UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Altex E-commerce

Programare Software

Derivarable 1

Balint Cătălin-Vasile
30236/1

Facultatea de Automatica si Calculatoare

Cuprins

1 Specificațile proiectului

1.1 Scopul proiectului

Proiectul propus își propune să creeze o imitație sau o clonă a retailerului de succes ALTEX, aducând o experiență de cumpărături online similară și oferind o gamă largă de produse electronice și electrocasnice. Scopul său este de a satisface nevoile clienților din întreaga țară, oferindu-le acces ușor și convenabil la produse de calitate, la prețuri competitive și cu servicii de înaltă calitate.

1.2 Obiectivele proiectului

Obiectivele proiectului sunt strâns legate de utilizarea tehnologiilor full stack, în special Spring și React, pentru a atinge nivelul optim de funcționalitate și experiență pentru utilizatori. Prin implementarea acestor obiective, ne propunem să valorificăm beneficiile și capacitățile acestor tehnologii pentru a crea o platformă de comerț electronic robustă și eficientă.

Reproducerea Experienței ALTEX Online: Utilizarea tehnologiilor full stack, cum ar fi Spring pentru backend și React pentru frontend, ne permite să reproducem experiența ALTEX în mediul online într-un mod eficient și autentic. Cu ajutorul Spring, putem gestiona logică de afaceri complexă și interacțiunea cu baza de date, în timp ce React ne permite să construim interfețe de utilizator moderne și interactive care să ofere o experiență similară cu cea a magazinelor fizice.

Diversitate și Disponibilitate a Produselor: Prin intermediul Spring Data JPA și a frameworkului React.js, putem gestiona eficient o gamă variată de produse și categorii în cadrul platformei noastre. Spring Data JPA facilitează interacțiunea cu baza de date și manipularea datelor, în timp ce React.js ne permite să prezentăm aceste produse într-un mod atrăgător și ușor de navigat pentru utilizatori.

Convenienț și Accesibilitate: Utilizând Spring Boot pentru backend și React.js pentru frontend, putem crea o experiență de cumpărături online convenabilă și accesibilă pentru utilizatori. Spring Boot ne permite să dezvoltăm rapid și să configurăm ușor aplicația noastră, în timp ce React.js ne oferă o interfață de utilizator reactivă și plină de funcționalități pentru a facilita navigarea și cumpărarea produselor.

Servicii de Înaltă Calitate: Cu ajutorul Spring Security și a altor module din cadrul Spring, putem asigura securitatea și confidențialitatea datelor utilizatorilor noștri, precum și să oferim o experiență de utilizator sigură și protejată împotriva fraudelor și a altor amenințări online. În plus, putem utiliza funcționalitățile avansate de gestionare a sesiunilor și a autentificării din Spring pentru a oferi o experiență de utilizare fără probleme și plăcută.

1.3 Cerințele proiectului

Platforma propusă va reprezenta un exemplu de excelență în domeniul comerțului electronic, folosind cele mai recente tehnologii și practici în dezvoltarea aplicațiilor full stack. Backendul va fi implementat folosind cadrul de lucru Spring Java, cunoscut pentru robustețea sa și capacitățile sale de gestionare a datelor și logică de afaceri. Acesta va asigura funcționarea fluentă a operațiilor CRUD pe entități, precum și gestionarea relațiilor complexe între entități, prin intermediul unui ORM (Object-Relational Mapping) eficient.

Frontend-ul va fi dezvoltat cu React.js, o bibliotecă JavaScript modernă și puternică, care va oferi o interfață de utilizator elegantă și interactivă. Utilizatorii vor beneficia de o experiență

de cumpărături online plăcută și intuitivă, cu interfețe prietenoase și funcționalități avansate pentru navigare, căutare și comparare a produselor.

Platforma va integra accesul diferit pentru utilizatorii obișnuiți și administratori, oferind funcționalități specifice pentru fiecare categorie de utilizatori. Utilizatorii obișnuiți vor avea posibilitatea de a naviga și a cumpăra produse, precum și de a-și gestiona conturile personale și istoricul comenzilor, în timp ce administratorii vor avea acces la funcționalități extinse pentru gestionarea produselor, utilizatorilor și comenzilor.

Cu o abordare profesională și integrată a dezvoltării, platforma va oferi o soluție robustă și scalabilă, menită să satisfacă cerințele și așteptările clienților săi. Integrarea tehnologiilor Spring Java și React.js va asigura performanțe excelente și o experiență de utilizare de neegalat, consolidându-și poziția ca un lider în domeniul comerțului electronic.

1.4 Arhitectura proiectului

Arhitectura proiectului se bazează pe o abordare modernă de tip client-server, în care frontend-ul și backend-ul comunică între ele pentru a oferi o experiență de utilizare fluidă și plăcută. Pe partea de backend, am adoptat o arhitectură bazată pe model-view-controller (MVC), folosind cadrul de lucru Spring Java. Am organizat codul în pachete distincte, cum ar fi modele pentru clasele Java reprezentând entitățile, repository pentru definirea interfețelor care extind JpaRepository pentru manipularea datelor în baza de date, și controlere pentru gestionarea endpoint-urilor care facilitează comunicarea dintre frontend și backend.

