

Universitatea
Transilvania
din Brașov
**FACULTATEA DE MATEMATICĂ
ȘI INFORMATICĂ**

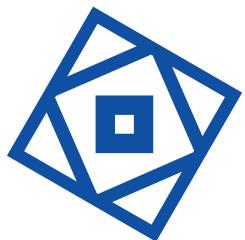
Programul de studii:
Informatică aplicată

LUCRARE DE DIPLOMĂ

Absolvent: Mihaiu Iulia-Ana-Maria

Coordonator: Lector universitar dr. Bocu Răzvan

Brașov, 2020



Universitatea
Transilvania
din Brașov
FACULTATEA DE MATEMATICĂ
ȘI INFORMATICĂ

LUCRARE DE DIPLOMĂ

Aplicație mobilă pentru gestiunea unui
cabinet stomatologic

Absolvent: Mihaiu Iulia-Ana-Maria

Coordonator: Lector universitar dr. Bocu Răzvan

Brașov, 2020

Cuprins

| | |
|---|-----------|
| 1 Introducere | 3 |
| 1.1 Context | 3 |
| 1.2 Motivația alegerii temei | 3 |
| 1.3 Cerințe funcționale și nefuncționale | 5 |
| 1.3.1 Cerințe funcționale | 5 |
| 1.3.2 Cerințe nefuncționale | 6 |
| 2 Descrierea tehnologiilor utilizate | 7 |
| 2.1 Limbajul Dart | 7 |
| 2.2 Firebase | 9 |
| 3 Proiectarea și implementarea aplicației | 16 |
| 3.1 Arhitectura aplicației | 16 |
| 3.2 Proiectarea bazei de date | 20 |
| 3.3 Implementare | 23 |
| 3.4 Evaluare | 30 |
| 4 Scenarii de utilizare | 35 |
| 4.1 Scenarii pacienți | 35 |
| 4.1.1 Scenariul 1 (Rezervare consultație) | 35 |
| 4.1.2 Scenariul 2 (Informații clinică) | 37 |
| 4.1.3 Scenariul 3 (Consultare istoric medical și programări cu-rente) | 37 |
| 4.2 Scenarii doctor | 39 |
| 4.2.1 Scenariul 1 (Confirmarea/refuzarea cererilor pentru con-sultații) | 39 |
| 4.2.2 Scenariul 2 (Furnizarea informațiilor preliminare) | 39 |
| 4.2.3 Scenariul 3 (Modificarea statusului unei consultații) | 40 |
| 4.2.4 Scenariul 4 (Accesarea istoricului consultațiilor unui pa-cient) | 41 |
| 4.2.5 Scenariul 5 (Salvarea detaliilor consultației) | 43 |

| | |
|---|-----------|
| 5 Concluziile lucrării și munca viitoare | 44 |
| 5.1 Concluzii | 44 |
| 5.2 Muncă viitoare | 45 |

Capitolul 1

Introducere

1.1 Context

Ideea dezvoltării unei aplicații mobile pentru gestiunea unui cabinet stomatologic și-a făcut loc în mintea mea în momentul în care am fost nevoită să iau loc în sala de așteptare a unui cabinet medical de cartier, ocazie cu care am observat că medicul stomatolog era singura persoană care manageria timpul și resursele avute la dispoziție, asistenta neavând decât un rol de suport. Astfel, am înțeles că există o legătură între timpul de reacție la apelurile telefonice, răsfoirea unei agende cu programările clientilor și timpul dedicat exclusiv manoperelor stomatologice.

Dezvoltarea aplicațiilor de asistență medicală este o zonă care se dezvoltă cu rapiditate. Aceste aplicații permit industriei medicale să fie mai eficientă, devenind mai puțin costisitoare.

În aceste zile, indiferent de ce dorești să faci te întrebă dacă există, de fapt, o aplicație pentru asta. Oamenii s-au atașat de device-urile lor nu numai pentru că le permit să-și verifice rețelele de socializare și să trimită prin e-mail oriunde ar merge, ci și pentru că pot face cumpărături, pot face programări, își pot face serviciile bancare și o mulțime de alte sarcini pentru care au nevoie de mai mult timp.

Aplicațiile mobile au devenit o parte din viața noastră de zi cu zi și sunt unul dintre principalele instrumente de sensibilizare a unei mărci, a unei afaceri.

1.2 Motivația alegerii temei

Practica stomatologică este o afacere și, ca și alte afaceri din zilele noastre, trebuie să fie reprezentată în spațiul digital pentru a îmbunătăți experiența cli-

entului. Printre motivele pentru care ar trebui creată o aplicație mobilă pentru practica stomatologică se află:

■ Utilitatea

Toate informațiile despre medic, clienți și planificări, manopere se regăsesc într-un singur loc. Se pot face orice modificări prin intermediul unui smartphone, tablete sau desktop.

■ Mobilitatea

În ziua de azi oamenii sunt mai aglomerati și posibilitatea de a vizualiza date pe device pare a fi cea mai atractivă. Totodată, are un efect bun asupra imaginii afacerii, dacă are o aplicație mobilă.

■ Feedback-ul

Pacienții pot evalua calitatea serviciilor de care au beneficiat prin postarea unor recenzii practice în aplicația clinicii. Astfel, un viitor pacient ar putea obține date despre activitatea clinicii din aplicația dedicată fără prea mult efort.

■ Înlocuirea suportului de hârtie cu unul digital

Datorită aplicării acestui concept, pacienții vor putea încărca unele dintre fișierele necesare.

■ Interactivitatea

Pentru unii oameni, pare înfricoșător și dureros să meargă la dentist. Aplicațiile mobile pot redirecționa atitudinile oamenilor către un mod mai pozitiv, datorită interacțiunilor, elementelor de gamificare, culorilor vii și a diferitelor note informative.

O aplicație mobilă este cea mai bună opțiune pentru cabinetele stomatologice care doresc să-și propulseze afacerea la nivelul următor. Ajută clinicele să modernizeze procesele administrative, precum și să reducă costurile. Mai mult, o aplicație mobilă pentru practica stomatologică poate crește implicarea pacienților, oferindu-le posibilitatea unei comunicări în timp real.

Scopul

Scopul acestei aplicații este reprezentat în primul rând de o strategie de business (vizibilitatea/notorietatea clinicii trebuie să depășească limitele cartierului), prin atragerea de noi clienți/activarea vechilor clienți, dar și de obținerea

rezultatelor scontate: creșterea randamentului pe client și respectiv maximizarea profitului cu minim de resurse. Pentru a rezolva aceste necesități, am creat aplicația denumită **Toothly**.

1.3 Cerințe funcționale și nefuncționale

În această secțiune, definim cerințele funcționale și nefuncționale pentru aplicația Toothly. Această aplicație va reduce semnificativ resursa de timp consumată cu activitățile administrative (call-center, programare) în detrimentul celor profesionale, va crește loialitatea/fidelitatea pacienților actuali și va atrage noi clienți.

Pentru a formaliza nevoile utilizatorilor și obiectivele aplicației, specificăm cerințele sale funcționale și nefuncționale.

O cerință funcțională descrie ceea ce se dorește de la un sistem software să îndeplinească. Cerințele nefuncționale impun restricții cu privire la modul în care sistemul va face acest lucru.

1.3.1 Cerințe funcționale

Toothly oferă o serie de caracteristici care acceptă modelul de afaceri al stomatologiei. În special, Toothly ar trebui să abordeze nevoile a două părți interesate:

1. Pacient: rezervă consultații într-un mod rapid;
2. Pacient: consultă informațiile despre clinică;
3. Pacient: consultă informațiile medicilor;
4. Pacient: consultă istoricul medical;
5. Pacient: consultă/accesează consultațiile rezervate;
6. Pacient: este notificat instant cu privire la modificările unei consultații;
7. Pacient: verifică programările viitoare;
8. Doctor: confirmă consultațiile cu pacienții;
9. Doctor: anulează consultațiile cu pacienții;
10. Doctor: furnizează instrucțiuni preliminare pentru pacient (înainte de a merge la consultatie);

11. Doctor: salvează detalii ale consultației;
12. Doctor: verifică istoricul medical al unui pacient;
13. Doctor: verifică programările viitoare;
14. Doctor: este notificat instant cu privire la modificările unei consultații;

1.3.2 Cerințe nefuncționale

Cerințele nefuncționale ale aplicației Toothly sunt:

1. Dimensiune mică.

Dimensiunea aplicației trebuie să fie mai mică decât 50 MB, permitând utilizatorului să o descarce în mai puțin de un minut.

2. Latență scăzută.

Aplicația ar trebui să se scaleze în funcție de numărul de utilizatori, oferind în același timp latență scăzută.

3. Operații care au un cost scăzut.

Aplicația ar trebui să producă niște costuri monetare mici pentru o clinică de dimensiuni medii. În special, operațiile legate de aplicație ar trebui să coste mai puțin de 1 % din totalul veniturilor lunare.

4. Disponibilitate.

Sistemul ar trebui să funcționeze în condiții corespunzătoare, cu un timp de oprire minim.

5. Confidențialitate.

Soluția ar trebui să asigure confidențialitatea cu privire la conținutul care trebuie protejat. În special, un medic poate accesa doar datele pacienților săi.

6. Scalabilitate.

Soluția ar trebui să suporte un număr de utilizatori în continuă creștere.

7. Testabilitate.

Ar trebui să fie posibil să se testeze soluția într-un mediu diferit de cel de producție.

Capitolul 2

Descrierea tehnologiilor utilizate

În acest capitol, se vor introduce informații generale asupra tehnologiilor utilizate în acest proiect: **Flutter și Firebase**. Se va prezenta o descriere a Flutter-ului, precum și arhitectura și componentele sale principale. În ceea ce privește Firebase, sunt discutate componentele utilizate în acest proiect: **Firebase Authentication, Cloud Firestore, Firebase Cloud Messaging și Funcțiile Cloud**.

2.1 Limbajul Dart

Dart este un limbaj de programare modern cu sursă deschisă, orientat pe obiecte (POO), dezvoltat original de către Google, utilizat pentru dezvoltarea de aplicații web, server, desktop, dar și mobile [1].

Un aspect interesant cu privire la Dart este faptul că este unul dintre puținele limbaje de programare care este potrivit pentru a fi compilat atât AOT (Ahead Of Time), oferind predicții de cod nativ mult mai rapid, cât și JIT (Just In Time), pentru o compilare super rapidă a ciclurilor de dezvoltare și schimbând fluxul de lucru în mai bine (lucru de foarte mare ajutor în funcția de Hot-Reload din Flutter).

Totodată, Dart reușește să creeze animații și tranzitii mai cursive, care se redau cu până la 60 fps (cadre/secundă), acest lucru fiind realizat datorită faptului că limbajul Dart, ca și JavaScript, este single-threaded, adică fiecare proces se execută individual până la terminarea sa și mai apoi se trece la următorul proces. Astfel, dezvoltatorul se asigură că funcțiile importante (inclusiv animațiile și tranzitiiile) sunt executate până la finalizare, fără ca acestea să fie întrerupte. În special, Dart este optimizat pentru client și este utilizat pentru a dezvolta aplicații pe mai multe platforme.

De asemenea, acest limbaj poate face gestionări de resurse automat și fără limitări de acces, fapt ce maximizează performanțele aplicației.

Flutter

Flutter este un set de instrumente, dezvoltat de către Google, conceput pentru elementele de UI (User Interface), făcând posibilă crearea unor aplicații atractive, compilate nativ pentru mobil, web și desktop dintr-o singură bază de cod. Flutter folosește limbajul de programare Dart și oferă o abordare cross-platform pentru dezvoltarea aplicațiilor: în principiu, dezvoltatorii pot crea aplicații Android și iOS cu același cod.

Flutter abordează dezvoltarea aplicațiilor cross-platform într-o manieră diferită, oferind propriul său set de elemente UI și un motor care implementează animațiile, grafica, lucrul cu fișierele și rețeaua de internet, pe lângă multe alte pachete standard.

Flutter permite mai multe moduri de testare și depanare a aplicațiilor Flutter [2, 3]. DevTools este o suită de instrumente care rulează online, ce permit implementarea unei aplicații Flutter. DevTools permite depanarea și profilarea memorilor. Mai multe elemente pot fi depanate: straturile de aplicații (widget-uri), animații și altele. Inspectorul Flutter pentru widget-uri permite să explorez arborii de widget-uri. Acest lucru ajută la înțelegerea aspectelor existente și la diagnosticarea erorilor pe layout-uri.

Widgets

Widget-urile în Flutter sunt unitatea de bază care compune o interfață de utilizator. Widget-urile descriu aspectul lor, având în vedere configurațiile și starea. La schimbarea stării, partea din widget care trebuie schimbată este reconstruită.

