ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Клієнт-серверний вебзастосунок CapyDo для організації персональних завдань із використанням TypeScript та PostgreSQL»

на здобуття освітнього ступеня бакалавра
зі спеціальності 126 Інформаційні системи та технології
освітньо-професійної програми «Інформаційні системи та технології»

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

_____ Юлія РЯБКО
(підпис) *(ім'я, ПРІЗВИЩЕ здобувача)*

Виконала: здобувачка вищої освіти групи ІСД-42
Юлія РЯБКО
(ім'я, ПРІЗВИЩЕ)

Керівник:
д.т.н.,доцент Ірина СРІБНА
(ім'я, ПРІЗВИЩЕ)

Рецензент: _____
(ім'я, ПРІЗВИЩЕ)

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інформаційних систем та технологій
Ступінь вищої освіти бакалавр
Спеціальність 126 Інформаційні системи та технології
Освітньо-професійна програма Інформаційні системи та технології

ЗАТВЕРДЖУЮ
Завідувач кафедрою ІСТ
_____ Каміла СТОРЧАК

« ____ » _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Рябко Юлія Василівна

1. Тема кваліфікаційної роботи: «Клієнт-серверний вебзастосунок CapyDo для організації персональних завдань із використанням TypeScript та PostgreSQL». Керівник: Ірина СРІБНА, д-р техн. наук, доцент, затверджено наказом ДУІКТ від «_20.02.2026__» березня 2026 р. № _51_.

2. Строк подання: «_01.06.2026_» травня 2026 року.

3. Вихідні дані: Node.js + Hono 4.8.3, PostgreSQL 16 + Prisma ORM 7.8.0, TypeScript 5.8.3, стандарт RFC 7519 (JWT), bcrypt, вимоги безпеки OWASP (XSS, CSRF, SQL-ін'єкції), Zod для валідації.

4. Зміст записки:
- Огляд систем управління завданнями (Todoist, Trello, Notion), формулювання вимог до застосунку.
- Обґрунтування технологічного стеку (TypeScript, Hono, PostgreSQL, Prisma, JWT, Zod).
- Проєктування архітектури, ER-моделі БД, діаграм варіантів використання та послідовності.

– Реалізація серверної (JWT-автентифікація, CRUD, валідація) та клієнтської частини (EJS-рендеринг, UI, API), тестування (модульне, інтеграційне, покриття).

5. Графічний матеріал (презентація): аналіз аналогів, вимоги до ПЗ, програмні засоби, діаграми використання та послідовності, архітектура ПЗ, ER-модель БД, екранні форми.

6. Дата видачі: «_20.02.2026_» березня 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Строк виконання	Примітка
1.	Підбір та аналіз науково-технічної літератури про управління завданнями та веб-розробку	10.02 – 25.02.2026	Аналіз аналогів, порівняння рішень
2.	Аналіз існуючих програмних рішень для управління завданнями та визначення вимог	26.02 – 10.03.2026	Функціональні та нефункціональні вимоги
3.	Вибір та обґрунтування технологічного стеку (TypeScript, Hono, PostgreSQL, Prisma, JWT)	11.03 – 20.03.2026	Порівняння альтернатив
4.	Проектування архітектури застосунку, ER-моделі БД та діаграм варіантів використання	21.03 – 25.03.2026	Дизайн системи
5.	Програмна реалізація серверної частини: автентифікація JWT, CRUD операції	26.03 – 10.04.2026	Розробка backend
6.	Реалізація клієнтської частини: EJS шаблони, рендеринг, взаємодія з API	11.04 – 20.04.2026	Розробка frontend
7.	Тестування застосунку, оформлення роботи: вступ, висновки, реферат	21.04 – 26.04.2026	Модульне та інтеграційне тестування
8.	Розробка демонстраційних матеріалів та скріншотів	26.04 – 02.05.2026	Підготовка презентації
9.	Попередній захист та фінальна коригування роботи	03.05 – 05.05.2026	Готовність до офіційного захисту

Здобувачка вищої освіти _____ Юлія РЯБКО
(підпис)

Керівник кваліфікаційної роботи _____ Ірина СРІБНА
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 53 стор., 27 табл., 17 рис., 29 джерел.

Мета роботи – проектування та розробка клієнт-серверного вебзастосунку CapyDo для організації персональних завдань із використанням TypeScript та PostgreSQL, що забезпечує безпечну автентифікацію та авторизацію, повний цикл CRUD-операцій над задачами та перевірену автоматизованими тестами якість реалізації.

Об'єкт дослідження – процеси керування персональними завданнями у вебсередовищі.

Предмет дослідження – методи та засоби реалізації клієнт-серверного вебзастосунку для управління завданнями із автентифікацією, авторизацією та збереженням даних у реляційній БД.

Короткий зміст роботи: проведено аналіз предметної області та порівняння існуючих рішень (Todoist, Trello, Notion). Обґрунтовано вибір технологічного стеку: TypeScript 5.8.3, фреймворк Hono 4.8.3, СУБД PostgreSQL з ORM Prisma 7.8.0, JWT-автентифікація, валідація Zod 4.x, шаблонізатор EJS. Спроектовано реляційну модель даних (2 сутності, 3 індекси, каскадне видалення) та маршрутну таблицю (18 ендпоінтів). Реалізовано модулі автентифікації та авторизації (bcrypt, JWT в httpOnly cookie, обмеження частоти запитів (rate limiting), CRUD задач із контролем доступу та серверний рендеринг сторінок. Проведено 98 автоматизованих тестів (Vitest); покриття коду – 88.52% за операторами, 88.73% за рядками, 91.93% за функціями, 73.41% за гілками.

Сфера використання: персональне планування задач; навчальна платформа для дисциплін із веброзробки, баз даних і кібербезпеки.

КЛЮЧОВІ СЛОВА: ВЕБЗАСТОСУНОК, TYPESCRIPT, POSTGRESQL, HONO, PRISMA, JWT, ZOD, ТЕСТУВАННЯ, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API – прикладний програмний інтерфейс (Application Programming Interface);

БД – база даних;

СУБД – система управління базами даних;

ЗВО – заклад вищої освіти;

ПЗ – програмне забезпечення;

ТЗ – технічне завдання;

UI – інтерфейс користувача (User Interface);

UX – досвід користувача (User Experience);

CRUD – операції створення, читання, оновлення, видалення (Create, Read, Update, Delete);

ESM – модульна система ECMAScript (ECMAScript Modules);

HTTP – протокол передачі гіпертексту (HyperText Transfer Protocol);

HTTPS – захищений протокол передачі гіпертексту (HTTP Secure);

JWT – JSON Web Token, відкритий стандарт передачі тверджень між сторонами;

ORM – об'єктно-реляційне відображення (Object-Relational Mapping);

SoC – розділення відповідальностей (Separation of Concerns);

SQL – мова структурованих запитів (Structured Query Language);

SSR – рендеринг на стороні сервера (Server-Side Rendering);

TTL – час існування (Time-to-Live);

UUID – унікальний ідентифікатор (Universally Unique Identifier);

YAGNI – принцип мінімальності реалізації (You Aren't Gonna Need It).

ВСТУП

Стрімкий розвиток інформаційних технологій та широке впровадження мережевих сервісів зумовили трансформацію підходів до організації праці та навчання. Фахівці різних галузей щоденно опрацьовують значний обсяг завдань, дотримуються встановлених термінів і контролюють виконання. У цьому контексті програмні засоби особистої продуктивності мають важливе практичне значення, оскільки сприяють структурованому управлінню інформацією та концентрації на пріоритетних задачах.

Наявні комерційні системи керування завданнями – Todoist, Trello, Notion та подібні – характеризуються надлишковою функціональністю для індивідуального сценарію: великою кількістю налаштувань, необхідністю реєстрації в хмарних сервісах та залежністю від зовнішніх постачальників. Водночас простіші рішення (нотатки у браузері, текстові файли) позбавлені базових механізмів безпеки й не дозволяють організувати пріоритизацію та відстеження статусів.

Актуальність визначається необхідністю створення легкого, безпечного та автономного інструменту для керування персональними завданнями, який відповідає сучасним стандартам безпеки вебзастосунків і вимогам інженерної якості програмного забезпечення.

Об’єкт дослідження – процеси керування персональними завданнями у вебсередовищі.

Предмет дослідження – методи та засоби реалізації клієнт-серверного вебзастосунку для управління завданнями із автентифікацією, авторизацією та збереженням даних у реляційній БД.

Метою роботи є розробка клієнт-серверного вебзастосунку CapyDo для організації персональних завдань із використанням TypeScript та PostgreSQL, що забезпечує безпечну взаємодію користувачів, інтуїтивно зрозумілий інтерфейс та підтверджену якість реалізації.

Для досягнення мети поставлено такі завдання:

- проаналізувати предметну область та функціональні вимоги;

- обрати технологічний стек і обґрунтувати архітектуру рішення;
- спроектувати структуру даних і серверні маршрути;
- реалізувати модулі автентифікації, авторизації та керування завданнями;
- забезпечити захист застосунку через валідацію, обмеження запитів і контроль доступу;
- провести модульне та інтеграційне тестування;
- оцінити результати розробки за кількісними показниками.

Методи дослідження: структурний аналіз, проектування реляційної БД, застосування клієнт-серверної архітектури, тестування програмного забезпечення, порівняльний аналіз технологій.

Практична цінність: реалізований застосунок можна використовувати як готовий персональний планувальник та як навчальний приклад побудови захищеного вебсервісу на основі TypeScript і PostgreSQL.

Апробація проміжних результатів дослідження здійснювалася шляхом публікації тез на всеукраїнських науково-практичних та науково-технічних конференціях. Тематика тез, зокрема аналіз JWT-вразливостей, оптимізація продуктивності Node.js-застосунків та NLP-аналіз тональності, була врахована під час формування вимог до безпеки, продуктивності та методології дослідження в даній роботі.

Структура кваліфікаційної роботи складається з вступу, чотирьох розділів, висновків та переліку посилань. У першому розділі проводиться аналіз предметної галузі та огляд існуючих рішень. У другому розділі обґрунтовується вибір технологічного стеку. У третьому розділі описано проектування архітектури та моделі даних. У четвертому розділі висвітлюється реалізація застосунку та результати тестування.

Наукова новизна роботи полягає у поєднанні формалізованого підходу до проектування архітектури, безпечної JWT-автентифікації та тестово-орієнтованої перевірки якості в межах персонального вебзастосунку. На відміну від типових навчальних CRUD-прикладів, у роботі реалізовано системний підхід до контролю

доступу, валідації даних і перевірки поведінки маршрутизації в граничних сценаріях.

Практичне значення отриманих результатів полягає в тому, що створений застосунок може бути використаний не лише як персональний інструмент керування задачами, а і як навчальна платформа для дисциплін із веброзробки, баз даних і кібербезпеки. Матеріали роботи придатні для використання у лабораторних і курсових роботах як приклад повного циклу створення клієнт-серверного ПЗ.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз предметної області

Цифрове управління задачами стало невід'ємною частиною сучасного підходу до організації праці та навчання. Збільшення кількості зобов'язань, дедлайнів і паралельних задач зумовлює попит на спеціалізовані інструменти планування, які допомагають структурувати та пріоритизувати роботу.

У задачах особистого тайм-менеджменту критичними є такі функції:

- створення задач із дедлайном і пріоритетом;
- визначення статусу виконання і рівня важливості;
- швидке редагування й видалення неактуальних записів;
- розмежування доступу до особистих даних;
- простий, зрозумілий інтерфейс без надмірного перевантаження.

Таблиця 1.1

Пріоритизація функціональних можливостей (MoSCoW)

Функція	Пріоритет	Обґрунтування
Реєстрація та автентифікація	Must	Базова умова персоналізації і захисту даних
CRUD-операції над задачами	Must	Ключова цінність застосунку
Контроль доступу до власних ресурсів	Must	Критично для безпеки
Валідація вхідних даних	Should	Підвищує надійність і стійкість до помилок
Пошук/фільтрація задач	Could	Поза межами персонального сценарію поточної версії
Командна взаємодія та ролі	Won't*	Поза межами персонального сценарію поточної версії

Примітка: Won't* – не входить у поточний обсяг реалізації, відповідає персональному (не командному) сценарію використання.

Дослідження у сфері продуктивності свідчать, що ефективність особистого планування напряму залежить від доступності інструменту: чим менше зусиль потрібно для внесення та перегляду задачі – тим вищий шанс, що користувач дійсно дотримуватиметься системи.

Для персонального сценарію важливіші швидкість доступу і ясність інтерфейсу, ніж складні корпоративні функції (дошки, ролі команд, багаторівневі робочі процеси). Це пояснює орієнтацію CapyDo на індивідуальне використання з мінімалістичною, але достатньою функціональністю: чотири можливі статуси задачі (не розпочато, в процесі, завершено, скасовано), три рівні важливості і необов'язковий дедлайн.

Важливою перевагою власного рішення є повний контроль над моделлю безпеки та даними: на відміну від хмарних сервісів, кастомний застосунок не передає персональних даних третім сторонам і не залежить від зовнішньої політики конфіденційності.

1.2 Функціональні вимоги

Система повинна забезпечувати повний цикл роботи з обліковим записом і задачами. Передбачено реєстрацію нового користувача, вхід за логіном і паролем, перегляд сторінки профілю та зміну облікових даних, а також вихід і видалення облікового запису. Щодо задач, система дозволяє створювати їх із зазначенням назви, статусу, важливості та дедлайну, переглядати список власних задач, редагувати наявні та видаляти їх.

Склад функціональних вимог із пріоритетами і зв'язком з маршрутами наведено в таблиці 1.2.

Таблиця 1.2

Функціональні вимоги із пріоритетами і маршрутами

Функція	HTTP-метод	Маршрут	Пріоритет
Реєстрація	POST	/signup	Обов’язкова
Автентифікація (login)	POST	/login	Обов’язкова
Вихід (очищення cookie-файлів)	DELETE	/:username/account	Обов’язкова
Перегляд профілю	GET	/:username/account	Обов’язкова
Оновлення даних профілю	POST	/:username/account-settings	Обов’язкова
Видалення акаунту	DELETE	/:username/account	Обов’язкова
Перегляд списку задач	GET	/:username/tasks	Обов’язкова
Створення задачі	POST	/:username/tasks/create	Обов’язкова
Редагування задачі	PUT	/:username/tasks/:taskId	Обов’язкова
Видалення задачі	DELETE	/:username/tasks/:taskId	Обов’язкова

1.3 Нефункціональні вимоги

Нефункціональні вимоги визначають якісні атрибути системи. Структурована сукупність вимог наведена в таблиці 1.3.

Таблиця 1.3

Нефункціональні вимоги до вебзастосунку CapyDo

Категорія	Вимога	Спосіб забезпечення
Безпека	Валідація вхідних даних, хешування паролів, HTTP-only cookie	Zod, bcrypt, Hono secureHeaders, sameSite=strict
Безпека	Обмеження частоти запитів	Rate limiter (200 req / 15 хв)
Надійність	Централізована обробка помилок, передбачувані коди	AppError + handleHttpError, HTTP 4xx/5xx
Продуктивність	Стиснення відповідей, кешування EJS-шаблонів	Hono compress, TTL-кеш 5 хв (в production)
Підтримуваність	TypeScript-типізація, розділення на шари	routes / services / middleware / lib
Тестованість	Unit- та integration-тести, вимірювання покриття	Vitest, 98 тестів, 88.73% покриття
Масштабованість	Розвиток без зміни архітектури	Ієрархічний поділ, stateless JWT, Prisma migrations

1.4 Постановка задачі

Необхідно побудувати вебзастосунок, в якому користувач після автентифікації отримує доступ лише до власних задач. Система повинна підтримувати повний цикл керування задачами та коректно обробляти помилки, не порушуючи цілісність даних. Технічно рішення має бути реалізоване на TypeScript із сервером Hono, PostgreSQL і Prisma.

Відповідність реалізації поставленим завданням наведено в таблиці 1.4.

Таблиця 1.4

Відповідність постановки задачі очікуваному результату

Завдання	Очікуваний результат	Статус
Реалізувати автентифікацію і авторизацію	JWT-токен (HS256) + bcrypt-хеш пароля + httpOnly cookie	Виконано
Реалізувати авторизацію	Перевірка username в URL + userId-фільтр у запитах БД	Виконано
Забезпечити CRUD задач	4 маршрути + сервісний шар	Виконано
Захистити персональні дані користувача	userId-фільтр + URL-перевірка	Виконано
Провести автоматизоване тестування	98 тестів, покриття 88.73%	Виконано
Забезпечити rate limiting і secure headers	200 req/15 хв, OWASP-рекомендації	Виконано
Реалізувати SSR-інтерфейс (EJS-шаблони)	Базові сторінки + кешування	Виконано

1.5 Аналіз програм-аналогів

Під час дослідження предметної області розглядалися три основні класи рішень: великі комерційні платформи керування проектами, мінімалістичні персональні планувальники та кастомні вебрішення навчального або локального призначення.

Todoist – один із найбільш поширених персональних планувальників. Сервіс надає зрозумілий інтерфейс, мобільні додатки і можливості фільтрації. Проте у стандартному (безкоштовному) плані функціональність обмежена, а дані зберігаються на серверах постачальника, що знижує контроль над конфіденційністю.

Trello – канбан-орієнтована платформа, побудована на основі дошок і карток. Підходить для командної роботи, однак для персонального планування може виявитися надлишково складною, особливо з огляду на широкий набір налаштувань і кількість UI-елементів.

Notion – багатофункціональний робочий простір, що поєднує нотатки, бази даних і задачі. Незважаючи на гнучкість, Notion вимагає значного часу для первинного налаштування і може бути надмірним для простих персональних задач.

CapyDo – кастомний вебзастосунок, спроектований спеціально для персонального сценарію з фокусом на безпеку, типізацію і тестованість. Не потребує зовнішніх підписок. Весь стек знаходиться під контролем розробника.

Порівняльна характеристика систем наведена у таблиці 1.5.

Таблиця 1.5

Порівняння систем управління задачами

Критерій	Todoist	Trello	Notion	CapyDo
Безкоштовний план	Так, з обмеженнями	Так, з обмеженнями	Так, з обмеженнями	Так, без обмежень
Обмеження безкоштовного плану	До 5 проєктів [27]	До 10 учасників, обмежені дошки [28]	Блоки обмежені для 2+ користувачів [29]	Відсутні
Збереження даних на власному сервері	Ні	Ні	Ні	Так
Вхід через логін і пароль	Так	Так	Так	Так
Підтримка OAuth (Google, Microsoft тощо)	Так	Так	Так	Ні
Дедлайни для задач	Так	Так	Так	Так
Пріоритети задач	Так (P1–P4)	Через мітки	Через властивості	Так (3 рівні)
Статуси задач	Виконано / Не виконано	Через переміщення між списками	Налаштовувані	4 фіксовані статуси
Командна робота	Так	Так	Так (платно)	Ні (персональний)
Мобільний застосунок	Так	Так	Так	Ні
Відкритий вихідний код	Ні	Ні	Ні	Так

У контексті теми кваліфікаційної роботи ключовим є третій підхід: створення спеціалізованого рішення, що закриває реальні потреби персонального планування і водночас демонструє інженерну якість на рівні структури коду, його тестованості та безпеки. CapyDo за більшістю критеріїв повністю відповідає цим вимогам і перевершує аналоги за рівнем прозорості реалізації безпеки. Разом з тим варто відзначити, що порівняно з великими комерційними платформами функціональні можливості CapyDo є обмеженими, а розширення для групової роботи чи інтеграції з іншими сервісами наразі не реалізовані, що може бути недоліком у певних сценаріях використання.

