

UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI
FACULTATEA DE INFORMATICĂ



LUCRARE DE LICENȚĂ

**Sistem Web PWA pentru managementul sarcinilor unei
echipe via GraphQL**

propusă de

Theodor-Ioan Spanțu

Sesiunea: *iunie/iulie, 2021*

Coordonator științific

Conf. dr. Sabin-Corneliu Buraga

UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI
FACULTATEA DE INFORMATICĂ

Sistem Web PWA pentru managementul sarcinilor unei echipe via GraphQL

Theodor-Ioan Spanțu

Sesiunea: *iunie/iulie, 2021*

Coordonator științific

Conf. dr. Sabin-Corneliu Buraga

Contents

1	Tehnologii folosite.....	4
1.1	Biblioteca <i>React</i>	4
1.2	GraphQL.....	5
2	Prezentarea aplicației.....	9
2.1	Studiu de caz	9
2.2	Cerințele aplicației.....	11
2.2.1	Modul Online	11
2.2.2	Modul Offline.....	11
3	Analiză și proiectare	13
3.1	Structuri de date.....	13
3.2	Interacțiunea cu utilizatorul	14
3.3	Descriere API	15
4	Implementare	17
4.1	<i>GraphQL</i>	17
4.1.1	Schema	17
4.1.2	Trimiterea cererilor	19
4.2	Cache dinamic	20
4.2.1	Tratarea interogărilor.....	20
4.2.2	Tratarea mutațiilor.....	21
4.2.3	Probleme apărute la implementare și rezolvări ale acestora	23
5	Manual de utilizare	25
6	Concluzii și idei de viitor	31

Introducere

Motivație

Pentru a înțelege mai bine motivația din spatele aplicației, să luăm următorul exemplu. Avem o echipă cu un număr de sarcini de îndeplinit pentru întocmirea unui proiect sau a unei teme ce constă în crearea unei aplicații. Dorim ca fiecărui membru să îi fie asignat un anumit număr de sarcini de realizat într-un anumit interval orar.

Coordonarea unei astfel de echipe poate fi foarte costisitoare fără o unealtă care să faciliteze acest proces. Problemele pot apărea mai ales atunci când membrii echipei despre care vorbim sunt îndepărtați și comunicarea între aceștia nu se poate face în mod direct. Majoritatea acestor probleme se rezumă la faptul că membrii echipei nu pot vizualiza în timp real stadiul în care se află proiectul.

Aplicația propusă rezolvă toate aceste probleme într-un mod foarte interactiv și eficient în ceea ce privește timpul de înțelegere. Principalele sale caracteristici fiind flexibilitatea, performanța, experiența oferită utilizatorilor conectați atât de pe telefonul mobil cât și de pe laptopul sau calculatorul personal și funcționalitatea cu sau fără acces la internet.

Contribuții personale

Bineînțeles, mai există aplicații ce au același scop și se ocupă de aceleași problemele enumerate în rândurile de mai sus. În următoarele rânduri vom discuta pe scurt despre ce aduce această aplicație în plus față de cele deja existente.

Aplicația propusă este un produs de tip aplicație web progresivă, deci acesta poate fi folosit atât din *browser* cât și instalat, la alegerea utilizatorului. În plus aplicația promite un mod de utilizare fără conexiune la internet ce constă în următoarele:

- opțiunea de a vedea toate paginile pe care a intrat utilizatorul în ultimul timp, înainte de pierderea conexiunii;
- opțiunea de a fi trimis pe o pagină specială când se încearcă accesarea unei pagini ce nu este salvată, pentru a îmbunătăți experiența utilizatorului;

- opțiunea de a face orice modificare ar fi posibilă și în modul online. Aceste modificări vor fi tratate când utilizatorului îi revine conexiunea la internet.

Despre acest mod vom vorbi mai detaliat în capitolul următor, mai exact în secțiunea **Modul Offline**.

Structura lucrării

Lucrarea este formată, după cum se poate observa și din cuprins, din 7 capitole, cu titluri sugestive, în fiecare dintre ele abordându-se o temă diferită.

- În capitolul **Tehnologii folosite** vom vorbi despre tehnologiile utilizate în realizarea aplicației și justificarea alegerilor făcute.
- În capitolul **Prezentarea aplicației** vom face un scurt studiu de caz și vom discuta într-un mod mai riguros decât în scurta introducere făcută anterior, despre cerințele aplicației și funcționalitățile pe care le oferă aceasta.
- În capitolul **Analiză și proiectare** vom vorbi despre alegerea bazei de date, despre diferite principii ce îmbunătățesc experiența utilizatorilor și despre structura anumitor module.
- În capitolul **Implementare** vom vorbi despre modul ales de implementare a diferitor funcționalități, despre problemele apărute pe parcurs și rezolvări ale acestora. În acest capitol putem găsi și câteva fragmente de cod preluate din aplicație.
- În capitolul **Manual de utilizare** vom descrie cum poate un utilizator folosi aplicația.
- În capitolul **Concluzii și idei de viitor** vom vorbi despre ideile finale și direcțiile în care se va îndrepta aplicația.

1 Tehnologii folosite

În acest capitol vom vorbi despre tehnologiile utilizate în dezvoltarea aplicației, motivul alegerii acestora în locul altor tehnologii asemănătoare și proprietățile pe care le aduc acestea aplicației noastre.

1.1 Biblioteca *React*

Ce este *React*? *React* este o bibliotecă *JavaScript*, creată și întreținută de compania *Facebook*, ideală pentru dezvoltarea de site-uri web și mai ales de *single page applications*, principala sa funcționalitate fiind crearea de componente reutilizabile.

În cele ce urmează, vom vedea de ce este această bibliotecă optimă pentru aplicația noastră. Pentru început ar trebui să ne întrebăm, în contextul acestei aplicații de ce am folosi o bibliotecă/un *framework* în general?

Această bibliotecă cât și alte *framework*-uri asemănătoare acesteia, cum ar fi *Angular* și *Vue.js* sunt o alegere excelentă pentru majoritatea aplicațiilor web, totuși aplicația propusă prezintă o nevoie sporită pentru o asemenea tehnologie, în special datorită interfeței destul de repetitive.

Să presupunem că avem o mulțime de liste, fiecare dintre acestea are în interiorul său o mulțime de sarcini și în cele din urmă o mulțime de utilizatori într-o echipă, situație similară celei din *Figura 21* de la pagina 28. Datorită tehnologiilor de genul acesta putem scrie cod pentru o singură componentă și să îl refolosim pentru toate cele asemănătoare acesteia, schimbând doar datele într-un mod dinamic, astfel în exemplul prezentat avem nevoie doar de trei componente, o componentă care să reprezinte listele, una care să reprezinte sarcinile, și una care să reprezinte utilizatorii.

O altă întrebare destul de interesantă pe care am putea să o punem este „De ce am putea considera biblioteca *React* o tehnologie mai potrivită decât celelalte numite mai sus pentru o astfel de aplicație?” După cum am menționat și în capitolul anterior această aplicație este de tipul aplicație web progresivă, de aceea trebuie să salvăm într-un depozit paginile primite de la *server*, pentru a le putea afișa în momentul pierderii conexiunii. Aici, într-o anumită măsură, *React* este foarte util datorită faptului că acesta permite, la crearea unei noi aplicații, selectarea opțiunii -- *template cra-template-pwa*. Cu această opțiune, proiectului nostru i se vor crea automat două

fișiere *JavaScript*, vitale pentru orice *PWA*, un *service worker* și un fișier responsabil cu înregistrarea sa. Odată înregistrat *service worker*-ul, acesta va salva automat conținutul static de pe paginile ulterior create de noi.

Alte două motive puțin mai generale pentru folosirea bibliotecii, dar raportate tot la aplicația noastră, sunt compatibilitatea cu celelalte tehnologii folosite, aplicația fiind realizată utilizând populara suită de tehnologii *MERN* și multitudinea de pachete specifice disponibile pentru dezvoltarea aplicațiilor. Exemple de astfel de pachete:

- *Apollo-client* și *Apollo-boost* prin intermediul cărora se realizează comunicarea cu serverul Node.js folosindu-se de interogări *GraphQL*.
- *React-beautiful-dnd*, un pachet extrem de relevant pentru o aplicație ca aceasta, întrucât îmbunătățește considerabil experiența utilizatorilor. În aplicația noastră, am folosit acest pachet pentru a adăuga funcționalitatea de *drag and drop* cardurilor, astfel utilizatorii pot schimba poziția cardurilor în listele de carduri după bunul plac.

1.2 GraphQL

Pentru început să vorbim în câteva rânduri despre ce este *GraphQL*. *GraphQL* este un limbaj de interogare pentru *API*-uri, dezvoltat de compania *Facebook* în anul 2012 și lansat public în anul 2015. Un *API GraphQL* diferă de unul *REST* prin următoarele:

- Numărul scăzut de rute pe care acesta îl prezintă.
Un *API GraphQL* folosește, în majoritatea cazurilor, o singură rută pe care primește diferitele cereri de preluare sau procesare a datelor, fără a depinde în vreun fel de tipul de date folosit de acestea, în timp ce un *API REST* necesită o rută pentru fiecare tip de date în parte.

