

UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAŞI  
FACULTATEA DE INFORMATICĂ



LUCRARE DE LICENȚĂ

**PocketIDE**

propusă de

**Cosmin Aftanase**

**Sesiunea:** iulie, 2021

Coordonator științific

**Asist. drd. Prelipcean Bogdan**

UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAŞI  
FACULTATEA DE INFORMATICĂ

# **PocketIDE**

**Cosmin Aftanase**

**Sesiunea:** *iulie, 2021*

Coordonator științific

**Asist. drd. Prelipcean Bogdan**

Avizat,

Îndrumător Lucrare de Licență

Titlul, Numele și prenumele \_\_\_\_\_

Data \_\_\_\_\_ Semnătura \_\_\_\_\_

**DECLARAȚIE privind originalitatea conținutului lucrării de licență**

Subsemnatul(a).....

domiciliul în .....  
născut(ă) la data de ....., identificat prin  
CNP,....., absolvent(a) al(a) Universității „Alexandru Ioan  
Cuza” din Iași, Facultatea de ..... specializarea  
....., promoția ....., declar pe  
propria răspundere, cunoscând consecințele falsului în declarații în sensul art.  
326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011  
art.143 al. 4 si 5 referitoare la plagiat, că lucrarea de licență cu  
titlul:\_\_\_\_\_ elaborată sub îndrumarea dl. /d-na  
\_\_\_\_\_, pe care urmează să o susțină în fața  
comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consumând inclusiv la introducerea conținutului său într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice în vederea facilitării fasificării de către cumpărător a calității de autor al unei lucrări de licență, de diploma sau de disertație și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rezultatul cercetării pe care am întreprins-o.

Dată azi, .....

Semnătură student .....

## DECLARAȚIE DE CONSUMÂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „*PocketIDE*”, codul sursă al programelor și celealte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Iași, *data*

Absolvent *Aftanase Cosmin*

---

(semnătura în original)

# Cuprins

<b>Introducere</b>	<b>5</b>
1.1. Context	5
1.2. Motivație	6
1.3. Contribuții	7
<b>Descrierea problemei</b>	<b>8</b>
<b>Abordări anterioare</b>	<b>9</b>
<b>Descrierea soluției</b>	<b>13</b>
4.1. Dezvoltarea și analiza cerințelor	13
4.1.1. Diagramă Use Case	13
4.1.2. Diagramă de Flux	15
4.2. Proiectarea arhitecturală	16
4.2.1. Tehnologii folosite	16
Cota de piață, căutarea unui software stack potrivit	16
Probleme cu React Native	18
Migrarea spre Aplicație Web Progresivă	19
4.2.2. Model de dezvoltare și arhitectură	20
Test Driven Development	20
Arhitectură bazată pe microservicii	22
4.2.3. Design backend & frontend	23
Model C4	23
Design Figma	28
4.3. Detalii implementare	30
4.3.1. TextReco - microserviciu de recunoștere a scrisului	30
Încercări proprii	30
Servicii Cloud	32
4.3.2. CodeFinder - microserviciu de alegere a recunoașterii corecte	35
4.3.3. Microservicii de compilare și rulare a codului	37
4.3.4. API gateway și middleware	38
4.3.5. Front End	41
<b>Concluziile lucrării</b>	<b>42</b>

# **Introducere**

## **1.1. Context**

PocketIDE este o aplicație web progresivă (PWA), care după cum sugerează și numele, a fost gândită în mod special pentru a fi folosită pe telefoanele mobile, având ca funcționalitate principală scrierea și salvarea codului, asemenea unui editor cum ar fi Visual Studio Code sau Sublime. Aceasta ca și noutate oferă utilizatorului posibilitatea de a desena pe un canvas sau a face poză la un text ce se dorește a fi recunoscut și transformat în cod. De asemenea, pe lângă scrierea de cod, aplicația permite utilizatorului și compilarea și rularea acestuia, întorcându-i atât afișările de ecran, cât și erorile și unde anume în cod se produc acestea.

Fiind o aplicație web progresivă, aceasta este multiplatformă, având interfață specială atât pentru desktop, cât și pentru telefon și sunt păstrate în cloud și în sincronizare toate modificările aduse de pe oricare platformă. De asemenea, alte beneficii ale PWA-ului sunt faptul ca aplicația poate fi descărcată și folosită offline, dar în acest caz va avea acuratețe mai scăzută în ceea ce privește partea de recunoaștere a scrisului și nu vor mai fi salvate în cloud și sincronizate datele, decât până la revenirea conexiunii la internet.

Așadar ca elemente de noutate față de alte compilatoare online, sunt faptul că PocketIDE se folosește de OCR pentru a facilita scrierea codului, este multiplatformă, descărcabilă și funcționează offline. Pe lângă acestea, aplicația oferă suport pentru mai multe tipuri de limbaje, printre care se enumeră: C, C++, Java, Haskell, Javascript, PHP, Python și Scala.

## 1.2. Motivație

Problema și soluțiile implementate au fost alese din mai multe puncte de vedere, raportate atât la problemele din comunitatea locală, cât și la dorința de dezvoltare personală și îmbogățire a cunoștințelor.

Ca și problemă pe care o rezolvă, aplicația este orientată în mod special către rezolvarea problemelor din educație.

În școli și facultăți la orele de informatică, algoritmii sunt de multe ori scriși de mană pe hârtie și pe tablă, iar după copiați manual pe calculator pentru a putea fi compilați, rulați și verificați. Acest proces în care codul este copiat de două ori, pe hârtie, apoi pe calculator, poate fi astfel eficientizat prin utilizarea OCR-ului și rularea codului direct din aplicație.

O altă problemă pe care o rezolvă PocketIDE estea cea a scrierii și testării codului, atunci când nu ai la dispoziție niciun dispozitiv de tip desktop. Folosirea unui compilator online și a tastaturii de pe telefon este o sarcină care necesită mult timp și răbdare. Codul poate fi scris astfel mult mai repede și ușor, folosindu-se un pix și un caiet sau folosindu-se un stilou de tip Stylus<sup>[1]</sup> pentru a scrie direct pe telefon.



Fig. 1: Stilou de tip Stylus [1]

Pe de altă parte, sunt pasionat de noile tehnologii și doresc să mă dezvolt pe această parte de web & software development. În facultate am lucrat doar cu monoliți și am dorit să experimentez crearea unei aplicații de tip microservicii.

De asemenea, am dorit să învăț mai multe despre standardele industriei și tehnologiile de ultimă generație și să creez un proiect modern, care să fie inovativ prin tema și ideile abordate.

### **1.3. Contribuții**

În realizarea acestui proiect am avut mai multe încercări de realizare a aplicației și abordări diferite de soluționare a problemei.

Ca și parte concretă ce se regăsește în produsul final, contribuția mea este interfața web, cele 10 microservicii ce se ocupă cu compilarea și rularea codului, serviciul pentru gestionarea utilizatorilor și proiectelor acestora, API Gateway-ul (implementat de mine, fără framework-uri), microserviciul ce apelează serviciile de OCR de la Azure Form Recognizer, OCR.space și Google Cloud Vision și microserviciul ce încearcă să calculeze, folosind un API de identificare a limbajului de programare, textul care se potrivește cel mai bine cu poza transmisă. De asemenea, alte contribuții sunt unit testele și integrarea tuturor componentelor în aplicația finală (toate microserviciile sunt hostate online, folosind platforma Heroku).

Pe partea teoretică sau care nu se regăsește în produsul final, contribuția mea a fost în mare parte legată de familiarizarea cu domeniile și tehnologiile complet noi.

Ca și domenii noi, au fost încercări, documentate, pe partea de machine learning, atât pentru partea de OCR, cât și pentru partea de identificare a limbajului de programare. În urma experimentelor, am fost nevoit să înlocuiesc codul și modelul creat de mine pentru OCR cu API-uri de la Google, Microsoft și OCR.space pentru a avea rezultate mai precise. De asemenea am întâmpinat dificultăți în obținerea de seturi de date pentru ambele programe de machine learning. Pentru cel de detectare al limbajului de programare, am creat un scrapper de pagini pe github, dar din mai multe motive, care apar menționate mai jos, a fost înlocuit și acesta cu un API extern.

Aplicația era de la început îndreptată către termenii OCR, multi-platformă și microservicii aşa că am încercat și experimentat mai multe tehnologii noi. Ca și cunoștințe de programare am intrat în contact cu limbaje și framework-uri noi, cum ar fi: React, React Native și Docker.

## Descrierea problemei

Se dorește realizarea unei aplicații care să ajute în învățământ, să permită rularea rapidă și directă a codului scris cu pixul pe hârtie sau cu creta pe tablă, fără a fi nevoie ca acesta să fie copiat manual la tastatură.

De asemenea, se dorește o modalitate inovativă de a scrie și rula cod mai ușor de pe telefon, pentru cazurile când utilizatorul nu are echipament de tip desktop la dispoziție.

Soluția acestei probleme trebuie să fie portabilă, intuitivă și ușor de folosit. Pentru acest lucru, aplicația ar trebui să fie în mod special destinată telefoanelor mobile, dacă nu chiar multi-platformă și să funcționeze atât pe mobile, cât și desktop.

Ca și funcționalități, aceasta ar trebui să aibă în primul rând un sistem de recunoaștere optică a caracterelor (OCR), fiind orientat în mod special spre recunoașterea scrisului de mână. Dacă e posibil, pentru a nu fi restricționați de conexiunea la internet, acest sistem ar fi de preferat să funcționeze și offline, chiar și cu o acuratețe mai scăzută.

Întrucât recunoașterea nu va fi mereu sătă la sătă precisă, va trebui să îi fie oferită utilizatorului posibilitatea de editare a textului recunoscut.

Totodată aplicația trebuie să permită utilizatorilor să ruleze și să compileze cod pentru mai multe tipuri de limbaje, ținta principală fiind cele care predomină în învățământul preuniversitar: C, C++ și Python.

Se dorește ca și această funcționalitate să poată fi folosită offline, deși aici rămâne de investigat dacă acest lucru este posibil pe telefoane mobile sau dacă este nevoie neapărat de un server spre care să se trimită date.

De asemenea, dacă nu este posibilă rularea offline, ar fi de preferat să existe o opțiune de salvare locală a datelor, astfel încât acestea să poată fi trimise din nou atunci când se restabilește conexiunea la internet.

O ultimă funcționalitate, ar fi abilitatea de a salva codul și rezultatul acestuia și de a putea să se share și să se țină în sincronizare datele de pe desktop și mobile.

## Abordări anterioare

La momentul scrierii documentației, din propria documentare, nu există niciun fel de aplicație care să aibă toate funcționalitățile prezentate mai sus sau care să conțină în același timp părțile principale, de recunoaștere a scrisului din poze și de rulare și compilare a codului identificat.

În schimb o aplicație cu spirit asemănător, care a fost folosită ca sursă de inspirație pentru proiect, este photomath. După cum se poate observa în Figura 1 de mai jos, photomath permite utilizatorilor să scanzeze formule matematice scrise pe hârtie și să le calculeze automat folosind aplicația. Pe lângă rezultat, photomath generează și afișează după scanare textul identificat și le oferă clientilor posibilitatea de a-l edita atât pentru cazul în care au apărut erori la partea de recunoaștere a scrisului, cât și pentru posibilitatea de a face schimbări și a experimenta cu valori diferite. Aceste două idei au avut un impact puternic asupra dezvoltării proiectului, regăsindu-se și în versiunea finală a PocketIDE-ului.

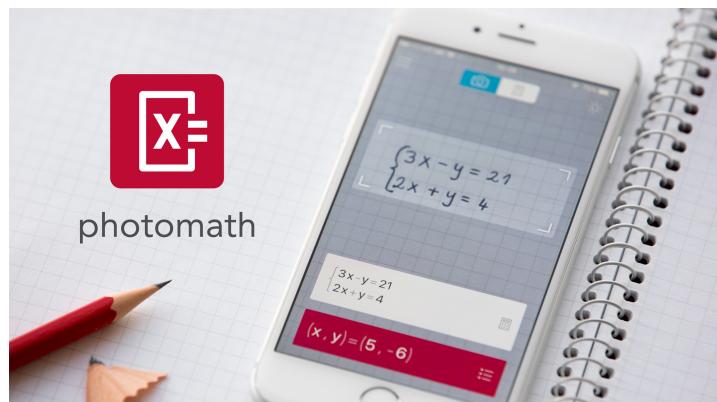


Fig. 1: photomath [2]

O altă funcționalitate interesantă a photomath, care se poate observa în Figura 2 de mai jos, este afișarea tuturor pașilor de rezolvare a problemei respective. Această idee a fost de asemenea inspirație pentru PocketIDE, ca pe lângă încercarea de rulare și afișare a unui rezultat, să se arate în caz de eroare une anume în cod se produce aceasta.

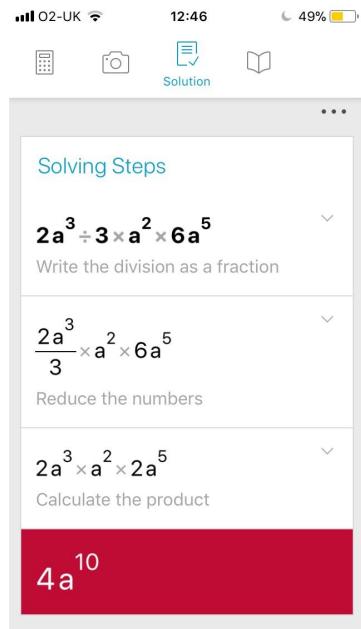


Fig. 2: Etape de rezolvare problemă, photomath [3]

Pe lângă photomath, alte aplicații asemănătoare se împart în două categorii: aplicații care utilizează recunoaștere optică a caracterelor și aplicații care compilează cod.

La partea de recunoaștere, nu sunt foarte multe lucruri de adăugat. După cum se poate vedea în Figura 3 de mai jos, există foarte multe aplicații de acest fel pe Google App Store. Acestea susțin faptul că problema noastră poate într-adevăr să aibă o soluție pentru platforma mobile.

