

UNIVERSITATEA TEHNICĂ „Gheorghe Asachi” din IAȘI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DOMENIUL: CALCULATOARE ȘI TEHNOLOGIA INFORMAȚIEI
SPECIALIZAREA: TEHNOLOGIA INFORMAȚIEI

Aplicație web bazată pe servicii ce facilitează interacțiunea
pacienților cu medicii

LUCRARE DE DIPLOMĂ

Coordonator științific
Ș.l. dr. Inf. Dumitriu Tiberius

Absolvent
Munteanu Letiția-Ioana

DECLARAȚIE DE ASUMARE A AUTENTICITĂȚII LUCRĂRII DE DIPLOMĂ

Subsemnata Munteanu Letiția-Ioana, legitimată cu seria XT nr. 779587, CNP 2981207226703, autorul lucrării Aplicație web bazată pe servicii ce facilitează interacțiunea pacienților cu medicii, elaborată în vederea susținerii examenului de finalizare a studiilor de licență organizat de către Facultatea de Automatică și Calculatoare din cadrul Universității Tehnice „Gheorghe Asachi” din Iași, sesiunea iulie a anului universitar 2021-2022, luând în considerare conținutul Art. 34 din Codul de etică universitară al Universității Tehnice „Gheorghe Asachi” din Iași (Manualul Procedurilor, UTI.POM.02 – Funcționarea Comisiei de etică universitară), declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române (legea 8/1996) și a convențiilor internaționale privind drepturile de autor.

Data
06.07.2022

Semnătura



Cuprins

Introducere	1
Capitolul I. Fundamente teoretice.	3
I.1 Terminologie specifică	3
I.2 Algoritmi utilizați în clasificare	8
Capitolul II. Arhitectura aplicației	10
II.1 Arhitectură generală	10
II.2 Arhitectura bazei de date	13
II.3 Servicii	16
II.4 Componente	18
Capitolul III. Implementarea și utilizarea aplicației	20
III.1 Interfață și funcționalități	20
III.2 Implementarea aplicației	28
Concluzii	36
Bibliografie	37
Anexe	39
Anexa 1: Codul pentru decodificarea JWT-ului	39
Anexa 2: Codul pentru funcția <code>successfulAuthentication</code>	39
Anexa 3: Implementarea filtrului de autorizare	40
Anexa 4: Codul pentru configurarea web	40
Anexa 5: Codul pentru încărcarea analizelor	41
Anexa 6: Codul pentru calcularea funcției MinGini	42
Anexa 7: Codul pentru funcția BuildTree	42
Anexa 8: Codul pentru funcția RandomForest	43
Anexa 9: Codul pentru funcția Predict	43
Anexa 10: Paginile aplicației client	44

Aplicație web bazată pe servicii ce facilitează interacțiunea pacienților cu medicii

Munteanu Letiția-Ioana
Rezumat

Tema aleasă are ca scop facilitarea interacțiunii dintre medici și pacienți printr-o aplicație web care pune la dispoziție o serie de servicii în acest scop. Astfel, utilizatorii își pot crea un cont la care mai apoi au acces, își pot face programări la medicii doriți, își pot încărca analizele în vederea aflării unui diagnostic și al unui set de medicații corespunzător fiecăruia și își pot vedea istoricul medical. Totodată, în vederea ajutării utilizatorilor în hotărârea unei specializări medicale spre care ar trebui să se îndrepte pentru un control, aplicația oferă un serviciu de prelucrare a datelor, ce constă în simptomele acestora și stabilirea unei specializări pe baza datelor introduse.

Atât contextul pandemic precum și interesul în a ajuta alte persoane au dus la alegerea temei de licență, dezvoltarea unei aplicații web ce reprezintă o platformă online pentru spitalizare, făcând astfel mai ușor de accesat serviciile medicale. Astfel, oamenii pot folosi această platformă pentru a beneficia de consultații online, fără a fi nevoiți să se deplaseze la centrele medicale.

Capitolul I cuprinde informații legate despre fundamentele teoretice. Aici sunt descrise în amănunt tehnologiile folosite la dezvoltarea aplicației atât la nivel de server, cât și la nivel de client. De asemenea sunt descriși și algoritmi de clasificare folosiți în aplicație.

Capitolul II cuprinde informații despre arhitectura aplicației, incluzând arhitectura generală a acesteia și arhitectura bazei de date. Este detaliat fiecare serviciu, interacțiunea dintre acestea, componentele serviciilor și de asemenea, legăturile dintre ele. Totodată acest capitol cuprinde și informații despre cum a fost folosit algoritmul de Machine Learning, Random Forest, în procesul de prelucrare a simptomelor introduse de către utilizatori și clasificarea acestora pentru stabilirea unei specializări medicale.

Capitolul III cuprinde informații despre modul de utilizare al aplicației, fiind arătate secvențele semnificative ale acesteia, precum și despre implementarea propriu zisă a aplicației. Sunt explicate părțile semnificative ale serviciilor ce compun aplicația și, de asemenea, modul în care s-a utilizat algoritmul Random Forest. Sunt descrise impedimentele avute pe parcursul implementării și metodele de rezolvare.

La final apar concluziile, anexele ce conțin secvențele cele mai semnificative din implementarea aplicației și bibliografia.

Componentele hardware folosite sunt laptop Lenovo Yoga 7i, memorie RAM 16GB și memorie de tip SSD de 1TB. Rularea aplicației presupune folosirea unei memorii RAM de 1250 MB/s de către IntelliJ IDEA și 1100 MB/s de către PyCharm, în care a fost implementată aplicația orientată pe server și 193 MB/s de către Visual Studio Code, în care a fost implementată aplicația client.

La nivelul componentelor software s-au utilizat următoarele tehnologii: Spring-Boot în versiunile 2.5.9, 2.6.7 și 2.7.0 împreună cu limbajul de programare Java având JDK 11, Python în versiunea 3.10.4 și React cu versiunea 17.0

Introducere

Contextul pandemiei COVID-19 a avut un impact semnificativ asupra tuturor domeniilor, generând schimbări majore în viețile oamenilor. Fiind nevoiți să reducem pe cât de mult posibil interacțiunea fizică cu alte persoane, deplasările în spații publice și să respectăm alte reguli specifice acestei perioade, activitatea în mediul online a luat amploare în majoritatea domeniilor. Odată cu această schimbare, multe dintre serviciile de care oamenii au nevoie s-au dezvoltat pe partea virtuală, facilitându-ne astfel utilizarea lor.

Viața fiecăruia presupune mai devreme sau mai târziu necesitatea serviciilor medicale, fie în situații limită, fie în situații de prevenție a acestora. Conform studiilor¹ făcute pe această temă s-a sesizat faptul că din cauza accesului limitat sau restricționat la sistemele de sănătate în perioada pandemiei, precum și din alte motive des întâlnite ca teama oamenilor de a intra în contact cu spitalele, cabinetele sau centrele medicale de frică de a nu se îmbolnăvi de COVID-19, aceștia renunță sau amână monitorizarea propriei sănătăți. Statisticile arată că 43% dintre românii din mediul urban au mers mai puțin sau deloc la medicul de familie, iar 50% la medicul specialist. Un procent de 23% nu și-au făcut deloc analize la sânge și un altul de 29% nu au apelat la investigații, monitorizări sau ecografii de monitorizare sau prevenție. Totodată, datele arată faptul că interesul pentru a merge la medic a fost mai scăzut mai ales în rândul persoanelor cu venituri mai mici. În contextul pandemiei, în rândul persoanelor cu afecțiuni cronice s-a înregistrat următorul procentaj: 46% au apelat mai rar sau deloc la medicul specialist și 42% nu și-au făcut analize de sânge.

Un alt factor important în accesarea serviciilor medicale îl joacă și disponibilitatea deplasării către un astfel de centru, înregistrându-se un număr mare de persoane private de asemenea servicii din cauza mediului în care trăiesc. Acesta este în strânsă legătură cu disponibilitatea financiară a oamenilor, deplasările în alte orașe sau centre regionale care dispun de servicii medicale fiind pentru mulți costisitoare. Așadar, îngrădiți de toți acești factori, oamenii aleg să-și lase sănătatea deoparte, uneori, neavând de ales.

O soluționare a acestei problematice majore ar fi permiterea accesului la servicii medicale oricărei persoane, luându-se în calcul faptul că mulți oameni nu locuiesc în zone ce permit deplasarea către un centru medical și că nu toți dispun de venituri suficiente pentru a face acest lucru în mod frecvent.

Există abordări anterioare ale acestei idei: platforme online care permit utilizatorilor accesarea serviciilor medicale puse la dispoziție de site-uri precum Regina Maria², Arcadia³, ZenMed⁴. Prin intermediul acestor platforme publicul are posibilitatea de a-și crea programări către doctorii aleși, de a-și vedea rezultatul analizelor pe baza unui cod unic per client și analiză sau de a afla informații despre posibilele topicuri de interes din aria medicală. Totuși, această abordare nu exclude necesitatea deplasării pacienților către centrele medicale.

O situație des întâlnită în viața de zi cu zi a oamenilor este incapacitatea de a ști în ce direcție trebuie să se îndrepte atunci când au nevoie de suport medical. Acest lucru este raportat mai ales în rândul oamenilor care nu au cunoștințe medicale care să-i ajute în acest sens sau în rândul tinerilor care nu au experiențe medicale din care să-și dea seama de acest lucru. Un ajutor pentru cei care necesită astfel de servicii dar nu știu spre ce arie medicală trebuie să apeleze ar fi

¹ <https://www.ipsos.com/ro-ro/retrospectiva-privind-accesarea-serviciilor-medicale-pandemie>

² <https://www.reginamaria.ro/>

³ <https://www.arcadiamedical.ro/>

⁴ <https://www.zenmedical.ro/>

existența unui mecanism care le permite oamenilor să își introducă simptomele avute iar aceștia să primească un răspuns cu specializarea medicală la care ar trebui să-și facă un control. Această abordare este una nouă, platformele menționate mai sus nepunând la dispoziție un astfel de serviciu.

Așadar, această temă de licență își propune crearea unei aplicații ce îndeplinește toate aceste condiții: ușurința accesării serviciilor medicale, posibilitatea folosirii lor fără a trebui să se deplaseze către centrele specializate în acest domeniu în situațiile care permit acest lucru și crearea unui serviciu prin care oamenii primesc recomandări în ce direcție ar trebui să se îndrepte atunci când nu știu, prin simpla introducere a simptomelor.

Capitolul I. Fundamente teoretice.

I.1 Terminologie specifică

Lucrarea de licență este dezvoltată cu ajutorul următoarelor tehnologii: partea de backend este implementată folosind o arhitectură bazată pe servicii. Pentru facilitarea acestui lucru se folosesc Spring Boot, utilizând ca limbaj de programare Java și FastApi, utilizând Python. Partea de frontend este realizată utilizând biblioteca ReactJS, codul fiind scris în JavaScript. În ceea ce privește procesarea datelor introduse de către utilizator pentru obținerea unei recomandări de specializare medicală s-a folosit un algoritm de Machine Learning, Random Forest. Pentru stocarea datelor s-au folosit ca baze de date MySQL și MongoDB.

Spring Boot este un tool care facilitează dezvoltarea de aplicații web, servicii și microservicii utilizând framework-ul **Spring**, făcând acest proces mai ușor și rapid prin cele trei capabilități oferite:

- autoconfigurare: aplicațiile sunt inițializate cu dependențe prestabilite, pe care utilizatorul nu trebuie să le configureze manual
- posibilitatea de a configura dependențele în funcție de cerințele proiectului
- crearea unei aplicații de sine stătătoare care rulează de una singură, fără a se baza pe surse externe, acest lucru fiind datorat încorporării unui server web la inițializarea aplicației (Tomcat 7 este serverul utilizat by default de Spring Boot) [3]

De asemenea, securitatea este asigurată într-o astfel de aplicație, Spring Boot având ca standard de securitate **Spring Security**.

Spring Security este un framework prin care se gestionează controlul accesului într-o aplicație dezvoltată în Java. Acesta oferă funcționalități atât din punct de vedere al autentificării, cât și din punct de vedere al autorizării, fiind foarte flexibil în ceea ce privește parametrizarea, personalizarea și configurarea în vederea satisfacerii cerințelor dezvoltatorilor de aplicații [12].

Infrastructura web a Spring Security se bazează în întregime pe filtre standard de servleți. Acesta nu folosește intern servleți sau alte framework-uri bazate pe servleți, așadar, nu are legături către o tehnologie web particulară. Lucrul cu `HttpServletRequest` și `HttpServletResponse` asigură flexibilitate în primirea cererilor de la diferiți clienți, cum ar fi: browsere, servicii web client, `HttpInvoker` sau aplicații AJAX. Spring Security menține intern o înlanțuire de filtre, fiecare având responsabilitate proprie. Acestea sunt adăugate sau șterse din configurație în funcție de serviciile cerute. De asemenea, ordinea filtrelor este importantă, acest lucru fiind dat de dependențele dintre filtre [13].

Așadar, Spring Security asigură funcționalități pentru: autentificare, autorizare și filtrarea servleților.

Autentificarea reprezintă procesul prin care se validează identitatea unui utilizator, pe când **autorizarea** este procesul prin care utilizatorului autentificat i se permite accesul la diferite resurse. Spring Security are o arhitectură proiectată să separe autentificarea de autorizare.

În contextul Spring Security, autentificarea este reprezentată de tokenul de autentificare sau pur și simplu, autentificarea. Acesta conține informațiile necesare procesului de autentificare, ele fiind:

- **numele de utilizator**, prin care este identificat userul, fiind de obicei o instanță a `UserDetails` atunci când autentificarea este făcută cu username și parolă
- **credențialele**, de obicei parola

- **permisiunile**, o colecție de roluri sau scopuri care vor fi folosite mai apoi în configurarea proceselor de autorizare

Autentificarea poate fi privită ca fiind input-ul către **AuthenticationManager**, API-ul care definește cum realizează filtrele Spring Security autentificarea. În urma unei astfel de cereri care a fost procesată de către AuthenticationManager prin metoda **authenticate**, tokenul de autentificare este salvat în **SpringContextHolder**, aici fiind salvate detaliile celor autentificați. Acesta folosește un thread local pentru a salva informațiile necesare, iar pentru a afla userul curent autentificat se folosește **SecurityContext**, care conține un obiect Authentication. Deși se folosește un singur thread pentru salvarea acestor informații, **ProviderManager**, va șterge informațiile sensibile din Authentication, obiectul returnat atunci când o cerere de autentificare este făcută cu succes. Provider Manager este cea mai folosită implementare a AuthenticationManager. Acesta iterează o cerere de autentificare printr-o listă de **AuthenticationProvider**, care realizează diferite tipuri de autentificare, până când găsește tipul de autentificare căutat [14], [15].

Există mai multe mecanisme de autentificare, cele mai populare fiind:

- autentificarea cu nume și parolă
- OAuth 2.0 Login
- SAML 2.0 Login
- Central Authentication Server (CAS)
- Remember me

Cel mai des întâlnit mecanism de autentificare este cel care folosește credențialele unui client, parolă și nume de utilizator, pentru a-i verifica identitatea. Pentru acest tip de autentificare există câteva metode prin care poate fi utilizat:

- Form Authentication
- Basic Authentication
- Digest Authentication

Majoritatea aplicațiilor folosesc Form Authentication, sau mai simplu spus, autentificarea făcută cu ajutorul unui formular HTML în care clientul își poate introduce numele de utilizator și parola. Flow-ul procesului de autentificare prin această metodă este următorul:

1. Clientul își introduce credențialele și le trimite, acest lucru concretizându-se într-o cerere care ajunge la server
2. Este ales filtrul corect care procesează cererea venită. Având în vedere că cererea trimisă este de tipul POST, conținând parola și numele utilizatorului, filtrul ales de către Spring Security este **UsernamePasswordAuthenticationFilter**.
3. Mai apoi, acest filtru este trimis mai departe către AuthenticationManager, care, prin implementarea sa, iterează prin lista de AuthenticationProvider și încearcă să facă autentificarea pe baza obiectului de tip Authentication primit
4. Dacă autentificarea nu s-a făcut cu succes, atunci SecurityContextHolder este eliberat și este invocat **AuthenticationFailureHandler**
5. Dacă autentificarea s-a făcut cu succes, **SessionAuthenticationStrategy** este notificat de acest lucru iar obiectul Authentication este setat în SecurityContextHolder [16], [17].