De exemplu, aplicația noastră folosește mai multe tipuri de date, trei dintre acestea fiind: tablele, sarcinile și utilizatorii. Pentru a prelua informația despre oricare dintre acestea voi face un apel la aceeași rută, care în cazul nostru este `/graphql`, un *API* de tip *REST* ar necesita cel puțin trei rute diferite pentru aceste operații: `/table`, `/sarcini`, `/utilizatori`.
- Numărul scăzut de verbe *HTTP* folosite.

Spre deosebire de un *API REST*, care prezintă o multitudine de verbe *HTTP*, cele mai importante fiind *GET*, *POST*, *PUT*, *DELETE*, *PATCH*. *API-ul GraphQL* prezintă posibilitatea folosirii a doar 2 verbe *HTTP*, care sunt aproape identice din punct de vedere funcțional, deci nu pot fi folosite concomitent. Este vorba de verbele *POST* și *GET*, alegerea verbului pe care dorim să îl folosim fiind doar o preferință.

Un exemplu de folosire a celor două verbe pentru aceeași interogare:

Dacă folosim verbul *POST* va trebui să adăugăm la *url-ul* cererii adresa */graphql* și la corpul său următorul obiect *JSON*.

```
{
  "query": "query aTest($arg1: String!) { test(who: $arg1) }",
  "operationName": "aTest",
  "variables": { "arg1": "me" }
}
```

Dacă folosim verbul *GET* va trebui să adăugăm la *url-ul* cererii următoarea adresă */graphql?query=query%20aTest(%24arg1%3A%20String!)%20%7B%20test(who%3A%20%24arg1)%20%7D&operationName=aTest&variables=%7B%22arg1%22%3A%22me%22%7D*.

Exemplu preluat de la <https://www.apollographql.com/docs/apollo-server/v1/requests/>

Am precizat că cele două verbe au „aproape” aceeași funcționalitate deoarece cea de-a două sintaxă nu permite folosirea mutațiilor. După cum vom vedea și în capitolele următoare aplicația noastră folosește doar verbul *POST*.

- Selecționarea datelor preluate.
GraphQL acordă dezvoltatorilor de aplicații, libertatea de a-și alege câmpurile pe care doresc să le preia încă de la trimiterea cererii spre *API*. Spre deosebire de un *API REST* care este mai restrictiv. De exemplu, un flux normal în cadrul unui *API REST* ar fi trimiterea cererii, primirea răspunsului ce conține și date nedorite, selectarea datelor.
- Suportul pentru datele imbricate.
O interogare *GraphQL* poate conține și date imbricate, acest lucru duce, practic, la un număr mai mic de cereri, astfel afectând într-un mod pozitiv performanța

aplicației. În rândurile ce urmează vom trece împreună și printr-un exemplu preluat din aplicație.

Unul din motivele principale pentru care am ales să folosesc *GraphQL* este performanța pe care o poate oferi acest limbaj de interogare a datelor prin asigurarea suportului pentru datele imbricate. Un exemplu mai concret este situația paginii unei table. Pe pagina aceasta sunt afișate în același timp următoarele câmpuri: listele de sarcini, sarcinile, utilizatorii. Toate aceste date fiind stocate în documente diferite din colecție. Dacă am alege să utilizăm un API de tip REST ar fi necesar să facem 3 apeluri diferite către server, lucru care nu poate decât să îngreuneze procesul și să afecteze performanța aplicației. Datorită *GraphQL* cu o singură solicitare putem aduce toate datele necesare în același timp, precum în exemplul din *Figura 1*.

```
const getBoardByIdQuery = gql`
  query($id: ID!){
    board(id: $id){
      id
      owner {
        id
      }
      users {
        id
      }
      taskLists {
        id
        tasks {
          id
          assignee {
            id
          }
        }
      }
    }
  }`
```

Figura 1 - Exemplu de interogare imbricată

Un alt motiv pentru care *GraphQL* este potrivit în cadrul acestei aplicații este dat de faptul că interogările conțin doar câmpurile dorite, practic nu mai este necesar ca rezultatul unei interogări să fie alcătuit din toate câmpurile prezente în tabelă/document, ci le putem selecta doar pe cele folosite. Lucru ce duce la transfer de date de dimensiuni mai mici între client și server, de unde rezultă și creșterea în performanță a aplicației. Tot în exemplul de mai sus putem observa cum alegem doar câmpul *id* pentru cele trei tipuri diferite de obiecte. Acesta nu este modul de

utilizare pe care l-am folosit în dezvoltarea aplicației, ci este un exemplu puțin simplificat pentru a înțelege mai bine noțiunile.

O altă funcționalitate oferită de *GraphQL* și o caracteristică extrem de relevantă în cazul acestei aplicații sunt subscripțiile și interogările repetate. Într-o echipa precum cea prezentată în Introducere, un demers perfect al lucrurilor ar implica ca fiecare membru al echipei să poată afla cât mai repede modul de desfășurare al proiectului și stadiul în care se află acesta. Din acest motiv și în situația aplicației noastre vom avea nevoie să primim informațiile despre table în timp real.

Pentru început să aflăm ce sunt și cum funcționează cele două metode. Subscripțiile și interogările repetate la anumite intervale de timp au aproape aceeași responsabilitate, îi conferă aplicației posibilitatea de a se desfășura în timp real. Totuși cele două metode au un mod extrem de diferit de funcționare.

Subscripțiile sunt legături permanente cu serverul *GraphQL*, ce permit serverului să notifice clientul de eventuale actualizări. Interogările repetate, după cum spune și numele acestora, sunt interogări care sunt executate în mod repetat la intervale de timp alese de dezvoltatorul aplicației.

Cu toate că ambele îndeplinesc aceeași funcție, acestea trebuie folosite în situații diferite. De exemplu subscripțiile ar trebui folosite în special în aplicații precum cele de mesagerie, unde ne dorim ca datele să se actualizeze cu orice schimbare din baza de date și conexiunea să persiste pe tot parcursul conversației. În schimb, dacă nu suntem atât de interesați de actualizarea instantanee a datelor, sau ne-am putea asuma această întârziere a lor pentru a obține un bonus de performanță, interogările repetate sunt alegerea corectă.

Luând în considerare aceste lucruri, în cadrul aplicației vom folosi interogările repetate la anumite intervale de timp.

2 Prezentarea aplicației

În acest capitol vom discuta despre cerințele aplicației și funcționalitățile pe care le prezintă și vom realiza un scurt studiu de caz pentru a înțelege mai bine situația actuală, făcând o comparație între aplicația propusă și unul din competitorii săi.

2.1 Studiu de caz

Problemele din acest subcapitol au fost prezentate într-un mod superficial și în capitolul Introducere, însă în cele ce urmează le vom detalia mult mai riguros prin prezentarea unei situații ce reflectă realitatea contemporană.

Un exemplu de aplicație ce rezolva problema expusă mai sus este, desigur, *Trello*. Aceasta este o aplicație foarte populară lansată în anul 2011, ce are ca scop creșterea productivității utilizatorilor săi prin folosirea interactivă a tablelor kanban.

Ce este o tablă kanban? Întrucât vom folosi destul de mult noțiunea de tablă kanban, ar fi o idee bună să aflăm mai întâi ce este aceasta. Tabla kanban este o unealta perfectă în special pentru administrarea și maximizarea eficienței proiectelor de tip agil. Tabla kanban este alcătuită dintr-o mulțime de liste, fiecărei liste urmând să i se atribue un anumit număr de sarcini și fiecare sarcină să fie atribuită unui membru al echipei. Tabla oferă posibilitatea de a vizualiza toate aceste elemente în timp real, membrilor echipei.

Care este secretul succesului din spatele aplicației? Acest produs a reușit să atragă suma sa extrem de mare de utilizatori datorită momentului potrivit al apariției, genul acesta de aplicații fiind doar niște idei la momentul respectiv. În plus au fost de ajutor și interfața sa interactivă, a sentimentului de recompensă oferit la terminarea unei sarcini, a fluxului de munca ușor de înțeles pentru noii utilizatori și a modelului lor de vânzare, aplicația fiind inițial gratis, utilizatorii urmând să plătească doar pentru subscripții speciale.

De ce este momentul acesta potrivit pentru crearea unei noi aplicații ce poate concura cu *Trello*? Având în vedere situația actuală, în urma pandemiei cu virusul SARS-CoV-2, tot mai multe firme aleg să își desfășoare activitatea ori într-un mod complet virtual, din confortul caselor personale, ori într-un mod hibrid, angajații urmând să vină la sedii doar în anumite zile ale săptămânii. Exemple de firme care au ales să facă această schimbare: *Microsoft*, *Twitter*, *Spotify*,

Amazon, Square etc. Ambele moduri au în comun faptul că angajații sunt din ce în ce mai îndepărtați unii de ceilalți și administrarea acestora devine din ce în ce mai dificilă, cu alte cuvinte administrarea activităților a tot mai multor echipe continuă să facă transferul spre mediul *online*.

