

**MINISTERUL EDUCAȚIEI**  
**LICEUL TEORETIC "GRIGORE MOISIL" TIMISOARA**

**TITLUL: DOCScan**

**PROFESORI COORDONATORI:**

- IORDAICHE CRISTINA
- ROŞU OVIDIU

**CANDIDAT:** Vlad-Andrei Tomici

SESIUNEA MAI 2024



# CUPRINS

<i>CUPRINS</i>	3
<i>MOTIVAREA ALEGERII TEMEI</i>	4
<i>STRUCTURA LUCRĂRII</i>	5
<i>Resurse utilizate pentru dezvoltare</i>	5
<i>Arhitectura aplicatiei</i>	6
<i>IMPLEMENTAREA TEHNICĂ</i>	7
<i>Crearea aplicației</i>	7
<i>Walkthrough prin aplicație</i>	8
<i>SISTEMUL DE LOGARE / AUTENTIFICARE</i>	10
<i>Arhitectura sistemului</i>	10
<i>Baza de date – integrarea Firebase Auth si Cloud Firestore</i>	10
<i>Integrarea Firebase</i>	11
<i>Implementarea software</i>	14
<i>SCANAREA DE DOCUMENTE CU CAMERA</i>	17
<i>Conversia de la imagine in digital</i>	17
<i>Aplicarea de filtre asupra scanarilor</i>	19
<i>Implementarea software</i>	20
<i>EXTRAGEREA DE TEXT DIN ARIA DE INTERES</i>	23
<i>Procesarea de text – folosirea de NLP</i>	23
<i>Post-procesarea de text si afisarea acestuia in aplicatie</i>	24
<i>Implementarea software</i>	24
<i>BIBLIOGRAFIE</i>	27

# MOTIVAREA ALEGERII TEMEI

Ideea DOCScan a pornit din dorința de a avea o aplicație mobilă pentru scanarea rapidă de documente. Pe lângă acest feature fundamental, dorința de lucru dinamic digital cu aceste documente a apărut și ea, astfel că aplicația aduce și funcționalități care să permită lucrul activ cu documente.



Aplicația elimină nevoia de scanare tradițională prin intermediul dispozitivelor separate și de conversie manuală a textului în digital. Tot ce are nevoie utilizatorul pentru a scana documente este telefonul mobil, alături de aplicația DOCScan. Potențialul de utilizare este mare, având o varietate de domenii în care poate să fie utilizată, precum educație, business, administrație sau medicină. De asemenea, ea este ideală pentru studenți, profesori, funcționari, antreprenori, pentru oricine are nevoie să gestioneze documente în mod frecvent.

DOCScan aduce un impact pozitiv în :

## 1. Eficiență

- ▶ crește productivitatea și eficiență în gestionarea documentelor
- ▶ automatizează procesul de extragere a textului în digital, economisind timp și efort
- ▶ facilitează organizarea documentelor digitale și stocarea lor eficientă

## 2. Accesibilitatea

- ▶ facilitează accesul la informații pentru diverse categorii de informații
- ▶ permite căutarea rapidă în documentele scanate și găsește rapid informații specifice

## 3. Durabilitate

- ▶ reduce dependența de documente fizice, contribuind la conservarea mediului

DOCScan este o aplicație mobilă care rulează exclusiv pe sistemul de operare iOS, fiind nevoie de o versiune de iOS >= 17.0.0

# STRUCTURA LUCRĂRII

DOCScan a fost structurată pe următoarele componente fundamentale:

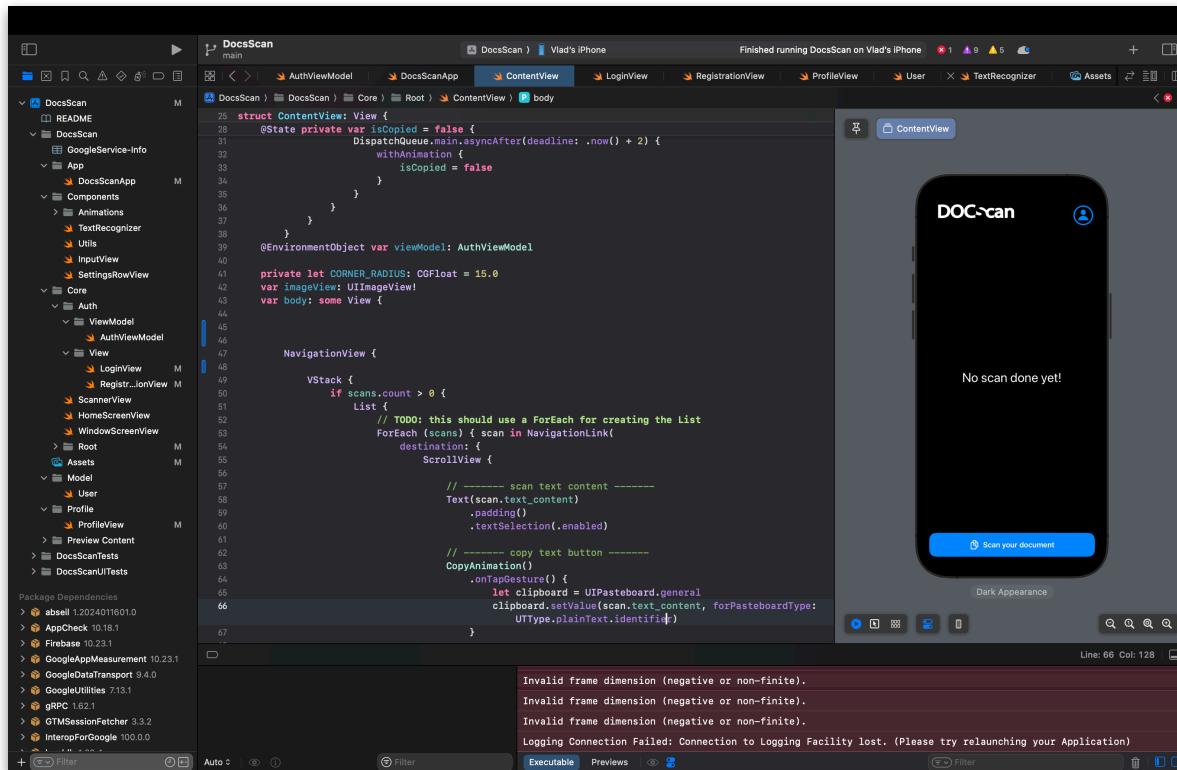
1. Sistemul de logare / autentificare a utilizatorilor
2. Sistemul de scanare a documentelor
3. Sistemul de detectare și extragere a textului din scanări

Toate aceste componente au fost dezvoltate pe rând, ca mai apoi să fie îmbinate în aplicație.

## Resurse utilizate pentru dezvoltare

Intreaga aplicație a fost scrisă în **Xcode**, una dintre principalele IDE-uri care se poate dezvolta software pentru dispozitivele Apple. Aceasta este un IDE, care oferă un set complet de instrumente pentru dezvoltarea software-ului, precum instrumente pentru scrierea codului (editor cu completare automată și verificare de sintaxă), proiectare vizuală a interfeței aplicației, depanare (identificare și remedierea erorilor), testare și livrarea aplicației către App Store.

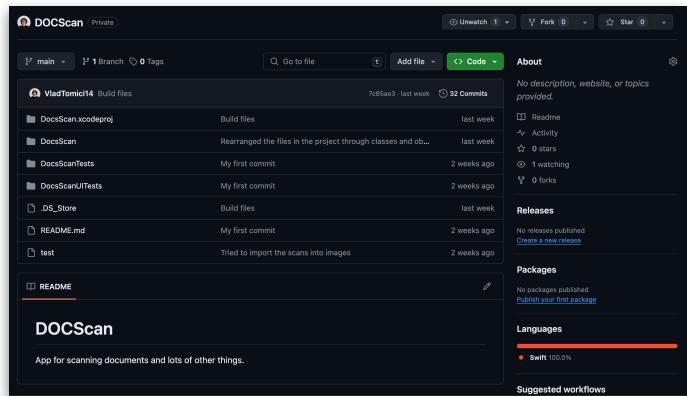
Xcode dispune, de asemenea, de un simulator care rulează aplicația pe un dispozitiv virtual, permitând testarea rapidă, fără să fie nevoie de conectarea la un dispozitiv fizic. (Img. 1)



(Img. 1) Interfața Xcode

Toate resursele media de tip icon au fost importate din colecția vastă de pictograme vectoriale **SF Symbols**.

