**UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI**

**FACULTATEA DE INFORMATICĂ**



LUCRARE DE LICENȚĂ

**HobbyMeet**

propusă de

***Ilie-Tudor Savu***

**Sesiunea**: *Iulie, 2020*

Coordonator științific

***Colab. Florin Olariu***

**UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI**

**FACULTATEA DE INFORMATICĂ**

**HobbyMeet**

***Ilie-Tudor Savu***

**Sesiunea**: *Iulie, 2020*

Coordonator științific

***Colab. Florin Olariu***

Cuprins

[Introducere 5](#_Toc43507964)

[Motivaţie 5](#_Toc43507965)

[Obiectiv 5](#_Toc43507966)

[Cerinţe funcţionale 6](#_Toc43507967)

[Capitolul I. Tehnologii utilizate 6](#_Toc43507968)

[1.1 React 7](#_Toc43507969)

[1.1.1 JSX 8](#_Toc43507970)

[1.1.2 Componente 8](#_Toc43507971)

[1.1.3 Starea şi ciclul vieţii componentelor 9](#_Toc43507972)

[1.2 Material-ui 10](#_Toc43507973)

[1.3 Firebase 10](#_Toc43507974)

[1.3.1 Autentificare 11](#_Toc43507975)

[1.3.2 Bază de date Firestore 11](#_Toc43507976)

[1.3.3 Sistem de depozitare a fişierelor 11](#_Toc43507977)

[1.4 ASP.NET Core 12](#_Toc43507978)

[1.4.1 Pachete NuGet 12](#_Toc43507979)

[Capitolul II. Arhitectura aplicaţiei 14](#_Toc43507980)

[2.1 Backend Firebase 15](#_Toc43507981)

[2.1.1 Modulul de autentificare 15](#_Toc43507982)

[2.1.2 Baza de date 16](#_Toc43507983)

[2.1.3 Firestore 16](#_Toc43507984)

[2.2 Backend ASP.NET Core 17](#_Toc43507985)

[2.2.1 Structura generală 17](#_Toc43507986)

[2.2.2 Structura bazei de date 17](#_Toc43507987)

[2.2.3 Controlere 18](#_Toc43507988)

[2.3 Frontend React 19](#_Toc43507989)

[2.3.1 Formularul de înregistrare 20](#_Toc43507990)

[2.3.2 Structura paginii principale 20](#_Toc43507991)

[Capitolul III. Detalii de implementare 23](#_Toc43507992)

[3.1 Web API 23](#_Toc43507993)

[3.1.1 Configurare 23](#_Toc43507994)

[3.1.2 Algoritmul de recomandări 24](#_Toc43507995)

[3.2 Client 28](#_Toc43507996)

[3.2.1 Înregistrare 28](#_Toc43507997)

[3.2.2 Navigație 29](#_Toc43507998)

[3.2.3 Profilul utilizatorului 30](#_Toc43507999)

[3.2.4 Modificarea contului 31](#_Toc43508000)

[3.2.5 Recomandări 32](#_Toc43508001)

[3.2.6 Notificări 33](#_Toc43508002)

[3.2.7 Conversații și lista de prieteni 35](#_Toc43508003)

[Concluzii finale 37](#_Toc43508004)

[Direcții de viitor 37](#_Toc43508005)

# Introducere

Aplicaţiile web sunt programe executate într-un browser web şi implementate folosind tehnologii precum: PHP, ASP, PEARL, PYTHON, HTML, CSS, JAVASCRIPT etc. Popularitatea acestui tip de aplicaţie este în creştere faţă de aplicaţiile clasice, adică cele instalate şi rulate la nivel de sistem de operare, datorita avantajelor oferite, precum independenţa faţă de sistemul de operare, lipsa instalării, scalabilitatea, etc.

Programarea unei aplicaţii web poate fi simplificată prin folosirea framework-urilor sau a librăriilor. Acestea facilitează evoluţia proiectului, datorită faptului că acestea permit dezvoltatorilor să concentreze resurse asupra problemelor specifice aplicaţiei, fără să mai fie nevoie de rezolvarea unor probleme comune, precum management-ul utilizatorilor. Folosirea framework-urilor poate reduce numărul de erori din program prin simplificarea codului. Adesea, aplicaţiile sunt exploatate de hackeri din cauza faptului că în codul acesteia există erori care pot fi abuzate. De asemenea, se promovează utilizarea unor practici bune precum GET după POST.

## Motivaţie

În contextul carantinei, pentru socializare cu persoane noi sau cunoscute, avem ca opţiune aplicaţiile online de social media. Consider că socializarea, în orice formă, este o nevoie de bază pentru oricine, şi cred că majoritatea oamenilor consideră la fel. De asemenea, datorită avansării în tehnologie, putem fi conectaţi pe o platformă de socializare, oriunde am fi în lume, de pe telefon, tabletă sau calculator.

## Obiectiv

Îmi propun realizarea unei aplicaţii care are ca scop aducerea laolaltă persoanelor cu hobby-uri comune. Această aplicaţie este destinată celor care simt nevoia să discute despre pasiunile lor.

## Cerinţe funcţionale

În momentul în care un utilizator va accesa aplicația, își va crea un cont în bazele de date. Contul va fi completat și personalizat în funcție de preferințele fiecărui utilizator.

Principalul scop al aplicației este întâlnirea online a persoanelor cu hobby-uri comune. Așadar, utilizatorii vor putea vizualiza alţi utilizatori puşi la dispoziție de către aplicație în momentul de după autentificare, când utilizatorul va opta să caute alte persoane.

Odată ce se cere o căutare, aplicaţia va furniza o listă care va conţine date despre alţi utilizatori, în funcţie de care se vor dispune mai multe profiluri de utilizatori şi opţiunea de a adăuga ca prieten şi de iterare prin această listă.

După ce utilizatorul va trimite o cerere de prietenie, cel ce primeşte cererea va fi notificat şi va avea opţiunea de a adăuga ca prieten sau de a refuza. Dacă cererea ar fi acceptată, aceştia vor avea pe fiecare adăugaţi în lista de prieteni şi vor putea comunica în timp real.

# Capitolul I. Tehnologii utilizate

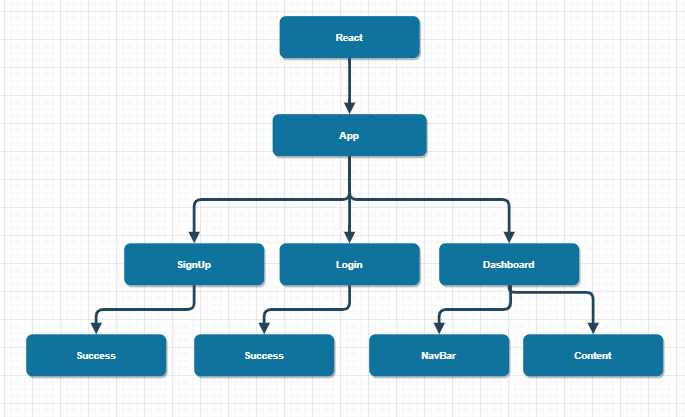
Pentru dezvoltarea aplicaţiei „HobbyMeet” s-au folosit următoarele tehnologii:

* + - React
    - Material-ui
    - Firebase
    - ASP.NET Core
    - Entity framework
    - SQL Server

## React

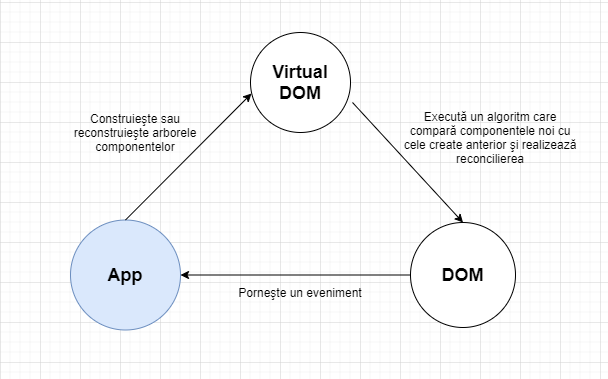
React este o librărie open-source care ajută la construirea interfeţelor de utilizator (UI). Interfeţele au structura arborescentă şi sunt compuse din componente. Un component este un mix de HTML şi JavaScript care compartimentează toată logica necesară pentru dispunerea unei secţiuni dintr-o anumită interfaţă UI. Fiecare dintre aceste componente poate fi construit succesiv în părţi mai complexe ale aplicaţiei, după cum se poate vedea şi în figura 1.1:

Figura 1.1.1 Arborele componentelor



Cu ajutorul acestei librării se pot crea cu uşurinţă aplicaţii SPA ( Single page aplications) care furnizează o experienţă a utilizatorului mai flexibilă, ascunzând complexitatea. Acest tip de aplicaţii sunt dinamice, deoarece permit reactualizarea conţinutului aproape instantaneu. De asemenea, utilizatori pot naviga pe multiple pagini fără să fie nevoie de reîncărcarea efectivă a paginii. Avantajul aplicaţiilor care folosesc React se realizează prin procesul din figura 1.2:

Figura 1.1.2 Relaţia aplicaţiei cu DOM



DOM-ul ( Document Object Model) virtual este un concept de programare în care se menţine o reprezentare virtuală a UI-lui şi care este sincronizat cu DOM-ul real de o librărie precum ReactDOM. Acest proces se numeşte reconciliere.

### 1.1.1 JSX

JSX ( JavaScript XML) este o extensie a sintaxei JavaScript folosită de obicei împreună cu React pentru a descrie un UI. De exemplu, această expresie foloseşte JSX:

|  |
| --- |
| const element = <h1>Hello, world!</h1>; |

Acest tip de sintaxă înlocuieşte funcţia createElement(), dar şi appendChild(), întrucât JSX poate conţine elemente JSX în interiorul altor elemente JSX:

|  |
| --- |
| const element = <div>  <h1>Hello World!</h1>  <h1>Hello World!</h1>  </div>; |

JSX necesită încapsularea celorlalte elemente într-un singur tag HTML.

