Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Програмна система для планування та моніторингу виконання особистих задач і досягнень. Серверна частина.
(тема)

Виконав:
Здобувач _4__ року навчання
групи ПЗПІ-21-1

Анастасія ЧЕРВЕНКО
(Власне ім'я, ПРІЗВИЩЕ)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Програмна інженерія
(повна назва освітньої програми)

Керівник проф. кафедри ПІ Зоя ДУДАР
(посада, Власне ім'я, ПРІЗВИЩЕ)

Допускається до захисту
Зав. кафедри _____ Кирило СМЕЛЯКОВ
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми Освітньо-професійна
Освітня програма Програмна Інженерія
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Червенко Анастасії Дмитрівні
(прізвище, ім’я, по батькові)

1. Тема роботи Програмна система для планування та моніторингу виконання особистих задач і досягнень. Серверна частина.
Затверджена наказом по університету від 19.05.2025р. №397Ст
2. Термін подання студентом роботи до екзаменаційної комісії 11.06.2025
3. Вихідні дані до роботи Розробити серверну частину програмної системи для планування та моніторингу виконання особистих задач і досягнень, яка буде забезпечувати взаємодію клієнтської та мобільної частин системи з базою даних та обробляти бізнес-логіку, з використанням мови програмування C#, фреймворку ASP.Net Core та системою управління базами даних MongoDB.
4. Перелік питань, що потрібно опрацювати в роботі
Вступ, аналіз предметної галузі, формування вимог до програмної системи, архітектура та проектування програмного забезпечення, опис прийнятих програмних рішень, тестування розробленого програмного забезпечення, висновки, додатки.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі	09.04.2025	*виконано*
2	Створення специфікації ПЗ	11.04.2025	*виконано*
3	Проектування ПЗ	17.04.2025	*виконано*
4	Розробка ПЗ	28.04.2025	*виконано*
5	Тестування ПЗ	15.05.2025	*виконано*
6	Оформлення пояснювальної записки	20.05.2025	*виконано*
7	Підготовка презентації та доповіді	23.05.2025	*виконано*
8	Попередній захист	05.06.2025	*виконано*
9	Нормоконтроль, рецензування	06.06.2025	*виконано*
10	Здача роботи у електронний архів	07.06.2025	*виконано*
11	Допуск до захисту у зав. кафедри	08.06.2025	*виконано*

Дата видачі завдання « 8 » « квітня » 2025р.

Здобувач _____
(підпис)

Керівник роботи _____ проф. кафедри ПІ Зоя ДУДАР
(підпис) (посада, Власне ім’я, ПРІЗВИЩЕ)

РЕФЕРАТ

Звіт з передатестаційної практики: 89 стор., 32 рис., 2 табл., 11 джерел.

КОНСТРУКТОР ЗАДАЧ, МОНІТОРІНГ ДОСЯГЕНЬ, НАВИЧКИ, ПІДТРИМКА, ЦІЛІ, ШТУЧНИЙ ІНТЕЛЕКТ, C#, MONGODB, REST API, VISUAL STUDIO.

Об’єкт розробки – серверна частина програмної системи для відстеження своїх досягнень або задач.

Мета розробки – створення серверної частини програмного продукту для планування та моніторингу особистих задач і досягнень, що сприятиме підтримці внутрішньої мотивації користувачів та забезпечить зручні засоби для відстеження прогресу у виконанні завдань, досягненні поставлених цілей і набутті навичок.

Метод рішення – середовище розробки Visual Studio 2022, мова програмування C#, платформа .NET Core, фреймворк ASP.NET Core Web API, система управління нереляційними базами даних MongoDB.

У результаті виконання роботи було розроблено гнучку та розширювану серверну частину повнофункціональної програмної системи для планування та моніторингу виконання особистих задач і досягнень.

ABSTRACT

TASK DESIGNER, ACHIEVEMENT MONITORING, SKILLS, SUPPORT, GOALS, ARTIFICIAL INTELLIGENCE, C#, MONGODB, REST API, VISUAL STUDIO.

The object of development is a server-side component of a software system for tracking personal achievements and tasks.

The goal of development is to create the backend of a software product designed for planning and monitoring personal tasks and achievements, aimed at supporting users' intrinsic motivation and providing convenient tools for tracking progress in completing tasks, achieving goals, and developing skills.

The solution method is using development environment Visual Studio 2022, programming language C#, .NET Core platform, ASP.NET Core Web API framework, non-relational database management system MongoDB.

As a result, a flexible and scalable server-side component of a full-featured software system for planning and monitoring the execution of personal tasks and achievements was developed.

ЗМІСТ

ВСТУП

У сучасному світі, що стрімко розвивається завдяки технологічному прогресу та соціальним трансформаціям, дедалі більше людей прагнуть до саморозвитку й особистісного зростання. Це дозволяє їм залишатися конкурентоспроможними, адаптуватися до викликів сьогодення та досягати гармонії між особистими цілями й темпом життя. У динамічному ритмі сучасності планування стало необхідним етапом на шляху до досягнення будь-якої мети. Водночас після її постановки вкрай важливо систематично відстежувати рівень власного прогресу, щоб підтримувати мотивацію та впевнено рухатись до результату.

Однак збереження стабільної мотивації, ефективне планування задач і контроль за власним розвитком є складним аналітичним процесом, що потребує якісних інструментів підтримки. У разі їх відсутності процес досягнення мети може бути суттєво ускладнений або навіть виявитися неможливим.

Основна мета програмної системи «Naviria» – надати користувачам інструмент для структурованого формування особистих цілей, розбиття їх на досяжні задачі, розвитку корисних звичок, відстеження досягнень і збереження мотивації у довгостроковій перспективі. Система поєднує логіку конструктора різних типів задач і підзадач, моніторинг прогресу у виконанні завдань, досягненні цілей, розвитку навичок, а також персоналізовану статистику, що дозволяє краще розуміти динаміку власного зростання.

З урахуванням важливості емоційної підтримки та залученості, «Naviria» впроваджує елементи гейміфікації, що включають систему рівнів користувачів, особисті досягнення, підтримку з боку інших учасників через соціальні механіки. Додатково вбудований персональний AI-асистент надає допомогу у формуванні задач, структуризації планів та відповідає на супутні запити користувача, що робить роботу із системою ще ефективнішою та зручнішою.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Огляд існуючих підходів та рішень у галузі

У сучасному інформаційно насиченому середовищі, де високий темп життя поєднується з постійною потребою до адаптації, цифрові платформи відіграють дедалі важливішу роль у підтримці особистісного зростання. Люди все частіше звертаються до технологій не лише як до інструментів продуктивності, а як до засобів для досягнення глибших цілей – формування звичок, розвиток навичок і збереження внутрішньої мотивації. У цьому контексті особливо актуальними стають системи, що інтегрують гейміфікацію, персоналізацію, планування й моніторинг діяльності.

Згідно з систематичним оглядом проведених досліджень гейміфіковані платформи, що включають елементи винагород, рівнів, досягнень і викликів, не лише підвищують мотивацію користувачів, а й сприяють розвитку когнітивних і соціальних компетенцій. Завдяки вбудованим механізмам зворотного зв’язку та статистики, користувачі отримують чітке уявлення про свій прогрес, що стимулює їх до подальших дій. Не менш важливою є персоналізація: адаптація платформи до стилю мислення, звичок і ритму життя користувача значно підвищує її ефективність і гнучкість [1].

Водночас, відсутність належних інструментів цифрового самоконтролю, призводить до частих відволікань, втрати концентрації й зниження продуктивності. Натомість навіть базові інструменти візуалізації прогресу та блокування відволікаючих факторів здатні істотно підвищити якість виконання завдань [2]. Це особливо актуально для систем, що мають на меті формування звичок та підтримку стабільної динаміки у досягненні цілей.

Ключову роль у цьому процесі відіграють саме компоненти планування та моніторингу. Дослідження доводять, що залучення студентів до планування власної діяльності, фіксації цілей та саморефлексії значно покращує навчальні результати. Учні, які використовували цифрові планери, демонстрували вищу продуктивність і краще розуміли, на якому етапі вони перебувають у досягненні

поставлених цілей. Це забезпечувало відчуття контролю над процесом і зміцнювало самооцінку [3].

Під час додаткового емпіричного підтвердження ефективності планування встановили, що здатність до попереднього планування та моніторингу є визначальною при вирішенні складних завдань. Учасники, які активно планували та контролювали хід виконання, досягали значно кращих результатів у ситуаціях з невизначеними або змінними умовами [4].

Таким чином розробка програмної системи «Naviria» повинна базуватися саме на принципах, підтверджених сучасними науковими дослідженнями. Поєднання гейміфікації (через досягнення, рівні, винагороди), персоналізації (через адаптацію задач до вимог користувача), планування (структуроване розбиття цілей на задачі) та моніторингу (візуалізація прогресу, аналітика, нагадування) дозволить створити платформу, яка не просто фіксує дії користувача, а формує в нього сталі навички саморегуляції, надає мотивацію та забезпечує сталий прогрес

1.2 Аналіз конкурентів

Далі розглянемо конкурентні рішення, які мають подібне призначення. Проаналізуємо кожен із них за ключовими критеріями, виявимо їхні переваги й обмеження та зіставимо з функціональністю програмної системи «Naviria» для кращого розуміння предметної галузі.

Перший конкурент – «Habitica» [5], додаток для управління завданнями, який перетворює ваші цілі на елементи рольової гри. Користувачі можуть створювати завдання, розвивати свого персонажа, отримувати нагороди та боротися з монстрами Додаток дозволяє об’єднуватися в партії (команди) та гільдії за інтересами, що створює соціальну відповідальність перед іншими учасниками й стимулює регулярне виконання задач. Окрім цього, «Habitica» відображає прогрес через систему рівнів, досягнень та статистики, що дозволяє користувачам наочно бачити результати своїх зусиль. У результаті кожна дія, пов’язана з особистим розвитком, отримує гейміфікований стимул, що підвищує рівень залученості та

внутрішньої мотивації кожного користувача і здійснює позитивний вплив на кількість виконаних завдань (див. рис. 1.1).

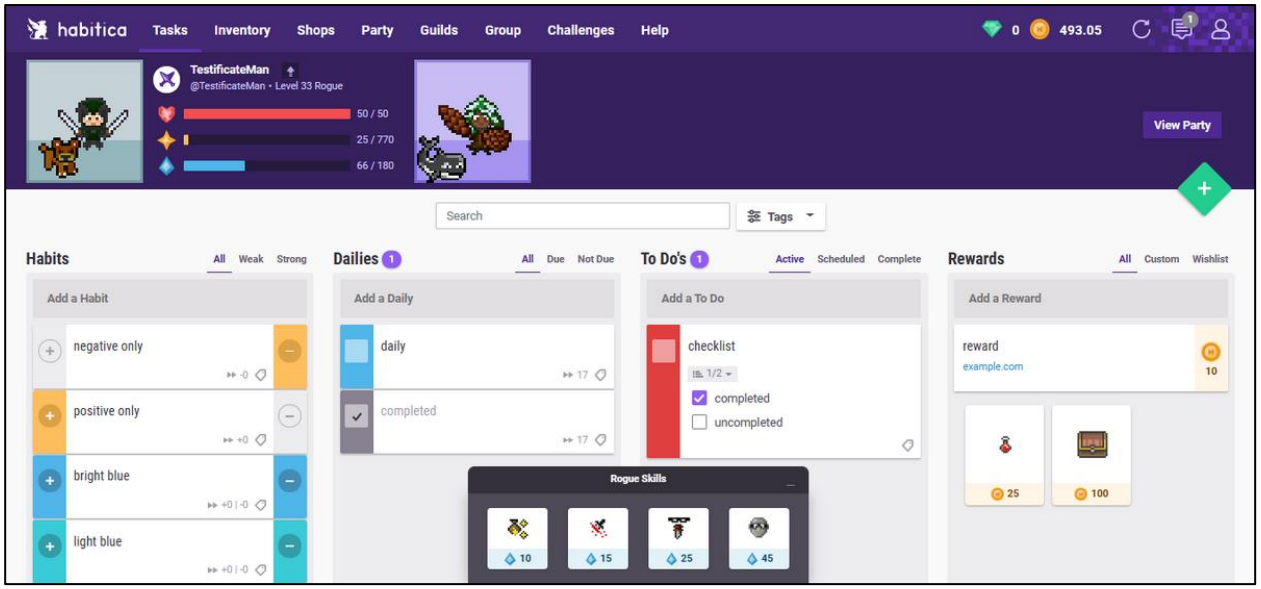

Рисунок 1.1 – Інтерфейс «Habitica»

Отже можемо виділити переваги описаного конкурента «Habitica»:

– гейміфікація у вигляді RPG-елементів, які стимулюють виконання завдань;
– соціальні функції, такі як партії та гільдії, для роботи в команді;
– візуалізація прогресу через рівні та досягнення персонажа;
– доступні версії для персональних комп’ютерів та мобільних пристроїв;
– можливість безкоштовного користування.

Аналогічним чином було виділено наступні недоліки системи:

– складний інтерфейс для нових користувачів;
– обмежена функціональність для аналізу довгострокових цілей;
– відсутність розширених інструментів аналітики;
– відсутня підтримка смарт-годинників.

Наступним конкурентом програмної системи «Naviria» є «Todoist» – один із найвідоміших інструментів для управління завданнями, що широко використовується як у професійному середовищі, так і для особистого планування [6]. Сервіс дозволяє користувачам структурувати свої задачі, створювати списки

справ, ставити дедлайни, пріоритети та відстежувати виконання. «Todoist» має багатоплатформну підтримку, включно з мобільними застосунками, вебзастосунками та персональними комп'ютерами (див. рис. 1.2), а також інтегрується з популярними сервісами для автоматизації та організації роботи.