Am utilizat **Git**, alături de **GitHub**, pentru întreagă versionare a codului. Astfel, am putut salva constant copii ale codului. Versiunile constant upgradeate ale aplicației pot să fie găsite aici: <https://github.com/VladTomici14/DOCScan>



Toate logo-urile și resursele media ale aplicației DOCScan au fost create de către **Tania Tomici**. (Img. 2) (Img. 3) (Img. 4)

**DOCscan**



**DOCscan**

(Img. 2) Logo text black

(Img. 3) Icon aplicatie

(Img. 4) Logo text blue

## Arhitectura aplicatiei

Întregul proiect este dezvoltat pe structura de clase și obiecte.

Pentru fiecare ecran al aplicației, am construit câte o clasă separată, care doar primește diverse informații și le randează. În principiu, întâlnim câte o clasă de genul pentru:

- profilul utilizatorului – primește din database detalii despre utilizator și le afișează
- o clasă pentru reprezentarea scanării făcute din baza de date

Din punct de vedere al modului de lucru, fiecare variabilă, obiect, fișier sau funcție, a primit un nume corespunzător funcționalității sale. Astfel, am adus o ușurință în parcursarea codului și soluționarea erorilor sau a bug-urilor.

# IMPLEMENTAREA TEHNICĂ

## *Crearea aplicației*

Aplicația este dezvoltată cu ajutorul limbajului de programare *Swift*, în strânsă legătură cu framework-ul vizual *SwiftUI*.

Swift este un limbaj de programare orientat pe obiecte (OOP). Deși sintaxa acestuia este mai concisă decât alte limbi care folosesc OOP (precum Java sau C++), principiile programării orientate pe obiecte rămân fundamentale și pentru Swift.

**Swift** este un limbaj de programare modern, creat de compania Apple, în anul 2014, special pentru a facilita crearea de aplicații iOS și macOS. Este apreciat pentru:

- ▶ Performanță - Codul este compilat direct în limbaj-mașină, rezultând aplicații rapide și eficiente
- ▶ Simplitate - Sintaxa să este ușor de învățat și utilizat
- ▶ Siguranță - Swift pune accent pe siguranța memoriei și previne erorile comune, sporind stabilitatea aplicațiilor
- ▶ Interoperabilitatea - Funcționează perfect cu Objective-C, permitând integrarea cu cod deja existent



**SwiftUI** este un framework vizual, lansat în anul 2019, care simplifică semnificativ crearea de interfețe grafice intuitive și interactive, acesta oferind:

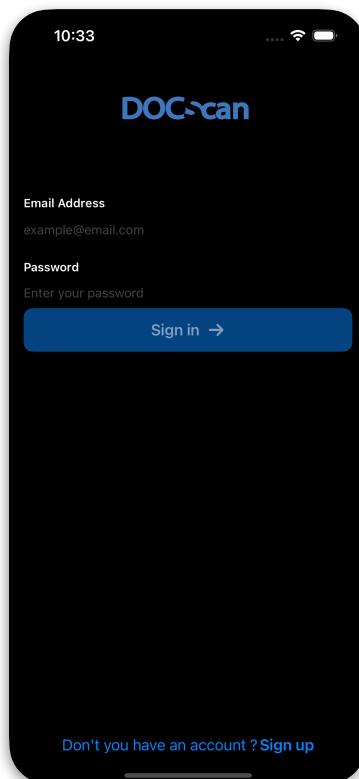
- ▶ Compozibilitate - Permite combinarea cu ușurință a componentelor vizuale individuale pentru crearea de interfețe complexe
- ▶ Preview live - Vizualizarea imediată a modificărilor aduse interfeței grafice, accelerând procesul de dezvoltare
- ▶ Animații fluide - Permite crearea de animații naturale și captivante
- ▶ Declarativitatea - Oferă un flux de lucru mai concis și mai intuitiv

Cu ajutorul acestui framework, aplicația a reusit să dobândească o interfață modernă și curată.

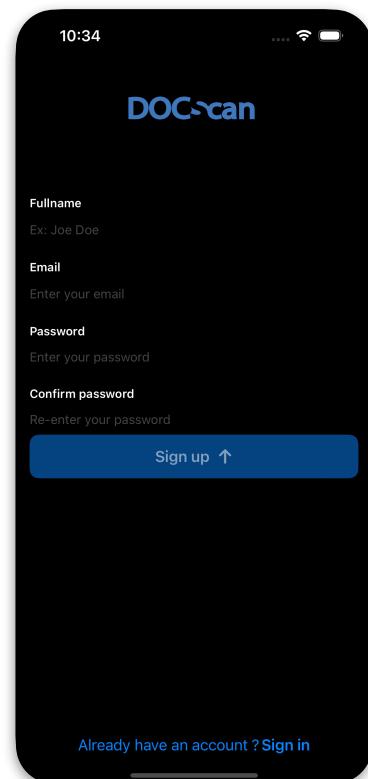
## Walkthrough prin aplicație

În momentul în care deschidem aplicația, primul aspect care va întâmpina utilizatorii este ecranul de primire. Aici, utilizatorii pot afla mai multe despre aplicație, în cazul în care nu sunt logați.

Ulterior, utilizatorul poate alege dacă își dorește să se logheze sau să își creeze un cont în aplicație.



(Img. 5) Ecranul de autentificare



(Img. 6) Ecranul de creare cont

În cazul în care utilizatorul dorește să se logheze, apasând butonul "Already have an account? Sign in", acesta va fi purtat pe ecranul de autentificare, unde poate să își introducă email-ul și parola. (Img. 5)

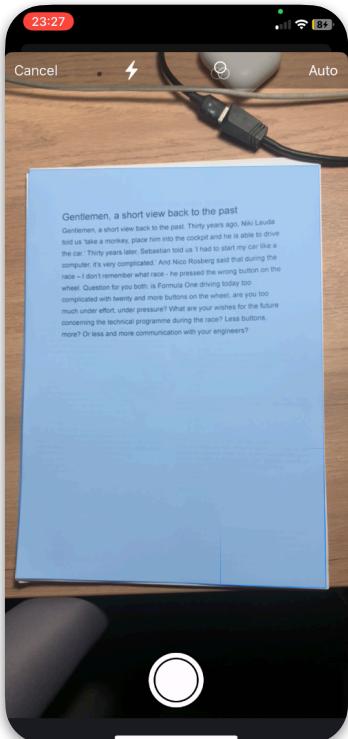
În caz contrar, dacă utilizatorul nu are un cont creat, acesta poate să facă asta cu usurință apasând butonul "Don't you have an account? Sign up". (Img. 6)

În cazul în care respectivul buton este apăsat, utilizatorul își poate crea un cont nou. Acestui i se vor cere date pentru creare. Dupa introducerea lor, acesta va avea acces în platformă. (Img. 7)



(Img. 7) Pagina de primire după logare

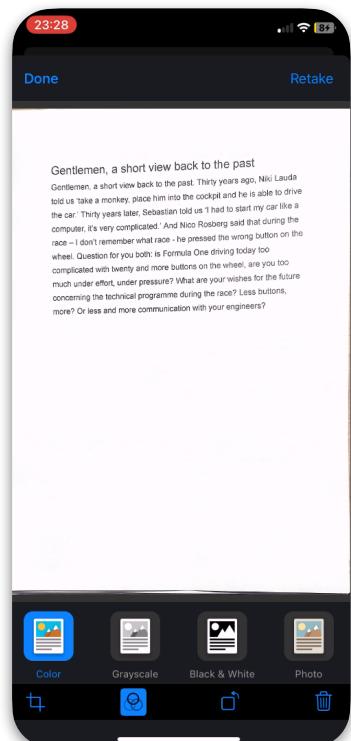
Pentru a face o scanare, utilizatorul trebuie sa apese butonul "Scan your document". (Img. 7)



Acesta va trebui sa pozitioneze telefonul deasupra documentului. Detectarea si efectuarea pozei se va face in mod automat. Zona albastra va reprezenta zona in care documentul este identificat. Practic, ea determina Aria De Interes a imaginii de intrare. (Img. 8) Dupa ce scanarea a fost efectuata, ea va aparea in colt stanga jos.