De ce ar alege cineva altă aplicație? Produsul *Trello* prezintă o problemă destul de mare la nivelul structurii sale, care poate deveni mult prea complexă și greu de înțeles, într-un timp foarte scurt, stările sarcinilor fiind exprimate în general cu ajutorul listelor, numărul listelor va rămâne constant în timp ce numărul sarcinilor va continua să crească până în punctul în care ajunge să creeze probleme de înțelegere.

O altă problemă a produsului *Trello* este faptul că acesta nu funcționează odată cu pierderea conexiunii decât dacă utilizăm aplicația nativă, lucru pe care o parte din utilizatori obișnuiesc să îl evite. În plus, aplicația nativă *Trello* este mult mai costisitoare din punct de vedere al spațiului de stocare. Aplicația noastră, fiind un produs *PWA*, nu ocupă mult spațiu de stocare, din *Figura 2* putem observa diferența dintre cele două aplicații, *task-fuel* ocupând aproximativ 270kB, iar *Trello* aproximativ 40MB.

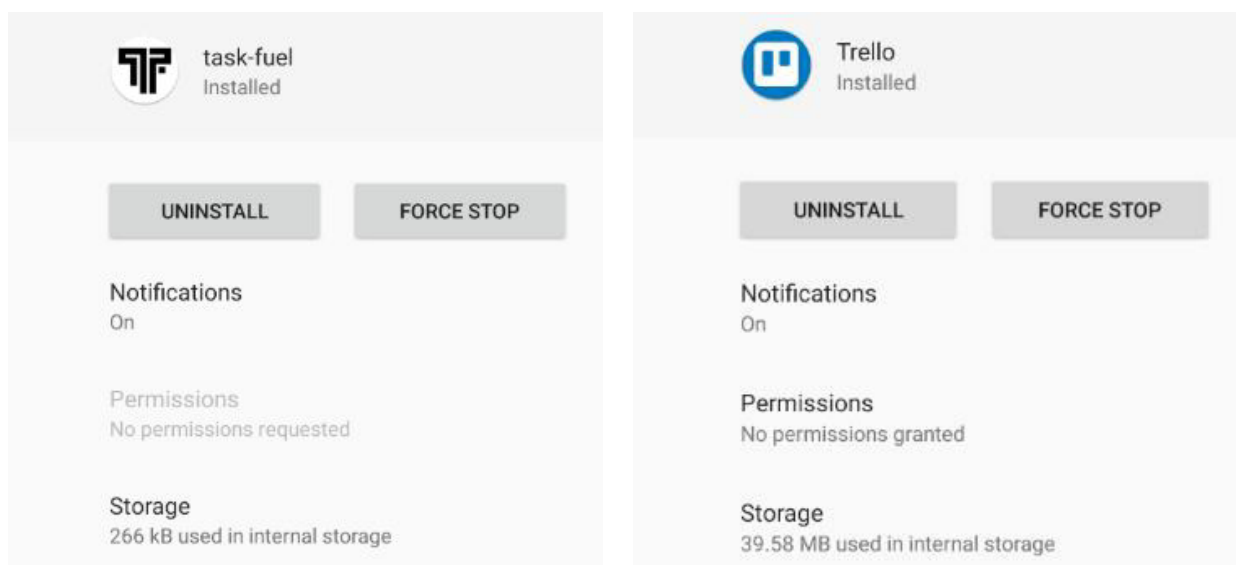


Figura 2- Comparație între cele două aplicații după spațiul de stocare ocupat

Întrunind toate aceste informații, putem deduce că o astfel de aplicație este o necesitate pentru echipele din întreaga lume, în plus, cum reiese și din capitolele anterioare, aplicația

propusă încercă să extragă toate avantajele de la competitorii săi și să aducă funcționalități noi, unde este cazul, pentru a oferi utilizatorilor o experiență cât mai plăcută și uniformă, fiind gândită a fi, oricând, la îndemâna oricui, indiferent de locul din care este utilizată, de conexiunea la internet sau de dispozitivul de pe care alege un utilizator să se conecteze.

2.2 Cerințele aplicației

2.2.1 Modul Online

În modul online, aplicația promite utilizatorilor săi o experiență de administrare a echipelor și asignare a sarcinilor între membrii săi cât mai interactivă. Aplicația oferă funcționalitate pentru următoarele:

- Crearea de noi table și crearea de echipe, întrucât tablele pot fi folosite atât pentru a administra proiecte personale, cât și proiecte în echipă.
- Crearea de noi sarcini și liste de sarcini. Fiecare sarcină trebuie să aparțină unei liste de sarcini, lista exprimând modulul din care face parte sarcina.
- Atribuirea sarcinilor către membrii echipei respective. Totuși aplicația permite și existența sarcinilor fără un utilizator responsabil pentru finalizarea acestora.
- Schimbarea stării sarcinilor și trimiterea emailurilor spre colaboratori în cazul finalizării acestora.
- Schimbarea ordinii sarcinilor în interiorul listei, pentru a face tabla mai ușor de citit și înțeles de către utilizatori.
- Vizualizarea ultimelor creări, ștergeri sau modificări aduse sarcinilor împreună cu data la care au fost acestea făcute.
- Vizualizarea informațiilor specifice unei anumite sarcini precum numele, descrierea, utilizatorul responsabil pentru finalizarea sa, intervalul orar, starea în care se află.
- Trimiterea informațiilor specifice sarcinilor, printr-un *URL* anonim, pentru a fi vizualizate și de utilizatorii care nu au acces la tabla respectivă.

2.2.2 Modul Offline

În modul offline aplicația încearcă să ofere o experiență cât mai apropiată de cea online, comportându-se asemănător unei aplicații native prin următoarele:

- Aplicația depozitează într-un mod progresiv în *cache* datele de pe paginile ce sunt accesate de utilizator. Astfel, când utilizatorul pierde conexiunea la internet, acesta poate accesa toate paginile pe care a navigat în trecutul relativ apropiat. Bineînțeles, paginile nu prezintă ultima versiune existentă a datelor, ci mai degrabă ultima variantă salvată. În plus datele nu vor putea fi actualizate până la restaurarea conexiunii.
- În momentul în care utilizatorul încearcă navigarea, în modul offline, spre o pagină care nu este depozitată în memoria *cache*, acesta va fi redirectat spre o pagină de *fallback* (Figura 3), această pagină este foarte importantă pentru a oferi utilizatorului senzația că nu a părăsit în vreun fel aplicația.
- Aplicația permite utilizatorilor săi să facă orice operație de modificare, cerere sau ștergere și în modul offline, stocând aceste cereri pentru a le trimite la revenirea conexiunii.

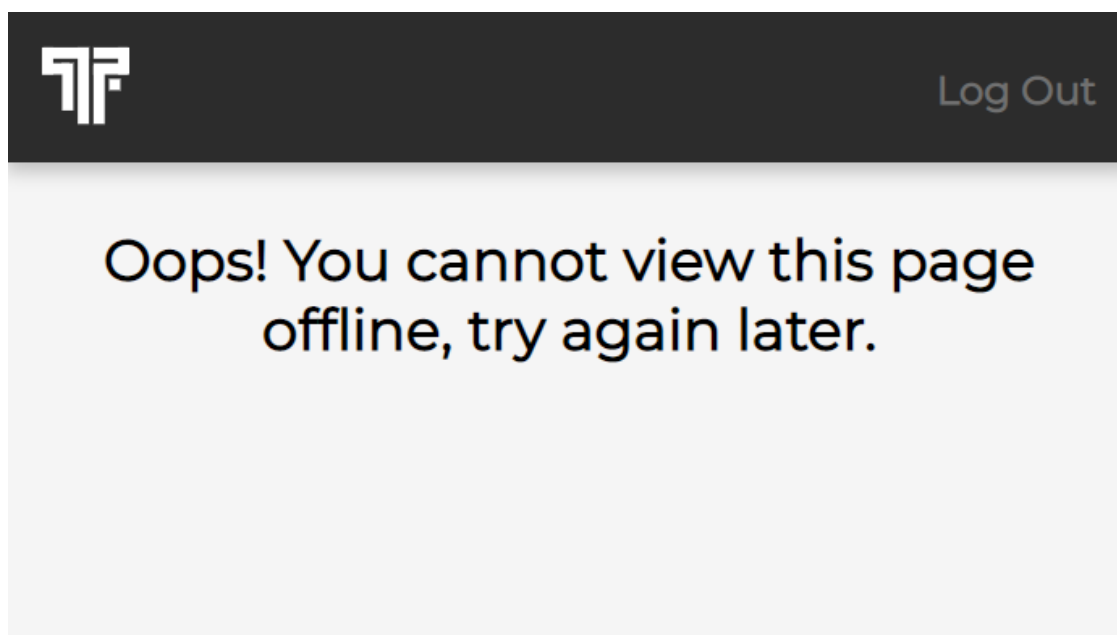


Figura 3 - Pagina pe care utilizatorul este redirectat la încercarea navigării spre o pagină ce nu este încă depozitată în *cache*

3 Analiză și proiectare

În acest capitol vom vorbi despre alegerea bazei de date folosite de către aplicație, despre diferite concepte folosite pentru a oferi utilizatorilor o experiență cât mai plăcută și vom vizualiza cum funcționează *API-ul GraphQL* al aplicației.

