділити велику кількість доступних функцій навіть у безкоштовній версії, висока гнучкість у налаштуванні видів відображення даних, а також наявність вбудованих інструментів для роботи з документами, чатами та трекінгу часу. Однак системі притаманні певні недоліки, зокрема значна складність опанування через перевантаженість інтерфейсу елементами керування, періодичні затримки у завантаженні сторінок та менш зручний мобільний додаток порівняно з основними конкурентами.

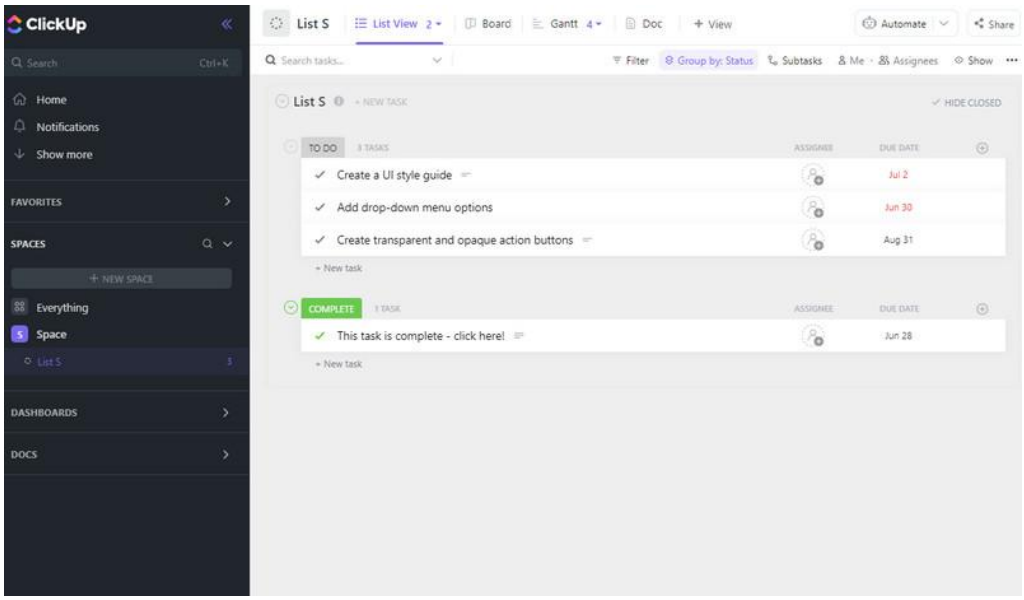


Рис. 1.5 Інтерфейс ClickUp

1.4.6 Порівняння аналогів із MiniTrello

Проведений мною аналіз показав, що ринок систем управління задачами охоплює широкий спектр рішень: від відносно легких Kanban-інструментів на кшталт Trello до багаторівневих корпоративних платформ типу Jira, Asana або ClickUp. Кожен із продуктів має власну цільову аудиторію, набір управлінських сценаріїв і рівень функціональної складності.

У межах бакалаврської роботи важливо не лише проаналізувати готові інструменти з позиції користувача, а й відтворити ключові інженерні рішення, які лежать в основі подібних систем. Саме тому MiniTrello розглядається не як спрощена копія комерційних сервісів, а як навчально-прикладна реалізація повного циклу розробки task management системи. У проєкті було поєднано найбільш практичні підходи, характерні для сучасних аналогів: візуальне керування потоком задач через Kanban-дошки, ітераційне планування за допомогою спринтів, рольову модель доступу, систему коментарів і сповіщень, а також аналітичні інструменти для оцінювання активності та завантаження команди.

Функціональне ядро розробленого застосунку сформовано навколо задачі як основної доменної сутності. На відміну від багатьох універсальних платформ, де управління задачами є лише одним із модулів великої екосистеми, у мене ця логіка винесена в центр архітектури. Kanban-дошка використовується не лише як візуальний шар, а як інтерфейс до бізнес-логіки: зміна статусу картки, призначення виконавця, додавання коментаря або оновлення строку виконання супроводжуються перевіркою прав доступу, валідацією даних, оновленням записів у PostgreSQL і, за потреби, фіксацією події в журналі активності.

Окремою перевагою є прозорість серверної та інформаційної архітектури. Під час розробки було спроєктовано нормалізовану реляційну базу даних на PostgreSQL, реалізовано серверну логіку на мові програмування Python із використанням фреймворку Flask, побудовано RESTful API для взаємодії клієнтської частини з backend-рівнем, а також передбачено механізми автентифікації, авторизації та контролю доступу до об’єктів.

Аналітичні можливості орієнтовані на практичні потреби командного планування. Замість обмеження звичайним списком задач система може використовувати теплові карти активності, показники завантаженості, KPI-метрики, аналіз строків виконання та інші інструменти, які допомагають оцінювати не тільки факт завершення окремих задач, а й загальну динаміку робочого процесу. У цьому аспекті власне рішення поєднує простоту базової Kanban-моделі з елементами більш зрілої проєктної аналітики.

Підсумовуючи, порівняльний аналіз Trello, Jira, Asana, Notion і ClickUp став практичним напрямком для визначення функціональних меж власної системи. Орієнтація на кращі практики індустрії дозволила інтегрувати в систему найбільш релевантні інженерні ідеї, серед яких особливий акцент зроблено на наочності kanban-дошки, функціональності спринтів, чіткій прив’язці задач до виконавців, моніторингу дедлайнів, модульній аналітиці та надійному рольовому розмежуванні доступу. Зведена порівняльна характеристика аналогів та розробленої системи наведена у таблиці 1.3.

Таблиця 1.3

Порівняльна характеристика програмних аналогів і розробленого web-застосунку

Критерій	Trello	Jira	Asana	Notion	ClickUp	MiniTrello
Організація задач	картки, списки, дошки	scrum/ kanban-проєкти	задачі та проєкти	сторінки, бази даних, подання	простори, списки, задачі	дошки, колонки, задачі
Kanban-дошки	основний режим роботи	налаштовуються через робочі процеси	доступні як один із режимів	через подання бази даних	доступні як окреме подання	основний робочий простір
Коментарі та вкладення	є в картках задач	є в задачах і підзадачах	є в задачах і проєктах	є в сторінках і записах	є в задачах	є в задачах
Спринти	через шаблони або power-ups	вбудована scrum-логіка	Через параметры проекта	через ручне налаштування баз	доступні у робочих процесах	реалізована sprint-модель

Продовження таблиці 1.3

Порівняльна характеристика програмних аналогів і розробленого web-застосунку

Критерій	Trello	Jira	Asana	Notion	ClickUp	MiniTrello
Аналітика та звітність	через power-ups або зовнішні сервіси	звіти, burndown, velocity	панелі звітності	через таблиці, фільтри й подання	аналітичні панелі та звіти	kpi, cpm, діаграма ґанта, теплова карта
Автоматизація процесів	butler-правила	правила автоматизації	правила та інтеграції	через шаблони та інтеграції	правила автоматизації	rule engine
Ролі та доступ	залежить від робочого простору й тарифу	детальна модель прав	ролі на рівні проєкту	ролі на рівні робочого простору	ролі на рівні простору й проєкту	rbac для дошок
Інтеграції	power-ups	marketplace	інтеграції з робочими сервісами	інтеграції робочого простору	інтеграції з зовнішніми сервісами	telegram, slack, google calendar

1.5 Визначення вимог до застосунку

Вимоги до вебзастосунку сформовано на основі аналізу предметної області, порівняння наявних систем управління задачами та практичних сценаріїв командної розробки. На цьому етапі важливо було визначити не лише перелік функцій, які має бачити кінцевий користувач, а й ті системні властивості, що забезпечують стабільність, керованість і придатність рішення до подальшого розвитку [5; 6].

Застосунок має підтримувати повний цикл роботи з проєктами та виконувати роль централізованого операційного середовища, у якому поєднуються task management, командна комунікація, контроль виконання та технічна аналітика.

Функціональні вимоги визначають поведінку системи з погляду користувача та залежать від його ролі в межах робочого простору або окремої дошки. До таких

ролей належать власник, адміністратор, учасник команди та користувач з обмеженими правами перегляду. Для кожної ролі мають бути передбачені відповідні сценарії доступу, допустимі операції та обмеження на зміну критичних даних, як показано у таблиці 1.4.

До основних функціональних вимог належать:

- реєстрація, вхід і керування профілем користувача;
- творення, редагування, видалення та налаштування дощок;
- створення, редагування, переміщення, архівування й видалення задач;
- призначення виконавців, пріоритетів і строків виконання;
- керування статусами задач через kanban-дошку;
- коментарі, вкладення, історія змін і запрошення учасників;
- розмежування доступу між власником, адміністратором і учасниками;
- сповіщення про зміни, дедлайни та важливі події;
- підтримка спринтів, повторюваних задач і автоматизацій;
- формування аналітики, kpi, звітів та експорту даних;
- налаштування зовнішніх інтеграцій, сповіщень і календарних сервісів.

Нефункціональні вимоги описують не окремі дії користувача, а якісні характеристики системи. Саме вони визначають, наскільки застосунок буде надійним, безпечним, масштабованим і зручним для супроводу після первинної реалізації. Для такого класу програмного забезпечення особливе значення мають архітектурна модульність, контроль доступу, стабільність роботи з даними, передбачувана поведінка API та можливість автоматизованого тестування ключових компонентів.

У таблиці 1.5 представлено ключові нефункціональні вимоги:

- контроль доступу через автентифікацію, авторизацію та ролі;
- збереження даних у реляційній базі з підтримкою зв’язків і транзакцій;
- модульна структура серверної частини, API та клієнтських сценаріїв;
- зрозумілий інтерфейс: створення задачі – до 3 кроків, зміна статусу – 1 drag-and-drop, доступ до функцій – до 2 переходів;

- стабільна робота: відповідь сторінок і api – до 1 с, збереження змін – до 1 с, команда – до 10 користувачів на дошці;
- можливість розширення функцій через сервісний шар;
- підтримка фонових процесів для задач, автоматизацій і сповіщень;
- коректна локалізація інтерфейсу кількома мовами: українською, англійською;
- захист від несанкціонованого доступу, CSRF, помилок введення та витоку даних;

Таблиця 1.4

Функціональні вимоги до web-застосунку

№	Функціональна вимога	Конкретизація для MiniTrello
1	Автентифікація користувача	реєстрація, вхід, вихід із системи, керування сесією, захист форм через csrf
2	Керування дошками	створення, перегляд, редагування, видалення та архівування дошок
3	Учасники та ролі	запрошення учасників, ролі власника, адміністратора й учасника, перевірка доступу
4	Задачі Kanban-дошки	назва, опис, статус, дедлайн, пріоритет, виконавець, переміщення між колонками
5	Контекст задачі	коментарі, чеклисти, мітки, вкладення та залежності між задачами
6	Сповіщення й повідомлення	системні сповіщення, особисті повідомлення, центр повідомлень
7	Планування	спринти, повторювані задачі, календарні нагадування
8	Аналітика	kpi, cpm, gantt-діаграма, heatmap, проблемні та прострочені задачі
10	Інтеграції	telegram, slack, google calendar, web push, email-сповіщення

Продовження табилиці 1.4

Функціональні вимоги до web-застосунку

№	Функціональна вимога	Конкретизація для MiniTrello
11	Експорт та імпорт	експорт у csv, xlsx, pdf, робота з json-операціями
12	Локалізація	українська, англійська локалізація інтерфейсу

Таблиця 1.5

Нефункціональні вимоги до web-застосунку

№	Нефункціональна вимога	Конкретизація
1	Модульність	розділення моделей, сервісів, представлень, api-рівня, шаблонів і статичних файлів
2	Безпека	хешування паролів, csrf-захист, rbac, rate limiting, 2fa/totp, oauth
3	Розгортання	dockerfile, docker compose, postgresql і redis для локального та серверного запуску
4	Тестованість	pytest, fixtures, api-тести, тести інтеграцій, безпеки та основних сценаріїв
5	Робота в реальному часі	socket.io з можливістю використання redis message queue
6	Фонові задачі	apscheduler, блокування jobs, повторювані задачі та планові перевірки
7	Супроводжуваність	alembic-міграції, структуровані сервіси, окремі модулі, підтримка сі
8	Обмеження файлів	контроль розміру завантажень і перевірка доступу до вкладень

Окрему групу становлять вимоги до інтерфейсу та UX-складової. Візуалізація Kanban-дошки повинна бути ергономічною, що дозволяє користувачеві орієнтуватися в потоці задач без додаткового інструктажу, а це, в свою чергу, передбачає впровадження зручних форм введення даних, гнучких фільтрів, підтримку тем оформлення та адаптивність сторінок для коректного відображення на пристроях із різною роздільною здатністю екрана.

Критерії до командної взаємодії базуються на моделі розмежування прав, наприклад: власник або адміністратор отримує повний інструментарій для конфігурування робочого простору, тоді як права звичайного учасника обмежені операційними діями в межах доступних йому задач. Такий спосіб гарантує цілісність структури проєкту та захист від випадкового спотворення налаштувань.

Інтеграційні вимоги охоплюють взаємодію з екосистемою сторонніх сервісів, таких як Slack, Telegram та Google Calendar. Система повинна безпечно зберігати токени, коректно обробляти помилки зв'язку та наочно показувати статус синхронізації. Це розширює можливості сповіщення та дозволяє автоматизувати нагадування про критичні дедлайни.

Щодо забезпечення безпеки, то вона є пріоритетною вимогою, тому що реалізується як на рівні клієнтського інтерфейсу, так і в серверній логіці API. Зокрема, комплекс захисних заходів передбачає впровадження механізмів захищеної автентифікації та контролю сесій, протидію типовим вразливостям (насамперед CSRF), використання двофакторної автентифікації (2FA), і заодно встановлення лімітів на частоту запитів для запобігання потенційним зловживанням.

Переходячі до вимог тестування, вони, зі свого боку, передбачають верифікацію всіх критичних сценаріїв: від реєстрації нового користувача до складних процесів експорту даних та спрацювання правил автоматизації. За рахунок такого комплексного підходу перевірка працездатності дозволяє переконатися у стабільності взаємодії між окремими модулями системи та відповідності фінального продукту заявленим критеріям якості.

2 ЗАСОБИ РЕАЛІЗАЦІЇ

2.1 Архітектурний підхід до розробки web-застосунку

Архітектурне рішення, представлене на рисунку 2.1, базується на багатошаровій структурі з чітким поділом між рівнем представлення, API-інтерфейсом, сервісним шаром, моделями даних і модулями фонових завдань. Опис кожного рівня архітектури наведено у таблиці 2.1.

Розподіл зон відповідальності такий:

- шаблони Jinja2 – генерують сторінки;
- JavaScript-модулі – забезпечують динамічну взаємодію на стороні клієнта;
- REST API – обробляє вхідні запити;
- сервісний шар – акумулює складну бізнес-логіку, виокремлюючи її з логіки маршрутизації.

Серверний компонент реалізовано на базі фреймворку Flask. Він координує обробку вебсторінок, авторизацію користувачів, валідацію форм та контроль прав доступу. Візуальна частина поєднує серверний рендеринг (Jinja2) і клієнтську інтерактивність (CSS, JavaScript). Цей стек дозволив реалізувати складні елементи інтерфейсу: динамічні Kanban-дошки, модальні вікна керування задачами, фільтри та аналітичні панелі, зберігаючи при цьому швидкість завантаження сторінок.

Архітектура враховує не лише стандартні HTTP-запити, а й події в реальному часі та регламентні фонові операції. Для синхронізації даних без перезавантаження сторінок інтегровано Flask-SocketIO. Планувальник APScheduler відповідає за автоматизацію та періодичні сценарії. Інфраструктурну підтримку (лімітування запитів, черги повідомлень, блокування паралельних процесів) забезпечує Redis. Для надійного зберігання структурованої інформації я вибрав СУБД PostgreSQL.

Поділ логіки на маршрути, сервіси та моделі спрощує супровід проєкту: маршрути виконують роль диспетчерів, моделі описують схему даних, а сервіси

реалізують алгоритми автоматизації та інтеграції. Така архітектура зменшує взаємозв'язок елементів і створює умови для легкого масштабування.

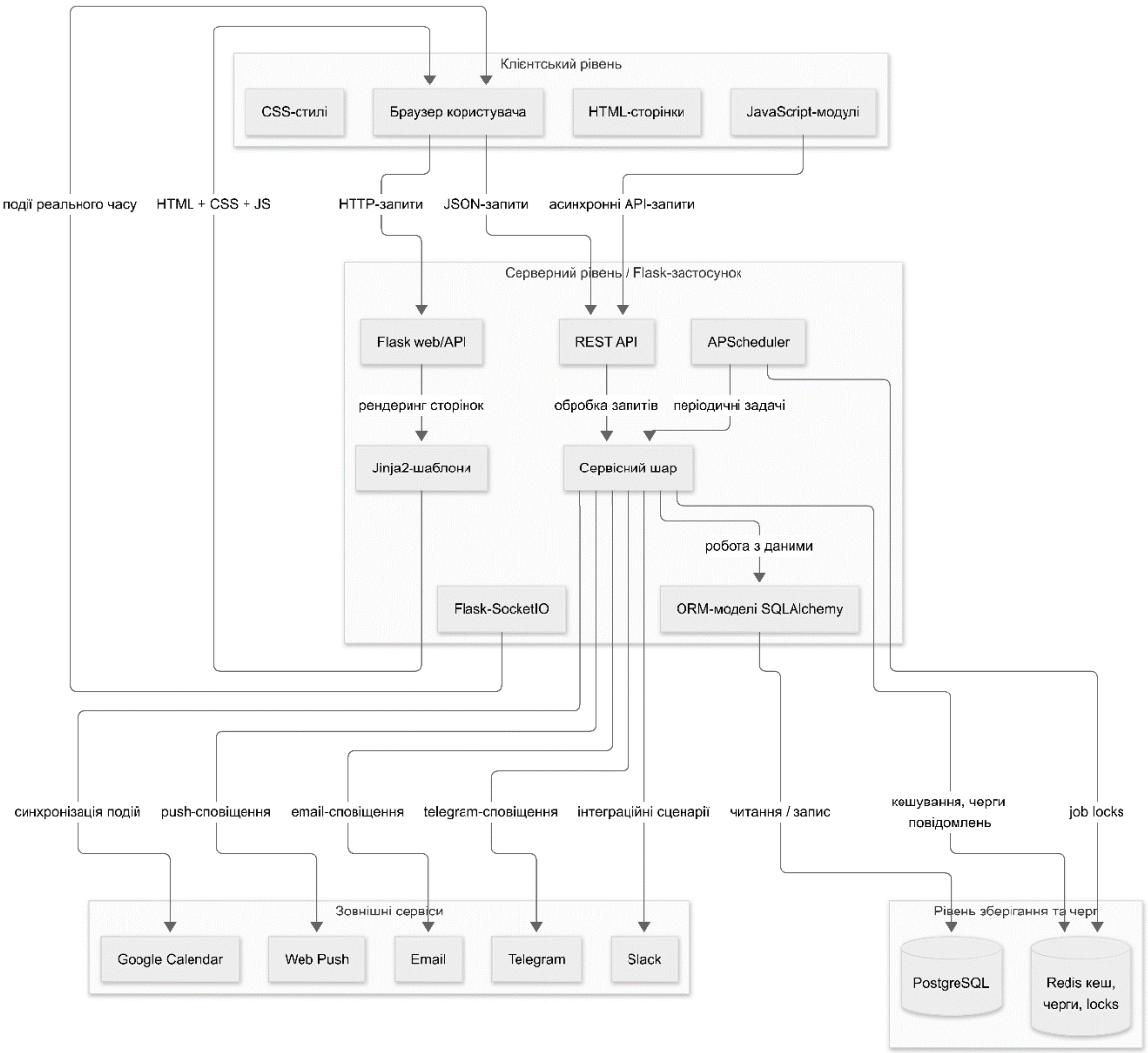


Рис. 2.1 Загальна архітектура MiniTrello

Я відмовився від архітектури Single Page Application (SPA) на користь серверного рендерингу (SSR) у поєднанні з ізольованими JavaScript-модулями. Це допомогло спростити структуру системи та мінімізувати кількість зовнішніх залежностей. Такий підхід дозволяє повноцінно реалізувати асинхронну взаємодію з API та динамічне оновлення елементів (наприклад, механізм перетягування карток), уникаючи при цьому надмірного ускладнення логіки маршрутизації на стороні клієнта.

Застосування патернів «Application Factory» та «Blueprints» забезпечило систематизацію коду у вигляді незалежних модулів. Це критично важливо для системи, що містить різнопланову функціональність: від аналітичних модулів до механізмів захисту даних. Модульна структура значно спрощує тестування, підключення нових розширень та адаптацію застосунку до різних конфігурацій середовища.

Отже, обране архітектурне рішення забезпечує баланс між гнучкістю розробки та стабільністю функціонування, інтегруючи сучасні методи обробки даних у реальному часі та надійні інструменти керування фоновими процесами.

Таблиця 2.1

Рівні архітектури web-застосунку

Рівень	Призначення	Основні компоненти
Web-рівень	формування сторінок і відображення інтерфейсу	jinja2, templates, css
Клієнтська логіка	інтерактивні дії на сторінках	javascript-модулі
API-рівень	обробка json-запитів від клієнтської частини	rest api, api-маршрути, контролери запитів
Сервісний шар	виконання бізнес-логіки та допоміжних операцій	сервіси обробки задач, сповіщень, автоматизацій та інтеграцій
Рівень моделей	опис сутностей предметної області	orm-моделі
Рівень даних	збереження користувачів, дошок, задач та пов’язаних даних	postgresql
Події реального часу	оновлення сповіщень і окремих дій без перезавантаження сторінки	flask-socketio, redis

Продовження таблиці 2.1

Рівні архітектури web-застосунку

Рівень	Призначення	Основні компоненти
Фонові задачі	виконання періодичних операцій і автоматизацій	apscheduler
Інтеграції	взаємодія із зовнішніми сервісами	telegram, slack, google calendar, web push

2.2 Python та Flask як основа серверної частини

Мову Python і фреймворк Flask для серверної логіки вибрано через їхню модульність, лаконічність і розвинену екосистему розширень. Flask дозволяє ефективно поєднувати генерацію HTML-контенту на стороні сервера з обслуговуванням REST API запитів. Для систем із високим рівнем інтерактивності це одне з оптимальних рішень [7].

Логіка застосунку не монолітна: вона розподілена між спеціалізованими модулями (керування задачами, аналітика, інтеграції зі Slack/Telegram, Web Push сповіщення). Таке розмежування знижує ризик регресійних помилок, коли вносяться зміни в конкретний блок системи.

Механізм «Application Factory» дозволяє динамічно створювати екземпляр застосунку із заданими параметрами конфігурації. Він спрощує управління налаштуваннями бази даних, параметрами безпеки та лімітами запитів і забезпечує гнучкість при переході між різними сценаріями використання.

Для структурування маршрутів використано механізм Blueprints, який групує ендпоінти за функціональним призначенням (наприклад, /api/, /auth/, /board/). Таке групування спрощує навігацію кодом і дозволяє розробникам одночасно працювати над незалежними модулями.

Систему автентифікації та керування сесіями реалізовано за допомогою Flask-Login. Контроль доступу не обмежується лише перевіркою факту входу.

Впроваджена багаторівнева рольова модель (Owner, Admin, Manager, Member, Viewer) регламентує дозволені дії користувача в межах конкретної дошки.

Захист даних на рівні форм і запобігання атакам CSRF забезпечує розширення Flask-WTF. Перевірка токенів виконується для кожного запиту, що змінює стан системи. Це один із обов'язкових елементів контуру безпеки вебзастосунку.

Впровадження ORM-шару Flask-SQLAlchemy разом з інструментом Flask-Migrate дозволило перенести опис сутностей (користувачі, спринти, вкладення) у площину об'єктно-орієнтованого програмування та автоматизувати оновлення структури таблиць [8; 9].

Функціональність розширена спеціалізованими інструментами, перелік та призначення яких у серверній частині наведено у таблиці 2.2.

Використанні інструменти:

- Flask-SocketIO – для обміну даними в реальному часі;
- Flask-Mail – для розсилання запрошень та сервісних повідомлень;
- Flask-Limiter – для захисту API від перевантажень та брутфорс-атак;
- Authlib – для реалізації OAuth-авторизації та взаємодії із зовнішніми сервісами.

Таблиця 2.2

Flask-розширення, використані у серверній частині

Розширення	Призначення	Приклад використання
Flask-Login	керування сесіями користувачів	вхід, вихід, захищені маршрути
Flask-WTF	робота з формами та csrf-захистом	авторизація, профіль, зміна даних
Flask-SQLAlchemy	orm-рівень для роботи з базою даних	моделі користувачів, дошок і задач
Flask-Migrate	керування міграціями бази даних	оновлення структури таблиць

Продовження таблиці 2.2

Flask-розширення, використані у серверній частині

Розширення	Призначення	Приклад використання
Flask-SocketIO	події реального часу	сповіщення, оновлення стану дошки
Flask-Limiter	обмеження частоти запитів	захист чутливих маршрутів
Flask-Mail	надсилання електронних листів	запрошення та службові повідомлення
Authlib	oauth-авторизація та зовнішні підключення	інтеграції із зовнішніми сервісами

2.3 Jinja2, HTML, CSS та JavaScript у клієнтській частині

Клієнтська частина поєднує серверний рендеринг Jinja2, HTML-шаблони, CSS-стилі та окремі JavaScript-модулі. Такий підхід дає змогу формувати основні сторінки на сервері, а інтерактивні дії виконувати без повного перезавантаження сторінки. Структура шаблонів та клієнтських модулів представлена у таблиці 2.3. Це доцільно для системи з Kanban-дошкою, модальними формами, вкладеннями, аналітичними блоками та сповіщеннями [7].

