



Universitatea  
Transilvania  
din Braşov  
FACULTATEA DE MATEMATICĂ  
ŞI INFORMATICĂ

# **LUCRARE DE LICENŢĂ**

## **Proiectul “Take care”**

**Absolvent:**  
**Coordonator ştiinţific:**

**Şerban Elena Mădălina**  
**Conf. Dr. Răzvan Bocu**

**BRAŞOV, 2023**

**Universitatea Transilvania din Brașov**

**Facultatea de Matematică și Informatică**

**Program de studii: Informatică**

## CUPRINS

<b>1.Introducere .....</b>	<b>5</b>
1.1 Scopul Proiectului .....	5
1.2 Motivația alegerii temei .....	6
1.3 Obiective propuse.....	7
1.4 Contribuții personale .....	8
1.5 Arhitectura generală a proiectului .....	9
1.6 Baza de date.....	10
1.6.1 Servicii Cloud .....	10
1.6.1 Firebase.....	11
<b>2. Aplicația Android .....</b>	<b>14</b>
2.1 Noțiuni teoretice .....	14
2.1.1 Limbajul java .....	14
2.1.2 Platforma Android Studio .....	15
2.1.2.1 Despre arhitectura Android Studio .....	15
2.1.3 Activități .....	17
2.1.4 Fragmente.....	18
2.1.5 Fișiere XML .....	19
2.2 Scopul aplicației Android.....	20
2.3 Integrarea cu Google Maps API .....	20
2.3.1 Integrarea în Android Studio.....	21
2.3.2 Afișare locației curente .....	22
2.4 Afișarea spitalelor și a rutelor .....	23
2.4.1 Citirea spitalelor din baza de date.....	23

2.4.2 Marcarea spitalelor pe hartă .....	25
2.4.3 Desenarea traseelor .....	29
2.4.4 Găsirea celui mai apropiat spital .....	31
2.4.5 Informații suplimentare despre spitale .....	32
2.5 Conectarea la internet – Broadcast receiver.....	34
2.6 Logarea utilizatorilor .....	37
2.7 Pagina de profil .....	38
<b>3. Aplicația Raspberry .....</b>	<b>41</b>
3.1 Despre Raspberry Pi .....	41
3.1.1 Conectarea ssh .....	42
3.2 Despre RFID .....	43
3.2.1 Conectarea hardware.....	43
3.2.2 Cardurile RFID .....	46
3.3 Conectarea la Firebase și biblioteci folosite .....	46
3.4 Codul aplicației .....	47
<b>4. Aplicația WEB.....</b>	<b>50</b>
4.1 Despre Html, Css și Javascript.....	50
4.2 Pagina principală .....	52
4.3 Modificarea și adăugarea detelor .....	55
<b>5.Concluzii .....</b>	<b>61</b>

# Capitolul 1

## INTRODUCERE

### 1.1 Scopul proiectului

Comunicarea între dispozitive și accesul instantaneu la informații în timp real reprezintă o cerință în creștere continuă, cu un impact semnificativ în domeniul medical.

Prin proiectul **“Take Care”** propunem îmbinarea mai multor tehnologii în vederea automatizării gestionării datelor din spitale.

În prima parte a proiectului avem în vedere implementarea unei aplicații pentru sistemul de operare Android cu ajutorul căreia utilizatorii își pot vedea locația curentă, cel mai apropiat spital, numărul de locuri disponibile din spitale în funcție de secție și rute către acestea. Aplicația mobilă este însoțită de un Raspberry 3B+ împreună cu un modul RFID RC522 ce actualizează în timp real atât numărul de locuri libere din spitale cât și istoricul pacientului.

Pentru partea administrativă, a fost dezvoltată o aplicație web care are la bază gestionarea și actualizarea datelor pacienților. Prin intermediul acestei aplicații urmărim stocarea centralizată a datelor pacienților și eficientizarea timpului acordat de personalul medical privind diagnosticarea și tratarea pacienților.

Distribuția cardului național de sănătate, care a început pe data de 19 septembrie 2014<sup>1</sup> pentru persoanele asigurate cu vârsta de peste 18 ani, marchează un pas semnificativ în modernizarea sistemului de asigurări de sănătate, o inițiativă ce a avut ca scop înființarea unui sistem mai eficient de gestionare a informațiilor și îmbunătățirea accesului la serviciile medicale.

Prin proiectul "Take Care", am propus accesul și monitorizarea pacienților în diferite secții ale spitalelor pe baza cardului de sănătate activat. Cardul de sănătate aflat în uz curent stochează date precum: numele și prenumele asiguratului, CID<sup>2</sup>, data nașterii și termenul de valabilitate al cardului, dar pentru utilizarea aplicațiilor propuse singurele date necesare sunt numărul de identificare al cardului, restul informațiilor fiind stocate în baza de date din cloud.

Prin folosirea aplicațiilor implementate se propune simplificarea procesul de înregistrare, autentificare și identificare a pacienților, astfel reducându-se nevoia de documente fizice și timpul de așteptare.

## 1.2 Motivația alegerii temei

Ideea temei de proiect a pornit de la dorința de a contribui la modernizarea sistemului de sănătate și a îmbunătăți accesul la servicii medicale de calitate.

Contextul pandemic din 2019 a dus la o revoluție tehnologică binevenită în domeniul medical.

Prin implementarea centrelor de testare și vaccinare cu programări online, generarea buletinelor de vaccinare și raportarea în timp real a numărului de cazuri pacienții și medicii au fost mai conectați ca niciodată.

"Până la sfârșitul lunii octombrie 2020, a fost creat un sistem electronic de informații pentru a îmbunătăți comunicarea dintre laboratoare, autoritățile locale din domeniul sănătății publice, medicii de familie și pacienți. Prin urmare, testele de diagnosticare sunt prelucrate în 24 de ore, rezultatele fiind trimise automat prin e-mail și mesaj text persoanei testate și medicului său de familie." (State of Health in the EU Romania, 2021)

---

<sup>1</sup> Conform CNAS

<sup>2</sup> Numărul de identificare al cardului național de asigurări sociale de sănătate

Cu toate acestea, pentru pacienții obișnuiți, lucrurile au rămas în mare parte neschimbate. Orice problemă medicală necesită în continuare resurse adecvate și asistență personală pentru a fi gestionată eficient.

În ciuda evoluției tehnologice specifice timpului actual, există încă provocări în asigurarea unei experiențe optime pentru pacienți. De multe ori, aceștia se confruntă

cu dificultăți în accesarea serviciilor medicale, obținerea informațiilor, sau gestionarea propriilor date medicale.

Prin realizarea acestui proiect am propus creșterea posibilității pacientului de a lua decizii cât mai informate în ceea ce privește accesul la servicii medicale. Prin accesul la numărul de locuri disponibile acesta este capabil să își estimeze timpul de așteptare și poate alege un alt spital disponibil din apropiere.

De asemenea, aplicația poate fi accesată de personalul medical în cazuri excepționale de urgențe : calamități, incendii su accidente multiple pentru a aduce pacienții la un centru cât mai benefic pentru nevoile lor.

Prin utilizarea tehnologiei și a comunicării dintre dispozitive, doresc să creăm un mediu mai eficient și conectat, care să aducă beneficii semnificative atât pentru pacienți, cât și pentru profesioniștii din sănătate.

### 1.3 Obiective propuse

Primul pas în cadrul oricărui proiect de tip software este cel de stabilire a obiectivelor propuse. Acest proces implică identificarea și definirea clară a funcționalităților și așteptărilor pe care proiectul trebuie să le îndeplinească.

Proiectul „Take Care” are ca obiective:

- 1. Realizarea unei interfețe intuitive si prietenoase pentru utilizator.** Se urmărește dezvoltarea unor aplicații mobile și web ce oferă o experiență intuitivă și ușor de utilizat atât pentru utilizator cât și pentru personalul medical și administrativ.
- 2. Prelucrarea și afișarea datelor in timp real.** Aplicațiile concepute au ca scop preluarea și afișarea în timp real a informațiilor relevante pentru utilizator cum ar fi: secțiile spitalelor,

numarul de locuri libere și rutele catre acestea împreuna cu propriul log medical cu vizite înregistrate.

**3. Conectarea diferitelor tehnologii utilizate.** Pentru integrarea tuturor tehnologiilor folosite și comunicarea eficientă a acestora se utilizează platforma cloud Firebase. Astfel ni se permite schimbul de date în timp real și stocarea în siguranță a informațiilor.

**4. Sugestia de rute către cele mai potrivite centre medicale** utilizând cereri către API-uri Google Maps și numărul de locuri libere cunoscut. Aplicația mobilă utilizează request-uri către Directions API de la platforma Google Maps pentru a furniza rute optime către spitalele disponibile ținând cont de locația utilizatorului, secția aleasă și numărul de locuri libere.

**5. Informarea avizată a asigraților medicali cu privire la propriile date de sănătate și a centrelor medicale disponibile.** Prin transparența datelor pacienților se propune creșterea implicării acestora în gestionare propriilor nevoi de sănătate și comunicarea mai eficientă cu personalul medical.

## 1.4 Contribuții personale

Diferit de simpla utilizare a platfomelor ca "Google Maps" pentru a afla ruta către cel mai apropiat spital aplicația propusă încearcă formarea unei rețele între utilizator și centrele medicale de specialitate.

Propunem conectarea cardului de sănătate la o aplicație mobilă, mereu la îndemana utilizatorului. Pentru cazurile de urgență bonusul acestui proiect este afișarea rutei către cel mai apropiat spital în ecranul principal fără a fi necesară apăsarea vreunui buton și fără utilizarea animațiilor sau logo-urilor înainte de încărcarea paginii principale, astfel punând accent pe criteriul de urgență în obținerea datelor.

Pacienții pot fi informați despre specializările, serviciile și condițiile acestor unități medicale, permițându-le să aleagă cea mai potrivită opțiune pentru nevoile lor de sănătate. Astfel aceștia pot alege un centru medical în funcție de distanță, reputație și accesibilitate.

O altă caracteristică este legată de profilul utilizatorului actualizat cu informații medicale direct de la sursă. Deși nenumărate aplicații medicale oferă posibilitatea de a adăuga și reține informații personale, acestea necesită ca utilizatorul să cunoască și să fi memorat



deja toate aceste date. Prin aplicația propusă este necesar doar să ne logăm în contul deja actualizat și administrat de personal specializat prin codul unic al cardului de sănătate și prinul acestuia.

În plus, aplicația în cauză permite utilizatorilor să își vizualizeze vizitele din spital după data de externare sortate în funcție de secție.

## 1.5 Arhitectura generală a proiectului

Arhitectura este un aspect important în dezvoltarea oricărui proiect software și are un impact mare asupra succesului și performanței oricărei aplicații. Putem spune că arhitectura este fundația pentru dezvoltarea și implementarea oricărei aplicații de tip software și oferă acestora o structură clară și coerentă.

În acest capitol ne vom axa pe modul în care componentele sistemului descris lucrează împreună, cum interacționează între ele și modul acestora de organizare.

Prin definirea clară a componentelor și a relațiilor dintre ele, putem înțelege fluxul de date și comunicarea între aplicația web, aplicația Android și cea utilizată de Raspberry Pi. Vom explora modalitățile prin care aceste componente interacționează și transmit datele necesare pentru funcționarea corectă a sistemului.

În acest proiect, arhitectura joacă un rol crucial în asigurarea unei interacțiuni eficiente între aplicații. Fiecare componentă are propriul rol specific și schimbă informații cu celelalte prin conectarea la platforma Firebase pentru a facilita stocarea și sincronizarea datelor în timp real.

Aplicația web este destinată personalului calificat din spitale și are ca scop administrarea și actualizarea datelor pacienților. Personalul medical și administratorii spitalelor pot accesa platforma web pentru a gestiona informațiile pacienților, inclusiv înregistrarea și actualizarea datelor sau activarea/dezactivarea cradurilor din baza de date.

Aplicația Android oferă utilizatorilor acces la informații despre spitalele din baza de date printre care: numărul de locuri disponibile în funcție de secția selectată, o rută de la locația curentă și ratingul conform Google Maps. Utilizatorii pot utiliza cardul de sănătate pentru autentificare și vizualizare a datelor personale.

Scopul dispozitivului Raspberry Pi, împreună cu modulul RFID RC522 este de a actualiza în timp real numărul de locuri libere din spitale și de a păstra istoricul vizitelor fiecărui pacient. Dispozitivul actualizează aplicația mobilă prin intermediul platformei Firebase, astfel asigurând sincronizarea și actualizarea în timp real a informațiilor.

Platforma Firebase este utilizată pentru a trimite datele către toate componentele sistemului. Aceasta oferă o modalitate rapidă și eficientă de colectare, stocare și trimitere de date între toate dispozitivele montate.[figura1.5.1]

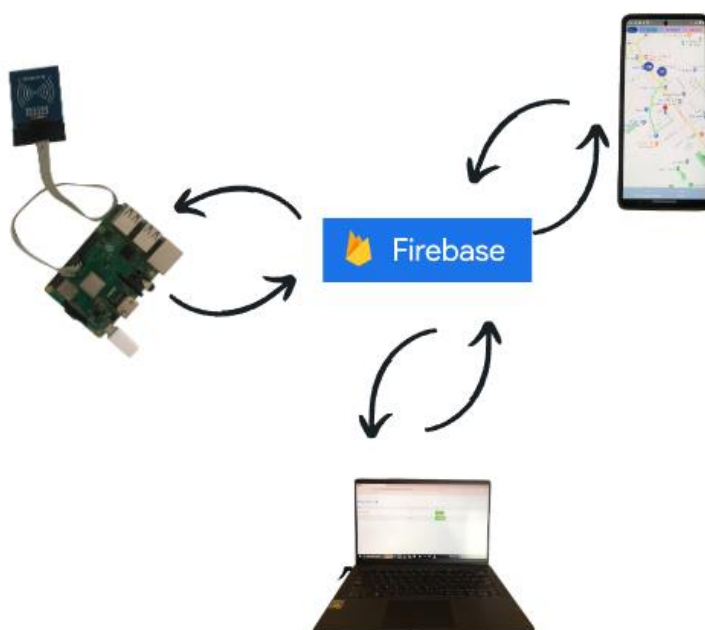


figura 1.5.1

Arhitectura discutată mai sus asigură o interacțiune eficientă între utilizatori și instituțiile medicale, facilitând astfel accesul la informații actualizate.

În capitolele ce urmează vom detalia în profunzime fiecare componentă a arhitecturii și vom evidenția interacțiunile și fluxurile de date dintre acestea.

## 1.6 Baza de date

### 1.6.1 Servicii Cloud

Odata cu creșterea conexiunii constante la internet dar și a nevoii de stocare și gestionare a datelor pe dispozitivele personale termenul de "cloud" a devenit unul din ce în ce mai

comun. Cel mai probabil, mulți cunosc acest termen din notificările periodice de pe telefoanele mobile sau chiar din abonamentele lunare către astfel de platforme.

Dar la ce se referă mai exact "cloud"?

Termenul de cloud reprezintă, general, un model de furnizare a serviciilor software prin intermediul internetului, astfel companiile pot alege să nu mai investească în resurse locale (software sau hardware) ci să plătească pentru servicii cloud în funcție de cât folosesc.

În cazul proiectului nostru platformele de tip cloud sunt esențiale atât pentru comunicarea între dispozitive cât și pentru stocarea permanentă a datelor.

### 1.6.2 Firebase

Platforma Firebase, inițial sub numele de „Envolv”, a fost implementată în anul 2011 de către James Tamplin și Andrew Lee și era dedicată integrării funcționalității de chat online în site-urile web ale utilizatorilor.

Ulterior, în anul 2014, Google a achiziționat platforma și i-a extins funcționalitățile.

În prezent, Firebase oferă o gamă largă de funcționalități, inclusiv stocare în cloud, autentificarea utilizatorilor, analize, mesagerie în timp real și multe altele.

Pentru proiectul descris am conectat cele trei aplicații (Android, web și Raspberry) la același proiect Firebase. Acest lucru a fost posibil prin utilizarea SDK<sup>3</sup>-ului specific Firebase pentru fiecare platformă.

Prin conectarea celor trei aplicații la același proiect Firebase am putut să partajăm date și să sincronizăm informațiile între aplicații în timp real. Astfel, orice modificare a datelor efectuată într-una dintre aplicații este reflectată automat în celelalte, oferind o experiență coerentă pentru utilizatori.

