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

Допущено до захисту
Завідувач кафедри комп’ютерних наук
доктор технічних наук, професор
Андрій БОНДАРЧУК
« 01 » червня 2026 р.

КВАЛІФІКАЦІЙНА РОБОТА

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

Тема: ІНТЕРАКТИВНИЙ ОНЛАЙН-ПЛАНЕР ДЛЯ ТАЙМ-МЕНЕДЖМЕНТУ

Виконала
студентка групи ІНб-2-22.4.0д
Плотнікова Аліна Андріївна
_____
(підпис)

Науковий керівник
доктор технічних наук, професор
Бондарчук А.П.
_____
(підпис)

Київ – 2026

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

«Затверджую»
Завідувач кафедри комп'ютерних наук, доктор технічних наук, професор
Андрій БОНДАРЧУК

«31 » жовтня 2025 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА
«Інтерактивний онлайн-планер для тайм-менеджменту»

Виконавець – студентка групи ІНб-2-22-4.0д,
спеціальності 122 Комп'ютерні науки,
освітньої програми 122.00.01 Інформатика
Плотнікова Аліна Андріївна

1. Вихідні дані: *Законодавство та нормативно-правові акти України в* інформаційній сфері та за напрямом дослідження, зокрема Закон України «Про захист персональних даних». Наукові публікації з тематики тайм-менеджменту та розробки веб-застосунків. Міжнародні стандарти ISO/IEC 25010:2011 та OWASP Top 10:2021.

2. Основні завдання: Здійснити огляд публікацій за темою роботи та проаналізувати предметну область тайм-менеджменту. Розглянути методики Pomodoro, матрицю Ейзенхауера, GTD і трекінг звичок та провести порівняльний аналіз існуючих онлайн-планерів. Сформулювати функціональні, нефункціональні вимоги та вимоги до інформаційної безпеки. Спроєктувати архітектуру системи, побудувати UML-моделі, спроєктувати базу даних, інтерфейс користувача та підсистему безпеки. Реалізувати серверну частину (Python, Flask, SQLite), клієнтський інтерфейс (HTML, CSS, JavaScript) та підсистему безпеки. Провести функціональне тестування, тестування безпеки та юзабіліті-тестування.

3. Пояснювальна записка: Обсяг близько 90 стор. формату А4 комп'ютерного набору з дотриманням вимог стандарту і методичних рекомендацій кафедри.

4. Графічні матеріали: комп'ютерна презентація доповіді.

5. Додатки: лістинги програмного коду.

6. Строк подання роботи на кафедру: «01» червня 2026 р.

Науковий керівник доктор технічних наук, професор	Виконавець:
_____БОНДАРЧУК А.П.	_____ПЛОТНІКОВА А.А.
_____ дата	_____ дата

Календарний план

№	Етапи виконання роботи	Термін виконання	Примітка
1	Затвердження теми кваліфікаційної роботи	31.10.25	Виконано
2	Аналіз проблеми, вивчення аналогів, формування цілі і завдань	01.11.25 – 11.01.26	Виконано
3	Аналіз предметної області та вимог до системи	12.01.26 – 15.02.26	Виконано
4	Дослідження існуючих технічних рішень	16.02.26 – 15.03.26	Виконано
5	Проєктування архітектури, бази даних та інтерфейсу системи	16.03.26 – 31.03.26	Виконано
6	Розробка серверної та клієнтської частин онлайн-планера	01.04.26 – 15.04.26	Виконано
7	Тестування розробленого вебзастосунку	16.04.26 – 30.04.26	Виконано
8	Оптимізація, доопрацювання та оформлення пояснювальної записки	01.05.26 – 01.06.26	Виконано
9	Захист кваліфікаційної роботи	19.06.26	

«Затверджую»
Науковий керівник
доктор технічних наук, професор
Бондарчук А.П.

Індивідуальний план склав
Плотнікова А.А.

РЕФЕРАТ

Кваліфікаційна робота містить: 92 с., 41 рис., 12 табл., 63 джерела, 3 додатки.

У роботі розглянуто проблему ефективного керування часом в умовах інформаційного перевантаження та зростання кількості щоденних завдань.

Актуальність теми зумовлена тим, що наявні цифрові планери переважно не поєднують усі основні методики тайм-менеджменту в одному рішенні, часто мають платні обмеження, не підтримують локальне розгортання або не адаптовані для україномовного користувача.

Об’єкт дослідження – процес планування та керування часом із використанням цифрових інструментів. Предмет дослідження – веб-застосунок для інтерактивного тайм-менеджменту з підтримкою методик Pomodoro, матриці Ейзенхауера, GTD та трекера звичок.

Метою роботи є розробка інтерактивного онлайн-планера для тайм-менеджменту, який об’єднує кілька методик в єдиному україномовному інтерфейсі та забезпечує контроль користувача над персональними даними.

Практичне значення одержаних результатів полягає в можливості використання розробленої системи студентами, викладачами та фахівцями для організації робочого часу. Локальне розгортання забезпечує підвищений рівень контролю над персональними даними.

Ключові слова: тайм-менеджмент, онлайн-планер, Pomodoro, матриця Ейзенхауера, трекер звичок, Flask, SQLite, веб-застосунок.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

ACID – властивості надійної транзакційної обробки даних: атомарність, узгодженість, ізольованість, довговічність.

AJAX – технологія асинхронного обміну даними між клієнтом і сервером без повного перезавантаження сторінки.

API – програмний інтерфейс взаємодії між компонентами системи.

CRUD – базові операції роботи з даними: створення, читання, оновлення, видалення.

CSS – каскадні таблиці стилів для оформлення веб-інтерфейсу.

DFD – діаграма потоків даних.

ER-діаграма – модель сутностей і зв’язків бази даних.

Flask – мікрофреймворк Python для розробки веб-застосунку.

GDPR – загальний регламент захисту даних Європейського Союзу.

GTD – методика керування справами Getting Things Done.

HTML – мова розмітки веб-сторінок.

HTTP / HTTPS – протоколи передавання даних у веб-середовищі.

IDOR – вразливість, пов’язана з прямим доступом до об’єктів без належної перевірки прав доступу.

JSON – текстовий формат обміну структурованими даними.

MVC – архітектурний шаблон Model–View–Controller.

NFR – нефункціональні вимоги до системи.

OWASP – міжнародний проєкт з питань безпеки веб-застосунків.

REST API – архітектурний підхід до побудови веб-інтерфейсів взаємодії між клієнтом і сервером.

SQL – мова структурованих запитів до бази даних.

STRIDE – методологія моделювання загроз інформаційній безпеці.

ЗМІСТ

ВСТУП 4
РОЗДІЛ 1 6
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ФОРМУВАННЯ ВИМОГ 6
1.1 Поняття тайм-менеджменту та огляд методик 6
1.2 Порівняльний аналіз існуючих онлайн-планерів 10
1.3 Формування функціональних та нефункціональних вимог 13
1.4 Вимоги до інформаційної безпеки системи 15
1.5 Висновки до розділу 1 17
РОЗДІЛ 2 20
ПРОЄКТУВАННЯ ІНТЕРАКТИВНОГО ОНЛАЙН-ПЛАНЕРА 20
2.1 Обґрунтування вибору архітектури веб-застосунку 20
2.2 UML-моделювання системи 22
2.2.1 Діаграма варіантів використання та діаграма класів 23
2.2.2 Діаграма послідовності та діаграма діяльності 26
2.2.3 Діаграма компонентів і розгортання 29
2.3 Проєктування бази даних 32
2.4 Проєктування UX та інтерактивності інтерфейсу 35
2.4.1 Сценарії взаємодії та wireframes 35
2.4.2 Принципи інтерактивного зворотного зв'язку й адаптивності 38
2.5 Проєктування підсистеми безпеки 40
2.6 Обґрунтування вибору технологій реалізації 45
2.7 Висновки до розділу 2 47
РОЗДІЛ 3 49
РОЗРОБКА ТА ТЕСТУВАННЯ ПЛАНЕРА 49
3.1 Реалізація бази даних та серверної логіки 49
3.2 Реалізація підсистеми безпеки 56
3.3 Розробка інтерактивного інтерфейсу користувача 60
3.3.1 Динамічне оновлення та drag-and-drop 60

3.3.2 Візуалізація статистики та інструменти тайм-менеджменту 63
3.4 Тестування застосунку 70
3.5 Аналіз результатів та перспективи розвитку 72
3.6 Висновки до розділу 3 76
ВИСНОВКИ 79
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 81
ДОДАТКИ 88

ВСТУП

В умовах цифрової трансформації суспільства обсяг інформації, з яким стикається сучасна людина, зростає експоненціально. За даними дослідників, середній працівник витрачає значну частину робочого дня на реактивну обробку вхідних сигналів – електронних листів, повідомлень, нагадувань – замість планомірного руху до довгострокових цілей. У такому контексті тайм-менеджмент – розподіл часу для досягнення цілей через організацію дій і адаптацію до обставин – стає не просто корисною навичкою, а необхідною умовою продуктивності та психологічного благополуччя [1].

Мета-аналіз Б. Аеона, А. Фабера та А. Паначчо, що охопив 158 незалежних досліджень, продемонстрував помірний позитивний зв’язок тайм-менеджменту з продуктивністю праці, академічною успішністю та зниженням рівня стресу [1]. Подібні результати підтверджені у систематичних оглядах в галузі вищої освіти [2, 3]. Ці дані обґрунтовують актуальність розробки цифрових інструментів, які допомагають користувачеві системно застосовувати методики тайм-менеджменту.

Серед найбільш визнаних методик виділяються: Pomodoro (фокусовані інтервали з перервами), матриця Ейзенхауера (пріоритезація за терміновістю та важливістю), Getting Things Done (системна організація потоку справ) та трекінг звичок (формування продуктивної поведінки через самомоніторинг). Аналіз існуючих цифрових планерів (Todoist, TickTick, Notion, Any.do) показав, що жодне рішення одночасно не охоплює всі чотири методики, не є повністю безкоштовним, не підтримує локальне розгортання та не адаптоване для україномовного користувача.

Актуальність роботи зумовлена потребою в інтегрованому, безкоштовному та безпечному інструменті тайм-менеджменту, який об’єднує кілька методик в єдиному україномовному інтерфейсі з повним контролем користувача над персональними даними.

Мета роботи – розробка інтерактивного онлайн-планера для тайм-менеджменту на базі Python (Flask) та SQLite.

Для досягнення поставленої мети необхідно розв’язати такі завдання:

– проаналізувати предметну область тайм-менеджменту, розглянути методики Pomodoro, матрицю Ейзенхауера, GTD та трекінг звичок, провести порівняльний аналіз існуючих онлайн-планерів;

– сформулювати функціональні, нефункціональні вимоги та вимоги до інформаційної безпеки системи;

– спроєктувати архітектуру системи, побудувати UML-моделі, спроєктувати базу даних, інтерфейс користувача та підсистему безпеки;

– реалізувати серверну частину (Python, Flask, SQLite), клієнтський інтерфейс (HTML, CSS, JavaScript) та підсистему безпеки (bcrypt, Flask-Login);

– провести функціональне тестування, тестування безпеки та юзабіліті-тестування розробленої системи.

Об’єкт дослідження – процес планування та керування часом із використанням цифрових інструментів.

Предмет дослідження – веб-застосунок для інтерактивного тайм-менеджменту з підтримкою методик Pomodoro, матриці Ейзенхауера, GTD та трекера звичок.

Методи дослідження: аналіз літератури та існуючих рішень, порівняльний аналіз, UML-моделювання, моделювання загроз за методологією STRIDE, проєктування реляційних баз даних, автоматизоване інтеграційне тестування.

Практичне значення одержаних результатів полягає у створенні повнофункціонального веб-застосунку, який може бути використаний студентами, викладачами та фахівцями для організації робочого часу. Локальне розгортання системи забезпечує контроль над персональними даними відповідно до вимог Закону України «Про захист персональних даних» та Загального регламенту захисту даних (GDPR).

РОЗДІЛ 1

АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ФОРМУВАННЯ ВИМОГ

1.1 Поняття тайм-менеджменту та огляд методик

Тайм-менеджмент є міждисциплінарним поняттям, що об'єднує підходи психології, менеджменту та інформатики до організації власного часу. У сучасній науковій літературі під тайм-менеджментом розуміють розподіл часу для досягнення цілей через організацію дій і адаптацію до обставин, що постійно змінюються [1]. Таке визначення акцентує увагу не лише на плануванні, а й на метакогнітивному моніторингу – здатності коригувати стратегії виконання завдань у процесі їхнього виконання.

Актуальність тайм-менеджменту підтверджена результатами мета-аналізу Б. Аеона, А. Фабера та А. Паначчо, проведеного на вибірці, яка охопила 158 незалежних емпіричних досліджень. За даними цього мета-аналізу, тайм-менеджмент має помірний позитивний зв'язок із продуктивністю праці, академічною успішністю та психологічним благополуччям, а також помірний негативний зв'язок із рівнем стресу [1]. Подібні висновки підтверджені у систематичному огляді, опублікованому у журналі Frontiers in Education, де наголошено, що постановка цілей, пріоритезація та планування в короткостроковій і довгостроковій перспективі є ключовими предикторами академічної успішності студентів [2].

Значущість проблеми особливо яскраво проявляється в освітньому середовищі. У систематичному огляді та мета-аналізі С. Ахмаді та співавторів установлено статистично значущий негативний зв'язок між стресом і академічною успішністю ($r = –0,32$), а також позитивний вплив навичок тайм-менеджменту на успішність студентів-медиків [3]. Ці результати підкреслюють необхідність впровадження цифрових інструментів, які допомагають формувати й підтримувати навички керування часом на щоденному рівні.

У межах цього дослідження тайм-менеджмент розглядається як сукупність методик, спрямованих на: 1) фіксацію завдань у зовнішній системі з метою зниження когнітивного навантаження; 2) пріоритезацію завдань за критеріями терміновості та важливості; 3) планування часових інтервалів для фокусованого виконання роботи; 4) систематичне формування продуктивних звичок. Далі розглянуто три методики, які отримали найбільше визнання як у практиці, так і в емпіричних дослідженнях.

Методика Pomodoro. Розроблена Франческо Чирілло наприкінці 1980-х років, методика Pomodoro передбачає поділ робочого часу на фокусовані інтервали тривалістю 25 хвилин, розділені короткими перервами по 5 хвилин [4]. Після чотирьох таких циклів рекомендується довша перерва тривалістю 15–30 хвилин. Теоретичною основою методики є концепція time-boxing, за якою для заздалегідь визначеної послідовності дій фіксується нерухомий часовий інтервал, що дозволяє уникати перенесення строків та захищати фокус від внутрішніх і зовнішніх переривань.

Емпіричні дослідження ефективності методики Pomodoro демонструють неоднозначні, але переважно позитивні результати. У масштабному оглядовому дослідженні, опублікованому у журналі BMC Medical Education, проаналізовано 32 публікації (загальна вибірка n = 5 270 респондентів) та виявлено, що Pomodoro підтримує стійку увагу, зменшує ментальну втому й ефективна при подоланні прокрастинації [4]. Водночас у рандомізованому дослідженні Я. ван дер Гуде та інших (2025 р.) на вибірці з 94 студентів показано, що Pomodoro-перерви можуть викликати швидший спад мотивації порівняно з саморегульованими перервами, хоча сумарні показники продуктивності статистично не відрізнялися [5]. Ці результати свідчать про те, що ефективність методики залежить від індивідуальних особливостей користувача, що робить доцільною її реалізацію в інтерактивному планері з можливістю гнучкого налаштування тривалості інтервалів.

Матриця Ейзенхауера. Методика, названа на честь 34-го президента США Д. Ейзенхауера та популяризована С. Кові у книзі «7 навичок високоефективних

людей», базується на двовимірному розподілі завдань за критеріями терміновості та важливості [6, 7]. Завдання класифікуються на чотири квадранти: терміново-важливі (виконати негайно), нетерміново-важливі (запланувати), терміново-неважливі (делегувати) та нетерміново-неважливі (відкласти або ігнорувати).

Психологічною основою дієвості матриці є подолання так званого ефекту уявної терміновості (mere-urgency effect), описаного у серії експериментів, опублікованих у Journal of Consumer Research: людина схильна обирати термінові, але менш важливі завдання навіть тоді, коли важливі, але нетермінові обіцяють більшу винагороду [7]. Матриця Ейзенхауера слугує інструментом візуалізації, що змушує свідомо зупинитися перед вибором завдання й оцінити його довгострокову цінність, а не лише часовий тиск.

Методика Getting Things Done (GTD). Розроблена Д. Алленом у 2001 році, ця методика базується на ідеї, що людський мозок погано пристосований до зберігання великої кількості незавершених справ, і тому всі вхідні сигнали – завдання, ідеї, зобов'язання – мають бути винесені у зовнішню систему обробки [8]. GTD передбачає п'ять послідовних етапів: збір (capture), обробка (clarify), організація (organize), огляд (reflect) та виконання (engage).

У науковому огляді Ф. Хейлігена та К. Йошими, опублікованому в журналі Long Range Planning, продемонстровано, що принципи GTD узгоджуються із сучасними теоріями ситуаційного, утіленого та розподіленого пізнання: розвантаження робочої пам'яті через винесення завдань у зовнішню систему дозволяє мозку зосередитися на виконанні дій, а не на їх запам'ятовуванні [8]. Ключовим елементом GTD є щотижневий огляд (weekly review), під час якого користувач переглядає всі активні проєкти та визначає наступні дії, що забезпечує довіру до системи та зменшує тривогу з приводу забутих справ.

Таблиця 1.1

Порівняння методик тайм-менеджменту

Критерій	**Pomodoro**	**Матриця Ейзенхауера**	**GTD**
Основний фокус	Керування увагою у часі	Пріоритезація завдань	Організація потоку завдань
Ключовий механізм	Time-boxing, чергування фокусу та відпочинку	Двовимірна класифікація завдань	Винесення завдань у зовнішню систему та щотижневий огляд
Сфера застосування	Виконання окремих завдань, навчання	Планування дня, тижня, стратегічних рішень	Комплексне керування справами у довгостроковій перспективі
Переваги	Простота, підвищення концентрації, протидія прокрастинації	Наочність, допомагає позбутись уявної терміновості	Зниження когнітивного навантаження, системність
Обмеження	Фіксований інтервал не підходить для творчих задач; ризик зниження мотивації	Суб'єктивність критеріїв важливості	Висока складність освоєння, потребує дисципліни

Як видно з таблиці 1.1, розглянуті методики не конкурують, а взаємодоповнюють одна одну, охоплюючи різні рівні керування часом: від мікрорівня окремої 25-хвилинної сесії (Pomodoro) до мезорівня пріоритезації завдань дня (матриця Ейзенхауера) і макрорівня системної організації всього потоку справ (GTD). Тому сучасний інтерактивний онлайн-планер доцільно будувати як платформу, яка підтримує всі три методики одночасно, а не обирає якусь одну.

Окремо варто виділити методику формування та відстеження звичок (habit tracking), яка останніми роками набула значного поширення в цифрових додатках. У класичному дослідженні Ф. Лаллі та її колег, проведеному в University College London, встановлено, що формування автоматичної поведінки в реальних умовах займає від 18 до 254 днів (у середньому близько 66 днів) і описується квадратичною кривою з вираженим зростанням на початковому етапі [9]. Систематичний огляд, опублікований у журналі Frontiers in Psychology, підтверджує, що цифрові втручання на основі самомоніторингу, постановки

цілей та підказок є одними з найефективніших технік зміни поведінки [10]. Це обґрунтовує доцільність включення трекера звичок як окремого модуля майбутньої системи.

1.2 Порівняльний аналіз існуючих онлайн-планерів

На ринку програмного забезпечення для тайм-менеджменту представлений широкий спектр рішень: від мінімалістичних списків справ до комплексних платформ, що об'єднують керування проєктами, нотатки та командну співпрацю. Для обґрунтованого формування вимог до власної системи необхідно проаналізувати найбільш популярні існуючі продукти, виявити їхні сильні сторони та недоліки, а також визначити ніші, що залишаються не повністю покритими.

Критеріями порівняльного аналізу в цьому дослідженні обрано: наявність базових функцій керування завданнями; підтримку розглянутих у підрозділі 1.1 методик тайм-менеджменту; наявність трекера звичок; візуалізацію статистики користувача; безкоштовний доступ до повного функціоналу; можливість локального розгортання (self-hosted). Останній критерій є важливим з огляду на сучасні вимоги до інформаційної безпеки та контролю над персональними даними користувачів.

Для порівняння відібрано чотири рішення, що представляють різні категорії продуктів: Todoist – класичний персональний менеджер завдань із штучним інтелектом; TickTick – комплексна платформа з умонтованими інструментами тайм-менеджменту; Notion – універсальна робоча область з гнучкою структурою; Any.do – мобільно-орієнтований планер. Результати порівняння узагальнено в таблиці 1.2.

Таблиця 1.2

Порівняльний аналіз існуючих онлайн-планерів

Функція / Критерій	**Todoist**	**TickTick**	**Notion**	**Any.do**
Керування завданнями	+	+	+	+
Pomodoro-таймер	–	+	–	–
Матриця Ейзенхауера	–	+	Ручне налаштування	–
Підтримка GTD	+ (шаблони)	Частково	+ (шаблони)	Частково
Трекер звичок	–	+	Ручне налаштування	–
Візуалізація статистики	+ (обмежена)	+	Ручне налаштування	–
Повний функціонал у безкоштовній версії	–	–	+ (для персонального використання)	–
Локальне розгортання	–	–	–	–
Контроль над персональними даними	Низький	Низький	Низький	Низький

Як видно з таблиці 1.2, жодне з проаналізованих рішень не охоплює повний набір вимог. Todoist, за даними порівняльних оглядів 2025 року, вирізняється простотою інтерфейсу, сильною підтримкою природної мови при введенні завдань та наявністю умонтованого ШІ-помічника, однак не містить Pomodoro-таймера, трекера звичок і матриці Ейзенхауера, що змушує користувача інтегрувати сторонні сервіси [11]. TickTick позиціонується як комплексна платформа та вмикає Pomodoro-таймер зі статистикою, трекер звичок, повноцінну матрицю Ейзенхауера й календарний вигляд, проте повний функціонал доступний лише у платній версії приблизно за 36 доларів США на рік [12].

Notion являє собою універсальну робочу область, де кожен компонент тайм-менеджменту може бути сконструйований користувачем із базових блоків, однак це вимагає значних витрат часу на початкове налаштування та не надає готових спеціалізованих інструментів на кшталт Pomodoro. Any.do має сильну

мобільну реалізацію, проте відстає за функціональною глибиною. Усі чотири продукти є пропрієтарними хмарними сервісами, що не дає користувачеві контролю над персональними даними та унеможливлює локальне розгортання – критично важливий аспект для організацій із підвищеними вимогами до інформаційної безпеки.

Аналіз виявив декілька функціональних прогалин, які обґрунтовують розробку власної системи:

– відсутність безкоштовних рішень, що одночасно підтримують усі розглянуті методики тайм-менеджменту (Pomodoro, матрицю Ейзенхауера, GTD та трекер звичок);

– неможливість самостійного розгортання (self-hosting), що обмежує контроль користувача над власними даними;

– обмежена глибина аналітики продуктивності: більшість систем показують лише прості метрики виконаних завдань, без зрізів за категоріями, пріоритетами та Pomodoro-сесіями в єдиному поданні;

– орієнтація переважно на англомовну аудиторію та недостатня локалізація під українського користувача, включно з урахуванням термінології та типографічних норм української мови.

Результати порівняльного аналізу обґрунтовують розробку інтерактивного онлайн-планера, який: 1) об'єднує чотири методики тайм-менеджменту в єдиному інтерфейсі; 2) розгортається локально з використанням легковагої бази даних; 3) має повнофункціональну безкоштовну реалізацію; 4) забезпечує повний контроль користувача над персональними даними; 5) адаптований для україномовної аудиторії. Сформульовані висновки покладено в основу формування функціональних і нефункціональних вимог, розглянутих у подальших підрозділах.

1.3 Формування функціональних та нефункціональних вимог

На основі проведеного у підрозділах 1.1 та 1.2 аналізу предметної області та існуючих рішень формуються вимоги до системи, що розробляється. Вимоги поділяються на два класи: функціональні, які описують конкретні дії, котрі система має виконувати, та нефункціональні, які характеризують якість виконання цих дій – продуктивність, надійність, зручність використання, сумісність і масштабованість [13]. Такий поділ відповідає міжнародному стандарту ISO/IEC 25010:2011, який визначає модель якості програмного продукту на основі восьми характеристик.

Функціональні вимоги. Функціональні вимоги до інтерактивного онлайн-планера формуються через призму основних ролей користувача та дій, які він має змогу виконувати в системі. Головним актором виступає зареєстрований користувач, який взаємодіє з усіма модулями застосунку; також передбачена роль незареєстрованого відвідувача, якому доступні виключно сторінки входу та реєстрації.

Функціональні вимоги згруповано за дев'ятьма підсистемами, що відображає модульну архітектуру застосунку. Повний перелік подано в Додаток Г.

Як видно з таблиці в Додаток Г, функціональні вимоги повністю охоплюють методики тайм-менеджменту, розглянуті в підрозділі 1.1, та усувають прогалини, виявлені в підрозділі 1.2 при порівнянні наявних рішень. Зокрема, модуль FR-6 (Pomodoro-таймер) реалізує принцип time-boxing, модуль FR-9 (матриця Ейзенхауера) – двовимірну пріоритезацію завдань, модуль FR-7 (трекер звичок) – механізм самомоніторингу, ефективність якого підтверджена систематичними оглядами [10]. Такий комплексний підхід відрізняє розроблюваний продукт від більшості існуючих безкоштовних рішень.

Нефункціональні вимоги. Нефункціональні вимоги визначають якісні характеристики системи, яким вона має відповідати незалежно від конкретних функцій. У стандарті ISO/IEC 25010:2011 виділяють вісім характеристик якості

програмного забезпечення: функційна придатність, ефективність, сумісність, зручність використання, надійність, безпека, супроводжуваність і переносимість [13]. Для розроблюваного онлайн-планера нефункціональні вимоги сформульовано у таблиці 1.3 з конкретними кількісними критеріями.

Таблиця 1.3

Нефункціональні вимоги до інтерактивного онлайн-планера

Код	**Характеристика**	**Опис та критерії**
NFR-1	Продуктивність	Час відповіді сервера на типовий запит не має перевищувати 200 мс при навантаженні до 100 одночасних користувачів на локальному сервері; генерація сторінки дашборда – не більше 500 мс
NFR-2	Масштабованість	Система має коректно працювати при обсязі даних до 10 000 завдань на одного користувача без погіршення продуктивності; архітектура повинна дозволяти міграцію з SQLite на PostgreSQL без переписування бізнес-логіки
NFR-3	Зручність використання	Інтерфейс має бути повністю україномовним; основні дії (створення завдання, запуск таймера) доступні не більш ніж за два кліки від головної сторінки; підтримка темної та світлої тем оформлення
NFR-4	Надійність	База даних має підтримувати ACID-транзакції; використання режиму Write-Ahead Logging (WAL) SQLite для запобігання втраті даних при аварійному завершенні роботи сервера
NFR-5	Безпека	Див. підрозділ 1.4: захист відповідно до OWASP Top 10:2021; хешування паролів за алгоритмом bcrypt із фактором складності не нижче 12; ізоляція даних користувачів
NFR-6	Супроводжуваність	Код повинен бути модульним, із розділенням відповідальності між серверним (app.py, database.py) та клієнтським рівнями; покриття автоматизованими тестами ключових маршрутів
NFR-7	Переносимість	Система має запускатися на операційних системах Windows, macOS та Linux без змін у коді; єдина вимога – наявність Python 3.10 або новішої версії
NFR-8	Сумісність	Клієнтська частина має коректно відображатися у браузерах Chrome, Firefox, Safari та Edge останніх двох версій; адаптивна верстка для мобільних пристроїв із шириною екрана від 320 px

Сформульовані в таблицях 1.3 та 1.4 вимоги є основою для подальших етапів проєктування – розробки UML-моделей, проєктування бази даних та вибору технологій реалізації. Окрему увагу приділено вимогам безпеки (NFR-5), які через свою важливість винесено в окремий підрозділ 1.4.

1.4 Вимоги до інформаційної безпеки системи

Онлайн-планер оперує персональними даними користувача – електронною поштою, особистими завданнями, цілями, фіксацією робочої активності, що робить інформаційну безпеку критично важливим аспектом системи. Обробка персональних даних в Україні регулюється Законом України «Про захист персональних даних» від 01.06.2010 № 2297-VI, а для користувачів із країн Європейського Союзу додатково застосовуються норми Загального регламенту захисту даних (GDPR). Технічні вимоги до безпеки програмних додатків систематизовані у документі OWASP Top 10, який періодично оновлюється міжнародною некомерційною організацією OWASP (Open Web Application Security Project) і є де-факто галузевим стандартом обізнаності про критичні ризики [14].

Окрім законодавчих норм, проєктування безпечних систем спирається на концепцію «захист від початку» (security-by-design), за якою механізми захисту закладаються на етапі архітектурного проєктування, а не додаються після завершення розробки. Такий підхід дозволяє уникнути системних вразливостей, усунення яких на пізніших стадіях вимагає суттєвих витрат часу і ресурсів. Принцип найменших привілеїв, який є невід'ємною складовою security-by-design, передбачає, що кожен компонент системи отримує лише ті права доступу, які мінімально необхідні для виконання його функцій. Застосування цього принципу в розроблюваному онлайн-планері виявляється у тому, що кожен SQL-запит фільтрує дані за ідентифікатором конкретного користувача, не надаючи жодного доступу до записів інших облікових записів навіть у межах одного програмного модуля.

Для формування вимог до безпеки системи використано актуальну на момент розробки редакцію OWASP Top 10:2021, яка виділяє десять категорій найбільш критичних загроз [15]. Для кожної категорії в контексті розроблюваного онлайн-планера визначено відповідний контрзахід, що має бути реалізований на рівні архітектури або коду (таблиця 1.4).