Acesta este compilat de *Babel*, care transformă JSX în sintaxă JavaScript, pentru a putea fi interpretat de browser.

### 1.1.2 Componente

Componentele ajută la împărţirea UI-ului în bucăţi independente şi refolosibile. Acestea sunt de fapt funcţii de JavaScript care acceptă input-uri arbitrare de la componentele parinte prin proprietatea „props” şi returnează elemente React care descriu ce ar trebui să apară pe ecran. De exemplu:

|  |
| --- |
| function Welcome(props) {  return <h1>Hello, {props.name}</h1>;  } |

Această funcţie are un parametru „props” care conţine date arbitrare şi returnează un element React (JSX).

|  |
| --- |
| class Welcome extends React.Component {  render() {  return <h1>Hello, {this.props.name}</h1>;  }  } |

De asemenea, componentele pot fi scrise ca si clase ES6. Această clasă este echivalentul funcţiei de mai sus.

### 1.1.3 Starea şi ciclul vieţii componentelor

Componentele pot fi dinamice datorită conceptului de stare ( state). Starea face ca un component să poată să schimbe informaţia între randări. Mai specific, starea este un obiect care menţine date în timpul ciclului de viaţă al componentului. Exemplu:

|  |
| --- |
| class Myclass extends React.Component {  constructor(){  this.state = { num: 123 }  }  } |

Starea poate fi schimbată prin utilizarea funcţiei setState(). Odată ce starea se modifică, componentul va modifica ce dispune în UI în funcţie de ce dade are în stare.

Ciclul de viaţă al unui component poate fi separat în 4 părţi:

* Iniţializare
* Montare
* Actualizare
* Demontare

Pe timpul iniţializării componentul îşi va seta starea şi proprietăţile. De obicei, acest proces se realizează într-un constructor. Odată ce iniţializarea s-a terminat, începe faza de montare, în care componentul se creează în DOM şi va fi afişat. În această fază se apelează funcţiile componentDidMount() şi componentWillMount(). În următoarea fază, adică actualizarea, starea componentului poate fi schimbată, iar astfel poate fi modificat conţinutul din UI în mod dinamic, iar funcţiile apelate sunt shouldComponentUpdate(), componentWillUpdate() şi componentDidUpdate(). În ultima fază, demontarea, componentul este scos din DOM şi se apelează funcţia componentWillUnmount().

## 1.2 Material-ui

Material-ui este o librărie open-source ce conţine componente React şi care implementează librăria Material Design de la Google.

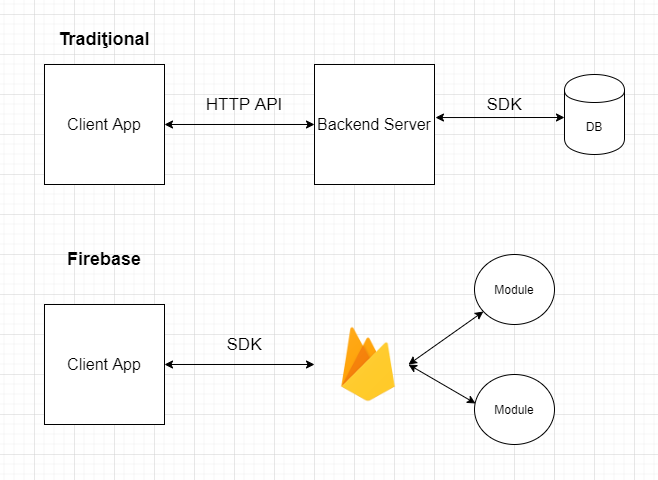
Avantantajele folosirii unei astfel de librării în detrimentul creeării fiecărui component de la 0 sunt:

* Integrarea uşoară într-un proiect
* Salvarea timpului care ar fi fost folosit la creerea de stiluri şi componente
* Aspectul plăcut al interfeţei de utilizator

## 1.3 Firebase

Firebase este o platformă cloud oferită de Google folosită pentru dezvoltarea aplicaţiilor mobile şi web. Această oferă o multitudine de unelte cu ajutorul cărora se pot construi aplicaţii, pe care în mod normal, dezvoltatorii ar fi trebuit să le implementeze. Poate fi considerat un BaaS adică „Backend as a service”, având în vedere serviciile pe care le oferă.

Figura 1.2.1 Firebase vs. tradiţional



### 1.3.1 Autentificare

Firebase Auth este un modul care conţine un sistem de autentificare care se foloseşte de email şi parolă. Acesta suportă, de asemenea, OAuth2 pentru Google, Facebook, Twitter şi GitHub.

Acest modul asigură identificarea utilizatorilor şi este esenţial pentru configurarea celorlalte module, mai ales dacă este nevoie să controlăm accesul la date pentru fiecare utilizator. De asemenea, autentificarea sigură se realizează cu uşurinţă, ceea ce în mod normal ar fi foarte complicat de implementat.

### 1.3.2 Bază de date Firestore

Firestore este numele bazei de date NoSQL oferită de Firebase. Conexiunea dintre aplicaţie şi baza de date se realizează prin un singur WebSocket, iar datorită acestui fapt, datele sunt sincronizate automat la viteza reţelei clientului. Firebase va trimite datele noi în momentul în care acestea sunt actualizate.

Deoarece baza de date este NoSQL, datele sunt memorate în documente sau conţin câmpuri cu valori. Aceste documente sunt la rândul lor o parte dintr-o colecţie, care poate fi folosită pentru organizare şi interogări. Documentele suporta multe tipuri de date, de la şiruri de caractere şi numere la obiecte complexe. De asemenea se pot crea subcolecţii în interiorul documentelor, cu ajutorul cărora se poate crea o structură de date ierarhică pe măsură ce baza de date creşte.

### 1.3.3 Sistem de depozitare a fişierelor

Firebase Storage furnizează o metodă uşoară pentru memorarea fişierelor binare. Adesea, este folosit pentru imagini, dar fişierele pot fi de orice tip.

Acest sistem de fişiere vine cu un set de reguli de securitate, care protejează împotriva utilizatorilor neautentificaţi, în timp ce oferă utilizatorilor autentificaţi drepturi detaliate de scriere şi citire.

## ASP.NET Core

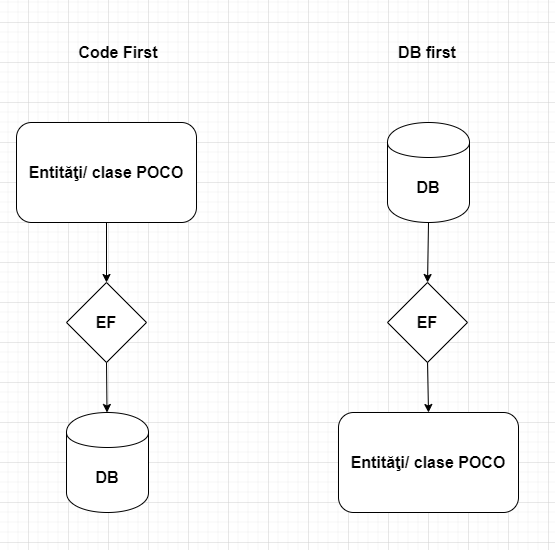
ASP.NET Core este un open-source framework care poate fi rulat pe multiple sisteme de operare precum Windows, Mac sau Linux. Acesta este folosit în dezvoltarea aplicaţiilor cloud, web, IoT, etc. Este bazat pe un set de pachete granulare NuGet, iar datorită acestui fapt, aplicaţia poate fi optimizată prin utilizarea pachetelor de care este nevoie.

Avantajele acestui framework sunt:

* Un sistem de configurare bazat pe mediul înconjurător
* Injecție de dependență încorporată
* Noi conducte ușoare și modulare de solicitare HTTP
* Posibilitatea de a găzdui pe IIS sau auto-gazdă în propriul proces
* Expediere complet ca pachete NuGet

Pe platforma ASP.NET Core se poate folosi Entity framework ( EF), un O/RM ( Object/ Relational Mapping) care dă posibilitatea dezvoltatorilor de a folosi un mecanism automat pentru accesarea şi memorarea de date într-o bază de date relaţională. EF oferă două modalităţi de implementare a modelului bazei de date, şi anume Code First sau DB first:

Figura 1.4.1



Modalitatea de implementare a bazei de date a fost „Code First”. Întâi au fost scrise clasele care reprezintă entităţile, care la rândul lor au fost convertite cu ajutorul EF în tabele.

### Pachete NuGet

Pentru dezvoltarea API-ului s-au folosit următoarele pachete:

* Aglomera 1.1.0
* Microsoft.AspNetCore.Cors 2.2.0
* Microsoft.EntityFrameworkCore 3.1.4
* Microsoft.EntityFrameworkCore.SqlServer 3.1.4
* Microsoft.EntityFrameworkCore.Tools 3.1.4

Aglomera este un pachet care furnizează algoritmi de clusterizare ierarhică. Aceştia au fost folosiţi pentru creerea listei de recomandări.

Microsoft.AspNetCore.Cors 2.2.0 este folosit pentru configurarea CORS ( Cross-Origin Resource Sharing). CORS este un mecanism care foloseşte antete HTTP adiţionale pentru a comunica către browser să dea acces unei aplicaţii web care rulează la o origine la resursele alese de pe altă origine. Acesta este folosit din motive de securitate.

Restul pachetelor au fost folosite pentru folosirea ORM-uli EF. Acesta este folosit pentru creearea migrărilor bazei de date. În cazul în care se va dori modificarea bazei de date, EF va facilita acest proces prin generarea automată a migrării.