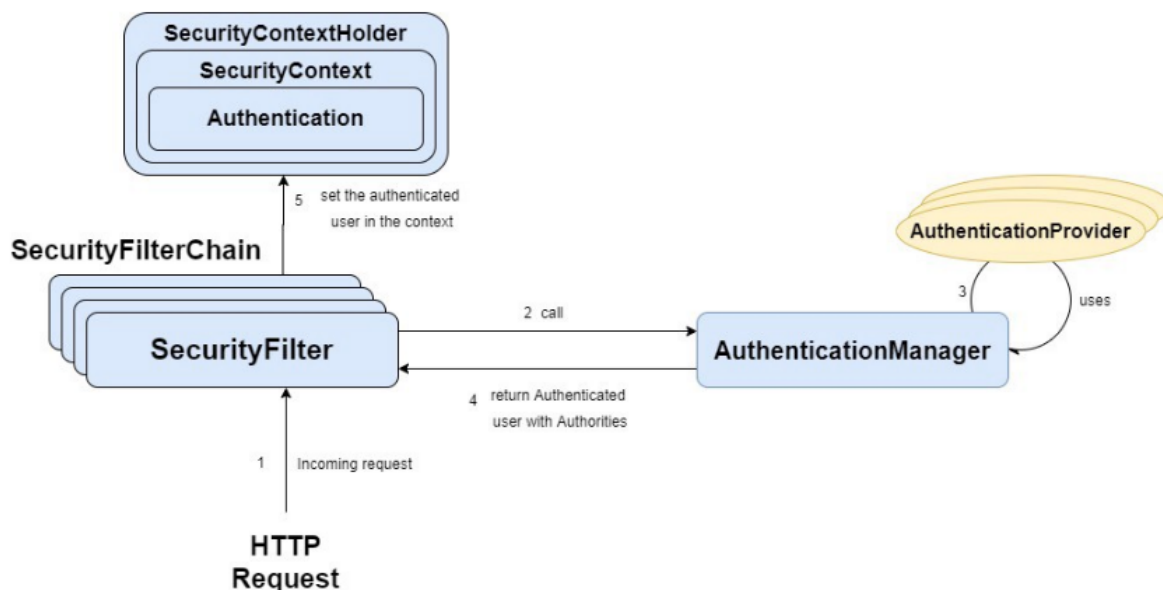


Fig.1.1.1: Procesul de autentificare [16]

Detaliile utilizatorilor sunt stocate într-o bază de date. Pentru a putea accesa detaliile acestora este necesară folosirea **UserDetailsService**, o interfață formată dintr-o singură metodă care returnează un obiect de tipul **UserDetails**, obiect ce conține toate informațiile unui client. De asemenea, pentru a putea verifica parola, care este stocată în baza de date sub formă criptată, trebuie creat un **Bean** corespunzător, care nu este altceva decât un obiect instanțiat și gestionat de către container-ul Spring, având ca scop furnizarea unui serviciu în acest caz. Bean-ul **PasswordEncoder** are ca scop stabilirea unui model de criptare a parolilor, de exemplu **BCryptPasswordEncoder**, oferind mai apoi funcționalități de decriptare și verificare a acestor date.

Autorizarea utilizatorilor către anumite resurse poate fi stabilită atât în configurarea web a aplicației adăugând în metoda **antMatchers** rolul pentru care accesul este permis, cât și la nivel de controller al aplicației, adăugând adnotarea **PreAuthorize** cu specificarea rolului corespunzător.

FastAPI este un framework modern și rapid folosit pentru dezvoltarea aplicațiilor web în Python. Principalele facilități oferite de acesta sunt:

- rapiditate, înregistrând performanțe înalte, fiind unul dintre cele mai rapide framework-uri valabile pentru Python
- reducerea erorilor, reducând 40% din erorile induse de către programatori
- robustețe, oferind cod pentru producție, având documentație interactivă

Acesta oferă flexibilitatea de a utiliza diferite servere pentru dezvoltarea aplicațiilor, unul dintre cele mai folosite fiind **Uvicorn** [18].

ReactJS este o librărie JavaScript folosită pentru a crea interfețe utilizator interactive. După cum este și denumirea, “React” reacționează la stările aplicației, făcând update-uri ori de câte ori se schimbă datele. React are o arhitectură bazată pe componente, fiecare administrându-și stările proprii. Acestea sunt compuse mai apoi pentru a construi un UI complex [8].

Componentele sunt segmente de cod independente și în același timp, reutilizabile, reprezentând baza construirii unei aplicații în React. Acestea au același scop ca și funcțiile JavaScript, dar în schimb, returnează elemente HTML. Există două tipuri de componente, iar acestea sunt:

- **componente funcționale**
- **componente de tip clasă**

Componentele funcționale sunt funcții JavaScript care pot avea sau nu parametri, proprietăți și care returnează elemente JSX (HTML și JavaScript). Acestea nu au stare proprie și nici metode care să gestioneze ciclul de viață al acestora, deși aceste lucruri pot fi adăugate folosind funcționalitățile oferite de către React, numite Hook-uri. Scopul principal al componentelor funcționale este acela de a afișa informații.

Componentele de tip clasă sunt mai complexe decât cele funcționale, oferind posibilitatea de a trimite date de la o clasă la alta. Spre deosebire de componentele funcționale, acestea au stare proprie și metode de gestionare a ciclului de viață, cum ar fi **componentDidMount**, **componentDidUpdate** sau **componentWillUnmount**. Există diferențe și din punct de vedere al sintaxei. Componentele de tip clasă trebuie să extindă **React Component** și trebuie să aibă o metodă numită **render** pentru a returna elemente **JSX**⁵ [19].

Hook-urile sunt noi funcționalități oferite de librăria React începând cu versiunea 16.8 având scopul de a gestiona starea componentelor funcționale. Acestea nu pot fi utilizate în componentele de tip clasă. Cele mai utilizate astfel de funcții sunt:

1. **Hook-urile de stare** prin care se poate seta sau se poate lua starea unei componente utilizând funcția **useState**.
2. **Hook-urile efect** care permit efectuarea unor acțiuni la randarea componentei, cum ar fi actualizarea DOM-ului, extragerea datelor de la un server sau procesarea lor, setarea anumitor informații. Acestea sunt echivalente cu metodele de gestiune a ciclului de viață al componentelor de tip clasă **componentDidMount**, **componentDidUpdate** și **componentWillUnmount**.
3. **Hook-uri personalizate** care sunt funcții JavaScript având o sintaxă specifică, numele funcției începând cu **use**.
4. **Hook-urile de context** care sunt folosite pentru a returna valoarea curentă unui anumit context. Acestea pot fi trimise într-o ierarhie de tip părinte-copil de componente.
5. **Hook-urile useCallback** folosite pentru a memora starea curentă a unei componente având ca scop ne-randarea componentelor copil atunci când nu este necesar.

⁵ JSX este o sintaxă asemănătoare XML/HTML utilizată de React, care extinde ECMAScript, astfel încât textul asemănător XML/HTML să poată coexista cu codul JavaScript. Sintaxa este destinată să fie utilizată de preprocesoare pentru a transforma textul asemănător HTML găsit în fișierele JavaScript în obiecte JavaScript standard. Practic, acesta permite scrierea de cod HTML și JavaScript în același fișier, expresiile fiind mai apoi transformate în cod JavaScript real [31].

6. **Hook-urile useMemo** care returnează valoare memorată a unei componente ajutând în optimizarea performanțelor.
7. **Hook-urile useRef** ce returnează o referință către un obiect prin proprietatea **current**, fiind mutabile. Scopul principal este de a accesa o componentă copil.
8. **Hook-urile useParams** ce returnează un obiect de tip cheie-valoare a parametrilor dinamici pentru URL-urile curente. Componentele copil moștenesc acești parametri de la componentele părinte [20], [21], [22].

De asemenea, mai există și **componente stilizate** care permit utilizarea CSS pentru a stiliza componentele React. Acestea oferă următoarele funcționalități:

- CSS automat prin monitorizarea componentelor randate într-o pagină și injectarea stilului specific acestora și nimic mai mult.
- Înlăturarea conflictelor date de numele claselor deoarece acestea generează nume unice pentru fiecare componentă stilizată, neexistând mai apoi duplicate.
- Stilizare simplă și dinamică, adaptând stilul unei componente bazat pe proprietățile acesteia.
- Menținere ușor de întreținut
- Prefixare CSS automată [23].

MySQL este o bază de date dezvoltată de către Oracle, fiind o sursă deschisă, accesibilă folosirii de publicul larg. Aceasta este o bază de date relațională, bazată pe limbajul structurat de interogare (Structured Query Language - SQL). Rulează pe platforme virtuale, cum ar fi Linux, UNIX sau Windows, fiind folosită mai ales în aplicații web, bazându-se pe un model client-server. Nucleul acesteia este serverul MySQL care gestionează toate comenzile sau instrucțiunile bazei de date. Este valabil ca un program separat într-o rețea de medii de lucru de tipul client-server online și de asemenea ca o librărie care poate fi adăugată separat în aplicații.

Avantajele acestei baze de date sunt:

- permite stocarea și accesarea datelor aflate în mai multe motoare de stocare precum InnoDB, CSV și NDB
- replicarea și partiționarea tabelor, oferind performanță și durabilitate
- suport pentru o gamă variată de date precum float, double, char, varchar, binary, varbinary, text, blob, date, time, datetime, timestamp, year, set, enum
- oferă securitate ridicată
- oferă suport pentru diferite protocoale de comunicație, cum ar fi TCP/IP pe diferite platforme [28].

MongoDB este o bază de date non-relațională folosită pentru a stoca volume mari de date. Structura acesteia este formată din colecții care conțin seturi de documente, ce sunt perechi de tip cheie-valoare. Oferă o flexibilitate mai mare spre deosebire de bazele de date relaționale, avantajele acesteia fiind:

- documentele pot varia în ceea ce privește numărul de proprietăți, mărime și conținut
- documentele nu trebuie să aibă o schemă predefinită, existând posibilitatea de a adăuga proprietăți pe parcurs
- scalabilitate
- modelul de date pus la dispoziție oferă posibilitatea de a crea ierarhii de relații pentru stocarea datelor complexe într-o manieră mai ușoară

- crearea indecșilor pentru îmbunătățirea performanțelor căutărilor, fiecare proprietate putând fi indexată
- load balancing, concept se referă la împărțirea datelor către mai multe instanțe de MonfoDB care pot rula pe diferite servere, aducând îmbunătățiri la nivel de system și totodată gestionând erorile hardware [29].

1.2 Algoritmi utilizați în clasificare

A. Arbori de decizie

Utilitatea arborilor de decizie poate fi observată în diferite contexte cum ar fi clasificarea textului, extragerea textului, compararea statistică a datelor, acestea fiind doar câteva exemple. Domeniul aplicabilității acestora este unul vast și variat, fiind o ustensilă foarte bună mai ales în domeniul medical, unde apare nevoia de a diagnostica unele boli, acest lucru având un aport considerabil în evoluția medicinei și tratarea bolilor.

Algoritmii bazați pe arbori de decizie oferă reguli de clasificare ușor de înțeles pentru oameni. În ciuda acestui fapt, există și unele dezavantaje, cel mai mare fiind sortarea atributelor atunci când arborele trebuie să separe un nod. Acest lucru devine costisitor, având ca efect scăderea eficienței atunci când datele folosite sunt de dimensiuni mari.

B. Random Forest

În 2001, Leo Breiman a prezentat ideea de Random Forest (Păduri aleatoare), dovedindu-se a înregistra performanțe mai bune decât alți algoritmi de clasificare precum Rețelele Neuronale sau Mașini Vectoriale de Sprijin (Support Vector Machine - SVM). Acest algoritm are la bază conceptul de selecție aleatoare de subseturi de date îmbinat cu folosirea de diferiți clasificatori, lucru ce oferă diversitate, dovedindu-se a fi mult mai eficient.

Random Forest este format dintr-o familie de vectori identic distribuiți aleator. Fiecare arbore este format din subseturi aleatorii de date sau caracteristici, selectate din întregul set de date. Acest concept poartă denumirea de Bagging și are ca efect îmbunătățirea predicțiilor, rezultând eficiențe ridicate.

Avantajele acestei abordări sunt:

- depășirea problemei de supraadaptare (engl. overfitting) în procesul de antrenare a rețelei
- oferă o modalitate eficientă de gestiune a datelor inconsistente
- oferă o acuratețe mai mare decât arborii de decizie simpli
- în fiecare arbore, subsetul de caracteristici este selectat aleatoriu la formarea nodului [24].

Arborii de decizie din Random Forest sunt formați din subseturi aleatorii de date din întregul set de date, având trei componente: nodurile de decizie, nodurile frunză sau terminale și nodul rădăcină. Algoritmii arborilor de decizie împarte setul de date în ramuri care la rândul lor sunt separate în alte ramuri, continuând în această manieră până se ajunge la nodul frunză care nu mai poate fi împărțit. Nodurile din arborii de decizie reprezintă attributele sau caracteristicile folosite pentru a prezice rezultatul [25].

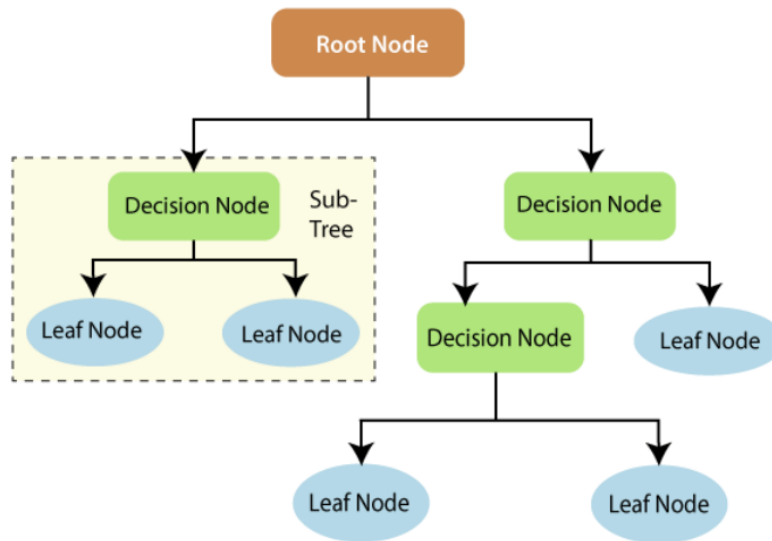


Fig. 1.2.1: Formarea arborilor de decizie [25]

Pe baza unor metrice de calculare se poate măsura gradul de incertitudine în alegerea subseturilor de date, acest lucru putând fi redus. Cele mai cunoscute două metrice sunt:

- entropia
- indexul Gini

Entropia poate fi definită ca ordinul de dezordine, indicând ca și selecție optimă caracteristica cu valoarea cea mai mică. Aceasta se calculează ca o sumă de produse folosind probabilitatea elementelor de a aparține unei clase, formula matematică fiind [26]:

$$E(S) = \sum_{i=1}^c - p_i \log_2 p_i$$

Indexul Gini reprezintă impuritatea, măsurând frecvența la care elementele alese în mod aleatoriu sunt etichetate greșit. Valoarea minimă a indexului Gini este 0, acesta fiind cel mai favorabil caz, atunci când toate elementele alese fac parte din clase distincte. Așadar, acest nod nu va mai fi împărțit. Cu cât indexul Gini este mai mic, cu atât va crește acuratețea. Acesta se calculează ca sumă între probabilitățile elementelor de a aparține unei clase, având formula matematică egală cu: [27]:

$$GiniIndex = 1 - \sum_j p_j^2$$

Capitolul II. Arhitectura aplicației

II.1 Arhitectură generală

Aplicația are o arhitectură bazată pe servicii SOA⁶ (engl. Service Oriented Architecture), fiind proiectată după modelul multistrat (engl. multilayer), format din trei straturi (engl. layer). Acestea sunt: layer-ul de prezentare, layer-ul aplicației și layer-ul de date, între care există o comunicare bidirecțională de tipul cerere-răspuns (engl. request-response).

Layer-ul de prezentare este primul și cel mai abstract layer, care, după cum este denumit, prezintă aplicația sub forma unei interfețe grafice utilizator, care comunică cu layerul aplicației. Acesta poate fi accesat prin intermediul oricărui dispozitiv client, cum ar fi: laptop, desktop, tabletă, telefon mobil ș.a.m.d. Pentru conținutul afișat utilizatorului, paginile web trebuie preluate de browserul web sau de altă componentă de prezentare care rulează pe dispozitivul client. Pentru prezentarea conținutului este necesar ca acest layer să interacționeze cu celelalte layere care sunt prezente înaintea acestuia.

Layer-ul aplicației este layer-ul de mijloc al arhitecturii, comunicând atât cu layerul de prezentare, cât și cu cel de date. Acesta cuprinde logica de afaceri (engl. business logic) a aplicației, care reprezintă de fapt un set de reguli necesare ca aplicația să ruleze conform standardelor stabilite de către dezvoltatori. Componentele acestui layer pot rula pe unul sau mai multe servere.

Layer-ul de date este cel mai de jos nivel al aplicației, ocupându-se cu stocarea și accesarea datelor din baza de date. Datele sunt stocate de obicei într-un server de baze de date sau orice dispozitiv care suportă accesul la date și asigură faptul că datele sunt oferite fără a expune acces la mecanismele de stocare și extragere de date. Acest lucru este realizat de nivelul de date prin furnizarea unor funcționalități sau API-uri (engl. application programming interface) care asigură o transparență completă a operațiunilor de date care sunt efectuate în acest nivel, fără a afecta layer-ul aplicației. De exemplu, actualizările la sistemele din acest nivel nu afectează layer-ul aplicației al acestei arhitecturi.

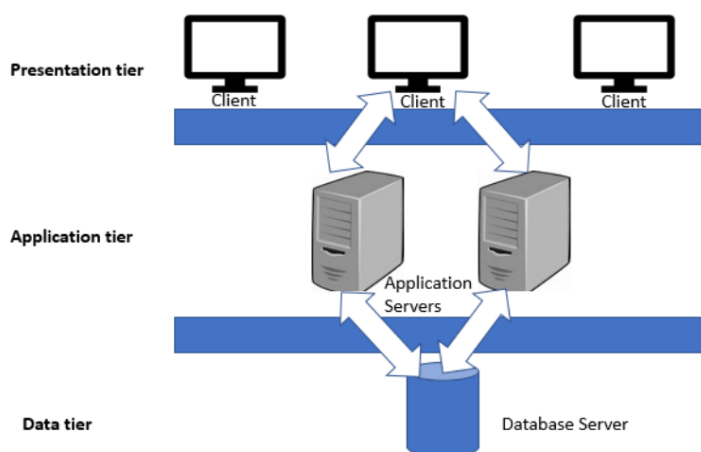


Fig. 2.2.1.1: Arhitectura multistrat [30].