1.6 Критерії оцінювання якості вебзастосунку

Для об'єктивного оцінювання ефективності реалізованого продукту сформовано критерії якості. Функціональна достатність передбачає покриття ключових сценаріїв – автентифікації, CRUD задач і керування обліковим записом. Безпека охоплює захист облікових даних, контроль доступу, валідацію вхідних даних та стійкість до некоректних запитів. Продуктивність оцінюється за швидкістю формування відповіді сервером, ефективністю запитів до БД та оптимізованістю рендерингу. Супровідність визначається зрозумілою структурою коду, безпекою типів і наявністю тестів. Масштабованість означає можливість розширення функцій без радикальної зміни архітектури.

Запропоновані критерії використовуються у подальших розділах як основа для обґрунтування технічних рішень і аналізу результатів. Детальні критерії із засобами вимірювання наведено в таблиці 1.6.

Таблиця 1.6

Критерії якості CapyDo та засоби їх вимірювання

Критерій	Складові	Засіб вимірювання
Функціональна повнота	Реєстрація, вхід, CRUD задач, керування акаунтом	Integration tests (Vitest)
Безпека	Валідація Zod, JWT, httpOnly cookie, rate limiting	Unit tests + ручна перевірка OWASP
Продуктивність	Час відповіді < 200 мс при нормальному навантаженні	Ручне вимірювання DevTools
Супровідність	TypeScript, ESLint, структурований код	Аналіз покриття (Vitest coverage)
Масштабованість	Чисте SoC, testable createApp()	Архітектурний огляд + код-ревью
Тестованість	88.53% statement coverage, 91.52% function coverage	Vitest --coverage (v8 provider)

1.7 Формалізація вимог через сценарії використання

Щоб забезпечити повноту реалізації вимоги подано у формі сценаріїв використання:

- користувач реєструється, отримує доступ до персонального простору задач;
- користувач входить у систему та переглядає актуальний перелік завдань;
- користувач створює нову задачу із заданими пріоритетом і терміном;
- користувач редагує параметри задачі відповідно до змін у планах;
- користувач видаляє неактуальні записи і підтримує чистоту робочого списку.

Кожен сценарій має пряме відображення в маршрутах, сервісних методах і тестах, що забезпечує простежуваність між вимогами та кодом.

Узагальнену модель сценаріїв використання наведено на рисунку 1.1.

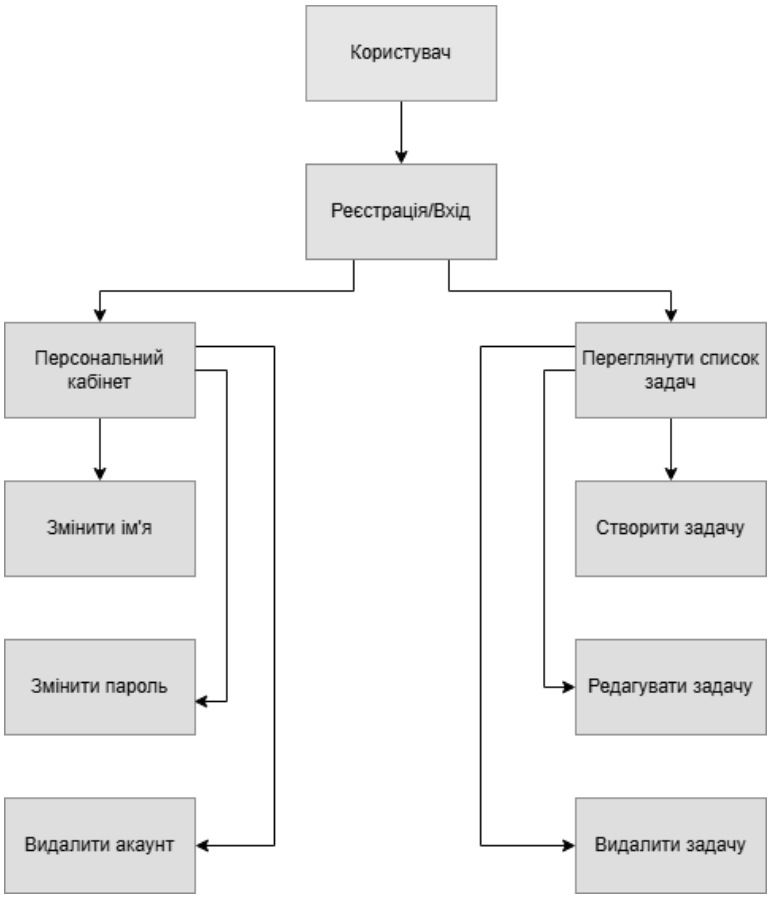


Рис. 1.1 Узагальнена use-case модель користувача CapyDo

Діаграма демонструє узагальнену модель взаємодії користувача з вебзастосунком CapyDo на рівні основних функціональних ролей. Центральним

актором є користувач, який виконує як публічні сценарії (реєстрація, вхід), так і приватні сценарії після автентифікації (робота з профілем і задачами). Діаграма підтверджує, що функціональне ядро системи охоплює повний життєвий цикл персональної задачі: створення, перегляд, редагування та видалення. Окремо відображено операції керування обліковим записом, що підкреслює завершеність користувацького контуру. Модель варіантів використання слугує трасувальним рівнем між вимогами розділу 1 і реалізацією маршрутів у розділі 4.

1.8 Методи дослідження та підходи до розробки

У роботі використано такі загальнонаукові та спеціальні методи:

Аналіз і синтез. Систематизація функціональних вимог і технологічних рішень дозволила сформувати цілісну архітектурну модель застосунку. Синтез виявлених вимог до структури даних, безпеки та інтерфейсу перетворився на конкретні технічні специфікації і схеми.

Порівняльний аналіз. Дослідження аналогів: Todoist, Trello, Notion – визначило ключові напрями диференціації CapyDo, а саме, відкритість реалізації, контроль над даними, наявність автоматизованого тестування.

Структурне проектування. Для визначення компонентів системи застосовано ієрархічне розбиття на шари: маршрути, сервіси, проміжне програмне забезпечення (middleware), база даних. Такий підхід відповідає принципам розділення відповідальностей (SoC) і спрощує підтримку.

Тестово-орієнтована перевірка якості. Для кожного сервісного методу та кожного маршруту написані unit- та integration-тести ще до реалізації або паралельно з нею. Це забезпечило раннє виявлення дефектів і знизило вартість їх усунення.

Статичний аналіз типів. Завдяки TypeScript перевірка коректності типів інтерфейсів, параметрів функцій і повернених значень виконується на етапі компіляції, а не під час виконання.

Моделювання сценаріїв використання. Для формалізації взаємодії між користувачем і системою застосовано UML-діаграми варіантів використання та діаграми послідовності. Це дозволило виявити граничні сценарії ще на етапі проектування та скоротити кількість змін під час реалізації.

Поєднання зазначених методів забезпечило системний і відтворюваний процес розробки, придатний для аудиту та повторного використання.

1.9 Методологія розробки програмного забезпечення

Розробку CapyDo виконано з дотриманням ітераційного підходу, що дозволяє поетапно нарощувати функціональність і отримувати зворотний зв'язок щодо кожного окремого модуля.

Загальна структура ітерацій:

1. Базовий сервер Hono + статичні сторінки + структура маршрутів.
2. Підключення PostgreSQL через Prisma, схема даних, CRUD для користувача.
3. Автентифікація (JWT, bcrypt, cookie), захищені маршрути.
4. CRUD для задач із валідацією і контролем доступу.
5. Middleware (rate limit, secure headers, кешування шаблонів).
6. Unit- та integration-тести, вимірювання покриття.
7. Доопрацювання UI, рефакторинг, документування.

Такий підхід відповідає принципам гнучкої розробки (agile), де кожна ітерація завершується працездатним та протестованим результатом. Це суттєво знижує ризик виявлення системних дефектів на фінальних етапах роботи.

У проекті застосовано низку технічних практик щодо організації роботи з кодом. Забезпечено чіткий поділ відповідальностей між шарами, завдяки чому маршрути не містять бізнес-логіки. Використання інтерфейсів і типів TypeScript забезпечує явний контракт між модулями. Централізована обробка помилок реалізована через AppError і handleHttpError, а вимірювання тестового покриття після кожної ітерації підтверджує повноту перевірки.

2. ВИБІР ТЕХНОЛОГІЧНОГО СТЕКУ

2.1 Обґрунтування вибору технологій

Вибір технологічного стеку для CapyDo ґрунтувався на кількох критеріях: безпека типів, продуктивність, зрілість екосистеми, доступність документації та відповідність освітнім цілям роботи.

TypeScript 5.8.3 забезпечує статичну типізацію та дозволяє виявляти помилки на етапі компіляції, а не в процесі виконання. Типи User, Task, CreateUserRequest, UpdateTaskRequest та інші явно визначені у директорії interfaces/, що гарантує узгодженість між шарами застосунку. TypeScript є стандартом де-факто для сучасного backend-розробки на Node.js [4].

Hono 4.8.3 – легкий веб-фреймворк, сумісний з Node.js, Deno та Cloudflare Workers. На відміну від Express, Hono має вбудовану підтримку JWT-middleware, helmet middleware, compress і TypeScript-генериків для контексту маршрутів. Це дозволяє писати типобезпечні обробники (c.get("user") повертає типізований об'єкт User) [5].

PostgreSQL 17 – реляційна СУБД із підтримкою транзакцій, складних індексів і зовнішніх ключів. PostgreSQL обрано через надійну репутацію, широку підтримку в хмарних провайдерів і нативну інтеграцію з Prisma ORM. Для CapyDo задіяно унікальні складені індекси та каскадне видалення [6].

Prisma 7.8.0 – ORM нового покоління, що генерує TypeScript-клієнт на основі схеми schema.prisma. Завдяки PrismaPg-адаптеру підтримується ефективне підключення через pg-пул. Prisma надає автозавершення, перевірку типів запитів і зручну міграційну систему [7].

EJS 3.1.10 – шаблонізатор для серверного рендерингу HTML. Обраний замість React/Vue через відповідність принципу YAGNI (You Aren't Gonna Need It): для персонального вебзастосунку без SPA-вимог серверний рендеринг є достатнім і ефективнішим рішенням [14].

Vitest 4.1.5 – фреймворк тестування з підтримкою TypeScript, ESM-модулів та вбудованим coverage-звітом. Швидкість запуску та сумісність з Hono тестовим клієнтом роблять його оптимальним вибором для unit- та integration-тестування [8].

Zod 4.x – бібліотека валідації з TypeScript-інферуванням типів. Схеми RegisterSchema, CreateTaskSchema і подібні описують обмеження даних і автоматично генерують відповідні TypeScript-типи, що виключає дублювання визначень [9].

Таблиця 2.1

Версії ключових залежностей CapyDo

Пакет	Версія	Призначення
typescript	5.8.3	Статична типізація
hono	4.8.3	HTTP-фреймворк
@prisma/client	7.8.0	ORM-клієнт для PostgreSQL
ejs	3.1.10	Серверний рендеринг шаблонів
zod	4.x	Валідація вхідних даних
bcrypt	6.0.0	Хешування паролів
vitest	4.1.5	Тестування та покриття
@hono/node-server	1.x	Node.js-адаптер для Hono
scss / sass	1.100.0x	Препроцесор стилів, компілюється у CSS

Такий стек забезпечує прозорість коду, суворе типобезпечне кодування і ефективне тестування – три основні виміри якості для сучасного серверного застосунку.

2.1.1 Порівняння Hono з альтернативними фреймворками

При виборі HTTP-фреймворку розглядалися три варіанти: Express, Fastify і Hono.

Express є найпоширенішим Node.js-фреймворком, однак не має вбудованої підтримки TypeScript і вимагає додаткових @types/express пакетів. Також Express не підтримує Edge Runtime і не має вбудованого JWT-middleware.

Fastify забезпечує вищу продуктивність, ніж Express, завдяки schema-based serialization, але має складніший API для маршрутизації і plugin-систему, яка надлишкова для невеликих застосунків.

Hono поєднує простий API (схожий на Express), вбудовані TypeScript-генерики, підтримку JWT, secureHeaders і compress без додаткових пакетів. Для масштабу CapyDo це оптимальний вибір.

Таблиця 2.2

Порівняння backend-фреймворків

Критерій	Express	Fastify	Hono
Вбудована орієнтація на TS	Низька	Середня	Висока
Продуктивність	Середня	Висока	Висока
Простота старту для малого проєкту	Висока	Середня	Висока
Edge/мультирантайм сумісність	Низька	Низька	Висока
Вбудовані security middleware	Низька	Середня	Висока
Підсумкова відповідність CapyDo	Середня	Висока	Найвища

Практична перевага Hono проявляється на рівні типізації контексту маршруту. У Express обробник приймає параметри типу Request і Response без прив'язки до конкретних змінних контексту, що вимагає явних приведень типів. У Hono змінні контексту оголошуються через TypeScript-дженерик Hono<{ Variables: Variables }>, де Variables — інтерфейс із точними типами. c.get("user") повертає типізований об'єкт User без приведень — помилки звернення до неіснуючих змінних виявляються на етапі компіляції, а не в runtime.

2.1.2 Обґрунтування вибору PostgreSQL замість MongoDB

Реляційна модель даних у PostgreSQL обрана свідомо – структура задач чітко визначена, зв'язок між users і task є строго 1:N і не потребує гнучкості документно-орієнтованих БД. PostgreSQL забезпечує сувору цілісність даних через зовнішні ключі та NOT NULL-обмеження, каскадне видалення задач при видаленні користувача (onDelete: Cascade), ефективну індексацію за userId і dateCreated для відсортованих вибірок, а також підтримку унікальних складених індексів (userId, name) для запобігання дублям.

Таблиця 2.3

Порівняння СУБД для зберігання даних CapyDo

Критерій	PostgreSQL 17	MongoDB 7	SQLite 3
Строга схема та типи даних	Так	Ні	Часткова
Зовнішні ключі та CASCADE	Так	Ні	Обмежено
Транзакційна цілісність	ACID (повна)	Часткова	ACID
Підтримка в Prisma ORM	Нативна	Є	Є
Підходить для продакшену	Так	Так	Ні (embedded)
Підсумкова відповідність	Найвища	Середня	Низька

2.1.3 Обґрунтування вибору bcrypt для хешування паролів

Для безпечного зберігання паролів у CapyDo обрано бібліотеку `bcrypt` версії 6.0.0. Bcrypt є адаптивним алгоритмом хешування: завдяки параметру cost factor (у проєкті – 10) кожен пароль поєднується з автоматично згенерованою сіллю, що робить неможливими атаки з заздалегідь обчисленими хешами (rainbow table). Зі збільшенням обчислювальних потужностей параметр складності можна підвищити без зміни архітектури.

Порівняно з альтернативами bcrypt забезпечує оптимальний баланс безпеки і продуктивності для застосунку з невисоким навантаженням на реєстрацію:

- SHA-256/MD5 – швидкі алгоритми без солі, непридатні для паролів через вразливість до dictionary та brute-force атак;
- Argon2 – переможець PHC (Password Hashing Competition) 2015 року, рекомендований OWASP; потребує налаштування параметрів пам'яті і паралелізму, що ускладнює конфігурацію в порівнянні з bcrypt;
- PBKDF2 – стандарт NIST, підходить для compliance-середовищ, але менш зручний у Node.js-екосистемі.

Bcrypt є стандартом де-факто для Node.js-застосунків, добре задокументований і нативно підтримується через пакет `node.bcrypt.js` [12].

У реалізації CapyDo функція bcrypt.hash(password, 10) викликається як під час реєстрації, так і під час зміни пароля в налаштуваннях акаунту. Порівняння

виконується через bcrypt.compare() без доступу до оригінального рядка, що виключає можливість витоку пароля навіть за умови компрометації сервісного шару.

2.1.4 Обґрунтування вибору JWT для автентифікації

Для автентифікації у CapyDo обрано JSON Web Token (JWT) відповідно до специфікації RFC 7519 [8]. JWT дозволяє реалізувати stateless автентифікацію: сервер не зберігає стан сесії – вся необхідна інформація міститься у підписаному токені. Токен зберігається в httpOnly cookie з атрибутами sameSite=strict і secure у production-режимі, що виключає доступ до токена з JavaScript-коду браузера і захищає від XSS- та CSRF-атак.

Порівняно з альтернативними підходами JWT є найбільш доцільним вибором для CapyDo. Серверні сесії (session store) потребують зовнішнього сховища – бази даних або кешу – для збереження стану сесій, тоді як JWT-підхід не вимагає додаткової інфраструктури. OAuth 2.0 орієнтований на системи зі стороннім провайдером автентифікації і є надлишковим для персонального застосунку з власним обліком. Підхід Cookie + HMAC є простішим, але менш стандартизованим: JWT натомість забезпечує стандартизований формат із полями iss, aud, exp, jti та перевірений інструментарій.

Токен CapyDo містить поля userId, jti (унікальний ідентифікатор для майбутньої інвалідації), iat, exp, iss та aud, підписаний алгоритмом HS256 із секретним ключем зі змінної середовища.

2.1.5 Порівняння бібліотек валідації: Zod vs Joi vs Yup

Для валідації вхідних даних обрано Zod 4.x – бібліотеку зі статичним TypeScript-інферуванням типів. Ключова перевага: схема Zod одночасно є джерелом правди для валідації та TypeScript-типів, що усуває дублювання визначень.

Таблиця 2.4

Порівняння бібліотек валідації

Критерій	Zod 4.x	Joi 17.x	Yup 1.x
TypeScript-інферування типу	Вбудоване	Часткове	Є
Підтримка ESM	Повна	Обмежена	Є
Розмір бандлу	Малий	Середній	Малий
Підтримка async-валідації	Є	Є	Є
Інтеграція з Hono	Нативна	Ручна	Ручна
Підсумкова відповідність	Найвища	Середня	Середня

Zod 4.x обрано завдяки нативній інтеграції з TypeScript і Hono, а також через підтримку ESM-модульної системи, яку використовує весь проєкт CapyDo.

2.1.6 Обґрунтування вибору Vitest для тестування

Для автоматизованого тестування обрано Vitest 4.1.5 – фреймворк тестування, розроблений з урахуванням TypeScript і ESM-модулів. Vitest сумісний з API Jest, що знижує поріг входу, але, на відміну від Jest, не потребує Babel-транспіляції для TypeScript і ESM.

Hono надає тестовий клієнт `hono/testing`, який дозволяє надсилати HTTP-запити безпосередньо до Hono-застосунку без запуску реального сервера. Vitest повністю підтримує цей підхід завдяки нативній ESM-підтримці.

