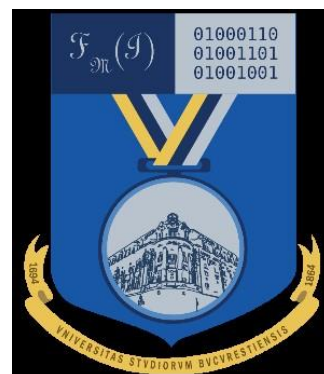


**UNIVERSITATEA DIN BUCUREȘTI**



**FACULTATEA DE MATEMATICĂ  
ȘI INFORMATICĂ**



**SPECIALIZAREA INFORMATICĂ**

**LUCRARE DE LICENȚĂ**

**PLATFORMĂ PENTRU MANAGEMENTUL UNEI LIVEZI DE  
MERI**

Absolvent

**ANDRONIC ALEXANDRA**

Coordonator științific

Lect. Dr. Carmen Chiriță

# Abstract

Aplicația "Măruleț" are ca principal obiectiv introducerea avantajelor tehnologiei în domeniul pomiculturii. Aceasta are o largă arie de acoperire fiind destinată atât administratorilor de livezi de meri, cât și persoanelor din comunitățile acestora care doresc să contribuie la activitățile specifice dintr-o livadă.

Una din funcționalitățile esențiale este implementarea unui sistem pentru programarea de operațiuni, ce depinde de un cumul de factori: temperatură, precipitații, problemă tratată, dar și stadiul de dezvoltare al fructului. Fiecare din acești factori influențează perioada și modul în care trebuie realizată o operațiune. Sistemul pentru programarea operațiunilor din aplicație analizează fiecare din aspectele precizate pentru ca utilizatorul să aleagă perioada și produsele optime în vederea efectuării tratamentelor.

Persoanele cu diferite calificări profesionale pot opta pentru locuri de muncă publicate de către cultivatorii locali. Aceștia stabilesc cerințele ce stau la baza acceptării în vederea angajării, o persoană având posibilitatea aplicării pentru un loc de muncă doar în condițiile specificate de cultivator. În acest fel, procesul de recrutare devine mai ușor de parcurs pentru ambele părți: applicant și angajator. De asemenea, domeniul agriculturii presupune dezvoltarea spiritului de echipă, respectiv a unei comunități bine consolidate, așadar cultivatorii pot lua legătura și dezvolta o varietate de subiecte din domeniul agriculturii folosind forumul integrat în aplicație ce prezintă numeroase funcții, printre care cele de filtrare, sortare, dar și publicarea de poze.

Deoarece fiecare investiție trebuie contorizată, cultivatorii vor avea acces la vizualizarea lor prin diferite grafice intuitive, actualizate automat la fiecare achiziție efectuată. În acest fel, cultivatorii își pot gestiona resursele financiare într-un mod mai eficient.

Prin intermediul funcționalităților descrise, implicit prin digitalizarea treptată a agriculturii, pomicultorul își poate gestiona eficient resursele, prin utilizarea unor tehnologii de actualitate din domeniul agriculturii.

# Abstract

The main objective of the app “Măruleț” is the introduction of the technological advantages in the field of pomiculture. This app is designed for both apple orchard administrators and for people willing to contribute to orchard activities, covering a large area in this domain.

One of the essential uses is a system implementation for programming operations, depending on certain factors: temperature, humidity, the treated issue, and development of the fruit as well. Any of these factors can influence the time and period in which an action must be done. The operation programming system found in the app analyzes each of the mentioned aspects for the user to pick the right period and products in order to facilitate the right treatments.

People with different professional qualifications can filter through the work opportunities offered by local growers. They establish requests that the work force must cover in order to be hired, people having the possibility of applying for a job only if they meet the conditions of the employer. This way, the process of hiring becomes easier to navigate through for both parties: the applicant and the employer.

The domain of agriculture often requires the development of teamwork, of a well-consolidated community, so growers can get in touch and develop a variety of subjects in the field of agriculture using the app’s integrated forum. The forum has many uses, some of which being filtering, sorting and posting photos.

Because every expense must be taken into account, the growers will have access to different intuitive graphs of their spendings which will be automatically refreshed after each buy. This way, they can manage their financial resources more efficiently.

Using the features described above, plus the gradual digitalization of agriculture, the fruiter can sort his resources efficiently using a new and updated technology made for aiding specifically in agriculture.

<b>1. Introducere .....</b>	<b>6</b>
1.1. Prezentare generală .....	6
1.2. Scopul și motivația alegerii temei.....	7
1.2.1 Motivația alegerii temei .....	7
1.2.2. Motivația alegerii tipului de aplicație .....	8
1.3. Starea actuală a aplicațiilor pentru pomicultură .....	9
1.4 Structura lucrării .....	10
<b>2. Prezentarea tehnologiilor folosite .....</b>	<b>11</b>
2.1. Mediul de dezvoltare .....	11
2.2 Frontend.....	12
2.2.1. HTML .....	12
2.2.2. CSS .....	13
2.2.3 Bootstrap .....	14
2.2.4 JavaScript.....	14
2.2.5 ReactJS.....	15
2.2.5.1. DOM Virtual .....	15
2.2.5.2. Sintaxa JSX .....	15
2.2.5.3. Tipuri de componente.....	17
2.2.5.4 Hooks .....	17
2.2.5.4.1 useState .....	18
2.2.5.4.2 useRef .....	18
2.2.5.4.3 useEffect .....	18
2.3 Backend .....	19
2.3.1. Cloud Computing.....	19
2.3.2. Tehnologie fără server .....	21
2.3.3. Firebase .....	21
2.3.3.1 NoSQL .....	22
2.3.3.2 Baza de date din Firebase.....	24
2.3.3.2.1 Operații CRUD în Firestore Database .....	25
<b>3. Implementarea aplicației .....</b>	<b>27</b>
3.1 Structura fișierelor .....	27
3.2 Conectarea la Firebase .....	28
3.3 Autentificarea folosind Firebase.....	30
3.3.1 React Context.....	31

3.3.2 Autentificare și conectare.....	33
3.3.2.1 Componentele de SignUp și Login .....	33
3.3.2.2 Conectare cu Google .....	35
3.3.2.3 Stocarea parolelor în Firebase .....	36
3.4 Definirea rutelor.....	36
3.5 Transmiterea proprietăților între componente .....	38
3.6 Modul de lucru al Firebase integrat în componentele React .....	39
3.6.1 Forum.....	40
3.6.2 Modulul pentru anunțuri de angajare .....	41
3.6.3 Programare operațiuni.....	43
3.7 React-Bootstrap .....	46
3.5.1 Form.....	46
3.5.2 Modal .....	48
3.5.3 Table .....	49
3.8 Open Weather API.....	51
3.9 Aplicație responsive.....	52
<b>4. Ghid de utilizare .....</b>	<b>54</b>
4.1 Înregistrarea și conectarea .....	54
4.2 Completarea profilului.....	54
4.3 Perspectiva cultivatorului .....	55
4.3.1 Profil .....	56
4.3.2 Vremea și programarea operațiunilor .....	56
4.3.3 Anunțuri angajare.....	60
4.3.4 Forum.....	61
4.3.5 Livada mea.....	62
4.3.5.1 Statistici.....	62
4.3.5.2 Facturi.....	63
4.4 Perspectiva angajatului .....	64
4.4.1 Aplicarea pentru un job.....	64
4.4.2 Acceptarea ofertelor .....	65
<b>5. Concluzii .....</b>	<b>67</b>
<b>Bibliografie .....</b>	<b>68</b>
<b>Lista figuri .....</b>	<b>69</b>

# 1. Introducere

## 1.1. Prezentare generală

“Măruleț” este o aplicație web dezvoltată utilizând biblioteca ReactJs împreună cu platforma Firebase. Cele două categorii principale de utilizatori sunt cultivatorul și angajatul ce trebuie să își creeze, pentru început, un cont în scopul utilizării aplicației. Cea mai importantă funcționalitate a contului de cultivator o reprezintă gestionarea operațiunilor care se fac pe tot parcursul anului într-o livadă de meri, programate de către utilizator în funcție de starea vremii, utilajele disponibile la acel moment de timp, dar și calificarea profesională a fiecărui angajat. Tot acesta, poate publica diferite locuri de muncă în funcție de necesitățile fiecărui stadiu de dezvoltare. Angajatul ale cărui caracteristici corespund celor cerute de către angajator în anunțul postat de el, poate aplica pentru acestea în vederea obținerii unui loc de muncă. De asemenea cultivatorul poate ține evidența cheltuielilor la zi, acestea fiind reprezentate prin grafice pentru o înțelegere mai ușoară.

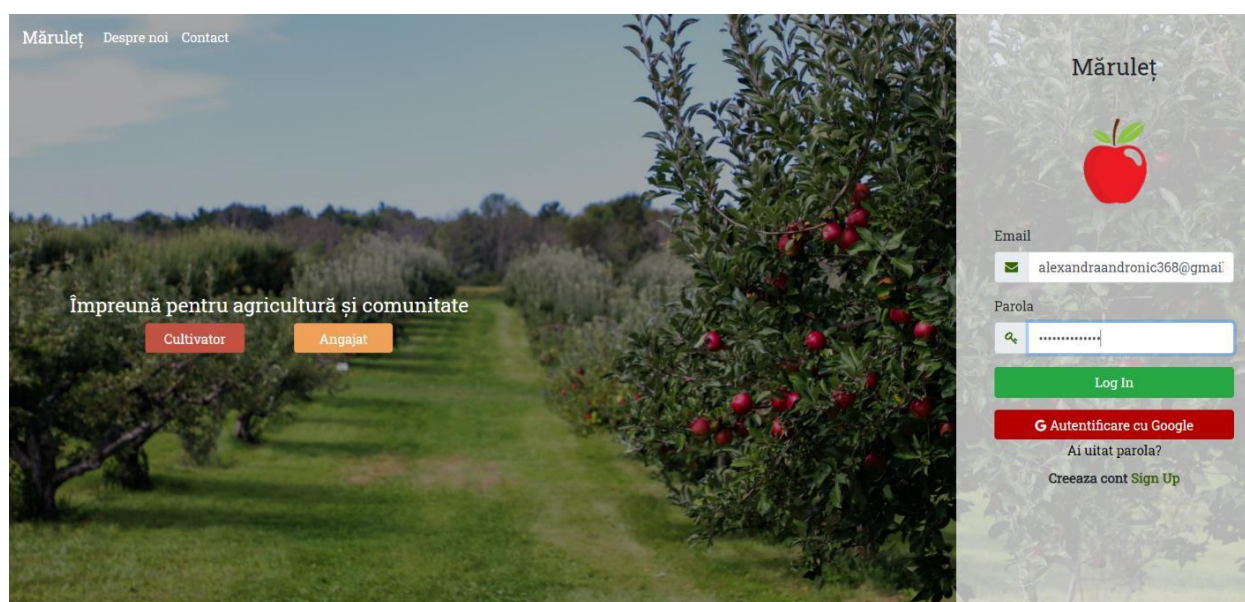


Fig 1.1 – Pagina autentificare din aplicația “Măruleț”

Una dintre componentele cheie ale aplicației este forumul. Prin intermediul acestuia, cultivatorii din diferite locații pot cere și oferi sfaturi, lucru ce în timp poate deveni baza unei comunități de pomicultori din diverse zone.

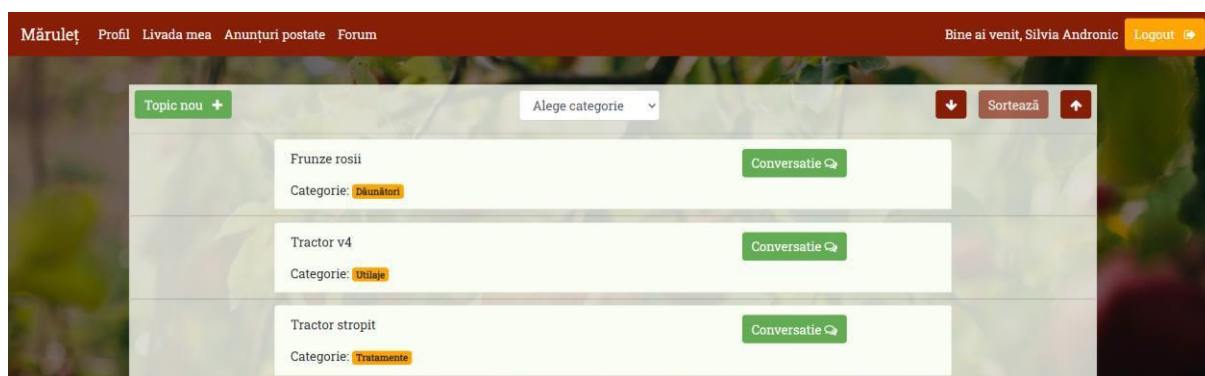


Fig 1.2 – Pagina forum

Consider că un astfel de tip de aplicație ar putea motiva noua generație să se implice mai mult în acest domeniu.

## 1.2. Scopul și motivația alegerii temei

### 1.2.1 Motivația alegerii temei

Agricultura reprezintă una din principalele surse de venit în țara noastră, iar în mediul rural, aceasta este ocupația majoritară a locuitorilor. Există foarte multe resurse care fac prielnică desfășurarea activităților agricole, de la solul fertil, suprafața arabilă extinsă, până la clima favorabilă. Însă, dacă nu este făcută într-un mod cât mai organizat și profesionist, poate aduce mari pierderi în producție, implicit financiare. Subdomeniul ales de mine pentru realizarea lucrării este pomicultura, o ramură a agriculturii care se ocupă de cultura pomilor fructiferi.

Pe lângă aceste aspecte prezentate mai sus, principala motivație care a dus la realizarea acestei aplicații, vine din copilărie. Având în jurul meu mulți pomicultori, în special de meri, am observat nevoia constantă de organizare și gestionare eficientă a resurselor de care dispun pe parcursul unui an.

Fiecare operațiune trebuie făcută la un moment de timp destul de precis, dar și prin mijloacele și cu produsele adecvate. De exemplu, în cazul merilor, tratamentele pentru diferiți dăunători trebuie aplicate cu mare atenție, în cantitatea exactă calculată în funcție de suprafața tratată. Tratamentele aplicate eronat pot afecta fructele, cât și consumul acestora.

Succesul unei culturi bogate constă în capacitatea cultivatorului de a-și utiliza resursele cât mai eficace și de a fi la curent cu noile tehnologii. Însă acest aspect este greu de întreținut atunci când nu există un sistem ajutător care să adune într-un singur loc toate informațiile necesare.

Așadar, prin documentare suplimentară, am implementat în această aplicație principalele funcționalități care din punctul meu de vedere sunt menite să-l ajute pe cultivator în tot procesul: de la starea vremii pe 8 zile pentru a ști când și ce tratament trebuie aplicat, până la centralizarea cheltuielilor obținute din achiziționarea de utilaje, substanțe și pomi, dar și forța de muncă de care are nevoie un agricultor pentru toate operațiunile din parcursul unui an.

### 1.2.2. Motivația alegerii tipului de aplicație

La baza alegerii construirii unei aplicații web, stă interesul propriu pentru acest domeniu, dar și cele care țin de structura acestor aplicații și al avantajelor aduse.

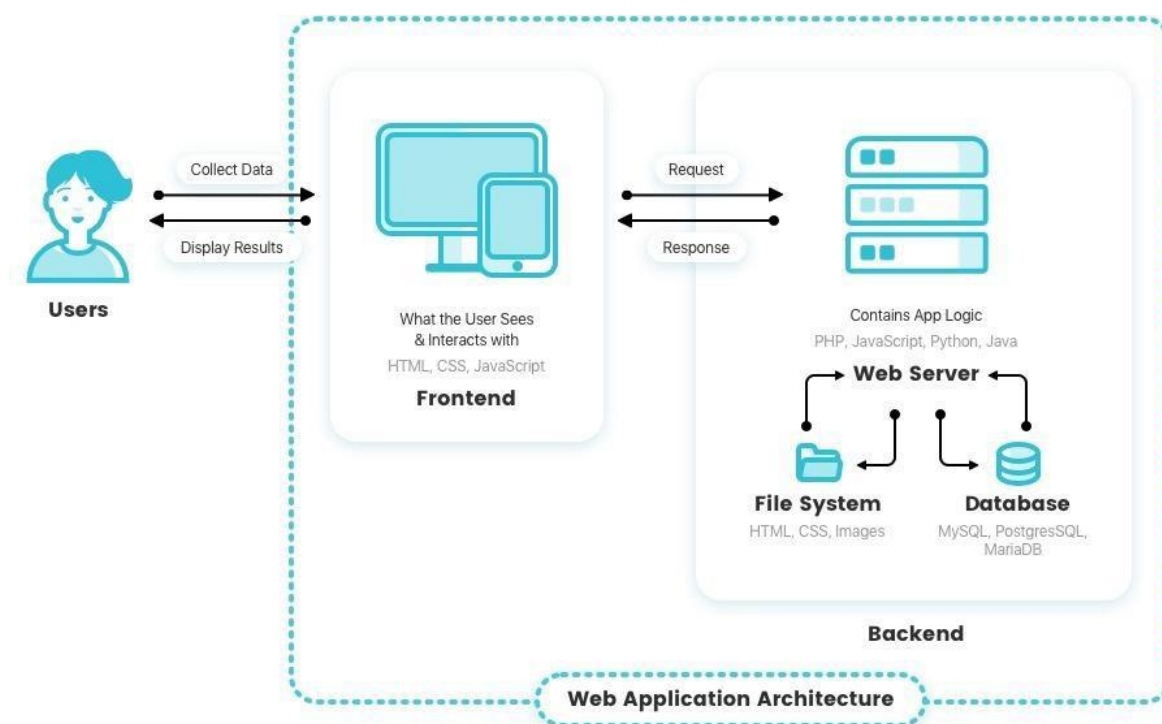


Fig 1.3 – Arhitectura unei aplicații web [1]

- În primul rând acestea sunt independente de sistemul de operare, putând fi accesate de pe orice browser, singura condiție fiind cea a unei conexiuni la internet.
- Spre deosebire de aplicațiile instalate pe dispozitivul utilizatorului care necesită actualizări la fiecare modificare, în cazul aplicațiilor web, modificările sunt vizibile utilizatorului fără ca acesta să mai facă acest pas.
- Indiferent de arhitectura acestora, utilizatorul nu trebuie să dispună de multe resurse pentru a putea accesa și utiliza aplicația.



- Spre deosebire de aplicațiile pentru mobil, care pot fi accesate doar de pe dispozitive de acest tip, aplicațiile web își adaptează aspectul în funcție de dimensiunea dispozitivului.

### **1.3. Starea actuală a aplicațiilor pentru pomicultură**

În domeniul pomiculturii, în România, încă nu se pune accent pe acest proces de centralizare al operațiunilor folosind tehnologia. După o scurtă cercetare, am observat că nu există nicio aplicație de acest tip în limba română. Printre aplicațiile ce au constituit o mică sursă de inspirație pentru această lucrare de licență se numără: Hectre și Farmable. Ambele aplicații sunt atât web cât și pentru mobil.

Hectre este o platformă cu ajutorul căreia cultivatorul poate să țină evidența culturilor sale, dar de asemenea să adauge anumite task-uri în vederea efectuării de operațiuni (irigații, tratamente, fertilizări). Dezvoltatorii nu oferă informații cu privire la costul aplicației. Un potențial utilizator este nevoit să parcurgă un proces foarte complex care implică programarea la o vizualizare a funcționalităților aplicației. Procesul propriu-zis de a-ți face cont în această aplicație este unul destul de complex. Nu dispune de un flux obișnuit pe care îl au toate aplicațiile care necesită un cont pentru a fi utilizate. Consider că cele menționate mai sus constituie un dezavantaj al aplicației Hectre. [2]

Farmable este de asemenea o aplicație web care are ca și caracteristică specială funcționalitatea de trasare a unui poligon ce reprezintă suprafața livezii. Astfel se calculează exact suprafața cultivată. [3]

Un dezavantaj major al acestor aplicații este acela că nu sunt și în limba română.

