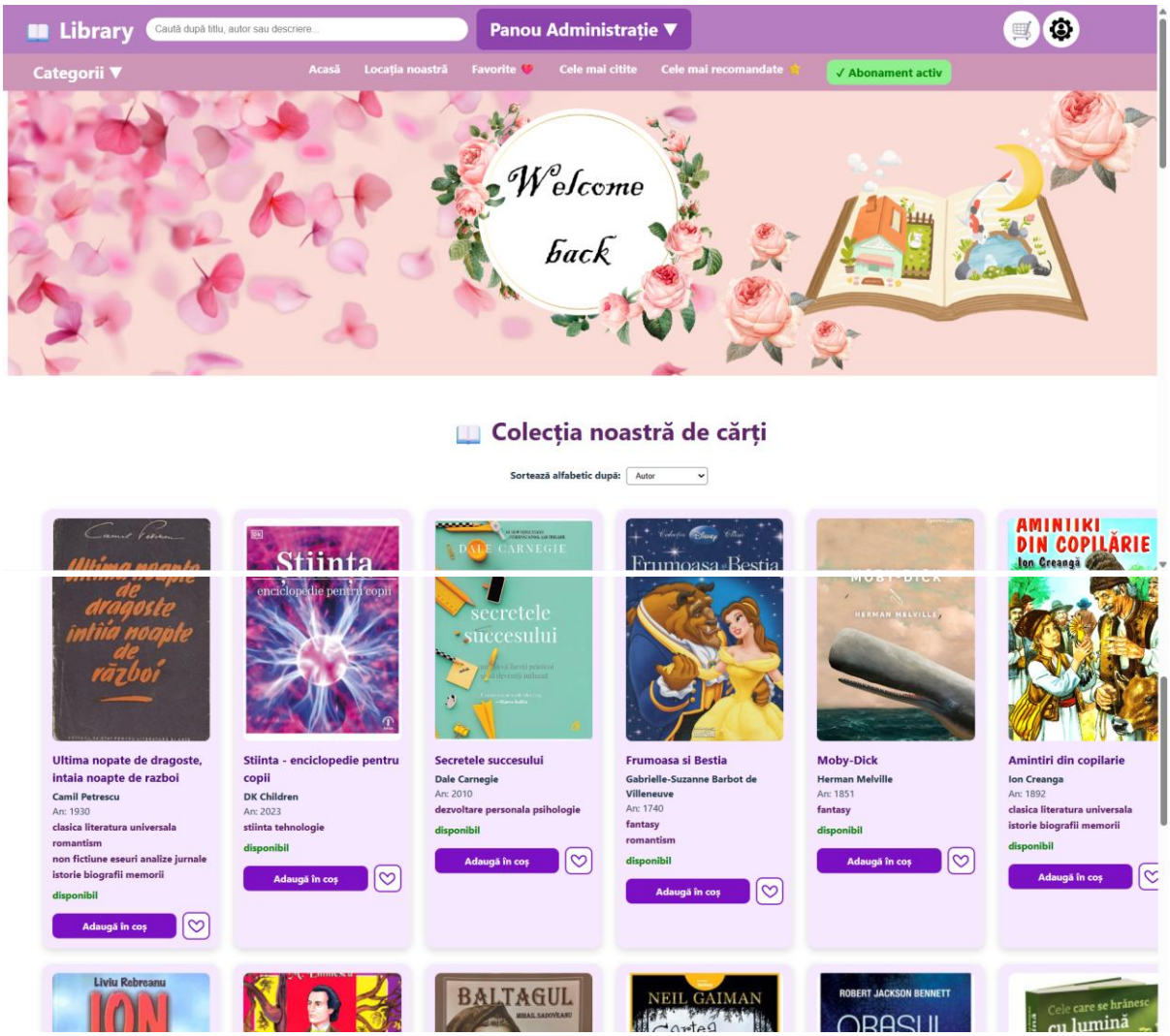
UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Biblioteca digitala



Studente: Berciu Oana-Georgiana
Szarka Francesca- Lara
Puicar Georgiana - Oana
Grupa 30234
Facultate de Calculatoare si Tehnologia Informatiei

Cuprins:

1. Specificațiile proiectului

1.1 Scopul proiectului

Scopul acestui proiect este dezvoltarea unei aplicații web de tip bibliotecă digitală, care permite gestionarea cărților și interacțiunea utilizatorilor cu acestea. Aplicația oferă funcționalități precum vizualizarea cărților disponibile/indisponibila, cautarea acestora,sortarea acestora, adaugarea intr-o lista de favorite,împrumutul și administrarea conținutului de către un utilizator cu rol de administrator.

Proiectul urmărește realizarea unei soluții funcționale, ușor de utilizat, care să integreze un backend robust și un frontend intuitiv, respectând principiile programării orientate pe obiecte și utilizând design pattern-uri adecvate.

1.2.Obiectivele proiectului

Obiectivul principal al acestui proiect este dezvoltarea unei aplicații web de tip bibliotecă digitală, realizată cu scop educațional, care să permită aprofundarea și aplicarea practică a conceptelor studiate în cadrul cursului, în special a tehnologiilor **Spring Boot** pentru partea de backend și **React** pentru partea de frontend.

În vederea atingerii acestui obiectiv general, proiectul urmărește îndeplinirea următoarelor obiective specifice:

- dobândirea unei înțelegeri aprofundate a arhitecturii aplicațiilor web de tip **client–server** și a modului de comunicare dintre frontend și backend prin intermediul serviciilor REST;
- aplicarea practică a framework-ului **Spring Boot** pentru dezvoltarea unui backend scalabil, incluzând utilizarea controllerelor, repository-urilor și mecanismelor de *dependency injection*;
- dezvoltarea unei interfețe web moderne utilizând **React**, cu accent pe componente reutilizabile, gestionarea stării și interacțiunea cu API-ul backend;
- implementarea și înțelegerea design pattern-urilor, pentru a obține un cod flexibil, extensibil și ușor de întreținut;
- consolidarea cunoștințelor privind programarea orientată pe obiecte și principiile de proiectare software, precum separarea responsabilităților și respectarea principiului *Open/Closed*;
- implementarea funcționalităților de bază ale unei aplicații de bibliotecă digitală,
- integrarea cunoștințelor teoretice dobândite în cadrul cursului cu o implementare practică, complet documentată.

1.3 Cerintele proiectului

Platforma propusă are ca obiectiv dezvoltarea unei **aplicații web moderne pentru gestionarea unei biblioteci digitale**, realizată conform bunelor practici din domeniul dezvoltării aplicațiilor full stack. Sistemul este conceput pentru a oferi o experiență de utilizare intuitivă, stabilă și eficientă, atât pentru utilizatorii obișnuiți, cât și pentru administratorii aplicației.

Backend-ul aplicației va fi implementat utilizând cadrul de lucru **Spring Boot**, recunoscut pentru robustețea și flexibilitatea sa în gestionarea logicii de business și a accesului la date. Acesta va asigura funcționarea fluentă a operațiilor de tip **CRUD** asupra entităților principale ale aplicației, precum cărțile și exemplarele acestora.

Aplicația va trebui să permită afișarea și administrarea colecției de cărți disponibile în sistem, precum și efectuarea unor operații esențiale, cum ar fi adăugarea, editarea și ștergerea cărților de către utilizatorul cu rol de administrator. În același timp, utilizatorii obișnuiți vor avea posibilitatea de a interacționa cu sistemul prin operații precum vizualizarea, rezervarea, împrumutul ,returnarea exemplarelor de carte, în conformitate cu regulile de funcționare stabilite.

Frontend-ul aplicației va fi dezvoltat utilizând **React.js**, o bibliotecă JavaScript modernă, care va oferi o interfață de utilizator dinamică și ușor de utilizat. Utilizatorii vor putea naviga eficient prin lista de cărți, vor putea aplica filtre și sortări pentru organizarea informațiilor afișate și vor primi feedback clar pentru acțiunile efectuate.

Platforma va implementa un sistem de acces diferențiat, care va permite separarea funcționalităților în funcție de rolul utilizatorului. Utilizatorii obișnuiți vor beneficia de funcționalități orientate spre explorarea și utilizarea resurselor disponibile, în timp ce administratorii vor avea acces la funcționalități extinse pentru gestionarea conținutului și a datelor din sistem.

Prin adoptarea unei arhitecturi de tip **client–server** și prin integrarea tehnologiilor **Spring Boot** și **React.js**, aplicația va oferi o soluție scalabilă și ușor de întreținut. Comunicarea dintre frontend și backend va fi realizată prin intermediul unui **API REST**, utilizând formate standard de schimb de date.

În ansamblu, platforma va reprezenta o soluție coerentă și bine structurată pentru gestionarea unei biblioteci digitale, oferind funcționalități esențiale și o experiență de utilizare adaptată nevoilor utilizatorilor.

1.4 Arhitectura aplicatiei

Arhitectura proiectului este construită pe o abordare modernă de tip **client–server**, în care partea de frontend și cea de backend comunică între ele pentru a oferi o experiență de utilizare fluentă, coerentă și eficientă. Această separare permite

dezvoltarea independentă a celor două componente și asigură o bună scalabilitate a aplicației.

Pe partea de backend, aplicația este dezvoltată utilizând cadrul de lucru **Spring Boot**, fiind organizată conform arhitecturii **Model–View–Controller (MVC)**. Codul sursă este structurat în pachete distincte, fiecare având un rol bine definit. Pachetul de modele conține clasele Java care reprezintă entitățile aplicației, pachetul de repository include interfețele care extind JpaRepository și asigură accesul la baza de date, iar pachetul de controllere gestionează endpoint-urile REST prin care frontend-ul interacționează cu backend-ul.

Accesul la date este realizat prin intermediul **Spring Data JPA**, care facilitează maparea obiect–relațională și permite manipularea eficientă a datelor persistente. Pentru stocarea informațiilor, a fost utilizat sistemul de gestionare a bazelor de date relaționale **MySQL**, ales datorită fiabilității, performanței și suportului larg oferit pentru aplicațiile web. Conectarea la baza de date se realizează prin intermediul driverului JDBC.

Pe partea de frontend, aplicația este dezvoltată utilizând **React.js**, o bibliotecă JavaScript modernă, care permite construirea interfețelor de utilizator pe baza unei arhitecturi component-based. Această abordare facilitează reutilizarea componentelor, gestionarea eficientă a stării aplicației și dezvoltarea unei interfețe dinamice și interactive. Utilizatorii beneficiază astfel de o experiență de utilizare intuitivă și responsivă.

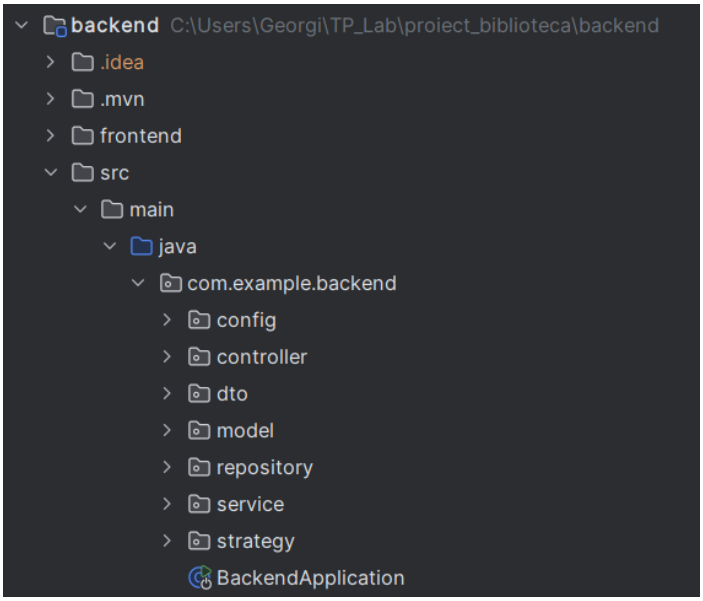
Comunicarea dintre frontend și backend se realizează prin intermediul unui **API REST**, utilizând formatul **JSON** pentru schimbul de date. Această abordare asigură o comunicare clară și standardizată între componentele aplicației.

Pentru gestionarea dependențelor și accelerarea procesului de dezvoltare, au fost utilizate biblioteci și tehnologii suplimentare precum **Spring Web**, **MySQL Driver**, care contribuie la reducerea codului boilerplate și la creșterea productivității în implementarea funcționalităților backend.

În ansamblu, arhitectura aleasă oferă o soluție robustă, scalabilă și ușor de întreținut, fiind potrivită pentru dezvoltarea unei aplicații web moderne de tip bibliotecă digitală.

1.5 Ierarhia structurala

În cadrul aplicației dezvoltate, clasele și interfețele au fost organizate în pachete distincte, cu scopul de a menține o ierarhie coerentă, clară și ușor de gestionat. Această structurare este esențială pentru organizarea logică a aplicației și pentru separarea responsabilităților între diferitele componente ale sistemului.



Pachetul de bază al aplicației este com.example.backend, care conține întreaga logică a aplicației backend. Acesta reprezintă nucleul aplicației și include toate subpachetele necesare pentru implementarea funcționalităților oferite de platforma de bibliotecă digitală.

Unul dintre cele mai importante subpachete este controller, care conține clasele responsabile de gestionarea cererilor HTTP primite din partea frontend-ului și de trimiterea răspunsurilor corespunzătoare. În acest pachet sunt definite clase precum BookController, BorrowController, AuthController, UserController, FavoriteController și RatingController, fiecare având rolul de a gestiona un anumit set de operații specifice aplicației, precum administrarea cărților, împrumuturile, autentificarea utilizatorilor sau gestionarea preferințelor.

Pachetul model conține clasele care reprezintă entitățile principale ale aplicației, mapate la structura bazei de date. Aceste clase definesc obiectele fundamentale ale sistemului, precum Book, User, Borrow, Favorite și Rating, precum și relațiile dintre acestea. Ele constituie baza asupra căreia sunt realizate operațiile de tip CRUD și logica de business.

Pentru accesul și manipularea datelor persistente, aplicația utilizează pachetul repository, care conține interfețele ce extind JpaRepository. Aceste interfețe, precum BookRepository, UserRepository, BorrowRepository, FavoriteRepository și RatingRepository, sunt responsabile de comunicarea cu baza de date și de realizarea operațiilor de bază asupra entităților, fără a fi necesară scrierea explicită a interogărilor SQL.

Logica de business a aplicației este concentrată în pachetul service, care include clase precum BookService, BorrowService și UserService. Aceste clase intermediază comunicarea dintre controllere și repository-uri și conțin regulile de

business necesare funcționării corecte a aplicației, contribuind la separarea clară a responsabilităților.

Un pachet distinct este dto, care conține clasele de tip **Data Transfer Object**, utilizate pentru transferul de date între frontend și backend. Aceste clase permit controlul informațiilor transmise și contribuie la securitatea și claritatea comunicării dintre componente.

Pachetul strategy este utilizat pentru organizarea logicii de sortare a cărților și conține interfața și clasele concrete care definesc diferitele strategii de sortare. Această structurare permite extinderea ușoară a funcționalităților de sortare și contribuie la un cod modular și ușor de întreținut.

De asemenea, pachetul config include clasele de configurare necesare aplicației, precum setările pentru securitate sau pentru comunicarea cu frontend-ul, asigurând funcționarea corectă a aplicației în diferite medii.

Clasa BackendApplication reprezintă punctul de intrare principal al aplicației Spring Boot, fiind responsabilă de inițializarea contextului aplicației și de pornirea acesteia.

