PROIECT AWJ

**I. Specificare functionalitati de baza si suplimentare**

Acest proiect are ca scop automatizarea partii logistice a unei librarii/biblioteci unde se proceseaza carti cu scopul de a usura munca persoanelor implicate in inventarierea/adaugarea/stergerea sau contorizarea tipului de carti si a reducerii costurilor implicate. Astfel, soluția propusă va contribui semnificativ la creșterea productivității, reducerea erorilor umane și oferirea unei experiențe mai bune atât pentru personal, cât și pentru utilizatorii finali.

Exista trei functionalitati de baza: adaugare carte (se adauga o carte noua specificandu-se fiecare camp ce trebuie introdus + constrangeri asupra anumitor campuri e.g.: nu poate exista o carte fara titlu etc) , editarea carte sau modificare parametrii (in cazul in care se doreste modificarea unor campuri e.g.: stocul s-a modificat / pretul s-a schimbat ) si stergerea unei carti (in vederea eliminarii inregistrarilor care nu mai sunt valide).

Si functionalitati complexe precum: confirmarea la adaugare/stergere/editare printr-un pop-up pe pagina principala care se asigura de validarea cererii facute de user; un search box deasupra tabelului prin care se poate face o cautare in functie de titlul cartii sau numele autorului iar tabelul se populeaza dinamic cu aceste entitati corespunzatoare; handle-uri personalizare pentru exceptii la runtime in caz ca ceva nepermis sau ceva ce nu trebuia sa se intample are loc (wrong API request/wrong entered values or not respecting the fields).

**II. Descrierea claselor/metodelor/atributelor proiectului**

Proiectul este compus din 5 ramuri principale: *model* (unde se afla clasa **Book** care contine ‘scheletul’ programului adica fiecare camp al unei carti, constrangerile si setters+getters),controller (unde se afla clasa **BookController** care face comunicarea intre frontend si backend si rutarea de cereri http), service (unde se afla clasa **BookService** care se ocupa de ’business logic’ adica ce face aplicatia si implementarea metodelor necesare pentru proiect),exception (unde se afla clasele care se ocupa de managementul exceptiilor prin clasa **GlobalExceptionHandler** ce implementeaza handle-uri pentru erori) si clasa de baza (**ProiectAwjApplication**) *<main-ul>* care porneste tot procesul. Si de asemenea mai exista si fisierul **books.json**. Acesta actioneaza pe post de baza de date in proiect si se ocupa de stocarea informatiilor dupa inchiderea programului, astfel incat datele salvate sa fie persistente si sa nu fie sterse dupa terminarea rularii programului.

**Clasa Book:**

Este compusa din setters+getters si un constructor care atribuie la apelarea clasei valori pentru cele 7 campuri (id, titlu, autor, anul publicatiei, genul cartii, pretul si stocul). Setters si getters nu sunt direct folositi ci am optat pentru o incapsulare cu lombok pentru a reduce codul de tip boilerplate. De asemenea, la crearea celor 7 variabile am adaugat si niste constrangeri de integritate pentru diferite atribute iar id-ul este de tip identitate generat cu biblioteca persistence.

**Clasa BookService:**

Sunt declarate 3 variabile( o lista de carti pentru un obiect de tip carte, fisierul in care vor fi salvate datele si un objectMapper care converteste datele din fisierul JSON in obiecte java si viceversa). Clasa are un constructor ce adauga o lista de carti intiale in caz ca fisierul nu exista iar in caz contrar incarca cartile deja existente din fisier. Metodele *loodBooks* si *saveBooks* se ocupa de prelucrarea cartilor din fisier prin salvarea lor respectiv incarcarea lor intr-o lista de tip carte. *getAllBooks* returneaza toate cartile aceasta este folosita in controller pt afisarea tabelului. Metodele *addOrUpdateBook, deleteBook* si *getBookById* sunt folosite pentru request-urile din frontend. Implementarea este destul de asemanatoare, aceste metode filtreaza in functie de id cartile pana gasesc sau nu cartea respectiva iar dupa aceea o sterg/adauga/dau update. Metoda privata de initializare *getInitialBooks* nu are un rol functional dar este totusi folositoare in caz ca fisierul nu exista sau are loc o rularea clean-slate (de la 0) aceasta creaza o lista cu 10 carti care sunt returnate la functia apelanta.

**Clasa BookController:**

Clasa controller se ocupa de request-urile pentru POST/PUT/DELETE/GET cu patru rute principale si implementarea celorlalte clase:

*/* --> pagina principala

*/api/books* --> pentru adaugare

*/api/books/search* --> pentru cautarea in functie de parametru

*/api/books/{id}* --> pentru update/delete in functie de id-ul cartii.

Metoda *getAllBooks* leaga frontend-ul de modelul carte pentru modificari dinamice ale atributelor.

Metoda *getBooks* incarca cartile din *BookService* pe ruta */api/books* si returneaza un response 200 cu valorile din payload (mai multe detalii in partea de Postman) iar *getBookById* returneaza payload-ul unei carti in functie de id-ul trimis din frontend daca cartea cu id-ul respectiv este gasita.

Metoda *searchBooks* cauta in functie de parametrul din URL daca exista o carte cu acel substring (titlu sau autor) in lista de carti si returneaza acele carti care indeplinesc criteriile. Daca URL nu contine nicio valoare sau doar spatii nu se modifica nimic.

Metodele *addBook* si *updateBook* valideaza detaliile despre fiecare care si afiseaza un mesaj corespunzator in caz ca totul este ok si cartea a fost creata/updated iar daca nu arunca o exceptie specifica din clasele de exceptii.In plus metoda *updateBook* modifica si atributele cartii (titlu,autor etc).

Metoda de *deleteBook* verifica daca id-ul este asociat unei carti si apeleaza metoda din *BookService* pentru stergerea cartii respective si un mesaj daca actiunea a fost indeplinita.

Metoda *validateBook* creaza o lista de erori (format string) si verifica daca toate constrangerile sunt indeplinite iar in caz contrar se adauga un mesaj specific la lista. La final daca lista de erori nu e goala se trimite in clasa de exceptii unde se verifica eroarea si se arunca o eroare generica in frontend cu campurile invalide.

