МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНЕ НЕКОМЕРЦІЙНЕ ПІДПРИЄМСТВО
"ДЕРЖАВНИЙ УНІВЕРСИТЕТ "КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ"
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач випускової кафедри
_____ Аліна САВЧЕНКО
«___»_____2024 р.

# КВАЛІФІКАЦІЙНА РОБОТА

**(ПОЯСНЮВАЛЬНА ЗАПИСКА)**

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ МАГІСТРА
ЗА ОСВІТНЬО-ПРОФЕСІЙНОЮ ПРОГРАМОЮ
«ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ПРОЕКТУВАННЯ»

**Тема: «Мобільний застосунок для оптимізованого управління подіями з використанням Jetpack Compose»**

Виконавець: Іван ЛАГОДА

Керівник: к.т.н., доцент Вікторія СИДОРЕНКО

Нормоконтролер: к.т.н., доцент Олена ТОЛСТІКОВА

КИЇВ 2024

ДЕРЖАВНЕ НЕКОМЕРЦІЙНЕ ПІДПРИЄМСТВО "ДЕРЖАВНИЙ УНІВЕРСИТЕТ "КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ"

Факультет *комп'ютерних наук та технологій*

Кафедра *комп'ютерних інформаційних технологій*

Спеціальність *122 «Комп'ютерні науки»*

Освітньо-професійна програма *«Інформаційні технології проектування»*

ЗАТВЕРДЖУЮ:
Завідувач кафедри
_____Аліна САВЧЕНКО
(підпис)
«___»_____2024 р.

**ЗАВДАННЯ**
**на виконання кваліфікаційної роботи**

*Лагоди Івана Дмитровича*
(ПІБ випускника)

1. Тема кваліфікаційної роботи: «Мобільний застосунок для оптимізованого управління подіями з використанням Jetpack Compose» затверджена наказом ректора № 1782/ст від 06.09.2024р.

2. Термін виконання роботи: з 26 серпня 2024 року по 03 грудня 2024 року.

3. Вихідні дані до роботи: застосунок на мові програмування Kotlin та з використанням технологій Retrofit 2, Room, Dagger Hilt і Jetpack Compose для демонстрації мобільного додатку для оптимізованого управління подіями.

4. Зміст пояснювальної записки: 1. Огляд предметної області. 2. Проектування мобільного додатку 3. Розробка та тестування мобільного додатку.

5. Перелік обов'язкового ілюстративного матеріалу: 1. Історія розвитку інструментів для управління подіями. 2. Проблеми існуючих інструментів упраління подіями. 3. Метод вирішення проблеми. 4. Вибір та обґрунтування інструментів проектування. 5. Архітектура мобільного додатку. 6. Розробка мобільного додатку. 7. Презентація основного функціоналу.

Календарний план-графік

| № з/п | Завдання | Термін виконання | Підпис керівника |
|---|---|---|---|
| 1 | Огляд та аналіз предметної області. Написання 1 розділу, представлення керівнику. | 26.08.2024-20.09.2024 | |
| 2 | Вибір та опис використаних технологій. Розробка веб-застосунку. Написання 2 розділу, представлення керівнику. | 23.09.2024-10.10.2024 | |
| 3 | Написання 3 розділу, представлення керівнику. | 11.10.2024-14.11.2024 | |
| 4 | Загальне редагування та друк пояснювальної записки. | 15.11.2024-19.11.2024 | |
| 5 | Проходження нормоконтролю, перепліт пояснювальної записки. | 20.11.2024-25.11.2024 | |
| 6 | Розробка тексту доповіді. Оформлення графічного матеріалу для презентації | 26.11.2024-03.12.2024 | |

7. Дата видачі завдання 26.08.2024р.

Керівник кваліфікаційної роботи _____ Вікторія СИДОРЕНКО
(підпис керівника)

Завдання прийняв до виконання _____ Іван ЛАГОДА
(підпис випускника)

# РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи на тему: «Розробка мобільного застосунку для оптимізованого управління подіями з використанням Jetpack Compose» містить: 83 сторінки, 77 рисунків, 17 інформаційних джерел.

**Об’єкт дослідження** – процес оптимізованого управління подіями.

**Предмет дослідження** – методи, засоби та технології розробки мобільного застосунку для оптимізованого управління подіями.

**Мета кваліфікаційної роботи** – розробка мобільного застосунку для оптимізованого управління подіями з використанням сучасних фреймворків, бібліотек та інструментів.

**Методи дослідження** – логічний, алгоритмічний аналіз, порівняльний, аналіз інформаційних джерел, моделювання та симуляція.

Результати кваліфікаційної роботи можуть бути використані для ознайомлення з процесом та методами розробки мобільних додатків, а також для ознаймоленням із їх архітектурою.

МОБІЛЬНИЙ ДОДАТОК, KOTLIN, JETPACK COMPOSE, RETROFIT 2, ROOM, DAGGER HILT.

**ЗМІСТ**

## ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

| | | |
|---|---|---|
| API | – | Application Programming Interface - інтерфейс прикладного програмування |
| JSON | – | JavaScript Object Notation – формат запису об’єктів JavaScript |
| HTTP | – | Hyper Text Transfer Protocol – протокол передачі гіпертексту |
| URL | – | Uniform Resource Locator - Уніфікований покажчик інформаційного ресурсу |
| CRUD | – | Create, Read, Update, Delete - чотири базові функції, які моделі повинні виконувати |
| UI | – | User Interface – графічний інтерфейс |

# ВСТУП

У сучасному світі, що стрімко змінюється, ефективне управління часом і організація діяльності стають дедалі важливішими. Зростаюча складність життєвих процесів, потреба в багатозадачності та вимога досягати високих результатів у короткі терміни створюють значний попит на інноваційні підходи до організації подій. Одним із таких підходів є використання мобільних додатків, що відкривають нові можливості для планування та управління часом.

Розвиток цифрових технологій значно змінив підхід до вирішення повсякденних завдань, включаючи планування особистих подій. Сучасні мобільні додатки пропонують зручний інструмент для автоматизації цього процесу, зменшуючи потребу в паперових органайзерах чи стаціонарних рішеннях. В умовах глобальної мобільності такі додатки стають необхідністю для зручного доступу до інформації у будь-який момент. Проте, незважаючи на широкий вибір рішень, багато користувачів стикаються з проблемою недостатньої адаптації існуючих додатків до їхніх персональних потреб. Це підкреслює актуальність створення індивідуальних рішень, які б відповідали конкретним запитам.

**Мета і завдання дослідження.** Метою роботи є розробка мобільного додатку для управління подіями, який задовольняє індивідуальні потреби користувача. Для досягнення цієї мети було поставлено такі завдання:

- аналіз сучасного стану ринку мобільних додатків для управління подіями;
- вибір відповідних технологій і методів розробки додатку;
- створення інтерфейсу користувача, орієнтованого на простоту та інтуїтивність;
- тестування додатку в реальних умовах.

**Об’єкт і предмет дослідження.** Об’єктом дослідження є процеси організації особистих подій. Предметом дослідження є розробка мобільного додатку для управління цими подіями.

**Методи дослідження.** У роботі використано методи порівняльного аналізу для вивчення існуючих додатків, а також методи програмування з акцентом на використання фреймворку Jetpack Compose для розробки мобільного рішення.

**Наукова новизна отриманих результатів.** Наукова новизна полягає в удосконаленні існуючих підходів до створення мобільного додатку для управління подіями, за рахунок інтеграції з іншими сервісами, кешування даних, офлайн синхронізації та відправки повідомлень перед початком події, що дозволяє значно покращити ефективність та швидкість організації заходів, забезпечуючи доступність, автоматизацію та гнучкість у процесі планування подій.

**Практичне значення отриманих результатів.** Результати дослідження мають прикладне значення, оскільки розроблений додаток може використовуватися користувачами для покращення організації особистого часу. Розроблене рішення демонструє високу гнучкість і може бути масштабоване для додаткових функцій у майбутньому.

**Особистий внесок здобувача.** У процесі виконання роботи здобувач самостійно здійснив аналіз ринку, обрав технології для розробки додатку, створив функціонал і провів тестування.

Результати кваліфікаційної роботи можуть бути використані для ознайомлення з процесом та методами розробки мобільних додатків, а також для ознаймоленням із їх архітектурою.

# РОЗДІЛ 1
# ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

## 1.1. Традиційні інструменти управління подіями

До появи сучасних цифрових інструментів управління подіями використовувались традиційні методи, які мали свої переваги та недоліки. Вони базувалися на ручній праці та простих інструментах, які з часом поступилися місцем більш інноваційним рішенням. Організатори подій ретельно фіксували всі деталі вручну, що вимагало значної точності та уваги. Планери дозволяли зручно організовувати розклад, проте будь-які зміни потребували фізичного редагування, що ускладнювало процес у разі великої кількості змін чи оновлень. Візуально це були барвисті або суворі книжки, заповнені сторінками з датами та місцем для записів.

Окреме місце посідали дошки для записів та оголошень (рис. 1.1). У багатьох офісах і організаціях встановлювали магнітні або коркові дошки, на яких кріпили нотатки, списки завдань і розклад подій. Цей інструмент був зручним для командної роботи, оскільки дозволяв учасникам швидко побачити оновлення та зрушення в планах. Проте така система залежала від фізичної присутності людей і не могла забезпечити віддалену взаємодію.

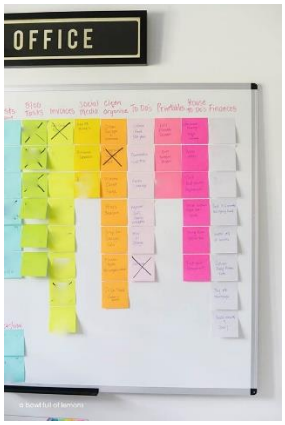



Рис. 1.1. Планування подій на дошці в офісі

З розвитком комп'ютерних технологій з’явилися електронні таблиці та текстові документи, такі як Microsoft Excel (рис. 1.2) і Word. Ці інструменти стали революційним кроком вперед у порівнянні з паперовими носіями, дозволяючи зберігати більші обсяги інформації, робити розрахунки, створювати списки учасників і бюджети. Електронні таблиці використовувалися для планування графіків та відстеження статусу завдань. Але через відсутність спеціалізованих функцій управління подіями і неможливість синхронізації з іншими системами, ці інструменти часто використовувалися лише як доповнення до інших методів.

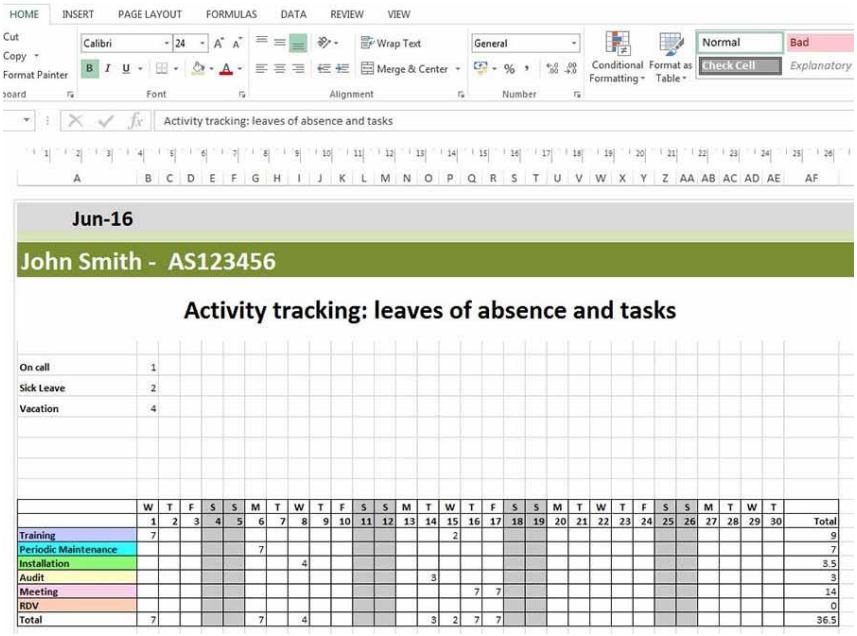



Рис. 1.2. Приклад таблиці в Microsoft Excel

Для комунікації застосовувалися телефонні дзвінки та електронна пошта. Дзвінки дозволяли швидко вирішувати нагальні питання і були важливим інструментом для координації дій, особливо в ситуаціях, де потрібна була оперативність. Електронна пошта стала основним засобом розповсюдження запрошень і підтверджень участі, що значно полегшило процес комунікації, але створило нові виклики, такі як управління великим обсягом повідомлень та відстеження відповідей.

Фізичні зустрічі та наради залишалися обов'язковими елементами для обговорення деталей і прийняття рішень (рис. 1.3). Організатори подій збиралися на особисті зустрічі, щоб розподілити ролі, затвердити програми та обговорити можливі ризики. Такий підхід сприяв ефективній комунікації та забезпечував особистий контакт, який підвищував рівень довіри між учасниками. Однак він також мав свої обмеження, особливо коли події охоплювали велику кількість людей з різних локацій.

Рис. 1.3. Фізична зустріч людей для розподілу ролей

### 1.2. Вплив технологій на управління подіями

Розвиток технологій докорінно змінив підходи до управління подіями, зробивши їх значно зручнішими та ефективнішими. У минулому організація

подій вимагала багато ручної роботи: ведення паперових записів, безперервні телефонні дзвінки та розсилання запрошень поштою. Зараз ці методи поступилися місцем автоматизованим рішенням, які знижують ризики помилок і оптимізують усі етапи планування.

Використання сучасних технологій дозволяє організаторам подій краще управляти часом і ресурсами. Онлайн-календарі та програми для організації завдань, такі як Google Calendar або Microsoft Outlook, дозволяють синхронізувати заходи з особистими і робочими графіками (рис. 1.4). Це забезпечує інтеграцію різних елементів планування в одному середовищі, спрощуючи координацію і мінімізуючи ймовірність конфліктів.

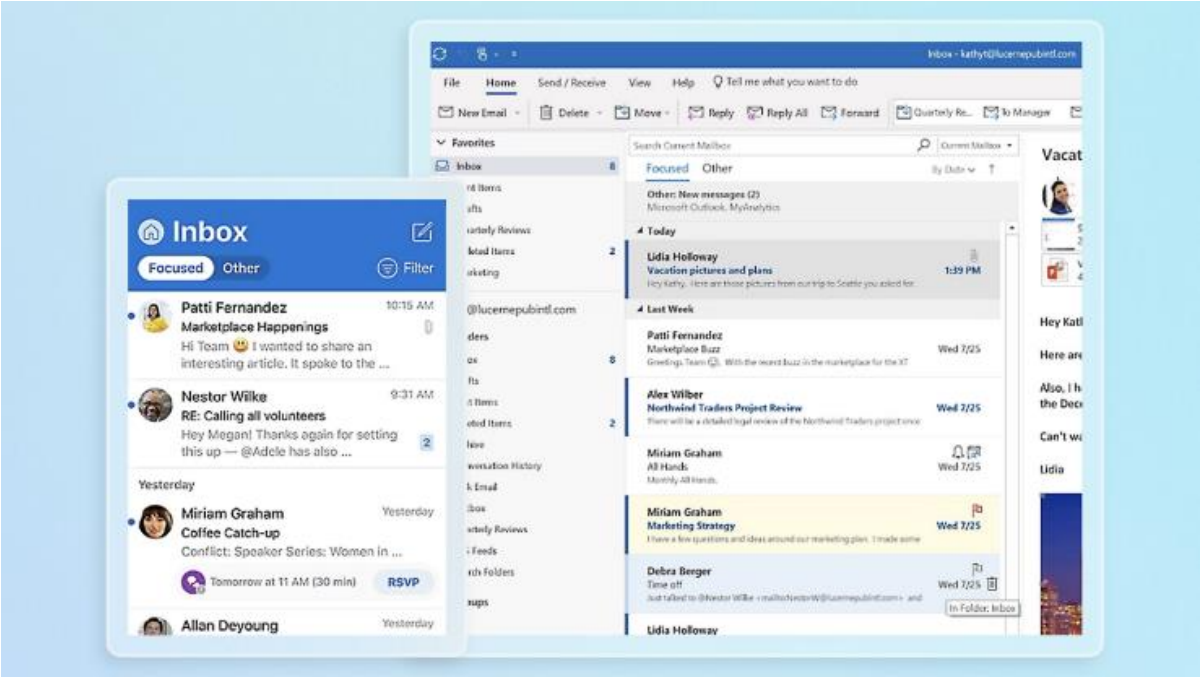



Рис. 1.4. Організація завдань з Microsoft Outlook

Додатково технології змінюють взаємодію з учасниками подій. Віртуальні платформи, такі як Zoom або Microsoft Teams, дали змогу проводити заходи онлайн, скорочуючи витрати на транспорт і логістику. Це стало особливо актуальним під час пандемії, коли традиційні зустрічі були неможливими. Онлайн-інструменти також дозволяють миттєво поширювати інформацію про зміни у

розкладі, надсилати нагадування та отримувати зворотний зв'язок, що значно підвищує оперативність і точність.

Ще однією перевагою є можливість використання аналітики. Сучасні технології дозволяють збирати дані про кількість відвідувачів, їхню активність і вподобання. Наприклад, додатки для продажу квитків, такі як Eventbrite, надають організаторам докладну статистику про аудиторію. Це дає змогу краще планувати наступні заходи, адаптувати їх під потреби учасників і підвищувати їхню залученість.

Варто зазначити, що інтеграція технологій у процес управління подіями також полегшила роботу невеликих організацій або індивідуальних користувачів. Завдяки доступним і простим у використанні мобільним додаткам, таким як Asana або Trello, кожен може ефективно організувати навіть особисті події. Такі рішення надають широкий набір функцій: створення завдань, відстеження їхньо-го прогресу, встановлення дедлайнів і навіть надсилання автоматичних нагаду-вань.

Технології також відкрили доступ до глобальної аудиторії. Наприклад, використання соціальних мереж, таких як Facebook чи Instagram, дозволяє не тільки просувати події, а й взаємодіяти з потенційними учасниками через коментарі, опитування чи рекламу (рис. 1.5). Це особливо важливо для залучення нової аудиторії та створення інтерактивного середовища.

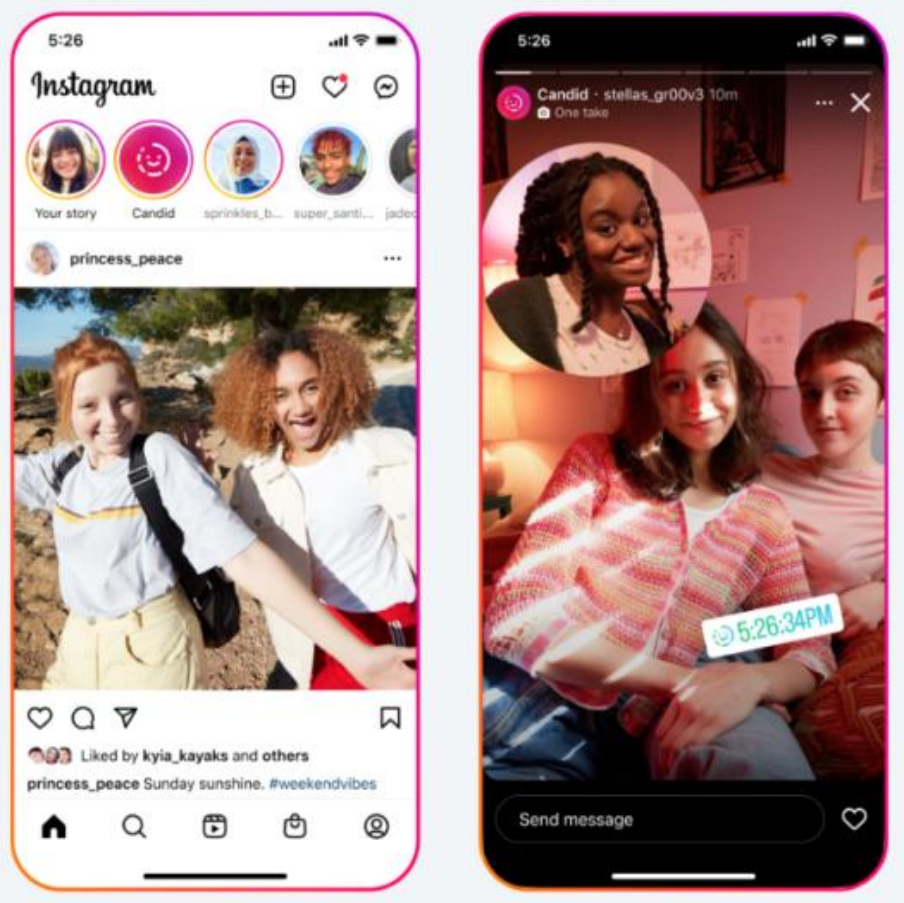



Рис. 1.5. Соціальна мережа Instagram

### 1.3. Сучасні інструменти управління подіями

Сучасні інструменти управління подіями пройшли значний шлях розвитку завдяки технологічним інноваціям, які змінили підходи до планування, організації та координації заходів. Ці інструменти інтегрують численні функції, полегшуючи роботу організаторів та підвищуючи ефективність комунікації і взаємодії з учасниками.

