EXPENSE TRACKER REPORT

APPLICATION PROGRAMMING – MOBILE COMPUTING

Víctor Sebastián Marticorena
287767

INTRODUCTION

Erasmus Expense Tracker is a modern Android app designed to help students and individuals track personal finances abroad. It allows users to:

- Record and categorize incomes and expenses.
- Set monthly budgets per category.
- Monitor financial stats through animated and interactive charts.
- Navigate between views using a custom scaffold with drawer and FAB integration.

This document consolidates the core technical architecture of the app across four main pillars:

1. **Data Persistence Layer**: SQLite via Jetpack Room (see below)
2. **Dashboard Visuals**: 3D-like charts and animated stats
3. **Navigation System**: Jetpack Navigation-Compose architecture
4. **Budget Tracking**: Logic and snackbar alerts for spending thresholds

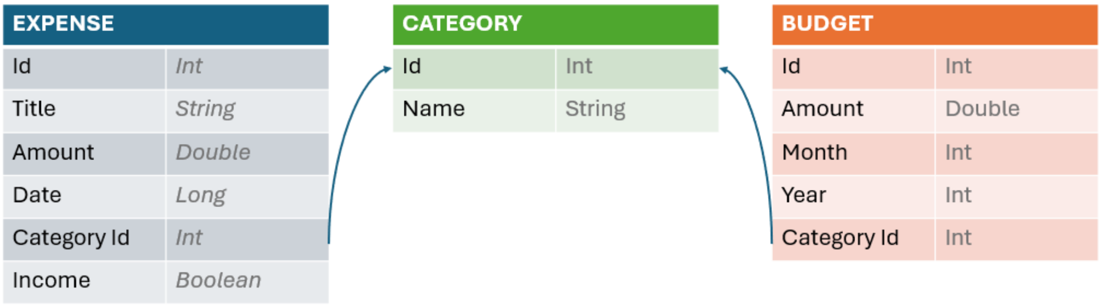
Each layer is structured using best practices in modern Android development (MVVM + Kotlin Flows + **Compose**: UI components, state handling...).

KEY FUNCTIONALITIES

Implementation of a SQLite database using Room library: expenses, categories and budgets

One of the main key features of this application is the integration of a database. The database that this project uses is composed of three main entities:

- **Expense**: represents each transaction. It has an ID, a name, a quantity, a date and an income attribute (true if value is positive, false if negative). Each expense is linked to a category ID.
- **Category:** The most basic entity. It has an ID and a name. It categorizes expenses and budgets.
- **Budgets:** It has an ID, a quantity and it is linked to a category, month and year.



EXPENSE	
Id	*Int*
Title	*String*
Amount	*Double*
Date	*Long*
Category Id	*Int*
Income	*Boolean*

CATEGORY	
Id	Int
Name	String

BUDGET	
Id	Int
Amount	Double
Month	Int
Year	Int
Category Id	Int

The implementation of SQLite through the Jetpack Room ORM consists on the following layers:

Layer	Classes / Files
Entity	*Expense, Category, Budget*
DAO	*ExpenseDao, CategoryDao, BudgetDao*
Database	*AppDatabase* (Room Database*)*
Repository	*ExpenseRepository*
View-Model	*ExpenseViewModel* (uses the repository*)*

Room generates the underlying SQL tables and CRUD statements at the same time as compilation based on the annotated Kotlin code. All data access is exposed as **Kotlin Flow** streams so the UI stays reactive.

1. **Entity Layer**

Each @***Entity*** is an annotated Kotlin data class. Room uses the metadata to build tables, primary keys, indices and foreign-key constraints

Let's show as an example the code of the Expense Entity (*Expense.kt*):

```
@Entity(
    tableName = "expenses",
    foreignKeys = [
        ForeignKey(
            entity = Category::class,
            parentColumns = ["id"],
            childColumns = ["categoryId"],
            onDelete = ForeignKey.CASCADE
        )
    ]
)
data class Expense(
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
    val title: String,
    val amount: Double,
    val date: Long,
    val categoryId: Int,
    val photoPath: String? = null,
    val income: Boolean = false
)
```

As you can see, it has the @Entity annotation and it is linked to the database table called *expenses*. **Foreign key** enforces that an *Expense* cannot exist without its *Category* (cascading delete). The *date* attribute is stored as *Long*, which keeps timezone math in Kotlin (which queries later will convert to text with using strftime).

2. **Dao layer** (*ExpenseDao.kt*...)

