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

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка web-застосунку для обліку і розподілу домашніх справ»

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

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

_____ Андрій СТЕПАНОВ
(підпис)

Виконав: здобувач вищої освіти групи ПД-43

Андрій СТЕПАНОВ

Керівник: Оксана ЗОЛОТУХІНА
к.т.н., доцент

Рецензент: _____

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Степанову Андрію Юрійовичу

1. Тема кваліфікаційної роботи: «Розробка web-застосунку для обліку і розподілу домашніх справ»
керівник кваліфікаційної роботи к.т.н., доцент Оксана ЗОЛОТУХІНА,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.

2. Строк подання кваліфікаційної роботи «02» червня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: офіційна документація Angular 19 та ASP.NET Core, матеріали щодо реалізації OAuth 2.0, наукові статті і технічна література з обліку та розподілу домашніх справ, приклади використання RxJS для реактивного програмування.

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

1. Аналіз предметної області та постановка задачі
2. Огляд існуючих рішень для обліку і розподілу домашніх справ
3. Розробка функціональних модулів системи
4. Тестування web-застосунку.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів
2. Вимоги до програмного забезпечення
3. Програмні засоби реалізації
4. Діаграма прецедентів
5. Діаграма класів
6. Схема бази даних
7. Алгоритм розподілу завдань
8. Екранні форми
9. Апробація результатів дослідження

6. Дата видачі завдання «25» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	25.02-04.03.2025	
2	Аналіз та дослідження специфіки розподілу домашніх справ з урахуванням потреб різних цільових аудиторій	05.03-13.03.2025	
3	Огляд існуючих рішень для обліку і розподілу домашніх справ, визначення функціональних та нефункціональних вимог до web-застосунку	14.03-18.03.2025	
4	Проектування web-застосунку для обліку і розподілу домашніх справ	18.03-23.03.2025	
5	Програмна реалізація web-застосунку	23.03-19.04.2025	
6	Тестування web-застосунку	20.04-28.04.2025	
7	Оформлення роботи: вступ, висновки, реферат	29.04-11.05.2025	
8	Розробка демонстраційних матеріалів	12.05-18.05.2025	
9	Попередній захист роботи	19.05-30.05.2025	

Здобувач вищої освіти _____ Андрій СТЕПАНОВ
(підпис)

Керівник
кваліфікаційної роботи _____ Оксана ЗОЛОТУХІНА
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 55 стор., 3 табл., 19 рис., 24 джерел.

Мета роботи – спрощення обліку і розподілу домашніх справ за рахунок використання web-застосунку.

Об'єкт дослідження – процес розподілу домашніх справ.

Предмет дослідження – програмне забезпечення для організації розподілу домашніх справ.

Короткий зміст роботи: У роботі проаналізовано проблеми розподілу домашніх справ та визначено вимоги до системи, що спрощує їх планування. Розроблено веб-застосунок з можливістю добровільного вибору завдань, відстеження виконання та використанням гейміфікації для підвищення мотивації. Реалізовано ключові функціональні можливості, зокрема: створення завдань із прив'язкою до конкретної дати та виконавця, а також реалізація механізмів реєстрації та авторизації користувачів. В роботі використано інструмент Figma для проєктування інтерфейсу користувача, ASP.NET Core для розробки серверної частини, Angular для реалізації клієнтської частини, Entity Framework Core для роботи з базою даних.

Сферою використання застосунку є організація обліку та розподілу домашніх завдань серед членів сім'ї або співмешканців.

КЛЮЧОВІ СЛОВА: WEB-ЗАСТОСУНОК, КЕРУВАННЯ ЗАВДАННЯМИ, ПЛАНУВАННЯ, РОЗПОДІЛ ОБОВ'ЯЗКІВ, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА, ANGULAR, ASP.NET CORE.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API (Application Programming Interface) — програмний інтерфейс застосунку.
ASP.NET Core — кросплатформний фреймворк для веб-додатків (Microsoft).
Angular — фреймворк для клієнтської частини (версія 19).
Angular Universal — інструмент для серверного рендерингу (SSR).
bcrypt — алгоритм хешування паролів.
Chart.js — бібліотека для візуалізації даних.
CSRF (Cross-Site Request Forgery) — міжсайтова підробка запитів.
DTO (Data Transfer Object) — об’єкт для передачі даних між рівнями системи.
EF Core (Entity Framework Core) — ORM для роботи з базами даних у .NET.
Figma — інструмент для дизайну інтерфейсів.
FluentEmail — бібліотека для відправки листів.
GitHub — платформа для керування версіями коду.
JWT (JSON Web Token) — токен для автентифікації.
LINQ (Language Integrated Query) — мова запитів у C#.
Luxon — бібліотека для роботи з датами/часом.
OAuth 2.0 — протокол авторизації (з PKCE).
OpenAPI — стандарт для документування REST API.
ORM (Object-Relational Mapping) — об’єктно-реляційне відображення.
PostgreSQL — реляційна СУБД.
PKCE (Proof Key for Code Exchange) — механізм захисту OAuth.
REST (Representational State Transfer) — архітектурний стиль API.
RxJS — бібліотека для реактивного програмування.
SHA-2 — алгоритм шифрування даних.
SPA (Single Page Application) — односторінковий застосунок.
SSR (Server-Side Rendering) — рендеринг на сервері.
Swiper — бібліотека для жестової навігації.
Swagger — інструмент для документування API.
UI (User Interface) — користувацький інтерфейс.
XSS (Cross-Site Scripting) — міжсайтовий скриптинг.
2FA (Two-Factor Authentication) — двофакторна автентифікація.

ВСТУП

Актуальність: Проблема розподілу домашніх справ є актуальною у всі часи. Незалежно від культурного чи історичного контексту, люди, які проживають разом, завжди потребували ефективної організації побутових завдань - таких як прибирання, приготування їжі, догляд за оселею тощо. За відсутності чіткого розподілу обов'язків можуть виникати конфлікти, нерівномірне навантаження та зниження рівня комфорту у спільному житті.

З розвитком цифрових технологій з'явилась можливість полегшити цей процес за допомогою спеціалізованих програмних рішень. Зокрема, web-застосунки дають змогу забезпечити доступ до системи обліку справ з будь-якого пристрою, у будь-який час, і дозволяють кільком користувачам взаємодіяти в межах однієї спільноти. Це дозволяє не лише оптимізувати побутові процеси, а й підвищити загальний рівень дисципліни та відповідальності серед користувачів.

З появою сучасних технологій стало можливим автоматизувати й цей аспект повсякденного життя. Використання веб-застосунків дозволяє ефективно управляти спільними завданнями, вести облік їх виконання та справедливо розподіляти відповідальність між усіма учасниками.

Об'єктом дослідження є процес розподілу домашніх справ.

Предметом дослідження є програмне забезпечення для організації розподілу домашніх справ.

Метою роботи є спрощення обліку і розподілу домашніх справ за рахунок використання web-застосунку.

Для досягнення поставленої мети в рамках бакалаврської роботи виконано наступні задачі:

1. Провести аналіз предметної області обліку домашніх справ з метою виявлення ключових проблем і потреб користувачів у сфері створення, розподілу та контролю виконання завдань.

2. Дослідити існуючі програмні рішення для управління домашніми справами, визначити їхні функціональні можливості та обмеження для формування обґрунтованих вимог до розроблюваної системи.

3. Сформулювати функціональні та нефункціональні вимоги до веб-застосунку, що забезпечують надійність, безпеку, зручність та ефективність управління домашніми справами.

4. Розробити архітектуру системи, враховуючи особливості предметної області, та створити структуру бази даних, що підтримує збереження, обробку і розподіл домашніх завдань.

5. Реалізувати ключові функціональні модулі веб-застосунку, зокрема автентифікацію користувачів, механізми створення, призначення та моніторингу виконання домашніх справ.

6. Провести функціональне тестування розробленої системи для перевірки коректності, стабільності та відповідності реалізованого функціоналу поставленим вимогам.

Методи дослідження:

- аналіз літературних та інтернет-джерел;
- синтез інформації;
- порівняльний аналіз;
- моделювання;
- експериментальна перевірка.

Практична значущість результатів полягає в можливості застосування розробленого web-застосунку для організації та контролю домашніх обов’язків у різнорівневих сімейних чи житлових колективних структурах, забезпечуючи адаптивність та зручність використання на різних пристроях.

1 ДОСЛІДЖЕННЯ ТЕМАТИКИ ТА ПРОБЛЕМНОЇ ОБЛАСТІ

1.1 Актуальність автоматизації побутових обов’язків

Організація побуту та рівномірний розподіл домашніх обов’язків завжди були важливими аспектами гармонійного життя як для окремих сімей, так і для спільнот, які проживають разом [1]. Домашні справи — це не лише прибирання, готування чи закупівля продуктів, а широкий спектр завдань, що потребують координації, планування й відповідальності. Задача організації ефективного розподілу обов’язків між декількома учасниками є частиною ширшого класу проблем групового планування і колаборативного управління. Такі задачі типові не тільки для домогосподарств, а й для гуртожитків, волонтерських ініціатив, малих колективів без чіткої формальної ієрархії.

Згідно з дослідженнями (зокрема, Хічій О.) розподіл домашніх справ є однією з основних причин сімейних конфліктів [2]. Це пов’язано з тим, що неефективний або несправедливий розподіл завдань призводить до накопичення незадоволення, виникнення напруженості між учасниками спільноти, а також до падіння загальної продуктивності.

Сучасні підходи, які як усні домовленості, ведення нотатників або розкладів на папері, є недостатньо гнучкими в умовах динамічного життя, коли плани змінюються на ходу, а учасники групи можуть мати різні графіки та пріоритети. Проблеми посилюються, коли група збільшується або коли частина учасників має обмежений рівень цифрової чи організаційної грамотності.

У цих умовах автоматизація процесів розподілу й обліку домашніх справ набуває особливої актуальності. Інфокомунікаційні технології — web- та мобільні застосунки, чат-боти, хмарні сервіси — стають важливими інструментами, які дозволяють синхронізувати інформацію між усіма учасниками, мінімізувати людські помилки, забезпечити гнучкість управління завданнями та підвищити прозорість процесів. Застосування push-сповіщень, інтеграція з платформами

комунікації (Slack, Telegram, Microsoft Teams) дають змогу розширити канали взаємодії, зробивши систему доступною з будь-якого пристрою та у будь-який час.

Важливо зазначити, що сама по собі автоматизація не гарантує успішного вирішення проблеми. Вибір правильного підходу до розподілу завдань має ключове значення:

− Централізований підхід (адміністратор призначає завдання) ефективний для контролю й забезпечує рівномірний розподіл ресурсів, особливо у великих командах чи на етапах кризового менеджменту. Проте такий підхід може призводити до формування ієрархічного тиску, зниження відчуття автономності, втрати внутрішньої мотивації учасників, а також зниження якості горизонтальної комунікації всередині команди, оскільки учасники орієнтуються лише на розпорядження зверху.

− Децентралізований підхід (учасники самі обирають завдання) стимулює автономність, відповідальність і особисту ініціативу. Це може покращувати внутрішню мотивацію та залученість, дозволяючи учасникам працювати над тим, що їм цікаво або в чому вони компетентні. Водночас такий підхід вимагає додаткових механізмів захисту від зловживань — наприклад, уникнення неприємних або складних задач, надмірної концентрації на легких чи престижних завданнях або дисбалансу навантаження між членами команди.

− Ротаційні й алгоритмічні підходи (розподіл завдань за попереднім навантаженням, часом, складністю, пріоритетом) дають змогу автоматизувати процес розподілу, роблячи його більш об'єктивним та прогнозованим. Вони враховують накопичену статистику (хто, коли і скільки працював), що дозволяє рівномірно розподіляти як прості, так і складні задачі. Проблема полягає в тому, що такі підходи менш ефективні на стартових етапах (коли дані ще не накопичені), а також вимагають ретельного налаштування алгоритмів і постійного моніторингу їхньої роботи.

− Гейміфікація (бали, досягнення, рейтинги, бейджі) допомагає підвищити залученість учасників, роблячи навіть рутинні завдання цікавішими та викликаючи позитивні емоції. Вона створює додатковий шар мотивації —

соціальної (рейтинг), внутрішньої (відчуття прогресу) або зовнішньої (нагороди, бонуси). Проте ефективність гейміфікації залежить від правильної побудови системи винагород і від того, наскільки вона відповідає культурі команди: не всі люди однаково мотивовані рейтингами чи бейджами, а надмірне змагання може породжувати конфлікти.

Розроблений веб-застосунок передбачає унікальний підхід до автоматизації: завдання створюються у системі й залишаються прихованими (розмитими), а зміст стає доступним тільки після добровільного вибору учасником. Таким чином, мінімізується суб’єктивний вплив при розподілі обов’язків, знижується ризик конфліктів, усувається потреба у прямому керуванні, що робить систему більш справедливою й гнучкою.

Окремої уваги заслуговує роль автоматизації з точки зору цифрової трансформації побуту. У сучасному суспільстві дедалі більше рутинних процесів перекладаються на технологічні рішення: від автоматизованих систем оплати рахунків до смарт-гаджетів для управління домашньою технікою. У цьому контексті застосунок для розподілу обов’язків є логічним кроком уперед — створенням цифрового помічника, який не лише полегшує життя, а й покращує міжособистісні взаємини.

На відміну від існуючих систем, орієнтованих переважно на контроль чи нагадування (наприклад, ToDo-додатки), запропоноване рішення орієнтоване на мотивацію й справедливий розподіл, з урахуванням реального стану навантаження учасників та без нав’язування завдань згори.

Таким чином, актуальність розробки веб-застосунку для обліку та розподілу домашніх справ полягає в:

− Соціальній значущості задачі. Система розподілу домашніх або групових завдань спрямована не лише на виконання практичних дій, але й на покращення гармонії та комунікації всередині малих груп. Вона допомагає формувати відчуття спільної відповідальності, підтримувати рівновагу між учасниками, знижувати рівень конфліктності й запобігати накопиченню невдоволення через нерівномірне навантаження. Таким чином, проєкт має

соціальну місію — зміцнення довіри, співпраці й взаємоповаги в побутових або робочих середовищах;

− Практичній важливості. Ефективність таких рішень проявляється у підвищенні загальної продуктивності групи завдяки прозорому та контрольованому розподілу завдань. Система робить процес гнучким (адаптуючи завдання під обставини або зміни складу групи) та прозорим (кожен бачить, хто й що робить, без прихованих очікувань чи непомічених зусиль). Це сприяє зменшенню дублювання завдань, втрат часу та ресурсів, а також допомагає уникнути ситуацій, коли частина завдань залишається без уваги;

− Технологічному потенціалі. Проєкт відкриває можливості впровадження сучасних підходів до автоматизації — від алгоритмічного розподілу навантаження й обліку статистики до впровадження гейміфікації (бали, бейджі, рейтинги, досягнення). Такі інструменти підвищують залученість користувачів, роблять взаємодію із системою цікавішою й забезпечують додаткову мотивацію для добровільної участі. Крім того, використання мобільних застосунків, чат-ботів або вебплатформ дозволяє інтегрувати систему у повсякденне життя учасників;

− Масштабованості. Система має високий потенціал для масштабування та адаптації до різних контекстів. Вона може застосовуватись не лише в межах окремої сім’ї чи гуртожитку, але й у волонтерських ініціативах, студентських клубах, невеликих робочих колективах, а також у коворкінгах чи спільнотах за інтересами. Завдяки гнучкій архітектурі й модульному підходу рішення можна легко переналаштувати під нові задачі, сценарії та кількість учасників.

1.2 Огляд і аналіз існуючих рішень

У цьому розділі проведено аналіз сучасних програмних продуктів, які призначені для обліку та розподілу домашніх обов’язків. Розглянуті додатки дозволяють створювати персоналізовані графіки прибирання та розподіляти завдання серед членів сім’ї або інших спільнот, що сприяє підвищенню ефективності виконання побутових обов’язків.

1.2.1 Аналіз застосунку Sweepy

Sweepy (https://play.google.com/store/apps/details?id=app.sweepy.sweepy) - програмний продукт, розроблений компанією Appsent, який забезпечує можливість створення індивідуальних графіків прибирання та автоматичного розподілу домашніх завдань між учасниками родини [4]. Основною метою застосунку є підвищення регулярності виконання побутових обов'язків та мотивація членів сім'ї до їх своєчасного виконання. Рисунок 1.1 відображає приклад застосунку Sweepy.

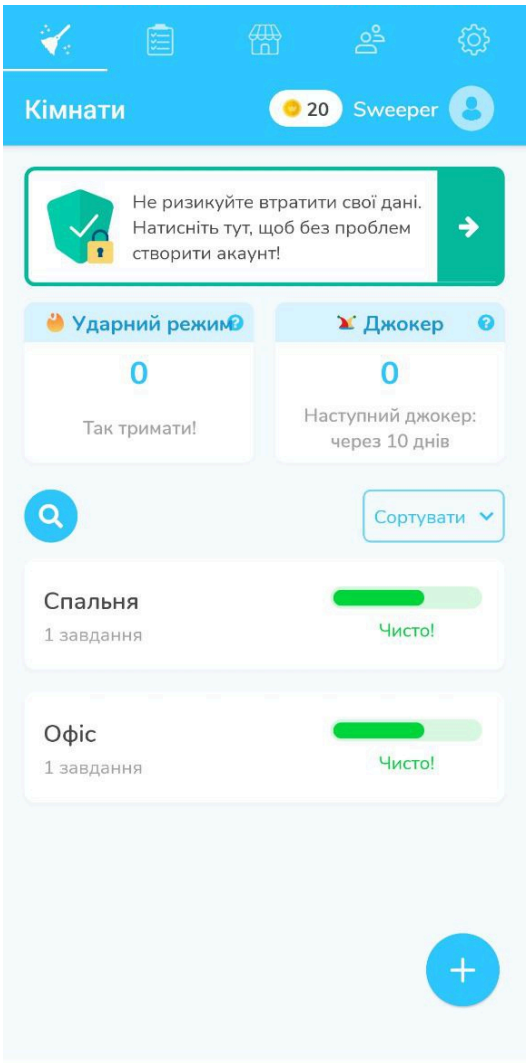


Рис. 1.1 Приклад інтерфейсу програми Sweepy

Основні переваги програмного продукту Sweepy полягають у його продуманному дизайні, спрямованому на підтримку мотивації та підвищення залученості користувачів. Серед ключових елементів виділяються гейміфікаційні механізми, які значно збагачують користувацький досвід. Це, зокрема, режим викликів, де користувачі можуть змагатися один з одним, приймаючи спеціальні завдання, що вимагають додаткової активності чи креативності. Система досягнень заохочує регулярність і поступове покращення: користувачі отримують значки, нагороди або бонуси за виконання певних умов, наприклад, прибирання без пропусків протягом тижня. Таблиця лідерів стимулює дружню конкуренцію всередині груп, роблячи процес більш цікавим та соціально залученим.

Важливим доповненням є віртуальний магазин та внутрішня валюта, які дозволяють користувачам обмінювати зароблені бали на віртуальні винагороди, персоналізувати застосунок або розблоковувати додаткові функції. Також варто відзначити спеціальні допоміжні засоби, що роблять використання Sweepy зручнішим: це можуть бути як інструменти для планування, так і мотиваційні нагадування, що підвищують ефективність роботи.

Ще однією сильною стороною є підтримка локалізації на восьми мовах, яка забезпечує доступність застосунку для користувачів з різних культурних і мовних середовищ. Це дозволяє Sweepy бути присутнім на глобальному ринку, а також сприяє інтеграції в мультикультурні родини чи колективи, де кілька мов використовуються паралельно.

Крім того, режим відпустки значно підвищує гнучкість застосунку. Користувачі можуть тимчасово відключатися від системи, не побоюючись втрати балів або псування статистики, що робить Sweepy більш дружнім до реального життя, де люди періодично змінюють свої розклади чи відсутні з об’єктивних причин. Не менш важливим є віртуальний талісман, який додає елемент гри та персоналізації. Такий персонаж не лише розважає, але й допомагає — наприклад, підказками, нагадуваннями чи позитивним підкріпленням, що формує емоційний зв’язок із застосунком.