În ansamblu, ierarhia de pachete și clase din aplicație oferă o structură bine definită și organizată, care facilitează dezvoltarea, testarea și întreținerea aplicației. Prin separarea clară a funcționalităților în module distincte, aplicația respectă bunele practici de proiectare software și asigură un cod curat, extensibil și ușor de gestionat.

1.6 Dependentele proiectului

Pentru implementarea aplicației backend a fost utilizat sistemul de management al dependențelor Maven, care permite gestionarea eficientă a bibliotecilor necesare și a ciclului de viață al aplicației. Configurarea dependențelor este realizată în fișierul pom.xml, acesta definind atât framework-urile principale utilizate, cât și bibliotecile auxiliare necesare dezvoltării și testării aplicației.

Proiectul are ca bază Spring Boot Starter Parent, care furnizează o configurație standardizată și versiuni compatibile pentru dependențele Spring, contribuind la stabilitatea și coerența aplicației. De asemenea, proiectul utilizează versiunea Java 17, asigurând acces la funcționalități moderne ale limbajului Java.

Pentru gestionarea persistenței datelor și a relațiilor dintre entități, a fost utilizată dependența Spring Boot Starter Data JPA. Aceasta facilitează maparea obiect–relațională prin intermediul JPA și Hibernate, permițând implementarea operațiilor de tip CRUD fără a fi necesară scrierea explicită a interogărilor SQL.

Dezvoltarea componentelor web ale aplicației este realizată cu ajutorul dependenței Spring Boot Starter Web, care oferă suport pentru construirea de aplicații

web și servicii REST. Aceasta include funcționalități esențiale pentru gestionarea cererilor HTTP, serializarea și deserializarea datelor, precum și expunerea endpoint-urilor REST utilizate de frontend.

Pentru a facilita procesul de dezvoltare și pentru a accelera ciclul de feedback, a fost inclusă dependența Spring Boot DevTools, care oferă funcționalități precum repornirea automată a aplicației la modificarea codului sursă. Această dependență este utilizată exclusiv în mediul de dezvoltare.

Conectarea aplicației la baza de date este realizată prin intermediul MySQL Connector/J, driverul oficial pentru comunicarea cu sistemul de gestionare a bazelor de date MySQL. Această dependență permite stabilirea conexiunii JDBC și executarea operațiilor asupra bazei de date.

Pentru reducerea codului boilerplate și creșterea productivității, a fost utilizată biblioteca Lombok. Aceasta permite generarea automată a metodelor uzuale, precum getter-ele, setter-ele și constructorii, contribuind la un cod mai curat și mai ușor de întreținut. Lombok este configurat și ca procesor de adnotări în cadrul proiectului.

În scopul testării aplicației, a fost inclusă dependența Spring Boot Starter Test, care oferă suport pentru testarea unităților și integrarea acestora, incluzând framework-uri precum JUnit și Mockito. Această dependență este utilizată exclusiv în contextul testelor.

În ansamblu, dependențele alese asigură o bază solidă pentru dezvoltarea unei aplicații backend robuste, scalabile și ușor de întreținut, oferind suport complet pentru dezvoltare, persistență, comunicare web și testare.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.5.7</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.example</groupId>
  <artifactId>backend</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>backend</name>
```

```
<description>Demo project for Spring Boot</description>
<url/>
<licenses>
  <license/>
</licenses>
<developers>
  <developer/>
</developers>
<scm>
  <connection/>
  <developerConnection/>
  <tag/>
  <url/>
</scm>
<properties>
  <java.version>17</java.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <version>8.0.33</version>
  </dependency>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
```

```
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <annotationProcessorPaths>
            <path>
              <groupId>org.projectlombok</groupId>
              <artifactId>lombok</artifactId>
            </path>
          </annotationProcessorPaths>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <configuration>
          <excludes>
            <exclude>
              <groupId>org.projectlombok</groupId>
              <artifactId>lombok</artifactId>
            </exclude>
          </excludes>
        </configuration>
      </plugin>
    </plugins>
  </build>