**Clasa GlobalExceptionHandler /w implementari:**

Aceasta clasa este mai mult un pachet format din 5 clase: 3 clase care sunt doar flags (la apelare implementeaza clasa parinte cu super si mesaj), 1 clasa *interfata..ish* (nu esta chiar o interfata dar pune bazele mesajului general care este trimis ca system **log** la finalul erorii un schelet mostly) si o clasa de baza (implementeaza toate celelalte clase si transmite system **log**-ul mai departe)

Clasa **ErrorResponse** creaza formatul erori sub un format JSON si declara variabilele + constructori. Clasa **GlobalExceptionHandler** are cate o metoda pentru fiecare tip de exceptie specifica si creaza un obiect *ErrorResponse* ce este returnat inapoi pe ruta unde s-a produs eroarea cu statusul specific erorii.

**III. Descrierea elementelor functionale oferite prin interfata grafica cu utilizatorul**

Interfata grafica (frontend-ul) este un fisier html (***book\_list.html***)unde se gasesc implementarile pe partea clientului prin ajax (pentru modificari dinamice si viteza crescuta) si biblioteca jQuery. Prima parte este formata din CSS si adauga un stil responsive si Retro-synthwave pentru pagina. Elementele functionale oferite sunt:

- un searchbox deasupra tabelului cu carti prin care se solicita utilizatorului introducerea unui substring iar apoi tabelul filtreaza dinamic doar cartile care respecta sau au in componenta acel substring

- tabelul propriu-zis cu carti care contine fiecare atribut al cartii si doua butoane in dreptul fiecarei entitati pentru stergere/editare

- un form pentru adugarea unei carti noi sau editarea uneia existente ce indeplinesc constragerile fiecarui atribut si cu buton de resetare pentru a curata toate valorile introduse

Fiecare element este responsive si dinamic adica se modifica in timp real si prin folosirea ajax se limiteaza modificarile globale ale paginii crescand performanta. Cu fiecare entitate nou adaugata tabelul se mareste.

Fișierul include multiple apeluri AJAX pentru comunicarea asincronă cu serverul (GET pt fetch/ POST&PUT pt adaugare&editare si DELETE pt stergere)

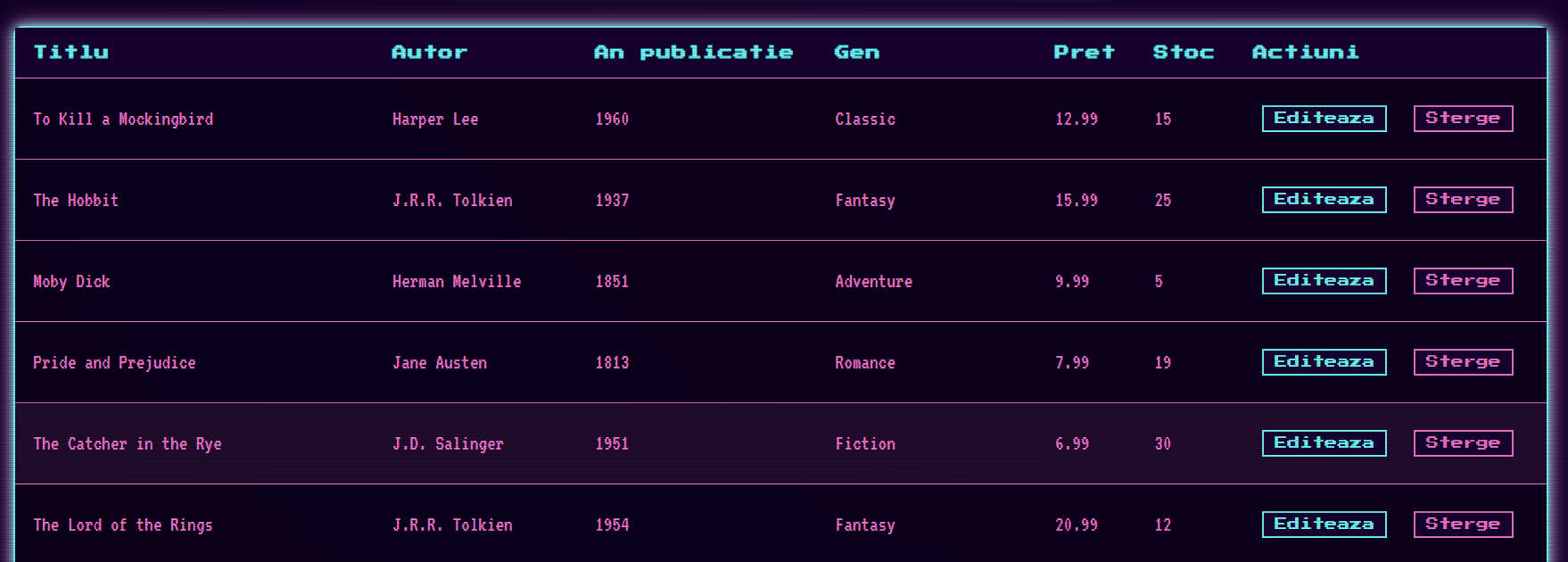
Thymeleaf este utilizat pentru a genera conținut dinamic (e.g., lista de cărți). Directive precum **th:each** și **th:text** sunt folosite pentru a popula datele dintr-un model server-side.



