1. Introducere
   1. Motivatie

Factorul care a stat la baza deciziei de a dezvoltă această aplicație este reprezentat de faptul că ne aflăm la începutul unei noi ere în educația noastră pe internet, odată cu apariția chatbot-urilor. Acestea pot furniza răspunsuri la întrebări din diverse domenii și sunt adesea considerate un înlocuitor perfect pentru motoarele de căutare tradiționale. Am ales să antrenez un model de recunoaștere vocală datorită fascinației mele pentru acest domeniu și pentru a înțelege cât mai bine funcționarea sistemelor de acest gen.

În ultimele decenii tehnologia a evoluat rapid, oferindu-ne noi instrumente pentru a ne ajută viață de zi cu zi. Inteligență artificială a cunoscut un avans semnificativ, oferind oportunități nelimitate de comunicare și cunoaștere. Un instrument apărut în ultimii ani, despre care marea majoritate a lumii nu este familiară cu el, este chatbot-ul ChatGPT. Chatbotul este un program de calculator conceput pentru a purta o conversație cu o ființă umană, în special prin internet \footnote{https://python.langchain.com/en/latest/index.html}. Consider că un chatbot bine antrenat ne poate scuti timp prețios din viețile noastre când vine vorba de a află ceva nou, fiind un bun înlocuitor pentru motoarele de căutare clasice. Un chatbot poate fi mai puternic decât un motor de căutare datorită abilităților sale avansate de procesare a limbajului natural și de învățare automată. În timp ce un motor de căutare oferă lista de site-uri web similare cu ce am căutat noi, un chatbot poate analiză și înțelege întrebarea și oferă un răspuns precis și personalizat, în funcție de context.

1.2 Introducerea in recunoasterea vocala si a chatbotiilor

Pentru acest proiect, recunoasterea vocala este deosebit de importanta, deoarece ofera o metoda rapida si convenabila de a comunica cu chatbotul, facand aceasta interactiune cat mai uman posibil. În acest context, am decis să îmi concentrez eforturile pe antrenarea unui model de recunoaștere vocală pentru limba română, cu ajutorul proiectului RobinASR. Acest proiect nu implica doar antrenarea unui model de recunoastere vocala pentru limba romana ci si proiectarea unei aplicatii mobile, care va utiliza acest model pentru o experienta cat mai naturala. Aplicatia va fi un chatbot, care va putea raspunde la orice tip de intrebari, iar pentru a aduce o memorie botului voi folosi libraria LangChain. LangChain este un framework pentru dezvoltarea aplicațiilor bazate pe modele de limbaj. Memoria este conceptul de stare persistentă între apelurile unui lanț/agent. LangChain oferă o interfață standard pentru memorie, o colecție de implementări de memorie și exemple de lanțuri/agenți care folosesc memoria.[[1]](#footnote-1)

În următoarele capitole, voi explora mai în detaliu tehnologia Langchain și modul în care acesta poate fi utilizat pentru a îmbunătăți performanța chatbotului. De asemenea, vom examina progresul realizat până în prezent în ceea ce privește recunoașterea vocală în limba română și voi amanunti metodele de antrenare a modelului RobinASR.

1. Limba romana si caracteristicile sale unice

In acest capitol vom explora trasaturile distincte ale limbii romane, care ar putea dificulta recunoasterea vocala.

2.1 Variatia dialectala si regionala

Limba română prezintă variații dialectale și regionale semnificative în ceea ce privește pronunția, gramatica și lexicul. Aceste diferente pot fi observate intre regiunile tarii sau a comunitatilor lingvistice din Romania. De exemplu, unele cuvinte sau expresii pot fi folosite într-un anumit mod într-o regiune și în alt fel în altă regiune. Aceste diferențe pot face dificilă recunoașterea vocală, deoarece sistemul trebuie să fie capabil să identifice și să se adapteze la aceste variații pentru a obține o acuratețe ridicată în recunoașterea vocală a vorbitorului. De exemplu, cuvântul "birou" poate fi pronunțat "biro" sau "biurou" în funcție de regiunea în care este vorbită limba. Astfel, sistemul de recunoaștere vocală trebuie să fie capabil să recunoască aceste variații și să le interpreteze corect.

* 1. Cuvinte compuse și cuvinte cu sens dublu

Limba română are o mare varietate de cuvinte compuse și cuvinte cu sens dublu, care pot crea confuzie în procesul de recunoaștere vocală. De exemplu, cuvântul "cașcaval" poate fi interpretat în două moduri diferite, fie ca un singur cuvânt, fie ca două cuvinte separate - "caș" și "caval". De asemenea, cuvintele compuse, cum ar fi "autobuz" sau "alb-negru", pot fi dificil de interpretat dacă sistemul de recunoaștere vocală nu este capabil să recunoască componentele cuvintelor.

1. Rezumat al cercetarilor existente privind recunoasterea vocala pentru limba romana

Deep learning este o tehnică de inteligență artificială care a devenit extrem de populară în ultimii ani, datorită capacității sale de a aborda probleme complexe în domenii precum recunoașterea imaginilor, recunoașterea vocală, traducerea automată și multe altele. Învățarea profundă a luat amploare în mare parte datorită avansurilor în puterea de calcul și disponibilitatea hardware-ului. Această evoluție a permis cercetătorilor și dezvoltatorilor să creeze și să antreneze modele de învățare profundă cu mai multe straturi, care pot fi utilizate într-o varietate de aplicații, de la recunoașterea imaginilor la traducerea automată a limbilor.

Retelele neuronale artificiale sunt inspirate din biologia creierului uman.[[2]](#footnote-2) In mod similar reteaua neuronala artificiala este compusa din unitati de procesare numite neuroni, care sunt conectati prin sinapse. Conducerea impulsurilor nervoase urmează principiul tot sau nimic (all or none)[[3]](#footnote-3). Dacă un neuron răspunde, trebuie să răspundă complet. Neuronul artificial cea mai mica si importanta parte a sistemului. Acesta este de fapt o functie de forma h(x) = sigma(wTx + b). Vectorul w e cunoscut ca „weights”, scalarul sau termenul liber, b este cunoscut ca „bias”. Tuplul dintre w si b se numeste „inference” si il voi nota cu tet. Functia sigma este o functie neliniara care este folosita ca functie de activare a neuronilor in retelele neuronale artificiale. Aceasta functie de activare simuleaza comportamentul natural al neuronilor, care urmeaza principiul tot sau nimic (all or none)[[4]](#footnote-4).

Revenind la parametrii tetha, pentru a ii determina avem nevoie sa antrenam reteaua neuronala artificiala. Pentru a antrena avem nevoie de date de test, compuse din inputul x si outputul y. Dupa ce facem predictia y-ului vrem sa stim daca e una corecta, astfel apare functia de loss. Functia de loss decide cat de diferit e y-ul prezis de algoritm fata de y-ul real, iar pe parcursul invatarii, putem observa cu ajutorul lui cat de bine invata modelul. Pentru a actualiza constant ponderile folosim procesul de backpropagation. Pentru a realiza aceasta, este nevoie de calculul gradientului funcției de eroare față de fiecare pondere în rețea. Ajustarea ponderilor se face în funcție de rata de învățare și de gradientul funcției de eroare în raport cu fiecare pondere din rețea, iar acest proces se repetă până când se obține

Deep learningul este bazat pe retelele neuronale pe mai multe straturi, acestea fiind de 3 categorii: retele neuronale fastforward(conexiunile dintre noduri nu pot forma cicluri), retele neuronale convolutionale si retele neuronale recurente.