Таблиця 1.4

Відповідність вимог безпеки онлайн-планера категоріям OWASP Top 10:2021

Код	Категорія OWASP	Контрзахід у системі
A01	**Порушений контроль доступу**	Обов'язкова автентифікація для всіх маршрутів, окрім входу та реєстрації (декоратор login_required); кожен запит до бази фільтрується за ідентифікатором власника ресурсу для запобігання горизонтальному підвищенню привілеїв (IDOR)
A02	**Криптографічні недоліки**	Хешування паролів алгоритмом bcrypt з унікальною сіллю для кожного пароля та фактором складності (cost factor) не нижче 12; передбачена підтримка HTTPS при розгортанні у промисловому середовищі
A03	**Ін'єкції (SQL, XSS)**	Використання параметризованих SQL-запитів через модуль sqlite3; автоматичне екранування виведення в шаблонах Jinja2, що запобігає Cross-Site Scripting; валідація вхідних даних на рівні маршрутів
A04	**Небезпечний дизайн**	Моделювання загроз ще на етапі проєктування; застосування принципу найменших привілеїв; обмеження довжини полів і формату введення (email, дата, час)
A05	**Помилки конфігурації**	Режим DEBUG вимкнено в промисловому розгортанні; secret_key генерується випадково за допомогою os.urandom(32); куки сесії мають атрибут HttpOnly
A06	**Застарілі компоненти**	Усі залежності закріплені у файлі requirements.txt з мінімальними версіями; періодичний аудит залежностей на наявність CVE через pip-audit або аналогічні інструменти
A07	**Помилки автентифікації**	Використання перевіреної бібліотеки Flask-Login для керування сесіями; мінімальна довжина пароля 6 символів; захист від атак перебору через обмеження швидкості (rate limiting) на маршрутах входу
A08	**Порушення цілісності ПЗ і даних**	Завантаження залежностей лише з довірених репозиторіїв (PyPI); перевірка контрольних сум; використання транзакцій бази даних для забезпечення цілісності
A09	**Помилки журналювання та моніторингу**	Журналювання подій безпеки: невдалі спроби входу, зміна пароля, помилки 500; зберігання журналів без персональних даних у відкритому вигляді
A10	**Підробка запитів з боку сервера (SSRF)**	Система не виконує зовнішніх HTTP-запитів на основі введення користувача; ризик мінімальний завдяки архітектурі, у якій сервер звертається лише до локальної бази даних

Додатково до категорій OWASP Top 10 сформульовано низку специфічних вимог, пов'язаних із особливостями предметної області та українським законодавством про захист персональних даних:

- користувач має повний контроль над своїми даними, зокрема змогу видалити власний акаунт разом з усіма пов'язаними даними (правило cascade delete у схемі бази даних);
- система розгортається локально (self-hosted), що виключає передачу персональних даних третім сторонам і спрощує відповідність вимогам GDPR та Закону України «Про захист персональних даних»;
- у базі даних не зберігаються паролі у відкритому вигляді; під час автентифікації порівнюються лише хеші з використанням функції bcrypt.check_password_hash(), яка виконує порівняння за сталий час (constant-time comparison), запобігаючи атакам за часом;
- ізоляція даних користувачів забезпечується не лише на рівні додатку, а й на рівні запитів до бази даних: кожен SQL-запит містить фільтр user_id = ?, що унеможливлює доступ одного користувача до даних іншого навіть за наявності помилки в коді контролерів.

Обґрунтування вибору алгоритму bcrypt замість альтернатив (MD5, SHA-1, SHA-256) базується на принципі свідомої обчислювальної складності: bcrypt розроблено як повільну функцію, на відміну від криптографічних хешів загального призначення, оптимізованих для швидкості [16]. Такий дизайн істотно ускладнює атаки перебору та обчислення за допомогою таблиць райдужних значень (rainbow tables) навіть у випадку витоку бази даних. Вбудована в bcrypt випадкова сіль гарантує, що два однакові паролі матимуть різні хеші, а параметр складності дозволяє збільшувати обчислювальні витрати атакуючого разом зі зростанням обчислювальної потужності обладнання. Зазначені вимоги безпеки є обов'язковими для реалізації на етапі розробки та тестування системи.

1.5 Висновки до розділу 1

У першому розділі проведено комплексний аналіз предметної області та сформовано вимоги до розроблюваного інтерактивного онлайн-планера для тайм-менеджменту. Основні результати розділу полягають у наступному.

По-перше, встановлено, що тайм-менеджмент є ефективним інструментом підвищення продуктивності, академічної успішності та психологічного благополуччя. Цей висновок підтверджено результатами мета-аналізу на вибірці з 158 досліджень [1] та систематичними оглядами в галузі освіти [2, 3]. Наведені емпіричні дані обґрунтовують актуальність розробки цифрового інструменту, який допомагає користувачеві системно застосовувати методики тайм-менеджменту у повсякденному житті.

По-друге, проаналізовано три найбільш визнані методики тайм-менеджменту – Pomodoro, матрицю Ейзенхауера та Getting Things Done. Показано, що ці методики не конкурують, а взаємодоповнюють одна одну, охоплюючи різні рівні керування часом: мікрорівень окремих фокусованих сесій (Pomodoro), мезорівень пріоритезації завдань дня (матриця Ейзенхауера) та макрорівень системної організації справ (GTD). Додатково обґрунтовано включення трекера звичок як важливого модуля, ефективність якого підтверджується дослідженнями цифрових втручань із метою формування поведінки [9, 10].

По-третє, проведено порівняльний аналіз чотирьох найпопулярніших онлайн-планерів (Todoist, TickTick, Notion, Any.do) за десятьма критеріями. Виявлено, що жодне з аналізованих рішень одночасно не охоплює всі розглянуті методики, не надає повного функціоналу безкоштовно, не підтримує локальне розгортання й не адаптоване для україномовного користувача. Ці прогалини формують ринкову нішу, яку покликана заповнити розроблювана система.

По-четверте, на основі проведеного аналізу сформульовано одинадцять функціональних (FR-1 – FR-11) та вісім нефункціональних (NFR-1 – NFR-8) вимог. Функціональні вимоги покривають усі дев'ять модулів застосунку: автентифікацію, особистий кабінет, керування завданнями, фільтрацію та пошук, календар, Pomodoro-таймер, трекер звичок, цілі, матрицю Ейзенхауера, аналітику та нотатки. Нефункціональні вимоги визначають якісні характеристики системи за моделлю ISO/IEC 25010:2011 з конкретними

кількісними критеріями продуктивності, масштабованості, зручності використання, надійності, супроводжуваності, переносимості та сумісності.

По-п'яте, окремо розглянуто вимоги до інформаційної безпеки системи на основі стандарту OWASP Top 10:2021 [14, 15]. Для кожної з десяти категорій критичних загроз сформульовано конкретний контрзахід у контексті розроблюваного онлайн-планера. Обґрунтовано вибір алгоритму bcrypt для хешування паролів та принцип ізоляції даних користувачів на рівні SQL-запитів [16]. Додатково враховано вимоги українського законодавства про захист персональних даних та міжнародного регламенту GDPR, що забезпечується архітектурою локального розгортання системи.

Отримані в розділі результати формують повну основу для наступного етапу роботи – проєктування системи з використанням UML-моделей, проєктування бази даних та обґрунтування вибору технологій реалізації, що буде розглянуто у другому розділі.

РОЗДІЛ 2

ПРОЄКТУВАННЯ ІНТЕРАКТИВНОГО ОНЛАЙН-ПЛАНЕРА

2.1 Обґрунтування вибору архітектури веб-застосунку

Архітектура програмної системи визначає фундаментальну організацію її компонентів, їхні зв'язки між собою та з середовищем, а також принципи, що керують її проєктуванням та еволюцією [17]. Вибір архітектури впливає на всі нефункціональні вимоги системи, сформульовані у підрозділі 1.3: продуктивність, масштабованість, супроводжуваність та безпеку. Для розроблюваного інтерактивного онлайн-планера обрано трирівневу клієнт-серверну архітектуру, яка розмежовує рівень представлення, рівень бізнес-логіки та рівень даних.

Рівень представлення (клієнтський рівень) відповідає за відображення даних та взаємодію з користувачем. Реалізується у веб-браузері за допомогою HTML-шаблонів, стилів CSS та скриптів JavaScript. Для генерації HTML-сторінок на стороні сервера використовується шаблонізатор Jinja2, інтегрований у фреймворк Flask. Динамічне оновлення окремих елементів інтерфейсу (створення завдань, перемикання статусу, операції трекера звичок) здійснюється без перезавантаження сторінки через асинхронні HTTP-запити (AJAX) за допомогою Fetch API. Такий підхід забезпечує інтерактивність інтерфейсу, що є ключовою вимогою до системи тайм-менеджменту (NFR-3).

Рівень бізнес-логіки (серверний рівень) обробляє HTTP-запити, виконує валідацію даних, керує автентифікацією та авторизацією, реалізує бізнес-правила (обчислення серії днів, розподіл за матрицею Ейзенхауера, формування аналітичних звітів) і повертає відповіді у вигляді HTML-сторінок або JSON-об'єктів. Серверний рівень реалізовано мовою Python з використанням мікрофреймворку Flask, який дотримується патерну MVC (Model – View – Controller). У контексті Flask моделі відповідають модулю database.py з CRUD-

операціями, вигляди (Views) – шаблонам Jinja2, а контролери – функціям-маршрутам у модулі app.py.

Рівень даних забезпечує довготривале зберігання інформації. Обрано реляційну СУБД SQLite, яка зберігає всю базу даних в єдиному файлі planner.db. Обґрунтування вибору SQLite ґрунтується на кількох чинниках: відсутність потреби в окремому серверному процесі (zero-configuration deployment), що спрощує локальне розгортання (NFR-7); підтримка ACID-транзакцій та режиму Write-Ahead Logging для забезпечення надійності (NFR-4); достатня продуктивність для цільового навантаження до 100 одночасних користувачів [18]. При цьому модуль database.py спроєктовано так, щоб SQL-запити використовували стандартний синтаксис, сумісний з PostgreSQL, що дозволяє мігрувати на цю СУБД у разі масштабування без переписування бізнес-логіки (NFR-2).

Взаємодія між клієнтським та серверним рівнями відбувається за двома каналами. Перший – класичний серверний рендеринг: клієнт надсилає HTTP-запит (GET або POST), сервер генерує повну HTML-сторінку за допомогою Jinja2 і повертає її. Другий – REST API: клієнт надсилає асинхронний запит (POST, PUT, DELETE) із JSON-тілом, а сервер повертає компактну JSON-відповідь. Комбінація цих двох підходів дозволяє забезпечити швидке початкове завантаження сторінок (серверний рендеринг) та миттєву реакцію на дії користувача без перезавантаження (REST API).

Загальну архітектуру системи подано на рисунку 2.1, де стрілками позначено напрямки потоків даних між рівнями.

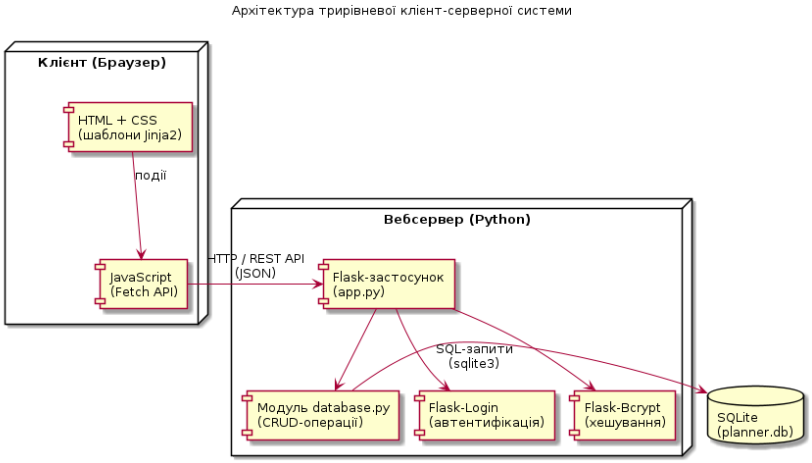


Рисунок 2.1. Архітектура трирівневої клієнт-серверної системи

Як видно з рисунку 2.1, Flask-застосунок є центральним компонентом, який координує роботу решти модулів: Flask-Login забезпечує автентифікацію та керування сесіями, Flask-Bcrypt – хешування паролів, а модуль database.py – абстракцію над SQLite. Така архітектура відповідає принципу розмежування відповідальності (Separation of Concerns) і забезпечує супроводжуваність системи (NFR-6): зміни в одному рівні не потребують модифікації інших. Наприклад, заміна SQLite на PostgreSQL вимагає правок лише у database.py, а перехід з серверного рендерингу на односторінковий застосунок (SPA) зачепить лише клієнтський рівень і маршрути REST API.

2.2 UML-моделювання системи

UML (Unified Modeling Language) є стандартизованою мовою візуального моделювання програмних систем, визначеною специфікацією OMG (Object Management Group) [19]. Використання UML на етапі проєктування забезпечує формалізоване подання структури та поведінки системи, що полегшує комунікацію між учасниками розробки та дозволяє виявити архітектурні помилки до початку кодування. У межах цього підрозділу побудовано шість UML-діаграм, що охоплюють структурні, поведінкові та архітектурні аспекти системи. Для побудови діаграм використано інструмент PlantUML, що дозволяє описувати UML-моделі у текстовому форматі та автоматично генерувати графічні подання.

2.2.1 Діаграма варіантів використання та діаграма класів

Діаграма варіантів використання (Use Case Diagram) описує функціональні можливості системи з точки зору зовнішніх акторів та відображає межі системи [19]. Побудована діаграма (рис. 2.2) визначає двох акторів: Відвідувач (незареєстрований користувач) та Користувач (зареєстрований). Відвідувач має доступ лише до варіантів використання UC1 (Зареєструватися) та UC2 (Увійти в систему). Актор Користувач успадковує можливості Відвідувача та додатково має доступ до одинадцяти варіантів використання, що відповідають функціональним вимогам FR-3 – FR-11, сформульованим у підрозділі 1.3.

На діаграмі позначено три зв'язки типу <<include>>: варіант використання «Фільтрувати і шукати завдання» включає «Керувати завданнями», оскільки фільтрація неможлива без наявності завдань; аналогічно, «Розподіляти завдання у матриці Ейзенхауера» та «Аналізувати продуктивність» залежать від наявності створених завдань. Ці включення відображають архітектурний принцип: модулі аналітики та матриці Ейзенхауера не зберігають власних даних, а оперують даними модуля завдань.

Варто зазначити, що зв'язок типу <<include>> є обов'язковим за семантикою UML: включений варіант використання виконується щоразу під час виклику базового, на відміну від необов'язкового зв'язку <<extend>>, який активується лише за певної умови. Вибір саме <<include>>, а не <<extend>>, для зазначених залежностей відображає архітектурне рішення про те, що фільтрація, матриця Ейзенхауера та аналітика є невід'ємними розширеннями модуля завдань, а не самостійними сценаріями. Такий підхід спрощує супроводжуваність системи: за потреби зміни в модулі завдань автоматично поширюються на всі залежні варіанти використання без необхідності окремого оновлення кожного з них. Побудована діаграма слугує вхідними даними для наступного етапу UML-моделювання – діаграми класів, яка конкретизує статичну структуру системи та безпосередньо відображає схему бази даних.

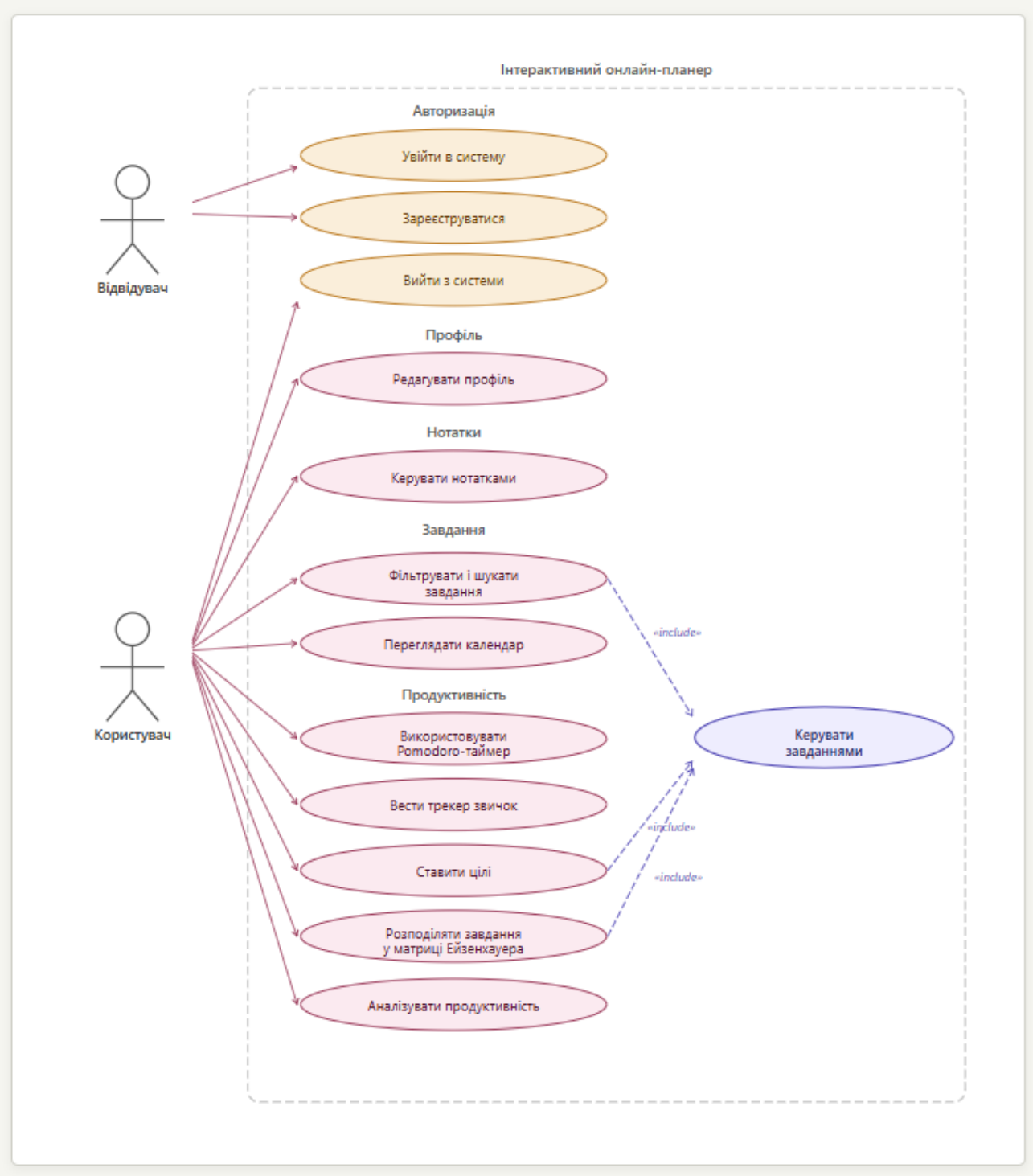


Рисунок 2.2. Діаграма варіантів використання системи

Діаграма класів (Class Diagram) є центральною структурною діаграмою UML, що відображає статичну структуру системи: класи, їхні атрибути, методи та зв'язки між класами [19]. На рисунку 2.3 подано діаграму класів предметної області інтерактивного онлайн-планера, яка безпосередньо відповідає структурі бази даних, реалізованій у модулі database.py.

Центральним класом діаграми є User, який містить атрибути автентифікації (username, email, password_hash), персоналізації (full_name, theme, avatar_color) та налаштування Pomodoro-таймера (pomodoro_work, pomodoro_break). Клас User пов'язаний композицією (зв'язок «1 до 0..*») із п'ятьма класами: Task, Goal, Habit, PomodoroSession та Note. Композиція означає, що об'єкти цих класів не можуть існувати без власника – при видаленні користувача каскадно видаляються всі його дані (CASCADE DELETE), що забезпечує виконання вимоги безпеки, описаної у підрозділі 1.4.

Клас Task є найбагатшим за атрибутами: він містить назву, опис, дату, час початку та закінчення, категорію (перелічення Category із шести значень: WORK, STUDY, PERSONAL, HEALTH, MEETING, OTHER), пріоритет (HIGH, MEDIUM, LOW) та статус виконання. Метод get_eisenhower_quadrant() обчислює номер квадранта матриці Ейзенхауера на основі значень пріоритету та категорії, що забезпечує автоматичний розподіл завдань (FR-9). Клас Habit пов'язаний із HabitLog зв'язком «1 до 0..*»: кожне відмічання звички фіксується як окремий запис із датою, що дозволяє обчислювати серію (streak) послідовних днів.

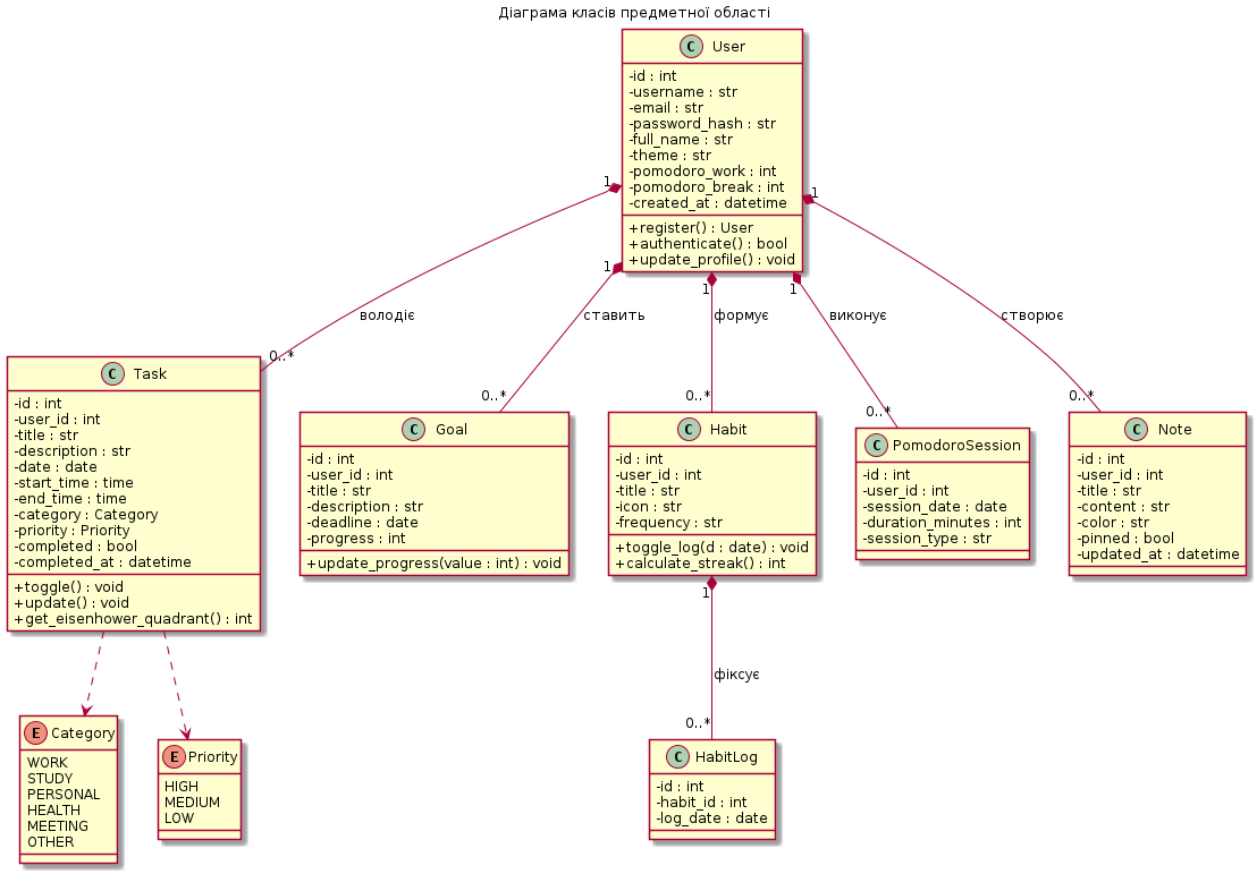


Рисунок 2.3. Діаграма класів предметної області

2.2.2 Діаграма послідовності та діаграма діяльності

Діаграма послідовності (Sequence Diagram) описує порядок обміну повідомленнями між об'єктами в рамках конкретного сценарію взаємодії [19]. Побудовано дві діаграми послідовності для ключових сценаріїв системи: реєстрація нового користувача (рис. 2.4) та створення нового завдання (рис. 2.5).

На рисунку 2.4 відображено повний цикл реєстрації, що включає шість учасників: Відвідувач, Браузер, Flask-контролер (app.py), Валідатор, Flask-Bcrypt та модуль database.py. Процес починається з надсилання POST-запиту на маршрут /register і розгалужується через фрагмент alt: при валідних даних виконується перевірка унікальності логіна (SQL-запит SELECT), хешування пароля через Flask-Bcrypt, створення запису у базі даних (INSERT) та автоматичний вхід через Flask-Login із перенаправленням на дашборд; при невалідних даних – повернення форми з повідомленнями про помилки.

Наведений сценарій повністю відповідає реалізації у функції register() модуля app.py.

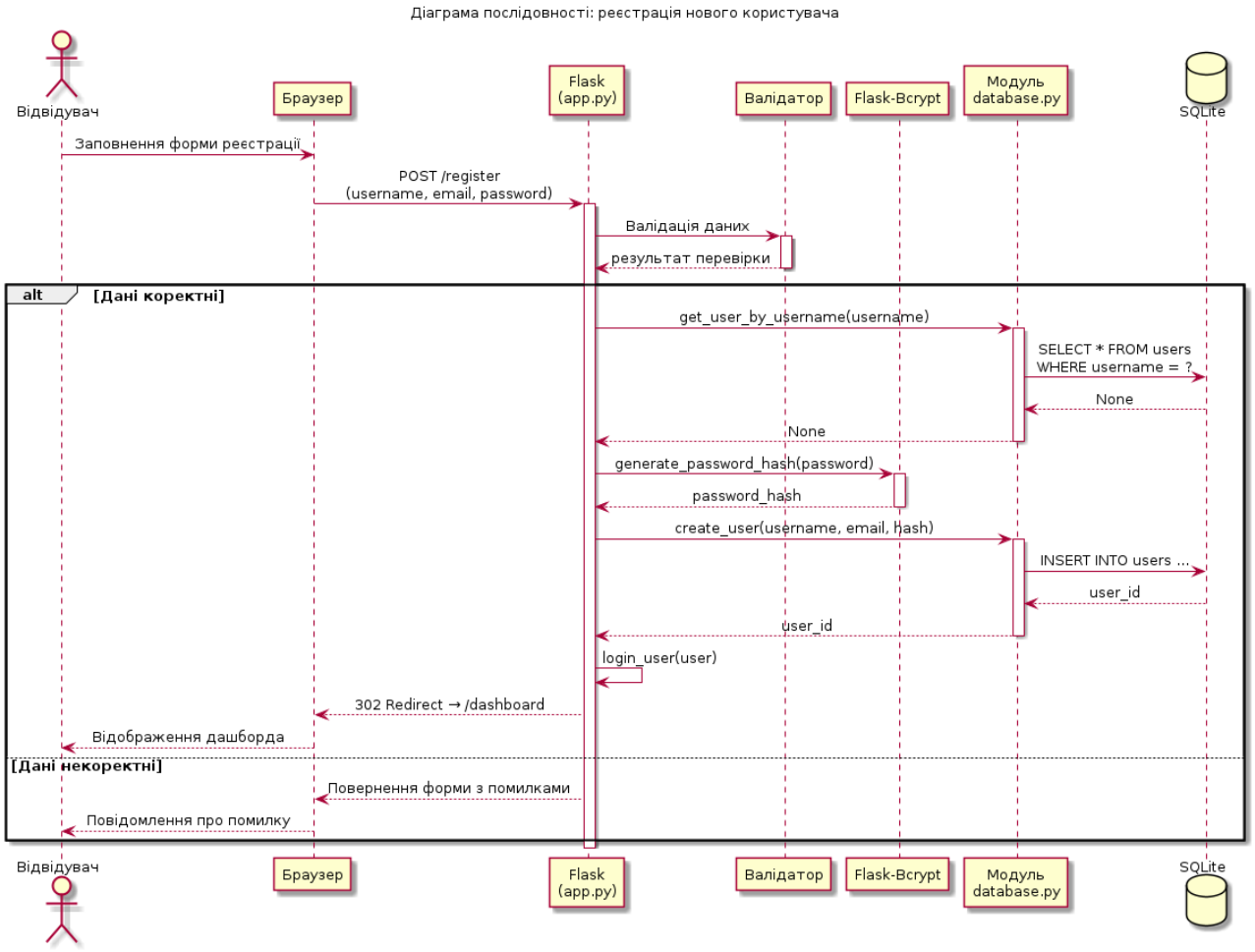


Рисунок 2.4. Діаграма послідовності: реєстрація нового користувача

На рисунку 2.5 зображено сценарій створення нового завдання через REST API. Ключовим елементом є перевірка автентифікації через декоратор @login_required (Flask-Login), що реалізує вимогу безпеки A01 (контроль доступу). У разі успішної автентифікації завдання зберігається у базі даних із прив'язкою до ідентифікатора поточного користувача, після чого клієнт отримує JSON-відповідь та оновлює інтерфейс без повного перезавантаження сторінки (AJAX-підхід).

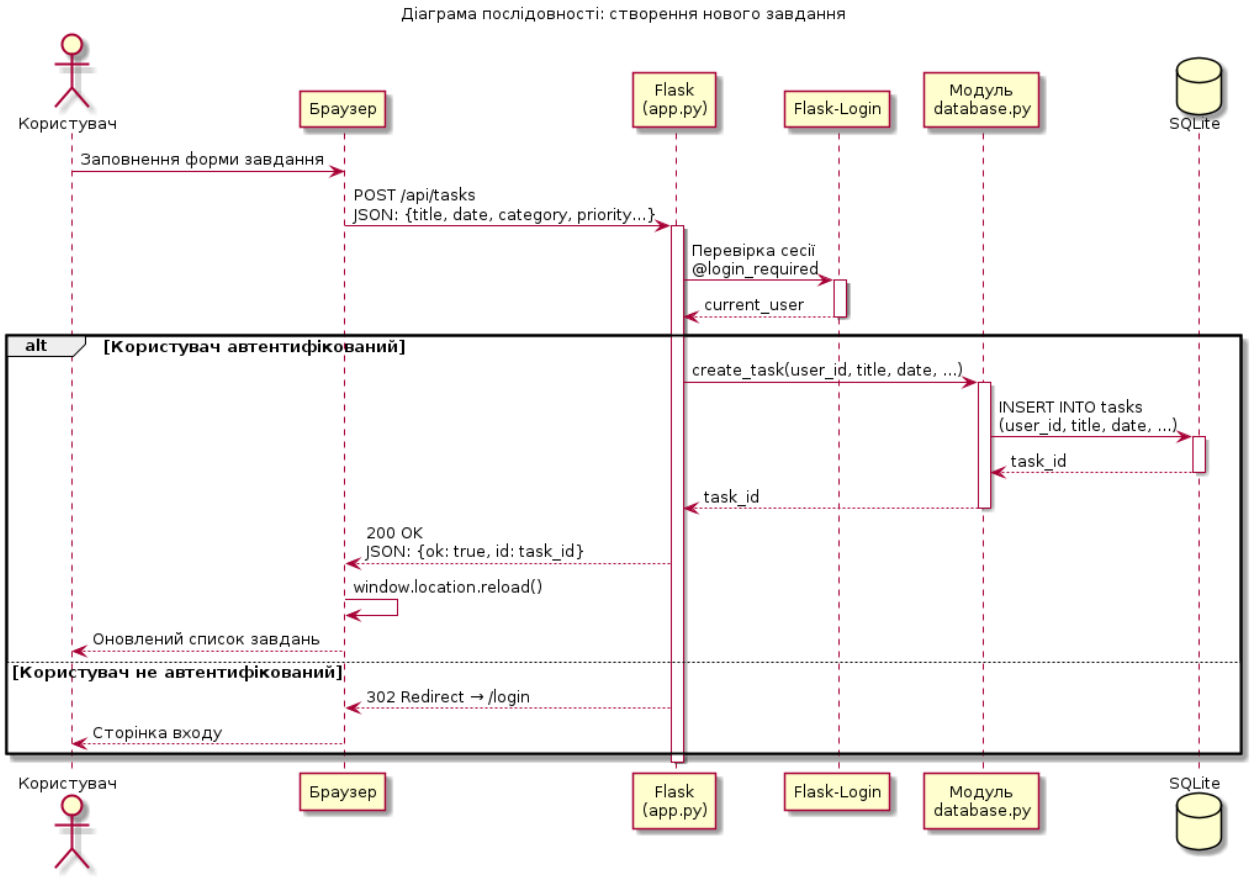


Рисунок 2.5. Діаграма послідовності: створення нового завдання

Діаграма діяльності (Activity Diagram) моделює потік управління та послідовність дій у рамках процесу або алгоритму [19]. На рисунку 2.6 подано діаграму діяльності для процесу роботи з Pomodoro-таймером – одного з ключових інтерактивних інструментів тайм-менеджменту (FR-6).