DAOs are just pure interfaces, which Room uses auto-creates the concrete classes. The DAOs include the basic CRUD operations (annotated accordingly: @Insert, @Delete...), plus some useful queries that we will use later (annotated as @Query). Some queries that were included:

- ***getAllExpenses*** (for Budget and Category as well): retrieves all expenses

```
SELECT * FROM expenses ORDER BY date DESC
```

- ***getTotalForCategoryInMonth:*** gets the sum of expenses of a given category and month (for keeping track of the budgets)

```
SELECT SUM(amount) FROM expenses
WHERE categoryId = :categoryId
  AND strftime('%m', date / 1000, 'unixepoch') = :monthStr
  AND strftime('%Y', date / 1000, 'unixepoch') = :yearStr
```

- ***getBudgetForCategoryMonthAndYear:*** gets the budget for a given category, month and year (for tracking the budgets)

```
 SELECT * FROM budgets WHERE categoryId = :categoryId AND month = :month  AND
year = :year
```

3. **Database class** (*AppDatabase.kt)*

Takes the previously declared entities and DAOs, and based on this creates the database *expenses_db*. During development, I included the *fallbackToDestructiveMigration()* in the database, so when I apply a migration (such as a change to one of the entities), the entire app does not collapse.

Migration Roadmap

Currently, as you can see in the code, the database's version is set to 4. This is because I performed the following migrations during development:

- ✓ v1 – tables Expense & Category
- ✓ v2 – add Budget table
- ✓ v3 – add photoPath column (Expense)
- ✓ v4 – change Budget.month to 1-based INT

4. **Repository** (*ExpenseRepository.kt*):

Its main functionality is to provides a clean API that isolates ViewModels/UI from Room specifics.

5. **ViewModel & reactive streams** (*ExpenseViewModel.kt*)

The ExpenseViewModel collects the *Flows (*which keep the Compose UI reactive and lifecycle-aware) on the main-safe dispatcher and exposes them as *State* via ***collectAsState***() in Composables. It also performs other operations in order to keep the main code as clean as possible, such as for example *checkBudgetStatus(),* which check the progress of each budget.

In conclusion, this setup provides a clean, testable and maintainable data layer ready for production once migrations replace destructive upgrades.

Generation of **3D graphics and animations** in user's dashboard, representing the expenses statistics using *MPAndroidChart*

For viewing some basic statistics, I included in the user dashboard some graphics to visualize the most important features of the database.

Library stack

Concern	Tech / API	Why it was chosen
Chart rendering	*MPAndroidChart* v3.1.0	Mature, hardware-accelerated OpenGL pipeline; built-in 3-D-style depth for Pie/Bar. Also exposes imperative animation calls (animateX/Y).
Compose bridge	*AndroidView* wrapper	Keeps us in 100 % Compose while embedding the legacy View-based charts.
Data feed	*Kotlin Flow* streams (from *Room*)	Live updates: charts refresh automatically when the underlying Flow emits.
Animations	*MPAndroidChart's* easing functions + small custom *ValueAnimators*	Gives us a 60 FPS entry animation and gentle colour fades on data change.

MPAndroidChart does not render true 3-D, but its OpenGL renderer paints pie slices and bars with an angled perspective (plus ambient shading), creating a somewhat 3-D illusion that works well on mobile.

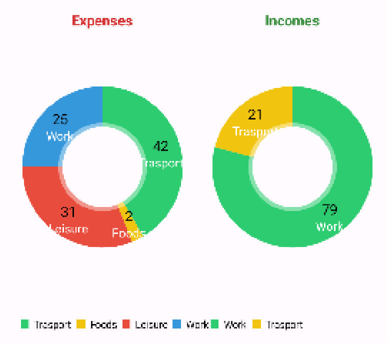
Architecture Diagram

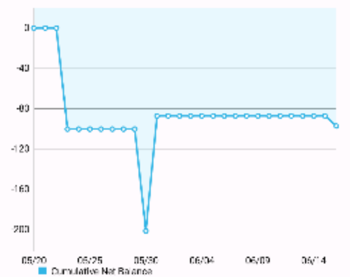
Room (SQLite) → Flow<Expense> / Flow<Category> → ExpenseViewModel (via VM) → →@Composable DashboardScreen (via collectAsState()) → AndroidView<PieChart | LineChart | BarChart> → MPAndroidChart (OpenGL surface)

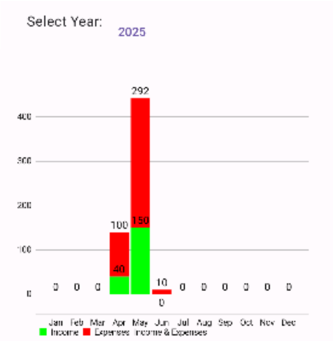
Data changes in the DB instantly propagate to the visible charts with no manual invalidate() calls. Compose recomposes it; the chart's invalidate() is invoked in the update block of each AndroidView wrapper.

