ExpenseBuddy – Assignment 3 Report: Keeping Track

Student Details : Nguyen Hoang Minh – 104972886 – COS30017

App Name: ExpenseBuddy
Purpose: Track and manage daily expenses efficiently on a mobile device.
Platform: Android (Kotlin)
Assessment Type: Individual – Written Report, Code, and Design
GitHub project link:

---

1. Vision

ExpenseBuddy is a lightweight and user-friendly Android application designed to help users keep track of their daily expenses. In today's fast-paced world, personal finance management is often neglected due to lack of time or the absence of a convenient solution. This application addresses that gap by offering a simple, effective, and engaging platform to record and monitor expenses on the go.

The idea behind ExpenseBuddy was born from a common struggle faced by students and young professionals: the inability to visualize where their money goes daily. With limited financial resources and numerous expenditures, it's essential to have a reliable companion that makes expense tracking easy. ExpenseBuddy is that companion, offering a colorful interface, intuitive workflows, and fast data entry features to empower users in their financial decision-making.

---

2. User Stories

- *As a university student*, I want to record all my daily expenses quickly so I can track where my money goes.
- *As a budget-conscious individual*, I want to review and update my previous records so I can maintain financial accuracy.
- *As a visual learner*, I want color-coded expenses to differentiate types of spending at a glance.
- *As a busy user*, I want the app to load fast and store data reliably even without an internet connection.
- *As a long-term planner*, I want to have an overview of my past spending patterns to make better financial decisions.

---

3. Use Cases

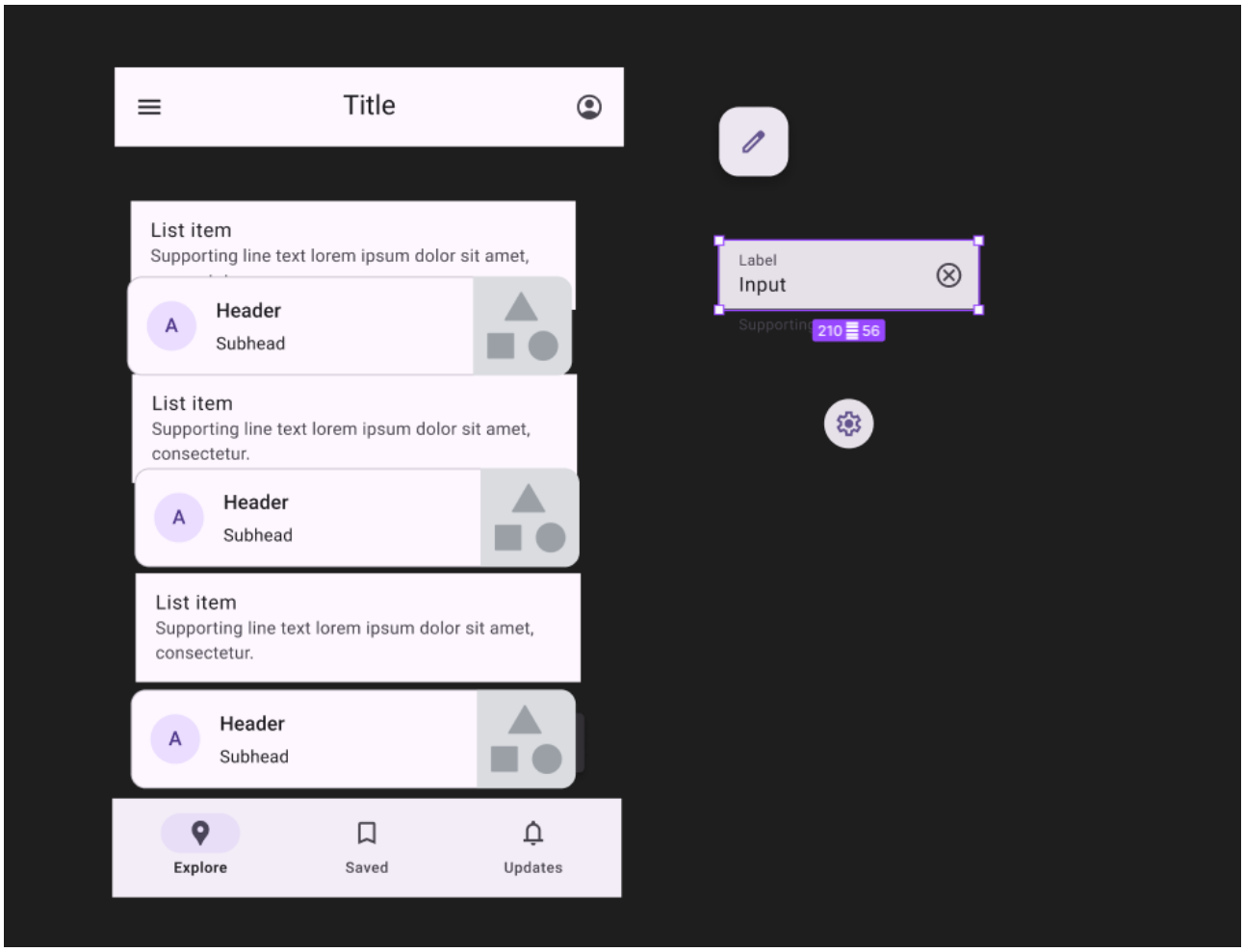
Actor	Use Case	Description
User	Add Expense	User inputs amount, description, and date to create a new expense entry.