De vorbit despre audio si speech……………………

Înainte de apariția deep learning-ului, tehnici precum Hidden Markov Models (HMMs) erau populare în industria de inteligență artificială. Modelul Markov Ascuns (HMM) este un model statistic propus pentru prima dată de Baum L.E. (Baum și Petrie, 1966) și folosește un proces Markov care conține parametri ascunși și necunoscuți. În acest model, parametrii observați sunt utilizați pentru a identifica parametrii ascunși, care sunt apoi utilizați pentru analiza ulterioară[[5]](#footnote-5). Printr-un exemplu in speech recognition putem presupune ca, cuvintele rostite sunt observate. , iar modelul HMM este utilizat pentru a deduce care sunt sunetele individuale care compun cuvintele respective. Aceste sunete sunt considerate ca fiind parametrii ascunși și sunt utilizați pentru a identifica cuvintele individuale rostite. Cu toate acestea, HMM-urile aveau limitări în ceea ce privește capacitatea lor de a înțelege semnificația contextului și de a procesa informații complexe.

Pentru a antrena un model de recunoastere a vorbirii in limba romana, este important sa cunoastem stadiul actual al cercetarii in acest domeniu. In acest sens se poate lua in considerare lucrarea de Andrei-Marius AVRAM, Vasile PAIŞ, Dan TUFIŞ, "Towards a Romanian End-to-End Automatic Speech Recognition Based on DeepSpeech2", prin care se prezinta reteaua neuronala, care foloseste de asemenea un model de limbaj statistic pentru imbunatatirea transcriptiilor. Lucrarea contine de asemenea extragerea feature-urilor din fisierele audio, acesteaua au fost impartite în ferestre de 20 ms și a calculat coeficienții cepstrali de frecvență Mel (MFCC) (Tufis et al. [1]). Consider ca optiunea de a determina feature-urile prin aceasta tehnica este una potrivita avand in vedere celelalte tehnici de extragere a feature-urilor printre care se numara Perceptual Linear Prediction(PLP). Un punct forte al MFCC-urilor este partea de preprocesare numita preemphasis. Preemphasis crește cantitatea de energie în frecvențele înalte. Pentru segmentele vocale precum vocalele, există mai multă energie la frecvențele inferioare decât la frecvențele mai înalte. Creșterea energiei de înaltă frecvență face ca informațiile în formați mai mari să fie mai disponibile modelului acustic.[[6]](#footnote-6).

In cele din urma reteaua neuronala este compusa din 2 straturi convoluționale, cu 32 de filtre: primul cu o dimensiune a kernelului de (41, 11) și un stride de (2, 2), iar al doilea cu o dimensiune a kernelului de (21, 11) și un stride de (2, 1).

“Proiectul ROBIN cu o durată de trei ani, coordonat de Universitatea Politehnica din București, Facultatea de Informatică,a fost lansat în 2018. Consorțiul ROBIN include experți în prelucrarea limbajului natural din Institutul de Cercetare pentru Inteligență Artificială "Mihai Drăgănescu" al Academiei Române (RACAI) și Universitatea Politehnica din București, în robotică de la Universitatea Politehnica din București și Institutul de Matematică al Academiei Române, specialiști renumiți în tehnologia Internet of Things (IoT) de la Universitatea din București și Universitatea Tehnică din Cluj-Napoca, și experți în proiectarea vehiculelor autonome de la Universitatea "Dunărea de Jos" din Galați.” Tufis et al. [2]

Proiectul este centrat in jurul a mai multor roboti: roboti destinati persoanelor cu nevoi speciale, roboti pentru interactiunea cu client, roboti pentru vehicule. Subcomponentele formate la baza acestor nevoi sunt:

-ROBIN-Social (aimas.cs.pub.ro/robin/en/robin-social/),

-ROBIN-Car (aimas.cs.pub.ro/robin/en/robin-car/),

-ROBIN-Context (aimas.cs.pub.ro/robin/en/robin-context/),

-ROBIN-Dialog (aimas.cs.pub.ro/robin/en/robin-dialog/)

-ROBIN-Cloud (aimas.cs.pub.ro/robin/en/robin-cloud/). Tufis et al. [2]

Scopul acestui proiect este de a facilita interacțiunea oamenilor cu roboții prin intermediul recunoașterii vocale în limba română.(am reprodus din introducerea lor). Proiectul a fost antrenat 230 de ore si a reusit o eroare per cuvant de 9.91% si eroare per caracter de 2.81%, cu cele mai celebre seturi de date din ultimii ani.

Chiar daca modelul a fost antrenat pe multe ore daca ne raportam la cate seturi de date in limba romana exista pana in prezent, acesta lasa loc de imbunatari, astfel apare lucrarea ROMANIAN SPEECH RECOGNITION EXPERIMENTS FROM THE ROBIN PROJECT, de aceeasi autori.

Proiectul abordează metode de îmbunătățire a vitezei și de perfecționare a traducerilor, luând în considerare particularitățile unice ale limbii române care pot reprezenta obstacole pentru modelul de recunoaștere vocală. Sistemul dispune de doua endpoint-uri REST, unul /transcribe, responsabil de convertirea audio-ului in text, iar celalalt /correct, primeste ca input text-ul si il corecteaza conform caracteristicilor limbii romane.

O alta abordare de acest gen este a lui Tufis et a. [2] in care este folosit toolkit-ul Kaldi care a obtinut o eroare per cuvant de 25%. Eroarea este intr-adevar mare, poate fi datorata setului de date de antrenament si de natura sa, daca este citit sau vorbit spontan.

Proiectul are in spatele sau abordari mai vechi si mai incete din punct de vedere al vitezei in care este recunoscut audio-ul. Un astfel de exemplu ar fi lucrarea “Kaldi-based DNN Architectures for Speech Recognition in Romanian”, de autorii Alexandru-Lucian Georgescu, Horia Cucu, Corneliu Burileanu.

* 1. Descrierea instrumentelor disponibile pentru antrenarea modelelor de recunoaștere vocală în limba română

O imbunatatire importanta cu care pot sa-mi pun amprenta acestui proiect este antrenarea cu ajutorul setului de date Common Voice. Common Voice este un set de date vocale disponibil public, alimentat de vocile colaboratorilor voluntari din întreaga lume. Persoanele care doresc să creeze aplicații vocale pot folosi setul de date pentru a antrena modele de învățare automată.\footnote { https://commonvoice.mozilla.org/en/about }. Common Voice detine seturi de inregistrari pentru limba romana de peste 200 de ore validate.

Modelul ASR al proiectului ROBIN a fost antrenat cu cele mai celebre data seturi de la aceea vreme, printre care se numara CoRoLa. Scopul creării acestui corpus este de a oferi o imagine obiectivă a limbii române actuale, atât în forma scrisă, cât și în cea vorbită, prin includerea unor texte diverse, care cuprind perioada de la 1989 până în prezent. \footnote{ <https://corola.racai.ro/>}. Corpusul in prezent este in jur de 300 de ore de inregistrari, înregistrări profesionale din diverse surse, posturi de radio, studiouri de înregistrare. Printre celelalte corpusuri de date amintim: OSCAR, Romanian Digits(RoDigits), Romanian Common Voice (RSV), Romanian Speech Synthesis (RSS), Read Speech Corpus(RSC) (Tufiș et a. [1]).