Având în vedere aspectele prezentate anterior, am decis ca aplicația să fie în limba română pentru faza inițială a dezvoltării pentru a putea fi ușor de utilizat de către orice pomicultor român. Programul tehnologic este adaptat în funcție de un cumul de factori specifici țării noastre fapt ce a stat la baza alegerii dezvoltării aplicației în limba română.

Modulul pentru găsirea locurilor de muncă publicate de cultivatori a venit ca o dorință de a crea o comunitate cât mai legată în această sferă a agriculturii formată din cât mai mulți oameni care contribuie la aceasta.

## 1.4 Structura lucrării

Această documentație este structurată în 5 capitole fiecare prezentând aspecte importante ale aplicației.

Primul capitol, **Introducerea**, descrie minimal funcționalitățile proiectului, dar și motivația ce a dus la dezvoltarea acestui tip de aplicație. Pentru a susține motivația ce a stat la baza alegerii temei, există și o prezentare a aplicațiilor existente deja în acest domeniu.

Al doilea capitol, cel al **Tehnologiilor utilizate**, prezintă câteva noțiuni generale despre fiecare. Accentul se pune mai mult pe dezvoltarea unor particularități care au dus la alegerea lor în implementarea aplicației.

Al treilea capitol explică implementarea propriu-zisă a aplicației folosind tehnologiile prezentate în capitolul doi.

În al patrulea capitol se regăsește un ghid de utilizare al aplicației, prezentând fluxul tuturor funcționalităților pentru ambele perspective (Cultivator și Angajat). Prin intermediul imaginilor adăugate, tutorialul este mult mai sugestiv.

Ultimul capitol cuprinde dezvoltările ulterioare ce se doresc a fi introduse în aplicație cât și câteva concluzii cu privire la modul de implementare și nu numai.

## 2. Prezentarea tehnologiilor folosite

În primul rând voi prezenta mediul de dezvoltare folosit, după care tehnologiile utilizate în dezvoltarea aplicației împreună cu motivația și caracteristicile lor.

### 2.1. Mediul de dezvoltare

**Visual Studio Code** IDE (Integrated Development Environment) oferit de Microsoft este o platformă care rulează pe Windows, Mac și Linux. Ecosistemul acesteia permite dezvoltarea de aplicații în numeroase limbaje de programare, câteva dintre ele fiind C++, C#, Java și Javascript. Dispune de o multitudine de funcționalități care ajută la scrierea de cod rapid și lizibil. Dintre acestea cele folosite de mine sunt:

- **Intelli-Sense** ajută la completarea inteligentă a codului doar prin tastarea literelor din începutul cuvântului. La baza acestei funcționalități stă analiza realizată pe codul sursă ce va genera o lista cu sugestii.
- Anumite secvențe de cod devin repetitive în funcție de limbajul pe care îl folosim. **Code Snippets** sunt anumite șabloane de cod care pot scurta timpul în care scriem aceste secvențe. Un caz reprezentativ este cel al componentelor din React, de tipul funcțiilor în cazul aplicației de față. Literele *rfc* vor genera automat scheletul unei componente funcționale. Prin folosirea acestei funcționalități se poate ajunge la scrierea de cod într-un mod foarte fluid, rapid și cu cât mai puține greșeli de sintaxă.

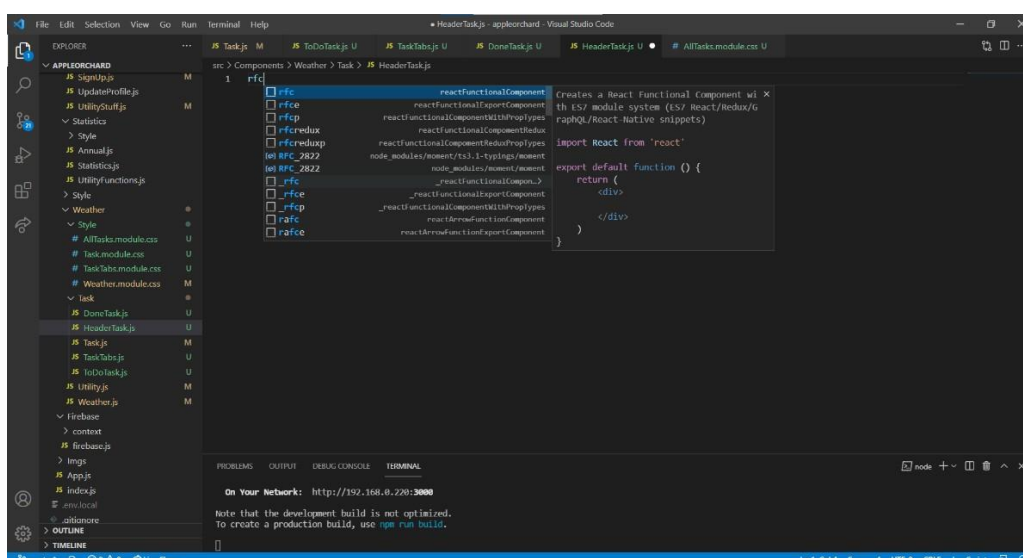


Fig 2.1 – Generarea unei componente de tipul unei funcții în React cu ajutorul Code Snippets

- Conectarea la Github (Version control) pentru menținerea modificărilor într-un mod cât mai organizat și eficient. Este indicat ca orice modificare să fie adăugată, astfel că orice funcționare necorespunzătoare poate să fie rezolvată prin revenirea la o versiune mai veche a proiectului.

## 2.2 Frontend

Partea de front-end, după cum îi spune și numele, este interacțiunea directă cu utilizatorul. Este de preferat ca aceasta să fie una intuitivă, cu un aspect plăcut și ușor de utilizat de toată lumea. În realizarea acestei părți am folosit limbajul Javascript, mai exact biblioteca React.js împreună cu sintaxa JSX, iar pentru stilizare CSS și Bootstrap.

Pentru a prezenta structura și caracteristicile bibliotecii ReactJS, voi descrie inițial câteva noțiuni despre HTML, CSS, Bootstrap și Javascript.

### 2.2.1. HTML

**HTML** (Hypertext Markup Language) este un limbaj utilizat în crearea paginilor web. Structura acestuia constă în tag-uri pentru diferite elemente ce se pot afla într-o pagină web (ex: titlu, text, paragrafe, imagini). Tag-urile au un set de attribute ce ne permit să le edităm aspectul, cum ar fi: culoare, dimensiune, aliniere. HTML5 este ultima versiune, dar și cea care a venit cu cele mai multe noutăți și îmbunătățiri din punctul de vedere al structurii unei pagini web. S-au introdus tag-uri ce ajută la organizarea mai bună în pagină a elementelor. Avem <header> pentru partea de sus a paginii, ce conține în general numele aplicației și meniul, <section> pentru a structura informația în corpul paginii, <footer> pentru informații adiționale care sunt plasate în partea de jos a unei pagini. De asemenea, și în implementarea acestei aplicații am încercat să păstrez un schelet al fiecărei paginii format din aceste elemente. Pentru utilizator va fi mult mai ușor să înțeleagă cum este informația distribuită în pagină dacă se va folosi această structură.

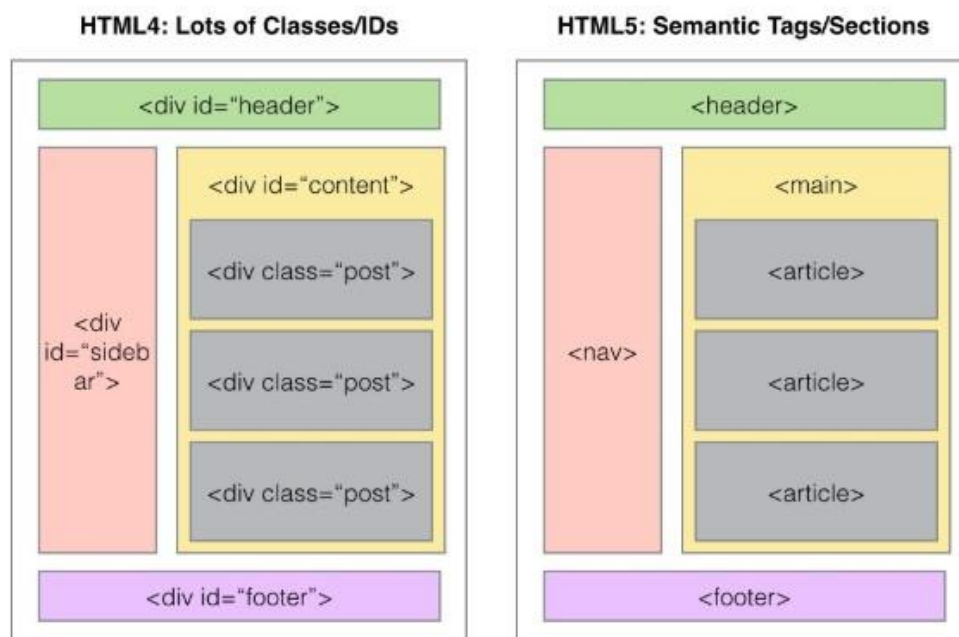


Fig 2.2 – Structura unei pagini din HTML4 vs HTML5 [4]

### 2.2.2. CSS

**CSS** (Cascading Style Sheets) este folosit pentru a stiliza și alinia conținutul de pe paginile web. Acesta se aplică doar pe cod **HTML**, pentru a-l formata. Există 3 moduri în care se poate scrie cod CSS. Acestea sunt:

- Fișier salvat cu extensia .css și adăugat printr-o referință (`<link />`) în codul HTML pe care dorim să îl stilizăm. Fișierele pot fi astfel reutilizate. Am ales această abordare datorită acestui fapt.
- Cod intern, scris între tag-urile `<style></style>` din interiorul lui `<head>`. Însă este de preferat folosirea acestui mod de integrare al stilizării doar în cazul în care se dorește aplicarea pe toată pagina și de asemenea presupune scrierea unei cantități mici de cod.
- Codul introdus direct în tag-urile html prin atributul style. Am utilizat și această modalitate de stilizare în cazul în care doream să modific doar un singur atribut al unui element și această modificare nu se repeta în cadrul fișierului respectiv.

CSS dispune de câteva avantaje care îl fac să fie folosit în continuare:

- Cu ajutorul lui pot fi construite și aplicații pentru alte dispozitive, cu un format diferit de afișare al design-ului, cum ar fi: tabletă, telefon.
- Se evită dublarea codului prin refolosirea fișierelor.

### 2.2.3 Bootstrap

**Bootstrap** este o bibliotecă lansată în anul 2011 de Twitter și este unul din cele mai folosite framework-uri pentru stilizarea paginilor web pentru toate tipurile de ecrane. Este open-source și are mulți dezvoltatori care contribuie pentru noi componente și îmbunătățirea ei. Această bibliotecă este o combinație între HTML, CSS, dar și JavaScript ceea ce duce la construirea de site-uri cu un aspect plăcut pe orice tip de ecran fără a scrie foarte mult cod. În documentația oficială se găsesc toate aceste clase pentru stilizarea elementelor folosind bootstrap. [5]

**React-Bootstrap** este de asemenea o bibliotecă compusă din React și Bootstrap. Diferența dintre aceasta și Bootstrap nativ este aceea că fiecare componentă este scrisă ca una din React, astfel nu mai este nevoie de dependențele jQuery. Componentele sunt de asemenea responsive și cantitatea de cod pentru a scrie o astfel de componentă este considerabil mai mică în comparație cu una scrisă în bootstrap. Am utilizat React-Bootstrap pentru a scrie diferite componente care se adaptează pentru toate tipurile de ecrane, dar și Bootstrap pentru a așeza informația în pagină mult mai ușor. React-Bootstrap a fost foarte folosit pentru diferite componente precum Form și Modal pe care le voi prezenta în capitolul dedicat implementării. [6]

### 2.2.4 JavaScript

**JavaScript** (prescurtat JS) este un limbaj de programare care introduce dinamicitate elementelor din paginile web. Poate fi utilizat în partea de front-end, respectiv back-end. În timp ce HTML și CSS dau structura statică a unei pagini web, Javascript oferă o interacțiune mult mai prietenoasă cu utilizatorul. O caracteristică foarte importantă o reprezintă faptul că funcțiile sunt de asemenea obiecte și pot fi transmise ca argumente altor funcții. Obiectele de tip **JSON** sunt proprii limbajului JavaScript ce au o structură foarte versatilă care permite transmiterea de date cu un format complex prin intermediul unui server. În continuare voi

prezenta biblioteca din Javascript folosită pentru a dezvolta aplicația ce constituie subiectul acestei documentații și anume ReactJS.

## **2.2.5 ReactJS**

**ReactJS** este o bibliotecă open-source din Javascript folosită pentru a construi interfețe de utilizator (en: user interface - UI). Aceasta a fost dezvoltată de către Facebook. În continuare are o comunitate foarte mare de dezvoltatori și oameni dornici să adauge cât mai multe funcționalități acestei biblioteci. Arhitectura aplicațiilor dezvoltate în React este de tipul “aplicație cu o singură pagină” (single-page-application). La baza acestui principiu stă faptul că orice componentă nouă pe care utilizatorul o vede, se încarcă dinamic prin supra-scrierea paginii anterioare, spre deosebire de alte tehnologii care aduc componenta în întregime de pe server, îngreunând încărcarea conținutului pe pagină. [7]

### **2.2.5.1. DOM Virtual**

DOM sau „Document Object Model” reprezintă structura HTML din aplicația noastră. Acesta este organizat sub forma unui arbore în care fiecare nod este un element de tip HTML. Cu ajutorul acestei ierarhii, dezvoltatorul poate modifica conținutul direct din cod JavaScript. La fiecare actualizare din acest arbore, modificarea va fi reflectată și utilizatorului, astfel că aceste operații devin costisitoare atunci când structura aplicației este foarte complexă și fiecare nod din arbore are la rândul său mai mulți descendenți.

DOM-ul Virtual din React reprezintă o copie după actualul DOM al aplicației. Costurile modificărilor din acest DOM Virtual sunt mult mai mici deoarece acestea nu sunt vizibile și utilizatorului. La fiecare actualizare apărută în aplicație se generează un DOM Virtual. Acest DOM creat va fi comparat cu cel anterior pentru a verifica ce elemente sunt sau nu modificate. Așadar, doar elementele care au suferit modificări vor fi din nou actualizate în DOM și afișate utilizatorului conținând actualizările realizate. Acest proces de comparare al DOM-urilor Virtuale poartă numele de „diffing”. Aceste costuri sunt mult diminuate prin intermediul DOM-ului Virtual și al operației de „diffing”. [8]

### **2.2.5.2. Sintaxa JSX**

La baza acestei biblioteci stă sintaxa JSX, o extensie a limbajului JavaScript. În JSX, elemente din HTML și codul JavaScript pot fi scrise împreună.

```
const helloText = <h1>Hello World!</h1>;
```

Sintaxă JSX

În secvența de mai sus este cod JSX, sintaxă folosită în React, însă problema cu aceasta este că nu poate fi recunoscută de către browser pentru că nu este cod JavaScript valid. Pentru ca motorul de căutare folosit să recunoască această sintaxă și să afișeze conținutul corespunzător, trebuie ca la compilare, codul să fie convertit în limbaj JavaScript valid folosind Babel. Acesta este un compilator pentru JavaScript care convertește orice versiune sau sintaxă de JavaScript în cod JavaScript ES5 care poate fi rulat pe orice browser. [9]

```
class Grower extends React.Component {  
  render() {  
    return <h1>This is a React component</h1>;  
  }  
}  
  
ReactDOM.render(<Grower />, document.getElementById('root'));
```

Fig 2.3 - Exemplu de componentă în React

Pentru ca textul din interiorul etichetelor să fie afișate pe ecran, JSX transformă acest cod la compilare în felul următor:

```
class Grower extends React.Component {  
  render() {  
    return React.createElement("h1", null, "This is a React component");  
  }  
}
```

Fig 2.4 - Cod executat de Babel pentru afișarea pe ecran

```
React.createElement(component, props, ...children)
```

Fig 2.5 – Definiția funcției `createElement` [10]

În exemplul din Figura 2.6 primul element, corespunzător parametrului **type** este `h1` deoarece acesta este elementul HTML pe care am dorit să-l afișăm în prima secvență de cod, deci acest parametru poate fi de tipul tag HTML sau chiar altă componentă reutilizată. Parametrul **props** reprezintă atributele pe care le are elementul, în cazul de față ar fi putut fi o stilizare CSS pentru tag-ul `h1`, dar cum aceasta nu există, parametrul va fi `null`. Ultimul



parametru, **children** reprezintă conținutul ce va fi văzut de către utilizator, de la un tag HTML până la o altă componentă, formându-se astfel structura arborescentă și anume DOM-ul prezentat în subcapitolul anterior.

JSX face ca scrierea de cod în React să fie mult mai ușoară prin similaritatea sa cu HTML. Pentru a putea introduce cod JavaScript în această sintaxă și a crea conținut dinamic pe pagină, este nevoie să introducem codul JS între acolade `{ }`. În acel moment, JSX interpretează ceea ce am scris ca și cod JavaScript. Însă există câteva elemente invalide ce nu se pot scrie într-o expresie JSX, cum ar fi: instrucțiuni `for` și `while`, declarare de variabile și funcții.

### 2.2.5.3. Tipuri de componente

În React se disting două modalități în care pot fi construite componentele: **clase** și **funcții**. Principalul avantaj al acestei biblioteci constă în faptul că aceste componente, indiferent de structura lor, pot fi reutilizate oriunde în aplicație.

Componentele de tip clasă merg pe același principiu al claselor din programarea orientată pe obiecte. Aceasta este formată la rândul ei din mai multe funcții pentru a adăuga funcționalitate componentei. Conținutul va fi afișat pe pagină doar în cazul în care acesta este în interiorul metodei `render()`. Pentru a crea o componentă de acest tip este necesară moștenirea din clasa **React.Component** care conține metoda `render()`.

Componentele funcționale sunt cel mai ușor de utilizat fiind funcțiile native din JavaScript. Spre deosebire de clase, nu au nevoie de metoda **`render()`** pentru a afișa conținutul.

În versiunea de React 16.8 s-au introdus hook-urile. Dacă până în acel moment, singura modalitate de a adăuga dinamicitate unei componente React, era ca aceasta să fie de tipul unei clase, această noutate rezolvă problema. Acestea pot fi folosite doar în componentele de tipul funcțiilor. În continuare voi prezenta funcțiile de tip hook folosite în dezvoltarea aplicației, la care voi face referire și în capitolul de implementare.

### 2.2.5.4 Hooks

Componentele funcționale în React nu pot reține starea curentă precum clasele, iar pentru a adăuga această proprietate, am folosit o categorie de funcții speciale, numite hook. Acestea

ajută la scrierea de cod mult mai ordonat și ușor de citit. Sunt similare funcțiilor din JavaScript însă există două reguli care trebuie respectate cu strictețe pentru a nu genera erori în cod.

1. Hook-urile nu se apelează din instrucțiuni *if*, *while* sau *for*.
2. Acestea se apelează doar din componente de tipul funcțiilor.