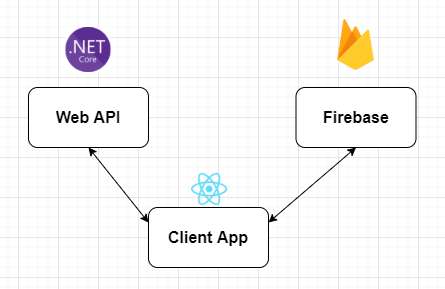
# Capitolul II. Arhitectura aplicaţiei

În acest capitol voi evidenția arhitectura aplicației, principalele module ale aplicației și modul de comunicare al acestora.

Aplicaţia a fost împărţită în 3 părţi:

* Client – realizat ca aplicaţie web cu React
* Firebase – backend pentru controlul utilizatorilor şi alte detalii
* Web API – server de backend pentru recomandările utilizatorilor

Figura 2.1.1 Componentele majore ale aplicaţiei



După cum se poate observa în figura de mai sus, aplicaţia client comunică concomitent cu api-ul ASP.NET Core şi Firebase.

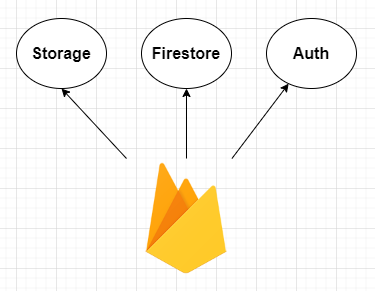
În continuare se vor explica şi prezenta arhitectura fiecărui component al aplicaţiei.

## 2.1 Backend Firebase

După cum am menţionat în capitolul anterior, am folosit 3 module din Firebase, şi anume:

* Autentificare
* Baza de date Firestore
* Firebase Storage

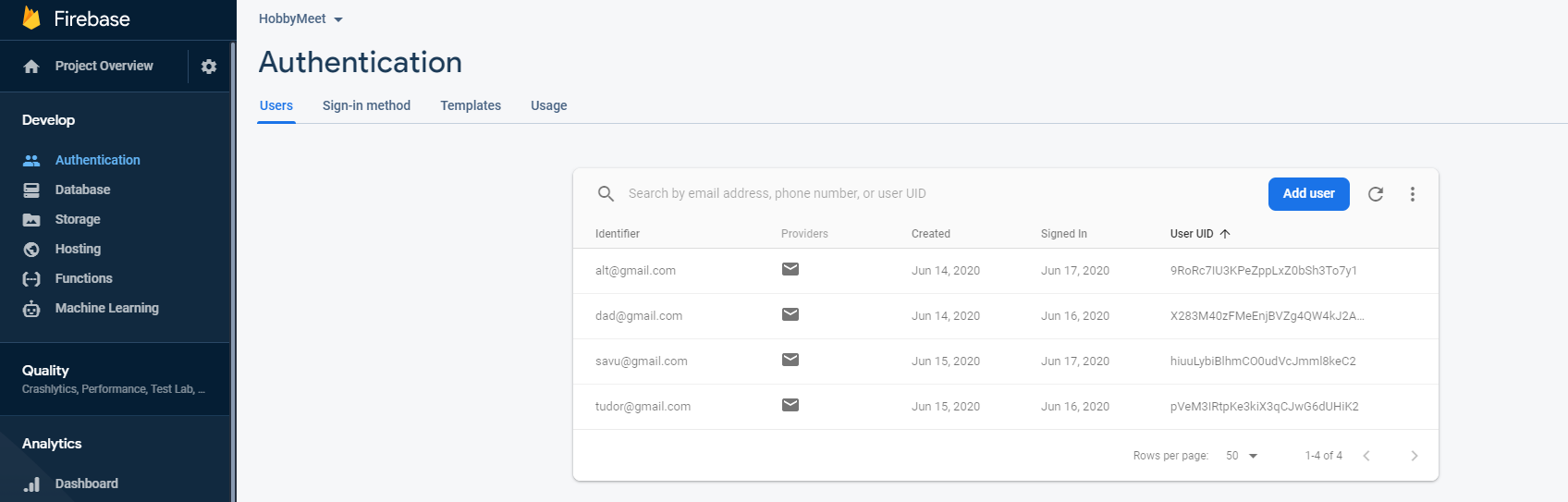
Figura 2.1.1 Module Firebase



### 2.1.1 Modulul de autentificare

Autentificarea se realizează pe baza unui e-mail şi parolă. Odată ce utilizatorul creează un cont, acesta va fi adăugat în lista conturilor:

Figura 2.1.2



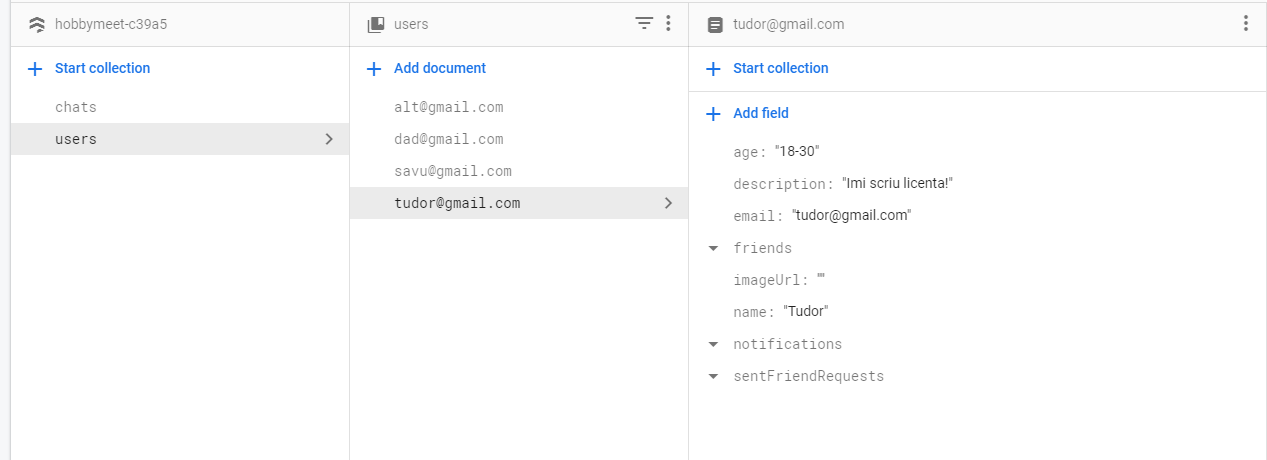
### 2.1.2 Baza de date

Baza de date fiind NoSQL, a fost împărţită în 2 colecţii:

* users
* chats

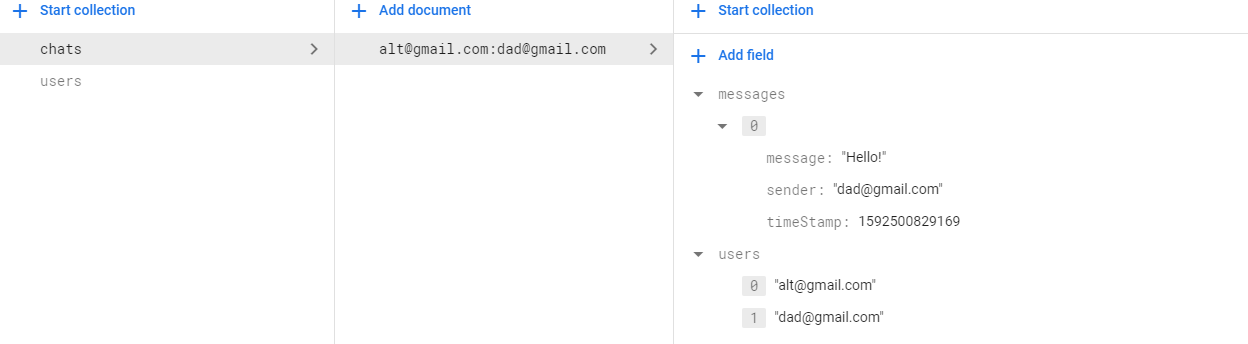
Colecţia „users” conţine documente numite în funcţie de e-mail-urile utilizatorilor. Aceste documente conţin detalii despre utilizator precum numele, o descriere, lista prietenilor, un link pentru imaginea de profil, etc.

Figura 2.1.2 Colecţia users



Colecţia „chats” conţine documente numite cu în funcţie de cine este implicat în conversaţia respectivă. Am convenit ca numele acestor document va începe cu primul e-mail în ordine alfabetică, apoi va fi legat de caracterul „:”, iar apoi de celălalt e-mail.

Figura 2.1.3 Colecţia chats

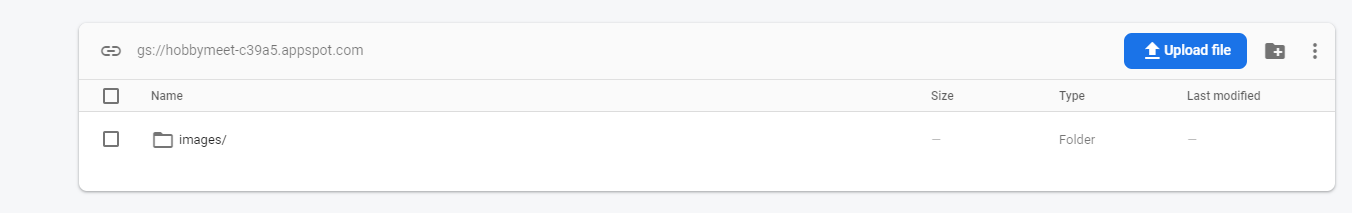


Mesajele sunt memorate într-un vector de obiecte care conţin mesajul, emiţătorul şi data la care a fost trimis.

