UNIVERSITATEA BABEŞ-BOLYAI

Facultatea de Ştiinţe Economice şi Gestiunea Afacerilor

Specializarea Informatică Economică

Lucrare de licență

Absolvent,

Anamaria **RAITA**

Coordonator ştiinţific,

Prof. univ. dr. Gheorghe-Cosmin **SILAGHI**

2022

UNIVERSITATEA BABEŞ-BOLYAI

Facultatea de Ştiinţe Economice şi Gestiunea Afacerilor

Specializarea Informatică Economică

Lucrare de licență

Website pentru gestionarea procesul de donare a sângelui în centrele de transfuzii cu backend în Java, frontend în React și bază de date MySQL

Absolvent,

Anamaria **RAITA**

Coordonator științific,

Prof. univ. dr. Gheorghe-Cosmin **SILAGHI**

2022

Rezumat

Lucrarea abordează gestionarea întregului flux de date și procese care au loc în cadrul unui proces de donare de sânge într-un centru de transfuzii, prin automatizarea și digitalizarea acestui act, utilizând un site web.

Pentru gestionarea datelor, am utilizat în backend tehnologia Spring și limbajul Java, deoarece oferă facilități pentru securitate integrate și un mod facil de a accesa o bază de date în MySQL. Pentru a interacționa cu utilizatorii, interfața aplicației va fi realizată în React și va face legătura cu partea de backend, ocupându-se și de partea de validare de date.

Diferența website-ului pe care îl propun față de ceea ce mai există este că îmbină partea de creare a programărilor și de interacțiune cu donatorii cu procesele pe care le au de făcut doctorii, și anume introducerea rezultatelor, confirmarea programărilor și introducerea unor eventuale apeluri la donare pentru diferiți pacienți.

Consider că în special pentru a încuraja actul de donare, cât și pentru a ține pasul cu digitalizarea, acest site web pe care îl propun îmbină toate facilitățile de care este nevoie pentru a gestiona acest proces, iar în cazul unei eventuale dorințe de dezvoltare, aceasta se poate realiza cu ușurință.

Cuprins

Figuri:

Introducere

Motivația alegerii

Tema abordată în această lucrare a fost motivată la bază de experiența mea personală, alături de alți factori din lumea înconjurătoare pe care îi voi detalia în cele ce urmează.

Un om major poate să doneze sânge o dată la aproximativ trei luni, mai exact 72 de zile. O donare poate salva aproximativ trei vieți. Așadar, un om eligibil pentru donare ar putea salva într-un an aproximativ nouă vieți. Aflând acestea, eu m-am hotărât să fiu unul dintre acei oameni, iar din anul 2018 am început să fiu un donator regulat.

Mergând destul de des la donare, m-am lovit de câteva lucruri pe care mi-aș dori să le schimb cu aplicația pe care o voi propune în această lucrare.

În primul rând, aplicațiile care există pentru a-ți face programare nu sunt în legătură cu site-ul web la Centrelor de Transfuzii, iar ca să vezi un istoric al programărilor și analizelor ai nevoie și de alt cod de analize pe lângă CNP, iar dacă nu ai fost destul de inspirat să îl notezi cât timp ai fost la ele.

De asemenea, nu există o interacțiune între oamenii care au nevoie de sânge și donatori, mai specific nu poți afla de pe aplicație dacă există cineva în stare gravă care are nevoie de sânge.

Iar aplicația Donorium pe care eu am folosit-o nu oferă nicio dovadă care să ateste că ți-ai făcut programare, iar astfel nu te poți bucura de avantajele pe care acest lucru ar trebui să ți le aducă.

Aplicația pe care o propun va trimite un e-mail la fiecare programare adăugată și la fiecare programare ștearsă.

Așadar, am ales să abordez această temă deoarece m-am lovit chiar eu de neajunsurile aplicațiilor care se presupune că ar trebui să automatizeze procesele. Mai mult decât atât, lumea nu este încurajată și motivată suficient să doneze sânge, iar eu consider că existența unui proces automatizat bine și plăcut pentru utilizator, fie ei donatori sau doctori, ar putea să popularizeze acest fapt și să încurajeze lumea să vină în centrele de transfuzii pentru a ajuta oamenii.

Existența unei astfel de aplicații consider că va îmbunătății implicarea civică în acest proces de donare a sângelui, va aduce beneficii și pentru doctori și implicit spitale, deoarece se automatizează acest proces, însă va fi un factor important și pentru cei care au nevoie de sânge și nu au unde să își facă nevoia auzită.

Alegerea metodologiei

Ca metodologie de lucru am oscilat între două abordări diferite, și anume din Modelul Agile, deoarece principiile susținute de această metodologiile (consultate http://agilemanifesto.org/, 2022) se potrivesc cu modul în care am ales eu să abordez lucrarea, mai specific, am început să dezvolt două funcționalități de bază pe care ulterior le-am dezvoltat după ce am aflat părerea publicului prin intermediul unui chestionar. Mai specific, am ales metodele Extreme Programming și Code and Fix, probabil folosind câte puțin din ambele. Am ales aceste două metode din mai multe motive. Principalul motiv a fost flexibilitatea și faptul că deși aveam idee în mare cam ce funcționalități va avea aplicația, nu am dorit să mă cramponez în pasul de proiectare prea mult, ci mi-am dorit să trec direct la partea de implementare, fiind deschisă să ajustez cerințele funcționale și eventual reprezentarea datelor.

Implementarea a fost făcută iterativ, în cicluri scurte de codare și testare. Inițial am proiectat baza de date după ideea pe care o aveam despre aplicație.

Ulterior, am trecut la implementarea logicii de backend, începând prin a implementa entitățile și metodele fără să iau în calcul cheile străine, deci fără să implementez și relațiile existente în baza de date în clasele din pachetul de entități din baza de date. Ulterior am testat API-urile definite în clasele din controlere în Postman, iar după ce le-am văzut pe acestea funcționale am trecut mai departe. Am implementat relațiile și apoi am adăugat cele necesare fiecărei metode pentru ca acestea să se potrivească cu schimbările făcute. Un ultim pas a fost să adaug clasele de securitate, să definesc cine poate accesa diferite metode și să fac mici modificări în logica unor metode sau să adaug metode noi, care aveau nevoie în prealabil de implementarea securității.

Iar la final, pentru scopuri de debugging, dar și informative am adăugat încă o clasă, pentru a scrie într-un fișier detalii despre fiecare cerere, dar și în consolă.

Pe măsură ce am avansat în cod și am testat, bineînțeles că am avut și mici probleme, bug-uri, pe care le-am fixat , și de aceea consider că se potrivește metodologia Code and Fix.

Modelul Waterfall ar fi fost clar nepotrivit, mai ales pentru că este primul proiect de mărimea aceasta pe care îl implementez și consider că este mai greu pentru începători o abordare fără testare pe parcurs.

Tehnologiile alese

Având în vedere că aplicația mea este una WEB, este potrivit ca aceasta să fie împărțită în două mari componente și anume: frontend și backend.

Partea de backend conținând operațiile CRUD (Create, Read, Update, Delete) și logică de business și accesul la baza de date. Așadar, pentru accesul la baza de date am ales să folosesc MySQL în combinație cu limbajul de programare Java. Voi începe cu motivul pentru care am ales Java (https://docs.oracle.com/en/java/ documentația oficială). În primul rând este un limbaj de programare orientat pe obiecte(mă voi referi de acum încolo ca limbaj OO). Acest lucru este util deoarece în momentul în care lucrăm cu diferite entități existente în lumea reală, cum ar fi în cazul meu Programări, Analize, Donatori, putem vedea foarte ușor cum aceste lucruri din viața reală pot fi transferate ca obiecte. De asemenea, atunci când proiectăm clase într-un limbaj OO ale cărui baze și principii le regăsim conform Weisfeld Matt (2009), trebuie să respectăm principiile SOLID(conform Martin Robert 2018) , iar acestea ne ajută și mai bine să izolăm codul pentru a găsi problema mai ușor, chiar dacă diferite porțiuni de cod colaborează. De asemenea, structura aceasta a programării OO ajută la o mai bună lizibilitate și organizare a codului, astfel încât poate fi înțeles și peste mai mult timp atunci când revii la el. Deoarece am ales ca limbaj de programare Java, acesta deja facilitează conectarea cu diferite servicii de baze de date și există chiar și diferite clase abstracte numite repository care au implementate unele operații pe baza de date. De aceea am ales să lucrez cu baza de date MySQL (https://dev.mysql.com/doc/ documentația oficială) deoarece conectarea cu Java este destul de ușor de realizat și cooperează bine împreună. De asemenea, am mai implementat proiecte mai mici în trecut folosind această combinație, iar cooperarea aceea mi-a oferit încredere să continui astfel. Din nou, deoarece aplicația este web a fost nevoie să aleg un framework (o tehnologie care oferă un cadru) potrivit, și anume Spring(https://docs.spring.io/springframework/docs/current/reference/html/ documentație oficială). Spring oferă acces ușor la baza de date prin intermediul JPA Repository(https://docs.spring.io/spring-data/jpa/docs/1.6.0.RELEASE/reference/html/jpa.repositories.html documentația oficială), astfel încât nu este necesar să scriu query-urile manual, ci diferite operații de bază există deja și accesul se face destul de rapid. De asemenea, serverul web HTTP Tomcat este deja integrat de către Spring. Iar în termeni de arhitectură, Spring are suport integrat pentru arhitectura de tip Layers, pe care o voi detalia mai târziu.

Pentru partea de frontend, deși puteam alege să lucrez doar cu html, css și javascript fără să folosesc vreo tehnologie anume, am ales totuși să folosesc React (https://reactjs.org/ documentația oficială). În primul rând, am ales să folosesc această tehnologie și din motive estetice, pentru că există unele elemente gata construite și stilizate chiar în React, dar și

integrarea elementelor din Bootstrap este destul de ușor de realizat, deoarece există elemente gata stilizate special realizate pentru React.

Mai mult decât atât, având în vedere că aveam nevoie să trimit cereri HTTP către API-uri din backend, React poate funcționa cu librăria axios(https://axios-http.com/docs/intro documentație oficială), în care modul de trimitere ar cererilor este adaptat în funcție de tipul fiecărei cereri, iar răspunsul primit sau datele trimise sunt în format JSON, format suportat de backend. De asemenea, un lucru noi introdus sunt React Hooks, Poți atribui cu ajutorul acestuia 'stări' anumitor pagini fără să fie obligatoriu să creezi clase și poți transmite date între pagini mai ușor cu ajutorul acestora. Motivul principal pentru care eu le-am folosit a fost pentru a stoca răspunsul cererilor HTTP, iar în funcție de caz puteam să afișez informația mai ușor în interfață doar utilizând 'starea' respective.

Structura lucrării

Lucrarea va conține mai multe capitole, fiecare abordând o etapă diferită în procesul de elaborare a lucrării.

În introducere am vorbit puțin despre motivația alegerii temei, metodologie și tehnologii alese. Subiectul cel din urmă va fi reluat mai în detaliu în capitolul de implementare, deoarece voi vorbi mai pe larg despre funcționalități și modul în care tehnologiile pe care le-am ales au facilitat implementarea acestora.

Ulterior, primul capitol intitulat Analiza de Business va fi orientat către potențialii beneficiari, cercetare atât a pieței, cât și a competiției și modalităților prin care s-au realizat aceste cercetări. De asemenea, în cadrul acestui capitol vom vorbi și despre obiective, vom ilustra procese și activități, utilizând diferite diagrame.

Capitolul doi se referă la Proiectarea Sistemului. În acest capitol voi elabora cerințele funcționale și non-funcționale ale sistemului, acompaniate de diferite diagrame care vor face acest lucru mai ușor de înțeles, atât pentru persoanele cu un fundal tehnic, cât și pentru celelalte.

De asemenea, voi prezenta modul în care am organizat aplicația, iar pentru a face acest lucru voi utiliza diferite diagrame UML. De asemenea, alte diagrame din standardul UML mă vor ajuta să ilustrez modul de circulație al datelor, flow-ul aplicației, arhitectura propusă pentru sistem și tot ceea ce mă va ajuta în explicarea cât mai clară a procesului de proiectare a aplicației.

Capitolul trei, cel de Implementare, va fi orientat mai mult către partea tehnică și va prezenta diferite secvențe de cod pentru a exemplifica modul de implementare al unor

operații. De asemenea, voi prezenta în special porțiunile de cod despre care consider că sunt mai interesante sau au o logică mai ieșită din comun. De asemenea voi prezenta tipurile de date cu care am lucrat, modul în care am preluat datele din JSON-uri în frontend, partea de securitate implementata cu JW Token. Aici vor acoperi toate aspectele tehnice din cod, atât cele mai simple, însă voi pune accent în special pe părțile care mi-au creat mici probleme sau m-au făcut să ies din zona de confort și să gândesc diferit.

Capitolul patru este dedicat unei activități foarte importante din viața unui produs software și a cărui importanță este uneori subestimată, și anume cel de Asigurarea Calității. Aici voi descrie strategiile de testare pe care le-am aplicat pentru a mă asigura că descopăr cât mai multe bug-uri, defecte, deoarece niciodată nu putem știi sigur că o aplicație este lipsită de probleme.

Ultimul capitol, cel al concluziilor în care voi prezenta o sinteză succintă a paragrafelor, eventuale dezvoltări viitoare și în general observații adiționale care ar adăuga o valoare în plus lucrării.

1.Anaiză de business

În acest capitol al lucrării voi dezvolta modul în care am concretizat ideea pe care am avut-o într-un proiect. Mai specific, voi vorbi despre competitori, despre validarea ideii pe piață, despre ceea ce își dorește publicul și despre modul în care am decis cu privire la caracteristicile funcționale ale website-ului.

În momentul în care o idee începe să se contureze, consider că este foarte important să stabilim dacă aceasta chiar rezolvă o problemă reală a publicului, acest lucru în primul rând, urmând să ne validăm ideea tot în raport cu publicul și eventual să o dezvoltăm în funcție de dorințele acestora. Ulterior este necesar să vedem dacă putem oferi ceva în plus publicului față de ceea ce există deja pe piață și cu ce am putea să ne diferențiem noi de restul, acest lucru fiind făcut printr-o identificare al minusului competitorilor și fructificarea acestuia.

Problema pe care aplicația mea dorește să o adreseze se referă la lipsa interesului oamenilor de a dona sânge. Deși prezența donatorilor este într-o continuă creștere, despre acest subiect pot vorbi atât din punct de vedere al statisticilor, deoarece putem vedea multitudinea de campanii în acest sens pe care le organizează atât Ministerul Sănătății, cât și diferite ONG-uri externe, însă și din punct de vedere propriu, deoarece chiar eu sunt un donator regulat.

În acest an, 2022 am donat sânge de peste zece ori, însă de fiecare dată Centrul de Transfuzii din Cluj-Napoca era destul de liber. Excepție au făcut puținele dăți în care din nefericire avuseseră loc diferite accidente, iar oamenii alegeau să doneze în special pentru acea persoană.

Așadar, această experiență m-a făcut să îmi pun următoarele întrebări de studiu înainte chiar să dezvolt acest website: De ce oamenii care nu au donat niciodată nu o fac și de ce aceia care au făcut-o o dată pentru cineva în mod special nu s-au reîntors.

Cu aceste câteva întrebări în minte, cu dorința de contribui la rezolvarea unei probleme pe care o cunosc cât de cât îndeaproape și cu o idee creionată de rezolvare, am început o analiză în detaliu.

În cele ce urmează vom trece împreună prin trei capitole importante. Deoarece această problemă nu este una nouă și soluții aparente s-au mai încercat, am ales să încep prin studiul competitorilor. Voi vedea ce oferă celelalte aplicații sau website-uri în primul sub-capitol. Ulterior, în subcapitolul Metode de extragere a cerințelor voi detalia modul în care am ales să mă consult cu publicul și concluziile trase din fiecare discuție directă sau indirectă, iar nu în ultimul rând, în ultimul capitol voi trage o concluzie cu privire la ceea ce consider că ar fi cerințele pe care, cel puțin pentru început, ar trebui să le incorporez în ceea ce voi dezvolta eu. În acel capitol voi încerca să fac o paralelă între răspunsurile publicului și dorințele lor, ceea ce există deja la competitori, ceea ce lipsește, ceea ce ar trebui păstrat și bineînțeles ceea ce ar trebui dezvoltat.

1.1 Competitori

Centrele de Transfuzii, mai precis website-urile lor sunt primele analizate în acest capitol, deși ele nu sunt neapărat competitori, ci mai degrabă colaboratori în viitor, totuși trebuie observat ce informații și facilități oferă aceștia pentru a putea analiza ceea ce le lipsește utilizatorilor.

În momentul actual, am analizat website-urile unor centre situate în orașe mari, și anume centrele din orașele Cluj-Napoca, Timișoara, Craiova, Iași și București. Deși la primele centre designul paginii și modul în care utilizatorul are acces la informații este destul de modern, cu cele din urmă există o problemă în acest sens.

Informația din cadrul acestora este multă și distribuită inegal pe pagini, stilizarea obiectelor de pe pagină este una rudimentară și foarte neatractivă pentru utilizator. Scrisul este înghesuit, mult și mic, iar experiența utilizatorului încă de la accesul pe prima pagină începe neplăcut.

Cu toate acestea, primele 3 se descurcă bine la acest capitol, informația este structurată pe secțiuni destul de bine, butoanele și prima pagină este fie simplistă și elegantă fie mai încărcată și jucăușă, însă poziționată astfel încât utilizatorul să aibă parte de o experiență plăcută atunci când dorește să intre pe site.

Conținutul în sine surprinde toate aspectele despre procesul de donare al sângelui, însă dacă ne gândim mai profund, primul contact al utilizatorului este cu aspectul, deci este foarte probabil ca utilizatorul să nu mai ajungă măcar la aceste informații.

De asemenea, pe niciunul dintre website-uri nu ne putem face o programare la donat și nu găsim trimiteri către celelalte centre din țară. Pe unele dintre site-uri de putem vedea online rezultatul analizelor, însă procesul este destul de greoi deoarece pe lângă CNP avem nevoie și de codul donării în sine, pe care dacă nu îl păstrăm de atunci nu avem cum să îl mai redobândim. Așadar, nu avem posibilitatea de a vedea undeva întreg istoricul nostru al donărilor.

De asemenea, tot în domeniul web, există diferite site-uri, care nu sunt corespondente unor instituții de sub tutela Ministerului Sănătății, și care încearcă să deruleze diferite campanii de conștientizare. Pe niciunul dintre aceste site-uri nu am găsit posibilitatea să îmi fac programare la donare, pe câte unele dintre ele am găsit punctual centrele de donare din București, însă nu din toată țara punctual. Cu toate acestea, pe fiecare site există detaliat informații cu privire la procesul de donare, condiții de admisibilitate și beneficii, de aceea consider că acesta este un aspect important pe care ar trebui să îl conțină și websiteul meu.

Cu toate acestea, există și aplicații pentru telefon care au încercat să umple acest gol al funcționalităților pe care site-urile centrelor de transfuzii nu îl oferă.

Prima dintre ele este Donorium, o aplicație al cărei utilizator încercat să fiu și eu, însă există câteva aspecte care nu sunt acoperite și pe care le consider importante, iar de aceea am ales să nu mai utilizez aplicația.

În primul rând, există anumite lucruri în plus pe care le oferă față de website-urile menționate anterior și anume:

-oferă posibilitatea de a-ți face programare

-poți selecta locația

-există secțiune pentru apeluri la donare

-oferă posibilitatea de a vedea cât timp a trecut de la ultima donare sau cât timp mai există până la programarea următoare

Cu toate acestea, există minusuri majore la fiecare dintre funcționalități: deși îți poți face programare, nu primești nicio dovadă în niciun mod că această programare este efectuată și îți aparține ție, singurul mod în care ai putea dovedi cumva că ai programare este să execuți o captură de ecran în care să fie vizibil cronometrul aferent. Nu există SMS sau e-mail de confirmare în care să regăsești detaliile programării, iar aceste nici nu există în vreo bază de date intermediară a centrului pentru a te identifica și nici un istoric al programărilor nu există.

Deși poți selecta locația, pentru București cel puțin, chiar dacă există și alte centre unde se poate dona cu excepția CTS București, acestea nu apar în aplicație.

Iar referitor la secțiunea de apeluri la donare, nu există o transparență cu privire la cine este autorul acelor apeluri, medicii sau cineva responsabil cu acest lucru. Și mai mult decât atât, pe perioada de 3 luni cât am utilizat aplicația și chiar când am consultat-o din nou pentru analiza acesteia, nu am observat niciun apel la donare existent în aplicație în toata țara. Acest lucru este practic imposibil, deoarece din păcate accidente se întâmplă mereu și lumea are nevoie mereu de sânge, deci această funcționalitate nu este complet și corect dezvoltată, așadar deși există cu numele, nu aduce vreun beneficiu real.

De asemenea, o aplicație care oferă mult mai multe este BloodDoChallange. În cadrul acestei aplicații există de asemenea toate funcționalitățile identificate la cealaltă, însă implementarea acestora este îmbunătățită. Aici poți vedea istoricul donărilor tale, însă bineînțeles fără rezultatele analizelor. De asemenea, nici în acest caz nu sunt acoperite toate centrele de transfuzii din țară.

De asemenea, nici în cadrul acestei aplicații o dovadă tipărită cum că aplicația s-a înregistrat nu există, e-mail sau SMS.

O altă problemă importantă care cuprinde tot ceea ce există este descentralizarea datelor. Faptul că aceste alternative, cât și sistemele centrelor în sine nu sunt integrate unele cu celelalte, deci nu pot face operații pe aceeași bază de date. De aceea pot apărea probleme de exemplu când decizi să treci de la o aplicație la cealaltă, deoarece pierzi data

ultimei donări. De asemenea, din punct de vedere al faptului că donatori vor să își vadă și analizele, procesul este puțin greoi în acest sens deoarece nu pot face acest lucru doar cu CNP-ul, ci mai au nevoie si de acel cod adițional. Faptul că întregul proces nu este centralizat într-un singur website/ aplicație este o problemă pentru donatori, deoarece mulți preferă să nu se complice. De asemenea, și lipsa de un design comun tuturor centrelor de transfuzii este o problemă și faptul că nu fac trimitere unele la celelalte.

Dacă ar fi să rezum această analiză, principala problemă este lipsa de integrare a părților implicate în proces și centralizarea datelor, deși probabil că la nivelul softului folosit în centre există o bază de date comună.

Așadar scopul meu final este să ofer un prototip pentru un website care ar putea fi divizat apoi la nivel de centre de transfuzii, însă cu un design comun, care să unifice atât funcționalitățile dorite de utilizator, cât și cele necesare pentru cealaltă fațetă, și anume centre, pacienți și medici.

1.2 Metode de extragere a cerințelor

În urma analizei prin propria perspectivă a alternativelor de pe piață, am decis că este important să discut și cu potențialii beneficiari și să observ ce și-ar dori aceștia de la o astfel de aplicație și care este punctul lor de vedere cu privire la problema lipsei de sânge, diferite motivații personale și eventuali stimuli care i-ar face să-și reconsidere poziția despre acest subiect sau care i-ar determina să doneze sânge.

Pentru a reuși să ajung totuși la un număr rezonabil de păreri, deoarece doream ca acest studiu să fie cât de cât reprezentativ am ales să aplic metoda chestionarului. Deși este posibil ca rezultatele pe care le voi obține să nu fie reprezentative pentru toate segmentele populației, acest lucru din prima persoanelor la care aș putea avea acces și la care ar ajunge chestionarul meu, totuși voi avea cel puțin o perspectivă bine definită asupra unor categorii de persoane, cu posibilitatea ca în viitor să pot ajunge și la altele, adaptând și dezvoltând aplicația în consecință.

Chestionarul întreg se regăsește în Anexa 1 a acestei lucrări, însă voi analiza răspunsurile punctual la fiecare dintre întrebările adresate în cadrul acestui sub-capitol și voi defini categoriile de persoane intervievate în funcție de întrebările de identificare.

Aș dori să încep prin analiza răspunsurilor la întrebările de identificare, deoarece cred că este important să avem definit un profil al categoriei de utilizator către care ne vom îndrepta atenția în mod principal. Bineînțeles, nu voi neglija nici informațiile oferite de celelalte categorii, însă este bine să încep cu ceva categoric și să extind ulterior dacă va fi cazul.

După cum putem observa, împărțirea pe sexe nu este tocmai egal echilibrată, însă nu există nici o majoritate covârșitoare în vreo direcție, așadar voi considera că mă adresez amândurora, deși cu precădere fetelor, deoarece acestea cumulează două treimi dintre aceste răspunsuri.

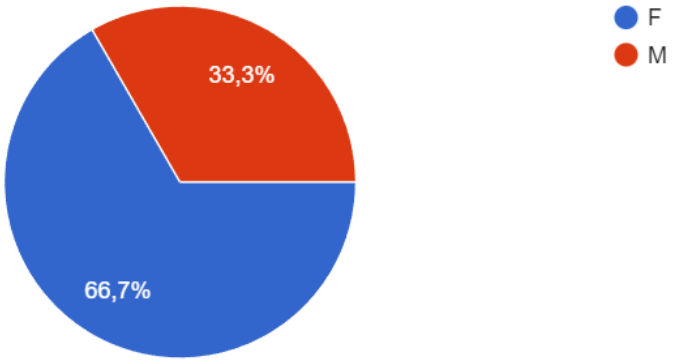


Figura 1 - Împărțirea pe sexe a respondenților chestionarului

Când vorbim despre împărțirea pe categorii de vârstă, rezultatele sunt mai evidente, în mod clar categoria de vârstă țintită este 18-24 de ani. Acest lucru este unul benefic din punctul meu de vedere, deoarece dacă un obicei bun apare devreme în viața cuiva acesta persistă pe parcursul vieții, în consecință este un lucru bun că voi putea identifica nemulțumirile și așteptările acestora.