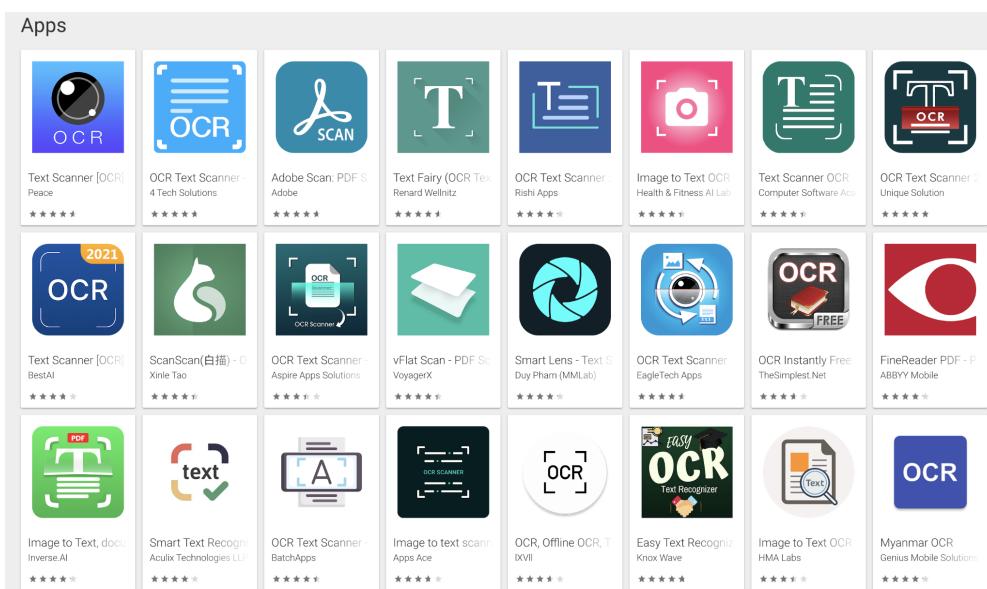


Figura 3: poze aplicații cu recunoaștere a scrisului, Google App Store

Din resursele aflate la îndemână, am aflat că această recunoaștere a pozelor este posibilă fie local folosindu-se libraria Tesseract, fie cu ajutorul unui SDK sau a unui API call. Toate variantele au fost luate în calcul în realizarea proiectului.

Deși un detaliu important de menționat este că dintre metodele descrise mai sus, cea care folosește Tesseract este singura care poate fi folosită dacă ne dorim cu adevărat să avem recunoaștere offline. Doar că Tesseract vine la rândul lui cu o restricție și anume faptul că motorul acestuia este scris în C++. După cum zice articolul <https://golb.hplar.ch/2019/07/ocr-with-tesseractjs.html>: "... Motorul Tesseract este scris în C++ și nu rulează într-un browser. Singura modalitate de utilizare a motorului C++ este prin trimitera imaginii dintr-o aplicație web către un server, rularea acesteia prin motor și trimiterea textului înapoi. Dar de câțiva ani, există un port JavaScript al motorului Tesseract C++, care rulează într-un browser și nu depinde de niciun cod de pe server. Biblioteca se numește Tesseract.js..." [4]

Așadar, folosindu-se Tesseract JS se poate folosi o variantă Javascript pentru recunoaștere, doar că singura condiție ar fi că aceasta poate fi rulată doar din browser sau dintr-un mediu de lucru cu Javascript, cum ar fi NodeJS. Există de asemenea o librărie creată pentru Android numită tesseract-android-tools.

Ca și aplicații care rulează cod pe telefon, există pe Google App Store și Apple Store mai multe care fac acest lucru după cum se poate vedea în Figura 4: compilatoare C++, Google Play Store și Figura 5: compilatoare Java, Google Play Store.

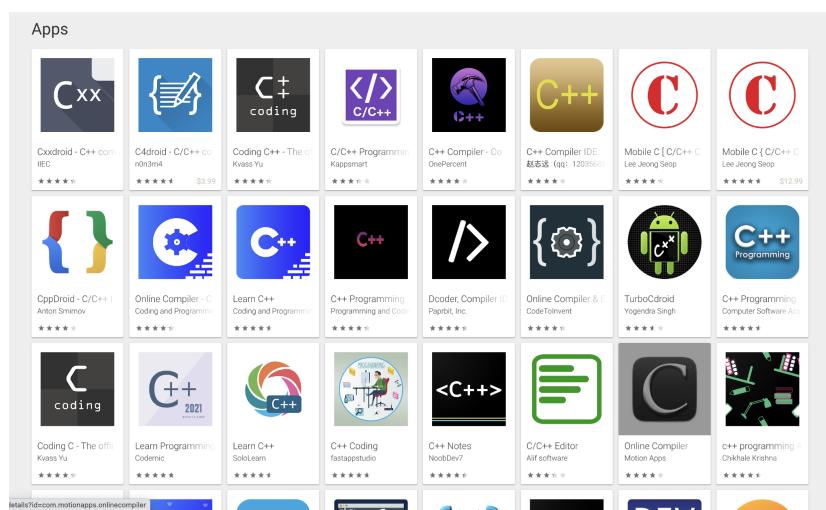


Fig. 4: compilatoare C++, Google Play Store

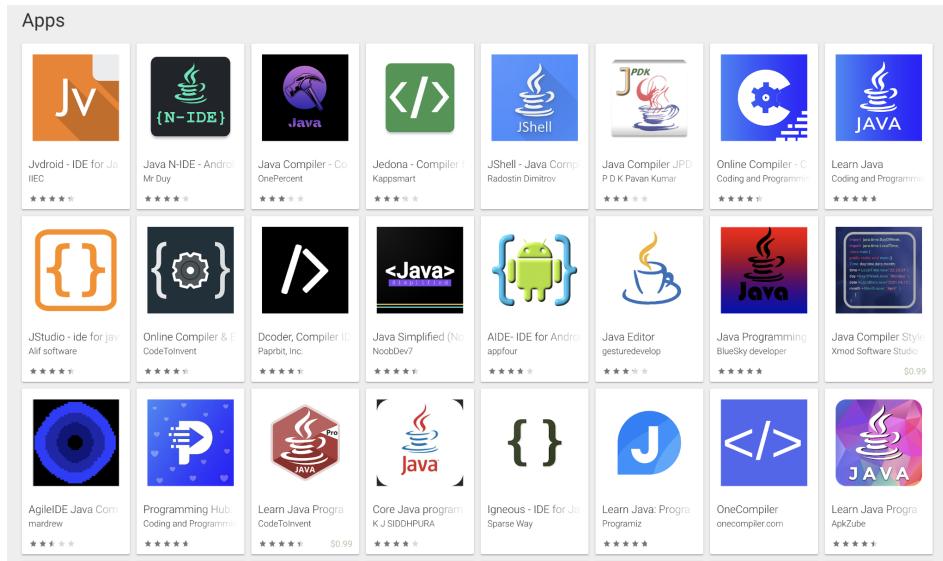


Fig 5. compilatoare Java, Google Play Store

Doar că aceste aplicații fac acest lucru într-o manieră singulară și independentă în ceea ce privește funcționalitatea. Mai exact, există compilatoare offline pe telefon atât pentru Java, cât și pentru C++, dar nu există compilatoare care să le facă pe ambele.

Deși compilarea și rularea offline este realizabilă din punct de vedere tehnic, rămâne de investigat dacă acest lucru este eficient din puncte de vedere al resurselor de memorare și de procesare, întrucât dorim să avem mai multe tipuri de limbaje.

De asemenea rămâne de investigat o abordare pentru aceste compilatoare, astfel încât proiectul să respecte principiile SOLID și să fie ușor de extins și menținut.

# Descrierea soluției

## 4.1. Dezvoltarea și analiza cerințelor

Pentru a putea începe dezvoltarea aplicației, a fost nevoie de o întocmire clară a cerințelor funcționale și nonfuncționale ale aplicației.

### 4.1.1. Diagramă Use Case

Un prim pas a fost crearea unei diagrame de tip Use Case în care să se evidențieze toate funcționalitățile aplicației și felul în care utilizatorul interacționează cu acestea.

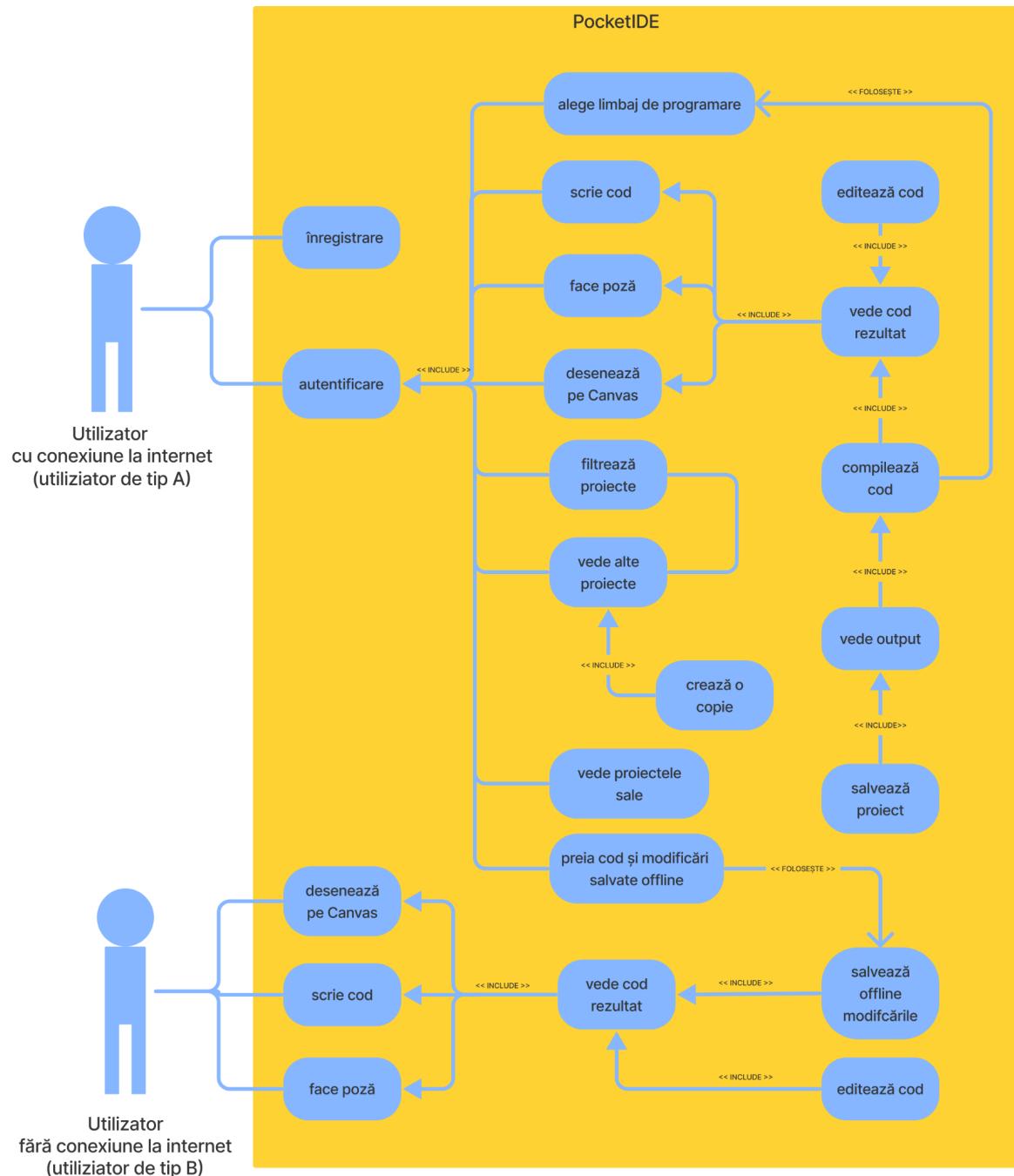


Fig 1: Diagramă de tip Use Case

După cum se poate observa mai sus în Figura 1: Diagramă de tip Use Case, aplicația are ca întâi 2 tipuri de utilizatori: utilizatori care au conexiune la internet și își creează cont pentru a folosi toate funcționalitățile aplicației (utilizatori de tip A) și utilizatori care și-au creat cont dar nu au acces la internet aşa că pot folosi doar o parte din funcționalități (utilizatori de tip B).

Această împărțire a utilizatorilor este cauzată de problema conexiunii cu baza de date și salvarea și sincronizarea datelor, acest lucru fiind imposibil de realizat în mediul offline.

De asemenea, un motiv pentru care cele două tipuri de Use Case-uri diferă aşa mult este faptul că folosirea unui compilator local prezintă riscuri de securitate, având în vedere că aplicația este de hostată online și există interacțiune între utilizatori.

Pentru acest motiv, compilarea și rularea se va realiza pe un server remote, iar în aplicație va fi afișat doar rezultatul.

De asemenea, în funcție de Software Stack-ul pe care îl vom folosi (termen detaliat în capitolul următor) este posibil ca această funcționalitate să nu fie posibilă.

Un utilizator de tip A, poate să facă următoarele: să își creeze un cont nou și să se autentifice cu acesta, poate vedea proiectele altor utilizatori, să le acceseze, să caute după titlu și să filtreze după anumite categorii (cum ar fi limbajul de programare), își poate vedea propriile proiecte și poate crea proiecte noi.

În cadrul unui proiect acesta poate să facă o poză cu telefonul și să o încarce pentru a putea fi recunoscut textul din ea, poate să deseneze în cadrul aplicației pe un Canvas codul ce se dorește a fi recunoscut sau îl poate scrie direct în editor.

Codul scris, poate fi reeditat sau extins cu alte poze, recunoașterea concatenează nu suprascrie, dar existența unui buton de „clear” poate fi folosită.

După ce se ajunge la forma dorită a codului, poate fi ales un limbaj de programare sau identificat în mod automat și apoi trimis către server un request pentru compilarea și rularea codului respectiv.

De asemenea proiectul poate fi salvat.

Un lucru greu de reprezentat în diagramă este faptul că dacă acesta vizualizează codul altor persoane, lucrurile menționate mai sus rămân în continuare valabile, însă atunci când se apasă butonul de „save”, se va crea o copie a acelui proiect și nu o va suprascrie dacă utilizatorul în cauză nu este autorul.

Un utilizator de tip B, are nevoie să fie deja conectat atunci când acesta pierde conexiunea la internet. Cu alte cuvinte, atunci când utilizatorul își descarcă pentru prima dată aplicația, va avea nevoie să își creeze un cont pentru a accesa funcționalitățile offline și evident pentru acest lucru, aplicația va trebui să aibă un mod de a stoca datele de autentificare local, dar într-un mod care să fie securizat (cum ar fi memorarea unui session id / token temporar)

Utilizatorul de tip B, nu poate vedea proiectele personale sau ale altor persoane, dar poate crea și edita proiecte offline care vor fi automat transmise către server atunci când revine conexiune la internet.

Funcționalitățile de făcut poză, desenat și scris cod apar în continuare și se va folosi o recunoaștere offline, cel mai probabil utilizându-se Tesseract.js.

#### 4.1.2. Diagramă de Flux

Figura 2: Diagramă de Flux de mai jos prezintă un flow general atunci când un utilizator interacționează cu o pagină/ ecran de proiect în cadrul aplicației. Scopul acesteia este de a complementa detaliile de mai sus din cadrul diagramei Use Case și de a oferi o perspectivă mai bună asupra felului în care aplicația interacționează cu utilizatorul.