Jinja2 використовується для побудови базового шаблону, сторінок авторизації, профілю, дошок та інших екранів. Шаблони отримують дані від Flask-маршрутів і формують HTML-структуру сторінки. Завдяки цьому повторювані елементи, наприклад навігація, підключення стилів, перемикач теми та загальна оболонка інтерфейсу, не дублюються в кожному окремому файлі.

Основою спільної структури виступає шаблон layout/base.html. Він задає каркас сторінок, підключає CSS-файли, JavaScript-модулі та елементи навігації. Такий підхід спрощує підтримку інтерфейсу: зміни в загальній оболонці можна внести в одному місці, не редагуючи кожну сторінку окремо.

JavaScript використовується для інтерактивної поведінки на сторінках. Він відповідає за роботу Kanban-дошки, відкриття модальних вікон, надсилання API-

запитів, обробку вкладень, оновлення аналітичних показників і взаємодію зі спринтами. Частина логіки винесена в загальні модулі, наприклад common/api.js для уніфікованої роботи із запитами та common/theme.js для керування темою інтерфейсу.

Окремі JavaScript-файли відповідають за конкретні функціональні частини дошки. Наприклад, модулі Kanban-логіки працюють зі статусами задач, модальні вікна використовуються для створення й редагування даних, модуль вкладень обробляє файли, а аналітичні модулі відповідають за відображення KPI та інших показників. Такий поділ робить клієнтську логіку зрозумілішою і зменшує ризик змішування різних сценаріїв в одному великому файлі.

CSS-частина складається з базових стилів, стилів компонування, сторінок дошок, авторизації, профілю, соціальних елементів і окремого шару дизайн-системи. Повторювані параметри інтерфейсу, зокрема кольори, відступи, радіуси, тіні та стани елементів, доцільно зберігати централізовано. Це полегшує подальше оновлення зовнішнього вигляду і допомагає підтримувати єдиний стиль на різних сторінках.

Поєднання Jinja2 і Vanilla JavaScript дозволяє залишити сторінки серверно рендереними, але водночас забезпечити потрібний рівень інтерактивності. У результаті клієнтська частина не перетворюється на складний SPA-застосунок, однак підтримує динамічну роботу дошки, фільтрів, форм, вкладень і аналітичних елементів.

Таблиця 2.3

Клієнтські шаблони та JavaScript-модулі web-застосунку

Компонент	Призначення	Роль у клієнтській частині
Базовий шаблон інтерфейсу	формування спільної структури сторінок	визначає загальний каркас web-застосунку, навігацію, підключення стилів і скриптів.

Продовження таблиці 2.3

Клієнтські шаблони та JavaScript-модулі web-застосунку

Компонент	Призначення	Роль у клієнтській частині
Шаблон Kanban-дошки	відображення робочого простору дошки	забезпечує виведення колонок, задач, панелі керування та пов’язаних елементів.
Модуль взаємодії з API	надсилання запитів до серверної частини	уніфікує обмін даними між клієнтським інтерфейсом і flask api.
Модуль керування темою	перемикання режиму оформлення інтерфейсу	забезпечує роботу світлої та темної теми з урахуванням налаштувань користувача.
Модуль Kanban-логіки	обробка дій із задачами на дошці	відповідає за переміщення карток (drag-and-drop), зміну статусів і оновлення стану дошки.
Модуль модальних вікон	робота з формами створення та редагування	забезпечує відкриття, закриття, заповнення й обробку модальних форм.
Модуль вкладень	керування файлами задач	реалізує завантаження, перегляд і видалення вкладень, пов’язаних із задачами.
Модуль KPI-аналітики	відображення аналітичних показників дошки	виводить метрики виконання задач, завантаження команди та стану робочого процесу.
Файл дизайн-системи	визначення єдиного візуального стилю	містить спільні стилі, css-змінні, відступи, кольори та стани елементів інтерфейсу.

2.4 PostgreSQL, SQLAlchemy та Alembic для роботи з даними

Для роботи з даними використано реляційну модель, оскільки предметна область містить складну ієрархію сутностей: користувачів, дошки, учасників, завдання, коментарі, вкладення, спринти, залежності, правила автоматизації, повторювані задачі, сповіщення та інтеграції. Така структура потребує чітких зв'язків між таблицями, контролю цілісності даних і можливості послідовно змінювати схему бази під час розвитку застосунку.

Основна СУБД у Docker-сценарії – PostgreSQL. Вона підходить для зберігання структурованих даних, підтримує зв'язки між таблицями, індекси, обмеження та транзакційність. Для вебзастосунку з дошками, ролями, задачами, вкладеннями й аналітикою це дуже важливо, оскільки більшість операцій пов'язана не з однією таблицею, а з кількома залежними сутностями.

Для взаємодії з базою даних використано SQLAlchemy через Flask-SQLAlchemy. ORM-рівень дозволяє описувати таблиці у вигляді моделей Python, визначати зв'язки між ними та працювати з даними без прямого написання SQL для кожної операції [8-10]. Наприклад, проміжна модель BoardMember не просто реалізує зв'язок між користувачем і дошкою, а й зберігає метадані про ролі та рівні доступу учасників. Основні ORM-моделі застосунку наведені у таблиці 2.4.

Окрему увагу приділено моделюванню залежностей між задачами. Для таких сценаріїв доцільно використовувати окрему структуровану сутність, а не зберігати зв'язки у вигляді текстового опису. Це необхідний крок для подальшої побудови CPM або Gantt-подання, адже важливо знати, яка задача є попередньою, яка залежить від неї і як ці зв'язки впливають на загальний план виконання.

Зміни структури бази даних контролюються за допомогою Alembic і Flask-Migrate [9]. Міграції фіксують додавання нових таблиць, полів, індексів або обмежень, а також послідовні кроки. Завдяки цьому оновлення схеми не виконується вручну без історії, а може бути відтворене в іншому середовищі. Це особливо важливо для проєкту, у якому модель даних поступово розширюється

новими модулями: спринтами, автоматизаціями, інтеграціями, сповіщеннями або налаштуваннями користувача.

Таблиця 2.4

Основні ORM-моделі web-застосунку

Модель	Призначення	Основні зв’язки
User	обліковий запис користувача	дошки, задачі, коментарі, повідомлення
Board	робочий простір для задач	учасники, задачі, спринти, правила
BoardMember	участь користувача в дошці	користувач, дошка, роль
Task	задача в межах дошки	коментарі, вкладення, виконавець, спринт
TaskComment	обговорення задачі	задача, автор коментаря
TaskAttachment	файли та посилання до задачі	задача, користувач, файл
Sprint	ітерація роботи	дошка, задачі, строки виконання
TaskDependency	залежність між задачами	попередня та наступна задача
BoardRule	правило автоматизації	дошка, умови, дії
RecurringTaskTemplate	шаблон повторюваної задачі	дошка, параметри повторення
Notification	сповіщення користувача	користувач, тип події, стан прочитання

2.5 Flask-SocketIO, Redis та APScheduler для реального часу і фонових задач

Окрім звичайних HTTP-запитів, веб-застосунок потребує механізмів для подій реального часу та періодичних операцій. До таких сценаріїв належать сповіщення, оновлення стану дошки, нагадування, повторювані задачі, автоматизації та синхронізація з окремими зовнішніми сервісами.

Для подій реального часу використано Flask-SocketIO [24]. Він миттєво доставляє сповіщення про зміни на дошках іншим учасникам проєкту без потреби вручну оновлювати сторінки. Також він підходить для швидкого відображення змін в інтерфейсі.

Redis виконує інфраструктурну роль. Він використовується для обмеження частоти запитів, черги повідомлень Socket.IO та механізмів блокування фонових задач [25]. Обмеження частоти потрібне для захисту чутливих маршрутів (наприклад, авторизації або API-операцій). Черга повідомлень спрощує передавання подій між процесами, а job locks зменшують ризик одночасного виконання однієї й тієї самої фонової операції.

Періодичні операції виконуються за допомогою APScheduler [26]. Він запускає повторювані задачі, нагадування і правила автоматизації. Такі дії не повинні залежати від ручного втручання, тому їх доцільно винести у фоновий режим. Наприклад, система може перевіряти наближення дедлайнів, створювати задачі за шаблоном або запускати правило, якщо виконано певну умову.

Для фонових операцій важливо уникати дублювання. Якщо застосунок працює в середовищі з кількома процесами, одна й та сама задача може випадково запуститися кілька разів. Тому використовуються механізми блокування та ознака відповідального scheduler-процесу, який унеможливлює дублювання сповіщень, повторюваних задач або автоматизованих дій.

У сукупності Flask-SocketIO, Redis і APScheduler доповнюють звичайну request-response модель Flask і забезпечують події реального часу, контроль

навантаження на API, фонову обробку і стабільніше виконання автоматизованих сценаріїв.

2.6 Docker та Docker Compose для розгортання

Контейнеризація використовується для підготовки відтворюваного середовища, в якому web-застосунок може працювати разом із потрібними сервісами. Dockerfile описує web-сервіс, а Docker Compose об'єднує його з PostgreSQL і Redis. Завдяки цьому не потрібно окремо налаштовувати СУБД, Redis та змінні середовища вручну на кожному комп'ютері [11]. Схема розгортання та склад сервісів представлені на рисунку 2.2 та у таблиці 2.5.

Організація роботи через Docker Compose передбачає функціонування кількох взаємопов'язаних вузлів, де вебсервіс відповідає за запуск серверної частини, PostgreSQL забезпечує надійне зберігання основних даних, а Redis залучається для вирішення інфраструктурних завдань, зокрема обмеження частоти запитів, управління чергами Socket.IO та обробки системних блокувань. Така структура відповідає архітектурі застосунку, бо він залежить не лише від Flask, а й від окремих сервісів для зберігання даних і допоміжної обробки.

Конфігураційні параметри передаються через змінні середовища. До них належать адреса бази даних, параметри Redis, секретні ключі, налаштування безпеки, Oauth-дані та інші службові значення. Це дозволяє відокремити код від конфігурації і не зберігати чутливі параметри безпосередньо в програмних файлах.

Після запуску контейнерів потрібно оновити схему бази даних за допомогою міграцій. Для цього використовується Flask-Migrate, який застосовує підготовлені Alembic-міграції. Такий порядок важливий, оскільки серверна частина очікує наявність потрібних таблиць, полів і зв'язків.

Окремо слід враховувати доступність залежних сервісів. Якщо PostgreSQL або Redis ще не готові до роботи, частина функцій застосунку не зможе виконуватися коректно. Тому в Docker-сценарії доцільно використовувати

перевірки стану сервісів і запускати web-процес лише після готовності основних залежностей.

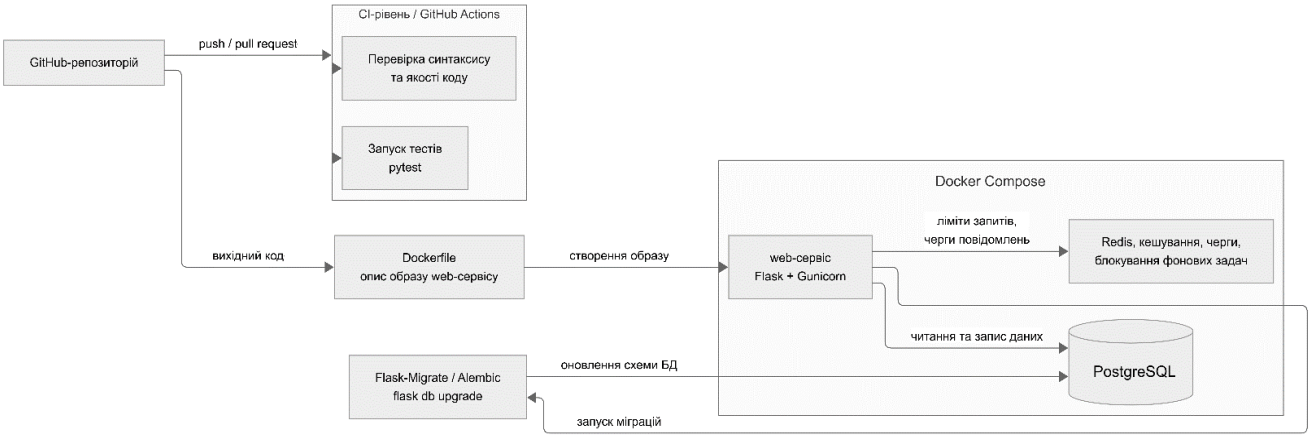


Рис. 2.2 Схема CI та Docker-розгортання web-застосунку

Таблиця 2.5

Сервіси Docker Compose для розгортання web-застосунку

Сервіс	Призначення	Дані або функції
web	запуск серверної частини	http-запити, web-сторінки, api
postgres	основна реляційна субд	користувачі, дошки, задачі, пов’язані таблиці
redis	інфраструктурний сервіс	rate limiting, socket.io queue, job locks
migrations	оновлення схеми бази даних	застосування flask-migrate / alembic
env	конфігурація середовища	секрети, адреси сервісів, параметри безпеки
volumes	збереження даних між запусками	файли бази даних і завантаження

3 ПРОЄКТУВАННЯ WEB-ЗАСТОСУНКУ

3.1 Загальна архітектура системи

Архітектура web-застосунку побудована за багаторівневим принципом. У ній можна виділити клієнтський рівень, серверний рівень, рівень даних, фонові задачі та зовнішні інтеграції. Клієнтська частина відповідає за відображення сторінок і взаємодію користувача з інтерфейсом. Серверна частина обробляє маршрути web-сторінок, REST API, перевірку доступу, бізнес-логіку та взаємодію з базою даних [2-4; 6; 30].

На рівні представлення використовуються HTML-шаблони, CSS-стилі та JavaScript-модулі. Шаблони формують структуру сторінок, стилі відповідають за зовнішній вигляд, а JavaScript забезпечує інтерактивні дії: роботу Kanban-дошки, модальних форм, фільтрів, вкладень, аналітичних блоків і оновлення окремих елементів сторінки без повного перезавантаження.

Серверний рівень реалізовано на Flask. Він поєднує web-маршрути, REST API, сервісний шар, ORM-моделі та допоміжні механізми. Маршрути приймають запити, API повертає структуровані JSON-відповіді, а сервісний шар виконує операції, які не варто розміщувати безпосередньо в маршрутах. До таких операцій належать робота з автоматизаціями, сповіщеннями, повторюваними задачами, інтеграціями та аналітичними розрахунками.

Рівень даних побудовано на ORM-моделях SQLAlchemy та PostgreSQL. Моделі описують основні сутності предметної області: користувачів, дошки, учасників, задачі, коментарі, вкладення, спринти, залежності, сповіщення та інтеграції. PostgreSQL використовується як основне сховище даних, оскільки більшість операцій у системі пов’язана з кількома взаємозалежними сутностями.

Фонові задачі виконуються за допомогою APScheduler. Цей механізм потрібний для повторюваних задач, нагадувань, автоматизацій та інших операцій, які мають запускатися без прямої дії користувача. Redis використовується для

допоміжних інфраструктурних сценаріїв: обмеження частоти запитів, черги подій Socket.IO та блокування повторного виконання фонових задач.

Взаємодія користувача із системою може відбуватися кількома шляхами. Під час відкриття сторінки браузер надсилає HTTP-запит, сервер перевіряє сесію, формує контекст і повертає HTML-сторінку. Для інтерактивних дій JavaScript-модулі надсилають JSON-запити до REST API. Якщо дія потребує швидкого інформування користувача, додатково створюється подія реального часу через Socket.IO або сповіщення. Загальну схему взаємодії модулів системи відображено на рисунку 3.1.

Для зовнішніх сервісів передбачено окремі інтеграційні сценарії. Вони включають збереження параметрів підключення, перевірку токенів, обробку помилок і синхронізацію даних із Telegram, Slack, Google Calendar, Web Push або email-сповіщеннями. Такий поділ дозволяє відокремити основну логіку задач від взаємодії із зовнішніми сервісами.

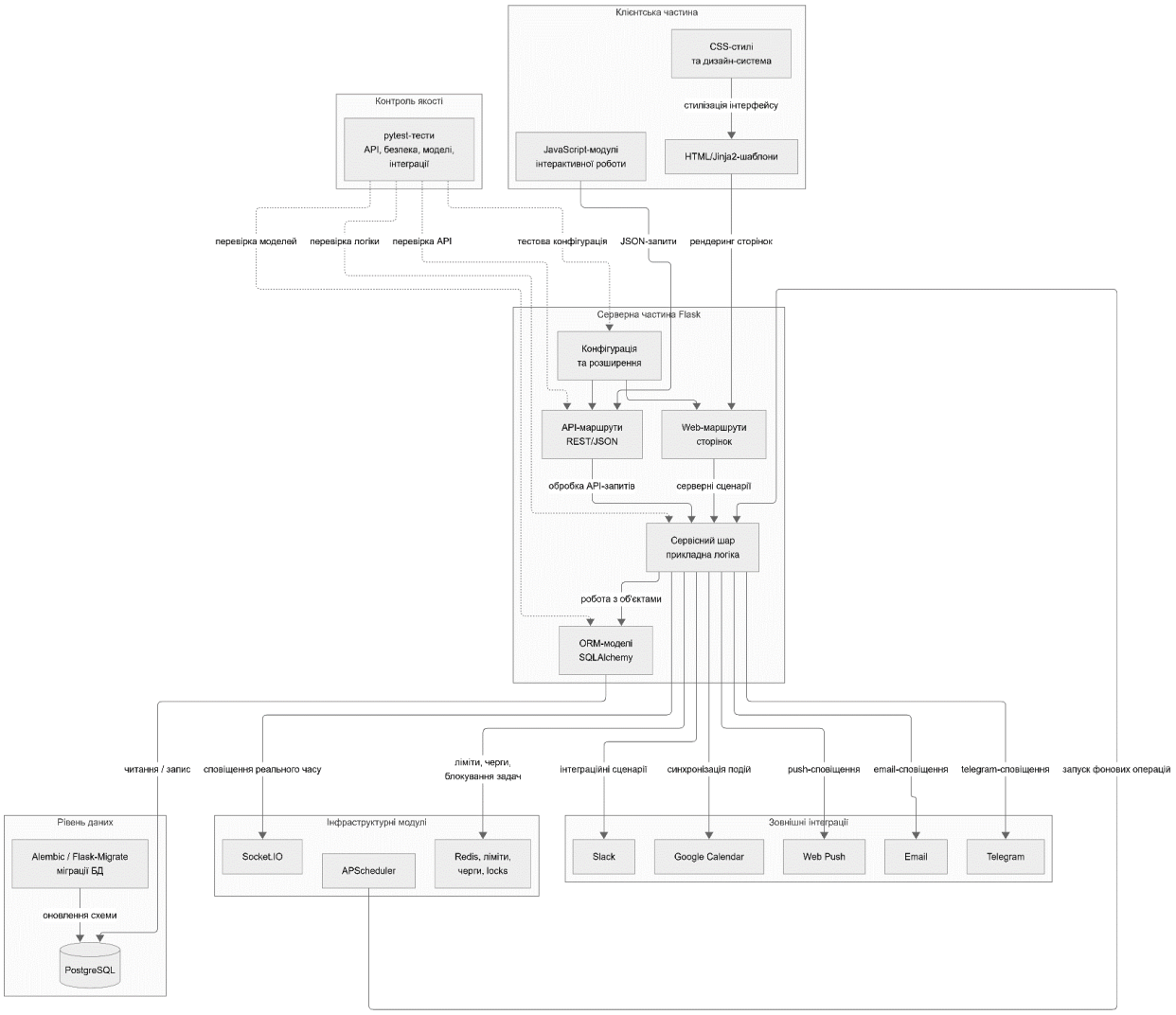


Рис. 3.1 Діаграма основних модулів MiniTrello

3.2 Ролі користувачів і сценарії використання

Ефективне керування робочим простором забезпечується через розмежування прав доступу та реалізацію типових послідовностей дій для кожного типу учасників.

Рольова модель системи.

Для ефективного розмежування доступу до функціональних можливостей системи застосовується рольова модель. Вона передбачає дворівневу структуру прав користувача:

- загальні права на рівні облікового запису (глобальний статус);

- спеціалізовані права у межах конкретної робочої дошки (локальні повноваження).

Такий підхід забезпечує гнучкість управління доступом: один і той же користувач може мати різні рівні повноважень у різних робочих просторах. Рольова модель та діаграма варіантів використання системи наведені у таблиці 3.1 та на рисунку 3.2 відповідно.

Опис ролей користувачів.

Система передбачає наступні типи ролей, кожна з яких має чітко визначений набір функціональних можливостей:

Гість:

- перегляд публічних сторінок системи;
- можливість реєстрації в системі;
- здійснення входу до системи.

Зареєстрований користувач:

- доступ до персонального профілю;
- створення та керування власними робочими дошками;
- налаштування індивідуальних параметрів (тема відображення, мова інтерфейсу, параметри безпеки).

Ролі на рівні робочої дошки.

Власник дошки - користувач з максимальним рівнем повноважень:

- керує робочим простором,
- управляє учасниками та їх ролями;
- відправляє запрошення до роботи над дошкою;
- здійснює налаштування дошки; виконує експорт даних.

Адміністратор – має широкий спектр керувальних функцій, аналогічних власнику, за винятком:

- не має права змінювати або видаляти власника дошки.

Менеджер – відповідає за організацію робочого процесу:

- планування та керування задачами;
- організація спринтів;

- встановлення пріоритетів;
- аналіз виконання задач.
- рівень контролю обмежений порівняно з власником та адміністратором.

Учасник – користувач, що виконує конкретні завдання:

- робота з задачами в межах наданих прав;
- додавання коментарів;
- прикріплення файлів та інших вкладень.

Переглядач – користувач з мінімальним рівнем доступу:

- має право лише на перегляд інформації;
- не може редагувати дані або виконувати будь-які дії.

Основні сценарії використання системи.

Практичне використання системи передбачає реалізацію наступних типових сценаріїв:

- реєстрація та авторизація користувачів;
- налаштування персонального профілю;
- створення робочих дошок;
- запрошення та управління учасниками проектів;
- формування задач та контроль за зміною їх статусів;
- обмін інформацією через коментарі та вкладення;
- перегляд та аналіз сповіщень системи;
- управління спринтами (етапами виконання проектів);
- аналіз ключових показників ефективності;
- налаштування механізмів автоматизації процесів;
- інтеграція зі сторонніми сервісами та системами.

Особливий сценарій: запрошення користувача на робочу дошку.

Механізм запрошення нового користувача на робочу дошку реалізований за допомогою спеціального алгоритму, що забезпечує контрольований доступ до робочого простору:

- генерація унікального токена для запрошення.

- визначення ролі, яку отримає новий учасник.
- формування персональної посилання для запрошення.
- автоматичне додавання користувача до робочої дошки з відповідними правами після прийняття запрошення.

Цей механізм виключає можливість несанкціонованого доступу через відкриті посилання та забезпечує високий рівень безпеки роботи з даними.