⁶ SOA este un model architectural care permite serviciilor comunicarea pe diferite platforme și limbaje de programare cu scopul de a construi aplicații. În SOA, un serviciu este o unitate software singular destinată îndeplinirii unor cerințe specifice. SOA permite comunicarea serviciilor folosind o cuplare scăzută, acest lucru referindu-se la independența clientului, care la rândul său poate fi un serviciu, de serviciul de care are nevoie [33].

În contextul aplicației layer-ul de prezentare este reprezentat de partea de frontend, formată din aplicația client dezvoltată cu ajutorul librăriei ReactJS. Acesta este o aplicație pe o singură pagină (engl. Single Page Application - SPA⁷) prin intermediul căruia vor fi randate elementele de conținut specifice fiecărei rute pe care o accesează utilizatorul.

Layer-ul aplicației este reprezentat de partea de backend care conține toată logica de business a aplicației, fiind format din mulțimea serviciilor dezvoltate cu ajutorul cadrelor (engl. framework⁸) Spring-Boot și FastAPI, utilizând ca limbaje de programare Java, respectiv Python.

Layer-ul de date este reprezentat de o colecție de API⁹-uri ce constă în mecanisme care oferă accesul la baza de date și prin care se pot face operații aferente, de tipul CRUD (create-read-update-delete), precum și mecanisme ce asigură persistența datelor.

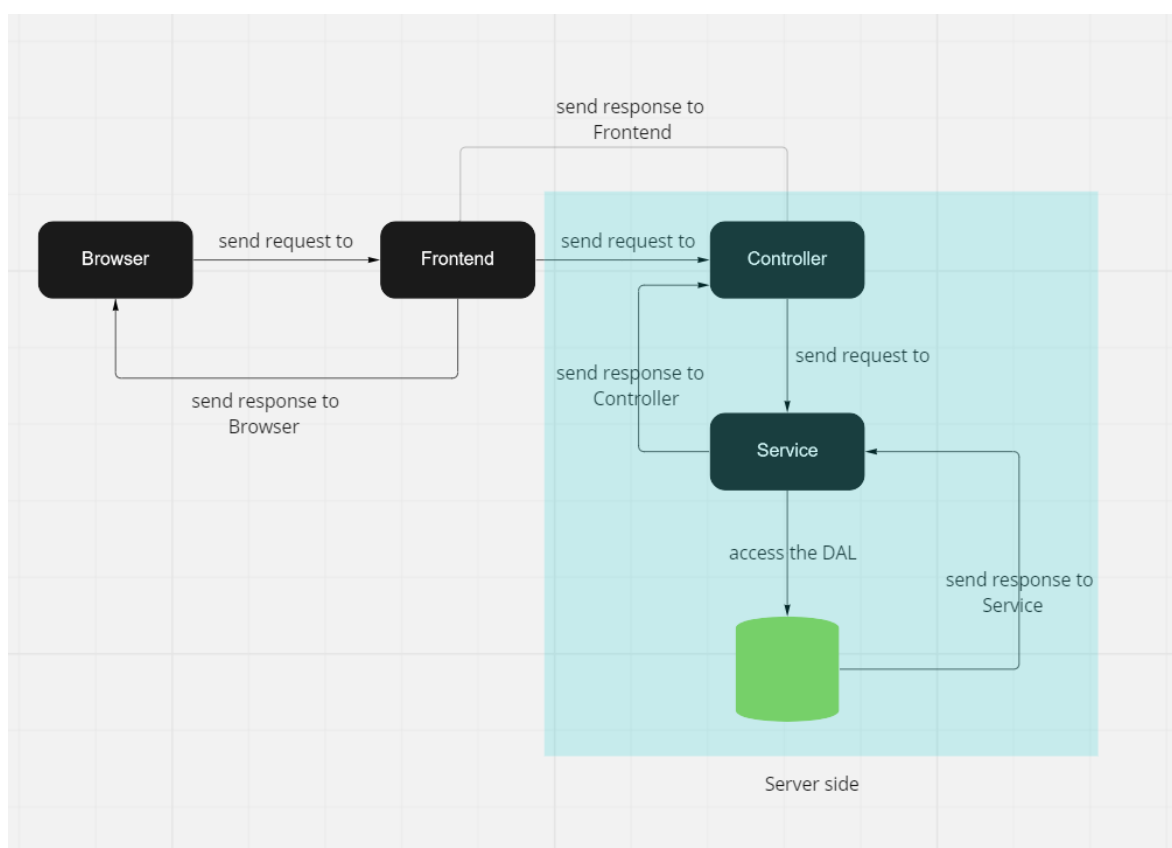


Fig. 2.2.1.2: Arhitectura generală a aplicației

Motivele alegerii arhitecturii bazate pe servicii și nu crearea unui monolit sunt date de diferențele

⁷ SPA (Single Page Application) reprezintă o implementare care încarcă un singur document web. Conținutul acestei pagini este actualizat prin API-uri JavaScript precum XMLHttpRequest, Fetch sau Axios atunci când trebuie afișat conținut diferit [34].

⁸ Framework-ul este o structură ce oferă facilitatea și suportul necesar construirii aplicațiilor software. Acesta servește drept o bază, ușurând munca dezvoltatorilor, astfel încât nu sunt nevoiți să înceapă de la 0. Framework-urile sunt de obicei asociate cu anumite limbaje de programare și variază în funcție de tipul de cerințe [32].

⁹ API (Application Programming Interface) reprezintă un set de definiții de subprograme, protocoale și unelte concretizate în funcționalități destinate dezvoltatorilor de aplicații având ca scop ușurarea muncii acestora [35].

majore ale acestor două tipuri arhitecturale.

Arhitectura monolit presupune implementarea aplicației într-o singură unitate, lucru ce vine cu dezavantaje, dar și cu avantaje, în funcție de tipul aplicației. Acest tip de arhitectură este potrivit aplicațiilor mici, deoarece componentele unui monolit sunt interconectate și interdependente. În asemenea situații sunt evidențiate avantajele monolitului:

- implementarea și integrarea se fac mult mai simplu și rapid
- sunt înregistrate performanțe mai ridicate datorită faptului că în acest tip de aplicații componentele comunică între ele mult mai rapid decât în cazul serviciilor, unde este nevoie de a face request-uri pentru a comunica cu alte servicii.

Arhitectura bazată pe servicii presupune realizarea întregii aplicații din componente independente, slab cuplate, care au ca scop implementarea logicii pentru o cerință specifică. Acest tip de arhitectură este destinat aplicațiilor mari, avantajele fiind garantate:

- reutilizarea serviciilor, care, datorită faptului că sunt slab cuplate pot fi refolosite în diferite aplicații fără a le influența pe celelalte
- mentenanța poate fi asigurată mai ușor și pot fi adăugate noi funcționalități fără a exista conflicte cu celelalte servicii
- fiabilitate ridicată, testarea și depanarea serviciilor fiind mai ușoară decât în cazul aplicațiilor monolit
- paralelism din punct de vedere al dezvoltării, posibilitate oferită de independența serviciilor unele față de altele

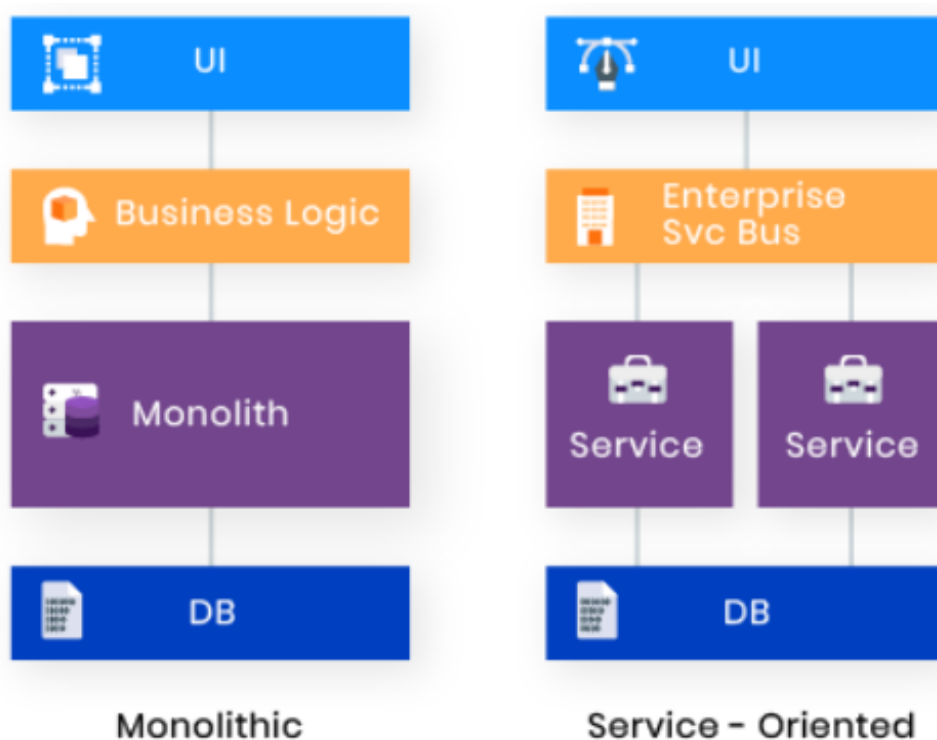


Fig. 2.2.1.3: Arhitectura monolit vs SOA [36].

II.2 Arhitectura bazei de date

Bazele de date folosite pentru stocarea datelor sunt MySQL și MongoDB. Având în vedere că entitățile din baza de date sunt strâns cuplate, între tabele existând relații, s-a ales folosirea unei baze de date relaționale pentru gestiunea acestora. Așadar tabele gestionate în MySQL sunt următoarele:

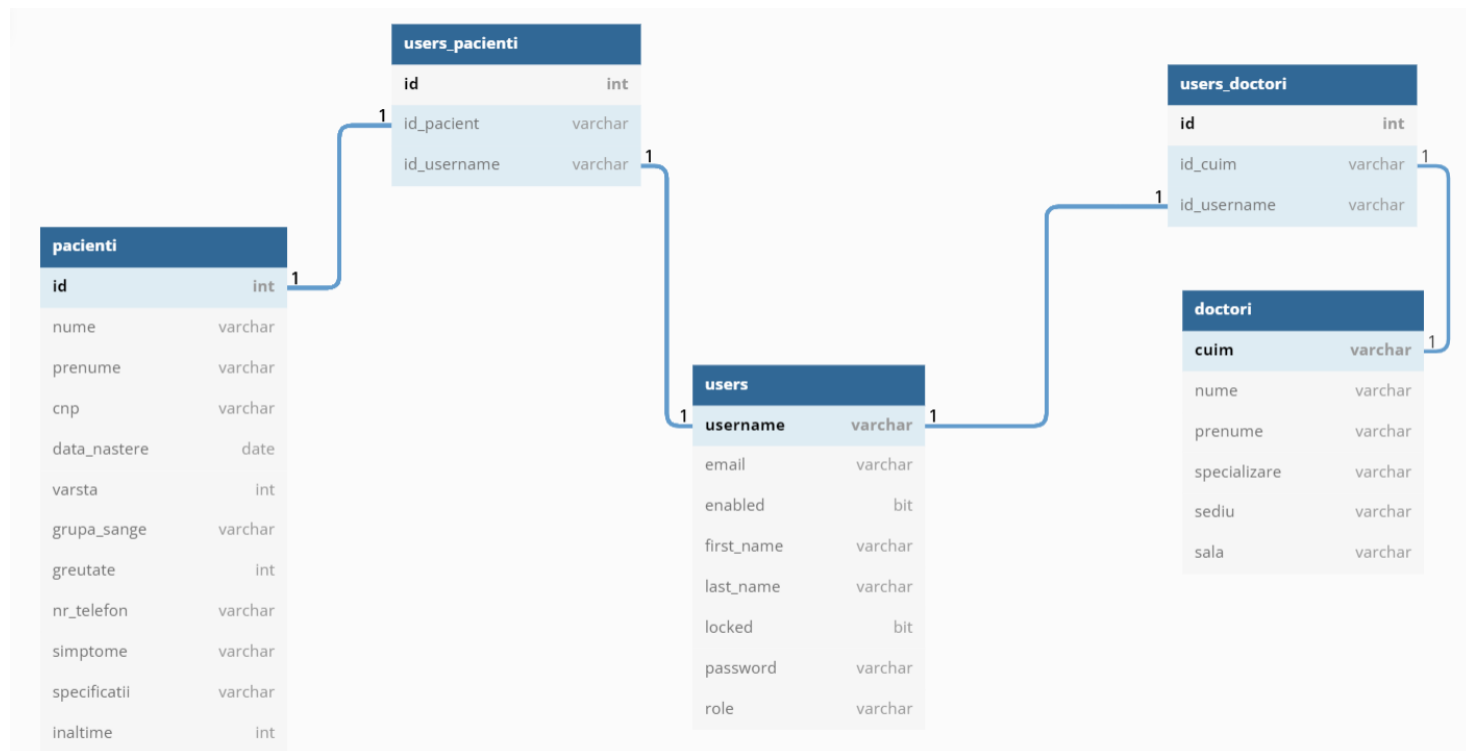


Fig. 2.2.1: Diagrama entitate-relație a tabelor cont-profil

Tabela “**users**” conține informații legate de contul utilizatorilor. Aceștia sunt identificați în mod unic pe baza unui nume de utilizator (engl. username). Datele necesare creării unui cont, pe lângă nume de utilizator, sunt email-ul, numele (engl. lastname), prenumele (engl. firstname), parola, care va fi stocată în baza de date sub formă codificată și rolul unui utilizator, informație ce joacă un rol foarte important în această aplicație. Informațiile care nu sunt obligatorii sunt cele referitoare la valabilitatea contului, enabled (valabil) și locked (blocat). Cele din urmă au rolul de a completa entitatea Users conform implementării UserDetails, fiind necesare la configurarea securității la nivelul logicii de business.

Tabela “**doctori**” conține informațiile referitoare la profilul unui doctor. Ca și cheie primară s-a folosit câmpul **cuim**, acesta reprezentând codul unic de identificare al medicilor. Profilul unui doctor cuprinde strict datele necesare despre acesta din punct de vedere al pacientului: nume, prenume, specializare, sediul și sala.

Tabela “**pacienți**” conține informațiile referitoare la profilul unui pacient, necesare din punct de vedere al unui medic, cum ar fi: cnp, greutate, înălțime, grupă de sânge, simptome, specificații. Aceștia sunt identificați printr-un id autoincrementat.

Tabela “**users _doctori**” este o tabelă de legătură între tabelele **users** și **doctori**, între acestea două stabilindu-se o relație de 1:1, fiecărui profil de doctor corespunzându-i un cont și implicit, fiecărui cont de doctor corespunzându-i un singur profil.

Tabela “**users _pacienti**” este o tabelă de legătură între tabelele **users** și **pacienti**, între care există o relație de 1:1, fiecărui profil de pacient corespunzându-i un cont și implicit, fiecărui cont de pacient corespunzându-i un singur profil.

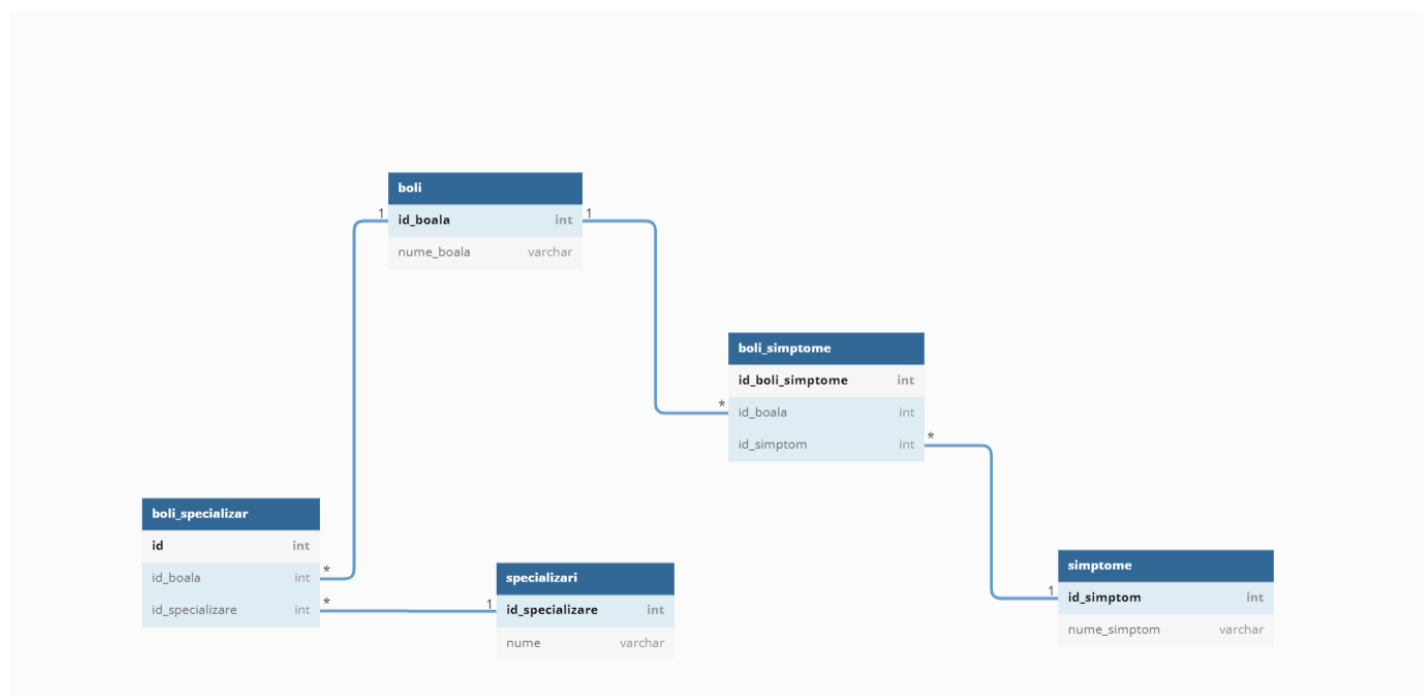


Fig. 2.2.2: Diagrama entitate-relație a tabelor boli-simptome-specializări

Tabela “**boli**” conține numele tuturor bolilor din baza de date. Acestea au fost adăugate în urma cercetării făcute pe acest subiect, fiind extrase din site-uri de specialitate precum Regina Maria și lucrări științifice precum Sinopsis-ul de medicină. Cheia primară a acestei tabele este un id autoincrementat.