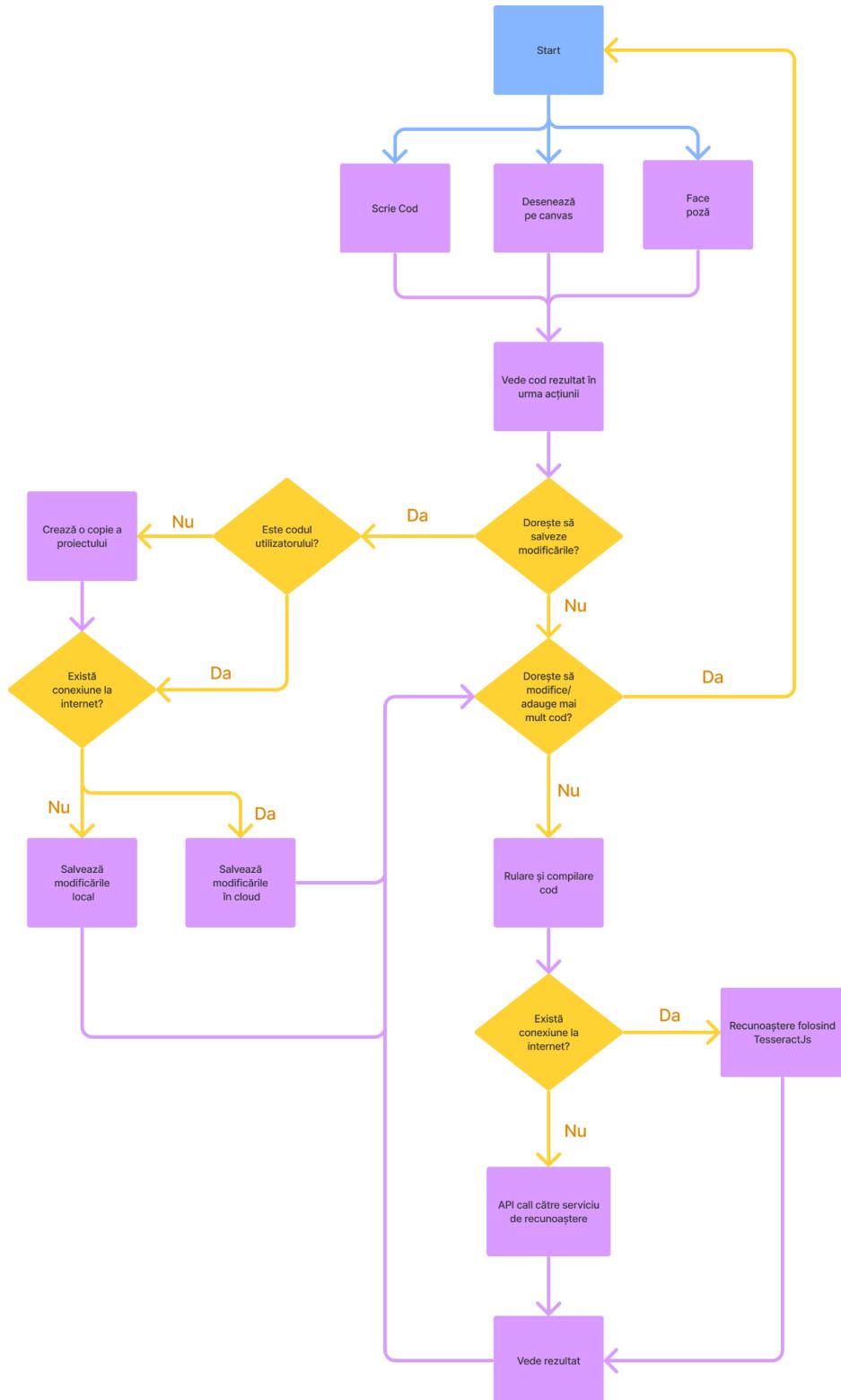


Fig. 2: Diagramă de Flux

## 4.2. Proiectarea arhitecturală

### 4.2.1. Tehnologii folosite

O primă problemă în începerea dezvoltării aplicației a fost nevoie de a găsi în primul rând un software stack potrivit. O scurtă descriere a ce înseamnă un software stack este următoarea: „Un technology stack, cunoscută și sub numele de software stack sau development stack, este un ecosistem de date care înrolează instrumentele, framework-urile și bibliotecile de bază utilizate pentru a construi și a rula aplicația. De exemplu, stack-ul complet al aplicațiilor Facebook include PHP, React, GraphQL, Cassandra, Hadoop, Swift și o serie de alte framework-uri. Un stack tehnologic este în general împărțit în două: partea client (frontend) și partea server (backend). În timp ce tehnologiile de backend includ framework-uri web, limbaje de programare, servere și sisteme de operare, tehnologiile de front end includ framework-uri și biblioteci HTML, CSS, JavaScript și UI.” [5]

Pentru realizarea aplicației a trebuit să fie luat în seamă software stack-ul ce se dorește a fi folosit, încrucișat acesta poate limita funcționalitățile prezentate mai sus în diagrama de flux și diagrama de use case.

#### Cota de piață, căutarea unui software stack potrivit

După cum este precizat și în descrierea problemei, aplicația se dorește a fi multi-platformă, construită atât pentru mobile cât și pentru desktop.

Din punct de vedere mobile, după cum se poate vedea mai jos în Figura 1: utilizare Android vs iPhone la nivel global, Android-ul este cel mai popular sistem pentru telefoane mobile. Totuși cota de piață ne arată că IOS-ul este la rândul lui foarte utilizat, pornind de la un procent de 13% în Africa, până la un procent de 56.4% în Statele Unite ale Americii, pentru anul 2020. Totodată, în Europa iPhone-ul are un procent destul de greu de ignorat de 30%.

Așadar când vine vorba de mobile, dorim să avem un stack care să fie flexibil și să țintească atât platforma Android, cât și IOS.

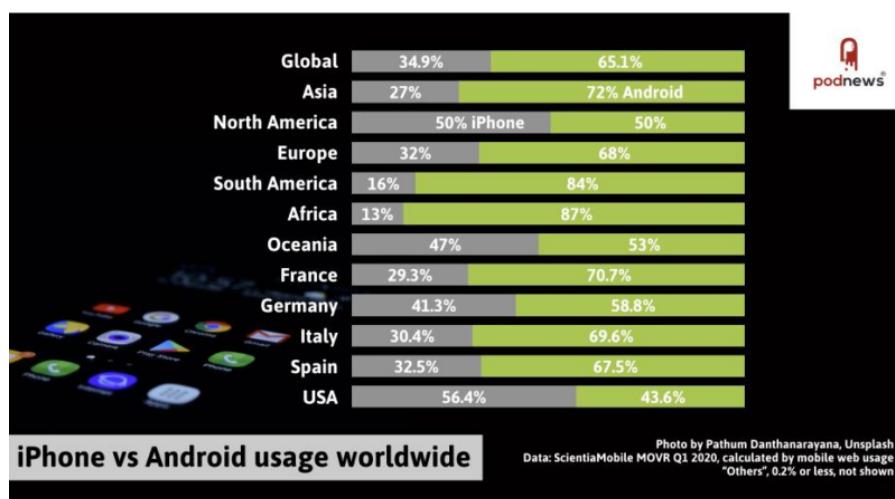


Fig. 1: utilizare Android vs iPhone la nivel global [6]

Conform Figurii 2: Top 5 Framework-uri multi-platformă pentru mobile development ca și framework-uri care să acopere atât partea de Android, cât și partea de IOS avem următoarele: Cordova, Ionic, Xamarine, NativeScript, React Native.

Atât în cadrul acestei figuri, cât și în cadrul Figurii 3: Top trenduri Google Search, se poate observa că React Native ieșe câștigător, acesta fiind în cele din urmă ales.

Totodată alegerea framework-ului React Native pe partea de mobile a influențat și alegerea framework-ului React ca framework pe partea de desktop, pentru web development, întrucât cele două se aseamănă extrem de mult și ambele au o curbă de învățare prietenosă pentru începători.

Pe partea de backend nu a contat tehnologia folosită, întrucât comunicarea cu React și React Native se realizează prin HTTP și API call-uri. Pentru a putea produce un produs minim viabil (MVP) și pentru a avea cât mai puține restricții am ales ca framework de backend Node Js.

The infographic compares five mobile development frameworks: Cordova, Ionic, Xamarin, NativeScript, and ReactNative. It uses icons and color coding to represent different features and characteristics. A legend at the bottom defines the icons:

- LANGUAGE:** HTML5 (Cordova, Ionic), C (Xamarin), JS (NativeScript, ReactNative)
- CODE SHARE:** FULL (Cordova, Ionic), MEDIUM (NativeScript), HIGH (ReactNative)
- RENDERING:** WEBVIEW (Cordova, Ionic), NATIVE (Xamarin, NativeScript, ReactNative)
- IOS COMPILING:** JIT (Cordova, Ionic), AOT (Xamarin), INTERPRETER (NativeScript, ReactNative)
- ANDROID COMPILING:** JIT (Cordova, Ionic), JIT / AOT (Xamarin), JIT (NativeScript, ReactNative)
- COST:** OPEN SOURCE (all)
- POPULARITY:** MEDIUM (Cordova, Ionic), LOW (Xamarin), MEDIUM (NativeScript), HIGH (ReactNative)
- 2020 SCORE:** 5 / 10 (Cordova), 6 / 10 (Ionic), 5 / 10 (Xamarin), 7 / 10 (NativeScript), 9 / 10 (ReactNative)

	Cordova	ionic	Xamarine	NativeScript	ReactNative
</> LANGUAGE	HTML5	HTML5	C	JS	JS
↔ CODE SHARE	FULL	FULL	MEDIUM	HIGH	MEDIUM
⟳ RENDERING	WEBVIEW	WEBVIEW	NATIVE	NATIVE	NATIVE
IOS COMPILING	JIT	JIT	AOT	INTERPRETER	INTERPRETER
ANDROID COMPILING	JIT	JIT	JIT / AOT	JIT	JIT
⌚ COST	OPEN SOURCE	OPEN SOURCE	UP TO \$1900	OPEN SOURCE	OPEN SOURCE
★ POPULARITY	MEDIUM	MEDIUM	LOW	MEDIUM	HIGH
⌚ 2020 SCORE	5 / 10	6 / 10	5 / 10	7 / 10	9 / 10

Fig. 2: Top 5 Framework-uri multi-platformă pentru mobile development [7]

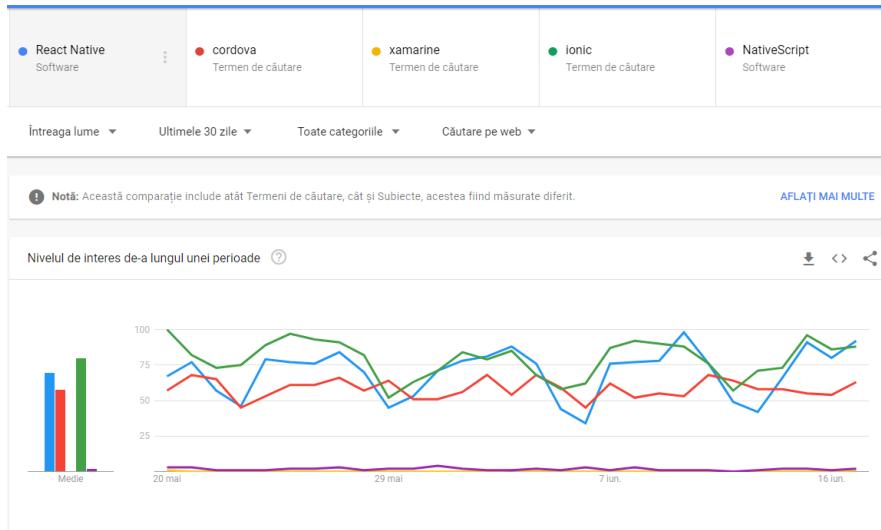


Fig. 3: Top tenduri Google Search

## Probleme cu React Native

Majoritatea dezvoltării aplicației a fost petrecută învățând cele două framework-uri noi React (care este mai mult librărie decât framework) și React Native.

Deși am reușit să asimilez rapid cunoștințe legate de React, React Native mi s-a părut mult mai greu de folosit, întrucât a prezentat foarte multe probleme care au îngreunat mai târziu dezvoltarea aplicației.

Una din principalele probleme a fost faptul că erorile prinse de React Native sunt foarte vagi și de regulă nu zic foarte multe legat de unde anume în cod se produc acestea.

După cum se poate observa în Figurile 4 de mai jos care surprinde o discuție dintr-un thread de pe Github, această opinie este de asemenea întâlnită și la alți developeri, postarea în cauză având 350 de like-uri și fiind una din cele mai populare din thread-ul respectiv care vorbește despre problemele acestui framework.

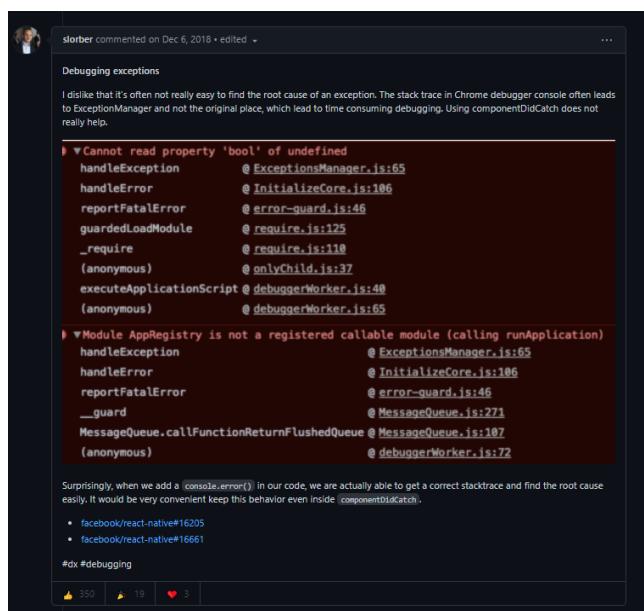


Fig. 4: Screenshot thread Github [8]

O altă problemă a fost faptul că React Native se mișcă mult mai încet decât React (fiind un framework construit peste acesta), este nevoie de instalarea unei aplicații numite Expo pe telefon pentru a putea testa codul și timpul de build este foarte lung.

Foarte des se întâmplă ca aplicația să nu își facă hot reload (adică să apară pe ecran modificările live asupra codului), deși acest lucru în mod normal nu ar trebui să se întâmple.

De regulă când apare această problemă, aplicația trebuie închisă și deschisă din nou, lucru care durează foarte mult, deoarece aceasta trebuie să își facă rebuild.