### 2.1.3 Firestore

Acest modul a fost folosit pentru memorarea imaginilor de profil ale utilizatorilor. A fost creat un fişier „images” care conţine toate imaginile.

Figura 2.1.4 Fişierul de imagini



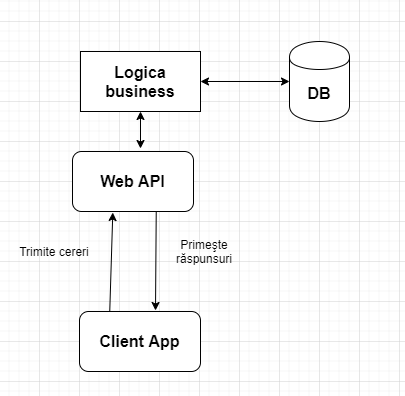
## 2.2 Backend ASP.NET Core

Am ales să folosesc un web API separat de Firebase pentru a putea realiza recomandările utilizatorului folosind un algoritm de clusterizare ierarhică. Voi intra în detalii despre algoritm în următorul capitol. Acest Rest API va accesa o bază de date SQL Server care va conţine hobby-urile şi e-mail-urile utilizatorilor.

### 2.2.1 Structura generală

API-ul va furniza aplicaţiei client date necesare pentru dispunerea recomandărilor. Clientul va apela API-ul prin cereri HTTP prin care se vor putea înregistra noi utilizatori prin metode POST, se vor putea furniza hobby-urile altor utilizatori, precum şi o listă cu toate hobby-urile din baza de date prin metode GET sau se vor putea modifica sau şterge utilizatorii prin PUT sau respectiv, DELETE.

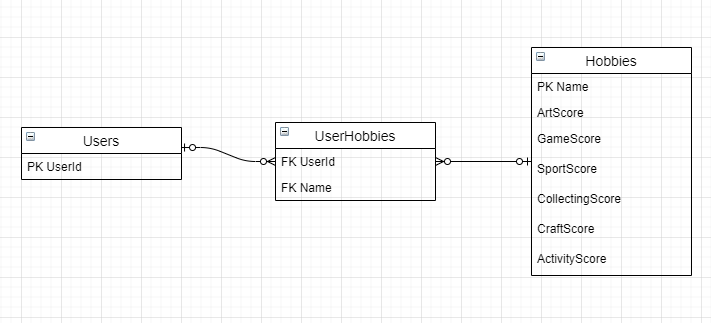
Figura 2.2.2



### 2.2.2 Structura bazei de date

Rolul acestei baze de date este de a memora hobby-urile fiecărui utilizator, pentru a putea calcula recomandările acestuia.

Figura 2.2.1 Tabelele bazei de date



După cum se observă în diagramă, relaţia dintre tabelele „Users” şi „Hobbies” este „many to many”. Aceasta a fost realizată prin adăugarea unui tabel de legătură numit „UserHobbies”. În acest tabel de legătură se va memora ce hobby-uri are un utilizator.

Scorurile desemnate în tabelul de hobby-uri reprezintă nişte valori acordate arbitrar pentru fiecare hobby în parte. Cu ajutorul lor, vom putea construi un scor total pentru fiecare utilizator şi să se creeze lista de recomandări.

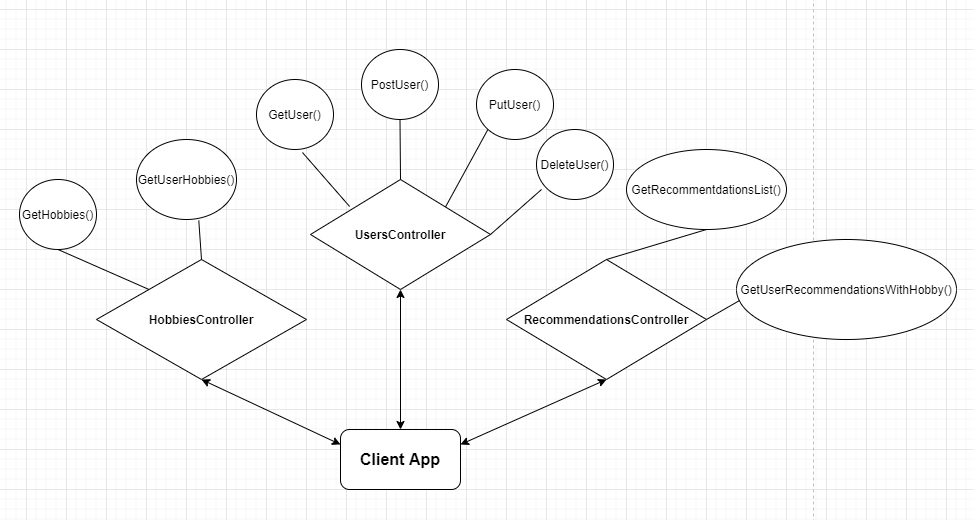
### 2.2.3 Controlere

API va dispune aplicaţiei client 3 controlere şi anume:

* HobbiesController – va furniza o listă cu toate hobby-urile din baza de date
* UsersController – se va ocupa de operaţiile CRUD pentru utilizatori
* RecommendationsController – va furniza lista cu recomandări de persoane

Fiecare controler va dispune multiple puncte de acces care vor apela anumite funcţii pe care aplicaţia client le va putea folosi.

Figura 2.2.2 Structura controlerelor



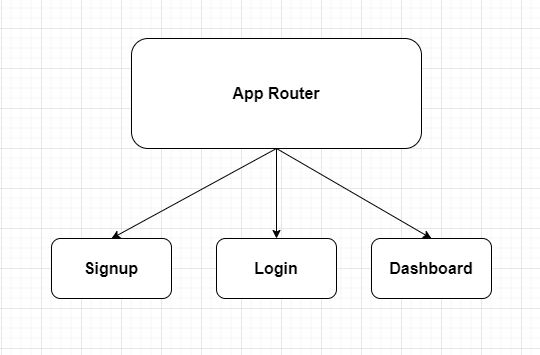
## 2.3 Frontend React

Aplicaţia web a fost împărţită în 3 pagini, şi anume:

* Signup – o pagina pentru înregistrare
* Login – o pagină pentru autentificare
* Dashboard – cea mai complexă pagină a aplicaţiei

Aceste pagini sunt accesate prin intermediul unui „Router”.

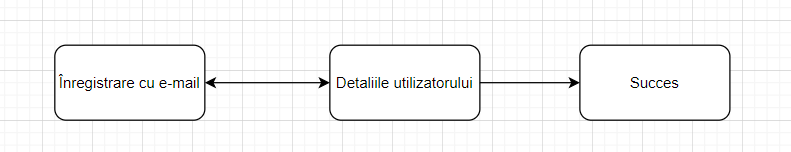
Figura 2.3.1 Rutarea aplicaţiei



### 2.3.1 Formularul de înregistrare

Acest formular este realizat în mai mulți pași și anume:

Figura 2.3.2 Structura formularului

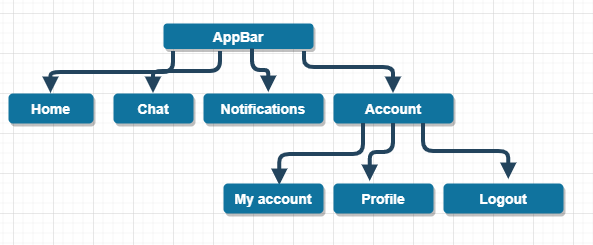


În primul formular se vor introduce e-mail-ul și parola care vor fi ulterior folosite pentru autentificare. În cel de al doilea formular se vor introduce detaliile utilizatorului, care cuprind: 3 hobby-uri, descriere, vârstă și nume. În final, utilizatorul va fi redirecționat la o pagină care îl va informa de succesul creării contului.

### 2.3.2 Structura paginii principale

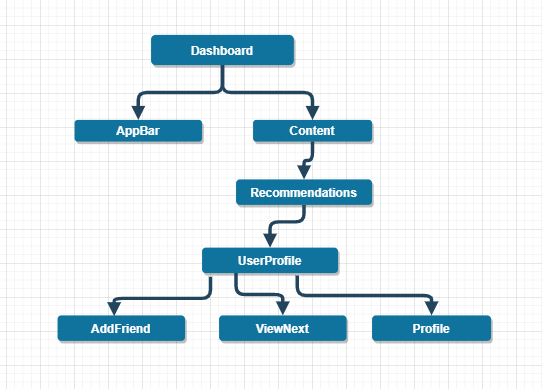
Pagina principală ( Dashboard) va avea conţinut diferit în funcţie de starea de autentificare a utilizatorului. Dacă acesta este neautentificat, va avea 2 opţiuni în navigaţia aplicaţiei şi anume autentificare sau înregistrare. În cazul alternativ, acesta va avea acces la mai multe componente ilustrate în următoarele diagrame.

Figura 2.3.3 Navigaţia aplicaţiei în cazul utilizatorului autentificat



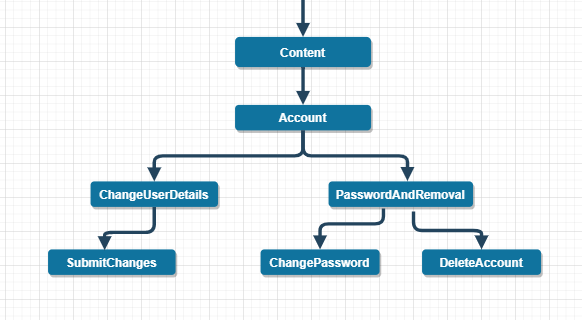
Iniţial, pe pagina principală, se vor dispune un tab de navigare şi recomandările utilizatorului. Conţinutul pagini, însă, se poate schimba în funcţie de butoanele de pe navigare. Următoarele figuri ilustrează structura generală a fiecărui component care poate fi dispus în conţinutul paginii.

