Facultatea de Matematică și Informatică

Universitatea București

Document de Analiză și Proiectare pentru Aplicația Software

Platformă de colaborare pentru cercetători în domeniul construcțiilor

Vocheci Maximilian

maximilian.vocheci@s.unibuc.ro

Data

06.06.2025

1. Introducere

1.1 Scopul documentului

Acest document oferă o documentație tehnică completă pentru o aplicație web de colaborare destinată cercetătorilor din domeniul construcțiilor. Scopul documentului este de a prezenta analiza, proiectarea și implementarea aplicației, acoperind cerințele, arhitectura sistemului, detaliile de implementare, modul de

testare, aspectele de deployment și concluziile rezultate. Documentul se adresează atât dezvoltatorilor implicați în proiect, cât și părților interesate care doresc să înțeleagă funcționalitatea și structura sistemului.

1.2 Domeniul de aplicare

Aplicația reprezintă o platformă de colaborare pentru cercetători în domeniul construcțiilor, oferind un mediu virtual în care cercetătorii pot împărtăși resurse, date și rezultate ale proiectelor lor. Platforma permite organizarea pe proiecte de cercetare, unde membrii pot colabora eficient prin partajarea de fișiere, anunțuri și informații relevante. De asemenea, sistemul gestionează autentificarea utilizatorilor cu roluri diferite (ex. cercetător, manager de proiect, administrator), asigurând că fiecare are acces la funcționalitățile și informațiile adecvate rolului său. Aplicația este gândită să faciliteze comunicarea și schimbul de cunoștințe într-un mod centralizat și securizat, îmbunătățind productivitatea echipelor de cercetare în construcții.

1.3 Definiții, acronime și abrevieri

1. **Aplicație web**: Aplicație accesibilă printr-un browser web, cu arhitectură client-server (frontend și backend).
2. **Frontend**: Partea de interfață a aplicației cu care interacționează utilizatorii (în acest proiect, pagini HTML cu Bootstrap).
3. **Backend**: Partea server a aplicației care conține logica de lucru și rulează pe server (în acest proiect, implementată în Python folosind Flask).
4. **Flask**: Un micro-framework web în limbajul Python, folosit pentru a crea rapid aplicații web (rute, managementul sesiunilor, templating etc.).
5. **SQLite**: Sistem de gestiune a bazelor de date relaționale, ușor și integrat, stocând datele într-un fișier local.
6. **Bootstrap**: Framework CSS frontend folosit pentru design responsive și componente UI predefinite.
7. **Autentificare**: Procesul de verificare a identității unui utilizator (login în aplicație cu nume utilizator și parolă).
8. **Autorizare**: Procesul de acordare a permisiunilor pe baza rolului utilizatorului (ex. doar anumiți utilizatori pot crea proiecte sau administra alți utilizatori).
9. **Administrator (Admin)**: Utilizator cu drepturi depline în aplicație, care poate gestiona alți utilizatori și configurații (ex. poate bloca un utilizator sau edita proiecte oricui).
10. **Manager de proiect**: Utilizator (de regulă, creatorul proiectului) cu drepturi de a administra un anumit proiect (ex. poate adăuga participanți, posta anunțuri în proiectul respectiv).
11. **Cercetător (utilizator obișnuit)**: Utilizator standard al platformei, cu acces la proiectele din care face parte, la resursele partajate și la funcțiile de colaborare permise.
12. **ERD**: „Entity-Relationship Diagram”, diagrama entități-relații ce modelează structura bazei de date (tabele și legături între ele).
13. **CRUD** :LCreate, Read, Update, Delete – operațiile de bază de gestionare a datelor (ex. crearea unui proiect, vizualizarea detaliilor, modificarea sau ștergerea lui).

1.4 Referințe

1. Documentația oficială Flask: Framework-ul utilizat pentru dezvoltarea backend-ului aplicației, oferind suport pentru rutare, sesiuni, template-uri etc.
2. Documentația oficială SQLite: Sistem de baze de date relațional folosit pentru stocarea persistentă a datelor aplicației (utilizatori, proiecte, fișiere etc.).
3. Documentația Bootstrap 5: Framework CSS folosit în frontend pentru design responsive și componente UI predefinite (formulare, butoane, navigație).
4. Repozitoriu GitHub al proiectului: Codul sursă al aplicației și fișierul requirements.txt ce listează toate pachetele Python utilizate (Flask, Flask-Login, Flask-WTF etc.).

2. Analiza cerințelor

2.1 Cerințe funcționale

Aplicația trebuie să îndeplinească următoarele cerințe funcționale esențiale, pentru a asigura colaborarea eficientă între utilizatori și gestionarea corectă a datelor:

1. **Autentificare și autorizare utilizatori**: Sistemul va permite utilizatorilor să își creeze cont și să se autentifice cu un nume de utilizator (sau email) și parolă. După logare, accesul la funcționalități va fi restricționat în funcție de rolul fiecărui utilizator. De exemplu, numai utilizatorii cu rol de administrator pot accesa panoul de administrare a utilizatorilor, iar doar managerii de proiect pot edita detaliile propriilor proiecte. Autentificarea gestionează sesiuni securizate astfel încât paginile interne să nu poată fi accesate fără login.
2. **Gestionarea utilizatorilor și rolurilor**: Aplicația va oferi facilități de administrare a utilizatorilor. Un utilizator administrator poate vizualiza lista utilizatorilor înregistrați, poate modifica rolurile acestora (de ex. promovarea unui utilizator la rol de admin sau blocarea unui utilizator necorespunzător) și poate suspenda accesul unor utilizatori (flag is_banned în baza de date, eventual cu o dată până la care este activ ban-ul). Fiecare cont de utilizator conține atribute precum nume, email, poză de profil (opțional), parolă (stocată securizat) și rol. Managementul conturilor include și posibilitatea de recuperare a parolei sau resetare (prin trimiterea unui email de resetare).
3. **Organizarea pe proiecte de cercetare**: Utilizatorii autentificați pot crea proiecte noi de cercetare, specificând detalii precum titlul proiectului, o descriere și (opțional) desemnând un manager sau echipă inițială. Fiecare proiect are un proprietar (creatorul inițial) și poate avea un manager de proiect (deseori aceeași persoană cu creatorul). Utilizatorii pot fi adăugați ca participanți la proiecte, fie de către managerul proiectului, fie de către un administrator. Doar membrii unui proiect pot vedea conținutul acelui proiect. Aplicația va oferi pagini dedicate: o pagină de listă a proiectelor (proprii sau publice, după caz) și pagină de detaliu proiect, unde se pot vedea informațiile proiectului, membrii participanți, fișierele încărcate și anunțurile postate.

4. **Încărcare și descărcare fișiere**: În cadrul fiecărui proiect, utilizatorii membri pot încărca fișiere relevante (documente, seturi de date, imagini etc.) care vor fi stocate pe server și referențiate în baza de date. Aplicația trebuie să valideze tipul fișierelor (de ex. să permită doar anumite extensii

relevante: PDF, DOCX, CSV, PNG, etc., pentru a preveni încărcarea de fișiere potențial periculoase executabile). Fiecare fișier încărcat este asociat cu proiectul respectiv și cu utilizatorul care l-a urcat (uploader). Membrii proiectului pot descărca fișierele încărcate, iar sistemul va înregistra metadate precum data încărcării, numele original al fișierului și dimensiunea. Această funcționalitate asigură un depozit centralizat de documente pentru fiecare proiect, facilitând accesul la ultimele versiuni ale documentelor de cercetare.

5. **Anunțuri și comunicări în proiect**: Pentru o mai bună colaborare, aplicația oferă un modul de anunțuri (Announcements) în interiorul fiecărui proiect. Participanții (sau doar managerul, în funcție de deciziile de design) pot posta mesaje tip anunț către toți membrii proiectului – de exemplu, actualizări privind progresul, notificări despre întâlniri sau rezultate noi. Fiecare anunț cuprinde conținutul text, data postării și autorul (utilizatorul care l-a publicat) și este afișat în pagina proiectului într-o listă cronologică. Această funcționalitate centralizează comunicarea importantă legată de proiect.
6. **Notificări prin email**: Sistemul este configurat să trimită notificări pe email către utilizatori pentru anumite evenimente. De exemplu, când un utilizator este adăugat într-un proiect nou, acesta poate primi un email de notificare cu detalii despre proiect. Similar, la postarea unui anunț nou într-un proiect, toți membrii proiectului pot primi o notificare prin email cu conținutul anunțului. De asemenea, funcționalități ca resetarea parolei folosesc emailul pentru a trimite link-ul de resetare. Implementarea notificărilor email folosește un server SMTP configurat în aplicație (de exemplu, Gmail SMTP prin extensia Flask-Mail) pentru a expedia automat mesaje către adresele de email înregistrate ale utilizatorilor. Notificările asigură că utilizatorii sunt la curent cu evenimentele importante fără a trebui să verifice manual în permanență platforma.

2.2 Cerințe non-funcționale

Pe lângă funcționalitățile de bază, aplicația trebuie să respecte și un set de cerințe non-funcționale care asigură calitatea sistemului în exploatare:

1. **Securitate**: Aplicația va proteja datele utilizatorilor și ale proiectelor. Parolele utilizatorilor sunt stocate în mod securizat (hash-uite cu un algoritm de criptare adecvat SHA-256 + salt). De asemenea, accesul la funcții este restricționat pe baza rolurilor (ex. verificări server-side pentru a nu permite acces la resurse de administrator de către un utilizator obișnuit). Comunicarea client-server se poate realiza peste HTTPS (TLS) dacă aplicația ar fi găzduită online, pentru a cripta datele tranzitate (în mediul de dezvoltare se folosește HTTP simplu, dar pentru producție aceasta ar fi o cerință obligatorie).

2. **Performanță** : Aplicația trebuie să răspundă într-un timp rezonabil la acțiunile utilizatorilor. Fiind o soluție bazată pe Flask și SQLite, și destinată unui număr relativ mic de utilizatori concurenți (în mediul academic sau grup restrâns de cercetători), performanța a fost considerată adecvată folosind aceste tehnologii. Operațiile uzuale (autentificare, navigare între pagini, încărcarea listei de proiecte) ar trebui să aibă timpi de răspuns sub ~1-2 secunde în condiții normale. Fișierele încărcate sunt servite eficient direct de pe disk prin Flask, iar dimensiunea acestora este limitată (se pot impune limite la upload, de ex. maxim câteva zeci de MB per fișier, pentru a nu afecta memoria). În eventualitatea extinderii numărului de utilizatori, se va avea în

vedere optimizarea interogărilor de baze de date și posibil migrarea la un sistem de baze de date mai robust (ex. PostgreSQL sau MySQL) pentru volum mare de date sau acces concurențial intens.

3. **Scalabilitate**: În forma actuală, aplicația este concepută pentru uz într-un mediu restrâns (facultate, institut de cercetare) cu câteva zeci de utilizatori activi și un număr moderat de proiecte. Nu s-au implementat soluții de scalare automată. Totuși, arhitectura poate fi extinsă: codul Flask poate rula pe un server mai puternic sau multiple instanțe (în modul production WSGI cu un server precum Gunicorn și un load balancer) dacă numărul de utilizatori crește. Pentru stocarea fișierelor, se poate utiliza un serviciu dedicat (ex. Amazon S3) dacă volumul devine foarte mare. Astfel, deși inițial proiectul nu e gândit pentru mii de utilizatori simultan, el poate fi adaptat ulterior cu efort moderat pentru a scala pe orizontală și verticală.
4. **Ușurința în utilizare (Usability)**: Interfața aplicației, realizată cu Bootstrap, este intuitivă și simplu de folosit. Navigarea este clară: utilizatorii se pot autentifica ușor, pot vedea proiectele într-o listă, pot accesa detaliile unui proiect și funcțiile disponibile (încărcare fișier, postare anunț) prin butoane vizibile. S-a urmărit respectarea unor principii de design UI/UX de bază, precum consecvența elementelor (aceleași stiluri pentru butoane de acțiune), mesaje de confirmare și eroare clare (ex. "Fișierul încărcat nu este permis" sau "Autentificare eșuată: date invalide"), și un design responsive care să funcționeze pe dispozitive diferite (Bootstrap oferă implicit compatibilitate mobilă).
5. **Portabilitate**: Aplicația poate fi rulată pe diferite platforme atâta timp cât există suport pentru Python. Fiind Python-based, ea poate funcționa pe Windows, Linux sau MacOS fără modificări (doar cu instalarea dependințelor specificate). Baza de date SQLite este un fișier portabil, ceea ce face ușoară mutarea datelor de pe un sistem pe altul (în scop de test sau deploy). Nu există dependențe legate de configurări speciale de sistem, astfel încât mediul de rulare poate fi adaptat ușor (inclusiv containerizare Docker, dacă se dorește).
6. **Mentenabilitate**: Codul este organizat într-un mod relativ modular (după cum se va detalia în secțiunea de implementare), facilitând înțelegerea și modificarea ulterioară. Folosirea unor framework-uri binecunoscute (Flask, SQLAlchemy, Flask-Login etc.) asigură că dezvoltatori familiarizați cu acestea pot interveni cu ușurință în cod. De asemenea, existența acestui document de analiză și proiectare sprijină mentenanța, oferind viitorilor dezvoltatori contextul și rațiunea deciziilor de design.
7. **Fiabilitate și disponibilitate**: Aplicația în modul de dezvoltare (development) rulează pe un singur server Flask și, nefiind încă distribuită public, nu are cerințe de uptime strict. În producție, pentru a asigura disponibilitatea 24/7, s-ar putea folosi un server web robust și mecanisme de repornire automată în caz de crash. Datorită simplității sale, aplicația are puține puncte de eșec: baza de date locală și sistemul de fișiere. Backup-uri periodice ale fișierului de baze de date și ale directorului de fișiere încărcate sunt recomandate pentru a evita pierderea datelor în caz de defecțiuni hardware.