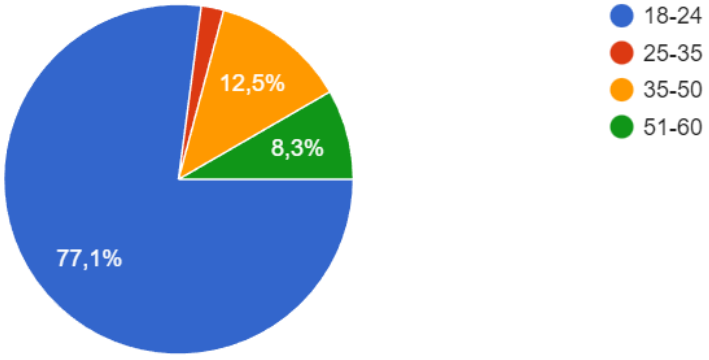


Figura 2 - Împărțirea pe categorii de vârstă a respondenților chestionarului

Nu în ultimul rând, atunci când vorbim despre împărțirea pe criterii educaționale, marea majoritate a respondenților sunt fie absolvenți de liceu, fie de facultate, așadar în principal vom vorbi de respondenți cu un nivel de educație ridicat.

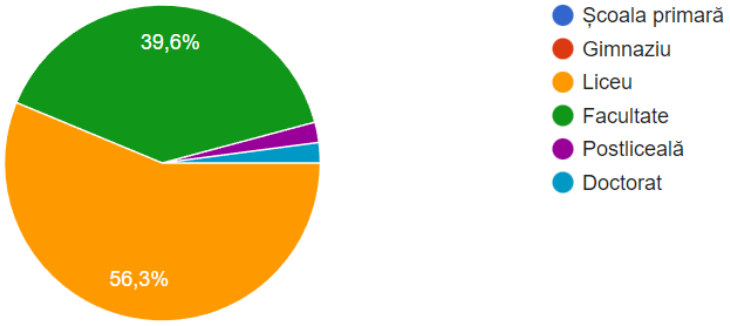


Figura 3 - Împărțirea pe criterii educaționale a respondenților chestionarului

Așadar, principala categorie de public a căror perspectivă o vom analiza sunt tinerii între 18 și 24 de ani, absolvenți de liceu sau de facultate, atât femei cât și bărbați, preponderent femei. Este important să observăm că vorbim totuși cu un public educat, deci despre care se presupune că este totuși cât de cât informat despre ceea ce se întâmplă în jur, deci teoretic și despre problema în cauză.

Pe parcursul chestionarului există patru întrebări închise cu variantele de răspuns Da/Nu. Inițial le vom parcurge pe acestea pe rând, ulterior voi trece la întrebarea închisă cu mai multe răspunsuri posibile, iar în final cele două întrebări deschide.

La întrebarea legată de eventuala existență în trecut a unei donări sanguine, 61% dintre respondenți susțin că nu au donat sânge niciodată.

De asemenea, când a venit vorba de cunoașterea beneficiilor, atât personale cât și ale pacienților, 53% dintre respondenți nu cunoșteau faptul că o donare poate salva până la 3 vieți, dar 75% cunoșteau faptul că se primesc bonuri de masă în valoare de 70 de lei și o zi liberă în urma donării.

De asemenea, am dorit să aflu dacă aceia care au mai donat sânge au folosit vreuna din aplicațiile pe care le-am menționat anterior printre competitori, însă 85% au menționat că nu au folosit o astfel de aplicație.

Cu toate acestea, am dorit să văd dacă o astfel de aplicație de tipul celei pe care o propun care să automatizeze procesul de programare la donare, primire a analizelor și care să trimită o dovadă a programării, toți cei care ar putea dona sânge (deoarece au existat puțini respondenți care nu pot dona din motive de admisibilitate) au răspuns că ar folosi o astfel de aplicație.

În ceea ce privește întrebarea cu mai multe variante de răspuns, aceasta reprezintă de fapt o încercare de a identifica principalele cauze pentru care oamenii consideră că există o lipsă de interes legată de donarea de sânge.

Am decis să evidențiez acest rezultat utilizând o diagramă Pareto.(conform https://asq.org/quality-resources/pareto , 2022) O diagramă de acest tip conține pe axa X

categoriile de interes, în cazul nostru motivele pentru care oamenii nu sunt interesați de acest proces, iar pe axa-Y frecvențele acestora. Principiul lui Pareto adaptat în cazul nostru ar spune că 80% dintre efecte, în cazul nostru 80% dintre oameni nu donează sânge din cauza primelor 20% de elemente pe care le-am identificat drept motive pentru care nu o fac.

Cauzele identificate enumerate de la dreapta la stânga după cum sunt ele prezente în diagramă sunt următoarele:

-Lipsa promovării procesului în sine de donare a sângelui

-Frica de ace

-Neinformarea cu privire la beneficiile donării de sânge

-Frica de faptul că acest proces ar putea face rău

-Durata de așteptare înainte și din cadrul procesului

-Alte motive

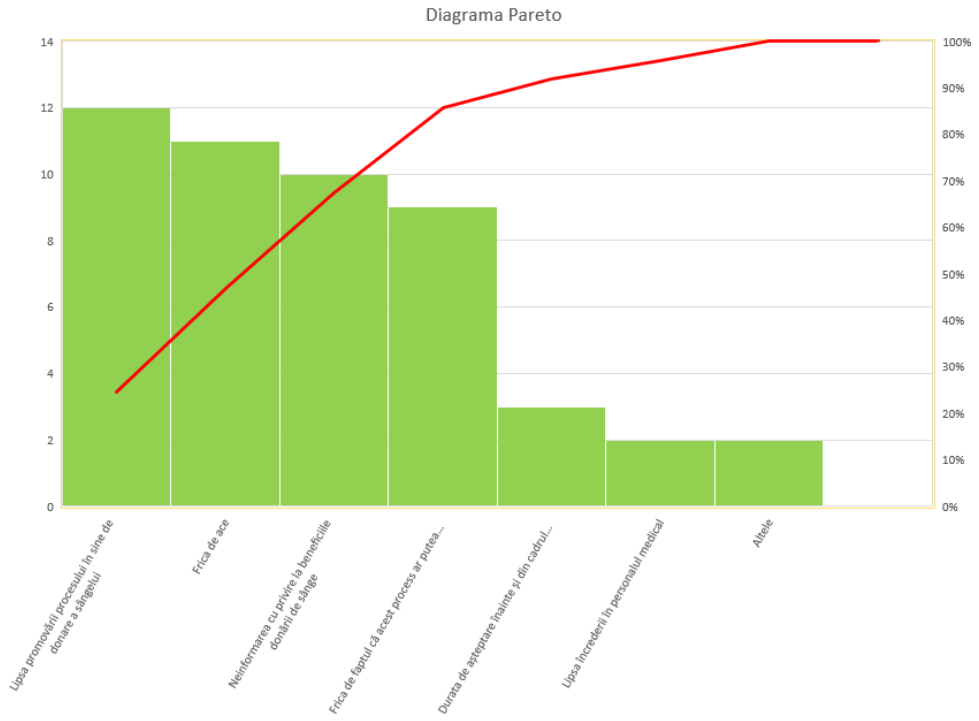


Figura 4 - Diagrama Pareto cu privire la cauzele lipsei de interes pentru donarea de sânge

Așadar, după cum putem observa, în principiu 50% dintre oameni consideră că principalele cauze sunt fie neinformarea cu privire la beneficii, fie neinformarea cu privire la proces în sine. Deși problema fricii de ace este cam greu de rezolvat prin intermediul unei aplicații web, celelalte cauze pot fi totuși adresate într-un fel sau altul. Dacă ar exista o aplicație care să însumeze toate funcționalitățile dorite de utilizator și de care este nevoie, aceasta ar putea fi promovată mult mai mult și ar putea fi mai ușor integrată în viața donatorului pentru a crea un obicei.

Deci, ceea ce putem observa este că oamenii și-ar dori și ar fi deschiși să folosească o astfel de aplicație și sunt dispuși să încerce ceva diferit, și anume să doneze sânge.

1.3 Culegerea cerințelor

Acest sub-capitol va evidenția în principiu cerințele funcționale pe care le-am ales pentru website, atât în funcție de ceea ce au răspuns utilizatorii, cât și în urma analizei competitorilor.

Au existat trei întrebări deschise în chestionar în cadrul cărora, deși au existat răspunsuri formulate diferit, bineînțeles, se pot observa anumite tipare sau răspunsuri ale căror esență este aceeași.

Prima întrebare se referă la motivația care i-ar determina pe oameni să își reconsidere decizia și să meargă să doneze sânge.

La acest capitol, părerile au fost foarte împărțite, însă pe majoritatea i-ar determina faptul că cineva drag are nevoie de ajutor sau că este lipsă acută de sânge. Problema este că acest lucru se întâmplă în fiecare zi, însă nu există transparență în acest sens, deoarece stocul de sânge nu este actualizat, deci nu este vizibil câtă nevoie este de sânge.

Totuși, alte persoane susțin că transparența întregului proces de donare și automatizarea acestuia i-ar face să își reconsidere decizia.

Așadar, cu funcționalitățile potrivite s-ar putea reuși acoperirea a majoritate dintre motivele care ar schimba situația din perspectiva majorității.

În final, există două întrebări în strânsă legătura una cu cealaltă, una dintre ele își dorește să identifice cel puțin două funcționalități pe care utilizatorii și-ar dori să le vadă într-o astfel de aplicație, cât și motivele pentru care ar folosi sau nu o astfel de aplicație.

Frecvent am regăsit ca motivație faptul că trăim într-o eră a digitalizării, iar digitalizarea acestui proces este natural să se întâmple. De asemenea, un factor important este că ar salva timp, că le-ar trimite un e-mail cu programarea și astfel nu ar mai uita, că este simplu de utilizat și la îndemâna oricui care are un dispozitiv conectat la Internet și în principal pentru ca ar ușura procesul de donare și parcurgerea acestuia.

În consecință, principalele funcționalități pe care tinerii și le doresc ar fi următoarele:

-Programarea pentru donare

-Primirea unei dovezi pentru programare

-Vizualizarea istoricului medical

-Informarea despre proces

-Vizualizarea rezultatelor analizelor

-Vizualizarea locațiilor de donare

-Vizualizarea centrelor în care este nevoie de donatori/ a cazurilor în sine care au nevoie de ajutor

Acestea vin la pachet automat cu anumite funcționalități pentru partea spitalului, și anume:

- Introducerea analizelor și confirmarea programărilor
- Gestionarea apelurilor la donare
- Vizualizarea stocurilor, deoarece acest lucru i-ar putea determina pe oameni să meargă să doneze
- Gestionarea stocului

Și bineînțeles, întregul sistem în sine va veni la pachet cu diferite cerințe de gestionare și administrare a instituției în sine, gestionarea conturilor medicilor. De asemenea, există și cerințe ascunse printre rânduri, deoarece evident ca să te poți programa este nevoie să te poți înregistra pe platformă și să îți poți accesa contul.

Toate aceste detalii vor fi explicate mai pe larg în capitolul de proiectare, însă am considerat că este importantă o imagine de ansamblu asupra dorințelor celor care vor folosi aplicația și că în principal cerințele au fost elaborate în colaborare cu publicul.

2.Proiectare

Acest capitol are rolul de a forma o imagine mai clară, de ansamblu, dar în unele locuri și mai particulară asupra modului în care este proiectată aplicația. Toate aceste detalii vor fi acompaniate de diagramele specifice și explicațiile acestor diagrame. Diagramele oferă o perspectivă vizuală mult mai bună asupra subiectului în discuție în funcție de tipul acesteia și deoarece vor fi generate urmărind standardul UML sunt ușor de urmărit de persoanele cu un fundal tehnic și cunoștințe în acest text. Cu toate acestea, printre ele se regăsesc diagrame care ar trebui să fie mai aproape de client, care acesta de multe ori nu are neapărat un astfel de fundal, însă aspectul acestora este destul de user-friendly, poate cel mai bun exemplu în acest sens fiind diagrama use-case, cea care va deschide acest capitol.

Deoarece capitolul anterior am discutat despre modul de culegere al cerințelor și am analizat atât răspunsurile primite la chestionar, cât și ceea ce oferă alte aplicații similare, făcând la final un rezumat despre care ar trebui să fie, în mare, cerințele sistemului, modelul use-case va deschide acest capitol deoarece proiectarea ulterioară ține cont de aceste cerințe, deși este concepută în așa fel încât poate face față volatilităților. Ulterior voi descrie arhitectura aleasă pentru sistem, motivația alegerii și o analiza a diagramei prin care voi

exemplifica. După ce aceste aspecte vor fi discutate, voi trece mai departe la un nivel de detaliu mai mare, explicând modul de gândire al bazei de date, al pachetelor și al claselor.

Iar mai departe, vom avea diferite diagrame prin care este exemplificat cursul utilizatorului prin website, și anume diagrama de stare și de activitate.

2.1 Modelul use-case și cerințele funcționale/non-funcționale ale sistemului

2.1.1 Diagrama Use-Case

Diagrama Use-Case este cel mai simplist mod în care putem afișa spre înțelegerea atât a noastră cât și a partenerilor toate funcționalitățile sistemului. Aceasta este ușor de citit și înțeles de către ceilalți deoarece aspectul ei este unul intuitiv și apropiat de lumea înconjurătoare. Actorii sunt exemplificați prin niște schițe ale unor omuleți, stick-man, iar sub ei este notat tipul actorului, iar interacțiunea lor cu sistemul este reprezentată printr-o linie dreaptă care-i unește cu funcționalitatea sau funcționalitățile existente pentru aceștia. Funcționalitățile sunt reprezentate în interiorul sistemului și scrise în ovale, într-un mod concis, dar relativ clar și sugestiv.

Figura 5 de mai jos ilustrează actorii pe care îi conține aplicația mea, împreună cu funcționalitățile fiecăruia. Despre acestea putem observa cu ajutorul săgeților că unele sunt comune celor trei actori, altele sunt specifice.

Diagrama întreagă conține 3 tipuri de actori: donatori, personal medical și administrator.

Cele mai puține cazuri de utilizare le are administratorul, care poate intra în cont sau poate ieși din cont, poate vedea stocul de sânge din toate spitalele și este responsabil cu gestiunea în baza de date a spitalelor și a personalului medical din acele spitale. Mai specific, poate edita informațiile de identificare a conturilor spitalului și poate adăuga exclusiv noi conturi de personal medical pentru spitalul respectiv. Conturile sunt doar de doctori, deoarece în fiecare spital există doar un administrator, iar contul acestuia va fi creat de administratorul de sistem, în acest caz de către mine manual. De asemenea, acesta poate dacă dorește să își schimbe parola.

Donatorul și Personalul Medical au și ei primele 3 funcționalități pe care le are Administratorul, însă în momentul în care doresc să intre în cont, în funcție de rolul asociat contului fiecare dintre ei dispune de diferite funcționalități după ce accesează contul.

Donatorul își poate gestiona programările, mai exact poate adăuga o programare dacă îndeplinește condițiile, adică au trecut minim 72 de zile de la ultima donare, iar atât timp cât programarea nu este confirmată, acesta o poate șterge în caz că s-a răzgândit. Donatorul va primi o înștiințare pe e-mail atât la crearea unei programări, cât și la ștergerea acesteia. De asemenea, donatorul își poate vedea istoricul donărilor, la care sunt atașate și rezultatele fiecărui test efectuat sângelui cu data respectivă la care s-a făcut programarea, aceste rezultate fiind și ele vizibile.

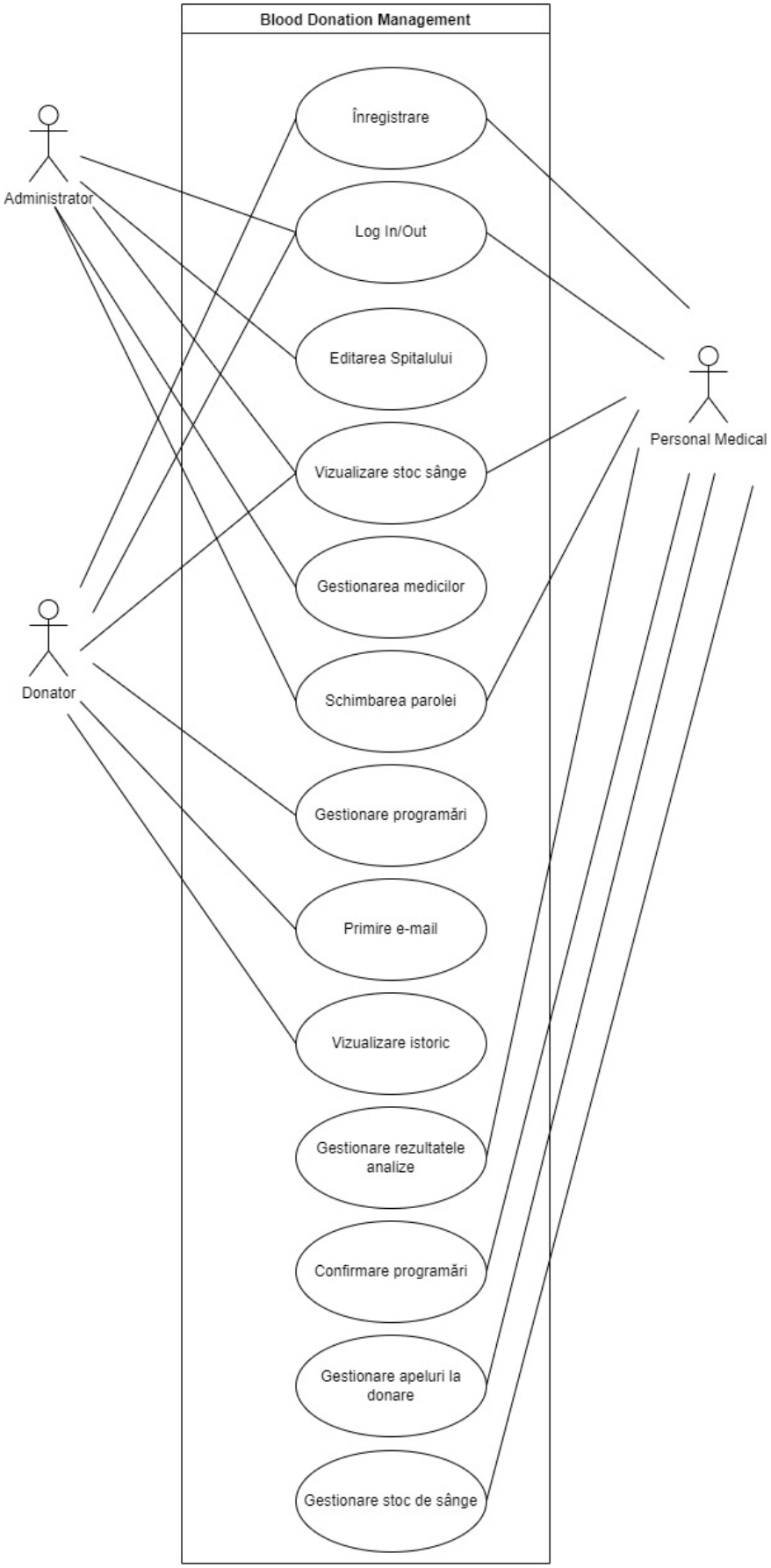


Figura 5 - Diagrama Use-Case a sistemului

Personalul medical are responsabilitatea de a confirma programările donatorilor care au îndeplinit până la capăt o donare. De asemenea, aceștia trebuie sa introducă datele testelor de sânge pentru fiecare donator. De asemenea, având în vedere că la fiecare de sânge confirmată stocul crește cu 0.45 litrii, pentru a reuși să menținem stocul real cineva trebuie să se ocupe de înregistrarea sângelui consumat. Așadar, la finalul zilei această responsabilitate revine personalului medical, de a introduce cantitatea consumată de sânge din fiecare grupă.

Și nu în ultimul rând, pot gestiona apelurile la donare. Adică, pot face o cerere pentru donare pentru un anumit pacient în cazul în care acesta are nevoie urgentă de sânge și o pot confirma o dată ce pacientul nu mai are nevoie de sânge sau stocul necesar s-a atins. De asemenea, aceste apeluri sunt vizibile din aplicație atât pentru doctori, însă ei pot vedea apelurile și programările corespunzătoare doar spitalului lor, însă donatorii pot vedea toate apelurile existente.

Detalierea cazurilor de utilizare:

1.Denumire caz de utilizare: Înregistrare

Actor: Donor

Scenariul principal:

Precondiții: - niciuna

Postcondiții: - o nouă înregistrare cu datele introduse de utilizator există în baza de date

Comportament: Atunci când un utilizator donator dorește să se înregistreze, acesta trebuie să introducă anumite date: nume, prenume, adresa de e-mail, parola, CNP, număr de telefon și grupa de sânge. În momentul în care se apasă butonul de creare a contului, dacă totul este în regulă, un nou cont este creat și o înregistrare specifică există în baza de date care conține toate informațiile transmise de utilizator.

Extensie: Posibile cazuri în care operația ar eșua ar fi atunci când e-mailul și CNP-ul care sunt identificatori unici au mai fost folosite în altă înregistrare.

2.Denumire caz de utilizare: Log In

Actor: Donator, Administrator, Personal Medical

Scenariul principal:

Precondiții: - să nu fie autentificați în cont

Postcondiții: - utilizatorii sunt autentificați în cont conform credențialelor și rolului

Comportament: Atunci când un utilizator dorește să își acceseze contul trebuie să introducă numele de utilizator și parola. Dacă există un cont cu datele corespunzătoare în baza de date, atunci utilizatorul este autentificați în cont.

Extensie: Posibile cazuri în care ar eșua operația ar fi atunci când numele de utilizator sau parola nu sunt introduse corect.

3.Denumire caz de utilizare: Log out

Actor: Administrator, Donator, Personal Medical

Scenariul principal:

Precondiții: - să fie autentificați cu unul dintre cele 3 tipuri de conturi

Postcondiții: - utilizatorul nu mai este autentificat și este redirecționat la pagina de acasă

Comportament: Atunci când utilizatorul este logat, paginile pe care le poate accesa sunt bazate pe contul care este asociat. Atunci când este apăsat butonul de log out, utilizatorul nu mai este logat în cont și are posibilitatea doar funcționalităților posibile pentru utilizatorul vizitator.

4.Denumire caz de utilizare: Vezi stocul de sânge

Actor: Administrator, Donator, Personal Medical

Scenariul principal:

Precondiții: - niciuna

Postcondiții: - un tabel care conține informații legate de stocul de sânge din fiecare spital este afișat în interfață

Comportament: Atunci când utilizatorul apasă butonul de vizualizare a stocului, indiferent dacă este autentificat sau nu, trebuie să ajungă pe o pagină unde îi este arătat tabelul cu informații legate de stocul de sânge al spitalelor.

5.Denumire caz de utilizare: Schimbare parola

Actor: Administrator, Personal Medical

Scenariul principal:

Precondiții: - să fie autentificați într-unul din tipurile de conturi Administrator/Personal Medical

Postcondiții: - parola veche nu mai există criptată în baza de date și este înlocuită cu noua parolă

Comportament: Atunci când utilizatorul este autentificat într-unul din conturile specificate mai sus el își poate schimba parola. Trebuie să introducă vechea parolă și să noua parolă. După ce schimbarea s-a produs, parola veche nu mai există asociată cu acest cont, iar autentificarea se face cu noua parolă.

Extensie: Un caz când această operație ar eșua este atunci când parola veche este introdusă greșit.

6.Denumire caz de utilizare: Editare spital

Actor: Administrator

Scenariul principal:

Precondiții: - să fie autentificat cu un cont de administrator

Postcondiții: - înregistrarea spitalului respectiv din baza de date va avea schimbate doar câmpurile care sunt editate, iar celelalte rămân la fel

Comportament: Atunci când un administrator este autentificat în cont, acesta poate schimba informații legate de spitalul pe care îl are atribuit. Dacă decide să schimbe un anumit câmp sau mai multe, doar valorile respective se vor schimba, iar celelalte vor rămâne la fel.

Extensie: Un caz când această operație ar eșua este atunci când datele introduse sunt duplicate.

7.Denumire caz de utilizare: Gestionare Medici

Actor: Administrator,

Scenariul principal:

Precondiții: - să fie autentificat cu un cont de administrator

Postcondiții: - o nouă înregistrare cu datele introduse de administrator există în baza de date în tabela corespunzătoare personalului medical.

Comportament: Atunci când administratorul dorește să adauge un cont specific personalului medical al spitalului său acesta trebuie să introducă informațiile necesare, cu excepția parolei, deoarece aceasta va fi generată automat ca fiind numele de utilizator și numărul de telefon, inițial.

Extensie: Un posibil caz în care această operație ar eșua este atunci când numele de utilizator sau numărul de telefon sunt duplicate.

8.Denumire caz de utilizare: Gestionare programări - Adăugare

Actor: Donator

Scenariul principal:

Precondiții: - să fie autentificat cu un cont de donator

Postcondiții: - o nouă înregistrare cu datele introduse de utilizator există în baza de date în tabela corespunzătoare programărilor

Comportament: Atunci când donatorul dorește să își facă o programare, acesta trebuie să aleagă o dată și oră disponibile, cât și spitalul, iar apoi după salvare, o înregistrare conținând aceste detalii va fi adăugată în baza de date în tabela specifică.

Extensie: Un posibil caz în care această operație ar eșua este atunci când utilizatorul are încă o programare viitoare sau când nu au trecut 72 de zile de la ultima donare.

9.Denumire caz de utilizare: Gestionare programări - Ștergere

Actor: Donator

Scenariul principal:

Precondiții: - să fie autentificat cu un cont de donator și să existe o programare viitoare

Postcondiții: - înregistrarea cu datele specifice celei șterse nu mai există în baza de date

Comportament: Atunci când donatorul dorește să își șteargă o programare viitoare, acesta trebuie doar să apese pe butonul respectiv, în cazul în care are o astfel de programare care poate fi ștearsă, iar apoi datele acestei programări nu vor mai exista în baza de date.

10.Denumire caz de utilizare: Primire e-mail

Actor: Donator,

Scenariul principal:

Precondiții: - să fie autentificat cu un cont de donator și să fi adăugat sau șters o programare

Postcondiții: - un e-mail cu datele programării adăugate/șterse este trimis având la destinatar e-mailul utilizatorului

Comportament: Atunci când donatorul dorește să își șteargă o programare viitoare sau să adauge una nouă, acesta primește un e-mail cu detaliile specifice programării în cauză.

Extensie: Această operație ar putea eșua în cazul în care nu există conexiune la internet sau adresa destinatarului nu există.

