UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de disertație

Aplicație mobilă pentru gestionarea finanțelor personale

Absolvent

George-Cristian Vasile

Coordonator științific

Conf.dr. Radu Gramatovici

București, iunie-iulie 2025

Rezumat

Scopul lucrării constă în dezvoltarea unei aplicații mobile menită să faciliteze gestionarea eficientă a finanțelor personale. Dezvoltarea aplicației va consta atât în scrierea codului pentru logică și interfață, cât și în proiectarea unei arhitecturi optimizate pentru dispozitive mobile, capabilă să gestioneze eficient un volum mare de tranzacții pentru a asigura o experiență fluidă utilizatorilor.

Aplicația va oferi utilizatorilor un pachet complex de funcționalități pentru gestionarea finanțelor personale. Aceasta va centraliza toate conturile bancare într-o singură aplicație, utilizatorul înregistrând și clasificând veniturile și cheltuielile manual sau prin încărcarea periodică a unor extrase de cont. Pe baza categorisirii cheltuielilor, utilizatorul va avea posibilitatea să-și monitorizeze cheltuielile prin stabilirea unor bugete dedicate. De asemenea, aplicația va pune la dispoziție diverse metode de analiză a tiparelor de cheltuieli ale utilizatorului prin afișarea unor statistici și rapoarte grafice.

Abstract

The purpose of the thesis is to develop a mobile application meant to facilitate efficient personal finances management. The development of the application will consist both in writing the code for the logic and interface and in designing an architecture optimized for mobile devices, capable of efficiently handling a large volume of data to ensure a fluid user experience.

The application will provide users with a complex set of functionalities meant for managing personal finances. It will centralize all bank accounts in a single application, allowing users to input and classify their income and expenses both manually and by periodically uploading bank statements. Based on the categorization of expenses, users will have the ability to monitor their spending by setting dedicated budgets. Additionally, the application will offer various methods of analyzing the user's spending patterns by displaying statistics and graphs.

Cuprins

Capitolul I

Introducere

1.1 Prezentarea generală a temei și aplicației

Tema lucrării reprezintă dezvoltarea unei aplicații mobile destinate gestionării finanțelor personale, integrând funcționalități ce vor ajuta utilizatorul să monitorizeze cheltuielile și să ia decizii informate pentru un control mai eficient a resurselor personale. Funcționalitățile propuse includ categorisirea cheltuielilor introduse atât manual cât și importate din extrase de cont, stabilirea de bugete per-categorie și prezentarea de rapoarte grafice și statistici bazate pe acestea. Totodată, aplicația va permite centralizarea tuturor conturilor bancare ale utilizatorului într-un punct comun, având capacitatea de a gestiona multiple monede.

Aplicația a fost dezvoltată cu ajutorul framework-ului *Flutter* în limbajul *Dart*. Acesta este foarte util în dezvoltarea aplicațiilor mobile cross-platform, fiind modern și foarte performant. Pentru stocarea datelor am ales *SQLite* prin intermediul bibliotecii *Drift* deoarece este o soluție eficientă, declarativă și ușor de implementat pentru gestionarea datelor relaționale pe un dispozitiv mobil.

Arhitectura aplicației este de tip monolit, întreaga logică a aplicației, codul pentru gestionarea datelor și cel pentru afișarea interfeței fiind integrat într-o singură unitate de dezvoltare. Deși arhitectura este de tip monolit, aceasta este structurată pe tipuri de clase distincte, denumite în continuare module, fiecare fiind responsabil pentru o funcționalitate bine-definită. Modulele aplicației sunt strâns legate, ceea ce limitează extensibilitatea dar asigură interacțiunea eficientă, fără a fi necesare protocoale suplimentare de comunicare.

1.2 Scopul și motivația alegerii temei

După explorarea unor aplicații existente cu același scop am identificat anumite limitări, precum suportul pentru o singură monedă, interfețe grafice neintuitive și absența unor statistici relevante pentru utilizatori. Având nevoie de o asemenea aplicație, am decis să implementez una care să integreze atât funcționalitățile de bază regăsite și în alte aplicații cât și funcționalitățile care lipseau și, în opinia mea, sunt necesare.

1.3 State of the art

Pentru dezvoltarea aplicației am utilizat tehnologii *state of the art* menite să faciliteze crearea unei aplicații performante cu un aspect modern. Alegerea acestor tehnologii evidențiază alinierea la cele mai noi și populare practici din industrie.

1.3.1 Flutter si riverpod

Flutter este un framework open-source pentru dezvoltarea unor aplicații frumoase, compilate nativ, pentru mai multe platforme, utilizând același cod sursă. Flutter este susținut și utilizat de Google, este de încredere pentru branduri renumite la nivel global și este întreținut de o comunitate de dezvoltatori din întreaga lume. Codul Flutter este compilat în cod mașină ARM sau Intel, precum și în JavaScript, pentru a oferi performanță rapidă pe orice dispozitiv.[1]

Riverpod este un framework reactiv pentru legarea datelor în aplicațiile Flutter/Dart. Acesta optimizează procesul de re-desenare a componentelor vizuale. Spre deosebire de alte soluții, acesta permite re-desenarea doar a widget-urilor direct dependente de starea schimbată, reducând astfel consumul de resurse și îmbunătățind performanța aplicației.[2]

Prin integrarea *Riverpod* cu *Flutter* am dezvoltat o interfață cu aspect modern care utilizează resursele în mod eficient și se adaptează corespunzător la mărimea ecranului dispozitivului pe care este afișată.

1.3.2 SQLite prin Drift

Drift este o bilbiotecă avansată pentru gestionarea bazelor de date în Dart și Flutter, care permite scrierea de interogări SQL sigure și eficiente, oferind fluxuri de date reactive care se actualizează automat. Drift suportă atât scrierea interogărilor în Dart, cât și direct în SQL, iar baza de date poate fi definită folosind o abordare code-first.[3]

Consider Drift/SQLite ca fiind state of the art deoarece, în opinia mea, este soluția optimă pentru implementarea unei baze de date relaționale integrate, atât pe dispozitive mobile cât și pe dispozitive embedded. Prin utilizarea aceste tehnologii am redus semnificativ timpul de dezvoltare, menținând performanța aplicației. De asemenea, baza de date și interogările sunt generate din cod Dart, nefiind necesară scrierea de cod SQL.

1.3.3 Tesseract

Tesseract este un motor OCR (Optical Character Recognition) open-source, capabil să recunoască textul imprimat în imagini și să-l convertească în format text editabil. Dezvoltat inițial de către HP între 1984 și 1994, a fost publicat ulterior în spațiul open-source și continuă să fie îmbunătățit. Versiunea 4.0 (utilizată în proiect) a introdus folosirea unui subsistem de rețele neuronale. [4]

Este discutabil dacă Tesseract rămâne, în prezent, o opțiune *state of the art* pentru OCR. Conform studiilor recente, modelele de deep learning au o acuratețe mai mare, în special pentru textele scrise de mână. [5] Totuși, aceste modele implică costuri monetare de utilizare, utilizarea de API-uri publice ceea ce ar risca protecția datelor cu caracter personal din extrasele de cont, sau în cazul celor ce pot fi rulate local costurile de procesare ar fi mult prea ridicate pentru un dispozitiv mobil.

În cadrul aplicației nu este necesară încărcarea de text scris de mână deoarece fișierele vor fi extrase de cont generate de către aplicații bancare, deci îmbunătățirea de performanță adusă de către modelele bazate pe deep learning nu ar fi destul de mare pentru a justifica costurile crescute.

Așadar, întrucât Tesseract este open-source și poate rula local cu costuri relativ scăzute de procesare consider că acesta este o metodă *state of the art* în contextul cerințelor aplicației. O alternativă viabilă ar fi Google ML Kit deoarece poate rula local și oferă o procesare mai rapidă ca Tesseract. Totuși, am ales Tesseract deoarece a avut o acuratețe mai mare în testele mele. Această îmbunătățire poate varia în funcție de metodele de preprocesare utilizate.

1.4 Structura lucrării

Lucrarea este structurată în cinci capitole.

Acest capitol conține o scurtă prezentare a temei și aplicației. În cadrul lui sunt prezentate toate noțiunile ce pot fi considerate *state of the art*. De asemenea, am adăugat descrieri pentru termenii preluați din limba engleză.

Al doilea capitol descrie cerințele stabilite înainte de crearea aplicației și modul în care acestea au influențat proiectarea arhitecturii. Acest capitol descrie și structura bazei de date și a entităților acesteia.

În al treilea capitol voi prezenta anumite detalii de implementare, explicând soluțiile adoptate pentru a livra cerințele identificate și modul în care acestea au influențat dezvoltarea aplicației.

În cadrul celui de-al patrulea capitol voi prezenta interfața, specificând funcționalitățile principale ale aplicației și tehnicile utilizate pentru îmbunătățirea experienței utilizatorilor.

Al cincilea capitol va conține concluziile, fiind atât analizate deciziile luate în fazele de proiectare si implementare cât și o retrospectivă asupra modului general de lucru, evidențiind lucrurile învățate pe parcursul proiectului.

1.5 Termeni

Am folosit anumiți termeni în limba engleză deoarece nu au un sinonim direct în limba română. Aceștia au fost utilizați cu înțelesurile următoare:

- State of the art: cele mai recente și inovatoare realizări dintr-un anumit domeniu, în cazul acesta dezvoltarea software
- Framework: o implementare abstractizată care ușurează dezvoltarea unei implementări mai specifice
- Cross-platform: capabilitatea unui sistem software de a rula pe mai multe sisteme de operare
- Open-source: sistem informatic al cărui cod sursă este accesibil oricui
- Embedded: sistem informatic integrat care rulează pe un dispozitiv hardware proiectat pentru a îndeplini sarcini specifice
- Backup: copie de rezervă a datelor unui sistem informatic
- Feedback: răspunsuri la acțiunile utilizatorului
- Widget: componentă fundamentală din *Flutter* ce organizează componentele vizuale și definește funcționalitatea acestora

Capitolul II

Proiectarea aplicației

2.1 Definirea cerințelor

Acest subcapitol va detalia cerințele funcționale și non-funcționale identificate în faza de proiectare. Acestea au fost extrase pe baza funcționalităților descrise în subcapitolele 1.1 și 1.2. Ordinea este dată de dependențele de implementare, cum ar fi necesitatea implementării bazei de date înainte de definirea conturilor și tranzacțiilor, și nu de importanța lor generală în contextul aplicației. Cerințele funcționale sunt marcate cu *F*, iar cele non-funcționale cu *N*.