Tabela “**simptome**” conține numele tuturor simptomelor din baza de date, care, la fel ca și bolile, au fost adăugate în urma cercetărilor făcute în acest scop. Cheia primară este un id autoincrementat.

Tabela “**boli_simptome**” este tabela de legătură rezultată în urma relației M:N între tabelele **boli** și **simptome**. Așadar, un simptom poate corespunde mai multor boli și o boală poate avea mai multe simptome. Cheia primară este un id autoincrementat.

Tabela “**specializări**” conține numele tuturor specializărilor din baza de date. Acestea au fost extrase în urma cercetării pe site-uri de specialitate. Cheia primară este un id autoincrementat.

Tabela “**boli_specializare**” este tabela de legătură între tabelele **boli** și **specializări**, între acestea existând o relație de M:M, astfel încât o boală poate corespunde mai multor specializări, iar o specializare cuprinde mai multe boli.

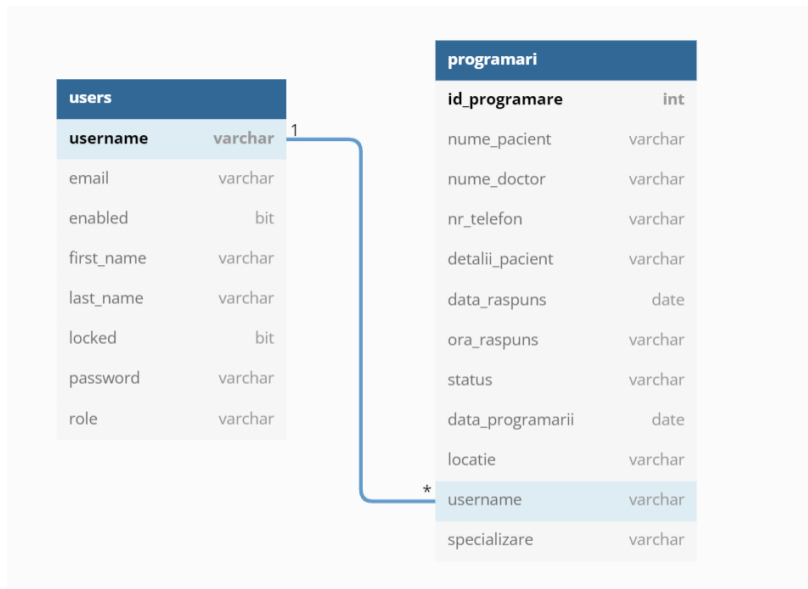


Fig. 2.2.3: Diagrama entitate-relație a tabelelor users-programări

Tabela “**programări**” conține informațiile esențiale creării unei programări, precum numele pacientului, numele doctorului și specializarea, aceste lucruri fiind necesare deoarece pot fi doctori cu același nume la specializări diferite. De asemenea, data programării este importantă deoarece un pacient nu își poate face mai mult de o programare în aceeași zi la același doctor. Aceasta este identificată în mod unic printr-un id autoincrementat, cuprinzând și câmpurile date ca răspuns de un doctor, cum ar fi locația, data de răspuns, aceasta fiind data la care pacientul este programat și ora. Statusul indică stadiul în care se află acest proces. În momentul în care pacientul își face o programare, statusul este cel de “**nou**” (engl. new). După ce doctorul a văzut cererea dar nu a răspuns acesta devine “**văzut**” (engl. seen), iar după ce doctorul a răspuns cererii acesta se transformă în “**acceptat**” (engl. accepted).

Pentru a stoca informațiile legate de analizele încărcate de un pacient s-a folosit MongoDB, acesta fiind mai flexibil în ceea ce privește gestionarea datelor. Colecțiile sunt create per utilizator (engl. user), conținând următoarele informații:

Medical Test
_id: String
username: String
doctorName: String
specialization: String
images: Array<String>
date: Date
status: String
doctorResponse: doctor

Fig. 2.2.4: Structura colecțiilor de analize

Colecțiile au numele utilizatorilor pentru a putea fi gestionate și accesate mai ușor. Acestea conțin câmpul “**images**” care reprezintă o listă de șiruri de caractere, ce sunt de fapt analizele încărcate de către pacienți. Aceste fișiere sunt convertite după ce sunt încărcate din imagini în Base64, un mecanism de codificare în formă binară, destinate stocării datelor sub această formă asupra canalelor care suportă doar date de tip text. Deoarece pacienții pot încărca mai multe fișiere, acestea sunt stocate sub forma unei liste. Data încărcării este necesară pentru evidența pacienților de către medici. Statusul indică stadiul în care se află acest proces, la încărcare fiind cel de “**nou**” (engl. new), în momentul vizualizării fiind “**văzut**” (engl. seen), iar după ce medical a trimis răspunsul, stocat în “doctorResponse”, acesta devenind “**răspuns**” (engl. responded).

II.3 Servicii

Din punct de vedere al serviciilor, aplicația este formată din următoarele servicii: Identity Provider, Profile, Programări, Get Medical Test, Upload Medical Tests și Data Processing, cel din urmă fiind dezvoltat în Python cu ajutorul FastAPI, resul fiind dezvoltate în Java, cu ajutorul Spring-Boot.

Identity Provider este serviciul care are ca scop creare conturilor utilizatorilor aplicației și conectarea la un cont. Acesta conține toată logica securității aplicației: configurarea web, filtrele de autentificare și autorizare, crearea de JWT¹⁰-uri (engl. JSON Web Token), decodificarea acestora și transmiterea răspunsurilor corespunzătoare. Este primul serviciu cu care utilizatorul interacționează, fie logându-se, fie creându-și un cont. Securitatea este făcută folosind Spring Security. Cu ajutorul rolului stocat în JWT este menținută securitatea în restul serviciilor, acesta având o importanță semnificativă în contextul aplicației. Există 3 roluri: admin, doctor și pacient, fiecare având opțiuni diferite în funcție de dreptul acordat rolului său.

Administratorul este cel care are acces asupra controlului bazei de date, fiind singurul cu drept de gestiune a conturilor utilizatorilor. Acesta are posibilitatea de a vedea toți utilizatorii și profilurile lor, atât cele ale doctorilor, cât și ale pacienților, putând chiar să îi și șteargă din baza de date. Administratorul este cel care creează conturile doctorilor, din motive de securitate acest lucru realizându-se la nivelul bazei de date, fără a fi expus la nivel de prezentare. Poate adăuga în baza de date informații medicale, cum ar fi noi boli, specializări medicale sau noi simptome și poate vedea toate aceste informații la nivelul interfeței grafice. De asemenea, administratorul are dreptul de a reseta serviciul care se ocupă cu prelucrarea datelor introduse de utilizatori în clasificarea simptomelor către o anumită specializare medicală.

Doctorul este cel care are cele mai multe drepturi din punct de vedere medical. Acesta poate adăuga în baza de date noi simptome sau boli și poate face legătura între simptomele corespunzătoare unei anumite boli. Din punct de vedere al creării de cont, doctorului nu îi este oferită această posibilitate, administratorul ocupându-se de acest lucru. În momentul conectării la contul acestuia, doctorului îi revine atribuția de a-și completa profilul cu informațiile personale. De asemenea, doctorul are acces la profilurile tuturor pacienților și la informațiile medicale din baza de date, precum și drept de resetare al serviciului care se ocupă cu prelucrarea simptomelor. Doctorul poate vedea analizele încărcate de pacienți doar lor, le pot investiga și pot oferi un răspuns în stabilirea unui diagnostic sau a unui set de medicații pe care pacientul ar trebui să îl urmeze. Din punct de vedere al programărilor, doctorii au dreptul de a răspunde la o programare, ei fiind cei care stabilesc data programării, ora și locația.

¹⁰ JWT (Jason Web Token) este un standard deschis (RFC 7519) care definește o modalitate compactă și automată de transmitere în siguranță a informațiilor între componente software sub forma unui obiect JSON. Aceste informații pot fi verificate și oferă securitate ridicată deoarece sunt semnate digital [36].

Pacienții sunt publicul țintă al acestei aplicații. Au dreptul de a-și crea un cont, de a-și completa profilul cu informațiile necesare, de a face o programare, de a încărca analize medicale către doctorul ales de aceștia, de a-și vedea istoricul medical și istoricul programărilor. De asemenea, au posibilitatea de a vedea toți doctorii în funcție de specializările medicale și toate specializările din sistem.

La nivel de cont, atât doctorilor cât și pacienților le este permis să-și vadă informațiile, să-și schimbe parola sau să-și șteargă contul. Din punct de vedere al profilului, ambelor categorii de utilizatori le este oferită posibilitatea de a și-l actualiza.

Profile este serviciul care are ca scop crearea unui profil atât pentru doctor, cât și pentru pacienți, de actualizarea sau ștergerea acestora, precum și extragerea informațiilor despre un anumit profil. De asemenea în acest serviciu se află logica pentru operațiile pe care userii le pot face, cum ar fi: adăugarea în baza de date de către doctor sau admin de noi boli, simptome sau specializări medicale, de crearea unei legături între simptome și bolile corespunzătoare acestora sau boli și specializările medicale care se ocupă cu tratarea acestor boli, de ștergere sau actualizare precum și de extragere din baza de date a acestor informații medicale. Pentru a menține securitatea aplicației, acest serviciu comunică cu **Identity Provider**, verificările fiind făcute asupra rolului pe care utilizatorul îl are, pentru acest lucru făcându-se requesturi pentru a decodifica JWT-ul.

Programări este serviciul care se ocupă cu gestiunea programărilor făcute de către pacienți. Pentru a crea o programare, pacientul trebuie să specifice doctorul și specializarea, acest lucru fiind necesar din cauza faptului că pot exista doctori cu același nume dar specializări diferite. Prin intermediul acestui serviciu pacienții și doctorii pot vedea programările făcute doar de ei și către ei, iar doctorii pot răspunde programărilor. Doctorul are flexibilitatea de a accepta o programare, specificând data la care este posibilă programarea, ora și locația. Pentru a asigura securitatea și a nu permite pacienților posibilitatea de a vedea informații de acest fel aparținând altor pacienți, serviciul de programări comunică cu serviciul Identity Provider, făcându-se request-uri de tip REST pentru a fi decodificat JWT-ul. Pentru validarea acestui lucru sunt comparate datele legate de numele de utilizator cu care pacientul s-a conectat și cel stocat în JWT.

Upload Medical Test este serviciul care se ocupă cu gestionarea încărcărilor de analize de către pacienți. Pentru a încărca analize medicale, pacientul trebuie să aleagă fișierele dorite și să selecteze doctorul către care vrea să trimită aceste informații. Analizele încărcate trebuie să fie în format de tip imagine, fiind acceptate extensiile JPG, JPEG și PNG. În completarea acestui serviciu vine serviciul **Get Medical Test** care se ocupă cu extragerea informațiilor din baza de date a analizelor încărcate. Aceste două servicii au fost separate din motive de implementare, intrând în conflict dependențele folosite. Pentru a asigura securitatea informațiilor și pentru a invalida posibilitatea pacienților de a vedea istoricul medical al altor pacienți, la fel ca și la celelalte servicii, este făcută verificarea între numele de utilizator cu care pacientul s-a conectat și cel stocat în JWT.

Data Processing este serviciul care are ca scop procesarea simptomelor introduse de către orice utilizator în stabilirea unei specializări medicale spre care aceștia ar trebui să se îndrepte pentru un control. Este un serviciu ce poate fi accesat de oricine, nu este condiționat de a avea cont. Pentru prelucrarea datelor a fost folosit algoritmul de Machine Learning, Random Forest, bazat pe indicii Gini minim.

II.4 Componente

Din punct de vedere al componentelor software, aplicația conține două categorii: componentele cu o arhitectură bazată pe servicii orientate către server și componentele orientate către client. În cadrul componentelor orientate către server sunt incluse serviciile exemplificate și explicate în subcapitolul anterior și o componentă de Machine Learning ce constă în implementarea algoritmului de clasificare Random Forest, ce face parte din serviciul **Data Processing**.

În categoria componentelor software orientate către client este inclusă aplicația dezvoltată în JavaScript, utilizând librăria ReactJS, construită pe baza componentelor funcționale. Accesul la starea unei componente este esențial în comunicarea cu serverul și îndeplinirea scopului aplicației, acela de a oferi utilizatorului informațiile necesare prin intermediul interfeței grafice. Deoarece aceste componente nu oferă mecanisme de gestiune a stărilor lor și nici ale ciclului de viață, s-au folosit hook-uri care sunt funcții prin care se poate accesa starea unei componente, Modul de utilizare al aplicației presupune înregistrarea cererilor venite de la client și transmiterea unui răspuns corespunzător cererii. Acest lucru presupune schimbarea stărilor anumitor componente atât când este făcut un request, cât și atunci când trebuie afișat răspunsul către utilizator. Pentru a gestiona acest lucru s-a folosit hook-ul **useState**, metodă prin care o anumită stare este setată corespunzător, astfel fiind trimise cereri valide către server. Câteva exemple de folosire ale acestei funcții sunt: momentul în care utilizatorul își introduce simptomele care trebuie trimise printr-o cerere la serviciul ce se ocupă cu prelucrarea acestora, aici folosindu-se funcția de stare prin care este setat mesajul cererii cu simptomele.

De asemenea pot exista și situații în care conținutul paginii ar trebui să se modifice în funcție de schimbările apărute la interacțiunea utilizatorului cu interfața grafică, lucru care produce modificări ale stării componente. Pentru aceasta s-a folosit funcția **useEffect** care are efectul de a randa diferite informații în concordanță cu modificările aferente. Un exemplu de folosire al acestei funcții este în contextul afișării formularelor de creare de cont sau de conectare la contul utilizatorului. Atunci când utilizatorul selectează opțiunea de creare de cont, va fi randat pe pagină formularul corespunzător, iar dacă acesta alege opțiunea de logare, pe pagină va fi afișat acest formular.

Posibilitatea folosirii unei structuri arborescente în arhitectura componentelor duce la necesitatea transmiterii unor informații din componenta părinte către componenta copil. Acest lucru presupune existența unui context vizibil acestei structuri prin care componentele copil au acces la informațiile de la părinți, operație posibilă prin folosirea hook-ului **useContext**. Acesta are ca scop crearea unui context vizibil ambelor tipuri de componente, prin care se pot face modificări asupra stării componentelor, în final având ca scop transmiterea de informații corespunzătoare clientului. În completarea scenariului descris în paragraful anterior s-a folosit funcția de context prin care deciziile utilizatorului duc la schimbarea variabilei de context. Acest lucru permite la nivel global accesul la schimbarea stării componente pentru a fi afișat formularul dorit.

Pentru a putea afișa anumite informații fără a reactualiza pagina utilizând **useEffect**, putem folosi hook-ul **useRef** pentru a avea o referință către obiectul din DOM pe care vrem să îl actualizăm. Actualizarea respectivă se face prin setarea conținutului referinței cu informațiile dorite. De exemplu, pentru a căuta anumite informații într-o fereastră de căutare și a primi ca răspuns doar rezultatele dorite, se setează informația ce conține filtrul de căutare folosind funcția menționată. Acest lucru se transpune într-o cerere la server care extrage informațiile aferente, acestea fiind mai apoi afișate clientului.

Navigarea între pagini este una dintre principalele funcționalități de care ar trebui să dispună o aplicație web, menținând-o pe aceasta interactivă și servindu-i drept unul dintre

scopuri. Acest lucru este întâlnit pe tot parcursul interacțiunii dintre client și browser, utilizatorul având posibilitatea de a accesa o altă pagină sau de a reveni la cea anterioară. Pentru furnizarea acestei funcționalități s-a folosit hook-ul **useNavigate**, care permite navigarea între pagini, după cum reiese și din nume. Pentru a folosi această funcție este de ajuns de a o apela, dându-i ca parametru calea către pagina dorită. Exemplele folosirii acestei funcții de navigație în contextul aplicației sunt vaste, câteva dintre ele fiind navigarea directă către pagina de autentificare după ce un client și-a făcut un cont, navigarea către paginile dorite prin selectarea unei opțiuni de care utilizatorul dispune la nivelul interfeței grafice prin apăsarea unor butoane corespunzătoare, sau navigarea către pagina de start atunci când credențialele utilizatorului expiră și acesta trebuie să se conecteze iar.