11.Denumire caz de utilizare: Vizualizare istoric

Actor: Donator

Scenariul principal:

Precondiții: - să fie autentificat cu un cont de donator și să aibă programări confirmate în trecut

Postcondiții: - toate programările trecute și viitoare vor fi afișate în interfață cu detaliile aferente, iar cele confirmate vor avea atașate rezultatele analizelor

Comportament: Donatorul poate vedea în interfață toate programările trecute sau cea viitoare dacă există cu detaliile aferente. Pentru programările confirmate în trecut există posibilitatea de a vedea rezultatele analizelor.

12.Denumire caz de utilizare: Gestionare rezultate analize

Actor: Personal medical

Scenariul principal:

Precondiții: - să fie autentificat cu un cont de personal medical și să existe în spitalul lui programări confirmate, fără analize adăugate

Postcondiții: - pentru programarea selectata, înregistrarea corespunzătoare analizelor va fi adăugată în baza de date

Comportament: Personalul medical poate adăuga rezultatele analizelor sângelui doar dacă programarea este confirmată, dar nu are analize deja adăugate. După ce este selectată programarea și sunt completate câmpurile necesare analizelor, o nouă înregistrare este adăugată în tabelul specific rezultatelor și este conectat cu programarea pentru care s-a adăugat.

13.Denumire caz de utilizare: Confirmare programări

Actor: Personal medical

Scenariul principal:

Precondiții: - să fie autentificat cu un cont de personal medical și să existe în spitalul lui programări neconfirmate

Postcondiții: - programarea selectată devine confirmată, fapt regăsit și în baza de date în câmpul respectiv

Comportament: Personalul medical poate confirma o programare din spitalul lui dacă aceasta nu a fost deja confirmată. Când este apăsat acest buton, câmpul respectiv este schimbat în baza de date pentru programarea respectivă.

14.Denumire caz de utilizare: Gestionare apeluri la donare - Adăugare

Actor: Personal medical

Scenariul principal:

Precondiții: - să fie autentificat cu un cont de personal medical

Postcondiții: - o nouă înregistrare cu datele introduse de utilizator există în baza de date în tabela corespunzătoare apelurilor la donare

Comportament: Atunci când personalul medical dorește să își adauge un nou apel la donare, acesta trebuie să introducă numele pacientului, grupa de sânge și condiția medicală, iar apoi după salvare, o înregistrare conținând aceste detalii va fi adăugată în baza de date în tabela specifică apelurilor la donare.

15.Denumire caz de utilizare: Gestionare apeluri la donare - Confirmare

Actor: Personal medical

Scenariul principal:

Precondiții: - să fie autentificat cu un cont de personal medical și să existe în spitalul lui apeluri la donare neconfirmate

Postcondiții: - apelul la donare selectat devine confirmat, fapt regăsit și în baza de date în câmpul respectiv

Comportament: Personalul medical poate confirma un apel la donare din spitalul lui dacă aceasta nu a fost deja confirmat atunci când nu mai este nevoie de sânge pentru acel pacient. Când este apăsat acest buton, câmpul respectiv este schimbat în baza de date pentru apelul la donare respectiv.

16.Denumire caz de utilizare: Gestionare stoc de sânge

Actor: Personal medical

Scenariul principal:

Precondiții: - să fie autentificat cu un cont de personal medical

Postcondiții: - stocului de sânge pentru grupa selectată din spitalul specific personalului medical îi scade cantitatea cu valoarea specificată

Comportament: Personalul medical poate gestiona stocul de sânge, adică să înregistreze cât sânge s-a consumat dintr-o anumită grupă pentru a avea o evidență a stocului. Personalul medical trebuie să introducă grupa și cantitatea consumată, iar această modificare se va vedea apoi în baza de date, iar noua cantitate a stocului grupei respective din spitalul personalului medical va scădea cu cantitatea introdusă.

Extensie: Această operație ar putea eșua dacă personalul medical introduce o cantitate consumată mai mare decât stocul disponibil pentru grupa respectivă.

2.1.2 Cerințe funcționale

Așadar, **cerințele funcționale** ale sistemului, după cum pot fi ele observate și din diagrama use-case, sunt următoarele:

- Aplicația va conține 3 tipuri de utilizator după cum urmează: administrator, personal medical și donator.
- Utilizatorul vizitator va avea funcționalitate limitată. El poate vedea stocul de sânge al spitalelor și se poate înregistra.
- Vizualizarea stocului de sânge al spitalelor este posibilă pentru toate tipurile de utilizatori, atât autentificați cât și neautentificați.

Pentru Administrator:

- Un spital are un singur administrator, adăugat manual în sistem. Nu există posibilitatea de a-și crea un cont.
- Acesta își poate schimba parola.
- Acesta poate edita informațiile spitalului la care este atribuit.
- Acesta poate crea conturi de personal medical pentru spitalul la care este atribuit.

Pentru Personalul Medical:

- Contul acestuia este creat de administrator.
- Poate intra și ieși din cont.
- Acesta își poate schimba parola.
- Poate vizualiza programările la donare din spitalul în care lucrează.
- Poate confirma programările la donare din spitalul în care lucrează.
- Poate adăuga analize programărilor confirmate din spitalul în care lucrează.
- Poate crea noi apeluri la donare pentru un pacient în caz că acesta are nevoie de sânge. Trebuie menționat numele, grupa de sânge și condiția medicală a pacientului.
- Poate înregistra cantitatea de sânge consumat într-o zi dintr-o anumită grupă.

Pentru donator:

- Acesta își poate crea un cont în aplicație.
- Poate intra și ieși din cont.
- Poate adăuga o programare dacă au trecut 72 de zile de la ultima programare confirmată. Nu poate avea mai mult de o programare viitoare activă.

- Poate șterge o programare neconfirmată.
- Primește e-mail când adaugă sau șterge o programare în care se regăsesc detaliile acesteia.
- Își poate vizualiza istoricul programărilor. Pentru programările confirmate își poate vedea istoricul analizelor.
- Poate vedea toate apelurile la donare, atât cele neconfirmate cât și cele confirmate.

2.1.3 Cerințe non-funcționale

Pentru enunțarea **cerințelor non-funcționale** voi utiliza clasificarea făcută de modelul FURPS(conform Watson Mike, 2006) și voi exemplifica așteptările aplicației mele de pe fiecare nivel.

F-ul modelului se referă la Funcționalitate în general, iar mai specific Capabilitate, Reutilizare și Securitate.

Pentru aplicația mea, prima cerință non-funcțională va fi legată de **Securitate**. Parolele nu vor fi ținute în clar în baza de date, ci criptate. Pentru a face acest lucru, framework-ul Spring oferă deja destul de multe opțiuni de securitate integrate, însă voi detalia în următorul capitol acest lucru. De asemenea, autentificarea va fi bazată pe token-uri generate la intrarea în cont și care au o anumită perioadă de expirare. Funcționalitățile vor fi protejate de necesitatea acestor token-uri, așadar dacă un utilizator nu are rolul potrivit, acesta nu poate accesa anumite funcționalități. De asemenea, este important ca atunci când facem programări sau accesăm programări și apeluri la donare acestea să fie protejate, iar modificările să poată fi făcute doar asupra programărilor care aparțin utilizatorului care face modificarea, sau în cazul medicilor doar asupra a ceea ce aparține spitalului lor.

Următorul aspect este legat de **Utilizabilitate**, mai specific UX, adică experiența utilizatorului pe pagină, consistență, estetică sau intuitivitate, toate acestea sunt calități pe care produsul meu trebuie să le posede. Navigarea între pagini trebuie să fie intuitivă și utilizatorul să vadă funcționalitățile pe care le are la dispoziție printr-un meniu de navigare aflat în antetul paginii. De asemenea, componente precum butoane, formulare, fonturi trebuie să fie consistente în toate paginile website-ului. Tema website-ului o să fie roșu, adică în fiecare pagină culoarea dominantă va fi aceasta.

R-ul modelului este abrevierea cuvântului Fiabilitate (Realiability în limba engleză) și acesta se referă frecvența erorilor și timpii de recuperare în cazul unei erori majore și severitatea erorilor. Cu siguranță că îmi doresc un produs final fiabil care să aibă o frecvență cât mai mică a erorilor și timpi de recuperare rapizi, însă consider că până în momentul în care websiteul ar fi ridicat pe un server și aș putea testa diferite lucruri, fiabilitatea este o cerință funcțională pe care nu o pot introduce printre cerințele non-funcționale deoarece mă aflu în imposibilitatea de a o testa.

Ulterior prin litera P ne referim la Performanță, și anume eficiență, timpi de răspuns sau resurse consumate. În acest caz, deoarece majoritatea funcționalităților necesită acces la baza de date, cerințele non funcționale de la acest capitol se vor îndrepta către timpii de acces la baza de date, respectiv de răspuns ai aplicației. Pentru a stabili o metrică în acest sens, atenția mea cade asupra unui studiu realizat în 2020, denumit "A Comparison of Database Drivers for MySQL" (https://www.cdata.com/kb/articles/mysql-comparison-2020.rst, 2022)având ca scop compararea diferitor conectori către o bază de date în MySQL, și anume CData JDBC Driver și MySQL Connector/J. Testul s-a făcut pe o operație de SELECT asupra unei tabele care conține 15 coloane cu tipuri de date diferite și pe un număr de aproximativ 10 000 000 de înregistrări, folosind C++ ca intermediar. Timpii operației de SELECT au fost între 177 și 306 secunde pentru 10 milioane de înregistrări. Testul a fost repetat pentru o operație de INSERT, însă a 1 milion de înregistrării și timpii au fost între 77 și 129 de secunde. Așadar, deoarece aplicația mea nu va avea momentan cel puțin atât de multe înregistrări în niciun tabel, și chiar dacă ar exista, în momentul în care avem un număr mare de date apelăm la conceptul de "Lazy Fetch" și extragem doar câte 50-100 de înregistrări pe rând, iar probabilitatea ca 1 milion de operații de INSERT să se petreacă simultan este destul de mica, am considerat că nicio operație care necesită acces la baza de date nu ar trebui să necesite un timp de răspuns mai mare de 3 secunde.

În final, ultima clasificare a modelului se referă la Suportabilitate, mai specific criterii de mentenanță, testare, portabilitate și extensibilitate. În acest sens, doresc ca aplicația mea să fie una ușor extensibilă, deoarece există posibilitatea adăugării unor noi funcționalități în viitor, așadar aceasta trebuie să fie bine modularizată pentru a fi ușor extensibilă pe viitor. De asemenea, în termini de testabilitate, aplicația trebuie să fie una ușor de testat în diferite scenarii, iar această testare să fie posibil de realizat atât din backend, cât și din frontend pentru a putea localiza eficient problemele.

Aspecte legate de îndeplinirea acestor cerințe vor fi detaliate în Capitolul patru, cel care face referință la asigurarea calității și vom vedea dacă acestea au fost îndeplinite.

2.2 Arhitectura Sistemului

2.2.1 Client-Server

Pentru arhitectura unui sistem care va fi un website deși poate fi flexibilă alegerea, una dintre arhitecturi pare a fi cea mai potrivită pentru o astfel de aplicație, mai ales ținând cont de faptul că anterior acestui paragraf am folosit termeni precum frontend și backend.

Pentru sistemul întreg, am ales o arhitectură de tipul Client-Server.(conform java.sun.com, 2022) În acest tip de arhitectură, actorul Client solicită resurse sau servicii, în timp ce actorul Server le distribuie, toate acestea reușind să se concretizeze datorită transmisiunii de date prin protocolul TCP, deoarece acesta ne asigură integritatea transmisiunii. Mai exact, ne asigură că toate datele au fost transmise la client, fiind transmise exact în ordinea pe care a stabilit-o serverul. Transmisiunea printr-un protocol TCP înseamnă așteptarea unui semnal de Acknowledge pentru a marca sfârșitul transmisiunii, așadar putem ști cu siguranță că datele nu se pierd.

În cazul website-ului meu, partea de Client o reprezintă Fronendul construit în React. Nu există niciun fel de acces la baza de date direct din React, iar toate cererile pe care utilizatorii le fac prin intermediul interfeței nu sunt procesate acolo. Ele sunt pasate mai departe către Server. În cazul meu, parte de server o reprezintă aplicația din Backend, Java-Spring. Aceasta mediază accesul la baza de date, procesează datele primite, efectuează diferite operații asupra datelor primite, adică oferă niște servicii și trimite mai departe un răspuns către client.

Pentru a formaliza și abstractiza schimbul de date cât mai mult, în partea de server sunt implementate corespunzător fiecărei operații diferite Web APIs (application programming interface), pentru a restricționa comunicarea la un singur format de date, deoarece în acest fel facilităm și parsarea acestora. Făcând acest lucru în partea de server, facilităm portabilitatea, deoarece în cazul în care dorim să schimbăm tehnologia de client acest lucru nu influențează partea de server pe care o putem refolosi ulterior.

Prin intermediul Web APIs, clientul trimite o cerere (request) în format HTTP (Hypertext Transfer Protocol) către server, iar acesta îi trimite înapoi un mesaj răspuns (response) de obicei în format JSON (JavaScript Object Notation). Acesta este un format al datelor independent de limbaj, cu o structura ierarhică și intuitiv ca aspect. Majoritatea tehnologiilor de frontend, inclusiv React, au funcții proprii de parsare a unui răspuns în format JSON, cât și funcții pentru convertirea datelor înapoi în acest format.

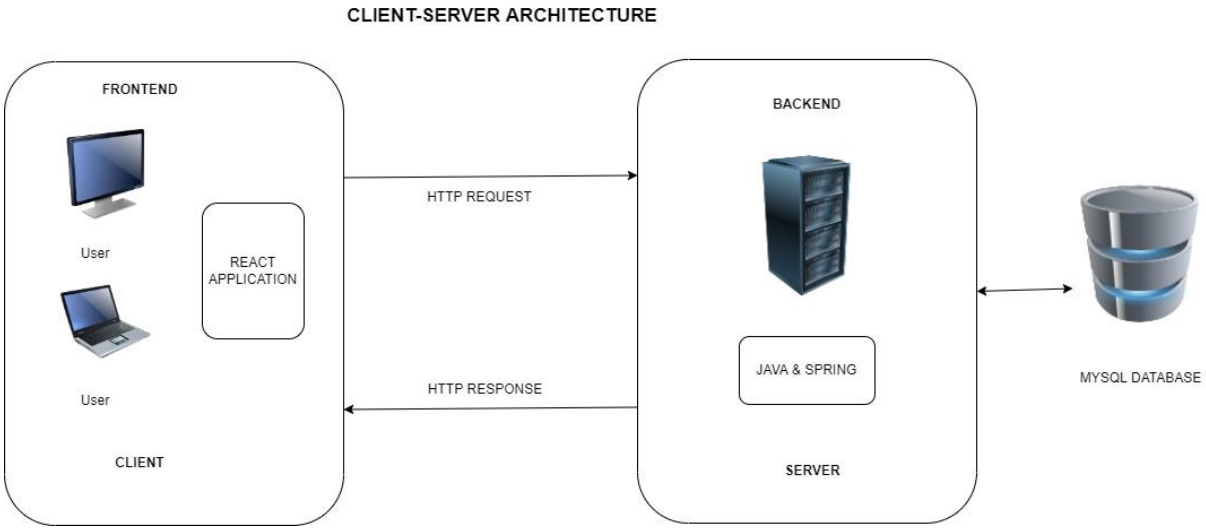


Figura 6 - Arhitectura Client-Server a sistemului

2.2.2 Arhitectura Serverului / Backendului - Layers

Deși am ales o arhitectură pentru întregul sistem, este natural că o arhitectură este necesară și pentru partea de server deoarece aceasta este mai complexă din punct de vedere al funcționalității, oferind servicii. Identificarea unei arhitecturi potrivite este un lucru esențial, deoarece în acest fel putem izola mai ușor eventuale probleme care apar, putem modulariza mult mai bine, iar codul scris este mult mai clar pentru programator, lizibil și ușor de înțeles.

Datorită faptului că lucrăm cu o bază de date, toate funcționalitățile incluzând accesul la acesta, că ulterior sau anterior anumite validări sau prelucrări sunt făcute asupra acestor date și datorită faptului că dorim să creăm anumite APIs prin care să accesăm serviciile, am considerat că cel mai potrivit este să abordez o arhitectură de tip Layers.

O arhitectură de tip Layers (conform Richards Mark,2015; Sommerville Ian, 2011)) generală vom vedea mai jos, pentru a putea discuta responsabilitățile fiecărui layer și modul în care este gândită ea în general, iar apoi voi avea o figură cu propria arhitectură pe nivele pe care am implementat-o.

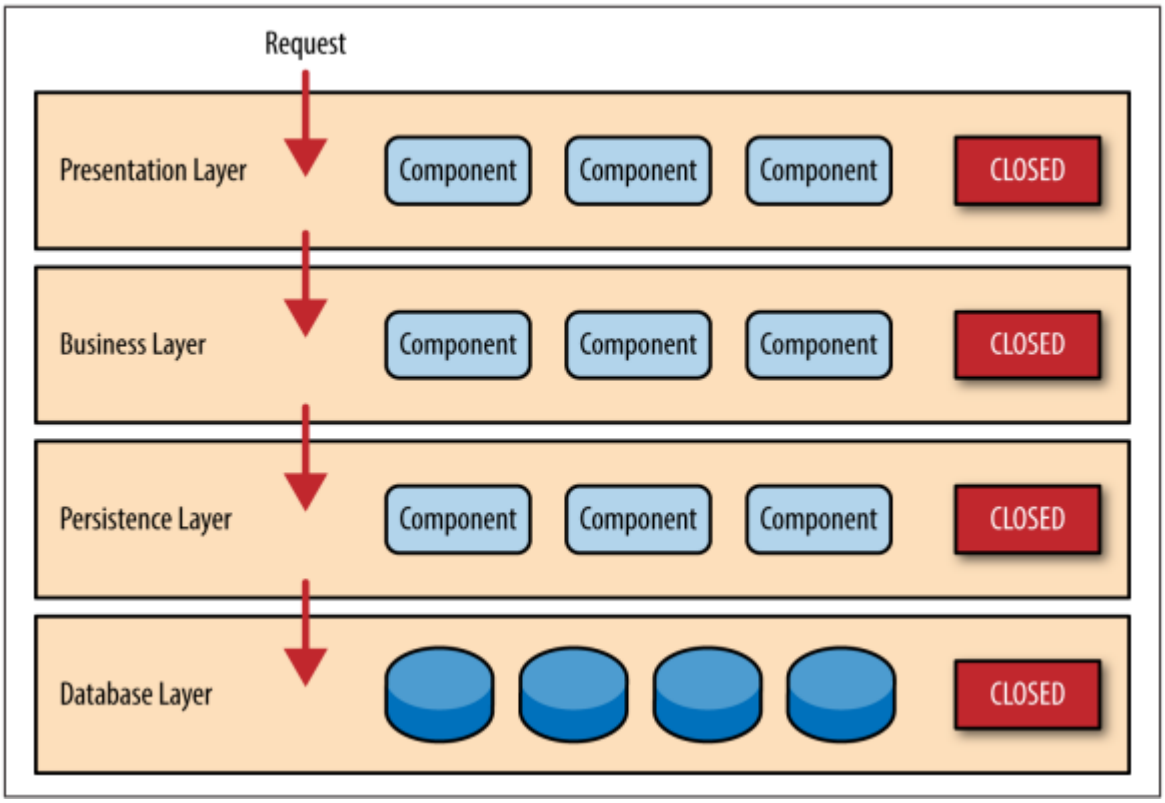


Figura 7 - Arhitectură Layered Generală

Ceea ce consider a fi cel mai benefic la acest tip de arhitectură este că aplicația este separată pe module, iar fiecare modul este răspunzător de un anumit lucru, acest aspect fiind și în concordanță cu primul principiu SOLID, cel al unei singure responsabilități. Deoarece fiecare nivel vine cu propria lui responsabilitate, este mult mai ușor să izolezi porțiuni de cod atunci când există diferite defecțiuni sau comportamente incorecte ale aplicației. În acest fel, pentru că poți izola bine codul, sursa erorilor este detectată mult mai ușor și bineînțeles poate fi fixată mult mai ușor.

În exemplul din Figura 7 avem un exemplu de arhitectură închisă, în care comunicarea dintre niveluri se face secvențial, așadar accesul la resurse se face controlat, nivelul de prezentare neputând să acceseze direct nivelul de persistență de exemplu.

Există 4 nivele principale într-o astfel de arhitectură, după cum urmează:

Database Layer – acest nivel reprezintă clasele corespunzătoare tabelelor din baza de date. Aici sunt reprezentate ca atribute câmpurile din baza de date, iar relațiile definite în baza de date sunt declarate și în clasele de aici. Este foarte important ca orice relație sau proprietate existentă aici să fi fost definită în prealabil și în baza de date sau vice-versa. În cazul în care nu sunt definite în ambele locuri este ca și cum nu ar fi fost definite deloc.

Persistence Layer- nivelul de persistență este corespunzător claselor de repository, clase care conțin query-urile pe care le vom aplica asupra bazei de date

Business Layer – nivelul de business reprezintă clasele de service și este responsabil pentru logica aplicației și eventuale validări. Folosește metode ale claselor de repository, dar datele pot fi manipulate, filtrate sau sortate, iar datele de input pot fi verificate și eventual gestionate eventuale mesaje de eroare sau excepții.

Presentation Layer- acest nivel de prezentare conține clasele de controller, și anume acele clase în care vom defini APIs pe care le vom folosi ulterior în partea de client. De asemenea, apelează mai departe metode ale claselor de service și poate face mici operații de procesare a datelor, dar în mare parte nu conține logică.

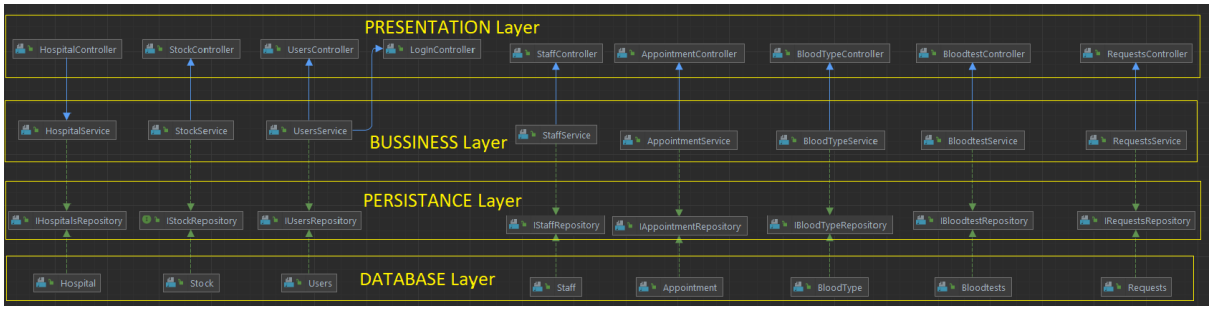


Figura 8 - Arhitectura Layered a sistemului în detaliu

După cum se poate observa, și arhitectura aplicației mele este una pe nivele, exceptând clasele de utilități pentru Securitate și logging pe care le consider separate, deoarece nu se încadrează într-un anumit nivel.

Denumirile claselor din nivelul de database coincid cu numele tabelelor din baza de date și reprezintă doar o transpunere a tabelelor în obiecte. Urmează nivelul de repository, unde există doar interfețe deoarece acestea extind JPA Repository. Ulterior urmează nivelul de business și presentation. Se poate observa cum comunicarea există doar între niveluri adiacente.

În cele ce urmează, pentru a intra în detaliile referitoare la proiectarea sistemului, voi folosi diferite diagrame realizate după standardul UML, consultat în cartea OMG Unified Modeling Language versiunea 2.4.1.

2.3 Diagrama de pachete

Sub-capitolul anterior am prezentat arhitectura sistemului și arhitectura părții de backend deoarece structura pachetelor și organizarea claselor din pachete a fost în strânsă legătură cu arhitectura pe care am ales-o. Așadar, voi avea câte un pachet corespunzător

fiecărui nivel din arhitectură. Pe lângă acestea, mai există 3 pachete de utilități care mi-au fost necesare.

Pachetul de securitate, deoarece este un aspect care nu are legătură cu un nivel anume, ci cu toate, deci încadrarea lui nu era acolo. De asemenea, pentru motive de înregistrare și depanare a aplicației, mai există un pachet care se ocupă cu urmărirea și păstrarea tuturor cererilor HTTP care sunt trimise către server. Nu în ultimul rând, mai există un pachet de roles, în care după cum îi spune și numele am definit câteva adnotări corespunzătoare cu tipurile de utilizatori, și anume doctor, admin și donator.

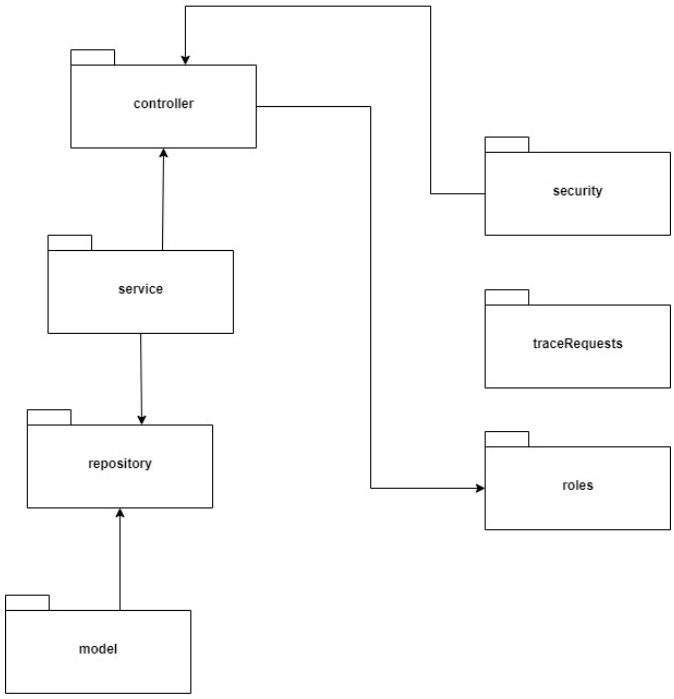


Figura 9 - Diagrama de pachete

Așadar, după cum se vede și din diagramă, pachetele corespunzătoare nivelurilor arhitecturii poartă un nume sugestiv, ele îndeplinind funcționalitatea specific nivelului al cărui nume îl poarta. Pachetul de Securitate conține clasele specifice și interacționează cu pachetul de controller, la fel și pachetul de roles, deoarece în controller este stability anterior fiecărei metode dacă este necesară sau nu o autoritate.