- Aplicația trebuie să ruleze pe dispozitive mobile și să se adapteze automat la diferite mărimi ale ecranelor. (*N)*
- Baza de date trebuie sa fie portabilă și stocată local pe dispozitiv din motive de protecție a datelor cu caracter personal. (*N)*
- Datele aplicației trebuie să fie exportabile și re-importabile deoarece unii utilizatori pot dori re-importarea datelor în anumite cazuri, cum ar fi schimbarea telefonului. (*F)*
- Utilizatorul va putea selecta o monedă principală. Aceasta va fi folosită atât pentru a converti automat sumele și tranzacțiile din conturi cu monede diferite cât și pentru a afișa statisticile și graficele din pagina principală într-o singură monedă. (*F)*
- Aplicația va colecta o listă de monede și cursuri valutare aferente, pentru a afișa conversii precise la moneda principală. (*F)*
- Utilizatorul va putea defini conturi bancare în orice monedă din lista disponibilă. (*F)*
- Adițional, conturile vor putea fi de tip "Depozit cu dobândă fixă" pentru care utilizatorul poate defini dobânda, impozitul și perioada de lichidare. (*F)*
- Aplicația va conține o listă implicită de categorii de tranzacții. (*F)*
- Utilizatorul va putea modifica și reordona lista de categorii. (*F*)
- Aplicația permite adăugarea manuală de tranzacții. (*F)*
- Aplicația permite adăugarea automată de tranzacții prin încărcarea unor extrase de cont. Acestea pot fi de tip CSV, XLS și PDF. Fișierele de tip PDF pot fi atât generate cât și scanate. (*F)*
- Aplicația trebuie să fie proiectată eficient pentru a rula fluid chiar și după inserarea unui număr mare de tranzacții (*N*)

- Aplicația va pune la dispoziție crearea de plăți periodice, care să fie de o anumită categorie cu o anumită sumă, pentru a ușura gestionarea abonamentelor și subscripțiilor. (*F)*
- Plățile periodice nu vor fi debitate automat din contul asignat deoarece, din diverse motive, tranzacția poate eșua sau poate fi întârziată. Așadar, utilizatorul va decide când se efectuează fiecare plată în mod manual. (*F)*
- Utilizatorul poate seta bugete per categorie de tranzacție și va fi notificat în cazul în care acestea sunt depășite. Adițional, trebuie să existe opțiunea creării unui buget general, ce include toate categoriile. (*F)*
- Tranzacțiile, categoriile acestora și conturile aferente vor fi ușor modificabile. Modificările vor fi afișate în timp real. (*F)*
- O excepție la cerința anterioară sunt categoriile permanente de plăți, care nu vor putea fi șterse (Transfer, Dobândă și Necunoscut). (*F)*
- Utilizatorul va putea efectua transferuri între conturile cu aceeași monede. Un transfer va crea două tranzacții cu categoria "Transfer", una pentru contul debitor și una pentru contul creditor. (*F*)
- La crearea unui cont de tip "Depozit" aplicația va genera automat o tranzacție viitoare la data lichidării cu tipul "Dobândă", cu valoarea dobânzii câștigate. (*F)*
- În cazul ștergerii unei categorii, tranzacțiile aferente vor fi convertite la categoria implicită "Necunoscut". (*F*)
- În cazul modificării soldului unui cont va fi creată automat o tranzacție cu categoria "Necunoscut". (*F)*
- Aplicația va calcula și afișa un sold total al conturilor, convertit la moneda principală. (*F)*
- Utilizatorul va putea exclude anumite conturi de la acest sold. Depozitele trebuie să fie excluse implicit. (*F)*
- Aplicația va conține o pagină dedicată pentru statistici si grafice. (*F)*
- Pagina va conține o secțiune unde să fie detaliate sumele tranzacțiilor per categorii, separate pentru venituri și cheltuieli. (*F)*
- Pagina va conține grafice ce ilustrează veniturile, cheltuielile și soldul total din ultimele șase luni. (*F)*

2.2 Arhitectura

Arhitectura este de tip monolit, fiind potrivită pentru o aplicație mobilă ce nu necesită separarea dintre logica de interfață și cea de manipulare a datelor. Pentru ușurarea implementării unei interfețe ce se actualizează în timp real am ales utilizarea librăriei *Riverpod,* fiind unul dintre cele mai mai moderne sisteme de gestionare a stărilor. Următoarea figură, desenată cu draw.io, ilustrează fluxul logicii în aplicație. Deși arhitectura este de tip monolit aceasta are o delimitare clară a scopului claselor/modulelor.

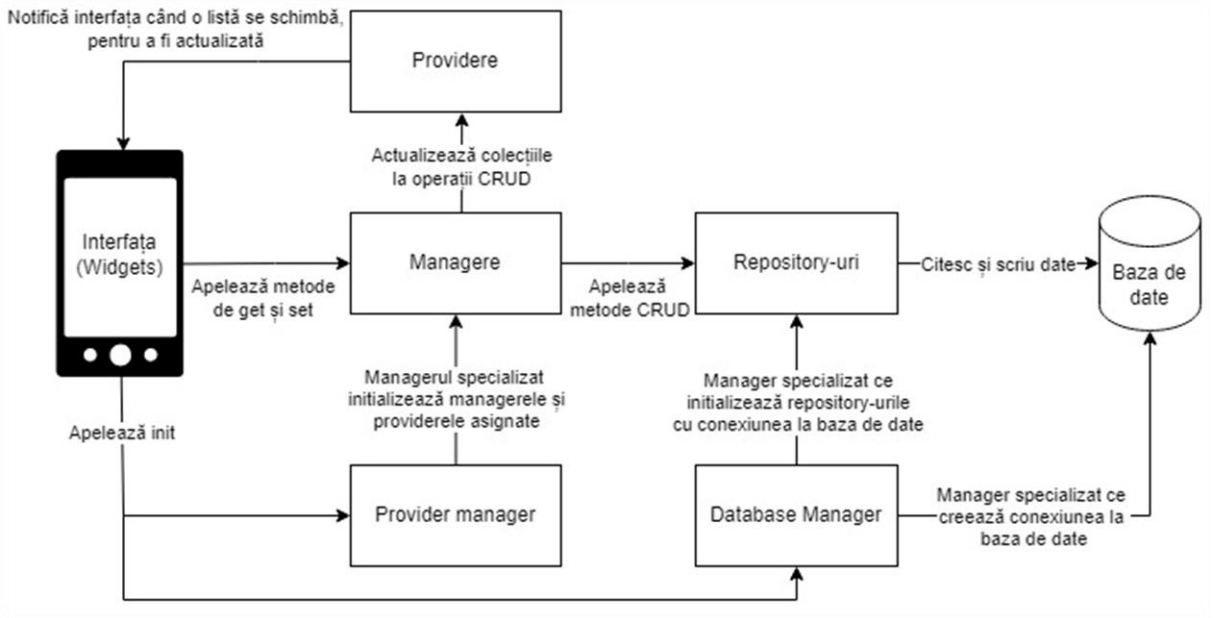

Figura 2.1 Fluxul logicii în aplicație

Widget-urile sunt componente specifice Flutter și definesc structura vizuală a interfeței. Ideal, aceste componente ar trebui să nu conțină deloc logică ce exclude maparea datelor. Datele de care au nevoie widget-urile se pot clasifica în două categorii. Prima categorie este reprezentată de datele care trebuie actualizate în timp real, pentru care se abonează la *Providere* (spre exemplu lista de tranzacții). A doua categorie conține date care pot fi calculate o singură dată, pentru care se apelează metode din *Managere* (spre exemplu, ratele de schimb valutar).

Providerele sunt componente specifice librăriei *Riverpod* și conțin liste de obiecte statice, stocate în memorie. Librăria gestionează în mod automat logica de notificare a componentelor abonate atunci când colecțiile se schimbă.

Managerele centralizează majoritatea logicii din aplicație și reduc complexitatea altor componente. Acestea sunt responsabile de manipularea și prelucrarea datelor legate de interfață. Unele managere (*Accounts*, *Transactions*, etc) au un *Provider* și un *Repository* echivalent și sunt responsabile de actualizarea acestora la orice operație care modifică

obiecte. De asemenea, pe diagrama de flux sunt vizibile 2 managere – *Provider Manager* și *Database Manager* – care au ca unic scop inițializarea datelor din managere, providere și repository-uri.

Repository-urile gestionează operațiile de bază CRUD (Create, Read, Update și Delete) și sunt singurul punct de interacțiune cu baza de date. Singura logică pe care acestea o pot conține sunt filtrele ce pot fi aplicate interogărilor SQL.

2.3 Baza de date

Pe baza diagramei de mai jos voi descrie scopul fiecărei tabele și deciziile de proiectare care au influențat structura acesteia. Nu voi include fiecare relație deoarece unele se subînțeleg.

