МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

КАФЕДРА КОМП’ЮТЕРНИХ НАУК

ВИПУСКНА РОБОТА

на тему:

«Віртуальний онлайн-тренер»

Завідувач

випускаючої кафедри **Довбиш А.С.**

Керівник роботи **Берест О.Б.**

Студента групи ІН – 63 **Яценко А.О.**

СУМИ 2020

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Кафедра комп'ютерних наук

Затверджую___________

Зав. кафедрою Довбиш А.С.

"_____" ____________2020 р.

ЗАВДАННЯ

до випускної роботи

Студентки четвертого курсу, групи ІН-63 спеціальності "Інформатика" денної форми навчання Яценко Анастасії Олександрівни.

Тема: "Віртуальний онлайн-тренер"

Затверджена наказом по СумДУ

№________ от ___________2020 р.

Зміст пояснювальної записки: 1) аналітичний огляд існуючих додатків; 2) постановка завдання; 3) опис основних компонентів архітектури мобільного додатку та критеріїв, що використовуються середовищем Android Studio; 5) розробка інтерфейсу додатку; 6) аналіз результатів тестування.

Дата видачі завдання "_____"____________2020 р.

Керівник випускної роботи __________ Берест О.Б.

Завдання приняв до виконання ____________ Яценко А.О.

РЕФЕРАТ

Записка: 62 стор., 34 рис., додаток, 10 джерел.

Об'єкт дослідження — вплив йоги на організм людини.

Мета роботи — розробка Android-додатку для популяризації занять йогою.

Предметом дослідження — є застосування сучасних методів та засобів розробки Android-додатку для популяризації занять йогою. Методи розробки базуються на технології Android Studio на мові програмування Kotlin, локальної бази даних Room та Android Data Binding Library.

Одержані результати полягають у розробленому та протестованому додатку.

Ключові слова: Android Studio, додаток, LiveData, Room, Kotlin, JetPack, компоненти, Data Binding Library, йога.

Зміст

Вступ 5
1 Аналіз предметної області 6
1.1 Опис предметної області 6
1.2 Огляд існуючих рішень 7
1.3 Постановка завдання 8
2 Проектування додатку 9
2.1 Опис використаних технологій 10
2.2 Проектування інтерфейсу 37
2.3 Інтерфейс додатку 39
3 Програмна реалізація 42
3.1 Інформаційна модель 42
3.2 Програмна реалізація сповіщень 44
4 Тестування 46
4.1 Проведені види тестування 46
4.2 Перевірка на працездатність додатку 47
Висновок 50
Список літератури 51
Додаток А. Лістинг коду 52

Вступ

У сучасному ритмі життя більшість людей хоче підтримувати свою фізичну форму за допомогою вправ. Найпоширеніший вид фізичної активності є фітнес, проте ціль цього додатку є ознайомлення користувачів із йогою. У більшості людей є деякі стереотипи щодо занять цим, наприклад, що це нудно, складно чи марна справа.

Проте, аби розвіяти ці думки, я вирішила створити додаток для початківців, щоб кожен між спробувати себе в заняттях йогою. Саме тому в ньому повинні бути представлені вправи різних напрямків.

Для створення додатку була використана така клієнт-серверна архітектура, щоб відповідала сучасним методам реалізації та не перевантажувала систему. А саме технологія Android Studio на мові програмування Kotlin, локальна база даних Room та Android Data Binding Library для прив'язки даних.

У практиці йоги відсутня боротьба і конкуренція, тому вона дійсно має цінність для людей, на відміну від фітнесу, який служить, наприклад, цілям ринку і капітала. Починаючи заняття йоги, цілком можливо ставити прості і, навіть, банальні цілі (вилікувати спину, зменшити трохи об'єму в стегнах або щось подібне). Саме тому я обрала створення додатку для досягнення вищеописаних цілей у зв'язку з пріорітетами сучасного світу. А саме – це доступність, практичність, скорочення витраченого часу та зрозумілість. Ці критерії були закладені в основі створення додатку.[6]

Можна вважати, що результатом дипломної роботи буде створений додаток, котрий ознайомить користувачів із практиками йоги, що в свою чергу допоможе урізноманітнити життя та вирішити деякі проблеми із самопочуттям.

1 Аналіз предметної області

1.1 Опис предметної області

У додатку будуть виконуватися такі процеси:

1. Створення/додання нотаток

- Створення теми замітки
- Написання основної частини(змісту)

2. Додання сповіщення

- Назва
- Зміст сповіщення
- Встановлювання часу спрацювання

3. Перегляд доступних практик

4. Видалення/редагування нотаток та сповіщень

Для початку роботи потрібно встановити на пристрій користувача та відкрити додаток. Щоб дізнатися навіщо це було створено, можна переглянути вкладку «Про додаток».

Рисунок 1.1 – Структура додатку

1.2 Огляд існуючих рішень

Більшість користувачів обирають додаток випадково, просто обираючи перший-ліпший у списку Play Market. Проте порівнюючи їх, можна знайти безліч недоліків. Наприклад, перше, що спадає на думку, це занадто багато інформації для початківця та купа непотрібного функціоналу, такого як обрати музику для занять чи цитати на кожен день. На мою думку, це зовсім непотрібно, оскільки відволікає від основної цілі: пізнання себе та свого тіла.

Саме тому, основним критерієм до мого додатку було те, щоб був мінімальний функціонал, який буде задовольняти критеріям зрозумілості. Для цього я обрала «Сповіщення», «Нотатки» і, звісно ж, вправи з йоги.

Вибір спинився на цьому функціоналі, оскільки, зазвичай, із власного досвіду, доводиться окремо занотовувати важливу інформацію, що стосується тренувань чи ставити собі сповіщення на телефон.

Також іноді заважає тренуванню реклама, котра з'являється час від часу і прибрати її не можна або ж лише купивши повну версію без реклами. Звісно, трапляється й те, що навчальні відео не можуть повністю загрузитися.

У створеному додатку всі вправи поділені на комплекси на окремі частини тіла. Наприклад: постава, спина, ноги чи вправи на гнучкість. Перевага додатку в простоті використання, тому що все чітко розділено, а кнопки розташовані так, щоб їх було легко знайти інтуїтивно.

1.3 Постановка завдання

Завданням роботи є:

- Створити Android-додаток
- Дослідити вплив йоги на організм людини
- Розглянути різні варіанти архітектури Android-додатоку та обрати найбільш оптимальний
- Дослідити компоненти Jetpack

- Розглянути уже раніше створені альтернативи-додатки

2 Проектування додатку

При проектуванні додатку були використані такі технології, як: liveData, Data repository pattern, View Model, MVVM pattern, Data binding library, Room database library, ConstraintLayout для створення інтерфейсу. Також додаток був створений у середовищі Android Studio з допомогою мови програмування Kotlin.

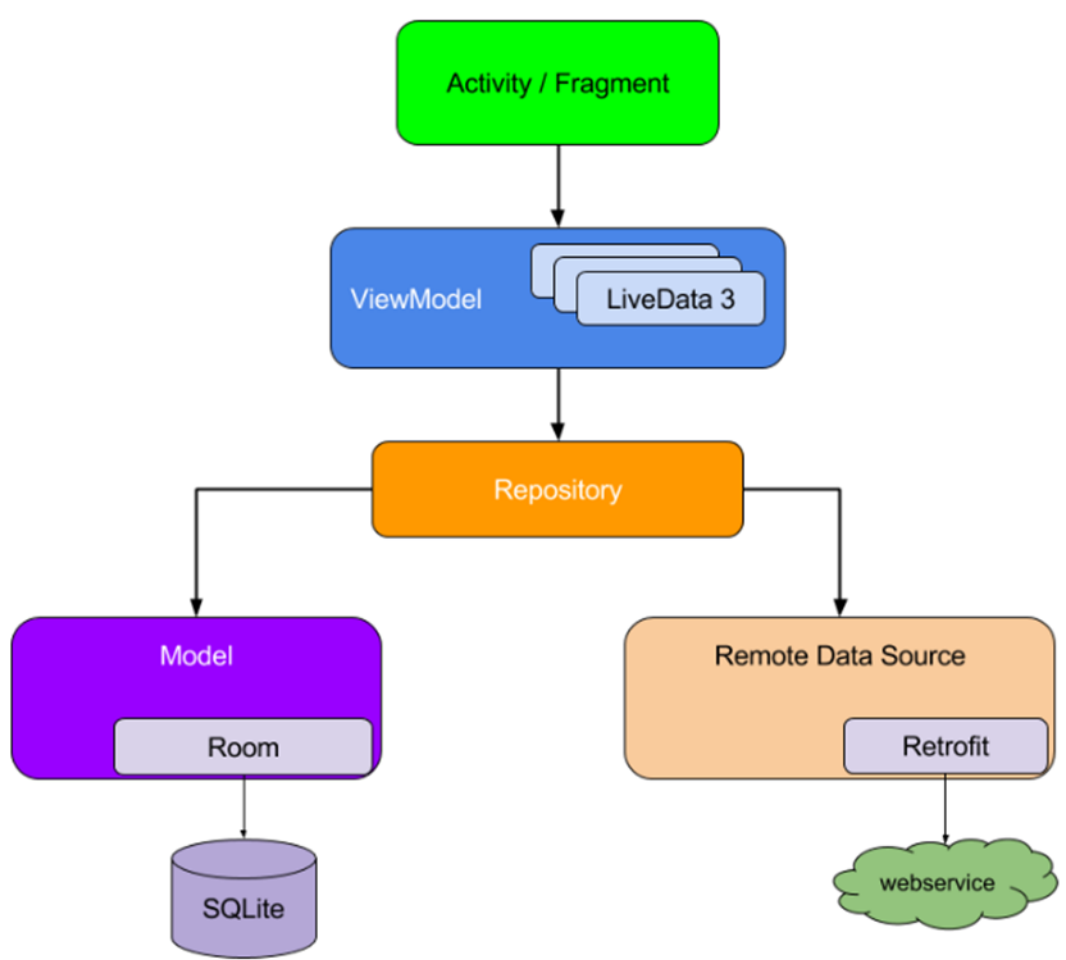


Рисунок 2.1 – Спроектована структура додатку

2.1Опис використаних технологій

-Android Jetpack

Jetpack - це набір бібліотек та інструментів, які допомагають розробникам легше створювати якісні програми. Ці компоненти допомагають дотримуватись рекомендацій, звільняють користувачів від написання стандартного коду, а також спрощують складні завдання, тому легко зосередитися на коді.

Jetpack включає в себе бібліотеки пакетів androidx. *, Відокремлені від API платформи. Він забезпечує зворотну сумісність і оновлюється частіше, ніж платформа Android, саме тому у користувача завжди є доступ до актуальних і найкращих версій компонентів Jetpack.

Компоненти Android Jetpack - це бібліотеки, котрі можна поєднувати та використовувати функціонал мови програмування Kotlin для підвищення продуктивності. Також вони створені для спільної роботи та можуть бути адаптовані індивідуально .[10]

Компоненти архітектури Android це набори бібліотек, котрі допомагають створювати надійні та підтримуючі додатки. Для початку роботи потрібно використовувати класи для управління життєвим циклом компонента інтерфейсу.

1. Нові компоненти, котрі враховують життєвий цикл, допомагають користувачу керувати своєю активністю і життєвими циклами. Спостерігати за змінами конфігурації, уникаючи витоків інформації, і легко завантажувати дані в призначений інтерфейс.

2. Для повідомлення і моніторингу про зміну інформаціїї в базі даних зручно використовувати об’єкти, що створюються LiveData.

3.ViewModel зберігає дані, що пов'язані з інтерфейсом, які не знищуються при поворотах в додатку.

4. Room - це бібліотека відображення об'єктів SQLite. Використовуючи цей компонент, можливо уникнути нечитабельного коду і легко перетворити дані таблиці SQLite в об'єкти Java. Room забезпечує перевірку часу компіляції операторів SQLite.

1. <u>Android Data Binding Library(частина Android Jetpack)</u>

Завдяки бібліотеці Data Binding Library можна прив'язати компоненти інтерфейсу в макетах до джерел даних в додатку за допомогою декларативного формату, а не програмно. Тобто Data Binding допомагає в роботі з View так, щоб користувачам не довелося писати занадто багато методів findViewById, setText, setOnClickListener і т.д.

Макети часто визначаються в діях з кодом, котрий викликає методи інфраструктури призначеного для інтерфейсу. Наприклад, наведений нижче код викликає findViewById () пошук TextView віджета і прив'язку його до userName властивості viewModel змінної:

```
findViewById < TextView > ( R . ID . образец _ текст ). apply {
    text = viewModel . userName
}
```

Рисунок 2.1.1 – Приклад коду

Нижче показано, як використовувати бібліотеку прив'язки даних для призначення тексту віджету. Це усуває необхідність виклику будь-якого з коду Java/Kotlin, показаного вище.

```
<TextView
    android:text="@{viewmodel.userName}" />
```

Рисунок 2.1.2 – Приклад коду

З'єднання компонентів у файлі макета дозволяє видаляти багато викликів інфраструктури інтерфейсу в діях користувача, роблячи їх простіше в обслуговуванні. Це також може поліпшити продуктивність вашого додатку і запобігти витоку пам'яті.

Бібліотека прив'язки даних автоматично генерує класи, необхідні для зв'язування уявлень в макеті з об'єктами даних. Бібліотека надає такі функції, як імпорт, змінні і включення, які користувач можете використовувати в своїх макетах.