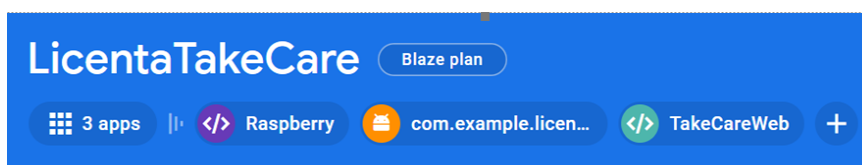


figura 1.6.1

Funcționalitățile Firebase folosite pentru proiectul discutat sunt : Realtime Database și Firestore.

Realtime Database este o bază de date NoSQL ce stochează ,în cazul nostru, datele specifice cardurilor de sănătate. Stocarea se face într-un fișier de tip JSON iar datele sunt organizate în noduri.

În cadrul proiectului nostru, avem un nod principal numit "UserCard" ce conține sub-noduri pentru fiecare card de sănătate. Fiecare card de sănătate este identificat printr-un ID unic, generat automat, și conține la randul său un alt subnod numit log ce reține înregistrările cardului.

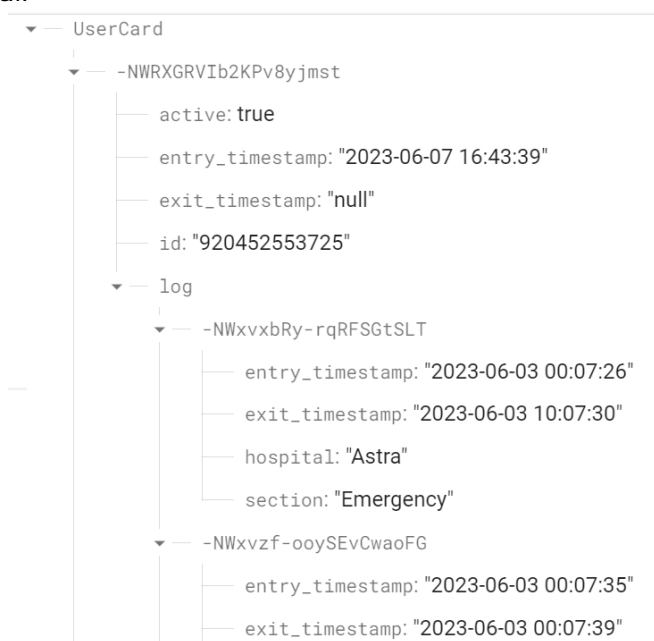


figura 1.6.2 –nodul „UserCard” din Realtime Database

Firestore, pe de altă parte, este tot o bază de date NoSQL dar ce utilizează colecții și documente pentru organizarea datelor.

În proiectul nostru Firestore a avut rolul de a gestiona datele spitalelor și a utilizatorilor. Deoarece datele personale ale utilizatorilor nu se schimbă des am considerat Firestore cea mai bună alegere.

Diferit de Realtime Database, Firestore este mai ușor de utilizat în ceea ce privește gestionarea și structurarea datelor. Pentru gestionarea informațiilor despre spitale, un avantaj al acestei baze de date a fost utilizarea tipul de date „GeoPoint”. Astfel am putut stoca coordonatele geografice ale fiecărui spital, ceea ce ne-a permis să construim funcționalități legate de localizare și distanță în aplicațiile noastre.

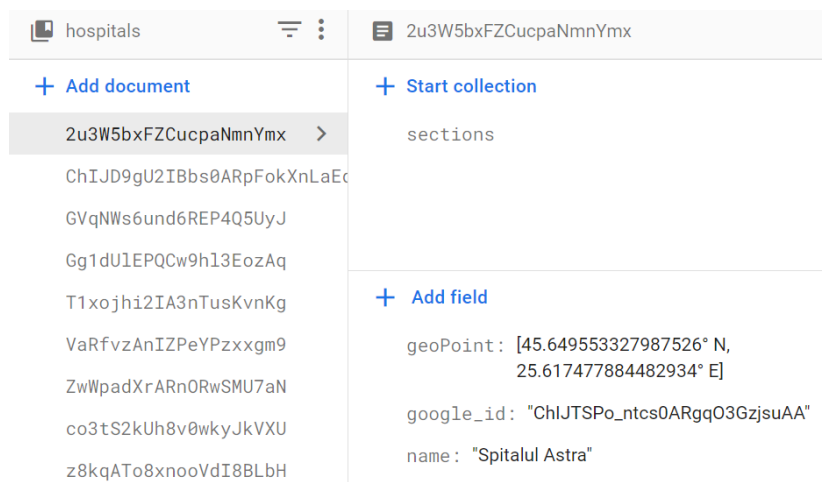


figura 1.6.3 – colecția „hospitals” din Firestore

## Capitolul 2

### APLICAȚIA ANDROID

#### 2.1 Noțiuni teoretice

**Android** este un sistem de operare bazat pe Linux, conceput în principal pentru dispozitive mobile cu touchscreen, precum smartphone-uri și tablete. A fost dezvoltat de către "Open Handset Alliance" și sponsorizat comercial de Google.

##### 2.1.1 Limbajul java

**Java** este un limbaj de programare bazat pe clase, puternic orientat pe obiecte, și utilizat pentru o gamă largă de aplicații software. Limbajul a fost lansat de James Gosling și Sun Microsystems în anul 1995 iar apoi achiziționat de Oracle Corporation în anul 2010.

Deși la prima vedere observăm asemănări majore între limbajele Java și C++ putem considera Java un limbaj mult mai prietenos cu multe avantaje cum ar fi:

- Managementul automat al memoriei : Java introduce termenul de "garbage collection"<sup>4</sup> prin care programele efectuează gestionarea automată a memoriei. Alocarea

---

<sup>4</sup> Colectarea automată a gunoiului

de memorie se face general prin operatorul "new" și apoi trimise în "heap"<sup>5</sup>. Atunci când un obiect devine inaccesibil este detectat și eliberat automat din memorie.

- Portabilitatea: codul scris în limbajul Java poate fi compilat într-un format intermediar numit "bytecode", care poate fi apoi executat pe orice mașină virtuală Java fără a necesita modificări de cod. Asta înseamnă că aplicațiile Java pot fi dezvoltate o singură

dată și apoi pot fi rulate pe diferite sisteme de operare, cum ar fi Windows, macOS sau Linux, fără a necesita rescrierea codului sursă.

- Bibliotecă standard bogată: reușim să utilizăm o bibliotecă extinsă sub numele de "Java Standard Library" ce acoperă o gamă largă de funcționalități. Prin intermediul acestei biblioteci avem acces la o multitudine de funcționalități comune și reutilizabile.

Limbajul Java stă la baza sistemului de operare Android, care alimentează de departe cea mai mare parte a smartphone-urilor din lume.

Google a dezvoltat un kit de dezvoltare software Android, care include un set complet de instrumente, biblioteci și documentație pentru a dezvolta aplicații.

Kitul de dezvoltare include și o mașină virtuală Java personalizată, cunoscută sub numele de Dalvik sau ART (Android Runtime) ce permite dezvoltatorilor să creeze și să ruleze aplicații în acest limbaj.

## 2.1.2 Platforma Android Studio

Android Studio este un mediu integrat de dezvoltare (IDE) specializat pe dezvoltarea aplicațiilor pentru sistemul de operare Android. Mediul de programare este construit pe baza platformei IntelliJ IDEA, un alt IDE popular utilizat în dezvoltarea aplicațiilor software.

### 2.1.2.1 Despre arhitectura Android Studio:

Arhitectura unui proiect în Android Studio implică de obicei utilizarea a două module principale: modulul de aplicație (app module) și modulul de bibliotecă (library module).

#### **App (Modulele aplicației)**

---

<sup>5</sup> Spațiu de memorie pentru obiecte create dinamic în timpul rulării programului

În Android Studio, modulele proiectului sunt organizate astfel încât să ofere un acces rapid și o gestionare eficientă a fișierelor sursă și cheilor proiectului. Fiecare modul al bibliotecilor este situat la nivelul fișierelor Gradle Scripts și conține următoarele directoare

- "manifests": Acest director conține doar fișierul AndroidManifest.xml al modulului respectiv ce conține informații esențiale despre aplicație și specificațiile sale cum ar fi: permisiunile necesare, descrierea aplicației și configurări ale dispozitivului.
- "java": Conține fișierele sursă Java ale modulului. Aici scriem codul aplicației și implementăm toate funcționalitățile inclusiv cele de testare.
- "res": Acest director conține toate resursele modulului. Acestea pot include fișiere XML, imagini, animații, coduri pentru culori, navigația aplicației, șiruri de caractere și alte resurse utilizate în interfața grafică a aplicației.

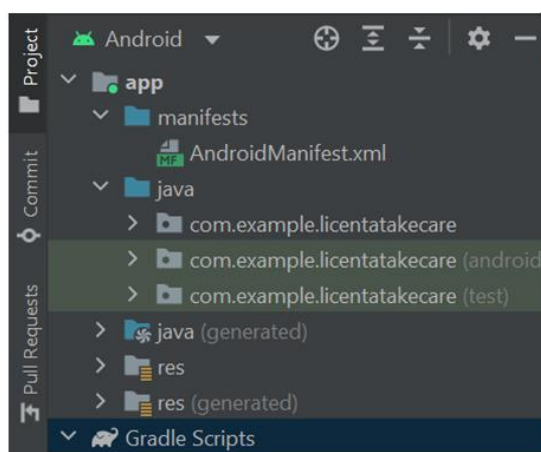


figura 2.1.1 –Arhitectura Android Studio

### Modulul Gradle (modulele bibliotecii) :

Reprezintă fișiere de configurare care folosesc limbajul de scripting Groovy pentru a defini și gestiona dependențele, setările și procesul de compilare al aplicației. Gradle este un sistem de automatizare a construirii (build automation) și de gestionare a dependențelor, utilizat în dezvoltarea aplicațiilor Android.[figura2.1.2]



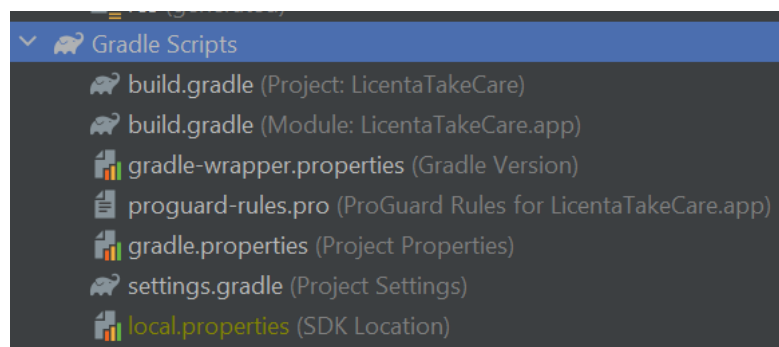


figura 2.1.2

În cadrul unui proiect Android, folosim două dintre acest tip de fișiere și anume:

1. Fișierul `build.gradle(Project)`: conține configurări și setări globale pentru întregul proiect. Aici se pot specifica informații precum versiunea plugin-ului Android Gradle, dependențele globale, configurația de compilare și alte setări globale.
2. Fișierul `build.gradle(Module)`: conține configurări specifice pentru modulul respectiv. Aici se pot adăuga dependențe specifice modulului, se pot specifica setările de compilare, se pot configura opțiuni de construire, precum și alte setări specifice modulului.

### 2.1.3 Activități

În cadrul unei aplicații Android, o activitate reprezintă componente de bază a interfeței utilizatorului. Activitățile sunt cele care permit afișarea de conținut și interacționarea cu utilizatorul.

În proiectul nostru, am implementat două activități reprezentate prin clase Java ce extind clasa `'android.app.Activity'`. Interfața grafică este implementată prin intermediul fișierelor XML<sup>6</sup> numite fișiere de "layout", care definesc aspectul și organizarea elementelor interfeței utilizatorului în cadrul activității respective.

Fiecare activitate are un ciclu de viață foarte bine definit ce permit sistemului de operare să gestioneze aplicația în mod corespunzător, să păstreze starea acesteia și să reacționeze la evenimente specifice.

Ciclul de viață al unei activități poate fi reprezentat astfel :

---

<sup>6</sup> Extensible Markup Language

1. Activitatea este creată și inițializată prin apelul metodei `onCreate()`. Probabil cea mai comună dintre metode, aici se realizează configurarea inițială a activității, inclusiv setarea conținutului grafic și inițializarea resurselor necesare.
2. Activitatea intră în starea `onStart()`, ceea ce înseamnă că devine vizibilă utilizatorului, dar nu este încă interactivă. Aici se pot face pregătiri suplimentare, cum ar fi începerea anumitor procese sau inițializarea componentelor.
3. După `onStart()`, activitatea intră în starea `onResume()`, moment în care devine activă și interactivă. Aici utilizatorul poate interacționa cu activitatea și aceasta poate răspunde la evenimente și acțiuni.
4. În cazul în care utilizatorul părăsește activitatea sau o altă activitate intră în prim-plan, activitatea curentă intră în starea `onPause()`. Aceasta indică faptul că activitatea își pierde temporar focusul și devine parțial vizibilă. În acest moment, puteți salva date sau starea activității pentru a o putea restaura ulterior.
5. Dacă activitatea este complet acoperită de o altă activitate sau utilizatorul navighează înapoi, activitatea intră în starea `onStop()`. În această fază, activitatea nu mai este vizibilă utilizatorului și poate fi închisă complet sau reluată ulterior.
6. În final, dacă utilizatorul închide aplicația, este apelată metoda `onDestroy()`. Aceasta indică faptul că activitatea este închisă complet și resursele asociate pot fi eliberate.

Activitățile pot trece în mod repetat prin aceste stări, în funcție de interacțiunea utilizatorului și de evenimentele care au loc.

#### 2.1.4 Fragmente

În cadrul aplicațiilor Android, fragmentele sunt componente reutilizabile integrate într-o activitate. Asemănător activităților, fragmentele reprezintă o parte vizuală a aplicației ce permite afișarea de conținut și interacționarea cu utilizatorul. Ele pot permite proiectarea mai modulară a activităților și facilitează transmiterea datelor dintre ecrane.

Fragmentele au un ciclu de viață asemănător cu activitățile însă nu pot exista în afara acestora.

Împărțirea în fragmente oferă o ușoară navigare între pagini prin intermediul unui graf de navigare aflat în folderul de resurse. Acesta există atât sub forma grafică [figura2.1.3]

cât și sub formă de cod cu diverse funcționalități ca: trimiterea de date între fragmente, implementarea animațiilor etc.

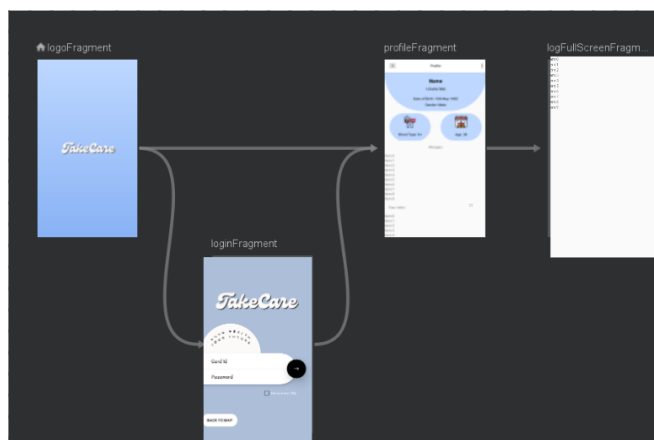


figura 2.1.3-navigația Activității MainActivity

### 2.1.5 Fișiere XML

Pentru realizarea acestui tip de fișiere este nevoie să cunoaștem “Extensible Markup Language”, un limbaj folosit pentru a defini și proiecta aspectul paginilor în cadrul activităților și fragmentelor. Scopul fișierelor XML este de a defini interfața utilizatorului Android și a ne reprezenta aplicația din punct de vedere vizual, asemănător cu modul în care limbajul HTML ajută la descrierea paginilor web.

Găsim fișierele XML în directorul “res”, subdirectorul “layout” și sunt specifice pentru fiecare ecran sau fragment al aplicației. Acestea sunt denumite în funcție de destinația sau scopul lor, cum ar fi “activity\_main.xml” pentru ecranul principal, “fragment\_login.xml” pentru fragmentul de logare sau “fragment\_list.xml” pentru o listă ce poate apărea într-un fragment sau activitate fără să ocupe neapărat întreaga pagină.

Orice fișier XML din Android Studio are asociate două ferestre principale: una pentru codul XML și una pentru vizualizarea design-ului.

Fereastra de cod ne permite editarea manuală a codului, unde definim structura, atributele și proprietățile elementelor UI<sup>7</sup> iar cea de design oferă o reprezentare grafică a aspectului paginii, astfel putem vizualiza și să modificăm elementele într-un cadru vizual.

