

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проекту (роботи)

бакалавра

(ступінь вищої освіти)

на тему РОЗРОБКА МОБІЛЬНОГО ЗАСТОСУНКУ

(назва теми)

ДЛЯ ТАЙМ-МЕНЕДЖМЕНТУ ТА ПЛАНУВАННЯ РОЗКЛАДУ.

DEVELOPMENT OF A MOBILE APPLICATION FOR TIME MANAGEMENT
AND SCHEDULE PLANNING

Виконав(ла): студент(ка) 4 курсу, групи КНТ-212сп

Спеціальності 122 Комп'ютерні науки

(код і найменування спеціальності)

забезпечення

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

БАРІНОВ Д.В.

(ПРИЗВИЩЕ та ініціали)

Керівник СЕРДЮК С.М.

(ПРИЗВИЩЕ та ініціали)

Рецензент ТИМЕНКО К.І.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет _____ КНТ _____
Кафедра _____ комп'ютерних наук _____
Ступінь вищої освіти _____ бакалавр _____
Спеціальність _____ 122 Комп'ютерні науки _____
(код і найменування)
Освітня програма (спеціалізація) _____ Комп'ютерні науки _____
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри _____

«_____» _____ 2025 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

БАРІНОВА Дмитра Володимировича

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Розробка мобільного застосунку для тайм-менеджменту та планування розкладу. Development of a Mobile Application for Time Management and Schedule Planning

керівник проєкту (роботи) к.т.н., доцент, СЕРДЮК Сергій Микитович,
(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від « 28 » квітня 2025 року № 209

2. Строк подання студентом проєкту (роботи) 04 червня 2025 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області. 2. Розробка архітектури програми. 3. Основні рішення з реалізації компонентів системи. 4. Керівництво програміста.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)

Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	СЕРДЮК С.М., доцент		
Нормоконтроль	ДЕЙНЕГА Л.Ю., ст. викладач		

7. Дата видачі завдання « 14 » квітня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області	2 тиждень	Розділ 1
3	Розробка архітектури програми	3 тиждень	Розділ 2
4	Розробка програми	4-5 тижні	Розділ 3
5	Тестування та експериментальне дослідження програми	6 тиждень	Розділ 4
6	Оформлення пояснювальної записки та документів до неї. Нормоконтроль та рецензування	7 тиждень	Додатки
7	Захист роботи	8 тиждень	

Студент(ка)

_____ Дмитро БАРИНОВ
(підпис) (Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи)

_____ Сергій СЕРДЮК
(підпис) (Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 139 с., 76 рис., 4 табл., 2 дод., 16 джерел.

МОБІЛЬНИЙ ЗАСТОСУНОК, РОЗКЛАД, ЗАВДАННЯ, КЕРУВАННЯ, ЛОКАЛЬНЕ ЗБЕРІГАННЯ.

Об'єкт роботи - процес управління щоденними завданнями та розкладом.

Предмет роботи – методи розробки мобільного застосунку для тайм-менеджменту та планування.

Мета роботи – розробити мобільний застосунок для тайм-менеджменту та планування, який забезпечить підвищення продуктивності роботи користувачів за рахунок автоматизації їх праці.

Матеріали, методи та технічні засоби: мова JavaScript та фреймворк React Native.

Результати. Мобільний застосунок, що надає користувачам можливість планувати та керувати своїми щоденними завданнями, додавати події до розкладу, налаштовувати нагадування. Застосунок підтримує локальне зберігання даних та забезпечує можливість роботи на декількох пристроях.

Висновки. Розроблений мобільний застосунок дозволяє користувачам ефективніше управляти своїм часом, організовувати завдання, а також отримувати своєчасні нагадування, що підвищує продуктивність.

Галузь використання – менеджмент.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 139 pages, 76 figures, 4 tables, 2 appendixes, 16 sources.

MOBILE APPLICATION, SCHEDULE, TASK MANAGEMENT, LOCAL STORAGE.

The object of work is methods of developing a mobile application for managing daily tasks.

Subject of the work is a mobile application for managing daily tasks and schedules.

Purpose of the work is to develop a mobile application for managing daily tasks and schedules that will increase user productivity by automating their work.

Materials, methods, and technical tools: JavaScript language and a React Native framework.

Results. A mobile application that allows users to plan and manage their daily tasks, add events to the schedule, and set reminders. The application supports local data storage and ensures usability across multiple devices.

Conclusions. The developed mobile application allows users to organize their time more effectively, optimize task management, and receive timely reminders, which contributes to increased productivity.

Field of use. Management.

ЗМІСТ

	С.
Перелік скорочень та умовних познач.....	8
Вступ.....	9
1 Аналіз предметної області	11
1.1 Постановка проблеми.....	11
1.2 Аналіз програмних аналогів	12
1.3 Розробка технічного завдання	19
1.3.1 Підстава для розробки.....	19
1.3.2 Мета розробки та її призначення.....	19
1.3.3 Основні вимоги до програмного продукту	19
1.3.4 Вимоги до технічних засобів, операційної системи та мови програмування.....	20
1.3.5 Вимоги до інформаційної та програмної сумісності.....	21
1.3.6 Вимоги до програмної документації.....	21
1.3.7 Порядок приймання тестування	21
1.4 Висновки за розділом	22
2 Розробка архітектури програми.....	23
2.1 Вибір середовища розробки.....	23
2.2 Вибір мови програмування	26
2.3 Проектування архітектури програмного забезпечення	30
2.4 Висновки за розділом 2	35
3 Основні рішення з реалізації компонентів системи	36
3.1 Алгоритм функціонування програмного забезпечення.....	36
3.2 Розробка структури бази даних	42
3.3 Розробка модулів програми	46
3.3.1 Головний модуль застосунку App.js	46
3.3.2 Контекст управління завданнями TaskContext	48
3.3.3 Модуль користувацького інтерфейсу TaskListScreen.....	52
3.3.4 Модуль користувацького інтерфейсу AddNoteScreen.....	59

3.3.5 Модуль користувацького інтерфейсу EditNoteScreen	67
3.3.6 Модуль користувацького інтерфейсу CalendarScreen	71
3.3.7 Модуль користувацького інтерфейсу SettingsScreen.....	75
3.3.8 Модуль навігації застосунку AppNavigator.....	77
3.4 Висновки за розділом	79
4 Керівництво програміста	80
4.1 Призначення та умови застосування програми	80
4.2 Характеристики програми	81
4.3 Структура програми	81
4.4 Вхідні та вихідні дані	84
4.4.1 Вхідні дані	84
4.4.2 Вихідні дані	84
4.5 Висновки за розділом	84
Висновки.....	86
Перелік джерел посилання	88
Додаток А Текст програми	90
Додаток Б Слайди презентації.....	132

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

БД	– база даних;
CLI	– Command-line interface;
JSON	– Java Script Object Notation;
ГБ	– гігабайт;
ОС	– операційна система;
ПК	– персональний комп'ютер;
HTML	– HyperText Markup Language;
JSX	– JavaScript XML;
XML	– Extensible Markup Language;
APK	– Android Package Kit;
IDE	– Integrated Development Environment;
SDK	– Software Development Kit;
API	– Application Programming Interface;
QR	– Quick Response;
TЗ	– технічне завдання;
CRUD	– Create, Read, Update, Delete.

ВСТУП

У щоденному ритмі багато людей намагаються тримати під контролем дедлайни, зустрічі, особисті справи — і не завжди це вдається. Іноколи через завантаженість, іноколи — через брак зручного інструменту. Незалежно від того, чи йдеться про студента з навчальним графіком, працівника офісу, чи просто людину, яка хоче встигати більше — всіх об'єднує одне: потреба якось систематизувати справи. Щось запам'ятовують у голові, щось записують на папері, а щось — у застосунках. Але останні, як показує практика, не завжди зручні.

Чимало популярних сервісів виглядають занадто складними або, навпаки, пропонують мінімум функцій. Деякі прив'язані до інтернету, інші — мають перевантажений або заплутаний інтерфейс. Усе це знижує ефективність і відлякує тих, хто хоче просто й швидко щось занотувати. До того ж, не кожен готовий витратити час на реєстрацію або підключення акаунтів для того, щоб просто додати собі справу на завтра.

Ці нюанси й стали основою для вибору теми дипломної роботи. Було поставлено конкретну мету: створити простий мобільний застосунок, який дозволяє користувачеві фіксувати щоденні завдання, розподіляти їх за категоріями та переглядати розклад у зручному вигляді — наприклад, через календар. Такий інструмент має бути легким у користуванні, не вимагати складних налаштувань і працювати навіть без підключення до інтернету.

Обраний формат, це мобільний застосунок, тому що, сьогодні смартфон є майже в кожного. Люди звикли до того, що телефон завжди поруч, тому логічно, що саме на ньому має бути інструмент, який допомагає тримати плани під контролем. Застосунок такого типу буде корисним і в особистому житті, і в навчанні, і в роботі — скрізь, де виникає потреба не загубитися в потоці справ.

Особливу увагу при проектуванні має бути приділено тому, щоб інтерфейс був інтуїтивним, а не громіздким. Щоб усе було зрозуміло ще до

прочитання інструкції. Крім того, застосунок має зберігати дані локально, без обов'язкової синхронізації з хмарою — це дозволяє використовувати його навіть там, де немає зв'язку.

Обрана тема дослідження відображає актуальний запит на зручні цифрові інструменти. Його реалізація здатна полегшити повсякденне планування великій кількості користувачів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Постановка проблеми

Коли справ і зобов'язань стає забагато ми починаємо нехтувати плануванням. Особливо це помітно в ситуаціях, коли потрібно поєднувати навчання, роботу, побутові справи та ще й не забути про особистий час. У наш час дуже багато інформації і треба щось запам'ятовувати, щось записувати на стікерах, а щось у месенджерах або випадкових нотатках. І все це перемішується, втрачається або просто не встигає оброблятися. Саме в такому середовищі потреба в елементарному контролі над власним часом перетворюється з бажання на необхідність. У таких випадках, потрібен застосунок, який зможе поєднати у звичній формі усі наші плани, думки та задачі.

Також треба приймати до уваги те, що застосунок повинен бути постійно під рукою. Так як застосунки які призначенні тільки для використання на ПК не дадуть нам змоги швидко отримати доступ до наших нотаток. На мою думку, універсальним рішенням буде створити застосунок для викорисання на мобільному телефоні, який завжди в нас під рукою. Але в такому разі в нас з'являється інша проблема, яка заключається в тому, що існує дві найбільш популярних операційних системи для мобільних телефонів, і цю проблему можна легко вирішити зробивши застосунок кросплатформним. У такому разі застосунок буде доступним на усіх мобільних телефонах.

Зараз існує досить багато застосунків для тайм-менеджменту та планування. Але, як не дивно, саме це часто й ускладнює вибір. Деякі застосунки виглядають як корпоративна платформа з сотнями налаштувань. Інші вимагають акаунт, підтвердження, синхронізацію з хмарою, ще й періодично відволікють рекламою. Усе це сприймається як перешкода — коли насправді хочеться просто відкрити програму і подивитися, що там на сьогодні заплановано. Такі рішення підіймають поріг входу для звичайного користувача. Тому застосунок повинен мати тільки найбільш необхідний

функціонал і бути з легким та з неперевантаженим інтерфейсом, який не буде плутати звичайного користувача та ускладнювати йому роботу. Також потрібно вирішити проблему з рекламою. Зараз на ринку пропонується досить багато умовно-безкоштовних застосунків в яких є досить агресивна рекламна інтеграція і це дуже заважає роботі з цими застосунками. Ще вони роблять частину функціоналу недоступним, що робить неможливим повноцінне використання. Тому треба зробити застосунок, який буде повністю безкоштовним, та не буде обмежувати користувача в використанні.

Окремо варто згадати ситуації, коли інтернету немає. Це може бути дорога, поїзд, або навіть місце з нестабільним зв'язком. У таких умовах частина популярних сервісів та застосунків виявляється абсолютно непридатною для використання. Вони не запускаються без доступу до мережі. І через це користувач не може отримати повноцінний доступ до своєї інформації. Крім цього, треба урахувати той факт, що додатки які використовують хмарні сховища зберігають вашу інформацію на серверах компаній і у разі збою в роботі цих серверів чи їх злому ви досить сильно ризикуєте втратити свої дані. Цю проблему можна виправити, якщо використовувати для зберігання власний мобільний пристрій замість послуг хмарного зберігання.

Якщо урахувати та виправити всі описані проблеми у цьому пункті, то ми отримаємо надійний, гнучкий, зручний та повністю безкоштовний мобільний застосунок який стане помічником у керуванні щоденними завданнями та розкладом.

1.2 Аналіз програмних аналогів

Наш світ не стоїть на місці і кожного дня ми отримуємо досить багато інформації. Постійно треба щось планувати, чи нотувати щоб не забути нічого важливого і все встигнути. По цій причині у сучасному світі досить великий попит на тайм-менеджмент, оскільки саме він допомагає нам в таких реаліях

зберігати драгоцінний час та все встигати нічого не забувши. Тому зараз як ніколи раніше є попит на інструменти, які спростять наше життя і допоможуть з менеджментом щоденних справ.

Якщо почати аналізувати усі інструменти які є для цього, ми зрозуміємо, що їх дуже багато і нема якогось одного найбільш популярного який би задовільняв усі наявні потреби користувача. Бо кожен застосунок пропонує свої рішення.

З великого списку застосунків було обрано три найпопулярніших на мою думку із якими я сам мав досвід користування, що допоможе мені повністю описати сильні та слабкі сторони цих застосунків та створити порівняльну таблицю. Це будуть наступні застосунки: Obsidian, Notion та Google Keep.

Почнемо з Notion. На мою думку цей застосунок більш популярний з названих трьох. Його цільова аудиторія — студенти та різні компанії. Далі будуть описані його недоліки та переваги.

Інтерфейс показано на рис. 1.1.

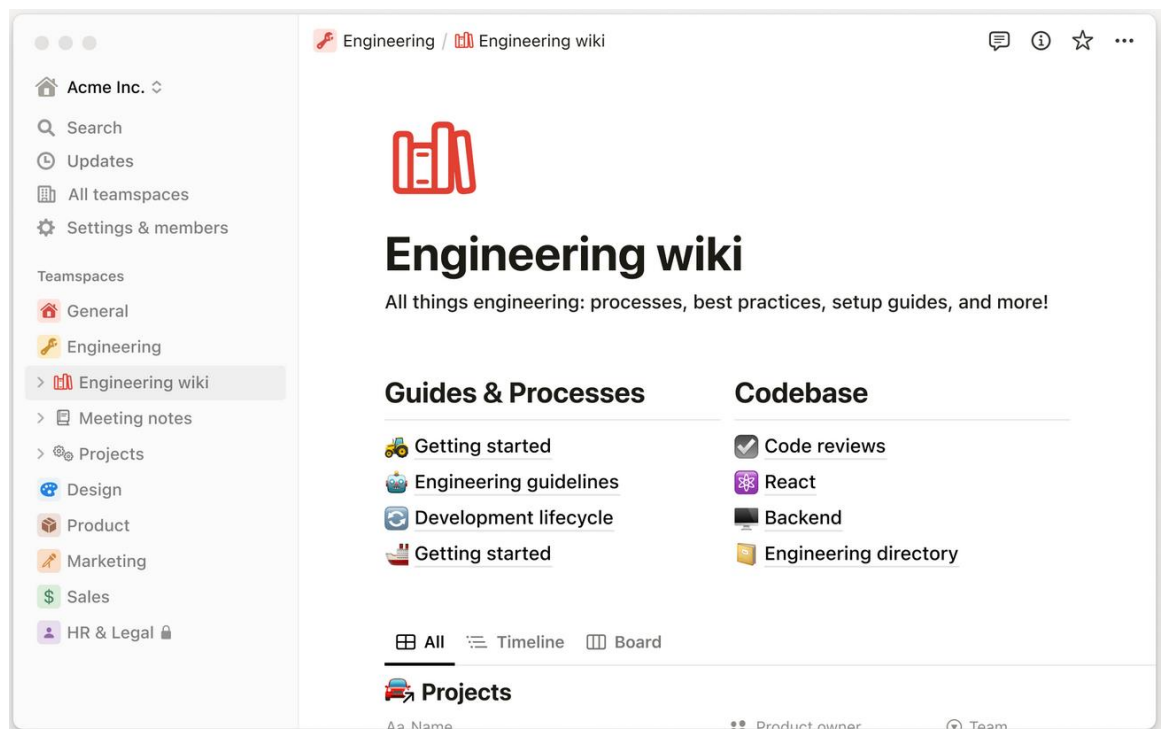


Рисунок 1.1 – Інтерфейс застосунку Notion

Переваги:

- універсальність. Можна створювати бази даних, списки, таблиці, календарі, дошки типу Kanban та документи;
- гнучкість. Кожен користувач може створити свою структуру — від простого списку справ до складної системи керування проєктами. Також є можливість будувати власні шаблони, комбінувати блоки, впорядковувати все за своїм баченням;
- підтримка співпраці. Можна надати доступ до сторінки іншим користувачам, редагувати в реальному часі. Це робить його досить цікавим варіантом для спільного планування;
- кросплатформеність. Підтримує роботу на Android, IOS, Windows, macOS, та в браузері. Також дані синхронізуються з хмарою;
- привабливий інтерфейс. Можна додавати обкладинки, іконки, кольори, що покращує візуальне сприйняття.

Недоліки:

- високий поріг входження. Через гнучкість багато користувачів губляться в налаштуваннях і структурі. Потрібно витратити час на освоєння;
- залежність від інтернету. Офлайн-режим працює обмежено. Нова інформація без підключення не завжди зберігається або синхронізується з помилками;
- складність для повсякденних завдань. Цей застосунок занадто громіздкий, якщо потрібно просто швидко записати справу чи нагадування;
- немає нативного застосунку для офлайн-баз даних. Хоч і є десктоп-версії, вся логіка зав'язана на хмарі. Немає моделі повністю автономної роботи [1].

Obsidian – це застосунок який дуже схожий на Notion, але має більше функціоналу. Саме цей застосунок я використовую кожного дня для планування особистих справ та робочих проєктів.

Інтерфейс показано на рис. 1.2.

Рисунок 1.2 – Інтерфейс застосунку Obsidian

Преваги:

- повна автономність. Obsidian працює повністю локально: усі файли зберігаються на комп'ютері чи на мобільному телефоні у форматі Markdown. Цей формат дозволяє відкривати ваші нотатки в усіх текстових редакторах. Це була основна перевага для мене при використанні цього застосунку;
- повністю безкоштовний. Весь функціонал окрім хмарного сховища безкоштовний;
- потужна система зв'язків між нотатками. Можна створювати посилання між записами, створювати карти знань і візуальні графи зв'язків. Це особливо зручно для відстеження складних або пов'язаних завдань;
- різноманітна кастомізація. Велика кількість плагінів і тем для оформлення. За допомогою плагінів користувач може додати увесь необхідний функціонал, як наприклад: календар, таблиці, mind-maps, todo-lists;

- швидкість і легкість. Легкий інтерфейс, швидко завантажується, не перевантажує систему. Придатний для користування навіть на слабких пристроях;

- підтримка кросплатформеності. Є застосунки для Windows, macOS, Linux, IOS.

Недоліки:

- не орієнтований на користування “з коробки”. Obsidian – це насамперед редактор для зв’язаних нотаток, а не планувальник. Щоб використовувати його для тайм-менеджменту та планування, потрібно власноруч встановлювати плагіни та налаштовувати шаблони;

- відсутність нативної системи нагадувань та розкладу. Немає push-сповіщень, календаря та інтеграції з системними сповіщеннями. Для повноцінного планування доводиться комбінувати Obsidian з іншими застосунками або використовувати складні плагіни;

- складність початку використання. Звичайний користувач може легко розгубитися, тому що інтерфейс не інтуїтивний. Щоб налаштувати базовий щоденник, потрібно ознайомитися з системою форматування markdown та структурою сховищ у Obsidian;

- обмеженість мобільної версії. Версія Obsidian для мобільних телефонів працює повільніше ніж десктопна. Не всі плагіни коректно підтримуються на мобільній версії застосунку;

- синхронізація тільки в ручну у безкоштовній версії. Щоб мати доступ до одних і тих самих даних на різних пристроях, треба налаштувати синхронізацію самостійно або купувати платну підписку на офіційну синхронізацію від Obsidian [2].

Google Keep – це популярний серед користувачів застосунок від Google, яким я сам раніше користувався.

Інтерфейс показано на рис. 1.3.

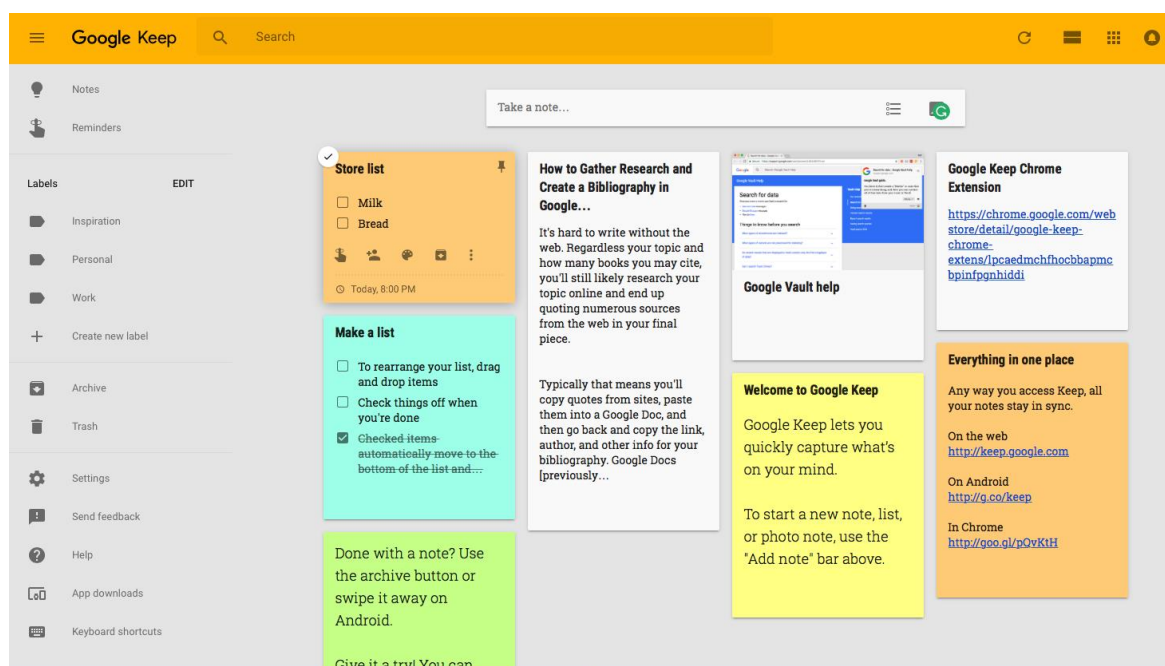


Рисунок 1.3 – Інтерфейс застосунку Google Keep

Переваги:

- простота у використанні. Інтерфейс мінімалістичний та зрозумілий з першого погляду. Не потрібно довго вчитися їм користуватися, розбиратися з шаблонами та плагінами;
- швидкість роботи. Google Keep дозволяє швидко створити нотатку в пару кліків та записати свою ідею чи іншу потрібну інформацію;
- інтеграція з Google-екосистемою. Прив'язка до облікового запису Google, що дає користувачу можливість працювати з Gmail, Google Calendar, Google Tasks. Автоматична синхронізація між усіма пристроями користувача;
- підтримка тегів і кольорового сортування. Нотатки можна сортувати за кольором і тегами, що дозволяє користувачу швидко орієнтуватися у своїх записах;
- підтримка кросплатформеності. Є додаток для мобільних операційних систем, таких як Android та IOS. Також є Web версія, на якій зручно користуватися на десктопних операційних системах, таких як Windows, Linux та macOS.