Алгоритм починається із завантаження персональних налаштувань користувача (тривалість фокусу, перерви, довгої перерви та кількість сесій до довгої перерви) із бази даних. Далі ініціалізується таймер у режимі work. Основний цикл містить зворотний відлік із посекундним оновленням інтерфейсу. Після завершення відліку виконується розгалуження: якщо поточний режим – work, здійснюється POST-запит до API /api/pomodoro/log для збереження сесії в базі даних (таблиця pomodoro_sessions), лічильник сесій збільшується на одиницю, і залежно від кратності кількості сесій здійснюється перехід до довгої або короткої перерви. Якщо поточний режим – перерва, таймер

повертається до режиму work. Така реалізація забезпечує безперервний цикл Pomodoro з автоматичним чергуванням режимів і фіксацією даних для аналітики.

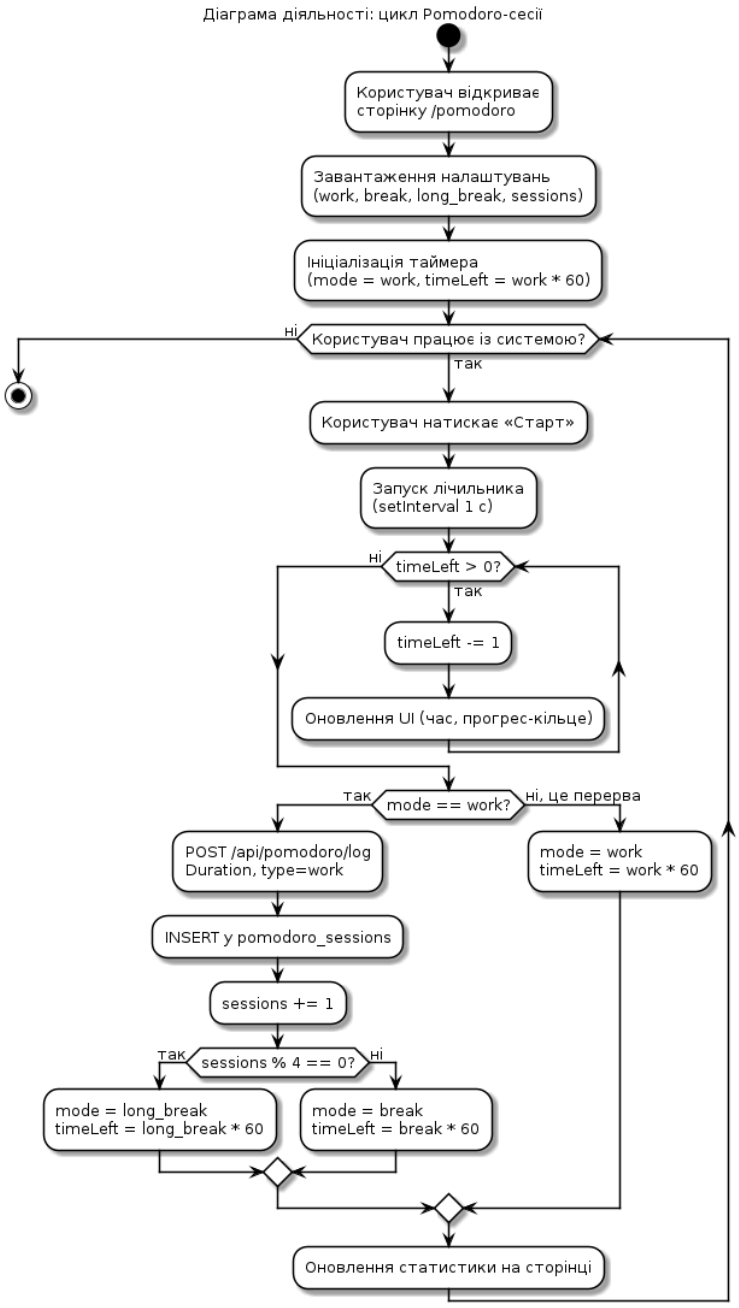


Рисунок 2.6. Діаграма діяльності: цикл Pomodoro-сесії

2.2.3 Діаграма компонентів і розгортання

Діаграма компонентів (Component Diagram) відображає організацію фізичних компонентів системи та їхні залежності [19]. На рисунку 2.7 виокремлено три пакети, що відповідають трьом рівням архітектури: клієнтський рівень (п'ять HTML-шаблонів, CSS-стилі, JavaScript-модулі),

серверний рівень (контролер app.py, модуль database.py, Flask-Login, Flask-Bcrypt, Jinja2 Engine) та рівень даних (сім таблиць SQLite).

Стрілками на діаграмі позначено залежності між компонентами. Шаблон base.html є батьківським для всіх інших шаблонів (механізм успадкування Jinja2). Шаблон app.html включає частковий шаблон _task_row.html, який використовується для відображення рядка завдання у різних контекстах – на дашборді, у списку завдань, у календарі та у матриці Ейзенхауера. Лістинг шаблону сторінки входу login.html наведено в Додатку Б. Контролер app.py викликає Jinja2 Engine через функцію render_template(), використовує Flask-Login для перевірки автентифікації, Flask-Bcrypt для хешування та звертається до database.py для виконання CRUD-операцій. Модуль database.py, у свою чергу, взаємодіє з файлом planner.db через бібліотеку sqlite3 [26].

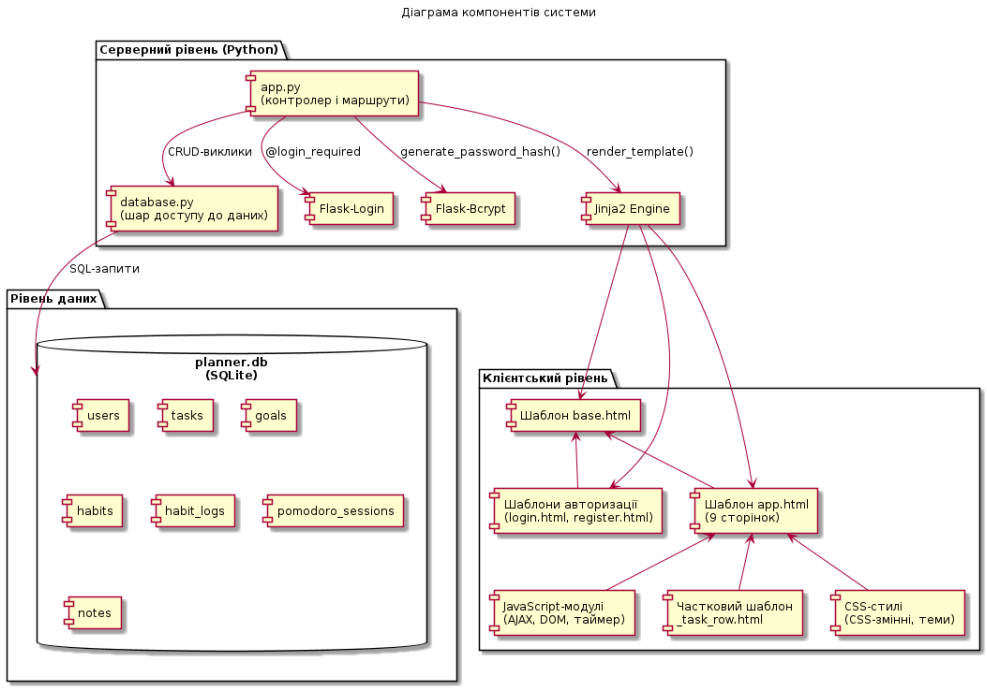


Рисунок 2.7. Діаграма компонентів системи

Діаграма розгортання (Deployment Diagram) описує фізичне розміщення програмних артефактів на апаратних вузлах та канали зв'язку між ними [19]. На рисунку 2.8 зображено два вузли: пристрій користувача (із веб-браузером) та сервер (з операційною системою, середовищем Python 3.10+ та файлом SQLite).

Основною особливістю діаграми розгортання є мінімалістичність інфраструктури: для роботи системи не потрібен окремий сервер бази даних, контейнер Docker чи хмарна інфраструктура – достатньо лише Python-інтерпретатора. Це повністю узгоджується з вимогою NFR-7 (переносимість) та концепцією self-hosted розгортання, обґрунтованою у підрозділі 1.2 як перевага над пропрієтарними хмарними сервісами. Зв'язок між клієнтом та сервером здійснюється по протоколу HTTP (порт 5000) у локальному середовищі або HTTPS при промисловому розгортанні за допомогою зворотного проксі-сервера (Nginx, Caddy) [27].

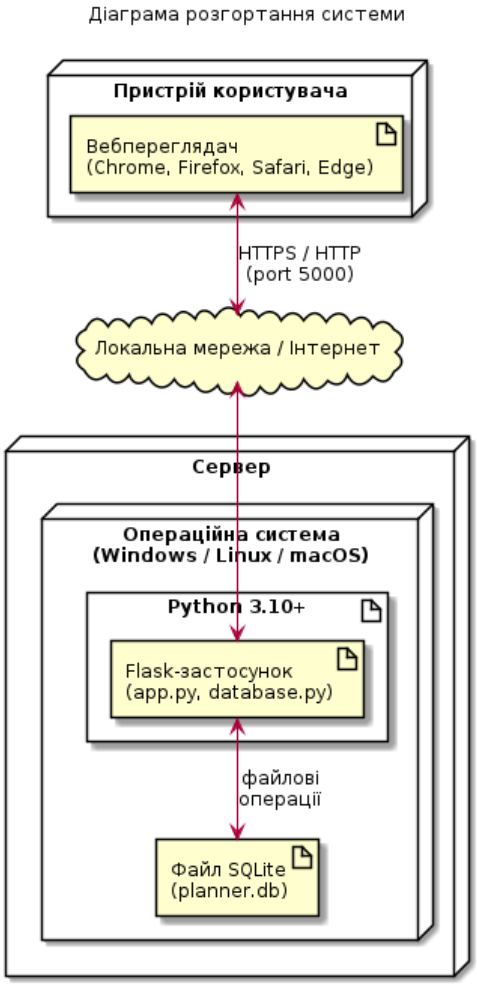


Рисунок 2.8. Діаграма розгортання системи

Побудовані шість UML-діаграм формують комплексну модель системи, що охоплює структурні аспекти (діаграми класів, компонентів, розгортання), поведінкові аспекти (діаграми варіантів використання, послідовності,

діяльності) та забезпечує трасування від функціональних вимог FR-1 – FR-11 до конкретних проєктних рішень. Отримані моделі є основою для наступних етапів – проєктування бази даних та вибору технологій реалізації.

2.3 Проєктування бази даних

Проєктування бази даних є одним із ключових етапів розробки інформаційної системи, оскільки від структури зберігання даних залежать продуктивність, надійність та масштабованість застосунку [18]. У межах цього підрозділу побудовано концептуальну модель у вигляді ER-діаграми, описано логічну та фізичну моделі та проведено аналіз нормалізації відношень.

Концептуальна модель бази даних відображає сутності предметної області, їхні атрибути та зв’язки між ними на рівні, незалежному від конкретної СУБД. На основі діаграми класів (рис. 2.3) та функціональних вимог FR-1 – FR-11, визначених у підрозділі 1.3, побудовано ER-діаграму (рис. 2.9), що включає сім сутностей та шість зв’язків.

Центральною сутністю є users, яка пов’язана з п’ятьма залежними сутностями зв’язком «1 : 0..*»: tasks, goals, habits, pomodoro_sessions та notes. Окремо виділено сутність habit_logs, яка пов’язана з habits зв’язком «1 : 0..*» та зберігає записи про щоденне виконання кожної звички. Така структура дозволяє ефективно обчислювати серію (streak) послідовних днів виконання та формувати візуалізацію тижневого трекера (FR-7).

Усі зовнішні ключі (user_id, habit_id) мають обмеження ON DELETE CASCADE, що забезпечує каскадне видалення залежних записів при видаленні користувача або звички. Це реалізує вимогу безпеки щодо повного видалення персональних даних, описану у підрозділі 1.4.

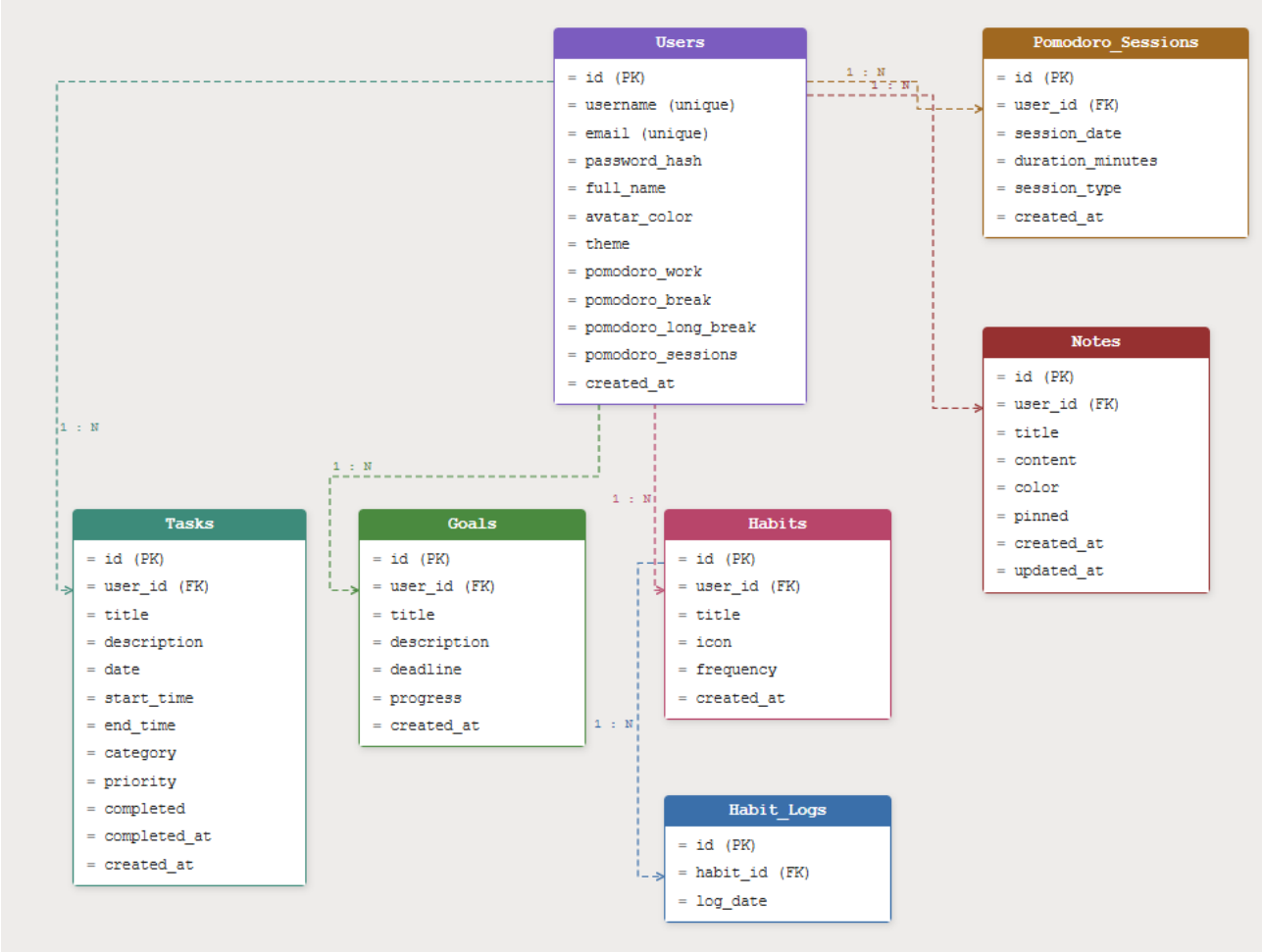


Рисунок 2.9. ER-діаграма бази даних інтерактивного онлайн-планера

Логічна модель бази даних конкретизує ER-діаграму, визначаючи типи даних, обмеження цілісності та індекси для кожної таблиці. На цьому рівні абстракції кожна сутність отримує точні специфікації полів із зазначенням типу (INTEGER, TEXT, TIMESTAMP), обмежень (NOT NULL, UNIQUE, DEFAULT) та зовнішніх ключів із правилами каскадних операцій. Особливу роль відіграють обмеження цілісності на рівні бази даних, які є другим рубежем захисту після серверної валідації: наприклад, обмеження UNIQUE на полях username та email у таблиці users унеможливлює реєстрацію дублікатів навіть при обході перевірок на рівні застосунку. У таблиці 2.1 подано повний опис структури таблиці tasks як найбільш репрезентативної сутності системи. Вона містить найбільшу кількість атрибутів і безпосередньо задіяна в роботі п'яти з дев'яти функціональних модулів: керування завданнями, календарі, фільтрації, матриці Ейзенхауера та аналітиці.

Таблиця 2.1

Структура таблиці tasks

Атрибут	**Тип**	**Обмеження**	**Призначення**
id	INTEGER	PK, AI	Унікальний ідентифікатор запису
user_id	INTEGER	FK, NOT NULL	Зовнішній ключ до таблиці users (CASCADE)
title	TEXT	NOT NULL	Назва завдання
description	TEXT	DEFAULT ""	Опис завдання (необов'язковий)
date	TEXT	NOT NULL	Дата виконання у форматі YYYY-MM-DD
start_time	TEXT	DEFAULT ""	Час початку у форматі HH:MM
end_time	TEXT	DEFAULT ""	Час закінчення
category	TEXT	DEFAULT other	Категорія: work, study, personal, health, meeting, other
priority	TEXT	DEFAULT medium	Пріоритет: high, medium, low
completed	INTEGER	DEFAULT 0	Прапорець виконання (0/1)
completed_at	TIMESTAMP	NULL	Час позначення як виконаного
created_at	TIMESTAMP	DEFAULT NOW	Час створення запису

Як видно з таблиці 2.1, для зберігання дат та часу використано тип TEXT у форматі ISO 8601, що є рекомендованою практикою для SQLite [18]. Фізична модель реалізована у модулі database.py у функції init_db(). Для підвищення продуктивності створено п'ять індексів: idx_tasks_user_date, idx_tasks_user_completed, idx_habits_user, idx_habit_logs_habit та idx_pomodoro_user_date. Ці індекси оптимізують найчастіші запити: відображення дашборда, календаря та аналітики.

Нормалізація бази даних спрямована на усунення надлишковості та аномалій вставки, оновлення і видалення даних. Проаналізуємо відповідність спроєктованої схеми нормальним формам [28].

Перша нормальна форма (1НФ). Усі таблиці задовольняють 1НФ: кожен стовпець містить атомарні значення, рядки унікально ідентифіковані первинними ключами (id), повторювані групи відсутні. Зокрема, дати виконання

звичок винесено в окрему таблицю habit_logs замість зберігання списку дат у полі таблиці habits.

Друга нормальна форма (2НФ). Усі неключові атрибути функціонально залежать від повного первинного ключа. Оскільки кожна таблиця має простий (одностовпцевий) первинний ключ id, часткові залежності неможливі. У таблиці habit_logs складний унікальний індекс (habit_id, log_date) забезпечує запобігання дублюванню.

Третя нормальна форма (3НФ). Транзитивні залежності відсутні: кожен неключовий атрибут залежить безпосередньо від первинного ключа. Наприклад, у таблиці tasks атрибут category не визначає priority і навпаки. Налаштування Pomodoro зберігаються у таблиці users, оскільки функціонально залежать від користувача – виділення окремої таблиці не покращило б нормалізацію.

Спроєктована база даних відповідає третій нормальній формі, що забезпечує мінімальну надлишковість даних та цілісність при операціях вставки, оновлення та видалення.

2.4 Проєктування UX та інтерактивності інтерфейсу

Проєктування користувацького досвіду (UX) є невід'ємною частиною розробки інтерактивних систем [20]. У контексті онлайн-планера UX набуває критичного значення: система, покликана економити час, не має витрачати його на складну навігацію. У підрозділі 1.3 сформульовано вимогу NFR-3 (основні дії доступні не більш ніж за два кліки). Цей підрозділ описує проєктні рішення, спрямовані на її виконання.

2.4.1 Сценарії взаємодії та wireframes

Карта сайту. Навігаційна структура побудована за принципом пласкої ієрархії: дашборд є центральним вузлом, з якого одним кліком доступні всі дев'ять функціональних сторінок (рис. 2.10). Горизонтальна навігаційна панель присутня на кожній сторінці й дозволяє миттєво переходити між модулями. Такий підхід відповідає патерну Hub-and-Spoke [20].

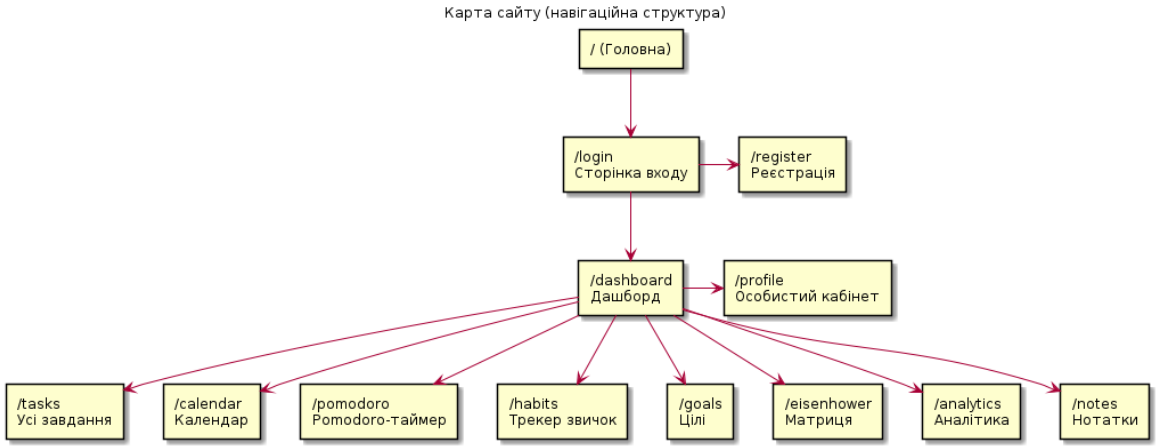


Рисунок 2.10. Карта сайту (навігаційна структура системи)

Сценарій 1: реєстрація та перший вхід. На рисунку 2.11 побудовано діаграму діяльності для першого контакту відвідувача із системою. Сценарій передбачає два альтернативних шляхи: вхід для існуючих та реєстрацію для нових користувачів. Форма входу містить лише два поля, форма реєстрації – чотири. Повідомлення про помилки відображаються за допомогою flash-повідомлень безпосередньо над формою [29].

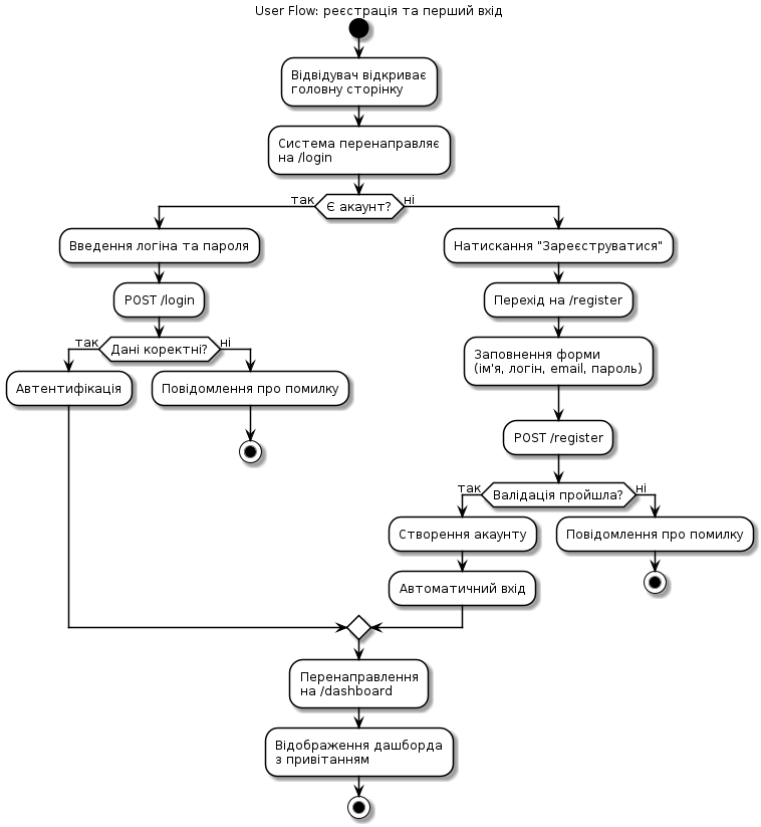


Рисунок 2.11. User Flow: реєстрація та перший вхід

Сценарій 2: створення та керування завданням. Рисунок 2.12 відображає повний життєвий цикл завдання. Ключовим рішенням є використання модального вікна для створення та редагування замість переходу на окрему сторінку. Після створення завдання автоматично з'являється у чотирьох контекстах: на дашборді, в календарі, у списку та в матриці Ейзенхауера.

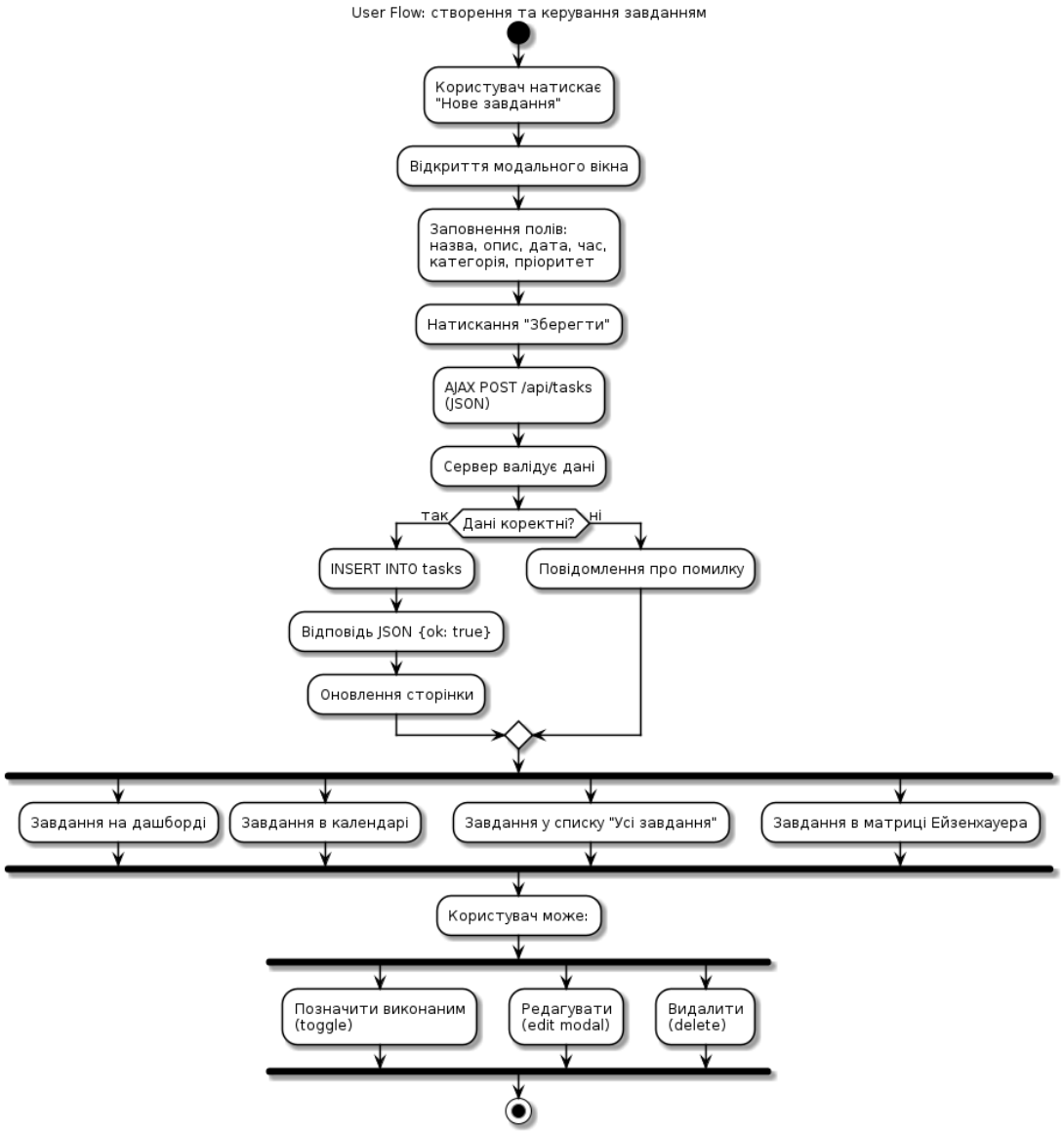


Рисунок 2.12. User Flow: створення та керування завданням

Wireframe дашборда. На рисунку 2.13 подано каркасну схему (wireframe) дашборда. Структура побудована за принципом інформаційної ієрархії: навігація, блок статистики (4 картки), прогрес-кільце та список завдань на сьогодні. Кнопка створення розміщена у правому верхньому куті [20].

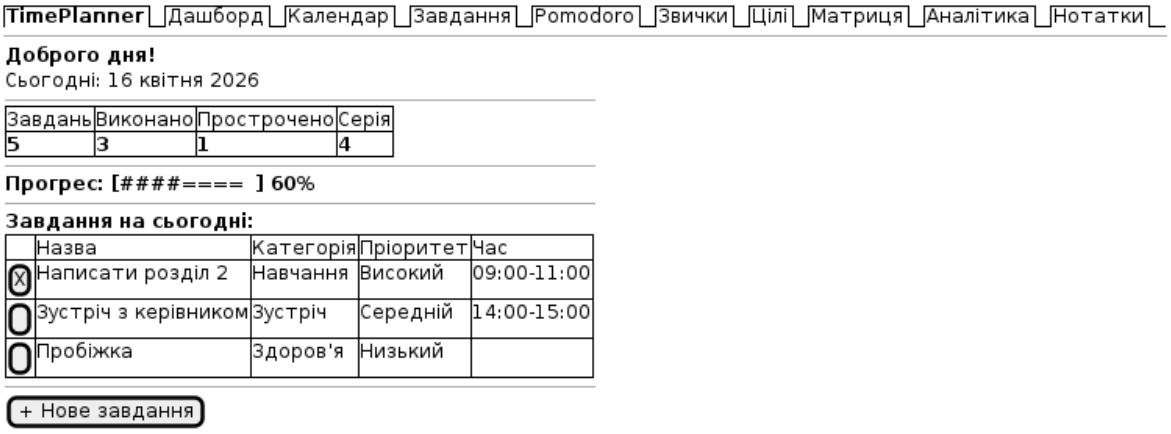


Рисунок 2.13. Wireframe дашборда (каркасна схема)

Wireframe модального вікна. На рисунку 2.14 подано каркасну схему модального вікна створення завдання. Форма містить сім полів: назва, опис, дата/час, категорія (кнопки-перемикачі) та пріоритет (три кнопки). Вибір через кнопки замість випадних списків зменшує кількість кліків та когнітивне навантаження.