Недоліки Sweepy, попри його численні переваги, теж потребують уваги. Насамперед, обмеження функціоналу лише на завданнях прибирання звужує сферу його використання. Це означає, що застосунок не підходить для ширших сценаріїв організації побутових справ, таких як планування закупівель, ремонтних робіт, догляду за рослинами чи хатніми тваринами, що могло б залучити ширшу аудиторію.

Відсутність розширеної статистики обмежує можливості користувачів для аналізу своєї продуктивності. Було б корисно мати докладні дані про виконані завдання, час, витрачений на них, рівень складності, зміни у прогресі з часом, порівняння між різними членами команди або навіть рекомендації для оптимізації роботи. Такий функціонал зробив би Sweepy не просто ігровим застосунком, а потужним аналітичним інструментом для підвищення ефективності.

Ще один недолік полягає в надмірній залежності мотивації користувачів від гейміфікації. Якщо користувачеві перестають бути цікавими внутрішні бали, значки чи таблиці лідерів, існує ризик, що він втратить мотивацію виконувати завдання, оскільки застосунок не формує внутрішніх стимулів або глибоких звичок. Це робить утримання користувачів більш складним у довгостроковій перспективі.

Крім того, відсутність підтримки багатопросторової роботи — тобто можливості одночасного управління завданнями для кількох окремих локацій (наприклад, квартири й дачі, кількох кімнат у гуртожитку або декількох офісних приміщень) — знижує гнучкість Sweepy для тих, хто має складнішу структуру обов’язків. Додаткова функція розділення груп чи просторів могла б значно розширити потенційні сценарії використання.

1.2.2 Аналіз застосунку Tidy: House Cleaning Schedule

Tidy: House Cleaning Schedule (https://play.google.com/store/apps/details?id=com.cklab.cleaning) - мобільний додаток для платформи Android, що допомагає створювати графіки прибирання

шляхом поділу великих завдань на менші підзадачі [5]. Вміст рисунка 1.2 демонструє інтерфейс програми Tidy.

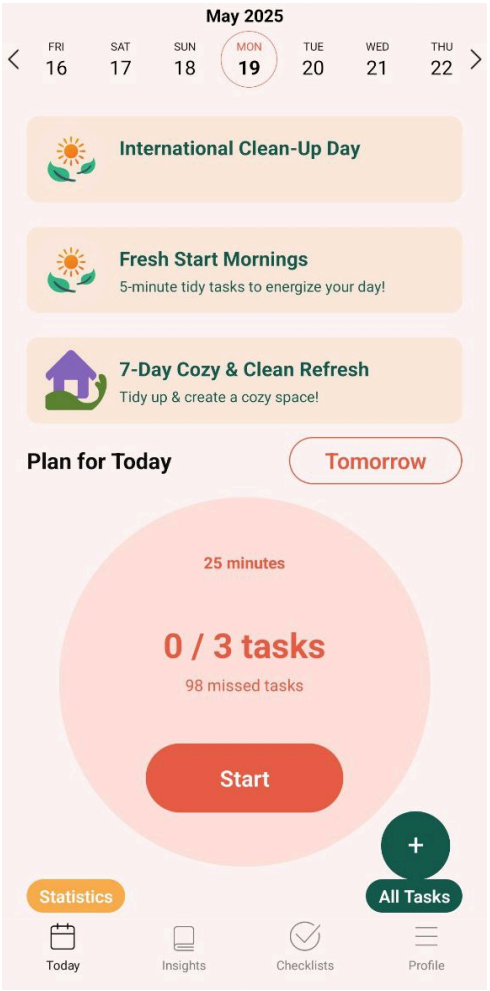


Рис. 1.2 Приклад інтерфейсу програми Tidy

Основні переваги програмного продукту:

По-перше, система пропонує інтерактивні поради у форматі сторіс, які значно полегшують процес адаптації нових користувачів. Завдяки такому сучасному і зрозумілому формату навчання, користувачі швидко отримують корисну інформацію, що сприяє комфортному старту і зменшує бар’єр входу у використання застосунку. Цей підхід робить навчання живим, цікавим і ненав’язливим.

Для ще більшого прискорення початку роботи в застосунку передбачена наявність готових шаблонів завдань, які можна легко адаптувати під власні потреби. Це дозволяє уникнути зайвих налаштувань і швидко організувати робочий процес, особливо для користувачів, які не хочуть витрачати час на створення завдань з нуля.

Гейміфікація — важливий елемент мотивації, що реалізований у вигляді системи досягнень за виконані дії. Користувачі отримують нагороди, значки або бали за регулярність, відповідальність та активність, що стимулює їх залишатися залученими і підвищує інтерес до рутинних завдань. Це перетворює буденне прибирання чи інші домашні справи у більш захопливий процес.

Ще одним корисним інструментом є таймер для підвищення концентрації, який допомагає розподілити робочий час ефективніше. Такий таймер підтримує користувача у фокусі, запобігає прокрастинації і допомагає виконувати завдання в короткі проміжки часу з максимальною продуктивністю [6].

Важливою функцією є і підтримка режиму відпустки, що дозволяє враховувати дні, коли користувач відсутній, без негативного впливу на загальну статистику та мотивацію інших учасників. Це робить застосунок більш гнучким і пристосованим до реального життя, де люди мають непередбачувані графіки.

Серед недоліків варто відзначити обмежені можливості персоналізації завдань. Хоча шаблони й прискорюють старт, користувачі можуть відчувати нестачу гнучкості у створенні завдань під свої унікальні потреби або специфіку. Це може знижувати зручність для тих, хто хоче тонко налаштовувати свої плани або робочі процеси.

Також варто звернути увагу на відсутність розширеної статистики, яка б дозволила користувачам детально аналізувати свої результати, бачити динаміку прогресу, порівнювати показники між різними періодами або учасниками. Наявність глибокої аналітики підвищила б цінність застосунку для користувачів, які прагнуть систематично покращувати свою продуктивність.

Крім того, наразі немає підтримки багатопросторової роботи, тобто можливості одночасно управляти завданнями в різних фізичних локаціях або

окремих групах. Це обмежує застосунок у випадках, коли користувачам потрібно розмежовувати обов'язки між кількома помешканнями, кімнатами, офісами чи іншими просторами. Така функціональність значно підвищила б гнучкість та масштабованість продукту.

1.2.3 Аналіз застосунку Nipto

Nipto (https://play.google.com/store/apps/details?id=com.nipto.niptoapp) - мобільний додаток, орієнтований на перетворення домашніх обов'язків у гру для сімей, пар або співмешканців [7]. На рисунку 1.3 зображено вікно Nipto.

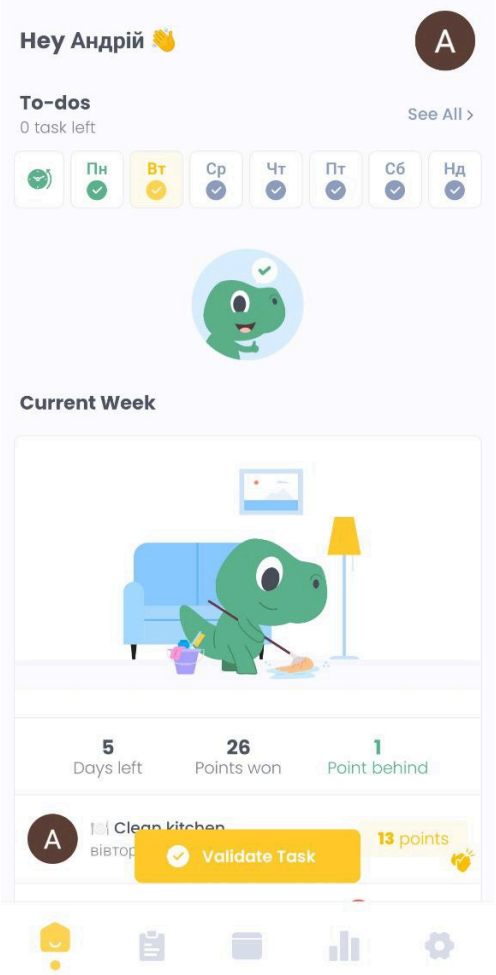


Рис. 1.3 Приклад інтерфейсу програми Nipto

Однією з ключових особливостей є гейміфікація через систему балів та нагород, яка стимулює мотивацію користувачів за рахунок створення приємних психологічних підкріплень. Користувачі отримують бали, значки, трофеї чи інші нагороди за виконання завдань, що допомагає перетворити рутинні обов’язки на захопливий процес, а також підтримує регулярність і дисципліну.

Застосунок вирізняється гнучкими налаштуваннями, які дозволяють користувачам обирати між режимами конкуренції (де учасники змагаються один з одним) або режимами індивідуальних цілей (де кожен орієнтується на власні результати без зовнішнього порівняння). Така гнучкість забезпечує комфорт як для людей, що люблять змагатися, так і для тих, хто воліє працювати самостійно у власному темпі.

Особливу увагу приділено орієнтації на сімейне користування, зокрема впровадженню підтримки дитячих акаунтів. Це дозволяє залучати дітей до спільних справ, виховувати у них відповідальність та створювати позитивну динаміку всередині сім’ї, де кожен член має свій вклад у спільний добробут.

Також варто відзначити наявність режиму відпустки, завдяки якому можна тимчасово виключати певних користувачів із розподілу завдань без впливу на загальні статистичні показники чи систему нагород. Це дуже зручно для сімей або команд, де хтось час від часу перебуває у від’їзді.

Ще однією сильною стороною є розширена статистика виконання завдань, яка дозволяє користувачам аналізувати свій прогрес, бачити детальну історію виконаних дій, визначати, які завдання найчастіше залишаються невиконаними, і загалом краще планувати свої обов’язки. Така аналітика сприяє більш свідомому підходу до організації спільної роботи.

Серед недоліків варто виділити відсутність підтримки кількох віртуальних просторів, що обмежує можливість ефективного розподілу груп користувачів. Це означає, що наразі всі учасники перебувають в одному загальному середовищі, без можливості розділити, наприклад, завдання для сім’ї, окремого гуртожитку чи волонтерської групи. Така функція могла б значно розширити масштаб застосування продукту.

Ще один недолік — складний інтерфейс, який може ускладнювати користування, особливо для новачків або менш досвідчених користувачів. Перевантаженість функціями, недостатньо інтуїтивні переходи між розділами чи відсутність зрозумілих підказок можуть знижувати загальну зручність та створювати бар'єри для залучення ширшої аудиторії.

1.2.4 Аналіз застосунку Tody

Tody (https://play.google.com/store/apps/details?id=com.looploop.tody) - мобільний застосунок для платформ iOS та Android, що допомагає організовувати та оптимізувати домашні справи [8]. Рисунок 1.4 ілюструє вигляд Tody.

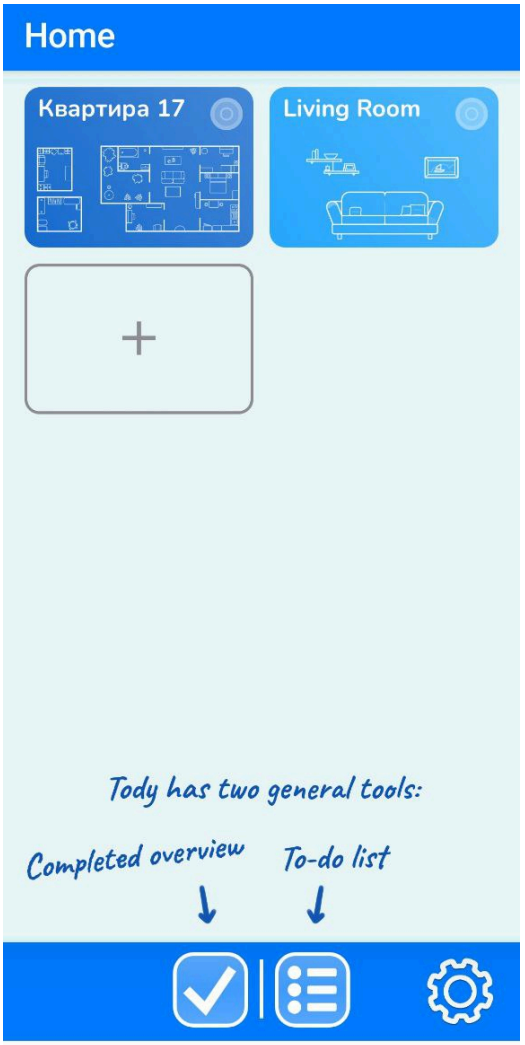


Рис. 1.4 Приклад інтерфейсу програми Tody

Однією з ключових переваг є можливість зонування обов’язків за кімнатами, ділянками або напрямками, що дозволяє більш чітко розподіляти завдання між учасниками відповідно до конкретних зон відповідальності. Це особливо корисно для великих домогосподарств або команд, де важливо не тільки знати, що потрібно зробити, але й де. Такий підхід забезпечує більш точне планування й запобігає плутанині.

Ще одна сильна сторона — використання шаблонів завдань, які допомагають швидко формувати план робіт без необхідності вручну прописувати всі деталі кожного разу. Це економить час, підвищує ефективність організації й дозволяє сфокусуватися на виконанні, а не на плануванні.

Застосунок надає деталізовану аналітику виконання завдань, що допомагає користувачам відслідковувати свій прогрес, виявляти слабкі місця, аналізувати розподіл часу та навантаження. Така статистика є цінним інструментом для вдосконалення командної взаємодії та підвищення загальної продуктивності.

Особливо варто підкреслити різноманітність інструментів, серед яких — таймер для концентрації, функція випадкового вибору завдань (для тих, хто не хоче самостійно приймати рішення), система оцінок (яка дозволяє відзначати якість виконання) та підтримка режиму відпустки (який зручно відключає тимчасово неактивних учасників від розподілу завдань без порушення загальної статистики).

Серед помітних обмежень — відсутність підтримки кількох контекстів або просторів, що означає, що застосунок не дозволяє одночасно управляти завданнями у різних групах або середовищах (наприклад, окремо для дому, офісу, волонтерського проєкту тощо). Це суттєво обмежує масштабованість продукту, оскільки користувачам доводиться або створювати кілька окремих акаунтів, або змішувати завдання в одному загальному списку.

Також важливий недолік — обмежений доступ, оскільки додаток підтримує лише один пристрій. Це створює проблеми для колективного використання: члени

групи не можуть паралельно синхронізувати свої дії з різних пристроїв, що знижує загальну гнучкість і ефективність, особливо для командної роботи.

На основі проведеного аналізу основних програмних продуктів, призначених для обліку та розподілу домашніх справ, доцільно виконати їх порівняння за ключовими характеристиками. Це дозволить виявити сильні та слабкі сторони кожного рішення, а також визначити, яким чином розроблюваний програмний продукт (назва власної програми) може заповнити існуючі прогалини або вдосконалити наявний функціонал. Зведені результати порівняння наведено у таблиці 1.1.

Таблиця 1.1

Порівняння характеристик аналогів та розроблюваного програмного продукту

	Sweepy	Tidy	Nipto	Tody	Rozpodil
Рулетка відповідального	-	-	-	+	+
Підтримка кількох просторів	-	-	-	-	+
Приховані завдання	-	-	-	-	+
Призначення відповідального	-	-	+	+	+

1.3 Функціональні та нефункціональні вимоги до системи

На початкових етапах розробки програмного забезпечення формалізація вимог до системи є критично важливим завданням. Чітке та повне визначення вимог дозволяє забезпечити відповідність розроблюваного продукту очікуванням кінцевих користувачів, знизити ризики виникнення помилок і непорозумінь між

учасниками проєкту, а також створити передумови для стабільної експлуатації програмного забезпечення після впровадження.

Вимоги до системи поділяються на дві основні категорії: функціональні та нефункціональні. Функціональні вимоги описують, яку поведінку має реалізовувати система, які сервіси повинні бути доступними та які задачі виконуватись. Вони включають опис сценаріїв взаємодії користувачів із системою, специфікацію оброблюваних даних, опис бізнес-логіки та умови виконання операцій.

Нефункціональні вимоги визначають характеристики якості системи, що не пов’язані безпосередньо з її функціональністю, але мають значний вплив на користувацький досвід та експлуатаційні властивості продукту. До них відносяться вимоги до продуктивності, масштабованості, надійності, безпеки, сумісності, зручності використання, доступності та підтримуваності.

Якісне документування обох типів вимог є необхідною умовою для ефективного проєктування архітектури системи, оптимального планування ресурсів, організації процесів тестування та проведення верифікації й валідації розробленого продукту. Вимоги повинні бути конкретними, однозначними, вимірюваними та перевірюваними, що забезпечує можливість їхньої подальшої реалізації та контролю.

Систематичне управління вимогами, що передбачає їхню ідентифікацію, аналіз, документування, трасування змін і контроль версій, є ключовим елементом інженерії програмного забезпечення та сприяє досягненню високої якості кінцевого продукту.

1.3.1 Функціональні вимоги

- Призначення домашніх справ користувачам

Призначення домашніх справ користувачам є центральною функцією системи, адже вона орієнтована на організацію розподілу обов’язків між користувачами, зокрема всередині однієї родини або невеликої команди. Для кожного завдання необхідно вказати виконавця, термін виконання (дедлайн),

статус (наприклад, заплановано, в процесі, завершено), а також опис. Система передбачає можливість призначення завдань як собі, так і іншим користувачам, забезпечує перегляд повного списку завдань, розподілених між усіма членами групи, а також дозволяє змінювати виконавців чи статуси завдань у разі зміни обставин. Така функціональність є ключовою, оскільки без неї продукт не може виконати свою основну задачу — ефективну координацію домашніх обов’язків.

- Гейміфікація процесу розподілу завдань

Гейміфікація особливо актуальна в контексті сімейного використання, коли залучення дітей чи підлітків може бути ефективнішим завдяки ігровим механікам.

- Перегляд, створення домашніх справ

Користувач повинен мати повний контроль над створенням справ. Важливо забезпечити простоту додавання нової справи (одним кліком), можливість вказати опис, відповідального, дедлайн і тип справи.

- Керування ролями користувачів

Система повинна реалізовувати ролі, серед яких хост (host) виступає головним користувачем із повним та виключним контролем над домогосподарством (простором): він створює простір, запрошує інших учасників, призначає їм ролі, а також має право змінювати або видаляти будь-які дані; його права є найвищими й не можуть бути обмежені іншими користувачами. Адміністратор отримує повний доступ до всіх функцій системи, включаючи управління ролями інших користувачів, створення й редагування завдань, а також налаштування системи. Звичайний користувач має змогу переглядати власні завдання та виконувати призначені йому обов’язки. Таке розмежування ролей дає змогу підтримувати контроль за роботою системи, мінімізувати ризики конфліктів і помилок під час розподілу відповідальності.

- Реєстрація та авторизація користувачів

Система має забезпечувати створення індивідуальних облікових записів користувачів за допомогою email та пароля, а також надавати механізм авторизації для збереження прогресу й персоналізації даних. Важливо передбачити можливість відновлення пароля у разі його втрати. Для підвищення зручності

користувачів система також повинна підтримувати авторизацію через сторонні сервіси, такі як Google або Facebook. Цей функціонал є фундаментом багатокористувацької роботи та надійного збереження персональної інформації.

1.3.2 Нефункціональні вимоги

- Шифрування чутливих даних

Для захисту персональної інформації (імена, паролі, електронні адреси) необхідно реалізувати сучасні методи шифрування. Рекомендується використовувати:

- алгоритми хешування паролів, такі як bcrypt, Argon2 або PBKDF2;
- захищені канали зв’язку (HTTPS, TLS);
- захист від атак типу SQL Injection, XSS, CSRF.

Виконання цих вимог гарантує відповідність основним принципам інформаційної безпеки.

- Інтерфейс, доступний для людей з обмеженими можливостями

Система повинна відповідати сучасним стандартам доступності, таким як WCAG 2.1, що передбачає:

- масштабування шрифтів;
- доступність для екранних читачів (screen readers);
- наявність текстових альтернатив для графічних елементів;
- можливість керування з клавіатури.

Це підвищує соціальну значущість продукту та розширює коло потенційних користувачів.

- Швидкість завантаження інтерфейсу

Інтерфейс системи має завантажуватись за час, що не перевищує 5 секунд при швидкості з’єднання 20 Мбіт/с. Це особливо важливо для мобільних користувачів, які можуть мати обмежені ресурси пристрою. Для цього створити:

- використання кешування;
- мінімізація обсягу даних, що передаються;
- оптимізація скриптів і стилів.