2.3 Cazuri de utilizare (Use cases)

Pentru a ilustra modul de interacțiune al actorilor (utilizatori) cu sistemul, descriem pe scurt principalele **scenarii de utilizare** ale aplicației:

1. **Înregistrare utilizator nou**: Un vizitator al platformei își creează un cont furnizând un nume de utilizator, email și parolă. Sistemul validează datele (de ex. unicitatea numelui de utilizator și a email-ului) și stochează noul cont.

2. **Autentificare (Login)**: Un utilizator existent introduce credențialele (email/parolă) pe pagina de login. Sistemul verifică datele față de cele din baza de date (parola este verificată prin compararea hash-urilor) și, dacă sunt corecte și utilizatorul nu este blocat, inițiază o sesiune autenticată. Utilizatorul este apoi redirecționat către pagina principală (dashboard) sau lista de proiecte. Dacă datele sunt greșite, se afișează un mesaj de eroare și autentificarea este refuzată. (Actor: utilizator înregistrat)

3. **Creare proiect nou**: Un utilizator autentificat (de regulă un cercetător) inițiază crearea unui proiect de cercetare prin completarea unui formular cu detaliile proiectului (nume, descriere etc.). Sistemul salvează proiectul în baza de date, atribuind automat utilizatorului curent calitatea de creator și manager al proiectului. Proiectul nou apare în lista de proiecte ale utilizatorului. (Actor: utilizator autentificat, cercetător sau admin)

4. **Adăugare participant la proiect**: Managerul unui proiect sau un administrator alege să adauge alți utilizatori în echipa proiectului. Acesta selectează utilizatorul (dintre cei existenți în sistem) și sistemul adaugă o înregistrare în tabelul de participație (user-proiect). Ulterior, noul membru va putea accesa proiectul și va primi notificare (prin email) că a fost adăugat. (Actori: manager de proiect / admin și utilizator invitat)

5. **Vizualizare detalii proiect**: Un utilizator membru al unui proiect accesează pagina proiectului respectiv. Sistemul afișează detaliile (nume, descriere, data creării, numele managerului), lista de membri participanți, lista de anunțuri și lista de fișiere încărcate. Utilizatorul poate naviga în aceste secțiuni pentru a vedea conversațiile (anunțuri) sau pentru a descărca fișierele disponibile. (Actor: utilizator membru al proiectului)

6. **Postare anunț nou**: Un membru al proiectului (sau doar managerul, în funcție de politici) scrie un mesaj de anunț și îl publică în cadrul proiectului. Sistemul salvează anunțul în baza de date cu referință la autor și proiect, marcând data și ora. Toți ceilalți membri ai proiectului pot vedea imediat anunțul în aplicație; în plus, sistemul declanșează trimiterea unui email de notificare către membrii proiectului, conținând conținutul anunțului. (Actor: membru proiect)

7. **Încărcare fișier**: Un membru al proiectului selectează un fișier de pe calculatorul său pentru a-l încărca în proiect (de exemplu, un raport PDF). După apăsarea butonului de upload, fișierul este trimis către server; backend-ul validează tipul și dimensiunea fișierului, îl salvează în directorul de fișiere al aplicației și creează o intrare în baza de date (tabelul de fișiere) cu meta informații. Pagina proiectului se actualizează pentru a afișa noul fișier în listă. (Actor: membru proiect)

8. **Descărcare fișier**: Un utilizator dintr-un proiect dorește să acceseze un fișier încărcat anterior. Din lista de fișiere a proiectului, el dă clic pe fișierul dorit; serverul primește cererea și trimite fișierul ca răspuns (cu anteturile corespunzătoare de descărcare). Browser-ul utilizatorului inițiază

descărcarea fișierului. (Actor: membru proiect)

9. **Gestionare utilizatori de către admin**: Un administrator accesează o pagină specială de administrare (/admin/users) , unde vede toți utilizatorii existenți. Aici poate: schimba rolul unui utilizator (ex. acorda sau revoca drepturi de admin), bloca/debloca un utilizator (setând flag-ul de ban și durata, dacă e cazul), sau șterge conturi nepotrivite. Când o astfel de acțiune este inițiată, sistemul o execută (ex. actualizează rolul în baza de date) și confirmă operația. (Actor: administrator)

10. **Deconectare (Logout)**: Un utilizator autentificat solicită ieșirea din cont (logout). Sistemul va invalida sesiunea curentă și va reda utilizatorului o pagină de confirmare sau îl va redirecționa către pagina publică de login. După deconectare, utilizatorul nu mai poate accesa pagini interne fără să se autentifice din nou.

Fiecare dintre aceste cazuri de utilizare corespunde unuia sau mai multor cerințe funcționale de mai sus. Diagramele UML de caz de utilizare (dacă ar fi incluse) ar arăta actorii (Utilizator, Administrator) și relațiile lor cu aceste funcționalități. În lipsa diagramei grafice, descrierile textuale de mai sus acoperă interacțiunile cheie din sistem.

2.4 Constrângeri

În realizarea și utilizarea aplicației au fost identificate următoarele **constrângeri**:

1. **Tehnologice**: S-a ales Flask ca framework principal, ceea ce înseamnă că aplicația este rulată într-un singur thread (în modul de dezvoltare) și folosește SQLite ca bază de date. Aceste alegeri simplifică implementarea, dar vin cu limitări de exemplu, SQLite nu suportă acces concurent intens, deci doi utilizatori care încearcă să scrie simultan pot cauza latențe. De asemenea, Flask (fără utilizarea unui server extern) nu este recomandat pentru producție heavy-load. Aceste constrângeri au fost acceptabile deoarece aplicația este destinată unui mediu controlat, cu număr redus de utilizatori simultani.

2. **Lipsa CI/CD și teste automate**: Proiectul, fiind unul academic/prototip, nu a inclus setarea unui pipeline continuu de integrare și livrare (CI/CD) și nici scrierea de teste automate unitare sau de integrare. Acest lucru înseamnă că orice actualizare de cod necesită testare manuală și deploy manual. Pentru utilizare reală pe scară largă, aceasta este o constrângere ce ar trebui adresată ulterior, deoarece crește riscul de introducere de bug-uri la modificări și face deploy-urile mai lente.

3. **Mediul de rulare**: Aplicația nu a fost implementată (deployată) pe un server public sau în cloud, ci rulează local. Astfel, distribuirea aplicației către utilizatori externi necesită setări suplimentare (instalarea mediului Python, a dependențelor, rularea manuală a serverului). Acesta este un impediment practic dacă s-ar dori ca un grup mai larg de cercetători să folosească platforma, deci ar fi necesar un efort de configurare a unui mediu server (fizic sau cloud) pentru a găzdui aplicația.

4. **Trimiterea de email**: Funcționalitatea de notificare pe email depinde de configurarea corectă a unui serviciu de email SMTP. În medii locale de test s-a folosit, spre exemplu, un cont Gmail cu setări de aplicație mai puțin securizată sau un server SMTP de test. În producție, ar trebui folosit un serviciu de mail robust, iar constrângerea aici este legată de disponibilitatea și configurarea acestor servicii (posibile probleme de livrare a email-urilor, necesitatea protejării credențialelor de email în configurațiile aplicației etc.).

5. **Spațiu de stocare**: Fișierele încărcate sunt stocate pe același server unde rulează aplicația, ceea ce, pe termen lung, poate reprezenta o constrângere de spațiu de stocare. Dacă utilizatorii încarcă fișiere mari sau numeroase, serverul ar putea rămâne fără spațiu dacă nu se fac curățări periodice sau dacă nu se montează un volum suplimentar. În versiunea actuală nu există un mecanism automat de limitare strictă a spațiului folosit sau de ștergere a fișierelor vechi, deci admin-ul ar trebui să monitorizeze manual acest aspect.

6. **Timp și resurse de dezvoltare**: Dat fiind că proiectul a fost realizat într-un interval restrâns (de ordinul câtorva săptămâni) și de o echipă mică (sau un singur dezvoltator), anumite funcționalități avansate nu au putut fi incluse (ex: versiuni ale fișierelor, comentarii la anunțuri, editor colaborativ de documente etc.). Aceasta a fost o constrângere practică ce a dus la prioritizarea cerințelor esențiale prezentate mai sus, restul rămânând pentru dezvoltări ulterioare.

2.4 Constrângeri

În realizarea și utilizarea aplicației au fost identificate următoarele **constrângeri**:

1. **Tehnologice**: S-a ales Flask ca framework principal, ceea ce înseamnă că aplicația este rulată într-un singur thread (în modul de dezvoltare) și folosește SQLite ca bază de date. Aceste alegeri simplifică implementarea, dar vin cu limitări – de exemplu, SQLite nu suportă acces concurent intens, deci doi utilizatori care încearcă să scrie simultan pot cauza latențe. De asemenea, Flask (fără utilizarea unui server extern) nu este recomandat pentru producție heavy-load. Aceste constrângeri au fost acceptabile deoarece aplicația este destinată unui mediu controlat, cu număr redus de utilizatori simultani.

2. **Lipsa CI/CD și teste automate**: Proiectul, fiind unul academic/prototip, nu a inclus setarea unui pipeline continuu de integrare și livrare (CI/CD) și nici scrierea de teste automate unitare sau de integrare. Acest lucru înseamnă că orice actualizare de cod necesită testare manuală și deploy manual. Pentru utilizare reală pe scară largă, aceasta este o constrângere ce ar trebui adresată ulterior, deoarece crește riscul de aparitii de bug-uri la modificări și face deploy-urile mai lente.

3. **Mediul de rulare**: Aplicația nu a fost implementată (deployată) pe un server public sau în cloud, ci rulează local. Astfel, distribuirea aplicației către utilizatori externi necesită setări suplimentare (instalarea mediului Python, a dependențelor, rularea manuală a serverului). Acesta este un impediment practic dacă s-ar dori ca un grup mai larg de cercetători să folosească platforma, deci ar fi necesar un efort de configurare a unui mediu server (fizic sau cloud) pentru a găzdui

aplicația.

4. **Trimiterea de email**: Funcționalitatea de notificare pe email depinde de configurarea corectă a unui serviciu de email SMTP. În medii locale de test s-a folosit, spre exemplu, un cont Gmail cu setări de aplicație mai puțin securizată. În producție, ar trebui folosit un serviciu de mail robust, iar constrângerea aici este legată de disponibilitatea și configurarea acestor servicii (posibile probleme de livrare a email-urilor, necesitatea protejării credențialelor de email în configurațiile aplicației etc.).

5. **Spațiu de stocare**: Fișierele încărcate sunt stocate pe același server unde rulează aplicația, ceea ce, pe termen lung, poate reprezenta o constrângere de spațiu de stocare. Dacă utilizatorii încarcă fișiere mari sau numeroase, serverul ar putea rămâne fără spațiu dacă nu se fac curățări periodice sau dacă nu se montează un volum suplimentar. În versiunea actuală nu există un mecanism automat de limitare strictă a spațiului folosit sau de ștergere a fișierelor vechi, deci admin-ul ar trebui să monitorizeze manual acest aspect.

6. **Timp și resurse de dezvoltare**: Dat fiind că proiectul a fost realizat într-un interval restrâns. Aceasta a fost o constrângere practică ce a dus la prioritizarea cerințelor esențiale prezentate mai sus, restul rămânând pentru dezvoltări ulterioare.

2.5 Riscuri

Pe parcursul analizei, au fost identificate câteva riscuri asociate proiectului și utilizării sale:

1. **Risc de securitate (atacuri cibernetice)**: Orice aplicație web se poate confrunta cu tentative de atac. În cazul de față, riscurile includ atacuri de tip SQL Injection (deși folosind SQLAlchemy sau ORM aceste riscuri scad, dar dacă ar exista concatenări naive de interogări SQL, ar fi o problemă), atacuri XSS prin câmpurile de input (dacă nu se sanitizează datele afișate în browser, un utilizator malițios ar putea posta un script într-un anunț, spre exemplu). De asemenea, conturile utilizatorilor pot fi ținta atacurilor de tip brute-force pentru ghicirea parolei. S-au atenuat unele riscuri folosind practici standard (framework-ul Flask WTF oferă protecție CSRF pentru formulare, parolele sunt criptate, etc.), însă lipsa unui audit de securitate amănunțit rămâne un risc.

2. **Risc de pierdere a datelor**: Folosind SQLite ca stocare, un fișier corupt sau șters accidental ar duce la pierderea întregii baze de date. De asemenea, fișierele încărcate pot fi pierdute dacă discul se strică sau datele nu sunt salvate. Fără un mecanism de backup implementat automat, există riscul ca o eroare de sistem să ducă la pierderea datelor de cercetare încărcate. Riscul se poate reduce prin backup manual periodic al fișierului de DB și al directorului de fișiere.