Недоліки:

- поверхневість функціоналу. У самому Google Keep немає календаря, повторюваних подій та системи дедлайнів;
- відсутність структури. Усі нотатки зберігаються в одній стрічці, це може сильно плутати користувача у своїх нотатках при їх великій кількості;
- немає підтримки категорій або проєктів. Не можна створити окрему область для конкретного проєкту чи групи завдань;
- немає розширеного формату нотаток. У Google Keep нотатка – це просто текст із можливістю вставити список або зображення. Немає таблиць, вкладених сторінок та прив'язок до інших нотаток;
- залежність від інтернету. Хоч мобільний застосунок і може надати обмежений функціонал але web версія не буде працювати без з'єднання з інтернетом [3].

Порівняльний аналіз аналогів представлено у табл. 1.1.

Таблиця 1.1 – Порівняння існуючих аналогів

Критерій	Google Keep	Notion	Obsidian
Безкоштовність	Так	Частково	Частково
Автономність	Частково	Частково	Так
Фільтрація нотаток	Ні	Так	Так
Кросплатформеність	Так	Так	Так
Легкість інтерфейсу	Так	Ні	Ні

За результатами порівняння можна побачити, що аналоги можуть пропонувати різні переваги але також вони мають і недоліки які можуть ускладнити повноцінне користування, як наприклад – частина функціоналу може бути доступна лише за платною підпискою. Через це для мене є актуальною задача по створенню власного застосунку, в якому будуть вирішені проблеми які є в аналогах.

Задача полягає у тому, щоб створити повністю безкоштовний мобільний застосунок, який повинен бути повністю автономним, що

забезпечить його надійність у місцях з відсутністю або нестабільним зв'язком до інтернету. Також застосунок повинен вирішувати проблему складного інтерфейсу і бути максимально дружнім і інтуїтивно зрозумілим для усіх користувачів. І також не дивлячись на те, що інтерфейс повинен бути легким, потрібно додати увесь потрібний функціонал, щоб застосунок закривав усі наявні потреби користувача. І до цього треба додати те, що у застосунку потрібно реалізувати кросплатформеність для того, щоб усі користувачі могли користуватися ним на операційних системах для мобільних телефонів, таких як Android та IOS.

1.3 Розробка технічного завдання

1.3.1 Підстава для розробки

Підставою для розробки служить завдання на дипломну кваліфікаційну роботу бакалавра на тему “Розробка мобільного застосунку для тайм-менеджменту та планування”, затверджене наказом № 209 від 28 квітня 2025 р. по НУ “Запорізька політехніка”.

1.3.2 Мета розробки та її призначення

Метою є розробка мобільного застосунку, за допомогою якого, можна керувати щоденними завданнями та розкладом.

1.3.3 Основні вимоги до програмного продукту

Мобільний застосунок повинен мати одного користувача. Користувач має володіти наступними навичками: володіння базовим функціоналом мобільних пристроїв на базі операційних систем Android або IOS, вміти користуватися застосунками для тайм-менеджменту та планування. Мобільний застосунок має надавати користувачу наступні можливості:

- додавати нотатки;
- зберігати нотатки;
- змінювати нотатки;
- видаляти нотатки;
- створювати нагадування на потрібний день та час;
- переглядати за допомогою календаря нотатки заплановані на той чи інший день натискаючи на відповідну дату;
- групувати свої нотатки за тегами;
- фільтрувати нотатки за тегами;
- змінювати мову застосунку.

Вимоги до інтерфейсу:

- інтуїтивно зрозумілий інтерфейс для зручності використання мобільного застосунку користувачем;
- наявність зручних елементів керування для легкого переходу між екранами застосунку.

Вхідними даними в мобільному застосунку є інформація, яку користувач буде вводити.

Основною вимогою до вихідних даних є збереження доданої інформації, або попередження про некоректне введення даних.

Вихідними даними в застосунку є збереження доданої інформації користувачем та її відображення на відповідному екрані.

1.3.4 Вимоги до технічних засобів, операційної системи і мови програмування

Програма, має розроблятися у вигляді мобільного застосунку. Він має бути написаний на JavaScript у середовищі розробки Microsoft Visual Studio Code 2020. Програма призначена для роботи на мобільному телефоні під керуванням ОС Android або IOS.

Мінімальні системні вимоги обумовлюються забезпеченням стабільної роботи ОС Android або IOS і мають бути наступними для ОС Android [4]:

- процесор з архітектурою ARMv7-A або новіший;
- мінімум 1 ГБ оперативної пам'яті;
- не менше ніж 8 ГБ вбудованої пам'яті.

Для ОС IOS [5]:

- процесор серії Apple A9 або новіший;
- мінімум 2 ГБ оперативної пам'яті;
- не менше ніж 16 ГБ вбудованої пам'яті.

1.3.5 Вимоги до інформаційної та програмної сумісності

Розроблені вихідні коди повинні бути написанні мовою JavaScript. Середовище розробки мобільного застосунку Microsoft Visual Studio 2020.

1.3.6 Вимоги до програмної документації

Програмна документація повинна включати: технічне завдання, керівництво програміста, керівництво користувача, опис програми та опис основних програмних модулів.

1.3.7 Порядок приймання та тестування

Для тестування та налагодження застосунку необхідно використовувати можливості Visual Studio Code для виявлення та усунення помилок в процесі розробки. Тестування повинно проводитися у вигляді виконання усіх можливих сценаріїв роботи користувача та мобільного застосунку.

1.4 Висновки за розділом

У цьому розділі було розглянуто застосунки для тайм-менеджменту та планування. Проаналізовано недоліки та переваги найпопулярніших застосунків. Були наведені основні поняття та визначення завдання розв'язуваної проблеми. Було розроблено технічне завдання де прописано призначення застосунку та сферу його використання, визначено вимоги до програми, користувача, технічних засобів, ОС та мови програмування.

2 РОЗРОБКА АРХІТЕКТУРИ ПРОГРАМИ

2.1 Вибір середовища розробки

Щоб визначити найбільш відповідне середовище для реалізації проєкту, необхідно проаналізувати наявні популярні середовища: Microsoft Visual Studio, Replit, Android Studio, Microsoft Visual Studio Code. Кожне з цих середовищ має як переваги так і недоліки, які розглядатимуться в аналізі.

Переваги Microsoft Visual Studio:

- повноцінне IDE з вбудованими інструментами для .NET, C++, C#, SQL Server;
- професійний інструмент для створення десктопних застосунків і серверної частини;
- має інтегроване середовище для дебагу, тестування, роботи з БД;
- потужна підтримка Windows Forms, WPF, UWP.

Недоліки:

- важке середовище, потребує значних системних ресурсів;
- погано підходить для мобільної кросплатформної розробки;
- потребує складнішої конфігурації для JavaScript/Node.js проєктів;
- основна орієнтація на Windows, обмежена підтримка інших платформ [6].

Переваги Replit:

- працює прямо в браузері. Не потрібно встановлювати нічого на комп'ютер – вся розробка відбувається онлайн;
- підтримка JavaScript, React, Node.js;
- інтеграція з GitHub;
- підтримка мобільної розробки через Expo;
- можливість роботи з декількох пристроїв;
- безкоштовна версія. Базові функції доступні безкоштовно, чого достатньо для більшості невеликих проєктів і навчання.

Недоліки:

- потреба в постійному інтернет-з'єднанні. Без мережі Replit просто не працює – не можна писати код офлайн;
- обмежена підтримка React Native. Працює лише через Expo, повноцінної підтримки для нативних функцій або складних інтеграцій нема. Не підходить для великих мобільних проєктів або кастомних компонентів;
- обмежені ресурси в безкоштовній версії. Обмеження за часом виконання, обсягом пам'яті, кількістю запитів;
- низька гнучкість конфігурації. Неможливо повністю контролювати середовище, залежності та пакунки, як у локальній розробці;
- слабка підтримка розширень та кастомних плагінів;
- не можна додати сторонні розширення або налаштувати середовище під себе [7].

Переваги Android Studio:

- офіційне IDE від Google для розробки Android-додатків;
- має візуальний редактор інтерфейсу, емулятор Android;
- глибока інтеграція з Gradle та Firebase;
- підтримує Kotlin і Java;
- розвинена система дебагу та тестування.

Недоліки:

- важке IDE, довго запускається, споживає багато ресурсів;
- не призначене для iOS або кросплатформеної розробки;
- може бути складним для початківців [8].

Переваги Microsoft Visual Studio Code:

- легкий, швидкий і кросплатформений;
- підтримує JavaScript, React, React Native, TypeScript з коробки;
- має потужну систему розширень;
- вбудована інтеграція з Git;
- безкоштовний, активно підтримується спільнотою та Microsoft.

Недоліки:

- це текстовий редактор, а не повноцінне IDE — багато функцій таких як дебаг, тестування та інші потребують встановлення розширень;
- обмежений функціонал без конфігурації плагінів;
- не найкращий варіант для розробки на C++/C#/Java [9].

Щоб зробити остаточний вибір між наведеними середовищами розробки були проаналізовані їх недоліки та переваги у вигляді табл. 2.1.

Таблиця 2.1 – Порівняння наведених середовищ для розробки

Критерій	Microsoft Visual Studio	Replit	Android Studio	Microsoft Visual Studio Code
Підтримка JavaScript	Обмежена	Повна	Часткова	Повна
Підтримка Ехро	Відсутня	Неповна, через шаблони	Відсутня	Повна, через плагіни
Підтримка React Native	Часткова	Часткова	Часткова	Повна
Підтримка Git/GitHub	Так	Так	Так	Так
Навантаження на систему	Високе	Низьке	Високе	Низьке
Інтерфейс	Складний	Простий	Складний	Простий
Розширюваність	Висока	Обмежена	Середня	Дуже висока
Вартість	Безкоштовна Community версія	Безкоштовна з платними функціями	Безкоштовна	Безкоштовна

З огляду на особливості проєкту, найдоцільнішим варіантом є використання Visual Studio Code. Visual Studio Code повністю підтримує усі інструменти, які потрібні при створенні застосунку для мобільних телефонів. Також великий вибір плагінів у середовищі дозволяє гнучко налаштувати його під всі свої потреби. Ще треба відмітити те, що Visual Studio Code має низькі вимоги до ресурсів ПК. У результаті ми маємо середовище, яке дуже добре підійде для розробки мобільного застосунку.

2.2 Вибір мови програмування

Одним з ключових етапів підготовки до розробки мобільного застосунку є вибір мови програмування. Для реалізації дипломного проєкту можна виділити такі мови, як: JavaScript, Kotlin та Python. Кожна з наведених мов має як недоліки, так і переваги, які будуть наведені нижче.

Переваги JavaScript:

- кросплатформеність. JavaScript дозволяє створювати мобільні застосунки для Android та iOS з єдиною кодовою базою, використовуючи фреймворки на кшталт React Native у поєднанні з набором інструментів Expo;
- велика спільнота і кількість бібліотек. JavaScript одна з найпоширеніших мов у світі, що має тисячі відкритих пакетів, готових компонентів і рішень;
- сучасні фреймворки. Забезпечують зручний компонентний підхід, швидке оновлення інтерфейсу (Hot Reload), зручну навігацію та інтеграцію зі сторонніми бібліотеками;
- легкий вхід для новачків. JavaScript має простий синтаксис, а поріг входження нижчий, ніж у мов, які потребують компіляції;
- активний розвиток інструментів. Постійні оновлення фреймворків, інструментів і середовищ розробки.

Недоліки:

- менша продуктивність у порівнянні з нативними мовами. Через наявність JavaScript bridge, який виступає мостом між JavaScript та нативним кодом. Через це складні або ресурсоємні застосунки можуть працювати повільніше;
- залежність від сторонніх бібліотек. Часто доводиться покладатися на сторонні пакети, деякі з яких погано підтримуються або мають конфлікти з новими версіями;
- нестабільність сумісності при оновленнях. Оновлення фреймворків чи пакетів іноді спричиняють конфлікти або потребу в переробці значної частини коду [10].

Переваги Kotlin:

- підтримується Google як основна мова для Android. Kotlin — офіційна мова розробки Android-застосунків з 2019 року, що гарантує глибоку інтеграцію з Android SDK та підтримку Android Studio;
- висока продуктивність. Код компілюється у нативний, що забезпечує нативну швидкодію;
- гнучка інтеграція з існуючим кодом. Kotlin можна комбінувати з іншими мовами;
- потужна підтримка в Android Studio. Вбудований дебаг, автодоповнення, інспекції коду, навігація — усе повністю інтегроване.

Недоліки:

- обмежена кросплатформеність у частині інтерфейсу. Kotlin дозволяє ділитися лише бізнес-логікою, а не UI;
- вища вимога до знань. Kotlin — складніша за JavaScript або Python, особливо для новачків, через більш строгую типізацію, асинхронність, DSL тощо;
- менша кількість готових рішень для iOS. На платформі iOS екосистема Kotlin Multiplatform поки що менш зріла: інтеграція складніша, а кількість стабільних рішень і документації — обмежена в порівнянні з Android;

- порівняно невелика спільнота для Kotlin Multiplatform. Хоча спільнота Kotlin загалом є досить активною, кросплатформений напрямок наразі характеризується меншою кількістю прикладів використання, готових шаблонів та доступної документації порівняно з альтернативними рішеннями [11].

Переваги Python:

- простота синтаксису. Python має легкий для читання й розуміння синтаксис, що знижує поріг входження для новачків та пришвидшує написання логіки застосунку;
- кросплатформність. У Python є фреймворки такі як Kivy та Beeware які дозволяють створювати застосунки, що працюють на Android та iOS;
- відкрите та безкоштовне середовище. Усі інструменти розробки, фреймворки та бібліотеки Python, включно з Kivy і BeeWare, мають відкритий код і не потребують ліцензій;
- можливість повторного використання логіки. У випадках, коли інтерфейс розробляється окремо, Python можна використовувати як бекенд-логіку, яку потім підключають до нативних або вебзастосунків.

Недоліки:

- обмежена підтримка мобільної розробки. Python не має фреймворку, що надає повноцінну підтримку мобільного UI з компонентами нативного вигляду;
- нестабільність роботи на iOS. Розгортання застосунків на iOS значно складніше і нестабільніше;
- відсутність широкої підтримки нативних можливостей. Такі функції, як push-сповіщення, біометрія, камера або геолокація, реалізуються складно або взагалі не підтримуються у більшості Python-фреймворків;
- менша продуктивність. Python виконується повільніше за нативні мови, що може впливати на плавність роботи застосунку, особливо на слабших мобільних пристроях;

— мала спільнота у сфері мобільної розробки. Незважаючи на популярність Python загалом, у галузі мобільної розробки кількість прикладів, документації та готових рішень — істотно нижча [12].

Після аналізу мов програмування їхні переваги та недоліки були узагальнені у табл. 2.2 для обґрунтування остаточного вибору.

Таблиця 2.2 – Порівняння мов програмування

Критерій	JavaScript	Kotlin	Python
Популярність використання	Дуже висока	Висока для Android але середня для IOS	Висока загалом, але низька у мобільній розробці
Наявність фреймворків для кросплатформеної розробки	React Native	Kotlin Multiplatform Mobile	Kivy, BeeWare
Простота вивчення	Так	Ні	Так
Швидкодія	Середня	Висока	Низька
Підтримка нативного функціоналу	Повна	Повна	Низька

У підсумку мова програмування JavaScript була обрана як найбільш оптимальна для реалізації проєкту, оскільки поєднує в собі низку важливих переваг таких як, широку підтримку, наявність популярного фреймворку для мобільної розробки, активну спільноту та відносну простоту у вивченні.

2.3 Проектування архітектури програмного забезпечення

Для розробки мобільного застосунку для тайм-менджменту та планування було обрано фреймворк React Native [13]. Цей фреймворк має наступні складові:

- react. Це основа React Native – відповідає за створення інтерфейсу за допомогою компонентів [14];
- react Native-компоненти. Це мобільні аналоги HTML-елементів – JSX, які працюють не в браузері, а в нативному середовищі. Вони обгортаються у платформозалежний код і перетворюються в елементи інтерфейсу Android чи iOS;
- javascript Bridge. Це "міст", через який React Native зв'язує JavaScript-код із нативними API Android і iOS;
- модулі для доступу до нативних API. React Native має набір модулів для доступу до камери, геолокації, роботи з bluetooth, роботи з файлами, сповіщеннями, датчиками тощо;
- metro bundler. Це білдер, який запускає проєкт React Native. Він пакує увесь код, JSX, стилі, зображення тощо у зрозумілий для телефону формат;
- hot reload. Миттєве оновлення інтерфейсу без перезапуску застосунку.

Переваги React Native:

- кросплатформність. Один і той самий код працює як на Android, так і на iOS, що значно економить час і ресурси при розробці;
- швидка розробка і оновлення. Завдяки функціям Hot Reload та Fast Refresh зміни в коді можна бачити миттєво, без повного перезапуску застосунку;
- мобільні компоненти з React-синтаксисом. Компоненти React Native JSX – схожі на HTML за синтаксисом, тому з ними зручно працювати навіть для початківців. Вони адаптовані до мобільних інтерфейсів;

- доступ до нативних API. Можна використовувати функції пристрою камеру, геолокацію, сповіщення – через готові або сторонні модулі;
- велика спільнота та екосистема. Багато готових бібліотек, прикладів, рішень типових проблем. Це пришвидшує розробку і допомагає уникати помилок;
- підтримка Ехро. Платформа Ехро ще більше спрощує запуск, тестування та розгортання застосунку без необхідності працювати з Android Studio.

Також у розробці мобільного застосунку важливим буде інструмент під назвою Ехро. Ехро – це платформа та набір інструментів, які використовуються для того, щоб зробити створення мобільних застосунків на React Native простішим і швидшим [15]. Ехро містить у собі наступні ключові складові:

- ехро SDK. Набір готових бібліотек і модулів, які дозволяють швидко додавати функціональність, такі як: камера, сповіщення, доступ до геолокації, робота з файлами, біометрія, акселерометр та інші сенсори;
- ехро Go. Мобільний застосунок, який дозволяє запустити свій проєкт на телефоні без збірки APK;
- ехро CLI. інструмент командного рядка, який дозволяє створювати нові проєкти, запускати їх в браузері чи телефоні, будувати фінальні збірки, оновлювати залежності;
- ехро Application Services. Хмарні сервіси, які дозволяють створювати продакшн-збірки для Play Market та App Store;
- ехро Snack. Онлайн-середовище для експериментів із React Native та Ехро прямо в браузері. Дозволяє швидко писати, тестувати та ділитися кодом.

Переваги Ехро:

- швидкий старт без складного налаштування. Ехро дозволяє створити мобільний застосунок за кілька хвилин. Не потрібно встановлювати Android Studio чи Xcode, що особливо зручно для початківців;

- готові модулі "з коробки". У Expo є вбудовані функції для роботи з камерою, геолокацією, сповіщеннями, файловою системою тощо. Не потрібно писати нативний код або вручну підключати бібліотеки;
- expo go. Замість того щоб збирати застосунок, достатньо відсканувати QR-код, і програма відкриється на реальному пристрої через застосунок Expo Go;
- підтримка кросплатформеності. Код, написаний з використанням Expo, працює на Android та iOS без змін;
- інтеграція з Expo Application Services. Можна легко збирати фінальні збірки у хмарі, без складного локального налаштування;
- підтримка Hot Reload. Зміни в коді автоматично застосовуються у застосунку без перезавантаження. Це прискорює розробку і тестування.

Застосунок складатиметься з чотирьох модулів, які реалізують окремі функції згідно з ТЗ. Загалом архітектура передбачає наявність чотирьох функціональних модулів: користувацький інтерфейс (UI), централізоване управління станом (Context API), локалізація, локальне сховище. На рис. 2.1 представлено модулі мобільного застосунку та їх функції.



Рисунок 2.1 – Модулі та їх функції

Для цього проєкту обрана компонентно-орієнтована архітектура з централізованим управлінням станом. У цієї архітектури є наступні переваги:

- розуміла структура проєкту. Кожен компонент відповідає за окрему частину інтерфейсу або логіки, що робить код упорядкованим;
- просте масштабування. Легко додавати нові екрани та компоненти;
- логіка зосереджена в одному місці;
- зменшення дублювання логіки. Функції додавання, видалення чи фільтрації завдань зберігаються в одному місці, а не копіюються в кожен екран або компонент;
- краща продуктивність при правильній оптимізації. Якщо компоненти підключаються лише до тих частин стану, які їм потрібні, застосунок працюватиме швидше і реагуватиме без зайвих оновлень;
- не потребує додаткових бібліотек.

Такий підхід до створення архітектури застосунку буде більш простим та практичним через те, що він легкий у реалізації, зрозумілий та не обтяжений зайвою структурою.

При проєктуванні мобільного застосунку повинна бути реалізована наступна структура:

- інтерфейс;
- логіка;
- локалізація;
- зберігання.

Кожен елемент архітектури має виконувати чітко визначену функцію. Важливо, щоб між окремими частинами системи не виникало надмірної залежності, оскільки це ускладнює модифікацію та підтримку програмного продукту. Такий підхід сприяє підтримці чистої, гнучкої та масштабованої структури. У разі потреби внесення змін до окремого функціонального модуля це не впливатиме на роботу всієї системи, що значно полегшує подальший розвиток і адаптацію застосунку до нових вимог. Для узагальнення основної

логіки функціонування мобільного застосунку було розроблено Use-case діаграму, представлену на рис. 2.2.