**Програмне забезпечення для управління проєктами** (наприклад, Trello, Asana, Jira) стало основою для координації завдань у рамках організації подій (рис. 1.6). Ці інструменти дозволяють розбивати великі проєкти на менші завдання, розподіляти обов’язки серед членів команди, відстежувати прогрес і дотримуватися дедлайнів. Візуальні елементи, такі як дошки, списки та календарі, допомагають візуалізувати процес роботи і надають можливість швидко вносити зміни.


Рис. 1.6. Програмне забезпечення Trello

**Системи управління подіями** (EMS) розроблені для комплексного контролю над усіма аспектами події. Вони охоплюють реєстрацію учасників, продаж квитків, управління списками відвідувачів, комунікацію з учасниками, а також звітність та аналіз після проведення заходу. Популярні системи, як-от Eventbrite (рис. 1.7), Cvent та Bizzabo, надають широкі можливості для автоматизації різних процесів та інтеграції з іншими платформами.

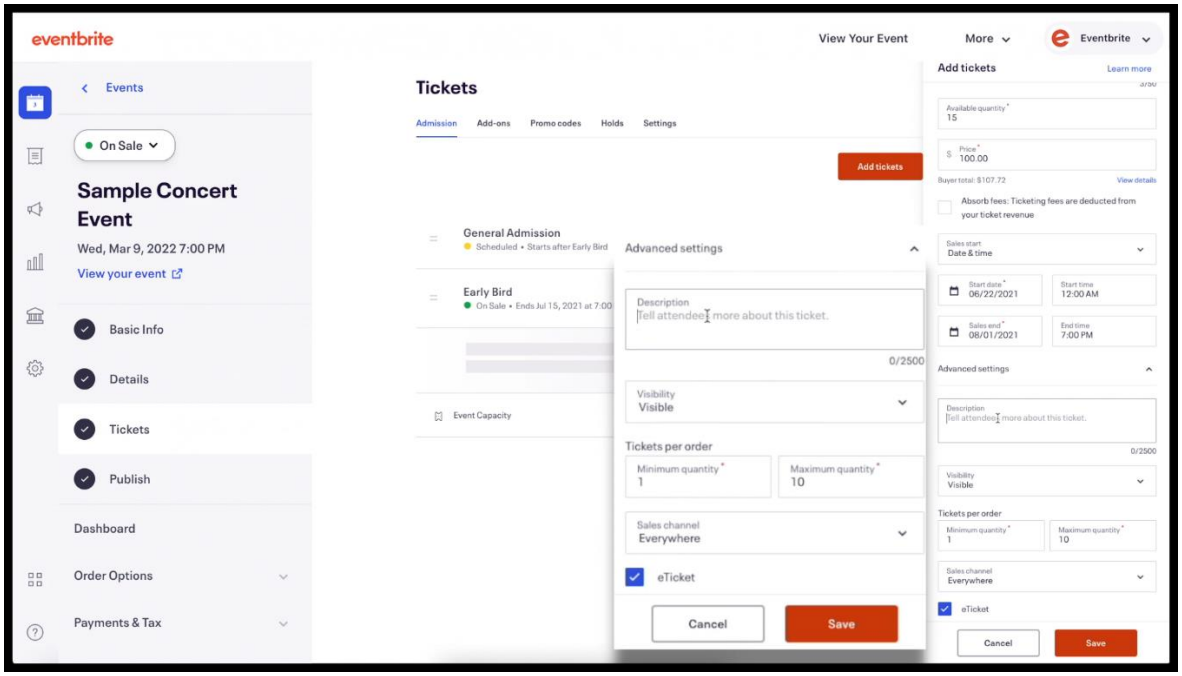



Рис. 1.7. Детальне управління подією в системі Eventbrite

### 1.4. Роль інтреграції інструментів управління подіями

Інтеграція інструментів управління подіями стала одним із ключових аспектів сучасного підходу до організації заходів (рис. 1.8). Вона дозволяє об’єднувати різні платформи, сервіси та додатки в єдину екосистему, забезпечуючи безшовну взаємодію між усіма етапами підготовки, проведення та аналізу події.

Рис. 1.8. Інтеграція різних інструментів

Сучасні події, навіть найпростіші, складаються з багатьох компонентів: розробка розкладу, запрошення учасників, управління реєстрацією, продаж квитків, обробка зворотного зв’язку, аналіз результатів тощо. Кожен із цих процесів вимагає інструментів для ефективного виконання. Інтеграція дозволяє забезпечити їхню взаємодію, скорочуючи час і ресурси, необхідні для окремого управління кожним елементом.

Наприклад, інтеграція систем реєстрації учасників із CRM-платформами, такими як Salesforce або HubSpot, забезпечує автоматичне збереження даних про учасників події. Це дозволяє відстежувати взаємодії з кожним клієнтом, налаштовувати персоналізовані запрошення та будувати довгострокові

стосунки. Такий підхід не лише підвищує зручність для організаторів, але й створює більш персоналізований досвід для учасників.

Ще одним прикладом є інтеграція інструментів продажу квитків, таких як Eventbrite або Ticketmaster, із платформами для соціального медіа. Це дозволяє організаторам просувати заходи безпосередньо в соцмережах, надаючи користувачам можливість миттєво купувати квитки або реєструватися на подію, не виходячи з додатку. Така інтеграція полегшує доступ до заходу для аудиторії, збільшуючи шанси на високий рівень участі.

Крім того, інтегровані системи управління подіями можуть включати аналітичні інструменти. Наприклад, платформи на кшталт Cvent дозволяють об'єднувати інформацію з різних джерел – реєстрацій, опитувань, даних із соціальних медіа – і представляти їх у зручній формі для аналізу. Це допомагає організаторам отримувати цінні інсайти, які можна використовувати для оптимізації майбутніх заходів.

Інтеграція також відіграє важливу роль у забезпеченні ефективної комунікації під час заходу. Наприклад, інтеграція між платформами для відеоконференцій (Zoom, Microsoft Teams) і календарями (Google Calendar, Outlook) дозволяє автоматично створювати події з посиланнями на віртуальні кімнати. Це мінімізує ризик помилок і полегшує процес планування для учасників.

Для великих подій інтеграція інструментів управління може включати інте-грацію з фізичними пристроями, такими як сканери квитків або системи навігації на місці проведення заходу. Це забезпечує більш зручний і швидкий доступ до послуг, а також покращує загальний досвід учасників.

### 1.5. Проблеми існуючих інструментів управління подіями

Існуючі інструменти управління подіями, незважаючи на їхню популярність, мають низку проблем, які ускладнюють їх ефективне використання для організації особистих і професійних завдань. Ці проблеми є наслідком недоліків у дизайні, обмежень функціональності та невідповідності сучасним технологічним і соціальним вимогам.

- **Складність і перевантаженість інтерфейсу.** Багато інструментів, як програмного, так і апаратного забезпечення, мають складний інтерфейс. Це особливо стосується старіших рішень або продуктів, орієнтованих на корпоративний сегмент. Недостатня інтуїтивність інтерфейсу змушує користувачів витрачати багато часу на навчання, що знижує загальну продуктивність.

- **Відсутність адаптивності.** Більшість інструментів не враховує індивідуальних потреб користувачів. Налаштування часто обмежені, що робить ці рішення універсальними, але не персоналізованими. Користувачі не можуть ефективно адаптувати інструменти під свої унікальні вимоги, такі як складність подій, частота повторень або пріоритетність задач.

- **Відсутність інтреграції.** Інструменти управління подіями часто не синхронізуються з іншими платформами та сервісами, такими як електронна пошта, календарі чи системи обміну повідомленнями. Це ускладнює координацію, оскільки користувачі змушені дублювати події або вручну переносити дані між різними інструментами.

- **Обмежений доступ офлайн.** Багато рішень залежить від постійного підключення до Інтернету. У випадках, коли зв'язок відсутній або нестабільний, доступ до інформації про події стає неможливим, що створює ризики для користувачів, які потребують надійного планування навіть у віддалених місцях.

- **Недостатня автоматизація нагадувань.** Часто інструменти обмежуються базовими функціями нагадувань, такими як прості сповіщення перед початком події. Відсутність гнучкості у виборі часу, способу сповіщення або їхньої частоти ускладнює процес планування для користувачів, які працюють із великими обсягами завдань.

- **Застарілі технології.** Частина інструментів управління подіями створювалася на базі технологій, які на сьогодні вважаються застарілими. Це ускладнює інтеграцію із сучасними платформами, уповільнює роботу

інструментів і призводить до невідповідності очікуванням користувачів, які звикли до швидких та інноваційних рішень.

- **Проблеми безпеки.** Деякі інструменти управління подіями мають слабкі місця у сфері захисту даних. Це стає критичним питанням, особливо коли йдеться про конфіденційні події або інформацію, що має значення для організацій. Проблеми безпеки можуть включати як витік даних, так і незахищеність при обміні інформацією.

### 1.6. Мобільні додатки як інноваційне рішення

Мобільні додатки стали невід'ємною частиною повсякденного життя, особливо для тих, хто активно планує свої заходи та організовує події для себе. Ці додатки забезпечують інноваційний підхід до самостійного управління подіями, роблячи процес більш зручним і ефективним.

Однією з головних переваг використання мобільних додатків для особистого планування є їхня доступність і зручність. Завдяки їм користувач має можливість створювати, редагувати та відстежувати події у будь-який час і з будь-якого місця. Це допомагає швидко адаптувати плани відповідно до зміни обставин та залишатися організованим без зайвих зусиль.

Мобільні додатки надають індивідуальні рішення, адаптовані до потреб кожного користувача. Завдяки можливості налаштування нагадувань, інтеграції з календарями та персоналізованих сповіщень, користувач може отримувати точну інформацію про свої плани та події, що дозволяє уникати забування важливих моментів.

Інноваційність таких додатків проявляється і в їхній здатності інтегруватися з іншими технологіями, такими як голосові помічники та системи на базі штучного інтелекту. Це дозволяє автоматизувати багато рутинних завдань, наприклад, створення нагадувань за допомогою голосових команд або отримання пропозицій щодо оптимізації розкладу на основі минулих дій та пріоритетів.

Мобільні додатки також надають можливість швидко і зручно збирати і переглядати інформацію про майбутні події. Інтерактивні функції дозволяють

отримувати доступ до заміток, зображень та інших матеріалів, які можуть бути корисними під час планування. Це робить процес підготовки до заходів більш організованим і легким.

### 1.7. Постановка завдання

У ході роботи над дипломним проєктом було запропоновано створити сучасний мобільний додаток для оптимізованого управління подіями. Основна мета — спроектувати мобільний додаток для управління подіями, що використовує Jetpack Compose для створення сучасного та ефективного інтерфейсу користувача. Основні завдання, які необхідно виконати в процесі розробки:

- розробити архітектуру та інтерфейс користувача для мобільного додатку, з можливістю створення і редагування подій та реєстрації користувачів;
- забезпечити інтеграцію технологій для оптимізації управління подіями, таких як автоматичне сповіщення користувачів про події, створення, видалення та редагування подій;
- забезпечити інтеграцію з сервісом Google Calendar, щоб синхронізувати його події з подіями мобільного додатку;
- забезпечити ефективну обробку даних та відклик додатку, застосовуючи принципи кешування та офлайн синхронізації для прискорення його роботи;
- протестувати роботу мобільного додатку для перевірки надійності функціонування основних можливостей та коректності взаємодії компонентів додатку.

### 1.8. Висновки до розділу 1

У розділі було детально розглянуто інструменти управління подіями, зокрема традиційні та сучасні рішення, а також роль мобільних додатків як інноваційного підходу для планування.

Традиційні інструменти, такі як паперові календарі та електронні таблиці,

хоча й залишаються популярними, мають свої обмеження, зокрема в плані гнучкості, доступності в реальному часі та інтеграції з іншими системами. Вони часто не можуть забезпечити високий рівень взаємодії та автоматизації, що є важливим для ефективного управління подіями у сучасному світі. Такі інструменти зазвичай вимагають багато часу для організації та не надають персоналізованого досвіду.

З іншого боку, сучасні інструменти для управління подіями, зокрема програмне забезпечення для управління проєктами, системи для реєстрації учасників та мобільні додатки, значно підвищили ефективність організації подій. Ці платформи інтегрують різноманітні функції, дозволяючи організаторам оперативно керувати завданнями, фінансами та комунікацією, а також взаємодіяти з учасниками в реальному часі.

Особливо важливою складовою є використання мобільних додатків для управління подіями. Вони дозволяють користувачам організовувати події, що робить процес планування більш зручним та персоналізованим. Мобільні додатки забезпечують доступність, інтерактивність і автоматизацію, дозволяючи зберігати інформацію про події, отримувати нагадування і легко вносити зміни у розклад. Це дає користувачам можливість ефективно управляти своїм часом та досягати поставлених цілей без зайвих зусиль.

Таким чином, розглянуті інструменти та технології підтверджують важливість використання сучасних підходів до управління подіями, що сприяє значному підвищенню ефективності та якості організації заходів. Це створює підґрунтя для подальшого розвитку мобільних додатків та інтеграції новітніх технологій у цю сферу.

# РОЗДІЛ 2
# ПРОГРАМНА РЕАЛІЗАЦІЯ МОБІЛЬНОГО ДОДАТКУ

## 2.1. Вибір та обґрунтування інструментів проектування

2.1.1 Середовище розробки

Для реалізації даного проєкту було обрано Android Studio, оскільки це офіційне середовище розробки для створення додатків на платформі Android, яке забезпечує потужний набір інструментів для ефективної розробки, тестування та налагодження програмного забезпечення (рис. 2.1). Однією з головних переваг є інтеграція з усіма компонентами Android, що дозволяє створювати додатки, максимально оптимізовані для мобільних пристроїв.

Android Studio підтримує основні мови програмування для розробки під Android — Java та Kotlin. Kotlin, зокрема, надає сучасні можливості для написання чистого, легкого для розуміння коду, що робить його дедалі популярнішим серед розробників. Крім того, середовище розробки пропонує зручний редактор коду з підсвіткою синтаксису, автоматичним доповненням та інструментами для рефакторингу, що допомагає значно знизити ймовірність помилок і підвищити продуктивність програміста.

Android Studio також підтримує створення графічних інтерфейсів користувача (UI) через дизайн-редактор, який дозволяє візуально компонувати елементи на екрані і бачити зміни в реальному часі. Це спрощує процес створення адаптивного інтерфейсу для різних розмірів екрану та типів пристроїв, забезпечуючи кращу сумісність і зручність використання додатка.

Однією з найбільших переваг є можливість інтеграції з різноманітними інструментами Google, такими як Firebase для зберігання даних, автентифікації користувачів, аналітики та відправки сповіщень. Це дозволяє знижувати витрати часу на розробку додаткових функцій і фокусуватися на основних задачах.

Вбудовані інструменти для тестування дозволяють проводити швидкі та ефективні юніт-тести, а також інтеграційні тести для перевірки правильності роботи додатка на різних типах пристроїв за допомогою емуляторів Android або фізичних пристроїв. Крім того, Android Studio дозволяє створювати звіти про помилки та аналізувати продуктивність програми, що важливо для оптимізації та усунення проблем на ранніх етапах розробки.

Завдяки підтримці різних версій Android та зручному управлінню залежностями через Gradle, Android Studio є ідеальним вибором для створення стабільних і масштабованих мобільних додатків, що сприяє зменшенню часу на розробку та тестування і забезпечує високу якість кінцевого продукту. [2]

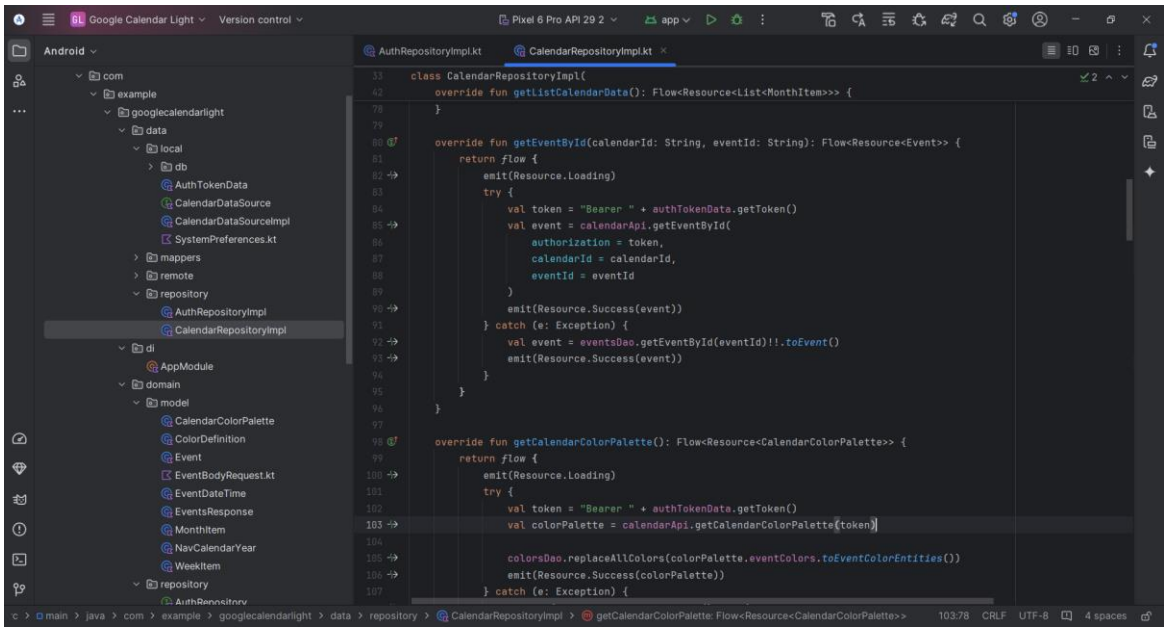



Рис. 2.1. Середовище розробки Android Studio

### 2.1.2 Мова програмування Kotlin

Для реалізації цього проєкту було обрано мову програмування Kotlin. Цей вибір зумовлений рядом переваг, які Kotlin надає для розробки додатків на платформі Android. Kotlin є офіційно підтримуваною мовою для Android-розробки, що означає, що вона оптимізована для роботи з Android SDK та екосистемою Android Studio.

Однією з основних переваг Kotlin є його компактність та виразність (рис. 2.2). Мова дозволяє значно скоротити обсяг коду порівняно з Java, що підвищує читабельність і зменшує ймовірність помилок. Завдяки підтримці функціонального програмування та лаконічним синтаксичним конструкціям, розробники можуть реалізовувати складну логіку в коротких і зрозумілих рядках коду.