<sup>7</sup> User Interface interfața care interacționează cu utilizatorul

De asemenea, în fișierele XML de layout, putem utiliza diferite elemente și atribute pentru a defini stiluri, dimensiuni, poziționarea, interacțiuni și multe altele. Aceste fișiere sunt esențiale pentru configurarea și personalizarea interfeței vizuale a aplicației Android și reprezintă o parte importantă a dezvoltării aplicațiilor mobile.

## 2.2 Scopul aplicației Android

Aplicația mobilă "Take Care" are ca scop principal facilitarea accesului utilizatorului la informații curente și personalizate.

Prin aplicația Android a proiectului "Take Care" ne propunem realizarea următoarelor funcționalități:

- Implementarea unei hărți interactive ce afișează locația utilizatorului.
- Afișarea locației spitalelor din baza de date împreună cu numărul de locuri libere.
- Găsirea celui mai apropiat centru medical și afișarea unei rute către acesta.
- Implementarea unui meniu interactiv care permite utilizatorului să vizualizeze locurile libere în funcție de secția selectată.
- Actualizarea traseelor în funcție de secția selectată.
- Colorarea traseelor în funcție de procentul de locuri libere din spitale și sugerarea unor trasee alternative.
- Actualizarea culorilor indicatoarelor spitalelor pentru fiecare secție în parte.
- Actualizarea locurilor libere în timp real.
- Afișare unor informații suplimentare despre spitale cum ar fi: numărul de telefon, adresa, ratingul (conform platformei GoogleMaps), timpul și distanța până la destinație.
- Logarea utilizatorului.
- Reținerea utilizatorului.
- Profilul utilizatorului cu datele personale și medicale ale acestuia (grupă sanguină, alergeni, vizite în spitale etc.)
- O listă ce conține vizitele din spitale în ordinea externării și colorate conform secției.
- Actualizarea în timp real a listei vizitelor din spitale.
- Delogarea utilizatorului.

## 2.3 Integrarea cu Google Maps API

Definim un API (Application Programming Interface) ca fiind un set de instrumente ce permit comunicare și interacționarea între diferite componente software.

Prin intermediul unui API, putem utiliza în aplicațiile noastre funcționalități și componente oferite de o companie software fără a fi necesar să înțelegem întreaga logică internă a acestora. API-urile furnizează metode, funcții și structuri de date prin intermediul cărora putem solicita serviciile de care avem nevoie.

În cazul integrării cu Google Maps este necesară obținerea unei chei de autentificare, un identificator unic ce va fi asociat proiectului nostru și pe care îl vom utiliza pentru cererile către platforma Google Maps.

După ce parcurgem toți pașii necesari pentru obținerea cheii API (descriși pe pagina Google for Developers) tot ce rămâne de făcut este să activăm serviciile API dorite din contul nostru "Google Developers" și să începem integrarea cu aplicația Android.

### 2.3.1 Integrarea cu Android Studio

Pentru a putea vizualiza harta este în aplicația Android începem prin adăugarea cheii unice în fișierul Android manifest în interiorul tagului <application> și a dependențelor specifice în cadrul fișierului build.gradle(Module).

Deoarece aplicația descrisă folosește mai multe servicii Google în fișierul nostru gradle se află următoarele dependențe:

```
//google maps
implementation 'com.google.android.gms:play-services-maps:18.0.0'
implementation 'com.google.maps.android:android-maps-utils:2.2.0'
//directions
implementation 'com.google.android.gms:play-services-location:17.0.0'
implementation 'com.google.maps:google-maps-services:0.15.0'
//retrofit
implementation 'com.squareup.retrofit2:retrofit:2.9.0'
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
```

figura 2.3.1

**com.google.android.gms:play-services-maps** este o dependență de bază pentru integrarea cu serviciul Google Maps. Aceasta oferă funcționalități esențiale pentru lucrul cu hărți, cum ar fi afișarea hărților, adăugarea de marcaje, gestionarea evenimentelor de interacțiune cu harta.

**com.google.maps.android:android-maps-utils** adaugă funcționalități suplimentare pentru lucrul cu hărți Google Maps pe dispozitive Android. Include utilități pentru lucrul cu marcatori, clustere de marcatori, desenarea poliliniilor și multe altele.

**com.google.android.gms:play-services-location** ne permite utilizarea serviciului de localizare. Aceasta furnizează metode pentru obținerea locației curente a dispozitivului și pentru monitorizarea schimbărilor de locație.

**com.google.maps:google-maps-services** permite adaugarea suport pentru comunicarea cu serviciile Google Maps, cum ar fi Directions API, Geocoding API și alte servicii oferite de Google Maps. Aceasta permite obținerea indicațiilor de direcție, geocodarea adresei etc.

**com.squareup.retrofit2:retrofit** și **com.squareup.retrofit2:converter-gson** sunt dependențe necesare pentru integrarea cu Retrofit, o bibliotecă populară pentru realizarea cererilor HTTP și gestionarea răspunsurilor în aplicații Android.

Pentru a utiliza harta în activitatea `MapsActivity`, vom implementa interfața `OnMapReadyCallback` și vom adăuga funcția specifică `onMapReady`. Această funcție este apelată atunci când harta este pregătită pentru a fi utilizată și ne oferă o referință către obiectul `GoogleMap`, prin intermediul căruia putem interacționa cu harta.

### 2.3.2 Afișare locației curente

Înainte de a implementa această funcționalitate este important să ne asigurăm că adăugăm referințele necesare în fișierul `AndroidManifest.xml`. În cazul aplicației noastre a fost nevoie următoarele linii de cod :

```
„<uses-permission android:name=\"android.permission.ACCESS_FINE_LOCATION\" />  
<uses-permission android:name=\"android.permission.ACCESS_COARSE_LOCATION\" />”
```

Aceste linii de cod ne oferă permisiunea de a accesa atât locația precisă, cât și locația aproximativă a dispozitivului.

Următorul pas este informarea utilizatorului cu privire la folosirea locației dispozitivului acestuia.

Funcția `getLocation()` din clasa `MapsActivity` este responsabilă pentru obținerea locației curente a utilizatorului. În primul rând, ea verifică dacă permisiunile necesare sunt acordate utilizând funcția `checkSelfPermission()`. Dacă permisiunile nu sunt acordate, se utilizează funcția `ActivityCompat.requestPermissions()` pentru a solicita permisiunea utilizatorului. În cazul nostru, specificăm atât permisiunea pentru locația exactă cât și pentru cea aproximativă. Rezultatul cererii de permisiune va fi gestionat în metoda `onRequestPermissionsResult()`.

Pentru aplicația dezvoltată este necesară locația exactă deci vom avea două cazuri după interacționarea cu utilizatorul: ne-a oferit sau nu locația exactă și vom naviga conform acestor informații. [figura 2.3.2]

Când un utilizator care nu a oferit locația precisă dorește să folosească una dintre funcționalitățile care au nevoie de aceste date va fi întrebat din nou dacă ne permite accesul către aceste date.

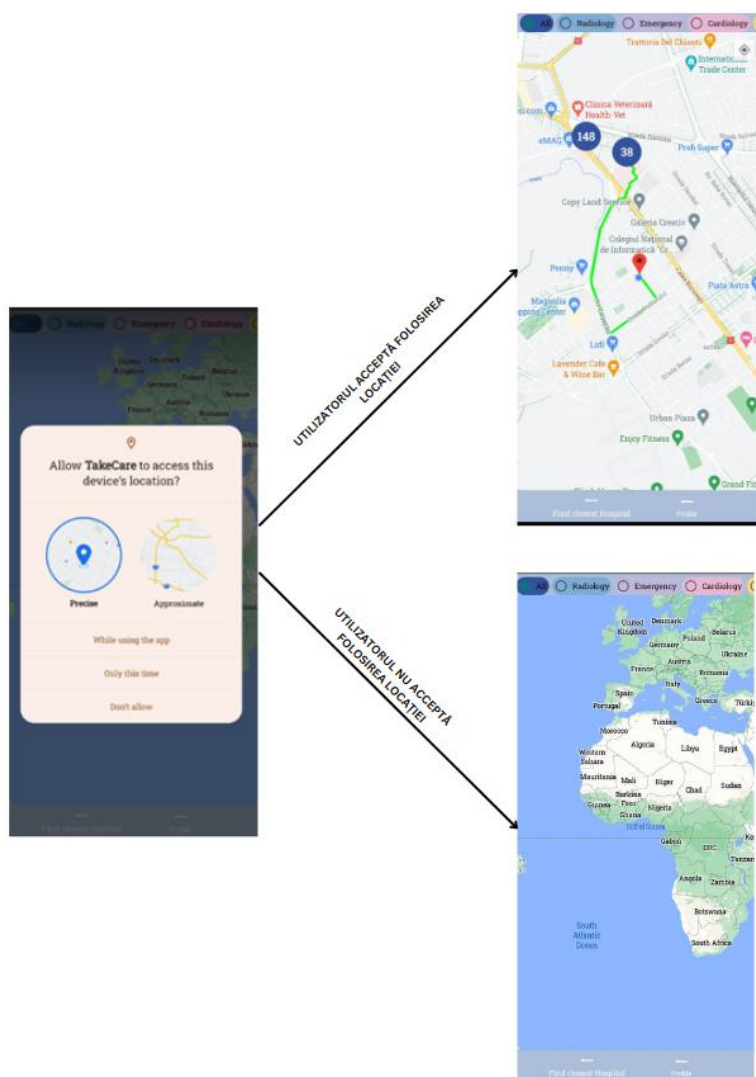


figura 2.3.2

## 2.4 Afișarea spitalelor și a rutelor

### 2.4.1 Citirea spitalelor din baza de date

Pentru obținerea datelor spitalelor din baza de date Firestore, aplicația folosește un mecanism numit „callback”. Asta înseamnă că operațiile de citire se desfășoară în fundal și nu blochează firul principal al aplicației, permițând în același timp continuarea funcționalităților aplicației ca: încărcarea hărții și localizarea utilizatorului.

Pentru gestionarea acestor date am început prin implementarea interfeței „HospitalsCallback” ce conține două metode: „onHospitalsRetrieved” pentru primirea listei de spitale și „onHospitalUpdated” pentru actualizarea datelor unui spital specific.

În interiorul metodei „onHospitalsRetrieved”, lista de spitale este asignată variabilei „mHospitals” pentru viitoare operații ce necesită aceste date iar apoi se apelează funcția „addHospitalsToMap” ce ne adăugă vizual spitalele pe hartă.

Atunci când se actualizează informațiile despre un spital în baza de date, este apelată metoda „onHospitalUpdated”. Această metodă se ocupă de actualizarea spitalelor existente pe hartă, astfel încât informațiile despre disponibilitatea locurilor să fie actualizate în timp real.

Pentru conectarea la baza de date am implementat clasa „HospitalsDao” în care se află funcția „getHospitalList” ce primește ca parametru interfața descrisă.

În cazul în care citirea spitalelor este realizată cu succes, se parcurg rezultatele și pentru fiecare document se obține un obiect de tip „Hospital”.

Pentru fiecare obiect de acest tip, obținem apoi o referință către colecția „sections” a fiecărui spital. La fel ca în cazul spitalelor, secțiile sunt parcurse și pentru fiecare document se obține un obiect de tip „Section”.

Secțiile sunt adăugate în lista de secții a spitalului curent, iar spitalul în sine este adăugat la lista de spitale.

Deoarece operațiile de citire a spitalelor și a secțiilor sunt realizate prin intermediul Firebase, care utilizează un fir de execuție separat pentru a efectua cererile către server, am ales să folosim un obiect de tip „AtomicInteger” numit „count” pentru a urmări numărul de spitale procesate. La fiecare completare a procesării unui spital și a secțiilor sale, numărul se decrementează. Când obiectul devine zero, înseamnă că toate spitalele au fost procesate și se apelează metoda „onHospitalsRetrieved” din callback, pentru a furniza lista completă de spitale către activitate.



De asemenea, se adaugă un listener de tip „SnapshotListener” pentru colecția „sections” a fiecărui spital. Acesta „ascultă” orice modificare la nivelul secțiilor și, în cazul în care există modificări, actualizează markerii corespunzători de pe hartă prin apelul metodei „onHospitalUpdated” din callback.

Astfel, prin utilizarea mecanismului de callback, aplicația poate citi și procesa datele din baza de date Firestore. Acest lucru asigură o interacțiune fluentă cu utilizatorul, în timp ce se obțin și actualizează informațiile despre spitale în timp real.

#### 2.4.2 Marcarea spitalelor pe hartă

Pentru afișarea unei iconițe personalizate și marcarea spitalelor pe hartă am ales implementarea unui obiect de tipul „BitmapDescriptor”, o clasă din „Google Maps Api” ce permite folosirea unei imagini bitmap personalizate pentru un marcator. [figura 2.4.1]

Obiectele de acest tip sunt atribuite markerelor pe hartă prin intermediul metodei „setIcon” a clasei Marker.

```
private BitmapDescriptor getMarkerIcon(ClusterMarker clusterMarker, int color) {
    TextDrawable drawable = TextDrawable.builder() TextDrawable.IShapeBuilder
        .beginConfig() TextDrawable.IConfigBuilder
        .textColor(color)
        .useFont(Typeface.DEFAULT_BOLD)
        .fontSize(50)
        .bold()
        .height(130)
        .width(130)
        .endConfig() TextDrawable.IShapeBuilder
        .buildRound(Integer.toString(clusterMarker.getNumAvailablePlaces()), getMarkerColor());
    int width = drawable.getIntrinsicWidth();
    int height = drawable.getIntrinsicHeight();
    Bitmap bitmap = Bitmap.createBitmap(width, height, Bitmap.Config.ARGB_8888);
    Canvas canvas = new Canvas(bitmap);
    drawable.setBounds(0, 0, canvas.getWidth(), canvas.getHeight());
    drawable.draw(canvas);
    return BitmapDescriptorFactory.fromBitmap(bitmap);
}
```

figura 2.4.1

Metoda getMarkerIcon primește un obiect „ClusterMarker” și un cod de culoare a textului: alb în mod normal și roșu pentru textul recent modificat. Metoda utilizează clasa „TextDrawable” pentru a crea o imagine bitmap personalizată. Prin intermediul acestei clase, configurăm stilul textului: culoarea, dimensiunea și fontul.

Forma markerului este dată de metoda „buildRound” ce primește numărul de locuri disponibile și culoarea de fundal în funcție de secția selectată de utilizator.

Următorul pas este de a crea o imagine bitmap pe baza obiectului „TextDrawable”. Se calculează dimensiunile bitmap-ului în funcție de dimensiunile „TextDrawable” și se creează un obiect Bitmap. Apoi, se desenează „TextDrawable” pe canvas-ul bitmap-ului astfel se obține rezultatul final și se returnează.

Această metodă este apoi folosită în funcția „onClusterItemRendered”.

Astfel, fiecare marker va avea o iconiță personalizată în funcție de numărul de locuri disponibile și culoarea secției selectate cu text alb în mod obișnuit și roșu dacă numărul de locuri a fost schimbat.[figura 2.4.2]

Pentru a monitoriza schimbările din Firestore am folosit un element specific acestei baze de date și anume metoda "addSnapshotListener". Atunci când apare o modificare în Firestore, addSnapshotListener declanșează metoda onEvent. Aceasta primește un obiect QuerySnapshot care conține documentele modificate. Parcurgând acest QuerySnapshot avem acces la fiecare document modificat.

Utilizând metoda getId() putem obține ID-ul documentului modificat. Comparăm apoi acest ID cu ID-urile spitalurilor existente în aplicație pentru a determina care spital a fost actualizat. Astfel, atunci când apare o modificare, se vor extrage noile date și actualizăm obiectul "Hospital" corespunzător.

Aplicația este apoi "notificată" prin apelul metodei "onHospitalUpdated".

În activitatea MapsActivity tratăm apelarea acestui callback prin căutarea cluster markerului corespunzător spitalului actualizat și apelarea metodei "updateHospitalChanged" din clasa „HospitalClusterRenderer”. Această metodă actualizează cluster markerul cu noul număr de locuri libere și schimbă culoarea textului timp de câteva secunde pentru a atrage atenția utilizatorului.

```
public void updateHospitalChanged(ClusterMarker clusterMarker) {
    Marker marker = clusterMarker.getMarker();
    if (marker != null) {
        marker.setIcon(getMarkerIcon(clusterMarker, Color.RED));
        new Handler().postDelayed(() -> {
            marker.setIcon(getMarkerIcon(clusterMarker, Color.WHITE));

        }, 3000); // 3 seconds
    } else {
        Log.e("Marker update", "Marker not found for cluster marker");
    }
}
```

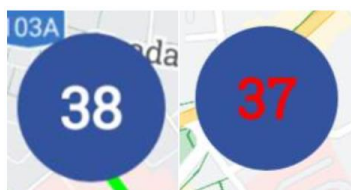


figura 2.4.2

Meniul de alegere al secției dorite este reprezentat în fișierul xml al activității ca un „RadioGroup” aflat în interiorul unui „HorizontalScrollView” ce permite derularea orizontală a conținutului în cazul în care acesta depășește lățimea ecranului.

