1. Scurtă descriere a proiectului

Aplicația dezvoltată este o platformă web care permite **încărcarea și procesarea imaginilor de documente de identitate**, având ca scop **extragerea automată a informațiilor textuale** folosind tehnici de OCR (Optical Character Recognition).

Sistemul este construit pe o arhitectură **client–server**, unde backend-ul este responsabil de logica de business, securitate, procesarea imaginilor și gestionarea evenimentelor interne, iar frontend-ul oferă o interfață web modernă pentru interacțiunea cu utilizatorul.

Pe lângă funcționalitatea principală de procesare a imaginilor, aplicația include un **sistem de notificări in-app**, utilizat pentru informarea utilizatorului asupra evenimentelor importante (de exemplu, finalizarea procesării unui document).

Sistemul de notificări este implementat folosind un mecanism de tip **event-driven**, bazat pe **Observer Pattern**, unde evenimentele sunt publicate din logica de business și tratate de listener-e dedicate. Notificările sunt persistate în baza de date și pot fi ulterior afișate în interfața utilizator.

Aplicația expune un **API REST securizat**, care permite:

- autentificarea utilizatorilor;
- upload-ul imaginilor;
- procesarea acestora prin OCR;
- gestionarea și afișarea notificărilor in-app.

2. Limbaje şi tehnologii folosite

2.1 Limbaje şi Framework-uri Backend (Core)

Java (versiunea 17 / 21)

Java este limbajul de programare principal utilizat pentru implementarea logicii de backend. Acesta oferă stabilitate, performanță și suport extins pentru aplicații enterprise.

Spring Boot

Spring Boot este framework-ul principal folosit pentru dezvoltarea aplicației backend. Acesta facilitează:

- configurarea rapidă a aplicației;
- gestionarea componentelor;
- expunerea API-urilor REST prin adnotări precum @RestController.

Spring Web

Modulul Spring Web este utilizat pentru:

- definirea endpoint-urilor HTTP (GET, POST);
- gestionarea request-urilor și response-urilor;
- upload-ul de fișiere folosind MultipartFile.

Spring Security

Spring Security este folosit pentru securizarea aplicației, gestionând:

- autentificarea utilizatorilor;
- autorizarea accesului la resursele protejate;
- protecția endpoint-urilor API.

2.2 Procesare imagini şi OCR (Feature principal)

Tess4J (wrapper pentru Tesseract OCR)

Tess4J este biblioteca utilizată pentru recunoașterea optică a caracterelor, permițând extragerea textului din imaginile încărcate de utilizator.

Java AWT & ImageIO

Librăriile standard Java AWT și ImageIO sunt folosite pentru **pre-procesarea imaginilor**, incluzând:

- redimensionare;
- rotire;
- binarizare;
- mascarea zonelor sensibile (ex. fața), înainte de rularea OCR-ului.

Apache Commons IO

Apache Commons IO este utilizată pentru manipularea fișierelor, facilitând lucrul cu:

- extensii de fișiere;
- căi de fișiere;
- operații de bază asupra fișierelor (ex. FilenameUtils).

2.3 Gestionare date și dependențe

Maven

Maven este folosit ca unealtă de build automation și pentru gestionarea dependențelor aplicației, toate librăriile fiind definite în fișierul pom.xml.

Lombok

Lombok este utilizat pentru reducerea codului boilerplate, prin generarea automată de:

- getter-e și setter-e;
- constructori;
- metode uzuale, folosind adnotări precum @Data.

H2 Database / PostgreSQL

Baza de date este utilizată pentru stocarea informațiilor despre utilizatori (în cazul implementării autentificării persistente).
H2 poate fi folosită pentru dezvoltare și testare, iar PostgreSQL pentru rulare în medii de producție.

2.4 Unelte de dezvoltare și testare

Postman

Postman este utilizat pentru testarea manuală a endpoint-urilor API, în special pentru:

- trimiterea request-urilor POST;
- testarea upload-ului de imagini;
- verificarea răspunsurilor backend-ului.

Visual Studio Code / IntelliJ IDEA

Aceste medii de dezvoltare (IDE-uri) sunt folosite pentru scrierea, testarea și depanarea codului backend.

2.5 Tehnologii Frontend (Interfața utilizator)

React

React este biblioteca principală utilizată pentru dezvoltarea interfeței utilizator. Aplicația este construită folosind **componente reutilizabile**, permițând o structură clară și actualizarea dinamică a interfeței pe baza stării aplicației.

