Data	Versiune	Autor	Comentarii

Cerinţele de sistem pentru E-commerce

1 Introducere

Această documentație descrie modul în care poate fi creat un sistem de e-commerce utilizând MySQL pentru gestionarea datelor, Java cu Spring Boot pentru logica de business și API-ul back-end, și Angular pentru interfața de utilizator (front-end). Arhitectura sistemului este bazată pe o abordare "client-server", în care partea front-end interacționează cu un API REST furnizat de Spring Boot, care la rândul său comunică cu baza de date MySQL.
Pentru gestionarea datelor, baza de date MySQL va conține tabele pentru stocarea informațiilor despre produse, utilizatori, comenzi și categorii. Tabelele sunt interconectate prin relații, ceea ce permite stocarea eficientă a datelor într-un format relațional. Tabelul pentru utilizatori gestionează informațiile de autentificare și profil, cel pentru produse păstrează detalii despre fiecare produs disponibil în magazin, iar tabelele pentru comenzi și detalii despre comenzi gestionează informațiile necesare pentru a urmări comenzile plasate de utilizatori și articolele din fiecare comandă. Categoriile sunt utilizate pentru a grupa produsele în funcție de tip sau categorie, facilitând organizarea acestora pe site-ul de e-commerce.

În ceea ce privește partea de back-end, Spring Boot va fi utilizat pentru a crea un API REST care expune diferite servicii pentru interacțiunea cu datele. Acest API va gestiona operațiuni precum înregistrarea și autentificarea utilizatorilor, vizualizarea produselor, plasarea comenzilor și gestionarea coșului de cumpărături. Cu ajutorul Spring Data JPA, operațiunile cu baza de date sunt simplificate, oferind metode predefinite pentru interacțiunea cu entitățile din baza de date. Hibernate, un ORM integrat în Spring Boot, este responsabil pentru maparea obiectelor Java la tabelele din baza de date. Pe lângă acestea, Spring Security va fi utilizat pentru gestionarea autentificării și autorizării, asigurând protecția datelor utilizatorilor și securitatea tranzacțiilor.

Pe partea de front-end, Angular va fi folosit pentru a construi o interfață dinamică și interactivă pentru utilizatori. Aplicația Angular va comunica cu API-ul back-end folosind cereri HTTP pentru a prelua date despre produse, pentru a gestiona coșul de cumpărături și pentru a finaliza comenzi. Componenta front-end va include pagini pentru vizualizarea produselor, detalii despre produse, gestionarea coșului de cumpărături și pagini de profil pentru utilizatori. Comunicarea între front-end și back-end va fi realizată prin API-uri REST, ceea ce înseamnă că datele sunt transmise într-un format JSON între Angular și Spring Boot.

Această abordare modulară permite scalabilitatea și întreținerea ușoară a aplicației. Baza de date MySQL stochează în siguranță datele esențiale, în timp ce API-ul Spring Boot gestionează logica de afaceri și interacțiunea cu datele, oferind în același timp o securitate robustă. Angular se ocupă de prezentarea datelor și interacțiunea cu utilizatorii, oferind o experiență modernă și fluidă pentru clienți. Această arhitectură, bazată pe standarde moderne, asigură atât eficiență în dezvoltare, cât și o experiență performantă pentru utilizatorii finali.

2 Prezentarea generală a actorilor si a diagramelor use-case

Actorii principali care interacționează cu sistemul sunt:

Actor	Tip	Descriere
Administrator	Uman	Persoana responsabilă pentru gestionarea sistemului e-commerce.
Client	Uman	Persoana care cumpără produse prin intermediul platformei.
Vizitator	Uman	Persoana care explorează site-ul fără a avea un cont.

3 Descrierea detaliată a actorilor si a diagramelor use-case

3.1 Actorii

3.1.1 Administrator

Tip: Uman

Descriere detaliată: Administratorul va avea rolul de a configura și menține platforma. Acesta va gestiona utilizatorii, va adăuga și actualiza produsele disponibile pe site și va analiza statisticile vânzărilor. Administratorul va avea acces la funcții avansate și va necesita autentificare.

Utilizări:

- Gestionarea conturilor de utilizator (creare, modificare, ștergere).
- Actualizarea stocului de produse.
- Generarea rapoartelor de vânzări și analiza performanțelor.

3.1.2 Client

Tip: Uman

Descriere detaliată: Clientul va putea naviga pe site, va avea un cont în care își va gestiona informațiile personale, iar după autentificare va putea comanda produse. De asemenea, clientul va putea urmări starea comenzilor și va avea opțiunea de a evalua produsele achiziționate.
Utilizări:

- Înregistrarea unui cont nou.
- Plasarea comenzilor și gestionarea coșului de cumpărături.
- Vizualizarea istoricului comenzilor și a statusului livrării.

3.1.3 Vizitator

Tip : Uman

Descriere detaliată: Vizitatorul va putea accesa platforma și va putea vizualiza produsele, dar nu va putea efectua achiziții până nu se va înregistra. Acesta va putea să se aboneze la newsletter pentru a primi informații despre oferte și promoții.

Utilizări:

- Navigarea prin paginile de produse.
- Abonarea la newsletter.
- Crearea unui cont pentru a deveni client.

3.2 Diagrame use-case

3.2.1 Diagrama use-case a administratorului

Descriere: Administratorul se va autentifica în sistem pentru a accesa funcțiile de gestionare a utilizatorilor și produselor. Va putea adăuga noi produse, modifica informațiile existente și va analiza datele de vânzare.

Pre-condiții și post-condiții:

- Administratorul trebuie să se autentifice.
- La final, acesta se va deautentifica.

3.2.2 Diagrama use-case a userului

Descriere: Clientul va crea un cont, se va autentifica și va putea plasa comenzi. După achiziție, va putea evalua produsele și va avea acces la istoricul comenzilor.

Pre-condiții și post-condiții:

- Clientul trebuie să fie autentificat pentru a plasa comenzi.
- La final, clientul se va deautentifica.

3.2.3 Diagrama use-case a vizitatorului

Descriere: Vizitatorul va putea naviga pe site, va putea vizualiza produsele și va avea opțiunea de a se înregistra pentru a deveni client.

Pre-condiții și post-condiții:

- Vizitatorul nu poate face achiziții fără a se înregistra.
- După înregistrare, acesta va avea acces la funcțiile clientului**.**

3.3 Lista de Functionalitati

4.1 Funcționalități de Administrare

- Gestionare utilizatori: Creare, modificare și ștergere conturi de utilizator.
- Gestionare produse: Adăugare, actualizare și ștergere produse din catalog.
- Analiză vânzări: Rapoarte privind performanța vânzărilor și trendurile de achiziție.

4.2 Funcționalități pentru Clienți

- Înregistrare și autentificare: Proces simplu de înregistrare și opțiuni de autentificare.
- Coș de cumpărături: Adăugarea, modificarea și eliminarea produselor din coș.
- Comandă și livrare: Plasarea comenzilor și opțiuni de livrare.

4.3 Funcționalități pentru Vizitatori

- Navigare: Accesibilitate la toate produsele fără autentificare.
- Newsletter: Abonare pentru a primi oferte și actualizări despre produse.
- Înregistrare: Opțiunea de a deveni client printr-un formular de înregistrare.

4 Analiza&Design:

4.1 Prezentare Generală

Proiectul propus este un sistem de e-commerce inspirat de platforma eMAG, dezvoltat folosind tehnologiile **Spring Boot**, **Angular**, și **MySQL**. Scopul principal este de a crea o platformă modernă, rapidă și sigură, care să ofere utilizatorilor posibilitatea de a naviga, căuta și achiziționa produse într-un mod simplu și eficient. Sistemul este proiectat pentru a fi scalabil și flexibil, răspunzând cerințelor complexe ale unui magazin online contemporan.

4.2 Fundamente Teoretice

Acest proiect se bazează pe arhitectura client-server, separând clar partea de back-end de partea de front-end. Back-end-ul, realizat în **Spring Boot**, implementează logica de afaceri și gestionează interacțiunile cu baza de date, respectând modelul **MVC (Model-View-Controller)**. Această separare între logică și interfață permite o experiență fluidă pentru utilizatori și o gestionare eficientă a datelor. În același timp, baza de date **MySQL** stochează informațiile despre produse, utilizatori, comenzi și stocuri, fiind optimizată pentru manipularea eficientă a datelor.