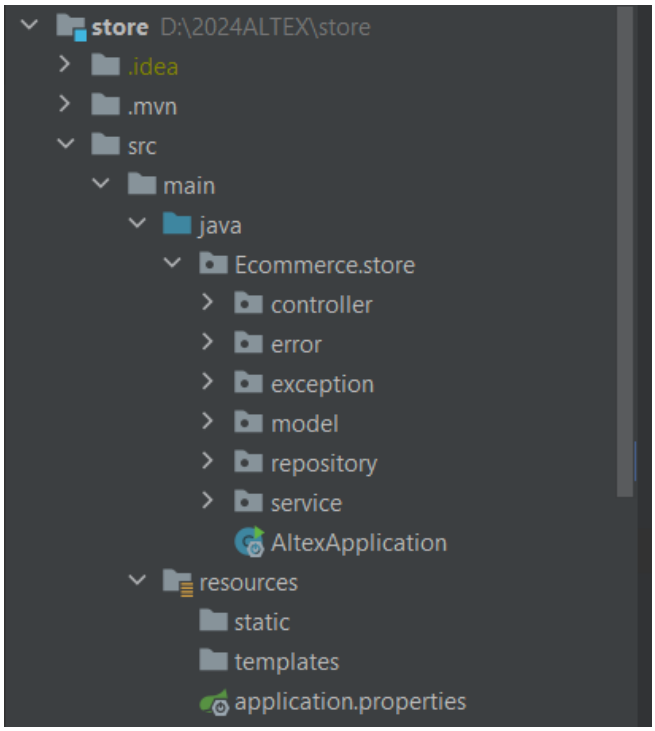
Pentru comunicarea cu baza de date, am optat pentru MySQL, un sistem de gestionare a bazelor de date relaționale puternic și fiabil, care utilizează portul JDBC 3306 pentru conexiunea la server. Această alegere se datorează fiabilității și performanței pe care MySQL le oferă în manipularea datelor.

Pe partea de frontend, am ales să utilizăm React.js, o bibliotecă JavaScript populară și eficientă, care oferă o abordare component-based pentru dezvoltarea interfețelor de utilizator. Această decizie a fost luată datorită flexibilității, performanței și scalabilității oferite de React.js în dezvoltarea de aplicații web moderne.

Pentru gestionarea datelor în backend, am folosit Spring Data JPA, împreună cu dependințele Lombok, Spring Web și MySQL Driver, pentru a asigura o integrare simplă și eficientă între aplicație și baza de date. Aceste tehnologii au fost alese datorită capacității lor de a accelera dezvoltarea și de a reduce cantitatea de cod necesară pentru a implementa funcționalități complexe în backend.

1.5 Ierarhia structurală

În cadrul aplicației noastre, am organizat clasele și interfețele în diferite pachete pentru a menține o ierarhie coerentă și ușor de gestionat. Această ierarhie de pachete și clase este esențială pentru structurarea logică a aplicației noastre și pentru gestionarea diferitelor aspecte ale funcționalității sale. În continuare, vom explora ierarhia de pachete și clase din aplicație.



Pachetul de bază al aplicației noastre este ‘Ecommerce.store‘, care conține întreaga logică a magazinului nostru online. În acest pachet, avem mai multe subpachete care împart funcționalitatea aplicației în module distincte pentru o gestionare mai ușoară și o organizare mai clară.

Unul dintre aceste subpachete este ‘controller‘, care conține clasele care gestionează cererile HTTP și răspunsurile asociate. Aici găsim clasele ‘AdminController‘, ‘PersonController‘ și ‘ProductCategoryController‘, care sunt responsabile pentru gestionarea operațiunilor asociate administratorilor, utilizatorilor și produselor din magazinul nostru.

În pachetul ‘model‘, definim clasele care reprezintă obiectele de bază ale aplicației noastre. Acestea includ clasele ‘Admin‘, ‘Category‘, ‘Person‘, ‘Product‘ și ‘User‘, care descriu entitățile noastre principale și relațiile între ele.

Pachetul ‘repository‘ conține interfețele și clasele care se ocupă de interacțiunea cu baza de date. Aici avem clasele ‘AdminRepository‘, ‘CategoryRepository‘, ‘PersonRepository‘, ‘ProductRepository‘ și ‘UserRepository‘, care sunt responsabile pentru realizarea operațiilor de bază cu entitățile respective.

În ceea ce privește gestionarea erorilor și excepțiilor, acestea sunt tratate în pachetul ‘error‘. Aici găsim clasa ‘UserNotFoundAdvice‘, care furnizează sfaturi personalizate pentru gestionarea excepțiilor legate de utilizatori care nu pot fi găsiți în sistem.

În plus, avem clasa ‘AltexApplication‘, care servește drept punct de intrare principal pentru aplicația noastră Spring Boot. Aceasta inițiază contextul aplicației și configurează setările inițiale necesare pentru funcționarea corectă a acesteia.

Ierarhia de pachete și clase din aplicația noastră oferă o structură bine definită și organizată, care facilitează dezvoltarea, testarea și întreținerea aplicației noastre de comerț electronic. Prin împărțirea funcționalității în module separate și clar definite, asigurăm un cod curat și ușor de gestionat, care să satisfacă cerințele și standardele noastre de calitate.

1.6 Dependințele proiectului

Pachetul Lombok reprezintă o alegere esențială pentru a simplifica dezvoltarea aplicației noastre Java. Lombok ne permite să reducem semnificativ cantitatea de cod boilerplate prin generarea automată a metodelor getter, setter și constructori, precum și a altor metode comune. Astfel, putem menține codul nostru mai curat și mai concis, concentrându-ne pe logica de afaceri a aplicației.

```
34
35
36                  <dependency>
37                          <groupId>org.projectlombok</groupId>
38                          <artifactId>lombok</artifactId>
39                          <optional>true</optional>
40                  </dependency>
```

Spring Web este o dependență crucială pentru dezvoltarea backend-ului aplicației noastre. Aceasta furnizează funcționalitățile necesare pentru crearea de aplicații web robuste și scalabile folosind cadrul de lucru Spring. Cu ajutorul Spring Web, putem defini controlerele noastre, gestiona cererile HTTP și răspunsurile, precum și configurațiile de securitate și gestionarea sesiunilor.

```
24                  <dependency>
25                          <groupId>org.springframework.boot</groupId>
26                          <artifactId>spring-boot-starter-web</artifactId>
27                  </dependency>
```