</project>
```

2.Cerinte functionale

2.1 AddBook

Funcționalitatea de adăugare a unei cărți este destinată utilizatorilor cu rol de administrator și permite introducerea de noi cărți în sistem. Aceasta este implementată la nivel de backend prin intermediul unui endpoint REST definit în cadrul clasei BookController.

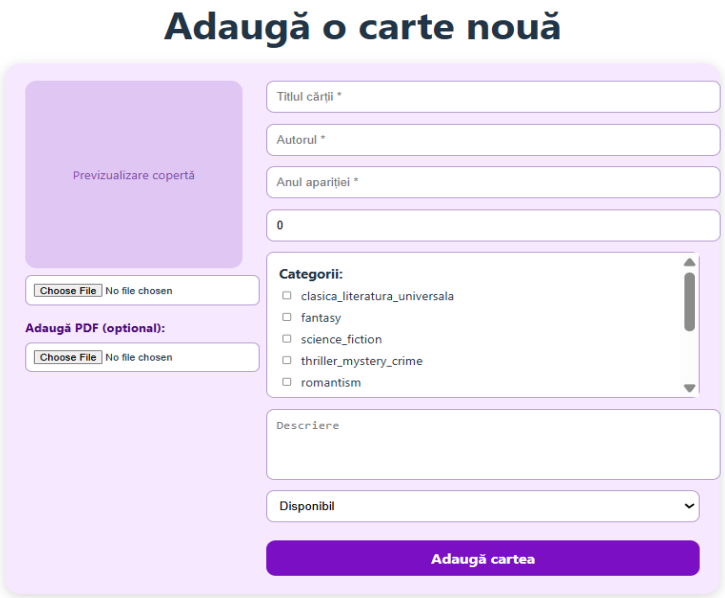
Metoda principală care gestionează această operație este addBook, care primește datele cărții sub forma unui obiect de tip AddBookRequest, precum și fișiere opționale de tip imagine și document PDF. Comunicarea dintre frontend și backend se realizează folosind o cerere HTTP de tip POST, cu conținut de tip multipart/form-data.

În cadrul metodei, aplicația verifică existența fișierelor încărcate și, dacă acestea sunt prezente, le salvează în sistemul de fișiere al aplicației. Ulterior, este creat un obiect de tip Book, ale cărui câmpuri sunt populate cu datele primite de la interfața web.

După crearea obiectului, cartea este salvată în baza de date prin apelul metodei save din cadrul BookRepository. În cazul în care operația se finalizează cu succes, backend-ul returnează un mesaj de confirmare către frontend, indicând faptul că adăugarea cărții a fost realizată cu succes.

În situația apariției unor erori pe parcursul procesului (de exemplu, probleme la salvarea fișierelor sau a datelor), aplicația returnează un mesaj de eroare corespunzător.

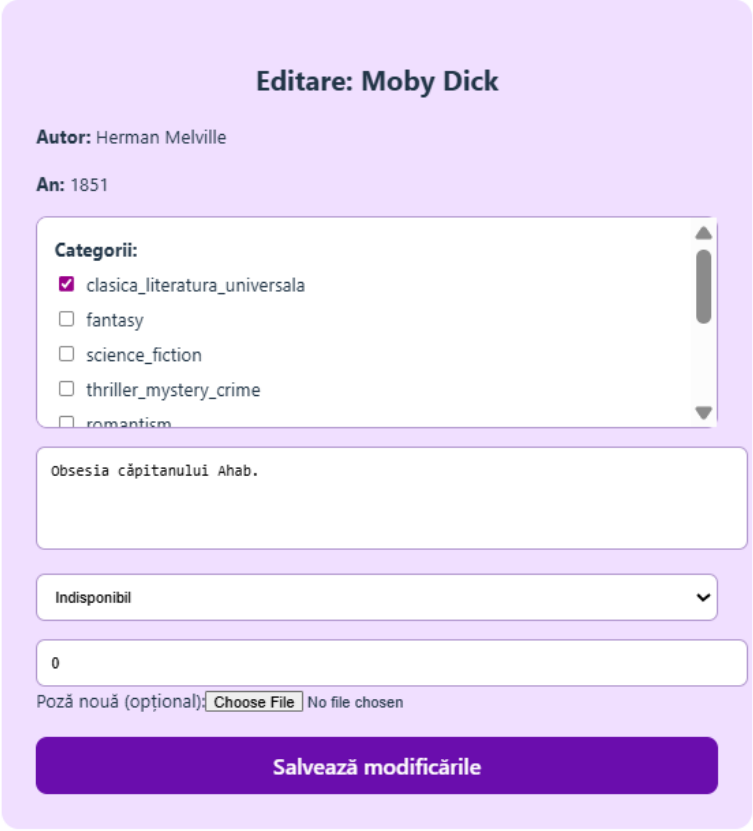
Prin această funcționalitate, sistemul asigură administrarea eficientă a colecției de cărți și menținerea actualizată a bazei de date.



2.2 EditBook

Aplicația trebuie să permită utilizatorilor cu rol de administrator editarea informațiilor unei cărți existente în sistem. Administratorul poate accesa funcționalitatea de editare dintr-un tabel care afișează lista de cărți, prin selectarea opțiunii „Editează” asociată unei cărți.

La selectarea acestei opțiuni, sistemul afișează un formular de editare precompletat cu datele existente ale cărții. Administratorul poate modifica informații precum descrierea, categoriile, stocul disponibil, statusul cărții, precum si imaginea.



După efectuarea modificărilor, administratorul finalizează operația prin apăsarea butonului „Salvare modificări”. Sistemul actualizează informațiile cărții și afișează un mesaj de confirmare în cazul în care operația este realizată cu succes.

In cazul apariției unor erori, sistemul va afișa un mesaj de eroare.

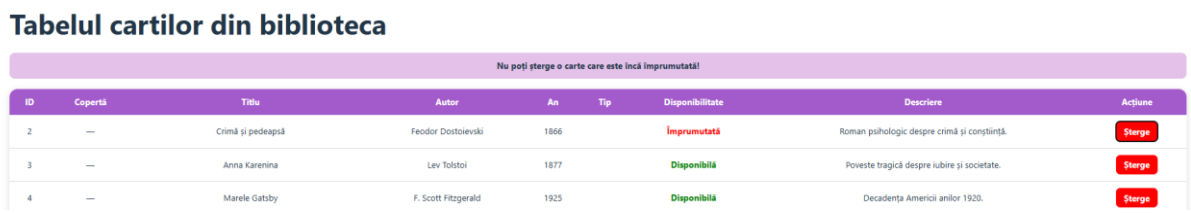
Prin această funcționalitate, aplicația asigură menținerea corectă și actualizată a informațiilor despre cărți, contribuind la o administrare eficientă a conținutului.

2.3 DeleteBook

Aplicația trebuie să permită utilizatorilor cu rol de **administrator** ștergerea unei cărți existente din sistem. Această funcționalitate este destinată gestionării eficiente a colecției de cărți și eliminării titlurilor care nu mai sunt relevante sau disponibile.

Administratorul poate iniția operația de ștergere din lista de cărți afișată în interfața web, prin selectarea opțiunii corespunzătoare pentru cartea dorită. Înainte de efectuarea ștergerii, sistemul verifică dacă respectiva carte există și dacă aceasta nu este asociată unui împrumut activ.

În cazul în care cartea este în prezent împrumutată, aplicația va bloca operația de ștergere și va afișa un mesaj de informare către administrator, indicând faptul că ștergerea nu este permisă. Dacă nu există împrumuturi active asociate cărții, sistemul va elimina cartea din baza de date și va actualiza informațiile corespunzătoare.



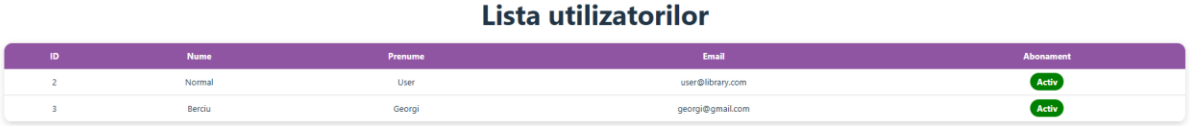
La finalizarea cu succes a operației, aplicația va afișa un mesaj de confirmare. În situația apariției unor erori, administratorul va fi informat printr-un mesaj corespunzător.

Prin această funcționalitate, aplicația asigură integritatea datelor și previne situațiile inconsistente, precum ștergerea unei cărți care este încă utilizată în cadrul sistemului.

2.4 ViewUserList

Aplicația trebuie să permită utilizatorilor cu rol de administrator vizualizarea listei utilizatorilor înregistrați în sistem. Această funcționalitate oferă administratorului posibilitatea de a monitoriza și gestiona utilizatorii aplicației.

Lista afișată conține informații relevante despre utilizatori și este disponibilă într-o interfață dedicată.



2.5 Adaugare la favorite

Aplicația trebuie să permită utilizatorilor autentificați adăugarea unei cărți în lista personală de favorite. Această funcționalitate oferă utilizatorilor posibilitatea de a marca rapid cărțile de interes, pentru a le putea accesa ulterior cu ușurință.

Utilizatorul poate adăuga o carte la favorite din lista de cărți afișată în aplicație, prin selectarea inimioarei,operatia inversa acesteia realizandu-se prin deselectarea inimioarei rosii. La finalizarea operației, sistemul va salva cartea în lista de favorite a utilizatorului și va oferi un feedback vizual corespunzător.

2.6 Afisare cele mai citite carti

Aplicația trebuie să ofere utilizatorilor acces la o pagină specială care afișează cele mai citite cărți din sistem. Această funcționalitate are rolul de a evidenția cărțile populare, pe baza numărului de împrumuturi efectuate pentru fiecare titlu.

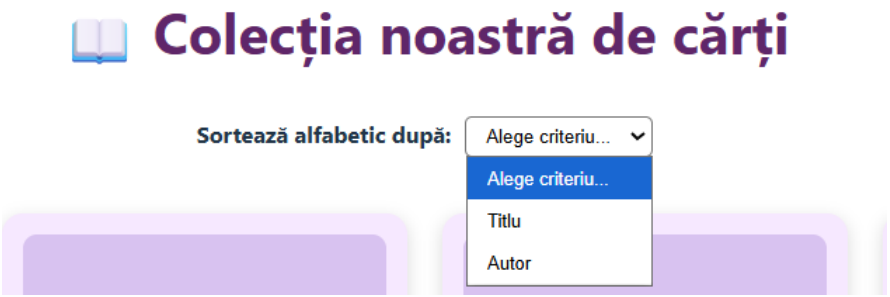
Pagina dedicată va afișa lista cărților ordonate descrescător în funcție de frecvența împrumuturilor, permițând utilizatorilor să descopere rapid titlurile cele mai apreciate sau frecvent accesate.

Utilizatorii pot accesa această pagină din interfața principală a aplicației, iar sistemul va furniza informații relevante despre fiecare carte afișată.

Prin această funcționalitate, aplicația îmbunătățește experiența de utilizare și facilitează explorarea colecției de cărți în funcție de popularitate.

2.7 Sortare alfabetica dupa criteriu

Aplicația trebuie să permită utilizatorilor sortarea listei de cărți afișate în interfața web, în funcție de un criteriu selectat. Pentru această funcționalitate, utilizatorul poate alege criteriul de sortare dintr-un element de tip dropdown, disponibil pe pagina principală a aplicației.



Utilizatorul are posibilitatea de a selecta sortarea alfabetică a cărților după titlu sau după autor, iar lista de cărți este actualizată automat în funcție de opțiunea aleasă. Această funcționalitate facilitează navigarea și identificarea rapidă a cărților dorite.

Sortarea se aplică asupra listei complete de cărți disponibile și nu modifică datele persistente din sistem, ci doar modul de afișare al acestora.

Prin această funcționalitate, aplicația oferă o experiență de utilizare îmbunătățită, permițând organizarea eficientă a informațiilor afișate.

2.8 Vizualizare categorii carti

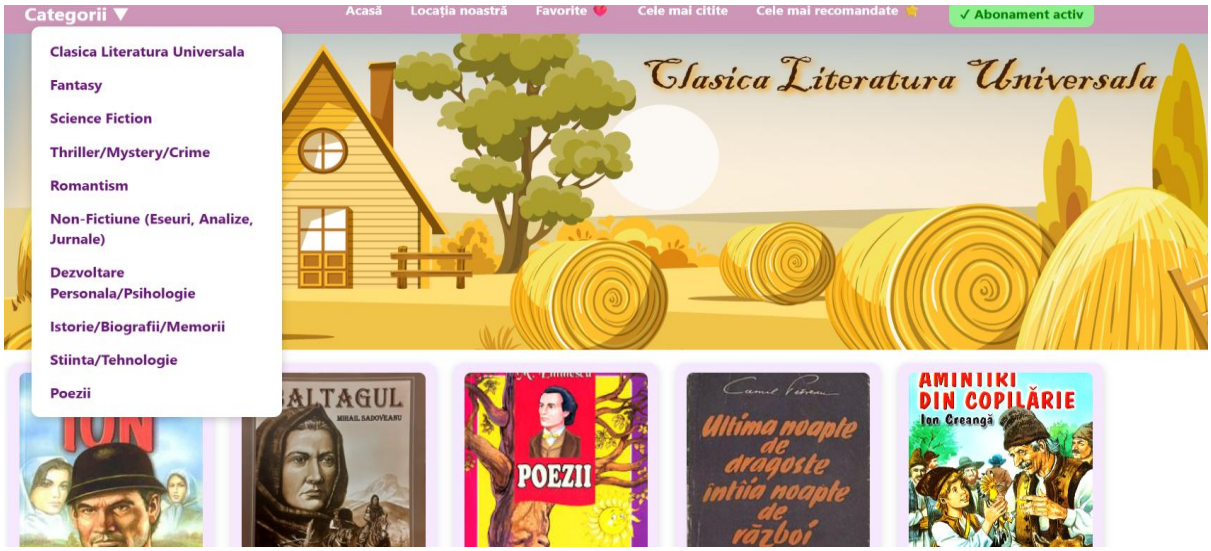
Funcționalitatea de vizualizare a categoriilor permite utilizatorilor să exploreze resursele bibliotecii digitale într-un mod organizat, filtrând conținutul în funcție de genul

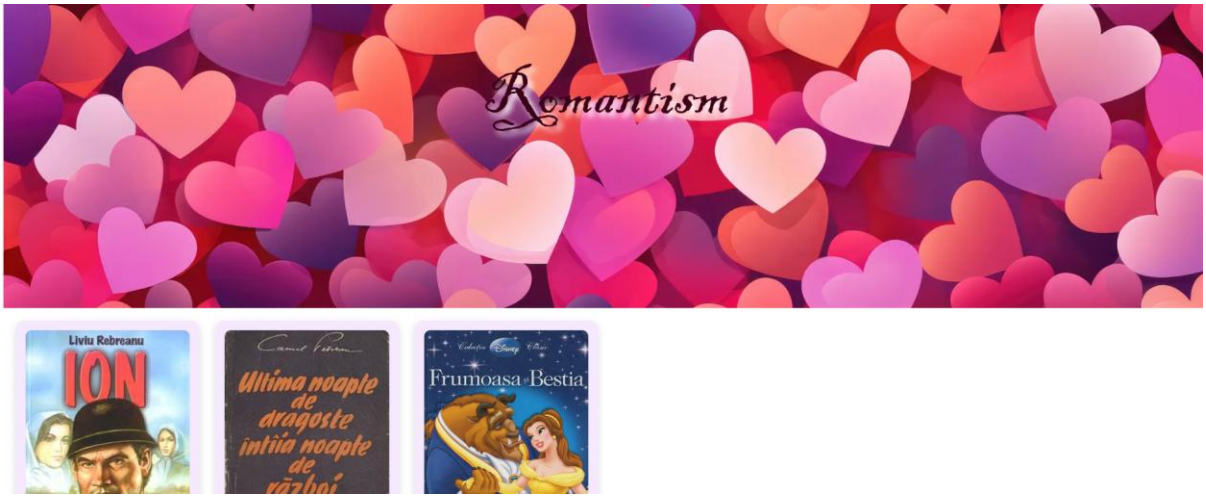
1. Implementarea la nivel de Backend

- **Endpoint dedicat**: În BookController, este definită o metodă de tip GET mapată la adresa /books/category/{category} care primește ca parametru numele categoriei selectate.
- **Logica de acces la date**: Controller-ul apelează metoda findByCategory din bookRepository pentru a returna din baza de date MySQL doar acele obiecte de tip Book care fac parte din categoria respectivă.
- **Schimbul de date**: Rezultatul este transmis către frontend sub formă de listă (JSON), asigurând o comunicare standardizată între componente.

2. Implementarea la nivel de Frontend

- **Interfață Dinamică**: Pagina CategoryPage utilizează hook-ul useParams din React pentru a identifica categoria dorită direct din URL.
- **Personalizare Vizuală**: Pentru a oferi o experiență de utilizare intuitivă, fiecare categorie beneficiază de stilizare proprie:
    - **Imagini de fundal tematice**: Maparea categoryImages afișează imagini specifice (ex: fantasy.jpg, science.jpg) adaptate vizual pentru fiecare gen.
    - **Stilizarea textului**: Titlurile categoriilor sunt formatate folosind fontul "Blackadder" și umbre de text (textShadow) configurate în categoryTextStyles pentru a reflecta atmosfera categoriei (ex: culori închise pentru Mystery, culori aurii pentru Literatură Clasică).



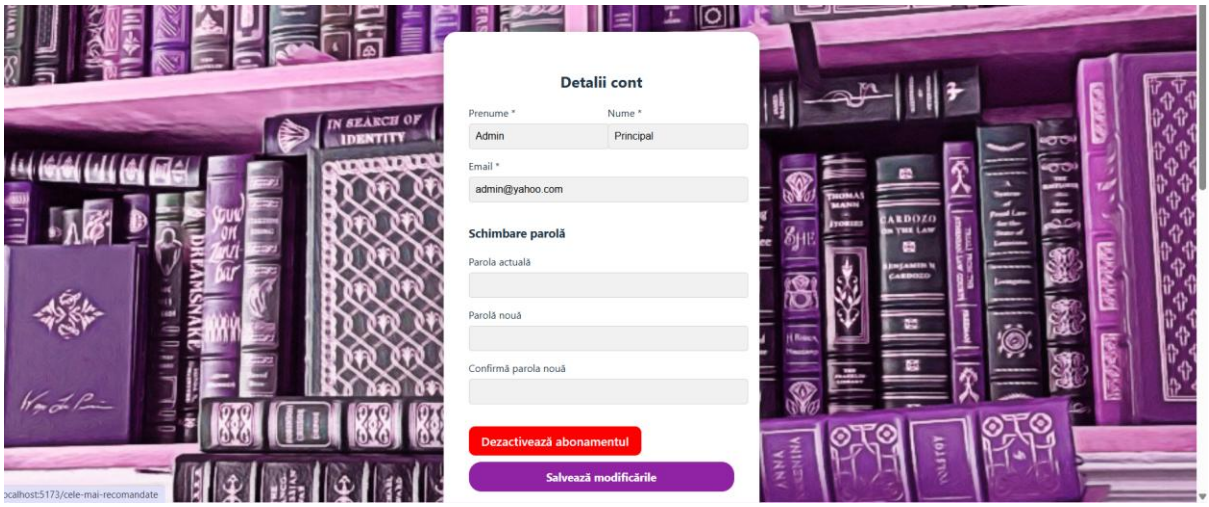


2.9 Vizualizare detalii cont + modificare

Această secțiune permite utilizatorului să vizualizeze și să își modifice detaliile contului personal într-o interfață intuitivă. Funcționalitățile principale includ:

1. **Vizualizarea informațiilor contului**
    - Utilizatorul poate vedea prenumele, numele, adresa de email și starea abonamentului curent (active, inactive, pending).
    - Datele sunt preluate din localStorage la încărcarea paginii și sincronizate automat cu eventualele modificări globale ale contului.
2. **Editarea detaliilor personale**
    - Utilizatorul poate modifica prenumele, numele și adresa de email.
    - Modificările sunt validate și trimise către backend printr-un PUT request, actualizând și datele din localStorage.
3. **Schimbarea parolei**
    - Utilizatorul poate introduce parola actuală și poate seta o parolă nouă, cu confirmare.
    - Input-ul de confirmare are feedback vizual care indică dacă parolele coincid.
4. **Gestionarea abonamentului**
    - Dacă abonamentul este activ, utilizatorul poate să îl dezactiveze printr-un buton dedicat.
    - Starea abonamentului se afișează vizual prin mesaje diferite pentru stările active, inactive și pending.

5. **Feedback vizual și UX**
    - Modificările reușite sunt confirmate printr-un mesaj alert.
    - Event-ul global userChanged permite sincronizarea datelor contului între mai multe componente.

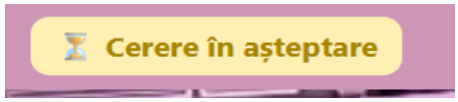


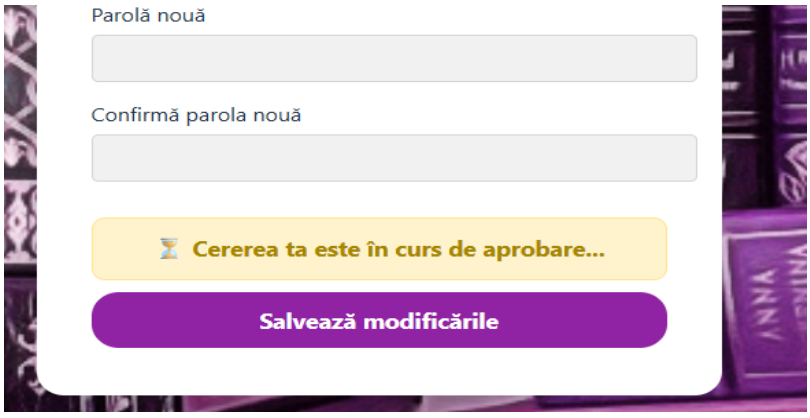
2.10 Cerere activare abonament de catre utilizatori + dezactivare abonament

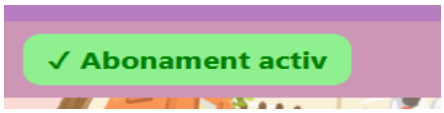
Această secțiune permite utilizatorilor să solicite activarea abonamentului și să gestioneze starea acestuia, oferind o interfață intuitivă și feedback vizual.

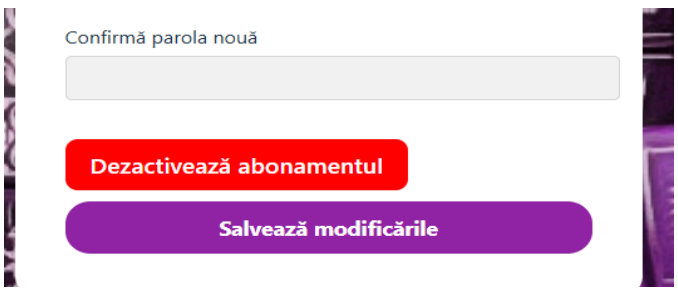
1. **Verificarea autentificării și a rolului**
    - Dacă utilizatorul nu este autentificat, aplicația afișează un mesaj corespunzător și nu permite activarea abonamentului.
    - Dacă utilizatorul este administrator (MANAGER), abonamentul este considerat activ implicit, iar opțiunea de activare nu este disponibilă.
2. **Cerere de activare a abonamentului**
    - Utilizatorul poate solicita activarea abonamentului printr-un click pe butonul „Da, doresc activarea abonamentului”.
    - Cererea este trimisă către backend (POST /api/request-subscription/{userId}) și starea abonamentului în frontend se schimbă în pending.
    - Evenimentele globale userChanged și pendingUpdated sunt declanșate pentru a sincroniza starea abonamentului în întreaga aplicație.

3. **Feedback vizual și navigare**
    - După cererea de activare, utilizatorul primește un mesaj de confirmare („Mulțumim! Așteaptă confirmarea administratorului.”).
    - Aplicația navighează automat către pagina principală după câteva secunde.
    - Există și opțiunea „Mă mai gândesc” care permite revenirea la pagina principală fără a trimite cererea.
4. **Gestionarea abonamentului**
    - Starea abonamentului (pending, active, inactive) este sincronizată cu localStorage și poate fi monitorizată prin evenimente globale, permițând componentei AccountDetails și altor componente să afișeze corect statusul abonamentului.





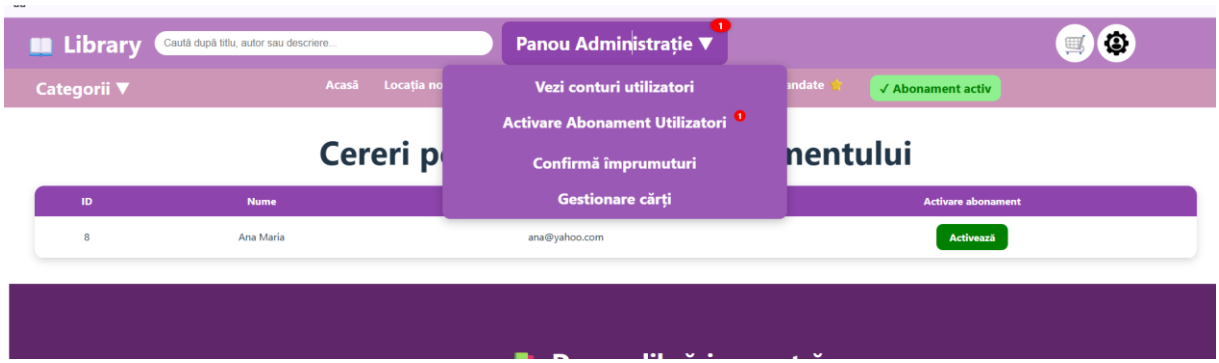




2.11 Confirmare abonament utilizatori de catre administrator

Această funcționalitate este dedicată utilizatorului cu rol de **Administrator** și are scopul de a valida accesul utilizatorilor la serviciile premium ale bibliotecii prin activarea manuală a abonamentelor.

Sistemul funcționează pe baza unui flux de aprobare. Atunci când un utilizator solicită un abonament, starea acestuia devine „pending” (în așteptare). Administratorul accesează o interfață dedicată unde poate vizualiza lista tuturor cererilor neprocesate și le poate aproba individual.



2.12 Istoric imprumuturi

Această funcționalitate permite utilizatorilor să monitorizeze activitatea lor în cadrul bibliotecii, oferind o evidență clară a cărților rezervate, împrumutate în prezent sau returnate în trecut.

1. Descrierea Interfeței

Pagina „Istoric împrumuturi” este concepută ca un tablou de bord personal unde fiecare înregistrare include detalii vizuale și informații temporale critice:

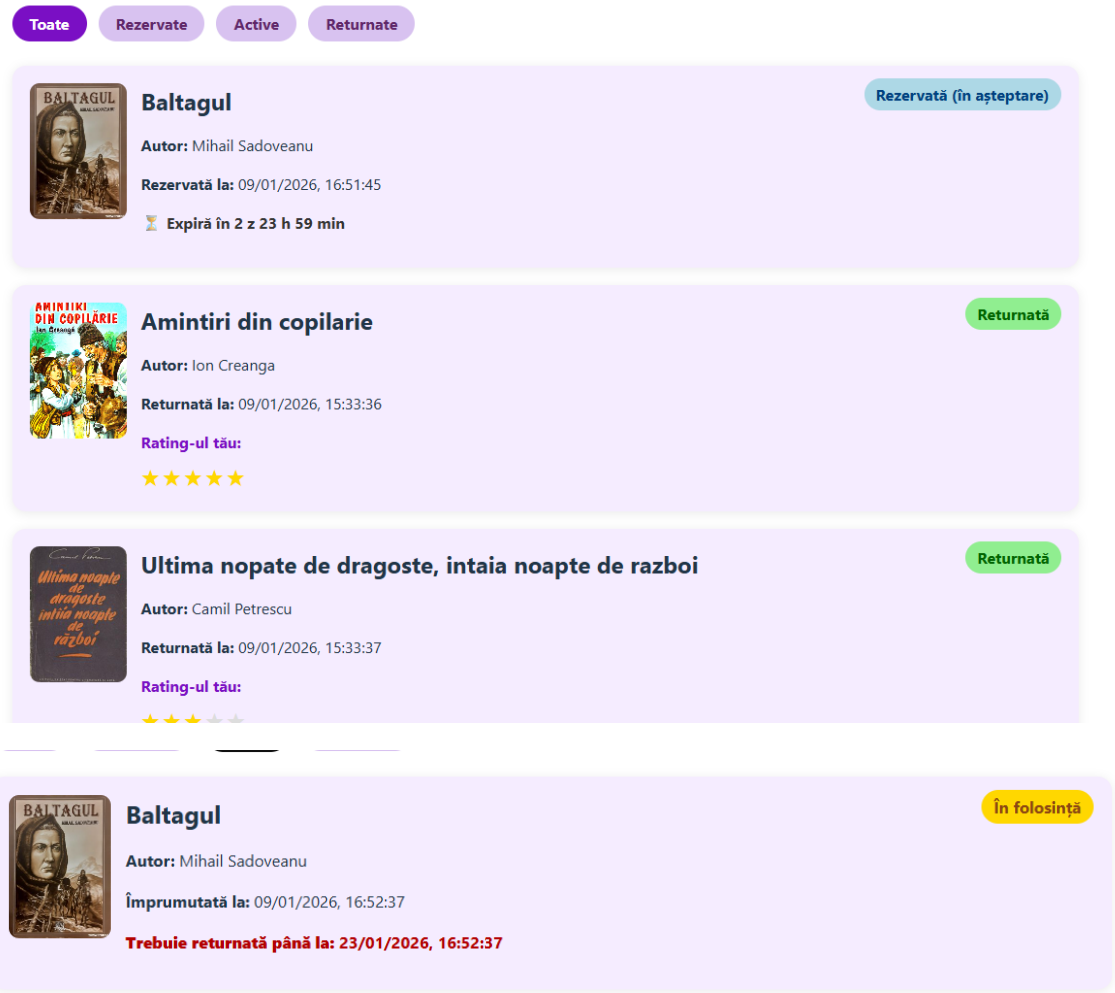
- **Identificarea Vizuală**: Fiecare rând afișează coperta cărții, titlul și autorul.
- **Filtrarea Datelor**: Utilizatorul poate organiza lista rapid folosind butoane de filtru pentru: „Toate”, „Rezervate” (în așteptarea ridicării), „Active” (cărți deținute în prezent) și „Returnate”.

2. Managementul Stărilor (Status)

Aplicația utilizează un sistem vizual de etichetare (badges) pentru a diferenția etapele unui împrumut:

- **Rezervată (Pending)**: Marcată cu albastru. Include un **cronometru în timp real** care calculează timpul rămas până la expirarea rezervării (getRemainingTime). Dacă timpul expiră, sistemul avertizează vizual utilizatorul.

- **În folosință (Active)**: Marcată cu galben/auriu. Afișează clar data limită de returnare (dueDate) cu un accent roșu pentru a preveni penalizările.
- **Returnată (Returned)**: Marcată cu verde. Reprezintă arhiva împrumuturilor finalizate.



2.13 Returnare carti

Această secțiune permite utilizatorilor să returneze cărțile împrumutate și să ofere feedback sub formă de rating. Funcționalitățile principale includ:

1. **Preluarea cărților active**
    - Aplicația face fetch către backend (/api/borrows/user/{userId}) și filtrează împrumuturile cu status active sau reserved.
    - Se afișează lista cărților cu titlu, autor, imagine și data împrumutului.

2. **Afișarea deadline-ului și starea împrumutului**
    - Calculul zilelor rămase până la returnare:
        - Dacă utilizatorul depășește termenul, se afișează avertizare roșie.
        - Dacă termenul se apropie (≤3 zile), se afișează mesaj portocaliu.
        - În rest, se afișează un mesaj verde cu zilele rămase.
3. **Feedback prin rating**
    - Utilizatorul poate acorda un rating cărții (1-5 stele) înainte de a returna cartea.
    - Rating-ul se afișează vizual cu stele și descriere text.

2.14 Imprumut carti

Această secțiune permite utilizatorilor să selecteze cărți pentru împrumut și să le rezerve. Funcționalitățile principale includ:

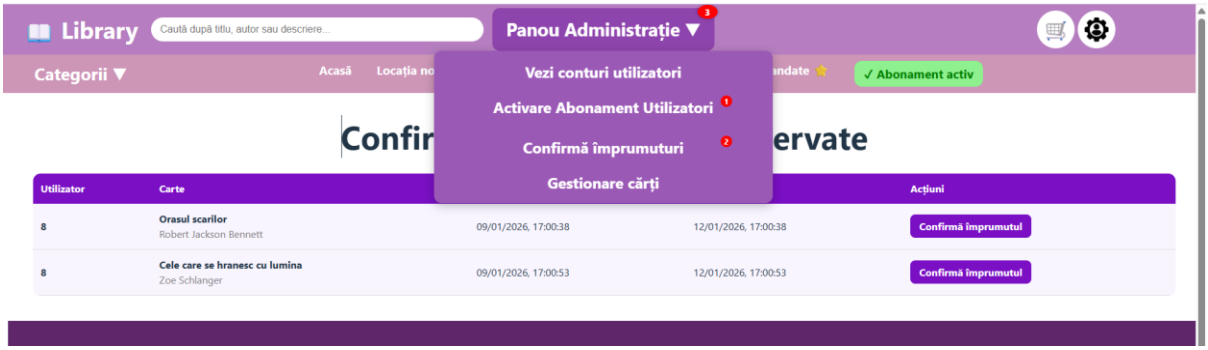
1. **Gestionarea coșului de împrumuturi**
    - Cărțile adăugate de utilizator se încarcă din localStorage.
    - Utilizatorul poate selecta ce cărți vrea să împrumute și poate elimina cărțile din coș.
2. **Selectarea cărților pentru împrumut**
    - Fiecare carte are un checkbox pentru a fi inclusă în rezervare.
    - Informațiile cărții (titlu, autor, an, tip, descriere și imagine) sunt afișate clar.

3. **Confirmarea rezervării**
    - După selectarea cărților, utilizatorul poate confirma rezervarea.
    - După rezervare:
        - Coșul se golește (localStorage actualizat).
        - Evenimentele cartChanged și reservationsChanged sunt declanșate pentru actualizarea altor componente.
        - Utilizatorul primește un mesaj de confirmare: „Ai 3 zile să ridici cartea”.

2.15 Confirmare imprumut carti de catre administrator

Această secțiune permite administratorului să gestioneze și să confirme împrumuturile rezervate de utilizatori. Funcționalitățile principale includ:

1. **Încărcarea rezervărilor în așteptare**
    - Se face un fetch către backend (/api/borrows/pending) pentru a prelua toate rezervările neconfirmate.
    - Rezervările afișează informații precum: utilizator, titlu carte, autor, data rezervării și data expirării.
2. **Confirmarea împrumuturilor**
    - Administratorul poate confirma orice rezervare folosind butonul „Confirmă împrumutul”.

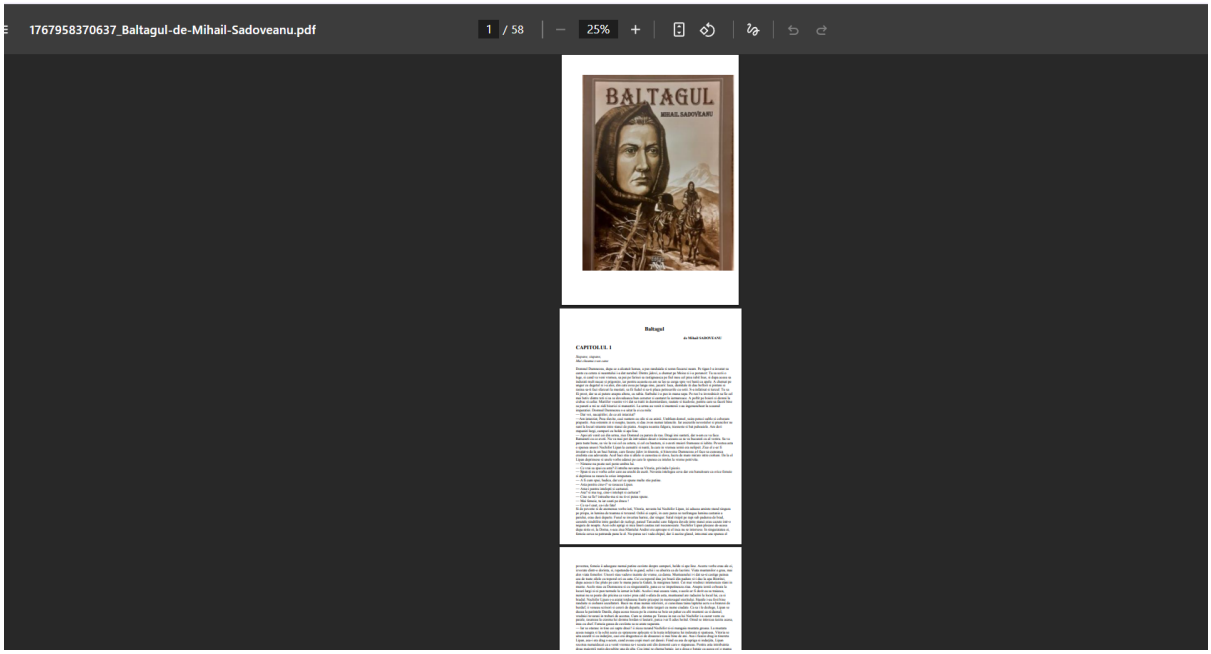


2.16 Posibilitatea de a citi carti online ca pdf

Această secțiune permite utilizatorilor să citească cărți direct în aplicație, fără a fi nevoie să mearga sa le ridice din magazin.

Funcționalități principale:

1. **Afișarea cărților disponibile pentru citire**
    - Lista cărților care pot fi citite online este preluată din backend.
    - Fiecare carte afișează titlu, autor și opțiunea de a deschide PDF-ul printr-un buton sub poza cartii.
2. **Control acces**
    - Funcționalitatea este disponibilă doar pentru utilizatorii autentificați care au abonament activ.
    - Dacă utilizatorul nu are acces la carte, butonul de citire este dezactivat sau afișează un mesaj informativ.



2.17 Cele mai recomandate

Aplicația pune la dispoziția utilizatorilor o secțiune dedicată **„Cele mai recomandate cărți”**, care afișează volumele cu cele mai bune evaluări primite din partea cititorilor.

Lista este generată **dinamic**, pe baza rating-urilor acordate de utilizatori, fiind preluate din backend doar cărțile care au primit cel puțin un vot. Pentru fiecare carte sunt afișate următoarele informații:

- titlul și autorul;
- anul apariției;
- coperta cărții;
- ratingul mediu (reprezentat vizual prin stele);
- numărul total de voturi;
- statusul cărții (disponibilă / indisponibilă).

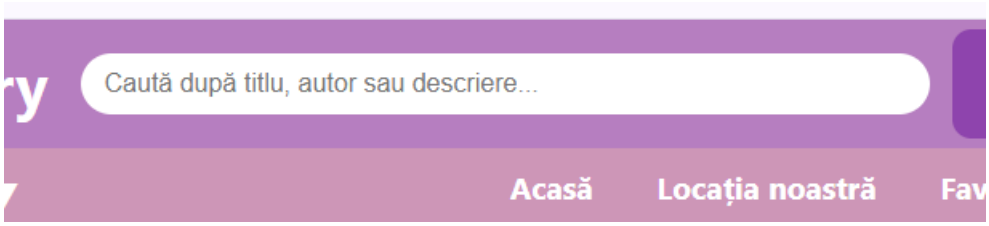
Utilizatorii pot:

- accesa **pagina de detalii a unei cărți** printr-un click pe copertă sau titlu;
- adăuga cartea în **coșul de împrumut**, dacă aceasta este disponibilă;
- marca o carte ca **favorită**, funcționalitate disponibilă doar utilizatorilor autentificați.

2.18 Cautare dupa titlu, autor sau descriere

Aplicația oferă utilizatorilor posibilitatea de a căuta cărți în bibliotecă folosind un câmp de căutare disponibil în bara de navigare. Căutarea se realizează după **titlu, autor sau descriere**, permițând identificarea rapidă a cărților dorite.

După introducerea termenului de căutare și apăsarea tastei *Enter*, utilizatorul este redirecționat către pagina de rezultate, unde sunt afișate toate cărțile care corespund criteriului introdus. Căutarea este realizată prin intermediul unui apel către backend, care filtrează cărțile în funcție de textul introdus.



2.19 Vizualizare termeni si conditii

Aplicația pune la dispoziția utilizatorilor o pagină dedicată pentru vizualizarea **Termenilor și Condițiilor**, accesibilă din interfața web. Această pagină are rolul de a informa utilizatorii cu privire la regulile de utilizare a platformei, drepturile și obligațiile părților, precum și aspectele legale legate de serviciile oferite.

Pagina este implementată ca o componentă React separată (*TermsAndConditions*), care afișează conținutul într-un mod structurat și ușor de parcurs, împărțit pe secțiuni tematice (introducere, utilizarea contului, servicii de împrumut, drepturi de autor, protecția datelor, limitarea răspunderii etc.).

La încărcarea paginii, titlul documentului este actualizat dinamic folosind hook-ul **useEffect**, pentru a reflecta corect secțiunea curentă (*„Termeni și Condiții | MyLibrary”*), contribuind astfel la o experiență de navigare mai clară și profesională.

Conținutul este afișat într-un layout responsive, cu accent pe lizibilitate și structurare vizuală, folosind stiluri inline pentru evidențierea titlurilor, paragrafelor și listelor. De asemenea, pagina include informații de contact și o secțiune de confirmare prin care utilizatorul este informat că utilizarea platformei implică acceptarea termenilor afișați.

📋 Termeni și Condiții

Ultima actualizare: Decembrie 2024

1. Introducere

Prezentul document („Termenii și Condițiile") stabilește condițiile contractuale aplicabile utilizării platformei MyLibrary, administrată de MyLibrary SRL, cu sediul în Cluj-Napoca, România. Accesarea sau utilizarea Platformei constituie acceptarea completă și necondiționată a acestor Termeni și Condiții.

2. Crearea și utilizarea contului

2.1. Pentru utilizarea serviciilor este necesară crearea unui Cont, folosind informații reale și complete.

2.2. Utilizatorul este responsabil pentru păstrarea confidențialității datelor de autentificare.

2.3. MyLibrary poate suspenda sau închide orice cont care încalcă prezentul document sau este folosit abuziv.

3. Serviciile de împrumut

3.1. Durata standard: 14 zile calendaristice.

3.2. Prelungirea: posibilă o singură dată, pentru încă 7 zile, dacă Produsul nu este rezervat de alt Membru.

2.20 Locatia bibliotecii

Aplicația MyLibrary pune la dispoziția utilizatorilor o pagină dedicată pentru vizualizarea **locației fizice a bibliotecii**, oferind informații clare și detaliate pentru facilitarea accesului.

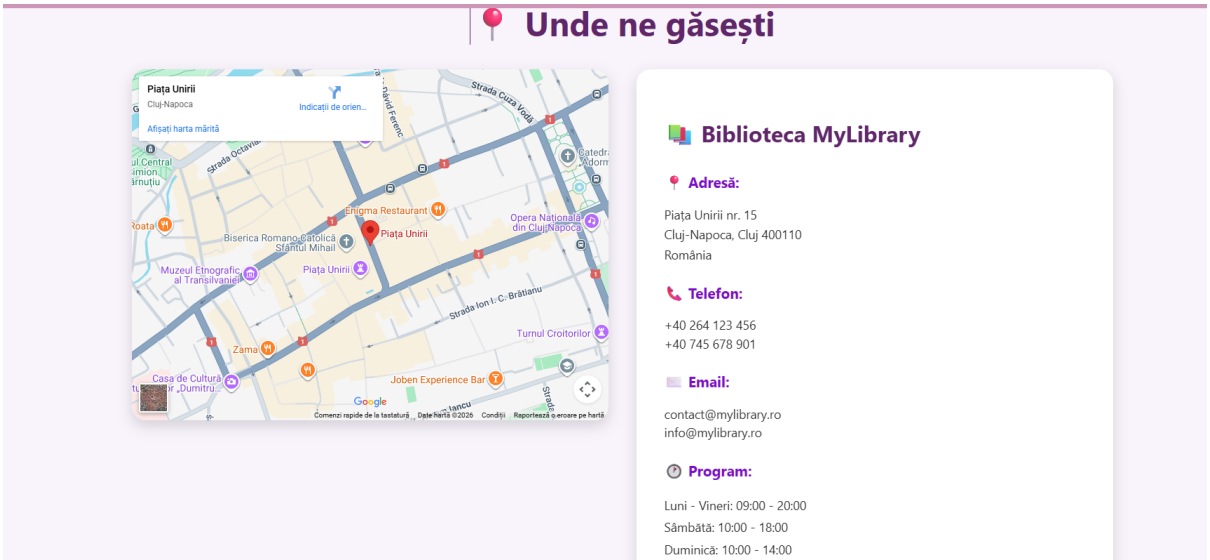
Funcționalitatea este implementată prin componenta React *LibraryLocation*, care afișează locația bibliotecii pe o hartă interactivă Google Maps, integrată prin intermediul unui element de tip *iframe*. Harta este configurată pentru adresa bibliotecii din Piața Unirii, Cluj-Napoca, permițând utilizatorilor să identifice rapid poziția exactă a clădirii.

La încărcarea paginii, titlul documentului este actualizat dinamic folosind hook-ul **useEffect**, pentru a reflecta secțiunea curentă (*„Locația noastră | MyLibrary”*), contribuind la o navigare mai intuitivă și coerentă.

Pe lângă hartă, pagina oferă informații esențiale precum:

- adresa completă a bibliotecii;
- datele de contact (telefon și email);
- programul de funcționare;
- indicații privind modalitățile de acces (cu mașina, transport public sau pe jos).

De asemenea, este inclusă o secțiune dedicată facilităților disponibile în incinta bibliotecii, precum acces pentru persoane cu dizabilități, WiFi gratuit, zone de lectură, spații de studiu și alte servicii utile pentru cititori.



3.Design Pattern

3.1.Strategy Pattern – Berciu Oana – Georgiana

Motivare alegere:

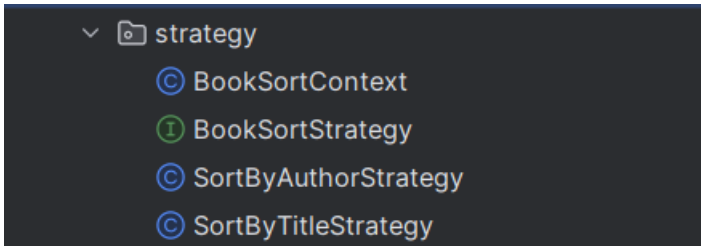
Pentru a îmbunătăți experiența utilizatorului pe pagina Home, a fost necesară introducerea unei funcționalități de sortare a cărților afișate. Într-o aplicație care gestionează un număr mare de titluri, afișarea unei liste statice de cărți poate îngreuna identificarea rapidă a informațiilor dorite.

Prin urmare, utilizatorului i se oferă posibilitatea de a ordona lista de cărți în funcție de criterii relevante, precum titlul sau autorul, direct din interfața web, folosind un element de tip *dropdown*. Această funcționalitate permite o navigare mai intuitivă și o accesare mai rapidă a conținutului dorit, fără a fi necesară reîncărcarea manuală a paginii sau aplicarea unor filtre complexe.

Mai mult, sortarea este realizată dinamic, în funcție de opțiunea selectată de utilizator, ceea ce oferă flexibilitate și personalizare a modului de afișare a informațiilor. Astfel, aplicația răspunde mai bine nevoilor utilizatorilor și oferă o interfață mai prietenoasă și mai ușor de utilizat.

Organizare cod:

Pentru a asigura o structură clară și o bună separare a responsabilităților, toate clasele implicate în sortare au fost grupate într-un pachet dedicat: strategy



Această organizare facilitează atât mentenanța codului, cât și extinderea ulterioară a funcționalității cu noi criterii de sortare.

Implementare:

Primul pas a fost definirea unei interfețe comune, BookSortStrategy, care stabilește contractul pentru toate strategiile de sortare. Interfața conține o singură metodă, sort, care primește o listă de cărți și returnează lista sortată.

Această abordare permite ca toate strategiile concrete să fie tratate uniform, indiferent de modul în care realizează sortarea.

Pentru fiecare criteriu de sortare a fost creată o clasă separată:

- SortByTitleStrategy – responsabilă de sortarea cărților după titlu
- SortByAuthorStrategy – responsabilă de sortarea cărților după autor

Fiecare clasă implementează interfața BookSortStrategy și conține logica specifică algoritmului de sortare. Sortarea este realizată folosind API-ul Stream din Java, ceea ce asigură un cod concis, lizibil și eficient.

Prin separarea algoritmilor în clase distincte, aplicația respectă principiul **Single Responsibility**, fiecare clasă având un rol bine definit.

Clasa BookSortContext reprezintă **contextul** din Strategy Pattern și are rolul de a selecta strategia de sortare potrivită în funcție de opțiunea aleasă de utilizator.

Aceasta primește, prin mecanismul de *dependency injection* oferit de Spring, un map de strategii disponibile, asociate unor chei (sortByTitle, sortByAuthor). În momentul în care este apelată metoda executeStrategy, contextul:

- identifică strategia corespunzătoare criteriului primit;
- verifică validitatea acesteia;
- deleagă efectiv sortarea către strategia selectată.

Contextul nu cunoaște detalii despre modul de implementare al fiecărei strategii, ceea ce duce la un nivel ridicat de decuplare între componente.

În BookController, funcționalitatea de sortare este integrată într-un mod simplu și clar. Controller-ul obține lista completă de cărți din baza de date folosind metoda

findAll(), apoi apelează BookSortContext, trimițând lista de cărți împreună cu strategia de sortare selectată. În final, primește lista sortată și o returnează către frontend.

Astfel, controller-ul rămâne responsabil doar de coordonarea fluxului de date, fără a conține logică de sortare propriu-zisă.

3.2. Observer pattern – Szarka Francesca Lara

Motivare alegere:

În cadrul aplicației, a fost necesară implementarea unui mecanism de actualizare în timp real a unor informații critice din interfața utilizatorului, precum: numărul de cărți din coș, statusul utilizatorului autentificat, cererile de activare a abonamentelor sau rezervările de cărți aflate în așteptare. Într-o aplicație complexă, care conține multiple componente independente (navbar, panou de administrare, pagini de utilizator), actualizarea manuală a acestor date sau reîncărcarea paginii ar fi dus la o experiență de utilizare deficitară.

Pentru a rezolva această problemă, a fost ales **Observer Pattern**, un pattern comportamental care permite unui obiect să notifice automat toate obiectele interesate atunci când apare o modificare de stare. Acest pattern este potrivit în mod special pentru aplicațiile React, unde componentele trebuie să reacționeze dinamic la schimbări de date, fără a fi puternic cuplate între ele. Astfel, interfața rămâne sincronizată cu starea aplicației, iar utilizatorul beneficiază de feedback vizual imediat, fără întreruperi sau reîncărcări suplimentare.

Organizare cod:

Implementarea Observer Pattern este realizată la nivelul componentei principale App, care joacă rolul de **observer global**, gestionând starea aplicației și reacționând la evenimentele declanșate în alte părți ale sistemului. Comunicarea dintre componente se face prin intermediul **evenimentelor custom de tip window.dispatchEvent**, evitând dependențele directe între componente.

Responsabilitățile sunt clar separate:

- componentele care modifică starea aplicației (ex: login, logout, rezervări, activare abonament) acționează ca **Subject**, emițând evenimente;
- componenta App și alte componente interesate se comportă ca **Observers**, ascultând aceste evenimente și actualizând starea locală corespunzătoare.

Această abordare contribuie la o arhitectură decuplată, ușor de extins și de întreținut.