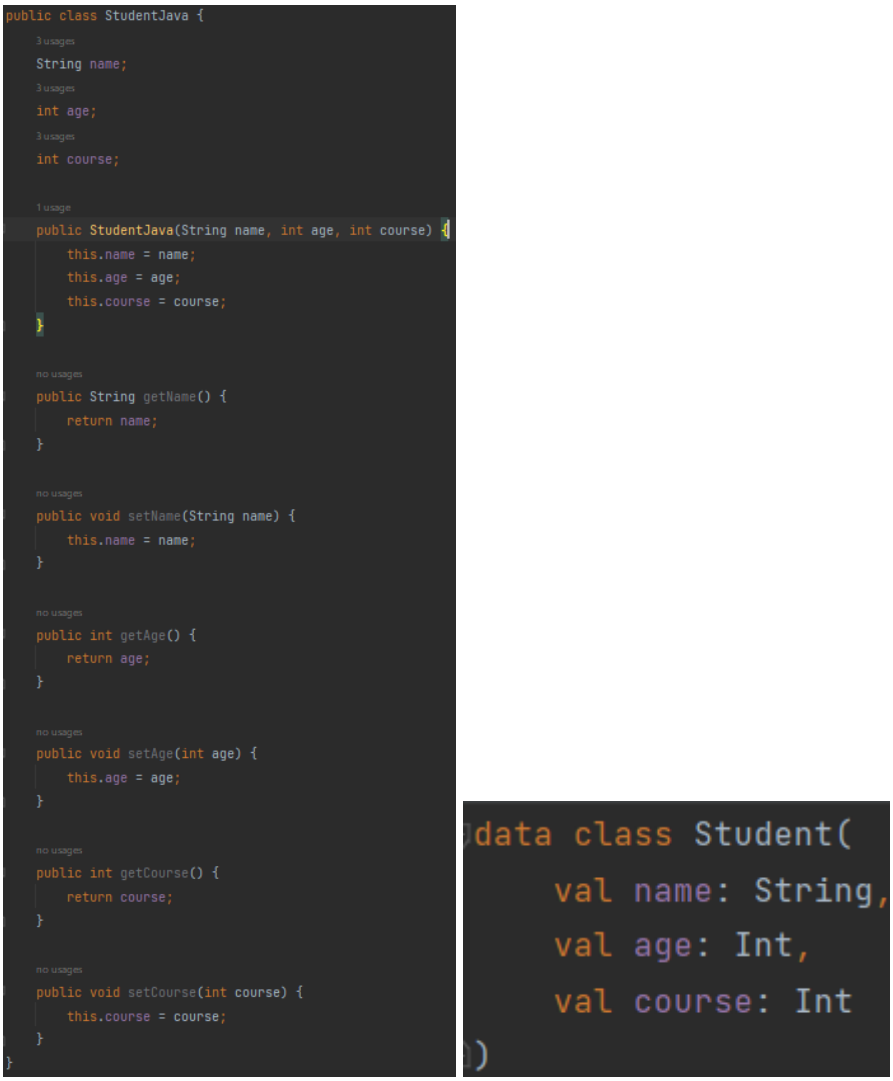



Рис. 2.2. Клас Student мовою Java (ліворуч) та мовою Kotlin (праворуч)

Kotlin також надає безпечну роботу з null-значеннями, що значно знижує ймовірність виникнення помилок типу NullPointerException, що є однією з найпоширеніших причин збоїв у програмах (рис. 2.3). Мова має вбудовані механізми для роботи з nullable типами, що дозволяє ефективно обробляти

ситуації, де значення можуть бути відсутні.

```
data class Student(
    val name: String,
    val age: Int,
    val course: Int,
    val scholarship: Int?
)
```

Рис. 2.3. Поле scholarship, яке може містити null-значення

Ще однією важливою перевагою Kotlin є сумісність з Java (рис. 2.4). Kotlin можна без проблем інтегрувати в проєкти, що вже використовують Java, що дозволяє поступово переходити на нову мову без необхідності повної переписування коду. Це особливо корисно для підтримки старих проєктів або при необхідності взаємодії з уже існуючими бібліотеками та фреймворками, написаними на Java.

```
fun main(args: Array<String>) {

    val student = StudentJava( name: "Ivan", age: 22, course: 5)
    student.course = 6
}
```

Рис. 2.4. Сумісність мови Kotlin з Java

Мова Kotlin також активно підтримує розширення функцій, що дозволяє додавати нові можливості до існуючих класів без необхідності їх зміни (рис. 2.5). Це робить код більш гнучким та розширюваним.

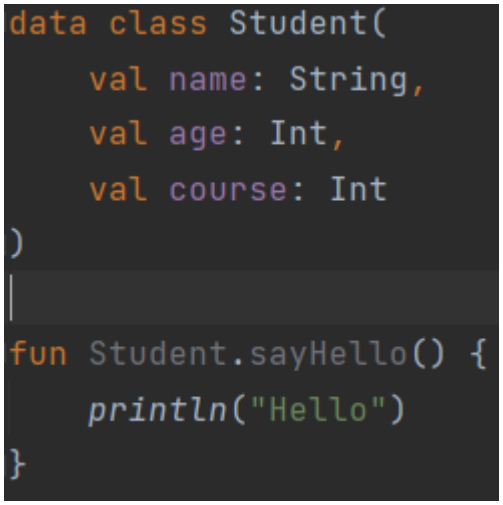



Рис. 2.5. Функція розширення sayHello

Нарешті, Kotlin підтримує сучасні функціональні можливості, такі як корутини, що дозволяють ефективно працювати з асинхронними операціями та багатозадачністю, що є важливим аспектом при розробці мобільних додатків, де потрібно зберігати високу продуктивність при обробці запитів до серверів, баз даних або виконанні інших тривалих операцій (рис. 2.6). [1]

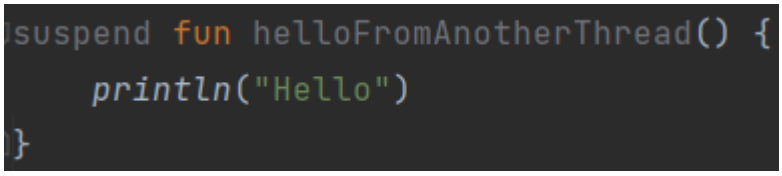



Рис. 2.6. Приклад асинхронної функції з ключовим словом suspend

### 2.1.3. ViewModel

ViewModel — це один із компонентів бібліотеки Android Jetpack, який використовується для управління даними, що необхідні для UI (інтерфейсу користувача), і їх збереження під час змін конфігурації, таких як поворот екрану.

Основні особливості ViewModel:

- Збереження даних під час змін конфігурації. Дані, що зберігаються у ViewModel, залишаються доступними навіть після знищення й повторного створення активності або фрагмента (наприклад, при зміні орієнтації екрана).

- Розділення логіки та UI. ViewModel допомагає відокремити бізнес-логіку від коду, що відповідає за відображення. Це забезпечує кращу підтримку коду, модульність і можливість повторного використання.
- Життєвий цикл ViewModel ViewModel прив'язаний до життєвого циклу власника (активності або фрагмента) й існує до моменту їх остаточного знищення.

### 2.1.4. Retrofit 2

Retrofit 2 — це бібліотека для Android, яка спрощує процес здійснення HTTP-запитів та взаємодії з вебсервісами. Вона дозволяє легко інтегруватися з RESTful API, що робить її популярним вибором серед розробників мобільних додатків. Основною метою Retrofit 2 є спрощення роботи з API через перетворення HTTP-запитів у методи Kotlin, використовуючи прості анотації.

Retrofit 2 працює на основі концепції перетворення API у форму інтерфейсу. Розробник визначає інтерфейс з описом HTTP-запитів (GET, POST, PUT, DELETE тощо), вказуючи кінцеві точки API та типи запитів. Бібліотека автоматично перетворює цей інтерфейс у реальні HTTP-виклики під час виконання програми, роблячи взаємодію з API простою та зрозумілою.

Однією з головних особливостей Retrofit 2 є її підтримка об’єктно-орієнтованого програмування та типів даних, зокрема для обробки JSON або XML відповідей API. Для цього бібліотека використовує конвертери (наприклад, GSON, Moshi), що дозволяють автоматично перетворювати дані, отримані у вигляді JSON, у Kotlin-об’єкти та навпаки. Це значно зменшує кількість ручної роботи для розробника та покращує читабельність коду.

Retrofit 2 підтримує також асинхронні операції, що дозволяє уникнути блокування основного потоку (UI) під час виконання довготривалих операцій, 13 таких як звернення до сервера. Завдяки цьому, додатки працюють більш плавно, а користувачі не стикаються з проблемами зависання інтерфейсу під час очікування на відповідь від серверу. Бібліотека використовує виклики на основі «зворотних викликів» (callbacks) для обробки відповідей від сервера.

Крім того, Retrofit 2 забезпечує гнучку конфігурацію та розширюваність. Вона дозволяє додавати різні інтерсептори для обробки HTTP-запитів та відповідей, що корисно для додавання логування або керування помилками. Наприклад, можна легко реалізувати обробку неочікуваних помилок, таких як помилки мережі або серверні збої, забезпечуючи надійність додатку. [5]

### 2.1.5. Kotlin Coroutines

Kotlin Coroutines — це потужний інструмент для керування асинхронними операціями в мові програмування Kotlin. Coroutines дозволяють виконувати операції, що потребують тривалого часу (наприклад, звернення до мережі, читання файлів, взаємодія з базами даних) без блокування головного потоку програми. Вони забезпечують простіший спосіб написання асинхронного коду, дозволяючи уникати складного й громіздкого управління потоками, яке часто зустрічається в традиційній багатопоточності.

Корутина — це легковаговий потокоподібний об’єкт, що дозволяє зупиняти та відновлювати виконання на певному етапі. На відміну від потоків (threads), корутини використовують менше ресурсів та можуть виконуватися одночасно на одному потоці без створення окремих системних потоків. Це робить їх ефективним способом для виконання асинхронних завдань.

До основних переваг корутин можна віднести:

- Вбудована підтримка скасування корутин: скасування відбувається автоматично через поточну ієрархію корутин.
- Менше випадків витоку пам’яті в додатку: використовується структурований паралелізм для виконання операцій у межах області.
- Легковажність: можна запускати багато корутин в одному потоці завдяки підтримці призупинення, яка не блокує потік, у якому 14 запущена корутина. Призупинення заощаджує пам’ять замість блокування, одночасно підтримуючи багато одночасних операцій.
- Інтеграція з Jetpack бібліотеками: багато Jetpack бібліотек включають розширення, які надають повну підтримку корутин. Деякі бібліотеки та кож

надають їхню власну область виконання корутин, яку можна використовувати дла структурованої багатопоточності. [3]

### 2.1.6. Room

Програми, які обробляють великі обсяги структурованих даних, можуть отримати значну користь від збереження цих даних локально. Найпоширенішим випадком використання є кешування відповідних фрагментів даних, щоб, коли пристрій не може отримати доступ до мережі, користувач усе ще міг переглядати їх у режимі офлайн.

Бібліотека стійкості Room забезпечує рівень абстракції над SQLite, щоб забезпечити вільний доступ до бази даних, одночасно використовуючи всю потужність SQLite.

Зокрема, Room надає такі переваги:

- Перевірка SQL-запитів під час компіляції.
- Зручні анотації, які зводять до мінімуму повторюваний і схильний до помилок шаблонний код.
- Оптимізовані шляхи міграції бази даних.

З огляду на ці міркування розробникам наполегливо рекомендується використовувати бібліотеку Room замість безпосереднього використання SQLite API.

Основні компоненти бібліотеки Room (рис. 2.7):

- Клас бази даних, який містить базу даних і служить основною точкою доступу для підключення до збережених даних додатку.
- Сутності даних, які представляють таблиці в базі даних додатку.
- Об’єкти доступу до даних (DAO), які надають методи, які додаток може використовувати для запиту, оновлення, вставки та видалення даних у базі даних.

Клас бази даних надає додатку екземпляри DAO, пов’язані з цією базою даних. У свою чергу, програма може використовувати DAO для отримання даних

із бази даних як екземплярів пов’язаних об’єктів сутності даних. Додаток також може використовувати визначені сутності даних для оновлення рядків із відповідних таблиць або для створення нових рядків для вставки. [6]

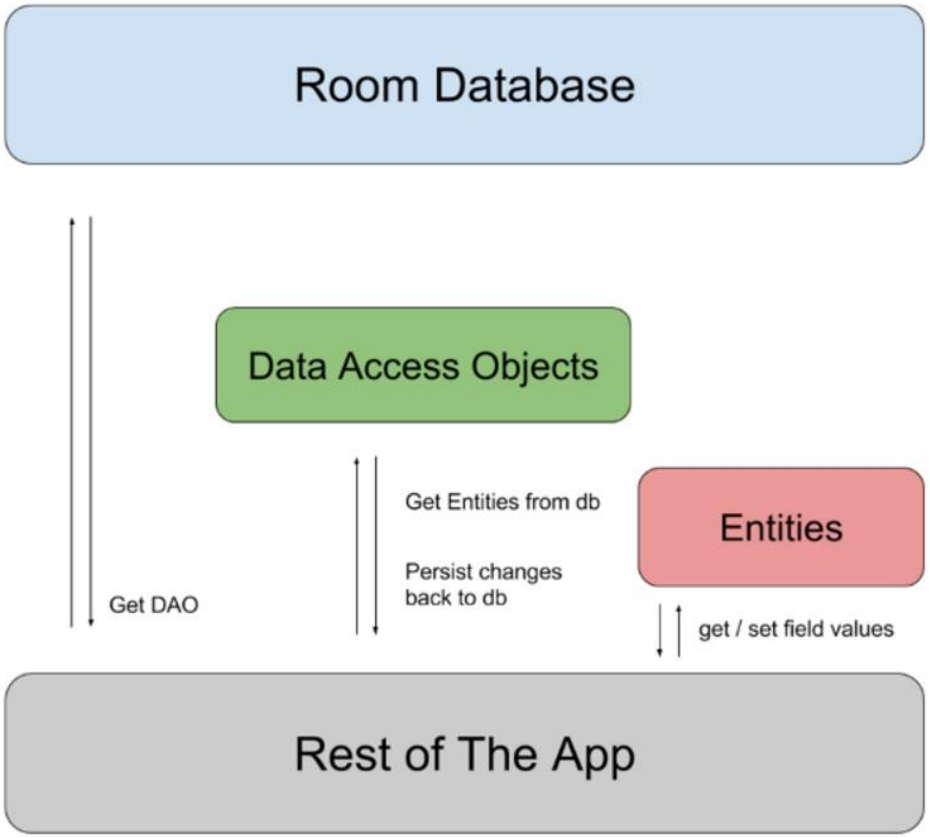



Рис. 2.7. Діаграма архітектури бібліотеки Room

2.1.7. Dagger Hilt

Dagger Hilt — це бібліотека для впровадження залежностей (Dependency Injection, DI) у додатках Android, розроблена на основі популярного DI-фреймворку Dagger. Hilt спрощує інтеграцію та використання Dagger в Android-додатках, зменшуючи складність конфігурації, яку розробники зазвичай зустрічають, використовуючи Dagger самостійно. Основною метою Hilt є автоматизація процесу керування залежностями в Android-додатках, що забезпечує зручний спосіб роботи з залежностями на рівні класів, активностей та фрагментів.

Залежності — це об’єкти, які клас використовує для виконання своєї роботи. Dagger Hilt допомагає керувати залежностями, створюючи їх автоматично і передаючи їх туди, де вони потрібні, таким чином спрощуючи процес створення та підтримки додатків.

Основні переваги використання Dagger Hilt:

- Спрощення конфігурації DI в Android: Hilt надає прості анотації для роботи з Android-компонентами, такими як Activity, Fragment, ViewModel, що полегшує налаштування DI в порівнянні з ручною конфігурацією Dagger.
- Автоматичне управління життєвим циклом: Hilt створює граф залежностей для кожного Android-компонента (наприклад, для Activity або Fragment), а також забезпечує автоматичне очищення та знищення залежностей після завершення життєвого циклу компонента. Це допомагає уникнути витоків пам’яті.
- Зменшення кількості шаблонного коду: Hilt автоматично генерує код, який відповідає за ініціалізацію та впровадження залежностей, зменшуючи кількість ручних налаштувань, таких як модулі та компоненти Dagger.
- Сумісність із іншими Android API: Hilt без проблем інтегрується з іншими інструментами та бібліотеками для Android, такими як Jetpack Navigation, WorkManager, ViewModel та інші. [4]

### 2.1.8. Jetpack Compose

Jetpack Compose — це сучасний фреймворк для створення інтерфейсу користувача (UI) в Android (рис. 2.8). Він є частиною набору інструментів Jetpack від Google, і головною його особливістю є декларативний підхід до побудови UI, який суттєво відрізняється від традиційного способу створення інтерфейсу через XML.

Основні особливості Jetpack Compose:

- Декларативний підхід: У Jetpack Compose UI будується декларативно, що означає, що розробник описує, яким має бути інтерфейс користувача для поточного стану даних, а фреймворк автоматично оновлює його, коли дані

змінюються. Це відрізняється від імперативного підходу в XML, де розробник мав самостійно керувати оновленнями елементів UI.

- Менше коду: Завдяки декларативному підходу, Jetpack Compose дозволяє створювати UI з меншою кількістю коду, ніж традиційний XML. Це спрощує роботу розробників та робить код зрозумілішим.
- Краща інтеграція з Kotlin: Compose повністю написаний на Kotlin і використовує всі можливості цієї мови, включаючи лямбда-вирази, функції як об’єкти першого класу, розширення функцій та корутини. Це дозволяє створювати більш гнучкий та зрозумілий код, який краще взаємодіє з логікою програми.
- Простота оновлень: У традиційному підході з XML, оновлення інтерфейсу може бути складним та вимагати великої кількості додаткового коду для відстеження змін у стані програми. У Compose, завдяки реактивному підходу, інтерфейс автоматично оновлюється, коли змінюється стан.
- Менше шаблонного коду: У старому методі потрібно було багато шаблонного коду для зв’язування XML-розмітки з логікою через Java або Kotlin. Compose усуває цю необхідність, оскільки UI і логіка тісно інтегровані через Kotlin.
- Гнучкість і розширюваність: Jetpack Compose дозволяє створювати кастомні компоненти набагато простіше, ніж це було в XML, де потрібно було створювати окремі класи для кастомних View. У Compose можна легко створювати свої UI-компоненти як звичайні функції.
- Швидкість розробки: Завдяки попередньому перегляду в Android Studio можна бачити зміни в UI в реальному часі без необхідності постійно компілювати весь додаток. Це суттєво пришвидшує цикл розробки та тестування.
- Краща підтримка анімацій: Compose пропонує вбудовані засоби для створення анімацій, які є більш інтуїтивними та простими у використанні в порівнянні з традиційними XML-анімаціями.[8]

```
@Composable
fun CenterText() {
    Box(
        modifier = Modifier.fillMaxSize(),
        contentAlignment = Alignment.Center
    ) {
        Text(
            text = "Hello World!",
            color = Color.Black
        )
    }
}
```

Рис. 2.8. Найпростіший приклад використання Jetpack Compose для відображення тексту посередині екрану

### 2.1.9. Coil

Coil (Coroutine Image Loader) — це легка і сучасна бібліотека для завантаження та кешування зображень в Android. Вона створена спеціально для Kotlin і використовує корутини для асинхронного завантаження зображень, що робить її швидкою, зручною і ефективною в порівнянні з іншими популярними бібліотеками, такими як Glide чи Picasso.

Чому бібліотека Coil краща ніж інші:

- Інтеграція з Kotlin: Coil написана на Kotlin і використовує всі його можливості, включаючи корутини, розширювані функції та інші сучасні концепції, що робить код більш зрозумілим і лаконічним.
- Асинхронне завантаження зображень: Coil використовує корутини для асинхронного завантаження зображень, що дозволяє завантажувати зображення у фоновому режимі без блокування основного потоку. Це покращує продуктивність і зменшує затримки під час завантаження.