4.3 Tehnologii Utilizate

Proiectul utilizează următoarele tehnologii esențiale:

- **Spring Boot** pentru partea de back-end, oferind o structură robustă și scalabilă pentru gestionarea logicii de afaceri și a comunicării cu baza de date.
- **Angular** pentru front-end, asigurând o interfață de utilizator dinamică și reactivă, cu funcționalități moderne pentru o experiență intuitivă.
- **MySQL** ca soluție pentru gestionarea bazei de date relaționale, capabilă să stocheze și să manipuleze volume mari de date, esențială pentru un sistem de e-commerce.

4.4 Arhitectura și Designul Sistemului

Sistemul este proiectat pe mai multe niveluri, fiecare având un rol bine definit:

1. **Front-end**: Dezvoltat în Angular, front-end-ul include module pentru gestionarea paginilor de produse, coșul de cumpărături, istoricul comenzilor, profilurile utilizatorilor și căutările avansate. Fluxurile reactive dintre componente asigură o experiență de utilizator rapidă și optimizată.
2. **Back-end**: Realizat în Spring Boot, back-end-ul gestionează logica aplicației, incluzând funcționalități precum gestionarea comenzilor, autentificarea utilizatorilor și actualizarea bazei de date. Controlerele și serviciile prelucrează cererile HTTP și coordonează schimbul de date dintre client și server.
3. **Baza de date**: Structura bazei de date MySQL este proiectată pentru a gestiona tabele complexe pentru produse, comenzi, utilizatori și istoricul tranzacțiilor. Optimizările efectuate permit manipularea eficientă a datelor, chiar și în condiții de utilizare intensivă.

4.5 Modul de Operare

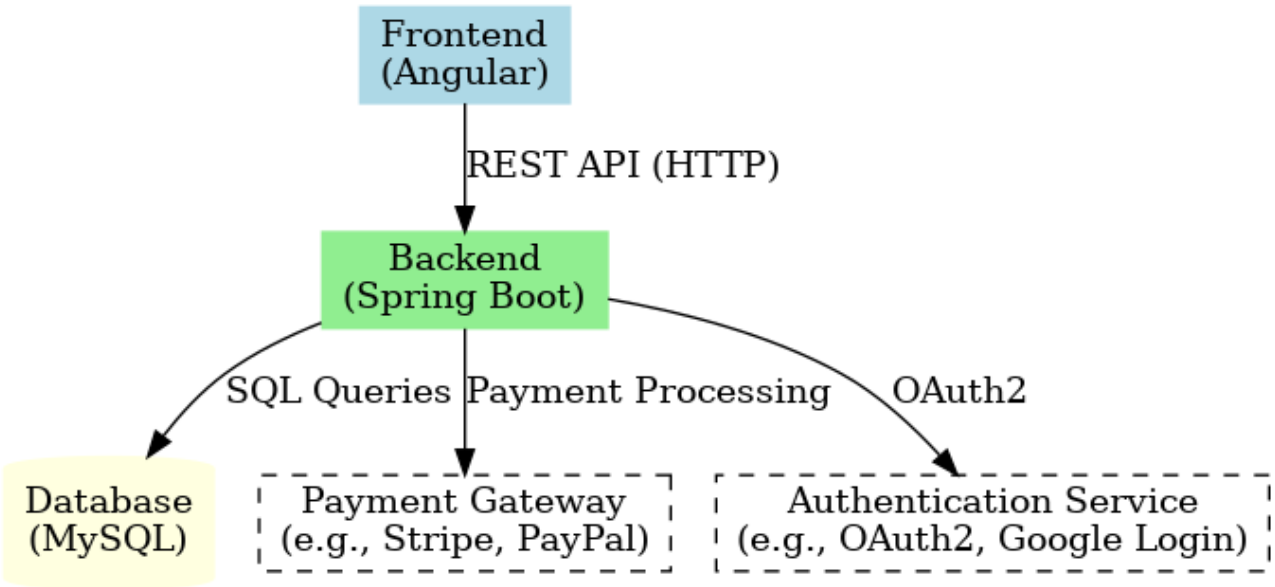
Platforma oferă o interactivitate intuitivă, structurat pe următoarele funcționalități principale:

- **Navigare**: Utilizatorii pot explora categorii diverse de produse și pot aplica filtre avansate pentru a găsi rapid ceea ce caută.
- **Autentificare și personalizare**: Utilizatorii au opțiunea de a se autentifica și de a-și gestiona profilurile, având acces la istoricul comenzilor și preferințele personale.
- **Coș de cumpărături**: Produsele pot fi adăugate, modificate sau eliminate din coș înainte de finalizarea comenzii.
- **Checkout și plată**: Sistemul oferă un proces de plată securizat, permițând finalizarea comenzilor într-un mod rapid și sigur.

5 Diagrame utilizate

5.1 System Architecture Diagram

- Diagrama oferă o vedere de ansamblu clară asupra arhitecturii sistemului si este o reprezentare vizuală a structurii unui sistem software, evidențiind principalele componente, relațiile dintre ele și fluxurile de date



5.2 Flowchart

- Un **flowchart** reprezintă grafic pașii unui proces sau algoritm, folosind simboluri pentru a ilustra decizii, acțiuni și fluxuri de date. Este utilizat pentru a vizualiza logic și simplu procesele complexe, facilitând înțelegerea și optimizarea acestora.

