МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ
ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

перший (бакалаврський)
(рівень вищої освіти)

на тему **Дослідження технологій персистентності даних Android-застосунків**

Виконав: студент 4 курсу, групи 6.1210 – пзс
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Програмне забезпечення систем
(код і назва освітньої програми)
Є.Т. Смирнов
(ініціали та прізвище)
Керівник к.ф.–м.н., доцент, доцент кафедри ЕІС та ПЗ
Г.П. Коломоєць
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «Дискус» Р.О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра електроніки, інформаційних систем та програмного забезпечення
Рівень вищої освіти _____перший (бакалавський)_____
Спеціальність ___121_Інженерія програмного забезпечення_____
(код та назва)
Освітня програма ___Програмне забезпечення систем_____
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри_____Тетяна КРИТСЬКА_
"_01_" __березня_____2024_року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Смирнова Єгора Теімуразовича
(прізвище, ім'я, по батькові)

1.Тема роботи Дослідження технологій персистентності даних Android-застосунків
керівник роботи Коломоєць Геннадій Павлович, к.ф.-м.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ЗНУ від 26.12.2023 № 2215-с

2. Строк подання студентом кваліфікаційної роботи 07.06.2024
3. Вихідні дані кваліфікаційної роботи бакалавра
- комплект нормативних документів ;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
- огляд та збір літератури стосовно теми кваліфікаційної роботи;
- огляд та аналіз існуючих рішень та аналогів;
- аналіз організації баз даних у операційній системі Android.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
_14_____слайдів презентації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата Завдання прийняв

6. Консультанти розділів бакалаврської роботи

7. Дата видачі завдання 01.03.2024

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів кваліфікаційної роботи	Примітка
1	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником	14.02 – 15.02.24	виконано
2	Аналіз предметної області	15.02 – 20.02.24	виконано
3	Аналіз існуючих методів рішення	21.02 – 25.02.24	виконано
4	Огляд та збір літератури стосовно теми кваліфікаційної роботи	26.02 – 27.02.24	виконано
5	Огляд та аналіз існуючих рішень та аналогів	28.02 – 29.02.24	виконано
6	Програмна реалізація застосунку	12.03 – 12.04.24	виконано
7	Участь у конференції «Молода наука - 2024»	17.04 – 18.04.24	виконано
8	Програмна реалізація запитів Android-застосунку до бази даних	24.05 – 25.05.24	виконано
9	Оформлення звіту	01.06 – 13.06.24	виконано

Студент _____ Є.Т.Смирнов
(підпис) (прізвище та ініціали)

Керівник роботи _____ Г.П. Коломоєць
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено
Нормоконтролер _____ І.А. Скрипник
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сторінок – 75

Рисунків – 9

Лістингів – 29

Джерел – 15

Смирнов Є.Т. Дослідження технологій персистентності даних Android-застосунків : кваліфікаційна робота бакалавра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник Г. П. Коломоєць. Запоріжжя : ЗНУ, 2024. 75 с.

У роботі проводиться дослідження технологій персистентності даних з використанням SQLite у мобільних застосунках для платформи Android. Досліджуються основні програмні засоби Android для роботи з SQLite, зокрема для організації CRUD-запитів, підтримки транзакцій, забезпечення цілісності даних, а також інтеграції з іншими компонентами Android-застосунку.

Для вивчення засобів організації персистентності даних у мобільних застосунках для платформи Android був розроблений мобільний застосунок з використанням СКБД SQLite для обліку персональних витрат та доходів.

Проведена апробація працездатності розробленого мобільного застосунку.

Ключові слова: *SQLite, мобільний застосунок, персистентність даних, транзакції, Android.*

ABSTRACT

Pages – 75

Figures – 9

Listings – 29

Sources – 15

Smirnov Y.T. Research on Data Persistence Technologies in Android Applications: Bachelor's qualification paper in the specialty 121 "Software Engineering" / Scientific supervisor G.P. Kolomoyets. Zaporizhzhia : ZNU, 2024. 75 p.

This paper explores data persistence technologies using SQLite in mobile applications for the Android platform. The study investigates the main Android software tools for working with SQLite, including organizing CRUD operations, supporting transactions, ensuring data integrity, and integrating with other components of an Android application.

To study the tools for organizing data persistence in mobile applications for the Android platform, a mobile application using the SQLite DBMS was developed to track personal expenses and income.

The functionality of the developed mobile application has been tested and validated.

Keywords: *SQLite, mobile application, data persistence, transactions, Android*.

ЗМІСТ

ВСТУП

Атуальність теми

У сучасному світі мобільні застосунки стають все більш популярними та необхідними для забезпечення різноманітних потреб користувачів. Завдяки їх зручності та доступності, користувачі використовують мобільні застосунки для комунікації, розваг, роботи, освіти та багато інших цілей. З цим зростанням популярності збільшується і обсяг даних, які зберігаються та обробляються в мобільних застосунках.

Для забезпечення зручності користувачів та функціональності застосунків необхідно забезпечити ефективне та надійне збереження даних на мобільних пристроях. Користувачі очікують, що їх дані будуть доступні та захищені в будь-який час, навіть при відсутності зв'язку з мережею або після вимкнення пристрою. Бази даних є основним інструментом для збереження та управління даними в мобільних застосунках. Вони дозволяють зберігати структуровану інформацію та здійснювати різноманітні операції з даними, такі як додавання, оновлення, видалення та пошук.

Персистентність даних в мобільних застосунках є ключовою для забезпечення надійності та доступності даних. Користувачі очікують, що їхні дані будуть збережені навіть після закриття застосунка або випадкового вимкнення пристрою. З цією метою, необхідно вивчати та застосовувати найбільш ефективні технології персистентності даних у мобільних застосунках [1]. Платформа Android постійно оновлюється та вдосконалюється, що призводить до появи нових можливостей та інструментів для розробки застосунків. Дослідження технологій персистентності даних у контексті останніх версій Android дозволить використовувати найсучасніші можливості для збереження та управління даними в мобільних застосунках.

З урахуванням зростання популярності мобільних застосунків, необхідності забезпечення надійного збереження даних, використання баз

даних у мобільних застосунках важливості персистентності даних для користувачів та швидкого розвитку платформи Android, дослідження технологій персистентності даних у мобільних застосунках на платформі Android є актуальною та важливою темою. Цей розділ визначає актуальність теми дослідження та вказує на її значення для подальших досліджень та практичного застосування.

Мета дослідження

Вивчення засобів організації персистентності даних у мобільних застосунках для платформи Android.

Завдання дослідження

Визначення функціональних вимог, проєктування та конструювання Android-застосунку, що зберігає дані у вбудованій до платформи Android СКБД SQLite.

Об’єкт дослідження

Об'єктом дослідження є технології персистентності даних у мобільних застосунках для платформи Android.

Предмет дослідження

Предметом дослідження є низькорівневі засоби Android для організації збереження даних у вбудованій СКБД SQLite та роботи з ними.

Методи дослідження

Для дослідження технологій персистентності даних у мобільних Android-застосунках були використані такі методи: аналіз предметної області за інформаційними джерелами, моделювання та аналіз даних, проєктування та

конструювання програмного застосунку із застосуванням технологій персистентності.

Практичне значення одержаних результатів

Практичне значення одержаних результатів полягає у напрацюванні кодової бази із використанням засобів SQLiteDatabase та SQLiteOpenHelper при розробці Android-застосунку для обліку персональних витрат та оцінки зручності та ефективності їх використання.

Апробація одержаних результатів

Результати дослідження були представлені на XVII науково-практичній конференції студентів, аспірантів, докторантів і молодих вчених Запорізького національного університету «Молода наука-2024» [15].

Глосарій

SQLite - це вбудована реляційна база даних з відкритим вихідним кодом, яка не потребує окремого серверного процесу, а надає бібліотеку, з якою програма компілюється і рушій стає складовою частиною програми [2], і зберігає дані в звичайних файлах.

Мобільний застосунок - це програмне забезпечення, розроблене для роботи на мобільних пристроях, таких як смартфони та планшети.

Персистентність даних - це властивість зберігати дані таким чином, щоб вони залишались доступними і після завершення роботи програми або перезавантаження пристрою.

Транзакції - це набір операцій з базою даних, які виконуються як єдине ціле. Якщо одна з операцій не вдається, всі зміни, зроблені в рамках транзакції, відкатуються.

Android - це операційна система, розроблена компанією Google, яка використовується на мобільних пристроях, таких як смартфони та планшети.

1 ТЕХНОЛОГІЇ ПЕРСИСТЕНТНОСТІ ПЛАТФОРМИ ANDROID

1.1 Засоби Android для персистентного зберігання даних

Для розвитку сучасних мобільних застосунків важливим є забезпечення персистентності даних, тобто здатність зберігати і відновлювати дані після завершення роботи з програмою або перезавантаження пристрою. У мобільному застосунку для операційної системи Android ефективне управління персистентними даними відіграє критичну роль у забезпеченні функціональності, продуктивності та зручності для користувачів.

При створенні мобільних застосунків важливо забезпечити збереження даних, введених користувачем, налаштувань, стану застосунка та іншої інформації, яка має залишатися доступною після закриття програми або вимкнення пристрою. Це дозволяє користувачам продовжувати використовувати застосунок без втрати даних та забезпечує зручний та безперебійний досвід.

Для забезпечення персистентності даних у мобільних застосунках використовуються різноманітні механізми, такі як локальні файли, бази даних, хмарні сховища тощо. Згідно з рекомендаціями з офіційного сайту розробників Android [3], існують чотири основні типи персистентних даних:

- Дані виключно застосунку: локальні файли, які використовуються тільки в межах одного застосунку і не доступні іншим.
- Дані, які застосунок поділяє з іншими застосунками: використовуються для обміну даними між різними застосунками.
- Вподобання користувачів: SharedPreferences або DataStore<Preferences>, які зберігають налаштування користувача та інші невеликі обсяги даних.
- Бази даних: використання вбудованої СКБД, SQLite або хмарних СКБД, таких як Firebase [4].

Кожен з цих механізмів має свої переваги та обмеження, і вибір конкретного залежить від функціоналу застосунка та вимог проекту. Використання інтерфейсів SharedPreferences або DataStore<Preferences> (наразі рекомендується використання останнього [5]) для збереження налаштувань застосунка є поширеним підходом для забезпечення персистентності простих даних, таких як налаштування застосунка, режими відображення тощо. Для збереження та керування структурованими даними часто використовуються вбудовані СКБД, такі як SQLite [6]. Ця СКБД надає потужні засоби зберігання даних у вигляді реляційних таблиць та здійснення операцій, таких як додавання, читання, оновлення та видалення записів. Для нашого дослідження ми зосередимося на останньому типі – базах даних, зокрема на використанні SQLite.

1.2 Засоби Android для роботи з базами даних

Android надає вбудовану СКБД SQLite для застосунків разом з низькорівневим API, до якого можна віднести [7]:

- Клас SQLiteDatabase: Це основний клас для роботи з базами даних у Android. Він забезпечує методи для виконання SQL-запитів, управління транзакціями та роботу з базою даних. За його допомогою можна створювати, читати, оновлювати та видаляти записи в базі даних – SQLiteDatabase надає широкий набір функцій для виконання CRUD-операцій (створення, читання, оновлення, видалення).
- Інтерфейс Cursor: Використовується для зчитування результатів SQL-запитів. Він забезпечує ітерацію по рядках результатів і доступ до стовпців. Cursor дозволяє переміщатися по результатах запиту, отримувати дані з конкретних стовпців кожного рядка і закривати набір результатів після завершення роботи. Це дозволяє

ефективно працювати з великими наборами даних, не завантажуючи всю базу даних у пам'ять.

- Клас SQLiteOpenHelper: Допоміжний клас, що полегшує управління створенням та оновленням бази даних. Він спрощує обробку створення та оновлення схем бази даних, автоматично викликаючи методи onCreate() та onUpgrade() при створенні або оновленні бази даних відповідно. SQLiteOpenHelper також забезпечує методи для відкриття бази даних у режимах читання і запису, що робить його зручним інструментом для керування життєвим циклом бази даних.