În strânsă legătură cu **useNavigate** este componenta prefecinită a **react-router-dom**, **BrowserRouter** care permite setarea rutelor și stabilirea conținutului acestora cu informațiile alese de către dezvoltator în funcție de cerințele aplicației. Așadar, navigarea către o pagină validă este posibilă doar dacă aceasta a fost specificată în prealabil în **BrowserRouter**. În cazul în care ea nu există, clientului nu i se afișează niciun conținut.

Un alt hook folosit este **useParams** care facilitează transmiterea de parametri setați în mod dinamic al componentelor URL de la componenta părinte la cea copil. Acesta este de ajutor atunci când avem nevoie de informații specifice despre un anumit topic, în cadrul aplicației fiind folosit pentru a putea trimite numele utilizatorului de la o pagină la alta atunci când este nevoie.

Pentru a face interacțiunea dintre utilizator și aplicație mai dinamică și interesantă s-au folosit și hook-uri ce permit monitorizarea activității clientului. Un exemplu este **useClickOutside**, funcție prin care se gestionează eventul click-ului în afara unei componente. În contextul aplicației acesta este folosit în fereastra de căutare, care atunci când afișează răspunsurile clientului se expandează. În momentul în care utilizatorul apasă click-ul în afara ferestrei, aceasta se strânge și revine la forma inițială.

Capitolul III. Implementarea și utilizarea aplicației

III.1 Interfață și funcționalități

Publicul țintă al acestei aplicații este format din persoanele care-și doresc utilizarea unei platforme ce pune la dispoziție servicii medicale. Acesta poate varia, clienții făcând parte din categorii diferite. Pot fi atât persoane cunoscătoare de tehnologie, cu studii superioare, care nu întâmpină dificultăți în folosirea aplicațiilor web, cât și persoane care nu dispun de aceste cunoștințe, pentru ei interacțiunea cu tehnologia fiind mai neprietenoasă. Așadar, aplicația este proiectată într-un mod simplu și intuitiv, permițând utilizarea acesteia de un public larg, ce nu impune deținerea unor cunoștințe despre domeniul tehnologic.

Pagina de start vine cu o serie de opțiuni pe care le poate accesa orice utilizator.

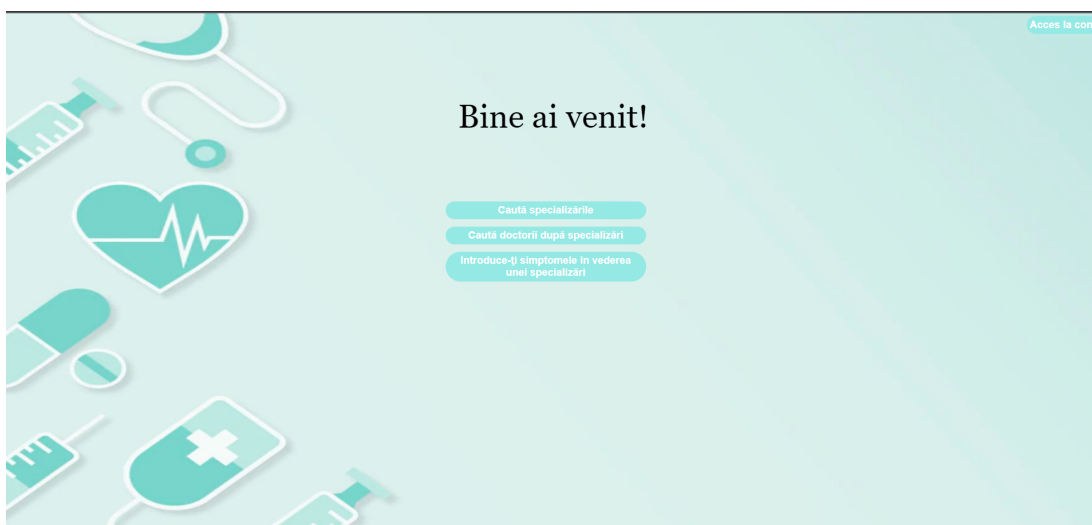


Fig. 3.3.1.1 Pagina de start a aplicației

Astfel, clienții pot vedea specializările medicale de care dispune această platformă pentru a putea decide dacă este în interesul lor accesul la aceasta și pot căuta doctorii în funcție de specializările existente. Un caz foarte des întâlnit la multe persoane este incapacitatea de a ști unde trebuie să apeleze pentru a-și face un control medical. Aplicația vine ca un ajutor în această privință, oferind un serviciu prin care, pe baza simptomelor introduse, oamenii primesc ca recomandare specializarea medicală către care trebuie să se îndrepte. Pentru acest lucru, utilizatorii trebuie să caute în fereastra de căutare simptomele pe care le au, după care trebuie să selecteze opțiunea de a vedea specializarea medicală (Fig. 3.3.1.2).

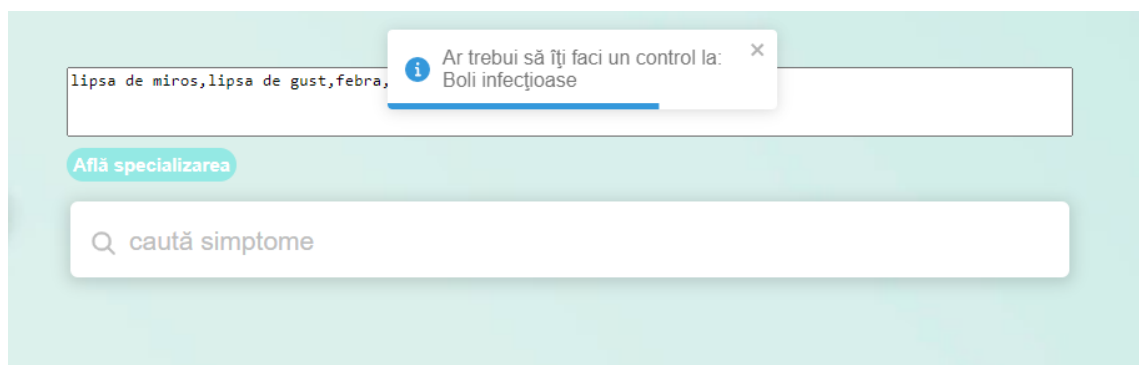
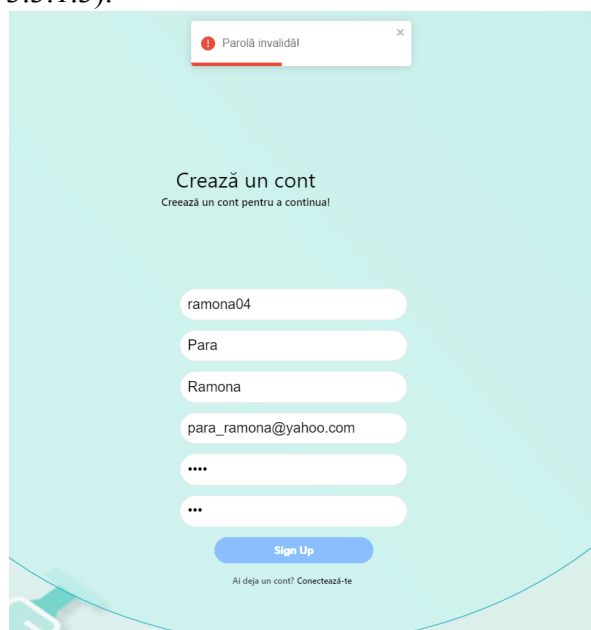


Fig. 3.3.1.2 Funcționalitatea serviciului de procesare a simptomelor

Deoarece este necesară crearea unui cont (Fig. 3.3.1.2) pentru a beneficia de celelalte servicii, utilizatorul are posibilitatea de a-și crea un cont. Pentru accesul la acest formular acesta trebuie să selecteze butonul din partea dreaptă de sus a paginii principale.

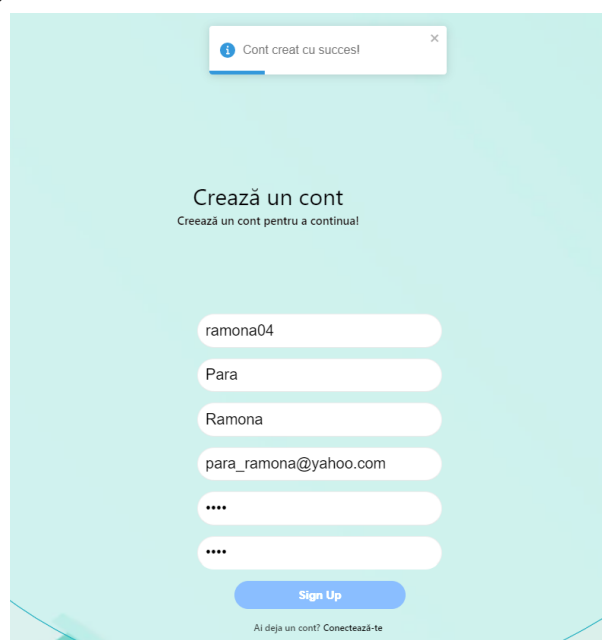
Acesta trebuie să completeze câmpurile formularului, introducând date valide. Trebuie să își introducă o parolă pe care trebuie să o confirme. În cazul în care parolele nu corespund, acesta este atenționat (Fig. 3.3.1.3).



The screenshot shows a web form titled "Crează un cont" (Create an account) with the subtitle "Creează un cont pentru a continua" (Create an account to continue). At the top, a red error message box says "Parolă invalidă!" (Invalid password!). The form contains several input fields: a text field with "ramona04", a text field with "Para", a text field with "Ramona", a text field with "para_ramona@yahoo.com", a password field with four dots, and a confirmation password field with three dots. A blue "Sign Up" button is at the bottom, with a link "Ai deja un cont? Conectează-te" (Already have an account? Log in) below it.

Fig. 3.3.1.3 Formularul de creare de cont introducând parole greșite

Dacă parolele corespund și celelalte câmpuri sunt valide, crearea contului se face cu succes (Fig. 3.3.1.4).



The screenshot shows the same "Crează un cont" form as in Fig. 3.3.1.3, but with a blue success message box at the top that says "Cont creat cu succes!" (Account created successfully!). The input fields and the "Sign Up" button remain the same.

Fig 3.3.1.4 Crearea de cont cu succes.

După ce-și creează contul, utilizatorul este redirecționat la pagina de logare, unde se poate conecta la contul său (Fig. 3.3.1.5)



Fig. 3.3.1.5 Conectarea la contul utilizatorului.

După ce se conectează, clientul trebuie să își completeze profilul, acest lucru fiind obligatoriu pentru a putea continua accesarea serviciilor (Fig 3.3.1.6).

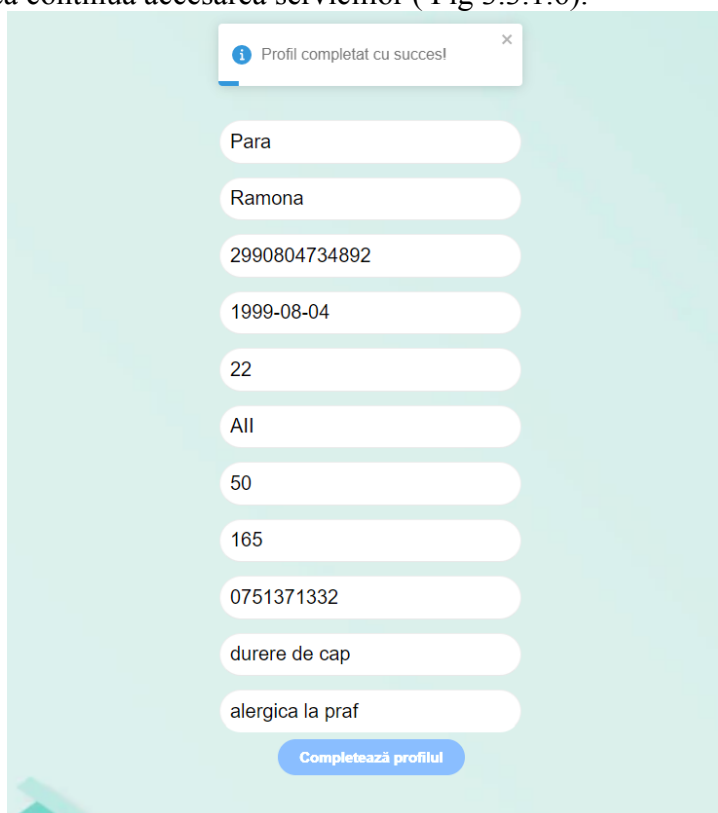


Fig. 3.3.1.6. Completarea profilului pacientului

După completarea profilului, utilizatorului îi sunt afișate opțiunile pe care le are în cadrul platformei (Fig. 3.3.1.7).



Fig. 3.3.1.7 Opțiunile pe care le poate accesa pacientul

Poate căuta doctorii după specializări, lucru ce ușurează procesul de programare (Fig. 3.3.1.8)



Fig. 3.3.1.8 Pagina în care utilizatorul poate căuta doctorii după specializări

De asemenea poate vedea care sunt specializările medicale de care dispune acest sistem (Fig. 3.3.1.9)

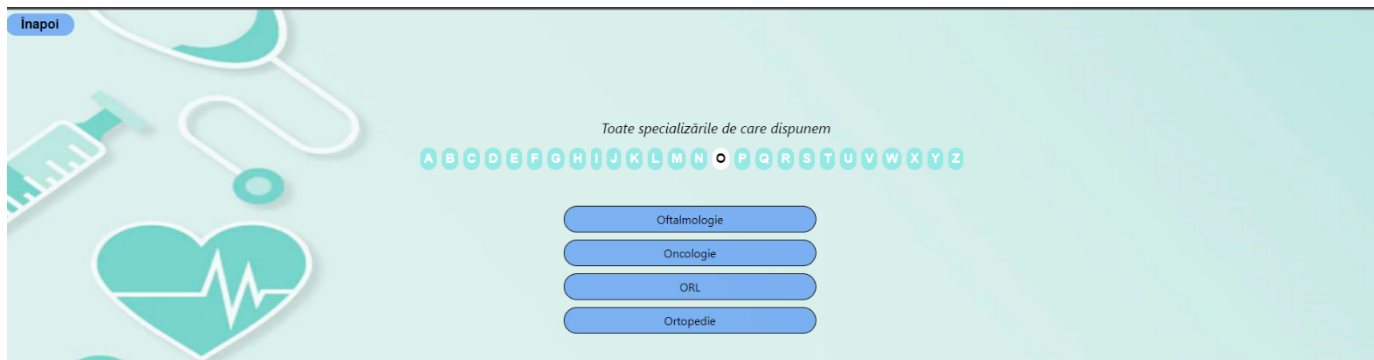
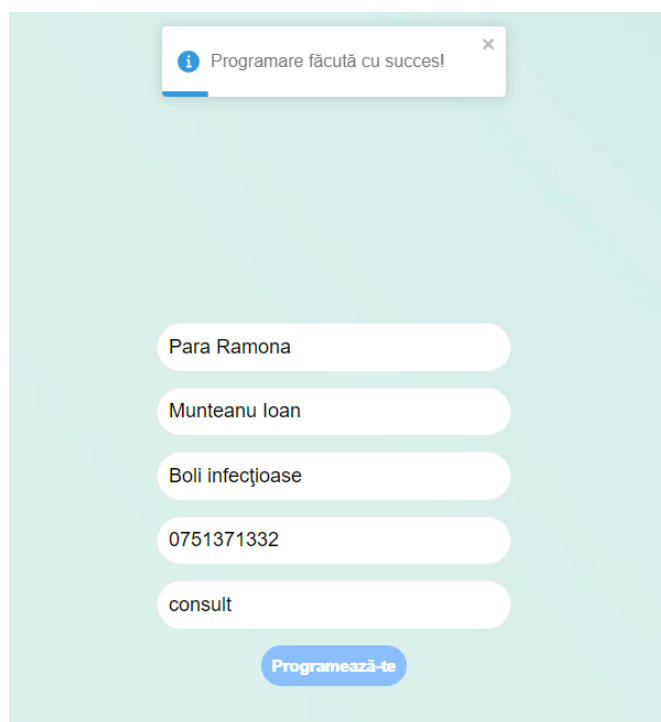


Fig. 3.3.1.9 Pagina în care utilizatorul poate vedea specializările disponibile

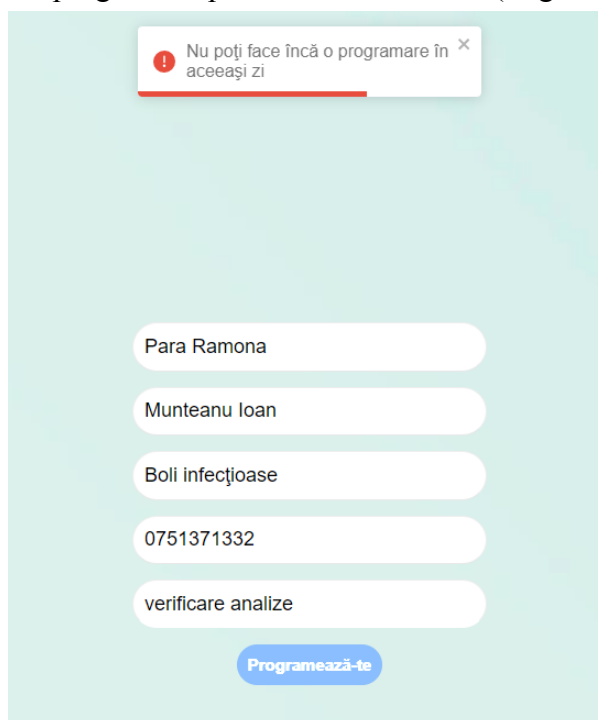
Își poate face o programare, selectând opțiunea aferentă. Aici, trebuie să completeze câmpurile formularului de programare și să trimită cererea de programare (Fig. 3.3.1.10).