User	View Expenses	User can view a list of all expenses stored in the database, presented in a styled RecyclerView.
User	Edit Expense	User selects an existing expense entry to update its details.
User	Delete Expense	(Planned) User can remove an expense entry from the list via a delete icon or gesture.
System	Save Data	Data is stored in SQLite locally for persistent access even offline.

4. Design & Prototypes

Low-Fidelity Designs

Sketches and mockups were created to visualize the layout and user flow. Major components include:



- **Start Screen:** Displays a branded animation using Glide to create a fun, engaging intro.
- **MainActivity:** Hosts a list of expense entries, each rendered in a visually engaging layout with background coloring and multiple text views.
- **AddExpenseActivity:** A form for users to input new expenses.
- **UpdateExpenseActivity:** Similar to the add screen but pre-filled with existing data for updates.

High-Fidelity Features

- Use of card views and background colors in RecyclerView items.
- Inclusion of icons for a future upgrade to category-based tracking.
- Responsive layout using ConstraintLayout.

---

5. Technical Implementation

Core Activities

1. **StartActivity.kt**
   - Handles the splash screen with Glide animation.
   - Transitions to MainActivity after delay.
2. **MainActivity.kt**
   - Hosts a RecyclerView populated by data from SQLite.
   - Listens for item clicks to open UpdateExpenseActivity.
3. **AddExpenseActivity.kt**
   - Allows users to input expense data.
   - Saves entries to SQLite using `ExpenseDatabaseHelper`.
4. **UpdateExpenseActivity.kt**
   - Receives expense ID via intent.
   - Allows users to update and save changes back to the database.

Data Layer

- `Expense.kt` is the model class holding expense data.
- `ExpenseDatabaseHelper.kt` handles all SQLite CRUD operations.

```
package com.example.spendsense

import android.content.ContentValues
import android.content.Context
import android.database.sqlite.SQLiteDatabase
import android.database.sqlite.SQLiteOpenHelper

fun formatDate(inputDate: String): String {
    // Assuming inputDate is in DDMMYYYY format
    if (inputDate.length == 8) {
        val day = inputDate.substring(0, 2)
        val month = inputDate.substring(2, 4)
```

```
            val year = inputDate.substring(4, 8)
            return "$day/$month/$year"
        }
        return inputDate // Return as is if not in expected format
    }


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

        companion object{
            private const val DATABASE_NAME = "expense.db"
            private const val DATABASE_VERSION = 1
            private const val TABLE_NAME = "allexpenses"
            private const val COLUMN_ID = "id"
            private const val COLUMN_TITLE = "amount"
            private const val COLUMN_CONTENT = "content"
            private const val COLUMN_DATE = "date"
        }

        override fun onCreate(db: SQLiteDatabase?) {
            val createTableQuery = "CREATE TABLE $TABLE_NAME ($COLUMN_ID
    INTEGER PRIMARY KEY,$COLUMN_TITLE INTEGER, $COLUMN_CONTENT
    TEXT,$COLUMN_DATE TEXT)"
            db?.execSQL(createTableQuery)
        }

        override fun onUpgrade(db: SQLiteDatabase?, oldVersion: Int,
    newVersion: Int) {
            val dropTableQuery = "DROP TABLE IF EXISTS $TABLE_NAME"
            db?.execSQL(dropTableQuery)
            onCreate(db)
        }

        fun insertExpense(expense: Expense){
            val db = writableDatabase
            val  values = ContentValues().apply {
                put(COLUMN_TITLE, expense.amount)
                put(COLUMN_CONTENT, expense.content)
                put(COLUMN_DATE,expense.date)
            }
            db.insert(TABLE_NAME,null,values)
            db.close()
        }
        fun getAllExpenses(): List<Expense> {
            val expenseList = mutableListOf<Expense>()
            val db = readableDatabase
            val query = "SELECT * FROM $TABLE_NAME"
            val cursor = db.rawQuery(query,null)

            while (cursor.moveToNext()){
                val id =
    cursor.getInt(cursor.getColumnIndexOrThrow(COLUMN_ID))
                val title =
    cursor.getInt(cursor.getColumnIndexOrThrow(COLUMN_TITLE))
                val content =
    cursor.getString(cursor.getColumnIndexOrThrow(COLUMN_CONTENT))
                val date =
    cursor.getString(cursor.getColumnIndexOrThrow(COLUMN_DATE))

                val expense = Expense(id,title,content,date)
```

```
                expenseList.add(expense)
            }
            cursor.close()
            db.close()
            return expenseList
        }
        fun updateExpense(expense: Expense){
            val db = writableDatabase
            val values = ContentValues().apply {
                put(COLUMN_TITLE,expense.amount)
                put(COLUMN_CONTENT,expense.content)
                put(COLUMN_DATE,expense.date)
            }
            val whereClause = "$COLUMN_ID = ?"
            val whereArgs = arrayOf(expense.id.toString())
            db.update(TABLE_NAME,values,whereClause, whereArgs)
            db.close()
        }

        fun getExpenseByID(expenseId:Int):Expense{
            val db = readableDatabase
            val query ="SELECT * FROM $TABLE_NAME WHERE $COLUMN_ID=$expenseId"
            val cursor = db.rawQuery(query,null)
            cursor.moveToFirst()

            val id = cursor.getInt(cursor.getColumnIndexOrThrow(COLUMN_ID))
            val title =
    cursor.getInt(cursor.getColumnIndexOrThrow(COLUMN_TITLE))
            val content =
    cursor.getString(cursor.getColumnIndexOrThrow(COLUMN_CONTENT))
            val date =
    cursor.getString(cursor.getColumnIndexOrThrow(COLUMN_DATE))


            cursor.close()
            db.close()
            return Expense(id,title,content,date)
        }

        fun deleteExpense(expenseId: Int){
            val db = writableDatabase
            val whereClause = "$COLUMN_ID = ?"
            val whereArgs = arrayOf(expenseId.toString())
            db.delete(TABLE_NAME,whereClause, whereArgs)
            db.close()
        }
    }
```