3. **Risc de eroare neprevăzută (bug-uri)**: Dat fiind că nu s-au scris teste automate, există întotdeauna posibilitatea ca anumite cazuri particulare să genereze erori în aplicație. De exemplu, un fișier cu un nume neobișnuit ar putea cauza probleme la descărcare, sau un caracter special într-un nume de proiect ar putea produce erori de afișare. Logger-ul de erori va surprinde aceste

situații, dar experiența utilizatorilor ar putea avea de suferit până la remedierea bug-urilor. Ca măsură proactivă, aplicația a fost testată manual pe cât mai multe scenarii uzuale, însă riscul unor bug-uri nedetectate rămâne.

3. Arhitectura și proiectarea sistemului

3.1 Arhitectura generală

Arhitectura aplicației este una de tip **web tradițional client-server**, în care:

1. **Clientul** este reprezentat de browser-ul web al utilizatorului, care afișează interfața utilizator (paginile HTML, stilizate cu CSS și Bootstrap, și cu puțin JavaScript pentru comportamente interactive minore). Interacțiunea se face prin cereri HTTP către server (de exemplu, când utilizatorul apasă un buton sau trimite un formular).

2. **Serverul (Backend-ul)** este aplicația Flask rulată pe un mediu Python, care primește cererile HTTP ale clienților, le procesează (în funcție de logică și de datele din baza de date) și returnează răspunsuri (de obicei pagini HTML generate sau redirecționări). Flask, fiind un micro-framework, a permis organizarea codului în rute (endpoints) care corespund diferitelor URL-uri ale aplicației (ex: /login, /projects, /project/<id> etc.), fiecare rută fiind asociată cu o funcție Python ce prelucrează cererea.

3. **Baza de date** este sistemul de stocare persistentă s-a folosit SQLite, ceea ce înseamnă că toate datele aplicației (utilizatori, proiecte, fișiere, anunțuri, relații de participare) sunt stocate într-un fișier local. Aplicația accesează baza de date prin intermediul unui layer ORM folosit (SQLAlchemy). Arhitectural, baza de date este integrată în același proces/server ca aplicația (nu este un server separat), ceea ce simplifică accesul.

Pe lângă aceste componente principale, arhitectura include și:

4. **Sistemul de fișiere** al serverului, folosit pentru a stoca fișierele încărcate de utilizatori. Există un director dedicat (ex. uploads/ sau files/) unde, la încărcare, fișierele sunt salvate cu un nume unic (de obicei prin atașarea unui hash sau ID la numele original, pentru a evita coliziuni). Baza de date reține calea sau numele fișierului, permițând serverului să știe de unde să-l ia la o cerere de descărcare.

5. **Componenta de email** nu este un serviciu separat, ci mai degrabă o configurare în cadrul aplicației backend. Flask, cu ajutorul modulului Flask-Mail, se conectează la un SMTP extern (cum ar fi serverul Gmail) pentru a trimite mesajele generate. Astfel, de fiecare dată când se trimite un email, serverul Flask inițiază o conexiune SMTP către serviciul de email configurat.

Rezultă o arhitectură monolitică, în sensul că aplicația Flask conține toată logica (autentificare, procesare cereri, acces BD, trimitere email, logică de afișare) într-un singur serviciu. Aceasta este adecvată pentru scopul și dimensiunea proiectului. În scenariul unei viitoare extinderi, arhitectura ar putea evolua (de

exemplu, separând un serviciu dedicat pentru trimitere de email-uri sau trecând la o arhitectură cu microservicii). Figura de mai jos ilustrează schematic principalele componente și interacțiunile lor.

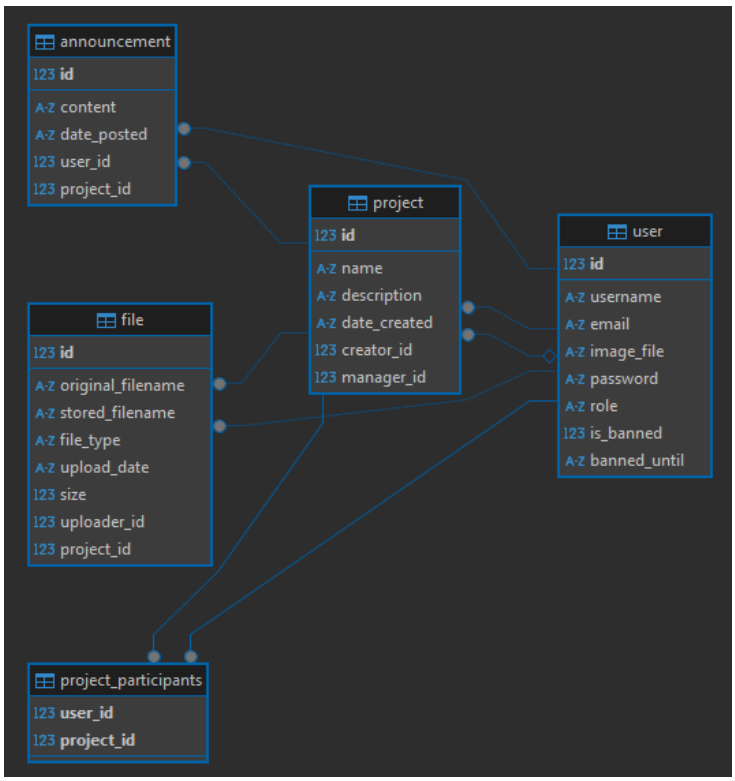


Figura 1: Modelul entităților și relațiilor (ERD) din baza de date a aplicației. Sunt reprezentate tabelele principale – Utilizator (User), Proiect (Project), Fișier (File), Anunț (Announcement) și ParticipareProiect (Project_Participants) – împreună cu atributele lor și relațiile dintre acestea.

În arhitectura logică, putem considera stratul de prezentare (interfața web în browser), stratul de logică de aplicație (codul Flask/Python care implementează regulile de business, ex. cine are voie să facă o acțiune, cum se procesează un fișier uploadat) și stratul de date (baza de date și sistemul de fișiere). Comunicarea între straturi se face prin metode standard: HTML Forms / HTTP Request către logică, interogări SQL către baza de date, citire/scriere fișiere pe disc, etc.

3.2 Modelul de date

Modelul de date al aplicației este unul relațional, fiind definit de baza de date SQLite. Așa cum se vede în diagrama ER de mai sus, există 5 entități principale în sistem, cu relațiile corespunzătoare între ele:

1. **User (Utilizator)** – reprezintă utilizatorii platformei. Atribute importante:

    1.1. id – Cheie primară unică pentru fiecare utilizator (identificator numeric).

    1.2. username – Nume de utilizator (unic în sistem, folosit la autentificare).

1.3. email – Adresa de email a utilizatorului (unic în sistem, folosită pentru notificări și, eventual, pentru autentificare/recuperare parolă).

1.4. password – Parola utilizatorului stocată sub formă de hash (pentru securitate, se criptează parola la creare; la autentificare se compară hash-ul parolei introduse cu cel din baza de date).

1.5. image_file – Numele fișierului imaginii de profil a utilizatorului (dacă există). Implicit ar putea fi un avatar generic (ex. default.jpg). Imaginile sunt stocate în directorul de profiluri.

1.6. role – Rolul utilizatorului în sistem (de exemplu, poate fi un text precum "admin" sau "user", sau un cod numeric care indică privilegiile). Acest câmp determină ce acțiuni are voie utilizatorul (ex. admin poate șterge proiecte, user obișnuit nu).

1.7. is_banned – Indicator (boolean) dacă utilizatorul este blocat (banat) în sistem. Dacă True, atunci login-ul este refuzat.

1.8. banned_until – dată până la care utilizatorul este blocat, dacă ban-ul este temporar. După această dată, contul devine automat activ din nou. (Dacă nu s-a implementat detaliat, poate rămâne nefolosit sau null mereu în versiunea curentă.)

2. **Project (Proiect)** – reprezintă un proiect de cercetare colaborativ. Atribute:

2.1. id – Cheia primară a proiectului (identificator numeric).

2.2. name – Numele proiectului (titlu).

2.3. description – O descriere text a proiectului, care poate conține detalii despre scopul și obiectivele proiectului.

2.4. date_created – Data creării proiectului (setată automat la inițializare). Ajută la ordonarea proiectelor cronologic sau la afișarea stării.

2.5. creator_id – Referință (cheie externă) către tabela User, indicând utilizatorul care a creat proiectul. Acest utilizator are implicit drepturi de manager asupra proiectului.

2.6. manager_id – Referință către User, indicând managerul proiectului (poate fi același cu creatorul sau alt utilizator dacă se transferă managementul). Managerul este responsabil de administrarea participanților și conținutului proiectului.

Relații: Fiecare proiect aparține unui creator (User) și are un manager (User). Relația cu User este de tip one-to-many (un utilizator poate crea/gestiona mai multe proiecte, dar un proiect are un

singur creator/manager). Între Project și User există și relația many-to-many prin entitatea Project_Participants (toți membrii proiectului).

3. **File (Fișier)** – reprezintă un fișier încărcat în cadrul unui proiect. Atribute:

    3.1. id – Cheie primară unică a fișierului.

    3.2. original_filename – Numele original al fișierului încărcat (așa cum era pe calculatorul utilizatorului). Acest nume este afișat în listă pentru ca utilizatorii să recunoască fișierul.

    3.3. stored_filename – Numele sub care fișierul este salvat pe server. De obicei, pentru a evita conflictele și a spori securitatea, fișierul nu e salvat exact cu numele original; aplicația generează un nume unic (ex. un hash sau un UUID plus extensia) și îl stochează pe disc. Acest câmp ține evidența acelui nume intern, astfel încât la descărcare serverul știe ce fișier de pe disc să trimită.

    3.4. file_type – Tipul de fișier sau extensia (ex. "pdf", "png", "docx" etc.). Acesta poate fi folosit pentru a afișa o iconiță relevantă sau pentru a verifica tipul la descărcare.

    3.5. upload_date – Data și ora la care fișierul a fost încărcat.

    3.6. size – Dimensiunea fișierului (în bytes). Poate fi utilă pentru a afișa informații utilizatorilor sau a impune limite.

    3.7. uploader_id – Cheie externă către User, indicând cine a încărcat fișierul. Astfel se știe autorul fiecărui fișier.

    3.8. project_id – Cheie externă către Project, indicând în ce proiect a fost încărcat fișierul. Practic, leagă fișierul de containerul său (proiectul).

Relații: Fiecare fișier aparține unui proiect (Project) și are un uploader (User). Deci există o relație many-to-one cu Project (mulți fișiere într-un proiect) și many-to-one cu User (un utilizator poate încărca mai multe fișiere). Dacă se șterge un proiect, ideal ar trebui șterse și fișierele asociate (ștergere în cascada).

4. **Announcement (Anunț)** – reprezintă un mesaj/anunț postat în cadrul unui proiect. Atribute:

    4.1. id – Cheie primară a anunțului.

    4.2. content – Conținutul text al mesajului. Probabil există o limită sau format (text simplu).

    4.3. date_posted – Data/ora când a fost publicat anunțul.

    4.4. user_id – Cheie externă către User, indicând cine a scris postarea (autorul anunțului).

4.5. project_id – Cheie externă către Project, indicând în ce proiect a fost postat anunțul.

Relații: Fiecare anunț aparține unui proiect și are un autor (user). Relațiile sunt many-to-one către Project și către User (un proiect poate avea multiple anunțuri; un utilizator poate posta mai multe anunțuri, posibil în proiecte diferite). În practică, la afișare, se combină aceste date: de exemplu, "Ion Popescu a postat la 2025-06-01: Lorem ipsum... " în cadrul proiectului X.

5. **Project_Participants (ParticipareProiect)** – reprezintă relația mulți-la-mulți între User și Project, adică tabelul care indică ce utilizatori fac parte din ce proiecte. Atribute:

   5.1. user_id – Cheie externă către User. (împreună cu project_id formează cheia compusă primară a tabelului, dacă s-a ales așa, sau există un id propriu al înregistrării de participare)

   5.2. project_id – Cheie externă către Project.

   Relații: Acest tabel nu are o entitate "independentă" de sine stătătoare, ci este pur și simplu o legătură. Fiecare înregistrare spune că user-ul X este membru în proiectul Y. În baza acestei relații, aplicația știe ce proiecte să afișeze fiecărui utilizator (doar cele unde există o înregistrare în Project_Participants cu user_id al lui). De asemenea, când un proiect este șters sau un utilizator este eliminat, înregistrările corespunzătoare din acest tabel trebuie șterse.

Pentru gestionarea bazei de date s-a utilizat fie un ORM (Object-Relational Mapper) (cum ar fi Flask-SQLAlchemy) definind modele Python pentru fiecare entitate de mai sus, fie interogări SQL directe executate prin modulul sqlite3. Folosirea unui ORM ar simplifica interacțiunea (de exemplu, definirea unei clase User cu atributele de mai sus și relații, apoi folosirea metodelor ORM pentru a crea sau interoga utilizatori). Având în vedere conținutul fișierului requirements.txt al proiectului, este foarte probabil că a fost folosit Flask-SQLAlchemy pentru definirea acestor modele și Flask-Migrate pentru gestionarea schemelor, însă documentația de față se concentrează pe modelul logic, nu pe implementarea specifică a accesului la date.

