

MINISTERUL EDUCAȚIEI



UNIVERSITATEA TEHNICĂ

DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

PLATFORMĂ ECOMMERCE PENTRU PRODUSE HANDMADE – APLICAȚIE WEB CU JAVA SPRING BOOT ȘI VUE.JS

PROIECT DE DIPLOMĂ

Autor: **Paula Maria MOCAN**

Conducător științific: **Prof. dr. ing. Honoriu VĂLEAN**

2025



Vizat,

DECAN
Prof. dr. ing. Vlad MUREȘAN

DIRECTOR DEPARTAMENT AUTOMATICĂ
Prof. dr. ing. Honoriu VĂLEAN

Autor: **Paula Maria MOCAN**

**Platformă eCommerce pentru produse handmade – Aplicație web cu Java
Spring Boot și Vue.js**

1. **Enunțul temei:** Dezvoltarea unei aplicații web de comerț online, pentru comercializarea produselor artisanale, folosind Java împreună cu Spring Boot, Vue.js și MySQL.
2. **Conținutul proiectului:** Pagina de prezentare, Declarație privind autenticitatea proiectului, Sinteza proiectului, Cuprins, Capitolul 1: Introducere, Capitolul 2: Studiu bibliografic, Capitolul 3: Analiză, proiectare, implementare, Capitolul 4: Concluzii și direcții de dezvoltare, Bibliografie.
3. **Locul documentării:** Universitatea Tehnică din Cluj-Napoca
4. **Data emiterii temei:** 16.12.2024
5. **Data predării:** 11.07.2025

Semnătura autorului

Semnătura conducătorului științific



**Declarație pe proprie răspundere privind
autenticitatea proiectului de diplomă**

Subsemnatul(a) **Paula Maria MOCAN**
 , legitimat(ă) cu CI seria ZS nr. 130244, CNP
6020118261961

autorul lucrării:

Platformă eCommerce pentru produse handmade – aplicație web cu Java
Spring Boot și Vue.js

elaborată în vederea susținerii examenului de finalizare a studiilor de licență la **Facultatea de Automatică și Calculatoare**, specializarea **Automatică și Informatică Aplicată**, din cadrul Universității Tehnice din Cluj-Napoca, sesiunea iulie 2025 a anului universitar 2024-2025, declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, pe baza cercetărilor mele și pe baza informațiilor obținute din surse care au fost citate, în textul lucrării, și în bibliografie.

Declar, că această lucrare nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române și a convențiilor internaționale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în fața unei alte comisii de examen de licență.

În cazul constatării ulterioare a unor declarații false, voi suporta sancțiunile administrative, respectiv, *anularea examenului de licență*.

Data

11.07.2025

Prenume NUME

Paula Maria MOCAN



SINTEZA

proiectului de diplomă cu titlul:

Titlul lucrării

Autor: **Paula Maria MOCAN**

Conducător științific: **Prof. dr. ing. Honoriu VĂLEAN**

1. Cerințele temei: Realizarea unei aplicații web de e-commerce cu funcționalități de magazin online și un panou de administrare pentru managementul conținutului.
2. Soluții alese: O arhitectură decuplată cu un API REST în Spring Boot și un frontend de tip SPA (Single-Page Application) construit cu Vue.js 3 și Pinia.
3. Rezultate obținute: O platformă funcțională cu management de produse și categorii, autentificare bazată pe token-uri și o interfață de administrare completă.
4. Testări și verificări: Funcționalitatea a fost validată prin teste manuale bazate pe scenarii de utilizare pentru rolurile de client și administrator.
5. Contribuții personale: Integrarea unui stack tehnologic modern (Vue/Pinia/Spring), proiectarea unei arhitecturi modulare și implementarea unui sistem de autorizare eficient.
6. Surse de documentare: Documentațiile oficiale ale tehnologiilor (Vue, Spring), cărți de specialitate și articole despre arhitecturi REST și SPA.

Semnătura autorului

Semnătura conducătorului științific

Cuprins

1	INTRODUCERE.....	2
1.1	CONTEXT GENERAL	2
1.2	OBIECTIVE.....	2
1.3	SPECIFICAȚII	4
1.3.1	<i>Cerințe funcționale.....</i>	<i>4</i>
1.3.2	<i>Cerințe nefuncționale.....</i>	<i>5</i>
1.3.3	<i>Structuri de date principale.....</i>	<i>7</i>
1.3.4	<i>Limitări și asumptii.....</i>	<i>7</i>
2	STUDIUL BIBLIOGRAFIC	7
2.1.1	<i>Arhitectura monolitică vs. Arhitectura decuplată</i>	<i>8</i>
2.1.2	<i>Aplicații Multi-Page (MPA) vs. Single-Page (SPA)</i>	<i>8</i>
3	ANALIZĂ, PROIECTARE, IMPLEMENTARE.....	13
3.1.1	<i>Abordare iterativă (Agile)</i>	<i>14</i>
3.1.2	<i>Proiectarea „API-First”</i>	<i>14</i>
3.1.3	<i>Fluxul de dezvoltare</i>	<i>15</i>
3.2	ANALIZA ȘI PROIECTAREA BACKEND-ULUI	17
3.2.1	<i>Proiectarea bazei de date</i>	<i>17</i>
3.2.2	<i>Arhitectura stratificată Spring Boot.....</i>	<i>19</i>
3.2.3	<i>Proiectarea API-ului RESTful</i>	<i>35</i>
3.2.4	<i>Gestionarea excepțiilor</i>	<i>37</i>
3.3	IMPLEMENTAREA COMPONENTEI FRONTEND.....	38
3.3.1	<i>Structura proiectului și organizarea fișierelor</i>	<i>39</i>
3.3.2	<i>Managementul centralizat al stării Pinia</i>	<i>41</i>
3.3.3	<i>Interceptarea și centralizarea apelurilor API</i>	<i>44</i>
3.3.4	<i>Rutarea avansată și securizarea navigării</i>	<i>45</i>
3.3.5	<i>Implementarea componentelor reactive „inteligente”</i>	<i>48</i>
3.3.6	<i>Implementarea componentelor vizuale cheie</i>	<i>49</i>
4	CONCLUZII.....	54
4.1	REZULTATE OBȚINUTE.....	54
4.2	DIRECȚII DE DEZVOLTARE.....	55
5	BIBLIOGRAFIE.....	56

1 Introducere

1.1 Context general

Într-o eră dominată de tehnologie și inovație digitală, comerțul electronic (cunoscut ca și eCommerce) a cunoscut o evoluție semnificativă în ultimele zeci de ani, oferind oportunități atât companiilor mari, cât și micilor producători independenți de a-și prezenta produsele în spațiul virtual. O dată cu extinderea accesului la internet și evoluția continuă a platformelor digitale, majoritatea comercianților aleg să își mute activitatea în mediul online. Un accent important se pune pe produsele realizate manual, care presupun o dexteritate de creare, personalizare și ambalare. Producătorii acestui tip de produse au câștigat popularitate pe platforme precum Etsy sau Amazon Handmade în străinătate în mod special, dar și prin magazine online dezvoltate independent.

Un astfel de magazin online presupune existența unui sistem informatic care să permită următoarele:

- Prezentarea în prealabil a produselor;
- Gestionarea comenzilor;
- Administrarea conturilor de utilizator;
- Procesarea plăților;
- Funcții auxiliare precum wishlist, filtrare după categorie sau notificări.

Lucrarea de față presupune procesul de realizare și dezvoltare a unei aplicații web de tip eCommerce, destinată comercializării produselor handmade. Partea de backend a platformei web este implementat în limbajul Java, folosind Spring Boot ca și framework. Baza de date este creată automat prin apelarea serverului de backend și se poate analiza prin intermediul sistemului MySQL. De asemenea, frontend-ul este realizat folosind JavaScript, cu ajutorul framework-ului Vue.js. Această alegere va garanta proiectului o separare clasă între logică și prezentare, precum și o experiență de utilizare intuitivă.

Lucrarea este structurată în șase capitole care abordează câte o etapă importantă în dezvoltarea aplicației. Primul capitol conține o introducere în temă, motivația alegerii și obiectivele propuse. Capitolul al doilea se axează pe un studiu bibliografic detaliat al tehnologiilor utilizate, precum Java, Spring Boot, Vue.js și MySQL. Capitolul al treilea este dedicat analizei, proiectării și implementării aplicației. Capitolul patru evidențiază rezultatele obținute și propune direcții de dezvoltare viitoare. Ultimul capitol include bibliografia utilizată.

1.2 Obiective

Lucrarea de față își propune implementarea și extinderea unei platforme de comerț electronic complet funcționale, numită Craftique, destinată comercializării de produse

realizate manual. Scopul este de a construi o aplicație web modernă, rapidă și ușor de utilizat, concepută pentru a sprijini accesul clienților la produse locale autentice.

Pentru realizarea acestui proiect, a fost necesară stabilirea unei serii de obiective bine definite, care vizează atât componenta de server (backend), cât și interfața destinată utilizatorului (frontend). Între cele două părți există o interacțiune constantă, mediată prin apeluri HTTP către un set de endpoint-uri RESTful, care permit transmiterea și preluarea datelor într-un mod standardizat și securizat. Baza de date relațională are un rol esențial în arhitectura aplicației, având responsabilitatea de a stoca informațiile fundamentale, precum datele utilizatorilor, detaliile produselor, conținutul coșului de cumpărături și al listei de dorințe. Relaționarea dintre frontend și backend este proiectată astfel încât să asigure consistența datelor și o experiență ușoară și rapidă pentru utilizatorul final, ajutând la funcționarea coerentă și eficientă a întregului sistem.

Obiectivele principale ale lucrării sunt:

- **Dezvoltarea unei arhitecturi decuplate:** Construirea aplicației de modelul Single-Page Application (SPA) cu Vue.js pentru frontend, complet separat de un API RESTful dezvoltat în Java și Spring Boot pentru backend.
- **Implementarea unui backend robust:** Crearea unui API REST logic și intuitiv, cu o arhitectură stratificată (Controller, Service, Repository) și stocarea datelor într-o bază de date prin intermediul Spring Data JPA.
- **Construirea unei interfețe frontend reactive:** Utilizarea Vue.js 3 și a Composition API pentru a dezvolta o interfață de utilizator modulară, alcătuită din componente reutilizabile, ușor de întreținut și optimizate pentru performanță.
- **Managementul centralizat al stării:** Implementarea bibliotecii Pinia pentru a gestiona starea globală a aplicației (autentificare, coș de cumpărături), facilitând partajarea coerentă a datelor între componente și menținerea unui flux logic în actualizarea acestora.
- **Crearea unui sistem de roluri (Client vs. Admin):** Implementarea unui sistem de autorizare care oferă o experiență diferită pentru utilizatorii cu rol de administrator, prin securizarea vizuală a interfeței (afișarea sau ascunderea condiționată a elementelor de management) și restricționarea accesului la rutele de administrare prin verificarea rolului utilizatorului în cadrul mecanismelor de navigație din Vue Router.
- **Realizarea unui flux eCommerce complet (de la navigare la finalizarea comenzii):** Implementarea funcționalităților esențiale ale aplicației, precum vizualizarea produselor și a categoriilor, adăugarea în coșul de cumpărături și în lista de dorințe, alături de un sistem de înregistrare și autentificare a utilizatorilor bazat pe token-uri, precum și o pagină de Checkout care simulează finalizarea comenzii prin golirea coșului și afișarea unui mesaj de confirmare.

În ansamblu, aceste obiective urmăresc, pe lângă construcția unei aplicații funcționale, o explorare practică a modului în care tehnologiile moderne precum Vue.js 3,

Pinia și Spring Boot pot fi unite și integrate pentru a crea soluții software robuste, scalabile și aliniate cu standardele actuale ale industriei.

1.3 Specificații

Subiectul acestei lucrări, aplicația Craftique, este o platformă web de comerț electronic destinată sprijinirii vânzării de produse artisanale. Arhitectura sistemului este proiectată pentru a deservi două categorii principale de utilizatori: clienții, care beneficiază de funcționalități pentru navigarea, căutarea și achiziționarea produselor, și administratorii, responsabili de gestionarea și actualizarea conținutului magazinului, inclusiv administrarea produselor și categoriilor aferente.

Platforma va fi dezvoltată ca o aplicație web modernă (Single-Page Application), cu o arhitectură decuplată ce constă într-un frontend interactiv construit în Vue.js și un backend robust, bazat pe servicii, implementat în Java cu framework-ul Spring Boot. Obiectivul principal este de a oferi o experiență de utilizare rapidă, sigură și intuitivă, similară cu cea a platformelor de eCommerce consacrate.

1.3.1 Cerințe funcționale

Cerințele funcționale descriu acțiunile și funcționalitățile specifice pe care sistemul trebuie să le îndeplinească. Acestea definesc comportamentul aplicației din perspectiva utilizatorului.

- Managementul autentificării și al rolurilor

Sistemul oferă funcționalități de gestionare a autentificării și autorizării utilizatorilor. Utilizatorii noi au posibilitatea de a-și crea un cont prin completarea unui formular de înregistrare ce solicită numele, prenumele, adresa de e-mail și o parolă, care trebuie confirmată. După înregistrare, utilizatorii se pot autentifica folosind adresa de e-mail și parola asociată contului. În legătură cu gestionarea accesului, aplicația deosebește două niveluri de permisiuni: Client sau Administrator. Fiecărui cont creat prin intermediul interfeței publice îi va fi atribuit implicit rolul de Client. În schimb, pentru a menține un nivel ridicat de securitate în controlul accesului administrativ, conturile de tip Administrator nu pot fi create din interfața aplicației, ci doar prin inserare directă în baza de date de către dezvoltator sau operator autorizat. De asemenea, utilizatorii autentificați pot opta în orice moment pentru deconectare din cont printr-o funcționalitate dedicată.

- Funcționalități pentru Client (utilizator public și autentificat)

Aplicația oferă funcționalități esențiale atât pentru vizitatori neautentificați, cât și pentru utilizatorii logați. Astfel, orice utilizator, indiferent de autentificare, poate să acceseze pagina principală, secțiunea „Despre noi”, lista completă a categoriilor disponibile, precum și pagina corespunzătoare produselor existente în sistem. De asemenea, este posibilă filtrarea produselor în funcție de o anumită categorie selectată de utilizator, fiecare categorie având alocată o pagină proprie, special concepută pentru a afișa doar produsele aferente acesteia, pentru a facilita accesul rapid și eficient al utilizatorilor la informațiile dorite.

Pentru fiecare produs disponibil în cadrul aplicației, este oferită o pagină individuală de detalii, concepută special pentru a furniza utilizatorilor informații esențiale și relevante. Această pagină include, în mod explicit, numele produsului, o descriere care evidențiază caracteristicile și utilitatea acestuia, prețul de vânzare, precum și o imagine reprezentativă menită nu doar să ofere o perspectivă vizuală clară asupra produsului, ci și să contribuie la creșterea interesului clientului față de achiziționarea acestuia.

O dată cu autentificarea în aplicație, utilizatorii vor beneficia de funcționalități suplimentare pentru îmbunătățirea experienței de cumpărare. Printre acestea se numără posibilitatea de a adăuga sau de a elimina produse din propriul coș de cumpărături, ajutând astfel personalizarea comenzilor în funcție de preferințe. De asemenea, utilizatorii pot gestiona o listă de produse favorite (wishlist), o listă unde pot fi salvate articolele de interes pentru o eventuală achiziție viitoare. În orice moment, aceștia au acces la conținutul actual al coșului, care este afișat împreună cu un sumar detaliat al produselor selectate și o estimare a costului total.

Nu în ultimul rând, aplicația include o pagină de Checkout care simulează procesul de finalizare a unei comenzi, oferind clientului o practică cât mai apropiată de cea a unui magazin online real. După completarea informațiilor cerute (numele și prenumele clientului, adresa de livrare și numărul de telefon), sistemul golește automat conținutul coșului de cumpărături, marcând astfel plasarea cu succes a comenzii.

- **Funcționalități pentru Administrator**

Utilizatorii cu rol de Administrator au posibilitatea, spre deosebire de clienții obișnuiți, de a accesa exclusiv un panou de administrare dedicat, securizat și inaccesibil celorlalți utilizatori. În cadrul acestui panou, administratorii au control complet asupra gestionării categoriilor și produselor disponibile în magazin. Aceștia pot adăuga categorii noi, specificând informații importante precum denumirea categoriei, o descriere relevantă și o imagine reprezentativă care să o identifice vizual în cadrul aplicației. De asemenea, aceștia pot vizualiza lista completă a categoriilor, modifica detaliile acestora sau pot elimina categorii existente. În mod similar, administratorii pot coordona produsele din magazin, efectuând operații CRUD: adăugarea de produse noi, alocarea acestora la categoriile corespunzătoare, vizualizarea întregului catalog de produse, actualizarea informațiilor despre produse și ștergerea celor care nu mai sunt disponibile.

În ceea ce privește asocierea imaginilor la produse și categorii, sistemul nu permite încărcarea de fișiere din interfață, ci oferă posibilitatea selectării dintr-un set de imagini predefinite stocate local în aplicația frontend, într-un director dedicat.

1.3.2 Cerințe nefuncționale

Cerințele nefuncționale definesc caracteristici de calitate ale sistemului, precum performanța, securitatea, scalabilitatea sau ușurința în utilizare. Spre deosebire de cerințele funcționale, acestea nu indică ce elemente funcționale oferă sistemul, ci specifică modul în care acestea trebuie implementate și comportamentul general al aplicației în diverse condiții.

- Performanță

Pentru a asigura o experiență rapidă și eficientă pentru utilizator, aplicația solicită mai multe cerințe de performanță. Timpul de încărcare inițial al interfeței frontend nu trebuie să depășească trei secunde pe o conexiune la internet de viteză medie. O dată încărcată pagina, navigarea între paginile aplicației, construită ca o aplicație de tip Single-Page Application, trebuie să fie experimentată ca o tranziție aproape instantanee, cu un timp de răspuns vizual de sub 500 milisecunde. În același timp, răspunsurile API livrate de backend pentru operațiunile de citire (GET) trebuie să se limiteze la un timp de răspuns de maximum 200 milisecunde în condiții de testare locală.