- Оптимізація контенту (зображень, текстів)

Усі графічні елементи мають бути стиснуті (наприклад, за допомогою WebP або іншого сучасного формату), а текстовий контент — організований таким чином, щоб його обсяг був мінімально необхідним для повноцінної роботи. Це дозволяє:

- зменшити навантаження на сервери;
- прискорити час відгуку системи;
- знизити витрати трафіку користувача.
- REST-архітектура API

Для побудови взаємодії між клієнтською частиною застосунку та сервером використовувати RESTful API. Основні принципи:

- використання стандартних HTTP-методів (GET, POST, PUT, DELETE);
- передача даних у форматі JSON;
- чітка структура URI;
- підтримка ідентифікації та авторизації через токени (наприклад, JWT).
- REST-архітектура забезпечує масштабованість, простоту інтеграції з іншими системами та підтримку багатоплатформності.

Цей комплекс вимог визначає основні рамки функціональності та технічної реалізації програмного забезпечення. Врахування як функціональних, так і нефункціональних вимог є основою якісної розробки програмного забезпечення. Функціональні вимоги описують, які саме завдання має виконувати система та як вона повинна поводитися у відповідь на дії користувача. Натомість нефункціональні вимоги визначають характеристики якості роботи: продуктивність, зручність, безпеку, масштабованість і доступність. Разом ці вимоги забезпечують створення не просто працездатного, а ефективного, зручного та надійного рішення. Такий підхід дозволяє системі відповідати сучасним очікуванням користувачів і витримувати навантаження реального середовища.

2 КОНЦЕПТУАЛЬНА АРХІТЕКТУРА І ТЕХНОЛОГІЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Загальна архітектура системи

Архітектура розробленого веб-застосунку побудована відповідно до принципів чистої (лукової) архітектури. Такий підхід дозволяє максимально розділити бізнес-логіку та зовнішні інтеграції, що є ключовим для створення підтримуваних, масштабованих і надійних програмних рішень. В основі цієї архітектури лежить принцип інверсії залежностей, що гарантує, що зовнішні рівні залежать від внутрішніх, але не навпаки. Це забезпечує можливість легкої заміни зовнішніх сервісів, бібліотек та фреймворків без зміни ядра системи. Архітектура передбачає наявність шести логічно розділених рівнів, кожен з яких має свою чітку роль, обов’язки та залежності.

2.1.1 Рівень представлення (UI Layer)

Цей рівень реалізований за допомогою Angular 19 і є основним каналом взаємодії користувача із системою. Він включає односторінковий застосунок (SPA), реалізований на основі сучасних підходів Angular, зокрема Standalone-компонентів, Signals для реактивного керування станом, а також Angular Universal для серверного рендерингу сторінок. Крім того, застосовуються такі бібліотеки, як RxJS (для обробки асинхронних потоків даних), Chart.js (для побудови інтерактивних діаграм та графіків), Swiper (для створення слайдерів і жестової навігації) та Luxon (для роботи з датами й часом). Особлива увага приділяється адаптивності інтерфейсу, що гарантує комфортну роботу як на десктопах, так і на мобільних пристроях.

2.1.2 API-рівень (Web API Layer)

Цей рівень побудований на основі ASP.NET Core Web API. Він є мостом між клієнтською частиною й серверною логікою, забезпечуючи обробку HTTP-запитів, авторизацію користувачів за допомогою JWT, валідацію вхідних даних та серіалізацію/десеріалізацію об’єктів. Документування API виконується засобами Swagger, що дозволяє легко інтегрувати систему з зовнішніми сервісами, тестувати кінцеві точки та формувати OpenAPI-специфікації. Кожен контролер на цьому рівні відповідає за конкретну предметну область, наприклад, керування завданнями, обліком користувачів або повідомленнями.

2.1.3 Прикладний рівень (Application Layer)

Тут реалізовані сценарії використання (use cases) — логіка, що визначає порядок виконання бізнес-правил залежно від запитів користувача. Цей рівень містить сервіси, які трансформують DTO (Data Transfer Objects), викликають доменні сервіси та координують роботу з інфраструктурою. Прикладний рівень також відповідає за централізацію валідації бізнес-правил, логічне групування операцій та делегування взаємодії з іншими рівнями системи.

2.1.4 Доменний рівень (Domain Layer)

Це ядро системи, де зосереджена бізнес-логіка. Доменний рівень складається з доменних сутностей (наприклад, Task, User, Household), агрегатів, сервісів і репозиторіїв, що оголошені через інтерфейси. Він не має зовнішніх залежностей, що дозволяє йому бути незалежним від деталей реалізації. Саме тут формуються головні бізнес-правила, інваріанти, правила валідації й взаємодії між сутностями. Цей рівень легко покривається модульними тестами, оскільки він не залежить від середовища виконання, баз даних чи сторонніх сервісів.

2.1.5 Рівень доступу до даних (Persistence Layer)

Доступ до даних реалізовано за допомогою Entity Framework Core, що забезпечує ORM-зв’язок із PostgreSQL. Цей рівень містить конкретні реалізації

репозиторіїв, конфігурації моделей через Fluent API, застосування міграцій та виконання запитів до БД. PostgreSQL обрано як надійну, масштабовану та багатофункціональну СУБД з підтримкою транзакцій, складних запитів і зручних механізмів індексації.

2.1.6 Інфраструктурний рівень (Infrastructure Layer)

Цей рівень реалізує зовнішні інтеграції, зокрема:

- надсилання електронних листів (FluentEmail);
- обробку токенів аутентифікації (JWT Bearer, Microsoft.IdentityModel.Tokens);
- серіалізацію JSON-даних (Newtonsoft.Json).

Інфраструктурні компоненти використовуються прикладним рівнем через оголошені інтерфейси, що дозволяє уникати прямої залежності від зовнішніх бібліотек.

Принципи побудови:

- Інверсія залежностей: залежності організовані у напрямку від зовнішніх рівнів до внутрішніх.
- Тестованість: ключові рівні (доменний та прикладний) легко покриваються модульними тестами.
- Розширюваність: нові сценарії та зовнішні сервіси можуть додаватися без порушення структури.
- Модульність: чіткий поділ відповідальностей дозволяє змінювати або оновлювати окремі компоненти без зміни всієї системи.

Обрана архітектура забезпечує відповідність сучасним вимогам до надійних, масштабованих і підтримуваних веб-застосунків. Завдяки застосуванню принципів чистої архітектури досягнуто високого рівня структурованості коду та забезпечено можливість подальшого розвитку системи.

2.2 Обґрунтування вибору технологій

Під час реалізації веб-застосунку для обліку та розподілу домашніх справ було прийнято рішення використовувати сучасний і перевірений технологічний стек, який забезпечує високу продуктивність, масштабованість, зручність розробки та відповідність принципам чистої архітектури. Вибір кожної технології є результатом аналізу вимог до системи, а також досвіду промислового використання подібних рішень у сучасній веб-розробці.

2.2.1 Frontend: Angular + TypeScript

Angular (версія 19) обрано як основний фреймворк для розробки клієнтської частини завдяки його модульності, підтримці реактивного програмування та можливості реалізації односторінкових застосунків (SPA) [9]. Angular надає вбудовану систему маршрутизації, сервісів для HTTP-запитів, механізми прив’язки даних, декларативні шаблони та підтримку реактивних форм. Наявність Standalone-компонентів і Signals у версії 19 дозволяє зменшити зв’язаність коду та спростити керування станом.

TypeScript як мова програмування забезпечує статичну типізацію, що значно зменшує кількість помилок на етапі компіляції та підвищує якість коду. TypeScript також полегшує рефакторинг, покращує читабельність коду і дозволяє використовувати об'єктно-орієнтовані парадигми.

2.2.2 SSR та рендеринг: Angular Universal + Express

Для покращення швидкості завантаження сторінок і SEO-оптимізації застосовано Angular Universal, що дозволяє виконувати рендеринг HTML на сервері. Це рішення особливо ефективне для пристроїв із повільним інтернет-з’єднанням. Серверний рендеринг реалізовано на базі Express [10], легковагового та гнучкого веб-фреймворку для Node.js, що використовується як проміжний шар між Angular та HTTP-сервером.

2.2.3 Візуалізація даних: Chart.js, Swiper, Luxon

Для візуального представлення статистики та аналітики застосовано Chart.js [11], що підтримує широкий набір типів графіків та дозволяє гнучко налаштовувати зовнішній вигляд діаграм. Інтерактивні елементи інтерфейсу реалізовані з використанням Swiper [12], який дозволяє створювати адаптивні слайдери та списки з мобільною підтримкою жестів.

Бібліотека Luxon [13] забезпечує зручну роботу з датами й часом — включно з форматуванням, локалізацією та календарними обчисленнями, що критично важливо для відображення розкладу домашніх справ.

2.2.4 Серверна частина: C#, .NET 9.0, ASP.NET Core

Серверна логіка реалізована мовою C# [14] із використанням .NET 9.0 [15] — сучасної кросплатформної платформи, що поєднує високу продуктивність, безпеку та підтримку багатьох парадигм розробки. Використання ASP.NET Core [16] як бази для Web API дозволило реалізувати REST-інтерфейс із підтримкою middleware, маршрутизації, DI-контейнера та авторизації. ASP.NET Core повністю інтегрується з чистою архітектурою та має вбудовану підтримку механізмів для побудови багаторівневих систем.

2.2.5 Доступ до даних: EF Core, PostgreSQL, Newtonsoft.Json

Для доступу до реляційної бази даних обрано Entity Framework Core (EF Core) [17] — ORM, що підтримує LINQ-запити, міграції, lazy loading та інші засоби зручної роботи з моделями даних. Це дозволяє уникнути прямого написання SQL-запитів та зосередитися на бізнес-логіці. В якості СУБД використовується PostgreSQL [18], яка є потужною, стабільною та масштабованою системою з відкритим кодом, що добре інтегрується з EF Core та підтримує складні типи даних.

Для серіалізації даних у формат JSON застосовано Newtonsoft.Json, що є де-факто стандартом у .NET для роботи з JSON і дозволяє гнучко налаштовувати перетворення об’єктів.

2.2.6 Аутентифікація та безпека: JWT, Microsoft.IdentityModel.Tokens

Аутентифікація реалізована за допомогою JWT (JSON Web Token), що дозволяє здійснювати безстанційне підтвердження особи користувача з мінімальними витратами на зберігання сесій. Валідація та обробка токенів виконуються засобами бібліотеки Microsoft.IdentityModel.Tokens [19], яка забезпечує підтримку сучасних алгоритмів підпису і шифрування.

2.2.7 Супутні інструменти: FluentEmail, Swagger, Modern-normalize

Відправлення системних повідомлень (наприклад, підтвердження реєстрації чи сповіщення) реалізовано через FluentEmail [20] — бібліотеку з підтримкою SMTP та шаблонів листів. Документування REST API здійснюється за допомогою Swagger [21], який автоматично генерує OpenAPI-специфікацію, що полегшує тестування та інтеграцію з іншими клієнтами.

Для уніфікації стилів між браузерами застосовується modern-normalize [22], що забезпечує базову консистентність CSS і спрощує розробку адаптивного інтерфейсу.

Застосований стек технологій забезпечує узгоджену та ефективну взаємодію між клієнтською і серверною частинами застосунку. Використання Angular та ASP.NET Core у поєднанні з чистою архітектурою [23] дозволяє розробляти розширювану систему з чітким розділенням відповідальностей, забезпечуючи зручну підтримку, модульність та відповідність сучасним вимогам до веб-розробки.

2.3 Модель API та контроль доступу

Веб-застосунок реалізує архітектуру RESTful API, побудовану відповідно до принципів чистої архітектури. Комунікація між клієнтською та серверною частинами здійснюється за допомогою HTTP-запитів, де дані передаються у форматі JSON. API складається з низки логічно ізольованих контролерів, кожен з

яких обслуговує конкретну предметну область. Такий підхід дозволяє забезпечити чітке розділення відповідальностей, підвищити масштабованість та спростити супровід системи.

2.3.1 Структура API

Розроблена програмна система використовує архітектуру RESTful API, що забезпечує стандартизовану взаємодію між клієнтською та серверною частинами, використовуючи протокол HTTP/HTTPS і формат обміну даними JSON. Контролери API згруповано відповідно до предметних зон функціональності, що дозволяє дотримуватися принципів розділення обов'язків (Separation of Concerns) та полегшує підтримку й масштабування коду.

Таблиця 2.1

Контролери додатку

Контролер	Основні функції
AuthController	Реєстрація користувачів, авторизація, підтвердження електронної пошти, логін через зовнішні провайдери, зміна пароля, вихід із системи.
EmailController	Перевірка, чи існує email-адреса в системі.
RoomController	Створення нової кімнати (групи), приєднання користувача до кімнати за кодом, отримання інформації про кімнату.
RoomUsersController	Отримання списку користувачів кімнати, видалення користувача з кімнати.

Продовження таблиці 2.1

Контролери додатку

Контролер	Основні функції
TaskController	CRUD-операції над домашніми справами: створення, редагування, перегляд, видалення завдань у межах кімнати.
TaskStatisticsController	Отримання статистики виконання завдань у кімнаті (наприклад, загальна кількість виконаних справ або частка участі).
TokenController	Відновлення, перевірка, видалення JWT-токенів доступу.
UserController	Отримання детальної інформації про конкретного користувача.
UserRoomsController	Отримання списку кімнат, до яких належить певний користувач.

2.3.2 Основні принципи побудови API

Розробка програмного інтерфейсу (API) здійснюється з урахуванням ключових архітектурних принципів, що забезпечують масштабованість, розширюваність, зрозумілість і підтримуваність системи. Центральним архітектурним підходом виступає REST (Representational State Transfer), що є найбільш поширеною парадигмою для побудови веб-сервісів. REST передбачає суворе дотримання семантики HTTP-методів: GET використовується для отримання ресурсів, POST — для створення нових, PUT — для оновлення існуючих, DELETE — для їхнього видалення. Такий підхід гарантує

передбачуваність поведінки ендпоінтів і забезпечує високу сумісність із широким спектром клієнтських застосунків. URI формуються за принципом відображення структури ресурсів (наприклад, /api/rooms/{id}/tasks), що дозволяє логічно організувати доступ до об’єктів системи. Завдяки цьому досягається як зручність роботи для розробників, так і відповідність сучасним стандартам побудови API.

Невід’ємним компонентом побудови API є валідація вхідних даних. Усі запити, що надходять до системи, перевіряються на рівні моделей DTO (Data Transfer Object) за допомогою атрибутів валідації. Стандартні атрибути, такі як Required, StringLength, EmailAddress, RegularExpression, дозволяють швидко впроваджувати перевірки на відповідність даних бізнес-вимогам без дублювання логіки у коді. При цьому контролери залишаються «тонкими» й не перевантажуються зайвими перевірками, оскільки відповідальність за валідність даних делегується моделі. У разі порушення правил валідації система автоматично формує стандартизовану відповідь із кодом помилки 400 Bad Request і повідомленням, що чітко пояснює, яке саме поле викликало помилку. Такий підхід сприяє зменшенню ризику проникнення некоректних чи шкідливих даних до бізнес-логіки, підвищує безпеку застосунку й робить взаємодію клієнта із сервером більш передбачуваною та дружньою.

Для підвищення прозорості та зручності роботи з API використовується Swagger (OpenAPI Specification) — сучасний інструмент для автоматичного документування. Swagger інтегрується безпосередньо з кодом застосунку й автоматично формує інтерактивну документацію на основі визначених ендпоінтів, моделей, типів даних та прикладів запитів. Це дозволяє розробникам швидко ознайомлюватися зі структурою сервісу, тестувати запити безпосередньо з веб-інтерфейсу та переконуватися у правильності формування запитів і відповідей. Крім того, Swagger істотно спрощує інтеграцію з іншими системами, оскільки надає зрозумілий і стандартизований опис API, який можна використовувати для генерації клієнтських SDK або написання контрактних тестів. Наявність актуальної документації є однією з ключових ознак якісного веб-сервісу, що спрощує підтримку та масштабування проєкту.

Важливою особливістю є чітке дотримання принципу логічної ізоляції між шарами застосунку. Контролери взаємодіють лише з прикладним рівнем (Application Layer) через чітко визначені інтерфейси, не містячи всередині себе бізнес-логіки. Такий підхід дозволяє дотримуватися принципу єдиної відповідальності (Single Responsibility Principle): контролери відповідають винятково за прийом запитів, виклик відповідних сервісів і повернення результатів клієнту. Вся бізнес-логіка винесена в окремі сервіси, які можна незалежно тестувати, модифікувати чи замінювати. Це підвищує гнучкість системи, спрощує впровадження змін і дає змогу забезпечити високий рівень підтримуваності. Логічна ізоляція також сприяє впровадженню патернів проектування, таких як Dependency Injection, що дозволяє ефективно управляти залежностями між компонентами, підвищувати тестованість системи та спрощувати її налаштування.

Завдяки дотриманню описаних принципів побудови API досягається високий рівень якості архітектури, що відповідає сучасним вимогам розробки веб-застосунків. Такий підхід гарантує масштабованість рішення, зручність підтримки, розширюваність функціоналу й забезпечення прозорої взаємодії між різними компонентами системи. Усі ці аспекти разом формують надійну основу для створення програмного продукту, здатного ефективно працювати як у середовищі з невеликою кількістю користувачів, так і при масштабуванні на велику аудиторію.

2.3.3 Контроль доступу

Система використовує механізм аутентифікації на основі JWT (JSON Web Token). Після успішної авторизації користувач отримує короткотривалий access-токен, який передається у заголовку Authorization: Bearer <token> для доступу до захищених ендпоінтів. Токен містить основну інформацію про користувача, зокрема його ідентифікатор, електронну пошту та термін дії.

Валідація токенів виконується за допомогою бібліотек JWT Bearer та Microsoft.IdentityModel.Tokens, які забезпечують перевірку підпису, терміну дії та

структури токена. Схема аутентифікації побудована як безстанційна: сервер не зберігає сесій, а покладається на токени, передані клієнтом.

У поточній реалізації серверна частина не виконує централізованої перевірки ролей, оскільки ролі користувача є контекстно-залежними — тобто можуть відрізнятись у різних кімнатах. Відповідно, перевірка повноважень (наприклад, чи є користувач власником кімнати або має право змінювати завдання) делегується клієнтській частині застосунку, яка отримує інформацію про роль користувача в рамках конкретної кімнати через відповідні API-запити.

Сервер перевіряє лише:

- чи є запит автентифікованим (наявність і валідність JWT);
- чи має користувач право доступу до конкретного ресурсу (наприклад, належність до кімнати).

Доступ до ендпоінтів обмежується за допомогою атрибута [Authorize]. Додатково, у специфічних випадках, може виконуватись перевірка приналежності користувача до певного контексту (наприклад, до кімнати), проте без використання централізованих політик на основі ролей.

2.4 Безпека

Застосунок реалізує кілька рівнів захисту від типових атак на веб-API.

- XSS-захист: Клієнтська частина реалізована на Angular, який за замовчуванням екранує шаблонні вирази, тим самим мінімізуючи ризик впровадження шкідливого JavaScript-коду.
- Захист токенів: Refresh-токен зберігається у cookie, а access-токен — у пам’яті клієнта. Це дозволяє зменшити ризик крадіжки токенів через XSS.

3 МОДЕЛЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ДАНИХ

3.1 Діаграма варіантів використання (Use Case)

Діаграма варіантів використання відображає основні сценарії взаємодії користувачів із системою.

Ролі користувачів:

- Анонімний користувач — має доступ до реєстрації та входу.
- Зареєстрований користувач — може створити кімнату або приєднатися до неї за кодом.
- Хост кімнати — створює кімнату, керує її учасниками та завданнями.
- Учасник (мембер) — виконує призначені завдання, переглядає список справ.
- Адміністратор — має розширені права доступу, контролює роботу системи, переглядає всі кімнати й користувачів, але не втручається у процес розподілу завдань.

Основні варіанти використання:

- Реєстрація та вхід у систему.
- Створення кімнати.
- Приєднання до кімнати за кодом.
- Перегляд та керування учасниками кімнати.
- Створення, редагування та видалення завдань.
- Виконання та позначення завдань як завершених.
- Вихід з кімнати або її повне видалення.
- Перегляд статистики.