Figura 2.3.4 Componentul de recomandări



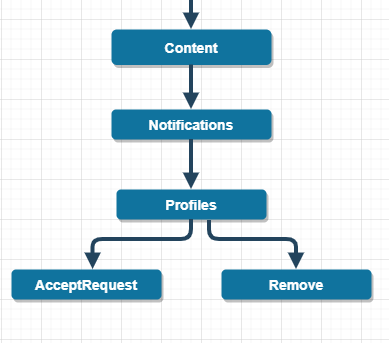
Componentul pentru modificarea contului conţine două formulare prin care utilizatorul va putea să modifice detaliile contului său precum hobby-urile alese, descrierea, etc. De asemenea, se va dispune şi opţiunea de a modifica parola sau de a şterge contul.

Figura 2.3.5 Componentul pentru modificarea contului



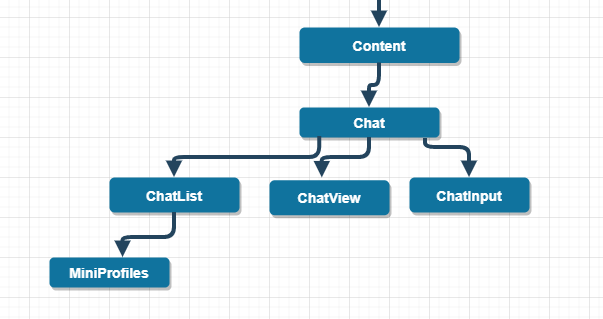
Componentul de notificări va conţine cererile de prietenie, care vor fi afişate ca nişte mini-profiluri ale utilizatorilor care au trimis cererile respective, alăturate de un buton de adăugare şi unul pentru ştergere.

Figura 2.3.6 Componentul de notificări



Componentul „chats” va avea ca parte din structură o listă cu prietenii utilizatorului, o vizualizarea a conversaţiei şi o intrare pentru scrierea mesajului.

Figura 2.3.7 Componentul de conversaţii



# Capitolul III. Detalii de implementare

În acest capitol se vor prezenta modalităţile de implementare a aplicaţiei client, a relaţiei acesteia cu Firebase şi web API-ul care furnizează recomadările clientului.

## 3.1 Web API

Acest web API va rula în permanență pentru a putea servi aplicația client cu datele necesare pentru creearea de recomandări și pentru a memora hobby-urile.

### 3.1.1 Configurare

Pentru configurarea API și pentru adăugarea de servicii, programul folosește clasa „Startup.cs”. Aici sunt alese ce componente de middleware sunt folosite, precum și configurarea CORS.

|  |
| --- |
| services.AddCors(options =>  {  options.AddPolicy(MyAllowSpecificOrigins, builder =>  {  builder.WithOrigins("http://localhost:3000")  .AllowAnyHeader()  .AllowAnyMethod();  });  });  ...  app.UseCors(options => options.WithOrigins("http://localhost:3000")); |

Se adaugă un serviciu CORS care permite accesul cu orice antet la orice metodă, iar variabila MyAllowSpecificOrigins reprezintă un șir de caractere care va fi folosit la adnotarea metodelor din controlere. De exemplu:

|  |
| --- |
| [HttpGet]  [EnableCors("\_myAllowSpecificOrigins")]  public async Task<ActionResult<IEnumerable<HobbyDTO>>> GetHobbies()  {  return await \_context.Hobbies.Select(h=> new HobbyDTO()  {  Name = h.Name  }).ToListAsync();  } |

Adnotarea HttpGet reprezintă faptul că metoda este de tip GET, iar EnableCors permite clientului acces la această metodă.

### 3.1.2 Algoritmul de recomandări

Pe lângă acest algoritm, am pus la îndemâna clientului o metodă mai simplă pentru obținerea recomandărilor. Aceasta va primi ca intrare de la client un nume de hobby după care vor fi selectați din baza de date utilizatori care sunt asociați cu acel hobby:

|  |
| --- |
| public async Task<ActionResult<IEnumerable<User>>>  GetUserRecommendationsWithHobby(string id, string hobby)  {  var usersWithHobby = \_context.UserHobbies  .Where(uh => uh.UserId != id && uh.Name==hobby)  .Select(uh => new User()  {  UserId = uh.UserId  });  return await usersWithHobby.ToListAsync();  } |

După cum am menționat în capitolul anterior, fiecare hobby are asociat câte un scor pentru 6 categorii. Acestea sunt:

* ArtScore – cât de mult un hobby este o artă
* ActivityScore – cât de mult un hobby implică o activitate în natură sau afară
* CollectingScore – cât de mult hobby-ul implică colecționare de obiecte
* SportScore – cât de mult un hobby este un sport sau implică efort fizic
* GameScore – cât de mult un hobby este un joc
* CraftScore – cât de mult de pune accentul pe creare sau asamblare

Fiecare dintre aceste scoruri au valori de la 0 la 100 și reprezintă cât de mult se încadrează un hobby într-o categorie. De exemplu, grădinăritul ar fi un hobby cu ActivityScore și CraftScore mari, în timp ce colecționarea de timbre ar avea CollectingScore și ArtScore mari. Aceste scoruri au fost acordate subiectiv, în funcție de cum cred eu că ar fi anumite hobby-uri.

Cu ajutorul acestor valori, putem crea un scor total pentru fiecare utilizator. Astfel aceștia pot fi vizualizați ca niște puncte într-un spațiu cu 6 dimensiuni, unde dimensiunile sunt scorurile hobby-urilor.

Algoritmul folosit este de tip botom-up ( de jos în sus). Acesta va forma inițial toate clusterele de tip singleton, adică clustere care conțin un singur punct/obiect. Următorul pas este combinarea clusterelor care au „similaritatea maximă”, adică vor fi combinate 2 clustere într-un alt cluster nou în funcție de metrica de similaritate folosită. Se va continua acest proces până când toate obiectele sunt conținute într-un singur cluster.

Obiectele/punctele au fost modelate astfel într-o clasă separată, unde se vor aduna scorurile pentru fiecare categorie a fiecărui hobby pe care îl poate avea un utilizator.

|  |
| --- |
| public class UserScore : IComparable<UserScore>  {  public string UserId { get; set; }  public int ArtScore { get; set; }  public int GameScore { get; set; }  public int SportScore { get; set; }  public int CollectingScore { get; set; }  public int CraftScore { get; set; }  public int ActivityScore { get; set; }  ... } |

Metrica folosită este euclidiană, calculându-se distanțele între instanțe folosind formula:

, unde n reprezintă numărul de dimensiuni.

|  |
| --- |
| class DssimilarityMetric : IDissimilarityMetric<UserScore>  {  public double Calculate(UserScore i1, UserScore i2)  {  return Math.Sqrt(  (i1.ActivityScore - i2.ActivityScore) \* (i1.ActivityScore - i2.ActivityScore)+  (i1.ArtScore - i2.ArtScore) \* (i1.ArtScore - i2.ArtScore)+  (i1.CollectingScore- i2.CollectingScore) \* (i1.CollectingScore - i2.CollectingScore)+  (i1.CraftScore - i2.CraftScore) \* (i1.CraftScore - i2.CraftScore)+  (i1.GameScore - i2.GameScore) \* (i1.GameScore - i2.GameScore)+  (i1.SportScore - i2.SportScore) \* (i1.SportScore - i2.SportScore));  }  } |

Având metrica și obiectele implementate, pachetul Aglomera se ocupă de restul algoritmului, inclusiv de actualizarea matricii de legătură ( linkage). Abordarea mea a fost de tip lazy loading în ceea ce privește calcularea scorurilor totale pentru fiecare utilizator. Acest fapt se poate observa în codul de mai jos.

|  |
| --- |
| public List<string> GetRecommendationsList(string id)  {  var dataSet = new HashSet<UserScore>();  foreach (var usr in \_context.Users)  {  dataSet.Add(GetUserScore(usr.UserId));  }  var metric = new DssimilarityMetric();  var linkage = new SingleLinkage<UserScore>(metric);  var algorithm = new AgglomerativeClusteringAlgorithm<UserScore>(linkage);  var clusteringResult = algorithm.GetClustering(dataSet);  return GetListFromClustResult(clusteringResult, id);  } |

Algoritmul va produce un obiect care conține un vector cu toate stadiile clusterizării. În penultimul stadiu, se vor forma două clustere, cu ajutorul cărora se va forma o mulțime de obiecte User care vor fi recomandate utilizatorului în funcție de plasarea obiectului propriu între cele două clustere. Acest proces este realizat de funcția GetListFromClustResult():

|  |
| --- |
| public List<string> GetListFromClustResult(ClusteringResult<UserScore> cr, string id)  {  List<string> cluster1 = new List<string>();  List<string> cluster2 = new List<string>();  User userObj = new User() {UserId = id};  int index = cr.Count - 2;  int i = 0;  foreach (var clust in cr[index])  {  foreach (var dp in clust)  {  if (i == 0)  {  cluster1.Add(dp.UserId);  }  else  {  cluster2.Add(dp.UserId);  }  }  i++;  }  if (cluster1.Contains(id) == true && cluster1.Count>1)  {  cluster1.Remove(id);  return cluster1;  }  cluster2.Remove(id);  return cluster2;  } |