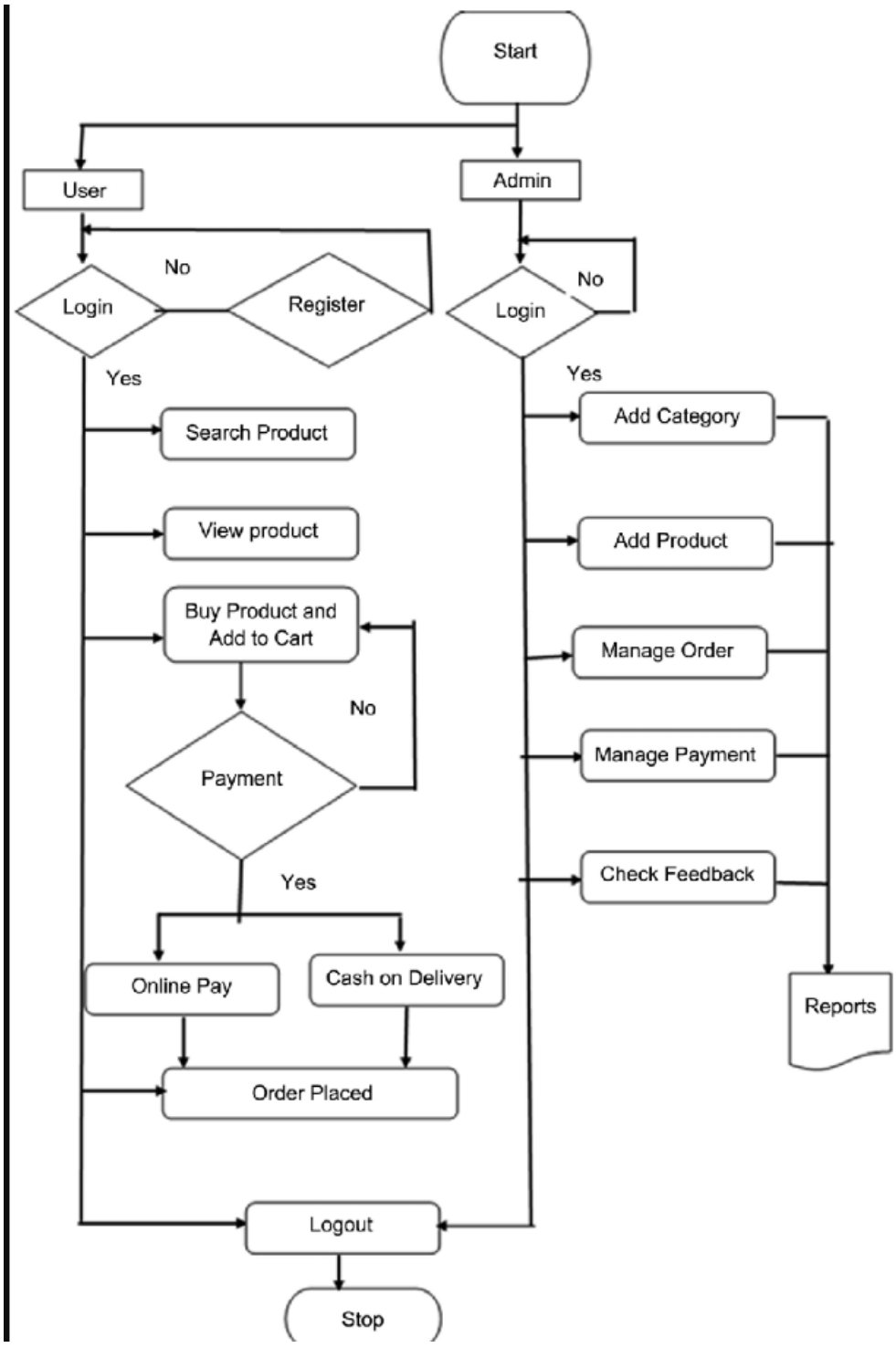


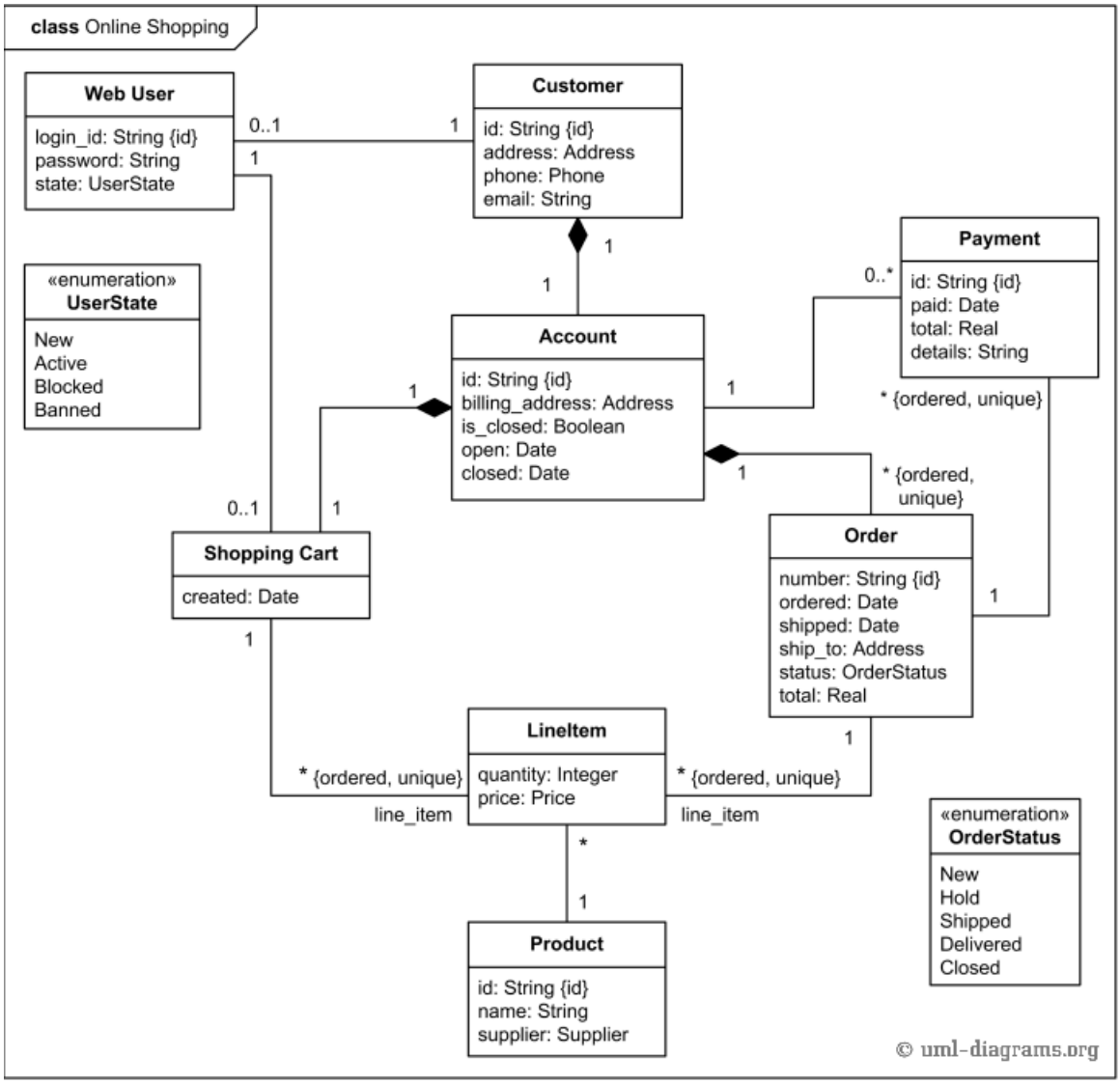
Diagrama de flux reprezintă un proces de e-commerce care implică două tipuri de utilizatori, Utilizator și Admin, fiecare având funcționalități specifice. Diagrama începe cu un nod de Start, care se ramifică în două căi: una pentru Utilizator și cealaltă pentru Admin. Fiecare cale definește un set de acțiuni specifice în funcție de rolul utilizatorului.

Pentru Utilizator, fluxul începe cu opțiunea de autentificare sau înregistrare. Dacă utilizatorul nu are un cont, este necesar să se înregistreze pentru a putea continua. După înregistrare sau dacă este deja înregistrat, utilizatorul trebuie să se autentifice pentru a accesa platforma. În cazul în care nu se autentifică, procesul se oprește. După autentificare, utilizatorul poate căuta produse pe platformă și poate vizualiza detaliile acestora. Dacă utilizatorul dorește să achiziționeze

un produs, îl poate adăuga în coșul de cumpărături și decide dacă finalizează comanda prin plată. În caz contrar, procesul se oprește aici. Dacă decide să finalizeze comanda, utilizatorul are două opțiuni de plată: online sau ramburs (Cash on Delivery). După ce plata este realizată, comanda este considerată plasată, iar utilizatorul poate alege să se delogheze din cont, ceea ce marchează sfârșitul procesului.

Pentru Admin, fluxul de lucru începe tot cu autentificarea. Dacă adminul nu se autentifică, procesul se oprește. Odată autentificat, adminul are acces la o serie de funcționalități administrative, printre care adăugarea de categorii și produse noi, gestionarea comenzilor și plăților, precum și verificarea feedback-ului primit de la utilizatori. În plus, adminul poate accesa rapoarte referitoare la activitatea de pe platformă. Aceste funcționalități permit adminului să gestioneze și să optimizeze platforma, asigurându-se că aceasta funcționează corect pentru utilizatori și că toate comenzile sunt procesate corespunzător.

5.3 Diagrama Clasa:



Această diagramă reprezintă un **Model UML de Clasă** pentru un sistem de cumpărături online, evidențiind principalele entități ale sistemului, atributele acestora și relațiile dintre ele. Este un model conceptual menit să descrie structura logică a sistemului de e-commerce.