#### 2.2.5.4.1 useState

Hook-ul **useState** este cel mai des folosit deoarece rolul acestuia este să reîncarce din nou componenta în momentul în care starea acesteia se schimbă, astfel că utilizatorul va vedea întotdeauna datele actualizate. [11]

**useState** returnează valoarea curentă a stării și o funcție de actualizare. Pentru a reține aceste valori se folosește ceea ce în JavaScript poartă numele de destructurare de vectori.

```
const [task, setTask] = useState([]);
```

Exemplu declarare pentru hook useState

Având în vedere exemplul de mai sus, la prima încărcare a paginii, variabila **task** va fi un vector fără niciun element. La fiecare actualizare a acestui vector folosind funcția **setTask**, pagina va fi încărcată din nou, componenta reținând starea nouă a vectorului **task**. Folosind destructurarea, valoarea curentă va fi stocată în variabila **task**, iar funcția de actualizare în **setTask**. Există o convenție pentru denumirea acesteia, având prefixul **set**.

#### 2.2.5.4.2 useRef

Adesea sunt necesare modificări fără a fi nevoie de reîncărcarea componentei în pagină, astfel că **useState** nu este potrivit pentru acest scenariu.

```
const refEmail = useRef();
```

Exemplu declarare pentru hook useRef

Valoarea lui **refEmail** poate fi accesată prin proprietatea referinței **.current**.

Spre deosebire de **useState**, care produce o reîncărcare a paginii, **useRef**, nu face acest lucru. Poate fi folosit, de exemplu, pentru a reține datele completate într-un formular, deoarece la fiecare schimbare a input-ului, pagina nu va fi reîncărcată din nou. [12]

#### 2.2.5.4.3 useEffect

Un alt hook utilizat des în aplicații este **useEffect**, fiind în strânsă legătură cu **useState**. În interiorul acestei funcții putem adăuga evenimente pentru care dorim apelarea după fiecare încărcare a paginii.

Există însă cazuri în care rularea după fiecare reîncărcare poate produce probleme de performanță. Ca soluție la această problemă, putem folosi vectorul de dependențe ce este un al doilea argument (opțional) pentru **useEffect**. Dacă dorim ca un efect să se întâmple doar o dată, este suficient ca vectorul de dependențe să nu conțină niciun element. Dacă reîncărcarea componentei trebuie să se execute doar după modificarea unei anumite stări, acea variabilă va fi conținută de vectorul de dependențe și reîncărcarea se va face doar după ce starea acestuia s-a schimbat. [13]

## 2.3 Backend

Modul de dezvoltare cel mai întâlnit este modelul client-server, care are la bază 2 părți: partea de client (client side) și cea de server (server side). Clientul comunică cu serverul prin intermediul unei conexiuni la internet, însă acest lucru nu mai este necesar dacă ambele entități se află pe aceeași mașină.

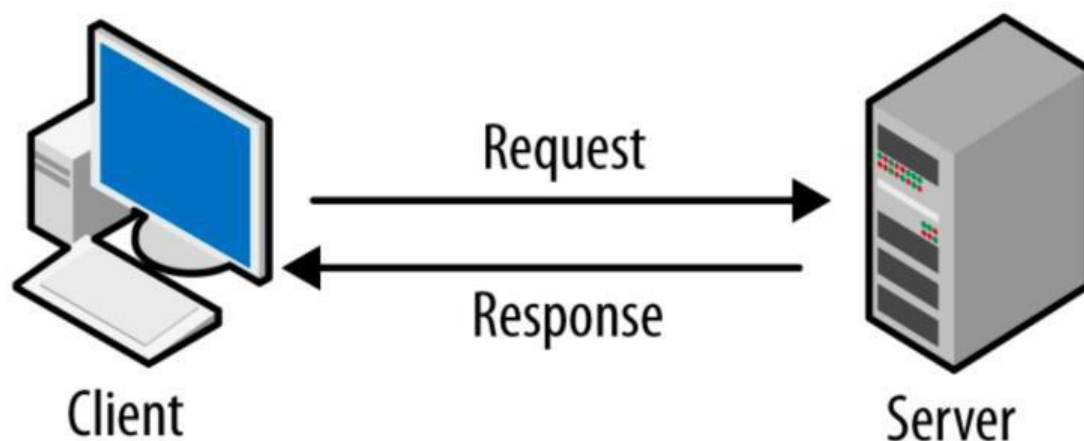


Fig 2.6 Arhitectură de tip Client – Server [14]

### 2.3.1. Cloud Computing

Datorită cursului opțional de **Cloud Computing** făcut în facultate, am ales ca pentru partea de backend să nu folosesc serverul tradițional, ci să aleg modelul de aplicație fără server (**Serverless**). Însă pentru a prezenta această noțiune de aplicație fără server, va trebui să explic ce reprezintă domeniul Cloud Computing.

Termenul de Cloud Computing reprezintă livrarea de servicii software la cerere de către dezvoltatorii și administratorii de aplicații, de la spații de stocare suplimentare până la putere de procesare.

Având în vedere amploarea pe care o are lumea tehnologiei în ultimul timp, resursele de care au nevoie companiile devin din ce în ce mai multe, implicit costisitoare. Administrarea acestora ocupă foarte mult timp, în care accentul s-ar putea pune pe produsul în sine și pe dezvoltarea lui conform cerințelor. Astfel s-a dezvoltat acest serviciu online de Cloud Computing ce dispune de funcționalități care nu necesită direct intervenția umană pentru a funcționa. În mod uzual, companiile folosesc tipul “pay as you go” care funcționează pe principiul “ plătești cât folosești” pentru a se evita risipa de resurse, dar și de finanțe. Astfel că, pentru o anumită sumă de bani, acest serviciu este oferit companiilor și nu numai.

Există trei mai categorii de Cloud: **SaaS**, **Paas**, **Iaas**. Voi prezenta câteva caracteristici despre fiecare:



Fig 2.7 – Tipurile de Cloud [15]

**SaaS** (Software-as-a-Service) reprezintă cel mai cuprinzător tip de Cloud. Aduce o întreagă aplicație care este gestionată de furnizor, astfel că dezvoltatorii nu trebuie să se mai preocupe de arhitectură sau mod de funcționare. Este necesară doar înțelegerea funcționalităților oferite, cât și o conexiune la internet. Folosim acest tip de Cloud în fiecare zi, de la serviciul web pentru mail (Outlook, Gmail) până la stocarea informațiilor într-o aplicație de tipul Dropbox. SaaS nu necesită instalare sau actualizare, acest fapt reprezentând un avantaj. [15]

**PaaS** (Platform-as-a-Service) are ca și caracteristică principală cea de deploy al aplicațiilor. Acesta oferă un cadru dezvoltatorilor pentru a-și construi și utiliza aplicația. Un exemplu pentru acest tip de cloud este Heroku. [15]

**IaaS** este tipul de cloud ce se aseamănă în mare parte cu PaaS, diferența fiind că se poate face inclusiv instalarea unui sistem de operare. [15]

Având în vedere că am explicat principalele tipuri de Cloud Computing și particularități ale acestora, voi descrie în continuare derivata numită Serverless Computing.

### **2.3.2. Tehnologie fără server**

La baza acesteia stă un furnizor de servere în Cloud. Responsabilitatea gestionării serverelor dispare în totalitate, astfel că atenția și timpul se pot îndrepta exclusiv spre partea de dezvoltare și cea de logică a aplicației. Aceste servere vor fi folosite doar la cerere, deci utilizatorii vor plăti pentru aceste servicii doar în funcție de cantitatea de resurse pe care au utilizat-o. Putem face o corespondență cu energia electrică sau consumul de gaz. Folosim aceste resurse doar când avem nevoie de ele și suntem taxați în funcție de consumul realizat. Acesta este și principiul pe care se bazează tehnologia fără server, toate facilitățile sunt folosite doar când avem nevoie de ele, evitându-se astfel risipa.

### **2.3.3. Firebase**

Firebase este un serviciu de cloud dezvoltat de Google pentru a crea aplicații mobile și web. [16]

Am ales ca în dezvoltarea aplicației să mă implic mai mult pe partea de implementare de funcționalități necesare utilizatorului și pe nevoile acestuia, astfel că principiul de serverless a fost acoperit în cazul de față de serviciile oferite de producătorul Firebase. Acesta oferă de la

autentificare prin diferite moduri, baze de date, stocare de fișiere până la diferite pachete pentru inteligență artificială.

Există două variante de folosire a serviciilor de Firebase în funcție de necesitățile fiecărui dezvoltator:

1. Prima variantă este cea gratis, Spark. Pune la dispoziție majoritatea funcționalităților cum ar fi: autentificarea prin mail cât și cea prin anumite servicii (Gmail, Facebook, Instagram), spațiu de stocare și baze de date. Având în vedere că este o variantă gratis, resursele oferite de furnizor sunt mult reduse. Se poate folosi în aplicații aflate în faza de dezvoltare când nevoia de resurse și spațiu este destul de mică. Doar 100 de utilizatori pot fi conectați simultan în acest plan. De asemenea și în aplicația prezentată am folosit pentru faza de dezvoltare tot acest plan Spark. [17]
2. Al doilea plan, Blaze, este tipul de serviciu “plătești cât consumi” (pay-as-you-go). Așadar, utilizatorul va plăti direct proporțional cu resursele folosite. [17]

Principalele funcționalități pe care le-am utilizat și eu în dezvoltarea aplicației sunt:

1. **Autentificarea** ce se poate face în două moduri: cea obișnuită în care utilizatorul își face un cont folosind adresa de mail, dar și prin intermediul rețelelor precum: Gmail, Facebook, Instagram. Sunt deja implementate toate funcționalitățile importante precum: recuperare și schimbare de parolă, cât și confirmare adresă de mail.
2. **Baza de date** oferită de Firebase este de tip NoSQL. În acest format datele sunt salvate în baza de date sub forma unor obiecte JSON. JSON (JavaScript Object Notation) este un tip de date de dimensiuni mici, folosit în special pentru transferul de date. Cu ajutorul acestuia se pot stoca date cu un format complex.

### 2.3.3.1 NoSQL

Din punctul de vedere al bazelor de date, sunt cunoscute două modalități de utilizare: relaționale și nerelaționale. Deși bazele de date relaționale sunt cele mai cunoscute și studiate, în ultimul timp, cele nerelaționale au fost incluse mai mult în dezvoltare, mai ales datorită Cloud Computingului și cantităților mari de date și variate ce se voiau a fi stocate într-o baza de date. În continuare doresc să prezint avantajele și dezavantajele celor două tipuri de reprezentare.

Niciuna din cele două baze de date nu reprezintă un înlocuitor pentru cealaltă. În funcție de tipul de aplicație dezvoltat, trebuie să alegem reprezentarea cea mai potrivită pentru a stoca și accesa datele într-un mod cât mai eficient.

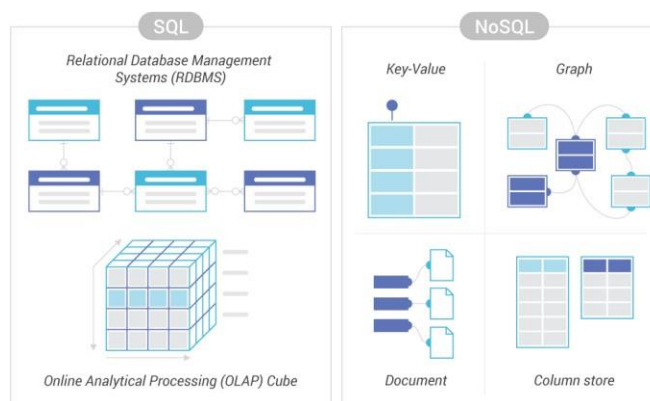


Fig 2.8 – Baze de date relaționale vs non-relaționale [18]

În bazele de tip SQL, datele sunt reprezentate sub formă de tabele fiind legate între ele prin relații de diferite tipuri. Fiecare linie va reține informații relevante pentru obiectele care sunt stocate în tabelul respectiv. Un dezavantaj al acestor reprezentări o constituie structura unui tabel. De exemplu, dacă o coloană este de tip număr, nu putem salva o valoare care este un șir de caractere. Acest lucru este posibil într-o baza de date NoSQL. În două înregistrări diferite, aceeași coloană poate conține date de tipuri diferite. Însă, în același timp, acest fapt poate aduce și inconsistențe în baza de date.

În ceea ce privește constrângerile, bazele de tip SQL sunt foarte rigide. Nu putem adăuga date în tabele până nu definim constrângeri cum ar fi: constrângeri de cheie primară. În schimb, în bazele NoSQL, datele se adaugă fără a defini aceste constrângeri. Nu este nevoie nici de definirea colecției înainte de inserarea primului element, cum se procedează cu tabelele din SQL. La prima inserare a unui element într-o colecție, se va crea colecția precizată împreună cu un document în care se va adăuga elementul respectiv. Așadar, NoSQL este potrivită pentru aplicațiile în care structura nu este determinată de la început din cauza complexității acesteia.

```
refCurrentUser
    .add({
      product: product,
      price: newPrice,
      quantity: newQuantity,
      month: month,
```

```

        year: year,
        currency: currency
    })
    .catch((err) => {
        console.log(err);
    });

```

Adăugare înregistrare în baza de date NoSQL din Firebase

### 2.3.3.2 Baza de date din Firebase

În Firebase se regăsesc două tipuri de baze de date: Firestore Database și Realtime Database. Ambele tipuri o au ca structură pe cea de tip NoSQL. În implementarea aplicației am folosit Firestore Database și voi argumenta alegerea acestuia prin comparație cu Realtime Database. Primul aspect care m-a determinat să aleg Firestore Database este modelul de date. Realtime Database stochează datele sub forma unui obiect JSON obișnuit, o structură care poate fi înțeleasă ușor pentru date care nu sunt complexe. Însă în momentul în care nivelele cresc, parcurgerea obiectului poate deveni dificilă. În Firestore Database datele sunt stocate în colecții ce sunt construite din documente, care în fond sunt formate din perechi cheie-valoare. De asemenea, fiecare colecție poate conține la rândul său o altă colecție, formându-se structura de arbore.

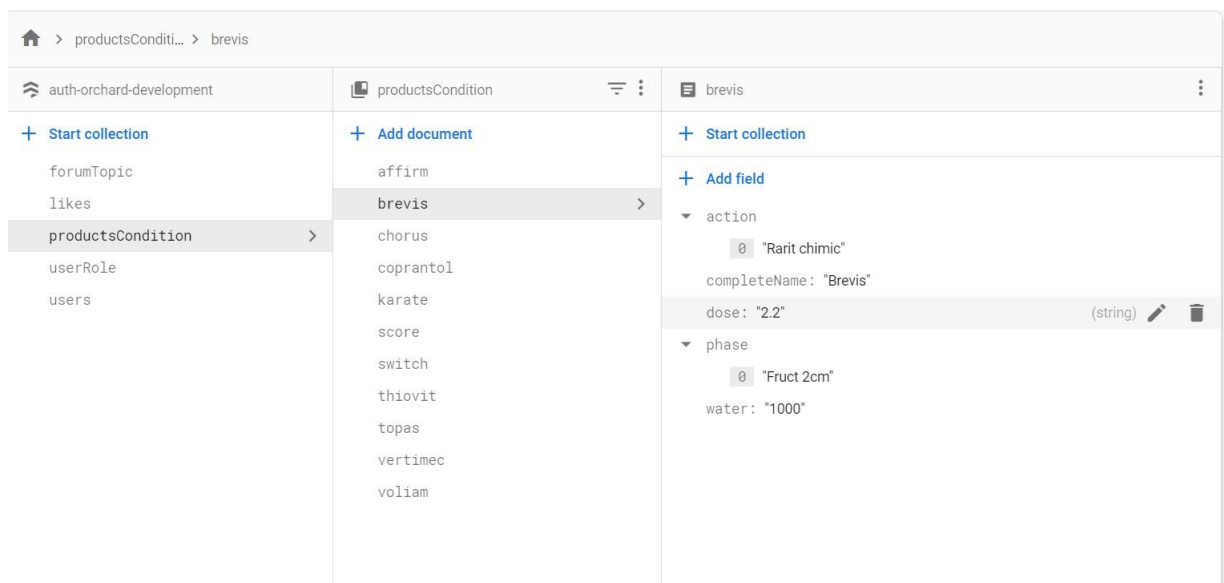


Fig 2.9 – Exemplu colecție și documente din Firebase Database



Un alt plus al Firestore Database este cel al interogărilor în baza de date. Se pot face interogări în funcție de mai multe câmpuri, ceea ce în Realtime Database nu este posibil dintr-o singură comandă. De asemenea, se poate accesa un document direct fără a mai fi nevoie de parcurgerea altor nivele din colecție.

```
const refAcceptedOffer = firebase.firestore().collection("users").doc(currentUser.uid).collection("onHold").where('status', '==', 'accepted offer');
```

Interogarea în baza de date cu o condiție pentru un singur câmp

### 2.3.3.2.1 Operații CRUD în Firestore Database

Acronimul CRUD reprezintă principalele operații care se pot face în orice bază de date, Create (C), Read (R), Update (U), Delete (D).

În Firestore Database adăugarea unei noi înregistrări se poate face în diferite moduri. Prima modalitate este adăugarea unei înregistrări într-un document, operațiune realizată cu `set()`. Dacă documentul precizat există, datele din documentul anterior vor fi suprascrise, însă dacă acesta nu există, documentul va fi creat odată cu adăugarea datelor în bază. Pentru a utiliza `set()` este necesară precizarea unui document în care dorim să adăugăm. [19]

```
refProfile = firebase.firestore().collection("users").doc(currentUser.uid);
refProfile
.set({
  firstName, lastName, age, email, address, phoneNumber, job, companyName, hasDriverLicense, driverCateg, sex, suprf
})
.catch((err) => {
  console.log(err);
});
```

Adăugare în baza de date folosind `set()`

În cazul în care dorim ca Firestore să genereze un id (identificator unic) de document automat, adăugarea se va face folosind `add()`. Aceasta diferă de `set()` prin faptul că întotdeauna creează un nou document și nu suprascrie datele. În implementarea aplicației, în majoritatea cazurilor, pentru adăugarea în bază, am folosit `add()`. [19]

Pentru citirea datelor din bază se pot folosi două metode: `get()` și `onSnapshot()`. Dacă dorim extragerea unor date o singură dată, fără a ține cont de eventualele modificări ce se pot face pe acestea, este suficient să folosim metoda `get()`. Spre deosebire de `get()`, `onSnapshot()`

verifică în mod constant dacă au avut loc actualizări pe datele citite, în caz afirmativ, această funcție se va apela din nou, iar datele returnate vor fi cele mai recente. Această funcție poartă numele de *listener*. [20]

Operația de actualizare pentru un document se realizează cu metoda `update()`, iar cea de ștergere cu `delete()`. Pentru ambele operațiuni trebuie specificat documentul pentru care se vrea modificarea sau ștergerea.

```
const refTask = firebase.firestore().collection("users").doc(currentUser.uid).  
collection("tasks").doc(taskId);  
refTask.update({status: "Done"});
```

Exemplu operație update

```
refCurrentUser.  
doc(product.id).delete().then(() => {console.log("Șters", product.id)})  
.catch((err) => {console.log(err)});
```

Exemplu operație delete

## 3. Implementarea aplicației

### 3.1 Structura fișierelor

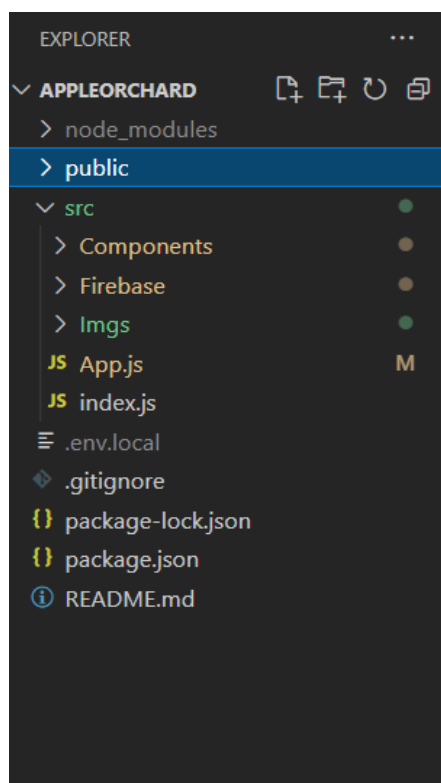


Fig 3.1 – Structurarea fișierelor

Un proiect în React are la bază două mari directoare: **src** și **public**. Cel mai important fișier din public este *index.html*. Structura acestuia este una foarte simplă, însă reprezintă baza pe care React o folosește pentru a afișa componentele sale. Este fișierul care se încarcă atunci când se execută comanda **npm start**. Toate componentele vor fi afișate în tag-ul de tip *div* care are `id="root"`.