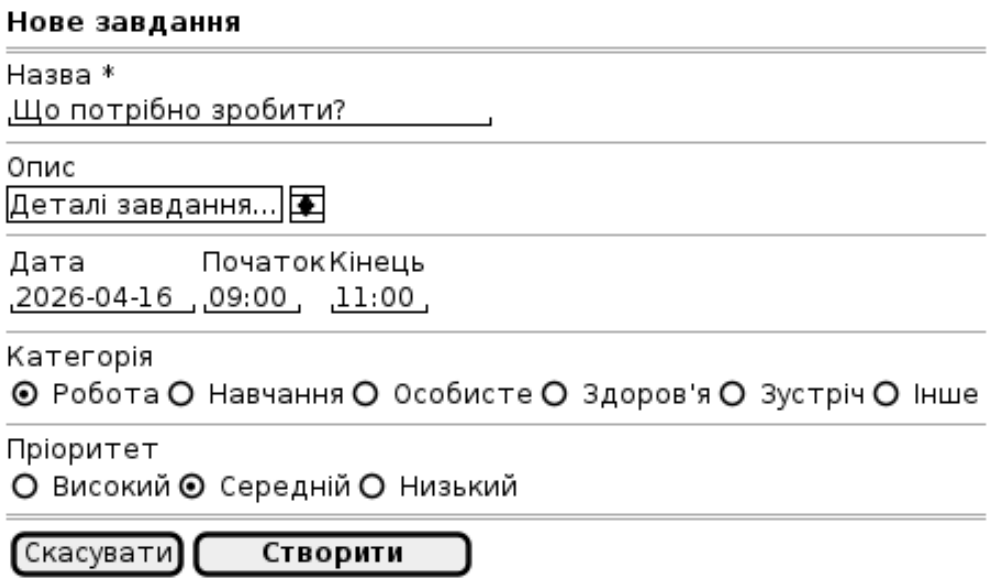


Рисунок 2.14. Wireframe модального вікна створення завдання

2.4.2 Принципи інтерактивного зворотного зв'язку й адаптивності

Інтерактивність інтерфейсу – здатність системи миттєво реагувати на дії користувача – є визначальною характеристикою сучасних веб-додатків. У

розробленій системі інтерактивний зворотний зв’язок реалізовано на кількох рівнях.

Динамічне оновлення без перезавантаження (AJAX). Усі CRUD-операції виконуються через REST API за допомогою Fetch API. При натисканні на чекбокс завдання JavaScript надсилає POST-запит на /api/tasks/{id}/toggle та миттєво змінює візуальний стан елемента. Час реакції при локальному розгортанні – 10–50 мс.

Pomodoro-таймер як інтерактивний компонент. Таймер реалізовано на стороні клієнта з посекундним оновленням цифрового відображення та SVG-кільця прогресу. Зміна кольору сигналізує про режим (фіолетовий – фокус, зелений – перерва, жовтий – довга перерва). Після завершення сесії автоматично надсилається AJAX-запит для збереження в базі.

Трекер звичок із візуальним зворотним зв’язком. Тижневий трекер – горизонтальний ряд із семи інтерактивних кнопок. При натисканні кнопка миттєво змінює колір та іконку (optimistic UI update), одночасно надсилаючи AJAX-запит на сервер.

Матриця Ейзенхауера як візуальний інструмент. Відображається як сітка з чотирьох карток із кольоровим верхнім бордюром. Розподіл завдань за квадрантами – автоматичний (пріоритет high = терміново, категорії work/study/meeting = важливо), що відповідає FR-9.

Адаптивність та підтримка тем. CSS-змінні визначають палітру. Перемикання теми – зміна класу на body без перезавантаження. Адаптивність для мобільних – CSS media queries: навігація стає прокручуваною при ширині < 640 px, таблиці – одноколонковими (NFR-8).

У таблиці 2.2 узагальнено проєктні рішення UX та їх відповідність нефункціональним вимогам.

Таблиця 2.2

Проєктні рішення UX та їх обґрунтування

Рішення	Обґрунтування	Вимога
Пласка навігація (Hub-and-Spoke)	Усі модулі доступні за один клік з будь-якої сторінки	NFR-3
Модальні вікна для CRUD	Збереження контексту; зменшення переходів	NFR-3
AJAX (Fetch API)	Миттєва реакція без перезавантаження	NFR-1, NFR-3
Кнопки-перемикачі (tag buttons)	Усі варіанти видимі одночасно	NFR-3
CSS-змінні для тем	Миттєве перемикання; єдина точка конфігурації	NFR-3
SVG-анімації (Pomodoro)	Плавний візуальний зворотний зв’язок	NFR-1
Optimistic UI update	Негайна візуальна зміна при toggle-операціях	NFR-1, NFR-3
Media queries (від 320 px)	Коректне відображення на мобільних	NFR-8

Як видно з таблиці 2.2, кожне проєктне рішення пов’язане з конкретною нефункціональною вимогою, що забезпечує трасування від вимог до реалізації. Описані принципи UX та інтерактивності будуть детально реалізовані у третьому розділі.

2.5 Проєктування підсистеми безпеки

Підсистема безпеки є наскрізним компонентом архітектури, що пронизує всі три рівні системи – представлення, бізнес-логіки та даних. У підрозділі 1.4 сформульовано вимоги безпеки на основі OWASP Top 10:2021; у цьому підрозділі виконано структурне моделювання загроз за методологією STRIDE [21] та описано архітектуру захисних механізмів.

STRIDE – це методологія моделювання загроз, розроблена компанією Microsoft, яка класифікує загрози за шістьма категоріями: Spoofing (підробка ідентичності), Tampering (підміна даних), Repudiation (відмова від авторства), Information Disclosure (розкриття інформації), Denial of Service (відмова в обслуговуванні) та Elevation of Privilege (підвищення привілеїв) [21].

Моделювання загроз виконується на основі діаграми потоків даних (DFD), яка визначає межі довіри між компонентами системи.

На рисунку 2.15 побудовано модель загроз STRIDE для інтерактивного онлайн-планера. Модель відображає три рівні системи, зовнішнього актора (зловмисника) та шість категорій загроз, спрямованих на конкретні компоненти. Стрілками позначено вектори атаки: підробка ідентичності (S) спрямована на форми автентифікації, підміна даних (T) – на клієнтський код через XSS, розкриття інформації (I) – на Flask-контролер, підвищення привілеїв (E) – на систему сесій Flask-Login, відмова від авторства (R) – на REST API, а відмова в обслуговуванні (D) – на базу даних [30].

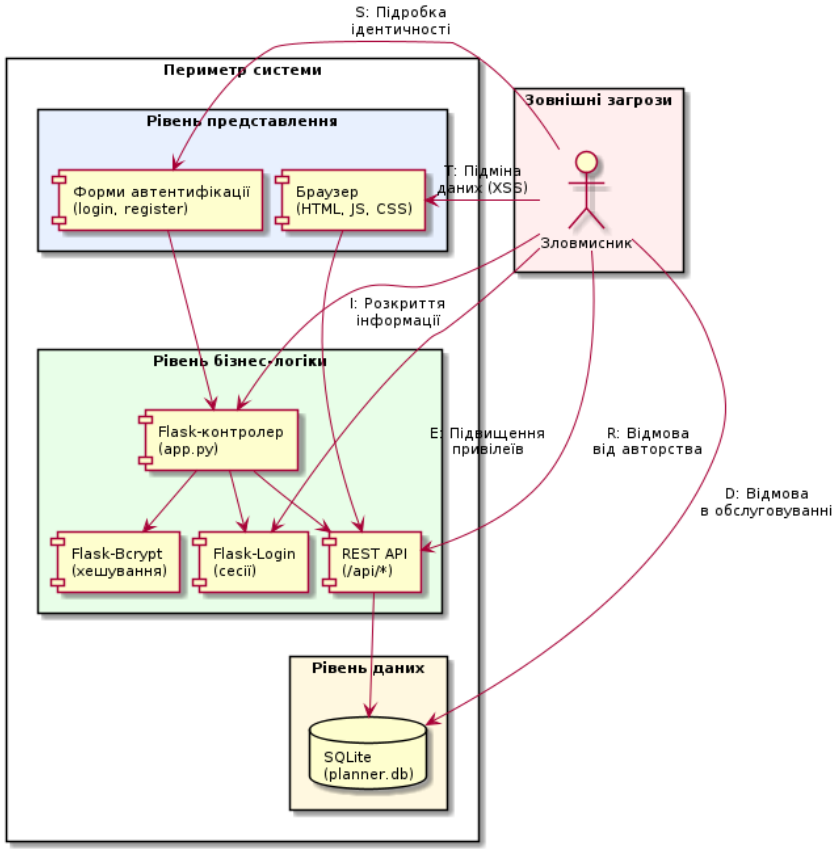


Рисунок 2.15. Модель загроз STRIDE для інтерактивного онлайн-планера

На рисунку 2.16 побудовано діаграму потоків даних (DFD рівня 1), яка відображає сім основних процесів системи, зовнішню сутність (користувача) та сховище даних (planner.db). Кожен потік даних позначено конкретним типом

інформації, що передається. DFD є основою для визначення меж довіри та ідентифікації точок, у яких необхідно застосувати захисні механізми.

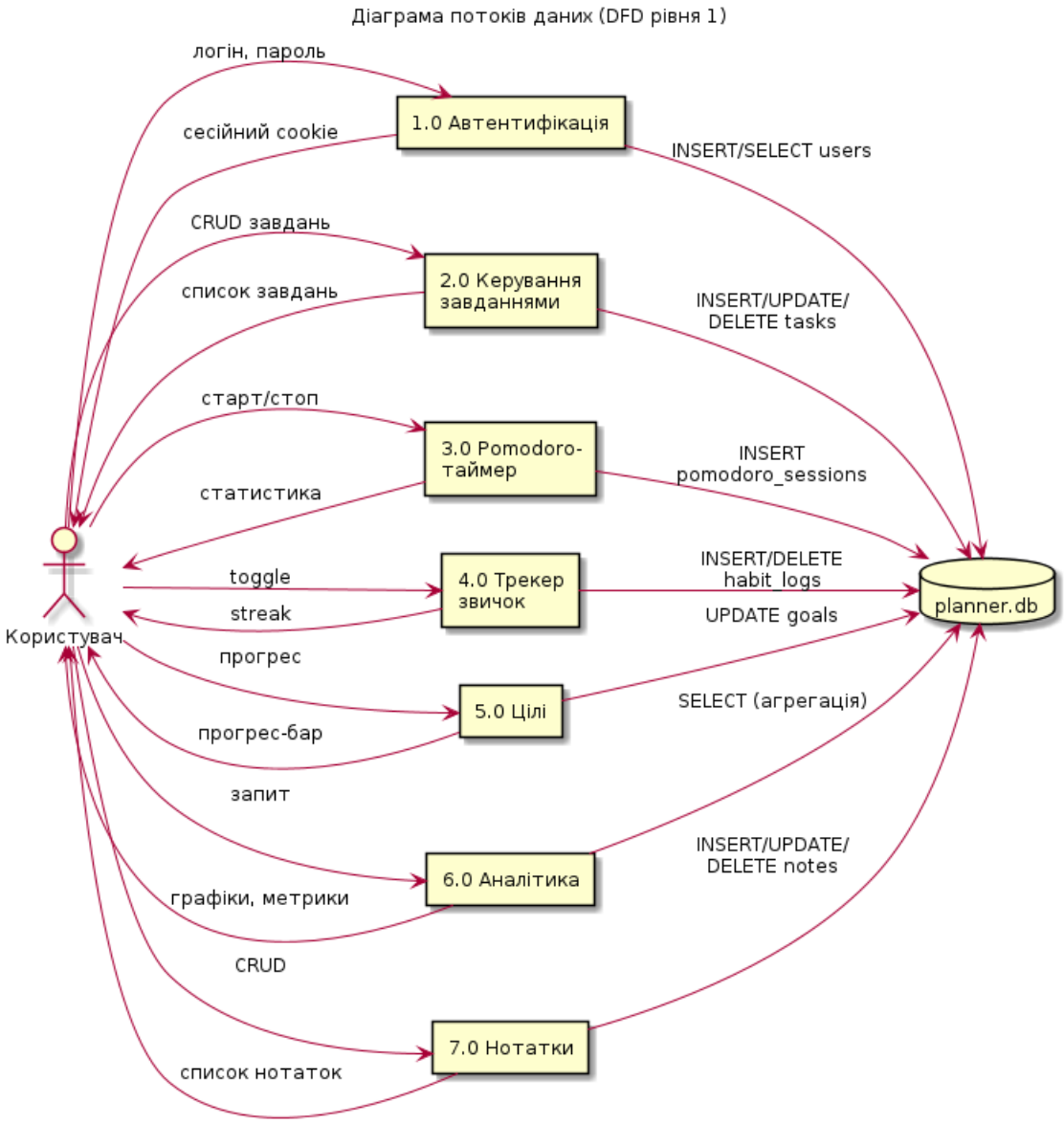


Рисунок 2.16. Діаграма потоків даних (DFD рівня 1)

У таблиці 2.3 подано повний аналіз загроз за STRIDE із зазначенням категорії, опису загрози, оцінки ризику та конкретного контрзаходу, реалізованого в системі.

Оцінку ризику для кожної загрози визначено за двома критеріями: імовірністю реалізації атаки в контексті локально розгорнутого веб-застосунку та потенційними наслідками для конфіденційності персональних даних

користувача. Саме поєднання цих двох факторів зумовило присвоєння статусу «високий ризик» чотирьом із шести категорій STRIDE.

Таблиця 2.3

Аналіз загроз за методологією STRIDE

Код	**Категорія**	**Опис загрози**	**Ризик**	**Контрзахід**
S	Spoofing	Зловмисник видає себе за іншого користувача	Високий	bcrypt (cost 12); Flask-Login сесії; валідація email; мін. довжина пароля 6
T	Tampering	Ін'єкція шкідливого коду (XSS) або SQL	Високий	Jinja2 auto-escape; параметризовані SQL sqlite3; валідація введення
R	Repudiation	Користувач заперечує виконання дії	Низький	Поля created_at, completed_at; журналювання Flask
I	Info Disclosure	Витік персональних даних або хешів	Високий	Фільтр user_id в SQL; HttpOnly cookies; DEBUG=False; лише bcrypt-хеші
D	DoS	Перевантаження сервера запитами	Середній	Rate limiting на /login; WAL-режим SQLite; PRAGMA journal_mode=WAL
E	Elev. of Privilege	Доступ до чужих даних через маніпуляцію ID	Високий	@login_required; WHERE user_id=? у кожному запиті (захист від IDOR)

Як видно з таблиці 2.3, чотири з шести категорій STRIDE оцінено як загрози високого ризику. Для кожної з них передбачено конкретний технічний контрзахід.

Процес автентифікації побудовано на основі бібліотеки Flask-Login, яка реалізує патерн сесійної автентифікації на стороні сервера [22]. Після успішної перевірки облікових даних створюється серверна сесія, ідентифікатор якої зберігається у cookie-файлі з атрибутом HttpOnly, що робить його недоступним для JavaScript-коду на стороні клієнта і запобігає крадіжці сесій через XSS-атаки.

Алгоритм автентифікації: 1) користувач надсилає POST-запит із логіном та паролем; 2) сервер шукає запис у таблиці users; 3) якщо знайдено, викликається bcrypt.check_password_hash() з constant-time comparison; 4) при збігу Flask-Login створює сесію; 5) при неспівпадінні – повідомлення без зазначення, яке поле невірне (захист від enumeration attack). Цей алгоритм відображено на діаграмі послідовності (рис. 2.4).

Авторизація реалізована на двох рівнях. На першому – декоратор @login_required перевіряє наявність активної сесії. На другому – кожен SQL-запит у database.py містить WHERE user_id = ?, де user_id передається із current_user. Навіть при маніпуляції ідентифікаторами у URL зловмисник не отримає чужих даних, оскільки фільтрація на рівні БД.

Хешування паролів виконується алгоритмом bcrypt із cost factor 12, тобто 2^12 = 4096 ітерацій Blowfish. Час перебору одного хешу на NVIDIA RTX 4090 становить приблизно 105 хешів/с [16], що робить атаку перебору на пароль 8+ символів практично нездійсненною. Кожен хеш містить унікальну випадкову сіль.

На рисунку 2.17 подано загальну архітектуру підсистеми безпеки, яка відображає розміщення захисних механізмів на кожному з трьох рівнів системи.

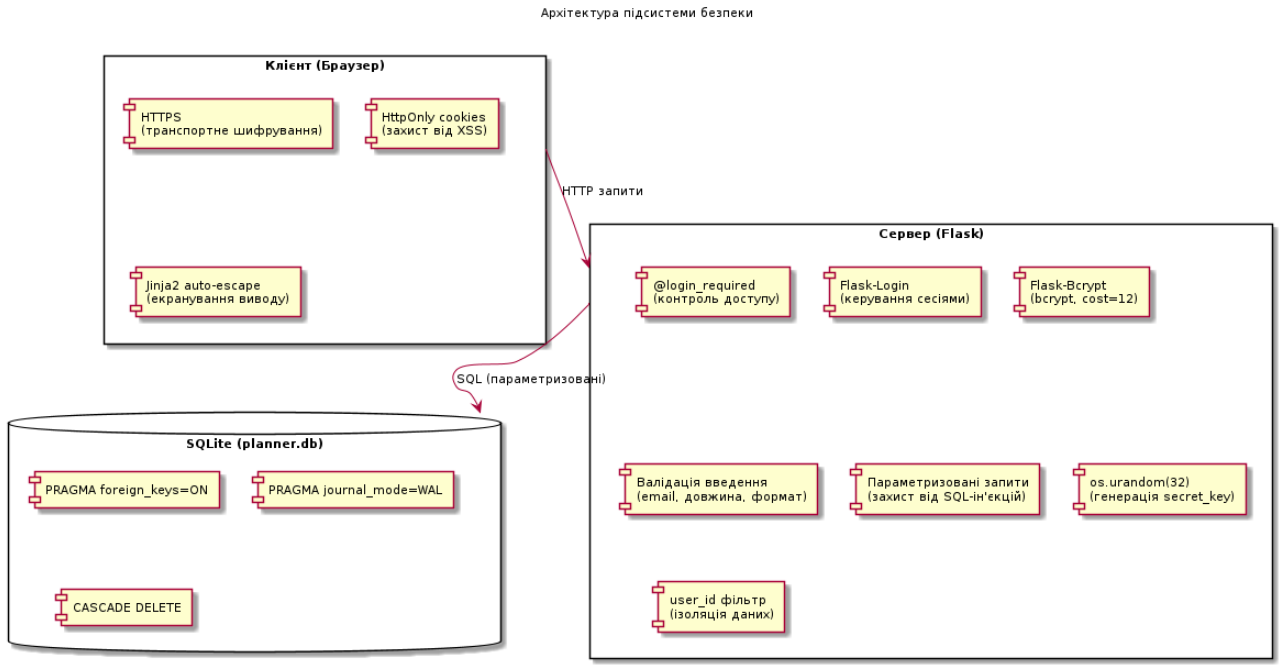


Рисунок 2.17. Архітектура підсистеми безпеки

Конкретні реалізації захисту від кожної з десяти категорій OWASP Top 10:2021 описано у таблиці 1.5 підрозділу 1.4. Нижче виділено ключові архітектурні рішення.

Захист від SQL-ін'єкцій (A03). Усі SQL-запити у database.py використовують параметризовані запити через знаки підстановки (?) бібліотеки

sqlite3. Жоден запит не формується конкатенацією рядків. Функція get_tasks() динамічно будує WHERE, але кожен параметр додається через список params. Навіть введення '; DROP TABLE tasks; -- буде інтерпретовано як рядковий літерал.

Захист від XSS (A03). Jinja2 за замовчуванням виконує auto-escape усіх змінних через {{ }}. Символи <, >, & замінюються на HTML-сутності. У системі відсутні місця з конструкцією {{ змінна | safe }}, що виключає XSS на рівні шаблонів.

Захист від порушеного контролю доступу (A01). Авторизація на двох рівнях: @login_required та user_id фільтр. API-маршрути (/api/tasks/{id}) перевіряють належність ресурсу поточному користувачеві перед UPDATE або DELETE. Це запобігає IDOR-атакам.

Захист від небезпечної конфігурації (A05). secret_key генерується os.urandom(32). DEBUG вимкнено для production. SESSION_COOKIE_HTTPONLY=True. PRAGMA foreign_keys=ON та journal_mode=WAL виконуються при кожному підключенні в get_db().

2.6 Обґрунтування вибору технологій реалізації

Вибір технологій визначається вимогами (підрозділи 1.3–1.4), архітектурою (підрозділи 2.1–2.5) та обмеженнями предметної області. Загальну структуру стеку подано на рисунку 2.18. Ключовим критерієм відбору кожного компонента стеку була відповідність принципу мінімальної достатності: перевага надавалася інструментам, які повністю задовольняють сформульовані вимоги без надлишкової складності та зайвих залежностей. Додатковими критеріями слугували зрілість технології, наявність активної спільноти підтримки, відкрита ліцензія та можливість локального розгортання без хмарної інфраструктури. Сукупність цих критеріїв дозволила сформувати цілісний стек, у якому кожен компонент доповнює інші та не створює архітектурних суперечностей.

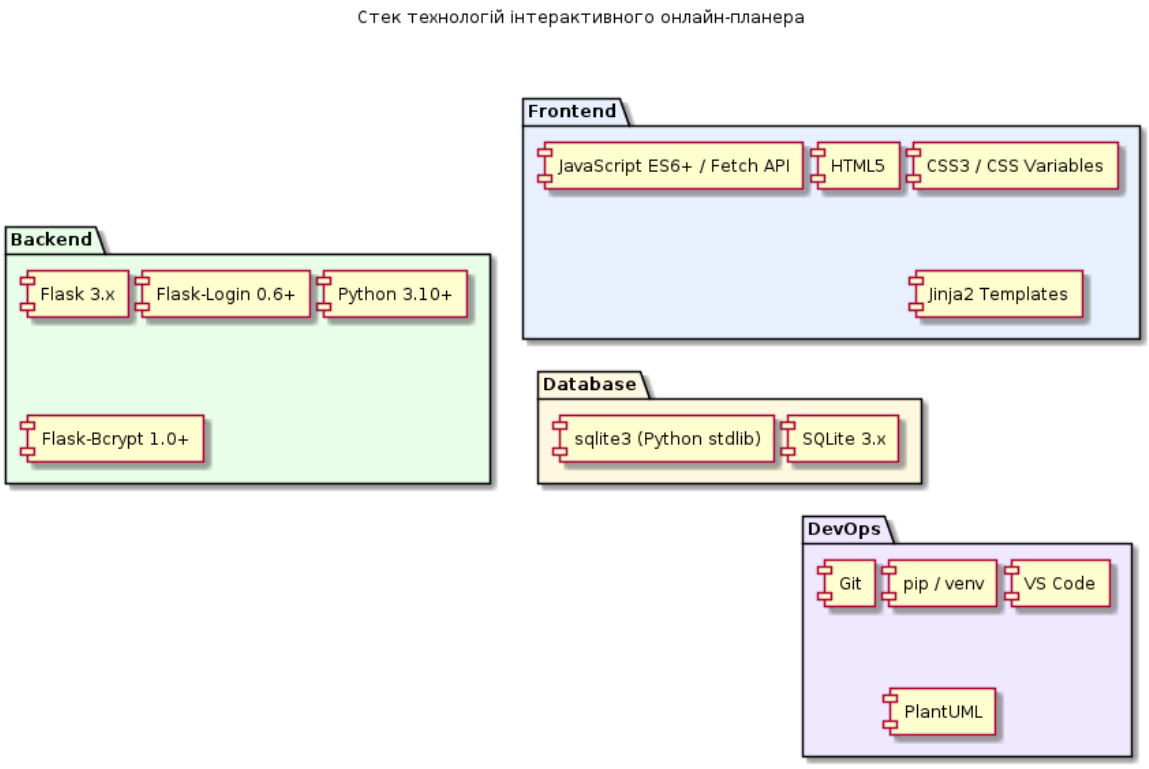


Рисунок 2.18. Стек технологій інтерактивного онлайн-планера

Обрано Python 3.10+: перше місце в індексі TIOBE 2025 [23], лаконічний синтаксис, вбудований sqlite3, широка екосистема. Серверний фреймворк – Flask 3.x: мікрофреймворк із мінімальним ядром (маршрутизація, Jinja2, обробка запитів), розширюваний через Flask-Login, Flask-Bcrypt [24]. У порівнянні з Django – нижчий поріг входу та менший обсяг коду (~450 рядків app.py).

Таблиця 2.4

Порівняння серверних фреймворків

Критерій	**Flask**	**Django**	**FastAPI**
Тип	Мікрофреймворк	Повнофункціональний	Мікрофреймворк
Поріг входу	Низький	Середній	Середній
Шаблонізація	Jinja2 (вбудована)	DTL (вбудована)	Немає (API-only)
ORM	Опціонально	Вбудована	Опціонально
REST API	Вбудовано (jsonify)	DRF (додатково)	Вбудовано
Серверний рендеринг	+	+	–
Розмір проєкту	Малий – середній	Середній – великий	Малий – середній

Flask оптимальний: серверний рендеринг Jinja2, пряма робота з sqlite3 без ORM, вбудований REST API, низький поріг входу. FastAPI не підтримує серверний рендеринг, Django надмірний для 7 таблиць.

SQLite 3.x: zero-configuration, файлова архітектура, ACID + WAL. Модуль sqlite3 входить до стандартної бібліотеки Python – відсутність зовнішніх залежностей (NFR-7). У database.py реалізовано власний шар абстракції з контекстним менеджером get_db() для автоматичного закриття з'єднань та відкату транзакцій [18].

Ванільний JavaScript ES6+ із Fetch API замість React/Vue/Angular – свідоме рішення для серверного рендерингу. CSS3 із CSS-змінними для тем (NFR-3). Адаптивність через Flexbox, Grid, Media Queries без Bootstrap/Tailwind. Шрифти: DM Sans (основний), JetBrains Mono (числові дані, таймер) з Google Fonts.

Flask-Login 0.6+: керування сесіями, @login_required, current_user, remember me [22]. Flask-Bcrypt 1.0+: обгортка над bcrypt для Python [16]. Обидві – стабільні проєкти з відкритим кодом [48].

VS Code з Python-дебагером (.vscode/launch.json – F5 для Flask). pip + venv для ізоляції залежностей. PlantUML для UML-моделювання (18 діаграм, рис. 2.1–2.18). Git з .gitignore для версіонування.

2.7 Висновки до розділу 2

У другому розділі виконано комплексне проєктування інтерактивного онлайн-планера. Основні результати полягають у наступному.

По-перше, обґрунтовано вибір трирівневої клієнт-серверної архітектури (рис. 2.1): рівень представлення (Jinja2 + CSS + Fetch API), рівень бізнес-логіки (Flask MVC + REST API) та рівень даних (SQLite з WAL).

По-друге, побудовано шість UML-діаграм: use case з 13 варіантами (рис. 2.2), класів із 7 класами (рис. 2.3), дві діаграми послідовності (рис. 2.4–2.5), діяльності Pomodoro (рис. 2.6), компонентів (рис. 2.7) та розгортання (рис. 2.8).

По-третє, спроєктовано реляційну базу даних із 7 таблиць, 6 зв'язків та 5 індексів (рис. 2.9, таблиця 2.1). Доведено відповідність 3НФ.

По-четверте, спроєктовано UX: карта сайту Hub-and-Spoke (рис. 2.10), два user flows (рис. 2.11–2.12), два wireframes (рис. 2.13–2.14) та 8 UX-рішень (таблиця 2.2).

По-п'яте, виконано моделювання загроз STRIDE (рис. 2.15, таблиця 2.3), побудовано DFD (рис. 2.16) та архітектуру безпеки (рис. 2.17). Описано дворівневу авторизацію, bcrypt (cost 12), захист від SQL-ін'єкцій, XSS, IDOR.

По-шосте, обґрунтовано стек: Python 3.10+ / Flask 3.x (таблиця 2.4) / SQLite / JS Fetch API / Flask-Login + Flask-Bcrypt / VS Code / PlantUML / Git. Усього у розділі 18 рисунків та 4 таблиці.

Отримані проєктні рішення формують повну основу для реалізації системи у третьому розділі.

РОЗДІЛ 3

РОЗРОБКА ТА ТЕСТУВАННЯ ПЛАНЕРА

3.1 Реалізація бази даних та серверної логіки

У цьому підрозділі описано практичну реалізацію серверної частини інтерактивного онлайн-планера на основі проєктних рішень, сформульованих у другому розділі. Реалізація виконана мовою Python з використанням фреймворку Flask та СУБД SQLite. Серверна частина складається з двох основних модулів: database.py (шар доступу до даних) та app.py (контролер і маршрути). Загальний обсяг серверного коду становить приблизно 900 рядків, з яких 450 припадає на модуль database.py та 450 – на app.py [49].

Фізична структура бази даних реалізована у функції init_db() модуля database.py, яка виконує SQL-скрипт створення семи таблиць при першому запуску застосунку. На рисунку 3.1 подано фрагмент цієї функції, що демонструє створення таблиць users та tasks. Кожна таблиця створюється з конструкцією IF NOT EXISTS, що забезпечує ідемпотентність – повторний виклик функції не призведе до помилки та не знищить існуючі дані. Повний лістинг модуля database.py наведено в Додатку В.

```
def init_db():
    """Ініціалізація структури бази даних."""
    with get_db() as conn:
        conn.executescript("""
        -- Таблиця користувачів
        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            username TEXT NOT NULL UNIQUE,
            email TEXT NOT NULL UNIQUE,
            password_hash TEXT NOT NULL,
            full_name TEXT DEFAULT '',
            avatar_color TEXT DEFAULT '#6366f1',
            theme TEXT DEFAULT 'dark',
            pomodoro_work INTEGER DEFAULT 25,
            pomodoro_break INTEGER DEFAULT 5,
            pomodoro_long_break INTEGER DEFAULT 15,
            pomodoro_sessions INTEGER DEFAULT 4,
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        );

        -- Таблиця завдань
        CREATE TABLE IF NOT EXISTS tasks (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            user_id INTEGER NOT NULL,
            title TEXT NOT NULL,
            description TEXT DEFAULT '',
            date TEXT NOT NULL,
            start_time TEXT DEFAULT ''
```

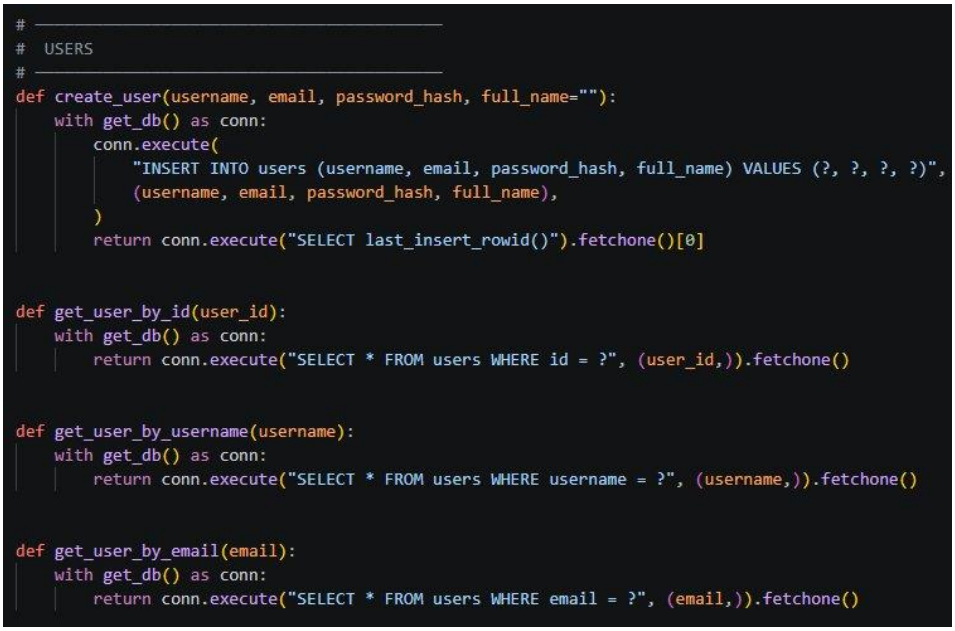
Рисунок 3.1. Функція init_db(): створення таблиць users та tasks із обмеженнями цілісності

Як видно з рисунку 3.1, таблиця users містить 12 стовпців, включно з полями персоналізації (avatar_color, theme) та індивідуальними налаштуваннями Pomodoro-таймера (pomodoro_work, pomodoro_break, pomodoro_long_break, pomodoro_sessions). Обмеження UNIQUE на полях username та email запобігає реєстрації дублікатів на рівні бази даних, що є другим рівнем захисту після серверної валідації. Таблиця tasks містить зовнішній ключ user_id із обмеженням NOT NULL, що реалізує зв'язок «один до багатьох» між користувачем та його завданнями, описаний на ER-діаграмі (рис. 2.9).