3.1 Structuri de date

Baza de date folosită de aplicație este una *NoSQL*, mai exact este vorba despre un *cluster MongoDB Atlas*. Am făcut această alegere datorită vitezei oferite de *MongoDB* și a compatibilității dintre aceasta și *Node.js*. Cele două tehnologii fiind întâlnite foarte des împreună. Unul dintre motivele compatibilității sporite dintre cele două tehnologii este faptul că *MongoDB* lucrează cu obiecte de tip *JSON*, iar conversia de la *JS* la *JSON* și înapoi este foarte simplă.

Un alt argument folosit de obicei, este flexibilitatea pe care o oferă o bază de date *NoSQL*, deoarece aceasta nu urmărește o anumită structură. Fiindcă aplicația folosește un *ODM*, toate intrările din baza de date vor urmări o schemă predefinită. Prin urmare, flexibilitatea va fi aceeași cu cea oferită de o bază de date *SQL*.

Unul dintre defectele acestui tip de baze de date este faptul că nu suportă tranzații. Tranzațiile reprezintă secvențe de operații ce se află în dependență directă una de cealaltă pentru a se finaliza. Acest aspect nu era neapărat relevant în dezvoltarea acestei aplicații, așa că în această privință o bază de date *NoSQL* ar fi la fel de potrivită precum una *SQL*.

Structura bazei de date este reprezentată în *Figura 4* și prezintă atât câmpurile pe care le poate prezenta un document din colecție, cât și relațiile dintre colecții. Cheile primare sunt precedate de simbolul unei chei, iar câmpurile ce nu pot fi nule sunt evidențiate prin eticheta *NN* așezată la sfârșit.

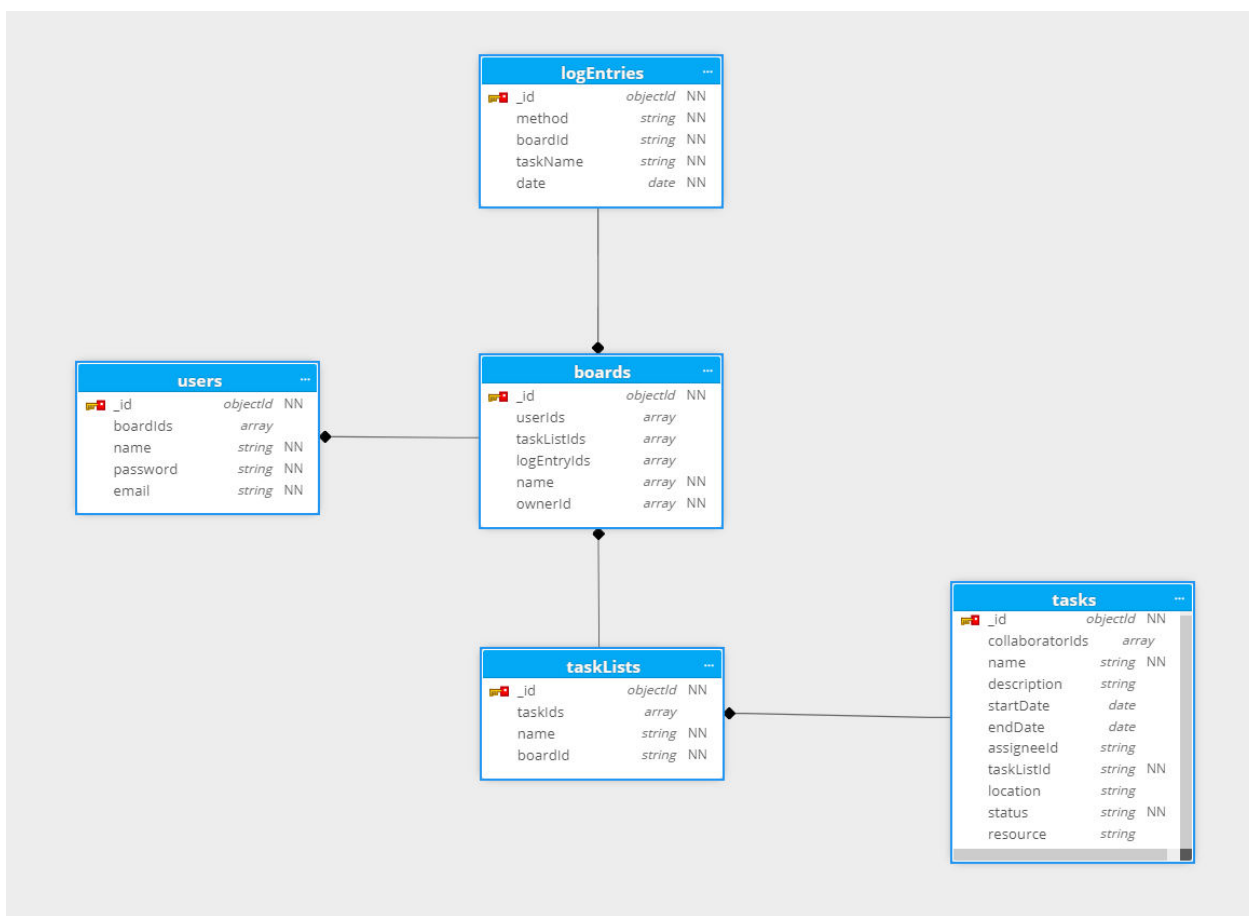


Figura 4 - Structura bazei de date

3.2 Interacțiunea cu utilizatorul

Interfața grafică a aplicației este proiectată în conformitate cu ultimele tendințe. Folosind un principiu numit *neumorphism design* reușim astfel, utilizând umbre de intensități diferite, să oferim iluzia unei interfețe tridimensionale.

Publicul țintă al aplicației este reprezentat de dezvoltatorii de aplicații. Cum aceștia sunt direct dependenți de o astfel de aplicație pentru finalizarea proiectelor și timpul petrecut vizualizând aplicația este destul de mare, interfața aplicației dorește să nu fie prea excentrică și obositoare. În același timp, interfața este proiectată pentru a fi ușor de citit și de înțeles de către noii utilizatori. Aceste trăsături sunt realizate întrunind următoarele concepte folosite de aplicație:

- fiecare buton prezintă efecte vizuale în momentul planării cursorului deasupra sa.
- Aplicația folosește un număr relativ scăzut de culori, majoritatea fiind nuanțe de alb și negru. Pentru a evidenția punctele importante din pagină, se folosesc culori

vii, deschise, precum culoarea albastru. Un exemplu de folosire a culorilor în acest mod este *Figura 19* de la pagina 25.

- Operațiile pe care le poate executa utilizatorul nu implică principii complicate. De exemplu pentru mutarea cardurilor în lista executată prin mișcarea de tragere și plasare.

Totodată aplicația este gândită pentru a fi cât se poate de flexibilă, pentru a putea fi folosită de pe orice dispozitiv și din orice loc. Din acest motiv interfața sa grafică este cât se poate de receptivă.

3.3 Descriere API

După cum am discutat și în capitolele anterioare, aplicația noastră folosește un *API GraphQL*. Acesta prezintă o singură rută la care sunt trimise toate cererile, `/graphql`. Din acest motiv, *API*-urile de acest tip nu pot fi documentate cu ajutorul unei unelte externe, cum este situația *API*-urilor *REST*. Cu toate acestea este considerat că aceste *API*-uri au posibilitatea de a se documenta singure cu ajutorul mediilor de dezvoltare integrate cum ar fi *GraphiQL* sau *GraphQLPlayground*. *API*-ul propus folosește primul mediu de dezvoltare, iar pentru a-l utiliza trebuie să accesăm din *browser* ruta `/graphql`.

Documentația generată poate fi vizualizată în *Figura 5*. În prima jumătate a figurii fiind reprezentate interogările, iar în a doua jumătate mutațiile. În reprezentarea figurilor sunt precizate și numele variabilelor necesare, împreună cu tipul lor de date și tipul de date al rezultatului. De exemplu, pentru a executa interogarea *boardByUserId* avem nevoie de o variabilă *id*, al cărei tip de date va fi *ID*. Rezultatul returnat va fi un obiect de tipul *Board*.

Mutațiile sunt operații care realizează modificări în baza de date, valoarea returnată de aceste operații fiind egală cu obiectul modificat. Excepție de la aceasta regulă fac mutațiile ce returnează valori booleene și mutațiile responsabile cu autentificarea, cele din urmă returnează un *authentication token*.