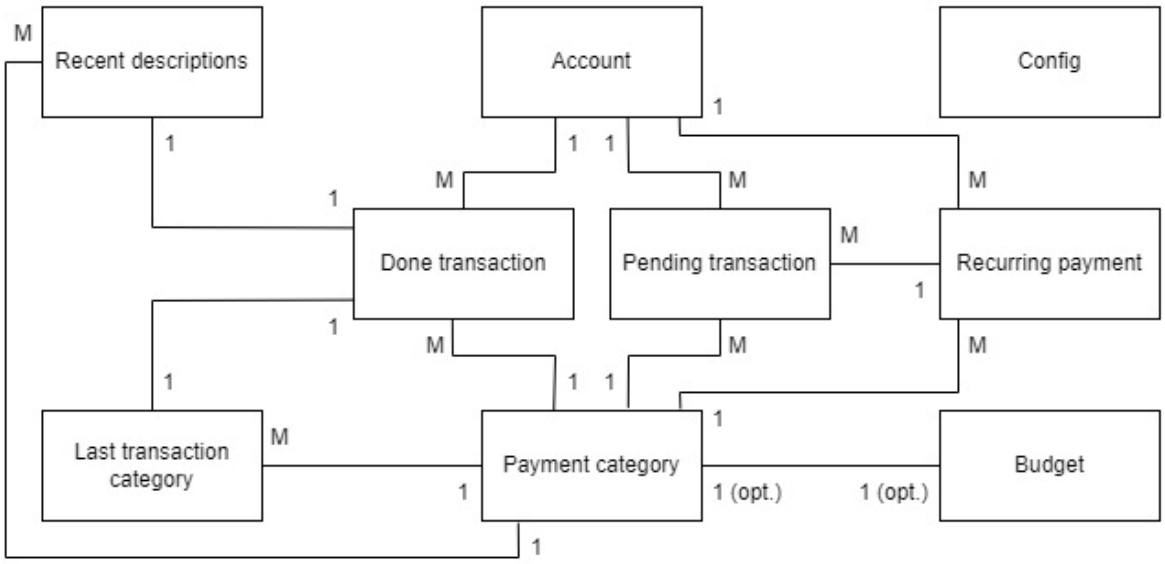

Figura 2.2 Diagrama entitate-relație

Tabela *Account* va conține informațiile despre conturi și depozite, cum ar fi numele și soldul actual. Acestea vor avea asignate multiple tranzacții și plăți planificate.

Tabelele *Done transaction* și *Pending transaction* modelează tranzacțiile ce au fost deja efectuate și cele ce sunt planificate să fie procesate la o anumită dată, respectiv. O tranzacție aparține de un cont și are o anumită categorie. Într-o aplicație desktop sau web aș fi structurat datele să fie stocate într-o singură tabelă dar am ales să le stochez separat deoarece separarea programatică a acestora ar fi impactat semnificativ performanța pe un dispozitiv mobil.

Tabela *Payment Category* conține informațiile unei categorii de tranzacție, cum ar fi numele, iconița și culoarea acesteia. O categorie poate avea un buget asignat.

Tabela *Budget* conține bugetul asignat unei categorii de cheltuieli. Relația cu categoriile de tranzacții este opțională deoarece bugetul ce nu este asignat unei categorii include toate cheltuielile dintr-o lună. Acesta este și motivul pentru care tabela *Payment Category* nu are o coloană opțională pentru *Budget*.

Tabela *Recurring Payment* include informațiile despre plățile planificate. O plată planificată conține informații precum data de începere, intervalul de debitare/creditare și durata acesteia. Aceasta are o anumită categorie și este asignată unui cont, putând avea multiple plăți planificate.

Tabela *Last Transaction Category* conține ultima categorie asignată unei tranzacții efectuate, în funcție de numele comerciantului. Aceste date sunt folosite pentru a sugera categorii în mod automat în funcție de cele asignate anterior unor tranzacții similare. Aceste date sunt redundante și puteau fi extrase din tabelele *Done transaction* și *Payment Category* dar ar fi necesitat o filtrare intensivă din punct de vedere a resurselor de procesare. Așadar, am ales performanța rapidă în detrimentul spațiului de stocare.

Tabela *Recent Descriptions* conține ultimele descrieri asignate tranzacțiilor dintr-o anumită categorie, ordonate după data procesării. La fel ca în cazul tabelei *Last Transaction Category* aceste date sunt redundante, dar sunt stocate duplicat pentru a îmbunătății performanța. Pentru a micșora impactul asupra spațiului de stocare *Repository-ul* echivalent va limita numărul de date stocate per categorie.

Tabela *Config* conține configurația persistentă a aplicației, mai precis numele de utilizator, moneda implicită și data ultimului backup utilizată la notificarea utilizatorului. *Repository-ul* tabelei se asigură ca aceasta conține o singură entitate de tip *ConfigModel*. Utilizarea unei tabele pentru stocarea unei astfel de configurații este exagerată dar am dorit să centralizez toată logica de persistență. În mod normal, o asemenea configurație ar fi stocată într-un fișier text cu format standardizat, precum .JSON.

Capitolul III

Implementarea aplicației

Acest capitol va intra în detaliile de implementare demne de menționat, în special pe partea de logică. Atât codul cât și comentariile din cadrul acestuia sunt scrise în limba engleză.

3.1 Baza de date

Baza de date este generată cu ajutorul paradigmei code-first, folosindu-se funcționalități SQLite din librăria Drift. Secvența de cod de mai jos conține un exemplu restrâns de definire a bazei de date.

```
@DriftDatabase(tables: [DoneTransaction, Budget])
class AppDatabase extends _$AppDatabase {
  final String dbPath;

  AppDatabase(this.dbPath) : super(NativeDatabase(File(dbPath)));

  @override
  int get schemaVersion => 5;

  @override
  MigrationStrategy get migration => MigrationStrategy(
        onUpgrade: (migrator, from, to) async {
          if (from == 1) {
            // Specify that any schema version newer than 1 contains these
changes
            await migrator.createTable(budget);
            await migrator.addColumn(
    doneTransaction, doneTransaction.isIncludedInStatistics);
          }
      );
}
```

Baza de date este definită prin adnotarea *@DriftDatabase*, care primește ca parametru o listă de tabele. Rularea comenzii `dart run build_runner build` generează automat clasa *_$AppDatabase* ce încapsulează întreaga logică de creare și migrare a bazei de date. Pentru actualizarea automată a bazei de date (numită migrare) pe dispozitive ce conțin o versiune mai veche trebuie să incrementăm parametrul *schemaVersion* și să definim un *MigrationStrategy* ce specifică toate modificările făcute. Instanța de *AppDatabase* trebuie creată, salvată și pasată repository-urilor.

O tabelă trebuie să extindă clasa *Table* din Drift. Secvența următoare de cod reprezintă o variantă restrânsă a definiției tabelei DoneTransaction. Aceasta exemplifică

atât crearea tabelei și definirea coloanelor cât și stabilirea cheilor externe, setarea valorilor implicite și specificarea constrângerilor.

```
import 'package:drift/drift.dart';

class DoneTransaction extends Table {
  IntColumn get id => integer().autoIncrement()();

  TextColumn get currency => text().withLength(min: 1, max: 4)();

  BoolColumn get isIncludedInStatistics =>
      boolean().withDefault(const Constant(true))();

  IntColumn get accountId =>
      integer().references(Account, #id, onDelete: KeyAction.cascade)();
}
```

Pentru a interoga baza de date se poate folosi tot o abordare code-first, folosindu-se metode specifice Drift. Următorul exemplu conține o interogare a tabelei *DoneTransaction*, fiind incluse clauze *WHERE, ORDER BY* și *JOIN*.

```
static Future<List<TransactionModel>> getTransactionsByTimeInterval(
    DateTime startDate, DateTime endDate) async {
  return await (_database.select(_database.doneTransaction)
        ..where((t) => t.date.isBetweenValues(startDate, endDate))
        ..orderBy([
          (t) => OrderingTerm(expression: t.date, mode: OrderingMode.desc)
        ]))
      .join([
    leftOuterJoin(
        _database.paymentCategory,
        _database.paymentCategory.name
            .equalsExp(_database.doneTransaction.categoryName))
  ]).map((row) {
    return TransactionModel.fromDoneData(
      row.readTable(_database.doneTransaction),
      row.readTable(_database.paymentCategory),
    );
  }).get();
}
```

3.2 Sistemul de backup

Aș fi putut crea un sistem de backup automat, prin intermediul Firebase, dar această opțiune ar fi necesitat atât resurse monetare pentru alocarea spațiului în cloud cât și includerea unei metode de obfuscare a datelor pentru protecția informațiilor cu caracter personal. Așadar, am optat pentru un sistem de backup manual.

Sistemul este proiectat să notifice utilizatorul că backup-ul anterior a expirat după 30 de zile, data acestuia fiind stocată în tabela *Config*. Informația este afișată atât în cadrul unui mesaj toast la pornirea aplicației cât și prin crearea unei notificări push din Android. Pentru afișarea acesteia aplicația are nevoie de permisiuni pentru notificări, permisiunea fiind verificată la fiecare pornire a aplicației cu ajutorul pachetului *permission_handler*.

Crearea unui backup generează un fișier de tip .db iar utilizatorul este responsabil de gestionarea acestuia.

Metoda aleasă are 3 dezavantaje principale. În primul rând, crearea backup-ului este încredințată utilizatorului, acesta putând să ignore mesajele. În al doilea rând, utilizatorul poate alege să stocheze fișierele de tip .db pe dispozitivul personal iar în cazul pierderii sau defectării acestuia ele vor fi pierdute. În al treilea rând, aplicația nu conține logică pentru verificarea versiunii bazei de date. În cazul în care se încearcă încărcarea unei baze de date salvate dintr-o versiune mai nouă într-o versiune mai veche pot apărea comportamente neașteptate.

3.3 Gestionarea monedelor și a cursului valutar

Pentru afișarea unui curs valutar la zi am optat pentru utilizarea API-ului public Frankfurter deoarece acesta nu are nevoie de chei și nu impune limite de utilizare.
Frankfurter este un API open-source gratis care urmărește cursurile de schimb de referință publicate de surse instituționale și non-comerciale, precum Banca Centrală Europeană. [5] Acest API rulează de mai mult de un deceniu dar, în cazul închiderii neprevăzute a acestuia, am inclus un mesaj de eroare și date implicite colectate manual pentru a asigura disponibilitatea aplicației, dar cu rate de conversie inexacte.

La pornirea aplicației *CurrencyManager* trimite 2 apeluri HTTP către API. Primul dintre ele colectează lista completă de monede suportate de API. Al doilea colectează ratele de schimb pentru moneda euro. Acestea sunt folosite ca valori de referință pentru calcularea ratelor de schimb dintre oricare 2 monede. Pentru calcularea ratelor de schimb se folosește următoarea formulă, unde $R_{A-->B}$ reprezintă rata de schimb de la A la B.

$$Rx \rightarrow y = \frac{R\ eur \rightarrow y}{R\ eur \rightarrow x}$$

Formula 3.1 – Calculul ratei de schimb valutar între moneda X și Y

CurrencyManager trimite aceste request-uri HTTP doar la pornirea aplicației, datele fiind stocate în colecții statice. În cazul trecerii miezului nopții în timpul folosirii aplicației datele nu sunt actualizate, fiind necesară o repornire a acesteia.