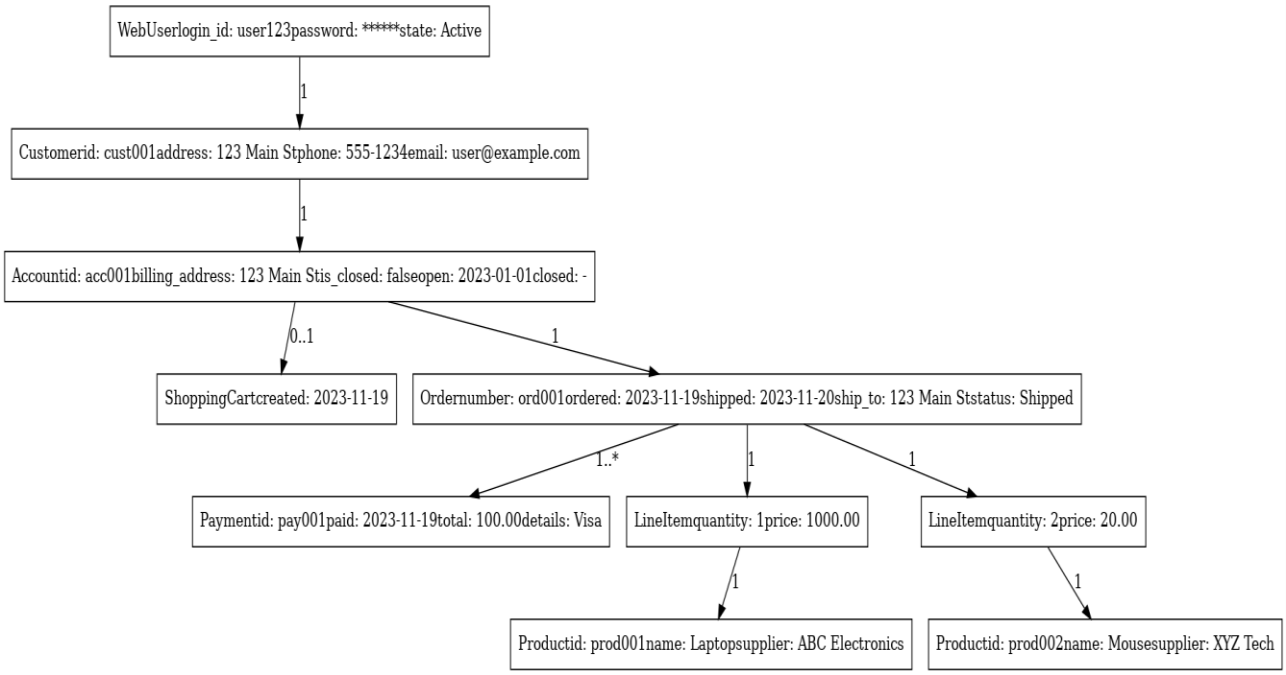
Entitatea **Web User** definește utilizatorii care accesează platforma, având atribute precum ID-ul de autentificare, parola și starea contului (ex: activ, blocat). Acești utilizatori pot fi asociați cu entitatea **Customer**, care include informații personale, precum adresa, telefonul și email-ul. Acest model asigură o separare clară între funcțiile de autentificare și gestionarea datelor clientului.

Entitatea **Account** este legată direct de un client și gestionează detalii legate de facturare, precum adresa de facturare, starea contului (deschis sau închis) și datele asociate activării acestuia. Fiecare cont poate fi asociat cu unul sau mai multe **Orders** (comenzi) și poate avea un coș de cumpărături activ, reprezentat prin entitatea **Shopping Cart**, care permite utilizatorului să adauge produse înainte de a finaliza achiziția.

Comenzile, reprezentate de entitatea **Order**, conțin detalii precum numărul unic al comenzii, datele plasării și livrării acesteia, precum și adresa de expediere. Starea comenzilor este gestionată printr-o enumerare (OrderStatus), care poate fi, de exemplu, New, Shipped sau Delivered. Fiecare comandă este legată de unul sau mai multe produse prin intermediul entității **LineItem**, care stochează informații despre cantitatea și prețul fiecărui produs din comandă.

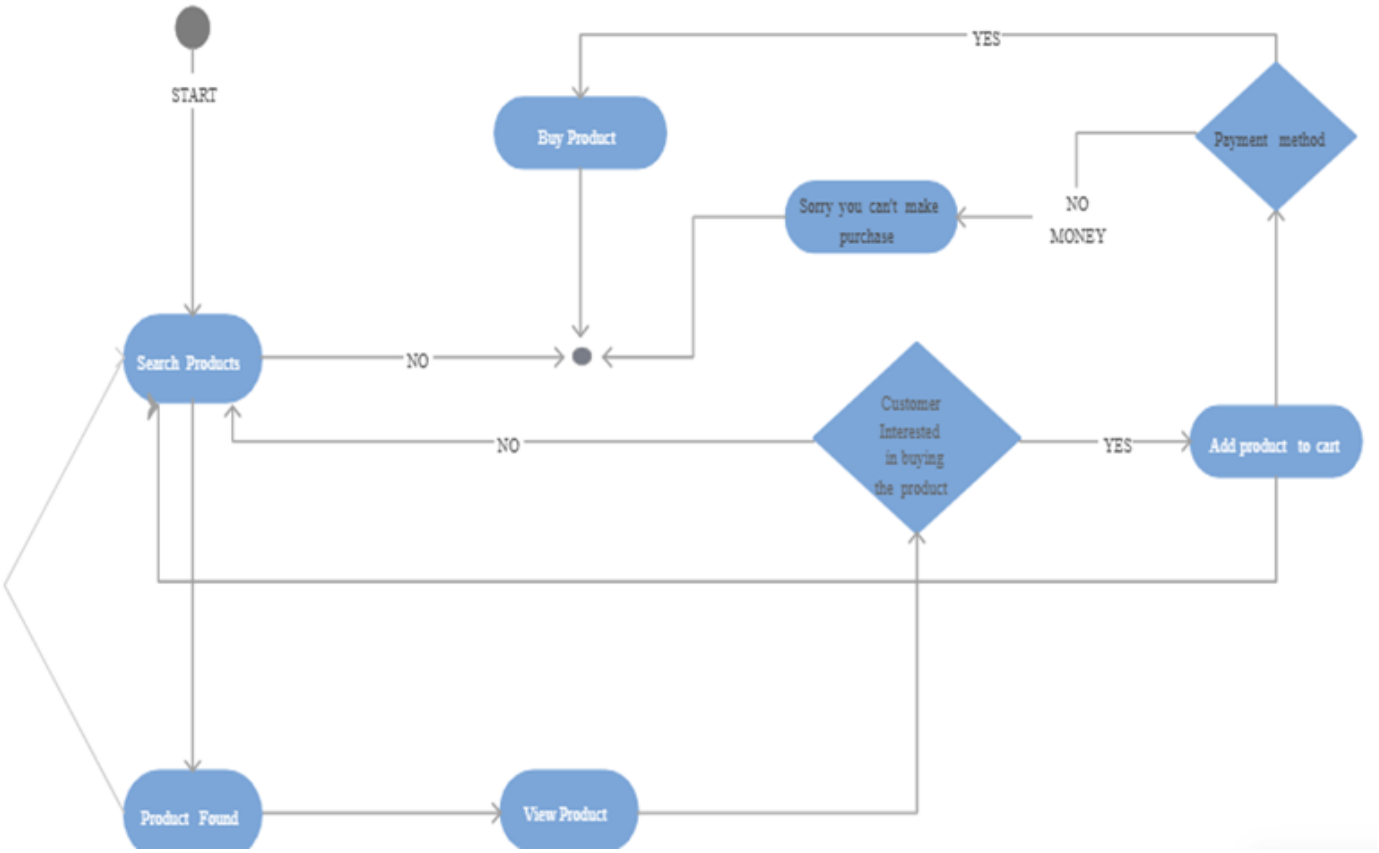
Plățile sunt gestionate de entitatea **Payment**, care include detalii despre sumele plătite, metoda utilizată și data tranzacției. O comandă poate avea una sau mai multe plăți asociate, în funcție de metodele disponibile. Produsele disponibile în sistem sunt descrise de entitatea **Product**, care include numele, ID-ul unic și furnizorul fiecărui produs.

5.4 Object Diagram



Această diagramă de obiect reprezintă instanțele specifice ale claselor definite în diagrama de clasă pentru un sistem de cumpărături online. Este utilizată pentru a oferi o imagine concretă a interacțiunilor dintre obiectele din sistem, ilustrând cum sunt conectate și cum colaborează pentru a realiza funcționalitățile principale.