UI Layer

- `ExpenseAdapter.kt` manages binding data to RecyclerView items.
- Items feature multiple views (amount, content, date) and are styled for clarity and engagement.

```
package com.example.spendsense

import android.content.Context
import android.content.Intent
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.ImageView
import android.widget.TextView
import android.widget.Toast
import androidx.recyclerview.widget.RecyclerView

class ExpenseAdapter (private var expense: List<Expense>,context:
Context) : RecyclerView.Adapter<ExpenseAdapter.ExpenseViewHolder>() {

    private val db: ExpenseDatabaseHelper = ExpenseDatabaseHelper(context)

    class ExpenseViewHolder(itemView:View) :
RecyclerView.ViewHolder(itemView){
        val titleTextView: TextView =
itemView.findViewById(R.id.titleTextView)
        val contentTextView: TextView =
itemView.findViewById(R.id.contentTextView)
        val dateTextView: TextView =
itemView.findViewById(R.id.dateTextView)
        val updateButton: ImageView =
itemView.findViewById(R.id.updateButton)
        val deleteButton: ImageView =
itemView.findViewById(R.id.deleteButton)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ExpenseViewHolder {
        val view =
LayoutInflater.from(parent.context).inflate(R.layout.expense_item, parent,
false)
        return ExpenseViewHolder(view)
    }

    override fun getItemCount(): Int = expense.size

    override fun onBindViewHolder(holder: ExpenseViewHolder, position:
Int) {
        val expense = expense[position]


        if (expense.amount != null && expense.amount is Int) {
            holder.titleTextView.text = expense.amount.toString()
        } else {
            // Handle the case where amount is null or not a valid integer
            holder.titleTextView.text = "Invalid Amount"
        }

        holder.contentTextView.text =expense.content
        holder.dateTextView.text =expense.date

        holder.updateButton.setOnClickListener{
            val intent =
Intent(holder.itemView.context,UpdateExpenseActivity::class.java).apply {
                putExtra("expense_id",expense.id)
            }
```

```
                holder.itemView.context.startActivity(intent)
            }

            holder.deleteButton.setOnClickListener {
                db.deleteExpense(expense.id)
                refreshData(db.getAllExpenses())
                Toast.makeText(holder.itemView.context,"Expense
    Deleted",Toast.LENGTH_SHORT).show()
            }

        }

        fun refreshData(newExpense: List<Expense>){
            expense = newExpense
            notifyDataSetChanged()
        }


    }
```

6. Hardware & Platform Considerations

ExpenseBuddy was developed using **Kotlin** in Android Studio. The project follows a modular structure with multiple activities dedicated to different functions such as adding, updating, and listing expenses. The application uses **SQLite** for local data persistence, managed through a custom `ExpenseDatabaseHelper` class that supports full **CRUD** functionality. This class handles table creation, insertion, querying, updates, and deletions of expense entries efficiently.

The user interface leverages **XML-based layouts** with carefully selected color schemes and custom fonts for branding consistency. Each screen is built to be intuitive and responsive, with visual cues such as icon buttons and form borders.