Vite

Vite este unealta de build și development folosită pentru frontend, oferind:

- pornire rapidă a serverului de dezvoltare;
- hot-reload instant;
- suport modern pentru aplicații React.

Tailwind CSS

Tailwind CSS este framework-ul CSS utilizat pentru stilizarea aplicației, permițând:

- realizarea unui design modern și responsive;
- menținerea consistenței vizuale;
- dezvoltare rapidă folosind clase utilitare.

JavaScript (ES6+)

JavaScript este limbajul utilizat pentru implementarea logicii de frontend, incluzând:

- gestionarea evenimentelor utilizatorului;
- controlul stării aplicației;
- interacțiunea cu componentele React.

Fetch API

Fetch API este folosit pentru comunicarea dintre frontend și backend, realizând:

- cereri de autentificare;
- upload-ul imaginilor către backend;
- preluarea răspunsurilor procesate (text OCR, status-uri).

HTML5 & JSX

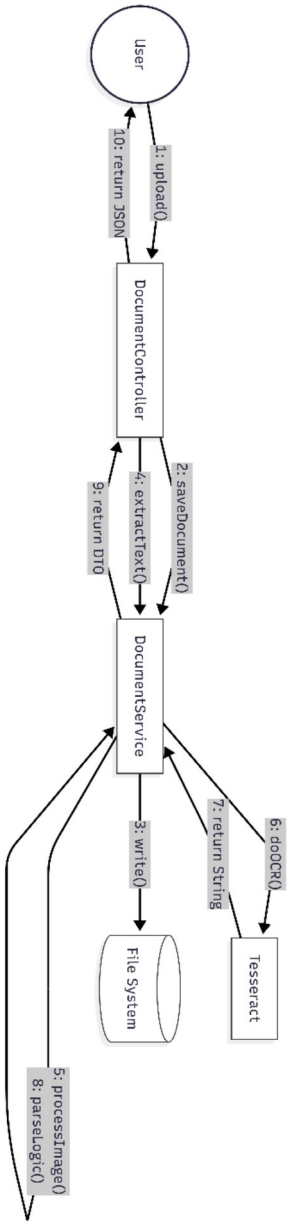
Structura paginilor este realizată folosind HTML5 și JSX, o extensie de sintaxă care permite combinarea logicii JavaScript cu structura vizuală a aplicației.