Таблиця 3.1

Ролі користувачів і доступні дії

Роль	Доступні дії	Обмеження
Гість	реєстрація, вхід, перегляд публічних сторінок	немає доступу до приватних дошок
Користувач	налаштування профілю, створення власних дошок	працює лише зі своїми даними та доступними дошками
Власник	повне керування дошкою, учасниками, ролями й налаштуваннями	діє в межах конкретної дошки
Адміністратор	керування учасниками, задачами, ролями та частиною налаштувань	не повинен змінювати права власника
Менеджер	робота із задачами, спринтами, пріоритетами та аналітикою	обмежені небезпечні адміністративні дії
Учасник	створення і виконання задач, коментарі, вкладення	дії залежать від ролі на дошці
Переглядач	перегляд дошки, задач і матеріалів	без права редагування

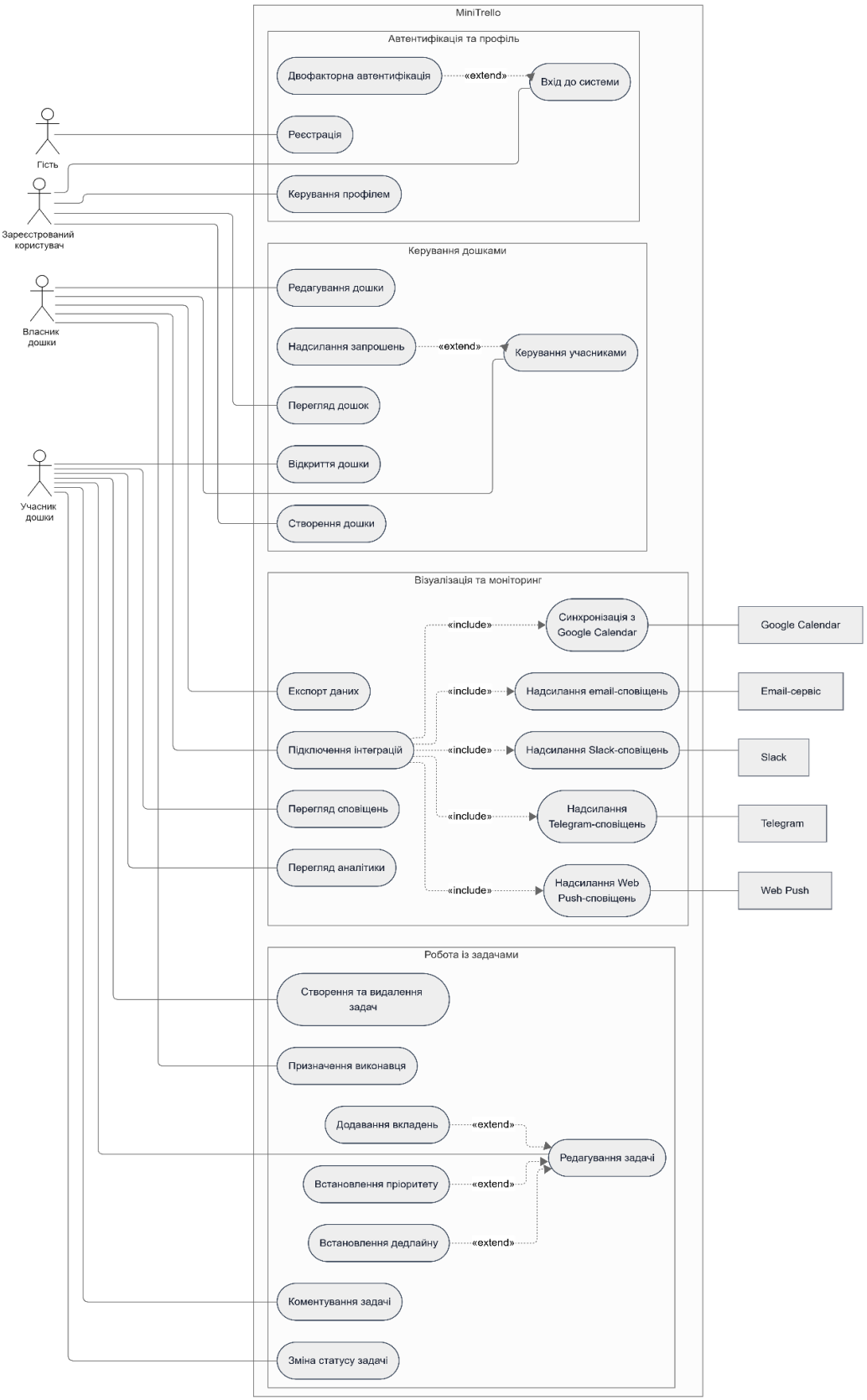


Рис. 3.2 Діаграма варіантів використання MiniTrello

3.3 Проєктування моделі даних

Модель даних охоплює сутності, необхідні для управління задачами, командною взаємодією, аналітикою, автоматизаціями та інтеграціями. Фундаментом інформаційної структури виступає сукупність таких сутностей, як користувачі, дошки та їхні учасники, задачі з коментарями, вкладення, спринти, залежності, сповіщення, а також моделі зовнішніх підключень. Така структура відповідає предметній області, оскільки задача не існує окремо від дошки, виконавця, строку, обговорення, матеріалів і пов’язаних дій.

Сутність User призначена для зберігання параметрів облікового запису, зокрема електронної пошти, хешу пароля, профільних полів, мови та теми інтерфейсу, а також глобальної ролі, OAuth-даних, налаштувань двофакторної автентифікації та кодів відновлення (recovery codes). Дошка описує робочий простір, у межах якого створюються задачі та визначаються учасники. Проміжна сутність BoardMember пов’язує користувача з дошкою і зберігає його роль у цьому робочому просторі.

Задача є центральною сутністю робочого процесу. Вона містить назву, опис, статус, пріоритет, строк виконання, виконавця, прогрес та інші поля, потрібні для планування і контролю. Коментарі й вкладення зберігають робочий контекст, а залежності між задачами дозволяють визначати послідовність виконання і будувати CPM або Gantt-подання.

Окрему групу становлять спринти та аналітичні дані. Спринт пов’язує набір задач із певним періодом роботи, а аналітичні модулі використовують статуси, дедлайни, залежності, активність і завантаження учасників для побудови показників. Такий підхід дозволяє оцінювати не лише окрему задачу, а й загальний стан робочого процесу.

Моделі інтеграцій відокремлені від основної логіки задач. SlackIntegration, TelegramIntegration, GoogleCalendarIntegration, GoogleCalendarTaskSync і PushSubscription зберігають параметри підключення, статуси, зашифровані токени

та службові дані. Це зменшує зв’язність між задачами й зовнішніми сервісами, а також спрощує контроль помилок синхронізації.

Якщо говорити про модель сповіщень, вона насамперед потрібна для інформування користувачів про події, які стосуються дошок, задач, запрошень, коментарів або інтеграцій. Сповіщення має зберігати отримувача, тип події, службові дані та стан прочитання. Це дозволяє користувачеві повернутися до важливих подій навіть після закриття сторінки або завершення сесії. Логічна модель даних та опис груп сутностей наведені у таблиці 3.2 та на рисунку 3.3.

Таблиця 3.2

Групи моделей даних web-застосунку

Група	Сутності	Призначення
Користувачі	user, usersettings	обліковий запис, профіль, мова, тема, параметри безпеки
Дошки	board, boardmember, invitetoken	робочий простір, учасники, ролі та запрошення
Задачі	task, taskdependency	створення задач, статуси, дедлайни, залежності
Контекст задач	taskcomment, taskattachment	обговорення, файли та матеріали
Спринти	sprint	ітераційне планування роботи
Автоматизації	boardrule, rulefiring	правила, умови та виконані автоматизовані дії
Сповіщення	notification, pushsubscription	інформування користувачів і web push

Продовження таблиці 3.2

Групи моделей даних web-застосунку

Група	Сутності	Призначення
Інтеграції	slackintegration, telegramintegration, googlecalendarintegration	підключення зовнішніх сервісів

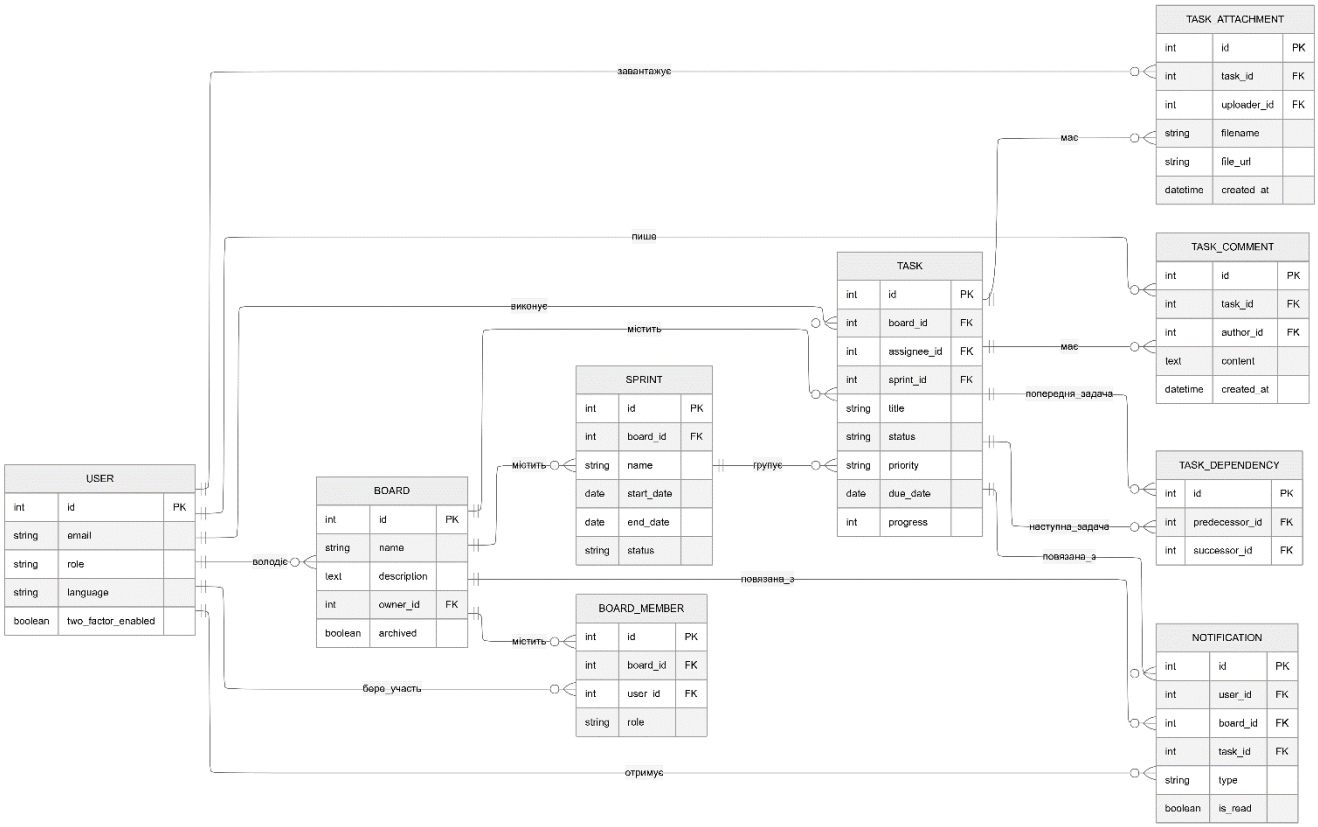


Рис. 3.3 ER-діаграма основних сутностей MiniTrello

3.4 Проєктування Kanban-дошки та задач

Kanban-дошка проєктується як робочий простір, у якому задачі розміщуються за статусами. Кожна колонка відповідає певному етапу виконання, при цьому картка задачі для забезпечення швидкого візуального перегляду акумулює такі ключові відомості, як назва, пріоритет, виконавець, строк, прогрес та допоміжні позначення. Повний опис, чеклист, мітки, вкладення, коментарі та залежності відкриваються у формі задачі.

Створення задачі починається з відкриття форми та введення основних даних. Після надсилання запиту сервер перевіряє CSRF-токен, сесію користувача, членство в дошці та наявність потрібних прав. Якщо перевірки пройдені успішно, задача записується в базу даних, а система може створити активність, сповіщення або оновити клієнтський інтерфейс. Якщо користувач не має доступу до дошки, запит має бути відхилений.

Зміна статусу задачі є не лише візуальною дією. Вона змінює стан елемента в робочому процесі та може впливати на активність, сповіщення, автоматизації й аналітичні показники. Схема процесу створення задачі та опис її полів наведені на рисунку 3.4 та у таблиці 3.3. Тому переміщення картки між колонками повинно супроводжуватися серверною обробкою, перевіркою прав і збереженням нового стану.

Додаткові властивості задачі підтримують роботу з реальними проєктами. Мітки допомагають групувати задачі за напрямами, чеклист деталізує виконання на дрібніші кроки, вкладення зберігають матеріали, а коментарі фіксують обговорення. Залежності між задачами потрібні для побудови критичного шляху та часових подань.

Для зручності роботи з великою кількістю задач передбачаються фільтри. Користувач може переглядати задачі за виконавцем, статусом, строком, пріоритетом або іншими параметрами. Збережені фільтри корисні для повторюваних сценаріїв, коли потрібно регулярно переглядати один і той самий набір задач.

Таблиця 3.3

Поля задачі та їх призначення

Поле задачі	Зміст	Призначення
title	назва задачі	швидка ідентифікація роботи
description	опис	пояснення очікуваного результату

Продовження таблиці 3.3

Поля задачі та їх призначення

Поле задачі	Зміст	Призначення
status	поточний стан	розміщення задачі в kanban-колонці
priority	пріоритет	визначення важливості виконання
due_date	строк виконання	контроль дедлайнів і ризиків
assignee_id	виконавець	закріплення відповідальної особи
labels	мітки	групування і фільтрація задач
checklist	підкроки	відстеження часткового прогресу
attachments	вкладення	збереження матеріалів задачі
dependencies	залежності	побудова cpm і gantt-подання

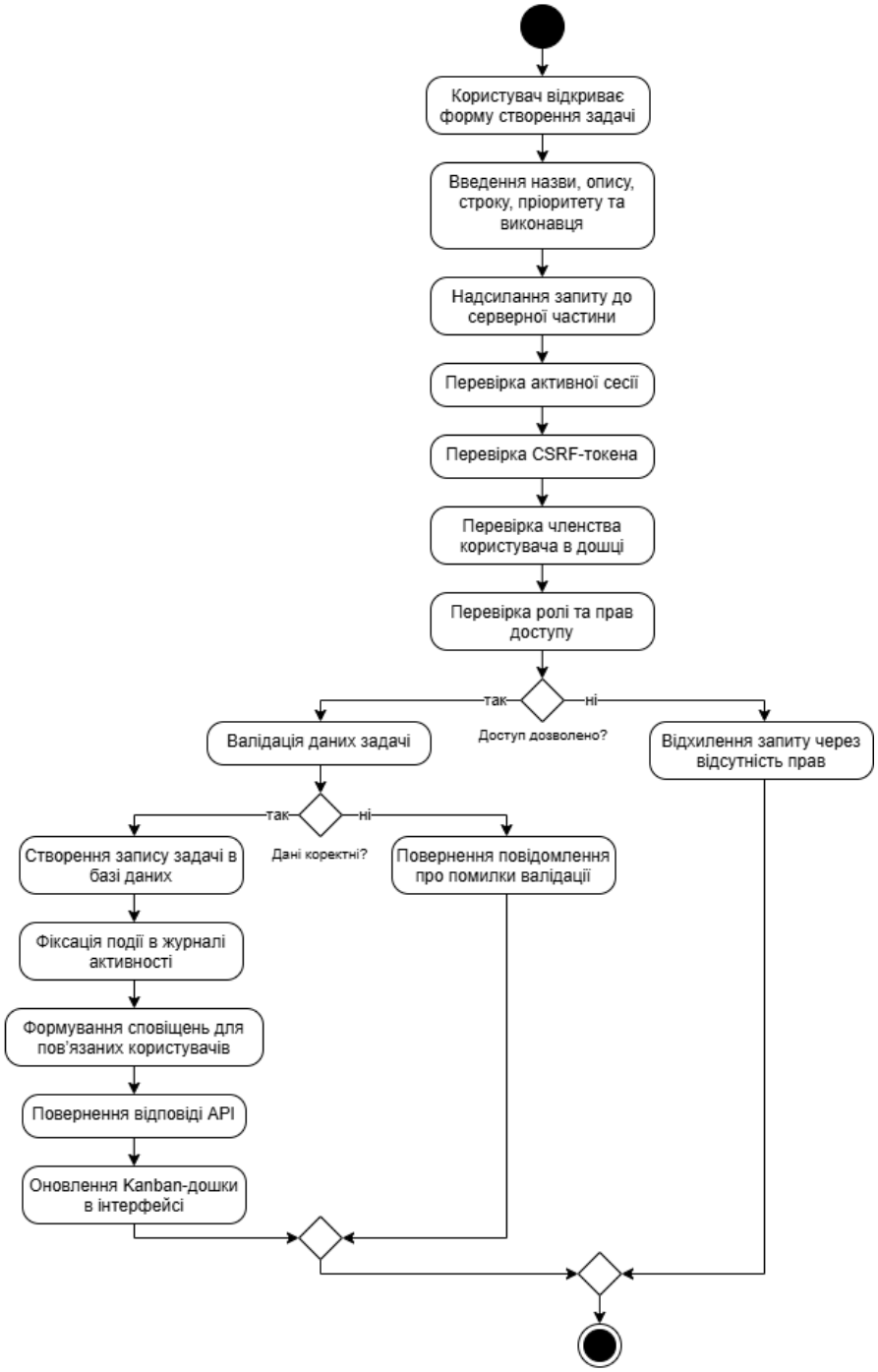


Рис. 3.4 Діаграма активності створення задачі

3.5 Проєктування командної взаємодії, запрошень і повідомлень

Командна взаємодія починається з додавання користувачів до робочого простору. Для цього використовується механізм запрошень, який пов’язує дошку, email або обліковий запис користувача, роль майбутнього учасника, токен і строк

дії. Запрошення має бути контрольованим за статусом, щоб уникнути повторного або несанкціонованого приєднання.

Після прийняття запрошення користувач стає учасником дошки з визначеною роллю. Це дозволяє одразу обмежити або надати потрібні дії: перегляд задач, створення коментарів, редагування карток, керування учасниками або доступ до аналітики. Такий підхід важливий для командної роботи, оскільки доступ до дошки не повинен бути однаковим для всіх.

Сповіщення створюються у відповідь на події, які потребують уваги користувача: запрошення до дошки, зміну задачі, новий коментар, наближення дедлайну, згадку або системну дію. Модель Notification зберігає тип події, отримувача, службові дані та стан прочитання. Це дозволяє користувачеві бачити важливі зміни навіть тоді, коли він не перебував на відповідній сторінці.

Для оперативної доставки подій можуть використовуватися Socket.IO та Web Push. Socket.IO підходить для оновлень у межах активної сесії, а Web Push дозволяє повідомляти користувача через браузерні push-сповіщення. При цьому кожне сповіщення повинно враховувати права доступу: користувач не має отримувати інформацію про дошку або задачу, до якої він не має доступу.

Прямі повідомлення та дружні зв’язки доповнюють дошкову взаємодію, але не замінюють коментарі до задач. Коментарі мають залишатися основним способом обговорення конкретного доручення, оскільки вони зберігають контекст безпосередньо біля задачі. Приватні повідомлення більше підходять для загальної комунікації між користувачами.

Email-запрошення використовується як додатковий канал для передачі посилання на приєднання до дошки. Загальна схема роботи системи сповіщень наведена на рисунку 3.5. Такий механізм корисний у випадках, коли користувач ще не працює в системі або не бачить внутрішні сповіщення. При цьому посилання має містити контрольований токен, а не відкривати доступ без перевірки.

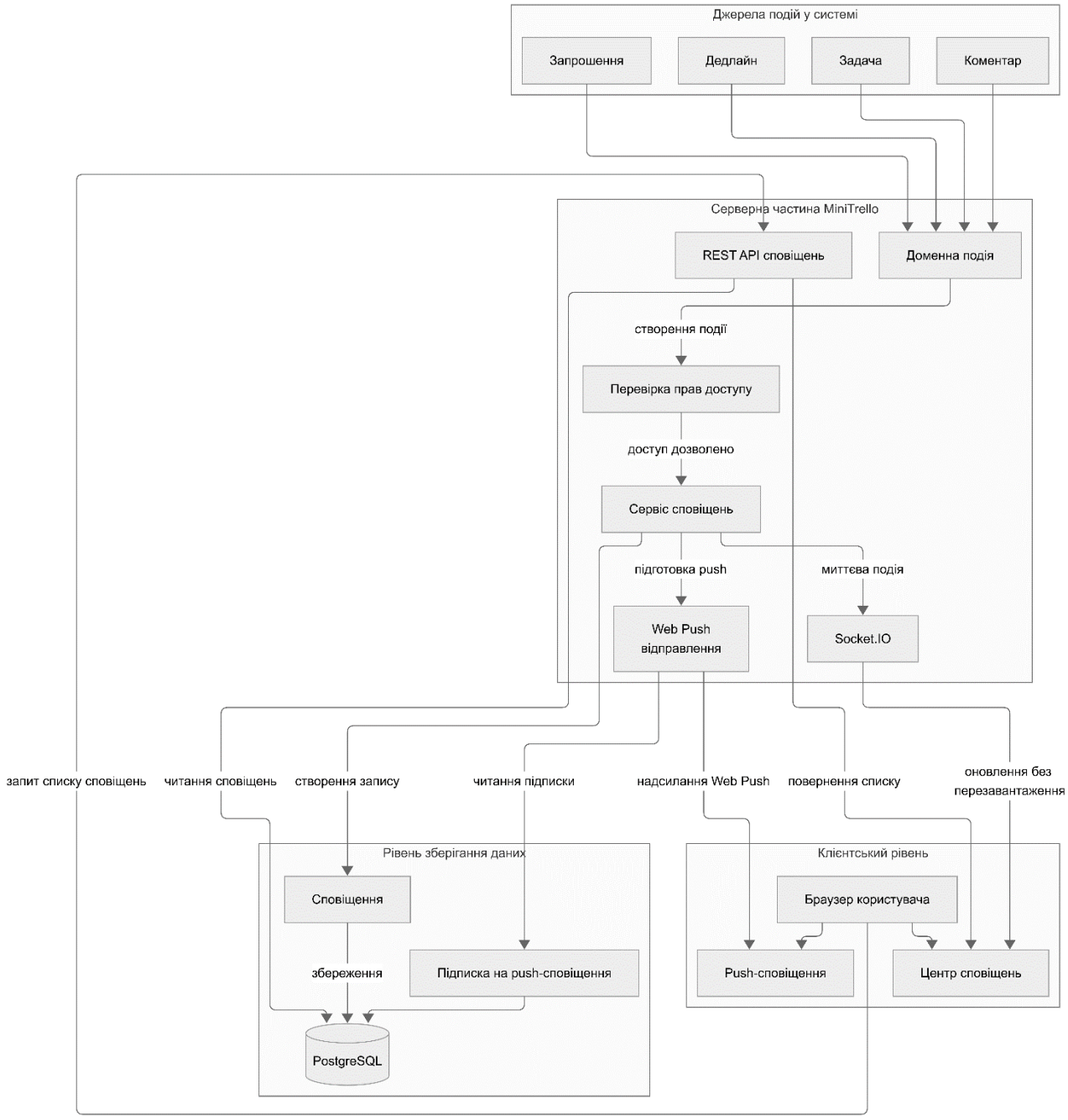


Рис. 3.5 Діаграма взаємодії компонентів підсистеми сповіщень

3.6 Проєктування аналітики та візуалізації робочих процесів

Аналітика проєктується як набір показників і візуальних подань, що спираються на фактичні дані задач. Її призначення полягає не лише у відображенні кількості виконаних або активних задач, а й у виявленні ризиків, залежностей, перевантаження учасників і проблемних елементів робочого процесу [12; 31; 32].

Основними напрямами є KPI, Gantt, CPM, heatmap, оцінювання перевантаження команди, список проблемних задач і дані для спринтів. KPI дають короткий зріз стану дошки: активні, завершені, прострочені та проблемні задачі. Показник перевантаження допомагає оцінити, чи не зосереджено занадто багато роботи на окремих учасниках.

Список проблемних задач потрібний для швидкого виявлення елементів, які створюють найбільший ризик для строків. До таких задач можуть належати прострочені, заблоковані або залежні від інших робіт елементи. Це допомагає керівнику дошки зосередитися не на всьому переліку задач, а на тих, що потребують першочергової уваги.

Метод критичного шляху використовує залежності між задачами для визначення послідовності робіт, яка найбільше впливає на загальний строк виконання. Gantt-подання відображає задачі у часовій площині та допомагає побачити накладання робіт, послідовність виконання і можливі затримки.

Heatmap використовується для відображення щільності активності або навантаження в певні періоди. Таке подання допомагає побачити пікові моменти роботи, нерівномірний розподіл активності та періоди, коли команда потребує додаткової уваги до планування.

Аналітика спринтів і velocity потрібні для оцінювання ітераційного виконання. Вони дозволяють порівнювати запланований та фактичний обсяг роботи, аналізувати завершені задачі й оцінювати динаміку команди в межах окремих періодів.

Важливо, що аналітичні подання не повинні формуватися вручну в окремих файлах. Вони мають спиратися на дані задач, статуси, дедлайни, спринти, залежності, активність і завантаження учасників. Це робить звітність більш актуальною і зменшує ризик розбіжностей між фактичним станом роботи та підготовленими вручну звітами. Схема аналітичних модулів та їх опис наведені на рисунку 3.6 та у таблиці 3.4.