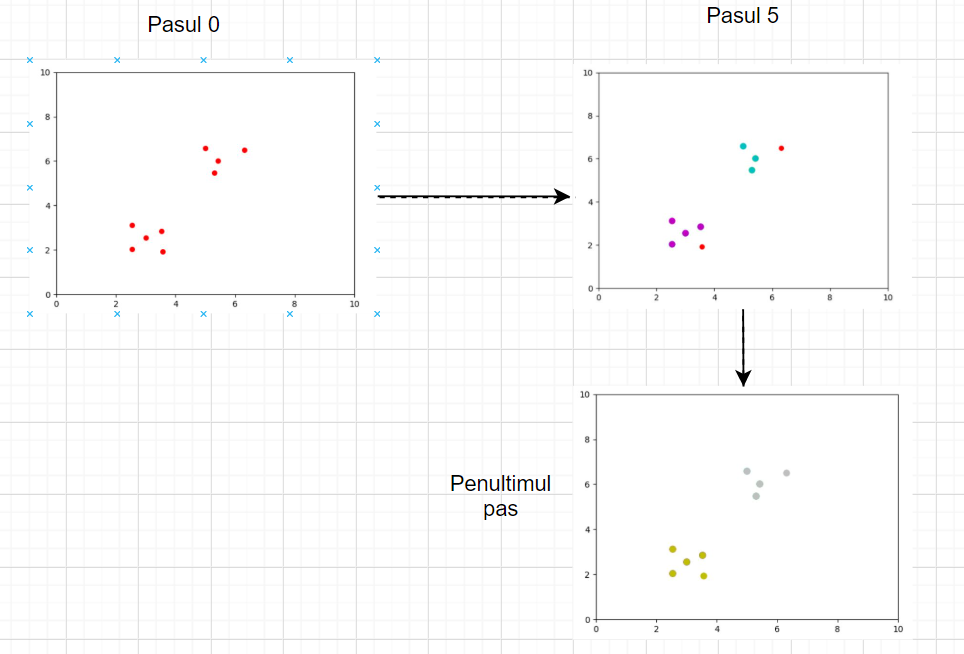
De asemenea, funcția acoperă un caz extrem, și anume acela în care obiectul utilizatorului pentru care se creează recomandările ajunge în penultimul stadiu al algoritmului de clustering sub formă de cluster singleton, adică cu un singur punct. Se va returna clusterul opus în acest caz.

Pentru a avea acces la aceste recomandări, aplicația client va face o simplă cerere de tip GET la metoda din controlerul recomandărilor.

|  |
| --- |
| [HttpGet(template: "{id}")]  [EnableCors("\_myAllowSpecificOrigins")]  public List<User> GetUserRecommendations(string id)  {  var usrList = new List<User>();  foreach (var email in GetRecommendationsList(id))  {  usrList.Add(new User()  {  UserId = email  });  }  return usrList;  } |

Am scris un mic script de python pentru a reprezenta grafic acest proces peste obiecte cu 2 dimensiuni:

Figura 3.1.1 Clusterizare ierarhică botom up



## 3.2 Client

### 3.2.1 Înregistrare

După ce sunt completate formularele de înregistrare, clientul va apela prin SDK-ul Firebase modulul de autentificare și va încerca să creeze un nou utilizator cu e-mail-ul și parola introduse mai devreme. Dacă această acțiune este completată cu succes, se va apela modulul Firestore pentru a creea un document nou asociat cu noul cont. Apoi, se va apela web API-ul unde se vor înregistra noul utilizator și hobby-urile selectate. Bineînțeles, datele introduse sunt întâi validate în formulare înainte de încercarea creării noului cont.

|  |
| --- |
| firebase  .auth()  .createUserWithEmailAndPassword(this.props.values.email, this.props.values.password)  .then(authRes => {  const fireUserObj = {  email: authRes.user.email,  friends: [],  messages: [],  name: this.props.values.name,  age: this.props.values.age,  description: this.props.values.description,  imageUrl: "",  notifications: [],  sentFriendRequests: []  }  firebase  .firestore()  .collection("users")  .doc(authRes.user.email)  .set(fireUserObj)  .then(() => {  axios.post("https://localhost:44379/api/users", {  "UserId": authRes.user.email,  "HobbyNames": this.props.values.hobbiesChosenByUser  }).then(apiSuccess => {  this.props.nextStep();  this.setState({ isLoading: false })  }, apiErr => {  this.setState({ error: "API error" })  this.setState({ isLoading: false })  });  }, dbErr => {  this.setState({ error: dbErr.message })  });  this.setState({ isLoading: false })  }, authErr => {  this.setState({ error: authErr.message })  this.setState({ isLoading: false })  }) |

Odată ce contul a fost creeat cu succes, utilizatorul se va putea autentifica.

### 3.2.2 Navigație

Inițial, dacă utilizatorul nu este autentificat, se vor afișa pe tab-ul de navigație opțiunile de login sau înregistrare. În caz contrar, se vor afișa patru pictograme care pot schimba conținutul paginii:

Figura 3.2.1 Navigația



Componenta de navigație va putea trimite date despre ce componentă trebuie afișată în conținutul paginii printr-o funcție care este primită de la componenta părinte. Această funcție poate primi un nume de component și anumite argumente:

|  |
| --- |
| handleComponentChange = (component, arg) => {      if (arg !== "") {        arg.title = "Your profile"        this.setState({ profileUserObj: arg }, () => {          this.setState({ componentToRender: component })        })      } else {        this.setState({ componentToRender: component })      }    }  ...  <Navbar handleComponentChange={this.handleComponentChange} /> |

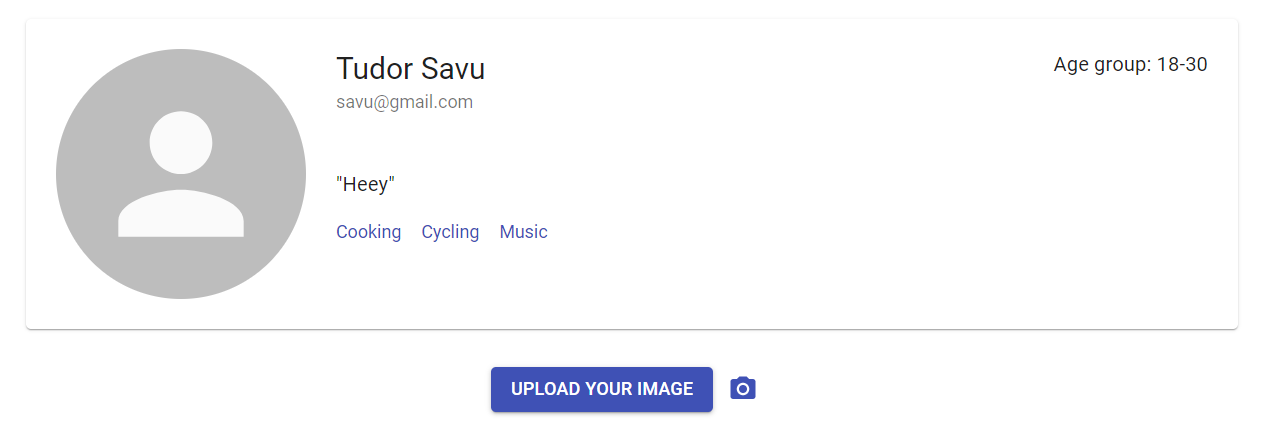
În funcție de numele primit de funcție, se va reactualiza starea paginii principale și se va afișa componentul ales. Acest lucru este posibil deoarece funcția render() a componentului conține o instrucțiune de tip switch() care va returna componentul cerut și deorece această funcție este apelată din nou atunci când starea s-a actualizat.

|  |
| --- |
| switch (this.state.componentToRender) {  case "dashboard":  return ...  case "account":  return ...  case "profile":  return ...  case "notifications":  return ...  case "chat":  return ...  default:  break;  } |

### 3.2.3 Profilul utilizatorului

Fiecare utilizator își va putea vizualiza profilul, unde există și opțiunea de a încărca o imagine.

Figura 3.2.2 Profil de utilizator

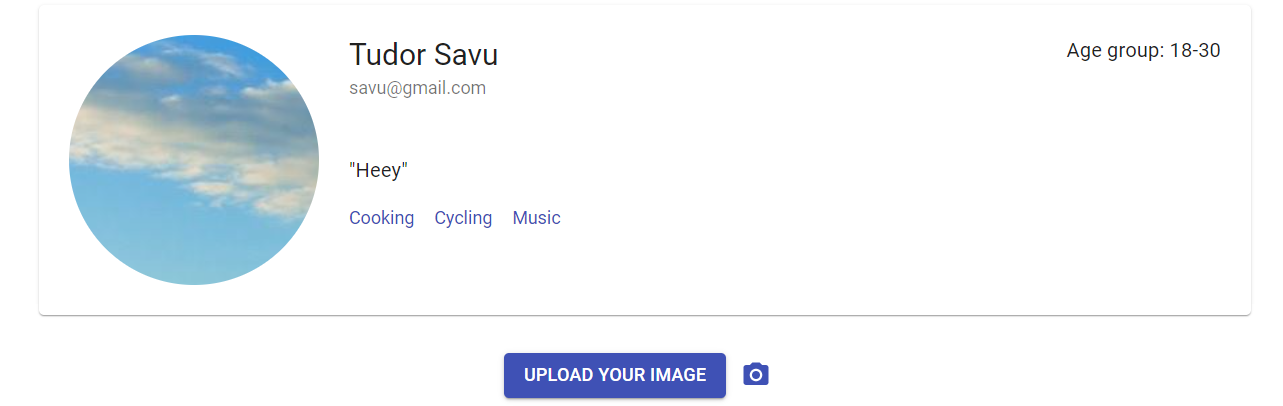


Utilizatorul va putea selecta o imagine apăsând butonul cameră, iar după va putea să o încarce pe Firestore:

|  |
| --- |
| handleUpload = () => {  if (this.state.image === null) {  alert("To upload a profile picture, you must first pick one by clicking the camera icon!");  return;  } else {  firebase.storage().ref(`images/${this.props.profileUserObj.email}`).put(this.state.image)  .then(  res => {  firebase.storage().ref(`images/${this.props.profileUserObj.email}`).getDownloadURL()  .then(url => {  this.setState({ url: url })  firebase.firestore().collection("users").doc(this.props.profileUserObj.email)  .update({  imageUrl: url  })  })  );  } |

Se va accesa fișierul de imagini unde se va adăuga sau schimba automat poza de profil încărcată. După acest proces se va actualiza documentul utilizatorului cu o referință la imaginea aleasă de acesta.