Переваги Vitest над Jest для CapyDo:

- нативна підтримка TypeScript без додаткової конфігурації;
- вбудований coverage-звіт через провайдер v8 (без istanbul-плагінів);
- швидший запуск тестів завдяки Vite-трансформації;
- watch-режим з інкрементальними перезапусками для розробки.

3. ПРОЕКТУВАННЯ СИСТЕМИ

3.1 Архітектура системи

Архітектура CapyDo реалізована за клієнт-серверною моделлю. Браузер надсилає HTTP-запити і отримує готові HTML-сторінки, сформовані на сервері. Такий підхід (SSR – server-side rendering) спрощує безпекову модель: усі токени зберігаються в httpOnly cookie й недоступні з клієнтського JavaScript.

Загальна схема архітектури зображена на рисунку 3.1.

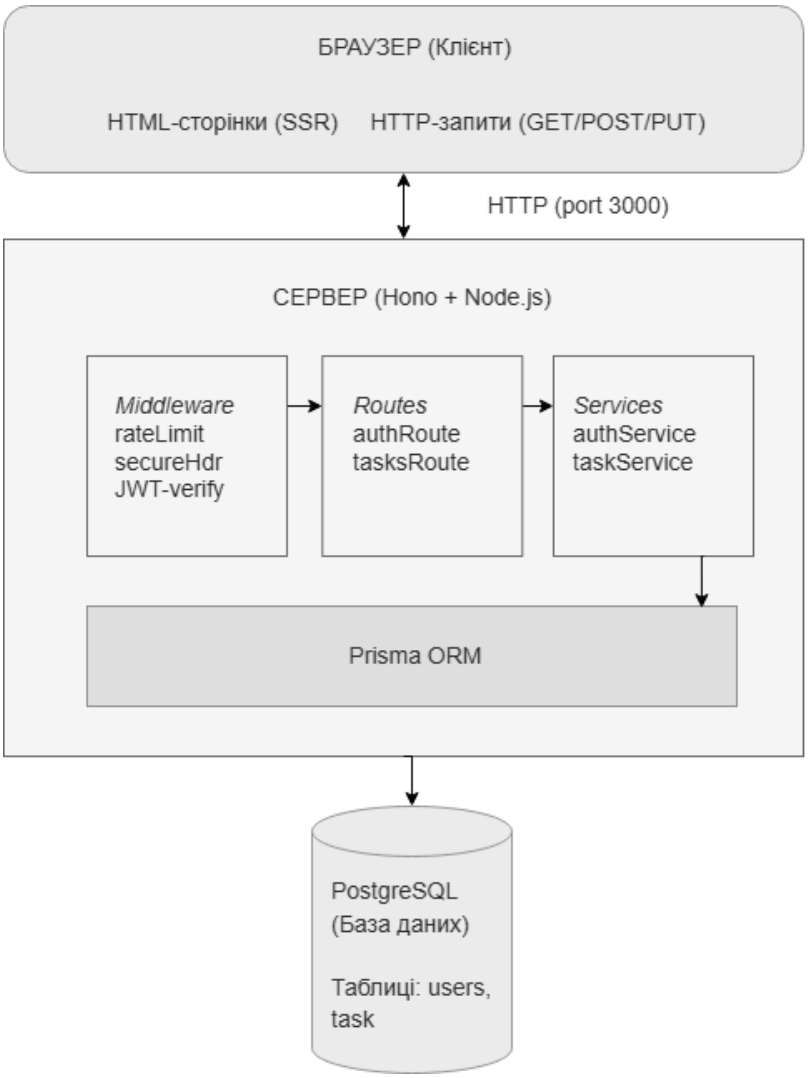


Рис. 3.1 Загальна архітектура вебзастосунку CapyDo

Діаграма відображає клієнт-серверну архітектуру CapyDo із чітким розподілом відповідальностей між шарами. Клієнтський браузер взаємодіє із

сервером через HTTP-запити, отримуючи SSR-сторінки та службові відповіді. На серверному рівні запит послідовно проходить middleware-шар, маршрутизацію, сервісну логіку і рівень доступу до даних через Prisma ORM.

Така композиція зменшує зв'язаність компонентів і спрощує супровід: зміни у валідації або безпеці ізольовані від бізнес-логіки, а зміни сервісів не потребують перебудови всього транспортного шару. Діаграма підтверджує відповідність архітектури принципам SoC і підтримує вимоги щодо безпеки та масштабованості.

Логічні шари системи детально описано в таблиці 3.1.

Таблиця 3.1

Логічні шари архітектури CapyDo

Шар	Файли/модулі	Відповідальність
Routes	authRoute.ts, tasksRoute.ts	Обробка HTTP-маршрутів, формування відповіді
Services	authService.ts, taskService.ts	Бізнес-логіка без HTTP-залежностей
Database	client.ts, schema.prisma	Prisma Client + схема даних; PostgreSQL
Middleware	rateLimit, validation, errorHandler	Валідація, rate limit, обробка помилок
Lib	render.ts, session.ts	EJS-рендер, JWT create/verify
Public/Views	_.ejs, _.css	Шаблони сторінок і статичні ресурси

3.2 Проектування бази даних

Схема включає дві сутності: users і task. ER-діаграма зображена на рисунку 3.2 подає реляційну модель даних CapyDo, у якій сутність users пов'язана із сутністю task відношенням один-до-багатьох. Кожна задача має посилання на власника через зовнішній ключ userId, що забезпечує логічну належність записів конкретному користувачу.

Модель передбачає каскадне видалення пов'язаних задач при видаленні облікового запису, завдяки чому виключається поява «осиротілих» записів. Для підвищення продуктивності вибірок передбачено індексацію за userId та складену індексацію для впорядкованого відображення задач. Унікальність логіна користувача та обмеження на рівні таблиць формують базовий рівень цілісності даних і підтримують коректну роботу CRUD-операцій.

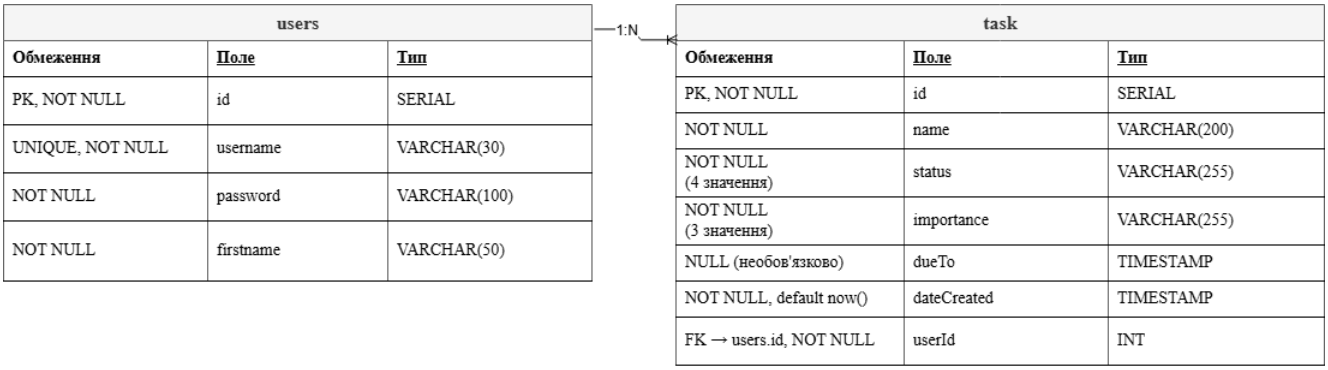


Рис. 3.2 ER-діаграма бази даних CapyDo

3.2.1 Логічна структура таблиць

Повний опис полів кожної таблиці наведений у таблицях 3.2 і 3.3.

Таблиця 3.2

Логічна структура таблиці users

Поле	Тип	Обмеження	Опис
id	SERIAL	PK, NOT NULL	Унікальний ідентифікатор
username	VARCHAR(30)	NOT NULL, UNIQUE	Логін користувача
password	VARCHAR(100)	NOT NULL	bcrypt-хеш паролю
firstname	VARCHAR(50)	NOT NULL	Ім'я користувача

Таблиця 3.3

Логічна структура таблиці task

Поле	Тип	Обмеження	Опис
id	SERIAL	PK, NOT NULL	Унікальний ідентифікатор
name	VARCHAR(200)	NOT NULL	Назва задачі
status	VARCHAR(255)	NOT NULL, default	not-started / in-progress / done / cancelled
importance	VARCHAR(255)	NOT NULL, default	low / medium / high
dueTo	TIMESTAMP	NULL (необов'язково)	Дедлайн виконання
dateCreated	TIMESTAMP	NOT NULL, default now()	Дата створення
userId	INT	NOT NULL, FK → users	Ідентифікатор власника

3.2.2 Індекси бази даних

У таблиці task реалізовано три індекси для забезпечення ефективності запитів. Склад індексів наведено в таблиці 3.4.

Таблиця 3.4

Індекси таблиці task

Назва індексу	Поля	Тип	Мета
idx_task_user	userId	Звичайний	Швидкий доступ до задач конкретного користувача
idx_task_user_date	userId, dateCreated DESC	Складений	Вибірка з сортуванням без додаткового ORDER BY
idx_task_user_name	userId, name	Унікальний	Запобігання дублікації назв задач

Схема включає дві сутності: users із полями id (SERIAL, PK), username (унікальний), password (bcrypt-хеш) і firstname, а також task із полями id (SERIAL, PK), name, status (VARCHAR), importance (VARCHAR), dueTo (необов'язкова дата), dateCreated та userId (FK). Зв'язок – один користувач може мати багато задач (1:N). Зовнішній ключ userId з каскадним видаленням гарантує, що при видаленні облікового запису всі його задачі теж видаляються.

Реалізовані обмеження та оптимізація охоплюють унікальний індекс на пару (userId, name), що запобігає дублюванню назв задач у межах одного користувача, індекс task(userId) для швидкого доступу до списку задач, складений індекс task(userId, dateCreated DESC) для впорядкованих вибірок без додаткового сортування та каскадне видалення задач при видаленні користувача.

3.3 Проектування маршрутів

Таблиця 3.5 містить повний перелік маршрутів застосунку з методами HTTP та рівнями доступу.

Таблиця 3.5

Маршрути вебзастосунку CapyDo

Метод	Маршрут	Опис	Доступ
GET	/	Головна сторінка	Публічний
GET	/about	Про проєкт	Публічний
GET	/contact	Контакти	Публічний
GET	/signup	Форма реєстрації	Публічний
POST	/signup	Обробка реєстрації	Публічний
GET	/login	Форма входу	Публічний
POST	/login	Обробка входу	Публічний
GET	/:username/account	Сторінка профілю	Приватний
GET	/:username/account-settings	Налаштування акаунту	Приватний
POST	/:username/account-settings	Оновлення даних акаунту	Приватний
DELETE	/:username/account	Видалення акаунту	Приватний
POST	/:username/account	Вихід із системи	Приватний
GET	/:username/tasks	Список задач	Приватний
GET	/:username/tasks/create	Форма створення задачі	Приватний
POST	/:username/tasks/create	Обробка створення задачі	Приватний
GET	/:username/tasks/:taskId	Форма редагування задачі	Приватний
PUT	/:username/tasks/:taskId	Оновлення задачі	Приватний
DELETE	/:username/tasks/:taskId	Видалення задачі	Приватний

У системі застосовано подвійне представлення частини маршрутів для сумісності з HTML-формами. Оскільки стандартні HTML-форми підтримують переважно GET/POST, для дій оновлення/видалення реалізовано POST-варіанти там, де це потрібно для браузерної взаємодії, а також окремі PUT/DELETE маршрути для семантично коректних HTTP-операцій.

Додатково до переліку маршрутів слід зафіксувати такі правила:

1. Для оновлення профілю підтримуються обидва методи: PUT і POST на /:username/account-settings.
2. Для виходу з акаунту використовується POST на: /:username/account.
3. Для видалення акаунту використовується DELETE на /:username/account.
4. Для оновлення задачі використовується PUT на /:username/tasks/:taskId.
5. Для видалення задачі використовується DELETE на /:username/tasks/:taskId.

Контроль доступу до приватних маршрутів реалізований через двоетапну перевірку. На першому етапі глобальний JWT-middleware перевіряє наявність і валідність токена в httpOnly cookie. Якщо токен відсутній або прострочений – запит

перенаправляється на /login з кодом 302. На другому етапі, вже всередині обробника маршруту, виконується порівняння username з URL-параметра з полем username авторизованого користувача з JWT-payload. Це унеможливлює горизонтальну ескалацію доступу: автентифікований користувач не зможе отримати доступ до маршруту іншого користувача, навіть маючи дійсний токен.

Оскільки стандарт HTML не підтримує методи PUT і DELETE у атрибуті method тега <form>, у CapyDo реалізовано подвійне представлення для частини маршрутів: POST-варіант для браузерної взаємодії та PUT/DELETE для семантично коректних HTTP-операцій.

Маршрути згруповані в два файли: authRoute.ts охоплює все, що стосується облікового запису, а tasksRoute.ts – повний CRUD для задач.

3.4 Модель безпеки

У системах захисту прийнято розрізняти два ключові поняття: автентифікація – процес перевірки особи користувача (хто він?), та авторизація – процес перевірки прав доступу до конкретного ресурсу (що йому дозволено?). У CapyDo реалізовано обидва рівні.

Автентифікація реалізована через JWT-токен, що видається після успішного входу та зберігається в httpOnly cookie. При кожному запиті глобальний middleware перевіряє підпис токена, його термін дії та обов'язкові поля jti та userId. Пароль користувача зберігається виключно у вигляді bcrypt-хешу – оригінальний пароль ніде не зберігається.

Авторизація реалізована на двох незалежних рівнях: перевірка збігу username з URL-параметром та фільтрація всіх операцій над задачами за userId у запитах до бази даних. Навіть якщо автентифікований користувач сформує запит на ресурс іншого користувача, обидві перевірки заблокують доступ незалежно одна від одної.

У проекті реалізовано JWT access token з контрольними полями jti та userId, зберігання токена в cookie з атрибутами httpOnly та sameSite=strict, bcrypt-хешування паролів, валідацію всіх форм через Zod-схеми, rate limiting з

поверненням Retry-After та X-RateLimit-* заголовків, а також helmet middleware і стиснення відповідей з урахуванням практик OWASP.

Таблиця 3.6

Матриця загроз і контрзаходів

Загроза	Рівень	Контрзахід у CapyDo
Підбір паролів (bruteforce)	Високий	Rate limiting + вимоги до складності пароля
Викрадення токена через JS	Високий	httpOnly cookie + sameSite=strict
Горизонтальна ескалація доступу	Високий	Перевірка username + фільтр userId у CRUD-запитах
Некоректні/шкідливі вхідні дані	Середній	Zod-валідація на всіх критичних формах
Витік внутрішніх деталей помилок	Середній	Централізований errorHandler (санітизовані відповіді)
Перевантаження сервісу запитами	Середній	Ліміти запитів, контрольні заголовки Retry-After

Варто зазначити, що перелічені контрзаходи є незалежними: відключення або обхід одного рівня не компрометує решту. Наприклад, навіть якщо зловмисник сформує запит із дійсним JWT-токеном, але з підробленим username в URL — перевірка на рівні маршруту поверне перенаправлення до власного профілю. Навіть якщо обидва попередні рівні було б обійдено, фільтр userId у запиті до бази даних не дозволить отримати або змінити дані іншого користувача.

3.5 Проектування функціональної моделі системи

Функціональна модель передбачає два стани користувача: неавторизований і авторизований. Для неавторизованого стану доступні публічні сторінки, реєстрація та вхід. Для авторизованого стану відкриваються персональні маршрути керування профілем і задачами.

У проектуванні застосовано принцип мінімально необхідних прав доступу: будь-яка операція над персональним ресурсом виконується лише після перевірки

належності цього ресурсу поточному користувачу. Це знижує ризик горизонтальної ескалації доступу.

3.5.1 Діаграми послідовності

Для критичних сценаріїв (реєстрація, вхід, CRUD-операції) діаграми послідовності наведено в розділі 4 (Рисунки 4.1 – 4.4). Винесення цих діаграм до розділу реалізації забезпечує прямий зв'язок між моделлю взаємодії та фактичною серверною логікою.

3.6 Проектування даних і забезпечення цілісності

Модель даних орієнтована на уникнення дублювання і підтримку цілісності даних. Дані користувача винесено в окрему таблицю users, тоді як дані задач зберігаються у таблиці task із зовнішнім ключем userId. Каскадне видалення пов'язаних задач запобігає залишенню «осиротілих» записів, а індексація userId і dateCreated підтримує ефективні вибірки. Така структура є основою для подальшого розширення – додавання тегів, категорій, нагадувань – без порушення базової логіки.

Крім того, поля status і importance реалізовано як рядкові типи (VARCHAR), а допустимі значення контролюються валідацією на рівні застосунку (Zod). Такий підхід забезпечує гнучкість моделі даних і водночас не допускає збереження невалідних значень. Детальні правила цілісності для кожної таблиці наведено в таблиці 3.7.

Таблиця 3.7

Правила цілісності даних

Таблиця	Поле	Обмеження	Механізм
users	id	PRIMARY KEY, унікальний	SERIAL / autoincrement()
users	username	UNIQUE, NOT NULL	Унікальний індекс (@unique)
users	firstname	NOT NULL	Обмеження схеми
users	password	NOT NULL	Обмеження схеми + bcrypt-хешування на рівні застосунку
task	id	PRIMARY KEY, унікальний	SERIAL / autoincrement()
task	userId	FOREIGN KEY – users(id), ON DELETE CASCADE	Зовнішній ключ із каскадним видаленням
task	(userId, name)	UNIQUE (у межах користувача)	Складений унікальний індекс
task	dueTo	NULL дозволено (необов’язкове поле)	Тип DATE, nullable
task	status	NOT NULL	VARCHAR; допустимі значення контролюються валідацією застосунку (Zod)
task	importance	NOT NULL	VARCHAR; допустимі значення контролюються валідацією застосунку (Zod)
task	dateCreated	NOT NULL, default now()	Значення за замовчуванням на рівні БД

3.7 Алгоритм обробки HTTP-запиту

Алгоритм проходження HTTP-запиту через основні шари вебзастосунку CapyDo зображено на рисунку 3.4. Мета цього алгоритму полягає в тому, щоб формалізувати послідовність дій системи від моменту отримання запиту до формування відповіді клієнту. На відміну від опису окремих модулів у попередніх підрозділах , тут розглядається наскрізний сценарій їхньої взаємодії в межах одного запиту.