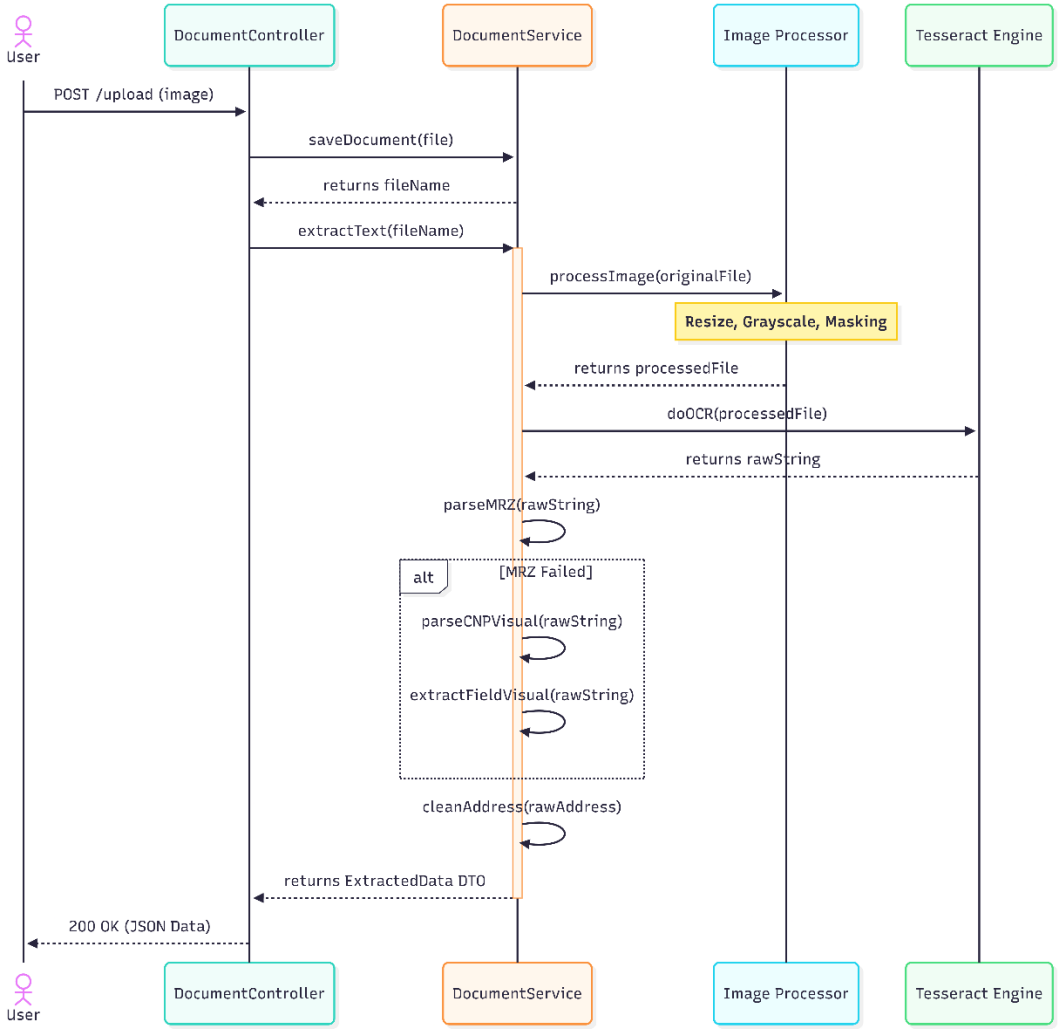
Instrumente frontend

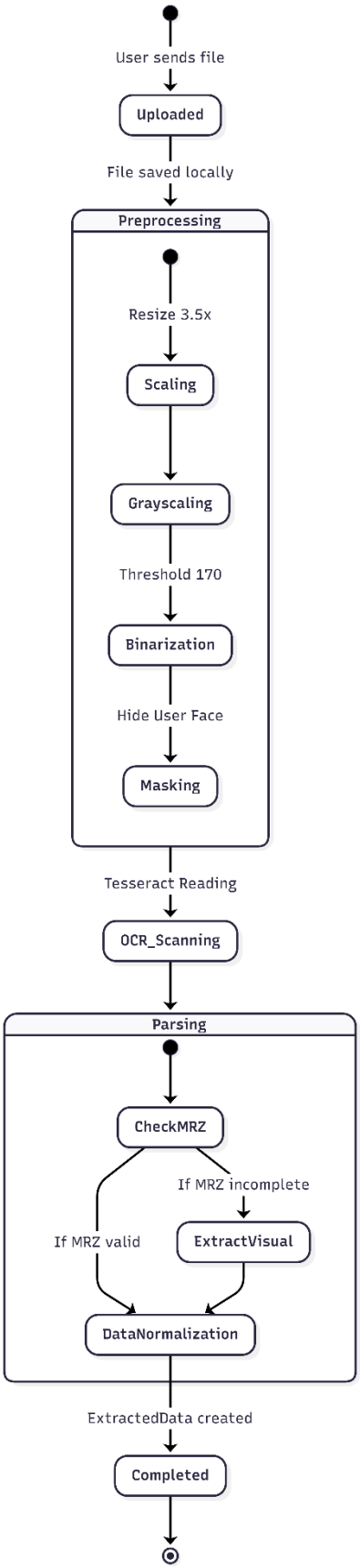
- **Node.js & npm** – pentru rularea aplicației și gestionarea dependențelor frontend;
- **Visual Studio Code** – IDE utilizat pentru dezvoltarea interfeței;
- **Browser Developer Tools** – pentru debug și testarea interfeței.