3.4 Tehnici de asigurare a scalabilității

Pentru asigurarea scalabilității aplicației am analizat scenariile rezonabile de utilizare a acesteia. Un utilizator normal nu va avea niciodată un număr excesiv de mare de conturi bancare, categorii de plată sau plăți recurente. De asemenea, tranzacțiile planificate pot fi mai numeroase dar nu ar trebui să depășească ordinul sutelor. Colecțiile de descrieri

recente și categorii de tranzacții nu sunt încărcate niciodată complet în memorie și sunt preluate doar când sunt necesare, deci pot fi de asemenea excluse.

Tranzacțiile efectuate sunt singurul factor ce poate impacta negativ performanța. După utilizarea îndelungată a aplicației acestea pot depăși ordinul zecilor de mii sau chiar al sutelor de mii. Așadar, încărcarea simultană a acestora în memoria RAM ar putea cauza probleme semnificative de performanță sau chiar închiderea acesteia.

Pentru prevenirea eventualelor probleme am ales filtrarea tranzacțiilor după data efectuării. Aplicația are filtre implicite pentru luna curentă, dar acestea pot fi schimbate oricând cu orice interval prin utilizarea interfeței. Mai multe detalii despre metodele de filtrare vor fi prezentate în capitolul IV. Partea relevantă acum este logica de filtrare a tranzacțiilor, care se setează din *TransactionFiltersProvider*. La fiecare modificare a filtrelor se interoghează baza de date pentru noul interval iar colecția rezultată se stochează în *FiltersProvider*, conform secvenței de cod:

```
void setDateFilters(
    WidgetRef ref, DateTime newStartTime, DateTime newEndTime) {
  state =
 state.copyWith(startDateTime: newStartTime, endDateTime: newEndTime);

  TransactionManager.filterTransactionsByTime(ref, newStartTime,
newEndTime);
}
```

3.5 Metode de importare a tranzacțiilor

Pentru importarea tranzacțiilor, utilizatorul poate încărca din dispozitiv fișiere cu formatele suportate - CSV, XLS și PDF. *TransactionImportManager* este managerul responsabil de această logică, specificând și formatele suportate. Singura bancă internațională disponibilă este Revolut, celelalte formate de extrase fiind specifice băncilor românești.

```
// Enumeration containing all available import methods
enum ImportMethod {
  revolutCsv,
  roRaiffeisenXlsx,
  roBcrCsv,
  roIngCsv,
  roBrdPdf,
  roBtPdf,
  roLibraPdf,
  roUnicreditPdf,
  roCecPdf,
}
```

Importarea tranzacțiilor din formatele CSV și XLS este trivială, utilizându-se pachetele *csv* și *excel* pentru încărcarea automată a liniilor din fișiere. După încărcarea

acestora este necesară doar identificarea celulelor ce conțin informații relevante și prelucrarea acestora.

În cazul încărcării unui fișier PDF, fișierele pot fi de două tipuri. Fișierele generate de aplicațiile bancare sunt bazate pe text, textul încorporat în fișier fiind ușor extractabil. Fișierele scanate nu dețin astfel de informații și trebuie tratate diferit. La încărcarea unui fișier PDF se încearcă inițial prima metodă, a doua fiind folosită doar dacă aceasta eșuează.

Comparativ cu extragerea tranzacțiilor din fișiere de tip tabel, extragerea din fișiere PDF este mai dificilă din două motive principale. În primul rând, extrasele de cont PDF generate conțin mult text ce trebuie ignorat (informații de contact, sold zilnic, sold total, logo-urile băncii, ș.a.m.d.). În al doilea rând, nu mai putem parcurge informațiile linie cu linie deoarece detaliile unei tranzacții se pot întinde pe linii multiple. Așadar, pentru fiecare format de fișier PDF trebuie identificat un anumit tipar pe baza căruia să extragem informațiile relevante, incremental pentru fiecare tranzacție.

3.5.1 Extragerea tranzacţiilor din fişiere PDF generate

Pentru extragerea textului din fișierele generate am folosit librăria *SyncFusion*, care pune la dispoziție o metodă ce parcurge fiecare pagină din document și returnează un string.

```
static Future<String> extractTextFromTextBasedPdf(String filePath) async {
  // Read the PDF file as bytes.
  final file = File(filePath);
  final bytes = await file.readAsBytes();

  // Open the PDF document using the Syncfusion library.
  final document = syncfusion_pdf.PdfDocument(inputBytes: bytes);

  // Extract text from the PDF.
  final text = syncfusion_pdf.PdfTextExtractor(document).extractText();

  // Dispose of the document to release resources.
  document.dispose();

  // Return the extracted text.
  return text;
}
```

Extragerea textului propriu-zis dintr-un asemenea fișier va avea o acuratețe de 100%. Din păcate, acest procent nu este reflectat în acuratețea tranzacțiilor extrase deoarece prin această metodă nu se pot identifica corect coloanele. Majoritatea băncilor folosesc termenii de debit și credit iar sumele tranzacțiilor nu au + și - ci sunt aliniate sub coloana respectivă. Din moment ce nu putem extrage alinierea sumelor în coloane, trebuie să deducem tipul tranzacției după anumite cuvinte cheie. Mai exact, descrierile tranzacțiilor

credit (pozitive) tind să conțină “încasare”, “depunere” sau “credit”. În mod evident, această metodă de diferențiere nu are o acuratețe perfectă.

3.5.2 Extragerea tranzacţiilor din fişiere PDF scanate

Pentru extragerea tranzacțiilor fin fișiere scanate sau fotografiate folosim Optical Character Recognition, prescurtat în continuare OCR. Metodele “state of the art” implică rețele neuronale și deep learning. După cum am menționat și motivat în secțiunea 1.3.3, am ales Tesseract, prin intermediul pachetului *flutter_tesseract_ocr*. Pentru a obține o acuratețe mai mare sunt aplicate diverse metode de preprocesare și postprocesare ce vor fi prezentate în cadrul acestei sectiuni. Partea de preprocesare are impactul cel mai mare asupra rezultatului final deoarece Tesseract nu poate produce rezultate utile dacă îi oferim imagini neclare sau înclinate excesiv.

Am ales metode deja testate pentru preprocesare, precum conversia la nuanțe de gri, aplicarea unui filtru Gaussian pentru reducerea zgomotului și binarizarea adaptivă care ajută textul să iasă în evidență față de fundal. [7] Fișierele PDF sunt convertite în imagini pagină cu pagină cu ajutorul pachetului *pdf_render*:

```
// Open the PDF file
document = await PdfDocument.openFile(pdfPath);

for (int pageIndex = 0; pageIndex < document.pageCount; pageIndex++) {
  final page = await document.getPage(pageIndex + 1);

  // Render the page as an image at a higher resolution for better results
  final pageImage = await page.render(
    width: (page.width * 3).toInt(), // Increase width by 3x.
    height: (page.height * 3).toInt(), // Increase height by 3x.
  );

  final imagePath = await _saveImageTemporarily(pageImage);

  // ... preprocess the image and run OCR

  await File(imagePath).delete();
}
```

Operațiile de preprocesare sunt efectuate cu ajutorul librăriei *opencv_core*:

```
var mat = await cv.imreadAsync(imagePath);

// Step 1: Convert the image to grayscale
mat = await cv.cvtColorAsync(mat, cv.COLOR_RGBA2GRAY);

// Step 2: Apply Otsu's thresholding to binarize the image.
(_, mat) =
    await cv.thresholdAsync(mat, 0 /* thresh */ , 255 /* maxval */,
cv.THRESH_BINARY | cv.THRESH_OTSU);


// Step 3: Apply Gaussian blur to reduce noise.
mat = await cv.gaussianBlurAsync(mat, (3, 3) /* kernel size */, 0);
```

Ajustarea înclinației este cel mai complex pas din preprocesare. Funcția - *correctSkew* folosește operații morfologice pentru a detecta liniile orizontale și verticale, care sunt foarte comune în extrase de cont sub forma unor tabele. Pe baza acestora se calculează unghiul dominant de înclinare utilizându-se metoda Hough Transform, iar unghiul este folosit pentru rotirea corespunzătoare a imaginii. Pentru o metodă similară de detectare a înclinării documentelor puteți consulta lucrarea "A Document Skew Detection Method Using Fast Hough Transform". [8] Am inclus implementarea metodei *correctSkew* în Anexa 1.

După preprocesarea datelor se rulează Tesseract pe fiecare pagină în parte. Un alt pas care îmbunătățește acuratețea este setarea unei liste de caractere permise:

```
return await FlutterTesseractOcr.extractText(
  imagePath,
  language: 'ron', // Romanian language support
  args: {
    "psm": psmMode, // Page segmentation mode
    "oem": "1", // OCR Engine Mode (1: Neural nets LSTM-based)
    "tessedit_char_whitelist":
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789ăâîșțĂÂÎȘȚ.,
-/ ", // Allowed characters
  },
);
```

Postprocesarea se rulează pe textul întreg, stocat ca string. Metodele au fost alese după analiza outputului. Acestea includ:

- Eliminarea caracterelor ce se repetă de mai mult de 3 ori
- Eliminarea caracterelor singulare înconjurate de spații
- Eliminarea liniilor mai scurte de 5 caractere
- Eliminarea liniilor care conțin mai puțin de 20% caractere alfanumerice
- Eliminarea liniilor care conțin punctuație în procent mai mare de 50%
- Condensarea spațiilor și tab-urilor multiple într-un singur spațiu

Pe outputul final trebuie să rulăm un proces similar cu cel descris în secțiunea anterioară. Descrierile tranzacțiilor sunt greu de extras, dar sumele și datele tranzacțiilor sunt extrase cu o acuratețe de aproximativ 80% pentru fișierele strâmbe sau neclare.

Capitolul IV

Prezentarea interfeței

În cadrul acestui capitol voi itera prin toate ecranele/paginile aplicației și voi prezenta interacțiunile acsetora. De asemenea, voi evidenția tehnicile utilizate pentru îmbunătățirea experienței de utilizator.