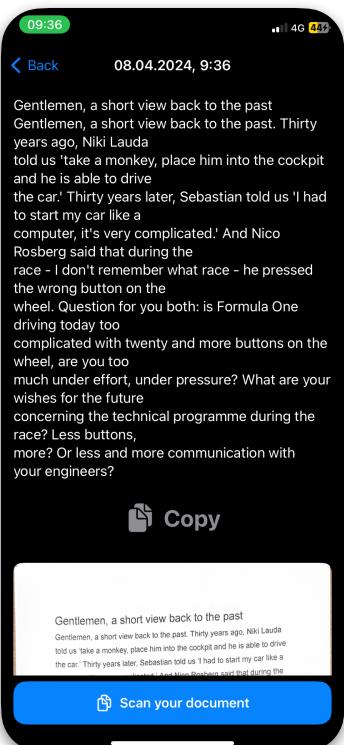
In cazul in care utilizatorul doreste sa aduca modificarile asupra acesteia, putem apasa pe stack-ul de imagini din partea stanga-jos a ecranului.

Pentru fiecare scanare facuta, putem sa editam modul in care se da crop la document, care este filtrul de culoare final sau daca dorim sa rotim scanarea. (Img. 9)



(Img. 8) Detectarea paginii

Odacă ce am terminat de adus modificări asupra documentului, putem apasa pe "Save". Butonul acesta marchează finalul procesului de scanare a documentului.



Dupa terminarea procesului de scanare, aplicatia urmeaza sa aiba o noua intrare generata cu ora si data efectuarii scanarii.

(Img. 9) Meniul de editare a scanarilor

Daca apasam pe aceasta, se poate vedea textul recunoscut si scanarea efectuata. In cazul in care utilizatorul doreste, poate copia rapid intregul continut al textului apasand butonul "Copy". (Img. 10)

Astfel, atat scanarea cat si textul detectat sunt salvate in cloud-ul DOCScan.

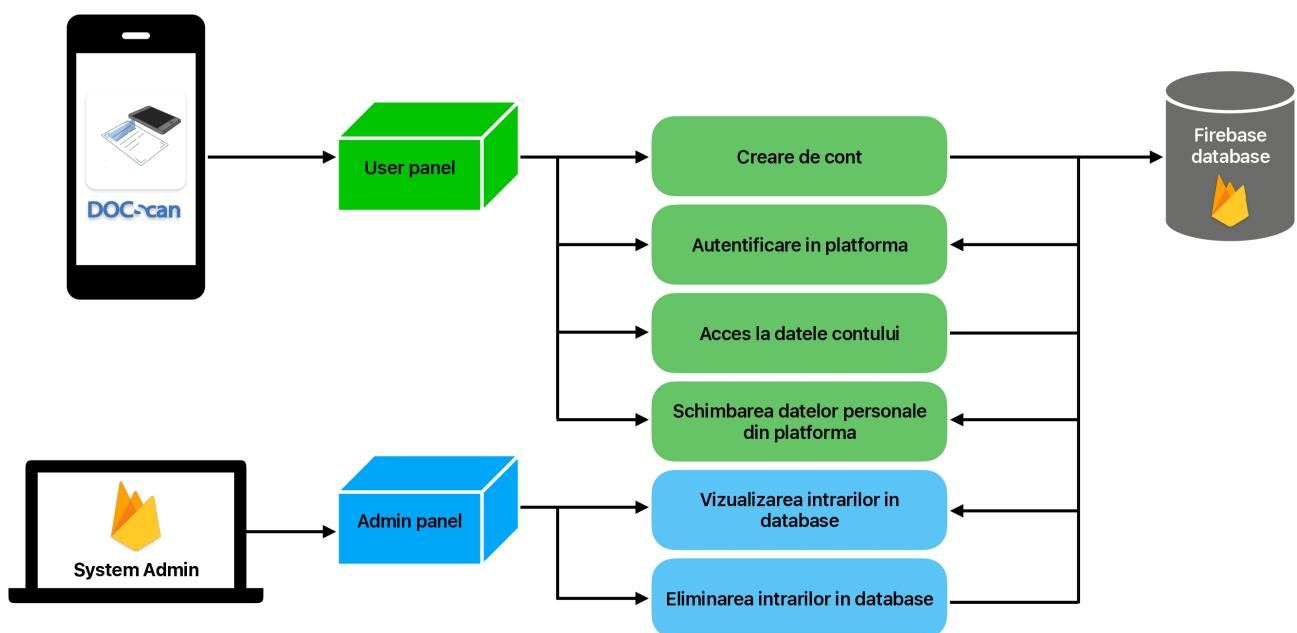
(Img. 10) Textul extras in urma recunoasterii

# SISTEMUL DE LOGARE / AUTENTIFICARE

In DOCScan, fiecare utilizator isi poate crea un cont.

Beneficiul adus este ca toate scanarile din aplicatie sunt salvate in cloud, permitand accesul la acestea de pe orice dispozitiv.

## Arhitectura sistemului



(Img. 11) Diagrama intregului sistem de autentificare

Pentru fiecare utilizator, in primul rand se vor salva in baza de date urmatoarele detalii despre acesta: numele intreg, email-ul si parola. Ele sunt folosite pentru autentificarea / logarea utilizatorului in contul sau.

De asemenea, fiecare utilizator va avea salvate intrarile sale in urma scanarilor.

## Baza de date – integrarea Firebase Auth si Cloud Firestore

Firebase este o platforma de dezvoltare pentru aplicatii mobile si web de la Google. Aceasta ofera o suita de instrumente care ajuta dezvoltatorii software sa construiasca, ruleze si sa analizeze aplicatiile mai usor.

Firebase se ocupa cu infrastructura serverului, astfel incat dezvoltatorii sa se poata concentra pe crearea de functionalitati ale aplicatiei. Aceasta ofera baze de date de tip NoSQL, autentificare pentru utilizatori, stocare in cloud si multe altele.

De asemenea, Firebase ofera o serie de caracteristici de securitate si confidentialitate intergrate care ajuta la protejarea datelor utilizatorilor.

Firebase cripteaza toate datele si tranzit si in repaus. Datele cu caracter senzitiv ale utilizatorului sunt salvate in cloud-ul DOCScan, utilizatorul fiind singura persoana care are acces la acestea.



Firebase Auth este un sub-serviciu, inclus in Firebase. Acesta ofera o gama larga de optiuni de autentificare, de la email si parola, pana la autentificare prin Google, Facebook, Apple ID sau prin numarul de telefon.

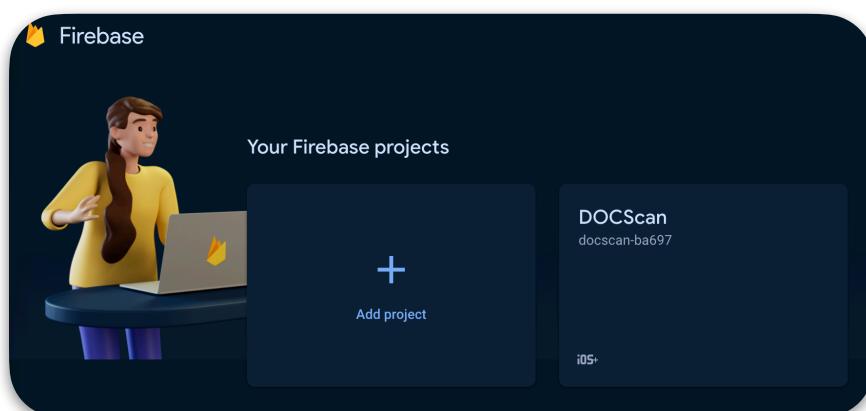
Firebase Realtime Database si Cloud Firestore ofera reguli de securitate bazate pe roluri care permit controlarea accesului si modificarea de date. Firestore, cunoscut si ca Cloud Firestore, este serviciul ce se ocupa de managment-ul de baze de date. Toate datele din sistemul de logare prin Firebase Auth, urmeaza sa fie salvate in Firestore, alaturi de toate scanarile facute de fiecare utilizator si de textul detectat in urma scanarii.

## **Integrarea Firebase**

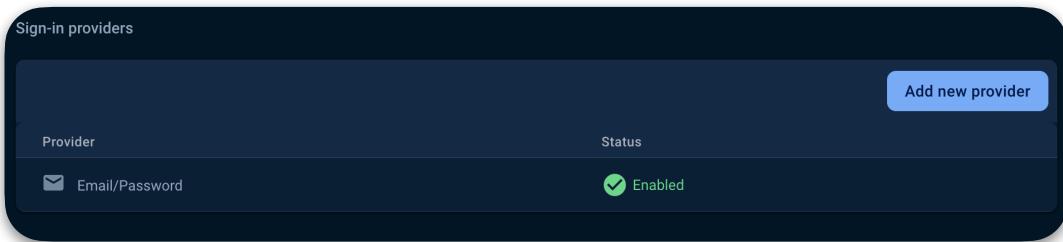
Inainte de toate, va fi nevoie sa ne cream un proiect nou in consola Firebase, pe care o gasim aici:

<https://console.firebaseio.google.com/>.