- Securitate

Comunicarea dintre frontend și backend se realizează printr-un mecanism de autentificare bazat pe JSON Web Tokens (JWT). După autentificare, utilizatorul primește un token care este trimis automat în antetul fiecărei cereri care necesită autentificare. Acest token confirmă identitatea utilizatorului, însă nu conține informații despre rolul acestuia. Verificarea drepturilor de acces, precum distingerea dintre utilizator obișnuit sau administrator, se realizează pe baza unui atribut boolean (`isAdmin`) asociat fiecărui cont de utilizator. Acest atribut este verificat atât în frontend, pentru a gestiona afișarea elementelor specifice panoului de administrare, cât și în backend, pentru a valida accesul la rutele administrative. Rolul de administrator poate fi stabilit doar prin modificarea directă a bazei de date, mai exact schimbarea valorii `isAdmin`, nefiind un mecanism de promovare sau retrogradare prin interfața aplicației. Endpoint-urile destinate funcționalităților de administrare (de exemplu: rutele de tip `/admin/**`) vor fi restricționate și accesibile exclusiv utilizatorilor autentificați cu rol de Administrator, orice încercare de acces neautorizat fiind respinsă de sistem.

- Utilizabilitate

Interfața aplicației trebuie să fie intuitivă și ușor de utilizat, atât pentru utilizatorii de tip clienți, cât și pentru administratori, asigurând o experiență de navigare fluidă. Aspectul vizual este conceput pentru a fi responsiv, astfel încât aplicația să se adapteze corect la o gamă variată de dispozitive, de la desktop-uri la tablete și telefoane mobile. În plus, utilizatorul trebuie să primească un răspuns vizual în timp real pentru acțiunile efectuate, prin mesaje de confirmare, eroare sau indicatoare de încărcare, contribuind astfel la o interacțiune clară și predictibilă cu sistemul. Acest lucru este realizat prin integrarea componentei SweetAlert2.

- Fiabilitate

Sistemul este proiectat pentru a gestiona erorile API într-un mod care pune pe primul loc experiența utilizatorului. Din acest motiv, în cazul unei operațiuni eșuate, interfața va afișa un mesaj prietenos și informativ, fără a expune detalii tehnice inutile. În același timp, eroarea completă este înregistrată în consola de dezvoltare pentru a putea fi analizată în amănunt. Această abordare asigură integritatea aplicației pe tot parcursul utilizării, aceasta rămânând stabilă și complet funcțională, prevenind blocajele și permițând utilizatorului să continue navigarea.

- **Mentenabilitate**

Calitatea codului, atât cel de frontend cât și cel de backend, este fundamentală pentru longevitatea proiectului. Structura acestuia urmează principii de claritate și modularitate: fiecare funcționalitate este organizată în componente logice, independente și reutilizabile. Se respectă convențiile de stil specifice fiecărei tehnologii, asigurând o bază de cod consistentă și predictibilă. Această abordare strategică ușurează nu doar o depanare eficientă, ci și o mentenanță simplificată și o scalabilitate pe termen lung.

1.3.3 Structuri de date principale

Pentru a îndeplini cerințele funcționale, au fost definite următoarele modele de date esențiale:

- **Users:** id, firstName, lastName, email, password, isAdmin (boolean).
- **Category:** id, categoryName, description, imageUrl.
- **Product:** id, name, description, price, imageUrl, categoryId (cheie străină către tabelul Category).
- **Cart/Wishlist:** structuri care leagă un User de unul sau mai multe obiecte Product, incluzând și quantity pentru coșul de cumpărături.
- **AuthenticationToken:** id, token (String), createdAt, userId (cheie străină către tabelul Users)

1.3.4 Limitări și asumptii

Proiectul, în forma sa actuală, prezintă următoarele limitări:

- Procesul de plată este simulat. Nu este integrat un procesator de plăți real (Stripe, Paypal, etc.).
- Comenzile nu sunt salvate persistent în baza de date după finalizarea comenzii.
- Sistemul nu include funcționalități avansate de eCommerce, precum recenzii de produse, coduri de discount sau filtrare sau sortare avansată a produselor. Acestea sunt considerate direcții viitoare de dezvoltare.

2 Studiul bibliografic

2.1. Introducere în arhitecturile web moderne

Dezvoltarea aplicațiilor web a cunoscut o evoluție accelerată, trecând de la modele arhitecturale simple, rigide, greu de întreținut, la paradigme moderne, complexe, care pun accent pe performanță, scalabilitate, modularitate și o experiență a utilizatorului îmbunătățită. O înțelegere a acestor arhitecturi este importantă pentru a justifica alegerile de proiectare realizate în cadrul proiectului Craftique. În literatura de specialitate, două modele arhitecturale domină discuțiile: arhitectura monolitică, unde toate componentele aplicației sunt interconectate și distribuite împreună, și arhitectura bazată pe

microservicii, care presupune divizarea aplicației în servicii independente, fiecare având propriile sale responsabilități bine definite și capacitatea de a fi scalat individual.

La nivelul interfeței cu utilizatorul, stilurile de implementare se împart în două categorii mari: aplicațiile Single-Page (SPA), care încarcă o singură pagină web și actualizează conținutul dinamic prin JavaScript, oferind o experiență fluidă, similară aplicațiilor desktop, și aplicațiile Multi-Page (MPA), unde fiecare interacțiune majoră cu utilizatorul duce la reîncărcarea unei noi pagini de pe server.

Alegerea între aceste modele arhitecturale și stiluri de implementare determină direct performanța, timpul de dezvoltare, mentenabilitatea și scalabilitatea aplicației, fiind un aspect fundamental în construcția oricărei soluții web moderne.

2.1.1 Arhitectura monolitică vs. Arhitectura decuplată

În mod tradițional, multe aplicații web, inclusive cele de tip eCommerce, au fost dezvoltate folosind arhitectura monolitică. În acest model, toate componentele sistemului (interfața cu utilizatorul, logica de business și accesul la date) sunt integrate într-o singură unitate de cod, gestionată și distribuită ca un întreg. Framework-uri precum Ruby on Rails, Django sau versiunile clasice de Spring MVC au fost construite în jurul acestei strategii. Principalul avantaj al arhitecturii monolitice constă în simplitatea inițială a procesului de dezvoltare, întrucât întreaga aplicație este livrată ca un singur artefact executabil. Cu toate acestea, pe măsură ce aplicația evoluează și devine tot mai complexă, acest tip de model devine unul problematic, întrucât unul din dezavantajele semnificative este dificultatea de scalare, întreaga aplicație trebuind scalată, deși doar anumite componente sunt intens utilizate. Alte dezavantaje ale acestui model constau în rigiditatea tehnologică (limitarea în adoptarea unor tehnologii diferite pentru componente individuale) și creșterea dificultății în mentenanță și testare.

Ca răspuns la limitările arhitecturii monolitice, a fost adoptat pe scară largă modelul arhitecturii decuplate, deseori asociat cu microserviciile. Acest tip de arhitectură presupune separarea completă a componentelor de backend și frontend. Backend-ul expune un API (Application Programming Interface), de obicei de tip RESTful, iar frontend-ul (sau orice alt client, cum ar fi o aplicație mobilă sau desktop) consumă acest API pentru a accesa logica de business și datele aplicației. Separarea celor două aduce o flexibilitate ridicată: echipele pot lucra independent asupra celor două componente, utilizând tehnologii diferite și adaptate nevoilor specifice ale fiecărei părți.

2.1.2 Aplicații Multi-Page (MPA) vs. Single-Page (SPA)

Alegerea stilului de implementare al interfeței frontend reprezintă, de asemenea, o decizie crucială în arhitectura unei aplicații web. Modelul clasic utilizat este cel al aplicațiilor de tip Multi-Page Application. În cadrul acestui model, fiecare acțiune a utilizatorului care presupune accesarea unor date noi, cum ar fi navigarea către o altă secțiune printr-un link, generează o nouă solicitare către server. Acesta procesează cererea, construiește conținutul necesar și returnează o pagină HTML completă. Acest stil de implementare este apreciat pentru simplitatea sa, robustețea în funcționare și compatibilitatea nativă cu motoarele de căutare, datorită faptului că fiecare pagină are un URL unic și conținutul este pre-randat pe server.

Aplicațiile Single-Page reprezintă, pe de altă parte, structura modernă în dezvoltarea interfețelor web. O aplicație de acest tip va încărca inițial un singur fișier HTML și un pachet JavaScript, iar navigarea și interacțiunile utilizatorului sunt gestionate integral în browser, prin intermediul JavaScript-ului. Atunci când sunt necesare date noi, aplicația efectuează cereri asincrone către API-ul backend și actualizează doar porțiunile relevante ale paginii, evitând astfel reîncărcarea completă. Conform lui Adam Freeman, în lucrarea „Pro Angular 9”, această abordare „crează o experiență de utilizare mai fluidă și mai rapidă, similară cu cea a unei aplicații native”.

2.2. Analiza Framework-urilor Frontend moderne

Universul JavaScript este caracterizat de trei framework-uri majore, care domină scena actuală: React, Angular și Vue.js. Alegerea unuia dintre acestea are un impact considerabil asupra productivității echipei de dezvoltare, a performanței aplicației, precum și asupra ușurinței cu care aceasta poate fi întreținută și extinsă pe termen lung.

React, dezvoltat și susținut de Facebook, nu este un framework complet, este o bibliotecă concentrată în mod particular pe implementarea interfețelor de utilizator. Acesta este valorificat pentru performanța sa, datorită utilizării DOM-ului virtual, precum și pentru ecosistemul vast. Totuși, pentru a realiza o aplicație completă, este necesară integrarea unor biblioteci suplimentare, managementul stării și alte funcționalități, ceea ce poate conduce la fenomenul de „oboseală decizională” (decision fatigue).

Angular, dezvoltat și susținut de Google, este un framework complet, scris în TypeScript și cu o arhitectură prescriptivă, care impune un mod clar și structurat de organizare a codului. Acesta oferă o soluție integrată care include rutare, client HTTP și alte instrumente importante. Este un framework foarte stabil și bine structurat, fiind alegerea potrivită pentru proiecte enterprise complexe. Cu toate acestea, complexitatea sa și numărul mare de concepte noi pot fi o provocare pentru dezvoltatori, mai ales pentru începători. Curba de învățare este destul de abruptă, ceea ce înseamnă că poate dura mai mult timp până când un dezvoltator ajunge să stăpânească acest cadru și să îl folosească eficient.

Vue.js, creat de Evan You, este un framework progresiv care reușește să îmbine flexibilitatea oferită de React cu structura organizată specifică Angular. Este apreciat pentru curba sa de învățare prietenoasă, precum și pentru performanțele comparabile cu cele ale lui React. Ecosistemul oficial al Vue, care include instrumente precum Vue Router și Pinia, distribuie o soluție completă pentru dezvoltarea de aplicații, fără a compromite simplitatea sau flexibilitatea. O dată cu lansarea versiunii Vue 3, introducerea Composition API aduce îmbunătățiri semnificative în modul de organizare a logicii componentelor și în reutilizarea codului, ceea ce face ca acest cadru de dezvoltare să devină o alegere din ce în ce mai regulată pentru proiectele moderne.

2.3. Managementul stării în aplicațiile frontend

În mod gradual, o dată cu creșterea complexității aplicației SPA, gestionarea stării (state management) se transformă într-o dificultate semnificativă. Starea reprezintă informațiile care descriu aspectul și comportamentul aplicației într-un anumit moment (de exemplu: cine este utilizatorul autentificat sau ce produse se află în coșul de cumpărături). Într-o aplicație complexă, diverse componente, neînrudite ierarhic, pot avea nevoie să acceseze sau să modifice aceeași bucată de stare.

În aplicațiile frontend complexe, gestionarea eficientă a stării devine o provocare majoră. Una dintre problemele cele mai des întâlnite este prop drilling-ul, care reprezintă procesul prin care datele sunt transmise de la o componentă părinte la componentele copil pe mai multe niveluri, folosind proprietăți (props). Cu cât ierarhia componentelor devine mai adâncă, acest model devine ineficient, greu de urmărit și dificil de întreținut. Orice modificare a structurii poate necesita refactorizarea numeroaselor componente intermediare care doar „transportă” datele fără a le utiliza direct.

Pentru a adresa această problemă, a fost introdus conceptul de management centralizat al stării. Această abordare presupune stocarea stării sistemului într-un singur obiect global denumit store, care devine „sursa unică de adevăr” (single source of truth) pentru toate componentele aplicației. Acest model a fost adus în prim-plan inițial de arhitectura Flux, dezvoltată de Facebook, și ulterior perfecționat și implementat într-un mod riguros prin biblioteca Redux.

În acest sistem, componentele nu modifică direct starea, ci ele apelează funcții specializate numite acțiuni (actions), care vor declanșa modificări asupra stării prin intermediul unor funcții numite reduceri (reducers) sau mutații (mutations, în cazul unor framework-uri precum Vue.js). Acest flux unidirecțional al datelor contribuie la o logică previzibilă în cadrul aplicației, facilitează procesul de depanare prin mecanisme precum time-travel debugging sau inspectarea stării, și susține o separare clară între logica de business și interfața utilizatorului.

Mai mult decât atât, un store centralizat permite refolosirea și sincronizarea facilă a stării între componente distincte, reduce dependențele dintre acestea și îmbunătățește scalabilitatea aplicației pe măsură ce aceasta se dezvoltă. Biblioteci precum Redux (pentru React) sau Pinia și Vuex (pentru Vue.js) pun la dispoziție astfel de mecanisme moderne și eficiente pentru gestionarea stării într-un mod clar, coerent și testabil.

În cadrul ecosistemului Vue, Pinia a devenit biblioteca oficială recomandată pentru gestionarea stării aplicației. Aceasta a fost concepută de membrii echipei Vue ca un succesor modern al Vuex, venind cu o serie de perfecționări importante în ceea ce privește simplitatea, modularitatea și compatibilitatea cu sistemele moderne de tipare și instrumente de dezvoltare.

Spre deosebire de Vuex, care propunea o structură rigidă și o sintaxă frecvent încărcată și dificil de gestionat în aplicații mari, Pinia oferă o abordare mai prietenoasă cu utilizatorul, bazată pe funcții, și este complet compatibilă cu Composition API din Vue 3. Pinia permite definirea a multiple „store-uri”, adică module de stare independente, fiecare responsabil pentru o anumită parte logică a aplicației, cum ar fi autentificarea

utilizatorului, coșul de cumpărături, preferințele UI, etc. Această modularitate susține organizarea clară a codului și simplifică întreținerea și scalabilitatea aplicației.

Un alt avantaj semnificativ constă în suportul nativ pentru tipizare oferit de TypeScript, asigurând dezvoltarea unui cod mai sigur și mai ușor de întreținut în aplicații de mari dimensiuni. De asemenea, Pinia pune la dispoziție un suport pentru instrumente moderne de depanare (precum devtools), funcționalități precum reținerea automată a stării și o integrare adecvată cu biblioteci externe.

Fluxul de lucru în Pinia urmează aceleași principii ca în alte soluții de state management: componentele pot citi starea din store-uri și o pot modifica doar prin metode definite explicit (actions), asigurând astfel o separare clară între logica de business (server) și interfața de utilizator. Această abordare contribuie la dezvoltarea unor aplicații mai previzibile, mai ușor de testat și de întreținut pe termen lung.

2.4. Arhitectura Backend cu Spring Boot

Spring Boot este un cadru de dezvoltare open-source, conceput pe baza ecosistemului Spring, creat pentru a facilita dezvoltarea rapidă a aplicațiilor enterprise Java. Acesta pune la dispoziție o configurație implicită („convention over configuration”) și mecanisme automate de inițializare a componentelor, eliminând nevoia de configurări XML extinse și reducând considerabil timpul de dezvoltare.

Unul dintre elementele fundamentale ale framework-ului Spring Boot este componenta Spring MVC, care oferă suport pentru implementarea modelului arhitectural Model-View-Controller (MVC). Acest model facilitează separarea clară între logica de business, interfața de prezentare și gestionarea cererilor HTTP, contribuind astfel la o structură mai organizată și ușor de întreținut a aplicației. În cadrul acestuia, controllerele (adnotate cu `@RestController`) definesc endpoint-uri care primesc cereri HTTP, procesează logica de procesare și returnează răspunsuri, în general sub formă de obiecte JSON. Acest model este utilizat frecvent în dezvoltarea de API-uri RESTful, datorită separării clare între componentele aplicației, care permite o structură modulară, ușurează testarea unitară și sprijină extinderea sau întreținerea aplicației într-un model eficient.

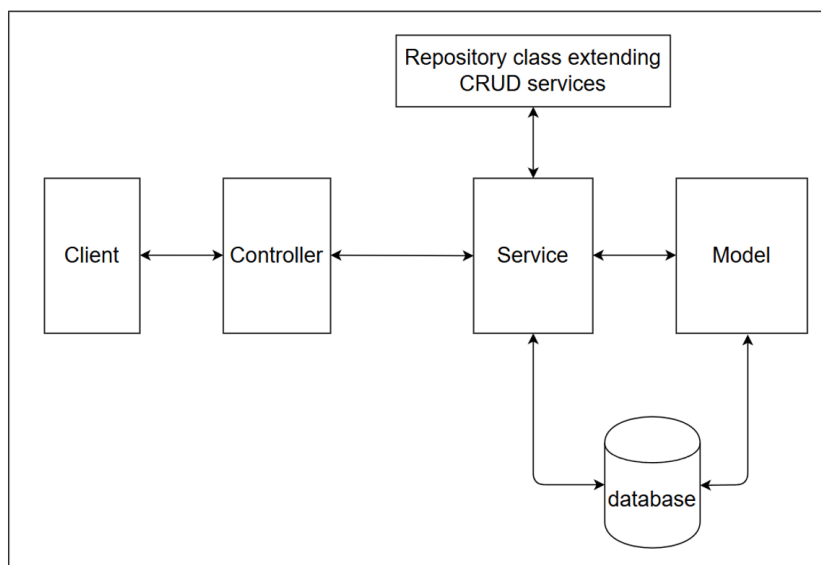


Figura 2.1. Arhitectura Spring

Pentru gestionarea persistenței datelor, Spring Boot oferă suport nativ pentru module precum Spring Data JPA, care furnizează un strat de abstractizare deasupra implementărilor JPA, cum ar fi Hibernate. Prin simpla definire a unor interfețe de tip Repository, dezvoltatorii beneficiază automat de metode CRUD (Create, Read, Update, Delete), fără a fi necesară scrierea manuală de cod SQL. Mai mult decât atât, interogările personalizate pot fi generate pe baza denumirii metodelor sau definite explicit cu ajutorul adnotărilor @Query. Această abordare accelerează procesul de dezvoltare, elimină necesitatea scrierii codului repetitiv și standard, și contribuie la o implementare mai clară, mai concisă și mai puțin predispusă la erori.