Ці функції бібліотеки легко співіснують з існуючими макетами. Наприклад, змінні прив'язки, які можна використовувати у виразах, визначаються всередині data елемента, який є рідним елементом кореневого елемента макета призначеного для користувача інтерфейсу. Обидва елементи обгорнуті в layout тег, як продемонстровано нижче:

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto">
    <data>
        <variable
            name="viewmodel"
            type="com.myapp.data.ViewModel" />
    </data>
    <ConstraintLayout... /> <!-- UI layout's root element -->
</layout>
```

Рисунок 2.1.3 – Приклад коду

Робота з об'єктами даних, що спостерігаються

Бібліотека прив'язки даних надає класи і методи для простого спостереження за змінами. Програмісту не потрібно турбуватися про оновлення призначеного для користувача інтерфейсу при зміні базового джерела даних. Бібліотека дозволяє зробити об'єкти, поля або колекції видимими.

Згенеровані класи прив'язки

Бібліотека прив'язки даних генерує класи прив'язки, котрі використовуються для доступу до змінних і уявленням макета.

Зв’язуючі адаптери

Для кожного виразу макета існує адаптер прив'язки, який робить виклики платформи необхідними для установки відповідних властивостей або слухачів. Наприклад, адаптер прив'язки відповідає про виклик setText () методу для установки властивості text або виклику setOnClickListener ().

Прив'язка видів компонування до компонентів архітектури

Бібліотека підтримки Android включає компоненти архітектури, які можна використовувати для розробки надійних, кого тестують і програм, які підтримуються. Можна використовувати компоненти архітектури з бібліотекою прив'язки даних, щоб спростити розробку інтерфейсу призначеного для користувача.

Двостороння прив'язка даних

Бібліотека прив'язки даних підтримує двосторонню прив'язку даних. Нотація, що використовується для цього типу прив'язки, підтримує можливість отримання змін у властивостях даних і одночасного спостереження за оновленнями.

2. View Model

ViewModel клас призначений для зберігання і управління даними, пов'язаними з призначеним для користувача інтерфейсом, а також враховує життєвий цикл. Клас дозволяє отримувати дані, щоб залишатися при змінах конфігурації, наприклад, як екран навігації чи обертання. ViewModel і LiveData допомагають зберігати стан активності при змінах конфігурації. Навіщо зберігати стан активності? Створивши додаток, у макет екрану додамо поле для введення тексту і кнопку. Також залишимо тут текстове поле за замовчуванням. Переконаємося, що кожен елемент макета екрану має ідентифікатор. За натисканням кнопки будемо відправляти набраний текст з поля editText в поле textView.[8]