Tot din consola Firebase urmeaza sa ne denumim proiectul, sa setam un sistem de logare, si nu in ultimul rand, sa cream baza de date a aplicatiei.



Pentru sistemul de autentificare, am ales ca singura metoda sa fie cea pe baza de email, in viitor dorind sa adaug si autentificarea prin servicii third-party. (Img. 12)



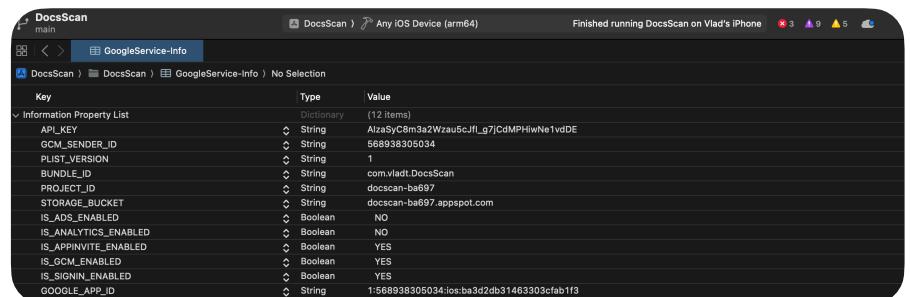
(Img. 12) Adaugarea de authentication providers din Firebase console

Pentru setarea Cloud Firestore, putem seta manual field-urile bazei de date, sau le putem introduce dinamic din intermediul aplicatiei. Pentru DOCScan, am preferat cea de-a doua implementare. (Img. 13)

Key	Type	Value
email	String	'bogalima6@gmail.com'
fullname	String	'Marian'
id	String	'pRXblyJN0vOoyifeCDKCRyXpFy72'

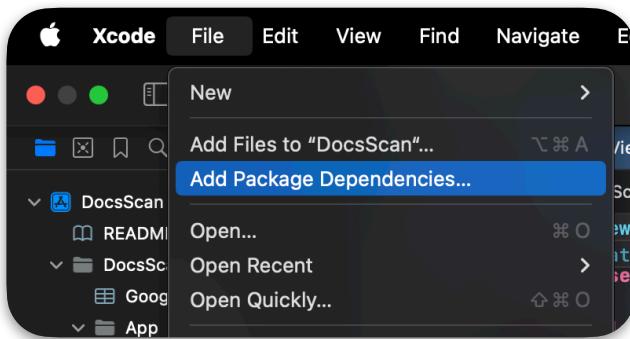
(Img. 13) Vizualizarea de database

Urmeaza ca Firebase sa ne genereze fisierul `GoogleService-Info.plist`, care contine toate detaliiile ce ne leaga aplicatia de Firebase Auth si Cloud Firestore din consola proiectului nostru creat in Firebase. (Img. 14)

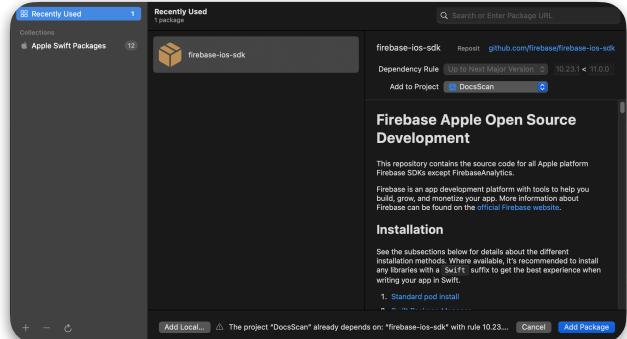


(Img. 14) Vizualizarea GoogleService-Info.plist

Odata ce am reusit sa facem asta, tot ce ramane este sa sa importam packages-urile necesare. Putem sa facem asta direct din Xcode. (Img. 15) (Img. 16)



(Img. 15) Adaugarea de dependencies in proiect



(Img. 16) Adaugarea Firebase in proiect

Acum ca am reusit sa instaliam toate dependențe-urile necesare pentru folosirea Firebase, tot ce ne ramane este sa initializam Firebase in aplicatia noastra. O sa facem asta in fisierul principal al aplicatiei DocsScanApp.swift.

```
DocsScanApp.swift

1 import SwiftUI
2 import Firebase
3
4 @main
5 struct DocsScanApp: App {
6
7     @StateObject var viewModel = AuthViewModel()
8
9     init() {
10         FirebaseApp.configure()
11     }
12
13     var body: some Scene {
14         WindowGroup {
15             ContentView()
16                 .environmentObject(viewModel)
17                 .preferredColorScheme(.dark)
18         }
19     }
20 }
21
```

In acest fisier, incepem prin a importa SwiftUI (framework-ul necesar pentru a crea interfețe grafice) și Firebase.

Decoratorul @main specifică faptul că structura DocsScanApp este punctul de intrare principal al aplicatiei.

In primul rand, urmează să cream o instanță a clasei AuthViewModel, care îi atribuim proprietatea viewModel. Clasa AuthViewModel este cea care gestionează autentificarea utilizatorului, alături de datele aplicatiei.

Initializatorul init() al aplicatiei apelează FirebaseApp.configure() pentru a configura conexiunea cu Firebase. În final proprietatea default var body: some Scene { } este cea care definește fereastra principală de vizualizare a aplicatiei.

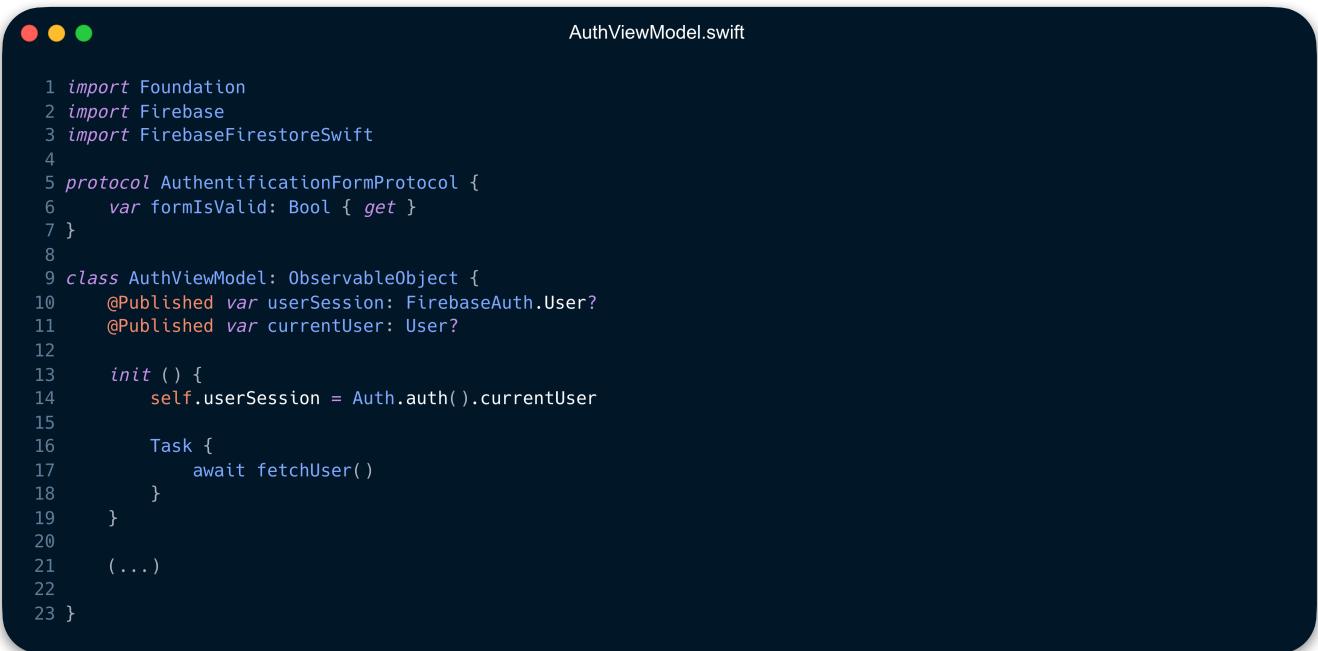
Astfel, am reusit să initializăm și să conectăm Firebase la aplicatie.