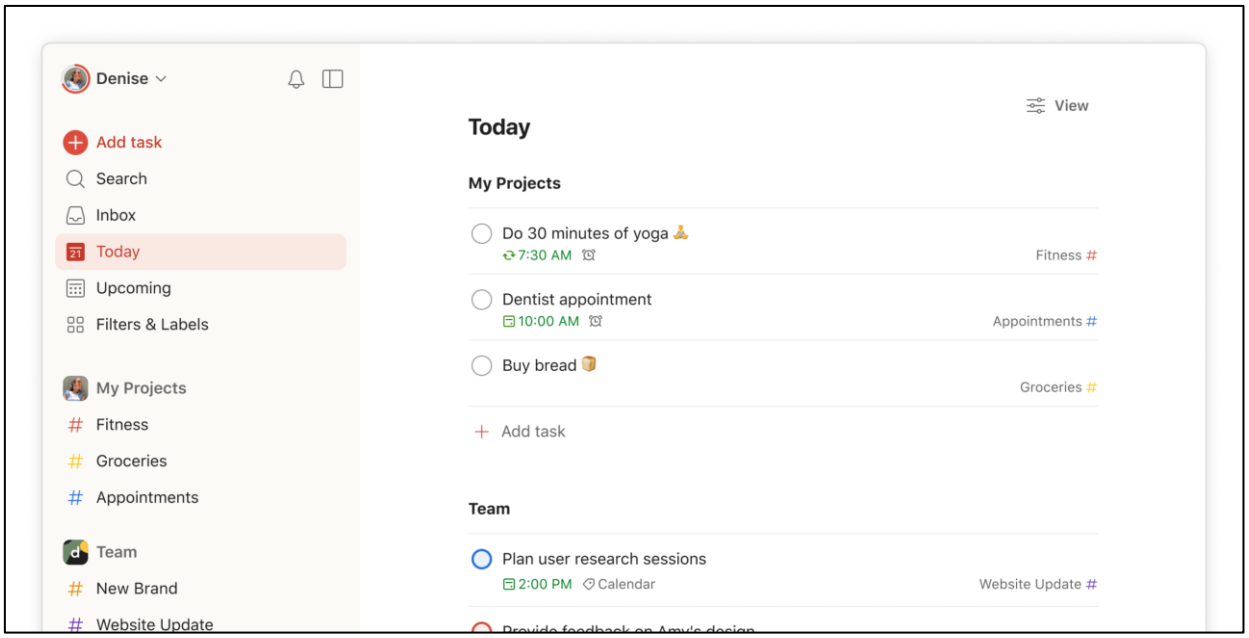

Рисунок 1.2 – Інтерфейс «Todoist»

«Todoist» підтримує концепцію «вхідної скриньки», категорій проектів та підзадач, що дозволяє користувачу гнучко формувати свою систему управління цілями. Серед найбільш корисних функцій – повторювані завдання, етикетки (теги), фільтри, а також базова аналітика продуктивності у вигляді «прогрес-барів» та лічильників активності. Отже відокремлено наступні переваги програмного застосунку «Todoist»:

- простий та зрозумілий інтерфейс;
- інтеграція з іншими сервісами, як Google Calendar, Slack, і Zapier, тощо;
- можливість створювати ієрархічні завдання з підзадачами;
- доступні версії для персональних комп'ютерів, браузерів та мобільних пристроїв;
- можливість безкоштовного використання;
- підтримка роботи з смарт-годинниками;
- можливість створення повторюваних завдань та тегів до них.

Також виділено недоліки обраної системи:

– відсутня гейміфікація;

– платні функції, такі як аналітика та історія виконаних завдань;

– відсутність інтерактивного спільного середовища для соціалізації.

Ще одним конкурентом програмної системи «Naviria» є «Trello» – відома платформа для організації роботи, що базується на принципах канбан-дошки [7]. Сервіс дозволяє візуально структурувати завдання за допомогою дошок, списків та карток, що робить його особливо зручним для проєктного менеджменту. «Trello» підтримує багатоплатформну роботу, включаючи версії для персонального комп'ютеру, браузеру та мобільних пристроїв. Також він дозволяє користувачам ефективно співпрацювати над спільними цілями (див. рис. 1.3).

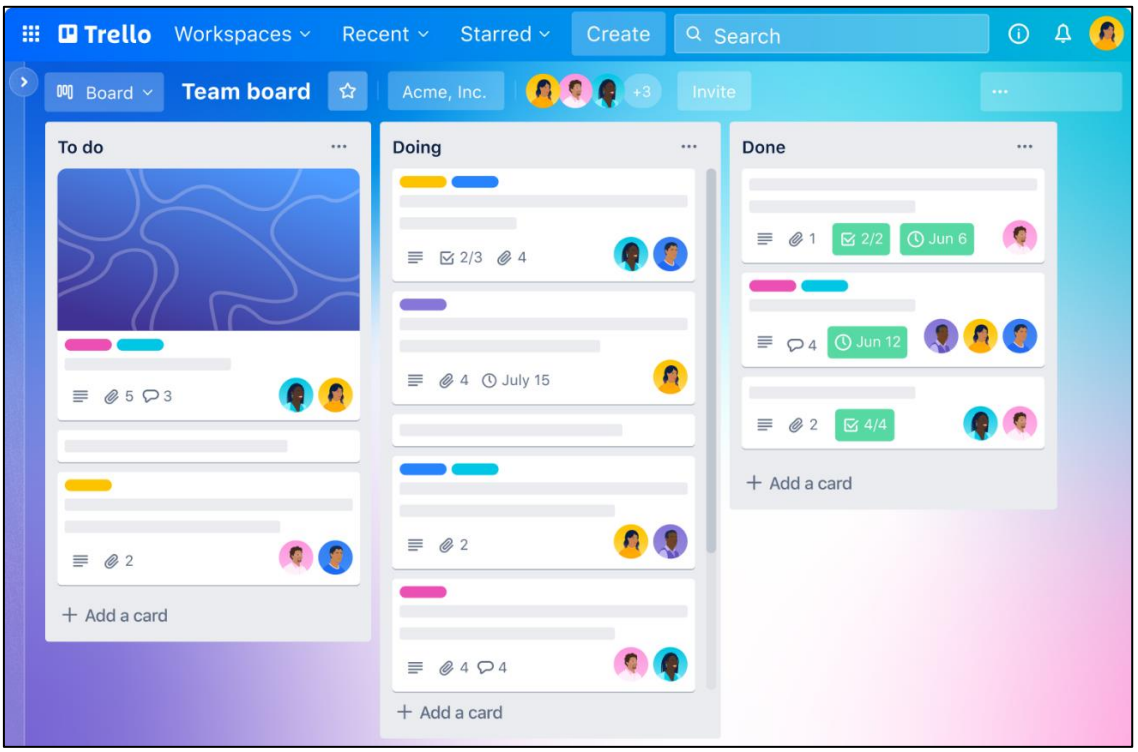

Рисунок 1.3 – Інтерфейс «Trello»

Основною перевагою «Trello» є його гнучка структура: користувачі можуть створювати необмежену кількість дошок для різних сфер життя чи проєктів, додавати картки для окремих завдань, переміщувати їх між списками («Очікує», «В роботі», «Виконано» тощо) та додавати дедлайни, чеклісти, вкладення, етикетки, коментарі. Система також дозволяє співпрацювати в реальному часі з іншими

учасниками, що робить її ефективною у командній роботі. Було відокремлено головні переваги системи:

– гнучкий інструмент для візуального управління проєктами;
– потужні функції для колаборації в командах;
– інтеграція з додатковими «power-ups» для розширення функціональності;
– доступні версії для персональних комп'ютерів, браузерів та мобільних пристроїв;
– можливість безкоштовного використання.

Також наведемо виділені недоліки системи:

– відсутність гейміфікації та мотиваційних елементів;
– не оптимізований для персонального трекінгу навичок;
– відсутня аналітика та звітність прогресу, підтримка смарт-годинників;
– відсутність функцій для заохочення довгострокового розвитку.

Наступним конкурентом програмної системи «Naviria» є «Notion», інтерфейс якого наведено на рисунку 1.4.

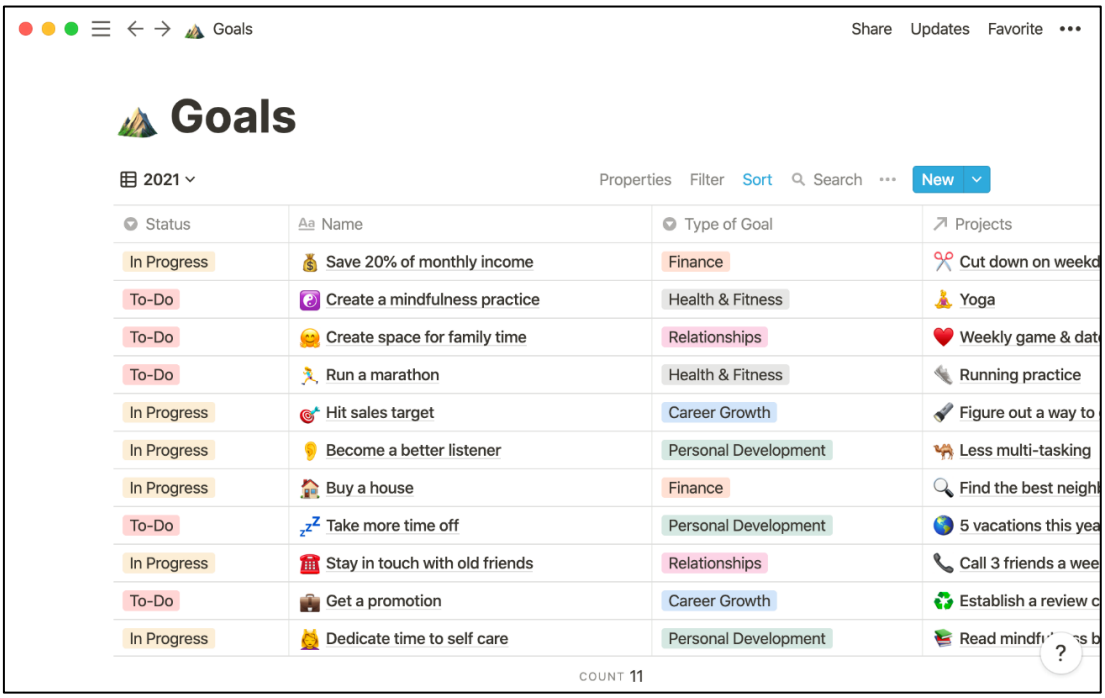

Рисунок 1.4 – Інтерфейс Notion

«Notion» – це багатофункціональний універсальний інструмент, який поєднує можливості ведення заміток, управління завданнями, створення баз даних

та організації особистих і командних просторів для роботи [8]. «Notion» позиціонує себе як «робоче місце все-в-одному», дозволяючи користувачам будувати власні цифрові робочі середовища – від простих списків справ до складних систем управління проєктами або персональних задач. Головною перевагою платформи є її виняткова гнучкість, бо інтерфейс і структура сторінок повністю налаштовуються під потреби користувача. Користувачі можуть створювати вбудовані таблиці, календарі, канбан-дошки, шаблони, сторінки, що вкладені одна в одну, а також встановлювати зв’язки між різними елементами. Notion надає інструменти для ведення як особистих нотаток, так і командної документації, проєктного планування, баз знань тощо. Таким чином було виділено наступні переваги:

- надзвичайна гнучність інструментів з можливістю налаштування під потреби користувача;
- сильна функція зберігання знань та персональних баз даних;
- підтримка командної роботи та інтеграція з іншими сервісами;
- доступні версії для персональних комп’ютерів, браузерів та мобільних пристроїв;
- можливість безкоштовного використання.

Аналогічно було виявлено наступні недоліки програмного продукту:

- високий поріг входу для нових користувачів через складність налаштувань;
- малий рівень гейміфікації або нагородної системи;
- відсутність інтегрованого трекінгу прогресу з аналітикою;
- відсутня підтримка смарт-годинників.

Ще одним конкурентом програмної системи «Naviria» є «Strides» – спеціалізований мобільний застосунок, призначений для моніторингу звичок і досягнення цілей [9]. Основна мета «Strides» – забезпечити користувачам зручний інтерфейс для постановки задач, відстеження прогресу та візуалізації результатів у вигляді графіків і діаграм. Застосунок орієнтований як на індивідуальне використання для саморозвитку, так і на підтримку щоденних звичок і рутин.

Strides дозволяє створювати чотири типи цілей: звички, SMART-цілі, середні значення (наприклад, кількість годин сну) та етапні цілі (milestones). Завдяки цьому користувачі можуть гнучко формувати персоналізований трекінг-сценарій – від щоденного споживання води до багаторічного плану з професійного розвитку. Кожна мета супроводжується календарем виконання, лічильником прогресу, нотатками та графічною візуалізацією даних (див. рис. 1.5).