5.5 Activity diagram

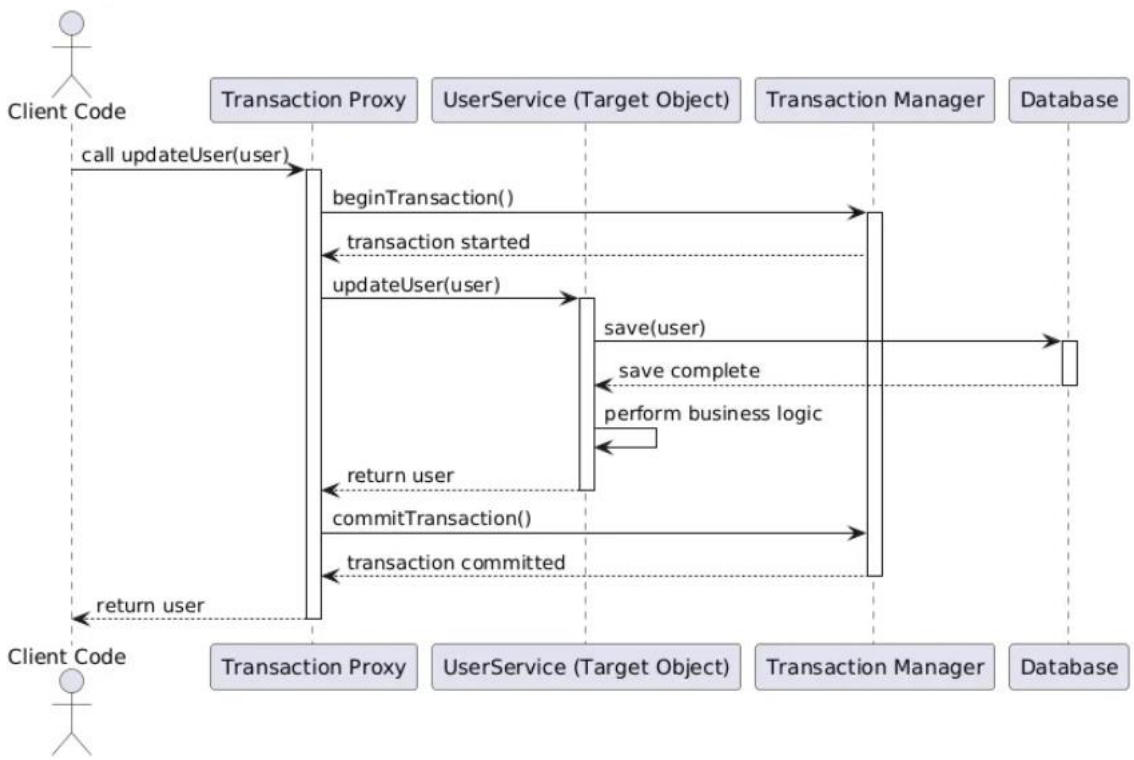


Această diagramă reprezintă un **Activity Diagram** pentru procesul de achiziție al unui produs într-un sistem de e-commerce. Diagrama începe cu utilizatorul care caută produse (Search Products). Dacă un produs este găsit, utilizatorul îl poate vizualiza (View Product) și decide dacă este interesat să îl cumpere (Customer Interested in buying the product).

Dacă utilizatorul decide să achiziționeze produsul, acesta este adăugat în coș (Add product to cart), iar utilizatorul selectează metoda de plată (Payment method). În cazul în care nu există suficiente fonduri (NO MONEY), utilizatorului i se afișează un mesaj de eroare (Sorry you can't make purchase). Dacă plata este validă, procesul continuă și achiziția este finalizată.

Diagrama descrie în mod clar pașii logici și deciziile din procesul de cumpărare, evidențiind posibilele ramificații și rezultate.

5.6 State transition diagram



Acesta este un diagramă de secvență pentru un proces de actualizare a unui utilizator într-un sistem de tranzacții, utilizând un mecanism de tip proxy de tranzacție. Diagrama evidențiază fluxul de mesaje între diferitele componente implicate, cum ar fi **Client Code**, **Transaction Proxy**, **UserService (Target Object)**, **Transaction Manager**, și **Database**.

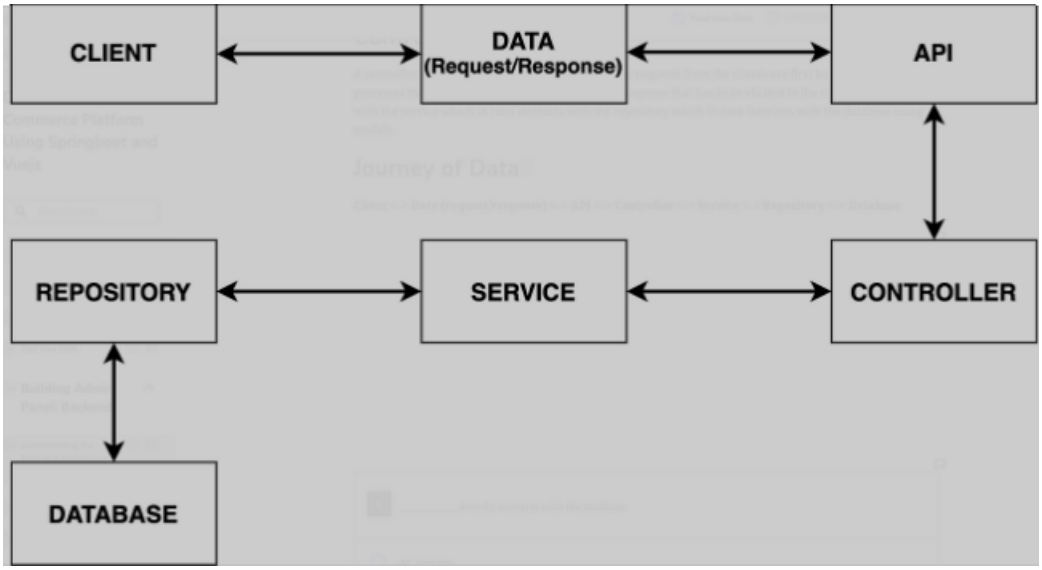
Procesul începe cu **Client Code**, care inițiază o solicitare de actualizare a unui utilizator prin apelul updateUser(user). Această solicitare este trimisă mai întâi către **Transaction Proxy**, o componentă intermediară care gestionează tranzacțiile. Înainte de a continua cu operația de actualizare, **Transaction Proxy** inițiază o tranzacție, apelând beginTransaction() din cadrul **Transaction Manager**. După ce tranzacția este inițiată, **Transaction Proxy** primește confirmarea că tranzacția a început.

Odată ce tranzacția a început, **Transaction Proxy** trimite apelul updateUser(user) către **UserService (Target Object)**, care este componenta responsabilă pentru manipularea logicii de afaceri specifice utilizatorului. **UserService** apelează metoda save(user) pentru a salva detaliile actualizate ale utilizatorului în **Database**. După ce operația de salvare este completă, baza de date returnează o confirmare către **UserService**, iar aceasta finalizează logica de afaceri specifică, dacă este necesar.

După ce operația de actualizare a utilizatorului este completă, **UserService** returnează obiectul utilizatorului actualizat către **Transaction Proxy**. **Transaction Proxy** finalizează procesul de tranzacție apelând commitTransaction() la **Transaction Manager**. În acest moment, tranzacția este comisă, iar schimbările sunt salvate definitiv în baza de date. **Transaction Proxy** primește confirmarea că tranzacția a fost comisă și returnează obiectul utilizatorului actualizat înapoi către **Client Code**.

În final, **Client Code** primește obiectul utilizatorului actualizat, indicând că procesul de actualizare a fost finalizat cu succes. Diagrama de secvență ilustrează clar cum funcționează un sistem de tranzacționare prin intermediul unui proxy de tranzacție, asigurându-se că toate operațiunile critice sunt însoțite de un proces de tranzacție controlat, începând cu inițierea și terminând cu comiterea modificărilor în baza de date. Acest mecanism asigură consistența și integritatea datelor în timpul operațiunilor de actualizare.