De asemenea, pachetele au fost grupate urmând principiile: high cohesion, low coupling. Primul dintre ele se referă la faptul că elementele similare ale unui modul ar trebui să fie grupate împreună, însă modulele între ele ar trebui să fie cât mai independente posibil unele de celelalte.

2.4 Diagrama bazei de date

Pentru proiectarea bazei de date am încercat să izolez cât mai mult tabelele astfel încât transpunerea lor în obiecte Java să fie cât mai intuitivă și să clasele create să respecte principiul singurei responsabilități.

După cum putem observa există 8 tabele corespunzătoare unor entități din lumea reală: hospitals- Spitale, users-donatori, bloodtype -grupa de sânge și rh-ul, stocks – stocul de sânge al spitalului, requests – apeluri la donare, appointments – programări, bloodtests – rezultatele analizelor și Staff- personalul medical.

Nu avem niciun tabel intermediar, deoarece am evitat relația Many-To-Many.

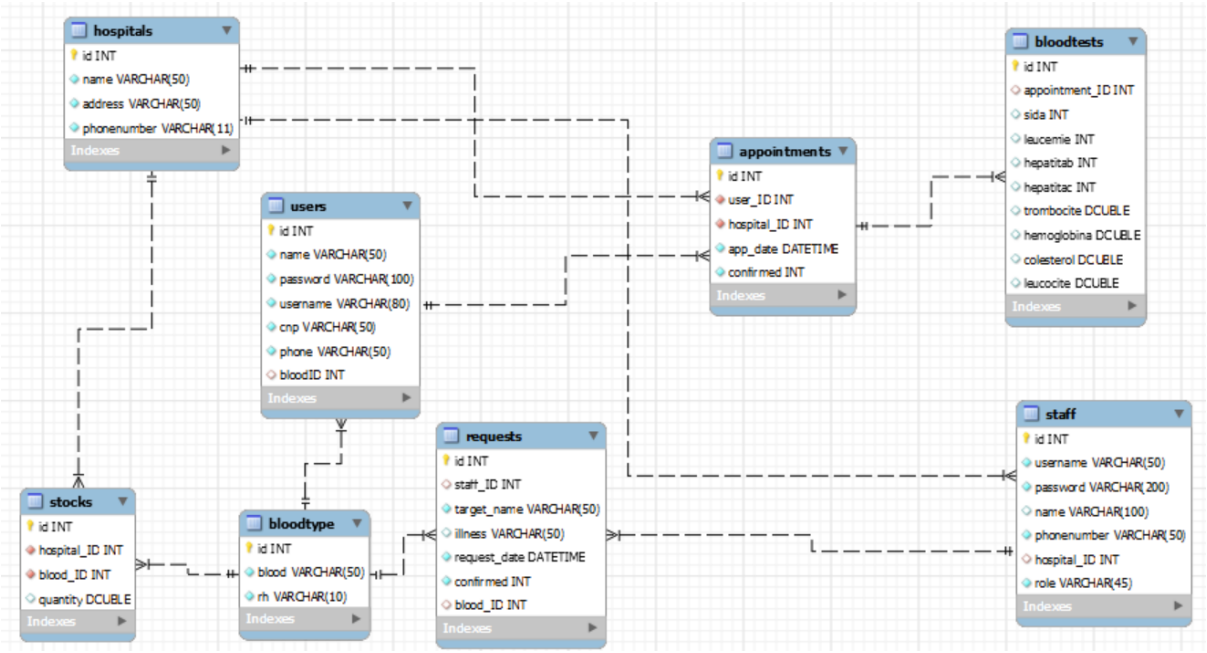


Figura 10 - Diagrama bazei de date

Între tabela bloodtype și stocks, este o relație de One-To-Many, deoarece o anumită grupă de sânge apare în mai multe înregistrări de stocuri, depinde de spital, însă o înregistrare de stoc are o singură grupă de sânge.

Între bloodtype și user avem o relațe de One-To-Many. Acest lucru este astfel deoarece un donator are o singură grupă sanguină, dar o grupă sangvină poate aparține mai multor donatori. Similar avem aceeași relație între bloodtype și requests (apelurile la donare).

De asemenea, între stocks si hospitals este o relație de One-To-Many, deoarece un spital poate avea mai multe stocuri, pentru că fiecare stoc corespunde unei grupe sanguine, însă un anumit stoc aparține doar unui spital.

Tot între hospital și Staff / Appointments (programări) există o relație de One-To-Many, deoarece într-un spital avem mai multe persoane ca fiind personal medical și mai multe programări realizate, dar o anumită programare este corespunzătoare unui singur spital, identic și în cadrul personalului medical. Între users (donatori) și appointment avem același tip de relație din același motiv.

Între appointment și bloodtest (teste de sânge) există o relație de One-To-One, deoarece doar programările confirmate au doar o analiză de sânge, iar o analiză aparține unei singure programări.

Între staff și requests (apeluri la donare), există o relație de One-To-Many, deoarece un doctor poate face mai multe apeluri la donare, iar un apel la donare aparține unui singur doctor.

2.5 Diagrama de clase

În momentul în care am ales o arhitectură de tip layers pentru organizarea backendului, practic pe fiecare nivel avem aceleași corespondențe între clase deoarece sunt reprezentantele aceluiași obiect, doar că pe niveluri diferite și cu altfel de responsabilități. Din diagrama arhitecturală putem observa cum comunică între ele în diferite nivele, așadar pentru simplitate și o mai bună înțelegere voi considera în diagrama de clase doar pachetul models pentru a evidenția cum comunică între ele clasele care se află pe același nivel.

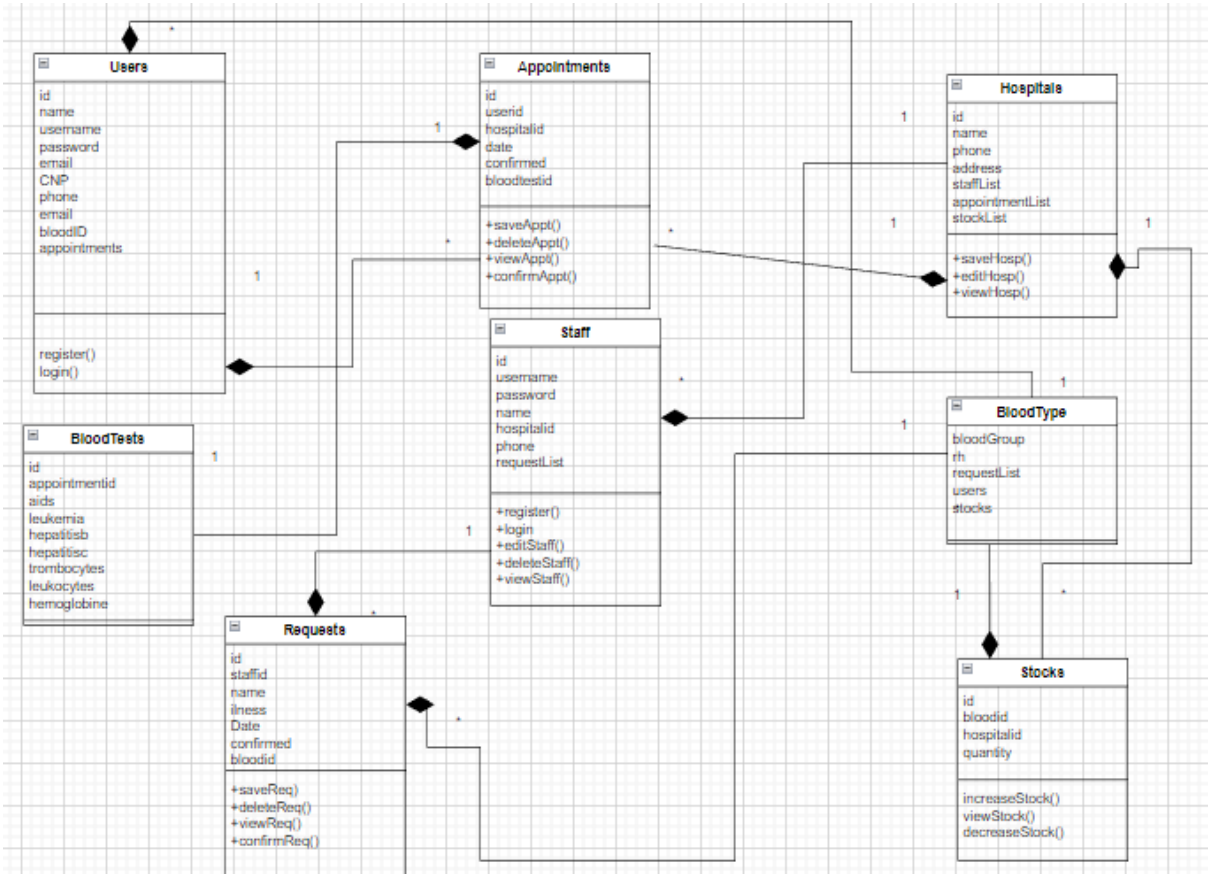


Figura 11 - Diagrama de clase

După cum putem observa, cam toate tabelele din baza de data au câte un corespondent în diagrama de clase. De asemenea și câmpurile se regăsesc aici ca atribute. Faptul că avem diferite relații se poate observa atât din săgețile care exemplifică relația de compoziție dintre clase, cât și datorită anumitor atribute. De exemplu în clasa de stocks, vom avea interacțiune cu clasa bloodtype, deci vom avea un obiect de acel tip, semnalizat prin câmpul de bloodid. De asemenea, în clasa de bloodtype avem o lista de stocuri, semnalizată prin câmpul stocks.

Prezența acestor atribute în diferite clase ne ajută să intuim relația dintre ele și să regăsim relațiile dintre entitățile din baza de date prezente și aici între clase.

2.6 Diagrama de activitate

Diagrama de activități reprezintă acțiunile pe care le poate face utilizatorul în interacțiunile lui cu sistemul, și față de diagrama de stări, accentul este pus pe acțiunile utilizatorului și nu pe stările în care aceste acțiuni îl duc.

Prima diagramă de activitate este pentru utilizatorul donator.

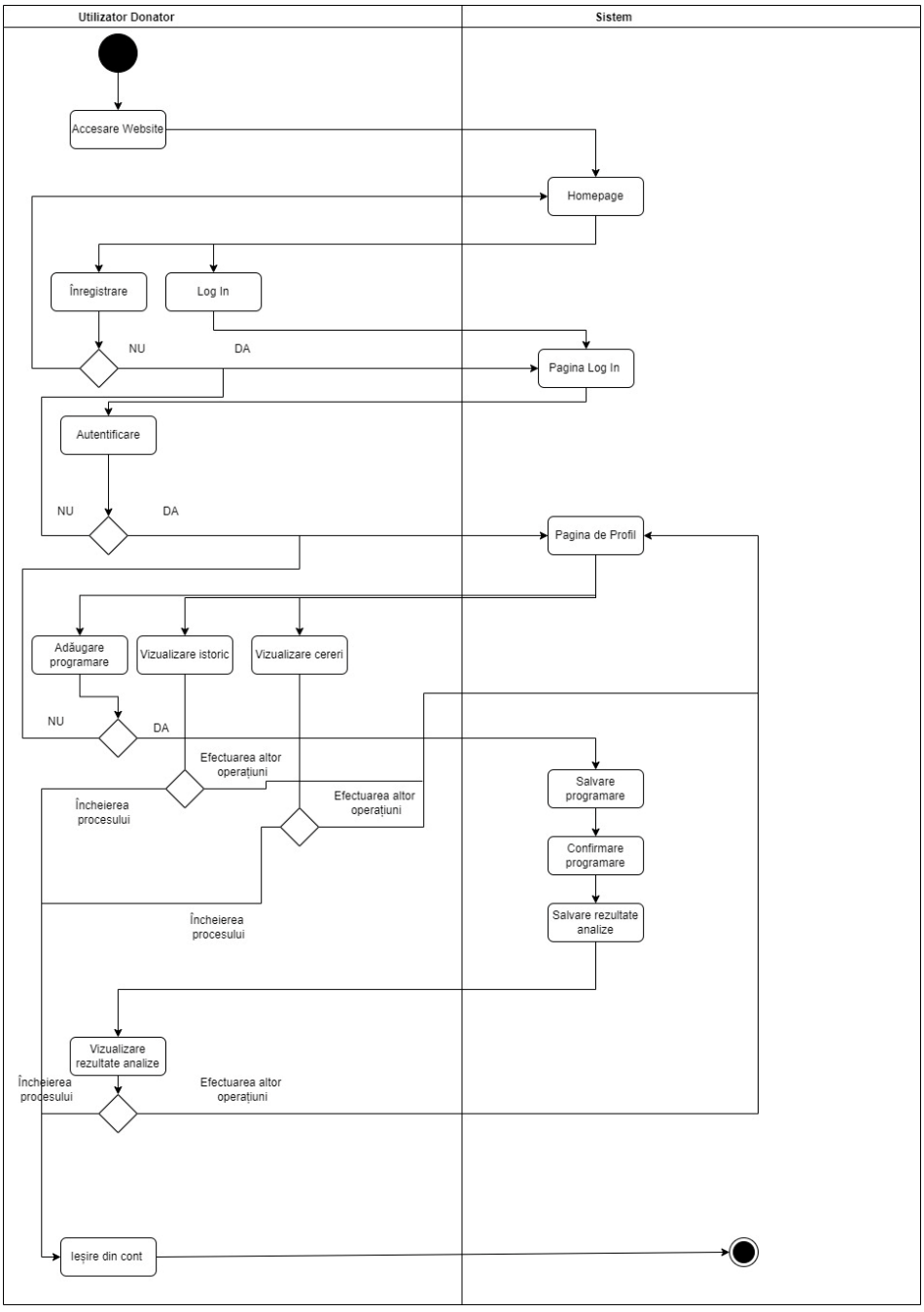


Figura 12 - Diagrama de activități pentru donator

La o primă interacțiunea, activitatea pe care o face donatorul este să acceseze website-ul. În acel moment, sistemul îi afișează Pagina Principală. În momentul când este pe pagina principală, utilizatorul are 2 alternative de activități. Prima dintre ele este să se înregistreze.

Dace alege să nu o facă, atunci va rămâne pe pagina principală. Dacă va alege să o facă, o dată ce aceasta este completă, sistemul îl redirecționează către pagina de log-in.

Sau, dacă are deja un cont, poate alege să dea click pe butonul respectiv. Dacă alege să nu o facă, rămâne pe aceeași pagină, dacă alege să o facă, va fi redirecționat pe pagina de Log In. O dată ajuns pe pagina de log in, actorul poate alege să se autentifice sau nu. Dacă alege să nu o facă, ajunge din nou pe Pagina Principală. Dacă alege să o facă, va fi redirecționat către pagina de profil. De acolo, poate alege să adauge o programare, să vizualizeze istoricul sau apelurile la donare. În cazul în care alege una dintre cele două din urmă, ulterior poate face alte operațiuni din pagina de profil sau dacă își va dori încheierea procesului, deci ieșirea din cont și încheierea sesiunii.

Dacă alege să o facă, sistemul va salva programarea, o va confirma și apoi va salva rezultatele analizelor. Ulterior, utilizatorul îți poate vedea rezultatul analizelor. Dacă apoi utilizatorul își va dori să efectueze alte operațiuni, va fi redirecționat către pagina de profil. Dacă își va dori încheierea procesului, va ieși din cont și sesiunea se sfârșește.

Diagrama de activități a administratorului (Figura 13) inițial este similar cu cea a donatorului, deoarece sunt funcționalități comune. Ulterior acesta are de ales dintre 3 activități posibile, și anume Editarea spitalului, Schimbarea parolei sau Adăugarea Medicilor. Dacă nu o alege pe niciuna dintre ele, atunci va rămâne pe pagina de profil. Însă dacă va face una dintre aceste activități începe interacțiunea cu sistemul și informațiile noi vor fi salvate sau respective validate și ulterior salvate. În final, utilizatorul poate vizualiza noile informații. Dacă dorește să efectueze alte operațiuni poate fi redirecționat către pagina de profil, iar dacă nu procesul este încheiat, va ieși din cont, iar această sesiune va fi încheiată.

De asemenea, există și o diagramă de activități specifică personalului medical. (Figura 14). Această diagramă are și ea anumite particularități , însă în principiu seamănă foarte mult cu cea a administratorului deoarece atât partea de accesare a contului cât și activitatea de schimbare a parolei sunt similare, în consecință reprezentate la fel.

Celelalte activități, cele de Editare a stocului, Adăugare a apelurilor și Introducere a analizelor, se comportă similar. Dacă utilizatorul alege să le facă, informațiile noi sunt salvate și apoi utilizatorul le poate vizualiza. Dacă alege să nu le facă, este redirecționat în pagina de profil.

Dacă acesta vrea să execute alte activități poate alege să o facă, iar dacă nu, poate alege ca procesele să se încheie, să iasă din cont, iar astfel sesiunea să se încheie.

Toate cele 3 diagrame demonstrează parcursul actorilor din cadrul aplicației, activitățile specifice acestora, interacțiunea lor cu sistemul, cât și diferențele între activitățile pe care le poate executa fiecare.

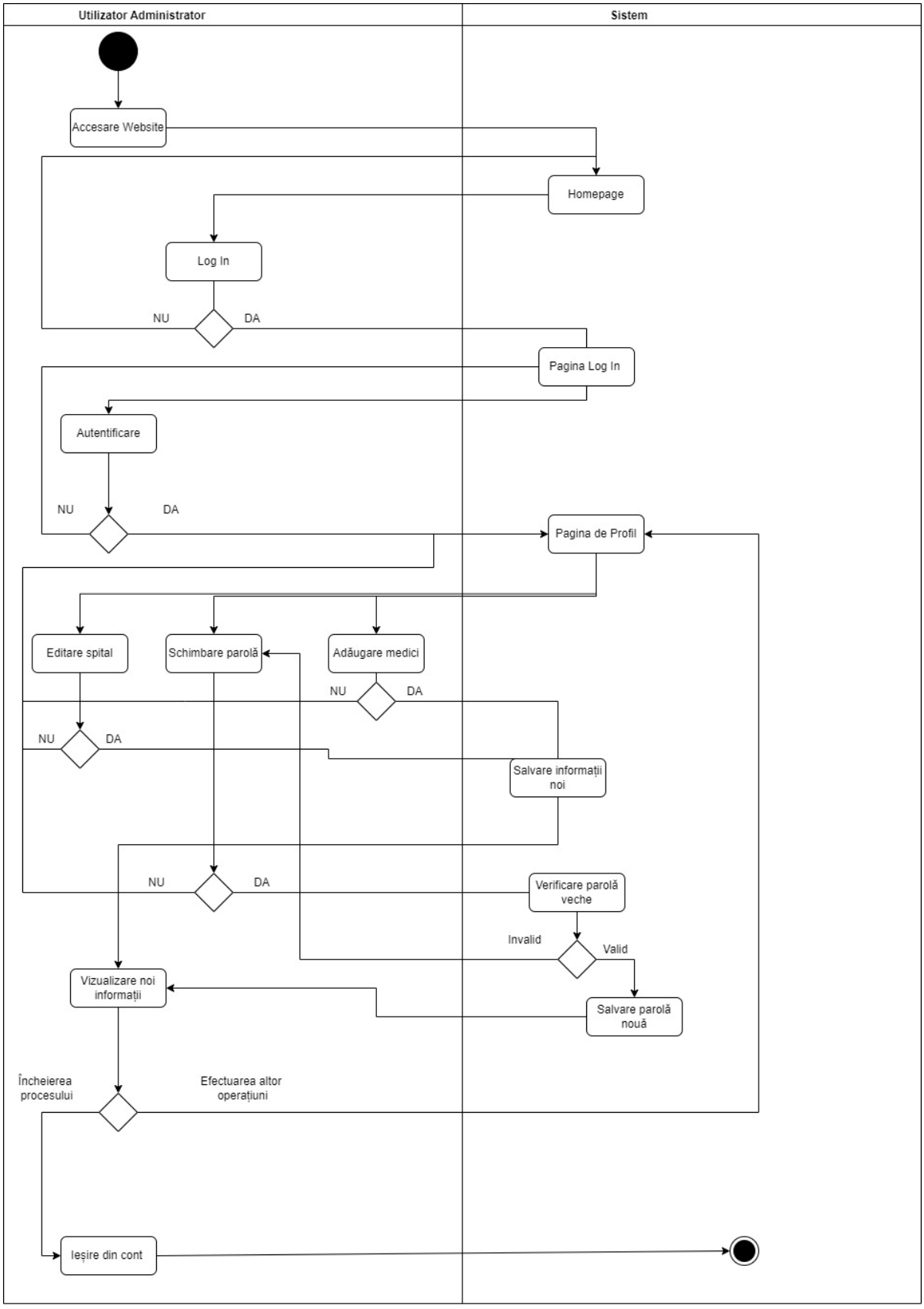


Figura 13 - Diagrama de activități pentru administrator

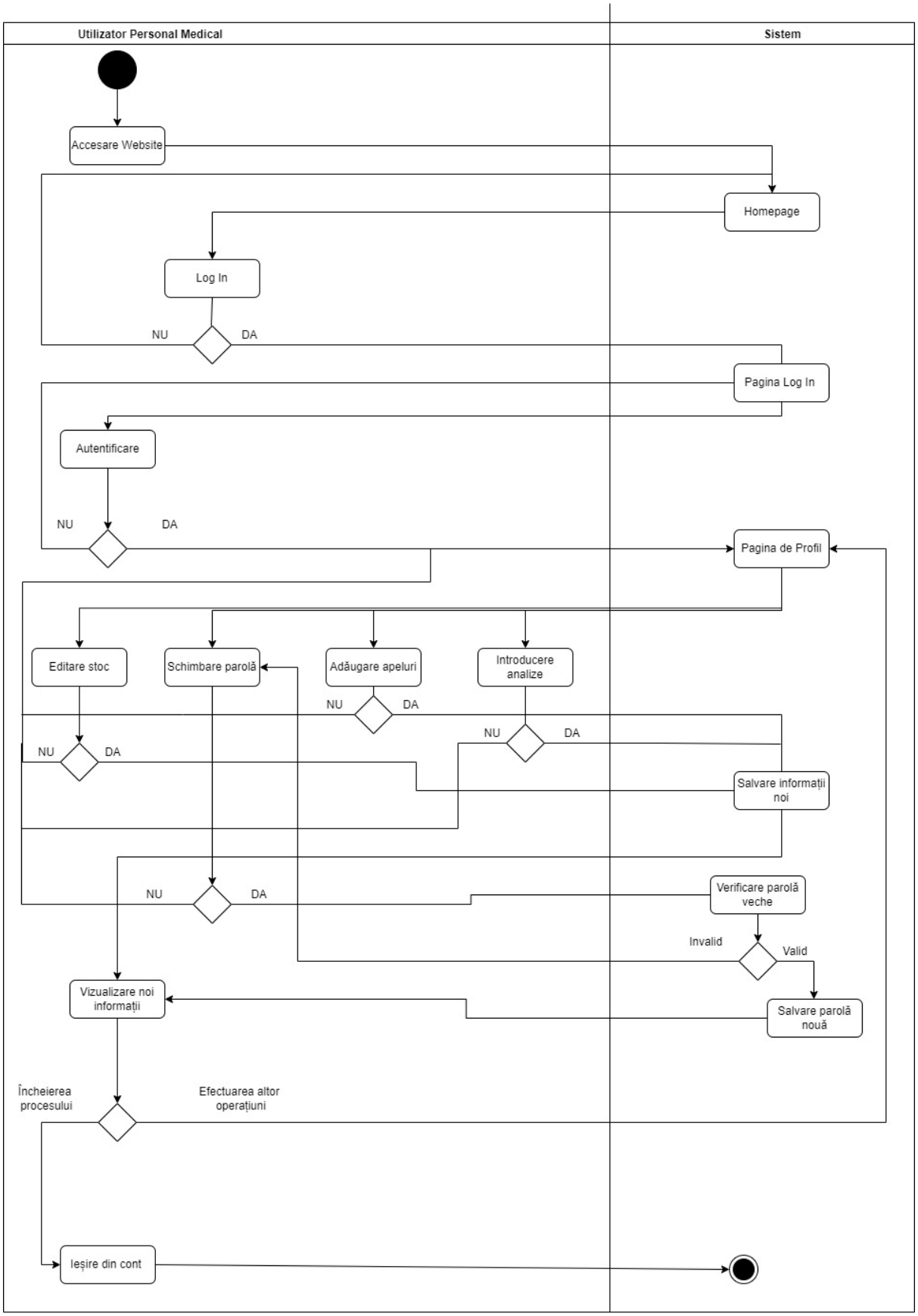


Figura 14 - Diagrama de activități pentru personalul medical

2.7 Diagrama de secvență

O diagramă de secvență, după cum și sugerează numele acesteia, ne arată ordinea în care diferite componente ale aplicației interacționează între ele în cadrul unui anumit proces în timp.

Pe verticală avem cu linie punctată verticală linia vieți obiectului respectiv, având marcate cu dreptunghiuri subțiri durata în care execuția este în subordinea acelui obiect. Marcate cu o săgeată plină avem comunicările dintre componente, de exemplu apelul unor funcții. Cu săgeată punctată avem marcată întoarcerea dintr-o funcție. Diagramă de secvență surprinde ordinea evenimentelor din momentul în care se face o cerere către server, până în momentul în care clientul primește răspunsul.

În cele ce urmează, am exemplificat câteva diagrame de secvență pentru procesele și actorii din cadrul aplicației mele.