În directorul *src* se regăsește un alt fișier important și anume *index.js*. Acesta este corespondentul lui *index.html*.

*App.js* este fișierul în care se află toate rutele pentru componentele din aplicație. Tot aici se definesc ce rute pot fi accesate în funcție de tipul de utilizator conectat la un moment de timp în aplicație. Sunt definite două tipuri de rute: cea pentru utilizatorul de tip **Cultivator** (Grower) și pentru **Angajat** (Employee). În funcție de rolul utilizatorului, completat de către acesta la crearea contului, îi vor fi afișate doar funcționalitățile disponibile tipului de cont pe care îl are.

Directorul *Firebase* conține configurarea firebase pentru aplicația noastră. O voi detalia în subcapitolul dedicat acesteia.

Ultimul director, dar și cel mai cuprinzător, conține toate componentele din aplicație împreună cu stilizarea acestora.

## 3.2 Conectarea la Firebase

Pașii pentru adăugarea Firebase în aplicația noastră sunt:

Condiția prealabilă pentru a utiliza platforma Firebase este deținerea unui cont Google pe care îl vom folosi la autentificarea în această aplicație.

1. Crearea unui proiect în Firebase.
2. Configurarea firebase în aplicația noastră

În continuare voi detalia pașii menționați mai sus.

**Primul pas** pentru a putea integra Firebase în proiectul nostru, este configurarea proiectului în platformă.

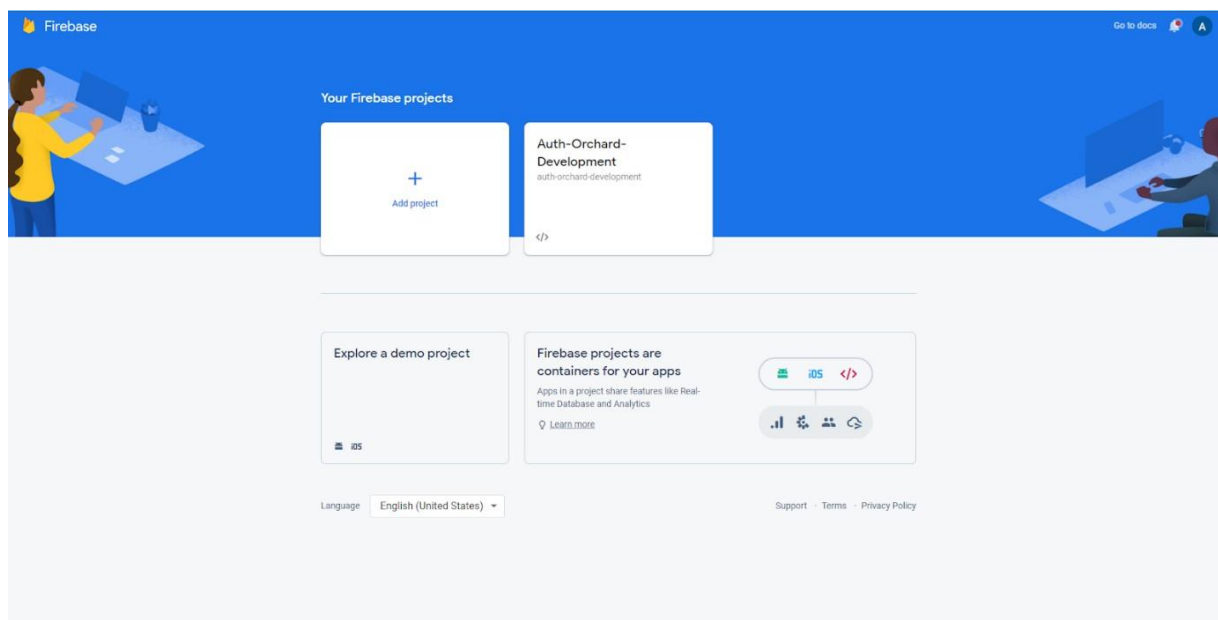


Fig 3.2 – Configurare proiect în Firebase

Pentru aplicația dezvoltată, am creat un proiect denumit **Auth-Orchard-Development**. Am folosit acest proiect pe tot parcursul dezvoltării aplicației.

**Al doilea** pas este configurarea firebase și în aplicația noastră. La crearea proiectului se generează chei pentru a putea utiliza toate funcționalitățile oferite de cloud. Valorile de configurare se pot găsi în meniu **Project Overview – Project Settings**.

Pentru a nu adăuga aceste date în clar în aplicație, toate variabilele sunt stocate într-un fișier de tip **.env.local**, sporind astfel gradul de securitate al aplicației. Datele din fișierele de tip **.env.local** se numesc **variabile de mediu** (environment variables). Variabilele de acest tip sunt adesea folosite atunci când în proiectul nostru integrăm și o componentă de tip cloud, în care avem nevoie de o legătură între cele 2 entități, serviciul de cloud și aplicația noastră. [21]

Pentru a putea asigura securitatea acestor fișiere în care am stocat datele pentru configurarea firebase, în momentul în care adăugăm modificările pe GitHub, am folosit **.gitignore**.

```
const app = firebase.initializeApp({
  apiKey: process.env.REACT_APP_ORCHARD_API_KEY,
  authDomain: process.env.REACT_APP_ORCHARD_AUTH_DOMAIN,
  databaseURL: process.env.REACT_APP_ORCHARD_DATABASEURL,
  projectId: process.env.REACT_APP_ORCHARD_PROJECT_ID,
  storageBucket: process.env.REACT_APP_ORCHARD_STORAGE_BUCKET,
  messagingSenderId: process.env.REACT_APP_ORCHARD_MESSAGING_SENDER_ID,
  appId: process.env.REACT_APP_ORCHARD_APP_ID
})
```

Fig 3.3 - Configurarea Firebase în aplicație folosind variabilele din fișierul **.env.local** [22]

Un alt avantaj, pe lângă cel al securității, este și cel al minimizării modificărilor ce trebuie făcute în cod în cazul modificării acestor variabile. De exemplu, dacă o cheie de autentificare ar fi trecută în clar în cod, la o modificare ulterioară, ar trebui să modificăm cheia în fiecare fișier ce o conține. Însă, în cazul în care folosim fișiere de tip **.env.local**, este suficient să modificăm cheia doar în acesta.

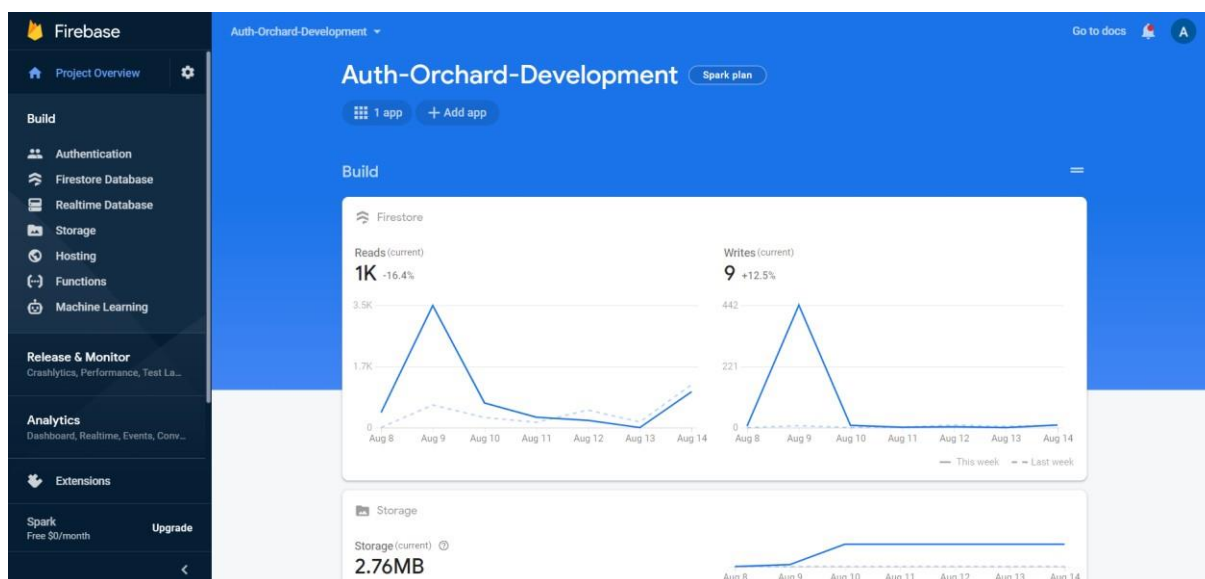


Fig 3.4 – Vizualizare meniu proiect Firebase

În partea stângă putem observa funcționalitățile pe care le oferă Firebase, cele utilizate de mine fiind **Authentication** pentru creare de conturi, **Firestore Database** pentru stocarea de date și **Storage** pentru stocarea de imagini pe care le adaugă utilizatorul în aplicație. De asemenea putem vedea o statistică a citirilor și scrierilor în baza de date dintr-o zi specificată.

### 3.3 Autentificarea folosind Firebase

Având în vedere că am configurat legătura cu Firebase, voi explica în continuare cum am implementat funcționalitățile oferite de cloud pentru modulul de autentificare.

Primul pas este alegerea modurilor de autentificare oferite utilizatorilor, acestea fiind: autentificarea cu mail și parolă, dar și cea folosind contul Google.

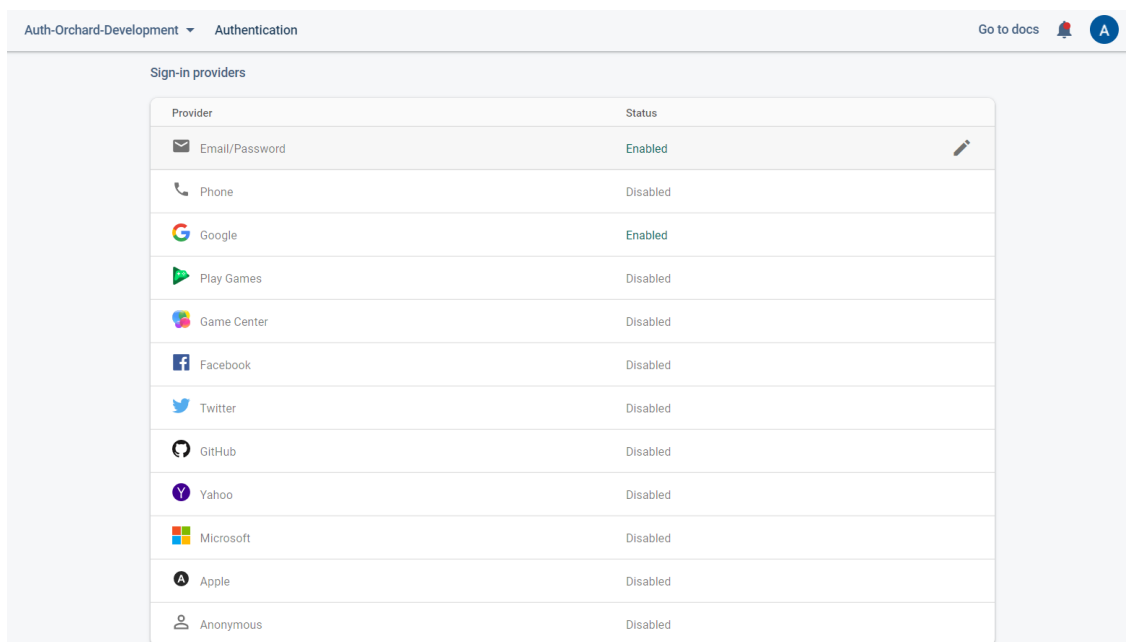


Fig 3.5 – Meniul pentru activarea diferitelor modalități de autentificare

Autentificarea unui utilizator se face prin introducerea de către acesta al unui mail și unei parole. Acestea vor fi ulterior datele lui de conectare în aplicație.

Pentru a prezenta tot procesul de implementare al autentificării, împreună cu restul funcționalităților adiacente acesteia, voi introduce noțiunea de context.

Pentru ca utilizatorul autentificat să poată fi accesat de oriunde din aplicație în timpul unei sesiuni de conectare, trebuie ca toate datele despre acesta, să fie accesibile printr-un mod cât mai eficient și ușor. Ca să putem face acest lucru, trebuie ca toate funcțiile și proprietățile caracteristice utilizatorului curent să fie transmise prin toate componentele aplicației.

### 3.3.1 React Context

Pentru ca utilizatorul curent să fie vizibil pentru toți descendenții din DOM, ar trebui ca toate metodele și proprietățile corespunzătoare lui să fie transmise către nivelele inferioare. React Context este o modalitate de a transmite aceste proprietăți, fără a repeta acest proces pentru fiecare componentă intermediară din arbore. În acest fel, se evită problema numită **prop drilling**, și anume procesul de transmitere al diferitelor proprietăți între o componentă dintr-un nivel superior din DOM și una dintr-un nivel inferior. Poate deveni un procedeu complex în

cazul în care, componentele intermediare nu au nevoie de acele proprietăți și reprezintă doar o zonă de tranziție până în momentul în care ajung în componenta de stop.

React Context poate fi asociat cu rolul unei variabile globale din orice limbaj de programare. Valorile acestora sunt vizibile oriunde din programul nostru, fără să le transmitem în vreun fel între componentele legate între ele. Într-o aplicație web, câteva exemple pentru care poate fi folosit contextul sunt: utilizatorul curent autentificat, limba în care este afișată aplicația, sau un anumit design care trebuie respectat de toate componentele. Însă, este de preferat ca acest mod de a transmite proprietățile între componente, să se facă doar dacă avem nevoie ca datele noastre să fie accesibile unor descendenți care sunt în diferite nivele din DOM. Aplicarea contextului pentru transmiterea unor date pentru un număr mic de nivele, poate duce la o încărcarea mai înceată a componentelor în pagină. [23]

```
// stabilim contextul  
const AuthContext = React.createContext();
```

Stabilirea contextului

În fiecare componentă din aplicație sunt necesare funcțiile de autentificare, înregistrare, logout și cel mai important, utilizatorul curent autentificat în aplicație. Componenta de tip **Provider**, va menține aceste valori în proprietatea *value*, pentru a fi *consumate* de toate componentele ce sunt incluse în acesta.

În componenta App.js, rutele definite vor fi incluse în componenta context definită mai sus, astfel asigurând vizibilitatea proprietăților definite mai sus în orice nivel din aplicație.



## 3.3.2 Autentificare și conectare

### 3.3.2.1 Componentele de SignUp și Login

Pentru a descrie modul în care am implementat componentele de înregistrare (SignUp) respectiv conectare (Login) în aplicație, voi prezenta pe scurt conceptul de promisiune din JavaScript.

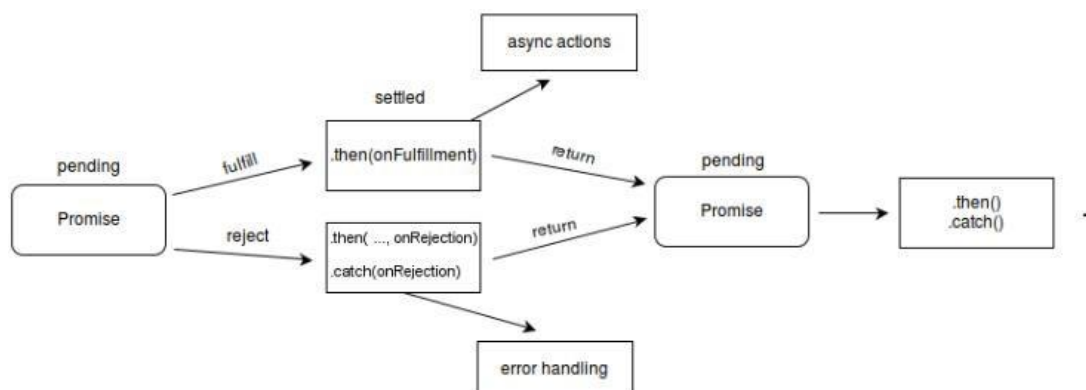


Fig 3.6 – Fluxul unei promisiuni în JavaScript [24]

Promisiunile reprezintă o cale de a lucra cu operațiile asincrone din JavaScript, precum înregistrarea sau conectarea unui utilizator într-o aplicație. Aceasta are trei stări:

1. Prima stare, cea inițială în care nu s-a realizat încă nicio operațiune – în așteptare (pending).
2. Rezolvată (fulfill).
3. Respinsă (reject).

Pentru a putea accesa funcționalitățile aplicației, utilizatorul este nevoit să își creeze un cont la prima conectare în aplicație. Componenta responsabilă de această utilitate este cea de SignUp, vizibilă prin apăsarea butonului de „Autentificare” din pagina principală. La baza acesteia se află un formular în care se completează cele 3 câmpuri: email, parolă și confirmarea parolei.

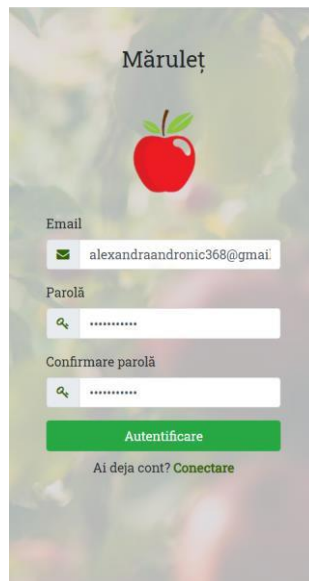


Fig 3.7 – Componenta de autentificare în aplicație

Pentru a putea reține în stare datele introduse în câmpurile pentru email și parolă, am folosit hook-ul de tip **useRef**. Așa cum am menționat în capitolul cu tehnologiile folosite, unde am descris modalitatea de funcționare pentru aceste tipuri de funcții, useRef nu determină o reîncărcare a paginii în momentul în care se schimbă conținutul elementului de tip Form.Control. [12]

```
<Form.Control type="email" ref={refEmail} required aria-
describedby="inputGroupPrependEmail"/>
```

Exemplu câmp formular de înregistrare

La completarea câmpului, valoarea corespunzătoare atributului ref din fiecare input, va putea fi accesată prin proprietatea **.current.value**. Însă, la o reîncărcare a paginii, valoarea câmpurilor și referințelor vor fi pierdute.

După ce utilizatorul va completa corect câmpurile și anume: email-ul are un format corespunzător și parolele vor respecta criteriile de siguranță, se va apela funcția de signup.

Funcția **signup** primește ca parametri email și parola. Aceasta este o funcție ce returnează o promisiune.

```
await signup(refEmail.current.value, refPassword.current.value);
```

Apelarea funcției pentru înregistrarea unui utilizator în aplicație [22]

În momentul în care aceasta este apelată, în Firebase se va căuta un alt cont cu același email, dacă acesta nu există, se va crea un cont având credențialele specificate de utilizator. Aceasta se află inițial în starea de *pending*. În cazul în care se va încerca crearea unui cont cu

un mail deja existent, promisiunea va fi cu statusul de *rejected* și autentificarea nu se va efectua cu succes, apărând o alerta cu un mesaj corespunzător. După crearea contului, conectarea se va face automat. De asemenea, un mail de confirmare al adresei, va fi trimis utilizatorului. [22]

La fiecare reîncărcare a paginii, avem nevoie ca datele despre utilizatorul curent conectat să fie disponibile la nivel global în aplicație, indiferent de componentă. Funcția `onAuthStateChanged` va urmări la fiecare operațiune datele despre utilizatorul curent autentificat, astfel că la modificarea unei componente, utilizatorul va rămâne conectat în aplicație. Această funcție mai poartă numele de *listener*.

