

UNIVERSITATEA TEHNICĂ „Gheorghe Asachi” din IAȘI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DOMENIUL: Calculatoare și Tehnologia Informației
SPECIALIZAREA: Tehnologia Informației

Lucrare de diplomă

Ș.l.dr.ing.
Vieriu George-Emil

Absolvent
Ionel Tudor-Paul

Iași, 2024

Iași, 2024

UNIVERSITATEA TEHNICĂ „Gheorghe Asachi” din IAȘI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DOMENIUL: Calculatoare și Tehnologia Informației
SPECIALIZAREA: Tehnologia Informației

CodeOverflow - platformă web Q&A pentru developeri

LUCRARE DE DIPLOMĂ

Ș.l.dr.ing.

Vieriu George-Emil

Absolvent

Ionel Tudor-Paul

Iași, 2024

Iași, 2024

DECLARAȚIE DE ASUMARE A AUTENTICITĂȚII LUCRĂRII DE LICENȚĂ

Subsemnatul IONEL TUDOR-PAUL, legitimat cu C.I. seria M.Z. nr. 874177, CNP 5010620226700, autorul lucrării Platformă web destinată transporturilor de mărfuri, elaborată în vederea susținerii examenului de finalizare a studiilor de licență organizat de către Facultatea de Automatică și Calculatoare din cadrul Universității Tehnice „Gheorghe Asachi” din Iași, sesiunea Iulie a anului universitar 2023-2024, luând în considerare conținutul art. 34 din Codul de etică universitară al Universității Tehnice „Gheorghe Asachi” din Iași (Manualul Procedurilor, UTI.POM.02 – Funcționarea Comisiei de etică universitară), declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române (Legea nr. 8/1996) și a convențiilor internaționale privind drepturile de autor.

Data

Semnătura

01-07-2024



Iași, 2024

CUPRINS

Introducere	1
Capitolul 1. Fundamentarea teoretică și documentarea bibliografică	3
Capitolul 1.1. Fundamente Teoretice	3
Capitolul 1.2. Proiecte Similare	5
Capitolul 1.3. Specificații funcționale	6
Capitolul 2. Proiectarea aplicației	7
Capitolul 2.1. Analiza platformei hardware	7
Capitolul 2.2. Proiectarea aplicației - idei generale	8
Capitolul 3. Implementarea Aplicației	17
Capitolul 3.1. Interfața cu utilizatorul	17
Capitolul 3.2. Stocarea informațiilor	21
Capitolul 3.3. Dificultăți întâmpinate și modalități de rezolvare	22
Capitolul 4. Testarea aplicației	24
Capitolul 4.1. Punerea în funcțiune a aplicației	24
Capitolul 4.2. Testarea sistemului	24
Concluziile Lucrării	31
Bibliografie	32
Anexa	33

CodeOverflow - platformă web Q&A pentru developeri

IONEL TUDOR-PAUL

REZUMAT

Industria IT este în continuă expansiune, iar numărul celor care doresc să devină ingineri software crește constant. În acest context, necesitatea de formare și educare continuă este esențială, de la învățarea conceptelor de bază, precum variabilele și structurile de control, până la concepte avansate precum designul orientat pe obiecte și arhitecturile de microservicii. Învățarea în domeniul IT este un proces continuu și provocator.

O soluție rapidă ar fi o aplicație web unde utilizatorii pot adresa și răspunde la întrebări, facilitând un mediu de învățare colaborativ. Utilizatorii ar putea crea profile, urmări subiecte de interes și primi notificări pentru răspunsuri noi sau întrebări relevante.

Dezvoltarea acestei aplicații ar trebui să folosească strategia de randare a conținutului pe partea serverului (SSR), care generează marcajul HTML static pe server, oferind browserului o pagină HTML complet redată. Acest proces implică preluarea datelor necesare de către server, compilarea componentelor JavaScript în HTML static și trimiterea documentului HTML către client. Ulterior, clientul descarcă fișierele JavaScript și atașează ascultători de evenimente componentelor printr-un proces numit hidratare. Această metodă optimizează performanța și experiența utilizatorului.

Introducere

Industria IT este în continuă creștere, astfel tot mai mulți doresc să devină ingineri software. Fiecare dintre aceștia are neclarități despre concepte de programare, iar nevoia de formare și educare continuă este mai mare ca niciodată. De la noțiuni de bază, cum ar fi variabilele și structurile de control, până la concepte avansate precum designul orientat pe obiecte și arhitecturile de microservicii, învățarea este un proces constant și provocator.

Pe măsură ce programatorii avansează în carieră, ei descoperă importanța abilităților non-tehnice, cum ar fi comunicarea eficientă, munca în echipă și gestionarea proiectelor. Aceste abilități contribuie la succesul lor în cadrul echipelor și proiectelor complexe.

Participarea activă în discuții tehnice și non-tehnice nu doar că sporește cunoștințele, dar și încurajează schimbul de idei și perspective diferite, facilitând astfel inovația și soluționarea creativă a problemelor. De asemenea, răspunsul la întrebări adresate de alții este o modalitate eficientă de a consolida propriile cunoștințe și de a dezvolta abilități de mentorat și leadership.

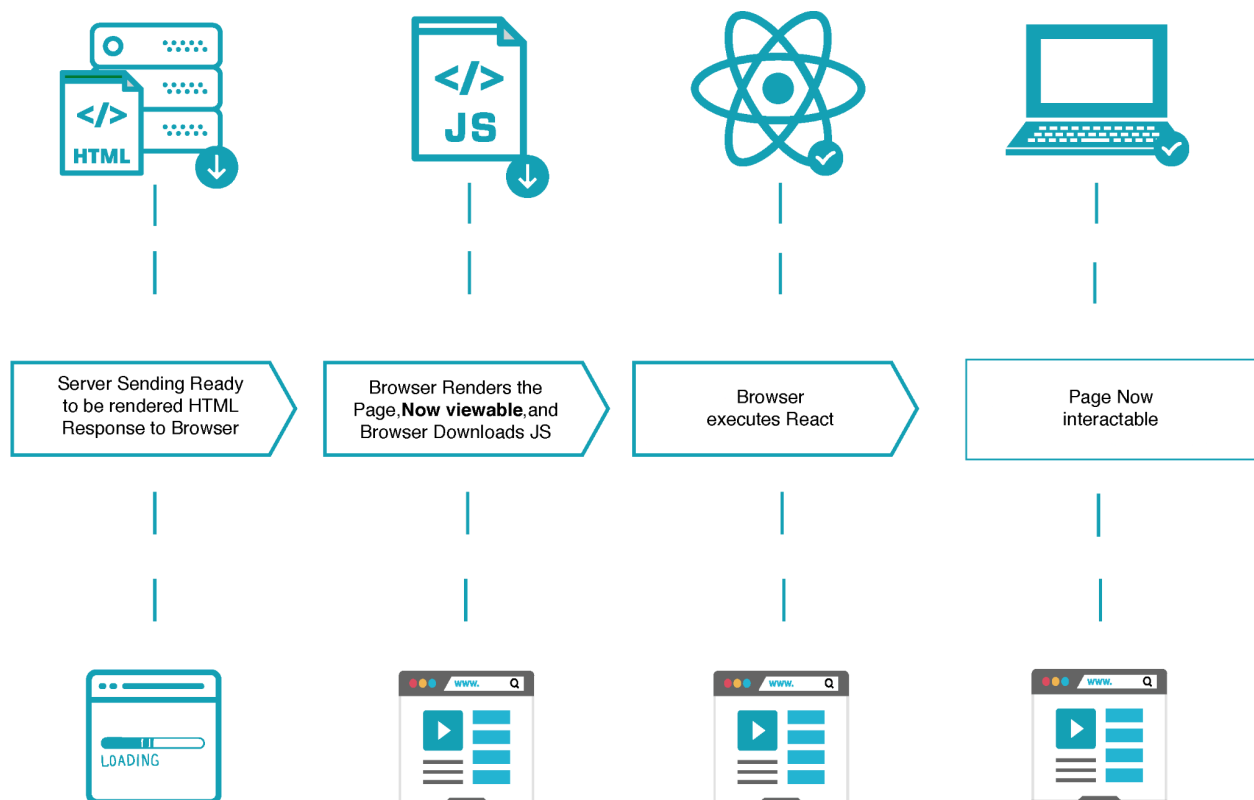
Pentru a rezolva problemele și pentru a-și dezvolta abilitățile tehnice și non-tehnice, aceștia ar trebui să pună și să răspundă la întrebări. Acest proces de interogare constantă ajută la clarificarea conceptelor, la aprofundarea cunoștințelor și la identificarea lacunelor în înțelegere. Întrebările pot fi adresate în diverse contexte, fie că este vorba de sesiuni de formare, forumuri online, comunități de practică sau discuții cu colegii și mentorii.

Soluția cea mai rapidă ar fi o aplicație web prin care oricine ar putea să adreseze o întrebare și să răspundă la o altă întrebare. O astfel de platformă ar permite utilizatorilor să își împărtășească cunoștințele și să beneficieze de experiența altora, facilitând astfel un mediu de învățare colaborativ și dinamic. Utilizatorii ar putea să își creeze profile, să urmărească subiectele de interes și să primească notificări atunci când apar răspunsuri noi sau întrebări relevante.

Pentru a dezvolta o astfel de aplicație cel mai bine ar fi să folosim strategia de randare a conținutului pe partea serverului(SSR). SSR, cunoscută și sub numele de redare universală sau izomorfă, este o metodă alternativă de redare pentru aplicațiile de tip single-page. SSR generează marcajul HTML static pe server, astfel încât browserul primește o pagină HTML complet redată. Acest lucru se realizează folosind un runtime backend, cum ar fi Node.js, care poate rula codul JavaScript pentru a construi componentele UI. O aplicație SSR procesează același cod TypeScript atât pe partea clientului, cât și pe partea serverului – de aceea este numită și redare universală.

Pe scurt, redarea pe partea serverului constă în următorii pași:

1. Cererea HTTP a clientului – Când utilizatorul introduce URL-ul în bara de adrese a browserului, se stabilește o conexiune HTTP cu serverul, apoi trimite serverului o cerere pentru documentul HTML.
2. Preluarea datelor – Serverul preia orice date necesare din baza de date sau din API-urile terțe.
3. Pre-redarea pe partea serverului – Serverul compilează componentele JavaScript în HTML static.
4. Răspunsul HTTP al serverului – Serverul trimite acest document HTML către client.
5. Încărcarea și redarea paginii – Clientul descarcă fișierul HTML și afișează componentele statice pe pagină.
6. Hidratarea – Clientul descarcă fișierul(ele) JavaScript încorporate în HTML, procesează codul și atașează ascultători de evenimente componentelor. Acest proces este numit și hidratare sau rehidratare.



1

¹ <https://www.reactpwa.com/docs/en/feature-ssr.html>

Capitolul 1. Fundamentarea teoretică și documentarea bibliografică

Capitolul 1.1. Fundamente Teoretice

Aplicatia la nivel de backend si la nivel de frontend este realizata in framework-ul Next.JS care este de tip fullstack.

La nivel frontend se folosesc componente React scrise în limbajul Typescript fiind stilizate cu ajutorul framework-ului bazat pe css Tailwind.css si librarii precum shaden si prism.css.

Backend-ul aplicației este integrat în același proiect Next.js, folosind API Routes și Server Components pentru a gestiona cererile HTTP și pentru a oferi date dinamic utilizatorilor. Utilizăm Server Components pentru a îmbunătăți performanța și scalabilitatea aplicației. Acestea permit executarea codului pe server, reducând timpul de încărcare a paginilor și oferind o experiență de utilizare mai rapidă.

Conectarea cu baza de date se face prin intermediul Mongoose ce este o librărie de tip Object Data Modeling pentru MongoDB. Am optat pentru MongoDB deoarece e un sistem de baze de date NoSQL avand o structura bazata pe documente.

React este o bibliotecă JavaScript, pentru construirea interfețelor de utilizator, în special a celor care necesită actualizări frecvente și dinamice. React facilitează crearea de aplicații web prin utilizarea unui model bazat pe componente, unde fiecare componentă reprezintă o parte a interfeței de utilizator și gestionează propria sa stare. Unul dintre avantajele principale ale React este eficiența sa, obținută prin utilizarea unui DOM virtual, care minimizează actualizările costisitoare ale DOM-ului real. React folosește JSX, un preprocesor care permite scrierea de cod HTML direct în JavaScript, făcând codul mai ușor de înțeles și de întreținut.

NEXT.js este un framework React astfel încât să faciliteze dezvoltarea aplicațiilor web moderne, oferind o serie de caracteristici integrate care simplifică procesele complexe. Una dintre aceste caracteristici este routarea automată, care elimină necesitatea configurării manuale a rutelor, oferirea unui suport nativ pentru rendering pe server (SSR) și generare statică (SSG), permițând dezvoltatorilor să aleagă metoda de redare cea mai potrivită pentru fiecare pagină. Acest lucru îmbunătățește performanța și SEO fără configurări suplimentare. Optimizarea automată a imaginilor este asigurată de componenta Image, care reduce dimensiunea fișierelor și le convertește în formate moderne, cum ar fi WebP, pentru a accelera încărcarea paginilor, permiterea și crearea de rute API în cadrul aplicației prin simpla adăugare de fișiere în directorul api, eliminând necesitatea configurării unui server backend separat și reducând complexitatea și efortul de întreținere. Framework-ul include suport pentru stilizarea componentelor folosind CSS și Sass, permițând importul fișierelor de stil direct în componente.

Redarea incrementală statică (ISR) permite actualizarea periodică a paginilor statice, combinând beneficiile generării statice și dinamice, astfel încât să puteți oferi conținut proaspăt fără a sacrifica performanța. Framework-ul oferă, de asemenea, suport nativ pentru TypeScript, facilitând scrierea de coduri tipizate și reducând erorile la compilare.

SSR, sau Server-Side Rendering, este o tehnică în dezvoltarea web prin care HTML-ul unei pagini este generat pe server în momentul cererii și apoi trimis către client. Aceasta este diferită de CSR (Client-Side Rendering), unde HTML-ul este generat de browser folosind JavaScript după ce pagina a fost încărcată.

În cadrul SSR, când un utilizator solicită o pagină web, serverul procesează cererea, rulează logica aplicației pentru a genera HTML-ul final și apoi trimite acest HTML către browserul utilizatorului. Acest proces asigură că paginile sunt livrate complet formate și gata de afișare, reducând timpul de încărcare perceput de utilizator.

Unul dintre cele mai mari avantaje ale SSR este optimizarea pentru motoarele de căutare (SEO). Deoarece conținutul paginii este disponibil imediat sub formă de HTML complet, motoarele de căutare pot indexa mai ușor și mai eficient paginile web, comparativ cu CSR, unde conținutul ar putea să nu fie complet disponibil până când JavaScript-ul nu este executat.

SSR poate îmbunătăți, de asemenea, performanța percepută, mai ales pentru utilizatorii cu conexiuni lente la internet sau dispozitive mai puțin performante. Deoarece HTML-ul este deja generat pe server, browserul utilizatorului trebuie doar să-l afișeze, fără a rula JavaScript complex pentru a construi pagina.

Rutarea în Next.js poate fi făcută în două feluri, prin Page Router sau prin App Router. Am ales App Router deoarece oferă o flexibilitate și scalabilitate mai mare pentru aplicațiile complexe, permițând gestionarea mai ușoară a stării și integrarea cu alte librării.

TypeScript este un superset sintactic al JavaScript care adaugă tipizare statică. Acest lucru înseamnă practic că TypeScript adaugă o sintaxă peste JavaScript, permițând dezvoltatorilor să adauge tipuri. JavaScript este un limbaj cu tipizare slabă. Poate fi dificil să înțelegi ce tipuri de date sunt transferate în JavaScript. În JavaScript, parametrii funcțiilor și variabilele nu au nicio informație despre tipuri! Astfel, dezvoltatorii trebuie să consulte documentația sau să ghicească pe baza implementării. TypeScript permite specificarea tipurilor de date care sunt transferate în cod și are capacitatea de a raporta erori atunci când tipurile nu se potrivesc.