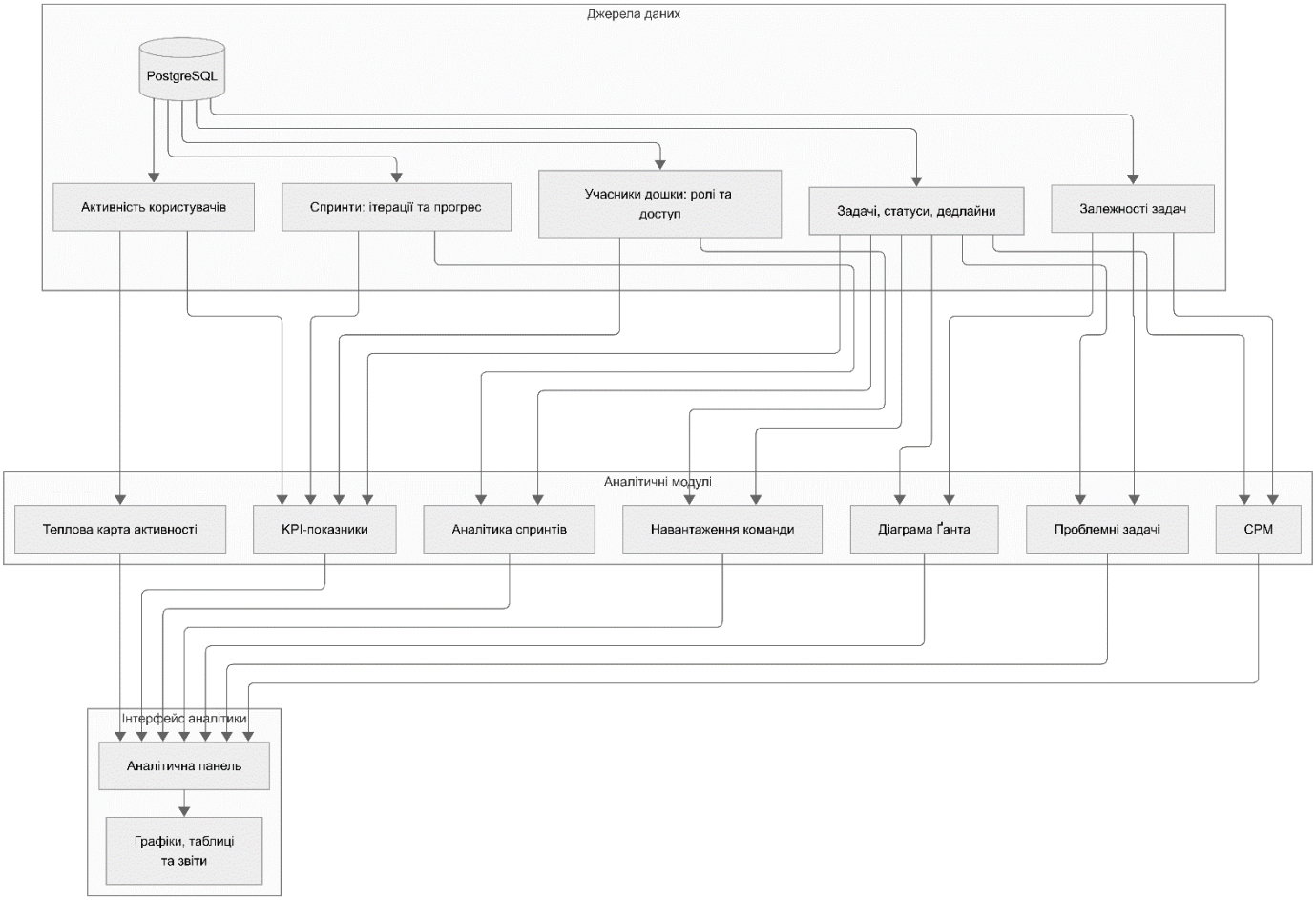


Рис. 3.6 Діаграма компонентів аналітичної підсистеми

Таблиця 3.4

Аналітичні модулі web-застосунку

Модуль	Дані	Значення
KPI	статуси, дедлайни, виконані задачі	короткий огляд стану роботи
Team Overload	виконавці, пріоритети, кількість задач	оцінювання ризику перевантаження
Problematic Tasks	прострочення, блокування, залежності	визначення задач, що потребують уваги
CPM	залежності між задачами	пошук критичного шляху
Gantt	дати початку, строки, залежності	часова схема виконання

Продовження таблиці 3.4

Аналітичні модулі web-застосунку

Модуль	Дані	Значення
Heatmap	активність і навантаження	виявлення пікових періодів
Sprint analytics	спринти, задачі, прогрес	оцінювання ітераційної роботи

3.7 Проєктування безпеки даних

Безпека даних проєктується на кількох рівнях:

- автентифікація користувачів;
- хешування паролів;
- захист сесій;
- CSRF-захист;
- рольова модель доступу;
- обмеження частоти запитів;
- двофакторна автентифікація;
- recovery codes;
- OAuth-підключення;
- захист токенів інтеграцій;
- контроль доступу до вкладень;
- API.

Автентифікація підтверджує особу користувача перед доступом до приватних сторінок і даних. Паролі не повинні зберігатися у відкритому вигляді, тому для них використовується хешування. Додатково двофакторна автентифікація та recovery codes зменшують ризик компрометації облікового запису у випадку втрати або розголошення пароля.

CSRF-захист потрібний для форм і запитів, які змінюють стан системи. Він зменшує ризик виконання небажаної дії від імені користувача. Це особливо

важливо для операцій створення, редагування або видалення задач, зміни профілю, роботи з вкладеннями та керування учасниками.

Рольова модель доступу повинна працювати не лише на рівні сторінок, а й на рівні API. Якщо користувач не є учасником дошки або не має потрібної ролі, запит на зміну задачі, видалення вкладення, перегляд аналітики або керування учасниками має бути відхилений. Це зменшує ризик обходу інтерфейсу через прямі HTTP-запити.

Обмеження частоти запитів використовується для захисту чутливих маршрутів. До таких маршрутів належать авторизація, відновлення доступу, операції з токенами, API-запити та інші дії, які можуть бути використані для перебору або надмірного навантаження.

Токени інтеграцій потрібно зберігати окремо від основних задач і захищати на рівні моделі. Для Telegram, Slack, Google Calendar і Web Push важливо контролювати не лише сам факт підключення, а й статус токена, помилки синхронізації та доступ користувача до відповідної дошки або сервісу.

Окрему увагу потрібно приділити вкладенням. Файли можуть містити службові або приватні матеріали, тому доступ до них має перевірятися так само, як доступ до задачі або дошки. Крім того, доцільно обмежувати розмір завантаження, щоб зменшити ризик перевантаження системи або неконтрольованого використання сховища.

Під час проєктування враховуються ризики, описані в OWASP Top 10: помилки автентифікації, порушення контролю доступу, недостатній криптографічний захист, небезпечна робота з токенами та некоректна обробка запитів. Для такого web-застосунку безпека має бути частиною архітектури, а не окремим доповненням після реалізації основних функцій. Основні ризики та механізми захисту наведені у таблиці 3.5.

Таблиця 3.5

Ризики безпеки та механізми їх зменшення

Ризик	Можливий наслідок	Механізм захисту
Підбір пароля	компрометація облікового запису	хешування паролів, rate limiting, 2fa
CSRF-атака	виконання дії від імені користувача	csrf-токени
Порушення контролю доступу	доступ до чужих дошок або задач	rbac і перевірка участі в дошці
Витік токенів інтеграцій	несанкціонований доступ до зовнішніх сервісів	захищене зберігання токенів
Неконтрольоване завантаження файлів	перевантаження або витік матеріалів	обмеження розміру і перевірка доступу
Ризики сесії	перехоплення або небажане використання сесії	безпечні cookie та перевірка автентифікації
Дублювання фонових задач	повторні сповіщення або дублікати задач	job locks
Помилки OAuth	неправильний redirect або підміна стану	state validation

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

4.1 Структура проєкту

Структура проєкту побудована за модульним принципом, що ілюструється на рисунку 4.1. Роль основних файлів та директорій наведена у таблиці 4.1. Окремо виділено моделі даних, сервіси прикладної логіки, API-маршрути, web-маршрути, шаблони, статичні ресурси, тести, конфігурацію, міграції та файли для розгортання. Такий поділ дозволяє підтримувати застосунок як цілісну систему, а не як набір непов’язаних скриптів.

До серверної частини належать моделі предметної області, сервіси, маршрути API, web-маршрути, форми, розширення Flask та файл ініціалізації застосунку. Моделі описують користувачів, дошки, задачі, ролі, сповіщення, спринти, автоматизації та інтеграції. Сервіси містять операції, які не варто розміщувати безпосередньо в маршрутах: перевірку ролей, аналітичні розрахунки, обробку інтеграцій, повторювані задачі, Web Push та email-повідомлення.

Клієнтська частина складається з Jinja2-шаблонів, CSS-файлів і JavaScript-модулів. Шаблони формують сторінки, CSS відповідає за оформлення, а JavaScript забезпечує інтерактивність Kanban-дошки, модальних вікон, фільтрів, вкладень, аналітики та спринтів.

Окремо розміщено файли конфігурації, Dockerfile, docker-compose.yml, requirements.txt, Alembic-міграції та CI-сценарій. Вони потрібні для керування залежностями, запуску сервісів, оновлення схеми бази даних і перевірки якості коду.

Таблиця 4.1

Структура проєкту web-застосунку

Файл	Роль	Приклад використання
app/models	моделі даних	user, board, task, sprint

Продовження таблиці 4.1

Структура проєкту web-застосунку

Файл	Роль	Приклад використання
app/services	прикладна логіка	rbac, аналітика, інтеграції, rule engine
app/views/api	api-маршрути	дошки, задачі, сповіщення, інтеграції
app/views/web	web-сторінки	маршрути сторінок і представлень
app/templates	html-шаблони	сторінки дошок, профілю, авторизації
app/static/js	клієнтська інтерактивність	kanban.js, modals.js, attachments.js
app/static/css	стилі інтерфейсу	базові стилі, дизайн-система
app/tests	автоматизовані перевірки	pytest-тести api, безпеки та інтеграцій
migrations	міграції бази даних	alembic-оновлення схеми
config.py	конфігурація	база даних, redis, безпека, пошта
Dockerfile	контейнер web-сервісу	запуск серверної частини
docker-compose.yml	склад сервісів	web, postgresql, redis

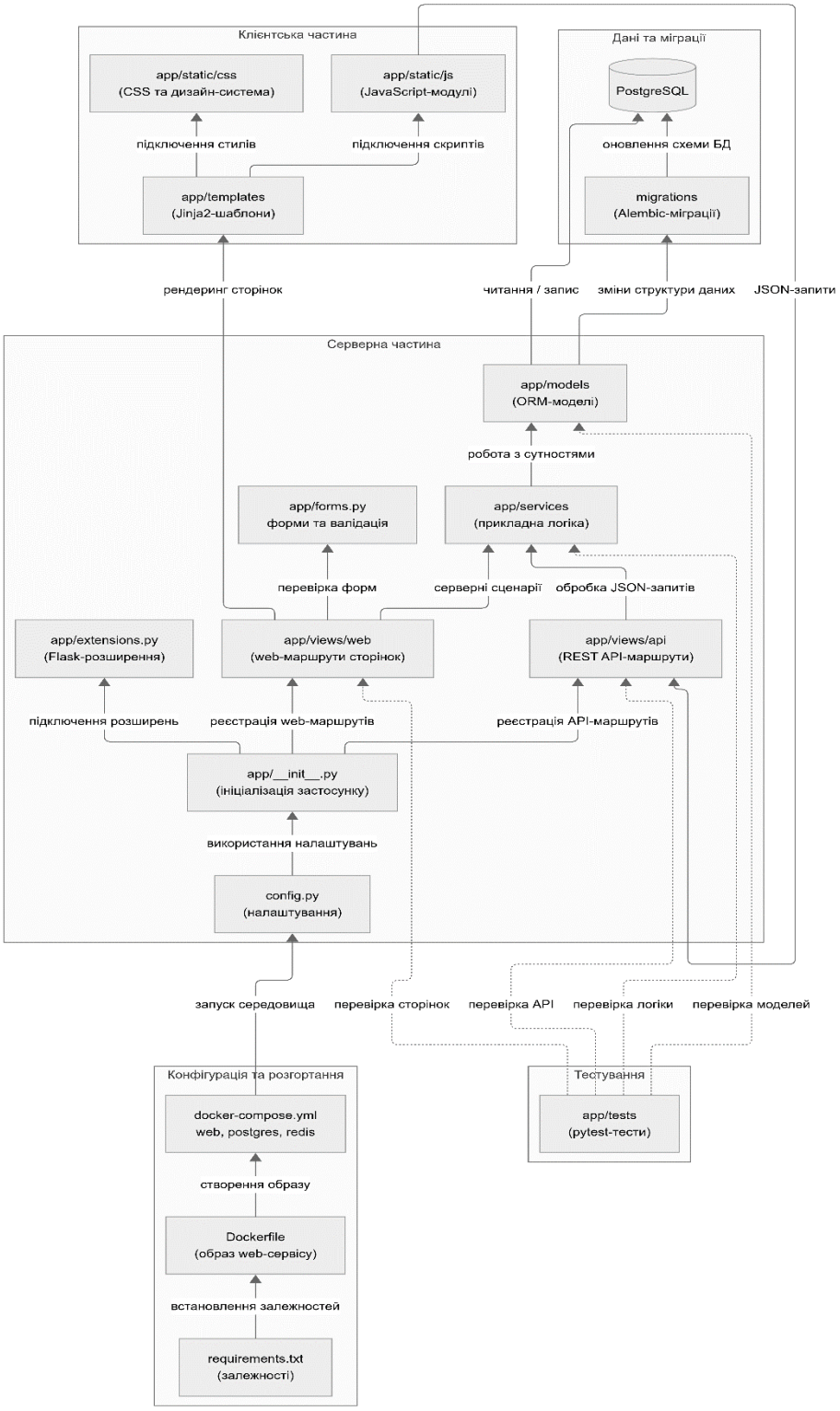


Рис. 4.1 Структура проєкту MiniTrello

4.2 Реалізація авторизації, профілю користувача та налаштувань

Модуль авторизації охоплює реєстрацію, вхід, вихід, сесію користувача, скидання пароля, двофакторну автентифікацію та OAuth-підключення.

Основні дані облікового запису зберігаються в моделі User:

- email;
- хеш пароля;
- профільні поля;
- мова;
- тема інтерфейсу;
- OAuth-дані;
- TOTP-секрет;
- recovery codes.

Паролі не зберігаються у відкритому вигляді. Для них використовується хешування, що зменшує ризик компрометації облікових записів у разі несанкціонованого доступу до бази даних. Під час входу система перевіряє введений пароль, створює сесію користувача і надалі використовує її для доступу до захищених сторінок. Схема автентифікації наведена на рисунку 4.2.

Двофакторна автентифікація реалізується через TOTP-коди. Користувач підтверджує вхід одноразовим кодом із застосунку автентифікації. Recovery codes потрібні для відновлення доступу, якщо користувач утратив доступ до генератора кодів [15; 17; 18].

Профіль користувача містить персональні налаштування, аватар, мову, тему, видимість профілю, дозвіл на запити в друзі та прямі повідомлення. Такі параметри стосуються не тільки зручності, а й контролю соціальної взаємодії всередині системи.

OAuth-підключення використовується для зовнішніх сценаріїв авторизації або інтеграцій. Такі дані мають зберігатися окремо від основної логіки задач, оскільки вони пов’язані з доступом до сторонніх сервісів і потребують додаткового захисту.

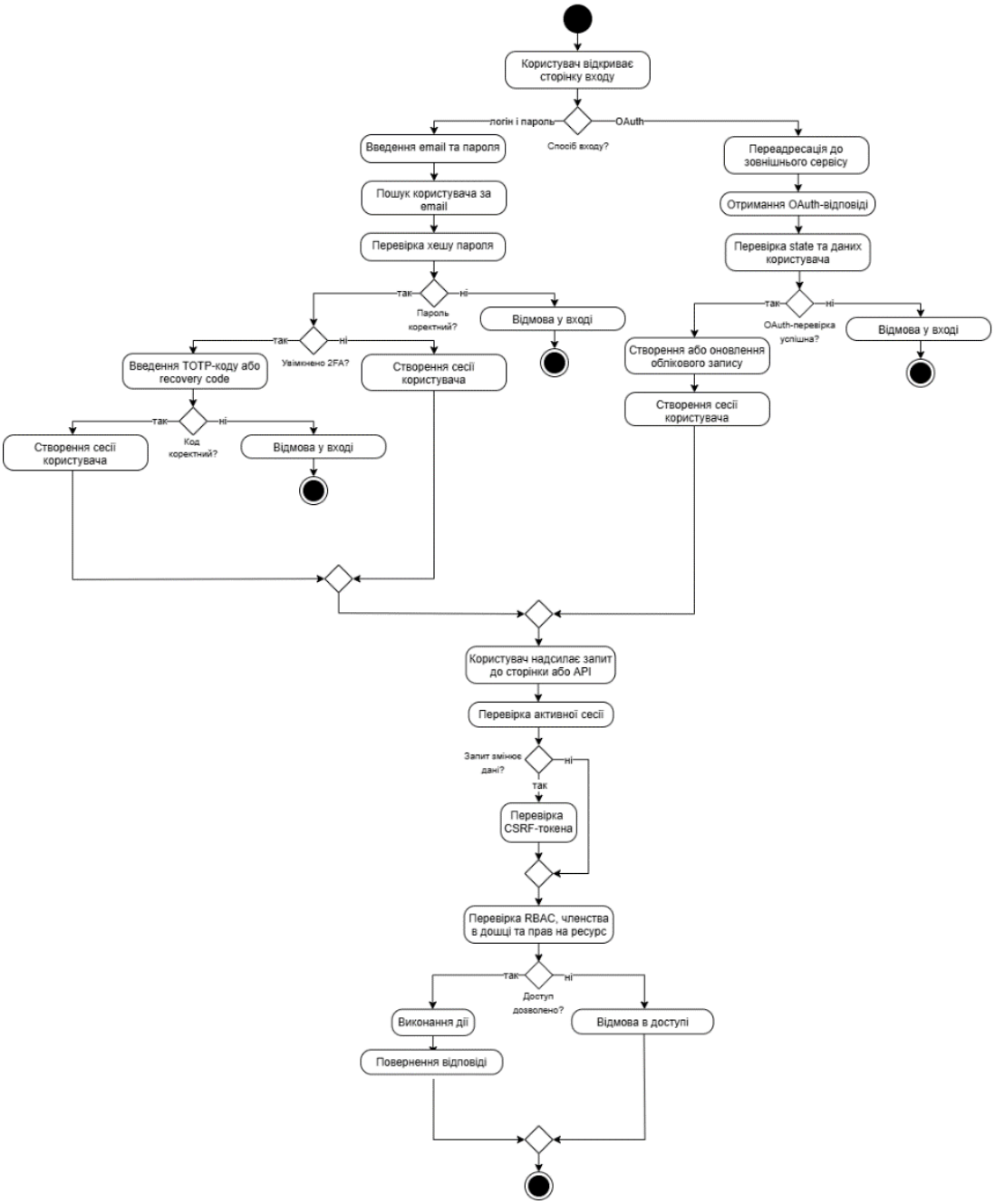


Рис. 4.2 Діаграма діяльності автентифікації та перевірки доступу

4.3 Реалізація дошок, задач, коментарів, вкладень і статусів

Дошка є основним робочим простором, у межах якого користувачі створюють задачі, керують статусами, додають учасників і переглядають аналітику. Для цього використовується модель Board, а участь користувачів у

дошці описується через BoardMember. Така структура дозволяє відокремити саму дошку від ролей і прав конкретних учасників.

Задача реалізована як основна одиниця Kanban-процесу. Вона містить назву, опис, статус, пріоритет, дедлайн, виконавця, мітки, прогрес, чеклист і зв’язок із дошкою. Статус задачі визначає її розміщення в колонці, а додаткові поля допомагають оцінити складність, строк виконання та відповідального учасника.

Операції з дошками і задачами виконуються через API з перевіркою сесії, членства в дошці, ролі користувача та коректності вхідних даних. Це потрібно для того, щоб зміни не виконувалися лише на рівні інтерфейсу. Серверна частина повинна самостійно перевіряти, чи має користувач право створювати, редагувати або видаляти дані.

Коментарі зберігаються окремо від задачі. Вони дозволяють фіксувати обговорення, уточнення та рішення без зміни основного змісту задачі. Вкладення також мають окрему модель і містять дані про файл, автора завантаження та зв’язок із задачею. Це допомагає зберігати робочі матеріали безпосередньо в контексті відповідного доручення.

Зміна статусу є однією з основних дій на Kanban-дошці. Вона оновлює стан задачі, може створювати запис активності, викликати сповіщення або запускати автоматизації. Завдяки цьому переміщення картки між колонками не зводиться до візуальної зміни, а відображається в даних застосунку.

Мітки, чеклисти, залежності та збережені фільтри розширюють роботу із задачами. Мітки допомагають групувати доручення, чеклист деталізує виконання, залежності потрібні для CPM і Gantt-подання, а фільтри спрощують перегляд задач за виконавцем, статусом, строком або пріоритетом.

Для уточнення логіки взаємодії між інтерфейсом користувача, серверною частиною та допоміжними сервісами системи було побудовано діаграму послідовностей, яка надає змогу побачити порядок обробки запиту від моменту введення даних користувачем до відображення нової задачі в інтерфейсі, як наведено на рисунку 4.3.

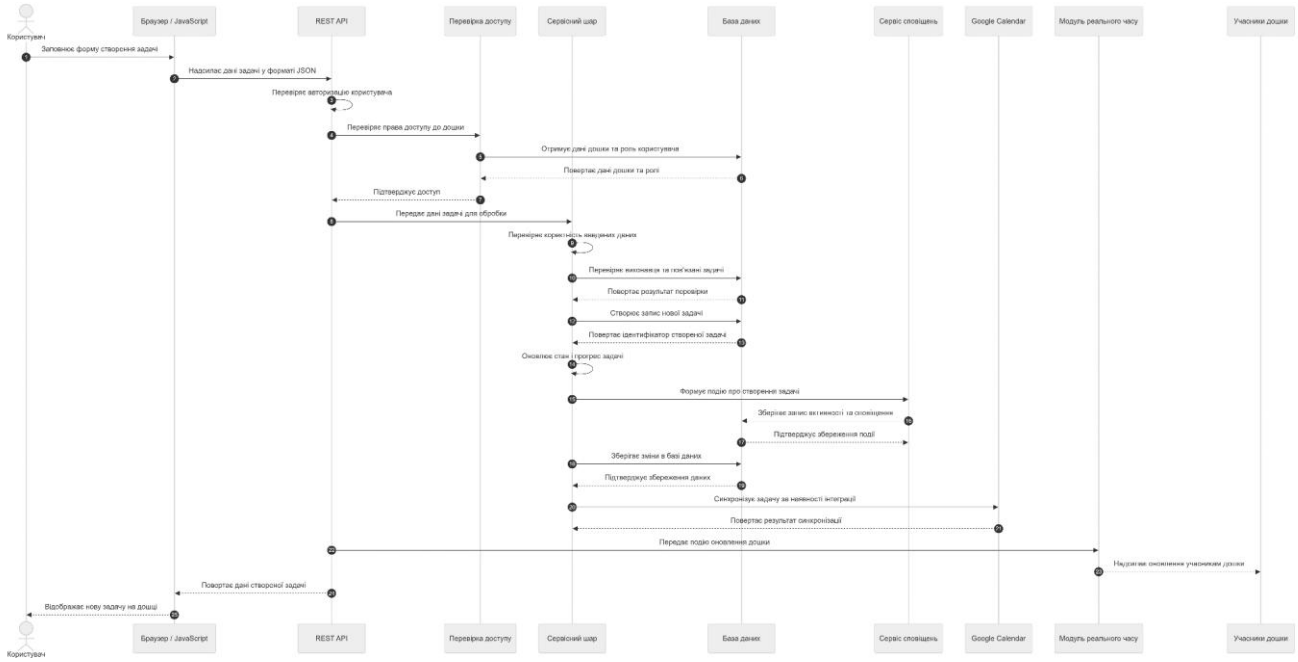


Рис. 4.3 Діаграма послідовностей створення задачі

4.4 Реалізація командної роботи, ролей і запрошень

Командна робота реалізована через членство користувачів у дошках. Модель BoardMember визначає, хто має доступ до конкретного робочого простору і яку роль виконує. Це дозволяє розмежувати права власника, адміністратора, менеджера, учасника та переглядача.

Рольова модель використовується для обмеження дій із задачами, учасниками, запрошеннями, інтеграціями, експортом та аналітикою. Наприклад, власник може керувати дошкою і ролями, менеджер працює із задачами та спринтами, а переглядач має доступ лише до читання. Такий підхід зменшує ризик випадкових або небажаних змін у спільному робочому просторі.

Запрошення до дошки виконуються через токени. InviteToken пов’язує дошку, email або користувача, майбутню роль і службові параметри. Це дозволяє передавати запрошення через email або внутрішні сповіщення, не відкриваючи дошку для всіх користувачів.

Зміна ролей є чутливою операцією, тому вона потребує окремої перевірки прав. Користувач не повинен мати можливість підвищити власні права або змінити

роль власника без відповідних повноважень. Такі перевірки виконуються на серверному рівні, незалежно від того, які елементи доступні в інтерфейсі.

Додатковий соціальний шар реалізовано через прямі повідомлення, дружні зв’язки та публічні профілі. Ці функції не замінюють коментарі до задач, але допомагають організувати взаємодію між користувачами в межах навчальної або малої команди.

4.5 Реалізація спринтів, аналітики, CPM, Gantt, KPI та heatmap

Спринти реалізовані як окрема сутність із назвою, метою, датою початку, датою завершення, статусом і набором пов’язаних задач. Це дозволяє відокремити поточну ітерацію від загального backlog і оцінювати виконання роботи в межах конкретного періоду.

Аналітичні модулі використовують фактичні дані задач: статуси, дедлайни, виконавців, залежності, активність і спринти. Завдяки цьому аналітика не формується вручну, а спирається на дані, які вже створюються під час роботи користувачів із дошкою.

Говорячи про KPI-модулі, то вони показують узагальнений стан роботи: кількість активних і завершених задач, прострочення, ризики, прогрес, проблемні задачі та навантаження команди. Для керівника дошки це дає більше інформації, ніж звичайне розміщення карток за статусами.

Team Overload допомагає оцінити, чи не зосереджено занадто багато задач на окремих учасниках. Для цього враховуються кількість задач, їхні пріоритети, строки та поточний стан. Такий показник корисний для перерозподілу роботи й запобігання перевантаженню.