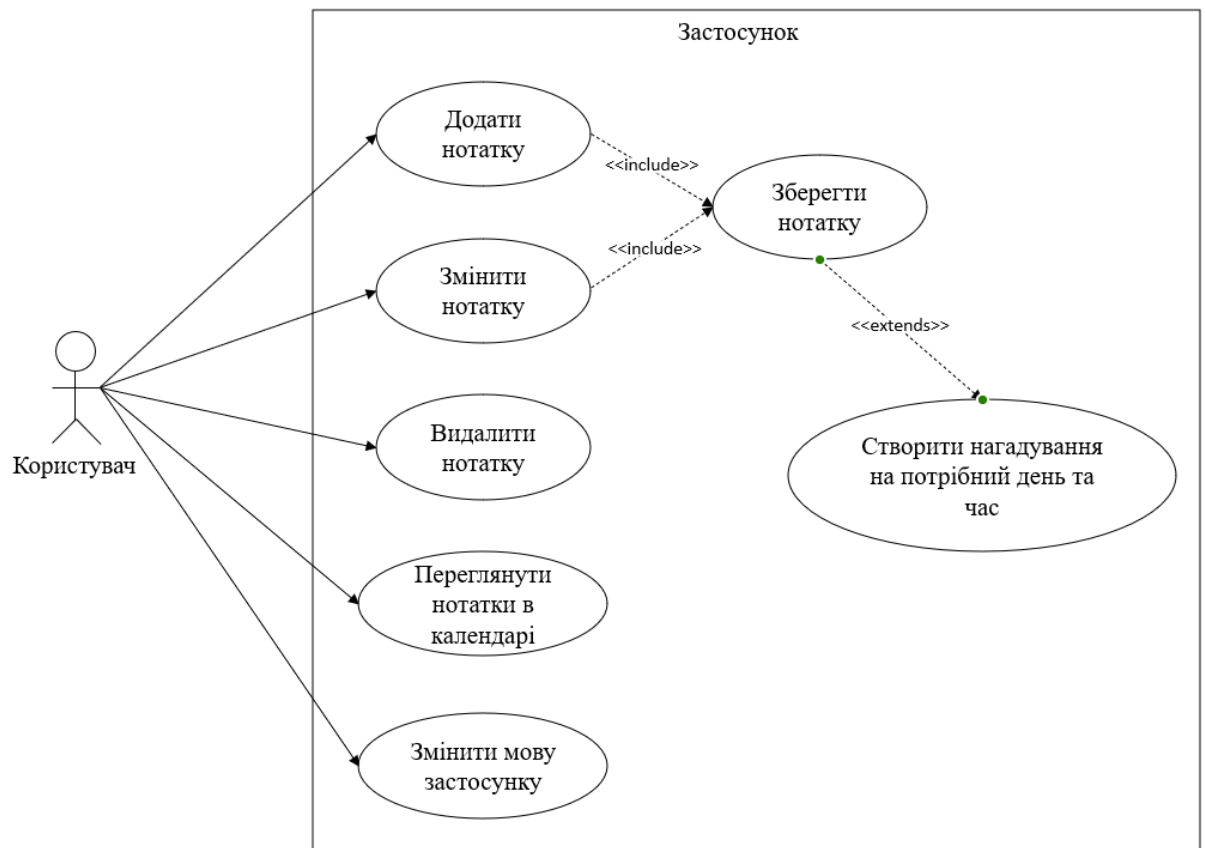


Рисунок 2.2 – Use-case діаграма

У межах обраної архітектури, компоненти інтерфейсу не зберігають власного стану завдань, а отримують усі необхідні дані та функції через Context API.

Context API – це вбудований механізм у React, який дозволяє передавати дані глибоко по дереву компонентів без потреби передавати налаштування вручну на кожному рівні. Завдяки цьому компоненти лише відображають інформацію або передають події. Такий підхід дозволяє зберігати чисту структуру застосунку, уникати дублювання логіки та спрощує оновлення даних у різних частинах інтерфейсу.

2.4 Висновки за розділом

У цьому розділі було обґрунтовано вибір фреймворку React Native для розробки мобільного застосунку. Також запропоновано доповнити цей фреймворк платформою під назвою Ехро, яка допоможе у створенні зручної та гнучкої архітектури застосунку. За допомогою цієї платформи можна відкривати проєкт застосунку на мобільному пристрої і тестувати його в режимі реального часу, без повної збірки застосунку.

Усе це буде працювати за моделлю компонентно-орієнтовної архітектури з централізованим управлінням станом, яка дозволить зробити зрозумілу структуру проєкту, зосередить логіку в одному місці та зменшить її дублювання. Ця модель дозволяє розширювати функціонал застосунку у майбутньому, без проблем із залежностями.

3 ОСНОВНІ РІШЕННЯ З РЕАЛІЗАЦІЇ КОМПОНЕНТІВ СИСТЕМИ

3.1 Алгоритм функціонування програмного забезпечення

Після запуску мобільного застосунку відбувається початкова ініціалізація його функціональних модулів. У першу чергу, застосунок підключає контекст управління завданнями `TasksContext`, який обгортає основну навігаційну структуру та забезпечує централізоване управління даними.

У межах ініціалізації викликається функція `loadTasks`, яка виконує асинхронне зчитування з локального сховища `AsyncStorage`. У результаті цього завантажуються раніше збережені об'єкти нотаток та встановлюються як початковий стан змінної `tasks`, що використовується всіма екранами. Код виклику функції `loadTasks` наведено у рис. 3.1.

```
const loadTasks = async () => {  
  try {  
    const stored = await AsyncStorage.getItem('tasks');  
    if (stored) setTasks(JSON.parse(stored));  
  } catch (e) {  
    console.error('Не вдалося завантажити завдання:', e);  
  }  
};
```

Рисунок 3.1 – Код виклику функції `loadTasks`

Крім того, на цьому етапі може бути зчитане поточне мовне налаштування та інші глобальні параметри, необхідні для правильної роботи інтерфейсу. Таким чином, ініціалізація забезпечує готовність застосунку до повноцінної роботи з персоніфікованими даними користувача одразу після запуску.

Після завершення процесу ініціалізації та підключення контексту управління завданнями застосунок автоматично переходить до головного екрана — `TaskListScreen`, який є початковим маршрутом у конфігурації

навігації AppNavigator. Цей екран відповідає за виведення основного списку нотаток користувача та забезпечує базові дії для взаємодії з ними.

Завдання для відображення отримуються із глобального контексту TasksContext за допомогою хука useContext. Компонент FlatList використовується для ефективного рендерингу списку елементів з можливістю вертикального прокручування. Код компоненту FlatList наведено у рис. 3.2.

```
<FlatList
  data={filteredTasks}
  keyExtractor={item => item.id.toString()}
  renderItem={renderTask}
```

Рисунок 3.2 – Код компоненту FlatList

Кожен елемент містить інформацію про назву нотатки, опис, категорію, дату нагадування, а також інструменти для взаємодії: кнопки видалення, редагування та позначення завдання як виконаного.

Модуль реалізує функцію фільтрації за категоріями, яка керується змінною selectedCategory в локальному стані. Значення фільтра обирається за допомогою компонента Picker, після чого список нотаток автоматично оновлюється без потреби перезавантаження. Також реалізована візуальна індикація виконаних завдань, які відображаються зі змінним стилем.

Видалення нотатки реалізовано через виклик функції confirmDelete, що дозволяє уникнути випадкової втрати даних. Ця функція викликає вікно Alert у якому потрібно підтвердити дію по видаленню нотатки або скасувати дію. Код виклику функції confirmDelete наведено у рис. 3.3.

```
const confirmDelete = (taskId) => {
  Alert.alert(
    t('confirmDeleteTitle'),
    t('confirmDeleteMessage'),
    [
      { text: t('cancel'), style: 'cancel' },
      { text: t('delete'), onPress: () => removeTask(taskId), style: 'destructive' }
    ]
  );
};
```

Рисунок 3.3 – Код виклику функції confirmDelete

Редагування нотатки ініціюється через навігацію на екран EditNoteScreen, куди передаються всі дані обраного запису як параметри маршруту.

Таким чином, TaskListScreen є функціональним центром взаємодії користувача із завданнями, забезпечуючи зручну навігацію, керування станом, доступ до основних операцій і інтеграцію з контекстом зберігання даних.

Наступним є модуль AddNoteScreen, який відповідає за створення нової нотатки шляхом заповнення форми користувачем. Перехід до цього екрана здійснюється через Drawer-навігацію або через спеціальну кнопку на головному екрані. Всі поля форми представлені як елементи користувацького інтерфейсу: текстові поля для заголовка та опису, випадаючий список для вибору категорії, а також інструмент вибору дати й часу для вказання дати та нагадування.

Після введення необхідних даних користувач натискає кнопку збереження нотатки, що ініціює виклик функції addTask, яка імпортується з контексту TasksContext. Код виклику функції addTask наведено у рис. 3.4.

```
const addTask = taskData => {
  const newTask = {
    id: Date.now(),
    ...taskData,
    completed: false
  };
  const updated = [newTask, ...tasks];
  setTasks(updated);
  saveTasks(updated);
};
```

Рисунок 3.4 – Код виклику функції addTask

Новостворений об'єкт додається до масиву `tasks` у контексті, після чого відбувається його серіалізація та збереження у `AsyncStorage` під ключем `tasks`. У разі успішного збереження виконується повернення на головний екран `TaskListScreen`, де нова нотатка автоматично відображається у списку.

Таким чином, модуль `AddNoteScreen` реалізує повноцінний процес додавання даних до внутрішнього сховища застосунку з використанням динамічної архітектури, що забезпечує миттєве оновлення інтерфейсу без потреби ручного оновлення екрана.

За редагування нотаток відповідає модуль `EditNoteScreen`. Редагування раніше створеної нотатки здійснюється через екран `EditNoteScreen`, який викликається з головного екрану `TaskListScreen` після натискання відповідної кнопки. Під час переходу до модуля редагування передаються всі параметри обраної нотатки у вигляді навігаційних параметрів, які автоматично заповнюють відповідні поля форми.

Інтерфейс `EditNoteScreen` аналогічний модулю створення нотатки `AddNoteScreen`, однак у даному випадку всі поля — такі як заголовок, опис, дата, нагадування і категорія — вже мають попередньо встановлені значення, що відповідають даним редагованого запису. Користувач може змінити будь-який з параметрів і зберегти оновлення.

При збереженні змін викликається функція `updateTask`, визначена у контексті `TasksContext`. Вона здійснює пошук відповідного об'єкта нотатки за унікальним ідентифікатором `id` і замінює його на оновлений варіант у масиві `tasks`. Після оновлення масиву новий стан серіалізується і зберігається у `AsyncStorage`, що забезпечує збереження змін між сесіями. Також функція `updateTask` використовується для позначення завдання як виконаного на екрані `TaskListScreen` при натисканні на відповідну кнопку. Код виклику функції `updateTask` наведено у рис. 3.5.

```
const updateTask = updatedTask => {
  const updated = tasks.map(t =>
    t.id === updatedTask.id ? updatedTask : t
  );
  setTasks(updated);
  saveTasks(updated);
};
```

Рисунок 3.5 – Код виклику функції updateTask

Таким чином, EditNoteScreen реалізує повноцінний цикл оновлення даних нотатки, забезпечуючи користувачу можливість коригувати вміст без створення нових записів, з миттєвим оновленням інтерфейсу після збереження.

Окрім традиційної взаємодії зі списком завдань, користувач має змогу переглядати нотатки за календарною логікою. Для цього в застосунку передбачено окремий модуль — CalendarScreen, який реалізує інтерфейс на основі календаря. Цей екран викликається через Drawer-навігацію та дозволяє швидко переглядати завдання, заплановані на певні дні.

При завантаженні CalendarScreen ініціалізується компонент бібліотеки react-native-calendars, який відображає інтерактивну сітку дат. Після натискання на певну дату користувачеві виводиться список усіх нотаток, у яких значення поля date збігається з обраним днем. Таким чином реалізується функціонал фільтрації нотаток за датою без потреби ручного пошуку або прокручування списку.

Фільтрація виконується на рівні компонента, де список нотаток tasks береться з контексту TasksContext, і за допомогою функції filter відбираються лише ті записи, які відповідають вибраній даті. Результат миттєво відображається на екрані, зберігаючи динамічність інтерфейсу.

Завдяки цьому модулю користувач отримує зручний візуальний інструмент планування, який значно підвищує функціональність застосунку в контексті щоденного тайм-менеджменту.

На додаток до функціональних можливостей керування нотатками та календарного планування, мобільний застосунок реалізує базову персоналізацію інтерфейсу. Цей функціонал представлений у модулі `SettingsScreen`, який дозволяє користувачеві змінювати мову інтерфейсу. Зміна мови доступна через окремий екран, до якого можна перейти за допомогою бічної навігації.

На момент розробки застосунків підтримує дві мови: українську та англійську. Перемикання між ними здійснюється за допомогою елемента `Picker`, що дозволяє обрати бажану локалізацію. Реалізація мовної адаптації базується на бібліотеці `react-i18next`, яка забезпечує автоматичну зміну мовних ресурсів у всіх компонентах інтерфейсу без потреби перезавантаження застосунку. Це досягається за рахунок динамічного оновлення стану локалізації, який доступний у межах усього дерева компонентів. Код модуля локалізації `i18n.js` наведено у рис. 3.6.

```
import i18n from 'i18next';
import { initReactI18next } from 'react-i18next';
import en from './locales/en.json';
import uk from './locales/uk.json';
import 'intl-pluralrules';
i18n.use(initReactI18next).init({
  resources: {
    en: { translation: en },
    uk: { translation: uk },
  },
  lng: 'en',
  fallbackLng: 'en',
  interpolation: {
    escapeValue: false,
  },
});
export default i18n;
```

Рисунок 3.6 – Код модуля локалізації `i18n.js`

Вибрана мова застосовується не лише до статичних елементів інтерфейсу, а й до динамічного вмісту, такого як категорії нотаток, заголовки, повідомлення про відсутність завдань тощо. Завдяки цьому забезпечується

повноцінна підтримка мультимовності, що підвищує доступність програмного забезпечення для ширшого кола користувачів.

Усі описані модулі застосунку функціонують у тісному зв'язку з внутрішнім сховищем даних, що забезпечує збереження інформації між сесіями. У межах цього програмного забезпечення база даних реалізована у вигляді локального сховища `AsyncStorage`, яке є частиною інфраструктури `React Native`. Цей механізм забезпечує просте, але ефективне збереження структурованих даних у форматі `JSON` у пам'яті пристрою.

Для виконання `CRUD` операцій використовуються наступні функції:

- `addTask` для створення нотаток;
- `loadTasks` для зчитування нотаток;
- `updateTask` для оновлення нотаток;
- `removeTask` для видалення нотаток.

Усі зміни в масиві `tasks` одразу синхронізуються з `AsyncStorage`, що гарантує цілісність і сталість даних після перезапуску застосунку або закриття сесії.

Операції взаємодії з базою даних є повністю асинхронними та не блокують головний потік виконання, що дозволяє підтримувати плавну роботу інтерфейсу навіть під час зчитування великих обсягів інформації. Такий підхід дозволяє реалізувати легку, незалежну від зовнішнього серверу, архітектуру збереження, яка є особливо ефективною для мобільних офлайн-застосунків.

3.2 Розробка структури бази даних

У рамках проєктування мобільного застосунку для тайм-менеджменту та планування передбачено використання `AsyncStorage` як основного засобу локального зберігання даних. `AsyncStorage` — це модуль керування локальним сховищем даних, що входить до екосистеми `React Native`. Він забезпечує асинхронне, постійне збереження даних у форматі пар ключ–значення, що дозволяє зберігати та зчитувати інформацію між сеансами використання

застосунку. На технічному рівні AsyncStorage виступає обгорткою над нативними засобами зберігання, і слугує простим API для роботи з локальними даними без необхідності підключення до серверу [16].

Після завантаження застосунку виконується функція loadTasks, яка реалізована у контексті TasksContext та викликається автоматично завдяки хуку useEffect. Хук useEffect наведено у рис. 3.7.

```
useEffect(() => {  
  loadTasks();  
}, []);
```

Рисунок 3.7 – Хук useEffect

Функція loadTasks перевіряє наявність даних у локальному сховищі за ключем 'tasks' через метод AsyncStorage.getItem. Якщо дані за цим ключем вже існують, вони завантажуються в оперативну змінну стану tasks, що забезпечує синхронізацію з інтерфейсом користувача. Якщо ж ключ 'tasks' ще не створено, то застосунок не виконує жодної ініціалізації вручну — замість цього база створюється автоматично під час першої взаємодії з нею.

Створення ключа 'tasks' і відповідного масиву завдань у AsyncStorage відбувається під час першого виклику функції saveTasks, яка, у свою чергу, викликається при додаванні, редагуванні або видаленні завдання. В цій функції використовується метод AsyncStorage.setItem, який зберігає масив завдань у форматі JSON за відповідним ключем. Таким чином, створення структури бази даних у цьому застосунку не потребує попереднього оголошення чи ініціалізації — вона формується динамічно при першій зміні даних. Функція saveTasks наведена у рис. 3.8.

```
// Збереження завдань у AsyncStorage
const saveTasks = async updated => {
  try {
    await AsyncStorage.setItem('tasks', JSON.stringify(updated));
  } catch (e) {
    console.error('Не вдалося зберегти завдання:', e);
  }
};
```

Рисунок 3.8 – Функція saveTasks

Зберігання реалізується за допомогою addTask у вигляді значень типу рядка, прив'язаних до унікальних ключів. Застосунок може в будь-який момент зчитувати, оновлювати або видаляти дані за допомогою асинхронних методів, що не блокують головний потік виконання. Таким чином, AsyncStorage дозволяє створювати легкі, швидкодіючі застосунки з базовими потребами в локальному зберіганні, такими як збереження налаштувань, сесій, нотаток або нагадувань.

У проєкті база даних представлена модулем масиву об'єктів нотаток, який зберігає інформацію, введену користувачем під час створення нотатки. Детальний опис наведено у табл. 3.1.

Таблиця 3.1 – Модуль масиву об'єктів нотаток

Поле	Тип даних	Опис поля
id	number	Унікальний ідентифікатор який генерується автоматично для кожної нотатки
title	string	Заголовок нотатки, який вводиться користувачем
description	string	Опис змісту нотатки
date	string	Дата і час нагадування

Кінець таблиці 3.1.

Поле	Тип даних	Опис поля
reminder	boolean	Нагадування
tags	string	Категорія

Об'єкт `taskData` формується на основі значень, які користувач вводить у поля форми на екрані `AddNoteScreen`. Кожне поле форми пов'язане зі змінною стану, що зберігається за допомогою хука `useState`. Після підтвердження введення, всі ці значення збираються в один об'єкт `taskData`, який передається у функцію `addTask` для подальшого збереження в локальній базі даних. Код форм з екрану `AddNoteScreen` наведено у рис. 3.9.

```
// Заголовок нотатки
const [title, setTitle] = useState('');

// Опис нотатки
const [description, setDescription] = useState('');

// Дата виконання нотатки (за замовчуванням поточна дата)
const [date, setDate] = useState(new Date());

// Час виконання нотатки (за замовчуванням поточний час)
const [time, setTime] = useState(new Date());

// Відображення вибору дати (показати/сховати DatePicker)
const [showDatePicker, setShowDatePicker] = useState(false);

// Відображення вибору часу (показати/сховати TimePicker)
const [showTimePicker, setShowTimePicker] = useState(false);

// Категорія нотатки (за замовчуванням "other")
const [category, setCategory] = useState('other');

// Унікальний ідентифікатор нотатки (null для нової нотатки)
const [taskId, setTaskId] = useState(null);

// Перемикач для ввімкнення/вимкнення сповіщень
const [enableNotification, setEnableNotification] = useState(false);
```

Рисунок 3.9 – Код форм з `AddNoteScreen`

3.3 Розробка модулів програми

3.3.1 Головний модуль застосунку App.js

Модуль App.js виконує роль центральної точки входу у мобільний застосунок. Його основне призначення — об'єднати ключові логічні та інтерфейсні компоненти, забезпечити початкову ініціалізацію середовища, активувати підтримку багатомовності, а також налаштувати глобальний контекст управління завданнями та маршрутизацію між екранами.

На початку створюються два навігаційні контейнери, які відповідають за різні рівні структури інтерфейсу. Drawer використовується для реалізації бокового меню навігації Drawer Navigation, що дозволяє перемикатися між основними розділами застосунку: списком завдань, календарем та налаштуваннями. Stack використовується для побудови вкладеної навігації між екранами у межах одного розділу — зокрема, між списком завдань, додаванням та редагуванням нотаток. Виклик навігаційних контейнерів наведено у рис. 3.10.

```
const Drawer = createDrawerNavigator();  
const Stack = createStackNavigator();
```

Рисунок 3.10 – Виклик навігаційних контейнерів

У складі модуля App.js функція MainNavigator() реалізує стек-навігацію, яка використовується для організації переходів між основними робочими екранами застосунку. Вона побудована на основі компонента Stack.Navigator з бібліотеки [@react-navigation/stack](https://reactnavigation.org/docs/stack-navigator).

У межах цього навігатора визначено три маршрути:

- taskList — головний екран зі списком завдань (TaskListScreen);
- addNote — екран створення нової нотатки (AddNoteScreen);
- editNote — екран редагування наявної нотатки (EditNoteScreen).

Для кожного маршруту задано параметри через options, які включають локалізовані назви екранів. Переклад здійснюється за допомогою функції t() з бібліотеки react-i18next, що забезпечує багатомовність інтерфейсу. Зокрема, назви вкладених екранів автоматично змінюються відповідно до обраної мови застосунку. Реалізація навігації між TaskList, AddNote та EditNote наведена на рис. 3.11.

```
function MainNavigator() {
  const { t } = useTranslation(); // Використання перекладу

  return (
    <Stack.Navigator>
      <Stack.Screen
        name="TaskList"
        component={TaskListScreen}
        options={{ title: t('taskList'), headerShown: false }} // Приховування заголовка
      />
      <Stack.Screen
        name="AddNote"
        component={AddNoteScreen}
        options={{ title: t('addNote') }}
      />
      <Stack.Screen
        name="EditNote"
        component={EditNoteScreen}
        options={{ title: t('editNote') }}
      />
    </Stack.Navigator>
  );
}
```

Рисунок 3.11 – Реалізація навігації між TaskList, AddNote та EditNote

За допомогою хука useTranslation() компонент отримує доступ до функції t() з бібліотеки react-i18next, що дає змогу локалізувати назви маршрутів та інші текстові елементи. Таким чином, уся навігація застосунку автоматично адаптується до вибраної мови інтерфейсу.

У блоці useEffect() виконується перевірка дозволів на надсилання локальних сповіщень через expo-notifications. Якщо відповідний дозвіл ще не було надано, застосунок ініціює запит на його отримання. Це забезпечує можливість подальшого використання функціоналу нагадувань для завдань.

Навігаційна структура застосунку, представлена компонентом `NavigationContainer`, розміщується всередині провайдера контексту `TasksProvider`. Завдяки цьому всі екрани, що знаходяться в межах навігації, автоматично отримують доступ до загального стану застосунку та функцій для роботи із завданнями. Реалізація ініціалізації контексту, навігації та запит дозволу на сповіщення наведена на рис. 3.12.

```
export default function App() {
  const { t } = useTranslation(); // Використання перекладу

  useEffect(() => {
    const requestPermissions = async () => {
      const { status } = await Notifications.getPermissionsAsync();
      if (status !== 'granted') {
        await Notifications.requestPermissionsAsync();
      }
    };

    requestPermissions();
  }, []);

  return (
    <TasksProvider>
      <NavigationContainer>
        <Drawer.Navigator initialRouteName="TaskListDrawer">
          <Drawer.Screen
            name="TaskListDrawer"
            component={MainNavigator}
            options={{ title: t('TaskListDrawer') }}
          />
          <Drawer.Screen
            name="Calendar"
            component={CalendarScreen}
            options={{ title: t('Calendar') }}
          />
          <Drawer.Screen
            name="Settings"
            component={SettingsScreen}
            options={{ title: t('settings') }}
          />
        </Drawer.Navigator>
      </NavigationContainer>
    </TasksProvider>
  );
}
```

Рисунок 3.12 – Ініціалізація контексту, навігації та запит дозволу на сповіщення

3.3.2 Контекст управління завданнями TaskContext

Модуль TaskContext реалізує централізовану логіку керування даними завдань у мобільному застосунку. Його основною метою є створення глобального контексту, який дозволяє усім компонентам додатку спільно використовувати та змінювати список завдань без необхідності передавати дані через параметри.