MongoDB este o bază de date orientată pe documente, cu scalabilitatea și flexibilitatea pe care le dorești și cu capacitățile de interogare și indexare de care ai nevoie. MongoDB stochează datele în documente flexibile, asemănătoare cu JSON, ceea ce înseamnă că câmpurile pot varia de la un document la altul, iar structura datelor poate fi schimbată în timp. Modelul de documente se mapă la obiectele din codul aplicației tale, făcând datele ușor de gestionat. Interogările, indexarea și agregarea în timp real oferă modalități puternice de acces și analiză a

datelor tale.[4]

Pentru a integra baza de date mai ușor voi folosi o bibliotecă ODM bibliotecă aleasă fiind Mongoose

ODM este un model de proiectare utilizat pentru a mappa structurile de date dintr-o bază de date la obiectele dintr-un limbaj de programare. Aceasta facilitează manipularea datelor într-un mod orientat pe obiecte, oferind dezvoltatorilor un mod simplu de a defini scheme, de a valida date, de a efectua interogări și de a gestiona relațiile dintre date, toate acestea într-un mod care este coerent cu modelul de programare orientată pe obiecte.

Mongoose este o bibliotecă Object Data Modeling (ODM) pentru Node.js și MongoDB. Aceasta facilitează definirea schemelor pentru documentele MongoDB, validarea datelor și interogarea acestora. Mongoose permite crearea și manipularea simplă a documentelor prin modele, oferă validare a datelor înainte de salvare, suport pentru middlewares și funcționalități puternice de interogare și populare a referințelor între colecții. Este utilizată pe scară largă pentru a gestiona datele într-un mod structurat și robust în aplicațiile Node.js care folosesc MongoDB.[10]

Capitolul 2.2. Proiecte Similare

Un pas esențial în dezvoltarea unei astfel de aplicații este investigarea și înțelegerea structurii unor proiecte similare din același domeniu de interes. Acest proces oferă o perspectivă extinsă asupra evoluției și tendințelor tehnologice, furnizând totodată un punct de pornire pentru evaluarea competitivității în această industrie.

Aplicații asemănătoare se pot face în mai multe framework-uri bazate pe diferite limbaje de programare.

Nuxt.js este un framework construit pe baza Vue.js, conceput pentru crearea aplicațiilor cu redare pe server și site-uri statice. Oferă o experiență de dezvoltare îmbunătățită prin caracteristici precum rutare automată, gestionarea stării și modularitate. Cu Nuxt.js, dezvoltatorii pot construi aplicații performante, optimizate pentru SEO, care se încarcă rapid și oferă o experiență excelentă utilizatorilor.

Django este un framework web de nivel înalt pentru Python, care încurajează dezvoltarea rapidă și designul curat și pragmatic. Combinat cu Django REST Framework, oferă o soluție robustă pentru construirea API-urilor RESTful și a aplicațiilor web cu redare pe server. Django include un sistem de templating puternic, autentificare, rutare și multe alte caracteristici care facilitează dezvoltarea aplicațiilor web complexe.

Ruby on Rails este un framework complet pentru dezvoltarea aplicațiilor web, scris în Ruby. Oferă tot ce este necesar pentru a crea aplicații web bazate pe baze de date, conform modelului Model-View-Controller (MVC). Ruby on Rails include instrumente pentru redare pe

server, dezvoltarea API-urilor și gestionarea migrărilor bazei de date, toate acestea facilitând dezvoltarea rapidă și eficientă a aplicațiilor web.

Laravel este un framework PHP conceput pentru dezvoltarea aplicațiilor web, cu o sintaxă elegantă și expresivă. Laravel oferă redare pe server, rutare, autentificare și un ORM robust numit Eloquent. Este cunoscut pentru ușurința de utilizare și pentru faptul că permite dezvoltatorilor să scrie cod clar și bine structurat. Laravel include, de asemenea, suport pentru API-uri RESTful și migrații ale bazei de date, facilitând dezvoltarea de aplicații web complete și scalabile.

ASP.NET Core este un framework C# cross-platform, de înaltă performanță, pentru construirea de aplicații moderne, bazate pe cloud și conectate la internet. Include suport pentru redare pe server cu Razor Pages și Blazor, permițând dezvoltatorilor să creeze aplicații web interactive și scalabile. ASP.NET Core oferă, de asemenea, o gamă largă de instrumente și biblioteci pentru autentificare, autorizare, rutare și acces la baze de date.

Capitolul 2.3. Specificații funcționale

Aplicația este de tip Întrebări și Răspunsuri, destinată utilizatorilor care doresc să adreseze întrebări și să primească răspunsuri de la alți utilizatori.

Utilizatorii trebuie să se înregistreze și să se autentifice pentru a putea posta întrebări și răspunsuri. Vor exista opțiuni de autentificare prin email/parolă și Github. Utilizatorii pot posta întrebări în diferite categorii din domeniul tehnologiei informației, având posibilitatea de a adăuga tag-uri relevante pentru fiecare întrebare și opțiunea de a adăuga descriere detaliată și cod aferent. Aplicația va include o funcționalitate de căutare avansată pe baza cuvintelor cheie, tag-uri și categorii, permițând filtrarea întrebărilor după popularitate, dată, număr de răspunsuri și alte criterii. Navigarea va fi intuitivă și prietenoasă. Utilizatorii pot răspunde la întrebările postate de alți utilizatori, având posibilitatea de a edita și șterge propriile răspunsuri și de a adăuga fișiere atașate la răspunsuri. Utilizatorii vor putea personaliza interfața aplicației (teme, layout-uri) și setările de confidențialitate și securitate. Interfața va fi responsivă, adaptându-se pentru dispozitive mobile, tablete și desktop-uri, oferind un UX/UI modern și intuitiv.

Capitolul 2. Proiectarea aplicației

Capitolul 3.1. Analiza platformei hardware

Am ales un laptop echipat cu procesorul Intel i7-9750H și sistemul de operare Windows pentru a implementa o aplicație web din mai multe motive bine fundamentate. În primul rând, procesorul Intel i7-9750H este un model de generație recentă, parte din seria Coffee Lake, oferind performanțe excelente datorită celor șase nuclee și a frecvenței de bază de 2.6 GHz, care poate ajunge până la 4.5 GHz în modul turbo. Această capacitate de procesare este esențială pentru dezvoltarea și testarea aplicațiilor web, care necesită putere de calcul semnificativă, mai ales în fazele de compilare și rulare a serverelor locale.

Pe lângă performanțele brute ale procesorului, i7-9750H oferă o eficiență energetică bună și o capacitate excelentă de multitasking, ceea ce este crucial atunci când se lucrează cu multiple instanțe de software, cum ar fi un server de baze de date, un server web și un mediu de dezvoltare integrat (IDE), toate rulând simultan. Aceasta îmi permite să dezvolt, testezi și să depanezi aplicația web fără întârzieri sau încetiniri semnificative, asigurând un flux de lucru neîntrerupt și productiv.

În ceea ce privește sistemul de operare, Windows este alegerea mea datorită compatibilității sale largi cu diverse instrumente de dezvoltare și software necesare pentru implementarea unei aplicații web. Windows oferă suport excelent pentru cele mai populare IDE-uri, cum ar fi Visual Studio Code, IntelliJ IDEA, și multe altele, precum și pentru servere locale precum XAMPP, WAMP sau Docker. De asemenea, Windows are o comunitate mare de dezvoltatori și o multitudine de resurse și tutoriale disponibile, ceea ce facilitează rezolvarea eventualelor probleme tehnice care pot apărea în timpul dezvoltării.

Un alt avantaj al utilizării Windows este integrarea sa perfectă cu suita Microsoft Office și alte instrumente de productivitate, care pot fi esențiale pentru documentația proiectului, managementul sarcinilor și comunicarea eficientă cu echipa. Mai mult, Windows oferă un mediu de lucru familiar și intuitiv, cu actualizări regulate de securitate și performanță, asigurând astfel un ecosistem stabil și sigur pentru dezvoltarea aplicațiilor web.

În acest sens, alegerea unui laptop cu procesor Intel i7-9750H și sistem de operare Windows pentru implementarea unei aplicații web se bazează pe combinația optimă de performanță, eficiență, compatibilitate software și suport extins, toate contribuind la un proces de dezvoltare fluid și productiv.

Capitolul 3.2. Proiectarea aplicației - idei generale

Aplicația la nivel de backend și la nivel de frontend este realizată în framework-ul Next.js 14 care este de tip fullstack.

La nivelul frontend-ului, se utilizează componente React scrise în TypeScript, stilizate cu ajutorul framework-ului CSS Tailwind.css, precum și cu librării precum shadcn și prism.css.[5][6][7]

Backend-ul aplicației este integrat în același proiect Next.js, folosind API Routes și Server Components pentru a gestiona cererile HTTP și pentru a oferi date dinamic utilizatorilor. Utilizăm Server Components pentru a îmbunătăți performanța și scalabilitatea aplicației. Acestea permit executarea codului pe server, reducând timpul de încărcare a paginilor și oferind o experiență de utilizare mai rapidă.

Conectarea cu baza de date se face prin intermediul Mongoose ce este o librărie de tip Object Data Modeling pentru MongoDB. Am optat pentru MongoDB deoarece e un sistem de baze de date NoSQL având o structură bazată pe documente.

```

{
  "answers": {
    "_id": "object",
    "author": "object",
    "question": "object",
    "content": "string",
    "createdAt": "object",
    "__v": "number"
  },
  "questions": {
    "_id": "object",
    "title": "string",
    "content": "string",
    "tags": "object",
    "views": "number",
    "author": "object",
    "answers": "object",
    "createdAt": "object",
    "__v": "number"
  },
  "users": {
    "_id": "object",
    "User_Id": "string",
    "name": "string",
    "username": "string",
    "email": "string",
    "password": "string",
    "picture": "string",
    "saved": "object",
    "joinedAt": "object",
    "__v": "number"
  }
},
{
  "interactions": {
    "_id": "object",
    "user": "object",
    "action": "string",
    "question": "object",
    "tags": "object",
    "createdAt": "object",
    "__v": "number",
    "answer": "object"
  },
  "tags": {
    "_id": "object",
    "__v": "number",
    "createdOn": "object",
    "followers": "object",
    "name": "string",
    "questions": "object"
  }
},

```

Figura 1: Colectiile aplicației și atributele acestora

Din figura 1 se poate observa că baza de date MongoDB conține mai multe colecții, fiecare având o structură specifică definită de diverse câmpuri și tipurile acestora.

Colecția "interactions" include următoarele câmpuri: "user" de tip object, "action" de tip string, "question" de tip object, "tags" de tip object, "createdAt" de tip object și "answer" de tip object. Această colecție este utilizată pentru a gestiona interacțiunile utilizatorilor, capturând acțiunile și întrebările asociate.

Colecția "tags" conține câmpurile: "createdOn" de tip object, "followers" de tip object, "name" de tip string și "questions" de tip object. Această colecție gestionează etichetele asociate cu întrebările, inclusiv informațiile despre momentul creării și despre cei care urmăresc aceste etichete.

Colecția "answers" include următoarele câmpuri: "author" de tip object, "question" de tip object, "content" de tip string și "createdAt" de tip object. Această colecție este utilizată pentru a gestiona răspunsurile la întrebări, capturând detalii despre autor și conținutul răspunsului.

Colecția "questions" conține câmpurile: "title" de tip string, "content" de tip string, "tags" de tip object, "views" de tip number, "author" de tip object, "answers" de tip object și "createdAt" de tip object. Această colecție este utilizată pentru a gestiona întrebările, inclusiv titlul, conținutul, etichetele asociate, numărul de vizualizări și răspunsurile primite.

Colecția "users" include următoarele câmpuri: "User_Id" de tip string, "name" de tip string, "username" de tip string, "email" de tip string, "password" de tip string, "picture" de tip string, "saved" de tip object și "joinedAt" de tip object. Această colecție este utilizată pentru a gestiona informațiile utilizatorilor, inclusiv detaliile de autentificare și profilul personal.

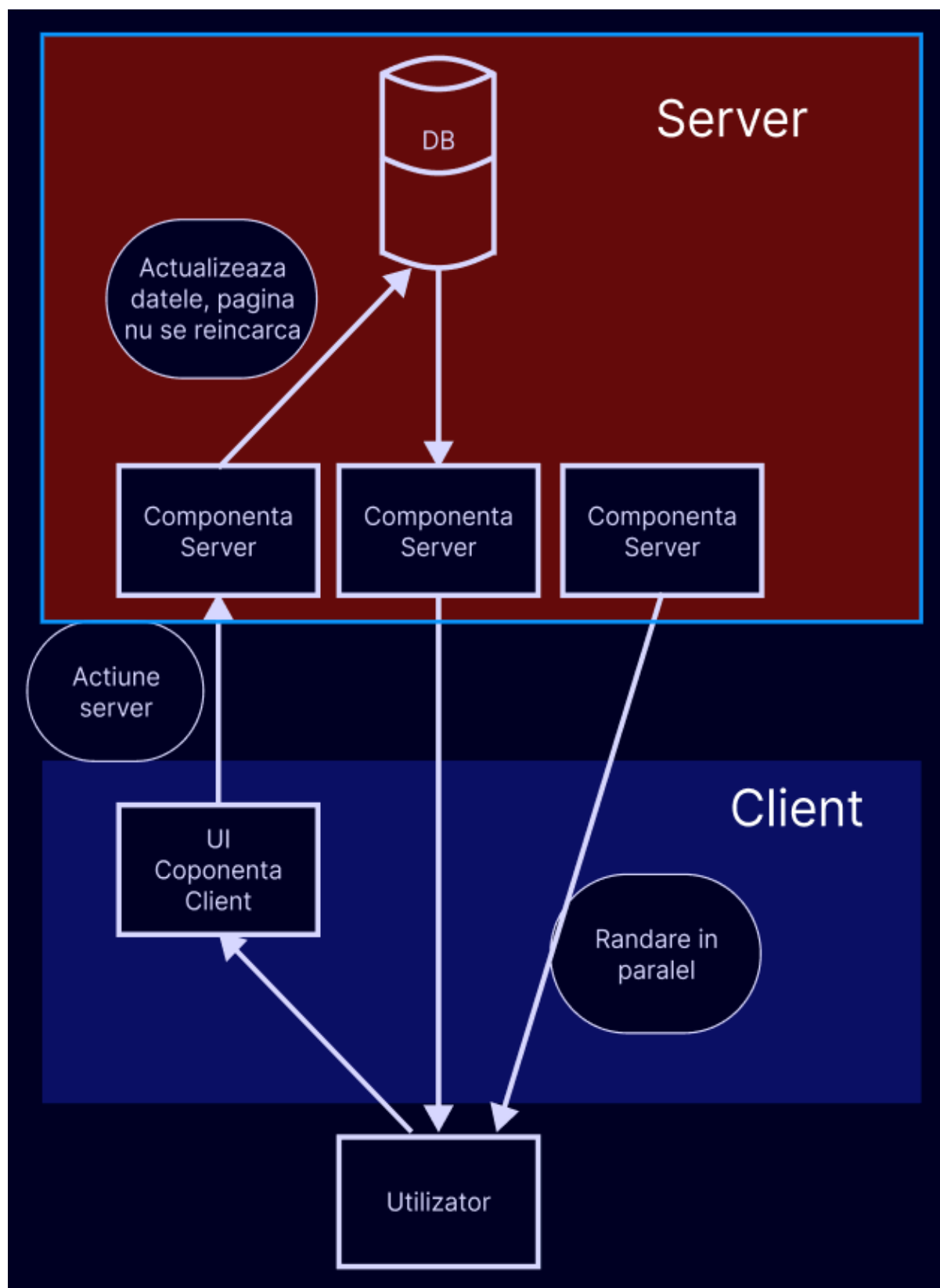


Figura 2: Schema bloc ce evidențiază funcționarea aplicației

Conform figurei 2, am acțiuni care se desfășoară pe server și mai multe componente, unele de tip server și altele de tip client. Acțiunile, reprezentate de fișiere cu funcții care rulează pe server, sunt cele care facilitează operațiile CRUD în baza de date. Componentele de tip server se încarcă doar dacă este accesată ruta website-ului

Fișierul **QuestionActions** are în prim plan modelul question și se referă la crearea de întrebări, ștergerea de întrebări, editarea și vizualizarea acestora. Aceste operații sunt esențiale pentru gestionarea eficientă a conținutului pe platforma de întrebări și răspunsuri. Mai precis:

Funcția **getQuestions** permite obținerea unei liste de întrebări pe baza unor criterii de căutare, filtrare și paginare, oferind utilizatorilor posibilitatea de a găsi rapid informațiile de care au nevoie.