## Implementarea software

Intregul sistem de logare / autentificare este structurat pe mai multe clase si obiecte.

Principala clasa care contine toate functionalitatile este `AuthViewModel.swift`. Urmeaza sa importam librariile Foundation, Firebase si `FirebaseFirestoreSwift`.

Pentru tipul clasei se foloseste protocolul `ObservableObject` pentru a notifica interfata grafica ori de cate ori proprietatile sale publicate se modifica.



```
AuthViewModel.swift

1 import Foundation
2 import Firebase
3 import FirebaseFirestoreSwift
4
5 protocol AuthenticationFormProtocol {
6     var formIsValid: Bool { get }
7 }
8
9 class AuthViewModel: ObservableObject {
10     @Published var userSession: FirebaseAuth.User?
11     @Published var currentUser: User?
12
13     init () {
14         self.userSession = Auth.auth().currentUser
15
16         Task {
17             await fetchUser()
18         }
19     }
20
21     (...)
```

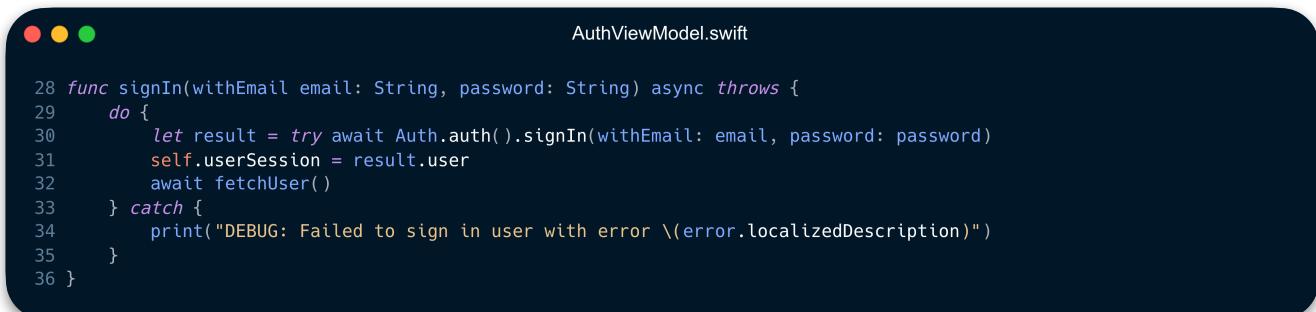
Variabila `userSession` stocheaza informatiile despre utilizatorul curent conectat, preluate din Firebase Auth. Aceasta este marcata cu `@Published`, ceea ce inseamna ca orice modificari asupra ei vor actualiza automat interfata grafica.

Variabila `currentUser` stocheaza date detaliate despre utilizator, preluate din Firestore dupa o autentificare cu succes. De asemenea, si ea este marcata cu `@Published`.

In initializarea clasei, avem un singur parametru, acesta fiind utilizatorul curent conectat la Firebase Auth folosind `Auth.auth().currentUser` si il atribuie lui `userSession`. Nu in ultimul rand, apeleaza asincron functia `fetchUser()` pentru a prelua date detaliate despre utilizator din Firestore.

Tot in aceasta clasa, avem prezente mai multe functii care aduc functionalitate sistemului de logare / autentificare.

Functia `signIn()` este asincrona, ea incercand sa autentifice un utilizator cu o adresa de email si o parola. Daca are succes, ea actualizeaza `userSession` cu informatiile utilizatorului conectat. Ulterior, apeleaza `fetchUser()` pentru a prelua date detaliate despre utilizator din Firestore.

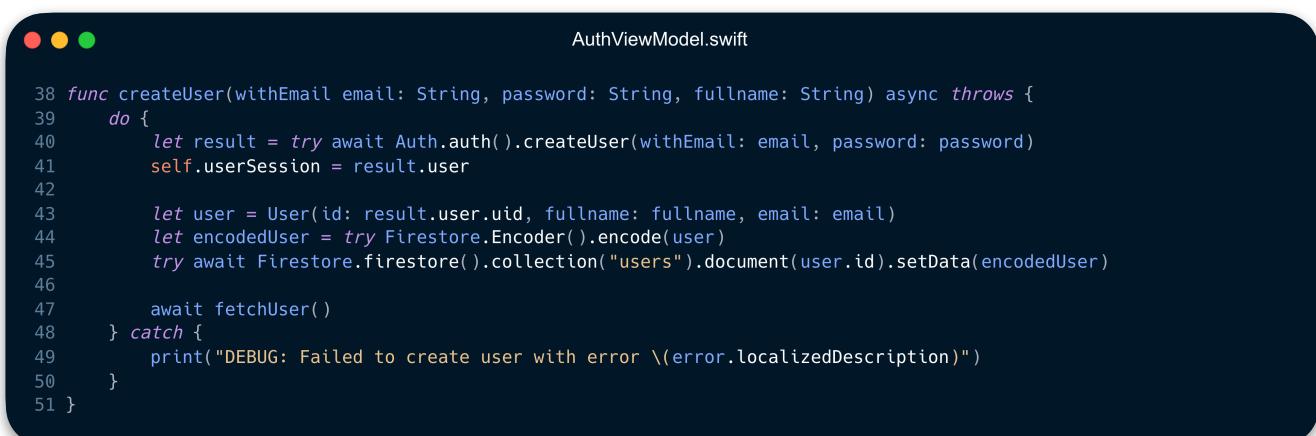


```
AuthViewModel.swift

28 func signIn(withEmail email: String, password: String) async throws {
29     do {
30         let result = try await Auth.auth().signIn(withEmail: email, password: password)
31         self.userSession = result.user
32         await fetchUser()
33     } catch {
34         print("DEBUG: Failed to sign in user with error \(error.localizedDescription)")
35     }
36 }
```

Functia `createUser()` este de asemenea asincrona si este responsabila pentru crearea unui cont nou de utilizator. Ea incearca sa creeze un nou utilizator in Firebase Auth. Daca are succes, actualizeaza `userSession` cu informatiile noului utilizator creat. De asemenea, creaza un obiect `User` care contine detalii suplimentare despre utilizator, cum ar fi numele complet al acestuia. Ulterior, codifica obiectul `User` intr-un format compatibil cu Firestore folosind metoda `Firestore.Encoder()`.

Functia salveaza dupa datele encriptate ale utilizatorului in Firestore sub colectia "utilizatori" cu un ID unic. In final, apeleaza functia `fetchUser()` pentru a prelua date detaliate despre utilizator din Firestore.



```
AuthViewModel.swift

38 func createUser(withEmail email: String, password: String, fullname: String) async throws {
39     do {
40         let result = try await Auth.auth().createUser(withEmail: email, password: password)
41         self.userSession = result.user
42
43         let user = User(id: result.user.uid, fullname: fullname, email: email)
44         let encodedUser = try Firestore.Encoder().encode(user)
45         try await Firestore.firestore().collection("users").document(user.id).setData(encodedUser)
46
47         await fetchUser()
48     } catch {
49         print("DEBUG: Failed to create user with error \(error.localizedDescription)")
50     }
51 }
```

Functia `signOut()` deconecteaza utilizatorul curent. Incercarea se face prin intermediul linii 55 de cod `Auth.auth().signOut()`. Ulterior, seteaza `userSession` si `currentUser` la valoarea `nil` (valoarea nula in Swift) pentru a sterge informatiile sesiunii si datele de autentificare ale utilizatorului.

```
AuthViewModel.swift
```

```
53 func signOut() {  
54     do {  
55         try Auth.auth().signOut() // signs out user on backend  
56         self.userSession = nil // wipes out user session and takes us back on login screen  
57         self.currentUser = nil // wipes out currentUser data  
58     } catch {  
59         print("DEBUG: Failed to sign out with error \(error.localizedDescription)")  
60     }  
61 }
```

Functia `fetchUser()` este asincrona si preia date detaliate despre utilizator din Firestore. Ea verifica daca un utilizator este conectat in prezent prin intermediul `Auth.auth().currentUser?.uid`, urmand sa recupereze datele utilizatorului din Firestore folosind ID-ul unic.

```
AuthViewModel.swift
```

```
67 func fetchUser() async {  
68     guard let uid = Auth.auth().currentUser?.uid else { return }  
69     guard let snapshot = try? await Firestore.firestore().collection("users").document(uid).getDocument() else {  
70         return }  
71     self.currentUser = try? snapshot.data(as: User.self)  
72     print("DEBUG: Current user is \(self.currentUser)")  
73 }
```

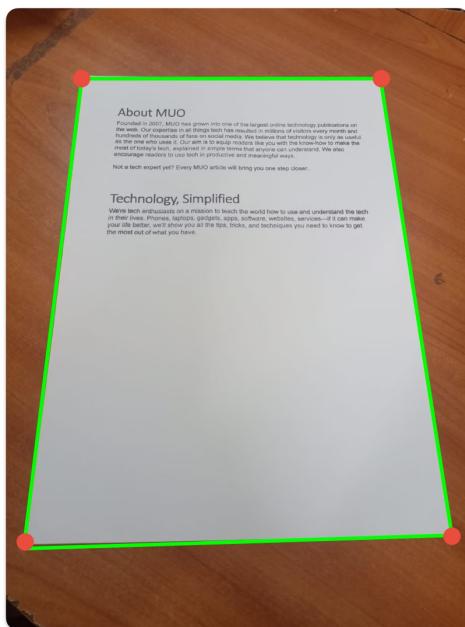
Daca recuperarea este reusita, va incerca sa decodeze datele preluate inapoi printr-un obiect `User` folosind `snapshot.data(as: User.self)`. In final, seteaza proprietatea `currentUser` cu datele decodificate ale utilizatorului.