A screenshot of a web form for booking an appointment. At the top, there is a success message in a white box with a blue border: "Programare făcută cu succes!" with an information icon on the left and a close icon on the right. Below this, the form contains five white input fields with rounded ends, each containing text: "Para Ramona", "Munteanu Ioan", "Boli infecțioase", "0751371332", and "consult". At the bottom of the form is a blue button with white text that says "Programează-te". The entire form is set against a light teal background.

Fig. 3.3.1.10 Crearea unei programări

Pentru a asigura o limitare asupra numărului de cereri de programare, un pacient nu își poate face mai mult de o programare pe zi la același doctor (Fig. 3.3.1.11)



A screenshot of the same appointment booking form as in Fig. 3.3.1.10, but with an error message. At the top, there is a red error message box: "Nu poți face încă o programare în aceeași zi" with a red exclamation mark icon on the left and a close icon on the right. The input fields below contain the same text as before: "Para Ramona", "Munteanu Ioan", "Boli infecțioase", "0751371332", and "verificare analize". The blue "Programează-te" button is still at the bottom. The background is light teal.

Fig. 3.3.1.11 Crearea unei programări în aceeași zi la același doctor

Este făcută o evidență asupra programărilor, astfel încât pacientul are posibilitatea să vadă toate programările făcute, precum și răspunsul doctorilor la acestea. În cazul în care doctorul nu a răspuns programării, clientului îi este afișat un mesaj corespunzător (Fig 3.3.1.12).



Fig. 3.3.1.12 Vizualizarea programărilor pe care un pacient și le-a făcut

O altă funcționalitate pusă la dispoziția utilizatorilor, care eficientizează și facilitează interacțiunea în contextul medical dintre aceștia și doctori este cea legată de transmiterea unui set de analize în vederea investigării lui, pentru a obține un diagnostic și un set de medicații necesar de urmat ca tratament. Astfel, clientul își poate trimite analizele către medicul dorit prin selectarea acestuia din fereastra de căutare. Acesta este atenționat printr-un mesaj în care i se specifică alegerea făcută pentru a evita momentele de neatenție și a face o selecție corectă. În cazul în care utilizatorul întâmpină această situație, poate cu ușurință să selecteze alt medic. În ajutorul acestui pas este pagina anterioară de căutare a doctorilor după specializări, astfel încât pacienții să știe în ce direcție se îndreaptă alegerea lor. Urmează încărcarea analizelor dorite care trebuie să fie fișiere în format JPG sau PNG. Numărul de imagini poate varia, nefiind restricționat de o singură astfel de instanță. Pentru trimiterea cererii către doctor, pacientul trebuie să apese butonul corespunzător, moment în care primește o notificare pe ecran cu rezultatul transmiterii cererii. Pentru a nu lăsa să fie încărcate din greșală mesaje goale, în cazul în care clientul vrea să trimită cererea fără a încărca nimic este atenționat de acest lucru iar cererea nu se trimite către doctor (Fig. 3.3.1.13).

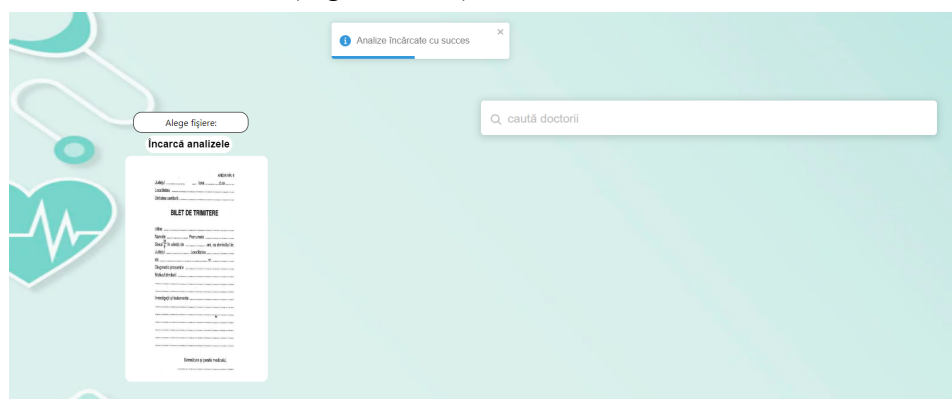


Fig. 3.3.1.13 Încărcarea analizelor

De asemenea, ca și la programări, este ținută evidența și în cazul analizelor, pacientul având posibilitatea să-și vadă istoricul medical. În cazul în care doctorul nu a dat niciun răspuns analizelor încărcate, pacientului îi este afișat un mesaj din care reiese acest lucru. De asemenea, clientul poate ști stadiul în care se află procesul de prelucrare al analizelor de către medic prin culoarea pe care o deține căsuța prin care are acces la vizualizarea analizelor, lucru valabil și la programări. Dacă este verde înseamnă că cererea este nouă și nu a fost văzută de doctor. Dacă este galbenă înseamnă că doctorul a văzut cererea și nu a răspuns, iar când acesta oferă un răspuns este afișat un fundal roșu (Fig. 3.3.1.14)

Fig. 3.3.1.14 Vizualizarea analizelor pe care un pacient le-a încărcat

În continuare, pacientul are ca opțiuni să-și vadă profilul și să modifice informațiile dorite prin selectarea opțiunii de a actualiza profilul. Dacă acesta dorește să actualizeze profilul dar nu a modificat nicio informație este înștiințat printr-un mesaj corespunzător. De asemenea, poate să-și vadă detaliile contului dar nu poate să modifice informațiile introduse în momentul creării contului. Poate să-și schimbe în schimb parola și poate să-și șteargă contul (Fig. 3.3.1.15).

Fig. 3.3.1.15 Schimbarea parolei

La nivel de informații personale, doctorul are aceleași opțiuni ca și pacientul din punct de vedere al profilului și al contului. Poate vedea profilurile tuturor pacienților și are opțiunea de a reseta serviciul care se ocupă cu funcționalitatea de prelucrare a simptomelor. De asemenea, fiind un cadru specializat în domeniul medical, doctorul poate adăuga în baza de date noi simptome sau boli și poate face legătura între acestea.

Poate vedea analizele medicale încărcate de pacienți și poate oferi un răspuns acestora. Inițial îi este afișată o listă cu numele tuturor pacienților care au trimis astfel de documente (Fig. 3.3.1.16) iar pentru a vedea analizele specifice fiecăruia trebuie să selecteze persoana dorită (Fig. 3.3.1.17)

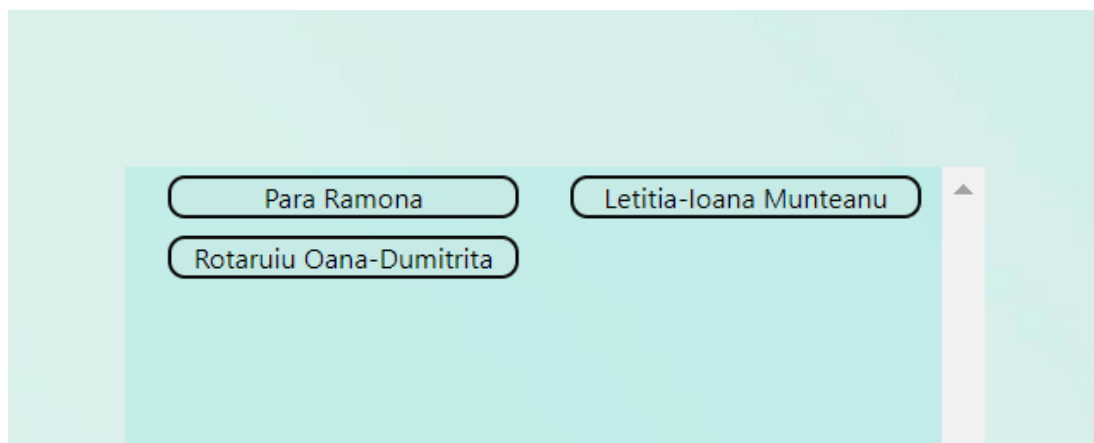


Fig 3.3.1.16 Lista cu pacienții care au încărcat analize la un anumit medic

Fig. 3.3.1.17 Vizualizarea propriu-zisă a analizelor

Doctorul își poate stabili după propriul plac orarul programărilor, știindu-și programul și valabilitatea (Fig. 3.3.1.18)

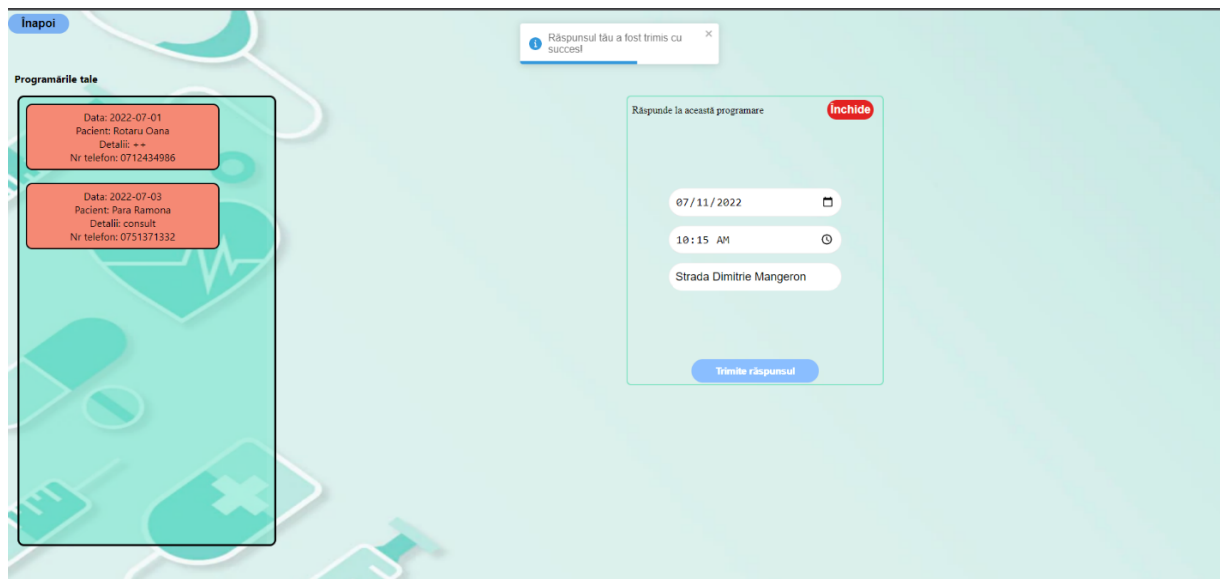


Fig. 3.3.1.18 Răspunderea la programările făcute

Pentru administrator există următoarele opțiuni: aflarea informațiilor despre doctori și pacienți, adăugarea informațiilor medicale și vizualizarea lor și resetarea serviciului de prelucrare a simptomelor. Acesta poate vedea toți doctorii existenți cu ajutorul unei ferestre de căutare. Poate accesa opțiunea de a le vedea profilul, și, având drepturi nelimitate, îi poate șterge din baza de date în situații corespunzătoare precum retragerea unui doctor din sistemul respectiv. La fel, și pentru pacienți sunt posibile aceleași opțiuni, de a-i căuta în baza de date, de a le vedea profilul și de a-l șterge atunci când este necesar acest lucru. Fiind persoana care are dreptul gestiunii bazei de date, administratorul poate adăuga informații medicale simple, precum noi boli, simptome sau specializări. Totuși, nefiind un cadru medical autorizat, acesta nu are dreptul de a face legătura dintre simptome și boli pentru a evita comiterea unor erori grave din punct de vedere medical. Această atribuție îi revine doctorului care are studiile și cunoștințele necesare realizării acestui proces. Pentru o vizualizare de ansamblu și pentru ușurarea muncii ca administrator, acesta poate vedea și la nivel de interfață informațiile medicale existente în baza de date. Are același drept ca și doctorul în resetarea serviciului care se ocupă cu prelucrarea simptomelor, această opțiune neavând efecte negative.

III.2 Implementarea aplicației

Aplicația este compusă din două mari module: partea de server și partea de client.

Partea de server are o arhitectură bazată pe servicii, întrucât este o aplicație ce necesită modularizare din cauza volumului de logică de business, fiecare serviciu având cerințe specifice. Totodată este necesar ca interacțiunea dintre acestea să fie una slab cuplată, întrucât unele servicii sunt independente de altele. Astfel, funcționalitatea unui serviciu nu ar trebui să fie influențată de cea a altui serviciu și nu ar trebui să fie afectată de eventualele erori sau situații de limită apărute la nivelul acestuia.

Aceasta este compusă din serviciile prezentate în capitolul II, subcapitolul III.3. Primul serviciu din această categorie este **Identity Provider**, cel care conține toată logica necesară securității aplicației, creării de cont și logării. De asemenea, în cadrul acestui serviciu s-a implementat și funcționalitatea de creare de JWT pe baza credențialelor clientului, acesta fiind

folosit mai apoi pentru asigurarea securității și în celelalte servicii care comunică cu acesta.

Crearea de JWT (Fig.3.3.2.1) este făcută folosind algoritmul HMAC (engl. hash-based message authentication code) care este un tip specific de codificare a mesajelor de autentificare folosind o cheie secretă publică. Acesta este folosit în verificarea integrității datelor și autenticitatea mesajelor. JWT-ul este format din 3 părți:

- **header**-ul ce conține algoritmul folosit în crearea JWT-ului
- **payload**-ul, format din datele clienților necesare din punct de vedere al securității: numele de utilizator și rolul său
- **semnătura** ce conține cheia cu care a fost semnat algoritmul

```
private final static Algorithm algorithm = Algorithm.HMAC256("secretkey".getBytes());

public static String accessTokenCreator(String username, String issuer, String role) {
    return JWT.create()
        .withSubject(username)
        .withExpiresAt(new Date(System.currentTimeMillis() + 120 * 60 * 1000))
        .withIssuer(issuer)
        .withClaim("role", role)
        .sign(algorithm);
}
```

Fig. 3.3.2.1: Funcția de creare de JWT

Decodificarea JWT-ului este făcută folosind același algoritm și aceeași cheie ca și la crearea lui. Acesta va oferi informațiile necesare validării identității utilizatorilor: numele de utilizator, rolul și valabilitatea acestuia: expirat sau invalid (Anexa 1).

Autentificarea folosită este una din cel mai des utilizate moduri de autentificare, cea prin nume de utilizator și parolă. Așadar, filtrul de autentificare ales este **UsernamePasswordAuthenticationFilter**. Desfășurarea (engl. flow) procesului de autentificare decurge în următorul mod: utilizatorul încearcă să se conecteze la contul său folosind credențialele necesare, nume de utilizator și parolă, proces a cărui implementare se află în funcția **attemptAuthentication** (Fig. 3.3.2.2) din clasa ce conține logica filtrului de autentificare, **CustomAuthenticationFilter**.

```
@Override
public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)
    throws AuthenticationException {

    String username = request.getParameter("username");
    String password = request.getParameter("password");
    Log.info("Username is: {} and password is: {}", username, password);
    UsernamePasswordAuthenticationToken authenticationToken = new UsernamePasswordAuthenticationToken(username,
                                                                                                     password);
    return authenticationManager.authenticate(authenticationToken);
}
```

Fig. 3.3.2.2 Funcția attemptAuthentication

După ce se încearcă autentificarea, se apelează funcția **loadUserByUsername** (Fig. 3.3.2.3), prin care este interogată baza de date pentru a stabili dacă un utilizator există și are dreptul să se autentifice cu credențialele introduse.

```
@Override
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

    System.out.println("load user by username\n");
    Users user = userRepository.findByUsername(username);
    SimpleGrantedAuthority role = null;

    if(user != null){
        log.info("User {} found in database", username);
        role = new SimpleGrantedAuthority(user.getRole());
        System.out.println("role from user service " + user.getRole());
    }
    else{
        log.error("User {} not found in database", username);
        throw new UserExceptionNotFound(username);
    }

    return new org.springframework.security.core.userdetails.User(user.getUsername(), user.getPassword(),
                                                                    Collections.singleton(role));
}
```

Fig. 3.3.2.3 Funcția loadUserByUsername

În cazul în care validările sunt îndeplinite este returnat un obiect de tip **UserDetails** care conține credențialele clientului și o listă cu rolul pe care îl are, necesar pentru permiterea accesului la anumite resurse. Este apelată mai apoi funcția **successfulAuthentication** (Anexa 2) unde sunt create două token-uri, unul de acces și unul de reîmprospătare (eng. refresh), acesta fiind răspunsul cererii de logare. Cel de acces este folosit pentru a stabili durata de timp în decursul căreia utilizatorul are dreptul de a trimite cereri în aplicație fără a fi nevoit să se conecteze la fiecare request, valabilitatea lui fiind de 2 ore. Token-ul de refresh are scopul de a stabili un interval de timp în care pot fi trimise cereri de recreare a token-urilor de acces după ce acestea au expirat, fără ca utilizatorul să fie nevoit să se reconecteze, acesta având o valabilitate de 4 ore. Așadar, după ce token-ul de acces a expirat, este făcută o cerere de creare a unui nou astfel de token, cât timp refresh token-ul este valid. După ce a expirat și token-ul de refresh, utilizatorul este deconectat, iar pentru a putea continua acesta trebuie să se autentifice din nou. Acestea sunt necesare din punct de vedere al securității, menținând permanența validării identității unui utilizator.