<div> <div><</div> <div>Schema</div> </div> <div>RootQueryType</div> <div>×</div>	<div> <div><</div> <div>Schema</div> </div> <div>RootMutation</div>
<div> <div>Q</div> <div>Search RootQueryType...</div> </div>	
No Description	
FIELDS	
task(id: ID): Task	addBoard(name: String! ownerId: ID! userIds: [ID] taskListIds: [ID] logEntryIds: [ID]): Board
user(id: ID): User	addTaskList(name: String! boardId: ID! taskIds: [ID]): TaskList
board(id: ID): Board	addUserToBoard(boardId: ID, userName: String): User
taskList(id: ID): TaskList	createBoard(name: String!, ownerId: ID!): Board
logEntry(id: ID): LogEntry	deleteBoard(id: ID!): Boolean
tasks: [Task]	deleteTaskList(id: ID!): Boolean
users: [User]	deleteTask(id: ID!): Boolean
boards: [Board]	updateTask(id: ID! boardId: ID! name: String description: String startDate: DateTime endDate: DateTime assigneeId: ID collaboratorIds: [ID] location: String status: String resource: String): Task
taskLists: [TaskList]	updateTaskList(id: ID!, taskIds: [ID!]!): Boolean
logEntries: [LogEntry]	
logEntriesByBoardId(id: ID): [LogEntry]	
boardsByUserId(id: ID): [Board]	
userByUsername(name: String): User	

Figura 5 - Documentație API creată cu ajutorul *GraphQL*, atât interogări cât și mutații

4 Implementare

În acest capitol vom vorbi despre anumite metode folosite pentru implementarea aplicației, despre problemele întâmpinate în momentul dezvoltării acesteia și, bineînțeles, despre rezolvarea problemelor pe care le vom enumera ulterior.

4.1 GraphQL

Să vorbim mai întâi despre Serverul *GraphQL* și mai exact despre cum am adăugat funcționalitățile *GraphQL* peste un server *Express* normal. Pentru implementare am folosit pachetele *graphql* și *graphql-express*.

4.1.1 Schema

Primul pas este crearea unei scheme *GraphQL*. Utilizăm pachetul *graphql* pentru a importa clasa *GraphQLSchema*, pe care urmează să o folosim pentru a instanția un obiect de acest tip. În figura următoare putem observa schema folosită în aplicația noastră.

```
module.exports = new GraphQLSchema({  
  query: RootQuery,  
  mutation: RootMutation  
});
```

Figura 6 - Fragment de cod, exemplu de schemă GraphQL

Această schema definește operațiile pe care le poate realiza serverul nostru. În exemplul anterior am folosit doar două tipuri de operații, interogări și mutații. Cu toate acestea unele scheme *GraphQL* folosesc și subscripții despre care am vorbit mai în amănunt în capitolul **Tehnologii folosite**.

Variabilele *RootQuery* și *RootMutation* sunt obiecte ale clasei *GraphQLObjectType*, care este la rândul său importată din pachetul *graphql*, aceste obiecte conțin toate interogările, respectiv mutațiile, pe care urmează să le implementăm. Exemplul următor este luat tot din aplicație, dar este simplificat, oferind funcționalitate doar pentru manipularea sarcinilor.

```

const RootMutation = new GraphQLObjectType({
  name: 'RootMutation',
  fields: {
    addTask: addTaskMutation,
    deleteTask: deleteTaskMutation,
    updateTask: updateTaskMutation,
  }
})

```

Figura 7 - Fragment de cod, exemplu de obiect ce conține mutații

```

const addUserToBoardMutation = {
  type: UserType,
  args: {
    boardId: {
      type: new GraphQLNonNull(GraphQLID)
    },
    userName: {
      type: new GraphQLNonNull(GraphQLString)
    }
  },
  resolve(parent, args, req) {
    //funcționalitatea dorită
  }
}

```

Figura 8 – Fragment de cod, exemplu de mutație

Pentru interogări și subscripții procesul este similar celui exemplificat mai sus.

Toate cele trei tipuri de operații prezintă următoarele câmpuri:

- Câmpul *type*, reprezintă tipul obiectului returnat de funcția *resolve*.
- Câmpul *args*, reprezintă totalitatea argumentelor pe care le va primi metoda aleasă, pe lângă numele argumentelor trebuie precizat în mod obligatoriu și tipul acestora. Pentru a ne asigura că acestea nu pot fi nule, trebuie să instanțiem clasa *GraphQLNonNull*.
- Funcția *resolve*. În această funcție adăugăm funcționalitatea pe care dorim să o aibă operația noastră.

4.1.2 Trimiterea cererilor

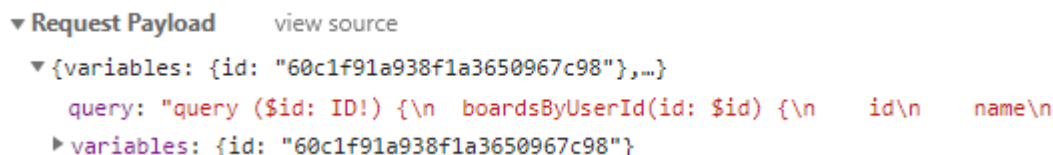
Pentru a facilita trimiterea cererilor și a ne asigura că acestea sunt înțelese de către *server*, folosim funcția *withApollo* din cadrul pachetului *react-apollo*, peste componenta care dorim să realizeze comunicarea.

După executarea acestui pas, obiectului responsabil cu comunicarea dintre componente, *props*, i se va adăuga un nou câmp, numit *client*, ce conține metodele *query* și *mutation* pe care le putem utiliza.

Aceste metode primesc drept parametru un obiect ce conține cel puțin interogarea, respectiv mutația. Putem găsi un exemplu de interogare la pagina 7 (*Figura 1*).

```
this.props.client.query({
  query: getBoardsByUserIdQuery,
  variables: {
    id: this.context.userId
  },
  fetchPolicy: 'no-cache'
}).then((result) => {
  //manipularea rezultatului
})
})
```

Figura 9 - Trimiterea unei cereri ce conține o interogare



```
▼ Request Payload view source
{variables: {id: "60c1f91a938f1a3650967c98"},...}
  query: "query ($id: ID!) {\n  boardsByUserId(id: $id) {\n    id\n    name\n  }\n}"
  variables: {id: "60c1f91a938f1a3650967c98"}
```

Figura 10 - Exemplu de *payload* al unei cereri trimise

În exemplul din *Figura 9* putem observa că pe lângă interogarea propriu-zisă, obiectul primește și variabilele pentru care urmează să se realizeze interogarea. Cererea trimisă spre server va avea o încărcătură asemănătoare cu cea prezentată în *Figura 10*

4.2 Cache dinamic

În subcapitolul **Biblioteca React** de la 4, am precizat că *React* prezintă o opțiune de creare a proiectelor plecând de la un șablon. Tot în acel capitol am precizat și că acest proiect a fost creat plecând de la șablonul *cra-template-pwa*, astfel proiectului nostru i se conferă abilitatea de a introduce în *cache* conținutul static de pe pagini. Conținutul static fiind conținutul ce nu așteaptă date de la server pentru a fi redat. Aceasta, de la sine, este o experiență a utilizatorilor mult mai calitativă decât mesajul de eroare ce apare, în mod implicit, în momentul pierderii conexiunii la internet.

Totuși cum am tratat conținutul dinamic? Pentru a trata conținutul dinamic de pe pagini, ca și în cazul celui static, trebuie să folosim tot *service worker*-i. Aceștia sunt fișiere *JavaScript* al căror fir de execuție funcționează concomitent cu firul de execuție al aplicației. Nu au acces direct la elementele din *DOM*, atribuția lor fiind să asculte pentru solicitări de intrare sau de ieșire.

În fragmentul de cod de mai jos, putem observa un exemplu de funcție prin care *service worker*-ul ascultă pentru solicitările ce folosesc verbul *HTTP POST*, trimise spre serverul nostru *GraphQL*.

```
self.addEventListener('fetch', async (event) => {
  let graphqlRegularExpression = new RegExp('graphql')
  if (event.request.method === 'POST' &&
      graphqlRegularExpression.test(event.request.url)) {
    await event.respondWith(getResponseByQueryType(event))
  }
})
```

Figura 11 – Fragment de cod, exemplu de *event listener* ce ascultă pentru cereri

După preluarea cererilor, acestea vor fi procesate de funcția *getResponseByQueryType*, care în final va returna o promisiune ce are posibilitatea de a fi rezolvată astfel încât funcția *respondWith* să reușească trimiterea unui răspuns la cererea făcută.

4.2.1 Tratarea interogărilor

Pentru tratarea interogărilor folosim o abordare de tip *Network First*, abordare ce se rezuma la faptul că răspunsul cererilor de interogare este în mod implicit cel primit de la server, fiind folosit *cache*-ul doar atunci când de la server nu ajunge un răspuns. Bineînțeles, pentru a

păstra doar date actualizate, introducem răspunsul în *cache* după fiecare cerere reușită sau, cu alte cuvinte, după fiecare răspuns venit de la server.

```
let body = await event.request.clone().json()
if(body.query.startsWith("query")){

    let cachedResponse = await getCache(event.request.clone())

    let fetchPromise = fetch(event.request.clone())
    .then((response) => {
        setCache(event.request.clone(), response.clone())
        return response
    })
    .catch((err) => {
        if (cachedResponse) { return cachedResponse } else {throw err }
    })

    return fetchPromise
}
```

Figura 12 - Fragment de cod responsabil pentru tratarea cererilor ce reprezintă interogări

În secvența de cod expusă în *Figura 12* putem observa comportamentul descris anterior, pentru fiecare cerere verificăm tipul acesteia, dacă acesta este interogare, atunci continuăm trimiterea cererii și așteptăm răspunsul. În cazul în care răspunsul ajunge, îl introducăm în *cache* și îl returnăm, iar în caz contrar verificăm dacă există deja un răspuns pentru cererea respectivă în *cache*.