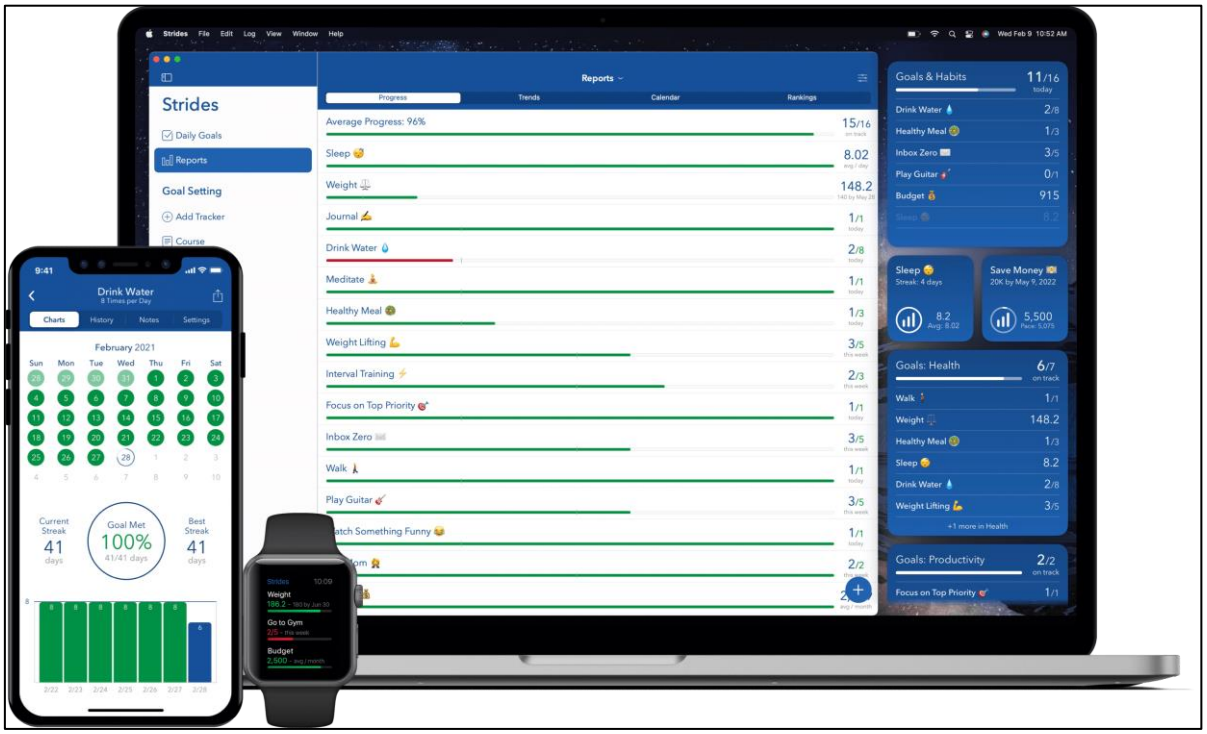

Рисунок 1.5 – Інтерфейс «Strides»

Таким чином було виділено наступні переваги:

– інтуїтивний інтерфейс для створення та відстеження цілей;
– гнучкі налаштування для короткострокових і довгострокових завдань;
– доступні версії для персональних комп’ютерів та мобільних пристроїв;
– можливість безкоштовного використання;
– візуалізація прогресу через детальні графіки та аналітику.

Також було виявлено низку недоліків:

– Відсутність соціальних функцій для взаємодії з іншими користувачами;
– Мінімальний елемент гейміфікації;
– Обмежена інтеграція з іншими платформами.

Ще одним конкурентом програмної системи «Naviria» є «Forest» – мобільний додаток, орієнтований на підвищення концентрації уваги під час роботи або навчання [10]. Його головна ідея базується на концепції «цифрового фокусу», де користувач вирощує віртуальне дерево щоразу, коли зосереджено працює протягом обраного проміжку часу. Якщо користувач перериває сесію (наприклад, відкривши інші додатки), дерево гине. З кожною успішною сесією формується «ліс продуктивності», що відображає прогрес (див. рис. 1.6).

Рисунок 1.6 – Інтерфейс «Forest»

«Forest» не є класичним таск-трекером або менеджером цілей, однак він виконує унікальну функцію – допомагає користувачеві бути присутнім у моменті, стимулюючи зосередженість та відповідальність за витрачений час. Особливістю додатка є також екологічна складова: за певну кількість віртуальних дерев, користувачі можуть зробити внесок у посадку реального дерева в рамках

партнерства з благодійною організацією «Trees for the Future», що додає етичної мотивації.

Таким чином було виділено наступні переваги системи:

– унікальний підхід до фокусування через візуалізацію прогресу;
– співпраця з друзями, вирощування спільного лісу;
– доступні версії для браузерів та мобільних пристроїв;
– сильна мотивація через вплив на реальний світ (посадка дерев).

Аналогічно у системи було знайдено низку недоліків:

– обмежена функціональність для відстеження довгострокових цілей;
– відсутність системи досягнень чи аналітичних інструментів;
– не призначений для комплексного керування цілями та навичками;
– відсутня можливість безкоштовного використання;
– відсутня підтримка смарт-годинників

За отриманою інформацією було складено таблицю порівняння конкурентів, що наведено на рисунку В.1 у додатку В.

1.3 Виявлення та вирішення проблем

Після детального аналізу предметної області та ключових конкурентів було виявлено низку функціональних недоліків, які характерні для багатьох популярних застосунків: відсутність розширеної аналітики, обмежена гейміфікація, слабка підтримка соціальних механік або нестача візуальних інструментів. Це дало змогу визначити напрями, в яких «Naviria» може отримати конкурентну перевагу. Далі наведено основні виявлені складності – як у проєктуванні власної системи, так і в недостатності існуючих рішень та шляхи їх подолання.

Більшість конкурентів надають лише базову статистику (лічильники виконаних завдань), або ж вона доступна лише у платній версії. Відсутність наочної візуалізації ускладнює відстеження довгострокових змін у системі. На основі таких конкурентів, як «Todoist», «Strides», та «Notion», було прийнято рішення розширити функції аналітики (додати статистичні звіти з виконаних завдань як особисто для кожного користувача, так і загальні для друзів у усіх

користувачів системи). За для демонстрації отриманих статистичних даних було вирішено впровадити інтерактивні діаграми, які дозволяють користувачам краще розуміти, як змінюється їхній прогрес.

Іншою важливою проблемою є недостатня гейміфікація, що не збільшує внутрішню мотивацію через відсутність системи досягнень, нагород або візуальних стимулів. З іншої сторони «Habitica» та «Forest» демонструють, як гейміфікація може мотивувати користувачів. За їх прикладом було вирішено додати ще більше варіантів нагород, а саме виділити спеціальні (особливі) досягнення та пропорційний розрахунок рівню користувача з зароблених ним балів. Також було прийнято рішення про необхідність дошок лідерів для ще більшої мотивації користувачів, бо з огляду на успіх «Trello», «Habitica» та «Notion», інтеграція соціальних функцій може підвищити цінність системи.

Також і схожих проєктах бракує механізмів для соціальної взаємодії, співпраці, мотиваційної підтримки або спільного досягнення цілей. Тому у «Naviria» було звернено особливу увагу на можливість соціальної підтримки користувачів між собою, шляхом надсилання мотивуючих повідомлень друзям.

Вагомою проблемою також є відсутність ієрархії задач, категоризації або адаптивного планування. З огляду на успіх «Notion» та «Strides» було взято до уваги необхідність створення ієрархічної структури завдань, де для кожної великої цілі можна додавати декілька підзадач для кращого структурування проектів. Також було зазначено необхідність розділення задач по категоріям та надання можливості користувачам створювати власні «теги» для завдань.

Таким чином було запропоновано рішення програмною системою «Naviria» на усі виявлені проблеми. Це дозволить системі «Naviria» не лише вийти на ринок із конкурентною перевагою, а й стати інструментом, що відповідає сучасним вимогам до гнучкого, мотиваційного та продуктивного особистісного розвитку кожної окремої людини.

1.4 Постановка задачі

Для задоволення потреб цільової аудиторії (користувачів, які прагнуть до особистісного розвитку, досягнення цілей та підтримки мотивації), необхідно розробити базу даних та серверну частину програмної системи «Naviria». Back-end частина повинна реалізовувати взаємодію з базою даних, забезпечити ефективну взаємодію з клієнтською частиною через REST API, підтримувати гнучке управління цілями та завданнями, зберігати і обробляти прогрес користувачів, формувати аналітичні звіти, реалізовувати механізми гейміфікації та підтримувати елементи соціальної взаємодії.

Результатом проєкту має стати розроблена база даних та серверна частина застосунку «Naviria», яка ефективно реалізовує бізнес-логіку системи, відповідає вимогам продуктивності, безпеки й гнучкості, та закладає основу для подальшого розширення функціоналу. Очікується, що система буде конкурентоспроможною, зручною для інтеграції, придатною для навантаження з боку великої кількості користувачів і здатною забезпечити сучасний мотиваційний досвід.

2 ФОРМУВАННЯ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

2.1 Постановка мети

Метою виконання роботи є розробка серверної частини програмної системи «Naviria» для відстеження своїх досягнень або задач.

У серверній частині системи повинна бути реалізована ключова функціональність, що забезпечує персоналізоване управління цілями та задачами користувачів. До основних механік належать створення ієрархічних цілей із підзадачами, планування дедлайнів, категоризація завдань, а також формування гейміфікованих елементів – таких як рівні, досягнення, таблиці лідерів і віртуальні нагороди. Важливою частиною є реалізація аналітичного модуля, який дозволяє генерувати статистичні звіти про прогрес користувача, а також підтримка інтерактивної взаємодії з іншими користувачами через систему друзів, спільних цілей і мотиваційного чату. Також система має підтримувати взаємодію з персональним AI-асистентом для генерації задач та допомоги користувачам з особистими питаннями. Особливу увагу необхідно приділити масштабованості, стабільності та безпеці серверної частини, щоб забезпечити безперервну роботу системи в умовах високого навантаження.

2.2 Загальний опис

У програмній системі мають бути реалізовані наступні функціональні вимоги:

- реєстрація нових користувачів системи;
- авторизація та автентифікація зареєстрованих користувачів системи;
- перегляд та редагування персонального профілю користувача;
- додавання, редагування та видалення персональних задач користувача;
- категоризація та систематизація ієрархії завдань по папкам та типізація підзадач для кожної задачі;
- графічна демонстрація аналітичних даних з відстеження процесу виконання завдань та результатів досягнених цілей;

- гейміфікація процесу виконання завдань (система рівнів, балів, таблиця лідерів);
- сповіщення про дедлайни, досягнення рівнів, отримання досягнень, підтримку від друзів, тощо;
- перегляд акаунтів інших користувачів, додавання їх у друзі, підтримка одне-одного мотивуючими цитатами;
- чат з персональним асистентом (штучним інтелектом).

Окрему увагу під час проєктування було приділено нефункціональним вимогам, що охоплюють аспекти масштабованості, безпеки, сумісності з різними пристроями та загальної якості роботи системи.

- система повинна підтримувати роботу з великою кількістю одночасно активних користувачів без помітного зниження продуктивності;
- архітектура має бути легко масштабованою за потреби;
- час відгуку сервера при обробці запитів не повинен перевищувати 2 секунд у більшості стандартних операцій;
- автентифікація користувачів повинна базуватися на сучасних протоколах (JWT);
- усі конфіденційні дані повинні зберігатися у базі даних лише у зашифрованому вигляді;
- база даних має зберігатися у хмарному сховищі з автоматичним резервним копіюванням щонайменше раз на тиждень;
- усі критичні помилки серверу, збої авторизації та системні винятки повинні логуватися для подальшої діагностики й аналізу;
- застосунок повинен коректно функціонувати як у браузері, так і у мобільній версії;
- фотографії профілів користувачів повинні зберігатися у зовнішньому хмарному сховищі (Cloudinary), без навантаження на сервер;
- застосунок має бути готовим до розширення API для інтеграції з іншими сервісами.

2.3 Загальні обмеження

Серверна частина програмної системи «Naviria» реалізується з використанням фреймворку ASP.NET Core (версія 8.0) мовою програмування C#. Для зберігання даних використовується MongoDB як нереляційна документно-орієнтована база даних, розміщена у хмарному середовищі (MongoDB Atlas). Для інтеграції AI-асистента використовується OpenAI GPT API, який дозволяє отримувати відповіді від штучного інтелекту. Виходячи з використання наведених технологій було визначено мінімальні системні вимоги:

- процесор: Intel Core i3-8100 або AMD Ryzen 3 3200G;
- оперативна пам’ять: 2–4 GB RAM;
- місце на диску: 10 GB SSD (для кешу, журналів, тимчасових файлів;
- операційна система: Windows 10/11 x64 або Ubuntu Server 22.04 LTS;
- .NET SDK: .NET 8.0 (ASP.NET Core);
- стабільне підключення до Інтернету для обробки зовнішніх API-запитів.

Зазначені вимоги розраховані на підтримку до 500 активних користувачів. У разі зростання навантаження, система має бути адаптована шляхом збільшення ресурсів та масштабування.

2.4 Припущення та залежності

Для забезпечення коректної розробки та оцінки функціональності системи було сформульовано низку припущень, які відображають очікувані умови експлуатації та поведінку користувачів:

- система вимагає постійного підключення до Інтернету для коректної роботи більшості функцій, включаючи обмін із сервером, сповіщення та взаємодію з AI-асистентом;
- користувач не може створювати або зберігати цілі, задачі та прогрес, якщо не зареєстрований та не автентифікований у системі;
- AI-асистент не генерує завдання або поради, якщо користувач не надав базову інформацію про свої цілі та уподобання;

- система не дозволяє отримувати досягнення або підвищення рівня, якщо задачі було виконано з порушенням дедлайнів;
- у разі втрати підключення до Інтернету під час взаємодії з сервером, зміни не будуть збережені до моменту повторного підключення.

Окрім внутрішньої логіки системи, програмна система «Naviria» залежить від низки зовнішніх сервісів і технологій, без яких її повноцінне функціонування неможливе. Нижче наведено ключові зовнішні залежності, які враховуються під час проєктування та розгортання серверної частини.