Funcția **createQuestion** facilitează adăugarea de noi întrebări, asigurându-se că acestea sunt corect asociate cu etichetele relevante și autorul, și actualizează instantaneu interfața utilizatorului prin revalidarea căii.

Funcția **getQuestionById** oferă detalii complete despre o întrebare specifică, inclusiv informații despre etichete și autor, ceea ce este crucial pentru vizualizarea detaliată a conținutului.

Funcția **deleteQuestion** asigură eliminarea completă a unei întrebări și a tuturor datelor asociate (răspunsuri, interacțiuni), menținând astfel baza de date curată și actualizată.

Funcția **editQuestion** permite modificarea titlului și conținutului unei întrebări existente, oferind flexibilitate utilizatorilor de a-și corecta sau actualiza întrebările.

Funcția **getHotQuestions** scoate în evidență întrebările cele mai vizualizate, ajutând la promovarea conținutului popular și relevant.

Funcția **getRecommendedQuestions** oferă utilizatorilor sugestii personalizate de întrebări bazate pe interacțiunile lor anterioare, îmbunătățind astfel experiența de utilizare prin prezentarea de conținut relevant.

Funcția **viewQuestion** crește numărul de vizualizări ale unei întrebări și înregistrează interacțiunea, contribuind la urmărirea popularității și implicării utilizatorilor.

Fișierul **AnswerActions** are în prim plan modelul answer și se referă la gestionarea răspunsurilor în cadrul platformei. Aceasta include operațiuni esențiale pentru crearea, obținerea și ștergerea răspunsurilor:

Funcția **createAnswer** permite utilizatorilor să adauge un nou răspuns la o întrebare existentă. După crearea răspunsului, acesta este asociat cu întrebarea relevantă și se înregistrează o interacțiune corespunzătoare pentru utilizator.

Funcția **getAnswers** recuperează lista de răspunsuri pentru o întrebare specifică, aplicând criterii de sortare și paginare pentru a organiza eficient informațiile afișate utilizatorilor.

Funcția `deleteAnswer` se ocupă de eliminarea unui răspuns din baza de date, actualizând în același timp întrebările și interacțiunile pentru a reflecta schimbările făcute.

Fișierul `SearchActions` are în vedere implementarea unei funcții de căutare globală care permite căutarea în mai multe tipuri de date din baza de date, cum ar fi întrebări, utilizatori, răspunsuri și etichete.

Funcția `globalSearch` primește parametri de căutare (`query` și `type`), se conectează la baza de date și caută în funcție de acești parametri. Dacă nu este specificat un tip de căutare sau dacă tipul nu este valid, funcția caută în toate modelele (`Question`, `User`, `Answer`, `Tag`) și returnează primele două rezultate pentru fiecare model. Dacă este specificat un tip de căutare valid, funcția caută în modelul corespunzător și returnează primele cinci rezultate.

Fișierul `TagActions` are în prim plan modelul tag și se referă la gestionarea etichetelor pe platformă. Aceasta include funcții pentru obținerea celor mai interacționate etichete, recuperarea tuturor etichetelor, obținerea întrebărilor asociate unei etichete specifice și identificarea celor mai populare etichete.

Funcția `getTopInteractedTags` obține cele mai interacționate etichete pentru un utilizator specific.

Funcția `getAllTags` recuperează toate etichetele din baza de date pe baza unor criterii de căutare, filtrare și paginare.

Funcția `getQuestionsByTagId` obține întrebările asociate unei etichete specifice, aplicând criterii de căutare și paginare.

Funcția `getTopPopularTags`: recuperează cele mai populare etichete pe baza numărului de întrebări asociate.

Fișierul `UserActions` are în prim plan modelul user și se referă la mai multe operații esențiale pentru gestionarea utilizatorilor:

Funcția `getUserById` obține un utilizator după ID, căutând în baza de date un utilizator cu un anumit id.

Funcția `createUser` creează un utilizator nou în baza de date folosind datele furnizate în parametrii funcției.

Funcția `getAllUsers` recuperează toți utilizatorii din baza de date, permițând opțiuni de căutare și filtrare, precum și paginare.

Funcția `getUserInfo` obține informații detaliate despre un utilizator specific, inclusiv numărul total de întrebări și răspunsuri postate de acesta.

Funcția `getUserAnswers` recuperează răspunsurile unui utilizator specific, permițând paginarea pentru a gestiona seturi mari de date.

Funcția `getUserQuestions` recuperează întrebările postate de un utilizator specific, permițând de asemenea paginarea și sortarea după anumite criterii.

Înregistrarea și autentificarea sunt componente esențiale ale aplicațiilor web moderne, care asigură securitatea și personalizarea experienței utilizatorilor. Ca librării am folosit `bcryptjs`, care este utilizată pentru securizarea parolelor prin algoritmul `bcrypt`, `uuid v4` pentru generarea unor ID-uri unice pentru fiecare utilizator și `NextAuth`, care conține mecanisme pentru autentificare folosind mai mulți provideri, inclusiv autentificarea prin credențiale și autentificarea prin GitHub.

Înregistrarea unui nou utilizator se realizează prin endpoint-ul de tip POST. Procesul consta în: primirea și validarea datelor care sunt preluate din cererea HTTP și validate pentru a se asigura că toate câmpurile necesare (email, parolă, nume, username) sunt prezente, conectarea la baza de date pentru verificarea existenței utilizatorului în baza de date prin căutarea unui document cu același email, securizarea parolei prin hashurare folosind librăria `bcryptjs`[8], un nou document utilizator este creat în baza de date cu datele validate și parola hash-uită. În plus, fiecare utilizator primește un id unic generat folosind librăria `uuid v4`.

Autentificarea utilizatorilor se realizează folosind `NextAuth` și mai mulți provideri de autentificare, inclusiv autentificarea prin credențiale (email și parolă) și GitHub. Procesul de autentificare implică: sunt configurați providerii de autentificare. În acest caz, avem un provider de credențiale și unul de GitHub, pentru providerul de credențiale, se validează utilizatorul prin verificarea emailului și compararea parolei hash-uite stocate în baza de date, după o sesiune JWT este creată pentru utilizatorul autentificat, care conține datele utilizatorului necesare pentru personalizarea experienței.[11]

Componenta server **Home** este prima cu care se întâlnește utilizatorul. Aceasta este alcătuită din mai multe componente client și servește drept punct central pentru afișarea întrebărilor. Printre componentele client utilizate se numără **Filter**, **HomeFilter**, **LocalSearch**, **NoResult**, și **QuestionCard**.

Filter și **HomeFilter** permit utilizatorilor să filtreze întrebările în funcție de diferite criterii, asigurându-se că aceștia pot naviga rapid și eficient printre întrebările disponibile. **LocalSearch** oferă funcționalitatea de căutare directă în cadrul întrebărilor, ajutând utilizatorii să găsească rapid informațiile de care au nevoie.

Componenta **QuestionCard** este responsabilă pentru afișarea individuală a fiecărei întrebări, incluzând detalii precum titlul, autorul, tagurile, numărul de vizualizări și răspunsuri, și data creării. Aceasta asigură o prezentare clară și concisă a fiecărei întrebări, facilitând interacțiunea utilizatorilor cu conținutul platformei.

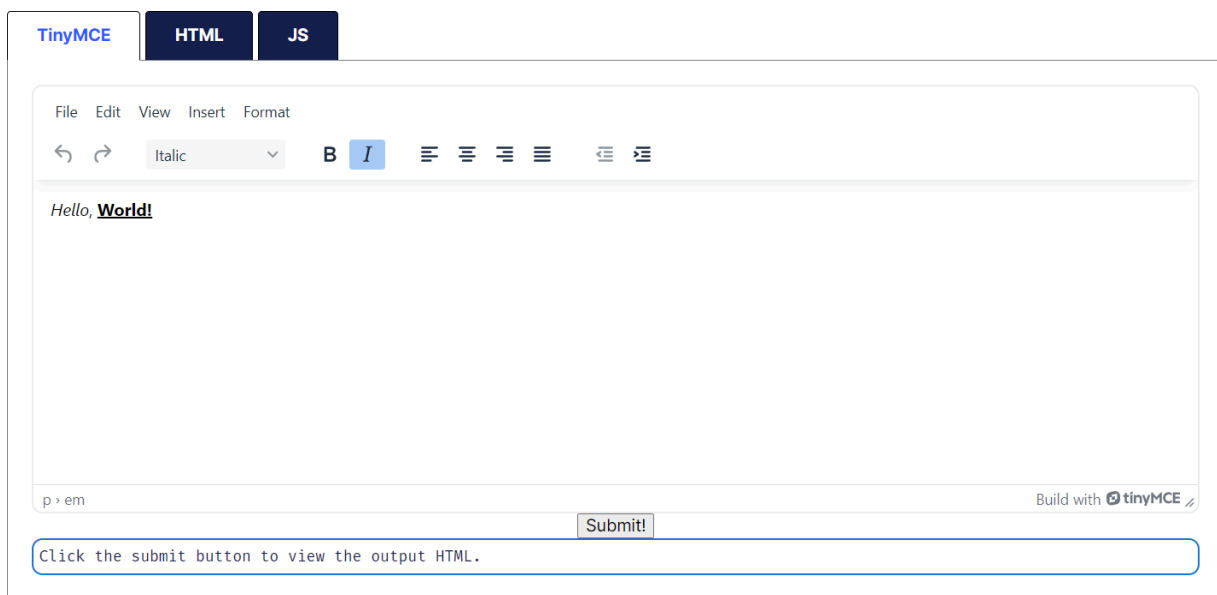
În cazul în care nu există întrebări care să corespundă criteriilor de căutare sau filtrare,

componenta **NoResult** este afișată pentru a informa utilizatorul că nu s-au găsit rezultate și pentru a-l încuraja să fie primul care adresează o întrebare.

Această structură modulară și bine organizată permite componentelor client să fie reutilizabile și ușor de întreținut, contribuind la o experiență de utilizare fluidă și intuitivă. Prin combinarea eficientă a cererilor asincrone pentru obținerea datelor și redarea condiționată a componentelor, **Home** asigură atât personalizarea experienței utilizatorului cât și performanța optimă a aplicației web.

Altă componentă importantă este cea pentru pagina **AskQuestions**. Aceasta este responsabilă pentru gestionarea procesului de creare a unei noi întrebări sau de editare a unei întrebări existente. Componenta server verifică mai întâi sesiunea utilizatorului pentru a asigura că utilizatorul este autentificat. În caz contrar, utilizatorul este redirecționat către pagina principală. Dacă utilizatorul este autentificat, componenta obține datele utilizatorului din baza de date și le transmite componentei **Question**.

Componentele client incluse în **Question** gestionează formularul de creare sau editare a întrebărilor. Folosind **react-hook-form** și **zod** pentru validarea formularului, componenta asigură că titlul întrebării, explicația detaliată și etichetele sunt completate corect. Editorul de text **TinyMCE** este utilizat pentru a oferi un mediu de editare avansat pentru explicația întrebării. Utilizatorii pot adăuga etichete prin intermediul unui câmp de input specializat, iar validarea asigură că fiecare etichetă este corespunzătoare.[3]



2

² <https://www.tiny.cloud/docs/tinymce/latest/introduction-to-tinymce/>

Pe lângă gestionarea formularului, componenta **Question** oferă feedback vizual utilizatorilor, indicând dacă formularul este în curs de trimitere și schimbând butonul de trimitere pentru a reflecta acest lucru (e.g., "Posting..." sau "Editing..."). În funcție de tipul de acțiune (creare sau editare), componenta face apeluri la funcțiile **createQuestion** sau **editQuestion** pentru a salva datele în baza de date.

Altă componentă importantă este cea pentru pagina **Community**. Aceasta este responsabilă pentru afișarea utilizatorilor înregistrați în sistem, oferind o interfață de căutare, filtrare și navigare prin paginile de rezultate.

Începând cu titlul principal "All Users", componenta își propune să prezinte vizual utilizatorii sistemului. Utilizând componente ca **LocalSearch**, utilizatorii pot căuta în mod local în lista de utilizatori, oferind o modalitate rapidă de găsire a unui anumit utilizator sau profil. Componenta **Filter** permite utilizatorilor să aplice filtre pe baza unor criterii predefinite din **UserFilters**, facilitând navigarea și rafinarea rezultatelor afișate.

Secțiunea principală a componentei utilizează **UserCard** pentru a afișa fiecare utilizator într-un format compact, fiecare card fiind generat din lista de utilizatori returnată de funcția **getAllUsers**. În cazul în care nu sunt utilizatori disponibili conform criteriilor de căutare și filtrare, se afișează un mesaj informativ care încurajează utilizatorii să se înregistreze, însoțit de un link către pagina de înregistrare pentru a încuraja noi utilizatori să se alăture comunității.

Pentru a gestiona navigarea între pagini, componenta folosește **Pagination**, permițând utilizatorilor să exploreze paginile de utilizatori disponibili. Aceasta adaugă un nivel suplimentar de interacțiune și permite utilizatorilor să navigheze rapid prin rezultatele care depășesc o singură pagină.

Pagina profilului este constituită din mai multe componente care împreună oferă o prezentare detaliată și interactivă a informațiilor despre utilizatorul vizat.

În partea superioară a paginii, se găsește o secțiune dedicată informațiilor de bază despre utilizator, precum imaginea de profil, numele și username-ul acestuia. Imaginea de profil este afișată într-o formă rotundă și este obținută din **baza de date**, asigurând o vizualizare estetică și personalizată a utilizatorului.

Sub aceste detalii, sunt afișate informații suplimentare relevante despre utilizator, cum ar fi website-ul portofoliului, locația și data la care s-a alăturat platformei, fiecare afișată utilizând componente **ProfileLink** sau funcția **getJoinedDate** pentru dată.

Secțiunea **Stats** prezintă statisticile utilizatorului, inclusiv numărul total de întrebări și răspunsuri date, oferind o imagine de ansamblu rapidă asupra activității sale pe platformă.

Componenta centrală a paginii este reprezentată de **Tabs**, care permite navigarea între diferitele secțiuni ale profilului. Utilizatorul poate comuta între "Top Posts" și "Answers", fiecare reprezentată de câte o file (Tab) în care sunt afișate componente specializate (**QuestionTab** și

AnswersTab) ce listă întrebările cele mai apreciate și răspunsurile date de utilizator.

Pagina cu etichete constă într-o serie de componente care colaborează pentru a oferi utilizatorilor posibilitatea de a căuta și filtra etichete relevante, precum și de a naviga printre ele într-un mod eficient și intuitiv.

În partea de sus a paginii, se află titlul "All Tags" care informează utilizatorii despre conținutul paginii. Imediat sub acesta, utilizatorii pot găsi un motor de căutare local (**LocalSearch**), care le permite să caute etichete specifice utilizând un câmp de căutare și un buton de filtrare.

Componenta **Filter** oferă opțiuni suplimentare de filtrare, bazate pe constantele definite în **TagFilters**, permițând utilizatorilor să rafineze rezultatele afișate în funcție de criterii specifice.

Secțiunea principală a paginii este dedicată afișării etichetelor. Dacă există etichete rezultate din căutare sau filtrare, acestea sunt afișate într-un format de carduri, fiecare conținând numele etichetei și numărul de întrebări asociate acesteia. Fiecare card de etichetă este un link (**Link**) către pagina detaliată a respectivei etichete. În cazul în care nu există etichete găsite, componenta **NoResult** este afișată, oferind utilizatorilor posibilitatea de a naviga către pagina de a pune o întrebare (**ask-question**).

În final, componenta **Pagination** este afișată la baza paginii, permițând utilizatorilor să navigheze prin diferitele pagini de rezultate, dacă acestea există, indicând numărul curent al paginii și dacă există pagini următoare disponibile (**isNext**).