```
  const onUserChange = () => {
    const updated = localStorage.getItem("user");
    setCurrentUser(updated ? JSON.parse(updated) : null);
  };

  const onCartChange = () => {
    updateCartCount();
  };

  window.addEventListener("userChanged", onUserChange);
  window.addEventListener("cartChanged", onCartChange);

  return () => {
    window.removeEventListener("userChanged", onUserChange);
    window.removeEventListener("cartChanged", onCartChange);
  };
}, []);
```

Implementare:

Primul pas a constat în definirea stărilor globale necesare pentru afișarea informațiilor dinamice în interfață, precum currentUser, cartCount, pendingCount și pendingBorrowCount. Aceste stări sunt gestionate în componenta App folosind hook-ul useState.

Pentru a implementa mecanismul de observare, au fost utilizate **event listeners** atașate obiectului global window, în cadrul hook-urilor useEffect. De exemplu, atunci când datele utilizatorului se modifică (login, logout sau actualizare abonament), este declanșat evenimentul userChanged, iar componenta App reacționează prin reîncărcarea utilizatorului din localStorage și actualizarea stării interne.

În mod similar, pentru actualizarea numărului de produse din coș, este utilizat evenimentul cartChanged, care notifică automat componentele observatoare atunci când coșul este modificat. Astfel, badge-ul asociat coșului de cumpărături este actualizat instantaneu, fără a fi necesară reîncărcarea paginii.

Pentru funcționalitățile din zona de administrare, precum cererile de activare abonament sau rezervările de cărți, sunt utilizate evenimente dedicate (pendingUpdated, pendingBorrowsUpdated, reservationsChanged). La primirea acestor evenimente, aplicația actualizează stările corespunzătoare și afișează în timp real indicatorii vizuali (bulinuțele roșii) din interfață.

În cadrul fiecărui useEffect, listener-ele sunt înregistrate la montarea componentei și eliminate la demontare, folosind funcția de cleanup, ceea ce previne scurgerile de memorie și comportamentele nedorite. Componenta observatoare nu

cunoaște detalii despre sursa modificărilor, ci doar reacționează la evenimentele primite, respectând principiul decuplării.

Prin această implementare, Observer Pattern permite sincronizarea eficientă a stării aplicației cu interfața utilizatorului, oferind o experiență fluidă și coerentă, atât pentru utilizatorii obișnuiți, cât și pentru administratori.

3.3. Repository pattern – Puicar Georgiana – Oana

Motivare alegere:

În dezvoltarea aplicației de bibliotecă digitală, a fost necesară implementarea unui mecanism robust și eficient pentru accesul la datele persistente din baza de date MySQL. Într-o aplicație complexă, unde există multiple entități (Book, User, Borrow, Favorite, Rating) și operații frecvente de tip CRUD (Create, Read, Update, Delete), scrierea manuală a interogărilor SQL pentru fiecare operație ar fi dus la un cod repetitiv, dificil de întreținut și predispus la erori.

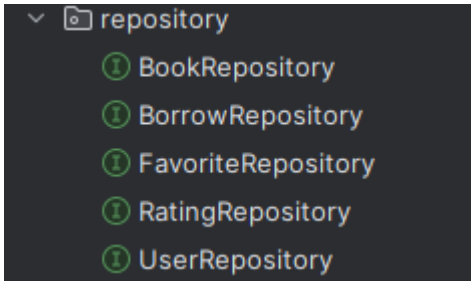
Repository Pattern a fost ales pentru a oferi o abstractizare clară între logica de business și stratul de acces la date. Acest pattern permite:

- **Separarea responsabilităților**: Logica de business rămâne în servicii, iar accesul la date este izolat în repository-uri
- **Reutilizarea codului**: Operațiile comune (save, findById, findAll, delete) sunt moștenite automat
- **Testabilitate îmbunătățită**: Repository-urile pot fi ușor înlocuite cu mock-uri în teste
- **Flexibilitate**: Schimbarea bazei de date sau a tehnologiei de persistență nu afectează restul aplicației
- **Interogări complexe declarative**: Prin Spring Data JPA, interogările pot fi definite prin simple metode Java, fără SQL explicit

Mai mult, **Spring Data JPA** oferă o implementare gata făcută a pattern-ului Repository, eliminând necesitatea scrierii codului boilerplate și permițând dezvoltatorilor să se concentreze pe logica specifică aplicației.

Organizare cod:

Toate interfețele repository au fost grupate în pachetul dedicat com.example.backend.repository, asigurând o structură clară și o bună separare a responsabilităților.



Fiecare repository corespunde unei entități din model și extinde interfața JpaRepository<T, ID> oferită de Spring Data JPA, unde:

- **T** = tipul entității (ex: Book, User)
- **ID** = tipul cheii primare (ex: Long)

Această organizare facilitează:

- Localizarea rapidă a operațiilor de acces la date pentru fiecare entitate
- Mentenanța și extinderea funcționalităților
- Respectarea principiului Single Responsibility

Implementare:

1. Interfața de bază JpaRepository

Toate repository-urile extind JpaRepository, care oferă automat metode standard:

- save(T entity) - salvează sau actualizează o entitate
- findById(ID id) - caută o entitate după ID
- findAll() - returnează toate entitățile
- deleteById(ID id) - șterge o entitate după ID
- count() - numără entitățile din tabel

2. BookRepository - Interogări personalizate

```java
package com.example.backend.repository;

import ...

public interface BookRepository extends JpaRepository<Book, Long> {

    @Query("SELECT b FROM Book b JOIN b.categories c WHERE LOWER(c) = LOWER(:category)")
    List<Book> findByCategory(@Param("category") String category);

    @Query("SELECT DISTINCT b FROM Book b WHERE " +
            "LOWER(b.title) LIKE LOWER(CONCAT('%', :query, '%')) OR " +
            "LOWER(b.author) LIKE LOWER(CONCAT('%', :query, '%')) OR " +
            "LOWER(b.description) LIKE LOWER(CONCAT('%', :query, '%'))")
    List<Book> searchBooks(@Param("query") String query);
}
```

Prin adnotarea @Query, se pot defini interogări JPQL personalizate pentru cazuri specifice, cum ar fi:

- Căutarea cărților după categorie (relațieMany-to-Many cu @ElementCollection)
- Căutarea full-text în multiple câmpuri

3. UserRepository - Query methods