- якщо користувач видаляє свій акаунт, усі пов’язані дані (досягнення, його задачі та підзадачі, історія взаємодій, тощо) видаляються без можливості відновлення;
- якщо користувач перевищив встановлену кількість запитів до AI-асистента, чат оновлюється і попередні повідомлення автоматично видаляються;
- у разі збоїв або перевантаження сервера можливе тимчасове обмеження в обробці сповіщень і оновлень у реальному часі;
- якщо автентифікація через сторонній сервіс (Google OAuth) недоступна, користувач може пройти лише базову авторизацію через електрону адресу поштової скриньки та пароль.

3 АРХІТЕКТУРА ТА ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Архітектурне проєктування серверної частини

Для написання серверної частини програмної системи була обрана платформа ASP.NET Core, яка забезпечує гнучкість, широкі можливості та продуктивність розробки. Системою управління баз даних була обрана MongoDB, яка відрізняється гнучкістю, масштабованістю та високою продуктивністю. Комунікація між сервером та іншими частинами здійснюється за допомогою використання REST API, бо він забезпечує простоту, гнучкість та стандартизацію взаємодії між клієнтами та серверами за допомогою HTTP-методів, сприяючи легкій інтеграції та ефективному обміну даними у системах. Розглянути взаємодію серверної частини з іншими елементами можна на UML діаграмі розгортання (див. рис. 3.1).

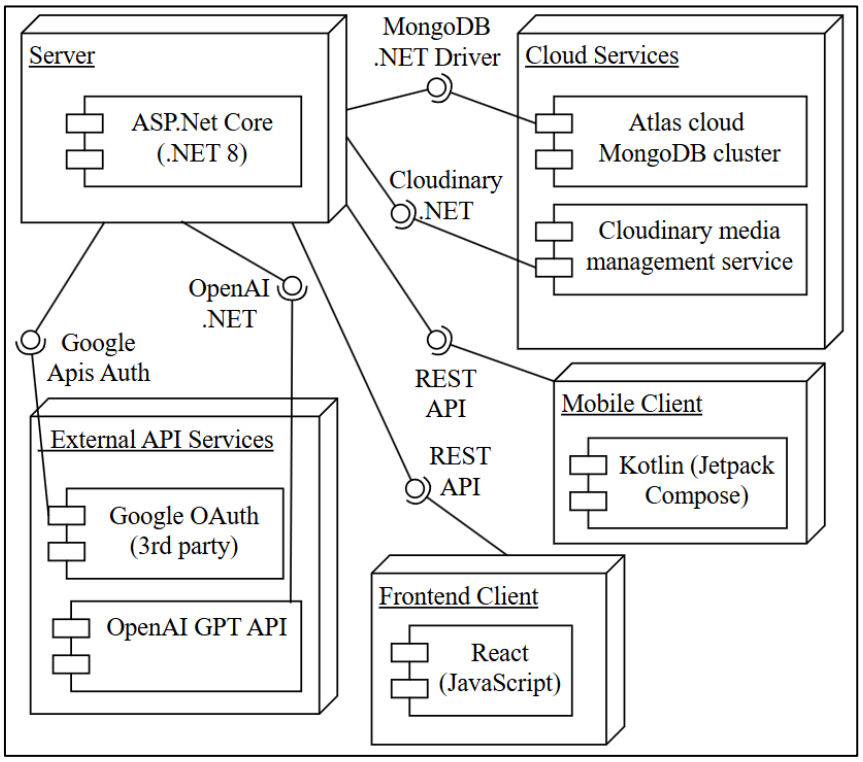

Рисунок 3.1 - UML діаграма розгортання програмної системи «Naviria»

На діаграмі розгортання зображено фізичну архітектуру програмної системи «Naviria». Дана діаграма відображає компоненти системи, програмні

модулі, а також способи їхньої взаємодії через мережеві інтерфейси. Архітектура системи побудована на основі принципів модульності, розділення відповідальностей і масштабованості, що дозволяє легко підтримувати, розширювати та адаптувати систему до нових вимог. У проєкті чітко розділено клієнтську частину (Frontend/Mobile), серверну логіку (Backend/API), хмарні сховища та зовнішні API-сервіси. Це відповідає сучасним архітектурним підходам типу service-oriented architecture. Серверна частина реалізована з використанням ASP.NET Core (.NET 8), що було обґрунтовано вище. Також система працює з хмарними сервісами, такими як MongoDB Atlas (що використовується як хмарне сховище для документно-орієнтованої бази даних, що дозволяє гнучко працювати з вкладеними структурами даних. Наприклад, це можуть бути вкладені підзадачі, досягнення, налаштування користувачів, тощо) та Cloudinary (спеціалізоване хмарне сховище, призначене для збереження та трансформації зображень та інших медіафайлів. Інтеграція здійснюється через Cloudinary .NET SDK). Також використовуються додаткові зовнішні API сервіси: OpenAI GPT API використовується для реалізації функціональності AI-асистента, який надає рекомендації, генерує задачі та консультує користувача та Google OAuth (3rd party), який забезпечує альтернативний спосіб автентифікації користувачів через їх Google-акаунти, що значно покращує UX і підвищує рівень безпеки.

Комунікація з цими сервісами здійснюється через REST API з використанням бібліотек OpenAI.NET та Google.Apis.Auth. Мобільний додаток розроблено мовою Kotlin із використанням Jetpack Compose – декларативного UI-фреймворку від Google. Мобільний клієнт взаємодіє із сервером через REST API отримуючи доступ до задач, прогресу, профілю, сповіщень та AI-функціоналу. Вебклієнт створено на базі React (JavaScript), яка є популярною бібліотекою для створення динамічних користувацьких інтерфейсів. React забезпечує високий рівень інтерактивності та дозволяє швидко створювати реактивні UI-компоненти.

Таким чином, основний протокол взаємодії між клієнтами та сервером – це REST API. Для звізку з базою даних використовується MongoDB .NET Driver. Для взаємодії з зовнішнімі сервісами – Cloudinary .NET, OpenAI .NET, Google Apis Auth. Усі компоненти функціонують через захищені HTTPS-з'єднання. Дана конфігурація системи дозволяє досягти гнучкої масштабованості, чіткої структуризації компонентів, а також забезпечити надійну взаємодію між усіма модулями. Така архітектура є оптимальною для розгортання сучасного багатофункціонального застосунку з високим рівнем персоналізації, соціалізації та інтеграції зі сторонніми AI-сервісами.

3.2 Проектування користувацьких сценаріїв взаємодії

Перед розробкою серверної частини необхідно дослідити усі можливі сценарії використання програмної системи користувачами за визначеними функціональними вимогами. Тож для визначення цих сценаріїв було розроблено UML діаграму прецедентів, де були визначені два актори: неавторизований та авторизований користувач. Такий поділ дозволяє ефективно реалізувати механізми обмеженого доступу, залежно від рівня автентифікації користувача (див. рис. Г.1 у додатку Г).

Неавторизований користувач має змогу виконувати лише базові дії: зареєструвати новий обліковий запис та авторизуватись у системі за вже створеним профілем. Після успішної авторизації користувач отримує доступ до розширеного функціоналу системи, орієнтованого на управління задачами, гейміфікацію досягнень і соціальну взаємодію.

Ключові сценарії використання охоплюють створення, редагування, перегляд та видалення особистих задач і підзадач, що дозволяє користувачу структурувати свої цілі. Додатково передбачено ведення особистої статистики та перегляд поточного прогресу, в тому числі інформації про досягнутий рівень, отримані бали та персональні досягнення. Ці функції тісно пов’язані з гейміфікованою природою системи, яка спрямована на підтримку внутрішньої мотивації користувача.

Особливу роль у системі відіграє інтегрований чат з AI-асистентом. Користувач має можливість ставити питання, отримувати персоналізовані відповіді, переглядати історію взаємодій. Комунікація з асистентом реалізована у вигляді включеного прецеденту «Отримати відповідь», що залежить від прецеденту «Задати питання».

Соціальна складова системи реалізована через функціонал пошуку та фільтрації інших користувачів, перегляду чужих профілів, додавання друзів, а також взаємної підтримки у вигляді мотиваційних дій. Надсилання, прийняття та відхилення запитів у друзі представлені як окремі прецеденти з додатковими умовами виконання.

Крім того, передбачено доступ до блоків аналітики: користувач може переглядати загальну статистику, таблиці лідерів, а також більш деталізовані звіти за категоріями задач або періодами часу.

Загалом, дана діаграма наочно демонструє логіку проєктування користувацької взаємодії із системою «Naviria», підтверджуючи орієнтованість системи на підтримку особистісного розвитку через задачі, досягнення, статистику та інтерактивний AI-компонент.

3.3 Моделювання структури даних програмної системи

Для програмної реалізації необхідно заздалегідь чітко визначити та детально описати усі необхідні для реалізації визначеного функціоналу сутності. Тому для визначення усіх необхідних сутностей та зв’язків між ними, для системи була створена ER-діаграма (див. рис. Г.2 у додатку Г). На даній ER-діаграмі представлено концептуальну модель структури даних програмної системи «Naviria», орієнтованої на підтримку користувачів у досягненні особистих цілей, задач, а також взаємодії через гейміфікацію, аналітику та соціальні функції. Модель демонструє ключові сутності системи, їхні атрибути та логічні зв’язки між ними. Структура є результатом аналізу функціональних вимог і сценаріїв використання системи, відображених у діаграмі прецедентів.

Центральною сутністю виступає колекція «Користувачі», яка містить атрибути, що дозволяють ідентифікувати користувача, зберігати його особисту інформацію (повне ім’я, гендер, дата народження, електрона адреса, пароль, друзі, повідомлення у майбутнє, дата та час реєстрації, чи користувач онлайн), статус облікового запису, рівень прогресу, а також відслідковувати активність. Особливу увагу приділено атрибутам, пов’язаним із персоналізацією (нікнейм, опис, посилання на фото), зручністю (настроювання сповіщень, відображення прогресу, перегляд статистичних даних про виконані завдання), а також гейміфікацією (бали, рівень, досягнення, досягнення, преміум-статус).

Користувач пов’язаний із кількома іншими колекціями. «Завдання» зберігають основну логіку дій користувача в системі. Кожне завдання містить гнучку конфігурацію полів: назва, опис, категорії (теги), дата дедлайну, пріоритет, включення прогресу тощо. Завдання можуть містити вкладену структуру підзадач, що враховується при зберіганні у форматі документно-орієнтованої бази даних. Таким чином реалізується принцип вкладеності в MongoDB, де підзадачі не відображаються як окрема сутність, а зберігаються як масив об’єктів всередині задачі.

Також присутня колекція «Папки» для ієрархічного структурування задач. Вони дозволяють групувати завдання за певними тематиками або напрямами (наприклад, навчання, фітнес, кар’єра) та мають базовий набір атрибутів: унікальний ідентифікатор, назву та дату створення.

Колекція «Категорії» реалізує типізацію задач, даючи змогу користувачеві фільтрувати, групувати та аналізувати завдання за заданими темами. Це особливо важливо для реалізації аналітики, яка розраховує статистику виконання задач у межах окремих категорій.

Для реалізації гейміфікації в систему інтегровано колекцію «Досягнення», де зберігається інформація про назву, опис, кількість балів і рідкісність. Досягнення зв’язуються з користувачем логікою many-to-many – у MongoDB це реалізується через вкладений масив або проміжну колекцію.

Колекція «Цитати» дозволяє зберігати надихаючі висловлювання, які відображаються користувачам як мотиваційний елемент. Кожна цитата містить текст і мову, що створює підґрунтя для багатомовної підтримки в майбутньому.

Функціонал взаємодії між користувачами реалізовано через таблицю «Запити у друзі», де зберігаються ідентифікатори відправника й отримувача, а також статус запиту (в очікуванні, прийнято, відхилено). Така структура дозволяє ефективно моделювати соціальну мережу у системі.

Користувачі також мають змогу отримувати системні повідомлення. Колекція «Сповіщення» зберігає текст, дату надсилання та статус прочитання. Окремо винесено «Повідомлення з чату з асистентом», де вказано зміст повідомлення, час створення та роль відправника (користувач чи асистент). Це дозволяє зберігати історію взаємодії з штучним інтелектом та формувати контекст персональних порад.

Кожна з сутностей описана набором атрибутів, які забезпечують унікальність (ID), описовість (назви, тексти), контроль стану (дата створення, статуси) і взаємозв’язок. Побудована модель відповідає особливостям нереляційної бази даних MongoDB, де допускається вкладеність, відсутність жорсткої схеми та використання масивів для реалізації зв’язків типу one-to-many і many-to-many. Запропонована структура забезпечує гнучкість, масштабованість і підтримку гейміфікаційних, соціальних та аналітичних механік у рамках єдиного цифрового середовища.

3.4 Проектування бази даних на основі ER-моделі

На основі попередньо сформованої ER-діаграми було побудовано логічну модель бази даних, адаптовану під використання в документно-орієнтованій системі управління базами даних MongoDB. Її наведено на рисунку Г.3 у додатку Г. Враховуючи особливості нереляційної моделі зберігання даних, така схема орієнтована на збереження пов’язаних структур у межах одного документа (вкладеність) та зменшення кількості звернень для пришвидшення роботи системи.

Логічна схема охоплює 10 основних колекцій: users, tasks, folders, categories, achievements, quotes, notifications, friends_requests, ai_chat_messages. Кожна колекція містить ключові поля з відповідними типами даних, які відповідають вимогам ефективної індексації, гнучкого пошуку та масштабованості системи.

У колекції users атрибути friends і achievements реалізовані як масиви ObjectId, що посилаються на відповідні колекції. Це типовий підхід у MongoDB, який дозволяє уникати створення проміжних таблиць і дає змогу здійснювати доступ до пов’язаних елементів у межах одного запиту. У ER-діаграмі ці зв’язки були відображені через окремі сутності або зв’язки між таблицями.