SEARCH BOX



FORM ADD/UPDATE



TABELUL PRINCIPAL

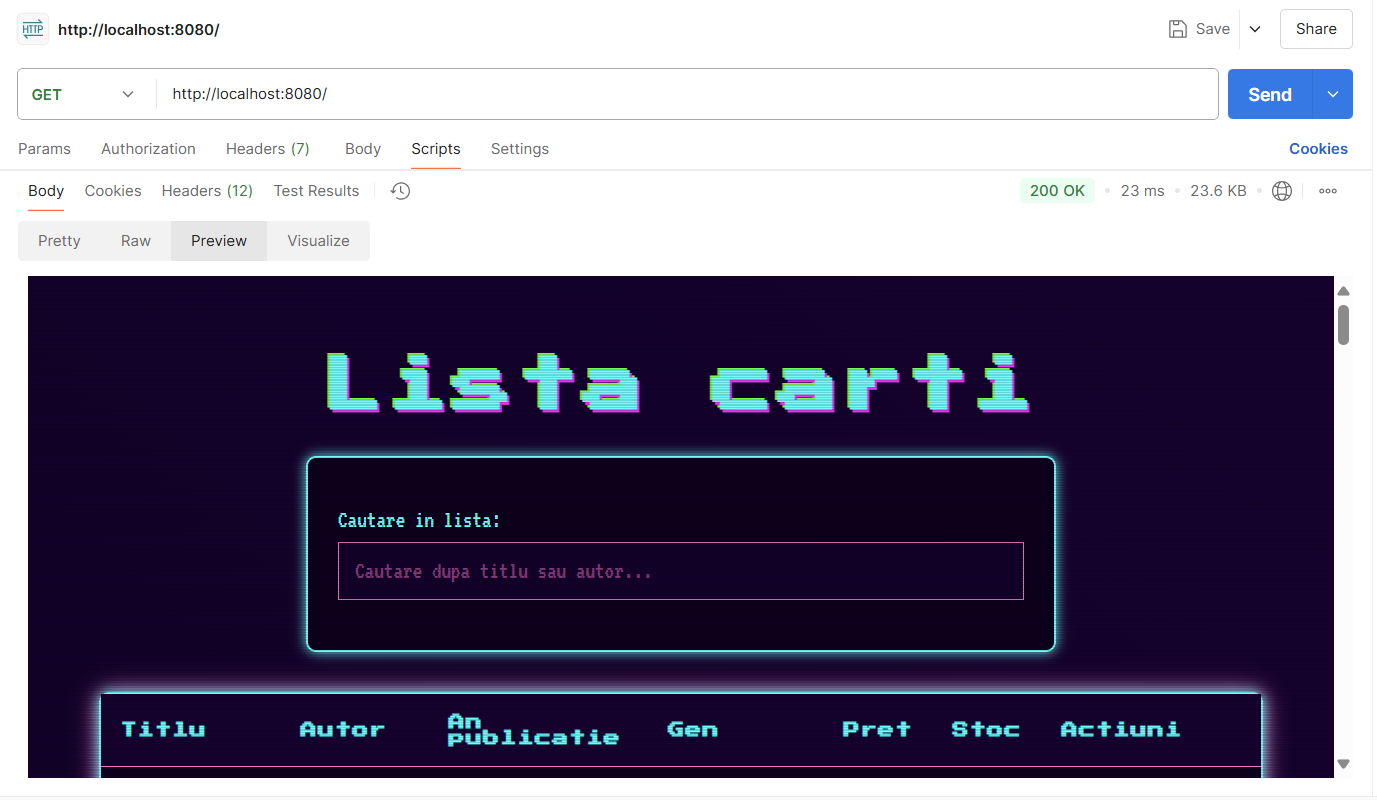
**IV. Aspecte privind testarea aplicatiei**

Din punct de vedere al testarii am inclus 3 clase de test pentru controller, service si book. De asemenea este inclusa si clasa default de test pentru main-ul aplicatiei.

Fiecare din aceste clase testeaza unele functionalitati ale programului prin testare unitara folosind pachetul Junit si metodele aferente (assert-urile) cat si framework-ul Mockito.

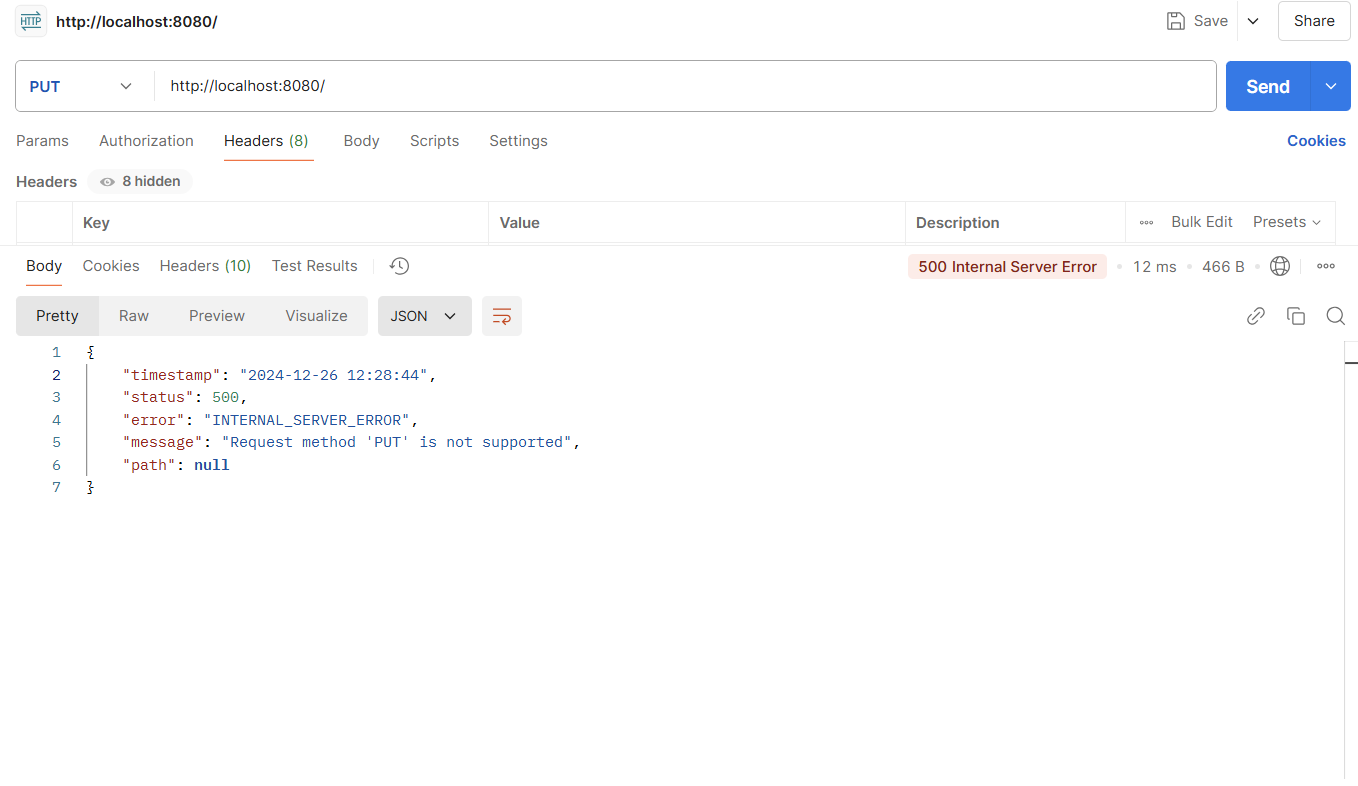
Am facut teste local direct din interfata grafica pentru adaugare/stergere/editare si cautare cat si teste pentru metodele http cu *PostMan* cu scopul de a analiza fiecare eroare ce poate fi prezenta pe rutele REST.