Spre deosebire de celelalte OSCAR este un corpus uriaș multilingv cu sursă deschisă care a fost obținut prin filtrarea de Common Crawl și prin gruparea textului rezultat după limbă. Versiunea în limba română conține aproximativ 11 GB de propoziții amestecate deduplicate. (Tufiș et a. [1])

Cele mai importante seturi de date in limba romana:

Table, modificat pe common voice

Romanian Speech Synthesis (RSS), creat pentru sinteza vocii și conține 4 ore de înregistrări audio realizate de o singură vorbitoare de sex feminin, folosind mai multe microfoane. Vorbitoarea a citit 4000 de propoziții extrase din romane, articole de ziare selectate pentru acoperirea di-foanelor și povești. Corpusul RSS a fost extins și cu peste 1700 de exprimări ale altor două vorbitoare de sex feminin, ajungând astfel la o durată totală de 5,5 ore de înregistrări audio. (Tufiș et a. [1])

Corpusul RoDigits conține 37,5 ore de înregistrări audio cu cifre vorbite de la 154 de vorbitori, cu vârste cuprinse între 20 și 45 de ani. Fiecare vorbitor a înregistrat 100 de clipuri cu 12 cifre românești generate aleatoriu, iar după validarea semi-automată, corpusul final a inclus 15.389 de fișiere audio. (Tufiș et a. [1])

Read Speech Corpus(RSC) este un set de date ce conține vorbire citită, colectat de către Laboratorul de Cercetare a Vorbirii și Dialogului. Aceste înregistrări au fost realizate în diverse condiții, folosind diferite tipuri de microfoane și sisteme de înregistrare audio, prin intermediul unei aplicații online dezvoltate de același grup de cercetare. Participanții la înregistrări au fost în principal studenți și cadre didactice din cadrul Facultății de Electronică, Telecomunicații și Tehnologia Informației de la Universitatea "Politehnica" din București. Corpusul este constituit dintr-o colecție de 136.120 de înregistrări audio, obținute de la 164 de vorbitori nativi ai limbii române. Georgescu et a. [3] Fiecare înregistrare conține fraze și cuvinte selectate din literatură, articole de știri online și vocabular specific limbii române. /footnote { <https://speed.pub.ro/downloads/speech-datasets/>}

Scopul acestui corpus este de a aduce o contribuție semnificativă în domeniul recunoașterii vocale în limba română, prin furnizarea unui set de date, având în vedere că multe dintre seturile de date existente în România nu sunt disponibile publicului larg.

Raportandu-ne la lucrarea lui Tufiș et a. [1] Common Voice avea in jur de 7 ore de audio transcris înregistrat de 79 de difuzoare in limba romana, comparativ cu cele 200 de ore pe care le detine in prezent.

**Flask**

Flask este un framework pentru dezvoltarea aplicatiilor web in Python, dezvoltat pentru a fi simplu de învățat și utilizat. Acesta este conceput pentru a oferi un set de functionalitati de baza, in rest fiind personalizabil. Printre functionalitatile de baza amintim: rutarea și gestionare a cererilor (facilitează definirea rutelor URL și gestionarea cererilor HTTP asociate acestora)., integrarea cu baza de date (nu impune o baza de date, dar ofera suport pentru acestea). Fiind un microframework, dar acest lucru nu înseamnă că întreaga aplicație trebuie să fie inclusă într-un singur fișier Python. Poți și ar trebui să folosești mai multe fișiere pentru programe mai mari, pentru a gestiona complexitatea.

Micro înseamnă că framework-ul Flask este simplu, dar extensibil. Poți lua toate deciziile: ce bază de date să folosești, dacă dorești un ORM etc., Flask nu decide pentru tine.

Flask este unul dintre cele mai populare framework-uri web, ceea ce înseamnă că este actualizat și modern. Poți extinde ușor funcționalitatea sa. Poți scala aplicația pentru aplicații complexe. \footnote{ <https://pythonbasics.org/what-is-flask-python/>}.

Am optat pentru utilizarea framework-ului Flask în locul Django, deoarece acesta este un framework independent, creat special pentru sisteme cu o complexitate mai redusă. În cazul meu, am ales Flask pentru a dezvolta funcționalitatea de recunoaștere vocală și pentru a comunica cu chatbotul utilizând biblioteca LangChain.

**Langchain**

LangChain reprezintă o bibliotecă care facilitează lucrul cu un model lingvistic extins(large language model). Un large language model e un model care are rolul de a prezice ce caracter/cuvant apare dupa o anumita secventa, de a evalua probabilitatea apariției a unui șir de cuvinte sau caractere într-un anumit limbaj.

Principiile fundamentale pe care se bazează LangChain, "Data-aware" și "Agentic"\footnote{ https://python.langchain.com/en/latest/index.html}, permit integrarea large language modelul cu diverse surse de date, cum ar fi motoarele de căutare, fișiere externe utilizate ca intrare sau servicii de stocare, cum ar fi Google Drive. Astfel, LangChain facilitează colaborarea cu aceste surse de date externe, extinzând astfel funcționalitatea acestuia în cadrul aplicațiilor.

Unul dintre elementele de baza ale acestei librarii este interfata standard pentru modele. Pentru a crea o instanta a unui model, putem seta modelul folosit, parametrul temperature, care seteaza cat de unic sa fie raspunsul generat, dar si numarul maxim de tokenuri folosite pentru a genera raspunsul.

Modelul folosit de mine “text-davinci-003” poate stoca pana la 4097 de tokenuri. Acesta poate îndeplini orice sarcină lingvistică cu o calitate mai bună, cu o producție mai lungă și cu o respectare consecventă a instrucțiunilor decât modelele Curie, Babbage sau Ada. De asemenea, suportă inserarea de completări în text. \footnote { <https://platform.openai.com/docs/models/gpt-3-5>}.

O imagine care conține text, Font, captură de ecran

Descriere generată automat

Memoria pe care Langchain este o caracteristica esentiala in constructia unui chatbot. LangChain ofera mai multe tipuri de memorie la dispozitie, printre care se afla: Conversation Buffer Memory, Conversation Summary Memory, Conversation Summary Buffer Memory, Conversation Entity Memory, Conversation Knowledge Graph Memory. Fiecare tip de memorie are avantaje si dezavantaje, pe care le voi prezenta in urmatoarea sectiune.

Comparatie intre memoriile oferite de LangChain:

Conversation Buffer Memory, permite stocarea tututor mesajelor, dintr-o conversatie și apoi extragerea mesajele într-o variabilă. Conversation Summary Memory, creează un rezumat al conversației de-a lungul timpului. Acest lucru poate fi util pentru a condensa informațiile din conversație de-a lungul timpului. ConversationBufferWindowMemory păstrează o listă a interacțiunilor din conversație de-a lungul timpului. Se utilizează numai ultimele K interacțiuni. Acest lucru poate fi util pentru a păstra o fereastră glisantă a celor mai recente interacțiuni, astfel încât memoria tampon să nu devină prea mare.

Entity Memory extrage cu ajutorul large language model-ului caracteristici cheie din mesaje si le salveaza. De exemplu din mesajul “Deven & Sam are working on a hackathon project”, entitatea “Sam” va avea asociata actiunea “Sam is working on a hackathon project with Deven.”

Conversation Knowledge Graph Memory utilizează un grafic de cunoștințe pentru a recrea memoria.

De exemplu pentru mesajul “Culoarea preferata a lui Sam este rosu”, graful va arata in felul urmator:{“subject”=”Sam”, “predicate”=”culoarea preferata”, “object”=”red”}.