Поле level_info в users подано як об’єкт (Object), що дозволяє зберігати деталізовану інформацію про рівень користувача (наприклад, кількість балів до наступного рівня, статус, дату переходу). Це розширення порівняно з ER-діаграмою, де рівень був представлений лише як просте числове поле. Аналогічно, у tasks збережено subtasks у вигляді масиву об’єктів, а не окремої сутності, що відповідає практиці зберігання вкладених елементів у MongoDB.

Сутність «Цитати» (quotes) залишена окремою колекцією, оскільки її зв’язок з іншими колекціями не є необхідним за функціональними вимогами до системи.

Повідомлення з чату з персональним асистентом ai_chat_messages виділено в окрему колекцію, оскільки такі повідомлення можуть мати інший характер зберігання, частоту оновлення та обсяг даних, ніж звичайні системні сповіщення (notifications).

Назви полів ідентифікаторів у кожній колекції приведено до стандартного формату _id типу ObjectId, як того вимагає MongoDB для автоматичного створення індексів. Назви полів у camelCase або snake_case (наприклад, created_at, last_seen, is_online) відповідають стилістиці документів JSON і забезпечують читабельність. Дата та час зберігаються у форматі Date, що дозволяє легко виконувати сортування, фільтрацію та агрегацію. Усі логічні значення подано через Boolean (наприклад, is_online, is_notification_on, is_pro_user, is_rare), що спрощує побудову умов у запитах.

Схема повністю відповідає структурі програмної системи «Naviria», яка передбачає створення особистих задач у межах категорій і папок, отримання досягнень за виконання, можливість додавання друзів, ведення чату з AI, а також отримання системних сповіщень. Логіка гейміфікації (бали, рівень, преміум-статус) і соціальної взаємодії (друзі, запити) реалізована у простий, але розширюваний спосіб.

У процесі трансформації ER-моделі в логічну схему для MongoDB було враховано особливості документо-орієнтованої моделі зберігання даних, що дозволило спростити реалізацію зв’язків, зменшити кількість колекцій, уникнути надмірної нормалізації та забезпечити високу продуктивність при зчитуванні документів. Обрана структура є гнучкою, масштабованою та оптимізованою для роботи в реальному часі із сучасними клієнтськими застосунками.

4 ОПИС ПРИЙНЯТИХ ІНЖЕНЕРНИХ РІШЕНЬ

4.1 Організація та управління хмарним сховищем даних

У межах розробки проєкту організація зберігання та обробки всіх даних користувачів, задач, підзадач, досягнень, статистики й інших сутностей здійснюється за допомогою сучасного хмарного рішення – MongoDB Atlas. Обраний підхід дозволяє забезпечити високу надійність, масштабованість, доступність і захищеність даних відповідно до актуальних вимог сучасних програмних систем.

Усі структуровані дані системи розміщуються у кластері MongoDB Atlas, який розгортається у хмарному середовищі з вибором оптимального регіону (Europe-west) відповідно до географії користувачів та вимог до швидкодії. База даних організована у вигляді окремої колекції для кожної бізнес-сутності (наприклад, користувачі, задачі, підзадачі, досягнення, тощо), що дозволяє логічно розмежувати дані та забезпечити ефективний доступ і масштабування.

Підключення до бази даних здійснюється через захищений з’єднувальний рядок, який зберігається у .Net secrets для забезпечення його безпечного зберігання. Для встановлення з’єднання використовується офіційний драйвер MongoDB для .NET, який інтегрується в DI-контейнер застосунку через відповідні сервіси. При запуску застосунку цей драйвер ініціалізує клієнтське підключення до хмарного кластера, здійснює автентифікацію, а також налаштовує серіалізацію/десеріалізацію даних із підтримкою дискримінаторів та поліморфізму.

MongoDB Atlas забезпечує гнучке управління доступом до даних через систему ролей та користувачів. Для кожного застосунку або сервісу створюється окремий користувач бази даних з мінімально необхідними правами (наприклад, тільки читання для аналітики, читання й запис для API). Доступ до кластера дозволяється лише з визначених IP-адрес або через VPN, що значно знижує ризики несанкціонованого доступу. З’єднання до бази даних здійснюється виключно по захищеному протоколу TLS/SSL, що гарантує цілісність та конфіденційність даних під час передачі мережею. Критичні налаштування (паролі, токени, рядки

підключення) не жорстко прописуються у коді, а використовують систему секретів і конфігурацій середовища, що відповідає кращим практикам DevOps та безпеки.

MongoDB Atlas автоматично виконує регулярне резервне копіювання даних у кластері відповідно до налаштувань. Додатково у системі реалізовано можливості ручного експорту та імпорту, для створення архівів колекцій, відновлення даних, а також інтегрування цих процесів у процедури CI/CD або disaster recovery.

Завдяки природній гнучкості MongoDB Atlas, база даних автоматично масштабується відповідно до зростання кількості користувачів і навантаження на систему. Платформа надає зручні інструменти моніторингу продуктивності (див. рис. 5.1), виявлення аномалій, аналізу запитів, що дозволяє підтримувати високу швидкодію навіть у пікові періоди.

Рисунок 4.1 – Моніторінг підключень та запитів у MongoDB Atlas

Як бачимо, застосування MongoDB Atlas у ролі хмарного сховища дозволяє гарантувати стабільність, надійність, гнучкість і безпеку зберігання даних, а також значно спрощує адміністративне управління, масштабування і підтримку всієї системи впродовж життєвого циклу проєкту.

4.2 Архітектурні принципи та структура серверної частини

Серверна частина програмного забезпечення розроблена за принципами багатошарової архітектури із чітким розділенням відповідальності та обов'язків між окремими компонентами системи. Такий підхід забезпечує модульність, масштабованість та спрощує підтримку й розвиток проєкту. На рисунку Г.4 у додатку Г зображено схему архітектури серверної частини програмного застосунку. Основі архітектури використано принцип «Separation of Concerns», що передбачає виділення окремих шарів для різних аспектів функціонування серверної частини програмної системи а саме:

- Контролери реалізують вхідні точки для обробки HTTP-запитів клієнтів. Вони отримують дані від клієнта, ініціюють відповідні бізнес-процеси через сервіси, повертають уніфіковані відповіді згідно зі стандартами REST;
- Сервіси (Services) містять бізнес-логіку застосунку, обробляють сценарії взаємодії з даними, координують роботу із зовнішніми сервісами, забезпечують трансформацію даних між сутностями (Entity) і DTO (Data Transfer Objects). Це дозволяє ізолювати складні алгоритми від контролерів та репозиторіїв;
- Репозиторії (Repositories) відповідають за доступ до даних у сховищі (MongoDB). Вони інкапсулюють деталі взаємодії з базою даних, реалізують CRUD-операції, пошук, фільтрацію та індексацію даних, забезпечують прозорість роботи із колекціями та їх структурою;
- Сутності (Entities) та DTO. Для представлення бізнес-даних у доменній моделі використовуються сутності, а для взаємодії з клієнтом — DTO, які забезпечують відокремлення внутрішньої структури від зовнішніх API та підвищують безпеку.

У проекті впроваджено повноцінну архітектуру на основі патерну Dependency Injection (DI), яка є однією з фундаментальних практик сучасної розробки програмного забезпечення на платформі ASP.NET Core. Використання DI

забезпечує слабке зв'язування між компонентами, істотно спрощує підтримку, тестування та розширення коду, а також дозволяє гнучко змінювати конкретні реалізації сервісів без втручання у бізнес-логіку додатку. Реєстрація усіх сервісів, репозиторіїв, стратегій, менеджерів, background-сервісів і допоміжних компонентів здійснюється централізовано у спеціалізованих класах конфігурації, таких як ApplicationLayerServices та ServiceCollectionExtensions. Розглянемо клас ApplicationLayerServices.cs, код якого наведено у пункті Д.1 у додатку Д. У ньому описується мапування інтерфейсів на їх конкретні реалізації через методи AddScoped, AddSingleton та інші залежно від бажаного життєвого циклу об'єкта. Таким чином, для кожного HTTP-запиту створюється власний екземпляр сервісу, що унеможливлює виникнення побічних ефектів при паралельній обробці запитів різними користувачами.

Особливістю архітектури є розмежування сервісів за функціональними доменами. До них відносяться сервіси для роботи з досягненнями, задачами, категоріями, користувачами, повідомленнями, аналітикою, хмарним зберіганням, обробкою зображень, автентифікацією, авторизацією, обробкою фонового логування, менеджменту резервного копіювання та відновлення бази даних, тощо. Загальна структура папок проекту наведена на рисунку Г.5 у додатку Г. Додатково, для впровадження складних сценаріїв бізнес-логіки використовуються патерни стратегії (наприклад, різні стратегії нарахування досягнень, кожна з яких реєструється як окремий сервіс для подальшої інжекції через конструктор).

Завдяки централізованій DI-конфігурації, здійснюється одночасно кілька важливих аспектів ініціалізації додатку, таких як: реєстрація бізнес-сервісів та репозиторіїв, налаштування підключення до бази даних MongoDB (через власні extension-методи), підключення middleware для глобальної обробки помилок, логування, крос-доменної взаємодії (CORS), генерації Swagger-документації, а також підключення зовнішніх API (наприклад, Cloudinary чи OpenAI через відповідні сервіси й об’єкти-конфігурації).

Окремо впроваджено реєстрацію hosted service. Це сервіси, що виконуються у фоновому режимі, незалежно від запитів користувача (наприклад, автоматичне

створення резервних копій бази даних раз на тиждень, розсилка сповіщень про дедлайни та перевірка виконаних задач згідно з розкладом).

Архітектурне рішення із чітким розділенням реєстрації сервісів, розширюваним DI-контейнером та централізованим middleware-процесом дає змогу швидко адаптувати додаток під зміни у вимогах, замінювати реалізації без зміни клієнтського коду, ізолювати сторонні сервіси, та впроваджувати додаткові механізми (логування, аудит, валідація, безпека) на рівні інфраструктури. Загалом, така організація є запорукою довготривалої підтримуваності, масштабованості та надійності програмної системи. Також для забезпечення стійкості до збоїв та масштабованості реалізовано асинхронну обробку основних операцій, що підвищує продуктивність системи при роботі з великою кількістю одночасних запитів від багатьох різних користувачів та клієнтів. Архітектурні рішення враховують принципи SOLID та підходи до Clean Architecture, що дозволяє розділити внутрішню логіку, інтерфейси і зовнішні залежності. Таким чином, побудова серверної частини ґрунтується на сучасних інженерних принципах, що забезпечують надійність, гнучкість і можливість подальшого масштабування інформаційної системи.

4.3 Організація структури управління задачами та підзадачами

У програмному продукті система управління задачами реалізована з урахуванням складних вимог до гнучкості, масштабованості та підтримки різних сценаріїв використання, що особливо відображено у моделюванні підзадач (subtasks). Ключовим архітектурним рішенням є використання поліморфної структури для зберігання, обробки та взаємодії із підзадачами різних типів у рамках однієї задачі, що дає змогу системі легко адаптуватися до зміни бізнес-логіки й упровадження нових функціональних можливостей без суттєвих змін у кодовій базі програми. Діаграма взаємодії класів наведена на рисунку Г.6 у додатку Г.

Кожна задача у системі може належати до одного з кількох типів: стандартна (standard), масштабована (scale), повторювана (repeatable) або така, що містить

підзадачі (with_subtasks). Тип задачі визначає її поведінку, набір підтримуваних властивостей та можливість вкладеності підзадач.

Класи задач наслідують базову структуру, у якій визначено загальні для всіх задач властивості (ідентифікатор, назва, опис, користувач, статус, терміни виконання, категорія, тощо). Для кожного типу задачі реалізовано окремий підклас:

- TaskStandard – це класична задача, що містить лише базовий набір характеристик та не містить підзадач;
- TaskScale – це масштабована задача із власною шкалою прогресу, поточним і цільовим значенням, наприклад, "прочитати 200 сторінок", "пройти 10 км", де користувач може додавати до поточного значення кількість прочинатих сторінок чи пройдених кілометрів. Одиниці вимірювання також самостійно встановлюються користувачем;
- TaskRepeatable – це задача, яку потрібно виконувати регулярно та відмічати виконаною (користувач самостійно обирає дні тижня для повторів);
- TaskWithSubtasks – це задача-композит, що може містити масив підзадач різних типів.

Вибір такої моделі дозволяє чітко розділяти логіку кожного типу задачі, зберігаючи водночас єдиний інтерфейс для роботи з ними через посилання на базовий тип. Для кожного типу задачі у DTO-структурах і в шарі мапінгу (TaskTypeMap) визначені відповідні об'єкти для створення, оновлення та перегляду (TaskStandardCreateDto, TaskScaleCreateDto тощо). Поліморфізм на рівні задач підтримується як на рівні сервісів/контролерів (через універсальні методи, які приймають базовий тип), так і на рівні збереження у базі даних. Для MongoDB впроваджено систему дискримінаторів (через task_type), що дозволяє автоматично відновлювати коректний підтип задачі під час десеріалізації.

Окрема увага приділяється задачам типу "with_subtasks". Тільки ці задачі можуть містити масив підзадач, оголошених через базовий абстрактний клас SubtaskBase. Базовий клас інкапсулює універсальні властивості для всіх підзадач

(ідентифікатор, назва, опис, статус виконання, тип), що забезпечує уніфікований підхід до роботи з колекцією підзадач незалежно від їхньої конкретної реалізації.