3.3 Tehnologii și stack de dezvoltare

Proiectul a fost dezvoltat folosind un stack tehnologic modern, concentrat pe ușurința de dezvoltare și simplitate, potrivit pentru un prototip academic. Lista tehnologiilor și bibliotecilor folosite include:

1. **Python 3.13** – Limbajul de programare principal folosit pentru logica serverului. Python a fost ales datorită sintaxei sale clare și existenței framework-ului Flask, care simplifică mult crearea rapidă de prototipuri web.

2. **Flask** – Micro-framework web pentru Python, responsabil de rutarea cererilor web și generarea răspunsurilor. Flask furnizează și mecanisme utile precum sistemul de template-uri Jinja2 (pentru a crea pagini HTML dinamic, pe baza datelor), gestionarea sesiunilor de autentificare și extensibilitate prin plugin-uri.

3. **Flask-Login** – Extensie Flask utilizată pentru a simplifica managementul autentificării și sesiunii utilizatorilor. Aceasta oferă funcții pentru logarea utilizatorilor, protejarea paginilor care necesită login (@login_required), gestionarea "remember me" etc. În proiect, Flask-Login este folosit pentru a reține ID-ul utilizatorului autentificat în sesiune și a redirecționa automat la pagina de login dacă cineva neautentificat încearcă să acceseze resurse protejate.

4. **Flask-WTF** – Extensie Flask pentru formulare web, bazată pe biblioteca WTForms. Asigură generarea ușoară a formularelor HTML direct din cod Python și, foarte important, include protecție CSRF (Cross-Site Request Forgery) automată. În această aplicație, Flask-WTF a fost folosit pentru formularele de autentificare/înregistrare, de creare proiect, de încărcare fișier etc., permițând și validări rapide (ex. validarea că un câmp nu e gol, că un email are format corect etc.).

5. **Flask-SQLAlchemy** – ORM (Object Relational Mapper) pentru bazele de date relaționale, integrat cu Flask. Acesta a fost folosit pentru a defini modelele de date (User, Project, File, Announcement, ProjectParticipants) ca clase Python și pentru a facilita operațiile cu baza de date SQLite fără a scrie manual interogări SQL pentru cele mai comune operații. De exemplu, pentru a căuta un utilizator după email: User.query.filter_by(email=form.email.data).first() în loc de a scrie SELECT. Totodată, s-a folosit SQLAlchemy pentru a inițializa baza de date și a crea automat tabelele pe baza modelelor (migrare inițială).

6. **SQLite** – Baza de date relațională folosită. Este integrată prin SQLAlchemy (care sub capotă folosește driver-ul sqlite3). S-a optat pentru SQLite deoarece nu necesită instalare sau configurare de server separat, fiind ideală pentru dezvoltare locală și pentru aplicații cu încărcare mică. Fișierul de DB este probabil denumit site.db sau app.db și este creat automat la prima rulare dacă nu există, conținând tabelele definite.

7. **Flask-Mail** – (posibil utilizat) Extensie pentru trimiterea de email-uri din Flask. Dacă notificările pe email au fost implementate, Flask-Mail a permis configurarea ușoară a serverului SMTP (prin setări ca MAIL_SERVER, MAIL_PORT, MAIL_USERNAME, MAIL_PASSWORD în configurația aplicației) și expedierea email-urilor printr-un obiect Mail.

8. **Werkzeug & Jinja2** – Biblioteci care vin odată cu Flask. Werkzeug este motorul WSGI ce rulează serverul web intern și oferă utilitare (de ex. pentru gestionarea fișierelor încărcate, securizarea cookie-urilor, hashing de parolă dacă s-a folosit werkzeug.security etc.). Jinja2 este motorul de template utilizat pentru a construi interfața (paginile HTML cu zone dinamice, precum liste de proiecte generate dinamic).

9. **HTML5, CSS3 și Bootstrap 5** – Tehnologiile frontend. Paginile sunt scrise în HTML și stilizate cu CSS; s-a folosit **Bootstrap 5** (printr-un link CDN în template-uri) pentru componente gata-făcute și design responsive. Bootstrap a oferit un layout uniform (navbar, formular de login stilizat, tabele pentru liste de date, butoane, alerte pentru mesaje de succes/eroare etc.). Prin urmare, nu a fost nevoie de un designer dedicat sau de CSS personalizat extensiv, întrucât Bootstrap acoperă nevoile de bază. S-au adăugat totuși probabil câteva stiluri proprii minore

pentru a ajusta aspectul (prin fișier CSS custom sau direct în pagini).

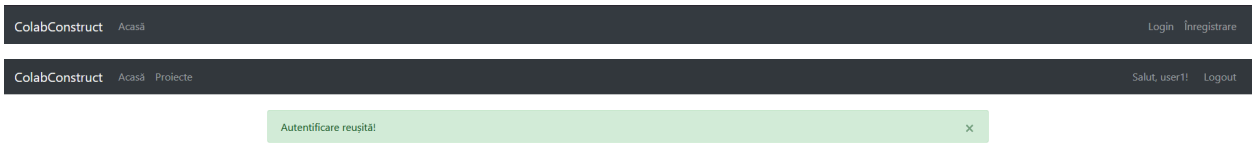
10. **JavaScript** – În mod minimal, s-a folosit JavaScript integrat în Bootstrap (ex. pentru componenta de navbar toggle pe mobil, sau pentru eventuale modale de confirmare). Aplicația nefiind una single-page și neavând cerințe de dinamicitate complexă, nu a folosit un framework JS modern (React/Angular etc.), ci doar scripturile clasice necesare Bootstrap-ului și poate câteva linii pentru funcții mici (ex. confirmare de ștergere proiect într-o fereastră modală).

Fișierul requirements.txt al proiectului conține toate pachetele Python utilizate, cu versiuni exacte. Printre acestea, conform specificațiilor proiectului, se regăsesc cele menționate mai sus: Flask, Flask-Login, Flask-WTF, Flask-Mail, Flask-Bootstrap (dacă s-a folosit o integrare specială cu Bootstrap, deși nu era obligatoriu), Flask-SQLAlchemy, email-validator (de obicei folosit cu WTForms pentru a valida formatul email-urilor), itsdangerous (pentru tokenuri de resetare parolă, vine ca dependență la Flask-Mail), etc. Toate aceste tehnologii lucrează împreună pentru a oferi o aplicație coerentă: Python/Flask pentru logică server, SQLite pentru stocare, HTML/Bootstrap pentru interfață.

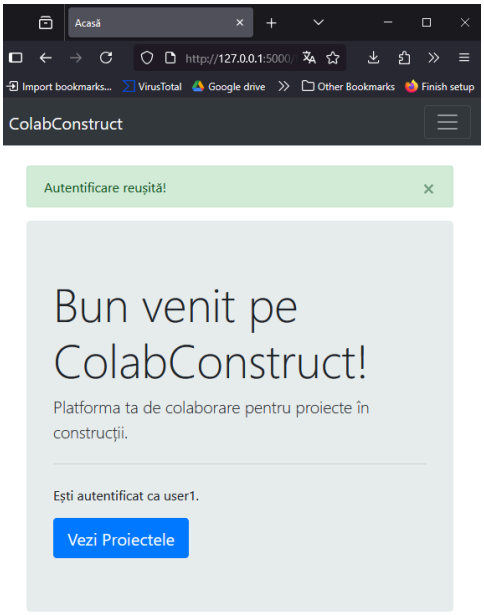
3.5 Interfața utilizatorului (UI/UX)

Interfața aplicației a fost proiectată cu simplitate în minte, dată fiind natura utilizatorilor (cercetători care doresc un mediu de colaborare ușor de folosit, nu un sistem complicat). Câteva aspecte cheie ale UI/UX:

1. **Design curat și familiar**: Prin utilizarea Bootstrap, s-a adoptat un stil vizual modern și consistent, aliniat cu așteptările utilizatorilor de aplicații web. Culorile și fonturile implicite Bootstrap asigură contrast și lizibilitate, iar componentele (butoane, carduri, bare de navigație) sunt ușor de recunoscut. De exemplu, bara de navigație de sus conține link-uri esențiale (către pagina de proiecte, către pagina de administrare, și un buton de Logout). Formularul de autentificare este centrat pe ecran, cu etichete clare la câmpuri și mesaje de eroare evidențiate cu roșu în caz de nevoie.

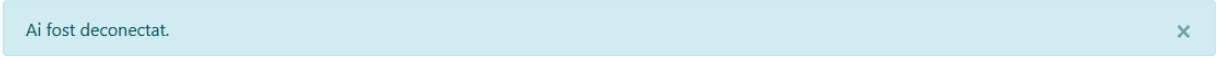


2. **Layout responsiv**: Fiind construit cu Bootstrap 5, layout-ul paginilor se adaptează la diferite rezoluții. Pe ecrane late (desktop), conținutul paginilor (listele de proiecte, formularele) este afișat în container central cu o lățime medie, pentru a fi ușor de citit. Pe dispozitive mobile, elementele se stivuiesc vertical – de exemplu, bara de navigație se transformă într-un buton "hamburger" pentru a economisi spațiu, iar listele sau tabelele pot defila orizontal dacă e nevoie. Astfel, cercetătorii pot accesa platforma și de pe laptop în laborator, și de pe telefon în șantier, având o experiență consecventă.



3. **Navigare și structurare**: După logare, utilizatorul vede o pagină principală / dashboard simplu, care, în implementarea curentă, este de fapt lista proiectelor la care are acces. Fiecare proiect din listă este afișat poate ca un card cu titlul și o mică descriere, sau ca rând într-un tabel cu posibilitate de clic. De acolo, utilizatorul poate da clic pe un proiect pentru a vedea detaliile. În pagina de detalii proiect, informația este structurată pe secțiuni: detalii de bază sus (nume, descriere, manager, eventual un buton "Editează" dacă utilizatorul curent e managerul), apoi anunțurile listate (în ordine cronologică inversă, ultimele primele) și fișierele listate (cu nume, uploader, dată). Fiecare listă are interfață pentru acțiuni: lângă "Fișiere" există un buton "Încarcă fișier" care deschide formularul de upload; lângă "Anunțuri" un buton "Adaugă anunț" etc. Dacă utilizatorul nu are dreptul să facă o acțiune, butonul respectiv fie nu apare, fie este inactiv cu un tooltip explicativ.

4. **Feedback și mesaje**: O parte importantă a UX este ca utilizatorul să știe ce se întâmplă în urma acțiunilor sale. Aplicația folosește mesaje flash (Flash messages din Flask) pentru a oferi feedback. De exemplu, după ce un proiect este creat cu succes, utilizatorul este redirecționat la pagina listei și vede un mesaj de tip alertă verde "Proiectul a fost creat cu succes." Dacă intervine o eroare (ex. eșuează încărcarea fișierului pentru că extensia nu e permisă), utilizatorului i se afișează un mesaj de eroare (alertă roșie) specificând problema. Aceste mesaje cresc încrederea utilizatorului în sistem, știind că acțiunea lui a avut (sau nu) efectul dorit.



5. **Elemente de convenienţă**: S-au adăugat mici detalii pentru ergonomie. De exemplu, în formularele de autentificare şi înregistrare, primul câmp este autofocus (cursorul apare direct în el). Butonul de submit este dezactivat până când câmpurile obligatorii sunt completate (unele validări se fac şi pe client, prin atribute HTML5, de exemplu required asigură că nu se poate trimite formularul gol). La listarea fişierelor, dacă lista devine lungă, se poate implementa paginaţie sau scroll intern; la anunţuri la fel. În pagina de administrare a utilizatorilor (pentru admin), userii pot fi listaţi într-un tabel şi poate există o funcţie de căutare după nume/email pentru a găsi rapid pe cineva. Aceste detalii, chiar dacă sunt opţionale, contribuie la o experienţă mai plăcută.

În ansamblu, interfaţa utilizator este simplă dar funcţională, concentrându-se pe funcţionalitate mai mult decât pe elemente grafice elaborate. Acest lucru a permis dezvoltarea rapidă şi facilitează adoptarea de către utilizatori, întrucât aplicaţia face câteva lucruri de bază şi le face într-un mod relativ intuitiv.

4. Implementare

4.1 Structura codului

Codul aplicaţiei este organizat într-un mod care reflectă simplitatea proiectului, menţinând totuşi o separare clară a preocupărilor. Fiind un proiect de dimensiuni mici, s-a optat pentru o structură monolitică minimală, fără a se fragmenta în prea multe module, pentru a uşura dezvoltarea. Structura principală a codului (fişiere şi directoare) este următoarea:

1. **app.py** – Punctul de intrare al aplicaţiei şi principalul fişier Python. Aici se configurează aplicaţia Flask (app = Flask(__name__)), se setează configuraţiile (secret key pentru sesiuni şi WTF CSRF, configuraţii de mail SMTP, calea către baza de date SQLite etc.), se initializează extensiile şi se definesc toate rutele (endpoints) împreună cu logica aferentă. De asemenea, tot în app.py sunt definite modelele de date (dacă nu s-au pus într-un modul separat) clasele User, Project, File, Announcement, Participant, fiecare moştenind de la db.Model (SQLAlchemy) cu coloanele definite ca atribute.

2. **templates/** – Directorul ce conţine toate şabloanele HTML (fişiere .html) folosite de aplicaţie. Flask, prin Jinja2, va reda aceste template-uri la cerere. Printre fişierele aşteptate aici:

    2.1. base.html – Template-ul de bază care conţine scheletul (header, navbar, eventual footer) şi în care se inserează paginile specifice. Utilizează sintaxa Jinja2 cu {% block content

%} {% endblock %} pentru a delimita secțiunea de conținut variabil.