- Маленький розмір: Coil — це одна з найменших бібліотек для завантаження зображень. Вона додає лише близько 200 КБ до розміру APK, що значно менше у порівнянні з Glide чи Picasso.
- Вбудована підтримка кешування: Coil автоматично кешує зображення на диску і в пам’яті, що покращує продуктивність програми, оскільки зображення не потрібно завантажувати з мережі щоразу, коли воно повторно використовується.
- Сумісність з Jetpack Compose: Coil підтримує Jetpack Compose, що дозволяє легко інтегрувати завантаження зображень у декларативний UI Compose. Це робить його чудовим вибором для сучасних Android-додатків, що використовують Compose.
- Ефективне управління ресурсами: Завдяки використанню низькорівневих API Android, таких як BitmapPool, Coil економно використовує пам’ять і ресурси, що робить його ефективним для обробки великої кількості зображень.
- Простота у використанні: Інтерфейс Coil спроектований таким чином, щоб бути максимально простим у використанні. Ви можете завантажити зображення в ImageView або будь-який інший View буквально кількома рядками коду.
- Підтримка трансформацій: Coil дозволяє легко застосовувати трансформації до зображень, такі як обрізка, зміна розміру або накладення ефектів.
- Гнучкість налаштувань: Coil пропонує широкий спектр налаштувань, які дозволяють тонко налаштувати процес завантаження зображень, наприклад, встановити власні параметри кешування, логування або обробки помилок. [9]

### 2.2. Опис файлової структури мобільного додатку

Оскільки мій проєкт написаний з використанням чистої архітектури, з MVVM в якості UI архітектури, в нього має бути в певній мірі чітко визначена структура з розбиттям на різні “шари”, які знаходяться в різних папках і містять

свої файли.

В моєму випадку це:

**data/** – директорія для шару даних, який відповідає за їх отримання, збереження і т.д;

- local/ - папка з локальними даними додатку;
  - db/ - папка для локальної бази даних;
    - EventColorEntity.kt – клас, який уособлює таблицю в локальній базі даних, яка містить кольори подій;
    - EventEntity.kt - клас, який уособлює таблицю в локальній базі даних, яка містить події;
    - EventsDao.kt – інтерфейс, який містить функції для операцій з подіями;
    - ColorsDao.kt - інтерфейс, який містить функції для операцій з кольорами подій;
    - EventDatabase – клас бази даних;
  - AuthTokenData.kt – клас, який містить функції для операцій з токеном авторизації користувача;
  - SystemPreferences.kt – клас, який містить функції для CRUD операцій пов’язаних з простими типами даних необхідних для додатку;
  - CalendarDataSource.kt – інтерфейс, який містить функції, які відповідають за отримання даних необхідних для побудови календаря;
  - CalendarDataSourceImpl.kt – реалізація інтерфейсу CalendarDataSource.kt;
- remote/ - папка з даними з серверу;
  - CalendarApi.kt – інтерфейс, який містить функції для отримання даних з Google Calendar Api.
- mappers/ - папка для файлів, які відповідають за перетворення моделей шару даних в моделі шару бізнес логіки і навпаки;
  - EventColorsMappers.kt – мапери для кольорів подій;
  - EventMappers.kt – мапери для подій;

- repository/ - папка для класів з реалізаціями інтерфейсів репозиторіїв;
  - CalendarRepositoryImpl.kt – клас, який реалізовує інтерфейс репозиторію CalendarRepositoty.kt;

**di/** – директорія для ін’єкції залежностей;

- AppModule.kt – клас, який містить функції, які відповідають за створення об’єктів необхідних як залежності для інших класів в додатку;

**domain/** - директорія для бізнес-логіки додатку;

- model/ - папка для моделей бізнес-логіки;
  - CalendarColorPalette.kt – модель для палітри кольорів календаря;
  - ColorDefinition.kt – модель для пари кольорів для переднього і заднього фону;
  - Event.kt – модель для події;
  - EventBodyRequest.kt – модель тіла PUT та UPDATE запитів до API;
  - EventDateTime.kt – модель для дати та часу події;
  - EventsResponse.kt – модель, яка представляє відповідь від API при запиті певного списку подій;
  - MonthItem.kt – модель, яка містить інформацію про місяць даних в календарі;
  - NavCalendarYear.kt – модель, яка містить інформацію про рік даних в навігаційному календарі;
  - WeekItem.kt – модель, з інформацією про дати початку та кінця тижня.
- repository/ - папка, яка містить інтерфейси репозиторііів;
  - CalendarRepository.kt – інтерфейс, який містить функції для CRUD операцій з календарем.
- use-case/ - папка, яка містить класи, які виконують одну певну дію в додатку (додати подію, видалити подію і т.д.)
  - DeleteEventUseCase.kt – клас, який відповідає за видалення події;

- GetCalendarColorPalette.kt – клас, який відповідає за отримання палітри кольорів для додатку;
- GetEventByIdUseCase.kt – клас, який відповідає за отримання події за її унікальним ідентифікатором;
- GetListCalendarDataUseCase.kt – клас, який відповідає за отримання даних для побудови календаря;
- GetMonthDatesUseCase.kt – клас, який відповідає за отримання дат місяця як списку чисел;
- GetNavCalendarMonthsUseCase.kt – клас, який відповідає за отримання списку місяців для навігаційного календаря;
- InsertEventUseCase.kt – клас, який відповідає за додавання нової події;
- SynchronizeEventsUseCase.kt – клас, який відповідає за офлайн синхронізацію подій в додатку;
- UpdateEventUseCase.kt – клас, який відповідає за оновленння події;

**ui/** - директорія для коду пов’язаного з графічним інтерфейсом додатку;

- screens/ - папка з екранами додатку;
  - add_or_update_event/ - папка для екрану додавання або редагування події;
    - AddOrUpdateEventScreen.kt – файл з кодом для побудови екрану для додавання або редагування події;
    - AddOrUpdateEventViewModel.kt – клас, який містить стан екрану для додавання або редагування події та функції необхідні для нього;
  - auth/ - папка для екрану авторизації користувача;
    - AuthScreen.kt – файл з кодом для побудови екрану авторизації користувача;
    - AuthViewModel.kt – клас, який містить стан екрану для авторизації користувача та функції необхідні для нього;

- calendar/ - папка для екрану з календарем;
    - CalendarScreen.kt – файл з кодом для побудови екрану календаря;
    - CalendarViewModel.kt – клас, який містить стан екрану календаря та функції необхідні для нього;
- event_details/ - папка для екрану з деталями події;
    - EventDetailsScreen.kt – файл з кодом для побудови екрану деталей події;
    - EventDetailsViewModel.kt – клас, який містить стан екрану деталей події та функції необхідні для нього;

- theme/ - папка для теми додатку;
    - Color.kt – файл з кольорами для теми додатку;
    - Theme.kt – файл з темою додатку;
    - Type.kt – файл з шрифтами для теми;

**util/** – директорія для допоміжних класів;

- connectivity_observer/ - папка для класів, які відповідають за відслідковування стану підключення до інтернету;
    - ConnectivityObserver.kt – інтерфейс з функціями для відслідковування стану підключення до інтернету;
    - NetworkConnectivityObserver.kt – реалізація інтерфейсу ConnectivityObserver;
- Constants.kt – клас з константами додатку;
- DateTimeParser.kt – клас для перетворення дати в форматі тексту в об’єкт;
- Resource.kt – клас-обгортка, який використовується для отримання відподі від серверу з трьома можливими станами – Loading, Error, Success.

**CalendarApplication.kt** – клас додатку;

**MainActivity.kt** – стартова точка запуску додатку.

## 2.3. Архітектура мобільного додатку

Мобільний додаток був спроектований і реалізований на чистій архітектурі, яка характеризується розділенням структури додатку на шари. Data-шар відповідає за отримання, збереження і загалом всі інші операції з даними потрібними додатку. Domain-шар містить бізнес-логіку додатку. Presentation-шар поділяється на MVVM архітектуру і містить код побудови графічного інтерфейсу додатку. DI-шар містить функції, які повертають екземпляри об’єктів і надають їх класам по всьому додатку, які їх потребують.

### 2.3.1. Data-шар

Data-шар містить джерела даних, мапери та реалізацію інтерфейсів репозиторіїв.

Віддаленим джерелом даних в додатку являється Google Calendar Api (рис. 2.9). Він містить різні функції для отримання, додавання, редагування та видалення даних. В моєму випадку це дані про кольорову палітру календаря та дані про публічні події та події авторизованого користувача.

```
@GET("colors")
suspend fun getCalendarColorPalette(
    @Header("Authorization") authorization: String
): CalendarColorPalette

@POST("calendars/{calendarId}/events")
suspend fun insertEvent(
    @Header("Authorization") authorization: String,
    @Path("calendarId") calendarId: String,
    @Body eventBody: EventBodyRequest
): Event

@PUT("calendars/{calendarId}/events/{eventId}")
suspend fun updateEvent(
    @Header("Authorization") authorization: String,
    @Path("calendarId") calendarId: String,
    @Path("eventId") eventId: String,
    @Body eventBody: EventBodyRequest
): Event
```

Рис. 2.9. Деякі з функцій в Calendar Api

Локальне джерело даних поділяється на локальну базу даних та класи, які відповідають за збереження та отримання певних даних примітивних типів,

наприклад, інформацію про те, чи вперше був запущений додаток, токен авторизації користувача і т.д. Локальна база даних має свою архітектуру і поділяється на Entity, Dao та  Database.

Entity – це таблиця в базі даних представлена у вигляді класу на мові програмування Kotlin, але з деякими необхідними анотаціями (рис. 2.10).

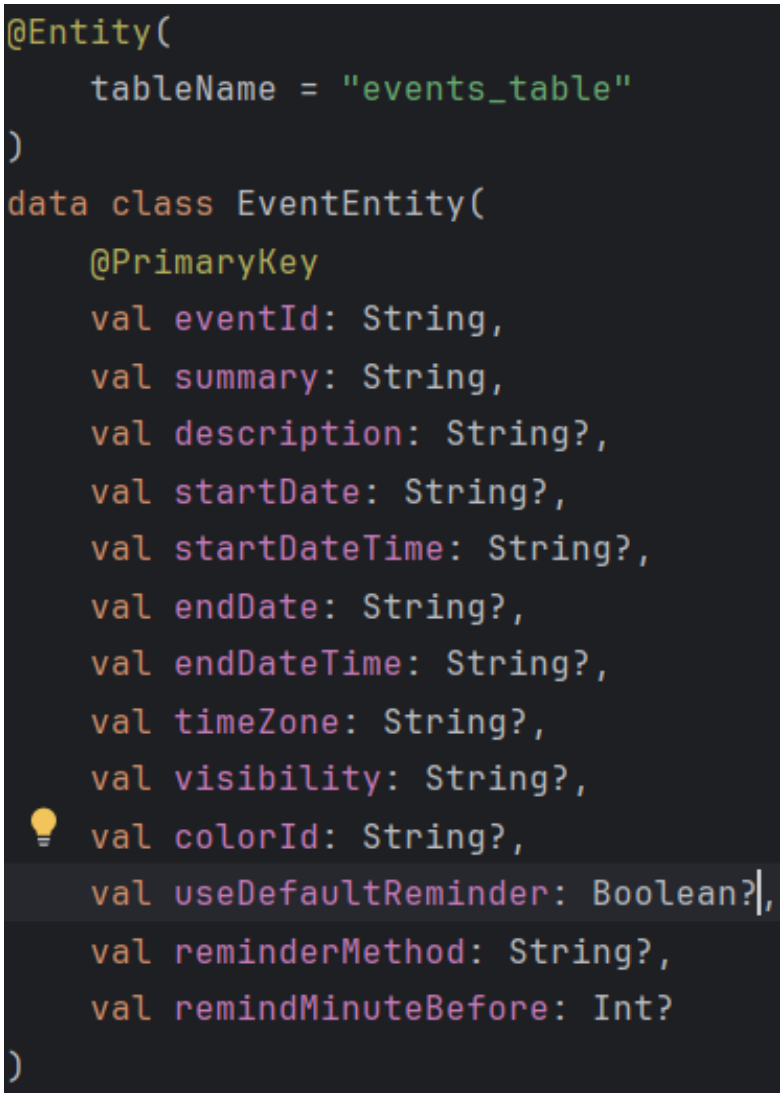



Рис. 2.10. Клас, який уособлює таблицю подій в базі даних

Dao (Data Access Object) – це інтерфейс, який містить функції для операцій з даними бази даних (рис. 2.11).

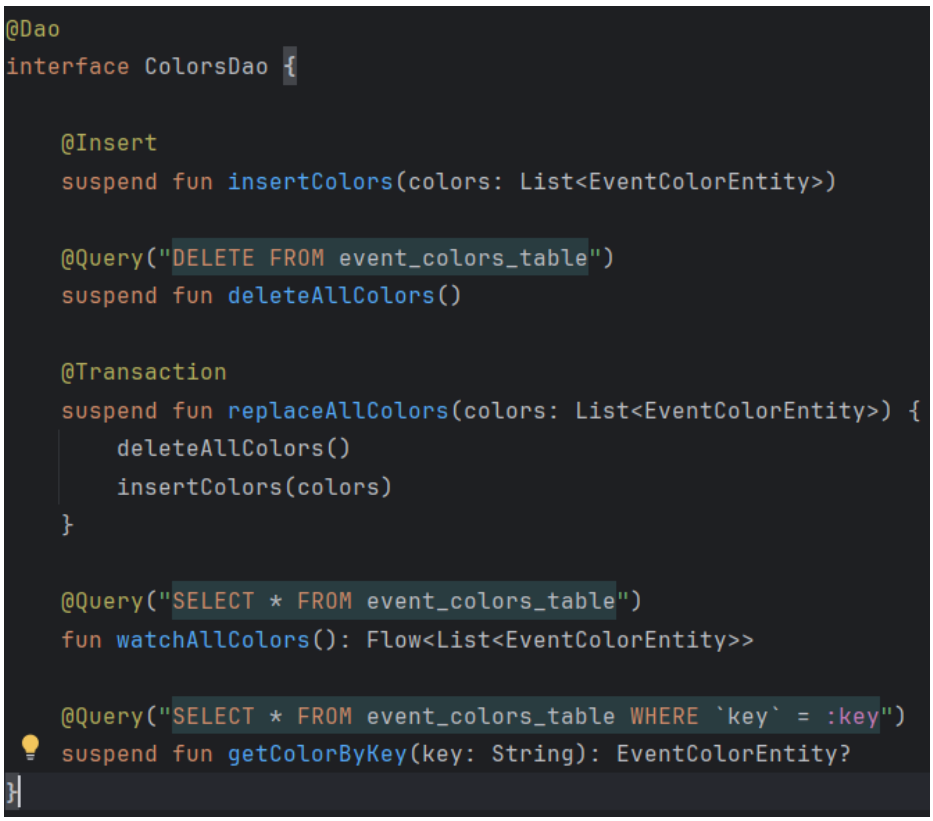



Рис. 2.11. Dao для операцій з таблицей кольорів в базі даних

Database – це абстрактний клас, який має наслідуватися від класу RoomDatabase та містити інформацію про таблиці бази даних, її об’єкти доступу та версію (рис. 2.12).

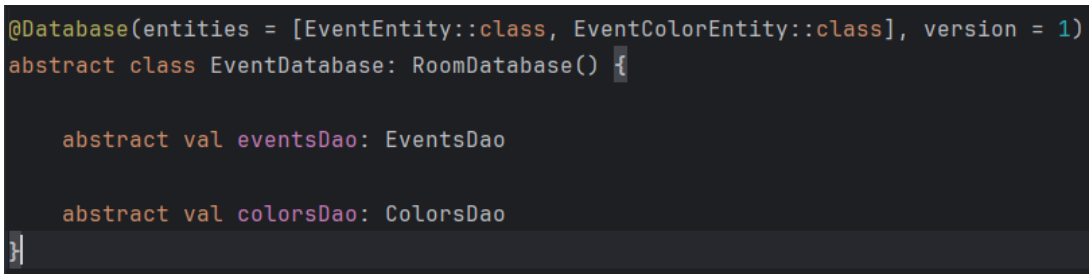



Рис. 2.12. Клас бази даних

Клас, який містить функції для збереження примітивних типів даних використовує так звані SharedPreferences (рис. 2.13). SharedPreferences – це такий “файлик”, в який можна помістити дані ключ-значення. Ключ має бути текстового формату, а значення може бути будь-яким примітивним типом даних.

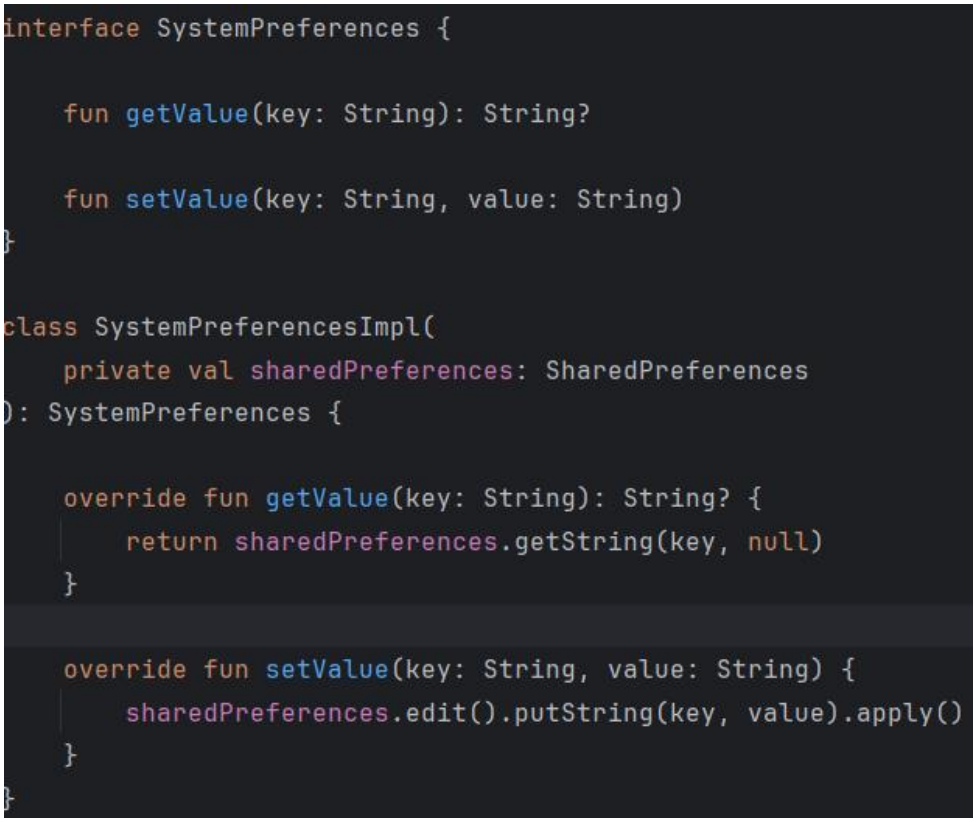



Рис. 2.13. Клас для збереження даних примітивного типу

Мапери це класи або функції, які відповідають за перетворення моделей шару даних в моделі domain-шару. В застосуонку це так звані функції розширення (рис. 2.14).


Рис. 2.14. Функція розширення для перетворення моделі даних

В додатку лише один репозиторій, який відповідає за всі можливі операції з даними. Приклад функції, яка повертає подію за її унікальним ідентифікатором, зображено на рис. 2.15.

```
override fun getEventById(calendarId: String, eventId: String): Flow<Resource<Event>> {
    return flow {
        emit(Resource.Loading)
        try {
            val token = "Bearer " + authTokenData.getToken()
            val event = calendarApi.getEventById(
                authorization = token,
                calendarId = calendarId,
                eventId = eventId
            )
            emit(Resource.Success(event))
        } catch (e: Exception) {
            val event = eventsDao.getEventById(eventId)!!.toEvent()
            emit(Resource.Success(event))
        }
    }
}
```

Рис. 2.15. Приклад функції з репозиторію

2.3.2. Domain-шар

Domain-шар містить моделі даних, інтерфейси репозиторіїв та випадки використання.