На основі SubtaskBase визначено три основні підтипи підзадач:

- SubtaskStandard – це елементарна підзадача, яка може мати стан виконана або невиконана (аналог чекбокса);
- SubtaskRepeatable – це підзадача з повтореннями, що підтримує масив днів виконання та облік відміток "check-in", аналогічно до TaskRepeatable;
- ScaleSubtask – це підзадача з власною прогрес-метрикою, цільовим і поточним значенням, аналоігчно до TaskRepeatable.

Усі підзадачі в задачі-композиті зберігаються у масиві, і під час серіалізації/десеріалізації за допомогою спеціальних поліморфних конвертерів (як для JSON, так і для BSON) кожна підзадача ідентифікується через дискримінатор (subtask_type). Усі мапи типів налаштовані таким чином, що відповідний підтип автоматично створюється на сервері при обробці даних, а у відповідь клієнт отримує повністю структурований масив з інформацією про тип кожного елементу.

Використання такого підходу повністю відповідає принципам SOLID, зокрема принципу відкритості/закритості, бо додавання нових типів підзадач можливе шляхом наслідування без модифікації існуючого коду логіки задач або контролерів API. Кожен підтип містить властивості й методи, притаманні саме його бізнес-логіці, що гарантує інкапсуляцію поведінки та відповідальності.

Для забезпечення підтримки складної поліморфної структури підзадач у масиві Subtasks кожної задачі у MongoDB, у проєкті застосовано гібридну систему серіалізації/десеріалізації, що інтегрує можливості як .NET (System.Text.Json), так і MongoDB (BsonClassMap). Такий підхід дозволяє прозоро зберігати й отримувати підзадачі різних підтипів у рамках одного масиву без втрати типізації та функціональності.

У шарі API і сервісів, для роботи з підзадачами через DTO, застосовуються спеціальні поліморфні конвертери (наприклад, клас PolymorphicJsonConverter<TBase>, програмний код якого наведено у пункті Д.2 у додатку Д). Їх основне завдання – правильно ідентифікувати конкретний підтип

підзадачі при серіалізації та десеріалізації даних. Для цього у кожному підтипі (наприклад, SubtaskStandard, SubtaskRepeatable, ScaleSubtask) обов'язково присутнє додаткове поле-дискримінатор (subtask_type), яке під час запису у JSON додається до кожного об'єкта підзадачі. Це поле містить значення, що однозначно відповідає підтипу (наприклад, "standard", "repeatable", "scale"). Коли клієнт надсилає масив підзадач через API, конвертер аналізує значення поля-дискримінатора (subtask_type) у кожному елементі. На основі мапи типів (див. програмний код класу SubtaskTypeMap.cs у пункті Д.3 у додатку Д) конвертер визначає, до якого класу .NET належить конкретний елемент, і виконує десеріалізацію саме у потрібний підтип, автоматично формуючи об'єкт з усіма відповідними властивостями. Це усуває необхідність у кастомних перевірках і гарантує коректність структури даних. Аналогічно, під час серіалізації підзадач у DTO для повернення у відповіді API, конвертер із метаданих об’єкта визначає підтип і автоматично додає дискримінатор у JSON. Таким чином, клієнт завжди отримує масив підзадач із вказаними типами, що дозволяє легко будувати універсальні інтерфейси для роботи з різнорідними підзадачами.

Для правильного зберігання поліморфної ієрархії у NoSQL-сховищі використовується механізм дискримінаторів MongoDB через BsonClassMap. У проєкті визначено окрему статичну конфігурацію, де кожен клас підзадачі реєструється у системі зі своїм дискримінатором (наприклад, standard, repeatable, scale). Кореневий клас (SubtaskBase) позначається як базовий, а всі наслідувані підтипи як конкретні реалізації. Коли документ задачі із вкладеним масивом підзадач зберігається у MongoDB, кожна підзадача автоматично отримує службове поле-дискримінатор (зазвичай, це subtask_type), яке додається у BSON-документ. При читанні документа з колекції, MongoDB аналізує значення дискримінатора та за допомогою зареєстрованих класів відновлює саме той тип .NET, який був збережений, не втрачаючи специфічні властивості підтипу.

Після десеріалізації даних з бази у доменні моделі (наприклад, масив SubtaskBase у TaskEntity), ці об'єкти мапуються на DTO (через SubtaskMapper), які також мають у собі поле типу і структуру, що відображає специфіку підтипу. Це

дозволяє клієнтському застосунку (чи іншим споживачам API) завжди отримувати коректно структуровані об’єкти, незалежно від їхньої ієрархії чи кількості типів у системі.

Завдяки поєднанню поліморфних JSON-конвертерів і системи дискримінаторів MongoDB, у проєкті досягається наскрізна підтримка складної ієрархії підзадач: на етапі прийому даних, їх обробки у сервісах, зберігання в базі даних та повернення через API. Це гарантує, що додавання нових підтипів підзадач, зміна структури чи розширення моделі не потребуватиме масштабного рефакторингу – усе автоматично підлаштовується завдяки універсальним конвертерам і дискримінаторам, які забезпечують повну типобезпечність і надійність збереження/читання даних.

Усі операції, пов’язані зі створенням, редагуванням, видаленням і переглядом задач, виконуються через уніфікований інтерфейс API, побудований на принципах REST. DTO, які приймають і повертають контролери, що організовані поліморфно. На стороні клієнта надсилається масив підзадач із зазначенням типу, а сервер автоматично десеріалізує кожну підзадачу у відповідний клас, що усуває потребу в ручному розборі структур та забезпечує коректну валідацію.

Контролери підтримують як оперування задачами в цілому (TaskController), так і окремими підзадачами (SubtaskController), включаючи спеціалізовані операції, наприклад, позначення повторюваної підзадачі як виконаної для певної дати, або масштабування прогресу у підзадачах з лічильниками. Всі перевірки типів виконуються автоматично за допомогою конвертерів і спеціальних мап, що дає змогу розширювати модель і функціонал системи без зміни коду клієнта.

Поліморфна модель підзадач у задачах не лише спрощує розробку та обслуговування системи, а й забезпечує її стратегічну масштабованість, зменшує ризик виникнення помилок при майбутньому рефакторингу, а також дозволяє впроваджувати нові сценарії використання (наприклад, підтримку вкладених підзадач, різні типи повторень чи специфічних лічильників) із мінімальним втручанням. Така структура гарантує однорідність логіки на всіх рівнях, починаючи від бази даних і до сервісного шару до API і клієнтського інтерфейсу,

що критично важливо для складних й довготривалих проєктів, орієнтованих на гнучкість, розширюваність і підтримуваність.

4.4 Математичні моделі й алгоритми гейміфікації

У серверній частині проєкту впроваджено багатокомпонентну систему гейміфікації, що забезпечує мотивацію користувачів до активного виконання завдань та взаємодії з сервісом. Дана підсистема базується на нарахуванні балів за різні види активності, прогресивній моделі рівнів, а також системі досягнень, які підкріплюють інтерес та залученість користувача. В основі цієї реалізації лежить кілька незалежних, але взаємопов'язаних механізмів.

Основним елементом гейміфікації є механізм нарахування балів досвіду (XP), які користувач отримує за завершення завдань, виконання підзадач, отримання нових досягнень, залучення друзів тощо. Система рівнів побудована так, щоб забезпечувати постійний інтерес до зростання: кожен наступний рівень вимагає все більшої кількості XP, що ускладнює та водночас підсилює цінність прогресу. Відповідність між накопиченою кількістю балів досвіду та рівнем користувача визначається спеціальною нелінійною формулою 4.1.

$$XP_{next} = \left\lceil \frac{50 * (level)^{2.2}}{10} \right\rceil * 10 \quad (4.1)$$

де XP_{next} – це кількість балів досвіду користувача, для наступного рівня;

$level$ – числове значення наступного рівню.

Ця формула використовується для обчислення кількості балів досвіду користувача, необхідних для переходу на наступний рівень. Завдяки такій математичній моделі підвищення рівня відбувається не лінійно, а експоненціально, що характерно для сучасних гейміфікованих систем і дає змогу утримувати увагу досвідчених користувачів, мотивуючи їх до подальшого розвитку. Оскільки рівень має цілочисельне значення, остаточне значення XP для переходу на наступний

рівень округлюється до найближчого десятка. Така математична модель ускладнює отримання кожного наступного рівня і стимулює користувача до активності.

Для розрахунку прогресу система знаходить поточний рівень користувача (для якого загальна кількість балів досвіду менша за порогове значення необхідне для переходу на наступний рівень), а також визначає частку прогресу до нового рівня як відношення різниці набраних балів до діапазону між двома рівнями, що наведено на формулі 4.2.

$$Progress = \frac{XP_{user} - XP_{current}}{XP_{next} - XP_{current}} \quad (4.2)$$

де $Progress$ – це частка виконання (від 0 до 1), що показує, скільки відсотків досвіду вже отримано на поточному рівні.

XP_{user} – це поточна кількість балів досвіду (XP), які має користувач;

XP_{next} – це кількість балів досвіду для переходу на наступний рівень

$XP_{current}$ – це кількість балів досвіду, яка необхідна для досягнення поточного рівня користувача («нижня межа» досвіду для цього рівня);

Ця формула дозволяє відобразити користувачу точний відсоток просування до наступного рівня та побудувати наочний інтерфейс мотивації.

Розрахунок балів за виконання завдання здійснюється із врахуванням низки факторів, що впливають на складність і значущість задачі, що відображено у формулі 4.3.

$$Points = 10 + 2 * Priority + 3 * Count_{subtasks} + Tags + RepeatBonus \quad (4.3)$$

де $Points$ – це кількість балів за завдання;

$Priority$ – це бонус за рівень встановленого пріоритету задачі;

$Count_{subtasks}$ – це бонус за кількість підзадач (кожна підзадача додає 3 бали);

$Tags$ – це кількість унікальних тегів, доданих до задачі;

$RepeatBonus$ – це додаткові бали за повторювані підзадачі, які враховують кількість відмічених днів (кожен відмічений день до 10 додає 2 бали).

Додатково, якщо завдання має встановлений дедлайн і виконане вчасно, підсумкові бали множаться на коефіцієнт 1.5. Якщо дедлайн пропущено, бали за завдання не нараховуються.

Після розрахунку балів при завершенні завдання система оновлює інформацію про користувача: додає бали, перераховує рівень, і, у випадку підвищення рівня, надсилає користувачу повідомлення з привітанням. Уся логіка перевірки прогресу інкапсульована у сервісі LevelService. При отриманні достатньої кількості балів користувач автоматично підвищує рівень.

Крім рівнів, для посилення мотивації впроваджено гнучку та масштабовану систему досягнень, яка дозволяє заохочувати користувачів до різноманітних форм активності. Досягнення можуть бути присуджені не лише за досягнення певної кількості виконаних задач, додавання нових друзів або завершення завдань із дедлайнами, а й за будь-які інші значущі події, що розширює сценарії взаємодії з платформою.

Архітектура підсистеми досягнень побудована на використанні патерну «стратегія» (див. програмний код інтерфейсу IAchievementStrategy.cs у пункті Д.4 у додатку Д), що дозволяє інкапсулювати перевірку конкретних умов для кожного типу досягнень у окремих класах-стратегіях. Це забезпечує максимальну гнучкість і дозволяє легко додавати нові типи досягнень без необхідності змінювати вже існуючу бізнес-логіку або повторно переписувати перевірки. Всі нові умови можуть бути реалізовані через створення нової стратегії, яка підключається до системи у вигляді нового класу, що реалізує відповідний інтерфейс.

Безпосереднє присвоєння досягнень користувачу здійснюється через спеціалізований сервіс-грантер (див. програмний код файлу AchievementGranter.cs у пункті Д.5 у додатку Д), що відповідає за додавання інформації про отримане досягнення у профіль користувача, з фіксацією часу отримання і унікального ідентифікатора досягнення. Це дозволяє коректно вести історію досягнень для кожного користувача, уникати повторного нарахування та забезпечує зручний облік успіхів. Такий підхід робить систему надзвичайно адаптивною. Розробники

можуть розширювати набір досягнень, не втручаючись у вже реалізовану логіку та не порушуючи існуючі механізми перевірки і нагородження.

Завдяки застосованій архітектурі, платформа легко масштабується для нових сценаріїв, а оновлення правил чи додавання нових досягнень не потребує суттєвих змін у загальній структурі коду або в API. Це забезпечує високий рівень підтримуваності, розширюваності та стійкості до помилок у довгостроковій перспективі розвитку проєкту.

Таким чином, система гейміфікації у даному застосунку реалізує комплексний підхід до мотивації користувача шляхом багаторівневого та багатофакторного нарахування балів, динамічного підвищення рівнів відповідно до прогресивної формули, а також впровадження спеціальних досягнень. Це забезпечує глибоку персоналізацію досвіду, підтримує високий рівень залученості й дозволяє адаптувати механіки мотивації відповідно до розвитку проєкту та потреб користувачів.

4.5 Система збору, обробки та аналітики статистичних даних користувачів

У серверній частині програмного комплексу реалізовано розвинуту систему збору, обробки й представлення статистичних даних щодо користувацької активності, виконання завдань, залучення до сервісу та досягнень. Функціональність статистики охоплює як індивідуальні показники користувачів, так і агреговані аналітичні зрізи на рівні спільноти, категорій та періодів часу.

Система статистики підтримує базові загальні показники: загальну кількість користувачів і завдань у системі, а також відсоток виконаних завдань, що дозволяє швидко оцінити загальний рівень активності на платформі. Для кожного користувача можна розрахувати кількість днів, що минули з моменту реєстрації, а також дізнатись, скільки часу пройшло з моменту запуску додатку загалом.