2.2. login.html și register.html – Pagini pentru autentificare și înregistrare. Acestea conțin formulare WTForms integrate, cu câmpuri pentru username/email, password și buton de submit.

2.3. projects_list.html – Pagina ce listează proiectele. Parcurge o listă de proiecte (pasată din backend) și le afișează, fiecare cu link către detalii. Dacă nu există proiecte, poate afișa un mesaj. Are și un buton "Adaugă Proiect" dacă utilizatorul dorește să creeze altul.

2.4. view_project.html – Pagina de detalii pentru un proiect anume. Afișează informațiile proiectului și conține secțiuni/partialuri incluse:

2.4.1. O secțiune cu detalii proiect și butoane (ex. dacă user-ul este manager: "Editează detalii", "Șterge proiect").

2.4.2. O secțiune cu lista de anunțuri: iterează prin project.announcements și afișează autor, data, conținut. Sub listă, un formular pentru a adăuga anunț nou (dacă permis).

2.4.3. O secțiune cu lista de fișiere: iterează prin /uploads și afișează nume, uploader, dată, mărime și un link/btn de descărcare (care lovește ruta de download). Sub listă, formular de upload fișier (dacă permis).

2.4.4. O secțiune cu lista de participanți la proiect: enumeră userii participanți. Dacă user-ul curent e manager, poate exista un formular de adăugare utilizator (introdu email sau select dropdown) și butoane de eliminare lângă fiecare participant.

Toate aceste template-uri moștenesc base.html, inserând conținut specific. Ele folosesc și variabile/flashed messages trimise de backend.

3. **uploads/** – Un director care nu e neapărat în static (pentru că accesul la fișiere se face prin rută protejată ca să verifice permisiuni), unde se stochează fișierele urcate de utilizatori. În mediul de dezvoltare, este accesibil local. În producție, ar putea fi pe un storage separat. Dar la implementare locală, de exemplu, fișierul cu nume ae12f3d4_report.pdf stocat aici e servit de o rută Flask /download/<file_id> care face send_from_directory din acest folder.

Structura codului este așadar destul de lineară și ușor de urmărit: se încarcă configurările, se definesc modelele și formularele, apoi se definesc rutele cu logica fiecăreia. Lipsa unei arhitecturi complicate (nu s-au folosit blueprint-uri multiple sau servicii externe) face ca aplicația să fie ușor de citit.

4.2 Module și componente principale

Deși codul rezidă în mare parte într-un singur modul (app.py), conceptual putem împărți aplicația în mai multe componente logice, fiecare acoperind un set de funcționalități:

1. **Componenta de autentificare & autorizare**: Include toate funcțiile și clasele legate de contul de utilizator. Aici intră:

    1.1. Formularele de Login și Register (validează input-ul, parolele etc.).

    1.2. Funcțiile de rută /login, /register, /logout.

    1.3. Utilizarea Flask-Login (login_user, login_required, current_user) și definirea funcției load_user(user_id) pentru LoginManager (necesară pentru a reîncărca utilizatorul din sesiune).

    1.4. Hashing-ul parolelor (generate_password_hash0).

    1.5. Mecanismul de *remember me* (dacă s-a bifat, sesiunea e extinsă).

    1.6. Verificarea rolurilor la accesul anumitor pagini (de exemplu un decorator sau simplu if current_user.role != 'admin': abort(403)).

    1.7. (Opțional) generarea token-urilor de resetare parolă și trimiterea emailului de resetare (dacă implementat).

Această componentă se asigură că doar utilizatorii legitimi pot intra în sistem și că își pot gestiona contul.

2. **Componenta de gestionare a proiectelor**: Conține logica pentru creare, vizualizare, editare, ștergere proiecte:

    2.1. Formularul de creare/ editare proiect (capturând numele, descrierea etc.).

    2.2. Ruta /create_project (sau integrată în /projects page modal).

    2.3. Ruta /project/<id> (GET afișare detalii, POST folosit pentru a edita detaliile dacă un formular de edit e pe aceeași pagină).

    2.4. Verificarea permisiunilor: numai creatorul/managerul sau adminul pot edita sau șterge un proiect. Ceilalți membri pot doar vizualiza.

    2.5. La ștergere proiect, se șterg din BD și toate legăturile (Project_Participants, File, Announcement asociate) și ideal și fișierele de pe disc.

    2.6. Relația cu componenta de utilizatori: adăugarea/ eliminarea participanților (care poate fi văzută ca logică a proiectului sau a utilizatorilor).

3. **Componenta de partajare fișiere**: Cuprinde tot ce ține de upload/download:

    3.1. Formularul de încărcare fișier

    3.2. Logica de pe server care primește fișierul: se folosește obiectul FileStorage din Flask (automat în request.files), se verifică extensia (os.path.splitext(filename) și comparare cu o listă permisă), se generează un nume sigur (uuid4()), se salvează fișierul fizic cu metoda upload_file(project_id) într-un director de upload.

    3.3. Crearea instanței File în baza de date cu meta-datele. Dacă totul ok, confirmare, altfel rollback și mesaj de eroare.

    3.4. Servirea fișierelor: fie prin send_from_directory într-o rută dedicată.

    3.5. Limitarea mărimii fișierelor: Flask permite config MAX_CONTENT_LENGTH pentru a preveni fișiere prea mari. E posibil să se fi setat (in codul current este 100* 1024*1024 , 100mb)

    3.6. Această componentă interacționează cu logging (dacă upload-ul eșuează, se loghează motivul).

4. **Componenta de comunicare (anunțuri & notificări)**:

    4.1. Formularul de adăugare anunț (în pagina proiect).

    4.2. Logica server de creare anunț: preia textul, îl salvează în BD (Announcement).

    4.3. După salvare, pregătește un email către toți participanții proiectului (exceptând poate autorul) cu conținutul anunțului. Folosește Flask-Mail: construiește un Message(subject="Nou anunț în proiectul X", recipients=[emails...], body=...) și mail.send(msg). Această trimitere se poate face sincron (imediat la postare) sau se poate decala (dar probabil imediat, dat fiind contextul).

    4.4. Afișarea anunțurilor în interfață (componentă deja acoperită în template).

    4.5. De asemenea, notificări email pot fi trimise și pentru alte evenimente, ca adăugare într-un proiect: atunci când un user e adăugat în Project_Participants, după db.session.commit(), se poate trimite un email de invitație/ notificare. Codul este similar: se definește subiect, destinatar (doar user-ul nou adăugat) și un text: "Ați fost adăugat în proiectul X de către Y. Conectați-vă pentru a vedea detalii.".

Această componentă asigură partea colaborativă a platformei, ținând toți membrii informați.

5. **Componenta de administrare utilizatori**:

5.1. Rută /admin/users care folosește un decorator current_user.role != 'admin'.

5.2. Aceasta extrage toți userii din BD și îi arată într-o listă. Poate oferi acțiuni la fiecare: schimbare rol (un dropdown cu rolueile posibile), blocare/deblocare (toggle la is_banned cu input pentru banned_until).

5.3. Acțiunile pot fi implementate fie ca mici formulare separate per user (la submit se face acțiunea și se redă pagina).

5.4. Mesaje de succes/flashes apar la ex. "Rolul utilizatorului X a fost actualizat".

5.5. Tot aici adminul poate șterge conturi. La ștergere trebuie șterse cascade: dacă un user e scos, fișierele uploadate de el, anunțurile sale se șterg.

5.6. Implementarea acestei componente atinge multe părți: baza de date (ștergeri, modificări rol), fișiere (șterge avatarul user-ului?), securitate (nu vreți ca un admin să se șteargă pe sine accidental, eventual confirmare dublă), notificări (nu cred că e cazul la șters user, poate trimitere de email de înștiințare a userului blocat? posibilitate).

Legăturile între componente:
Aceste componente nu sunt izolate complet, ele interacționează strâns, dat fiind contextul monolitic. De exemplu, când un user e șters de admin (componenta de administrare utilizatori), se folosesc și metode din componenta de proiect (pentru a șterge referințe la proiecte). Când un fișier e încărcat (componenta fișiere), se creează un anunț automat(“Fișierul a fost încărcat cu succes!”)

De asemenea, implementarea a avut grijă la tranzacțiile bazei de date: folosind SQLAlchemy, operațiile sunt atomice la nivel de sesiune. De exemplu, la crearea unui proiect, se adaugă Project și Participant (pentru creator) și poate altceva, apoi se face o singură dată db.session.commit(); dacă una din adăugări eșuează (ex. integritate referențială), nicio schimbare nu se persistă.

În concluzie, chiar dacă toate aceste componente se află într-un singur serviciu, codul a fost organizat logic pe funcționalități, făcând mai ușoară navigarea și întreținerea.

4.3 Fluxul de date prin aplicație

Fluxul de date descrie modul în care informația este preluată de la utilizator, procesată și stocată, apoi returnată către utilizator sau alt sistem. Vom exemplifica câteva fluxuri importante end-to-end pentru a vedea succesiunea pașilor:

Fluxul general al autentificării (Login):

1. **Input utilizator:** Utilizatorul navighează la pagina de autentificare (/login) și introduce email-ul și parola în formular, apoi apasă butonul "Autentificare".

2. **Trimiterea cererii:** Browser-ul trimite o cerere HTTP POST către serverul Flask la ruta /login, conținând datele formularului (email, parolă și tokenul CSRF).

3. **Procesare pe server:** Funcția corespunzătoare /login primește datele. Folosind WTForms, se validează formatul (e-mail valid, parola nu e goală etc.). Apoi se caută în baza de date utilizatorul cu email-ul furnizat (User.query.filter_by(email=...).first()). Dacă nu este găsit sau dacă parola nu corespunde (se verifică hash-ul stocat folosind Bcrypt), autentificarea eșuează. Dacă este găsit și parola e corectă, se verifică și flag-ul is_banned. Dacă user-ul e blocat, se refuză autentificarea. Altfel, se consideră autentificare reușită.

4. **Actualizare stare sesiune:** În caz de succes, login_user(user, remember=...) este apelat. Aceasta setează în sesiunea utilizatorului (cookie) un token care identifică userul logat. Dacă remember=True, un cookie persitent se va salva pentru a menține login-ul și după închiderea browser-ului.

5. **Stocare date:** Nu se modifică nimic în baza de date la login (în afară de eventual actualizarea unui timestamp de ultim login dacă s-a implementat, dar aici nu e menționat). Datele autentificării (ID-ul user) sunt stocate în memorie (sesiune).

6. **Răspuns către utilizator:** Serverul generează un răspuns redirect, trimițând userul către pagina de proiecte (/projects). În cazul unei autentificări eșuate, serverul reîncarcă pagina de login cu un mesaj flash de eroare.

7. **Prezentare rezultat:** Browser-ul, la primirea redirect-ului, cere /projects. Serverul verifică că există current_user (sesiune validă) – ceea ce e, deci permite accesul – extrage proiectele userului și returnează pagina HTML. Utilizatorul vede acum lista de proiecte, semn că autentificarea a reușit, iar numele său poate apărea în colț (ex. "Logat ca: Ion").

Fluxul de descărcare a unui fișier:

1. **Input utilizator:** În lista de fișiere a unui proiect, utilizatorul face clic pe numele unui fișier (sau pe un buton "Descarcă").

2. **Cerere către server:** Browser-ul inițiază o cerere GET către, de exemplu, /download/45 (unde 45 e id-ul fișierului dorit). Această rută este protejată de login_required, deci trebuie să fii autentificat.

3. **Procesare server:** Funcția atașată la /download/<file_id> caută în BD fișierul cu id-ul 45 (File.query.get_or_404(45)). Găsește intrarea, vede project_id. Verifică că current_user este participant la acel proiect sau admin; dacă nu, returnează 403 Forbidden. Dacă are acces, apelează send_from_directory(upload_folder, file.stored_filename, as_attachment=True, download_name=file.original_filename). Aceasta pregătește un răspuns HTTP de tip fișier atașat (Content-Disposition cu numele original).

4. **Stocare/actualizare:** Nu se modifică nimic în BD (poate opțional incrementare la un counter de descărcări, dar nu cred).

5. **Răspuns:** Serverul trimite răspunsul cu antet Content-Type corespunzător (dacă e PDF -> application/pdf etc.) și cu corpul fișierului (octeții).

6. **Browser acțiune:** Browser-ul detectează headerul de fișier atașat, declanșează mecanismul de descărcare. Utilizatorul vede fereastra de salvare sau fișierul începe să se descarce în locația prestabilită.

7. **Feedback:** În UI nu a apărut nimic vizibil în pagina web (descărcarea e în afara paginii). Dacă descărcarea eșuează (server down sau link expirat), browserul va arăta un mesaj de eroare.

Fluxul de postare a unui anunț și trimiterea notificărilor:

1. **Input utilizator:** În pagina proiectului, utilizatorul scrie un mesaj în formularul de anunț (ex. "Mâine la ora 10: ședință de coordonare.") și apasă "Postează".

2. **Cerere:** Browser-ul trimite POST la /project/<id> (sau altă rută dedicată anunțului) cu conținutul anunțului.