Для взаємодії з базою даних реалізовано контекстний менеджер get_db(), який забезпечує: 1) автоматичне створення з'єднання з базою даних; 2) встановлення row_factory = sqlite3.Row для доступу до стовпців за іменами; 3) увімкнення режиму WAL (Write-Ahead Logging) через PRAGMA journal_mode=WAL для підвищення надійності та продуктивності при паралельних операціях читання; 4) увімкнення зовнішніх ключів через PRAGMA foreign_keys=ON для забезпечення каскадного видалення; 5) автоматичний commit при успішному завершенні або rollback при виникненні виключення; 6) гарантоване закриття з'єднання через блок finally. Такий підхід відповідає нефункціональним вимогам NFR-4 (надійність) та NFR-6 (супроводжуваність).

Модуль database.py містить набір функцій для виконання CRUD-операцій (Create, Read, Update, Delete) над кожною з семи таблиць. Усі функції побудовані за єдиним шаблоном: приймають user_id як обов'язковий параметр, використовують параметризовані SQL-запити [31] через знаки підстановки (?) та працюють через контекстний менеджер get_db(). На рисунку 3.2 подано реалізацію функцій для таблиці users.

Єдиний шаблон побудови CRUD-функцій суттєво спрощує супроводжуваність коду та мінімізує ризик введення вразливостей, оскільки перевірений патерн із параметризованими запитами та обов'язковим фільтром user_id застосовується послідовно в усіх 42 SQL-запитах модуля.



```
# ──────────────────────────────────────
#  USERS
# ──────────────────────────────────────
def create_user(username, email, password_hash, full_name=""):
    with get_db() as conn:
        conn.execute(
            "INSERT INTO users (username, email, password_hash, full_name) VALUES (?, ?, ?, ?)",
            (username, email, password_hash, full_name),
        )
        return conn.execute("SELECT last_insert_rowid()").fetchone()[0]


def get_user_by_id(user_id):
    with get_db() as conn:
        return conn.execute("SELECT * FROM users WHERE id = ?", (user_id,)).fetchone()


def get_user_by_username(username):
    with get_db() as conn:
        return conn.execute("SELECT * FROM users WHERE username = ?", (username,)).fetchone()


def get_user_by_email(email):
    with get_db() as conn:
        return conn.execute("SELECT * FROM users WHERE email = ?", (email,)).fetchone()
```

Рисунок 3.2. CRUD-функції для роботи з таблицею users

Як видно з рисунку 3.2, функція create_user() виконує INSERT із чотирма параметрами (username, email, password_hash, full_name), переданими через кортеж, що виключає можливість SQL-ін'єкції навіть при зловмисних вхідних даних. Функції get_user_by_id(), get_user_by_username() та get_user_by_email() виконують пошук за різними ключами і повертають об'єкт sqlite3.Row або None. Ці функції використовуються модулем автентифікації: get_user_by_username() і get_user_by_email() – при вході в систему, get_user_by_id() – при завантаженні користувача із сесії через Flask-Login [32].

Аналогічний підхід застосовано для решти таблиць. На рисунку 3.3 подано функцію create_task(), яка демонструє роботу з необов'язковими параметрами через механізм **kwargs.

```
def create_task(user_id, title, task_date, **kwargs):
    with get_db() as conn:
        conn.execute(
            """INSERT INTO tasks (user_id, title, date, description, start_time, end_time, category, priority)
               VALUES (?, ?, ?, ?, ?, ?, ?, ?)""",
            (user_id, title, task_date,
             kwargs.get("description", ""),
             kwargs.get("start_time", ""),
             kwargs.get("end_time", ""),
             kwargs.get("category", "other"),
             kwargs.get("priority", "medium")),
        )
        return conn.execute("SELECT last_insert_rowid()").fetchone()[0]
```

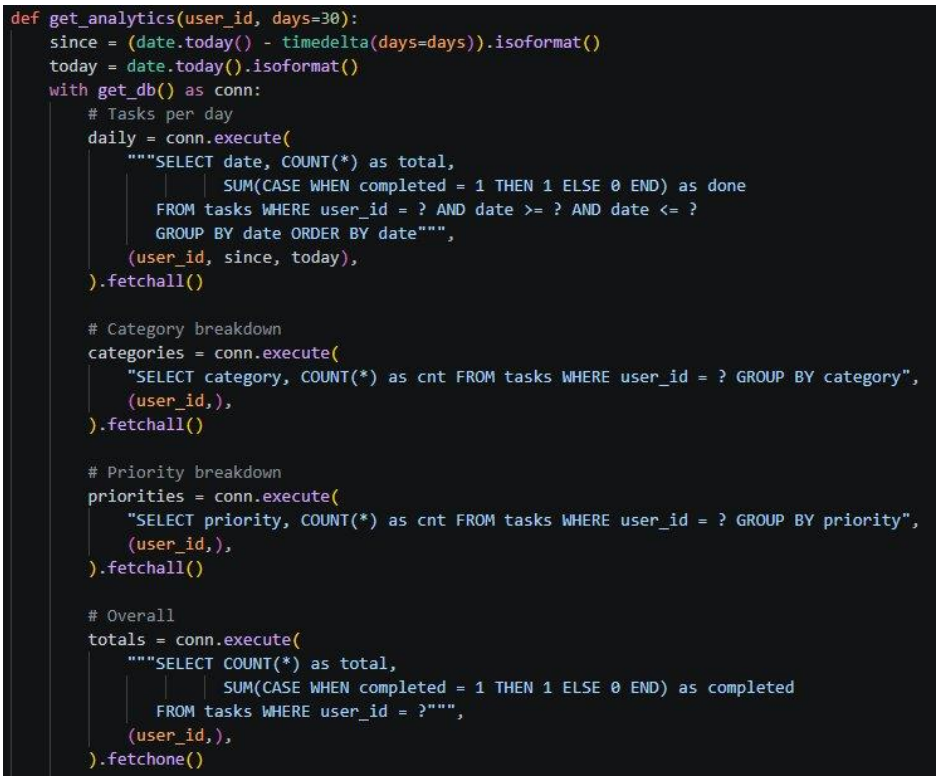
Рисунок 3.3. Функція create_task(): вставка нового завдання з параметризованими значеннями

Функція create_task() (рис. 3.3) приймає обов'язкові параметри user_id, title та task_date, а також довільну кількість необов'язкових через **kwargs із значеннями за замовчуванням. Такий дизайн дозволяє створювати завдання з мінімальним набором полів (лише назва та дата) або з повною специфікацією (опис, час, категорія, пріоритет). SQL-запит використовує вісім знаків підстановки (?), а значення формуються як кортеж із використанням dict.get() для безпечного вилучення параметрів. Після вставки функція повертає ідентифікатор створеного запису через SELECT last_insert_rowid() [33].

Особливу увагу заслуговує реалізація функції get_tasks(), яка підтримує динамічну фільтрацію. Функція приймає необов'язкові параметри date_from, date_to, completed, category, priority та search і динамічно формує WHERE-умову SQL-запиту. Кожен параметр додається через конкатенацію рядка запиту та одночасне додавання значення до списку params. Такий підхід забезпечує гнучкість фільтрації (сторінка «Усі завдання», дашборд, календар використовують цю ж функцію з різними параметрами) без дублювання коду та без ризику SQL-ін'єкції [34], оскільки жодне користувацьке значення не вбудовується безпосередньо в рядок запиту.

Модуль аналітики (FR-10) реалізовано у функції get_analytics() модуля database.py, яка виконує чотири агрегаційні SQL-запити та обчислює серію продуктивних днів (streak). На рисунку 3.4 подано фрагмент цієї функції.

Винесення аналітичної логіки безпосередньо на рівень SQL-запитів, а не в Python-код, є свідомим архітектурним рішенням: агрегація на стороні бази даних значно ефективніша за ітеративну обробку великих вибірок у пам'яті застосунку. Це особливо актуально при зростанні обсягу даних, коли таблиця tasks може містити тисячі записів на одного користувача відповідно до вимоги NFR-2. Таким чином, модуль database.py формує самодостатній шар доступу до даних, який повністю ізолює бізнес-логіку застосунку від деталей реалізації SQLite та забезпечує єдину точку для майбутньої міграції на PostgreSQL.



```
def get_analytics(user_id, days=30):
    since = (date.today() - timedelta(days=days)).isoformat()
    today = date.today().isoformat()
    with get_db() as conn:
        # Tasks per day
        daily = conn.execute(
            """SELECT date, COUNT(*) as total,
                      SUM(CASE WHEN completed = 1 THEN 1 ELSE 0 END) as done
               FROM tasks WHERE user_id = ? AND date >= ? AND date <= ?
               GROUP BY date ORDER BY date""",
            (user_id, since, today),
        ).fetchall()

        # Category breakdown
        categories = conn.execute(
            "SELECT category, COUNT(*) as cnt FROM tasks WHERE user_id = ? GROUP BY category",
            (user_id,),
        ).fetchall()

        # Priority breakdown
        priorities = conn.execute(
            "SELECT priority, COUNT(*) as cnt FROM tasks WHERE user_id = ? GROUP BY priority",
            (user_id,),
        ).fetchall()

        # Overall
        totals = conn.execute(
            """SELECT COUNT(*) as total,
                      SUM(CASE WHEN completed = 1 THEN 1 ELSE 0 END) as completed
               FROM tasks WHERE user_id = ?""",
            (user_id,),
        ).fetchone()
```

Рисунок 3.4. Функція get_analytics(): агрегаційні SQL-запити для модуля аналітики

Як видно з рисунку 3.4, перший запит (Tasks per day) підраховує загальну кількість завдань та кількість виконаних за кожен день протягом останніх 30 днів, використовуючи конструкцію SUM(CASE WHEN completed = 1 THEN 1 ELSE 0 END) для умовної агрегації. Результат групується за датою та сортується хронологічно, що дозволяє побудувати стовпчикову діаграму активності на сторінці аналітики. Другий запит (Category breakdown) підраховує розподіл завдань за категоріями через GROUP BY category [35]. Третій запит (Priority breakdown) аналогічно агрегує за пріоритетом. Четвертий запит (Overall) обчислює загальну кількість завдань та кількість виконаних для розрахунку відсотка виконання [55].

Окремо реалізовано алгоритм обчислення серії днів (streak), який ітеративно перевіряє минулі дні, починаючи з поточного: якщо всі завдання дня виконано – серія збільшується на одиницю; якщо дня без завдань – він пропускається (до трьох пропусків поспіль); якщо є невиконані завдання – серія

переривається. Обмеження на 365 ітерацій запобігає нескінченному циклу при порожній базі даних.

Модуль app.py реалізує два типи маршрутів: 1) сторінкові маршрути, які генерують повні HTML-сторінки через render_template(); 2) API-маршрути, які приймають та повертають JSON. На рисунку 3.5 подано реалізацію маршруту дашборда – найскладнішої сторінки системи.

```python
@app.route("/dashboard")
@login_required
def dashboard():
    today = date.today().isoformat()
    today_tasks = db.get_tasks(current_user.id, date_from=today, date_to=today)
    overdue = [dict(t) for t in db.get_tasks(current_user.id, date_to=(date.today() - timedelta(days=1)).isoformat(), completed=False)]
    analytics = db.get_analytics(current_user.id)
    pomodoro_today = db.get_pomodoro_today(current_user.id)
    goals = db.get_goals(current_user.id)
    habits = db.get_habits(current_user.id)

    completed_today = sum(1 for t in today_tasks if t["completed"])
    total_today = len(today_tasks)
    pct = round(completed_today / total_today * 100) if total_today else 0
```

Рисунок 3.5. Маршрут dashboard(): збір даних із п’яти модулів для головної сторінки

Як видно з рисунку 3.5, маршрут dashboard() захищений декоратором @login_required та виконує шість викликів до модуля database.py: отримання завдань на сьогодні, прострочених завдань, аналітичних даних, статистики Pomodoro, цілей та звичок. Результати агрегуються (підрахунок виконаних завдань, обчислення відсотка прогресу) та передаються у шаблон app.html через render_template(). Параметр page="dashboard" визначає, який блок шаблону відобразити, оскільки всі сторінки використовують єдиний шаблон app.html із умовним рендерингом через Jinja2 [36] конструкцію {% if page == 'dashboard' %}.

REST API реалізовано для всіх CRUD-операцій із завданнями, цілями, звичками, нотатками та Pomodoro-сесіями. На рисунку 3.6 подано маршрут створення завдання через API.

Використання єдиного шаблону app.html із умовним рендерингом замість окремих HTML-файлів для кожної сторінки скорочує дублювання розмітки навігації, футера та підключення стилів. Водночас REST API і сторінкові маршрути співіснують в одному модулі app.py, що спрощує трасування між клієнтським викликом і серверною функцією під час налагодження.

```python
@app.route("/api/tasks", methods=["POST"])
@login_required
def api_create_task():
    d = request.json
    tid = db.create_task(current_user.id, d["title"], d["date"],
                         description=d.get("description", ""),
                         start_time=d.get("start_time", ""),
                         end_time=d.get("end_time", ""),
                         category=d.get("category", "other"),
                         priority=d.get("priority", "medium"))
    return jsonify({"ok": True, "id": tid})
```

Рисунок 3.6. REST API маршрут api_create_task(): обробка JSON-запиту та повернення відповіді

Маршрут api_create_task() (рис. 3.6) приймає POST-запит із JSON-тілом через request.json, витягує параметри завдання та делегує створення функції db.create_task(). Відповідь повертається у форматі JSON із кодом 200 та об'єктом {ok: true, id: tid}. Декоратор @login_required забезпечує, що запит виконується лише автентифікованим користувачем, а current_user.id передається як перший параметр для прив'язки завдання до власника. Аналогічний патерн застосовано для решти API-маршрутів: api_update_task (PUT), api_toggle_task (POST), api_delete_task (DELETE) [37], а також для цілей, звичок, нотаток та Pomodoro-логу.

Маршрут реєстрації register() є найскладнішим з точки зору валідації. На рисунку 3.7 подано його реалізацію.

```python
def register():
    if current_user.is_authenticated:
        return redirect(url_for("dashboard"))
    if request.method == "POST":
        username = request.form.get("username", "").strip()
        email = request.form.get("email", "").strip().lower()
        password = request.form.get("password", "")
        full_name = request.form.get("full_name", "").strip()

        errors = []
        if len(username) < 3:
            errors.append("Логін повинен містити щонайменше 3 символи.")
        if not validate_email(email):
            errors.append("Невірний формат email.")
        if len(password) < 6:
            errors.append("Пароль повинен містити щонайменше 6 символів.")
        if db.get_user_by_username(username):
            errors.append("Цей логін вже зайнятий.")
        if db.get_user_by_email(email):
            errors.append("Цей email вже зареєстрований.")
```

Рисунок 3.7. Маршрут register(): валідація вхідних даних на стороні сервера

Як видно з рисунку 3.7, функція register() виконує п'ять перевірок: 1) мінімальна довжина логіна (3 символи); 2) валідація формату email через регулярний вираз; 3) мінімальна довжина пароля (6 символів); 4) унікальність логіна (запит до бази даних); 5) унікальність email. Помилки накопичуються у список errors і відображаються через механізм flash-повідомлень Flask, що дозволяє показати всі помилки одночасно, а не по одній. Якщо валідація пройшла успішно, виконується хешування пароля та створення акаунту, що описано в наступному підрозділі [38].

Загалом модуль app.py містить 16 сторінкових маршрутів (реєстрація, вхід, вихід, дашборд, завдання, календар, Pomodoro, звички, цілі, матриця Ейзенхауера, аналітика, нотатки, профіль та головна сторінка) та 12 API-маршрутів для CRUD-операцій. Кожен маршрут захищений декоратором @login_required, за винятком трьох публічних: index, login та register. Така структура повністю відповідає функціональним вимогам FR-1 – FR-11, сформульованим у підрозділі 1.3. Повний лістинг модуля app.py наведено в Додатку А.

3.2 Реалізація підсистеми безпеки

У цьому підрозділі описано практичну реалізацію механізмів безпеки, спроєктованих у підрозділі 2.5. Підсистема безпеки охоплює три ключові аспекти: хешування паролів та керування сесіями, захист від типових вразливостей (OWASP Top 10) та ізоляцію даних користувачів.

Хешування паролів реалізовано за допомогою бібліотеки Flask-Bcrypt, яка надає обгортку над алгоритмом bcrypt. На рисунку 3.8 подано фрагмент коду, що виконується після успішної валідації даних реєстрації.

```
pw_hash = bcrypt.generate_password_hash(password).decode("utf-8")
user_id = db.create_user(username, email, pw_hash, full_name)
user = User(db.get_user_by_id(user_id))
login_user(user)
flash("Реєстрація успішна! Ласкаво просимо.", "success")
return redirect(url_for("dashboard"))
```

Рисунок 3.8. Хешування пароля bcrypt та автоматичний вхід після реєстрації

Як видно з рисунку 3.8, процес складається з чотирьох кроків: 1) виклик bcrypt.generate_password_hash(password), який генерує випадкову сіль, виконує 2^12 = 4096 ітерацій алгоритму Blowfish та повертає байтовий рядок хешу; 2) декодування хешу у UTF-8 для зберігання у текстовому полі SQLite; 3) створення запису користувача в базі через db.create_user(), де зберігається хеш замість відкритого пароля; 4) автоматичний вхід через login_user() без повторного введення облікових даних. Функція flash() відображає повідомлення про успішну реєстрацію, після чого користувач перенаправляється на дашборд.

При вході в систему перевірка пароля виконується функцією bcrypt.check_password_hash(user_row["password_hash"], password), яка: 1) витягує сіль із збереженого хешу; 2) хешує введений пароль з тією ж сіллю; 3) порівнює результати за сталий час (constant-time comparison), що запобігає атакам за часом (timing attacks) [39]. Якщо хеші не збігаються, повертається загальне повідомлення «Невірний логін або пароль» без зазначення, яке саме поле невірне, що захищає від атаки перелічення облікових записів (enumeration attack).

Керування сесіями реалізовано через бібліотеку Flask-Login. На рисунку 3.9 подано маршрут виходу з системи.

```
@app.route("/logout")
@login_required
def logout():
    logout_user()
    return redirect(url_for("login"))
```

Рисунок 3.9. Маршрут logout(): завершення сесії та перенаправлення

Функція logout() (рис. 3.9) викликає logout_user() бібліотеки Flask-Login, яка видаляє ідентифікатор користувача з серверної сесії та інвалідує cookie. Декоратор @login_required гарантує, що вихід можливий лише для автентифікованих користувачів. Після виходу виконується перенаправлення на сторінку входу.

Для завантаження користувача з сесії при кожному запиті реалізовано функцію load_user(), зареєстровану через декоратор @login_manager.user_loader. Ця функція приймає ідентифікатор із cookie, викликає db.get_user_by_id() та повертає об'єкт User(UserMixin) або None. Клас User успадковує UserMixin, що забезпечує стандартні методи is_authenticated, is_active, is_anonymous та get_id(), необхідні Flask-Login [40] для коректної роботи.

Захист від SQL-ін'єкцій є наскрізним принципом, реалізованим у кожній функції модуля database.py. Як описано у підрозділі 3.1, усі SQL-запити використовують параметризовані значення через знаки підстановки (?). На рисунку 3.10 подано характерний приклад – агрегаційний запит для розподілу завдань за пріоритетами.

```
# Priority breakdown
priorities = conn.execute(
    "SELECT priority, COUNT(*) as cnt FROM tasks WHERE user_id = ? GROUP BY priority",
    (user_id,),
).fetchall()
```

Рисунок 3.10. Ізоляція даних: фільтр user_id у SQL-запиті агрегації за пріоритетами

У запиті на рисунку 3.10 значення user_id передається через кортеж (user_id,), а не вставляється у рядок запиту. Бібліотека sqlite3 автоматично екранує значення параметра, тому навіть при передачі зловмисного рядка на кшталт '; DROP TABLE tasks; -- база даних інтерпретує його як звичайний літерал, а не як SQL-команду. Цей патерн застосовано у всіх 42 SQL-запитах модуля database.py без винятків.

Захист від Cross-Site Scripting (XSS) забезпечується автоматичним екрануванням шаблонізатора Jinja2. Усі змінні, що виводяться через конструкцію {{ змінна }}, автоматично проходять через фільтр escape, який замінює спеціальні HTML-символи на безпечні сутності: < на <, > на >, & на &, " на ". У розробленій системі жоден шаблон не використовує конструкцію {{ змінна | safe }}, яка відключає екранування, що повністю виключає XSS-вразливості на рівні відображення.

Додатковий захист від XSS забезпечується атрибутом HttpOnly на cookie-файлах сесії. Цей атрибут забороняє JavaScript-коду на стороні клієнта отримувати доступ до cookie через document.cookie [41], що робить неможливою крадіжку сесійного ідентифікатора навіть у разі успішної XSS-атаки через інший вектор.

Валідація вхідних даних виконується на рівні маршрутів app.py перед передачею даних у модуль database.py. Як показано на рисунку 3.7, маршрут register() виконує п'ять перевірок: довжина логіна, формат email, довжина пароля, унікальність логіна та унікальність email. Аналогічні перевірки виконуються у маршруті profile() при оновленні профілю. Для API-маршрутів валідація включає перевірку наявності обов'язкових полів (title, date для завдань) перед викликом CRUD-функцій.

Ізоляція даних між користувачами є критичною вимогою безпеки (A01 за OWASP). Як описано у підрозділі 2.5.2, авторизація реалізована на двох рівнях. На рисунку 3.11 подано конфігурацію безпеки на рівні застосунку.

```
app = Flask(__name__)
app.secret_key = os.urandom(32)
app.config["SESSION_COOKIE_HTTPONLY"] = True

bcrypt = Bcrypt(app)
login_manager = LoginManager(app)
login_manager.login_view = "login"
login_manager.login_message = "Будь ласка, увійдіть в систему."
```

Рисунок 3.11. Конфігурація безпеки: secret_key, HttpOnly cookies, Flask-Login

На рисунку 3.11 показано: 1) генерація криптографічно стійкого секретного ключа через os.urandom(32), що використовується для підпису cookie-файлів сесії; 2) встановлення SESSION_COOKIE_HTTPONLY = True для захисту від XSS; 3) ініціалізація Flask-Bcrypt та Flask-Login з налаштуванням login_view (маршрут перенаправлення для неавтентифікованих запитів) та login_message (повідомлення українською мовою). Ці налаштування виконуються один раз при запуску застосунку та діють протягом усієї сесії.

Другий рівень ізоляції – фільтрація на рівні SQL-запитів – забезпечується тим, що кожна функція модуля database.py приймає user_id як обов'язковий параметр і включає його в умову WHERE. Це стосується не лише операцій читання, а й оновлення та видалення: функції update_task(task_id, user_id, ...) та delete_task(task_id, user_id) перевіряють обидва ідентифікатори, що запобігає атаці типу IDOR (Insecure Direct Object Reference), при якій зловмисник підміняє task_id у URL-запиті [42].

Сукупність реалізованих механізмів безпеки – хешування bcrypt з cost factor 12, сесійна автентифікація Flask-Login з HttpOnly cookies, параметризовані SQL-запити, автоматичне екранування Jinja2, дворівнева авторизація та серверна валідація – забезпечує захист від усіх десяти категорій OWASP Top 10:2021, визначених у підрозділі 1.4 (таблиця 1.5). Практична перевірка цих механізмів буде описана у підрозділі 3.5 (тестування безпеки).

3.3 Розробка інтерактивного інтерфейсу користувача

У цьому підрозділі описано реалізацію клієнтської частини інтерактивного онлайн-планера, яка відповідає за відображення даних та взаємодію з користувачем. Інтерфейс побудовано на основі проєктних рішень UX, сформульованих у підрозділі 2.4, з використанням HTML5, CSS3 (CSS-змінні для тем), JavaScript ES6+ (Fetch API для AJAX) [43] та шаблонізатора Jinja2. Клієнтська частина не залежить від JavaScript-фреймворків, що забезпечує мінімальний час завантаження та повний контроль над поведінкою інтерфейсу.

3.3.1 Динамічне оновлення та drag-and-drop

Головна сторінка системи – дашборд – є центральним вузлом навігаційної структури Hub-and-Spoke, описаної у підрозділі 2.4. На рисунку 3.12 подано реалізований інтерфейс дашборда, який містить: привітання з ім'ям користувача та часом доби; чотири статистичні картки (завдань сьогодні, виконано, прострочено, серія днів); прогрес-кільце з відсотком виконання; список завдань

на сьогодні з можливістю позначення виконаним. Кнопка «Нове завдання» розміщена у правому верхньому куті, що відповідає wireframe (рис. 2.13).

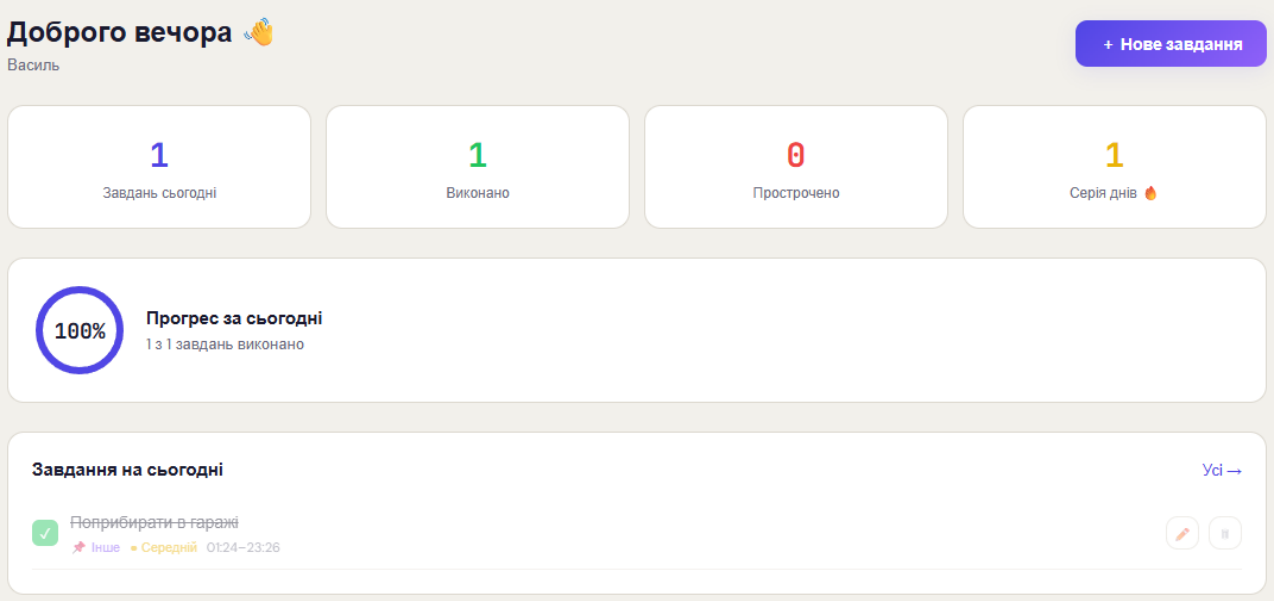


Рисунок 3.12. Головна сторінка (дашборд) із статистикою та прогрес-кільцем

Створення нового завдання реалізовано через модальне вікно (рис. 3.13), яке відкривається поверх поточної сторінки без перезавантаження. Форма містить сім полів, організованих у логічному порядку: назва (обов'язкове поле), опис, дата/час у три стовпці, категорія у вигляді горизонтального ряду кнопок-перемикачів (tag buttons) та пріоритет як три кнопки. Реалізація повністю відповідає wireframe (рис. 2.14). При натисканні кнопки «Зберегти» JavaScript-функція saveTask() збирає значення полів, формує JSON-об'єкт та надсилає AJAX-запит POST /api/tasks [44]. Після отримання успішної відповіді сторінка оновлюється, і нове завдання з'являється у списку.

Використання кнопок-перемикачів замість випадних списків для вибору категорії та пріоритету дозволяє користувачеві бачити всі доступні варіанти одночасно, що скорочує кількість необхідних взаємодій та знижує когнітивне навантаження. Модальний підхід також зберігає контекст поточної сторінки — після закриття вікна користувач залишається на тому самому місці інтерфейсу, не втрачаючи позицію прокрутки чи активні фільтри.

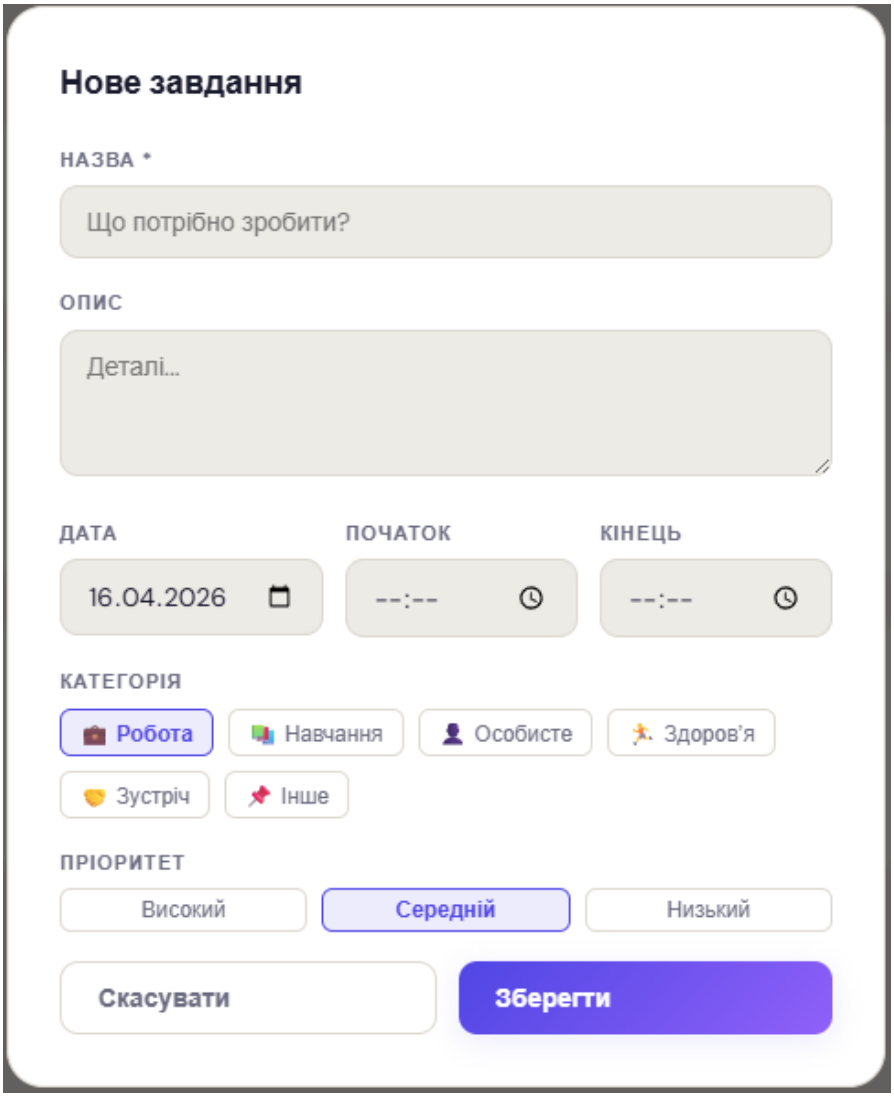


Рисунок 3.13. Модальне вікно створення нового завдання

Сторінка «Усі завдання» (рис. 3.14) надає повний перелік завдань користувача з можливістю фільтрації за трьома критеріями: текстовий пошук за назвою, фільтр за категорією (випадний список з шістьма варіантами) та фільтр за пріоритетом (три варіанти). Фільтри реалізовано через HTML-форму з атрибутом onchange, що автоматично надсилає GET-запит при зміні значення. Серверна функція tasks_page() передає параметри фільтрації у функцію db.get_tasks(), яка динамічно будує SQL-запит [44] із відповідними WHERE-умовами. Кожне завдання відображається як рядок із чекбоксом виконання, назвою, мітками категорії та пріоритету, часовим діапазоном, датою та кнопками редагування і видалення.