Моделі даних в domain-шарі це класи, які представляють собою складні об’єкти і містять лише змінні (рис. 2.16).

```
@Parcelize
data class Event(
    @SerializedName("id")
    val eventId: String? = null,
    val summary: String? = null,
    val description: String? = null,
    val start: EventDateTime? = null,
    val end: EventDateTime? = null,
    val visibility: String? = null,
    val colorId: String? = null,
    val reminders: Reminders? = null
): Parcelable
```

Рис. 2.16. Модель події

Інтерфейс репозиторія містить функції для доступу до даних з шару даних додатку (рис. 2.17).

```
interface CalendarRepository {

    fun getListCalendarData(): Flow<Resource<List<MonthItem>>>

    suspend fun getNavCalendarMonths(): List<LocalDate>

    fun getEventById(calendarId: String, eventId: String): Flow<Resource<Event>>

    fun getCalendarColorPalette(): Flow<Resource<CalendarColorPalette>>

    fun insertEvent(eventBody: EventBodyRequest): Flow<Resource<Event>>

    fun deleteEvent(eventId: String): Flow<Resource<Unit>>

    fun updateEvent(eventId: String, eventBody: EventBodyRequest): Flow<Resource<Event>>

    fun synchronizeEvents(): Flow<Resource<Unit>>
}
```

Рис. 2.17. Інтерфейс репозиторія

Ключовим словом suspend позначаються ті функції, які можуть призупиняти поток виконання і мають виконуватися в області виконання корутини.

Корутина – це легковаговий поток, яка виконується в контексті звичайного потоку і в одному потоці може бути безліч корутин, які виконуються незалежно одна від одної.

Інші функції, які не позначені ключовим словом suspend в якості значення на виході мають flow. Flow – це потік даних, який може містити різний набір даних, тобто, якщо звичайна функція може повернути лише одне значення, то flow може містити декілька значень. Ці значення можуть бути отримані лише в області виконання корутини, тому ці функції теж не призупиняють поток виконання і виконуються асинхронно.

Use case або випадок використання – це клас який містить лише одну функцію, яка виконує лише одна поставлену задачу і як правило це класи які

викликають функції репозиторіїв. Перевизначивши оператор invoke в класі ці класи можна викликати як функції. Приклади таких класів зображено на рис. 2.18 та на рис. 2.19.

```
class GetMonthDatesUseCase @Inject constructor() {

    operator fun invoke(year: Int, month: Int): List<LocalDate?> {
        val daysInMonth = YearMonth.of(year, month).lengthOfMonth()
        val firstDayOfMonth = LocalDate.of(year, month, dayOfMonth: 1)
        val firstDayOfWeek = firstDayOfMonth.withDayOfMonth( dayOfMonth: 1).dayOfWeek.value
        val totalDays = 42

        // Create a list to hold the dates
        val dates = MutableList(totalDays) { null as LocalDate? }

        // Fill the list with dates for the given month
        for (i in 0 ≤ until < daysInMonth) {
            val date = firstDayOfMonth.plusDays(i.toLong())
            val position = (firstDayOfWeek + i - 1) % totalDays
            dates[position] = date
        }

        return dates
    }
}
```

Рис. 2.18. Випадок використання, який повертає список дат місяця

```
class GetListCalendarDataUseCase @Inject constructor(
    private val calendarRepository: CalendarRepository
) {

    operator fun invoke(): Flow<Resource<List<MonthItem>>> {
        return calendarRepository.getListCalendarData()
    }
}
```

Рис. 2.19. Випадок використання, який повертає список місяців для календаря, але просто викликає функцію репозиторію

### 2.3.3. Presentation-шар

Presentation-шар містить екрани додатку та представляє собою архітектуру MVVM (Model-View-ViewModel), де Model це випадки виконання, View це екран

додатку, а ViewModel – клас, який поєднує між собою View та Model і містить стан екрану, функції необхідні для нього та захищає дані від змін конфігурації додатку. Також до шару графічного інтерфейсу відноситься тема додатку, яка складається з усіх необхідних кольорів і шрифтів.

Додаток для оптимізованого управління подіями містить 4 екрани – екран авторизації користувача, екран додавання, або редагування події, екран деталей події та екран календаря.

Весь графічний інтерфейс побудований на Jetpack Compose. Цей фреймворк спрощує і прискорює процес побудови екранів додатку (рис. 2.20), зменшує повторюваність коду та вірогідність витоків пам'яті.

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun DatePickerModal(
    onDateSelected: (Long?) -> Unit,
    onDismiss: () -> Unit
) {
    val datePickerState = rememberDatePickerState()

    DatePickerDialog(
        onDismissRequest = onDismiss,
        confirmButton = {
            TextButton(onClick = {
                onDateSelected(datePickerState.selectedDateMillis)
                onDismiss()
            }) {
                Text( text: "OK")
            }
        },
        dismissButton = {
            TextButton(onClick = onDismiss) {
                Text( text: "Cancel")
            }
        }
    ) {
        DatePicker(state = datePickerState)
    }
}
```

Рис. 2.20. Функція побудови на Jetpack Compose, яка показує на екрані вікно з вибором дати та часу

До кожного екрану відноситься свій ViewModel-клас. ViewModel містить стан екрану, який отримується екраном на основі реактивного підходу. Стан екрану представлений у вигляді різного типу Flow (рис. 2.21). Flow - це потік даних, які, наприклад, екран може відслідковувати. Як тільки нові дані потрапляють в потік, екран відразу їх отримує і оновлюється на їх основі без необхідності оновлення екрану вручну.

```
private val _titleText = MutableStateFlow( value: "")
val titleText = _titleText.asStateFlow()

private val _isAllDayEvent = MutableStateFlow( value: false)
val isAllDayEvent = _isAllDayEvent.asStateFlow()

private val _eventStartDate = MutableStateFlow(LocalDateTime.now())
val eventStartDate = _eventStartDate.asStateFlow()

private val _eventEndDate = MutableStateFlow(LocalDateTime.now())
val eventEndDate = _eventEndDate.asStateFlow()

private val _eventStartTime = MutableStateFlow(getRoundedTime())
val eventStartTime = _eventStartTime.asStateFlow()

private val _eventEndTime = MutableStateFlow(getRoundedTime().plusHours( hours: 1).truncatedTo(ChronoUnit.MINUTES))
val eventEndTime = _eventEndTime.asStateFlow()

private val _showDatePicker = MutableStateFlow( value: false)
val showDatePicker = _showDatePicker.asStateFlow()

private val _showTimePicker = MutableStateFlow( value: false)
val showTimePicker = _showTimePicker.asStateFlow()

private val _eventNotification = MutableStateFlow<NotificationTime?>( value: null)
val eventNotification = _eventNotification.asStateFlow()
```

Рис. 2.21. Стан екрану представлений у вигляді Flow

ViewModel також містить необхідні функції, які викликає екран (рис. 2.22). Функції надають нові значення Flow і в свою чергу оновлюють екран додатку.

```
fun selectDate(date: Long) {
    val dateTime = LocalDateTime.ofInstant(Instant.ofEpochMilli(date), ZoneId.systemDefault())
    selectingDate?.let { value ->
        when(value) {
            EventDate.Start -> {
                if(dateTime >= LocalDateTime.now()) {
                    _eventStartDate.value = dateTime
                }
                if(eventStartDate.value > eventEndDate.value) {
                    _eventEndDate.value = eventStartDate.value
                }
            }
            EventDate.End -> {
                if(dateTime >= LocalDateTime.now()) {
                    _eventEndDate.value = dateTime
                }
                if(eventStartDate.value > eventEndDate.value) {
                    _eventStartDate.value = dateTime
                }
            }
        }
    }
    selectingDate = null
}
```

Рис. 2.22. Функція в ViewModel, яка відповідає за вибір дати

### 2.3.4. DI-шар

DI-шар містить різні модулі з функціями для створення екземплярів класів для подальшої їх ін’єкції в класи, які їх потребують. В застосунку лише один модуль, який створює екземпляри, які існують впродовж всієї роботи додатку. Для цього модуль потрібно позначити необхідною анотацією (рис. 2.23).

```
@Module
@InstallIn(SingletonComponent::class)
object AppModule {
```

Рис. 2.23. Модуль ін’єкції залежностей

Кожна функція в модулі теж має бути позначена своїми анотаціями, які будуть визначати поведінку створення екземплярів. Кожен екземпляр класу є єдиним екземпляром даного класу на весь додаток (рис. 2.24).

```
@Provides
@Singleton
fun provideCalendarApi(): CalendarApi {
    val interceptor = HttpLoggingInterceptor()
    interceptor.setLevel(HttpLoggingInterceptor.Level.BODY)
    val client = OkHttpClient.Builder().addInterceptor(interceptor).build()

    val gson = GsonBuilder().setDateFormat("yyyy-MM-dd'T'HH:mm:ss").create()
    return Retrofit.Builder()
        .baseUrl(BASE_URL)
        .client(client)
        .addConverterFactory(GsonConverterFactory.create(gson))
        .build()
        .create()
}
```

Рис. 2.24. Функція створення екземпляру інтерфейсу CalendarApi

2.3.5. Папка util/

В даній папці знаходяться класи з допоміжним функціоналом.

Клас, який відповідає за відслідковування статусу підключення до інтернету зображено на рис. 2.25.

```
class NetworkConnectivityObserver(
    private val context: Context
): ConnectivityObserver {

    private val connectivityManager =
        context.getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager

    override fun observe(): Flow<ConnectivityObserver.Status> {
        return callbackFlow {
            val callback = object : ConnectivityManager.NetworkCallback() {
                override fun onAvailable(network: Network) {
                    super.onAvailable(network)
                    launch { send(ConnectivityObserver.Status.Available) }
                }

                override fun onLosing(network: Network, maxMsToLive: Int) {
                    super.onLosing(network, maxMsToLive)
                    launch { send(ConnectivityObserver.Status.Losing) }
                }

                override fun onLost(network: Network) {
                    super.onLost(network)
                    launch { send(ConnectivityObserver.Status.Lost) }
                }

                override fun onUnavailable() {
                    super.onUnavailable()
                    launch { send(ConnectivityObserver.Status.Unavailable) }
                }
            }

            connectivityManager.registerDefaultNetworkCallback(callback)
            awaitClose {
                connectivityManager.unregisterNetworkCallback(callback)
            }
        }.distinctUntilChanged()
    }
}
```

Рис. 2.25. Клас для віслідковування статусу підключення до інтернету

Клас-об’єкт з константами додатку (рис. 2.26). За допомогою ключового слова Object, ми можемо звертатися до полів класу як до статичних, тобто не через екземпляр класу, а через його назву.

```
object Constants {

    const val PREF_ACCESS_TOKEN = "access token"

    const val PREF_FIRST_LAUNCH = "first launch"

    const val BASE_URL = "https://www.googleapis.com/calendar/v3/"
}
```

Рис. 2.26. Клас з константами

Клас-об’єкт DateTimeParser, який містить лише одну функцію для перетворення дати з текстового формату у формат об’єкта (рис. 2.27).

```
object DateTimeParser {

    fun parse(dateTime: String): LocalDateTime {
        return if(dateTime.contains( other: "+")) {
            LocalDateTime.parse(dateTime, DateTimeFormatter.ISO_OFFSET_DATE_TIME)
        } else {
            LocalDateTime.parse(dateTime, DateTimeFormatter.ofPattern( pattern: "yyyy-MM-dd'T'HH:mm:ss"))
        }
    }
}
```

Рис. 2.27. Клас для перетворення дати

Клас Resource, який містить різні стани відповіді з серверу – Loading, Error та Success (рис. 2.28).

```
sealed class Resource<out T : Any> {
    data class Success<out T : Any>(val data: T) : Resource<T>()
    data class Error(val exception: Exception) : Resource<Nothing>()
    data object Loading : Resource<Nothing>()
}
```

Рис. 2.28. Клас Resource

2.3.6. MainActivity

MainActivity це стартова точка запуску додатку. Тут міститься опис навігаційних маршрутів між екранами додатку, а також інший необхідний код.

Приклад маршруту навігації в додатку зображено на рис. 2.29.

```
composable<CalendarScreen> {
    val viewModel: CalendarViewModel = hiltViewModel()

    LaunchedEffect(key1 = status) {
        if(status == ConnectivityObserver.Status.Available) {
            synchronizeEventsUseCase().collect { result ->
                if(result is Resource.Success) {
                    viewModel.getOrUpdateListCalendarData()
                }
            }
        }
    }

    CalendarScreen(
        viewModel = viewModel,
        navController = navController,
        onEventClick = { eventId, calendarId ->
            navController.navigate(EventDetails(eventId, calendarId))
        },
        onSignOut = {
            lifecycleScope.launch {
                googleAuthUiClient.signOut()
                navController.popBackStack()
            }
        }
    )
}
```

Рис. 2.29. Опис екрану календаря в маршрутах навігації

## 2.4. Опис та реалізація основного функціоналу

2.4.1. Авторизація користувача

Авторизація користувача відбувається через сервіс Google, тому що в додатку присутня інтеграція з Google Calendar.

Всі потрібні функції для авторизації знаходяться в одному класі – GoogleAuthUiClient. В ньому також присутні дві необхідні змінні – googleSignInOptions та googleSignInClient (рис. 2.30).

```
private val googleSignInOptions = GoogleSignInOptions.Builder(GoogleSignInOptions.DEFAULT_SIGN_IN)
    .requestEmail()
    .requestScopes(Scope(CalendarScopes.CALENDAR))
    .build()

private val googleSignInClient = GoogleSignIn.getClient(context, googleSignInOptions)
```

Рис. 2.30. Змінні класу GoogleAuthUiClient

При першому запуску додатку користувачу необхідно пройти авторизацію. Для цього спочатку потрібно сформувати intent. В android розробці intent — це один із ключових компонентів, що дозволяє взаємодію між різними частинами додатка або між додатками. Його можна уявити як “намір” виконати певну дію, наприклад, запустити активність, відкрити веб-сторінку або надіслати повідомлення. За допомогою функції buildSignInIntent можна його сформувати.

```
fun buildSignInIntent(): Intent {
    return googleSignInClient.signInIntent
}
```

Рис. 2.31. Функція buildSignInIntent

Далі функцію buildSignInIntent (рис. 2.31) треба викликати на екрані авторизації, передавши її параметром для іншої функції (рис. 2.32), щоб відкрити вікно вибору Google аккаунтів користувача. Після вибору аккаунту користувачем, функція поверне нам інший intent, але вже з інформацією про аккаунт.

```
val launcher = rememberLauncherForActivityResult(contract = ActivityResultContracts.StartActivityForResult()) {
    it.data?.let { intent ->
        val credential = googleAuthUiClient.getGoogleAccountCredentialFromIntent(intent)
        viewModel.saveAuthToken(credential)
    }
}

LaunchedEffect(key1 = true) {
    googleAuthUiClient.getLastSignedInAccountCredential()?.let { credential ->
        viewModel.saveAuthToken(credential)
    } ?: run {
        launcher.launch(googleAuthUiClient.buildSignInIntent())
    }
}
```

Рис. 2.32. Місце виклику функції buildSignInIntent

Після отримання об’єкта intent, можна вже отримати інформацію про аккаунт користувача викликавши функцію getGoogleAccount CredentialFromIntent, передавши їй даний intent (рис. 2.33).

```
fun getGoogleAccountCredentialFromIntent(intent: Intent): GoogleAccountCredential {
    val task = GoogleSignIn.getSignedInAccountFromIntent(intent)
    val account = task.result

    val credential = GoogleAccountCredential.usingOAuth2(context, listOf(CalendarScopes.CALENDAR))
    credential.selectedAccount = account.account

    return credential
}
```

Рис. 2.33. Функція GetGoogleAccountCredentialFromIntent

Потім з об’єкта GoogleAccountCredential можна отримати токен авторизації, який вже збережеться локально і буде використовуватися для авторизованих запитів до сервісу Google Calendar Api (рис. 2.34).

```
fun saveAuthToken(credential: GoogleAccountCredential) {
    viewModelScope.launch(Dispatchers.IO) {
        saveAuthTokenUseCase(credential.token)
        _navToMainScreen.emit( value: true)
    }
}
```

Рис. 2.34. Функція для отримання і збереження токену авторизації

В процесі авторизації в додаток потрібно реалізувати й інший випадок, коли користувач вже пройшов авторизацію, щоб отримати об’єкт GoogleAccountCredential з останнього авторизованого аккаунта. Для цього використовується функція getLastSignedInAccountCredential (рис. 2.35).

```
fun getLastSignedInAccountCredential(): GoogleAccountCredential? {
    val account = GoogleSignIn.getLastSignedInAccount(context)
    return account?.account?.let {
        GoogleAccountCredential.usingOAuth2(context, listOf(CalendarScopes.CALENDAR)).apply {
            selectedAccount = it
        }
    }

}
```

Рис. 2.35. Функція getLastSignedInAccountCredential

В момент коли відкривається додаток, першим є екран авторизації, де ми і перевіряємо чи вже є авторизований Google аккаунт. Ми викликаємо функцію getLastSignedInAccountCredential і якщо вона повертає якесь значення, то система перенаправляє користувача на наступний екран, а якщо ні, то відбувається процес авторизації (рис. 2.36).

```
LaunchedEffect(key1 = true) {
    googleAuthUiClient.getLastSignedInAccountCredential()?.let { credential ->
        viewModel.saveAuthToken(credential)
    } ?: run {
        launcher.launch(googleAuthUiClient.buildSignInIntent())
    }
}
```

Рис. 2.36. Перевірка на вже авторизований аккаунт користувача

У випадку, якщо користувач хоче змінити вибраний при авторизації Google аккаунт, викликається функція signOut (рис. 2.37).

```
fun signOut() {
    googleSignInClient.signOut()
}
```

Рис. 2.37. Функція signOut

### 2.4.2 Кешування даних

Кешування даних у мобільному додатку потрібне для підвищення його продуктивності та оптимізації використання ресурсів. Воно дозволяє зберігати часто використовувані або тимчасові дані локально на пристрої, щоб уникнути повторних запитів до сервера або тривалих операцій обробки. Це зменшує затримки при завантаженні контенту, економить інтернет-трафік і знижує навантаження на сервери. Крім того, кешування забезпечує кращий користувацький досвід, оскільки додаток може працювати швидше та

залишатися функціональним навіть у разі відсутності стабільного інтернет-з'єднання.

Кешування реалізоване в репозиторії додатку, оскільки саме репозиторій, згідно чистій архітектурі, повинен вирішувати, які дані віддавати під час виклику функції, з серверу чи з бази даних. Коли сервер повертає нам певні дані по запиту додатку, то ми їх відразу зберігаємо локально і у випадку відсутності доступу до інтернету чи отриманні помилки з серверу використовуємо локальні джерела даних.

В мобільному додатку для оптимізованого управління подіями застосовується кешування для подій та палітри кольорів календаря.

Запит до серверу відбувається в try-catch блоці, який перехоплює всі помилки і в разі викикнення однієї з них виконає аналогічний запит локально. Всі функції, які будуть продемонстровані далі, працюють по цьому принципу.

Під час запиту додатком списку публічних і приватних подій викликається функція getListCalendarData (рис. 2.38). Одразу після отримання списків подій з серверу, вони кешуються.

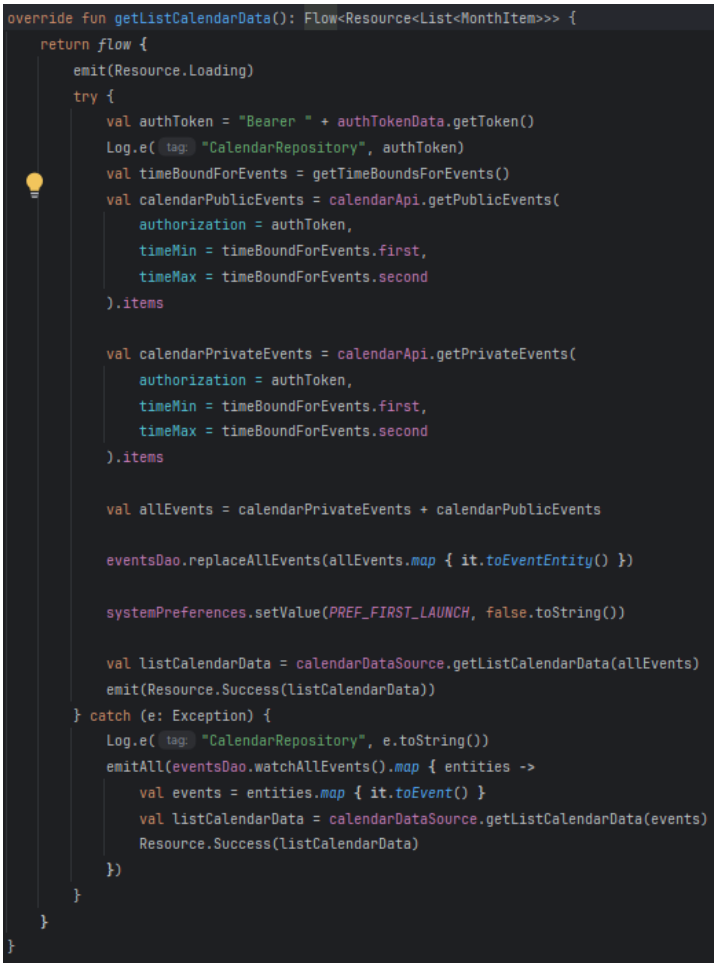



Рис. 2.38. Функція для отримання списків подій

Під час запиту додатком детальної інформації про подію, викликається функція getEventById (рис. 2.39).