In cazul in care apar erori in timpul recuperarii din oricare din functiile de mai sus, nu se mai returna nimic, datele nefiind modificate, iar eroarea va fi printata in build terminal-ul aplicatiei.

# SCANAREA DE DOCUMENTE CU CAMERA

Scanarea de documente cu camera este compusa din 2 tehnici ce trebuie implementate:

1. Identificarea Ariei De Interes a paginii – ea reprezinta zona din imagine in care se afla documentul pe care dorim sa il scanam
2. Aplicarea de filtre asupra scanarilor – ajuta la transformarea imaginii capturate intr-un format mai curat si mai prietenos cu mediul digital.



Imaginea de intrare



Imaginea finală

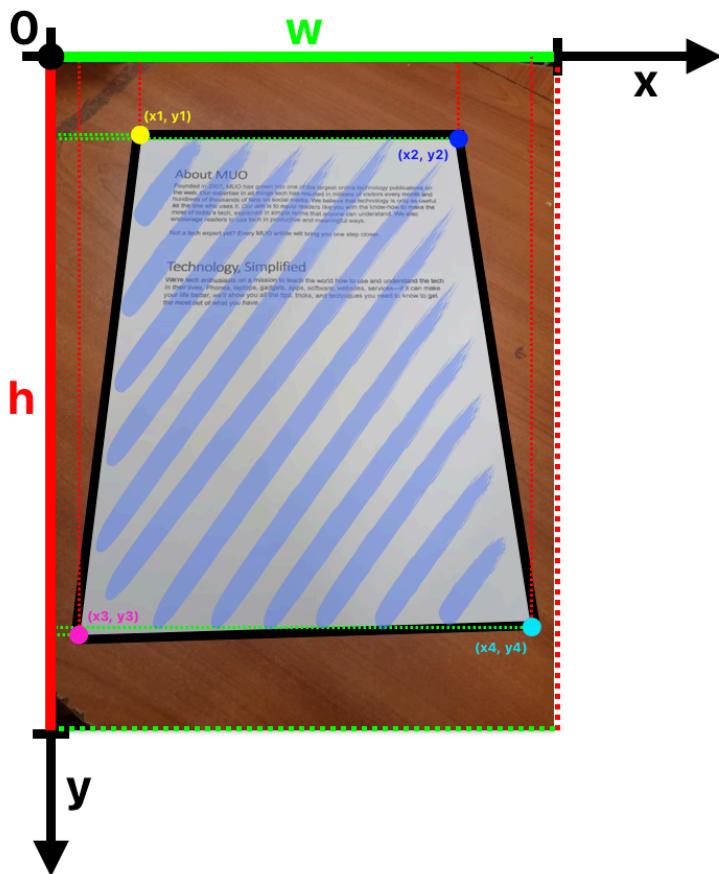
(conturul verde reprezintă Aria De Interes determinată de aplicație)

## Conversia de la imagine in digital

Aplicatia isi doreste mai intai de toate sa identifice toate colturile paginii pe care dorim sa o scana.

Pentru acesta, o sa cream o axa de tip ( $xOy$ ) pentru poza efectuata de aplicatie. Folosindu-ne de niste algoritmi de detectie a pozitionarii paginei in cadrul imaginii. (Img. 17)

Cu ajutorul punctelor, deja putem identifica *Aria De Interes (ADI)* al imaginii, aceasta fiind documentul pe care dorim sa il scanam.

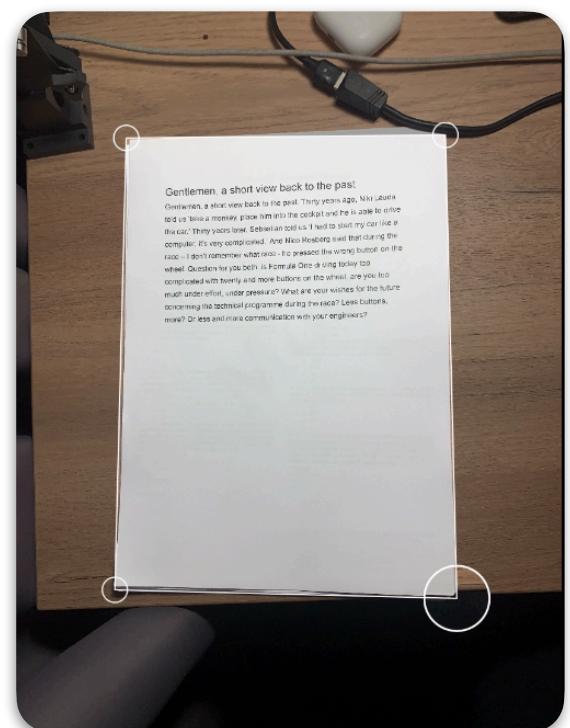


(Img. 17) Reprezentarea ( $xOy$ ) in raport cu imaginea

In cazul in care apar anumiti factori ce pot ingreuna procesul de efectuare a imaginii de intrare, aplicatia se poate sa nu fie capabila sa identifice ADI in mod corespunzator. Astfel, in cazul in care utilizatorul nu este multumit de modul in care ADI a fost selectat, acesta poate sa aleaga modul manual de identificare ADI. (Img. 18)

Odata ce reusim sa avem ADI pentru input image-ul pe care l-am dat, urmeaza ca aplicatia sa realizeze o conversie din ADI intr-o noua imagine.

Identificarea ADI se face in mod automat, neavand nevoie ca utilizatorul sa aleaga colturile imaginii in mod manual.



(Img. 18) Alegerea manuala a punctelor

## **Aplicarea de filtre asupra scanarilor**

Fiecare document necesita o procesare de culoare dupa scanare. Aceasta metoda face ca pagina scanata sa fie mai usor de citit pe telefon / calculator, mai usor de partajat cu altii si asigura o mai buna acuratete a continutului.



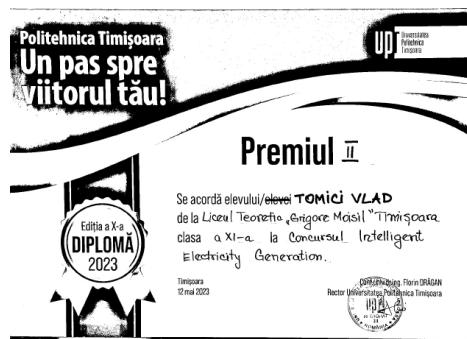
(Img. 19) *Color*



(Img. 20) *Photo*



(Img. 21) *Grayscale*



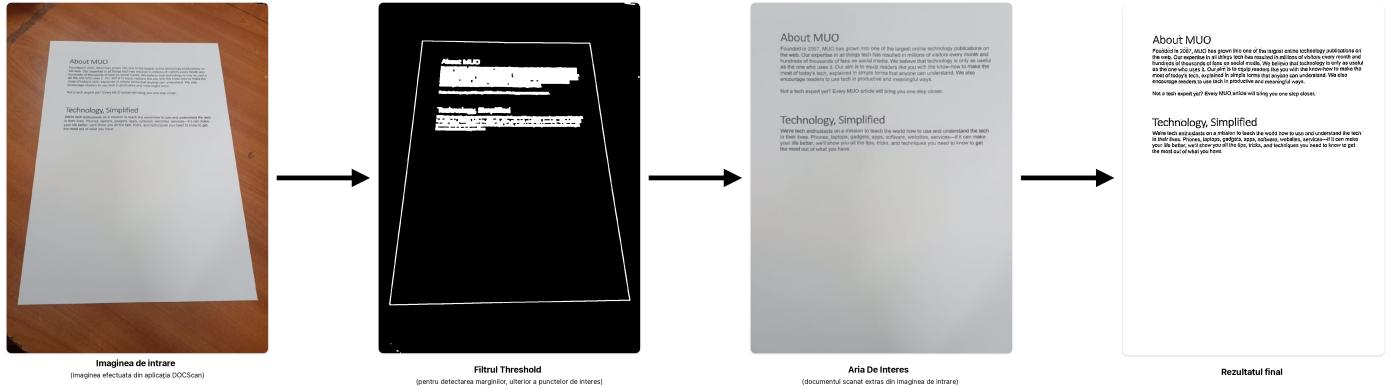
(Img. 22) *Black&White*

Pentru aplicarea unui filtru, aplicatia dispune de un algoritm care preia toti pixelii din ADI (Aria De Interes) a documentului scanat si ii proceseaza in functie de filtrul ales.