Pentru fiecare „RadioButton” am setat culoarea specifică secției respective.[figura 2.4.3]

Pentru gestionarea cazului de schimbare a secției alese ne folosim de funcția „onCheckedChanged” ce primește grupul radio și id-ul noii selecții. Această funcție cheamă metoda de actualizare a marcatorilor de pe hartă și redesenarea traseelor în funcție de noua disponibilitate.[figura 2.4.4]

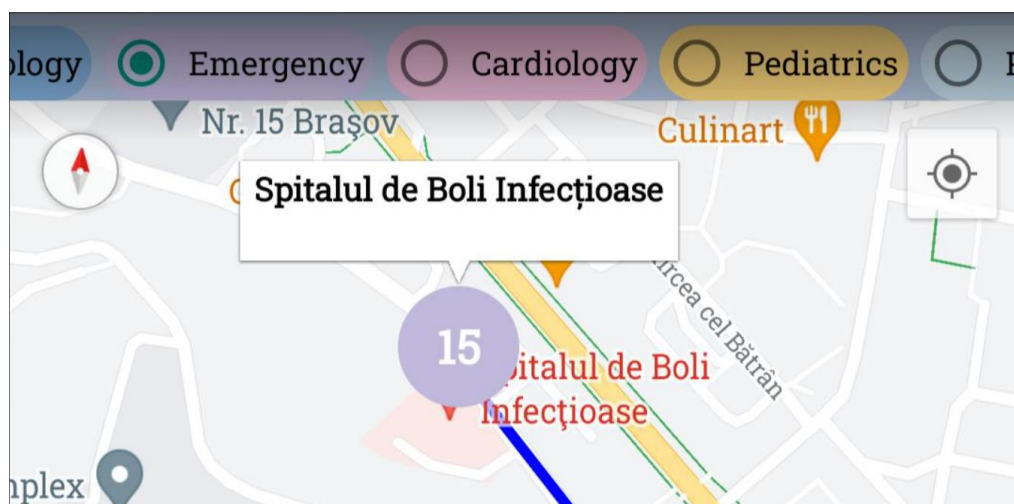


figura 2.4.3 – meniul de alegere a secțiilor

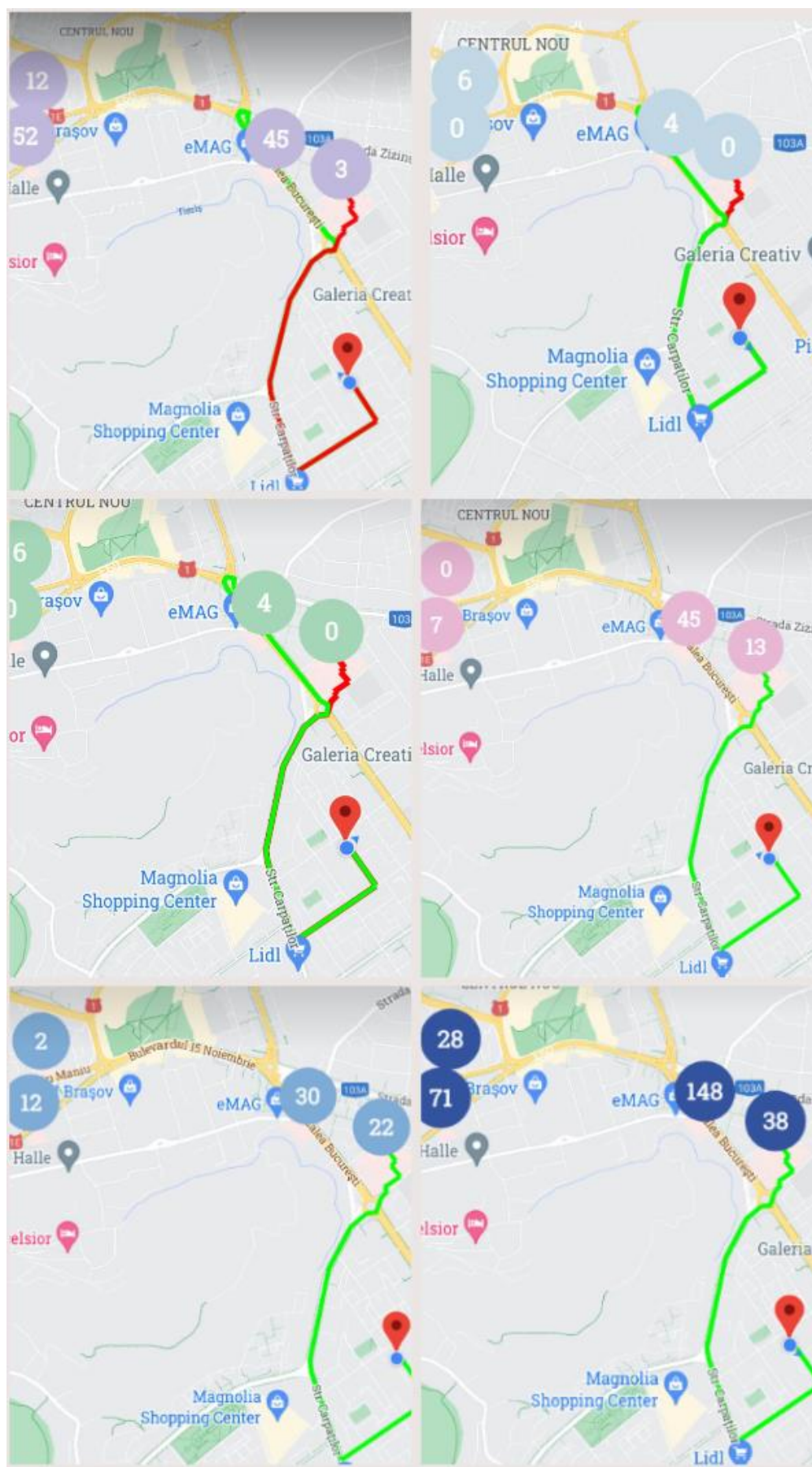


figura 2.4.4-actualizarea în funcție de secția aleasă

### 2.4.3 Desenarea traseelor

Traseele reprezintă o componentă esențială în aplicația noastră. Acestea oferă utilizatorilor direcții precise pentru a ajunge la spitalele dorite. Pentru a realiza această funcționalitate, am folosit două instrumente puternice: biblioteca Retrofit și serviciul Google Maps Directions API.

Pentru a afla de ce clase vom avea nevoie primul pas a fost testarea prin platforma „Postman” ce ne permite să vizualizăm conținutul răspunsurilor API și implementarea claselor necesare. [figura 2.4.5]

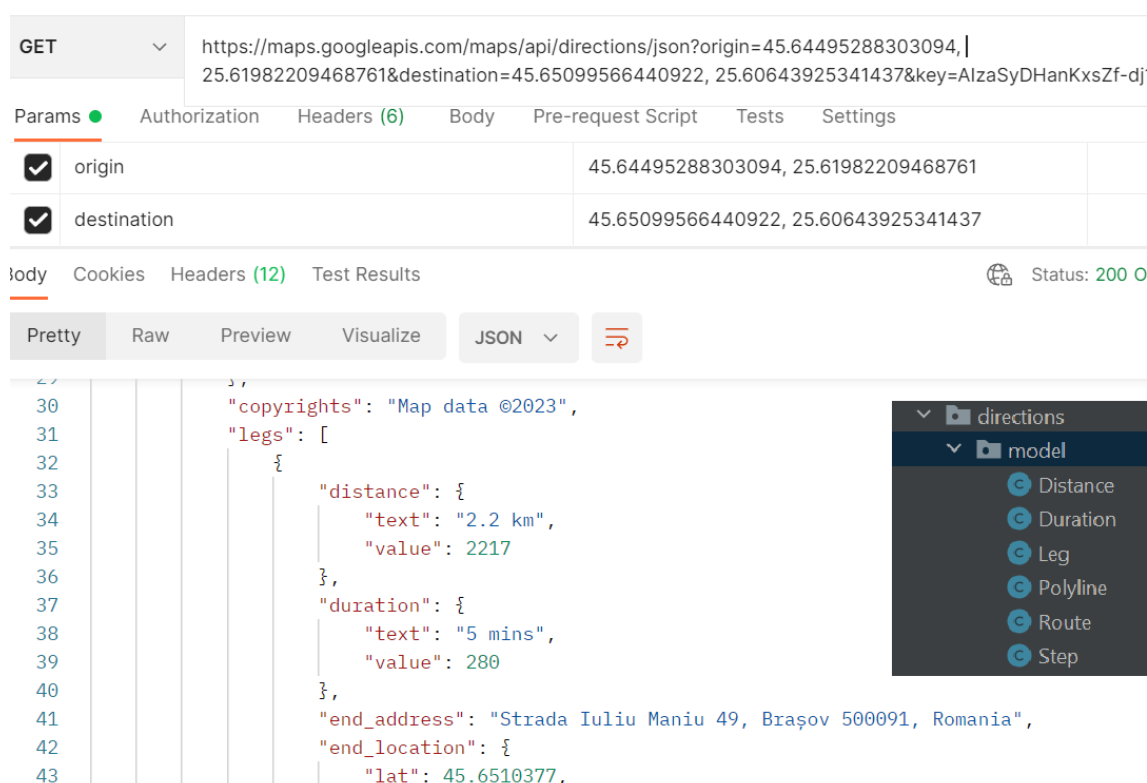


figura 2.4.5

Pentru cererea traseelor am implementat clasa „HospitalRouteGenerator”. Aceasta este responsabilă pentru generarea traseului din locația utilizatorului către un spital dat folosind serviciul „Google Maps Directions API”.

În interiorul acestei clase regăsim funcția „displayDirectionsToHospital”. Funcția primește locația curentă și spitalul către care dorim să aflăm traseul. Cu aceste informații formăm URL<sup>8</sup>-ul cererii către API-ul de direcții asemănător cu cel descris în Postman.

```
String url = "https://maps.googleapis.com/maps/api/directions/json?" +
    "origin=" + latLng.latitude + "," + latLng.longitude +
```

<sup>8</sup> Uniform Resource Locator o adresă web

```
"&destination=" + hlatLng.latitude + "," + hlatLng.longitude +
"&key=" + mContext.getResources().getString(R.string.google_maps_key);
```

Folosim biblioteca „Retrofit” pentru a realiza cererea HTTP<sup>9</sup>. Aceasta ne oferă un mod simplu și eficient de a comunica cu serviciile web și de a gestiona răspunsurile lor.

Am definit o interfață numită "DirectionsApiInterface" în care am denumit metoda "getDirections". Aceasta reprezintă cererea HTTP de tip GET către serviciul Directions API. Adnotarea @GET deasupra metodei specifică faptul că aceasta va fi utilizată pentru a efectua solicitări de tip GET către URL-ul format mai sus.

Dacă răspunsul este unul cu succes putem desena pe harta ruta obținută. Serviciul Directions API returnează un vector de rute posibile, dar în cazul nostru vom alege doar prima rută, deoarece aceasta este, de obicei, cea mai eficientă sau cea mai scurtă.

Pentru a desena ruta pe hartă, vom parcurge etapele rutei. Pentru fiecare etapă, vom obține punctele polyline-ului asociat utilizând metoda "step.getPolyline().getPoints()".

```
public void onSuccess(Route route) { for (Leg leg : route.getLegs()) {
    for (Step step : leg.getSteps()) {
        List<LatLng> points =
PolyUtil.decode(String.valueOf(step.getPolyline().getPoints()));
        PolylineOptions polylineOptions = new PolylineOptions()
            .addAll(points)
            .color(Color.BLUE)
            .width(10);
        Polyline polyline = mGoogleMap.addPolyline(polylineOptions);
        mRoutePolylines.add(polyline); // Add the polyline to the list
    }
}
```

Această metodă este apelată pentru găsirea automată a celui mai apropiat spital sau atunci când utilizatorul apasă pe unul din markerule de pe hartă. În al doilea caz, clasa "HospitalClusterRenderer" apelează automat metoda "onMarkerClick" ce identifică spitalul asociat markerului respectiv iar apoi desenează rute către acesta și afișează panoul de informații suplimentare.

---

<sup>9</sup> Hypertext Transfer Protocol -un protocol de comunicare utilizat pentru transferul de informații pe internet.

#### 2.4.4 Găsirea celui mai apropiat spital

Pentru ordonarea spitalelor în funcție de distanța față de utilizator folosim un nou apel API, de data aceasta către "Distance Matrix Api". Pentru aceasta, folosim coordonatele de localizare ale utilizatorului și lista de spitale disponibile.

În funcție de numărul de spitale din lista noastră, construim un șir de destinații pentru cererea către API. Fiecare destinație este reprezentată de o pereche de coordonate de latitudine și longitudine, separate prin virgulă, iar între ele se folosește caracterul "|" ca delimitator.

Apelul către API ne furnizează un obiect de tip "DistanceMatrixResponse" care conține informații despre distanța și durata călătoriei pentru fiecare combinație de origine și destinație. În cazul nostru, având o singură origine (locația utilizatorului) și mai multe destinații (spitalele), matricea rezultată va avea o singură linie și un număr de coloane corespunzător cu numărul de spitale din baza de date.

	Spitalul Astra	Spitalul de Boli Infecțioase
Locația utilizatorului	distance : 2,293 km duration: 6 min	distance: 3,948 km duration: 9 min

Figura 2.4.6- reprezentare tabelară a răspunsului cererii către Distance Matrix API

Astfel, parcurgem matricea pe linia 0 și atribuim fiecărui spital valori pentru distanță și durata de călătorie de la origine la destinația respectivă.

Acestea vor fi folosite atât pentru ordonarea spitalelor în funcție de distanță cât și pentru fragmentul de informații suplimentare, cu o singură cerere API.

În situația în care cel mai apropiat spital are o capacitate de ocupare care depășește 15% din numărul total de locuri disponibile, vom furniza o rută alternativă către următorul cel mai apropiat spital disponibil. Astfel gestionăm situațiile în care un anumit spital este supraaglomerat și oferim utilizatorului opțiuni alternative pentru a accesa un spital care poate avea o capacitate de ocupare mai mică dar în continuare foarte aproape de locația acestuia.



### 2.4.5 Informații suplimentare despre spitale

Fragmentul de informații suplimentare ne oferă o prezentare detaliată a spitalului selectat. Acesta conține informații precum imaginea spitalului, numele, rating-ul, timpul estimat de călătorie, distanța față de locația utilizatorului, adresa completă și numărul de telefon. De asemenea, utilizatorul poate accesa butonul de direcții pentru a obține indicații de navigare către spital folosind aplicația Google Maps.

Pentru implementarea acestei funcționalități, am utilizat o metoda "onMarkerClick", în cadrul căreia gestionăm evenimentul de apăsare pe un marker de pe hartă.

În cadrul metodei, obținem obiectul ClusterMarker asociat markerului apăsător și extragem spitalul corespunzător. Apoi, se apelează metoda "showDirectionsPanel" și se transmite spitalul ca argument.

Pentru a obține detaliile spitalului, se folosește API-ul Google Places. Fiecare spital din baza de date are un google\_id unic, pe care îl vom folosi pentru a căuta informații suplimentare în API. Pentru a realiza acest lucru, apelăm clasa HospitalDetailsGenerator și apelăm metoda displayHospitalDetails.

Asemănător codului explicat anterior, se construiește URL-ul de apel către API-ul Google, specificând ID-ul spitalului, câmpurile dorite pentru extragerea informațiilor și cheia API :

```
String fields =
"name,formatted_address,formatted_phone_number,website,opening_hours,rating,photo";
String url = "https://maps.googleapis.com/maps/api/place/details/json?" +
"place_id=" + placeld +
"&fields=" + fields +
"&key=" + mContext.getResources().getString(R.string.google_maps_key);
```

După construirea URL-ului de apel către API-ul Google Places, utilizăm obiectul Call pentru a realiza cererea către serverul API. Apelăm metoda enqueue pe obiectul call și pasăm un obiect Callback pentru a trata răspunsul.