Pentru fiecare etichetă în parte, pagina oferă utilizatorilor posibilitatea de a vizualiza și căuta întrebările asociate acelei etichete. Componentele utilizate asigură o experiență interactivă și bine organizată.

Secțiunea principală a paginii afișează întrebările relevante utilizând componenta **QuestionCard**. Dacă există întrebări rezultate din căutare, acestea sunt afișate sub formă de carduri, fiecare card conținând detalii precum titlul întrebării, etichetele asociate, autorul, numărul de vizualizări, răspunsuri și data creării. În cazul în care nu există întrebări găsite, componenta **NoResult** este afișată, oferind utilizatorilor posibilitatea de a pune o întrebare nouă printr-un link către pagina de "ask-question".

La finalul paginii, componenta **Pagination** permite utilizatorilor să navigheze prin diferitele pagini de rezultate, indicând numărul curent al paginii și dacă există pagini următoare disponibile (**isNext**).

Capitolul 3. Implementarea Aplicației

Capitolul 4.1. Interfața cu utilizatorul

La nivel de interacțiune a utilizatorului cu aplicația, s-a implementat o interfață web utilizând componente React.js, biblioteca shdcn/ui. Aceasta cuprinde mai multe componente: un layout ce cuprinde partea de autentificare și înregistrare și un layout ce conține aplicația propriu-zisă, accesul paginilor este accesibil tuturor utilizatorilor dar conținutul acestora este parțial vizibil dacă nu ești autentificat.

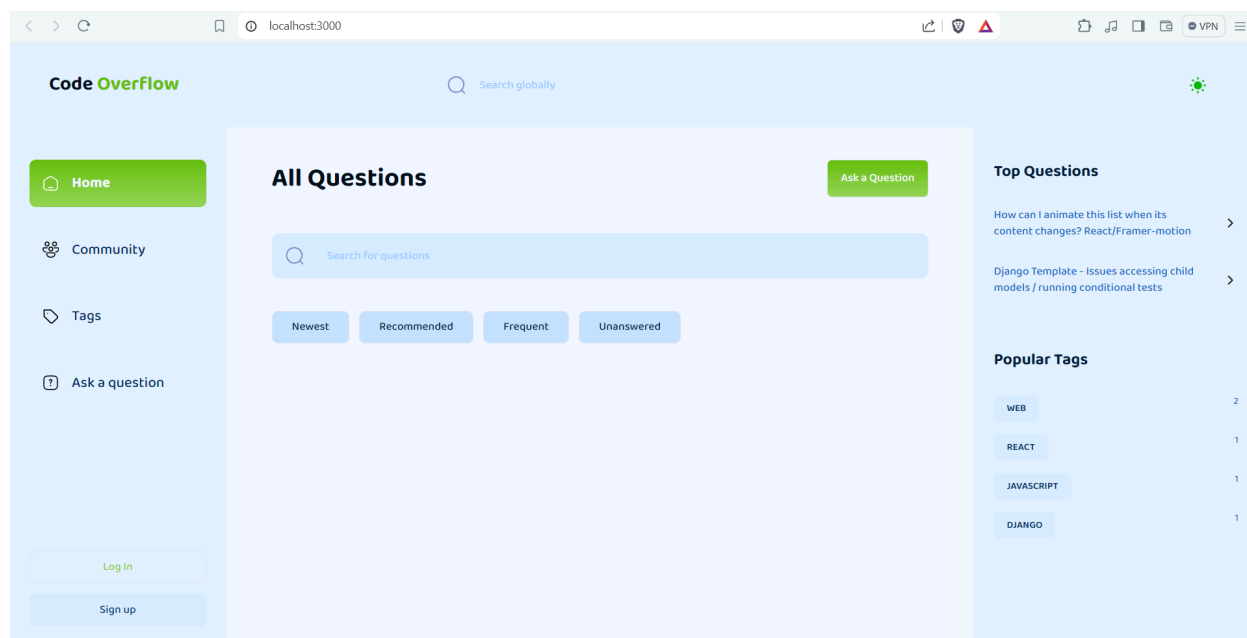
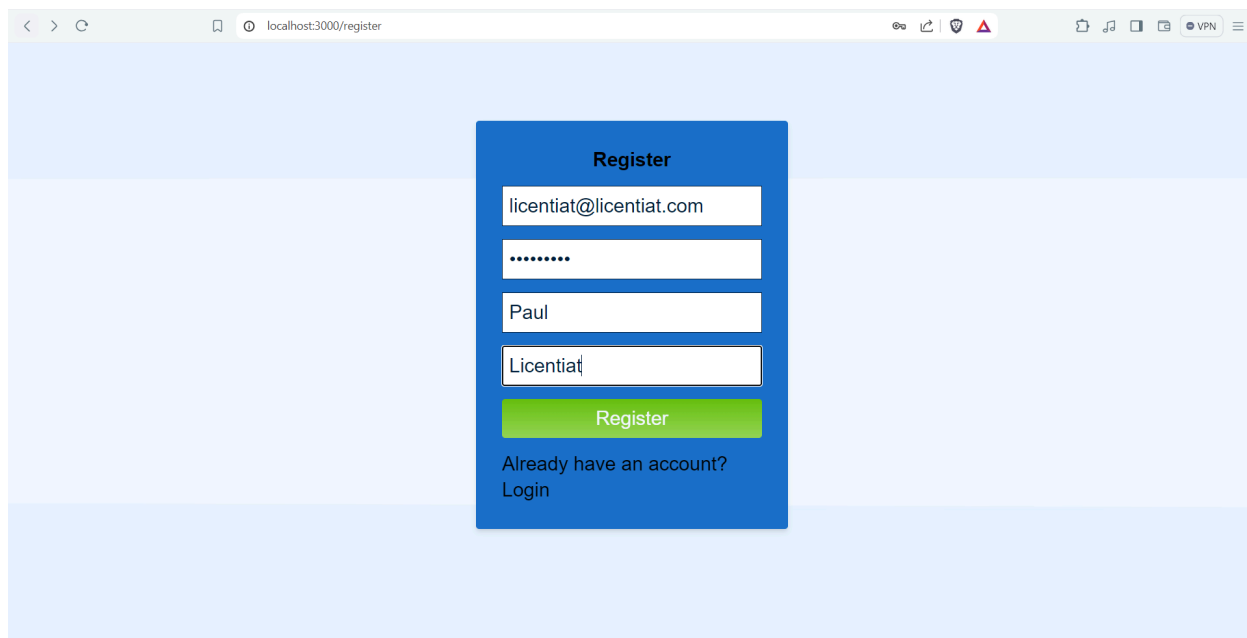


Figura 3: Pagina principală pe mod tematic luminos utilizator neautentificat

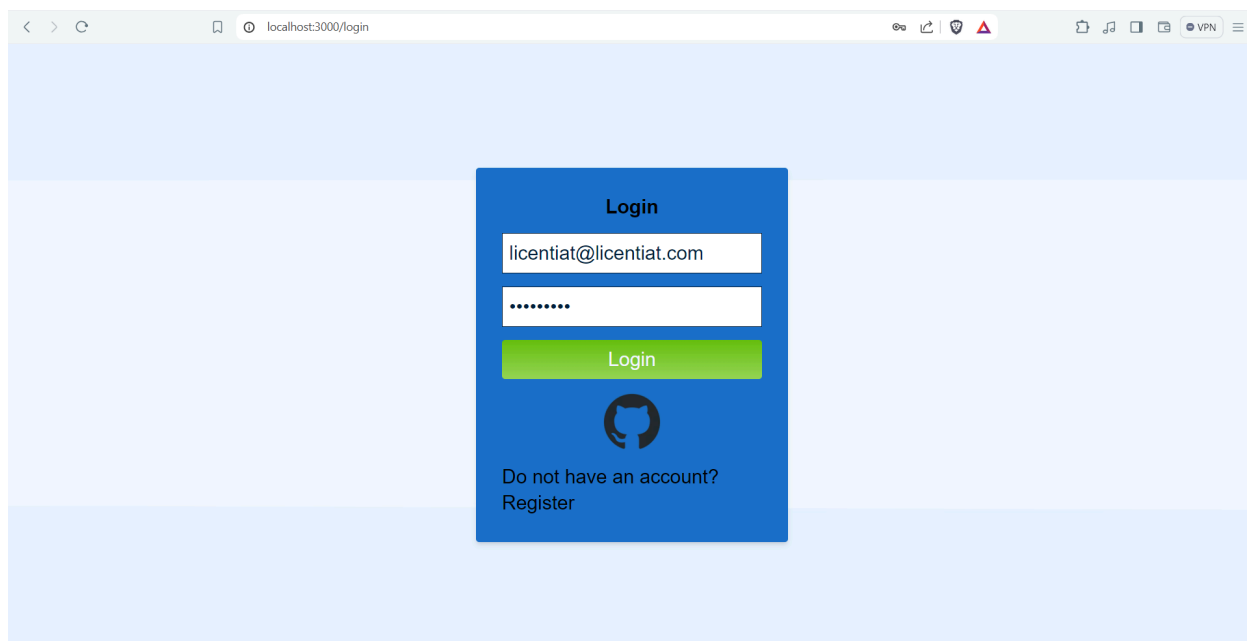
În cadrul accesării aplicației se poate observa că pagina este accesibilă și sunt prezente doar informații de bază cum ar fi ce pagini pot fi accesate, iconița din dreapta sus care indică tematica aplicației și opțiunea de a te înregistra sau autentifica.



The screenshot shows a web browser window with the address bar displaying 'localhost:3000/register'. The page has a light blue background. In the center, there is a blue rectangular form titled 'Register'. Inside the form, there are four input fields: the first contains 'licentiat@licentiat.com', the second contains '.....', the third contains 'Paul', and the fourth contains 'Licentiat'. Below these fields is a green button labeled 'Register'. At the bottom of the form, there is a link that says 'Already have an account? Login'.

Figura 4: Pagina de inregistrare

Pagina de inregistrare cu campurile completate si cu optiunea de inregistrare daca ai deja un cont.



The screenshot shows a web browser window with the address bar displaying 'localhost:3000/login'. The page has a light blue background. In the center, there is a blue rectangular form titled 'Login'. Inside the form, there are two input fields: the first contains 'licentiat@licentiat.com' and the second contains '.....'. Below these fields is a green button labeled 'Login'. Below the button is a GitHub logo. At the bottom of the form, there is a link that says 'Do not have an account? Register'.

Figura 5: Pagina de autentificare

Pagina de autentificare cu campurile completate și cu opțiunea de a te autentifica cu un provider sau de a te înregistra dacă nu ai cont. Datorită librăriei Next-Auth este posibil autentificarea cu un cont de la un provider, cum se poate observa în figura 5 am ales providerul Github fără sa fie nevoie de înregistrare în aplicație.[9]

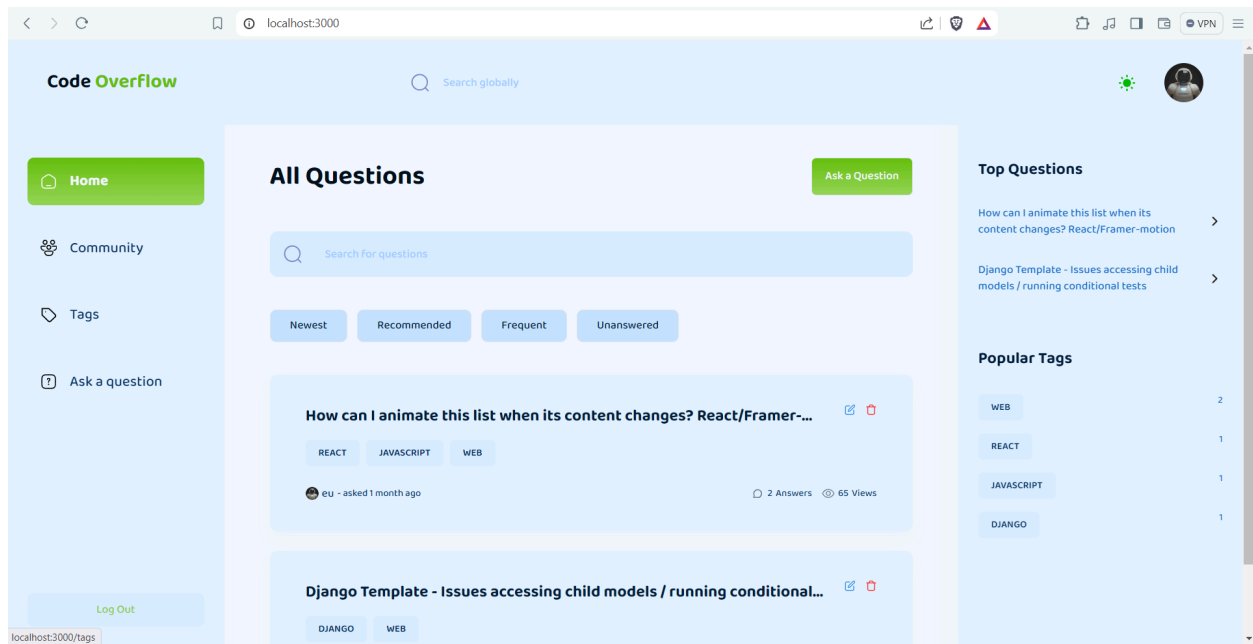


Figura 6: Pagina principală pe mod tematic luminos utilizator autentificat

După ce ne-am autentificat apare poza asociată cu utilizatorul și apare conținutul din baza de date, acum e posibil sa raspunda la intrebarile existente și să pună întrebări. Ca strategie folosim javascript web token ce e valabil 2 ore de cand s-a făcut autentificarea.