3. Diagrame

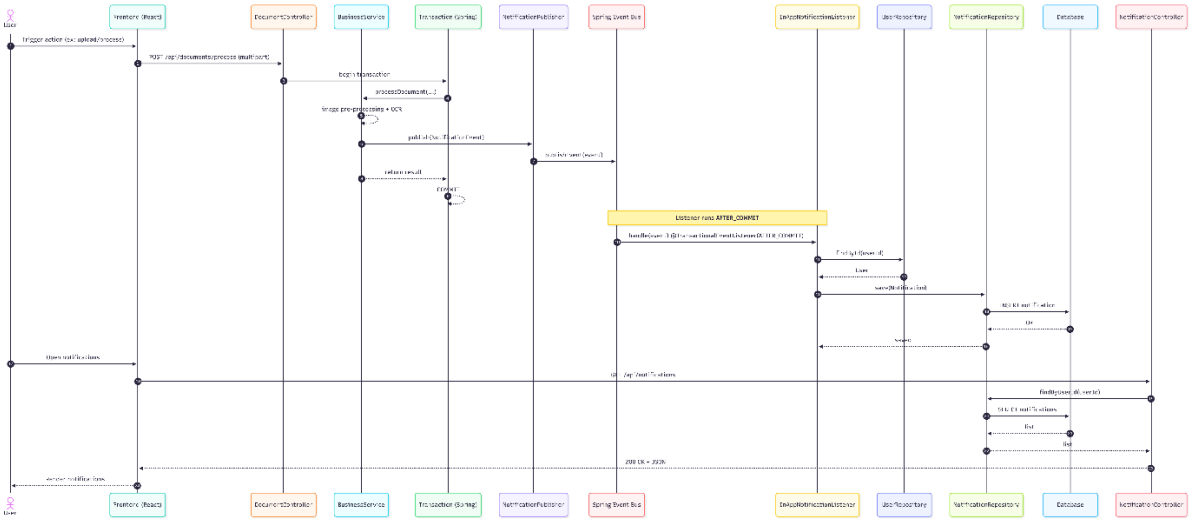
a. Balint Alex

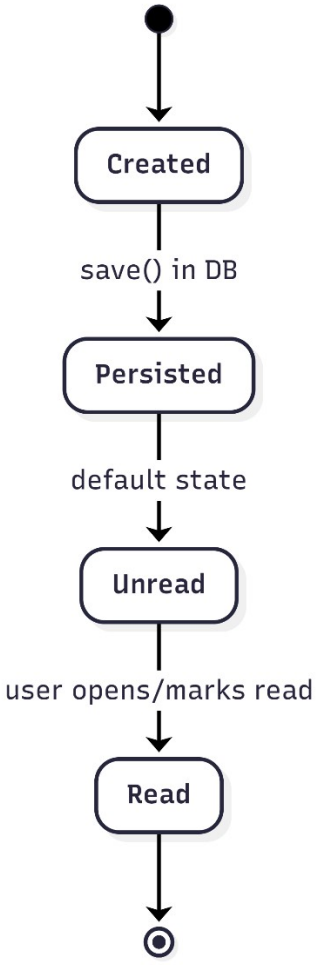



User
DocumentController
DocumentService
Image Processor
Tesseract Engine
POST /upload (image)
saveDocument(file)
returns fileName
extractText(fileName)
processImage(originalFile)
Resize, Grayscale, Masking
returns processedFile
doOCR(processedFile)
returns rawString
parseMRZ(rawString)
alt
[MRZ Failed]
parseCNPVisual(rawString)
extractFieldVisual(rawString)
cleanAddress(rawAddress)
returns ExtractedData DTO
200 OK (JSON Data)
User
DocumentController
DocumentService
Image Processor
Tesseract Engine

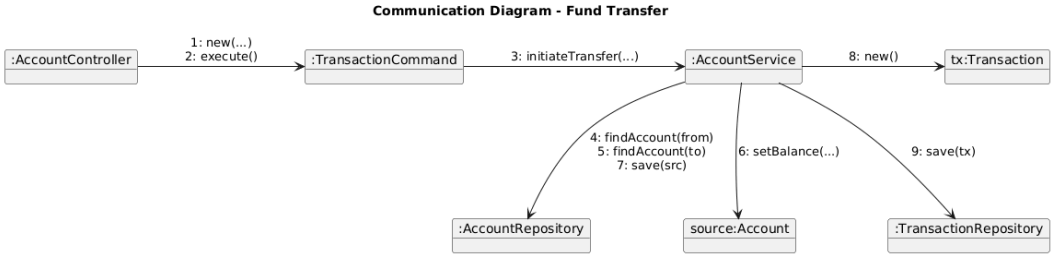

User sends file
Uploaded
File saved locally
Preprocessing
Resize 3.5x
Scaling
Grayscaling
Threshold 170
Binarization
Hide User Face
Masking
Tesseract Reading
OCR_Scanning
Parsing
CheckMRZ
If MRZ incomplete
If MRZ valid
ExtractVisual
DataNormalization
ExtractedData created
Completed