De asemenea React Native dă foarte des crash, câteodată din motive foarte banale.

Un exemplu, descris și în thread-ul de mai sus, este faptul că poți scrie următorul inline css: `style={{width: "100%"} }`  și să uiți să adaugi "%", rezultatul fiind un crash și nevoie de a redeschide și a aștepta să-și facă din nou rebuild proiectul.

### **Migrarea spre Aplicație Web Progresivă**

În timpul dezvoltării PocketIDE-ului am întâlnit o alternativă interesantă la React Native, cu potențial foarte mare pe partea de ușurare a cerinței de multi-platformă și anume aplicații web progresive, prescurtat PWA - Progressive Web Application.

După cum descrie developer.mozilla: “Progressive Web Apps are web apps that use emerging web browser APIs and features along with traditional progressive enhancement strategy to bring a native app-like user experience to cross-platform web applications. Progressive Web Apps are a useful design pattern, though they aren't a formalized standard. PWA can be thought of as similar to AJAX or other similar patterns that encompass a set of application attributes, including use of specific web technologies and techniques.” [9]

Pe scurt acestea sunt niște aplicații web, care se comportă ca și cum ar fi aplicații native, fiind descărcabile și utilizabile offline. Din punct de vedere tehnic acest lucru este posibil deoarece browserele au căpătat foarte multe funcționalități de-a lungul timpului care le permit să se comporte asemeni unei aplicații pe Android sau IOS (pot folosi camera, microfonul, pot trimite notificări, pot folosi gps și location, etc).

De asemenea, un alt avantaj este faptul că întreg proiectul poate fi realizat după trecerea pe PWA folosind o singură tehnologie pe partea de front end și anume React. Astfel se reduce considerabil timpul de dezvoltare al proiectului, fără a fi nevoie să fie scris în React Native pentru mobile și apoi „tradus” în React pentru desktop.

Sigurele condiții pentru a transforma proiectul web în PWA au fost să hostez site-ul pe o rețea securizată (HTTPS), să am un service worker care rulează (acesta fiind folosit pentru a face cache la date și a face ca aplicația să meargă offline) și să am un fișier Manifest de tip JSON, care să conțină date despre numele aplicației, URL de start, mărimi și nume iconițe folosite, etc.

*Un detaliu important pentru capituloare care urmează este că trecerea pe PWA a fost realizată mai târziu, modelele C4 de exemplu având vechiul software stack folosit.*

#### 4.2.2. Model de dezvoltare și arhitectură

Pentru a porni procesul de dezvoltare al aplicației a fost nevoie de alegerea unei metodologii de dezvoltare și o arhitectură din cele multe disponibile, astfel încât acestea să fie potrivite cu tehnologiile alese și cerințele problemei.

##### Test Driven Development

Ca și modele populare de dezvoltare avem la dispoziție următoarele: Ad-hoc, modelul cascadă, modelul spirală, RUP, Agile, etc.

Pentru acest proiect am ales Test Driven Development (TDD) sau dezvoltare bazată pe testare. Pe scurt TDD are la bază următorul flux surprins în Figura 1 de mai jos: se scrie cod de test care eșuează, se scrie/rescrie cod astfel testul să nu mai eșueze, se face refactorizare sau se modifică sau se adaugă cod nou de test cu funcționalități dorite și se repetă procesul.

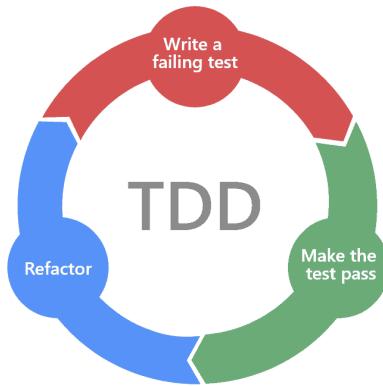


Fig. 1: Flux dezvoltare bazată pe testare [10]

Motivația acestei alegeri a fost nevoie de a dezvolta rapid aplicația, într-un mod sigur și ușor de refactorizat, modularizat și testat. Alt motiv a fost existența librăriei de testare Jest, care e foarte ușor și din punctul meu de vedere intuitiv și plăcut de folosit.

În Figura 2 de mai jos, se poate vedea un exemplu din suită de teste din versiunea finală care testează funcționalitatea de login din controllerul de utilizatori:

```
it("can add user", async () => {
  //arrange
  const body = {
    "username": "cosmin0123",
    "email": "cosmin",
    "phoneNumber": "123",
    "uid": "123"
  }
  const req = { params: {}, query: {}, body }

  //act
  await user.createUser(req, res)

  //assert
  expect(status).toBe(201)
})
```

Fig. 2: Test login/create utilizator

În figurile 3 și 4 de mai jos, se pot observa alte tipuri de teste: unul pozitiv pentru microserviciul de compilare și rulare a programelor C și un test negativ pentru API Gateway, folosindu-se un limbaj nerecunoscut de aplicație

```
it("c hello world program should return accordingly", async () => {
  const body = JSON.stringify({
    "code": "#include <stdio.h> \r\n int main() { \r\n printf(\"Hello World!\\"); \r\n return 0;}"
  });

  const config = {
    method: "post",
    url: url,
    headers: {
      "Content-Type": "application/json",
      "Authorization": `Bearer ${JWT}`
    },
    data: body
  };

  const { data } = await axios(config)
  expect(data).toHaveProperty("stdout", "Hello World!")
  expect(data).toHaveProperty("stderr", "")
  expect(data).toHaveProperty("error", null)
})
```

Fig. 3: Test program „Hello World”, limbaj C

```
it("unsupported language should return error", async () => {
  const config = {
    method: "post",
    url: `${url}/compile`,
    headers: {
      "Content-Type": "application/json",
      "Authorization": `Bearer ${JWT}`
    },
    data: {
      language: "HolyC",
      code: "U0 Hello(){\\\"Hello World\\n\\\"};"
    }
  };

  let error = null
  try {
    await axios(config)
  } catch (err) {
    error = err.response.data
  }
  expect(error).not.toBeNull()
  expect(error).toHaveProperty("message", "Language not supported!")
})
```

Fig. 4: Test negativ, limbaj neacceptat

## Arhitectură bazată pe microservicii

Totodată pentru acest proiect am avut de ales dintre două tipuri de arhitecturi: monolit și microservicii.

După cum se poate observa mai jos în Figura 5: Monolit vs Microservicii, diferența dintre acestea este decuplarea codului și logicii de business. Fiecare microserviciu este un program independent cu Data Access Layer și Business Logic propriu.

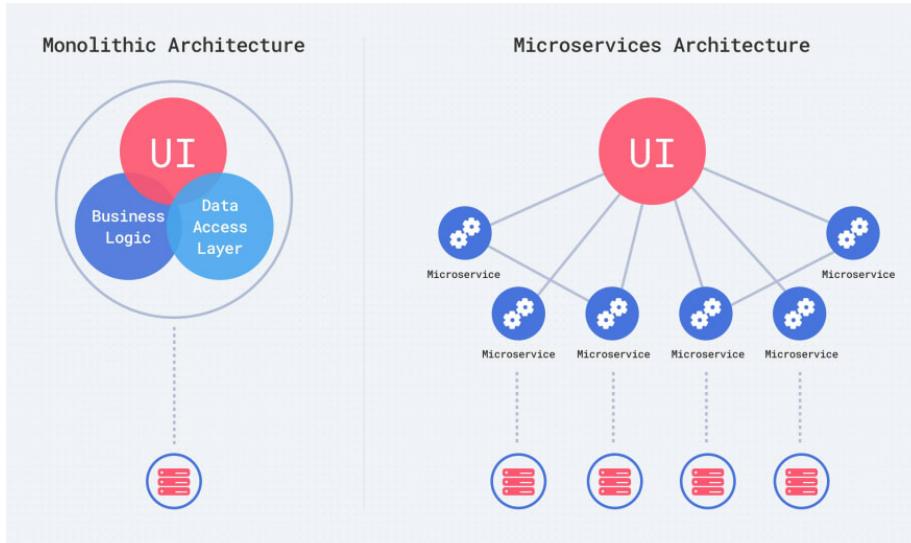


Fig 5. Monolit vs Microservicii [11]

Pentru acest proiect am ales construirea unei aplicații de tip microservicii.

Un motiv principal pentru acest lucru este cauzat de componenta de rulare și compilare a codului. Aici pot apărea foarte multe BUG-uri și exploatari de sistem și nu este o idee bună să fie legată de restul logicii aplicației, în special de componente care lucrează cu datele utilizatorilor și proiectelor.

De asemenea un alt motiv este legat de posibila blocării unui compilator.

Într-o arhitectură bazată pe microservicii, dacă unul din acestea se strică sau blochează restul continua să funcționeze. Astfel aplicația este mult mai ușor de menținut.

Alte motiv suplimentar ar fi posibilitatea foarte ușoară de extindere cu compilatoare pentru limbi suplimentare (în cerință au fost propuse doar limbajele C, C++ și Python).

Și nu în ultimul rând, îmi doresc să creez o aplicație cu microservicii, întrucât până la dezvoltarea acestui proiect, am lucrat doar cu monoliți, fiind o oportunitate bună de a descoperi și învăța lucruri noi.

#### 4.2.3. Design backend & frontend

##### Model C4

Pentru a putea vizualiza mai bine arhitectura pe partea de backend a aplicației am realizat următoarele diagrame C4 pentru nivelele Context, Containere și Componente, după cum se poate observa în figurile de mai jos.

Motivul pentru care am omis ultimul nivel din C4, și anume Cod, este faptul că această parte nu poate fi descrisă cu acuratețe la început întrucât necesită experimentare și prototyping, în special pe partea de recunoaștere a scrisului și rulare a codului.

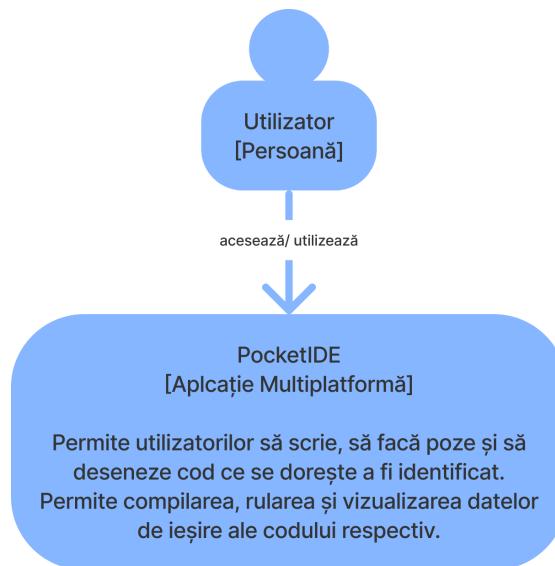


Fig. 1: C4 - Nivel Context

După cum se poate observa în Figura 1 primul nivel este destul de trivial întrucât aplicația își propune să fie independentă și fără servicii și software-uri extreme. Aceasta este un pas înapoi de la toate detaliile menționate până în acest moment pentru a crea o privire de ansamblu a ceea ce își propune proiectul să fie.

O posibilă extindere a acestei diagrame, în cazul în care crearea unui microserviciu care să recunoască scrisul eșuează se poate observa mai jos în „Figura 2: C4 - Nivel Context fără microserviciu local de recunoaștere”.

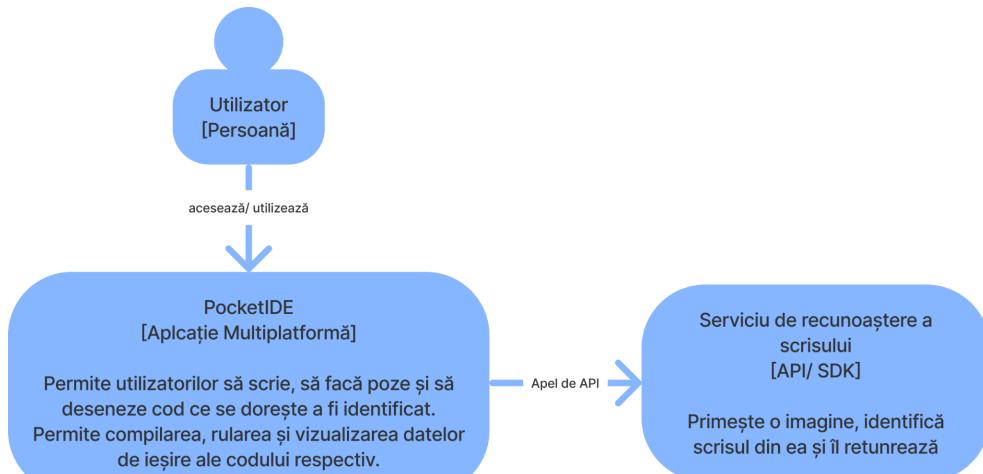


Fig 2: C4 - Nivel Context fără microserviciu local de recunoaștere

În figura 3 de mai jos, se pot observa elementele principale ale aplicației. Această fiind pe deoparte web, va trebui să fie hostată online și să servească utilizatorii cu imagini statice și cu aplicația single page realizată cu ajutorul librăriei React. Totodată aplicația este și mobile, creată cu ajutorul framework-ului React Native (care mai târziu în development a fost înlocuit cu aplicație web progresivă).

Este de remarcat faptul că se dorește păstrarea tuturor funcționalităților atât pe partea de web, cât și pe cea de mobile. Ambele trimit requesturi HTTP către o aplicație API pentru a obține toate datele necesare identificării, rulării și compilării codului. Aplicația API folosește la rândul ei requesturi HTTP către microservicii, întrucât este o arhitectură bazată pe acestea. Microserviciile prezente sunt cele de compilare a codului C, C++ și Python și de recunoaștere a scrisului.