Spring Data JPA reprezintă o componentă esențială pentru interacțiunea cu baza de date în cadrul aplicației noastre. Aceasta oferă un set de abstracții și instrumente puternice pentru gestionarea operațiilor CRUD și a relațiilor între obiecte și tabele în baza de date. Utilizând Spring Data JPA, putem simplifica și accelera procesul de dezvoltare a accesului la date, permițându-ne să ne concentrăm mai mult pe logica de afaceri a aplicației.

```
20                  <dependency>
21                          <groupId>org.springframework.boot</groupId>
22                          <artifactId>spring-boot-starter-data-jpa</artifactId>
23                  </dependency>
```

MySQL Driver este o dependență esențială pentru conectarea aplicației noastre la baza de date MySQL. Acesta furnizează un set de instrumente și funcționalități necesare pentru a stabili și gestiona conexiunea între backend și baza de date MySQL. Prin intermediul MySQL Driver, putem efectua operațiuni de citire și scriere în baza de date, asigurând persistența datelor și integritatea acestora în cadrul aplicației noastre full-stack.

```
29                  <dependency>
30                          <groupId>com.mysql</groupId>
31                          <artifactId>mysql-connector-j</artifactId>
32                          <scope>runtime</scope>
33                  </dependency>
```

2 Cerintele funcționale

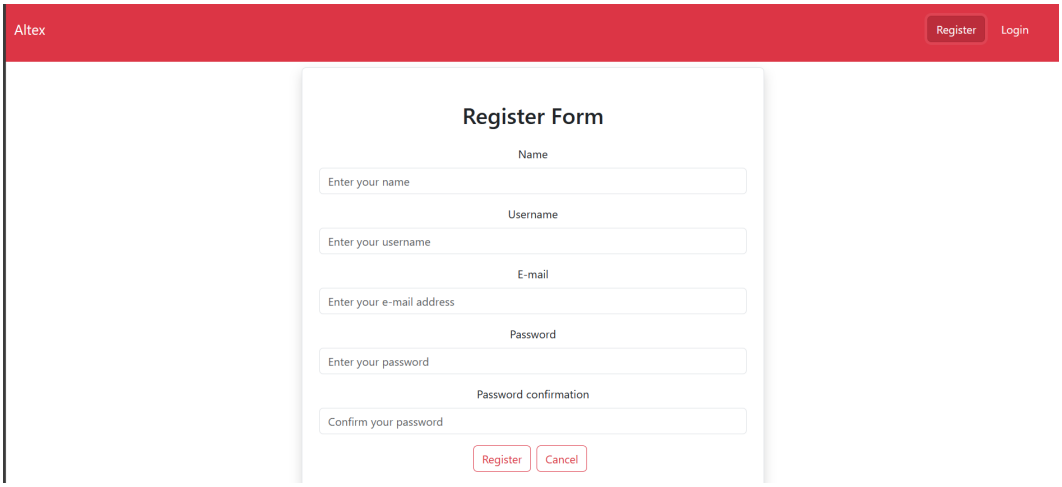
2.1 Sign up

Sistemul trebuie să permită utilizatorilor să se înregistreze furnizându-și adresa de e-mail și o parolă. La înregistrare, utilizatorii trebuie să poată alege un rol (USER sau ADMIN). Cu toate acestea, pentru a menține simplificată experiența de înregistrare, sistemul poate să permită doar înregistrarea ca USER, dar să aibă deja un ADMIN existent în baza de date.

Pentru funcționalitatea de Sign Up, sistemul trebuie să ofere utilizatorilor posibilitatea de a-și crea conturi furnizând adresa lor de e-mail și o parolă. În momentul înregistrării, utilizatorii ar trebui să poată selecta un rol, fie ca USER sau ADMIN. Totuși, pentru a menține o experiență de înregistrare simplificată, s-ar putea permite doar înregistrarea ca USER, având deja un ADMIN existent în baza de date. În acest mod, se asigură că doar utilizatorii obișnuiți pot înregistra conturi noi, în timp ce administratorii sunt predefiniți în sistem.

Pentru a garanta unicitatea adresei de e-mail furnizate, sistemul trebuie să efectueze o verificare în baza de date și să furnizeze un mesaj de eroare în cazul în care adresa de e-mail este deja înregistrată. În plus, pentru a asigura corectitudinea introducerii parolei, utilizatorului i se va cere să reintroducă parola într-un câmp de confirmare a parolei. Doar parola va fi stocată în baza de date, întrucât nu este necesar să se stocheze parțial sau complet datele sensibile ale utilizatorilor.

Implementarea acestor cerințe ar trebui să ofere o experiență de autentificare și înregistrare eficientă și securizată pentru utilizatorii sistemului, asigurând în același timp că informațiile necesare sunt colectate și stocate în mod adecvat în baza de date.



Metoda POST este utilizată pentru a permite utilizatorilor să își înregistreze conturi noi. Atunci când un utilizator trimite o cerere HTTP POST către endpoint-ul /person, această cerere este gestionată de clasa PersonController, care este responsabilă pentru tratarea cererilor legate de persoane.