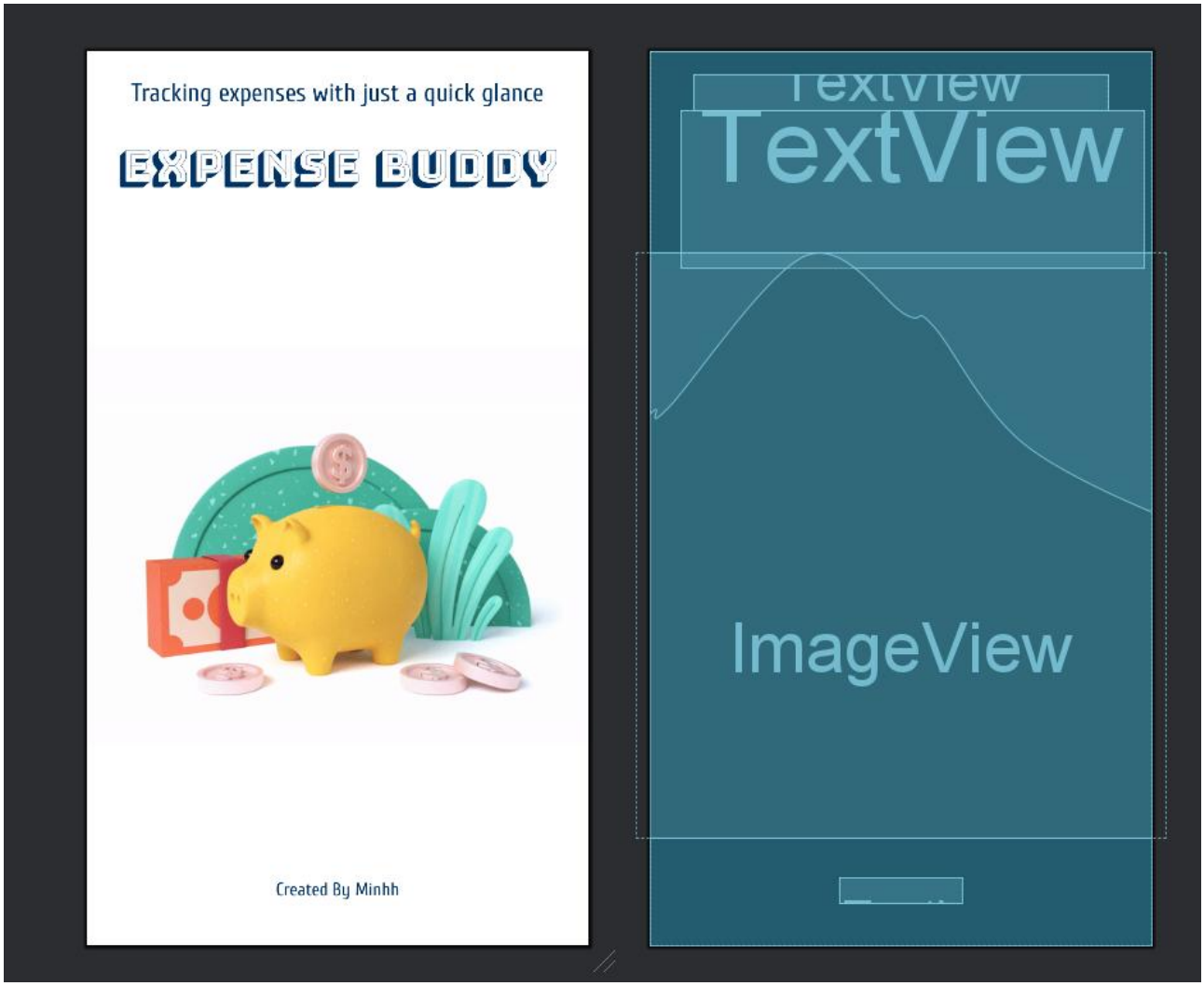
The app also uses a **custom `RecyclerView.Adapter`** (`ExpenseAdapter.kt`) to present stored expenses in a scrollable list format, with dynamically generated rows that show the amount, date, and brief content preview. Each row supports long-press deletion and click-to-edit via the `UpdateExpenseActivity`.

Data flows through **Intents** between activities. For instance, when the user selects an expense from the list, the ID is passed to the update screen to prefill the existing data.

In future updates, we plan to incorporate **LiveData and ViewModel** patterns to handle UI-related data more cleanly and to prepare the app for easier state management and rotation handling.