Un alt pilon important al ecosistemului Spring este Spring Security, un framework dedicat autentificării și autorizării. Acesta oferă suport complet pentru gestionarea sesiunilor, protecția contra atacurilor comune (inclusiv CSRF și XSS), precum și integrarea cu protocoale moderne precum OAuth2 sau JWT (JSON Web Tokens). Spring Security este modular și extensibil, ceea ce îl face potrivit atât pentru aplicații simple, cât și pentru sisteme complexe la scară largă.

Prin integrarea armonioasă a unor componente cheie precum Spring MVC, Spring Data JPA și Spring Security, cadrul de dezvoltare Spring Boot oferă o platformă completă pentru implementarea de aplicații web robuste, scalabile și ușor de întreținut. Fiecare componentă acoperă un aspect important al arhitecturii aplicației: Spring MVC asigură gestionarea logicii de prezentare și a cererilor HTTP, Spring Data JPA simplifică persistența datelor și interacțiunea cu bazele de date relaționale, iar Spring Security oferă un nivel ridicat de protecție și control al accesului. Această abordare modulară, combinată cu o configurare minimă și un ecosistem bogat de extensii, face din Spring Boot o alegere frecventă atât în mediul enterprise, cât și în cel academic, unde este apreciat pentru claritatea structurii și ușurința de utilizare.

2.5. Concluzii ale analizei bibliografice

Analiza studiului bibliografic al tehnologiilor web evidențiază clar tendința spre adoptarea unor arhitecturi moderne, decuplate, care separă responsabilitățile între frontend și backend. În particular, arhitectura de tip Single-Page Application s-a impus ca soluția preferată pentru proiectele de eCommerce datorită experienței îmbunătățite oferite utilizatorilor, prin navigare rapidă și interactivitate fluidă, fără reîncărcări complete ale paginii.

În ceea ce privește partea de frontend, framework-ul Vue.js 3, împreună cu noul său Composition API, oferă un model flexibil și modular pentru organizarea codului, facilitând dezvoltarea componentelor reutilizabile și gestionarea eficientă a logicii aplicației. În completarea acestuia, Pinia asigură un management centralizat și simplificat al stării, permițând sincronizarea ușoară a datelor între componente și un control clar al fluxurilor de date.

Pe de altă parte, pentru backend, utilizarea unui API REST construit cu Spring Boot asigură o infrastructură robustă și scalabilă, capabilă să gestioneze cereri multiple simultane și să servească datele necesare frontend-ului în mod eficient. Ecosistemul Spring oferă suport extins pentru persistența datelor, securitate și configurare automată, ceea ce accelerează procesul de dezvoltare și asigură o arhitectură solidă.

Prin combinarea acestor tehnologii, proiectele de tip eCommerce pot beneficia de un echilibru optim între performanță, scalabilitate și productivitate, răspunzând atât nevoilor utilizatorilor finali, cât și celor ale echipelor de dezvoltare. Astfel, deciziile de proiectare adoptate sunt în acord cu cele mai bune practici actuale din industrie și reflectă tendințele tehnologice relevante pentru dezvoltarea aplicațiilor web moderne.

3 Analiză, proiectare, implementare

Acest capitol constituie nucleul prezentei lucrări, detaliind etapele concrete de analiză, proiectare, implementare și testare a platformei de eCommerce Craftique. Secțiunile ce urmează vor prezenta metodologia de dezvoltare aleasă, deciziile arhitecturale, structurile de date, implementare tehnică a componentelor cheie atât pentru backend, cât și pentru frontend, și, în final, validarea funcționalității sistemului prin scenarii de testare. Scopul este de a oferi o descriere completă și reproductibilă a procesului de dezvoltare, care să reflecte aplicarea practică a conceptelor teoretice discutate anterior.

3.1. Metodologia de proiectare și dezvoltare

Procesul de realizare a aplicației a urmat o metodologie de dezvoltare modernă, orientată spre livrarea incrementală și adaptabilă a funcționalităților. Luând în considerare natura aplicației, o platformă eCommerce cu interacțiuni între client și server, s-a ales o abordare iterativă, inspirată din principiile metodologiei Agile. Deși proiectul a

fost dezvoltat individual, separarea clară a sarcinilor pe module și funcționalități a permis o organizare eficientă și o evoluție controlată a codului.

3.1.1 Abordare iterativă (Agile)

Metodologia Agile presupune o dezvoltare incrementală, bazată pe cicluri scurte de livrare (numite sprinturi), fiecare având ca scop dezvoltarea și validarea unui subset clar de funcționalități. Această abordare permite adaptarea rapidă la schimbări, testarea timpurie a componentelor și reducerea apariției de riscuri tehnice.

Chiar dacă proiectul nu a implicat o echipă de dezvoltare în sens tradițional, principiile Agile au fost respectate prin împărțirea muncii în „sprinturi conceptuale”, fiecare concentrat pe o componentă principală a aplicației:

- **Sprintul 1:** Implementarea funcționalităților de autentificare și înregistrare a utilizatorilor, definirea modelului de utilizator și configurarea inițială a bazei de date.
- **Sprintul 2:** Managementul categoriilor de produse: crearea modelului de categorie, interfața de administrare, afișarea produselor pe categorii.
- **Sprintul 3:** Adăugarea și gestionarea produselor: CRUD complet pentru produse, încărcare de imagini, afișare pe homepage și în paginile dedicate.
- **Sprintul 4:** Implementarea coșului de cumpărături și a wishlist-ului: stocarea locală a produselor selectate, interacțiunea cu backend-ul.
- **Sprintul 5:** Integrarea funcționalităților și testarea completă a fluxului de comandă.

Această abordare a oferit claritate și control asupra evoluției aplicației, precum și o mai bună prioritzare a sarcinilor în funcție de complexitate și dependențe.

3.1.2 Proiectarea „API-First”

Un aspect fundamental în dezvoltarea sistemului a fost aplicarea abordării API-First, care presupune proiectarea și definirea interfețelor backend (API-urile REST) înainte de implementarea interfeței vizuale (frontend-ul).

Această metodologie aduce multiple avantaje:

- Separarea clară a responsabilităților între client și server;
- Facilitarea dezvoltării paralele: chiar dacă frontend-ul și backend-ul au fost dezvoltate de aceeași persoană, această separare permite ca cele două părți să fie dezvoltate, testate și extinse independent;
- Standardizare: toate interacțiunile dintre componentele aplicației sunt bine definite, consistente și documentate;
- Ușurință în testare: endpoint-urile API pot fi testate individual, folosind instrumente precum Postman, înainte ca interfața grafică să le utilizeze.

În cadrul proiectului, înainte de scrierea efectivă a codului frontend, au fost definite structurile de date returnate de backend, precum și rutele HTTP necesare (de exemplu: GET /list, POST /signin, DELETE /delete/{cartItemId}). Acest lucru a permis o integrare

simplă și previzibilă între cele două părți ale aplicației, reducând semnificativ timpul necesar depanării sau adaptării codului la schimbări.

3.1.3 Fluxul de dezvoltare

Dezvoltarea aplicației a urmat o structură clar definită, organizată în pași logici, fiecare bazându-se pe rezultatele pasului anterior. Această organizare secvențială a permis un control mai bun asupra întregului proces, precum și o testare și validare riguroasă a fiecărei etape.

a) Configurarea mediului de dezvoltare

Primul pas a constat în pregătirea mediului de lucru, atât pentru backend, cât și pentru frontend. Pentru server, s-a utilizat IntelliJ IDEA ca IDE principal și Java 17, împreună cu Spring Boot. Baza de date aleasă a fost MySQL, iar pentru accesarea acesteia am folosit MySQL Workbench. Pe partea de frontend, am instalat Node.js și npm, care sunt indispensabile pentru gestionarea dependențelor JavaScript, precum și Vue CLI, util pentru generarea structurii aplicației Vue. Codul frontend a fost scris în Visual Studio Code.

Această configurare a oferit un mediu coerent de dezvoltare, în care s-a putut lucra eficient cu ambele părți ale aplicației (client și server), în mod paralel.

b) Modelarea bazei de date și a entităților JPA

O dată stabilite cerințele aplicației, următorul pas a fost modelarea logică și fizică a bazei de date. Au fost identificate entitățile precum: User, Product, Category, Cart, Wishlist și am definit atributele corespunzătoare fiecăreia.

Aceste entități au fost apoi implementate în cod folosind Java Persistence API (JPA), împreună cu Hibernate ca furnizor (provider). Anotările precum @Entity, @Table, @Id, @GeneratedValue, @OneToMany, @ManyToOne au fost utilizate pentru a configura maparea dintre clasele Java și tabelele bazei de date.

Relațiile dintre entități (de exemplu, un produs aparține unei categorii, un utilizator are un wishlist) au fost configurate corespunzător, asigurând integritatea datelor și optimizarea interogărilor.

c) Implementarea logicii de business

Stratul de logică de business a fost organizat în clase de tip Service, care conțin metodele principale pentru manipularea datelor. Fiecare entitate are un serviciu asociat (de exemplu, ProductService, UserService) care interacționează cu baza de date prin intermediul interfețelor de tip Repository, furnizate de Spring Data JPA.

Aceste servicii conțin logica pentru operațiuni precum: adăugarea unui produs în coșul de cumpărături sau în lista de dorințe, căutarea de produse după categorie, autentificarea utilizatorului, validări de date, gestionarea coșului și multe altele. Astfel, logica aplicației este izolată de stratul de acces la date și de cel de prezentare, respectând principiul separării responsabilităților.

d) Expunerea logicii prin API-uri REST

Funcționalitățile din servicii au fost puse la dispoziția frontend-ului prin intermediul unor controller-e REST. Acestea sunt clase Java adnotate cu `@RestController`, în care se definesc metode mapate la rute HTTP (`@GetMapping`, `@PostMapping`, etc.).

Fiecare metodă din controller corespunde unui endpoint, care primește cereri HTTP, procesează datele (cu ajutorul serviciilor) și returnează răspunsuri în format JSON. De exemplu:

```
@GetMapping("/products/list")
public ResponseEntity<List<ProductDto>> getProducts() {
    List<ProductDto> products = productService.getAllProducts();
    return new ResponseEntity<>(products, HttpStatus.OK);
}
```

Această structură RESTful a permis o comunicare standardizată și clară între frontend și backend.

e) Testarea endpoint-urilor cu Postman

Înainte de integrarea cu frontend-ul, toate endpoint-urile au fost testate individual cu ajutorul instrumentului Postman. Pentru fiecare metodă definită într-un controller, s-au verificat:

- Validarea răspunsurilor (format JSON, câmpuri corecte);
- Codurile de răspuns HTTP (200 OK, 404 Not Found, 400 Bad Request, etc.);
- Comportamentul în scenarii de eroare (date lipsă, autentificare eșuată);
- Răspunsul la date incorecte sau lipsă.

Această etapă a fost extrem de importantă pentru identificarea timpurie a erorilor și pentru asigurarea funcționalității corecte a API-ului.

f) Dezvoltarea componentelor frontend cu Vue.js

După validarea backend-ului, s-a trecut la dezvoltarea interfeței vizuale. Cu ajutorul Vue.js 3 și a sistemului său de componente, fiecare pagină a aplicației (de exemplu: pagina de acasă, lista de produse, autentificare, coș de cumpărături) a fost implementată sub forma unei componente independente.

Fiecare componentă a fost construită folosind Composition API, ceea ce a permis o mai bună organizare a codului, reutilizarea logicii și o structură clară. Designul paginilor a fost realizat cu ajutorul CSS și framework-uri UI.

g) Gestionarea stării aplicației cu Pinia

Pentru a evita trecerea repetată a datelor între componente (prop drilling), s-a utilizat Pinia, soluția oficială de management al stării în ecosistemul Vue 3. S-au definit mai multe store-uri, fiecare responsabil pentru o anumită funcționalitate: unul pentru utilizator (autentificare, rol), unul pentru coș, altul pentru lista de dorințe etc.

Aceste store-uri sunt reactive și permit accesul la datele centrale ale aplicației din orice componentă, reducând semnificativ complexitatea logicii frontend.

h) Integrarea frontend-backend prin Axios

Comunicarea dintre interfața vizuală și server s-a realizat cu Axios, o bibliotecă JavaScript pentru trimiterea cererilor HTTP. Din componentele Vue, s-au efectuat apeluri asincrone către endpoint-urile API pentru operațiuni precum:

- Preluarea produselor (GET /products);
- Autentificarea utilizatorului (POST /signin);
- Adăugarea produselor în coșul de cumpărături (POST /cart/add);
- Ștergerea unui produs din lista de dorințe (DELETE /wishlist/delete/{productId})

Răspunsurile au fost apoi procesate și afișate utilizatorului în interfață, în mod reactiv.

i) Implementarea securității aplicației

Securitatea aplicației a fost tratată într-o manieră simplificată, adoptată scopului educațional. În locul unei soluții complete precum Spring Security, s-au implementat verificări manuale în controller-e, pe baza rolului utilizatorului și a unei stări de autentificare stocate în frontend.

De exemplu, accesul la rutele de administrare a produselor este permis doar dacă utilizatorul este autentificat și are rol de admin. Într-o aplicație de producție, acest mecanism ar fi înlocuit cu autentificare bazată pe token-uri JWT, protecție CSRF și gestionarea sesiunilor la nivel de server.

3.2 Analiza și proiectarea Backend-ului

Backend-ul unei aplicații web moderne trebuie să ofere un set coerent de servicii care să răspundă cerințelor de funcționalitate, scalabilitate, securitate și mentenanță. În cadrul acestei lucrări, serverul a fost implementat folosind cadrul de dezvoltare Spring Boot, unul dintre cele mai populare cadre de lucru Java pentru dezvoltarea rapidă a aplicațiilor web și a serviciilor RESTful.

3.2.1 Proiectarea bazei de date

Pentru stocarea datelor, s-a ales utilizarea sistemului de gestiune a bazelor de date relaționale MySQL. Motivele alegerii includ:

- Stabilitate și maturitate: MySQL este un SGBD (sistem de gestiune a bazelor de date) robust, utilizat pe scară largă în proiecte comerciale și open-source.
- Compatibilitate excelentă cu Spring Data JPA: Permite o integrare facilă prin Hibernate.
- Performanță și suport pentru tranzacții ACID, esențiale într-o aplicație de eCommerce.

Diagrama Entitate-Relație a fost proiectată pentru a reflecta clar relațiile dintre entitățile principale: utilizatori, produse, categorii, sesiuni de autentificare și liste de dorințe. Fiecare entitate este reprezentată ca o tabelă, iar legăturile dintre ele sunt realizate prin chei străine.

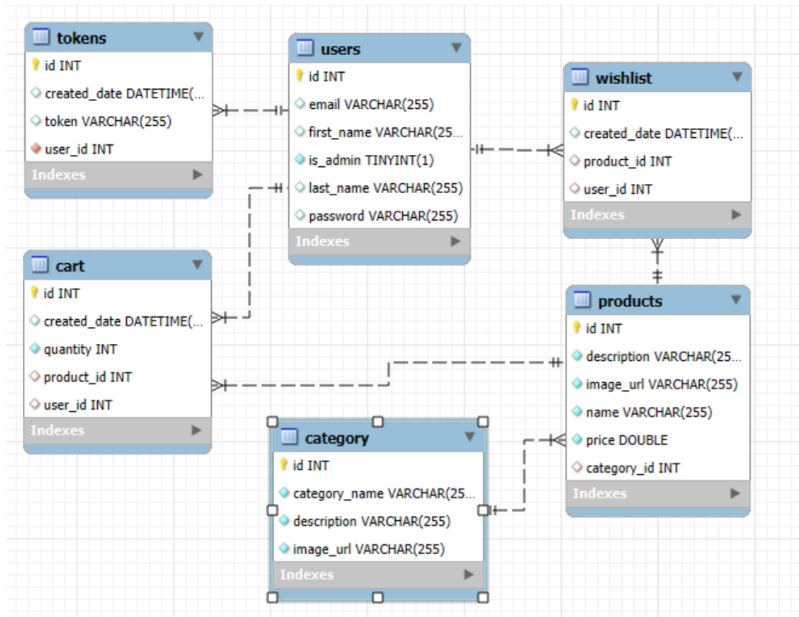


Figura 3.1. Diagrama ERD a bazei de date

Tabelul 3.1. Descrierea structurii bazei de date

Nume Tabel	Coloană	Tip de date	Constrângeri	Descriere
users	id	INT	PK, AUTO_INCREMENT	Identificator unic al utilizatorului
	first_name	VARCHAR(255)	NOT NULL	Prenume
	last_name	VARCHAR(255)	NOT NULL	Nume de familie
	email	VARCHAR(255)	NOT NULL, UNIQUE	Adresa de email
	password	VARCHAR(255)	NOT NULL	Parola
	is_admin	BOOLEAN	DEFAULT FALSE	Drepturi administrative
products	id	INT	PK, AUTO_INCREMENT	Identificator produs
	name	VARCHAR(255)	NOT NULL	Nume produs
	description	VARCHAR(255)	-	Descriere detaliată
	price	DOUBLE	NOT NULL	Preț
	image_url	VARCHAR(255)	-	Link imagine produs
	category_id	INT	FK -> category(id)	Legătura către categorie
category	id	INT	PK, AUTO_INCREMENT	Identificator categorie
	name	VARCHAR(255)	NOT NULL, UNIQUE	Nume categorie
	description	VARCHAR(255)	-	Descriere

	image_url	VARCHAR(255)	-	Link imagine categorie
tokens	id	INT	PK, AUTO_INCREMENT	ID token
	created_date	DATETIME	-	Data creării
	token	VARCHAR(255)	NOT NULL, UNIQUE	Token JWT
	user_id	INT	FK -> products(id)	Legătura către produs
cart	id	INT	PK, AUTO_INCREMENT	Identificator coș de cumpărături
	created_date	DATETIME	-	Data creării
	quantity	INT	NOT NULL	Cantitate
	product_id	INT	FK -> products(id)	Legătura către produse
	user_id	INT	FK -> users(id)	Legătura către utilizatori
wishlist	id	INT	PK, AUTO_INCREMENT	Identificator listă de dorințe
	created_date	DATETIME	-	Data creării
	product_id	INT	FK -> products(id)	Legătura către produse
	user_id	INT	FK -> users(id)	Legătura către utilizatori

3.2.2 Arhitectura stratificată Spring Boot

Aplicația backend urmează modelul stratificat classic al arhitecturii Spring Boot, care permite o separare clară a responsabilităților:

- Controller: expune API-ul public (endpoint-urile HTTP);
- Service: conține logica de business;
- Repository: interacționează direct cu baza de date.