Fiind în interiorul unui `useEffect`, `onAuthStateChanged` se va reapela la fiecare reîncărcare a paginii, menținând astfel datele despre utilizatorul curent conectat prin setarea: `setCurrentUser(user)`.

Componenta de conectare, **Login**, funcționează pe același principiu ca și cea de **SignUp**. La o conectare ulterioară creării de cont, utilizatorul va introduce mail-ul și parola, iar Firebase va căuta în baza de date un utilizator ce corespunde datelor introduse de acesta în sesiunea curentă. În cazul unei completări eronate, utilizatorul va primi o eroare.

### 3.3.2.2 Conectare cu Google

O altă modalitate de conectare în aplicație este cea folosind contul de Google. La apăsarea butonului Conectare cu Google se va deschide o fereastră în care putem alege un cont de Google salvat în browser. În funcție de setările fiecărui utilizator pentru stocarea parolelor, la conectare se va cere parola sau nu.

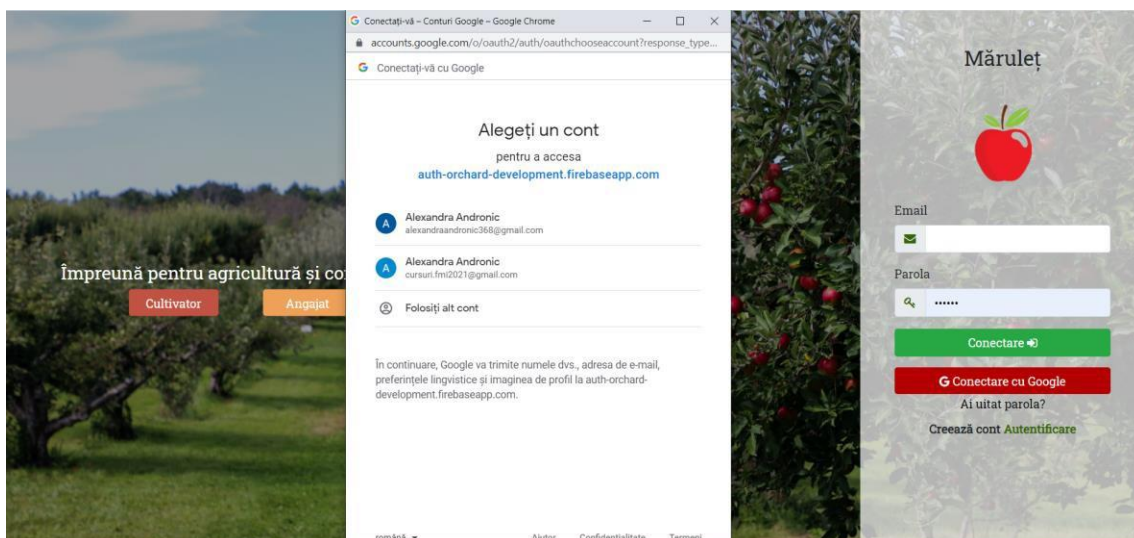


Fig 3.8 – Conectare folosind contul Google

Pentru a putea configura și acest tip de înregistrare, trebuie să instanțiem un nou obiect folosind `GoogleProvider`. Am ales ca modalitate de alegere al contului, deschiderea unei ferestre de tip pop-up, aici vor apărea conturile disponibile în browser, dar va exista și posibilitate adăugării unui nou cont google. [25]

```
export const googleProvider = new firebase.auth.GoogleAuthProvider();
```

Instanțiere obiect pentru conectare cont google

### 3.3.2.3 Stocarea parolelor în Firebase

Un aspect foarte important al fiecărei aplicații îl reprezintă securitatea. Datele utilizatorilor trebuie să fie stocate în siguranță, cu prioritate parolele. Firebase folosește propriul algoritm de stocare criptată a parolelor. Indiferent de modul de transmitere al parolei către funcțiile de autentificare, în clar sau criptate cu un algoritm ales de dezvoltator, Firebase va folosi propriul algoritm. Parametrii de hash pentru parole se pot vizualiza din modulul de utilizatori, la secțiunea **Password Hash Parameters**. Firebase generează acești parametri pentru fiecare proiect creat. [26]

## 3.4 Definirea rutelor

Având în vedere că toate funcționalitățile oferite de aplicație trebuie să poată fi accesate doar pentru utilizatorii conectați, am definit rute private. Astfel că, pentru un utilizator neconectat, paginile accesibile vor fi doar acelea de înregistrare în aplicație și conectare.

În directorul **Routes** din *src* am definit două rute separate pentru cele două tipuri de utilizatori din aplicație, Cultivator (**Grower**) și Angajat (**Employee**). În momentul primei conectări, utilizatorul completează de asemenea și rolul dorit, Cultivator sau Angajat. În funcție de acest rol, funcționalitățile fiecărui tip de cont se schimbă.

```
<Grower path="/task" component={Task} />
```

Exemplu rută cont Cultivator

```
<Employee path="/see-posts" component={SeePosts} />
```

Exemplu rută cont Angajat

Atunci când utilizatorul dorește să acceseze o anumită pagină, în componentele de tip Grower și Employee se va face verificarea rolului din profilul acestuia.

```
const refRole = firebase.firestore().collection("userRole").doc(currentUser.uid);
```

Interogare bază de date pentru a găsi rolul utilizatorului curent conectat

După cum am specificat și în subcapitolul ce prezintă structura proiectului, fișierul App.js conține toate rutele aplicației. Toate acestea sunt cuprinse în componenta <AuthProvider/> prezentată mai sus, responsabilă de transmiterea utilizatorului curent conectat în toată aplicația fără a mai face acest lucru explicit.

```
<Router>
  <AuthProvider>
    <Switch>
      <Route path="/neauth-home" component={Home} />
      <Route path="/signup" component={SignUp} />
      <Route path="/login" component={Login} />
      <Route path="/addprofile" component={AddProfile} />

      { /* Rutele pentru utilizatorii de tip cultivatori */ }
      <Grower path="/weather" component={Weather} />
      <Grower path="/task" component={Task} />
      <Grower path="/forum" component={ForumMainPage} />

      .....
      { /* Rutele pentru utilizatorii de tip angajati */ }

      <Employee path="/see-posts" component={SeePosts} />
```

```

        <Route component={NotFoundPage} />
      </Switch>
    </AuthProvider>
  </Router>

```

Fișierul App.js și rutele aplicației

Fiecare rută conține calea (*path*), dar și componenta ce trebuie încărcată în pagină (*component*). **Switch** va căuta prima rută care corespunde celei date de către utilizator prin intermediul componentelor Link sau Redirect.

În momentul în care utilizatorul dorește să acceseze o pagină pentru care nu are drepturi, acesta va fi redirecționat către pagina de Login. Dacă dorește accesarea unei pagini pentru care calea nu este recunoscută de componenta Switch, utilizatorul va fi redirecționat către pagina de NotFound, de unde poate accesa din nou pagina principală sau va fi redirecționat automat în 10 secunde.

```

return <Redirect to="/neauth-home" />

```

Redirecționarea către pagina de Login



Fig 3.9 - Pagina NotFound

## 3.5 Transmiterea proprietăților între componente

În subcapitolul **React Context** am prezentat o metodă de transmitere de proprietăți între componente folosind Contextul. Însă aceasta este o modalitate folosită atunci când dorim transmiterea de date între componente care se află în nivele diferite în DOM sau în momentul

în care dorim ca acestea să fie transmise tuturor componentelor. O transmitere “de mână” ar lua mult timp, dar ar putea genera și erori. Atunci când vrem ca unele date să fie transmise doar printr-un număr mic de nivele, putem folosi metoda transmiterii explicite de date.

Un caz concret în care am folosit transmiterea explicită este cel al request-ului către API-ul pentru vreme. În prima pagină ce îi este afișată utilizatorului după conectare, va apărea o secțiune cu datele despre vremea din ziua curentă. Însă aceste date sunt necesare și în componentele ce se vor încărca la apăsarea butonului **Vreme**, astfel că am decis să transmit colecția cu aceste date pentru a nu mai face request-ul și în următoarele pagini. În acest fel pot evita atingerea numărului maxim de apeluri către acest API pentru planul gratuit.

```
const element = <Welcome name="Sara" />;
```

Fig 3.10 – Transmiterea de date în mod explicit între componente [27]

Spre deosebire de modul de transmitere al proprietăților din Figura 3.10, am ales transmiterea acestora folosind componenta Link din React.

```
<Link to={{
  pathname: '/weather',
  state: {data}
}}><Button className={styles.progTreat}>Vremea &nbsp; <i className=
"fa fa-arrow-right" aria-hidden="true"></i></Button></Link>
```

Transmitere de date în componenta MainPage către componenta Weather [28]

*Pathname* reprezintă calea spre pagina pe care dorim să o încărcăm, iar datele sunt transmise prin intermediul *state*. Astfel am evitat calcularea acestora în fiecare componentă în care aveam nevoie de aceste date.

Pentru a accesa datele primite în componenta destinație, am folosit funcția de tip hook `useLocation`. Aceasta ne permite să accesăm datele din componenta Link.

```
const weatherData = location.state.data;
```

Modalitate de accesare a datelor transmise de componenta Link

## 3.6 Modul de lucru al Firebase integrat în componentele React

În continuare voi descrie modul de funcționare al componentelor principale pentru a exemplifica modul de lucru cu baza de date din Firebase Cloud integrat în componentele din React.

### 3.6.1 Forum

Componenta **Forum** permite utilizatorul postarea de topicuri privind diferite subiecte din pomicultură. Adăugarea unui topic se face prin click pe butonul **Topic nou** ce va deschide o fereastră Modal cu un formular ce conține datele necesare postării topicului. Noutatea în această componentă o reprezintă adăugarea de fișiere, mai exact imagini. Stocarea acestora se face în **Storage Firebase**. Aceasta are la bază Realtime Database.

Primul pas pentru a putea afișa și aduce imaginile din baza de date este importarea `firebase/storage` în fișierul de configurare al Firebase. Pentru ca la adăugarea topicului să se permită încărcarea unei imagini de pe dispozitivul utilizatorului, trebuie să specificăm tipul câmpului în care se va face adăugarea imaginii. Tipul input-ului trebuie să fie de forma *file*.

```
<Form.Control type="file" onChange={onFileChange} aria-  
describedby="inputGroupPrependTopicImage"
```

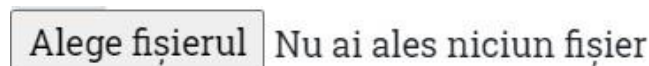


Fig 3.11 - Input de tip fișier

Funcția dată ca argument evenimentului `onChange`, va salva imaginea în starea componentei, pentru ca mai apoi aceasta să fie adăugată în Storage. Funcția ce realizează încărcarea imaginii în baza de date este una asincronă, astfel că definiția acesteia necesită plasarea cuvântului cheie `async`, iar adăugarea propriu-zisă, `await`.

```
const addTopic = async (e, topicName, question, topicCateg)
```

Definiția funcției pentru adăugarea unui nou topic

```
await fileRef.put(file);
```

Adăugarea imaginii în Storage Firebase folosind funcția `put` [29]

În înregistrarea adăugată în baza de date, în documentul generat în colecția „forumTopic”, am salvat de asemenea și url-ul imaginii pentru a-l accesa în atributul `src` al `<img />`. URL-ul este accesat prin proprietatea referinței `getDownloadURL`.



### 3.6.2 Modulul pentru anunțuri de angajare

Această funcționalitate este prezentă în ambele tipuri de conturi. Cultivatorul publică un anunț pentru un tip de job, punând la dispoziție o listă de cerințe pe care angajatul trebuie să le îndeplinească, dar și o scurtă descriere. La click pe butonul **Adaugă anunț** va apărea un formular ce conține câmpuri aferente informațiilor descrise mai sus. Cele mai importante câmpuri din acest formular sunt cele de Permis conducere, respectiv Categoriile permis, câmp afișat doar în cazul completării cu DA al primului câmp precizat. Câmpul apare sub forma mai multor căsuțe ce pot fi bifate sau nu, fiecare corespunzând unei categorii de permis auto.

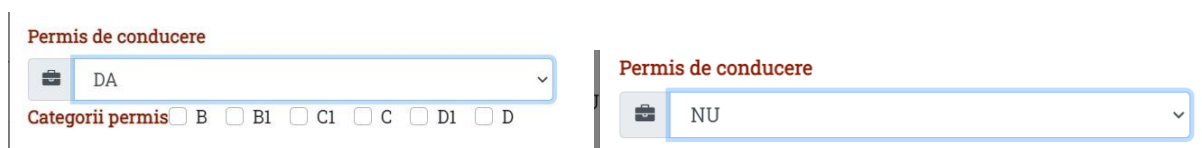


Fig 3.12 - Câmpuri din formularul pentru adăugare nou anunț angajare

Pentru a da efectul de dinamicitate formularului în cazul în care utilizatorul alege opțiunea DA pentru Permis conducere, am folosit hook-ul **useState**. La orice schimbare a informației din input (DA sau NU), evenimentul ce se va declanșa la onChange va modifica starea și va duce la apariția câmpului Categoriile permis (pentru DA).

```
const [driverLicense, setDriverLicense] = useState('NU');
```

La fiecare reîncărcare, starea driverLicense va fi setată pe NU

```
<Form.Control as="select" aria-describedby="inputGroupPrependDriverLicense"
               value={driverLicense} onChange={(e) => setDriverLicense(e.target.value)}>
  *****
</Form.Control>
```

După validarea formularului, datele sunt adăugate în baza de date, în colecția **jobPosts**.

Utilizatorul de tip angajat va putea vizualiza toate locurile de muncă publicate de toți cultivatorii. De asemenea, se va putea face filtrarea lor în funcție de locație. La fiecare introducere a unei locații, se vor căuta și afișa doar acele înregistrări din baza de date ce au câmpul **location** egal cu cel completat de utilizator. Folosind **useState** pentru actualizarea

permanentă a stării în momentul introducerii unei locații, pagina se va reîncărca automat, deoarece, după cum am explicat și în subcapitolul dedicat funcțiilor hook, useState produce o reîncărcare automată la schimbarea stării. Fiecare card ce conține anunțul unui cultivator, prezintă câteva informații despre locul de muncă publicat. Utilizatorul de tip angajat va avea opțiunea de **Salvează job**, iar butonul **Aplică** îi va apărea doar în cazul în care categoriile de permis ale acestuia corespund cu cele din anunț.

Pentru a afla aceste date ale angajatului, citim din baza de date, informațiile aferente profilului, completate la înregistrarea în aplicație.

```
const refProfile = firebase.firestore().collection("users").doc(currentUser.uid);
```

Profilul utilizatorului curent

```
function getProfile() {
  refProfile.onSnapshot((doc) => {
    profile.current = doc.data();
    userDriverCateg.current = doc.data().driverCateg;
  })
}
```

Extragerea informațiilor aferente categoriile de permis

După procesul de aplicare al utilizatorului de tip angajat, cererea se va adăuga în colecția **onHold** cu statusul „In asteptare”, iar cultivatorul va vedea toate aceste cereri în secțiunea „Cereri”.

```
const refOnHoldPosts = firebase.firestore().collection("users").doc(growerId).collection("onHold");
```

```
refOnHoldPosts
  .add({
    employeeId: emplId,
    growerId: growerId,
    postId: postId,
    status: 'In asteptare',
    lastName: profile.current.lastName,
    firstName: profile.current.firstName,
    driverLicense: profile.current.hasDriverLicense,
    postName: postName,
    postDescription: postDescription,
    categories: categories,
    year: year
```

```

    })
    .catch(err => {
      console.log(err);
    });

```

Adăugarea în onHold cu status ‘In asteptare’

Pentru a optimiza procesul de căutare în baza de date, în cazul afișării cererilor pentru cultivatori, am adăugat filtrarea după câmpul *status* folosind clauza *where*. Acesta are două opțiuni, **Acceptare cerere** și **Respingere**. În cazul acceptării, statusul cererii va fi schimbat în *,accepted’*, iar cultivatorul va trebui să introducă o ofertă salarială, câmp adăugat în înregistrare. În caz contrar, cel al respingerii cererii, acesta trebuie să introducă și motivația deciziei, iar statusul va fi modificat în *,rejected’*.

```

const refOnHoldPosts = firebase.firestore().collection("users").doc(currentUser.uid).collection("onHold").where('status', '==', 'In asteptare');

```

Extragerea anunțurilor cu status ‘In asteptare’

Utilizatorul de tip angajat, va vedea starea tuturor cererilor sale în submeniul **Oferte salariale**. Pentru cererile ce au status *,accepted’* din partea angajatorului, acesta are opțiunea de a **Accepta** sau **Respinge** oferta salarială. În cazul acceptării, statusul se va schimba în *,accepted offer’*, altfel în *,rejected offer’*.

### 3.6.3 Programare operațiuni

Această funcționalitate leagă componentele angajați, utilaje și vreme în vederea programării unei operațiuni la un moment potrivit, cât și atribuirea acesteia unui angajat. Acest lucru presupune extragerea informațiilor necesare din baza de date, cât și o preprocesare a acestora. Deoarece doresc ca toate datele să fie aduse la încărcarea paginii, folosesc hook-ul **useEffect**. Extrag datele pentru utilajele cultivatorului, respectiv angajații. În plus, programarea de tratamente necesită o listă de substanțe împreună cu problemele pe care le tratează, dar și stadiul de dezvoltare în care se aplică. Pentru a face acest lucru, am considerat schema de tratament Syngenta, după care am făcut o colecție în baza de date ce conține toate substanțele din această schemă. [30]

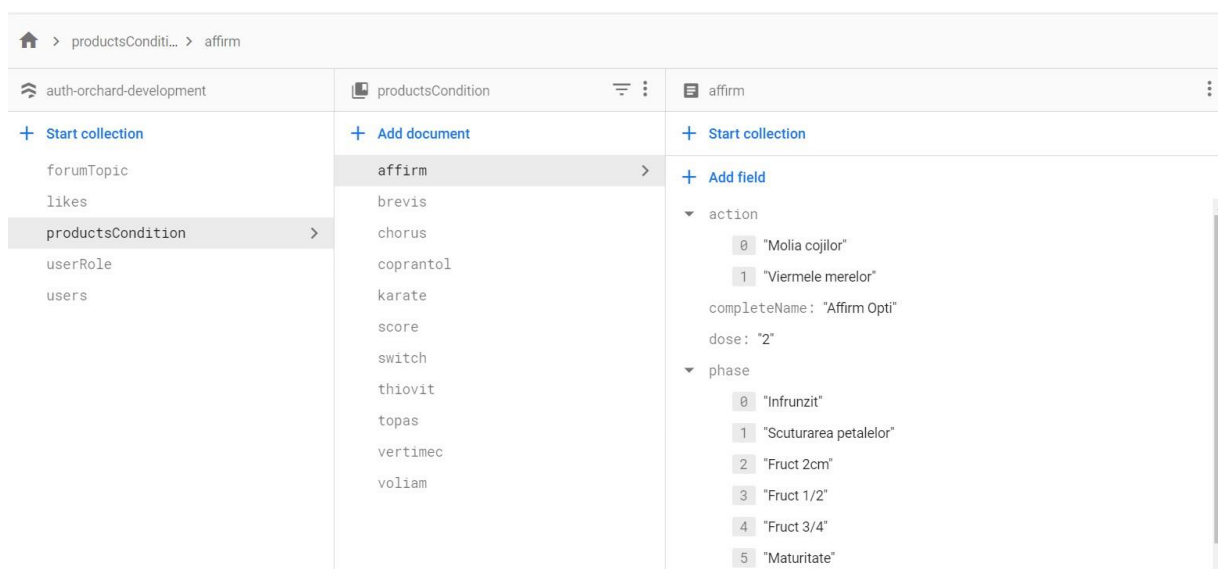


Fig 3.13 - Colecția cu produsele și probleme pe care le tratează

Câmpul *action* reține problema pe care produsul o tratează, *dose* reprezintă cantitatea calculată în kilograme care se aplică pe un hectar, iar *phase* reține stadiul în care se poate aplica substanța respectivă. Datele despre vreme vor fi primite de la componenta anterioară prin intermediul componentei <Link />.