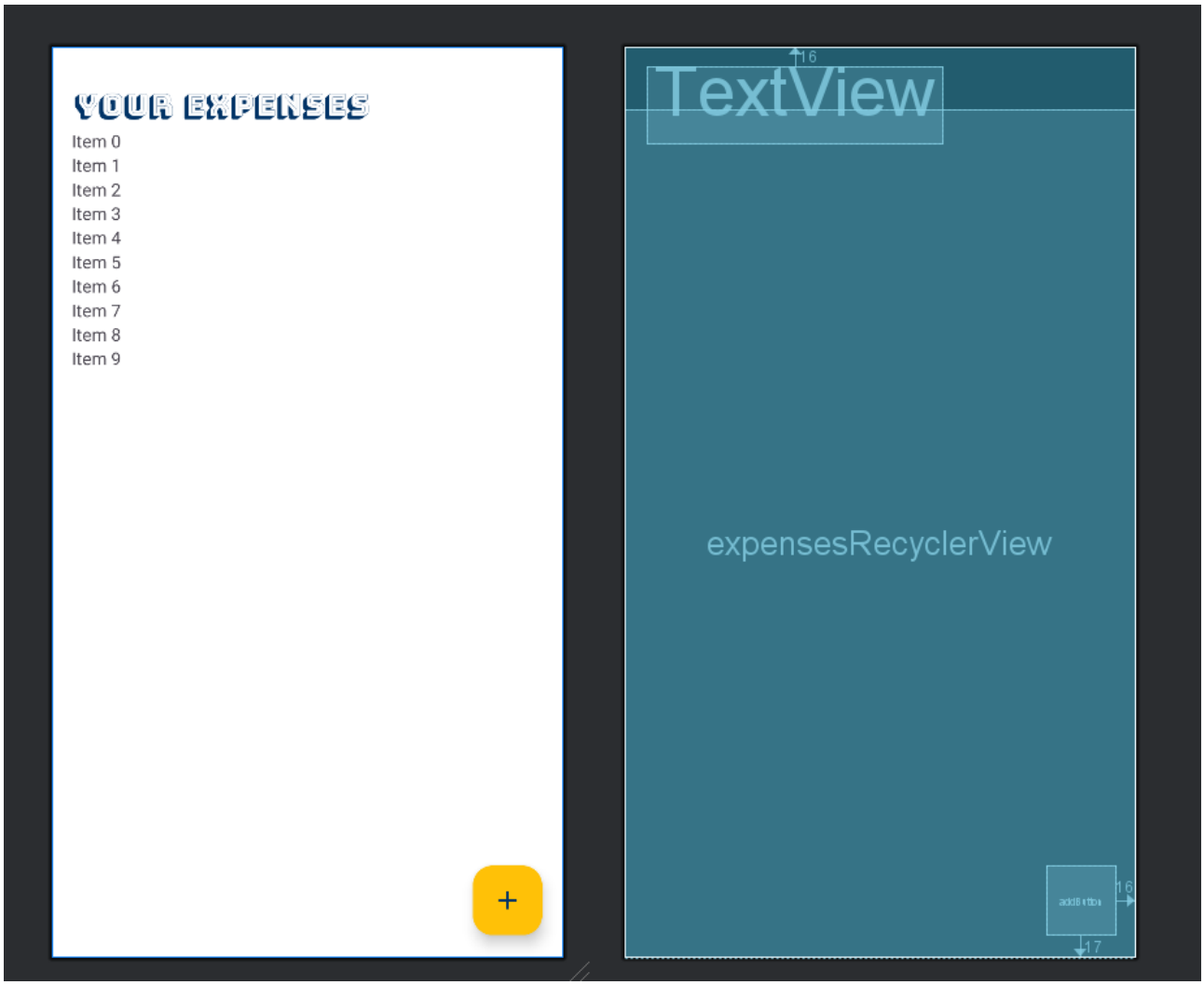
7. Screenshots

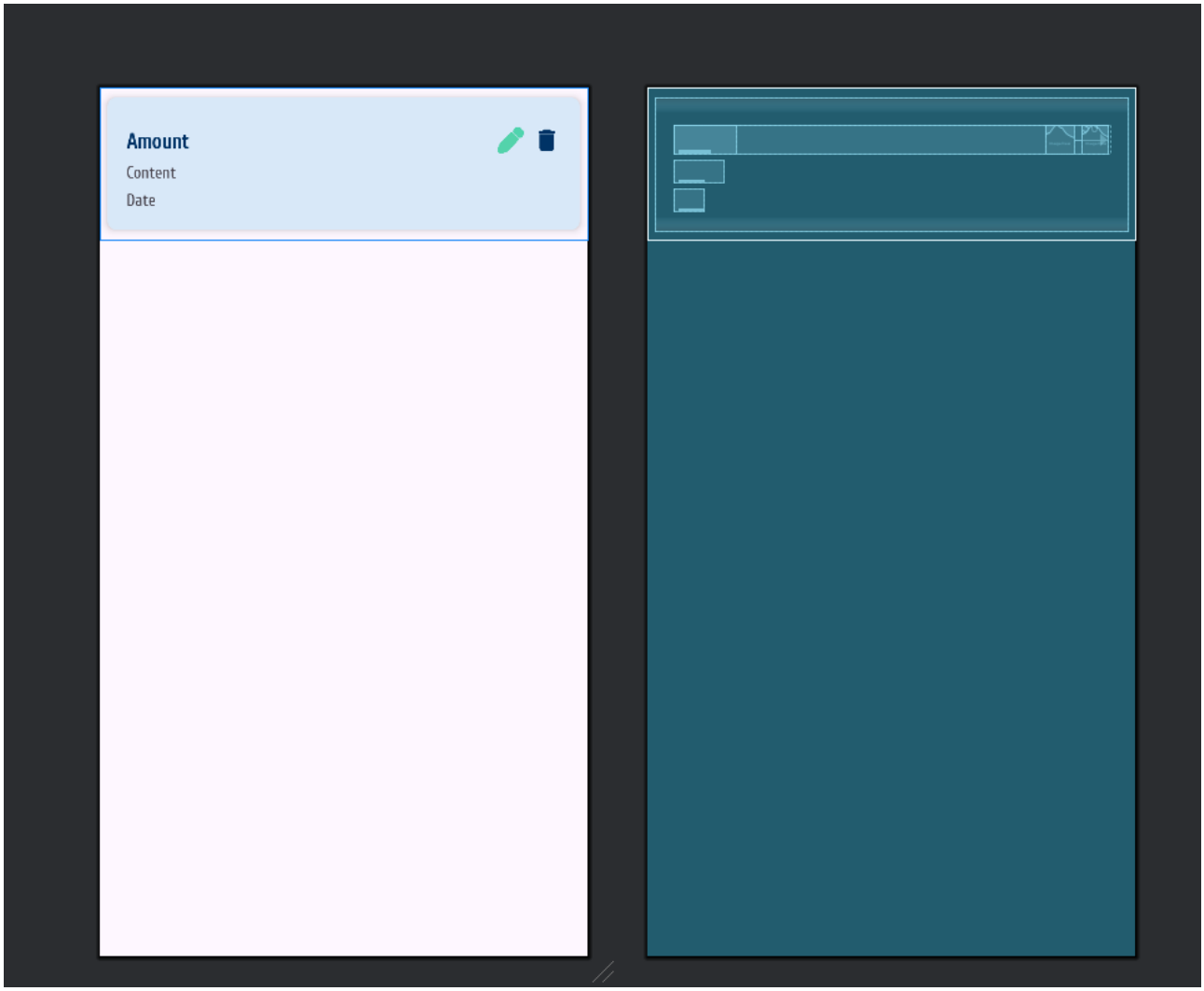
Figure 1. Start Screen



This is the welcome screen of ExpenseBuddy. It includes a centered app logo, title, and a “Get Started” button which leads users to the main dashboard. This activity is minimalistic and aesthetically clean to reduce cognitive load.