Room є бібліотекою об'єктно-реляційного відображення (ORM) для Android, яка спрощує роботу з SQLite [8]. Вона надає зручний API для взаємодії з базою даних, автоматично генерує SQL-запити і забезпечує типобезпеку. Room інтегрується з іншими компонентами Android Architecture Components, такими як LiveData і ViewModel, що забезпечує реактивне програмування і спрощує управління станом застосунка. Room дозволяє розробникам працювати з базою даних через об'єкти Java або Kotlin, зменшуючи кількість коду, який потрібно писати для виконання операцій з базою даних.

Room надає API для створення та управління базою даних за допомогою анотацій та об'єктів Java/Kotlin.

Переваги та недоліки SQLite та Room:

Переваги SQLite:

- Потужні можливості для зберігання структурованих даних: SQLite забезпечує надійне зберігання і управління даними, підтримуючи складні SQL-запити та транзакції.
- Безпосередній контроль: Низькорівневий доступ до бази даних дозволяє розробникам мати повний контроль над SQL-запитами та оптимізацією.

Недоліки SQLite:

- Складність налаштування: Робота з низькорівневими API вимагає написання значної кількості коду для створення та управління базами даних, а також для обробки міграцій.
- Високий рівень ручного кодування: Необхідність вручну писати SQL-запити і обробляти результати, що може призвести до помилок та зниження продуктивності розробки.

Переваги Room:

- Типобезпека і простота використання: Room забезпечує типобезпеку і спрощує роботу з базою даних, автоматично генеруючи SQL-запити і забезпечуючи інтеграцію з іншими компонентами Android.
- Автоматичне управління міграціями: Room забезпечує автоматичне управління міграціями бази даних, що спрощує оновлення структури бази даних.
- Легка інтеграція з Android Architecture Components: Room інтегрується з компонентами, такими як LiveData і ViewModel, що полегшує управління даними і станом додатка.

Недоліки Room:

- Надмірність для простих завдань: Використання Room може бути надмірним для збереження простих налаштувань або невеликих обсягів даних. У таких випадках SharedPreferences або файлові системи можуть бути більш підходящими.
- Абстракція: Високий рівень абстракції може приховувати деталі виконання запитів, що може бути недоліком для оптимізації складних запитів.

Реляційні бази даних, такі як SQLite і Room, забезпечують потужні можливості для зберігання і управління структурованими даними у мобільних застосунках. SQLite надає низькорівневий доступ до бази даних і дозволяє

виконувати складні SQL-запити, тоді як Room спрощує роботу з базою даних, надаючи об'єктно-реляційне відображення і інтеграцію з Android Architecture Components. Вибір між SQLite і Room залежить від вимог проекту, складності даних і необхідного рівня абстракції.

1.3 Постановка завдання

Мета кваліфікаційної роботи — вивчення засобів організації персистентності даних у мобільних застосунках для платформи Android.

Постановка завдання дослідження включає наступні етапи:

гляд і аналіз існуючих рішень: розгляд та порівняння існуючих засобів для організації баз даних у операційній системі Android.

ивчення SQLiteDatabase, Cursor та SQLiteOpenHelper: огляд доступних методів, вивчення документації, ознайомлення з основними принципами роботи та можливостями.

ормулювання функціональних вимог до Android-застосунку, який буде демонструвати можливості по організації та роботі з вбудованою СКБД SQLite.

роектування застосунку: на цьому етапі визначається загальна архітектура програмного продукту, включаючи вибір архітектурного підходу, розподіл функціональності між компонентами застосунку та визначення способів комунікації між ними.

оделювання схеми даних Android-застосунку.

роектування інтерфейсу користувача: на цьому етапі розробляється дизайн інтерфейсу користувача, визначається, які екрани будуть у застосунку, які елементи керування будуть на кожному з екранів, та як вони будуть взаємодіяти між собою.

онструювання застосунку: на цьому етапі виконується написання коду застосунку відповідно до визначених раніше вимог і архітектури. Це включає розробку класів, які взаємодіють з базою

даних, створення відповідних функцій для додавання, читання, оновлення та видалення даних, а також реалізацію логіки взаємодії з користувачем.

Кожен із цих етапів важливий для успішного створення Android-застосунку з використанням вбудованої бази даних. Після завершення всіх етапів застосунок може бути підготовлений для апробації.

2 ДОСЛІДЖЕННЯ ПРОГРАМНИХ ЗАСОБІВ ЗАБЕЗПЕЧЕННЯ ПЕРСИСТЕНТНОСТІ ДАНИХ ADNROID-ЗАСТОСУНКА

2.1 Аналіз технологій, використаних при створенні Android-застосунку

Розробка Android-застосунків вимагає використання широкого спектру інструментів та технологій, які забезпечують функціональність, персистентність даних, інтерактивність та безпеку.

Мова програмування

Kotlin – це сучасна мова програмування, яка була офіційно підтримана Google для розробки Android-застосунків у 2017 році [9]. Kotlin забезпечує високу продуктивність, захист від помилок (завдяки null-безпеці) та кращу синтаксичну структуру, що дозволяє писати менш об'ємний та більш читабельний код.

Основні переваги Kotlin:

- Null-безпека: Kotlin має вбудовані механізми для уникнення помилок, пов'язаних з null значеннями, що є однією з найпоширеніших проблем у Java.
- Лаконічний синтаксис: код на Kotlin зазвичай коротший та більш зрозумілий порівняно з Java.
- Інтероперабельність з Java: Kotlin повністю сумісний з Java, що дозволяє використовувати обидві мови в одному проекті.
- Функціональне програмування: Kotlin підтримує функціональні стилі програмування, що дозволяє писати більш виразний та зрозумілий код.

Нижче показаний приклад, який показує, як Kotlin робить код більш лаконічним та безпечним завдяки вбудованим механізмам роботи з null-значеннями та можливостям функціонального програмування.

Лістинг 1 *Приклад коду на Kotlin*

```
@Entity(tableName = "users")
data class User(
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
    val name: String,
    val email: String? = null // Nullable field
)
@Dao
interface UserDao {
    @Query("SELECT * FROM users")
    fun getAllUsers(): LiveData<List<User>>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertUser(user: User)
}
@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
// Usage example
fun main() {
    val db = Room.databaseBuilder(
        applicationContext,
        AppDatabase::class.java, "database-name"
    ).build()

    val userDao = db.userDao()

    // Inserting a user using coroutine
    GlobalScope.launch {
```

```
        userDao.insertUser(User(name = "John Doe", email =
"john.doe@example.com"))
    }

    // Observing LiveData with null-safe operator
    userDao.getAllUsers().observe(this, Observer { users ->
        users?.forEach { user ->
            println("User: ${user.name}, Email:
${user.email ?: "No Email"}")
        }
    })
```

Null-безпечні оператори:

- Поле email оголошене як nullable (String?).
- Використання оператора ?: для надання значення за замовчуванням, якщо email є null (user.email ?: "No Email").

Функціональне програмування:

- Використання корутин (GlobalScope.launch) для асинхронного виконання операцій з базою даних.
- Використання лямбда-функції у observe для обробки змін у

Серидовище розробки (IDE)

Android Studio — це офіційне інтегроване середовище розробки (IDE) для Android від Google [10]. Воно базується на IntelliJ IDEA і надає всі необхідні інструменти для розробки, налагодження та тестування Android-застосунків.

Основні можливості Android Studio:

- Редактор коду: Підтримка коду Java та Kotlin з підсвічуванням синтаксису, автодоповненням та рефакторингом.

- Емулятор Android: Інтегрований емулятор для тестування додатків на різних версіях Android та різних пристроях.
- Інструменти для налагодження: Можливість налагодження коду, моніторингу продуктивності та аналізу використання пам'яті.
- Gradle: Інтеграція з Gradle для автоматизації збірки проектів, управління залежностями та конфігурації проектів.

СКБД SQLite та інструмент для перегляду даних

SQLite є вбудованою реляційною базою даних, яка дозволяє зберігати дані у вигляді реляційних таблиць. Вона забезпечує потужні можливості для роботи з даними, підтримуючи складні SQL-запити та транзакції.

У проєктах Android база даних SQLite зазвичай зберігається у внутрішній пам'яті застосунка. Файл бази даних знаходиться в директорії data/data/<package_name>/databases/. Ця директорія є приватною для застосунка, що забезпечує безпеку даних.

Наприклад, якщо ваш застосунок має ім'я com.example.myapp, файл бази даних буде знаходитися за адресою: /data/data/com.example.myapp/databases/.

DB Browser для SQLite — це інструмент з графічним інтерфейсом користувача, який дозволяє переглядати, створювати та редагувати файли бази даних SQLite [11]. Він забезпечує зручний спосіб роботи з базами даних, що особливо корисно для розробників під час тестування та налагодження застосунків.

Основні можливості DB Browser (SQLite):

- Створення та редагування структур таблиць
- Внесення даних до таблиць
- Виконання SQL-запитів
- Перегляд та експорт даних

Система побудови та управління проєктом:

Gradle — це система автоматичного збирання, яка використовується для управління залежностями, компіляцією коду, тестуванням і створенням

пакетів для застосунків [12]. Gradle забезпечує гнучкість та можливість налаштування процесів збірки, що дозволяє ефективно управляти проєктами будь-якої складності.

Цей набір технологій забезпечує сучасний підхід до розробки, дозволяючи створювати ефективні та надійні Android-застосунки.

2.2 Засоби Android для роботи з СКБД SQLite

У розробці Android-застосунків, для роботи з базами даних SQLite використовуються кілька важливих класів та інструментів, які полегшують управління даними. Нижче розглянемо основні з них.

Клас SQLiteOpenHelper забезпечує зручний спосіб створення, оновлення та управління базами даних. Він включає такі основні методи:

- onCreate(SQLiteDatabase db): викликається при першому створенні бази даних. Використовується для створення початкових таблиць та початкових даних.
  ion, int newVersion): викликається при оновленні версії бази даних. Використовується для внесення змін у структуру бази даних.

```
class MyDatabaseHelper(context: Context) :
SQLiteOpenHelper(context, DATABASE_NAME, null,
DATABASE_VERSION) {



        // Створення таблиці
```

```
        // Оновлення бази даних
```

Лістинг 2. *Приклад використання SQLiteOpenHelper*

Клас **SQLiteDatabase** представляє базу даних та забезпечує методи для виконання CRUD-операцій (створення, читання, оновлення, видалення):

додає новий рядок у таблицю.

- String groupBy, String having, String orderBy): виконує запит до таблиці.

  hereArgs): оновлює існуючі рядки у таблиці.
- delete(String table, String whereClause, String[] whereArgs): видаляє рядки з таблиці.

```
val dbHelper = MyDatabaseHelper(context)
val db = dbHelper.writableDatabase

val values = ContentValues().apply {
    put("name", "John Doe")
    put("email", "john.doe@example.com")
}

val newRowId = db.insert("users", null, values)
```

Лістинг 3. *Приклад використання SQLiteDatabase для вставки даних*

Клас **Cursor** використовується для читання даних з бази даних. Він забезпечує доступ до результатів запиту:

```
val cursor = db.query(
    "users",   // Таблиця
    arrayOf("id", "name", "email"), // Колонки
    null, // Умови вибору (WHERE)
    null, // Аргументи вибору
    null, // Групування
```

```
    null, // Умови групування
    null  // Сортування
)

with(cursor) {
    while (moveToNext()) {
        val id = getInt(getColumnIndexOrThrow("id"))
        val name = getString(getColumnIndexOrThrow("name"))
        val email =
getString(getColumnIndexOrThrow("email"))
        // Використання даних
    }
}
cursor.close()
```

Лістинг 4. *Приклад використання курсорів*

2.2.1 Клас SQLiteOpenHelper

SQLiteOpenHelper — це допоміжний клас у Android SDK, який полегшує створення, оновлення та управління базами даних SQLite. Він надає методи для створення та управління версіями бази даних, що робить процес роботи з базою даних більш організованим та безпечним.