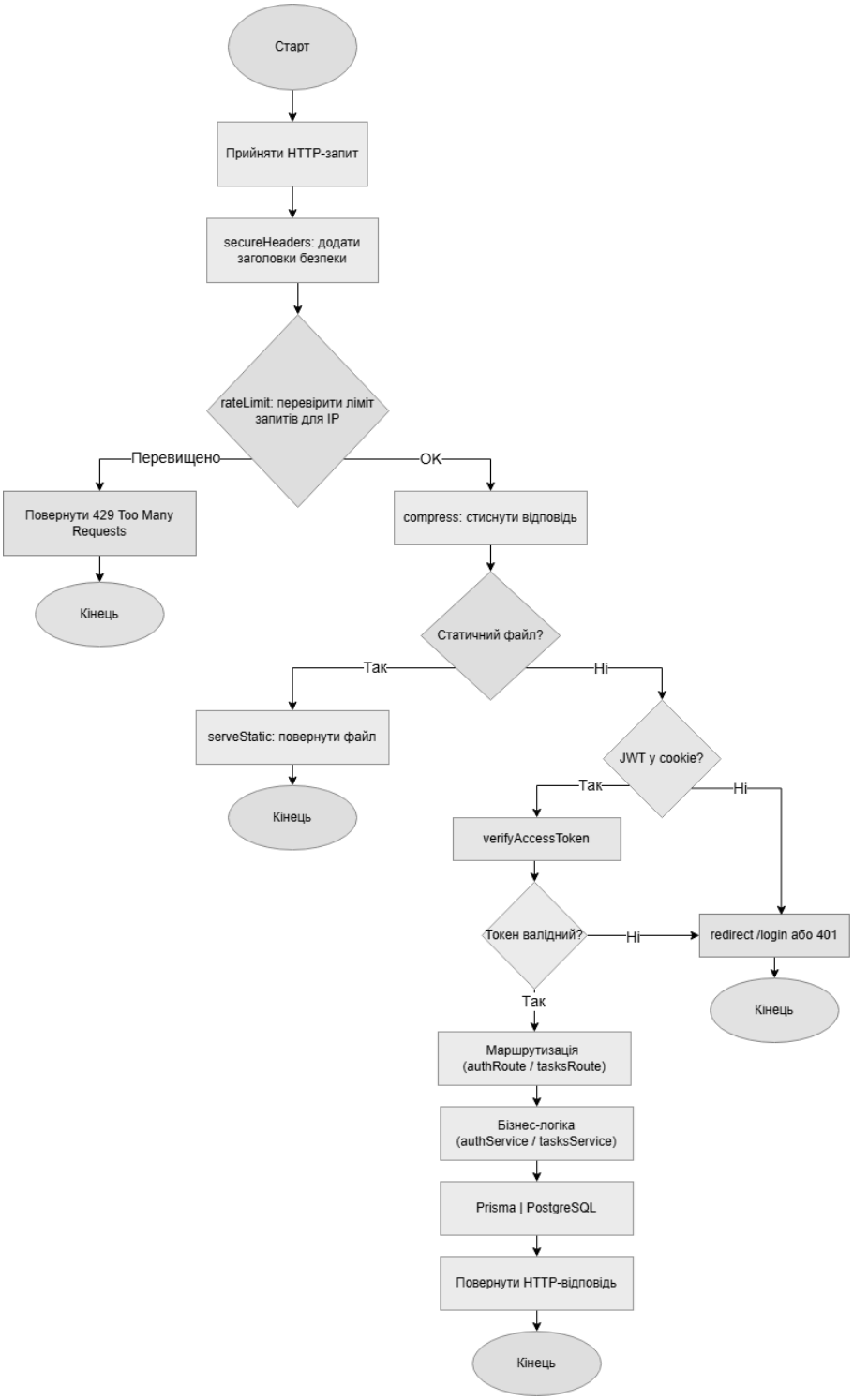


Рис. 3.4 Алгоритм обробки HTTP-запиту у CapyDo

Алгоритм відображає реальний конвеєр обробки: початкові middleware безпеки, перевірку автентифікації, маршрутизацію, виконання бізнес-логіки, звернення до бази даних і повернення результату у форматі HTML/JSON або через

перенаправлення. Такий підхід дозволяє показати не лише перелік компонентів системи, а й їхній порядок та роль у загальному циклі обробки.

Покроковий опис алгоритму наведено в таблиці 3.8.

Таблиця 3.8

Покроковий алгоритм обробки HTTP-запиту

Крок	Дія	Відповідальний модуль
1	Прийняття TCP-з'єднання і HTTP-запиту	Node.js HTTP-сервер
2	Додавання secure headers	helmet middleware
3	Перевірка rate limit (IP-лічильник)	rateLimit middleware
4	Стиснення відповіді (gzip/br)	compress middleware
5	Верифікація JWT-токена (з cookie)	Global JWT middleware
6	Статичні файли – повернути одразу	serveStatic middleware
7	Маршрутизація до потрібного маршруту	authRoute / tasksRoute
8	Бізнес-логіка (CRUD, автентифікація)	authService / taskService
9	Отримання даних з PostgreSQL	Prisma Client
10	Рендеринг EJS-шаблону / перенаправлення	render.ts / c.redirect()
11	Повернення HTTP-відповіді клієнту	Hono response pipeline

Стандартизований алгоритм спрощує супровід і дозволяє прогнозовано керувати поведінкою системи. Кожен крок реалізований у відповідному middleware-модулі та покритий тестами.

3.8 Структура файлів проекту

Структура проекту CapyDo побудована за модульним принципом і відображає логічний поділ на конфігураційні файли, прикладну логіку, інтерфейс користувача та компоненти роботи з даними. На кореневому рівні розміщено службові файли налаштування проєкту, зокрема package.json, tsconfig.json і змінні середовища .env. Тут також знаходиться директорія prisma зі схемою schema.prisma, яка описує модель даних для ORM-рівня.

Для ініціалізації бази даних використовується окремий SQL-файл schema.sql. Такий підхід дає змогу розділити початкове створення таблиць у PostgreSQL і подальшу роботу Prisma Client, який генерується командою npx prisma generate. Це забезпечує прозорість схеми даних і спрощує супровід як SQL-рівня, так і ORM-частини застосунку.

Основна частина коду зосереджена у директорії src, яка поділена на кілька логічних блоків. У routes/ реалізовано обробники authRoute.ts і tasksRoute.ts, що приймають HTTP-запити, перевіряють параметри та передають керування сервісному шару. У services/ містяться authService.ts і taskService.ts, де зосереджено бізнес-логіку без прямої залежності від HTTP-контексту. Така організація спрощує unit-тестування і робить сервісні методи повторно використовуваними.

Директорія middleware/ об’єднує наскрізні механізми обробки запитів. Модуль errorHandler.ts забезпечує централізовану обробку помилок, rateLimit.ts відповідає за обмеження частоти запитів, staticCache.ts використовується для кешування статичних ресурсів, а validation.ts реалізує серверну валідацію вхідних даних. У lib/ розміщено допоміжні модулі render.ts, який відповідає за рендеринг EJS-шаблонів із TTL-кешуванням, та session.ts, де реалізовано створення, перевірку й відкликання JWT-токенів.

Для типізації предметної області використано директорію interfaces/, де зосереджено TypeScript-інтерфейси User, Task, CreateUserRequest, UpdateTaskRequest та інші допоміжні типи. Вони забезпечують узгодженість даних між різними шарами застосунку на етапі компіляції. Шаблони сторінок розміщено у views/, а статичні ресурси, зокрема скомпільовані CSS-файли та зображення, — у public. Така структура сприяє зручній навігації по проєкту, спрощує підтримку коду та створює передумови для подальшого розширення функціоналу.

4. РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ВЕБЗАСТОСУНКУ CAPYDO

4.1 Реалізація серверного ядра

Серверна частина CapyDo реалізована на основі фреймворку Hono з Node.js-адаптером і відповідає за обробку HTTP-запитів, застосування механізмів безпеки, маршрутизацію та формування відповіді клієнту. Додаток запускається на порту 3000 та підтримує коректне завершення роботи через обробку системних сигналів SIGINT і SIGTERM. Такий підхід забезпечує передбачувану поведінку сервера під час розробки та в умовах експлуатації.

Обробка запиту в системі побудована як послідовний ланцюжок middleware. Спочатку застосовуються безпекові заголовки, після чого виконується перевірка частоти запитів і стиснення відповіді. Далі глобальний механізм перевіряє наявність і валідність JWT-токена, що зберігається в cookie. Така послідовність дозволяє відсіяти небажані або потенційно шкідливі запити ще до переходу до бізнес-логіки, зменшуючи навантаження на систему та підвищуючи її стійкість.

Після проходження middleware запит передається до маршрутного рівня, де визначається відповідний обробник для публічних або приватних сценаріїв. У разі необхідності відбувається звернення до сервісного шару, який виконує операції автентифікації, керування профілем або CRUD-операції із задачами. Для доступу до даних використовується Prisma ORM, що спрощує взаємодію з PostgreSQL та підвищує типобезпечність коду.

Окремим елементом серверного ядра є рендеринг EJS-шаблонів. У production-режимі шаблони кешуються в пам’яті з TTL 5 хвилин, що зменшує кількість файлових операцій і покращує швидкодію під час повторних звернень до сторінок. Статичні ресурси, зокрема стилі та зображення, обслуговуються через serveStatic без залучення бізнес-логіки, що дозволяє чітко розмежувати обробку динамічного і статичного контенту.

Узагальнено серверне ядро CapyDo поєднує в собі механізми безпеки, маршрутизації, доступу до даних і серверного рендерингу. Така структура є компактною, прозорою для тестування і придатною для подальшого супроводу.

Вона забезпечує стабільну роботу застосунку та відповідає вимогам до клієнт-серверних вебсистем із приватним доступом до персональних даних.

4.2 Реалізація автентифікації, авторизації та облікового запису

Підсистема автентифікації та авторизації в CapyDo реалізує контроль доступу до персональних ресурсів користувача і забезпечує цілісність сесійного контексту під час кожного запиту. Після успішного входу сервер генерує JWT access-токен з терміном дії 1 година, підписує його алгоритмом HS256 і встановлює в httpOnly cookie. Такий спосіб зберігання зменшує ризик компрометації токена через клієнтський JavaScript.

Перевірка автентичності виконується глобальним middleware, який обробляє cookie auth-token, верифікує підпис токена, строк дії та наявність обов'язкових полів userId і jti. У разі невалідного токена користувач не отримує приватний контекст доступу, а система переходить до безпечної гілки обробки запиту. Додатково при кожному запиті підтверджується наявність користувача в базі даних, що унеможливлює роботу з неактуальними сесіями.

Авторизація реалізована на двох рівнях. На рівні маршруту виконується порівняння username у URL із username поточного авторизованого користувача.
На рівні доступу до даних усі операції над задачами виконуються з обов'язковою фільтрацією за userId. Поєднання цих перевірок захищає від горизонтальної ескалації доступу, навіть якщо користувач намагається вручну підмінити ідентифікатор у маршруті.

Послідовність реєстрації нового користувача зображена на рисунку 4.1, а послідовність входу в систему – на рисунку 4.2. У сценарії реєстрації дані форми проходять валідацію, перевірку унікальності username, після чого пароль хешується через bcrypt і створюється новий запис користувача. Для сценарію входу виконується перевірка облікових даних, створення сесійного токена та перенаправлення до персонального профілю.

Оновлення облікового запису підтримує зміну firstname і, за потреби, зміну пароля. Якщо новий пароль передано, він повторно проходить валідацію і зберігається лише у вигляді bcrypt-хешу. Вихід з акаунту виконується через очищення cookie auth-token. Виклик revokeAccessToken() збережено як підготовчий інтерфейс для майбутньої серверної моделі відкликання токенів; у поточній реалізації фактичне завершення сесії забезпечується саме видаленням cookie.

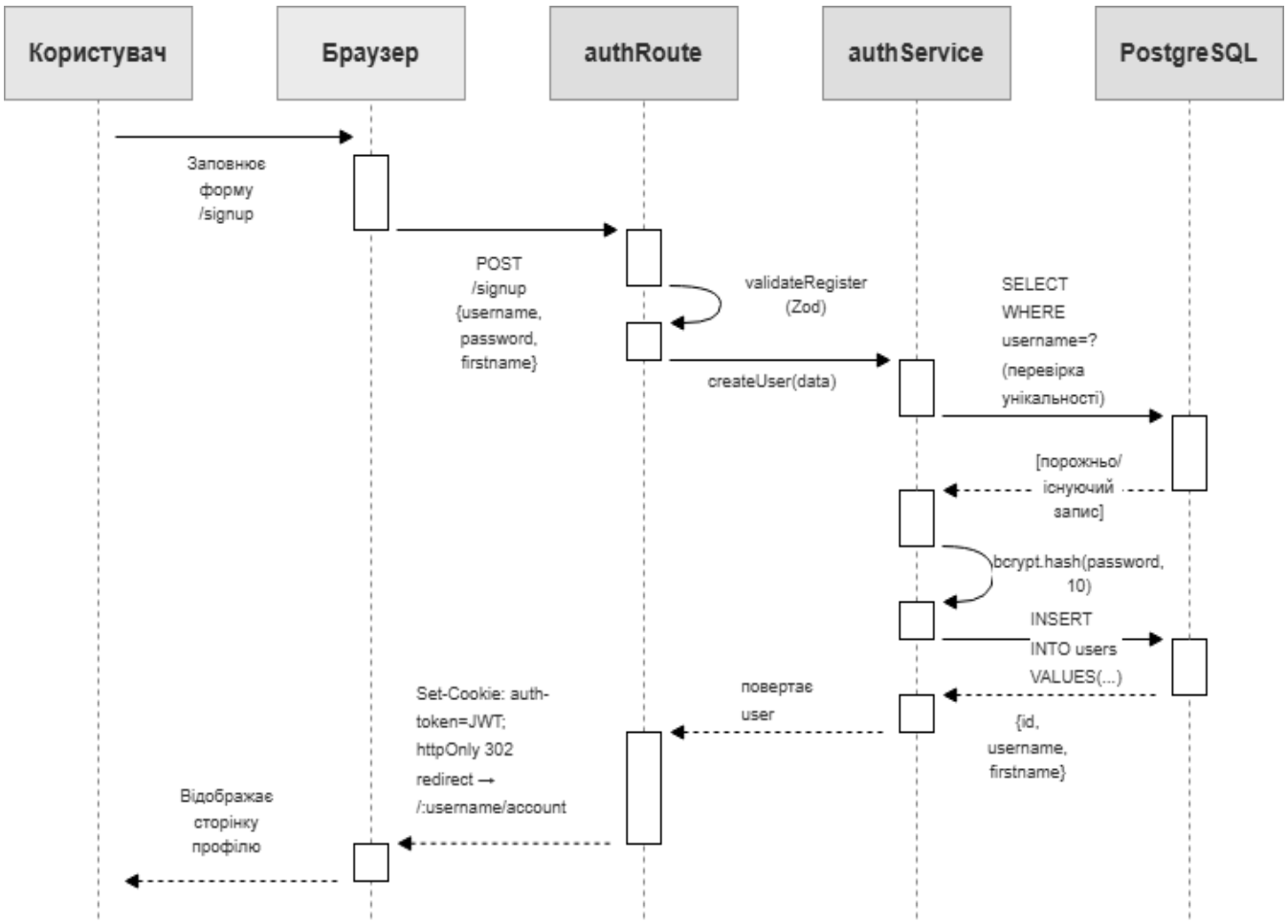


Рис. 4.1 Діаграма послідовності реєстрації користувача

Послідовність входу в систему зображена на рисунку 4.2.

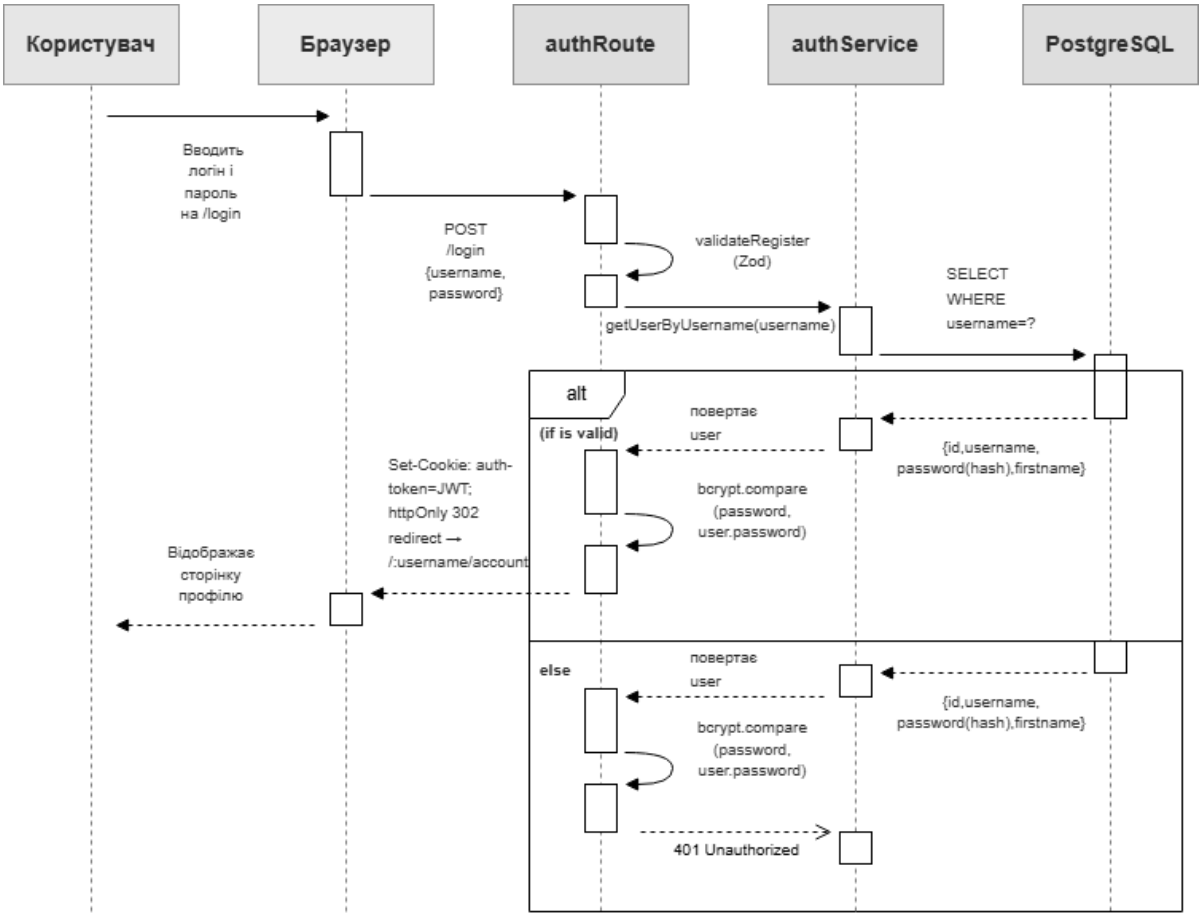


Рис. 4.2 Діаграма послідовності входу в систему

Реєстрація виконується у 5 кроків:

1. Приймання даних форми.
2. Валідація полів username/password/firstname через Zod.
3. Перевірка унікальності username в БД.
4. Хешування пароля через bcrypt з автоматичним salt-генеруванням.
5. Створення користувача в БД і видача JWT у httpOnly cookie.

Оновлення акаунту у 3 кроки:

1. Зміна firstname та пароля з повторною валідацією.
2. Перевірка відповідності username в URL поточному авторизованому користувачу.
3. При зміні пароля – обов'язкове хешування перед збереженням.

Вихід з акаунту у 2 кроки:

1. Очищення cookie.
2. Виклик revokeAccessToken() як підготовчого інтерфейсу для майбутнього server-side blacklist токенів.