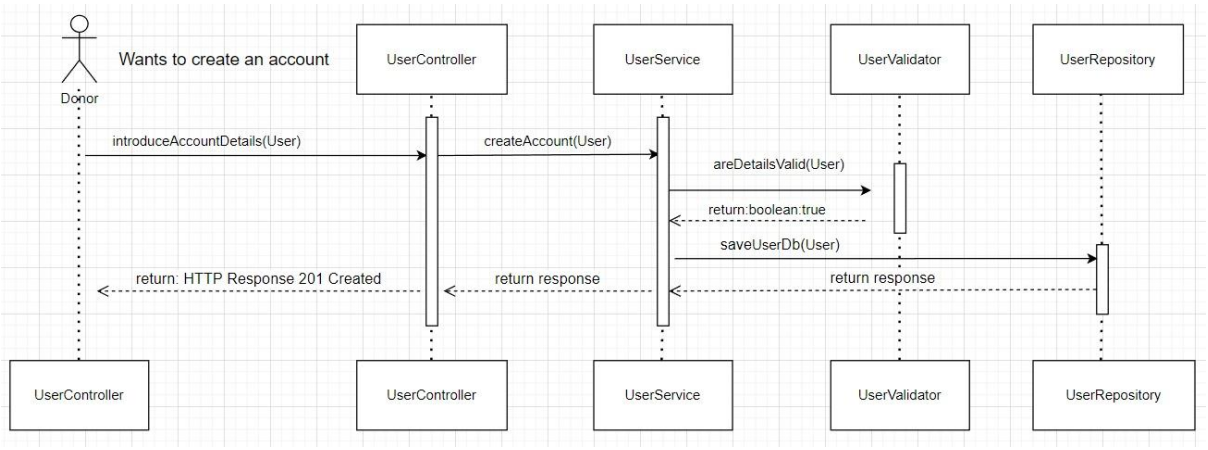


Figura 15 - Diagramă de secvență pentru crearea unui cont - caz succes

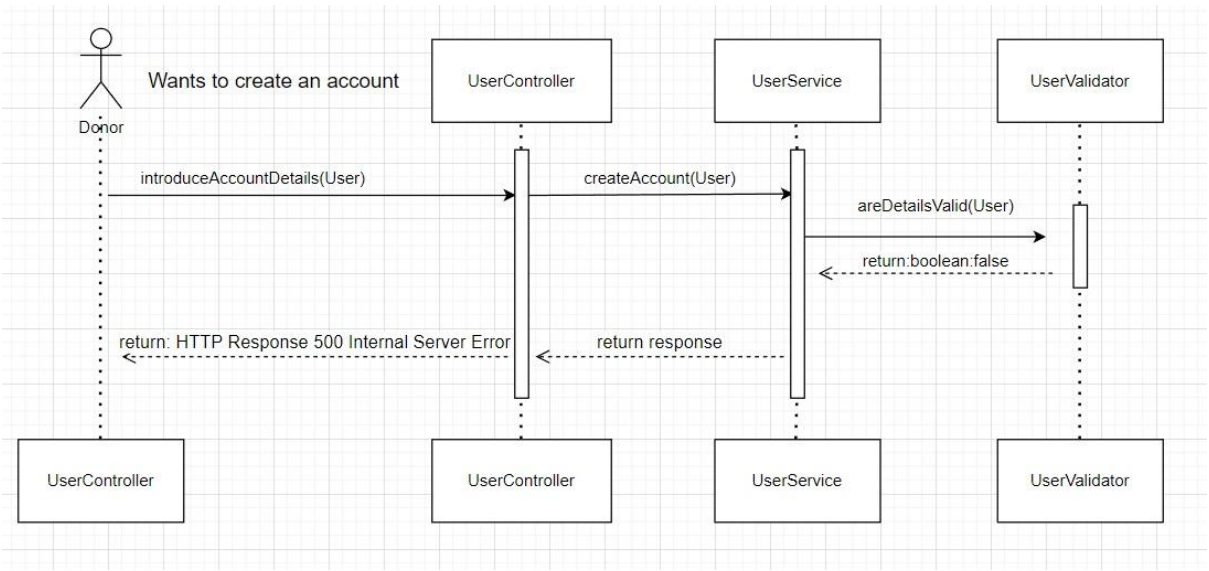


Figura 16 - Diagramă de secvență pentru crearea unui cont - caz eșec

Figurile 15 și 16 reprezintă o diagramă de secvență a aceluiași proces, însă ceea ce este diferit este rezultatul procesului. În cazul figurii 15 crearea contului a fost realizată cu succes, însă în cazul figurii 16 acest lucru nu a fost posibil.

După cum putem observa, ambele procese încep asemănător, cu Donatorul care trimite o cerere către serverul web. Această cerere este preluată de către clasa specifică de controller și este apelată mai departe metoda din service care se ocupă cu crearea unui cont. Ulterior, datele trec mai departe printr-un validator, care ne indică posibilitatea de a crea un cont cu datele trimise. În cazul figurii 15, acest validator returnează adevărat, iar datele sunt pasate mai departe către clasele de repository care se ocupă cu operația de salvare în sine. După ce aceasta a fost realizată, răspunsul 201 Created parcurge din nou în sens invers fiecare nivel din diagramă până revine la client, înștiințându-l că s-a creat contul. În cazul figurii 16, din momentul în care validatorul nu permite creare contului, un răspuns de 500 Error este trimis înapoi către client, înștiințându-l că nu s-a putut crea contul.

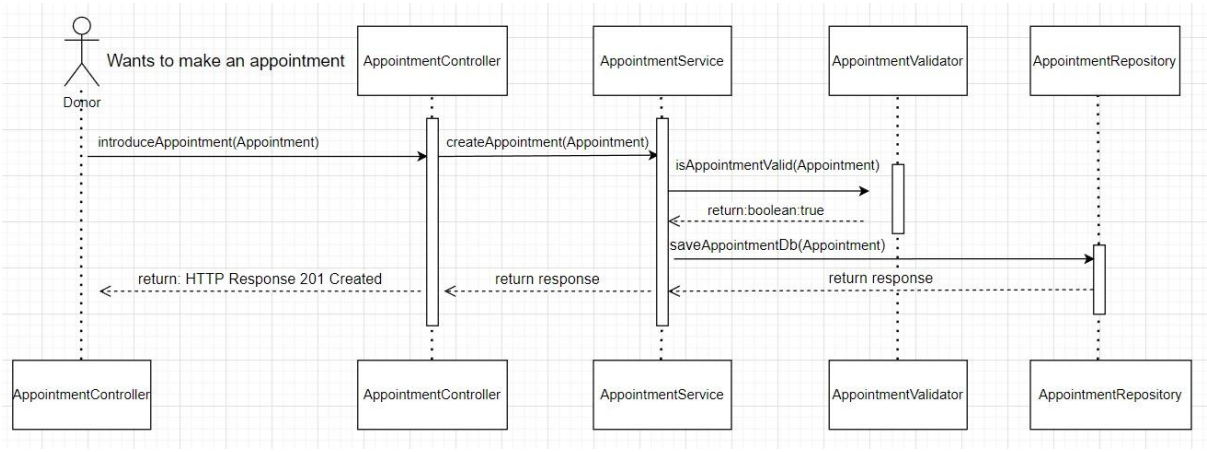


Figura 17 - Diagramă de secvență pentru crearea unei programări

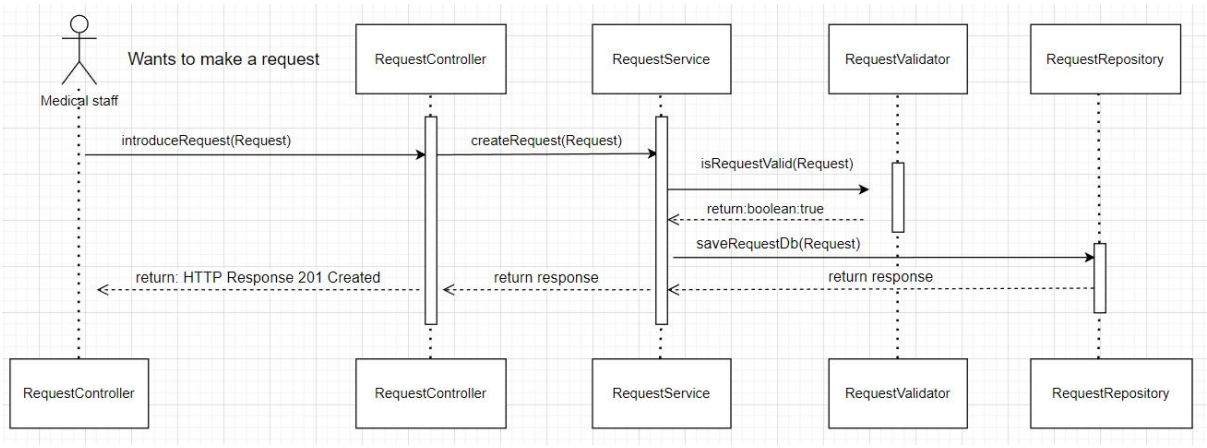


Figura 18 - Diagramă de secvență pentru crearea unui apel la donare

Diagramele de secvență pentru celelalte procese sunt similare, deoarece folosind arhitectura layered, clasele corespunzătoare unui proces au la bază obiectul principal al procesului și sunt implementate în aceeași manieră. Figurile 17 și 18 exemplifică un proces de adăugare a unei programări, respectiv al unui apel la donare. După cum putem vedea, deși

avem de-a face cu obiecte diferite, procese diferite și actori diferiți, ordinea evenimentelor este precum în procesul figurii 15, doar că avem diferite clase care preiau controlul.

2.8 Diagrama de stare

Diagrama de stare reprezintă un mod simplu prin care putem exemplifica diferitele "stadii" ale unui actor în cadrul aplicației, stadii pe care le numim stări. Pe măsură ce actorul interacționează cu aplicația și ia anumite decizii, stările prin care trece se pot schimba, depinzând de ce decizie a luat anterior.

Starea corespunde unui set de condiții care sunt satisfăcute în intervalul de timp în care sistemul se află în starea respectivă, a unei situații în care se așteaptă apariția unui alt eveniment sau efectuarea unei activități.

Activitatea este un stimul care produce tranziția între stări, o acțiune care se întâmplă la un anumit moment în timp. Diagrama conține o stare inițială, o stare finală și diferite stări intermediare, alături de acțiunile care determină evoluția stărilor.

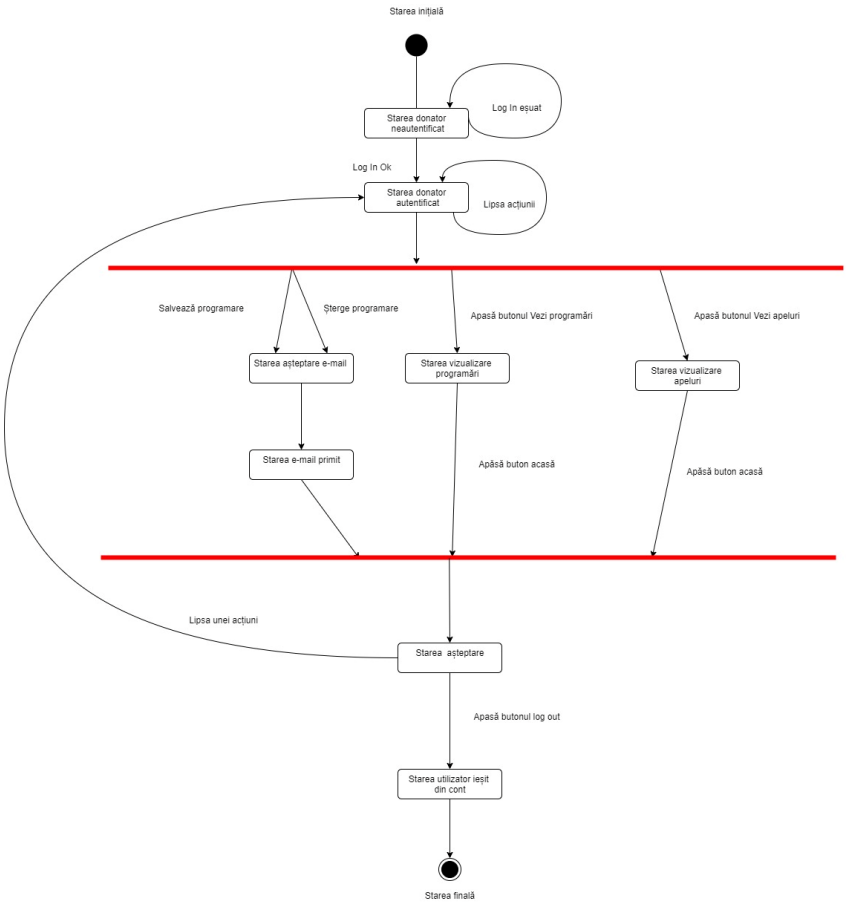


Figura 19 - Diagramă de stare donator

Diagrama din figura 19 reprezintă stările sistemului în funcție de acțiunile donatorului. Inițial pornim din starea Donator neautentificat, unde tranziția merge fie spre aceeași stare în caz că autentificarea este eșuată, fie în starea Donator autentificat în cazul unui log-in realizat cu succes.

Din această stare există 3 tranziții posibile în funcție de acțiunea pe care o va face donatorul. Dacă va apăsa butonul de vizualizare programări, tranziția va fi către starea Vizualizare programări, analog pentru starea Vizualizare apeluri.

În starea de așteptare e-mail se poate ajunge prin 2 acțiuni diferite, fie se adaugă o programare, fie se șterge. Ulterior după puțin timp, fără nicio intervenție din exterior, se trece în starea E-mail primit deoarece durează câteva secunde până se trimite e-mailul.

Din aceste stări, dacă se apasă butonul Acasă, se revine în starea de așteptare până utilizatorul dorește să facă altă acțiune. Dacă acesta apasă butonul de log out, se trece la starea utilizator ieșit din cont, fiind starea finală.

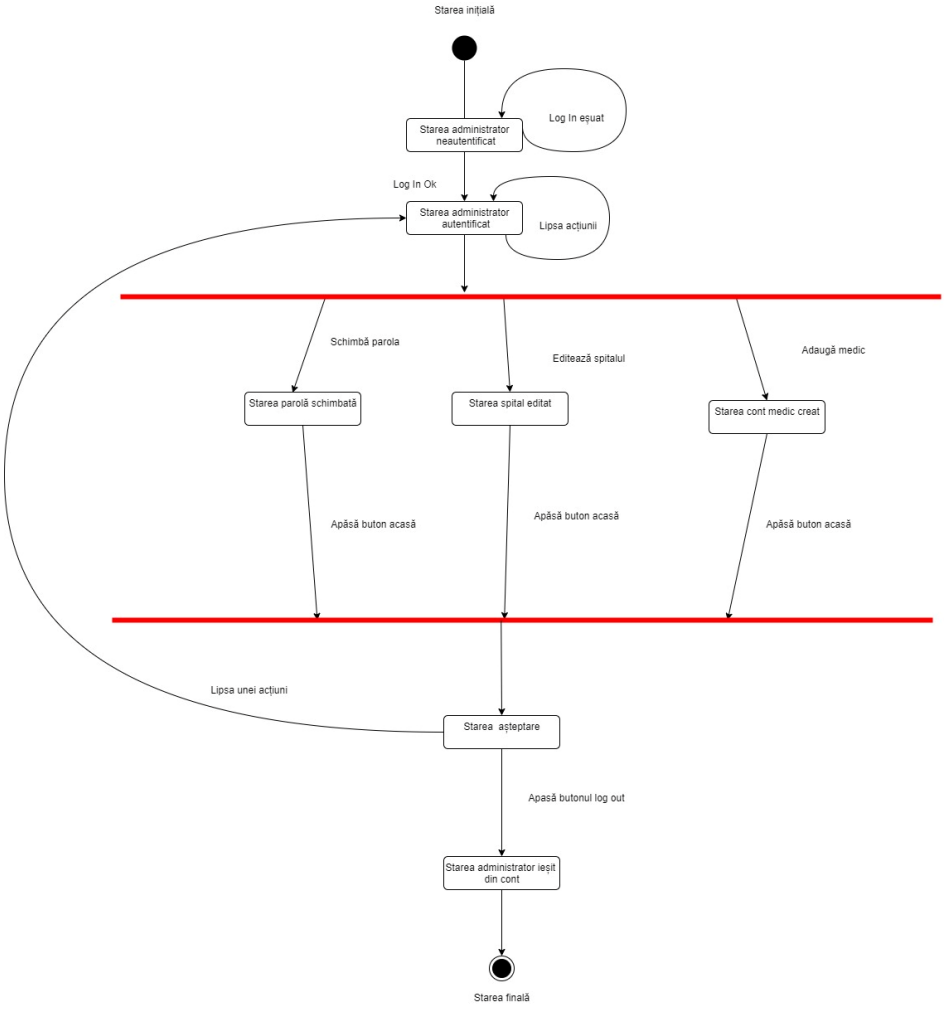


Figura 20- Diagramă de stare administrator

Avem o diagramă similară și pentru Administrator în care partea de autentificare se repetă, însă stările posibile din cea de Administrator autentificat sunt diferite. În funcție de acțiune se trece în stările Parolă schimbată, Spital editat sau Cont de medic creat. Din aceste trei stări, modul în care se trece în Starea de așteptare și ulterior în starea de Personal medical ieșit din cont sunt similare cu cele omoloage din diagrama donatorului.

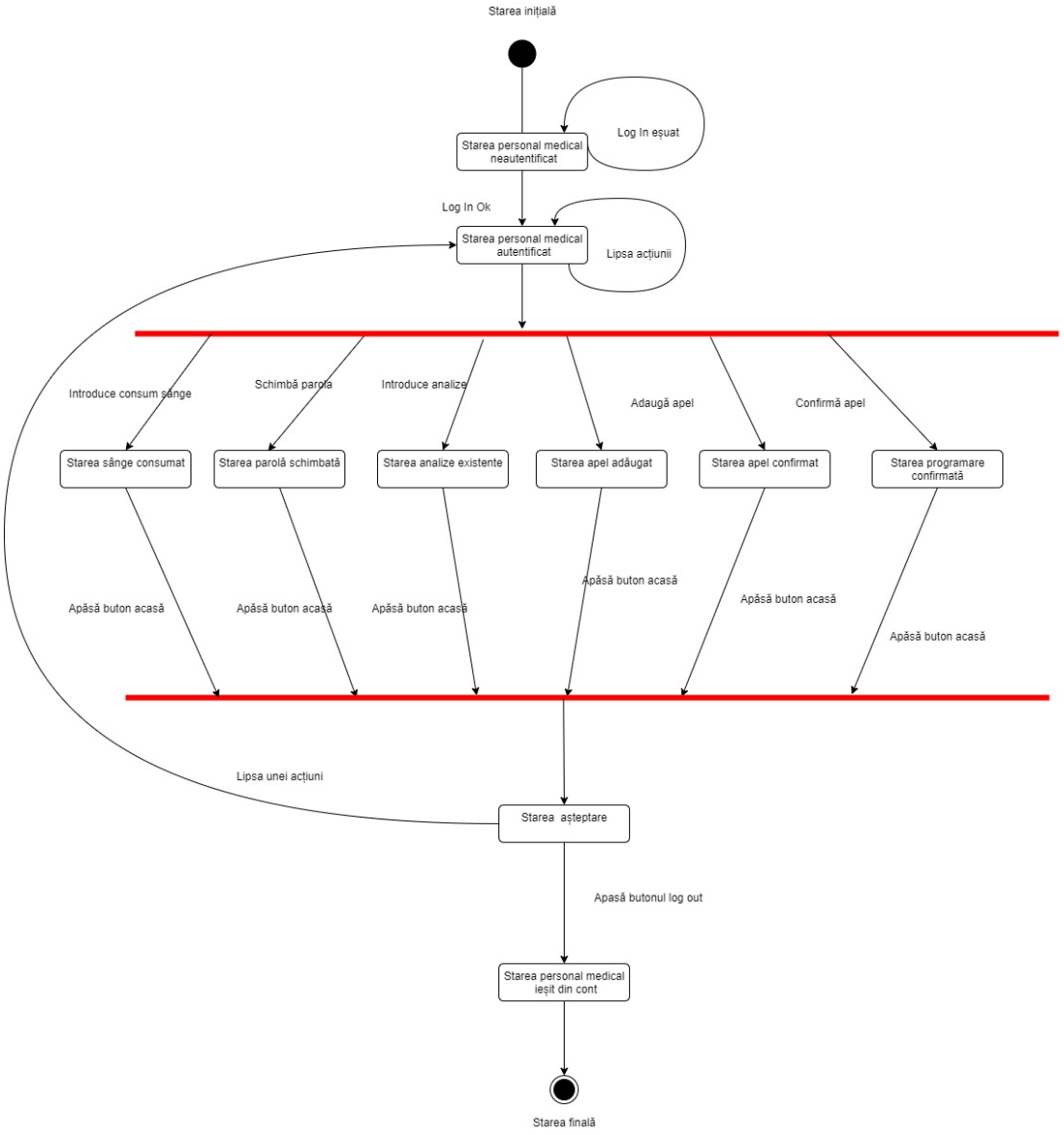


Figura 21- Diagramă de stare personal medical

Ultima diagramă de stare este cea care arată tranziția stărilor din perspectiva personalului medical. Tranzițiile inițiale care privesc operațiile de autentificare sunt similare și în cazul acestei diagrame. Tranzițiile diferite față de celelalte diagrame sunt cele din starea de Autentificat, deoarece există următoarele stări posibile: Starea Sânge consumat, Parolă schimbată, Analize existente, Apel adăugat, Apel confirmat și Programare confirmată, din nou depinzând de acțiunea pe care o face personalul medical. Din nou, tranzițiile către starea de așteptare și starea finală sunt similare cu cele din diagramele de sări precedente.

3.Implementare

3.1 Backend-Server

Acest subcapitol este împărțit în alte 4 subcapitole deoarece îmi doresc ca primele 3 dintre ele să surprindă un aspect, iar cel de-al treilea altul. Având în vedere că majoritatea funcționalităților surprind operații pe baza de date, am ales să urmărim în primele 3 subcapitole acest aspect. Implementarea operațiilor CRUD este similară între obiecte, așa că am ales să urmărim fluxul complet de proiectare al operațiilor specifice tabelului de programări. Am ales acest lucru deoarece acest tabel conține tipuri de date diferite, mai multe tipuri de relații și este cel care a avut nevoie în principiu și de cea mai complexă logică adiacentă. Dacă voi considera că există o altă metodă din cadrul unui alt obiect care merită discutată o voi introduce la finalul parcurgerii acestui flux.

Consider că subiectul de securitate este unul separat, deoarece pentru mine a fost un aspect mai dificil și diferit pe care nu l-am mai implementat anterior, dar este un subiect de importanță majoră pentru orice produs software care ajunge pe piață, așadar consider că merită o discuție separată.

3.1.1 Tipuri de date și relații – Models & Repository

În acest subcapitol voi trata modul în care s-au transpus tabelele de date în obiecte Java având ca exemplu de urmărit tabelul de Programări (appointments în baza de date). Mai specific, vom analiza structura claselor din pachetul de models și cel de repository, corespunzătoare primelor două nivele din arhitectura de tip Layers. Deși pachetul de repository nu reprezintă în sine un tip de date sau relații, am ales să îl încadrez aici deoarece aici sunt tratate de fapt query-urile corespunzătoare operațiilor de tip CRUD.

Un precondiție înainte de a începe crearea claselor este ca programatorul să permită programului accesul la baza de date. Acest lucru se realizează cu ajutorul fișierului application.proprieties, fișier în care trebuie să avem declarat întregul URL către baza de date pe care vrem să o folosim, numele de utilizator și parola pe care le folosim pentru a accesa serverul de MySQL. Fișierul de application.properties în acest punct trebuie să arate similar:

```
spring.datasource.url=jdbc:mysql://localhost:3306/bloodmanagem
ent

spring.datasource.username=root

spring.datasource.password=root
```

În primul rând, toate clasele care reprezintă date ce pot fi persistate într-o bază de date trebuie să înceapă cu adnotarea @Entity. Fiecare instanță a unei clase de acest fel reprezintă o nouă înregistrare în baza de date. Ulterior, trebuit să folosim adnotarea de tip @Table, pentru a face legătura între numele clasei și tabelul pe care aceasta îl reprezintă. Deși făcând acest lucru am putea denumi clasa într-o manieră diferită decât tabelul, o bună practică ar fi să nu facem acest lucru pentru a evita confuziile. Având în vedere că în baza de date fiecare tabel are un identificator unic, adică o cheie primară, acest lucru se regăsește și în clasa specifică

din Java, cu adnotarea @Id. Putem alege strategia de generare a id-ului, în cazul meu am ales IDENTITY, adică în fiecare tabel cheia primară va fi generată începând cu valoarea 1 și va fi auto-incrementată, comparativ cu strategia AUTO în care există un tabel adițional care menține ultimul id prezent în baza de date, iar orice înregistrare nouă din orice tabel va primi valoarea veche incrementată. Este ca și cum am avea un numărător global pentru toate tabelele.

Pentru atributele următoare care reprezintă coloanele din baza de date avem adnotarea @Column. Dacă numele coloanei din baza de date diferă de numele atributului pe care dorim să îl folosim, trebuie să precizăm acest lucru în dreptul acestei adnotări. Pe lângă această posibilitate, în cazul în care dorim anumit proprietăți speciale pentru acea coloană, ele trebuie exprimate tot aici: de exemplu nullable=false dacă nu acceptăm inserarea unui câmp nul sau unique=true dacă dorim ca acel câmp să nu accepte duplicate. Toate atributele sunt declarate private pentru a nu le putea accesa decât prin intermediul metodelor de get și set. Tipul de date Date este corespondentul tipului de date Date din MySQL, String corespondentul lui Varchar, Integer corespondentul numerelor întregi, Int sau a datelor de tip Tinyint, care reprezintă valorile 0 sau 1.