4.2.2 Tratarea mutațiilor

Pentru tratarea mutațiilor am folosit o abordare numită *Request Deferrer*, ce constă în depozitarea cererilor în *cache*, în timp ce utilizatorul nu are conexiune la internet, și re-trimiterea acestora în momentul în care îi revine conexiunea.

```

if(body.query.startsWith("mutation")){
  if (!navigator.onLine) {

    return serializeForMutation(event.request).then(async (serialized) => {

      let queue = (await idbGet("queue", mutationStore)) || []
      queue.push(serialized)
      idbSet('queue', queue, mutationStore)
      return returnDummyResponse()

    })
  }
}

```

Figura 13 – Fragment de cod responsabil pentru tratarea cererilor ce reprezintă mutații

Cu ajutorul obiectului *navigator* putem verifica dacă utilizatorul este sau nu conectat la internet. Iar în cazul cel din urmă, obiectul de tip *Request* trebuie adăugat în lista aflată în tabela *mutationStore*.

```

async function flushQueue() {
  return await idbGet("queue", mutationStore).then((queue) => {
    queue = queue || []
    if (!queue.length) {
      return Promise.resolve()
    }
    return sendInOrder(queue).then(function() {
      return idbSet('queue', [], mutationStore)
    })
  })
}

```

Figura 14 – Preluarea cozii din *IndexedDB* și eliminarea conținutului acesteia

```

function sendInOrder(requests) {
  var sending = requests.reduce((prevPromise, serialized) => {
    return prevPromise.then(() => {
      return deserializeForMutation(serialized).then((request) =>{
        return fetch(request)
      })
    })
  }, Promise.resolve())
  return sending
}

```

Figura 15 - Trimiterea cererilor în ordinea în care au fost făcute

În acest exemplu (*Figura 15*) putem observa modalitatea prin care conținutul cozii este eliminat și trimiterea cererilor în ordinea în care au ajuns. După ce reconstruim obiectele ce reprezintă cererile, utilizând funcția *fetch*, trimitem cererile la adresa precizată în câmpul *url* al acestora.

4.2.3 Probleme apărute la implementare și rezolvări ale acestora

Cache API este sistemul ales de majoritatea dezvoltatorilor de aplicații web pentru depozitarea cererilor și răspunsurilor corespunzătoare acestora. Totuși una dintre problemele acestuia este faptul că nu permite depozitarea solicitărilor de tip *POST*.

După cum am menționat și în capitolul **Tehnologii folosite**, în secțiunea **GraphQL**, aplicația noastră comunică în mare parte doar prin solicitări de acest tip.

Pentru a rezolva această problemă, vom alege ca stocarea să fie făcută cu ajutorul *IndexedDB*. Acest procedeu îngreunează puțin procesul deoarece este necesar ca obiectul să fie descompus în momentul stocării și recompus în momentul preluării sale din baza de date.

```
async function serializeForQuery(response) {  
  
  let serializedHeaders = {}  
  for (var entry of response.headers.entries()) {  
    serializedHeaders[entry[0]] = entry[1]  
  }  
  
  let serialized = {  
    headers: serializedHeaders,  
    status: response.status,  
    statusText: response.statusText  
  };  
  
  serialized.body = await response.json()  
  return serialized  
}
```

Figura 16 – Descompunerea răspunsului interogării

În funcția prezentată anterior (*Figura 16*) primim drept parametru un obiect de tipul *Response*, pe care îl descompunem pentru a-l putea introduce în baza de date. Pentru a crea obiectul descompus trebuie să adăugăm unui nou obiect câmpurile celui inițial. Știind că obiectul

pe care vrem să îl descompunem este de tipul *Response*, majoritatea câmpurilor pot fi adăugate în mod direct, totuși două dintre câmpuri fac excepție de la această regulă:

- Câmpul *headers*. Acest proces presupune iterarea prin câmpul *headers* al obiectului primit drept parametru și asignarea perechilor de date de tipul cheie-valoare la noul obiect.

```
▼ Response Headers    View source
Access-Control-Allow-Origin: *
Connection: keep-alive
Content-Length: 3466
Content-Type: application/json; charset=utf-8
Date: Sun, 20 Jun 2021 10:40:55 GMT
ETag: W/"d8a-ZJenz3G1kg1quneau9HrNKftRDI"
Keep-Alive: timeout=5
X-Powered-By: Express
```

Figura 17 - Exemplu de valoare pe care o poate lua câmpul *headers*

- Câmpul *body*, a cărui valoare poate fi obținută apelând metoda asincronă *json()* peste obiectul de tip *Response*.

```
▼ data: {task: {id: "60ce11e058ff3a1ec45b6ac6", name: "Adăugare tablă", description: "",...}}
▼ task: {id: "60ce11e058ff3a1ec45b6ac6", name: "Adăugare tablă", description: "",...}
  ▼ assignee: {id: "60c1f91a938f1a3650967c98", name: "theodor.spantu", __typename: "User"}
    id: "60c1f91a938f1a3650967c98"
    name: "theodor.spantu"
    __typename: "User"
  ► collaborators: {id: null, name: null, __typename: "User"}
    description: ""
    endDate: "2021-06-23T15:00:00.000Z"
    id: "60ce11e058ff3a1ec45b6ac6"
    location: "Romania"
    name: "Adăugare tablă"
    resource: null
    startDate: "2021-06-23T11:00:00.000Z"
    status: "in progress"
    __typename: "Task"
```

Figura 18 - Exemplu de valoare pe care o poate lua câmpul *body*

5 Manual de utilizare

În acest capitol vom discuta despre modul de utilizare și instalare al aplicației și în același timp despre funcționalitatea oferită de aceasta într-un mod mai detaliat. În plus, vom discuta puțin și despre structura aplicației care este, cum am spus și în capitolul **Prezentarea aplicației**, în secțiunea **Studiu de caz**, puțin diferită față de cea a competitorilor săi.

Prima pagină pe care o vedem în momentul conectării la site-ul web este, bineînțeles, pagina de înregistrare (*Figura 19*) ce ne permite să ne creăm un cont pentru a continua navigarea în interiorul aplicației. După ce ne creăm propriul cont și ne autentificăm, o să avem acces la pagina principală, unde putem vizualiza tablele ce ne sunt disponibile la momentul actual.

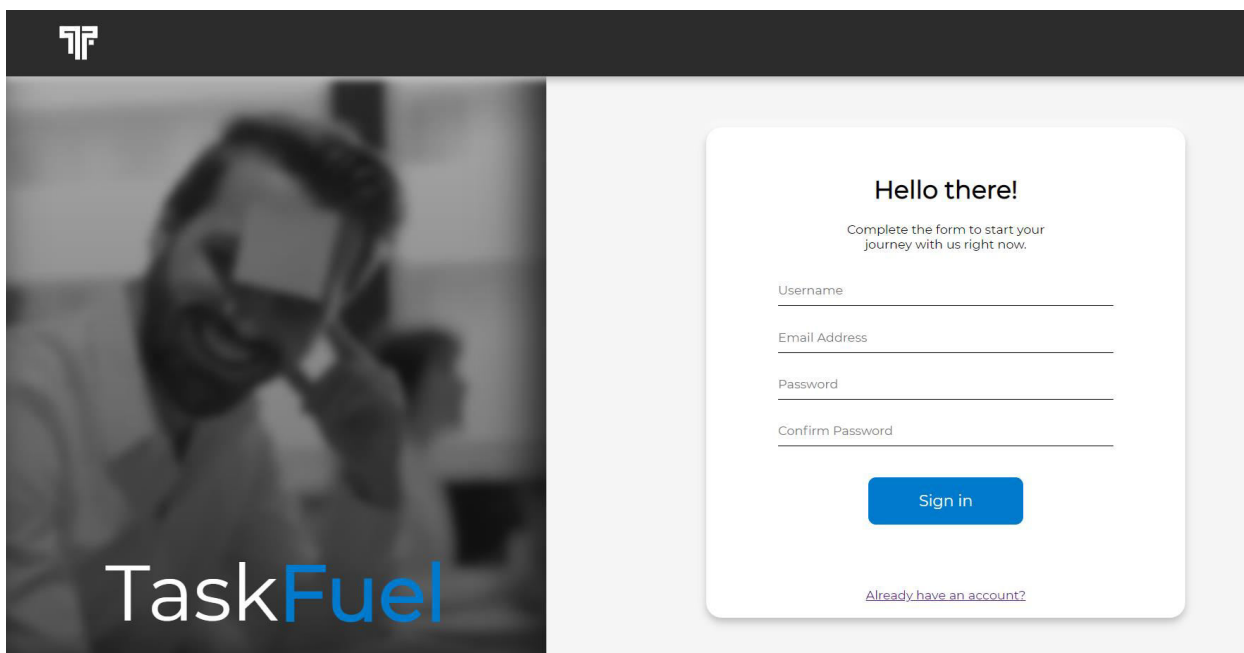


Figura 19 - Pagina de înregistrare

Pentru a crea o noua tablă, putem apăsa pe butonul ce se află în meniul din partea stângă, *Add board*. Singurul câmp pe care va trebui să îl completăm este numele tablei. De îndată ce am creat tablă, o putem vizualiza pe pagina principală sub forma unui card.