```java
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByEmail(String email);
    boolean existsByEmail(String email);
    List<User> findBySubscriptionStatus(String subscriptionStatus);
}
```

Spring Data JPA generează automat implementarea metodelor pe baza numelui acestora:

- findBy... - caută entități după un criteriu
- existsBy... - verifică existența
- Query derivation elimină nevoia de SQL explicit

4. BorrowRepository - Interogări complexe cu agregare

Această interfață demonstrează:

- **Sortarea automată** prin OrderByIdDesc
- **Agregări și grupări** pentru statistici (cele mai citite cărți)
- **Interogări de numărare** pentru validări busines

```java
@Repository  6 usages  GeorgiBerciu28 +2
public interface BorrowRepository extends JpaRepository<Borrow, Long> {
    List<Borrow> findByUserIdOrderByIdDesc(Long userId);  1 usage  Franci0128
    List<Borrow> findByStatus(String status);  2 usages  Puicar_Georgiana

    List<Borrow> findByUserIdAndStatus(Long userId, String status);  no usages  Puicar_Georgiana
```

5. RatingRepository - Operații statistice

```java
@Repository  2 usages  Puicar_Georgiana
public interface RatingRepository extends JpaRepository<Rating, Long> {

    Optional<Rating> findByUserIdAndBookId(Long userId, Long bookId);  2 usages  Puicar_Georgiana

    List<Rating> findByBookId(Long bookId);  1 usage  Puicar_Georgiana

    @Query("""  1 usage  Puicar_Georgiana
        SELECT r.bookId, AVG(r.rating), COUNT(r)
        FROM Rating r
        GROUP BY r.bookId
        HAVING COUNT(r) >= :minRatings
        ORDER BY AVG(r.rating) DESC, COUNT(r) DESC
    """)
    List<Object[]> findTopRatedBooks(@Param("minRatings") long minRatings);
}
```

Permite calcularea rating-urilor medii și identificarea celor mai bine evaluate cărți.

6. Integrarea cu Service Layer

Repository-urile sunt injectate în clasele de servicii prin @Autowired:

```java
@Service  2 usages  GeorgiBerciu28
public class BookService {

    @Autowired
    private BookRepository bookRepository;

    public Optional<Book> getBookById(Long id) { return bookRepository.findById(id); }