Сигнатура класу **SQLiteOpenHelper:**

```
public abstract class SQLiteOpenHelper implements
AutoCloseable {
    private static final String TAG =
SQLiteOpenHelper.class.getSimpleName();
    private final Context mContext;
    @UnsupportedAppUsage
    private final String mName;
    private final int mNewVersion;
    private final int mMinimumSupportedVersion;
```

```
    private SQLiteDatabase mDatabase;
    private boolean mIsInitializing;
    private                SQLiteDatabase.OpenParams.Builder
mOpenParamsBuilder;
...
 }
```

Лістинг 1. *Сигнатура класу SQLiteOpenHelper*

Пояснення змінних класу:

- Context mContext: Контекст додатка, який передається до конструктора. Він використовується для доступу до системних ресурсів та файлової системи додатка.
- String mName: Ім'я файлу бази даних. Якщо значення null, база даних буде створена в пам'яті (in-memory).
- int mNewVersion: Нова версія бази даних, що використовується для оновлення бази даних, коли структура змінюється.
- SQLiteDatabase mDatabase: Об'єкт бази даних SQLite, що використовується для виконання SQL-операцій.
- boolean mIsInitializing: Прапорець, що вказує, чи виконується ініціалізація бази даних.
  atabase.OpenParams.Builder mOpenParamsBuilder: Об'єкт для побудови параметрів відкриття бази даних.

Методи SQLiteOpenHelper та їх використання:

Методи, що використовуються для створення/відкриття/закриття бази даних:

онструктор: Викликається для створення екземпляра

Лістинг 2. *Приклад виклику екземпляра*

Пояснення параметрів конструктора:

Контекст застосунка, який надає доступ до системних ресурсів, файлової системи та дозволяє виконувати операції, такі як створення нових компонентів

- String name: Ім'я файлу бази даних. Якщо значення null, база даних буде створена в пам'яті (in-memory), тобто дані не будуть збережені після закриття застосунка.
- SQLiteDatabase.CursorFactory factory: Фабрика курсорів, яка дозволяє розробникам створювати власні об'єкти курсорів для керування результатами запитів. Може бути null, тоді використовується стандартна реалізація.

  n: Версія бази даних. Використовується для управління оновленнями структури бази даних. При зміні версії викликається метод onUpgrade.

етод getReadableDatabase() використовується для отримання об'єкта бази даних у режимі читання. Він повертає "огорнутий" об'єкт SQLiteDatabase, яким керує об'єкт SQLiteOpenHelper.

Пояснення:

- getReadableDatabase(): Метод, що повертає об'єкт SQLiteDatabase у режимі тільки для читання. Це означає, що цей об'єкт можна використовувати для виконання SELECT-запитів та інших операцій, які не змінюють дані в базі.
- Виконання SELECT-запиту: awQuery(String sql, String[] selectionArgs): Метод rawQuery виконує SQL-запит та повертає об'єкт Cursor, який містить результати цього запиту.

Якщо для об'єкта, отриманого з getReadableDatabase(), спробувати виконати запит, який змінює дані (наприклад, INSERT, UPDATE або DELETE), то база даних буде автоматично відкриватися у режимі читання-

запису. Це означає, що ви зможете виконувати запити на зміну даних, але це може вплинути на продуктивність і цілісність даних.

Лістинг 3. *Приклад команди getReadableDatabase*

етод getWritableDatabase() використовується для отримання об'єкта бази даних у режимі запису. Він дозволяє виконувати операції, які змінюють дані в базі даних, такі як вставка, оновлення та видалення. Метод, повертає об'єкт SQLiteDatabase у режимі читання-запису. Це означає, що через цей об'єкт можна виконувати всі операції з базою даних, включаючи ті, які змінюють дані.

Лістинг 4. *Приклад команди getWritableDatabase*

Клас ContentValues використовується для зберігання пар "ключ-значення", де ключі — це імена колонок, а значення — дані, які потрібно вставити або оновити.

lose(): Закриває базу даних, "огорнену" об'єктом SQLiteOpenHelper звільняючи всі ресурси.

```
public synchronized void close() {
    if (mDatabase != null && mDatabase.isOpen()) {
        mDatabase.close();
    }
```

Лістинг 5. *Приклад команди close*

Методи життєвого циклу БД:

nCreate(SQLiteDatabase db): Викликається при першому створенні бази даних. Використовується для створення схеми даних.

Лістинг 6. *Приклад для створення таблиць*

Викликається при оновленні бази даних (наприклад, при зміні схеми). Використовується для обробки змін структури бази даних між версіями.

```
public void onUpgrade(SQLiteDatabase db, int oldVersion,
int newVersion) {
    db.execSQL("DROP TABLE IF EXISTS expenses");
    onCreate(db);
}
```

Лістинг 7. *Приклад команди для оновлення бази даних*

nOpen(SQLiteDatabase db): Викликається щоразу при відкритті бази даних. Використовується для виконання налаштувань або початкових операцій.

```
    // Додаткові операції при відкритті бази даних
```

Лістинг 8. *Приклад команди для onOpen*

Таким чином, SQLiteOpenHelper забезпечує організований та безпечний спосіб роботи з базами даних у Android-застосунках, спрощуючи процеси створення, оновлення та управління базами даних.

2.2.2. Клас SQLiteDatabase

Методи виконання запитів:

xecSQL(String sql): Використовується для виконання SQL-команд, які не повертають дані (наприклад, CREATE, INSERT, UPDATE,

Лістинг 9. *Приклад виконання SQL-команд*

): Використовується для виконання SELECT-запитів, що повертають Cursor з результатами.

Лістинг 10. *Приклад виконання SELECT-запитів*

3. insert(String table, String nullColumnHack, ContentValues values): Використовується для вставки нового рядка в таблицю.

Лістинг 11. *Приклад команди insert*

update(String table, ContentValues values, String whereClause, String[] whereArgs): Використовується для оновлення рядків у таблиці.

```
ContentValues values = new ContentValues();
values.put("amount", 150.0);
int count = db.update("expenses", values, "id = ?", new
String[] {"1"});
```

Лістинг 12. *Приклад команди update*

5. delete(String table, String whereClause, String[] whereArgs): Використовується для видалення рядків з таблиці.

```
int count = db.delete("expenses", "id = ?", new String[]
{"1"});
```

Лістинг 13. *Приклад використання delete*

Методи підтримки транзакцій

У SQLite, транзакції забезпечують цілісність даних, дозволяючи групувати кілька операцій у межах однієї транзакції. Якщо будь-яка з операцій у транзакції не вдається, всі зміни можуть бути відкочені, щоб забезпечити консистентність бази даних. Основні методи для управління транзакціями в класі SQLiteDatabase включають:

- beginTransaction(): Починає нову транзакцію.
- setTransactionSuccessful(): Викликається для позначення транзакції як успішної. Тільки після цього транзакція буде зафіксована.
- endTransaction(): Завершує поточну транзакцію. Якщо транзакція була успішною, зміни фіксуються, інакше відкочується.
- inTransaction(): Перевіряє, чи поточна транзакція активна.

Нижче наведено приклад використання цих методів для організації транзакції:

```
val db = getWritableDatabase()
try {
    db.beginTransaction()
    val values1 = ContentValues().apply {
        put("type", "expense")
        put("amount", 50.0)
        put("description", "Dinner")
        put("date", "2023-06-01")
    }
    db.insert("expenses", null, values1)
    val values2 = ContentValues().apply {
        put("type", "expense")
        put("amount", 30.0)
        put("description", "Transport")
        put("date", "2023-06-02")
```

```
    }
    db.insert("expenses", null, values2)
    db.setTransactionSuccessful()
} catch (e: Exception) {
    e.printStackTrace()
} finally {
    db.endTransaction()
}
if (db.inTransaction()) {
    println("Транзакція активна")
} else {
    println("Транзакція неактивна")
}
```

Лістинг 14. *Приклад використання методів для організації транзакції*

Пояснення:

(): Метод викликається для початку нової транзакції. Це забезпечує, що всі наступні операції будуть виконані в межах однієї транзакції.

(): Метод викликається, щоб позначити транзакцію як успішну. Лише після цього зміни будуть зафіксовані.

(): Метод завершує транзакцію. Якщо транзакція була позначена як успішна (setTransactionSuccessful() було викликано), зміни фіксуються. В іншому випадку, всі зміни будуть відкочені.

(): Метод перевіряє, чи є активна транзакція.

Методи роботи з БД цілком:

- openDatabase(String path, SQLiteDatabase.CursorFactory factory, int flags):
- Відкриває або створює базу даних за вказаним шляхом.
- close(): Закриває базу даних, звільняючи всі ресурси.

- getVersion(): Повертає версію бази даних.
- setVersion(int version): Встановлює версію бази даних.

2.2.3. Інтерфейс Cursor:

Інтерфейс Cursor використовується для роботи з результатами SQL-запитів. Він надає методи для навігації по результатах запиту та отримання даних з кожного рядка. Основна мета інтерфейсу Cursor — забезпечити зручний доступ до даних, отриманих в результаті виконання SQL-запитів.

Cursor дозволяє розробникам:

- Переміщуватися по рядках результатів.
- Отримувати дані з поточного рядка за індексом колонки або за ім'ям колонки.
- Перевіряти кількість рядків та колонок у результатах запиту.

Об'єкти **Cursor** отримуються шляхом виконання SQL-запитів за допомогою методів класу **SQLiteDatabase**, таких як **query()**, **rawQuery()**, або через використання **ContentResolver** для доступу до контент-провайдерів. Метод **query()**: Виконує запит до бази даних та повертає об'єкт **Cursor**. Метод **rawQuery()**: Виконує довільний SQL-запит та повертає об'єкт **Cursor**. Отримання **Cursor** за допомогою **ContentResolver**: Використовується для доступу до даних через контент-провайдери.

Методи інтерфейсу Cursor:

- moveToFirst(): Переміщує курсор на перший рядок результатів. oveToNext(): Переміщує курсор на наступний рядок результатів.
- moveToLast(): Переміщує курсор на останній рядок результатів.
- moveToPosition(int position): Переміщує курсор на вказану позицію.
- getColumnIndex(String columnName): Повертає індекс вказаної колонки.

- getString(int columnIndex): Повертає значення вказаної колонки як рядок.
- getInt(int columnIndex): Повертає значення вказаної колонки як ціле число.
- getDouble(int columnIndex): Повертає значення вказаної колонки як число з плаваючою точкою.

Клас ContentValues:

Клас ContentValues використовується для зберігання пар "ключ-значення", що представляють дані, які вставляються або оновлюються в базі даних. Він "огортає" приватну колекцію ArrayMap<String, Object>, де ключі є назвами колонок, а значення — відповідними даними, які зберігаються в цих колонках.

Основне призначення класу ContentValues — забезпечити зручний спосіб роботи з даними, що вставляються або оновлюються в таблицях бази даних SQLite. Це робить код більш читабельним та організованим, дозволяючи розробникам додавати дані у вигляді пар "ключ-значення".

Методи для додавання значень:

- put(String key, String value): Додає значення типу String.
- put(String key, Integer value): Додає значення типу Integer.
- put(String key, Double value): значення типу Double.
- put(String key, Boolean value): Додає значення типу Boolean.
- clear(): Очищує всі значення у ContentValues.

Таким чином, класи SQLiteDatabase, Cursor та ContentValues забезпечують ефективний і зручний інструментарій для роботи з базами даних у Android-застосунках, надаючи різноманітні методи для виконання запитів, підтримки транзакцій та обробки результатів.

SQLite є потужним та надійним інструментом для зберігання даних у мобільних застосунках. Клас SQLiteOpenHelper значно спрощує процес створення, оновлення та управління базами даних, забезпечуючи розробникам

зручний та ефективний спосіб роботи з реляційними базами даних у середовищі Android. Завдяки своїй простоті та функціональності, SQLite залишається одним з найпопулярніших виборів для реалізації персистентності даних у мобільних застосунках.