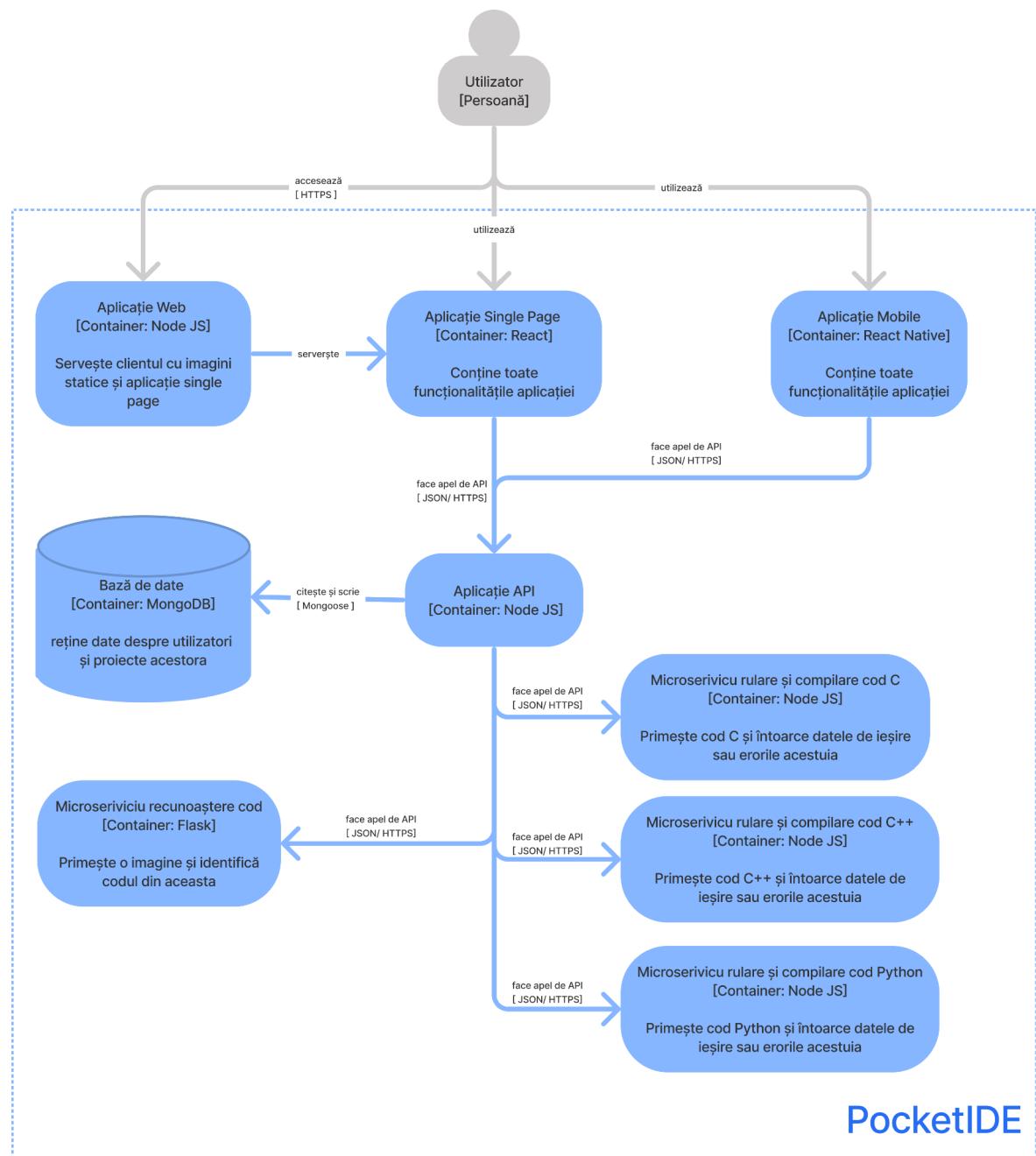


Fig. 3: C4 - Nivel Containere

Bineînțeles, ca și în cazul de mai sus, dacă nu se poate crea un microserviciu de recunoaștere a scrisului, se va fi folosi un API extern. Figura 4 de mai jos evidențiază această structură la nivel de containere a sistemului

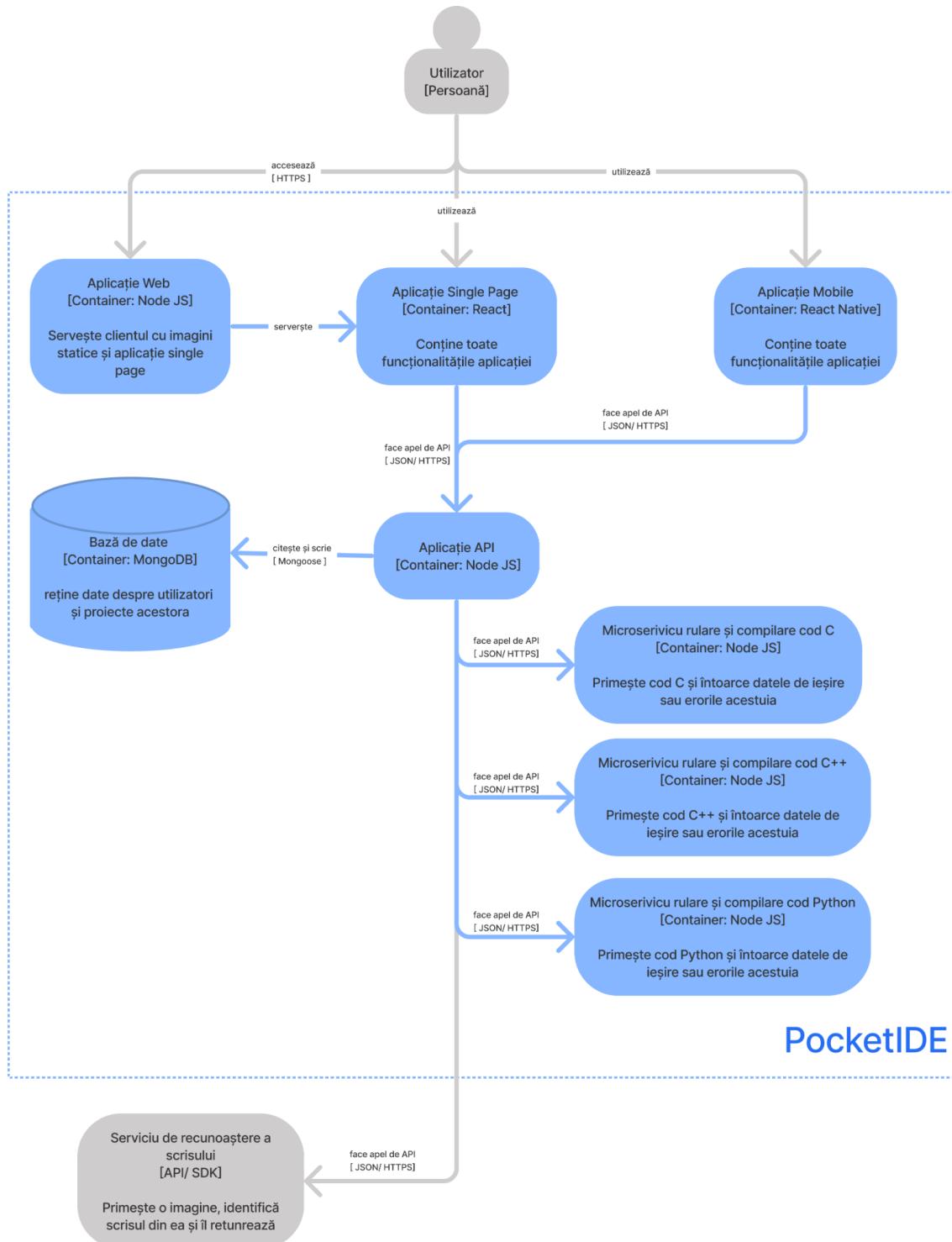


Fig 4: C4 - Nivel Containere fără microserviciu local de recunoaștere

După cum se poate vedea în Figura 5, nivelul al treilea, și în acest caz ultimul, este cel al componentelor și afișează detaliile aplicației API.

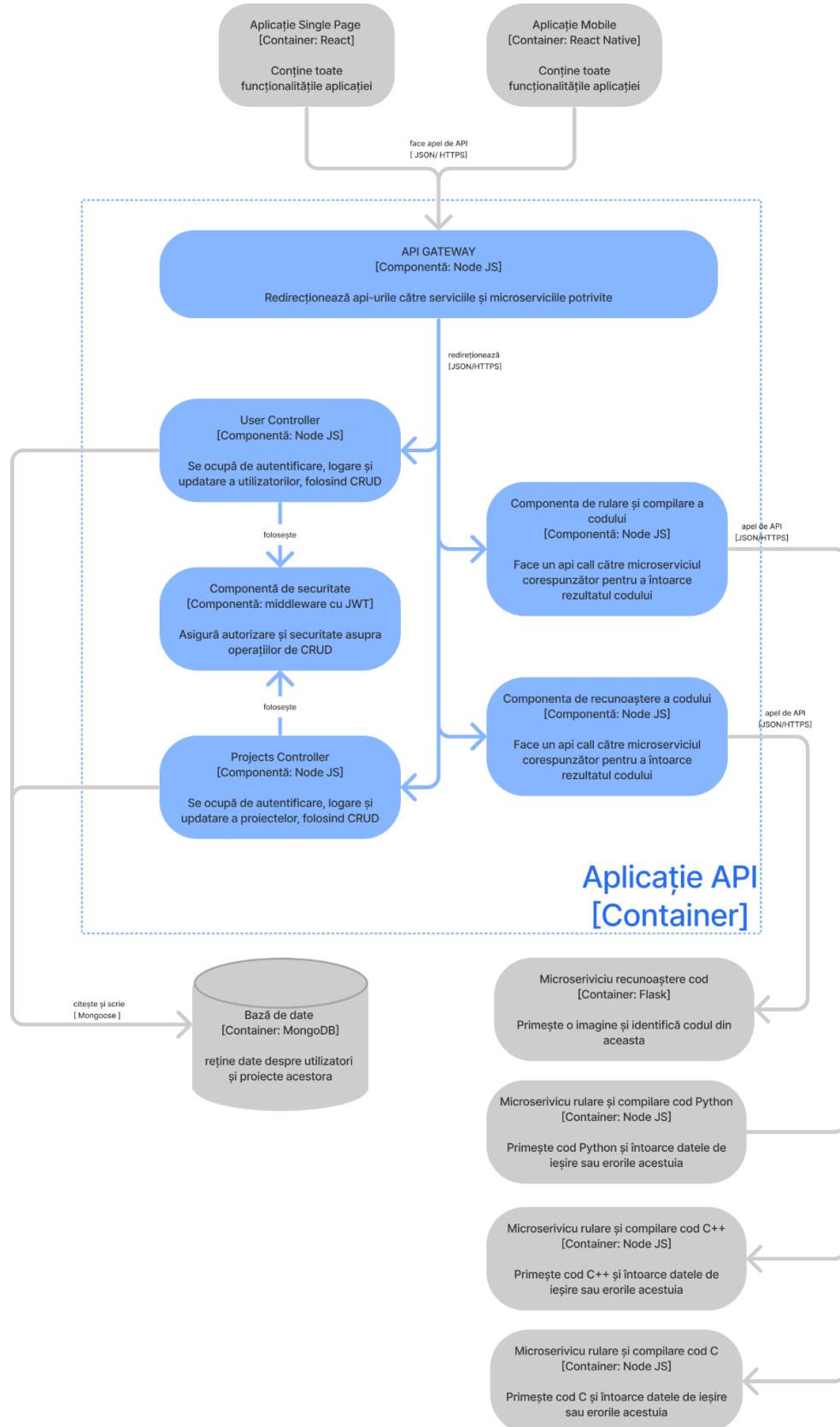


Fig 5: C4 - Nivel Componente

Din Figura 5 de mai sus, se observă nevoie unei componente de securitate, aici fiind folosit cel mai probabil un middleware și un JSON Web Token (JWT).

Procesul este destul de simplu, fiecare utilizator când se autentifică primește un JWT care conține date despre acesta (cum ar fi id) și care este semnat cu o cheie secretă pe server.

Pentru a vedea dacă un utilizator este autorizat să facă un request, se va verifica atât semnătura, cât și id-ul și rolul acestuia atunci când face request-ul.

Întrucât avem ca entități în baza de date utilizatori și proiecte, acestea vor avea la rândul lor componente și mai exact controlere cu operații de CRUD - Create, Read, Update, Delete, care vor folosi acest middleware.

De asemenea, având o arhitectură bazată pe microservicii include automat existența unui API Gateway care să redirecționeze API call-urile către serviciile și microserviciile corecte.

Reprezentarea sistemului fără serviciu local ar fi extrem de asemănătoare cu reprezentarea normală în acest caz. Microservicii nu au avut nevoie de crearea unui model de nivel 3, întrucât acestea sunt directe și atomice și nu se cunosc din prima detaliu de implementare.

În Figura 6 de mai jos se poate vedea o privire de ansamblu a modelului C4.

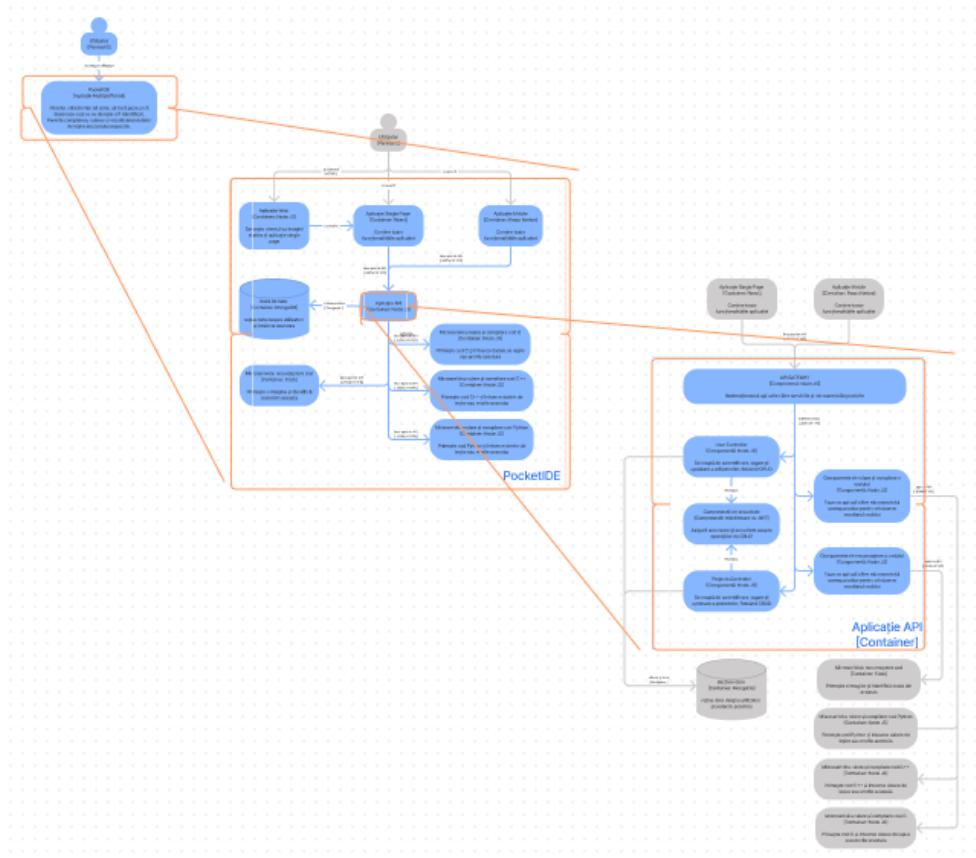


Fig 6: Privire de ansamblu model C4

## Design Figma

Pentru a putea vizualiza cum ar trebui să arate aplicația pe partea de front-end am folosit un tool asemănător cu Adobe XD, numit Figma.