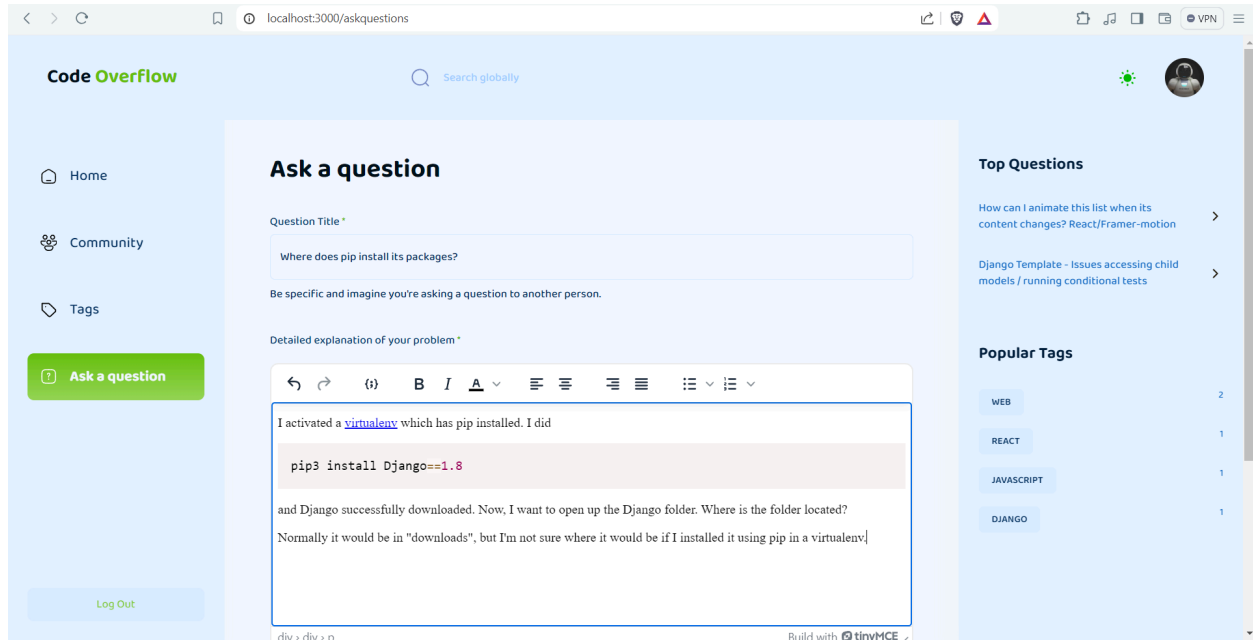


Figura 7: Pagina ask a question cu campurile completate

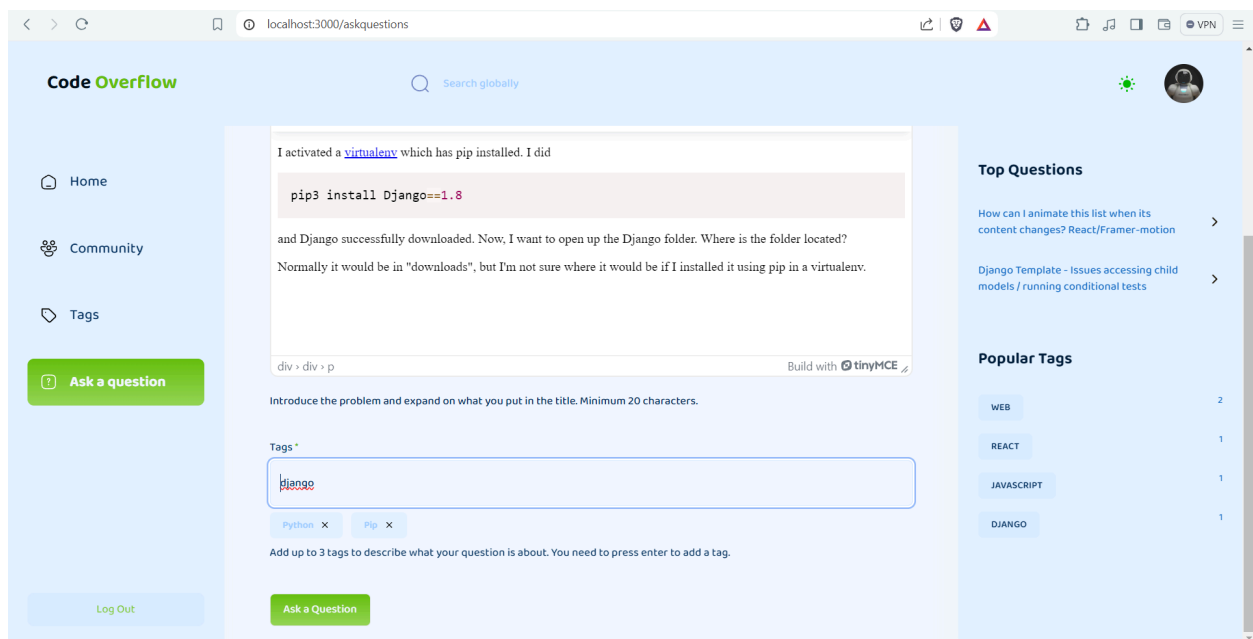


Figura 8: Pagina ask a question cu adaugarea etichetelor pentru intrebare

Cand ești autentificat este vizibil conținutul paginii și poți interacționa cu campurile formularului pentru a pune o întrebare. Datorita librăriei tinymce și cheii API specifice pot folosi editorul text de tip WYSIWYG HTML (What-You-See-Is-What-You-Get) în care pot adăuga secvențe de cod specifice limbajului de programare dorit.

Capitolul 4.2. Stocarea informațiilor

În cadrul acestei aplicații, s-a implementat o bază de date de tip NOSQL bazat pe documente MongoDB. Aceasta reprezintă o soluție open-source ce oferă performanțe ridicate și un nivel de compatibilitate bun. Serverul este de tip Cloud și rulează în Mongo Atlas și baza de date poate fi accesată din fișierele cu funcții server cu URL-ul:

```
MONGO_URL = mongodb+srv://eu:eu@cluster0.kkkpfjb.mongodb.net/licenta?retryWrites=true&w=majority&appName=Cluster0
```

Figura 9: Url-ul pentru Mongo din fișierul .env

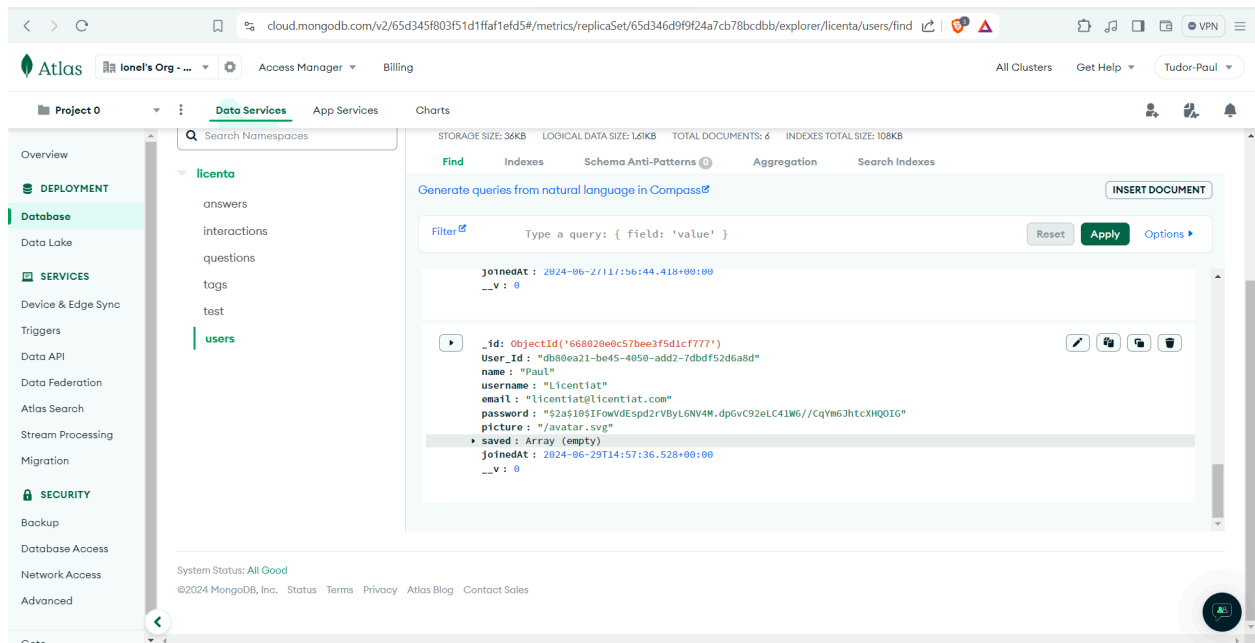


Figura 10: Documentul creat in figura 4 salvat in colectia users

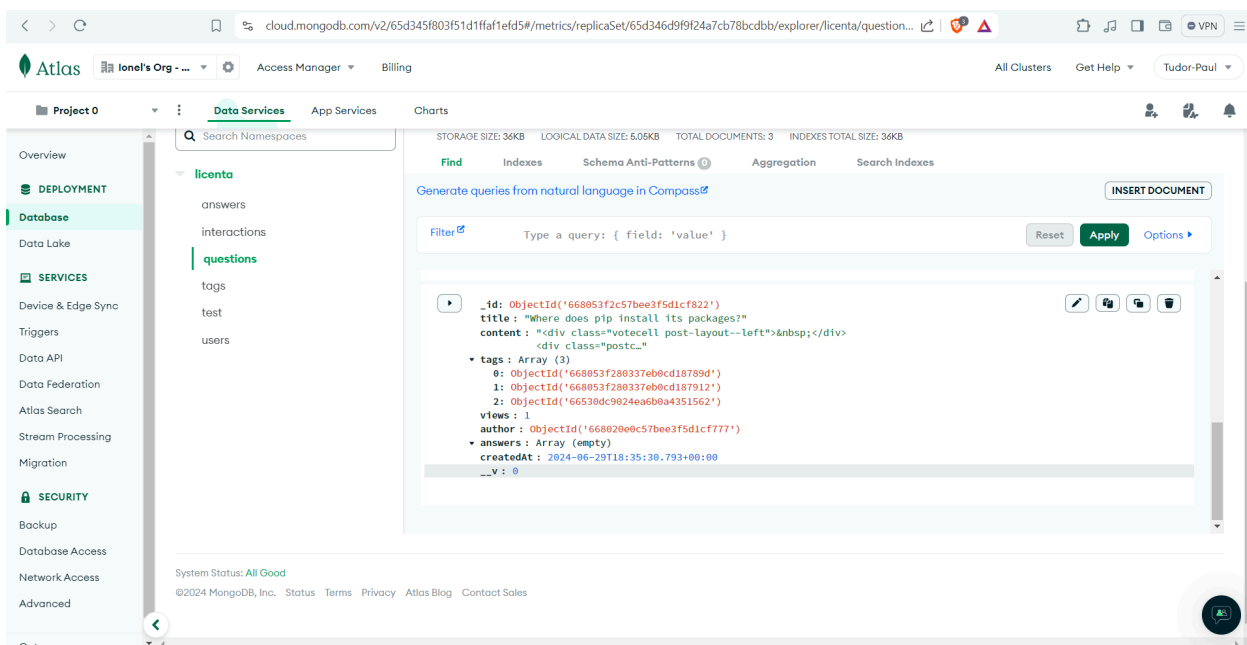


Figura 11: Documentul creat (intrebarea) in figura 7 si figura 8 salvat in colectia questions

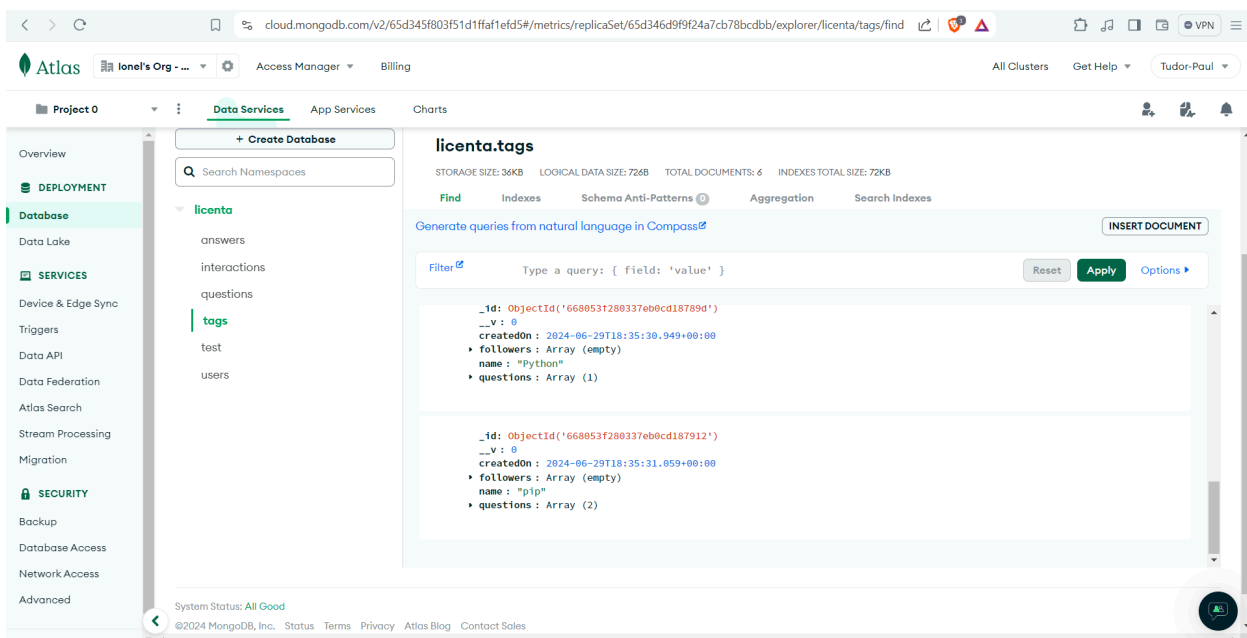


Figura 12: Documentele create (etichetele) in figura 8 salvat in colectia tags

Capitolul 4.3. Dificultăți întâmpinate și modalități de rezolvare

Problema majoră a fost cu hidratarea, hidratarea înseamnă ca fiecare HTML generat este asociat cu codul JavaScript minim necesar pentru acea pagină. Când o pagină este încărcată de browser, codul său JavaScript rulează și face pagina complet interactivă. Problema consta în

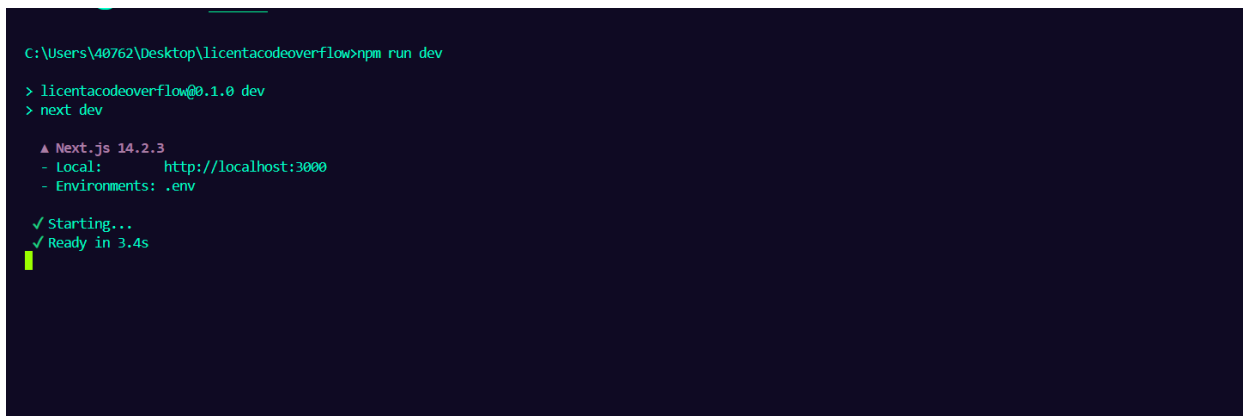
faptul că, în anumite cazuri, pot apărea neconcordanțe între HTML-ul generat pe server și cel generat pe client. Aceste neconcordanțe pot duce la erori de hidratare, cum ar fi elemente lipsă, diferențe în structură sau comportament neașteptat al componentelor. Pentru a rezolva aceasta problema am inițializat stările astfel încât să fie consistente între server și client.

O alta problema fost cu validarea datelor, astfel am folosit zod ca sa rezolv aceasta problema. este o bibliotecă TypeScript first pentru validarea și analizarea datelor, care permite definirea schemelor de validare flexibile și ușor de utilizat[2].

Capitolul 4. Testarea aplicației

Capitolul 5.1. Punerea în funcțiune a aplicației

Lansarea în execuție a aplicației se realizează utilizând comanda `npm run dev`. Rezultatul acesteia este



```
c:\Users\40762\Desktop\licentacodeoverflow>npm run dev
> licentacodeoverflow@0.1.0 dev
> next dev

  ▲ Next.js 14.2.3
  - Local:      http://localhost:3000
  - Environments: .env

✓ Starting...
✓ Ready in 3.4s
```

Figura 13: Lansarea în execuție

Astfel, dezvoltatorul este informat că aplicația a fost compilată cu succes și că utilizează portul 3000 al rețelei localhost.

Capitolul 5.2. Testarea sistemului

Testarea software este un proces esențial în dezvoltarea aplicațiilor, având scopul de a asigura calitatea și fiabilitatea produsului final. Aceasta implică verificarea și validarea software-ului pentru a descoperi erori, defecte sau neconformități față de cerințele specificate. Testarea acoperă mai multe aspecte, fiecare având un rol distinct în asigurarea calității. Testarea unităților vizează verificarea funcționalităților individuale ale componentelor software, cum ar fi funcții sau metode. Este adesea automatizată și folosește cadre de testare specifice. În contrast, testarea de integrare se concentrează pe interacțiunile dintre componentele software, având scopul de a detecta problemele care pot apărea atunci când modulele individuale sunt combinate. Testarea funcțională verifică dacă aplicația funcționează conform cerințelor specificate, implicând testarea interfețelor utilizatorilor și a altor funcționalități majore pentru a asigura comportamentul corect al aplicației. Testarea de sistem evaluează comportamentul complet al sistemului în ansamblu, incluzând testarea tuturor componentelor și modul în care acestea funcționează împreună. Testarea de acceptare asigură că software-ul îndeplinește cerințele și așteptările utilizatorilor finali, fiind adesea realizată de către aceștia sau de echipa de QA și

include testarea beta. Testarea de performanță măsoară performanța aplicației în diferite condiții de încărcare și stres, evaluând timpul de răspuns, viteza și stabilitatea sistemului. Testarea de securitate identifică vulnerabilitățile și asigură că datele sunt protejate împotriva atacurilor și accesului neautorizat, fiind crucială pentru aplicațiile care gestionează date sensibile.