3. **Procesare server:** Serverul primește datele, creează un obiect Announcement: ann = Announcement(content=form.content.data, project_id=project.id, user_id=current_user.id, date_posted=now). Adaugă la BD și commit. După commit, extrage lista de participanți la proiect (mai exact adresele de email ale tuturor userilor în Project_Participants unde project_id = id, exceptând current_user eventual). Construiește un email: subiect "[Proiect X] Anunț nou de la <user.name>", corp cu textul anunțului și un link spre proiect. Pentru fiecare destinatar se trimite email.

4. **Stocare:** Anunțul e salvat în baza de date. Email-urile nu se "salvează" în sistem, sunt acțiuni de ieșire. S-ar putea totuși loga într-o tabelă de log mesaje trimise (nu cred că are așa ceva implementat, deci doar fișierul de log).

5. **Răspuns:** Serverul face redirect la pagina proiectului (PRG pattern din nou) cu mesaj flash "Anunț postat cu succes."

6. **Prezentare:** Browser-ul reîncarcă pagina proiectului, unde acum la începutul listei de anunțuri apare noul anunț cu numele utilizatorului și data. Ceilalți membri, dacă sunt online în același timp, nu văd nimic nou până la refresh (nu e in temps real chat, eventual se poate implementa un refresh periodic; nu specifică deci probabil userii vor vedea când reîncarcă sau primesc email). Concomitent, fiecare membru primește email (în afara aplicației web) pe care îl poate vedea în clientul de mail, fiind notificat imediat.

Aceste fluxuri demonstrează cum circulă datele între browser, server, baza de date şi alte servicii (email). Observăm că arhitectura fiind una simplă, fluxurile sunt lineare şi folosesc mecanisme HTTP standard (formular -> POST -> redirect -> GET -> afişare).

Nu există fluxuri asincrone complexe sau prelucrări offline; totul se întâmplă în contextul cererii. De exemplu, la upload fişier, până nu termină de scris fişierul şi de actualizat BD, nu trimite răspuns (ceea ce e ok pentru fişiere mici, dar la fişiere mari poate duce la timpi mai mari de răspuns; într-o variantă evoluată, ai pune un worker background). Pentru scopul actual, fluxurile sincrone sunt suficiente.

5. Testare

5.1 Strategia de testare

Datorită constrângerilor de timp şi faptului că proiectul nu a inclus testare automată, strategia de testare adoptată a fost preponderent testarea manuală a funcţionalităţilor, pe baza scenariilor de utilizare identificate. Scopul a fost să se asigure că toate cerinţele funcţionale sunt îndeplinite corect şi că aplicaţia se comportă stabil pentru cazurile normale de utilizare.

Abordarea a inclus:

1. **Testare exploratorie manuală**: Am folosit aplicaţia ca un utilizator final, încercând toate funcţiile: înregistrare cont, login, creare proiect, adăugare fişiere, postare anunţuri, logout, etc. S-au urmărit atât traseele "pozitive" (fluxuri normale cu date valide) cât şi cele "negative" (introducerea de date invalide sau acţiuni nepermise) pentru a observa comportamentul aplicaţiei.

2. **Testare pe roluri diferite**: S-au creat trei conturi (unul cu rol de admin, unul cu rol de manager si altul de user normal) pentru a verifica diferenţele de interfaţă şi permisiuni. S-a testat că un user normal nu poate accesa paginile de admin (ex. încercând să acceseze manual URL-ul /admin) şi că primeşte refuz 403. De asemenea, că butoanele de administrare apar doar la admin.

3. **Testare de integrare între module**: Deşi nu formalizată, s-a verificat manual că integrarea dintre componente funcţionează. De exemplu, după crearea unui proiect, acel proiect apare în lista de proiecte şi poate fi accesat; după adăugarea unui user într-un proiect, userul respectiv îl vede în lista sa; dacă un user este blocat de admin, acesta nu se mai poate autentifica; după postarea unui anunţ, ceilalţi primesc email.

4. **Testare de regresie informală**: Pe măsură ce s-au adăugat funcţii noi (conform commit-urilor, funcţionalităţile au fost adăugate incremental), s-a retestat manual întregul flux pentru a se asigura că nicio schimbare nu a rupt altă parte a aplicaţiei. De exemplu, după implementarea a download-ului de fişiere, s-a verificat că upload-ul de fişiere încă funcţionează ca înainte, iar dacă se încalcă criteriile pentru fişiere apare un mesaj de eroare..

Strategia a fost deci una iterativă: dezvoltare de funcție -> testare manuală a funcției -> testare rapidă a funcțiilor anterioare pentru confirmare -> trecere la următoarea funcționalitate.

5.2 Tipuri de teste aplicate

În cadrul efortului de testare manuală, s-au acoperit următoarele tipuri de teste:

1. **Testarea funcțională**: Verificarea că fiecare cerință funcțională este realizată. Pentru fiecare element din lista de cerințe funcționale (secțiunea 2.1), s-au conceput unul sau mai multe scenarii de test. De exemplu:

    1.1. *Autentificare:* test pozitiv (login cu credențiale corecte) -> așteptat să reușească; test negativ (login cu parolă greșită) -> așteptat să afișeze eroare și să nu logheze userul; test cont blocat (setează manual is_banned la True în DB, încearcă login) -> așteptat mesaj "cont blocat".

    1.2. *Înregistrare utilizator:* test înregistrare normală -> contul apare în DB și se poate autentifica; test înregistrare cu email duplicat -> așteptat mesaj de eroare (formularul ar trebui să arate "Email deja folosit", implementat cu WTForms validator).

    1.3. *Creare proiect:* test nume proiect minim/ maxim (vede dacă sunt limite, ex. peste 100 caractere? dacă da, testez limită); test crearea proiectului și verificare că apare în listă și în DB.

    1.4. *Încărcare fișier:* test cu fișier permis (PNG, mic) -> apare în listă, se poate descărca; test cu fișier nepermis (script .exe) -> aplicația refuză și arată mesaj de eroare fără să se adauge nimic în DB; test fișier mare (peste limită dacă e setată) -> refuzat cu mesaj.

    1.5. *Descărcare fișier:* test descărcare de pe un browser -> fișierul se salvează și este corect; test descărcare de către user neparticipant (introduc manual URL-ul ca alt user) -> serverul dă 403, în browser apare "Forbidden".

    1.6. *Postare anunț:* test normal (text scurt) -> apare în pagină, trimite email; test injecție HTML (scriu <script>alert('x')</script>) -> în pagină ar trebui să apară ca text neinterpretat (pentru a preveni XSS, Jinja2 by default escape-uieste, testez că nu se execută scriptul).

    1.7. *Notificări email:* test resetare parolă (dacă implementat) -> primit email cu token; test anunț -> toți destinatarii primesc email; test invite proiect -> destinatar primește email. Acestea au necesitat acces la cutiile poștale de test (s-au folosit adrese reale, cele personale).

    1.8. *Administrare utilizatori:* test schimbare rol -> după acțiune, userul are noile permisiuni (ex. devine admin, se verifică că poate accesa /admin); test blocare -> userul nu se mai

poate loga; test ștergere -> userul dispare din listă, nu mai e în DB, proiectele lui rămân orfane? (dacă da, se vede cum arată proiectul fără manager, eventual se decide auto-atribuire la admin).

2. **Testarea UI/UX (testare de interfață)**: S-a parcurs aplicația asigurându-ne că elementele UI sunt intuitive și funcționează:

    2.1. Verificat că toate butoanele și link-urile fac ceea ce indică (de ex. click pe logo duce la pagina corectă, butonul "Șterge proiect" deschide confirmare și apoi șterge).

    2.2. Verificat aspectul pe diferite rezoluții: redimensionat fereastra browser-ului pentru a observa cum se adaptează (pe mobil, meniul devine burger etc.).

    2.3. Verificat mesaje de eroare/succes: apar la momentul potrivit și dispar la navigare (Flash messages).

    2.4. Verificat că formularele nu permit trimiterea dacă lipsesc date obligatorii (ex. butonul "Submit" dezactivat sau browser-ul indică "fill this field" datorită atributelor required).

3. **Testarea de securitate de bază**:

    3.1. Incercări de acces direct la URL-uri neautorizate: fără a fi logat, introdus manual /projects -> așteptat redirect la login (test reușit, Flask-Login ar trebui să facă asta); ca user obișnuit, accesat manual /admin -> așteptat pagina 403 Forbidden (testat).

    3.2. Test CSRF: încercat să trimit un formular fără token (posibil greu manual, dar s-a observat în rețeaua browserului că fiecare POST include tokenul și că dacă l-am eliminat (prin editare manuală a HTML în DevTools) serverul a refuzat "CSRF token missing" deci protecția e activă).

    3.3. Test input injection: XSS (cum am zis mai sus), SQL injection (încercat să introduc '; DROP TABLE users; -- în câmpuri de text, de exemplu la nume proiect, și verificat că fie e scăpat, fie oricum ORM-ul paramatrizează și nu are efect cel mai important că nu "stric" nimic; test reușit, nimic rău nu s-a întâmplat, textul a apărut ca literali în UI sau a fost filtrat).

    3.4. Parole: verificat că sunt stocate criptat în DB (uitându-ne direct în fișierul SQLite după înregistrare, parola apărea ca hash nu ca text clar).

4. **Testare de performanță informală**: Nu s-a folosit un tool dedicat, dar s-a observat cum se mișcă aplicația cu date de test:

    4.1. S-au adăugat ~10 proiecte, ~5 utilizatori, ~20 fișiere, ~15 anunțuri pentru a simula o utilizare reală moderată. Aplicația a continuat să răspundă rapid, paginile încărcându-se

sub o secundă pe localhost.

4.2. S-a testat și comportamentul la fișiere mai mari (ex. un PDF de 5MB), upload-ul a durat câteva secunde, ceea ce e normal, și aplicația nu a dat timeout (Flask by default are un limită destul de mare, deci a fost ok).

5. **Testare pe diferite medii**: În principal s-a testat pe mediul de dezvoltare local (Windows, cu Flask dev server). Pentru a simula cum ar merge în alt mediu, s-a rulat și pe o mașină Linux (VirtualBox) – verificând că instalarea dependențelor prin pip install -r requirements.txt funcționează și aplicația rulează identic. De asemenea, s-a deschis aplicația și în browsere diferite (Chrome, Firefox) pentru a asigura compatibilitatea UI, ceea ce nu a ridicat probleme fiind vorba de HTML standard și Bootstrap.

5.3 Plan de testare (cazuri de test exemplare)

Deși testarea a fost preponderent manuală, putem sumariza într-un **plan de testare** câteva cazuri de test esențiale pentru fiecare funcționalitate, cu rezultatele așteptate:

1. **Test: Înregistrare cu date valide**

   1.1. **Precondiție:** Utilizatorul nu este logat, se află pe pagina de Register.

   1.2. **Pași:** Introduce username="testuser", email="test@example.com", password="Test123!", confirm password la fel. Apasă "Înregistrare".

   1.3. **Rezultat așteptat:** Contul este creat, apare mesaj "Contul a fost creat, vă puteți autentifica", apoi redirecționat la Login. În baza de date, tabela users conține un user nou cu username testuser, parolă hash-uită, rol implicit "user". Un email de bun venit se trimite (dacă implementat).

2. **Test: Login eșuat (parolă greșită)**

   2.1. **Precondiție:** Există un cont user1 cu parolă cunoscută "abc".

   2.2. **Pași:** În formularul de login, introducem emailul user1 (cel folosit la înregistrare) și o parolă greșită "xyz".

   2.3. **Rezultat așteptat:** Autentificarea nu are loc. Aplicația reafișează pagina de login cu un mesaj flash "Autentificare eșuată". Sesiunea utilizator rămâne neautentificată. În log, se poate nota opțional încercarea eșuată.

3. **Test: Creare proiect și vizualizare**

3.1. **Precondiție:** Utilizatorul "Alice" este logat.

3.2. **Pași:** Navighează la "Adaugă Proiect". Completează Name="Proj A", Description="Descriere proiect A". Submit.

3.3. **Rezultat așteptat:** Este redirecționată la pagina proiectului nou "Proj A". Mesaj flash "Proiect creat". În baza de date, Project cu nume "Proj A" există, având creator_id = Alice.id, manager_id = Alice.id. În Project_Participants există o intrare (Alice, ProjA). În UI, pagina proiectului arată titlul, descrierea, 0 anunțuri, 0 fișiere, 1 participant (Alice).

4. **Test: Încărcare fișier permis**

4.1. **Precondiție:** Alice se află în pagina proiectului Proj A, are formular de upload. Fișier test: "document.pdf" de 50KB existent local.

4.2. **Pași:** Selectează "document.pdf", apasă Upload.

4.3. **Rezultat așteptat:** Pagina se reîncarcă, mesaj "Fișierul a fost încărcat". În lista de fișiere apare "document.pdf", uploader Alice, data curentă. Pe server, în folderul uploads, există un fișier, ex "5f2c9d_document.pdf" (nume schimbat). În DB, o intrare File cu original_filename="document.pdf", stored_filename="5f2c9d_document.pdf", project_id=ProjA.id, uploader_id=Alice.id. Alice poate da click pe fișier și descarcă exact conținutul original (fișierul e intact).

5. **Test: Încărcare fișier nepermis**

5.1. **Precondiție:** Alice pe aceeași pagină proiect. Fișier test: "malware.exe".