4.1 Pagina de bun venit

Anumite date trebuie configurate înainte de începerea folosirii aplicației. Așadar, în cazul în care tabela *Config* este goală, este afișată pagina de bun venit. Aceasta include setarea unui nume de utilizator, monedei implicite, modificarea categoriilor de plăți și crearea unor conturi inițiale.

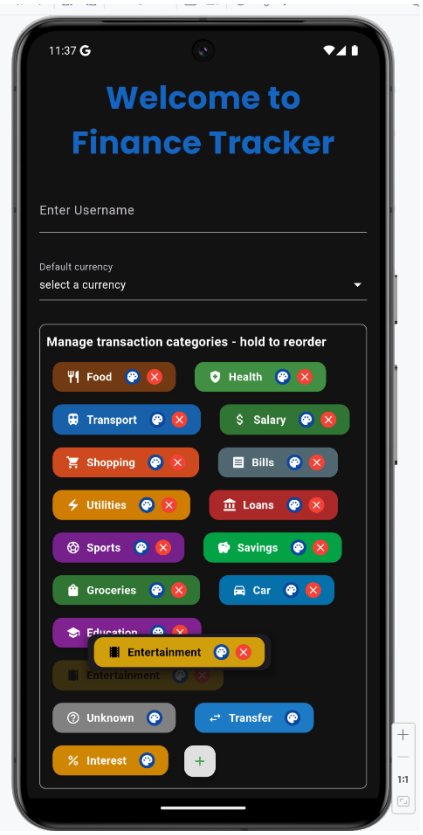

Figura 4.1 - Mesajul de bun venit și secțiunea de modificare a categoriilor

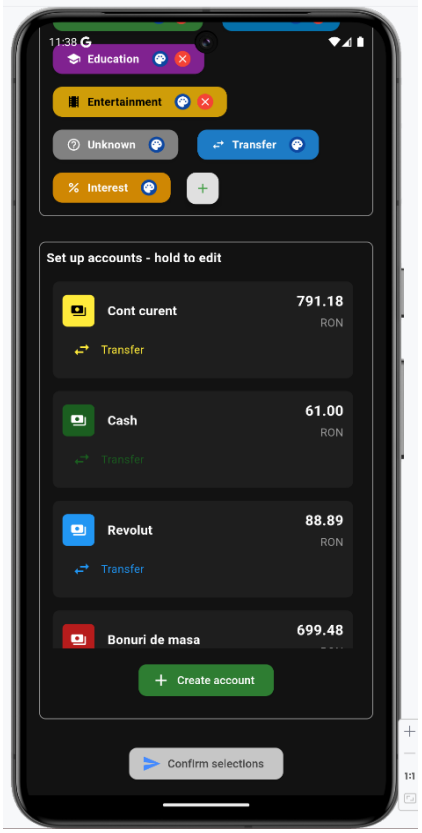

Figura 4.2 - Secțiunea de creare a conturilor și butonul de confirmare

Selecția monedei implicite se face cu ajutorul unui selector din pachetul *currency_picker* fiind date ca opțiuni monedele colectate în secțiunea 3.3. Categoriile pot fi reordonate prin metoda drag and drop, această ordine fiind păstrată în dialogul de adăugare a tranzacțiilor.

4.2 Meniul de tip sertar

Meniul de tip sertar (sau drawer) poate fi accesat prin apăsarea butonului aflat în colțul din dreapta sus și expune opțiuni precum modificarea categoriilor de plată, funcționalitatea de import/export, un dialog care să expună utilizatorului ratele de schimb valutar și opțiunea modificării monedei implicite (sau de referință).

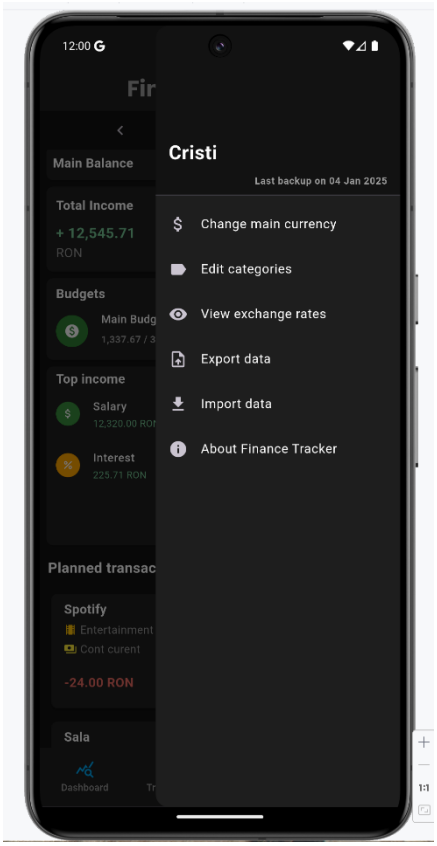

Figura 4.3 – Meniul de tip sertar

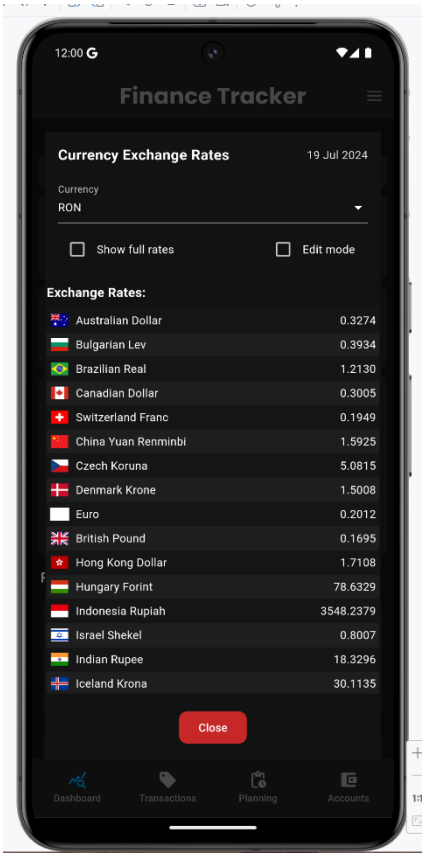

Figura 4.4 – Dialogul de curs valutar

4.3 Pagina cu conturi

Pagina cu conturi este accesată prin a 4-a opțiune din meniul inferior. Secțiunea superioară conține 2 totaluri ale soldurilor conturilor. Cel din stânga este bilanțul ce apare și în pagina principală și exclude conturile ce sunt marcate ca excluse, iar cel din dreapta include toate conturile. Secțiunea inferioară conține o listă derulabilă de conturi și depozite. Lista este reordonabilă, iar ordinea este păstrată și în dialogul de adăugare a tranzacțiilor. Fiecare cont afișează soldul și un buton de transfer. Depozitele afișează informații suplimentare, precum data lichidării și dobânda câștigată. Apăsarea scurtă pe un cont afișează o listă cu toate tranzacțiile acestora. Apăsarea lungă afișează un meniu cu toate opțiunile de modificare a acestuia.

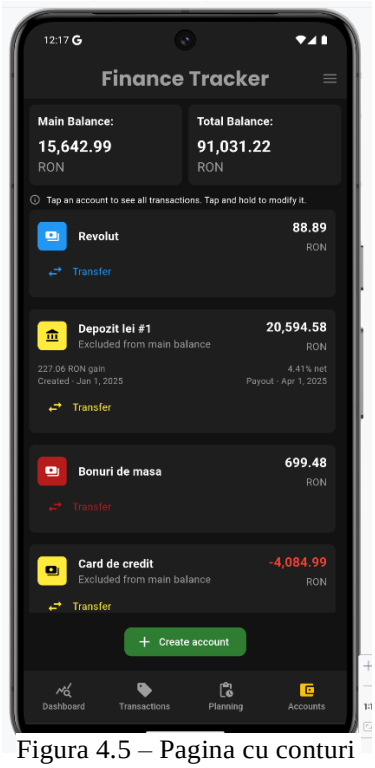

Figura 4.5 – Pagina cu conturi

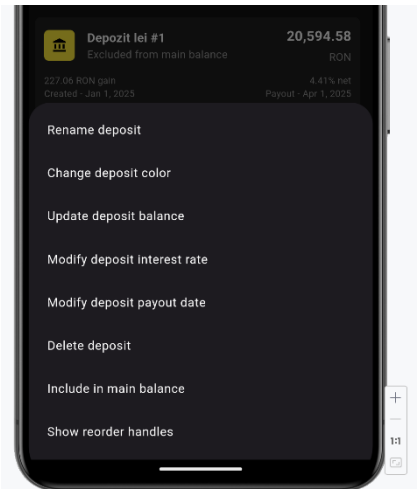

Figura 4.6 -Meniul de modificare a unui depozit

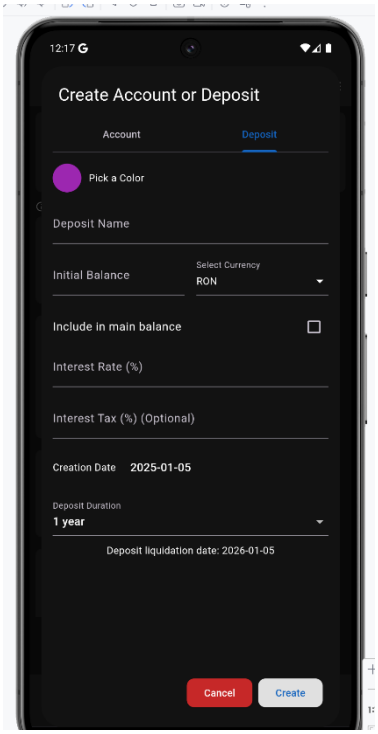

Figura 4.7 – Dialogul de creare a unui cont

4.4 Pagina de planificare

Pagina de planificare poate fi accesată prin al 3-lea buton de pe meniul inferior și este împărțită în 2 secțiuni, prima pentru plăți recurente și a doua pentru bugete. Plățile planificate sunt procesate din lista de tranzacții și pagina principală. Pagina de bugete nu arată și gradul acestora de folosire, informația fiind afișată pe pagina principală.