```
1 button.setOnClickListener {
2     textView.text = editText.text
3 }
4
```

Рисунок 2.1.4 – Приклад коду

Активіті знищується і створюється заново не тільки при повороті пристрою, а при різних змінах конфігурації, якщо не зберігати стан і тоді дані текстового поля будуть втрачені. Для збереження тексту в textView можна скористатися об'єктом savedInstanceState, який представлений в якості параметра функції onCreate (). Об'єкт savedInstanceState має тип Bundle, котрий представляє собою набір пар "ключ - значення" і може бути використаний для збереження попереднього стану активіті.

У випадку якщо потрібно зберегти список значень, а не одне значення, то при зміні конфігурації пристрою дані будуть знищені і їх доведеться завантажувати заново.

Потрібно пам’ятати, що контролерам інтерфейсу часто доводиться виконувати асинхронні виклики, для повернення яких може знадобитися деякий

час. Контролер UI повинен керувати цими викликами і очищати їх після знищення, щоб уникнути можливих витоків пам'яті. Це управління вимагає великого обслуговування, і в разі, коли об'єкт відтворюється для зміни конфігурації, це марна трата ресурсів, оскільки об'єкту доведеться повторити виклики, котрі були зроблені раніше.

Контролери інтерфейсу призначені для відображення даних інтерфейсу, реагування на дії користувача або обробки взаємодії з операційною системою. Присвоєння надмірної відповідальності контролерам інтерфейсу користувача може привести до того, що один клас намагається виконувати всі поставлені задачі, а не поділити їх. Таким чином, покладання надмірної відповідальності на контролери значно ускладнює тестування.

Компоненти Android Jetpack надають допоміжний клас ViewModel для контролера, який відповідає за підготовку даних для інтерфейсу. Об'єкти ViewModel автоматично зберігаються під час змін конфігурації так, що дані, котрі містяться в них відразу ж стають доступні для наступного активіті або фрагменту.

На рисунку нижче наведено як ViewModel взаємодіє з життєвим циклом активіті. Представлені різні стани життєвого циклу дії та показано час життя ViewModel.

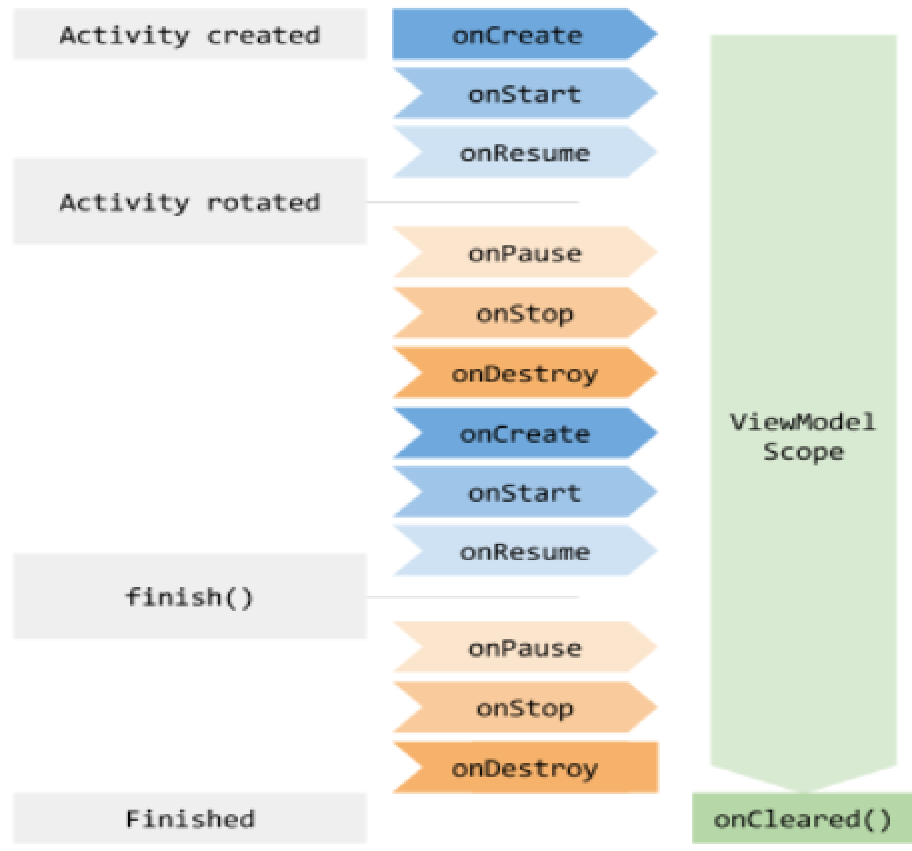


Рисунок 2.1.5 – Приклад життєвого циклу активіті

При перестворенні активіті ViewModel залишається активним і використовується в новоствореному активіті.

Наприклад, якщо програмісту потрібно відобразити список користувачів у додатку, потрібно реалізувати отримання і збереження списку користувачів не в коді активіті, а у ViewModel.

ViewModel об'єкти призначені для того, щоб пережити конкретні екземпляри уявлень або LifecycleOwners. ViewModel об'єкти можуть містити LiveData об'єкти.

<u>Обмін даними між об'єктами:</u>

Дуже часто певна кількість фрагментів повинні тримати зв'язок один з одним. Якщо уявити собі загальний випадок фрагментів основної деталі, коли є фрагмент, в якому користувач вибирає елемент зі списку, а інший фрагмент,

котрий відображає вміст вибраного елемента. Обидва фрагменти повинні визначати опис інтерфейсу, а дія користувача повинна пов'язувати їх разом.

Ця проблема може бути вирішена за допомогою ViewModel об'єктів. Ці фрагменти можуть спільно ViewModel використовувати свою область дії для обробки цієї взаємодії.

Заміна завантажувача за допомогою ViewModel

Такі класи завантажувачів використовуються для синхронізації даних в інтерфейсі додатка з базою даних. Можна використовувати ViewModel з кількома іншими класами, щоб замінити завантажувач. Використання ViewModel відокремлює контролер, що призначений для інтерфейсу користувача, від операції завантаження даних, це означає, менше надійних посилань між класами.

В одному з поширених підходів до використання завантажувачів додаток може використовувати CursorLoader для спостереження за вмістом бази даних. Коли значення в базі даних змінюється, завантажувач автоматично запускає перезавантаження даних і оновлює інтерфейс:

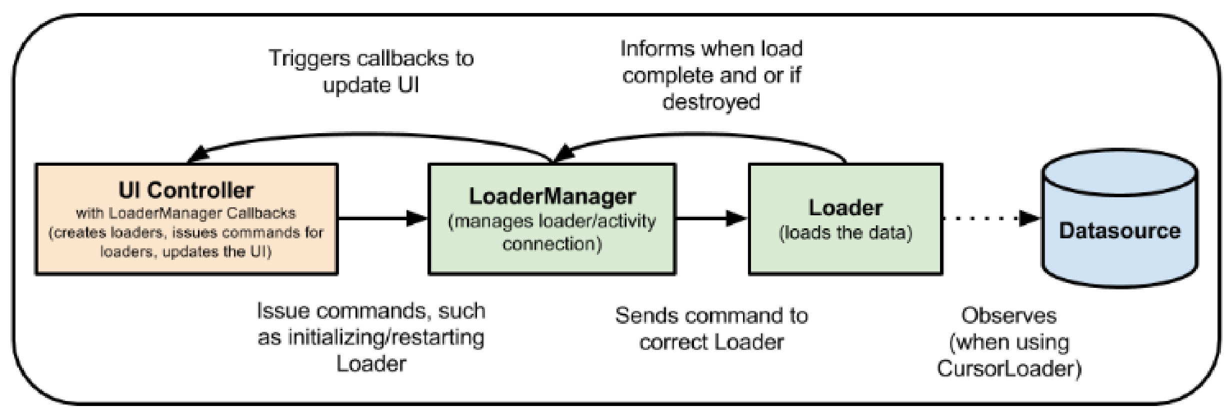


Рисунок 2.1.6 – Приклад завантаження даних з допомогою завантажувачів

ViewModel працює з Room і LiveData для заміни завантажувача. ViewModel гарантує, що дані виживають при зміні конфігурації пристрою. Room

інформує користувача LiveData про зміну бази даних, а LiveData, в свою чергу, оновлює інтерфейс зі зміненими даними. [8]

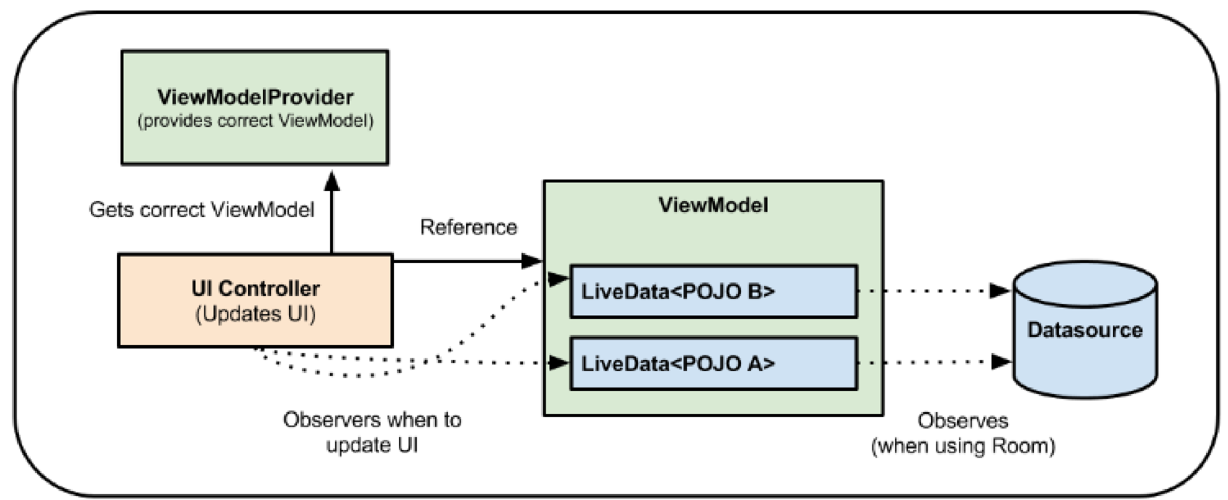


Рисунок 2.1.7 – Приклад завантаження даних за допомогою ViewModel

3.MVVM pattern

Model-View-ViewModel (тобто MVVM) - це шаблон архітектури клієнтських додатків, який був запропонований Джоном Госсманом (John Gossman) як альтернатива шаблонами MVC і MVP при використанні технології зв'язування даних (Data Binding). Його концепція полягає у відділенні логіки представлення даних від бізнес-логіки шляхом винесення її в окремий клас для більш чіткого розмежування.

Назва складається з трьох частин:

Model - описує використовувані в додатку дані. Моделі можуть містити логіку, безпосередньо пов'язану з цими даними, наприклад, логіку валідації властивостей моделі. У той же час модель не повинна містити ніякої логіки, пов'язаної з відображенням даних і взаємодією з візуальними елементами управління.

View - власне, це і є layout екрану, в якому розташовуються всі необхідні віджети для відображення інформації.

ViewModel - об'єкт, в якому описується логіка поведінки View в залежності від результату роботи Model. Можна назвати його моделлю поведінки View. Це може бути як форматування тексту, так і логіка управління видимістю компонентів або відображення станів, таких як завантаження, порожні екрани і

т.д. Також в ній описується поведінка, яке було ініційовано користувачем (введення тексту, натискання на кнопку чи свайп). [4]

Model-View-ViewModel - це шаблон рівня уявлення, який дуже добре працює з Data Binding. Ось діаграма патерну MVVM:

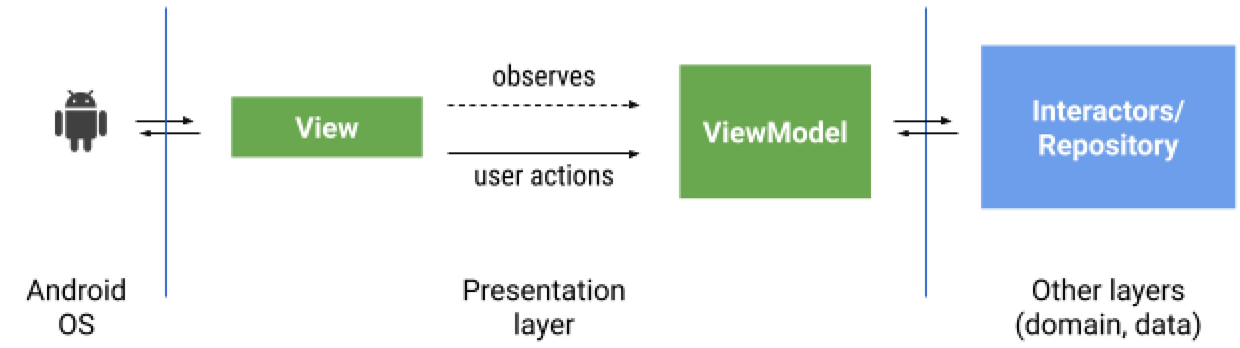


Рисунок 2.1.8 – Шаблон Model-View-ViewModel

Якщо ж розглянути що дає MVVM, то можна вивести декілька пунктів:

Гнучкість розробки. Цей підхід підвищує зручність роботи в команді, тому поки один член команди працює над стилізацією екрану - інший, паралельно, описує логіку отримання даних і їх обробки;

Тестування. Така структура спрощує написання тестів і процес створення об'єктів. Також, в більшості випадків не потребується автоматизоване UI-тестування, тому що можна обернути unit-тестами сам ViewModel;

Розмежування логіки. За рахунок більшого розмежування код стає більш гнучким і простим у підтримці і, звісно, лаконічним і читабельним. Кожен модуль відповідає за свою конкретну функцію.

4.Data repository pattern

Патерн від англійського слова Pattern - зразок, шаблон. У програмуванні це поняття має на увазі використання певного підходу чи алгоритму, який вже існує для вирішення проблеми в тій чи іншій ситуації.

Одним з найбільш часто використовуваних патернів при роботі з даними є патерн 'Repository pattern'. Репозиторій дозволяє абстрагуватися від конкретних

підключень до джерел даних, з якими працює програма, і є проміжною ланкою між класами, котрі безпосередньо взаємодіють з даними, і рештою програми.

Ймовірно, найбільш важливою відмінністю репозиторіїв є те, що вони являють собою колекції об'єктів. Репозиторії представляють колекції. Як зберігати колекції - це просто деталь реалізації. Колекція, котра містить сутності та може фільтрувати і повертати результат назад в залежності від вимог програми .[2]

У світі програмування програмісти звикли до циклу запит / відповідь, котрий завершується. Все, що прийшло ззовні і не збереглося - пішло назавжди, в цій точці, але не всі платформи працюють саме так.

Простіше кажучи, репозиторій - це особливий вид надійних колекцій, які можна використовувати знову, щоб зберігати і фільтрувати суті.

Припустимо, є одне підключення до бази даних MS SQL Server. Однак, якщо в якийсь момент часу захотіти змінити підключення з MS SQL до бд MySQL. При стандартному підході навіть у невеликому додатку, що здійснює вибірку, додавання, зміну та видалення даних, довелося б зробити велику кількість змін. Або в процесі роботи програми в залежності від різних умов ми хочемо використовувати два різних підключення. Таким чином, репозиторій додає програмі гнучкість в роботі з різними типами підключень.

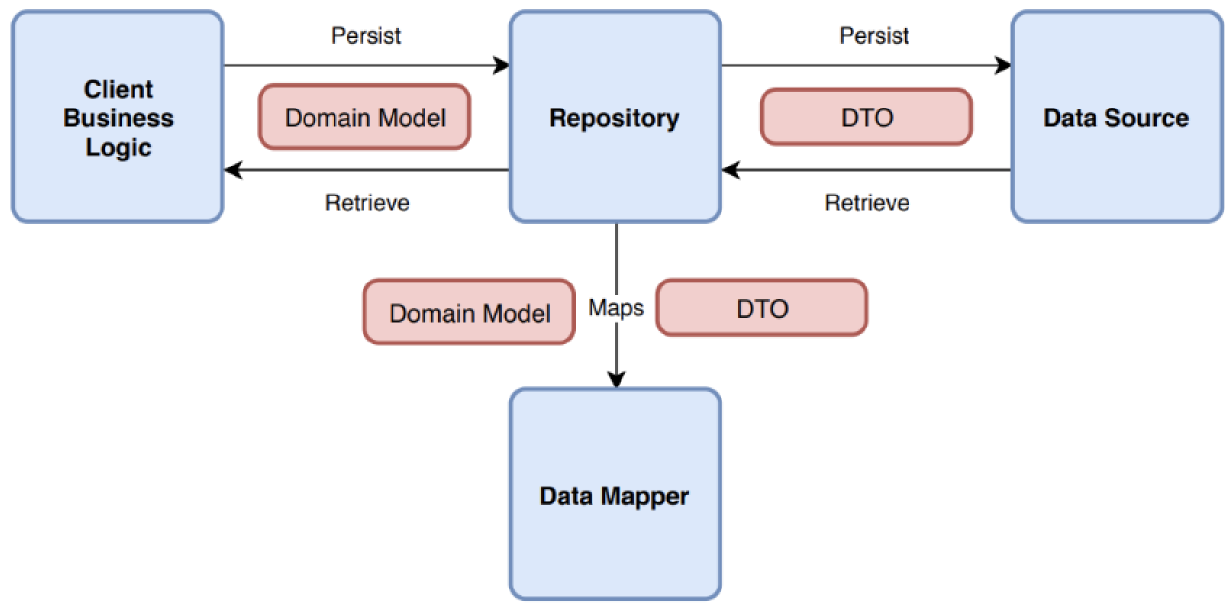


Рисунок 2.1.9 – Приклад Data repository pattern

Ключовий момент шаблону сховища в тому, що потрібен домен і розробники щосили намагаються зробити це. Доменні моделі в основному представляють бізнес-правила для всього підприємства.

Існує 3 види доменних моделей:

Сутність: сутність - це простий об'єкт, який має ідентифікатор (ID) і є потенційно змінним.

Об'єкт значення: незмінний об'єкт без ідентичності.

Спільний корінь: Об'єкт, який зв'язується разом з іншими об'єктами (в основному кластер пов'язаних об'єктів).

У простих доменах ці моделі будуть дуже схожі на моделі баз даних і мереж, але у них все ще буде багато відмінностей:

- Доменні моделі змішують дані і процеси, і їх структура найбільше підходить для додатку.

- Модель предметної області вільна від каркасів, а її структура підтримує багатозначні атрибути і використовує шаблон порожніх об'єктів (поля НЕ обнуляються).[2]

Завдяки цьому поділу:

1) розробка застосування стає простіше, оскільки не потрібно перевіряти нульові значення, а багатозначні атрибути не змушують нас відправляти всю модель цілком.

2) зміни в джерелах даних не впливають на наші політики високого рівня.

3) погані реалізації Backend не впливають на наші політики високого рівня

5.Room database library

Room - це бібліотека зіставлення об'єктів бази даних, яка спрощує доступ до бази даних в додатках Android. Бібліотека Room забезпечує рівень абстракції над SQLite, щоб забезпечити більш надійний доступ до бази даних при одночасному використанні всіх можливостей SQLite.

Бібліотека допомагає створити кеш даних на пристрої, на якому виконується ваша програма. Цей кеш, який дозволяє користувачам переглядати наповнення програми, незалежно від того, чи є у користувачів підключення до Інтернету.

Додатки, які обробляють нетривіальні обсяги структурованих даних, можуть значно виграти від локального збереження цих даних. Найбільш поширеним варіантом використання є кешування відповідних частин даних. Таким чином, коли пристрій не може отримати доступ до мережі, користувач все ще може переглядати цей контент, поки він знаходиться в автономному режимі. Будь-які зміни вмісту, ініційовані користувачем, потім синхронізуються з сервером після того, як пристрій знову підключається.[7]

В Room 3 основних компоненти:

База даних: містить тримач бази даних і є основною точкою доступу для основного з'єднання з постійними реляційними даними вашого застосування.

Клас @Database повинен відповідати таким вимогам:

1) Бути абстрактним класом, який розширюється RoomDatabase.

2) Включити анотацію до списку об'єктів, пов'язаних із базою даних.

3) Містить абстрактний метод, який має 0 аргументів і повертає клас, позначений як @Dao.

Під час виконання ви можете отримати екземпляр Database, викликавши Room.databaseBuilder () або Room.inMemoryDatabaseBuilder ().

Сутність: представляє таблицю в базі даних.

DAO: містить методи, що використовуються для доступу до бази даних.[7]

Додаток використовує базу даних Room для отримання доступу до даних об'єктів або DAO, пов'язаних з цією базою даних. Потім додаток використовує кожен DAO для отримання сутностей з бази даних і збереження будь-яких змін цих сутностей назад в базу даних. Нарешті, додаток використовує об'єкт для отримання і установки значень, що відповідають стовпцям таблиці в базі даних.

Зв’язок між різними складовими Room ілюструється на рисунку:

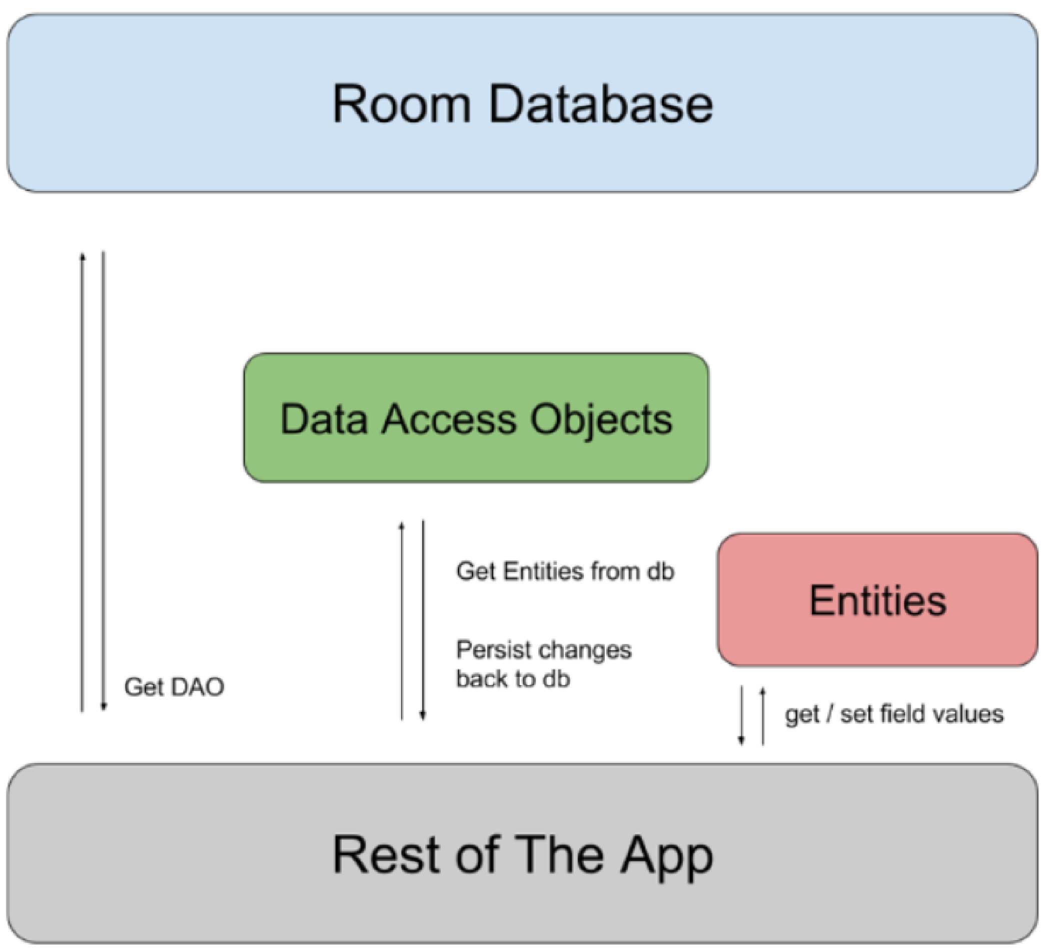


Рисунок 2.1.10 – Схема роботи Room DataBase

6. Створення додатку для Android з використанням складових архітектури Android: Room, ViewModel та LiveData.

Компоненти архітектури Android є набором бібліотек, котрі допомагають розробляти надійні, тестовані і підтримувані додатки з більш широкими можливостями управління життєвим циклом і збереженням даних.

Для дипломної роботи було розроблено програму, котра приймає деякі дані від користувача (LiveData), зберігає їх у локальній базі даних (Room) і відображає його на екрані (ViewModel).

Етапи реалізації

Потрібно дотримуватися цих кроків для реалізації компонентів архітектури Android в нашому додатку:

1) Додати Залежності

Нам потрібно додати компоненти Room і Lifecycle. В основному, Room - це бібліотека відображення об'єктів бази даних, яка використовується для доступу до бази даних. У життєвому циклі є набір класів, таких як ViewModel і LiveData, які ми будемо використовувати для управління життєвим циклом нашого застосування.

2) Встановлення кімнати(Room)

3) Live Data

LiveData з бібліотеки життєвого циклу для спостереження за змінами даних. Це спостерігається клас даних, і він також обізнаний про життєвий цикл, що означає, що він буде оновлювати компонент, який знаходиться в стані активного життєвого циклу.

4) Створення класу сховища / шару уявлення

Це клас, в якому ми будемо перевіряти, чи одержуємо дані з API або локальної бази даних, або що ми поміщаємо логіку вибірки з бази даних в цей клас.

5) Реалізація ViewModel

Це також частина бібліотеки життєвого циклу. Це допоможе надати дані між сховищем і призначеним для користувача інтерфейсом. Це зберігає дані про зміни конфігурації і змушує існуючу ViewModel повторно з'єднатися з новим екземпляром власника.

6) Додати RecyclerView

Створюємо макет програми та додаємо основні activity.

7) Заповнюємо базу даних

Тут ми заповнюємо дані під час запуску програми, а до цього ми також видаляємо існуючі дані.

8) З’єднуємо інтерфейс і дані

Щоб відобразити дані з бази даних, нам потрібен спостерігач, який буде спостерігати за змінами даних, LiveData в ViewModel.

9) Створити AddNoteActivity

Ми створюємо Activity, де користувач може вводити дані. Тому ми створюємо AddNoteActivity. Цими Activity є кнопки для відправки даних, поля для заповнення.

7. Live Data

LiveData - це клас даних, який можна спостерігати в рамках даного життєвого циклу. LiveData враховує життєвий цикл, що означає, що він враховує життєвий цикл інших компонентів додатка, таких як дії, фрагменти або служби. Це гарантує, що LiveData оновлює тільки ті компоненти програми, які знаходяться в стані активного життєвого циклу. Це означає, що Observer може додаватися в парі з a LifecycleOwner, і цей спостерігач буде повідомляти про зміни упакованих даних, тільки якщо спарений LifecycleOwner знаходиться в активному стані. LifecycleOwner вважається активним, якщо його статус дорівнює STARTED або RESUMED. Спостерігач, доданий через observeForever (Observer), вважається завжди активним і, таким чином, завжди буде повідомлений про зміни. [1]

Спостерігач, доданий за допомогою життєвого циклу, буде автоматично видалений, якщо відповідний життєвий цикл переходить в стан DESTROYED. Це особливо корисно для дій і фрагментів, де вони можуть безпечно спостерігати LiveData і не турбуватися про витоки: вони будуть негайно відписані після їх знищення. Крім того, LiveData має onActive () і onInactive () методи для отримання повідомлень, коли число активних Observers змінюється від 0 до 1.

Це дозволяє LiveData вивільняти будь які ресурси, коли у нього немає спостерігачів, які активно спостерігають. Цей клас призначений для зберігання окремих полів даних ViewModel, але також може використовуватися для спільного використання даних між різними модулями в додатку.

Використання LiveData гарантує наступні переваги:

1) Гарантує, що ваш інтерфейс відповідає вашому стану даних

LiveData повідомляє Observer об'єкти про зміну стану життєвого циклу. Можна об'єднати свій код для поновлення інтерфейсу корисувача в цих Observer об'єктах. Замість оновлення призначеного для інтерфейсу користувача при кожній зміні даних додатка, спостерігач може оновлювати призначений для користувача інтерфейс при кожній зміні.

2) Немає витоків пам'яті

Спостерігачі прив'язуються до Lifecycle об'єктів і прибирають за собою, коли пов'язаний з ними життєвий цикл руйнується.

3) Немає збоїв через припинення діяльності

Якщо життєвий цикл спостерігача неактивний, наприклад, в разі активності в задньому стеку, він не отримує ніяких подій LiveData.

4) Немає ручної обробки життєвого циклу

Компоненти для інтерфейсу користувача просто спостерігають відповідні дані і не зупиняють і не відновлюють спостереження. LiveData автоматично керує всім цим, оскільки під час спостереження він знає про зміни стану життєвого циклу.

5) Завжди в курсі даних

Якщо життєвий цикл стає неактивним, він отримує найостанніші дані після повторної активації. Наприклад, операція, яка була в фоновому режимі, отримує останні дані відразу після того, як повертається на передній план.

6) Правильні зміни конфігурації

Якщо дія або фрагмент відтворюється через зміну конфігурації, наприклад, ротації пристрою, він негайно отримує останні доступні дані.

7) Спільне використання ресурсів

Ви можете розширити LiveData об'єкт, використовуючи шаблон, щоб обернути системні служби, щоб вони могли бути використані в додатку. У LiveData об'єкт підключається до системної служби один раз, а потім будь-якому спостерігачеві, який потребує ресурс може просто спостерігати за LiveData об'єкт.

<u>Робота з об’єктами LiveData</u>

1) Створіть екземпляр LiveData для зберігання даних певного типу. Зазвичай, це робиться в ViewModel класі.

2) Створіть Observer об'єкт, який визначає onChanged () метод, який керує тим, що відбувається, коли в LiveData зберігаються дані об'єкта. Зазвичай, ви створюєте Observer об'єкт в контролері призначеного для користувача інтерфейсу, наприклад, дія або фрагмент.

3) Прикріпіть Observer об'єкт до LiveData об'єкту, використовуючи observe () метод. Цей метод приймає LifecycleOwner об'єкт. Це підписує Observer об'єкт на LiveData об'єкт, в слід чого йому повідомляється про зміни. Зазвичай приєднують Observer об'єкт в контролері призначеного для користувача інтерфейсу, наприклад, в діяльності або фрагменті.[3]

Коли оновлюється значення, збережене в LiveData об'єкті, він запускає всіх зареєстрованих спостерігачів, поки приєднаний LifecycleOwner знаходиться в

активному стані. LiveData дозволяє спостерігачам контролера призначеного для інтерфейсу користувача підписатися на оновлення. Коли дані, що містяться в LiveData об'єкті, змінюються, призначений для користувача інтерфейс автоматично оновлюється у відповідь.

Спостереження за об'єктами LiveData

У більшості випадків onCreate () метод компонента додатка є підходящим місцем для початку спостереження LiveDataоб'єкта з наступних причин:

1. Щоб система не виконувала надлишкові виклики з onResume () методу дії або фрагмента.

2. Щоб переконатися, що у дії або фрагмента є дані, які вони можуть відобразити, як тільки вони стануть активними. Як тільки компонент програми знаходиться в STARTED стані, він отримує останнє значення від LiveData об'єктів, які він спостерігає. Це відбувається тільки в тому випадку, якщо LiveData об'єкт був встановлений.

Як правило, LiveData доставляє оновлення тільки при зміні даних і тільки для активних спостерігачів. Винятком з цього поведінки є те, що спостерігачі також отримують оновлення, коли вони переходять з неактивного в активний стан. Крім того, якщо спостерігач змінюється з неактивного на активний вдруге, він отримує оновлення тільки в тому випадку, якщо значення змінилося з моменту останнього активування.

Оновлення об'єктів LiveData

LiveData не має загальнодоступних методів для оновлення збережених даних. MutableLiveData Клас виставляє setValue (T) і postValue (T) методи публічно, і ви повинні використовувати їх, якщо вам потрібно змінити значення, яке зберігається в LiveData об'єкті. Зазвичай MutableLiveData використовується в ViewModel і ViewModel тільки тоді виставляє незмінні LiveData об'єкти для

спостерігачів. Після того, як встановили спостерігача, можна оновити значення LiveData об'єкту.

Використання LiveData з кімнатою(Room)

Бібліотека сталості Room підтримує запити, які повертають LiveData об'єкти. Спостережувані запити пишуться як частина об'єкта доступу до бази даних (DAO).

Room генерує весь необхідний код для поновлення LiveData об'єкта при оновленні бази даних. Згенерований код виконує запит асинхронно у фоновому потоці, коли це необхідно. Цей шаблон корисний для синхронізації даних, що відображаються в інтерфейсі, з даними, що зберігаються в базі даних.

Об'єднати декілька джерел LiveData

MediatorLiveData является подклассом, LiveData котрий дозволяє об'єднувати декілька джерел LiveData. Потім спостерігачі об'єктів запускаються щоразу, коли змінюється будь-якої з вихідних об'єктів LiveData.

Наприклад, якщо в LiveData інтерфейсі є об'єкт, який можна оновити з локальної бази даних або мережі, ви можете додати до цього MediatorLiveData об'єкту наступні джерела:

-LiveData Об'ект, пов'язаний з даними, що зберігаються в базі даних.

-LiveData Об'ект, пов'язаний з даними доступу з мережі.

Діяльність повинна тільки спостерігати за MediatorLiveData об'ектом, щоб отримувати оновлення з обох джерел.[3]

8. Constaint Layout

Для побудови адаптивного інтерфейсу був використаний Constaint Layout. ConstraintLayout дозволяє створювати великі і складні макети без вкладених груп уявлень. Усі уявлення розташовуються відповідно до відносин між

спорідненими поглядами й батьківським макетом, але це більш гнучкий RelativeLayout і простий у використанні редактор макетів Android Studio. Вся міць ConstraintLayout доступна безпосередньо з візуальних інструментів редактора макетів, тому що API макета і редактор макетів були спеціально створені одне для одного. Таким чином, ви можете створити свій макет ConstraintLayout цілком, перетягуючи його замість редагування XML.

<u>Огляд обмежень</u>

Щоб визначити позицію виду в ConstraintLayout, потрібно додати принаймні одне горизонтальне і одне вертикальне обмеження. Кожне обмеження є з'єднанням або вирівнювання з іншим видом, батьківським макетом. Кожне обмеження визначає положення виду уздовж вертикальної або горизонтальної осі; тому кожна вистава має мати як мінімум одне обмеження для кожної осі. Проте, це лише для полегшення редагування, якщо при запуску макета на пристрої уявлення не має обмежень, воно малюється в позиції [0,0] (верхній лівий кут).

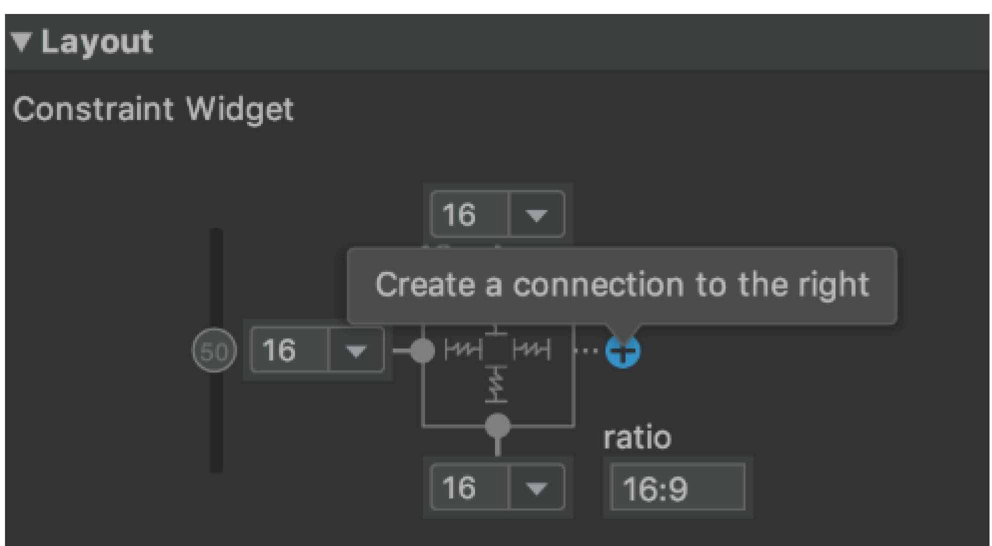


Рисунок 2.1.12 – Приклад обмежень активіті

9. Navigation UI

Компонент навігації включає в себе NavigationUI клас. Цей клас містить статичні методи, котрі керують навігацією за допомогою верхньої панелі програми, панелі навігації та нижньої навігації.

Верхня панель додатку. Бар топ додаток забезпечує послідовне місце вздовж верхньої частини програми для відображення інформації і дій з поточного екрану.

Рисунок 2.1.13 – Скрін верхньої панелі додатку

NavigationUI містить методи, які автоматично оновлюють контент у верхній панелі програми, коли користувачі переміщаються в додатку. Наприклад, NavigationUI використовує мітки призначення на графіку навігації, щоб підтримувати актуальність заголовка верхньої панелі програми. При використанні NavigationUI з реалізаціями верхньої панелі програми, мітка, яку прикріплюють до адресатів, може автоматично заповнюватися з аргументів, наданих адресату, використовуючи формат {argName} у мітці.