Розробку модуля було здійснено з використанням функціонального підходу, на основі бібліотеки React. Для створення контексту застосовано вбудований інструмент createContext, а для надання доступу до контексту — компонент-провайдер TasksProvider, який огортає всю навігаційну структуру застосунку. На рис 3.13 наведено інструмент createContext.

```
// Створюємо контекст для завдань  
export const TasksContext = createContext();
```

Рисунок 3.13 – Інструмент createContext

У середині провайдера реалізовано локальний стан tasks за допомогою хука useState. Він зберігає масив об'єктів, кожен з яких представляє окреме завдання з полями: id, title, description, date, reminder, category, completed. Хук useState наведено у рис 3.14.

```
// Провайдер контексту  
export const TasksProvider = ({ children }) => {  
  const [tasks, setTasks] = useState([]);
```

Рисунок 3.14 – Хук useContext

Окремим елементом логіки модуля TaskContext є функція loadTasks, відповідальна за початкове завантаження збережених даних про завдання з локального сховища AsyncStorage. Вона реалізована як асинхронна функція,

що дає змогу виконувати операції читання без блокування основного потоку інтерфейсу.

У межах `loadTasks` здійснюється спроба отримати збережене значення за ключем `'tasks'`. Якщо дані присутні, вони розбираються з формату JSON за допомогою `JSON.parse` і передаються у стан `setTasks`, що забезпечує відображення актуального списку завдань у всіх компонентах, які використовують контекст.

У разі виникнення помилки під час зчитування даних, вона фіксується у консоль, що дозволяє діагностувати потенційні проблеми на етапі налагодження застосунку. Реалізація завантаження завдань з локального сховища наведена у рис. 3.15.

```
// Завантаження завдань з AsyncStorage
const loadTasks = async () => {
  try {
    const stored = await AsyncStorage.getItem('tasks');
    if (stored) setTasks(JSON.parse(stored));
  } catch (e) {
    console.error('Не вдалося завантажити завдання:', e);
  }
};
```

Рисунок 3.15 – Завантаження завдань з локального сховища

У `TaskContext` прописані наступні функції для роботи з даними:

- `loadTasks` — асинхронна функція для завантаження масиву завдань із локального сховища `AsyncStorage`;
- `saveTasks` — збереження оновленого масиву завдань у сховищі під ключем `'tasks'`;
- `addTask` — формує новий об'єкт завдання, додає його до масиву та зберігає;
- `updateTask` — оновлює завдання за `id`, замінюючи його новим об'єктом;
- `removeTask` — видаляє завдання з масиву за його ідентифікатором;

– toggleTaskCompleted — перемикає статус виконання конкретного завдання.

На рис. 3.16 наведено функції для роботи з даними.

```
// Збереження завдань у AsyncStorage
const saveTasks = async updated => {
  try {
    await AsyncStorage.setItem('tasks', JSON.stringify(updated));
  } catch (e) {
    console.error('Не вдалося зберегти завдання:', e);
  }
};

// Додає нове завдання (за замовчуванням не виконане)
const addTask = taskData => {
  const newTask = {
    id: Date.now(),
    ...taskData,
    completed: false // поле для позначення виконаних
  };
  const updated = [newTask, ...tasks];
  setTasks(updated);
  saveTasks(updated);
};

// Оновлює наявне завдання (використовується й для markCompleted)
const updateTask = updatedTask => {
  const updated = tasks.map(t =>
    t.id === updatedTask.id ? updatedTask : t
  );
  setTasks(updated);
  saveTasks(updated);
};

// Видаляє завдання
const removeTask = id => {
  const updated = tasks.filter(t => t.id !== id);
  setTasks(updated);
  saveTasks(updated);
};

// Перемикає стан виконання завдання
const toggleTaskCompleted = id => {
  const updated = tasks.map(t =>
    t.id === id ? { ...t, completed: !t.completed } : t
  );
  setTasks(updated);
  saveTasks(updated);
};
```

Рисунок 3.16 – Функції для роботи з даними

Для ініціалізації даних при першому запуску застосунку використовується хук `useEffect`, який викликає `loadTasks()` одразу після монтування компонента. Таким чином, при кожному запуску застосунку користувач отримує актуальний список раніше збережених завдань. У п. 3.2 на рис 3.7 наведено хук `useEffect`.

Щоб при першому запуску застосунку завдання одразу були завантажені з локального сховища, використано хук `useEffect`, який викликає функцію `loadTasks()` одразу після монтування компонента. Використання хука `useEffect` для ініціалізації даних після монтування компонента наведено у рис. 3.17.

```
// Використовуємо useEffect для завантаження завдань при першому рендері компонента
useEffect(() => {
  loadTasks(); // Завантажує завдання з AsyncStorage або іншого джерела
}, []); // Порожній масив залежностей означає, що цей ефект виконається лише один раз
```

Рисунок 3.17 – Використання хука `useEffect` для ініціалізації даних після монтування компонента

У кінці компонента `TasksProvider` контекст `TasksContext.Provider` повертає значення, які стають доступними для всіх дочірніх компонентів. До них входять масив завдань `tasks` та основні функції керування: `addTask`, `updateTask`, `removeTask`, `loadTasks`, `toggleTaskCompleted`. Реалізація структури переданих даних наведена у рис. 3.18.

```
// Повертаємо TasksContext.Provider, щоб надати доступ до завдань і функцій управління ними
return (
  <TasksContext.Provider
    value={{
      tasks, // Масив завдань
      addTask, // Функція для додавання нового завдання
      updateTask, // Функція для оновлення існуючого завдання
      removeTask, // Функція для видалення завдання
      toggleTaskCompleted, // Функція для перемикання статусу виконання завдання
      loadTasks, // Функція для завантаження завдань
    }}
  >
    {children} {/* Вкладені компоненти, які отримують доступ до контексту */}
  </TasksContext.Provider>
);
```

Рисунок 3.18 – Реалізація структури переданих даних

Такий підхід до централізованого керування даними дозволив досягти модульності та забезпечити чисту архітектуру у проєкті.

3.3.3 Модуль користувацького інтерфейсу TaskListScreen

Модуль TaskListScreen реалізує головний екран застосунку, де відображається список усіх створених користувачем завдань. Його функціональність охоплює перегляд, фільтрацію, розгортання опису, відмітку про виконання, редагування та видалення нотаток. Цей екран є центральним елементом взаємодії користувача із застосунком.

Розробка модуля виконана з використанням функціонального компонента React. Для доступу до глобального стану використано хук useContext, який підключається до TasksContext, звідки отримуються масив завдань tasks та функції управління removeTask та updateTask. Це дозволяє компоненту безпосередньо взаємодіяти з даними та виконувати всі необхідні операції без дублювання логіки. Отримання даних та функцій управління з TasksContext у TaskListScreen наведено у рис. 3.19.

```
// Отримуємо завдання, функцію видалення та оновлення з контексту  
const { tasks, removeTask, updateTask } = useContext(TasksContext);
```

Рисунок 3.19 – Отримання даних та функцій управління з TasksContext у TaskListScreen

Для збереження локального стану компонент використовує декілька окремих хуків useState:

- selectedCategory — визначає активну категорію для фільтрації завдань;
- expandedTasks — об'єкт, який відслідковує, які завдання розгорнуті для перегляду повного опису.

Фрагмент коду з використанням хуків useState для локального стану у модулі TaskListScreen наведено у рис. 3.20.

```
// Стан для вибраної категорії (за замовчуванням "all")
const [selectedCategory, setSelectedCategory] = useState('all');
// Стан для розгорнутих завдань (зберігає ID завдань, які розгорнуті)
const [expandedTasks, setExpandedTasks] = useState({});
```

Рисунок 3.20 – Використання хуків useState для локального стану у модулі TaskListScreen.

Фільтрація реалізована за допомогою компонента Picker, що дозволяє змінювати активну категорію й динамічно оновлювати список завдань на екрані. Застосовано умовну логіку: при виборі All відображаються всі нотатки, інакше — лише відповідної категорії. Фільтрація за категорією у TaskListScreen наведена у рис. 3.21.

```
{/* Фільтр завдань за категорією */}
<View style={styles.filterRow}>
  <Text style={styles.label}><t('filterBy')>:</Text>
  <Picker
    selectedValue={selectedCategory}
    onValueChange={setSelectedCategory}
    style={styles.picker}
  >
    <Picker.Item label={t('allTasks')} value="all" />
    <Picker.Item label={t('category.work')} value="work" />
    <Picker.Item label={t('category.shopping')} value="shopping" />
    <Picker.Item label={t('category.home')} value="home" />
    <Picker.Item label={t('category.projects')} value="projects" />
    <Picker.Item label={t('category.other')} value="other" />
  </Picker>
</View>
```

Рисунок 3.21 – Фільтрація за категорією у TaskListScreen

Для забезпечення візуальної ідентифікації категорій завдань у застосунку створено об'єкт categoryColors, у якому кожній категорії відповідає свій унікальний колір. Це дозволяє відображати категорії у відповідному кольоровому оформленні. Реалізація призначення кольорів категоріям завдань наведена у рис. 3.22.

```
// Об'єкт для визначення кольорів категорій завдань
const categoryColors = {
  work: '#A7C7E7',
  shopping: '#B5EAD7',
  home: '#FFDAC1',
  projects: '#CBAACB',
  other: '#E0E0E0',
};
```

Рисунок 3.22 – Призначення кольорів категоріям завдань

Використання categoryColors у TaskListScreen наведено у рис. 3.23.

```
{/* Відображення категорії завдання з кольором */}
<Text style={[styles.taskCategory, { color: categoryColors[item.category] }]}>
  ({t(`category.${item.category}`)})
</Text>
```

Рисунок 3.23 – Використання categoryColors у TaskListScreen

Список завдань відображається через компонент FlatList, що забезпечує продуктивний рендер великої кількості елементів. Використання компонента FlatList для рендеру списку завдань у TaskListScreen наведено у рис. 3.24.

```
{/* Відображення списку завдань або повідомлення про їх відсутність */}
{filteredTasks.length > 0 ? (
  <FlatList
    data={filteredTasks} // Дані для списку
    keyExtractor={item => item.id.toString()} // Унікальний ключ для кожного елемента
    renderItem={renderTask} // Функція рендерингу завдання
  />
) : (
  <Text style={styles.noTasksText}>{t('noTasks')}</Text> // Повідомлення, якщо завдань немає
)}
```

Рисунок 3.24 – Використання компонента FlatList для рендеру списку завдань у TaskListScreen

Кожне завдання має набір кнопок: розгортання опису, позначення як виконаного, редагування та видалення. Кнопки керування завданнями у компоненті TaskListScreen наведені у рис. 3.25.

```

{/* Кнопка позначити завдання виконаним */}
<TouchableOpacity
  style={styles.completeButton}
  onPress={() => toggleComplete(item)}
>
  <Text style={styles.completeText}>✓</Text>
</TouchableOpacity>

{/* Кнопка редагування завдання */}
<TouchableOpacity
  style={styles.editButton}
  onPress={() => editTask(item)}
>
  <Text style={styles.editText}>✎</Text>
</TouchableOpacity>

{/* Кнопка видалення завдання */}
<TouchableOpacity
  style={styles.deleteButton}
  onPress={() => confirmDelete(item.id)}
>
  <Text style={styles.deleteText}>✕</Text>
</TouchableOpacity>

```

Рисунок 3.25 – Кнопки керування завданнями у компоненті
TaskListScreen

Кнопка редагування нотатки працює за допомогою функції `editTask`, яка приймає як параметр об'єкт завдання `task`, після чого за допомогою методу `navigate` з бібліотеки `@react-navigation/native` ініціює перехід на екран редагування нотатки. Водночас разом із переходом передається об'єкт відповідної нотатки як параметр маршруту.. Реалізація функції `editTask` наведена у рис. 3.26.

```

const editTask = (task) => {
  navigation.navigate('EditNote', { task });
};

```

Рисунок 3.26 – Реалізація функції `editTask`

У межах модуля `TaskListScreen` реалізовано функціональність відмітки завдань як виконаних. Це дозволяє користувачу ефективно стежити за прогресом виконання власних справ. Відповідну логіку реалізовано у функції `toggleComplete`. Реалізація функції `toggleComplete` наведена у рис. 3.27.

```
// Позначити задачу виконаною
const toggleComplete = (task) => {
  | updateTask({ ...task, completed: !task.completed });
};
```

Рисунок 3.27 – Реалізація функції toggleComplete

При видаленні застосовується компонент Alert для підтвердження дії, що підвищує надійність взаємодії. Код компоненту Alert наведено у рис. 3.28.

```
// Функція підтвердження видалення завдання
const confirmDelete = (taskId) => {
  Alert.alert(
    t('confirmDeleteTitle'), // Заголовок підтвердження
    t('confirmDeleteMessage'), // Повідомлення підтвердження
    [
      { text: t('cancel'), style: 'cancel' }, // Кнопка скасування
      { text: t('delete'), onPress: () => removeTask(taskId), style: 'destructive' } // Кнопка видалення
    ]
  );
};
```

Рисунок 3.28 – Код компоненту Alert

Щоб реалізувати зручне відображення довгих описів нотаток, у модулі TaskListScreen передбачено можливість розгортання та згортання опису для кожного окремого завдання за допомогою відповідної кнопки. Реалізація кнопки розгортання нотатки наведена у рис. 3.29.

```
<View style={styles.taskItem}>
  /* Кнопка розгортання/згортання опису завдання */
  <TouchableOpacity onPress={() => toggleExpand(item.id)}>
    <Text style={styles.expandButton}>
      | {expandedTasks[item.id] ? '∨' : '>'}
    </Text>
  </TouchableOpacity>
```

Рисунок 3.29 – Кнопка розгортання нотатки

Функціонал розгортання нотатки реалізовано за допомогою функції `toggleExpand`, яка дозволяє перемикати стан розгортання нотатки. Функція `toggleExpand` наведена у рис. 3.30.

```
// Перемикання розгортання опису
const toggleExpand = (taskId) => {
  setExpandedTasks(prev => ({ ...prev, [taskId]: !prev[taskId] }));
};
```

Рисунок 3.30 – Функція `toggleExpand`

Окремим елементом є кнопка додавання нового завдання, яка через обробник `navigation.navigate('AddNote')` переадресовує користувача на екран створення нотатки. Навігація реалізована через бібліотеку `@react-navigation/native`. Реалізація кнопки додавання нового завдання наведена у рис. 3.31.

```
{/* Кнопка додавання нового завдання */}
<TouchableOpacity
  style={styles.addButton}
  onPress={() => navigation.navigate('AddNote')}
>
  <Text style={styles.addButtonText}>+</Text>
</TouchableOpacity>
```

Рисунок 3.31 – Реалізація кнопки додавання нового завдання

Для реалізації підтримки багатомовності застосовано бібліотеку `i18n-js`, з якої імпортовано хук `useTranslation`. За його допомогою інтерфейс автоматично підлаштовується під вибрану мову. Усі текстові елементи отримуються через функцію `t('ключ')`. Хук перекладу наведено на рис. 3.32.

```
const { t } = useTranslation();
```

Рисунок 3.32 – Хук перекладу

На рис. 3.33 зображено інтерфейс екрану TaskListScreen, що демонструє головну сторінку застосунку зі списком завдань. Користувач має змогу здійснювати фільтрацію завдань за категорією, розгортати повний опис, а також виконувати основні дії: відмічати виконані нотатки, редагувати або видаляти нотатки. У нижній частині екрану розташована кнопка створення нового завдання, що забезпечує швидкий доступ до модуля додавання нотатки.



Рисунок 3.33 – Інтерфейс екрану TaskListScreen

3.3.4 Модуль користувацького інтерфейсу AddNoteScreen

Модуль AddNoteScreen реалізує інтерфейс створення нового завдання в мобільному застосунку. Його основна функція — забезпечити зручне введення всіх необхідних полів задачі, включаючи назву, опис, дату, категорію

та параметр нагадування, а також зберегти введену інформацію до глобального стану через контекст.

Розробку здійснено у вигляді функціонального компонента з використанням хуків `useState` для зберігання локального стану введених даних:

- `title` — текстове поле для введення заголовка завдання;
- `description` — текстове поле для додаткової інформації;
- `date` — значення обраної дати;
- `reminder` — булевий прапорець для активації нагадування;
- `category` — обрана категорія завдання зі списку.

Реалізація зберігання локального стану наведена у п.3.2 на рис. 3.9.

У графічному інтерфейсі реалізовано форму введення основних параметрів нотатки. Для заголовка та опису використовуються елементи `TextInput`, що дозволяють користувачу вводити текстові дані. Поле заголовка є однорядковим, а поле опису підтримує багаторядкове введення. Обидва елементи мають локалізовані текстові підказки, що забезпечують зручність взаємодії.

Для вибору категорії нотатки застосовується компонент `Picker`, який надає користувачеві можливість обрати одну з кількох передбачених категорій: робота, покупки, домашні справи, проєкти або інше. Обране значення зберігається у стані компонента для подальшого використання під час збереження нотатки. Такий підхід забезпечує гнучкість у структуризації інформації та адаптацію до потреб користувача.

Для вибору категорії використано компонент `Picker`, що дозволяє користувачу призначити завдання до однієї з них. Обране значення зберігається у полі об'єкта `taskData` та передається до контексту. Реалізація створення форм для введення даних нотатки наведена у рис. 3.34.

```

{/* Поле вводу заголовка */}
<TextInput
  placeholder={t('titlePlaceholder')} // Підказка для заголовка
  value={title} // Значення заголовка
  onChangeText={setTitle} // Оновлення заголовка
  style={styles.input}
/>
{/* Поле вводу опису */}
<TextInput
  placeholder={t('descriptionPlaceholder')} // Підказка для опису
  value={description} // Значення опису
  onChangeText={setDescription} // Оновлення опису
  multiline={true} // Дозволяє кілька рядків
  numberOfLines={4} // Кількість рядків
  style={[styles.input, styles.textArea]}
/>

{/* Вибір категорії */}
<Text style={styles.label}>{t('selectCategory')}:</Text>
<Picker
  selectedValue={category} // Вибрана категорія
  onValueChange={(itemValue) => setCategory(itemValue)} // Оновлення категорії
  style={styles.picker}
>
  <Picker.Item label={t('category.work')} value="work" />
  <Picker.Item label={t('category.shopping')} value="shopping" />
  <Picker.Item label={t('category.home')} value="home" />
  <Picker.Item label={t('category.projects')} value="projects" />
  <Picker.Item label={t('category.other')} value="other" />
</Picker>

```

Рисунок 3.34 – Реалізація створення форм для введення даних нотатки

Для реалізації вибору дати виконання завдання використано компонент `DateTimePicker` із бібліотеки `@react-native-community/datetimepicker`. Спочатку користувач натискає на елемент `TouchableOpacity`, що викликає функцію `setShowDatePicker(true)` — це вмикає відображення вікна вибору дати. Поточне значення дати виводиться у форматі, відповідному до локалі користувача, що реалізовано за допомогою `Intl.DateTimeFormat`.

У разі зміни значення в `DateTimePicker`, нова дата зберігається у стані компонента за допомогою функції `setDate(selectedDate)`, а саме вікно вибору дати автоматично закривається. Такий підхід забезпечує гнучкий і зручний для користувача інтерфейс вибору дати. Реалізацію вибору дати нагадування наведено у рис. 3.35.

```

{/* Вибір дати */}
<TouchableOpacity onPress={() => setShowDatePicker(true)} style={styles.dateTimeButton}>
  <Text style={styles.dateTimeText}>
    {t('selectedDate')}: {new Intl.DateTimeFormat(i18n.language, { day: 'numeric', month: 'long', year: 'numeric' }).format(date)}
  </Text>
</TouchableOpacity>
{showDatePicker && (
  <DateTimePicker
    value={date} // Поточна дата
    mode="date" // Режим вибору дати
    display="default"
    onChange={(event, selectedDate) => {
      setShowDatePicker(false); // Закриваємо вибір дати
      if (selectedDate) {
        setDate(selectedDate); // Оновлюємо дату
      }
    }}
  >
  </DateTimePicker>
)}
)}

```

Рисунок 3.35 – Реалізація вибору дати нагадування

Для вибору часу виконання завдання реалізовано окремий інтерфейс, аналогічний вибору дати. При натисканні на елемент `TouchableOpacity` викликається функція `setShowTimePicker(true)`, що активує компонент `DateTimePicker` у режимі вибору часу.

У випадку, якщо користувач обирає нове значення, подія `onChange` зберігає його в стан компонента за допомогою функції `setTime(selectedTime)`. Реалізація вибору часу нагадування наведено у рис. 3.36.

```

{/* Вибір часу */}
<TouchableOpacity onPress={() => setShowTimePicker(true)} style={styles.dateTimeButton}>
  <Text style={styles.dateTimeText}>
    {t('selectedTime')}: {time.toString().split(':').slice(0, 2).join(':')}
  </Text>
</TouchableOpacity>
{showTimePicker && (
  <DateTimePicker
    value={time} // Поточний час
    mode="time" // Режим вибору часу
    display="default"
    onChange={(event, selectedTime) => {
      setShowTimePicker(false); // Закриваємо вибір часу
      if (selectedTime) {
        setTime(selectedTime); // Оновлюємо час
      }
    }}
  >
  </DateTimePicker>
)}
)}

```

Рисунок 3.36 – Реалізація вибору часу нагадування

Для реалізації можливості надсилання сповіщень застосунок при першому завантаженні екрана створення завдання виконує запит на дозвіл до системних сповіщень. Це реалізовано за допомогою хука `useEffect`, в якому викликається асинхронна функція `requestPermissions`. Хук `useEffect` наведено у рис. 3.37.

```
useEffect(() => {
  // Запит дозволів для сповіщень
  const requestPermissions = async () => {
    const { status } = await Notifications.requestPermissionsAsync();
    if (status !== 'granted') {
      // Відображення повідомлення, якщо дозволи не надані
      Alert.alert(t('error'), t('notificationPermissionDenied'));
    }
  };
});
```

Рисунок 3.37 – Хук `useEffect`

У цій функції використовується метод `Notifications.requestPermissionsAsync()` з бібліотеки `expo-notifications`, який відкриває стандартний діалог операційної системи для надання користувачем дозволу. Якщо користувач відмовляється, програма відображає відповідне попередження за допомогою `Alert.alert`, використовуючи локалізовані повідомлення через функцію `t()`.

Це покращує взаємодію з користувачем, попереджаючи його про обмежену функціональність у разі відмови.