3 Розробка Android-застосунку з організацією персистентності із використанням SQLite

3.1 Вимоги до застосунку

Смартфони та планшети надають користувачам доступ до безлічі сервісів, зокрема, застосунки для управління особистими фінансами набули великої популярності завдяки їх здатності допомагати користувачам слідкувати за своїми витратами, доходами, створювати бюджети та досягати фінансових цілей.

Згідно зі статистикою, ринок мобільних застосунків для управління фінансами зростає з кожним роком [13]. Це зумовлено зростаючим попитом на зручні та ефективні інструменти, які дозволяють користувачам контролювати свої фінанси в реальному часі. Багато людей прагнуть покращити своє фінансове становище, що робить такі застосунки надзвичайно затребуваними. Фінансові застосунки допомагають користувачам уникати зайвих витрат, планувати великі покупки та досягати фінансової стабільності.

Функції фінансових застосунків:

- Моніторинг витрат: Користувачі можуть реєструвати всі свої витрати, від невеликих покупок до великих витрат, що дозволяє отримати повну картину своїх фінансових операцій.
- Моніторинг доходів: Користувачі можуть відстежувати свої доходи з різних джерел, таких як заробітна плата, бонуси, інвестиційні прибутки тощо.
- Бюджетування: Застосунки дозволяють створювати бюджети для різних категорій витрат, що допомагає уникнути перевитрат та ефективно планувати фінанси.
- Аналіз та звіти: Інтерактивні звіти та графіки допомагають користувачам аналізувати свої фінансові звички та виявляти області для покращення.

- Управління рахунками: Застосунки дозволяють управляти різними рахунками, такими як банківські рахунки, кредитні картки, електронні гаманці тощо.
- Нагадування та повідомлення: Застосунки можуть надсилати користувачам нагадування про оплату рахунків або досягнення фінансових цілей.

Предметною областю даного проекту є управління особистими фінансами. Основною задачею застосунка є надання користувачам можливості ефективно відстежувати свої витрати та доходи, управляти рахунками, створювати бюджети та отримувати статистичну інформацію про свої фінансові операції. Застосунок повинен бути інтуїтивно зрозумілим, безпечним та надійним, забезпечуючи користувачам зручний інтерфейс та необхідні функції для управління фінансами.

Основні поняття та категорії:

- Користувач: Фізична особа, яка використовує застосунок для управління своїми фінансами. Кожен користувач має свій обліковий запис, де зберігаються всі його фінансові дані.
- Рахунок: Місце зберігання фінансових коштів користувача. Це може бути банківський рахунок, готівка, електронний гаманець або інший тип рахунку. Користувач може мати кілька рахунків для різних цілей.
- Витрати: Грошові суми, які користувач витрачає на різні потреби. Витрати можуть бути класифіковані за категоріями (наприклад, харчування, транспорт, розваги) для більш детального аналізу.
- Доходи: Грошові суми, які користувач отримує з різних джерел. Доходи також можуть бути класифіковані за категоріями (наприклад, заробітна плата, дивіденди, подарунки).

- Категорії витрат: Групи, в які об'єднуються витрати за певними критеріями. Це допомагає користувачам легше відстежувати свої витрати за різними сферами життя.
- Категорії доходів: Групи, в які об'єднуються доходи за певними критеріями. Це дозволяє користувачам аналізувати джерела своїх доходів.
- Транзакція: Фінансова операція, яка включає витрату або надходження коштів на певний рахунок. Кожна транзакція містить інформацію про дату, суму, категорію та опис.

Функціональні вимоги:

- Управління рахунками (додавання, редагування, видалення).
- Управління витратами та доходами (додавання, редагування, видалення).
- Забезпечення цілісності даних та підтримка транзакцій.
- Генерація звітів та статистики.
- Інтуїтивно зрозумілий та зручний інтерфейс.

Нижче наведені основні вимоги до функціоналу застосунку:

1. Зберігання даних
    - Фінансові операції: Зберігати інформацію про всі фінансові операції користувача, включаючи суму, тип операції (витрата чи дохід), дату, опис та пов'язаний рахунок.
    - Категорії витрат і доходів: Зберігати типи фінансових операцій для категоризації витрат і доходів.
    - Рахунки користувача: Зберігати інформацію про банківські рахунки, кредитні картки, електронні гаманці, включаючи назву рахунку та його баланс.

правління рахунками

- Користувачі повинні мати можливість додавати нові рахунки, редагувати існуючі та видаляти рахунки.

- Відображення загального балансу для всіх рахунків.

атегоризація фінансових операцій

- Можливість призначення категорій для кожної фінансової операції (наприклад, "Продукти", "Розваги", "Зарплата").
- Відображення статистики по категоріях витрат і доходів.

ористувацький інтерфейс

- Головний екран: Відображення поточного балансу всіх рахунків, останніх транзакцій та швидкий доступ до основних функцій.
- Екран додавання операцій: Зручний інтерфейс для додавання нових витрат або доходів, включаючи вибір категорії, рахунку та дати.
- Екран перегляду рахунків: Список всіх рахунків з можливістю додавання, редагування та видалення.
- Статистичний екран: Відображення графіків та діаграм для аналізу витрат і доходів по категоріях та рахунках.

Нефункціональні вимоги:

родуктивність

- Швидке завантаження та відображення даних.
- Ефективне виконання запитів до бази даних навіть при великій кількості записів.

адійність

- Забезпечення безперебійної роботи застосунку.
- Використання транзакцій для забезпечення цілісності даних при виконанні фінансових операцій.

3. Юзабіліті
   - Інтуїтивно зрозумілий інтерфейс.
   - Легкість використання для користувачів будь-якого рівня технічної підготовки.

Цей застосунок допомагає користувачам досягати своїх фінансових цілей, покращувати фінансову грамотність та зменшувати витрати.

Забезпечення персистентності даних за допомогою SQLite дозволяє зберігати важливу інформацію локально на пристрої користувача, що робить застосунок доступним навіть в умовах відсутності Інтернет-з'єднання. SQLite надає потужні можливості для управління даними, підтримуючи складні SQL-запити та транзакції, що забезпечує цілісність та надійність даних.

Таким чином, розробка застосунку з використанням сучасних технологій та підходів до організації персистентності даних відповідає потребам користувачів у ефективному управлінні особистими фінансами. Дотримання визначених вимог дозволяє створити стабільний, безпечний та зручний у використанні продукт, який сприяє покращенню фінансового становища користувачів.

3.2 Архітектура застосунка

Архітектура мобільного застосунка є одним з найважливіших аспектів, який впливає на його ефективність, підтримку, масштабованість та зручність у використанні. Добре спроектована архітектура забезпечує розділення відповідальностей, покращує тестованість та сприяє повторному використанню коду. У цьому розділі розглядається архітектура застосунка для управління особистими фінансами, визначаютьсяосновні компоненти та їх взаємодія, а також описується використання SQLite для забезпечення персистентності дани в цьому застунку.

Архітектурний підхід

Для побудови застосунка буде використано архітектурний підхід Model-View-ViewModel (MVVM), який є рекомендованим для розробки Android-застосунків. MVVM забезпечує чітке розділення відповідальностей між різними компонентами застосунка, що полегшує їх тестування та підтримку [14].

Основні компоненти архітектури

Архітектура застосунку складається з декількох основних шарів: Model, View та ViewModel, кожен з яких має свою роль у забезпеченні функціональності застосунка:

Model — Шар Model відповідає за управління даними та бізнес-логіку застосунка. Він включає в себе класи, Expences, Type та Account, які представляють схему даних.

Опис основних частин файлу Gradle:

Файл build.gradle є основним файлом конфігурації проекту, який визначає залежності, налаштування компілятора та інші параметри. Нижче наведено файл основних частин build.gradle для проекту Android:

```
android {
    compileSdk = 34
defaultConfig {
        minSdk = 26
        targetSdk = 34
        versionCode = 1
        versionName = "1.0" //
    }
    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_1_8
        targetCompatibility = JavaVersion.VERSION_1_8
    }
    kotlinOptions {
        jvmTarget = "1.8"
    }
    composeOptions {
        kotlinCompilerExtensionVersion = "1.5.1"
    }
dependencies {
    implementation 'androidx.appcompat:appcompat:1.3.0'
```

```
    implementation
'androidx.constraintlayout:constraintlayout:2.0.4'
    implementation
'androidx.recyclerview:recyclerview:1.2.0'
    implementation 'androidx.lifecycle:lifecycle-
extensions:2.2.0'
    implementation
'com.google.android.material:material:1.3.0'
    testImplementation 'junit:junit:4.13.2'
    androidTestImplementation
'androidx.test.ext:junit:1.1.2'
    androidTestImplementation
'androidx.test.espresso:espresso-core:3.3.0'
}
```

Лістинг 15. *Вміст файлу build.gradle*

Детальний опис файлу Gradle:

1. android: основний блок конфігурації Android-проекту.
   - compileSdkVersion 34: версія SDK, яка використовується для компіляції проекту.
   - defaultConfig: містить основні параметри конфігурації проекту:
   - m
   - i
   - ersionCode 1: внутрішній номер версії додатку.
   - VersionName "1.0": відображувана версія додатку.
   - SourceCompatibility: Сумісність вихідного коду з Java 8.
   - TargetCompatibility: Сумісність цільового коду з Java 8.
   - JvmTarget: JVM для Kotlin.
   - KotlinCompilerExtensionVersion: Версія розширення компілятора для Jetpack Compose.
     k
     V
     o
     n

4. dependencies: визначає залежності проекту.

: Material Design library версії 1.4.0

Core KTX бібліотека

Lifecycle Runtime KTX бібліотека

: Compose UI бібліотека

: Compose UI Graphics бібліотека

- implementation(libs.androidx.ui.tooling.preview): Інструменти попереднього перегляду для Compose

: Material Design 3 бібліотека

AppCompat бібліотека

: Media3 Common бібліотека

Файлова структура проекту Android організована таким чином, щоб забезпечити легкість навігації та управління різними компонентами проекту. На рисунках 1 та 2 представлена файлова структура застосунку.

Опис файлів та їх призначення:

- ExpensesDatabaseHelper.kt: клас, що забезпечує управління базою даних SQLite, включаючи створення таблиць, оновлення структури бази даних та виконання CRUD-операцій.
- MainActivity.kt: головна активність застосунку, яка відображає основний інтерфейс користувача.

sActivity.kt: активність для управління рахунками користувача.

Іідображати дані з бази даних у вигляді списку, розширює CursorAdapter та містить логіку для прив'язки даних курсора до елементів списку в лейаутах.

: дата-клас для представлення витрат.

- Account.kt: дата-клас для представлення рахунків.

дата-клас визначає тип витрати.

- account_item.xml: XML-розмітка елемента акаунту в списку акаунтів.