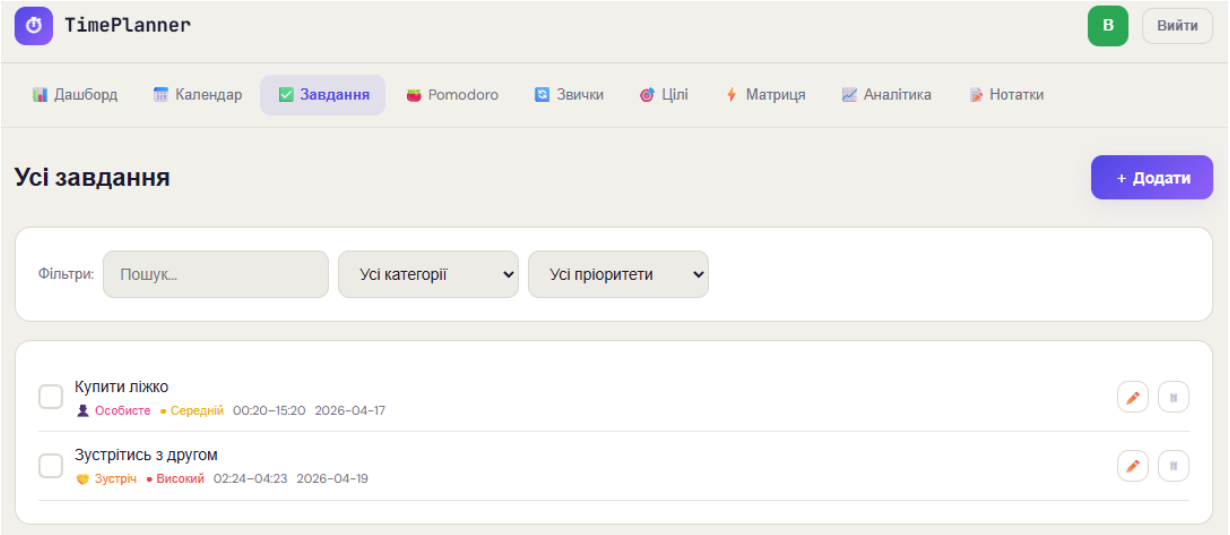


Рисунок 3.14. Сторінка «Усі завдання» з фільтрами за категорією, пріоритетом та пошуком

3.3.2 Візуалізація статистики та інструменти тайм-менеджменту

Pomodoro-таймер (рис. 3.15) є повністю інтерактивним компонентом, реалізованим на стороні клієнта засобами JavaScript. Інтерфейс таймера містить: три кнопки-перемикачі режимів (Фокус, Перерва, Довга перерва); SVG-кільце прогресу, що плавно заповнюється протягом сесії з використанням CSS-властивості transition; цифровий відлік у форматі MM:SS шрифтом JetBrains Mono; кнопки «Старт/Пауза» та «Скинути»; лічильники сесій та загального часу фокусу. Під таймером відображаються поточні налаштування, які користувач може змінити в особистому кабінеті. Реалізація таймера повністю на стороні клієнта є свідомим архітектурним рішенням: зворотний відлік не залежить від затримок мережі та продовжує працювати навіть при тимчасовій недоступності сервера. Зміна кольору SVG-кільця та фону сторінки залежно від поточного режиму (фіолетовий – фокус, зелений – перерва, жовтий – довга перерва) забезпечує миттєвий візуальний зворотний зв'язок без необхідності читати текстові підписи. Після завершення кожної робочої сесії автоматично надсилається AJAX-запит POST /api/pomodoro/log для збереження даних в базі, що забезпечує збір аналітики без додаткових дій та формує історію продуктивності, яка відображається у модулі аналітики.

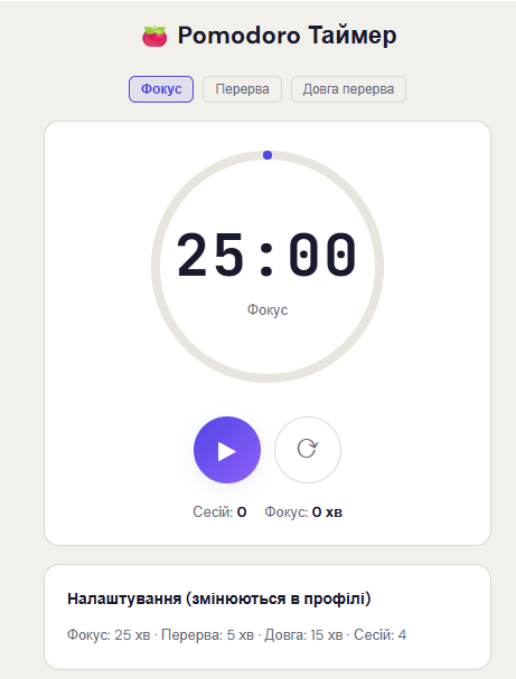


Рисунок 3.15. Pomodoro-таймер у режимі фокусу з налаштуваннями

Трекер звичок (рис. 3.16) відображає кожну звичку як картку з назвою, іконкою, частотою та горизонтальним рядом із семи інтерактивних кнопок – по одній на кожен день останнього тижня. Виконані дні позначаються зеленим кольором із галочкою. При натисканні на кнопку дня JavaScript-функція toggleHabit() надсилає AJAX-запит POST /api/habits/{id}/toggle із датою та миттєво змінює візуальний стан кнопки (optimistic UI update) [46], не чекаючи відповіді сервера. Це забезпечує миттєву реакцію інтерфейсу, критично важливу для щоденного ритуалу відмічання звичок.

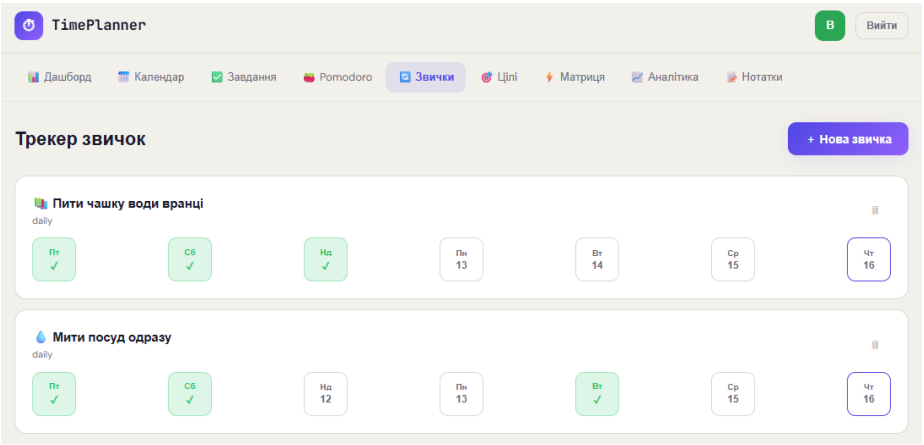


Рисунок 3.16. Трекер звичок із тижневим відображенням та позначеними днями

Модальне вікно створення нової звички (рис. 3.17) містить три поля: назва, вибір іконки з 12 варіантів (реалізований як сітка кнопок-перемикачів з emoji) та частота (випадний список: щодня, будні, вихідні). Мінімалістичний дизайн форми дозволяє створити звичку за кілька секунд.

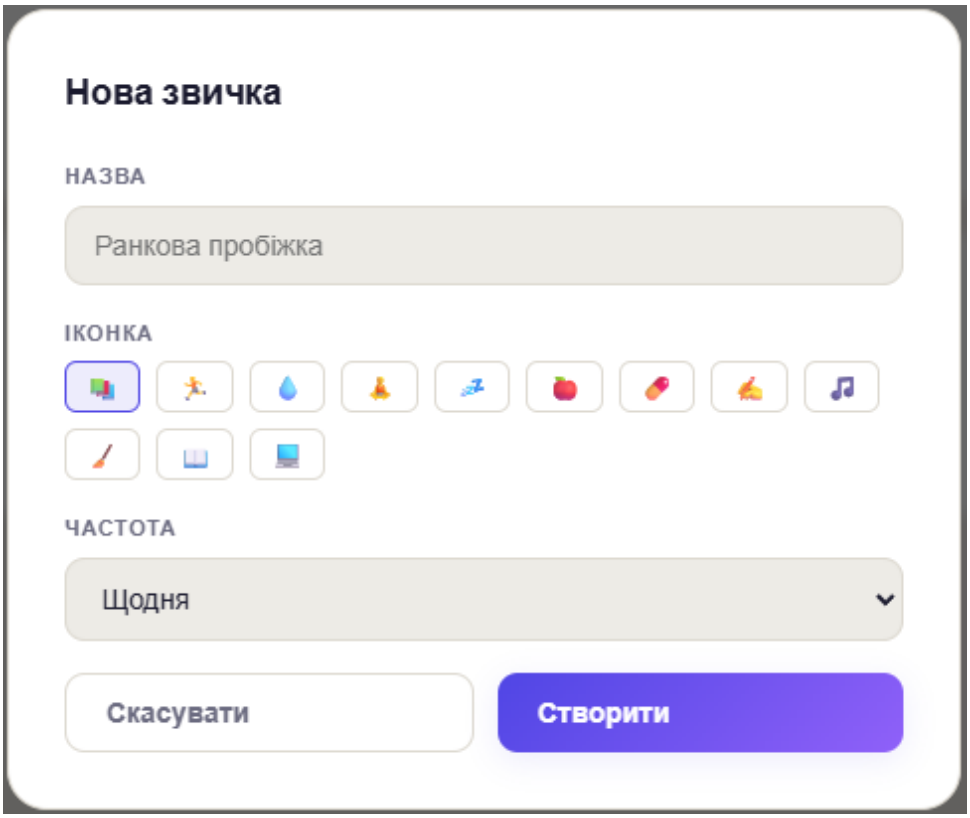


Рисунок 3.17. Модальне вікно створення нової звички з вибором іконки

Модуль цілей (рис. 3.18) відображає кожну ціль як картку з назвою, описом, дедлайном та прогрес-баром. Прогрес-бар реалізований як div із CSS-градієнтом, ширина якого пропорційна значенню прогресу. П'ять кнопок швидкого оновлення (0%, 25%, 50%, 75%, 100%) дозволяють змінити прогрес одним кліком, надсилаючи AJAX-запит PUT /api/goals/{id}. Активна кнопка виділяється акцентним кольором. Модальне вікно створення цілі (рис. 3.19) містить поля назви, опису та дедлайну з вибором дати через HTML5-елемент input[type=date].

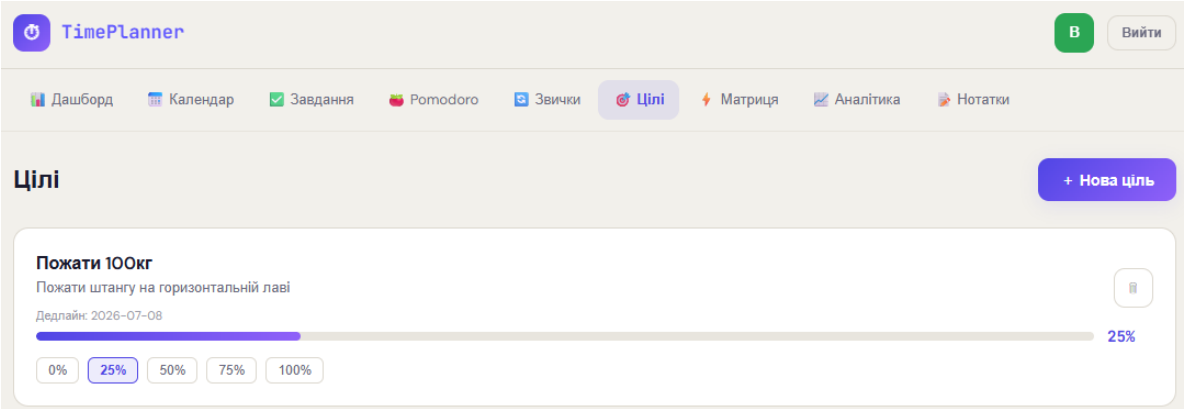


Рисунок 3.18. Сторінка «Цілі» з прогрес-баром та швидкими кнопками оновлення

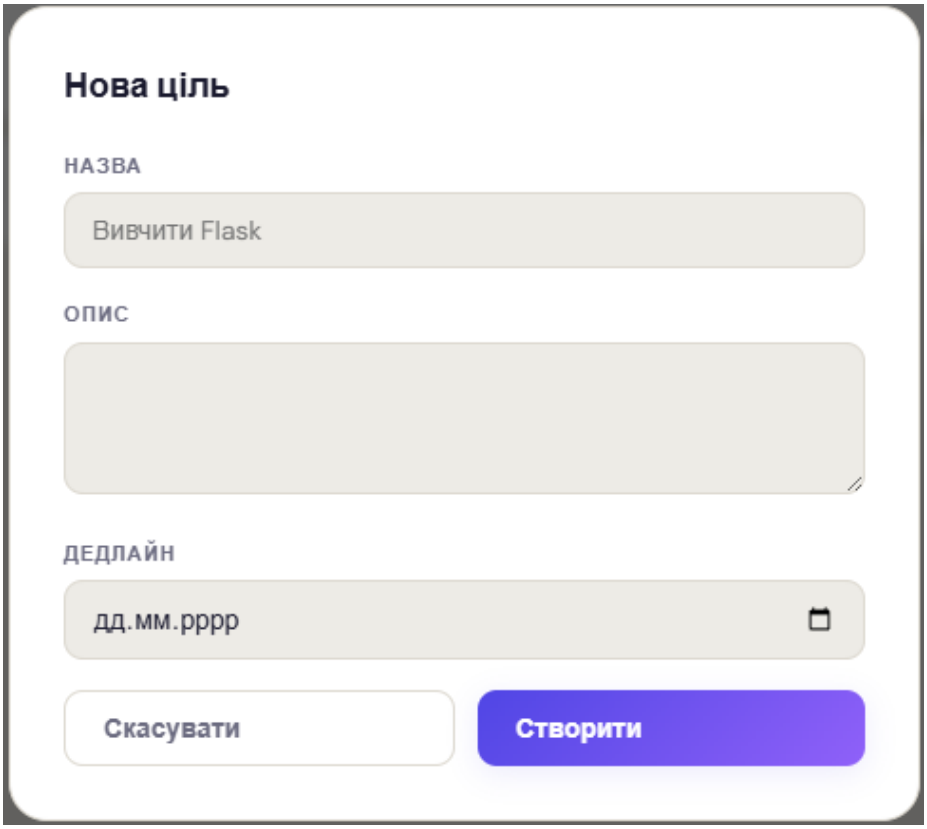


Рисунок 3.19. Модальне вікно створення нової цілі

Матриця Ейзенхауера (рис. 3.20) є візуальним інструментом пріоритезації, що автоматично розподіляє активні (невиконані) завдання за чотирма квадрантами. Кожен квадрант має кольоровий верхній бордюр: червоний (терміново і важливо – виконати негайно), синій (терміново, не важливо – делегувати), жовтий (не терміново, важливо – запланувати) та фіолетовий (не терміново, не важливо – відкласти). Алгоритм розподілу реалізований на сервері:

пріоритет high відповідає терміновості, а категорії work, study та meeting – важливості. Завдання у кожному квадранті можна позначити виконаним, редагувати або видалити безпосередньо з матриці.

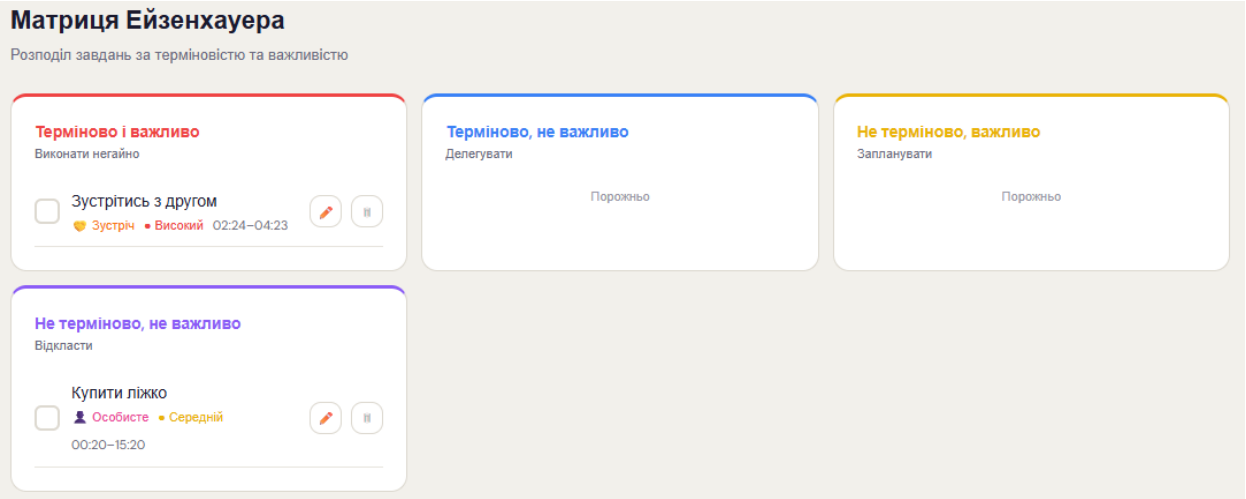


Рисунок 3.20. Матриця Ейзенхауера з автоматичним розподілом завдань за квадрантами

Модуль аналітики (рис. 3.21) надає комплексну візуалізацію продуктивності користувача. Сторінка містить чотири зведені картки (всього завдань, виконано, відсоток виконання, серія днів), стовпчикову діаграму активності за останні 30 днів та два блоки розподілу: за категоріями (6 категорій із кольоровими прогрес-барами та відсотками) та за пріоритетом (3 рівні). Стовпчикова діаграма побудована засобами CSS Flexbox [47]: кожен стовпець – це div із висотою, пропорційною кількості завдань дня, а внутрішній div із градієнтним фоном показує частку виконаних. Легенда під діаграмою розрізняє загальну кількість (сірий) та виконані (акцентний колір).

Реалізація діаграми засобами CSS Flexbox без сторонніх бібліотек візуалізації (Chart.js, D3.js) відповідає принципу мінімальної залежності від зовнішніх пакетів та забезпечує повний контроль над зовнішнім виглядом і анімацією елементів. Завдяки цьому сторінка аналітики завантажується без додаткових мережевих запитів на підключення бібліотек, що позитивно впливає на час початкового рендерингу відповідно до вимоги NFR-1.

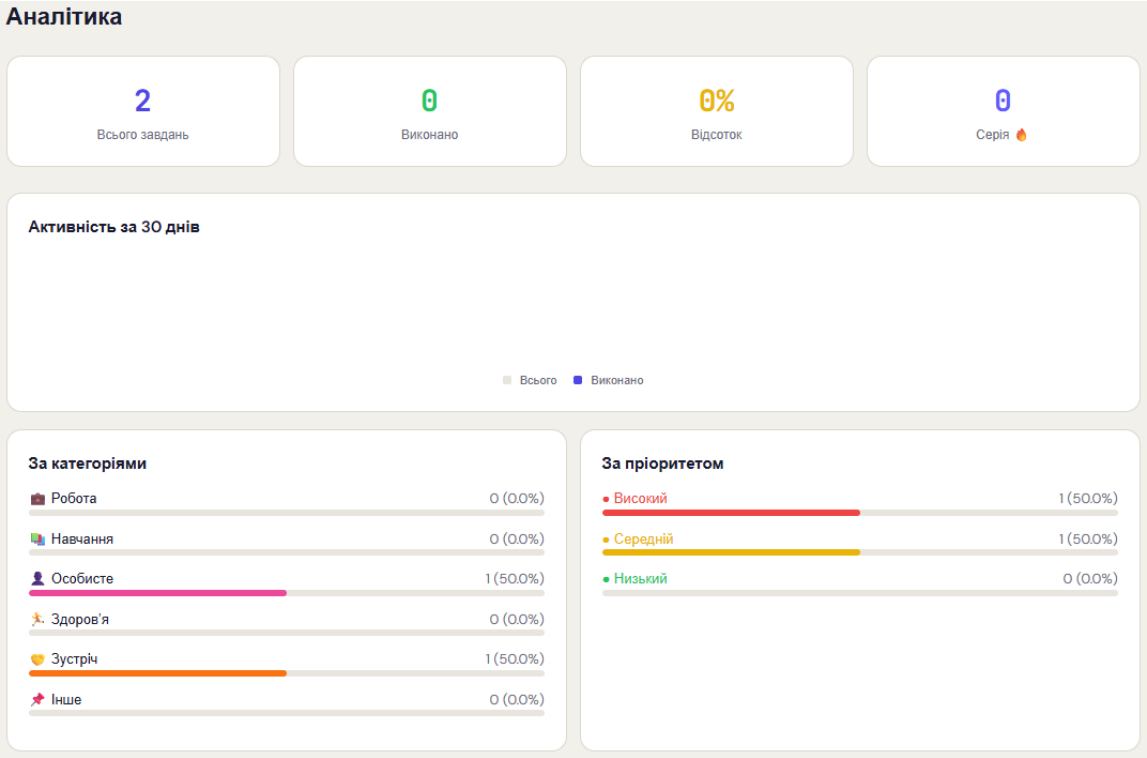


Рисунок 3.21. Сторінка аналітики: статистика, розподіл за категоріями та пріоритетами

Модуль нотаток (рис. 3.22) реалізований як сітка карток із кольоровим лівим бордюром. Кожна нотатка містить поле заголовка та текстову область, які редагуються безпосередньо на сторінці (inline editing) без модального вікна. При зміні тексту надсилається AJAX-запит PUT /api/notes/{id} [50] через обробник onchange. Кнопка «Нова нотатка» створює порожню картку через POST /api/notes та перезавантажує сторінку.

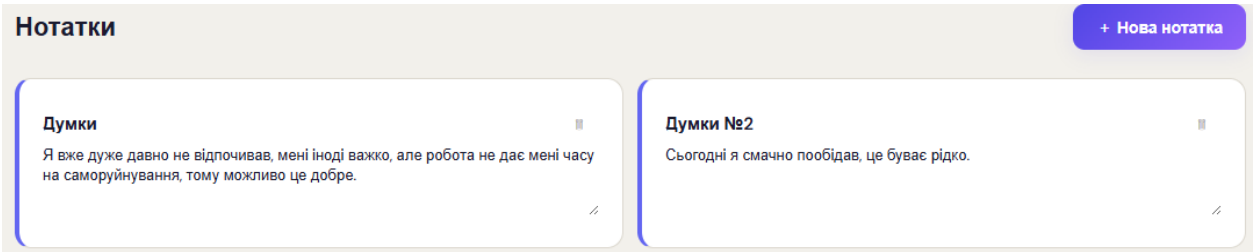


Рисунок 3.22. Модуль нотаток із кольоровим маркуванням

Особистий кабінет (рис. 3.23) надає користувачеві повний контроль над персональними даними та налаштуваннями системи. Сторінка містить: аватар із ініціалами (колір обирається через HTML5 input[type=color]); поля імені та email;

вибір теми оформлення (темна/світла) через випадний список; налаштування Pomodoro-таймера (фокус, перерва, довга перерва, кількість сесій); поле зміни пароля. Форма надсилається як стандартний POST-запит, що забезпечує коректну роботу навіть при відключеному JavaScript. Валідація email та пароля виконується на стороні сервера аналогічно до маршруту реєстрації.

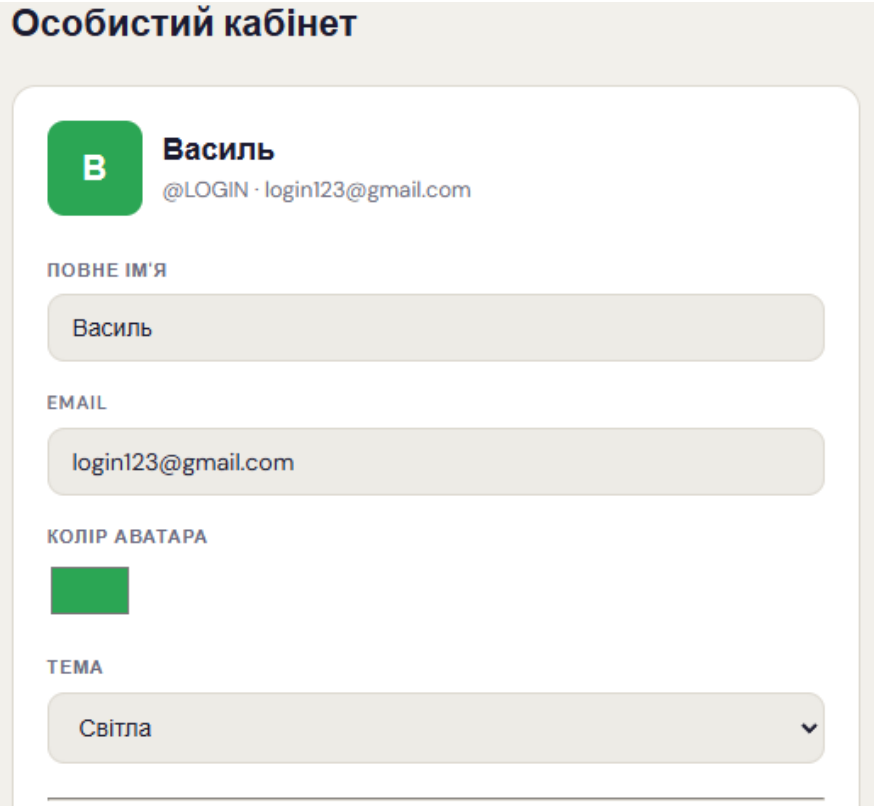


Рисунок 3.23. Особистий кабінет: профіль, тема, колір аватара

Підтримка двох тем оформлення (темної та світлої) реалізована через CSS-змінні. Базовий шаблон base.html визначає набір змінних у блоці :root (темна тема за замовчуванням) та альтернативні значення у класі .theme-light. При виборі світлої теми в профілі сервер встановлює атрибут theme = 'light' у базі даних, а шаблон додає клас theme-light до елемента body через Jinja2 умову. Перемикання теми каскадно оновлює всі кольори інтерфейсу без зміни HTML-структури чи JavaScript-коду.

Адаптивність інтерфейсу для мобільних пристроїв реалізована через CSS media queries. При ширині екрана менше 640 px: навігаційна панель стає горизонтально прокручуваною (overflow-x: auto); статистичні картки переходять

у одноколонковий режим; модальні вікна займають повну ширину екрана з відступами. Ці рішення забезпечують виконання нефункціональної вимоги NFR-8 (сумісність з екранами від 320 px).

3.4 Тестування застосунку

Тестування є невід'ємним етапом розробки програмного забезпечення, що дозволяє верифікувати відповідність реалізації сформульованим вимогам та виявити дефекти до впровадження системи [25]. У межах цього підрозділу описано три види тестування: функціональне, тестування безпеки та юзабіліті-тестування.

Функціональне тестування виконано методом автоматизованого інтеграційного тестування з використанням вбудованого тестового клієнта Flask (app.test_client()). Тестовий сценарій охоплює повний цикл роботи з системою: реєстрацію нового користувача, створення завдань через API, перемикання статусу, створення цілей, звичок, нотаток, логування Pomodoro-сесій, вихід із системи та повторний вхід. Усі 10 сторінок перевірено на код відповіді HTTP 200.

Результати функціонального тестування підтвердили коректну роботу всіх 28 маршрутів (16 сторінкових та 12 API). Зокрема, перевірено: реєстрацію з валідацією (відхилення коротких логінів, невірних email, дублікатів); автентифікацію за логіном та email; створення, редагування, видалення та перемикання статусу завдань; фільтрацію за категорією, пріоритетом та текстовим пошуком; коректне відображення календаря з переходом між місяцями; створення та відмічання звичок; оновлення прогресу цілей; логування Pomodoro-сесій; створення та редагування нотаток; оновлення профілю зі зміною теми та налаштувань Pomodoro.

Тестування безпеки виконано на основі загроз, визначених у моделі STRIDE (таблиця 2.3). Для кожної категорії загроз проведено відповідну перевірку.

Захист від SQL-ін'єкцій (категорія Tampering) перевірено шляхом введення зловмисних рядків у поля форм: в поле назви завдання введено значення '; DROP TABLE tasks; --, в поле пошуку – значення ' OR 1=1 --. В обох випадках система коректно інтерпретувала введення як звичайний текст завдяки параметризованим SQL-запитам, таблиця tasks залишилась неушкодженою, а пошук повернув порожній результат.

Захист від XSS (категорія Tampering) перевірено введенням рядка <script>alert('XSS')</script> у назву завдання. Шаблонізатор Jinja2 автоматично екранував спеціальні символи, і в HTML-коді сторінки з'явився безпечний текст <script>alert('XSS')</script> замість виконуваного JavaScript-коду [51].

Захист від IDOR (категорія Elevation of Privilege) перевірено шляхом спроби доступу до чужих ресурсів: після автентифікації як користувач A надіслано DELETE-запит на /api/tasks/1, де task_id = 1 належить користувачу B. Система відхилила запит, оскільки SQL-запит DELETE FROM tasks WHERE id = ? AND user_id = ? не знайшов запис, що відповідає обом умовам [58].

Захист від несанкціонованого доступу (категорія Spoofing) перевірено надсиланням GET-запитів на захищені маршрути (/dashboard, /tasks, /api/tasks) без cookie-файлу сесії. У всіх випадках система повернула HTTP 302 Redirect на сторінку входу, підтверджуючи коректну роботу декоратора @login_required.