NavigationUI використовує AppBarConfiguration об'єкт для управління поведінкою кнопки навігації у верхньому лівому кутку області відображення. Поведінка навігаційної кнопки змінюється в залежності від того, чи знаходиться користувач в пункті призначення верхнього рівня. Місця призначення верхнього рівня не відображають кнопку «Вгору» на верхній панелі програми, оскільки там немає місця призначення більш високого рівня. За замовчуванням пункт призначення є єдиним пунктом призначення верхнього рівня.

Коли користувач знаходиться в пункті призначення верхнього рівня, кнопка навігації стає значком ящика, якщо пункт призначення використовує значок DrawerLayout. Якщо пункт призначення не використовує DrawerLayout, кнопка навігації прихована. Коли користувач знаходиться в будь-якому іншому місці призначення, кнопка навігації відображається як кнопка «Вгору». Щоб налаштувати кнопку навігації, використовуючи тільки початковий пункт призначення як кінцеву точку маршруту верхнього рівня, потрібно створити AppBarConfiguration об'єкт.

У деяких випадках може знадобитися визначити кілька пунктів призначення верхнього рівня замість використання пункту призначення за замовчуванням. Використання а BottomNavigationView є поширеним випадком для цього, коли в додатку можуть бути родинні екрани, які не мають ієрархічного зв'язку один з одним, і кожен з них може мати свій власний набір пов'язаних адресатів. Додавання верхньої панелі програми до активіті добре працює, коли макет панелі програми однаковий для кожного місця призначення в додатку.

Наприклад, один з пунктів призначення може використовувати стандарт Toolbar, а інший - AppBarLayout для створення більш складної панелі додатків з вкладками, як показано на рисунку:

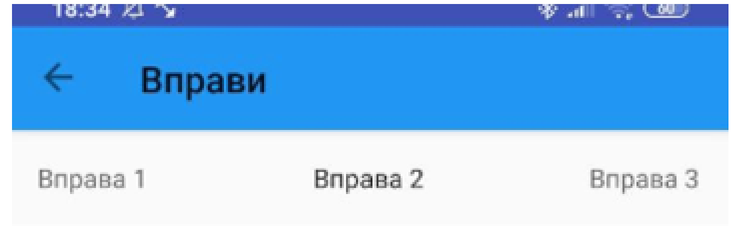


Рисунок 2.1.14 – Скрін навігації в додатку

Панель навігації – це панель інтерфейсу користувача, в якій відображається головне меню навігації. Ящик з'являється, коли користувач торкається значка ящика на панелі додатків або коли користувач проводить пальцем по лівому краю екрану.

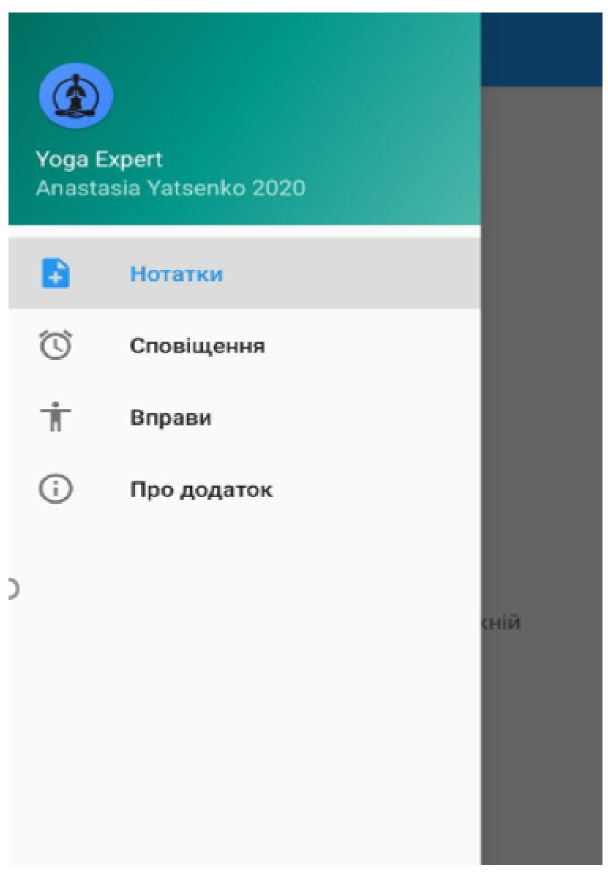


Рисунок 2.1.15 – Скрін панелі навігації

10.Kotlin

Kotlin - статично тіпізованій мова програмування, что працює поверх Java Virtual Machine и розробляється компанією JetBrains. Також компілюється в JavaScript і в виконуваний код ряду платформ через інфраструктуру LLVM.

Автори ставили за мету – створити мову більш лаконічну, ніж Java, і простішу Scala. Наслідком спрощення в порівнянні зі Scala стали більш швидка компіляція і найкраща підтримка мови в IDE. Мова повністю сумісна з Java, що дозволяє java-розробникам поступово перейти до його використання; зокрема, в Android мову вбудовується за допомогою Gradle, що дозволяє для існуючого

android-Додатки впроваджувати нові функції на Kotlin без переписування програми повністю.[5]

11. Kotlin vs Java

-Перевіряючі виключення

Одне з основних відмінностей між Java і Kotlin полягає в тому, що в останньому немає умов для перевіряючих винятків (checked exception). Отже, немає необхідності відловлювати або оголошувати будь-які винятки.

Якщо розробник, який працює на Java, вважає, що використання коду try / catch в коді дратує, то упущення, зроблене Kotlin, можна вважати бажаним зміною. Однак, протилежністю буде, якщо розробник вважає, що перевіряються винятку потрібні, сприяючи відновленню після помилок і створення надійного коду. У цьому випадку це можна вважати для Kotlin плюсом і мінусом, в залежності від підходу до розробки.

-Стислість коду

Порівняння класу Java з еквівалентним класом Kotlin демонструє лаконічність коду Kotlin. Для тієї ж операції, що виконується в класі Java, клас Kotlin вимагає менше коду.

Наприклад, конкретний сегмент, де Kotlin може значно скоротити загальний обсяг стандартного коду, - це findViewByIds. Розширення Kotlin в Android дозволяють імпортувати посилання на View в файл Activity. Це дає можливість працювати з цією виставою, як якщо б воно було частиною Activity. Це явно можна віднести до плюсів Котлін.

-Класи даних

У повнорозмірних проектах зазвичай є кілька класів, призначених виключно для зберігання даних. Хоча ці класи практично не мають функціональності, розробнику необхідно написати багато стандартного коду на

Java. Зазвичай розробник повинен визначити конструктор і кілька полів для зберігання даних, функції геттери і сеттери для кожного з полів, а також функції equals (), hashCode () і toString ().

У Kotlin є дуже простий спосіб створення таких класів. Розробнику досить включити тільки ключове слово data в визначення класу, і все - компілятор сам подбає про все. Така зручність створення класів, в питаннях для Kotlin «за і проти», явно свідчить на його користь.

-Вбудовані функції

Змінні, до яких здійснюється доступ в тілі функції, називаються замиканнями. Використання функцій вищого порядку може істотно збільшити час виконання обчислень. Кожна функція в Kotlin є об'єктом, і він захоплює замикання.

І класи, і функції вимагають виділення пам'яті. Вони, поряд з віртуальними викликами, вимагають певних витрат на час виконання. Таких додаткових витрат можна уникнути, вставивши лямбда-вирази в Kotlin. Одним з таких прикладів є функція lock ().

На відміну від Kotlin, Java не забезпечує підтримку вбудованих функцій. Проте, компілятор Java здатний виконувати вбудовування з використанням методу final. Це так, тому що методи final не можуть бути перевизначені підкласами. Крім того, виклик методу final дозволяється під час компіляції.Таке нововведення також сприймаються як плюси Котлина.[5]

Android Studio

Android Studio - це інтегроване середовище розробки (IDE) для роботи з платформою Android, анонсована 16 травня 2013 року на конференції Google I / O. IDE перебувала у вільному доступі починаючи з версії 0.1, опублікованій в травні 2013, а потім перейшла в стадію бета-тестування, починаючи з версії 0.8, яка була випущена в червні 2014 року.

Android Studio, заснована на програмному забезпеченні IntelliJ IDEA від компанії JetBrains, - офіційний засіб розробки Android додатків. Дане середовище розробки доступне для Windows, OS X і Linux. 17 травня 2017, на щорічній конференції Google I / O, Google анонсував підтримку мови Kotlin, використовуваного в Android Studio, як офіційної мови програмування для платформи Android на додаток до Java і C ++.[9]

2.2 Проектування інтерфейсу

В Android Studio, щоб створити новий проект:

1. Якщо у вас немає відкритого проекту, в Welcome екрані запрошення, натисніть Start a new Android Studio Project.

2. Якщо у вас відкритий проект, в меню File, виберіть New> New Project.

3. На першому екрані Choose your project на вкладці Phone and Tablet виберіть Empty Activity або ж Basic Activity:

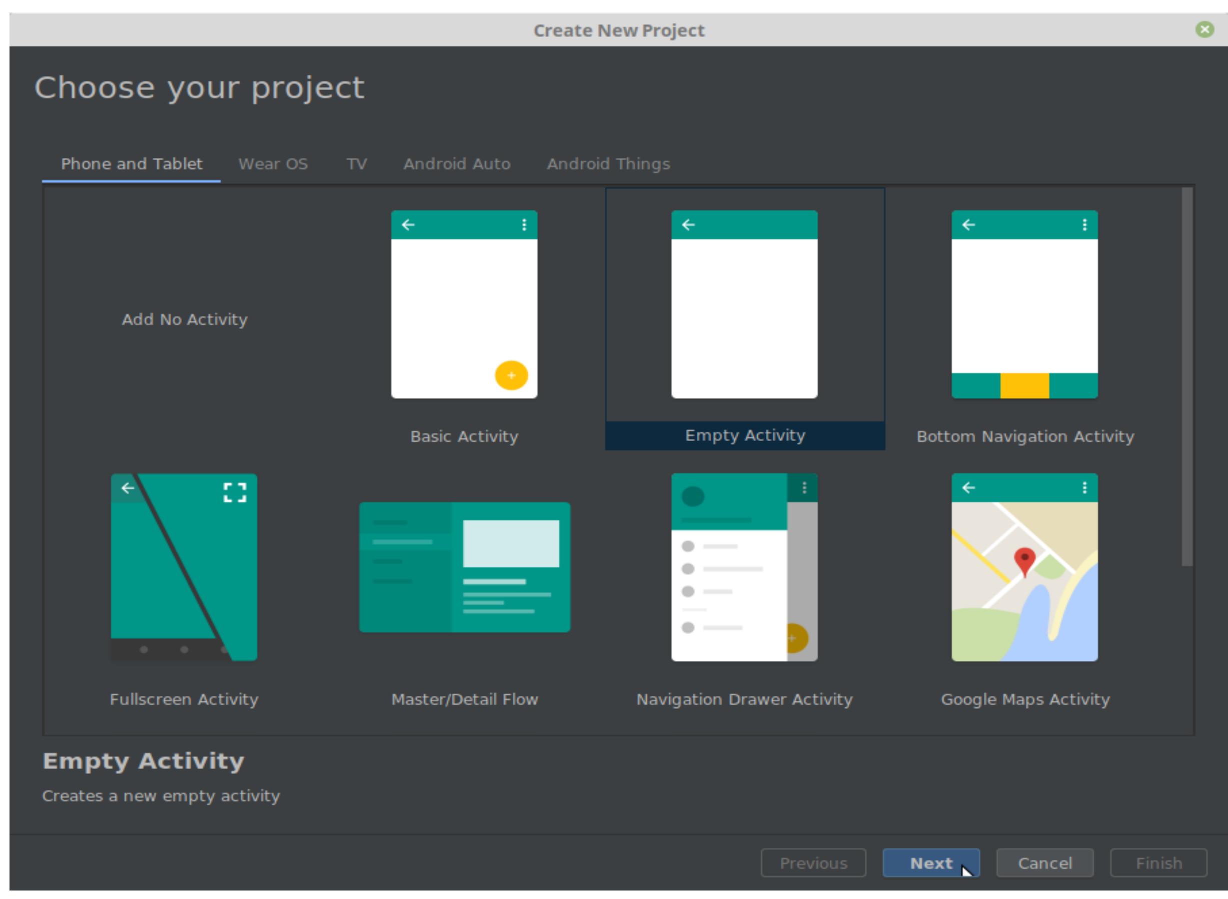


Рисунок 2.2.1 - Вигляд початкової сторінки в Android Studio

У цьому випадку середовище розробки створить порожній основний екран-activity.

Activity (їх ще називають діяльність або активність) є однією з відмінних рис Android framework. По суті, активи - це екрани андроїд програми. Коли користувач запускає додаток, стартує основний екран-activity, наприклад, зі списком листів у поштовій програмі, а коли користувач вибирає певний контент для перегляду, наприклад, лист в списку, то воно відкривається на іншому екрані-activity.

• Натисніть Next

• На наступному екрані Configure your project, заповніть поля:

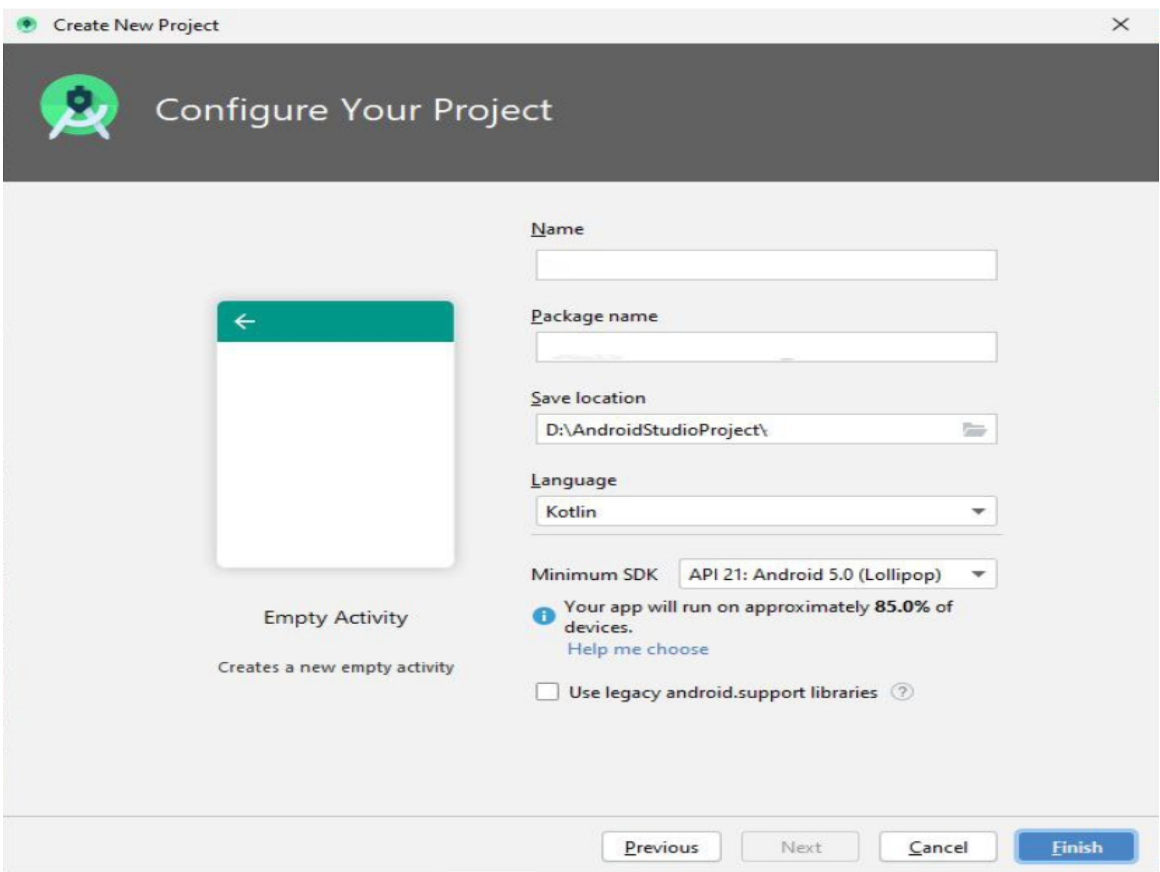


Рисунок 2.2.2 - Скріншот сторінки при створені проекту

Після цих кроків починається саме створення проекту за допомогою Kotlin.

2.3 Інтерфейс додатку

Нижче наведені скріншоти інтерфейсу створеного додатку.

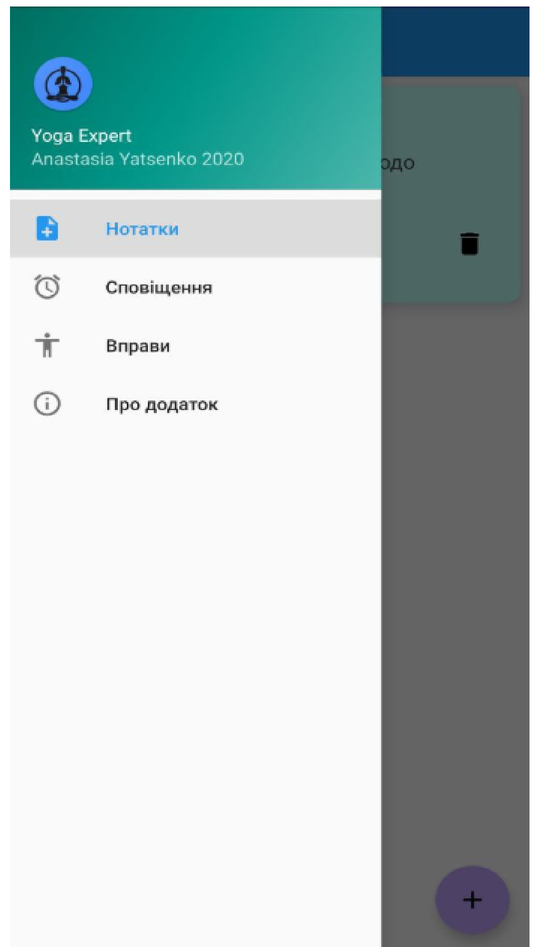


Рисунок 2.3.1 – Панель навігації

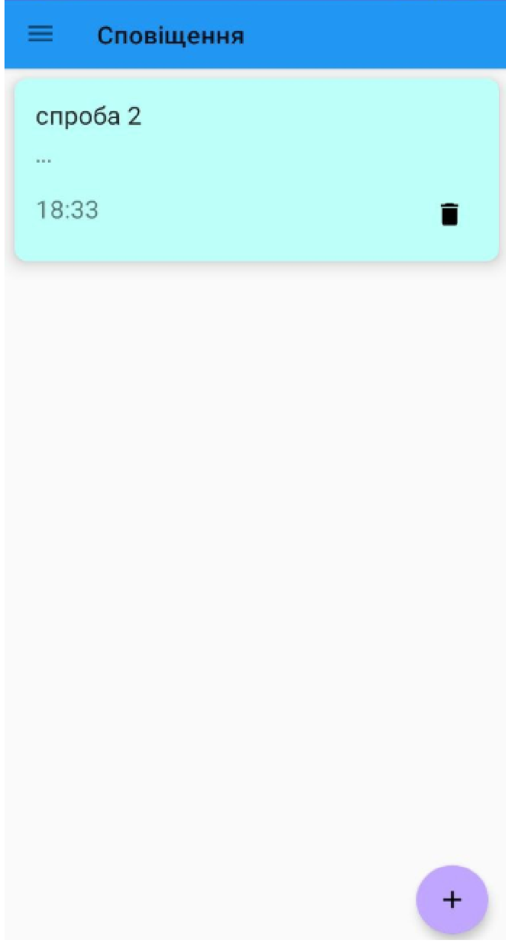
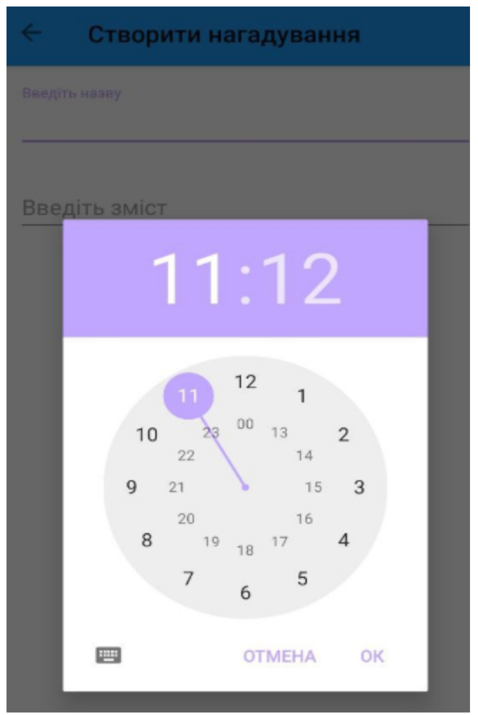


Рисунок 2.3.2 – Вкладка сповіщення



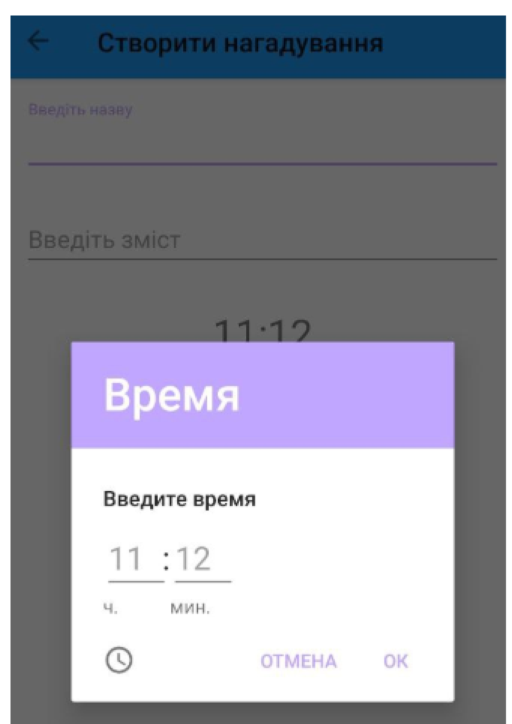


Рисунок 2.3.3 – Створення нагадування Рисунок 2.3.4 - Створення нагадування

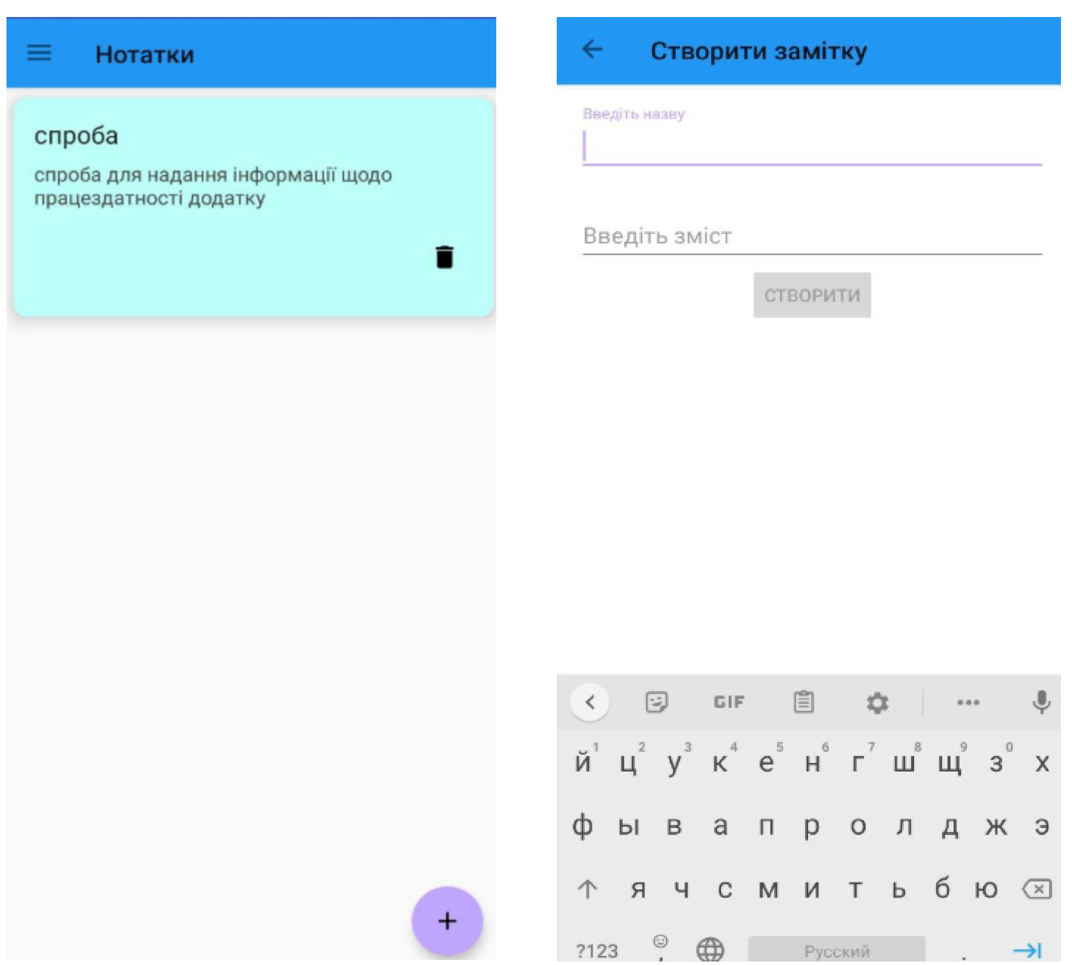


Рисунок 2.3.5 – Вкладка нотаток Рисунок 2.3.6 – Створення нотаток

Рисунок 2.3.7 – Вкладка вправ   Рисунок 2.3.8 – Вкладка про додаток

3 Програмна реалізація

3.1 Інформаційна модель

Android - операційна система для комунікаторів, планшетних комп'ютерів, цифрових програвачів, наручних годинників, нетбуків і смартфонів, заснована на ядрі Linux[10].

Програмні продукти під операційну систему Android розробляються в основному з використанням Java, проте даний додаток побудований на мові програмування Kotlin. Цей вибір був зроблений саме таким чином, оскільки мова програмування Kotlin на даний момент набирає обертів та стає більш популярною та прогресивною, оскільки має значні переваги перед Java. Саме тому багато програмістів пишуть на Kotlin, проте в будь якому випадку можна поєднати ці дві мови в одному проекті, оскільки вони є сумісними. Скомпільований програмний код (разом з усіма файлами ресурсів та іншої необхідної інформацією) упаковується в спеціальний файл-архів, Android

Package, цей файл має розширення * .apk. Саме він надалі поширюється як програма і інсталюється на мобільні пристрої. Також при створенні додатку в Android Studio мовою Kotlin, створюються різні класи, проте всі вони поділяються на дві групи: Kotlin класи та .xml розширення (відповідає за інтерфейс).

Мова Kotlin є однією з наймолодших та найактуальнішою в сімействі мов програмуваня і була розроблена з розрахунку на те, щоб професійний програміст міг легко її опанувати (після вивчення Java), оскільки саме вона поки що є основою програмування та більшість ПП побудовані з її допомогою, та ефективно використовувати за рахунок лаконічності. За основу взятий синтаксис JavaScript – безсумнівно, однієї з найбільш популярних мов програмування сучасності. Проте, Kotlin - це цілком самостійна мова програмування. Ідеологічно ж Kotlin побудована дещо інакше ніж Java. Розробники Kotlin грунтувалися на досвіді розробки програм на Java та Scala і прагнули позбутися мінусів використання цих мов прогамування, які зарекомендували себе непевними. Автори ставили перед собою ціль створити лаконічнішу та типо-безпечнішу мову, ніж Java, і простішу, ніж Scala. Наслідками спрощення, порівняно з Scala стали також швидша компіляція та краща підтримка IDE. Взагалі, інтерфейси Kotlin більш прості та прозорі для розуміння. Написати на Kotlin програму з графічним інтерфейсом не становить складностей після вивчення Java.

Для реалізації проекту було обрано середовище розробки Android Studio. Android Studio — інтегроване середовище розробки (IDE) для платформи Android, представлене 16 травня 2013 року на конференції Google I/O менеджером по продукції корпорації Google — Еллі Паверс (Ellie Powers). 8 грудня 2014 року компанія Google випустила перший стабільний реліз Android Studio 1.0. Android Studio прийшов на зміну плаґіну ADT для платформи Eclipse. Середовище побудоване на базі сирцевих текстів продукту IntelliJ IDEA Community Edition, що розвивається компанією JetBrains. Android Studio

розвивається в рамках відкритої моделі розробки та поширюється під ліцензією Apache 2.0.

Середовище розробки адаптоване для виконання типових завдань, що вирішуються в процесі розробки застосунків для платформи Android. У тому числі у середовище включені засоби для спрощення тестування програм на сумісність з різними версіями платформи та інструменти для проектування застосунків, що працюють на пристроях з екранами різної роздільності (планшети, смартфони, ноутбуки, годинники, окуляри тощо)[9].

3.2 Програмна реалізація сповіщень

(Kotlin class)