**POSTMAN checks:**

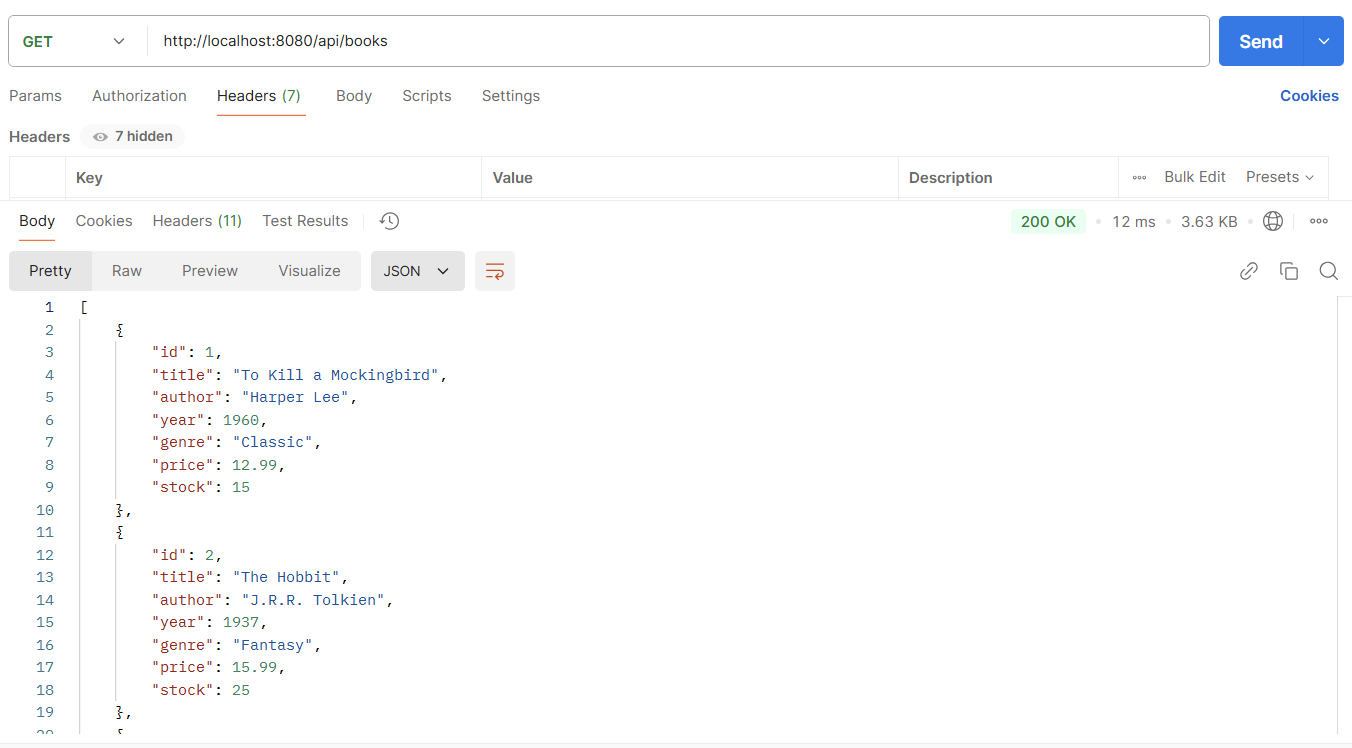




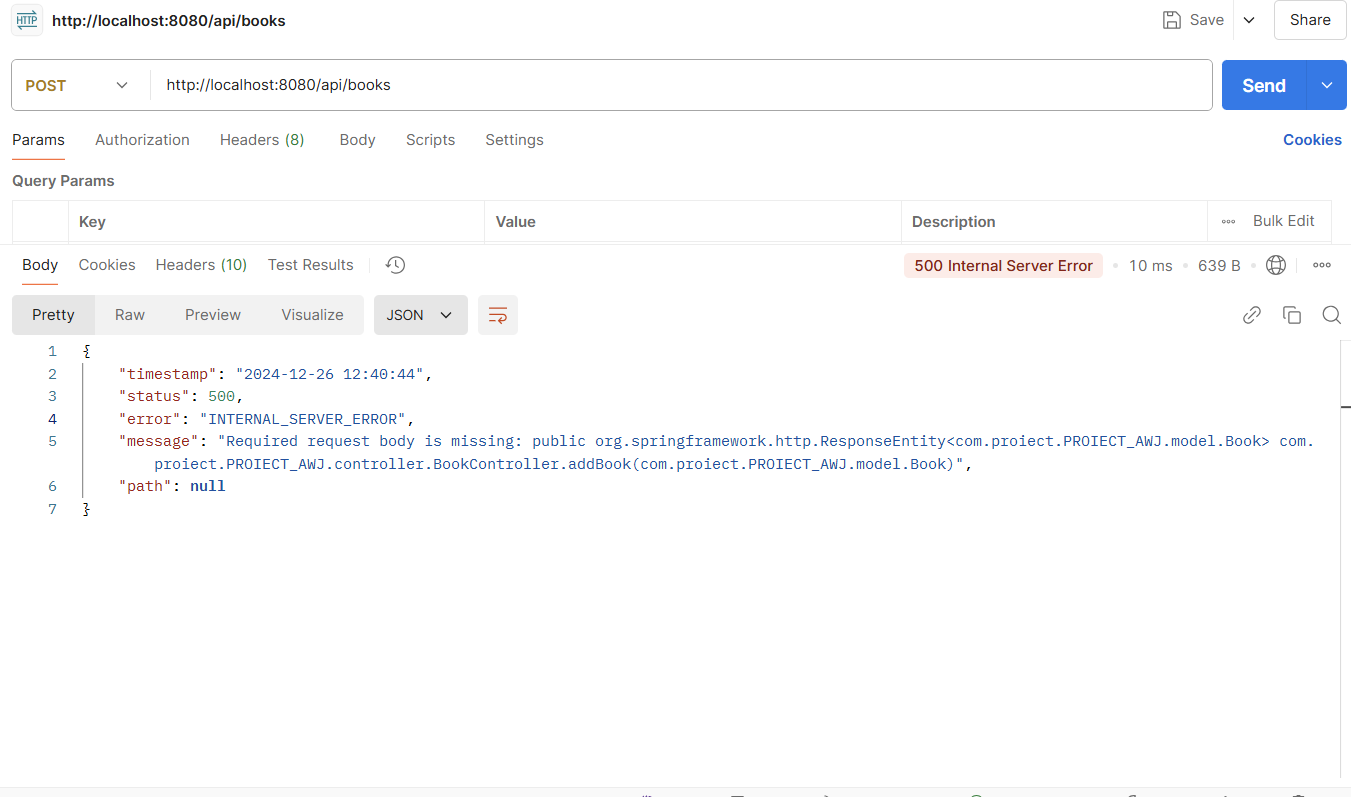
Se poate observa ca ruta de baza (root /) are permise doar optiunile de GET/HEAD/OPTIONS iar pentru GET avem pagina principala a aplicatiei.Dar pentru o ruta interzisa primim un log-ul de eroare de mai jos. Acest log face referire la clasa **GlobalExceptionHandler** mesajul primit fiind unul diferit fata de cel generic acesta fiind vizibil doar in depanarea/debugging-ul aplicatiei si nu propriu zis in frontend cand apare o eroare.Deci de aici se observa utilitatea pachetului exception care ajuta la debugging-ul rutelor HTTP.