У поточній реалізації revokeAccessToken() не змінює стан серверного сховища відкликаних токенів, тому фактичне завершення сесії забезпечується саме видаленням cookie auth-token. Після цього браузер не надсилає токен у наступних запитах, а користувач вважається неавтентифікованим.

4.3 Реалізація керування задачами

Підсистема керування задачами реалізована через окремий сервісний шар, що забезпечує ізоляцію бізнес-логіки від HTTP-рівня і спрощує супровід коду.

Основні операції охоплюють повний CRUD-цикл: отримання списку задач, створення нового запису, оновлення параметрів і видалення задачі.

Для перегляду використовується вибірка за userId з сортуванням за dateCreated у спадному порядку, що забезпечує відображення найновіших задач на початку списку. Створення задачі виконується після серверної валідації полів і завжди супроводжується прив'язкою запису до поточного користувача. Дедлайн залишається необов'язковим параметром, що відповідає персональному характеру застосунку.

Оновлення і видалення реалізовано з обов'язковою фільтрацією за парою ідентифікаторів id і userId. Така умова виключає зміну або видалення чужих даних навіть у випадку ручної підміни taskId у URL. Отже, контроль належності ресурсу виконується не лише на рівні маршруту, а й безпосередньо на рівні операцій із базою даних.

Послідовність виконання CRUD-операцій над задачами наведено на рисунку 4.3, а деталізацію методів сервісного шару подано в таблиці 4.2. Запропонована організація узгоджується з рекомендаціями OWASP щодо зниження ризику IDOR-атак (Insecure Direct Object Reference), оскільки кожна модифікуюча операція підтверджує належність даних автентифікованому користувачу.

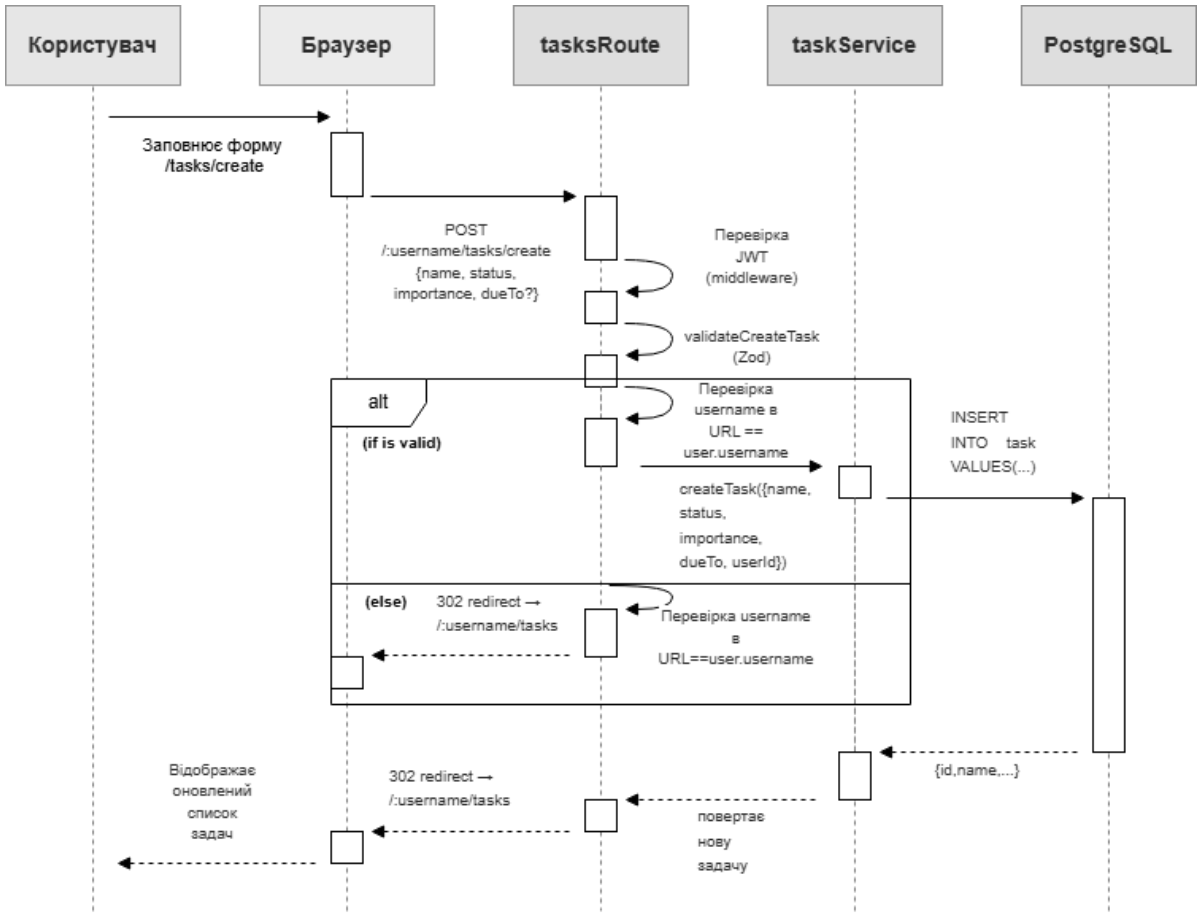


Рис. 4.3 Діаграма послідовності CRUD-операцій із задачами

Детальний опис CRUD-операцій наведено в таблиці 4.2.

Таблиця 4.2

Опис CRUD-операцій taskService

Метод	HTTP	Маршрут	Призначення	Безпека
getTasksBy UserId	GET	/:username/tasks	Вибірка задач за userId	JWT + username-перевірка
createTask	POST	/:username/tasks /create	Створення задачі з прив'язкою до userId	JWT + Zod-валідація
getTaskById	GET	/:username/tasks /:taskId	Отримання однієї задачі для редагування	JWT + userId-фільтр
changeTask	POST	/:username/tasks /:taskId	Оновлення статусу, важливості, дедлайну	JWT + WHERE id AND userId
deleteTask	DELETE	/:username/tasks /:taskId	Видалення задачі за id і userId	JWT + WHERE id AND userId

Такий підхід є стандартною практикою в OWASP для захисту від IDOR-атак (Insecure Direct Object Reference), де зловмисник намагається маніпулювати чужими ресурсами через підбір ідентифікатора.

4.4 Реалізація валідації та обробки помилок

У CapyDo валідація вхідних даних виконується централізовано через Zod-схеми до переходу до сервісного шару. Такий підхід запобігає потраплянню некоректних значень у бізнес-логіку та базу даних і забезпечує однакові правила перевірки для всіх критичних маршрутів.

Для користувацьких форм контролюються формат і межі полів: username допускає від 3 до 30 символів (латиниця, цифри, нижнє підкреслення), password – від 8 до 100 символів із вимогою наявності великої літери, малої літери та цифри, firstname – від 1 до 50 символів. Для задач застосовуються обмеження на допустимі значення status та importance, а поле dueTo перевіряється як валідна дата і може бути порожнім.

Обробка помилок побудована за принципом передбачуваних відповідей. У разі порушення правил валідації повертається код 400 Bad Request із деталями некоректних полів. Помилки доступу формуються окремо з кодами 401/403, що дозволяє коректно розділяти випадки неавтентифікованого і забороненого доступу.

Неочікувані винятки перехоплюються глобальним errorHandler, який повертає уніфіковану відповідь 500 Internal Server Error без розкриття внутрішніх технічних деталей реалізації. У результаті підсистема валідації та обробки помилок виконує не лише функцію контролю коректності даних, а й підтримує загальну надійність і безпечність серверного контуру.

4.5 Інтерфейс користувача

Інтерфейс побудовано на EJS-шаблонах зі спільним layout. Сторінки охоплюють базові сценарії користувача. Перелік сторінок зі значенням EJS-файлу наведено в таблиці 4.3.

Таблиця 4.3

Сторінки інтерфейсу CapyDo

EJS-файл	Маршрут	Опис	Доступ
index.ejs	/	Головна сторінка	Публічний
about.ejs	/about	Про проект	Публічний
contact.ejs	/contact	Контакти	Публічний
sign_up.ejs	/signup	Форма реєстрації	Публічний
login.ejs	/login	Форма входу	Публічний
account.ejs	/:username/account	Перегляд профілю	Приватний
account_settings.ejs	/:username/account-settings	Налаштування облікового запису	Приватний
tasks.ejs	/:username/tasks	Список задач	Приватний
task_create.ejs	/:username/tasks/create	Форма створення задачі	Приватний
task_update.ejs	/:username/tasks/:taskId	Форма редагування задачі	Приватний
layout.ejs	всі сторінки	Спільний шаблон (навігація, footer)	Всі

Інтерфейс спроєктовано за принципом мінімальної кількості дій для типових сценаріїв персонального планування. Основні операції (створення, редагування, видалення задачі, перехід до налаштувань акаунту) винесено у видимі елементи керування без прихованої навігації. Це зменшує когнітивне навантаження та підвищує швидкість виконання повторюваних дій.

Окрему увагу приділено узгодженості клієнтської та серверної валідації. Структура полів у формах відповідає очікуваним серверним схемам, що забезпечує передбачувану поведінку інтерфейсу у разі помилок введення та зменшує ризик неконсистентних даних.

Для приватних сторінок використано персоналізовану маршрутизацію з параметром username, що забезпечує чітке розмежування користувацького простору і логічно пов’язує UI з моделлю авторизації на сервері.

4.5.1 Візуальна демонстрація інтерфейсу

Візуальна верифікація інтерфейсу підтверджує відповідність реалізованих сторінок функціональним вимогам, описаних у розділах 1 і 3. Наведені скріншоти демонструють як штатні сценарії взаємодії, так і обробку помилок валідації без

втрати контексту форми, що підвищує зручність користування та зменшує ймовірність помилкових дій.

1. Головна сторінка

Показано стартову сторінку з навігацією до публічних і приватних сценаріїв.

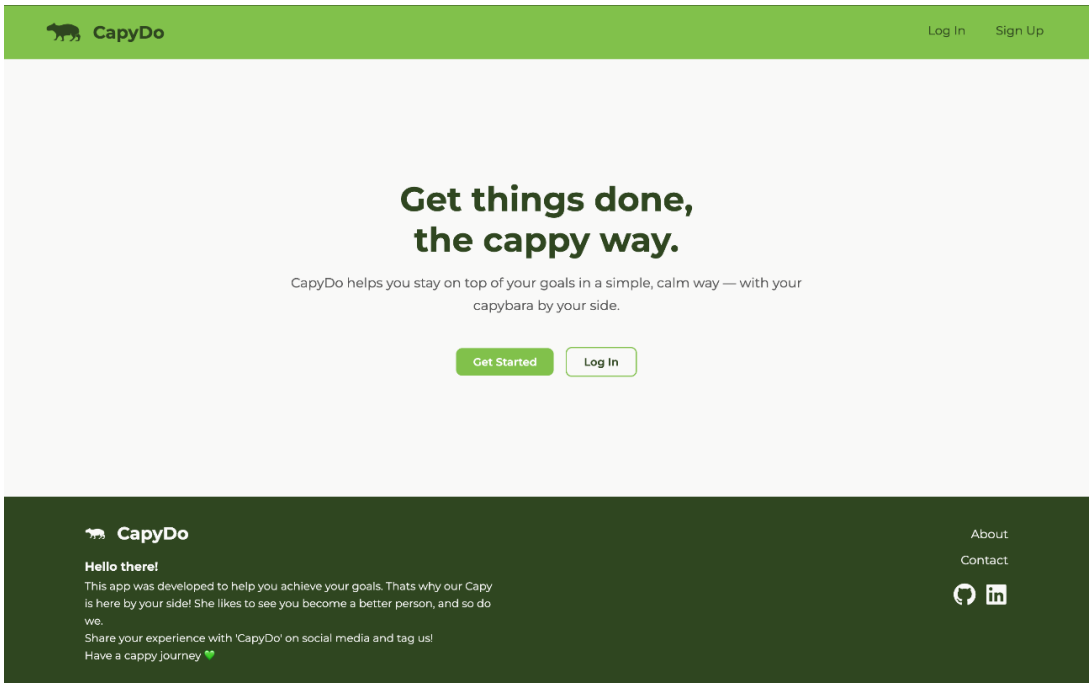


Рис 4.4 Головна сторінка

2. Реєстрація /signup (звичайний стан)

Форма містить поля firstname, username, password та клієнтську структуру введення.

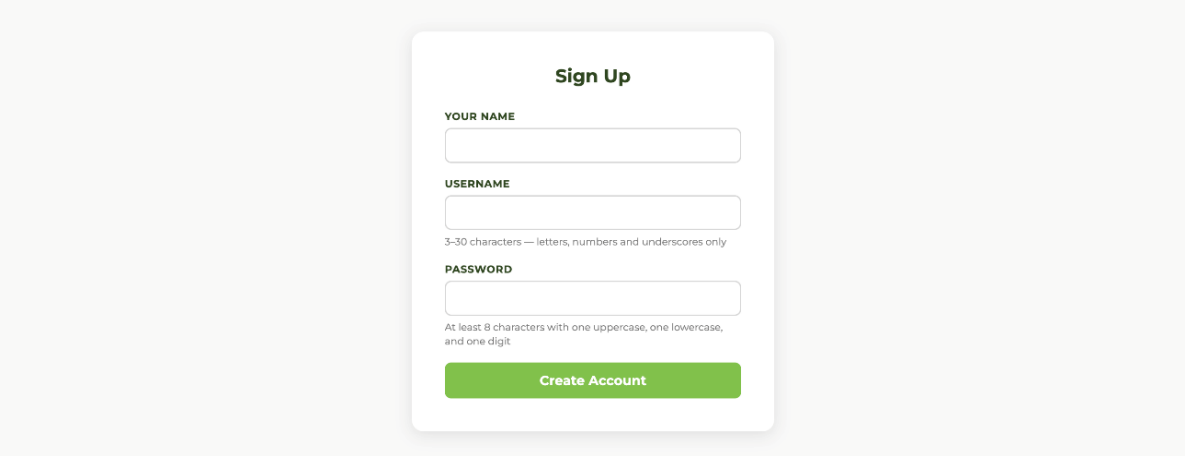


Рис 4.5 Форма реєстрації

3. Реєстрація /signup (невалідний пароль)

   Відображення помилки валідації у формі реєстрації.

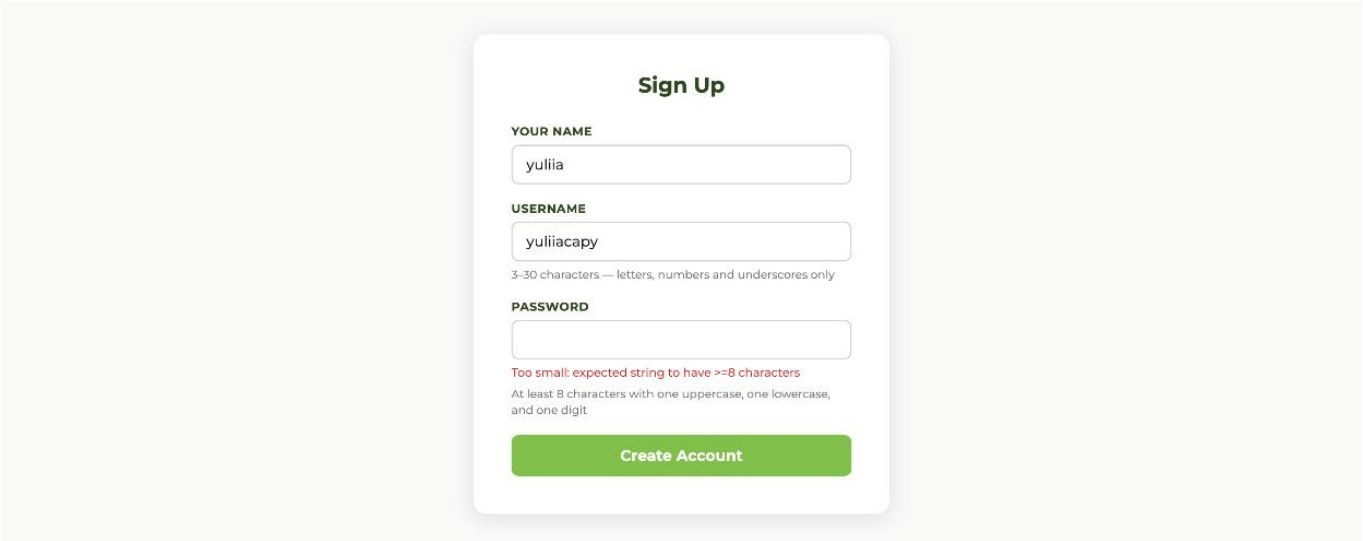


Рис 4.6 Форма реєстрації при невалідному паролі

4. Логін /login (неправильний пароль)

   Помилка автентифікації відображається у формі без переходу на технічну сторінку.

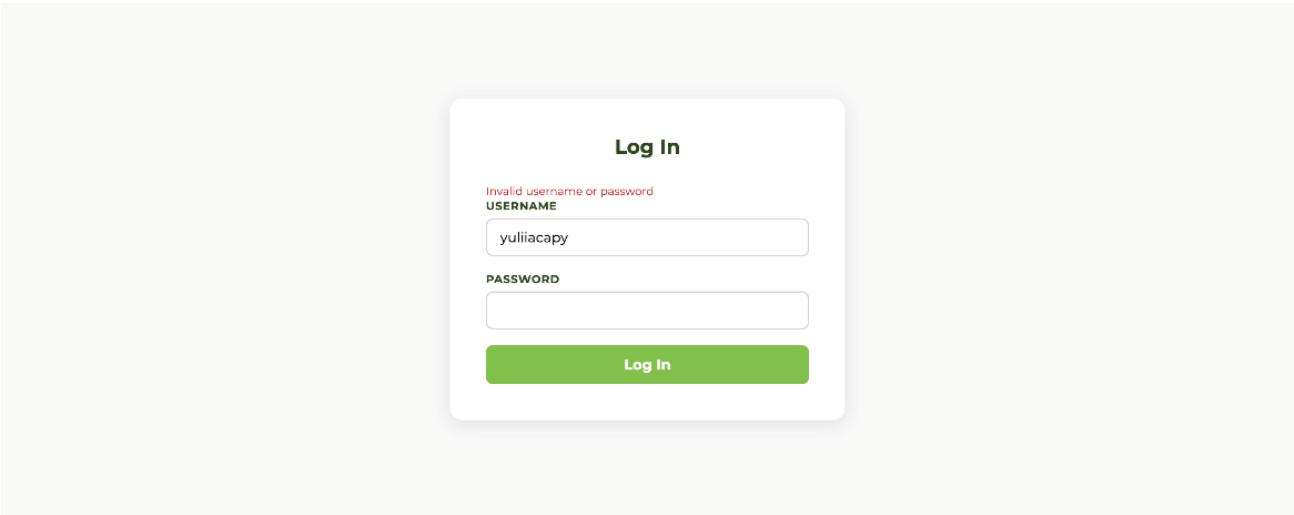


Рис 4.7 Форма входу в акаунт

5. Сторінка задач /:username/tasks

   Показано список задач, елементи керування редагуванням та видаленням.