```
override fun getEventById(calendarId: String, eventId: String): Flow<Resource<Event>> {
    return flow {
        emit(Resource.Loading)
        try {
            val token = "Bearer " + authTokenData.getToken()
            val event = calendarApi.getEventById(
                authorization = token,
                calendarId = calendarId,
                eventId = eventId
            )
            emit(Resource.Success(event))
        } catch (e: Exception) {
            val event = eventsDao.getEventById(eventId)!!.toEvent()
            emit(Resource.Success(event))
        }
    }
}
```

Рис. 2.39. Функція для отримання детальної інформації про подію

Під час запиту додатком інформації про кольорову палітру календаря, викликається функція getCalendarColorPalette (рис. 2.40).

```
override fun getCalendarColorPalette(): Flow<Resource<CalendarColorPalette>> {
    return flow {
        emit(Resource.Loading)
        try {
            val token = "Bearer " + authTokenData.getToken()
            val colorPalette = calendarApi.getCalendarColorPalette(token)

            colorsDao.replaceAllColors(colorPalette.eventColors.toEventColorEntities())
            emit(Resource.Success(colorPalette))
        } catch (e: Exception) {
            emitAll(colorsDao.watchAllColors().map { entities ->
                val colorPalette = CalendarColorPalette(
                    calendarColors = emptyMap(),
                    eventColors = entities.toEventColors()
                )
                Resource.Success(colorPalette)
            })
        }
    }
}
```

Рис. 2.40. Функція getCalendarColorPalette

Під час запиту додатку на додавання нової події, викликається функція insertEvent (рис. 2.41).

```
override fun insertEvent(eventBody: EventBodyRequest): Flow<Resource<Event>> {
    return flow {
        emit(Resource.Loading)
        try {
            val token = "Bearer " + authTokenData.getToken()
            val event = calendarApi.insertEvent(
                authorization = token,
                calendarId = "primary",
                eventBody = eventBody
            )
            eventsDao.insertEvent(event.toEventEntity())
            emit(Resource.Success(event))
        } catch (e: Exception) {
            Log.e( tag: "CalendarRepository",  msg: "insert error: $e")
            val eventEntity = buildEventEntity( eventId: null, eventBody)
            eventsDao.insertEvent(eventEntity)
            emit(Resource.Success(eventEntity.toEvent()))
        }
    }
}
```

Рис. 2.41. Функція insertEvent

Під час запиту додатку на редагування події, викликається функція updateEvent (рис. 2.42).

```
override fun updateEvent(eventId: String, eventBody: EventBodyRequest): Flow<Resource<Event>> {
    return flow {
        emit(Resource.Loading)
        try {
            val token = "Bearer " + authTokenData.getToken()
            val event = calendarApi.updateEvent(
                authorization = token,
                calendarId = "primary",
                eventId = eventId,
                eventBody = eventBody
            )
            eventsDao.insertEvent(event.toEventEntity())
            emit(Resource.Success(event))
        } catch (e: Exception) {
            Log.e( tag: "CalendarRepository",  msg: "update error: $e")
            val eventEntity = buildEventEntity(eventId, eventBody)
            eventsDao.insertEvent(eventEntity)
            emit(Resource.Success(eventEntity.toEvent()))
        }
    }
}
```

Рис. 2.42. Функція updateEvent

Під час запиту додатку на видалення події, викликається функція deleteEvent (рис. 2.43).

```
override fun deleteEvent(eventId: String): Flow<Resource<Unit>> {
    return flow {
        emit(Resource.Loading)
        try {
            val token = "Bearer " + authTokenData.getToken()
            calendarApi.deleteEvent(
                authorization = token,
                eventId = eventId
            )
            eventsDao.deleteEvent(eventId)
            emit(Resource.Success(Unit))
        } catch (e: Exception) {
            eventsDao.deleteEvent(eventId)
            emit(Resource.Success(Unit))
        }
    }
}
```

Рис. 2.43. Функція deleteEvent

### 2.4.3. Офлайн синхронізація

Офлайн синхронізація в мобільному додатку потрібна для забезпечення безперервної роботи користувача навіть без доступу до інтернету. Вона дозволяє додатку зберігати локально внесені зміни, переглядати дані, які були завантажені раніше, і синхронізувати ці зміни із сервером, щойно з'явиться з'єднання. Це важливо для покращення користувацького досвіду, особливо в умовах нестабільної мережі, і гарантує, що дані не будуть втрачені, а робота користувача залишатиметься ефективною незалежно від доступності інтернету.

Офлайн синхронізація в нашому додатку застосовується лише для подій, так як тільки з ними користувач виконує всі можливі операції, та відбувається через функцію synchronizeEvents, яка написана в репозиторії, тому що він має доступ до різних джерел даних. Функція доволі велика, тому розберемо її фрагментами по порядку.

Всі виклики до серверу в даній функції виконуються в try-catch блоці, щоб відловити всі помилки.

Спочатку потрібно отримати список приватних подій користувача з серверу (рис. 2.44).

```
val apiPrivateEvents = calendarApi.getPrivateEvents(
    authorization = token,
    timeMin = timeBoundForEvents.first,
    timeMax = timeBoundForEvents.second
).items
```

Рис. 2.44. Отримання приватних подій користувача з серверу

Далі отримуємо список приватних подій користувача з бази даних. Для цього отримуються всі події, а потім за допомогою функції filter відфільтровуються ті що нам потрібні (рис. 2.45).

```
val dbPrivateEvents = eventsDao.getAllEvents().filter {
    it.visibility != "public"
}
```

Рис. 2.45. Отримання приватних подій користувача з бази даних

Після отримання необхідних даних в нас далі знаходиться два цикли, в яких переглядаються всі події і з бази даних, і з серверу, порівнюються по їх унікальних ідентифікаторах та по повній ідентичності всіх полів. Якщо якась з подій на сервері відрізняється від події в базі даних, подія на сервері оновлюється (рис. 2.46).

```
for(dbEvent in dbPrivateEvents) {
    for(apiEvent in apiPrivateEvents) {

        // CHECK FOR ITEMS TO UPDATE
        if(dbEvent.eventId == apiEvent.eventId && dbEvent.toEvent() != apiEvent) {
            calendarApi.updateEvent(
                authorization = token,
                calendarId = "primary",
                eventId = apiEvent.eventId,
                eventBody = dbEvent.toEventBodyRequest()
            )
        }

    }
}
```

Рис. 2.46. Порівняння подій для їх оновлення

Потім аналогічним способом переглядаються події з локального і віддаленого джерел даних та знаходяться події з унікальним ідентифікатором, які є в базі даних, але немає на сервері. Якщо такі події знайшлися, то вони додаються на сервер (рис. 2.47).

```
// CHECK FOR ITEMS TO INSERT
for(dbEvent in dbPrivateEvents) {
    if(!apiPrivateEvents.map { it.eventId }.contains(dbEvent.eventId)) {
        calendarApi.insertEvent(
            authorization = token,
            calendarId = "primary",
            eventBody = dbEvent.toEventBodyRequest()
        )
    }
}
```

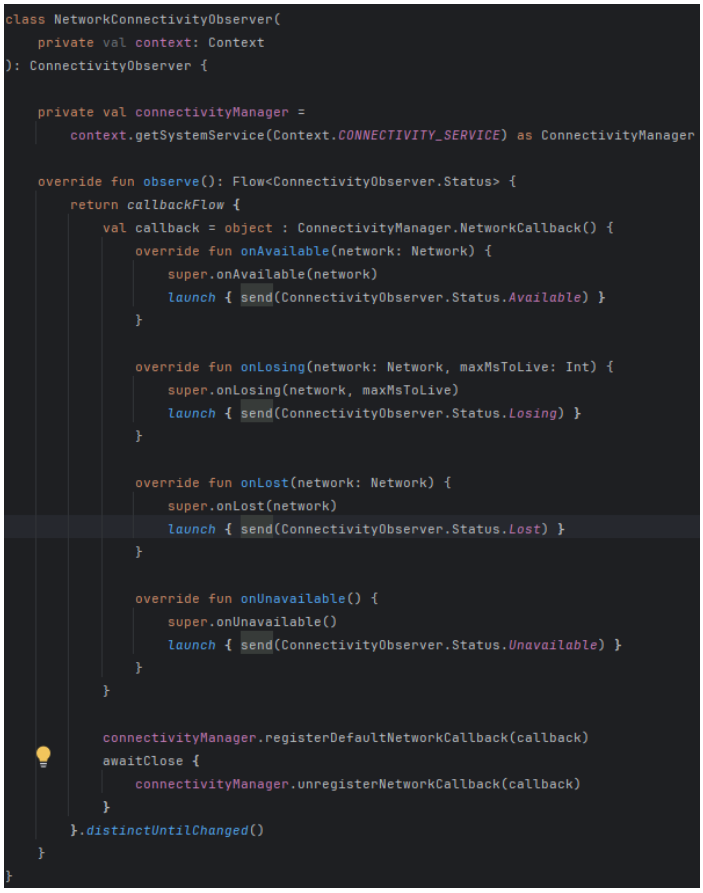
Рис. 2.47. Процес пошуку відсутніх подій на сервері для їх подальшого додавання

Далі потрібно знайти події, які були видалені локально, але все ще залишаються на сервері. Тут потрібно додати умовний оператор на перевірку першого запуску додатку, тому що дані з серверу ще не були отримані і у випадку даної операції можуть бути всі звідти видалені (рис. 2.48).

```
//CHECK FOR ITEMS TO DELETE
if(systemPreferences.getValue(PREF_FIRST_LAUNCH) != null) {
    val dbEventIds = dbPrivateEvents.map { it.eventId }.toSet()
    val eventsToDeleteFromApi = apiPrivateEvents.filter { apiEvent ->
        !dbEventIds.contains(apiEvent.eventId)
    }
    for(event in eventsToDeleteFromApi) {
        calendarApi.deleteEvent(
            authorization = token,
            eventId = event.eventId!!
        )
    }
}
```

Рис. 2.48. Процес пошуку локально видалених подій для їх подальшого видалення на сервері

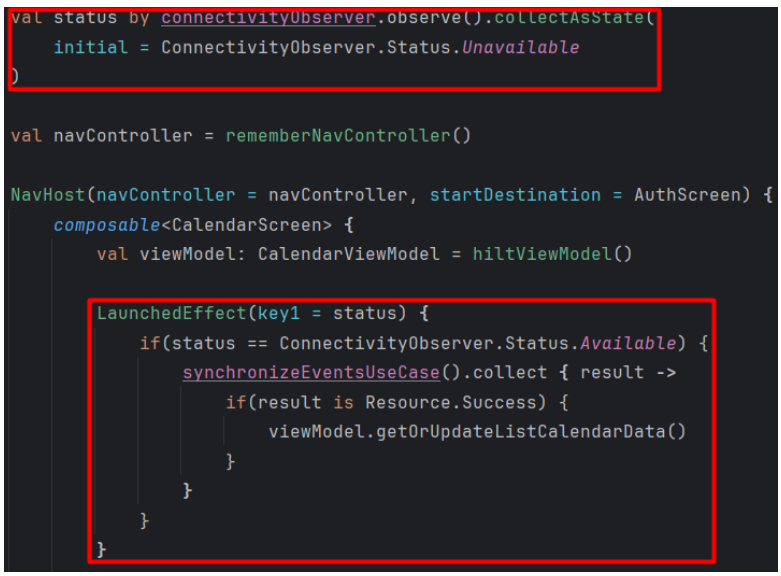
Функцію synchronizeEvents потрібно викликали, коли з’являється підключення до інтернету. Для цього в додатку написаний клас NetworkConnectivityObserver (рис. 2.49), який після будь-яких змін в статусі підключення, відразу сповістить про це.



```
class NetworkConnectivityObserver(
    private val context: Context
): ConnectivityObserver {

    private val connectivityManager =
        context.getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager

    override fun observe(): Flow<ConnectivityObserver.Status> {
        return callbackFlow {
            val callback = object : ConnectivityManager.NetworkCallback() {
                override fun onAvailable(network: Network) {
                    super.onAvailable(network)
                    launch { send(ConnectivityObserver.Status.Available) }
                }

                override fun onLosing(network: Network, maxMsToLive: Int) {
                    super.onLosing(network, maxMsToLive)
                    launch { send(ConnectivityObserver.Status.Losing) }
                }

                override fun onLost(network: Network) {
                    super.onLost(network)
                    launch { send(ConnectivityObserver.Status.Lost) }
                }

                override fun onUnavailable() {
                    super.onUnavailable()
                    launch { send(ConnectivityObserver.Status.Unavailable) }
                }
            }

            connectivityManager.registerDefaultNetworkCallback(callback)
            awaitClose {
                connectivityManager.unregisterNetworkCallback(callback)
            }
        }.distinctUntilChanged()
    }
}
```

Рис. 2.49. Клас NetworkConnectivityObserver

Стан підключення до інтернету відслідковується в стартовій точці додатку, де і викликається функція synchronizeEvents та оновлюються дані на головному додатку (рис. 2.50).



```
val status by connectivityObserver.observe().collectAsState(
    initial = ConnectivityObserver.Status.Unavailable
)

val navController = rememberNavController()

NavHost(navController = navController, startDestination = AuthScreen) {
    composable<CalendarScreen> {
        val viewModel: CalendarViewModel = hiltViewModel()

        LaunchedEffect(key1 = status) {
            if(status == ConnectivityObserver.Status.Available) {
                synchronizeEventsUseCase().collect { result ->
                    if(result is Resource.Success) {
                        viewModel.getOrUpdateListCalendarData()
                    }
                }
            }
        }
```

Рис. 2.50. Відслідковування підключення до інтернету та офлайн синхронізація

2.4.4. Сповіщення

Щоб відправляти сповіщення, потрібно спочатку створити канал для них. Це можна зробити в головному класі додатку CalendarApplication, викликавши функцію createNotificationChannel, при старті додатку (рис. 2.51).

```
@HiltAndroidApp
class CalendarApplication: Application() {

    override fun onCreate() {
        super.onCreate()
        createNotificationChannel()
    }

    private fun createNotificationChannel() {
        if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            val channel = NotificationChannel(
                ReminderNotificationService.REMINDER_CHANNEL_ID,
                 name: "Reminders",
                NotificationManager.IMPORTANCE_DEFAULT
            )
            channel.description = "Used to send notifications about events"

            val notificationManager = getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
            notificationManager.createNotificationChannel(channel)
        }
    }
}
```

Рис. 2.51. Клас CalendarApplication

Після цього відповідний канал з’явиться в налаштування додатку (рис. 2.52).

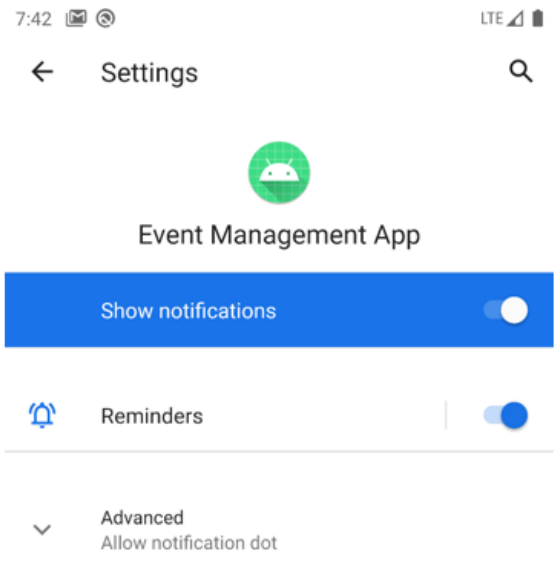



Рис. 2.52. Канал Reminders для відправки сповіщень

Далі потрібно створити клас ReminderNotificationService, який буде відповідати за створення та відправку сповіщення на пристрій (рис. 2.53).

```
class ReminderNotificationService(
    private val context: Context
) {

    private val notificationManager = context.getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager

    companion object {
        const val REMINDER_CHANNEL_ID = "reminder_channel"
    }

    fun showNotification(message: String) {
        val activityIntent = Intent(context, MainActivity::class.java)
        val activityPendingIntent = PendingIntent.getActivity(
            context,
            requestCode: 1,
            activityIntent,
            PendingIntent.FLAG_IMMUTABLE
        )
        val notification = NotificationCompat.Builder(context, REMINDER_CHANNEL_ID)
            .setContentText(message)
            .setContentIntent(activityPendingIntent)
            .build()