Acesta este un tool foarte puternic, gratuit, cu foarte multe funcționalități, specializat în crearea de wireframe-uri, design-uri și prototipuri.

Un motiv pentru care am ales Figma peste Adobe XD pentru partea de UI/UX design este faptul că acesta îți generează cod CSS pentru componentele create, acestea fiind ușor de preluat și utilizat în aplicație.

De asemenea faptul că este bazat pe browser face ca Figma să fie mult mai accesibil.

După cum se poate observa în Figura 7 de mai jos, în mod normal în realizarea unui design UI/UX au loc 3 etape:

- partea de wireframe care se concentrează pe funcționalități și unde anume se află poziționate acestea
- partea de mockup care prezintă felul în care arată aplicația, folosindu-se evident date mock-up-ute
- partea de prototip în care este prezentată partea de mockup plus interacțiunea utilizatorului cu aplicația prin diferite evenimente: scroll, apăsat pe butoane, etc.

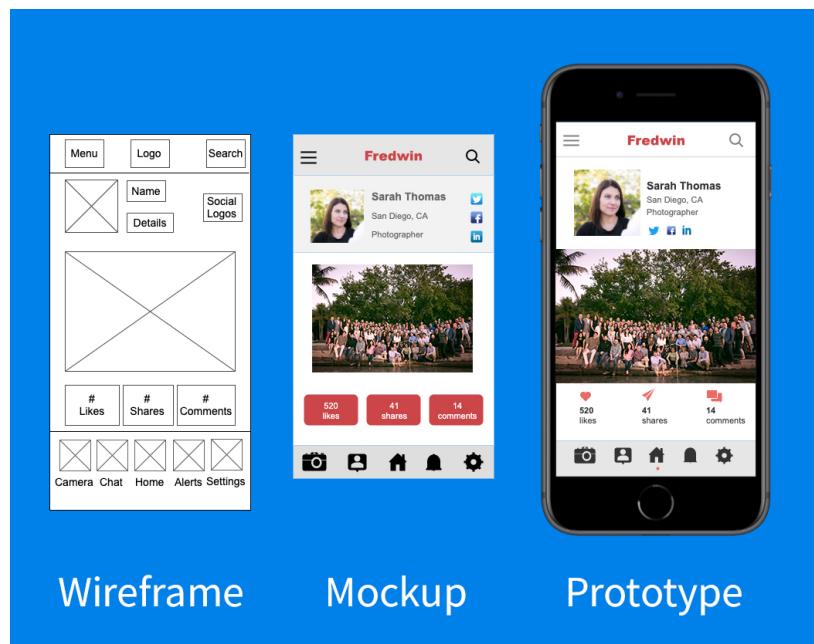


Fig. 7: wireframe vs mockup vs prototype [12]

Întrucât interfața la PocketIDE nu trebuie să fie foarte complexă, am optat doar pentru realizarea unui Mockup. Pentru acesta a fost nevoie de alegerea unor tematici, cum ar fi fonturile și schema de culori, fiind selectate cele din Figura 8 de mai jos.



Fig. 8: fonturi și schema de culori

În figura 9 se poate vedea design-ul final după care va fi creată aplicația de frontend folosindu-se React și React Native. Abordarea la design este „mobile first” întrucât se dorește să fie în primul și primul rând folosită pe mobile.

Folosindu-se principiile de web responsiveness, se poate crea o versiune care să arate bine pentru desktop, acest lucru fiind ușor de realizat de exemplu cu ajutorul a media query-urilor din CSS.

După cum se poate vedea, designul a fost creat după diagrama de use case descrisă în capitolul “4.1. Dezvoltarea și analiza cerințelor” și evidențiază componentele principale ale aplicației.

În primele două poze din figura de mai jos se pot observa crearea și vizualizarea de proiecte.

În poza a treia se poate vedea pagina/ ecranul unui proiect în cadrul căruia se poate alege opțiunea de a desena (lucru sugerat de butonul cu creion ca iconă), se poate alege opțiunea de a face poză (lucru sugerat de butonul cu cameră ca iconă) și opțiunea de a scrie cod direct (lucru sugerat de butonul cu cod ca iconă).

În ultima poză se poate observa ecranul special pentru afișarea rezultatului sau a erorii codului rulat.

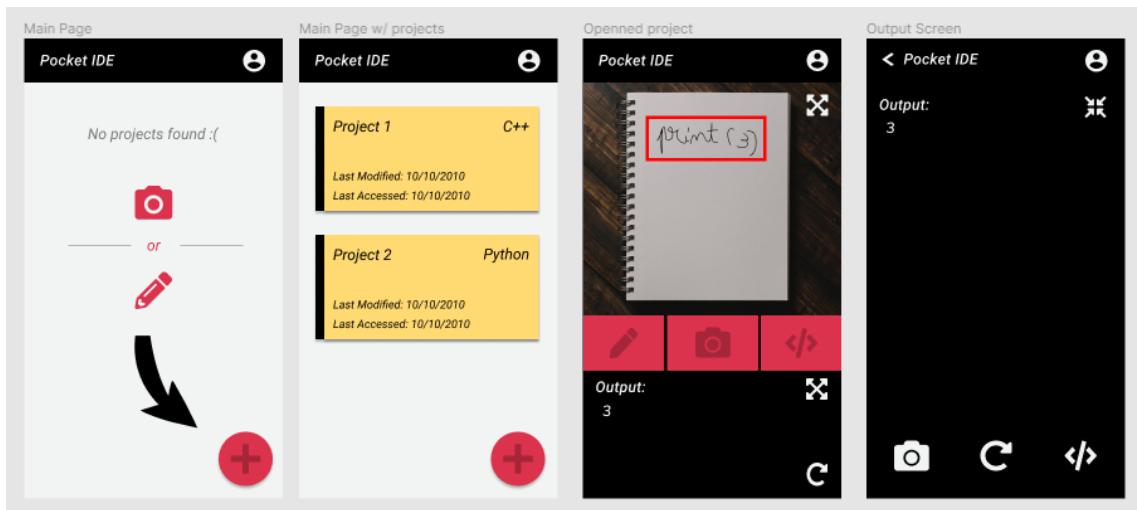


Fig. 9: Design final aplicație, Figma

## 4.3. Detalii implementare

În cele ce urmează voi prezenta soluțiile de cod implementate și încercările întâmpinate în dezvoltarea aplicației.

### 4.3.1. TextReco - microserviciu de recunoștere a scrisului

#### Încercări proprii

Inițial în cadrul proiectului mi-am propus să utilizez propriul serviciu de recunoaștere. După cum se poate observa mai sus în capitolele anterioare, planul era ca acesta să fie inclus într-un microserviciu.

Modelul în sine se dorea a fi construit în Python, folosindu-se librăriile OpenCV și TensorFlow cu ajutorul cărora să se poată prelucrarea ușor imaginile și să se poată construi o rețea neuronală conoluțională.

Apoi planul era ca acest model să fie preluat și folosit într-un server de Flask pentru a servi request-urile de recunoaștere a imaginilor venite de la API Gateway.

Pe această parte au fost nevoie de încercări și cercetare și am întâmpinat multe probleme.

O primă problemă pe care am întâmpinat-o a fost găsirea unui set de date suficient de mare și relevante de tipul EMNIST cu structură și etichete corecte.

EMNIST este prescurtarea la Extended MNIST, sau MNIST Extins. Așadar, pentru a explica ce anume este EMNIST trebuie explicat în primul rând ce înseamnă MNIST.

Baza de date MNIST - Modified National Institute of Standards and Technology, este o bază de date ce conține cifre scrise de mâna, având 60,000 de imagini de antrenare și 10,000 de imagini pentru testare. [13]

Exemple de caractere ce apar în această bază de date se poate observa mai jos în Figura 1: Exemple MNIST.



Fig. 1: Exemple MNIST [14]

Așadar dacă MNIST conține cifre scrise de mâna, EMNIST conține atât cifre, cât și litere.

Din nefericire nu am reușit să găsesc un set de date suficient de bun care să le conțină pe ambele. La momentul scrierii documentației există de pildă pe popularul site Kaggle, doar 19 astfel de seturi de date, care nu sunt foarte relevante și utilizabile.

De asemenea, pe lângă litere și cifre, pentru a putea identifica cod vom avea nevoie și de caractere speciale cum ar fi plusuri sau accolade.

O altă problemă întâmpinată a fost încercarea de a crea un model.

Fără prea multă cunoștință de Python și machine learning am încercat cu ajutorul informațiilor și documentației de pe internet să creez propriul model de recunoaștere a caracterelor.

Pentru a fi puțin mai prietenos pentru mine ca începător, în special până la găsirea unui set de date care să conțină și litere, am încercat să creez un model pentru partea de MNIST.

O problemă, deși modelul avea acuratețe de 98% și a mers să fie adăugat la un server de Flask, era faptul că am întâmpinat multe probleme și inexactități atunci când am încercat să-l testeze manual (cu alte cuvinte modelul făcea overfit).

De exemplu trei teste pentru serverul respectiv de Flask au fost făcute folosind următoarele imagini de mai jos ce se pot observa în Figura 2 - cifra patru, Figura 3 - cifra patru scrisă înclinată și Figura 4 - cifra opt.

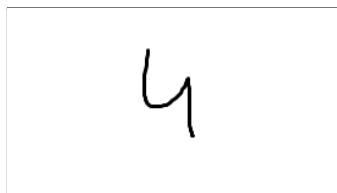


Fig. 2: cifra patru

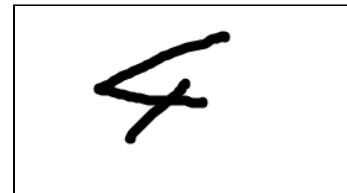


Fig. 3: cifra patru scrisă înclinată

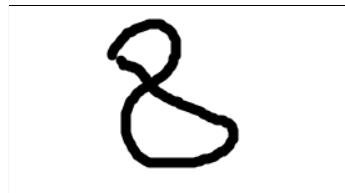


Fig. 4: cifra opt

Rezultatele au fost următoarele:

- 1) Pentru Figura 2 și pentru multe alte cifre scrise „frumos”, fără înclinare sau zgromot, serverul a întors corect cifra corespunzătoare ca răspuns.
- 2) Pentru Figura 3 și pentru multe alte cifre cu scris înclinat sau tremurat serverul nu a reușit să întoarcă cifra corectă, deși din punct de vedere uman, este foarte ușor de recunoscut faptul că de exemplu cifra de mai sus este patru.
- 3) Pentru Figura 4, cifra opt, deși scrisă mai puțin frumos, nu primit o etichetă corectă, serverul identificându-se ca fiind cifra șase.

În final, întrucât modelul nu era de încredere și nu am reușit să găsesc un set de date potrivit, am preferat înlocuirea lui cu API call-uri către servicii externe care se ocupă mult mai bine de acest lucru.

## Servicii Cloud

M-am documentat legat de servicii externe de recunoaștere a scrisului de mână și singurele opțiuni gratuite găsite (dar limitate) au fost Azure Form Recognizer și Google Cloud Vision.

Am încercat să testeze ambele servicii separat, însă am întâmpinat probleme de acuratețe la ambele API-uri aşa că am decis să le combin și să încerc să obțin cel mai bun rezultat.

De asemenea pe lângă acestea două, a fost adăugat și OCR.Space pentru recunoașterea textului de tipar, deși după cum se poate vedea în Figura 5: Azure vs GCloud vs OCR.SPACE de mai jos, acestea se descurcă foarte bine la rândul lor la acest tip de recunoaștere, fiind capabile să identifice chiar mai multe litere.

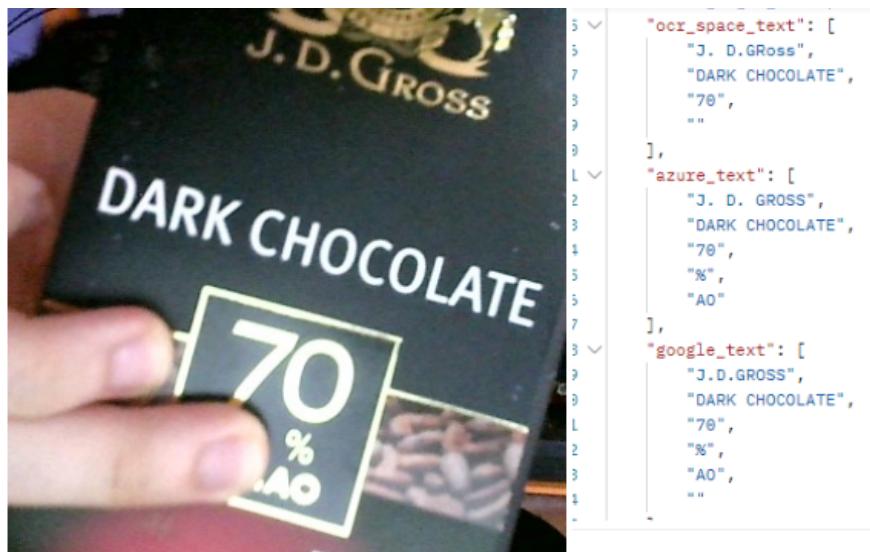


Figura 5: Azure vs GCloud vs OCR.SPACE

În figurile 6, 7 și 8 de mai jos se pot observa mai multe teste și rezultatele acestora.