- activity_main.xml: XML-розмітка головного екрану.
  s.xml: XML-розмітка екрану управління рахунками.
- dialog_add_account.xml: XML-розмітка додавання нового акаунту.
- dialog_add_expense.xml: XML-розмітка діалогового вікна для додавання нової витрати або доходу.
- dialog_edit_expense.xml: XML-розмітка додавання нової витрати.
- expense_list_item.xml: XML-розмітка визначає вигляд окремого елемента витрати в списку.
- history_item.xml: XML-розмітка для відображення історії витрат.
- strings.xml: файл, що містить текстові ресурси застосунку.
  файл, що містить стилі застосунку.
- AndroidManifest.xml: маніфест файлу, що визначає основні компоненти додатку, такі як активності, сервіси, приймачі тощо

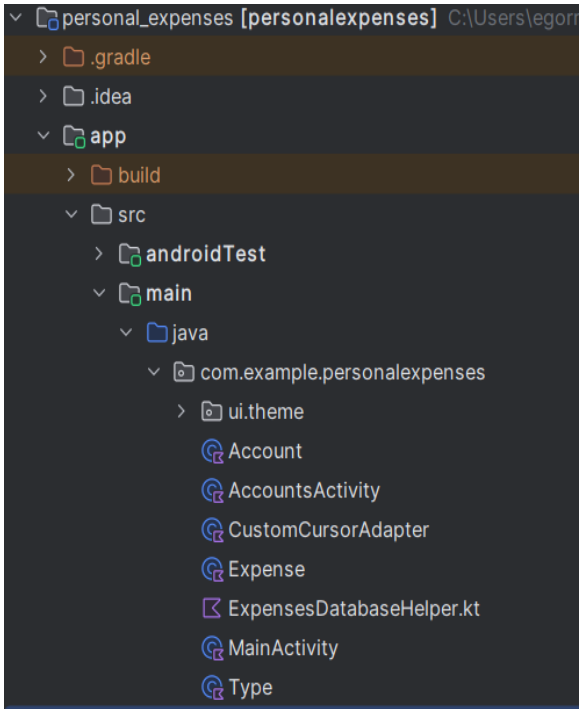

Рис.1. *Файлова структура головних компонентів*

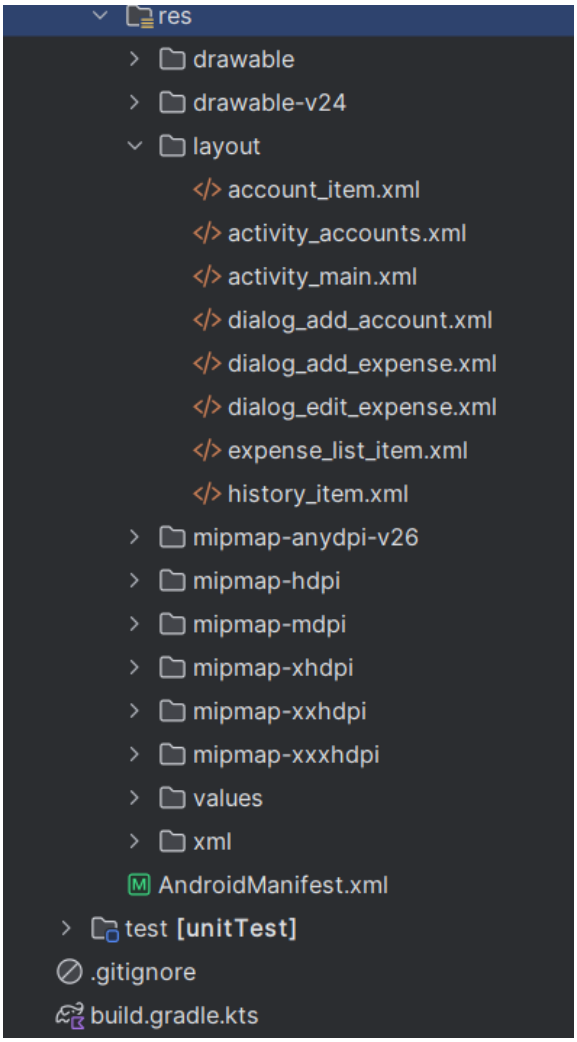

Рис.2. *Файлова структура головних сторінок*

3.3 Проектування бази даних

Проектування бази даних є ключовим етапом розробки застосунка, який забезпечує надійне зберігання, швидкий доступ та обробку даних. У цьому розділі розглядається детальне проектування бази даних для Android-застосунку з управління особистими фінансами, який використовує SQLite як механізм зберігання даних. Визначаються основні таблиці, їх атрибути, зв’язки між таблицями.

Основні вимоги до бази даних:

Для забезпечення ефективного управління фінансами користувача необхідно зберігати різноманітні типи даних, такі як інформація про витрати, доходи, рахунки користувача, категорії витрат та доходів. Основні вимоги до бази даних включають:

- Зберігання витрат та доходів: Необхідно зберігати інформацію про всі фінансові операції користувача, включаючи суму, тип операції (витрата чи дохід), дату, опис та пов'язаний рахунок.
- Управління рахунками: Користувачі можуть мати кілька рахунків, таких як банківські рахунки, кредитні картки, електронні гаманці. Необхідно зберігати інформацію про назву рахунку та його баланс.
- Категоризація фінансових операцій: Для більш детального аналізу витрат та доходів необхідно мати можливість категоризувати фінансові операції.
- Зв’язки між таблицями: Для забезпечення цілісності даних необхідно визначити зв’язки між таблицями, такі як зв'язок між витратами/доходами та рахунками.

Схема бази даних показана на 2 рисунку.

Для зручності та ефективності зберігання даних ми створено наступні таблиці:

- accounts: Зберігає інформацію про рахунки користувача.
- expenses: Зберігає інформацію про витрати.
- types: Зберігає типи транзакцій (дохід або витрата).

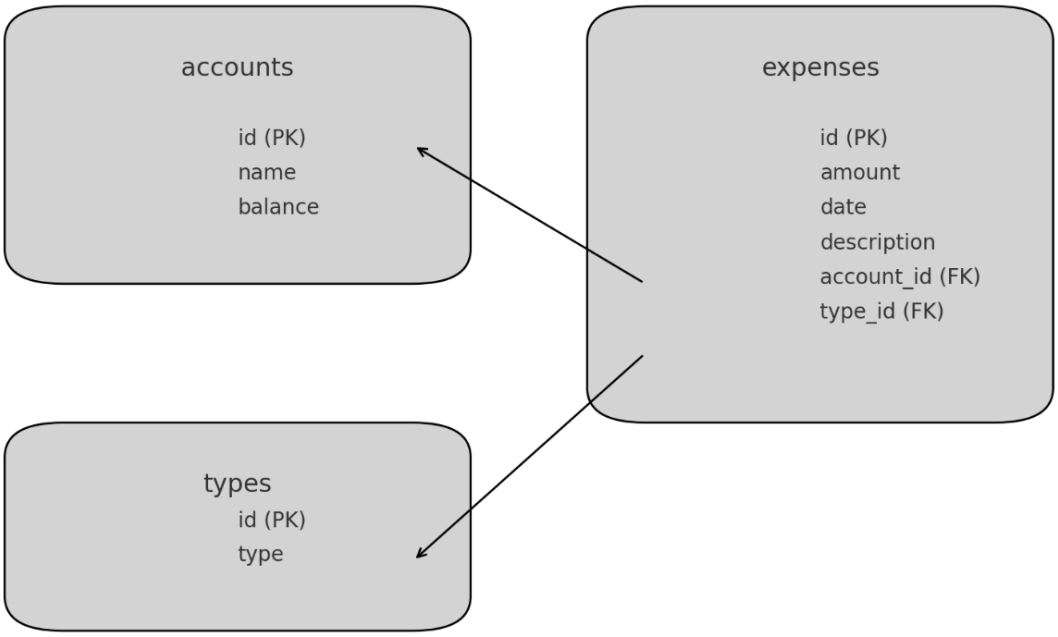

Рис.*3. Схема бази даних*

ccounts: Таблиця для зберігання інформації про рахунки користувача.

- id (PRIMARY KEY): Унікальний ідентифікатор рахунку.
- name: Назва рахунку.

Поточний баланс рахунку.

xpenses: Таблиця для зберігання інформації про витрати.

- id (PRIMARY KEY): Унікальний ідентифікатор витрати.
- amount: Сума витрати.
- date: Дата витрати.
- description: Опис витрати.
- account_id (FOREIGN KEY): Ідентифікатор рахунку, з якого була здійснена витрата.
- type_id (FOREIGN KEY): Ідентифікатор типу витрати.

ypes: Таблиця для зберігання типів транзакцій (дохід або витрата).

- id (PRIMARY KEY): Унікальний ідентифікатор типу.
- type: Назва типу (income або expense).

Моделі даних:

Це класи, що представляють основні сутності застосунка, такі як Expense (витрата), Income (дохід) та Account (рахунок).

```
data class Expense(
    val id: Long,
    val typeId: Long,
    val amount: Double,
    val date: LocalDateTime,
    val description: String,
    val accountId: Long
)
data class Account(
    val id: Long,
    val name: String,
    val balance: Double
)
data class Type(
    val id: Long,
    val type: String
)
```

Листинг 20. *Класи, що представляють основні сутності застунка*

3.4 Проектування інтерфейсу

Проєктування інтерфейсу користувача (UI) є важливим аспектом розробки Android-застосунку, оскільки від нього залежить зручність і ефективність взаємодії користувача з застосуноком.

Шар View відповідає за відображення даних та взаємодію з користувачем. Він включає всі елементи інтерфейсу користувача (UI), такі як активності та фрагменти.

Основні Вимоги до Інтерфейсу:

Для забезпечення зручності користувача необхідно створити інтерфейс, який дозволить легко виконувати наступні дії:

- Додавання, редагування та видалення витрат і доходів.
- Управління рахунками користувача.
- Перегляд статистики витрат та доходів.
- Навігація між різними екранами додатку.

Основні Компоненти Інтерфейсу:

- головна активність застосунку, яка відображає основний інтерфейс користувача для управління витратами та доходами.
- AccountActivity - активність для управління рахунками користувача.

Основні методи **MainActivity:**

MainActivity - головна активність, яка відображає основний інтерфейс користувача для управління витратами та доходами. Ця активність включає відображення списку витрат та доходів, додавання нових витрат та доходів, а також відображення загальної статистики.

- o
  n
  C
- reate(): Метод, який викликається при створенні активності. Ініціалізує інтерфейс користувача та налаштовує
- setupHistoryListView(): Метод для налаштування ListView, який відображає історію витрат і доходів.
- updateHistory(): Метод для оновлення даних в ListView.
- updateStatistics(): Метод для оновлення статистики витрат та доходів.
- showAddExpenseDialog(): Метод для відображення діалогового вікна додавання нової витрати або доходу.
- showEditDeleteDialog(): Метод для відображення діалогового вікна редагування або видалення витрати.
  l
  e
  savedInstanceState): Метод, який викликається при створенні активності. Ініціалізує інтерфейс користувача та налаштовує

- showEditExpenseDialog(): Метод для відображення діалогового вікна редагування витрати.
- showDeleteAllConfirmationDialog(): Метод для відображення підтвердження видалення всіх даних.робляє запити від View до ViewModel також зберігає стан інтерфейсу та обробляє зміни даних.

MainActivity:

На рисунку 4 показаний основний екран, він відображає список витрат та доходів у вигляді ListView. Також містить кнопки для додавання нової витрати або доходу, видалення всіх записів та перегляду статистики. Діалогове вікно додавання витрати/доходу: містить поля для введення суми, дати, часу, опису та вибору типу (дохід або витрата). Діалогове вікно редагування/видалення витрати: містить поля для редагування даних витрати або доходу, а також кнопки для збереження змін або видалення запису. Статистика: відображає загальну суму доходів та витрат.

Рис.4.*Головний екран застосунку*

На рисунку 5 екран активності для управління рахунками користувача. Вона дозволяє додавати нові рахунки, редагувати та видаляти існуючі, а також переглядати список рахунків.

- Основний екран: Відображає список рахунків у вигляді ListView. Містить кнопку для додавання нового рахунку.
- Діалогове вікно додавання рахунку: Містить поля для введення назви рахунку та початкового балансу.
- Діалогове вікно редагування/видалення рахунку: Містить поля для редагування даних рахунку та кнопки для збереження змін або видалення запису.

Основні методи **AccountActivity:**

- onCreate(Bundle savedInstanceState): Метод, який викликається при створенні активності. Ініціалізує інтерфейс користувача та налаштовує основні компоненти.
- setupAccountListView(): Метод для налаштування ListView, який відображає список рахунків.
- updateAccountList(): Метод для оновлення даних в ListView.
- showAddAccountDialog(): Метод для відображення діалогового вікна додавання нового рахунку.
- showEditDeleteAccountDialog(): Метод для відображення діалогового вікна редагування або видалення рахунку.
- showEditAccountDialog(): Метод для відображення діалогового вікна редагування рахунку.