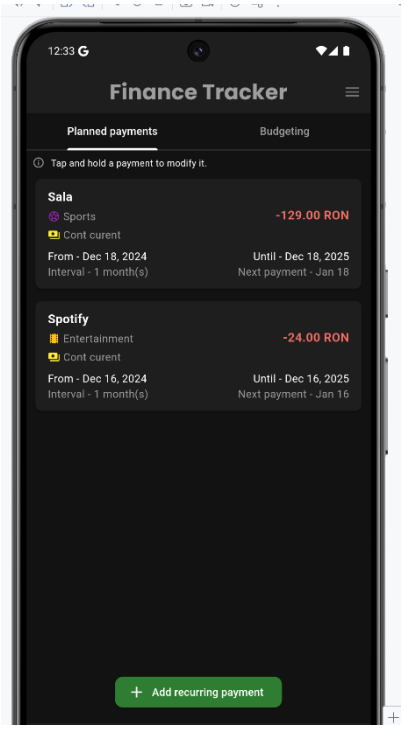

Figura 4.8 – Secțiunea de plăți planificate

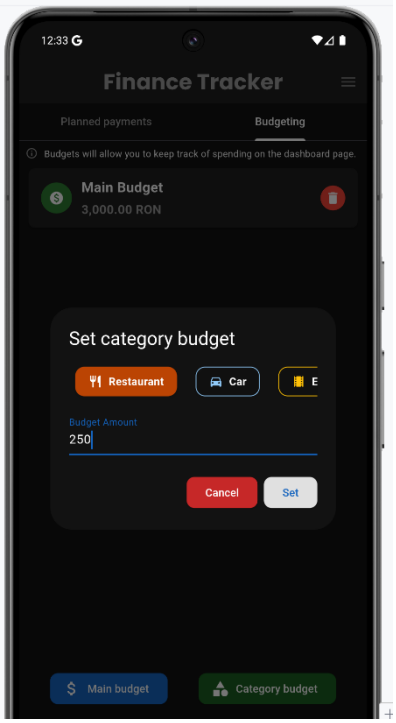

Figura 4.9 – Secțiunea de bugete

4.5 Lista de tranzacții

Lista de tranzacții poate fi accesată prin intermediul celui de-al doilea buton din meniul inferior. Aceasta este grupată pe zile. Pentru fiecare zi este calculat un total de încasări și cheltuieli. Prin apăsare lungă pe o tranzacție se deschide un meniu de unde aceasta poate fi ștearsă, modificată sau ascunsă. Tranzacțiile ascunse sunt marcate cu un ochi tăiat și sunt excluse atât din bilanțul zilnic cât și din statisticile din pagina principală.

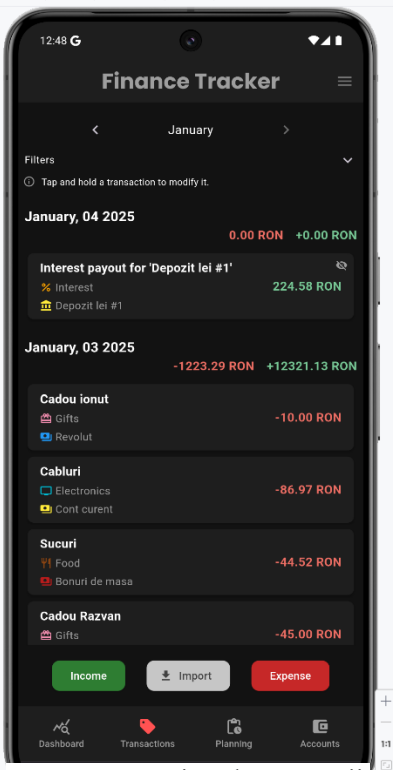

Figura 4.10 – Lista de tranzacții

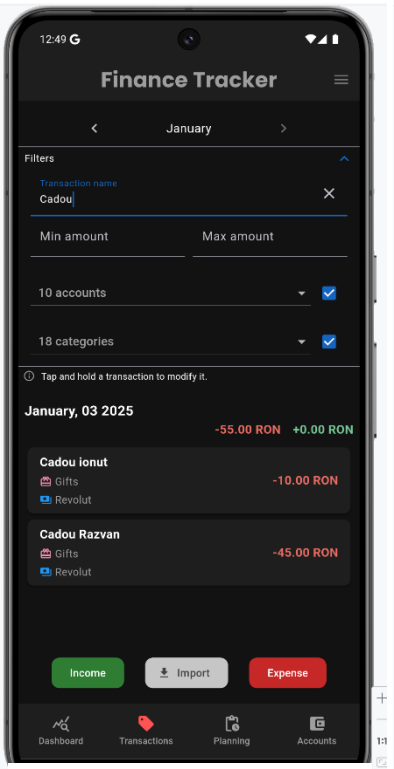

Figura 4.11 – Exemplu de filtre

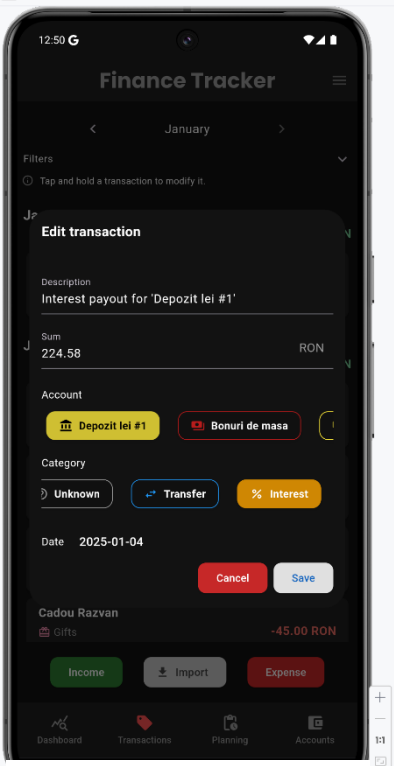

Figura 4.12 – Dialogul de editare

Filtrarea se poate face după nume, sumă, conturi și categorii. Aceste filtre afectează doar lista de tranzacții, sunt ascunse sub un evantai și pot fi extinse pentru aplicare (Figura 4.11). Filtrul de dată selectabil din secțiunea superioară a paginii afectează atât lista de tranzacții cât și pagina principală. Acesta este setat implicit pentru luna curentă (descris în secțiunea 3.4) și expune funcționalitate atât pentru schimbarea lunii prin utilizarea săgeților cât și pentru selectarea unui interval personalizat prin apăsarea pe numele lunii, acțiune care deschide un dialog separat.

Cele 3 butoane din partea inferioară permit adăugarea de tranzacții. Dialogurile pentru “Income” și “Expense” sunt similare cu cel din figura 4.12 și sunt separate pentru a fluidiza fluxul de adăugare a unei tranzacții, evitându-se nevoia de a adăuga un minus pentru fiecare tranzacție negativă. Butonul de import deschide un dialog ce permite selectarea unui format și încărcarea unui extras de cont (figura 4.13). În cazul în care

fișierul corect este încărcat și tranzacțiile sunt extrase, utilizatorul este redirecționat către dialogul de încărcare a listei de unde acesta le poate selecta și modifica (figura 4.14).

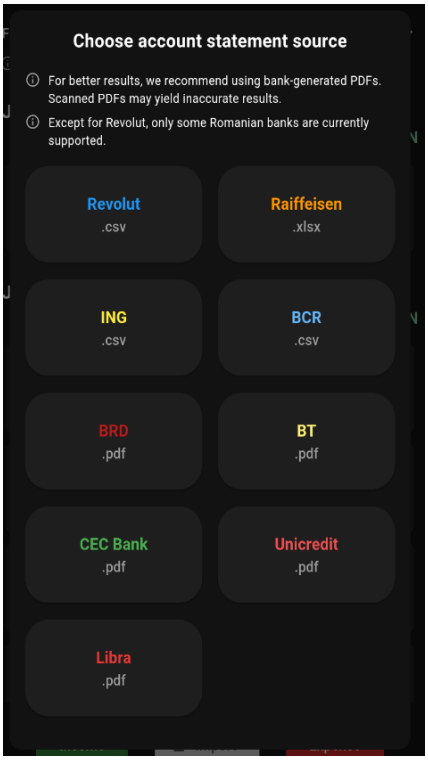

Figura 4.13 – Dialogul de selectare a metodei de importare

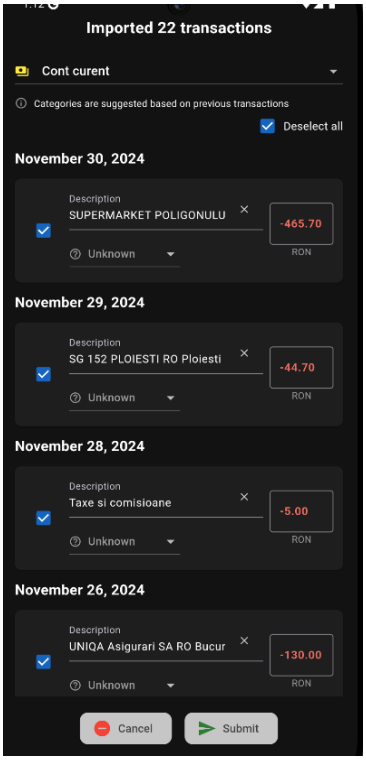

Figura 4.14 – Dialogul de revizuire a tranzacțiilor

4.6 Pagina principală

Pagina principală este opțiunea implicită din meniul inferior și conține o multitudine de informații utile pentru utilizator (Figura 4.15). Toate sumele sunt convertite automat la moneda implicită. Graficele din această secțiune sunt desenate cu ajutorul pachetului *fl_chart*.

Prima secțiune afișează bilanțul total (al conturilor incluse, descris în subcapitolul 4.3) și totalul încasărilor și cheltuielilor pe perioada selectată. Apăsarea scurtă pe încasări sau cheltuieli deschide un dialog ce afișează tranzacțiile procentual într-un piechart similar cu cel pentru categorii din Figura 4.16.

A doua secțiune conține informații despre gradul de utilizare a bugetelor, desenate cu ajutorul pachetului *flutter_fillbars*. Această secțiune se ajustează automat în cadrul selectării unui interval mai mare de o lună și afișează un gradient de la verde la roșu în funcție de apropierea procentului de 100%.