Pentru tipul de date de tip Date, trebuie să specificăm printr-o adnotare modul mai exact în care dorim să fie ținutele datele în baza de date, și anume în cazul meu adnotarea respectivă de tip TIMESTAMP este echivalentul lui Datetime, adică specifică faptul că am nevoie să rețin atât data cât și timpul.

```
@Entity
@Table(name = "appointments")
public class Appointment {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(nullable = false)
    @Temporal(TemporalType.TIMESTAMP)
    private Date app_date;

    @Column(nullable = false)
    private Integer confirmed;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "hospital_ID")
    private Hospital hospital;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "user_ID")
    private Users users;

    @OneToOne(mappedBy = "appointment", cascade = CascadeType.ALL)
    private Bloodtests bloodtests;
```

Relațiile trebuie reprezentate folosind adnotarea specifică tipului pe care ni-l dorim: OneToOne, OneToMany, ManyToOne sau ManyToMany. În cazul de față, pentru relația de tip OneToOne avem adnotarea respectivă urmată de atributul de legătură din clasa cu care are această relație și având declarat ca atribut un obiect de acel tip, în acest caz Bloodtest. Corespondentul său în clasa de Bloodtest este următorul:

```
@OneToOne(cascade = CascadeType.ALL)

    @JoinColumn(name = "appointment_ID", referencedColumnName = "id")

    private Appointment appointment;
```

Aici avem explicit coloana din tabelul de bloodtests care corespunde acestui atribut și coloană din tabelul appointments cu care se creează legătura. În acest caz, appointment_ID este cheia străină. Ceea ce se regăsește în dreptul adnotării se referă la faptul că atunci când este șters un obiect de tipul appointment din baza de date este șters și obiectul bloodtest corespunzător. Dacă ar fi să traducem în limbaj natural această relație, practic în clasa de Appointments avem un atribut de tipul Bloodtest și în clasa de Bloodtest avem un atribut de tipul Appointment.

Următorul tip de relație pe care îl găsim în ceastă clasă este de tipul @ManyToOne, ceea ce înseamnă că în clasa corespondentă există relație prezentă cu adnotarea @OneToMany. De exemplu, în cazul relației cu tabelul de donatori, aceasta s-ar traduce astfel: un donator poate avea mai multe programări, însă o anumită programare este a unui singur donator.

```
@OneToMany(mappedBy = "users", cascade = CascadeType.ALL,

            orphanRemoval = true)

    private List<Appointment> appointmentList;
```

După cum putem observa, câteva aspecte din dreptul adnotării sunt similare, cu excepția indicației legate de orphanRemoval, ceea ce se referă la faptul că atunci când ștergem un părinte, în acest caz un utilizator, se vor șterge toți copii, adică toate programările utilizatorului. O altă diferență față de relația anterioară este faptul că, fiind vorba despre o relație în care utilizatorul are mai multe programări, acest lucru se evidențiază folosind ca atribut o listă de programări. De asemenea, în clasa de programări mai putem observa în dreptul relației faptul că modul de a face fetch (adică de a extrage datele) este eager (“ nerăbdător”). Acest lucru se referă la faptul că, atunci când avem 2 entități ca în acest caz, atunci când extragem un obiect de tipul Appointment, se extrage automat și atributul de tip User. În cazul opus, dacă fetchType ar fi fost de tipul Lazy, adică leneș, obiectul de tip User se extrage doar în cazul în care este apelat metoda respectivă de get din clasa Appointment. Acest lucru este util deoarece uneori se poate întâmpla ca mai des să folosim obiectul în sine, fără să fie nevoie și de obiectele cu care este într-o relație, așadar extragerea se face mai rapid. Totuși, în cazul meu, în marea majoritate a timpului aveam nevoie de obiecte împreună cu cele cu care aveau o anumită relație, așadar am folosit tipul eager. În cazul în care baza de date devine mai mare și eventual se extinde, voi schimba cu tipul lazy.

Referitor la celelalte metode pe care le regăsim în clasă, bineînțeles vom avea metode de tip get- pentru a accesa valorile uni anumit atribut ale unui obiect și set- pentru a atribuit anumite valori unui obiect, de obicei folosite în momentul în care se editează o înregistrare deja existentă.

În termeni de constructori, este important să avem un constructor gol atât timp cât definim și un constructor cu argumente, pentru că în cazul în care nu există, compilatorul creează automat unul. Acest lucru este necesar deoarece deși există un constructor cu argumente, JPA nu cunoaște modul în care sunt folosite acele argumente, și anume că vor fi puse ca valori acelor câmpuri. Așadar, modul în care se creează obiectele este următorul: constrctorul fără argumente creează un obiect nou, fără valori. Iar ulterior, când este apelat constructorul cu argumente tot ce face acesta este să atribuie valori atributelor unui obiect deja existent. Ceea ce este un aspect important în acest caz este faptul că atributul corespunzător cheii primare, cel care se generează automat, nu trebuie să fie inclus în constructor.

De asemenea, un alt aspect important este următorul, datorită faptului că relațiile sunt specificate în fiecare clasă, de exemplu un obiect de tip User are un obiect de tip Appointment și vice-versa, atunci când ulterior am încercat să trimit o cerere de tipul GET, răspunsul primit în format JSON apărea sub un format recursiv și nu se termina niciodată. Acest lucru era datorat faptului că bineînțeles existau referințe de la un obiect la celalalt, de exemplu utilizatorul Ana avea referință la Programarea1, Programarea1 avea referință la utilizatorul Ana și așa mai departe. Pentru a rezolva această problemă, am folosit adnotarea următoare într-una dintre clasele care erau implicate în orice relație, depinde de modul în care doream să primesc fișierul JSON:

```
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler",
"appointmentList"})
```

Această adnotare îi spune fișierului JSON să ignore atributul de appointmetList al obiectului User, astfel spărgând acest ciclu de referințe infinite.

În principiu, acesta este modul după care au fost modelate clasele corespondente tabelelor din baza de date, cu tipurile de date și relațiile aferente. Referitor la clasele de tip repository, ele sunt de fapt niște interfețe care extind interfața JPA Repository. În cazul acesta putem observa reliefate unul dintre principiile programării OO, și anume abstractizarea. Noi nu avem acces la modul în care sunt implementate metodele, însă le putem folosi doar apelându-le. O interfață de acest tip arată astfel:

```
@Repository

public interface IAppointmentRepository extends
JpaRepository<Appointment,Integer> {}
```

Trebuie specificat tipul obiectului și tipul id-ului. De asemenea, ceea ce mai putem face cu ajutorul acestei interfețe este faptul că ne putem defini metode adiționale corespunzătoare obiectelor noastre, deoarece cele existente deja de tipul save, delete sau

findById sunt generice. Există o anumită convenție de denumire a acestora în funcție de query-ul pe care ne dorim să îl generăm, însă îl putem și extrage pentru a ne asigura că este cel pe care ni-l dorim. De exemplu, pentru obiectul de tip Bloodtest, mi-am dorit o metodă care să extragă o înregistrare cu o anumită grupă de sânge și un rh.

```
@Query("select b from BloodType b where b.blood = ?1 and b.rh = ?2")

    BloodType findByBloodAndRH(String blood, String rh);
```

De exemplu, pentru metodele de tipul select .. where, acestea sunt denumite de tipul findByX And/Or Y, după cum putem observa și în exemplul de mai sus. Cu toate acestea, având în vedere că folosesc Java, un mod de a evita scrierea de astfel de query-uri este folosirea unei metode de getAll care le extrage pe toate și le filtrează ulterior folosind Java Streams, însă este mai rapid dacă extragem direct datele de care avem nevoie pe cât posibil.

Cum spuneam anterior, JPA oferă metode generice de save- adaugă o înregistrare nouă, delete- șterge o înregistrare în funcție de id dacă există, dacă nu nimic nu se întâmplă, findById- returnează un obiect în funcție de id sau returnează null dacă nu există, findAll- returnează toate obiectele dintr-un anumit tabel sau listă vidă dacă nu există nicio înregistrare. Pentru a edita un obiect, se folosește tot metoda de save, însă ca parametru se oferă un obiect care conține și id-ul acestuia specificat. Dacă metoda de save găsește id-ul, editează o înregistrare deja existentă, dacă nu adaugă una nouă.

3.1.2 Implementare funcționalități – Service

Clasele de service sunt cele responsabile cu implementarea funcționalităților efective și cu logica de business specifică fiecărui obiect. Clasele de service conțin și documentație de tipul JavaDoc. Cu ajutorul acestora putem specifica anterior fiecărei metode care sunt parametrii acesteia și ce reprezintă, ceea ce returnează metoda și eventual explicații adiacente referitoare la implementare.

De asemenea, la începutul fiecărei clase de tip service, deoarece se folosește de operațiile implementate de clasele de repository, avem adnotarea @Autowired și repository-ul respectiv. Această adnotare injectează un obiect de acel tip. Este foarte utilă mai ales când vine vorba de interfețele de repository, deoarece noi nu avem acces la metodele din interiorul lor, deci nu putem “construi” un obiect funcțional, iar Spring face deja acest lucru pentru noi.

Cum spuneam, voi relua și în acest caz analiza pe exemplu pentru clasa de Appointment. Având în vedere că utilizatorul primește e-mail atunci când șterge sau primește o programare, am avut nevoie de anumite configurări din nou în fișierul de application.properties. Trebuie specificat tipul de protocol și serviciul de e-mail pe care îl voi folosi, în acest caz protocolul smtp și serviciul gmail, deoarece voi trimite de pe adresa mea personală. De asemenea portul pe care se realizează conexiunea și modul de securizare, în cazul meu SSL, adresa de e-mail de pe care se vor trimite e-mailurile și o parolă generată prin intermediul g-mail pentru a-i permite aplicației din Intelij accesul de a trimite e-mailuri în numele meu.

După ce aceste configurări au fost făcute putem continua cu implementarea. Am injectat diferite obiecte de care ne vom folosi, în special repository-ul aferent clasei, eventual alte clase de service în cazul în care metodele au efect și asupra altor obiecte și în cazul de față un obiect de tipul JavaMailSender pentru a reuși să trimit e-mailul. Pentru motive de estetică, am declarant pentru acest exemplu diferite modele de parsare a datelor de tip Date, pentru că aveam nevoie de data și ora separate atunci când trimiteam un e-mail.

Cea mai simplă metodă este cea de extragere a tuturor înregistrărilor din baza de date, deoarece nu necesită procesare adițională. Aceasta doar apelează metoda aferentă din clasa de repository și returnează o listă cu programările găsite.

```
    public List<Appointment> getAll() {

        return (List<Appointment>) iAppointmentRepository.findAll();

    }
```

Referitor la metoda de getById, este similară celei anterioare, singura diferență este că tipul rezultatului are în față un alt tip, Optional, pentru situația în care obiectul nu există în baza de date, caz în care se returnează null.

```
public Appointment getById(Integer id) {

        Optional<Appointment> appointment =
iAppointmentRepository.findById(id);

        return appointment.orElse(null);

    }
```

Metoda următoare este modalitatea despre care menționat mai devreme, atunci când este nevoie de date filtrate, însă nu am creat un query special pentru acest lucru. Doream să extrag toate programările neconfirmate din baza de date, astfel că le-am extras pe toate și am făcut o filtrare după câmpul de confirmed, păstrându-le pe cele cu valoarea 0 și formând o listă cu acestea.

```
public List<Appointment> getAllUnconfirmed() {

    return (List<Appointment>)
iAppointmentRepository.findAll().stream().filter(o -> o.getConfirmed() ==
0).collect(Collectors.toList());
}
```

Metoda de mai jos a necesitat puțin mai multă procesare adiacentă, deoarece scopul acesteia era să confirme că o donare de sânge a avut loc. Aceasta va găsi după id programarea pe care dorim să o confirmăm, apoi va extrage spitalul în care s-a donat și grupa de sânge. Vom selecta stocul de sânge pentru grupa găsită corespunzător spitalului respectiv, apoi incrementăm acest stoc cu 0.45 litri, deoarece aceea este cantitatea donată. Mai departe, cu ajutorul unei metode de set, schimbăm câmpul de confirmed și facem update programării prin metoda de save.

```
  public void changeConfirmed(Integer id){

        Appointment appointment= this.getById(id);



        Hospital h=appointment.getHospital();

        BloodType b=appointment.getUsers().getBloodtype();

        List <Stock> s= stockService.getAll().stream().filter(o->
o.getHospital().getName().equals(h.getName()) &&

                o.getBloodtype().equals(b)).collect(Collectors.toList());

        s.get(0).setQuantity(s.get(0).getQuantity()+0.45);

        appointment.setConfirmed(1);



        iAppointmentRepository.save(appointment);

    }
```

Următoarea metodă are ca scop ștergerea programării cu id-ul pasat ca argument și trimiterea unui e-mail de înștiințare. Inițial căutăm programarea cu id-ul respective și ulterior verificăm dacă există. Dacă aceasta există, trimitem un e-mail de înștiințare astfel: creăm un nou mesaj căruia îi setăm expeditorul ca fiind e-mailul meu personal, cel din application.properties. Ulterior, setăm destinatarul ca fiind e-mailul utilizatorului care are această programare. Definim subiectul și construim mesajul. În interiorul mesajului am trimis informații specifice precum numele celui care are programarea, data și ora. Iar la final, am apelat metoda din repository responsabilă de ștergerea programării.

```
    public void deleteAppointment(Integer id){

        Appointment appointment= this.getById(id);

        if(appointment!=null){

            SimpleMailMessage mes2 = new SimpleMailMessage();

            mes2.setFrom("anamariaraita@gmail.com");

            Users users=
usersService.getById(appointment.getUsers().getId());

            mes2.setTo(users.getUsername());

            mes2.setSubject("Detalii Anulare Programare Donare");

            mes2.setText("Buna ziua, "+users.getName()+"!\n\n" +"Va
informam ca ati anulat programarea la donare din data de
"+simpleDateFormat.format(appointment.getApp_date())
```

```
                    +" la ora
"+simpleDateFormattime.format(appointment.getApp_date())+".\n\n"

                    +"Va invitam sa reveniti oricand pe site pentru a
adăuga o programare noua.\n\n"+

                    "Va dorim o zi minunata in continuare!"+"\n\n"+"Echipa
BloodDonation.");

            mailSender.send(mes2);

        }

        iAppointmentRepository.deleteById(id);

    }
```

Metoda de adăugare a unei programări este cea care conține cel mai mare volum de logică adițională. Aceasta primește ca parametru un obiect de tipul Appointment, cu ajutorul căruia extrage donatorul care vrea să o creeze și spitalul în care se va face programarea. Ulterior avem o listă în care păstrăm toate datele programărilor trecute. Facem acest lucru deoarece diferența între două donări trebuie să fie de 72 de zile. Sortăm această listă și verificăm dimensiunea ei. Dacă dimensiunea este 0, înseamnă că utilizatorul nu a mai avut programări și putem lăsa diferența de zile precum am inițializat-o, cu valoarea 90 (o valoare arbitrat aleasă) care este mai mare decât 72. Dacă valoarea este diferită de 1, adică au existat programări trecute, vom calcula diferența între cea mai recentă programare și data dorită pentru programarea actuală, exprimată în zile.

Dacă diferența este mai mica decât 72, metoda va returna null deoarece nu se poate face o programare. Altfel, vom trimite un e-mail cu detaliile programării: spitalul, adresa, numărul de contact, data și ora și vom salva programarea.

```
public Appointment saveAppointment(Appointment appointment){

    Users users= usersService.getById(appointment.getUsers().getId());
    Hospital
hospital=hospitalService.getById(appointment.getHospital().getId());

    List<Date> datesOfApp=new ArrayList<>();
    List<Appointment> appointmentList=users.getAppointmentList();
    for (Appointment a:appointmentList) {
        datesOfApp.add(a.getApp_date());
    }

    List<Date> sorted = datesOfApp.stream()
            .sorted(Comparator.comparingLong(Date::getTime))
            .collect(Collectors.toList());

    long diffrence= 90;
    if(sorted.size()!=0){
      Date  lastApp=sorted.get(sorted.size()-1);
        Date appDate=appointment.getApp_date();
```

```
        long diff = appDate.getTime() - lastApp.getTime();
        TimeUnit time = TimeUnit.DAYS;
        diffrence = time.convert(diff, TimeUnit.MILLISECONDS);
    }
    if(diffrence < 72){
        return null;
    }
    else{
        SimpleMailMessage mes = new SimpleMailMessage();
        mes.setFrom("anamariaraita@gmail.com");
        mes.setTo(users.getUsername());
        mes.setSubject("Detalii Programare Donare");
        mes.setText("Buna ziua, "+users.getName()+"!\n" +"Va multumim ca
ati ales sa salvati o viata!\n\n"+
                "Va asteptam in data de: "+
simpleDateFormat.format(appointment.getApp_date()) +" la ora "
+simpleDateFormattime.format(appointment.getApp_date())  +" la spitalul "
                +hospital.getName()+".\n\n"+"Adresa acestuia este
urmatoarea: "+hospital.getAddress()
                +".\n\n"+"Pentru informatii suplimentare va rugam sa ne
contactati la numarul " +
                "de telefon: "+hospital.getPhonenumber()+".\n\n"+"Va dorim
o zi minunata in continuare!"+"\n\n"+"Echipa BloodDonation.");
        mailSender.send(mes);


        return iAppointmentRepository.save(appointment);
    }

}
```

Pentru restul claselor, abordarea funcționalităților a fost similară doar cu logica corespunzătoare obiectelor respective. De exemplu, atunci când adăugat un donator sau personal medical, înainte să apelez metoda de save, cu ajutorul lui set criptam parola respectivă și o salvam astfel.

3.1.3 Implementare funcționalități – Controller

Clasele de controller sunt responsabile pentru stabilirea API-urilor fiecărei metode, ceea ce în limbaj natural s-ar denumi stabilirea rutelor corespunzătoare fiecărei metode.

La începutul fiecărei clase vom regăsi următoarea secvență:

```
@CrossOrigin(origins = "http://localhost:3000")

@Controller

@RequestMapping(value = "/appointment")
```

Deoarece vom trimite cererea dintr-o origine diferită, pentru a evita erorile de CORS, specificăm faptul că autorizăm cererile care provin dintr-o altă origine, atât timp cât aceasta este cea pe care am menționat-o acolo. De asemenea, faptul că avem adnotarea de RequestMapping acolo ne arată că toate rutele pe care le vom defini mai jos în această clasă

pentru diferite metode vor avea /appointment înainte. Am făcut acest lucru deoarece în acest fel am putut menține rute similare între metode care au aceeași responsabilitate dar pentru obiecte diferite, iar identificarea obiectului se face după modul în care începe ruta respectivă.

Ulterior, definirea rutelor arată astfel:

```
@RequestMapping(method = RequestMethod.GET, value = "/unconfirmed")

@ResponseBody

@HasDoctorAuthority

    public List<Appointment> getAllUnconfirmed() {

        UserDetails principal = (UserDetails)
SecurityContextHolder.getContext()

                .getAuthentication().getPrincipal();

        Staff staff=staffService.findByUsername(principal.getUsername());

        return appointmentService.getAllUnconfirmed().stream().filter(o ->
o.getHospital().getName().equals(staff.getHospital().getName())).collect(Co
llectors.toList());

    }
```

După cum putem observa, în acest caz menționăm faptul că este o metodă de tip GET, adică extragem înregistrări din baza de date. Ruta care urmează după /appointments este /unconfirmed, deoarece este în concordanță cu metoda pe care o voi apela, și anume cea care returnează toate programările neconfirmate. De asemenea, adnotarea care o precedă @HasDoctorAuthority este o adnotare definită de mine, care are la bază conceptul de @PreAuthorize din Spring, și anume faptul că ruta respectivă poate fi accesată doar dacă utilizatorul are rolul de personal medical. Primul lucru din corpul metodei este faptul că aceasta interceptează numele utilizatorului care face cererea, iar apoi găsim utilizatorul respectiv după nume. Avem nevoie de acest lucru deoarece dorim să returnăm doar programările neconfirmate din cadrul spitalului în care activează doctorul respectiv, pentru că nu dorim să aibă acces la ele cineva dinafară, așadar înainte să returnăm rezultatul îl filtrăm după spitalul respectiv.

Pentru adăugarea unei programări avem o metodă de tip POST, deoarece trimitem date pe care vrem să le persistăm în baza de date.

```
@RequestMapping(method = RequestMethod.POST, value = "/save")

@ResponseBody

@HasUserAuthority

    public void saveAppointment(@RequestBody Appointment appointment) {

        UserDetails principal = (UserDetails)
SecurityContextHolder.getContext()
```

```
                .getAuthentication().getPrincipal();
        Users users=usersService.findByUsername(principal.getUsername());
        appointment.setUsers(users);
        appointmentService.saveAppointment(appointment);
    }
```

Abordarea este una similară, doar donatorul are autorizare pentru această rută și din nou interceptăm cine face cererea deoarece dorim ca un utilizator să poată adăuga o programare doar pentru el însuși. După ce setăm faptul că donatorul acestei programări este chiar cel care a făcut cererea, salvăm programarea.

Abordarea este similară pentru celelalte metode din această clasă și pentru celelalte clase din controller. Toate au în comun faptul că trebuie specificat tipul metodei și ruta, unele dintre ele fiind cu autorizare și de obicei la acestea și interceptăm utilizatorul.

3.1.4 Securitate

Pentru a implementa partea de securitate am utilizat anumite facilități pe care le oferp Spring. Pe lângă partea efectivă de criptare a parolelor în baza de date, mi-am dorit ca partea de autentificare să se realizeze prin intermediul jwtoken. Acesta reprezintă un șir de caractere obținute în urma unui algoritm de hashing în care se regăsesc numele utilizatorului, parola și rolul. Am dorit să am această implementare deoarece prin intermediul ei am reușit să creez o sesiune de log-in in interfață și să securizez anume API. Practic, doream ca la unele dintre metode să aibă acces doar utilizatori cu un anumit rol. Pentru a începe implementarea, am avut din nou nevoie de o configurare in application.properties în care să specificăm cheia folosită pentru algoritmul de generare al tokenului.

Această parte este constituită din mai multe clase, însă voi discuta despre cele mai importante. Prima dintre ele, numită JwtTokenUtil este responsabilă cu operațiile de creare și validare a tokenului. Deoarece implementează interfața Serializable, este nevoie de o un identificator pentru a indica modul în care se serializează obiectul. De asemenea, tot aici setăm un timp de valabilitate al tokenului. Există diferite metode implementate cu ajutorul cărora putem extrage din token numele utilizatorului și data de expirare. De asemenea, avem și o metodă care cunoaște cheia folosită în algoritmul de hashing și poate parsa tokenul pentru a obține informațiile dorite.

Metoda de validare este destul de simplă, deoarece compară numele utilizatorului care este logat cu numele pe care îl extrage din token și verifică dacă nu a expirat.

Metoda mai interesantă este cea de generare a tokenului:

```
   private String doGenerateToken(Map<String, Object> claims, String
subject) {
```

```
        return
Jwts.builder().setClaims(claims).setSubject(subject).setIssuedAt(new
Date(System.currentTimeMillis()))

                .setExpiration(new Date(System.currentTimeMillis() +
JWT_TOKEN_VALIDITY * 1000))

                .signWith(SignatureAlgorithm.HS512, secret).compact();

    }
```

Pentru a construi tokenul trebuie să setăm ceea ce se numește “claims”, adică cine generează tokenul, pentru cine, id-ul și data de expirare. De asemenea, aici desemnăm și algoritmul de hashing în combinație cu cheia.

Clasa JWTUserDetailsService implementează interfața din Spring numită UserDeatilsService, așa că trebuie să suprascriem metoda loadByUsername care are scopul de a extrage utilizatorul în cauză din baza de date.

Deoarece am partea de log-in comună pentru ambele tipuri de utilizatori, am implementat această metodă astfel:

-caut utilizatorul după nume în ambele tabele care conțin utilizatori

-verific dacă cumva am găsit într-unul dintre tabele, și în acel caz îl returnez pe cel pe care l-am găsit

-dacă nu am găsit în niciun tabel acel utilizator nu există

-singurul dezavantaj ar fi aici faptul că trebuie să mă asigur că nu există nume de utilizator comune, ceea ce am validat din interfață deoarece donatori au ca nume o adresă de e-mail, iar numele personalului medical nu este acceptat sub forma unei adrese de e-mail

De asemenea, în funcție de tabelul în care le găsesc creez obiectul de tipul respectiv și ofer utilizatorului rolul pe care îl are.

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



        Users user1 = userService.findByUsername(username);

        Staff staff=staffService.findByUsername(username);

        Set<GrantedAuthority> authorities =new HashSet<>();

        if (user1 == null && staff==null) {

            throw new UsernameNotFoundException("User not found with
username: " + username);

        } else if(user1!=null && staff==null){
```

```
            authorities.add(new SimpleGrantedAuthority("user"));
            User user = new User(user1.getUsername(), user1.getPassword(),
                    authorities);
            return user;
        }else{

            if(staff.getRole().equals("admin")){
                authorities.add(new SimpleGrantedAuthority("admin"));
            }
            else{
                authorities.add(new SimpleGrantedAuthority("doctor"));
            }
            User user = new User(staff.getUsername(), staff.getPassword(),
                    authorities);
            return user;
        }
    }
```

De asemenea, există 2 clase care lucrează împreună, JWTRequest și Response, una dintre ele fiind responsabilă de păstrarea numelui de utilizator și a parolei, cealaltă având rolul de a crea răspunsul la cererea HTTP care conține tokenul generat.

De asemenea, mai există o clasă care se ocupă cu filtrarea cererilor, JwtRequestFilter, care din nou implementează o interfață deja existentă în Spring și suprascrie o metodă. Acesta verifică dacă se poate extrage tokenul din șirul primit. Acesta trebuie să înceapă cu sintagma Bearer și să fie valid.

Dacă tokenul este valid, autentificarea are loc și tokenul este generat.

În final, o clasă foarte important este cea de WebSecurityConfig, care la rândul ei extinte WebSecurityConfigurerAdapter existentă în Spring. Trebuie să suprascriem metoda configure, care ne arată pentru ce rute sunt autorizate cereri. În cazul în care avem rute care nu sunt menționate aici, vom avea eroare de CORS și răspunsul 401, Unauthorized. Doar pentru că rutele sunt aici acest lucru nu înseamnă că ele sunt autorizate automat. Inițial, dacă rutele sunt aici se verifică autorizarea prezentă în controller. Dacă acolo nu există o altă restricție, cererea merge mai departe, iar dacă este nevoie de autorizare se verifică dacă utilizatorul o are.

3.2 Frontend-Client

Partea de client s-au frontend a fost implementată în React, o tehnologie care se bazează pe limbajele html, css și javascript. De asemenea, am folosit anumite componente predefinite ori de către React ori de Boostrap pentru a avea o stilizare cât mai estetică. Principiul din React este că fișierul care conține cod css este separat de fișierul care conține cod javascrip și html, însă există mici inserții legate de stilul unei componente care se pot realiza și la nivelul fișierelor .js.

Deoarece structura paginilor este similară, cu diferite componente html sau stilizări, voi încerca să discut integral exemplul unei pagini, de exemplu cea care va arăta medicului programările confirmate din spitalul său. De asemenea, voi mai discuta modul în care se realizează autentificarea și modul în care se trimite o cerere de tip POST care necesită autorizare, deoarece în cazul autentificării nu este nevoie de așa ceva.

Inițial, fișierul de App.js este cel în care se stabilesc rutele aplicației, cât și eventuale proprietăți pe care le dorim prezente în toată aplicația. În cazul meu, am dorit ca paddingul să fie 0 peste tot și de asemenea să existe un footer cu copyright cu numele meu. De asemenea la nivel de componente reutilizabile, am definit câte un meniu de navigare pentru fiecare tip de utilizator, inclusiv cel neautentificat. În fiecare pagină, se face o verificare și în funcție de utilizatorul autentificat este afișat meniul de navigare corespunzător. Diferite proprietăți de stil sunt stabilite într-un fișier css, astfel încât toate meniurile să arate similar și doar butoanele să difere.