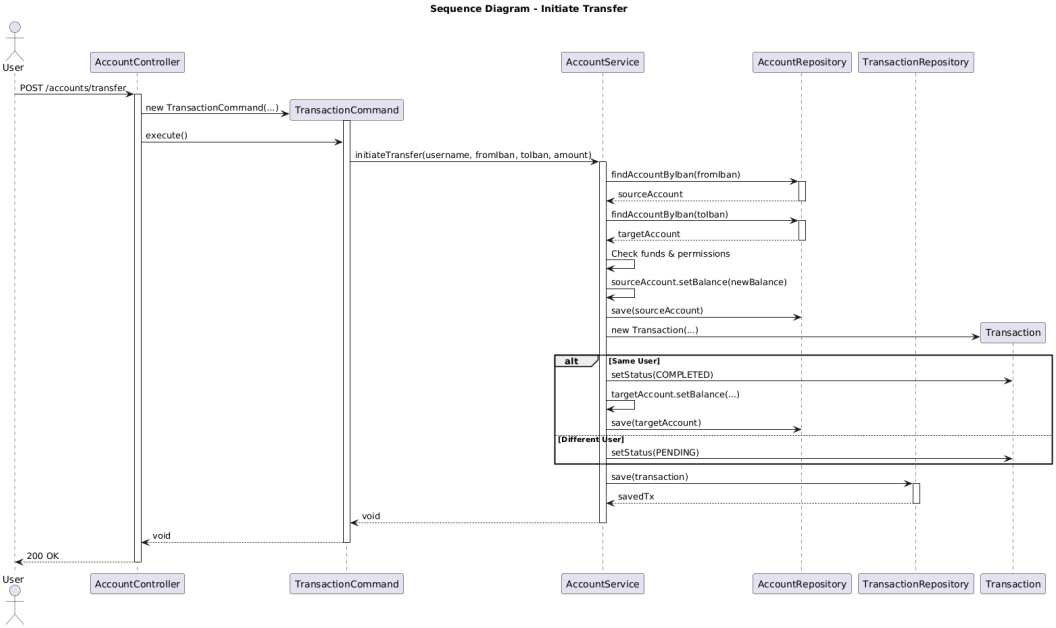
b. Berar Alexandra

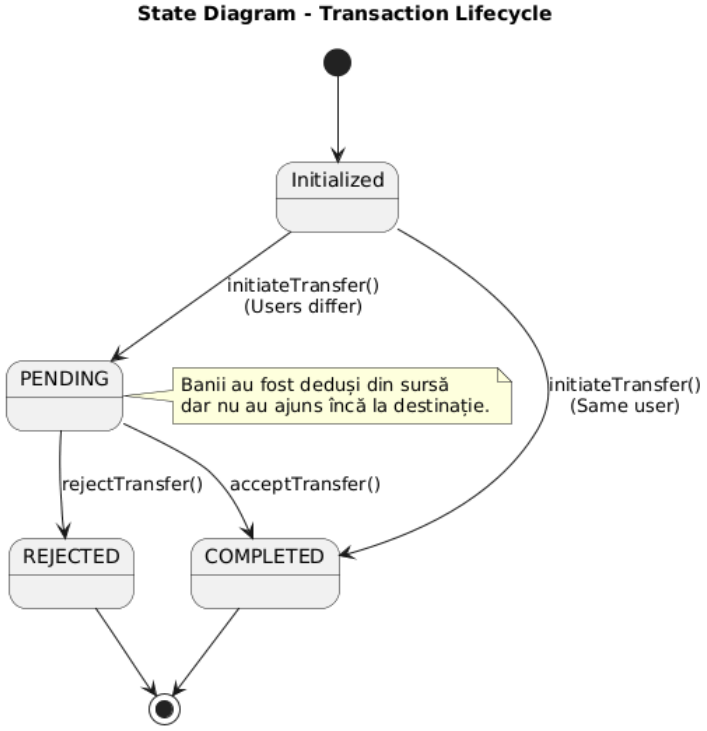

Listener runs AFTER_COMMIT


Created
save() in DB
Persisted
default state
Unread
user opens/marks read
Read

c. Afrăsinei Şerban







4. Design Pattern

a. Balint Alex

Problema Identificata:

Problema principala e ca DocumentController-ul trebuie sa extraga date structurate dintr-o imagine uploadata, dar treaba asta e destul de complexa tehnic. Avem de-a face cu o gramada de subsisteme: operatii de I/O pe disc, procesare de imagine (rotiri, scalari, binarizare), interactiunea cu Tesseract (care e o librarie externa de C++) si o logica de parsing destul de stufoasa (Regex pentru MRZ vs Regex vizual, plus data cleaning).