Kotlin si Jetpack Compose

Kotlin este un limbaj de programare static, orientat pe obiecte, dezvoltat inițial de JetBrains (o companie de software) și lansat în 2011. Acesta s-a inspirat din mai multe limbaje de programare, inclusiv (dar nu numai) Java, Scala, C# și Groovy. Una dintre ideile principale din spatele Kotlin este aceea de a fi pragmatic, adică de a fi un limbaj de programare util pentru dezvoltarea de zi cu zi, care ajută utilizatorii să își facă treaba prin intermediul funcțiilor și instrumentelor sale. Astfel, o mulțime de decizii de proiectare au fost și încă sunt influențate de cât de benefice sunt aceste decizii pentru utilizatorii Kotlin.\footnote{https://kotlinlang.org/spec/introduction.html}

In ultimii ani, acest limbaj a castigat o popularitate semnificativa pentru dezvoltarea de aplicatii pe dispozitive mobile cu Android. Kotlin a câștigat popularitate în rândul dezvoltatorilor Android datorită compatibilității sale cu Java, ceea ce a permis adoptarea ușoară a limbajului în ecosistemul existent.

Jetpack Compose este un toolkit modern care functioneaza impreuna cu Kotlin pentru crearea interfetei cu utilizatorul. \footnote{https://developer.android.com/jetpack/compose}.

Acesta simplifică și accelerează dezvoltarea interfețelor utilizator pe Android, dând viață aplicațiilor dvs. cu mai puțin cod, instrumente puternice și API-uri Kotlin intuitive. Face construirea interfețelor de utilizator Android mai rapidă și mai ușoară. În timpul creării Compose, am lucrat cu diferiți parteneri care au experimentat toate aceste beneficii la prima mână și ne-au împărtășit câteva dintre concluziile lor. \footnote{ https://developer.android.com/jetpack/compose/why-adopt#less-code}.

Modul usor de a folosi Jetpack compose in proiecte, a constituit un motiv solid pentru folosirea acestuia. Astfel, putem crea functii adnotate cu cuvantul cheie “@Composable”, iar acestea vor fi afisate pe ecran.

@Composable

fun MyScreen() {

MaterialTheme {

Column {

Text(text = "Salut, Jetpack Compose!")

Button(onClick = { /\* Acțiunea dvs. \*/ }) {

Text(text = "Apasă-mă!")

}

}

}

}

--x

În mod tradițional, dezvoltarea interfețelor utilizator pentru aplicațiile Android presupunea lucrul cu fișiere XML și gestionarea componentelor UI în mod programatic. Această abordare a dus adesea la un cod complex și redundant, făcând din dezvoltarea interfețelor pentru utilizator o sarcină consumatoare de timp și predispusă la erori. În plus, natura imperativă a abordării tradiționale făcea dificilă întreținerea și actualizarea interfețelor de utilizare pe măsură ce aplicația se dezvolta.

Google a introdus Jetpack Compose ca soluție pentru dezvoltarea UI-ului într-un mod mai eficient și intuitiv. Jetpack Compose valorifică puterea lui Kotlin, limbajul preferat de Google pentru dezvoltarea Android, pentru a oferi un set de instrumente de interfață utilizator declarativ.

În ultimii ani, în lumea dezvoltării de aplicații mobile Android au apărut schimbări semnificative odată cu apariția Jetpack Compose. Jetpack Compose este un set de instrumente moderne introdus de Google care simplifică și accelerează procesul de creare UI-ului, atractiv și interactiv pentru aplicațiile Android, care se aliniază perfect cu indicațiile din Material Design

Material Design oferă un set de indicații și componente care asigură o experiență de utilizare coerentă și încântătoare pe diferite dispozitive Android. Acesta se concentrează pe principii precum elevația, layout-urile receptive, mișcarea semnificativă și tipografia coerentă, printre altele. Jetpack Compose îmbrățișează aceste principii și oferă un set bogat de componente Material Design și capacități de tematizare pentru a simplifica implementarea Material Design în aplicațiile Android.

Atunci când lucrează cu Jetpack Compose, dezvoltatorii pot accesa și utiliza cu ușurință componentele Material Design, cum ar fi butoanele, cardurile, dialogurile, sertarele de navigare și multe altele. Aceste componente sunt concepute pentru a fi flexibile și personalizabile, permițând dezvoltatorilor să le adapteze aspectul și comportamentul pentru a se potrivi nevoilor aplicației lor. Prin utilizarea acestor componente Material Design preinstalate, dezvoltatorii pot economisi timp și efort în proiectarea și implementarea elementelor comune de interfață utilizator, asigurând un aspect și o senzație consecventă în întreaga aplicație.

În plus față de componente, Jetpack Compose oferă capacități puternice de tematizare. Dezvoltatorii pot defini o temă care să încapsuleze stilul vizual al aplicației și să o aplice la diverse elemente de interfață. Acest lucru permite realizarea unor scheme de culori, tipografii și alte atribute stilistice consecvente în întreaga aplicație. Cu sprijinul tematizării Material Design, dezvoltatorii pot crea cu ușurință interfețe de utilizator coerente din punct de vedere vizual și plăcute din punct de vedere estetic, care se aliniază cu principiile generale de branding și design ale aplicației lor.

În plus, Jetpack Compose permite o personalizare profundă dincolo de componentele Material Design furnizate. Dezvoltatorii își pot construi propriile componente personalizate și elemente de interfață utilizator care respectă liniile directoare Material Design. Această flexibilitate le permite să creeze interfețe de utilizator unice și de marcă, păstrând în același timp principiile de bază ale Material Design.

Dagger-Hilt

Dependency injection este un principiu semnificativ în dezvoltarea aplicațiilor Android. Aceasta ajută la decuplarea componentelor, promovează reutilizarea și facilitează dezvoltarea modulară. Dagger-Hilt este una din librăriile populare pentru gestionarea dependențelor în aplicațiile Android.

Dagger Hilt simplifică procesul de injectare a dependențelor prin reducerea codului și prin furnizarea unei abordări standardizate a gestionării dependențelor. Acesta este construit peeste framework-ul Dagger, cunoscut pentru eficiența și performanța sa. Cu Hilt, dezvoltatorii pot profita de avantajele Dagger, bucurându-se în același timp de o experiență mai raționalizată și mai intuitivă, special concepută pentru dezvoltarea Android.

Caracteristici și beneficii cheie:

• Integrare cu componentele Android - Dagger Hilt se integrează perfect cu componentele Android careau lifecycle, cum ar fi Activity, Fragment, servicii și componente UI. Acesta oferă adnotări și obiecte predefinite care facilitează injectarea dependențelor în aceste clase specifice Android. Hilt se ocupă de gestionarea lifecycle-ului și se asigură că acestea sunt create și distruse în mod corespunzător, fără memory leaks.

• Sintaxă simplă - Hilt simplifică sintaxa de injectare a dependențelor în comparație cu Dagger. Acesta introduce adnotări precum @AndroidEntryPoint și @Inject care generează automat codul necesar pentru injectarea dependențelor în componentele Android. Acest lucru reduce volumul de cod care trebuie scris și de configurare necesară, ceea ce duce la un cod mai curat și mai concis.

• Furnizarea automată a dependențelor - Hilt generează automat codul necesar pentru a furniza dependențe acolo unde acestea sunt necesare. Dezvoltatorii pot adnota clase sau interfețe cu @Inject, iar Hilt se ocupă de crearea și furnizarea instanțelor necesare atunci când sunt solicitate. Această furnizare automată elimină necesitatea configurării manuale și reduce șansele de erori sau omisiuni.