Mai întâi am testat posibilitatea de a rula aplicația pe mai multe browsere pentru a asigura compatibilitatea și funcționalitatea consistentă pe diverse platforme. Testarea aplicației pe mai multe browsere este crucială deoarece utilizatorii folosesc o varietate de browsere cu diferite motoare de redare, standarde de compatibilitate și interpretări ale codului HTML, CSS și JavaScript.

Fiecare browser poate gestiona elementele web diferit, ceea ce poate duce la comportamente variate ale aplicației. Prin testarea pe mai multe browsere, se identifică și se corectează erorile specifice fiecărui browser, asigurând o experiență uniformă pentru toți utilizatorii, indiferent de browserul utilizat. De asemenea, acest tip de testare contribuie la accesibilitatea și utilizabilitatea aplicației, îmbunătățind satisfacția și încrederea utilizatorilor.

Compatibilitatea pe diverse browsere este esențială și pentru a preveni problemele de performanță și de securitate care pot varia între browsere. Prin asigurarea că aplicația funcționează corect pe toate platformele majore, se minimizează riscul de a pierde utilizatori din cauza incompatibilităților tehnice și se maximizează accesibilitatea aplicației pe piața digitală.

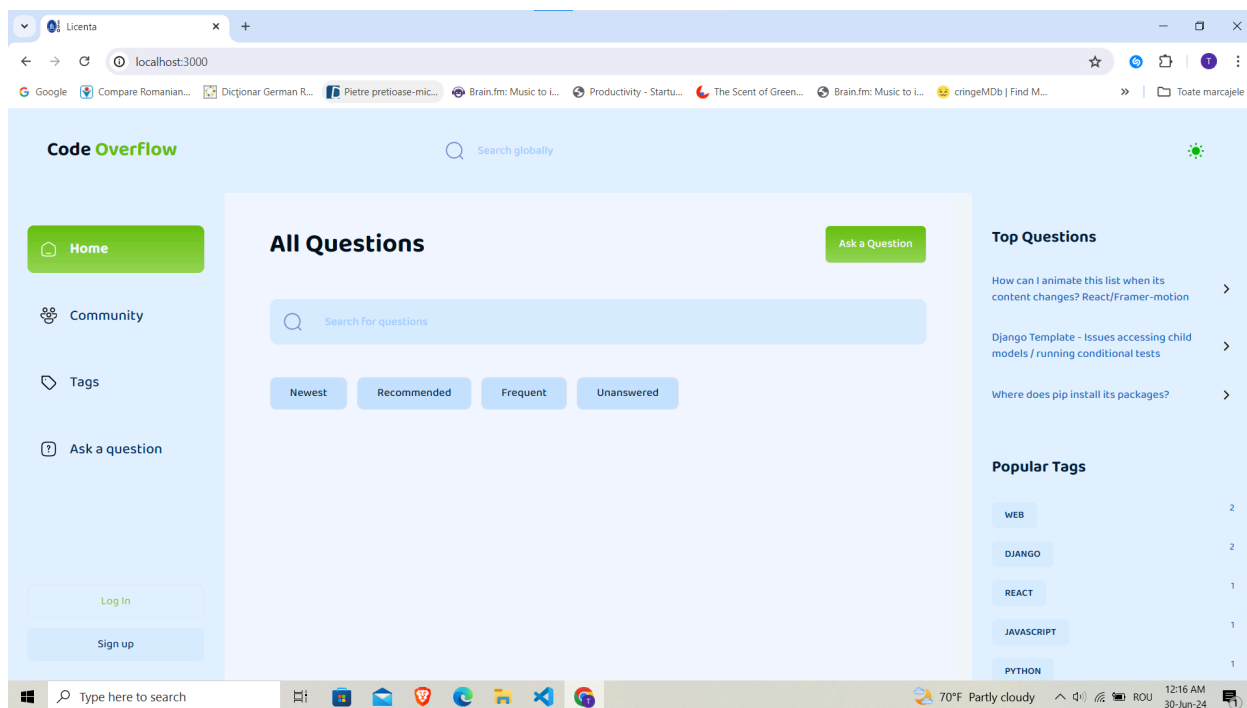


Figura 14: Lansarea aplicatiei in Google Chrome

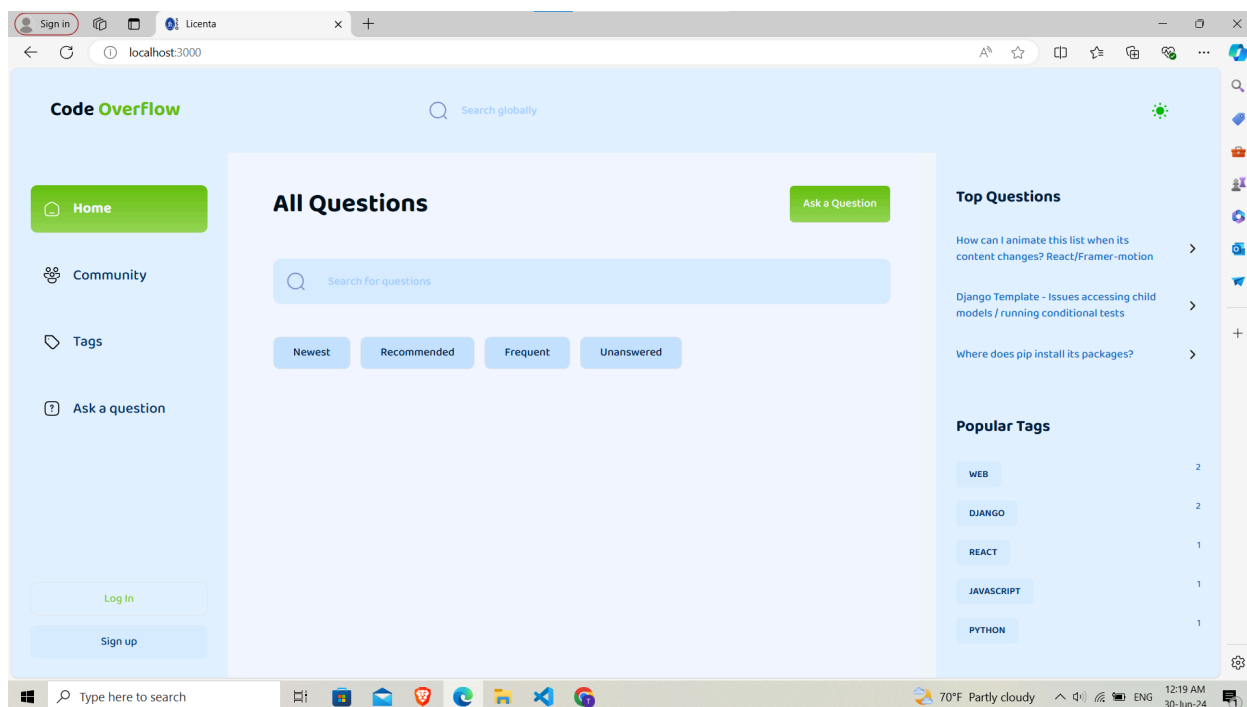


Figura 15: Lansarea aplicatiei in Microsoft Edge

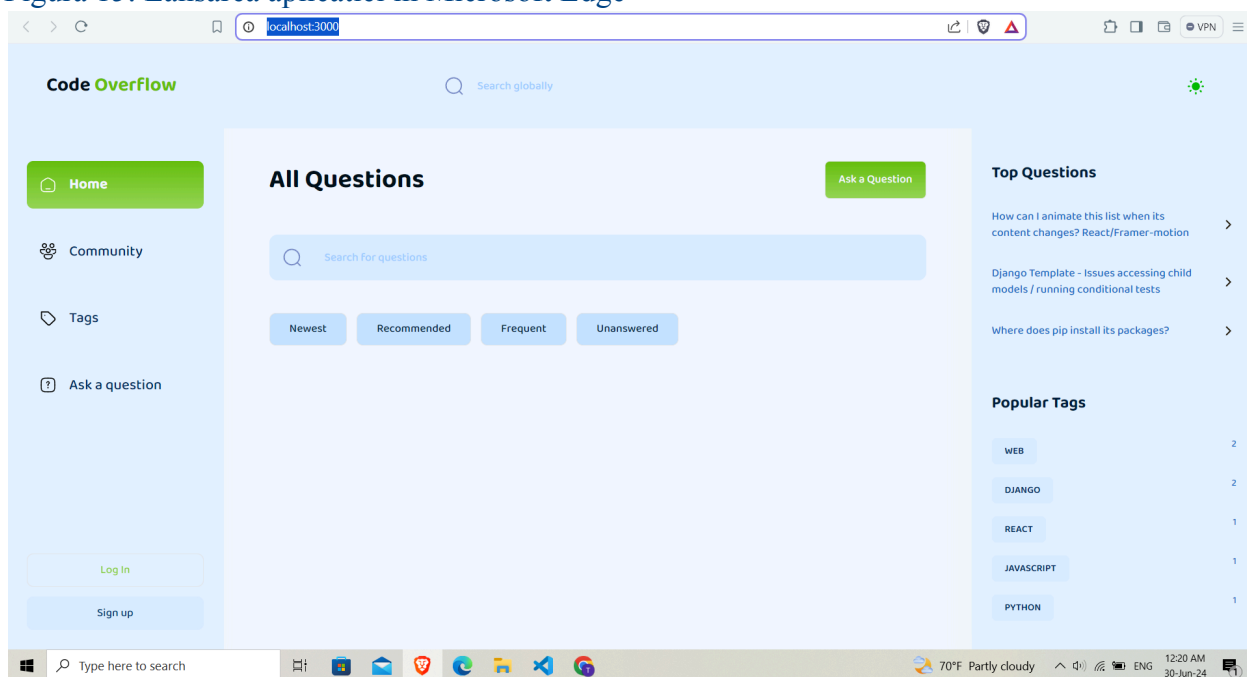


Figura 16: Lansarea aplicației în Brave

Mai întâi am testat posibilitatea de a rula aplicația pe mai multe browsere pentru a asigura compatibilitatea și funcționalitatea consistentă pe diverse platforme. Testarea aplicației pe mai multe browsere

În cadrul proiectului de față, s-au executat o serie de teste funcționale menite a verifica funcționarea corectă a aplicației:

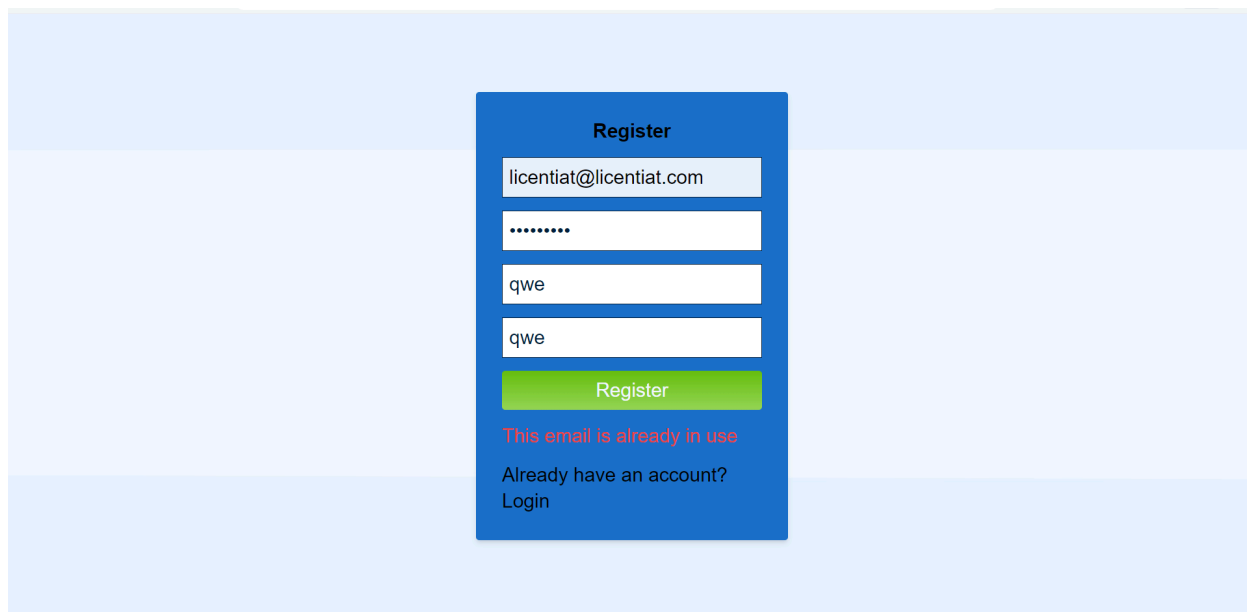


Figura 17: Înregistrarea cu un mail existent(Rezultat dorit sa imi arate mesaj ca mail-ul e deja utilizat)

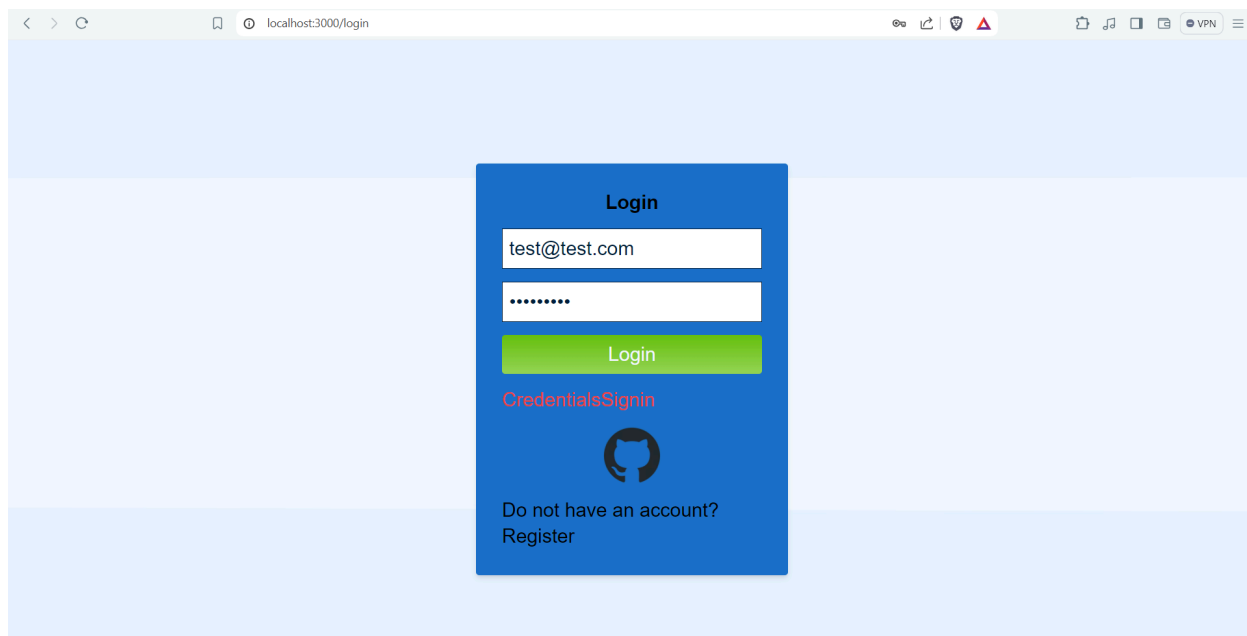


Figura 18: Autentificarea cu un mail inexistent(Rezultat dorit sa imi arate mesaj ca nu este contul)

Ask a question

Question Title *

Be specific and imagine you're asking a question to another person.
String must contain at least 5 character(s)

Detailed explanation of your problem *

Introduce the problem and expand on what you put in the title. Minimum 20 characters.
String must contain at least 20 character(s)

Tags *

Add tags...