Усі ці взаємозв’язки та дії наочно представлені на рисунку 3.1 — діаграмі варіантів використання веб-застосунку для обліку і розподілу домашніх справ.

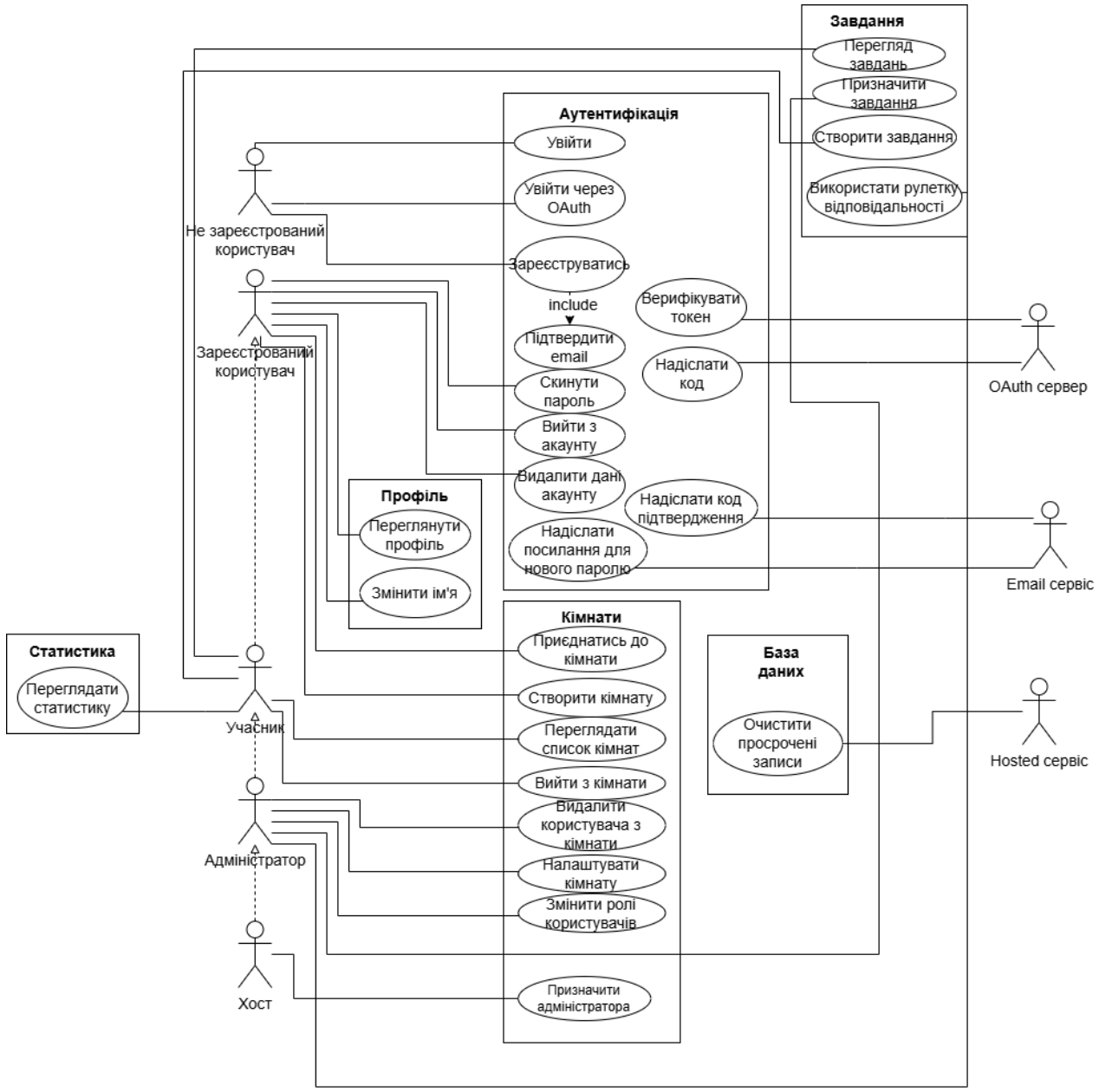


Рис. 3.1 Діаграма варіантів використання веб-застосунку обліку і розподілу домашніх справ

3.2 Діаграма класів

На діаграмі класів представлено фрагмент загальної архітектури програмної системи, який охоплює модуль роботи із завданнями (Assignments). Цей фрагмент ілюструє структуру основних сутностей, їх атрибути, методи та взаємозв’язки між компонентами, що беруть участь в обробці, зберіганні та управлінні завданнями.

Дана діаграма є частиною загального модельного представлення системи та слугує базою для реалізації відповідної логіки на рівні домену, сховища даних і API.

Основні класи:

- Assignment — доменна сутність, що містить атрибути Id, Title, Description, DueDate та Status. Статус представлений у вигляді перерахування TaskStatuses (Pending, InProgress, Completed).
- AssignmentRepository — реалізує інтерфейс IAssignmentRepository і відповідає за виконання запитів до бази даних щодо отримання та оновлення завдань.
- DatabaseContext — контекст бази даних, який містить набір сутностей Assignments і конфігурується через клас AssignmentConfiguration.
- UnitOfWork — координатор доступу до репозиторіїв, що забезпечує транзакційну цілісність при виконанні операцій.
- TaskController — контролер рівня API, який обробляє HTTP-запити для створення, оновлення, отримання та видалення завдань.
- CreateTaskDto — структура передачі даних, яка використовується для створення нового завдання.

Основні зв’язки:

- Зв’язок інжекції залежностей між TaskController та UnitOfWork.
- Агрегація між UnitOfWork та AssignmentRepository.
- Залежність між AssignmentRepository та DatabaseContext, що надає доступ до набору даних Assignments.
- Використання між DatabaseContext та AssignmentConfiguration, який налаштовує правила зберігання сутності Assignment.
- Реалізація інтерфейсів: AssignmentRepository реалізує IAssignmentRepository, а AssignmentConfiguration — IEntityTypeConfiguration<Assignment>.

Структура основних класів системи та їхні взаємозв’язки детально показані на рисунку 3.2 — діаграмі класів веб-застосунку для обліку і розподілу домашніх справ.

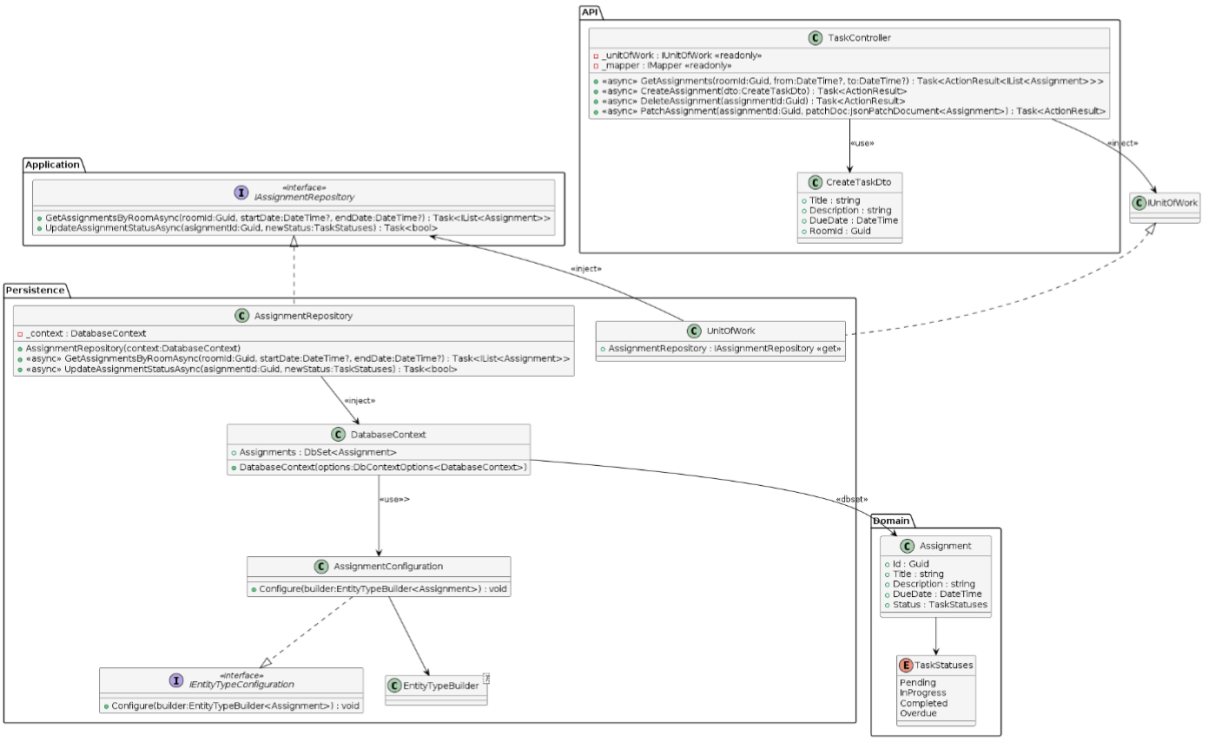


Рис. 3.2 Діаграма класів веб-застосунку для обліку і розподілу домашніх справ

Ця частина діаграми класів демонструє взаємодію компонентів, пов’язаних із обробкою завдань, і є основою для реалізації відповідної бізнес-логіки. Інші аспекти системи, такі як управління користувачами або кімнатами, не охоплені в межах цієї діаграми, але проєктуються за аналогічними принципами.

3.3 Структура бази даних

На діаграмі представлено структуру бази даних системи управління завданнями у багатокористувацькому середовищі. Архітектура підтримує розділення користувачів на кімнати (Rooms), створення завдань (Assignments), автентифікацію, а також додаткові механізми безпеки, такі як токени та

двофакторна автентифікація. Основна мета — забезпечити гнучке управління користувачами, кімнатами та завданнями в контексті спільної роботи.

Основні класи:

- Users — основна таблиця користувачів. Містить базову інформацію: унікальний ідентифікатор, ім’я користувача, підтвердження пошти та посилання на фото. Один користувач може бути пов’язаний із багатьма кімнатами через проміжну таблицю RoomUsers.
- Rooms — представлення логічних груп (кімнат), в яких працюють користувачі. Кожна кімната має назву та унікальний код.
- RoomUsers — проміжна таблиця для зв’язку Users ↔ Rooms (багато-до-багатьох). Додатково містить атрибут Role, що вказує роль користувача в конкретній кімнаті.
- Assignments — завдання, які належать до певної кімнати. Мають обов’язкові поля Title, Status, DueTime, CreatedAt і RoomId, а також опціональні Description і CompletedAt. Завдання може мати автора (UserId) та виконавця (AssignedToId), які посилаються на таблицю Users. Поле Status зберігається як текст, але функціонально поводиться як enum (Pending, InProgress, Completed).
- ResetPasswords, RefreshTokens, TwoFactorCodes — допоміжні таблиці для реалізації безпеки (відновлення паролю, рефреш токени, 2FA-коди). Всі пов’язані з конкретним користувачем через UserId.
- UsersCredentials — зберігає облікові дані користувача (email і хеш пароля), що відокремлені від основної інформації для кращої безпеки та масштабованості.

Основні зв’язки:

- Зв’язок багато-до-багатьох між Users та Rooms реалізовано через таблицю RoomUsers.
- Один Assignment завжди належить одній Room.
- Кожне Assignment може бути створено одним автором (UserId) і призначене одному виконавцю (AssignedToId) — обидва є зовнішніми ключами до Users.

– Таблиці ResetPasswords, RefreshTokens, TwoFactorCodes і UsersCredentials мають зв’язок один-до-одного з таблицею Users.

Організація даних та зв’язки між основними сутностями системи відображені на рисунку 3.3 — схемі бази даних веб-застосунку для обліку і розподілу домашніх справ.

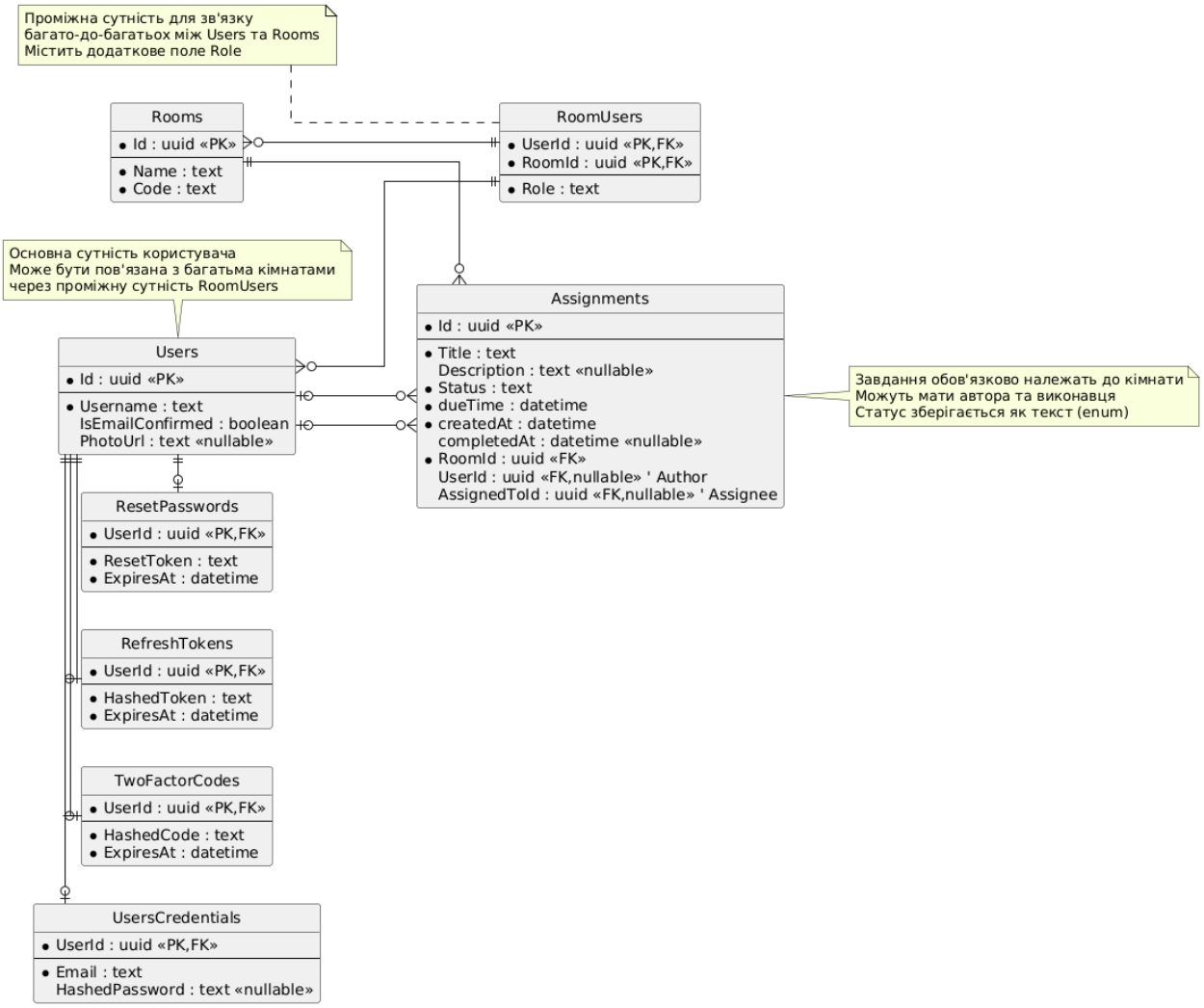


Рис. 3.3 Схема бази даних веб-застосунку для обліку і розподілу домашніх справ

Ця схема є основою для реалізації доменної логіки, що включає управління доступом, створення та обробку завдань, а також безпечну автентифікацію користувачів.

3.4 Діаграма активностей

Діаграма активностей ілюструє бізнес-логіку процесу розподілу домашніх завдань між користувачами в рамках кімнати. Сценарії побудовані з урахуванням ролей користувачів (хост, адміністратор, учасник) та типу обраного способу закріплення завдання. Передбачено три основні підходи до розподілу:

- Призначення виконавця при створенні завдання. Цей варіант доступний лише хостам та адміністраторам. Завдання створюється з одразу визначеним виконавцем, якого обирає користувач із відповідними правами. Мембери не мають доступу до цього способу.
- Самостійне відкриття “невидимого” завдання користувачем. Після створення завдання без виконавця, воно стає тимчасово “заблюреним” — зміст приховано до моменту натискання кнопки “показати”. Після відкриття конкретне завдання автоматично закріплюється за тим користувачем, який його відкрив. Цей підхід додає елемент гейміфікації до процесу виконання справ.
- Призначення випадкового завдання випадковому учаснику. Адміністратор або хост має змогу ініціювати процес через натискання кнопки “рулетка”. У результаті, одне з доступних “невидимих” завдань випадковим чином призначається одному з учасників кімнати. Якщо таких завдань немає, система повідомляє про помилку. Мембери не мають доступу до даного функціоналу. Після того як завдання стає закріпленим, незалежно від способу, користувач може натиснути кнопку “виконати”, що завершує процес.

Помилкові сценарії, які враховано на діаграмі:

- Мембер намагається призначити виконавця при створенні завдання.
- Мембер намагається натиснути кнопку “рулетка”.
- У системі відсутні завдання, які можна випадково розподілити.

Усі переходи реалізуються натисканням відповідних кнопок у застосунку. Вибір алгоритму закріплення визначає наступні дії системи.

Послідовність дій під час розподілу домашнього завдання у системі представлена на рисунку 3.4 — діаграмі активностей веб-застосунку для обліку і розподілу домашніх справ.

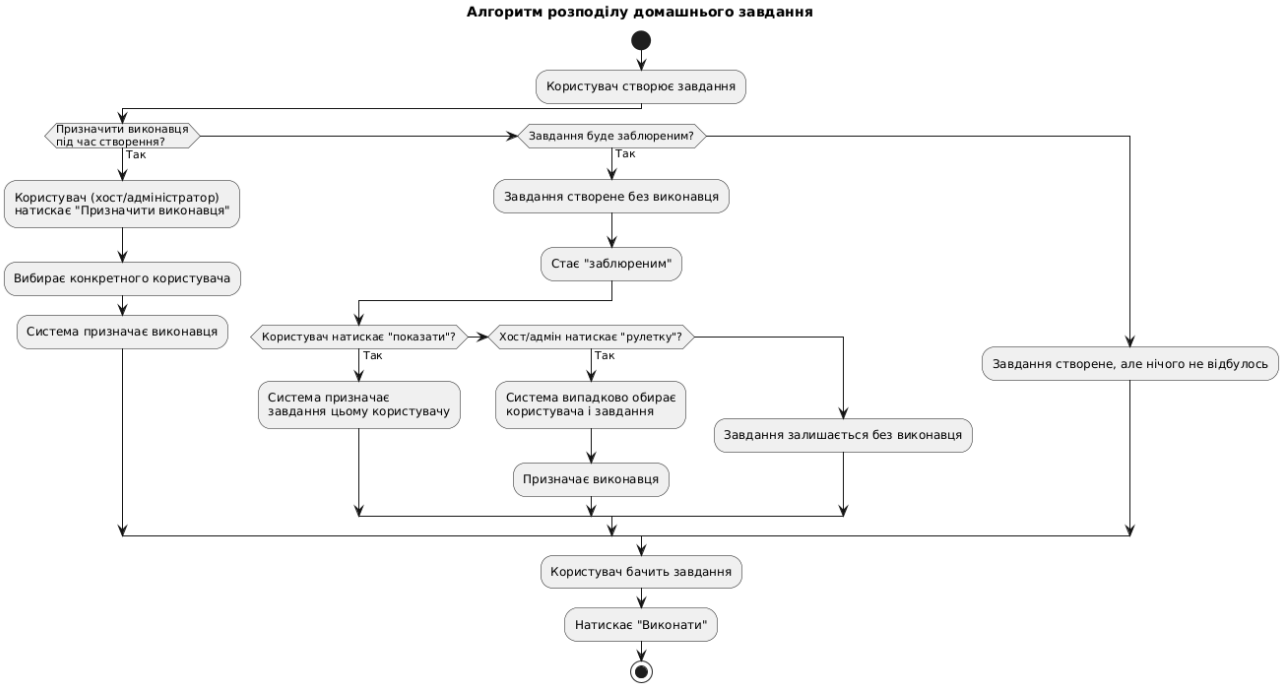


Рис. 3.4. Діаграма активностей розподілу домашнього завдання у веб-застосунку для обліку і розподілу домашніх справ

3.5 Демонстрація екранних форм користувача

У даному підрозділі представлено ключові інтерфейсні форми розробленого веб-застосунку, які ілюструють реалізацію основних функціональних вимог, визначених у розділі 1.3. Застосування графічних інтерфейсів користувача є важливим компонентом системи, оскільки забезпечує зручність взаємодії користувача з інформаційною системою та підвищує ефективність виконання основних операцій.