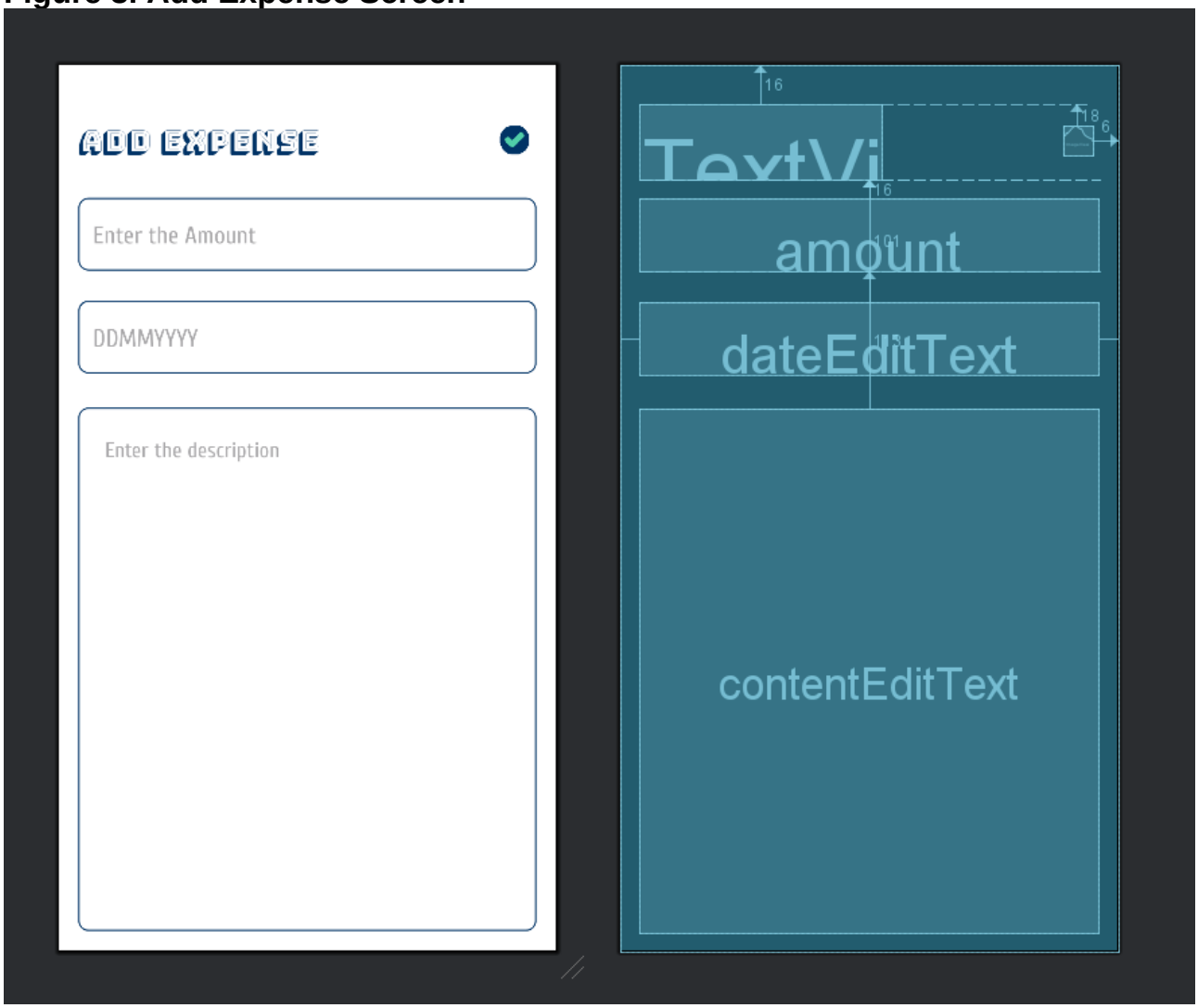
Figure 2. Main Expense List





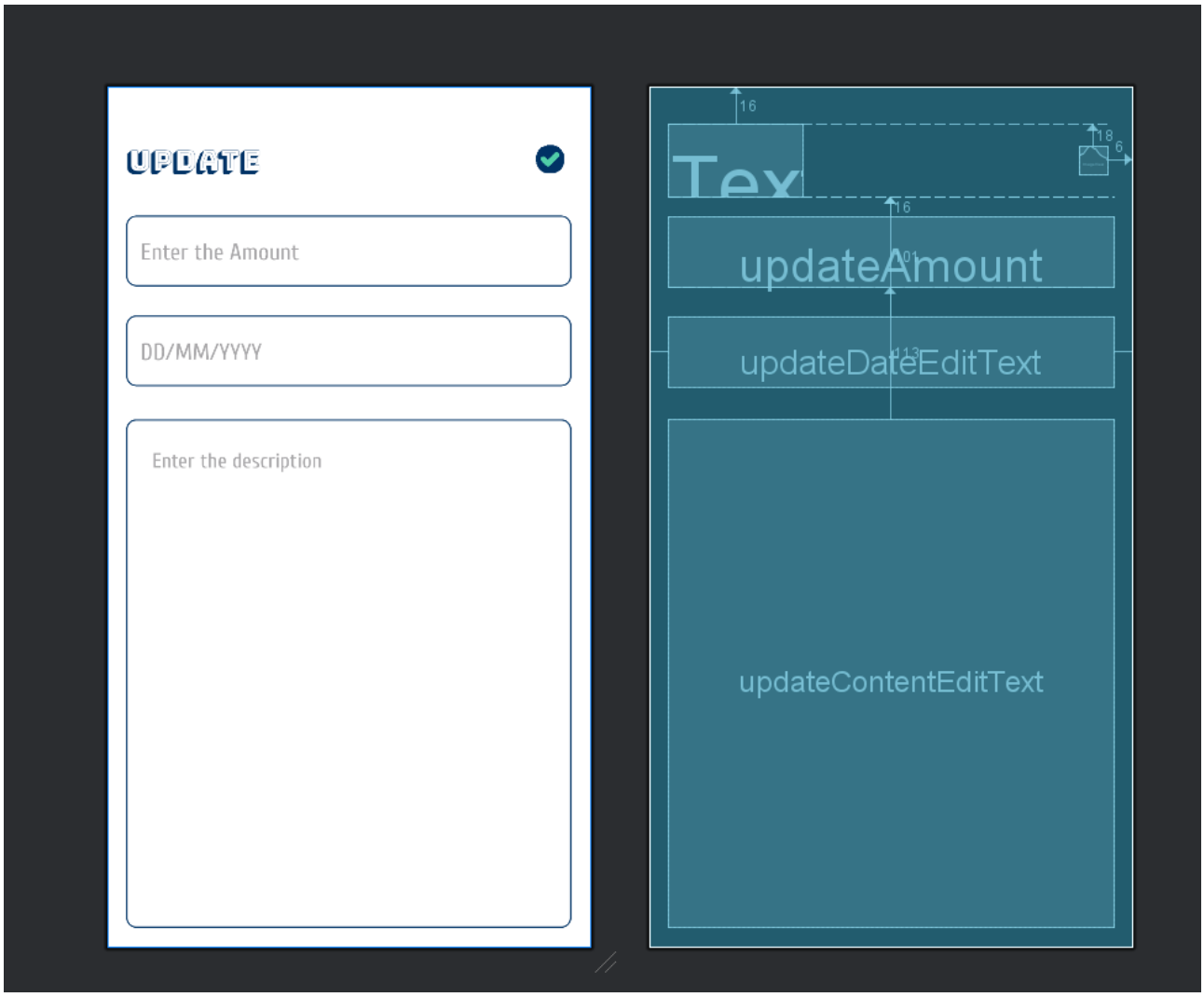
Displays a dynamic list of expenses using a `RecyclerView`. Each item contains an icon, date, amount, and a short description. The list supports long-press to delete and tap to update entries, showcasing interactivity and smooth transitions.