Filtrele *Color* (Img. 19), *Grayscale* (Img. 21) si *Black&White* (Img. 22) fac procesare de pixeli, schimband culoarea acestora, in timp ce filtrul *Photo* (Img. 20) returneaza direct aria de interes a imaginii de intrare, fara a aplica modificari de culoare a pixelilor.

Utilizatorul are posibilitatea sa aleaga ce filtru doreste sa aplice, inainte de efectuarea imaginii de intrare.

De asemenea, tot din aplicatie, utilizatorul are posibilitatea de a alege daca sa se oloseasca flash-ul camerei pentru efectuarea imaginii de intrare.



(Img. 23) Schematizarea intregului proces de scanare a documentelor cu camera

## Implementarea software

Toata partea de scanare are loc in fisierul `ScannerView.swift`, atat extragerea ADI, cat si aplicarea de filtre de culoare sau modificarea documentului scanat.

Incepem cu structura `ScanData` care este utilizata pentru a modela datele asociate cu o intrare de scanare. Printre proprietati, se numara:

- ▶ `id = UUID()` genereaza un identificator unic universal (UUID) pentru fiecare instant a structurii, asigurandu-se ca fiecare scanare are un identificator distinct
- ▶ `date: String` stocheaza data si ora la care a fost efectuata scanarea documentului
- ▶ `text_content: String` contine textul din documentul scanat

```
DocsScanApp.swift

89 struct ScanData: Identifiable {
90     var id = UUID()
91     var date: String
92     let text_content: String
93
94     init(text_content: String) {
95         self.text_content = text_content
96         self.date = Date().formatted()
97     }
98 }
```

Tot in aceasi structura, avem definit un constructor de initializare `init()` care primeste textul extras `text_content` ca parametru obligatoriu.

`self.text_content = text_content` atribuie textul extras la proprietatea structurii `self.text_content`.

`self.date = Date().formatted()` obtine data si ora curenta si o formateaza intr-un string, ulterior atribuind la proprietatea structurii `self.date`.

Structura principala pentru detectia de document poarta acelasi nume ca si fisierul de origine: ScannerView. Aceasta structura are rolul de a incorpora un element de tip VNDocumentCameraViewController din framework-ul Vision.

```
ScannerView.swift

24 struct ScannerView: UIViewControllerRepresentable {
25
26     let scannedImagesHandler = ScannedImagesHandler()
27
28     (...)

30     init(completionHandler: @escaping ([String]?) -> Void) {
31         self.completionHandler = completionHandler
32     }
33 }
```

Folosim o singura proprietate scannedImagesHandler: ScannedImagesHandler care stocheaza o instanta a unui obiect pentru gestionarea imaginilor scanate.

De asemenea, initializatorul init() primeste un closure de tip ([String]?) -> Void care va apela ulterior cu rezultatele scanarii. In caz de eroare, va returna nil.

Functia makeCoordinator() creeaza un obiect de tip Coordinator care actioneaza ca intermediar intre view-ul SwiftUI si controller-ul UIKit.

```
ScannerView.swift

28 func makeCoordinator() -> Coordinator {
29     return Coordinator(
30         completion: completionHandler,
31         scannedImagesHandler: scannedImagesHandler
32     )
33 }
```

Clasa Coordinator actioneaza ca delegat pentru VNDocumentCameraViewController, gestionand evenimentele legate de scanare.

completionHandler: ([String]?) -> Void este prima proprietate a clasei si reprezinta closure-ul primit din ScannerView care va fi apelat cu rezultatele scanarii.

scannedImagesHandler: ScannedImagesHandler este cea de-a doua proprietate a clasei si reprezinta obiectul pentru gestionarea imaginilor scanate.

Functia documentCameraViewController() se apeleaza cand scanarea documentului este finalizata cu succes. Eea creeaza un TextRecognizer pentru a extrage textul din scanare.

```
ScannerView.swift

35 final class Coordinator: NSObject, VNDocumentCameraViewControllerDelegate {
36     private let completionHandler: ([String]?) -> Void
37     private let scannedImagesHandler: ScannedImagesHandler
38
39     init(completion: @escaping ([String]?) -> Void, scannedImagesHandler: ScannedImagesHandler) {
40         self.completionHandler = completion
41         self.scannedImagesHandler = scannedImagesHandler
42     }
43
44
45     func documentCameraViewController(_ controller: VNDocumentCameraViewController, didFinishWith scan: VNDocumentCameraScan) {
46         let recognizer = TextRecognizer(cameraScan: scan)
47
48         recognizer.recognizeText(withCompletionHandler: completionHandler)
49
50         // ----- creating the array of scanned images -----
51         for pageNumber in 0..
```

Interam prin paginile scanate si extragem din fiecare imaginea, pentru ca ulterior sa le adaugam in array-ul `scannedImagesHandler`.

In caz de eroare, se apeleaza functia `documentCameraViewController()`.

```
ScannerView.swift

60 func documentCameraViewController(_ controller: VNDocumentCameraViewController, didFailWithError error: Error) {
61     completionHandler(nil)
62 }
63
```

Mai departe, in structura `ScannerView`, intalnim functia `makeUIViewController()` care creeaza un `VNDocumentCameraViewController` si atribuie `Coordinator` ca delegat al acestuia.

```
ScannerView.swift

70 func makeUIViewController(context: Context) -> UIViewController {
71     let viewController = VNDocumentCameraViewController()
72     viewController.delegate = context.coordinator
73     return viewController
74 }
75 }
```

# EXTRAGEREA DE TEXT DIN ARIA DE INTERES

Dupa ce aplicatia a reusit sa extraga imaginea din scanarea facuta, aceasta urmeaza sa procesata de un algoritm de inteligenta artificiala pentru detectia de caractere si cuvinte.

Libraria Vision din cadrul Swift, ne permite sa efectuam diferite operatii de procesare media. Toata procesarea se intampla pe dispozitivul utilizatorului pentru a mari performanta si pentru a proteja datele utilizatorului.

## Procesarea de text – folosirea de NLP

Extragerea de text din documente se poate realiza in 2 metode. Prima pune accent pe viteza de procesare a algoritmului, respectiv pe acuratetea rezultatelor in cazul celei de-a doua. In cazul DOCS, am mizat pe implementarea cu o viteza mai mare de procesare. Am ales acest lucru datorita faptului ca acuratetea in acest caz nu este grav afectata, rezultatele obtinute nefiind substantial diferite.