• Scoped dependencies - Hilt oferă o gamă scope-uri predefinite, inclusiv @Singleton, @ActivityScoped, @FragmentScoped și multe altele. Aceste scope-uri permit dezvoltatorilor să controleze ciclul de viață și disponibilitatea dependențelor. Dependențele create și partajate automat în cadrul scope-ului specificat, asigurând coerența și prevenind crearea de obiecte inutile.

• Integrare cu ViewModel - Hilt oferă o integrare perfectă cu componenta Android ViewModel, facilitând injectarea dependențelor în componenta ViewModel. Utilizând adnotarea @HiltViewModel, dezvoltatorii pot defini dependențe pentru aceasta componentă, dependențele rămânând în viață

• Suport pentru testare - Dagger Hilt simplifică testarea prin furnizarea de mecanisme convenabile pentru înlocuirea dependențelor cu implementări simulate sau false. Dezvoltatorii pot utiliza adnotările de testare ale Hilt, cum ar fi @UninstallModules, pentru a personaliza injectare dependențelor în mod special pentru scenarii de testare. Acest lucru permite un control și o izolare mai bună a dependențelor în timpul testelor unitare.

• Adoptare și migrare treptată - Hilt suportă adoptarea graduală, permițând dezvoltatorilor să introducă injectarea dependențelor în mod incremental în proiectele existente. Oferă compatibilitate cu componentele Dagger existente, permițând dezvoltatorilor să migreze de la Dagger la Hilt treptat, modul cu modul.

Java si Spring Boot:

Spring Boot este un framework Java open-source care a revoluționat modul în care dezvoltăm și implementăm aplicații Java. Spring Boot se concentrează pe simplificarea dezvoltării, configurării și implementării aplicațiilor Java, oferind un cadru robust și eficient pentru construirea de aplicații de înaltă calitate. Dezvoltatorii din întreaga lume își încep călătoria de codare învățând Java. Flexibil și ușor de utilizat, Java este preferatul dezvoltatorilor pentru o varietate de aplicații - totul, de la aplicații de social media, web și jocuri până la aplicații de rețea și aplicații de întreprindere.

\footnote{https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-java-spring-boot/}

Speech Recognition library

Am ales pentru partea de recunoastere a vorbirii sa folosesc libraria SpeechRecognition. SpeechRecognition este o bibliotecă pentru efectuarea recunoașterii vorbirii, cu suport pentru mai multe motoare și API-uri, online și offline.\footnote{https://github.com/Uberi/speech\_recognition/tree/master} Aceasta ofera suport pentru recunoasterea vocala de la Google. Modul de utilizare a acestei librarii este unul usor..

În cadrul aplicației mele, am optat pentru utilizarea bibliotecii SpeechRecognition pentru a converti înregistrările audio în text. Această bibliotecă utilizează API-ul Google Cloud, în loc să folosesc proiectul ROBIN, pe care îl antrenez.

Am luat această decizie întrucât funcționalitatea oferită de SpeechRecognition furnizează transcrieri mai precise și într-un timp mai scurt. De asemenea, am avut în vedere faptul că am nevoie să transform în text înregistrări audio scurte, cu o durată de până la 10 secunde, pentru a le utiliza în mesaje. Astfel, timpul necesar pentru transformarea acestor înregistrări este extrem de important.

Prin alegerea SpeechRecognition, am obținut rezultatele dorite într-un mod eficient și rapid, asigurându-ne că transformarea în text a înregistrărilor se realizează într-un timp optim, în concordanță cu necesitățile aplicației noastre.

Pinecone:

Embeddings sunt reprezentări numerice generate de modelele de inteligență artificială, care capturează caracteristicile și relațiile complexe ale datelor. Deoarece aceste embeddings pot avea o dimensiune mare și sunt dificil de gestionat, avem nevoie de o soluție specializată pentru stocarea și interogarea lor.

Pinecone este o bază de date vectorială specializată, concepută pentru a gestiona eficient acest tip de date. O bază de date vectorială, cum este Pinecone, oferă funcționalități optimizate pentru stocarea și interogarea încorporărilor vectoriale. Aceasta combină avantajele unei baze de date tradiționale, cum ar fi scalabilitatea și capacitățile de interogare complexe, cu specializarea în gestionarea încorporărilor vectoriale.

Prin utilizarea Pinecone și altor baze de date vectoriale, putem beneficia de o performanță superioară în stocarea și căutarea eficientă a acestor embeddings, facilitând dezvoltarea de aplicații AI și îmbunătățind rezultatele și experiențele utilizatorilor.

Introducere in chatboti si speech recognition

În domeniul modelelor de limbaj natural, există câteva modele extrem de semnificative dezvoltate de companii precum OpenAI și Google. Printre acestea se numără GPT de la OpenAI, Bert de la Google, LaMDA de la Google, PaLM de la Google și LLaMA de la Meta AI. Deși Google a dezvoltat mai multe variante și implementări ale acestor modele, în opinia mea, cel mai relevant model este GPT de la OpenAI. În prezent, versiunea stabilă și actuală a acestui model este GPT-3.5.

Modelul GPT-3.5 reprezintă o versiune avansată a modelului GPT-3, fiind succesorul acestuia. Ambele serii de modele aduc îmbunătățiri și funcționalități remarcabile. În ceea ce privește GPT-3.5, amintim urmatoarele modele: code-davinci-002, text-davinci-002, text-davinci-003 și gpt-3.5-turbo. Pe de altă parte, GPT-3 include versiunile davinci și text-davinci-001 (conform lui Junjie Ye et al.).

- Davinci: baza modelelor din seria GPT-3, care poate înțelege și genera limbaj natural cu o calitate superioară.

- Text-davinci-001: un model InstructGPT(genereaza instrucțiuni și răspunsuri într-o varietate de domenii și contexte. Acest model este antrenat pe o vastă colecție de texte care cuprind instrucțiuni și informații practice provenite din diverse surse, inclusiv manuale, tutoriale și ghiduri.) bazat pe davinci.

- Code-davinci-002: cel mai performant model Codex, care este un descendent al GPT-3 și care stă la baza modelului modelele din seria GPT-3.5, cu date de instruire care conțin atât limbaj natural, cât și miliarde de linii de cod sursă din surse disponibile public, inclusiv coduri din depozitele publice GitHub. (Junjie Ye et al.).

- Text-davinci-002: un model InstructGPT bazat pe code-davinci-002. (Junjie Ye et al.).

- Text-davinci-003: o versiune îmbunătățită a text-davinci-002, dar antrenată cu metoda Proximal Policy. Optimizare Proximală (PPO). (Junjie Ye et al.).

- Gpt-3.5-turbo: cel mai capabil model GPT-3.5 și optimizat pentru chat la 1/10 din costul textdavinci-. 003. (Junjie Ye et al.).

Revenind la celelalte large language modele, o abordare interesanta o gasesc ca fiind Alpaca. Alpaca este un model de limbaj natural, care provine de la LLaMa, fiind rafinat. Ceea ce este unic la Alpaca este cât de mic și rentabil este. Cu doar 7 miliarde de parametri, Alpaca este la fel de bun ca modelul text-davinci-003 de la OpenAI. Acesta poate fi rulat local, fara a fi necesara conexiunea la internet. \footnote{ <https://beebom.com/how-run-chatgpt-like-language-model-pc-offline/>}.