5.2. **Pași:** Selectează "malware.exe", Upload.

5.3. **Rezultat așteptat:** Încărcarea este refuzată imediat după submit. Pagina se reîncarcă sau tot acolo rămâne, afișând un mesaj de eroare "Tip de fișier nepermis". Nicio intrare nu se adaugă în DB, fișierul nu se salvează pe disc. Sesiunea rămâne stabilă.

6. **Test: Postare anunț și notificare email**

6.1. **Precondiție:** În Proj A, există doi membri: Alice și Bob. Bob are email real configurat. Bob nu este online.

6.2. **Pași (Alice):** În Proj A page, la secțiunea Anunțuri, scrie "Salut, am urcat fișierul de date." și apasă Postează.

6.3. **Rezultat așteptat:** Pe pagina web, anunțul apare imediat listat cu autor Alice, data acum. Mesaj "Anunț adăugat". În DB, Announcement nou cu content dat, user_id=Alice,

project_id=ProjA. Sistemul trimite email către Bob (și eventual Alice însă și-a trimis sieși? de preferat nu). Bob primește un email cu subiect ce conține "Proj A" și corp "Salut, am urcat fișierul de date." plus cine a trimis. (Dacă Bob era online și refresh, vede anunțul).

7. **Test: Schimbare rol utilizator de către admin**

   7.1. **Precondiție:** Utilizator "Charlie" este admin, alt utilizator "Dave" este normal. Charlie logat.

   7.2. **Pași:** Charlie merge la pagina administrare (Adminsitrator utilizatori). Găsește pe Dave în listă, la coloană Rol selectează "admin" (sau apasă buton "Fă admin"). Confirmă dacă e vreo confirmare.

   7.3. **Rezultat așteptat:** Pagina se reîncarcă, Dave apare acum cu rol "admin". În DB, Dave.role = 'admin'. Dave se poate acum loga (sau dacă era logat, la refresh primește drepturi noi, oricum). Log-ul poate nota "User X changed role of Y to admin".

8. **Test: Bănare utilizator**

   8.1. **Precondiție:** Charlie admin, Dave user. Dave logat în alt browser eventual.

   8.2. **Pași:** Charlie apasă "Blochează" la Dave. Introduce eventual dată de debanare sau permanent. Confirmă.

   8.3. **Rezultat așteptat:** În DB, Dave.is_banned = True, banned_until (dacă dat). Pagina admin arată Dave ca "blocat". Dacă Dave încearcă să navigheze sau să folosească site-ul în browserul său autenticat, la următoarea cerere protejată fie va fi dat afară (logout force), fie dacă încearcă re-login îi spune "cont blocat". Ca implementare simplă, probabil dacă Dave e deja logat, nu îl dă automat afară, dar dacă face logout nu mai poate intra sau dacă expiră sesiunea gata.

9. **Test: Ștergere proiect**

   9.1. **Precondiție:** Alice manager pe Proj A, Proj A are ceva conținut.

   9.2. **Pași:** Alice apasă "Șterge Proiect", confirmă la prompt.

   9.3. **Rezultat așteptat:** Este redirecționată la lista de proiecte cu mesaj "Proiect șters". În DB, Proj A și toate anunțurile și fișierele asociate nu mai există (șterse cascada). În folderul uploads, fișierele Proj A pot fi de asemenea șterse (dacă implementat manual). Bob (participant) nu mai vede Proj A în lista lui.

10. **Test: Logout**

    10.1. **Precondiție:** Orice utilizator logat.

    10.2. **Pași:** Apasă "Logout" din meniu.

    10.3. **Rezultat așteptat:** Sesiunea este curățată (cookie-ul login invalidat). Utilizatorul ajunge la pagina de login cu mesaj "Ai fost deconectat" (opțional). Dacă încearcă să apese "Back" la o pagină protejată, este redirecționat la login (nu poate vedea conținutul cached fără login datorită @login_required reacționând la acces).

Aceste teste acoperă funcționalitatea de bază. Rezultatele obținute în timpul testării manuale au fost satisfăcătoare, rezolvându-se micile probleme descoperite (de ex., inițial poate validările de formular au fost incomplete, sau s-a corectat logică de permisiuni). Nu au fost identificate defecte critice majore în urma acestor teste, iar aplicația s-a dovedit stabilă în condiții normale de utilizare.

Totuși, pentru a crește încrederea în aplicație, se recomandă ca pe viitor să se elaboreze teste automate:

1. **Teste unitare** pentru funcțiile esențiale (ex: o funcție care validează extensia fișierului să fie testată cu diferite inputuri).

2. **Teste de integrare** folosind un client de test Flask (Flask provide test_client), pentru a simula cereri la rute și a verifica răspunsuri și efecte în baza de date. Acest lucru ar fi util mai ales la autentificare și permisiuni.

3. **Teste end-to-end** (posibil cu Selenium sau similar) dacă s-ar dori simularea unui browser real complet.

În contextul proiectului actual, testarea manuală riguroasă a fost considerată suficientă pentru a atinge un nivel bun de calitate înainte de prezentare sau livrare.

6. Deployment și mentenanță

6.1 Strategie de deployment

La momentul redactării documentației, aplicația nu a fost încă deployată pe un mediu de producție sau pe un server accesibil public. Ea a fost rulată și testată local, folosind serverul de dezvoltare Flask. Strategia de deployment viitoare ia în considerare pașii și ajustările necesare pentru a face trecerea de la mediul de dezvoltare la un mediu de producție stabil:

1. **Alegerea infrastructurii**: O primă decizie este unde va rula aplicația. Fiind o aplicație Python Flask, există mai multe opțiuni:

    1.1. Un VPS (Virtual Private Server) sau un server on-premises, unde să se instaleze manual Python și dependințele, configurând apoi un server WSGI (Gunicorn/ uWSGI) + un server web (Nginx/Apache) pentru a servi aplicația. Aceasta oferă mult control și este potrivit dacă instituția (ex. universitatea) are un server propriu.

    1.2. Un serviciu de Platform-as-a-Service precum Heroku, PythonAnywhere, Railway etc., care simplifică deploy-ul. De exemplu, pe Heroku s-ar face push la codul proiectului (inclusiv un Procfile care rulează Gunicorn cu app:app). Heroku suportă Flask ușor și are add-on pentru SQLite (sau preferabil Postgres, deoarece SQLite nu e bun pentru mediu multi-server).

    1.3. Containerizare Docker: Se poate crea un Dockerfile pentru aplicație (bazat pe o imagine python:3.10), se includ codul și se instalează dependințele, se expune portul Flask. Containerul se poate rula pe orice host ce are Docker, sau pe un cluster (Kubernetes, Docker Swarm). Aceasta ar asigura consistența mediului de rulare. De exemplu, un Dockerfile ar copia sursele și rula Gunicorn. Pnetru SQLite nu sunt probleme, doar volumul fișierului DB și al upload-urilor trebuie mapat pe storage persistent.

    1.4. Serverless: mai puțin potrivit pentru această aplicație (datorită upload-urilor de fișiere și sesiuni persistente), deci nu s-a luat în calcul.

2. **Configurarea mediului de producție**: Oricare ar fi platforma aleasă, trebuie adaptate câteva setări:

    2.1. Generarea unei SECRET_KEY securizate pentru producție (momentan folosesc o variabila dintr-un fișier .env și este o parola slabă) . Aceasta se stochează ca variabilă de mediu pe server și aplicația o citește (app.config['SECRET_KEY'] = os.environ.get('SECRET_KEY')).

    2.2. Setarea variabilelor de configurare pentru email (MAIL_USERNAME, parole) tot ca secrete de mediu, nu în cod. Astfel, se păstrează siguranța credentialelor.

    2.3. Schimbarea modului debug: app.run(debug=False) sau echivalent. În producție, debug trebuie dezactivat pentru securitate (nu expunem traceback-uri utilizatorilor).

    2.4. **Logging**: În producție, log-urile pot fi redirecționate către stdout (dacă e container) sau către un serviciu de logging centralizat. De asemenea, se poate crește nivelul la WARNING pentru a nu avea fișiere imense. În mod ideal, se configurează și rotirea log-urilor (log rotation).

2.5. **Database**: SQLite poate fi folosit în continuare dacă traficul e mic și doar o instanță de aplicație va rula. Dar pentru robustețe, ar fi recomandată migrarea la un DB server, ex. PostgreSQL. Codul cu SQLAlchemy face tranziția relativ ușoară (doar schimbarea DATABASE_URI). Dacă se rămâne pe SQLite, asigurarea backupului fisierului devine crucială.

2.6. **File storage**: Fișierele uploadate în producție ar trebui stocate într-o cale permanentă. Dacă se folosește Docker, atunci trebuie montat un volum la container, altfel fișierele se pierd la redeploy. O variantă mai robustă este stocare externă (ex. Amazon S3 bucket) și servirea direct de acolo, însă asta necesită modificări în cod (nu majore, dar extra nu implementat încă). Pentru început, se poate păstra stocarea locală dar cu backup.

3. **Procesul de deploy efectiv**:

3.1. **Instalare dependințe**: Se rulează pip install -r requirements.txt pe server sau se include în Dockerfile. Astfel se obțin Flask și celelalte.

3.2. **Migrare baza de date**: Dacă DB-ul e gol la început, se poate rula db.create_all() (SQLAlchemy) pentru a crea schema. Dacă sunt migrations (Flask-Migrate), se rulează flask db upgrade. Apoi, eventual, populare cu cont admin inițial.

3.3. **Pornirea aplicației**: În loc de flask run, se folosește un server WSGI. Un exemplu: gunicorn app:app --bind 0.0.0.0:8000 --workers 4. Gunicorn va porni 4 procese worker, gestionând mai bine cererile decât serverul dev. Apoi un server Nginx poate sta în față la port 80, delegând intern la gunicorn (sau Heroku face asta intern).

3.4. **Configurare domeniu**: Dacă aplicația e destinată accesului general, se va avea un URL (ex. un subdomeniu gen collab.example.edu). DNS-ul trebuie configurat către server, iar serverul web (Nginx) configurat să servească acel domeniu. De asemenea, instalarea unui certificat SSL (folosind Let's Encrypt de ex.) pentru HTTPS. Aceasta nu afectează aplicația Flask direct, dar e parte a configurării de deploy.

3.5. **Testing post-deploy**: După ce aplicația e pornită în noul mediu, se repetă testele critice manual (sanity check): se creează un user, se loghează, se fac câțiva pași. Asta asigură că diferențele de mediu (sistem de fișiere, case sensitivity la Linux vs Windows etc.) nu au introdus bug-uri.

Momentan, utilizatorii folosesc aplicația local sau pe rețeaua internă de test. Dar planul de deployment, conform celor de mai sus, ar fi implementat pentru a oferi acces platformei într-un mediu real. Documentația de față, împreună cu scripturile (ex. un Dockerfile sau un requirements.txt complet), constituie baza pentru oricine ar prelua sarcina deploy-ului.

6.2 Administrarea și monitorizarea

În absența unui deployment activ, administrarea platformei s-a limitat la acțiuni manuale efectuate de dezvoltator în mediul de dezvoltare. Însă, odată ce aplicația ar fi lansată, este necesar un plan de administrare și monitorizare pentru a asigura continuitatea serviciului și o performanță optimă:

1. **Administrarea utilizatorilor și conținutului**: Rolul de Administrator în aplicație permite deja gestionarea utilizatorilor (schimbare rol, blocare, ștergere) și a proiectelor (ștergere, eventual moderarea conținutului anunțurilor dacă ar fi nevoie). Un aspect administrativ adițional ar fi monitorizarea conținutului încărcat pentru a preveni abuzul – de exemplu, ca admin, să inspectezi periodic fișierele încărcate (de teama conținutului nepotrivit) sau anunțurile (să nu derive discuțiile în off-topic). Acest proces ar fi manual, neexistând filtre automate de conținut.

2. **Monitorizarea erorilor și jurnalelor**: Admin-ul tehnic (cel care gestionează serverul) ar trebui să verifice regulat fișierul de log al aplicației pentru erori. În caz de repetare a unor erori, se va anunța echipa de dezvoltare pentru remediere. Pentru a ușura asta, se poate instala un sistem de centralizare a log-urilor (ex. configurarea Flask să trimită log-urile critice prin email sau într-un dashboard extern). Momentan, metoda este manuală: accesarea serverului și consultarea logului.

3. **Monitorizare performanță**: Într-un scenariu de utilizare reală, se pot urmări metrici precum utilizarea CPU/RAM a serverului, timpul de răspuns al aplicației sub încărcare, dimensiunea bazei de date și a spațiului de fișiere utilizat. Instrumente precum Prometheus/Grafana sau chiar monitorizarea oferită de hosting (ex. New Relic, AWS CloudWatch dacă în cloud) pot fi implementate. Pentru început însă, se poate recurge la scripturi simple sau monitorizare manuală: verificarea procesului Gunicorn, log-urilor de acces, etc.

4. **Backup și recuperare**: Un aspect esențial al administrării este realizarea de backup-uri regulate. Cel puțin fișierul SQLite și directorul de upload trebuie salvate periodic (zilnic sau săptămânal) pe un mediu separat. De exemplu, un script cron care copiază site.db și uploads/ într-un storage extern (sau le uploadează într-un bucket securizat). În caz de dezastru (corupere, ștergere accidentală), adminul trebuie să poată restaura aceste fișiere. Testarea procedurii de restaurare este la fel de importantă – de exemplu, verificat că un backup copiat peste datele curente aduce sistemul la starea dorită.