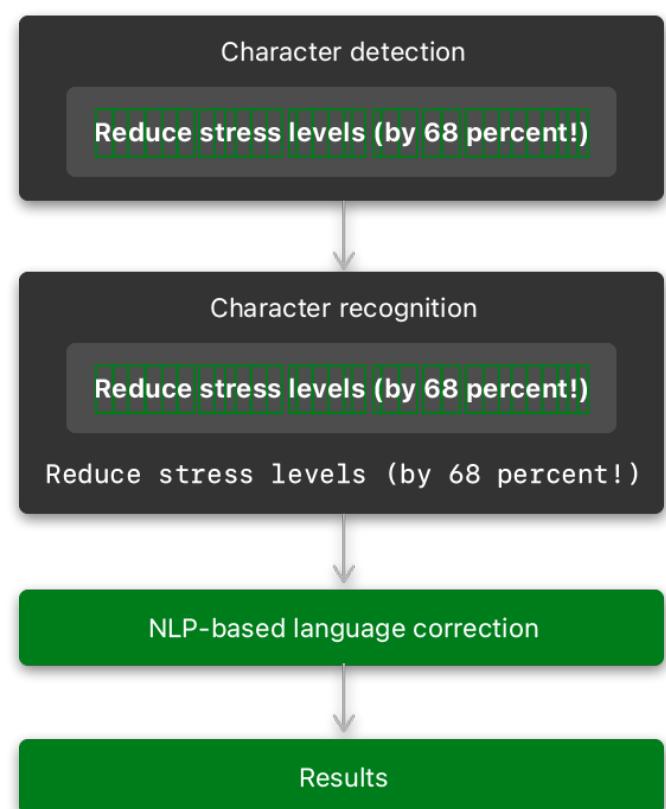
In primul rand, algoritmul recunoaste in mod individual toate caracterele din continutul media transmis catre el, implementare asemănatoare cu modul traditional de Recunoastere Optica a Caracterelor (OCR – Optical Character Recognition). Dupa, urmeaza ca fiecare caracter sa fie recunoscut si sa fie identificat cu valoarea corespunzatoare din codul ASCII.

Aplicand o retea neuronală, putem sa identificam care ar fi cuvintele si propozitiile din imaginea de intrare, ca mai apoi sa construim String-ul final.

Inainte de a returna variabila care contine textul extras din imaginea din intrare, o sa trecem rezultatul printr-un model de tip NLP (Natural Language Processing) pentru ca rezultatul extragerii din text sa fie a.

In final, este aplicat un algoritm de NLP (Natural Language Processing) care are posibilitatea de intelegera a cuvintelor si propozitiilor, astfel incat rezultatul string-ul final in urma executarii sa aiba o precizie mare.

Ca si fiinte umane, rationamentul uman este semnificativ diferit de un model de tip NLP. Oamenii se pot folosi de intuitie, creativitate,



emotie si o inteleger mult mai profunda a contextului, in raport cu un astfel de algoritm, pentru perceperea unei propozitii sau a unui fragment de text. Astfel, NLP are nevoie de mai multe repere pentru a reusi sa extraga continutul in mod eficient dintr-un text. NLP studiaza conexiunile dintre:

- modul in care gandim, din punct de vedere neurologic
- limbajul pe care il folosim, din punct de vedere lingvistic
- tipurile noastre de comportament, din punct de vedere al structurii de text

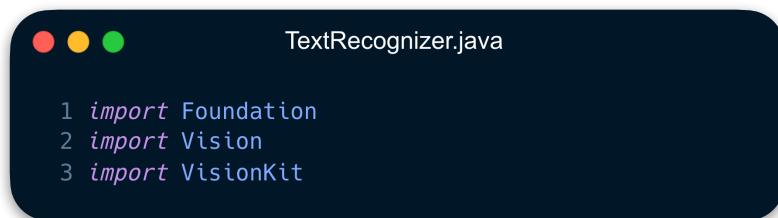
## ***Post-procesarea de text si afisarea acestuia in aplicatie***

Textul detectat din document urmeaza sa fie stocat intr-o variabila de tip String, ulterior fiind afis in cadrul subsecțiunii special dedicata pentru

Textul extras din document se poate copia cu usurinta prin apasarea butonului copy, mai departe putand sa fie lipit intr-un posibil mesaj, document sau alta locatie.

## ***Implementarea software***

Intreaga parte de recunoastere a textului, se afla in fisierul TextRecognizer.swift. In primul rand incepem prin importarea librariilor Foundation, Vision si VisionKit. Cele 2 din urma sunt folosite pentru procesarea de imagini si continut media.



TextRecognizer.swift

```
1 import Foundation
2 import Vision
3 import VisionKit
```

Urmeaza ca toata sectiunea dedicata detectiei de text sa fie implicata in clasa care poarta acelasi nume ca si fisierul.



```
12 final class TextRecognizer {
13     let cameraScan: VNDocumentCameraScan
14     init (cameraScan: VNDocumentCameraScan) {
15         self.cameraScan = cameraScan
16     }
17
18     private let queue = DispatchQueue(
19         label: "scan-codes",
20         qos: .default,
21         attributes: [],
22         autoreleaseFrequency: .workItem
23     )
24
25     (...)
```

Prima data, urmeaza sa declarăm singurul parametru al funcției: variabila `cameraScan`. Prin intermediul librăriei `VisionKit`, îi putem parametrui un tip special, care ne pune la dispozitie toate atributiile necesare pentru detectie și procesare.

Functia `returnImages()` returneaza un vector care contine toate imaginile in urma scanarii. Cu ajutorul acesteia, putem sa vizualizam paginile scanate in aplicatie.



```
TextRecognizer.java

25 func returnImages () -> [CGImage] {
26     let images = (0..<self.cameraScan.pageCount).compactMap({
27         self.cameraScan.imageOfPage(at: $0).cgImage
28     })
29
30     return images
31 }
```

Functia `recognizeText()` este cea care se ocupa de tot procesul de recunoastere a textului. In esenta, aceasta functie efectueaza detectarea de text pe fiecare pagina a documentului scanat si returneaza textul recunoscut printr-un handler de finalizare.



```
TextRecognizer.swift

35 func recognizeText (withCompletionHandler completionHandler: @escaping ([String]) -> Void) {
36     queue.async {
37         let images = (0..<self.cameraScan.pageCount).compactMap({
38             self.cameraScan.imageOfPage(at: $0).cgImage
39         })
40
41         let imagesAndRequests = images.map({ (image: $0, request: VNRecognizeTextRequest()) })
42         let textPerPage = imagesAndRequests.map {
43             image,
44             request -> String in
45             let handler = VNIImageRequestHandler(cgImage: image, options: [:])
46
47             do {
48                 try handler.perform([request])
49                 guard let observations = request.results else { return "" }
50
51                 return observations.compactMap({
52                     $0.topCandidates(1).first?.string
53                 }).joined(separator: "\n")
54             } catch {
55                 print(error)
56                 return ""
57             }
58         }
59         DispatchQueue.main.async {
60             completionHandler(textPerPage)
61         }
62     }
63 }
```

Ea primește ca parametru de intrare un handler de finalizare. Acest handler este o funcție care va fi apelata mai tarziu cu textul extras. Functia furnizeaza un array de String-uri prin handler-ul de finalizare, unde fiecare String reprezinta textul recunoscut dintr-o singura pagina.

Blocul `queue.async` ruleaza recunoasterea textului pe un thread secundar pentru a evita blocarea thread-ului principal. Functia parurge toate paginile documentului scanat folosind

o bucla, procesand fiecare pagina a documentului, ca ulterior imaginile procesate sa fie stocate in array-ul `images`.

Fiecare imagine este combinata cu un obiect nou de tip `VNRecognizeTextRequest`. Acest obiect ii spune librariei `Vision` sa efectueze recunoasterea textului pentru imaginea corespunzatoare.

Rezultatul este un `imagesAndRequests`, un tuples array, unde fiecare element contine o imagine si cererea sa asociata de recunoasterea de text.

Ulterior, o sa parcurgem fiecare pereche imagine-cerere printr-o structura de tip bucla. Pentru fiecare imagine, se creeaza un `VNImageRequestHandler`. Cu ajutorul sutchurii do { } catch { } urmeaza sa incercam sa facem recunoasterea de text.

`handler.perform([request])` este linia de cod care incearca sa efectueze cererea de recunoastere de text. Daca ea are succes, putem recupera rezultatele din `request.results`.

Inainte de returnarea finala, urmeaza sa procesam rezultatele. Combinam textul extras din intreaga pagina folosind `joined(separator: "\n")`, separand fiecare linie cu caracterul newline.

In cazul in care apar erori in timpul recunoasterii, urmeaza ca aplicatia sa printeze eroarea si sa returneze un string gol.

Dupa procesarea tuturor paginilor, folosim `DispatchQueue.main.async` pentru revenirea la thread-ul principal. In final, apelam handler-ul de finalizare cu textul extras din toate paginile ca argument.

# BIBLIOGRAFIE

- [https://www.youtube.com/watch?v=QJHmhLGv-\\_0](https://www.youtube.com/watch?v=QJHmhLGv-_0)
- <https://developer.apple.com/>
- <https://github.com/>
- <https://developer.apple.com/sf-symbols/>
- <https://firebase.google.com/docs>
- [https://developer.apple.com/documentation/vision/recognizing\\_text\\_in\\_images](https://developer.apple.com/documentation/vision/recognizing_text_in_images)
- <https://gemini.google.com/>
- <https://www.pngwing.com/>