Figura 3.2.3

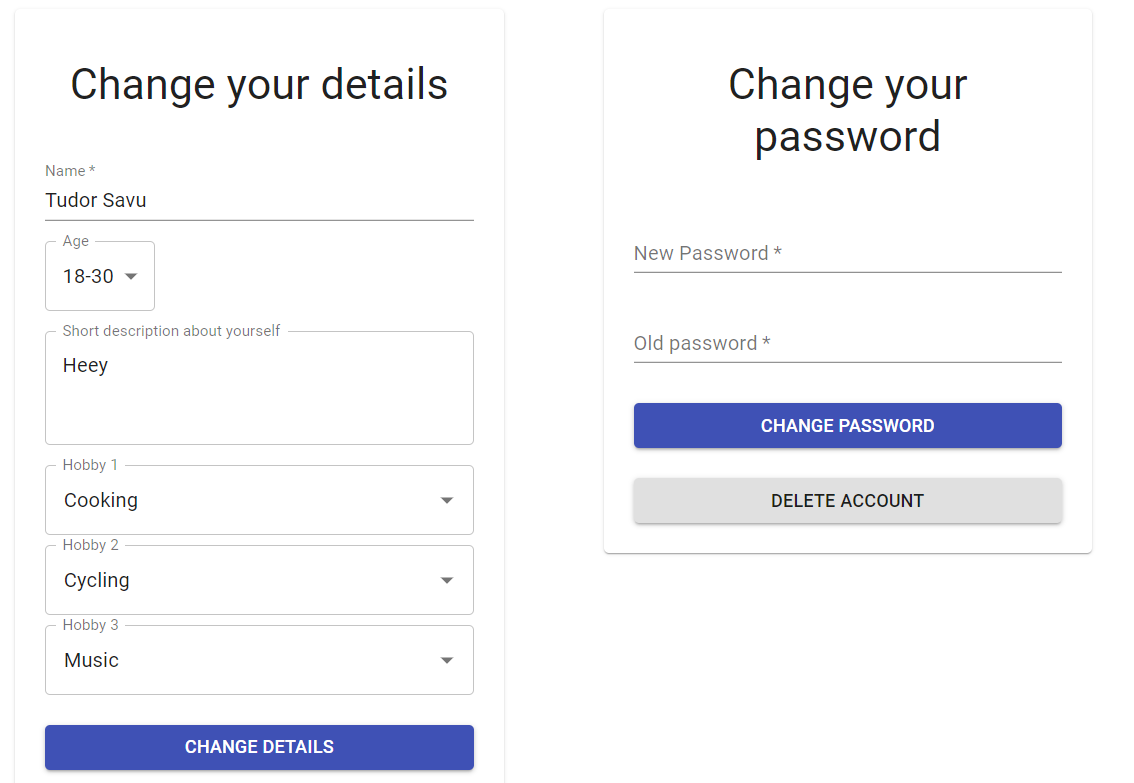


### 3.2.4 Modificarea contului

Pentru a putea modifica datele unui cont se pun la dispoziție două formulare prin care utilizatorul poate schimba detalii sau șterge contul. Asemănător cu înregistrarea, înainte ca datele modificate să fie trimise la Firestore și API, sunt întâi validate, iar apoi se vor apela API-ul cu o cerere PUT, respectiv Firebase.

|  |
| --- |
| submitChanges = () => {  if (this.validate() === true) {  const userHobbies = {  UserId: this.props.userObj.email,  HobbyNames: [this.state.hobby1, this.state.hobby2, this.state.hobby3]  }  axios.put("https://localhost:44379/api/users", userHobbies).then(() => {  firebase  .firestore()  .collection("users")  .doc(this.props.userObj.email)  .update({  name: this.state.name,  age: this.state.age,  description: this.state.description,  }).then(() => {  alert("Your details have been updated.")  })  })  }  } |

Figura 3.2.4 Formularele pentru modificarea contului



Înainte ca un utilizator să își poată schimba parola sau să își șteargă contul, Firebase va necesita o re-autentificare din motive de securitate.

|  |
| --- |
| deleteAccount = () => {  const response = prompt("Type your password to remove your account.")  this.setState({ deletionPassword: response }, () => {  const user = firebase.auth().currentUser;  const credential = firebase.auth.EmailAuthProvider.credential(  user.email, this.state.deletionPassword  )  user.reauthenticateWithCredential(credential).then(() => {  axios.delete("https://localhost:44379/api/Users/" + user.uid).then(() => {  user.delete().then(() => {  firebase.firestore().collection("users").doc(user.email).delete().then(() => {  this.props.history.push("/login")  })  })  })  }).catch(error => {  this.setState({ error: "Deletion password is invalid!" })  })  })  } |

### 3.2.5 Recomandări

Primul component care se dispune utilizatorului în momentul accesării pagini principale este cel de recomandări. Inițial se vor încărca din componentul părinte în funcția de montare, o listă cu e-mail-urile utilizatorilor recomandați.

|  |
| --- |
| componentDidMount() {  firebase.auth().onAuthStateChanged((user) => {  if (user) {  this.setState({ user: user })  axios.get("https://localhost:44379/api/recommendations/"+user.email)  .then(res => {  this.setState({ recommendations: res.data })  })  ... |

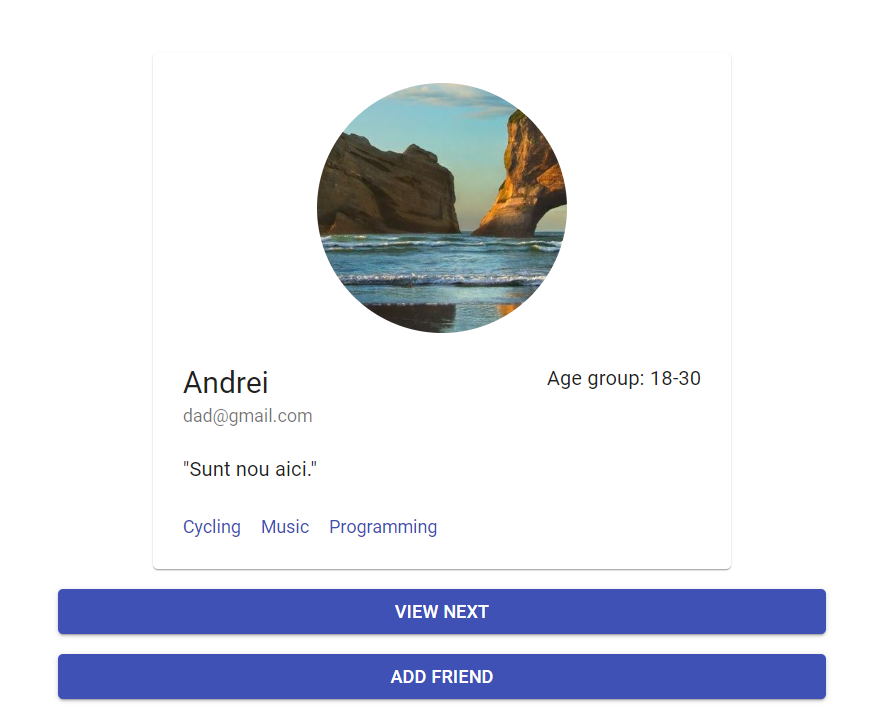
Această listă este, apoi, filtrată în funcție de ce prieteni , ce cereri de prietenie a trimis și de la cine a primit notificări de prietenie deja utilizatorul.

|  |
| --- |
| filterFriends = () => {  const data = this.state.recommendations;  var fri = this.state.userObj.friends  const not = this.state.userObj.notifications  const req = this.state.userObj.sentFriendRequests  const res = [...fri, ...req, ...not]  for (let i = 0; i < data.length; i++) {  const element = data[i].userId;  if (res.includes(element)) {  data.splice(i, 1);  i = i - 1;  }  }  } |

Apoi, în componentul de recomandări, se va itera prin această listă, iar la fiecare moment al iterației se vor lua date despre utilizatorul recomandat din Firestore și API, cu ajutorul cărora se vor construi profilurile din UI prin reactualizarea stării.

|  |
| --- |
| firebase.firestore().collection("users").doc(email).get().then(doc => {  axios.get("https://localhost:44379/api/users/" + email).then(hobbies => {  this.setState({  current: {  age: doc.get("age"),  description: doc.get("description"),  email: doc.get("email"),  name: doc.get("name"),  url: doc.get("imageUrl"),  hobbies: hobbies.data  }  })  })  }) |

Figura 3.2.5 Profil de recomandare



### 3.2.6 Notificări

Atunci când se trimite o cerere de prietenie, se va trimite o notificare utilizatorului căruia îi este destinată cererea și se va adăuga aceasta cerere în lista de cereri trimise a emițătorului. Acest lucru este făcut pentru filtrarea recomandărilor ambilor utilizatori.