Această separare în straturi contribuie la o organizare logică și coerentă a codului, ceea ce sporește claritatea, lizibilitatea și ușurează navigarea în cadrul proiectului.

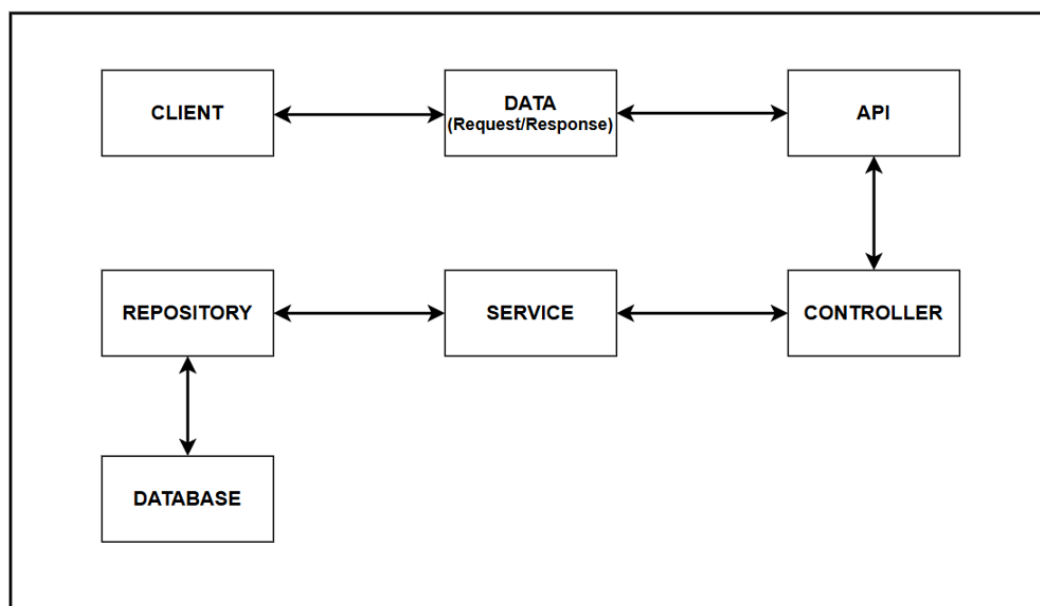


Figura 3.2. Interacțiunea componentelor Spring Boot

Fiecare cerere HTTP este procesată în ordinea: Controller -> Service -> Repository, urmând principiul responsabilității unice pentru fiecare componentă.

➤ **UserController.java**

Unul dintre cele mai importante componente ale acestui strat este controller-ul pentru gestionarea utilizatorilor. Acesta este responsabil pentru administrarea operațiunilor fundamentale legate de utilizatori: înregistrarea (sign-up) și autentificarea (sign-in). Fiind o componentă esențială pentru securitatea și funcționalitatea platformei, implementarea sa respectă principiile arhitecturii RESTful.

Clasa este adnotată cu `@RestController`, adnotare specifică framework-ului Spring, combinând `@Controller` și `@ResponseBody`. Acest lucru indică faptul că toate metodele din această clasă vor returna direct date în format JSON, fiind ideale pentru construirea unui API Rest. Adnotarea `@RequestMapping(„/users”)` stabilește calea de bază (URL-ul rădăcină) pentru toate endpoint-urile definite în acest controller, centralizând astfel rutele legate de utilizatori.

`UserController` nu conține logica de business propriu-zisă. În schimb, prin injectarea dependenței `UserService` (folosind adnotarea `@Autowired`), acesta delegă responsabilitatea procesării datelor către stratul de servicii. Această separare a responsabilităților (Separation of Concerns) este un principiu fundamental în dezvoltarea software, ducând la un cod mai curat, mai ușor de întreținut și de testat.

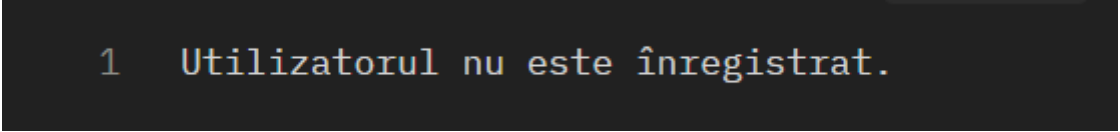
Controller-ul expune două endpoint-uri principale:

1. `POST /users/signup`: Acest endpoint gestionează crearea unui cont nou. Primește datele utilizatorului (precum nume, email, parolă) într-un obiect de tip DTO (Data Transfer Object), `SignupDto`, prin corpul cererii (`@RequestBody`). Aceste date sunt apoi transmise metodei `signup` din `UserService` pentru validare și

persistență în baza de date. Răspunsul este un `ResponseDto` generic, care informează clientul despre succesul sau eșecul operațiunii.

2. `POST /users/signin`: Acest endpoint se ocupă de autentificarea utilizatorilor existenți. Primește credențialele (email și parolă) prin intermediul unui `SignInDto`. `UserService` verifică aceste informații, iar în caz de succes, generează un token de autentificare. Controller-ul returnează acest token (încapsulat într-un `SignInResponseDto`) către client. Token-ul va fi utilizat ulterior pentru a autoriza accesul la resursele protejate ale aplicației.

Prin această implementare, `UserController` acționează ca o interfață clară și sigură între client și serverul aplicației, asigurând un flux de date coerent și respectând standardele de proiectare ale API-urilor moderne.



1 Utilizatorul nu este înregistrat.

Figura 3.3. Testarea `/signin` pentru un utilizator inexistent

➤ `CategoryController.java`

Continuând implementarea nivelului de control, `CategoryController` este componenta dedicată gestionării datelor referitoare la categoriile de produse. Rolul său este de a expune endpoint-uri prin care aplicația client poate interoga și obține informații despre categoriile definite în sistem. Spre deosebire de `UserController`, care se concentrează pe acțiuni de tip `POST`, `CategoryController` exemplifică utilizarea metodei `GET` pentru operațiuni de citire (read), conform principiilor REST.

Controller-ul expune o metodă pentru listarea tuturor categoriilor și, mai important, un endpoint specific pentru preluarea unei singure categorii pe baza ID-ului său. Implementarea acestui endpoint din urmă demonstrează o practică esențială în dezvoltarea API-urilor: gestionarea explicită a codurilor de stare HTTP pentru a comunica eficient cu clientul.

O caracteristică remarcabilă este modul în care controller-ul gestionează situația în care resursa cerută nu este găsită. Prin utilizarea unui bloc `try-catch`, acesta interceptează o excepție customizată, `ResourceNotFoundException`, aruncată de service atunci când un ID nu este valid. Această abordare permite returnarea unui răspuns HTTP semantic corect, cu statusul `404 NOT FOUND`, în loc de o eroare generică de server (`500 Internal Server Error`). Logica este încapsulată elegant în metoda `getCategoryById`, așa cum se poate observa mai jos:

```
@GetMapping("/{categoryId}")
public ResponseEntity<Category> getCategoryById(@PathVariable("categoryId")
Integer categoryId) {
    try {
        Category category = categoryService.getCategoryById(categoryId);
        return new ResponseEntity<>(category, HttpStatus.OK);
    } catch (ResourceNotFoundException e) {
```

```

        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}

```

Prin această strategie care utilizează clasa `ResponseEntity` pentru a controla atât corpul răspunsului, cât și codul de stare HTTP, `CategoryController` asigură o comunicare robustă și previzibilă cu aplicația client. Acesta este informat clar nu doar despre succesul unei cereri (cu 200 OK), ci și despre motivele specifice ale unui eșec (cu 404 NOT FOUND), aspect crucial pentru un API REST bine proiectat.

➤ **ProductController.java**

Acest controller servește drept interfața REST pentru operațiunile de vizualizare a produselor. Este mapat pe ruta de bază `/products` și se integrează strâns cu `ProductService` pentru a orchestra logica de business.

Un aspect esențial implementat în acest controller este utilizarea modelului de proiectare DTO (Data Transfer Object). În loc să expună direct entitățile `Product` din baza de date, metoda `getProducts()` returnează o listă de obiecte `ProductDto`. Această abordare reprezintă o practică recomandată în dezvoltarea de API-uri din mai multe motive:

- **Decuplare:** decuplează formatul datelor expuse clientului de structura internă a tabelelor din baza de date. Orice modificare la nivel de entitate nu va afecta în mod direct contractul API.
- **Control:** Permite o selecție fină a câmpurilor care sunt trimise către client, evitând expunerea accidentală a datelor sensibile sau irelevante.
- **Eficiență:** DTO-urile pot fi optimizate pentru a conține doar datele necesare, reducând astfel volumul de date transmis și îmbunătățind performanța.

Metoda care implementează acest concept este prezentată mai jos. Se poate observa cum controller-ul primește DTO-urile de la `ProductService` și le încapsulează într-un `ResponseEntity` cu statusul 200 OK.

```

@GetMapping("/")
public ResponseEntity<List<ProductDto>> getProducts() {
    List<ProductDto> products = productService.getAllProducts();
    return new ResponseEntity<>(products, HttpStatus.OK);
}

```

Pentru preluarea unui produs individual, controller-ul expune endpoint-ul `GET /{productId}`. Acesta menține strategia robustă de gestionare a erorilor, deja stabilită în `CategoryController`, returnând un status 404 NOT FOUND dacă produsul nu este găsit.

```

1  {
2    "timestamp": "2025-06-29T11:15:12.524+00:00",
3    "status": 404,
4    "error": "Not Found",
5    "trace": "org.springframework.web.servlet.resource.NoResourceFoundException: No static resource admin/
           products/%7B65%7D.%5Cn%5Ct%5Ct org.springframework.web.servlet.resource.ResourceHttpRequestHandler.

```

Figura 3.4. Rezultatul apelării metodei GET pe un produs inexistent

În final, `ProductController` consolidează arhitectura API-ului prin implementarea unor practici esențiale. Pe de o parte, adoptarea obiectelor `ProductDto` pentru listarea generală a produselor demonstrează o proiectare matură, decuplând reprezentarea

internă a datelor de cea expusă clientului. Pe de altă parte, controller-ul menține coerența în gestionarea erorilor, utilizând aceeași strategie robustă (`ResponseEntity` cu `HttpStatus.NOT_FOUND`) ca și celelalte componente, asigurând un comportament predictibil al API-ului.

➤ **AdminController.java**

În completarea funcționalităților de vizualizare oferite de `CategoryController`, arhitectura aplicației include o componentă dedicată exclusiv operațiunilor administrative: `AdminController`. Acesta este proiectat pentru a securiza metodele de creare, modificare și ștergere a categoriilor (operațiunile CUD din acronimul CRUD), care nu trebuie să fie accesibile utilizatorilor de tip clienți.

Principala sa particularitate constă în implementarea unui mecanism de autorizare, care reprezintă un pilon fundamental pentru securitatea platformei. Toate endpoint-urile definite în această clasă sunt prefixate cu `/admin`, semnalând clar natura lor restrictivă. Mai mult, fiecare metodă primește un parametru suplimentar, `token`, extras din cererea HTTP.

Acest `token` este trimis imediat către `PermissionService`, un serviciu specializat cu unicul scop de a verifica rolul utilizatorului. Această verificare reprezintă primul și cel mai important pas în execuția oricărei cereri către acest controller. În cazul în care verificarea eșuează, controller-ul oprește imediat procesarea și returnează un răspuns HTTP cu statusul `403 FORBIDDEN`, comunicând clientului că nu deține autorizația necesară pentru acțiunea respectivă.

```

1  {
2      "success": false,
3      "message": "Acces interzis",
4      "timestamp": "2025-06-29T12:06:23.785479300"
5  }
```

Figura 3.5. Accesul interzis al clienților neautorizați la metodele CUD

Fragmentul de cod de mai jos, extras din metoda de creare a unei noi categorii, ilustrează acest flux de securitate:

```

@PostMapping("/create")
public ResponseEntity<ApiResponse> createCategory(@RequestBody Category
category, @RequestParam("token") String token) {
    if (!permissionService.isAdmin(token)) {
        return new ResponseEntity<>(new ApiResponse(false, "Acces interzis"),
HttpStatus.FORBIDDEN);
    }
    categoryService.createCategory(category);
    return new ResponseEntity<>(new ApiResponse(true, "Categoria a fost
creată."), HttpStatus.CREATED);
}
```

Pe lângă mecanismul de autorizare, `AdminController` gestionează cu atenție validările de business. De exemplu, înainte de a actualiza o categorie, se asigură că

aceasta există deja, returnând un status 404 NOT FOUND în caz contrar. Răspunsurile sunt standardizate prin utilizarea unui obiect custom ApiResponse, care oferă clientului un mesaj clar despre rezultatul operațiunii.

```

1  {
2      "success": false,
3      "message": "Categoria nu există",
4      "timestamp": "2025-06-29T12:13:31.020301"
5  }
```

Figura 3.6. Testarea modificării unei categorii care nu există

Prin urmare, AdminCategoryController acționează ca un gardian pentru datele sensibile, separând în mod clar logica de administrare de cea de vizualizare publică și implementând un strat esențial de securitate bazat pe roluri.

➤ AdminProductController.java

În paralel cu AdminCategoryController, componenta AdminProductController este responsabilă pentru gestionarea securizată a ciclului de viață al produselor. Acest controller implementează operațiunile de creare, modificare și ștergere (Create, Update, Delete) pentru produse, funcționalități expuse prin ruta de bază /admin/products și rezervate exclusiv utilizatorilor cu rol administrativ.

Design-ul său urmează același model de securitate riguros: fiecare endpoint solicită un token de autentificare pe care îl validează imediat folosind PermissionService. Acest mecanism de protecție, care returnează un status 403 FORBIDDEN la orice tentativă de acces neautorizat, este consistent în toate componentele de administrare, formând un strat de securitate uniform și robust pentru întreaga aplicație.

O responsabilitate cheie în acest controller, pe lângă autorizare, este efectuarea unor validări esențiale înainte de a invoca logica aplicației din ProductService. De exemplu, la crearea sau actualizarea unui produs, controller-ul verifică mai întâi dacă categoria specificată în ProductDto există în baza de date. Această validare este realizată printr-o interogare directă a CategoryRepo. Plasarea acestei verificări în controller reprezintă o alegere practică în organizarea codului: se previne trimiterea unor date invalide către stratul de servicii, respingând cererile incorecte la cel mai timpuriu punct posibil și simplificând astfel logica din ProductService.

Fragmentul de cod următor ilustrează acest flux complet: autorizare, validare și, în final, execuția acțiunii:

```

@PostMapping("/add")
public ResponseEntity<ApiResponse> createProduct(@RequestBody ProductDto
productDto, @RequestParam("token") String token) {
    if (!permissionService.isAdmin(token)) {
        return new ResponseEntity<>(new ApiResponse(false, "Acces
interzis."), HttpStatus.FORBIDDEN);
    }
    Optional<Category> optionalCategory =
```

```

categoryRepo.findById(productDto.getCategoryId());
    if (!optionalCategory.isPresent()) {
        return new ResponseEntity<>(new ApiResponse(false, "category does not
exist"), HttpStatus.BAD_REQUEST);
    }
    productService.createProduct(productDto, optionalCategory.get());
    return new ResponseEntity<>(new ApiResponse(true, "product has been
added"), HttpStatus.CREATED);
}

```

Rezumând, AdminProductController nu funcționează doar ca un simplu delegat către stratul de servicii, ci joacă un rol activ în validarea datelor și în aplicarea politicilor de securitate, asigurând integritatea și robustețea operațiunilor de gestionare a produselor.

➤ **CartController.java**

Coșul de cumpărături reprezintă o componentă interactivă și esențială în experiența oricărui utilizator pe o platformă de comerț electronic. CartController este responsabil cu orchestrarea acestei funcționalități, expunând un set de endpoint-uri sub ruta /cart pentru a permite utilizatorilor să își gestioneze produsele selectate.

O caracteristică definitorie a acestui controller este că toate operațiunile sale sunt securizate și contextuale. Spre deosebire de controllerele publice care expun date generale, CartController condiționează fiecare acțiune de existența unui utilizator autentificat valid. Acest mecanism de protecție este implementat consistent la începutul fiecărei metode. Mai întâi, apelul `authenticationService.authenticate(token)`, se validează token-ul furnizat în cerere. Această singură linie de cod acționează ca un punct de control de securitate: dacă token-ul este invalid, procesarea este întreruptă imediat și se returnează un răspuns de eroare, prevenind orice acces neautorizat.

O dată ce autenticitatea token-ului este confirmată, controller-ul extrage entitatea `User` corespunzătoare prin intermediul `User user = authenticationService.getUser(token)`. Acest obiect `User` devine apoi parametrul central care este transmis mai departe către `CartService`. Astfel, se garantează că toate modificările, fie că este vorba de adăugarea, vizualizarea sau ștergerea unui produs, se aplică exclusiv coșului de cumpărături al utilizatorului care a inițiat cererea.