5.7 Communication Diagram

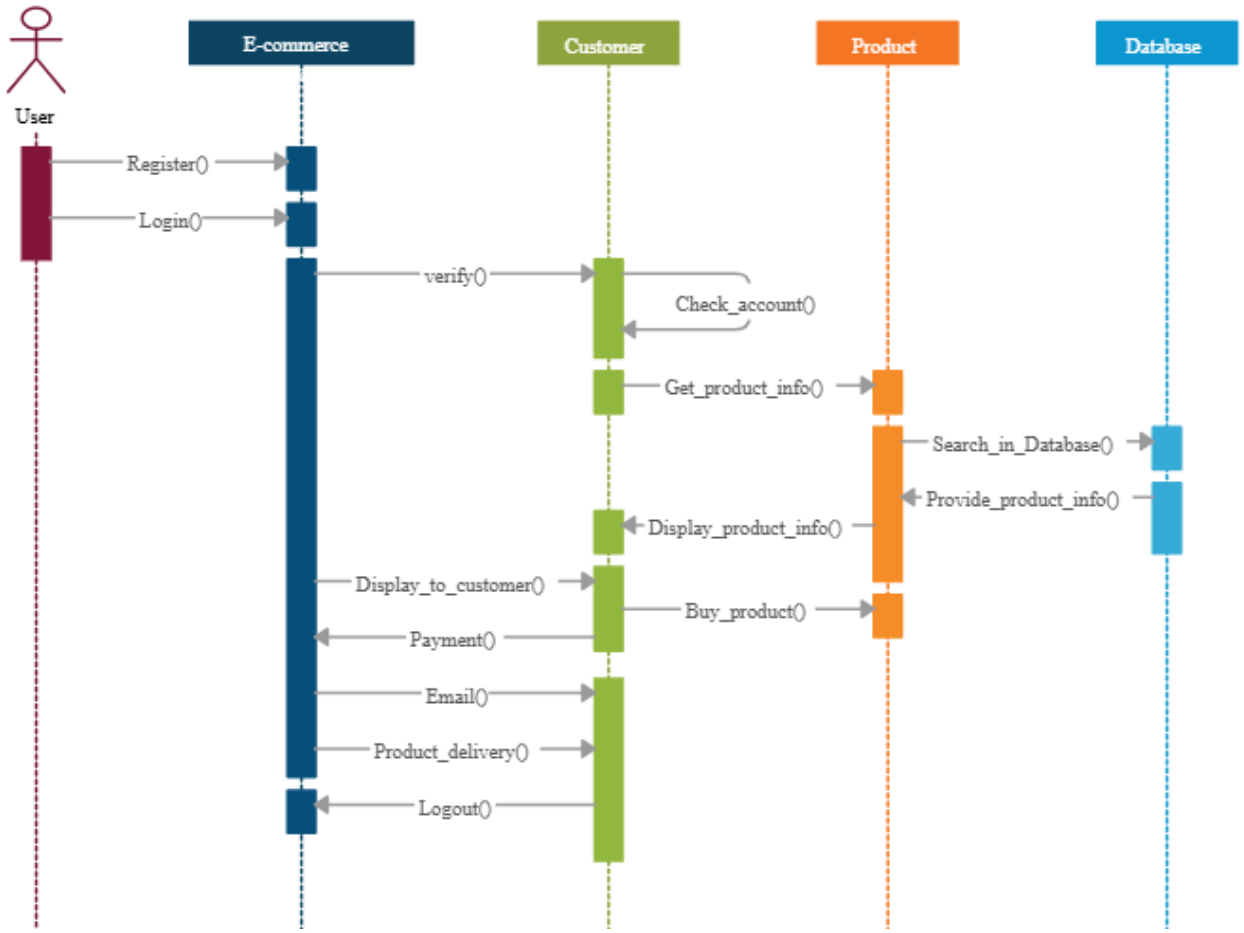


Începând cu utilizatorul (Client), cererile HTTP (Request) sunt trimise către API-ul sistemului, care gestionează interacțiunile printr-un **Controller**. Controller-ul este responsabil pentru primirea cererilor și trimiterea răspunsurilor (Response) înapoi către client.

Controller-ul transmite cererea către componenta **Service**, care conține logica de afaceri a aplicației. Service-ul este responsabil pentru prelucrarea datelor și apelarea următoarei componente, **Repository**, pentru interacțiunea cu baza de date.

Repository-ul gestionează operațiile directe asupra bazei de date (Database), cum ar fi citirea, scrierea sau actualizarea informațiilor. Acesta trimite datele înapoi către Service, care le prelucrează și le trimite înapoi către Controller. Controller-ul, în final, returnează răspunsul către client.

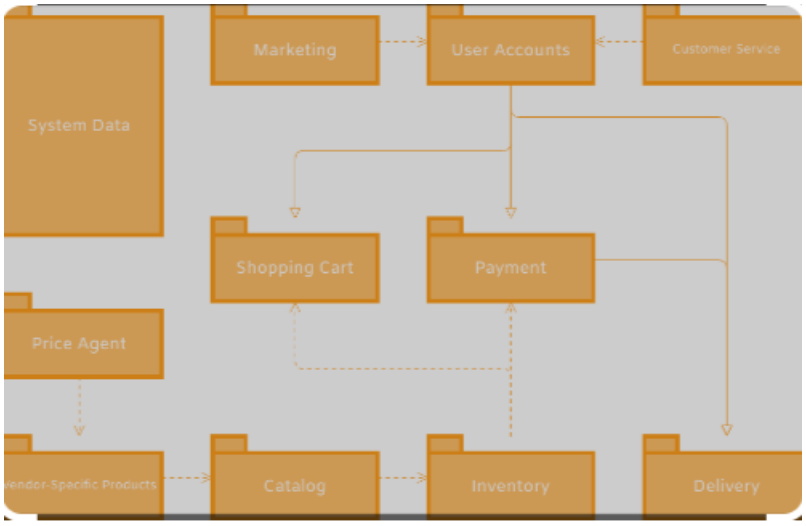
5.8 Sequence Diagram



Această **Sequence Diagram** descrie interacțiunile dintre utilizator și componentele unui sistem de e-commerce. Utilizatorul începe prin a se autentifica (Login()) sau a se înregistra (Register()), iar aceste acțiuni sunt gestionate de componenta **E-commerce**, care coordonează procesul. Sistemul verifică datele clientului prin componenta **Customer** și baza de date (Check_account()).

Când utilizatorul caută un produs, componenta **Product** interoghează baza de date (Search_in_Database()), iar informațiile sunt afișate utilizatorului (Display_product_info()). Dacă utilizatorul decide să cumpere un produs (Buy_product()), sistemul procesează plata (Payment()), trimite confirmări prin email (Email()) și inițiază livrarea produsului (Product_delivery()). În final, utilizatorul se deconectează (Logout()).

5.9 Diagrama de pachete



Aceasta este o **Package Diagram** (diagrama de pachete) care ilustrează organizarea și relațiile dintre modulele principale ale unui sistem de e-commerce. Fiecare pachet reprezintă o componentă logică sau funcțională a sistemului.

System Data este nucleul sistemului, gestionând datele esențiale care sunt accesate de alte module.

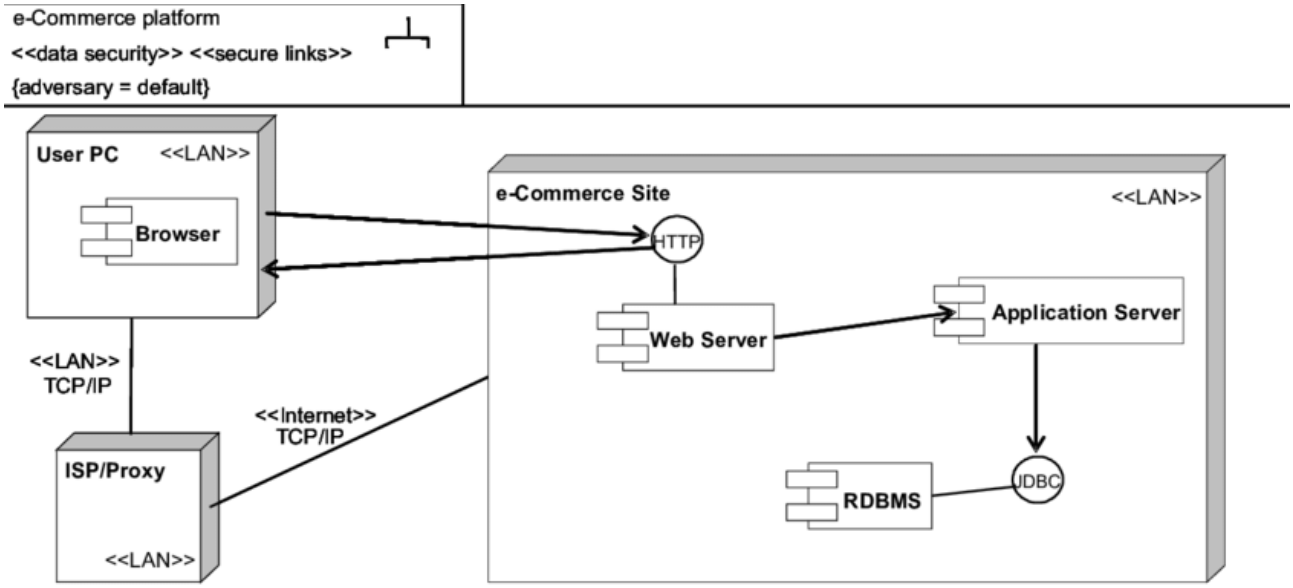
User Accounts și **Marketing** gestionează funcționalitățile utilizatorilor și campaniile de promovare.

Shopping Cart și **Payment** se ocupă de procesele de cumpărare, cum ar fi adăugarea produselor în coș și procesarea plăților.