The main dashboard includes 3 main graphs:

- **Pie Chart** (*PieChart*.kt): Two different pie charts that display 1.expenses and 2.incomes by category.
    - **Radial gradient + sliceSpace**: subtle depth shadow.
    - **selectionShift** pops a slice forward on touch
    - **animateY** sweeps the pie in from the bottom edge with *EaseInOutQuad* easing for a subtle entry.
- **Line Chart**(*LineChart*.kt): A line graph that displays the net balance over the last 28 days.
    - **Filled area** with alpha plus a darker *lineWidth* creates a "plane" floating above the dashboard card.
    - Horizontal *EaseInCubic* makes the line draw from left to right, mirroring timeline flow.
- **Stacked Bar Chart**(*StackedBarChartView*.kt). A stacked bar chart with a year selector what shows the incomes vs expenses in every month of the selected year
    - *MPAndroidChart* renders bars with a **top bevel + side shadow**, supplied by its OpenGL shader







Navigation using Jetpack Navigation

The application is composed of the following screens:

- **Main Dashboard**: displays the current Net Balance of the user, the three most recent transactions and the graphs explained before
- **Expense List**: displays all the expenses, displayed as green if income and red if expense. At the top it is possible to navigate between months of the year, with two arrows (left/right) used to move between months. Also, some filters were included (a browser for name, an income/expense filter and a category filter). There is a **+** where you tap and you can add a new expense using a form(applicable also to budgets and categories).

- **Expense Form**: You can fill the fields and add a new expense: If you tap the day field, it opens a calendar where you can select a date (using the Date and Locale Handling libraries)
- **Category List**: Displays the list of the existing categories
- **Category Form:** Add a new category.
- **Budget List:** List of the existing budgets. Above each budget, there is a progress bar corresponding to each budget which shows how near you are from exceeding it.
- **Budget Form**: Add a new budget.

In order to navigate between pages, we implemented the following components:

Layers	File	Responsibility
Route Definitions	*Screens.kt* (sealed class*)*	One immutable source for all routes and arguments builders
Navigation Graph	*NavGraph.kt*	Declarative mapping Screen → @Composable using *NavHost*.
Host & Controller	*MainScaffold.kt*	Creates *rememberNavController*(), uses *ModalNavigationDrawer*, *TopAppBar*, FAB (to show the **+** button), and involves NavGraph.
Screens	ui/screens/**	Passive destinations

The Screen involves a destinations for each of the views discussed before, including the edit pages (which are the same as the forms but with a selected ID).

Why use sealed class in Screens.kt: adding a new screen requires only one extra object.

FAB logic: by observing *currentBackStackEntry*, the scaffold toggles its FAB and chooses the correct *Add* destination (when (currentRoute) { … }). No manual Boolean state is required.

Budget implementation

Finally, we implemented the budgets functionalities: when you are near exceeding a budget, show an alert. When you exceed a budget completely, show a different alert.

Users can assign a monthly limit per category (or a global -1 All Categories bucket). They are warned when 90 % or 100 % of any budget is reached. In the ViewModel, it checks the state of the budget. If reaching these limits show a Snackbar (similar to the Toast) advising of this event.

DIFFICULTIES ENCOUNTERED

Building the Erasmus Expense Tracker presented a number of technical and design challenges, especially when it came to synchronizing UI state with the underlying database, managing calendar-based filtering, and ensuring type-safe navigation across dozens of screens.

Roughly **80–85% of the code was written by me**, including the database layer, all core composables, filters, custom chart wrappers, and user input forms. I also implemented the overall architecture using MVVM and Jetpack Compose.

The remaining **15–20% was guided with the help of AI**, especially for more advanced or low-level parts of the app. This includes:

- The snackbar warning logic tied to budget thresholds (with Flow + coroutine channels)
- Complex usage of MPAndroidChart for stacked bar and pie charts
- Certain navigation route patterns and ViewModel factory setup