5. **Actualizare și mentenanță aplicație**: Adminul trebuie să programeze ferestre de mentenanță pentru a aplica actualizări ale codului sau pentru a migra versiuni. Asta implică anunțarea utilizatorilor (poate tot printr-un anunț general în platformă sau email) că la o anumită oră platforma va fi temporar indisponibilă. Apoi oprirea serverului, aplicarea noii versiuni (git pull / redeploy container), rularea eventualelor migrări de DB, repornirea. Datorită naturii academice a proiectului, frecvența update-urilor nu e mare, dar tot trebuie planificată.

6. **Gestionarea conturilor de e-mail**: Dacă se folosește un cont Gmail pentru trimiterea emailurilor, adminul trebuie să se asigure că acel cont are setările necesare (accesul aplicații mai puțin sigure – acum Gmail a restricționat asta, deci ar fi preferabil generarea unui App Password sau folosirea unui serviciu de email dedicat). De asemenea, monitorizarea inbox-ului de la care se trimit

mailurile (în caz de bounce-uri sau notificări).

7. **Scalare manuală**: Dacă se observă creşteri de încărcare, adminul poate decide mutarea pe un server mai puternic sau creşterea numărului de worker-e Gunicorn. Aceasta e o intervenție manuală, dar face parte din mentenanță. De exemplu, dacă log-urile arată constant multe cereri simultane şi întârzieri, se pot aloca 2 CPU în plus şi 2 workeri Gunicorn suplimentari.

În rezumat, administrarea curentă necesită implicare umană şi observație directă. Pe termen lung, dacă platforma devine critică pentru o comunitate largă, s-ar impune adoptarea unor soluții automate:

A. un *stack ELK (Elasticsearch-Logstash-Kibana)* pentru agregarea log-urilor,

B. *monitoring* cu alerte (ex. folosirea unui uptime monitor extern, care să notifice adminul prin email/SMS dacă site-ul cade),

C. *proces de deploy automat* (CI/CD) ca să reducă erorile manuale la update,

D. *container orchestration* (Kubernetes) dacă ajunge la nevoia de a rula pe mai multe instanțe.

Pentru stadiul actual însă, un admin dedicat (sau dezvoltatorul însuşi) care urmăreşte starea aplicației şi reacționează la probleme este considerat suficient.

6.3 Plan de mentenanță

Mentenanța aplicației presupune atât rezolvarea problemelor apărute după livrare, cât şi actualizarea şi îmbunătățirea continuă a sistemului. Planul de mentenanță pentru acest proiect cuprinde:

1. **Mentenanță corectivă (bug fixing)**: Dacă utilizatorii sau monitorizarea detectează bug-uri în funcționalitate, acestea vor fi documentate (ideal într-un sistem de tracking al erorilor) şi adresate conform severității lor. De exemplu, o eroare critică (aplicația se blochează la upload fişier peste 16MB) ar necesita o intervenție imediată prin patch/hotfix. Erori minore de UI sau mesaje greşite pot fi grupate şi rezolvate în următoarea versiune planificată. Procesul ar fi:

    1.1. Reproducerea bug-ului într-un mediu de test (folosind eventual o clonă a bazei de date dacă e ceva legat de date).

    1.2. Diagnosticarea cauzei (analiză de cod, verificare log).

    1.3. Implementarea corecției şi testarea ei local.

    1.4. Deploy patch în producție (dacă e urgent) sau cumulat cu alte modificări (dacă e minor).

    1.5. Actualizarea documentației, dacă bug-ul a fost cauzat de o neînțelegere a comportamentului aşteptat (uneori şi userii trebuie informați asupra unor modificări).

2. **Mentenanță adaptivă**: Adaptarea la noi cerințe de mediu sau infrastructură. De exemplu, dacă se decide mutarea bazei de date la PostgreSQL pentru scalabilitate, codul va trebui adaptat (verificare compatibilitate queries, instalare drivere, migrare date). Sau dacă se schimbă furnizorul de email (de la Gmail la un SMTP intern), se vor actualiza configurațiile și eventual modul de autentificare. Un alt exemplu: dacă se containerizează aplicația, vor fi create configuri specifice (health checks, secrets management).

3. **Mentenanță perfectivă (îmbunătățiri și noi funcționalități)**: Chiar dacă proiectul acoperă cerințele inițiale, este de așteptat ca utilizatorii sau stakeholderii să dorească extinderi pe viitor. Posibile îmbunătățiri deja identificate:

    3.1. Implementarea unui sistem de versiuni de fișiere: în loc ca utilizatorii să vadă doar ultima versiune a unui document, să poată încărca o nouă versiune și vechea să rămână arhivată. Ar implica modificări la modelul de date și UI.

    3.2. Adăugarea de comentarii sau forum pe proiect, în completarea anunțurilor (care sunt un mod de comunicare one-to-many). Poate un modul de discuții pe subiecte.

    3.3. Introducerea de etichetare/organizare a proiectelor (ex. categorii, cuvinte cheie) pentru a ușura căutarea proiectelor relevante.

    3.4. Integrarea cu servicii externe: de exemplu, conectarea la o API bibliografică sau la un sistem de management al referințelor, dacă cercetătorii au nevoie.

    3.5. Optimizări de UI: poate un *dark mode* (preferință tot mai comună), sau localizarea interfeței în engleză dacă se extinde la colaboratori internaționali.

    3.6. Automatizarea testării și deploy-ului: din punct de vedere al procesului de dezvoltare, perfective ar fi adăugarea de teste unitare și de integrare, configurarea unui pipeline CI (ex. GitHub Actions) care la fiecare commit rulează testele. Astfel, orice dezvoltare viitoare se face cu siguranță că nu rupe funcționalitatea existentă.

4. **Mentenanță preventivă**: Aceasta include acțiuni de prevenire a problemelor înainte ca ele să devină critice. Exemplu: actualizarea periodică a dependențelor pentru a beneficia de patch-uri de securitate. Dacă Flask sau alte biblioteci emit versiuni noi care rezolvă vulnerabilități, se vor planifica upgrade-uri controlate (test întâi local cu noua versiune, apoi în producție). De asemenea, monitorizarea log-urilor face parte tot din mentenanță preventivă – dacă se observă multe tentative eșuate de login (posibil bruteforce), se poate implementa preventiv un sistem de rate-limiting la login.

5. **Suport pentru utilizatori**: În planul de mentenanță intră și asigurarea unui canal de suport. Utilizatorii (cercetătorii) ar trebui să știe cum pot raporta o problemă sau cere asistență (ex. un email al adminului, sau un formular de contact în aplicație). Răspunsul prompt la aceste solicitări face parte din mentenanță. În plus, documentația de utilizare (un scurt ghid cum se folosește

platforma) ar putea fi întreținut și actualizat cu noile funcții.

6. **Plan de retragere (eventual)**: Orice aplicație ar trebui să aibă și un plan de ce se întâmplă dacă, în viitor, se renunță la ea sau se înlocuiește. Asta presupune cum se extrag datele (ex. export proiecte și fișiere într-o structură de backup pe care utilizatorii s-o primească). Deși nu e un subiect plăcut, e o bună practică să fie menționat pentru completitudine.

Pe durata mentenanței, comunicarea cu toți actorii e importantă: echipa de dezvoltare, adminul de sistem și utilizatorii finali. Se vor programa actualizările majore astfel încât să afecteze minim activitatea (de ex., update-urile se fac în afara orelor de lucru tipice ale cercetătorilor). Se vor documenta schimbările (changelog) și dacă sunt modificări vizibile pentru utilizatori, se vor anunța, eventual punând anunț pe platformă ("În data X platforma va fi indisponibilă între 18:00-19:00 pentru upgrade. După upgrade, veți avea posibilitatea să…").

7. Concluzii

Proiectul de față o platformă web de colaborare pentru cercetători în domeniul construcțiilor a atins cu succes obiectivele inițiale, oferind un set de funcționalități esențiale într-un mediu unitar și ușor de folosit. Prin intermediul aplicației implementate cu Flask (Python), SQLite și Bootstrap, utilizatorii pot să:

1. își creeze conturi și să se autentifice în siguranță;

2. gestioneze proiecte de cercetare, invitând colegi și partajând resurse;

3. încarce și descarce fișiere relevante pentru munca lor, având certitudinea că toți membrii proiectului accesează versiunea corectă a documentelor;

4. comunice eficient prin anunțuri centralizate, fiind notificați prompt prin email despre noutăți;

5. beneficieze de o interfață intuitivă care nu necesită pregătire tehnică specială.

Din punct de vedere tehnic, proiectul demonstrează o arhitectură robustă pentru scara sa: modelul de date relațional este bine definit și extensibil, iar folosirea Flask și a extensiilor sale a condus la un cod clar și concis. Securitatea de bază este acoperită prin autentificare, hashing-ul parolelor, protecția CSRF și verificarea permisiunilor pe rute, ceea ce oferă încredere că datele sensibile sunt protejate adecvat. De asemenea, sistemul de logging implementat asigură vizibilitate asupra funcționării interne și ajută la depanarea eventualelor probleme.

Proiectul a fost testat manual în mod iterativ, scenariile uzuale de utilizare funcționând conform așteptărilor. Utilizatorii de diferite roluri pot interacționa conform restricțiilor stabilite, iar erorile întâlnite pe parcursul dezvoltării au fost corectate. Chiar dacă nu există teste automate, aplicația s-a dovedit stabilă în condiții normale, nefiind identificate defecte majore în versiunea finală prezentată.

Un **aspect notabil** este că, deși aplicația nu a fost încă lansată public (deployată), baza a fost pregătită pentru a permite un astfel de pas fără eforturi semnificative. Este nevoie de configurări suplimentare minore pentru mediul de producție (server WSGI, setări de email și securitate), însă codul și infrastructura actuală sunt în mare parte reutilizabile. Documentația de față va ghida procesul de deploy, asigurând că nimic important nu este omis.

Acest proiect constituie un *proof of concept* și totodată un punct de plecare pentru dezvoltări viitoare. În cadrul concluziilor, putem evidenția câteva direcții de dezvoltare ulterioară:

1. **Scalabilitate și performanță**: Pentru un număr restrâns de utilizatori, soluția actuală este adecvată. Însă, dacă platforma ar fi adoptată de o comunitate mai mare, ar fi recomandată migrarea la tehnologii mai puternice (o bază de date server, caching, poate chiar împărțirea pe microservicii). De asemenea, implementarea unor teste de stres ar fi utilă pentru a cunoaște limitele sistemului.

2. **Experiența utilizatorilor**: Feedback-ul colectat (informal) de la primii utilizatori sugerează că aplicația este ușor de folosit. Totuși, îmbunătățiri ar fi posibile, cum ar fi: funcții de căutare (căutare proiect după cuvinte cheie), posibilitatea de a comenta direct pe fișiere sau anunțuri (pentru discuții mai fine), sau integrarea unui calendar pentru a programa întâlniri de proiect. Aceste funcționalități ar spori valoarea platformei pentru cercetători.

3. **Securitate**: Deși securitatea de bază este acoperită, în viitor s-ar putea adăuga niveluri suplimentare, având în vedere natura potențial sensibilă a datelor de cercetare. De exemplu, suport pentru autentificare cu doi factori (2FA) pentru conturi, audit trail mai detaliat (cine a descărcat ce și când), criptarea fișierelor stocate (dacă se consideră necesar pentru protejarea datelor în repaus).

4. **Mentenanță și comunitate**: Proiectul ar putea fi publicat pe un repository (deja este pe GitHub) și dacă devine util mai multor grupuri, s-ar putea forma o comunitate care să contribuie la cod. Practicile de engineering precum codul sursă versiunut, CI/CD și issue tracking ar fi binevenite pentru a gestiona contribuțiile și bug-urile pe viitor.

În ansamblu, platforma realizată demonstrează cum unelte open-source și o arhitectură bine gândită pot satisface nevoi specifice unui domeniu (cercetarea în construcții) cu un efort relativ redus. Prin centralizarea colaborării într-un sistem dedicat, cercetătorii își pot coordona mai bine eforturile și pot economisi timp prețios, evitând haosul schimbului de emailuri și documente dispersate.

Proiectul a reprezentat totodată o ocazie de învățare valoroasă: s-au consolidat cunoștințele privind dezvoltarea web cu Flask, s-au practicat conceptele de proiectare a bazelor de date și securizare a aplicațiilor, și s-a experimentat cu ciclul complet de viață al dezvoltării software (de la analiza cerințelor, la implementare, testare și documentare).

În concluzie, aplicația este pregătită să iasă din faza de prototip și, cu mici ajustări pentru producție, să fie pusă la dispoziția comunității țintă. Se așteaptă ca prin utilizarea acestei platforme, colaborarea între

cercetători să devină mai eficientă, transparența în proiecte să crească, iar gestionarea datelor și comunicărilor să fie mult îmbunătățită față de metodele tradiționale. Aceasta va contribui, sperăm, la realizarea de progrese mai rapide și la o mai bună organizare în proiectele de cercetare din domeniul construcțiilor.

Documentația de față va servi ca referință pentru oricine va continua munca pe acest proiect, asigurând o tranziție ușoară și permițând extinderea sa într-un mod coerent și informat.