La click pe butonul **Programează operațiune**, utilizatorul completează inițial doar numele operațiunii, dar și data. În cazul introducerii unei date anterioare celei curente, acesta nu va putea continua completarea formularului și va primi o eroare. În funcție de temperaturile din ziua aleasă de utilizator, vor apărea mesaje de avertizare în cazul unor temperaturi neprielnice pentru programarea de operațiuni. Dacă temperatura va fi peste 25 de grade Celsius sau sub 0 grade Celsius (pentru tratamentele de primăvară), utilizatorul va vedea atenționările aferente.

```
if(p.temp.day > 25)
{
    // daca temperatura este mai mare de 25 de grade => este indicat sa nu se administreze tratamente in acele zile
    items.push({id: id, data: dt, temperature: p.temp.max});
    hot = true;
    id += 1;
}
if(p.temp.day < 0)
{
    // temperaturi scazute
    items.push({data: dt, temperature: p.temp.max});
    cold = true;
}
```

## Condițiile pentru administrarea de tratamente

```
if(param_date < currentDate)
{
    showAlert(<Alert variant="warning">Nu poate fi programata o operati
une cu data anterioara datei curente</Alert>);
    showAlertDate(true);
}
```

Verificare pentru alegerea datei de către utilizator

După introducerea unei date valide, vor apărea câmpurile pentru setarea unei ore (ulterioară celei curente), dar și al tipului locului de muncă pe care trebuie să-l aibă angajații care vor putea fi atribuiți acestei operațiuni. După completarea acestora, vor apărea câmpurile **Durată operațiune** și **Tip operațiune**. Dacă tipul operațiunii este **Tratament**, vor apărea două câmpuri de tipul select ce conțin stadiul de dezvoltare și problemă tratată.

```
if(p.phase.includes(phase) && p.action.includes(problem))
{
    matchingProducts.push({id: p.id, name: p.completeName, dose: p
.dose, water: p.water});
}
```

Căutarea produselor ce se pot aplica în funcție de un stadiu de dezvoltare și o problemă

Următorul pas este căutarea utilajelor disponibile. Pentru a le găsi, voi parcurge colecția ce conține task-urile, recalculat la fiecare adăugare a unei noi operațiuni în baza de date, și voi verifica dacă pentru ora aleasă de utilizator, același utilaj este deja atribuit altei operațiuni.



Fig 3.14 - Listarea utilajelor disponibile

În funcție de utilajul ales de utilizator, vor apărea doar acei angajați ce au categoriile de permis necesare utilajului ales, dar care au și tipul locului de muncă egal cu cel ales la al doilea câmp de completat.

```
if(!ok && jobType == emp.postName && emp.growerId == currentUser.uid)
{
    // daca este liber un angajat
    // acum trebuie sa verificam daca are acele categorii de permi
s
    if(mach.driverCateg.length > emp.categories.length )
    {
        if(mach.driverCateg.includes(emp.categories))
        {
            items.push(emp);
        }
    }
}
```

Verificarea pentru alegerea angajaților

De asemenea, un lucru important de menționat este acela că un utilizator de tip angajat poate avea statut de angajat pentru mai mulți cultivatori, deci este important să verificăm disponibilitatea în funcție de toate operațiunile care îi sunt atribuite la momentul de timp dorit.

Datele aferente operațiunii pentru care s-au completat datele, vor fi adăugate de asemenea în colecția *tasks*.

## 3.7 React-Bootstrap

În continuare voi prezenta câteva componente din React-Bootstrap folosite în dezvoltarea aplicației. Aceste pachete pot fi găsite împreună cu documentația și comanda de instalare pe site-ul [npmjs.com](https://www.npmjs.com). Componentele utilizare frecvent în dezvoltarea acestei aplicații sunt: **Form**, **Card**, **Modal** și **Table**. [31]

### 3.5.1 Form

Componenta de tip formular Form, reprezintă corespondenta elementului <form> din HTML. Aceasta are numeroase avantaje în comparație cu cea nativă din HTML. Aspectul este în mod vizibil unul îmbunătățit, fiind mult mai ordonat față de cel din HTML. [32]

Am utilizat această componentă atunci când doream ca utilizatorul să introducă date pentru ca ulterior acestea să fie adăugate în baza de date și accesate la nevoie. Prima interacțiune cu un astfel de formular este chiar din prima pagină, formularul de conectare sau înregistrare în aplicație. Pentru a exemplifica funcționarea acestei componente, voi descrie implementarea componenteii de adăugare a profilului după prima conectare în aplicație.

Fiecare câmp din formular este compus din denumirea câmpului, numit *label*, dar și din zona în care se face completarea efectivă numită *input*.

```
<Form.Label htmlFor="lastname"><strong className={styles.tags}>Nume</strong></Form.Label>
```

Exemplu *label* în React-Bootstrap pentru câmpul Nume

```
<Form.Control ref={lastName} type="text" placeholder="Nume" aria-describedby="inputGroupPrependLastName" required className={styles.formInputControl}/>
```

Exemplu *input* în React-Bootstrap pentru câmpul Nume

Referința *ref* este variabila care va reține valoarea introdusă de utilizator pentru câmpul **Nume** fără a produce de fiecare dată o reîncărcare a paginii. Prin setarea atributului *type* ca *text*, valoarea introdusă de utilizator va fi întotdeauna un string. Acesta poate avea diferite valori în funcție de informația pe care dorim să o salvăm. Atributul *required* obligă utilizatorul să completeze câmpul ce are această proprietate setată.

Pentru ca informațiile completate să fie adăugate în baza de date, trebuie ca acestea să fie validate. Validarea datelor din formular se realizează prin setarea proprietății *validated*. Submiterea formularului se va realiza doar după ce această proprietate va fi setată la *true*.

```
if (form.checkValidity() === true)
```

Funcție ce verifică validarea formularului

În cazul completării eronate al unul câmp, vor apărea semne de atenționare.

Fig 3.15 - Exemplu validare formular adăugare profil

După completarea corectă a formularului, datele vor fi adăugate în baza de date, în colecția denumită **users**, în documentul ce are id-ul utilizatorului curent autentificat (accesat prin proprietatea `UID`). Am ales această abordare, de a transmite explicit id-ul, pentru ca interogarea bazei de date pentru un utilizator să se realizeze mult mai ușor și eficient, dar și pentru a mă asigura că pentru fiecare utilizator înregistrat în aplicație există un singur profil asociat.

```
const refProfile = firebase.firestore().collection("users").doc(currentUser.uid);
```

Calea pentru adăugarea profilului în documentul aferent utilizatorului curent

```
refProfile
.set({
  firstName, lastName, age, email, address, phoneNumber, job, company
  Name, hasDriverLicense, driverCateg, sex, suprf
})
.catch((err) => {
  console.log(err);
});
```

Adăugarea în baza de date a profilului

### 3.5.2 Modal

Componentele de tip **Modal** sunt ferestre de tip pop-up, ce apar la apăsarea unui buton. Acestea sunt poziționate peste restul componentelor din pagină și dispar la apăsarea în orice zonă din afara componentei **Modal**, însă acest comportament poate fi modificat de dezvoltator. Am folosit această componentă pentru mesaje de atenționare, dar și pentru completarea de formulare, atunci când nu doream să mai încarc conținutul deja existent al paginii. [33]



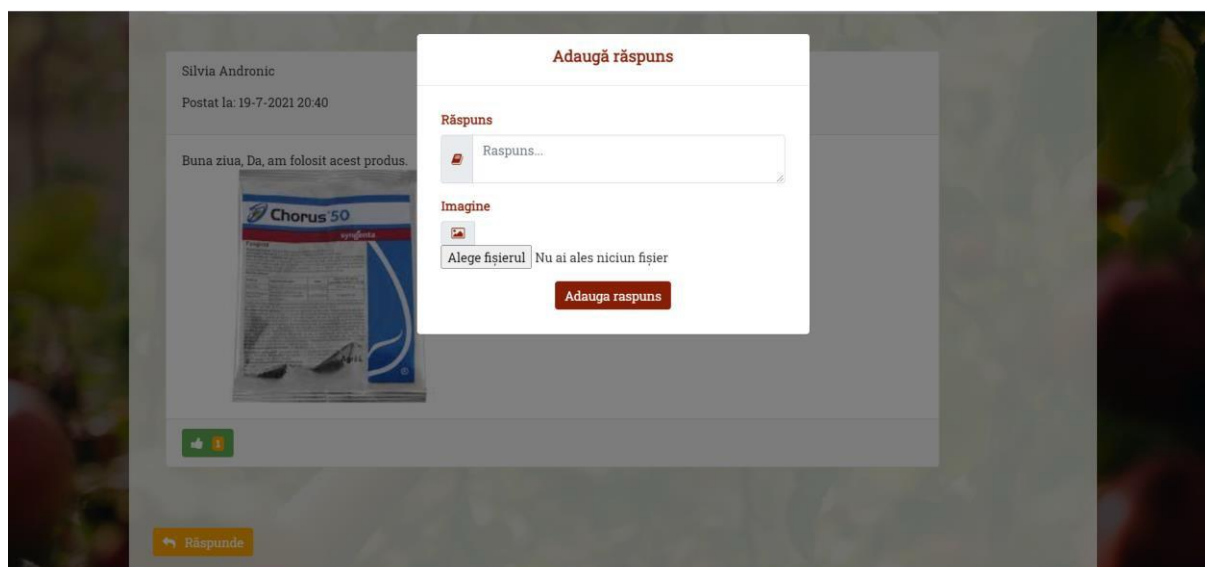


Fig 3.16 - Exemplu fereastră Modal pentru adăugarea de răspuns în Forum

### 3.5.3 Table

O altă componentă frecvent utilizată este **Table**. Voi prezenta în continuare tabelul realizat în secțiunea de *Facturi*, ce conține paginare și filtrare.

<input type="text" value="Cauta factura"/>				
Soi	Pret (lei)	Cantitate (total)	Sterge	Detalii
Red Melba	1250	25		
<b>Total</b> 1250 lei			<< < > >> 1 / 1 Pagina: <input type="text" value="1"/>	
<input type="button" value="+ Adauga factura"/>				

Fig 3.17 - Exemplu componentă Table cu paginare și filtrare

Pentru a adăuga funcțiile de paginare și filtrare, am folosit funcțiile de tip hook pentru tabele: [useTable](#), [usePagination](#) și [useFilters](#). [34]

Primul pas pentru crearea acestui tip de tabel este stabilirea capului de tabel și al datelor ce vor fi conținute în coloanele respective. Funcția hook [useTable](#) stabilește structura tabelului. Acesta primește parametrii *columns* și *data*, funcții ce trebuie să fie [memoizate](#).

Parametrul *columns* este o funcție memoizată, adică reapelarea și recalcularea unei valori se vor executa doar dacă valoarea n-a mai fost calculată într-un apel anterior al funcției.

```
const columns = React.useMemo(
  () => [
    {
      Header: 'Nume produs',
      accessor: 'product'
    },
    {
      Header: 'Preț (lei)',
      accessor: 'price'
    },
    {
      Header: 'Cantitate (kg)',
      accessor: 'quantity'
    }
  ],
  []
);
```

Exemplu funcție **columns**

Pentru a da caracterul de funcție memoizată, am folosit hook-ul **useMemo** din React. Acesta returnează un vector ce conține numele coloanelor (Header), dar și numele variabilelor ale căror date vor fi stocate în tabel (accessor). Deoarece vectorul de dependențe nu conține nicio valoare, această funcție va fi calculată doar la încărcarea componentei în pagină. **useMemo** îmbunătățește de asemenea performanța în aplicația noastră.

Următorul pas este adăugarea altor funcționalități pe care le dorim ca și argumente în hook-ul **useTable**. Pentru tabelul implementat, au mai fost adăugate și funcționalitățile de paginare și filtrare după nume.

```
useTable(
  {
    columns, data, initialState: {pageIndex: 0, pageSize: 4},
  },
  useFilters,
  usePagination
)
```

Funcția **useFilters** este folosită pentru filtrarea liniilor tabelelor în funcție de un input dat de către utilizator în **Caută factură**. Am ales să fac această filtrare după **Nume (Soi, în cazul facturilor pentru pomi)**.

## 3.8 Open Weather API

În implementarea modului pentru programarea operațiunilor în funcție de vreme, am folosit OpenWeather API. Acesta este un serviciu online care livrează date despre vreme și nu numai pentru orice locație geografică prin intermediul unui API. În funcție de planul ales pentru folosirea aplicației, aceasta oferă de la datele meteorologice din ziua curentă, până la vremea din următoarele 16 zile și chiar hărți privind poluarea din zona precizată. Am ales planul gratuit ce permite 60 de request-uri pe minut și 1000000 pe lună. Documentația fiecărui API este explicată în detaliu, de asemenea și parametrii acestora. [35]

Pentru a primi datele pentru vremea pe următoarele opt zile ce vor fi necesare în programarea operațiunilor, am folosit următorul request pentru care voi explica semnificația parametrilor, găsită și în documentație.

```
https://api.openweathermap.org/data/2.5/onecall?lat={lat}&lon={lon}&exclude={part}&appid={API key}
```

Fig 3.18 – Apel API OpenWeather

Datele vor fi generate în funcție de latitudinea și longitunea poziției geografice în care ne aflăm într-un moment de timp. Acestea au fost determinate folosind funcția `getCurrentPosition` din JavaScript. Temperatura este afișată în grade Celsius prin setarea parametrului **units=metric**, dar și descrierea vremii să fie în limba română prin **lang=ro**.

Pentru a primi datele necesare din request-ul către API-ul precizat mai sus, am folosit metoda `fetch` din React. Reprezintă cea mai accesibilă modalitate de a primi aceste date. Pentru a face un request de tip **GET** (primim date de la server), cum este cel din cazul de față, trebuie să includem URL-ul, dar și parametrii necesari pentru a primi datele în formatul dorit. Latitudinea și longitudinea nu vor fi date în clar, ci vor fi calculate în funcție de locul în care se află utilizatorul în momentul accesării aplicației. Acest request este făcut în interiorul funcției `useEffect` ce are ca vector de dependențe variabilele de latitudine și longitudine, astfel că, reîncărcarea paginii se va face în momentul în care aceste date se vor schimba. De asemenea, fiecare request are nevoie de o cheie unică generată la crearea contului în aplicația OpenWeather Map, regăsită la datele contului personal. Am dorit ca răspunsul primit să-l convertesc într-un format mai ușor de parcurs și citit, astfel că după ce datele vor fi primite de la server, vor fi transformate în obiecte de tip JSON. [36]

## 3.9 Aplicație responsive

Conceptul de *responsive* face referire la adaptarea dinamică a design-ului în funcție de dimensiunile dispozitivului sau de orientarea sa. În aplicația prezentată, această funcționalitate a fost realizată cu ajutorul componentelor din React-Bootstrap, dar și cu CSS.

Multe dintre componentele din React-Bootstrap cum ar fi Modal, Button sau Form au deja implementată adaptabilitatea de diferite ecrane. Deoarece am dorit un control mai mare asupra alinierii în pagină în funcție de dispozitivul de pe care este accesată aplicația, am folosit și regulile oferite de CSS și anume **media queries**. [37]

Media queries reprezintă blocuri alcătuite din diferite reguli CSS definite de dezvoltator (clase și id-uri) ce vor fi aplicate codului HTML doar în cazul în care condițiile privind rezoluția ecranului sunt îndeplinite. Am adaptat designul aplicației pentru trei tipuri de ecrane: desktop, telefon și tabletă.

Pentru a testa această funcționalitate am folosit Inspector Element din browser. Cu ajutorul acestuia se poate simula vizualizarea aplicației de pe diferite dispozitive.

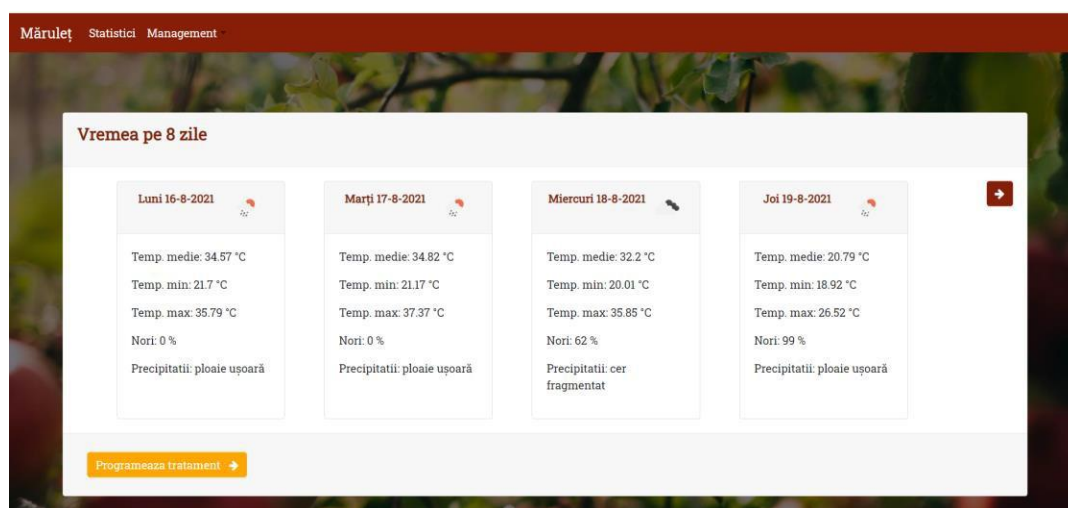


Fig 3.19 – Pagină vreme vizualizată de pe laptop

Pentru telefon am considerat următorul interval de rezoluții:

```
@media (min-width: 300px) and (max-width: 500px)
```

Condiție pentru aplicare reguli ecran telefon

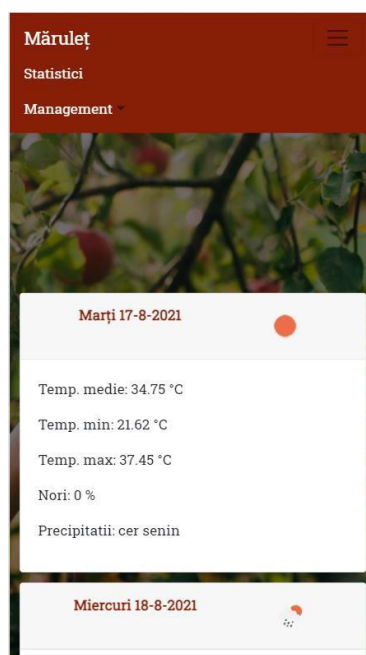


Fig 3.20 – Pagină vreme vizualizată de pe mobil

Astfel că, dacă rezoluția ecranului de pe care este accesată aplicația se încadrează între 300px și 500px, CSS va aplica inclusiv codul din acest bloc. Dacă secvența conține selectori deja definiți în afara blocului, aceștia vor fi suprascrise. Pentru ca aceste reguli scrise în interiorul blocurilor media să aibă efect, trebuie să fie poziționate la sfârșitul fișierelor CSS.