3.5.1 Форма реєстрації нового користувача

Форма реєстрації реалізована для забезпечення можливості створення нового облікового запису користувача. Вона містить поля для введення імені, адреси електронної пошти, пароля та підтвердження пароля. Передбачено валідацію введених даних у реальному часі, включно з перевіркою формату електронної пошти та відповідності пароля і його підтвердження. Для підвищення зручності реалізована також можливість швидкої реєстрації через акаунт Google, що дає змогу автоматично імпортувати базові облікові дані користувача. Після успішної реєстрації користувач перенаправляється до інтерфейсу входу в систему або автоматично авторизується, залежно від способу реєстрації.

Приклад форми реєстрації зображено на рисунку 3.5.

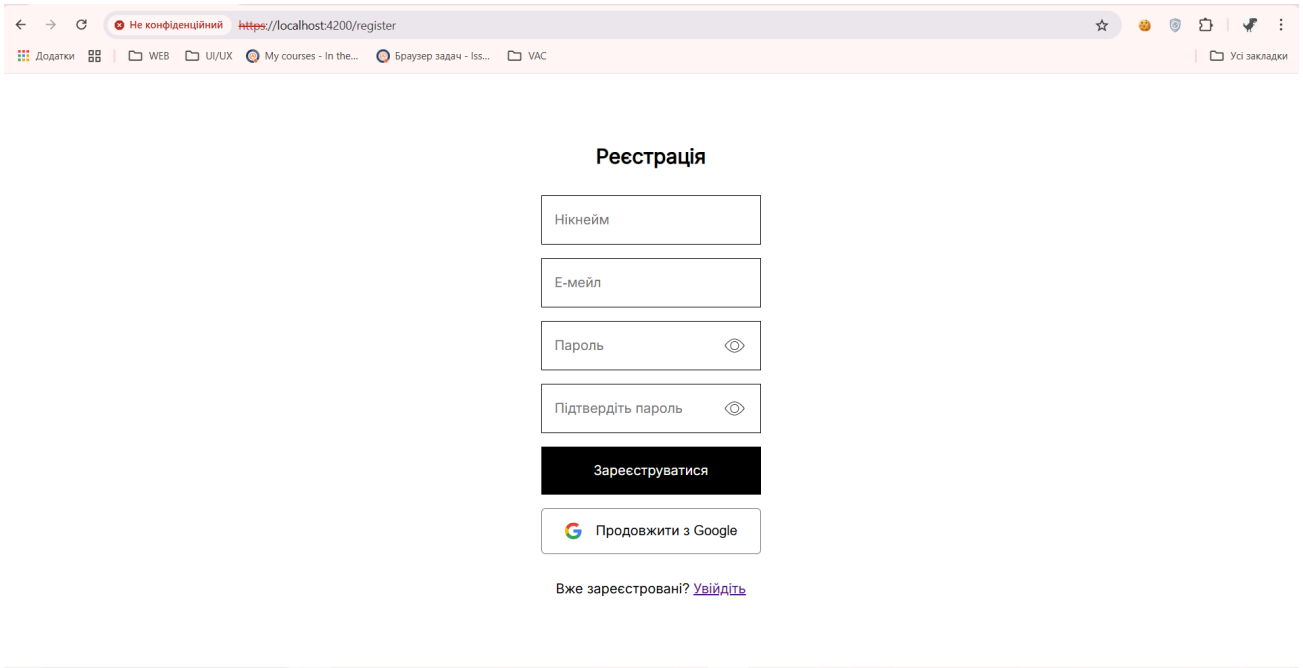


Рис. 3.5 Форма реєстрації нового користувача

3.5.2 Форма авторизації користувача (входу до системи)

Форма авторизації призначена для автентифікації зареєстрованих користувачів за допомогою логіна (електронної пошти) та пароля. У разі некоректного введення облікових даних виводиться інформативне повідомлення про помилку. Крім стандартного способу входу, користувачі також мають змогу

увійти до системи за допомогою Google-акаунта. Такий підхід сприяє зменшенню часу доступу до функціоналу системи та підвищенню безпеки автентифікації. Після успішної авторизації користувач потрапляє на головну сторінку особистого кабінету.

Форма для авторизації зображена на рисунку 3.6.

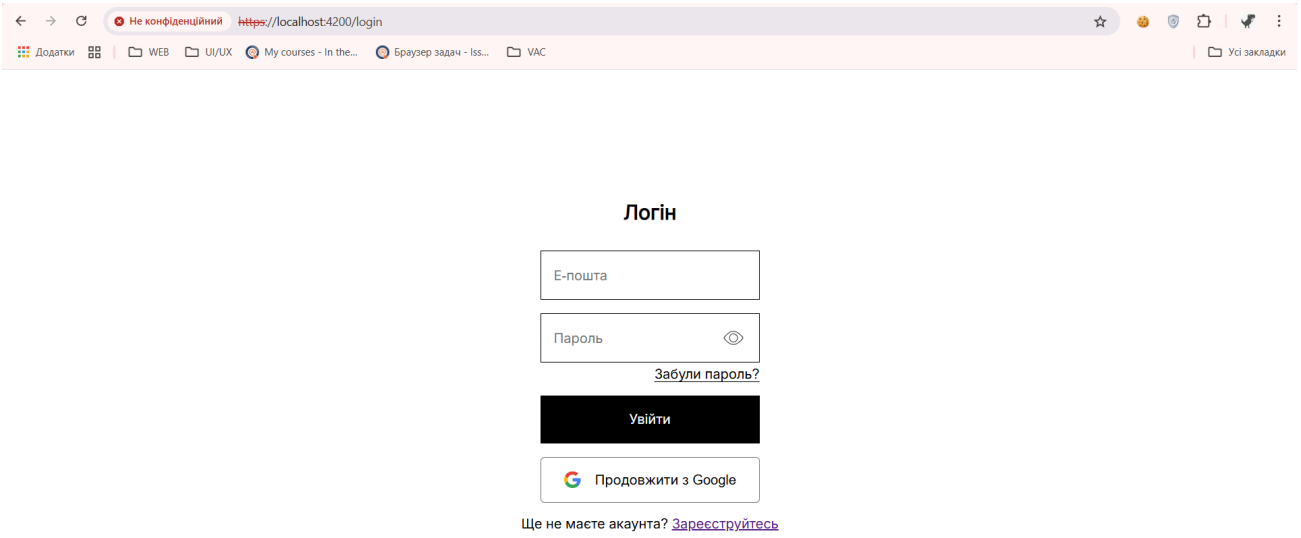


Рис. 3.6 Форма авторизації користувача

3.5.3 Форма створення нової побутової справи

Ця форма дає змогу створити нову справу в межах віртуальної кімнати. Передбачено поле для назви справи, її опису, виконавця а також дедлайну виконання.

На рисунку 3.7 зображено приклад форми для створення нової побутової справи.

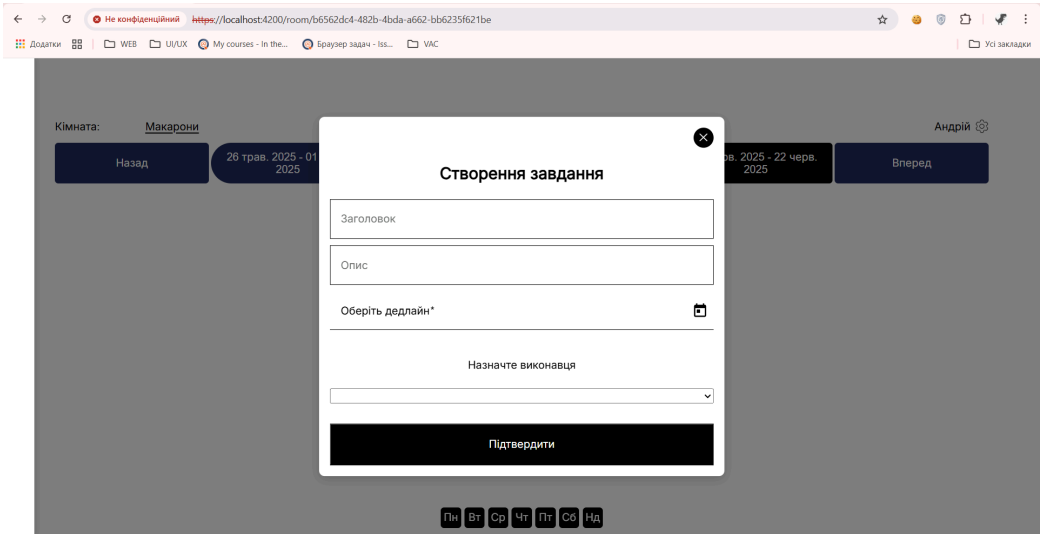


Рис. 3.7 Форма створення нової побутової справи

3.5.4 Головна форма користувача (інтерфейс головного екрану)

Головна форма є центральною частиною інтерфейсу застосунку, що відображає поточні справи користувача, стан виконання завдань, а також дає змогу взаємодіяти з іншими модулями. Вона містить доступ до створення нових справ, а також швидкий доступ до функцій кімнати.

Головна форма застосунку зображено на рисунку 3.8.

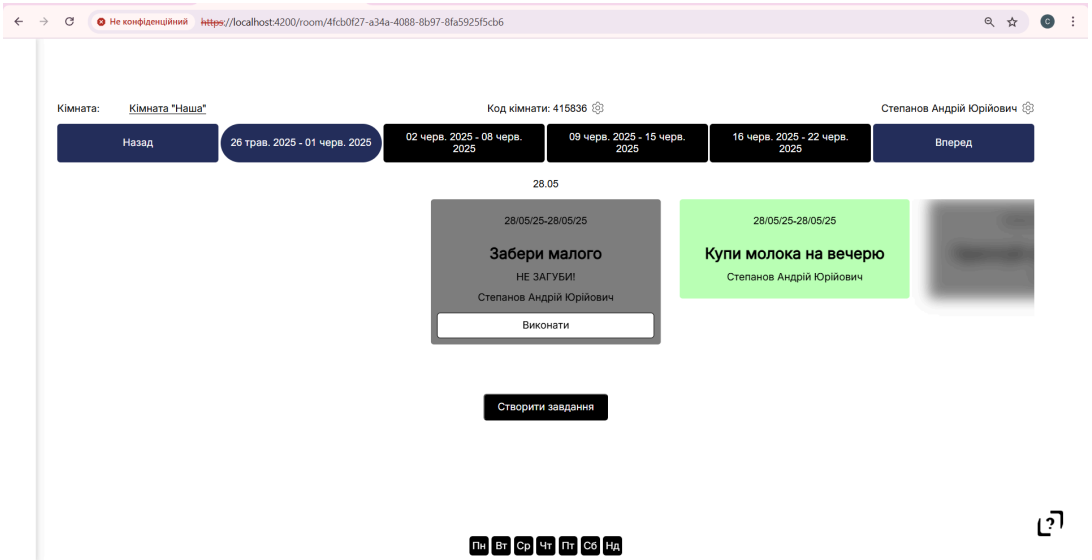


Рис. 3.8 Головна форма користувача

3.5.5 Форма керування складом учасників та параметрами віртуальної кімнати

Ця форма реалізує функціонал адміністрування кімнати. Користувач із відповідними правами може переглядати список учасників, змінювати їхні ролі (учасник, адміністратор, хост), видаляти користувачів або змінювати назву кімнати та її опис. Такий підхід забезпечує гнучке управління ієрархією доступу до справ.

Приклад форми налаштувань кімнати зображено на рисунку 3.9.

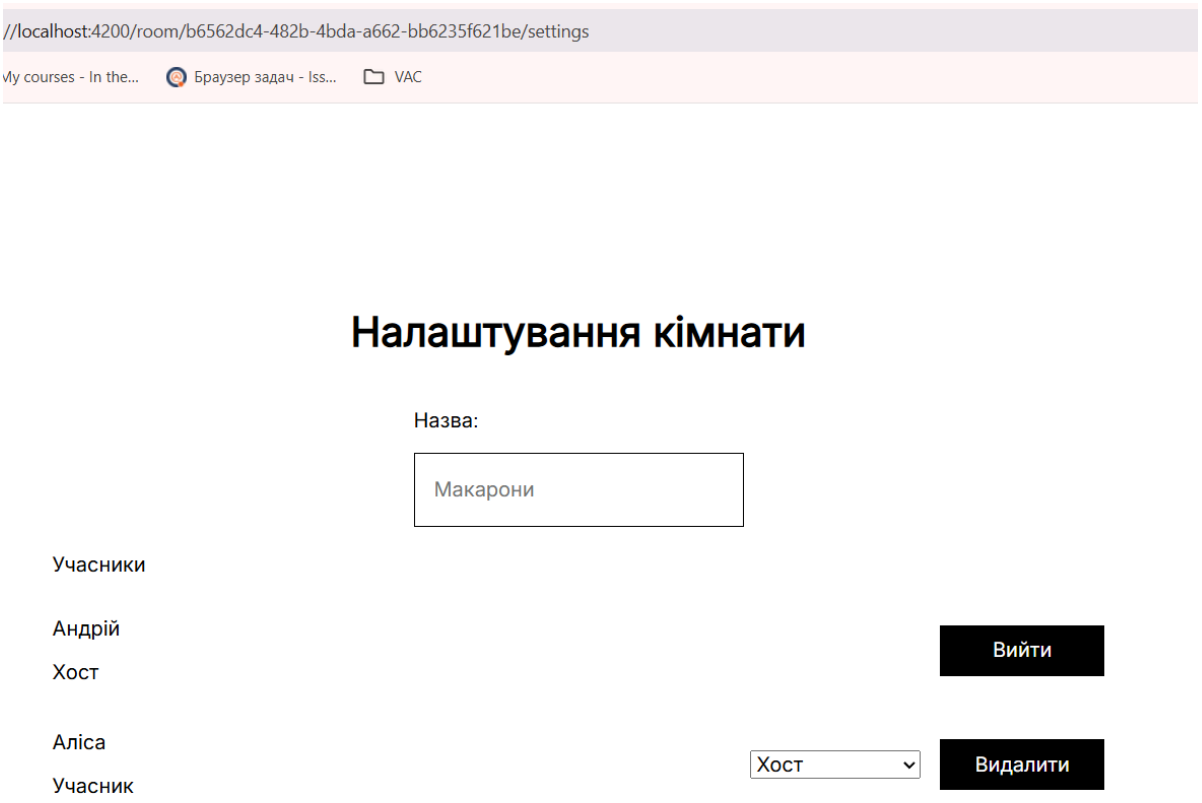


Рис. 3.9 Форма керування складом учасників та параметрами віртуальної кімнати

3.5.6 Форми відображення інформаційних повідомлень та обробки помилок

У застосунку реалізовані кастомізовані форми повідомлень, які інформують користувача про результати виконання дій (успіх, попередження, помилка). Повідомлення стисло пояснюють суть події та, за потреби, містять кнопки для повторної спроби або скасування дії. Це сприяє покращенню юзабіліті та знижує ризик втрати даних або помилок при взаємодії з системою. На рисунках 3.10 та 3.11 зображені приклади повідомлень (інформаційне та помилка)

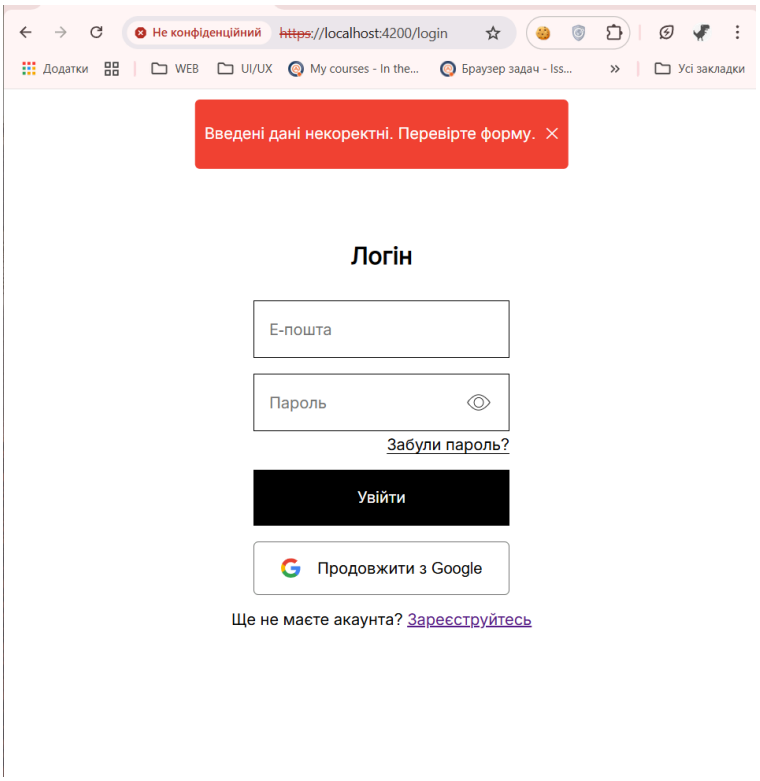


Рис. 3.10 Приклад повідомлення про помилку

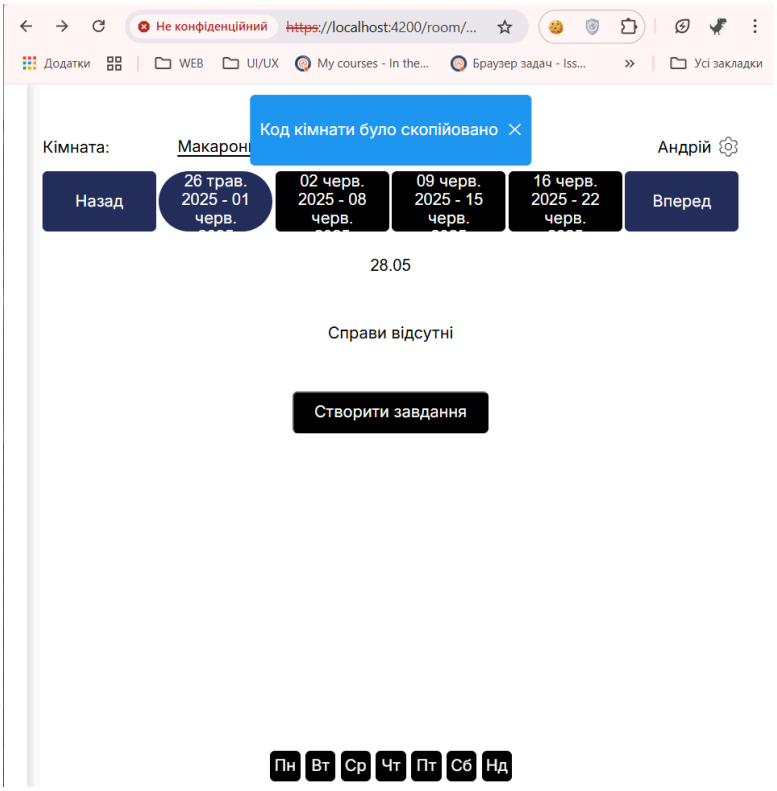


Рис. 3.10 Приклад інформаційного повідомлення

Таким чином, описані форми інтерфейсу забезпечують користувачам зручну, зрозумілу та ефективну взаємодію із системою. Вони враховують усі ключові функціональні можливості, необхідні для виконання основних сценаріїв використання. Завдяки цьому користувачі можуть легко орієнтуватися в застосунку, швидко знаходити потрібні розділи та виконувати заплановані дії. Інтерфейс дозволяє гнучко працювати як з простими, так і зі складними задачами, що значно розширює можливості системи. Також важливо, що інтерфейс адаптовано під потреби різних ролей, зокрема хостів, учасників і адміністраторів. Це гарантує персоналізований досвід для кожної категорії користувачів. У результаті система стає максимально зрозумілою, доступною й привабливою для всіх, хто з нею працює.