    public Book saveBook(Book book) { return bookRepository.save(book); }
```

4.Diagrame-Berciu Oana-Georgiana

Pentru a evidenția modul de funcționare al aplicației și interacțiunea dintre compo

nentele acesteia, au fost realizate **trei tipuri de diagrame UML**: diagrama de secvență, diagrama de comunicare și diagrama de stări. Fiecare diagramă surprinde un aspect diferit al sistemului și contribuie la o mai bună înțelegere a arhitecturii și a fluxurilor de date.

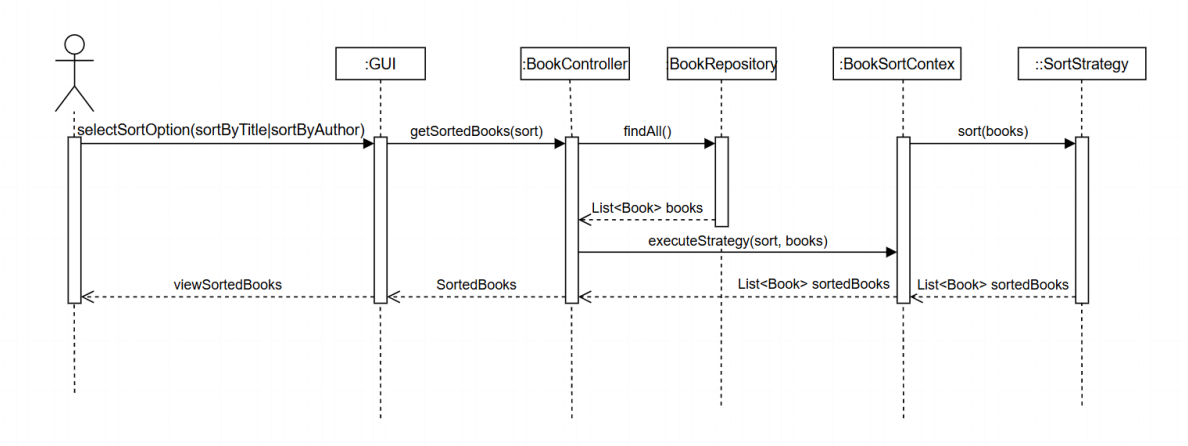
4.1 Diagrama de secventa

Diagrama de secvență ilustrează ordinea cronologică a interacțiunilor dintre utilizator, interfața web și componentele backend, pentru funcționalitatea de sortare a cărților.

Participanti: Utilizatorul si sistemul

Fluxul se desfășoară după cum urmează:

- Utilizatorul selectează criteriul de sortare (după titlu sau după autor) din meniul dropdown afișat pe pagina Home.
- Interfața web (GUI) trimite o cerere către backend, apelând metoda getSortedBooks, împreună cu criteriul de sortare selectat.
- BookController primește cererea și solicită lista completă de cărți din baza de date prin intermediul metodei findAll() din BookRepository.
- BookRepository returnează către BookController lista de cărți nesortată.
- BookController delegă procesul de sortare către BookSortContext, trimițând lista de cărți și strategia de sortare selectată.
- BookSortContext identifică strategia corespunzătoare criteriului primit și apelează metoda sort() a strategiei concrete.
- Strategia de sortare returnează lista de cărți sortată către BookSortContext.
- BookSortContext transmite lista sortată înapoi către BookController.
- BookController returnează rezultatul către interfața web, sub forma unei liste de cărți sortate.
- Interfața web afișează utilizatorului lista de cărți ordonată conform criteriului selectat.



4.2 Diagrama de comunicare

Diagrama de comunicare evidențiază modul în care obiectele colaborează pentru realizarea operației de adăugare a unei cărți, punând accent pe mesajele schimbate între acestea.

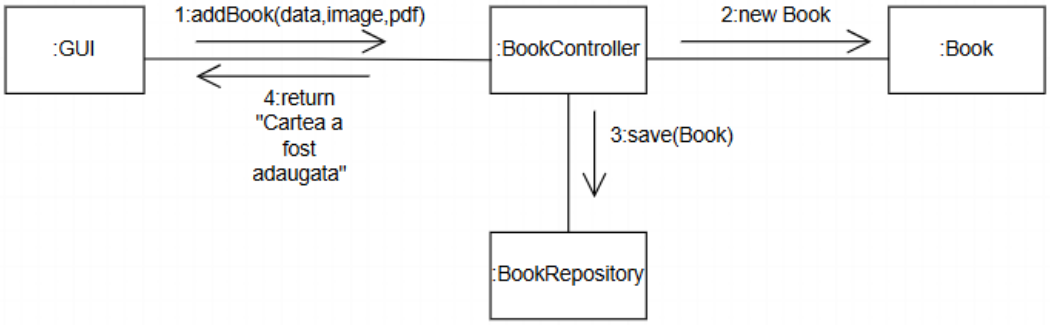
Participanti:Admin- inițiază operația de adăugare a unei cărți;

Obiecte:

- **GUI (Interfața web)** – colectează datele introduse de admin și trimite cererea către backend;
- **BookController** – coordonează procesul de adăugare a cărții;
- **Book** – obiectul de tip entitate care reprezintă cartea creată;
- **BookRepository** – responsabil de persistarea cărții în baza de date.

Fluxul se desfășoară după cum urmează:

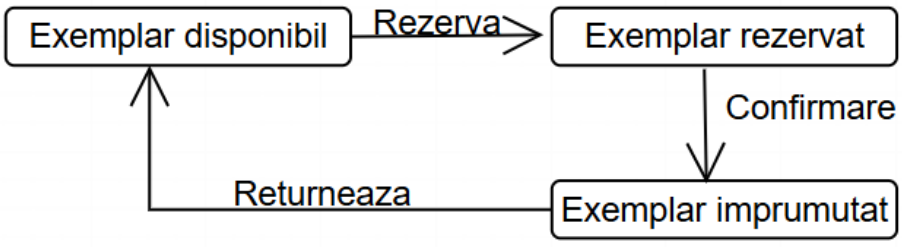
- Administratorul completează formularul de adăugare a unei cărți în interfața web și inițiază acțiunea de salvare.
- Interfața web (GUI) trimite cererea de adăugare către BookController, împreună cu datele cărții și fișierele asociate (imagine, PDF).
- BookController creează un nou obiect de tip Book și setează atributele acestuia pe baza datelor primite.
- BookController trimite obiectul Book către BookRepository, apelând metoda de salvare pentru persistarea acestuia în baza de date.
- BookRepository salvează cartea.
- BookController returnează un mesaj de confirmare către interfața web, indicând faptul că operația de adăugare a fost realizată cu succes.
- Interfața web afișează administratorului mesajul de confirmare.



4.3 Diagrama de stari

Această diagramă de stări a fost realizată pentru a descrie operația de împrumut a unui exemplar de carte din cadrul aplicației web. Diagrama modelează modul în care starea unui exemplar se modifică pe parcursul procesului de împrumut, în funcție de acțiunile utilizatorilor:

- Inițial, exemplarul de carte se află în starea **Disponibil**, fiind pregătit pentru a fi împrumutat.
- În urma unei acțiuni de **rezervare**, exemplarul trece din starea *Disponibil* în starea **Rezervat**.
- După confirmarea împrumutului de catre admin, exemplarul își schimbă starea din *Rezervat* în **Împrumutat**.
- Pe durata împrumutului, exemplarul rămâne în starea **Împrumutat**, nefiind disponibil pentru alți utilizatori.
- La efectuarea operației de **returnare**, exemplarul trece din starea *Împrumutat* înapoi în starea **Disponibil**, putând fi din nou rezervat sau împrumutat.



5. Diagrame – Szarka Francesca - Lara

5.1 Diagrama de secventa

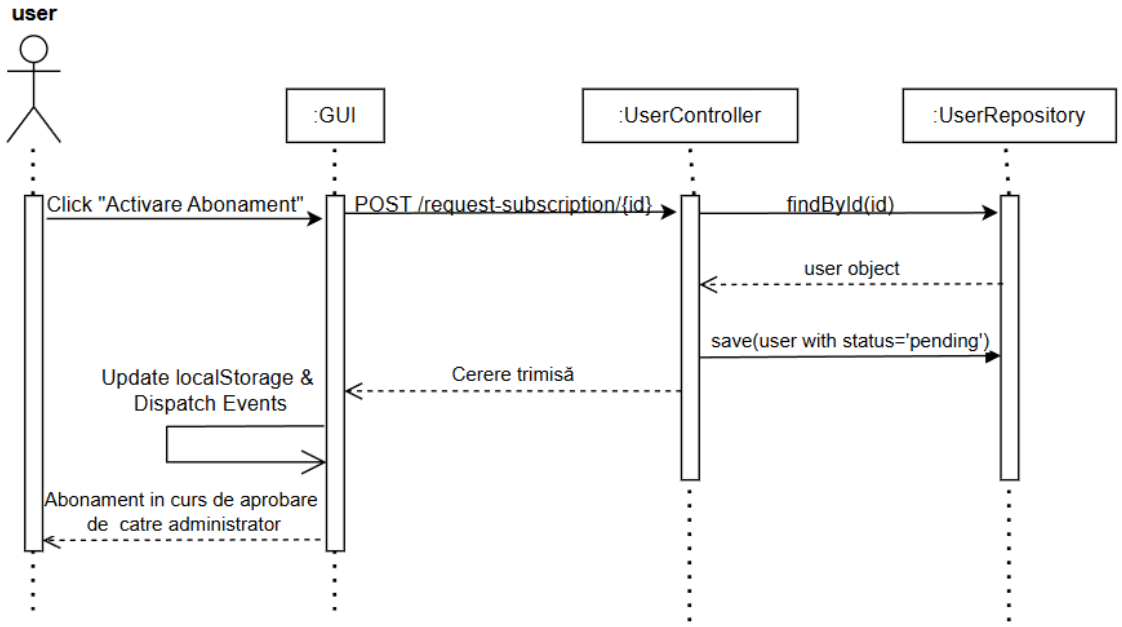
Diagrama de secvență ilustrează ordinea cronologică a interacțiunilor dintre Utilizator, Administrator, Interfața Web (React) și componentele Backend (Spring Boot), pentru procesul de solicitare și aprobare a abonamentului.

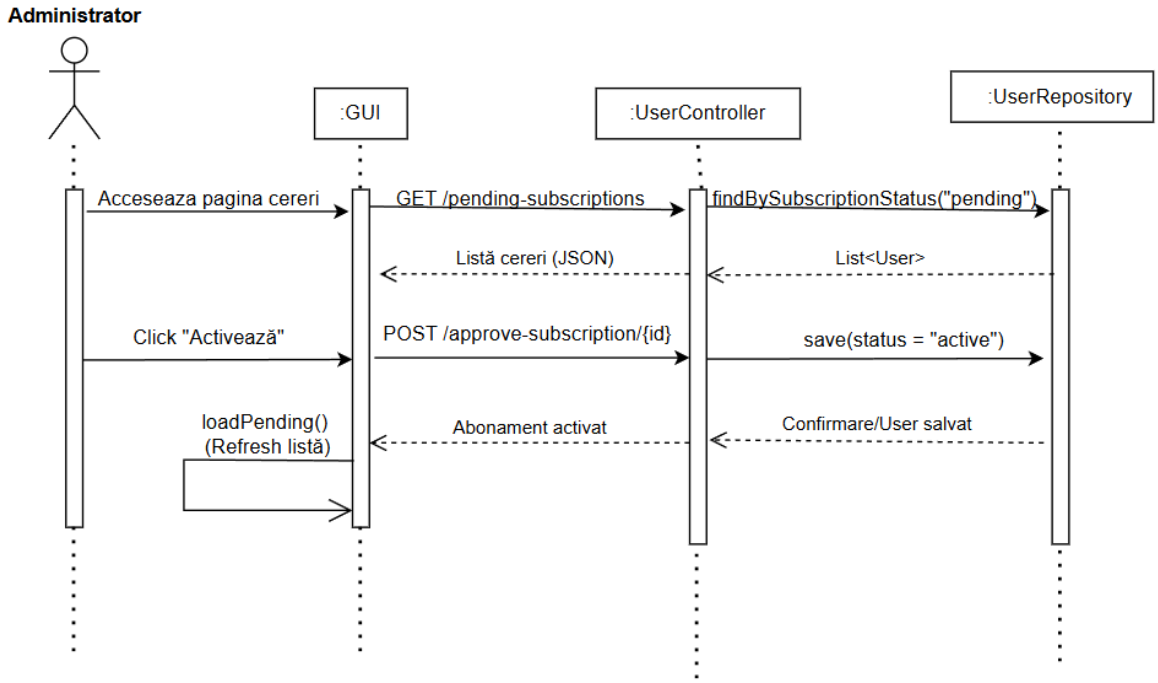
Participanți:

- **Utilizator**: Cel care solicită abonamentul.
- **Manager (Admin)**: Cel care aprobă cererea.
- **GUI (Interfața Web)**: Componentele React (Abonament și ActivareAbonament).
- **UserController**: Controller-ul REST care gestionează rutele API.
- **UserRepository**: Interfața de acces la baza de date.

Fluxul se desfășoară după cum urmează:

1. **Solicitarea (Utilizator):** Utilizatorul apasă butonul "Da, doresc activarea abonamentului".
2. **Trimitere Cerere:** GUI apelează API-ul requestSubscription(id).
3. **Actualizare Stare (Pending):** UserController găsește utilizatorul în UserRepository, îi setează statusul pe "pending" și salvează modificarea.
4. **Notificare Vizuală:** GUI actualizează localStorage și emite un eveniment local (userChanged) pentru a reflecta starea de așteptare în interfață.
5. **Vizualizare Cereri (Admin):** Administratorul accesează pagina de activări, iar GUI apelează getPendingSubscriptions().
6. **Preluare Date:** UserController solicită de la UserRepository lista utilizatorilor cu statusul "pending" și o returnează către GUI.
7. **Aprobare:** Administratorul apasă butonul "Activează", declanșând apelul către approveSubscription(id).
8. **Activare Finală:** UserController actualizează statusul utilizatorului în "active" prin UserRepository.
9. **Confirmare:** Sistemul returnează confirmarea, iar lista de cereri din interfață este reîmprospătată.





5.2 Diagrama de comunicare

Diagrama evidențiază colaborarea dintre obiectele sistemului pentru crearea unui cont nou, punând accent pe fluxul de validare și persistența datelor.

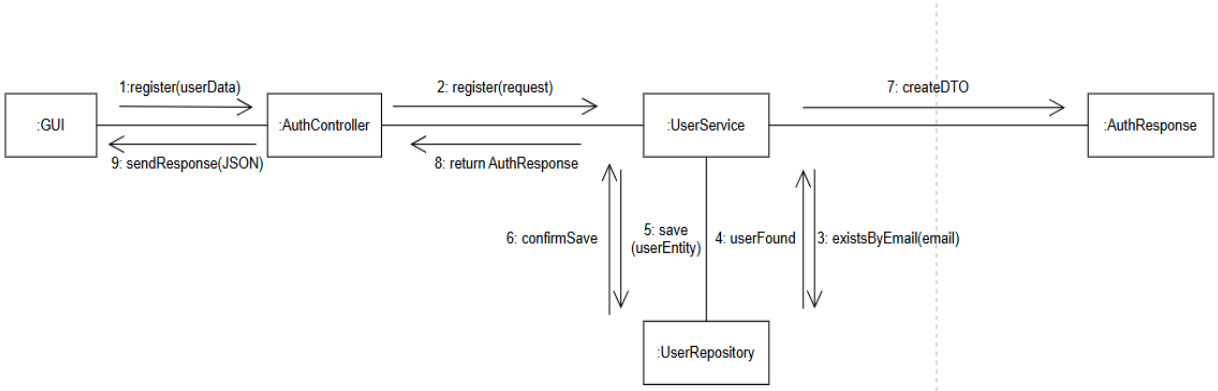
Participanți:

- **Utilizator**: Inițiază operația de creare cont.

- **GUI (Interfața Web)**: Colectează datele (Nume, Email, Parolă) și face validarea preliminară.
- **AuthController**: Primește cererea și o direcționează către serviciul de utilizatori.
- **UserService**: Execută logica de business (verifică domeniul @library și disponibilitatea email-ului).
- **UserRepository**: Interfața care salvează efectiv noul obiect User în baza de date.
- **AuthResponse (DTO)**: Obiectul care transportă confirmarea succesului înapoi la interfață.

Fluxul Mesajelor (Numerotarea pentru desen)

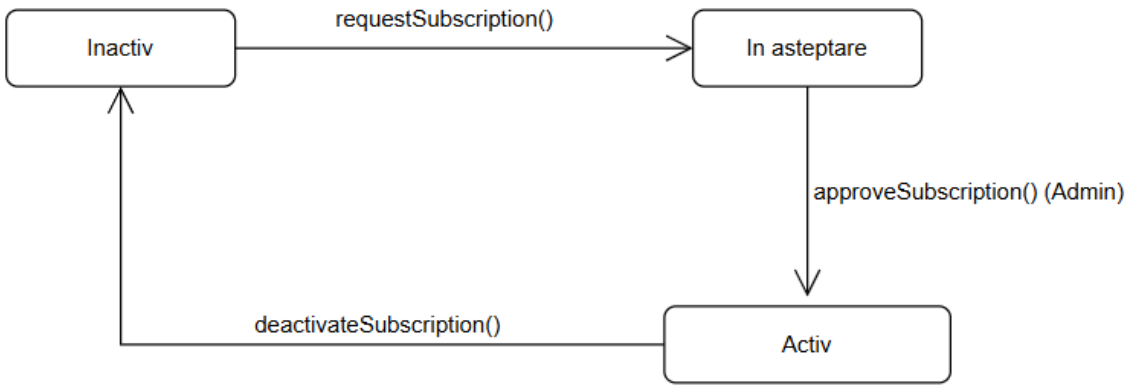
1. **1: register(userData)** → Utilizatorul trimite formularul completat.
2. **2: register(RegisterRequest)** → GUI apelează endpoint-ul /api/auth/register.
3. **3: register(request)** → AuthController apelează metoda de înregistrare din UserService.
4. **4: existsByEmail(email)** → UserService verifică prin UserRepository dacă email-ul este deja în baza de date.
5. **5: save(UserEntity)** → Dacă verificările trec, UserService solicită salvarea noului utilizator.
6. **6: savedUser** ← UserRepository confirmă salvarea și returnează obiectul creat.
7. **7: createSuccess(message)** → UserService creează obiectul AuthResponse cu mesajul "Registration successful!".
8. **8: return AuthResponse** ← UserService returnează răspunsul către AuthController.
9. **9: sendResponse(JSON)** ← AuthController trimite răspunsul final către interfața React.



5.3 Diagrama de stari

Această diagramă a fost realizată pentru a descrie fluxul de aprobare al unui abonament în cadrul aplicației. Ea modelează modul în care solicitarea unui utilizator trece prin diferite stări, de la trimiterea cererii până la activarea finală:

- **Punctul de Start** : Reprezintă momentul în care utilizatorul decide să devină membru activ.
- **Starea: Inactiv (Implicit)**: Imediat după înregistrare, utilizatorul se află în această stare. Nu are drept de împrumut, dar poate vizualiza catalogul.
- **Starea: În așteptare (Pending)**: În urma acțiunii de „Solicitare Abonament" din interfața React, cererea trece în această stare. În acest moment, datele sunt vizibile în panoul administratorului.
- **Starea: Activ (Active)**: După ce administratorul verifică și apasă butonul de aprobare, starea se modifică în **Activ**. Acum utilizatorul poate rezerva și împrumuta cărți.
- **Starea: Respins / Anulat**: Dacă administratorul consideră că datele sunt incorecte sau dacă utilizatorul își retrage cererea, aceasta revine în starea **Inactiv** sau trece într-o stare de **Respins**.



6. Diagrame – Puicar Georgiana - Oana

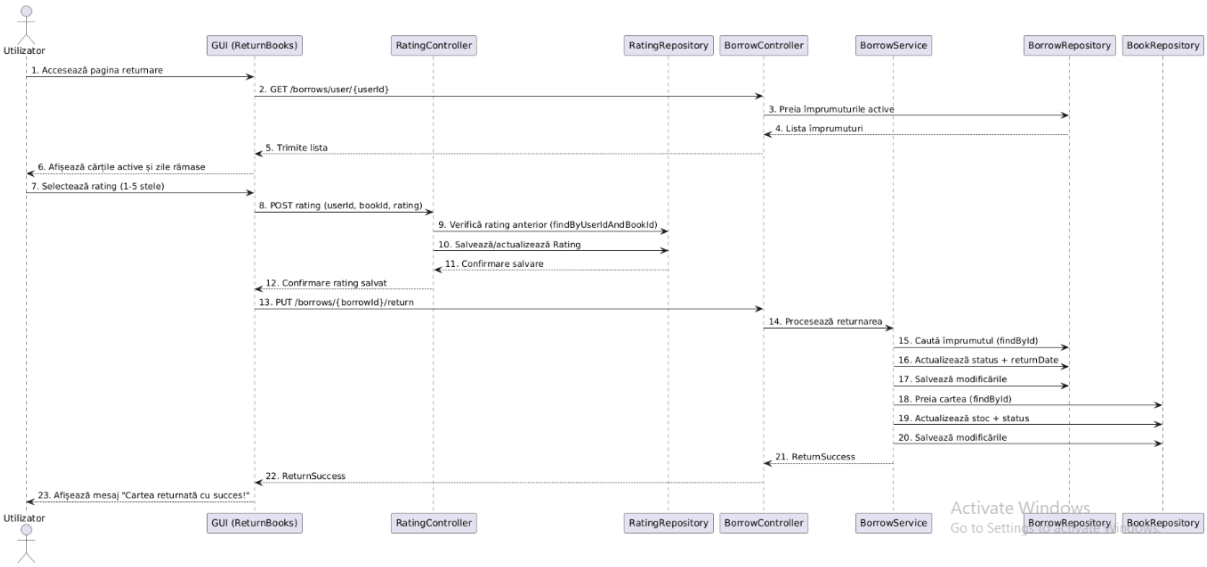
6.1 Diagrama de secvență

Diagrama de secvență arată interacțiunile dintre **utilizator**, GUI și componentele backend pentru returnarea unei cărți și acordarea unui rating.

Participanți: Utilizator, GUI (ReturnBooks), RatingController, RatingRepository, BorrowController, BorrowService, BorrowRepository, BookRepository.

Fluxul principal:

1. Utilizatorul accesează pagina de returnare; GUI preia lista de împrumuturi active prin BorrowController și BorrowRepository și o afișează.
2. Utilizatorul selectează un rating; GUI trimite datele către RatingController, care verifică și salvează rating-ul prin RatingRepository, apoi confirmă GUI.
3. GUI trimite cererea de returnare către BorrowController; BorrowService actualizează statusul împrumutului și data returnării în BorrowRepository.
4. BorrowService actualizează stocul și statusul cărții prin BookRepository.
5. Confirmarea returnării este transmisă înapoi către GUI, care afișează utilizatorului mesajul de succes și reîncarcă lista de împrumuturi.



6.2 Diagrama de comunicare

Diagrama de comunicare evidențiază modul în care obiectele colaborează pentru realizarea operației de **returnare a unei cărți cu acordarea unui rating**, punând accent pe mesajele schimbate între acestea.