Metoda newPerson din PersonController primește obiectul JSON care conține detaliile persoanei nou înregistrate. În primul rând, această metodă apelează serviciul PersonService pentru a seta rolul persoanei ca USER, folosind metoda savePersonRole. Aceasta este o etapă esențială pentru a asigura că toți utilizatorii noi sunt înregistrați cu rolul corespunzător.

```
11 @Service
12 public class PersonService {
```

```
13
14      @Autowired
15      private PersonRepository personRepository;

29      @Transactional
30      public void sentToEntity(Person person) {
31          if (person.getPersonRole().equals(PersonRole.ADMIN)){
32              Admin admin = new Admin();
33              admin.setPerson(person);
34              adminRepository.save(admin);
35              person.setAdmin(admin);
36              personRepository.save(person);
37
38          } else if (person.getPersonRole().equals(PersonRole.USER)){
39              User user = new User();
40              user.setPerson(person);
41              userRepository.save(user);
42              person.setUser(user);
43              personRepository.save(person);
44          }
45      }
```

După ce rolul persoanei este setat, obiectul Person este salvat în baza de date utilizând obiectul PersonRepository. În același timp, metoda sentToEntity din PersonService este apelată pentru a gestiona crearea entităților asociate și stabilirea relațiilor între acestea și persoană.

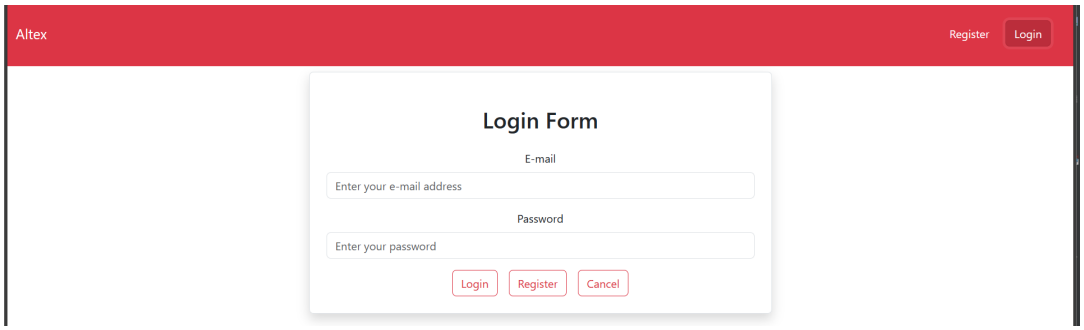
În cadrul metodei sentToEntity, se verifică rolul persoanei și se creează și se salvează obiectele User sau Admin corespunzătoare în funcție de acesta. Relațiile între persoană și entitățile asociate sunt stabilite, iar obiectul Person actualizat, care include informațiile generate de baza de date, este salvat din nou.

Astfel, prin intermediul acestei metode POST, aplicația noastră permite înregistrarea eficientă și securizată a utilizatorilor noi, asigurând că toate informațiile necesare sunt gestionate și stocate corespunzător în baza de date.

```
25      @PostMapping("/person")
26      Person newPerson(@RequestBody Person newPerson) {
27          personService.savePersonRole(newPerson);
28          personRepository.save(newPerson);
29          personService.sentToEntity(newPerson);
30          return personRepository.save(newPerson);
31      }
```

2.2 Sign in

Metoda loginUser este o parte critică a sistemului nostru de autentificare și este responsabilă pentru gestionarea cererilor de autentificare trimise de către utilizatori. Această metodă este implementată ca un endpoint de tip GET și este accesibilă prin URL-ul /login. În urma unei cereri HTTP GET către acest endpoint, metoda primește două parametri: adresa de e-mail și parola utilizatorului. Acești parametri sunt trimiși ca parametri de interogare în cadrul cererii HTTP.



Principalele adnotări folosite pentru a configura această metodă sunt @GetMapping și @RequestParam. Adnotarea @GetMapping specifică că această metodă va fi apelată atunci când este trimisă o cerere HTTP GET către endpoint-ul /login. Pe de altă parte, adnotarea @RequestParam este folosită pentru a extrage parametrii de interogare din URL-ul cererii HTTP și pentru a le asocia variabilelor locale din metoda loginUser.

În interiorul metodei, se începe prin căutarea în baza de date a unei persoane care să corespundă adresei de e-mail primite ca parametru. Aceasta se realizează prin apelarea metodei findByEmail a obiectului personRepository. Rezultatul căutării este încapsulat într-un obiect Optional, care poate conține o persoană sau poate fi gol în cazul în care nu există o persoană cu adresa de e-mail specificată.

Dacă obiectul Optional conține o persoană, se verifică dacă parola furnizată de utilizator se potrivește cu parola stocată în baza de date pentru acea persoană. Aceasta se face prin comparația parolei primite ca parametru cu parola stocată în obiectul Person. Dacă parolele coincid, este returnat un răspuns HTTP 200 (OK) împreună cu obiectul Person asociat cu adresa de e-mail dată.

În caz contrar, dacă parolele nu coincid, este returnat un răspuns HTTP 401 (Unauthorized), semnalând că autentificarea a eșuat din cauza unei parole greșite. În plus, dacă nu este găsită nicio persoană cu adresa de e-mail specificată, este returnat un răspuns HTTP 404 (Not Found), indicând că adresa de e-mail nu este asociată cu niciun cont din sistem.

```
54      @GetMapping(value = "/login")
55      public ResponseEntity<Person> loginUser(@RequestParam String email, @RequestParam String
56          Optional<Person> optionalPerson = personRepository.findByEmail(email);
57
58          if (optionalPerson.isPresent()) {
59              Person person = optionalPerson.get();
60
61              if (person.getPassword().equals(password)) {
62                  return ResponseEntity.ok(person);
63              } else {
64                  return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();
65              }
66          } else {
67              return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
68          }
69      }
```

2.3 Acces User/Admin

În aplicația noastră, accesul utilizatorilor și administratorilor este gestionat în funcție de rolurile atribuite fiecărei persoane. Pentru a permite navigarea către paginile corespunzătoare în funcție de rolul utilizatorului sau administratorului, am implementat o metodă role în PersonController și o funcționalitate de rutare în codul JavaScript al aplicației.