```
  <Nav>

          <Bars />

          <NavBtn>

            <NavBtnLink to='/stock'>Vezi stocurile</NavBtnLink>

            <NavBtnLink to='/confirmed'>Programările
confirmate</NavBtnLink>

            <NavBtnLink to='/unconfirmed'>Programările
viitoare</NavBtnLink>

            <NavBtnLink to='/apeluri'>Adaugă cerere</NavBtnLink>

            <NavBtnLink to='/update'>Raport stoc</NavBtnLink>

            <NavBtnLink to= '/' onClick={()=>logout()}>Log Out</NavBtnLink>

          </NavBtn>

          <div  style={{marginTop:"20px"}}>

          <NavBtn >

          Donează sânge, fii erou!
```

```
          <NavBtnLink to= '/doctorpage'> <BiUserCircle
size={'2em'}/></NavBtnLink>

          </NavBtn>

          </div>

        </Nav>
```

Sunt definite butoanele și pagina la care duc acestea, un text motivațional pentru utilizatori și un buton realizat cu ajutorul unei iconițe de utilizator importată cu numele BiUserCircle.

Pentru pagina de Log In, există un formular care preia datele, iar atunci când se apasă butonul de trimitere, următoarea funcție este realizată:

```
    const [errorMessage, setMessage] = useState("");

    async function handleSubmit(event) {

        localStorage.removeItem("user");

        localStorage.removeItem("role");

        var { uname, pass} = document.forms[0];

      try {

        var response= await
axios.post("http://localhost:8080/getAuthority",{username: uname.value,
password: pass.value})

        var response2= await
axios.post("http://localhost:8080/login",{username: uname.value, password:
pass.value})

        localStorage.setItem('role',JSON.stringify(response.data));

        localStorage.setItem('user',JSON.stringify(response2.data.token));



       if( JSON.parse(localStorage.getItem("role"))[0].authority ===
"admin" )

        navigate("/adminpage")

        if( JSON.parse(localStorage.getItem("role"))[0].authority ===
"doctor" )

        navigate("/doctorpage")

        if( JSON.parse(localStorage.getItem("role"))[0].authority ===
"user" )

        navigate("/userpage")
```

```
      } catch (error) {
        setMessage("Credentiale incorecte");
      }
    }
```

După cum putem observa, inițial am folosit un state hook oferit de React. Acestea sunt folosite pentru a păstra o stare, o valoare între diferite apeluri de funcție. Primul argument, în cazul meu errorMessage este o variabilă cu o anumită valoare iar setMessage este modul în care putem atribui valori acestei variabile. Ar fi într-o oarecare măsură echivalent cu variabilele globale din alte limbaje.

Aceste hooks sunt foarte utile deoarece, în exemplul de față, eu doresc să afișez un mesaj de eroare în funcție de rezultatul operației de autentificare. Inițial acest mesaj este gol, iar dacă autentificarea eșuează este schimbat. Această schimbare apare în timp real în interfață, deoarece variabila este folosită în componenta html și când valoarea ei se schimbă se și observă acest lucru.

De asemenea, sunt foarte benefice în cazul în care așteptăm un răspuns de la o cerere de tip GET și dorim să populăm pagina cu acel răspuns. Putem folosi variabila respectivă în construirea componentelor de html, iar pe măsură ce ea se schimbă, adică primim răspuns de la request, vedem acest lucru și în interfață cu ușurință.

Funcția este asincronă și așteaptă răspunsul celor două cereri POST, cereri care vor returna tokenul și rolul utilizatorului. După aceea, stocăm tokenul în storage pentru a genera o sesiune în acest fel și în funcție de rol îl stocăm și pe acesta. Totuși, la început, ștergem eventuale valori care ar fi putut rămâne în storage pentru orice eventualitate. Ulterior, în cazul în care este prezent un răspuns favorabil, se navighează mai departe la următoarea pagină în funcție de tipul de utilizator, iar dacă utilizatorul cu acel nume și parolă nu există, primim un răspuns nefavorabil și rămânem tot pe pagina de log in. Acesta este un tip de cerere post a cărei header nu necesită token, deoarece nu are nevoie de autorizare, însă conține body (corpul cererii) reprezentat de datele din formular.

Următoarea cerere de tip POST necesită autorizare, nu are un body, în schimb necesită așa numiții query-params, care reprezintă niște parametrii atașați URL-ului. Sunt utili pentru cazul în care nu dorim să trimitem un obiect întreg, ci doar unul, două câmpuri.

```
async function handleSubmit(event) {
        var { gr,cuan} = document.forms[0];
        var obj = JSON.parse(localStorage.getItem('user'));
        let headers2 = {
            "Content-type": "application/json; charset=UTF-8",
            "Authorization": 'Bearer '+ obj
```

```
            };
            const params = new URLSearchParams({
               gr: gr.value
             }).toString();
             const params2 = new URLSearchParams({
               cuan: cuan.value
             }).toString();
            try {
               var response= await
axios.post("http://localhost:8080/stock/edit?"+params+"&"+params2,
                    {headers: headers2}
                    )
             } catch (error) {
               setMessage("Canitatea introdusa este gresita!");
             }
    }
```

După cum se vede, inițial folosim JSON.parse pentru a extrage tokenul din locul în care este stocat în browser. Apoi construim headerul cererii. La secțiunea de Authorization, știm că tokenul nostru trebuie să înceapă cu sintagma Bearer, așa că alcătuim un șir format din această sintagmă și tokenul în sine.

Ulterior, definim cei doi parametrii de tipul URLSearchParams, pentru că se va face conversia automată la formatul necesar, și anume numeparametru=valoare. Apoi construim URLul atașând cei doi parametrii, atașăm headerul construit și trimitem cererea.

Exemplul în cauză surprinde funcționalitatea de consum de stoc, adică trebuie să trimitem o cantitate mai mică decât stocul disponibil. Dacă primim un răspuns bun, modificarea s-a realizat cu succes, altfel vom primi o eroare si semnalăm că este introdusă o cantitate greșită.

Așadar, în cele ce urmează voi discuta integral pagina care conține toate programările confirmate din spitalul personalului medical autentificat. Voi trata inițial partea de javascript și apoi cea de html.

Există câteva funcții mai mici, al căror scop este doar de mă ajuta să afișez estetic anumite tipuri de date. De exemplu:

```
    function fromint(nr){
```

```
        if(nr == 1){
            return "Confirmată."
        }
        else
            return "Neconfirmată."
    }

    function fromBool(nr){
        if(nr == 1){
            return "Pozitiv"
        }
        else
            return "Negativ"
    }

    function toDate(date){
        return date?.split('T')[0]
    }
    function toTime(date){
        return date?.split('T')[1].substring(0,5)
    }
```

Prima funcție fromint mă ajută să afișez starea programărilor. Cum spuneam mai devreme, în baza de date acest atribut este reținut ca număr, 0 pentru o programare neconfirmată și 1 altfel. Deși în această pagină am doar programări confirmate, am refolosit această funcție așadar nu am mai modificat-o. Modul de funcționare este simplu, dacă primește ca parametru valoarea 1 afișează șirul de caracter “Confirmată” și vice-versa.

Deoarece în această pagină vom afișa după caz și rezultatele analizelor, avem nevoie și de funcția fromBool. Funcționează pe același principiu ca funcția anterioară, însă fiind vorba de un rezultat în urma analizei parametrilor corespunzători unei boli, aceștia afișează Pozitiv sau Negativ.

Următoarele două funcții au ca scop afișarea estetică a datei și orei programării. Prima funcție returnează tot ceea ce găsește până la T (deoarece litera T desparte data și ora în

formatul JSON al răspunsului), iar cealaltă returnează primele 5 caractere de după T, deoarece vrem să obținem doar ora și minutul, nu și secundele.

Următoarea funcție folosită returnează o componentă în funcție de obiectul care este dat ca parametru. Deoarece ne aflăm într-o pagină cu programări confirmate, urmează ca doctoral să introducă analize la programările confirmate la care nu au fost introduse încă. Pentru celelalte, rezultatele sunt vizibile.

```
   function bloodtestButton(appointment){



        if (appointment.bloodtests)

        return   <Button variant="success"
onClick={()=>togglePopup2(appointment)}>Vezi Rezultatele</Button>

        else

        return <Button variant="info"
onClick={()=>togglePopup(appointment?.id)} >Adauga Analize</Button>

    }
```

Așadar, dacă obiectul appointment are câmpul bloodtests diferit de null, adică există rezultate adăugate, vom afișa pentru acea programare butonul conform căruia poate vedea rezultatele, altfel dacă nu există deci este null, afișăm butonul conform căruia poate adăuga rezultate.

De asemenea, pentru a trimite cererea GET către server și a extrage programările pentru a le afișa în pagină încă de când on accesăm folosim un hook de tip Effect.

```
      useEffect(() => {

        var obj = JSON.parse(sessionStorage.getItem('user'));

        let headers2 = {

            "Content-type": "application/json; charset=UTF-8",

            "Authorization": 'Bearer '+ obj

             };

        axios.get("http://localhost:8080/appointment/confirmed",{ headers:
headers2})

            .then(res => {

                setData3(res.data);

            })

            .catch(err => {
```

```
                console.log(err);

            })

    }, [])
```

Și în acest caz trimit o cerere de tip GET în a cărui header trimit tokenul, deoarece doar personalul medical poate accesa această metodă, iar pentru a gestiona răspunsul folosesc un state hook pentru ca apoi voi putea afișa în interfață.

Mai avem prezente diferite funcții care fac vizibil sau nu popup-urile pe care le-am folosit fie pentru a introduce rezultatele analizelor, fie pentru a le afișa. De asemenea, mai există și o funcție care trimite o cerere de tip post, însă este similară cu cea pe care am discutat-o anterior.

Cu ajutorul următoarei condiții verific tipul utilizatorului care este autentificat și dacă este autentificat:

```
var data=JSON.parse(localStorage.getItem("role"))[0].authority

if(JSON.parse(localStorage.getItem("user")) && data==="doctor")
```

Iar dacă avem utilizatorul potrivit returnez pagina cu datele la care are acces doctorul, altfel utilizatorul este întâmpinat cu un mesaj care îi spune că nu este autorizat și îl invită către pagina de log in.

Pentru a afișa programările am folosit o componentă predefinită, și anume un Card, afișând programările precum ar fi niște cărți de vizită.

```
   {data3.map((appointment) => (

            <Card bg='danger' style={{ width: '18rem',margin:'8px' }}>

            <Card.Header style={{fontWeight: "bold"}}>Spitalul:
{appointment?.hospital?.name} <img src="fp.png" height='30px'
style={{float:'right'}}></img></Card.Header>

            <Card.Img variant="top"  />

            <Card.Body>

                <Card.Title> {fromint(appointment?.confirmed)}</Card.Title>

                <Card.Text>

                Programarea este în data de:<br></br>

                 {toDate(appointment?.app_date)}<br></br>

                La ora: {toTime(appointment?.app_date)}<br></br>

                Adresa: {appointment?.hospital?.address} <br></br>

                Contact la nr: {appointment?.hospital?.phonenumber}
```

```
                </Card.Text>
                {bloodtestButton(appointment)}
            </Card.Body>
            </Card>
               ))}
```

Funcția .map care se regăsește chiar la început ar fi echivalentul unui foreach. Adică, pentru fiecare element conținut în data3, în acest caz programări, deci pentru fiecare programare afișează câte un card de forma definită ulterior cu informațiile aferente programării pe care o procesează atunci. Cu ajutorul acoladelor introducem în html variabile folosite în partea de javascript. Semnul întrebării așezat după obiectul principal verifică dacă nu cumva obiectul este undefined (nedefinit). Dacă ar fi nedefinit și semnul întrebării nu ar exista, nu ar putea să efectueze operatorul "." și am primi o eroare.

Ulterior alte componente de html sunt două popup-uri, însă sunt similare deci vom vorbi numai despre unul dintre ele, cel care reprezintă un formular.

Acesta conține mai multe câmpuri, care pot fi completate fie selectând anumite valori, fie introducând date. De exemplu:

```
<Form.Group>
                    <Form.Label>HEPATITA C</Form.Label>
                    <Form.Select aria-label="Default select example" name=
"hc">
                    <option value='0'>Nu</option>
                    <option value='1'>Da</option>
                    </Form.Select>
                </Form.Group>
                <Form.Group>
                    <Form.Label>TROMBOCITE</Form.Label>
                    <Form.Control type="number"  step="0.1" required name=
"tr" onChange={(event) =>
                                event.target.value < 0
                                    ? (event.target.value = 0)
                                    : event.target.value
                            } />
                </Form.Group>
```

În cazul câmpului Hepatita C, avem un câmp de tipul select. Deși atunci când se trimite formularul valorile pe care le preia acest câmp sunt 0 sau 1, in interfață utilizatorul le va vedea de tipul Da sau Nu.

De asemenea în cazul Trombocitelor, câmpul este de tip number, așadar nu se pot introduce altceva decât numere. Având în vedere faptul că am definit un step, se pot introduce numere de tip float, cu virgulă. Verificare din cadrul onChange nu dă voie acestui câmp să primească valori negative, iar orice încercare de a face acest lucru va schimba automat valoarea câmpului la 0.

De asemenea, în cazul în care avem câmpuri în care valorile selectate sunt înregistrări din baza de date putem face și acest lucru. În cazul meu, în cazul în care de exemplu personalul medical introduce stocul consumat acesta trebuie să selecteze grupa de sânge pentru care se consumă cantitatea. De exemplu:

```
<Form.Select aria-label="Default select example" name= "gr">

                    { data2.map((bloodtype) => (

                                    <option
value={[bloodtype.id]}>{[bloodtype.blood," ",bloodtype.rh]}</option>

                                ))