Controller-ul implementează operațiuni logice complete pentru gestiunea coșului:

- Adăugarea (POST /add): permite adăugarea unui produs în coș, folosind un `AddToCartDto` pentru a structura datele de intrare.
- Vizualizarea (GET /): returnează conținutul coșului, agregat într-un `CartDto` care include lista de produse și, potențial, costul total.
- Ștergerea (DELETE /delete/{cartItemId}): elimină un articol specific din coș pe baza ID-ului său.
- Golirea (POST /clear): elimină toate produsele din coșul utilizatorului curent.

În concluzie, CartController este un exemplu elocvent de implementare a unei funcționalități securizate și personalizate. Prin integrarea strânsă cu `AuthenticationService`, acesta asigură că fiecare interacțiune este atât de sigură, cât și

corect asociată contului utilizatorului, formând astfel un nivel de încredere și funcționalitate în cadrul aplicației.

➤ **WishlistController.java**

În completarea funcționalităților tranzacționale ale coșului de cumpărături, platforma oferă utilizatorilor și o listă de dorințe (wishlist). Această facilitate este gestionată de WishlistController, o componentă care, deși servește un scop diferit (salvarea produselor pentru o achiziție ulterioară, nu imediată), demonstrează coerența arhitecturală a aplicației prin reutilizarea aceluiași modele de securitate și interacțiune.

Asemenea CartController, WishlistController este o componentă securizată în întregime, unde fiecare operațiune este condiționată de autentificarea utilizatorului. Fluxul de lucru pentru validarea accesului este identic, reflectând o strategie de proiectare unitară: fiecare metodă invocă mai întâi `authenticationService.authenticate(token)` pentru a confirma validitatea sesiunii, apoi obține obiectul `User` corespunzător cu linia de cod `User user = authenticationService.getUser(token)`. Acest user este apoi pasat către `WishlistService`, asigurând că toate operațiunile se desfășoară în contextul corect al contului de utilizator.

Acest design, unde securitatea precede orice logică de business, este ilustrat elocvent în metoda `getWishlist`, care recuperează lista de dorințe pentru utilizatorul curent.

```
@GetMapping("/")
public ResponseEntity<List<ProductDto>> getWishlist(@RequestParam("token")
String token) {

    authenticationService.authenticate(token);
    User user = authenticationService.getUser(token);
    List<ProductDto> productDtos = wishlistService.getWishlistForUser(user);
    return new ResponseEntity<>(productDtos, HttpStatus.OK);
}
```

Controller-ul expune endpoint-urile standard pentru gestionarea unei colecții:

- Adăugarea (POST /add): permite utilizatorilor să adauge un produs în lista de dorințe, utilizând un `AddToWishlistDto` pentru a transporta informațiile necesare;
- Vizualizarea (GET /): spre deosebire de coșul de cumpărături, care ar putea agrega date suplimentare precum costul total, vizualizarea listei de dorințe returnează direct o listă de obiecte `ProductDto`. Această implementare reflectă natura mai simplă a wishlist-ului: o colecție de produse de interes, fără un context tranzacțional imediat.
- Ștergerea (DELETE /delete/{productId}): oferă funcționalitatea de a elimina un produs specific din listă.

Prin această implementare, WishlistController nu doar că oferă o funcționalitate valoroasă pentru retenția utilizatorilor, dar o face într-un mod care consolidează practicile de securitate și design stabilite în întreaga aplicație, demonstrând o arhitectură robustă și ușor de întreținut.

➤ UserService.java

Unul dintre cele mai importante componente ale acestui strat este UserService, care organizează toate operațiunile legate de conturile de utilizator: înregistrarea și autentificarea. Acesta depinde de UserRepository pentru a interacționa cu baza de date și de AuthenticationService pentru a gestiona token-urile de sesiune, demonstrând o bună separare a responsabilităților.

Metoda signUp gestionează procesul de creare a unui cont nou. Fiind o operațiune critică ce implică scrieri multiple în baza de date, este marcată cu adnotarea @Transactional. Aceasta asigură atomicitatea operațiunii: fie toți pașii reușesc, fie întreaga tranzacție este anulată (rollback) în caz de eroare, prevenind astfel stări de date inconsistente. Fluxul logic începe cu o validare esențială de business. Cu ajutorul următoarei secvențe de cod, se face verificarea adresei de e-mail de către server, în caz că aceasta a fost sau nu utilizată deja, aruncând o excepție CustomException dacă acest lucru este adevărat, pentru a garanta unicitatea conturilor:

```
if (userRepo.existsByEmail(signupDto.getEmail())) {  
    throw new CustomException("User already exists.");  
}
```

După crearea și salvarea utilizatorului, o altă operațiune crucială este generarea token-ului inițial, urmată de persistența acestuia, delegată către AuthenticationService. Această succesiune de acțiuni formează nucleul procesului de înregistrare.

```
final AuthenticationToken authenticationToken = new  
AuthenticationToken(user);  
authenticationService.saveConfirmationToken(authenticationToken);  
  
return new ResponseDto("success", "Contul a fost creat cu succes.");
```

Metoda signIn se ocupă de validarea credențialelor. Procesul începe prin a căuta utilizatorul în baza de date după adresa de e-mail. Dacă utilizatorul nu este găsit, se aruncă o excepție specifică, AuthenticationFailException. Un aspect important de analizat este mecanismul de validare a parolei. În implementarea curentă, verificarea se face prin comparația directă a parolelor. Într-o aplicație de producție, această abordare este inacceptabilă din punct de vedere al securității. O implementare robustă ar presupune ca la înregistrare parola să fie criptată cu un algoritm puternic (de exemplu BCrypt), iar la autentificare, parola furnizată de utilizator să fie criptată cu același algoritm și comparată cu valoarea stocată în baza de date. Această simplificare din proiectul curent servește unui scop didactic, însă evidențiază un punct critic în securitatea oricărei aplicații web.

```
if (Objects.isNull(user)) {  
    throw new AuthenticationFailException("Utilizatorul nu este  
înregistrat.");  
}  
  
if (!user.getPassword().equals(signInDto.getPassword())) {  
    throw new AuthenticationFailException("Parolă incorectă.");  
}
```

Dacă autentificarea reușește, serviciul obține (sau creează, dacă nu există) un token de autentificare pentru utilizator și îl returnează într-un `SignInResponseDto`, alături de statusul de administrator al utilizatorului.

➤ **AuthenticationService.java**

Dacă `UserService` se ocupă de gestionarea entităților de utilizatori (cine sunt ei), `AuthenticationService` este componenta dedicată gestionării și validării sesiunilor lor (dacă sunt cine pretind a fi). Acest serviciu este un pilon al securității aplicației, acționând ca o sursă unică de adevăr (single source of truth) pentru starea de autentificare a unui utilizator. Rolul său principal este de a decupla logica de manipulare a token-urilor de restul logicii de business. Principalele responsabilități ale acestui serviciu sunt:

- Persistența token-urilor: salvarea unui nou token în baza de date la înregistrarea sau autentificarea unui utilizator, prin metoda `saveConfirmationToken`.
- Recuperarea datelor: furnizarea de metode pentru a obține utilizatorul asociat unui token (`getToken(user)`) sau, mai important, pentru a obține utilizatorul asociat unui token (`getUser(String token)`).
- Validarea sesiunii: verificarea validității unui token primit într-o cerere.

Cea mai importantă funcționalitate este încapsulată în metoda `authenticate`, care este invocată de toate controllerele ce gestionează resurse protejate. Această metodă acționează ca un gardian de securitate (security gatekeeper), având o singură responsabilitate: să determine dacă un token este valid sau nu, aruncând o excepție în caz de eșec.

```
public void authenticate(String token) throws AuthenticationFailException {
    if (Objects.isNull(token)) {
        throw new AuthenticationFailException("token-ul este null");
    }
    if (Objects.isNull(getUser(token))) {
        throw new AuthenticationFailException("token invalid sau expirat");
    }
}
```

Acest scurt bloc de cod realizează o verificare robustă pentru a se asigura că token-ul nu este nul, apoi, prin apelul `getUser(token)`, încearcă să găsească utilizatorul corespunzător. Dacă oricare dintre aceste condiții eșuează, metoda aruncă o excepție customizată, `AuthenticationFailException`. Această abordare (fail-fast) este preferabilă returnării unui boolean, deoarece centralizează gestionarea erorilor de autentificare și permite straturilor superioare (controllere) să reacționeze uniform la eșecuri, de exemplu, prin returnarea unui status HTTP 401 Unauthorized sau 403 Forbidden.

În concluzie, `AuthenticationService` este un exemplu de componentă cu o singură responsabilitate bine definită (Single Responsibility Principle), esențială pentru decuplarea și centralizarea logicii de securitate în cadrul aplicației.

➤ **PermissionService.java**

O componentă importantă în orice aplicație care presupune roluri diferite ale utilizatorilor este verificarea drepturilor de acces. În cadrul acestei aplicații, rolurile sunt

împărțite în utilizatori obișnuiți (clienți) și administratori, iar verificarea permisiunilor se face printr-un serviciu dedicat, `PermissionService`.

Această clasă oferă un punct unic de verificare a permisiunilor administrative ale utilizatorului, pe baza unui token de autentificare. Ea este utilizată în special în controller-urile care expun funcționalități rezervate exclusiv administratorilor, cum ar fi adăugarea sau ștergerea de produse.

Clasa este marcată cu adnotarea `@Service(„permissionService”)`, ceea ce o face disponibilă ca un bean Spring și permite injectarea sa în alte componente. Metoda principală este `isAdmin(String token)`, care returnează `true` dacă token-ul corespunde unui utilizator cu drepturi administrative:

```
public boolean isAdmin(String token) {  
    if (token == null) return false;  
    AuthenticationToken authToken = authenticationService.getToken(token);  
    if (authToken == null || authToken.getUser() == null) {  
        return false;  
    }  
    User user = authToken.getUser();  
    return user.isAdmin();  
}
```

Metoda verifică dacă token-ul este null, caută token-ul înregistrat în baza de date prin `AuthenticationService`, verifică dacă utilizatorul asociat există, iar la final returnează valoarea atributului `isAdmin` al utilizatorului (boolean).

Prin centralizarea verificărilor de permisiuni în clasa `PermissionService`, se asigură o separare clară a responsabilităților și se evită duplicarea codului de verificare în mai multe locuri ale aplicației. Acest model simplu dar eficient contribuie la o arhitectură curată și ușor de întreținut.

➤ **CategoryService.java**

Continuând implementarea stratului de servicii, `CategoryService` este componenta responsabilă de gestionarea întregii logici de business asociate cu categoriile de produse. Acesta acționează ca un intermediar esențial între stratul de control (`CategoryController` și `AdminCategoryController`) și stratul de persistență (`CategoryRepo`), încapsulând toate regulile și operațiunile necesare pentru manipularea datelor despre categorii.

Serviciul oferă implementări pentru operațiunile standard CRUD, fiecare metodă având o responsabilitate clar definită:

- Crearea și Listarea: metodele `createCategory` și `listCategory` sunt implementări directe care delegă sarcina direct către `CategoryRepo`, folosind metodele `save` și `findAll`. Ele nu conțin logică complexă, ci doar facilitează accesul la operațiunile de bază ale depozitului de date.
- Actualizarea (`editCategory`): această metodă exemplifică modelul standard de actualizare în JPA/Hibernate. Mai întâi, entitatea `Category` existentă este preluată din baza de date folosind `categoryRepo.getById(categoryId)`. Apoi, câmpurile acestei entități gestionate sunt modificate cu noile valori primite

în `updateCategory`. În final, `categoryRepo.save(category)` este apelat pentru a salva modificările.

- Verificarea și Ștergerea: În cazul ștergerii, metoda `deleteCategory` aplică un principiu de programare defensivă. Înainte de a încerca să șteargă o categorie, verifică existența acesteia prin `!categoryRepo.existsById(categoryId)`. Dacă resursa nu este găsită, aruncă o excepție customizată, `categoryNotExistsException`, oferind astfel un feedback clar și specific despre natura erorii.

În metoda `getCategoryById` se află o demonstrație elegantă a practicilor moderne de codificare din acest serviciu. Aceasta gestionează cazul în care o categorie nu este găsită într-un mod concis și expresiv, folosind API-ul `Optional` din Java.

```
public Category getCategoryById(Integer categoryId) {
    return categoryRepo.findById(categoryId)
        .orElseThrow(() -> new ResourceNotFoundException("Category not
found with id: " + categoryId));
}
```

Această abordare, care utilizează o expresie lambda în `orElseThrow`, este superioară unui bloc `if-else` tradițional din mai multe motive. Este mai lizibilă, reduce codul redundant și centralizează logica de tratare a cazului „nu a fost găsit” într-o singură linie. Prin aruncarea unei excepții specifice, `ResourceNotFoundException`, serviciul permite straturilor superioare să intercepteze această eroare și să o traducă într-un răspuns HTTP corespunzător, cum ar fi `404 NOT FOUND`.

Prin urmare, `CategoryService` reprezintă un exemplu clasic de componentă de serviciu bine proiectată. Acesta încapsulează logica de business specifică domeniului său, comunică exclusiv cu stratul de persistență prin `CategoryRepository` și implementează o strategie robustă de gestionare a erorilor, contribuind la modularitatea și mentenabilitatea generală a aplicației.

➤ **ProductService.java**

`ProductService` este componenta centrală care implementează logica aplicației pentru toate operațiunile legate de produse. Acționând ca un orchestrator, acest serviciu, pe lângă că intermediază comunicarea cu `ProductRepository`, este responsabil și pentru o sarcină esențială în arhitectura API: conversia între entitățile de domeniu (`Product`) și obiectele de transfer de date (`ProductDto`).

O caracteristică fundamentală a acestui serviciu este utilizarea excesivă a modelului `Data Transfer Object (DTO)`. Niciodată nu se expune entitatea `Product` direct către straturile superioare. În schimb, `ProductService` oferă metode dedicate pentru a transforma datele.

- `getProductDto(Product product)`: o metodă ajutătoare a cărei unică responsabilitate este de a mapa câmpurile unei entități `Product` într-un `ProductDto`.
- `getAllProducts()`: această metodă preia lista completă de produse din baza de date, iterează prin ea și folosește metoda de mapare de mai sus pentru a

construi o listă de DTO-uri. Astfel, contractul API este decuplat de modelul de date intern, permițând o mai mare flexibilitate și securitate.

În ceea ce privește operațiunile de modificare (create, update), serviciul demonstrează o separare clară a responsabilităților. De exemplu, metoda `createProduct(ProductDto productDto, Category category)` primește ca parametru nu doar un DTO, ci și entitatea `Category` deja validată și preluată din baza de date. Acest lucru indică faptul că responsabilitatea de a verifica dacă o categorie există a fost delegată unui strat superior (în acest caz, `AdminProductController`), simplificând astfel logica din `ProductService` și permițându-i să se concentreze pe crearea efectivă a produsului.

Metoda de actualizare, `updateProduct`, este un exemplu elocvent al robusteții implementării, combinând mai multe practici recomandate.

```
@Transactional
public void updateProduct(Integer productId, ProductDto productDto, Category
category) throws ProductNotFoundException {
    Product product = productRepo.findById(productId)
        .orElseThrow(() -> new ProductNotFoundException("Produsul cu ID-
ul " + productId + " nu a fost gasit"));
    product.setDescription(productDto.getDescription());
    product.setImageUrl(productDto.getImageUrl());
    product.setName(productDto.getName());
    product.setPrice(productDto.getPrice());
    product.setCategory(category);

    productRepo.save(product);
}
```

Această metodă este marcată cu `@Transactional`, asigurând că actualizarea în baza de date este o operațiune atomică. Mai întâi, se preia produsul existent folosind un model „find-or-throw” pentru a se asigura că modificarea se aplică unei entități valide. Apoi, datele din DTO sunt transferate pe entitatea `Product`, care este ulterior salvată.

`ProductService` este mai mult decât un simplu intermediar. Acesta gestionează activ ciclul de viață al datelor despre produse, aplică modelul DTO pentru a proteja și a structura API-ul, și implementează operațiuni tranzacționale sigure, formând astfel un pilon central în arhitectura de business a aplicației.

➤ **CartService.java**

`CartService` este o componentă esențială care gestionează una dintre cele mai interactive și dinamice părți ale aplicației: coșul de cumpărături. Spre deosebire de serviciile care se ocupă de date mai statice (precum categoriile), `CartService` încapsulează o logică de business complexă, contextuală și specifică fiecărui utilizator. Acesta nu se limitează la simple operațiuni CRUD; el orchestrează reguli, calcule și validări de securitate.

Principalele responsabilități ale `CartService` sunt:

- Logica de adăugare și actualizare inteligentă: metoda `addToCart` nu este o simplă inserare în baza de date. Ea implementează o regulă de business

fundamentală pentru experiența utilizatorului: dacă un produs este adăugat în coș, iar acesta există deja, serviciul nu creează o nouă intrare, ci actualizează cantitatea celei existente. Acest lucru este realizat prin interogarea `cartRepo.findByUserAndProduct`, o metodă customizată a depozitului de date, care permite gestionarea eficientă a stării coșului.

```
if (existingCartItem != null) {
    existingCartItem.setQuantity(existingCartItem.getQuantity() +
addToCartDto.getQuantity());
    cartRepo.save(existingCartItem);
} else {
    Cart newCartItem = new Cart();
    newCartItem.setProduct(product);
    newCartItem.setUser(user);
    newCartItem.setQuantity(addToCartDto.getQuantity());
    newCartItem.setCreatedDate(new Date());
    cartRepo.save(newCartItem);
}
```

- Agregarea și transformarea datelor: când un utilizator solicită vizualizarea coșului său, `CartService` face mai mult decât să returneze o listă de articole. Metoda `listCartItems` demonstrează rolul serviciului în agregarea datelor: aceasta iterează prin articolele din coș, le transformă în obiecte `CartItemDto` și, simultan, calculează costul total (`totalCost`). Rezultatul este un obiect `CartDto` complex, care încapsulează atât lista de produse, cât și valoarea agregată, fiind gata de a fi consumat direct de către interfața client.
- Validări de securitate la nivel de resursă: un aspect critic implementat este autorizarea la nivel de resursă. La ștergerea unui articol din coș (`deleteCartItem`), serviciul nu doar că validează existența articolului (`optionalCart.isEmpty()`), ci execută o verificare de proprietate crucială: `if (cart.getUser() != user)`. Această linie de cod asigură că un utilizator nu poate șterge, accidental sau intenționat, un articol din coșul altui utilizator, chiar dacă ar recunoaște ID-ul acestuia. Această validare de ownership este un pilon al securității în aplicațiile multi-utilizator.

```
if (cart.getUser() != user) {
    throw new CustomException("cart item does not belong to user: " + cartItemId);
}
cartRepo.delete(cart);
```

➤ **WishlistService.java**

În mod similar cu `CartService`, componenta `WishlistService` gestionează o colecție de produse specifică fiecărui utilizator, însă implementează reguli de business diferite, adaptate naturii unei liste de dorințe.