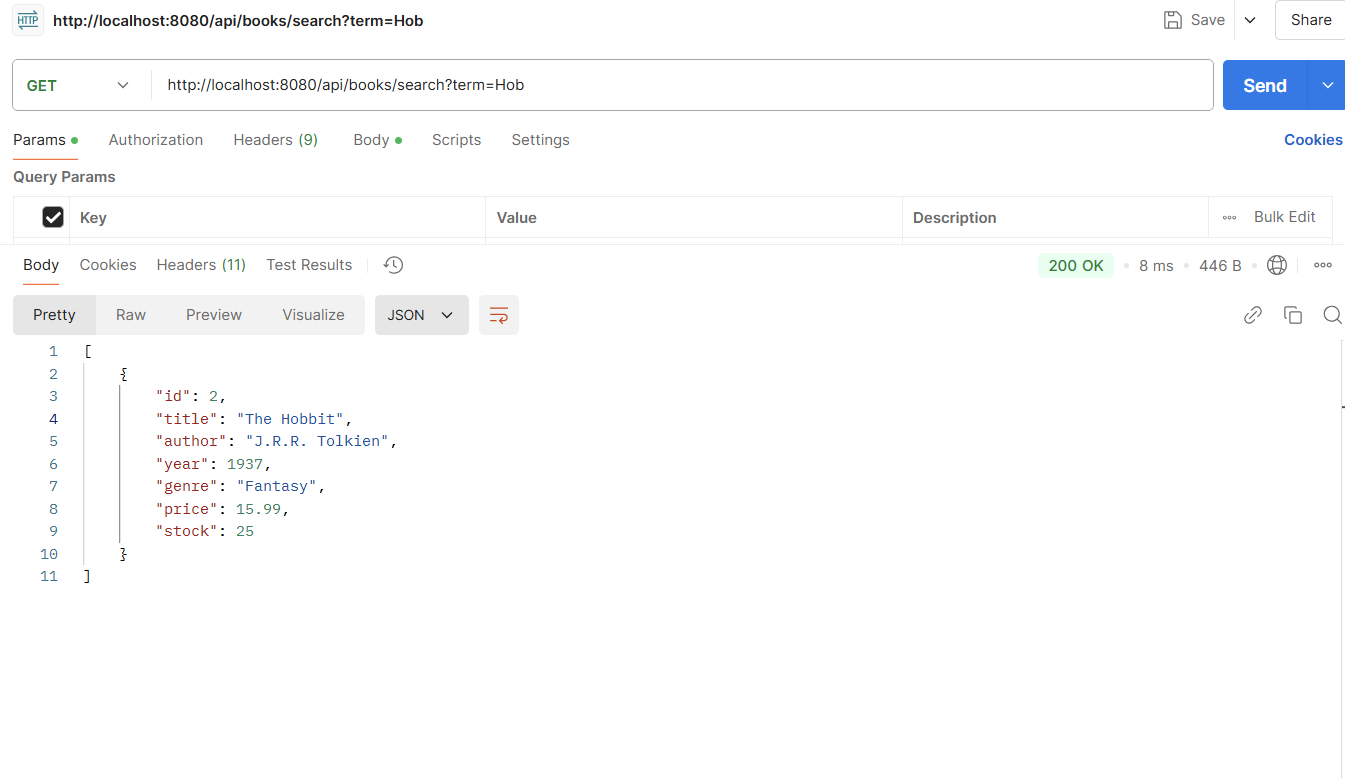
Pentru ruta /api/books metoda GET returneaza toate cartile din fisierul JSON