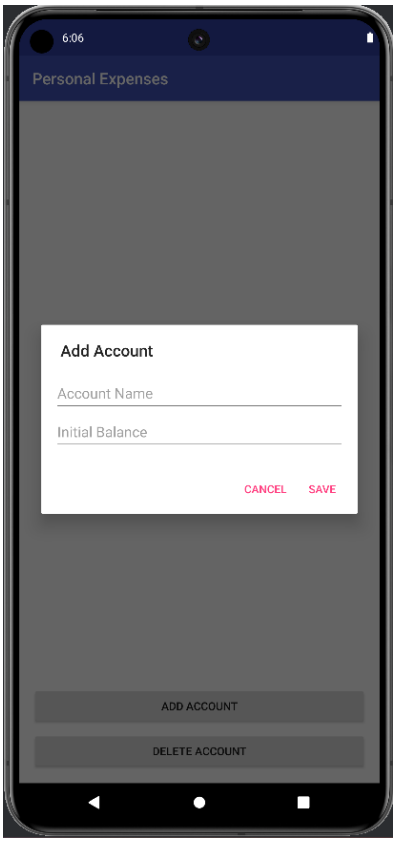

Рис.5.*Екран управління рахунками*

Створення **XML-Лайоутів**

У проєктуванні інтерфейсу користувача для Android-застосунку використовується мова розмітки XML для створення інтерфейсних компонентів. Нижче показані компоненти, які використовуються у XML-лайоутах:

Лайоут

Основний екран MainActivity, код XML розмітки наведений у Додатку А, та на рисунку 4 показаний результат цієї розмітки.

- ListView: Відображає список витрат та доходів.

  Відображає загальну статистику доходів та витрат.
- Button: Кнопки для додавання нової витрати або доходу, видалення всіх записів та завершення роботи застосунку.

Діалогове Вікно Додавання Витрати/Доходу

Діалогове вікно для додавання витрати або доходу, код XML розмітки наведений у Додатку Б, та на рисунку 5 показаний результат цієї розмітки.

- EditText: Поля для введення суми, дати, часу та опису.
- RadioGroup та RadioButton: Вибір типу (дохід або витрата).
- Button: Кнопки для збереження або скасування введених даних.

AccountActivity Лайоут

Основний екран AccountActivity, код XML розмітки наведений у Додатку В та на рисунку 9 показаний релузьтат цієї розмітки.

- ListView: Відображає список рахунків.
- Button: Кнопка для додавання нового рахунку.

Діалогове Вікно Додавання Рахунку

Діалогове вікно для додавання нового рахунку, код XML розмітки наведений у Додатку Д та на рисунку 7 показаний релузьтат цієї розмітки.

- EditText: Поля для введення назви рахунку та початкового балансу.
- Button: Кнопки для збереження або скасування введених даних.

Проєктування інтерфейсу користувача для застосунку з управління особистими фінансами включає створення зручного та інтуїтивно зрозумілого UI, який дозволяє користувачам легко взаємодіяти з застосунком. Використання ViewModel забезпечує ефективну взаємодію між Model та View, зберігає стан інтерфейсу та обробляє зміни даних. Правильна організація інтерфейсу сприяє підвищенню ефективності використання додатку та забезпечує позитивний користувацький досвід.

Результатом розробки є інтерфейс, який дозволяє користувачам можливість управляти витратами та доходами, рахунками, а також переглядати статистику.

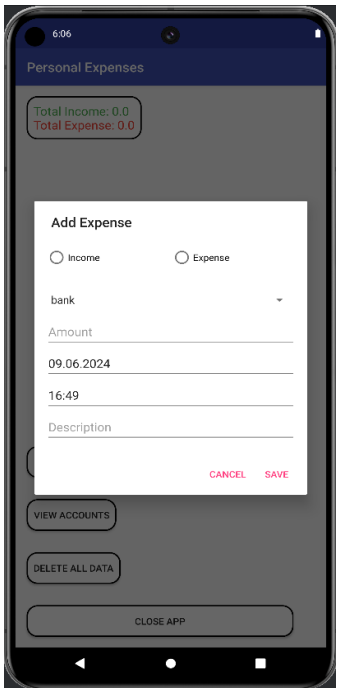

Рис.6.*Вікно додавання витрат/надходжень*

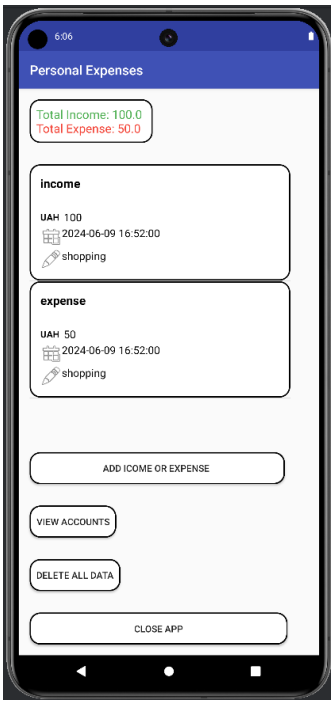

Рис.7.*Головний екран зі списком доданих транзакцій*

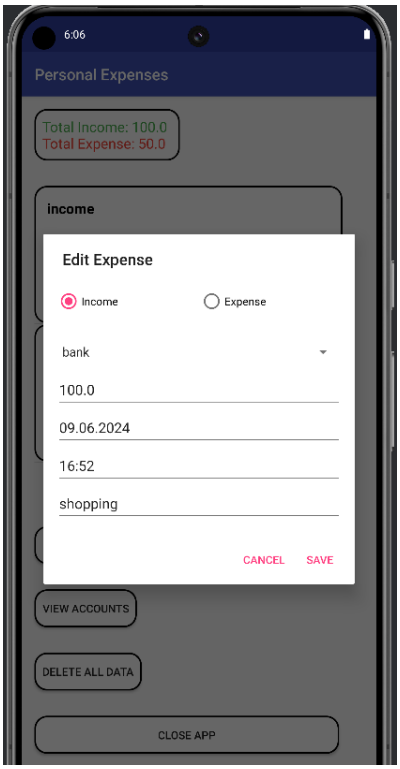

Рис.8.*Вікно з редагуванням витрат/надходжень*

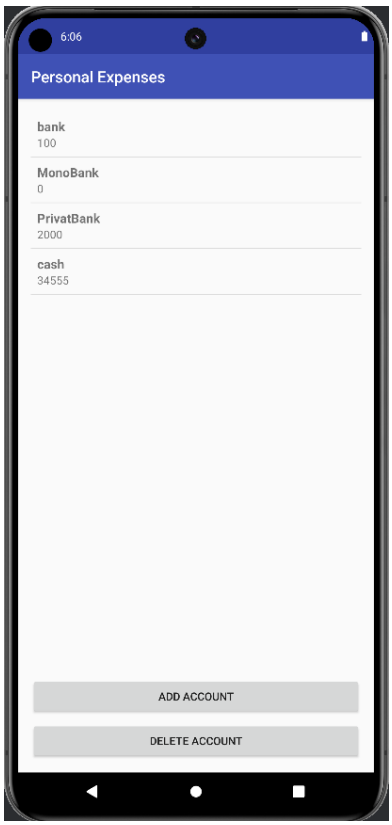

Рис.9.*Екран зі списком всіх рахунків*

3.5 Реалізація функціональності застосунку

Реалізація функціональності застосунку для управління особистими фінансами включає декілька ключових компонентів, таких як створення та управління базою даних SQLite, реалізація основних CRUD (Create, Read, Update, Delete) операцій, а також забезпечення цілісності даних за допомогою транзакцій. У цьому розділі детально розглядаються основні аспекти реалізації, включаючи структуру бази даних, ініціалізацію бази даних, та приклади використання основних методів.

Проектування бази даних для Android-застосунку з управління особистими фінансами включає створення трьох основних таблиць: accounts, expenses та types. Використання зовнішніх ключів дозволяє забезпечити зв'язки між таблицями та підтримувати цілісність даних.

SQL Запити для створення таблиць:

```
-- Створення таблиці types
CREATE TABLE types (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    type TEXT UNIQUE
);

-- Створення таблиці accounts
CREATE TABLE accounts (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    account_name TEXT UNIQUE,
    balance REAL
);

-- Створення таблиці expenses
CREATE TABLE expenses (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
```

```
    type_id INTEGER,
    account_id INTEGER,
    amount REAL,
    time TEXT,
    description TEXT,
    FOREIGN KEY(type_id) REFERENCES types(id),
    FOREIGN KEY(account_id) REFERENCES accounts(id)
);
```

Лістинг 22. *Створення таблиць*

Для ініціалізації бази даних використовується клас **ExpensesDatabaseHelper**, який успадковує **SQLiteOpenHelper**. У ньому визначаються методи **onCreate()** та **onUpgrade(),** які відповідають за створення таблиць та оновлення структури бази даних відповідно.

onCreate(): Метод **onCreate** викликається при першому створенні бази даних. Він відповідає за створення всіх необхідних таблиць і початкове заповнення деяких з них.

```
override fun onCreate(db: SQLiteDatabase) {
    db.execSQL(
        "CREATE TABLE $TABLE_TYPES (" +
                "$COL_ID INTEGER PRIMARY KEY AUTOINCREMENT,"
+
                 "$COL_TYPE TEXT UNIQUE)"
    )
    db.execSQL(
        "CREATE TABLE $TABLE_ACCOUNTS (" +
                "$COL_ID INTEGER PRIMARY KEY AUTOINCREMENT,"
+
                 "$COL_ACCOUNT_NAME TEXT UNIQUE," +
                 "$COL_BALANCE REAL)"
    )
    db.execSQL(
```

```
        "CREATE TABLE $TABLE_EXPENSES (" +
                "$COL_ID INTEGER PRIMARY KEY AUTOINCREMENT,"
+
                "type_id INTEGER," +
                "account_id INTEGER," +
                "$COL_AMOUNT REAL," +
                "$COL_TIME TEXT," +
                "$COL_DESCRIPTION TEXT," +
                "FOREIGN KEY(type_id) REFERENCES
$TABLE_TYPES($COL_ID)," +
                "FOREIGN KEY(account_id) REFERENCES
$TABLE_ACCOUNTS($COL_ID))"
    )
    val types = arrayOf("income", "expense")
    for (type in types) {
        val contentValues = ContentValues().apply {
            put(COL_TYPE, type)
        }
        db.insert(TABLE_TYPES, null, contentValues)
    }
    val contentValues = ContentValues().apply {
        put(COL_ACCOUNT_NAME, "Default Account")
        put(COL_BALANCE, 0.0)
    }
    db.insert(TABLE_ACCOUNTS, null, contentValues)
}
```

Лістинг 23. *Приклад методу onCreate*

Створення таблиці **types**: Цей SQL-запит створює таблицю **types** з двома колонками: id (PRIMARY KEY) і type (унікальний тип транзакції).

Створення таблиці **accounts**: Цей SQL-запит створює таблицю **accounts** з трьома колонками: id (PRIMARY KEY), account_name (унікальна назва рахунку) і balance (баланс рахунку).

Створення таблиці **expenses**: Цей SQL-запит створює таблицю **expenses** з шістьма колонками: id (PRIMARY KEY), type_id (зовнішній ключ, який посилається на types.id), account_id (зовнішній ключ, який посилається на accounts.id), amount (сума витрати), time (дата і час витрати) і description (опис витрати).

Попереднє заповнення таблиці **types** типовими значеннями: Цей блок коду, що показаний на лістингу 23, вставляє два початкові значення в таблицю types: "income" та "expense".

Попереднє заповнення таблиці **accounts** стандартним рахунком: Цей блок коду, що показаний на лістингу 23, вставляє початковий запис у таблицю accounts з назвою "Default Account" і балансом 0.0.

onUpgrade: Метод **onUpgrade** викликається, коли версія бази даних змінюється. Він видаляє старі таблиці та викликає **onCreate** для створення нових.

```
override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int,
newVersion: Int) {
    db.execSQL("DROP TABLE IF EXISTS $TABLE_EXPENSES")
    db.execSQL("DROP TABLE IF EXISTS $TABLE_TYPES")
    db.execSQL("DROP TABLE IF EXISTS $TABLE_ACCOUNTS")
    onCreate(db)
}
```

Лістинг 24. *Приклад методу onUpgrade*

Видалення існуючих таблиць: Ці SQL-запити, що показані на лістингу 24, видаляють таблиці **expenses**, **types** та **accounts**, якщо вони існують. Це робиться для того, щоб при оновленні бази даних не виникало конфліктів через стару структуру таблиць.