Principala diferență față de coșul de cumpărături se observă în metoda `addToWishlist`. Spre deosebire de coș, care actualizează cantitatea, o listă de dorințe conține doar intrări unice. Serviciul impune această regulă printr-o verificare preventivă:

```
if (!wishlistRepo.existsByUserAndProduct(user, product)) {
    Wishlist wishlistItem = new Wishlist(user, product);
    wishlistRepo.save(wishlistItem);
}
```

care oprește adăugarea unui produs dacă acesta există deja în listă.

Această abordare mai simplă se reflectă și în metoda de vizualizare, `getWishlistForUser`. Aceasta se concentrează exclusiv pe transformarea produselor asociate în `ProductDto`-uri, fără a necesita calcule precum costul total, specific coșului de cumpărături. La fel ca și `CartService`, acesta demonstrează o bună practică de compoziție între servicii, delegând sarcina de conversie către `ProductService`, evitând astfel duplicarea logicii de mapare.

În esență, `WishlistService` adaptează modelul de gestiune al unei colecții de produse, eliminând complexitatea tranzacțională a coșului și concentrându-se pe o experiență de utilizator simplă și directă.

➤ **UserRepository.java**

Aceasta este interfața responsabilă pentru gestionarea persistenței entităților de tip `User`. Prin simpla extindere a `JpaRepository<User, Integer>`, `UserRepository` obține automat acces la metode esențiale precum `save()`, `findById()`, `findAll()` și `deleteById()`.

Pe lângă aceste operațiuni standard, `UserRepository` definește o metodă de interogare derivată (derived query method):

```
User findByEmail(String email);
```

Aceasta este o caracteristică puternică a Spring Data JPA. Fără a necesita implementare, cadrul de dezvoltare interpretează numele metodei și generează automat interogarea SQL corespunzătoare (`SELECT * FROM users WHERE email = ?`). Acest mecanism asigură un cod curat, extrem de lizibil și fără a necesita scrierea manuală a interogărilor SQL pentru cazuri comune.

➤ **TokenRepository.java**

`TokenRepository` gestionează persistența entităților `AuthenticationToken`, care sunt esențiale pentru securitatea aplicației. La fel ca și `UserRepository`, acesta extinde `JpaRepository` pentru a beneficia de operațiunile CRUD standard.

Interfața definește două metode de interogare derivate, fiecare servind un scop distinct în logica de autentificare:

```
AuthenticationToken findByUser(User user);

AuthenticationToken findByToken(String token);
```

- `findByUser` este utilizată pentru a recupera token-ul unui utilizator existent în timpul procesului de autentificare.
- `findByToken` este crucială pentru a valida token-urile primite în cererile către resurse protejate, permițând identificarea utilizatorului asociat.

Aceste două metode transformă TokenRepository în instrumentul de bază pentru AuthenticationService, facilitând o gestionare eficientă și rapidă a sesiunilor de utilizator.

➤ **CategoryRepository.java**

Aceasta este interfața pentru entitățile Category. Faptul că nu definește nicio metodă suplimentară demonstrează eficiența abstracției JpaRepository. Ale cărui funcționalități standard sunt complet suficiente pentru a satisface toate cerințele de business legate de gestionarea categoriilor.

➤ **ProductRepository.java**

La fel ca și CategoryRepository, interfața pentru entitățile Product nu definește nicio metodă customizată. Și în acest caz, funcționalitățile standard moștenite de la JpaRepository sunt suficiente pentru a acoperi cerințele logice ale ProductService, de la listarea produselor până la căutarea după ID.

➤ **CartRepository.java**

Acesta demonstrează cum stratul de persistență poate fi extins pentru a oferi suport direct pentru reguli de business mai complexe. Pe lângă operațiunile CRUD moștenite, acesta definește trei metode de interogare derivate, fiecare având un rol specific în logica CartService.

```
List<Cart> findAllByUserOrderByCreateDateDesc(User user);

Cart findByUserAndProduct(User user, Product product);

@Transactional
void deleteByUser(User user);
```

- `findAllByUserOrderByCreateDateDesc`: această metodă este utilizată pentru a afișa conținutul coșului unui utilizator, sortând articolele în ordine descrescătoare a datei de creare, o funcționalitate intuitivă pentru utilizatorul final.
- `findByUserAndProduct`: este o interogare esențială pentru a verifica dacă un produs specific există deja în coșul unui utilizator, permițând astfel actualizarea cantității în loc de crearea unei intrări noi.
- `deleteByUser`: marcată cu `@Transactional` pentru a garanta integritatea datelor, această metodă oferă o modalitate eficientă de a goli complet coșul unui utilizator, de exemplu, după finalizarea unei comenzi.

Aceste metode personalizate abstractizează complet complexitatea interogărilor, oferind CartService-ului o interfață clară și expresivă pentru a-și implementa logica.

➤ **WishlistRepository.java**

Similar cu arhitectura CartRepo, depozitul de date WishlistRepository oferă un set de metode de interogare derivate, specializate pentru a susține logica unică a unei liste de dorințe. Acesta extinde JpaRepository pentru a-i moșteni funcționalitățile de bază CRUD.

```
List<Wishlist> findAllByUserOrderByCreateDateDesc(User user);
boolean existsByUserAndProduct(User user, Product product);
```

```
@Transactional
void deleteByUserAndProduct(User user, Product product);
```

- `findAllByUserOrderByCreateDateDesc`: similar cu implementarea din `CartRepository`, stabilește o convenție de design consistentă pentru preluarea colecțiilor de produse sortate.
- `existsByUserAndProduct`: această metodă este esențială pentru a preveni adăugarea duplicatelor în lista de dorințe. Aceasta returnează un simplu boolean, fiind o interogare mai eficientă pentru acest caz de utilizare.
- `deleteByUserAndProduct`: o metodă precisă, tranzacțională, care permite eliminarea unui articol specific din lista de dorințe pe baza utilizatorului și a produsului, fără a necesita ID-ul entității `Wishlist` în sine.

Prin aceste metode, `WishlistRepository` oferă `WishlistService`-ului unelte de acces la date optimizate și semantice, care reflectă direct regulile de business ale funcționalității.

3.2.3 Proiectarea API-ului RESTful

API-ul este organizat în mod RESTful, respectând convențiile de metodă HTTP și structura semantică a URL-urilor. Acest lucru permite o interacțiune clară, previzibilă și ușor de integrat cu frontend-ul sau alte sisteme externe.

Tabelul 3.2. Endpoint-uri `UserController`

Metodă HTTP	Cale	Descriere	Request Body	Răspuns
POST	<code>/users/signup</code>	Înregistrează un utilizator nou în sistem	<code>SignupDto</code> {firstName, lastName, email, password}	<code>ResponseDto</code> {status, message}
POST	<code>/users/signin</code>	Autentică un utilizator existent și returnează un token de sesiune și rolul său	<code>SignInDto</code> {email, password}	<code>SignInResponseDto</code> {status, token, isAdmin}

Tabelul 3.3. Endpoint-uri `CategoryController` și `ProductController`

Metodă HTTP	Cale	Descriere	Request Body	Răspuns
GET	<code>/categories/</code>	Returnează lista tuturor categoriilor de produse	N/A	<code>List<Category></code>
GET	<code>/categories/{categoryId}</code>	Returnează o singură	N/A	<code>Category</code>

		categoriei pe baza ID-ului		
GET	/products/	Returnează lista tuturor produselor sub formă de DTO	N/A	List<ProductDTO>
GET	/products/{productId}	Returnează un singur produs pe baza ID-ului	N/A	Product

Tabelul 3.4. Endpoint-uri AdminCategoryController și AdminProductController

HTTP	Cale	Descriere	Request Body	Răspuns
POST	/admin/categories/create	Creează o nouă categorie	Category{ categoryName , description, imageUrl}	ApiResponse { success, message}
POST	/admin/categories/update/{categoryId}	Actualizează o categorie existentă	Category{ categoryName , description, imageUrl}	ApiResponse { success, message}
DELETE	/admin/categories/delete/{categoryId}	Șterge o categorie existentă	N/A	ApiResponse {success, message}
POST	/admin/products/add	Adaugă un produs nou	ProductDto{ name, imageUrl, price, description, categoryId}	ApiResponse { success, message}
POST	/admin/products/update/{productId}	Actualizează un produs existent		ApiResponse { success, message}
DELETE	/admin/products/delete/{productId}	Șterge un produs existent	N/A	ApiResponse { success, message}

Tabelul 3.5. Endpoint-uri CartController

HTTP	Cale	Descriere	Request Body	Răspuns
POST	/cart/add	Adaugă un produs în coș	AddToCartDto {productId, quantity}	ApiResponse{ success, message}
GET	/cart/	Returnează conținutul coșului	N/A	CartDto{ cartItems: [CartItemDto], totalCost}
DELETE	/cart/delete/{cartItemId}	Șterge un articol specific din coș	N/A	ApiResponse{ success, message}
POST	/cart/clear	Golește complet coșul	N/A	ApiResponse{ success, message}

Tabelul 3.6. Endpoint-uri WishlistController

HTTP	Cale	Descriere	Request Body	Răspuns
POST	/wishlist/add	Adaugă un produs în lista de dorințe	AddToWishlistDto {productId, quantity}	ApiResponse{ success, message}
GET	/wishlist/	Returnează conținutul listei de dorințe	N/A	List<ProductDto>
DELETE	/wishlist/delete/{cartItemId}	Șterge un articol specific din lista de dorințe	N/A	ApiResponse{ success, message}

Acest endpoint este consumat în mod asincron de frontend prin cereri axios, permițând un flux SPA fluid și rapid.

3.2.4 Gestionarea excepțiilor

Gestionarea excepțiilor este o componentă esențială în arhitectura unei aplicații backend robuste. Într-o aplicație RESTful modernă, este important ca erorile să fie tratate centralizat și să fie comunicate clientului într-un mod standardizat, lizibil și predictibil. O

abordare coerentă a erorilor nu doar îmbunătățește experiența dezvoltatorilor care consumă API-ul, ci și facilitează depanarea și mentenanța aplicației.

În cadrul aplicației, a fost adoptat un model de gestionare personalizată a excepțiilor, care implică definirea unor clase de excepții proprii și utilizarea adnotărilor oferite de Spring pentru a intercepta și transforma aceste excepții în răspunsuri HTTP adecvate.

Un exemplu de excepție personalizată este următoarea:

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) {
        super(message);
    }
}
```

Structura clasei `ResourceNotFound` este simplă: ea conține un constructor care acceptă un mesaj ce va fi transmis mai departe pentru a fi inclus în răspunsul HTTP. La nivel de semnificație, această excepție servește drept un semnal clar că o anumită resursă identificabilă printr-un ID sau alt criteriu nu a fost găsită, motiv pentru care serverul va răspunde cu un cod HTTP 404 – Not Found.

Pentru a trata excepțiile într-un mod unitar, aplicația include o clasă de tip `@ControllerAdvice`. Aceasta acționează ca un interceptor global pentru excepții aruncate în timpul executării metodelor din controllere.

Un exemplu de răspuns JSON în caz de eroare este următorul:

```
{
  "status": 404,
  "timestamp": "2025-06-28T13:45:12.321Z",
  "message": "Produsul cu ID-ul specificat nu a fost găsit."
}
```

Această abordare contribuie la separarea preocupărilor, întrucât logica de tratare a erorilor nu mai poluează codul controllerelor și oferă un răspuns clar clientului în cazul unor probleme.

3.3 Implementarea componentei Frontend

Componenta frontend a platformei Craftique a fost dezvoltată ca o Single-Page Application (SPA) utilizând cadrul de dezvoltare Vue.js, versiunea 3. Această secțiune va detalia deciziile arhitecturale, structura proiectului și implementarea tehnică a componentelor cheie, oferind o perspectivă completă asupra modului în care a fost construită interfața client.

3.3.1 Structura proiectului și organizarea fișierelor

Pentru a asigura o mentenanță facilă și o scalabilitate ridicată, proiectul a fost organizat într-o structură de directoare modulară și intuitivă. Fiecare folder are o responsabilitate clar definită, conform principiului separării responsabilităților (Separation of Concerns).

└─ dist/	# Directorul pentru build-ul de producție (generat automat)
└─ node_modules/	# Dependențele proiectului (generate de npm/yarn)
└─ public/	# Resurse statice, copiate direct în build
└─ images/	# Imagini publice
└─ logo_fb.png	# Iconița aplicației
└─ index.html	# Punctul de intrare HTML al aplicației
└─ src/	# Directorul principal cu codul sursă
└─ assets/	# Resurse procesate de sistemul de build (CSS, imagini)
└─ components/	# Componente UI reutilizabile
└─ CategoryBox.vue	# Componentă pentru afișarea unei categorii
└─ FooterPart.vue	# Componenta pentru subsolul paginii
└─ NavBarPart.vue	# Bara de navigație principală
└─ ProductBox.vue	# Componentă pentru afișarea unui produs
└─ plugins/	# Plugin-uri și configurări pentru biblioteci externe
└─ axios.js	# Configurare centralizată pentru clientul HTTP Axios
└─ router/	# Configurația pentru rutare
└─ index.js	# Definirea tuturor rutelor aplicației (ex: /home, /cart)
└─ stores/	# Managementul stării globale (Pinia/Vuex)
└─ auth.js	# Stocare token, date utilizator, status autentificare
└─ config.js	# Stocare variabile de configurare (ex: URL-ul API-ului)
└─ views/	# Componente de tip pagină (mapate de router)
└─ Admin/	
└─ AdminDashboard.vue	
└─ AdminPage.vue	
└─ Category/	
└─ AddCategory.vue	
└─ AdminCategories.vue	
└─ AllCategories.vue	
└─ EditCategory.vue	

```

|   |   └─ ListProducts.vue
|   └─ Product/
|   |   └─ AddProduct.vue
|   |   └─ AdminProducts.vue
|   |   └─ AllProducts.vue
|   |   └─ EditProduct.vue
|   |   └─ ShowDetails.vue
|   └─ AboutView.vue
|   └─ CartView.vue
|   └─ CheckOut.vue
|   └─ HomeView.vue
|   └─ NotFound.vue
|   └─ SigninView.vue
|   └─ SignupView.vue
|   └─ WishListView.vue
└─ App.vue                # Componenta rădăcină a aplicației
└─ main.js                # Punctul de intrare al aplicației (inițializare Vue)
└─ .gitignore             # Fișiere și directoare ignorate de Git
└─ babel.config.js        # Configurare pentru Babel (transpiler JavaScript)
└─ package.json           # Lista de dependențe și script-uri ale proiectului
└─ vue.config.js          # Configurare specifică pentru Vue CLI

```

- **public/** : Acest director conține resursele statice care sunt copiate direct la rădăcina aplicației finale, fără a fi procesate de sistemul de build. Aici se află fișierul `index.html` (punctul de intrare al SPA-ului), `logo_fb.png` și un subdirector `images/` care conține imaginile de produse și categorii.
- **src/** : Acesta este directorul principal al codului sursă.
 - **assets/** : Conține resurse statice (imagini, fonturi) care sunt parte integrantă a componentelor și care vor fi procesate, optimizate și incluse în pachetul final de către sistemul de build. Logo-ul principal și alte elemente de design sunt stocate aici.
 - **components/** : Conține componente Vue reutilizabile, de prezentare (cunoscute ca „dumb components”). Rolul lor este să afișeze date primite prin props, fără a conține logică de business proprie. Exemple: `ProductBox.vue`, `CategoryBox.vue`, `NavbarPart.vue`.
 - **plugins/** : Un director dedicat configurării plugin-urilor externe. Conține fișierul `axios.js`, unde a fost configurată o instanță `apiClient` centralizată cu interceptori pentru a gestiona automat `baseURL`-ul și adăugarea token-ului la cereri.

- router/ : Definește logica de navigare. Fișierul index.js mapează căile URL la componentele corespunzătoare și conține gardianul de navigație global (router.beforeEach), pentru securitatea rutelor.
- stores/ : Reprezintă nucleul managementului stării cu Pinia. Conține store-uri modulare: auth.js pentru starea de autentificare și config.js pentru constantele aplicației.
- views/ : Conține componentele de tip „pagină” (sau „smart components”). Fiecare fișier de aici corespunde unei rute și este responsabil pentru propria sa logică, inclusiv încărcarea datelor de la API. Acestea sunt organizate în subdirectoare logice: Admin/, Category/, Product/, etc.

Această structură clară permite oricărui dezvoltator să înțeleagă rapid organizarea proiectului și să localizeze cu ușurință codul relevant pentru o anumită funcționalitate.

3.3.2 Managementul centralizat al stării Pinia

Una dintre deciziile arhitecturale fundamentale a fost utilizarea bibliotecii Pinia pentru managementul stării globale. Într-o aplicație de eCommerce, date precum statusul de autentificare al utilizatorului, rolul acestuia (administrator sau client) și numărul de articole din coșul de cumpărături trebuie să fie accesibile și sincronizate între multiple componente neînrudite.

A fost definit un store principal, auth.js pentru a centraliza aceste informații. Acest store acționează ca o „sursă unică de adevăr” pentru toate datele și operațiunile legate de starea utilizatorului (dacă este logat, ce rol are, etc.), asigurând consistența datelor în întreaga aplicație.

Store-ul este organizat în trei secțiuni logice, conform convențiilor Pinia:

a) Starea (State)

Secțiunea state este o funcție care returnează starea inițială a store-ului. Este nucleul reactiv al sistemului. La inițializarea aplicației, starea este populată cu datele din localStorage-ul browser-ului, asigurând persistența sesiunii utilizatorului.