        notificationManager.notify( id: 1, notification)
    }
}
```

Рис. 2.53. Клас ReminderNotificationService

Потім потрібно створити клас NotifyWorker (рис. 2.54), який буде відповідати за сам процес запланової відправки сповіщення. Це потрібно зробити через інструмент WorkManager, який є однією із бібліотек від Google.

WorkManager у мобільному додатку потрібен для виконання задач, які мають працювати у фоновому режимі, навіть якщо додаток закритий або пристрій перезавантажено. Він дозволяє надійно запускати завдання, які потребують гарантованого завершення, наприклад, завантаження файлів, синхронізацію даних із сервером чи обробку великих обсягів інформації. WorkManager автоматично враховує обмеження системи, такі як стан батареї чи доступ до інтернету, і виконує задачі, лише коли для цього є сприятливі умови. Це робить його зручним інструментом для управління фоновими операціями в Android-додатках.

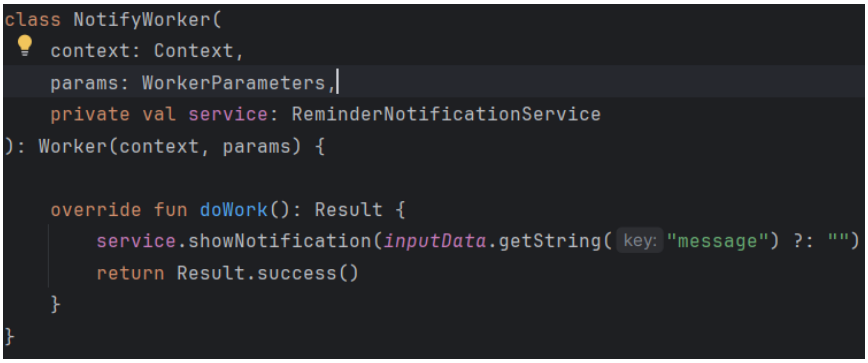



Рис. 2.54. Клас NotifyWorker

В момент додавання користувачем нового сповіщення, воно додається в чергу кодом, зображеним на рис. 2.55.

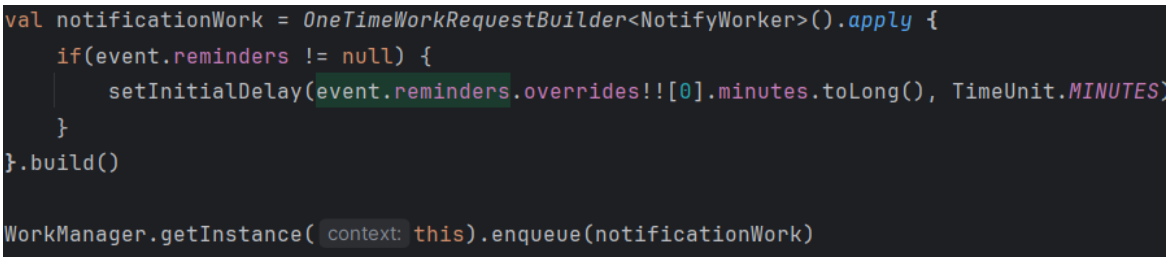



Рис. 2.55. Додавання сповіщення в чергу

### 2.5. Висновки до розділу 2

У даному розділі було детально розглянуто процес створення мобільного додатку для оптимізованого управління подіями, який базується на чистій архітектурі. Такий підхід забезпечує чітку структурованість додатку, його функціональність, легкість у розширенні та підтримці. Це дозволяє створити продукт, який відповідає сучасним вимогам ринку мобільних додатків.

Першим етапом реалізації проекту стало обрання відповідних мов програмування та технологій. Основною мовою розробки було вибрано Kotlin, що є оптимальним вибором для створення мобільних додатків під платформу Android. Ця мова поєднує простоту, сучасний синтаксис, потужні функціональні можливості та високу продуктивність. Особливу увагу приділено використанню інструментів, що входять до складу Kotlin, таких як Kotlin Coroutines для роботи

з асинхронними процесами та Kotlin Flows для управління потоками даних. Це дозволило значно підвищити стабільність і продуктивність додатку.

Архітектура проекту реалізована у відповідності до принципів Clean Architecture, що передбачає поділ додатку на кілька ключових шарів. Кожен з цих шарів виконує конкретну функцію:

- Data-шар відповідає за управління даними додатку. Це включає роботу з локальною базою даних, реалізованою за допомогою Room, та інтеграцію з мережевим інтерфейсом, створеним на основі Retrofit 2.
- Domain-шар включає основну бізнес-логіку додатку, яка забезпечує обробку даних і реалізацію основних функцій.
- Presentation-шар займається відображенням даних та взаємодією з користувачем. Для створення інтерфейсу використано Jetpack Compose, що дозволяє будувати сучасні, адаптивні та інтерактивні інтерфейси.
- DI-шар (Dependency Injection) реалізовано з використанням бібліотеки Dagger Hilt, що спрощує управління залежностями та забезпечує ефективне використання ресурсів.

Було проаналізовано кожен з шарів архітектури, їх складові, а також механізми взаємодії між ними. Такий підхід не лише забезпечує чітке розмежування функціональності, але й сприяє зручності в доопрацюванні або масштабуванні проекту.

Основний функціонал додатку охоплює всі необхідні операції з подіями, включаючи їх створення, редагування, перегляд і видалення, що забезпечує користувачеві повний контроль над розкладом. Впроваджено механізми кешування для збереження даних у локальній пам'яті, що дозволяє швидко отримувати доступ до них навіть за відсутності інтернету. Додаток також підтримує офлайн синхронізацію, яка забезпечує роботу з подіями без мережевого підключення та синхронізацію змін після відновлення доступу до мережі. Крім того, реалізовано функцію сповіщень, яка своєчасно інформує користувачів про майбутні події, допомагаючи ефективно управляти часом.

# РОЗДІЛ 3
# ПРЕЗЕНТАЦІЯ ТА ТЕСТУВАННЯ МОБІЛЬНОГО ДОДАТКУ

## 3.1. Запуск додатку

Для запуску додатку не потрібно ніяких додаткових налаштувань, якщо він не встановлений на мобільному телефоні або на емуляторі, то його потрібно встановити і просто запустити, як звичайний мобільний додаток.

## 3.2. Презентація мобільного додатку

При першому запуску додатку користувач потрапляє на екран авторизa-ції. Екран авторизації не містить ніякого графічного інтерфейсу і користувачу буде відразу показане випадаюче вікно для вибору Google аккаунту (рис. 3.1). Якщо на пристрої немає Google аккаунтів, система перенаправить користувача на процес його створення. Авторизація користувача відбувається саме через Google сервіс, тому що в додатку присутня інтеграція з Google Calendar Api і для подальшої роботи потрібен токен авторизації з аккаунту користувача.

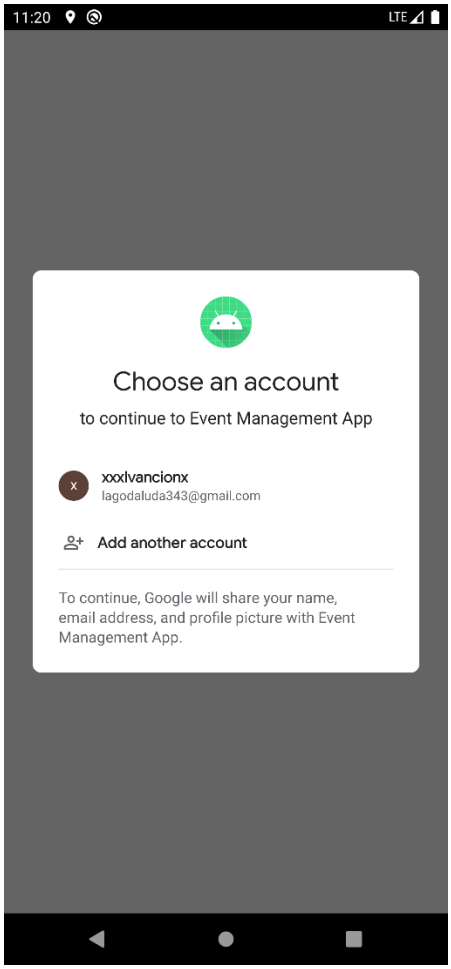



Рис. 3.1. Екран авторизації

Після успішної авторизації користувача буде перенаправлено на головне вікно додатку з календарем. Календар має відображення у вигляді вертикального списку по місяцях. Після кожного ілюстративного зображення місяця показується список з тижнів та подій з датами в їх межах (рис. 3.2). Всі події можна поділити на публічні, тобто офіційні свята в Україні, та приватні, тобто події створені користувачем.

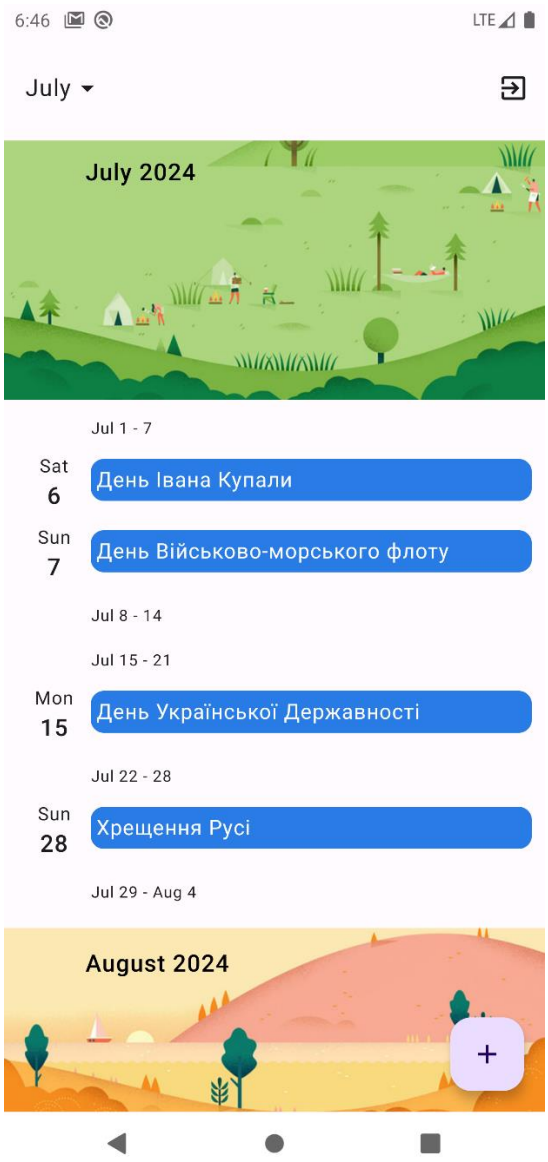



Рис. 3.2. Головний екран додатку

На верхній панелі додатку (рис. 3.3) знаходиться текст із назвою першого видимого місяця на екрані та кнопка виходу. При натисненні кнопки виходу користувача буде перенаправлено на екран авторизації, де йому знову буде необхідно її пройти. При натисненні на текст місяця розгорнеться навігаційний календар із списком дат на певний місяць та самим списком місяців (рис. 3.4), який можна прокручувати. Якщо натиснути на один із місяців, то основний календар буде прокручений до потрібного місяця.


Рис. 3.3. Верхня панель головного екрану в згорнутому вигляді

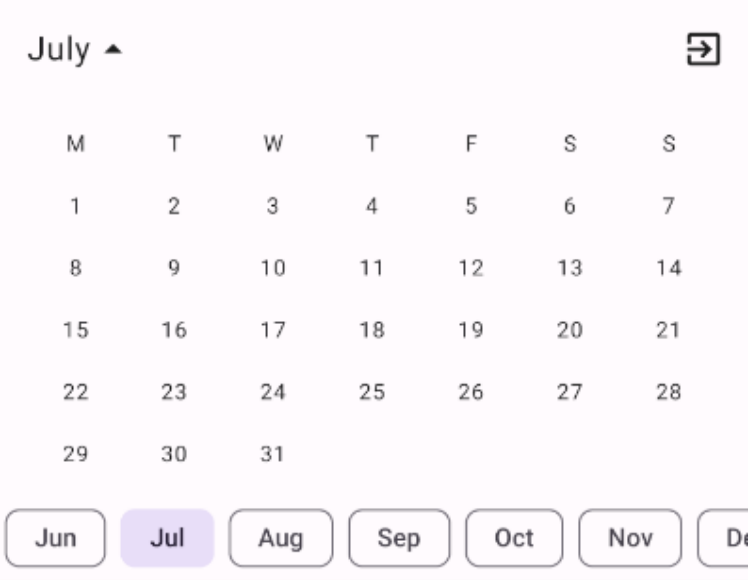



Рис. 3.4. Верхня панель екрану в розгорнутому вигляді

При натисканні на будь-які з подій, додаток перенаправляє користувача на екран деталей події, де він може ознайомитися з більш розгорнутою інформацією, наприклад, повною його назвою, детальнішою інформацією про дату та час та інформацією про встановлені сповіщення. Екран деталей для приватних (рис.3.6) і публічних (рис. 3.5) подій виглядають трішки по різному, так як публічні події користувач не може редагувати чи видаляти.

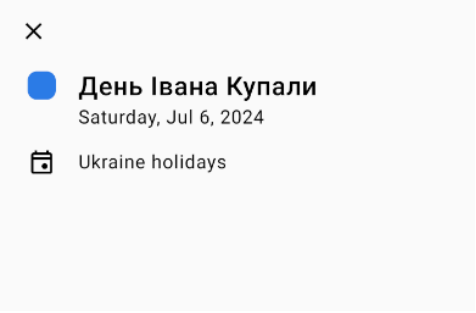



Рис. 3.5. Екран деталей публічної події

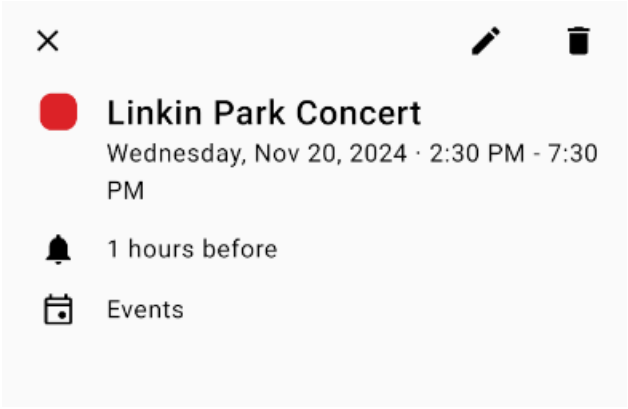



Рис. 3.6. Екран деталей приватної події

При натисканні на кнопку видалення, подія відразу видалиться. При натисканні на кнопку редагування, користувача буде перенаправлено на екран редагування події (рис. 3.7), де він може змінити назву, колір відображення події в календарі, налаштувати сповіщення та дату і час події.

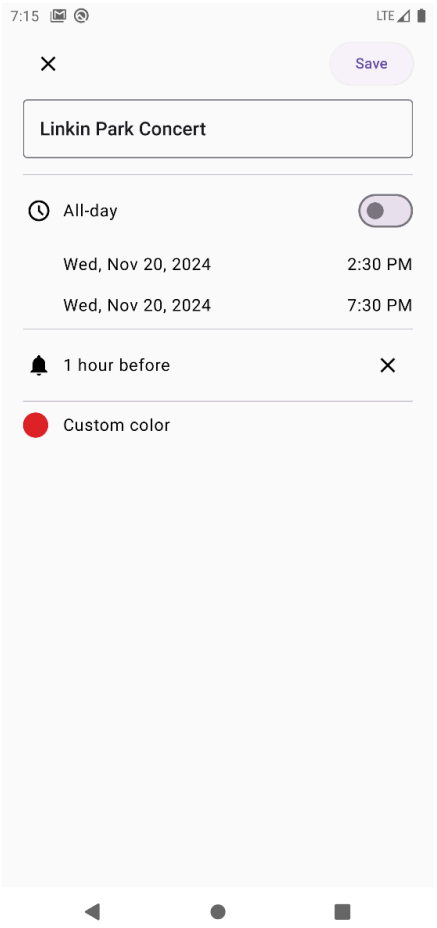



Рис. 3.7. Екран редагування події

При натисканні на текстове поле, користувач може змінити назву події (рис. 3.8).

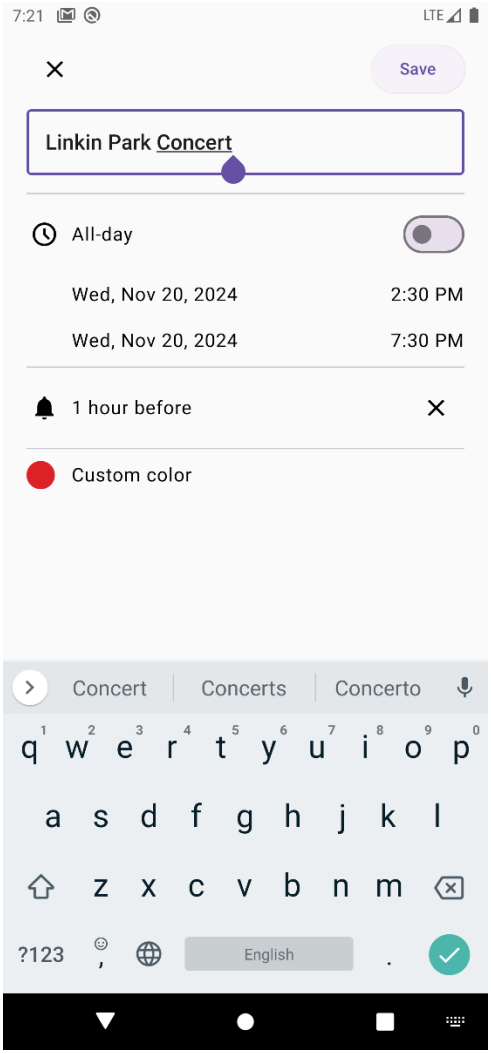



Рис. 3.8. Редагування назви події

При натисканні на дату початку чи кінця події, на екрані з’являється нове вікно з вибором дати (рис. 3.9), де користувач може обрати потрібне йому значення, а потім натиснути на кнопку “ОК”, щоб зберегти його.

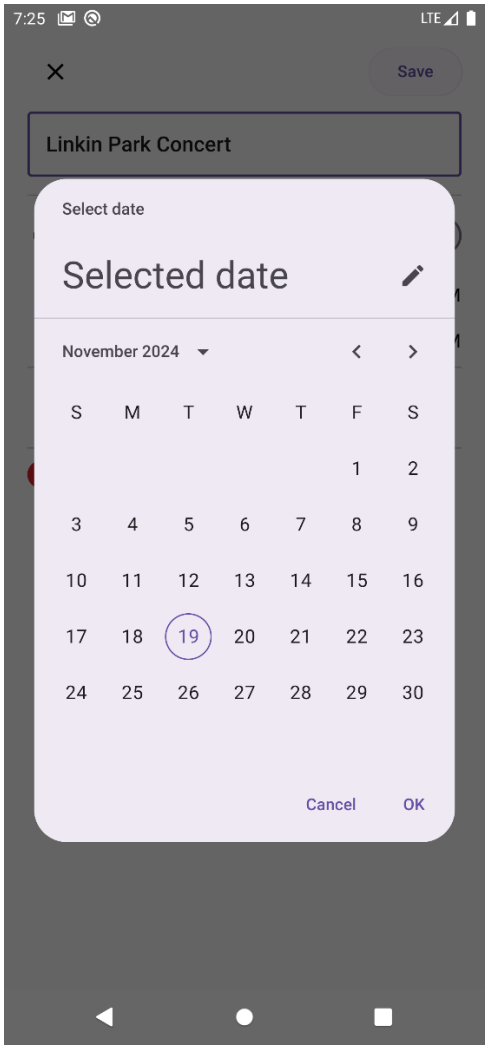



Рис. 3.9. Вікно вибору дати

У верхньому правому куті вікна вибору дати можна натиснути на кнопку редагування, щоб переключити вибір дати в ручний формат, де користувач може ввести дату з клавіатури пристрою (рис. 3.10).

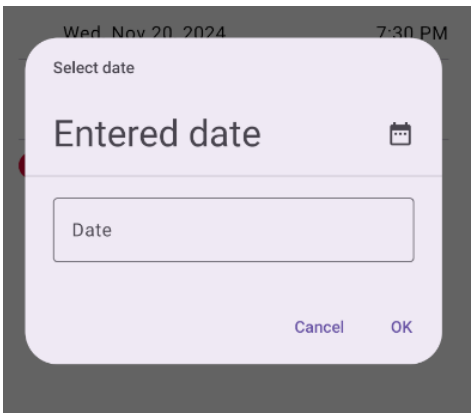



Рис. 3.10. Вікно вибору дати в ручному форматі

При натисканні на час початку чи кінця події, на екрані з’являється нове вікно з вибором часу (рис. 3.11), де користувач може обрати потрібне йому значення, а потім натиснути на кнопку “ОК”, щоб зберегти його.

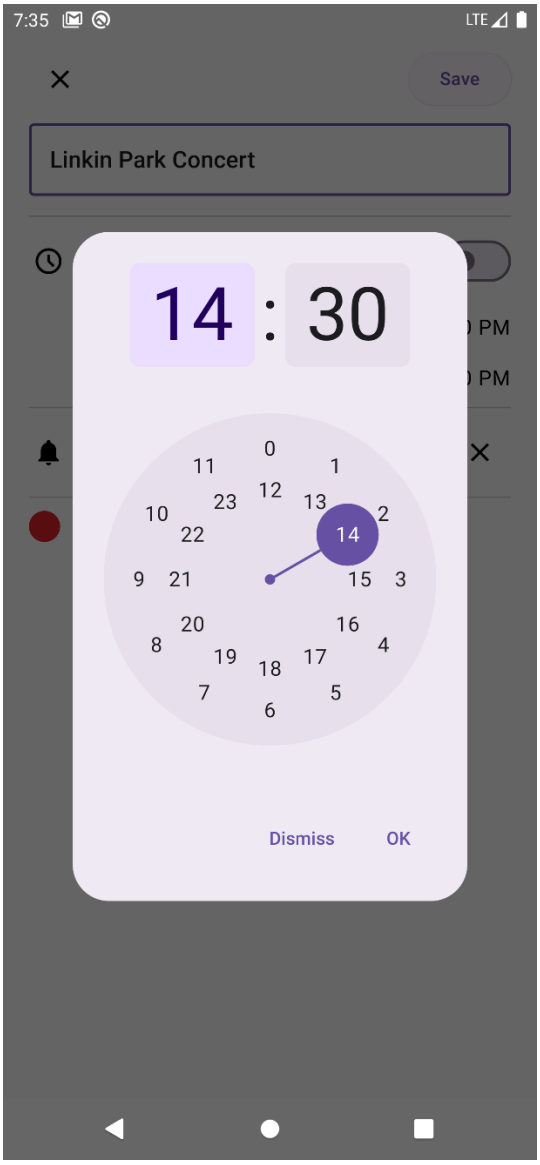



Рис. 3.11. Вікно вибору часу

У верхній частині вікна вибору часу знаходиться вибрана година та хвилини. Якщо натиснути на одне із цих значень, то користувачу буде перемикатися вікно, де він обирає години або хвилини. Для годин це вікно на рисунку показаному вище, а для хвилин аналогічне, тільки де користувач може обирати значення від 0 до 60.

Щоб додати сповіщення для події, користувач може натиснути на кнопку з текстом “Add notification”. На екрані з’явиться нове вікно з вибором для налаштуваня сповіщення (рис. 3.12), де користувачу потрібно буде натиснути на потрібний варіант. Якщо ж сповіщення для події вже є, то користувач може видалити старе натиснувши на кнопку видалення навпроти нього, а потім вже додати нове.

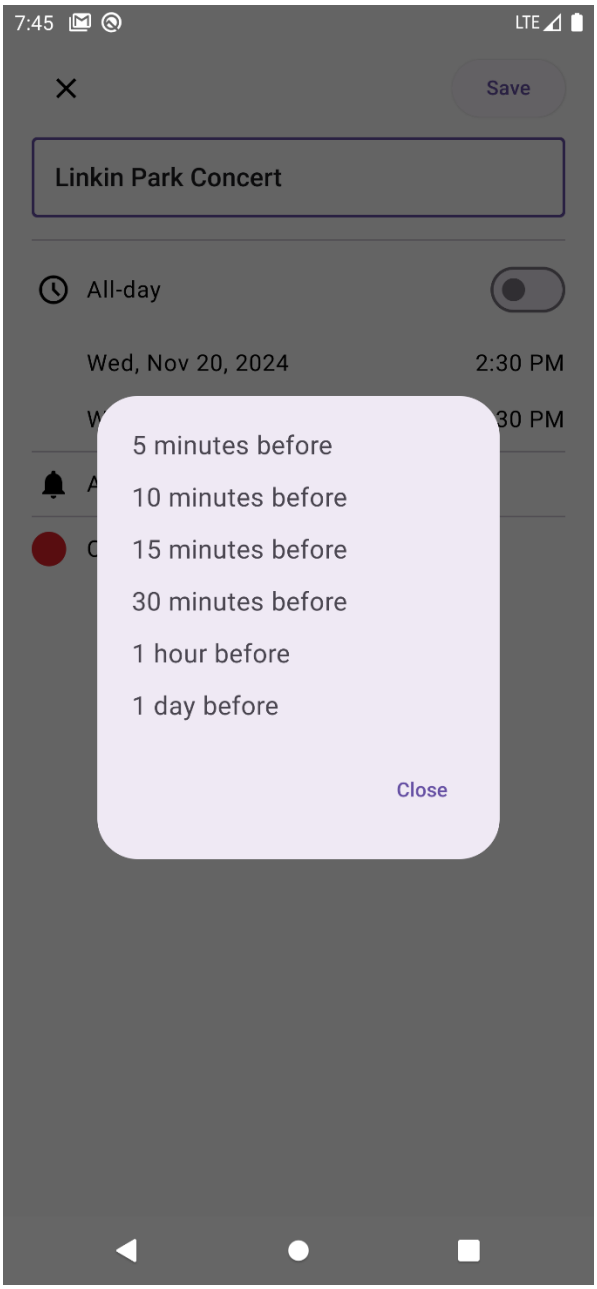



Рис. 3.12. Вікно вибору для сповіщення

Якщо користувач хоче змінити колір відображення події в календарі, то він повинен натиснути на кнопку вибору кольору в самому кінці екрану редагування. Після цього додаток покаже нове вікно з вибором кольорів і щоб вибрати один з них користувачу треба натиснути на нього (рис. 3.13).

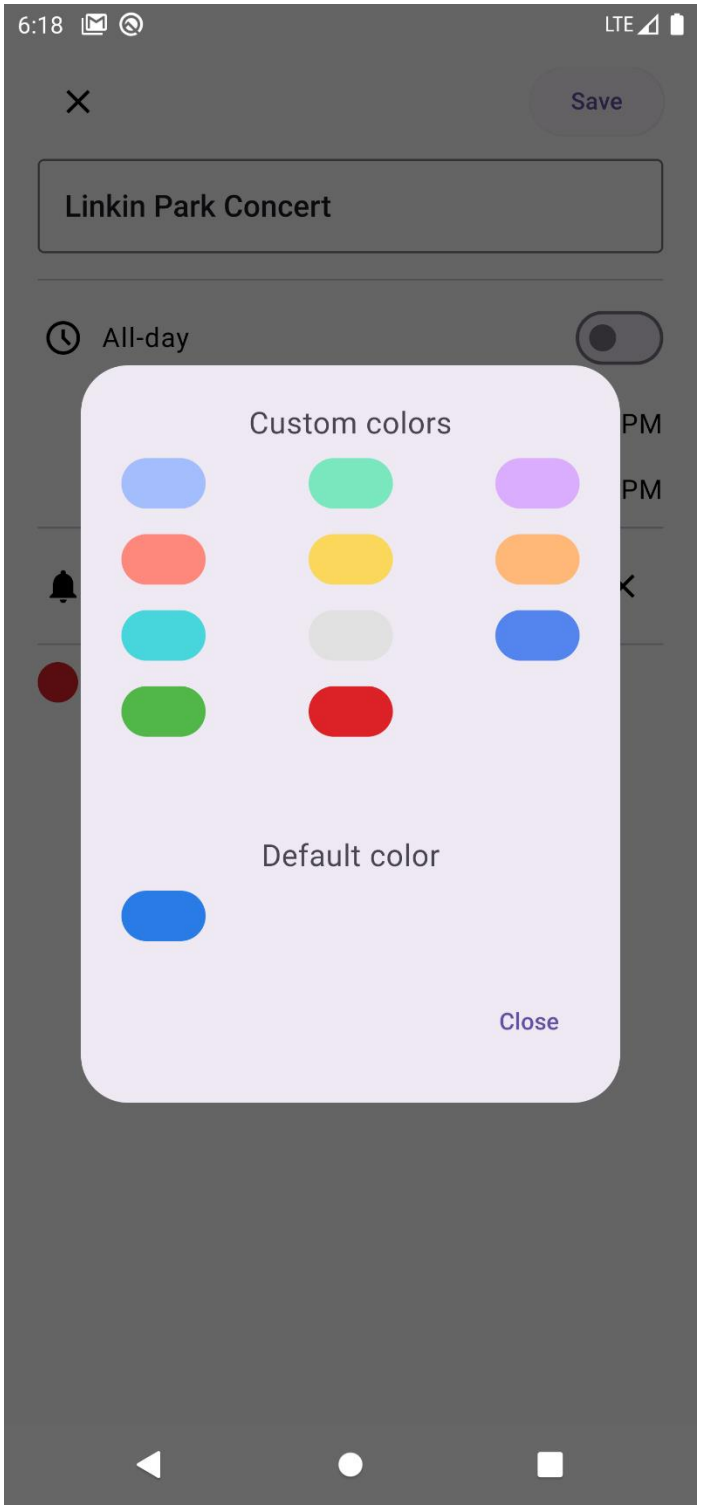



Рис. 3.13. Вибір кольору відображення події

Після всіх операцій редагування які провів користувач, він повинен натиснути на кнопку “Save” в правому верхньому кутку екрану редагування, щоб підтвердити всі зміни. Після цього додаток перенаправить користувача на головний екран з календарем та подіями.

Щоб додати нову подію, користувачу потрібно натиснути на кнопку додавання події в правому нижньому кутку головного екрану (рис. 3.14).

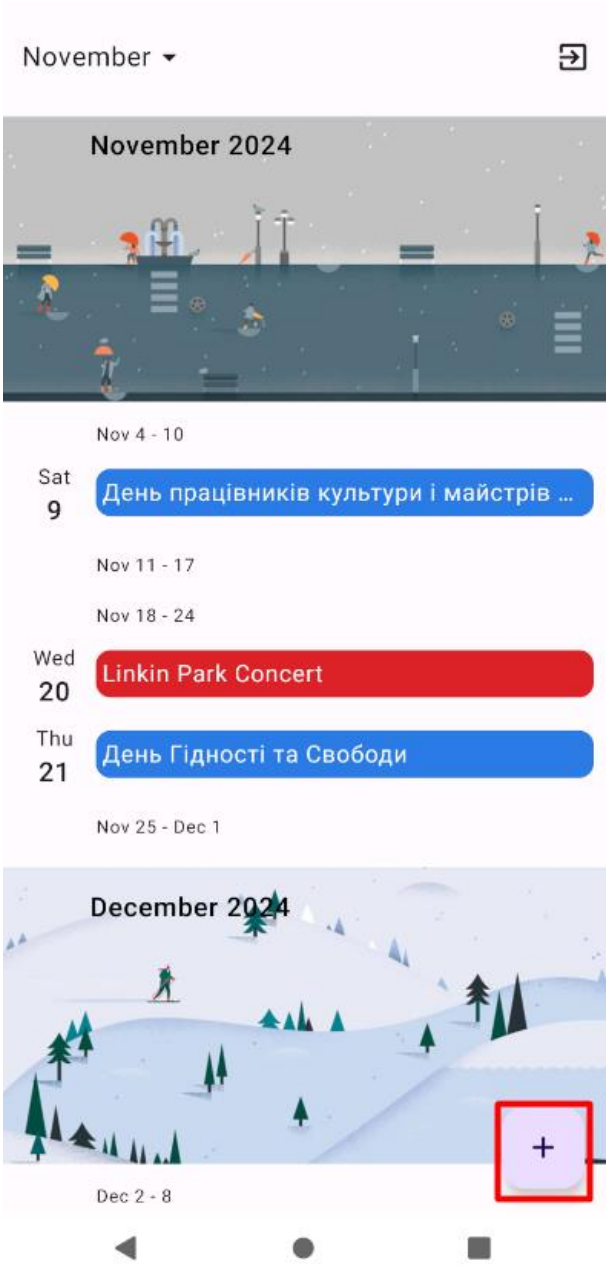



Рис. 3.14. Кнопка додавання події на головному екрані

### 3.3. Можливі покращення

Незважаючи на те, що мобільний додаток повністю справляється із поставленим завданням є ряд покращень, що можливі у майбутньому. Такі покращення сприятимуть удосконаленню програмного застосунку та додаванню нових функцій.

Наприклад, підтримка декількох Google аккантів. Завдяки цьому користувач зможе авторизуватися по якомусь із своїх аккаунтів, а потім додавати інші, щоб отимати список існуючих подій з них.