```
package com.fitnessexpert.notification
private const val CHANNEL_ID = "com.fitnessexpert.TRAINING_REMINDERS"
fun Context.showNotification(item: Reminder) = NotificationManagerCompat.from(this).notify(
    item.id.hashCode(), NotificationCompat.Builder(
        this,
        CHANNEL_ID
    )
        .setSmallIcon(R.drawable.ic_fitness_center_black_24dp)
        .setContentTitle(item.title)
        .setContentText(item.content)
        .setShowWhen(true)
        .setAutoCancel(true)
        .setContentIntent(mainActivityIntent)
        .setPriority(NotificationCompat.PRIORITY_DEFAULT).build()
)
fun Context.createNotificationChannel() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        val name = getString(R.string.channel_name)
```

```
        val descriptionText = getString(R.string.channel_description)
        val importance = NotificationManager.IMPORTANCE_DEFAULT
        val channel = NotificationChannel(CHANNEL_ID, name, importance).apply {
            description = descriptionText
        }
        // Register the channel with the system
        val notificationManager: NotificationManager =
            getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
        notificationManager.createNotificationChannel(channel)
    }
}
fun Context.cancelNotification(item: Reminder) =
    NotificationManagerCompat.from(this).cancel(item.id.hashCode())
private val Context.mainActivityIntent
    get() =
        PendingIntent.getActivity(this, 0, Intent(this, MainActivity::class.java).apply {
            flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_CLEAR_TASK
        }, 0)
```

(xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<layout xmlns:tools="http://schemas.android.com/tools"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:android="http://schemas.android.com/apk/res/android"><data><import
type="android.view.View"/><variable
type="com.fitnessexpert.ui.reminders.RemindersViewModel" name="model"/><variable
type="com.fitnessexpert.ui.reminders.RemindersFragment.Presenter"
name="presenter"/></data><androidx.constraintlayout.widget.ConstraintLayout
tools:context=".ui.reminders.RemindersFragment" android:layout_height="match_parent"
android:layout_width="match_parent"><androidx.recyclerview.widget.RecyclerView
android:layout_height="match_parent" android:layout_width="match_parent"
app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"
android:id="@+id/remindersList"/><com.google.android.material.floatingactionbutton.FloatingA
ctionButton android:layout_height="wrap_content" android:layout_width="wrap_content"
android:id="@+id/floatingActionButton" app:srcCompat="@drawable/ic_add_black_24dp"
app:layout_constraintEnd_toEndOf="parent" app:layout_constraintBottom_toBottomOf="parent"
android:transitionName="create_reminder_transition" android:onClick="@{() ->
presenter.onAdd()}" android:layout_marginBottom="@dimen/activity_vertical_margin"
android:layout_marginEnd="@dimen/activity_horizontal_margin"/><androidx.constraintlayout.
widget.ConstraintLayout android:layout_height="wrap_content"
android:layout_width="wrap_content" app:layout_constraintEnd_toEndOf="parent"
```

```
app:layout_constraintBottom_toBottomOf="parent" app:layout_constraintTop_toTopOf="parent"
app:layout_constraintStart_toStartOf="parent" android:visibility="@{model.isListEmpty ?
View.VISIBLE : View.GONE}"><ImageView android:layout_height="50dp"
android:layout_width="50dp" android:id="@+id/imageView2"
app:layout_constraintEnd_toEndOf="parent" app:layout_constraintTop_toTopOf="parent"
app:layout_constraintStart_toStartOf="parent" android:tint="@color/text_tertiary"
android:src="@drawable/ic_alarm_black_24dp"/><TextView
android:layout_height="wrap_content" android:layout_width="wrap_content"
android:id="@+id/textView4" app:layout_constraintEnd_toEndOf="parent"
app:layout_constraintStart_toStartOf="parent" app:layout_constraintWidth_max="200dp"
app:layout_constraintTop_toBottomOf="@+id/imageView2"
android:textColor="@color/text_tertiary" android:text="@string/empty_reminders"
android:gravity="center"
android:layout_marginTop="@dimen/nav_header_vertical_spacing"/><Button
android:layout_height="wrap_content" android:layout_width="wrap_content"
android:id="@+id/button2" app:layout_constraintEnd_toEndOf="parent" android:onClick="@{()
-> presenter.onAdd()}" app:layout_constraintStart_toStartOf="parent"
app:layout_constraintTop_toBottomOf="@+id/textView4" android:text="@string/create"
android:layout_margin="@dimen/activity_vertical_margin"/></androidx.constraintlayout.widget.
ConstraintLayout></androidx.constraintlayout.widget.ConstraintLayout></layout>
```