În cadrul metodei role din PersonController, se primește adresa de e-mail și parola utilizatorului și se caută în baza de date o persoană cu aceste date de autentificare. Dacă persoana este găsită și parola corespunde cu cea stocată în baza de date, metoda returnează rolul asociat persoanei. În caz contrar, se returnează rolul implicit al persoanei.

```
71      @GetMapping(value = "/role")
72      public PersonRole role(@RequestParam String email, @RequestParam String password) {
73          Optional<Person> optionalPerson = personRepository.findByEmail(email);
74
75          if (optionalPerson.isPresent()) {
76              Person person = optionalPerson.get();
77
78              if (person.getPassword().equals(password)) {
79                  return person.getPersonRole();
80              }
81          }
82          return optionalPerson.get().getPersonRole() ;
```

În partea de frontend a aplicației, utilizăm o funcție onSubmit pentru a gestiona procesul de autentificare. După ce primim un răspuns de la endpoint-ul /login care confirmă autentificarea cu succes, efectuăm o altă cerere către endpoint-ul /role pentru a obține rolul asociat utilizatorului sau administratorului autentificat.

Dacă rolul returnat este ADMIN, rutăm aplicația către pagina destinată administratorului, iar dacă rolul este USER, rutăm către pagina destinată utilizatorului. Acest lucru se realizează cu ajutorul funcției navigate din pachetul @reach/router, care ne permite să navigăm între rutele definite în aplicație.

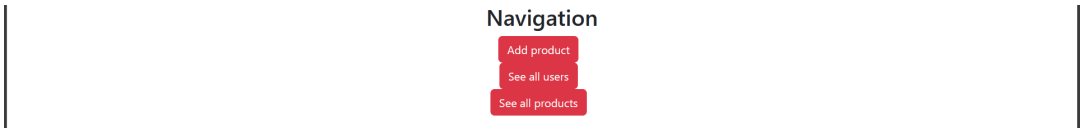
În acest mod, utilizatorii și administratorii sunt direcționați către interfețele corespunzătoare în funcție de rolurile lor, oferindu-le o experiență personalizată și securizată în cadrul aplicației noastre. Această abordare îmbunătățește nu numai accesul utilizatorilor la funcționalitățile adecvate, dar și securitatea generală a aplicației.

```
18    const onSubmit = async (e) => {
19      e.preventDefault();
20
21      try {
22        const response = await axios.get(
23          `http://localhost:8080/login?email=${email}&password=${password}`
24        );
25
26        const loggedInUser = response.data;
27
28        if (loggedInUser) {
29          console.log("Login successful:", loggedInUser);
```

```
30          const role = await axios.get(
31              `http://localhost:8080/role?email=${email}&password=${password}`
32          );
33          if (role.data=="ADMIN") {
34            navigate("/admin");
35          } else {
36            navigate("/user");
37          }
38        } else {
39          console.error("Login failed");
40        }
41      } catch (error) {
42        console.error("Error during login:", error);
43      }
44    };
```

2.4 CRUD-uri

Operațiile CRUD (Create, Read, Update, Delete) reprezintă un set de operații fundamentale utilizate în gestionarea datelor în cadrul sistemelor software. Fiecare operație are un rol distinct în manipularea informațiilor.



2.4.1 PersonController

Operația de creare (Create) este gestionată prin intermediul metodei newPerson, asociată cu ruta POST /person. Această metodă primește un obiect Person sub formă de corp de solicitare și îl salvează în baza de date utilizând personRepository.save(newPerson). Înainte de salvare, sunt efectuate și alte operații specifice, precum gestionarea rolurilor și trimiterea unor informații către alte entități.

Operația de citire (Read) este implementată în două metode distincte. Prima metodă, getAllPersons, asociată cu ruta GET /persons, returnează o listă care conține toate persoanele din baza de date. A doua metodă, getPersonById, asociată cu ruta GET /person/id, furnizează detaliile unei persoane specifice, identificate prin intermediul ID-ului său unic.

Operația de actualizare (Update) este gestionată de metoda updatePerson, asociată cu ruta PUT /user/id. Această metodă primește un obiect Person actualizat sub formă de corp de solicitare, împreună cu ID-ul persoanei care urmează să fie actualizată. Persoana este identificată în baza de date, iar dacă este găsită, se actualizează câmpurile relevante cu noile valori și se salvează modificările.

Operația de ștergere (Delete) este gestionată de metoda deletePerson, asociată cu ruta DELETE /person/id. Această metodă primește ID-ul persoanei care urmează să fie ștearsă. Înainte de a efectua ștergerea, se verifică dacă persoana există în baza de date. În cazul în care persoana nu este găsită, se aruncă o excepție UserNotFoundException, altfel persoana este ștearsă din baza de date.

Aceste operații CRUD furnizează funcționalitățile de bază necesare pentru gestionarea persoanelor în cadrul sistemului de comerț electronic, permitând crearea, citirea, actualizarea și ștergerea acestora.

2.4.2 ProductCategoryController

Operația de creare (Create) este gestionată prin intermediul metodei createProduct, asociată cu ruta POST /product. Această metodă primește un obiect Product sub formă de corp de solicitare și îl salvează în baza de date utilizând productRepository.save(product). În timpul procesului de creare a produsului, se verifică și se adaugă categoriile corespunzătoare în cazul în care acestea nu există deja.

Pentru operația de citire (Read), este furnizată metoda getAllProducts, asociată cu ruta GET /products, care returnează o listă ce conține toate produsele din baza de date. De asemenea, există și metoda getProductNameById, asociată cu ruta GET /product/id, care returnează numele unui produs specific identificat după ID-ul său.

Operația de actualizare (Update) este gestionată de metoda updateProductName, asociată cu ruta PUT /product/id. Această metodă primește un ID de produs și un obiect Product actualizat sub formă de corp de solicitare. Noul nume al produsului este actualizat, iar modificările sunt salvate în baza de date.