Catalog, **Inventory**, și **Delivery** sunt modulele responsabile de stocarea informațiilor despre produse, gestionarea stocurilor și livrarea comenzilor.

Price Agent oferă suport pentru actualizarea prețurilor și comparații, interacționând cu alte pachete precum Catalog sau Marketing.

5.10 Deployment Diagram



Aceasta este o **Deployment Diagram** (diagrama de desfășurare) care descrie distribuția componentelor unui sistem de e-commerce pe diferite noduri hardware și modul în care acestea comunică între ele. Diagrama ilustrează infrastructura fizică a sistemului și conexiunile dintre componentele software și hardware.

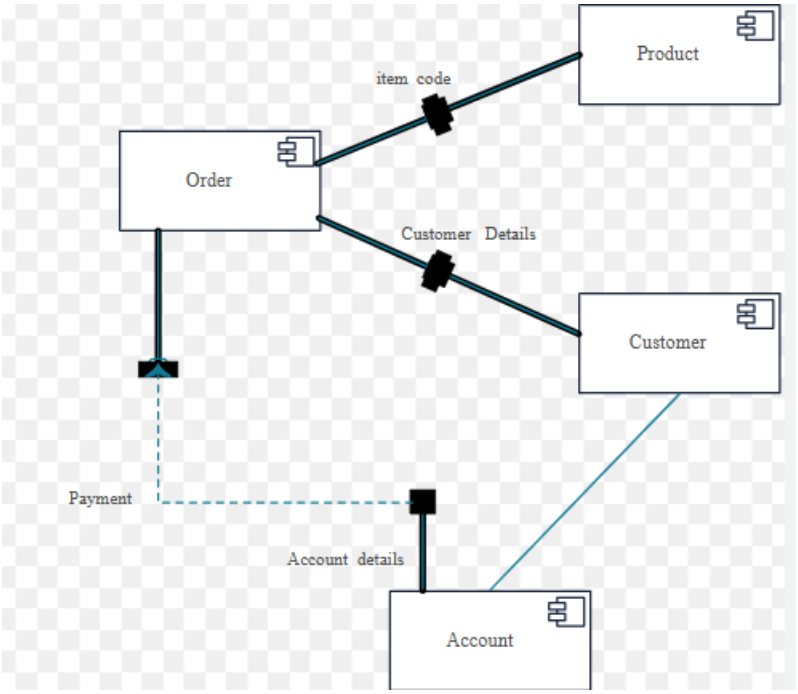
Utilizatorul interacționează cu sistemul printr-un browser de pe propriul calculator (**User PC**). Browser-ul trimite cereri HTTP către serverul web al platformei e-commerce, iar conexiunea se face printr-un **ISP/Proxy**, utilizând protocolul **TCP/IP** pentru acces la internet. Această parte evidențiază modul în care utilizatorii accesează platforma de pe rețele locale sau publice.

Sistemul e-commerce este găzduit pe o infrastructură de servere care include:

- **Web Server**: Gestionează cererile HTTP primite de la utilizator și le transmite către serverul de aplicații.
- **Application Server**: Conține logica principală a aplicației, procesează cererile și coordonează interacțiunea cu baza de date.
- **RDBMS** (Relational Database Management System): Stochează datele utilizatorilor, comenzile și informațiile despre produse, fiind conectat la serverul de aplicații prin intermediul **JDBC** (Java Database Connectivity).

Conexiunile dintre componente sunt securizate, așa cum este indicat de elementele de securitate incluse (<<data security>> și <<secure links>>). Protocoalele utilizate, precum **HTTP** și **TCP/IP**, sunt specificate pentru a evidenția fluxul de comunicație între componente.

5.11 Component Diagram



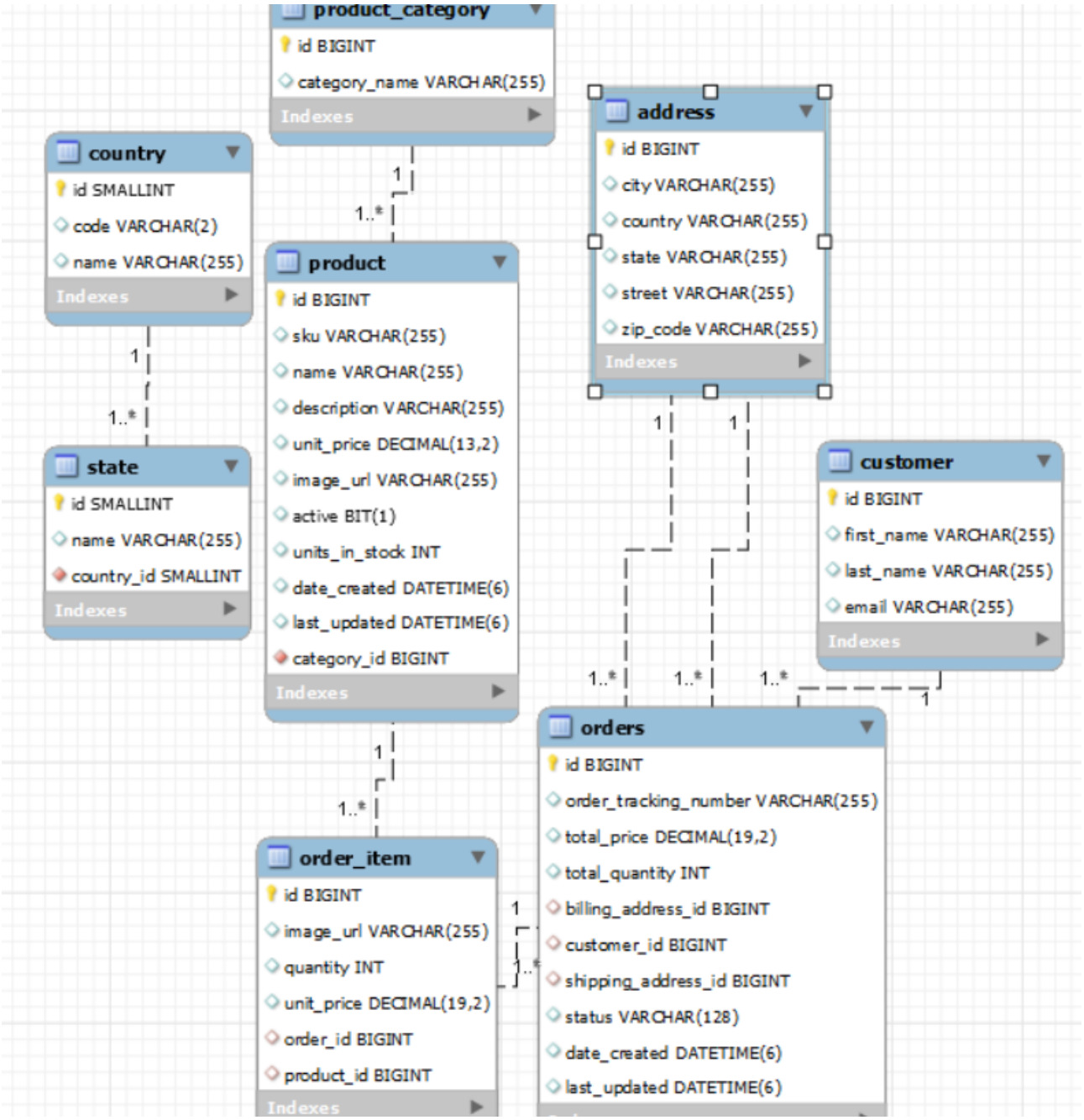
Această diagramă de clasă evidențiază structura logică a unui sistem de e-commerce, concentrându-se pe relațiile dintre principalele entități: comenzi, produse, clienți și conturi. Este esențială pentru înțelegerea modului în care datele sunt interconectate în sistem și pentru a sprijini proiectarea funcționalităților.

Clasa centrală este **Order** (Comandă), care gestionează informațiile despre comenzile plasate de clienți. Aceasta are relații directe cu clasa **Product**, utilizând un item code unic pentru a identifica produsele incluse într-o comandă. De asemenea, clasa **Order** este conectată la clasa **Customer**, asigurând legătura dintre comenzi și clienți, prin intermediul detaliilor acestora (Customer Details). În plus, clasa **Order** interacționează cu clasa **Account** pentru a gestiona informațiile de plată și facturare.