4 Тестування

4.1 Проведені види тестування

Після створення додатку, згідно з правилами, потрібно провести тестування, щоб бути впевнений у його працездатності. При перевірці, зазвичай, знаходять дефекти. з котрими система не працює взагалі або ж некоректно відповідає функціонал. Це виконується для того, щоб забезпечити працездатність додатку та виправити недоліки до релізу.

Тестування за своєю специфікою поділяється на два види: ручне та автоматизоване. Під час ручного тестування всі операції виконуються мануально, тобто самі перевіряємо і оцінюємо, як працює та чи інша функція. Автоматизоване тестування дає змогу перевірити код реалізації функцій, за допомогою якого маємо можливість протестувати фунціонал, і перевірити, чи повертається потрібний результат.

Отже, були проведені наступні види тестування додатку:

- Інсталяційне тестування

Додаток був успішно встановлений на наступні версії Android: Android 10, Android 8.1 Oreo, Android 6.0 Marshmallow.

- Продуктивність

Додаток стабільно функціонував при низькому заряді акумулятора, разом з іншими додатками, при поганому покритті мережі. Проведено тестування при різних типах підключення до мережі інтернет(2g, 3g, WiFi), яке успішно завершилося.

- Функціональне тестування

Весь функціонал був перевірений на працездатність.

Тестування пройшло успішно, додаток готовий для використання та експлуатації, фатальних багів виявлено не було.

4.2 Перевірка на працездатність додатку

У цьому розділі представлена працездатність додатку. Нижче представлені скріншоти роботи:

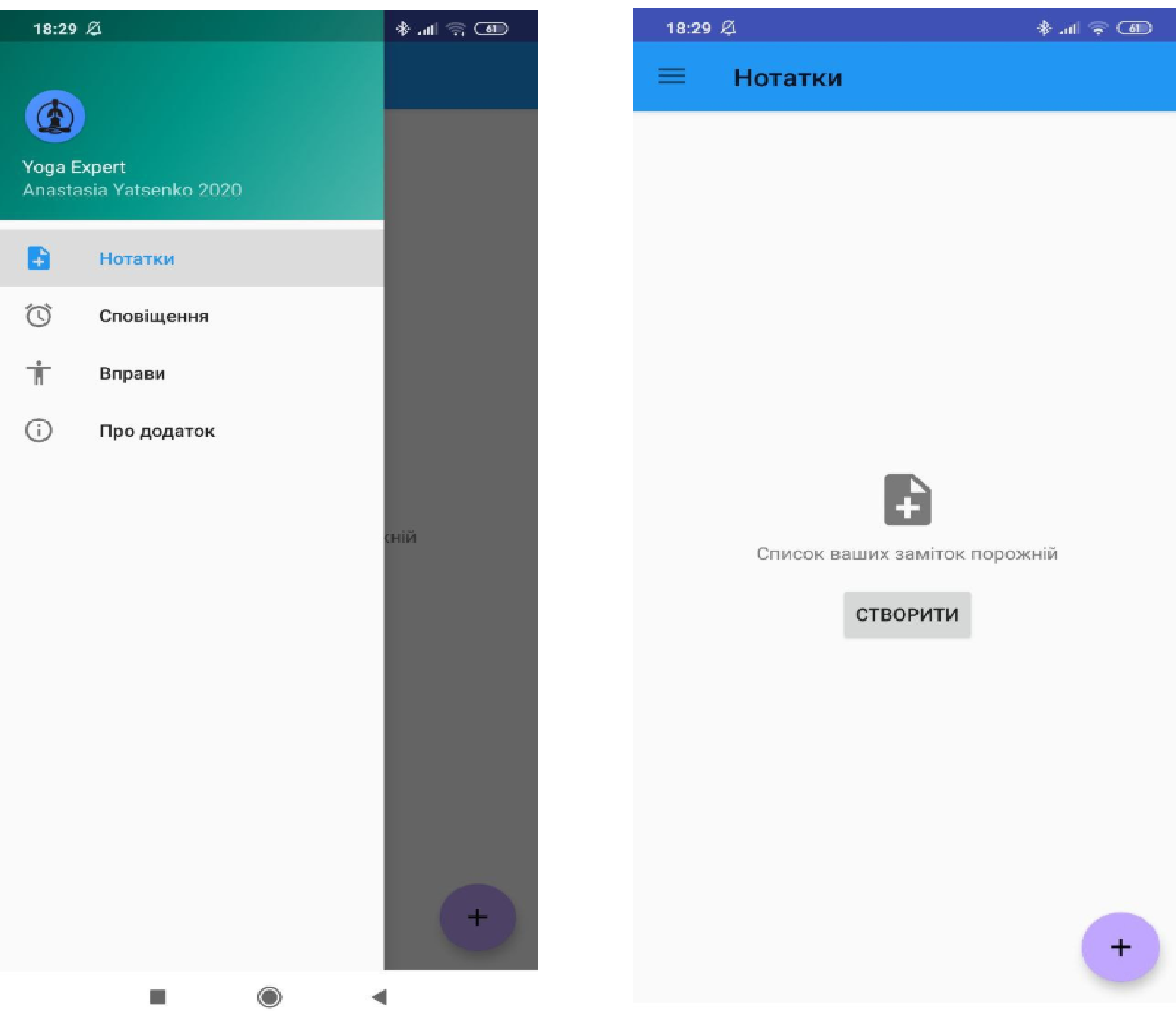
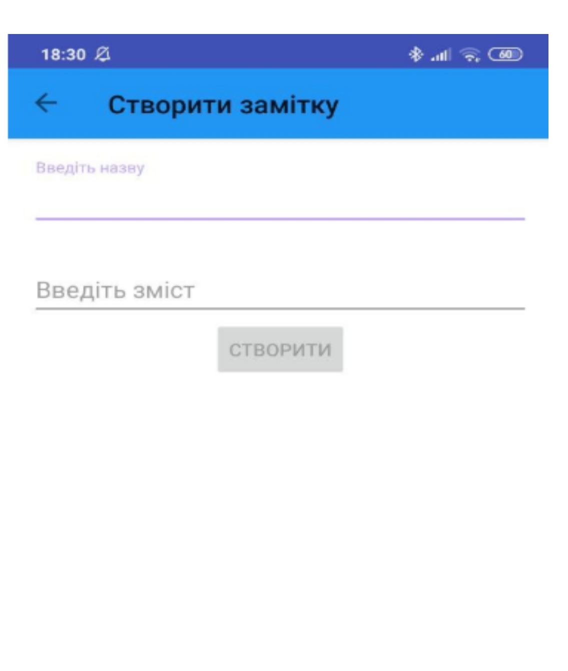


Рисунок 4.2.1 – Головна панель навігації

Рисунок 4.2.2 – Створення нотаток



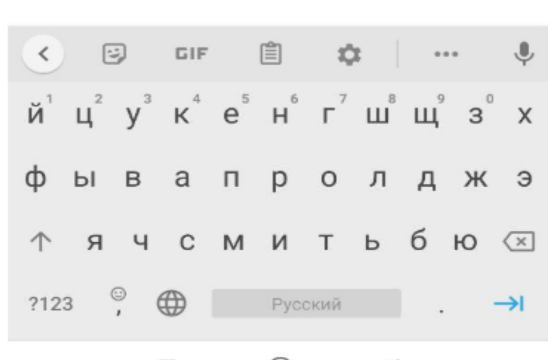


Рисунок 4.2.3 – Створення замітки

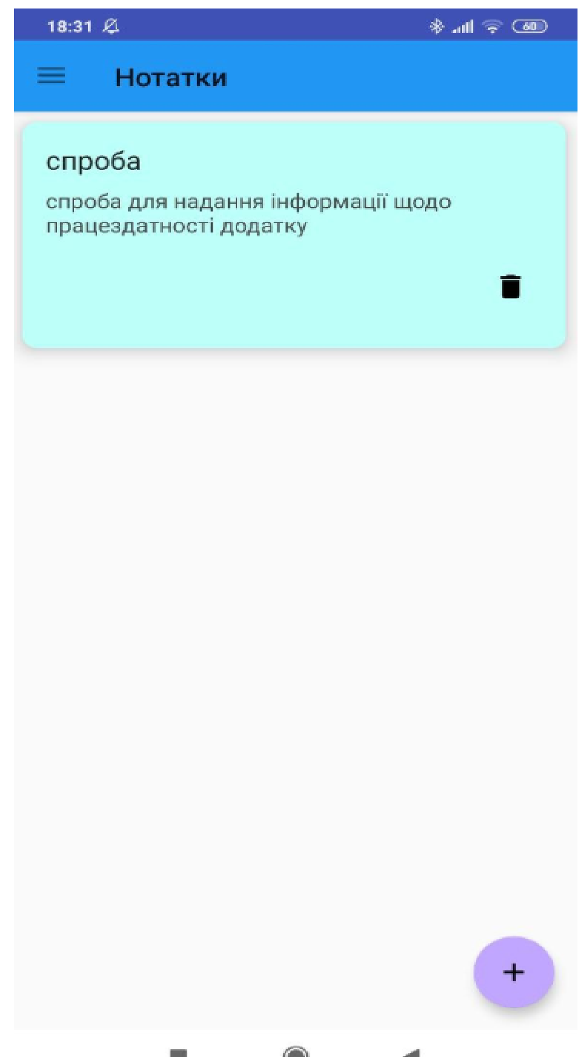


Рисунок 4.2.4 – Створена нотатка

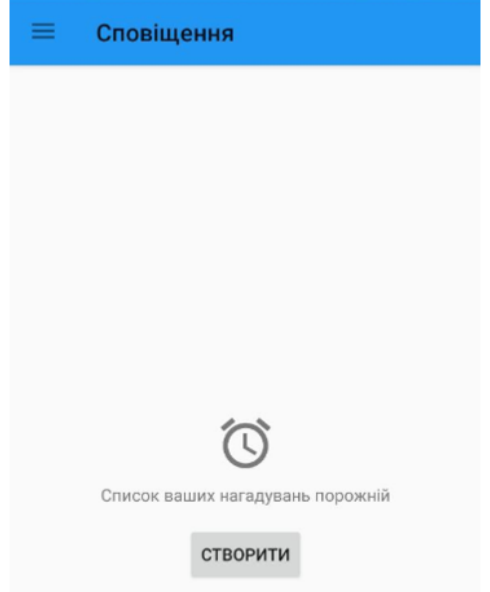


Рисунок 4.2.5 – Створення нагадування

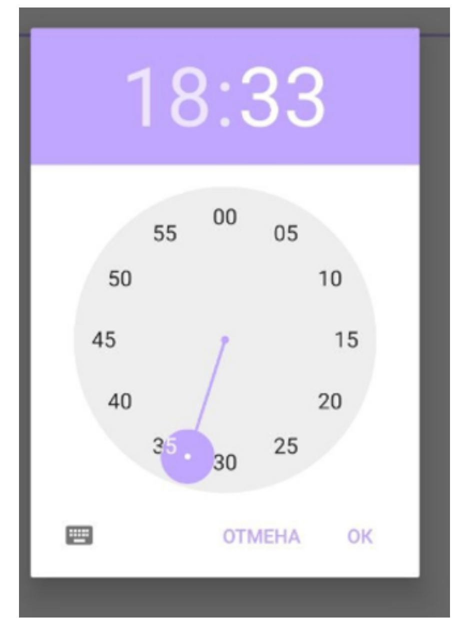


Рисунок 4.2.6 – Встановлення часу

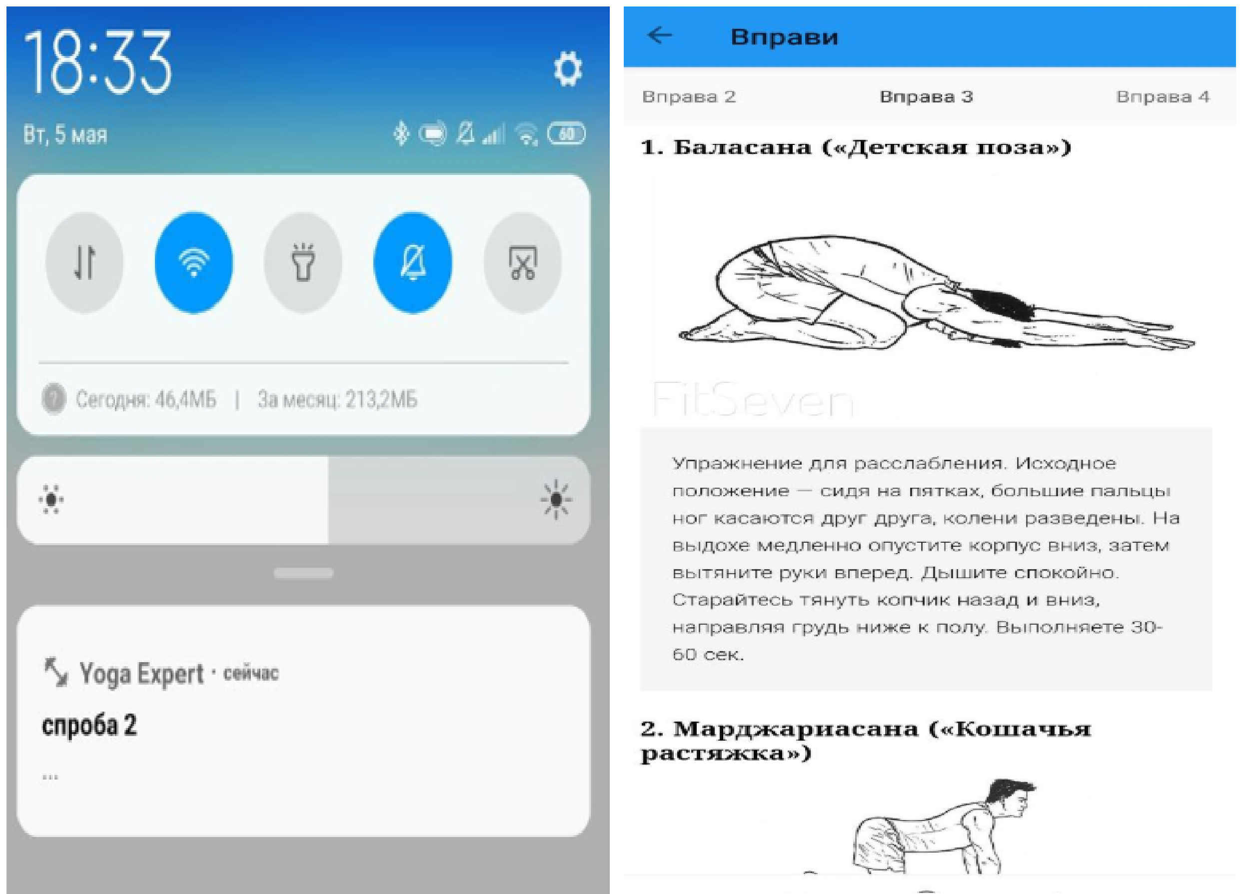


Рисунок 4.2.7 – Скрін сповіщення Рисунок 4.2.8 – Скрін поля «Вправи»

Виходячи з результатів на скріншотах додатку, можна сказати, що функціонал додатку працює. Для роботи з додатком потрібно спочатку встановити його на смартфон (файл *.apk), оновлення версії Android мінімум до 6 та 16 MB вільної пам’яті. Після встановлення додаток не потребує авторизації, тому можна починати роботу.