У модулі реалізовано можливість керування сповіщеннями за допомогою графічного перемикача `Switch`, що розміщений у контейнері `View`. Поруч із перемикачем відображається текстовий елемент, який інформує користувача про призначення цього елемента, перекладений відповідно до поточної мови інтерфейсу за допомогою функції локалізації.

Стан перемикача контролюється змінною `enableNotification`, яка зберігається у локальному стані за допомогою хука `useState`. Зміна положення тумблера автоматично викликає функцію `setEnableNotification`, що оновлює

відповідне значення. Це дозволяє користувачеві самостійно обирати, чи бажає він отримувати сповіщення про заплановані завдання. Реалізація перемикача наведена у рис. 3.38.

```

/* Перемикач для увімкнення/вимкнення сповіщень */
<View style={styles.switchContainer}>
  <Text>{t('enableNotification')}

```

Рисунок 3.38 – Реалізація перемикача

Якщо користувач увімкнув сповіщення за допомогою перемикача enableNotification, то створюється дата й час нагадування шляхом поєднання обраних користувачем значень дати та часу. З використанням модуля expo-notifications виконується заплановане створення локального сповіщення через функцію scheduleNotificationAsync. Реалізацію функції scheduleNotificationAsync наведено у рис. 3.39.

```

// Планування сповіщення, якщо увімкнено
if (enableNotification) {
  const trigger = new Date(date);
  trigger.setHours(time.getHours());
  trigger.setMinutes(time.getMinutes());

  Notifications.scheduleNotificationAsync({
    content: {
      title: t('reminderTitle'), // Заголовок сповіщення
      body: t('reminderBody', { title }), // Тіло сповіщення
    },
    trigger, // Час сповіщення
  });
}

navigation.goBack(); // Повернення на попередній екран
};

```

Рисунок 3.39 – Реалізація функції scheduleNotificationAsync

Щоб уникнути створення некоректних або порожніх записів, у модулі `AddNoteScreen` реалізовано перевірку заповнення заголовка перед збереженням завдання. Це реалізовано у функції `saveNote`, яка викликається при спробі підтвердження введення. Функцію `saveNote` наведено у рис. 3.40.

```
const saveNote = () => {
  // Перевірка, чи введено заголовок
  if (!title.trim()) {
    Alert.alert(t('error'), t('titleRequired')); // Відображення помилки, якщо заголовок порожній
    return;
  }
}
```

Рисунок 3.40 – Функція `saveNote`

Умова `if (!title.trim())` перевіряє, чи поле заголовка не є порожнім або заповненим лише пробілами. Якщо заголовок не введено, користувач отримує повідомлення про помилку через `Alert.alert`, де текст повідомлення локалізується за допомогою функції `t()`.

Такий механізм забезпечує базову валідацію введених даних, покращує користувацький досвід та запобігає збереженню неповних завдань.

Після заповнення всіх полів та натискання кнопки збереження викликається функція `addTask`, яка безпосередньо приймає об'єкт із поточними значеннями: `title`, `description`, `date`, `reminder` та `category`. Цей об'єкт формується та передається до глобального контексту через `TasksContext`, де зберігається в локальній базі даних `AsyncStorage`. Після додавання нового завдання виконується автоматичне повернення на головний екран за допомогою `navigation.goBack()`. Виклик функції `addTask` наведено у рис. 3.41.

```
const { addTask, updateTask } = useContext(TasksContext);
```

Рисунок 3.41 – Виклик функції `addTask`

Під час збереження нотатки всі введені користувачем дані — заголовок, опис, дата, час і категорія — об'єднуються в єдиний об'єкт `newTask`. До нього також додається унікальний ідентифікатор: або вже

наявний у разі редагування, або згенерований автоматично. Цей об'єкт передається до відповідних функцій для збереження або оновлення завдання, що забезпечує зручність і послідовність подальшої обробки даних у програмі. Створення нового об'єкта завдань наведено у рис 3.42.

```
// Створення нового об'єкта завдання
const newTask = {
  id: taskId || Date.now(), // Використовуємо існуючий ID або генеруємо новий
  title,
  description,
  date: formattedDate,
  time: formattedTime,
  category,
};
```

Рисунок 3.42 – Створення нового об'єкта завдань

Для реалізації функціоналу редагування наявних нотаток у модулі AddNoteScreen використано хук useEffect, який перевіряє наявність переданої нотатки через параметри маршруту. Якщо така нотатка є, її дані використовуються для попереднього заповнення форми, що забезпечує зручне оновлення вже створених записів. Реалізацію функціоналу редагування завдань наведено у рис. 3.43.

```
useEffect(() => {
  // Якщо передано завдання через route.params, заповнюємо поля
  if (route.params?.task) {
    const { task } = route.params;
    setTitle(task.title); // Встановлюємо заголовок
    setDescription(task.description); // Встановлюємо опис
    setCategory(task.category); // Встановлюємо категорію
    setTaskId(task.id); // Встановлюємо ID завдання
  }
}, [route.params?.task]); // Виконується при зміні route.params.task
```

Рисунок 3.43 – Реалізація функціоналу редагування завдань

Таким чином, модуль користувацького інтерфейсу задовільняє усім пунктам технічного завдання. На рис. 3.44 відображено модуль користувацького інтерфейсу AddNoteScreen.

Рисунок 3.44 – Модуль користувацького інтерфейсу AddNoteScreen

3.3.5 Модуль користувацького інтерфейсу EditNoteScreen

Компонент EditNoteScreen реалізує інтерфейс для редагування вже існуючої нотатки. Він отримує дані нотатки через параметри навігації `route.params` у вигляді об'єкта `task`, який містить усі збережені поля обраного завдання. Це дозволяє одразу ініціалізувати стан форми значеннями редагованої нотатки, що підвищує зручність користування застосунком.

На початку модуля здійснюється імпорт необхідних бібліотек та контексту `TasksContext`, що забезпечує централізований доступ до всіх CRUD-

операцій. Далі компонент оголошується як функціональний компонент з деструктуризацією `route` та `navigation`. Контекст `TasksContext` використовується для отримання функції `updateTask`, яка відповідає за оновлення існуючої нотатки у базі даних `AsyncStorage`. Також об'єкт `task` отримується з параметрів маршруту. Ініціалізація компонента `EditNoteScreen` та отримання параметрів маршруту наведена на рис. 3.45.

```
// Компонент для редагування нотатки
const EditNoteScreen = ({ route, navigation }) => {
  const { updateTask } = useContext(TasksContext); // Отримуємо функцію оновлення завдань із контексту
  const { task } = route.params; // Отримуємо переданий параметр із даними нотатки
```

Рисунок 3.45 – Ініціалізація компонента `EditNoteScreen` та отримання параметрів маршруту

У компоненті `EditNoteScreen` за допомогою хука `useState` створюються змінні стану, які зберігають поточні значення всіх полів форми редагування нотатки. Це дозволяє динамічно оновлювати вміст полів у міру введення користувачем і забезпечує повноцінну взаємодію з формою.

Кожен стан ініціалізується на основі відповідного поля з об'єкта `task`, який було передано з попереднього екрана. Ініціалізація станів форми редагування нотатки наведена у рис. 3.46.

```
// Стани для збереження даних форми
const [title, setTitle] = useState(task.title || ''); // Стан для заголовка
const [description, setDescription] = useState(task.description || ''); // Стан для опису
const [category, setCategory] = useState(task.category || 'other'); // Стан для категорії
const [date, setDate] = useState(task.date ? new Date(task.date) : new Date()); // Стан для дати
const [time, setTime] = useState(task.time ? new Date(`${task.date}T${task.time}`) : new Date()); // Стан для часу
const [showDatePicker, setShowDatePicker] = useState(false); // Стан для відображення вибору дати
const [showTimePicker, setShowTimePicker] = useState(false); // Стан для відображення вибору часу
const [enableNotification, setEnableNotification] = useState(task.enableNotification || false); // Стан для перемикача нагадувань
```

Рисунок 3.46 – Ініціалізація станів форми редагування нотатки

Такий підхід дозволяє компоненту зберігати всі необхідні значення для редагування нотатки, обробляти взаємодію з елементами керування та формувати оновлений об'єкт завдання перед збереженням.

Перед збереженням змін функція `handleSave` виконує початкову перевірку обов'язкового поля заголовка нотатки. Це критично важливо для забезпечення цілісності даних, адже кожне завдання повинно мати осмислену назву, що відображає його зміст. Реалізація валідації заголовку наведена у рис. 3.47.

```
// Перевірка на порожній заголовок перед збереженням
const handleSave = async () => {
  if (!title.trim()) {
    Alert.alert('Помилка', 'Заголовок не може бути порожнім!'); // Перевірка на порожній заголовок
    return;
  }
}
```

Рисунок 3.47 – Реалізація валідації заголовку у `EditNoteScreen`

Після перевірки коректності введених даних виконується попередня обробка дати й часу, оскільки збереження інформації у форматі `Date` є непридатним для подальшої роботи з JSON та для планування сповіщень. Щоб стандартизувати значення, об'єкт дати конвертується у рядок у форматі ISO, після чого з нього виділяється лише частина, що відповідає даті — без часу. Це дозволяє отримати дату у форматі `YYYY-MM-DD`, яка є загальноприйнятим стандартом зберігання. Аналогічно, об'єкт часу перетворюється у текстовий рядок, з якого виділяються лише години та хвилини, аби сформувати зручний формат `HH:mm`. На рис. 3.48 відображена стандартизація дати та часу у потрібний формат.

```
// Форматування дати та часу
const formattedDate = date.toISOString().split('T')[0]; // Формат дати: YYYY-MM-DD
const formattedTime = time.toString().split(':').slice(0, 2).join(':'); // Формат часу: HH:mm
```

Рисунок 3.48 – Стандартизація дати та часу у потрібний формат

Після обробки дати та часу створюється новий об'єкт `updatedTask`, який об'єднує всі оновлені дані користувача з існуючою структурою завдання. За допомогою оператора розгортання `...task` до об'єкта копіюються всі поля попередньої версії нотатки, зокрема її ідентифікатор, а зверху

перезаписуються ті властивості, які були змінені у формі: заголовок, опис, категорія, дата, час та стан нагадування.

Готовий об'єкт передається до функції `updateTask`, яка була отримана з глобального контексту `TasksContext`. Ця функція відповідає за оновлення завдання у загальному масиві всіх завдань, а також за перезапис оновленого списку у локальне сховище `AsyncStorage`.

У разі, якщо перемикач нагадування `enableNotification` був активований, додатково виконується планування локального сповіщення за допомогою модуля `expo-notifications`. Для цього створюється об'єкт типу `Date`, що об'єднує обрану користувачем дату та час.

На завершення виконання функції викликається `navigation.goBack()`, що повертає користувача на попередній екран, де вже відображатиметься оновлена версія редагованої нотатки. Реалізація логіки оновлення даних та обробки сповіщень у `EditNoteScreen` наведена у рис. 3.49.

```
// Створення оновленого об'єкта завдання
const updatedTask = {
  ...task, // Зберігаємо всі інші поля завдання
  title, // Оновлений заголовок
  description, // Оновлений опис
  category, // Оновлена категорія
  date: formattedDate, // Оновлена дата
  time: formattedTime, // Оновлений час
  enableNotification, // Оновлений стан нагадування
};

updateTask(updatedTask); // Оновлюємо завдання через контекст

// Якщо нагадування увімкнене – плануємо його
if (enableNotification) {
  const trigger = new Date(date); // Створюємо тригер для нагадування
  trigger.setHours(time.getHours());
  trigger.setMinutes(time.getMinutes());

  await Notifications.scheduleNotificationAsync({
    content: {
      title: 'Нагадування', // Заголовок сповіщення
      body: `Нотатка: ${title}`, // Тіло сповіщення
    },
    trigger, // Час сповіщення
  });
}

navigation.goBack(); // Повертаємося на попередній екран
```

Рисунок 3.49 – Реалізація логіки оновлення даних та обробки сповіщень у `EditNoteScreen`

Графічне оформлення модулю EditNoteScreen таке ж саме як і у модуля AddNoteScreen. На рис. 3.50 відображено модуль користувацького інтерфейсу EditNoteScreen.

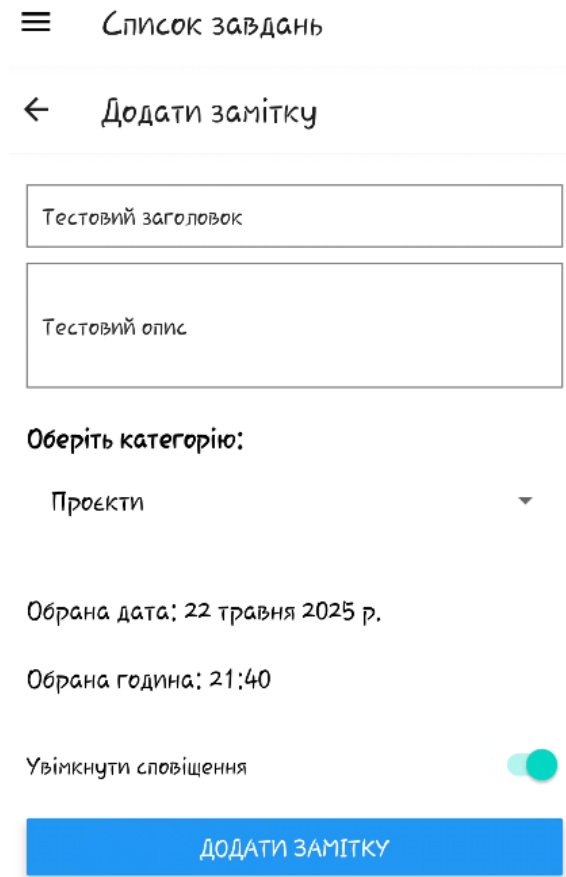


Рисунок 3.50 – Модуль користувацького інтерфейсу EditNoteScreen

3.3.6 Модуль користувацького інтерфейсу CalendarScreen

Модуль CalendarScreen реалізує інтерфейс календаря, який дозволяє користувачеві зручно переглядати завдання, заплановані на певні дні. Він виконує роль навігаційного інструменту, що забезпечує візуальне представлення дати створення або нагадування нотатки у форматі календарної сітки. Завдяки цьому модулю користувач отримує змогу швидко орієнтуватися у розподілі завдань за часом, не вдаючись до ручного пошуку серед усього списку.

У компоненті `CalendarScreen` використовується бібліотека `react-native-calendars`, яка підтримує зміну мови відображення місяців, днів тижня та позначення поточної дати. Для реалізації багатомовної підтримки застосунку спочатку задаються параметри локалізації для української та англійської мов.

Кожна локаль містить масиви з повними та скороченими назвами місяців і днів тижня, а також позначення для поточної дати. Це дозволяє коректно відображати календар відповідно до активної мови інтерфейсу. Локалізація `CalendarScreen` наведена у рис. 3.51.

```
// Налаштування локалізації календаря для української мови
LocaleConfig.locales['uk'] = {
  monthNames: [
    'Січень', 'Лютий', 'Березень', 'Квітень', 'Травень', 'Червень',
    'Липень', 'Серпень', 'Вересень', 'Жовтень', 'Листопад', 'Грудень'
  ],
  monthNamesShort: [
    'Січ', 'Лют', 'Бер', 'Кві', 'Тра', 'Чер',
    'Лип', 'Сер', 'Вер', 'Жов', 'Лис', 'Гру'
  ],
  dayNames: [
    'Неділя', 'Понеділок', 'Вівторок', 'Середа', 'Четвер', 'П'ятниця', 'Субота'
  ],
  dayNamesShort: ['Нд', 'Пн', 'Вт', 'Ср', 'Чт', 'Пт', 'Сб'],
  today: 'Сьогодні'
};

// Налаштування локалізації календаря для англійської мови
LocaleConfig.locales['en'] = {
  monthNames: [
    'January', 'February', 'March', 'April', 'May', 'June',
    'July', 'August', 'September', 'October', 'November', 'December'
  ],
  monthNamesShort: [
    'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
    'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'
  ],
  dayNames: [
    'Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday'
  ],
  dayNamesShort: ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'],
  today: 'Today'
};

const CalendarScreen = () => {
  const { t, i18n } = useTranslation();
  const { tasks } = useContext(TasksContext);
  const [selectedDate, setSelectedDate] = useState('');
  const [locale, setLocale] = useState(i18n.language);

  // При зміні мови оновлюємо локаль календаря
  useEffect(() => {
    LocaleConfig.defaultLocale = i18n.language === 'uk' ? 'uk' : 'en';
    setLocale(i18n.language);
  }, [i18n.language]);
};
```

Рисунок 3.51 – Локалізація `CalendarScreen`

У компоненті `CalendarScreen` змінна `markedDates` визначає, які дати мають бути виділені в календарі. Це об'єкт спеціального формату, що використовується бібліотекою `react-native-calendars` для відображення візуальних маркерів — вибраної дати та наявності нотаток. Формування цього об'єкта здійснюється у два етапи. По-перше, якщо користувач вибрав певну дату, ця дата позначається як вибрана: їй призначається прапорець `selected: true` та колір виділення `selectedColor: 'blue'`. Це дозволяє візуально підсвітити поточно активну дату у календарі. По-друге, метод `reduce()` проходить по всіх завданнях у масиві `tasks` і створює набір дат, які мають хоча б одну нотатку. Для кожної такої дати в об'єкті створюється ключ з відповідним значенням — прапорцем `marked: true` та кольоровою точкою `dotColor: '#2196F3'`.

Таким чином, об'єкт `markedDates` поєднує інформацію про вибрану дату та всі дати, пов'язані з існуючими завданнями, забезпечуючи гнучке і наочне відображення у календарному компоненті. Формування об'єкта `markedDates` для позначення подій у календарі наведено у рис. 3.52.

```
// Відмітки для календаря: вибрана дата + дати з нотатками
const markedDates = {
  ...(selectedDate && {
    [selectedDate]: { selected: true, selectedColor: 'blue' }
  }),
  ...tasks.reduce((acc, task) => {
    const date = task.date;
    acc[date] = { marked: true, dotColor: '#2196F3', ...(acc[date] || {}) };
    return acc;
  }, {})
};
```

Рисунок 3.52 – Формування об'єкта `markedDates` для позначення подій у календарі

У компоненті `CalendarScreen` реалізовано динамічну фільтрацію завдань залежно від обраної дати. Змінна `tasksForSelectedDate` формується за допомогою методу `filter()`, який перебирає весь масив завдань і повертає лише ті елементи, дата яких збігається зі значенням `selectedDate`, обраним у календарі.

У візуальній частині інтерфейсу компонент Calendar відображається з урахуванням локалізації, тем оформлення, та налаштувань першого дня тижня. Вибір дати onDayPress оновлює стан selectedDate, що автоматично запускає повторний рендер списку завдань. Після календаря відображається блок із переліком завдань.

Список завдань реалізований за допомогою компонента FlatList, що забезпечує ефективне відображення навіть великої кількості елементів. Кожен елемент списку містить назву завдання title, категорію, а також опис, якщо він заданий. У разі, якщо на обрану дату не припадає жодного завдання, відображається відповідне повідомлення, що інформує користувача про відсутність нотаток. Реалізація динамічної фільтрації завдань залежно від обраної дати наведена у рис. 3.53.

```
// Фільтрація завдань за вибраною датою
const tasksForSelectedDate = tasks.filter(task => task.date === selectedDate);

return (
  <View style={styles.container}>
    <View style={styles.calendarContainer}>
      <Calendar
        key={locale} // Перерендер при зміні мови
        onDayPress={day => setSelectedDate(day.dateString)}
        markedDates={markedDates}
        // Налаштування: понеділок – перший день тижня
        firstDay={1}
        // Тема відображення
        theme={{
          selectedDayBackgroundColor: 'blue',
          todayTextColor: 'red',
          arrowColor: 'orange',
          monthTextColor: 'black',
          textDayFontSize: 16,
          textMonthFontSize: 20,
          textDayHeaderFontSize: 14
        }}
      />
    </View>

    <View style={styles.tasksContainer}>
      <Text style={styles.tasksHeader}>
        {selectedDate
          ? `${t('notesFor')} ${selectedDate}`
          : t('selectDate')}
      </Text>
      <FlatList
        data={tasksForSelectedDate}
        keyExtractor={item => item.id.toString()}
        renderItem={({ item }) => (
          <View style={styles.taskItem}>
            <Text style={styles.taskTitle}>{item.title}</Text>
            <Text style={styles.taskCategory}>
              {t('category.${item.category}')}
            </Text>
            {item.description ? <Text>{item.description}</Text> : null}
          </View>
        )}
        ListEmptyComponent={
          <Text style={styles.noTasksText}>{t('noTasks')}</Text>
        }
      />
    </View>
  </View>
)
```

Рисунок 3.53 – Реалізація динамічної фільтрації завдань залежно від обраної дати

Таким чином, цей блок завершує повноцінну реалізацію функціональності календаря з інтеграцією локалізованого інтерфейсу, фільтрації даних та адаптивного відображення змісту. Модуль користувацького інтерфейсу CalendarScreen наведено на рис. 3.54.

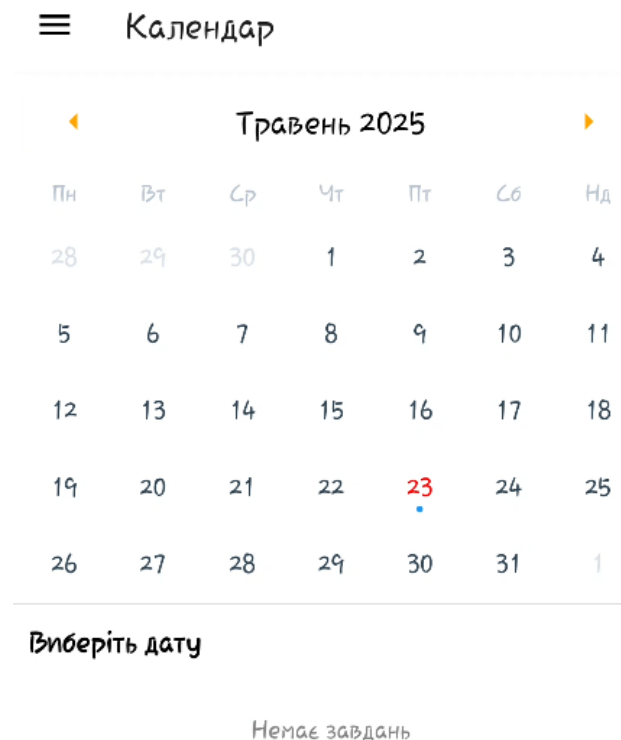


Рисунок 3.54 – Модуль користувацького інтерфейсу CalendarScreen

3.3.7 Модуль користувацького інтерфейсу SettingsScreen

Компонент SettingsScreen відповідає за налаштування інтерфейсу користувача, зокрема за зміну мови застосунку. Основна логіка зосереджена навколо інтеграції з бібліотекою `i18next`, яка використовується для забезпечення багатомовної підтримки.