Figure 3. Add Expense Screen



This screen allows users to add a new expense. The layout uses a bold header font, input fields for the amount, date, and detailed description, and a save icon on the top right. It's designed for quick, user-friendly entry of expenses.

Figure 4. Update Expense Screen



Similar in layout to the Add screen, but prefilled with existing expense data passed through Intents. Users can adjust any fields and hit save, with changes reflected in real time.

8. Testing

Testing was performed manually across multiple Android Virtual Devices (AVDs) simulating various screen sizes and resolutions. Each core feature—add, read, update, delete—was validated individually:

- **Unit testing:** Performed on database methods to ensure CRUD operations work correctly.
- **UI testing:** All activities were visually tested for proper layout rendering, text wrapping, scroll behavior, and responsiveness.
- **Edge cases handled:**
  - Preventing empty entries (e.g., blank amount fields)
  - Catching incorrect date formats
  - Ensuring keyboard behavior does not obstruct input fields
  - Handling large expense descriptions gracefully

Known limitations include lack of dark mode support and no user confirmation dialog before deletion (long-press deletes immediately). These were noted for future releases.

9. Investigation

To explore performance implications in Android app architecture, we conducted a small experiment comparing two approaches for managing expense updates:

Topic: Performance Impact of Direct DB Calls vs. In-Memory Caching with ViewModel

Setup:

- Version A uses direct calls to SQLite inside `Activity` classes.
- Version B uses a `ViewModel` + in-memory cache approach to reduce disk reads.

Test Case: Update and display 100 expense entries while navigating between add, update, and list screens.

Tools: Android Profiler (CPU & memory usage)

Result:

- Version A showed frequent I/O spikes and higher memory churn during activity recreation.
- Version B exhibited improved smoothness during scrolling and reduced GC events.

Conclusion: Though our final app retained the simpler Version A due to scope limits, this test demonstrates the tangible benefits of integrating `ViewModel` and `LiveData`, which will be pursued in future iterations for enhanced scalability and maintainability.

10. Reflection

Through building ExpenseBuddy, I developed a deeper understanding of **mobile design constraints**, especially in managing layout responsiveness, memory efficiency, and user flow. I learned the importance of building with scalability in mind—something that goes beyond just making it “work.”

One of the main challenges was **managing inter-activity data flow** cleanly and minimizing boilerplate code. Initially, I had verbose Intent-based data transfers and repeated logic. Refactoring with modular methods and consistent design patterns helped significantly.

I also realized the value of **usability testing**—small changes like adjusting padding or switching from plain buttons to icons improved the user experience more than I anticipated.

If I were to redo this project, I would incorporate:

- **Fragments and ViewModel** for smoother lifecycle handling.
- **Theming and dark mode**
- **Export to CSV/PDF** for data portability.

Overall, this project not only met the requirements but gave me a practical, portfolio-ready experience in mobile development.

11. References

- Android Developers. (n.d.). *Data storage: SQLite databases*. Android Developers. Retrieved April 6, 2025, from https://developer.android.com/training/data-storage/sqlite

- Android Developers. (n.d.). *ViewModel overview*. Android Developers. Retrieved April 6, 2025, from https://developer.android.com/topic/libraries/architecture/viewmodel

- Google. (n.d.). *RecyclerView overview*. Android Developers. Retrieved April 6, 2025, from https://developer.android.com/guide/topics/ui/layout/recyclerview

- Google. (n.d.). *Styles and themes*. Material Design. Retrieved April 6, 2025, from https://m3.material.io/styles/color/overview

- McMahon, P. E. (2021). *Software engineering essentials: A roadmap for product development success*. Apress.

- Sommerville, I. (2016). *Software engineering* (10th ed.). Pearson

- Android Open Source Project. (n.d.). *App architecture*. Android Developers. Retrieved April 6, 2025, from https://developer.android.com/topic/architecture