În cadrul fragmentului „DirectionsPanelFragment”, utilizăm urmatorul cod pentru a afișa informațiile pe interfața utilizatorului:

```
public void onDetailsReceived(String name, String address, String phone, String website,
double rating, boolean isOpen, String photoUrl) {
```



```

hospitalName.setText(name);
hospitalRating.setRating((float) rating);
hospitalTimeToGetThere.setText(s_hospitalTime);
hospitalDistance.setText(s_hospitalDistance + " km");
hospitalAdress.setText(address);
hospitalNumber.setText(phone);

Glide.with(requireContext())
    .load(photoUrl)
    .placeholder(R.drawable.logo_light)
    .listener(new RequestListener<Drawable>() {
        @Override
        public boolean onLoadFailed(@Nullable GlideException e, Object model,
Target<Drawable> target, boolean isFirstResource) {
            return false;
        }

        @Override
        public boolean onResourceReady(Drawable resource, Object model,
Target<Drawable> target, DataSource dataSource, boolean isFirstResource) {
            hideLoadingScreen();
            showContentViews();
            return false;
        }
    })
    .into(hospitalImage);
}

```

Pentru a afișa imaginea spitalului utilizăm biblioteca Glide. În metoda dată se utilizează metoda "load" a bibliotecii, specificând URL-ul fotografiei spitalului. Prin intermediul metodei "listener" se gestionează evenimentele de încărcare a imaginii. În cazul în care încărcarea a avut succes, se ascunde ecranul de încărcare și se afișează conținutul fragmentului.

În cadrul fragmentului "DirectionsPanelFragment", am implementat și funcționalitatea butonului de direcții. Acest buton oferă utilizatorului posibilitatea de a obține indicații de navigare către spitalul selectat folosind aplicația Google Maps.

În interiorul metodei `onClick`, se extrag coordonatele geografice ale spitalului (latitudine și longitudine) și numele spitalului (label). Aceste informații sunt utilizate pentru a construi un obiect `Uri`<sup>10</sup>, reprezentând adresa de navigare către locația spitalului în aplicația Google Maps. Adresa de navigare este specificată în formatul `"google.navigation:q=latitude,longitude&label=label"`.

În continuare, se verifică dacă dispozitivul utilizatorului are instalată aplicația Google Maps prin apelul metodei `resolveActivity`. Dacă aplicația este instalată, utilizatorul va fi redirecționat către aplicația Google Maps cu indicațiile de navigare către spital. În caz contrar, se afișează un mesaj de eroare prin intermediul unui `Toast`, informând utilizatorul că aplicația Google Maps nu este instalată pe dispozitivul său.



figura 2.4.7

## 2.5 Conectarea la internet – Broadcast receiver

Pentru a asigura funcționalitatea aplicației noastre, este esențial să monitorizăm și să gestionăm conectivitatea la internet a dispozitivului. Pentru acest scop, am implementat clasa „`InternetConnectivityChecker`”, care utilizează un `BroadcastReceiver` pentru a detecta schimbările de conectivitate a rețelei în sistemul de operare Android.

Un `Broadcast Receiver` (Receptor de emisie) este o parte importantă a aplicațiilor Android care permite aplicației să primească și să reacționeze la diverse evenimente care apar în sistemul de operare al dispozitivului.

În cazul nostru, Broadcast Receiver-ul din clasa "InternetConnectivityChecker" este utilizat pentru a monitoriza schimbările de conectivitate a rețelei. Atunci când dispozitivul se conectează sau se deconectează de la internet, sistemul de operare trimite un mesaj de eveniment către Broadcast Receiver. Acesta preia mesajul și poate notifica alte părți ale aplicației despre schimbarea de conectivitate.

Pentru a începe monitorizarea conectivității la internet, se utilizează metoda "start()" a clasei "InternetConnectivityChecker". Aceasta creează un IntentFilter care filtrează evenimentele de conectivitate "CONNECTIVITY\_ACTION" și înregistrează BroadcastReceiver-ul asociat prin intermediul metodei "registerReceiver()" a contextului.

Pentru a opri monitorizarea conectivității, se utilizează metoda "stop()" a clasei. Aceasta dezactivează BroadcastReceiver-ul prin apelul metodei "unregisterReceiver()".

Pentru a informa utilizatorul cu privire la aceste schimbări am implementat interfața "InternetConnectivityListener" ce conține o singură metodă: „onInternetConnectivityChanged (boolean isConnected)”. Această metodă primește un parametru boolean "isConnected" care indică dacă dispozitivul s-a conectat sau deconectat de la internet.

Acum, pentru a comunica utilizatorului datele primite de BroadcastReceiver, putem implementa această interfață și suprascrie metoda "onInternetConnectivityChanged"

@Override

```
public void onInternetConnectivityChanged(boolean isConnected) {  
    if (isConnected) {  
        showOverlay(false);  
        if (needToReload == true) {  
            needToReload = false;  
            reloadMapsActivity();  
        }  
    } else {  
        showOverlay(true);  
        needToReload = true;  
    }  
}
```

Când starea conectivității la internet se schimbă, metoda este apelată automat și primește starea curentă a conectivității prin parametrul "isConnected".

Dacă dispozitivul este conectat la internet (isConnected= true), se șterge suprapunerea de pe hartă (showOverlay(false)) și se verifică dacă este necesară reîncărcarea activității. Dacă trebuie să se reîncarce, se apelează metoda reloadMapsActivity().

Dacă dispozitivul nu este conectat la internet (isConnected==false), se afișează o suprapunere (showOverlay(true)) și se setează variabila needToReload ca true pentru a memora faptul că activitatea hărții trebuie reîncărcată atunci când se restabilește conectivitatea la internet.

Indicația vizuală cu privire la starea de conectare la internet este afișată prin intermediul fragmentului "fragment\_error". [figura 16]

Acesta fragment este inclus în fișierul XML al activității MapsActivity cu ajutorul tag-ului <include> și este inițial setat să fie invizibil. În metoda "showOverlay" setăm vizibilitatea acestui fragment și posibilitatea utilizatorului de a interacționa cu harta :

```
private void showOverlay(boolean show) {  
    if (show) {  
        overlayLayout.setVisibility(View.VISIBLE);  
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_NOT_TOUCHABLE,  
            WindowManager.LayoutParams.FLAG_NOT_TOUCHABLE);  
    } else {  
        overlayLayout.setVisibility(View.GONE);  
  
        getWindow().clearFlags(WindowManager.LayoutParams.FLAG_NOT_TOUCHABLE);  
    }  
}
```

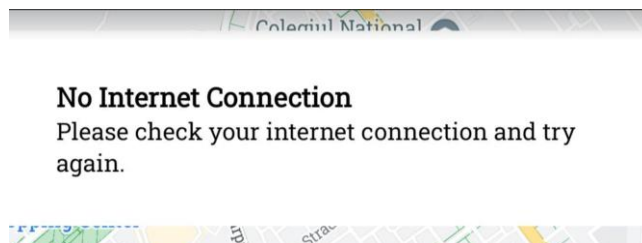


figura 2.5.1

## 2.6 Logarea utilizatorilor

Pentru a permite utilizatorilor să se autentifice în aplicație și să acceseze funcționalitățile disponibile, am implementat un sistem de logare bazat pe ID-ul cardului de sănătate și PIN-ul asociat acestuia, reprezentat de data de naștere a utilizatorului.

La apăsarea butonului de login id-ul este căutat în documentul Firestore „Users” iar dacă este găsit extragem anul nașterii și îl comparăm cu ”parola” introdusă.

Pentru păstrarea utilizatorilor logați am folosit ”SharedPreferences”, o clasă Android ce ne permite să stocăm informațiile de autentificare ale utilizatorului într-un fișier specific, care rămâne disponibil chiar și după închiderea și repornirea aplicației. Astfel, utilizatorii pot rămâne autentificați și pot accesa rapid funcționalitățile aplicației fără a fi nevoie să reintroducă datele de autentificare de fiecare dată.

Folosind clasa ”SharedPreferencesHelper” stocăm id-ul utilizatorului utilizat. În fragmentul de logo verificăm starea acestui id și navigăm corespunzător: către profil sau către pagina de login.

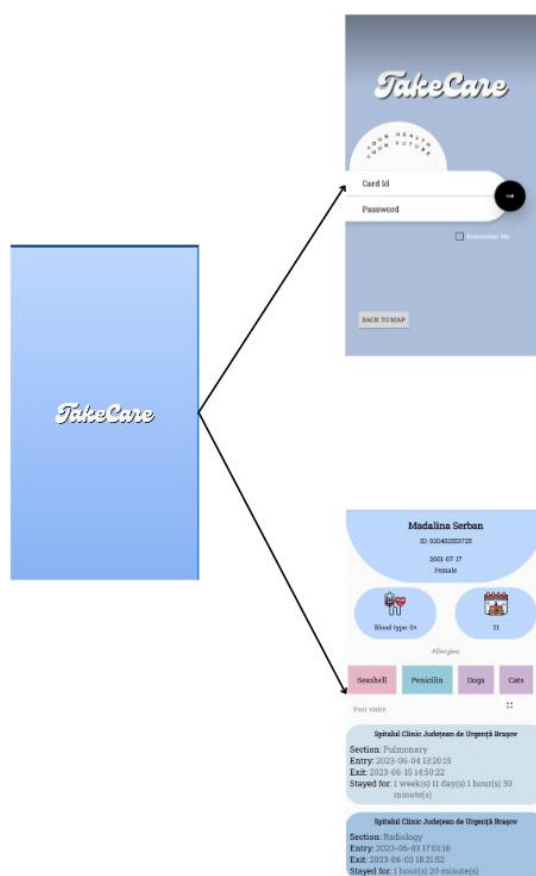


figura 2.6.1

## 2.7 Pagina de profil

Pagina de profil este un element central al aplicației noastre mobile, aici utilizatorul poate vedea conexiunea dintre aplicație și cardul său de sănătate. Această pagină conține informații vitale despre utilizator, precum grupa sanguină, alergii și vârsta. Toate acestea fiind actualizate de personal medical autorizat.

În plus, utilizatorul are acces la istoricul său medical, care conține informații despre secțiile medicale vizitate, datele consultărilor, spitalul și secția la care s-au efectuat, precum și durata petrecută în fiecare dintre aceste locuri.

Prin intermediul paginii de profil, utilizatorul poate avea o imagine de ansamblu asupra stării sale de sănătate și poate accesa rapid și ușor informațiile medicale relevante. Această pagină poate fi deosebit de utilă în situații de urgență, atunci când este necesară furnizarea rapidă a informațiilor medicale.

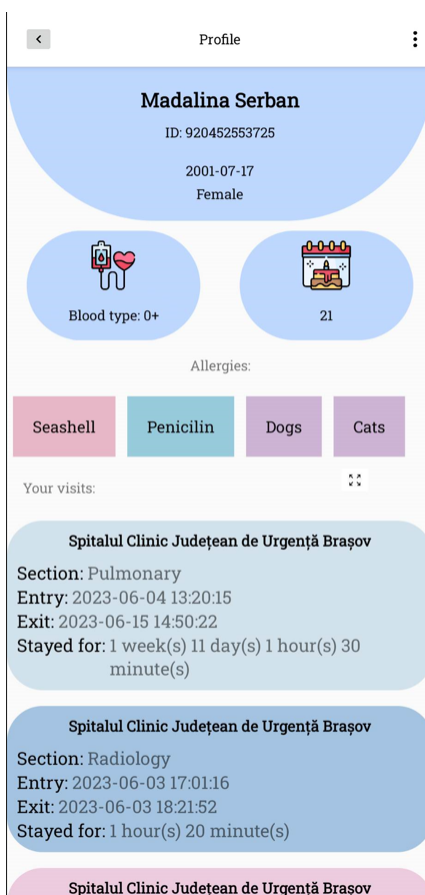


figura 2.7.1

Pentru ambele liste din acest fragment am folosit RecyclerView, o componentă Android care permite afișarea eficientă a unei liste de elemente. Pentru fiecare listă, am creat un fișier XML separat pentru elementul din listă, numit "item\_allergy.xml" și "log\_item\_layout.xml". Acest fișier definește aspectul vizual al unui element din lista respectivă, precum informațiile despre alergen sau detalii despre vizită. De asemenea, am creat și două adaptoare separate, pentru fiecare listă. Aceste adaptoare extind clasa RecyclerView.Adapter și gestionează procesul de afișare și actualizare a datelor în RecyclerView.

Fiecare adaptor conține o clasă internă ViewHolder care reprezintă un element din listă și păstrează referințe către elementele "item" din fișierele XML. Această metodă permite o eficiență sporită, deoarece fișierele XML sunt reutilizate pentru fiecare element din listă, reducând astfel memoria și timpul necesar pentru afișarea datelor. Adaptoarele sunt setate în fragmentul de profil prin intermediul metodei setAdapter() al fiecărui RecyclerView.

Pentru lista de vizite, care poate suferi modificări frecvente, am considerat că este necesar să se asigure actualizarea în timp real a informațiilor. Pentru realizarea acestei funcționalități, am folosit un mecanism de „ascultare” a modificărilor în timp real din baza de date Firebase. Am utilizat un „ChildEventListener”, un element specific pentru Realtime Database, ce ne permite să primim notificări în timp real despre evenimente care apar în baza de date. Astfel, atunci când o nouă înregistrare este adăugată sau când una existentă este modificată, aplicația este notificată instantaneu și lista de vizite este actualizată automat. Există cinci metode definite în cadrul unui „ChildEventListener” dar, în cazul nostru, ne-am concentrat pe două dintre ele: „onChildAdded” și „onChildChanged”.

În ambele cazuri, fie că este vorba despre adăugarea unei noi înregistrări sau modificarea uneia existente, apelăm metoda notifyDataSetChanged(). Această metodă este parte a clasei RecyclerView.Adapter și are rolul de a notifica RecyclerView că datele s-au schimbat și că lista de elemente trebuie reafișată.

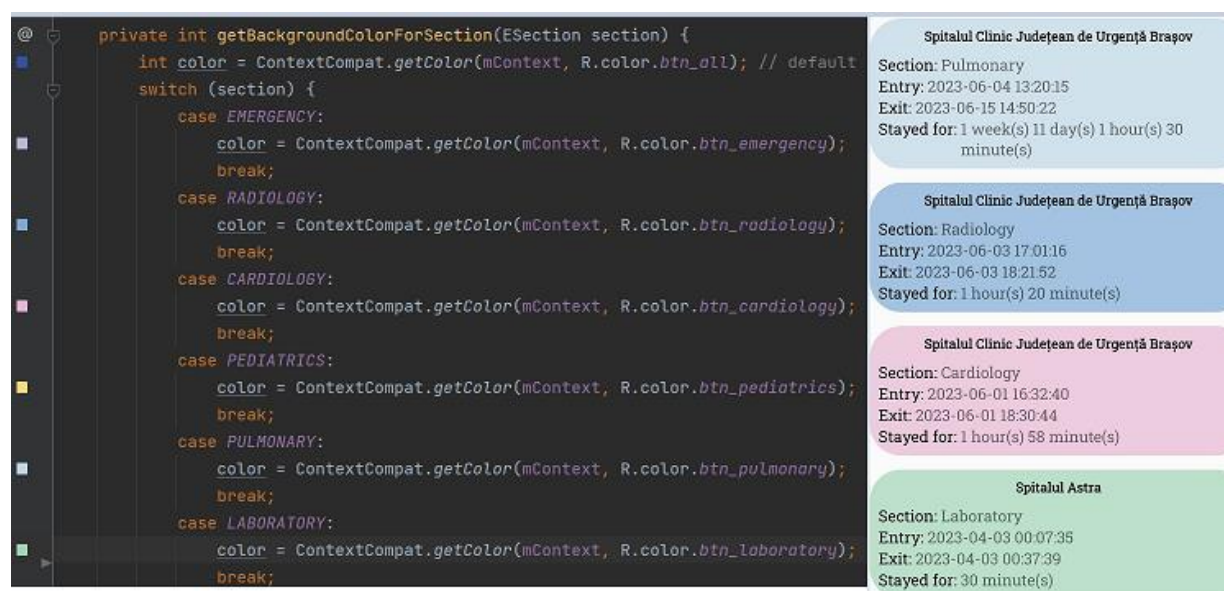
Am ales să folosim culori diferite pentru fundalul elementelor din listă în funcție de secția vizitată sau tipul alergiei. Această funcționalitate este realizată în metoda bind din ViewHolder-ul listei specifice.

```

public void bind(LogEntry logEntry) {
    entryTimestampTextView.setText(logEntry.getEntryTimestamp());
    exitTimestampTextView.setText(logEntry.getExitTimestamp());
    sectionTextView.setText(logEntry.getSection());
    hospitalTextView.setText(logEntry.getHospital());
    timeTextView.setText(logEntry.getTimeDifference());
    ESection section =
ESection.valueOf(logEntry.getSection().toUpperCase(Locale.ROOT));
    int backgroundColor = getBackgroundColorForSection(section);
    GradientDrawable drawable = new GradientDrawable();
    drawable.setShape(GradientDrawable.RECTANGLE);
    drawable.setCornerRadii(new float[] {90, 90, 90, 90, 90, 90, 90, 90});
    drawable.setColor(backgroundColor);
    linear_background.setBackground(drawable);
}

```

Am ales ca fundalul, în cazul listei de vizite, să fie un dreptunghi de tipul „GradientDrawable” căruia i-am rotunjit colțurile iar apoi i-am ales culoarea de fundal specifică. Astfel, clasa ViewHolder ne permite tratarea individuală a fiecărui element din listă.