Autorizarea folosită este una de tipul **OncePerRequestFilter** (Anexa 3), care, după cum reiese și din numele acesteia, este un proces ce se execută o dată pentru fiecare request. Aici se verifică dacă autentificarea a fost făcută, se decodifică token-ul de acces și dacă acesta este expirat sau invalid este trimis ca răspuns faptul că utilizatorul nu are dreptul de a mai face request-uri. În caz contrar, este setat în contextul de securitate general utilizatorul autentificat, conținând și lista de drepturi pe care le are, lucru ce va permite sau nu ca acesta să facă anumite cereri.

Configurarea web a securității (Anexa 4) are ca scop stabilirea accesului către diferite rute în funcție de rolul pe care îl are utilizatorul. De asemenea, există și rute accesibile oricărui utilizator, cum ar fi cea de logare sau de crearea unui cont.

Ca și funcționalități existente în cadrul acestui serviciu sunt menționate următoarele:

- căutarea unui utilizator după email
- căutarea unui utilizator după nume sau prenume
- actualizarea contului unui utilizator
- ștergerea contului unui utilizator
- verificarea parolei unui utilizator folosind **BCryptPasswordEncoder**

Următorul serviciu este cel care se ocupă cu crearea profilului pacienților și doctorilor, **Profile**. În acest serviciu sunt încapsulate și operațiile pe care utilizatorii le pot face. Printre funcționalitățile acestui serviciu se enumeră:

- crearea profilului de pacient/doctor
- căutarea unui pacient după id, prenume, nume, CNP
- actualizarea profilului unui pacient/doctor
- ștergerea profilului unui pacient/doctor
- căutarea unui doctor după nume, prenume, cod unic de identificare, specializare
- legarea profilului unui pacient/doctor de contul său
- adăugarea/ștergerea unei noi specializări în/din baza de date
- căutarea unei specializări după nume
- adăugarea/ștergerea unui simptom în/din baza de date
- căutarea unei boli după nume
- adăugarea unei boli în baza de date
- legarea unui simptom de o boală
- legarea unei boli de o specializare

Serviciul **Programări** are ca scop crearea unei programări pe care doar pacientul poate să o facă și extragerea informațiilor aferente din baza de date în funcție de rol. Astfel, pacientul își poate vedea doar propriile programări, iar doctorul poate vedea doar programările făcute la acesta. În implementarea **funcției de creare** a unei programări sunt verificate informațiile utilizatorului care a trimis cererea, în sensul că, acesta nu poate face mai mult de o programare în aceeași zi la același doctor. Pentru acest lucru, se parcurge lista ce conține programările clientului, iar dacă mai există o programare care conține aceeași dată și același nume de doctor de la aceeași specializare, este aruncată o excepție prin care utilizatorul este înștiințat că nu poate face mai mult de o programare în aceeași zi în contextul menționat. Deoarece toate informațiile unei programări sunt stocate în aceeași tabelă în baza de date, **funcția de actualizare**, este folosită în momentul în care doctorul trimite răspunsul către o programare. Clientul nu are posibilitatea să-și modifice programarea făcută, această funcție servind drept funcția prin care doctorul îi oferă un răspuns pacientului. În continuare, urmează **funcțiile de extragere a informațiilor** programărilor făcute. Pentru **funcția care îi întoarce pacientului** programările sale este făcută o interogare în baza de date în care sunt căutate programările după numele de utilizator. Pentru a asigura confidențialitatea datelor, este verificat dacă username-ul cu care s-a logat pacientul este același cu cel stocat în baza de date. Pentru funcția care îi întoarce doctorului programările trimise către el este făcută o interogare în baza de date în după numele doctorului și specializarea sa. Pentru a menține confidențialitatea datelor este făcută o verificare și asupra numelui de utilizator prin care se validează dacă numele cu care s-a logat este același cu cel din

baza de date, în caz contrar aruncându-se o excepție prin care se specifică că cel care încearcă să acceseze datele nu are autoritate asupra acestui lucru.

Serviciile care permit gestionarea analizelor sunt **UploadMedicalTest** și **GetMedicalTest**. În primul serviciu s-a făcut implementarea **încărcării analizelor** medicale, stocate sub forma unor documente ale colecțiilor formate după numele utilizatorului care a făcut cererea. Documentele încărcate trebuie să fie în format JPG sau PNG. Pentru asigurarea securității este făcut un request către serviciul Identity Provider. În cazul în care JWT-ul pacientului nu este expirat și nici invalid, se setează colecția pe numele utilizatorului iar mai apoi se salvează documentul creat (Anexa 5). Funcționalitățile celui de-al doilea serviciu menționat presupun 4 metode: una prin care **pacientul își poate vedea analizele încărcate**, implementarea acestuia constând în verificarea username-ului pentru a asigura confidențialitatea și returnarea documentelor incluse în colecția sa. Pentru necesitățile doctorului sunt implementate **funcțiile care returnează toți utilizatorii care și-au încărcat analize la el și funcția care returnează setul de analize pentru fiecare utilizator**. Funcția de **update** este folosită atunci când doctorul trimite un răspuns pacientului, un document fiind format atât din ce informații trimite pacientul, cât și din informațiile trimise de către doctor.

Ultimul serviciu aflat în categoria componentelor de backend este **Data Processig**, cel care clasifică simptomele introduse de către utilizatori pentru obținerea unei specializări medicale corespunzătoare tratării posibilelor afecțiuni. În cadrul acestuia s-a realizat implementarea algoritmului Random Forest bazat pe indicele Gini minim (Anexa 6).

Pentru a calcula indicele Gini minim se parcurg următorii pași: sunt luate toate atributele din setul de date, acestea fiind reprezentate de simptomele existente și valorile acestora. În cazul de față, valorile atributelor sunt da sau nu, reprezentând faptul că acel simptom există sau nu. Pentru fiecare valoare a atributului se generează un subset de date care conține doar valoare specificată a acestuia. Este calculată apoi lungimea acestui subset, reprezentând numărul de astfel de apariții, după care este calculată probabilitatea ca fiind raportul dintre lungimea menționată anterior și numărul total de instanțe ale setului întreg de date. Urmează calcularea apartenenței la o clasă din subsetul de date, în acest caz clasa fiind de fapt boala la care conduce existența simptomului respectiv. Formula de calcul pentru aceasta este:

$$gini = 1 - \left(instances * \frac{1}{instances} \right)^2$$

Această valoare reprezintă numărul de instanțe din subsetul de date care aparțin unei anumite clase, aici boală. Deoarece în setul de date nu există nicio boală formată din aceleași simptome, s-a ales simplificarea formulei și folosirea celei de mai sus. La final, valoarea mai sus calculată este înmulțită cu probabilitatea calculată anterior, fiind valoarea finală a indexului Gini pentru atributul ales cu valoarea dată. Indexul Gini minim este ales ca fiind cel cu valoarea cea mai mică din toate valorile calculate pentru acest subset, deoarece o valoare mai mică a acestuia duce la o acuratețe mai ridicată.

Urmează alcătuirea arborilor care compun pădurea de arbori, fiind formați din noduri ce conțin de fapt simptomele care conduc la o anumită boală (Anexa 7). În cazul în care toate instanțele din setul de date aparțin aceleiași clase și implicit indicele Gini va fi nul, se alege primul element al acestei liste ca și nod. Dacă lista atributelor este goală, ca și nod va fi ales acela cu clasa care apare cel mai frecvent. În cazul în care niciuna dintre aceste situații nu se întâmplă, este calculat indicele gini minim conform celor de mai sus, din care va rezulta atributul cu cea mai mică valoare a acestei metrici. Pentru fiecare valoare a acestui atribut, este generat un

set de date care conține doar valoare specificată a atributului, în acest caz setul de date care conține simptomul respectiv ca atribut și setul de date care nu îl conține. Dacă nu există astfel de subseturi, nodul va fi format din secvența de date care conține clasa cel mai frecvent întâlnită. Altfel, va fi creată o listă ce conține valoarea specificată a atributului în cauză și nodurile vor fi formate recursive din secvențele rezultate.

Pădurea de arbori (Anexa 8) este formată din arborii creați după explicațiile de mai sus de data aceasta dintr-un subset de date generat aleator din întregul set de date și o listă de atribute generate în aceeași manieră. Astfel, copacii ce alcătuiesc pădurea sunt formați din date diferite, lucru ce mărește acuratețea predicțiilor.

Fiecare arbore va avea o valoare proprie a predicției, în cazul de față a bolii ce rezultă dintr-o combinație de simptome. Predicția finală va fi dată de numărul maxim al apariției în mulțimea arborilor (Anexa 9), aceasta fiind reprezentată de o boală. Pentru a specifica specializarea medicală corespunzătoare bolii rezultate, se va face o interogare în baza de date, iar utilizatorul va primi ca răspuns specializarea la care este legată boala rezultată.

Setul de date pe care s-au făcut testele este făcut manual, adăugând simptome corespunzătoare anumitor boli în urma cercetării făcute în această direcție pe site-uri de specialitate și cu ajutorul unei persoane specializate în domeniul medical. Deoarece baza de date pentru acestea nu este una vastă este posibil ca predicția oferită de acest algoritm să nu fie una reală. De aceea, pentru primirea unui răspuns cât mai precis, trebuie introduse de către utilizator cât mai multe simptome. Acuratețea ar putea crește cu extinderea bazei de date.

Partea de client este formată din aplicația dezvoltată cu ajutorul librăriei ReactJS, utilizând ca limbaj de programare JavaScript. Aceasta este concretizată în interfața grafică prin care utilizatorul comunică cu serverul, trimițând cereri și primind răspunsuri corespunzătoare cererilor. Structura interfeței este formată din paginile menționate în anexa 10.

Una dintre implementările care trebuie menționate este cea a ferestrei de căutare (engl. search bar) [39]. Aceasta are o logică complexă ce presupune trimiterea cererii către server, filtrând informațiile în concordanță cu textul introdus de client. SearchBar-ul este reprezentat de o componentă care are anumite proprietăți și care include un câmp în care sunt introduse informațiile pe baza cărora ar trebui să se facă filtrările. Cererea este trimisă către server în momentul în care este detectată modificarea conținutului, așadar pentru fiecare caracter s-ar trimite o nouă cerere, lucru ce nu este dorit deoarece ar produce scăderea performanțelor la nivel de server și o latență ridicată. Pentru a acoperi astfel de situații, s-a implementat mecanismul numit **debounce**. Acesta este un hook prin care se realizează monitorizarea tastării, făcându-se un callback prin care este setat un timp de așteptare. În momentul în care clientul tastează, timpul este setat pe 0 urmând ca din momentul în care textul introdus nu mai este schimbat și implicit clientul nu mai tastează să se aștepte câteva secunde, iar abia mai apoi să se trimită. Atunci când cererea este procesată și s-a primit un răspuns de la server, informațiile primite sunt afișate în interiorul SearchBar-ului, moment în care acesta se expandează. La detectarea apăsării clickului în afara acestuia, fereastra revine la forma inițială. Acest lucru se transpune în cod prin mărirea și micșorarea unor proprietăți CSS height (înălțime), schimbate în funcție de situația prezentată. Componenta de SearchBar conține la rândul său o altă componentă prin care este afișată singular informația primită. Astfel, pentru o cerere în care avem ca răspuns 10 date, sunt afișate tot atâtea subcomponente. Prin intermediul acestor subcomponente este posibilă selectare și mai apoi primirea de informații suplimentare ale rezultatelor căutării. Această funcționalitate a fost folosită pentru căutarea doctorilor, pacienților și simptomelor.

Un alt lucru care merită menționat este implementarea formularelor de creare de cont și logare [40]. Acestea sunt incluse într-o componentă ce are forma unei elipse. Există posibilitatea

de a selecta opțiunea formularului dorit, moment în care elipsa își schimbă forma și este încărcată opțiunea clientului. Acest lucru s-a realizat prin următoarele funcții, prin care se setează opțiunea dorită și se realizează o animație în momentul selectării (Fig. 3.3.2.4):

```
const playExpandingAnimation = () =>{
  setExpanded(true);
  setTimeout(() => {
    setExpanded(false)
  }, expandingTransitions.duration * 1000 - 1500);
};

const switchToSignUp = () => {
  playExpandingAnimation();
  setTimeout(() => {
    setActive("signup");
  }, 400)
}

const switchToSignIn = () => {
  playExpandingAnimation();
  setTimeout(() => {
    setActive("signin");
  }, 400)
}
```

Fig. 3.3.2.4 Implementarea funcțiilor de setare a opțiunii și animație

O altă funcționalitate care face parte din categoria acelor mai deosebite este cea prin în interfața grafică este ilustrată actualizarea statusului analizelor sau programărilor încărcate, pe baza unui cod de culori. După cum s-a menționat anterior, dacă cererea este nouă și nu este văzută de doctor apare cu fundal verde. Dacă în schimb doctorul a văzut-o ea este galbenă, iar dacă doctorul a răspuns la cerere, este roșie. Acest lucru s-a realizat prin implementarea unei funcții care face parte din containerul în care sunt afișate răspunsurile cererilor, prin care se returnează o culoare diferită în funcție de statusul primit de la server, avut aici ca parametru (Fig. 3.3.2.5):

```
background-color: ${props => {
  if(props.status === "new")
    return '■ #2ac819';
  else if(props.status === "seen")
    return '■ #e5d52d';
  else if(props.status === "accepted")
    return '■ #f58974';
}};
```

Fig. 3.3.2.5 Funcția ce setează culoarea în conformitate cu statusul

Pentru a schimba starea statusului, în momentul în care doctorul face click pe o analiză sau o programare, este trimisă o cerere către server prin care se actualizează statusul informației corespunzătoare. Acest status este modificat iar în momentul în care doctorul trimite răspunsul către client cu succes.

Un rol important în scopul acestei aplicații este funcționalitatea care permite încărcarea fișierelor ce reprezintă analizele pacienților. Această implementare a constituit una dintre dificultățile întâlnite pe parcursul dezvoltării aplicației, întrucât acesta necesită folosirea promise-urilor¹¹, acest lucru având un comportament nedeterminat la nivelul interfeței grafice deoarece încărcarea de imagini se face asincron prin intermediul File Reader. Implementarea soluției constă în folosirea unui await¹² prin care se așteaptă mai întâi încărcarea tuturor imaginilor și apoi afișarea lor. Aceasta a fost preluată de pe site-ul de specialitate Stackoverflow¹³.

Alte funcționalități implementate la nivelul aplicației client, care sunt doar menționate:

- actualizarea informațiilor printr-o cerere de tip update
- ștergerea informațiilor printr-o cerere de tip delete
- extragerea informațiilor printr-o cerere de tip get
- crearea de noi entități printr-o cerere de tip post

¹¹ Promise-urile sunt obiecte folosite în operații care se execută asincron și reprezintă eventuala finalizare sau eșec precum și valoarea rezultată acesteia [37].

¹² Await-ul este un operator folosit în contextul unei operații asincrone având ca scop așteptarea setării unui obiect de tip promise [38].

¹³ <https://stackoverflow.com/questions/13975031/reading-multiple-files-with-javascript-filereader-api-one-at-a-time>

Concluzii

Tema aleasă oferă posibilitatea folosirii unei platforme ce pune la dispoziție servicii medicale ușurând accesul la acestea și celor care nu dispun de resursele necesare, vizita la medic fiind transpusă în mediul virtual prin funcționalitatea încărcării analizelor de către pacienți.

Implementarea aplicației, ce înglobează diferite tehnologii de actualitate, este transpusă într-o serie de servicii ce facilitează interacțiunea pacienților cu medicii și vine în ajutorul oamenilor cu funcționalități care nu sunt oferite de către alte astfel de platforme.

Prin observarea dificultății în ceea ce privește alegerea unei specializări medicale compatibile cu nevoile persoanelor, un prim punct forte al acestei aplicații este oferirea serviciului prin care aceștia primesc o recomandare care s-ar plia pe cerințele lor.

Un al doilea punct forte al aplicației este posibilitatea accesării acestui serviciu de către un public larg, nefiind nevoiți să fie înregistrați în sistemul medical pentru acest lucru.

Oferirea posibilității de a crea un cont și de a-l utiliza în vederea folosirii serviciilor medicale ulterioare se enumeră în această listă. De asemenea, prin serviciul de care dispun pacienții pentru a-și încărca analizele, nefiind astfel nevoiți să se deplaseze la medic, lucru care nu este oferit de alte platforme, aplicația ia un plus valoare.

Programarea la medic și posibilitatea doctorului de a răspunde în conformitate cu valabilitatea programului său este un alt lucru de menționat în lista avantajelor întâlnite în cadrul aplicației.