Există două tipuri de widget-uri în Flutter: stateless și stateful. Denumirile sunt intuitive, fiind evident că widget-ul stateless este exact ceea ce denuște, un widget fără o stare a datelor. Widget-urile stateless primesc argumente de la widget-ul părinte. Când widget-ul este construit, folosește aceste argumente pentru inițializarea sa. Cu toate acestea, widget-urile stateless nu iau în considerare interacțiunea cu utilizatorul.

Un lucru de remarcat este faptul că ambele tipuri de widget-uri se comportă la fel, adică se reconstruiesc cu fiecare cadru, diferență constând în obiectul State din widget-ul stateful, în care se salvează starea widget-ului în fiecare cadru și o restabilește.

Widget-urile de bază, care fac parte din majoritatea aplicațiilor sunt considerate următoarele:

- Text: creează text
- Row: creează un layout în direcția orizontală
- Column: creează un layout în direcția verticală
- Stack: plasează widget-uri unul peste altul
- Container: creează un element vizual dreptunghiular.(De obicei, containerul include alte câteva widget-uri.)

Un dezvoltator poate utiliza widget-uri personalizate pentru a construi aplicații, de exemplu Material Design Widget. Există o piață uriașă pentru widget-uri, numită Widget Catalog [4]. Un widget important este widget-ul Navigator. Navigator permite tranziția între ecranele aplicației. Widget-urile sunt unitatea fundamentală a Flutter-ului și sunt gestionate folosind ciclul de viață al widget-urilor. Un widget stateful este inițializat atunci când se apelează `createState()`. Widget-ul este creat și este disponibil pentru a fi utilizat. Când nu mai este nevoie de un obiect de stare, funcția `dispose()` poate fi apelată pentru a-l șterge.

2.2 Firebase

Firebase este platforma Google pentru dezvoltare de aplicații, care te ajută să creezi, să îmbunătățești și ulterior să extinzi aplicația personală [5, 6]. Pentru că dezvoltatorii preferă să se focuseze mai mult pe interacțiunea/experiența cu aplicația în sine, platforma aceasta oferă un set de instrumente, acoperind o plajă largă de servicii, pe care dezvoltatorii ar fi trebuit, în mod normal, să le implementeze de la zero. Printre aceste instrumente se numără: autentificare, analiză de date, baze de date, configurare și stocare de fișiere.

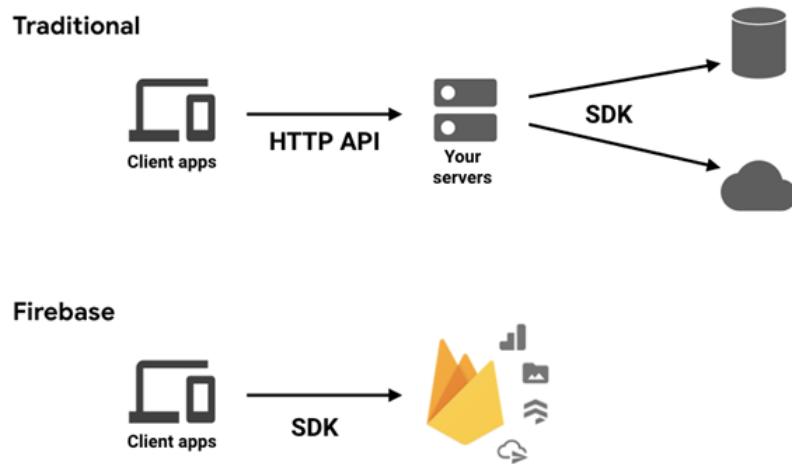


Figura 2.1: Modelul tradițional vs Modelul cu Firebase

Serviciile sunt găzduite în cloud, fiecare produs având componente de back-end care sunt menținute și acționate de către Google, iar SDK-urile (Software Development Kit) clientului oferite de Firebase interacționează direct cu aceste servicii de back-end, fără a avea nevoie de un intermediar între aplicație și serviciu. Totodată, accesul administrativ către fiecare dintre aceste produse se face prin intermediul consolei Firebase.

Firebase poate fi considerată ca BaaS (back-end ca serviciu) sau PaaS (Platform as a Service). Providerii BaaS permit găzduirea și gestionarea back-end-urilor pentru aplicații (în mare parte mobile). PaaS este o platformă de calcul găzduită pe Cloud care oferă timpul de rulare și mediul de lucru pentru dezvoltatori, aşa cum este descris mai sus. Firebase permite mai multe caracteristici decât furnizarea unui back-end [7]. Prin utilizarea unui PaaS, este posibilă reducerea considerabilă a timpului de dezvoltare, prin externalizarea unor sarcini care ar necesita multe resurse temporale și umane către Firebase (cum ar fi autentificarea, baza de date și partea analitică). Prin urmare, nu este nevoie să gestionăm infrastructura, permitând dezvoltatorilor să se concentreze pe aplicații.

Unul dintre avantajele Firebase, comparativ cu alte PaaS, precum Heroku sau Salesforce, este faptul că oferă o bază de date NoSQL fiabilă și are o comunitate activă în creștere. De fapt, Firebase reprezintă mai mult de 40 % din cota de piață a PaaS [8].

Firebase Authentication

Firebase Authentication [9] permite gestionarea proceselor de control al accesului clientilor. Controlul accesului reprezinta restrictia selectiva la un set de resurse, incluzand autentificare, autorizare si raspundere. In autentificare, un client ofera doveda identitatii sale (de exemplu, printr-o combinatie de utilizator-parola). Autorizarea se intampla atunci cand sistemul de control al accesului acorda sau refuză accesul la resurse. Sistemele de control al accesului pot acorda sau refuza unui client autenticat executarea unei anumite actiuni referitoare la o resursa, asa cum este prezentata in politicile de control de acces corespunzatoare. Politicile de control de acces sunt seturi de reguli care determina cand un utilizator specific ar trebui sa aiba acces la o anumita resursa. Aceste politici de control de acces sunt implementate in logica de afaceri si in regulile de acces la baza de date (explicate ulterior).

Firebase Authentication este un serviciu back-end construit pe baza proto-coalelor OAuth 2.0 si OpenID Connect. OpenID Connect 1.0 [10] este un strat de identitate construit deasupra protocolului OAuth 2.0, care permite verificarea identitatii utilizatorului final. OAuth 2.0 este protocolul standard al industriei pentru autorizare [11, 12]. Protocolul defineste fluxurile de informații intre client și server, care se bazeaza, de obicei, pe JSON Web Tokens (JWT) [13, 14]. JSON Web Token (JWT) este un mijloc compact, cu URL sigur, de a reprezenta claim-urile care trebuie transferate intre doua parti.

Figura 2.2 prezinta claim-urile corespunzatoare (in partea dreapta a figurii) asociate cu token-ul (partea stanga a figurii). Pe langa claim-uri, jetonul contine o structura JWS (JSON Web Signature), care permite activarea claim-urilor. Intrucat token-urile nu pot fi modificate cu usurinta (pentru ca sunt semnate digital), acest proces de activare a claim-urilor permite aplicatiilor sa identifice clientii si sa restricioneze sau sa le permita accesul la anumite servicii, pe baza claim-urilor pe care acesteia le detin. De exemplu, ar putea fi de dorit sa se permita numai privilegii de administrare pentru utilizatori cu o revendicare "rol: admin".

Pentru ca un utilizator sa se poate conecta in aplicatie, mai intai se preiau datele de autentificare de la utilizator. Aceste credintiale pot fi adresa de email a utilizatorului si parola sau un token OAuth de la un provider de identitate federata. Apoi, aceste credintiale se trimit catre Firebase Authentication SDK. Serviciile de back-end vor verifica apoi acele credintiale si vor returna un raspuns clientului.

Firebase Authentication permite simplificarea acestor procese: dupa inregistrare, se poate crea un token de autentificare care contine aceste claim-uri. Acest token permite accesarea informatiilor utilizatorului, oferind o metoda fidelă pentru desfășurarea proceselor de autorizare.

| | |
|--|--|
| <pre>eyJhbGciOiJSUzI1NiIsImtpZCI6ImMzjI3NjU0 MmJmZmU0NWU5OGMyMGQ2MDN1YmUyYmExMTc2ZWRh MzMlCJ0eXAi0iJKV1QifQ.eyJyb2xIjoiY2xpZ W50IiwiaXNzIjoiaHR0cHM6Ly9zZWN1cmV0b2tlb i5nb29nbGUuY29tL3Rvb3RobHktYTg1ODEiLCJhd WQi0iJ0b290aGx5LWE4NTgxIiwiYXV0aF90aW1I joxNTkyODc2NTc0LCJ1c2VyX2lkIjoiVWxLTmtIb U1INmJENHNSWVhT0VppeHlGa1d0MiIsInN1YiI6I 1VsS05rSG1NSDZiRDRzU1lyUzlaaXh5RmtXTjIiL CJpYXQiojE1OTI4NzY1NzQsImV4cCI6MTU5Mjg4M DE3NCwiZW1haWwiOijpdWxpYS5taWhhaXU50EBnb WFpbC5jb20iLCJ1bWFpbF92ZXJpZml1ZCI6dHJ1Z SwiZmlyZWJhc2UiOnsiaWR1bnRpdGllcyI6eyJlb WFpbCI6WyJpdWxpYS5taWhhaXU50EBnbWFpbC5jb 20iXX0sInNpZ25faW5fcHJvdmIkZXIi0iJwYXNzd 29yZCJ9fQ.P4y0QAzu_nv2Vks38rMI- H0jI5Er6hPDVVVh9Ai2JAcBWWbBaR- 9p7U46Rd1fWksql_e6xGhgrLHL9wVbej3rJlirmb WWqPPqf7Us7ec8mTcZU2cUafTm--8SwWZIg0- j921WmG7S4YdyzxdkJ2E0VgGg-ahs2F_1- E8PjoHxSU74zksAB2kdhGQNimEQhbCqC01zqAkhr pi6Ld-</pre> | <p>HEADER: ALGORITHM & TOKEN TYPE</p> <pre>"alg": "RS256", "kid": "c3f276542bffe45e98c20d603ebe2ba1176eda33", "typ": "JWT" }</pre> <p>PAYOUT: DATA</p> <pre>{ "role": "client", "iss": "https://securetoken.google.com/toothly-a8581", "aud": "toothly-a8581", "auth_time": 1592876574, "user_id": "U1KNKhmMH6bD4sRYXS9ZixyFkWN2", "sub": "U1KNKhmMH6bD4sRYXS9ZixyFkWN2", "iat": 1592876574, "exp": 1592880174, "email": "iulia.mihaiu98@gmail.com", "email_verified": true, "firebase": { "identities": { "email": ["iulia.mihaiu98@gmail.com"] }, "sign_in_provider": "password" } }</pre> |
|--|--|

Figura 2.2: Partea a unui JWT emis, apartinând unui utilizator Firebase [14]

Cloud Firestore

Cloud Firestore este o bază de date NoSQL orientată pe documente și găzduită în cloud. Spre deosebire de bazele de date SQL, aceasta nu are tabele sau rânduri, ci, în schimb, datele se stochează sub formă de documente, fiecare document fiind identificat prin denumirea sa și conținând un set de perechi de tipul cheie-valoare. Documentele sunt stocate la rândul lor în colecții.

Totodată, Cloud Firestore are câteva **funcționalități** care ne pot fi foarte utile.

■ Modelul de date

Documentele pot conține subcolecții și obiecte imbricate, fiecare dintre acestea putând include câmpuri de tip primitiv sau obiecte complexe ca liste.

■ Interogări

Se pot utiliza interogări ce extrag individual documente specifice sau extrag toate documentele dintr-o colecție, în funcție de parametrii datei interogării.

Interogările pot include filtre înlănțuite multiplu și, de asemenea, să combine filtrarea cu sortarea. Fiind indexate în mod implicit, face ca performanța executării interogării să crească semnificativ.

■ **Modificări în timp real și suport pentru offline**

Cloud Firestore utilizează sincronizarea datelor pentru a modifica în timp real informația de pe fiecare dispozitiv conectat.

Cât despre suportul pentru offline, serviciul depozitează datele pe care aplicația le utilizează în mod constant, pentru ca aplicația să poată vizualiza, modifica sau interoga datele chiar dacă dispozitivul nu este conectat la internet. Când acesta restabilește conexiunea la internet, Cloud Firestore sincronizează orice schimbări locale, în ordinea în care acestea au fost făcute.

Cloud Firestore este o alegere populară a bazelor de date pentru dezvoltatorii care lucrează cu Firebase, folosită chiar și în studii științifice [15].

Firebase Cloud Messaging

Firebase Cloud Messaging este o soluție de trimitere a mesajelor, care permite notificarea unei aplicații client. Notificările sunt bazate pe evenimente, care necesită un declanșator(trigger)[16]. În timp ce principalele obiective ale notificărilor sunt păstrarea clientilor activi și interactivitatea, obiectivul notificărilor poate fi acela de a susține caracteristicile esențiale ale unei aplicații.