## Capitolul 3

### APLICAȚIA RASPBERRY PI

#### 3.1 Despre Raspberry Pi

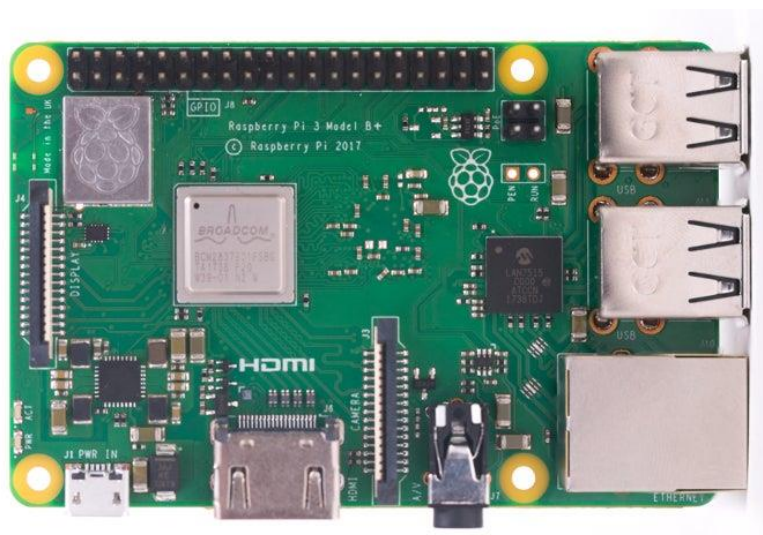


figura 3.1.1 – RaspberryPi 3 Model B+

Raspberry Pi este numele unei serii de calculatoare create de către „Raspberry Pi Foundation”, o organizație caritabilă din Marea Britanie. Scopul acestei fundații este de a promova educația în domeniul calculatoarelor și de a facilita accesul la tehnologie și cunoștințe în acest domeniu.

Lansat în anul 2012, Raspberry Pi a devenit rapid popular în întreaga lume datorită prețului accesibil și a capacităților sale versatile. Placa Raspberry Pi conține toate componentele unui calculator tradițional, cum ar fi procesor, memorie RAM, porturi de conectivitate și chiar sloturi pentru carduri de memorie. Cu toate acestea, dimensiunile reduse și costul scăzut fac ca Raspberry Pi să fie o opțiune ideală pentru proiecte de hobby, învățare a programării și dezvoltare de soluții IoT<sup>11</sup>.

Raspberry Pi rulează pe sistemul de operare Linux, dar suportă și alte sisteme de operare precum Raspbian, o distribuție specială de Linux optimizată pentru această platformă. Placa Raspberry Pi oferă și un set de pini GPIO (general purpose input/output), permițând conectarea și controlul diferitelor componente electronice, cum ar fi senzori.

Pentru a începe să ne scriem codul Python pe un Raspberry Pi vom alimenta placa folosind portul micro-USB și ne vom conecta la un monitor folosind un cablu HDMI. Putem conecta elemente periferice ca mouse și tastatură folosim porturile USB iar conexiunea la internet se poate face atât prin cablu Ethernet cât și prin WiFi. Întregul sistem de operare și datele sunt salvate pe un card microSD, în cazul nostru, de 16GB.

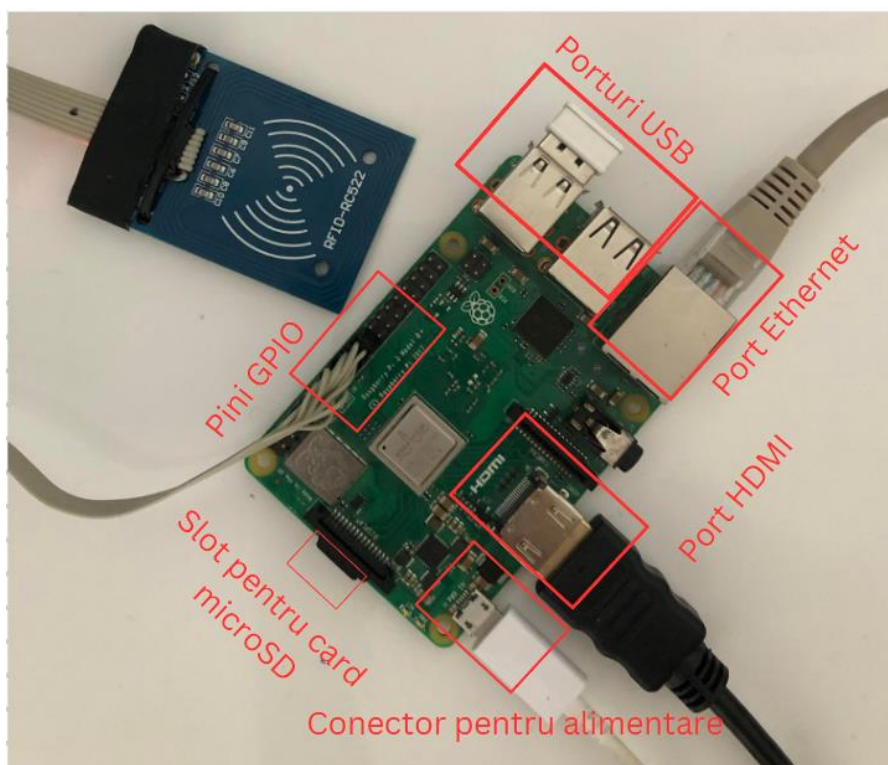


figura 3.1.2

---

<sup>11</sup> Internet of Things

### 3.1.1 Conectarea ssh

Odata ce suntem conectați la internet, pentru a scuti din timpul de pregătire, ne putem conecta la laptopul personal pentru a scrie cod oriunde, fără utilizarea unui monitor și a perifericelor.

SSH (Secure Shell) este un protocol de rețea care permite conectarea și administrarea unui dispozitiv de la distanță într-un mod sigur. În cazul Raspberry Pi, utilizarea SSH ne permite să ne conectăm la placa Raspberry Pi și să o gestionăm prin linia de comandă sau prin intermediul unui editor de text.

Prin intermediul IDE-ului Visual Studio Code procesul este unul simplu. Tot ce trebuie să facem este să instalăm extensia "Remote - SSH" din Visual Studio Code și să introducem corect informațiile de conectare. Pentru a afla adresa IP corespunzătoare plăcuței putem folosi comanda "hostname -I" iar parola default este chiar "raspberrypi". Dacă laptopul și plăcuța sunt conectate la aceeași rețea de internet conectarea ar trebui să decurgă fără probleme. Cu toate aceste configurări putem acum să ne deconectăm plăcuța de la monitor și periferice și să scriem cod de pe computerul personal.

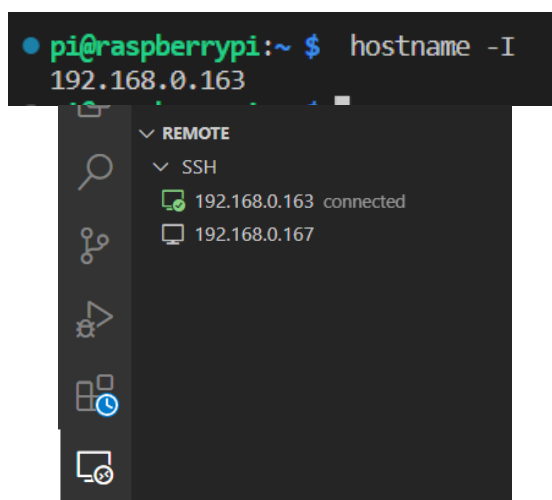


figura 3.1.3

## 3.2 Despre RFID

Modulul RFID RC522 este un dispozitiv care poate citi și scrie informații pe cartele RFID. Acesta funcționează la o frecvență specifică de 13.56 MHz, prin crearea unui câmp electromagnetic în jurul său prin care comunică cu etichetele RFID care sunt plasate în apropiere. Acesta poate detecta și citi informațiile stocate precum ID-ul unic desemnat fiecărui tag.

### 3.2.1 Conectarea hardware

Am conectat acest dispozitiv la plăcuța Raspberry folosind pinii GPIO ai plăcuței, pinii modului RFID și opt cabluri de tip conector (mamă-mamă) în felul următor:

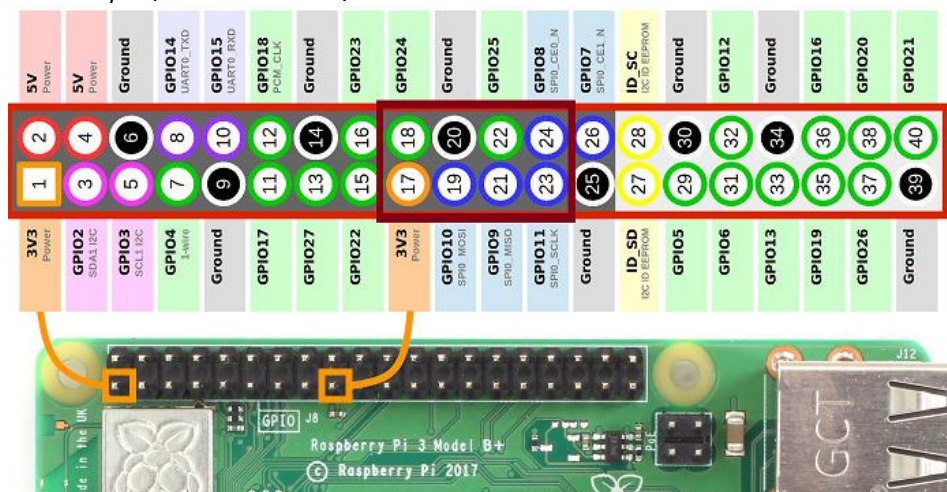


figura 3.2.1

Conform figurii de mai sus, pinii la care ne-am conectat sunt 17,18,19,20,21,22,23 și 24.

Pinii de alimentare:

Pinul de alimentare (3.3V) - Acesta trebuie conectat la pinul de alimentare de 3.3V (pinul 17) al plăcuței Raspberry Pi.

Pinul GND (Ground) - Acesta este pinul de conexiune la masă (0V) și trebuie conectat la un pin GND corespunzător de pe plăcuța Raspberry Pi.(pinul 20)

Pinii de comunicație de date:

Pinul SDA (Serial Data Line) - Acesta este pinul pentru transferul datelor între modulul RFID și Raspberry Pi și trebuie conectat la un pin GPIO disponibil pe plăcuță, cum ar fi GPIO24 (pinul 18).

Pinul MOSI (Master Out Slave In) - Acesta este pinul prin care Raspberry Pi trimite date către modulul RFID și trebuie conectat la un alt pin GPIO disponibil, cum ar fi GPIO10 (pinul 19).

Pinul MISO (Master In Slave Out) - Acesta este pinul prin care modulul RFID trimite date către Raspberry Pi și trebuie conectat la un alt pin GPIO disponibil, cum ar fi GPIO9 (pinul 21).

Pinul SCK (Serial Clock) - Acesta este pinul pentru sincronizarea comunicației între modulul RFID și Raspberry Pi și trebuie conectat la un alt pin GPIO disponibil, cum ar fi GPIO11 (pinul 23).

Pinii de control:

Pinul RST (Reset) - Acesta este pinul utilizat pentru a reseta modulul RFID și poate fi conectat la un pin GPIO disponibil, cum ar fi GPIO25 (pinul 22).

Pinul IRQ (Interrupt Request) - Acesta este pinul utilizat pentru a genera o cerere de întrerupere către Raspberry Pi atunci când apar evenimente specifice în modulul RFID. Conexiunea acestui pin este opțională și poate fi conectat la un pin GPIO disponibil pe plăcuță, dacă este necesară funcționalitatea de întrerupere.

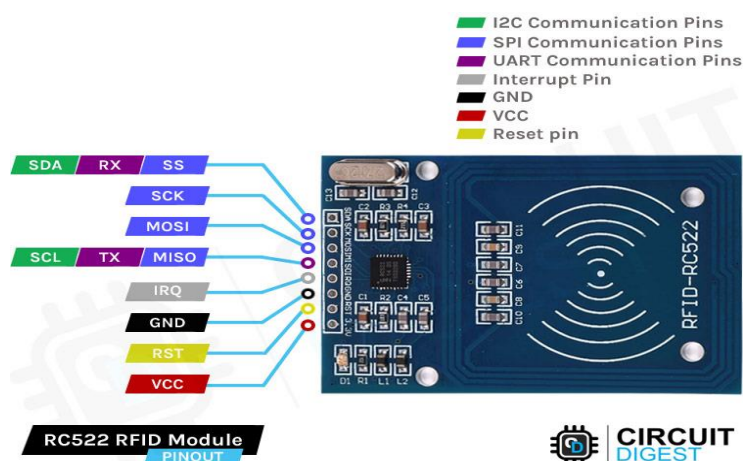


figura 3.2.1

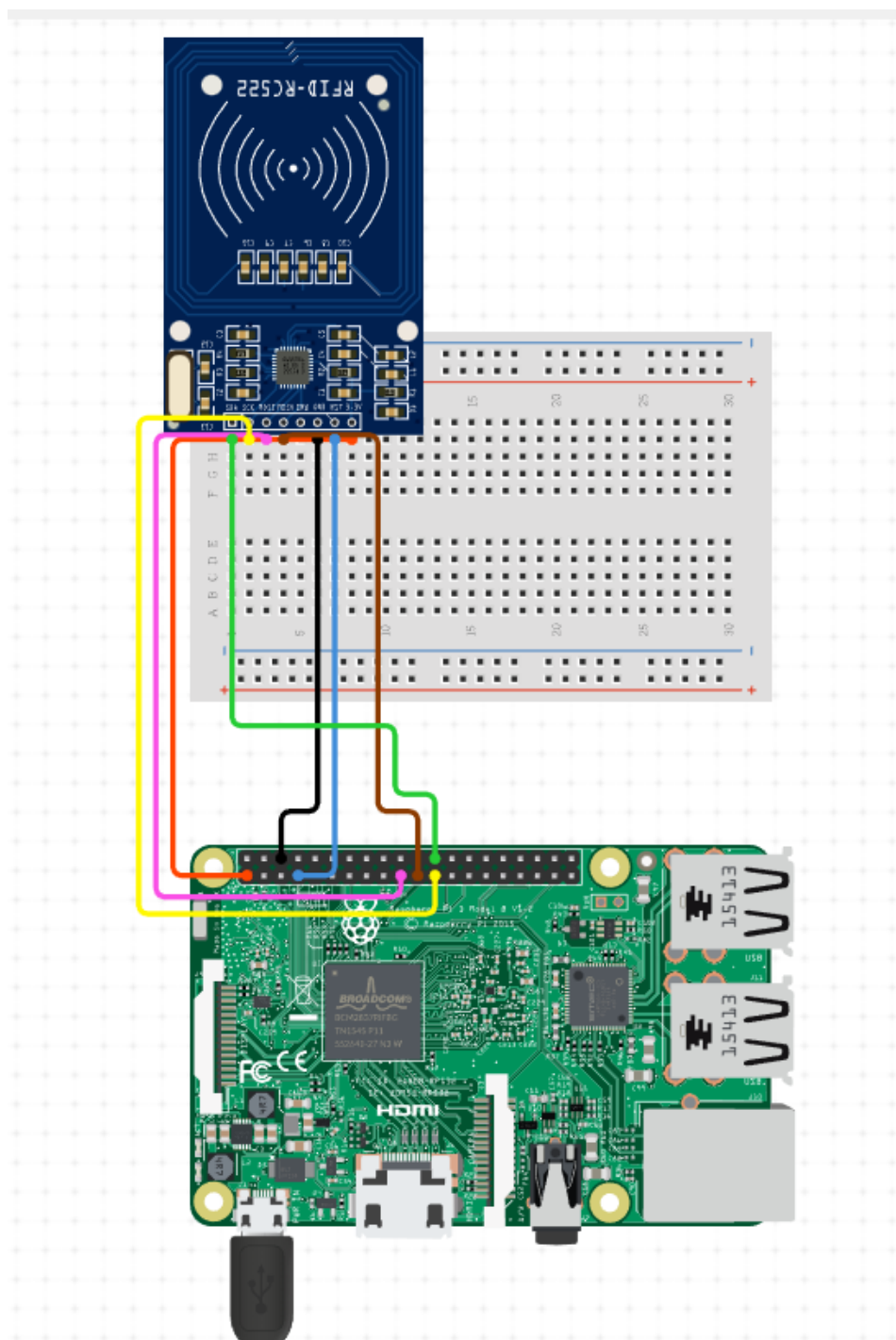


figura 3.2.3- Schema în circuit.io a conectării fizice a elementelor

### 3.2.2 Cardurile RFID

Cardurile RFID reprezintă o componentă esențială a tehnologiei de identificare cu radiofrecvență (RFID). Acestea sunt dispozitive electronice ce utilizează o combinație de cip RFID și o antenă încorporată pentru a permite transferul de date între card și cititor.