În *Figura 20* putem observa un exemplu de pagină principală deja populată cu carduri.

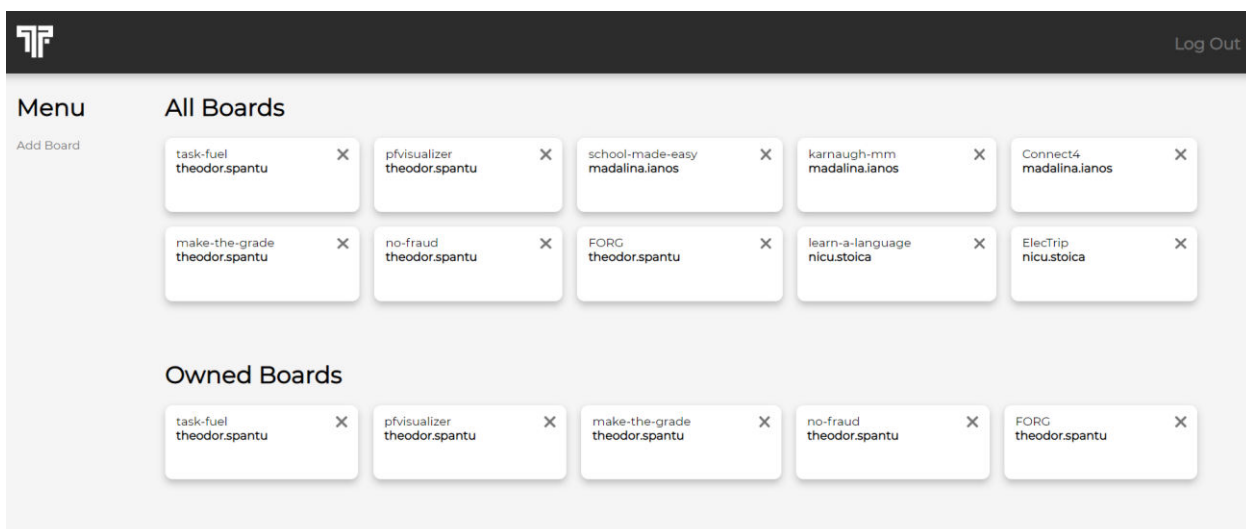


Figura 20 – Exemplu de pagină principală populată

După crearea cu succes a unui card, avem posibilitatea de a apăsa pe el pentru a naviga spre pagina responsabilă tablei respective.

Tabla urmează modelul unei table kanban, despre care am vorbit în capitolul **Prezentarea aplicației**, cu mici schimbări aduse structurii acesteia. Să vizualizăm mai bine ce este schimbat? Ei bine, listele în acest model tablei kanban ar trebui să fie de fapt stările prin care trece o sarcină până la finalizarea acesteia, un exemplu scurt de liste într-o tabla kanban autentică:

- De făcut
- În desfășurare
- În revizuire
- Terminat

Aplicația noastră oferă funcționalitatea de a asigura fiecărei sarcini o stare, așa că listele noastre ar trebui să fie mai degrabă module la care se lucrează concomitent, un exemplu destul de general pentru o aplicație web:

- Pagina de înregistrare
- Pagina utilizatorului
- Securitate
- Erori de corectat

Acest tip de tablă adaugă o dimensiune nouă unei table autentice kanban, fapt ce o structurează mult mai bine și o face mai ușor de citit, în schimbul unui minim de efort depus în plus, deoarece listele nu pot fi permanente, ci trebuie schimbate periodic.

Dacă tabla este abia creată, pagina ar trebui să prezinte un singur utilizator, utilizatorul autentificat. În plus aceasta nu ar trebui să prezinte sarcini, liste de sarcini sau înregistrări în istoric. În acest moment ar trebui să ne creăm și o echipă de colaboratori cărora să asignăm diferitele acțiuni. Acest lucru poate fi făcut din meniul *Collaborators*, apăsând pe butonul *Add collaborators* și completând formularul cu numele viitorului colaborator.

Pentru a popula tabla cu sarcini vom crea pentru început, în mod obligatoriu, o listă, după modelul expus puțin mai sus. Am precizat faptul că este o necesitate să creăm mai întâi lista, deoarece aplicația nu permite utilizatorilor să creeze sarcini care nu aparțin de o anumită listă.

După realizarea cu succes a operației descrise anterior, putem să creăm o sarcină în interiorul listei mai sus numite, această operațiune necesită puțin mai multe date față de cele de până acum. Pentru început trebuie să îi dăm un nume sugestiv și adecvat sarcinii, astfel încât colaboratorii să-și poată da seama de natura sa doar citind acest câmp, în caz că numele pe care dorim să i-l asignăm sarcinii este prea lung, avem posibilitatea de a-l descompune, păstrând doar ideea principală în titlu și adăugând ideile secundare la descrierea sarcinii. Tot cu ajutorul acestui formular putem asigna sarcinii și un reprezentant care urmează să o îndeplinească și intervalul orar în care trebuie acesta îndeplinit.

Sarcinile au 4 stări, cu nume sugestive, în care se pot afla, echivalente cu numele listelor dintr-o tablă kanban: neînceput, anulat, în desfășurare, terminat. Asignarea unei noi stări schimbă totodată și interfața cardului, adăugând astfel o iconiță sugestivă pentru a facilita înțelegerea tablei.

Cum aplicația prezintă și funcționalități *drag and drop* pentru cardurile acțiunilor, o tablă cât mai inteligibilă ar trebui să aibă în partea superioară sarcinile neîncepute și cele care sunt în desfășurare, deoarece este o probabilitate mai mare să avem nevoie să vizualizăm date despre acestea decât despre cele deja terminate sau cele anulate. O altă recomandare este de a avea sarcinile mereu sortate după stare, pentru a nu deveni paleta de culori deranjantă din cauza multitudinii de culori amestecate. Un bun exemplu este cel din *Figura 21*.

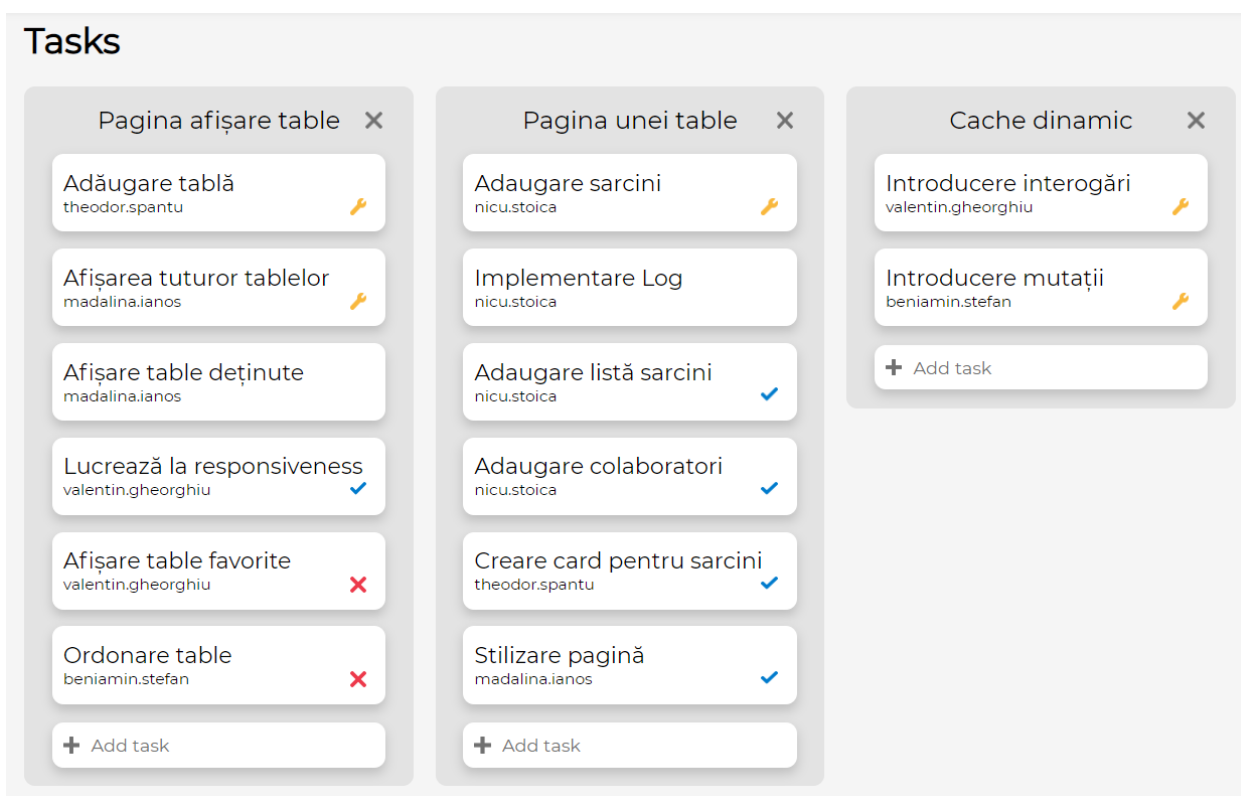


Figura 21 – Exemplu de aranjare a sarcinilor în interiorul tablei

În momentul în care o sarcină este marcată drept terminată, atât reprezentantul cât și colaboratorii primesc un email pe adresele aferente conturilor lor.

