FinanceApp – Gerenciador Financeiro Pessoal

Aplicativo Android com Persistência Local e Arquitetura MVVM

Thalles Augusto Monteiro Martins

Universidade Federal de Itajubá – Campus Itabira

26/06/2025

Sumário

1 Concepção do Projeto

2 Funcionalidades Implementadas

3 Conclusão

Concepção do Projeto

Objetivo do Trabalho

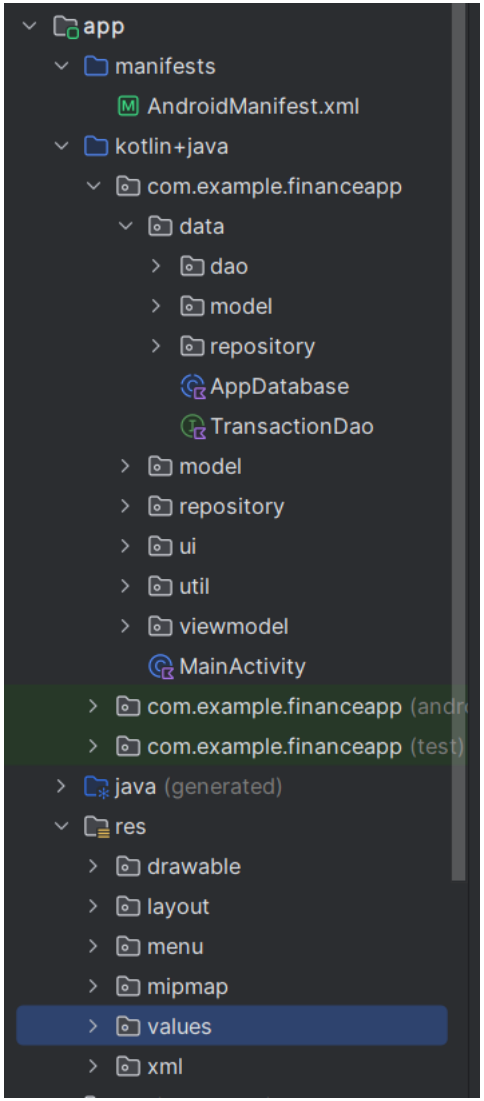
Desenvolver um aplicativo Android de tema livre contendo pelo menos **cinco funcionalidades distintas**, persistência de dados local e interface que siga as diretrizes do Material Design.

FinanceApp foi concebido como um gerenciador financeiro pessoal que permite registrar receitas, despesas e acompanhar o saldo em tempo real, focando em:

- Persistência local com **Room**;
- Arquitetura **MVVM**;
- Interface intuitiva baseada em **Material 3**.

Visão Geral da Arquitetura

- **Model**: classe `Transaction` mapeada como entidade Room;
- **DAO**: `TransactionDao` – operações CRUD e consultas filtradas;
- **Repository**: abstrai a fonte de dados, expondo *Flows*;
- **ViewModel**: `TransactionViewModel` – lógica de negócio e estado UI;
- **View**: `MainActivity` + `RecyclerView` + diálogos de cadastro/filtro.



Persistência de Dados com Room

- Entidade anotada com `@Entity` no pacote `model`: `Transaction`.
- Conversores de tipo utilizados para `Date` e enum `TransactionType`.
- Banco de dados singleton implementado via padrão `Room.databaseBuilder` no pacote `data`.

Algoritimo: Trecho de `AppDatabase.kt` (pacote `data`)

```
1  package com.example.financeapp.data
2
3  import android.content.Context
4  import androidx.room.*
5  import com.example.financeapp.model.Transaction
6
7  @Database(entities = [Transaction::class], version = 1)
8  @TypeConverters(Converters::class)
9  abstract class AppDatabase : RoomDatabase() {
10     abstract fun transactionDao(): TransactionDao
11     companion object {
12         @Volatile
13         private var INSTANCE: AppDatabase? = null
14
15         fun getDatabase(context: Context): AppDatabase {
16             return INSTANCE ?: synchronized(this) {
17                 val instance = Room.databaseBuilder(
18                     context.applicationContext,
19                     AppDatabase::class.java,
20                     "finance_database"
21                 ).build()
22                 INSTANCE = instance
23                 instance
24             }
25         }
26     }
27 }
28
```

Funcionalidades Implementadas

Resumo das Funcionalidades

1. **Cadastro** de transações (receitas ou despesas);
2. **Listagem** com filtro por tipo e intervalo de datas;
3. **Edição** de transações existentes;
4. **Exclusão** de transações;
5. **Ordenação** da lista e cálculo de totais em tempo real.