Виклик методу **onCreate**: Виклик методу **onCreate,** що показаний на лістингу 24, для створення нової структури бази даних з новими таблицями.

Для забезпечення основної функціональності застосунку, використовуються CRUD методи, ці методи важливі для забезпечення цілісності даних та коректності операцій.

CRUD Методи, що використовуються в проекті:

Клас **ExpensesDatabaseHelper** містить кілька методів для управління базою даних SQLite, що реалізують CRUD операції: створення (**Create**), читання (**Read**), оновлення (**Update**) та видалення (**Delete**) даних.

Create (Створення) - Метод insertExpense, який показаний на лістингу 25, додає нову витрату або дохід до таблиці expenses.

```
fun insertExpense(type: String, accountId: Long, amount:
Double, time: LocalDateTime, description: String): Boolean
{
    val db = this.writableDatabase
    var transactionSuccessful = false
    db.beginTransaction()
    try {
        // Знаходження type_id для заданого типу
        val typeCursor = db.query(TABLE_TYPES,
arrayOf(COL_ID), "$COL_TYPE = ?", arrayOf(type), null,
null, null)
        if (!typeCursor.moveToFirst()) {
            typeCursor.close()
            throw IllegalArgumentException("Type not
found")
        }
        val typeId =
typeCursor.getInt(typeCursor.getColumnIndexOrThrow(COL_ID))
        typeCursor.close()

        // Знаходження балансу рахунку для заданого
accountId
        val accountCursor = db.query(TABLE_ACCOUNTS,
arrayOf(COL_ID, COL_BALANCE), "$COL_ID = ?",
arrayOf(accountId.toString()), null, null, null)
        if (!accountCursor.moveToFirst()) {
```

```
            accountCursor.close()
            throw IllegalArgumentException("Account not
found")
        }
        var balance =
accountCursor.getDouble(accountCursor.getColumnIndexOrThrow
(COL_BALANCE))
        accountCursor.close()

        if (type == "expense" && balance < amount) {
            throw IllegalArgumentException("Insufficient
balance")\
  val contentValues = ContentValues().apply {
            put("type_id", typeId)
            put("account_id", accountId)
            put(COL_AMOUNT, amount)
            put(COL_TIME, time.format(formatter))
            put(COL_DESCRIPTION, description)
        }
        val result = db.insert(TABLE_EXPENSES, null,
contentValues)
        if (result == -1L) {
            throw IllegalStateException("Failed to insert
expense")
        }

        if (type == "income") {
            balance += amount
        } else {
            balance -= amount
        }
        val accountUpdateValues = ContentValues().apply {
            put(COL_BALANCE, balance)
        }
        val updateResult = db.update(TABLE_ACCOUNTS,
accountUpdateValues, "$COL_ID = ?",
arrayOf(accountId.toString()))
        if (updateResult == -1) {
            throw IllegalStateException("Failed to update
account balance")
        }
```

```
            db.setTransactionSuccessful()
            transactionSuccessful = true
        } finally {
            db.endTransaction()
        }
        return transactionSuccessful
}
```

Лістинг 25. *Метод insertExpense*

Пояснення:

- Пошук type_id: Метод виконує запит до таблиці types для отримання type_id, що відповідає заданому типу ("income" або

- Перевірка балансу рахунку: Виконується запит до таблиці accounts для отримання поточного балансу рахунку. Якщо тип транзакції - "expense" і баланс менше суми витрати, викидається виняток.
- Вставка нового запису: Створюється ContentValues з даними про нову транзакцію, і виконується вставка в таблицю expenses. Якщо вставка не вдалася, викидається виняток.
- Оновлення балансу рахунку: Оновлюється баланс рахунку залежно від типу транзакції: додавання суми для доходу або віднімання для витрати. Потім цей новий баланс записується назад у таблицю accounts.
- Транзакція завершується, і база даних закривається. Метод повертає true, якщо транзакція була успішною.

Read (Читання) - Метод **getAllExpenses** отримує всі записи з таблиці expenses, виконуючи SQL-запит для вибору всіх стовпців. Метод повертає курсор, який використовується для ітерації через результати запиту.

```
fun getAllExpenses(): Cursor {
    val db = this.readableDatabase
    return db.rawQuery(
```

```
        "SELECT e.$COL_ID AS _id, t.$COL_TYPE,
a.$COL_ACCOUNT_NAME, e.$COL_AMOUNT, e.$COL_TIME,
e.$COL_DESCRIPTION " +
                "FROM $TABLE_EXPENSES e " +
                "JOIN $TABLE_TYPES t ON e.type_id =
t.$COL_ID " +
                "JOIN $TABLE_ACCOUNTS a ON e.account_id =
a.$COL_ID " +
                "ORDER BY e.$COL_TIME DESC", null)
}
```

Лістинг 26. *Метод getAllExpenses*

Виконується SQL-запит, що об'єднує таблиці **expenses**, **types** та **accounts**, для отримання всіх записів з таблиці expenses з відповідними типами та назвами рахунків. Повертається **Cursor**, який можна використовувати для отримання даних в застосунку.

Метод **getStatistics** обчислює загальні доходи та витрати, виконуючи SQL-запити для сумування значень у таблиці **expenses** залежно від типу операції.

Метод **getExpenseById** отримує конкретний запис про витрату або дохід за його ідентифікатором. Виконується SQL-запит з умовою на основі переданого ідентифікатора, і результати повертаються у вигляді об'єкта **Expense**.

Update (Оновлення) - Метод **updateExpense**, який позаканий на лістингу 26, оновлює існуючий запис про витрату або дохід.

```
fun updateExpense(id: Long, type: String, accountId: Long,
amount: Double, time: LocalDateTime, description: String):
Boolean {
    val db = this.writableDatabase
    db.beginTransaction()
    return try {
        val oldData = getExpenseData(db, id)
```

```
        val typeId = getTypeId(db, type)
        var balance = getAccountBalance(db, accountId)

        balance = revertOldBalance(balance, oldData)
        balance = updateBalance(balance, amount, type)
        updateAccountBalance(db, accountId, balance)

        db.update(TABLE_EXPENSES, ContentValues().apply {
            put("type_id", typeId)
            put("account_id", accountId)
            put(COL_AMOUNT, amount)
            put(COL_TIME, time.format(formatter))
            put(COL_DESCRIPTION, description)
        }, "$COL_ID=?", arrayOf(id.toString()))

        db.setTransactionSuccessful()
        true
    } finally {
        db.endTransaction()
    }
```

Лістинг 27. *Метод updateExpense*

Отримання існуючих деталей витрати: Виконується запит до таблиці expenses для отримання існуючих даних про витрату: **type_id**, **account_id** та **суми (amount)**. Виконується запит до таблиці **types** для отримання нового **type_id**, відповідного заданому типу транзакції. Виконується запит до таблиці **accounts** для отримання нового **account_id** та поточного балансу для заданого рахунку. Виконується запит до таблиці **types** для отримання старого типу транзакції. Оновлюється баланс рахунку, віднімаючи або додаючи стару суму залежно від типу транзакції. Оновлюється баланс рахунку, додаючи або віднімаючи нову суму залежно від нового типу транзакції. Оновлений баланс

записується назад у таблицю **accounts**. Оновлюється запис у таблиці **expenses** з новими даними.

Delete (Видалення) - Метод **deleteExpense**, який показаний на лістингу 27, видаляє конкретний запис про витрату або дохід за його ідентифікатором. Він включає виконання SQL-запиту з умовою на основі переданого ідентифікатора, видаляючи відповідний запис з таблиці expenses.

```
fun deleteExpense(id: Long): Boolean {
    val db = this.writableDatabase
    db.beginTransaction()
    return try {
        val expenseData = getExpenseData(db, id)
        var balance = getAccountBalance(db,
expenseData.accountId)
        balance = revertOldBalance(balance, expenseData)
        updateAccountBalance(db, expenseData.accountId,
balance)

        if (db.delete(TABLE_EXPENSES, "$COL_ID=?",
arrayOf(id.toString())) == -1) {
            throw IllegalStateException("Failed to delete
expense")
        }

        db.setTransactionSuccessful()
        true
    } finally {
        db.endTransaction()
    }
```

Лістинг 28. *Метод Delete*

Метод **deleteAllExpenses** видаляє всі записи з таблиці expenses, виконуючи SQL-запит для очищення таблиці.

```
fun deleteAllExpenses() {
    val db = this.writableDatabase
    db.delete(TABLE_EXPENSES, null, null)
    db.close()
```

Лістинг 29. *Метод deleteAllExpenses*

Ці методи забезпечують базові операції для управління даними у застосунку, дозволяючи користувачам додавати, читати, оновлювати та видаляти фінансові записи.

Для забезпечення цілісності даних у базі даних використовується механізм транзакцій. Транзакція дозволяє групувати кілька операцій у єдину логічну одиницю роботи, яка виконується атомарно. Це означає, що всі операції у межах транзакції або виконуються успішно, або відкатуються у разі помилки.

У проекті використовується транзакційний механізм для додавання нових витрат або доходів. Основні кроки, що виконуються в межах транзакції:

Ініціалізація бази даних для запису: Метод починає з отримання екземпляра **SQLiteDatabase** у режимі запису.

Початок транзакції: Викликається **db.beginTransaction()**, що ініціалізує транзакцію бази даних. Це забезпечує атомарність операцій.

Пошук ідентифікатора типу (**type_id**) та балансу рахунку: Виконується запит до таблиці **types** для отримання **type_id** на основі значення **type**. Якщо такий тип не знайдено, кидається виключення. Далі виконується запит до таблиці **accounts** для отримання балансу рахунку (**balance**). Якщо рахунок не знайдено, кидається виключення.

Перевірка достатності балансу: Якщо тип операції - витрата **(expense**), перевіряється, чи достатньо коштів на рахунку для здійснення операції. У разі недостатнього балансу кидається виключення.

Підготовка даних для вставки: Створюється об'єкт **ContentValues,** який містить усі необхідні дані для вставки нового запису в таблицю expenses.

Вставка нового запису: Виконується вставка нового запису до таблиці expenses за допомогою методу **db.insert().** Якщо вставка не вдалася, кидається виключення.

Оновлення балансу рахунку: Баланс рахунку оновлюється залежно від типу операції: збільшується для доходу (**income**) або зменшується для витрати (**expense**). Оновлений баланс зберігається у таблиці **accounts** за допомогою методу **db.update().**

Завершення транзакції: Якщо всі операції пройшли успішно, транзакція завершується викликом **db.setTransactionSuccessful()** і **db.endTransaction()**. У разі помилки транзакція відкочується.

Повернення результату: Метод повертає **true**, якщо транзакція пройшла успішно, і **false** в іншому випадку.

Метод **insertExpense** демонструє використання транзакцій для забезпечення цілісності даних при додаванні нових витрат або доходів:

Ініціалізація бази даних для запису: Метод починається з отримання екземпляра **SQLiteDatabase** у режимі запису, щоб мати можливість виконувати операції вставки та оновлення.

Початок транзакції: Ініціюється транзакція викликом методу **db.beginTransaction().** Це означає, що всі подальші операції в межах цієї транзакції будуть виконані як єдине ціле.

Пошук ідентифікатора типу (**type_id**) та балансу рахунку: Виконується запит до таблиці **types**, щоб знайти ідентифікатор типу операції (**type_id**). Якщо тип не знайдено, кидається виключення. Далі виконується запит до таблиці **accounts**, щоб знайти баланс рахунку (**balance**). Якщо рахунок не знайдено, кидається виключення.

Перевірка достатності балансу: Якщо тип операції - витрата (**expense**), перевіряється, чи достатньо коштів на рахунку для здійснення операції. Якщо баланс недостатній, кидається виключення.

Підготовка даних для вставки: Створюється об'єкт **ContentValues**, який містить усі необхідні дані для вставки нового запису в таблицю expenses.

Вставка нового запису: Виконується вставка нового запису до таблиці expenses за допомогою методу **db.insert()**. Якщо вставка не вдалася, кидається виключення.