CPM використовується для визначення критичного шляху на основі залежностей між задачами. Якщо одна задача блокує іншу, її затримка може вплинути на строк завершення всього ланцюжка. Gantt-подання відображає задачі у часовій площині, а heatmap допомагає оцінити інтенсивність активності або навантаження.

4.6 Реалізація автоматизацій, повторюваних задач і rule engine

Автоматизації реалізовані через правила, які описують тригер, умову та дію. Для цього використовуються моделі BoardRule і RuleFiring. Правило визначає, на яку подію потрібно реагувати, а запис виконання допомагає контролювати, чи не була одна й та сама дія виконана повторно для однакової ситуації.

Rule engine може реагувати на зміну статусу, наближення дедлайну, неактивність задачі або інші події на дошці. Дією може бути створення сповіщення, зміна службового поля, підвищення пріоритету або інша операція, передбачена логікою дошки. Такий механізм зменшує кількість ручних дій і допомагає підтримувати стабільність робочого процесу.

Повторювані задачі реалізовані через RecurringTaskTemplate. Шаблон містить параметри майбутньої задачі, правило повторення, активність шаблону і дату останнього створення. Фоновий планувальник перевіряє ці шаблони та створює задачі відповідно до заданого розкладу.

Для таких операцій важлива ідемпотентність. Якщо фоновий процес запускається повторно, система не повинна створювати дублікати задач або сповіщень. Саме тому використовуються службові поля на кшталт last_spawned_date та механізми контролю виконання.

Автоматизації не замінюють ручне керування дошкою, але доповнюють його. Вони корисні для типових повторюваних дій: нагадувань, реакцій на прострочення, створення регулярних задач і повідомлень виконавцям.

4.7 Реалізація інтеграцій: Telegram, Slack, Google Calendar, Web Push, email

Інтеграції винесено в окремі моделі та сервіси, щоб не змішувати зовнішні токени з основною логікою задач. Схема інтеграцій наведена на рисунку 4.3. Для цього використовуються моделі TelegramIntegration, SlackIntegration, GoogleCalendarIntegration, GoogleCalendarTaskSync і PushSubscription. Вони

зберігають параметри підключення, статуси, зовнішні ідентифікатори, токени та службові дані.

Telegram і Slack використовуються для надсилання повідомлень про події дошки. Такі повідомлення можуть стосуватися змін у задачах, наближення строків або інших дій, які потребують уваги учасників команди.

Google Calendar відповідає за синхронізацію задач або дедлайнів із календарем. Для цього важливо зберігати зв’язок між задачею і зовнішньою календарною подією, а також контролювати статус синхронізації та можливі помилки.

Web Push забезпечує браузерні push-сповіщення на основі підписок користувача. Такий механізм корисний для швидкого інформування навіть тоді, коли користувач не перебуває безпосередньо на сторінці дошки.

Email використовується для запрошень і службових повідомлень. Запрошення повинні містити контрольовані посилання з токеном, щоб приєднання до дошки не відбувалося без перевірки.

Токени інтеграцій належать до чутливих даних. Вони не повинні передаватися в клієнтський інтерфейс або зберігатися разом із відкритими даними задач. Помилки зовнішніх сервісів також не повинні порушувати роботу дошки: якщо інтеграція тимчасово недоступна, основні функції застосунку мають залишатися працездатними.

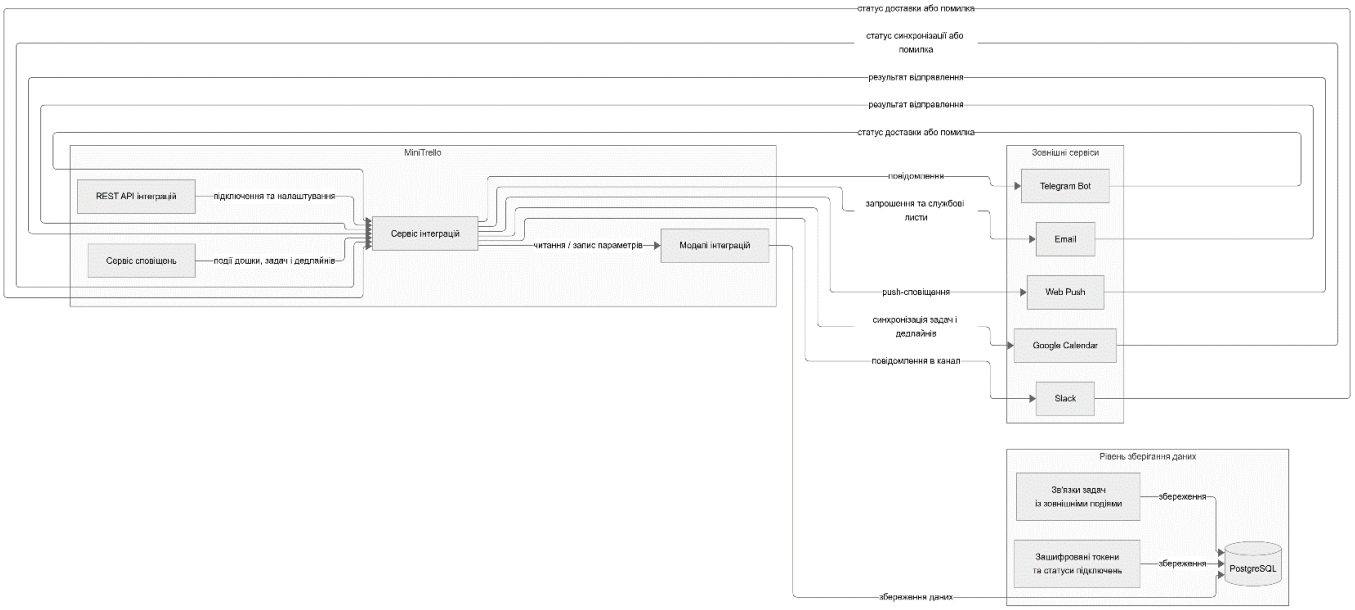


Рис. 4.4 Діаграма взаємодії компонентів інтеграційного модуля

4.8 Реалізація експорту, імпорту та звітності

Експорт потрібний для перенесення даних за межі web-застосунку, підготовки звітів або резервних копій. У проєкті передбачено формати CSV, XLSX та PDF. Для XLSX використовується openpyxl, а для PDF – ReportLab.

CSV підходить для простого табличного вивантаження. XLSX зручний для подальшої роботи з даними в електронних таблицях. PDF використовується для звітних матеріалів, які потрібно передати або зберегти без подальшого редагування.

Звітність має включати основні дані дошки: задачі, статуси, дедлайни, виконавців, пріоритети та інші поля, потрібні для аналізу роботи. Такий експорт зменшує потребу вручну копіювати дані з інтерфейсу в окремі файли.

Операції експорту повинні враховувати права доступу. Користувач не має отримувати дані дошки, до якої він не належить, навіть якщо знає прямий API-маршрут. Тому перед формуванням файлу сервер перевіряє сесію, членство в дошці та роль користувача.

Імпорт, якщо він використовується, потребує перевірки структури вхідних даних. Система має контролювати обов’язкові поля, типи значень, допустимі

статуси, коректність дат і зв’язок із дошкою. Це зменшує ризик пошкодження даних або створення некоректних задач.

4.9 Реалізація захисту даних

Захист даних реалізовано на кількох рівнях: автентифікація користувачів, хешування паролів, захист сесій, CSRF-токени, рольова модель доступу, обмеження частоти запитів, двофакторна автентифікація, recovery codes, OAuth-підключення, захист токенів інтеграцій і контроль доступу до вкладень та API.

Хешування паролів захищає облікові записи від зберігання паролів у відкритому вигляді. Сесії та cookies використовуються для підтримки стану входу користувача, а захищені параметри cookies зменшують ризики небажаного використання сесії.

CSRF-захист застосовується до форм і запитів, які змінюють стан системи. Це важливо для авторизації, профілю, задач, вкладень, ролей і налаштувань дошки. Перевірка CSRF-токена зменшує ризик виконання дії від імені користувача без його наміру.

RBAC-перевірки виконуються для операцій із дошками, задачами, учасниками, вкладеннями, аналітикою та інтеграціями. Факт входу в систему сам по собі не означає, що користувач має право змінювати будь-яку дошку або переглядати чужі дані.

Rate limiting обмежує надмірну кількість запитів до чутливих маршрутів. Це особливо важливо для входу, відновлення доступу, API-операцій і дій, які можуть бути використані для перебору або перевантаження системи.

Двофакторна автентифікація і recovery codes підсилюють захист облікового запису. Навіть якщо пароль став відомий сторонній особі, для входу потрібне додаткове підтвердження.

Окремо захищаються токени інтеграцій. Telegram, Slack і Google Calendar можуть надавати доступ до зовнішніх сервісів, тому такі дані не повинні відкрито передаватися в інтерфейс або зберігатися без контролю. Вкладення також

потребують перевірки доступу: файл задачі не можна віддавати користувачеві, який не має прав на відповідну дошку.

4.10 Інструменти тестування і контролю якості

Для перевірки працездатності web-застосунку використовується pytest [27; 28]. Тестування охоплює основні сценарії роботи, які наведено у таблиці 4.2: авторизацію, API, дошки, задачі, CSRF-захист, обмеження частоти запитів, двофакторну автентифікацію, повторювані задачі, автоматизації, інтеграції, сповіщення, локалізацію, спринти та аналітичні модулі.

Тести потрібні не лише для перевірки успішних сценаріїв, вони також мають контролювати помилки доступу, некоректні дані, відсутність прав, прострочені або недійсні токени, перевищення лімітів запитів і помилки зовнішніх сервісів. Це особливо важливо для системи, де користувачі мають різні ролі, а частина операцій змінює стан дошок, задач або інтеграцій.

Для підготовки тестового середовища використовуються fixtures. Вони дають змогу створювати тестових користувачів, дошки, задачі, ролі та клієнт для надсилання запитів. Завдяки цьому перевірки стають повторюваними: кожен тест отримує контрольовані вхідні дані й може перевірити очікуваний результат.

Окремий напрям становить тестування API. Такі перевірки мають підтверджувати, що маршрути коректно обробляють JSON-запити, повертають очікувані статуси, змінюють дані в базі та правильно реагують на помилки. Для задач, дошок, ролей і коментарів важливо перевіряти не тільки створення або редагування, а й заборонені дії для користувачів без відповідних прав.

Для інтеграцій важливо ізолювати зовнішні сервіси. Тести мають перевіряти поведінку застосунку при успішній відповіді, помилці авторизації, недоступності сервісу або некоректному токені. Такий підхід дозволяє контролювати власну логіку без залежності від фактичної доступності Telegram, Slack, Google Calendar або інших зовнішніх систем.

Контроль якості також підтримується GitHub Actions [1; 29]. CI-сценарій дозволяє запускати перевірки після змін у репозиторії, контролювати синтаксис і зменшувати ризик внесення помилок у вже реалізовані сценарії. Це робить тестування частиною процесу розробки, а не окремою дією наприкінці роботи.

Таблиця 4.2

Напрями тестування веб-застосунку

Напрям	Що перевіряється	Приклад сценарію
API	json-маршрути, статуси відповідей, зміна даних	дошки, задачі, коментарі
Авторизація	сесія користувача, вхід і вихід	login/logout
CSRF-захист	наявність і перевірка токенів	форми та state-changing запити
Rate limiting	обмеження надмірної кількості запитів	захист чутливих маршрутів
2FA	totp і recovery codes	підтвердження входу
RBAC	права доступу за ролями	дії власника, учасника, переглядача
Автоматизації	правила дошки та їх виконання	rule engine
Повторювані задачі	створення задач за шаблоном	recurring tasks
Інтеграції	поведінка при успішних і помилкових відповідях	telegram, slack, google calendar
Локалізація	відображення сторінок різними мовами	публічні сторінки, auth-сторінки

4.11 Тестування web-застосунку

Для контролю якості передбачено автоматизовані тести, які охоплюють API, авторизацію, CSRF-захист, rate limiting, двофакторну автентифікацію, задачі,

дошки, повторювані задачі, rule engine, інтеграції, локалізацію, спринти та аналітичні модулі. Таке тестування дозволяє перевіряти не лише окремі функції, а й типові сценарії взаємодії користувача із системою. Основні напрями автоматизованого тестування та перелік основних тестових сценаріїв наведені на рисунку 4.4 та у таблиці 4.3.

API-тести перевіряють створення, оновлення і видалення сутностей, реакцію на некоректні дані, помилки доступу та правильність відповідей. Для дошок і задач важливо перевіряти не тільки успішні дії, а й ситуації, коли користувач не має потрібної ролі.

Тести авторизації охоплюють вхід, вихід, сесію, захищені маршрути, CSRF і поведінку форм. Для 2FA перевіряються сценарії з правильним кодом, помилковим кодом і recovery codes.

Окремий напрям становить перевірка безпеки. Сюди належать CSRF, rate limiting, RBAC, доступ до вкладень, захист чутливих маршрутів і контроль операцій, які змінюють стан дошки або задачі.

Для інтеграцій важливо ізолювати зовнішні сервіси. Тести мають перевіряти поведінку системи при успішній відповіді, помилці авторизації, некоректному токені або недоступності сервісу. Це дозволяє контролювати власну логіку без залежності від фактичної доступності Telegram, Slack, Google Calendar або Web Push.

Аналітичні модулі перевіряються за даними задач, дедлайнів, залежностей, спринтів і статусів. Для recurring tasks і rule engine важливо контролювати не тільки створення записів, а й уникнення дублювань, коректність переходів станів і роботу фонової логіки.

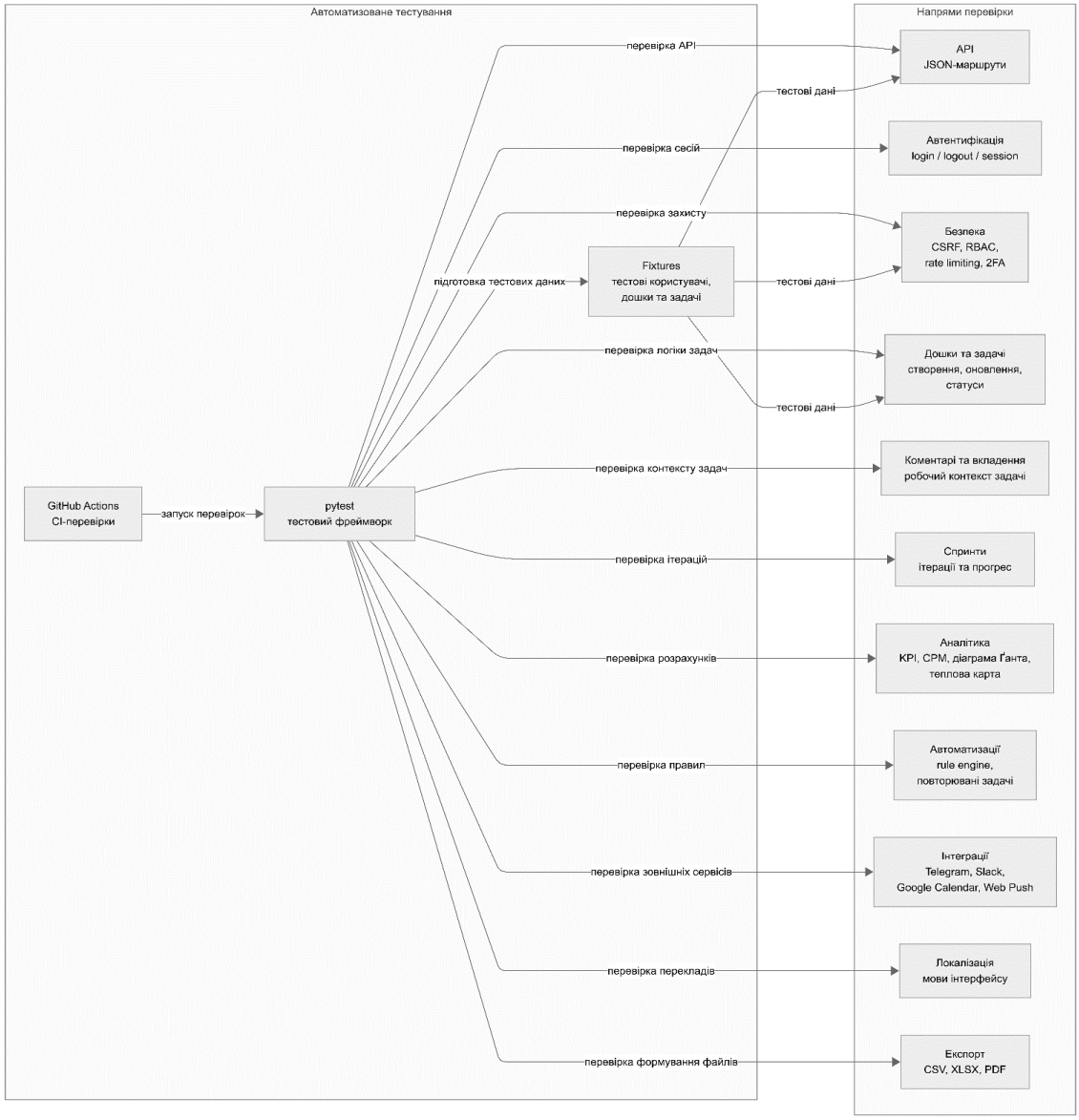


Рис. 4.5 Напрями тестування

Таблиця 4.3

Основні тестові сценарії web-застосунку

№	Сценарій	Передумови	Дії	Очікуваний результат	Напрям
1	Реєстрація користувача	підготовлені коректні дані	надіслати форму реєстрації	створено обліковий запис	auth
2	Вхід до системи	користувач існує	ввести email і пароль	створено сесію користувача	auth

Продовження таблиці 4.3

Основні тестові сценарії web-застосунку

№	Сценарій	Передумови	Дії	Очікуваний результат	Напрям
3	Неправильний пароль	користувач існує	ввести помилковий пароль	доступ відхилено	security
4	CSRF-захист	відсутній або некоректний токен	надіслати state-changing запит	запит відхилено	security
5	Rate limiting	багато повторних запитів	виконати серію запитів	спрацьовує обмеження	security
6	2FA	увімкнена двофакторна автентифікація	ввести totp-код	доступ підтверджено або відхилено	auth
7	Створення дошки	користувач авторизований	створити дошку	дошку збережено	boards
8	Запрошення учасника	користувач має права власника	сформувати запрошення	створено token-запрошення	roles
9	Зміна ролі	учасник уже доданий	змінити роль	роль оновлено або дію заборонено	rbac
10	Створення задачі	користувач має доступ до дошки	надіслати дані задачі	задачу створено	tasks

Продовження таблиці 4.3

Основні тестові сценарії web-застосунку

№	Сценарій	Передумови	Дії	Очікуваний результат	Напрям
11	Зміна статусу	задача існує	перемістити задачу	статус оновлено	kanban
12	Коментар	задача існує	додати коментар	коментар збережено	tasks
13	Вкладення	користувач має доступ	завантажити файл	вкладення додано або відхилено	files
14	Recurring task	існує активний шаблон	запустити фонову перевірку	створено задачу без дублювання	background
15	Rule engine	налаштовано правило	виконати подію-тригер	правило спрацьовує один раз	automation
16	Спринт	створена дошка і задачі	додати задачі до спринту	спринт оновлено	sprints
17	KPI	є задачі зі статусами	запросити показники	повернуто розраховані дані	analytics
18	CPM	є залежності між задачами	побудувати критичний шлях	визначено послідовність задач	analytics
19	Gantt	задачі мають дати	сформувати подання	повернуто часову структуру	analytics

Продовження таблиці 4.3

Основні тестові сценарії web-застосунку

20	Telegram / Slack	інтеграцію налаштовано	надіслати подію	повідомлення сформовано або помилку оброблено	integrations
21	Google Calendar	є задача з дедлайном	синхронізувати календар	створено або оновлено подію	integrations
22	Web Push	існує підписка	створити сповіщення	push-повідомлення підготовлено	notifications
23	Локалізація	вибрано мову	відкрити сторінку	текст відображено потрібною мовою	localization
24	CSV/XLSX/PDF	користувач має доступ до дошки	виконати експорт	файл сформовано	export

4.12 Результати роботи та демонстрація інтерфейсу

Результатом роботи є web-застосунок для управління задачами та візуалізації робочих процесів. Він поєднує Kanban-дошки, задачі, коментарі, вкладення, ролі, запрошення, сповіщення, прямі повідомлення, спринти, аналітику, автоматизації, повторювані задачі, інтеграції та експорт даних.

Реалізація охоплює серверну архітектуру, модель даних, міграції, REST API, клієнтські JavaScript-модулі, шаблони інтерфейсу, механізми безпеки, автоматизовані тести та Docker-розгортання. Це дозволяє розглядати результат не

лише як набір сторінок, а як повноцінний програмний продукт із розділеними рівнями відповідальності.

Демонстрацію інтерфейсу доцільно будувати навколо послідовного сценарію роботи користувача. Спочатку користувач реєструється або входить у систему, після цього створює дошку, додає учасників, формує задачі, призначає виконавців, додає коментарі та вкладення, змінює статуси, переглядає сповіщення, аналізує показники, виконує експорт і за потреби підключає інтеграції.

Головна сторінка, зображена на рисунку 4.6, пояснює призначення застосунку і веде користувача до реєстрації або входу. Сторінки авторизації поєднують форми, CSRF-захист і повідомлення про помилки. Після входу користувач переходить до списку доступних дошок і може створити новий робочий простір, як показано на рисунку 4.7.

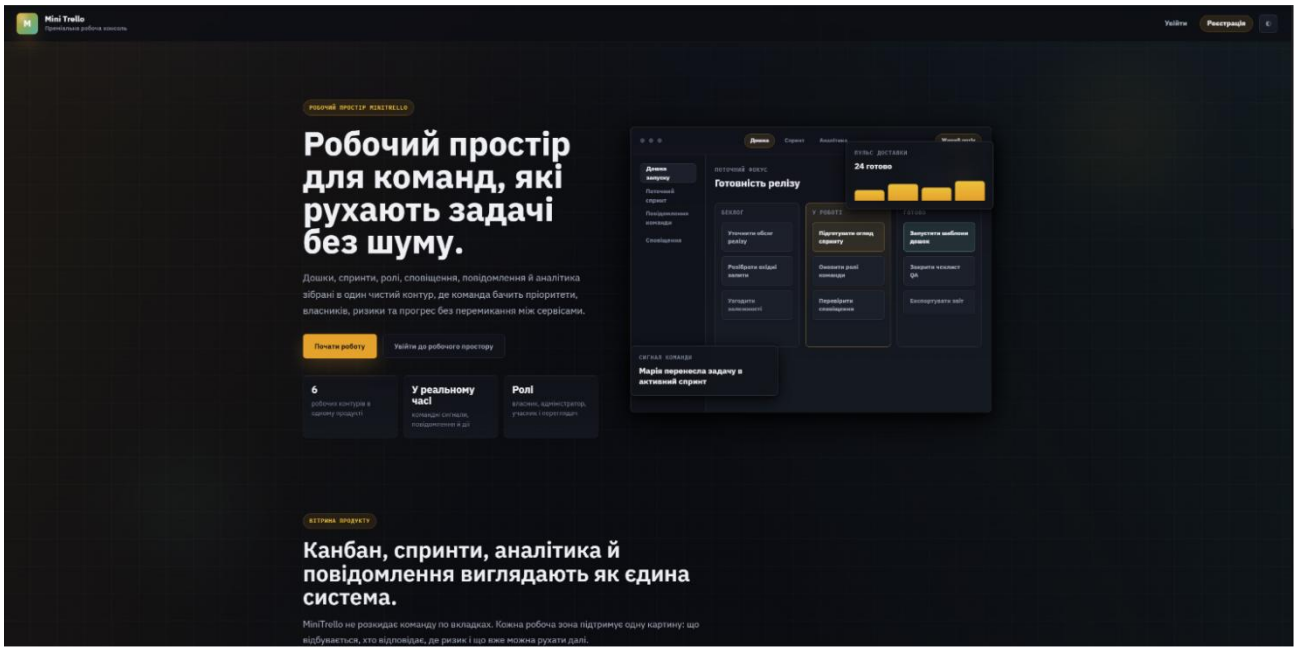


Рис. 4.6 Головна сторінка застосунку

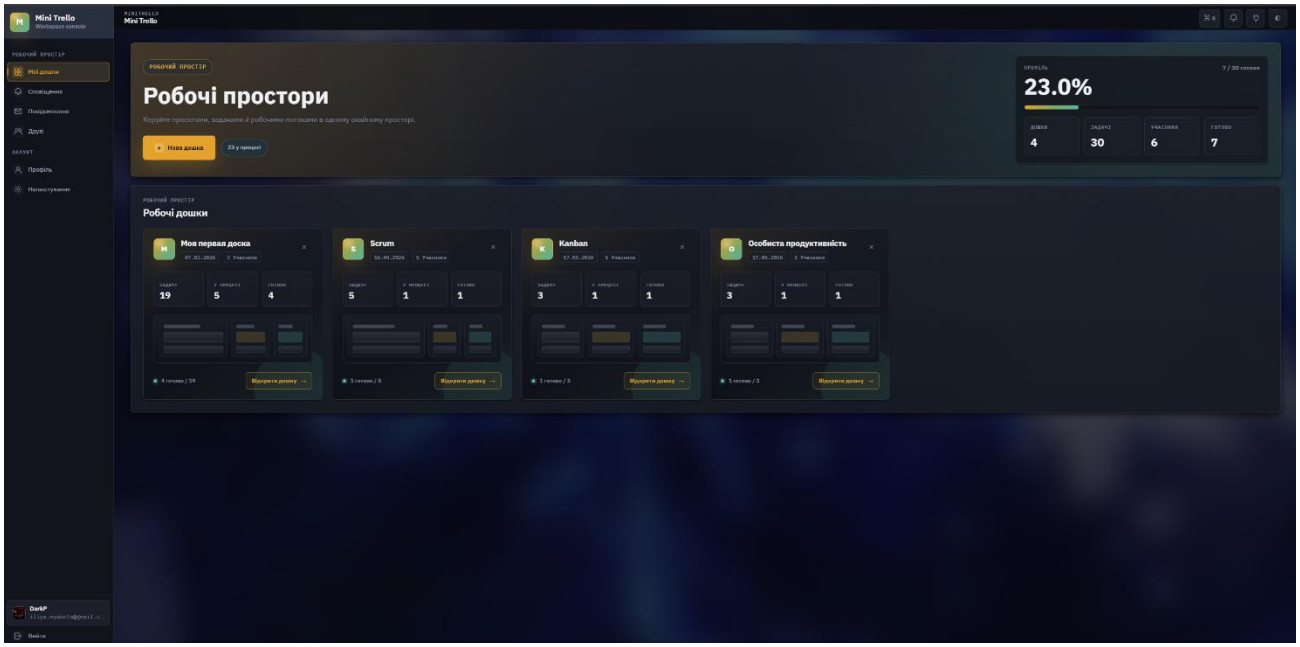


Рис. 4.7 Робочий стіл користувача зі списком усіх його дошок

Kanban-дошка є центральним елементом інтерфейсу. На рисунку 4.8 відображено колонки, картки задач, виконавців, дедлайни, пріоритети, фільтри та модальні форми. Форма задачі, наведена на рисунку 4.9, дає змогу редагувати опис, строк, виконавця, чеклист, мітки, вкладення та інші параметри.