                                }

                    </Form.Select>
```

Inițial este nevoie de o cerere de tip GET pe care am implementat-o cu ajutorul unui hook de tip Effect pentru a avea grupele de sânge imediat ce pagina se încarcă, iar data2 le conține. În intermediul tagului Form.Select, din nou se aplică o funcție de tip .map pentru a parcurge lista cu toate grupele de sânge obținute. Și în acest caz, deși valoarea câmpului respectiv din formular va fi id-ul grupei respective, utilizatorul va vedea în interfață grupa de sânge și rh-ul.

4.Asigurarea calității

Termenul de asigurare a calității, uneori abreviat QA, se referă la cumulul de procese și procedee prin care se verifică și confirmă dacă produsul îndeplinește obligațiile contractuale față de client, îndeplinește funcționalitățile hotărâte și orice alte cerințe non-funcționale care s-au stabilit între cele două părți.

Deși unul dintre principiile testării este faptul că testarea exhaustivă este imposibilă, adică nu ne putem asigura niciodată că aplicația noastră nu are niciun defect, alt principiu și realitatea ne demonstrează că salvăm mai mult timp și efort dacă găsim posibile defecte cât mai devreme în cadrul dezvoltării unui produs software.

Testarea la toate nivelurile și sub toate formele este un pas foarte important în dezvoltarea și livrarea unui produs deoarece îți demonstrează atât dacă ai dezvoltat produsul

care trebuie (testare de acceptanță) și dacă ai construit produsul cum trebuie (testarea calității).

Cu toate acestea, trebuie să fim foarte atenți atunci când testăm o aplicație, mai ales propria noastră aplicație, deoarece cu cât gradul de subiectivitate și cunoaștere a procesului de dezvoltare este mai mare, cu atât mai puține șanse avem să găsim defecte. De asemenea, trebuie să ținem minte că și absența defectelor este un defect în sine, dar și că există șansa ca în cazul în care testăm tot cu aceleași date, eventual porțiunea de cod să funcționeze, însă substraturile problemei să nu fie rezolvate, acesta numindu-se paradoxul pesticidului.

De asemenea, un aspect foarte important este și testarea regresivă, adică atunci când fixăm un defect este nevoie să retestăm aplicația pentru a fi siguri că nu am introdus o problemă în altă parte.

Pentru aplicațiile mari, a căror interfață nu se schimbă constant și care au o frecvență mare de lansare a versiunilor noi cu defecte fixate, este recomandată testarea automată, însă în cazul meu, deoarece websiteul este unul comparativ mic cu alte aplicații existente, am considerat că este suficientă testarea manuală.

4.1 Testare manuală

Testarea manuală (conform Black Rex, Graham Dorothy, Van Veenendaal Erik, 2012) este o formă primitivă de testare, deoarece nu se folosesc deloc instrumente și procese automatizate, însă este destul de eficientă dacă stăm să ne gândim la faptul că de fapt utilizatorii aplicațiilor sunt tot oameni, deci este foarte posibil ca având o persoană care testează să gândească unele cazuri de testare în care ar putea ajunge și clientul.

Scopul principal este acela de a găsi erori cât mai devreme în stadiul de dezvoltare și de a ne asigura că produsul nostru respectă cerințele funcționale stabilite inițial.

Atunci când se face testare manuală, printre altele, există două tipuri principale de testare: testare black-box și white-box. Cea din urmă este realizată împreună cu un alt programator de obicei, deoarece necesită cunoștințe asupra codului și modul intern de funcționare al programului. Un nume informal al acestei tehnici ar fi Code Review, adică un alt programator sau un grup de programatori analizează codul scris de o persoană relativ la o funcționalitate fără ca programul să ruleze efectiv. Așadar, având în vedere că nu am dispus de o altă persoană, nu aveam cum să aplic această tehnică deoarece analiza era una subiectivă.

Cu toate acestea, am încercat să respect convențiile nescrise pentru a avea un cod curat și lizibil. Aici mă refer la convențiile de denumire ale atributelor și metodelor, astfel încât numele să fie sugestiv cu funcționalitatea metodei sau cu ceea ce exprimă atributul. De asemenea, importurile pachetelor sunt menținute toate în partea inițială a fiecărui fișier, metodele get, set și constructorii se află în această ordine în clase cu spațierea și indentarea potrivită. De asemenea, am încercat să modularizez cât mai bine și să păstrez majoritatea

metodelor sub 20 de rânduri de cod, iar ca parametrii am încercat să mențin numărul lor sub 4.

Așadar, o tehnică mai potrivită și care mă ajută să îmi păstrez un grad de obiectivitate este testarea de tip black-box.

4.1.1 Testare black-box

În acest tip de testare(conform Black Rex et al 2012), cel care o execută se plasează în perspectiva utilizatorului extern sau a clientului. Această testare implică faptul că aplicației testate nu i se cunoaște structura internă a codului sau detalii de implementare, ci doar funcționalitățile pe care trebuie să le îndeplinească. De aceea se începe cu acest pas, de a analiza foarte bine cerințele funcționale și non-funcționale ale aplicației. Ulterior, se determină scenariile pozitive, adică inputurile valide și scenariile negative, adică inputurile invalide, cele care ar trebui să genereze excepții gestionate de sistem. Rezultatele așteptate sunt stabilite de cel care testează consultând cerințele, construind apoi pe baza pașilor de execuție, a valorile de intrare și a rezultatelor se construiesc cazurile de testare. Ulterior, după executare, rezultatele sunt comparate cu cele așteptate și eventuale defecte sunt rezolvate.

Atât în cazurile în care am testat cu valori valide sau invalide, am decis să folosesc atât Postman, pentru a testa APIs din backend, cât și ulterior după ce am implementat interfața în React am testat de acolo, folosind atât rezultatele vizuale pentru a cuantifica rezultatul, cât și Developer Tools oferite de orice browser de internet.

Din Postman atunci când testăm selectăm tipul metodei dorite, trimitem informațiile în format JSON și le primim în același format. Ceea ce este un avantaj sau dezavantaj la Postman este faptul că evită erorile de CORS (Cross-origin resource sharing), astfel încât testarea este mai ușoară, însă din acest motiv am avut dificultăți atunci când am testat din browser deoarece am avut de-a face cu aceste erori atunci când am trimis cereri din browser.

Testând separat din Postman, am reușit să izolez mai bine anumite erori care au apărut pe parcurs, astfel atunci când am trecut la implementarea interfeței eram sigură că funcționalitățile implementate în Java funcționează corect și potențiale erori au fost generate de modul în care trimit cererile din React.

4.1.2 Testare pozitivă

Această modalitate(conform Black Rex et al 2012) implică faptul că alegem un set de date valide și verificăm în special că sistemul se comportă corect atunci când primește un set de date valid și execută funcționalitatea la care ne așteptăm.

Figura 22 reprezintă unul dintre aceste teste pozitive, anume un caz de cerere POST, mai exact un log in executat cu succes, adică aceste credențiale există în această corespondență în baza de date. După cum putem observa, scenariul este unul de succes, deoarece am primit ca răspuns un token generat și mesajul 200OK. Din acest mesaj deducem că cererea s-a realizat cu succes și am primit token-ul pe care îl vom folosi pentru generarea sesiunii. După cum putem observa, această metodă nu necesită autorizare, așadar nu a fost nevoie să trimitem în headerul cererii niciun token.

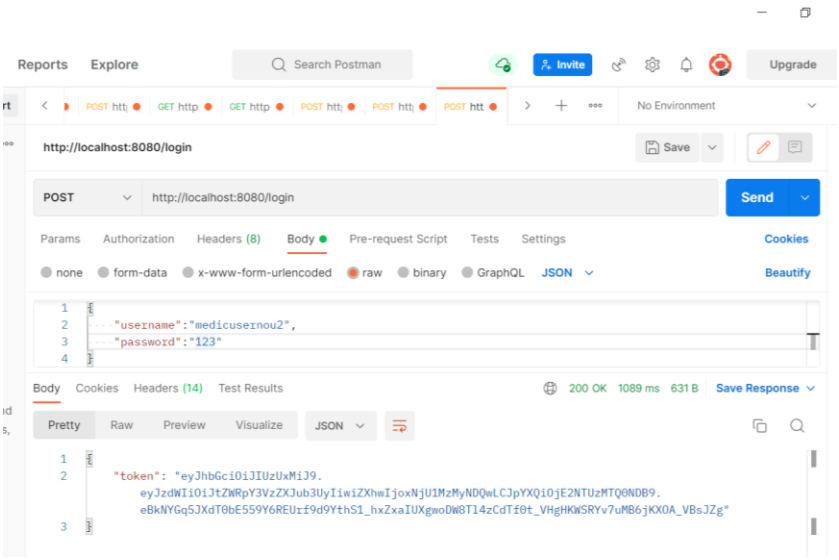


Figura 22 - Testare Postman - Log In Succes

În Figura 23 de asemenea este testată o metodă de GET prin care vrem să vedem dacă putem extrage toate apelurile la donare. Această cerere deși nu are un body, are nevoie de autorizare din partea unui cont de admin. Această autorizare se realizează prin trimiterea unui token în headers, după cum putem observa și din imagine că am trimis.

În momentul în care am trimis un token potrivit, adică cel generat în urma accesării contului unui admin. Când această cerere este trimisă, ca răspuns am primit în format JSON toate apelurile la donare.

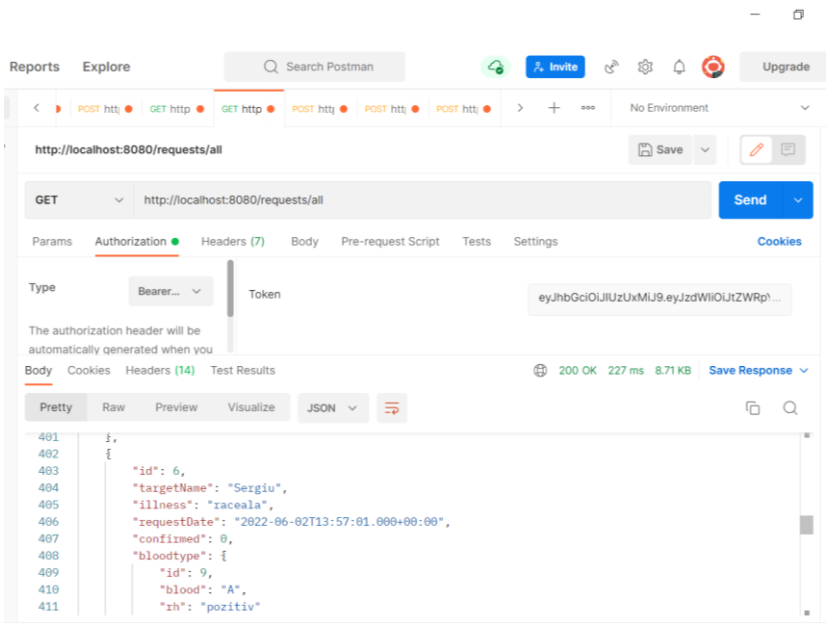


Figura 23 - Testare Postman Apeluri la Donare – Succes

O problemă pe care am întâlnit-o pe parcursul testelor în Postman, însă pe care am rezolvat-o relative repede, a fost generate de relațiile pe care le avem între anumite obiecte din Java. De exemplu, în figură se poate observa faptul că un obiect de tipul request deține un obiect de

tipul bloodtype, deoarece avem între ele o relație de One-To-Many. Așadar, în request avem un bloodtype, în bloodtype avem o listă de requesturi. Însă, fiecare request din listă are la rândul lui un bloodtype și așa mai departe, astfel încât primeam un fișier JSON recursive. Această problem a fost rezolvată relativ ușor, adăugând în fișierul de Java o adnotare @JSONIgnoreProperties, care îi spune în acest caz obiectului de bloodtype să ignore atributul de lista de request, așadar scăpând de această recursivitate.

Mai departe, voi exemplifica un test făcut din interfață legat de extragerea programărilor din baza de date specifice unui utilizator. În acest caz, pentru a demonstra corectitudinea, voi face o paralelă cu înregistrările din MySQL pentru a observa că au fost afișate programările corespunzătoare acelui utilizator cu a cărui cont m-am conectat.

În acest caz, m-am conectat cu utilizatorul al cărui id este numărul 5. După cum putem observa în tabelul de programări, acest utilizator are două programări confirmate pe care le putem observa afișate în interfață.

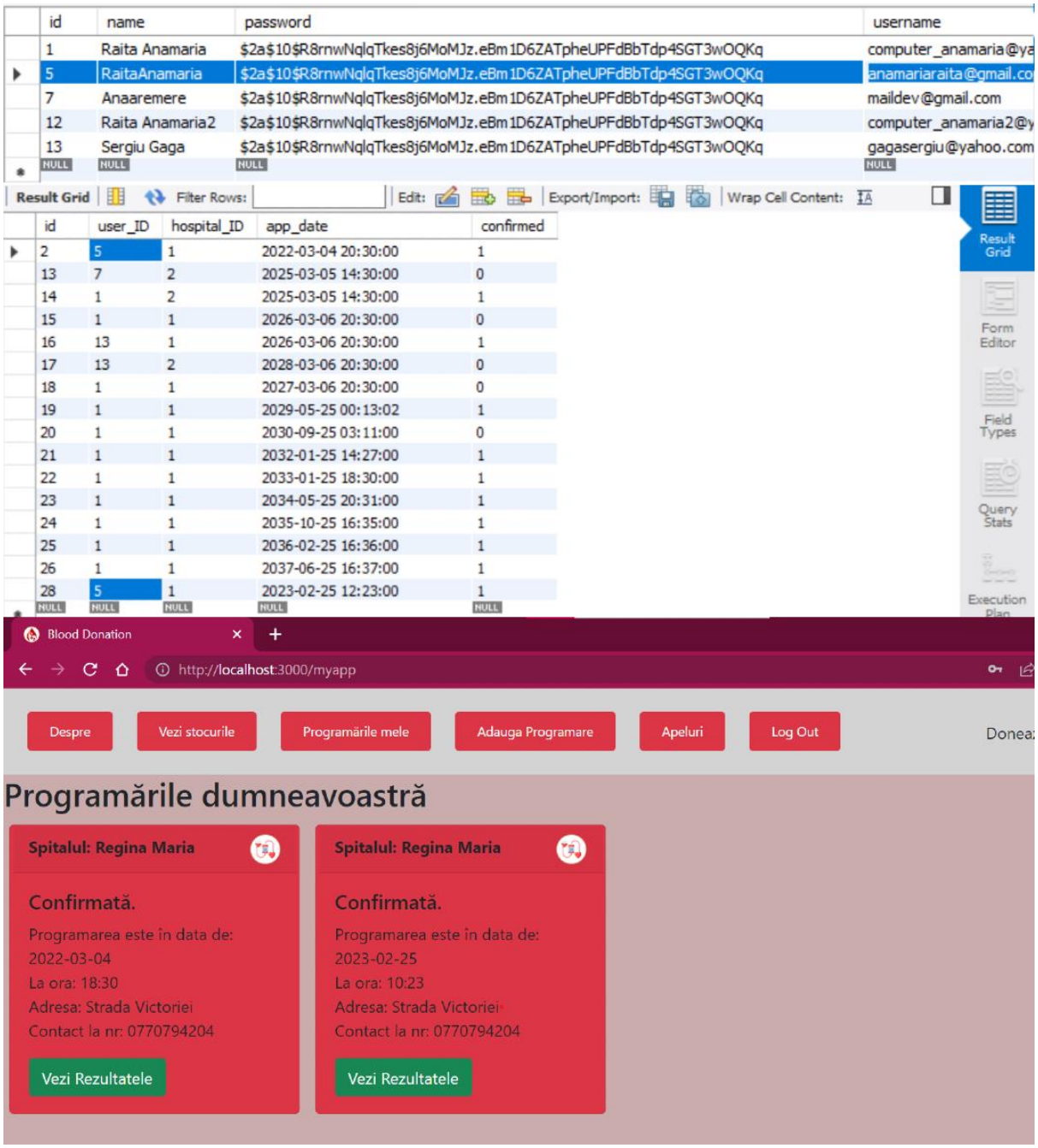


Figura 24 - Testare interfață - Programările utilizatorului

4.1.3 Testare negativă

Această modalitate(conform Black Rex et al 2012) implică faptul că alegem un set de date invalide și verificăm în special că sistemul se comportă corect atunci când primește un set de date invalid. Rezultatul așteptat ar fi ori că sunt generate excepțiile potrivite în conformitate cu ceea ce este invalid la date ori că gestionează corect un comportament neașteptat, neautorizat sau neobișnuit al utilizatorului.

În cele ce urmează, voi prezenta câte un exemplu din fiecare situație enunțată mai sus și modul în care se comportă aplicația mea.

Figura 24 reprezintă o încercare de trimitere a unei cereri GET pentru a extrage apelurile la donare din baza de date. De această dată, tokenul folosit este unul corespunzător unui cont de personal medical, așadar răspunsul așteptat este faptul că răspunsul cererii este un status de tip Forbidden, deoarece utilizatorul de tip doctor nu are acces la acest tip de metodă.

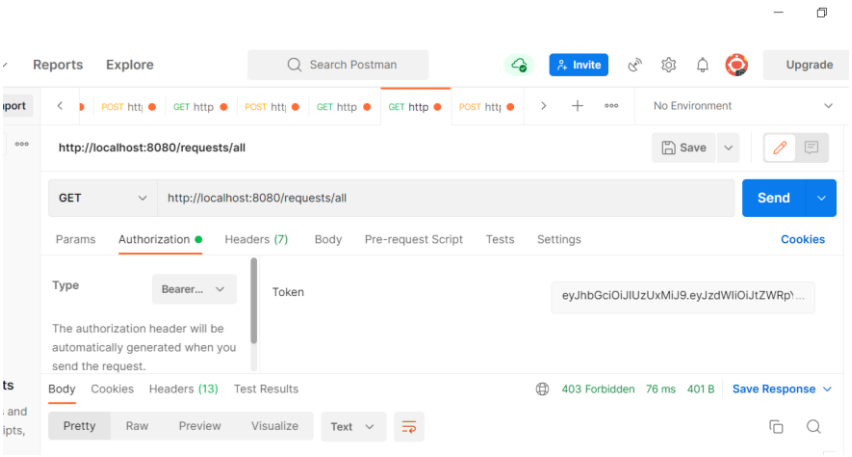


Figura 25 - Testare Postman Apeluri la Donare – Eșec

Această situație reprezintă un exemplu de comportament neașteptat din partea unui utilizator, deoarece el nu are acces la această metodă. După cum putem observa, răspunsul este un status 403 Forbidden, așadar aplicația a gestionat corect acest tip de comportament.

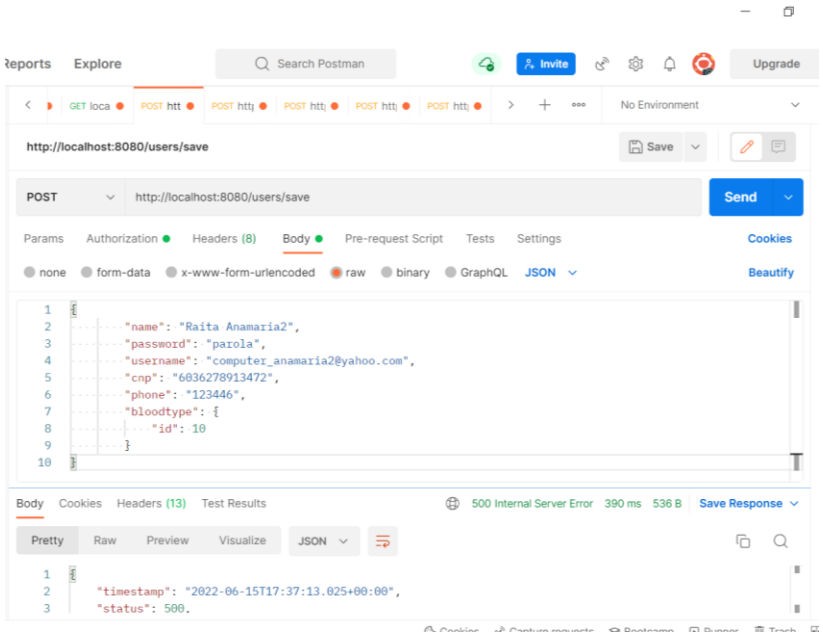


Figura 26 - Testare Postman - Înregistrare utilizator - Eșec

Figura 25 reprezintă cealaltă situație menționată, și anume atunci când introducem date invalide. În acest caz, e-mailul introdus în câmpul de username există deja în baza de date. Având în vedere că o restricție pentru acest câmp de a fi unic, atunci când se încearcă

înregistrarea unui utilizator cu un username care mai există deja operația nu trebuie să fie posibilă, ci să primim o excepție.

După cum putem observa, răspunsul obținut este un status de 500 Internal Server Error, adică cererea nu s-a concretizat și utilizatorul nu s-a creat. Reacția la datele invalide este una corectă și în acest caz.

Pentru acest tip de testare am ales exemplific următoarea situație: atunci când un utilizator nu este conectat deloc sau nu este conectat cu un cont care are un anumit rol, acesta nu are acces la anumite pagini.

În cazul de mai jos ceea ce ar trebui să apară atunci când suntem conectați cu un cont de personal medical sunt programările neconfirmate din cadrul spitalului la care este atribuit acest cont.

Dacă nu este conectat sau este conectat cu un rol diferit, pagina nu se va încărca având informațiile obișnuite, ci va afișa un mesaj de atenționare cum că utilizatorul nu este autorizat în această zonă și i se oferă posibilitatea redirecționării spre pagina de log in.

În cazul de față, se poate observa că suntem autentificați cu un cont de utilizator donator, deci nu putem accesa această pagină. Mesajul afișat este cel așteptat, deci comportamentul neautorizat este gestionat corect.

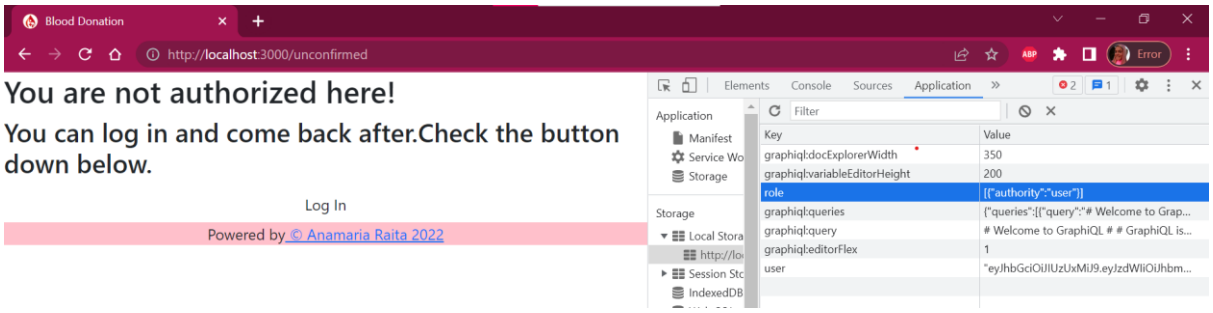


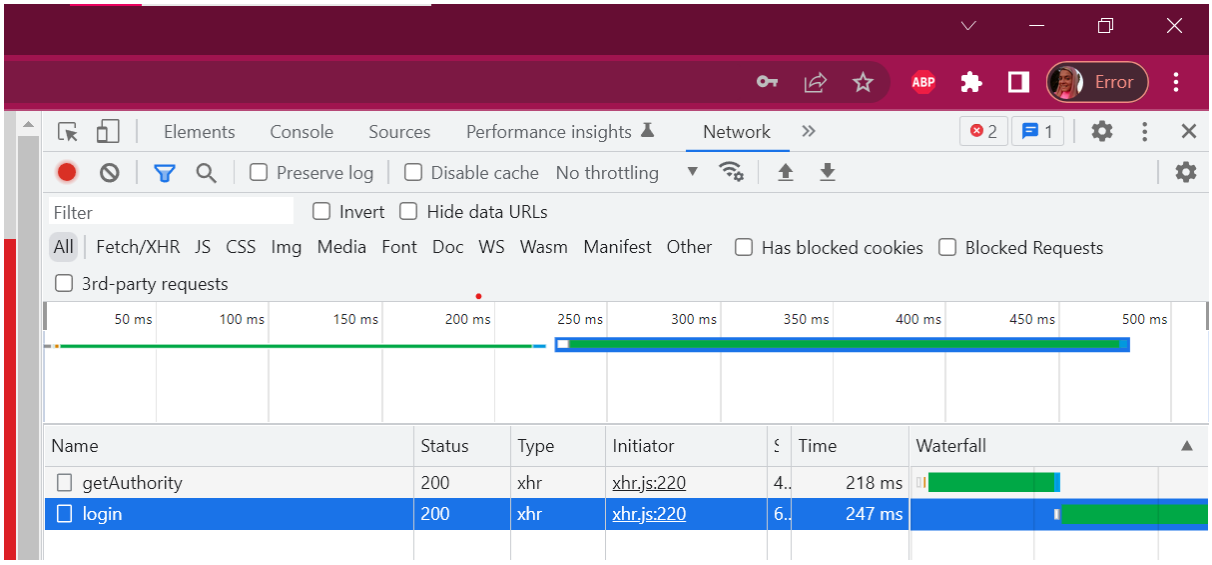
Figura 27 - Testare interfață - Acces neautorizat

4.1.4 Testare a performanței

Testarea de performanță (conform Black Rex et al 2012) în cazul particular al aplicației mele se referă la viteza de trimitere a cererilor și primire a răspunsurilor. Am ales să testez și acest lucru deoarece printre cerințele non-funcționale ale aplicației se află și o cerință legată de viteza de răspuns.

Așadar, cu ajutorul Developer Tools am reușit să măsor vitezele anumitor tipuri de cereri și a răspunsurilor lor. Deoarece este destul de greu să măsor timpii tuturor cererilor aplicației, am ales să măsor timpul cererii POST de log-in, ai unei cereri de DELETE și ai unei cereri de GET, în acest caz alegând obiectul cu cele mai multe înregistrări în baza de date.

Pentru metoda de tip POST, am ales să testez cererile pentru log in și getAuthority, aceasta din urmă fiind responsabilă cu identificarea rolului utilizatorului. Deoarece în procesul de autentificare sunt făcute ambele, am decis să le analizez împreună. După cum se poate observa, fiecare dintre ele se execută în aproximativ 0.2 secunde și ele se execută secvențial, întreg procesul de autentificare durează aproximativ 0.5 secunde, ceea ce se încadrează în parametrii stabiliți în cerințe.



Pentru metoda de tip DELETE exista o singură variantă de testare a acesteia în aplicație, și anume ștergerea unei programări. După cum putem observa, și această cerere se execute într-o secundă, ceea ce este un timp bun.

Figura 28 - Cerere POST viteză de răspuns

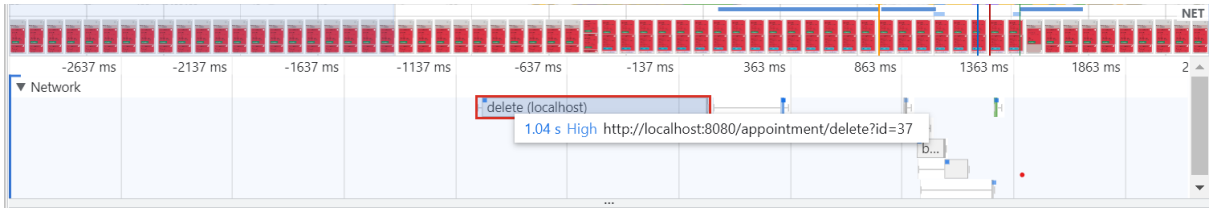


Figura 29 - Cerere DELETE viteză de răspuns

La cererea de tip GET, am ales să analizez situația în care extragem din baza de date programările neconfirmate dintr-un spital. Pentru cererea în cauză, numărul de înregistrări existente era de 10, iar timpul de execuție este de 0.5 secunde. Așadar, chiar și în cazul în care vom avea pe viitor mai multe date, în pagină ele vor fi încărcate în grupuri de câte 50 să zicem, așadar timpul de încărcare va fi sub 3 secunde și în acest caz.

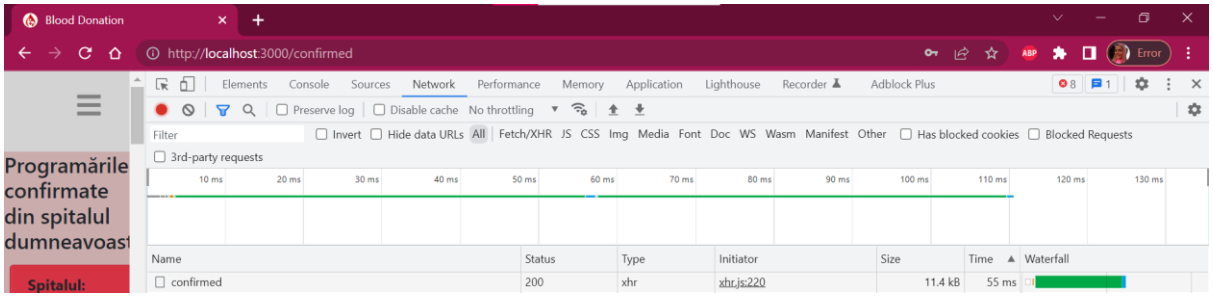


Figura 30 - Cerere GET viteză de răspuns

Ca o concluzie a capitolului de Asigurare a Calității aș spune că atât cerințele funcționale, cât și cele non-funcționale pe care le-am stabilit la început au fost îndeplinite. Deși consider că am efectuat o testare destul de riguroasă, niciodată nu putem fi siguri că aplicația nu are defecte. Cu toate acestea, pe parcursul dezvoltării am întâlnit diferite erori. Cele mai frecvente erori din interfață făceau referire fie la modul în care trimiteam token-ul în headerul de la cererile HTTP, fie modul în care parcurgeam răspunsul primit în format JSON sau modul în care alegeam să aranjez datele în cererile de tip GET sau POST din axios. Am avut și anumite dificultăți din punct de vedere al aranjării în pagină, dar acelea au fost ușor de reperat și rapid de remediat.

În ceea ce privește partea de backend din Java, erorile cele mai frecvente s-au datorat relațiilor din baza de date, deoarece uneori când salvam anumite înregistrări nu procedam corect cu modul în care făceam actualizări ale obiectelor cu care relaționa. Toate defectele din Java au fost descoperite destul de ușor, deoarece din cauza neconcordanței între ceea ce doream să salvez și ceea ce se salva îmi dădeam seama că este o problemă. Am testat oricum incremental pe parcursul dezvoltării aplicației, iar testarea finală a avut rol atât ca testare de acceptanță, pentru a vedea că am îndeplinit toate cerințele, cât și pentru a observa că din cauza dimensiunii aplicației nu am trecut nimic cu vederea.

Concluzii

Această lucrare abordează întreg procedeul pe care l-am parcurs pentru pentru a implementa un website funcțional, care ar putea rezolva o problemă prezentă în lumea înconjurătoare. Ceea ce putem constata este că atât demersul tehnic, cât și demersul non-tehnic anterior sunt la fel de importante. Este important să intrăm în contact cu publicul țintă, să aflăm ce așteptări are și să folosim metodologii care ne permit flexibilitate și adaptabilitate la ceea ce își doresc clienții. De asemenea, este important modul în care ne alegem arhitectura sistemului, deoarece cu cât reușim în implementare o modularizare mai mare și o separare a funcționalităților, cu atât mai ușor putem refolosi diferite componente și putem extinde aplicația introducând eventuale noi funcționalități.

Consider că pentru eventuale direcții viitoare ar fi important ca atenția să cadă asupra unor detalii vitale pentru utilizator. Ceea ce mi-am dorit în această lucrare a fost să reușesc să implementez în mare toate funcționalitățile de care am constatat că este nevoie. Pentru viitor mi-ar plăcea să rafinez aceste funcționalități, de exemplu în intermediul apelurilor la donare să filtrăm după grupa de sânge. De asemenea, o funcționalitate bazată pe generarea unor punctaje și urmate de premii pentru donatorii activi ar fi o idee de viitor potrivită. De asemenea, pe partea de business, ca idee de viitor ar fi util identificarea unor parteneri de talie mare prin intermediul cărora să promovez aplicația. De asemenea, aș dori să testez și anumiți parametrii legați de capacitatea aplicației și să dezvolt în acel sens, să suporte mulți utilizatori simultan.

Consider că puțin rafinat, ar fi un produs software reușit care ar reuși să integreze majoritatea funcționalităților pentru ambele părți: donator și spital și care ar putea contribui la rezolvarea acestei probleme: interesul scăzut pentru donarea de sânge.

Bibliografie

Black Rex, Graham Dorothy, Van Veenendaal Erik(2012), *Foundations of Software Testing – ISTQB®Certification*, Editura Cengage Learning, SUA

Martin Robert (2018*), Clean Architecture*, Editura Prentice Hall, SUA

Richards Mark (2015), *Software Architecture Patterns*, Editura O'Reilly Media, Inc, SUA

Sommerville Ian (2011), *Software engineering*, Editura Boston: Pearson, SUA

Watson Mike (2006), *Managing Smaller Projects: A Practical Approach*, Editura Multi-Media Publications Inc, SUA

Weisfeld Matt (2009), *The Object-Oriented Thought Process, Third Edition*, Editura Addison-Wesley, SUA

*** "Manifesto for Agile Software Development" (2001), http://agilemanifesto.org/ 2001, consultat la data de 3.06.2022

*** ""What is a Pareto Chart? Analysis & Diagram | ASQ", https://asq.org/quality-resources/pareto, consultat la data de 3.06.2022.

*** OMG (2011). OMG Unified Modeling Language (OMG UML), Superstructure, V2.4.1

***_Distributed Application Architecture_, https://web.archive.org/web/20110406121920/http://java.sun.com/developer/Books/jdbc/ch07.pdf , consultat la data de 1.05.2022

*** Documentație oficială Spring,https://docs.spring.io/springframework/docs/current/reference/html/ , consultat la data de 1.06.2022

***Documentație oficială MySQL, https://dev.mysql.com/doc/ , consultat la data de 1.06.2022

***Documentație oficială Java, https://docs.oracle.com/en/java/ , consultat la data de 1.06.2022

***Documentație oficială ReactJs, https://reactjs.org/ , consultat la data de 1.06.2022

***Documentație oficială JPA Repository, https://docs.spring.io/spring-data/jpa/docs/1.6.0.RELEASE/reference/html/jpa.repositories.html, consultat la data de 16.04.2022

***Documentație oficială Axios, https://axios-http.com/docs/intro , consultat la data de 25.04.2022

*** A Comparison of Database Drivers for MySQL, https://www.cdata.com/kb/articles/mysql-comparison-2020.rst ,consultat la data de 10.04.2022

Anexe

Anexa 1. Chestionar aplicat potențialilor beneficiari

Titlu chestionar: Blood Donation Management System

Detalii: Aplicația își dorește să automatizeze si faciliteze procesul de donare al sângelui atât din punct de vedere al donatorului, cât și din punct de vedere al spitalului/medicilor. În principiu, să fie o interfață între acestia și să contribuie la sporirea interesului oamenilor pentru a dona sânge și a salva vieți.

Întrebarea 1: Ați mai donat sânge înainte?

Răspunsuri: Da

Nu

Întrebarea 2: Care considerați că este cauza principală a lipsei interesului înspre donarea de sânge?

Răspunsuri: Neinformarea cu privire la beneficiile donării de sânge

Lipsa promovării procesului în sine de donare a sângelui

Frica de ace

Frica de faptul că acest process ar putea face rău

Lipsa încrederii în personalul medical

Durata de așteptare înainte și din cadrul procesului

Altele – introduse individual de către utilizator

Întrebarea 3: Cunoașteți faptul că o donare de sânge poate salva până la 3 vieți?

Răspunsuri: Da

Nu

Întrebarea 4: Cunoașteți faptul câ în urma donării se primesc bonuri de masă în valoare de 70 de lei și o zi liberă?

Răspunsuri: Da

Nu

Întrebarea 5: În cazul în care răspunsul la prima întrebare a fost Da, ați folosit vreodată orice fel de aplicație pentru a vă face programarea?

Răspunsuri: Da

Nu

Întrebarea 6: Dacă ați răspuns Nu la prima întrebare, care ar fi lucrul care v-ar determina să mergeți să donați sânge?

Răspunsuri: Întrebare deschisă, răspuns propriu pentru fiecare respondent.

Întrebarea 7: Credeți că o aplicație care să automatizeze procesul de programare la donare, primire a analizelor și care să vă trimită un reminder v-ar fi de folos în cazul în care vă gândiți să mergeți să donați?

Răspunsuri: Da

Nu

Întrebarea 8: În opinia dumneavoastră, care ar fi cel puțin 2 lucruri care se pot face prin intermediul acestei aplicații?

Răspunsuri: Întrebare deschisă, răspuns propriu pentru fiecare respondent.

Întrebarea 9: Ați utiliza o astfel de aplicație?

Răspunsuri: Da

Nu

Întrebarea 10: Indiferent de răspunsul anterior, în secțiunea următoare vă rog motivați-l.

Răspunsuri: Întrebare deschisă, răspuns propriu pentru fiecare respondent.

Întrebarea 11: Care este sexul dumneavoastra?

Răspunsuri: F

M

Întrebarea 12: Care este vârsta dumneavoastră?

Răspunsuri: 18-24

25-35

35-50

51-60

Întrebarea 13: Ultimul nivel de educație complet

Răspunsuri: Școala primară

Gimnaziu

Liceu

Facultate

Postliceală