## 4. Ghid de utilizare

### 4.1 Înregistrarea și conectarea

Indiferent de tipul de utilizator, înregistrarea unui cont în aplicație, cât și conectarea, vor avea același flux. Primul pas în vederea accesării funcționalităților este crearea unui cont, utilizatorul alegând între înregistrarea cu mail și parola, urmată de confirmarea contului, sau cea folosind contul Google.

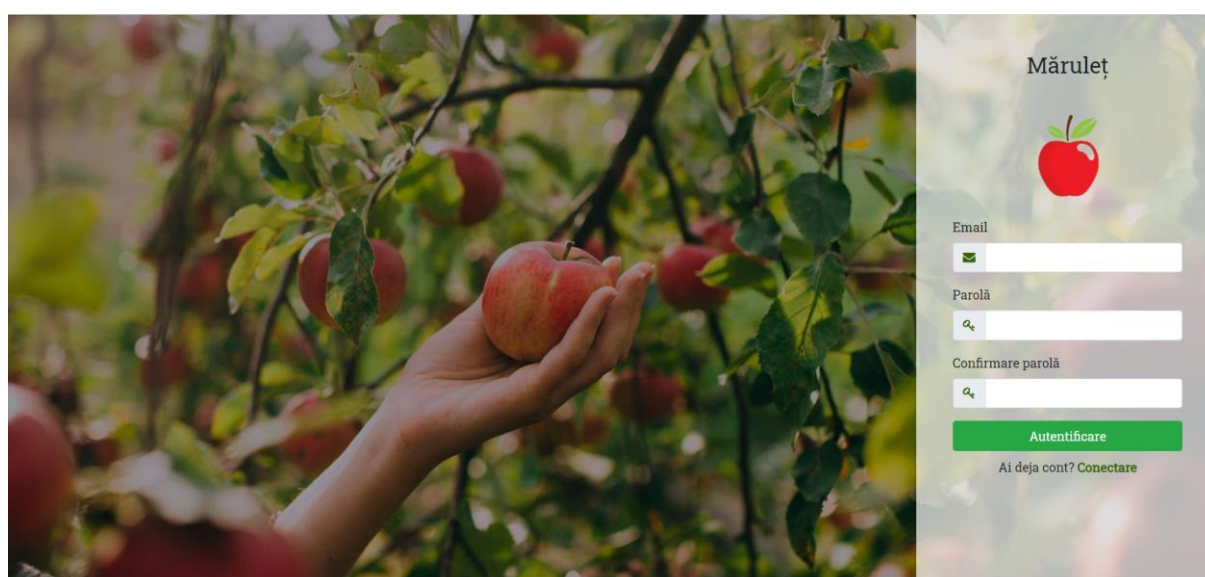


Fig 4.1 – Formular pentru înregistrarea unui cont în aplicație

Acesta va trebui să completeze o adresă de mail validă, cât și o parolă formată din cel puțin opt caractere, o cifră și un caracter special. Autentificarea este refuzată în cazul nerespectării acestor criterii. În oricare din aceste cazuri, vor apărea mesajele aferente erorilor pentru o ușoară remediere a problemei.

### 4.2 Completarea profilului

După prima conectare în aplicație, indiferent de modul de realizare al contului, utilizatorul va fi redirecționat automat către o pagina în care va completa date precum: nume, prenume, vârstă, adresă. Iar în funcție de tipul de cont ales, cultivatorul va completa numele companiei și suprafața livezii, iar angajații vor completa datele despre calificarea lor profesională.

**Funcție**

**Suprafață**

**Nume companie**

Fig 4.2 – Secțiune profil cultivator

**Funcție**

**Permis de conducere**

**Categorie permis**  
☐ B ☐ B1 ☐ C1 ☐ C ☐ D1 ☐ D

Fig 4.3 – Secțiune profil angajat

## 4.3 Perspectiva cultivatorului

Proprietarii de livezi adulte de meri pot alege aplicația “Măruleț” pentru a gestiona și ține evidența operațiunilor care se fac pe parcursul anului în livada lor în funcție de anumiți factori, cum ar fi: temperatură, precipitații, problema care se dorește a fi tratată. De asemenea, prin intermediul acesteia, pot angaja persoane cu diferite calificări profesionale cu scopul de a le atribui operațiuni din livada lor.

Pagina principală afișată imediat după conectare conține vremea din ziua curentă, cât și următoarea operațiune ce trebuie efectuată, ulterioară orei curente.

**Măruleț** Profil Livada mea Anunțuri postate Forum

Bine ai venit, Silvia Andronic Logout

**Vremea la 25-8-2021 22:35**

Temperatura: 23.04 °C  
Temp. resimțita: 23.18 °C  
Nori: 0 %  
Umiditate: 68 %  
Precipitații: cer senin

Vremea →

**Stropit acarieni**

Data: 25-08-2021  
Angajat: Andronic Alexandru  
Utilaj: Tractor v4  
Substanțe utilizate: Chorus50  
Doza: 1.49kg

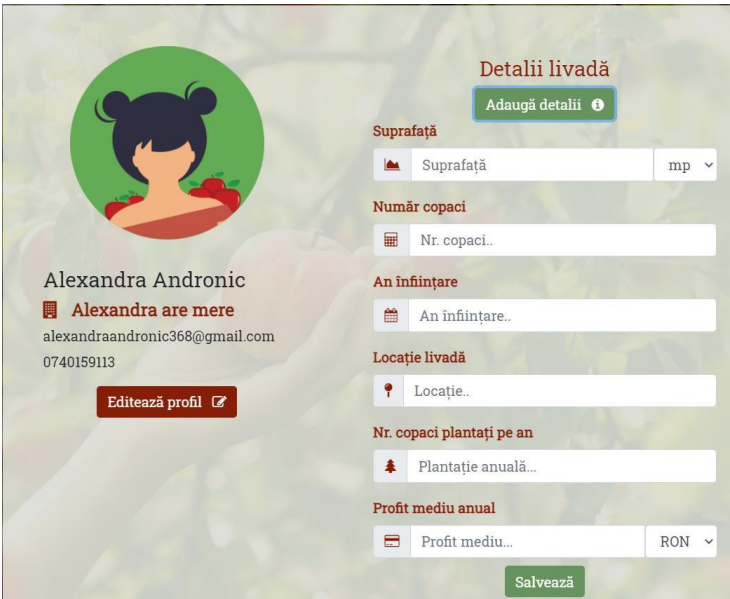
Vezi operațiuni

Fig 4.4 – Pagina principală utilizator de tip cultivator



### 4.3.1 Profil

În pagina dedicată profilului, datele adăugate la prima conectare pot fi vizualizate, dar și modificate prin apăsarea butonului **Editează profil**. Va apărea un formular ce conține datele existente, care pot fi modificate. Dacă informațiile introduse sunt valide, acestea vor fi actualizate și afișate conform modificărilor făcute. Tot în această pagină, utilizatorul poate adăuga date suplimentare despre livada sa, cum ar fi: profitul mediu dintr-un an, soiurile cultivate sau numărul pomilor, prin apăsarea butonului **Adaugă detalii**.



**Detalii livadă**  
Adaugă detalii ⓘ

**Suprafață**  
Suprafață mp ▾

**Număr copaci**  
Nr. copaci..

**An înființare**  
An înființare..

**Locație livadă**  
Locație..

**Nr. copaci plantați pe an**  
Plantație anuală...

**Profit mediu anual**  
Profit mediu... RON ▾

Salvează

Fig 4.5 – Profil utilizator conectat de tip cultivator

### 4.3.2 Vremea și programarea operațiunilor

Vremea reprezintă factorul cel mai important care influențează perioada în care se pot executa toate operațiunile în domeniul agricol. Astfel că, utilizatorul are acces la datele despre vreme chiar din pagina principală. Secțiunea este dedicată datelor privind vremea din ziua curentă. Prin accesarea butonului **Vreme**, utilizatorul va fi redirecționat către pagina în care sunt afișate datele pentru vremea pe 8 zile. Aici se pot vizualiza informații despre probabilitatea precipitațiilor, dar și avertizări privind fenomenele extreme.





Fig 4.6 – Informațiile despre datele meteorologice

Prin apăsarea butonului **Programează tratament**, utilizatorul este redirecționat către pagina în care vor fi programate operațiunile dorite: tratamente, operațiuni pentru sol, irigații sau fertilizări. În această pagină sunt vizibile operațiunile programate în ziua curentă, toate operațiunile ulterioare datei curente, dar și cele finalizate. În cazul în care angajatul atribuit acestei operațiuni va seta statusul ca fiind „Finalizat”, aceasta nu va trece direct în secțiunea de **Operațiuni finalizate**. Va ajunge în tabelul cultivatorului, având butonul din coloana Finalizat cu un design diferit (un ceas), însemnând că aceasta necesită o aprobare din partea cultivatorului pentru a fi declarată ca fiind o operațiune finalizată. De asemenea, pentru o utilizare mai ușoară, ambele tipuri de butoane conțin informații auxiliare despre evenimentul declanșat la click pe fiecare buton.

Pentru a seta statusul unei operațiuni în “Finalizat”, se face click pe butonul din coloana **Finalizat**, în acest moment, operațiunea va dispărea din secțiunea operațiunilor programate și va fi vizibilă în **Operațiuni finalizate**. De asemenea, acestea pot fi eliminate și din această secțiune, însă prin această operațiune, nu va mai putea fi accesată nicio informație despre aceasta.

Operațiune	Data	Ora	Angajat	Finalizat
Stropit acarieni	2021-08-29	21:00	Nicusor Andronic	
Arat	2021-08-29	23:00	Nicusor Andronic	

Fig 4.7 – Pagină programare operațiune

Pentru adăugarea unei operațiuni, se face click pe butonul **Programează operațiune** ce va deschide un formular de tip pop-up. Completarea câmpurilor se va face în mod progresiv, afișarea unor câmpuri făcându-se în funcție de valoarea câmpurilor anterioare. Nu vor putea fi programate operațiuni înainte de data sau ora curentă accesării aplicației. De asemenea, dacă temperaturile din ziua respectivă vor depăși 25 de grade sau vor fi sub 0 grade Celsius, vor apărea avertizări în acest sens pentru a se evita programarea unei operațiuni într-o zi care nu favorizează realizarea acesteia. Un alt criteriu este cel al precipitațiilor, dacă în ziua programării sunt anunțate precipitații, utilizatorul va primi de asemenea avertizări.

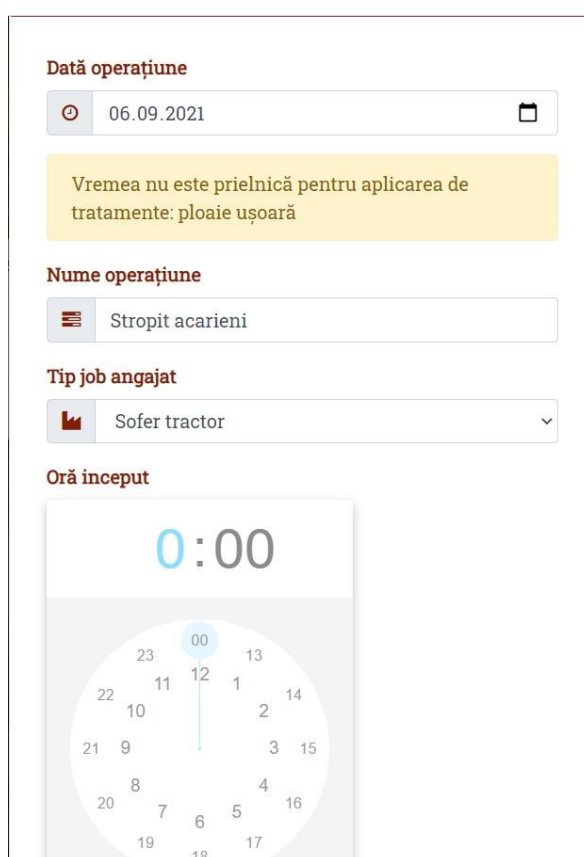


Fig 4.8 – Primii pași pentru programare operațiune

Dacă tipul operațiunii va fi **Tratament** utilizatorul va trebui să aleagă stadiul de dezvoltare în care se află fructul, respectiv problema pe care dorește să o trateze. În funcție de valorile acestor câmpuri, vor fi afișate doar produsele corespunzătoare stadiului de dezvoltare și problemei. Doza de substanță va fi calculată automat în funcție de alegerea produsului și suprafața tratată. Astfel utilizatorul va ști exact ce cantitate de produs trebuie aplicată pentru a respecta programul tehnologic.

La fiecare pas, utilizatorul are opțiunea de renunțare a programării operațiunii prin click pe butonul **Renunță**. Butonul pentru salvarea detaliilor va apărea doar în momentul completării tuturor detaliilor necesare.

The screenshot shows a web form for configuring a task. It contains the following elements:

- Tip operațiune**: A dropdown menu with a list icon and the selected value "Tratament".
- Durată operațiune (h)**: A text input field with a clock icon and the value "1".
- Stadiul de dezvoltare**: A dropdown menu with a list icon and the selected value "Urechuse".
- Problemă**: A dropdown menu with a list icon and the selected value "Rapan".
- Substanțe recomandate pentru tratament**: A dropdown menu with a list icon and the selected value "Chorus 50".
- Doza calculată este:** 1.13 kg
- Caută utilaj**: A blue button.
- ...**: An ellipsis indicating more options.
- Renunță**: A blue button.

Fig 4.9 – Alegerea tipului de tratament

Indiferent de tipul operațiunii, utilizatorul va trebui să aleagă utilajul cu care va fi efectuat acest task. Cultivatorul va putea alege dintr-o listă de utilaje disponibile la acel moment de timp, care nu mai sunt atribuite și altor sarcini la momentul de timp ales. Utilajele listate vor fi cele adăugate în modulul de facturi. În funcție de utilajul ales, vor apărea doar angajații ce dețin categoria de permis de conducere necesară pentru utilajul ales și care de asemenea, nu sunt distribuiți pentru alte operațiuni. Pentru reprogramarea unui task este necesară marcarea sa ca fiind finalizat, după care ștergerea din secțiunea de **Operațiuni finalizate** sau eliminarea acestuia din secțiunea cu toate programările ulterioare datei curente.



Fig 4.10 – Alegerea utilajului respectiv a angajatului

### 4.3.3 Anunțuri angajare

Cultivatorul poate veni în sprijinul comunității sale prin publicarea de diferite locuri de muncă în domeniul pomiculturii. Acesta poate adăuga job-uri având diferite criterii de acceptare al angajaților. Pagina principală conține locurile de muncă adăugate de cultivator. Adăugarea unui anunț nou se face prin click pe butonul **Adaugă anunț**.

**Anunt nou**

---

**Nume post**

Sofer tractor ▼

**Scurta descriere**

Descriere

**Permis de conducere**

DA ▼

☐ B ☐ B1 ☐ C1 ☐ C ☐ D1 ☐ D

**Localitate**

Localitate

**Salveaza**

Fig 4.11 – Formular adăugare nou job

În secțiunea **Cereri** utilizatorul poate vedea cererile diferitelor persoane pentru un anumit job postat de acesta. Prin apăsarea butonului **Acceptă**, se va deschide o fereastră în care cultivatorul va trebui să introducă oferta salarială, în caz contrar, prin apăsarea butonului

**Respingere cerere**, acesta va trebui să specifice motivului respingerii. În momentul schimbării statusului cererii, aceasta va dispărea din această secțiune.

În secțiunea **Oferte acceptate** cultivatorul poate vedea angajații, și anume persoanele care au acceptat oferta salarială transmisă de acesta la apăsarea butonului **Acceptă**.



Fig 4.12 - Cererile pentru angajare

#### 4.3.4 Forum

Interacțiunea cu restul cultivatorilor este importantă pentru a putea împărtăși și cere sfaturi în legătură cu subiectele din pomicultură. Utilizatorul poate posta diferite topicuri pe forum, la care poate atașa imagini, dar îi poate atribui inclusiv o categorie din care face parte. Adăugarea se face apăsând butonul **Topic nou**, după care va apărea o fereastră în care se vor completa detaliile necesare postării topicului. Este obligatorie completarea campurilor, excepție făcând câmpul pentru încărcarea imaginii. Fiecare topic se desfășoară sub forma unei discuții în care participanții pot da like unui răspuns și pot de asemenea să adauge imagini în răspunsurile lor, accesarea conversației se face prin apăsarea butonului **Conversație**. Pagina principală oferă posibilitatea sortării postărilor în funcție de dată, crescător sau descrescător, dar și filtrarea acestora în funcție de categoria ce prezintă interes pentru utilizator. Inițial, toate topicurile sunt afișate.

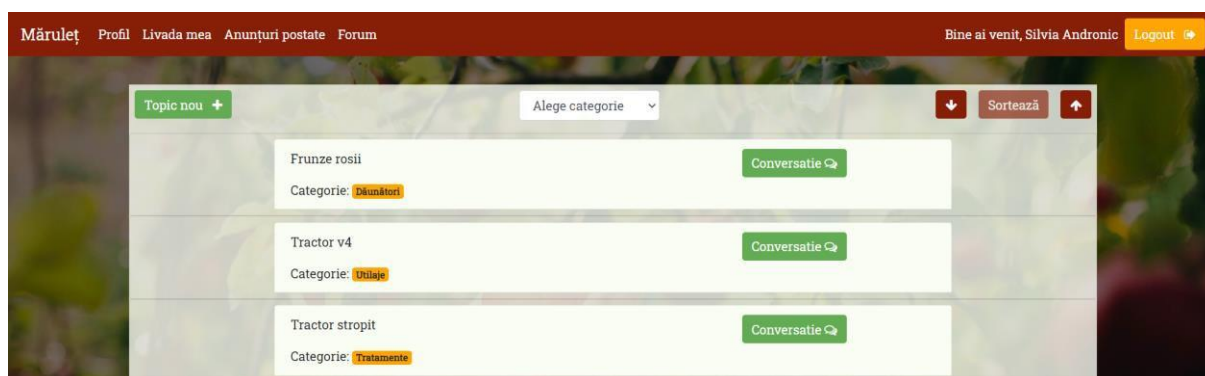


Fig 4.13 – Pagină principală forum

## 4.3.5 Livada mea

### 4.3.5.1 Statistici

Cultivatorul poate vizualiza în pagina **Statistici**, date despre cheltuielile făcute în anul curent. Pentru a vizualiza statisticile privind principalele cheltuieli în funcție de un an precizat, se poate completa câmpul vizibil în partea de sus a paginii. Acesta nu va permite introducerea unui an mai mare decât anul curent. Graficele se actualizează automat în momentul în care utilizatorul adaugă o factură.

Fig 4.14 - Câmp în care se completează anul pentru care se dorește generarea graficelor



Fig 4.15 – Pagina grafice pentru evidența investițiilor și angajaților

### 4.3.5.2 Facturi

Utilizatorul poate adăuga date despre achizițiile în materie de utilaje, substanțe și pomi. Această pagină este accesibilă din submeniul **Management**.

Pagina cuprinde 3 secțiuni: **Substanțe**, **Utilaje** și **Pomi**. În fiecare dintre ele utilizatorul poate înregistra o factură (achiziție). Odată adăugată, valoarea acesteia va fi adăugată la totalul facturilor de acel tip. De asemenea, utilizatorul poate șterge o factură din tabel prin apăsarea butonului din coloana **Șterge**, actualizându-se iarăși totalul. Se poate descărca un fișier PDF prin apăsarea butonului din coloana **Detalii** în vederea listării acesteia de exemplu.