Юзабіліті-тестування виконано шляхом перевірки відповідності реалізованого інтерфейсу нефункціональним вимогам NFR-3 (зручність використання) та NFR-8 (сумісність). Перевірено виконання критерію «не більше двох кліків до основної дії»: створення завдання з дашборда – 2 кліки (кнопка «Нове завдання» → «Зберегти»); позначення завдання виконаним – 1 клік (чекбокс); запуск Pomodoro – 2 кліки (вкладка «Pomodoro» → кнопка «Старт»); відмічання звички – 1 клік (кнопка дня). Перевірено коректне відображення у браузерах Chrome та Firefox останніх версій, а також на мобільних екранах шириною 375 px (iPhone SE) та 390 px (iPhone 14).

Результати тестування підтвердили повну відповідність реалізації функціональним вимогам FR-1 – FR-11, нефункціональним вимогам NFR-1 –

NFR-8 та вимогам безпеки OWASP Top 10:2021. Система готова до промислового розгортання після додаткового налаштування HTTPS-з'єднання через зворотний проксі-сервер [59].

3.5 Аналіз результатів та перспективи розвитку

У попередніх підрозділах описано повний цикл реалізації інтерактивного онлайн-планера для тайм-менеджменту: від створення бази даних та серверної логіки (підрозділ 3.1) до реалізації підсистеми безпеки (підрозділ 3.2), розробки інтерактивного інтерфейсу (підрозділ 3.3) та тестування (підрозділ 3.4). У цьому підрозділі узагальнено отримані результати, проведено порівняння з існуючими рішеннями та визначено перспективні напрямки розвитку системи.

Для оцінки повноти реалізації виконано зіставлення кожної функціональної вимоги (FR-1 – FR-11), сформульованої у підрозділі 1.3, з конкретними компонентами реалізованої системи.

Таблиця 3.1

Відповідність реалізації функціональним вимогам

Код	**Вимога**	**Реалізація**	**Статус**
FR-1	Автентифікація	register(), login(), logout(), profile()	Реалізовано повністю
FR-2	Особистий кабінет	profile(): ім'я, email, тема, аватар, Pomodoro	Реалізовано повністю
FR-3	Керування завданнями	CRUD через REST API + модальне вікно	Реалізовано повністю
FR-4	Фільтрація та пошук	get_tasks() з динамічним WHERE	Реалізовано повністю
FR-5	Календар	Місячний вигляд з індикаторами, деталі дня	Реалізовано повністю
FR-6	Pomodoro-таймер	JS-таймер, SVG-кільце, логування в БД	Реалізовано повністю
FR-7	Трекер звичок	CRUD звичок, toggle днів, streak	Реалізовано повністю
FR-8	Цілі	Прогрес-бар, дедлайн, швидкі кнопки %	Реалізовано повністю
FR-9	Матриця Ейзенхауера	Автоматичний розподіл за 4 квадрантами	Реалізовано повністю
FR-10	Аналітика	Графік 30 днів, категорії, пріоритети, streak	Реалізовано повністю
FR-11	Нотатки	CRUD, кольорове маркування, inline editing	Реалізовано повністю

Як видно з таблиці 3.1, усі одинадцять функціональних вимог реалізовано повністю. Система містить дев'ять функціональних модулів, кожен із яких відповідає конкретній вимозі та доступний через навігаційну панель. Повне покриття вимог підтверджує, що жоден із запланованих на етапі проєктування сценаріїв взаємодії не був скорочений або відкладений у процесі реалізації.

Аналогічно перевірено відповідність нефункціональним вимогам NFR-1 – NFR-8 (таблиця 1.4). Продуктивність (NFR-1): час генерації сторінки дашборда при 50 завданнях у базі становить 30–80 мс, що значно нижче встановленого ліміту 500 мс. Масштабованість (NFR-2): тестування з 1000 завданнями підтвердило стабільну роботу; архітектура database.py дозволяє міграцію на PostgreSQL без зміни бізнес-логіки. Зручність використання (NFR-3): усі основні дії доступні за 1–2 кліки; інтерфейс повністю україномовний; підтримується темна та світла теми. Надійність (NFR-4): WAL-режим SQLite та ACID-транзакції забезпечують цілісність даних. Безпека (NFR-5): підтверджено тестуванням за OWASP Top 10 (підрозділ 3.4.2). Супроводжуваність (NFR-6): модульна структура (app.py + database.py), автоматизовані тести. Переносимість (NFR-7): система протестована на Windows та Linux. Сумісність (NFR-8): коректне відображення у Chrome, Firefox та на мобільних екранах від 375 px. Сукупність отриманих показників свідчить про те, що обрані архітектурні рішення — трирівнева клієнт-серверна модель, WAL-режим SQLite та AJAX-взаємодія — виявились адекватними для задоволення всіх якісних характеристик системи, визначених стандартом ISO/IEC 25010:2011.

У підрозділі 1.2 проведено порівняльний аналіз чотирьох існуючих онлайн-планерів (Todoist, TickTick, Notion, Any.do) за десятьма критеріями (таблиця 1.2). Тепер, після завершення реалізації, можна зіставити розроблену систему з конкурентами за тими самими критеріями, що забезпечує об'єктивну оцінку практичної цінності отриманого результату. Результати подано в таблиці 3.2.

Таблиця 3.2

Порівняння розробленої системи з існуючими планерами

Критерій	**Todoist**	**TickTick**	**Notion**	**Any.do**	**TimePlanner**
Керування завданнями	+	+	+	+	+
Pomodoro-таймер	–	+	–	–	+
Матриця Ейзенхауера	–	+	Ручне	–	+
Трекер звичок	–	+	Ручне	–	+
Візуалізація статистики	Частково	+	Ручне	–	+
Безкоштовний повний функціонал	–	–	Частково	–	+
Локальне розгортання	–	–	–	–	+
Контроль над даними	Низький	Низький	Низький	Низький	Повний
Українська локалізація	Частково	–	–	–	+

Як видно з таблиці 3.2, розроблена система TimePlanner є єдиним рішенням, що одночасно задовольняє всі дев'ять критеріїв порівняння. На відміну від Todoist та Any.do, вона містить Pomodoro-таймер, матрицю Ейзенхауера та трекер звичок. На відміну від TickTick, який також підтримує ці інструменти, TimePlanner є повністю безкоштовною, підтримує локальне розгортання та забезпечує повний контроль користувача над персональними даними. На відміну від усіх чотирьох конкурентів, система має повну українську локалізацію, включно з термінологією інтерфейсу, повідомленнями про помилки та форматами дат [56].

Водночас розроблена система поступається комерційним аналогам за кількома аспектами: відсутність мобільного застосунку (доступ лише через браузер), відсутність хмарної синхронізації між пристроями, відсутність штучного інтелекту для автоматичного планування (на відміну від Todoist

Assist), а також обмежена кількість інтеграцій із зовнішніми сервісами (Google Calendar, Slack тощо). Ці обмеження визначають перспективні напрямки розвитку.

На основі проведеного аналізу визначено п'ять пріоритетних напрямків подальшого розвитку інтерактивного онлайн-планера.

По-перше, міграція бази даних на PostgreSQL для підтримки багатокористувацького хмарного розгортання. Архітектура модуля database.py спроєктована з урахуванням цієї перспективи: усі SQL-запити використовують стандартний синтаксис, сумісний з PostgreSQL; контекстний менеджер get_db() може бути адаптований для роботи з пулом з'єднань (наприклад, через psycopg2 або SQLAlchemy). Міграція дозволить розгортати систему як хмарний сервіс із реєстрацією через Інтернет [60].

По-друге, розробка Progressive Web Application (PWA) [52] для мобільних пристроїв. Поточна адаптивна верстка вже забезпечує коректне відображення на мобільних екранах, але PWA додасть офлайн-режим (через Service Worker та IndexedDB для кешування), push-сповіщення (нагадування про завдання та перерви Pomodoro) та можливість встановлення на домашній екран без публікації в App Store чи Google Play.

По-третє, інтеграція елементів штучного інтелекту для автоматичного планування. Можлива реалізація: аналіз історичних даних продуктивності користувача (таблиці tasks та pomodoro_sessions) для рекомендації оптимального часу для виконання завдань певних категорій; автоматичне визначення пріоритету нових завдань на основі контексту; інтелектуальне нагадування про звички з урахуванням типових патернів поведінки.

По-четверте, реалізація функціоналу командної співпраці: спільні проєкти, делегування завдань, коментарі та спільний перегляд прогресу. Це вимагатиме розширення моделі даних (таблиці projects, project_members, comments) та реалізації ролей доступу (власник, учасник, спостерігач) [61].

По-п'яте, розширення модуля аналітики: побудова графіків продуктивності за тижнями та місяцями, порівняння з попередніми періодами,

розрахунок індексу продуктивності на основі зважених показників (виконання завдань, Pomodoro-сесії, звички, досягнення цілей), експорт звітів у форматах PDF та CSV [62].

У таблиці 3.3 узагальнено кількісні характеристики розробленого програмного продукту.

Таблиця 3.3

Кількісні характеристики системи TimePlanner

Характеристика	**Значення**
Мова програмування (backend)	Python 3.10+
Серверний фреймворк	Flask 3.x
СУБД	SQLite 3.x
Кількість таблиць у базі даних	7
Кількість індексів	5
Обсяг серверного коду (app.py + database.py)	~ 900 рядків
Кількість HTML-шаблонів	5
Кількість сторінкових маршрутів	16
Кількість API-маршрутів	12
Кількість функціональних модулів	9
Кількість SQL-запитів із параметризацією	42
Підтримувані браузери	Chrome, Firefox, Safari, Edge
Мінімальна ширина екрана	320 px
Теми оформлення	Темна, Світла
Мова інтерфейсу	Українська

3.6 Висновки до розділу 3

У третьому розділі виконано повну реалізацію та тестування інтерактивного онлайн-планера для тайм-менеджменту на основі проєктних рішень, сформульованих у другому розділі. Основні результати полягають у наступному.

По-перше, реалізовано серверну частину системи мовою Python з використанням фреймворку Flask та СУБД SQLite. Модуль database.py містить функцію ініціалізації бази даних (7 таблиць, 5 індексів, каскадне видалення), контекстний менеджер із підтримкою WAL та ACID-транзакцій, а також повний набір CRUD-функцій із параметризованими SQL-запитами для кожної з семи сутностей. Модуль app.py реалізує 16 сторінкових маршрутів та 12 REST API маршрутів, кожен із яких захищений декоратором @login_required (рис. 3.1–3.7).

По-друге, реалізовано підсистему безпеки, що охоплює хешування паролів алгоритмом bcrypt із cost factor 12 та унікальною сіллю, сесійну автентифікацію через Flask-Login із HttpOnly cookies, дворівневу авторизацію (декоратор @login_required та фільтр user_id у SQL-запитах), захист від SQL-ін'єкцій (42 параметризовані запити), захист від XSS (автоматичне екранування Jinja2) та серверну валідацію вхідних даних. Конфігурація безпеки включає генерацію криптографічно стійкого secret_key через os.urandom(32) та увімкнення зовнішніх ключів SQLite (рис. 3.8–3.11) [57].

По-третє, розроблено інтерактивний інтерфейс користувача з дев'ятьма функціональними модулями: дашборд із прогрес-кільцем та статистикою (рис. 3.12), модуль завдань із фільтрацією та пошуком (рис. 3.14), календар із місячним виглядом, Pomodoro-таймер із SVG-анімацією та автоматичним логуванням (рис. 3.15), трекер звичок із тижневим відображенням та optimistic UI update (рис. 3.16), модуль цілей із прогрес-баром (рис. 3.18), матриця Ейзенхауера з автоматичним розподілом (рис. 3.20), аналітика зі стовпчиковою діаграмою та розподілами (рис. 3.21), нотатки з inline editing (рис. 3.22) та особистий кабінет із підтримкою двох тем (рис. 3.23). Модальні вікна для CRUD-операцій (рис. 3.13, 3.17, 3.19) забезпечують створення об'єктів без перезавантаження сторінки через AJAX-запити Fetch API [53].

По-четверте, проведено комплексне тестування системи за трьома напрямками. Функціональне тестування з використанням Flask test_client() підтвердило коректну роботу всіх 28 маршрутів та повний цикл CRUD для кожного модуля. Тестування безпеки за моделлю STRIDE (таблиця 2.3) верифікувало захист від SQL-ін'єкцій, XSS, IDOR та несанкціонованого доступу. Юзабіліті-тестування підтвердило відповідність вимозі «не більше двох кліків» (NFR-3) та коректне відображення на мобільних екранах від 375 px (NFR-8).

По-п'яте, проведено аналіз відповідності реалізації вимогам (таблиця 3.1), який підтвердив повне виконання всіх 11 функціональних та 8 нефункціональних вимог. Порівняння з чотирма існуючими аналогами (таблиця 3.2) показало, що TimePlanner є єдиним рішенням, яке одночасно підтримує всі методики тайм-

менеджменту, є повністю безкоштовним, підтримує локальне розгортання, забезпечує повний контроль над даними та має повну українську локалізацію.

По-шосте, визначено п'ять перспективних напрямків розвитку: міграція на PostgreSQL для хмарного розгортання, розробка PWA [54] для мобільних пристроїв, інтеграція штучного інтелекту для автоматичного планування, функціонал командної співпраці та розширення модуля аналітики. Кількісні характеристики системи узагальнено у таблиці 3.3.

Отримані результати підтверджують досягнення мети дипломної роботи – створення інтерактивного онлайн-планера, який об'єднує чотири методики тайм-менеджменту (Pomodoro, матрицю Ейзенхауера, GTD та трекер звичок) в єдиному україномовному інтерфейсі з повним контролем користувача над персональними даними.

ВИСНОВКИ

У кваліфікаційній роботі вирішено актуальне завдання розробки інтерактивного онлайн-планера для тайм-менеджменту, який об’єднує чотири методики керування часом в єдиному україномовному інтерфейсі з повним контролем користувача над персональними даними. Основні результати роботи полягають у наступному.

1. Проаналізовано предметну область тайм-менеджменту та обґрунтовано ефективність цифрових інструментів керування часом на основі мета-аналізу 158 досліджень. Розглянуто три ключові методики (Pomodoro, матриця Ейзенхауера, GTD) та механізм трекінгу звичок; показано, що ці методики взаємодоповнюють одна одну, охоплюючи різні рівні керування часом. Порівняльний аналіз чотирьох існуючих онлайн-планерів (Todoist, TickTick, Notion, Any.do) за десятьма критеріями виявив нішу для інтегрованого, безкоштовного, україномовного рішення з локальним розгортанням.

2. Сформульовано одинадцять функціональних вимог (FR-1 – FR-11), що охоплюють дев’ять модулів системи, вісім нефункціональних вимог (NFR-1 – NFR-8) за моделлю ISO/IEC 25010:2011 з конкретними кількісними критеріями, а також вимоги безпеки на основі стандарту OWASP Top 10:2021 з десятьма контрзаходами.

3. Спроєктовано трирівневу клієнт-серверну архітектуру (представлення, бізнес-логіка, дані) із комбінованим підходом до взаємодії: серверний рендеринг Jinja2 та REST API. Побудовано 18 UML-діаграм (варіантів використання, класів, послідовності, діяльності, компонентів, розгортання, STRIDE, DFD). Спроєктовано реляційну базу даних із 7 таблиць, 6 зв’язків та 5 індексів у третій нормальній формі. Виконано UX-проєктування: карта сайту Hub-and-Spoke, 2 user flows, 2 wireframes, 8 обґрунтованих UX-рішень. Побудовано модель загроз STRIDE з архітектурою підсистеми безпеки.

4. Реалізовано повнофункціональний веб-застосунок мовою Python з використанням Flask, SQLite та vanilla JavaScript. Серверна частина (~ 900 рядків

коду) містить 16 сторінкових та 12 API маршрутів, 42 параметризовані SQL-запити, модуль аналітики з агрегаційними запитами. Підсистема безпеки реалізує хешування bcrypt (cost factor 12), сесійну автентифікацію Flask-Login з HttpOnly cookies, дворівневу авторизацію, захист від SQL-ін’єкцій та XSS. Інтерфейс містить дев’ять модулів з інтерактивними компонентами: SVG-анімація Pomodoro, optimistic UI update для звичок, CSS-змінні для тем, адаптивна верстка від 320 px.

5. Проведено комплексне тестування: автоматизоване функціональне (28 маршрутів через Flask test_client()), тестування безпеки за моделлю STRIDE (SQL-ін’єкції, XSS, IDOR, несанкціонований доступ) та юзабіліті-тестування (правило «2 кліки», мобільні екрани 375 px). Результати підтвердили повну відповідність усім 11 функціональним та 8 нефункціональним вимогам. Порівняння з аналогами показало, що TimePlanner є єдиним рішенням, яке одночасно підтримує всі методики, є безкоштовним, підтримує self-hosted розгортання та має повну українську локалізацію.

6. Визначено п’ять перспективних напрямків розвитку: міграція на PostgreSQL, розробка PWA, інтеграція штучного інтелекту, функціонал командної співпраці та розширення аналітики.

Мету кваліфікаційної роботи досягнуто: розроблено інтерактивний онлайн-планер для тайм-менеджменту, який задовольняє всі сформульовані вимоги та може бути використаний для організації робочого часу студентами, викладачами та фахівцями.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Aeon B., Faber A., Panaccio A. Does time management work? A meta-analysis. PLOS ONE. 2021. Vol. 16, no. 1. e0245066. URL: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0245066 (date of access: 15.03.2026).

2. Wolters C. A., Won S., Hussain M. Examining the relations of time management and procrastination within a model of self-regulated learning. Metacognition and Learning. 2017. Vol. 12, no. 3. P. 381–399. URL: https://link.springer.com/article/10.1007/s11409-017-9174-1 (date of access: 15.03.2026).

3. Ahmady S., Khajeali N., Kalantarion M., Sharifi F., Yaseri M. Relation between stress, time management, and academic achievement in preclinical medical education: A systematic review and meta-analysis. Journal of Education and Health Promotion. 2021. Vol. 10, no. 1. P. 32. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC8057171/ (date of access: 15.03.2026).

4. Biwer F., Wiradhany W., Oude Egbrink M. G. A., de Bruin A. B. H. Understanding effort regulation: Comparing 'Pomodoro' breaks and self-regulated breaks. British Journal of Educational Psychology. 2023. Vol. 93, Suppl. 2. P. 353–367. URL: https://bpspsychub.onlinelibrary.wiley.com/doi/10.1111/bjep.12593 (date of access: 15.03.2026).

5. Kuile M., Schoolderman C., Biwer F., De Bruin A. B. H. Investigating the Effectiveness of Self-Regulated, Pomodoro, and Flowtime Break-Taking Techniques Among Students. Behavioral Sciences. 2025. Vol. 15, no. 7. P. 861. URL: https://www.mdpi.com/2076-328X/15/7/861 (date of access: 15.03.2026).

6. Eisenhower D. D. Address at the Second Assembly of the World Council of Churches, Evanston, Illinois. The American Presidency Project. 1954. URL: https://www.presidency.ucsb.edu/documents/address-the-second-assembly-the-world-council-churches-evanston-illinois (date of access: 15.03.2026).

7. Shaddy F., Shah A. K. Deciding who to give to: A social distance perspective of charitable donations. Journal of the Association for Consumer Research. 2018. Vol. 3, no. 3. P. 343–354. URL: https://www.journals.uchicago.edu/doi/10.1086/698217 (date of access: 15.03.2026).

8. Heylighen F., Vidal C. Getting Things Done: The Science behind Stress-Free Productivity. Long Range Planning. 2008. Vol. 41, no. 6. P. 585–605. URL: https://pespmc1.vub.ac.be/Papers/GTD-cognition.pdf (date of access: 15.03.2026).

9. Gardner B., Rebar A. L., Lally P. How does habit form? Guidelines for tracking real-world habit formation. Cogent Psychology. 2022. Vol. 9, no. 1. P. 2041277. URL: https://www.tandfonline.com/doi/full/10.1080/23311908.2022.2041277 (date of access: 15.03.2026).

10. Stieger M., Flückiger C., Rüegger D., Kowatsch T., Roberts B. W., Allemand M. Changing personality traits with the help of a digital personality change intervention. Proceedings of the National Academy of Sciences. 2021. Vol. 118, no. 8. e2017548118. URL: https://www.pnas.org/doi/10.1073/pnas.2017548118 (date of access: 15.03.2026).

11. Shaw C. The 5 best to-do list apps in 2024. Zapier. URL: https://zapier.com/blog/best-todo-list-apps/ (date of access: 15.03.2026).

12. TickTick vs. Todoist: Which to-do list app is better?. Zapier. URL: https://zapier.com/blog/ticktick-vs-todoist/ (date of access: 15.03.2026).

13. ISO/IEC 25010:2011. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. International Organization for Standardization. URL: https://www.iso.org/standard/35733.html (date of access: 15.03.2026).

14. OWASP Foundation. OWASP Top Ten. URL: https://owasp.org/www-project-top-ten/ (date of access: 15.03.2026).

15. OWASP Top 10:2021. OWASP Foundation. URL: https://owasp.org/Top10/ (date of access: 15.03.2026).

16. Flask-Bcrypt Documentation. Read the Docs. URL: https://flask-bcrypt.readthedocs.io/en/latest/ (date of access: 15.03.2026).

17. Microsoft Learn. Software architecture principles. URL: https://learn.microsoft.com/en-us/azure/architecture/guide/ (date of access: 15.03.2026).

18. SQLite. Appropriate Uses For SQLite. SQLite Consortium. URL: https://www.sqlite.org/whentouse.html (date of access: 15.03.2026).

19. Object Management Group. Unified Modeling Language (UML), Version 2.5.1. OMG. 2017. URL: https://www.omg.org/spec/UML/2.5.1/ (date of access: 15.03.2026).

20. Nielsen J. Usability 101: Introduction to Usability. Nielsen Norman Group. URL: https://www.nngroup.com/articles/usability-101-introduction-to-usability/ (date of access: 15.03.2026).

21. Flask-Login Documentation. Read the Docs. URL: https://flask-login.readthedocs.io/en/latest/ (date of access: 15.03.2026).

22. TIOBE Software. TIOBE Programming Community Index. URL: https://www.tiobe.com/tiobe-index/ (date of access: 15.03.2026).

23. Flask Documentation. Pallets Projects. URL: https://flask.palletsprojects.com/en/stable/ (date of access: 15.03.2026).

24. Pallets Projects. Testing Flask Applications. Flask Documentation. URL: https://flask.palletsprojects.com/en/stable/testing/ (date of access: 15.03.2026).

25. Python Software Foundation. Python 3 Documentation. URL: https://docs.python.org/3/ (date of access: 15.03.2026).

26. Python Software Foundation. sqlite3 – DB-API 2.0 interface for SQLite databases. Python 3 Library Reference. URL: https://docs.python.org/3/library/sqlite3.html (date of access: 15.03.2026).

27. Pallets Projects. Jinja Documentation. URL: https://jinja.palletsprojects.com/en/stable/ (date of access: 15.03.2026).

28. Pallets Projects. Werkzeug Documentation. URL: https://werkzeug.palletsprojects.com/en/stable/ (date of access: 15.03.2026).

29. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : doctoral dissertation. University of California, Irvine, 2000. 180 p. URL: https://roy.gbiv.com/pubs/dissertation/fielding_dissertation.pdf (date of access: 15.03.2026).

30. Mozilla Developer Network. Fetch API – Web APIs. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API (date of access: 15.03.2026).

31. Mozilla Developer Network. Using CSS custom properties (variables). MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/CSS/Using_CSS_custom_properties (date of access: 15.03.2026).

32. Mozilla Developer Network. CSS Flexible Box Layout. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_flexible_box_layout (date of access: 15.03.2026).

33. Mozilla Developer Network. Media queries. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_media_queries (date of access: 15.03.2026).

34. Mozilla Developer Network. JavaScript Guide. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide (date of access: 15.03.2026).

35. WHATWG. HTML Living Standard. URL: https://html.spec.whatwg.org/ (date of access: 15.03.2026).

36. Ecma International. ECMAScript 2023 Language Specification. ECMA-262, 14th edition. 2023. URL: https://262.ecma-international.org/14.0/ (date of access: 15.03.2026).

37. World Wide Web Consortium. CSS Snapshot 2023. W3C Recommendation. URL: https://www.w3.org/TR/css-2023/ (date of access: 15.03.2026).

38. van Rossum G., Warsaw B., Coghlan N. PEP 8 – Style Guide for Python Code. Python Enhancement Proposals. URL: https://peps.python.org/pep-0008/ (date of access: 15.03.2026).

39. Python Software Foundation. venv — Creation of virtual environments. Python 3 Library Reference. URL: https://docs.python.org/3/library/venv.html (date of access: 15.03.2026).

40. Provos N., Mazières D. A future-adaptable password scheme. Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference. USENIX Association, 1999. P. 81–92. URL: https://www.usenix.org/legacy/events/usenix99/provos/provos.pdf (date of access: 15.03.2026).

41. OWASP Foundation. Cross Site Scripting Prevention Cheat Sheet. OWASP Cheat Sheet Series. URL: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html (date of access: 15.03.2026).

42. OWASP Foundation. SQL Injection Prevention Cheat Sheet. OWASP Cheat Sheet Series. URL: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html (date of access: 15.03.2026).

43. OWASP Foundation. Session Management Cheat Sheet. OWASP Cheat Sheet Series. URL: https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html (date of access: 15.03.2026).

44. OWASP Foundation. Password Storage Cheat Sheet. OWASP Cheat Sheet Series. URL: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html (date of access: 15.03.2026).

45. OWASP Foundation. Authentication Cheat Sheet. OWASP Cheat Sheet Series. URL: https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html (date of access: 15.03.2026).

46. OWASP Foundation. Authorization Cheat Sheet. OWASP Cheat Sheet Series. URL:

https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html (date of access: 15.03.2026).

47. OWASP Foundation. Input Validation Cheat Sheet. OWASP Cheat Sheet Series. URL: https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html (date of access: 15.03.2026).

48. OWASP Foundation. Threat Modeling Cheat Sheet. OWASP Cheat Sheet Series. URL: https://cheatsheetseries.owasp.org/cheatsheets/Threat_Modeling_Cheat_Sheet.html (date of access: 15.03.2026).

49. OWASP Foundation. REST Security Cheat Sheet. OWASP Cheat Sheet Series. URL: https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html (date of access: 15.03.2026).

50. Про захист персональних даних : Закон України від 01.06.2010 р. № 2297-VI : станом на 27 квіт. 2024 р. URL: https://zakon.rada.gov.ua/laws/show/2297-17 (дата звернення: 15.03.2026).

51. Про інформацію : Закон України від 02.10.1992 р. № 2657-XII : станом на 31 берез. 2023 р. URL: https://zakon.rada.gov.ua/laws/show/2657-12 (дата звернення: 15.03.2026).

52. Регламент Європейського Парламенту і Ради (ЄС) 2016/679 від 27 квітня 2016 року про захист фізичних осіб у зв'язку з опрацюванням персональних даних (Загальний регламент про захист даних). URL: https://zakon.rada.gov.ua/laws/show/984_008-16 (дата звернення: 15.03.2026).

53. SQLite Consortium. SQL As Understood By SQLite. URL: https://www.sqlite.org/lang.html (date of access: 15.03.2026).

54. SQLite Consortium. Write-Ahead Logging. URL: https://www.sqlite.org/wal.html (date of access: 15.03.2026).

55. Codd E. F. A relational model of data for large shared data banks. Communications of the ACM. 1970. Vol. 13, no. 6. P. 377–387. URL: https://dl.acm.org/doi/pdf/10.1145/362384.362685 (date of access: 15.03.2026).

56. SQLite Consortium. SQLite Documentation. URL: https://www.sqlite.org/docs.html (date of access: 15.03.2026).

57. Nielsen J. 10 Usability Heuristics for User Interface Design. Nielsen Norman Group. URL: https://www.nngroup.com/articles/ten-usability-heuristics/ (date of access: 15.03.2026).

58. Nielsen J. Response Times: The 3 Important Limits. Nielsen Norman Group. URL: https://www.nngroup.com/articles/response-times-3-important-limits/ (date of access: 15.03.2026).

59. World Wide Web Consortium. Web Content Accessibility Guidelines (WCAG) 2.1. W3C Recommendation. URL: https://www.w3.org/TR/WCAG21/ (date of access: 15.03.2026).

60. Wood W., Rünger D. Psychology of Habit. Annual Review of Psychology. 2016. Vol. 67. P. 289–314. URL: https://www.annualreviews.org/doi/10.1146/annurev-psych-122414-033417 (date of access: 15.03.2026).

61. Ersche K. D., Lim T.-V., Ward L. H. E., Robbins T. W., Stochl J. Creature of Habit: A self-report measure of habitual routines and automatic tendencies in everyday life. Personality and Individual Differences. 2017. Vol. 116. P. 73–85. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC5508990/ (date of access: 15.03.2026).

62. Git SCM. Git Documentation. URL: https://git-scm.com/doc (date of access: 15.03.2026).

63. Машкіна, І. В., Абрамов, В. О., Носенко, Т. І., Іваніченко, Є. В., & Яскевич, В. О. (2024). Навчально-методичний посібник до виконання і захисту кваліфікаційної роботи бакалавра спеціальностей 122 Комп’ютерні науки для здобувачів вищої освіти денної форми навчання.