Operația de ștergere (Delete) este implementată prin metoda deleteProduct, asociată cu ruta DELETE /product/id. Această metodă primește ID-ul produsului care urmează să fie șters. Înainte de ștergere, toate referințele către produs în tabela productcategory sunt eliminate, iar apoi produsul însuși este șters din baza de date.

Aceste operații CRUD oferă funcționalitățile necesare pentru gestionarea produselor și categoriilor în cadrul sistemului de comerț electronic, inclusiv crearea, citirea, actualizarea și ștergerea acestora.

2.5 ORM relations

În ceea ce privește relația dintre o entitate Person și entitățile User și Admin, aceasta este definită printr-o relație de tip Many-to-One. Astfel, o persoană poate fi asociată cu un singur utilizator sau un singur administrator, iar un utilizator sau un administrator poate fi asociat cu mai multe persoane. În implementarea Java prezentată, s-au folosit adnotările @ManyToOne și @JoinColumn pentru a stabili aceste relații. Adnotarea @JsonIgnore a fost utilizată pentru a evita serializarea recursivă în cazul serializării obiectelor JSON, prevenind astfel buclele infinite. Aceste configurări asigură că structura relației este corect definită în cadrul aplicației și că interacțiunea între entități se realizează în mod corespunzător.

```
19      @ManyToOne(cascade = CascadeType.ALL)
20      @JoinColumn(name = "admin_id", nullable = true)
21      @JsonIgnore
22      private Admin admin;
23
24      @ManyToOne(cascade = CascadeType.ALL)
25      @JoinColumn(name = "user_id", nullable = true)
26      @JsonIgnore
27      private User user;
```

3 Use case UML

Diagrama de cazuri de utilizare (Use Case Diagram) în limbajul de modelare UML (Unified Modeling Language) este o tehnică utilizată în ingineria software pentru a defini și a prezenta interacțiunile dintre actori (utilizatori externi) și sistem. Aceasta oferă o perspectivă structurală asupra funcționalităților oferite de sistem și modul în care acestea sunt accesate de către actori.

Un caz de utilizare reprezintă o situație sau un scenariu în care sistemul este utilizat de către unul sau mai mulți actori pentru a atinge un anumit obiectiv. Aceste cazuri de utilizare sunt reprezentate sub formă de elipse în diagrama UML, iar relațiile dintre cazurile de utilizare și actori sunt ilustrate prin linii de asociere.

În diagrama de cazuri de utilizare, actorii sunt entitățile externe care interacționează cu sistemul, fie că sunt utilizatori umani, alte sisteme sau dispozitive externe. Cazurile de utilizare sunt acțiuni sau secvențe de acțiuni care descriu modul în care actorii interacționează cu sistemul pentru a atinge anumite obiective.

3.1 Use case USER

Use-Case: Gestionarea contului și vizualizarea produselor

Level: Principal (core)

Primary Actor: Utilizatorul (USER)

Main success scenario:

Utilizatorul dorește să își creeze un cont:

- Actorul accesează funcția de "Sign up".
- Sistemul solicită introducerea datelor necesare pentru înregistrare (ex: nume, email, parolă).
- Utilizatorul completează detaliile și confirmă înregistrarea.
- Sistemul validează datele și creează contul utilizatorului.
- Utilizatorul primește confirmarea că contul a fost creat cu succes.

Utilizatorul dorește să se autentifice în contul său existent:

- Actorul accesează funcția de "Sign in".
- Sistemul solicită introducerea adresei de email și a parolei.
- Utilizatorul furnizează detaliile de autentificare.
- Sistemul validează datele și autentifică utilizatorul.
- Utilizatorul primește confirmarea că autentificarea a fost efectuată cu succes.

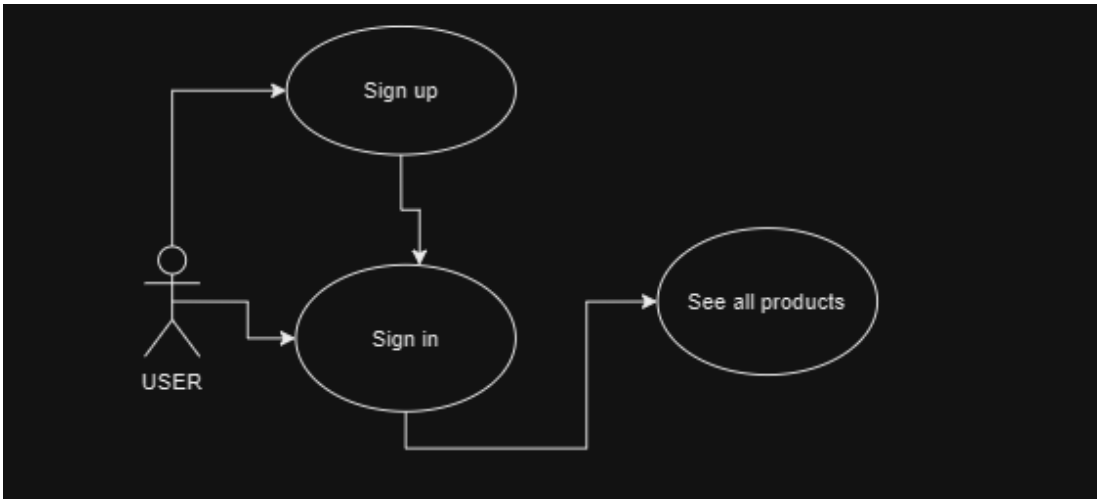
După autentificare, utilizatorul dorește să vizualizeze toate produsele disponibile:

- Actorul accesează funcția de "See all products".
- Sistemul afișează o listă cu toate produsele disponibile în magazin.