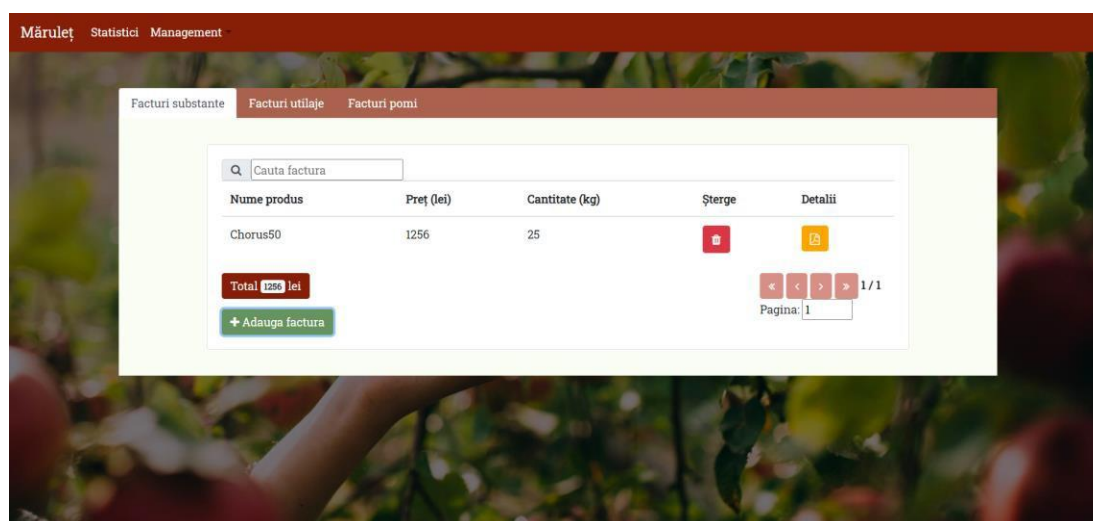




Fig 4.16 – Pagină facturi substanțe

Utilizatorul poate de asemenea să aleagă unitățile de măsură și valuta în care dorește să adauge cantitățile și prețul, afișarea acestora făcându-se însă în KG respectiv RON după convertirea aferentă.

Pe pagină vor fi vizibile implicit doar câte cinci facturi, vizualizarea tuturor făcându-se prin folosirea săgeților din partea de jos. O altă modalitate de navigare între pagini este introducerea numărului de pagină în câmpul **Pagină**.

## 4.4 Perspectiva angajatului

Pagina principală a utilizatorului de tip angajat conține operațiunile ce îi sunt atribuite și care necesită finalizare în ziua curentă.

După apăsarea butonului **Finalizat**, operațiunea atribuită dispare din pagina principală vizibilă angajatului.

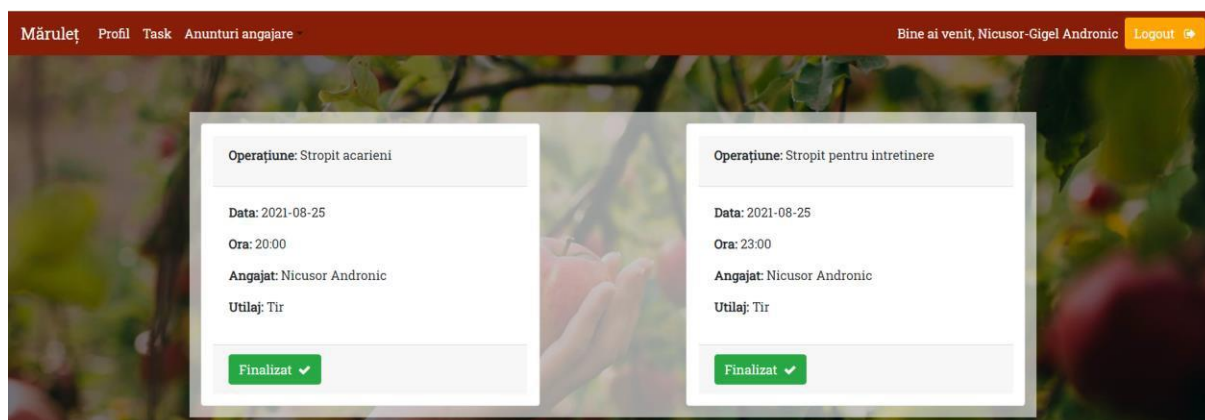


Fig 4.17 – Pagina principală utilizator de tip angajat

### 4.4.1 Aplicarea pentru un job

Ca și persoană ce nu are drept de cultivator în aplicație, utilizatorul de tip angajat poate doar vizualiza anunțurile publicate de diferiți cultivatori, din orice zonă, dar și operațiunile ce îi sunt atribuite. În submeniul **Anunțuri publicate** din **Anunțuri angajare**, acesta poate vedea și aplica, dacă îndeplinește condițiile necesare, pentru locurile de muncă disponibile. De asemenea, dacă dorește filtrarea acestora în funcție de zonă în care dorește să lucreze, poate



face acest lucru prin introducerea locației în câmpul **Locație**. În mod implicit, vor apărea toate anunțurile, din orice locație.

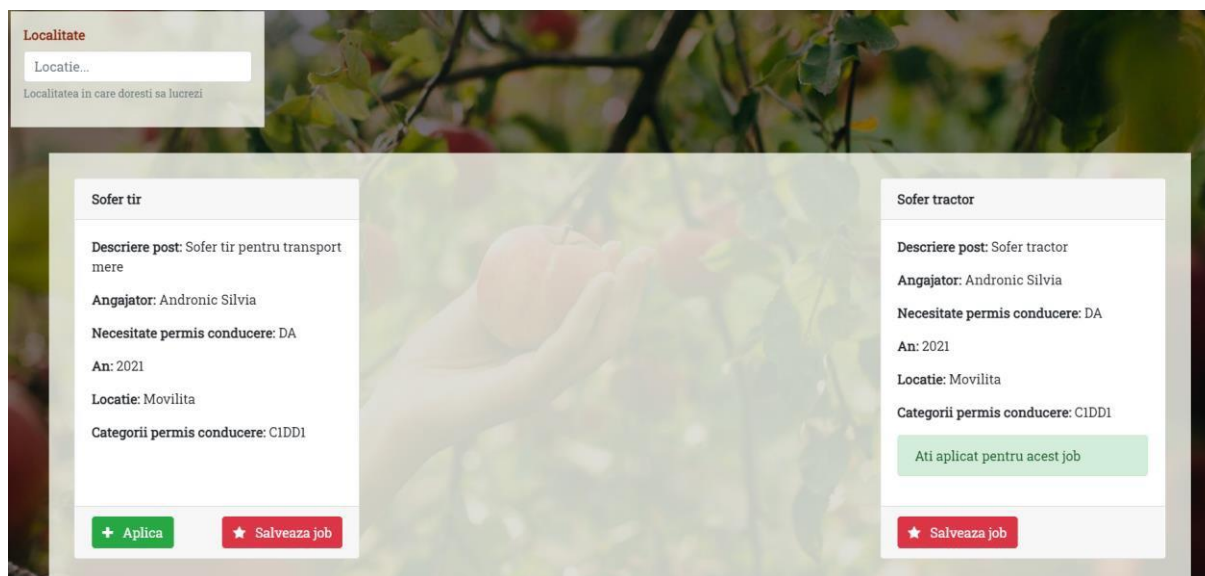


Fig 4.18 – Pagină anunțuri publicate

Anunțurile sunt dispuse sub forma unor carduri, având o descriere minimală a job-ului. Butonul care permite aplicarea pentru un loc de muncă, va apărea doar în cazul în care caracteristicile profilului corespund cerințelor angajatorului, în caz afirmativ, va apărea un buton **Aplică**. Dacă acesta este apăsat, va fi afișată o fereastră cu un mesaj de confirmare. Aplicarea de doua ori pentru același job nu este posibila, butonul **Aplică** va dispărea după prima apăsare a acestuia.

## 4.4.2 Acceptarea ofertelor

După urmarea fluxului de aplicare pentru un loc de muncă în care cererea ajunge la angajator, iar acesta o poate accepta sau respinge, aceasta ajunge înapoi la aplicant. În cazul unui accept din partea angajatorului, următorul pas este acela al acceptării sau respingerii ofertei salariale.

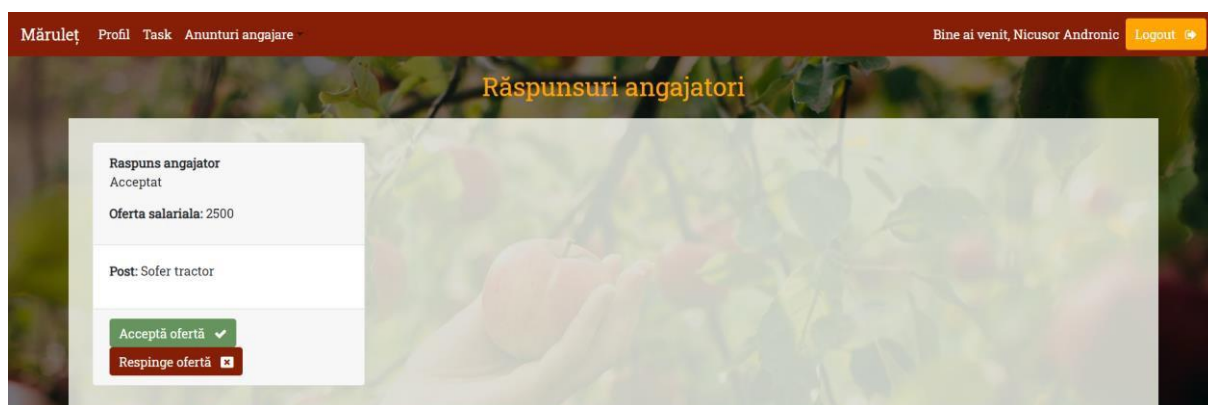


Fig 4.19 – Cereri acceptate

## 5. Concluzii

Aplicația Măruleț este o aplicație destinată domeniului pomicol dezvoltată pentru a ajuta agricultorii din acest domeniu, deoarece acesta este un domeniu în care oamenii trebuie să acționeze împreună. Introducerea funcționalităților de publicare de locuri de muncă, dar și cea de forum își propune crearea unei comunități care sprijină tehnologizarea acestei arii. În acest mod se pot conecta persoane cu diferite experiențe în domeniu și astfel agricultura din Romania poate deveni un subiect de interes inclusiv pentru generația tânără, acesta reprezentând un obiectiv pe care aș dori să-l ating prin dezvoltarea unei aplicații de acest tip.

O provocare întâmpinată a fost cea a trasării unui flux al aplicației. Fiind un domeniu foarte vast și într-o continuă schimbare, am dorit ca funcționalitățile oferite utilizatorilor să fie unele standard, care să rămână de actualitate indiferent de modificările pe care aceștia le pot aduce modului în care își desfășoară activitatea pe teren.

Aș dori să extind aplicația în mai multe direcții, câteva din ele fiind cele axate pe comunicarea dintre cultivator și angajat, dar și îmbunătățirea experienței pe care cultivatorul o are.

Adăugarea unei secțiuni în care cultivatorul va putea vizualiza o hartă actualizată automat în funcție de locația sa la momentul accesării aplicației pentru a putea găsi mult mai ușor magazinele de specialitate în vederea achiziționării materialelor necesare. De asemenea, o dezvoltare auxiliară este aceea a unui chat ce permite comunicarea mult mai facilă între cultivator și angajat.

# Bibliografie

- [1] „Reinvently”: <https://reinvently.com/blog/fundamentals-web-application-architecture/>. [Accesat 01-08-2021].
- [2] „Hectre,”: <https://hectre.com/>. [Accesat 01-08-2021].
- [3] „Farmable”: <https://www.farmable.tech/>. [Accesat 02-08-2021].
- [4] „Vikingcodeschool”: <https://www.vikingcodeschool.com/html5-and-css3/html5-semantic-tags>. [Accesat 03-08-2021].
- [5] „Bootstrap”: <https://getbootstrap.com/>. [Accesat 04-08-2021].
- [6] „React-Bootstrap”: <https://react-bootstrap.github.io/>. [Accesat 04-08 -2021].
- [7] „ReactJS Docs”: <https://reactjs.org/docs/getting-started.html>. [Accesat 05-08-2021].
- [8] „Virtual DOM”: <https://reactjs.org/docs/faq-internals.html>. [Accesat 05-08-2021].
- [9] „JSX”: <https://reactjs.org/docs/introducing-jsx.html>. [Accesat 05-08-2021].
- [10] „JSX in depth”: <https://reactjs.org/docs/jsx-in-depth.html>. [Accesat 06-08-2021].
- [11] „Hook state”: <https://reactjs.org/docs/hooks-state.html>. [Accesat 07-08-2021].
- [12] „Hooks reference”: <https://reactjs.org/docs/hooks-reference.html#userref>. [Accesat 07-08-2021].
- [13] „Effect hook”: <https://reactjs.org/docs/hooks-reference.html#useeffect>. [Accesat 08-08-2021].
- [14] „Madooei”: [https://madooei.github.io/cs421\\_sp20\\_homepage/client-server-app/](https://madooei.github.io/cs421_sp20_homepage/client-server-app/). [Accesat 10-08-2021].
- [15] „Redhat”: <https://www.redhat.com/en/topics/cloud-computing/public-cloud-vs-private-cloud-and-hybrid-cloud>. [Accesat 10-08-2021].
- [16] „Firebase”: <https://firebase.google.com/>. [Accesat 11-08-2021].
- [17] „Firebase pricing”: <https://firebase.google.com/pricing>. [Accesat 12-08-2021].
- [18] „Scylladb”: <https://www.scylladb.com/resources/nosql-vs-sql/>. [Accesat 13-08-2021].
- [19] „Firebase Docs”: <https://firebase.google.com/docs/firestore/manage-data/add-data#node.js> . [Accesat 15-08-2021].
- [20] „Firebase Docs”: <https://firebase.google.com/docs/firestore/query-data/listen>. [Accesat 15-08-2021].
- [21] „Environment Variables”: <https://nextjs.org/docs/basic-features/environment-variables>. [Accesat 18-08-2021].
- [22] „Firebase Docs” : <https://firebase.google.com/docs/auth/web/start>. [Accesat 20-08-2021].

- [23] „ReactJS React Context”: <https://reactjs.org/docs/context.html>. [Accesat 19-08-2021].
- [24] „JS Doc”:  
[https://developer.mozilla.org/enUS/docs/Web/JavaScript/Reference/Global\\_Objects/Promise](https://developer.mozilla.org/enUS/docs/Web/JavaScript/Reference/Global_Objects/Promise).  
[Accesat 21-08-2021].
- [25] „Firebase Google Auth”: <https://firebase.google.com/docs/auth/web/google-signin?hl=ro>.  
[Accesat 25-08-2021]
- [26] „Firebase hash password”: <https://firebaseopensource.com/projects/firebase/scrypt/>. [Accesat 25-08-2021].
- [27] „ReactJS Components”: <https://reactjs.org/docs/components-and-props.html>. [Accesat 26-08 - 2021].
- [28] „Reactrouter”: <https://reactrouter.com/web/api/Hooks>. [Accesat 25-08-2021].
- [29] „Firebase Storage”: <https://firebase.google.com/docs/storage/web/upload-files>. [Accesat 25-08 - 2021].
- [30] „Syngenta”: <https://www.syngenta.ro/program-tehologic-mar>. [Accesat 25-08-2021].
- [31] „npmjs”: <https://www.npmjs.com/>. [Accesat 25-08-2021].
- [32] „React Forms”: <https://react-bootstrap.github.io/components/forms/>. [Accesat 26-08-2021].
- [33] „React Modal”: <https://react-bootstrap.github.io/components/modal/>. [Accesat 26-08-2021].
- [34] „React-table”: <https://react-table.tanstack.com/docs/overview>. [Accesat 26-08-2021].
- [35] „Openweathermap”: <https://openweathermap.org/api/one-call-api>. [Accesat 24-08-2021].
- [36] „ReactJs Fetch”: <https://reactjs.org/docs/faq-ajax.html>. [Accesat 24-08-2021].
- [37] „CSS Media Query”:  
[https://developer.mozilla.org/enUS/docs/Web/CSS/Media\\_Queries/Using\\_media\\_queries](https://developer.mozilla.org/enUS/docs/Web/CSS/Media_Queries/Using_media_queries).  
[Accesat 24-08-2021].

## Lista figuri

1. Fig 1.1 - Pagina autentificare din aplicația “Măruleț”
2. Fig 1.2 - Pagina forum
3. Fig 1.3 - Arhitectura unei aplicații web [1]
4. Fig 2.1 - Generarea unei componente de tipul unei funcții în React cu ajutorul Code Snippets
5. Fig 2.2 - Structura unei pagini din HTML4 vs HTML5 [4]

6. Fig 2.3 - Exemplu de componentă în React
7. Fig 2.4 - Cod executat de Babel pentru afișarea pe ecran
8. Fig 2.5 - Definiția funcției createElement [10]
9. Fig 2.6 - Arhitectură de tip Client – Server [14]
10. Fig 2.7 - Tipurile de Cloud [15]
11. Fig 2.8 - Baze de date relaționale vs non-relaționale [18]
12. Fig 2.9 - Exemplu colecție și documente din Firebase Database
13. Fig 3.1 - Structurarea fișierelor
14. Fig 3.2 - Configurare proiect în Firebase
15. Fig 3.3 - Configurarea Firebase în aplicație folosind variabilele din fișierul .env.local [22]
16. Fig 3.4 - Vizualizare meniu proiect Firebase
17. Fig 3.5 - Meniul pentru activarea diferitelor modalități de autentificare
18. Fig 3.6 - Fluxul unei promisiuni în JavaScript [24]
19. Fig 3.7 - Componenta de autentificare în aplicație
20. Fig 3.8 - Conectare folosind contul Google
21. Fig 3.9 - Pagina NotFound
22. Fig 3.10 - Transmiterea de date în mod explicit între componente [25]
23. Fig 3.11 - Input de tip fișier
24. Fig 3.12 - Câmpuri din formularul pentru adăugare nou anunț angajare
25. Fig 3.13 - Colecția cu produsele și probleme pe care le tratează
26. Fig 3.14 - Listarea utilajelor disponibile
27. Fig 3.15 - Exemplu validare formular adăugare profil
28. Fig 3.16 - Exemplu fereastră Modal pentru adăugarea de răspuns în Forum
29. Fig 3.17 - Exemplu componentă Table cu paginare și filtrare
30. Fig 3.18 - Apel API OpenWeather
31. Fig 3.19 - Pagină vreme vizualizată de pe laptop
32. Fig 3.20 - Pagină vreme vizualizată de pe mobil
33. Fig 4.1 - Formular pentru înregistrarea unui cont în aplicație
34. Fig 4.2 - Secțiune profil cultivator
35. Fig 4.3 - Secțiune profil angajat
36. Fig 4.4 - Pagina principală utilizator de tip cultivator
37. Fig 4.5 - Profil utilizator conectat de tip cultivator
38. Fig 4.6 - Informațiile despre datele meteorologice

- 39. Fig 4.7 - Pagină programare operațiune
- 40. Fig 4.8 – Primii pași pentru programare operațiune
- 41. Fig 4.9 – Alegerea tipului de tratament
- 42. Fig 4.10 – Alegerea utilajului respectiv a angajatului
- 43. Fig 4.11 - Formular adăugare nou job
- 44. Fig 4.12 - Cererile pentru angajare
- 45. Fig 4.13 - Pagină principală forum
- 46. Fig 4.14 - Câmp în care se completează anul pentru care se dorește generarea  
graficelor
- 47. Fig 4.15 - Pagina grafice pentru evidenta investițiilor și angajaților
- 48. Fig 4.16 - Pagină facturi substanțe
- 49. Fig 4.17 - Pagina principala utilizator de tip angajat
- 50. Fig 4.18 - Pagină anunțuri publicate
- 51. Fig 4.19 - Cereri acceptate