În cele din urmă, oferirea tuturor acestor servicii într-o formă simplă, ușor de folosit de către un public variat din punct de vedere al cunoștințelor tehnologice, face ca aplicația să fie una de actualitate, care cuprinde satisfacerea nevoilor dintr-o arie largă a persoanelor ce au nevoie de utilizarea serviciilor medicale.

Se poate observa că aplicația nu dispune de posibilitatea schimbării parolei în cazul în care un client și-a uitat-o. Acest lucru poate fi trecut în lista cu îmbunătățirile ulterioare care pot fi aduse aplicației.

În ceea ce privește partea de Machine Learning, o îmbunătățire semnificativă ar fi extinderea bazei de date cu mult mai multe boli și simptome pentru mărirea acurateței și oferirea unui răspuns mult mai exact. Tot la nivelul acestui topic, o dezvoltare ulterioară ar putea fi și folosirea altor algoritmi de clasificare în vederea comparării rezultatelor obținute și formarea unei statistici în acest sens.

Din punct de vedere al aplicației client ar putea fi aduse îmbunătățiri în ceea ce privește aranjarea componentelor ce compun paginile interfeței utilizator.

Bibliografie

- [1]. <https://docs.oracle.com/javaee/6/firstcup/doc/gkhoy.html>, ultima vizitare martie, 2022
- [2]. <https://www.ibm.com/cloud/learn/java-spring-boot>, ultima vizitare martie, 2022
- [3]. <https://spring.io/projects/spring-boot>, ultima vizitare martie, 2022
- [4]. https://www.youtube.com/watch?v=zvLZLFWrbAA&t=503s&ab_channel=SpringDeveloper, ultima vizitare martie, 2022
- [5]. <https://docs.spring.io/spring-framework/docs/current/reference/html/overview.html>, ultima vizitare martie, 2022
- [6]. <https://medium.com/java-epic/which-web-server-is-used-by-spring-boot-4e024106923f>, ultima vizitare martie, 2022
- [7]. <https://docs.spring.io/spring-boot/docs/2.0.9.RELEASE/reference/html/howto-embedded-web-servers.html>, ultima vizitare aprilie, 2022
- [8]. <https://reactjs.org/>, ultima vizitare martie, 2022
- [12]. <https://spring.io/projects/spring-security>, ultima vizitare iunie, 2022
- [13]. <https://docs.spring.io/spring-security/site/docs/3.0.x/reference/security-filter-chain.html>, ultima vizitare iunie, 2022
- [14]. <https://docs.spring.io/spring-security/site/docs/3.0.x/apidocs/org/springframework/security/core/Authentication.html>, ultima vizitare iunie, 2022
- [15]. <https://docs.spring.io/spring-security/reference/servlet/authentication/architecture.html#servlet-authentication-authentication>, ultima vizitare iunie, 2022
- [16]. <https://medium.com/@haytambenayed/how-does-spring-security-work-internally-525d359d7af>, ultima vizitare iunie, 2022
- [17]. <https://docs.spring.io/spring-security/reference/servlet/authentication/passwords/form.html>, ultima vizitare iunie, 2022
- [18]. <https://fastapi.tiangolo.com/>, ultima vizitare iunie, 2022
- [19]. <https://levelup.gitconnected.com/types-of-react-components-a38ce18e35ab>, ultima vizitare iunie, 2022
- [20]. <https://reactrouter.com/docs/en/v6/hooks/use-params>, ultima vizitare iunie, 2022
- [21]. <https://www.freecodecamp.org/news/react-hooks-fundamentals/>, ultima vizitare iunie, 2022
- [22]. <https://www.javatpoint.com/react-hooks>, ultima vizitare iunie, 2022
- [23]. <https://styled-components.com/docs/basics>, ultima vizitare iunie, 2022
- [24]. https://www.researchgate.net/publication/259235118_Random_Forests_and_Decision_Trees, ultima vizitare iunie, 2022
- [25]. <https://www.section.io/engineering-education/introduction-to-random-forest-in-machine-learning/>, ultima vizitare iunie, 2022
- [26]. <https://towardsdatascience.com/entropy-how-decision-trees-make-decisions-2946b9c18c8>, ultima vizitare iunie, 2022
- [27]. <https://quantdare.com/decision-trees-gini-vs-entropy/>, ultima vizitare iunie, 2022
- [28]. <https://www.techtarget.com/searchoracle/definition/MySQL>, ultima vizitare iunie, 2022
- [29]. <https://www.guru99.com/what-is-mongodb.html>, ultima vizitare iunie, 2022
- [30]. <https://hub.packtpub.com/what-is-multi-layered-software-architecture/>, ultima vizitare iunie, 2022
- [31]. <https://www.reactenlightenment.com/react-jsx/5.1.html>, ultima vizitare iulie, 2022
- [32]. <https://www.codecademy.com/resources/blog/what-is-a-framework/>, ultima vizitare iulie, 2022

- [33]. <https://www.techtarget.com/searchapparchitecture/definition/service-oriented-architecture-SOA> ultima vizitare iulie, 2022
- [34]. <https://developer.mozilla.org/en-US/docs/Glossary/SPA> ultima vizitare iulie, 2022
- [35]. https://support.shutterstock.com/s/article/what-is-an-api?language=en_US&pl=PPC_GOO_BK_IG_DSA-591767523279&cr=c&kw=&c3apid=71700000062580390&gclid=CjwKCAjw2f-VBhAsEiwAO4lNeGe8igPt34pi214LFxI2u-AXl6lk2571pJmcxPjsQ4ORchlrF4M-pxoC-BUQAvD_BwE, ultima vizitare iulie, 2022
- [36]. <https://medium.com/@JalelTounsi/monolith-soa-microservices-or-serverless-43dd60e29756>, ultima vizitare iulie, 2022
- [36]. <https://jwt.io/introduction>, ultima vizitare iulie, 2022
- [37]. https://developer.mozilla.org/enUS/docs/Web/JavaScript/Reference/Global_Objects/Promise, ultima vizitare iulie, 2022
- [38]. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await>, ultima vizitare iulie, 2022
- [39]. https://www.youtube.com/watch?v=llnmWntmUns&ab_channel=CoderOne ultima vizitare aprilie, 2022
- [40]. https://www.youtube.com/watch?v=-bll7l-BKQI&ab_channel=CoderOne

Anexe

Anexa 1: Codul pentru decodificarea JWT-ului

```
public class JwtDecoder {

    private final static String SECRET_KEY = "secretkey";
    private final static Algorithm algorithm = Algorithm.HMAC256(SECRET_KEY.getBytes());

    public static String jwtDecoder(String token) {
        try {
            String JwtToken = token.substring("Bearer ".length());
            JWTVerifier verifier = JWT.require(algorithm).build();
            DecodedJWT decodedJWT = verifier.verify(JwtToken);
            String role = decodedJWT.getClaim("role").toString().replace("\\", "");
            return decodedJWT.getSubject() + " " + role;
        }
        catch (Exception e) {
            if (e.getMessage().contains("expired")) {
                return "Expired";
            }
            else {
                return "Invalid";
            }
        }
    }
}
```

Anexa 2: Codul pentru funcția successfulAuthentication

```
@Override
protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse
response, FilterChain chain, Authentication authentication) throws IOException, ServletException {

    User user = (User) authentication.getPrincipal();
    String accessToken = accessTokenCreator (user.getUsername(),
                                             request.getRequestURL().toString(),
                                             user.getAuthorities().stream().map(GrantedAuthority::getAuthority)
                                             .collect(Collectors.toList()).get(0));
    String refreshToken = refreshTokenCreator(user.getUsername(),
                                             request.getRequestURL().toString());

    Map<String, String> tokens = new HashMap<>();
    tokens.put("accessToken", accessToken);
    tokens.put("refreshToken", refreshToken);
    response.setContentType(APPLICATION_JSON_VALUE);
    new ObjectMapper().writeValue(response.getOutputStream(), tokens);
}
```

Anexa 3: Implementarea filtrului de autorizare

```
public class CustomAuthorizationFilter extends OncePerRequestFilter{

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain
    filterChain) throws ServletException, IOException {

        if(request.getServletPath().equals("/api/login") ||
            request.getServletPath().equals("/api/userController/refreshToken")) {
            filterChain.doFilter(request,response);
        }
        else {
            String authorizationHeader = request.getHeader(AUTHORIZATION);

            if(authorizationHeader != null && authorizationHeader.startsWith("Bearer ")){
                String decodedJWT = jwtDecoder(authorizationHeader);

                if(decodedJWT.equals("Expired") || decodedJWT.equals("Invalid")){

                    response.setStatus(FORBIDDEN.value());
                    response.setContentType(APPLICATION_JSON_VALUE);
                    ResponsePojo res = new ResponsePojo(FORBIDDEN.value(),decodedJWT);
                    new ObjectMapper().writeValue(response.getOutputStream(), res); //res
                    return;
                }

                String username = decodedJWT.split(" ")[0];
                String role = decodedJWT.split(" ")[1];
                SimpleGrantedAuthority userRole = new SimpleGrantedAuthority(role);
                UsernamePasswordAuthenticationToken authenticationToken = new
                UsernamePasswordAuthenticationToken(username, null, Collections.singleton(userRole));
                SecurityContextHolder.getContext().setAuthentication(authenticationToken);
                filterChain.doFilter(request,response);
            }
            else{
                filterChain.doFilter(request,response);
            }
        }
    }
}
```

Anexa 4: Codul pentru configurarea web

```
@Override
protected void configure(HttpSecurity http) throws Exception {

    CustomAuthenticationFilter customAuthenticationFilter = new
        CustomAuthenticationFilter(authenticationManagerBean());
    customAuthenticationFilter.setFilterProcessesUrl("/api/login");
    http.cors().and().csrf().disable();

    http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    http.authorizeRequests().antMatchers(POST, "/api/createAccount/register", "/api/login").permitAll();
}
```

```

http.authorizeRequests().anyRequest().authenticated();
http.addFilter(customAuthenticationFilter);
http.addFilterBefore(new CustomAuthorizationFilter(), UsernamePasswordAuthenticationFilter.class);
}

```

Anexa 5: Codul pentru încărcarea analizelor

```

public MedicalTest addMedicalAnalysis(ClientRequestPojo requestPojo, String jwt) throws
IOException {

```

```

    MedicalTest uploadedAnalysis = new MedicalTest();
    RestTemplate restTemplate = new RestTemplate();

```

```

    try{
        String url = "http://localhost:8889/api/jwt/decodeJwt";
        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_JSON);
        headers.setBearerAuth(jwt.substring(7));
        HttpEntity<String> request = new HttpEntity<>("", headers);

```

```

        ResponseEntity<String> response = restTemplate.exchange(url, HttpMethod.GET,
request, String.class);
        String username = Objects.requireNonNull(response.getBody()).split(" ")[0];

```

```

        if(username.equals(requestPojo.getUsername())) {
            clientRepository.setCollectionName(requestPojo.getUsername());

```

```

            uploadedAnalysis.setDoctorName(requestPojo.getDoctorName());
            uploadedAnalysis.setSpecialization(requestPojo.getSpecialization());
            uploadedAnalysis.setStatus("new");
            uploadedAnalysis.setImages(requestPojo.getImages());
            uploadedAnalysis.setDate(LocalDate.now());
            uploadedAnalysis.setUsername(requestPojo.getUsername());

```

```

            clientRepository.save(uploadedAnalysis);

```

```

        }
        else{
            throw new GlobalException("Validare a identitatii falsa!");
        }

```

```

    }
    catch (Exception ex){
        if(ex.getMessage().startsWith("403")){
            if(ex.getMessage().contains("Invalid")){
                throw new JwtException("invalid");
            }
            else if(ex.getMessage().contains("Expired")){
                throw new JwtException("expired");
            }
        }
    }
}

```

```
}  
  
return uploadedAnalysis;  
}
```

Anexa 6: Codul pentru calcularea funcției MinGini

```
def MinGini(dataprocess, attributeList):  
    minGini = 2.0  
    minAttribute = "  
  
    for attribute in attributeList:  
        attributeValues = dataprocess.getAttribValues(attribute)  
        giniValue = 0.0  
  
        for attributeValue in attributeValues:  
            attributeValueSubset = dataprocess.getSubset(attribute, attributeValue)  
            instances = len(attributeValueSubset)  
            probability = instances / dataprocess.instanceCount  
            giniV = 1 - (instances * (1 / instances) ** 2)  
            giniValue += probability * giniV  
  
        if giniValue < minGini:  
            minGini = giniValue  
            minAttribute = attribute  
  
    return minAttribute
```

Anexa 7: Codul pentru funcția BuildTree

```
def BuildTree(ds, branchName, attributeList):  
    node = Node()  
    node.branchName = branchName  
    data_process = DatasetProcessor(ds)  
  
    if data_process.getGini() == 0:  
        node.name = list(ds[data_process.className])[0]  
        return node  
  
    if len(attributeList) == 0:  
        node.name = data_process.getLabelWithMaxCount()  
        return node  
  
    A = MinGini(data_process, attributeList)  
    node.name = A  
  
    AttributeValues = data_process.getAttribValues(A)
```

```

for value in AttributeValues:
    subset = data_process.getSubset(A, value)

    if len(subset) == 0:
        node.children.append(data_process.getLabelWithMaxCount())
    else:
        newAttribList = []
        for attribute in attributeList:
            if attribute != A:
                newAttribList.append(attribute)

        node.children.append(BuildTree(subset, value, newAttribList))

return node

```

Anexa 8: Codul pentru funcția RandomForest

```

def RandomForest(trainingDataset):
    roots = []
    training_set = DatasetProcessor(trainingDataset)
    depth = 15

    print(trainingDataset)

    for i in range(depth):
        random_subset = training_set.generateRandomSubset()
        random_symptoms = training_set.generateRandomAttribList()
        root = BuildTree(random_subset, "", random_symptoms)
        roots.append(root)
    return roots

```

Anexa 9: Codul pentru funcția Predict

```

def Predict(dataset, roots):
    diseases = []

    for j in range(len(roots)):
        aux = None
        node_copy = roots[j]
        maxiter = 0
        while node_copy.children and maxiter <= 500:
            for child in node_copy.children:
                if child.branchName == dataset[node_copy.name][0]:
                    aux = child
                    pass
            maxiter += 1
            node_copy = aux
        if node_copy:
            diseases.append(node_copy.name)

```

```

frequency = pd.value_counts(np.array(diseases))
print(frequency.index[0:3])
return frequency.index[0]

```

Anexa 10: Paginile aplicației client

```

export default function App () {

return(
  <Router>

    <Routes>
      <Route exact path = '/' element = {<HomePage/>} />
      <Route exact path = '/login' element = {<LoginPage/>} />

      <Route exact path = '/specializations' element = {<Specializations/>} />
      <Route exact path = '/diseases' element = {<Diseases/>} />
      <Route exact path = '/symptomsData' element = {<SymptomsDataProcess/>} />
      <Route exact path = '/symptoms' element = {<Symptoms/>} />
      <Route exact path = '/searchDoctors' element = {<SearchDoctors/>} />

      <Route exact path = '/completePatientProfile/:username' element =
{<PatientProfileForm/>} />
      <Route exact path = '/homePatient/:username' element =
{<HomePagePatient/>} />
      <Route exact path = '/makeAnAppointment' element =
{<MakeAnAppointment/>} />
      <Route exact path = '/seeYourAppointments' element =
{<SeeYourAppointments/>} />
      <Route exact path = '/uploadAnalysis' element = {<UploadAnalysis/>} />
      <Route exact path = '/seeYourAnalysis' element = {<SeeYourAnalysis/>} />
      <Route exact path = '/seePatientProfile/:username' element =
{<SeePatientProfile/>} />
      <Route exact path = '/accountSettings/:username' element =
{<AccountPatientSettings/>} />
      <Route exact path = '/seeAccountDetails/:username' element =
{<SeeAccountDetails/>} />
      <Route exact path = '/changePassword/:username' element =
{<ChangePassword/>} />

      <Route exact path = '/homeDoctor/:username' element =
{<HomePageDoctor/>} />
      <Route exact path = '/seeDoctorProfile/:username' element =
{<SeeDoctorProfile/>} />
      <Route exact path = '/patientsInfo' element = {<PatientsComponent/>} />
      <Route exact path = '/updatePatients/:id' element = {<UpdatePatientsForm/>} />

```



```

        <Route exact path = '/seeMedicalAnalysis' element = {<SeeMedicalAnalysis/>}/>
        <Route exact path = '/seeClientAnalysis/:clientName' element =
{<SeeClientAnalysis/>}/>
        <Route exact path = '/doctorAppointments' element =
{<SeeDoctorAppointments/>}/>
        <Route exact path = '/drMedicalManagement' element =
{<DoctorMedicalManagement/>}/>
        <Route exact path = '/completeDoctorProfile/:username' element =
{<DoctorProfileForm/>}/>

```

```

        <Route exact path = '/adminPage' element = {<AdminPage/>}/>
        <Route exact path = '/doctorsInfo' element = {<DoctorsComponent/>}/>
        <Route exact path = '/updateDoctors/:id' element = {<UpdateDoctorsForm/>}/>
        <Route exact path = '/adminAddmedicalInfo' element =
{<AdminAddMedicalInformation/>}/>
        <Route exact path = '/medicalInfo' element = {<MedicalInformation/>}/>

```

```

        </Routes>
    </Router>
);
}

```