Для відстеження індивідуального прогресу й формування рекомендацій реалізовано підрахунок кількості відміток виконання (check-in) для повторюваних підзадач (як у розрізі одного користувача, так і для окремої підзадачі). Це дає змогу

вести детальну історію регулярної активності користувача та стимулювати його до послідовної роботи над цілями.

Важливим аналітичним інструментом є візуалізація виконаних задач у розрізі категорій. Для цього формуються дані для побудови "кругових діаграм" (pie chart), які можуть відображати як індивідуальну структуру виконаних завдань по категоріях, так і колективні показники (користувач разом із друзями та глобальна статистика всіх користувачів платформи). Такий підхід дозволяє виявити переваги та найбільш популярні напрями саморозвитку серед аудиторії. Ще однією функціональною особливістю для побудови візуального графічного матеріалу є підрахунок виконаних задач за місяцями (TasksCompletedPerMonth), що дає змогу будувати графіки динаміки активності користувачів, їхніх друзів чи всієї спільноти. Це допомагає не лише аналізувати особисту продуктивність, а й виявляти сезонні тенденції та піки/спади у залученні.

Реалізовано також механізм отримання даних для побудови таблиць лідерів застосунку, які розподіляють користувачів за рівнем, кількістю балів, відсотком виконаних завдань і кількістю досягнень. Алгоритм сортування гарантує, що у топі опиняються найбільш залучені та ефективні користувачі, а також дозволяє швидко і прозоро оцінити свій прогрес відносно інших учасників платформи.

Загальна архітектура статистичної підсистеми побудована із дотриманням принципів масштабованості й розширюваності. Всі сервіси статистики ізольовані у відповідних класах та інтерфейсах, що дає змогу легко додавати нові метрики чи змінювати алгоритми розрахунку без впливу на іншу бізнес-логіку застосунку. Таким чином, система статистики в цьому проєкті не лише задовольняє поточні аналітичні потреби, а й закладає основу для подальшого розвитку, включаючи побудову рекомендаційних моделей, інтелектуального супроводу користувача та виявлення потенційних напрямів для удосконалення функціоналу сервісу.

4.6 Інтеграція із зовнішніми інформаційними сервісами та API

У межах розробки програмного комплексу було здійснено інтеграцію з низкою зовнішніх інформаційних сервісів і API, що забезпечують як ключову, так

і допоміжну функціональність для користувачів системи. Така інтеграція дозволяє досягти високого рівня автоматизації, розширити функціональні можливості продукту та забезпечити відповідність сучасним вимогам до масштабованих хмарних сервісів.

Значну роль у роботі з мультимедійними даними, а саме зберіганні, обробці та розповсюдженні зображень, відіграє інтеграція із зовнішньою хмарною платформою Cloudinary. Cloudinary є провідним сервісом у сфері хмарного зберігання і обробки медіафайлів, який надає розробникам інтерфейс (SDK), оптимізований для швидкої та безпечної роботи із зображеннями. У межах проєкту був реалізований спеціалізований сервіс CloudinaryService (програмний код наведено у пункті Д.6 у додатку Д), який через офіційну бібліотеку Cloudinary (CloudinaryDotNet) забезпечує завантаження зображень у хмарне сховище та повернення гарантовано унікального захищеного посилання (URL) на файл. Користувацькі фото, надходять від клієнтської частини у вигляді файлів, після чого обробляються сервером і відправляються на Cloudinary із зазначенням окремої папки для логічного структурування. Вибір саме цього сервісу був зумовлений необхідністю гарантувати збереження і доступність зображень незалежно від основного серверного середовища, забезпечити автоматичне масштабування, конвертацію та високу швидкість віддачі контенту кінцевим користувачам без залучення додаткових локальних ресурсів.

Другою важливою зовнішньою інтеграцією є Google OAuth 2.0. Це розповсюджений та відомий механізм автентифікації через Google-акаунт, що істотно підвищує зручність і безпеку реєстрації користувачів. Сервіс GoogleAuthService (програмний код наведено у пункті Д.7 у додатку Д) використовує офіційну бібліотеку Google.Apis.Auth для перевірки валідності ID-токена, виданого Google, та отримання підтверджених даних профілю користувача (зокрема email). Після валідації даних відбувається пошук відповідного користувача у локальній базі даних і генерація JWT-токена для подальшої авторизації в системі. Таке рішення дає змогу зменшити поріг входу для користувачів, знижує ризик реєстрації фейкових облікових записів, автоматизує

перевірку електронної пошти, а також відповідає вимогам сучасної цифрової безпеки та GDPR.

Особливу цінність для кінцевих користувачів має інтеграція з OpenAI API через SDK бібліотеку, що дозволяє впровадити в систему інтелектуального асистента для генерації відповідей на текстові запити й автоматизованого створення задач користувача. Сервіс AssistantChatService через OpenAI (ChatGPT) здійснює динамічний діалог із користувачем у режимі реального часу, зберігає історію останніх 20 повідомлень, а також інтерпретує та валідує результати у вигляді структурованого JSON для створення задач із підзадачами. Саме OpenAI був обраний як найбільш гнучкий і потужний інструмент для реалізації асистента завдяки високій якості мовного моделювання, підтримці широкого спектру сценаріїв і швидкій інтеграції. Його використання дозволяє підвищити UX, підвищити залученість і персоналізацію сервісу.

Загалом вибір зазначених зовнішніх сервісів був зумовлений високою надійністю, доведеними перевагами продуктивності, відкритою документацією, можливістю масштабування, відповідністю стандартам безпеки та підтримкою необхідних функцій для SaaS-платформи. Інтеграція із Cloudinary дозволяє відмовитися від зберігання медіафайлів на власних серверах, делегуючи це професійній екосистемі. Залучення Google OAuth підвищує рівень захисту облікових записів і зручність входу. Використання OpenAI API формує конкурентну перевагу продукту, забезпечуючи інтелектуальну допомогу та автоматизацію рутинних процесів. Усі ці рішення гармонійно інтегруються у серверну архітектуру, значно підвищують гнучкість і сучасність розробленого програмного забезпечення та закладають міцний фундамент для подальшого масштабування і розвитку проєкту.

4.7 Специфікація REST API

Серверна частина програмного забезпечення реалізує повноцінний RESTful API, який забезпечує взаємодію клієнтських застосунків із даними та бізнес-логікою через чітко визначені ендпоінти, що відповідають сучасним стандартам

розробки веб-сервісів. Кожен контролер виступає окремим шаром взаємодії з відповідною сутністю предметної області, забезпечуючи розмежування відповідальностей, масштабованість і високий рівень безпеки. У таблиці В.2 у додатку В наведено повний список та опис усіх реалізованих ендпоінтів.

Реалізована підтримка основних CRUD-операцій (створення, читання, оновлення, видалення) для основних бізнес-сутностей: користувачі, задачі, підзадачі, категорії, папки, досягнення, цитати, друзі, запити в друзі, повідомлення (нотифікації), статистика і резервне копіювання. Всі ці ендпоінти структуровані за принципом ресурсно-орієнтованої адресації, що забезпечує логічність, передбачуваність та простоту інтеграції зі сторонніми сервісами чи клієнтськими застосунками. Контролери взаємодіють із відповідними сервісами, що інкапсулюють бізнес-логіку і відповідають за обробку даних. Наприклад, контролер задач надає методи для отримання переліку задач користувача, деталізованого перегляду, створення, редагування, видалення задач, а також для групування задач по папках, відмітки повторюваних задач як виконаних тощо. Підзадачі, які мають поліморфну структуру (стандартні, повторювані, масштабовані), обробляються окремими ендпоінтами з перевіркою типу підзадачі на рівні сервісу. Для забезпечення реєстрації та аутентифікації користувачів API підтримує класичний логін по email і паролю, а також інтеграцію з Google OAuth. Після успішної автентифікації користувач отримує JWT-токен, який використовується для подальших авторизованих запитів. Окрім цього, передбачені ендпоінти для часткового та повного оновлення профілю користувача, завантаження аватара, отримання інформації про досягнення, взаємодії з іншими користувачами (додавання в друзі, пошук користувачів, надсилання та прийняття запитів у друзі, тощо).

Особливу увагу приділено забезпеченню зворотного зв'язку з користувачем через систему повідомлень (notification), а також відстеженню статистики виконання задач та активності у застосунку. Відповідні контролери забезпечують отримання як загальної, так і детальної статистики по окремих категоріях чи підзадачах, що дозволяє реалізувати аналітику для користувача системи.

Для роботи з резервними копіями передбачені ендпоінти для створення, завантаження, імпорту резервних копій бази даних мануально, що підвищує надійність і дає змогу відновлення даних у разі непередбачуваних ситуацій.

Усі ендпоінти ретельно документуються, передбачають повернення інформативних статус-кодів HTTP (200, 201, 204 для успішних операцій, 400 для некоректних запитів, 404 для відсутніх ресурсів, 409 для конфліктів, 422 для помилок валідації, 500 для внутрішніх помилок сервера) та забезпечують обробку виняткових ситуацій із записом у журнал усіх виникнувших помилок.

Завдяки такому підходу до побудови API система відзначається високою гнучкістю, масштабованістю та зручністю для інтеграції з різноманітними клієнтськими платформами. Це забезпечує не лише коректну роботу бізнес-логіки, а й захист даних, прозорість взаємодії та якісну підтримку користувацьких сценаріїв із боку серверної частини.

4.8 Механізми обробки помилок і валідація вхідних даних

Важливою складовою забезпечення надійності та безпеки інформаційної системи є впровадження комплексних механізмів обробки помилок і багаторівневої валідації вхідних даних. У розробленому програмному продукті ці механізми реалізовано як на рівні API, так і на рівні бізнес-логіки, із використанням сучасних практик і спеціалізованих засобів середовища .NET.

Валідація даних здійснюється поетапно для кожної вхідної моделі (DTO), яка надходить до системи через API. Усі DTO класів супроводжуються відповідними анотаціями атрибутів ([Required], [MaxLength], [MinLength], [Range], [RegularExpression] тощо), що дозволяє автоматично перевіряти коректність полів ще на етапі прийому запиту контролерами. Таким чином, неправильний формат електронної пошти, недостатня довжина паролю чи недопустимі символи у текстових полях автоматично призводять до генерації помилки зі зрозумілим повідомленням для користувача.

Для більш складних бізнес-правил застосовуються спеціалізовані сервіси валідації, наприклад, UserValidationService, код якого наведено у пункті Д.8 у

додатку Д. Цей сервіс не лише перевіряє формат і зміст полів, але й здійснює асинхронну перевірку на унікальність адреси електронної поштової скриньки та нікнейму в базі даних, перевіряє коректність віку користувача, відсутність майбутніх дат у полі дати народження, а також проводить інші перевірки згідно з бізнес-вимогами (наприклад, віковий ценз, обмеження кількості балів, тощо). Для підвищення гнучкості реалізовано окремі методи для валідації створення (ValidateCreateAsync), оновлення (ValidateUpdate) та часткової модифікації користувача (ValidatePatch), що дозволяє динамічно реагувати на контекст запиту.

Окремим етапом, особливо для всіх вхідних текстових даних (наприклад, повідомлень користувача, описів задач чи пошукових запитів), застосовується сервіс MessageSecurityService (див. програмний код у пункті Д.9 у додатку Д) для перевірки на шкідливий чи підозрілий вміст. Його програмний код наведено у додатку Г ДАДТОК. Цей сервіс виконує пошук небезпечних патернів (наприклад, ознаки SQL-ін’єкції, XSS, ключові слова на кшталт script, SELECT, DROP, підозрілі вирази OR 1=1 тощо), а також аналізує повідомлення за допомогою регулярних виразів. У разі виявлення таких загроз сервіс блокує обробку отриманого повідомлення, логуючи відповідну подію та генеруючи спеціалізоване виключення (SuspiciousMessageException), чим запобігає потенційним атакам на систему.

Для забезпечення стабільності роботи застосунку кожен публічний метод контролера обгорнуто у блоки обробки виключень (try-catch), які дозволяють розмежувати типові помилки валідації, бізнес-логіки та внутрішні помилки сервера. Усі некоректні запити (наприклад, відсутність обов’язкових параметрів, передача невірних даних) повертають клієнту відповідь із статус-кодом 400 (BadRequest) та детальним описом проблеми. У випадку, якщо дані не знайдено (наприклад, користувача або задачу з певним ідентифікатором), формується відповідь із статусом 404 (NotFound). Помилки автентифікації й авторизації повертають коди 401 чи 403. Непередбачені внутрішні помилки або збої в роботі сервісів призводять до відповіді з кодом 500 (Internal Server Error) та записуються у систему логування для подальшого аналізу й усунення. Таким чином усі помилки,

які виникнуть у сервіса, будуть оброблені централізовано й детально логовані за допомогою вбудованих засобів логування ASP.NET Core, що дозволяє відстежувати нештатні ситуації, виявляти уразливості та проводити аудит системи.

Загалом у проєкті впроваджено багаторівневу систему перевірки та обробки даних, що поєднує можливості атрибутної валідації моделей, бізнес-логіки, спеціалізованих сервісів контролю безпеки й централізованої обробки помилок. Це дозволяє ефективно захищати систему від некоректних або шкідливих вхідних даних, забезпечувати високий рівень безпеки, стабільності й передбачуваності функціонування всіх API-сервісів у будь-яких сценаріях експлуатації.