Extensions:

- Dacă datele introduse în timpul înregistrării nu sunt valide sau există deja un cont asociat cu adresa de email introdusă, sistemul afișează un mesaj de eroare și solicită utilizatorului să furnizeze informații valide sau să încerce să se autentifice în contul existent.
- Dacă datele de autentificare introduse nu sunt valide sau nu corespund cu un cont existent, sistemul afișează un mesaj de eroare și solicită utilizatorului să reintroducă datele corecte sau să își recupereze parola.

- Dacă utilizatorul autentificat nu are permisiunea de a vizualiza produsele (de exemplu, dacă contul său este suspendat sau închis), sistemul îl va redirecționa către o pagină corespunzătoare de eroare sau îl va informa că nu are acces la această funcționalitate.



3.2 Use case ADMIN

Use-Case: Gestionarea funcționalităților ca administrator
Level: Principal (core)
Primary Actor: Administratorul (ADMIN)
Main success scenario:

Administratorul dorește să se autentifice în sistem:

- Actorul accesează funcția de "Sign in".
- Sistemul solicită introducerea adresei de email și a parolei administratorului.
- Administratorul furnizează detaliile de autentificare.
- Sistemul validează datele și autentifică administratorul.
- Administratorul primește confirmarea că autentificarea a fost efectuată cu succes.

După autentificare, administratorul are trei opțiuni disponibile:

a) Adăugare de produse noi:
- Administratorul accesează funcția "Add Products".
- Sistemul permite introducerea detaliilor noului produs (ex: nume, descriere, preț, etc.).
- Administratorul completează informațiile necesare și confirmă adăugarea produsului.
- Sistemul validează datele și adaugă noul produs în baza de date.

b) Vizualizare a tuturor utilizatorilor:
- Administratorul accesează funcția "See All Users".
- Sistemul afișează o listă cu toți utilizatorii înregistrați în sistem.

c) Vizualizare a tuturor produselor:
- Administratorul accesează funcția "See All Products".
- Sistemul afișează o listă cu toate produsele disponibile în magazin.

După vizualizarea tuturor utilizatorilor, administratorul are trei opțiuni suplimentare:

a) Editare utilizator:
- Administratorul selectează opțiunea "Edit User" pentru un anumit utilizator.
- Sistemul permite administratorului să modifice informațiile utilizatorului (ex: nume, email, rol, etc.).
- Administratorul finalizează modificările, iar sistemul validează și actualizează datele utilizatorului în baza de date.

b) Vizualizare detalii utilizator:
- Administratorul selectează opțiunea "View User" pentru a vizualiza detaliile unui anumit utilizator (ex: nume, email, istoric de achiziții, etc.).

c) Ștergere utilizator:
- Administratorul selectează opțiunea "Delete User" pentru a șterge un anumit utilizator.
- Sistemul solicită confirmarea administratorului.
- După confirmare, utilizatorul este șters din baza de date.

După vizualizarea tuturor produselor, administratorul are trei opțiuni suplimentare:

a) Editare produs:
- Administratorul selectează opțiunea "Edit Product" pentru un anumit produs.
- Sistemul permite administratorului să modifice informațiile produsului (ex: nume, descriere, preț, etc.).
- Administratorul finalizează modificările, iar sistemul validează și actualizează datele produsului în baza de date.

b) Vizualizare detalii produs:
- Administratorul selectează opțiunea "View Product" pentru a vizualiza detaliile unui anumit produs (ex: nume, descriere, stoc disponibil, etc.).

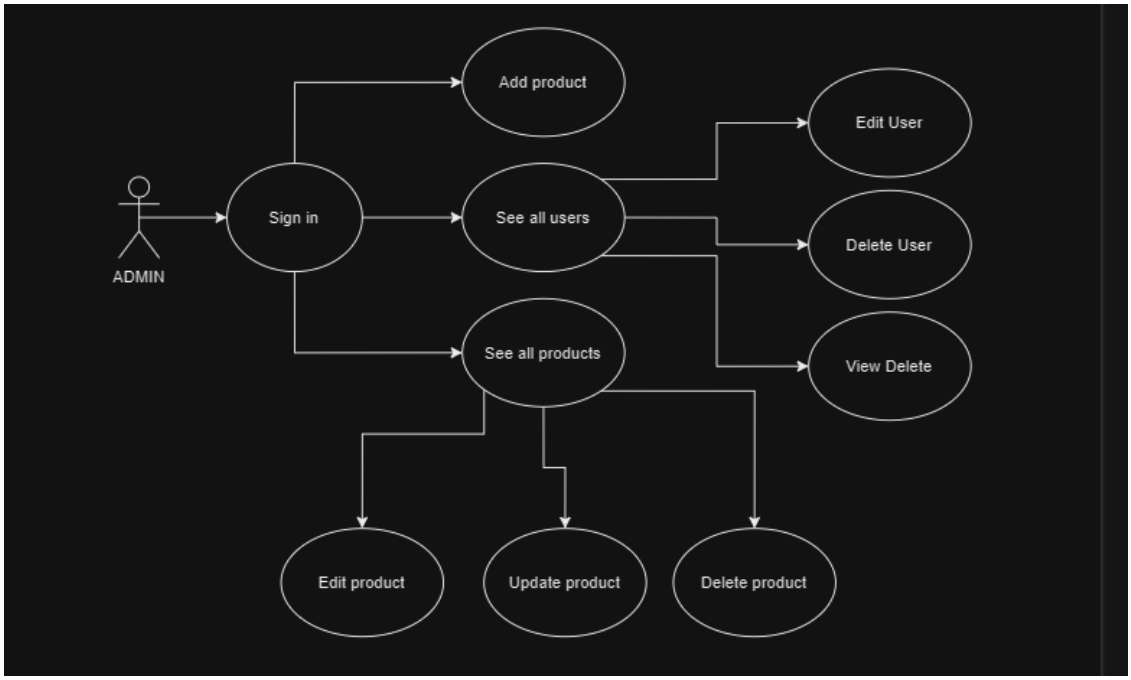
c) Ștergere produs:

- Administratorul selectează opțiunea "Delete Product" pentru a șterge un anumit produs.
- Sistemul solicită confirmarea administratorului.
- După confirmare, produsul este șters din baza de date.