Оновлення балансу рахунку: Баланс рахунку оновлюється залежно від типу операції: збільшується для доходу (**income**) або зменшується для витрати (**expense**). Оновлений баланс зберігається у таблиці **accounts** за допомогою методу **db.update().**

Завершення транзакції: Якщо всі операції пройшли успішно, транзакція завершується викликом **db.setTransactionSuccessful()**, після чого викликається **db.endTransaction()**. Це гарантує, що всі зміни будуть збережені. У разі помилки транзакція відкочується і всі зміни анулюються.

Повернення результату: Метод повертає **true**, якщо транзакція пройшла успішно, і **false** в іншому випадку.

MainActivity є головним екраном застосунку для управління особистими фінансами. Вона відповідає за відображення загального балансу, останніх транзакцій, а також надає можливість додавання нових витрат або доходів. Нижче надається детальний опис основних компонентів та методів MainActivity.

- ExpensesDatabaseHelper: Клас для управління базою даних SQLite. Він забезпечує доступ до даних і виконує основні CRUD-операції.
- TextView для відображення балансу: Текстове поле для відображення загального балансу.
- ListView для історії транзакцій: Список для відображення останніх транзакцій.

  для додавання нових транзакцій: Кнопка для додавання нових витрат або доходів.

- Button для видалення всіх транзакцій: Кнопка для видалення всіх витрат.
- Button для закриття застосунку: Кнопка для завершення роботи застосунку.
- SimpleCursorAdapter: Адаптер для зв'язку між базою даних та

- Змінна для збереження вибраного запису: Використовується для збереження ідентифікатора вибраної транзакції.

Опис основних методів:

onCreate() - Метод onCreate викликається при створенні активності. Він ініціалізує основні компоненти інтерфейсу, створює об'єкт ExpensesDatabaseHelper для доступу до бази даних, встановлює обробники подій для кнопок, і викликає методи setupHistoryListView() та updateStatistics().

setupHistoryListView() - Метод setupHistoryListView налаштовує ListView для відображення історії транзакцій. Він визначає зв'язок між стовпцями бази даних і елементами інтерфейсу, використовує SimpleCursorAdapter для підключення даних з бази до ListView, і встановлює адаптер для ListView.

updateHistory() - Метод updateHistory оновлює список транзакцій після додавання, редагування або видалення запису.

updateStatistics() - Метод updateStatistics отримує загальний дохід і витрати за допомогою методу getStatistics з ExpensesDatabaseHelper. Він обчислює загальний баланс та відображає його у statisticsTextView.

showAddExpenseDialog() - Метод showAddExpenseDialog відображає діалогове вікно для додавання нової витрати або доходу. Користувач може ввести суму, дату, час, опис, тип операції та вибрати рахунок. Після підтвердження введені дані додаються до бази даних, і оновлюється баланс та список транзакцій.

showEditDeleteDialog() - Метод showEditDeleteDialog відображає діалогове вікно з опціями редагування або видалення вибраної транзакції.

showEditExpenseDialog() - Метод showEditExpenseDialog відображає діалогове вікно для редагування вибраної витрати або доходу. Користувач може змінити суму, дату, час, опис, тип операції та вибрати рахунок. Після підтвердження зміни зберігаються у базі даних, і оновлюються баланс та список транзакцій.

showDeleteAllConfirmationDialog() - відображає діалогове вікно підтвердження видалення всіх транзакцій. Користувач може підтвердити або відмінити дію.

Цей детальний опис допомагає зрозуміти, як MainActivity функціонує в рамках застосунку, і які методи та компоненти використовуються для реалізації основної функціональності.

3.6 Аналіз ефективності та зручності використаних засобів забезпечення персистентності даних застосунку

Розглянемо ефективність та зручність використаних засобів організації персистентності.

Ефективність:

1. Швидкість доступу до даних - SQLite є вбудованою базою даних, яка забезпечує високу швидкість доступу до даних. Всі дані зберігаються локально на пристрої користувача, що дозволяє миттєво читати і записувати інформацію без необхідності звертатися до віддаленого сервера. Це особливо важливо для мобільних застосунків, де швидка взаємодія з базою даних безпосередньо впливає на користувацький досвід. Наприклад, у проекті управління особистими фінансами це означає, що користувачі можуть швидко додавати нові витрати або доходи, оновлювати баланс рахунку і переглядати історію транзакцій без затримок.

2. Підтримка транзакцій - SQLite підтримує транзакції, що дозволяє групувати кілька операцій в єдину логічну одиницю роботи. Це забезпечує атомарність операцій – або всі операції виконуються успішно, або жодна з них не виконується. У проекті це забезпечує цілісність даних під час додавання, редагування або видалення транзакцій, знижуючи ризик некоректного запису даних. Наприклад, під час додавання нової витрати у методі insertExpense використовується транзакція, що дозволяє гарантувати, що всі операції з оновлення балансу рахунку і вставки нової витрати будуть виконані атомарно.

3. Мінімальні накладні витрати - SQLite не потребує окремого серверного програмного забезпечення або налаштування, що знижує накладні витрати на адміністрування. Це дозволяє зосередити зусилля на розробці функціональності застосунку, а не на налаштуванні та підтримці бази даних. У даному проекті це означає, що розробники можуть легко інтегрувати SQLite у застосунок і почати працювати з базою даних без додаткових кроків по налаштуванню серверної інфраструктури.

Зручність:

1. Простота використання - SQLite є легким у використанні і має простий синтаксис SQL, що робить його доступним для розробників з різним рівнем підготовки. У проекті застосовуються прості SQL-запити для створення таблиць, вставки, оновлення та видалення даних, що спрощує розробку та підтримку коду. Наприклад, для створення таблиць використовується метод onCreate у класі ExpensesDatabaseHelper, де визначені SQL-запити для створення таблиць types, accounts і expenses.

2. Вбудована підтримка в Android – Android надає вбудовану підтримку для роботи з SQLite через клас SQLiteOpenHelper, що спрощує інтеграцію бази даних в застосунок. У проекті ExpensesDatabaseHelper успадковує цей клас, забезпечуючи зручний спосіб створення та оновлення структури бази даних. Це дозволяє розробникам легко керувати життєвим циклом бази даних і автоматично виконувати оновлення структури при зміні версії бази даних.

3. Робота без інтернет-з'єднання - Оскільки SQLite є локальною базою даних, застосунок може працювати без постійного інтернет-з'єднання. Це важлива перевага для користувачів, які можуть продовжувати вести облік своїх фінансів навіть в умовах обмеженого доступу до мережі. У проекті управління особистими фінансами це дозволяє користувачам вводити нові витрати та доходи, переглядати історію транзакцій і оновлювати баланс рахунків у будь-який час і в будь-якому місці.

Недоліки:

1. Обмежена масштабованість - SQLite підходить для невеликих та середніх обсягів даних, але не є оптимальним рішенням для великих даних або високонавантажених систем. Якщо застосунок розростатиметься і кількість даних значно збільшиться, може виникнути потреба в міграції до більш масштабованої системи керування базами даних, такої як PostgreSQL або MySQL.

2. Відсутність багатокористувацької підтримки - SQLite не підтримує багатокористувацький доступ на рівні сервера, що може бути обмеженням для деяких застосунків. У випадку управління особистими фінансами це не є критичним, оскільки дані зберігаються локально для кожного користувача. Однак, якщо в майбутньому буде необхідність у забезпеченні спільного доступу до даних або синхронізації між різними пристроями, може знадобитися використання серверної бази даних з відповідною підтримкою.

Механізм транзакцій забезпечує цілісність даних у базі даних. Наприклад, у методі insertExpense транзакція використовується для того, щоб всі операції по додаванню витрати або доходу, а також оновленню балансу рахунку виконувалися як єдине ціле. У випадку виникнення помилки, всі зміни, зроблені в межах транзакції, відкотяться, запобігаючи частковому оновленню даних у базі.

Використання SQLite для забезпечення персистентності даних у проекті управління особистими фінансами виявилося ефективним та зручним

рішенням. Швидкий доступ до даних, підтримка транзакцій та простота інтеграції з Android роблять SQLite оптимальним вибором для даного типу застосунків. Незважаючи на деякі обмеження щодо масштабованості та багатокористувацької підтримки, переваги використання SQLite значно переважають недоліки, особливо у контексті мобільного застосунку, орієнтованого на індивідуальне використання.

ВИСНОВКИ

процесі виконання кваліфікаційної роботи були вивчені джерела, присвячені дослідженням у сфері персистентності даних у мобільних застосунках для платформи Android.

ід час розробки застосунку було використано SQLite як основний механізм для забезпечення персистентності даних. Це рішення було обрано завдяки його легкості інтеграції, високій швидкості доступу до даних та підтримці транзакцій.

ід час проектування архітектури застосунку була використана модель

підтримки застосунку, покращує його масштабованість та зручність у використанні.

еалізація функціональності включала розробку основних модулів для взаємодії з базою даних, обробки транзакцій та збереження даних про витрати, доходи та рахунки користувача. Були використані сучасні методи програмування та дотримані кращі практики для забезпечення ефективності та надійності системи.

результаті роботи було розроблено мобільний застосунок для управління особистими фінансами. Застосунок надає користувачам можливість відстежувати свої витрати та доходи, керувати рахунками та отримувати статистику про фінансовий стан.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Bill Phillips, Kristin Marsicano, Chris Stewart Android Programming: The Big Nerd Ranch Guide, 2019. 576 с.
2. Save data using SQLite. Android Developers Site. URL: https://developer.android.com/training/data-storage/sqlite (Дата звернення 10.04.2024).
3. Офіційний сайт розробників Android. URL: https://developer.android.com (дата звернення 10.04.2024).
4. Mark Spikerman Firebase Essentials - Android Edition, 2016. 164с.
5. Preferences DataStore and Proto DataStore. URL: https://developer.android.com/topic/libraries/architecture/datastore (дата звернення 10.04.2024).
6. Kreibich J.A. Using SQLite, O'Reilly Media, Inc., 2010. 503с
7. Про СКБД. SQLite. URL: https://uk.wikipedia.org/wiki/SQLite (Дата звернення 10.04.2024).
8. Save data in a local database using Room. URL: https://developer.android.com/training/data-storage/room (Дата звернення 10.04.2024).
9. Aigner S., Elizarov R., Isakova S., Jemerov D., Kotlin in Action, 2nd Edition, 2024. 564с.
10. Android studio official site. URL: https://developer.android.com/studio (Дата звернення 10.04.2024).
11. DB Browser for SQLite. URL: https://sqlitebrowser.org (Дата звернення 10.04.2024).
12. Gradle Documentation. URL: https://docs.gradle.org/current/userguide/userguide.html (Дата звернення 10.04.2024).

13. Ринок мобільних застосунків для управління фінансами зростає з кожним роком | Grand View Research. URL: https://www.grandviewresearch.com/industry-analysis/mobile-application-market (Дата звернення 10.04.2024).
14. Guide to app architecture. URL: https://developer.android.com/topic/architecture (Дата звернення 10.04.2024).
15. Смирнов Є.Т., Коломоєць Г.П. Дослідження технологій персистентності даних Android-застосунків. Збірник наукових праць студентів, аспірантів, докторантів і молодих вчених «Молода наука-2024» / Т.5. Запорізький національний університет. – Запоріжжя : ЗНУ, 2024. с. 200.

Декларація
академічної доброчесності
здобувача ступеня вищої освіти ЗНУ

Я, Смирнов Єгор Теімуразович, студент 4 курсу, форми навчання денної, Інженерного навчально-наукового інституту, спеціальність 121 Інженерія програмного забезпечення, адреса електронної пошти ipz20bd-203@stu.zsea.edu.ua, — підтверджую, що написана мною кваліфікаційна робота на тему «**Дослідження технологій персистентності даних Android-застосунків**» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

згоден/згодна на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 17.06.2024 _____ Смирнов Єгор Теімуразович
(студент)

Дата 18.06.2024 _____ Коломоєць Геннадій Павлович
(науковий керівник)