Також можна розширити список можливої інформації для події, щоб користувач, наприкла, міг додавати опис події, прикріпляти якісь документи або додавати місце проведення події, відкриваючи йому карту.

Серед можливих функцій також є можливість інтеграції з іншими сервісами, які пропонують схожий функціонал, наприклад, Microsoft Outlook Calendar або Apple Calendar. Таким чином користувач буде мати доступ до створених ним подій в цих сервісах.

Для продовження розробки застосунку, а також залучення інших розробників можна зробити відкритим вихідний код за допомогою сервісу GitHub. Завдяки цьому будь-хто зможе переглянути програмний код, завантажити та відредагувати його. Таким чином різні програмісти зможуть запропонувати свої рішення та покращення для застосунку.

У такого підходу також є свої переваги:

1. Перевірка безпеки та надійності. Завдяки відкритому вихідному коду, багато експертів може аналізувати код на предмет потенційних вразливостей та помилок безпеки. Це дозволяє швидко виявляти та виправляти проблеми безпеки, що забезпечує більшу надійність та захищеність додатку.

2. Гнучкість та адаптованість. З відкритим вихідним кодом додаток можна змінювати та адаптувати до власних потреб. Користувачі можуть вносити зміни, додавати нові функції або інтегрувати з іншими інструментами. Це дозволяє кожному користувачеві налаштувати додаток під свої індивідуальні потреби та використовувати його у своїх проектах.

3. Вільна ліцензія та розповсюдження. Завдяки вільній ліцензії, додаток може бути розповсюджуваним та використовуваним безкоштовно. Це зробить його доступним для широкої аудиторії та сприятиме його поширенню серед користувачів.

4. Взаємодія з іншими проектами: Відкритий вихідний код мобільного додатку дає можливість інтегрувати його з іншими проектами або інструментами. Це сприяє співпраці та обміну даними з іншими розробниками та сприяє створенню екосистеми зі спільними цілями та завданнями.

### 3.4. Висновки до розділу 3

У даному розділі було проаналізовано основні можливості та функціональність розробленого мобільного додатку для управління подіями. Для його використання необхідно встановити додаток на мобільний пристрій або емулятор. Після інсталяції та запуску додаток готовий до роботи без додаткових налаштувань.

Головний екран додатку забезпечує доступ до базових функцій планування подій, таких як створення, редагування та перегляд запланованих завдань. Інтерфейс інтуїтивно зрозумілий, що дозволяє користувачам швидко освоїти додаток і використовувати його для особистих цілей. Особливістю є можливість інтеграції з Google календарем, що спрощує синхронізацію запланованих подій.

У ході тестування було перевірено основні функції додатку, зокрема налаштування нагадувань та всі можливі маніпуляції з подіями. Додаток демонструє стабільну роботу, швидкий відгук і мінімальне споживання ресурсів пристрою, що робить його зручним у повсякденному використанні.

На даному етапі мобільний додаток виконує заявлені функції, забезпечуючи користувачеві доступ до ефективного та зручного інструменту для управління подіями. У подальшому можливе розширення функціональності для підвищення індивідуалізації та інтеграції з іншими сервісами.

# ВИСНОВКИ

Сьогодні мобільні додатки стали невід'ємною частиною нашого повсякденного життя, відіграючи ключову роль у спрощенні багатьох завдань. Їх широке використання сприяє ефективній організації часу, особливо в умовах сучасного динамічного темпу життя. Серед численних функцій, які вони виконують, важливе місце займає управління подіями та планування особистих чи професійних завдань. Зростаюча складність нашого щоденного розкладу потребує доступних, зручних та багатофункціональних інструментів, які могли б гнучко адаптуватися до індивідуальних потреб користувачів. Розробка мобільного додатку для організації подій є відповіддю на цей запит, адже він поєднує функціональність із простотою використання, забезпечуючи максимальну ефективність.

У рамках даної роботи було розглянуто проблему управління подіями та запропоновано інноваційне рішення у вигляді мобільного додатку. Проведений аналіз існуючих рішень виявив певні обмеження, такі як відсутність достатньої інтеграції з популярними платформами, складність використання для новачків, обмежений набір функцій та залежність від платних підписок. На основі цього аналізу було обрано відповідні засоби проєктування та розробки, що дозволило створити продукт, який враховує всі ці недоліки. Основною платформою для розробки було обрано популярну мобільну операційну систему, яка забезпечує широкий доступ до додатку для різних категорій користувачів. Використання сучасних мов програмування та спеціалізованих фреймворків дало змогу забезпечити високу якість програмного забезпечення, зокрема створити зручний, інтуїтивно зрозумілий інтерфейс, який відповідає вимогам сучасної цільової аудиторії.

Розроблений додаток дозволяє користувачам створювати, редагувати та організовувати особисті події, надаючи можливість налаштування персоналізованих нагадувань. Однією з ключових функцій є інтеграція з Google Календарем, яка забезпечує зручність синхронізації даних між додатком та цим сервісом. Така функціональність спрощує доступ до запланованих подій і

дозволяє ефективно використовувати існуючі дані, уникаючи дублювання інформації. У процесі тестування додатку було підтверджено його стабільність, швидкодію та відповідність заявленим функціям. Це свідчить про високу якість розробки та правильність обраного підходу до реалізації проєкту.

Додаток також націлений на доступність: він є інструментом, який може бути використаний як досвідченими, так і початківцями. Незалежно від рівня технічного досвіду, користувачі можуть легко опанувати його функціонал. Це досягається завдяки простому, але водночас естетичному дизайну, який відповідає принципам зручності використання (UX/UI).

Перспективи вдосконалення додатку є обнадійливими. Серед них можна виділити додавання нових функцій, таких як розширена інтеграція з іншими популярними сервісами, зокрема Microsoft Outlook, iCloud чи Trello. Це дозволить зробити додаток більш універсальним і привабливим для ширшої аудиторії. Також можна передбачити можливість розробки функції геолокації для подій, що дасть змогу автоматично відображати місце проведення запланованих заходів.

Особливу увагу слід приділити питанню безпеки даних користувачів, що є критично важливим у сучасному цифровому середовищі. У цьому контексті публікація програмного коду додатку на платформі GitHub дозволить залучити спільноту розробників до роботи над вдосконаленням продукту, зокрема в аспектах безпеки. Відкритий код сприятиме прозорості проєкту та забезпечить можливість швидко усувати потенційні вразливості. Окрім того, це дозволить залучити нові ідеї та технічні рішення, що зробить додаток ще більш функціональним і безпечним.

В результаті роботи було створено мобільний додаток, який здатний ефективно задовольняти сучасні запити користувачів щодо управління подіями. Він є не лише корисним інструментом для особистої організації, але й потужним рішенням для професійного застосування. Завдяки своїм перевагам у зручності, доступності та багатофункціональності, додаток має значний потенціал для подальшого розвитку та вдосконалення.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. What is Kotlin and why use it? [Електронний ресурс]. – режим доступу: https://www.techtarget.com/whatis/definition/Kotlin (дата звернення: 25.11.2024)
2. Meet Android Studio [Електронний ресурс]. – режим доступу: https://developer.android.com/studio/intro (дата звернення: 25.11.2024)
3. Kotlin Coroutines on Android [Електронний ресурс]. – режим доступу: https://developer.android.com/kotlin/coroutines (дата звернення: 25.11.2024)
4. Dependency injection with Hilt [Електронний ресурс]. – режим доступу: https://developer.android.com/training/dependency-injection/hilt-android (дата звернення: 25.11.2024)
5. Retrofit in Android [Електронний ресурс]. – режим доступу: https://medium.com/@KaushalVasava/retrofit-in-android-5a28c8e988ce (дата звернення: 22.11.2024)
6. Save data in a local database using Room [Електронний ресурс]. – режим доступу: https://developer.android.com/training/data-storage/room (дата звернення: 21.11.2024)
7. Understanding Jetpack Compose – part 1 of 2 [Електронний ресурс]. – режим доступу: https://medium.com/androiddevelopers/understanding- jetpack-compose-part-1-of-2-ca316fe39050 (дата звернення: 25.11.2024)
8. Understanding Jetpack Compose – part 1 of 2 [Електронний ресурс]. – режим доступу: https://medium.com/androiddevelopers/understanding- jetpack-compose-part-1-of-2-ca316fe39050 (дата звернення: 15.11.2024)
9. Exploring Coil: A powerful image loading library for android [Електронний ресурс]. – режим доступу: https://medium.com/androiddevelopers/understanding-jetpack-compose-part-1- of-2-ca316fe39050 (дата звернення: 25.11.2024)
10. What is an android app? [Електронний ресурс]. – режим доступу: https://www.builder.ai/glossary/android-app (дата звернення: 25.11.2024)
11. Friedman, M. Android Programming: The Big Nerd Ranch Guide / M. Фрідман. — 4-те вид. — США: Big Nerd Ranch, 2020. — 560 с.

12. González, S. Kotlin for Android Developers / С. Гонсалес. — 2-ге вид. — США: O'Reilly Media, 2020. — 400 с.

13. Elkhodary, M. Android Development with Kotlin / М. Ельходарі. — 1-ше вид. — США: Packt Publishing, 2021. — 350 с.

14. Meier, R. Professional Android / Р. Майєр. — 5-те вид. — США: Wrox, 2019. — 1000 с.

15. Sharma, R. Android Programming for Beginners / Р. Шарма. — 1-ше вид. — Нью-Йорк: Apress, 2020. — 470 с.

16. Sandler, R. The Busy Coder's Guide to Android Development / Р. Сендлер. — 12-те вид. — США: CommonsWare, 2020. — 1150 с.

17. Положення про кваліфікаційні роботи (проєкти) — Національний авіаційний університет (nau.edu.ua) [Електронний ресурс] – Режим доступу доресурсу: http://surl.li/dwhqah (дата звернення: 25.11.2024)