A treia secțiune afișează primele 3 categorii ordonate după suma totală, atât pentru încasări cât și pentru cheltuieli. Apăsarea scurtă pe una dintre ele deschide un dialog modal ce afișează o descompunere procentuală a categoriilor sub forma unui piechart (Figura 4.16).

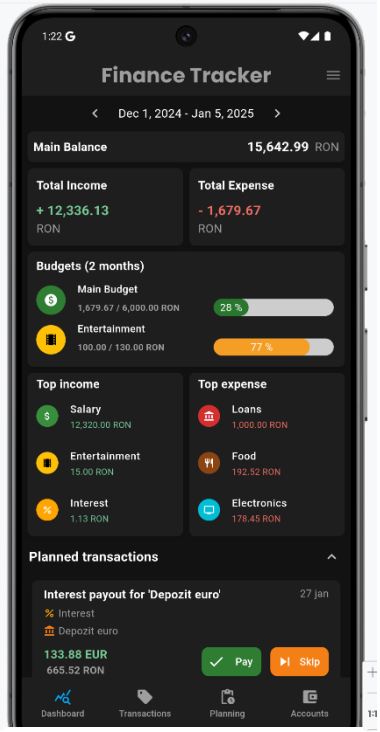

Figura 4.15 – Pagina principală

Figura 4.16 – Grafic de descompunere a cheltuielilor per categorie

A patra secțiune conține o listă scrollabilă de plăți planificate ce arată mereu plățile planificate din luna curentă, indiferent de filtrele selectate. Deoarece aceasta poate fi lungă în unele cazuri am făcut-o comprimabilă la apăsarea săgeții din dreapta titlului prin intermediul unui acordeon.

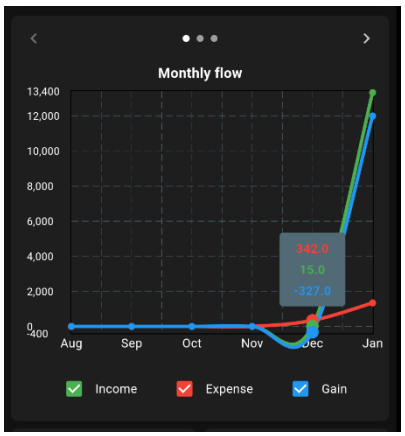

Figura 4.17 – Grafic pentru fluxul de valută

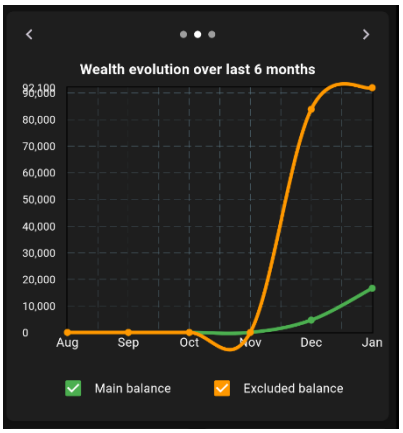

Figura 4.18 – Graficul balanțelor totale

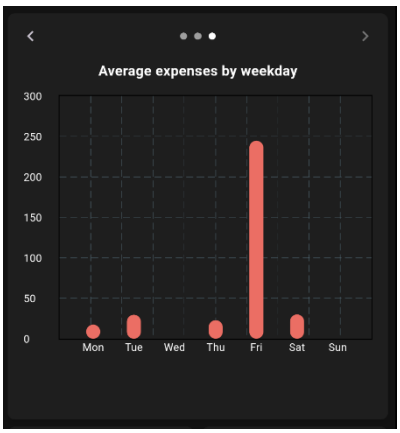

Figura 4.19 – Grafic pentru tipare de cheltuieli

A cincea secțiune conține un selector de grafice care ajută utilizatorul să afle mai multe despre tiparele sale de cheltuieli. Primul grafic (Figura 4.17) ilustrează fluxul de valută lunar pentru încasări (verde), cheltuieli (roșu) și diferența dintre acestea (albastru). Al doilea grafic (Figura 4.18) ilustrează evoluția bilanțului total pentru ultimele 6 luni, atât pentru bilanțul principal (verde) cât și pentru cel ce include toate conturile (galben), acesta fiind implicit ascuns. Din cauza unor limitări de implementare acest grafic nu se ajustează în funcție de filtrul pentru date și arată constant evoluția ultimelor 6 luni. Al treilea grafic (Figura 4.19) ilustrează cheltuielile medii pentru fiecare zi a săptămânii. Această informație poate ajuta anumiți utilizatori să identifice surse eliminabile de cheltuieli dacă există cheltuieli foarte mari într-o anumită perioadă a săptămânii (de exemplu, tinerii tind să cheltuie mai mulți bani în weekend decât in zilele lucrătoare).

Ultima secțiune conține o listă cu primele cele mai mari 10 cheltuieli din perioada selectată pe partea stângă, și cheltuielile totale folosite pe abonamente în intervalul selectat pe partea dreaptă. Aceste informații sunt relevante doar în cazuri rare și de aceea am ales să le afișez în finalul paginii.

4.7 Metode de îmbunătăţire a experienţei de utilizator

4.7.1 Eficientizarea fluxurilor

Am impus eliminarea pașilor inutili din fluxurile de adăugare a datelor. De exemplu, în cazul tranzacțiilor am separat inserarea de încasări și cheltuieli în două butoane pentru a expedia procesul prin eliminarea nevoii de a introduce semnul – pentru fiecare cheltuială. Acest pas este important pentru că majoritatea tranzacțiilor sunt, în general, cheltuieli.

Tot în fluxul de adăugare manuală a tranzacțiilor am adăugat integrarea cu tabela *Recent Descriptions,* formularul sugerând descrieri anterioare utilizatorului în funcție de categoria selectată. Lista este de asemenea filtrată în funcție de conținutul câmpului.

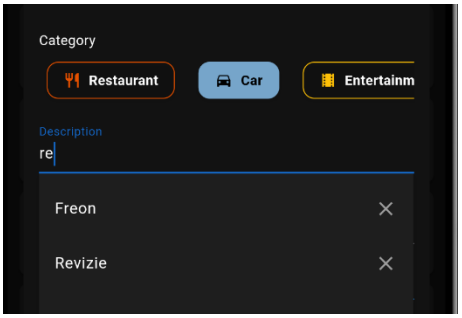

Figura 4.20 – Sugestii pentru categoria ‘Car’

Pentru a optimiza fluxul de adăugare automată a tranzacțiilor (din extrase de cont) am adăugat funcționalitatea de sugerare automată a categoriei, folosind tabela *LastTransactionCategory*. După importare fiecare tranzacție are o descriere preluată din extrasul de cont, această descriere fiind de obicei numele comerciantului. Probabilitatea ca utilizatorul să o schimbe pentru a reflecta descrierea reală este mare. Am ales salvarea descrierii inițiale sub numele de *vendorId* și folosirea acesteia pentru recomandarea unor categorii pentru tranzacțiile ce au același *vendorId*. Așadar, cu cât sunt importate mai multe tranzacții cu atât procesul lor de importare va deveni mai rapid.

O altă problemă reală pe care am încercat să o rezolv este importarea extraselor de cont suprapuse. În acest caz, dorim ca aplicația să excludă automat tranzacțiile deja adăugate. Am folosit ca metodă de detecție combinația de data tranzacției, suma și numele comerciantului menționat anterior. Folosind această metodă putem îmbunătăți semnificativ timpul de filtrare a tranzacțiilor pentru acest caz special.

4.7.2 Claritate

Vizualizarea distincției dintre tranzacții pozitive și negative poate fi obositoare când vorbim despre liste de tranzacții foarte lungi. Pentru a îmbunătăți ușurința de citire a listei am adăugat două culori distinctive folosite în întreaga aplicație, roșu deschis pentru cheltuieli și verde deschis pentru încasări. Am realizat ulterior că aceste culori nu sunt distinctive pentru utilizatorii daltoniști, impactând accesibilitatea.

În cazul acțiunilor ce nu sunt ușor de dedus am adăugat etichete cu informații în secțiunile relevante, similare cu cel din figura de mai jos.

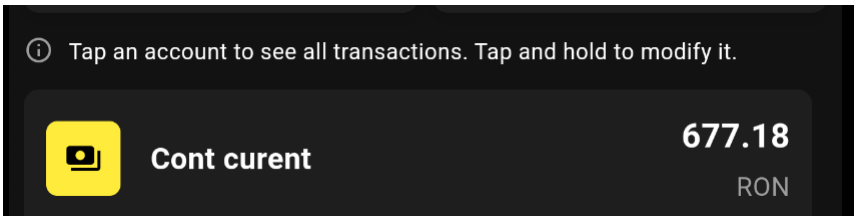

Figura 4.21 – Informarea utilizatorului despre funcționalități

4.7.3 Alinierea stilurilor

Pentru a menține un aspect vizual plăcut și a preveni eventualele confuzii am setat constrângeri pentru standardizarea stilului, care includ dar nu se limitează la:

- Câmpurile pentru introducerea datelor moștenesc stilul *Material Underlined Input*
- Butoanele din interfață extind același stil
- Fiecare bloc de conținut are același fundal, cu margini și rotunjiri ale colțurilor identice
- Dimensiunile și culorile fonturilor coincid între pagini, în funcție de stilul de heading

4.7.4 Feedback

Notificarea utilizatorului despre starea curentă a aplicației este una dintre cele mai complexe și importante părți ale domeniului de experiență a utilizatorului.

În cazul acțiunilor ce pot dura mult timp este important să notificăm utilizatorul ca aplicația este într-un proces de prelucrare a datelor și aceasta nu s-a blocat sau a înghețat. Pentru asta am implementat un dialog standardizat (Figura 4.21) ce previne utilizatorul din a-l închide și descrie acțiunile în desfășurare. În mod ideal, acest mesaj nu ar fi blocant și ar apărea într-o secțiune separată în timp ce utilizatorul poate continua folosi aplicația, dar unele acțiuni sunt constrânse să fie blocante de factori externi. De exemplu, în cazul încărcării tranzacțiilor, aplicația ar rula încet deoarece procesul consumă multe resurse de procesare.