Daca am fi pus toata logica asta direct in Controller, codul ar fi iesit un "spaghetti code" imposibil de citit, foarte greu de testat si extrem de "tightly coupled" de detaliile de implementare (gen procesarea pixelilor).

Solutia (Design Pattern):

Am rezolvat asta folosind Facade Pattern prin clasa DocumentService. Practic, clasa functioneaza ca o "fatada" care abstractizeaza si ascunde toata complexitatea din spate. Pe Controller nu-l intereseaza (si nici nu trebuie sa stie) ca noi rotim imaginea cu -90 de grade sau ca folosim un threshold de 170 la binarizare. Controller-ul apeleaza o singura metoda simpla: extractText(fileName).

Detalii Implementare:

DocumentService este Fatada propriu-zisa. Metoda extractText e punctul de intrare (entry point). Ea doar dirijaza apelurile catre metodele private din spate (processImage, parseMRZ, cleanAddress), comportandu-se ca un wrapper care ia input-ul brut (numele fisierului) si returneaza un rezultat curat (obiectul ExtractedData).

b. Berar Alexandra

Problema identificată

Aplicația trebuie să genereze notificări in-app atunci când apar anumite evenimente în sistem (ex: finalizarea procesării OCR, finalizarea unei acțiuni importante etc.).
O implementare directă, în care serviciul principal creează și salvează notificarea în mod explicit, duce la:

- cuplare puternică între logica de business și modulul de notificări;
- cod mai greu de întreținut (orice schimbare în notificări afectează serviciile principale);
- dificultate în extindere (dacă ulterior se adaugă notificări prin email / push, ar trebui modificată logica de business peste tot).

Soluția oferită de Observer Pattern

Observer Pattern rezolvă această problemă printr-un mecanism publish–subscribe:

- componenta care detectează evenimentul (publisher) publică un eveniment fără să știe cine îl tratează;
- una sau mai multe componente (observers/listeners) se abonează și reacționează la eveniment, executând acțiunile necesare.

Avantajele obținute:

- decuplare între business logic și notificări;
- notificările pot fi extinse prin adăugarea de noi listeners, fără modificarea codului existent;
- crește claritatea arhitecturii (evenimentele devin puncte standard de integrare).

Detalii de implementare

Implementarea este realizată folosind mecanismul de evenimente din Spring:

- Publisher: NotificationPublisher publică un NotificationEvent în momentul în care are loc o acțiune relevantă.
- Event: NotificationEvent transportă informația minimă necesară (ex: userId, message, type, createdAt).
- Observer/Listener: InAppNotificationListener tratează evenimentul și creează o entitate Notification care este persistată în baza de date prin NotificationRepository.

c. Afrăsinei Şerban

1. Problema identificată în proiect

În dezvoltarea componentei de tranzacţii a proiectului Banking Management System, o dificultate majoră a fost cuplarea strânsă (tight coupling) dintre `AccountController` (stratul care primeşte cererile HTTP) şi logica complexă de business din `AccountService`. Fără o structură clară, controller-ul risca să devină aglomerat cu detalii specifice fiecărui tip de operaţiune bancară (validări de parametri, selecţia metodei corecte din service), ceea ce făcea codul rigid şi greu de testat. De exemplu, simpla adăugare a unui nou tip de operaţiune ar fi necesitat modificarea codului existent în controller, încălcând principiile de "clean code" şi făcând dificilă gestionarea uniformă a erorilor sau a fluxului de execuţie pentru diversele acţiuni ale utilizatorului (depuneri, retrageri, transferuri).