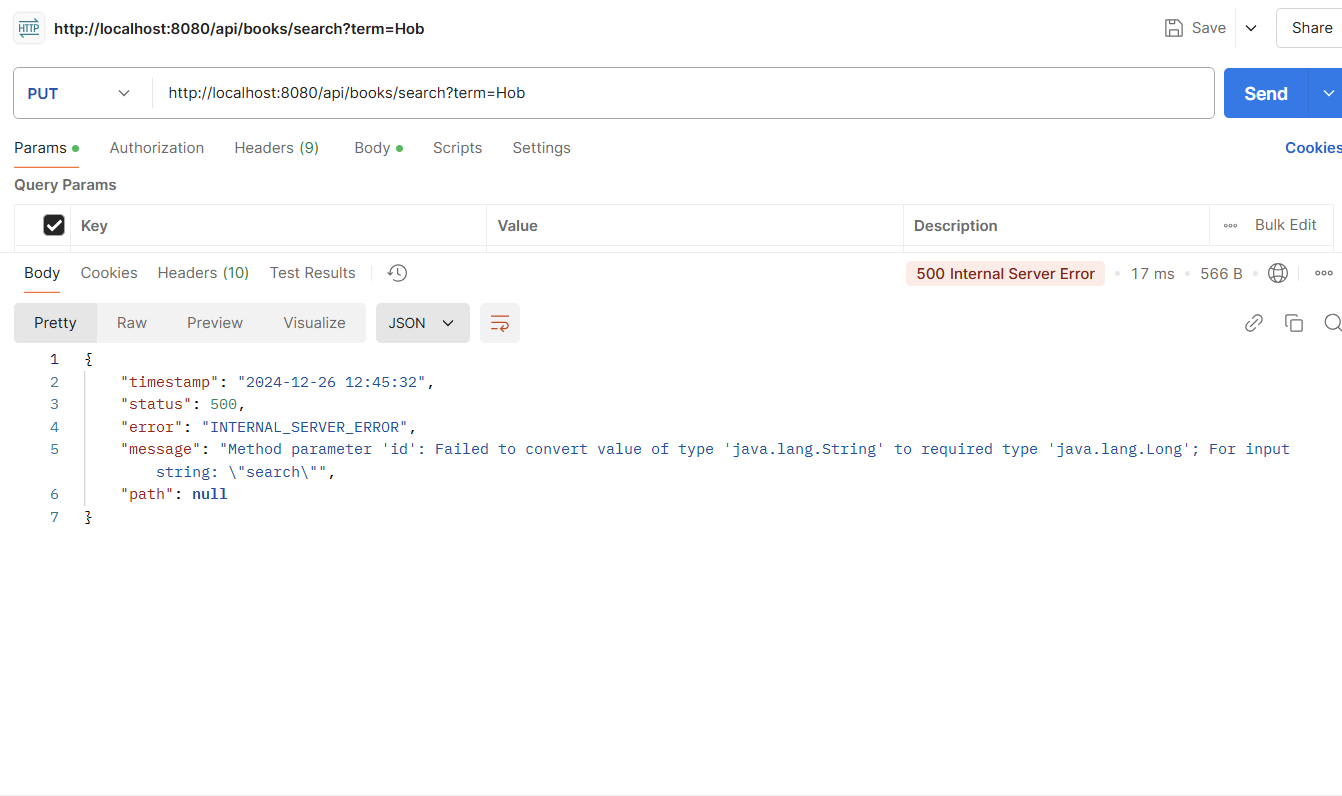
Iar metoda POST scoate o eroare



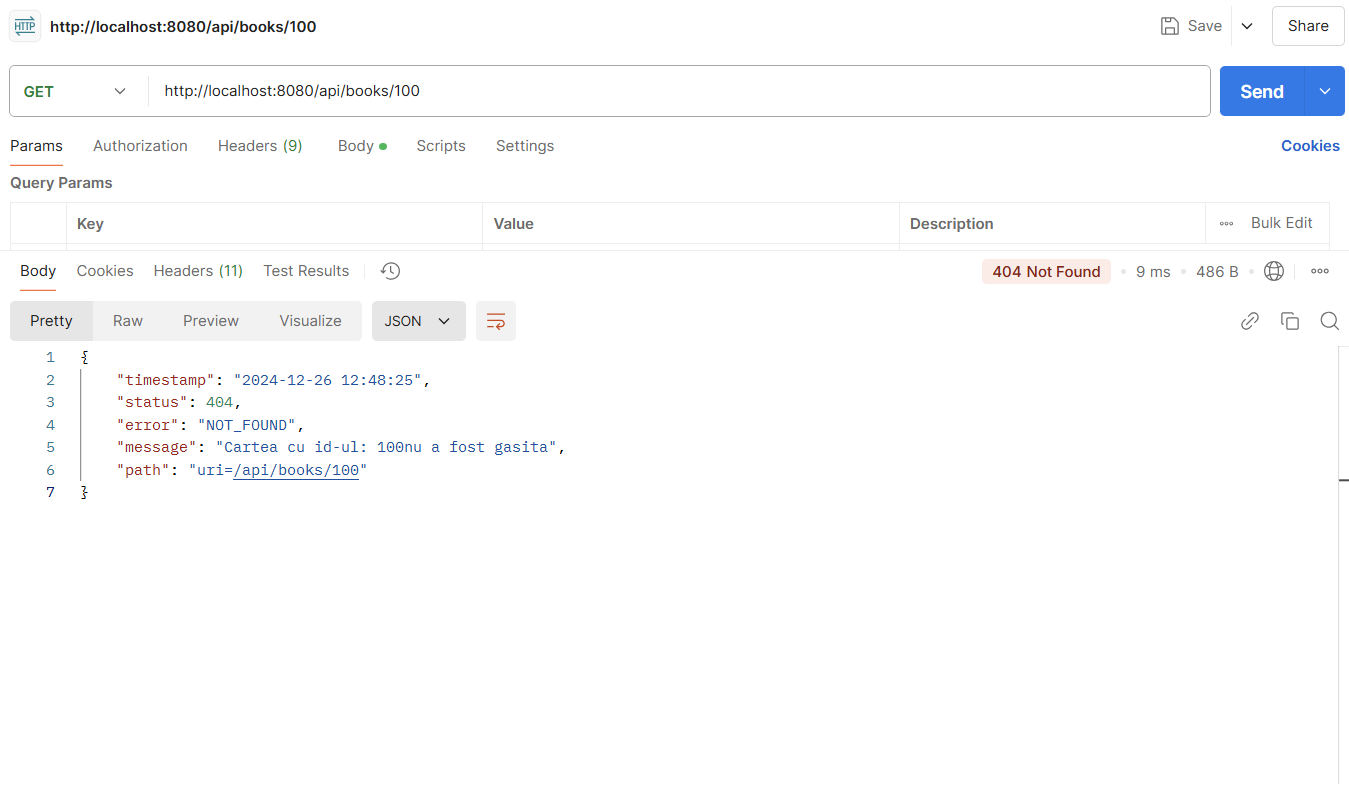
Ruta /api/books/search primeste un parametru din frontend si pe baza acelui parametru GET returneaza cartile specifice