Pentru a utiliza Alpaca local, este necesar un calculator decent, chiar și fără o placă grafică, deși timpul de răspuns al modelului poate fi mai lent. De asemenea, este necesară o capacitate de stocare liberă de 8 GB.

Cu toate acestea, în proiectul meu, am ales să nu utilizez Alpaca, deși este o opțiune convenabilă pentru momentele in care nu aplicatia nu este conectata la internet. Motivul constă în faptul că acest model nu este deloc fiabil pe dispozitivele mobile datorită modului de instalare și cerințelor de putere hardware pe care le necesită.

* Use case uri importante
* Arhitectura
* Backend arhitectura desen python + backend
* Client server
* Arhitectura pe frontend
* Antrenare ai
* Ce dataset am folosit? Cat de lung e…
* Ce configurari am facut
* Cum am schimbat mp3 in wav
* Proiectul asr ce mai are, window function, asta se aplica cand vorbesc despre configurari
* Confusion matrix, wer, character er, swot??
* De pe ce laptop am rulat
* Tehnologii
* Cloud
* Pinecone
* Bazele de date
* La robin mai am de vorbit despre kaldi
* Compar putin si speechrecognition cu robin ca timp

|  |  |  |  |
| --- | --- | --- | --- |
| Nume | Login | | |
| Actori | User | | |
| Descriere | Utilizatorul primește acces la aplicație după introducerea datelor sale | | |
| Pre-condiții | - | | |
| Post-condiții | Utilizatorul primește acces la aplicație | | |
| Flux | **Nr.** | **Acțiune utilizator** | **Răspunsul sistemului** |
| 1 | Utilizatorul intra in aplicatie |  |
| 2 |  | Încărcarea interfeței de login |
| 3 | Utilizatorul introduce user-ul și parola și apasă butonul de login |  |
| 4 |  | Serverul verifică datele introduse și deschide fereastra principală a aplicatiei, adica un nou chat. |

|  |  |  |  |
| --- | --- | --- | --- |
| Nume | Send Message | | |
| Actori | User | | |
| Descriere | Utilizatorul trimite unui mesaj chatbotului | | |
| Pre-condiții | Utilizatorul trebuie sa fie logat in aplicatie | | |
| Post-condiții | Utilizatorul primește raspuns la mesajul trimis | | |
| Flux | **Nr.** | **Acțiune utilizator** | **Răspunsul sistemului** |
| 1 | Navigarea catre un chat existent sau un chat nou |  |
| 2 |  | Incarcarea mesajelor existente chatului |
| 3 | Utilizatorul introduce un mesaj si apasa pe butonul “send” |  |
| 4 |  | Serverul primeste mesajul, il proceseaza si ii trimite un raspuns utilizatorului |

|  |  |  |  |
| --- | --- | --- | --- |
| Nume | Send Pdf Message | | |
| Actori | User | | |
| Descriere | Utilizatorul trimite unui mesaj intr-un chat bazat pe PDF chatbotului | | |
| Pre-condiții | Utilizatorul trebuie sa fie logat in aplicatie | | |
| Post-condiții | Utilizatorul primește raspuns la mesajul trimis | | |
| Flux | **Nr.** | **Acțiune utilizator** | **Răspunsul sistemului** |
| 1 | Navigarea catre un chat existent marcat cu simbolul de PDF |  |
| 2 |  | Incarcarea mesajelor existente chatului |
| 3 | Utilizatorul introduce un mesaj si apasa pe butonul “send” |  |
| 4 |  | Serverul primeste mesajul, il proceseaza si ii trimite un raspuns utilizatorului |
| Flux alternativ | 1 | Navigarea catre un chat nou si apasarea butonului PDF pentru a selecta un fisier din file explorer |  |
|  | 2 |  | Sistemul stocheaza fisierul selectat |
|  | 3 | Utilizatorul introduce un mesaj si apasa butonul “send” |  |
|  | 4 |  | Serverul primeste mesajul, il proceseaza si ii trimite un raspuns utilizatorului |

|  |  |  |  |
| --- | --- | --- | --- |
| Nume | Send Voice Message | | |
| Actori | User | | |
| Descriere | Utilizatorul trimite un mesaj vocal chatbotului | | |
| Pre-condiții | Utilizatorul trebuie sa fie logat in aplicatie | | |
| Post-condiții | Utilizatorul primește un text corespunzator cu ce a inregistrat | | |
| Flux | **Nr.** | **Acțiune utilizator** | **Răspunsul sistemului** |
| 1 | Navigarea catre un chat existent sau un chat nou |  |
| 2 |  | Incarcarea mesajelor existente chatului |
| 3 | Utilizatorul tine apasat butonul “microphone” in timp ce rosteste mesajul dorit |  |
| 4 |  | Serverul primeste mesajul, il transforma in text si il trimite inapoi clientului |

Aplicația are o arhitectură client-server, în care clientul este o aplicație Android, iar serverul este compus din două module: un modul Spring Boot și un modul Flask. Pentru a implementa funcționalitatea de inteligență artificială în aplicație, care poate fi accesată doar cu ajutorul librăriilor Python, a fost necesară împărțirea codului între cele două module. In urmatoarele sectiuni voi prezenta cele cele doua module si aplicatia Android.

Backend.

In continuare pentru prezentarea componentei de Spring Boot a aplicatiei, voi aborda urmatoarea structura: endpointuri, servicii, repository(baze, de date), securitate, testare.

Endpointurile puse la dispozitie sunt cele responsabile de autentificare, de mesaje , conversatii si de recunoasterea vocii. Mai exact clasa MessageController, un controller REST gestioneaza cererile legate de message din aplicatie: metodele de saveMessage si de saveMessageWithPdf, care primeste ca parametru un MultipartFile, reprezentand PDF-ul la baza caruia este conversatia.

Componenta de servicii a aplicatiei este in mare parte controlata de clasa MessageServiceImpl, care se ocupa de crearea de mesaje si conversatii, apelul la repository, dar si la clasa de PythonServiceImpl. Aceasta se ocupa de apelul la modulul Flask al aplicatiei.

Modalitatea de comunicare intre module este prin apeluri HTTP, cu ajutorul clasei RestTemplate. Aceasta este un client HTTP sincron utilizat pentru a trimite cereri HTTP către servicii web externe și a primi răspunsuri de la acestea.

ResponseEntity<MessageResponse> response = restTemplate.postForEntity(  
 pdfMessageUrl,  
 requestEntity,  
 MessageResponse.class  
);

n ceea ce privește partea de repository, utilizez pachetul Spring JPA și interfața JpaRepository pentru a defini interfețele asociate obiectelor din aplicație. În exemplul dat, am definit o interfață care utilizează paginare pentru a prelua mesajele în funcție de conversația din care fac parte. Pentru a realiza acest lucru, am implementat interfața PagingAndSortingRepository.

public interface MessageRepository extends JpaRepository<Message, Integer>, PagingAndSortingRepository<Message, Integer> {  
 List<Message> findByChatId(Integer chatId);  
  
 Page<Message> findAllByChatIdOrderByIdDesc(Integer chatId, Pageable pageable);  
  
 void deleteByChatId(Integer chatId);  
}

Pentru gestionarea bazei de date, folosesc PostgreSQL și Flyway. PostgreSQL este un sistem de gestiune a bazelor de date relaționale puternic și fiabil. Flyway este o unealtă specializată în gestionarea migrațiilor bazelor de date. În esență, migrațiile sunt schimbările structurale pe care dorim să le aplicăm într-o bază de date pe măsură ce dezvoltăm aplicația.