Php x Javascript x C x Cpp x

Add up to 3 tags to describe what your question is about. You need to press enter to add a tag.
Array must contain at most 3 element(s)

Ask a Question

Figura 19: Încercarea postarii unei întrebări cu mai multe nereguli (titlul mai scurt de 5 caractere, explicația mai scurta de 20 de caractere și adaugarea a mai mult de 3 etichete)

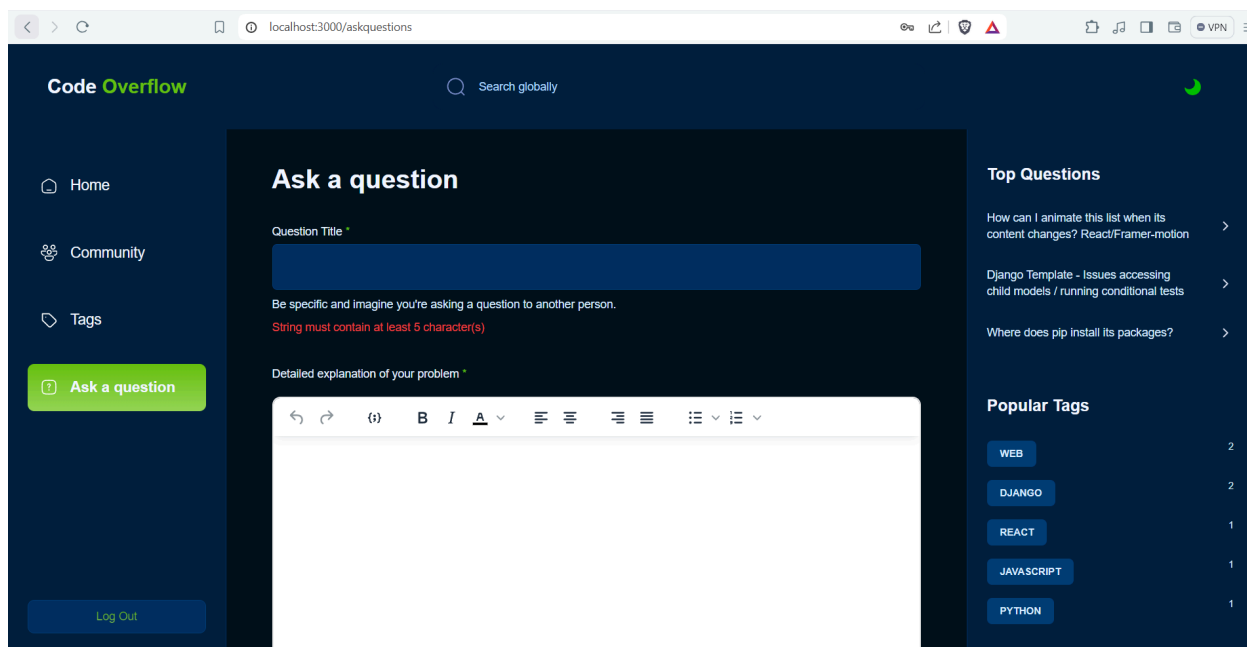
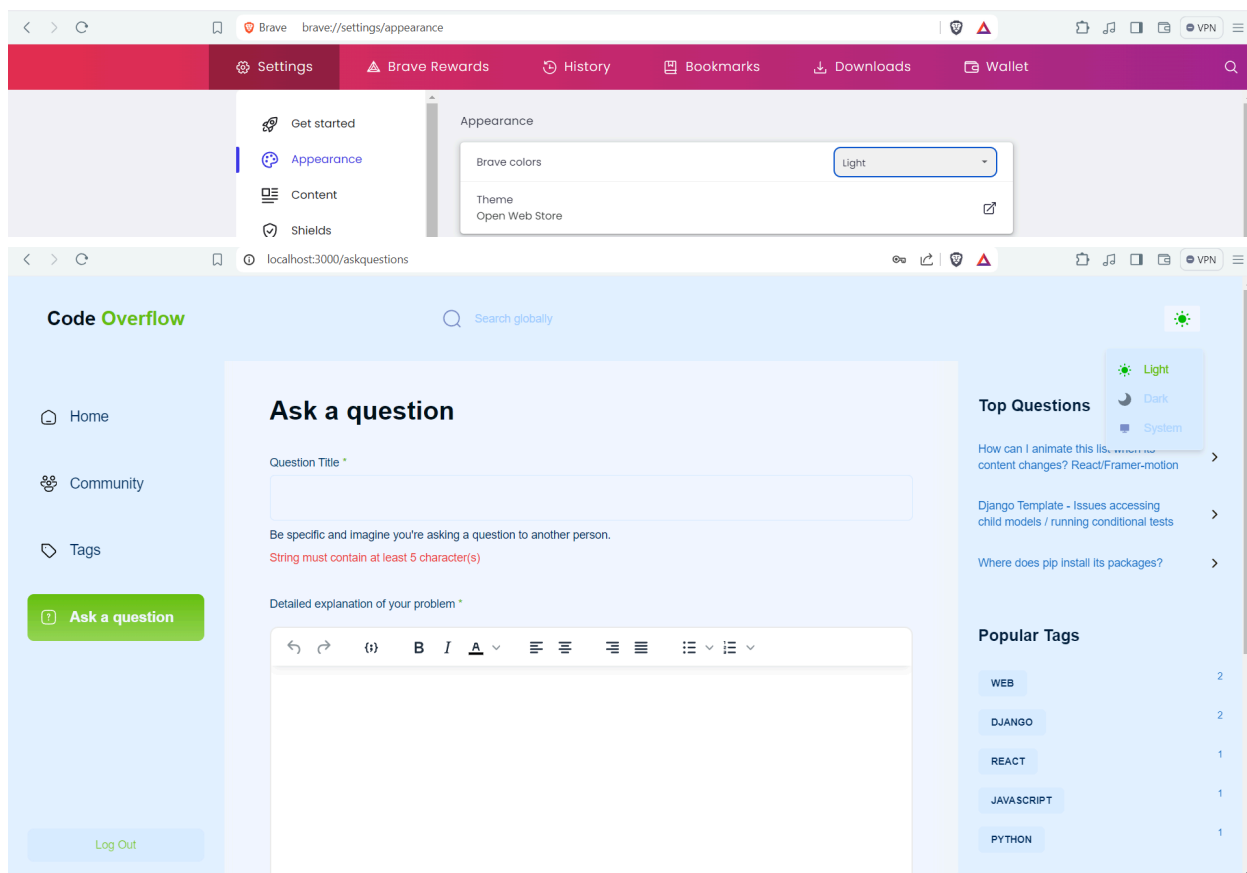


Figura 20: La fel ca în figura 19 utilizând tematică întunecată



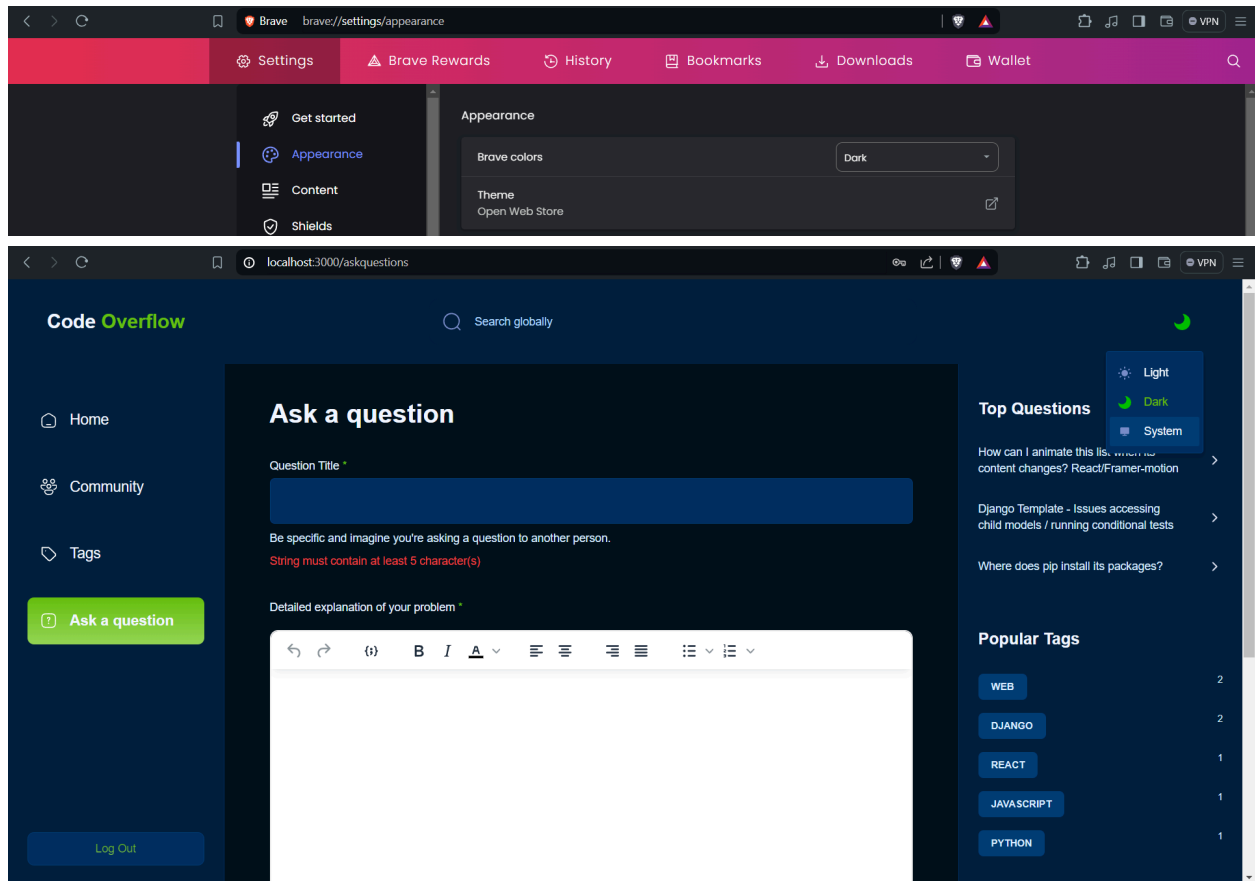


Figura 21: Testăm funcționarea schimbării tematică conform browserului

Concluziile Lucrării

În concluzie, pot spune că aplicația implementată servește unui domeniu popular în zilele noastre și permite navigarea cât mai ușoară a utilizatorilor pentru a beneficia de informații cât mai rapid. Printr-o structură bine organizată și componente interconectate, aplicația îmbunătățește semnificativ experiența utilizatorilor. Funcționalitățile de bază, cum ar fi filtrarea întrebărilor, crearea întrebărilor și capacitatea de a răspunde la acestea, sunt integrate eficient pentru a răspunde nevoilor utilizatorilor. Securitatea datelor utilizatorilor este tratată cu prioritate. Aplicația utilizează token-uri JWT pentru autentificare și autorizare, asigurând că doar utilizatorii autentificați pot accesa anumite funcționalități. Parolele sunt criptate înainte de a fi stocate în baza de date, iar utilizarea filtrelor JWT garantează că toate cererile sunt verificate și sigure.

Îmbunătățiri

Pot fi multe idei de implementat în cadrul aplicației, cum ar fi capacitatea de a comunica între utilizatori direct sau afișarea sau aplicarea pentru locuri de munca cu etichete asemănătoare cu etichetele întrebărilor postate și răspunsurilor date. O altă funcționalitate importantă ar putea fi implementarea unui sistem de notificări, astfel încât utilizatorii să fie informați în timp real despre răspunsurile la întrebările lor, mesajele primite de la alți utilizatori sau noi oportunități de muncă ce corespund etichetelor de interes.

Bibliografie

1. Vercel, <https://nextjs.org/docs>
2. Collin McDonnel, <https://zod.dev/>
3. Tiny, <https://www.tiny.cloud/docs/tinymce/latest/>
4. Shannon Bradshaw, Eoin Brazil, Kristina Chodorow “MongoDB The Definitive Guide Powerful and Scalable Data Storage 3rd Edition”, 2019
5. Shadcn, <https://ui.shadcn.com/docs>
6. Ayush Saini, “Holi Theme for prism.js”, <https://github.com/PrismJS/prism-themes/blob/master/themes/prism-holi-theme.css>
7. Noel Rappin, “Modern CSS with Tailwind: Flexible Styling Without the Fuss”, 2022
8. Nick Campbell, <https://www.npmjs.com/package/bcrypt>
9. Github, <https://docs.github.com/en/apps/oauth-apps/building-oauth-apps/authorizing-oauth-apps>
10. Mongoose, <https://mongoosejs.com/docs/>
11. Vercel, <https://authjs.dev/reference/nextjs>

Anexa

app/api/register/route.ts

```
import { dbConnect } from "@lib/db";
import User from "@model/model_user";
import bcrypt from "bcryptjs";
import { NextResponse, NextRequest } from "next/server";
import { v4 as uuidv4 } from "uuid";

export const POST = async (req: NextRequest) => {
  try {
    const body = await req.json();
    console.log("app/api/register/route: Request Body:", body);

    const { email, password, name, username } = body;

    if (!email || !password || !name || !username) {
      return new NextResponse("Missing required fields", { status: 400 });
    }

    await dbConnect();

    const existingUser = await User.findOne({ email });
    if (existingUser) {
      return new NextResponse("User already exists", { status: 400 });
    }

    const hashedPassword = await bcrypt.hash(password, 10);
    const user = new User({
      User_Id: uuidv4(),
      email,
      password: hashedPassword,
      name,
      username,
    });
```

```

    picture: "/avatar.svg",
  });

  await user.save();
  return NextResponse.json("User created", { status: 200 });
} catch (err: any) {
  console.error("app /api/register/route: Error creating user:", err);
  return new NextResponse("Internal Server Error", { status: 500 });
}
};

```

[lib/actions/AnswerAction.ts](#)

```

"use server";

import Answer from "@model/model_answer";
import { dbConnect } from "../db";
import {
  CreateAnswerParams,
  DeleteAnswerParams,
  GetAnswersParams,
} from "@types/shared.t";
import Question from "@model/model_question";
import { revalidatePath } from "next/cache";
import Interaction from "@model/model_interaction";

export async function createAnswer(params: CreateAnswerParams) {
  try {
    await dbConnect();

    const { content, author, question, path } = params;

    const newAnswer = await Answer.create({ content, author, question });

    // Add the answer to the question's answers array
    const questionObject = await Question.findByIdAndUpdate(question, {

```

```

    $push: { answers: newAnswer._id },
  });

  await Interaction.create({
    user: newAnswer.author,
    action: "answer",
    question: newAnswer.question,
    answer: newAnswer._id,
    tags: questionObject.tags,
  });

  revalidatePath(path);
} catch (error) {
  console.log(error);
  throw error;
}
}

export async function getAnswers(params: GetAnswersParams) {
  try {
    await dbConnect();

    const { questionId, sortBy, page = 1, pageSize = 10 } = params;

    const skipAmount = (page - 1) * pageSize;

    let sortOptions = {};

    switch (sortBy) {
      case "recent":
        sortOptions = { createdAt: -1 };
        break;
      case "old":
        sortOptions = { createdAt: 1 };
        break;

      default:

```

```

        break;
    }

    const answers = await Answer.find({ question: questionId })
        .populate("author", "_id name picture")
        .sort(sortOptions)
        .skip(skipAmount)
        .limit(pageSize);

    const totalAnswer = await Answer.countDocuments({
        question: questionId,
    });

    const isNextAnswer = totalAnswer > skipAmount + answers.length;

    return { answers, isNextAnswer };
} catch (error) {
    console.log(error);
    throw error;
}

export async function deleteAnswer(params: DeleteAnswerParams) {
    try {
        await dbConnect();

        const { answerId, path } = params;

        const answer = await Answer.findById(answerId);

        if (!answer) {
            throw new Error("Answer not found");
        }

        await answer.deleteOne({ _id: answerId });
        await Question.updateMany(
            { _id: answer.question },

```

```

    { $pull: { answers: answerId } }
  );
  await Interaction.deleteMany({ answer: answerId });

  revalidatePath(path);
} catch (error) {
  console.log(error);
}
}