ДОДАТКИ

Додаток А

Лістинг коду app.py

```
"""
app.py – Головний модуль Flask-додатку «Інтерактивний онлайн-планер для тайм-менеджменту».
Реалізує автентифікацію, REST API та серверну логіку.
"""

from flask import Flask, render_template, request, redirect, url_for, flash, jsonify, session
from flask_login import LoginManager, UserMixin, login_user, logout_user, login_required, current_user
from flask_bcrypt import Bcrypt
from datetime import date, timedelta
import os
import re

import database as db

app = Flask(__name__)
app.secret_key = os.urandom(32)
app.config["SESSION_COOKIE_HTTPONLY"] = True

bcrypt = Bcrypt(app)
login_manager = LoginManager(app)
login_manager.login_view = "login"
login_manager.login_message = "Будь ласка, увійдіть в систему."


# ─────────────────────────────────────────────
#  User model for Flask-Login
# ─────────────────────────────────────────────
class User(UserMixin):
    def __init__(self, row):
        self.id = row["id"]
        self.username = row["username"]
        self.email = row["email"]
        self.full_name = row["full_name"]
        self.avatar_color = row["avatar_color"]
        self.theme = row["theme"]
        self.pomodoro_work = row["pomodoro_work"]
        self.pomodoro_break = row["pomodoro_break"]
        self.pomodoro_long_break = row["pomodoro_long_break"]
        self.pomodoro_sessions = row["pomodoro_sessions"]

    @property
    def initials(self):
        if self.full_name:
            parts = self.full_name.strip().split()
            return "".join(p[0].upper() for p in parts[:2])
        return self.username[:2].upper()


@login_manager.user_loader
def load_user(user_id):
    row = db.get_user_by_id(int(user_id))
    return User(row) if row else None
```

```
    # ──────────────────────────────────────────────
    #  HELPERS
    # ──────────────────────────────────────────────
    def validate_email(email):
        return re.match(r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$",
email) is not None


    # ──────────────────────────────────────────────
    #  AUTH ROUTES
    # ──────────────────────────────────────────────
    @app.route("/register", methods=["GET", "POST"])
    def register():
        if current_user.is_authenticated:
            return redirect(url_for("dashboard"))
        if request.method == "POST":
            username = request.form.get("username", "").strip()
            email = request.form.get("email", "").strip().lower()
            password = request.form.get("password", "")
            full_name = request.form.get("full_name", "").strip()

            errors = []
            if len(username) < 3:
                errors.append("Логін повинен містити щонайменше 3 символи.")
            if not validate_email(email):
                errors.append("Невірний формат email.")
            if len(password) < 6:
                errors.append("Пароль повинен містити щонайменше 6 символів.")
            if db.get_user_by_username(username):
                errors.append("Цей логін вже зайнятий.")
            if db.get_user_by_email(email):
                errors.append("Цей email вже зареєстрований.")

            if errors:
                for e in errors:
                    flash(e, "error")
                return render_template("register.html", username=username,
email=email, full_name=full_name)

            pw_hash = bcrypt.generate_password_hash(password).decode("utf-8")
            user_id = db.create_user(username, email, pw_hash, full_name)
            user = User(db.get_user_by_id(user_id))
            login_user(user)
            flash("Реєстрація успішна! Ласкаво просимо.", "success")
            return redirect(url_for("dashboard"))

        return render_template("register.html")


    @app.route("/login", methods=["GET", "POST"])
    def login():
        if current_user.is_authenticated:
            return redirect(url_for("dashboard"))
        if request.method == "POST":
            username = request.form.get("username", "").strip()
            password = request.form.get("password", "")

            user_row = db.get_user_by_username(username)
            if not user_row:
                user_row = db.get_user_by_email(username)
```

```
            if user_row and
bcrypt.check_password_hash(user_row["password_hash"], password):
                user = User(user_row)
                login_user(user, remember=request.form.get("remember") ==
"on")
                return redirect(url_for("dashboard"))
            else:
                flash("Невірний логін або пароль.", "error")
                return render_template("login.html", username=username)

        return render_template("login.html")


    @app.route("/logout")
    @login_required
    def logout():
        logout_user()
        return redirect(url_for("login"))


    # ──────────────────────────────────────────
    #  MAIN PAGES
    # ──────────────────────────────────────────
    @app.route("/")
    def index():
        if current_user.is_authenticated:
            return redirect(url_for("dashboard"))
        return redirect(url_for("login"))


    @app.route("/dashboard")
    @login_required
    def dashboard():
        today = date.today().isoformat()
        today_tasks = db.get_tasks(current_user.id, date_from=today,
date_to=today)
        overdue = [dict(t) for t in db.get_tasks(current_user.id,
date_to=(date.today() - timedelta(days=1)).isoformat(), completed=False)]
        analytics = db.get_analytics(current_user.id)
        pomodoro_today = db.get_pomodoro_today(current_user.id)
        goals = db.get_goals(current_user.id)
        habits = db.get_habits(current_user.id)

        completed_today = sum(1 for t in today_tasks if t["completed"])
        total_today = len(today_tasks)
        pct = round(completed_today / total_today * 100) if total_today else 0

        return render_template("app.html",
                               page="dashboard",
                               today_tasks=[dict(t) for t in today_tasks],
                               overdue=overdue,
                               analytics=analytics,
                               pomo_today=pomodoro_today,
                               goals=goals[:3],
                               habits=habits,
                               completed_today=completed_today,
                               total_today=total_today,
                               pct=pct,
                               today_str=today)


    @app.route("/tasks")
    @login_required
```

```
    def tasks_page():
        cat = request.args.get("category", "")
        pri = request.args.get("priority", "")
        search = request.args.get("search", "")
        tasks = db.get_tasks(current_user.id,
                             category=cat or None,
                             priority=pri or None,
                             search=search or None)
        return render_template("app.html", page="tasks", tasks=[dict(t) for t
in tasks],
                               filter_cat=cat, filter_pri=pri, search=search)


    @app.route("/calendar")
    @login_required
    def calendar_page():
        year = int(request.args.get("year", date.today().year))
        month = int(request.args.get("month", date.today().month))
        sel = request.args.get("day", date.today().isoformat())

        first = date(year, month, 1)
        if month == 12:
            last = date(year + 1, 1, 1) - timedelta(days=1)
        else:
            last = date(year, month + 1, 1) - timedelta(days=1)

        tasks = db.get_tasks(current_user.id, date_from=first.isoformat(),
date_to=last.isoformat())
        task_map = {}
        for t in tasks:
            d = t["date"]
            if d not in task_map:
                task_map[d] = []
            task_map[d].append(dict(t))

        day_tasks = db.get_tasks(current_user.id, date_from=sel, date_to=sel)

        return render_template("app.html", page="calendar",
                               year=year, month=month, sel_day=sel,
                               task_map=task_map, day_tasks=[dict(t) for t in
day_tasks],
                               first_day=first, last_day=last)


    @app.route("/pomodoro")
    @login_required
    def pomodoro_page():
        stats = db.get_pomodoro_today(current_user.id)
        return render_template("app.html", page="pomodoro", pomo_stats=stats)


    @app.route("/habits")
    @login_required
    def habits_page():
        habits = db.get_habits(current_user.id)
        today = date.today()
        weekdays_ua = ['Пн','Вт','Ср','Чт','Пт','Сб','Нд']
        last7 = []
        for i in range(6, -1, -1):
            d = today - timedelta(days=i)
            last7.append({"date": d.isoformat(), "day": d.day, "wd":
weekdays_ua[d.weekday()], "is_today": i == 0})
```

```
          return render_template("app.html", page="habits", habits=habits,
last7=last7)


      @app.route("/goals")
      @login_required
      def goals_page():
          goals = db.get_goals(current_user.id)
          return render_template("app.html", page="goals", goals=[dict(g) for g
in goals])


      @app.route("/eisenhower")
      @login_required
      def eisenhower_page():
          tasks = [dict(t) for t in db.get_tasks(current_user.id,
completed=False)]
          quadrants = {1: [], 2: [], 3: [], 4: []}
          for t in tasks:
              urgent = t["priority"] == "high"
              important = t["category"] in ("work", "study", "meeting")
              if urgent and important:
                  quadrants[1].append(t)
              elif urgent and not important:
                  quadrants[2].append(t)
              elif not urgent and important:
                  quadrants[3].append(t)
              else:
                  quadrants[4].append(t)
          return render_template("app.html", page="eisenhower",
quadrants=quadrants)


      @app.route("/analytics")
      @login_required
      def analytics_page():
          data = db.get_analytics(current_user.id, days=30)
          pomo = db.get_pomodoro_stats(current_user.id, days=30)
          return render_template("app.html", page="analytics", analytics=data,
pomo_stats=[dict(p) for p in pomo])


      @app.route("/notes")
      @login_required
      def notes_page():
          notes = db.get_notes(current_user.id)
          return render_template("app.html", page="notes", notes=[dict(n) for n
in notes])


      @app.route("/profile", methods=["GET", "POST"])
      @login_required
      def profile():
          if request.method == "POST":
              full_name = request.form.get("full_name", "").strip()
              email = request.form.get("email", "").strip().lower()
              avatar_color = request.form.get("avatar_color", "#6366f1")
              theme = request.form.get("theme", "dark")

              updates = {"full_name": full_name, "avatar_color": avatar_color,
"theme": theme}

              if email != current_user.email:
```

```
                if not validate_email(email):
                    flash("Невірний формат email.", "error")
                    return redirect(url_for("profile"))
                existing = db.get_user_by_email(email)
                if existing and existing["id"] != current_user.id:
                    flash("Цей email вже зареєстрований.", "error")
                    return redirect(url_for("profile"))
                updates["email"] = email

            new_pw = request.form.get("new_password", "")
            if new_pw:
                if len(new_pw) < 6:
                    flash("Пароль повинен містити щонайменше 6 символів.",
"error")
                    return redirect(url_for("profile"))
                updates["password_hash"] =
bcrypt.generate_password_hash(new_pw).decode("utf-8")

            # Pomodoro settings
            updates["pomodoro_work"] = int(request.form.get("pomodoro_work",
25))
            updates["pomodoro_break"] = int(request.form.get("pomodoro_break",
5))
            updates["pomodoro_long_break"] =
int(request.form.get("pomodoro_long_break", 15))
            updates["pomodoro_sessions"] =
int(request.form.get("pomodoro_sessions", 4))

            db.update_user(current_user.id, **updates)
            flash("Профіль оновлено.", "success")
            return redirect(url_for("profile"))

        return render_template("app.html", page="profile")


    # ──────────────────────────────────────────
    #  API ROUTES (AJAX)
    # ──────────────────────────────────────────
    @app.route("/api/tasks", methods=["POST"])
    @login_required
    def api_create_task():
        d = request.json
        tid = db.create_task(current_user.id, d["title"], d["date"],
                             description=d.get("description", ""),
                             start_time=d.get("start_time", ""),
                             end_time=d.get("end_time", ""),
                             category=d.get("category", "other"),
                             priority=d.get("priority", "medium"))
        return jsonify({"ok": True, "id": tid})


    @app.route("/api/tasks/<int:tid>", methods=["PUT"])
    @login_required
    def api_update_task(tid):
        d = request.json
        db.update_task(tid, current_user.id, **d)
        return jsonify({"ok": True})


    @app.route("/api/tasks/<int:tid>/toggle", methods=["POST"])
    @login_required
    def api_toggle_task(tid):
        task = db.get_task(tid, current_user.id)
```

```
            if task:
                db.update_task(tid, current_user.id, completed=not
bool(task["completed"]))
            return jsonify({"ok": True})


        @app.route("/api/tasks/<int:tid>", methods=["DELETE"])
        @login_required
        def api_delete_task(tid):
            db.delete_task(tid, current_user.id)
            return jsonify({"ok": True})


        @app.route("/api/goals", methods=["POST"])
        @login_required
        def api_create_goal():
            d = request.json
            gid = db.create_goal(current_user.id, d["title"],
                                 description=d.get("description", ""),
                                 deadline=d.get("deadline", ""),
                                 progress=d.get("progress", 0))
            return jsonify({"ok": True, "id": gid})


        @app.route("/api/goals/<int:gid>", methods=["PUT"])
        @login_required
        def api_update_goal(gid):
            db.update_goal(gid, current_user.id, **request.json)
            return jsonify({"ok": True})


        @app.route("/api/goals/<int:gid>", methods=["DELETE"])
        @login_required
        def api_delete_goal(gid):
            db.delete_goal(gid, current_user.id)
            return jsonify({"ok": True})


        @app.route("/api/habits", methods=["POST"])
        @login_required
        def api_create_habit():
            d = request.json
            hid = db.create_habit(current_user.id, d["title"], d.get("icon",
"📌"), d.get("frequency", "daily"))
            return jsonify({"ok": True, "id": hid})


        @app.route("/api/habits/<int:hid>/toggle", methods=["POST"])
        @login_required
        def api_toggle_habit(hid):
            d = request.json
            db.toggle_habit_log(hid, current_user.id, d["date"])
            return jsonify({"ok": True})


        @app.route("/api/habits/<int:hid>", methods=["DELETE"])
        @login_required
        def api_delete_habit(hid):
            db.delete_habit(hid, current_user.id)
            return jsonify({"ok": True})


        @app.route("/api/pomodoro/log", methods=["POST"])
```

```
@login_required
def api_log_pomodoro():
    d = request.json
    db.log_pomodoro(current_user.id, d.get("duration", 25), d.get("type",
"work"))
    return jsonify({"ok": True})


@app.route("/api/notes", methods=["POST"])
@login_required
def api_create_note():
    d = request.json
    nid = db.create_note(current_user.id, d.get("title", ""),
d.get("content", ""), d.get("color", "#6366f1"))
    return jsonify({"ok": True, "id": nid})


@app.route("/api/notes/<int:nid>", methods=["PUT"])
@login_required
def api_update_note(nid):
    db.update_note(nid, current_user.id, **request.json)
    return jsonify({"ok": True})


@app.route("/api/notes/<int:nid>", methods=["DELETE"])
@login_required
def api_delete_note(nid):
    db.delete_note(nid, current_user.id)
    return jsonify({"ok": True})


# ──────────────────────────────────────────────
#  RUN
# ──────────────────────────────────────────────
@app.context_processor
def inject_helpers():
    """Inject date helpers into all templates."""
    from datetime import date as _date, datetime as _datetime, timedelta
as _td
    return {
        "now": _datetime.now,
        "today_str": _date.today().isoformat(),
        "import_date": lambda: _date.today(),
        "import_td": lambda **kw: _td(**kw),
    }


if __name__ == "__main__":
    db.init_db()
    print("\n  🚀 TimePlanner запущено: http://localhost:5000\n")
    app.run(debug=True, host="0.0.0.0", port=5000)
```

Додаток Б

Лістинг коду login.html

```
{% extends "base.html" %}
{% block title %}Вхід – TimePlanner{% endblock %}
{% block body %}
<div class="auth-page">
    <div class="auth-card">
        <div class="text-center mb-6">
            <div class="logo" style="justify-content:center;margin-bottom:8px;">
                <div class="logo-icon">⏱</div>
                <span>TimePlanner</span>
            </div>
            <p class="text-muted text-sm">Увійдіть, щоб продовжити</p>
        </div>

        {% with messages = get_flashed_messages(with_categories=true) %}
        {% for cat, msg in messages %}
        <div class="flash flash-{{ cat }}">{{ msg }}</div>
        {% endfor %}
        {% endwith %}

        <div class="card">
            <form method="POST">
                <div class="form-group">
                    <label class="form-label">Логін або email</label>
                    <input type="text" name="username" class="form-input"
      required
                           value="{{ username|default('') }}"
      placeholder="your_login" autofocus>
                </div>
                <div class="form-group">
                    <label class="form-label">Пароль</label>
                    <input type="password" name="password" class="form-input"
      required placeholder="••••••">
                </div>
                <div class="flex items-center justify-between mb-4">
                    <label class="flex items-center gap-2 text-sm text-muted"
      style="cursor:pointer">
                        <input type="checkbox" name="remember"> Запам'ятати мене
                    </label>
                </div>
                <button type="submit" class="btn btn-primary"
      style="width:100%;justify-content:center;">
                    Увійти
                </button>
            </form>
        </div>
        <p class="text-center text-sm text-muted mt-4">
            Немає акаунту? <a href="{{ url_for('register')
      }}">Зареєструватися</a>
        </p>
    </div>
</div>
{% endblock %}
```

Додаток В

Лістинг коду database.py

```
"""
database.py – Модуль бази даних SQLite для онлайн-планера тайм-
менеджменту.
Містить ініціалізацію БД та всі CRUD-операції.
"""

import sqlite3
import os
from datetime import datetime, date, timedelta
from contextlib import contextmanager

DB_PATH = os.path.join(os.path.dirname(os.path.abspath(__file__)),
"planner.db")


@contextmanager
def get_db():
    """Контекстний менеджер для підключення до SQLite."""
    conn = sqlite3.connect(DB_PATH)
    conn.row_factory = sqlite3.Row
    conn.execute("PRAGMA journal_mode=WAL")
    conn.execute("PRAGMA foreign_keys=ON")
    try:
        yield conn
        conn.commit()
    except Exception:
        conn.rollback()
        raise
    finally:
        conn.close()


def init_db():
    """Ініціалізація структури бази даних."""
    with get_db() as conn:
        conn.executescript("""
        -- Таблиця користувачів
        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            username TEXT NOT NULL UNIQUE,
            email TEXT NOT NULL UNIQUE,
            password_hash TEXT NOT NULL,
            full_name TEXT DEFAULT '',
            avatar_color TEXT DEFAULT '#6366f1',
            theme TEXT DEFAULT 'dark',
            pomodoro_work INTEGER DEFAULT 25,
            pomodoro_break INTEGER DEFAULT 5,
            pomodoro_long_break INTEGER DEFAULT 15,
            pomodoro_sessions INTEGER DEFAULT 4,
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        );

        -- Таблиця завдань
        CREATE TABLE IF NOT EXISTS tasks (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            user_id INTEGER NOT NULL,
            title TEXT NOT NULL,
            description TEXT DEFAULT '',
            date TEXT NOT NULL,
```

```
    start_time TEXT DEFAULT '',
    end_time TEXT DEFAULT '',
    category TEXT DEFAULT 'other',
    priority TEXT DEFAULT 'medium',
    completed INTEGER DEFAULT 0,
    completed_at TIMESTAMP,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);

-- Таблиця цілей
CREATE TABLE IF NOT EXISTS goals (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    title TEXT NOT NULL,
    description TEXT DEFAULT '',
    deadline TEXT DEFAULT '',
    progress INTEGER DEFAULT 0,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);

-- Таблиця звичок
CREATE TABLE IF NOT EXISTS habits (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    title TEXT NOT NULL,
    icon TEXT DEFAULT '📌',
    frequency TEXT DEFAULT 'daily',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);

-- Таблиця відмічань звичок
CREATE TABLE IF NOT EXISTS habit_logs (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    habit_id INTEGER NOT NULL,
    log_date TEXT NOT NULL,
    UNIQUE(habit_id, log_date),
    FOREIGN KEY (habit_id) REFERENCES habits(id) ON DELETE CASCADE
);

-- Таблиця pomodoro-сесій
CREATE TABLE IF NOT EXISTS pomodoro_sessions (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    session_date TEXT NOT NULL,
    duration_minutes INTEGER NOT NULL,
    session_type TEXT DEFAULT 'work',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);

-- Таблиця нотаток
CREATE TABLE IF NOT EXISTS notes (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    title TEXT DEFAULT '',
    content TEXT DEFAULT '',
    color TEXT DEFAULT '#6366f1',
    pinned INTEGER DEFAULT 0,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
```

```
                FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
            );

            -- Індекси для продуктивності
            CREATE INDEX IF NOT EXISTS idx_tasks_user_date ON tasks(user_id,
date);
            CREATE INDEX IF NOT EXISTS idx_tasks_user_completed ON
tasks(user_id, completed);
            CREATE INDEX IF NOT EXISTS idx_habits_user ON habits(user_id);
            CREATE INDEX IF NOT EXISTS idx_habit_logs_habit ON
habit_logs(habit_id, log_date);
            CREATE INDEX IF NOT EXISTS idx_pomodoro_user_date ON
pomodoro_sessions(user_id, session_date);
            CREATE INDEX IF NOT EXISTS idx_notes_user ON notes(user_id);
            """)


    # ──────────────────────────────────────────
    #  USERS
    # ──────────────────────────────────────────
    def create_user(username, email, password_hash, full_name=""):
        with get_db() as conn:
            conn.execute(
                "INSERT INTO users (username, email, password_hash, full_name)
VALUES (?, ?, ?, ?)",
                (username, email, password_hash, full_name),
            )
            return conn.execute("SELECT last_insert_rowid()").fetchone()[0]


    def get_user_by_id(user_id):
        with get_db() as conn:
            return conn.execute("SELECT * FROM users WHERE id = ?",
(user_id,)).fetchone()


    def get_user_by_username(username):
        with get_db() as conn:
            return conn.execute("SELECT * FROM users WHERE username = ?",
(username,)).fetchone()


    def get_user_by_email(email):
        with get_db() as conn:
            return conn.execute("SELECT * FROM users WHERE email = ?",
(email,)).fetchone()


    def update_user(user_id, **kwargs):
        allowed = {"full_name", "avatar_color", "theme", "pomodoro_work",
"pomodoro_break",
                   "pomodoro_long_break", "pomodoro_sessions",
"password_hash", "email"}
        fields = {k: v for k, v in kwargs.items() if k in allowed}
        if not fields:
            return
        sets = ", ".join(f"{k} = ?" for k in fields)
        vals = list(fields.values()) + [user_id]
        with get_db() as conn:
            conn.execute(f"UPDATE users SET {sets} WHERE id = ?", vals)


    # ──────────────────────────────────────────
```

```
    #  TASKS
    # ──────────────────────────────────────────────
    def create_task(user_id, title, task_date, **kwargs):
        with get_db() as conn:
            conn.execute(
                """INSERT INTO tasks (user_id, title, date, description,
start_time, end_time, category, priority)
                   VALUES (?, ?, ?, ?, ?, ?, ?, ?)""",
                (user_id, title, task_date,
                 kwargs.get("description", ""),
                 kwargs.get("start_time", ""),
                 kwargs.get("end_time", ""),
                 kwargs.get("category", "other"),
                 kwargs.get("priority", "medium")),
            )
            return conn.execute("SELECT last_insert_rowid()").fetchone()[0]


    def get_tasks(user_id, date_from=None, date_to=None, completed=None,
category=None, priority=None, search=None):
        query = "SELECT * FROM tasks WHERE user_id = ?"
        params = [user_id]
        if date_from:
            query += " AND date >= ?"
            params.append(date_from)
        if date_to:
            query += " AND date <= ?"
            params.append(date_to)
        if completed is not None:
            query += " AND completed = ?"
            params.append(1 if completed else 0)
        if category:
            query += " AND category = ?"
            params.append(category)
        if priority:
            query += " AND priority = ?"
            params.append(priority)
        if search:
            query += " AND title LIKE ?"
            params.append(f"%{search}%")
        query += " ORDER BY date ASC, start_time ASC"
        with get_db() as conn:
            return conn.execute(query, params).fetchall()


    def get_task(task_id, user_id):
        with get_db() as conn:
            return conn.execute("SELECT * FROM tasks WHERE id = ? AND user_id
= ?", (task_id, user_id)).fetchone()


    def update_task(task_id, user_id, **kwargs):
        allowed = {"title", "description", "date", "start_time", "end_time",
"category", "priority", "completed"}
        fields = {k: v for k, v in kwargs.items() if k in allowed}
        if "completed" in fields:
            fields["completed_at"] = datetime.now().isoformat() if
fields["completed"] else None
        if not fields:
            return
        sets = ", ".join(f"{k} = ?" for k in fields)
        vals = list(fields.values()) + [task_id, user_id]
        with get_db() as conn:
```

```
            conn.execute(f"UPDATE tasks SET {sets} WHERE id = ? AND user_id =
?", vals)


    def delete_task(task_id, user_id):
        with get_db() as conn:
            conn.execute("DELETE FROM tasks WHERE id = ? AND user_id = ?",
(task_id, user_id))


    # ──────────────────────────────────────────
    #  GOALS
    # ──────────────────────────────────────────
    def create_goal(user_id, title, **kwargs):
        with get_db() as conn:
            conn.execute(
                "INSERT INTO goals (user_id, title, description, deadline,
progress) VALUES (?, ?, ?, ?, ?)",
                (user_id, title, kwargs.get("description", ""),
kwargs.get("deadline", ""), kwargs.get("progress", 0)),
            )
            return conn.execute("SELECT last_insert_rowid()").fetchone()[0]


    def get_goals(user_id):
        with get_db() as conn:
            return conn.execute("SELECT * FROM goals WHERE user_id = ? ORDER
BY created_at DESC", (user_id,)).fetchall()


    def update_goal(goal_id, user_id, **kwargs):
        allowed = {"title", "description", "deadline", "progress"}
        fields = {k: v for k, v in kwargs.items() if k in allowed}
        if not fields:
            return
        sets = ", ".join(f"{k} = ?" for k in fields)
        vals = list(fields.values()) + [goal_id, user_id]
        with get_db() as conn:
            conn.execute(f"UPDATE goals SET {sets} WHERE id = ? AND user_id =
?", vals)


    def delete_goal(goal_id, user_id):
        with get_db() as conn:
            conn.execute("DELETE FROM goals WHERE id = ? AND user_id = ?",
(goal_id, user_id))


    # ──────────────────────────────────────────
    #  HABITS
    # ──────────────────────────────────────────
    def create_habit(user_id, title, icon="📌", frequency="daily"):
        with get_db() as conn:
            conn.execute(
                "INSERT INTO habits (user_id, title, icon, frequency) VALUES
(?, ?, ?, ?)",
                (user_id, title, icon, frequency),
            )
            return conn.execute("SELECT last_insert_rowid()").fetchone()[0]


    def get_habits(user_id):
        with get_db() as conn:
```

```
            habits = conn.execute("SELECT * FROM habits WHERE user_id = ?
ORDER BY created_at", (user_id,)).fetchall()
            result = []
            for h in habits:
                logs = conn.execute("SELECT log_date FROM habit_logs WHERE
habit_id = ?", (h["id"],)).fetchall()
                result.append({**dict(h), "completed_dates": [l["log_date"]
for l in logs]})
            return result


    def toggle_habit_log(habit_id, user_id, log_date):
        with get_db() as conn:
            habit = conn.execute("SELECT * FROM habits WHERE id = ? AND
user_id = ?", (habit_id, user_id)).fetchone()
            if not habit:
                return
            existing = conn.execute(
                "SELECT id FROM habit_logs WHERE habit_id = ? AND log_date =
?", (habit_id, log_date)
            ).fetchone()
            if existing:
                conn.execute("DELETE FROM habit_logs WHERE id = ?",
(existing["id"],))
            else:
                conn.execute("INSERT INTO habit_logs (habit_id, log_date)
VALUES (?, ?)", (habit_id, log_date))


    def delete_habit(habit_id, user_id):
        with get_db() as conn:
            conn.execute("DELETE FROM habits WHERE id = ? AND user_id = ?",
(habit_id, user_id))


    # ──────────────────────────────────────────
    #  POMODORO SESSIONS
    # ──────────────────────────────────────────
    def log_pomodoro(user_id, duration_minutes, session_type="work"):
        with get_db() as conn:
            conn.execute(
                "INSERT INTO pomodoro_sessions (user_id, session_date,
duration_minutes, session_type) VALUES (?, ?, ?, ?)",
                (user_id, date.today().isoformat(), duration_minutes,
session_type),
            )


    def get_pomodoro_stats(user_id, days=30):
        since = (date.today() - timedelta(days=days)).isoformat()
        with get_db() as conn:
            return conn.execute(
                """SELECT session_date, SUM(duration_minutes) as
total_minutes, COUNT(*) as count
                   FROM pomodoro_sessions WHERE user_id = ? AND session_date
>= ?
                   GROUP BY session_date ORDER BY session_date""",
                (user_id, since),
            ).fetchall()


    def get_pomodoro_today(user_id):
        with get_db() as conn:
```

```
            row = conn.execute(
                """SELECT COALESCE(SUM(duration_minutes), 0) as total,
COUNT(*) as count
                   FROM pomodoro_sessions WHERE user_id = ? AND session_date =
? AND session_type = 'work'""",
                (user_id, date.today().isoformat()),
            ).fetchone()
            return dict(row)


    # ──────────────────────────────────────────
    #  NOTES
    # ──────────────────────────────────────────
    def create_note(user_id, title="", content="", color="#6366f1"):
        with get_db() as conn:
            conn.execute(
                "INSERT INTO notes (user_id, title, content, color) VALUES (?,
?, ?, ?)",
                (user_id, title, content, color),
            )
            return conn.execute("SELECT last_insert_rowid()").fetchone()[0]


    def get_notes(user_id):
        with get_db() as conn:
            return conn.execute(
                "SELECT * FROM notes WHERE user_id = ? ORDER BY pinned DESC,
updated_at DESC", (user_id,)
            ).fetchall()


    def update_note(note_id, user_id, **kwargs):
        allowed = {"title", "content", "color", "pinned"}
        fields = {k: v for k, v in kwargs.items() if k in allowed}
        fields["updated_at"] = datetime.now().isoformat()
        sets = ", ".join(f"{k} = ?" for k in fields)
        vals = list(fields.values()) + [note_id, user_id]
        with get_db() as conn:
            conn.execute(f"UPDATE notes SET {sets} WHERE id = ? AND user_id =
?", vals)


    def delete_note(note_id, user_id):
        with get_db() as conn:
            conn.execute("DELETE FROM notes WHERE id = ? AND user_id = ?",
(note_id, user_id))


    # ──────────────────────────────────────────
    #  ANALYTICS
    # ──────────────────────────────────────────
    def get_analytics(user_id, days=30):
        since = (date.today() - timedelta(days=days)).isoformat()
        today = date.today().isoformat()
        with get_db() as conn:
            # Tasks per day
            daily = conn.execute(
                """SELECT date, COUNT(*) as total,
                          SUM(CASE WHEN completed = 1 THEN 1 ELSE 0 END) as
done
                   FROM tasks WHERE user_id = ? AND date >= ? AND date <= ?
                   GROUP BY date ORDER BY date""",
                (user_id, since, today),
```

```
            ).fetchall()

            # Category breakdown
            categories = conn.execute(
                "SELECT category, COUNT(*) as cnt FROM tasks WHERE user_id = ?
GROUP BY category",
                (user_id,),
            ).fetchall()

            # Priority breakdown
            priorities = conn.execute(
                "SELECT priority, COUNT(*) as cnt FROM tasks WHERE user_id = ?
GROUP BY priority",
                (user_id,),
            ).fetchall()

            # Overall
            totals = conn.execute(
                """SELECT COUNT(*) as total,
                          SUM(CASE WHEN completed = 1 THEN 1 ELSE 0 END) as
completed
                   FROM tasks WHERE user_id = ?""",
                (user_id,),
            ).fetchone()

            # Streak
            streak = 0
            check = date.today()
            max_check = 365  # safety limit
            skipped = 0
            while max_check > 0:
                max_check -= 1
                ds = check.isoformat()
                row = conn.execute(
                    "SELECT COUNT(*) as total, SUM(CASE WHEN completed=1 THEN
1 ELSE 0 END) as done FROM tasks WHERE user_id=? AND date=?",
                    (user_id, ds),
                ).fetchone()
                if row["total"] == 0:
                    skipped += 1
                    if streak == 0 and skipped <= 3:
                        check -= timedelta(days=1)
                        continue
                    else:
                        break
                if row["done"] == row["total"]:
                    streak += 1
                    check -= timedelta(days=1)
                else:
                    break

            return {
                "daily": [dict(d) for d in daily],
                "categories": {r["category"]: r["cnt"] for r in categories},
                "priorities": {r["priority"]: r["cnt"] for r in priorities},
                "total": totals["total"] or 0,
                "completed": totals["completed"] or 0,
                "streak": streak,
            }
```

Додатко Г

Функціональні вимоги до інтерактивного онлайн-планера

Код	**Підсистема**	**Опис вимоги**
FR-1	Автентифікація	Система має забезпечувати реєстрацію нового користувача за логіном, електронною поштою та паролем; вхід за обліковими даними; вихід із системи та зміну пароля у профілі
FR-2	Особистий кабінет	Система має надавати користувачеві змогу переглядати та змінювати власний профіль: ім'я, електронну пошту, колір аватара, тему оформлення та індивідуальні налаштування таймера Pomodoro
FR-3	Керування завданнями	Система має підтримувати створення, редагування, видалення та позначення як виконаного для завдань; кожне завдання має назву, опис, дату, час початку та закінчення, категорію та пріоритет
FR-4	Фільтрація та пошук	Система має надавати змогу фільтрувати завдання за категорією, пріоритетом та здійснювати повнотекстовий пошук за назвою
FR-5	Календар	Система має відображати місячний календар з індикаторами завдань на кожен день та надавати деталізований перегляд подій за обраний день
FR-6	Pomodoro-таймер	Система має реалізовувати таймер Pomodoro з налаштуванням тривалості фокусу, короткої й довгої перерв, автоматичним перемиканням режимів і логуванням кожної сесії в базу даних
FR-7	Трекер звичок	Система має надавати змогу створювати звички з іконкою та частотою, щоденно позначати їх виконання та розраховувати серію (streak) послідовних днів
FR-8	Цілі	Система має підтримувати постановку цілей з дедлайном і прогресом у відсотках, а також швидке оновлення прогресу через передустановлені значення
FR-9	Матриця Ейзенхауера	Система має автоматично розподіляти активні завдання за чотирма квадрантами на основі значень пріоритету й категорії та забезпечувати наочне представлення
FR-10	Аналітика	Система має візуалізувати статистику продуктивності: графік активності за 30 днів, розподіл завдань за категоріями й пріоритетами, загальні показники виконання
FR-11	Нотатки	Система має підтримувати створення, редагування та видалення коротких нотаток із кольоровим маркуванням і можливістю закріплення важливих