În timpul dezvoltării, este posibil să adăugăm sau să modificăm tabele, coloane, indici sau alte aspecte ale bazei de date. Flyway ne ajută să gestionăm aceste modificări prin intermediul fișierelor de migrație.

Un fișier de migrație conține instrucțiuni SQL care descriu schimbările pe care dorim să le aplicăm în baza de date. Flyway urmărește aceste fișiere și le aplică în ordine pe măsură ce versiunea aplicației noastre evoluează. De asemenea, Flyway ține evidența migrațiilor care au fost deja aplicate în baza de date, astfel încât să evităm duplicarea modificărilor.

@RestController  
@RequestMapping("/messages")  
@RequiredArgsConstructor  
public class MessageController {  
 private final MessageService messageService;  
  
 @PostMapping  
 @ResponseStatus(HttpStatus.*CREATED*)  
 public MessageResponse saveMessage(@RequestBody MessageRequest messageRequest) {  
 return messageService.save(messageRequest);  
 }  
  
 @GetMapping("/{chatId}/{page}")  
 public List<MessageResponse> getMessages(@PathVariable Integer chatId, @PathVariable Integer page) {  
 return messageService.getMessagesByChat(chatId, page);  
 }  
  
 @PostMapping("/pdf")  
 public MessageResponse saveMessageWithPdf(@RequestPart MultipartFile file, @RequestPart String text) {  
 return messageService.saveMessageWithPdf(new MessagePdfRequest(file, text));  
 }  
}

Flask:

Modulul Flask oferă trei endpointuri principale pentru funcționalitatea de inteligență artificială: generarea de mesaje obișnuite, generarea de mesaje pe baza unui fișier PDF și conversia fișierelor audio în text.

Primul endpoint, primeste ca input o lista formata din mesajele dintre utilizator si bot corespunzatoare unei conversatii, pentru a putea fi incarcate in memoria botului, cu ajutorul librariei Langchain, descrise in capitolele anterioare. Clasa ConversationChain, este clasa care controleaza conversatia, ei ii asociem modelul de limbaj si tipul de memorie pe care dorim sa-l utilizam.

Am folosit pentru memorie tipul de memorie „Conversation Summary Buffer Memory”, pe care l-am considerat cel mai avantajos din punct de vedere al timpului, al costului si al preciziei.

memory = ConversationSummaryBufferMemory(llm=OpenAI(), max\_token\_limit=2500)

ConversationChain este obiectul caruia i-am asociat LLM-ul si memoria incarcata anterior. Am lasat parametrul de verbose pe True, pentru a putea vedea afisarile cu prompturile pe care le primeste modelul.

summary\_chain = ConversationChain(  
 llm=OpenAI(model="text-davinci-003"),  
 verbose=True  
)  
summary\_chain.memory = memory

Endpoint-ul următor are responsabilitatea de a genera mesaje bazate pe un fișier PDF. Pentru a realiza conexiunea între LLM (Language Model) și fișierul PDF, am utilizat Pinecone. Flow-ul acestei funcționalități este reprezentat în figura de mai jos.

Pentru a realiza acest lucru, am folosit obiectul RecursiveCharacterTextSplitter din Langchain pentru a împărți fișierul PDF în obiecte de tip Document, fiecare având o dimensiune de 1000 de caractere (chunk-uri). Ulterior, am utilizat un obiect de tip embedding de la OpenAI pentru a converti aceste documente în embeddings, adică într-un vector de numere. Aceste embeddings au fost încărcate într-un index Pinecone.

Mesajul va fi reprezentat de asemenea intr-un embedding, iar Pinecone va identifica legătura dintre documentele stocate în index și mesajul respectiv. După identificarea embeddings-urilor asociate cu mesajul, acestea vor fi utilizate în prompt-ul modelului de limbaj, permițându-i acestuia să furnizeze un răspuns corespunzător la întrebări.

O imagine care conține text, captură de ecran, Font, diagramă

Descriere generată automat

Ultimul endpoint se ocupă de conversia mesajelor audio în text. Pentru această funcționalitate, am optat pentru utilizarea bibliotecii SpeechRecognition, care folosește API-ul Google Cloud. Pentru a putea realiza conversia audio, am transformat fișierul într-un obiect de tip AudioFile, pe care l-am folosit in functia recognize.

r = sr.Recognizer()  
audio\_file = sr.AudioFile(filename)  
  
with audio\_file as source:  
 audio = r.record(source)  
  
return {"text": r.recognize\_google(audio, language='ro-RO')}

Android:

În cadrul proiectului, am aplicat arhitectura Model-View-ViewModel (MVVM) în dezvoltarea aplicației mele Android. MVVM este o abordare arhitecturală modernă care facilitează o separare clară a responsabilităților și o gestionare eficientă a codului, permițând îmbunătățirea experienței utilizatorului și ușurarea procesului de mentenanță. Modelul reprezintă datele și logica de business a aplicației, asigurând o gestionare eficientă a acestora. View-ul se concentrează pe afișarea informațiilor și interacțiunea cu utilizatorul, fără a conține logica de business. ViewModel-ul îndeplinește rolul de intermediar între Model și View, asigurând gestionarea logicii de afișare a datelor și facilitând comunicarea cu Modelul. Acesta furnizează View-ului metode și obiecte observabile(state-uri) pentru a actualiza interfața grafică în mod reactiv, în concordanță cu modificările apărute în Model.

Aplicatia oferă trei ecrane principale: ecranul de autentificare, de inregistrare și ecranul principal, care cuprinde conversațiile și mesajele si un drawer.

Aplicatia dispune de 2 navigari: o navigare intre ecranelele principale ale aplicatiei login/register si home si inca o navigare pentru home screen, cu ajutorul careia se navigheaza de la un chat la altul. Motivul pentru care am optat la cele 2 navigari este pentru ca atunci cand navighez intre chaturi sa se faca recompozitia doar la ecranul de chat. Astfel am reusit sa separ componenta de drawer si ceea de chat in 2 componente independente.

În cadrul arhitecturii MVVM, am implementat două ViewModele principale: drawerViewModel și chatViewModel, care corespund celor două componente de ecran. Partea de View a aplicației este reprezentată de funcțiile Composable, cum ar fi ChatScreen, DrawerScreen și LoginScreen. Pentru componenta de Model, am utilizat clase de date (data class) pentru Chat și Message. Clasele de repository sunt apelate de viewModele, iar aceste clase repository apelează clasele de data source, care la rândul lor fac apel la interfețele de API create cu ajutorul bibliotecii Retrofit.

Motivul pentru care am ales sa existe un pas in plus la apelurile api-urilor este pentru a putea face gestionarea erorilor mai usor, cu ajutorul obiectului Result:

suspend fun login(loginRequest: LoginRequest): Result<TokenHolder> {  
 return try {  
 Result.success(authApi.login(loginRequest))  
 } catch (e: Exception) {  
 Result.failure(e)  
 }  
}

Ecranul de Login sau Register apare doar la prima interactiune cu aplicatia, fiindca tokenul jwt primit in urma loginului este salvat in UserPreferences, prin libraria datastore. Astfel in momentul in care apare ecranul Splash al aplicatiei, este verificata existenta tokenului in UserPreferences, iar daca exista utilizatorul va fi redirectionat direct in pagina principala a aplicatiei.

init {  
 *viewModelScope*.*launch* **{** val token = sessionManager.token.firstOrNull()  
 if (token.*isNullOrEmpty*()) {  
 \_targetDestination.value = *LOGIN\_ROUTE* } else {  
 *println*("home\_route")  
 \_targetDestination.value = *HOME\_ROUTE* }  
 **}**}

Ecranul Home contine: drawer-ul mentionat mai sus este un ModalNavigationDrawer care contine lista de chaturi existente, butonul de „Logout” si butonul de „New Chat”.