4 ТЕСТУВАННЯ ТА ОПТИМІЗАЦІЯ ПРОДУКТИВНОСТІ ЗАСТОСУНКУ

4.1 Загальний підхід

У процесі розробки web-застосунок проходив ручне функціональне тестування, спрямоване на перевірку стабільності ключових користувацьких сценаріїв, таких як створення та розподіл домашніх справ між учасниками. З огляду на обмеженість часу розробки, автоматизовані модульні тести (unit tests) не впроваджувалися, оскільки першочерговою метою була повноцінна реалізація функціоналу.

Під час тестування оцінювалася коректність обробки дій користувача, відповідність результатів очікуваній поведінці згідно з вимогами, а також стійкість до помилкових сценаріїв, зокрема введення некоректних або неповних даних. Тестування здійснювалося в сучасних браузерах (Google Chrome, Firefox), а також у середовищі з обмеженими ресурсами (старий ноутбук із 4 ГБ оперативної пам’яті), що дозволило перевірити роботу застосунку в умовах, наближених до реальних.

4.2 Функціональні перевірки

Тестування охоплювало основні дії користувача:

- додавання нової справи;
- призначення відповідального виконавця;
- зміна статусу виконання;
- реєстрація користувачів;
- зміна ролі користувача.

Таблиця 5.1

Основні функції та результати перевірки

Функція	Очікувана поведінка	Результат
Реєстрація користувача	Створення нового облікового запису, автоматичний вхід	Користувач створений, видано токен доступу, перехід на головну сторінку
Призначення виконавця	Відображення прізвища у справі	Ім’я виконавця відображається у деталях справи
Зміна статусу виконання	Візуальне оновлення, запис у базу даних	Статус змінено з «В процесі» на «Виконано», відображено у UI, збережено в базі.
Створення нової справи	Додавання до списку, збереження у базі	Справа додана до списку, запис у базі даних підтверджено

4.3 Перевірка стійкості до помилок

Застосунок показав стабільну поведінку при некоректних діях користувача. Помилки обробляються без зупинки роботи системи, забезпечуючи коректний зворотний зв’язок та збереження загальної стабільності.

Приклади перевірених ситуацій:

− Надсилання порожньої форми — користувачу виводиться повідомлення про помилку, запобігається відправлення неповного запиту. Приклад повідомлення з помилкою зображено на рисунку 4.1.

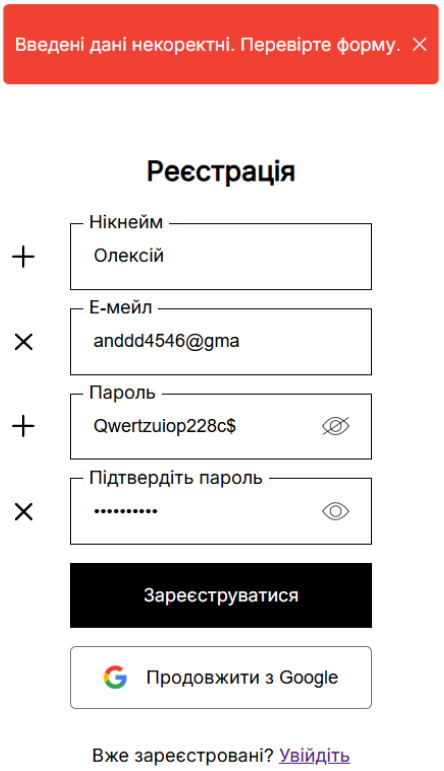


Рис. 4.1 Екранна форма з повідомленням про помилку при надсиланні порожньої форми

– Перехід на неіснуючу сторінку — виконується перенаправлення на головну сторінку, при цьому URL очищається від некоректного фрагмента; Приклад переходу на неіснуючу сторінку зображено на рисунку 4.2 (до переходу) та рисунку 4.3 (після переходу)

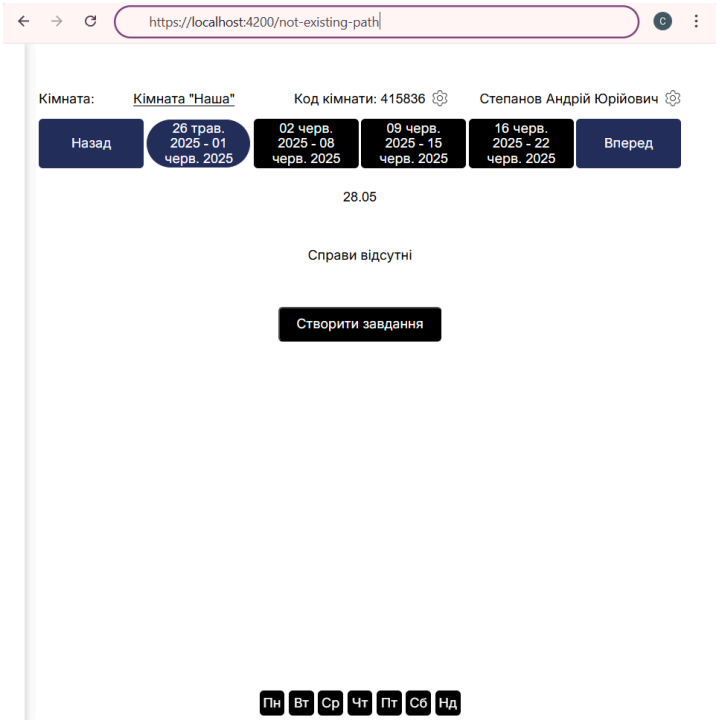


Рис. 4.2 Екранна форма під час введення некоректної адреси сторінки перед переходом

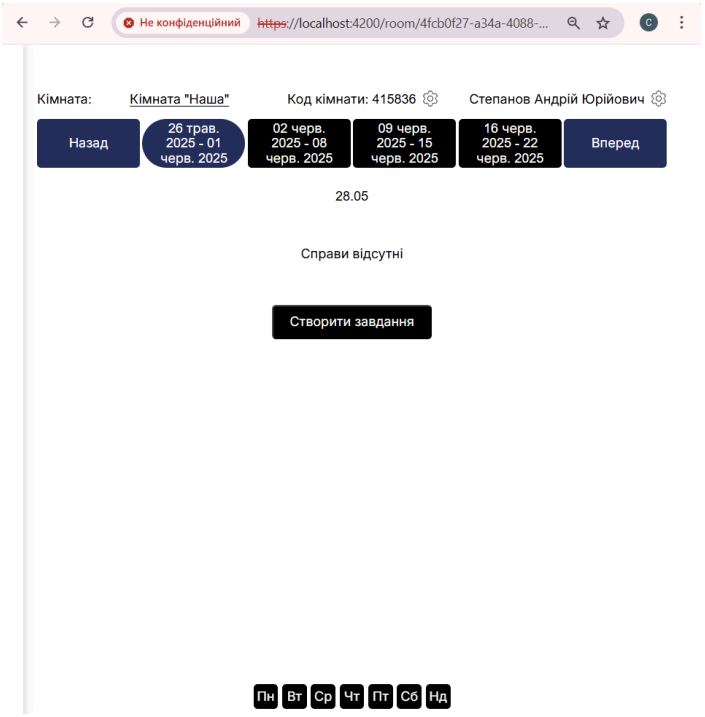


Рис. 4.3 Екранна форма після автоматичного перенаправлення на головну сторінку

– Відсутність токену — здійснюється спроба автоматичного оновлення через refresh-токен; якщо це не вдається, користувач автоматично перенаправляється на сторінку логіну. Оновлення токену нижче на рисунку 4.4.

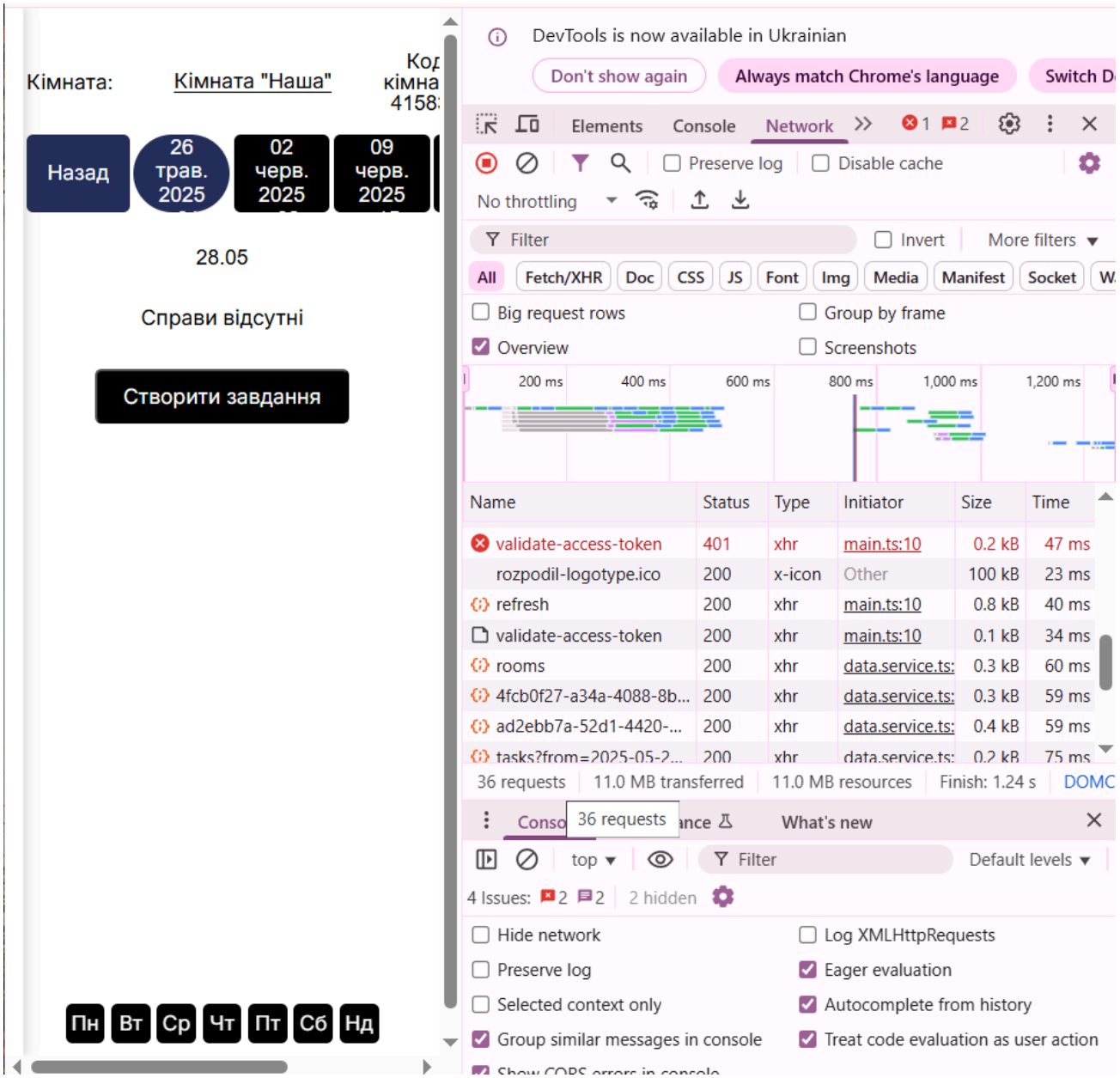


Рис. 4.4 Екранна форма обробки втрати авторизації через відсутність токена

4.4 Оцінка швидкодії

Проведено неформальний аналіз продуктивності за часом реакції на основні дії. Навантаження моделювалося шляхом додавання великої кількості справ (до 100).

Було отримано дані:

- Завантаження інтерфейсу: < 3 с при 500 записах
- Додавання справи: ≈ 0.5 с
- Зміна статусу: миттєво (менше 200 мс)

Загалом результати свідчать про те, що продуктивність перебуває в межах норми, без помітних затримок або блокувань UI.

4.5 Оптимізація

У процесі розробки застосовано низку базових оптимізаційних рішень, зокрема:

- Встановлено вбудований таймер, що обмежує частоту запитів і запобігає надмірному навантаженню (антиспам-механізм);
- Використано механізм Angular Hydration для скорочення часу першого відображення контенту;
- Реалізовано логіку з умовною підпискою RxJS [23], яка забезпечує повторне використання поточного запиту у разі повторного звернення до ресурсу.
- Реалізовано механізм лінивого завантаження (lazy-loading) маршрутів, що дозволяє розділити застосунок на окремі модулі, які завантажуються лише за потреби. Такий підхід мінімізує початковий обсяг переданих даних, зменшує час завантаження початкової сторінки (initial load time) та оптимізує використання мережевих і обчислювальних ресурсів, підвищуючи загальну продуктивність і масштабованість веб-застосунку.

4.6 Подальші напрямки удосконалення

Для подальшого підвищення продуктивності, масштабованості та надійності розроблюваної системи доцільно реалізувати низку технічних рішень, кожне з яких спрямоване на вирішення конкретних проблем ефективності обробки даних, мережевої взаємодії та стабільності функціонування.

Впровадження клієнтського кешу.

Застосування кешування на клієнтській стороні (наприклад, через localStorage, IndexedDB або вбудовані механізми кешування Angular HttpClient) дозволяє істотно зменшити кількість повторних HTTP-запитів до серверної частини. Це, у свою чергу, забезпечує зниження мережевого навантаження, прискорення часу відгуку інтерфейсу, а також забезпечує роботу з даними навіть у разі тимчасової втрати інтернет-з’єднання.

IndexedDB надає можливість зберігати структуровані дані великого обсягу без обмежень, притаманних localStorage, і дозволяє здійснювати складні запити до кешованих записів. Angular HttpClient з інтегрованими кеш-інтерцепторами дозволяє створювати централізовані стратегії кешування, наприклад: кешування GET-запитів на час сесії або до появи нових даних. Це забезпечує як скорочення часу завантаження, так і зменшення затримок (latency) при повторних операціях.

Розширення системи обробки помилок з логуванням подій.

З технічного погляду, обробка помилок повинна включати централізоване перехоплення винятків як на клієнтській стороні (Angular Global Error Handler), так і на серверній стороні (через middleware, фільтри винятків або сервіси логування). Логування має охоплювати критичні події, винятки, помилки виконання, проблеми з мережею та інші інциденти, що впливають на стабільність роботи.

Зібрані логи мають зберігатися у централізованому сховищі (наприклад, Elasticsearch, сервіси типу Sentry, Logstash), що дозволяє здійснювати ретроспективний аналіз, будувати аналітичні звіти та знаходити закономірності виникнення збоїв. Такий підхід знижує час пошуку й усунення несправностей,

підвищує стійкість до збоїв та гарантує більш надійне функціонування при навантаженнях.

Впровадження серверної пагінації та сортування.

При роботі з великими обсягами даних передавання повного списку записів на клієнт є неефективним як з погляду продуктивності, так і з точки зору мережевих ресурсів. Серверна пагінація дозволяє обмежувати обсяг даних, що передаються в одному запиті, лише до потрібної порції (наприклад, 20–50 записів), а сортування — забезпечувати впорядкування даних ще на етапі формування відповіді.

На рівні API це реалізується через передачу параметрів page, pageSize, sortField і sortOrder у запитах, що дає змогу серверу здійснювати оптимізовані SQL-запити (наприклад, із використанням LIMIT/OFFSET або ROW_NUMBER() у разі складних сценаріїв).

У результаті мінімізується використання пам’яті на клієнті, пришвидшується завантаження даних, а також знижується навантаження на сервер при масштабуванні системи.

Інтеграція SignalR або WebSocket для обміну даними в реальному часі.

Стандартні підходи періодичного опитування (polling) є неефективними при роботі з динамічними даними, оскільки вони створюють значне навантаження на сервер через регулярні запити, навіть за відсутності нової інформації.

Технології двостороннього обміну даними, такі як WebSocket або SignalR, дозволяють підтримувати постійне з’єднання між клієнтом і сервером, де оновлення передаються лише за фактом зміни стану. SignalR додатково забезпечує автоматичне переключення між WebSocket, Server-Sent Events або Long Polling залежно від можливостей клієнта та сервера, що робить інтеграцію більш гнучкою й сумісною.

Таке рішення особливо корисне для сценаріїв з високими вимогами до актуальності даних, наприклад: оновлення статусу завдань, сповіщення, синхронізація між користувачами. Воно значно знижує середній час реакції

системи, а також оптимізує використання серверних ресурсів при одночасному обслуговуванні великої кількості клієнтів.

Таким чином, запровадження описаних технічних механізмів дозволяє суттєво підвищити ефективність програмного продукту за рахунок оптимізації передачі та обробки даних, підвищення стійкості до збоїв, зменшення часу відгуку системи й забезпечення безперервної роботи у багатокористувацькому середовищі. Такі заходи є критичними при підготовці системи до промислової експлуатації, особливо з урахуванням перспектив масштабування та підвищення навантаження.

ВИСНОВКИ

1. Проведено аналіз предметної області обліку домашніх справ, що дозволило визначити ключові проблеми й потреби користувачів, для яких розробляється веб-застосунок.

2. Проаналізовано існуючі програмні рішення з управління домашніми завданнями, їхні функції та обмеження, що допомогло сформувати вимоги до системи з урахуванням специфіки розподілу домашніх справ між користувачами.

3. Визначено та задокументовано конкретні функціональні й нефункціональні вимоги, які забезпечують надійне збереження, редагування і призначення домашніх справ у рамках веб-застосунку.

4. Спроєктовано архітектуру системи з урахуванням особливостей предметної області, розроблено структуру бази даних і серверної логіки, які підтримують управління та контроль виконання домашніх завдань.

5. Реалізовано ключові модулі веб-застосунку: автентифікацію користувачів, механізми створення, редагування, розподілу та відстеження статусу домашніх справ, а також інтуїтивний інтерфейс користувача на Angular.

6. Проведено ручне функціональне тестування, що підтвердило коректність і стабільність роботи системи у ключових сценаріях обліку та розподілу домашніх справ.

7. Робота пройшла апробацію на наукових конференціях, за результатами апробації опубліковано тези доповідей:

Степанов А.Ю., Золотухіна О.А. Використання реактивного програмування в web-застосунку для обліку і розподілу домашніх справ // Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в ІКТ». Збірник тез. 24.04.2025, ДУІКТ, м. Київ. (подано до друку).

Степанов А.Ю. Захист інформації у web-застосунку для обліку і розподілу домашніх справ // XII Всеукраїнська науково-практична конференція молодих учених «Інформаційні технології - 2025». Збірник тез. 15.05.2025, КСУ імені Бориса Грінченка, Київ, Україна, С. 330-331.

ПЕРЕЛІК ПОСИЛАНЬ

1. Malone T. W., Crowston K. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*. New York, 1994. Т. 26, вип. 1. С. 87–119. ISSN 0360‑0300. URL: https://dl.acm.org/doi/10.1145/174666.174668.

2. Хічій О., Сівчук П. І. Сімейні конфлікти. *V Міжнародна студентська науково-технічна конференція «Природничі та гуманітарні науки. Актуальні питання»*, Тернопіль, Україна, 28–29 квіт. 2022 / ред.: Б. Андрушків та ін. ; Міністерство освіти і науки України, Тернопільський національний технічний університет імені Івана Пулюя, Маріборський університет (Словенія), Технічний університет в Кошице (Словаччина), Каунаський технологічний університет (Литва), Львівський національний університет імені Івана Франка, Гірничо-металургійна академія ім. Станіслава Сташиця (Польща), Луцький національний технічний університет, Чернівецький національний університет імені Юрія Федьковича, Вроцлавський економічний університет (Польща), Університет технологій та економіки імені Хелени Ходковської (Польща), Донбаська державна машинобудівна академія. Тернопіль : Тернопільський національний технічний університет імені Івана Пулюя, 2022. С. 113–114.

3. Aas J. J., Grønbeck S. R. R. Gamification of Chores : дис. на здобуття ступеня магістра наук з інформатики / Norwegian University of Science and Technology. Trondheim, 2021. 189 с. URL: https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2787234/no.ntnu%3ainspera%3a80744115%3a20906316.pdf?sequence=1&isAllowed=y.