Figura 2.3 prezintă diferite modalități de a trimite notificări: prin consola din Firebase sau programatic (prin apeluri HTTP la Firebase). În mod obișnuit, notificările sunt generate de funcții, și nu manual, deoarece acestea ar putea suporta costuri mari în ceea ce privește resursele umane.

Cloud Functions

Cloud Functions este un framework serverless care expune un back-end, pe care aplicațiile Flutter îl pot utiliza [17]. În Cloud Functions, este posibilă definirea funcțiilor scrise în limbajele de programare Javascript sau Typescript, care gestionează evenimentele declanșate de funcțiile Firebase, cum ar fi Cloud Firestore.

Avantajul utilizării Cloud Functions este că dezvoltatorul nu are nevoie să configureze un serviciu back-end care să susțină aplicația lor: gestionarea infrastructurii este făcută de Google. Spre deosebire de alte servicii Firebase, Cloud Functions sunt complet separate de client - acesta este motivul pentru

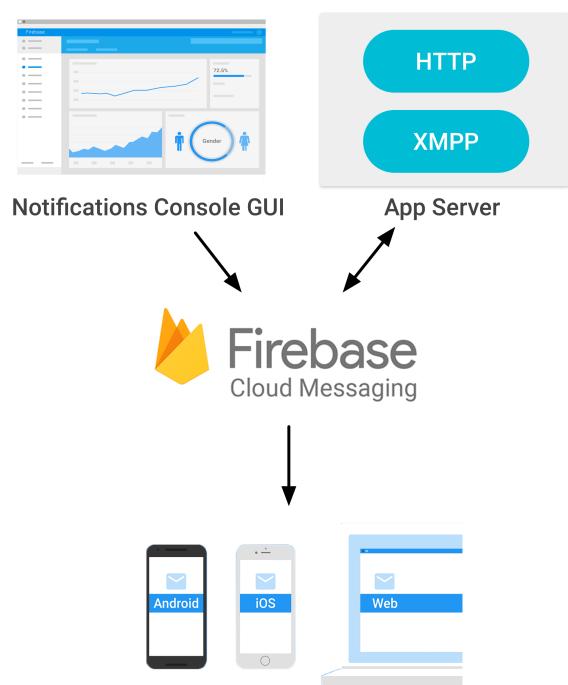


Figura 2.3: Notificări generate print diferențiate mijloace [16]

care ar trebui plasată logica de afaceri aici. În plus, Google scalează automat infrastructura, în funcție de gradul utilizării aplicației.

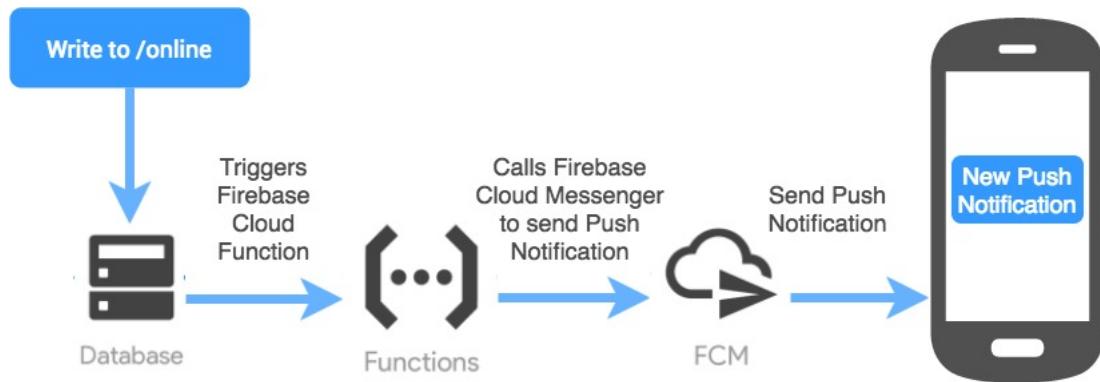


Figura 2.4: Posibile utilizări ale serviciului Cloud Functions [18]

Figura 2.4 reprezintă o utilizare a serviciului Cloud Function. În acest exemplu, o funcție cloud monitorizează colecția “online”, declanșând un eveniment când un nou document este creat, un document este actualizat sau un document este șters. Acest eveniment declanșează Firebase Cloud Messaging, care creează o notificare push și o trimite către aplicația mobilă.

Capitolul 3

Proiectarea și implementarea aplicației

În acest capitol se va prezenta o descriere amănunțită a sistemului aplicației dezvoltate, urmărind: soluțiile folosite pentru a proiecta arhitectura, implementarea și toate modulele utilizate. Astfel, se va descrie în detaliu fiecare dintre componentele care formează acest proiect.

3.1 Arhitectura aplicației

Pentru că Flutter este un toolkit modern și open-source, comunitatea se mărește pe zi ce trece, lucru ce face ca o multitudine de opinii legate de cele mai bune practici să fie discutate în mod continuu. Astfel că, nu s-a ajuns la un consens privind o arhitectură generală, lăsând deci la latitudinea dezvoltatorilor modelarea arhitecturii într-un mod corespunzător/favorabil funcționalităților pe care aceștia doresc să le implementeze ulterior.

În figura 3.1 este ilustrată schematic arhitectura sistemului. După cum putem vedea, aceasta este alcătuită din două componente: cea locală și cea externă.

■ Componența locală

Componența locală conține trei straturi: UI/Presentation Layer (reprazentat de widget-uri), Logic Layer (reprazentat de controlerile pentru elementele de UI), Domain Layer (reprazentat de Model și Servicii), la care se adaugă și pachetele Dart și Flutter.

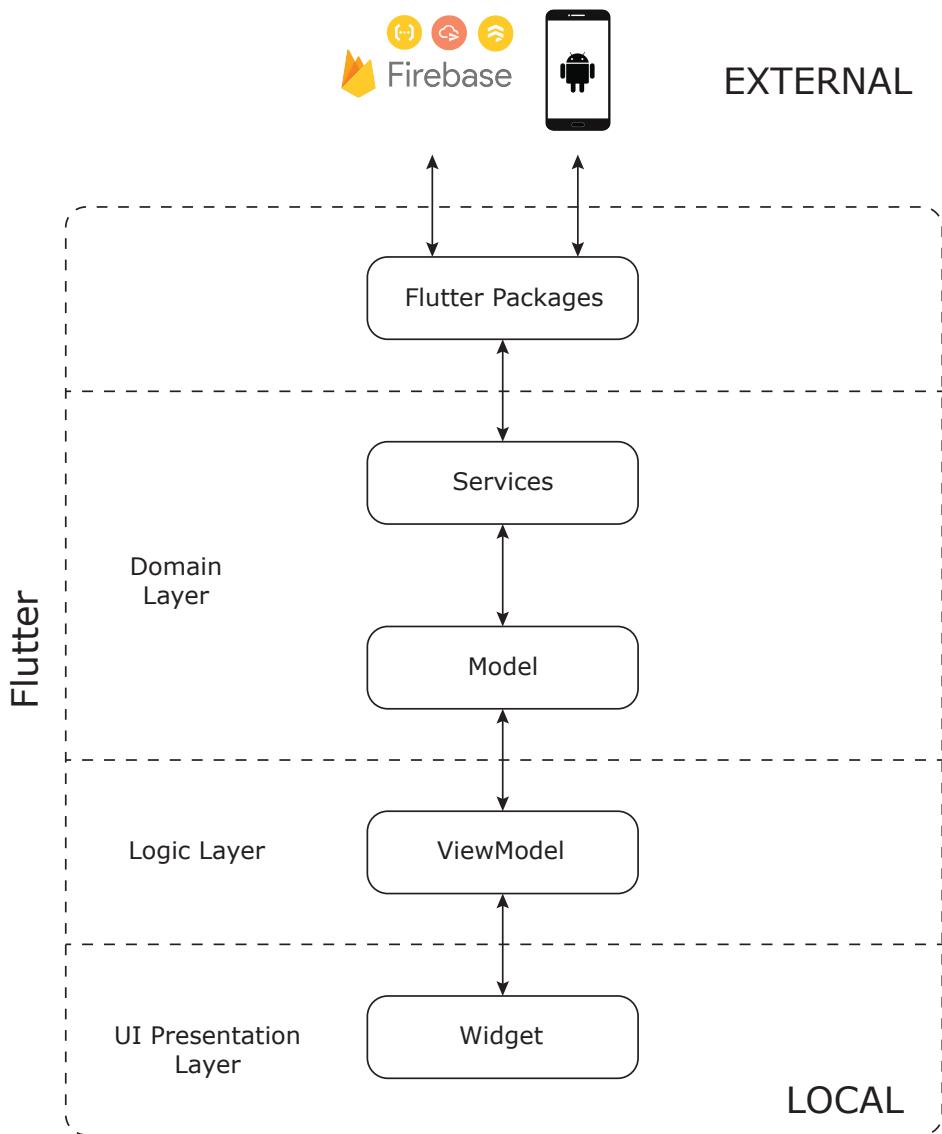


Figura 3.1: Prezentarea arhitecturii aplicației cu Flutter&Firebase

UI/Presentation Layer

După cum am mai precizat și în capitolul anterior, conceptul de bază al Flutter-ului este: „În Flutter, totul este un widget”. Widget-urile sunt în esență elemente UI, utilizate pentru a crea interfața cu utilizatorul aplicației.

În Flutter, aplicația este ea însăși un widget, fiind widget-ul de la cel mai

înalt nivel, iar UI-ul său se construiește folosind unul sau mai mulți copii (tot widget-uri), care se reconstruiesc la rândul lor folosind widget-urile copiilor lor. Acest principiu, de compozitie, ne ajută să creăm o interfață cu utilizatorul de orice complexitate.

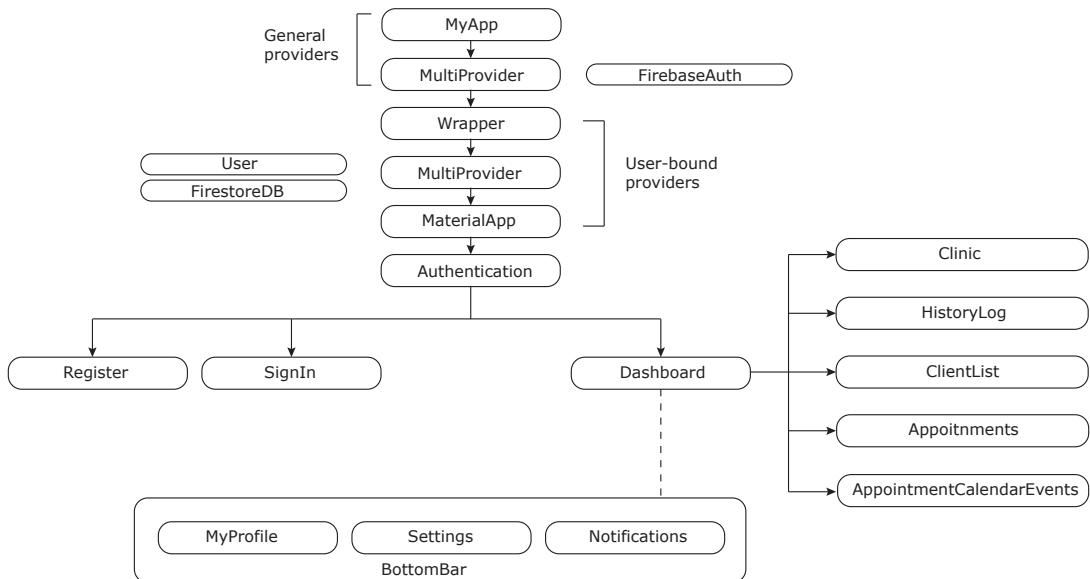


Figura 3.2: Diagrama simplificată a arborelui de widget-uri în aplicația Toothly

În figura 3.2 am reprezentat într-un mod simplificat arborele de widget-uri al aplicației dezvoltate. Începând cu nivelul cel mai înalt, avem providerii generali, printre care se află providerul de autentificare Firebase. Prin intermediul acestuia putem apela funcțiile necesare conectării, înregistrării sau delogării utilizatorului.

Pe următorul nivel se află clasa de Wrapper, împreună cu multi-providerii ce necesită ca utilizatorul să fie logat pentru a putea accesa funcționalitățile lor. Acest widget are rolul de a redirectiona utilizatorul după verificarea credențialelor către ecranul corespunzător.

În cazul în care utilizatorul nu a rămas logat sau credențialele acestuia sunt invalide, va fi redirectionat către builder-ul Authentication spre a se conecta/inregistra. Când autentificarea are loc cu succes, utilizatorul va fi redirectionat către ecranul de Dashboard, ce va conține opțiunile corespunzătoare rolului său. Din acest ecran, prin intermediul unor GridCard-uri se navighează spre ecranul opțiunii alese (ex: Clinic, HistoryLog, ClientList, Appointments) sau se poate utiliza bara de jos a Scaffold-ului (disponibilă în tot subarborele Dashboard-ului),

pentru a putea naviga către ecranul profilului personal, ecranul notificărilor sau cel pentru setări.