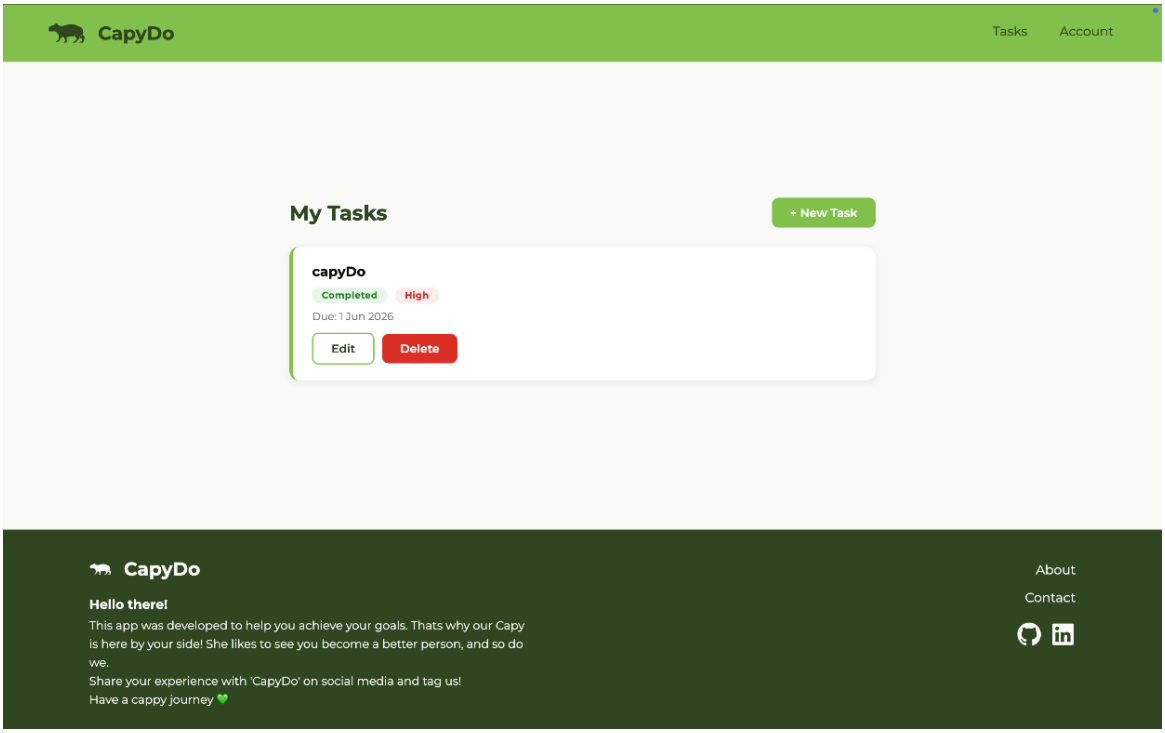


Рис 4.8 Сторінка задач

6. Створення задачі /:username/tasks/create

   Форма демонструє статус, важливість і не обов'язковий дедлайн.

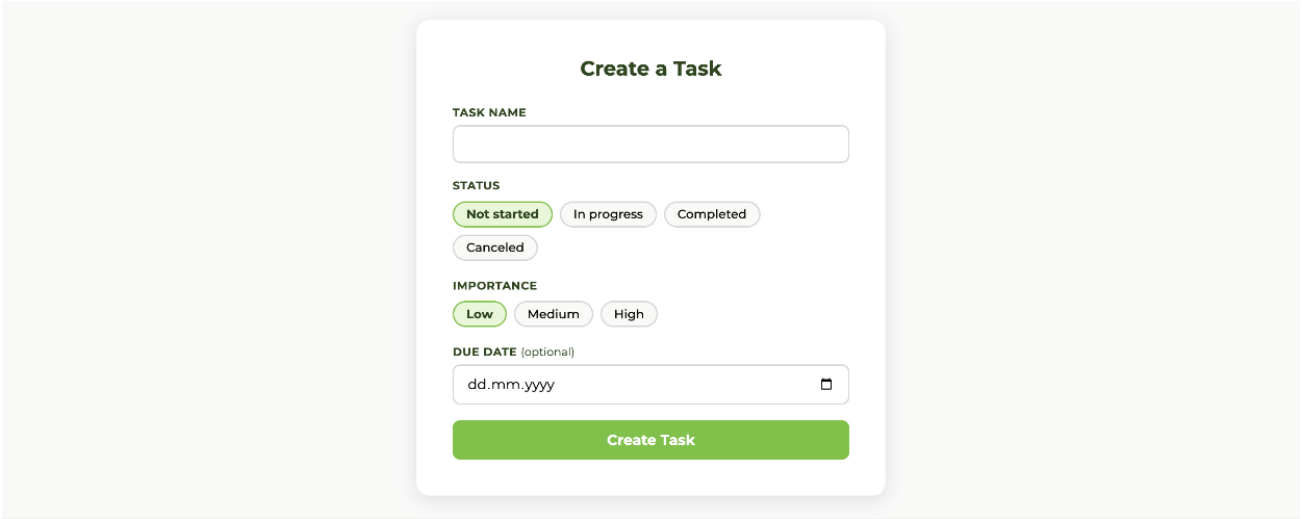


Рис 4.9 Форма створення задачі

7. Редагування задачі /:username/tasks/:taskId

Показано зміну статусу, важливості та дедлайну існуючої задачі.

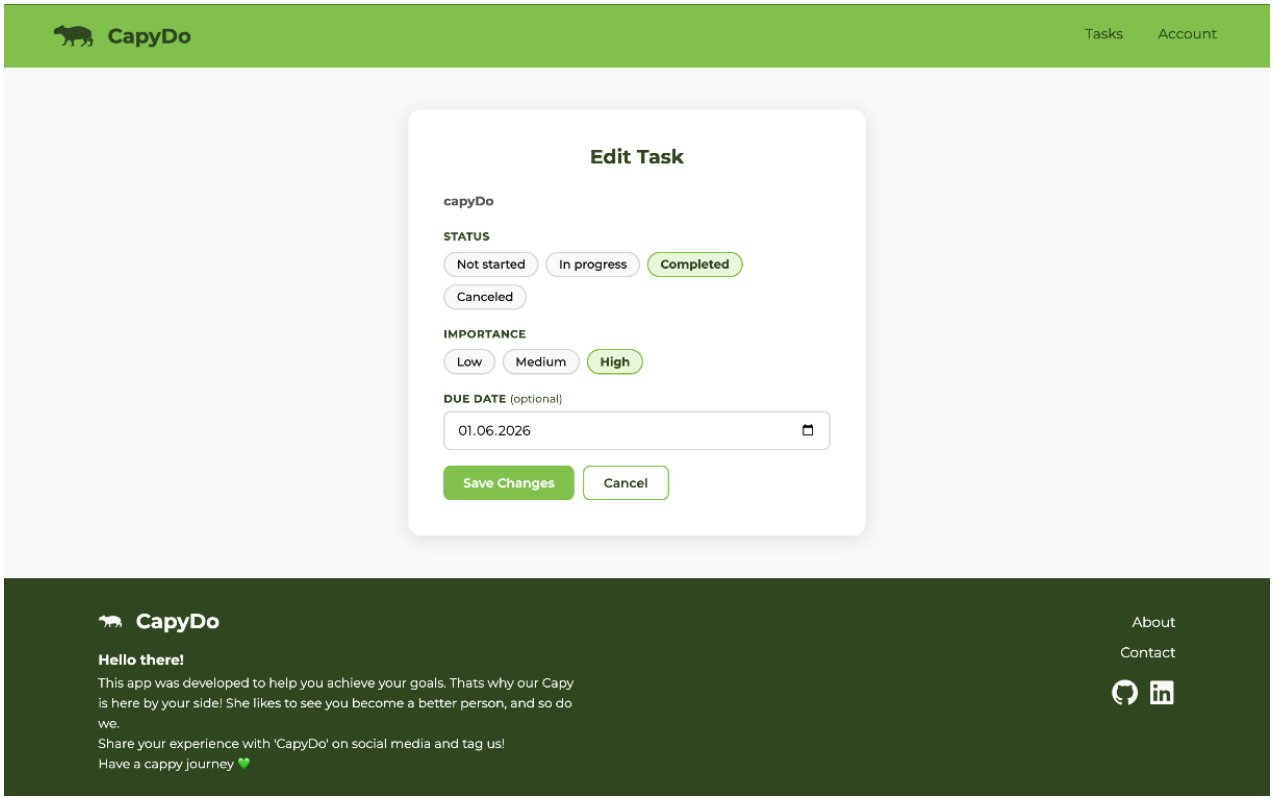


Рис 4.10 Сторінка редагування задачі

8. Налаштування акаунту /:username/account-settings

Система підтримує зміну firstname без обов'язкового введення нового паролю.

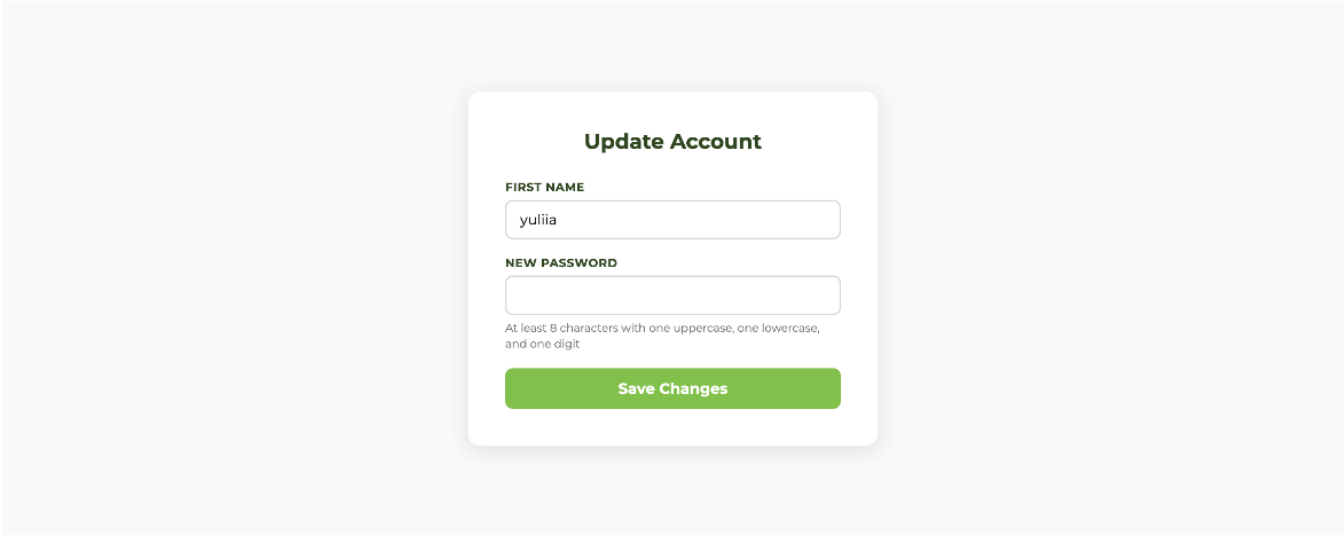


Рис 4.11 Форма налаштувань акаунту

4.6 Типові сценарії користувацької взаємодії

Сценарій реєстрації передбачає послідовні кроки: введення даних, валідація, перевірка унікальності, створення користувача, встановлення cookie і перенаправлення у профіль. Це забезпечує швидкий «вхід» користувача в систему без додаткових проміжних дій. Сценарій керування задачами передбачає створення, перегляд, редагування і видалення з обов'язковим контролем належності запису користувачу.

Детальну матрицю тестових сценаріїв наведено в підрозділі 4.6.1. Нижче описано підхід кожного сценарію з погляду користувача.

Сценарій 1 – Реєстрація нового користувача.

Користувач вводить ім'я, логін і пароль на сторінці /signup, після чого Zod-схема перевіряє формат полів – при некоректних даних користувач повертається на форму з відповідним повідомленням. Далі виконується перевірка унікальності логіна: у разі дублікату повертається помилка 409. Якщо перевірки пройдено успішно, bcrypt хешує пароль, дані користувача зберігаються в БД, JWT-токен записується в httpOnly cookie, а користувач перенаправляється до /:username/account.

Сценарій 2 – Створення задачі.

Авторизований користувач відкриває сторінку /:username/tasks/create і заповнює форму, вказуючи назву, статус, важливість та необов'язковий дедлайн. Zod-валідація перевіряє поля, допускаючи лише коректні enum-значення. Після успішної валідації taskService.createTask зберігає задачу з прив'язкою до userId користувача, після чого відбувається перенаправлення до /:username/tasks з оновленим списком задач.

Сценарій 3 – Оновлення облікового запису.

Авторизований користувач переходить на сторінку /:username/account-settings. Система підтримує два незалежні оновлення: зміна firstname виконується окремо від зміни пароля – поле нового пароля є необов'язковим. Якщо нове значення firstname передане без пароля, Zod-схема пропускає поле пароля без

помилки. Якщо передано новий пароль – він проходить повторну валідацію (мінімум 8 символів, наявність великої літери, малої та цифри) і хешується через bcrypt.hash() перед збереженням. Після успішного оновлення система повертає перенаправлення на сторінку акаунту.

Сценарій 4 – Спроба несанкціонованого доступу.

Неавторизований користувач намагається відкрити URL /:username/tasks напряму. Глобальний JWT-middleware не виявляє cookie auth-token – запит перенаправляється на /login з кодом 302. Якщо авторизований користувач alice намагається відкрити /:bob/tasks – перша перевірка (збіг username в URL) блокує запит і перенаправляє на /:alice/tasks. Жодної інформації про задачі користувача bob при цьому не повертається.

Сценарій 5 – Видалення облікового запису.

Авторизований користувач ініціює видалення акаунту зі сторінки профілю. Перед виконанням операції система перевіряє належність URL-параметра поточному користувачу. Після успішної перевірки обліковий запис видаляється з бази даних, cookie auth-token очищується, а користувач перенаправляється на головну сторінку.

Завдяки каскадному видаленню пов’язані задачі цього користувача видаляються автоматично. Це гарантує цілісність даних і виключає появу «осиротілих» записів у таблиці задач.

4.6.1 Перевірені сценарії

Покрито такі групи сценаріїв (Таблиця 4.5):

Таблиця 4.5

Матриця тестових сценаріїв

№	Група тестів	Тип	Кількість тестів	Результат
1	Реєстрація (успішна/дублікат/невалід)	Integration	6	Pass
2	Вхід (успішний/неправильний пароль)	Integration	5	Pass
3	Доступ без токена до приватних маршрутів	Integration	8	Pass
4	Доступ з чужим username в URL	Integration	4	Pass
5	CRUD задач (успішні сценарії)	Integration	10	Pass
6	CRUD задач (невалідні дані)	Integration	6	Pass
7	Видалення акаунту	Integration	3	Pass
8	Rate limit (перевищення ліміту)	Unit	5	Pass
9	Middleware errorHandler	Unit	8	Pass
10	Валідація Zod-схем	Unit	12	Pass
11	Рендеринг EJS	Unit	7	Pass
12	JWT session (create/verify/revoke)	Unit	8	Pass
13	authService (CRUD користувача)	Unit	10	Pass
14	taskService (CRUD задач)	Unit	4	Pass
Всього			98	Pass

4.7 Реалізація захисту на рівні прикладної логіки

Захист реалізовано багаторівнево: токен зберігається в HTTP-only cookie, автентифікація перевіряється в глобальному middleware, збіг username в URL з авторизованим користувачем контролюється окремо, а операції update і delete прив'язуються до userId безпосередньо на рівні запитів до БД. Поєднання цих перевірок формує практичну модель багаторівневого захисту, що критично важливо для вебзастосунків із персональними даними. Перелік рівнів захисту наведено в таблиці 4.4.

Таблиця 4.4

Рівні захисту в CapyDo

Рівень	Тип	Механізм	Технічна реалізація
1	Автентифікація	Зберігання httpOnly cookie	sameSite=strict, відсутність в JS-контексті
2	Автентифікація	Перевірка JWT у глобальному middleware	verifyAccessToken(), виняток при помилці
3	Авторизація	Перевірка username в URL	Порівняння URL-параметра з user.username
4	Авторизація	userId-фільтрація в CRUD-запитах	WHERE id=? AND userId=? у Prisma-запитах
5	Валідація	Zod-валідація вхідних даних	Схеми RegisterSchema, CreateTaskSchema
6	Захист мережі	Rate limiting на рівні IP	200 req / 15 хв, заголовок Retry-After
7	Захист мережі	Secure headers (OWASP)	helmet middleware, CSP, X-Frame-Options, HSTS тощо
8	Захист мережі	Санітизація помилок	errorHandler не віддає внутрішніх деталей

Захист на будь-якому одному рівні не дає зловмиснику доступу до чужих ресурсів або можливості маніпулювати чужими задачами.

4.8 Модульність і розширюваність архітектури

Архітектура CapyDo побудована на принципі розділення відповідальностей: маршрути, сервіси, middleware і допоміжні модулі ізольовані за ролями та мають чіткі межі взаємодії. Route-рівень відповідає за HTTP-обробку і формування відповіді, service-рівень інкапсулює бізнес-логіку, middleware реалізує наскрізні механізми безпеки та валідації, а бібліотечні модулі забезпечують рендеринг і роботу із сесією.

Така декомпозиція підвищує підтримуваність: зміни правил валідації не вимагають переробки сервісного шару, а модифікації бізнес-правил не порушують логіку маршрутизації чи рендерингу. У результаті система залишається керованою при поступовому ускладненні функціоналу.

Наявне автоматизоване тестування підсилює модульну архітектуру, оскільки кожен шар перевіряється у відповідному контексті: окремо для локальної логіки і

окремо для інтеграційної поведінки маршрутів. Це знижує ризик регресій і робить подальші зміни безпечнішими.

4.9 Організація тестування

У проекті використано два рівні тестування:

Unit-тести: middleware (errorHandler, rateLimit, staticCache, validation), рендеринг (render.ts), сесії (session.ts), сервіси (authService, taskService).

Integration-тести: HTTP-маршрути (authRoute, tasksRoute), авторизаційні сценарії, перенаправлення, перевірка гілок доступу.

Для проведення тестування був використаний фреймворк Vitest. Загальна кількість: 11 тестових файлів, 98 тестів (усі успішно пройшли).

Тестовий клієнт Hono (hono/testing) дозволяє надсилати HTTP-запити безпосередньо до застосунку без запуску реального TCP-сервера та підключення до бази даних. Залежності від зовнішніх ресурсів (Prisma, bcrypt, JWT) замінені на стаби (vi.mock), що забезпечує детермінованість і швидкість виконання тестів незалежно від стану оточення.

На рис. 4.12 зображено структуру автоматизованого тестування CapyDo з поділом на модульний та інтеграційний рівні. Модульні тести перевіряють middleware, бібліотечні модулі та сервісний шар ізольовано, що дозволяє швидко локалізувати дефекти у конкретному компоненті. Інтеграційні тести охоплюють поведінку маршрутів, перевірки авторизації, редиректи та коректність обробки типових користувацьких сценаріїв.