За допомогою хука `useTranslation()` отримується об'єкт `i18n`, що дозволяє програмно змінювати активну мову, а також функція `t()`, яка використовується для локалізованого виведення текстів. Поточна мова

зберігається у локальному стані `selectedLanguage`, ініціалізованому значенням `i18n.language`.

Функція `handleLanguageChange` викликається при зміні вибраної мови у випадаючому списку. Вона оновлює як стан компонента, так і активну мову в контексті перекладу через `i18n.changeLanguage(language)`. Це забезпечує негайне оновлення мови в усьому застосунку.

Інтерфейс реалізовано за допомогою компонента `Picker`, який дозволяє користувачеві обрати одну з доступних мов — українську або англійську. Всі стилі для елементів інтерфейсу задаються через об'єкт `styles`, що визначений окремо.

Таким чином, цей блок реалізує просту й ефективну механіку перемикання мов, дозволяючи користувачеві миттєво адаптувати інтерфейс відповідно до своїх мовних уподобань. Реалізація зміни локалізації наведена у рис. 3.55.

```
const SettingsScreen = () => {
  const { t, i18n } = useTranslation();
  const [selectedLanguage, setSelectedLanguage] = useState(i18n.language);

  const handleLanguageChange = (language) => {
    setSelectedLanguage(language);
    i18n.changeLanguage(language);
  };

  return (
    <View style={styles.container}>
      <Text style={styles.container}>
        <Text style={styles.label}>{t('changeLanguage')}</Text>
        <View style={styles.pickerContainer}>
          <Picker
            selectedValue={selectedLanguage}
            onValueChange={handleLanguageChange}
            style={styles.picker}
          >
            <Picker.Item label="Українська" value="uk" />
            <Picker.Item label="English" value="en" />
          </Picker>
        </View>
      </View>
    </View>
  );
};
```

Рисунок 3.55 – Реалізація зміни локалізації

На рис. 3.56 зображено модуль користувацького інтерфейсу SettingsScreen.

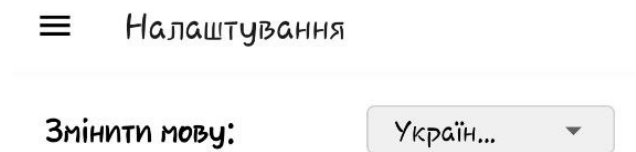


Рисунок 3.56 – Модуль користувацького інтерфейсу SettingsScreen

3.3.8 Модуль навігації застосунку AppNavigator

Модуль бічної навігації реалізує логіку переходу між основними екранами застосунку за допомогою висувного меню Drawer Navigation. Він базується на використанні бібліотеки `@react-navigation/drawer`, що є частиною екосистеми React Navigation і спеціально адаптована для мобільних інтерфейсів. Цей тип навігації забезпечує зручний доступ до ключових розділів застосунку, зокрема екрана списку завдань `TaskListScreen`, календаря `CalendarScreen` та налаштувань `SettingsScreen`. Реалізація навігації між екранами за допомогою Drawer Navigation наведена на рис. 3.57.

```
const Drawer = createDrawerNavigator();

export default function AppNavigator() {
  return (
    <NavigationContainer>
      <Drawer.Navigator initialRouteName="Tasks">
        <Drawer.Screen name="Tasks" component={TaskListScreen} />
        <Drawer.Screen name="Calendar" component={CalendarScreen} />
        <Drawer.Screen name="AddNote" component={AddNoteScreen} options={{ drawerLabel: () => null }} />
        <Drawer.Screen name="Settings" component={SettingsScreen} />
      </Drawer.Navigator>
    </NavigationContainer>
  );
}
```

Рисунок 3.57 – Реалізація навігації між екранами за допомогою Drawer Navigation

Бічна панель відкривається свайпом або натисканням на кнопку меню у верхній частині екрану та відображає перелік доступних маршрутів. Кожен пункт меню веде до відповідного модуля, що дає змогу користувачеві швидко перемикатися між розділами без необхідності повертатися назад або використовувати вкладки. У структурі проєкту реалізація навігації винесена в окремий модуль — AppNavigator, що об'єднує навігаційні стеки та визначає маршрути для всіх екранів. Модуль навігації застосунку AppNavigator зображено на рис. 3.58.

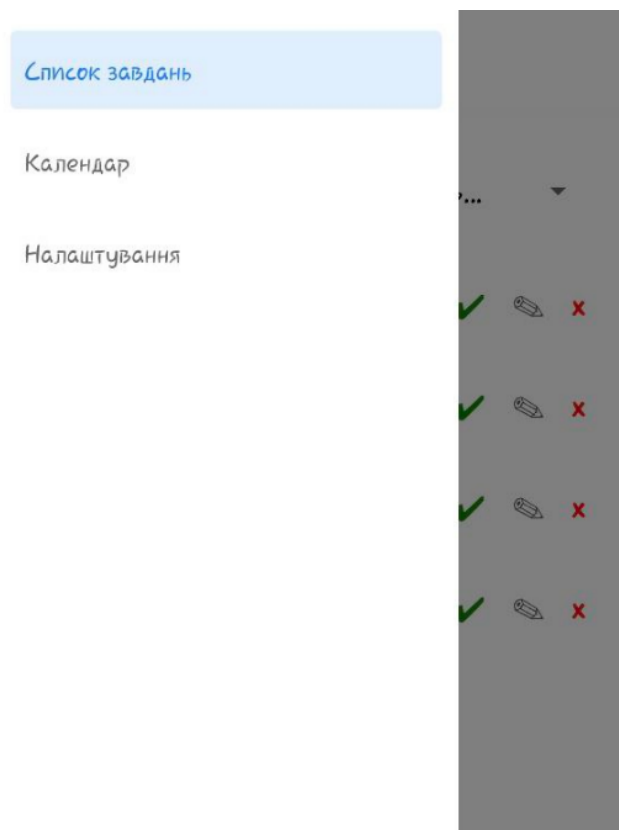


Рисунок 3.58 – Модуль навігації застосунку AppNavigator

Таким чином, модуль бічної навігації виконує роль інтерфейсного каркасу, який забезпечує інтуїтивне керування навігацією, підвищує зручність користування застосунком та дозволяє масштабувати структуру екранів без ускладнення навігаційної логіки.

3.4 Висновки за розділом

У третьому розділі було здійснено детальний опис процесу розробки мобільного застосунку для тайм-менеджменту та планування. Сформовано архітектурну структуру програми, розроблено логічну модель бази даних на основі AsyncStorage, а також реалізовано функціональні модулі, що забезпечують основні сценарії взаємодії з нотатками: створення, редагування, видалення, фільтрацію, перегляд за календарем та зміну мови інтерфейсу.

Кожен модуль було охарактеризовано з позиції його призначення, організації інтерфейсу та взаємодії з централізованим сховищем даних. Окрему увагу приділено модулю TasksContext, який відповідає за централізоване управління станом застосунку та забезпечує узгоджену роботу з локальним зберіганням інформації.

Усі розроблені компоненти інтегровані в єдину архітектуру, що відповідає технічним вимогам та забезпечує стабільну і зручну взаємодію користувача з програмою. Реалізований функціонал дозволяє ефективно виконувати поставлені завдання та створює підґрунтя для подальшого розширення можливостей застосунку.

4 КЕРІВНИЦТВО ПРОГРАМІСТА

4.1 Призначення та умови застосування програми

Розроблений мобільний застосунок призначений для ефективної організації особистого часу користувача. Його основною метою є надання зручного інструменту для фіксації, редагування та перегляду щоденних завдань, нотаток і нагадувань. Програма дозволяє структурувати інформацію за категоріями, встановлювати нагадування на конкретний день і час, а також відстежувати заплановані справи за допомогою вбудованого календаря.

Призначенням застосунку є оптимізація процесу планування повсякденних справ, підвищення продуктивності та уникання пропусків подій або завдань. Завдяки простій логіці взаємодії користувач має змогу швидко занотовувати важливу інформацію, редагувати її за потреби та отримувати своєчасні візуальні нагадування.

Застосунок орієнтовано на широку аудиторію користувачів, які мають потребу в особистому плануванні та фіксації завдань. Насамперед, програма може бути корисною для студентів, які бажають ефективно контролювати розклад занять, дедлайни й інші навчальні активності. Також її функціонал буде зручним для працівників офісного типу або фахівців, яким важливо структурувати робочі завдання, зустрічі та короткі нотатки.

Крім того, застосунок підходить для звичайних користувачів, які прагнуть впорядкувати особисті справи: покупки, побутові завдання, нагадування про події тощо. Простий інтерфейс та мінімальні вимоги до навичок користування дозволяють використовувати програму без попередньої підготовки, що робить її доступною для широкого кола осіб — незалежно від віку та професії.

Однією з ключових умов для користувача є наявність базових навичок користування мобільним пристроєм — інтерфейс програми розроблено таким чином, щоб забезпечити максимально просту та інтуїтивну взаємодію.

4.2 Характеристики програми

Програма працює за умови запуску на мобільному пристрої. Функціонал програмного застосунку наведено у п 1.3.3. Вимоги до технічних засобів та операційної системи наведені у п 1.3.4.

Щоб запустити програмне забезпечення, потрібно встановити файл TaskManagerApp.apk та відкрити програму Task Manager на пристрої з операційною системою Android або iOS.

Програмне забезпечення реалізовано з використанням мови JavaScript у середовищі Visual Studio Code 2020, на базі фреймворку React Native та інструментарію Expo.

4.3 Структура програми

Мобільний застосунок для тайм-менеджменту та планування має наступні характеристики:

а) архітектура:

1) компонентно-орієнтована архітектура з централізованим керуванням станом: застосунок побудований за принципом розділення на функціональні компоненти React, з використанням Context API для централізованого зберігання та обробки даних. Такий підхід забезпечує гнучкість, масштабованість та зручне повторне використання коду;

б) технології:

1) javascript — основна мова програмування, що використовується для створення логіки застосунку;

2) react native — фреймворк для розробки кросплатформених мобільних застосунків, що дозволяє створювати інтерфейс користувача за допомогою компонентів;

3) expo — набір інструментів, який спрощує розробку, тестування та збірку застосунку;

4) `visual studio code` — середовище розробки, що використовувалося для написання коду та організації проєкту;

в) бібліотеки:

1) `@react-native-async-storage/async-storage` — бібліотека для зберігання даних локально у форматі ключ-значення. Використовується для збереження масиву завдань у форматі JSON;

2) `@react-navigation/native` — основна бібліотека для реалізації навігації між екранами. Дозволяє будувати вкладені меню та навігаційні стекові переходи;

3) `react-native-paper` — компонентна бібліотека для побудови інтерфейсу, яка базується на принципах Material Design;

4) `i18n-js` — бібліотека для реалізації багатомовності. Дозволяє перекладати інтерфейс відповідно до вибраної мови;

5) `expo-localization` — надає доступ до мовних налаштувань пристрою, що дозволяє автоматично визначати мову інтерфейсу;

6) `react-native-vector-icons` — забезпечує доступ до колекції іконок, які використовуються в інтерфейсі;

7) `@react-native-community/datetimepicker` — компонент для вибору дати та часу, використовується при створенні нагадувань;

г) функціональність:

1) створення завдань — користувач може додати нове завдання із зазначенням назви, опису, дати, категорії та нагадування;

2) редагування завдань — наявні записи можна змінити, відкривши форму редагування;

3) видалення завдань — будь-яке завдання можна видалити зі списку, після чого воно буде прибрано як з інтерфейсу, так і з локального сховища;

4) відмітка про виконання — завдання можна позначити як виконане, що змінює його відображення у списку;

5) фільтрація завдань — реалізовано сортування за групами для зручного перегляду;

6) навігація між екранами — реалізовано нижню навігаційну панель, яка дозволяє переходити між основними розділами застосунку;

7) підтримка локалізації — інтерфейс підтримує українську та англійську мову;

г) інтерфейс користувача:

1) мінімалістичний дизайн — інтерфейс застосунку побудований за принципами простоти та зручності, без надмірних візуальних елементів, що дозволяє користувачеві сконцентруватися на основному функціоналі;

2) адаптивність — інтерфейс коректно відображається на пристроях з різними розмірами екранів, забезпечуючи комфортне користування як на смартфонах, так і на планшетах;

3) структуроване меню навігації — реалізовано бічну навігаційну панель з текстовими полями для швидкого доступу до основних екранів: списку завдань, календаря та налаштувань;

4) візуальні індикатори стану завдань — виконані завдання відображаються з перекресленим текстом, що дозволяє швидко відрізнити їх від активних;

5) форма створення та редагування завдань — реалізована у вигляді зручного інтерфейсу з полями вводу, перемикачами, вибором дати та категорії.

Мобільний застосунок для керування щоденними завданнями та розкладом є зручним інструментом для планування справ у повсякденному житті. Він поєднує в собі простоту інтерфейсу, гнучкість налаштувань та стабільність роботи, забезпечуючи користувачеві ефективний контроль над особистим часом.

4.4 Вхідні та вихідні дані

4.4.1 Вхідні дані

Вхідні дані в застосунку TaskManagerApp формуються користувачем під час створення або редагування завдань. Основними елементами вхідних даних є: назва завдання, його опис, дата та обраний час нагадування, категорія та позначка виконання. Крім того, користувач має можливість обирати мову інтерфейсу. Усі ці параметри зберігаються локально та безпосередньо впливають на логіку роботи застосунку, відображення елементів на екрані та порядок взаємодії з користувачем.

4.4.2 Вихідні дані

Вихідними даними застосунку TaskManagerApp є інформація, що відображається користувачеві в результаті взаємодії з системою. Зокрема, це список створених завдань із зазначенням їх статусу виконання, назви, опису, категорії та встановленого нагадування. Дані відображаються у вигляді головного списку, календаря або відфільтрованих категорій залежно від вибраних користувачем параметрів. Також до вихідних даних належать повідомлення нагадування про заплановані завдання та поточні мовні налаштування, які зберігаються й враховуються при подальшому запуску програми.

4.5 Висновки за розділом

У даному розділі було розглянуто загальну структуру, функціональні можливості та особливості реалізації мобільного застосунку для тайм-менджменту та планування. Описано основні технології, бібліотеки та архітектурні рішення, що забезпечили зручність використання, автономність і підтримку багатомовності.

Було охарактеризовано основні вхідні та вихідні дані, а також реалізовано логіку їх обробки. Надійне збереження інформації без потреби в інтернеті робить застосунок універсальним помічником для щоденного використання в різних сферах життя.

Окремо розглянуто етап тестування, у ході якого було перевірено працездатність усіх основних компонентів інтерфейсу та логіки збереження, що підтвердило коректність функціонування застосунку відповідно до поставлених вимог.

Таким чином, отримані результати свідчать про те, що програмне забезпечення реалізовано якісно, у повній відповідності до поставлених цілей, і може ефективно використовуватись на практиці.

ВИСНОВКИ

У дипломній кваліфікаційній роботі бакалавра було розглянуто проблему ефективного управління особистим часом та організації щоденних завдань у мобільному середовищі. Основна увага була зосереджена на розробці зручного, автономного застосунку, який дозволяє користувачам планувати справи, отримувати нагадування та структуровано керувати своїм розкладом без необхідності підключення до інтернету. Для досягнення цієї мети було виконано наступні завдання:

- аналіз предметної області: виконано аналіз існуючих мобільних застосунків для тайм-менеджменту та планування та виявлено їх переваги та недоліки, що дозволило обґрунтовано сформулювати вимоги до майбутнього програмного забезпечення;

- розробка технічного завдання: сформульовано вимоги до застосунку, включаючи вимоги до мобільного пристрою. Результатом розробки технічного завдання стало чітке формулювання функціональних та технічних вимог до мобільного застосунку, що визначило його основні можливості, інтерфейс, логіку роботи та мінімальні характеристики пристрою для стабільної роботи;

- вибір середовища розробки та мови програмування: обрано оптимальне середовище та мова програмування для виконання поставлених цілей. Результатом вибору середовища розробки та мови програмування стало забезпечення ефективної, кросплатформної розробки мобільного застосунку з використанням JavaScript та фреймворку React Native у середовищі Visual Studio Code, що дозволило пришвидшити процес створення та розгортання програми;

- розробка архітектури: обрано архітектурну модель застосунку та окреслено ключові структурні компоненти системи, у результаті, створено логічно впорядковану структуру застосунку, яка забезпечує зрозумілу взаємодію між його компонентами, спрощує масштабування та підтримку, а

також забезпечує ефективне керування станом за допомогою Context API та логічний поділ функціональності між екранами та модулями збереження даних;

- реалізація функціоналу: реалізовано основну логіку, що охоплює обробку введених користувачем даних, керування станом застосунку, збереження інформації у локальній базі даних та взаємодію з елементами інтерфейсу.

У результаті розробки реалізовано мобільний застосунок для тайм-менеджменту та планування, який має наступні можливості:

- створення та редагування нотаток: додавання заголовку, опису, категорії та нагадування;
- фільтрація нотаток: за категоріями, датою та станом виконання;
- локалізація застосунку: у застосунку підтримується англійська та українська мова.

Розроблений застосунок повністю відповідає вимогам, визначеним у технічному завданні, і забезпечує користувачам зручний та функціональний інструмент для тайм-менеджменту та планування.

Усі завдання кваліфікаційної роботи бакалавра повністю виконано.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Notion. Help & Support – Documentation [Electronic resource]. – Access mode: <https://help.obsidian.md/>.
2. Obsidian. Developer Documentation [Electronic resource]. – Access mode: <https://docs.obsidian.md/>.
3. Google Keep Help – Documentation [Electronic resource]. – Access mode: <https://support.google.com/keep/>.
4. System requirements for Android devices [Electronic resource]. – Access mode: <https://source.android.com/docs/compatibility>.
5. Apple. About iOS 15 Updates [Electronic resource]. – Access mode: <https://support.apple.com/en-us/108051>.
6. Visual Studio [Electronic resource]. – Access mode: <https://visualstudio.microsoft.com/>.
7. Replit Documentation [Electronic resource]. – Access mode: <https://docs.replit.com/>.
8. Android Studio Documentation [Electronic resource]. – Access mode: <https://developer.android.com/studio>.
9. Visual Studio Code Documentation [Electronic resource]. – Access mode: <https://code.visualstudio.com/docs/>.
10. Посібник JavaScript [Електронний ресурс]. – Режим доступу: <https://developer.mozilla.org/uk/docs/Web/JavaScript/Guide/>.
11. Kotlin Documentation [Electronic resource]. – Access mode: <https://kotlinlang.org/docs/home.html>.
12. Python Documentation [Electronic resource]. – Access mode: <https://docs.python.org/3/>.
13. React Native Documentation [Electronic resource]. – Access mode: <https://reactnative.dev/docs/getting-started/>.
14. React Documentation [Electronic resource]. – Access mode: <https://react.dev/learn/>.

15. Expo Documentation [Electronic resource]. – Access mode:
<https://docs.expo.dev/>.

16. AsyncStorage Documentation [Electronic resource]. – Access mode:
<https://react-native-async-storage.github.io/async-storage/>.

ДОДАТОК А
Текст програми

App.js

```

import React, { useEffect } from 'react';
import { NavigationContainer } from '@react-navigation/native';
import { createDrawerNavigator } from '@react-navigation/drawer';
import TaskListScreen from './screens/TaskListScreen';
import CalendarScreen from './screens/CalendarScreen';
import SettingsScreen from './screens/SettingsScreen';
import { createStackNavigator } from '@react-navigation/stack';
import AddNoteScreen from './screens/AddNoteScreen';
import EditNoteScreen from './screens/EditNoteScreen';
import { TasksProvider } from './context/TasksContext';
import 'i18n'; // Підключення i18n для локалізації
import { useTranslation } from 'react-i18next';
import * as Notifications from 'expo-notifications';

const Drawer = createDrawerNavigator();
const Stack = createStackNavigator();

function MainNavigator() {
  const { t } = useTranslation(); // Використання перекладу

  return (
    <Stack.Navigator>
      <Stack.Screen
        name="TaskList"
        component={TaskListScreen}
        options={{ title: t('taskList'), headerShown: false }} // Приховування
заголовка
      />
    </Stack.Navigator>
  );
}

```

```

    <Stack.Screen
      name="AddNote"
      component={AddNoteScreen}
      options={{ title: t('addNote') }}
    />
    <Stack.Screen
      name="EditNote"
      component={EditNoteScreen}
      options={{ title: t('editNote') }}
    />
  </Stack.Navigator>
);
}

export default function App() {
  const { t } = useTranslation(); // Використання перекладу

  useEffect(() => {
    const requestPermissions = async () => {
      const { status } = await Notifications.getPermissionsAsync();
      if (status !== 'granted') {
        await Notifications.requestPermissionsAsync();
      }
    };

    requestPermissions();
  }, []);

  return (
    <TasksProvider>

```

```

<NavigationContainer>
  <Drawer.Navigator initialRouteName="TaskListDrawer">
    <Drawer.Screen
      name="TaskListDrawer"
      component={MainNavigator}
      options={{ title: t('TaskListDrawer') }}
    />
    <Drawer.Screen
      name="Calendar"
      component={CalendarScreen}
      options={{ title: t('Calendar') }}
    />
    <Drawer.Screen
      name="Settings"
      component={SettingsScreen}
      options={{ title: t('settings') }}
    />
  </Drawer.Navigator>
</NavigationContainer>
</TasksProvider>
);
}

```

TaskContext.js

```

import React, { createContext, useState, useEffect } from 'react';
import AsyncStorage from '@react-native-async-storage/async-storage';

// Створюємо контекст для завдань
export const TasksContext = createContext();

```

```

// Провайдер контексту
export const TasksProvider = ({ children }) => {
  const [tasks, setTasks] = useState([]);

  // Завантаження завдань з AsyncStorage
  const loadTasks = async () => {
    try {
      const stored = await AsyncStorage.getItem('tasks');
      if (stored) setTasks(JSON.parse(stored));
    } catch (e) {
      console.error('Не вдалося завантажити завдання:', e);
    }
  };

  // Збереження завдань у AsyncStorage
  const saveTasks = async updated => {
    try {
      await AsyncStorage.setItem('tasks', JSON.stringify(updated));
    } catch (e) {
      console.error('Не вдалося зберегти завдання:', e);
    }
  };

  // Додає нове завдання (за замовчуванням не виконане)
  const addTask = taskData => {
    const newTask = {
      id: Date.now(),
      ...taskData,
      completed: false // поле для позначення виконаних
    };
  };

```

```

};

const updated = [newTask, ...tasks];
setTasks(updated);
saveTasks(updated);
};

// Оновлює наявне завдання (використовується й для markCompleted)
const updateTask = updatedTask => {
  const updated = tasks.map(t =>
    t.id === updatedTask.id ? updatedTask : t
  );
  setTasks(updated);
  saveTasks(updated);
};

// Видаляє завдання
const removeTask = id => {
  const updated = tasks.filter(t => t.id !== id);
  setTasks(updated);
  saveTasks(updated);
};

// Перемикає стан виконання завдання
const toggleTaskCompleted = id => {
  const updated = tasks.map(t =>
    t.id === id ? { ...t, completed: !t.completed } : t
  );
  setTasks(updated);
  saveTasks(updated);
};