Logic Layer

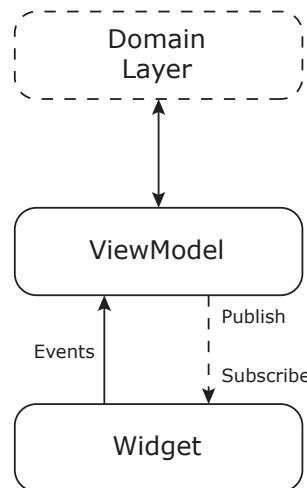


Figura 3.3: Diagrama fluxurilor

În figura 3.1, datele vin din mediul exterior și se propagă către servicii, ViewModels și până la widget-uri, iar în figura 3.3 putem vedea cum fluxul unui apel merge în direcția opusă. Widget-urile pot apela metode din cadrul ViewModels sau din cadrul serviciilor și, la rândul lor, acestea pot apela API-uri în pachetele Dart externe.

Componentele care fac parte din același strat al sistemului nu trebuie să știe despre existența componentelor aflate în straturile de mai jos. Astfel că, ViewModels nu importă niciun cod UI pentru problemă, în schimb, widget-urile se abonează ca ascultători, în timp ce modelele de vizualizare publică actualizări atunci când se schimbă ceva, reactualizând partea de UI.

Domain Layer

Serviciile pot transforma datele pe care le primesc prin intermediul pachetelor Dart externe și mai apoi să le pună la dispoziția celorlalte straturi prin API-uri specifice domeniului. De asemenea, nu pot avea o stare proprie sau să poată modifica starea unui widget. Astfel că, atunci când lucrăm cu Firebase putem

implementă un serviciu de serializare a datelor. Când se citesc date, acest serviciu transformă **stream**-uri (fluxuri de date) de perechi cheie-valoare, aduse din documentele aflate în Firebase, în modele de date imuabile.

Pentru a face widget-urile să aibă informația actualizată în timp real, cea mai simplă soluție este să folosim StreamBuilder-ul (un widget care se construiește pe baza ultimelor snapshot-uri primite din stream, când datele din stream se modifică, acest widget se reconstruiește).

În acest mod, deși obiectele sunt imuabile și serviciile nu conțin o stare, ViewModels se ocupă de schimbarea stărilor, ajutându-se de componenta de legătură StreamBuilder.

■ Componenta externă

Datorită utilizării serviciilor oferite de către Firebase, arhitectura este una „serverless”, ceea ce facilitează munca depusă de un dezvoltator care, în mod obișnuit ar fi trebuit să construiască un server, să implementeze API-uri web și mai apoi să facă și mențenanța acestuia. Astfel, Firebase ne ajută nu doar să minimizăm timpul de producție, ci și să maximizăm performanțele aplicației.

Acțiunile executate asupra bazei de date Cloud Firestore sunt în mod asincron realizate, folosind o instanță a bazei de date ca mecanism de conectare și logica aflată în Cloud Functions. Codul de back-end aflat în Cloud Functions se rulează automat, cererea/modificările se procesează, iar sistemul primește răspunsul și updatează UI-ul în timp real.

De asemenea, utilizând Firebase Cloud Messaging putem notifica clienții în legătură cu noi modificări produse în baza de date. Acest lucru necesită ca funcțiile de trimisere a notificărilor să fie implementate tot în Cloud Functions, fiind declanșate ulterior de modificările produse în anumite colecții din baza de date.

3.2 Proiectarea bazei de date

Baza de date a aplicației este reprezentată de Cloud Firestore, iar datele stocate în aceasta sunt protejate prin intermediul serviciului Firebase Authentication. Astfel, se permite accesul doar persoanelor autorizate. De asemenea, se recomandă utilizarea Cloud Firestore Security Rules pentru a seta permisiunile de citire și scriere, în funcție de rolul utilizatorului.

Utilizatorii care s-au înregistrat în aplicație pot fi ușor vizualizați în cadrul consolei Firebase configurată proiectului Toothly (figura 3.4). Crearea unui cont de utilizator se realizează utilizând o adresă de email și o parolă în cadrul aplicației, lucrul acesta putând fi realizat și direct din consolă. După crearea cu succes

a utilizatorului, acesta va avea atribuit un identificator unic, pe care îl vom putea utiliza ulterior ca ID pentru documentul corespunzător datelor personale ale utilizatorului.

The screenshot shows the Firebase Authentication console interface. At the top, there's a header with the project name "Toothly" and a "Go to docs" link. Below the header, the title "Authentication" is displayed, followed by a navigation bar with tabs: "Users" (which is selected), "Sign-in method", "Templates", and "Usage". A search bar is located above a table. The table has columns: "Identifier", "Providers", "Created", "Signed In", and "User UID". There are five rows of data in the table, each representing a user account with their email, provider (email), creation date, sign-in date, and unique User UID.

| Identifier | Providers | Created | Signed In | User UID |
|--------------------------|-----------|--------------|--------------|------------------------------|
| ana@yahoo.com | ✉ | Jun 20, 2020 | Jun 20, 2020 | HvzBEMr7wGYxUCPBKvSoCw7KF... |
| julia.mihaiu98@gmail.com | ✉ | Jun 20, 2020 | Jun 24, 2020 | UlKNkHmMH6bD4sRYXS9ZixyFk... |
| cristina@toothly.ro | ✉ | Jun 19, 2020 | Jun 23, 2020 | ZvrPuMwrbLRMtaDB9iTlfhnvgzv2 |
| ion@toothly.ro | ✉ | Jun 19, 2020 | Jun 20, 2020 | ezjD7Bia9vevuBmXbrshLSFRzFV2 |
| maramih.2007@yahoo.com | ✉ | Jun 19, 2020 | Jun 20, 2020 | zC4lZFTnM1WgZQ8uPBdrW6FmP... |

Figura 3.4: Consola Firebase Authentication

Pentru înregistrarea/autentificarea unui utilizator în aplicație, am implementat serviciul AuthService, care conține o instanță a clasei FirebaseAuth, prin intermediul căreia putem apela funcția de care avem nevoie: signInWithEmailAndPassword/createUserWithEmailAndPassword.

Exemplu logare utilizator:

```
final FirebaseAuth _auth = FirebaseAuth.instance;
AuthResult result = await _auth.signInWithEmailAndPassword(
  email: email, password: password);
FirebaseUser user = result.user;
```

Baza de date a aplicației Toothly conține patru colecții:

- **users**, colecție în care sunt salvate informații despre utilizator (spre exemplu: emailul, numele, numărul de telefon), fiecare document având ca ID uid-ul utilizatorului corespunzător; de asemenea, fiecare utilizator va avea o subcolecție de tokeni de tipul Firebase Cloud Messaging, utilizată pentru a putea trimite notificări acestora;

- **appointments**, colecție în care sunt salvate informațiile programărilor (de exemplu: id clientului și cel al doctorului, detalii despre consultație, statusul programării);

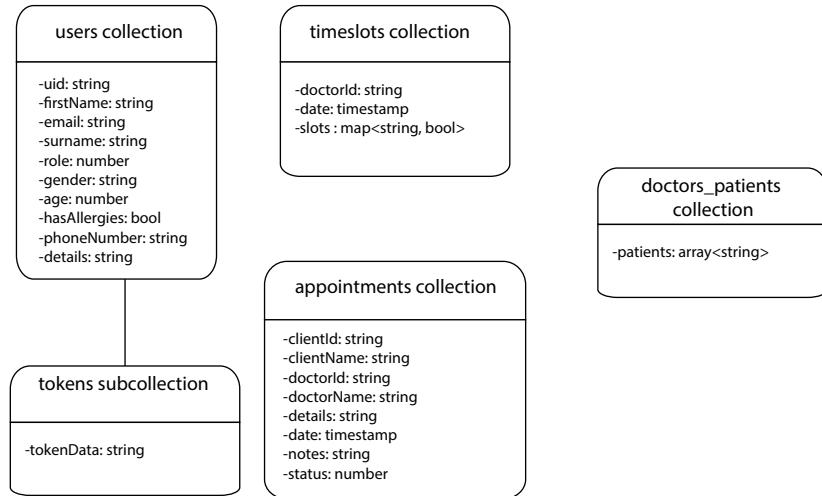


Figura 3.5: Colecții Cloud Firestore

- **doctors_patients**, este o colecție în care pentru fiecare doctor este stocată o listă a id-urilor utilizatorilor care îi sunt pacienți: documentul are ca ID uid-ul doctorului.

- **timeslots**, reprezintă o colecție în care fiecare document corespunde unei date și unui doctor, acestea conținând uid-ul doctorului, data și un map de ore în care doctorul este disponibil.

Datele salvate în Cloud Firestore pot fi vizualizate și în consola Firebase (figura 3.6). Acestea sunt stocate în format JSON și sunt sincronizate în timp real între utilizatorii aplicației. Pentru a prelua datele dintr-o colecție am implementat serviciul DatabaseService, care conține o instanță a clasei Firestore, prin intermediul căreia putem crea referințe la colecții, ca mai apoi să accesăm documentele care ne interesează sau să putem face interogări pentru a prelua datele necesare.

Exemplu:

```

final CollectionReference usersCollection =
    Firestore.instance.collection('users');
  
```

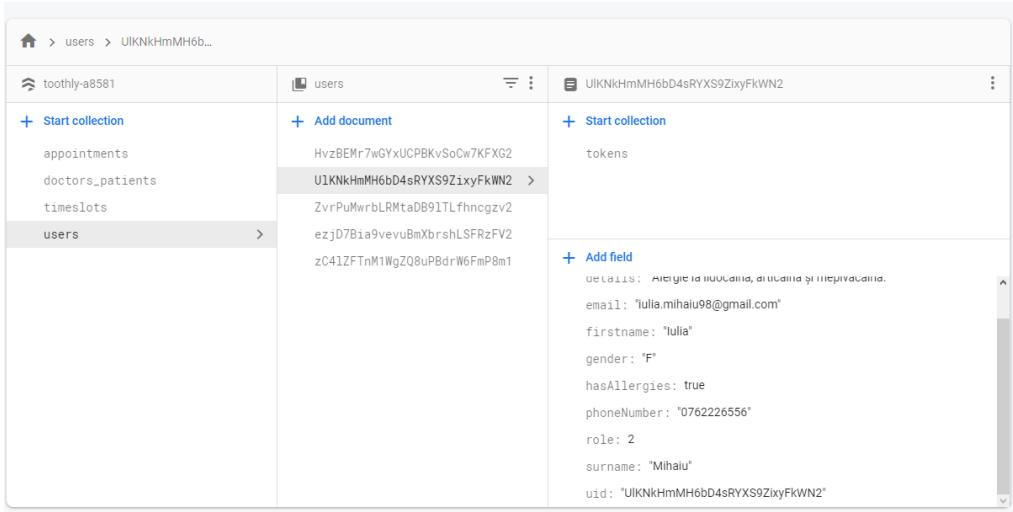


Figura 3.6: Consolă colecții Cloud Firestore

3.3 Implementare

În această secțiune se vor relata detaliiile legate de modul implementării componentelor aplicației.

Dacă în Android, View-ul este baza a tot ceea ce apare pe ecran (butoane, bare de instrumente, input pentru text, totul este un View), în Flutter echivalentul aproximativ al unui View este considerat widget-ul. Nu se poate pune un semn de egalitate între acestea însă, începând să implementezi în Flutter, cu timpul vei asocia widget-ul cu „modul de a declara și a construi interfața cu utilizatorul”.

În timp ce un View este desenat o singură dată și nu se redesenează până nu este apelată invalidarea, widget-urile au o durată de viață diferită: sunt imuabile și există doar până când trebuie schimbată. De fiecare dată când widget-urile sau starea lor se schimbă, Flutter creează un nou arbore de instanțe ale widget-urilor.

În Flutter, widget-urile sunt imuabile și nu pot fi actualizate direct, în schimb trebuie să lucrăm cu starea widget-ului. Așa s-a ajuns la conceptul de StatefulWidget și StatelessWidget. Denumirile sunt intuitive, fiind evident că StatelessWidget este exact ceea ce denumește, un widget fără o stare a datelor. Un lucru de remarcat este faptul că ambele tipuri de widget-uri se comportă la fel, adică se reconstruiesc cu fiecare cadru, diferența constând în obiectul State din StatefulWidget, în care se salvează starea widget-ului în fiecare cadru și o restabilește.

Dacă în Android se scriu layout-uri în XML, în Flutter layout-urile sunt re-

prezentate printr-un arbore de widget-uri.

Astfel că, widget-ul ales să stea la baza fiecărui ecran propriu-zis al aplicației mele este un Scaffold. Acesta este alcătuit din trei componente principale: o bară de aplicație (appBar), un corp (body) și o bară de navigație care se află în partea de jos a ecranului (bottomNavigationBar).