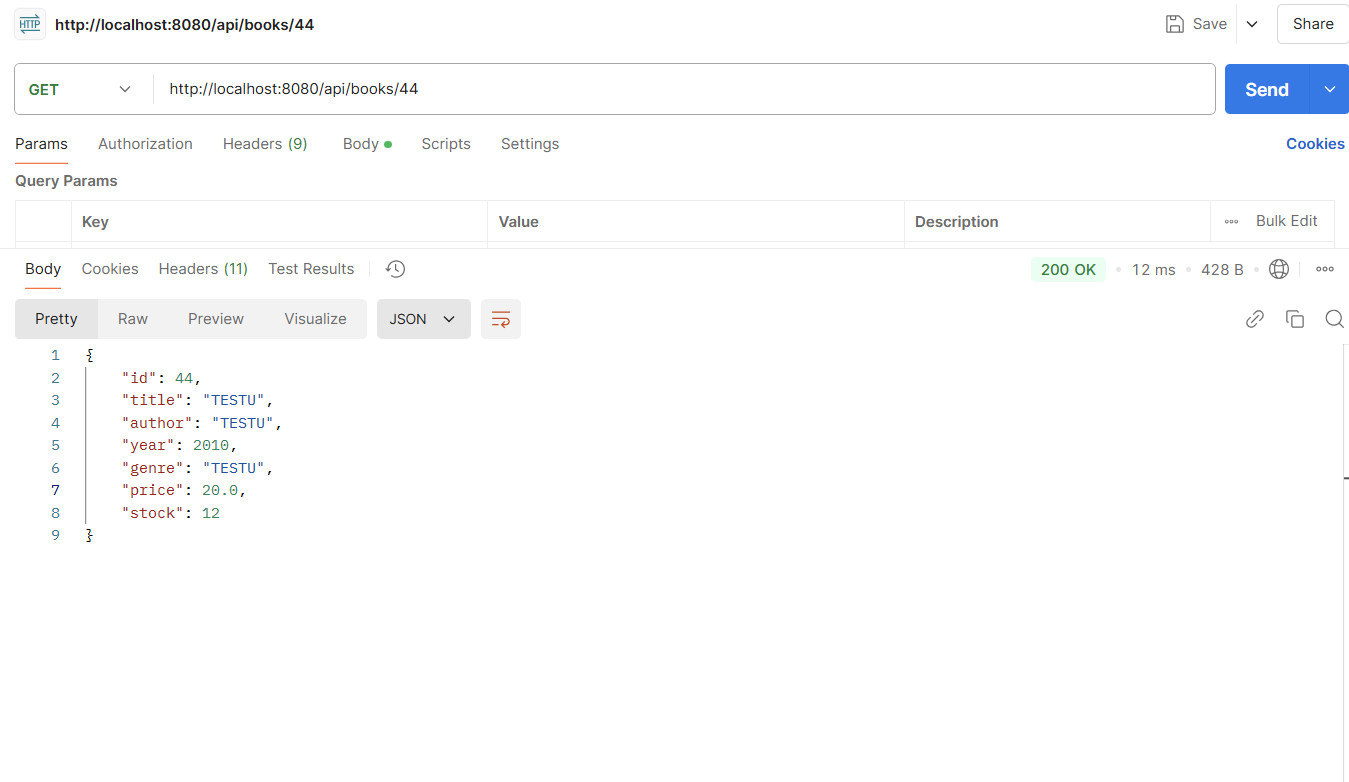


Iar o alta metoda ca PUT returneaza o eroare

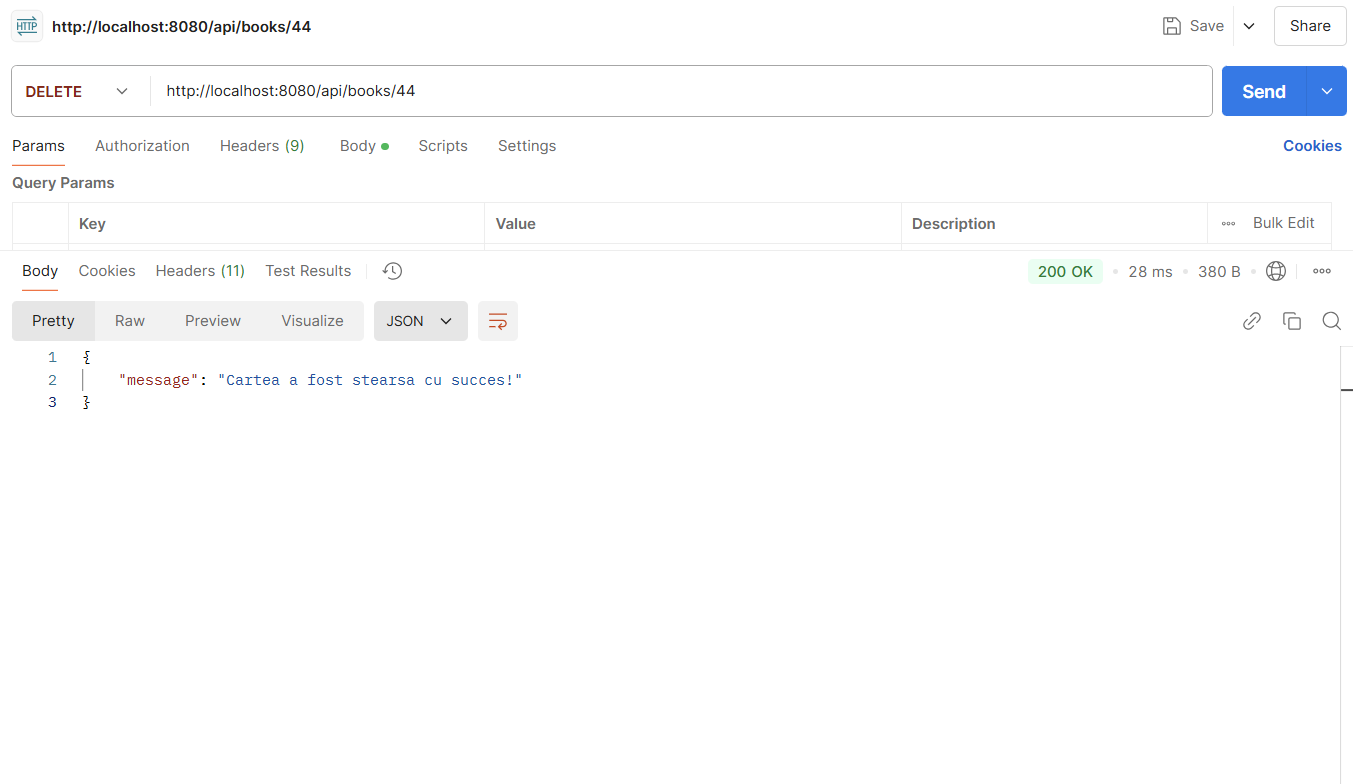


Pentru ruta /api/books/{id} GET returneaza ori eroare in caz ca id-ul nu exista in fisierul JSON