Todas as funcionalidades são demonstradas nas telas seguintes com trechos de código.

1. Cadastro de Transações

Algoritimo: Método addTransaction em TransactionViewModel

```
1 fun addTransaction(transaction: Transaction) {
2     viewModelScope.launch {
3         repository.insertTransaction(transaction)
4     }
5 }
```

O diálogo `dialog_transaction.xml` provê um formulário validado. A ação de salvar invoca o *ViewModel*, e o registro é imediatamente persistido via DAO.

2. Listagem com Filtro/Busca

Algoritimo: getFilteredTransactions

```
1  fun getFilteredTransactions(type: TransactionType?, startDate: Date,
       endDate: Date) {
2      viewModelScope.launch {
3          if (type != null) {
4              repository.getTransactionsByTypeAndDateRange(type, startDate,
         endDate)
5                  .collect { filtered ->
6                      _transactions.value = filtered
7                  }
8          } else {
9              loadTransactions()
10         }
11     }
12 }
```

O usuário escolhe o tipo de transação e o intervalo de datas em um diálogo de filtro. A lista (`RecyclerView`) é atualizada reativamente com `StateFlow`.

3. Edição de Transações

Algoritimo: updateTransaction

```
1 fun updateTransaction(transaction: Transaction) {
2     viewModelScope.launch {
3         repository.updateTransaction(transaction)
4     }
5 }
```

Um duplo clique em um item abre o mesmo diálogo, agora pré-preenchido. Após salvar, o `StateFlow` reflete a alteração sem recarregar a tela.

4. Exclusão de Transações

Algoritimo: deleteTransaction

```
1 fun deleteTransaction(transaction: Transaction) {
2     viewModelScope.launch {
3         repository.deleteTransaction(transaction)
4     }
5 }
```

Um *swipe* na lista aciona o `ItemTouchHelper`, que confirma a exclusão com uma `Snackbar` oferecendo a opção de desfazer.

5. Ordenação e Totais

Algoritimo: sortTransactionsByDate

```
1 fun sortTransactionsByDate() {
2     viewModelScope.launch {
3         _transactions.value = _transactions.value.sortedBy { it.date }
4     }
5 }
```

Botões no `AppBar` permitem alternar a ordenação por data ou valor. Totais de receitas e despesas são calculados via combinação de `Flow`s no *ViewModel*.

Interface e Navegação

- Tela inicial com `SplashScreen` e `MainActivity` contendo `Toolbar`.
- `FloatingActionButton` para adicionar novas transações.
- Menus na `AppBar` para filtros, ordenação e ajuda.
- Navegação baseada em diálogos, adotando o padrão **single-activity**.

Qualidade de Código e Boas Práticas

- Arquitetura **MVVM**, com separação clara entre `model`, `repository`, `viewmodel` e `ui`.
- Injeção de dependência simples com `ViewModelProvider.Factory`.
- Uso de corrotinas e `StateFlow` para atualização reativa da UI.
- Convenções idiomáticas do Kotlin e comentários pontuais no código.

Conclusão

Conclusão

O **FinanceApp** cumpre todos os requisitos propostos:

- Cinco funcionalidades principais implementadas com sucesso;
- Persistência local assegurada, mesmo após reiniciar o app;
- Interface moderna seguindo diretrizes do Material Design;
- Código limpo, organizado, testável e documentado.

O projeto demonstra domínio em persistência de dados, lógica de negócio e aplicação de boas práticas no desenvolvimento Android.