În componenta AppBar este specificată denumirea ecranului și, de asemenea, se pot adăuga acțiuni pe aceasta prin intermediul unor butoane de tip FlatButton.icon. Exemplu din clasa MyProfile:

```
AppBar(  
    title: Text('Profil utilizator'),  
    backgroundColor: Swatches.green2.withOpacity(1),  
    elevation: 0.0,  
    actions: <Widget>[  
        FlatButton.icon(  
            onPressed: () =>  
                Navigator.push(  
                    context,  
                    MaterialPageRoute(builder: (context) => EditForm()),  
                ),  
            icon: Icon(Icons.edit),  
            label: Text('Edit')),  
    ],  
);
```

Pentru componenta BottomBarNavigation am implementat o bară de navigare care va fi comună pentru toate ecranele aplicației pe care utilizatorul le poate accesa când este autentificat.

```
final bottomBar = Container(  
    height: 60.0,  
    child: BottomAppBar(  
        color: Swatches.green1.withOpacity(1),  
        elevation: 0.0,  
        child: Row(  
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,  
            children: <Widget>[  
                new LayoutBuilder(builder: (context, constraint){  
                    return \textbf{IconButton}(  
                        icon: Icon(Icons.dashboard, color: Colors.white),  
                        iconSize: iconSize,  
                        padding: EdgeInsets.symmetric(),
```

```

        onPressed: () {
            Navigator.of(context).pushNamedAndRemoveUntil('/dashboard', ModalRoute.withName('/sign_in'));
        },
    )),
    new LayoutBuilder(...),
    new LayoutBuilder(...),
    new LayoutBuilder(...)
],
),
),
),
);

```

Am inclus butoanele de navigare într-un widget LayoutBuilder pentru a putea transmite contextul în navigator, cu scopul de a ne redirecționa spre ecranul dorit.

În ceea ce privește componenta body a Scaffold-ului, aceasta diferă de la un ecran la altul, ea fiind Container-ul pentru conținutul copiilor din ierarhia Scaffold-ului. Mai exact, elementele pe care vrem să le vedem într-un ecran se vor adăuga în acea ierarhie.

În continuare voi vorbi despre widget-urile alese pentru a implementa componenta body în modulele aplicației mele.

Authenticate

Widget-ul Authenticate este folosit pentru a putea comuta simplu între ecranele SignIn și Register, prin intermediul unui boolean numit showSignIn. Prin intermediul unui RaisedButton toggle își va schimba valoarea și va anunța widget-ului să-și schimbe starea.

```

bool showSignIn = true;

void toggleView(){
    setState(() {
        showSignIn = !showSignIn;
    });
}

@Override
Widget build(BuildContext context) {
    if(showSignIn){

```

```

        return SignIn(toggleView: toggleView);
    }else{
        return Register(toggleView: toggleView);
    }
}

```

Atât în SignIn, cât și în Register widget-ul folosit este un Form. Acest form are o cheie globală ce va fi folosită pentru a verifica și valida câmpurile de completat, înainte de a se trimite și a fi procesate datele formularului.

Dashboard

În acest ecran am folosit GridView-ul. Pentru afișarea opțiunilor am creat widget-ul personalizat MyOptionWidget care are la bază un widget de tip Card asupra căruia am adăugat un wrapper de tip InkWell pentru a putea utiliza acțiunea onTap() pe acestea.

```

class MenuOptionWidget extends StatelessWidget {
    final MenuOption option;
    final Function function;

    MenuOptionWidget({this.option, this.function});

    @override
    Widget build(BuildContext context) {
        return Card(
            margin: EdgeInsets.all(15.0),
            child: InkWell(
                splashColor: Swatches.myPrimaryMint,
                onTap: () => function.call(),
                child: Center(
                    child: Padding(
                        padding: const EdgeInsets.all(30.0),
                        child: Column(
                            children: <Widget>[
                                Icon(
                                    option.iconData,
                                    color: Swatches.green1,
                                    size: 60.0),
                                FittedBox(fit: BoxFit.fitHeight,
                                    child: Text(

```

```

        option.optionText,
        softWrap: true,
        overflow: TextOverflow.ellipsis,maxLines: 2))
    ],
    textDirection: TextDirection.ltr,
),
),
),
),
),
);
}
}

```

Dacă majoritatea widget-urilor Stateful au o implementare în care se utilizează şablonul vanilla (când creăm un widget Stateful se creează automat și o clasă care extinde starea acestuia), în cazul nostru dorind să avem logica separată de UI, folosim şablonul mixed, adică pe lângă clasa Dashboard și DashboardState, mai creăm clasa DashboardView (Stateless widget, reconstruit pe baza stării), care conține toate elementele de UI, iar executarea unor anumitor acțiuni se va face prin intermediul unei instanțe DashboardState, în care se află logica necesară.

Opțiunile din Dashboard sunt diferite în funcție de rolul utilizatorului. Astfel, atunci când se creează meniul pentru un tip de utilizator, se mapează valorile opțiunilor care corespund rolului acestuia.

În exemplul următor exemplificăm crearea GridView-ului și de asemenea folosirea instanței stării pentru a apela o funcție ce (re)construiește elementele în noua stare.

```

Widget _gridView(UserData userData) {
return Flexible(
  child: Padding(
    padding: const EdgeInsets.all(20.0),
    child: GridView(
      gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
        crossAxisCount: 2, crossAxisSpacing: 15),
      children: \textbf{state._mapOptions(userData)}
      , ),
    ),
  );
}

```

MyProfile

Componenta de body a Scaffold-ului din acest ecran conține un widget de tip Column în care se adaugă elemente de tip Text cu diferite stiluri/decorații. De asemenea, atunci când un doctor accesează profilul unui pacient de-al său, acesta vizualizează și un widget de tip Row, în care se află butoane de tip FlatButton, acestea având în funcția onPressed() diferite opțiuni. Prin intermediul pachetului url_launcher se pot lansa diferite url-uri. Astfel că, am implementat funcții pentru inițierea unui apel telefonic, trimitera unui mail, trimitera de mesaje sms, cât și a mesajelor prin intermediul aplicației WhatsApp,

EditForm

În acest ecran s-a folosit, la fel ca în ecranele de SignIn si Register, un widget de tip Form, pentru a prelua și procesa cererea de modificare a profilului unui utilizator.

Appointments

Componenta body a acestui este alcătuită dintr-un buton de tipul Dropdown, personalizat pentru cerințele necesare. Acesta preia datele doctorilor dintr-un stream și le mapează ca elemente de tip DropdownMenuItem. Când un element din dropdown este selectat, starea este schimbată și astfel se va reconstrui body-ul, de data aceasta cu un widget de tipul TableCalendar, implementat personalizat și denumit AppointmentsCalendar), care este populat cu sloturile disponibile ale doctorului selectat. Prin intermediul funcției onSelectedDay(), atunci când vom selecta o zi în care există înregistrări, se va afișa o listă de tip ListView cu intervalele orare disponibile. La apăsarea unui interval, în funcția onTap() se invocă navigatorul și se face o redirecționare a ecranului către widget-ul AppointmentsForm.

AppointmentsForm

În acest ecran s-a folosit un widget de tip FormBuilder, în momentul în care suntem redirecționați din Appointments, câmpurile sunt completate cu datele selectate anterior. Singurul câmp care mai trebuie completat este de tipul FormBuilderTextField. Prin apăsarea butonului de tip RaisedButton se va apela funcția pentru verificare și validare a câmpurilor formularului și mai apoi acestea vor fi trimise spre procesare.

AppointmentsCalendarEvents

În componența body a acestui ecran am reutilizat widget-ul mai sus menționat, AppointmentsCalendar, populat tot prin stream, de această dată doar cu programările utilizatorului curent, în cazul pacienților, sau, în cazul doctorilor, cu toate programările pacienților săi. Prin intermediul funcției onSelectedDay(), atunci când vom selecta o zi în care există programări, acestea se vor afișa într-un ListView. Aceste elemente sunt de tip ListTile pe care le-am personalizat prin intermediul unui Container și a unui widget de tip Column, astfel putând adăuga și un rând de butoane.

ClientList

În acest ecran, bara de sus folosește un SearchDelegate pentru a putea construi cele trei stări ale căutării:

```
@override  
Widget buildLeading(BuildContext context) {  
    return IconButton(  
        icon: Icon(Icons.arrow_back),  
        onPressed: () {  
            close(context, null);  
        },  
    );  
}  
  
@override  
Widget buildResults(BuildContext context) {  
    return Container(...);  
}  
  
@override  
Widget buildSuggestions(BuildContext context) {  
    return StreamBuilder<List<UserData>>(...);}
```

În prima stare interogarea este goală, astfel că se afișează lista tuturor pacienților. Pe măsură ce adăugăm litere, lista se updatează cu o listă de sugestii, iar când executăm comanda de căutare ni se va returna lista cu toate rezultatele găsite.

3.4 Evaluare

Soluția implementată satisfacă toate cerințele funcționale definite în secțiunea 1.3.1. Această evaluare a cerințelor nefuncționale are în vedere nevoile principalelor părți interesate ale sistemului: pacienți și medici.

Atât pacientul, cât și medicul reprezintă utilizatorii finali, dar motivatiile lor sunt diferite, în special dacă un medic deține o clinică dentară. Pentru a putea reduce din acest fapt, realizăm evaluarea în două prisme diferite: cât de mult este utilizată aplicația noastră și care sunt costurile aplicației.

Un utilizator final este interesat de o aplicație care poate fi adaptată pentru nevoile sale, care este ușor de utilizat și are cerințe mici de depozitare. De fapt, dimensiunea aplicației este foarte importantă, deoarece cu cât o aplicație este mai mare, cu atât necesită mai mult spațiu pe un dispozitiv. Cu cât este mai mare aplicația, cu atât va dura mai mult pentru a o descărca. Astfel, ne concentrăm pe depozitare și lăsăm testarea cu utilizatorii pentru munca viitoare.

Pe de altă parte, este de dorit ca medicii și administratorii clinicii să știe de cât spațiu de memorie este nevoie pentru a salva programările, utilizatorii, medicii, intervalele de timp și.a. În timp ce partea de back-end a aplicației utilizează Firebase, utilizarea lunată are un cost variabil (deși există un plan gratuit care acoperă scopurile de testare).

Dorim mai apoi să studiem care este asocierea între scalabilitatea aplicației (în termeni de utilizatori /medici) și costuri.

Dimensiunea aplicației

Pentru a testa dimensiunea aplicației, am compilat implementarea finală, rulând comanda:

```
flutter build apk --target-platform=android-arm64
```

Am obținut următoarele rezultate:

```
Built build/app/outputs/apk/release/app-armeabi-v7a-release.apk (7.7MB)  
și
```

```
Built build/app/outputs/apk/release/app-arm64-v8a-release.apk (8.0MB)
```

Acest lucru ne oferă dimensiunea aplicației compilate (APK) pentru telefoanele mobile pe Android pe 32 și 64 biți, respectiv 7,7 și 8,0 MB. Aceasta permite oricărui utilizator cu o conectivitate standard la internet de 1 MB/sec să descarce aplicația Toothly în aproximativ opt secunde. Instalarea unei astfel de aplicație este de asemenea rapidă.

Astfel, putem concluziona că dimensiunea aplicației satisfacă cerința nefuncțională 1 (dimensiune mică), prezentată în secțiunea 1.3.2.

Costurile de menținere a aplicației

Vom studia costurile de întreținere în diferite scenarii, în situația în care clinicele alocă 1% din veniturile lor pentru cheltuieli aferente aplicației.

Scenariul 1:

Clinică de mici dimensiuni (cu 2 medici), fiecare cu câte 8 pacienți noi pe zi. Fiecare consultație costă în medie 100 lei. Clinica produce $2*8*5*4*100 = 32.000$ lei lunar. Bugetul alocat lunar aplicației este de 320 lei.

Scenariul 2:

Clinica de dimensiuni medii (cu 10 medici), fiecare cu câte 8 pacienți noi pe zi. Fiecare consultație costă 100 lei. Clinica produce $10*8*5*4*100 = 160.000$ lei pe lună. Bugetul alocat lunar aplicației este de 1.600 lei.

Scenariul 3:

Clinică de dimensiuni mari (cu 25 de medici), fiecare cu câte 8 pacienți noi pe zi. Fiecare consultație costă 100 lei. Clinica produce $50*8*5*4*100 = 400.000$ RON pe lună. Bugetul alocat lunar aplicației este de 4.000 lei.

Pentru a calcula costurile asociate fiecărui scenariu, am folosit un calculator de prețuri de la Firebase [19]. Principalii factori care influențează prețul aplicației sunt: numărul de GB stocați pe Cloud Firestore, citirile și scrierile de documente, numărul de invocări ale Cloud Functions. Trebuie menționat faptul că estimarea este una pesimistă (valorile reale sunt în cel mai rău scenariu la fel de mare ca cel arătat aici).