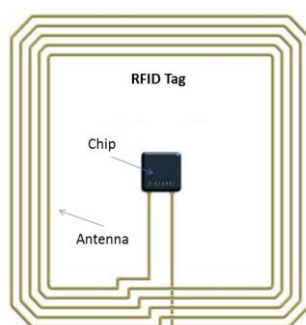


figura 3.2.4

De-a lungul anilor, cardurile RFID au devenit din ce în ce mai populare și sunt utilizate într-o gamă largă de aplicații, de la accesul în clădiri și controlul accesului în transportul public, până la urmărirea activelor în industrie.

Pentru proiectul nostru cardurile RFID sunt esențiale. Acestea ne permit să identificăm și să autentificăm utilizatorii, să gestionăm accesul în clădiri și să monitorizăm activitatea pacienților într-un mod eficient și securizat.

Prin atribuirea unui card RFID pentru fiecărui pacient, putem reține traseul internărilor sau vizitelor medicale ale acestuia prin simpla apropiere a cardului de cititor.

Deși cardurile RFID pot memora mai multe date, în cazul nostru, am ales să ne concentrăm doar pe utilizarea ID-ului unic înregistrat pe card. Aceasta ne permite să identificăm pacientul în baza de date și să obținem rapid și eficient informațiile relevante despre el, cum ar fi numele, data de naștere, alergiile, istoricul vizitelor etc dar și să trimitem date cum ar fi: ora și data de intrare sau ieșire.

### 3.3 Conectarea la Firebase și biblioteci folosite

Pentru a ne conecta la platforma Firebase de pe dispozitivul Raspberry Pi am a trebuie să începem cu câteva comenzi în terminal:

Primul pas, fie că avem nevoie de Firebase sau nu este să ne asigurăm că avem Python instalat pe plăcuța noastră. Folosim comanda : `"python3 --version"` putea a vedea versiunea currentă de pe dispozitivul nostru. Dacă nu este instalat folosim comanda `"sudo apt-get install python3"` pentru a instala Python 3. Pentru proiectul nostru folosim Python 3.7.3.

Odată ce Python este instalat putem începe integrarea cu baza de date. Pentru a interacționa cu serviciul Firebase este necesară biblioteca Pyrebase. Pentru instalarea acesteia rulăm comanda: `"pip install pyrebase"`

Biblioteca Pyrebase este o bibliotecă Python ce ne permite comunicarea cu toate serviciile Firebase. Astfel interacționăm atât cu Firestore cât și cu Realtime Database, serviciile folosite în proiectul nostru. Versiunea folosită pentru proiectul nostru este 3.0.27. Este important ca toate bibliotecile folosite să fie compatibile între ele.

```
pi@raspberrypi:~ $ pip show pyrebase
Name: Pyrebase
Version: 3.0.27
Summary: A simple python wrapper for the Firebase API
Home-page: https://github.com/thisbejim/Pyrebase
```

figura 3.3.1

După instalarea Pyrebase, putem importa biblioteca în codul nostru Python utilizând instrucțiunea "import pyrebase".

O altă bibliotecă esențială este „firebase\_admin”. Aceasta este o bibliotecă oficială a Firebase dezvoltată de Google. Biblioteca oferă o interfață mai avansată și puternică pentru a interacționa cu serviciile Firebase. Firebase Admin SDK este destinat utilizării în mediul "server-side", oferind capacități mai complexe de administrare a aplicației și securitate. Pentru această bibliotecă am folosit versiunea 5.0.0.

```
pi@raspberrypi:~ $ pip show firebase-admin
Name: firebase-admin
Version: 5.0.0
Summary: Firebase Admin Python SDK
Home-page: https://firebase.google.com/docs/admin/setup/
```

figura 3.3.2

Pentru a avea acces la bazele de date create avem nevoie de introducerea unor credențiale. Acestea sunt disponibile în Firebase în meniul "Settings". Adăugăm informațiile din credențiale într-un fișier JSON (în cazul nostru, "service\_account.json") și să specificăm calea către acest fișier în codul nostru.

### 3.4 Codul aplicației

Un pas crucial pentru realizarea aplicației noastre a fost conectarea pacuței la modulul RFID-RC522. Se importă bibliotecile necesare pentru controlul pinilor GPIO ce ajută la conectarea cu modulul și gestionarea acestuia. De asemenea, importăm și biblioteca mfrc522 care ne permite să interacționăm cu modulul RFID-RC522.

```
import RPi.GPIO as GPIO
from mfrc522 import SimpleMFRC522
```



```
# Configurarea pinilor GPIO
GPIO.setmode(GPIO.BOARD)
```

```
# Inițializarea modului RFID
reader = SimpleMFRC522()
```

RPi.GPIO este biblioteca utilizată pentru controlul pinilor GPIO ai plăcuței Raspberry Pi. Prin importul acestei biblioteci și inițializarea modului GPIO în modul GPIO.BOARD, putem configura și controla pinii fizici ai Raspberry Pi.

Biblioteca mfrc522 este utilizată pentru a interacționa cu modulul RFID-RC522. Importând clasa SimpleMFRC522 din această bibliotecă, putem crea un obiect reader care ne permite să citim și să scriem date către și de la cardurile RFID conectate.

Odata ce variabila reader este inițializată cu "SimpleMFRC522()" tot ce trebuie să facem pentru a citi id-ul de pe cartelă este un simplu reader.read() după care apelăm funcția check\_card\_id(id). Aici verificăm, pentru început, dacă id-ul citit există în baza de date Firestore și dacă cardul asociat este activat.

În cazul în care ambele cerințe sunt îndeplinite verificăm dacă există o dată de intrare fără o dată de ieșire corespunzătoare pentru acest card. În acest caz, vom considera noua înregistrare a cardului ieșirea din secție și vom crește numărul de locuri disponibile, altfel această înregistrare este intrarea în secție iar numărul de locuri valabile scade. În figura 3.4.1 observăm modul în care gestionăm aceste evenimente. Atunci când perechea "entry-timestamp", "exit-timestamp" este completă aceste două valori devin din nou "null" iar datele sunt trimise în subnodul log pentru a fi gestionate de aplicația Android.

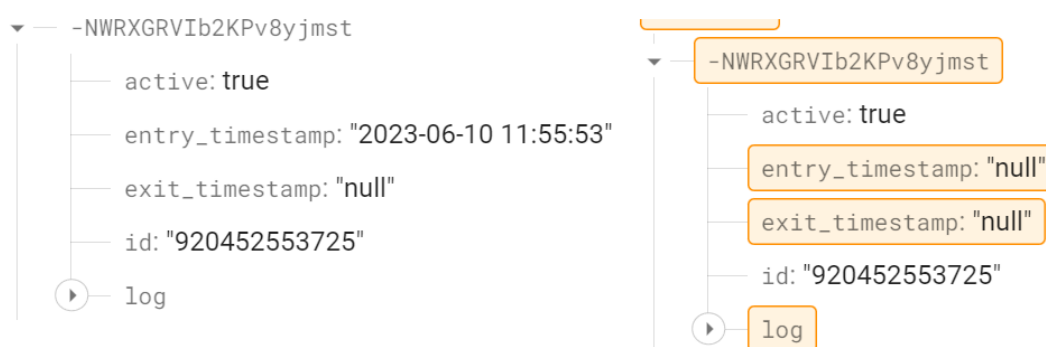


Figura 3.4.1-Gestionarea intrărilor și ieșirilor în baza de date

Pentru a realiza acest eveniment funcția "check\_card\_id(card\_id)" se folosește atât de lista de „UserCard” din Realtime Database cât și de cea de spitale din Firestore.

```
empty_spots = doc.to_dict().get('empty_spots')
found = False
for user in users.each():
    if user.val().get('id') == card_id and user.val().get('active') == True:
        now = datetime.datetime.now(timezone)
        date_time = now.strftime("%Y-%m-%d %H:%M:%S")
        entry_timestamp=user.val().get('entry_timestamp')
        exit_timestamp=user.val().get('exit_timestamp')
        if entry_timestamp == "null":
            db.child("UserCard").child(user.key()).update({"entry_timestamp": date_time})
            print("Added -1 to the available spots")
            doc_ref.update({"availability": firestore.Increment(-1)})
            print("Entry sent to the database")
            print("Wait 3 seconds")
            time.sleep(3)
        elif exit_timestamp == "null":
            db.child("UserCard").child(user.key()).update({"exit_timestamp": date_time})
            exit_timestamp=user.val().get('exit_timestamp')
            db.child("UserCard").child(user.key()).child("log").push({"entry_timestamp": entry_timestamp, "exit_timestamp": date_time})
            db.child("UserCard").child(user.key()).update({"entry_timestamp": "null"})
            db.child("UserCard").child(user.key()).update({"exit_timestamp": "null"})
            print("Added 1 to the available spots")
            doc_ref.update({"availability": firestore.Increment(1)})
            print("Entry and exit sent to the database")
            print("Wait 3 seconds")
            time.sleep(3)
        found=True
```

figura 3.4.2 - Actualizarea datelor la scanarea cardului de sănătate activat

Firecare plăcuță are o secție specifică pentru care modifică aceste date. În exemplul dat am setat modificare locurilor libere pentru spitalul "Spitalul Clinic Județean de Urgență Brașov", secția cardiologie. Verificăm dacă este cazul unei intrări sau a unei ieșiri din secție, conform datelor din nodul asociat cardului pacientului, și actualizăm datele după caz. Adăugăm o pauză de 3 secunde după scanarea fiecărui card pentru a evita scanarea accidentală.

Dacă cardul nu este găsit sau este găsit dar nu este activat vom afișa un mesaj de aneționare si nicio procesare a datelor nu va avea loc.

```
if user.val().get('active') == False:
    print("Card is not active")
    print("Wait 3 seconds")
    time.sleep(3)
```

figura 3.4.3

## Capitolul 4

### APLICAȚIA WEB

Așa cum a fost descris și în primul capitol, aplicația web are un rol administrativ, vital pentru proiectul „Take Care”. Prin integrarea unei aplicații web am propus un mod intuitiv și ușor de utilizat pentru personalul medical de a adăuga, modifica și activa sau dezactiva cardurile de sănătate.

De exemplu, atunci când un pacient nu mai are asigurare medicală sau își declară cardul de sănătate pierdut personalul autorizat are posibilitatea să modifice aceste date. Dacă pacientul își schimbă numele sau descoperă o nouă alergie aceste date vor fi, de asemenea, modificate pentru a reflecta starea curentă a utilizatorului.

Astfel, utilizatorii vor fi mereu informați cu date corecte, actuale și primite de la personal autorizat.

#### 4.1 Despre Html, Css și Javascript

HTML (HyperText Markup Language) este limbajul de marcare utilizat pentru a structura și afișa conținutul unei pagini web. Folosind elemente și etichete specifice, putem crea structura unei pagini web și putem adăuga diverse elemente, cum ar fi texte, imagini, tabele etc.

CSS (Cascading Style Sheets) este un limbaj de stilizare care permite să definim aspectul și prezentarea elementelor HTML. Cu ajutorul CSS, putem controla culorile, fonturile, dimensiunile și alte proprietăți vizuale ale elementelor HTML.

JavaScript este un limbaj de programare utilizat pentru a face paginile web interactive. Cu JavaScript, putem crea și modifica dinamic conținutul unei pagini web, putem răspunde la evenimente precum click-uri sau apăsări de taste și putem comunica cu serverul pentru a obține sau a trimite date.

Combinând aceste 3 limbaje am creat o pagină web interactivă pentru gestionarea cardurilor de sănătate.

## 4.2 Pagina principală

Pagina principală din aplicația noastră web ne permite vizualizarea cardurilor de sănătate înregistrate împreună cu posibilitatea de activare/dezactivare a acestora sau chiar ștergerea lor.

Primul pas în realizarea acestei pagini a fost declararea unui fișier HTML ("index.html") ce se ocupă de definirea structurii și de conținutul paginii. Acesta utilizează diferite elemente și etichete pentru a crea o interfață vizuală interactivă și plăcută pentru utilizatori.

Am declarat titlul paginii utilizând eticheta "<h1>" în interiorul secțiunii "<body>".

Culoarea și stilul textului sunt date apoi de fișierul „styles.css” unde îi oferim etichetei h1 o margine și o culoare adecvată.

În același mod am stilizat toate elementele din pagina noastră, butoanele au o culoare de fundal verde (#4CAF50), text alb și o margine rotunjită:

```
button {  
  padding: 10px 20px;  
  background-color: #4CAF50;  
  color: white;  
  border: none;  
  border-radius: 4px;  
  cursor: pointer;  
}
```

Tabelul, folosit pentru afișarea cardurilor de sănătate înregistrate, a fost stilizat cu lățimea de 100%, pentru a se adapta la dimensiunea ecranului. Celulele antetului (th) au un fundal de culoare gri deschisă, în timp ce celulele de conținut (td) au aceeași culoare de fundal, aplicată la fiecare al doilea rând. [figura 4.2.1]

### Medical Cards

ID	Active	Action
920452553728	false	<div>Activate</div> <div>Delete</div>
920452553725	true	<div>Deactivate</div> <div>Delete</div>

figura 4.2.1

Asemănător proiectelor discutate anterior datele folosite în aplicație sunt preluate din Firebase. Configurările acestei baze de date sunt specificate în fișierul "firebase\_config.js".

Pentru a utiliza Firebase în aplicația web, am inclus fișierul "firebase\_config.js" în fișierul "index.html" prin utilizarea unei etichete "<script>". Astfel, putem accesa și utiliza funcționalitățile oferite de Firebase pentru a interacționa cu baza de date și a prelua informațiile necesare pentru afișarea cardurilor de sănătate înregistrate.

De popularea tabelului este responsabil fișierul "app.js".

În interiorul acestuia declarăm referință către nodul "UserCard" din baza de date și parcurgem toate subnodurile. Pentru fiecare subnod, se obțin datele specifice și se creează o nouă linie (<tr>) în tabel.