Така організація тестів формує збалансовану стратегію якості: швидкі локальні перевірки поєднуються з перевірками наскрізної взаємодії компонентів. У сукупності це знижує ризик регресій під час змін у коді та підтверджує, що механізми безпеки і бізнес-логіка працюють узгоджено.

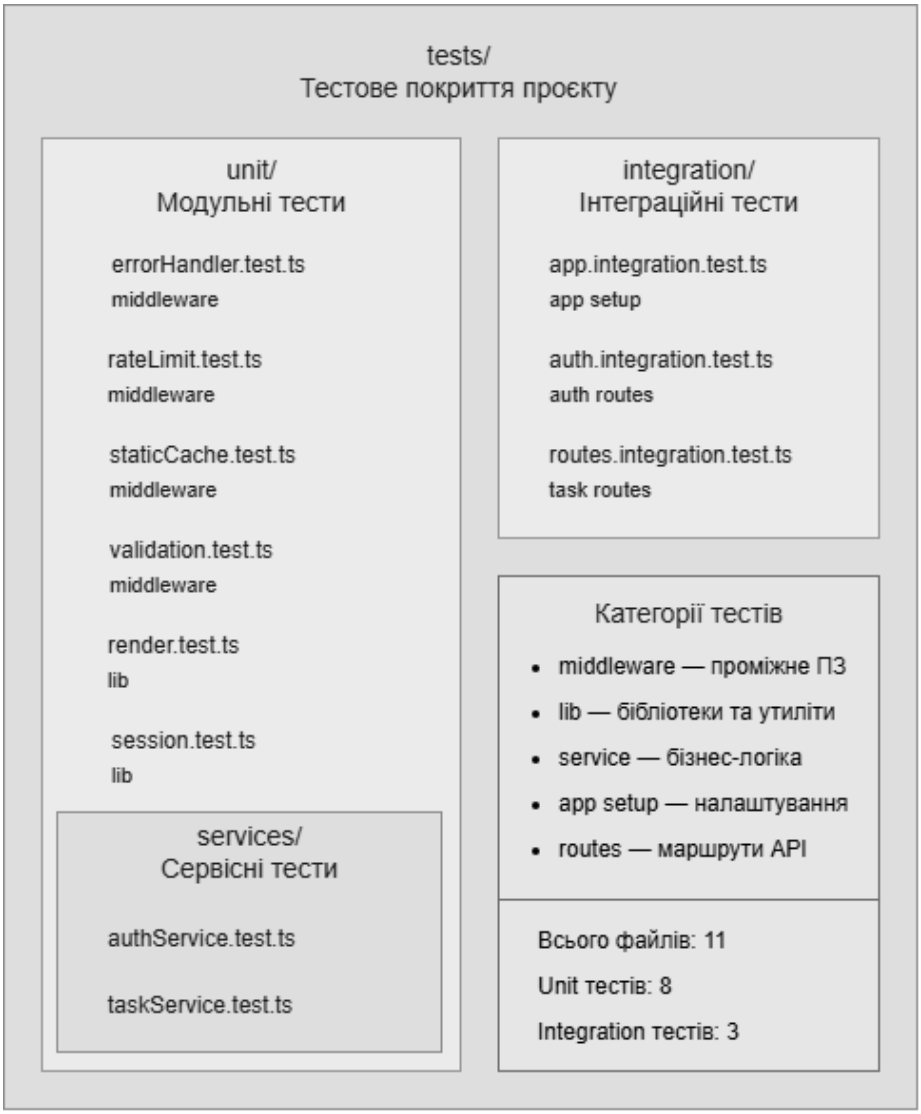


Рис. 4.12 Структурна діаграма тестових файлів у CapyDo

4.9.1 Результати покриття

За результатами запуску npm run test:coverage отримано підсумкові метрики покриття коду (Таблиця 4.6) та деталізацію по ключових файлах (Таблиця 4.7).

Таблиця 4.6

Зведені метрики покриття коду

Метрика	Значення	Оцінка
Statements	88.52%	Добре
Branches	75.41%	Задовільно
Functions	91.93%	Добре
Lines	88.73%	Добре
Всього тестів	98	Усі пройшли (pass)

Таблиця 4.7

Покриття коду за ключовими модулями

Файл	Statements	Branches	Functions	Lines
middleware/errorHandler.ts	100%	100%	100%	100%
middleware/rateLimit.ts	100%	100%	100%	100%
middleware/staticCache.ts	100%	100%	100%	100%
middleware/validation.ts	100%	100%	100%	100%
lib/render.ts	100%	100%	100%	100%
lib/session.ts	100%	100%	100%	100%
services/authService.ts	~92%	~80%	~95%	~92%
services/taskService.ts	~90%	~78%	~93%	~90%
routes/authRoute.ts	94.64%	~78%	~90%	94.64%
routes/tasksRoute.ts	76.54%	~62%	~82%	76.54%

Примітка: значення у стовпцях з "~" є розрахунковими на основі загальних підсумків тестового звіту.

Ці значення підтверджують достатній рівень перевірки критичних компонентів системи, особливо middleware і сервісного шару, де досягнуто 100% покриття.

4.10 Методика перевірки якості

Тестову перевірку доцільно розглядати як комплекс із декількох рівнів, що охоплює функціональну перевірку коректності реалізації сценаріїв, негативне тестування для невалідних параметрів, перевірку доступу до приватних ресурсів для неавторизованих запитів, а також перевірку поведінки системи при помилках і крайових умовах. Такий підхід забезпечує не лише перевірку «щасливих» сценаріїв, а й контроль поведінки застосунку в реальних експлуатаційних умовах. Деталізовану методику тестування наведено в таблиці 4.8.

Таблиця 4.8

Методика тестової перевірки CapyDo

Рівень перевірки	Опис	Тип тесту
Функціональна коректність	Перевірка щасливих сценаріїв CRUD, auth	Integration
Негативне тестування	Невалідні вхідні дані, спотворені помилки	Unit + Int.
Перевірка автентифікації	Доступ без токена, з чужим username	Integration
Перевірка middleware	Rate limit, error handler, validation, cache	Unit
Перевірка JWT-сесії	create / verify / revoke (stub)	Unit
Перевірка сервісів	CRUD authService, taskService	Unit

Особлива увага приділена негативному тестуванню — перевірці поведінки системи при некоректних вхідних даних. Для кожного маршруту, що приймає форму, перевіряються сценарії з порожніми полями, перевищенням максимальної довжини, недопустимими символами та некоректними enum-значеннями. Це відповідає принципу «fail fast»: система повинна відхиляти некоректні дані якомога раніше, повертаючи чіткі повідомлення про помилку, а не допускати їх до бізнес-логіки або бази даних.

Для тестування авторизаційних сценаріїв використовується підхід із підробленими токенами: тести перевіряють поведінку системи при наявності прострочених токенів, токенів із невірним підписом і токенів без обов'язкових полів (jti, userId). Всі ці сценарії повинні повертати 401 або перенаправлення на /login без витоку внутрішніх деталей.

4.11 Інтерпретація метрик покриття

Інтерпретація метрик покриття дає змогу оцінити не лише обсяг виконаного коду під час тестування, а й збалансованість перевірки різних типів логіки.

Показник Lines = 88.73% свідчить, що переважна частина реалізованої поведінки підтверджена автоматизованими тестами. Значення Statements = 88.52% є близьким до Lines, що вказує на узгодженість результатів і відсутність суттєвих «сліпих зон» на рівні базових інструкцій.

Метрика Functions = 91.93% демонструє, що майже всі функціональні блоки системи фактично викликаються в тестових сценаріях. Водночас Branches = 73.41% фіксує резерв для подальшого посилення перевірки альтернативних гілок: виняткових ситуацій, відмов інфраструктури, а також малоймовірних комбінацій вхідних параметрів. Саме ця метрика найчутливіше відображає повноту негативного тестування.

Інтерпретацію кожної метрики та її практичне значення наведено в таблиці 4.9.

Таблиця 4.9

Інтерпретація метрик покриття

Метрика	Значення	Інтерпретація
Statements	88.52%	Більшість операторів виконуються хоча б один раз під час тестів
Branches	75.41%	Рідкі гілки (інфразбої, edge case) не повністю покриті
Functions	91.93%	Майже всі оголошені функції викликаються хоча б один раз
Lines	88.73%	Дуже близький до Statements, підтверджує якість
Middleware	100%	Повне покриття – пріоритетний продуктивний результат
Lib (render, session)	100%	Повне покриття – критичний код безпеки

Отримані значення свідчать про високий рівень тестової зрілості проєкту: критичні компоненти middleware і бібліотечні модулі мають 100% покриття, а загальні інтегральні показники перевищують типовий практичний орієнтир 80%.

Це створює надійну основу для експлуатації системи та подальшого розвитку. Пріоритетним напрямом удосконалення залишається підвищення branch coverage за рахунок додаткових сценаріїв негативного тестування та перевірок крайових випадків.

Практичне значення наведених метрик полягає в тому, що вони можуть використовуватися як об'єктивний індикатор регресій під час подальших ітерацій. За умови внесення змін у маршрути автентифікації, валідації або операції над задачами зниження показників покриття буде сигналом до доповнення тестового набору. Таким чином, метрики виконують не лише підсумкову, а й керувальну функцію в процесі супроводу програмного продукту.

Окремо варто зазначити, що нерівномірність між Functions і Branches є типовою для вебзастосунків, де значна частина альтернативної логіки належить до обробки помилок і захисних перевірок. Для CapyDo це насамперед сценарії некоректних cookie, невалідних route-параметрів, конфліктних станів сутностей та відмов з боку бази даних. Розширення саме цих сценаріїв дозволить підвищити branch coverage без штучного ускладнення тестової архітектури.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи розроблено клієнт-серверний вебзастосунок CapyDo для організації персональних завдань із використанням TypeScript та PostgreSQL.

Під час виконання роботи:

Проаналізовано предметну область персонального task-менеджменту, виявлено потребу в легкому, безпечному і контрольованому рішенні для індивідуального використання. Встановлено, що існуючі комерційні продукти (Todoist, Trello, Notion) мають обмеження у сфері контролю над даними та не забезпечують прозорості реалізованих механізмів безпеки.

Обґрунтовано вибір технологічного стеку: TypeScript 5.8.3, Hono 4.8.3, PostgreSQL 17, Prisma 7.8.0, EJS 3.1.10, Vitest 4.1.5. Кожен компонент обраний виходячи з критеріїв безпеки типів даних, продуктивності і відповідності масштабу задачі.

Спроектовано реляційну модель даних із двома таблицями (users, task), зовнішніми ключами, каскадним видаленням і трьома оптимізованими індексами для ефективних вибірок. Розроблено повну маршрутну таблицю з 18 ендпоінтів із розмежуванням публічного і приватного доступу.

Реалізовано модулі реєстрації та автентифікації (bcrypt-хешування, JWT у httpOnly cookie), керування обліковим записом (зміна даних, видалення), повний CRUD для задач із контролем належності ресурсу конкретному користувачу.

Забезпечено захист застосунку на кількох рівнях: валідація Zod, helmet middleware, rate limiting, перевірка відповідності username в URL авторизованому користувачу, прив'язка UPDATE/DELETE операцій до userId у запитах до БД.

Проведено 98 автоматизованих тестів (unit і integration) із використанням Vitest. Загальне покриття становить 88.73% за рядками, 91.93% за функціями. Middleware і сервісний шар досягли 100%-го покриття.

Наукова новизна роботи полягає у поєднанні системного підходу до безпеки (багаторівневий захист: валідація + JWT + cookie policy + rate limiting + userId-фільтрація), TypeScript-типізації на всіх рівнях і тестово-орієнтованого процесу розробки в межах повного навчального проекту.

Практична цінність реалізованого застосунку виявляється у кількох вимірах:

- як персональний інструмент планування, готовий до продуктивного використання;
- як навчальна платформа для курсів із веброзробки, баз даних і кібербезпеки;
- як еталонний приклад структури TypeScript-проекту з тестами, middleware і ORM.

Отримані результати відповідають поставленій меті та демонструють придатність CapyDo для практичного використання як персонального вебінструмента планування задач.

ПЕРЕЛІК ПОСИЛАНЬ

1. Рябко Ю.В., Срібна І.М., Аналіз вразливостей JWT-автентифікації у застосунках на основі Node.JS та методи їх усунення // Всеукраїнська науково-технічна конференція «Технології цифрового розвитку: програмні системи та децентралізація». Збірник тез. 01-03.05.2026, ДУІКТ, Київ, Україна
2. Рябко Ю.В., Срібна І.М., Методи оптимізації продуктивності серверних застосунків на основі NODE.JS // Всеукраїнська науково-технічна конференція «Технології цифрового розвитку: програмні системи та децентралізація». Збірник тез. 01-03.05.2026, ДУІКТ, Київ, Україна
3. Рябко Ю.В., Срібна І.М., Порівняльний аналіз NLP-методів для аналізу тональності відгуків користувачів на платформах електронної комерції // VI Всеукраїнська науково-практична конференція «Сучасні інтелектуальні інформаційні технології в науці та освіті». Збірник тез. 15.05.2026, ДУІКТ, Київ, Україна
4. TypeScript Documentation. URL: https://www.typescriptlang.org/docs/ (дата звернення: 25.03.2026).
5. Hono Documentation. URL: https://hono.dev/docs/ (дата звернення: 25.03.2026).
6. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/ (дата звернення: 31.03.2026).
7. Prisma Documentation. URL: https://www.prisma.io/docs/ (дата звернення: 31.03.2026).
8. Vitest Documentation. URL: https://vitest.dev/ (дата звернення: 05.04.2026).
9. Zod Documentation. URL: https://zod.dev/ (дата звернення: 05.04.2026).
10. OWASP Top Ten Project. URL: https://owasp.org/www-project-top-ten/ (дата звернення: 10.04.2026).
11. RFC 7519: JSON Web Token (JWT). URL: https://datatracker.ietf.org/doc/html/rfc7519 (дата звернення: 10.04.2026).

12. MDN Web Docs: Using HTTP cookies. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Cookies (дата звернення: 10.04.2026).

13. Node.js API Documentation. URL: https://nodejs.org/docs/latest/api/ (дата звернення: 10.04.2026).

14. EJS Documentation. URL: https://ejs.co/ (дата звернення: 15.04.2026).

15. node.bcrypt.js repository. URL: https://github.com/kelektiv/node.bcrypt.js (дата звернення: 15.04.2026).

16. Електронний репозиторій ДУІКТ: кафедра Інформаційних систем та технологій. URL: https://duikt.edu.ua/ua/reposit?path=Кафедра+Інформаційних+систем+та+технологій (дата звернення: 20.04.2026).

17. MDN Web Docs: REST. URL: https://developer.mozilla.org/en-US/docs/Glossary/REST (дата звернення: 20.04.2026).

18. HTTP Status Codes Documentation. URL: https://httpwg.org/specs/rfc7231.html (дата звернення: 20.04.2026).

19. OWASP SQL Injection Prevention. URL: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html (дата звернення: 20.04.2026).

20. CORS Specification. URL: https://fetch.spec.whatwg.org/#http-cors-protocol (дата звернення: 20.04.2026).

21. MDN Web Docs: Authentication on the Web. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Types_of_attacks (дата звернення: 24.04.2026).

22. Express.js Documentation. URL: https://expressjs.com/ (дата звернення: 24.04.2026).

23. API Best Practices and Security Guidelines. URL: https://restfulapi.net/ (дата звернення: 24.04.2026).

24. npm Registry: Popular JavaScript Packages. URL: https://www.npmjs.com/ (дата звернення: 24.04.2026).

25. OWASP CSRF Prevention Cheat Sheet. URL: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html (дата звернення: 24.04.2026).

26. Web Security Testing Guide . URL: https://owasp.org/www-project-web-security-testing-guide/ (дата звернення: 24.04.2026).

27. Todoist Pricing. URL: https://todoist.com/pricing (дата звернення: 25.04.2026).

28. Trello Pricing. URL: https://trello.com/pricing (дата звернення: 25.04.2026).

29. Notion Pricing. URL: https://www.notion.so/pricing (дата звернення: 25.04.2026).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ПРЕЗЕНТАЦІЯ ДО КВАЛІФІКАЦІЙНОЇ БАКАЛАВРСЬКОЇ РОБОТИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

ПРЕЗЕНТАЦІЯ ДО КВАЛІФІКАЦІЙНОЇ
БАКАЛАВРСЬКОЇ РОБОТИ

на тему:

«Клієнт-серверний вебзастосунок CapyDo для організації персональних завдань із використанням
TypeScript та PostgreSQL»

Виконала: здобувачка вищої освіти групи ІСД-42
Юлія РЯБКО

Керівник: д-р техн. наук, доцент
Ірина СРІБНА

Київ-2026

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи – проектування та розробка клієнт-серверного вебзастосунку CapyDo для організації персональних завдань із використанням TypeScript та PostgreSQL, що забезпечує безпечну автентифікацію та авторизацію, повний цикл CRUD-операцій над задачами та перевірену автоматизованими тестами якість реалізації.

Об'єкт дослідження – процеси керування персональними завданнями у вебсередовищі.

Предмет дослідження – методи та засоби реалізації клієнт-серверного вебзастосунку для управління завданнями із автентифікацією, авторизацією та збереженням даних у реляційній БД.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Проаналізувати предметну область та функціональні вимоги.
2. Обрати технологічний стек і обґрунтувати архітектуру рішення.
3. Спроектувати структуру даних і серверні маршрути.
4. Реалізувати модулі автентифікації, авторизації та керування завданнями.
5. Забезпечити захист застосунку через валідацію, обмеження запитів і контроль доступу.
6. Провести модульне та інтеграційне тестування.
7. Оцінити результати розробки за кількісними показниками.

АНАЛІЗ АНАЛОГІВ

Критерій	Todoist	Trello	Notion	CapyDo
Безкоштовний план	Так, з обмеженнями	Так, з обмеженнями	Так, з обмеженнями	Так, без обмежень
Обмеження безкоштовного плану	До 5 проєктів	До 10 учасників, обмежені дошки	Блоки обмежені для 2+ користувачів	Відсутні
Збереження даних на власному сервері	Ні	Ні	Ні	Так
Вхід через логін і пароль	Так	Так	Так	Так
Підтримка OAuth	Так	Так	Так	Ні
Дедлайни для задач	Так	Так	Так	Так
Пріоритети задач	Так (P1–P4)	Через мітки	Через властивості	Так (3 рівні)
Статуси задач	Виконано / Не виконано	Через переміщення між списками	Налаштовувані	4 фіксовані статуси
Командна робота	Так	Так	Так (платно)	Ні (персональний)
Мобільний застосунок	Так	Так	Так	Ні
Відкритий вихідний код	Ні	Ні	Ні	Так

ВИМОГИ ДО СИСТЕМИ

Функціональні вимоги

- Реєстрація нового користувача
- Автентифікація (вхід за логіном і паролем)
- Авторизація (перевірка JWT, контроль доступу до ресурсів)
- Перегляд профілю та оновлення облікових даних
- Вихід із системи та видалення акаунту
- Перегляд списку власних задач
- Створення задачі (назва, статус, важливість, дедлайн)
- Редагування та видалення задач

ВИМОГИ ДО СИСТЕМИ

Нефункціональні вимоги

- Безпека - валідація Zod, bcrypt-хешування, httpOnly cookie, rate limiting (200 req/15 хв)
- Надійність - централізована обробка помилок (AppError, HTTP 4xx/5xx)
- Продуктивність - стиснення відповідей, кешування EJS-шаблонів (TTL 5 хв)
- Підтримуваність - TypeScript-типізація, розподіл на шари (routes / services / middleware / lib)
- Тестованість - 96 тестів Vitest, покриття 88,5%
- Масштабованість - stateless JWT, ієрархічна архітектура, Prisma migrations

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

TypeScipt

Typed superset JavaScript, що додає статичну типізацію та покращує якість коду.