ModalNavigationDrawer(  
 drawerState = drawerState,  
 drawerContent = **{** ModalDrawerSheet(modifier = Modifier.*fillMaxWidth*(.8f)) **{** DrawerScreen(  
 viewModel = viewModel,  
 onNewChatClick = onNewChatClick,  
 onNewPdfChatClick = onNewPdfChatClick,  
 onChatClick = onChatClick,  
 onLogoutSuccessful = onLogoutSuccessful  
 )  
 **}  
 }**,  
 content = content  
)

De asemenea ecranul Home contine si ecranul chaturilor, care este compus dintr-un Scaffold:

Scaffold(  
 topBar = **{** AppBar(  
 chatTitle = state.chatTitle,  
 onMenuClick = onMenuClick,  
 onDeleteClick = **{** viewModel.deleteChat()  
 **}**,  
 onUpdateClick = **{** chatTitle **->** viewModel.updateChat(chatTitle)  
 **}**,  
 onNewPdfChatClick = onNewPdfChatClick,  
 )  
 **}**,  
 bottomBar = **{** MessageSection(  
 messageState = viewModel.messageState,  
 onSendMessage = **{** text **->** viewModel.sendMessage(text)  
 **}**,  
 onSendPdfMessage = **{** text, path **->** viewModel.onSendPdfMessage(text, path)  
 **}** )  
 **}**) **{** MessagesSection(  
 modifier = Modifier.*padding*(  
 bottom = **it**.calculateBottomPadding(),  
 top = **it**.calculateTopPadding()  
 ),  
 state = state,  
 messageState = messageState,  
 loadMoreMessages = viewModel::loadMoreMessages  
 )  
**}**

Componenta Scaffold ajută la organizarea interfeței utilizatorului într-o structură coerentă. Este utilizat pentru a crea o structură pentru o aplicație de chat.

Bara de sus (topBar) conține un AppBar personalizat, care afișează titlul conversației, butoanele de meniu și permite utilizatorului să efectueze acțiuni precum ștergerea conversației sau actualizarea titlului acesteia.

Bara de jos (bottomBar) conține o secțiune de mesaje (MessageSection), unde utilizatorul poate introduce și trimite mesaje text sau se poate inregistra pentru a trimite un mesaj audio.

În interiorul Scaffold, conținutul principal al ecranului este reprezentat de MessagesSection. Acesta afișează mesajele existente și oferă opțiunea de a încărca mai multe mesaje prin paginare, atunci când utlizatorul face scroll in sus. Folosește un Modifier pentru a ajusta spațiul între conținutul secțiunii și marginile ecranului.

Secțiunea MessagesSection este definită de codul de mai jos, care folosește LazyColumn pentru a afișa mesajele într-o listă.

LazyColumn(  
 state = listState,  
 modifier = modifier.*padding*(horizontal = 16.*dp*),  
 reverseLayout = true  
) **{** if (messageState.isLoading) {  
 item **{** LoadingMessageItem() **}** }  
 item **{** if (state.isLoading) {  
 Row(  
 modifier = Modifier  
 .*fillMaxWidth*()  
 .*padding*(8.*dp*),  
 horizontalArrangement = Arrangement.Center  
 ) **{** CircularProgressIndicator()  
 **}** }  
 **}** *itemsIndexed*(state.items) **{** index, message **->** if (index >= state.items.size - 1 && !state.endReached && !state.isLoading) {  
 loadMoreMessages()  
 }  
 MessageItem(  
 messageText = message.text,  
 type = message.type,  
 isLast = index == state.items.*lastIndex* )  
 **}  
}**

În acest cod, LazyColumn este utilizat pentru a crea o listă de elemente afișate vertical. El primește câteva parametri pentru a controla comportamentul și aspectul listei. Parametrul state este utilizat pentru a derula lista mesajelor la ultimul mesaj, atunci cand se trimite un mesaj sau se primeste un mesaj.

În interiorul LazyColumn, se găsesc mai multe elemente (item și itemsIndexed) care definesc modul în care sunt afișate mesajele.

Primul element este LoadingMessageItem(), care este afișat doar atunci când messageState.isLoading este true. Acesta este utilizat pentru a afișa un indicator de încărcare în timp ce se așteaptă răspunsul la un mesaj.

//TODO poza cu loading

CircularProgressIndicator este componenta care se afiseaza in timp ce se incarca mesajele de la server.

Apoi, sunt afișate elemente individuale pentru fiecare mesaj în lista state.items. În interiorul acestei bucle, există o verificare pentru a încărca mai multe mesaje (loadMoreMessages()) atunci când se ajunge la sfârșitul listei și încă nu s-a atins limita (state.endReached) și nu se încarcă (state.isLoading).

Pentru fiecare mesaj, se afișează un MessageItem, care primește textul mesajului, tipul acestuia și indică dacă este ultimul mesaj din listă (isLast). Aceasta permite afișarea corespunzătoare a stilurilor sau indicatorilor vizuali pentru ultimul mesaj.

Am utilizat biblioteca Wave Recorder pentru înregistrarea vocii. Aceasta este o bibliotecă scrisă în Kotlin, permite înregistrarea audio în diferite formate. Este ușor de utilizat: instantiem obiectul WaveRecorder cu locația unde dorim să salvăm înregistrarea audio și apelăm funcțiile startRecording/stopRecording.

val filePath:String = externalCacheDir?.absolutePath + "/audioFile.wav"

val waveRecorder = WaveRecorder(filePath)

waveRecorder.startRecording()

waveRecorder.stopRecording()

\footnote{https://github.com/squti/Android-Wave-Recorder}.

Pentru extragerea fisierelor PDF am folosit libraria Compose Multiplatform File Picker, personalizata pentru proiectele Jetpack Compose.

Pentru a face vizibila componenta FilePicker setez parametrul showFilePicker pe true la apasarea unui buton. Functia handleUri, primeste un path de tipul „android file” salveaza fisierul in cache, pentru a putea fi trimis la backend.

FilePicker(show = showFilePicker, fileExtensions = *listOf*("pdf")) **{** path **->** showFilePicker = false  
 val myPath = uriPathFinder.handleUri(context, Uri.parse(path!!.path))  
 onNewPdfChatClick(myPath!!)  
 Toast.makeText(  
 context,  
 "Pdf file saved successfully!",  
 Toast.*LENGTH\_SHORT* ).show()  
**}**

\footnote{ <https://github.com/Wavesonics/compose-multiplatform-file-picker>}.

1. https://python.langchain.com/en/latest/index.html [↑](#footnote-ref-1)
2. <https://www.udemy.com/course/speaker-recognition/learn/lecture/31414280#content> [↑](#footnote-ref-2)
3. ibidem [↑](#footnote-ref-3)
4. Ibidem [↑](#footnote-ref-4)
5. https://www.sciencedirect.com/topics/medicine-and-dentistry/hidden-markov-model [↑](#footnote-ref-5)
6. https://jonathan-hui.medium.com/speech-recognition-feature-extraction-mfcc-plp-5455f5a69dd9 [↑](#footnote-ref-6)