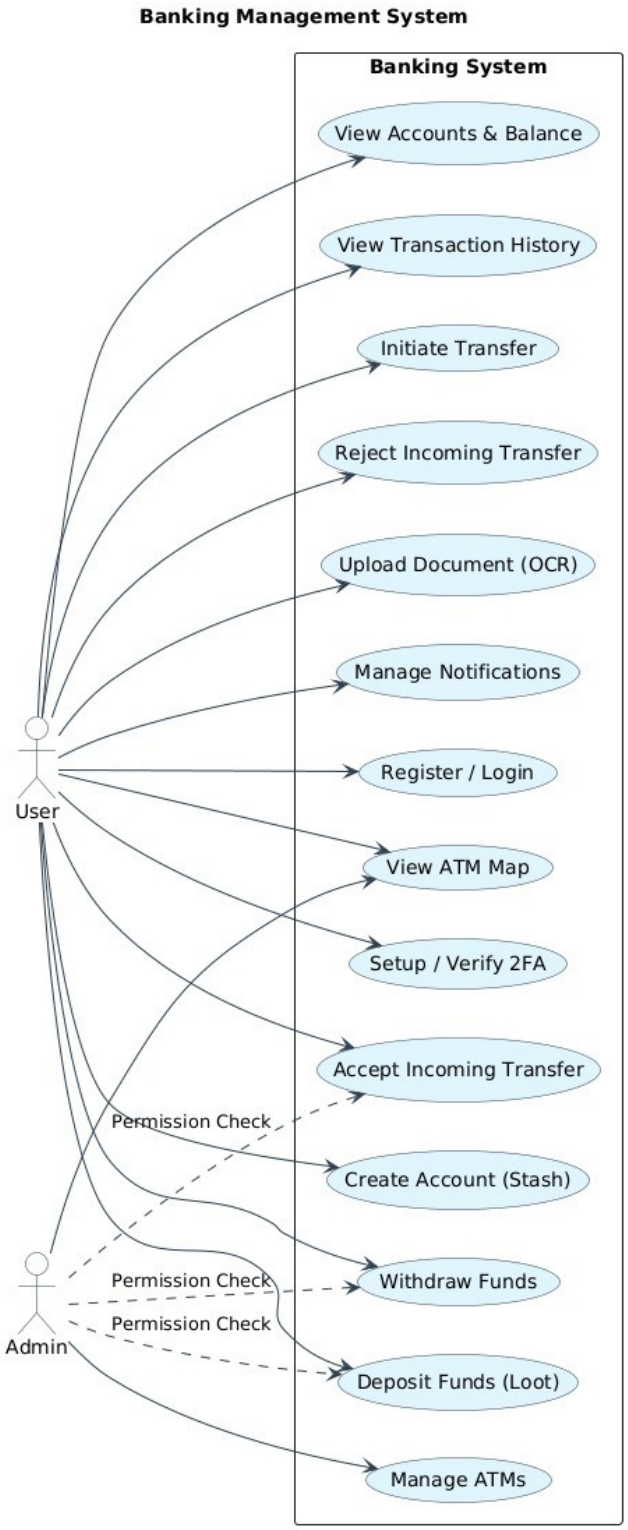
2. Soluţia oferită de Command Pattern

Command Pattern rezolvă aceste probleme prin încapsularea unei cereri (request) într-un obiect de sine stătător, separând astfel responsabilitatea emiterii comenzii de responsabilitatea executării ei. În loc ca `AccountController` să apeleze direct şi explicit metodele din `AccountService`, el creează şi manipulează obiecte de tip "Command" care conţin toate informaţiile necesare pentru a efectua acţiunea. Această abordare permite tratarea uniformă a tuturor operaţiunilor bancare printr-o interfaţă comună, oferind flexibilitate şi extensibilitate; noi comenzi pot fi adăugate oricând fără a schimba codul clientului (controller-ul), iar logica de execuţie rămâne izolată în clasele specifice.

3. Detalii de implementare

Implementarea concretă se bazează pe interfaţa `BankCommand`, care defineşte metoda contract `execute()`. Clasele concrete `DepositCommand`, `WithdrawCommand` şi `TransactionCommand` implementează această interfaţă şi acţionează ca intermediari: ele stochează starea necesară execuţiei (IBAN, sumă, username) şi referinţa către `AccountService` (Receiver-ul care deţine logica reală). În `AccountController`, atunci când un endpoint este apelat (de exemplu `/deposit`), se instanţiază comanda corespunzătoare (`new DepositCommand(...)`) şi se apelează metoda `execute()`, care la rândul ei declanşează metoda `depositFunds` din service.

5. Cerințe funcționale și diagrama use-case



Înregistrare și autentificare utilizator

Sistemul trebuie să permită utilizatorilor crearea unui cont și autentificarea în aplicație folosind credențiale valide. Accesul la funcționalitățile sistemului este permis doar utilizatorilor autentificați.