Pentru fiecare card de sănătate, se adaugă celule (<td>) în rândul tabelului. O celulă conține ID-ul cardului, alta conține informații despre starea de activitate a cardului și celulele finale conțin butoane pentru schimbarea stării de activitate și pentru ștergerea cardului.

```
snapshot.forEach(function(childSnapshot) {
  var data = childSnapshot.val();
  var databaseId = childSnapshot.key;
  var active = data.active;
  var id = data.id;
  var row = $('<tr>');
  var idCell = $('<td>').text(id);
  row.append(idCell);
  var activeCell = $('<td>').text(active);
  row.append(activeCell);
  var button = $('<button>').text(active ? 'Deactivate' : 'Activate');
```

```

button.click(function() {
    var confirmed = confirm("Are you sure you want to change the status of this
card?");
    if (confirmed) {
        var newActiveValue = !active;
        firebase.database().ref('UserCard/' + databaseId +
'/active').set(newActiveValue);
    }
});
var actionCell = $('<td>').append(button);
row.append(actionCell);

var deleteButton = $('<button>').text('Delete');
deleteButton.addClass('delete-button');
deleteButton.click(function() {
    var confirmed = confirm("Are you sure you want to delete this card?\n\nThis
action will only delete the card information.");
    if (confirmed) {
        firebase.database().ref('UserCard/' + databaseId).remove();
    }
});

var deleteCell = $('<td>').append(deleteButton);
row.append(deleteCell);

```

Butonul de schimbare a stării de activitate are un text care variază în funcție de starea curentă a cardului (activat sau dezactivat). La apăsarea acestui buton, utilizatorul primește o cerere de confirmare, iar dacă o aprobă, starea cardului este inversată și actualizată în baza de date.[figura 4.2.2]

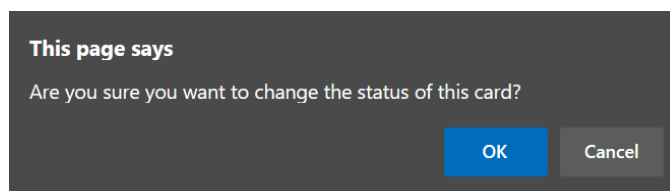


figura 4.2.2

Similar, pentru butonul de ștergere, adăugăm un eveniment de click care solicită confirmarea utilizatorului prin intermediul unei ferestre de dialog. Dacă utilizatorul confirmă, ștergem cardul corespunzător din baza de date prin expresia "firebase.database().ref('UserCard/' + databaseId).remove();" [figura 4.2.3]

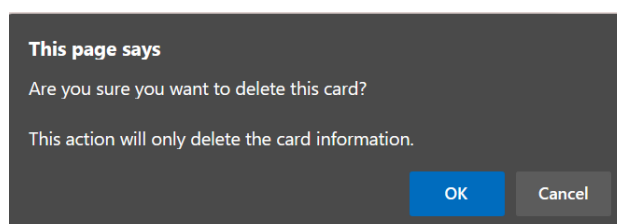


figura 4.2.3

De asemenea, am adăugat și un eveniment click pe fiecare linie din tabel, utilizând "row.click(function() {...})". Acest eveniment permite personalului să navigheze către pagina de detalii a unui card atunci când se face click pe o linie din tabel.

În același mod, am declarat și un buton pentru adăugarea unui nou utilizator. Atunci când acest buton este apăsat, vom fi redirecționați către aceeași pagină HTML care este utilizată pentru actualizarea datelor unui utilizator existent, însă vom folosi un fișier

JavaScript diferit ce permite personalului să completeze informațiile pentru un nou utilizator, în mod similar cu modul de actualizare a datelor unui utilizator deja existent.

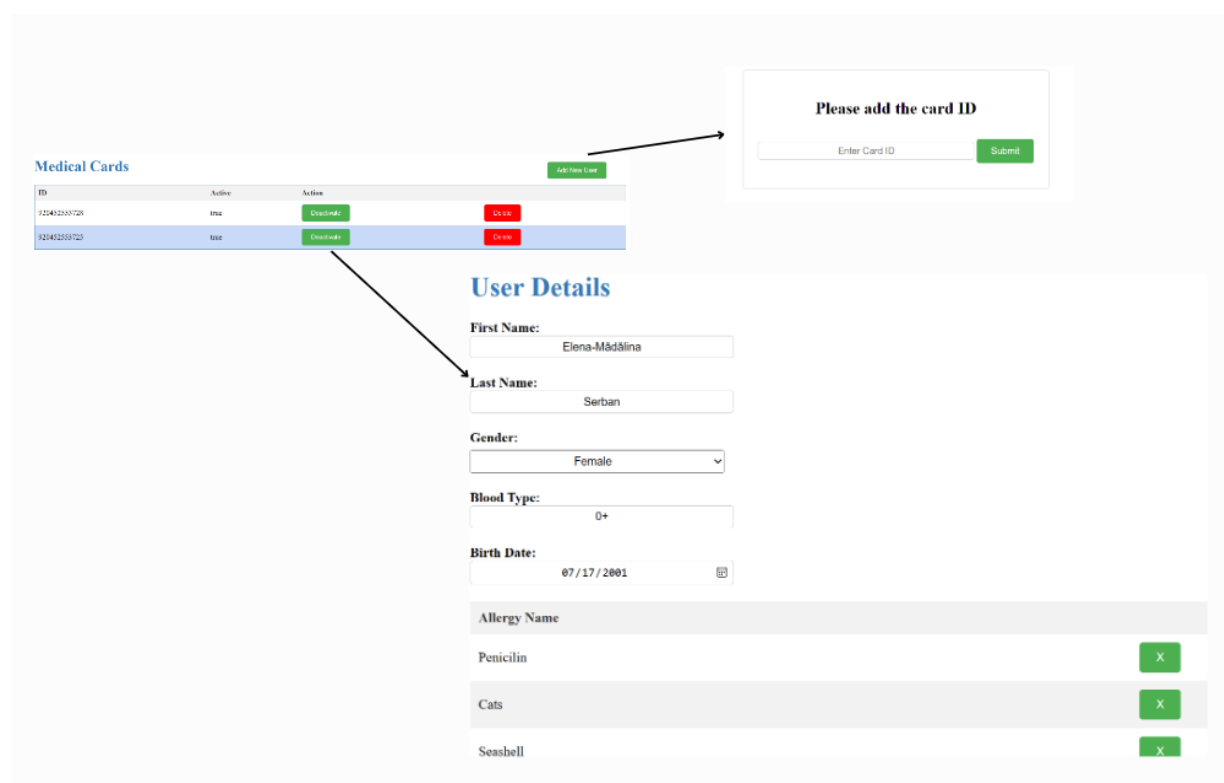


figura 4.2.4 – adăugarea/modificarea unui utilizator

## 4.3 Modificarea și adăugarea detelor

### 4.3.1 Modificarea datelor

Pentru a permite modificarea datelor utilizatorului selectat, navigăm către o nouă pagină "userdetails.html?id=" + id. Această pagină primește ca parametru un "id" care reprezintă identificatorul utilizatorului selectat.

Pentru a popula formularul cu datele existente din baza de date, extragem id-ul utilizatorului din parametrul URL al paginii și apoi căutăm documentul corespunzător în baza de date. Astfel, obținem informațiile despre utilizatorul selectat, cum ar fi numele, prenumele, sexul, grupa de sânge, data de naștere și alergiile. Aceste date sunt apoi utilizate pentru a completa câmpurile corespunzătoare ale formularului din pagina "userdetails.html", astfel încât administratorul să poată vedea și modifica valorile actuale ale acestor câmpuri.

Astfel, completăm automat câmpurile formularului HTML.[figura 4.3.1]

```
var id = urlParams.get('id');
var userDocRef = firebase.firestore().collection('Users').doc(id);
userDocRef.get().then(function (doc) {
  if (doc.exists) {
    userData = doc.data();
    document.getElementById('firstName').value = userData.first_name;
    document.getElementById('lastName').value = userData.last_name;
    document.getElementById('gender').value = userData.gender;
    document.getElementById('bloodType').value = userData.blood_type;
    document.getElementById('birthDate').value =
      formatDate(userData.birth_date);
  }
});
```

## User Details

**First Name:**

**Last Name:**

**Gender:**

**Blood Type:**

**Birth Date:**

Allergy Name	
Penicilin	X
Cats	X
Seashell	X

**New Allergy**

**Add**

**Update**

figura 4.3.1



Pentru a popula tabelul de alergii din pagina utilizatorului, se utilizează vectorul de referințe către colecția "Allergies" din baza de date. Acest vector conține identificatorii (ID-urile) alergiilor asociate utilizatorului.[figura 4.3.2]

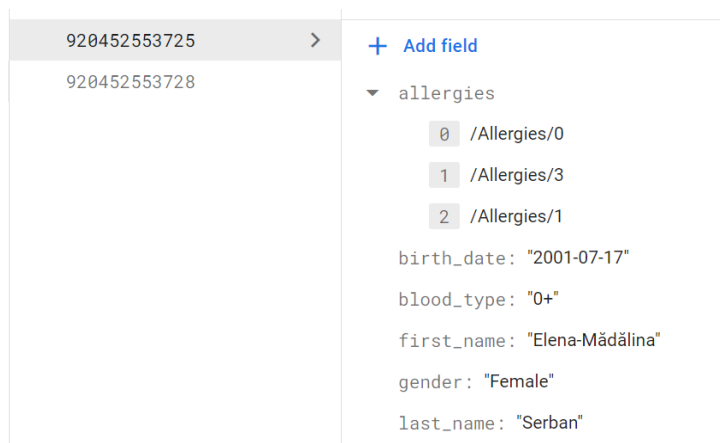


figura 4.3.2

În continuare, se utilizează funcția "fetchAllergyNames(allergies)" pentru a obține numele alergiilor asociate utilizatorului. Această funcție primește vectorul de referințe către alergii și returnează o listă cu numele alergiilor.

Pentru fiecare alergie, se adaugă o nouă linie în tabel, care conține numele alergiei și un buton de ștergere asociat. Butonul de ștergere are un eveniment de click care apelează funcția „deleteAllergy(index)” pentru a șterge alergia corespunzătoare atât din tabel cât și din vectorul de alergii al utilizatorului în baza de date.

Allergy Name	
Penicilin	X
Cats	X
Seashell	X

figura 4.3.3

Pentru a popula selectorul cu alergii disponibile, am iterat prin fiecare element din colecția "Allergies" și am verificat dacă se află sau nu deja în lista de alergii a utilizatorului. Dacă acesta nu suferă deja de această alergie, am utilizat metoda „createElement()" pentru a crea un element „<option>". Apoi, am utilizat metoda "appendChild()" pentru a adăuga opțiunea creată în selector.

```

if (!isExistingAllergy) {
  var option = document.createElement('option');
  option.value = allergyData.name;
  option.textContent = allergyData.name;
  option.setAttribute('data-allergy-id', doc.id); // Set the ID as a custom data
  attribute
  allergySelect.appendChild(option);
}

```

Setăm valoarea "textContent" a elementului „option” cu numele alergiei și se adaugă un atribut personalizat "data-allergy-id" cu valoarea „doc.id”, care reprezintă ID-ul din baza de date al alergiei. Astfel, se păstrează o referință către alergie pentru a o putea adăuga ulterior la profilul utilizatorului.

figura 4.3.4-selectarea unei noi alergii

După ce am obținut valoarea selectată, o vom putea adăuga la profilul utilizatorului când se apasă butonul "Add".

figura 4.3.5

Butonul "Update" este utilizat pentru a actualiza datele existente ale utilizatorului pe baza tuturor modificărilor făcute în formularul de editare. Când administratorul apasă pe butonul "Update", urmează să fie preluate valorile actualizate din formular.

Pentru a obține valorile actualizate, accesăm elementele HTML corespunzătoare și extragem noile valori introduse. Folosim fișierul "formValidation.js" pentru a valida datele introduse în formularul de editare și apoi le trimitem către Firestore.

```
if (validateForm()) {  
  
    // Get the form values  
    var updatedData = {  
        first_name: updateForm.firstName.value,  
        last_name: updateForm.lastName.value,  
        gender: updateForm.gender.value,  
        blood_type: updateForm.bloodType.value,  
        birth_date: updateForm.birthDate.value,  
    };  
  
    // Update the Firestore document  
    userDocRef.update(updatedData)  
        .then(function () {  
            showAlert('User data updated successfully.', 'success');  
        })  
        .catch(function (error) {  
            showAlert('Error updating user data.', 'error');  
        });  
}
```

#### 4.3.2 Aăugarea unui card nou

Pentru adăugarea unui nou card și a informațiilor specifice utilizatorului ce îl deține folosim o abordare similară celei pentru modificare.

De data aceasta primul pas este introducerea unui ID valid și unic [figura 4.3.6]. Pentru a verifica dacă id-ul este unic folosim o tehnică asemănătoare celei pentru verificarea unicității alergiilor de mai sus iar pentru verificarea corectitudinii codului introdus folosim o expresie regulată cu următoarele criterii:

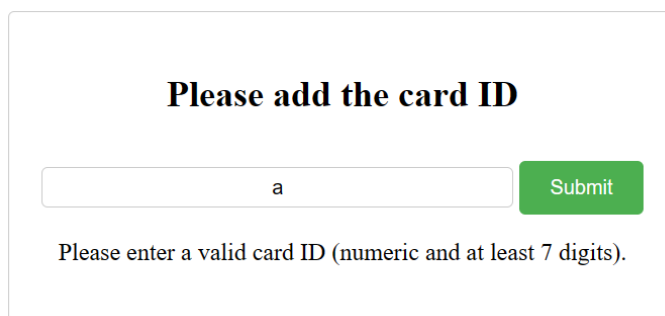
^ - începutul șirului de caractere

[0-9] - un caracter numeric

{7,} - cel puțin 7 astfel de caractere consecutive

\$ - sfârșitul șirului de caractere

```
if (!/^[0-9]{7,}$/ .test(cardId)) {  
    messageElement.textContent = 'Please enter a valid card ID (numeric and at least  
7 digits).';  
    return;  
}
```



**Please add the card ID**

Please enter a valid card ID (numeric and at least 7 digits).

figura 4.3.6

Dacă ID-ul introdus este unic și conform vom naviga către aceeași pagină HTML, de data aceasta necompletată, ca pentru modificări dar cu un fișier JavaScript diferit.

```
<script>
  var urlParams = new URLSearchParams(window.location.search);
  var id = urlParams.get('id');
  var userDocRef = firebase.firestore().collection('Users').doc(id);

  userDocRef.get().then(function (doc) {
    if (doc.exists) {
      var script = document.createElement('script');
      script.src = 'existingUser.js';
      document.body.appendChild(script);
    } else {
      // User is new, load newUser.js
      var script = document.createElement('script');
      script.src = 'newUser.js';
      document.body.appendChild(script);
    }
  }).catch(function (error) {
    console.log('Error fetching user data:', error);
  });
</script>
```

Iar apoi ne comportăm similar cu procesul de modificare a datelor.

## Capitolul 5

### Concluzii

Prin proiectul "TakeCare", ne-am propus să dezvoltăm o aplicație care să acopere în profunzime nevoia de acces la informații în timp real. Scopul principal al proiectului a fost îmbinarea mai multor tehnologii pentru a gestiona automat datele din spitale și pentru a îmbunătăți accesul la informații medicale în timp real.

Am dorit ca implementarea acestui proiect să aibă un impact semnificativ asupra creșterii eficienței în gestionarea datelor medicale și îmbunătățirea accesului la informații medicale de înaltă calitate. Utilizatorii beneficiază acum de informații în timp real privind disponibilitatea locurilor din spitale, a ratingului acestora, a distanței fața de locația lor curentă și multe altele, ceea ce le permite să ia decizii mai informate în ceea ce privește accesul la serviciile medicale.

Proiectul "TakeCare" și-a propus astfel să reprezinte un pas important în direcția unei asistențe medicale mai accesibile, mai transparente și mai orientate către nevoile individuale ale pacienților.

## Bibliografie și webografie

- [1] (2021) State of Health in the EU Romania: [https://health.ec.europa.eu/system/files/2022-01/2021\\_chp\\_romania\\_romanian.pdf](https://health.ec.europa.eu/system/files/2022-01/2021_chp_romania_romanian.pdf)
- [2] Barry A. Burd, Java Programming for Android Developers. For Dummies, 2nd Edition. October 28, 2016
- [3] Conf. Lucian M. Sasu, P. (2 iunie 2022). Medii vizuale de programare. 12.
- [4] **Desenarea pe harta GoogleMaps**  
<https://developers.google.com/maps/documentation/android-sdk/reference/com/google/android/libraries/maps/model/BitmapDescriptor>
- [5] **RecyclerView și ViewHolder**  
<https://developer.android.com/reference/androidx/recyclerview/widget/RecyclerView.ViewHolder>
- [6] **Despre Firebase**  
*GeeksForGeeks*: <https://www.geeksforgeeks.org/firebase-introduction/>
- [7] <https://www.educative.io/answers/what-is-firebase>
- [8] <https://firebase.google.com/docs/firestore/query-data/listen>
- [9] **RaspberryPi**  
 Edgardo Peregrino · 2023 Raspberry Pi in 30 Days
- [10] [The Pi4J Project – Pin Numbering - Raspberry Pi 3B+](#)
- [11] Soham Kamani ,Full Stack Web Development with Raspberry Pi 3,(2017)
- [12] **Despre RFID**  
<https://lastminuteengineers.com/how-rfid-works-rc522-arduino-tutorial/>
- [13] [RC522 RFID Tag Reading with the Raspberry Pi - Raspberry Pi Spy \(raspberrypi-spy.co.uk\)](#)
- [10] **Html, CSS și Javascript**  
 Learning Web Design: A Beginner's Guide to HTML, CSS, JavaScript, and Web GraphicsProgramming
- [11] Robin WieruchThe, Road to React with Firebase(2019)