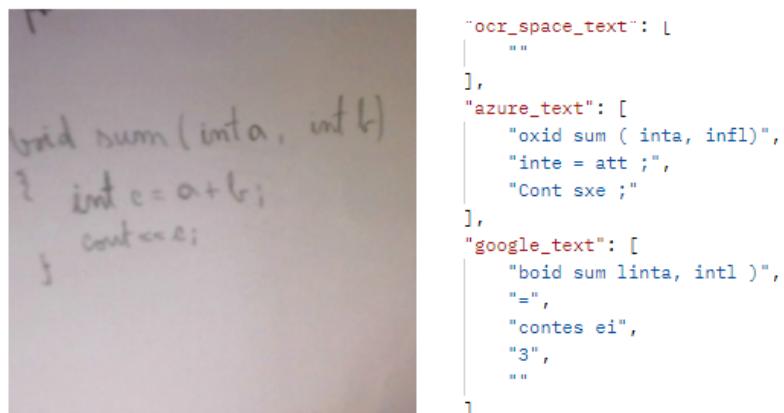
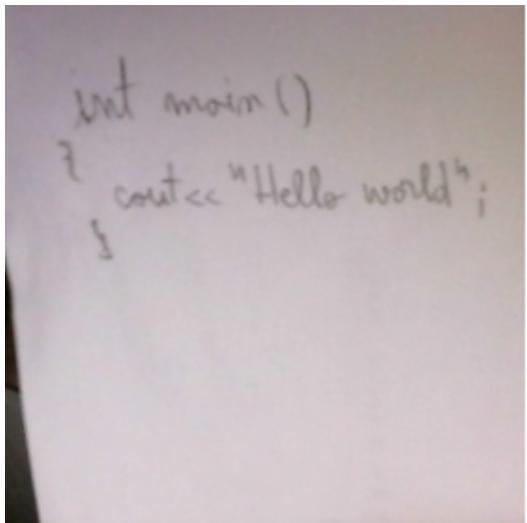
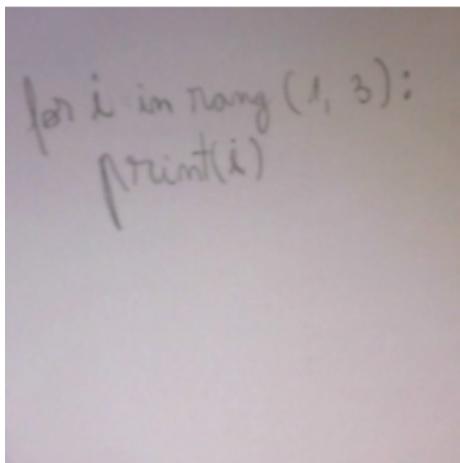


Figura 6: recunoaștere funcție C++



```
"ocr_space_text": [  
    ""  
,  
    "azure_text": [  
        "int main ( )",  
        "coutec \\"Hello world\\\" ;",  
        "-"  
,  
    "google_text": [  
        "ent main()",  
        "2",  
        "couter \\"Hello world\"",  
        "5",  
        "  
"]
```

Figura 7: recunoaștere funcție C++



```
"ocr_space_text": [  
    ""  
,  
    "azure_text": [  
        "187 1 in Tang ( 1, 3 ) :",  
        "print(i)"  
,  
    "google_text": [  
        "for i in rang (1, 3);",  
        "vrinte)",  
        "  
"]
```

Figura 8: recunoaștere program Python

După cum se pot vedea în figurile 6, 7 și 8 de mai sus, nici Google, nici Microsoft nu au acuratețe perfectă, mai ales dacă poza nu este făcută clar.

O altă problemă este cauzată de folosirea caracterelor speciale, care sunt foarte folosite în programare, dar care sunt etichetate de obicei greșit sau ratate complet de cele două. De exemplu, după cum se poate vedea în Figura 7, acolada deschisă este văzută ca doi și acolada închisă este văzută ca cinci de către GCloud, în timp ce Azure nu le identifică deloc.

În mare, majoritatea caracterelor sunt identificate oarecum corect, inexactitățile putând fi corectate de utilizator. Totuși pentru o experiență cât mai plăcută vom crea un microserviciu suplimentar care se va folosi de acesta pentru a returna cea mai bună variantă.

De precizat o problemă pe care am întâmpinat-o, pe lângă setup-ul dificil la Google Cloud Vision, a fost faptul că Form Recognizer cere ca date de intrare un link către o imagine în loc de un fișier sau un text Base64.

Așadar acest microservicu, pe care l-am numit TextReco, se ocupă pe lângă primirea unui imaginii în format Base64 și cu salvarea acestuia și generarea unui link pentru a putea fi accesat de către Azure.

O altă problemă întâmpinată aici a fost faptul că în development, microserviciul este hostat local și Form Recognizer nu poate să-l acceseze.

Așa că am folosit ca soluție pentru development un serviciu extern gratuit dar limitat pentru stocarea de imagini, numit ImageBB.

#### **4.3.2. CodeFinder - microserviciu de alegere a recunoașterii corecte**

Codefinder folosește cele 3 date de ieșire ale OCR.Space, Google Cloud Vision și Azure Form Recognizer, returnate de TextReco, și încearcă să găsească la fiecare linie din poza transmisă în request, soluția cea mai bună.

În cele ce urmează voi explica algoritmul.

După cum a fost menționat în capitolul trecut, OCR space este folosit pentru recunoașterea literelor de tipar, lucru pe care îl au de asemenea în comun și Google Cloud și Azure, prin urmare acesta va fi ignorat.

În continuare vom nota cele două servicii rămase astfel:

- Google Cloud Vision - G
- Azure Form Recognizer - A

TextReco returnează atât pentru G, cât și pentru A un vector de string-uri, fiecare element de ordin  $x$  din vector reprezentând linia de ordin  $x$  din textul identificat în imagine.

CodeFinder compară ambele linii din fiecare serviciu și folosind machine learning îl alege pe cel care are probabilitatea cea mai mare de a fi scris în limbajul de programare selectat de utilizator.

După acesta trece pe liniile următoare din G și A și repetă până când una din ele se termină.

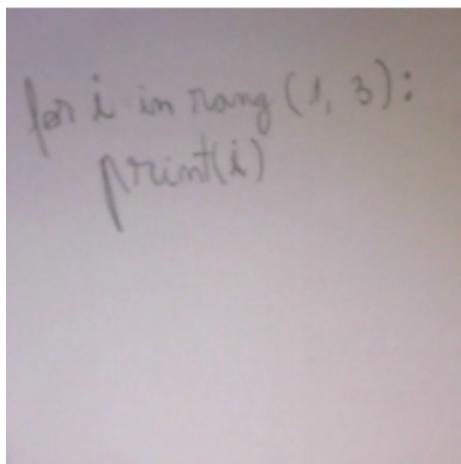
După cum am menționat mai sus există probleme în ceea ce privește partea de identificare a limbajului. În acest caz, dacă G sau A au o linie cu un singur caracter sunt șanse mari ca aceasta să fie acoladă.

Pentru acest caz, este „hardcoded” adăugarea unei accolade și se avansează pe linia următoare pentru serviciul respectiv.

Pentru partea de machine learning am folosit un serviciu suplimentar numit Algorithmia, în care am folosit un API de identificare al limbajului de programare.

Mai jos, în Tabelul 1: alegere output program Python, se poate vedea un exemplu pentru felul în care funcționează acest algoritm, cât și diferența dintre datele de ieșire așteptate și cele reale. După cum se poate observa outputul la CodeFinder are doar un singur caracter greșit, în timp ce GCloud are cinci erori și Azure are patru erori.

Input:



```
"ocr_space_text": [
    ""
],
"azure_text": [
    "187 1 in Tang ( 1, 3 ) :",
    "print(i )"
],
"google_text": [
    "for i in rang (1, 3);",
    "vrinte)",
    ""
]
```

Limbaj: Python

Algoritm:

G[0]: "for i in rang (1, 3);" ~> 0.99840835230122 probabilitate  
A[0]: "187 1 in Tang (1, 3):" ~> 0.40185112431339 probabilitate

-----  
=> este ales G[0]

G[1]: "vrinte)" ~> 0.16953487559461 probabilitate  
A[1]: "print(i )" ~> 0.94805396926533 probabilitate

-----  
=> este ales A[1]

Final A

Output: G[0] + "\n" + A[1]  
= "for i in rang (1, 3);\nprint(i )"

Text Original:

```
for i in rang (1,3):
    print(i)
```

Text final:

```
for i in rang (1,3);
print(i )
```

Număr caractere greșite algoritm: **1**

Număr caractere greșite GCloud: **5**

Număr caractere greșite Azure: **4**

Tabel 1: alegere output program Python

### **4.3.3. Microservicii de compilare și rulare a codului**

Pentru partea de rulare și compilare a codului am stabilit ca fiecare limbaj că aibă propriul microserviciu pentru a fi mai ușor de menținut și de extins.

De asemenea, cu această arhitectură, dacă unul dintre ele are o vulnerabilitate de sistem sau se blochează, celelalte microservicii nu vor fi afectate.

Pentru a putea rula orice cod și a-i vedea erorile și datele de ieșire am folosit modulul `child_process` din NodeJS și mai exact metoda `child_process.exec()`.

Cu ajutorul acestei metode am avut posibilitatea de a executa comenzi Shell în Node și de a le vedea outputul și erorile prin intermediul funcției de callback dată ca parametru.

Un motiv pentru care am avut nevoie de abilitatea de executare de comenzi shell a fost pentru a putea compila și executa fișiere.

O problemă care apare în această abordare este faptul că în funcție de librăriile instalate și sistemul folosit, e posibil ca acest lucru să nu funcționeze, în special dacă vrem să hostăm microserviciile online.

De pildă pe Windows nu poate fi executată comanda „gcc” fără a face setup-ul necesar sau comanda „python” fără a instala pachetul în cauză.

Așa că o soluție pentru a putea rula oricând microserviciile și a le putea hosta online a fost includerea acestora într-o imagine docker în care să aibă deja instalate toate dependințele necesare unei rulări corespunzătoare.

Astfel nu va mai fi nevoie de instalarea niciunui pachet suplimentar, imaginile respective putând fi rulate din prima atât în cloud, cât și local.

Astfel fiecare microserviciu se reduce în mare parte la următoarele instrucțiuni, evidențiat mai jos în Figura 1: compilare și rulare cod C++, având evident ceteva excepții în funcție de limbaj, dacă e compilat (cum e C++) sau dacă e interpretat (cum e Python):

- 1) se primește requestul de la API Gateway și se extrage din body codul ce se dorește a fi compilat;
- 2) se generează un id unic pentru fiecare cod și se crează un fișier ce are ca nume acel id (acest lucru fiind făcut pentru a nu exista conflicte între fișiere) în care este scris codul respectiv folosind modulul File System din NodeJs;
- 3) după scrierea codului în fișier este executată cu ajutorul metodei `exec` comanda de compilare (în figura de mai jos fiind folosită comanda `g++`);
- 4) pentru a salva spațiu, este șters fișierul compilat;
- 5) folosind librăria `exec` din nou, este rulat executabilul creat la pasul 3;
- 6) folosind callback-ul la `exec`, sunt returnate ca response în format JSON erorile și afișările de ecran, salvate în variabilele: „`stdout`”, „`stderr`” și „`error`”;
- 7) este șters fișierul executabil;

```

29 app.post("/", middleWare, (req, res) => {
30   try {
31     const data = req.body.code
32     const fileName = `${uuidv4()}.cpp`
33
34     fs.writeFile(fileName, data, (err) => {
35       if (err) return res.status(500).send(err)
36
37       const executableName = `${uuidv4()}`
38       exec(`g++ ${fileName} -o ${executableName}`, options, (error, stdout, stderr) => {
39         if (stderr || err) return res.json({ stdout, stderr, error })
40
41         fs.unlinkSync(fileName)
42         exec(`./${executableName}`, options, (error, stdout, stderr) => {
43           res.json({ stdout, stderr, error })
44           fs.unlinkSync(executableName)
45         })
46       })
47     })
48   }
49
50 } catch (err) {
51   console.log("CPP => ", err)
52   res.sendStatus(500)
53 }
54 )

```

Fig.1: compilare și rulare cod C++

În cazul Python nu va mai fi partea de compilare, folosindu-se o singură comandă de exec.

După crearea și testarea microserviciilor de C, C++ și Python, am hotărât extinderea acestora cu limbaje suplimentare, adăugând următoarele: Haskell, Java, Scala, JavaScript și PHP.

După cum se poate vedea în Figura 2:Dockerfile C++ vs Java, niște probleme pe care le-am întâmpinat au fost legate de construirea imaginilor docker.

De pildă pentru C, C++ și JavaScript a fost de ajuns să importăm imaginea node care aveau deja incluse toate librăriile necesare, în timp ce pentru Java a fost nevoie de construirea unui DockerFile mai complex, care să descarce atât openjdk pentru compilarea și rularea fișierelor Java, cât și node pentru a putea rula serverul de Express.

<pre> 1 FROM node:12 2 WORKDIR /app 3 COPY package*.json . 4 RUN npm install 5 COPY . . 6 ENV PORT=8083 7 EXPOSE 8083 8 CMD [ "npm", "start"] </pre>	<pre> 1 FROM openjdk:16-slim-buster 2 3 RUN apt-get update; apt-get install -y curl \ 4   &amp;&amp; curl -sL https://deb.nodesource.com/setup_14.x   bash - \ 5   &amp;&amp; apt-get install -y nodejs \ 6   &amp;&amp; curl -L https://www.npmjs.com/install.sh   sh 7 WORKDIR /app 8 COPY package*.json . 9 RUN npm install 10 COPY . . 11 ENV PORT=8085 12 EXPOSE 8085 13 CMD [ "npm", "start"] </pre>
--	--

Fig. 2: Dockerfile C++ vs Java

De asemenea alte probleme pe care le-am întâmpinat au fost în cadrul Java și Scala. Pentru a putea compila fișierele respective, numele claselor principale trebuiau să corespundă cu numele fișierelor. Pentru acest caz am folosit regex pentru a le căuta și redenumi corespunzător.

#### 4.3.4. Api gateway și middleware

În final proiectul are următoarele microservicii ce pot fi observate mai jos în Figura 1: Microservicii PocketIDE. Acestea sunt containerizat în imagini Docker și pot fi accesate fie local, fie online de pe platforma Heroku unde sunt hostate.

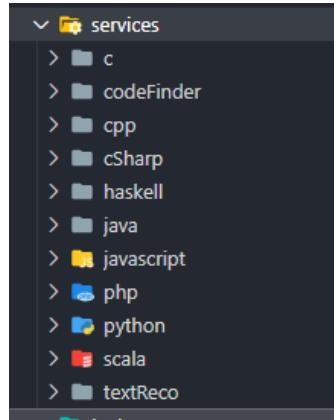


Fig. 1: Microservicii PocketIDE

Pentru a putea să le folosim atât în dezvoltare, cât și în producție am creat un API Gateway personalizat care implementează această soluție. Pe scurt, acest API Gateway este la rândul lui un server de Express ce conține un JSON cu toate microserviciile din aplicație.