```

lib/actions/UserAction.ts

```

"use server";

import { FilterQuery } from "mongoose";
import Answer from "@model/model_answer";
import User from "@model/model_user";
import { dbConnect } from "../db";
import {
  CreateUserParams,
  GetAllUsersParams,
  GetUserByIdParams,
  GetUserStatsParams,
} from "@types/shared.t";
import Question from "@model/model_question";

export async function getUserById(params: { userId: string }) {
  try {
    await dbConnect();

    const { userId } = params;

    const user = await User.findOne({ User_Id: userId });
    return user;
  } catch (error) {
    console.log("lib/actions/UserAction.ts: Error getting user:", error);
  }
}

```

```

        throw error;
    }
}

export async function createUser(userData: CreateUserParams) {
    try {
        dbConnect();

        const newUser = await User.create(userData);

        return newUser;
    } catch (error) {
        console.log("lib/actions/UserAction.ts: Error creating user:", error);
        throw error;
    }
}

export async function getAllUsers(params: GetAllUsersParams) {
    try {
        dbConnect();

        const { searchQuery, filter, page = 1, pageSize = 10 } = params;
        const skipAmount = (page - 1) * pageSize;

        const query: FilterQuery<typeof User> = {};

        if (searchQuery) {
            query.$or = [
                { name: { $regex: new RegExp(searchQuery, "i") } },
                { username: { $regex: new RegExp(searchQuery, "i") } },
            ];
        }

        let sortOptions = {};

        switch (filter) {
            case "new_users":

```

```

        sortOptions = { joinedAt: -1 };
        break;
    case "old_users":
        sortOptions = { joinedAt: 1 };
        break;

    default:
        break;
}

const users = await User.find(query)
    .sort(sortOptions)
    .skip(skipAmount)
    .limit(pageSize);

const totalUsers = await User.countDocuments(query);
const isNext = totalUsers > skipAmount + users.length;

return { users, isNext };
} catch (error) {
    console.log("lib/actions/UserAction.ts: Error getting all users:", error);
    throw error;
}
}

export async function getUserInfo(params: GetUserByIdParams) {
    try {
        dbConnect();

        const { userId } = params;

        const user = await User.findOne({ User_Id: userId });

        if (!user) {
            throw new Error("User not found");
        }
    }
}

```



```

const totalQuestions = await Question.countDocuments({ author: user._id });
const totalAnswers = await Answer.countDocuments({ author: user._id });

return {
  user,
  totalQuestions,
  totalAnswers,
};
} catch (error) {
  console.log("lib/actions/UserAction.ts: Error getting user info:", error);
  throw error;
}
}

export async function getUserAnswers(params: GetUserStatsParams) {
  try {
    dbConnect();

    const { userId, page = 1, pageSize = 10 } = params;

    const skipAmount = (page - 1) * pageSize;

    const totalAnswers = await Answer.countDocuments({ author: userId });

    const userAnswers = await Answer.find({ author: userId })
      .skip(skipAmount)
      .limit(pageSize)
      .populate("question", "_id title")
      .populate("author", "_id User_Id name picture");

    const isNextAnswer = totalAnswers > skipAmount + userAnswers.length;

    return { totalAnswers, answers: userAnswers, isNextAnswer };
  } catch (error) {
    console.log(
      "lib/actions/UserAction.ts: Error getting user answers:",
      error
    );
  }
}

```

```

    );
    throw error;
  }
}

export async function getUserQuestions(params: GetUserStatsParams) {
  try {
    dbConnect();

    const { userId, page = 1, pageSize = 10 } = params;

    const skipAmount = (page - 1) * pageSize;

    const totalQuestions = await Question.countDocuments({ author: userId });

    const userQuestions = await Question.find({ author: userId })
      .sort({ createdAt: -1, views: -1 })
      .skip(skipAmount)
      .limit(pageSize)
      .populate("tags", "_id name")
      .populate("author", "_id User_Id name picture");

    const isNextQuestions = totalQuestions > skipAmount + userQuestions.length;

    return { totalQuestions, questions: userQuestions, isNextQuestions };
  } catch (error) {
    console.log(
      "lib/actions/UserAction.ts: Error getting user questions:",
      error
    );
    throw error;
  }
}

```

[lib/actions/QuestionAction.ts](#)

```

/* eslint-disable camelcase */
"use server";

import Question from "@model/model_question";
import Tag from "@model/model_tag";
import { dbConnect } from "../db";
import {
  CreateQuestionParams,
  DeleteQuestionParams,
  EditQuestionParams,
  GetQuestionByIdParams,
  GetQuestionsParams,
  RecommendedParams,
  ViewQuestionParams,
} from "@types/shared.t";
import User from "@model/model_user";
import { revalidatePath } from "next/cache";
import Answer from "@model/model_answer";
import Interaction from "@model/model_interaction";
import { FilterQuery } from "mongoose";

export async function getQuestions(params: GetQuestionsParams) {
  try {
    await dbConnect();

    const { searchQuery, filter, page = 1, pageSize = 10 } = params;

    const skipAmount = (page - 1) * pageSize;

    const query: FilterQuery<typeof Question> = {};

    if (searchQuery) {
      query.$or = [
        { title: { $regex: new RegExp(searchQuery, "i") } },
        { content: { $regex: new RegExp(searchQuery, "i") } },
      ];
    }
  }
}

```

```

let sortOptions = {};

switch (filter) {
  case "newest":
    sortOptions = { createdAt: -1 };
    break;
  case "frequent":
    sortOptions = { views: -1 };
    break;
  case "unanswered":
    query.answers = { $size: 0 };
    break;
  default:
    break;
}

const questions = await Question.find(query)
  .populate({ path: "tags", model: Tag })
  .populate({ path: "author", model: User })
  .skip(skipAmount)
  .limit(pageSize)
  .sort(sortOptions);

const totalQuestions = await Question.countDocuments(query);

const isNext = totalQuestions > skipAmount + questions.length;

return { questions, isNext };
} catch (error) {
  console.log(error);
  throw error;
}
}

export async function createQuestion(params: CreateQuestionParams) {
  try {

```

```

await dbConnect();

const { title, content, tags, author, path } = params;

const question = await Question.create({
  title,
  content,
  author,
});

const tagDocuments = [];

for (const tag of tags) {
  const existingTag = await Tag.findOneAndUpdate(
    { name: { $regex: new RegExp(`^${tag}$`, "i") } },
    { $setOnInsert: { name: tag }, $push: { questions: question._id } },
    { upsert: true, new: true }
  );

  tagDocuments.push(existingTag._id);
}

await Question.findByIdAndUpdate(question._id, {
  $push: { tags: { $each: tagDocuments } },
});

await Interaction.create({
  user: author,
  action: "ask_question",
  question: question._id,
  tags: tagDocuments,
});

revalidatePath(path);
} catch (error) {
  console.log(error);
}

```

```
}
```

```
export async function getQuestionById(params: GetQuestionByIdParams) {
  try {
    await dbConnect();

    const { questionId } = params;

    const question = await Question.findById(questionId)
      .populate({ path: "tags", model: Tag, select: "_id name" })
      .populate({
        path: "author",
        model: User,
        select: "_id User_Id name picture",
      });

    return question;
  } catch (error) {
    console.log(error);
    throw error;
  }
}
```

```
export async function deleteQuestion(params: DeleteQuestionParams) {
  try {
    await dbConnect();

    const { questionId, path } = params;

    await Question.deleteOne({ _id: questionId });
    await Answer.deleteMany({ question: questionId });
    await Interaction.deleteMany({ question: questionId });
    const tags = await Tag.find({ questions: questionId });
    await Tag.updateMany(
      { questions: questionId },
      { $pull: { questions: questionId } }
    );
  }
}
```

```

for (const tag of tags) {
  const remainingQuestions = await Question.countDocuments({
    _id: { $in: tag.questions },
  });
  if (remainingQuestions === 0) {
    await Tag.deleteOne({ _id: tag._id });
  }
}

  revalidatePath(path);
} catch (error) {
  console.log(error);
}
}

export async function editQuestion(params: EditQuestionParams) {
  try {
    await dbConnect();

    const { questionId, title, content, path } = params;

    const question = await Question.findById(questionId).populate("tags");

    if (!question) {
      throw new Error("Question not found");
    }

    question.title = title;
    question.content = content;

    await question.save();

    revalidatePath(path);
  } catch (error) {
    console.log(error);
  }
}

```

```

export async function getHotQuestions() {
  try {
    await dbConnect();

    const hotQuestions = await Question.find({}).sort({ views: -1 }).limit(5);

    return hotQuestions;
  } catch (error) {
    console.log(error);
    throw error;
  }
}

export async function getRecommendedQuestions(params: RecommendedParams) {
  try {
    await dbConnect();

    const { userId, page = 1, pageSize = 20, searchQuery } = params;

    // find user
    const user = await User.findOne({ User_Id: userId });

    if (!user) {
      throw new Error("user not found");
    }

    const skipAmount = (page - 1) * pageSize;

    const userInteractions = await Interaction.find({ user: user.User_id })
      .populate("tags")
      .exec();

    const userTags = userInteractions.reduce((tags, interaction) => {
      if (interaction.tags) {
        tags = tags.concat(interaction.tags);
      }
    }, []);
  }
}

```



```

    return tags;
  }, []);

const distinctUserTagIds = [
  // @ts-ignore
  ...new Set(userTags.map((tag: any) => tag._id)),
];

const query: FilterQuery<typeof Question> = {
  $and: [
    { tags: { $in: distinctUserTagIds } },
    { author: { $ne: user._id } },
  ],
};

if (searchQuery) {
  query.$or = [
    { title: { $regex: searchQuery, $options: "i" } },
    { content: { $regex: searchQuery, $options: "i" } },
  ];
}

const totalQuestions = await Question.countDocuments(query);

const recommendedQuestions = await Question.find(query)
  .populate({
    path: "tags",
    model: Tag,
  })
  .populate({
    path: "author",
    model: User,
  })
  .skip(skipAmount)
  .limit(pageSize);

const isNext = totalQuestions > skipAmount + recommendedQuestions.length;

```

```

    return { questions: recommendedQuestions, isNext };
  } catch (error) {
    console.error("Error getting recommended questions:", error);
    throw error;
  }
}

export async function viewQuestion(params: ViewQuestionParams) {
  try {
    await dbConnect();

    const { questionId, User_Id } = params;

    // Update view count for the question
    await Question.findByIdAndUpdate(questionId, { $inc: { views: 1 } });

    if (User_Id) {
      const existingInteraction = await Interaction.findOne({
        user: User_Id,
        action: "view",
        question: questionId,
      });

      if (existingInteraction) return console.log("User has already viewed.");

      // Create interaction
      await Interaction.create({
        user: User_Id,
        action: "view",
        question: questionId,
      });
    }
  } catch (error) {
    console.log(error);
    throw error;
  }
}

```

```
}
```

lib/actions/SearchAction.ts

```
"use server";

import Question from "@model/model_question";
import { dbConnect } from "../db";
import { SearchParams } from "@types/shared.t";
import User from "@model/model_user";
import Answer from "@model/model_answer";
import Tag from "@model/model_tag";

const SearchableTypes = ["question", "answer", "user", "tag"];

export async function globalSearch(params: SearchParams) {
  try {
    await dbConnect();

    const { query, type } = params;
    const regexQuery = { $regex: query, $options: "i" };

    let results = [];

    const modelsAndTypes = [
      { model: Question, searchField: "title", type: "question" },
      { model: User, searchField: "name", type: "user" },
      { model: Answer, searchField: "content", type: "answer" },
      { model: Tag, searchField: "name", type: "tag" },
    ];

    const typeLower = type?.toLowerCase();

    if (!typeLower || !SearchableTypes.includes(typeLower)) {
      for (const { model, searchField, type } of modelsAndTypes) {
        const queryResults = await model
```

```

        .find({ [searchField]: regexQuery })
        .limit(2);

results.push(
  ...queryResults.map((item) => ({
    title:
      type === "answer"
        ? `Answers containing ${query}`
        : item[searchField],
    type,
    id:
      type === "user"
        ? item.UserId
        : type === "answer"
          ? item.question
          : item._id,
  })))
);
}
} else {
  const modelInfo = modelsAndTypes.find((item) => item.type === type);

  console.log({ modelInfo, type });
  if (!modelInfo) {
    throw new Error("Invalid search type");
  }

  const queryResults = await modelInfo.model
    .find({ [modelInfo.searchField]: regexQuery })
    .limit(5);

  results = queryResults.map((item) => ({
    title:
      type === "answer"
        ? `Answers containing ${query}`
        : item[modelInfo.searchField],
    type,
  })))
};

```

```

      id:
        type === "user"
          ? item.UserId
        : type === "answer"
          ? item.question
          : item._id,
    ));
  }

  return JSON.stringify(results);
} catch (error) {
  console.log(`Error fetching global results, ${error}`);
  throw error;
}
}

```

[lib/actions/TagAction.ts](#)

```

"use server";

import User from "@model/model_user";
import {
  GetAllTagsParams,
  GetQuestionsByTagIdParams,
  GetTopInteractedTagsParams,
} from "@types/shared.t";
import Tag, { ITag } from "@model/model_tag";
import Question from "@model/model_question";
import { FilterQuery } from "mongoose";
import { dbConnect } from "../db";

export async function getTopInteractedTags(params: GetTopInteractedTagsParams) {
  try {
    await dbConnect();

```

```

const { userId } = params;

const user = await User.findById(userId);

if (!user) throw new Error("User not found");

return [
  { _id: "1", name: "exemple_tag" },
  { _id: "2", name: "exemple_tag2" },
];
} catch (error) {
  console.log(error);
  throw error;
}
}

export async function getAllTags(params: GetAllTagsParams) {
  try {
    await dbConnect();

    const { searchQuery, filter, page = 1, pageSize = 10 } = params;
    const skipAmount = (page - 1) * pageSize;

    const query: FilterQuery<typeof Tag> = {};

    if (searchQuery) {
      query.$or = [{ name: { $regex: new RegExp(searchQuery, "i") } }];
    }

    let sortOptions = {};

    switch (filter) {
      case "popular":
        sortOptions = { questions: -1 };
        break;
      case "recent":

```

```

        sortOptions = { createdAt: -1 };
        break;
    case "name":
        sortOptions = { name: 1 };
        break;
    case "old":
        sortOptions = { createdAt: 1 };
        break;

    default:
        break;
}

const totalTags = await Tag.countDocuments(query);

const tags = await Tag.find(query)
    .sort(sortOptions)
    .skip(skipAmount)
    .limit(pageSize);

const isNext = totalTags > skipAmount + tags.length;

return { tags, isNext };
} catch (error) {
    console.log(error);
    throw error;
}
}

export async function getQuestionsByTagId(params: GetQuestionsByTagIdParams) {
    try {
        await dbConnect();

        const { tagId, page = 1, pageSize = 10, searchQuery } = params;
        const skipAmount = (page - 1) * pageSize;

        const tagFilter: FilterQuery<ITag> = { _id: tagId };

```

```

const tag = await Tag.findOne(tagFilter).populate( {
  path: "questions",
  model: Question,
  match: searchQuery
  ? { title: { $regex: searchQuery, $options: "i" } }
  : {},
  options: {
    sort: { createdAt: -1 },
    skip: skipAmount,
    limit: pageSize + 1,
  },
  populate: [
    { path: "tags", model: Tag, select: "_id name" },
    { path: "author", model: User, select: "_id User_Id name picture" },
  ],
});

if (!tag) {
  throw new Error("Tag not found");
}

const isNext = tag.questions.length > pageSize;

const questions = tag.questions;

return { tagTitle: tag.name, questions, isNext };
} catch (error) {
  console.log(error);
  throw error;
}
}

export async function getTopPopularTags() {
  try {
    await dbConnect();

```



```
const popularTags = await Tag.aggregate([
  { $project: { name: 1, numberOfQuestions: { $size: "$questions" } } },
  { $sort: { numberOfQuestions: -1 } },
  { $limit: 5 },
]);

return popularTags;
} catch (error) {
  console.log(error);
  throw error;
}
}
```