Extensions:

- În cazul în care datele introduse în timpul autentificării nu sunt valide sau nu corespund cu un cont de administrator existent, sistemul afișează un mesaj de eroare și solicită reintroducerea datelor corecte sau oferă posibilitatea recuperării parolei.
- Dacă administratorul nu are drepturile necesare pentru a accesa anumite funcționalități (de exemplu, dacă este un cont de tip "guest"), sistemul îi va afișa un mesaj corespunzător și îi va limita accesul la anumite operații.
- În cazul în care administratorul încearcă să efectueze operații care necesită permisiuni suplimentare (cum ar fi ștergerea unui utilizator sau a unui produs), sistemul poate solicita

confirmarea suplimentară a administratorului pentru a preveni ștergerile accidentale sau neautorizate.



4 Specificații suplimentare

4.1 Cerințe non-funcționale

4.1.1 Performanță

Performanța este esențială pentru un magazin online, deoarece utilizatorii așteaptă o experiență rapidă și fără întârzieri. Astfel, avem nevoie de o cerință non-funcțională care să asigure că timpul de răspuns al aplicației noastre este scăzut, iar paginile se încarcă rapid atunci când sunt accesate de către utilizatori. Acest lucru este crucial pentru a menține angajamentul utilizatorilor și pentru a crește conversiile.

4.1.2 Scalabilitate

Un magazin online trebuie să poată gestiona o creștere bruscă a traficului și a volumului de date, în special în timpul perioadelor de vârf, cum ar fi sărbătorile sau campaniile de reduceri. Prin urmare, avem nevoie de o cerință non-funcțională care să asigure că aplicația noastră este scalabilă și poate fi extinsă fără probleme pentru a gestiona cerințele crescânde ale utilizatorilor. Aceasta poate implica utilizarea serviciilor cloud pentru a crește resursele de calcul și stocare în mod dinamic în funcție de necesități.

4.1.3 Fiabilitatea

Fiabilitatea este esențială pentru un magazin online, deoarece orice indisponibilitate sau cădere a sistemului poate duce la pierderi de vânzări și la scăderea încrederii utilizatorilor. Avem nevoie de cerințe non-funcționale care să asigure că aplicația noastră este stabilă și că poate funcționa fără probleme pe termen lung. Implementarea redundanței în infrastructură

și a mecanismelor de backup poate contribui la asigurarea unei funcționări fără probleme a sistemului.

4.1.4 Uzabilitate

Uzabilitatea este o altă cerință non-funcțională importantă pentru un magazin online, deoarece utilizatorii trebuie să poată naviga ușor în site și să găsească rapid produsele dorite. Avem nevoie de o interfață utilizator intuitivă și prietenoasă, care să ofere o experiență plăcută utilizatorilor și să îi încurajeze să revină pentru achiziții ulterioare. Testele de utilizabilitate și feedback-ul utilizatorilor pot fi folosite pentru a îmbunătăți continuu experiența utilizatorului și pentru a face ajustări în funcție de nevoile și preferințele acestora.

4.2 Design constraints

Există anumite constrângeri de design care au fost mandate și trebuie respectate în implementarea sistemului. Aceste constrângeri includ alegeri legate de limbajul de programare, procesele software, uneltele de dezvoltare, arhitectura și altele.

Pentru partea de backend a aplicației noastre, am ales să utilizăm Java Spring în cadrul mediului de dezvoltare IntelliJ IDEA. Java Spring oferă o platformă robustă și scalabilă pentru dezvoltarea aplicațiilor web, iar IntelliJ IDEA este un mediu de dezvoltare puternic și popular, care oferă unelte avansate pentru dezvoltarea și testarea aplicațiilor Java.

Pentru baza noastră de date, am optat pentru un sistem de gestiune a bazelor de date MySQL, care este un sistem de bază de date relațional fiabil și larg utilizat în industrie. MySQL oferă suport pentru operațiuni complexe de interogare și manipulare a datelor, iar modelul său relațional este potrivit pentru stocarea și gestionarea datelor noastre.

Pentru testarea și validarea funcționalităților backend-ului, folosim Postman, o platformă de testare API-uri care ne permite să testăm și să debugăm endpoint-urile din controllerele noastre într-un mod eficient și intuitiv. Postman oferă funcționalități avansate pentru testarea automatizată, monitorizarea și documentarea API-urilor noastre.

În ceea ce privește partea de frontend a aplicației noastre, am ales să folosim React, o bibliotecă JavaScript modernă și populară pentru dezvoltarea interfețelor de utilizator interactive și dinamice. React oferă un mod eficient de a construi componente reutilizabile și de a gestiona starea aplicației noastre într-un mod clar și predictibil.

Pentru a completa dezvoltarea frontend-ului, am instalat pachetele Axios, Bootstrap și react-router-dom. Axios este o librărie pentru efectuarea de cereri HTTP din JavaScript, Bootstrap este un framework CSS popular pentru stilizarea interfeței de utilizator, iar react-router-dom este o bibliotecă pentru gestionarea rutelor și navigarea în aplicația noastră React.

În ceea ce privește arhitectura aplicației noastre, am ales să urmăm modelul de arhitectură Model-View-Controller (MVC), care ne permite să separăm logic interfața utilizatorului, logica de afaceri și logica de gestionare a datelor în straturi distincte și bine definite. Această abordare ne ajută să creăm aplicații modulare, ușor de întreținut și scalabile.