În ceea ce privește stocarea există cinci colecții: users, tokens, timeslots, appointments și doctors_pacients. Este posibilă calcularea medie de stocare necesară într-o perioadă de o lună¹. Dimensiunile sirurilor sunt calculate ca număr de UTF-8 octeți codați + 1. Dimensiunea unui nume de document este dată de dimensiunea ID-ului colecției și dimensiunea ID-ului documentului din calea către document, plus 16 octeți suplimentari. Dimensiunea unui ID de document este fie dimensiunea sirului pentru un ID de tip string sau 8 octeți pentru un ID de tip int.

Spațiul de stocare ocupat (exprimat în octeți) este:

- users collection (6). Document ID: 8; age (4 + 8); details (8 + 20, în medie); email (6 + 10), firstName (10 + 8); gender (7 + 8); hasAllergies (13 + 1); phoneNumber (12); surname (8 + 10, în medie), role (5 + 2); uid (4 + 8)
- tokens collection (7): token (6 + 33);
- appointment (12): clientId (9 + 8); clientName (10 + 10, în medie); date (5 + 8), details (8 + 20, în medie); doctorId (8 + 8); doctorName (11 + 10, în medie); notes (5 + 100, în medie); status (6 + 8);

¹<https://firebase.google.com/docs/firestore/storage-size>

- doctors_patients (16): patientNumber x 8;
- timeslots (10): date (5 + 8); doctorId (8 + 8); timeslots (9 + 8(13+1))

Spațiul de stocare solicitat de indecșii bazei de date poate fi dat la o parte, deoarece este foarte scăzut (<1Kbyte).

Dimensiunea unui document din colecția users este 166 de octeți; un document din colecția tokens are 46 de octeți; un document din colecția appointments are 246 de octeți; un document din colecția timeslots are 160 de octeți; un document din colecția doctors_patients are dimensiune variabilă, în funcție de numărul de pacienți/medici (16 octeți + numărul de pacienți x 8 octeți).

Spațiul de stocare (în octeți) necesar, pe lună, este dat de relația:

$$\begin{aligned} bytes = & pacieni * 166 + doctori * 166 + tokens * 160 + \\ & appointments * 246 + 16 * doctori * \left(\frac{pacieni}{doctori} * 8 \right) + \\ & timeslots * 46 \end{aligned} \quad (3.1)$$

Într-o lună este necesară crearea a cel mult 320 de intervale de timp, 320 de pacienți, 640 token-uri de mesagerie Firebase Cloud pentru acei pacienți (presupunând că fiecare pacient accesează Toothly de pe 2 dispozitive diferite), 320 de întâlniri, 320 doctors_patients. $320 / D^8$ este numărul de pacienți pe medic, presupunând că pacienții sunt repartizați în mod egal între medici.

Spațiul total de stocare necesar într-o lună (exprimat în octeți) este de:

$$\begin{aligned} 320 * 166 + 2 * 166 + 640 * 160 + 320 * 246 + 16 * 2(320/2)^8 + 320 * 46 = 290252 \\ \text{bytes} = 0.27\text{MB}, 320 * 166 + 10 * 166 + 640 * 160 + 320 * 246 + 16 * 10(320/10)^8 \\ + 320 * 46 = 291580 = 0.28\text{MB} și 320 * 166 + 25 * 166 + 640 * 160 + 320 * 246 + \\ 16 * 25(320/25)^8 + 320 * 46 = 294070 = 0.28\text{MB}. \end{aligned}$$

Întrucât toate valorile sunt mai mici decât cota gratuită (1024 MB), aceasta nu implică niciun cost.

În ceea ce privește citirile din baza de date, numărul de acestea pe pacient este dat de relația:

$$CitiriP_i = T * D \quad (3.2)$$

unde T = numărul de intervale de timp, D = numărul de medici.

Cum sunt în jur de 40 zile lucrătoare în două luni, numărul de intervale de timp este de $8 * 40 = 320$ de intervale de timp.

Numărul de citiri pe pacient, pe două luni, este dat de $320 * 2$, $320 * 10$ și $320 * 25$ (pentru scenariile 1,2 și 3). Considerăm că toate intervalele de timp sunt umplute și, astfel, există un pacient nou pe zi care verifică toate consultațiile. În realitate, intervalele de timp trecute nu sunt citite, dar le includem ca

parte a celei mai grave cazuri. Numărul total de pacienți este același cu numărul de intervale de timp, 320. Numărul total de citiri efectuate de clienții sunt $320*320*2 = 204.800$, $320*320*10 = 1.024.000$ și $320*320*25 = 2.560.000$, în scenariile 1, 2 și 3.

Citirile efectuate de medici corespund verificării profilurilor lor de pacienți și istoricul consultărilor. Dacă presupunem că medicul preia informații de la 8 pacienți în prima zi, de la 16 pacienți în ziua 2, de la 24 de pacienți în ziua a 3-a și aşa mai departe, numărul de lecturi din ziua n este dat de relația:

$$\sum_{1}^{20} 8n = 8(1 + 2 + \dots + n) = 4n(n + 1). \quad (3.3)$$

$n = 20$ (1 month), medicii efectuează 1680 de citiri pe lună. La citirile respective, ar trebui adăugat numărul de lecturi date consultând fiecare istoricul pacientului o dată (320 citiri). Numărul de citiri este de aproximativ 2000. Cum numărul gratuit de citiri pe Firebase este de 1.500.000, citirile de la medici pot fi neglijate.

Rezumând, în scenariile 1 și 2 nu este necesar să plătiți nimic (deoarece numărul de citiri este sub numărul de citiri gratuite). Pentru scenariul 3, costurile asociate cu citirile ar fi 149 USD (650 lei).

Numărul de scrieri cuprinde înregistrarea pacienților, înregistrarea intervale de timp și adăugarea de note de la consultări (de două ori, presupunând medicul lasă note înainte și după programare). Numărul total de scrieri ar fi dat de formula:

$$Total\ citiri = (P + T + notes) * D \quad (3.4)$$

, unde P este numărul de pacienți, T este numărul de intervale de timp, notes este numărul total de note create și D numărul de medici. Totalul numărul de scrieri este dat de relațiile $(320 + 320 + 640) * 2 = 2.560$, $(320 + 320 + 640) * 10 = 12.800$, respectiv $(320 + 320 + 640) * 25 = 32.000$. Deoarece rezultatele sunt inferioare cotei (600.000), scrierea nu include un cost.

Invocările funcțiilor cloud se fac când un utilizator este înregistrat, când o notă este schimbată și când se schimbă starea unei consultări. Presupunând că notele unei consultări sunt schimbate de două ori ($320 + 320$), starea unei programări este schimbată de două ori ($320 + 320$) și sunt înregistrări 345 de utilizatori (320 pacienți + 25 medici), în cel mai rău caz, există $320 \times 5 + 25$ apeleuri = 1.625 apeluri la Cloud Functions.

Deoarece numărul invocărilor gratuite este de 2.000.000, back-end-ul funcțiilor Cloud nu suportă niciun cost. În scenariile 1 și 2, costul pentru aplicație este 0 lei, deoarece cererea aplicației nu depășește cota gratuită. În scenariul 3, costurile estimate sunt de 149 USD (640 lei), deoarece există o multime

de lecturi emise de sistem. Traficul în termeni de citiri și modificări, precum și apelurile către funcțiile Cloud nu sunt semnificative, și astfel nu se suportă niciun cost. Spațiul de stocare necesar este practic 0, fiind necesari cca. 0,3 MB din cota de 1024 MB disponibile. Pentru scenariul 3, bugetul alocat pentru aplicație este 4.000 RON, ceea ce este aproximativ echivalent cu 920 \$. Prin urmare, sunt utilizati numai aproximativ 16% din bugetul disponibil. Cu toate acestea, această evaluare nu a luat în considerare costurile asociate cu cele-lalte dimensiuni ale dezvoltării aplicației: dezvoltarea în sine, asistență pentru clienti, marketing, licențe legale, software etc. Putem concluziona că dimensiunea aplicației satisface cerința nefuncțională 2, prezentate în secțiunea 1.3.2 .

Alte cerințe nefuncționale

Avantajul utilizării Firebase constă în faptul că serviciile sunt oferite pentru a reduce complexitatea dezvoltării și pentru a crește numărul de funcționalități implementate. Prin utilizarea Firebase sunt satisfăcute unele cerințe nefuncționale, astfel:

1. Funcția de scalare automată a Firebase satisface cerințele nefuncționale 2 (latență scăzută), 4 (disponibilitate) și 6 (scalabilitate).
2. Efectuarea implementărilor asigură cerința nefuncțională 5 (confidențialitate), deoarece medicii pot accesa doar datele pacientilor lor.
3. Instrumentele de testare din Firebase, cum ar fi DevTools sau Firebase Cli (prin emulatoare Firebase offline), permit definirea și rularea testelor pe aplicația dezvoltată. Aceasta îndeplinește cerința nefuncțională 7 (testabilitate).

Capitolul 4

Scenarii de utilizare

Un scenariu de utilizare este o listă de acțiuni/pași care definesc interacția dintre un rol (actorul principal al scenariului) și un sistem pentru a-și atinge scopul propus.

În acest capitol voi prezenta câteva scenarii de utilizare ale aplicației, însotite de capturi de ecran pentru a putea vizualiza și pașii făcuți de un anumit tip de utilizator cu scopul de a atinge un obiectiv.

În scenariile următoare ne vom referi la utilizatorul cu rol de pacient **Alice**, iar la utilizatorul cu rol de doctor **Cristina Coman**.

Alice este student intern la Universitatea din Ploiești. Având de făcut niște intervenții stomatologice, ea a accesat cabinetele medicale din proximitate. Primul site listat pe motorul de căutare a fost cel al cabinetului Ogodent. Pe pagina principală a site-ului există un link către aplicația mobilă a cabinetului (Toothly). Aceasta a descărcat aplicația și pentru a-i putea accesa funcționalitățile, Alice va trebui să-și creeze un cont. La autentificare a folosit adresa de email și o parola, ca mai apoi să completeze un formular cu datele personale.

4.1 Scenarii pacienți

În această secțiune voi prezenta scenariile de utilizare din perspectiva unui pacient.

4.1.1 Scenariul 1 (Rezervare consultație)

Alice dorește să rezerve o consultăție. Aceasta deschide aplicația, se loghează cu credențialele și mai apoi alege din meniul principal opțiunea Calendar, fiind redirecționată către ecranul în care se poate alege slotul disponibil

pentru o consultăție. În acest ecran, după ce ea alege doctorul la care vrea să-și rezerve programarea, se afișează un calendar, iar zilele în care există sloturi disponibile sunt semnalizate. Aceasta alege o zi. O listă de intervale orare este afișată și Alice selectează intervalul orar convenabil pentru ea. După alegerea intervalului, este redirectionată spre formularul de trimisere a cererii, în care specifică problema pe care aceasta o are și apoi apasă pe butonul Trimite. După ce a fost trimisă cererea aceasta primește o notificare pentru trimiterea cererii cu succes.

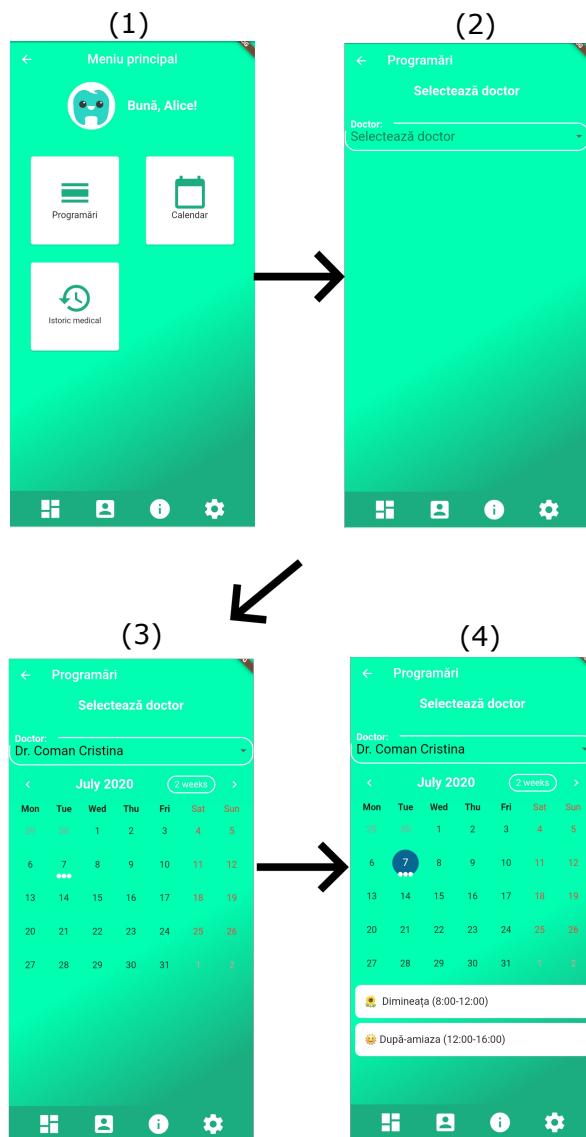


Figura 4.1: Rezervare consultăție

4.1.2 Scenariul 2 (Informații clinică)

Alice vrea să poată consulta informațiile clinicii. Acesta deschide aplicația, intră în meniul principal și alege din bara de navigare opțiunea Info clinică. Ecranul care apare conține informații generale ale clinicii și o listă de doctori.