4. Appsent. Sweepy. 2020. URL: https://play.google.com/store/apps/details?id=app.sweepy.sweepy.

5. Artem Chakhouski. Tidy: House Cleaning Schedule. 2022. URL: https://play.google.com/store/apps/details?id=com.cklab.cleaning.

6. Parkinson C. N. Parkinsons Law: Or The Pursuit Of Progress : paperback – international edition. London : Penguin UK, 1986. 128 с. ISBN 978‑0140091076.

7. Nipto. Nipto: Split Household Chores. 2020. URL: https://play.google.com/store/apps/details?id=com.nipto.niptoapp.

8. LoopLoop Aps. Tody - Smarter Cleaning. 2019. URL: https://play.google.com/store/apps/details?id=com.looploop.tody.

9. Angular. Home • Angular. URL: https://angular.dev

10. Express - Node.js web application framework. Express - Node.js web application framework. URL: https://expressjs.com.

11. Chart.js | Chart.js. Chart.js | Open source HTML5 Charts for your website. URL: https://www.chartjs.org/docs/latest.

12. Swiper - The Most Modern Mobile Touch Slider. Swiper. URL: https://swiperjs.com/swiper-api.

13. luxon - Immutable date wrapper. Site not found · GitHub Pages. URL: https://moment.github.io/luxon/#/.

14. C# Guide - .NET managed language. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/dotnet/csharp.

15. .NET documentation. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/dotnet/?WT.mc_id=dotnet-35129-website.

16. ASP.NET documentation. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-9.0.

17. Overview of Entity Framework Core - EF Core. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/ef/core.

18. PostgreSQL: Documentation. PostgreSQL: The world's most advanced open source database. URL: https://www.postgresql.org/docs.

19. Microsoft.IdentityModel.Tokens Namespace - Microsoft Authentication Library for .NET. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/dotnet/api/microsoft.identitymodel.tokens?view=msal-web-dotnet-latest.

20. GitHub - lukencode/FluentEmail: All in one email sender for .NET. Supports popular senders (SendGrid, MailGun, etc) and Razor templates. GitHub. URL:

https://github.com/lukencode/FluentEmail.

21. REST API Documentation Tool | Swagger UI. API Documentation & Design Tools for Teams | Swagger. URL: https://swagger.io/tools/swagger-ui.

22. GitHub - sindresorhus/modern-normalize: Normalize browsers' default style. GitHub. URL: https://github.com/sindresorhus/modern-normalize?tab=readme-ov-file.

23. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall, 2017. 432 p.

24. Introduction | Learn RxJS. Introduction | Learn RxJS. URL: https://www.learnrxjs.io.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка web-застосунку для обліку і розподілу домашніх справ

Виконав студент 4 курсу
групи ПД-43
Степанов Андрій Юрійович
Керівник роботи
К.т.н, доц, доцент кафедри ІПЗ Золотухіна Оксана Анатоліївна

Київ – 2025

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи спрощення обліку і розподілу домашніх справ за рахунок використання web-застосунку

Об’єкт дослідження процес розподілу домашніх справ

Предмет дослідження програмне забезпечення для організації розподілу домашніх справ

ЗАДАЧІ ДИПЛОМНОЇ РОБОТИ

1. Провести аналіз предметної області обліку домашніх справ з метою виявлення ключових проблем і потреб користувачів у сфері створення, розподілу та контролю виконання завдань.
2. Дослідити існуючі програмні рішення для управління домашніми справами, визначити їхні функціональні можливості та обмеження для формування обґрунтованих вимог до розроблюваної системи.
3. Сформулювати функціональні та нефункціональні вимоги до веб-застосунку, що забезпечують надійність, безпеку, зручність та ефективність управління домашніми справами.
4. Розробити архітектуру системи, враховуючи особливості предметної області, та створити структуру бази даних, що підтримує збереження, обробку і розподіл домашніх завдань.
5. Реалізувати ключові функціональні модулі веб-застосунку, зокрема автентифікацію користувачів, механізми створення, призначення та моніторингу виконання домашніх справ.
6. Провести функціональне тестування розробленої системи для перевірки коректності, стабільності та відповідності реалізованого функціоналу поставленим вимогам.

3

АНАЛІЗ АНАЛОГІВ

	Sweepy	Tidy	Tody	Nipto	Rozpodil
Статистика	-	+	+	+	+
Таблиця лідерів	+	-	-	+	+
Рулетка відповідального	-	-	+	-	+
Декілька просторів	-	-	-	-	+
Призначення відповідального	-	-	+	+	+
Приховані завдання	-	-	-	-	+

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

1. Реєстрація та авторизація користувачів
2. Перегляд, створення домашніх справ
3. Призначення домашніх справ користувачам
4. Гейміфікація процесу розподілу завдань через реалізацію рулетки відповідального (випадковий вибір користувача на завдання) та секретних завдань (заблюрені завдання, що розкриваються лише після вибору).
5. Керування ролями користувачів

Нефункціональні вимоги:

1. Завантаження інтерфейсу не більше ніж за 5 секунд на швидкості завантаження до 20 Мбіт/с
2. Чуттєві дані шифруються за допомогою bcrypt або sha-2
3. Тексти, зображення та ресурси повинні бути стиснуті для зменшення обсягу
4. Інтерфейс доступний для людей з обмеженими можливостями
5. API побудовано згідно з принципами REST
6. Передбачено ролі користувачів: адміністратор, хост, мембер.

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Visual Studio Code

GitHub

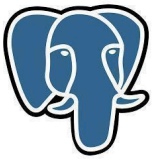
ASP.NET

Swiper.js

Visual Studio

Angular

RxJS

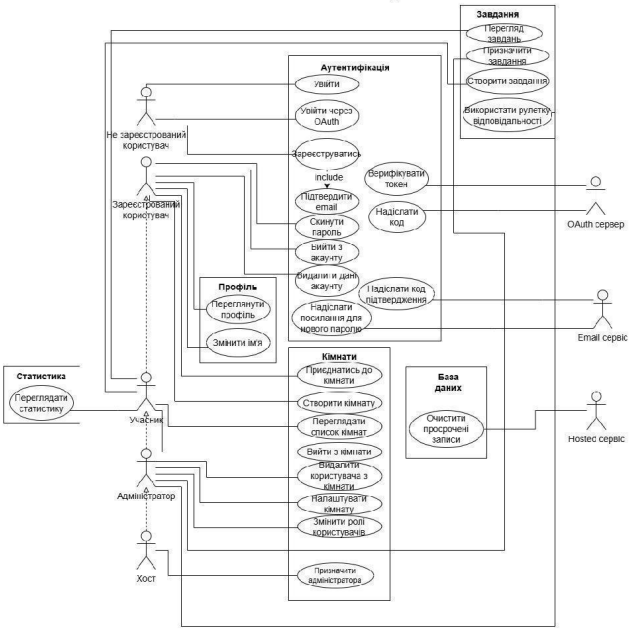


PostgreSQL

Chart.js

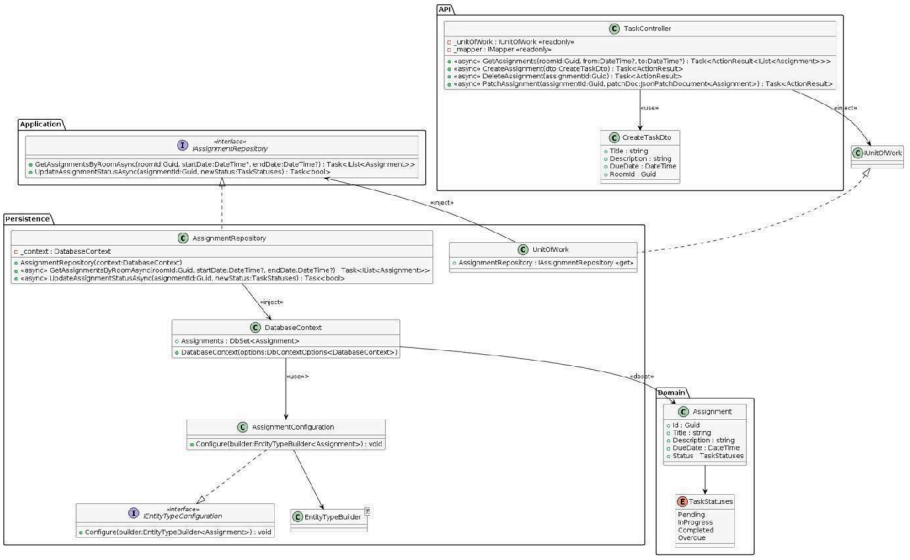
6

ДІАГРАМА ПРЕЦЕДЕНТІВ



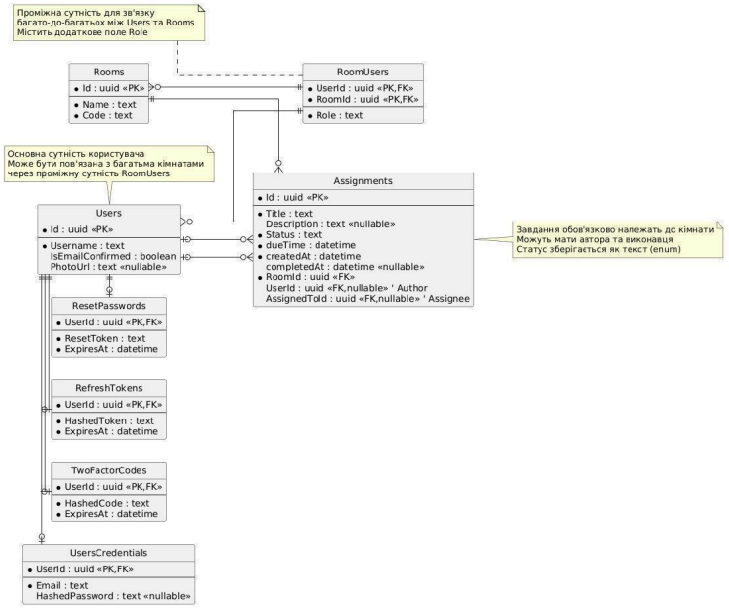
7

ДІАГРАМА КЛАСІВ

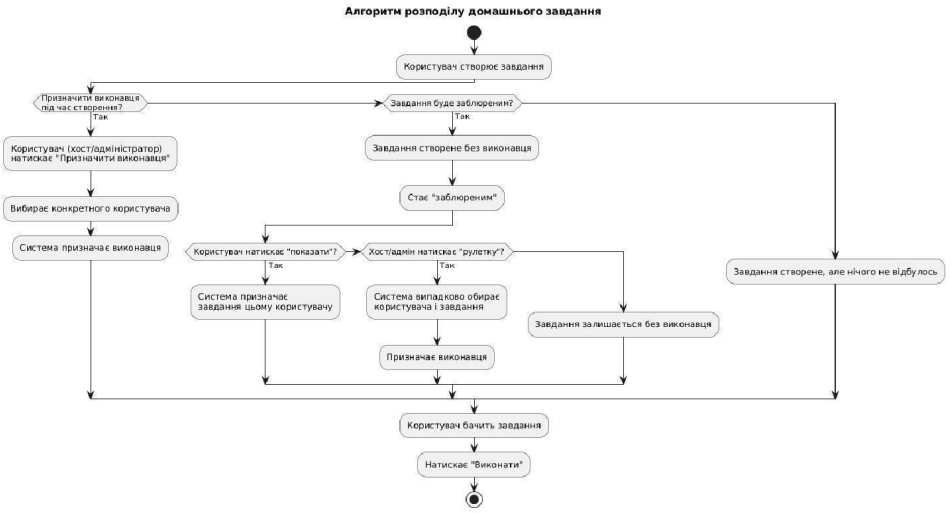


8

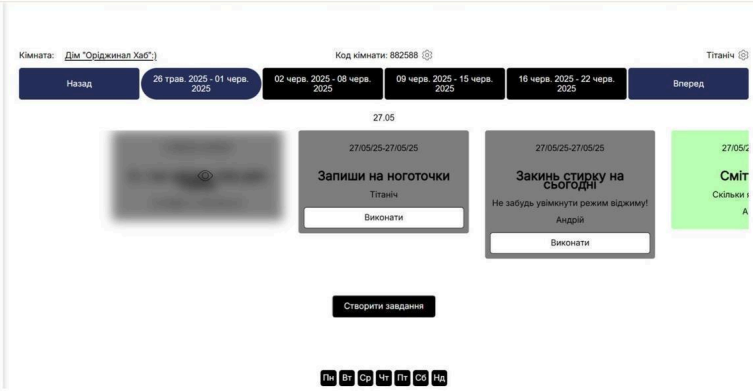
СХЕМА БАЗИ ДАНИХ



АЛГОРИТМ РОЗПОДІЛУ ЗАВДАНЬ



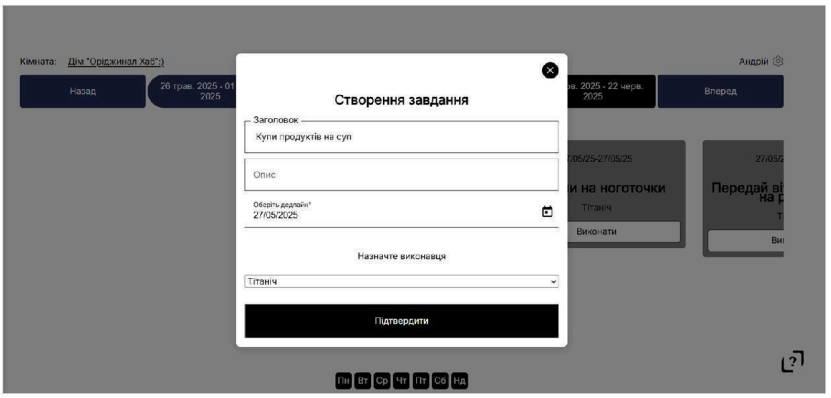
ЕКРАННІ ФОРМИ



Головний екран web-застосунку

11

ЕКРАННІ ФОРМИ



Екранна форма для створення завдання

12

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Степанов А.Ю., Золотухіна О.А. Використання реактивного програмування в web-застосунку для обліку і розподілу домашніх справ // Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в ІКТ». Збірник тез. 24.04.2025, ДУІКТ, м. Київ. К.: ДУІКТ, 2025. С. 75-77.
2. Степанов А.Ю. Захист інформації у web-застосунку для обліку і розподілу домашніх справ // XII Всеукраїнська науково-практична конференція молодих учених «Інформаційні технології - 2025». Збірник тез. 15.05.2025, КСУ імені Бориса Грінченка, Київ, Україна, С. 330 - 331.

ВИСНОВКИ

1. Проведено аналіз предметної області обліку домашніх справ, що дозволило визначити ключові проблеми й потреби користувачів, для яких розробляється веб-застосунок.
2. Проаналізовано існуючі програмні рішення з управління домашніми завданнями, їхні функції та обмеження, що допомогло сформувати вимоги до системи з урахуванням специфіки розподілу домашніх справ між користувачами.
3. Визначено та задокументовано конкретні функціональні й нефункціональні вимоги, які забезпечують надійне збереження, редагування і призначення домашніх справ у рамках веб-застосунку.
4. Спроєктовано архітектуру системи з урахуванням особливостей предметної області, розроблено структуру бази даних і серверної логіки, які підтримують управління та контроль виконання домашніх завдань.
5. Реалізовано ключові модулі веб-застосунку: автентифікацію користувачів, механізми створення, редагування, розподілу та відстеження статусу домашніх справ, а також інтуїтивний інтерфейс користувача на Angular.
6. Проведено ручне функціональне тестування, що підтвердило коректність і стабільність роботи системи у ключових сценаріях обліку та розподілу домашніх справ.

ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

```
using AutoMapper;
using Microsoft.AspNetCore.JsonPatch;
using Microsoft.AspNetCore.Mvc;
using Rozpodil.API.Dtos.Requests.Task;
using Rozpodil.API.Dtos.Responses.TaskStatistics;
using Rozpodil.Application.Interfaces.Repositories;
using Rozpodil.Domain.Entities;

namespace Rozpodil.API.Controllers
{
   [Route("api/rooms/{roomId}/tasks")]
   [ApiController]
   public class TaskController : ControllerBase
   {
      private readonly IUnitOfWork _unitOfWork;
      private readonly IMapper _mapper;
      public TaskController(
            IUnitOfWork unitOfWork,
            IMapper mapper
         )
      {
         _unitOfWork = unitOfWork;
         _mapper = mapper;
      }

      [HttpGet]
        public async Task<ActionResult<IList<Assignment>>>
GetAssignments(
            [FromRoute] Guid roomId,
            [FromQuery] DateTime? from,
            [FromQuery] DateTime? to
         ) {

         var tasks = await _unitOfWork.AssignmentRepository
            .GetAssignmentsByRoomAsync(roomId, from, to);

         return Ok(tasks);
      }

         // TODO: налаштуватир created відповідно до його
обов'язкових параметрів
      [HttpPost]
                        public async Task<ActionResult>
CreateAssignment(CreateTaskDto dto)
      {
                                    var user = await
_unitOfWork.UserRepository.GetUserByIdAsync(dto.UserId);
                                   var room = await
_unitOfWork.RoomRepository.GetRoomById(dto.RoomId);

         if (user == null && room == null)
         {
            return BadRequest();
         }

         var assignment = new Assignment
         {
            Title = dto.Title,
            Description = dto.Description,
            dueTime = dto.Deadline,
            createdAt = dto.CreatedAt,
            User = user,
            Room = room,
            Status = Domain.Enums.TaskStatuses.Pending,
            AssignedToId = dto.AssignedToId
         };
                                                                  await
_unitOfWork.AssignmentRepository.CreateAssignmentAsync(
assignment);
         if (assignment.AssignedToId != null)
                                                assignment.Status =
Domain.Enums.TaskStatuses.InProgress;

         await _unitOfWork.SaveChangesAsync();

         return Created();
      }

      [HttpDelete("{assignmentId}")]
      public async Task<ActionResult> DeleteAssignment(Guid
assignmentId)
      {
                                    var assignment = await
_unitOfWork.AssignmentRepository.GetAssignmentByIdAsyn
c(assignmentId);

         if (assignment == null)
            return NoContent();

                                                                  await
_unitOfWork.AssignmentRepository.DeleteAssignment(assign
ment);
         await _unitOfWork.SaveChangesAsync();

         return NoContent();
      }

      [HttpPatch("{assignmentId}")]
      public async Task<ActionResult> PatchAssignment(
            Guid assignmentId,
                  [FromBody] JsonPatchDocument<Assignment>
patchDoc
         )
      {
         if (patchDoc == null)
            return BadRequest();

                                    var assignment = await
_unitOfWork.AssignmentRepository.GetAssignmentByIdAsyn
c(assignmentId);

         if (assignment == null)
            return NotFound();

         patchDoc.ApplyTo(assignment);

         await _unitOfWork.SaveChangesAsync();

         return NoContent();
      }

      [HttpGet("statistics/completed")]
                        public async Task<List<UserTaskDto>>
GetCompleteStatistics(
            Guid roomId,
            [FromQuery] DateTime? from,
            [FromQuery] DateTime? to,
            [FromQuery] List<Guid>? excludedUserIds
         )
      {
                              var statisticsValueObject = await
_unitOfWork.AssignmentRepository
               .GetTaskStatisticsCompleteAsync(roomId, from, to,
excludedUserIds);
```

```
                                                    return
_mapper.Map<List<UserTaskDto>>(statisticsValueObject);
        }
    }
}

using AutoMapper;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Rozpodil.API.Dtos.Requests.Auth;
using Rozpodil.API.Dtos.Responses;
using Rozpodil.Application.Commands;
using Rozpodil.Application.Common;
using Rozpodil.Application.Common.Interfaces;
using Rozpodil.Application.Interfaces.Auth;
using Rozpodil.Application.Interfaces.Repositories;
using Rozpodil.Application.Interfaces.Security;
using System.Threading.Tasks;

namespace Rozpodil.API.Controllers
{
    [AllowAnonymous]
    [Route("api/[controller]")]
    [ApiController]
    public class AuthController : ControllerBase
    {
        private readonly IMapper _mapper;
        private readonly IAuthService _authService;
        private readonly IOAuthService _oauthService;

        // TODO: стерти
        private readonly IUnitOfWork _unitOfWork;
        private readonly IHasherService _hasherService;
                        private readonly ITransactionManager
_transactionManager;

        public AuthController(
            IMapper mapper,
            IAuthService authService,
            IOAuthService oAuthService,
            IUnitOfWork unitOfWork,
            IHasherFactory hasherFactory,
            ITransactionManager transactionManager
        )
        {
            _mapper = mapper;
            _authService = authService;
            _oauthService = oAuthService;
            _unitOfWork = unitOfWork;
                                                _hasherService =
hasherFactory.GetHasher(Application.Common.Enums.Hasher
Type.Sha256);
            _transactionManager = transactionManager;
        }

        [HttpPost("register")]
          public async Task<ActionResult> Register([FromBody]
RegisterUserRequest registerUserRequest)
        {
                                var registerUserCommand =
_mapper.Map<RegisterUserCommand>(registerUserRequest);
                                    var result = await
_authService.RegisterUser(registerUserCommand);

            if (result.Success)
            {
                return Accepted();
            }

            return result.Error switch
            {
                ErrorType.Conflict => Conflict(),
                ErrorType.Internal => StatusCode(500)
            };
        }

        [HttpGet("login")]
            public async Task<ActionResult> Login([FromQuery]
LoginUserRequest loginUserRequest)
        {
                                    var loginUserCommand =
_mapper.Map<LoginCommand>(loginUserRequest);
                                        var result = await
_authService.LoginUserAsync(loginUserCommand, 7);

            if (result.Success)
            {
                                                    return
Ok(_mapper.Map<AccessTokenResponse>(result.Data));
            }

            return BadRequest();
        }

        [HttpPost("oauth")]
                                                public async
Task<ActionResult<AccessTokenResponse>>
AuthenticateWithProvider (
                    [FromBody] ExternalAuthenticationRequest
externalAuthenticationRequest
            )
        {
                        var externalAuthenticationCommand =
_mapper.Map<ExternalAuthenticationCommand>(externalAut
henticationRequest);
                                    var result = await
_oauthService.AuthenticateExternalUserAsync(externalAuthen
ticationCommand);

            if (result.Success)
            {
                                                    return
Ok(_mapper.Map<AccessTokenResponse>(result.Data));
            }

            return BadRequest(result.Error);
        }

        [HttpPost("verify-code")]
                                                public async
Task<ActionResult<AccessTokenResponse>>
VerifyEmailAndLogin                     ([FromBody]
EmailConfirmationRequest emailConfirmationRequest)
        {
                            var emailVerificationCommand =
_mapper.Map<EmailConfirmationCommand>(emailConfirmati
onRequest);
                                    var result = await
_authService.VerifyEmailAndLoginAsync(emailVerificationCo
mmand, 7);

            if (result.Success)
            {
                                                    return
Ok(_mapper.Map<AccessTokenResponse>(result.Data));
            }
```

```
            return BadRequest();
        }

        [HttpPost("resend-email")]
                        public async Task<ActionResult>
ResendEmailVerificationCode(
                        ResendEmailConfirmationCodeRequest
resendEmailConfirmationCodeRequest
            )
        {
                    var resendEmailConfirmationCodeCommand =
_mapper.Map<ResendEmailConfirmationCodeCommand>(res
endEmailConfirmationCodeRequest);
                                        var result = await
_authService.ResendEmailVerificationCode(resendEmailConfir
mationCodeCommand);

            if (result.Success)
            {
                return Ok();
            }

            return BadRequest();
        }

        // TODO: прибрати говнокод
        [HttpPost("logout")]
        public async Task<ActionResult> LogoutUser()
        {
            var refreshToken = Request.Cookies["RefreshToken"];
            if (refreshToken != null)
            {
                                                        await
_transactionManager.ExecuteInTransactionAsync(async () =>
                {
                                var refreshTokenEntity = await
_unitOfWork.RefreshTokenRepository.GetByHashedTokenAsy
nc(_hasherService.Hash(refreshToken));

                    if (refreshTokenEntity != null)
                    {
                                                        await
_unitOfWork.RefreshTokenRepository.DeleteRefreshToken(ref
reshTokenEntity);
                        await _unitOfWork.SaveChangesAsync();
                    }
                });
            }

            return NoContent();
        }

        [HttpPost("reset-password")]
                        public async Task<ActionResult>
ResetPassword(PasswordResetRequest passwordResetRequest)
        {
                            var passwordResetCommand =
_mapper.Map<PasswordResetCommand>(passwordResetRequ
est);
                                        var result = await
_authService.SendResetPasswordLink(passwordResetComman
d);

            if (result.Success)
                return Ok();

            return BadRequest();
        }
    }
```

```
}

import { Routes } from '@angular/router';
import { userRoomsResolver } from
'./core/resolvers/user-rooms/user-rooms.resolver';
import { userResolver } from
'./core/resolvers/user/user.resolver';
import { authGuard } from './core/guards/auth/auth.guard';
import { userRoomResolver } from
'./core/resolvers/user-room/user-room.resolver';
import { roomUsersResolver } from
'./core/resolvers/room-users/room-users.resolver';
import { roomResolver } from
'./core/resolvers/room-resolver/room.resolver';
import { roomTasksResolver } from
'./core/resolvers/room-tasks-resolver/room-tasks.resolver';

export const routes: Routes = [
    {path: '',
        redirectTo: 'room',
        pathMatch: 'full'
    },
    {path: 'login', loadComponent: () =>
        import('./features/login/login.component').then(component
=> component.LoginComponent),
        data: { preload: true }
    },
    {path: 'register', loadComponent: () =>

import('./features/registration/registration.component').then(co
mponent => component.RegistrationComponent)
    },
    {path: 'reset-password', loadComponent: () =>

import('./features/password-reset/password-reset.component').t
hen(component => component.PasswordResetComponent)
    },
    {path: 'reset-password/:token', loadComponent: () =>

import('./features/new-password/new-password.component').th
en(component => component.NewPasswordComponent)
    },
    {path: 'verify-email', loadComponent: () =>

import('./features/verification-code/verification-code.componen
t').then(component                                          =>
component.VerificationCodeComponent)
    },
    {path: 'callback', loadComponent: () =>

import('./features/callback/callback.component').then(compone
nt => component.CallbackComponent)
    },
    {path: 'room', loadComponent: () =>

import('./features/room/room-actions/room-actions.component')
.then(c => c.RoomActionsComponent),
        resolve: {
            room: userRoomResolver
        },
        canActivate: [authGuard]
    },
    {path: 'room/create', loadComponent: () =>

import('./features/room/room-creation/room-creation.componen
t').then(component => component.RoomCreationComponent),
        canActivate: [authGuard]
    },
    {path: 'room/join', loadComponent: () =>
```

```
import('./features/room/room-joining/room-joining.component')
.then(component => component.RoomJoiningComponent),
      canActivate: [authGuard]
   },
   {path: 'room/:id', loadComponent: () =>

import('./features/room/room.component').then(component =>
component.RoomComponent),
      canActivate: [authGuard],
      resolve: {
         userRooms: userRoomsResolver,
         room: roomResolver,
         user: userResolver,
         tasks: roomTasksResolver,
         roomUsers: roomUsersResolver
      }
   },
   {path: 'room/:id/settings', loadComponent: () =>

import('./features/room-settings/room-settings.component').the
n(component => component.RoomSettingsComponent),
      resolve: {
         roomUsersRoles: roomUsersResolver,
         room: roomResolver
      },
      canActivate: [authGuard]
   },
   {path: '**', redirectTo: 'room', pathMatch: 'full'}
];

import {
  Component,
  OnDestroy,
  OnInit,
  signal,
  ViewChild
} from '@angular/core';

import {
  FormGroup,
  ReactiveFormsModule,
  FormBuilder,
  AbstractControl,
  FormControl
} from '@angular/forms';

import {
  Router,
  RouterLink
} from '@angular/router';

import {
  firstValueFrom,
  Subject,
  takeUntil
} from 'rxjs';

import { InputFieldComponent } from
"../../core/components/input-field/input-field.component";
import { PasswordFieldComponent } from
"../../core/components/password-field/password-field.compone
nt";
import { CheckmarkComponent } from
"../../core/components/checkmark/checkmark.component";
import { passwordNamedValidators } from
'./validators/password-named-validators';
import { createEmailNamedValidators } from
'./validators/email-named-validators';
import { usernameNamedValidators } from
'./validators/username-named-validators';
import { passwordRepetitionNamedValidators } from
'./validators/password-repetition-named-validators';
import { getValidatorsPair } from
'../../core/validators/utils/validator-type-guards';
import { FieldHintsPopoverComponent } from
"../../core/components/field-hints-popover/field-hints-popover.c
omponent";
import { AuthService } from
'../../core/services/authentication/auth-service/auth.service';
import { GoogleAuthActionButtonComponent } from
"../../core/components/google-auth-action-button/google-auth-a
ction-button.component";
import { HttpClient } from '@angular/common/http';
import { CombinedValidator } from
'../../core/validators/named-combined-validator';
import { ToastFactory } from
'../../core/services/toast-service/factories/toast-factory';
import { ToastType } from
'../../core/services/toast-service/models/toast-types';
import { ToastService } from
'../../core/services/toast-service/toast.service';

@Component({
  selector: 'app-registration',
  imports: [
    RouterLink,
    InputFieldComponent,
    PasswordFieldComponent,
    ReactiveFormsModule,
    CheckmarkComponent,
    FieldHintsPopoverComponent,
    GoogleAuthActionButtonComponent
],
  standalone: true,
  templateUrl: './registration.component.html',
  styleUrl: './registration.component.scss'
})

// TODO: подумати над фабріками, може зробити усім
валідаторам

export class RegistrationComponent implements OnInit,
OnDestroy {
                @ViewChild('username') usernameField:
InputFieldComponent | undefined;
    @ViewChild('email') emailField: InputFieldComponent |
undefined;
                @ViewChild('password') passwordField:
PasswordFieldComponent | undefined;
  @ViewChild('passwordRepetition') passwordRepetitionField:
PasswordFieldComponent | undefined;

  public focusedDefault = signal(false);
  public registerWithGoogleUrl: string = "";
  public loginUrl: string = "/login";
  public registrationForm!: FormGroup;
                public usernameNamedValidators =
usernameNamedValidators;
  public emailNamedValidators!: CombinedValidator[];
                public passwordNamedValidators =
passwordNamedValidators;
  public get passwordRepetitionNamedValidators() {
    const passwordControl = this.getControl('password');
                                                    return
passwordRepetitionNamedValidators(passwordControl!);
  }
```

```
  private destroy$ = new Subject<void>();

  public constructor(
    private router: Router,
    private formBuilder: FormBuilder,
    private _authService: AuthService,
    private _http: HttpClient,
    private _toastService: ToastService
  ) { }

  public ngOnDestroy(): void {
    this.destroy$.next();
    this.destroy$.complete();
  }

  public ngOnInit(): void {
                                this.emailNamedValidators =
createEmailNamedValidators(this._http);
    this.initForm();
  }

   public getControl(name: string): AbstractControl<any, any> |
null {
    return this.registrationForm.get(name);
  }

  public async registerWithFormAsync(): Promise<void> {
    if(this.registrationForm.invalid) {
          this._toastService.show(ToastType.Error, "Введені дані
некоректні. Перевірте форму.");
      return;
    }

            const { passwordRepetition, ...dataToSend } =
this.registrationForm.value;
                                                        await
this._authService.registerWithFormAsync(dataToSend)
    const email = this.registrationForm.get('email')!.value;
    sessionStorage.setItem('email', email);
    this.router.navigate(['verify-email']);
  }

  public changeToLogin() {
    this.router.navigate(
      ['/login'],
      { replaceUrl: true }
    );
  }

  private initForm() {
                const [usernameSync, usernameAsync] =
getValidatorsPair(usernameNamedValidators);
                const [passwordSync, passwordAsync] =
getValidatorsPair(passwordNamedValidators);
    const [emailSync, emailAsync] = getValidatorsPair(
      createEmailNamedValidators(this._http)
    );

    this.registrationForm = this.formBuilder.group({
      username: new FormControl<string>(
        '',
        {
          validators: usernameSync,
          asyncValidators: usernameAsync,
          updateOn: 'change',
          nonNullable: true
        }
      ),
      email: ['', emailSync, emailAsync],
      password: ['', passwordSync, passwordAsync],
      passwordRepetition: ['']
    });

    this.setPasswordMismatchValidator();
  }

  private setPasswordMismatchValidator(): void {
    const passwordControl = this.getControl('password');
                    const passwordRepetitionControl =
this.getControl('passwordRepetition');

    if (passwordControl && passwordRepetitionControl) {
      const [passwordRepetitionSync, passwordRepetitionAsync]
= getValidatorsPair(
        this.passwordRepetitionNamedValidators
      );

      passwordRepetitionControl.setValidators(
        passwordRepetitionSync
      );

      passwordRepetitionControl.setAsyncValidators(
        passwordRepetitionAsync
      );

      passwordControl.valueChanges
        .pipe(takeUntil(this.destroy$))
        .subscribe(() => {
          passwordRepetitionControl.updateValueAndValidity();
        })
    }
  }
}
```

```
import { AfterViewInit, Component, OnInit, signal } from
'@angular/core';
import { ActivatedRoute, Router } from '@angular/router';
import { CommonModule } from '@angular/common';
import { DropdownMenuComponent } from
"../../core/components/dropdown-menu/dropdown-menu.compo
nent";
import { IRoom } from
'../../core/types/interfaces/room-interface';
import { PaddingComponent } from
"../../core/components/padding/padding.component";
import { SettingButtonComponent } from
"../../core/components/setting-button/setting-button.component"
;
import { IUser } from '../../core/types/interfaces/user-interface';
import { CalendarComponent } from
"../../core/components/calendar/calendar.component";
import { Task } from '../../core/types/interfaces/task';
import { TaskCreationDialogComponent } from
"../../core/components/task-creation-dialog/task-creation-dialog
.component";
import { DateTime } from 'luxon';
import { DataService } from
'../../core/services/data-service/data.service';
import { UUID } from 'crypto';
import { ChartsComponent } from
"../../core/components/charts/charts.component";
import { SliderComponent } from
"../../core/components/slider/slider.component";
import { ToastService } from
'../../core/services/toast-service/toast.service';
import { ToastType } from
'../../core/services/toast-service/models/toast-types';
```

```
import { IUsersRoles } from '../../core/types/interfaces/users-roles';
import { TokenService } from '../../core/services/authentication/token-service/token.service';
import { RoomRole } from '../../core/types/room-role-enum';

@Component({
  selector: 'app-room',
  imports: [
    CommonModule,
    DropdownMenuComponent,
    PaddingComponent,
    SettingButtonComponent,
    CalendarComponent,
    TaskCreationDialogComponent,
    ChartsComponent,
    SliderComponent
],
  standalone: true,
  templateUrl: './room.component.html',
  styleUrl: './room.component.scss'
})

export class RoomComponent implements OnInit, AfterViewInit {
  public rooms: IRoom[] = [];
  public tasksForDay: Task[] = [];
  public user: IUser | null = null;
  public userRole: IUsersRoles | null = null;
  public selectedRoom: IRoom | null = null;
  public roomUsers: IUsersRoles[] = [];
  public tasksForSelectedDate: Task[] = [];
  public taskCreationDialogIsVisible = signal<boolean>(false);
  public areChartsVisible = signal<boolean>(false);
  public roomRole = RoomRole;

  constructor (
    private _route: ActivatedRoute,
    private _router: Router,
    private _dataService: DataService,
    private _toastService: ToastService,
    private _tokenService: TokenService
  ) { }

  public ngOnInit(): void {
    this._route.data.subscribe({
      next: (data) => {
        this.rooms = data['userRooms'];
        this.selectedRoom = data['room'];
        this.user = data['user'];
        this.tasksForSelectedDate = data['tasks'];
        this.roomUsers = data['roomUsers'];
      }
    })
  }

  public ngAfterViewInit(): void {
      this.userRole = this.roomUsers.find(user => user.id === this._tokenService.getUserId())!;
  }

  public async loadTasks(day: DateTime) {
                    this.tasksForSelectedDate = await this._dataService.getRoomTasks(this.selectedRoom?.id as UUID, day, day)
  }

  public changeAreChartsVisible(isVisible: boolean): void {
    this.areChartsVisible.set(isVisible);
  }

  public async toggleRoom(room: IRoom) {
    this._router.navigate([`/room/${room.id}`]);
  }

  public openTaskCreationDialog() {
    this.taskCreationDialogIsVisible.set(true);
  }

  public openRoomSettings() {

this._router.navigate([`/room/${this.selectedRoom?.id}/settings`])
  }

  public copyRoomCode() {

navigator.clipboard.writeText(this.selectedRoom!.code).then(()=>
      this._toastService.show(ToastType.Info, "Код кімнати було скопійовано")
    );
  }

  public openUserSettings() {

  }

  public async selectTaskByRandom() {
          var userIds = this.roomUsers.map(roomUsers => roomUsers.id);

    this.tasksForDay
  }
}

import { inject } from '@angular/core';
import { CanActivateFn, Router } from '@angular/router';
import { TokenService } from '../../services/authentication/token-service/token.service';
import { catchError, map, of } from 'rxjs';
import { HttpClient } from '@angular/common/http';

export const authGuard: CanActivateFn = (route, state) => {
  const tokenService: TokenService = inject(TokenService);
  const router: Router = inject(Router);
  const http: HttpClient = inject(HttpClient);

  const token = tokenService.getAccessToken();
    return http.get('/api/token/validate-access-token').pipe(
      map(() => true),
      catchError(() => {
        return of(false);
      })
    );
};

import { HttpErrorResponse, HttpInterceptorFn, HttpRequest, HttpHandlerFn } from '@angular/common/http';
import { inject } from '@angular/core';
import { TokenService } from '../../services/authentication/token-service/token.service';
import { catchError, Observable, switchMap, of, throwError, BehaviorSubject, filter, take } from 'rxjs';
import { Router } from '@angular/router';
import { SKIP_AUTH } from '../../http-context/skip-auth-context';
```

```
export const authInterceptor: HttpInterceptorFn = (req, next)
=> {
  let isRefreshing = false;
  let refreshSubject = new BehaviorSubject<boolean>(false);
  const service = inject(TokenService);
  const router = inject(Router);
   const excludedPaths = ['/.well-known/openid-configuration',
'/api/token/refresh'];

  if (excludedPaths.some(path => req.url.includes(path)))
    return next(req);

  const accessToken = service.getAccessToken();
  const clonedRequest = accessToken
    ? req.clone({
        setHeaders: {
          Authorization: `Bearer ${accessToken}`
        }
      })
    : req;

  return next(clonedRequest).pipe(
    catchError(error => handleAuthError(
      error,
      service,
      router,
      req,
      next,
      isRefreshing,
      refreshSubject
    ))
  );
};

function handleAuthError(
  error: HttpErrorResponse,
  tokenService: TokenService,
  router: Router,
  req: HttpRequest<any>,
  next: HttpHandlerFn,
  isRefreshing: boolean,
  refreshSubject: BehaviorSubject<boolean>
): Observable<any> {
  if (error.status === 401) {
    if (!isRefreshing) {
      isRefreshing = true;
      refreshSubject.next(false);

      return tokenService.refreshToken().pipe(
        switchMap(() => {
          isRefreshing = false;
          refreshSubject.next(true);
          const accessToken = tokenService.getAccessToken();
          const clonedRequest = req.clone({
            setHeaders: {
              Authorization: `Bearer ${accessToken}`
            }
          });
          return next(clonedRequest);
        }),
        catchError((err) => {
          isRefreshing = false;
          refreshSubject.next(false);
          tokenService.revokeToken();
          router.navigateByUrl('/login');
          return throwError(() => new Error("Authentication Error:
Redirected"));
        })
      );
    } else {
      return refreshSubject.pipe(
        filter(isRefreshed => isRefreshed),
        take(1),
        switchMap(() => {
          const clonedRequest = req.clone({
            setHeaders: {
                                                    Authorization: `Bearer
${tokenService.getAccessToken()}` // Отримуємо оновлений
токен
            }
          });
          return next(clonedRequest);
        })
      );
    }
  }

  router.navigateByUrl('/login');
  return throwError(() => error);
}
```