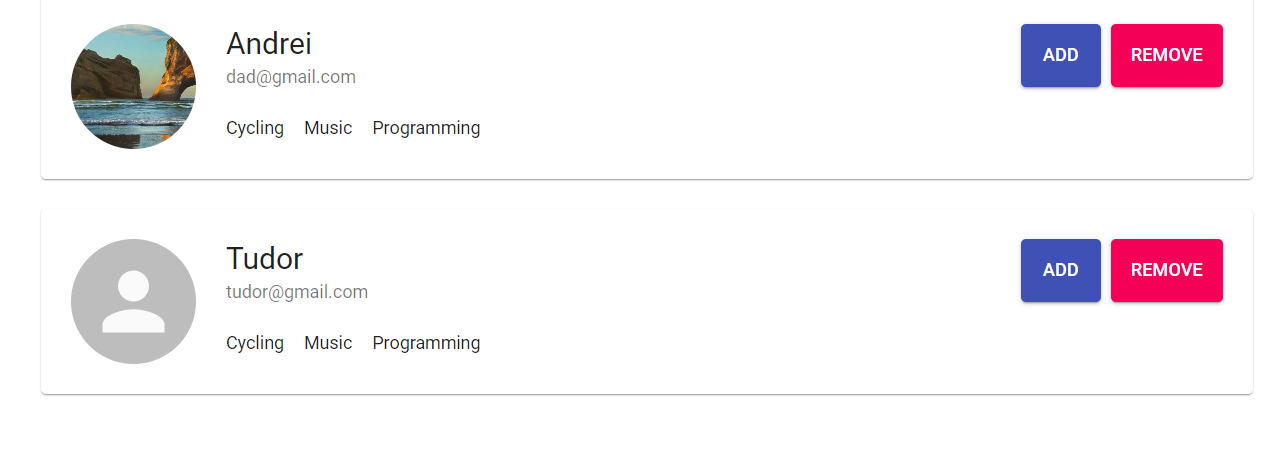
|  |
| --- |
| sendFriendRequest = () => {  const loggedUser = firebase.auth().currentUser.email;  firebase  .firestore()  .collection("users")  .doc(loggedUser)  .update({  sentFriendRequests: firebase.firestore.FieldValue.arrayUnion(this.state.current.email)  })  firebase  .firestore()  .collection("users")  .doc(this.state.current.email)  .update({  notifications: firebase.firestore.FieldValue.arrayUnion(loggedUser)  })  } |

Lista notificărilor este încărcată în pagina principală, iar apoi este trimisă către componentul notificărilor unde pentru fiecare intrare din lista de notificări se va apela Firestore și API-ul pentru mai multe date despre cine a trimis notificarea.

|  |
| --- |
| const notif = this.props.userObj.notifications  for (let i = 0; i < notif.length; i++) {  const email = notif[i];  firebase.firestore().collection("users").doc(email).get()  .then(doc => {  axios.get("https://localhost:44379/api/users/" + email).then(res => {  const userObj = {  email: doc.get("email"),  name: doc.get("name"),  imageUrl: doc.get("imageUrl"),  hobbies: res.data  }  this.setState({ notiData: [...this.state.notiData, userObj] })  })  })  }  } |

Notificările vor fi dispuse într-o listă care vor include 2 butoane, unul pentru adăugare ca prieten, iar celălalt pentru ștergere.

Figura 3.2.6 Notificări



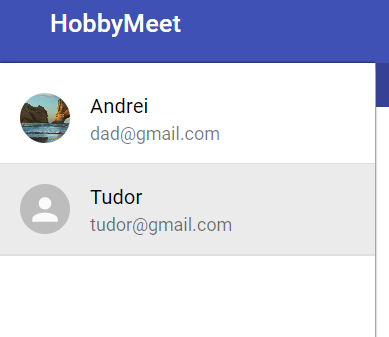
### 3.2.7 Conversații și lista de prieteni

În momentul în care utilizatorul accesează acest component, se vor căuta în baza de date conversațiile acestuia cu prietenii săi. Documentele de conversație conțin un vector cu e-mail-urile celor doi participanți la conversație.

|  |
| --- |
| firebase.firestore().collection("chats").where("users", "array-contains", email)  .onSnapshot(res => {  const chats = res.docs.map(\_doc => \_doc.data());  this.setState({ chats: chats }, () => { })  }) |

De asemenea se vor încărca și date despre prietenii utilizatorului pentru a fi dispuși într-o listă în UI.

Figura 3.2.7 Notificări



Se poate alege unul dintre prietenii din listă, iar la apăsarea pe unul din ei, se va deschide componentul de conversație.

Reamintesc faptul că documentele de conversație au fost denumite în funcție de e-mail-urile utilizatorilor după o convenție. Numele acestor documente este alcătuit din primul e-mail în ordine alfabetică, urmat de „:”, iar apoi urmat de celălalt e-mail. În concluzie, pentru obținerea documentului de conversație corect avem nevoie de e-mail-ul utilizatorului autentificat și de cel al prietenului selectat din listă. Ordinea alfabetică poate fi aflată prin memorarea celor 2 e-mail-uri într-un vector care poate fi ordonat cu ușurință.

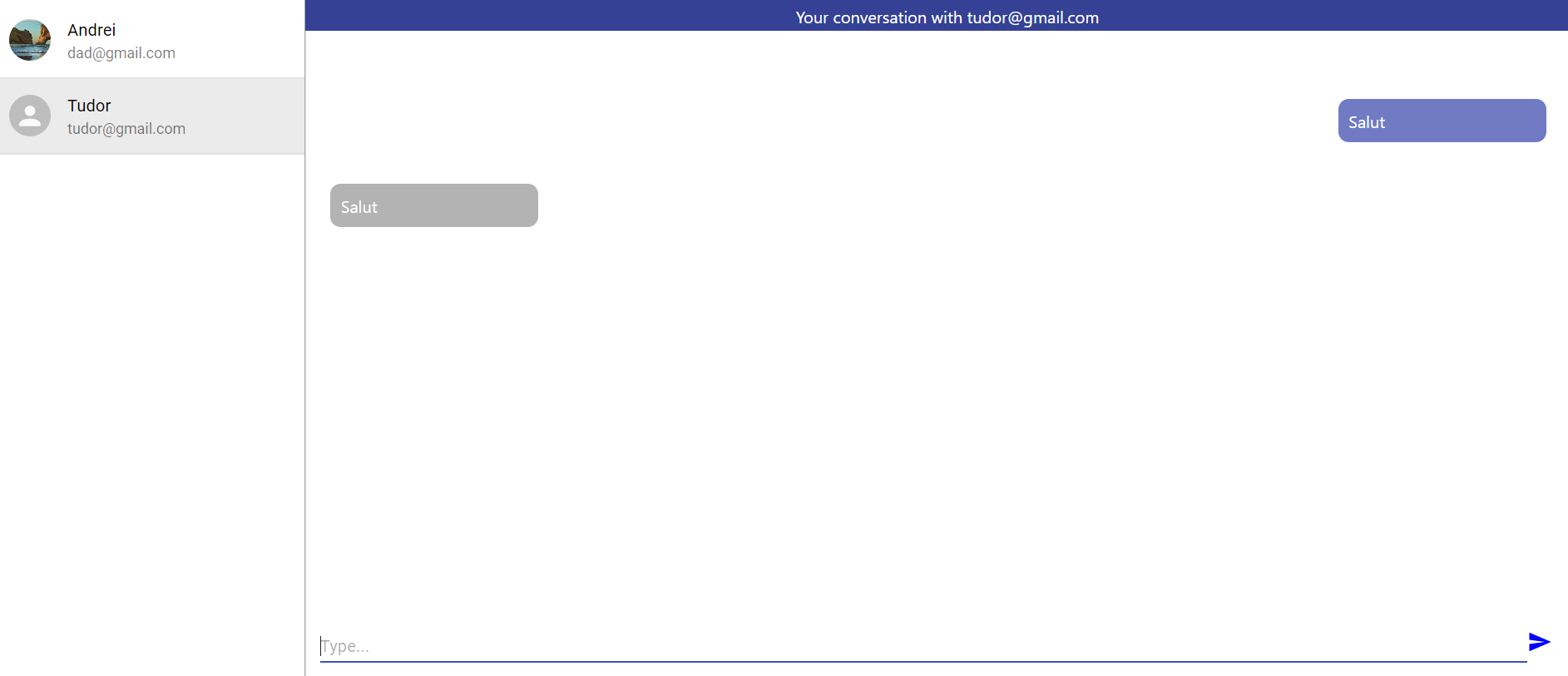
|  |
| --- |
| buildDocKey = (friend) => [this.props.userObj.email, friend].sort().join(':'); |

Astfel, când un mesaj va fi trimis vom putea construi cheia necesară pentru reactualizarea documentului în baza de date Firestore.

|  |
| --- |
| submitMsg = (message) => {  const docKey = this.buildDocKey(this.state.chats[this.state.selectedChat].users.filter(usr => usr !== this.props.userObj.email))  firebase.firestore().collection("chats").doc(docKey).update({  messages: firebase.firestore.FieldValue.arrayUnion({  sender: this.props.userObj.email,  message: message,  timeStamp: Date.now()  }),  receiverHasRead: false  }).then(() => { console.log("succes") }, () => { console.log("fail") })  } |

Componentul de vizualizare a conversației se va actualiza automat, iar mesajele vor fi vizibile instantaneu.

Figura 3.2.8 Conversație



# Concluzii finale

Consider că HobbyMeet este o aplicație utilă deoarece oferă posibilitatea cunoașterii și a comunicării în timp real cu persoane cu interese similare.

## Direcții de viitor

Pe viitor, aplicația ar putea fi portată și pentru tablete sau telefoane și ar putea fi extinsă prin următoarele funcționalități:

• adăugarea unui sistem de rating al utilizatorilor

• creearea de camere de conversații doar pentru un singur hobby

**Bibliografie**

<https://reactjs.org/>

<https://material-ui.com/>

<https://firebase.google.com/docs>

<https://docs.microsoft.com/en-us/dotnet/core/>

<https://pedrodbs.github.io/Aglomera/html/b6f2b623-5f2d-453d-b5ed-d3b9e713cb41.htm>