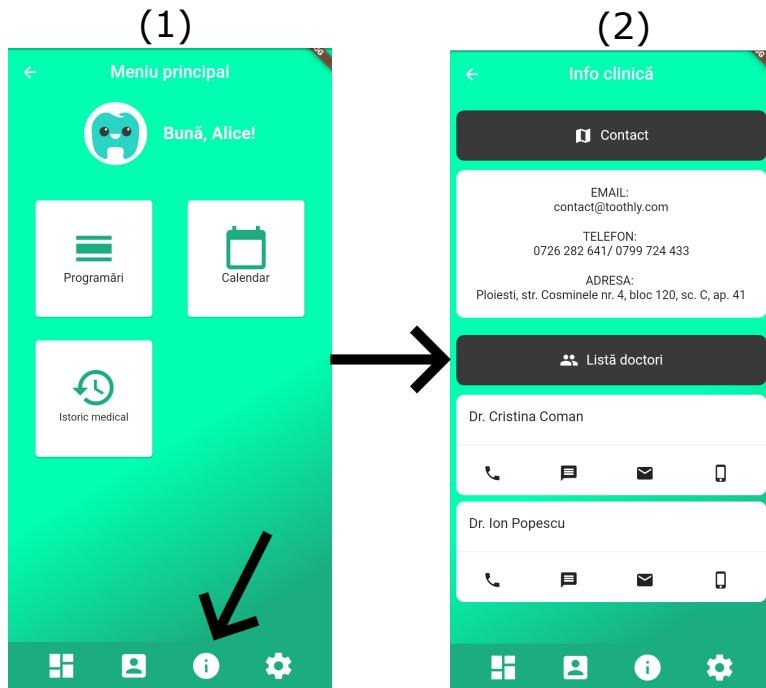


Figura 4.2: Informații clinică

4.1.3 Scenariul 3 (Consultare istoric medical și programări cu-rente)

Alice vrea să vizualizeze programările și statusul acestora. Ea intră în meniul principal al aplicației și are două posibilități:

- Prima posibilitate:

Alege din meniul principal opțiunea Programări. Se încarcă ecranul programărilor, afișându-se un calendar. În acesta sunt evidențiate datele în care Alice are stabilite programările. Când selectează o dată, i se afișează o listă cu programările din ziua respectiva, putând vedea detaliile acesteia și de asemenea notițele făcute de doctor cu informațiile preliminare.

- A doua posibilitate:

Alege din meniul principal opțiunea Istoric medical. Se încarcă istoricul consultărilor. Alice poate consulta atât lista programărilor viitoare, cât și lista consultărilor trecute cu detalii corespunzătoare acestora.

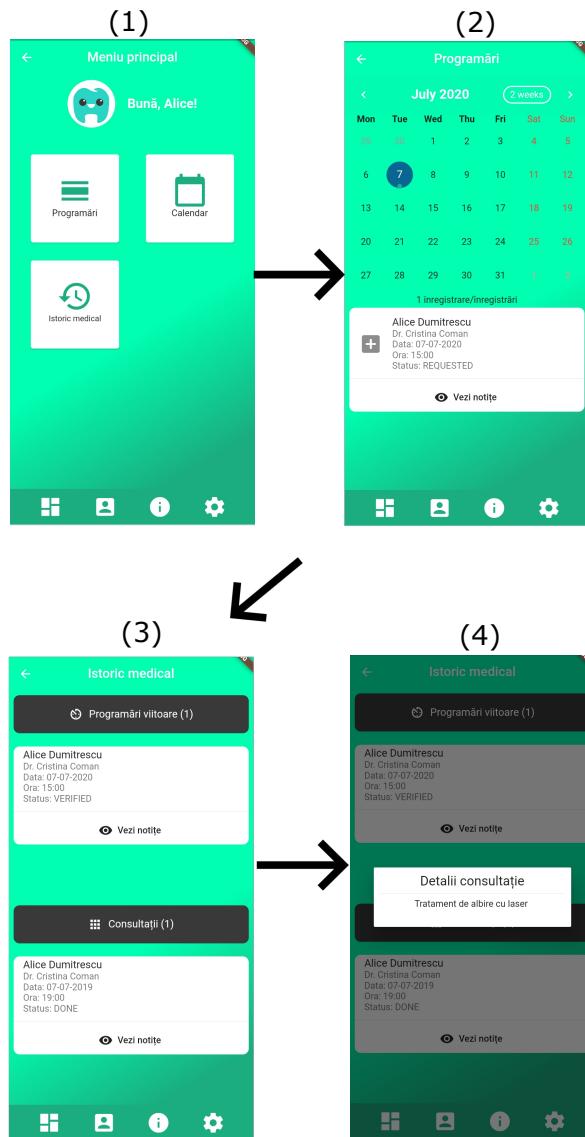


Figura 4.3: Consultare istoric medical și programări curente

4.2 Scenarii doctor

În această secțiune voi prezenta scenariile de utilizare din perspectiva unui doctor.

4.2.1 Scenariul 1 (Confirmarea/refuzarea cererilor pentru consultații)

Dr. Cristina Coman deschide meniul principal al aplicației și alege din bara de navigare opțiunea Notificări. Ecranul care apare conține o listă cu cererile de consultații de la pacienți. Acesta are pentru fiecare cerere opțiunea de a o confirma sau a o refuza. Apăsând pe butonul de confirmare aceasta acceptă clientul ca fiind pacientul său. În cazul butonului de refuzare, se va trimite automat pacientului o notificare.

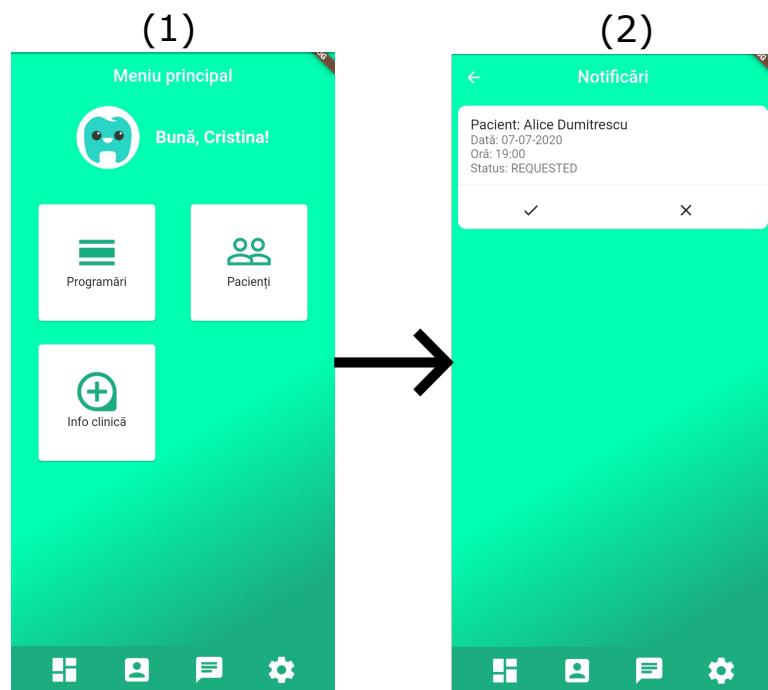


Figura 4.4: Confirmarea/refuzarea cererilor pentru consultații

4.2.2 Scenariul 2 (Furnizarea informațiilor preliminare)

Dr. Cristina Coman dorește să notifice pacientul înainte ca acesta să vină la consultatie, informându-l asupra unor detalii preliminare (spre exemplu: "Să nu

bea cafea în ziua consultației pentru că anestezicul își va face efectul mai greu” sau „Să realizeze o radiografie în prealabil și să o aducă la cabinet”).

Doctorul selectează din meniul principal opțiunea Programări. Ecranul care apare conține un calendar. Doctorul selectează consultația căreia dorește să-i atribuie informațiile apăsând pe butonul de Editare notiță. Apare o fereastră de dialog cu un câmp de introducere a textului în care scrie informația și apoi apasă pe butonul Editează. Dacă modificarea a fost făcută cu succes, atât ea, cât și clientul primesc o notificare.

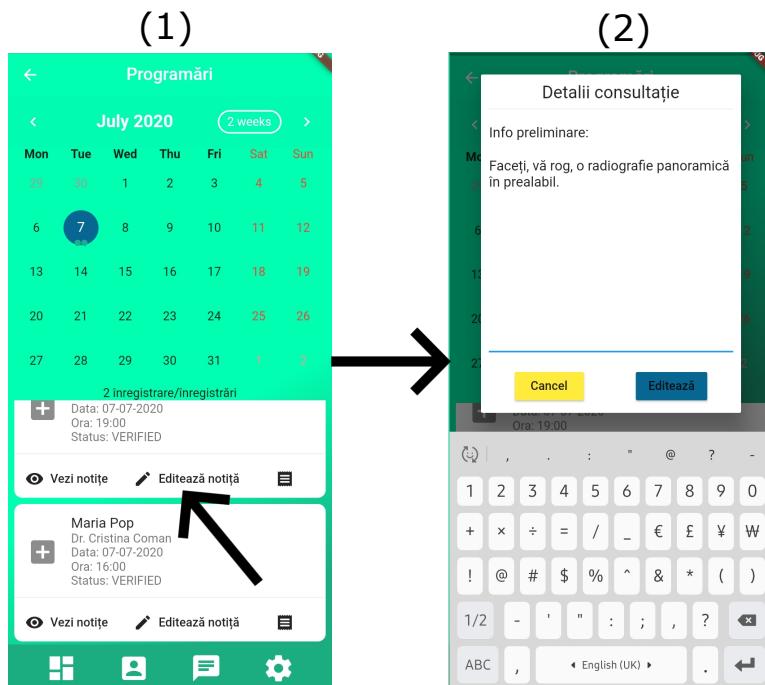


Figura 4.5: Furnizarea informațiilor preliminare

4.2.3 Scenariul 3 (Modificarea statusului unei consultații)

Dr. Cristina Coman vrea să schimbe statusul unei consultații. Aceasta alege din meniul principal opțiunea de Programări. Ecranul care apare conține un calendar al programărilor, aceasta selectează programarea căreia dorește să-i modifice statusul și apasă pe butonul de editare a statusului. Apare o fereastră de dialog care conține o listă a statusurilor din care poate alege. În momentul în care doctorul alege starea și apasă pe butonul Editează, dacă modificarea s-a făcut cu succes, atât ea, cât și clientul primesc o notificare.

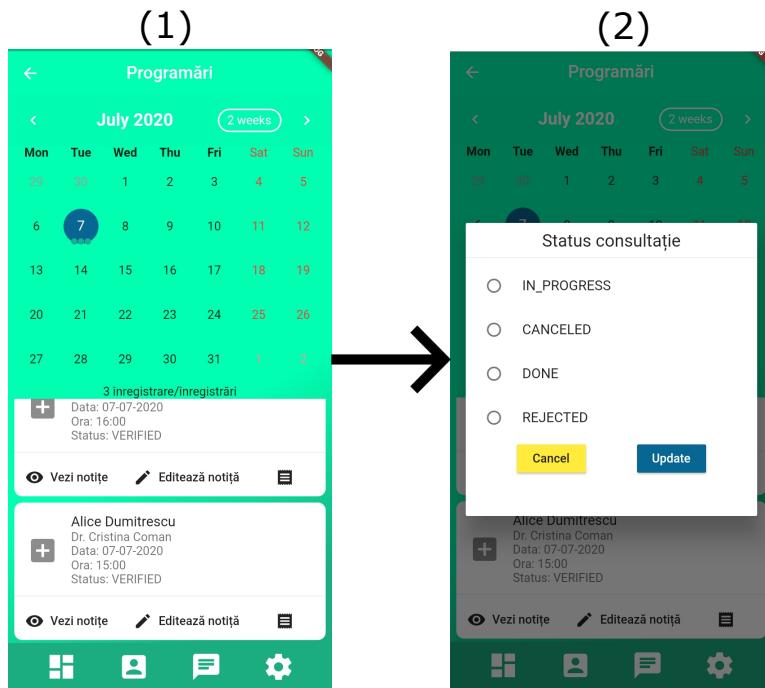


Figura 4.6: Modificarea statusului unei consultații

4.2.4 Scenariul 4 (Accesarea istoricului consultațiilor unui pacient)

Dr. Cristina Coman dorește să vizualizeze lista pacienților săi și să le poată vedea profilul și istoricul consultațiilor, cât și programările viitoare. Aceasta selectează din meniul principal opțiunea Pacienți. În ecranul care apare este vizibilă o listă a pacienților și doctorul caută după nume, prin intermediul barei de căutare, pacientul căruia dorește să-i vadă informațiile. Pentru accesarea informațiilor există două opțiuni: vizualizarea profilului pacientului sau vizualizarea istoricului consultațiilor. Doctorul selectează a doua opțiune și astfel vede istoricul consultațiilor.