După cum se poate observa în Figura 2: Hartă Microservicii, am creat un obiect ce conține URL-urile ce poate fi apelate pentru rularea serviciilor respective, având valori diferite în funcție de mediul de lucru (dezvoltare sau producție).

```
const environment = process.env.NODE_ENV || require("./secret.js").NODE_ENV
let serviceMap = { //production
  "codeFinder": "https://cosmin-afta-code-finder.herokuapp.com/",
  "textReco": "https://cosmin-afta-text-reco.herokuapp.com/",
  "javascript": "https://cosmin-afta-javascript.herokuapp.com/",
  "python": "https://cosmin-afta-python.herokuapp.com/",
  "c": "https://cosmin-afta-c.herokuapp.com/",
  "cpp": "https://cosmin-afta-cpp.herokuapp.com/",
  "java": "https://cosmin-afta-java.herokuapp.com/",
  "php": "https://cosmin-afta-php.herokuapp.com/",
  "haskell": "https://cosmin-afta-haskell.herokuapp.com/",
  "scala": "https://cosmin-afta-scala.herokuapp.com/"
}
if (environment === "development") {
  serviceMap = {
    "codeFinder": "http://localhost:8079/",
    "textReco": "http://localhost:8080/",
    "javascript": "http://localhost:8081/",
    "python": "http://localhost:8082/",
    "c": "http://localhost:8083/",
    "cpp": "http://localhost:8084/",
    "java": "http://localhost:8085/",
    "php": "http://localhost:8086/",
    "haskell": "http://localhost:8087/",
    "scala": "http://localhost:8088/"
  }
}
```

Fig. 2: Hartă Microservicii

Pentru a putea evita abuzurile sau încercările de a corupe microservicile, acestea au fost protejate cu JWT, API Gateway-ul fiind singurul care are acces către ele, putând astfel fiu ușor de monitorizat fiecare request.

Totodată API Gateway-ul are la rândul lui un middleware ce folosește JWT, acesta fiind folosit pentru a permite doar utilizatorilor autentificați să facă requesturi.

#### 4.3.5. Front End

Pentru partea de front end am folosit inițial React și React Native. Iar în timpul dezvoltării aplicației în cele din urmă am trecut pe varianta Progressive Web Application, rămânând doar cu React.

Pentru a asigura lucrul offline și a face aplicația descărcabilă, am folosit service workers și am urmat toți pașii necesari pentru această direcție. După cum se poate vedea în Figura 1 de mai jos, după rularea unei Lighthouse din Google Chrome, raport acestuia a confirmat faptul că aplicația este într-adevăr un PWA.

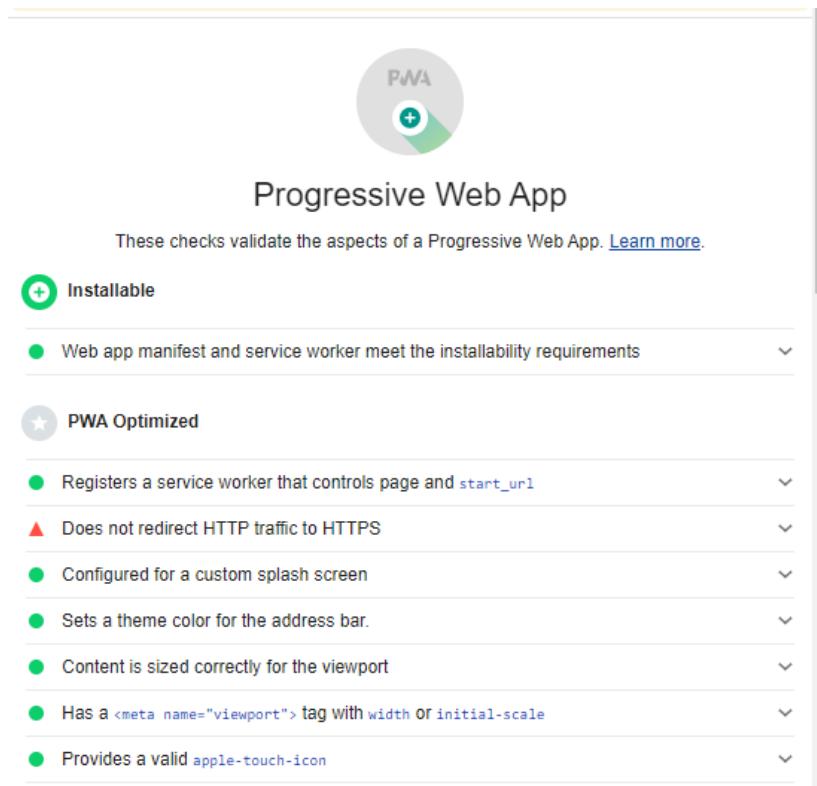


Fig. 1: Raport Lighthouse

Ca și framework-uri și pachete suplimentare pentru partea de front end am folosit Axios pentru a face request-uri către server și React Router pentru a putea imita senzația unei aplicații mvc și pentru a putea avea linkuri către proiectele create de utilizatori.

De asemenea am folosit un serviciu extern numit Firebase pentru a putea crea o autentificare cu telefonul și am creat folosind React Router și hook-ul UseContext rute autorizate pentru pagina de Home și Projects.

Totodată am folosit Tailwind ca framework de CSS, cu ajutorul căruia am reușit să dezvolt foarte repede partea de UI. Această parte a urmat evident design-ul din Figma, având în final următoarea formă surprinsă mai jos în Figura 2: ecrane aplicație.

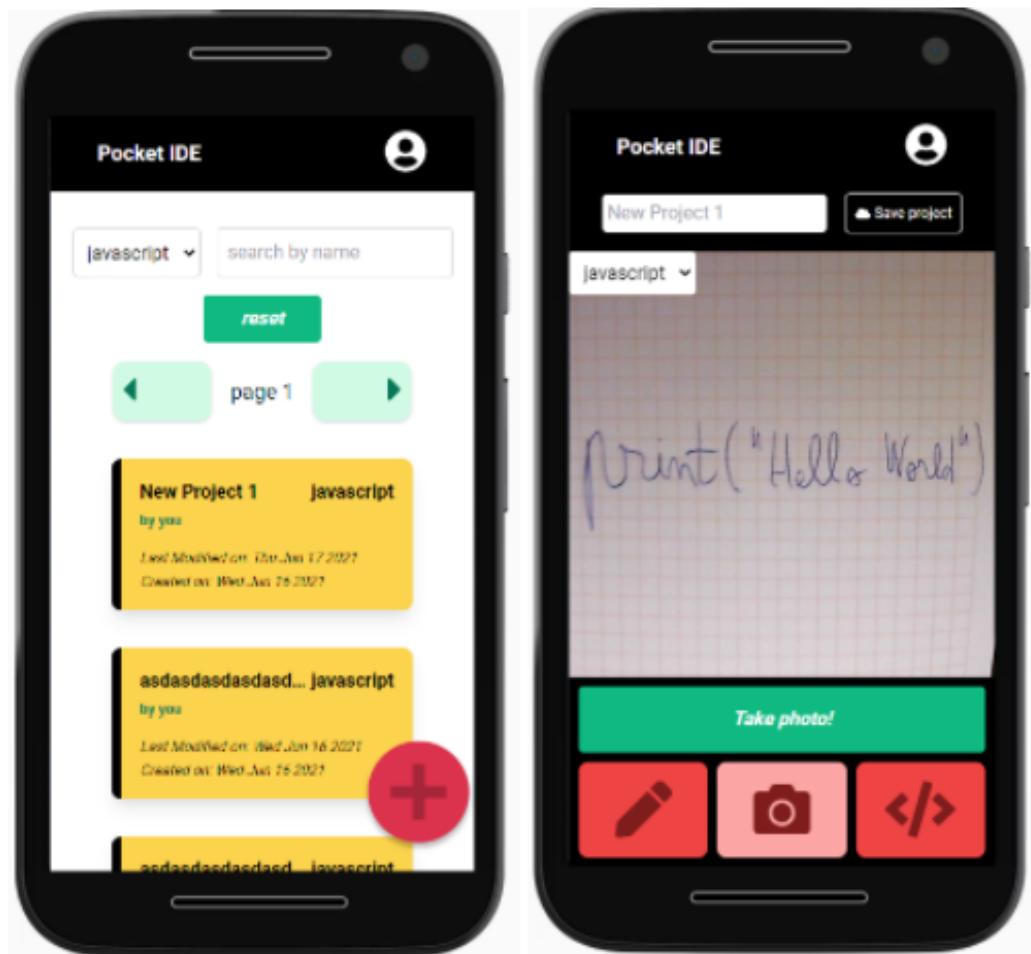


Fig. 2: ecrane aplicație

# Concluziile lucrării

Forma finală a proiectului poate fi observată mai jos în Figura 1: Arhitectura Finală.

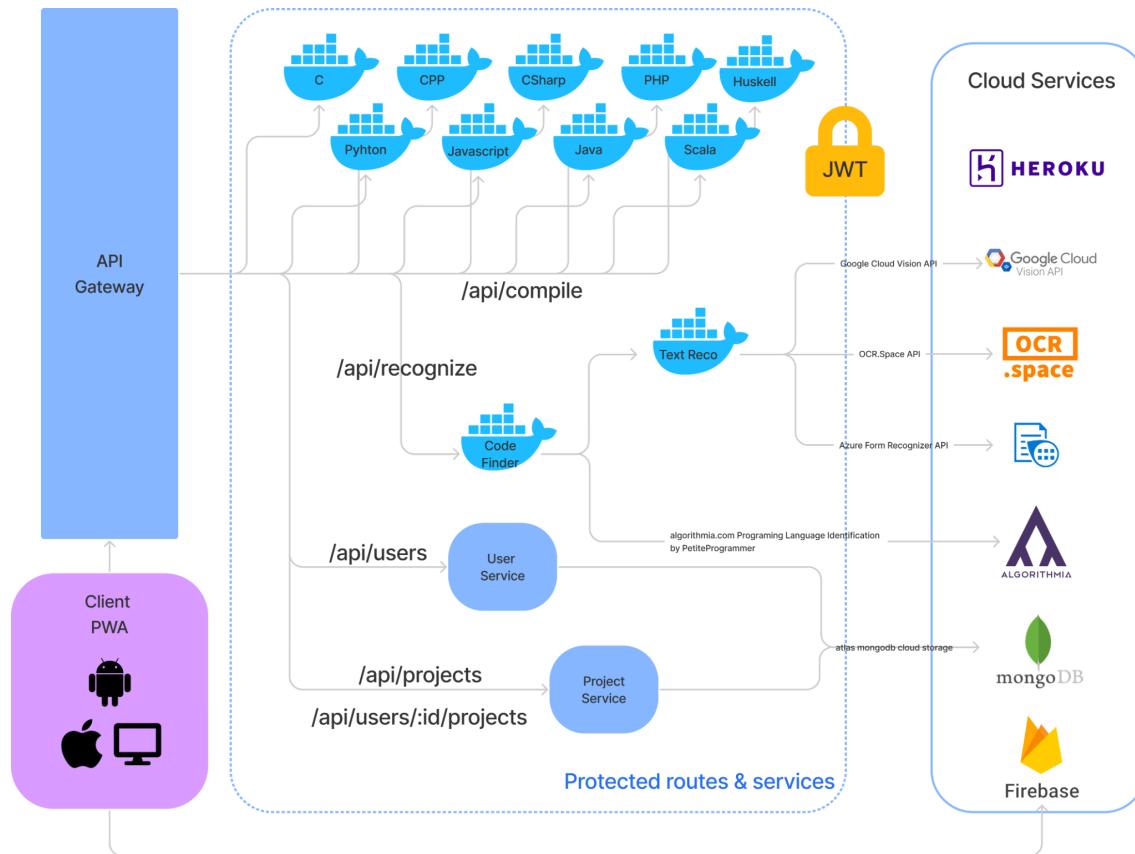


Fig. 1: Arhitectura Finală

În final, PocketIDE este o aplicație cu o arhitectură bazată pe microservicii, fiind dezvoltată folosind metoda de Test Driven Development.

Pe partea de client aceasta folosește principiile și tehnologiile Progressive Web Application și cu ajutorul a CSS, React și service workers oferă o interfață atractivă și performantă ce poate fi utilizată pe orice platformă, fiind descărcabilă și funcțională în mediul offline.

Pe partea de server, microserviciile asigură modulizarea și extensibilitatea sistemului, iar folosirea de middleware asigură securitate și control asupra entităților care interacționează cu acestea.

Pentru partea de recunoaștere a scrisului sunt folosite servicii cloud care să asigure acuratețea soluției.

Iar pentru partea de compilare și rulare a codului a fost adăugat suport pentru un set larg de limbaje printre care se enumera: C, C++, Python, Java, Javascript, Scala, Haskell, PHP.

În opinia mea, proiectul a fost finalizat cu succes, îndeplinind toate cerințele funcționale și nonfuncționale care au fost propuse în descrierea problemei.

O potențială direcție de viitor ar fi finisarea proiectului și crearea unui parteneriat cu Facultatea de Informatică pentru a fi, după cum am precizat și în motivație, folosit în sistemul educativ de către elevi și studenți, pentru a eficientiza timpul necesar testării și rulării algoritmilor scriși de mână.

# Bibliografie

[1] Figură stilou de tip stylus:

<https://image.pushauction.com/0/0/aac4e237-8491-4332-8546-a0a9c8e8eefa/d01dca83-52f5-4cd7-9356-bab7b8bb81b8.jpg>

[2] Figură photomath:

<https://ohhmybug.com/wp-content/uploads/2018/04/Photomath3.jpg>

[3] Figură etape de rezolvare problemă, photomath: <https://i.redd.it/l245ik09vp811.jpg>

[4] Articol Tesseract: <https://golb.hplar.ch/2019/07/ocr-with-tesseractjs.html>

[5] Ce e un technology stack:

<https://www.fingent.com/blog/top-6-tech-stacks-that-reign-software-development-in-2020/>

[6] Figură utilizare Android vs IPhone la nivel global:

<https://www.ajournalofmusicalthings.com/are-you-an-ios-or-android-person-heres-how-the-global-battle-shapes-up/>

[7] Figură Top 5 Framework-uri multi-platformă pentru mobile development:

<https://blog.testproject.io/2020/04/06/top-5-cross-platform-mobile-development-frameworks-comparison/>

[8] Figură Screenshot thread Github

<https://github.com/react-native-community/discussions-and-proposals/issues/64>

[9] Descriere Progressive web apps (PWAs):

[https://developer.mozilla.org/en-US/docs/Web/Progressive\\_web\\_apps](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps)

[10] Flux dezvoltare bazată pe testare:

<https://marsner.com/blog/why-test-driven-development-tdd/>

[11] Fig Monolit vs Microservicii:

[https://dev.to/alex\\_barashkov/microservices-vs-monolith-architecture-4l1m](https://dev.to/alex_barashkov/microservices-vs-monolith-architecture-4l1m)

[12] Figură wireframe vs mockup vs prototype:

<https://www.aha.io/roadmapping/guide/product-management/wireframe-mockup-prototype>

[13] Descriere MNIST:

[https://en.wikipedia.org/wiki/MNIST\\_database](https://en.wikipedia.org/wiki/MNIST_database)

[14] Figură exemple MNIST:

[https://en.wikipedia.org/wiki/MNIST\\_database#/media/File:MnistExamples.png](https://en.wikipedia.org/wiki/MNIST_database#/media/File:MnistExamples.png)