```

```

useEffect(() => {
  loadTasks();
}, []);

return (
  <TasksContext.Provider
    value={{
      tasks,
      addTask,
      updateTask,
      removeTask,
      toggleTaskCompleted,
      loadTasks,
    }}
  >
    {children}
  </TasksContext.Provider>
);
};

```

TaskListScreen.js

```

import React, { useContext, useState } from 'react';
import { View, FlatList, Text, StyleSheet, TouchableOpacity, Alert } from
'react-native';

import { useTranslation } from 'react-i18next';
import { Picker } from '@react-native-picker/picker';
import { TasksContext } from '../context/TasksContext';

```

```

// Об'єкт для визначення кольорів категорій
const categoryColors = {
  work: '#A7C7E7',
  shopping: '#B5EAD7',
  home: '#FFDAC1',
  projects: '#CBAACB',
  other: '#E0E0E0',
};

const TaskListScreen = ({ navigation }) => {
  const { t } = useTranslation();
  // Отримуємо завдання, функцію видалення та оновлення з контексту
  const { tasks, removeTask, updateTask } = useContext(TasksContext);
  // Стан для вибраної категорії (за замовчуванням "all")
  const [selectedCategory, setSelectedCategory] = useState('all');
  // Стан для розгорнутих завдань (зберігає ID завдань, які розгорнуті)
  const [expandedTasks, setExpandedTasks] = useState({});

  // Функція підтвердження видалення завдання
  const confirmDelete = (taskId) => {
    Alert.alert(
      t('confirmDeleteTitle'), // Заголовок підтвердження
      t('confirmDeleteMessage'), // Повідомлення підтвердження
      [
        { text: t('cancel'), style: 'cancel' }, //Кнопка скасування
        { text: t('delete'), onPress: () => removeTask(taskId), style: 'destructive'
} // Кнопка видалення
      ]
    );
  };
};

```

```

// Тепер передаємо тільки task
const editTask = (task) => {
  navigation.navigate('EditNote', { task });
};

// Перемикання розгортання опису
const toggleExpand = (taskId) => {
  setExpandedTasks(prev => ({ ...prev, [taskId]: !prev[taskId] }));
};

// Позначити задачу виконаною
const toggleComplete = (task) => {
  updateTask({ ...task, completed: !task.completed });
};

const filteredTasks = (selectedCategory === 'all'
  ? tasks
  : tasks.filter(task => task.category === selectedCategory)
).slice().sort((a, b) => (b.completed ? 1 : 0) - (a.completed ? 1 : 0));

const renderTask = ({ item }) => {
  if (!item) return null;

  return (
    <View style={styles.taskItem}>
      {/* Кнопка розгортання/згортання опису */}
      <TouchableOpacity onPress={() => toggleExpand(item.id)}>
        <Text style={styles.expandButton}>
          {expandedTasks[item.id] ? '∨' : '>'}
        </Text>
      </TouchableOpacity>
    </View>
  );
};

```

```

</Text>
</TouchableOpacity>

<View style={styles.taskContent}>
  <Text
    style={[
      styles.taskTitle,
      item.completed && styles.taskTitleCompleted
    ]}
  >
    {item.title}
  </Text>
  {expandedTasks[item.id] && (
    <Text style={styles.taskDescription}>{item.description}</Text>
  )}
  {/* Відображення категорії завдання за кольором */}
  <Text style={[styles.taskCategory, { color:
categoryColors[item.category] }]}>
    ({t(`category.${item.category}`)})
  </Text>
</View>
  {/* Кнопка позначити завдання виконаним */}
  <TouchableOpacity
    style={styles.completeButton}
    onPress={() => toggleComplete(item)}
  >
    <Text style={styles.completeText}> ✓ </Text>
  </TouchableOpacity>

  {/* Кнопка редагування завдання */}

```

```

<TouchableOpacity
  style={styles.editButton}
  onPress={() => editTask(item)}
>

  <Text style={styles.editText}>✎.</Text>
</TouchableOpacity>

  { /* Кнопка видалення завдання */ }
  <TouchableOpacity
    style={styles.deleteButton}
    onPress={() => confirmDelete(item.id)}
  >
    <Text style={styles.deleteText}>x</Text>
  </TouchableOpacity>
</View>
);
};

return (
  <View style={styles.container}>
    { /* Фільтр завдань за категорією */ }
    <View style={styles.filterRow}>
      <Text style={styles.label}>{t('filterBy')}:</Text>
      <Picker
        selectedValue={selectedCategory}
        onValueChange={setSelectedCategory}
        style={styles.picker}
      >
        <Picker.Item label={t('allTasks')} value="all" />
        <Picker.Item label={t('category.work')} value="work" />

```

```

    <Picker.Item label={t('category.shopping')} value="shopping" />
    <Picker.Item label={t('category.home')} value="home" />
    <Picker.Item label={t('category.projects')} value="projects" />
    <Picker.Item label={t('category.other')} value="other" />
  </Picker>
</View>

  {/* Відображення списку завдань або повідомлення про їх
відсутність */}
  {filteredTasks.length > 0 ? (
    <FlatList
      data={filteredTasks} // Дані для списку
      keyExtractor={item => item.id.toString()} // Унікальний ключ для
кожного елемента
      renderItem={renderTask} // Функція для рендерингу завдання
    />
  ) : (
    <Text style={styles.noTasksText}>{t('noTasks')}</Text> //
Повідомлення, якщо завдань немає
  )}

  {/* Кнопка додавання нового завдання */}
  <TouchableOpacity
    style={styles.addButton}
    onPress={() => navigation.navigate('AddNote')}
  >
    <Text style={styles.addButtonText}>+</Text>
  </TouchableOpacity>
</View>
);

```

```
};
```

```
const styles = StyleSheet.create({
  container: { flex: 1, padding: 20, backgroundColor: '#fff' },
  filterRow: {
    flexDirection: 'row',
    alignItems: 'center',
    justifyContent: 'space-between',
    marginBottom: 10,
  },
  label: { fontSize: 16, fontWeight: 'bold' },
  picker: { width: 150 },

  taskItem: {
    flexDirection: 'row',
    alignItems: 'center',
    paddingVertical: 8,
  },
  expandButton: { fontSize: 18, paddingHorizontal: 8 },

  taskContent: { flex: 1 },
  taskTitle: { fontSize: 16, fontWeight: 'bold' },
  taskTitleCompleted: { textDecorationLine: 'line-through', color: 'gray' },
  taskDescription: { fontSize: 14, color: 'gray' },
  taskCategory: { fontSize: 14, fontWeight: 'bold' },

  completeButton: { paddingHorizontal: 8 },
  completeText: { fontSize: 18, color: 'green' },

  editButton: { paddingHorizontal: 8 },
```

```

editText: { fontSize: 18, color: '#555' },

deleteButton: { paddingHorizontal: 8 },
deleteText: { color: 'red', fontSize: 18, fontWeight: 'bold' },

noTasksText: { textAlign: 'center', marginTop: 20, fontSize: 14, color:
'gray' },

addButton: {
  position: 'absolute',
  bottom: 20,
  alignSelf: 'center',
  width: 50,
  height: 50,
  backgroundColor: '#2196F3',
  borderRadius: 25,
  justifyContent: 'center',
  alignItems: 'center',
  elevation: 5,
},
addButtonText: { color: 'white', fontSize: 24, fontWeight: 'bold' },
});

export default TaskListScreen;

```

AddNoteScreen.js

```

import React, { useState, useContext, useEffect } from 'react';
import { View, TextInput, Button, TouchableOpacity, Text, StyleSheet,
Switch, Alert } from 'react-native';

```

```

import { Picker } from '@react-native-picker/picker';
import DateTimePicker from '@react-native-community/datetimepicker';
import { useNavigation } from '@react-navigation/native';
import { useTranslation } from 'react-i18next';
import { TasksContext } from '../context/TasksContext';
import * as Notifications from 'expo-notifications';

const AddNoteScreen = ({ route, navigation }) => {
  const { addTask, updateTask } = useContext(TasksContext);
  const { t, i18n } = useTranslation();

  const [title, setTitle] = useState("");
  const [description, setDescription] = useState("");
  const [date, setDate] = useState(new Date());
  const [time, setTime] = useState(new Date());
  const [showDatePicker, setShowDatePicker] = useState(false);
  const [showTimePicker, setShowTimePicker] = useState(false);
  const [category, setCategory] = useState('other');
  const [taskId, setTaskId] = useState(null);
  const [enableNotification, setEnableNotification] = useState(false); //

```

Перемикач для сповіщень

```

useEffect(() => {
  // Запит дозволів для сповіщень
  const requestPermissions = async () => {
    const { status } = await Notifications.requestPermissionsAsync();
    if (status !== 'granted') {
      // Відображення повідомлення, якщо дозволи не надані
      Alert.alert(t('error'), t('notificationPermissionDenied'));
    }
  }

```

```
};
```

```
requestPermissions();
}, []);
```

```
useEffect(() => {
  // Якщо передано завдання через route params, заповнюємо поля
  if (route.params?.task) {
    const { task } = route.params;
    setTitle(task.title); // Встановлюємо заголовок
    setDescription(task.description); // Встановлюємо опис
    setCategory(task.category); // Встановлюємо категорію
    setTaskId(task.id); // Встановлюємо ID завдання
  }
}, [route.params?.task]); // Виконується при зміні route.params.task
```

```
const saveNote = () => {
  // Перевірка, чи введено заголовок
  if (!title.trim()) {
    Alert.alert(t('error'), t('titleRequired')); // Відображення помилки, якщо
заголовок порожній
    return;
  }
}
```

```
const formattedDate = date.toISOString().split('T')[0];
const formattedTime = time.toTimeString().split(':').slice(0, 2).join(':');
```

```
// Створення нового об'єкта завдання
const newTask = {
```

id: taskId || Date.now(), // Використовуємо існуючий ID або
генеруємо новий

```
title,
description,
date: formattedDate,
time: formattedTime,
category,
};
```

```
if (taskId) {
  updateTask(newTask);
} else {
  addTask(newTask);
  // Планування сповіщення, якщо увімкнено
  if (enableNotification) {
    const trigger = new Date(date);
    trigger.setHours(time.getHours());
    trigger.setMinutes(time.getMinutes());

    Notifications.scheduleNotificationAsync({
      content: {
        title: t('reminderTitle'), // Заголовок сповіщення
        body: t('reminderBody', { title }), // Тіло сповіщення
      },
      trigger, // Час сповіщення
    });
  }
}

navigation.goBack(); // Повернення на попередній екран
};
```

```

return (
  <View style={styles.container}>
    {/* Поле вводу заголовка */}
    <TextInput
      placeholder={t('titlePlaceholder')} // Підказка для заголовка
      value={title} // Значення заголовка
      onChangeText={setTitle} // Оновлення заголовка
      style={styles.input}
    />
    {/* Поле вводу опису */}
    <TextInput
      placeholder={t('descriptionPlaceholder')} // Підказка для опису
      value={description} // Значення опису
      onChangeText={setDescription} // Оновлення опису
      multiline={true} // Дозволяє кілька рядків
      numberOfLines={4} // Кількість рядків
      style={[styles.input, styles.textArea]}
    />

    {/* Вибір категорії */}
    <Text style={styles.label}>{t('selectCategory')}</Text>
    <Picker
      selectedValue={category} // Вибрана категорія
      onValueChange={(itemValue) => setCategory(itemValue)} //
Оновлення категорії
      style={styles.picker}
    >
      <Picker.Item label={t('category.work')} value="work" />
      <Picker.Item label={t('category.shopping')} value="shopping" />

```

```

<Picker.Item label={t('category.home')} value="home" />
<Picker.Item label={t('category.projects')} value="projects" />
<Picker.Item label={t('category.other')} value="other" />
</Picker>

```

```

{/* Вибір дати */}
<TouchableOpacity  onPress={() =>  setShowDatePicker(true)}
style={styles.dateTimeButton}>
  <Text style={styles.dateTimeText}>
    {t('selectedDate')}: {new Intl.DateTimeFormat(i18n.language, { day:
'numeric', month: 'long', year: 'numeric' }).format(date)}
  </Text>
</TouchableOpacity>
{showDatePicker && (
  <DateTimePicker
    value={date} // Поточна дата
    mode="date" // Режим вибору дати
    display="default"
    onChange={(event, selectedDate) => {
      setShowDatePicker(false); // Закриваємо вибір дати
      if (selectedDate) {
        setDate(selectedDate); // Оновлюємо дату
      }
    }}
  />
)}

{/* Вибір часу */}
<TouchableOpacity  onPress={() =>  setShowTimePicker(true)}
style={styles.dateTimeButton}>

```

```

<Text style={styles.dateTimeText}>
  {t('selectedTime')}: {time.toTimeString().split(':').slice(0, 2).join(':')}
</Text>
</TouchableOpacity>
{showTimePicker && (
  <DateTimePicker
    value={time} // Поточний час
    mode="time" // Режим вибору часу
    display="default"
    onChange={(event, selectedTime) => {
      setShowTimePicker(false); // Закриваємо вибір часу
      if (selectedTime) {
        setTime(selectedTime); // Оновлюємо час
      }
    }}
  />
)}

{/* Перемикач для увімкнення.вимкнення сповіщень */}
<View style={styles.switchContainer}>
  <Text>{t('enableNotification')}</Text>
  <Switch
    value={enableNotification} // Значення перемикача
    onValueChange={setEnableNotification} // Оновлення стану
    перемикача
  />
</View>

  <Button title={t('addNote')} onPress={saveNote} />
</View>

```

```
);  
};  
  
const styles = StyleSheet.create({  
  container: {  
    flex: 1,  
    padding: 20,  
    backgroundColor: '#fff',  
  },  
  input: {  
    height: 40,  
    borderColor: 'gray',  
    borderWidth: 1,  
    marginBottom: 10,  
    padding: 10,  
  },  
  textArea: {  
    height: 80,  
  },  
  label: {  
    marginTop: 10,  
    fontSize: 16,  
    fontWeight: 'bold',  
  },  
  picker: {  
    height: 50,  
    marginBottom: 20,  
  },  
  switchContainer: {  
    flexDirection: 'row',
```

```

    alignItems: 'center',
    justifyContent: 'space-between',
    marginVertical: 10,
  },
  dateTimeButton: {
    marginVertical: 10,
  },
  dateTimeText: {
    fontSize: 16,
  },
});

```

```
export default AddNoteScreen;
```

EditNoteScreen.js

```

import React, { useState, useContext } from 'react';
import { View, TextInput, Button, StyleSheet, Text, Switch,
TouchableOpacity, Alert } from 'react-native';
import { Picker } from '@react-native-picker/picker';
import DateTimePicker from '@react-native-community/datetimepicker';
import * as Notifications from 'expo-notifications';
import { useTranslation } from 'react-i18next';
import { TasksContext } from '../context/TasksContext'; // Імпортуємо
контекст завдань

const EditNoteScreen = ({ route, navigation }) => {
  const { updateTask } = useContext(TasksContext); // Отримуємо функцію
оновлення завдань із контексту
  const { t, i18n } = useTranslation(); // Використовуємо локалізацію

```

```
const { task } = route.params; // Отримуємо переданий параметр із даними нотатки
```

```
// Стани для збереження даних форми
```

```
const [title, setTitle] = useState(task.title || ""); // Стан для заголовка
```

```
const [description, setDescription] = useState(task.description || ""); // Стан для опису
```

```
const [category, setCategory] = useState(task.category || 'other'); // Стан для категорії
```

```
const [date, setDate] = useState(task.date ? new Date(task.date) : new Date()); // Стан для дати
```

```
const [time, setTime] = useState(task.time ? new Date(`${task.date}T${task.time}`) : new Date()); // Стан для часу
```

```
const [showDatePicker, setShowDatePicker] = useState(false); // Стан для відображення вибору дати
```

```
const [showTimePicker, setShowTimePicker] = useState(false); // Стан для відображення вибору часу
```

```
const [enableNotification, setEnableNotification] = useState(task.enableNotification || false); // Стан для перемикача нагадувань
```

```
// Функція для збереження змін
```

```
const handleSave = async () => {
```

```
  if (!title.trim()) {
```

```
    Alert.alert(t('error'), t('titleRequired')); // Використовуємо ключі локалізації для помилки
```

```
    return;
```

```
  }
```

```
// Форматування дати та часу
```

```
const formattedDate = date.toISOString().split('T')[0]; // Формат дати:
YYYY-MM-DD
```

```
const formattedTime = time.toTimeString().split(':').slice(0, 2).join(':'); //
Формат часу: HH:mm
```

```
// Створення оновленого об'єкта завдання
```

```
const updatedTask = {
  ...task, // Зберігаємо всі інші поля завдання
  title, // Оновлений заголовок
  description, // Оновлений опис
  category, // Оновлена категорія
  date: formattedDate, // Оновлена дата
  time: formattedTime, // Оновлений час
  enableNotification, // Оновлений стан нагадування
};
```

```
updateTask(updatedTask); // Оновлюємо завдання через контекст
```

```
// Якщо нагадування увімкнене — плануємо його
```

```
if (enableNotification) {
  const trigger = new Date(date); // Створюємо тригер для нагадування
  trigger.setHours(time.getHours());
  trigger.setMinutes(time.getMinutes());
}
```

```
await Notifications.scheduleNotificationAsync({
  content: {
    title: t('reminderTitle'), // Заголовок сповіщення
    body: t('reminderBody', { title }), // Тіло сповіщення
  },
  trigger, // Час сповіщення
});
```

```

    });
}

navigation.goBack(); // Повертаємося на попередній екран
};

return (
  <View style={styles.container}>
    {/* Поле вводу для заголовка */}
    <TextInput
      placeholder={t('titlePlaceholder')} // Підказка для заголовка
      value={title}
      onChangeText={setTitle}
      style={styles.input}
    />
    {/* Поле вводу для опису */}
    <TextInput
      placeholder={t('descriptionPlaceholder')} // Підказка для опису
      value={description}
      onChangeText={setDescription}
      multiline={true}
      numberOfLines={4}
      style={[styles.input, styles.textArea]}
    />

    {/* Вибір категорії */}
    <Text style={styles.label}>{t('selectCategory')}</Text>
    <Picker
      selectedValue={category}
      onValueChange={(itemValue) => setCategory(itemValue)}
    />
  </View>
);

```

```

        style={styles.picker}
      >
        <Picker.Item label={t('category.work')} value="work" />
        <Picker.Item label={t('category.shopping')} value="shopping" />
        <Picker.Item label={t('category.home')} value="home" />
        <Picker.Item label={t('category.projects')} value="projects" />
        <Picker.Item label={t('category.other')} value="other" />
      </Picker>

      { /* Вибір дати */ }
      <TouchableOpacity    onPress={()    =>    setShowDatePicker(true)}
style={styles.dateTimeButton}>
        <Text style={styles.dateTimeText}>
          {t('selectedDate')}: {new Intl.DateTimeFormat(i18n.language, { day:
'numeric', month: 'long', year: 'numeric' }).format(date)}
        </Text>
      </TouchableOpacity>
      {showDatePicker && (
        <DateTimePicker
          value={date}
          mode="date"
          display="default"
          onChange={(event, selectedDate) => {
            setShowDatePicker(false);
            if (selectedDate) {
              setDate(selectedDate);
            }
          }}
        />
      )}
    )}

```

```

    {/ * Вибір часу */}
    <TouchableOpacity    onPress={()    =>    setShowTimePicker(true)}
style={styles.dateTimeButton}>
    <Text style={styles.dateTimeText}>
        {t('selectedTime')}: {time.toTimeString().split(':').slice(0, 2).join(':')}
    </Text>
</TouchableOpacity>
{showTimePicker && (
    <DateTimePicker
        value={time}
        mode="time"
        display="default"
        onChange={(event, selectedTime) => {
            setShowTimePicker(false);
            if (selectedTime) {
                setTime(selectedTime);
            }
        }}
    />
)}

{/ * Перемикач для увімкнення/вимкнення нагадувань */}
<View style={styles.switchContainer}>
    <Text>{t('enableNotification')}</Text>
    <Switch
        value={enableNotification}
        onValueChange={setEnableNotification}
    />
</View>

```

```

    { /* Кнопка для збереження змін */ }
    <Button title={t('saveNote')} onPress={handleSave} />
  </View>
);
};

```

```

const styles = StyleSheet.create({
  container: {
    flex: 1,
    padding: 20,
    backgroundColor: '#fff',
  },
  input: {
    height: 40,
    borderColor: 'gray',
    borderWidth: 1,
    marginBottom: 10,
    padding: 10,
  },
  textArea: {
    height: 80,
  },
  label: {
    marginTop: 10,
    fontSize: 16,
    fontWeight: 'bold',
  },
  picker: {
    height: 50,
  },
});

```

```

    marginBottom: 20,
  },
  switchContainer: {
    flexDirection: 'row',
    alignItems: 'center',
    justifyContent: 'space-between',
    marginVertical: 10,
  },
  dateTimeButton: {
    marginVertical: 10,
  },
  dateTimeText: {
    fontSize: 16,
  },
});

```

```
export default EditNoteScreen;
```

CalendarScreen.js

```

import React, { useState, useEffect, useContext } from 'react';
import { View, Text, FlatList, StyleSheet } from 'react-native';
import { Calendar, LocaleConfig } from 'react-native-calendars';
import { useTranslation } from 'react-i18next';
import { TasksContext } from '../context/TasksContext';

```

```
// Налаштування локалізації календаря для української мови
```

```

LocaleConfig.locales['uk'] = {
  monthNames: [
    'Січень', 'Лютий', 'Березень', 'Квітень', 'Травень', 'Червень',

```

```

        'Липень', 'Серпень', 'Вересень', 'Жовтень', 'Листопад', 'Грудень'
    ],
    monthNamesShort: [
        'Січ', 'Лют', 'Бер', 'Кві', 'Тра', 'Чер',
        'Лип', 'Сер', 'Вер', 'Жов', 'Лис', 'Гру'
    ],
    dayNames: [
        'Неділя', 'Понеділок', 'Вівторок', 'Середа', 'Четвер', 'П'ятниця',
        'Субота'
    ],
    dayNamesShort: ['Нд', 'Пн', 'Вт', 'Ср', 'Чт', 'Пт', 'Сб'],
    today: 'Сьогодні'
};

// Налаштування локалізації календаря для англійської мови
LocaleConfig.locales['en'] = {
    monthNames: [
        'January', 'February', 'March', 'April', 'May', 'June',
        'July', 'August', 'September', 'October', 'November', 'December'
    ],
    monthNamesShort: [
        'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
        'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'
    ],
    dayNames: [
        'Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
        'Saturday'
    ],
    dayNamesShort: ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'],
    today: 'Today'
};