```
state: () => ({
  token: localStorage.getItem('token') || null,
  isAdmin: localStorage.getItem('isAdmin') === 'true',
  cartCount: 0,
}),
```

- token: stochează token-ul de autentificare JWT al utilizatorului. La prima încărcare, încearcă să citească token-ul din localStorage. Dacă nu găsește nimic, valoarea sa implicită este nul, indicând un utilizator nelogat.
- isAdmin: un flag boolean care indică dacă utilizatorul logat are permisiuni de administrator. Se utilizează o comparație strictă (=== 'true') la citirea din localStorage, deoarece acesta stochează toate valorile ca string-uri. Dacă cheia nu există, valoarea va fi false. Această proprietate este esențială pentru afișarea/ascunderea condiționată a elementelor UI de administrare.

- cartCount: un număr care stochează cantitatea de articole din coșul de cumpărături, afișată în Navbar.

b) Getters

Secțiunea getters conține proprietăți „computed” derivate din state. Acestea oferă o modalitate curată și reutilizabilă de a accesa starea pre-procesată, fără a duplica logica în componente.

```
getters: {
  isLoggedIn: (state) => !!state.token,
},
```

- isLoggedIn: acest getter simplu returnează true dacă state.token conține un string (nu este null) și false în caz contrar. Prin folosirea dublei negații (!!), orice valoare "truthy" (un string) este convertită în true, iar orice valoare "falsy" (null, undefined, string gol) este convertită în false. Componentele pot acum să verifice authStore.isLoggedIn în loc să facă propria lor verificare a existenței token-ului.

c) Acțiunile (Actions)

Acțiunile sunt metodele care au permisiunea de a modifica starea (state). Toate modificările stării de autentificare se fac exclusiv prin aceste acțiuni, asigurând un flux de date controlat și predictibil.

```
actions: {
  setToken(newToken, userIsAdmin) {
    const isAdminValue = !!userIsAdmin;
    this.token = newToken;
    this.isAdmin = isAdminValue;
    localStorage.setItem('token', newToken);
    localStorage.setItem('isAdmin', isAdminValue);
  },
  setCartCount(newCount) {
    this.cartCount = newCount;
  },
  clearAuth() {
    this.token = null;
    this.isAdmin = false;
    this.cartCount = 0;
    localStorage.removeItem('token');
    localStorage.removeItem('isAdmin');
  },
  async initializeAuth() {
    this.token = localStorage.getItem('token');
    if (this.token) {
```

```

        const configStore = useConfigStore();
        try {
            const response = await
axios.get(`${configStore.baseURL}cart/`);
            this.cartCount = response.data.cartItems.length;
        } catch (error) {
            console.error("Failed to fetch cart on init:", error);
            if (error.response && [401,
403].includes(error.response.status)) {
                this.clearAuth();
            }
        }
    }
}
}

```

- `setToken(newToken, userIsAdmin)`: Această acțiune este apelată de componenta `Signin.vue` după un login reușit.
 - Primește ca parametri token-ul și flag-ul de administrator de la backend.
 - Actualizează proprietățile reactive `this.token` și `this.isAdmin`.
 - Salvează aceste valori și în `localStorage` pentru a asigura persistența stării la următoarea vizită. Implementarea este robustă, asigurându-se că salvează întotdeauna un string boolean ('true' sau 'false') pentru `isAdmin`.
- `clearAuth()`: Apelată la logout. Are rolul de a reseta complet starea de autentificare:
 - Setează `token` și `isAdmin` la valorile lor inițiale (null și false).
 - Șterge cheile corespunzătoare din `localStorage`, completând procesul de deconectare.
- `async initializeAuth()`: O acțiune asincronă esențială pentru sincronizarea datelor specifice utilizatorului la încărcarea aplicației sau după login.
 - Verifică dacă există un token.
 - Dacă da, face un apel API securizat (prin adăugarea header-ului `Authorization`) către endpoint-ul `/cart/` pentru a obține numărul de produse din coș.
 - Actualizează `this.cartCount`, ceea ce determină re-randarea automată a indicatorului numeric de pe iconița de coș din `Navbar`.
 - Include o logică robustă de eroare: dacă token-ul a expirat și serverul returnează 401 sau 403, acțiunea `clearAuth()` este apelată automat pentru a deloga utilizatorul.

Prin această structură modulară și centralizată, store-ul `auth.js` devine "creierul" aplicației, decuplând eficient componentele de logica de business și facilitând o dezvoltare scalabilă și ușor de întreținut.

3.3.3 Interceptarea și centralizarea apelurilor API

O provocare majoră în aplicațiile frontend complexe este gestionarea apelurilor către API-ul backend într-un mod curat și consistent. Repetarea logicii de construcție a URL-urilor și de adăugare a token-urilor de autentificare în fiecare componentă este ineficientă și predispusă la erori. Pentru a rezolva această problemă în proiectul Craftique, a fost implementat un pattern avansat, folosind interceptorii oferiți de biblioteca axios.

A fost creat un fișier dedicat, `src/plugins/axios.js`, care definește o instanță axios personalizată, numită `apiClient`. Această instanță este apoi importată și utilizată în întreaga aplicație în locul instanței axios implicite. Scopul acestui fișier este de a centraliza și automatiza logica de pregătire a fiecărei cereri HTTP.

```
const apiClient = axios.create();

apiClient.interceptors.request.use(
  (config) => {
    const authStore = useAuthStore();
    const configStore = useConfigStore();
    config.baseURL = configStore.baseURL;
    const token = authStore.token;
    if (token) {
      config.params = {...config.params, token: token};
    }
    return config;
  },
  (error) => {
    return Promise.reject(error);
  }
);
export default apiClient;
```

Interceptorul de request, implementat prin `apiClient.interceptors.request.use(...)`, este o funcție care „prinde” fiecare cerere înainte ca aceasta să fie trimisă către server. Acesta primește obiectul de configurare a cererii (`config`) ca parametru, îl modifică, și apoi îl returnează pentru a continua procesul. În Craftique, interceptorul îndeplinește două sarcini esențiale:

- Setarea automată a `baseURL`: La fiecare apel, interceptorul accesează store-ul `config.js` și adaugă automat `baseURL`-ul (<http://localhost:8081/>). Astfel, URL-ul final devine <http://localhost:8081/products>. Acest lucru decuplează componentele de adresa specifică a backend-ului; o eventuală schimbare a domeniului serverului necesită modificarea unei singure linii de cod, în `config.js`.
- Adăugarea automată a token-ului de autentificare: aceasta este logica de securizare. Interceptorul accesează store-ul `auth.js` pentru a verifica dacă un utilizator este logat (adică dacă `authStore.token` există).
 - Dacă un token este găsit, acesta este adăugat automat ca parametru de query la URL-ul cererii. De exemplu, o cerere POST către

/admin/categories/create va fi transformată automat într-o cerere către /admin/categories/create?token=...

- Operatorul Spread (...config.params): a fost utilizat pentru a asigura că orice alți parametri de query existenți pe cerere sunt păstrați, iar parametrul token este doar adăugat la listă.

Prin implementarea acestui interceptor, codul din componente a devenit semnificativ mai curat și mai concentrat pe logica sa specifică.

3.3.4 Rutarea avansată și securizarea navigării

Vue Router reprezintă componenta principală când vine vorba de crearea unei experiențe SPA și în securizarea accesului către diferite secțiuni ale aplicației.

a) Rute îmbricate (nested routes) pentru panoul de administrare

A fost definită o nouă rută părinte /admin care încarcă o componentă de afișare, AdminPage.vue. Această componentă conține elemente comune ale panoului de administrare, precum un meniu lateral pentru a ușura accesul la celelalte componente. Toate celelalte pagini de administrare (de exemplu: lista de categorii, formularul de adăugare a unui produs) sunt definite ca și rute copil (children). Astfel, navigarea între paginile de administrare vor schimba doar zona de conținut principal, astfel păstrând meniul lateral constant, ceea ce va oferi o experiență de utilizare unitară.

```
{
  path: '/admin',
  component: Admin,
  meta: { requiresAdmin: true },
  children: [
    {
      path: '',
      name: 'AdminDashboard',
      component: AdminDashboard,
    },
    {
      path: 'categories',
      name: 'Category',
      component: AdminCategories,
    },
    {
      path: 'category/add',
      name: 'AddCategory',
      component: AddCategory,
    },
    {
```

```

    path: 'category/edit/:id',
    name: 'EditCategory',
    component: EditCategory,
  },

  {
    path: 'products',
    name: 'AdminProduct',
    component: AdminProducts,
  },
  {
    path: 'product/add',
    name: 'AddProduct',
    component: AddProduct,
  },
  {
    path: 'product/edit/:id',
    name: 'EditProduct',
    component: EditProduct,
  },
],
}

```

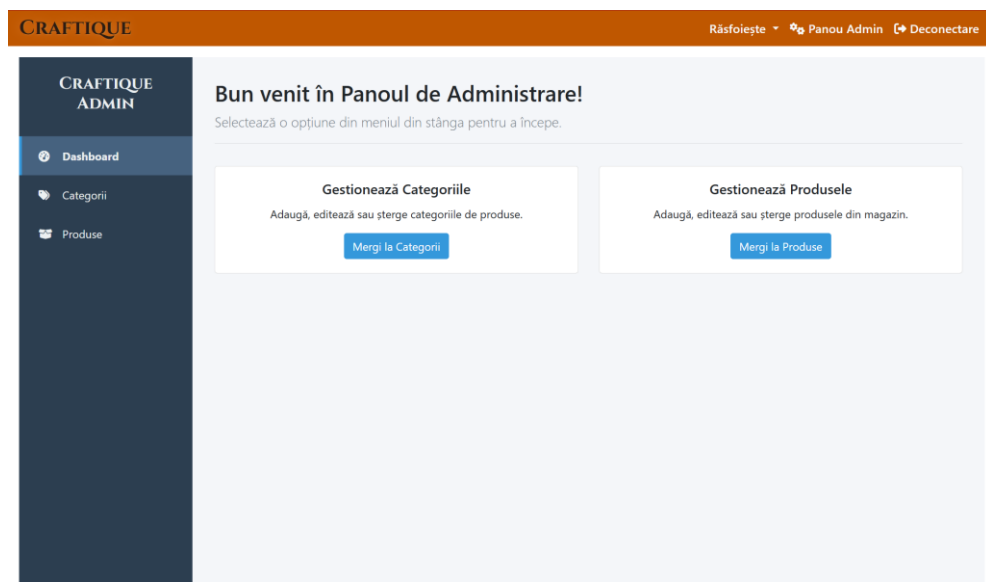


Figura 3.7. Panoul de administrare

b) Gardianul de navigație global (Navigation Guard)

Pentru protejarea rutelor, a fost implementat un gardian global folosind `router.beforeEach`. Acesta reprezintă o funcție care se execută înaintea fiecărei schimbări de rută și acționează ca un gardian al aplicației.

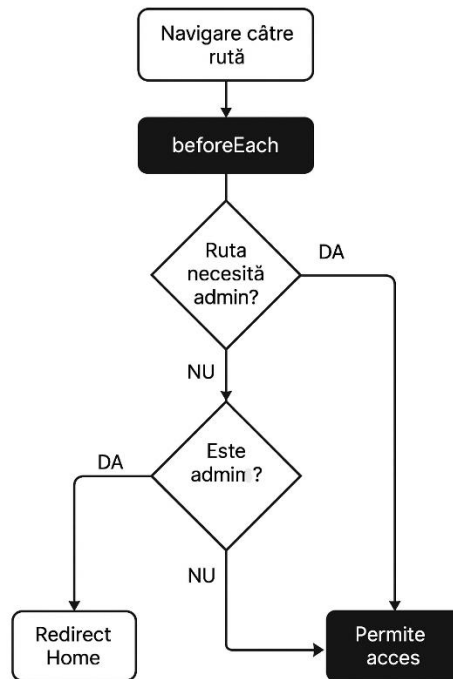


Figura 3.8. Diagrama de flux

Gardianul implementat verifică proprietățile meta ale unei rute solicitate:

- `meta: { requiresAdmin: true }`: dacă o rută are acest tag, gardianul face o verificare a store-ului Pinia dacă `authStore.isLoggedIn` și `authStore.isAdmin` sunt adevărate în același timp. În caz contrar, va redirecționa utilizatorul la pagina principală.
- `meta: { requiresAuth: true }`: Dacă o rută necesită doar autentificarea utilizatorului (de exemplu, `/cart`), gardianul va verifica doar dacă `authStore.isLoggedIn` este adevărat, altfel va redirecționa utilizatorul la pagina de autentificare.

```

router.beforeEach((to, from, next) => {
  const authStore = useAuthStore();

  const requiresAdmin = to.matched.some(record => record.meta.requiresAdmin);
  const requiresAuth = to.matched.some(record => record.meta.requiresAuth);

  if (requiresAdmin) {
    if (authStore.isLoggedIn && authStore.isAdmin) {
      next();
    } else {
      next({ name: 'Home' });
    }
  }
});
    
```

```

    }
    return;
  }

  if (requiresAuth) {
    if (authStore.isLoggedIn) {
      next();
    } else {
      next({ name: 'Signin' });
    }
  }
  return;
}

next();
});

```

Această implementare va asigura un nivel robust de securitate, oprind accesul neautorizat al clienților la URL-urile protejate.

3.3.5 Implementarea componentelor reactive „inteligente”

Paginile din directorul `/views` sunt construite ca și componente „inteligente”, responsabile pentru propria lor logică. Un exemplu potrivit este `AddProduct.vue`.

Figura 3.9. Formularul de adăugare a unui produs

Această componentă evidențiază mai multe concepte avansate:

- Încărcarea de date la montare: se folosește `onMounted` pentru a apela o funcție asincronă care încarcă lista de categorii de la backend pentru a popula primul dropdown.

- Dropdown-uri dependente și reactive: a fost implementată o proprietate computed (`filteredImages`) care „reacționează” în funcție de selecția din dropdown-ul de categorii. Astfel, când este selectată o anumită categorie, această proprietate se re-evaluează automat și va returna lista filtrată de imagini corespunzătoare categoriei selectate anterior, populând instantaneu al doilea dropdown.
- Feedback vizual pentru utilizator: componenta gestionează stările de așteptare (pentru încărcarea categoriilor) și `isAdding` (pentru procesul de trimitere a formularului), afișând spinneri și dezactivând butoane pentru a oferi o experiență clară și a preveni acțiunile duplicate.

```
const filteredImages = computed(() => {
  if (!categoryId.value) return [];
  const selectedCategory = categories.value.find(cat => cat.id ===
categoryId.value);
  if (selectedCategory) {
    return imageLibrary.productsByCategory[selectedCategory.categoryName] ||
[];
  }
  return [];
});
```

3.3.6 Implementarea componentelor vizuale cheie

Pe lângă arhitectura de bază, funcționalitatea și aspectul vizual al platformei Craftique sunt realizate printr-un set de componente Vue modulare. În cele ce urmează, vor fi detaliate implementările celor mai reprezentative componente, ilustrând aplicarea practică a conceptelor de reactivitate, props și afișare condiționată.

a) Componenta `NavbarPart.vue`

Bara de navigare este un element de UI extrem de complex, deoarece starea sa se modifică în timp real la statusul de autentificare în aplicație și la rolul utilizatorului. Componenta a fost proiectată pentru a fi complet reactivă la schimbările din store-ul `Pinia auth.js`.

Bara de navigare implementează o logică clară pentru a afișa meniuri diferite pentru trei tipuri de utilizatori: utilizator neautentificat, utilizator tip client și utilizator tip administrator.

- Pentru utilizatori neautentificați: sunt afișate meniurile „Răsfoiește” și un meniu „Cont” cu opțiunile „Creare cont” și „Autentificare”. Deși pictograma de coș de cumpărături este vizibilă, acesta va redirecționa utilizatorul către pagina de autentificare.
- Pentru clienți: Meniul „Cont” se va schimba și va afișa o opțiune de acces la „Lista de dorințe” și o opțiune de „Deconectare”.
- Pentru administratori: Va fi vizibil un meniu către „Panoul de administrare”.

Acest comportament dinamic este realizat prin directive v-if care identifică direct starea din store-ul Pinia, precum în exemplul de mai jos:

```
<li v-if="authStore.isAdmin" class="nav-item">
  <router-link class="nav-link text-light" :to="{ name:
'AdminDashboard' }">
    <i class="fas fa-cogs"></i> Panou Admin
  </router-link>
</li>
<li v-if="authStore.isAdmin" class="nav-item">
  <a class="nav-link text-light" href="#" @click="signout">
    <i class="fas fa-sign-out-alt"></i> Deconectare
  </a>
</li>
```



Figura 3.10. Bara de navigare pentru un administrator



Figura 3.11. Bara de navigare pentru un client

Pe lângă afișarea condiționată, componenta administrează și evenimentele de click pentru deschiderea meniurilor dropdown și pentru deconectarea utilizatorului, unde este apelată acțiunea `clearAuth()` din store-ul Pinia, evidențiind fluxul complet de date.

b) Componentele `ProductBox.vue` și `CategoryBox.vue`

Aceste două componente stau la baza fundamentului vizual al listelor de produse și categorii. Ele au fost proiectate ca și componente de prezentare „proaste” (dumb components), a căror responsabilitate unică este de a afișa datele primite prin props, făcându-le astfel reutilizabile.

Ambele componente utilizează structura de „card” de la Bootstrap, pentru a asigura un aspect vizual unitar. Au fost aplicate tehnici avansate de CSS precum `display: flex` și `flex-direction: column` pe corpul cardului, pentru a garanta că butoanele de acțiune („Vezi detalii”, „Editează”) sunt întotdeauna aliniate la baza cardului, indiferent de lungimea textului.

Pentru a menține un aspect curat, trunchierea descrierilor lungi a fost implementată nu în template, ci într-o proprietate computed, care va returna descrierea completă sau o descriere prescurtată, în funcție de lungimea acesteia.