Prisma ORM

Сучасний ORM для TypeScript і Node.js з автоматичною типізацією та міграціями.

NPM

Стандартний менеджер пакетів для JavaScript, що входить до складу Node.js

Hono

Легкий і швидкий веб фреймворк для Edge та Node.js. Мінімалістичний та гнучкий.

PostgreSQL

Відкрита реляційна база даних з розширеними можливостями

EJS

Простий шаблонізатор для JavaScript, що дозволяє генерувати HTML.

Node.js

JavaScript runtime-середовище на базі V8, що дозволяє виконувати код на сервері.

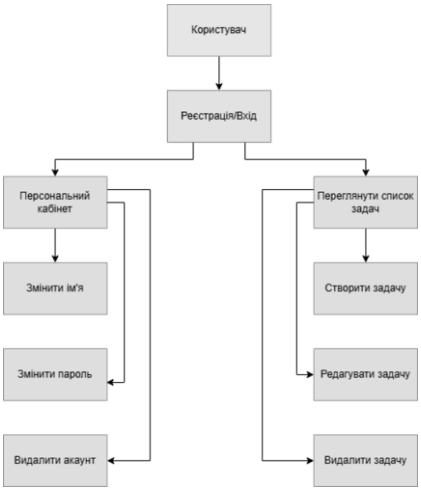
JWT

JSON Web Token для безпечної передачі інформації між сторонами у вигляді токенів.

SASS

Розширення CSS з змінними, вкладенням, міксинами та іншими корисними можливостями.

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ ТА ER-МОДЕЛЬ ДАНИХ



users

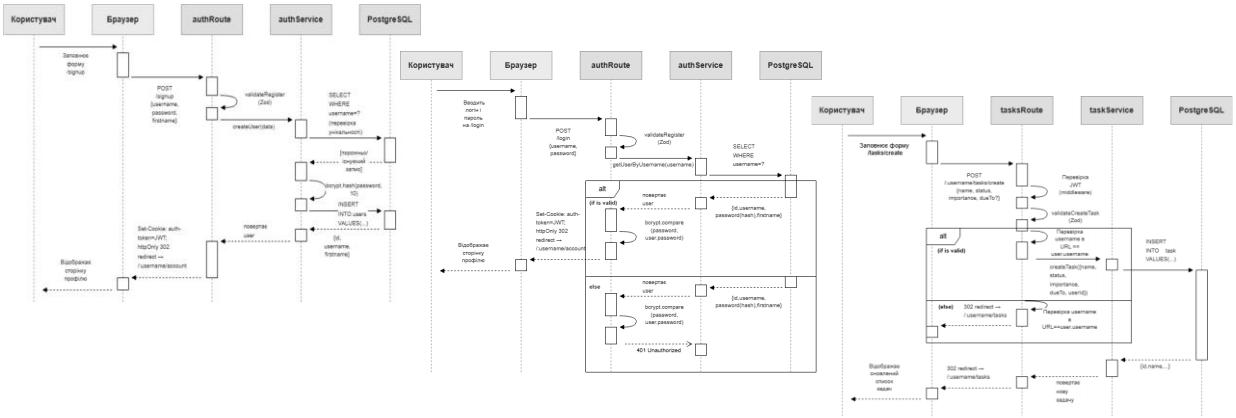
Обмеження	Поле	Тип
PK, NOT NULL	id	SERIAL
UNIQUE, NOT NULL	username	VARCHAR(30)
NOT NULL	password	VARCHAR(100)
NOT NULL	firstname	VARCHAR(50)

1:N

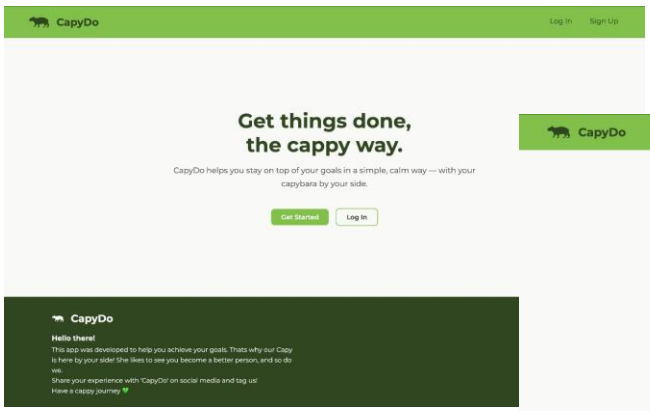
task

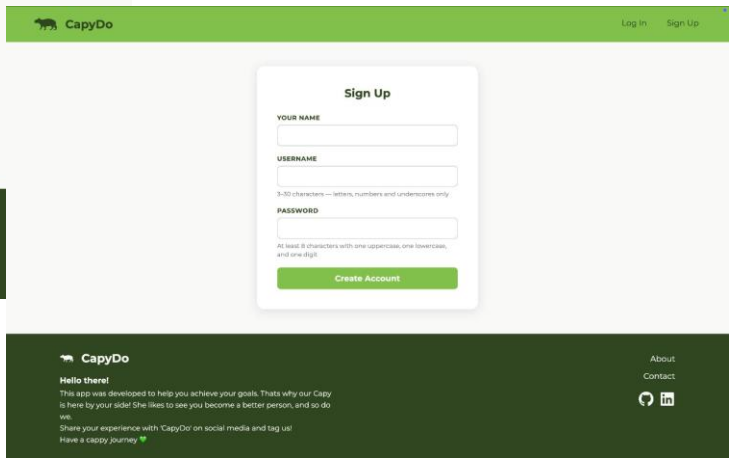
Обмеження	Поле	Тип
PK, NOT NULL	id	SERIAL
NOT NULL	name	VARCHAR(200)
NOT NULL (4 значення)	status	VARCHAR(255)
NOT NULL (3 значення)	importance	VARCHAR(255)
NULL (необов'язково)	dueTo	TIMESTAMP
NOT NULL, default now()	dateCreated	TIMESTAMP
FK → users.id, NOT NULL	userId	INT

Діаграми послідовності реєстрації / входу / операцій із задачами

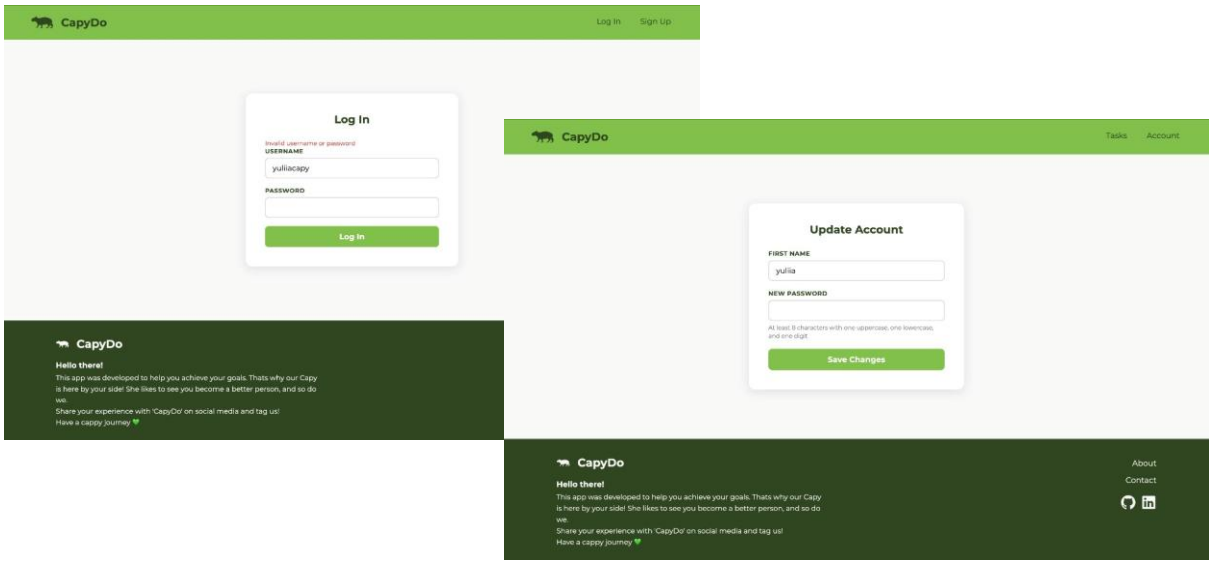


ЕКРАННІ ФОРМИ

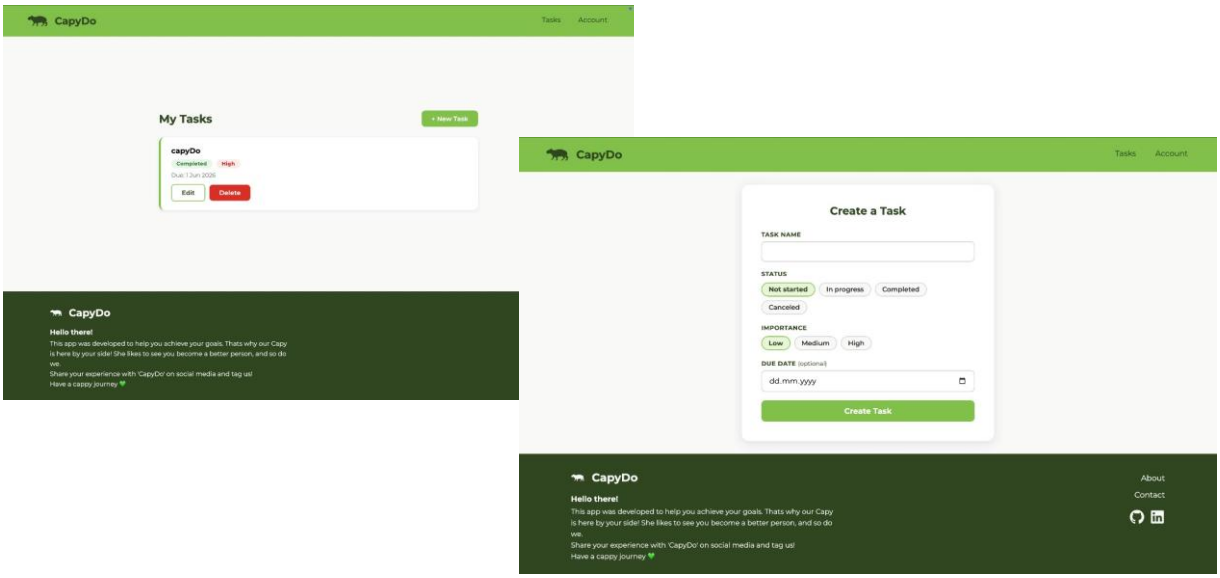




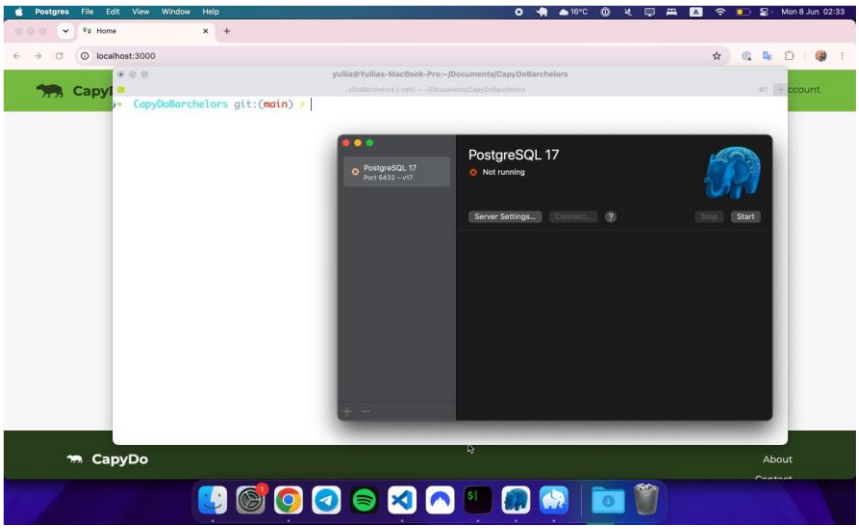
ЕКРАННІ ФОРМИ



ЕКРАННІ ФОРМИ



ВІДЕОДЕМОНСТРАЦІЯ РОБОТИ



АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Рябко Ю.В., Срібна І.М., Аналіз вразливостей JWT-автентифікації у застосунках на основі Node.JS та методи їх усунення // Всеукраїнська науково-технічна конференція «Технології цифрового розвитку: програмні системи та децентралізація». Збірник тез. 01-03.05.2026, ДУІКТ, Київ, Україна

2. Рябко Ю.В., Срібна І.М., Методи оптимізації продуктивності серверних застосунків на основі NODE.JS // Всеукраїнська науково-технічна конференція «Технології цифрового розвитку: програмні системи та децентралізація». Збірник тез. 01-03.05.2026, ДУІКТ, Київ, Україна

3. Рябко Ю.В., Срібна І.М., Порівняльний аналіз NLP-методів для аналізу тональності відгуків користувачів на платформах електронної комерції // VI Всеукраїнська науково-практична конференція «Сучасні інтелектуальні інформаційні технології в науці та освіті». Збірник тез. 15.05.2026, ДУІКТ, Київ, Україна

ВИСНОВКИ

- Розроблено клієнт-серверний вебзастосунок CapyDo на TypeScript, Hono, PostgreSQL, Prisma
- Спроектовано реляційну БД та 18 API-ендпоінтів із розмежуванням доступу
- Реалізовано автентифікацію (bcrypt + JWT у httpOnly cookie) та повний CRUD задач
- Забезпечено багаторівневий захист: Zod-валідація, rate limiting, secure headers, userId-фільтрація
- Проведено 98 автоматизованих тестів (Vitest): покриття 88,73% рядків, 91,93% функцій
- Застосунок придатний як персональний планувальник, навчальна платформа та еталонний TypeScript-проект

ДЯКУЮ ЗА УВАГУ!

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

Ініціалізація серверного застосунку (app.ts)

Основна функція ініціалізації Hono-застосунку з middleware та маршрутизацією:

```typescript
export function createApp(): Hono<{ Variables: Variables }> {
  const app = new Hono<{ Variables: Variables }>();
  const service = new authService();

  app.use("*", secureHeaders());
  app.use("*", createRateLimiter({ windowMs: 15 * 60 * 1000, max: 200 }));
  app.use("*", compress());

  app.use("*", async (c, next) => {
    const token = getCookie(c, "auth-token");
    if (!token) {
      c.set("isLoggedIn", false);
      await next();
      return;
    }

    try {
      const payload = verify(token, getJwtSecret());
      if (await isTokenRevoked(payload)) {
        c.set("isLoggedIn", false);
        await next();
        return;
      }
      const user = await service.getUserById(payload.userId);
      if (!user) {
```

```
      c.set("isLoggedIn", false);
      await next();
      return;
    }
    c.set("jwtPayload", payload);
    c.set("user", user);
    c.set("isLoggedIn", true);
  } catch {
    c.set("isLoggedIn", false);
  }
  await next();
});

app.use("*", serveStatic({ root: "./public" }));
app.use("*", staticCacheHeaders);
app.route("/", authRoute);
app.route("/", taskRoute);
app.notFound((c) => c.json({ error: "Not found" }, 404));
app.onError((error, c) => handleHttpError(error, c));
return app;
}
```

Rate Limiting Middleware (rateLimit.ts)

Реалізація обмеження кількості запитів на рівні IP-адреси:

```typescript
export function createRateLimiter(options: RateLimitOptions) {
  return async function rateLimiter(c: Context, next: Next) {
    const key = getClientKey(c);
```

```
    const now = Date.now();
    const current = rateLimitStore.get(key);

    if (!current || current.resetAt <= now) {
      rateLimitStore.set(key, { count: 1, resetAt: now + options.windowMs });
      await next();
      return;
    }

    if (current.count >= options.max) {
      const retryAfter = Math.ceil((current.resetAt - now) / 1000);
      c.header("Retry-After", String(Math.max(1, retryAfter)));
      c.header("X-RateLimit-Limit", String(options.max));
      c.header("X-RateLimit-Remaining", "0");
      c.header("X-RateLimit-Reset", String(Math.ceil(current.resetAt / 1000)));
      return c.text("Too many requests", 429);
    }

    current.count += 1;
    rateLimitStore.set(key, current);
    c.header("X-RateLimit-Remaining", String(options.max - current.count));
    await next();
  };
}
```

JWT Сесія (session.ts)

Видача та верифікація JWT-токенів з RFC 7519 payload:

```typescript
export async function createAccessToken(userId: number): Promise<string> {
  const now = Math.floor(Date.now() / 1000);
  return sign(
    {
      userId,
      jti: randomUUID(),
      type: "access",
      iat: now,
      exp: now + ONE_HOUR_SECONDS,
      iss: "capydo",
      aud: "capydo-web",
    },
    getJwtSecret(),
  );
}

export async function verifyAccessToken(
  token: string,
): Promise<JwtPayload | null> {
  try {
    return (await verify(token, getJwtSecret())) as JwtPayload;
  } catch {
    return null;
  }
}
```

```

**EJS Рендеринг з TTL-кешуванням (render.ts)**

TTL-кешування шаблонів у production-середовищі для оптимізації I/O:

```typescript
async function readTemplate(filePath: string): Promise<string> {
  const useCache = process.env.NODE_ENV === "production";
  if (!useCache) {
    return readFile(filePath, "utf-8");
  }
  const cached = templateCache.get(filePath);
  const now = Date.now();
  if (cached && cached.expiresAt > now) {
    return cached.content;
  }
  const content = await readFile(filePath, "utf-8");
  templateCache.set(filePath, { content, expiresAt: now + TEMPLATE_TTL_MS });
  return content;
}

export async function renderTemplate(
  templatePath: string,
  data: Record<string, unknown>,
  filename?: string,
): Promise<string> {
  const content = await readTemplate(templatePath);
  return ejs.render(content, data, { filename, async: true });
}
```
```

Обробка помилок (errorHandler.ts)

Централізована обробка HTTP-помилок з відповідними кодами:

```typescript
export async function handleHttpError(
  error: Error,
  c: Context,
): Promise<Response> {
  const isDev = process.env.NODE_ENV === "development";

  if (error instanceof ValidationError) {
    return c.json({ error: "Validation failed", details: error.details }, 400);
  }

  if (error instanceof AuthError) {
    return c.json({ error: "Authentication failed" }, 401);
  }

  if (error instanceof NotFoundError) {
    return c.json({ error: "Resource not found" }, 404);
  }

  if (isDev) {
    console.error("[ERROR]", error);
  }

  return c.json({ error: "Internal server error" }, 500);
}
```