```

```
};
```

```
const CalendarScreen = () => {
  const { t, i18n } = useTranslation();
  const { tasks } = useContext(TasksContext);
  const [selectedDate, setSelectedDate] = useState("");
  const [localeKey, setLocaleKey] = useState(0); // Стан для примусового
перерендеру
```

```
  // При зміні мови оновлюємо локаль календаря
  useEffect(() => {
    LocaleConfig.defaultLocale = i18n.language === 'uk' ? 'uk' : 'en';
    setLocaleKey(prevKey => prevKey + 1); // Оновлюємо ключ для
примусового перерендеру
  }, [i18n.language]);
```

```
  // Відмітки для календаря: вибрана дата + дати з нотатками
  const markedDates = {
    ...(selectedDate && {
      [selectedDate]: { selected: true, selectedColor: 'blue' }
    }),
    ...tasks.reduce((acc, task) => {
      const date = task.date;
      acc[date] = { marked: true, dotColor: '#2196F3', ...(acc[date] || {}) };
      return acc;
    }, {})
  };
};
```

```
  // Фільтрація завдань за вибраною датою
```

```
const tasksForSelectedDate = tasks.filter(task => task.date ===
selectedDate);
```

```
return (
  <View style={styles.container}>
    <View style={styles.calendarContainer}>
      <Calendar
        key={localeKey}          // Примусовий перерендер при зміні мови
        onDayPress={day => setSelectedDate(day.dateString)}
        markedDates={markedDates}
        firstDay={1}              // Понеділок — перший день тижня
        theme={{
          selectedDayBackgroundColor: 'blue',
          todayTextColor: 'red',
          arrowColor: 'orange',
          monthTextColor: 'black',
          textDayFontSize: 16,
          textMonthFontSize: 20,
          textDayHeaderFontSize: 14
        }}
      />
    </View>

    <View style={styles.tasksContainer}>
      <Text style={styles.tasksHeader}>
        {selectedDate
          ? `${t('notesFor')} ${selectedDate}`
          : t('selectDate')}
      </Text>
      <FlatList
```

```

data={tasksForSelectedDate}
keyExtractor={item => item.id.toString()}
renderItem=(({ item }) => (
  <View style={styles.taskItem}>
    <Text style={styles.taskTitle}>{item.title}</Text>
    <Text style={styles.taskCategory}>
      {t(`category.${item.category}`)}
    </Text>
    {item.description ? <Text>{item.description}</Text> : null}
  </View>
)}
ListEmptyComponent={
  <Text style={styles.noTasksText}>{t('noTasks')}</Text>
}
/>
</View>
</View>
);
};

const styles = StyleSheet.create({
  container: { flex: 1, backgroundColor: '#fff' },
  calendarContainer: {
    borderBottomWidth: 1,
    borderBottomColor: '#ddd'
  },
  tasksContainer: {
    flex: 1,
    padding: 10
  },
});

```

```
tasksHeader: {
  fontSize: 16,
  fontWeight: 'bold',
  marginBottom: 10
},
taskItem: {
  paddingVertical: 8,
  borderBottomWidth: 1,
  borderBottomColor: '#eee'
},
taskTitle: {
  fontWeight: 'bold',
  fontSize: 16
},
taskCategory: {
  fontSize: 14,
  color: 'gray',
  fontStyle: 'italic',
  marginBottom: 4
},
noTasksText: {
  textAlign: 'center',
  marginTop: 20,
  fontSize: 14,
  color: 'gray'
}
});

export default CalendarScreen;
```

SettingScreen.js

```

import React, { useState } from 'react';
import { View, Text, StyleSheet, TouchableOpacity, Linking } from 'react-
native';
import { Picker } from '@react-native-picker/picker';
import { useTranslation } from 'react-i18next';

const SettingsScreen = () => {
  const { t, i18n } = useTranslation();
  const [selectedLanguage, setSelectedLanguage] = useState(i18n.language);

  const handleLanguageChange = (language) => {
    setSelectedLanguage(language);
    i18n.changeLanguage(language);
  };

  return (
    <View style={styles.container}>
      { /* Зміна мови */ }
      <View style={styles.row}>
        <Text style={styles.label}>{t('changeLanguage')}</Text>
        <View style={styles.pickerContainer}>
          <Picker
            selectedValue={selectedLanguage}
            onValueChange={handleLanguageChange}
            style={styles.picker}
          >
            <Picker.Item label="Українська" value="uk" />
            <Picker.Item label="English" value="en" />

```

```

    </Picker>
  </View>
</View>
</View>
);
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    padding: 20,
    justifyContent: 'flex-start',
    backgroundColor: '#fff',
  },
  row: {
    flexDirection: 'row',
    alignItems: 'center',
    justifyContent: 'space-between',
    marginBottom: 20,
  },
  label: {
    fontSize: 18,
    fontWeight: 'bold',
    textAlignVertical: 'center',
  },
  pickerContainer: {
    width: 150,
    height: 30,
    justifyContent: 'center',
    backgroundColor: '#f0f0f0',
  },
});

```

```

    borderWidth: 1,
    borderColor: '#ccc',
    borderRadius: 5,
  },
  picker: {
    fontSize: 16,
    height: 30,
    textAlign: 'center',
  },
});

```

```
export default SettingsScreen;
```

AppNavigator.js

```

import React from 'react';
import { NavigationContainer } from '@react-navigation/native';
import { createDrawerNavigator } from '@react-navigation/drawer';
import TaskListScreen from '../screens/TaskListScreen';
import CalendarScreen from '../screens/CalendarScreen';
import AddNoteScreen from '../screens/AddNoteScreen';
import SettingsScreen from '../screens/SettingsScreen';

const Drawer = createDrawerNavigator();

export default function AppNavigator() {
  return (
    <NavigationContainer>
      <Drawer.Navigator initialRouteName="Tasks">
        <Drawer.Screen name="Tasks" component={TaskListScreen} />

```

```

        <Drawer.Screen name="Calendar" component={CalendarScreen} />
        <Drawer.Screen name="AddNote" component={AddNoteScreen}
options={{ drawerLabel: () => null }} />
        <Drawer.Screen name="Settings" component={SettingsScreen} />
    </Drawer.Navigator>
</NavigationContainer>
);
}

```

i18n.js

```

import i18n from 'i18next';
import { initReactI18next } from 'react-i18next';
import en from './locales/en.json';
import uk from './locales/uk.json';
import 'intl-pluralrules';
i18n.use(initReactI18next).init({
  resources: {
    en: { translation: en },
    uk: { translation: uk },
  },
  lng: 'en',
  fallbackLng: 'en',
  interpolation: {
    escapeValue: false,
  },
});

export default i18n;

```

uk.json

```
{  
  "selectCategory": "Оберіть категорію",  
  "category": {  
    "work": "Робота",  
    "shopping": "Покупки",  
    "home": "Домашні справи",  
    "projects": "Проекти",  
    "other": "Інше"},  
  "welcome": "Ласкаво просимо",  
  "settings": "Налаштування",  
  "addNote": "Додати замітку",  
  "selectLanguage": "Виберіть мову",  
  "english": "Англійська",  
  "ukrainian": "Українська",  
  "TaskListMain": " ",  
  "allTasks": "Всі завдання",  
  "titlePlaceholder": "Заголовок",  
  "descriptionPlaceholder": "Опис",  
  "editNote": "Редагування нотатки",  
  "selectedDate": "Обрана дата",  
  "notesFor": "Нотатки на",  
  "selectedTime": "Обраний час",  
  "notesOn": "Нотатки на",  
  "noTasks": "Немає завдань",  
  "AddNote": "Додати замітку",  
  "saveNote": "Зберегти замітку",  
  "EditNote": "Редагувати замітку",  
  "TaskListDrawer": "Список завдань",
```

```

"filterBy": "Фільтрування за групою",
"Calendar": "Календар",
"Settings": "Налаштування",
"deleteConfirmationTitle": "Видалити завдання",
"deleteConfirmationMessage": "Ви впевнені, що хочете видалити це
завдання?",
"cancel": "Скасувати",
"delete": "Видалити",
"confirmDeleteTitle": "Видалення нотатки",
"confirmDeleteMessage": "Ви впевнені, що хочете видалити цю
нотатку?",
"changeLanguage": "Змінити мову",
"userInfo": "FAQ",
"error": "Помилка",
"titleRequired": "Заголовок обов'язковий",
"technicalSupport": "Технічна підтримка",
"enableNotification": "Увімкнути сповіщення",
"calendar": {
    "monthNames": ["Січень", "Лютий", "Березень", "Квітень",
"Травень", "Червень", "Липень", "Серпень", "Вересень", "Жовтень",
"Листопад", "Грудень"],
    "monthNamesShort": ["Січ", "Лют", "Бер", "Кві", "Тра", "Чер", "Лип",
"Сер", "Вер", "Жов", "Лис", "Гру"],
    "dayNames": ["Неділя", "Понеділок", "Вівторок", "Середа",
"Четвер", "П'ятниця", "Субота"],
    "dayNamesShort": ["Нд", "Пн", "Вт", "Ср", "Чт", "Пт", "Сб"],
    "today": "Сьогодні"},
    "noTasksForThisDay": "Немає задач на цей день",
    "selectDate": "Виберіть дату"
}

```

en.json

```

{
  "selectCategory": "Select Category",
  "category": {
    "work": "Work",
    "shopping": "Shopping",
    "home": "Home",
    "projects": "Projects",
    "other": "Other"},
  "notesFor": "Notes for",
  "welcome": "Welcome",
  "addNote": "Add Note",
  "settings": "Settings",
  "error": "Error",
  "confirmDeleteTitle": "Confirm Delete Task",
  "confirmDeleteMessage": "Are you sure you want to delete this task?",
  "titleRequired": "Title is required",
  "enableNotification": "Enable notifications",
  "selectLanguage": "Select Language",
  "english": "English",
  "ukrainian": "Ukrainian",
  "TaskListMain": " ",
  "AddNote": "Add Note",
  "saveNote": "Save Note",
  "EditNote": "Edit Note",
  "TaskListDrawer": "Task List",
  "Calendar": "Calendar",
  "allTasks": "All Tasks",
  "selectedDate": "Selected Date",

```

```

    "selectedTime": "Selected Time",
    "notesOn": "Notes on",
    "noTasks": "No tasks",
    "titlePlaceholder": "Title",
    "descriptionPlaceholder": "Description",
    "Settings": "Settings",
    "filterBy": "Filter by group",
    "deleteConfirmationTitle": "Delete Task",
    "deleteConfirmationMessage": "Are you sure you want to delete this
task?",
    "cancel": "Cancel",
    "delete": "Delete",
    "changeLanguage": "Change language",
    "noTasksForThisDay": "No tasks for this day",
    "selectDate": "Select a date",
    "userInfo": "FAQ",
    "editNote": "Edit Note",
    "technicalSupport": "Technical Support",
    "calendar": {
        "monthNames": ["January", "February", "March", "April", "May",
"June", "July", "August", "September", "October", "November", "December"],
        "monthNamesShort": ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
"Aug", "Sep", "Oct", "Nov", "Dec"],
        "dayNames": ["Sunday", "Monday", "Tuesday", "Wednesday",
"Thursday", "Friday", "Saturday"],
        "dayNamesShort": ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"],
        "today": "Today"}
    }

```

ДОДАТОК Б
Слайди презентації



Рисунок Б.1 – Слайд 1

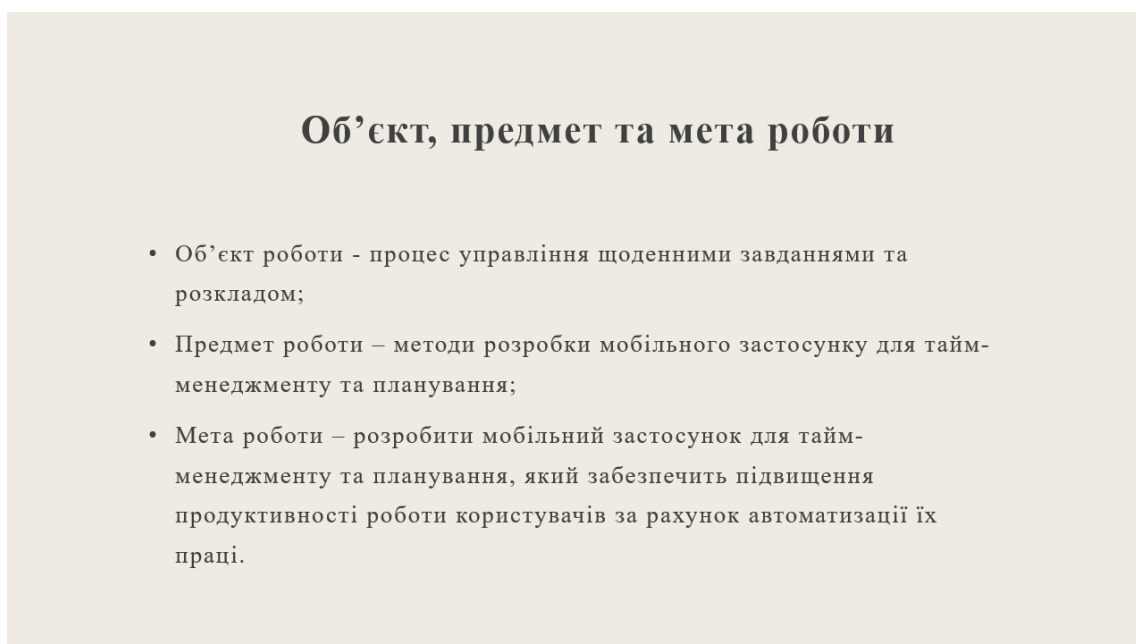


Рисунок Б.2 – Слайд 2

Постановка завдання

1. Аналіз проблемної області
2. Огляд існуючих рішень
3. Розробка технічного завдання
4. Вибір мови програмування та фреймворків
5. Вибір середовища програмування
6. Розробка архітектури програми
7. Реалізація функціоналу застосунку

Рисунок Б.3 – Слайд 3

Порівняння існуючих аналогів

Критерій	Google Keep	Notion	Obsidian
Безкоштовність	Так	Частково	Частково
Автономність	Частково	Частково	Так
Фільтрація нотаток	Ні	Так	Так
Кросплатформеність	Так	Так	Так
Легкість інтерфейсу	Так	Ні	Ні

Рисунок Б.4 – Слайд 4

Вибір мови програмування

Критерій	JavaScript	Kotlin	Python
Популярність використання	Дуже висока	Висока для Android але середня для IOS	Висока загалом, але низька у мобільній розробці
Наявність фреймворків для кросплатформеної розробки	React Native	Kotlin Multiplatform Mobile	Kivy, BeeWare
Простота вивчення	Так	Ні	Так
Швидкодія	Середня	Висока	Низька
Підтримка нативного функціоналу	Повна	Повна	Низька

Рисунок Б.5 – Слайд 5

Вибір середовища розробки

Критерій	Microsoft Visual Studio	Replit	Android Studio	Microsoft Visual Studio Code
Підтримка JavaScript	Обмежена	Повна	Часткова	Повна
Підтримка Expo	Відсутня	Неповна, через шаблони	Відсутня	Повна, через плагіни
Підтримка React Native	Часткова	Часткова	Часткова	Повна
Підтримка Git/GitHub	Так	Так	Так	Так
Навантаження на систему	Високе	Низьке	Високе	Низьке
Інтерфейс	Складний	Простий	Складний	Простий
Розширюваність	Висока	Обмежена	Середня	Дуже висока
Вартість	Безкоштовна Community версія	Безкоштовна з платними функціями	Безкоштовна	Безкоштовна

Рисунок Б.6 – Слайд 6

Архітектура програмного забезпечення



Рисунок Б.7 – Слайд 7

Прецедент “Дії користувача в мобільному застосунку”

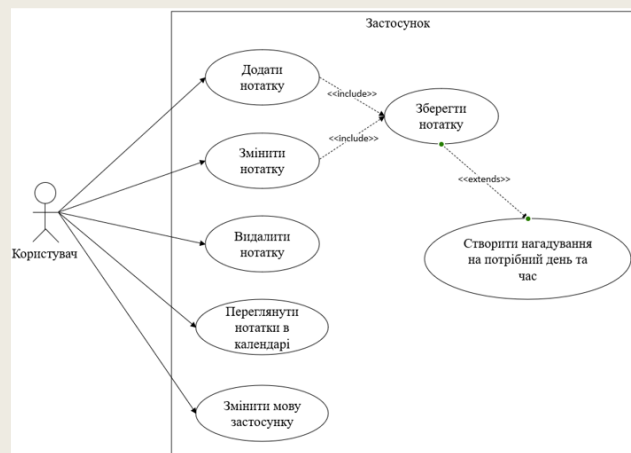


Рисунок Б.8 – Слайд 8

Структура об'єкта нотатки у локальній базі
даних застосунку

Поле	Тип даних	Опис поля
id	number	Унікальний ідентифікатор який генерується автоматично для кожної нотатки
title	string	Заголовок нотатки, який вводиться користувачем
description	string	Опис змісту нотатки
date	string	Дата і час нагадування
reminder	boolean	Нагадування
tags	string	Категорія

Рисунок Б.9 – Слайд 9

Приклад реалізації функціоналу застосунку

Реалізація відображення списку завдань

```
/* Відображення списку завдань або повідомлення про їх відсутність */
{filteredTasks.length > 0 ? (
  <FlatList
    data={filteredTasks} // Дані для списку
    keyExtractor={item => item.id.toString()} // Унікальний ключ для кожного елемента
    renderItem={renderTask} // Функція рендерингу завдання
  />
) : (
  <Text style={styles.noTasksText}>{t('noTasks')}</Text> // Повідомлення, якщо завдань немає
)>
```

Створення нового об'єкта нотатки

```
// Створення нового об'єкта завдання
const newTask = {
  id: taskId || Date.now(), // Використовуємо існуючий ID або генеруємо новий
  title,
  description,
  date: formattedDate,
  time: formattedTime,
  category,
};
```

Реалізація фільтру завдань за категорією

```
<View style={styles.container}>
  /* Фільтр завдань за категорією */
  <View style={styles.filterRow}>
    <Text style={styles.label}>{t('filterBy')}</Text>
    <Picker
      selectedValue={selectedCategory}
      onValueChange={setSelectedCategory}
      style={styles.picker}>
      <Picker.Item label={t('allTasks')} value="all" />
      <Picker.Item label={t('category.work')} value="work" />
      <Picker.Item label={t('category.shopping')} value="shopping" />
      <Picker.Item label={t('category.home')} value="home" />
      <Picker.Item label={t('category.projects')} value="projects" />
      <Picker.Item label={t('category.other')} value="other" />
    </Picker>
  </View>
</View>
```

Реалізація ініціалізації станів форми для редагування нотатки

```
// Стан для зберігання даних форми
const [title, setTitle] = useState(task.title || ''); // Стан для заголовка
const [description, setDescription] = useState(task.description || ''); // Стан для опису
const [category, setCategory] = useState(task.category || 'other'); // Стан для категорії
const [date, setDate] = useState(task.date ? new Date(task.date) : new Date()); // Стан для дати
const [time, setTime] = useState(task.time ? new Date(`${task.date}${task.time}`) : new Date()); // Стан для часу
const [showDatePicker, setShowDatePicker] = useState(false); // Стан для відображення вибору дати
const [showTimePicker, setShowTimePicker] = useState(false); // Стан для відображення вибору часу
const [enableNotification, setEnableNotification] = useState(task.enableNotification || false); // Стан для перемикача нагадувань
```

Рисунок Б.10 – Слайд 10

Налаштування локалізації у застосунку

Приклад налаштування локалізації

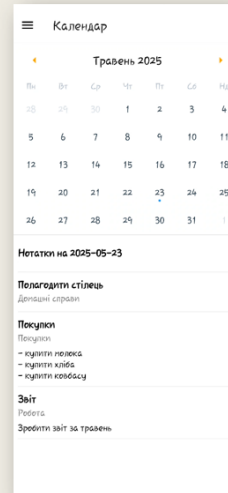
```
// Налаштування локалізації календаря для української мови
localeconfig.locales["uk"] = {
  monthNames: [
    'Січень', 'Лютий', 'Березень', 'Квітень', 'Травень', 'Червень',
    'Липень', 'Серпень', 'Вересень', 'Жовтень', 'Листопад', 'Грудень'
  ],
  monthNamesShort: [
    'Січ', 'Лют', 'Бер', 'Квіт', 'Трав', 'Чер',
    'Лип', 'Сер', 'Вер', 'Жов', 'Лис', 'Гру'
  ],
  dayNames: [
    'Неділя', 'Понеділок', 'Вівторок', 'Середа', 'Четвер', 'П'ятниця', 'Субота'
  ],
  dayNamesShort: ['На', 'Пн', 'Вт', 'Ср', 'Чт', 'Пт', 'Сб'],
  today: 'сьогодні'
};
```

Приклад налаштування перемикача мови у коді

```
const SettingsScreen = () => {
  const { t, i18n } = useTranslation();
  const { selectedLanguage, setSelectedLanguage } = useState(i18n.language);

  const handleLanguageChange = (language) => {
    setSelectedLanguage(language);
    i18n.changeLanguage(language);
  };

  return (
    <View style={styles.container}>
      <Text style={styles.title}>{t('settings.title')}</Text>
      <Text style={styles.label}>{t('changeLanguage')}</Text>
      <View style={styles.pickerContainer}>
        <Picker
          selectedValue={selectedLanguage}
          onValueChange={handleLanguageChange}
          style={styles.picker}>
          <Picker.Item label="Українська" value="uk" />
          <Picker.Item label="English" value="en" />
        </Picker>
      </View>
    </View>
  );
};
```



CalendarScreen



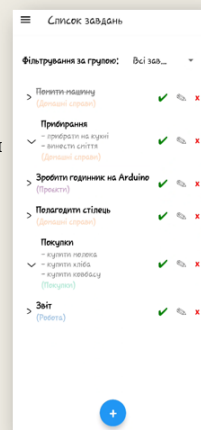
SettingsScreen

Рисунок Б.11 – Слайд 11

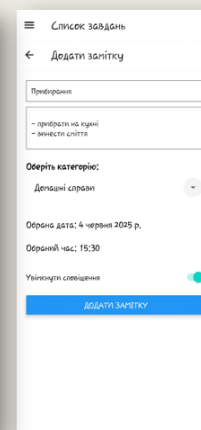
Користувачькі функції застосунку

За допомогою цих графічних інтерфейсів та їх функціоналу, користувач має змогу здійснювати наступні операції у застосунку:

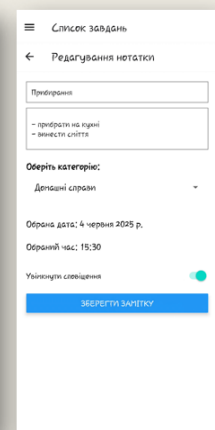
- Перегляд створених нотаток;
- Фільтрація нотаток за групою;
- Додавання нотаток;
- Позначення нотаток, як виконаних;
- Редагування нотаток;
- Видалення нотаток.



TaskListScreen



AddNoteScreen



EditNoteScreen

Рисунок Б.12 – Слайд 12

Висновки

У дипломній роботі було виконано наступні завдання:

- Проаналізовано питання ефективного зберігання нотаток і зручного управління щоденними справами;
- Здійснено аналіз сучасних мобільних застосунків для нотаток та виявлено їх переваги та недоліки;
- Виконано проектування архітектури мобільного застосунку;
- Розроблено мобільний застосунок для зберігання нотаток з можливістю автономної роботи.

Рисунок Б.13 – Слайд 13