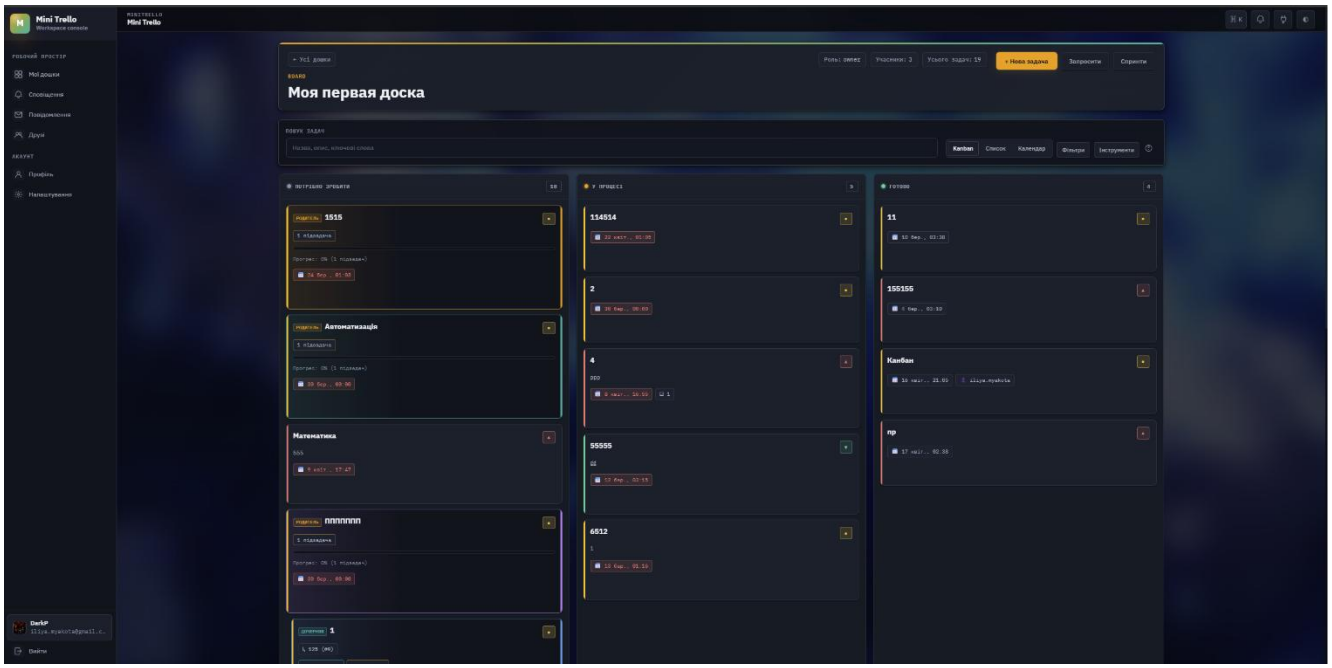


Рис. 4.8 Головний екран Kanban-дошки (з колонками та картками).

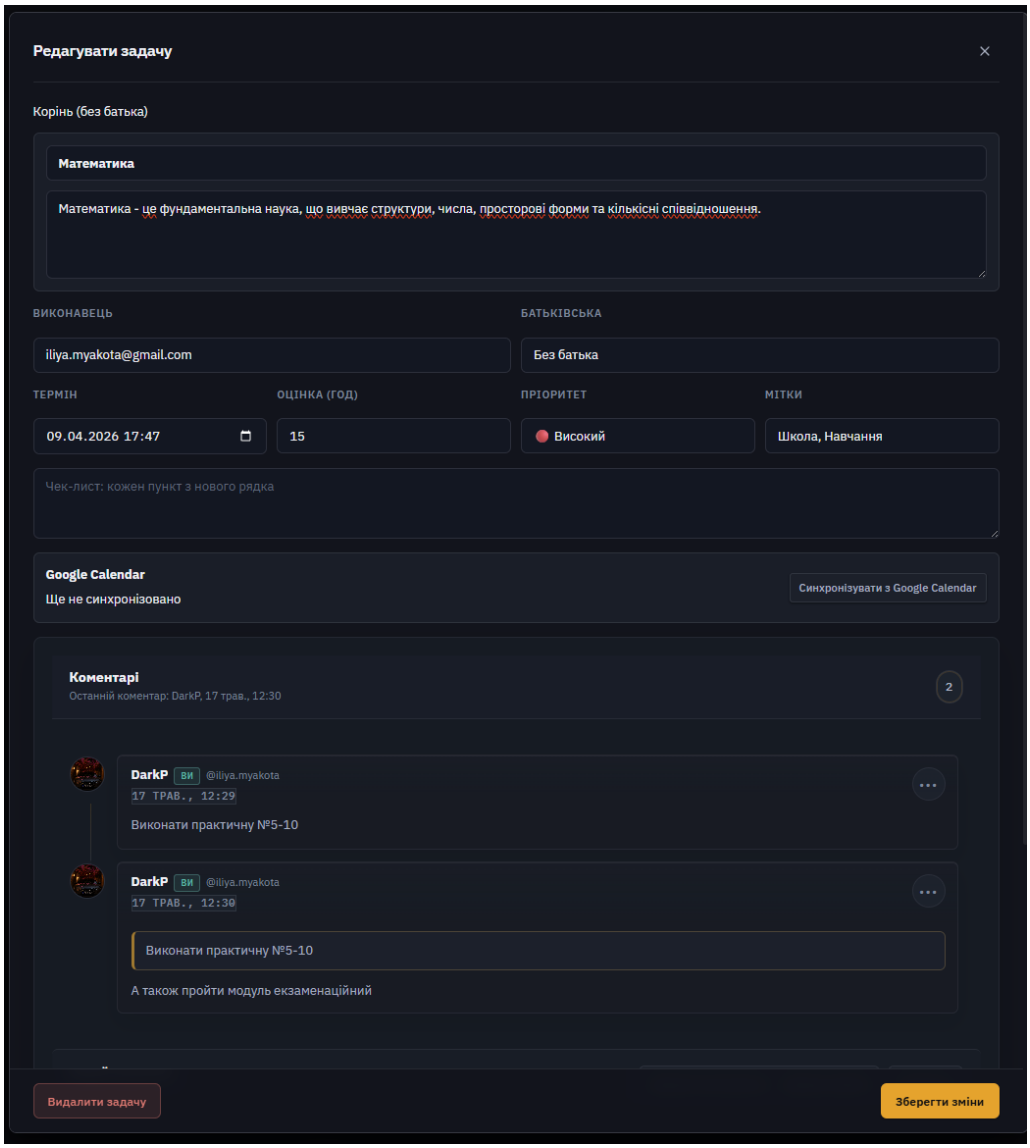


Рис. 4.9 Модальне вікно конкретної задачі

Коментарі й вкладення зберігають робочий контекст задачі. Сповіщення та Web Push інформують користувачів про важливі події без потреби вручну перевіряти кожну дошку. Профіль містить налаштування мови, теми, аватара та параметри безпеки.

Аналітичні екрани, представлені на рисунку 4.10, демонструють KPI, CPM, Gantt, heatmap, спринти та оцінювання перевантаження команди. Експорт у CSV, XLSX або PDF дозволяє підготувати зовнішній звіт за даними дошки. А на рисунку 4.11 продемонстрована вкладка інтеграції з Telegram, Slack, Google Calendar, Web Push і email, які розширюють можливості комунікації та нагадувань.

Рис. 4.10 Екран аналітики KPI

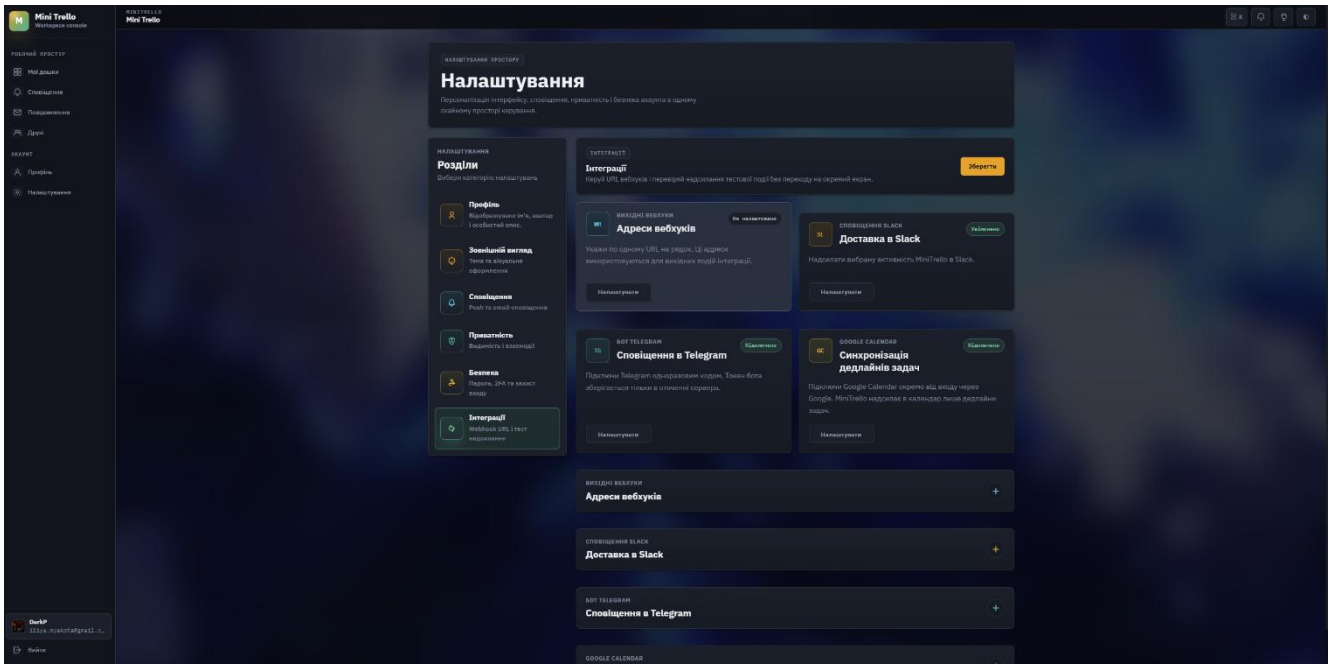


Рис. 4.11 Сторінка Інтеграції

Подальший розвиток доцільно пов’язати з деталізацією журналу аудиту, розширенням прав доступу на рівні окремих дій, підтримкою SSO, створенням мобільного клієнта, оптимізацією аналітики для великих дошок і покращенням документації API.

ВИСНОВКИ

У кваліфікаційній роботі розроблено web-застосунок для управління задачами та візуалізації робочих процесів. У процесі виконання роботи розглянуто особливості командної взаємодії, організації задач, контролю строків, розподілу відповідальних осіб, роботи з вкладеннями, коментарями, статусами та сповіщеннями. Визначено, що для малих команд і навчальних проєктів важливо мати не лише дошку задач, а й механізми контролю доступу, аналітики, автоматизації та зручного розгортання системи.

Під час аналізу програмних аналогів розглянуто Trello, Jira, Asana, Notion і ClickUp. Trello використано як приклад простої карткової моделі, Jira – як систему з розвиненими процесами розробки, Asana – як сервіс для координації командної роботи, Notion – як гнучкий робочий простір із базами даних, а ClickUp – як платформу з багатьма поданнями задач. Порівняння показало, що ці рішення мають широкі можливості, однак значною мірою залежать від SaaS-моделі, тарифних обмежень і закритої внутрішньої реалізації.

На основі аналізу предметної області та аналогів сформовано функціональні й нефункціональні вимоги до розроблюваного застосунку. До функціональних вимог віднесено авторизацію, роботу з дошками й задачами, керування ролями, запрошення, коментарі, вкладення, сповіщення, прямі повідомлення, спринти, аналітику, автоматизації, повторювані задачі, інтеграції та експорт даних. Нефункціональні вимоги охоплюють модульність архітектури, захист даних, тестованість, підтримку подій реального часу, виконання фонових задач і супровідність програмного коду.

Обґрунтовано вибір засобів реалізації. Серверну частину побудовано на Flask, для формування HTML-сторінок використано Jinja2, а клієнтську інтерактивність реалізовано за допомогою Vanilla JavaScript. Для зберігання даних застосовано PostgreSQL, роботу з моделями організовано через SQLAlchemy, а зміни структури бази даних контролюються за допомогою Alembic. Redis

використано для допоміжних інфраструктурних механізмів, Flask-SocketIO – для подій реального часу, APScheduler – для фонових операцій, Docker Compose – для запуску середовища, а pytest і GitHub Actions – для контролю якості.

У межах проєктування визначено загальну архітектуру системи, логічну модель даних, ролі користувачів, основні сценарії використання, структуру Kanban-дошки, процес створення задачі, роботу сповіщень, інтеграційні сценарії та механізми захисту даних. Основу моделі становлять користувачі, дошки, учасники, задачі, коментарі, вкладення, спринти, залежності, правила автоматизації, шаблони повторюваних задач, сповіщення та моделі зовнішніх підключень.

У програмній реалізації поєднано модулі авторизації, профілю користувача, дошок, задач, коментарів, вкладень, ролей, запрошень, спринтів, аналітики, автоматизацій, повторюваних задач, інтеграцій, експорту та захисту даних. Безпека забезпечується хешуванням паролів, захистом сесій, CSRF-токенами, рольовою моделлю доступу, обмеженням частоти запитів, двофакторною автентифікацією, recovery codes, OAuth-підключеннями, контролем токенів інтеграцій і перевіркою доступу до вкладень та API.

Розроблений застосунок пройшов комплексне тестування, яке охоплювало API, дошки, задачі, авторизацію, CSRF-захист, rate limiting, двофакторну автентифікацію, повторювані задачі, rule engine, інтеграції, локалізацію, спринти та аналітичні модулі. Це рішення також дає змогу контролювати не лише окремі функції, а й типові сценарії взаємодії користувача із системою.

Таким чином, проєкт відповідає поставленій меті роботи: процес управління задачами покращено шляхом створення web-застосунку, який об’єднує Kanban-дошки, командну взаємодію, аналітичні модулі, автоматизації, інтеграції та експорт даних. Подальший розвиток доцільно пов’язати з деталізацією журналу аудиту, розширенням прав доступу на рівні окремих дій, підтримкою SSO, створенням мобільного клієнта, оптимізацією аналітики для великих дошок і покращенням документації API.

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

1. М’якота І.А., Замрій І.В. Проєктування та реалізація веб-додатку для управління задачами та візуалізації робочих процесів. Матеріали ІІ Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. – К.: ДУІКТ, 2025, С.80-83.

2. М’якота І.А., Замрій І.В. Забезпечення безпеки даних у web-застосунку mini trello для управління завданнями та командної взаємодії. Матеріали Всеукраїнської науково-практичної конференції «Цифрова трансформація кібербезпеки», 29 квітня 2026 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. – К.: ДУІКТ, 2026, С.53-54.

ПЕРЕЛІК ПОСИЛАНЬ

1. Farley D. Modern Software Engineering: Doing What Works to Build Better Software Faster. Boston: Addison-Wesley Professional, 2021. 256 p.
2. Bass L., Clements P., Kazman R. Software Architecture in Practice. 4th ed. Boston: Addison-Wesley Professional, 2021. 464 p.
3. Ford N., Richards M., Sadalage P., Dehghani Z. Software Architecture: The Hard Parts: Modern Trade-Off Analyses for Distributed Architectures. Sebastopol: O'Reilly Media, 2021. 462 p.
4. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol: O'Reilly Media, 2021. 616 p.
5. A Guide to the Project Management Body of Knowledge (PMBOK Guide). 7th ed. Newtown Square: Project Management Institute, 2021. 370 p.
6. ISO/IEC 25010:2023. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Product quality model. Geneva: International Organization for Standardization, 2023.
7. Flask Documentation. Pallets Projects. URL: https://flask.palletsprojects.com/ (дата звернення: 09.05.2026).
8. SQLAlchemy Documentation. SQLAlchemy. URL: https://docs.sqlalchemy.org/ (дата звернення: 09.05.2026).
9. Alembic Documentation. SQLAlchemy. URL: https://alembic.sqlalchemy.org/ (дата звернення: 09.05.2026).
10. PostgreSQL Documentation. PostgreSQL. URL: https://www.postgresql.org/docs/ (дата звернення: 09.05.2026).
11. Docker Compose Documentation. Docker Docs. URL: https://docs.docker.com/compose/ (дата звернення: 09.05.2026).
12. Ozkan N., Bal S., Gurgen Erdogan T., Gok M. S. Scrum, Kanban or a Mix of Both? A Systematic Literature Review. Annals of Computer Science and Information Systems. 2022. Vol. 30. P. 883-893. URL: https://doi.org/10.15439/2022F143

13. The Kanban Guide. Kanban Guides. URL: https://kanbanguides.org/the-kanban-guide/ (дата звернення: 09.05.2026).

14. Schwaber K., Sutherland J. The Scrum Guide. Scrum Guides. URL: https://scrumguides.org/ (дата звернення: 09.05.2026).

15. Souppaya M., Scarfone K., Dodson D. Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities. NIST Special Publication 800-218. Gaithersburg: National Institute of Standards and Technology, 2022. 42 p. URL: https://doi.org/10.6028/NIST.SP.800-218

16. ISO/IEC 27001:2022. Information security, cybersecurity and privacy protection – Information security management systems – Requirements. Geneva: International Organization for Standardization, 2022.

17. OWASP Top Ten Web Application Security Risks. OWASP Foundation. URL: https://owasp.org/www-project-top-ten/ (дата звернення: 09.05.2026).

18. OWASP Session Management Cheat Sheet. OWASP Cheat Sheet Series. URL: https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html (дата звернення: 09.05.2026).

19. Trello Pricing. Atlassian. URL: https://trello.com/pricing (дата звернення: 09.05.2026).

20. Jira Pricing. Atlassian. URL: https://www.atlassian.com/software/jira/pricing (дата звернення: 09.05.2026).

21. Asana Pricing. Asana. URL: https://asana.com/pricing (дата звернення: 09.05.2026).

22. Getting started with projects and tasks. Notion Help Center. URL: https://www.notion.com/help/guides/getting-started-with-projects-and-tasks (дата звернення: 09.05.2026).

23. Automations feature availability and limits. ClickUp Help Center. URL: https://help.clickup.com/hc/en-us/articles/23477062949911-Automations-feature-availability-and-limits (дата звернення: 09.05.2026).

24. Flask-SocketIO Documentation. Read the Docs. URL: https://flask-socketio.readthedocs.io/ (дата звернення: 09.05.2026).

25. Redis Documentation. Redis. URL: https://redis.io/docs/ (дата звернення: 09.05.2026).

26. APScheduler Documentation. Read the Docs. URL: https://apscheduler.readthedocs.io/ (дата звернення: 09.05.2026).

27. ISO/IEC/IEEE 29119-1:2022. Software and systems engineering – Software testing – Part 1: General concepts. Geneva: International Organization for Standardization, 2022.

28. pytest Documentation. pytest. URL: https://docs.pytest.org/ (дата звернення: 09.05.2026).

29. GitHub Actions Documentation. GitHub Docs. URL: https://docs.github.com/actions (дата звернення: 09.05.2026).

30. ISO/IEC/IEEE 42010:2022. Software, systems and enterprise – Architecture description. Geneva: International Organization for Standardization, 2022.

31. Weflen E., MacKenzie C. A., Rivero I. V. An influence diagram approach to automating lead time estimation in Agile Kanban project management. Expert Systems with Applications. 2022. Vol. 187. Article 115866. URL: https://doi.org/10.1016/j.eswa.2021.115866

32. Senapathi M., Drury-Grogan M. L. Systems Thinking Approach to Implementing Kanban: A case study. Journal of Software: Evolution and Process. 2021. Vol. 33, no. 4. Article e2322. URL: https://doi.org/10.1002/smr.2322

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛ

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

Розробка web-застосунку для управління задачами та візуалізацією робочих процесів

Виконав студент 4 курсу
групи ПД-44
М’якота Ілля Андрійович
Керівник роботи
д-р тех.наук, проф., зав. кафедри ІПЗ Замрій Ірина Вікторівна

Київ – 2026

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи - покращення процесу управління задачами та візуалізацією робочих процесів шляхом розробки web-застосунку.

Об’єкт дослідження - процес управління задачами та візуалізацією робочих процесів у командній роботі.

Предмет дослідження - методи та програмні засоби розробки web-застосунку для управління задачами та візуалізації процесів.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Проаналізувати предметну область управління задачами.
2. Провести порівняльний аналіз аналогів і наявних вебсистем.
3. Визначити функціональні та нефункціональні вимоги до MiniTrello.
4. Обґрунтувати вибір технологічного стеку (Python, Flask, PostgreSQL).
5. Спроєктувати архітектуру, модель даних і основні сценарії взаємодії користувачів.
6. Реалізувати вебзастосунок із використанням обраних технологій.
7. Провести тестування застосунку.

3

АНАЛІЗ АНАЛОГІВ

Характеристики порівняння	Trello	Jira	Asana	Notion	ClickUp	MiniTrello
Організація задач	картки, списки, дошки	Scrum / Kanban-проєкти	задачі та проєкти	сторінки, бази даних, подання	простори, списки, задачі	дошки, колонки, задачі
Kanban-дошки	основний режим роботи	налаштовуються через робочі процеси	доступні як один із режимів	через подання бази даних	доступні як окреме подання	основний робочий простір
Коментарі та вкладення	є в картках задач	є в задачах і підзадачах	є в задачах і проєктах	є в сторінках і записах	є в задачах	є в задачах
Спринти	через шаблони або Power-Ups	вбудована Scrum-логіка	через проєктні налаштування	через ручне налаштування баз	підтримуються у робочих процесах	реалізована sprint-модель
Аналітика та звітність	через Power-Ups або зовнішні сервіси	звіти, burndown, velocity	панелі звітності	через таблиці, фільтри й подання	аналітичні панелі та звіти	KPI, CPM, діаграма Ґанта, теплова карта
Автоматизація процесів	Butler-правила	правила автоматизації	правила та інтеграції	через шаблони та інтеграції	правила автоматизації	Rule engine
Ролі та доступ	залежить від робочого простору й тарифу	детальна модель прав	ролі на рівні проєкту	ролі на рівні робочого простору	ролі на рівні простору й проєкту	RBAC для дошок
Інтеграції	Power-Ups	Marketplace	інтеграції з робочими сервісами	інтеграції робочого простору	інтеграції з зовнішніми сервісами	Telegram, Slack, Google Calendar

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

1) Реєстрація, вхід і керування профілем користувача.
2) Створення, редагування, видалення та налаштування дошок.
3) Створення, редагування, переміщення, архівування й видалення задач.
4) Призначення виконавців, пріоритетів і строків виконання.
5) Керування статусами задач через kanban-дошку.
6) Коментарі, вкладення, історія змін і запрошення учасників.
7) Розмежування доступу між власником, адміністратором і учасниками.
8) Сповіщення про зміни, дедлайни та важливі події.
9) Підтримка спринтів, повторюваних задач і автоматизацій.
10) Формування аналітики, KPI, звітів та експорту даних.
11) Налаштування зовнішніх інтеграцій, сповіщень і календарних сервісів.