Participant inițiator:

- **Utilizator** – inițiază procesul de returnare și acordă rating-ul.

Obiecte implicate:

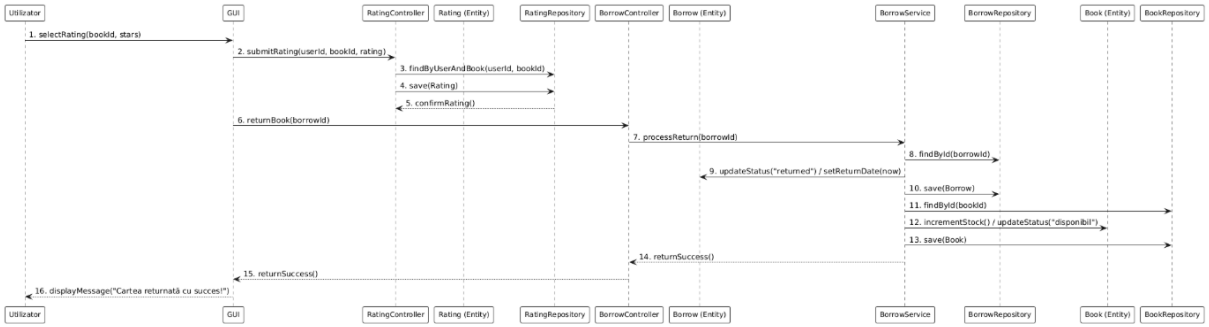
- **GUI (Interfața web)** – colectează input-ul utilizatorului și coordonează cererile către backend.
- **RatingController** – gestionează salvarea rating-ului utilizatorului.

- **Rating (Entity)** – obiectul care stochează evaluarea.
- **RatingRepository** – responsabil de persistența rating-ului în baza de date.
- **BorrowController** – coordonează procesul de returnare a cărții.
- **Borrow (Entity)** – obiectul care reprezintă împrumutul curent.
- **BorrowService** – conține logica de business pentru procesarea returnării.
- **BorrowRepository** – persistă modificările împrumutului.
- **Book (Entity)** – obiectul care reprezintă cartea și stocul său.
- **BookRepository** – actualizează datele cărții în baza de date.

Fluxul mesajelor (pas cu pas):

1. **Selectarea rating-ului:** Utilizatorul selectează un rating pentru carte în interfața web (stele 1–5).
2. **Transmiterea rating-ului:** GUI trimite cererea către RatingController, incluzând userId, bookId și rating.
3. **Verificarea rating-ului anterior:** RatingController verifică, prin RatingRepository, dacă utilizatorul a mai evaluat această carte.
4. **Salvarea rating-ului:** Dacă este necesar, RatingRepository salvează sau actualizează entitatea Rating.
5. **Confirmarea rating-ului:** RatingRepository returnează confirmarea către RatingController.
6. **Inițierea returnării:** GUI trimite cererea de returnare a cărții către BorrowController.
7. **Procesarea returnării:** BorrowController apelează BorrowService pentru a efectua logica de returnare.
8. **Accesarea împrumutului:** BorrowService preia datele împrumutului din BorrowRepository.
9. **Actualizarea împrumutului:** BorrowService schimbă statusul împrumutului în „returned” și setează data returnării.
10. **Persistarea împrumutului:** Modificările sunt salvate prin BorrowRepository.
11. **Accesarea cărții:** BorrowService preia cartea asociată împrumutului din BookRepository.

12. **Actualizarea stocului:** Stocul cărții este incrementat și statusul este setat la „disponibil” dacă stocul > 0.
13. **Persistarea cărții:** BookRepository salvează modificările pentru carte.
14. **Confirmarea returnării:** BorrowService trimite confirmarea către BorrowController.
15. **Transmiterea feedback-ului:** BorrowController returnează confirmarea către GUI.
16. **Afișarea mesajului:** GUI afișează utilizatorului mesajul „Cartea returnată cu succes!”.

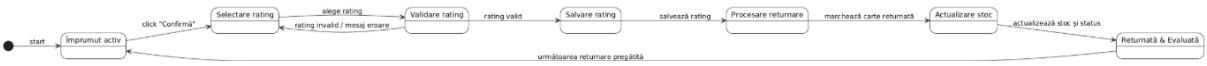


6.3 Diagrama de stări

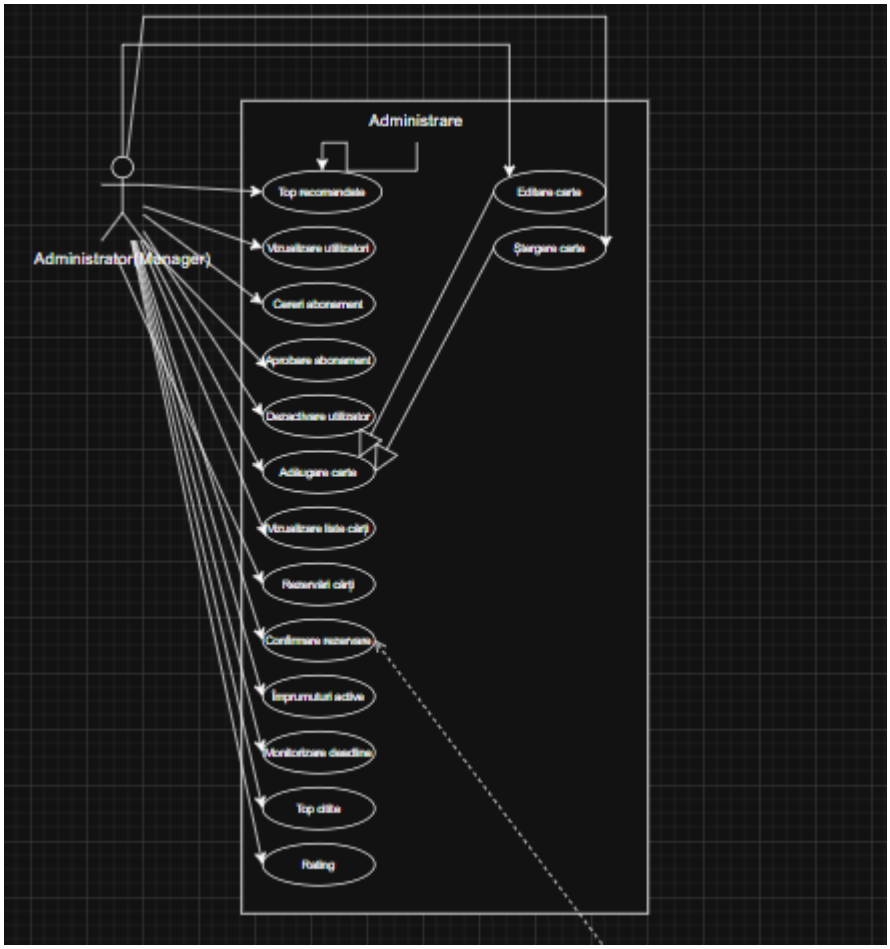
Această diagramă de stări descrie modul în care se modifică starea unui împrumut și a rating-ului asociat în aplicația web MyLibrary, pe parcursul procesului de returnare a unui exemplar de carte:

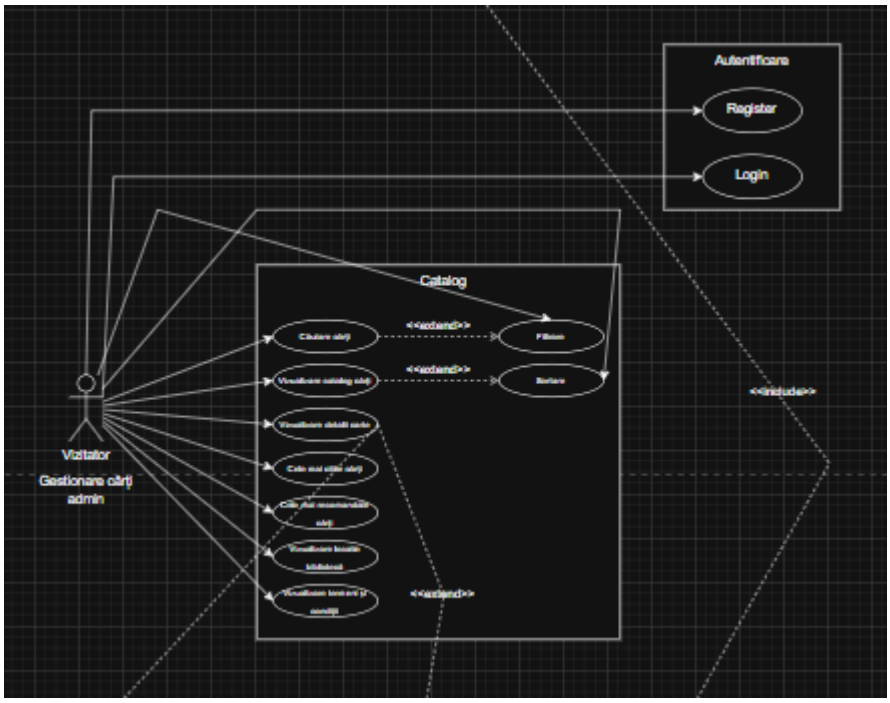
1. **Împrumut activ**
    - Starea inițială a împrumutului este „Împrumut activ” (cartea este la utilizator și nu poate fi împrumutată de altcineva).
    - Aceasta reprezintă exemplarul care urmează să fie returnat.
2. **Selectare rating**
    - Utilizatorul alege un rating pentru carte (1–5 stele).
    - Rating-ul este temporar selectat în interfață și așteaptă confirmarea.
    - Această stare apare după ce utilizatorul dă click pe „Confirmă returnarea”.

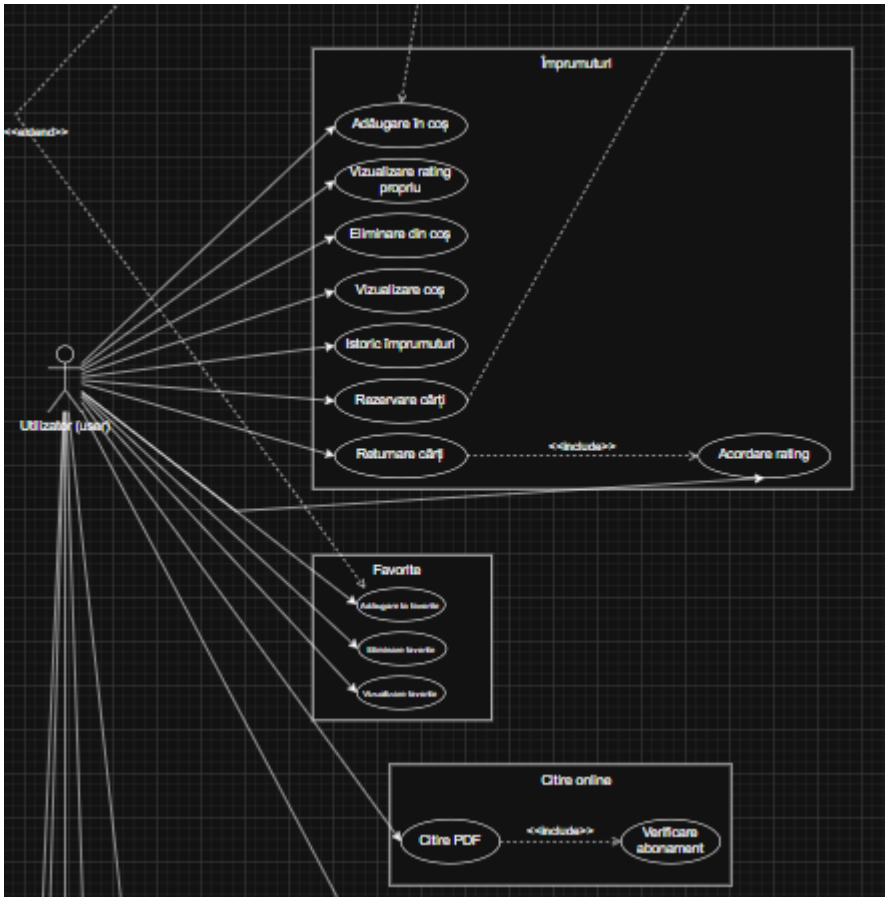
3. **Validare rating**
    - Sistemul verifică dacă rating-ul introdus este valid (1–5 stele).
    - Dacă rating-ul este invalid, fluxul revine în starea „Selectare rating” și se afișează un mesaj de eroare.
    - Dacă rating-ul este valid, fluxul trece la următoarea stare.
4. **Salvare rating**
    - Rating-ul valid este salvat în baza de date.
    - Această operație asigură că evaluarea utilizatorului este înregistrată și legată de carte.
5. **Procesare returnare**
    - Sistemul marchează împrumutul ca returnat și setează data returnării.
    - Cartea devine eligibilă pentru a fi rezervată sau împrumutată de alt utilizator.
6. **Actualizare stoc**
    - Stocul cărții este incrementat (+1).
    - Dacă stocul devine mai mare decât 0, statusul cărții se schimbă în „Disponibil”.
7. **Returnată & Evaluată**
    - Operația este finalizată: împrumutul apare ca returnat, iar rating-ul este vizibil în istoricul utilizatorului.
    - Această stare se leagă înapoi la „Împrumut activ” pentru următoarea operațiune de returnare.

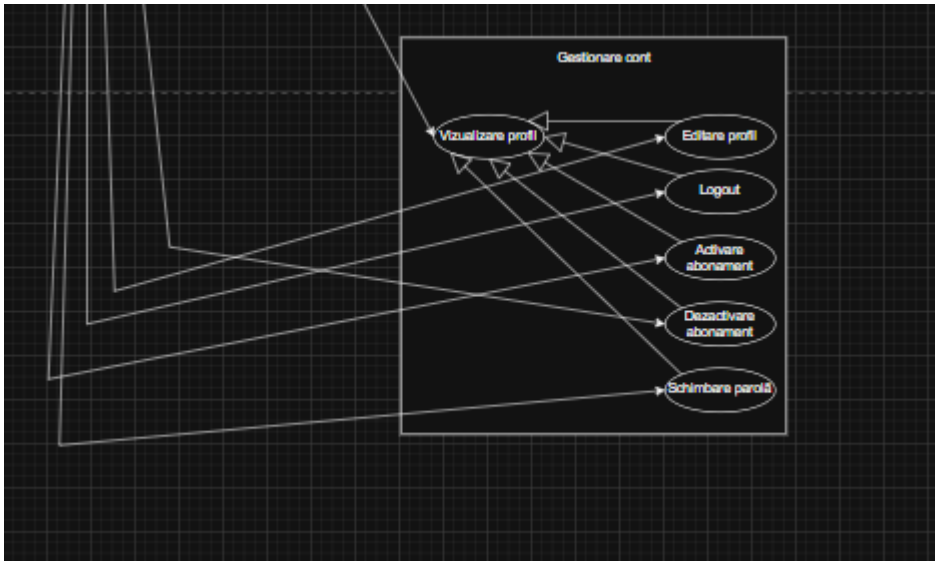


7. Diagrama use case









8. Cerințe non-funcționale

8.1 Viteză și performanță

8.1.1 Timp de răspuns

Aplicația trebuie să fie rapidă și să nu îi facă pe utilizatori să aștepte:

- Pagina principală trebuie să se deschidă în **maximum 2 secunde**
- Căutările și filtrele trebuie să funcționeze în **maximum 1 secundă**
- Actualizările în timp real (ex: coșul de cumpărături, rezervările) trebuie să apară **aproape instant**, în mai puțin de **0,1 secunde**

8.1.2 Capacitate

Sistemul trebuie să poată face față unui număr mare de utilizatori și date:

- Cel puțin **100 de utilizatori** pot folosi aplicația în același timp fără probleme
- Baza de date trebuie să suporte **minimum 10.000 de cărți**
- Aplicația trebuie să poată procesa **cel puțin 50 de acțiuni pe secundă**

8.1.3 Scalabilitate

Aplicația trebuie să poată crește ușor în timp:

- Să fie simplu de adăugat funcții noi fără a rescrie tot sistemul
- Să poată suporta o creștere a numărului de utilizatori de **aproximativ 3 ori** în următorii 2 ani

8.2 Securitate

8.2.1 Autentificare și drepturi

Datele utilizatorilor trebuie protejate:

- Parolele nu sunt salvate ca text, ci într-o formă sigură
- Există tipuri diferite de utilizatori (de exemplu: utilizator normal și administrator)
- Dacă un utilizator nu este activ timp de **30 de minute**, va fi deconectat automat

8.2.2 Protecția datelor

Informațiile personale trebuie tratate cu grijă:

- Toate datele transmise între aplicație și server sunt criptate
- Datele personale sunt gestionate conform legilor europene (GDPR)
- Sistemul este protejat împotriva atacurilor care încearcă să fure sau să modifice datele

8.2.3 Verificarea datelor introduse

Pentru a evita erori și abuzuri:

- Toate datele introduse de utilizatori sunt verificate
- Fișierele încărcate (imagini, documente) sunt controlate ca tip și dimensiune
- Sistemul este protejat împotriva atacurilor comune din web

8.3 Fiabilitate

8.3.1 Disponibilitate

Aplicația trebuie să fie disponibilă aproape tot timpul:

- Sistemul trebuie să funcționeze **cel puțin 99% din timp**
- Dacă apare o problemă, utilizatorul vede un mesaj clar
- Acțiunile importante (împrumuturi, returnări) trebuie să fie realizate corect, fără pierderi de date

8.3.2 Recuperare în caz de probleme

Pentru situații neprevăzute:

- Sistemul păstrează un istoric al erorilor pentru a fi reparate rapid
- Baza de date este salvată automat **în fiecare zi**
- Aplicația trebuie să revină la funcționare în **maximum 10 minute** după o problemă gravă

8.3.3 Corectitudinea datelor

Datele trebuie să rămână corecte:

- Legăturile dintre informații sunt protejate
- Nu se pot șterge date importante (ex: cărți împrumutate)
- Orice modificare este înregistrată (cine a făcut-o și când)

8.4 Ușurință în utilizare

8.4.1 Interfață

Aplicația trebuie să fie ușor de folosit:

- Design simplu și intuitiv
- Utilizatorul primește feedback pentru fiecare acțiune
- Mesajele de eroare explică clar ce s-a întâmplat și ce trebuie făcut

8.4.2 Accesibilitate

Aplicația trebuie să fie accesibilă tuturor:

- Funcționează bine pe calculator
- Culorile și textul sunt ușor de citit
- Fonturile au dimensiuni confortabile

8.4.3 Ajutor pentru utilizatori

- Există pagini de ajutor și termeni și condiții
- Funcțiile mai complicate sunt explicate prin mesaje ajutătoare
- Administratorii au acces la documentație detaliată

8.5 Întreținere și dezvoltare

8.5.1 Calitatea codului

- Codul este clar și bine organizat
- Fiecare parte a aplicației are un rol bine definit
- Se evită repetarea inutilă a codului

8.5.2 Testare

- Aplicația este testată automat pentru a evita erori
- Se folosesc teste pentru părți mici și pentru funcționarea generală
- Cel puțin **70% din aplicație** este acoperită de teste

8.5.3 Documentație tehnică

- Părțile mai complicate sunt explicate în comentarii
- Diagramele aplicației sunt actualizate atunci când apar schimbări importante

8.6 Compatibilitate și configurare

8.6.1 Compatibilitate

- Aplicația funcționează în cele mai populare browsere
- Serverul poate rula pe orice sistem de operare important
- Baza de date poate fi schimbată ușor dacă este nevoie

- **8.6.2 Configurare**
- Setările sunt separate de cod
- Aspectul aplicației poate fi personalizat

8.7 Respectarea regulilor și legilor

8.7.1 Standarde

- Aplicația respectă regulile generale de dezvoltare software
- Codul este organizat logic și eficient
- Baza de date este bine structurată

8.7.2 Legislație

- Sunt respectate regulile privind protecția datelor personale
- Utilizatorii își pot șterge contul și datele
- Termenii și condițiile trebuie acceptați în mod explicit

9.Bibliografie

- **Spring Boot Reference Documentation**
  https://docs.spring.io/spring-boot/docs/current/reference/html/
  Documentația oficială Spring Boot, utilizată pentru implementarea backend-ului și configurarea aplicației.
- **Spring Data JPA Reference Documentation**
  https://docs.spring.io/spring-data/jpa/docs/current/reference/html/
  Sursă utilizată pentru gestionarea persistenței datelor și maparea obiect–relațională.
- **React Official Documentation**
  https://react.dev/
  Documentația oficială React, utilizată pentru dezvoltarea interfeței frontend și gestionarea componentelor.
- **Oracle – Java Platform, Standard Edition Documentation**
  https://docs.oracle.com/en/java/
  Referință pentru limbajul Java și caracteristicile sale utilizate în proiect.
- **MySQL Documentation**
  https://dev.mysql.com/doc/
  Documentația oficială MySQL, utilizată pentru configurarea și utilizarea bazei de date relaționale.