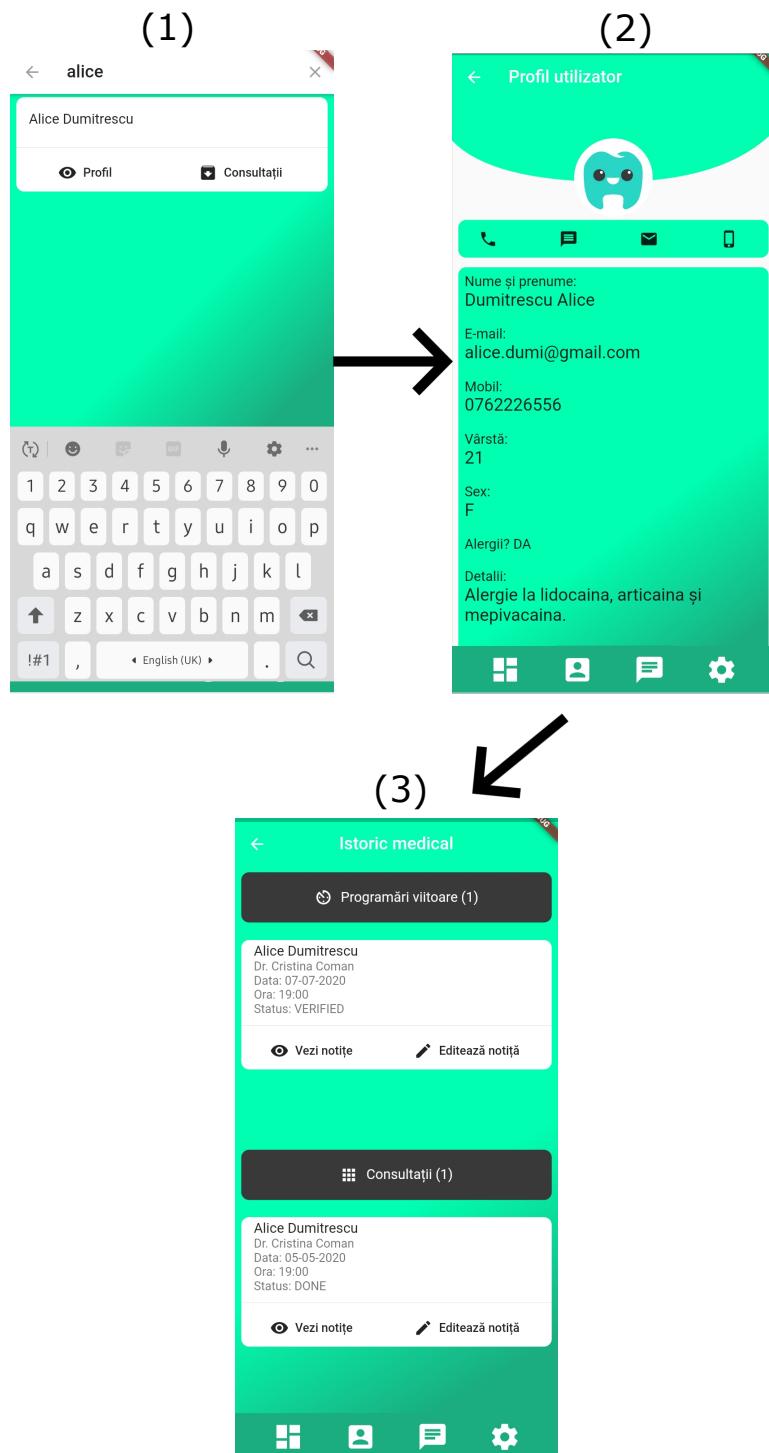


Figura 4.7: Accesarea istoricului consultărilor unui pacient

4.2.5 Scenariul 5 (Salvarea detaliilor consultației)

Dr. Cristina Coman vrea să atribuie consultației unui pacient detalii despre procedurile pe care aceasta îi le-a aplicat pacientului. Aceasta deschide meniul principal și alege opțiunea de Programări. Selectează consultația pe care dorește să o modifice. Își schimbă statusul în Terminată și apoi apasă pe butonul de editare a notișelor. În fereastra de dialog care are un câmp de introducere de text, doctorul adaugă informațiile despre proceduri aplicate și apasă pe butonul Editează. Dacă modificarea s-a făcut cu succes, atât ea, cât și clientul primesc o notificare.

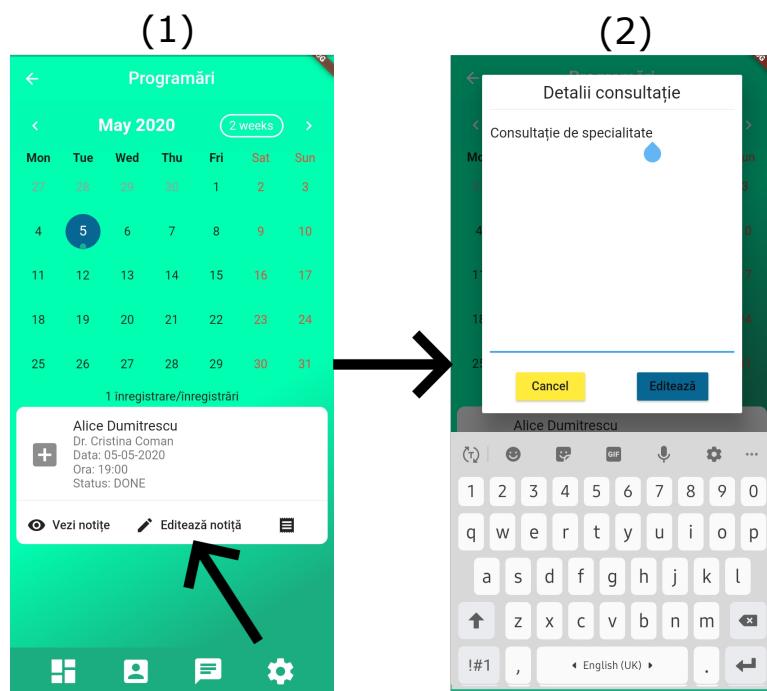


Figura 4.8: Salvarea detaliilor consultației

Capitolul 5

Concluziile lucrării și munca viitoare

5.1 Concluzii

Realitatea ne arată că majoritatea oamenilor își împart viața între serviciu, familie și ... gadget-uri, cele mai la îndemână fiind telefonul mobil sau tableta (în detrimentul „demodatului” desktop). Putem concluziona că omul este din ce în ce mai dependent de aplicațiile mobile. De aici și necesitatea de a dezvolta aplicații mobile pentru absolut orice (auzi din ce în ce mai des în jur „există o aplicație pentru asta?”).

Nici domeniul Stomatologiei nu duce lipsă de asemenea aplicații. Motivele pentru care m-am gândit să dezvolt o asemenea aplicație personalizată și nu una generală nu sunt de neglijat:

- exclusivitatea ei (nu depinde de alții, de update-uri, de oprirea aplicației);
- flexibilitatea ei (o poți personaliza după nevoile proprii, dar și ale clientilor);
- siguranța ei (o aplicație mobilă personalizată este mai eficientă în fața amenințărilor digitale);
- este întotdeauna în vizorul clientilor, fiind listată cu numele său, nu generic;
- costurile de întreținere sunt net inferioare celei generale.

Este dificil să găsești o aplicație bună care să satisfacă atât nevoile pacienților, dar și a medicilor. Pacienții au probleme în comunicarea cu medicii și își doresc să obțină consultația necesară doar în câteva click-uri și foarte rapid. Medicii nu manifestă flexibilitate în transmiterea cu ușurință a instrucțiunilor către pacienții lor.

Pentru a rezolva aceste lipsuri, am dezvoltat o aplicație mobilă numită Toothly. Aplicația satisfacă cu succes atât cerințele funcționale, cât și nefunctoriale de care are nevoie o clinică de dimensiuni mici (2 medici), medii (10 medici) sau mari (25 medici) pentru operațiunile sale. În mod special, în cuprinsul aces-

tei lucrări am demonstrat că aplicația noastră ocupă doar 8 MB și ar necesita în jur de 149 de dolari (640 lei) pe lună pentru menenanță (pentru o clinică de dimensiuni mari, cost ce reprezintă o mică parte din venituri - sub 1% din acestea).

Pe scurt, Toothly este o aplicație care poate îmbunătăți productivitatea în clinici, oferind avantaje mari pentru utilizatorii ei, la un cost foarte mic.

5.2 Muncă viitoare

Ca orice lucru din lumea noastră materială, aplicația mobilă poate fi îmbunătășită sau extinsă, capacitatea de dezvoltare fiind nelimitată, soluțiile propuse de dezvoltator/utilizator putând face obiectul unor analize în detaliu folosind metode de evaluare moderne: evaluarea euristică (de exemplu: euristică Jakob Nielsen [20]), evaluarea predictivă suplimentară folosind GOMS model [21] - lucru pe care l-am făcut, parțial, prezicând costurile asociate cu aplicația), dar și ca urmare unor evaluări ale utilizatorilor.

Bibliografie

- [1] Google, "Dart programming language | Dart," 2020. [Online]. Available: <https://dart.dev/>
- [2] "DevTools - Flutter," 2020. [Online]. Available: <https://flutter.dev/docs/development/tools/devtools>
- [3] Google, "Using the Flutter inspector - Flutter," 2020. [Online]. Available: <https://flutter.dev/docs/development/tools/devtools/inspector>
- [4] "Widget catalog - Flutter," 2020. [Online]. Available: <https://flutter.dev/docs/development/ui/widgets>
- [5] L. Moroney, *The Definitive Guide to Firebase*. Apress, 2017.
- [6] Google, "Firebase homepage," 2020. [Online]. Available: <https://firebase.google.com/>
- [7] "Firebase features," 2020. [Online]. Available: <https://firebase.google.com/>
- [8] Datanyze, "Firebase vs Heroku Competitor Report | Platforms as a Service," 2020. [Online]. Available: <https://www.datanyze.com/market-share/paas--445/firebase-vs-heroku>
- [9] Google, "Firebase Authentication," 2020. [Online]. Available: <https://firebase.google.com/docs/auth/>
- [10] OpenID Team, "OpenID Connect | OpenID," 2014. [Online]. Available: <https://openid.net/connect/>
- [11] OAuth Team, "OAuth 2.0 Homepage," 2020. [Online]. Available: <https://oauth.net/2/>
- [12] OAuthTeam, "RFC 6749 - The OAuth 2.0 Authorization Framework," 2012. [Online]. Available: <https://tools.ietf.org/html/rfc6749>

- [13] M. Jones, "JSON Web Algorithms (JWA)," Tech. Rep., may 2015. [Online]. Available: <https://www.rfc-editor.org/info/rfc7518>
- [14] JSON Team, "JSON Web Tokens - jwt.io." [Online]. Available: <https://jwt.io/>
- [15] H. Varshney, A. S. Allahlooh, and M. Sarfraz, "IoT Based eHealth Management System Using Arduino and Google Cloud Firestore," in Proceedings - 2019 International Conference on Electrical, Electronics and Computer Engineering, UPCON 2019. Institute of Electrical and Electronics Engineers Inc., 2019.
- [16] Google, "Firebase Cloud Messaging," p. 2020. [Online]. Available: <https://firebase.google.com/docs/cloud-messaging>
- [17] "Cloud Functions for Firebase," 2020. [Online]. Available: <https://firebase.google.com/docs/functions/>
- [18] R. Franks, "Electricity Monitoring App - Send Push Notifications using Firebase Cloud Functions," 2017. [Online]. Available: <https://rigaroo.dev/electricity-monitor-notifications-firebase-cloud-functions/>
- [19] Google, "Firebase Pricing," 2020. [Online]. Available: <https://firebase.google.com/pricing/#blaze-calculator>
- [20] J. Nielsen, "Heuristic Evaluation: How-To: Article by Jakob Nielsen," 1994. [Online]. Available: <https://www.nngroup.com/articles/how-to-conduct-a-heuristic-evaluation/>
- [21] J. Bonnie and D. Kieras, "GOMS Model Work," 1996. [Online]. Available: [https://web.eecs.umich.edu/\\$\sim\\\$kieras/goms.html](https://web.eecs.umich.edu/$\sim\$kieras/goms.html)