Нефункціональні вимоги:

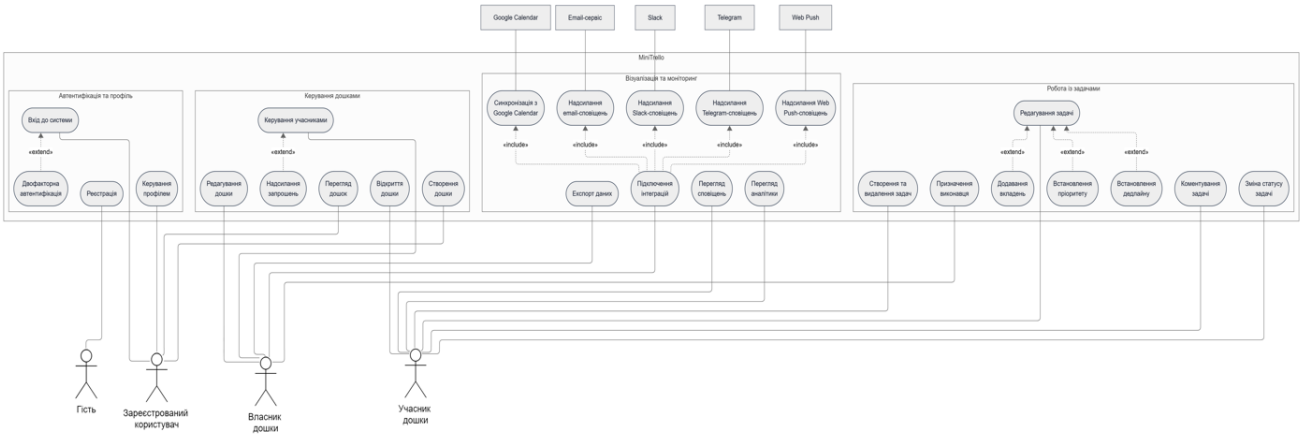
1) Контроль доступу через автентифікацію, авторизацію та ролі.
2) Збереження даних у реляційній базі з підтримкою зв'язків і транзакцій.
3) Модульна структура серверної частини, API та клієнтських сценаріїв.
4) Зрозумілий інтерфейс:
   - створення задачі – до 3 кроків,
   - зміна статусу – 1 drag-and-drop,
   - доступ до функцій – до 2 переходів.
5) Стабільна робота:
   - відповідь сторінок і API – до 1 с,
   - збереження змін – до 1 с,
   - команда – до 10 користувачів на дошці.
6) Можливість розширення функцій через сервісний шар.
7) Підтримка фонових процесів для задач, автоматизацій і сповіщень.
8) Коректна локалізація інтерфейсу кількома мовами:
   - українська,
   - англійська.
9) Захист від несанкціонованого доступу, CSRF, помилок введення та витоку даних.

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

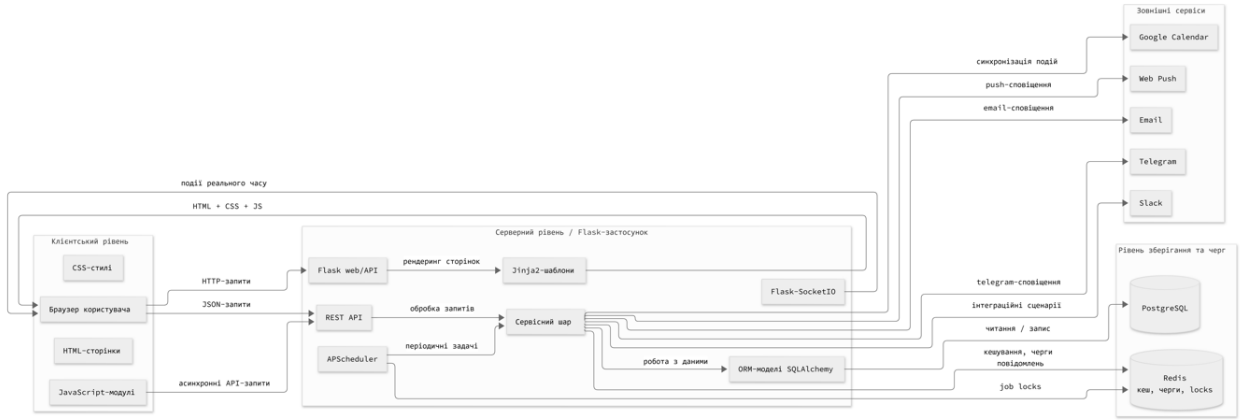
6

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ



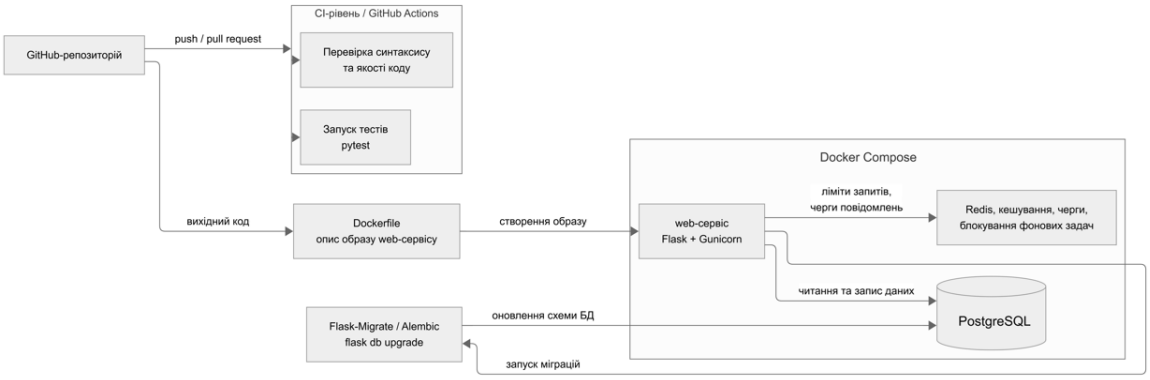
7

АРХІТЕКТУРА ПРОЄКТУ



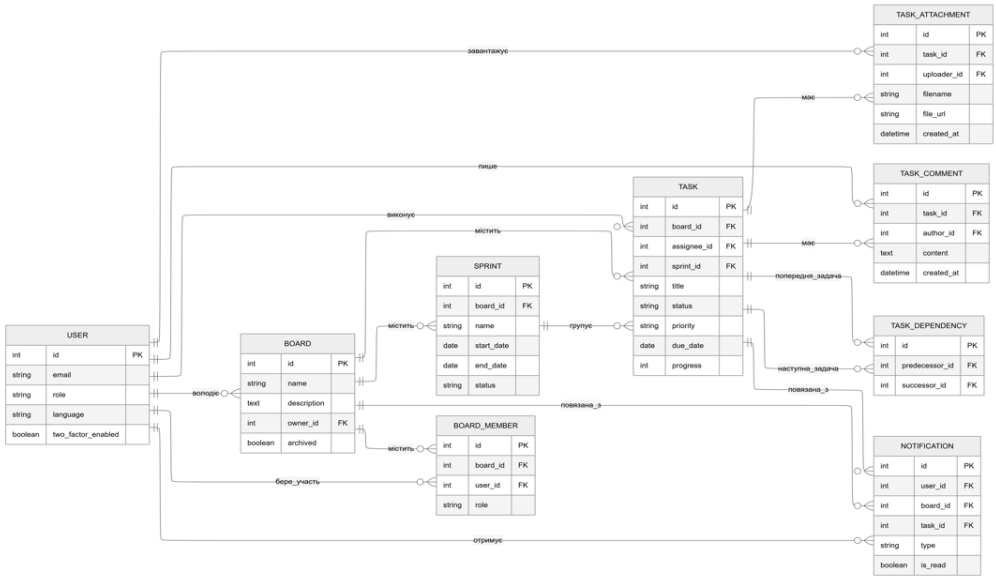
8

СХЕМА CI ТА DOCKER-РОЗГОРТАННЯ



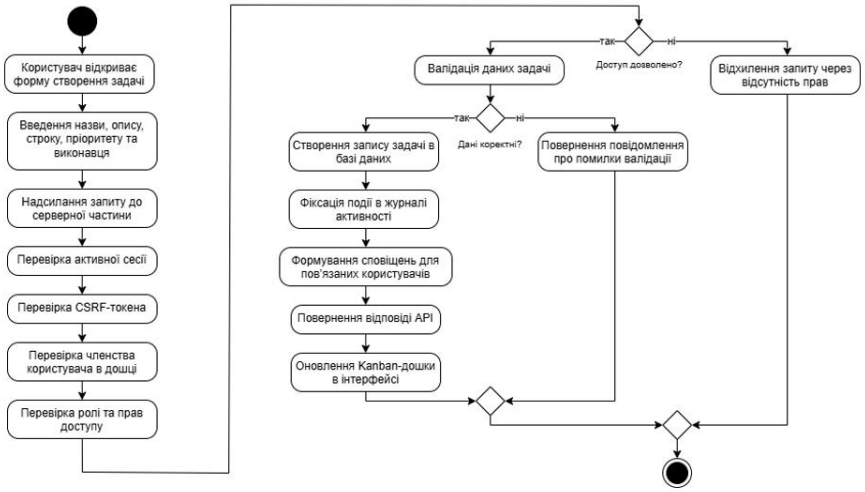
9

ER-ДІАГРАМА СУТНОСТЕЙ



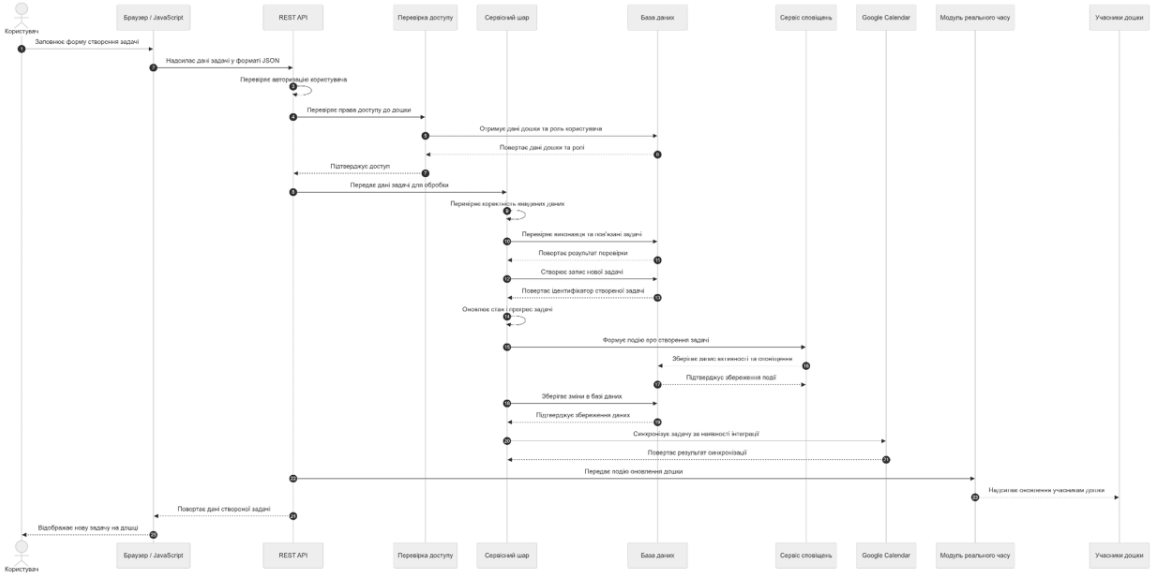
10

ДІАГРАМА АКТИВНОСТІ СТВОРЕННЯ ЗАДАЧІ



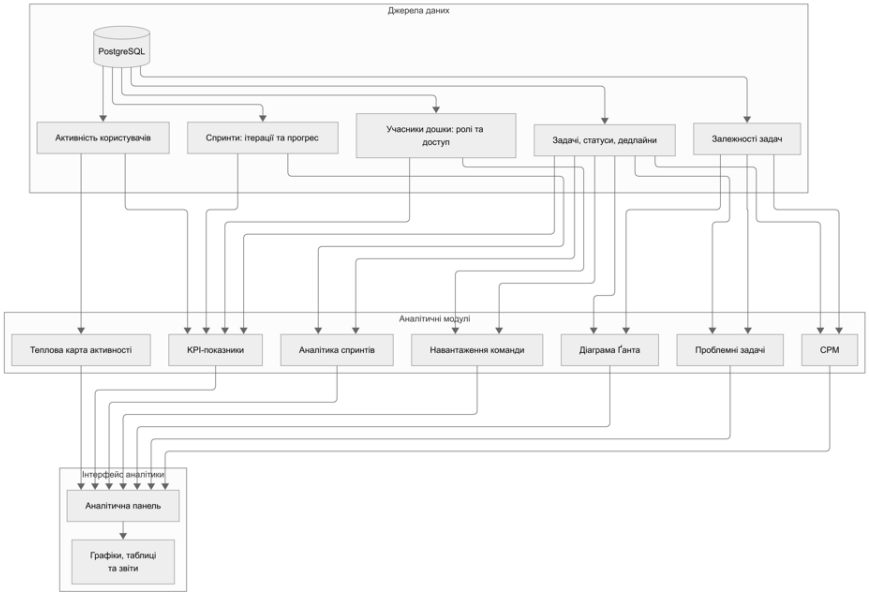
11

ДІАГРАМА ПОСЛІДОВНОСТЕЙ СТВОРЕННЯ ЗАДАЧІ



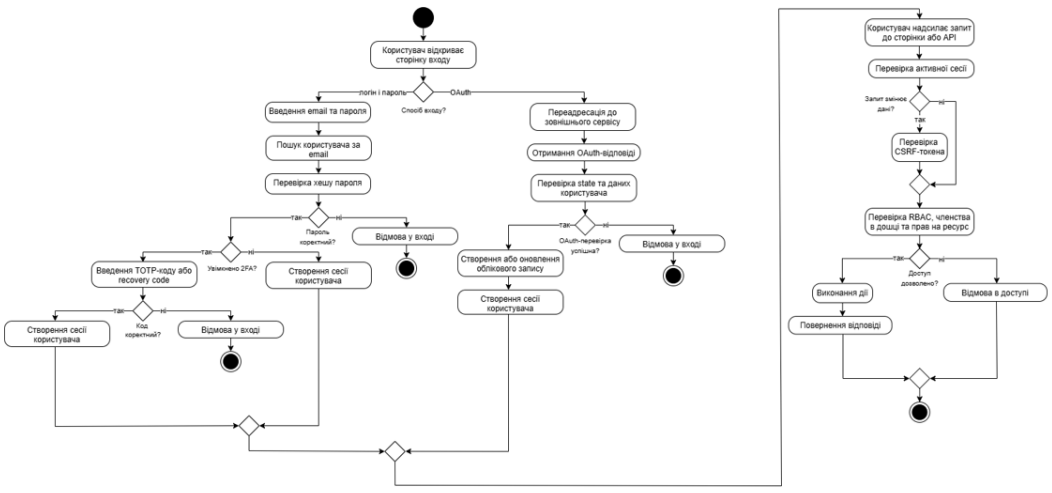
12

ДІАГРАМА КОМПОНЕНТІВ АНАЛІТИЧНОЇ ПІДСИСТЕМИ



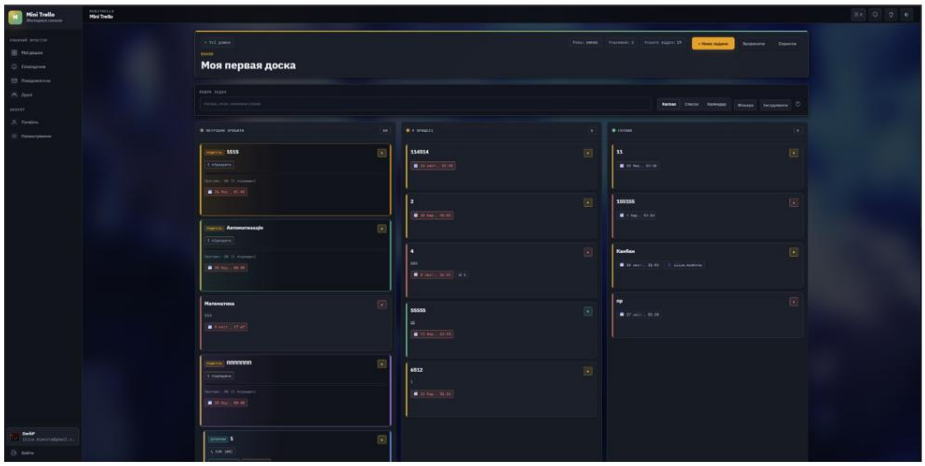
13

ДІАГРАМА ДІЯЛЬНОСТІ АВТЕНТИФІКАЦІЇ ТА ПЕРЕВІРКИ ДОСТУПУ



14

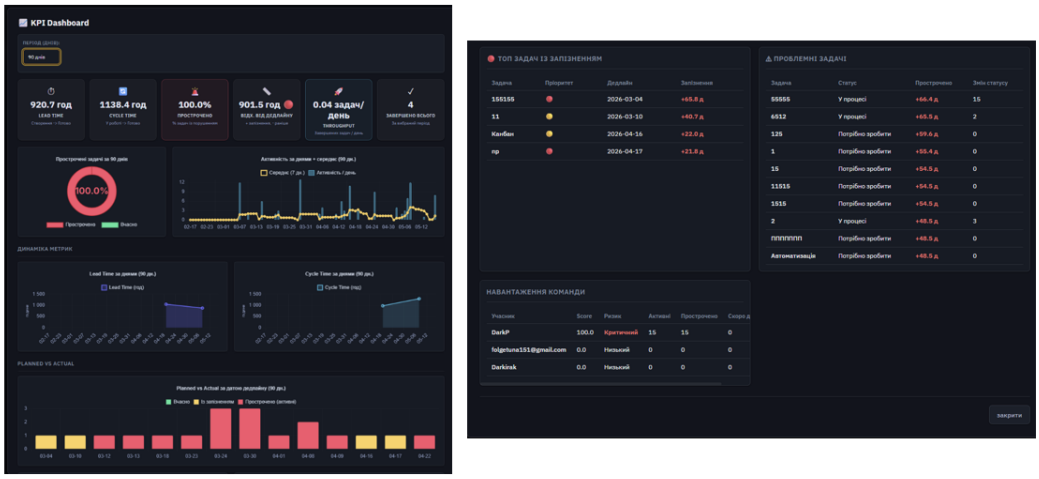
ЕКРАННІ ФОРМИ



Робочий простір користувача

15

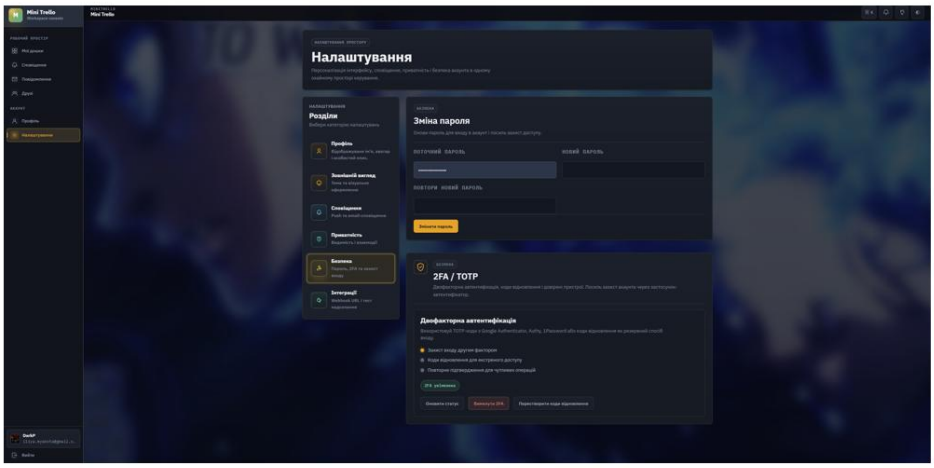
ЕКРАННІ ФОРМИ



Графічне представлення динаміки показників ефективності роботи команди

16

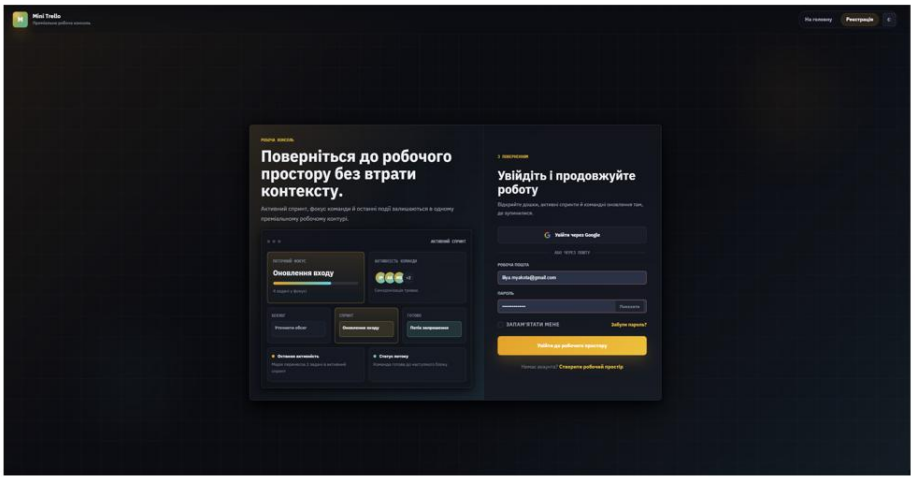
ЕКРАННІ ФОРМИ



Панель налаштувань безпеки: керування обліковими даними та двофакторною автентифікацією (2FA)

17

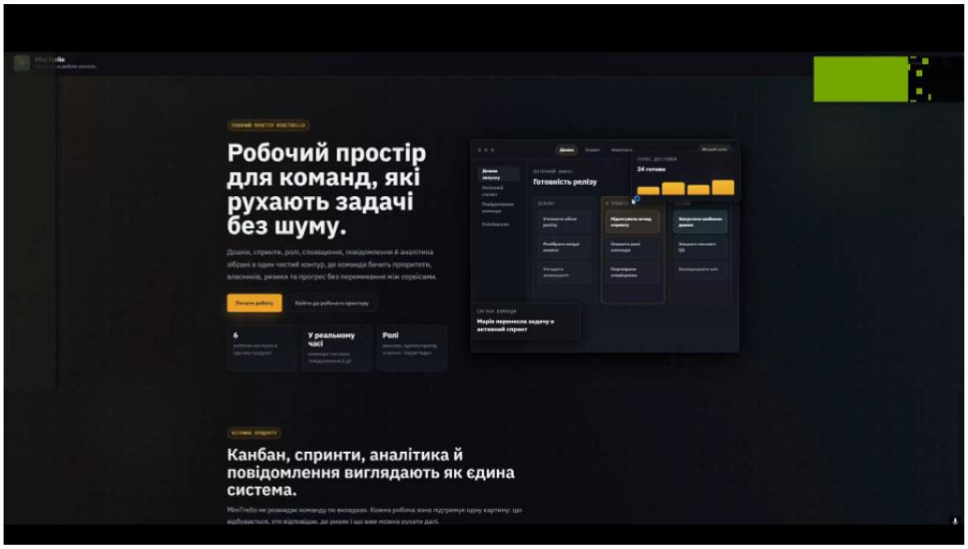
ЕКРАННІ ФОРМИ



Інтерфейс модуля автентифікації та авторизації користувачів

18

Демонстрація застосунку



19

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. М’якота І.А., Замрій І.В. Проєктування та реалізація веб-додатку для управління задачами та візуалізації робочих процесів. Матеріали ІІ Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. – К.: ДУІКТ, 2025, С.80-83.

2. М’якота І.А., Замрій І.В. Забезпечення безпеки даних у web-застосунку mini trello для управління завданнями та командної взаємодії. Матеріали Всеукраїнської науково-практичної конференції «Цифрова трансформація кібербезпеки», 29 квітня 2026 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. – К.: ДУІКТ, 2026, С.53-54.

20

ВИСНОВКИ

1. Проаналізовано предметну область управління задачами та проведено порівняння існуючих вебсистем, зокрема Jira, Trello, Asana, Notion і ClickUp. На основі цього визначено визначено функціональні і нефункціональні вимоги до MiniTrello.
2. Обґрунтовано вибір технологічного стеку Python, Flask і PostgreSQL та спроєктовано архітектуру застосунку, реляційну модель даних і основні сценарії взаємодії користувачів.
3. Реалізовано web-застосунок MiniTrello, який підтримує Kanban-дошки, керування задачами, спринти, ролі доступу, коментарі, вкладення, аналітику та інтеграційні можливості.
4. Проведено тестування розробленого рішення, за результатами якого підтверджено коректну роботу основних функцій застосунку, зокрема авторизації, керування дошками й задачами, сповіщень, аналітичних модулів, інтеграцій, CSRF-захисту та рольової моделі доступу.

ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

Лістинг Б.1 – Ініціалізація Flask-застосунку та підключення модулів
Файл: app/__init__.py

```
def create_app(config_class: type = Config) -> Flask:
    app = Flask(__name__)
    app.config.from_object(config_class)

    @app.after_request
    def set_csp(response):
        csp_policy = (
            "default-src 'self'; "
            "script-src 'self' 'unsafe-inline' 'unsafe-eval' "
            "https://cdn.jsdelivr.net https://unpkg.com
https://cdn.socket.io "
            "https://accounts.google.com; "
            "style-src 'self' 'unsafe-inline' "
            "https://cdn.jsdelivr.net https://fonts.googleapis.com; "
            "font-src 'self' https://fonts.gstatic.com data:; "
            "img-src 'self' data: blob:
https://lh3.googleusercontent.com; "
            "connect-src 'self' https://cdn.jsdelivr.net
https://unpkg.com "
            "https://cdn.socket.io ws: wss:; "
            "frame-src https://accounts.google.com;"
        )
        response.headers["Content-Security-Policy"] = csp_policy
        return response

    app.config.setdefault("SESSION_COOKIE_HTTPONLY",
True)
    app.config.setdefault("SESSION_COOKIE_SAMESITE",
"Lax")
    app.config.setdefault("SESSION_COOKIE_SECURE",
False)

    @app.before_request
    def persist_requested_language():
        requested_lang = (request.args.get("lang") or
"").strip().lower()
        if requested_lang in TRANSLATIONS:
            session[LANGUAGE_SESSION_KEY] =
requested_lang

    db.init_app(app)
    login_manager.init_app(app)
    migrate.init_app(app, db)
    socketio_options = {"cors_allowed_origins": "*"}
    socketio_message_queue =
app.config.get("SOCKETIO_MESSAGE_QUEUE")
    if socketio_message_queue:
        socketio_options["message_queue"] =
socketio_message_queue
    socketio.init_app(app, **socketio_options)
    mail.init_app(app)

    limiter.init_app(app)
    csrf.init_app(app)

    _init_error_handlers(app)

    _init_oauth(app)
    _init_swagger(app)
    _init_telegram_cli(app)

    from app.views.api.web_push import api_web_push_bp
    app.register_blueprint(main_bp)
    app.register_blueprint(auth_bp, url_prefix="/auth")
    app.register_blueprint(api_v1_bp, url_prefix="/api/v1")
    app.register_blueprint(api_tasks_bp)
    app.register_blueprint(api_analytics_bp)
    app.register_blueprint(api_boards_bp)
    app.register_blueprint(api_sprints_bp)
    app.register_blueprint(api_admin_bp)
    app.register_blueprint(api_profile_bp)
    app.register_blueprint(api_notification_center_bp)
    app.register_blueprint(api_friends_bp)
    app.register_blueprint(api_messages_bp)
    app.register_blueprint(api_recurring_tasks_bp)
    app.register_blueprint(api_rules_bp)
    app.register_blueprint(api_integrations_bp)
    app.register_blueprint(api_web_push_bp)
```

Лістинг Б.2 – Моделі дошки, учасника дошки та задачі
Файл: app/models/board.py; app/models/task.py

```
class Board(db.Model):
    __tablename__ = "board"

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(200), nullable=False)
    owner_id = db.Column(
        db.Integer,
        db.ForeignKey("user.id"),
        nullable=False,
        index=True,
    )
    created_at = db.Column(db.DateTime,
default=datetime.utcnow, nullable=False)

    tasks = db.relationship(
        "Task",
        backref="board",
        lazy="selectin",
        cascade="all, delete-orphan",
    )

    members = db.relationship(
        "BoardMember",
        backref="board",
        lazy="selectin",
        cascade="all, delete-orphan",
    )

    activities = db.relationship(
        "Activity",
        backref="board_ref",
        cascade="all, delete-orphan",
    )

class BoardMember(db.Model):
    __tablename__ = "board_member"

    id = db.Column(db.Integer, primary_key=True)
    board_id = db.Column(

class Task(db.Model):
    __tablename__ = "task"

    id = db.Column(db.Integer, primary_key=True)

    title = db.Column(db.String(200), nullable=False)
```

```
    description = db.Column(db.Text, nullable=True)
    status = db.Column(db.String(20), nullable=False,

default="TODO", index=True)

    due_date = db.Column(db.DateTime, nullable=True,
index=True)
    priority = db.Column(db.String(10), nullable=False,
default="medium", index=True)
    labels = db.Column(db.String(255), nullable=True)
    checklist = db.Column(db.Text, nullable=True)

    created_at = db.Column(
        db.DateTime,
        default=datetime.utcnow,
        nullable=False,
        index=True,
    )
    updated_at = db.Column(
        db.DateTime,
        default=datetime.utcnow,
        onupdate=datetime.utcnow,
        nullable=False,
    )

    user_id = db.Column(
        db.Integer,
        db.ForeignKey("user.id"),
        nullable=False,
        index=True,
    )
    board_id = db.Column(
        db.Integer,
        db.ForeignKey("board.id"),
        nullable=False,
        index=True,
    )
    assignee_id = db.Column(
        db.Integer,
        db.ForeignKey("user.id"),
        nullable=True,
        index=True,
    )

    comments = db.relationship(
        "TaskComment",
        backref="task",
        lazy="selectin",
        cascade="all, delete-orphan",
    )

    activities = db.relationship(
        "Activity",
        backref="task",
        lazy="selectin",
        cascade="all, delete-orphan",
    )

    attachments = db.relationship(
        "TaskAttachment",
        backref="task",
        lazy="selectin",
        cascade="all, delete-orphan",
    )
```

Лістинг Б.3 – Модель користувача, пароль і двофакторна автентифікація

Файл: app/models/user.py

```
class UserRole(str, enum.Enum):
    VIEWER = "viewer"
    MANAGER = "manager"
    ADMIN = "admin"

@login_manager.user_loader
def load_user(user_id: str) -> User | None:

    return db.session.get(User, int(user_id))

class User(UserMixin, db.Model):
    __tablename__ = "user"

    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(120), unique=True,
nullable=False, index=True)

    password_hash = db.Column(db.String(255), nullable=True)

    created_at = db.Column(db.DateTime,
default=datetime.utcnow, nullable=False)

    display_name = db.Column(db.String(100), nullable=True)
    bio = db.Column(db.Text, nullable=True)
    avatar_url = db.Column(db.String(500), nullable=True)
    profile_banner_url = db.Column(db.String(500),
nullable=True)

    timezone = db.Column(db.String(50), nullable=False,
default="UTC", server_default="UTC")
    language = db.Column(db.String(10), nullable=False,
default="ru", server_default="ru")
    theme = db.Column(db.String(20), nullable=False,
default="dark", server_default="dark")

    role = db.Column(
        db.Enum(UserRole, name="user_role_enum",
values_callable=lambda x: [e.value for e in x]),
        nullable=False,
        default=UserRole.MANAGER,
        server_default=UserRole.MANAGER.value,
        index=True,
    )

    oauth_provider = db.Column(db.String(30), nullable=True,
index=True)
    oauth_id = db.Column(db.String(255), nullable=True,
index=True)

    password_reset_token = db.Column(db.String(100),
nullable=True, unique=True, index=True)

        back_populates="user",
        uselist=False,
        lazy="selectin",
        cascade="all, delete-orphan",
    )
    google_calendar_task_syncs = db.relationship(
        "GoogleCalendarTaskSync",
        back_populates="user",
        lazy="selectin",
        cascade="all, delete-orphan",
    )
    telegram_integration = db.relationship(
        "TelegramIntegration",
        back_populates="user",

        self.password_reset_token = None
        self.password_reset_expires = None
```

```
    def generate_totp_secret(self) -> str:
        secret = pyotp.random_base32()
        self.totp_secret = secret
        self.totp_enabled = False

    return secret

    def get_totp_uri(self, issuer: str = "Mini Trello") -> str |
None:
        if not self.totp_secret:
            return None
        return
pyotp.TOTP(self.totp_secret).provisioning_uri(name=self.emai
l, issuer_name=issuer)

    def verify_totp(self, code: str) -> bool:
        if not self.totp_secret:
            return False
        code = (code or "").strip().replace(" ", "")
        if not code:
            return False
        return bool(pyotp.TOTP(self.totp_secret).verify(code,
valid_window=1))

    def generate_recovery_codes(self, count: int | None = None)
-> list[str]:
        if count is None:
            try:
                count =
int(current_app.config.get("TOTP_RECOVERY_CODES_CO
UNT", 10))
            except Exception:
                count = 10

        plain_codes: list[str] = []
        hashed_codes: list[str] = []

        for _ in range(max(1, count)):
            raw = secrets.token_hex(4).upper()
            code = f"{raw[:4]}-{raw[4:]}"
            plain_codes.append(code)
            hashed_codes.append(generate_password_hash(code))

        self.totp_recovery_codes = hashed_codes
```

Лістинг Б.4 – Рольова модель доступу до дошок і вкладень

Файл: app/services/rbac.py

```
    return decorator

def require_manager(fn: Callable) -> Callable:
    return api_require_role("manager")(fn)

def require_admin(fn: Callable) -> Callable:
    return api_require_role("admin")(fn)

def viewer_readonly(fn: Callable) -> Callable:
    @functools.wraps(fn)
    def wrapper(*args, **kwargs):
        if not current_user.is_authenticated:
            return jsonify({"error": "Unauthorized"}), 401

        current_role = getattr(current_user, "role", None)
        current_role_value = current_role.value if
hasattr(current_role, "value") else str(current_role)

        if current_role_value == UserRole.VIEWER.value and
request.method != "GET":
            return jsonify({"error": "Forbidden: viewer role is read-
only"}), 403
        return fn(*args, **kwargs)

    return wrapper
def get_board_member(board_id: int, user=None) ->
BoardMember | None:
    user = user or current_user
    if not user.is_authenticated:
        return None
    return BoardMember.query.filter_by(board_id=board_id,
user_id=user.id).first()

def get_board_role(board_id: int, user=None) -> str | None:
    user = user or current_user

    if not user.is_authenticated:
        return None

    if has_global_role(UserRole.ADMIN.value, user):
        return "owner"

    member = get_board_member(board_id, user)
    return member.role if member else None

def has_board_role(board_id: int, required_role: str,
user=None) -> bool:
    actual = get_board_role(board_id, user)
    if actual is None:
        return False
    return BOARD_ROLE_ORDER.get(actual, 0) >=
BOARD_ROLE_ORDER.get(required_role, 0)

def _extract_board_id(board_or_id) -> int:
    if isinstance(board_or_id, int):
        return board_or_id

def is_board_member(board_id: int, user_id: int) -> bool:
    return BoardMember.query.filter_by(board_id=board_id,
user_id=user_id).first() is not None

def can_manage_members(board_id: int, user=None) -> bool:
    return has_board_role(board_id, "admin", user)

def can_manage_sprints(board_id: int, user=None) -> bool:
    return has_board_role(board_id, "admin", user)

def can_create_tasks(board_id: int, user=None) -> bool:
    return has_board_role(board_id, "member", user)

def can_edit_task(board_id: int, task=None, user=None) ->
bool:
    user = user or current_user

    if has_board_role(board_id, "admin", user):
        return True

    if not has_board_role(board_id, "member", user):
        return False

    if task is None:
        return True

    return getattr(task, "user_id", None) == user.id or
getattr(task, "assignee_id", None) == user.id
```

Лістинг Б.5 – API створення задачі та зміни її статусу

Файл: app/views/api/tasks.py

```
def create_task():
    data = request.get_json(silent=True) or {}
```

```
    board_id = data.get("board_id")
    if not board_id:
        return json_error("board_id is required", 400)
    board_ = get_board_or_404(int(board_id))
    require_board_role(board_.id, "member")

    title = (data.get("title") or "").strip()
    if not title:
        return json_error("Title is required", 400)

    description = (data.get("description") or "").strip() or None
    due_date = parse_due_date(data.get("due_date"))

    priority = (data.get("priority") or "medium").lower()
    if priority not in ("low", "medium", "high"):
        priority = "medium"

    labels = (data.get("labels") or "").strip() or None
    checklist_items = data.get("checklist") or []

    assignee_id = data.get("assignee_id")
    assignee = None
    if assignee_id:
        assignee = db.session.get(User, int(assignee_id))
        if not assignee or not is_board_member(board_.id,
assignee.id):
            return json_error("assignee must be a board member",
400)

    parent_id = data.get("parent_id")
    parent = None
    if parent_id:
        parent = db.session.get(Task, int(parent_id))
        if not parent or parent.board_id != board_.id:
            return json_error("parent task not found", 404)

    estimate = data.get("estimate_hours")
    if estimate is not None and str(estimate).strip() != "":
        try:
            estimate = float(estimate)
        except (ValueError, TypeError):
            estimate = None
    else:
        estimate = None

    task = Task(
        title=title,
        description=description,
        status="TODO",
        user_id=current_user.id,
        board_id=board_.id,
        due_date=due_date,
        priority=priority,
        labels=labels,
        checklist=checklist_from_list(checklist_items),
        assignee=assignee,
        parent=parent,
        estimate_hours=estimate,
    )

    auto_set_due_date(task)
    validate_parent_assignment(task, parent)

def update_task_status(task_id: int):
    task = db.session.get(Task, task_id)
    if not task:
        return json_error("not found", 404)

    board_ = get_board_or_404(task.board_id)
    require_board_role(board_.id, "member")

    if task.children and len(task.children) > 0:
        return json_error("Parent task status is derived from
subtasks", 400)

    data = request.get_json(silent=True) or {}
    new_status = data.get("status")
    if new_status not in ("TODO", "IN_PROGRESS",
"DONE"):
        return json_error("Invalid status", 400)

    old_status = task.status
    task.status = new_status

    if new_status == "DONE" and old_status != "DONE":
        task.done_at = datetime.utcnow()
    elif new_status != "DONE" and old_status == "DONE":
        task.done_at = None

    recompute_task_leaf_progress(task)
    recompute_ancestors(task)

    log_activity(
        board_id=board_.id,
        action="task_moved",
        user_id=current_user.id,
        task_id=task.id,
        details=_task_event_details(task, from_status=old_status,
to_status=new_status),
        notify_users=_task_notification_recipients(task),
    )

    db.session.commit()
    best_effort_sync_task(current_user.id, task)
    emit_task_moved(board_.id, task.id, old_status, new_status)
    return jsonify({"message": "ok"})
```

Лістинг Б.6 – Розрахунок показника перевантаження команди

Файл: app/services/analytics.py

```
def get_team_overload(board_id: int, window_days: int = 30) -
> dict:
    now = datetime.utcnow()
    due_soon_until = now + timedelta(days=3)

    members =
BoardMember.query.filter_by(board_id=board_id).all()
    rows: dict[int, dict] = {}
    for member in members:
        user = member.user
        rows[member.user_id] = {
            "user_id": member.user_id,
            "email": user.email if user else "",
            "display": user.display_name if user and getattr(user,
"display_name", None) else (user.email if user else ""),
            "role": member.role,
            "active_tasks": 0,
            "in_progress_tasks": 0,
            "high_priority_tasks": 0,
            "overdue_tasks": 0,
            "due_soon_tasks": 0,
            "estimated_hours": 0.0,
        }

    active_tasks = Task.query.filter(
        Task.board_id == board_id,
        Task.status.in_(("TODO", "IN_PROGRESS")),
    ).all()

    for task in active_tasks:
```

```
        owner_id = task.assignee_id or task.user_id
        if owner_id not in rows:
            continue

        row = rows[owner_id]
        row["active_tasks"] += 1

if task.status == "IN_PROGRESS":
            row["in_progress_tasks"] += 1
        if (task.priority or "").lower() == "high":
            row["high_priority_tasks"] += 1

        due = _naive(task.due_date) if task.due_date else None
        if due and now > due:
            row["overdue_tasks"] += 1
        elif due and now <= due <= due_soon_until:
            row["due_soon_tasks"] += 1

        if task.estimate_hours:
            row["estimated_hours"] += float(task.estimate_hours)

    for row in rows.values():
        raw_score = (
            row["active_tasks"] * 10
            + row["in_progress_tasks"] * 5
            + row["high_priority_tasks"] * 8
            + row["overdue_tasks"] * 20
            + row["due_soon_tasks"] * 10
            + min(row["estimated_hours"] * 2, 40)
        )
        score = min(100, round(raw_score, 1))
        if score >= 85:
            risk = "critical"
        elif score >= 65:
            risk = "high"
        elif score >= 35:
            risk = "medium"
        else:
            risk = "low"

        row["estimated_hours"] = round(row["estimated_hours"],
1)
        row["score"] = score
        row["risk"] = risk

    items = sorted(rows.values(), key=lambda item:
(item["score"], item["active_tasks"]), reverse=True)
```

Лістинг Б.7 – Виконання правил автоматизації задач
Файл: app/services/rule_engine.py

```
_PRIORITY_UPGRADE: dict[str, str] = {
    "low": "medium",
    "medium": "high",
}

class RuleEngineService:

    JOB_ID = "rule_engine_evaluation"
    _DEFAULT_INTERVAL_MINUTES = 30

    def __init__(self, app: "Flask") -> None:
        self._app = app

        self._TRIGGERS: dict[str, str] = {
            "deadline_approaching":
"_trigger_deadline_approaching",
            "task_stalled": "_trigger_task_stalled",
            "status_is": "_trigger_status_is",
        }
        self._ACTIONS: dict[str, str] = {
            "escalate_priority": "_action_escalate_priority",
            "notify_assignee": "_action_notify_assignee",
            "move_to_status": "_action_move_to_status",
        }

    def run(self) -> None:
        with self._app.app_context():
            self._process()

    def evaluate_rule(
        self,
        rule: BoardRule,
        *,
        today: date | None = None,
    ) -> dict:

        today = today or datetime.now(timezone.utc).date()

        trigger_name = self._TRIGGERS.get(rule.trigger_type)
        action_name = self._ACTIONS.get(rule.action_type)

        if trigger_name is None:
            logger.warning(
                "RuleEngineService: unknown trigger_type=%r for
rule id=%d",
                rule.trigger_type, rule.id,
            )
            return {"checked": 0, "fired": 0, "error": f"unknown
trigger_type: {rule.trigger_type!r}"}

        if action_name is None:
            logger.warning(
                "RuleEngineService: unknown action_type=%r for
rule id=%d",
                rule.action_type, rule.id,
            )
            return {"checked": 0, "fired": 0, "error": f"unknown
action_type: {rule.action_type!r}"}

        trigger_fn = getattr(self, trigger_name)
        action_fn = getattr(self, action_name)

        candidates = self._get_candidate_tasks(rule)
        fired = 0

        for task in candidates:
            if self._already_fired(rule.id, task.id, today):
                continue

            if not trigger_fn(rule, task):
                continue

            try:
                with db.session.begin_nested():
                    action_fn(rule, task)
                    db.session.add(
                        RuleFiring(
                            rule_id=rule.id,
                            task_id=task.id,
                            fired_date=today,
                        )
                    )
                    db.session.flush()
                fired += 1
            except Exception:
                logger.exception(
                    "RuleEngineService: failed to fire rule_id=%d for
task_id=%d",
                    rule.id, task.id,
```

```
            )

        return {"checked": len(candidates), "fired": fired, "error":
None}

    def _action_escalate_priority(self, rule: BoardRule, task:
Task) -> None:
        old_priority = (task.priority or "medium").lower()

new_priority = _PRIORITY_UPGRADE.get(old_priority)

        if new_priority is None:
            return

        original_due = task.due_date

        task.priority = new_priority

        if original_due and original_due.tzinfo is not None:
            task.due_date = original_due.replace(tzinfo=None)
        else:
            task.due_date = original_due

        db.session.flush()

        try:
            log_activity(
                board_id=task.board_id,
                action="rule_escalated_priority",
                user_id=None,
                task_id=task.id,
                details={
                    "rule_id": rule.id,
                    "rule_name": rule.name,
                    "trigger": rule.trigger_type,
                    "old_priority": old_priority,
                    "new_priority": new_priority,
                    "task_title": task.title,
                },
                notify_users=list({u for u in [task.assignee_id,
task.user_id] if u}),
            )
        except Exception:
```

Лістинг Б.8 – Клієнтська логіка переміщення карток Kanban-дошки
Файл: app/static/js/boards/kanban.js

```
  kanban.updateTaskStatusAPI = async function
updateTaskStatusAPI(taskId, newStatus) {
    ensureApi();
    return api.fetchJSON(`/api/v1/tasks/${taskId}/status`, {
      method: "PATCH",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ status: newStatus }),
    });
  };

  function normalizeColumnKey(value) {
    const raw = String(value ||
"").trim().toLowerCase().replace(/-/g, "_");

  kanban.initDragAndDrop = function initDragAndDrop() {
    if (typeof window.Sortable === "undefined") return;

    Object.keys(kanban.COLUMNS_MAP).forEach((colId) =>
{
      const el = $(colId);
      if (!el) return;

      new window.Sortable(el, {
        group: "kanban-board",
        animation: 150,
        delay: 100,
        delayOnTouchOnly: true,
        ghostClass: "sortable-ghost",
        filter: '[data-has-children="1"]',

        onMove: (evt) => {
          return (
            evt.related?.dataset?.hasChildren !== "1" &&
            evt.dragged?.dataset?.hasChildren !== "1"
          );
        },

        onEnd: async (evt) => {
          const itemEl = evt.item;
          const newColId = evt.to?.id;
          const oldColId = evt.from?.id;
          const taskId = itemEl?.dataset?.id;
          const newStatus =
kanban.COLUMNS_MAP[newColId];

          if (!taskId || !newStatus || newColId === oldColId)
return;

          const revert = () => {
            if (evt.oldIndex >= 0 && evt.from) {
              const children = Array.from(evt.from.children);
              const beforeNode = children[evt.oldIndex] || null;
              evt.from.insertBefore(itemEl, beforeNode);
            } else if (evt.from) {
              evt.from.appendChild(itemEl);
            }
          };

          try {
            await kanban.updateTaskStatusAPI(taskId, newStatus);
            const task = (state.lastTasks || []).find((t) =>
Number(t.id) === Number(taskId));
            if (task) {
              task.status = newStatus;
            }
            window.dispatchEvent(new
CustomEvent("minitrello:tasks:changed", {
              detail: { board_id: state.CURRENT_BOARD_ID,
task_id: Number(taskId), reason: "moved" },
            }));
          } catch (err) {
            revert();
            err.message = err.message || t("task_status_save_failed",
"Не удалось сохранить статус задачи");
            alert(err.message || t("task_status_save_failed", "Не
удалось сохранить статус задачи"));
          }
        },
```

Лістинг Б.9 – Уніфіковані API-запити з CSRF-захистом
Файл: app/static/js/common/api.js

```
  function getCsrfToken() {
    const meta = document.querySelector('meta[name="csrf-
token"]');
    return meta ? (meta.getAttribute('content') || '') : '';
  }

  const SAFE_METHODS = new Set(['GET', 'HEAD',
'OPTIONS', 'TRACE']);

  async function parseResponse(resp) {
    const contentType = resp.headers.get("content-type") || "";
```

```
  if (contentType.includes("application/json")) {
    try {
      return await resp.json();
    } catch {
      return {};
    }
  }

  try {
    const text = await resp.text();
    return text ? { message: text } : {};
  } catch {
return {};
  }
 }

 api.fetchJSON = async function fetchJSON(url, options = {})
{
  const method = (options.method || 'GET').toUpperCase();

  const csrfHeaders = {};
  if (!SAFE_METHODS.has(method)) {
    const token = getCsrfToken();
    if (token) {
      csrfHeaders['X-CSRFToken'] = token;
    }
  }

  const resp = await fetch(url, {
    credentials: "same-origin",
    ...options,
    headers: {
      Accept: "application/json",
      ...csrfHeaders,
      ...(options.headers || {}),
    },
  });

  const data = await parseResponse(resp);
```

Лістинг Б.10 – Docker Compose-конфігурація розгортання

Файл: docker-compose.yml

```
services:
 web:
  build: .
  restart: always
  ports:
   - "5000:5000"
  env_file:
   - .env
  environment:
   -
DATABASE_URL=postgresql://postgres:root@db:5432/minitr
ello
    - REDIS_URL=redis://redis:6379/0
    - JOB_LOCK_REDIS_URL=redis://redis:6379/1
    - RATELIMIT_STORAGE_URL=redis://redis:6379/2
    - SOCKETIO_MESSAGE_QUEUE=redis://redis:6379/0
    - SCHEDULER_ENABLED=1
    - JOB_LOCK_ENABLED=1
  depends_on:
    db:
      condition: service_healthy
    redis:
      condition: service_healthy

 telegram-bot:
  build: .
  restart: always
  command: python -m flask --app app:create_app telegram-
poll
  healthcheck:
    disable: true
  env_file:
   - .env
  environment:
   -
DATABASE_URL=postgresql://postgres:root@db:5432/minitr
ello
    - REDIS_URL=redis://redis:6379/0
    - JOB_LOCK_REDIS_URL=redis://redis:6379/1
    - RATELIMIT_STORAGE_URL=redis://redis:6379/2
    - SOCKETIO_MESSAGE_QUEUE=redis://redis:6379/0
    - SCHEDULER_ENABLED=0
    - JOB_LOCK_ENABLED=1
    -
TELEGRAM_DELETE_WEBHOOK_ON_POLLING=${TEL
EGRAM_DELETE_WEBHOOK_ON_POLLING:-true}
  depends_on:
    db:
      condition: service_healthy
    redis:
      condition: service_healthy

 db:
  image: postgres:15
  restart: always
  environment:
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: root
    POSTGRES_DB: minitrello
  ports:
   - "5432:5432"
  volumes:
   - postgres_data:/var/lib/postgresql/data
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U postgres -d
minitrello"]
    interval: 10s
    timeout: 5s
    retries: 10

 redis:
  image: redis:7-alpine
  restart: always
```