```
const truncatedDescription = computed(() => {
  if (props.product.description && props.product.description.length > 65) {
    return props.product.description.substring(0, 65) + '...';
  }
})
```

```
return props.product.description;
});
```

Elementul care permite acestor componente să fie flexibile este opțiunea `showEditButton`: Boolean. Componenta părinte va fi cea care decide dacă butonul trebuie afișat și va transmite această decizie prin prop. `ProductBox` și `CategoryBox` pur și simplu vor executa instrucțiunea.

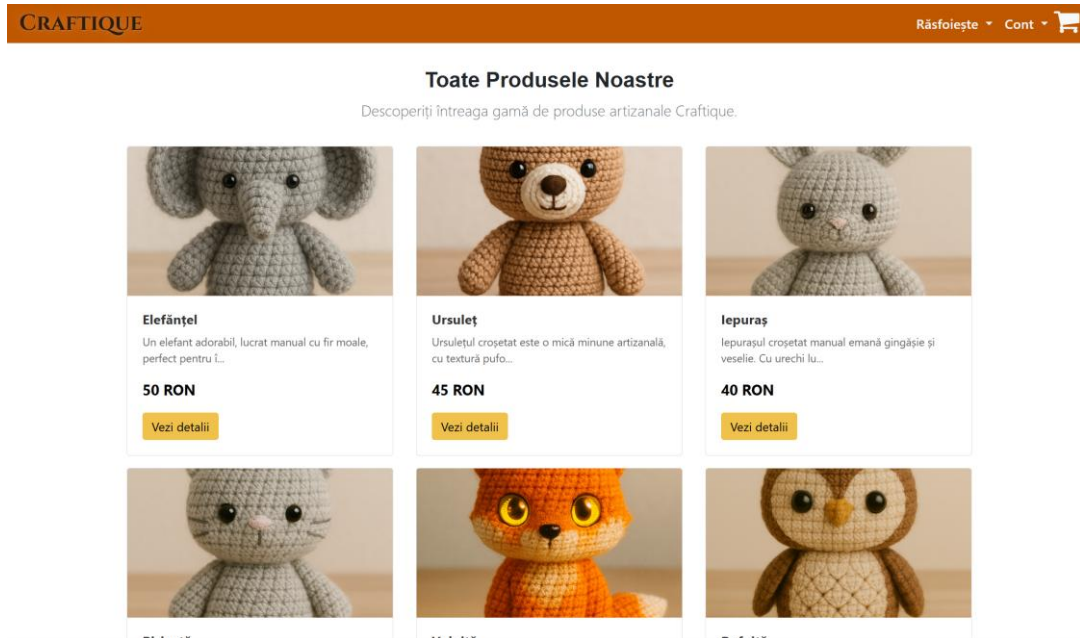


Figura 3.12. Pagina cu cardurile produselor adaptate clienților

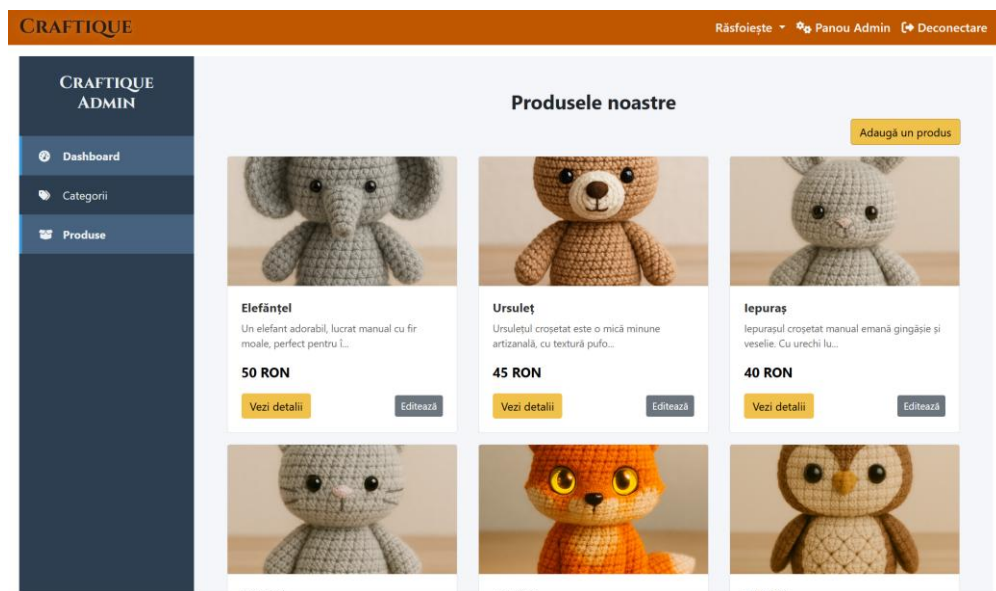


Figura 3.13. Pagina cu cardurile produselor adaptate administratorului

c) Componenta `EditCategory.vue` și `EditProduct.vue`

Formularele de editare implică o provocare suplimentară în comparație cu cele de adăugare: ele trebuie să încarce și să afișeze în primă fază datele existente ale entității, înainte de a permite modificarea lor.

The screenshot shows the 'Editează categoria' (Edit category) form in the CRAFTIQUE ADMIN interface. The form is located on the right side of the dashboard, with a sidebar on the left containing 'Dashboard', 'Categorii', and 'Produse'. The form fields are: 'Nume' (Name) with the value 'Ornamente de Crăciun Artizanale Rustice', 'Descriere' (Description) with the value 'Ornamente de Crăciun unice, lucrate manual cu drag.', and 'Imagine' (Image) with the value 'categorie-decoratiuni.png'. Below the image field is a preview image of Christmas decorations. At the bottom of the form is a yellow button labeled 'Salvează modificările' (Save changes).

Figura 3.15. Formularul de editare a unei categorii

The screenshot shows the 'Editează produsul' (Edit product) form in the CRAFTIQUE ADMIN interface. The form is located on the right side of the dashboard, with a sidebar on the left containing 'Dashboard', 'Categorii', and 'Produse'. The form fields are: 'Categorie' (Category) with the value 'Lumânări Artizanale Parfumate', 'Nume' (Name) with the value 'Trandafir', 'Descriere' (Description) with the value 'Parfumul romantic al trandafirilor se îmbină cu un design delicat. Această lumânare roz pal e perfectă pentru seri romantice sau momente de răsfăt.', 'Imagine' (Image) with the value 'produs-lumanari-trandafir.jpg', and 'Preț' (Price) with the value '35'. Below the image field is a preview image of a candle and roses. At the bottom of the form is a blue button labeled 'Salvează Modificările' (Save changes).

Figura 3.16. Formularul de editare a unui produs

Indiferent dacă este un produs sau o categorie, la accesarea unei rute de editare, componenta corespunzătoare devine autonomă și va iniția un proces de încărcare a datelor în hook-ul de ciclu de viață `onMounted`.

1. Extragerea ID-ului: folosind `useRoute()` de la Vue Router, ID-ul entității ce este editată este extras direct din parametrii URL.
2. Apeluri API asincrone: componenta realizează unul sau mai multe apeluri API folosind instanța `apiClient`.
3. Popularea stării locale: răspunsurile JSON de la server sunt folosite pentru a popula variabile reactive locale, care vor servi drept model de date pentru formular.


```
const fetchData = async () => {
  try {
    const [productResponse, categoriesResponse] = await Promise.all([
      apiClient.get(`products/${productId}`),
      apiClient.get('categories/')
    ]);

    product.value = productResponse.data;
    categories.value = categoriesResponse.data.data || categoriesResponse.data;

    if (product.value && !product.value.categoryId && product.value.category) {
      product.value.categoryId = product.value.category.id;
    }
  } catch (err) {
    console.error("Failed to fetch initial data:", err);
    error.value = "Nu s-au putut încărca datele necesare pentru editare.";
  } finally {
    loading.value = false;
  }
};
```

O dată ce obiectul reactiv (product sau category) a fost populat, puterea reactivității din Vue este folosită pentru a lega fiecare element al formularului la o proprietate corespunzătoare a obiectului, prin directiva v-model. Această tehnică asigură două lucruri simultan: pre-populare automată (la încărcarea datelor, câmpurile formularului vor afișa automat valorile curente ale entității) și sincronizare în timp real (orice modificare făcută în interfața grafică va actualiza instantaneu obiectul JavaScript din memorie).

La trimiterea formularului, funcția de edit este declanșată. Deoarece v-model a menținut obiectul product sau category sincronizat cu interfața, procesul de trimitere devine simplu:

1. Construirea obiectului DTO: se creează un nou obiect care conține doar câmpurile relevante, citite direct din proprietățile obiectivului reactiv;
2. Apel API de actualizare: se realizează un request POST la endpoint-ul corespunzător de update din backend, trimițând obiectul DTO în corpul cererii;
3. Feedback și redirectionare: după un răspuns de succes de la server, se afișează o notificare SweetAlert2 pentru a confirma acțiunea, iar utilizatorul este redirectionat automat la pagina de listare, unde sunt observate modificările efectuate.

```
const editCategory = async () => {
  isSaving.value = true;
  editError.value = null;

  const updatedCategory = {
    categoryName: category.value.categoryName,
    description: category.value.description,
```

```
        imageUrl: category.value.imageUrl,
    };
    try {
        await apiClient.post(`admin/categories/update/${categoryId}`,
updatedCategory);

        await Swal.fire({
            title: "Succes!",
            text: "Categoria a fost actualizată.",
            icon: "success",
        });

        router.push({ name: 'Category' });
    } catch (err) {
        console.error("Failed to edit category:", err);
        editError.value = "A apărut o eroare la salvarea modificărilor.";
    } finally {
        isSaving.value = false;
    }
};
```

4 Concluzii

4.1 Rezultate obținute

La finalul acestui proiect, obiectivul principal al lucrării, proiectarea și implementarea unei platforme de eCommerce complet funcționale, Craftique, a fost atins cu succes. Rezultatul concret constă într-o aplicație web full-stack, modernă și robustă, care demonstrează aplicarea practică a unor arhitecturi și tehnologii de vârf din industria software actuală.

Rezultatele obținute pot fi sintetizate astfel:

1. Dezvoltarea unei arhitecturi decuplate și scalabile: s-a realizat cu succes o separare clară între componenta backend (API RESTful construit în Spring Boot) și componenta frontend (Single-Page Application construită în Vue.js). Această arhitectură API-first a ajutat la o dezvoltare modulară, a asigurat o scalabilitate superioară și a permis ca cele două componente să evolueze independent și să deservească în viitor și alți clienți.
2. Implementarea unui backend robust: a fost construit un server de backend aproape complet, capabil să gestioneze logica de business esențială unui magazin online. Prin utilizarea Spring Data JPA, s-a realizat o persistență

eficientă a datelor într-o bază de date MySQL, iar prin Spring Boot, s-a expus un set de endpoint-uri REST intuitive și conforme cu standardele.

3. Construirea unei interfețe utilizator moderne și reactive: s-a dezvoltat un frontend interactiv și performant cu Vue.js 3. Prin adoptarea Composition API și a managementului stării cu Pinia, s-a creat o interfață cu componente decuplate, reutilizabile și ușor de întreținut. Mecanismul de „state management” a fost esențial pentru sincronizarea în timp real a datelor între diverse părți ale UI-ului.
4. Implementarea unui sistem funcțional de roluri: a fost realizată o diferențiere clară între funcționalitățile pentru clienți și pentru administratori. Aceasta a fost implementată pe trei niveluri: vizual (ascunderea butoanelor), de navigare (protejarea rutelor cu Vue Router Navigation Guards) și de API (verificare manuală a permisiunilor în controllerele de admin), rezultând un sistem sigur și coerent.

Comparativ cu platformele eCommerce monolitice tradiționale, arhitectura SPA decuplată a proiectului Craftique oferă o experiență de utilizare superioară. Viteza de navigare între pagini este semnificativ mai mare, deoarece se elimină reîncărcarea completă a paginii, aducând interacțiunea mai aproape de cea a unei aplicații native.

Față de studii sau proiecte ce se concentrează exclusiv pe o singură tehnologie, prezenta lucrare are o contribuție personală semnificativă în integrarea end-to-end a unui stack tehnologic modern și divers:

Contribuția principală constă în proiectarea și implementarea fluxului de date complet, de la interfața reactivă Vue.js, prin apelurile API securizate, până la logica de business și persistența în baza de date cu Spring Boot și JPA.

O altă contribuție personală este proiectarea și implementarea unui sistem de autorizare pragmatic, care, deși nu folosește întregul framework Spring Security, demonstrează o înțelegere solidă a principiilor de securizare a API-urilor prin validarea token-urilor la nivel de controller.

Dezvoltarea unei arhitecturi frontend bazate pe Pinia și Composition API, aliniată cu cele mai recente standarde din ecosistemul Vue.js, reprezintă o altă contribuție importantă, demonstrând abilitatea de a lucra cu paradigme moderne de dezvoltare.

În concluzie, lucrarea nu se rezumă doar la a construi un alt magazin online, ci reprezintă un studiu de caz complet despre cum tehnologiile moderne, alese și integrate corect, pot duce la crearea unor aplicații web performante, sigure și scalabile.

4.2 Direcții de dezvoltare

Platforma Craftique, în forma sa actuală, reprezintă o aplicație de bază solidă și complet funcțională. Cu toate acestea, orice proiect software este predispus spre evoluție. În cele ce urmează vor fi prezentate niște direcții de dezvoltare pentru a extinde și îmbunătăți semnificativ platforma:

1. Implementarea completă a securității cu Spring Security și JWT:

- Cea mai importantă dezvoltare viitoare ar fi înlocuirea mecanismului manual de verificare a token-urilor cu un sistem complet, bazat pe JSON Web Tokens și un filtru de autentificare personalizat în Spring Security. Acest lucru ar oferi un nivel de securitate de producție, permițând funcționalități precum expirarea token-urilor și o administrare mai robustă a permisiunilor prin adnotări.
- 2. Integrarea unui procesator de plăți real:
 - Simularea actuală a checkout-ului ar putea fi înlocuită cu o integrare reală cu un serviciu precum Stripe sau PayPal. Acest lucru ar implica integrarea unui sistem de plăți și gestionarea procesului de tranzacție, transformând aplicația într-o platformă de eCommerce complet funcțională.
- 3. Managementul comenzilor și al istoricului:
 - Crearea unei noi entități Order în baza de date pentru a salva permanent comenzile plasate de utilizatori reprezintă o altă idee de viitor esențială. Se pot adăuga noi secțiuni, precum „Istoric comenzi” pentru clienți, și o interfață de management al comenzilor în panoul de administrare.
- 4. Funcționalități avansate de eCommerce:
 - Filtrarea și sortarea produselor: adăugare de opțiuni de filtrare (după preț, nouitate) și sortare în paginile de listare a produselor.
 - Sistem de căutare: implementare unei bare de căutare pentru interogarea API-ului backend pentru a găsi produse după nume sau descriere,
 - Recenzii și rating: permiterea clienților de a lăsa recenzii și să acorde rating-uri produselor achiziționate.
- 5. Testare automată:
 - Pentru a asigura o fiabilitate pe termen lung, este sugerată adăugarea testelor automate: teste unitare pentru logica de business din backend și teste de componente pentru frontend.

Aceste direcții de dezvoltare pot transforma prototipul actual Craftique într-o platformă de eCommerce matură, complexă și pregătită pentru mediul de producție.

5 Bibliografie

[1] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. Sebastopol, CA: O'Reilly Media, 2015.

[2] M. Fowler, "MonolithFirst," 2014. [Online].
Available: <https://martinfowler.com/bliki/MonolithFirst.html>.

[3] Google Developers, "Rendering on the Web," 2020. [Online].
Available: <https://web.dev/articles/rendering-on-the-web>.

[4] A. Freeman, *Pro Angular 9*. New York, NY: Apress, 2020.

- [5] D. Abramov, "Presentational and Container Components," 2015. [Online]. Available: https://medium.com/@dan_abramov/presentational-and-container-components-a0066c0d87a5.
- [6] The Vue.js Team, "Official Vue.js Documentation," 2022. [Online]. Available: <https://vuejs.org/>.
- [7] D. Abramov and A. Clark, "Redux Documentation," 2023. [Online]. Available: <https://redux.js.org/>.
- [8] E. Posva, "Pinia Documentation," 2022. [Online]. Available: <https://pinia.vuejs.org/>.
- [9] C. Walls, *Spring in Action, 5th ed.*. Shelter Island, NY: Manning Publications, 2019.
- [10] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, Dept. of Information and Computer Science, University of California, Irvine, 2000.
- [11] The Spring Team, "Spring Boot Reference Documentation," 2023. [Online]. Available: <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/>.
- [12] Oracle Corporation, "MySQL 8.0 Reference Manual," 2023. [Online]. Available: <https://dev.mysql.com/doc/refman/8.0/en/>.
- [13] L. Richardson and M. Ruby, *RESTful Web APIs*. Sebastopol, CA: O'Reilly Media, 2013.
- [14] Eclipse Foundation, "Jakarta Persistence 3.1 Specification," 2022. [Online]. Available: <https://jakarta.ee/specifications/persistence/3.1/>.
- [15] The Axios Community, "Axios Documentation," 2023. [Online]. Available: <https://axios-http.com/docs/intro>.
- [16] SmartBear, "What is API-First Design?," 2021. [Online]. Available: <https://swagger.io/resources/articles/what-is-api-first-design/>.
- [17] K. Beck *et al.*, "Manifesto for Agile Software Development," 2001. [Online]. Available: <https://agilemanifesto.org/>.
- [18] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA: Addison-Wesley, 2002.
- [19] Baeldung, "Spring DTOs for REST APIs," 2023. [Online]. Available: <https://www.baeldung.com/java-dto-pattern>.