4.9 Забезпечення інформаційної безпеки та контролю доступу

Забезпечення інформаційної безпеки та ефективний контроль доступу до ресурсів є ключовими аспектами проєкту, що реалізовані на декількох рівнях програмної архітектури із дотриманням сучасних стандартів захисту даних і користувацьких сесій. Система використовує механізми аутентифікації на основі токенів стандарту JWT (JSON Web Token), що забезпечує надійність і масштабованість захисту при роботі з REST API. Після успішного входу (через класичну реєстрацію або інтеграцію з Google OAuth 2.0), користувачу видається JWT, який містить основні ідентифікаційні дані та контрольні клейми, що дозволяють однозначно визначити його права доступу протягом терміну дії токена. Валідація токенів і автоматичне оновлення їх життєвого циклу здійснюється через сервіси аутентифікації (IAuthService, IGoogleAuthService, IJwtService). Усі захищені контролери й методи API марковані атрибутами [Authorize], що автоматично блокує спроби доступу для неавторизованих користувачів.

Архітектура проєкту передбачає сувору перевірку прав користувача на виконання критичних операцій. Усі сервіси виконують додаткові перевірки, наприклад, співставлення ідентифікатора користувача, що виконує операцію, з власником ресурсу, перевірку наявності відповідних прав (наприклад, зміна тільки власних задач або профілю), а також перевірку рівня доступу при роботі з чутливими або адміністративними функціями.

Для попередження поширених атак, пов’язаних із впровадженням шкідливого коду (SQL-ін’єкції, XSS, тощо), у системі впроваджено багаторівневу перевірку усіх вхідних текстових даних за допомогою MessageSecurityService. Цей сервіс здійснює аналіз повідомлень, описів, пошукових запитів тощо на предмет наявності небезпечних патернів, ключових слів, підозрілих виразів, і у разі виявлення загрози блокує подальшу обробку з детальним логуванням інциденту.

Крім цього, для забезпечення контролю за діями користувачів і моніторингу системи всі критичні події, помилки, спроби несанкціонованого доступу та атипові дії детально фіксуються у журналах (логах) із використанням вбудованого механізму логування ASP.NET Core.

Усі користувацькі паролі зберігаються лише у вигляді хешів, для чого застосовується стандартний сервіс хешування ASP.NET (PasswordHasher). Доступ до хмарної бази даних (MongoDB Atlas) здійснюється по захищеному каналу з використанням унікальних ключів і паролів, які не зберігаються у відкритому вигляді в коді, а передаються через конфігураційні файли та змінні середовища. Для підвищення надійності й захисту від втрати даних у проєкті реалізовано автоматизоване резервне копіювання бази та можливість відновлення з файлів резервних копій. Резервні копії створюються автоматично раз на 7 днів. Усі резервні копії, що були створені більше ніж місяць назад, автоматично видаляються за виключенням ситуації відсутності нових копій. У такому випадку одна копія буде збережена, навіть якщо вона була створена більше ніж місяць назад. Це зроблено з огляду на необхідність забезпечення захисту від втрати даних та економії пам’яті.

Таким чином, комплексна система безпеки та контролю доступу, що реалізована в даному програмному продукті, забезпечує надійний захист даних користувачів, стійкість до типових векторів атак, гарантує автентичність та цілісність транзакцій, а також дозволяє ефективно масштабувати рішення без ризику для безпеки на різних етапах його експлуатації.

5 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

5.1 Методи тестування та автоматизованого документування

Якість, надійність та відповідність функціональних вимог реалізованої системи забезпечуються за рахунок поєднання декількох підходів до тестування та використання сучасних засобів автоматизованого документування API.

Основні сценарії роботи з REST API перевірялися вручну із застосуванням таких інструментів, як Swagger UI та Postman. Swagger UI інтегровано у сам застосунок, що дає змогу швидко переглядати доступні ендпоінти, структуру вхідних і вихідних даних, автоматично генерувати приклади запитів, а також здійснювати тестування основних операцій (GET, POST, PUT, DELETE) безпосередньо через веб-інтерфейс (див. рисунок 6.1).

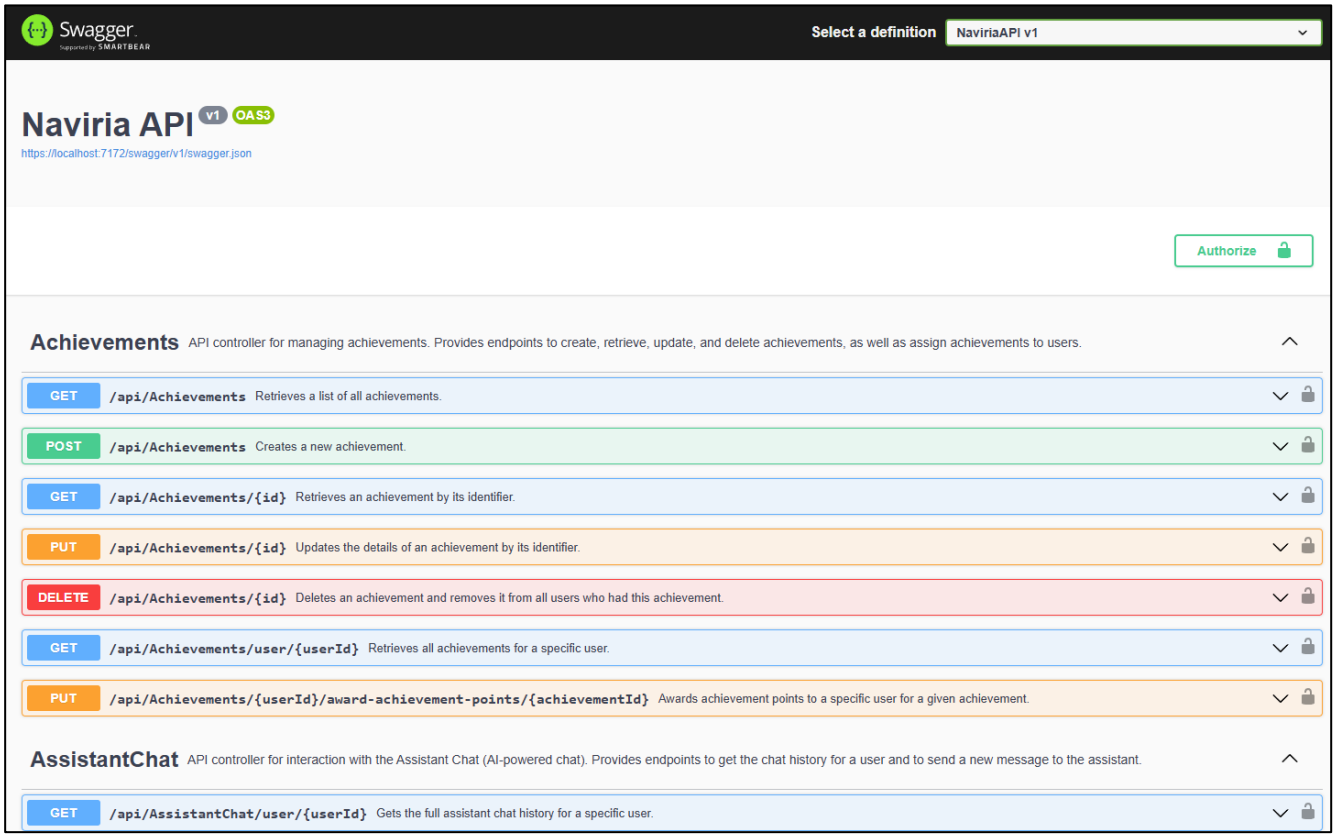

Рисунок 5.1 – Інтерфейс Swagger UI

Це дозволяє переконатися у правильності валідації, обробки помилок, відповідності статус-кодів та взаємодії з даними, зокрема при роботі зі складними структурами, як-от поліморфні підзадачі чи багаторівневі DTO.

Для розширеного функціонального тестування та симуляції реальної поведінки клієнта використовувався Postman. З його допомогою моделювалися різні користувацькі сценарії, включаючи багатокрокові запити, роботу із токенами аутентифікації, перевірку правильності авторизації, а також виявлення типових edge-case ситуацій, наприклад, некоректних даних, неіснуючих ідентифікаторів, відмов через валідацію тощо.

Вся документація для API генерується автоматично за рахунок інтеграції Swagger (OpenAPI) у структуру проєкту. Кожен ендпоінт містить XML-коментарі, що детально описують призначення, параметри, коди відповідей і приклади використання. Це забезпечує не лише зручність розробки та супроводу, а й спрощує інтеграцію сторонніх клієнтів і модулів, оскільки повна й актуальна документація завжди доступна через Swagger UI або у вигляді специфікації OpenAPI.

Комплексне тестування програмного забезпечення із застосуванням Swagger, Postman, автоматизоване документування всіх API-інтерфейсів, забезпечують високу якість продукту, скорочують час на виправлення помилок і прискорюють впровадження нових функцій. Це дозволяє підтримувати відповідність заявленим вимогам, гарантувати захищеність, стабільність і зручність роботи з системою як для кінцевих користувачів, так і для розробників та інтеграторів.

6 ВПРОВАДЖЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

6.1 Визначення плану впровадження

З метою забезпечення послідовного, контрольованого та ефективного переходу проєкту від етапу розробки до експлуатації кінцевими користувачами було розроблено план впровадження програмної системи розробляється, який наведено у таблиці 6.1.

Таблиця 6.1 План впровадження програмної системи «Naviria»

Назва етапу	Дата початку	Дата завершення
Підготовка середовища	2025-09-08	2025-09-10
Розгортання бази даних та серверу	2025-09-11	2025-09-13
Завантаження початкових даних	2025-09-14	2025-09-15
Інтеграційне та системне тестування	2025-09-16	2025-09-20
Тестування із залученням користувачів (beta)	2025-09-21	2025-09-25
Публікація застосунку у маркетплейсі	2025-09-28	2025-09-28
Період моніторингу та підтримки	2025-09-29	2025-10-10

Основними критеріями підготовки плану впровадження є поетапне тестування всіх компонентів системи, організація навчання користувачів, забезпечення стабільної роботи у середовищі та готовність до подальшої підтримки та розвитку. План впровадження містить як організаційні, так і технічні етапи, що забезпечують плавний перехід від розробки до експлуатації продукту. Така структура дозволяє уникнути більшості типових ризиків, швидко виявити та усунути недоліки, а також забезпечити позитивний досвід кінцевих користувачів із моменту запуску системи.

ВИСНОВКИ

У результаті виконання роботи було розроблено серверну частину програмної системи «Naviria», що забезпечує повний цикл персоналізованого управління цілями, задачами та досягненнями користувачів. У процесі проєктування проведено аналіз предметної області, визначено функціональні вимоги, спроєктовано архітектуру застосунку й структуру хмарної бази даних, що розміщена у кластері MongoDB Atlas. Реалізовано ключові бізнес-процеси, зокрема створення ієрархічних задач із підзадачами різних типів, планування дедлайнів, категоризацію завдань, а також комплексну систему гейміфікації (прогресивні рівні, досягнення, таблиці лідерів і віртуальні нагороди).

Система підтримує сучасну інтеграцію з зовнішніми інформаційними сервісами (Cloudinary, OpenAI API, Google OAuth), що розширює можливості для користувачів та спрощує роботу з мультимедійними файлами, штучним інтелектом та авторизацією. Особливу увагу приділено інформаційній безпеці, валідації вхідних даних, контролю доступу та захисту від небезпечних запитів, що гарантує стабільну й безпечну експлуатацію сервісу. Важливою частиною стала реалізація аналітичного модуля для отримання статистичних звітів про активність користувачів і прогрес, а також підтримка інтерактивної взаємодії через систему друзів, спільних цілей і чатів.

У роботі застосовано принципи чистої архітектури, патерни проєктування (Dependency Injection, Strategy, Polymorphism) і сучасні підходи до автоматизованого тестування та документування API. Розроблений серверний застосунок є масштабованим, підтримуваним і готовим до подальшого розширення функціоналу відповідно до майбутніх бізнес-вимог.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Jaramillo-Mediavilla L., Basantes-Andrade A., Cabezas-González M., Casillas-Martín S. Impact of Gamification on Motivation and Academic Performance: A Systematic Review // Education Sciences. – 2024. – Vol. 14, No. 6. – Article 639. – DOI: 10.3390/educsci14060639.

2. Biedermann S., Fütterer T., Fuchs M. The role of digital self-control tools in student concentration and productivity // Education and Information Technologies. – 2024. – Vol. 29. – Article 35. – DOI: https://doi.org/10.1007/s10639-023-12198-2

3. Chung H.Q., Chen V., Olson C.B. Enhancing Adolescent Writers' Revision through Goal Setting, Planning, and Reflection: A Study of the Writer's Practice Toolkit – 2021. – Vol. 34. – P. 295–319. – DOI: 10.1007/s11145-021-10186-x.

4. Greiff S., Funke J. The Role of Planning in Complex Problem Solving – 2018. – Vol. 127. – P. 156–168. – DOI: 10.1016/j.compedu.2018.08.006.

5. Habitica – Habitica Team [Електронний ресурс]. URL: https://habitica.com/static/home (дата звернення: 10.05.2025).

6. 6. Todoist – Doist [Електронний ресурс]. URL: https://www.todoist.com/ (дата звернення: 10.05.2025).

7. Trello – Atlassian [Електронний ресурс]. URL: https://trello.com/ (дата звернення: 10.05.2025).

8. Notion – Notion Labs Inc. [Електронний ресурс]. URL: https://www.notion.com/ (дата звернення: 10.05.2025).

9. Strides – StridesApp.com [Електронний ресурс]. URL: https://www.stridesapp.com/ (дата звернення: 10.05.2025).

10. Forest: Stay focused – Seekrtech [Електронний ресурс]. URL: https://forestapp.cc/ (дата звернення: 10.05.2025).

11. Github.NureChervenkoAnastasiia/2025_B_PI_PZPI-21-1_Chervenko_A_D. URL: https://github.com/NureChervenkoAnastasiia/2025_B_PI_PZPI-21-1_Chervenko_A_D