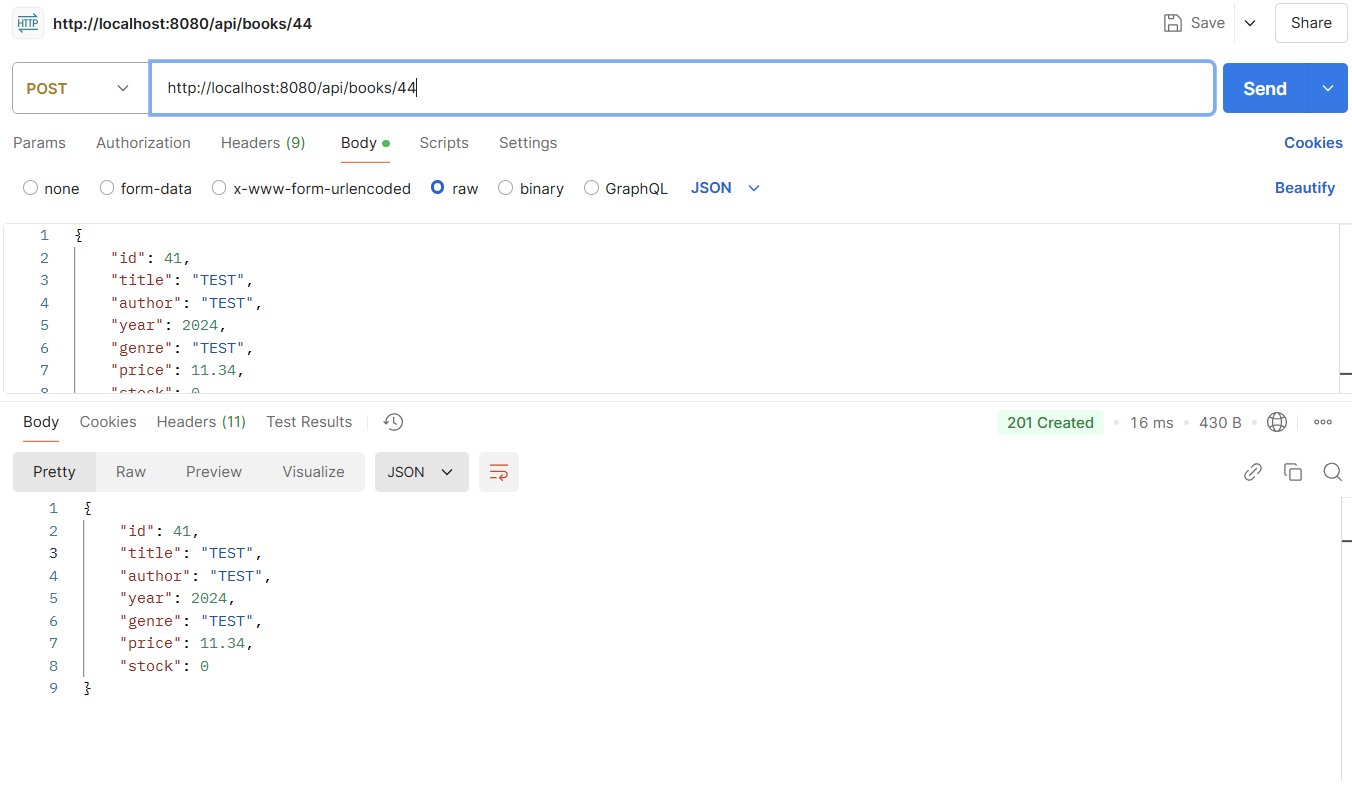
Sau fetch la cartea cu id-ul respectiv in caz ca exista



Metoda DELETE afiseaza mesajul de informare de mai jos daca operatia s-a realizat cu succes



Iar POST/PUT afiseaza argumentele trimise pe ruta dupa modificare (e.g.: am modificat cartea cu id 41 si am primit inapoi confirmarea ca modificarile au fost realizate)



**V. 5 idei despre cum poate fi imbunatatita aplicatia**

1. Datele sunt salvate intr-un fisier JSON acest lucru fiind util din punct de vedere al simplicitatii de stocare dar salvarea informatiilor intr-o baza de date sau server ar fi fost cel mai ideal plus ca un fisier json aduce si o parte de unstructured databases in the mix ceea ce nu este indicat cand vorbit de un tabel simplu ca cel prezentat.

2. De asemenea daca se adauga carti direct in fisierul JSON indexul curent nu va fi modificat deoarece acesta este setat sa se updateze cu adaugarea unei carti din frontend (aici se salveaza un index local al site-ului ce tine minte unde a ramas ultima carte) deci daca se adauga carti manual direct in fisier si dupa se doreste adaugarea unei carti noi de pe site aceasta va suprascrie un id deja existent care a fost adaugat manual in functie de ultimul id stocat pe site. Acest lucru se poate remedia prin verificarea fisierului la deschidere si setarea indexului la ultima valoare din fisier. Totusi aspectul acesta nu impiedica buna functionare deoarece nu s-a pus problema adaugarii datelor manul in fisier ci doar din interfata grafica.

3. Pe partea de frontend, cel mai evident lucru este ca o daca cu scalarea datelor tabelul va devenii mai lung ceea ce de la sine aduce in discutie o eroare pe partea de design: ca sa ajungi la form-ul *de sub tabel* (add/update) va fi nevoie de un scroll destul de lung. Acest lucru evidentiaza ca designul este bun atat timp cat aplicatia este la o dimesiune relativ redusa cand vorbim de numarul de carti din tabel. O solutie pentru ramp-up(scalarea) datelor ar fi mutarea form-ului de add/update deasupra tabelului sau intr-o parte cu acesta fapt ce ar elimina inconvenientul de mai sus.

4. Clasele de exceptii si handle-ul global sunt destul de complicat de inteles puteau sa fie mai simplificate cel putin pe partea de implementare din punctul de vedere al autorului si puteau sa aiba o functionalitate mai semificata pentru utilizator aceste clase la momentul actual doar servesc un ***nice touch*** partea de backend pentru debugging.

5. In proiect mai exista si o clasa de Config neimplementata care in teorie trebuie folosita pentru o parte de securitate dat fiind necesitatea in zilele noastre. Aceasta urma sa implementeze web tracking pentru request-uri astfel incat sa nu fie prea multe accesari permise intr-un timp prea scurt, sa blocheze XSS (CrossSite Scripting) si CSRF (CrossSite Request Forgery).