În momentul creării, ștergerii sau schimbării stării unei sarcini, această operațiune va fi adăugată în istoric și va putea fi vizualizată de toți colaboratorii în secțiunea de înregistrări din partea stângă. Mai exact datele ce vor apărea în acea secțiune sunt numele sarcinii, noua stare în care a intrat aceasta și data realizării schimbării.

De observat faptul că pe cardul abia creat nu sunt afișate toate informațiile, ci doar numele, reprezentantul și eventual starea în care se află o sarcina. Pentru a vedea toate informațiile unui task trebuie să apăsăm pe cardul acestuia, acțiune care ne va redirecționa către pagina specifică sarcinii.

Ajunși acolo avem posibilitatea de a vizualiza toate datele asociate sarcinii respective (vezi și *Figura 22*) și de a exporta aceste date printr-un *URL* anonim pentru a o împărtăși și utilizatorilor care nu au acces la tabla respectivă.



Figura 22 – Exemplu de pagină asociată unei anumite sarcini

Acum că am trecut prin toate funcționalitățile aplicației, să vorbim puțin și despre cum putem instala această aplicație pe telefonul mobil. Aplicația poate fi instalată în două moduri:

- Produsul fiind un *PWA*, oferă funcționalitatea de a reaminti utilizatorului că aplicația nu este încă instalată și îi permite instalarea prin apăsarea unui singur buton.
- În cazul în care nu apare notificarea de instalare în timp util, sau nu vrem să așteptăm această notificare, putem să instalăm manual aplicația, selectând din meniul *browser*-ului opțiunea *Install App*.

După ce executăm oricare dintre cei doi pași, aplicația va fi instalată și o scurtătură va fi creată pe *Home screen*. În *Figura 23* sunt exemplificați pașii de mai sus și scurtătura creată după executarea lor.

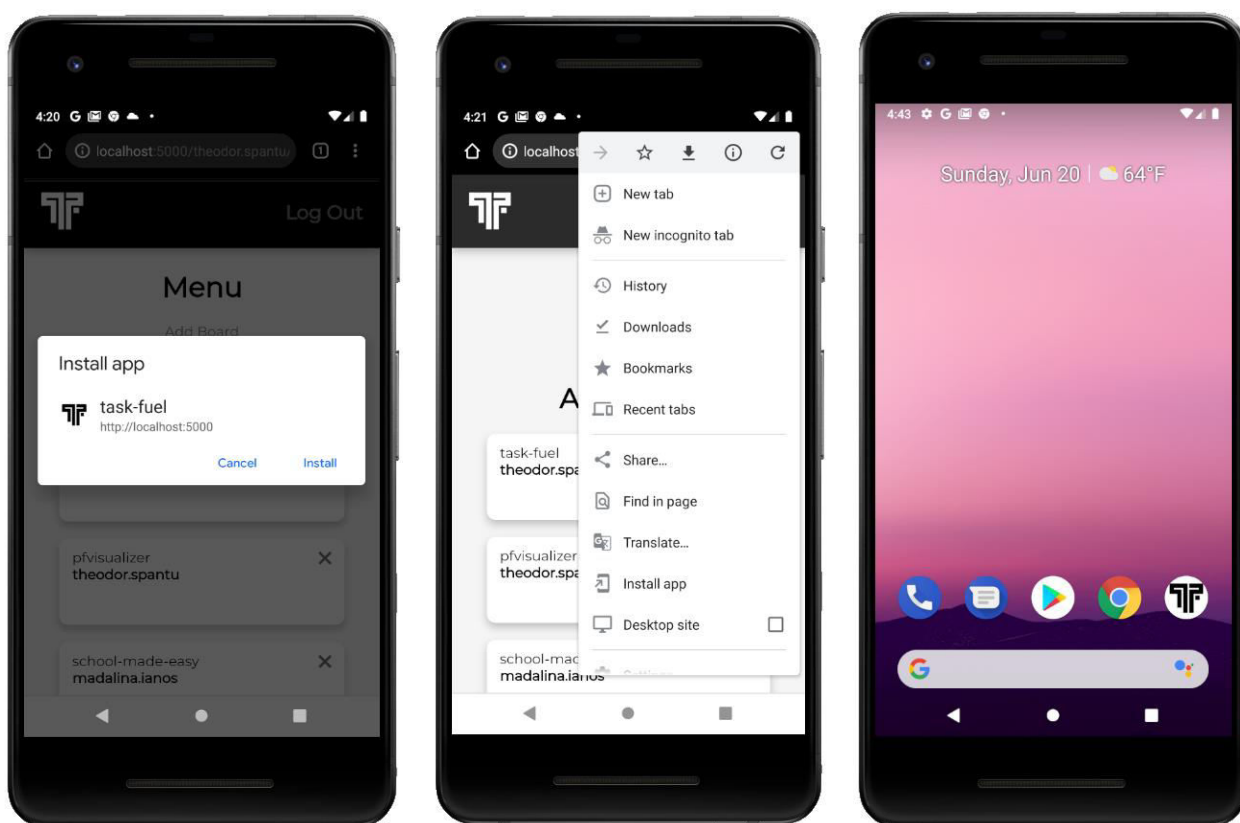


Figura 23 - Cele două metode de instalare și scurtătura aplicației pe *Home Screen*

În acest moment putem folosi produsul fără a ne conecta din *browser*. Aplicația este făcută să imite cât mai bine o aplicație nativă, din acest motiv nu va mai apărea partea superioară ce conține ruta actuală și setările.

6 Concluzii și idei de viitor

Lucrarea surprinde tendințele actuale ale aplicațiilor deja existente pe piață și propune dezvoltarea unei aplicații care încearcă pe cât posibil să însumeze toate calitățile competitorilor săi și să aducă în același timp funcționalitate nouă produselor deja existente, oferind astfel utilizatorilor o experiență offline cât mai apropiată de cea online.

Această aplicație ar putea fi îmbunătățită prin următoarele:

- Implementarea de statistici la nivel de tablă sau chiar la nivel de profil al utilizatorului pentru a putea vizualiza dacă abordarea aleasă pentru dezvoltarea unui anumit proiect este optimă.
- Implementarea de notificări, trimise în momentul finalizării cu succes a unei sarcini, concomitent cu trimiterea emailurilor.
- Implementarea posibilității de a viziona sarcinile într-o manieră orientată mai mult în jurul intervalelor de timp.
- Implementarea listelor de verificare la nivelul sarcinilor. Folosite pentru a împărti fiecare sarcină în mai multe partiții, rezolvarea sarcinii urmând să fie reprezentată de rezolvarea fiecărui element din lista.
- Implementarea sarcinilor repetate. Folosite de cei ce au de îndeplinit sarcini ce se repetă la anumite intervale de timp.

Bibliografie

- 1) Farid Said Tahirshah, *Comparison between Progressive Web App and Regular Web App*, 2019, <https://www.diva-portal.org/smash/get/diva2:1334458/FULLTEXT02.pdf>
- 2) Camille Oggier, *How fast GraphQL is compared to REST APIs*, 2020, https://www.theseus.fi/bitstream/handle/10024/340318/Thesis_Camille_Oggier.pdf?sequence=2
- 3) Kevin Farrugia, *A Beginner's Guide To Progressive Web Apps*, 2016, <https://www.smashingmagazine.com/2016/08/a-beginners-guide-to-progressive-web-apps/>
- 4) Matt Gaunt, *Service Workers : An Introduction*, <https://developers.google.com/web/fundamentals/primers/service-workers>, 2020
- 5) Blessing Krofegha, *Understanding Client-Side GraphQL with Apollo-Client In React Apps*, <https://www.apollographql.com/blog/graphql/examples/full-stack-react-graphql-tutorial/>, 2020
- 6) Pete LePage, *What does it take to be installable?*, <https://web.dev/install-criteria/>, 2021
- 7) * * *, GraphQL : <https://graphql.org/learn/>
- 8) * * *, express-graphql : <https://graphql.org/graphql-js/running-an-express-graphql-server/>
- 9) * * *, apollo-client : <https://www.apollographql.com/docs/react/api/react/hoc/>
- 10) * * *, React : <https://reactjs.org/tutorial/tutorial.html>
- 11) * * *, Mongoose ODM : <https://mongoosejs.com/docs/guide.html>
- 12) * * *, JSON Web Tokens : <https://hasura.io/blog/best-practices-of-using-jwt-with-graphql/>
- 13) * * *, Network first : <https://serviceworke.rs/strategy-network-or-cache.html>
- 14) * * *, Request Deferrer : <https://serviceworke.rs/request-deferrer.html>