Висновок

Виконана робота присвячена створенню додатку на операційній системі Android. Під час створення було пройдено всі етапи, а саме:

1. На етапі аналізу предметної області було проведено огляд та аналіз кількох додатків, що реалізують схожі функції предметної області.

2. На етапі проектування програмної системи розроблено архітектуру та використані сучасні методи та компоненти архітектури бібліотеки JetPack.

3. На етапі програмної реалізації з використанням об’єктноорієнтованого підходу в інтегрованому середовищі розробки програмного забезпечення Android Studio, реалізовано функції системи, котрі написані на мові програмування Kotlin. Розроблено інтерфейс програми.

4. На етапі тестування була проведена перевірка працездатності додатку та його функціоналу.

Результатом дипломної роботи є створений додаток, котрий ознайомить користувачів з практиками йоги, що в свою чергу допоможе урізноманітнити життя та вирішити деякі проблеми із самопочуттям.

Список літератури

1. Огляд liveData [Електронний ресурс] – Режим доступу: https://developer.android.com/topic/libraries/architecture/livedata Назва з екрану.

2. Data repository pattern в Android[Електронний ресурс] – Режим доступу: https://proandroiddev.com/the-real-repository-pattern-in-android-efba8662b754 Назва з екрану.

3. Події на базі LiveData Android [Електронний ресурс] – Режим доступу: https://habr.com/ru/post/468749/ .

4. Model-View-ViewModel [Електронний ресурс] – Режим доступу: https://ru.wikipedia.org/wiki/Model-View-ViewModel .

5. Kotlin vs Java [Електронний ресурс] – Режим доступу: https://itvdn.com/ru/blog/article/kotlin-vs-java

6. Внутренняя инженерия. Путь к радости. Практическое руководство от йога – 2019 р. ст.58-62

7. Room основи [Електронний ресурс] – Режим доступу: https://startandroid.ru/ru/courses/architecture-components/27-course/architecture-components/529-urok-5-room-osnovy.html

8. ViewModel [Електронний ресурс] – Режим доступу: https://startandroid.ru/ru/courses/architecture-components/27-course/architecture-components/527-urok-4-viewmodel.html

9. Android Studio [Електронний ресурс] – Режим доступу: https://ru.wikipedia.org/wiki/Android_Studio

10. Компоненти Jetpack [Електронний ресурс] – Режим доступу: https://habr.com/ru/post/451112/

Додаток А. Лістинг коду

Kotlin класи

```
import com.fitnessexpert.R

import com.fitnessexpert.databinding.ItemNoteBinding

import com.fitnessexpert.model.entities.Note

import com.fitnessexpert.ui.notes.NotesFragment


class NotesListAdapter(

    val lifecycleOwner: LifecycleOwner,

    val presenter: NotesFragment.Presenter

) : ListAdapter<Note, NotesListAdapter.ChatHolder>(

    object : DiffUtil.ItemCallback<Note>() {

        override fun areItemsTheSame(oldItem: Note, newItem: Note) =

            oldItem.id == newItem.id


        override fun areContentsTheSame(

            oldItem: Note, newItem: Note

        ) = oldItem == newItem

    }

) {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int) = ChatHolder(

        DataBindingUtil.inflate(

            LayoutInflater.from(parent.context),

            R.layout.item_note, parent, false

        )

    )

 override fun onBindViewHolder(holder: ChatHolder, position: Int) =

        holder.bind(getItem(position))


    inner class ChatHolder(

        private val binding: ItemNoteBinding
```

```
    ) : RecyclerView.ViewHolder(binding.root) {

        fun bind(item: Note) {
            binding.item = item
            binding.presenter = presenter
            binding.lifecycleOwner = lifecycleOwner
            binding.executePendingBindings()
        }
    }
package com.fitnessexpert.adapter
import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.databinding.DataBindingUtil
import androidx.lifecycle.LifecycleOwner
import androidx.recyclerview.widget.DiffUtil
import androidx.recyclerview.widget.ListAdapter
import androidx.recyclerview.widget.RecyclerView
import com.fitnessexpert.R
import com.fitnessexpert.databinding.ItemReminderBinding
import com.fitnessexpert.model.entities.Reminder
import com.fitnessexpert.ui.reminders.RemindersFragment

class RemindersListAdapter(
    val lifecycleOwner: LifecycleOwner,
    val presenter: RemindersFragment.Presenter
) : ListAdapter<Reminder, RemindersListAdapter.ChatHolder>(
    object : DiffUtil.ItemCallback<Reminder>() {
        override fun areItemsTheSame(oldItem: Reminder, newItem: Reminder) =
            oldItem.id == newItem.id

        override fun areContentsTheSame(
```

```
            oldItem: Reminder, newItem: Reminder
        ) = oldItem == newItem
    }
) {
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int) = ChatHolder(
        DataBindingUtil.inflate(
            LayoutInflater.from(parent.context),
            R.layout.item_reminder, parent, false
        )
    )
override fun onBindViewHolder(holder: ChatHolder, position: Int) =
        holder.bind(getItem(position))

    inner class ChatHolder(
        private val binding: ItemReminderBinding
    ) : RecyclerView.ViewHolder(binding.root) {

        fun bind(item: Reminder) {
            binding.item = item
            binding.presenter = presenter
            binding.lifecycleOwner = lifecycleOwner
            binding.executePendingBindings()
        }
    }
package com.fitnessexpert.extension

import android.content.Context
import androidx.activity.ComponentActivity
import androidx.activity.viewModels
import androidx.annotation.MainThread
import androidx.appcompat.app.AlertDialog
```

```
import androidx.fragment.app.Fragment
import androidx.fragment.app.activityViewModels
import androidx.fragment.app.viewModels
import androidx.lifecycle.ViewModel
import androidx.lifecycle.ViewModelProvider
import com.fitnessexpert.R

inline fun <reified VM : ViewModel> factory(noinline creator: () -> VM): ViewModelProvider.Factory {
   return object : ViewModelProvider.Factory {
      override fun <T : ViewModel?> create(modelClass: Class<T>): T {
         if (modelClass.isAssignableFrom(VM::class.java)) {
            @Suppress("UNCHECKED_CAST")
            return creator() as T
         }
         throw IllegalArgumentException("Unknown ViewModel class $modelClass")
      }
   }
}

@MainThread
inline fun <reified VM : ViewModel> ComponentActivity.viewModels(noinline creator: () -> VM): Lazy<VM> =
   viewModels { factory(creator) }

@MainThread
inline fun <reified VM : ViewModel> Fragment.viewModels(noinline creator: () -> VM): Lazy<VM> =
   viewModels { factory(creator) }

@MainThread
inline fun <reified VM : ViewModel> Fragment.activityViewModels(noinline creator: () -> VM): Lazy<VM> =
   activityViewModels { factory(creator) }
```

```
fun Context.showConfirmationDialog(message: String, onConfirm: () -> Unit) =
   AlertDialog.Builder(this)
       .setTitle(R.string.are_you_sure)
       .setMessage(message)
       .setNegativeButton(android.R.string.no) { _, _ -> }
       .setPositiveButton(android.R.string.yes) { _, _ -> onConfirm() }
       .show()
```

—

```
package com.fitnessexpert.model

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import com.fitnessexpert.model.dao.NotesDao
import com.fitnessexpert.model.dao.RemindersDao
import com.fitnessexpert.model.entities.Note
import com.fitnessexpert.model.entities.Reminder

@Database(entities = [Note::class, Reminder::class], version = 1, exportSchema = false)
abstract class FitnessExpertDatabase : RoomDatabase() {
abstract fun notesDao(): NotesDao
   abstract fun remindersDao(): RemindersDa
 companion object {

       @Volatile
       private var INSTANCE: FitnessExpertDatabase? = null

       fun getDatabase(context: Context): FitnessExpertDatabase {
           val temp = INSTANCE
           if (temp != null) return temp
```

```
            synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    FitnessExpertDatabase::class.java,
                    "fitness_expert_Database"
                ).build()
                INSTANCE = instance
                return instance
            }
        }
    ----
package com.fitnessexpert.model

import com.fitnessexpert.model.dao.NotesDao
import com.fitnessexpert.model.dao.RemindersDao
import com.fitnessexpert.model.entities.Note
import com.fitnessexpert.model.entities.Reminder

class NotesRepository(private val notesDao: NotesDao, private val remindersDao: RemindersDao) {
    suspend fun insertOrReplace(note: Note) = notesDao.addNote(note)
    suspend fun insertOrReplace(reminder: Reminder) = remindersDao.addReminder(reminder)
    suspend fun deleteNote(note: Note) = notesDao.deleteNote(note)
    suspend fun deleteReminder(reminder: Reminder) = remindersDao.deleteReminder(reminder)
    fun getNotes() = notesDao.getNotes()
    fun getReminders() = remindersDao.getReminders()
    suspend fun getRemindersAsync() = remindersDao.getRemindersAsync()
    fun findNote(id: String) = notesDao.findNote(id)
    fun findReminder(id: String) = remindersDao.findReminder(id)
    suspend fun findReminderAsync(id: String) = remindersDao.findReminderAsync(id)
}
```

```
private const val CHANNEL_ID = "com.fitnessexpert.TRAINING_REMINDERS"

fun Context.showNotification(item: Reminder) = NotificationManagerCompat.from(this).notify(
    item.id.hashCode(), NotificationCompat.Builder(
        this,
        CHANNEL_ID
    )
        .setSmallIcon(R.drawable.ic_fitness_center_black_24dp)
        .setContentTitle(item.title)
        .setContentText(item.content)
        .setShowWhen(true)
        .setAutoCancel(true)
        .setContentIntent(mainActivityIntent)
        .setPriority(NotificationCompat.PRIORITY_DEFAULT).build()
)

fun Context.createNotificationChannel() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        val name = getString(R.string.channel_name)
        val descriptionText = getString(R.string.channel_description)
        val importance = NotificationManager.IMPORTANCE_DEFAULT
        val channel = NotificationChannel(CHANNEL_ID, name, importance).apply {
            description = descriptionText
        }
        // Register the channel with the system
        val notificationManager: NotificationManager =
            getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
        notificationManager.createNotificationChannel(channel)
    }
}
```

```
fun Context.cancelNotification(item: Reminder) =
    NotificationManagerCompat.from(this).cancel(item.id.hashCode())


private val Context.mainActivityIntent
    get() =
        PendingIntent.getActivity(this, 0, Intent(this, MainActivity::class.java).apply {
            flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_CLEAR_TASK
        }, 0)
```

```
package com.fitnessexpert.ui.about


import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment
import com.fitnessexpert.R


class AboutFragment : Fragment() {
override fun onCreateView(
            inflater: LayoutInflater,
            container: ViewGroup?,
            savedInstanceState: Bundle?
    ): View = inflater.inflate(R.layout.fragment_about,container,false)
}
```

```
class NotesFragment : Fragment() {


    private val notesViewModel by viewModels {
        NotesViewModel(FitnessExpertDatabase.getDatabase(requireContext()))
    }


    override fun onCreateView(
```

```
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ) = DataBindingUtil.inflate<FragmentNotesBinding>(
        inflater,
        R.layout.fragment_notes,
        container,
        false
    ).apply {
        lifecycleOwner = this@NotesFragment
        model = notesViewModel
        val viewPresenter = Presenter()
        presenter = viewPresenter

        notesList.adapter = NotesListAdapter(
            this@NotesFragment,
            viewPresenter
        )

    }.root

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        notesViewModel.notes.observe(viewLifecycleOwner, Observer {
            (notesList.adapter as? NotesListAdapter)?.submitList(it)
        })
    }
```

```
package com.fitnessexpert.ui.notes


import androidx.lifecycle.*
import com.fitnessexpert.model.FitnessExpertDatabase
```

```
import com.fitnessexpert.model.entities.Note
import com.fitnessexpert.model.NotesRepository
import kotlinx.coroutines.launch

class NotesViewModel(database: FitnessExpertDatabase) : ViewModel() {
    fun deleteNote(item: Note) = viewModelScope.launch {
        repository.deleteNote(item)
    }

    private val repository =
        NotesRepository(database.notesDao(), database.remindersDao())

    val notes = repository.getNotes()
    val isListEmpty = notes.map { it.isEmpty() }
}
```

```
class RemindersFragment : Fragment() {

    private val remindersViewModel by viewModels {
        RemindersViewModel(FitnessExpertDatabase.getDatabase(requireContext()))
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ) = DataBindingUtil.inflate<FragmentRemindersBinding>(
        inflater,
        R.layout.fragment_reminders,
        container,
        false
    ).apply {
```

```
        lifecycleOwner = this@RemindersFragment
        model = remindersViewModel
        val viewPresenter = Presenter()
        presenter = viewPresenter

        remindersList.adapter = RemindersListAdapter(
            this@RemindersFragment,
            viewPresenter
        )
    }.root

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        remindersViewModel.reminders.observe(viewLifecycleOwner, Observer {
            (remindersList.adapter as? RemindersListAdapter)?.submitList(it)
        })
    }
inner class Presenter {
        fun onAdd() = findNavController().navigate(
            RemindersFragmentDirections.createReminderAction(title = getString(R.string.create_reminder))
        )
fun onEdit(item: Reminder) = findNavController().navigate(
            RemindersFragmentDirections.editReminderAction(getString(R.string.edit_reminder), item)
        )
 fun onDelete(item: Reminder) {
            requireContext().showConfirmationDialog(getString(R.string.want_to_delete_reminder)) {
                deleteReminder(item)
            }
        }
  private fun deleteReminder(item: Reminder) {
            requireContext().apply {
```

```
            cancelNotification(item)

            cancelScheduledNotification(item)

        }

        remindersViewModel.deleteReminder(item)

    ----------------------

package com.fitnessexpert.ui.reminders


import androidx.lifecycle.*

import com.fitnessexpert.model.FitnessExpertDatabase

import com.fitnessexpert.model.NotesRepository

import com.fitnessexpert.model.entities.Note

import com.fitnessexpert.model.entities.Reminder

import kotlinx.coroutines.launch

class RemindersViewModel(database: FitnessExpertDatabase) : ViewModel() {

    fun deleteReminder(item: Reminder) = viewModelScope.launch {

        repository.deleteReminder(item)

    }

 private val repository =

        NotesRepository(database.notesDao(), database.remindersDao())

val reminders = repository.getReminders()

    val isListEmpty = reminders.map { it.isEmpty() }

}
```