În timpul utilizării aplicației pot apărea atât erori așteptate, cât și neanticipate. Pentru a preveni coruperea stării aplicației este esențială utilizarea blocurilor try/catch. Totuși, această acțiune nu este suficientă de una singură, fiind necesară și notificarea utilizatorului despre eroare. Un buton care nu generează nicio reacție sau un dialog care se închide fără a efectua acțiunea așteptată și fără a oferi un răspuns clar impactează experiența de utilizator în mod negativ. Așadar, am implementat un dialog standardizat care notifică utilizatorul despre eventualele erori (Figura 4.22).

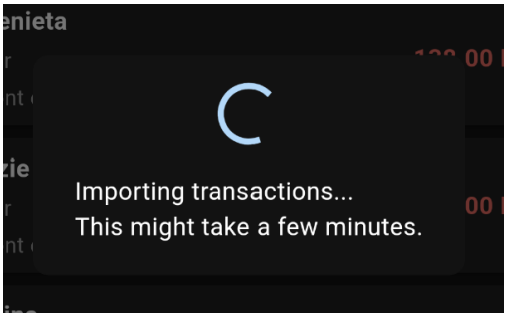

Figura 4.22 – Dialogul de încărcare

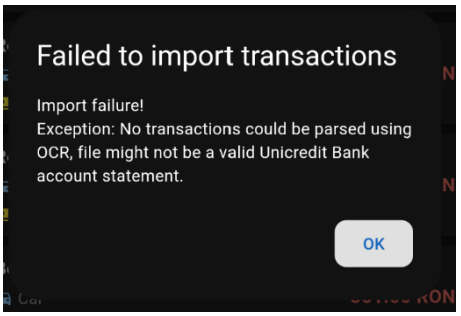

Figura 4.23 – Dialogul de notificare al erorilor

Capitolul V

Concluzii

Consider că aplicația descrisă în cadrul acestei lucrări rezolvă cu succes cerințele stabilite în faza de proiectare. Aceasta are un aspect modern și rulează fluid pe dispozitivele testate, îndeplinind toate necesitățile identificate în subcapitolul 2.1. Soluția oferită este una atractivă, oferind un pachet complet de funcționalități pentru gestionarea finanțelor personale.

Aplicația a fost testată cu seturi mari de tranzacții generate, menținându-și performanța și dovedind scalabilitate. Aceasta are un aspect potrivit atât pe telefoane cu ecran comun (19.5:9) cât și pe telefoane cu ecran îngust (22:9).

Framework-ul *Flutter* conține o multitudine de funcționalități ce ușurează dezvoltarea unei interfețe frumoase și rapide. Totuși, am identificat multiple erori la rulare ce nu sunt detectate de compilator, precum widget-uri ce trebuie forțate să se actualizeze manual și erori de așezare ce se întâmplă când un container *Expanded* este plasat într-un container fără constrângeri de mărime. Cu toate acestea, pot recomanda framework-ul ca fiind o opțiune *state of the art* în realizarea aplicațiilor de orice tip.

Metoda de importare a tranzacțiilor prin OCR este mai rapidă decât introducerea manuală, dar necesită atenție sporită din partea utilizatorilor. Deși acuratețea acesteia este de ~80% pentru identificarea tranzacțiilor în sine și a datei/sumei acestora este necesară o parcurgere paralelă a tranzacțiilor importate și extraselor de cont, pentru că nu toate tranzacțiile sunt detectate. Retrospectiv, ar fi trebuit să forțez încărcarea de extrase generate pentru a putea garanta acuratețea datelor.

Cel mai important aspect învățat pe parcursul realizării aplicației este importanța testării și feedback-ului de la utilizatori reali. Am început să folosesc aplicația pe telefonul personal și am descoperit multiple erori netratate, deși am testat manual riguros fiecare componentă implementată. De asemenea, după utilizarea îndelungată am observat anumite fluxuri ce pot fi îmbunătățite prin eliminarea unor acțiuni redundante.

Bibliografie

[1] Flutter - Build apps for any screen, https://flutter.dev (accesat la 31.12.2024)

[2] R. Rousselet, Riverpod, https://riverpod.dev (accesat la 31.12.2024)

[3] Setup - Drift, https://drift.simonbinder.eu/setup/ (accesat la 31.12.2024)

[4] Tesseract User Manual | tessdoc, https://tesseract-ocr.github.io/tessdoc/ (accesat la 04.01.2025)

[5] K. Chaudhary, B. Raghav, *EASTER: Efficient and Scalable Text Recognizer*, arXiv:2008.07839, August 2020, https://arxiv.org/abs/2008.07839

[6] https://frankfurter.dev (accesat la 03.01.2025)

[7] R. L. Kshetry, *Image preprocessing and modified adaptive thresholding for improving OCR*, arXiv:2111.14075. Noiembrie 2021, https://arxiv.org/abs/2111.14075

[8] P. Bezmaternykh, D. Nikolaev, *A Document Skew Detection Method Using Fast Hough Transform*, arxiv:1912.02504, Decembrie 2019, https://arxiv.org/abs/1912.02504

Anexa 1 – Funcția de corecție a înclinării unei pagini scanate

```
static Future<cv.Mat> _correctSkew(cv.Mat inputMat) async {
  const pi = math.pi;

  // Determine structuring element sizes for morphological operations
  const scale = 20; // Larger scale detects longer lines
  final horizontalSize = (inputMat.cols / scale).floor().clamp(1, 50);
  final verticalSize = (inputMat.rows / scale).floor().clamp(1, 50);

  // Define structuring elements for horizontal and vertical lines
  final horizontalStructure =
      cv.getStructuringElement(cv.MORPH_RECT, (horizontalSize, 1));
  final verticalStructure =
      cv.getStructuringElement(cv.MORPH_RECT, (1, verticalSize));

  // Extract horizontal lines using erosion and dilation
  cv.Mat horizLines = inputMat.clone();
  horizLines = await cv.erodeAsync(horizLines, horizontalStructure);
  horizLines = await cv.dilateAsync(horizLines, horizontalStructure);

  // Extract vertical lines using erosion and dilation
  cv.Mat vertLines = inputMat.clone();
  vertLines = await cv.erodeAsync(vertLines, verticalStructure);
  vertLines = await cv.dilateAsync(vertLines, verticalStructure);


  // Analyze horizontal and vertical lines
  final horizResult = await analyzeLines(horizLines);
  final vertResult = await analyzeLines(vertLines);

  // Calculate the dominant angles for horizontal and vertical lines
  final horizAngle = weightedMedianAngle(
      horizResult["angles"] as List<double>,
      horizResult["lengths"] as List<double>);
  final vertAngle = weightedMedianAngle(vertResult["angles"] as List<double>,
      vertResult["lengths"] as List<double>);

  // --- Determine the Final Skew Angle ---
  double? finalAngle;
  double horizTotalLength = (horizResult["lengths"] as List<double>).isEmpty
      ? 0
      : (horizResult["lengths"] as List<double>).reduce((a, b) => a + b);
  double vertTotalLength = (vertResult["lengths"] as List<double>).isEmpty
      ? 0
      : (vertResult["lengths"] as List<double>).reduce((a, b) => a + b);

  // Use the angle set with the longer total line length for skew correction
  if (horizTotalLength > vertTotalLength && horizAngle != null) {
    finalAngle = horizAngle;
  } else if (vertAngle != null) {
    finalAngle = vertAngle;
  }

  // If no significant skew angle is found, return the original image
  if (finalAngle == null || finalAngle.abs() * 180 / pi < 0.5) {
    return inputMat;
  }

  // Convert the skew angle to degrees
  final angleInDegrees = finalAngle * 180 / pi;

  // --- Apply Rotation to Correct Skew ---
  final center = cv.Point2f(inputMat.cols / 2, inputMat.rows / 2);
  final rotationMatrix = cv.getRotationMatrix2D(center, angleInDegrees, 1.0);

  // Rotate the image to correct skew
  cv.Mat corrected =
      cv.warpAffine(inputMat, rotationMatrix, (inputMat.cols, inputMat.rows));

  return corrected;
}
```

```
// Analyze lines to compute dominant angles using Hough Transform
Future<Map<String, dynamic>> analyzeLines(cv.Mat lineMask) async {
  // Detect edges using the Canny algorithm.
  final edges = await cv.cannyAsync(lineMask, 50, 150);

  // Detect lines using the probabilistic Hough Transform
  const rho = 1.0; // Distance resolution in pixels
  const theta = pi / 180; // Angle resolution in radians
  const threshold = 80; // Minimum votes for a line
  double minLineLength = 50; // Minimum line length in pixels
  double maxLineGap = 20; // Maximum gap between line segments

  final houghLines = await cv.HoughLinesPAsync(edges, rho, theta, threshold,
      minLineLength: minLineLength, maxLineGap: maxLineGap);

  List<double> angles = [];
  List<double> lengths = [];

  if (houghLines.rows > 0) {
    final rawData = houghLines.data;
    final bd = rawData.buffer.asByteData();

    for (int i = 0; i < houghLines.rows; i++) {
      // Calculate line length; simplified to make the function fit on one page
      final length = length between line points;

      // Ignore very short lines.
      if (length < minLineLength) continue;

      // Calculate the angle of the line
      final angle = math.atan2(dy, dx);

      // Normalize the angle to the range [-pi/2, pi/2]
      double normalizedAngle = angle;
      while (normalizedAngle > pi / 2) {
        normalizedAngle -= pi;
      }
      while (normalizedAngle <= -pi / 2) {
        normalizedAngle += pi;
      }

      angles.add(normalizedAngle);
      lengths.add(length);
    }
  }

  return {"angles": angles, "lengths": lengths};
}

// --- Compute the Weighted Median Angle ---
double? weightedMedianAngle(List<double> angles, List<double> lengths) {
  if (angles.isEmpty) return null;

  // Calculate the total length of all lines
  final totalLength = lengths.reduce((a, b) => a + b);

  // Create pairs of angles and lengths for sorting
  final angleLengthPairs =
      List.generate(angles.length, (i) => [angles[i], lengths[i]]);
  angleLengthPairs.sort((a, b) => a[0].compareTo(b[0]));

  double cumulative = 0.0;
  for (var pair in angleLengthPairs) {
    cumulative += pair[1];
    if (cumulative >= totalLength / 2) {
      return pair[0];
    }
  }
  return null;
}
```