Vizualizarea conturilor și a soldului

Sistemul trebuie să permită utilizatorului vizualizarea conturilor bancare asociate și a soldului curent pentru fiecare cont.

Inițierea unui transfer

Sistemul trebuie să permită utilizatorului inițierea unui transfer de fonduri către un alt cont, prin specificarea sumei și a contului destinatar.

Depunerea de fonduri

Sistemul trebuie să permită utilizatorului depunerea de fonduri într-un cont bancar, cu actualizarea soldului.

Retragerea de fonduri

Sistemul trebuie să permită utilizatorului retragerea unei sume de bani dintr-un cont, doar în condițiile existenței unui sold suficient.

Crearea unui cont bancar

Sistemul trebuie să permită utilizatorului crearea unui nou cont bancar, care va fi asociat utilizatorului autentificat.

Încărcarea documentelor pentru procesare OCR

Sistemul trebuie să permită utilizatorului încărcarea imaginilor de documente de identitate pentru procesare OCR, cu validarea formatului fișierului.

Procesarea documentelor și extragerea textului

Sistemul trebuie să preproceseze imaginile încărcate și să extragă automat informațiile textuale folosind tehnici de OCR, returnând rezultatul către utilizator.

Gestionarea notificărilor

Sistemul trebuie să genereze notificări in-app pentru evenimente relevante (de exemplu operații bancare) și să permită utilizatorului vizualizarea acestora.

Configurarea și verificarea autentificării în doi pași (2FA)

Sistemul trebuie să permită utilizatorului configurarea și utilizarea autentificării în doi pași pentru acțiuni sensibile.

Vizualizarea hărţii ATM-urilor

Sistemul trebuie să permită utilizatorului vizualizarea locațiilor ATM-urilor disponibile pe o hartă interactivă.

5. Cerinţe non-funcţionale

Performanţă

Sistemul trebuie să ofere timpi de răspuns adecvați pentru o experiență optimă a utilizatorului:

- operațiile uzuale (autentificare, vizualizare conturi, notificări) trebuie procesate într-un timp rezonabil;
- operațiile costisitoare (procesarea OCR) pot avea un timp de execuție mai mare, fără a bloca funcționarea generală a aplicației;
- sistemul de notificări trebuie să funcționeze asincron.

Fiabilitate şi consistenţă

Sistemul trebuie să mențină consistența datelor și corectitudinea operațiilor:

- tranzacțiile financiare trebuie finalizate complet sau anulate;
- notificările trebuie generate doar pentru operații finalizate cu succes;
- datele bancare (solduri, tranzacții) trebuie să fie consistente în orice moment.

Utilizabilitate

Aplicația trebuie să fie ușor de utilizat de către utilizatorii finali:

- interfața trebuie să fie intuitivă și clar structurată;
- utilizatorul trebuie să primească feedback vizual pentru acțiunile efectuate;
- mesajele de eroare trebuie să fie clare și ușor de înțeles.

6. READ ME

Cerinţe preliminare

Pentru rularea aplicației este necesară instalarea următoarelor componente:

- **Docker Desktop**, utilizat pentru rularea bazei de date PostgreSQL;
- **Java JDK** (versiunea 25);
- **Node.js** (versiunea 18 sau mai recentă);

- un mediu de dezvoltare (ex: IntelliJ IDEA, Visual Studio Code).

Pornirea bazei de date

Aplicația utilizează o bază de date **PostgreSQL**, rulată într-un container Docker.
În directorul rădăcină al proiectului se execută comanda:

docker-compose up -d

Această comandă pornește containerul bazei de date în fundal.
Conexiunea la baza de date este configurată automat în backend.

Pornirea aplicației backend

Aplicația backend este implementată folosind **Spring Boot** și poate fi pornită fie din IDE, fie din linia de comandă.

Pornire din linia de comandă:

./mvnw spring-boot:run

Backend-ul va rula implicit pe portul:

- **http://localhost:8081**

Pornirea aplicației frontend

Frontend-ul este implementat folosind **React**, **Vite** și **Tailwind CSS**.
Pentru pornirea acestuia, se deschide un terminal nou și se execută comenzile:

cd front_banking

npm install

npm run dev

Aplicația frontend va fi disponibilă la adresa:

- **http://localhost:5173**