Clasa **Product** (Produs) reprezintă articolele disponibile pentru vânzare și este legată de comenzi prin coduri unice. Aceasta permite identificarea clară a produselor incluse în fiecare comandă. În același timp, clasa **Customer** (Client) conține informațiile despre clienți și este conectată atât la comenzi, cât și la conturi, asigurând gestionarea datelor personale și a relației cu comenzile.

Clasa **Account** (Cont) joacă un rol esențial în administrarea informațiilor financiare. Aceasta gestionează plățile (Payment) și legătura cu detaliile financiare ale clienților, fiind conectată la comenzi pentru a facilita procesarea tranzacțiilor.

5.12 UML Diagram



Această diagramă ER (Entity-Relationship) reprezintă structura logică a unei baze de date pentru un proiect de e-commerce. Este utilizată pentru a defini entitățile principale ale sistemului, atributele acestora și relațiile dintre ele, oferind o bază solidă pentru proiectarea și implementarea bazei de date.

Entitatea **Customer** (Client) reprezintă utilizatorii platformei și stochează informații precum Customer_ID (cheia primară), Name, Email, Address și Phone. Fiecare client poate plasa mai multe comenzi, stabilind o relație între entitățile Customer și Order. De asemenea, clienții pot efectua plăți, ceea ce creează o legătură între Customer și Payment.

Entitatea **Order** (Comandă) conține detalii despre comenzile plasate de clienți, incluzând atribute precum Order_ID, Order_date, TotalAmount și Status. Comenzile sunt asociate clienților prin Customer_ID și pot include mai multe produse, gestionate prin intermediul entității **OrderItem**. Aceasta din urmă detaliază articolele din comandă, incluzând cantitatea (Quantity) și subtotalul (Subtotal) pentru fiecare produs.

Entitatea **Product** (Produs) stochează informații despre articolele disponibile, incluzând atribute precum Product_ID, Name, Price, Description și StockQuantity. Produsele sunt asociate comenzilor prin entitatea intermediară OrderItem, care le conectează și permite gestionarea detaliată a articolelor din fiecare comandă.

Entitatea **Payment** (Plată) gestionează informațiile despre plățile efectuate, având atribute precum PaymentID, Amount, PaymentDate și Status. Plățile sunt asociate clienților prin relația Customer-Payment, legând informațiile financiare de clienții care efectuează achizițiile.

Explicația codului backend (Spring)

I. Entity layer:

Această clasă, denumită Product, reprezintă o entitate JPA care este mapată la un tabel din baza de date denumit product. Aceasta conține informații despre un produs, cum ar fi categoria, SKU, numele, descrierea, prețul, imaginea, și dacă este activ.

```
@Entity
@Table(name ="product")
@Data
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @ManyToOne
    @JoinColumn(name = "category_id", nullable = false)
    private ProductCategory category;

    @Column(name = "sku")
    private String sku;

    @Column(name = "name")
    private String name;

    @Column(name = "description")
    private String description;

    @Column(name = "unit_price")
    private BigDecimal unitPrice;

    @Column(name = "image_url")
    private String imageUrl;

    @Column(name = "active")
    private boolean active;
```

II. Repository layer:

ProductRepository este un **DAO (Data Access Object)** care gestionează operațiunile de acces la date pentru entitatea Product. Această interfață extinde JpaRepository, ceea ce îi permite să utilizeze metode predefinite și personalizate pentru operațiuni CRUD (Create, Read, Update, Delete) și pentru căutări mai avansate în baza de date.

Anotarea @RepositoryRestResource:

Această anotare este specifică Spring Data REST. Ea face ca acest repository să fie expus automat ca un **RESTful endpoint**. Transformă metodele repository-ului în puncte de acces API (e.g., /products).

De exemplu:

- Metoda findByCategoryId devine accesibilă printr-un endpoint REST (e.g., /products/search/findByCategoryId).

```
14  @RepositoryRestResource
15  public interface ProductRepository extends JpaRepository<Product,Long> {
16
        no usages
17      Page<Product> findByCategoryId(@RequestParam("id") Long id, Pageable pageable);
18
        no usages
19      Page<Product>findByNameContaining(@RequestParam("name") String name, Pageable pageable);
20  }
21
```

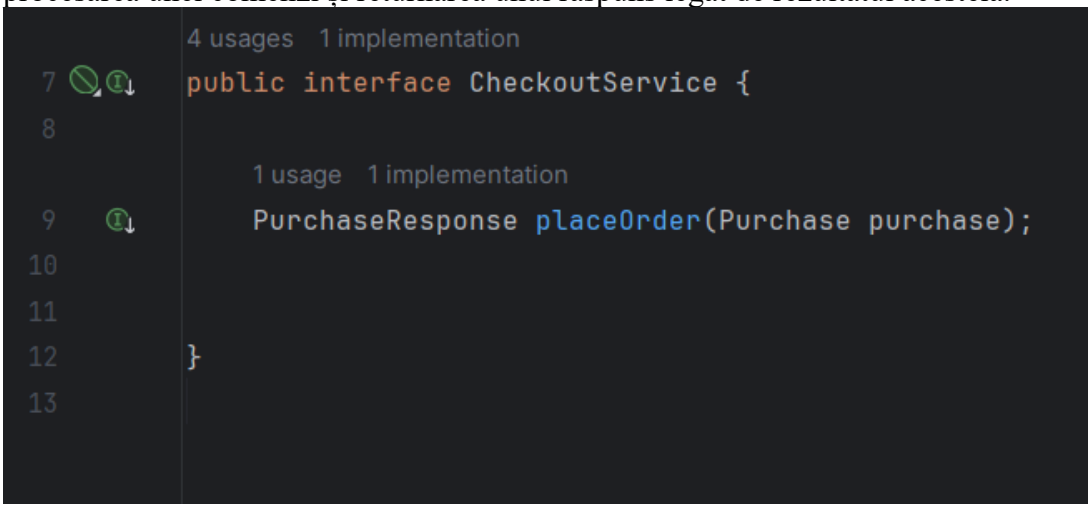
III. Dto layer

Clasa Purchase este un **DTO (Data Transfer Object)** utilizat pentru transferul de date între straturile serviciu și controler mai exact pentru procesul de checkout (plasarea unei comenzi) într-o aplicație eCommerce. Este proiectată pentru a agrega și organiza toate informațiile necesare unei comenzi într-un singur obiect, simplificând astfel transferul datelor.

```
11  @Data
12  public class Purchase {
13      private Customer customer;
14      private Address shippingAddress;
15      private Address billingAddress;
16      private Order order;
17      private Set<OrderItem> orderItems;
18
19
20  }
21
```

IV. Service Interface layer

CheckoutService este o interfață care definește contractul pentru operațiunile de checkout ale aplicației. În acest caz, metoda principală, placeOrder, este responsabilă pentru procesarea unei comenzi și returnarea unui răspuns legat de rezultatul acesteia.



V. Service implementation layer

Procesează o comandă plasată de un client, salvând datele în baza de date și returnând un răspuns cu numărul unic de urmărire al comenzii.
Anotarea @Transactional: Asigură că întreaga operațiune este tratată ca o tranzacție atomică. Dacă apare o eroare, toate modificările efectuate până la acel punct sunt anulate (rollback).

Parametri: Purchase purchase: Obiect DTO care conține toate informațiile necesare plasării comenzii (client, adrese, produse, și detalii despre comandă).
Returnează: PurchaseResponse: Răspuns care conține un număr unic de urmărire a comenzii (orderTrackingNumber), utilizat pentru a confirma comanda plasată.

Punctul critic al functiei:
customerRepository.save(customer);

- **Salvează clientul în baza de date** (Datorită relației bidirecționale dintre client și comandă (de obicei definită în entitățile JPA), comanda și produsele comandate sunt salvate automat datorită propagării tranzacției)

```java
@Override
@Transactional
public PurchaseResponse placeOrder(Purchase purchase) {

    Order order = purchase.getOrder();

    String orderTrackingNumber = generateOrderTrackingNumber();
    order.setOrderTrackingNumber(orderTrackingNumber);

    Set<OrderItem> orderItems = purchase.getOrderItems();
    orderItems.forEach(item -> order.add(item));

    order.setBillingAddress(purchase.getBillingAddress());
    order.setShippingAddress(purchase.getShippingAddress());

    Customer customer = purchase.getCustomer();

    String theEmail = customer.getEmail();
    Customer customerFromDB = customerRepository.findByEmail(theEmail);
    if(customerFromDB != null)
    {
        customer = customerFromDB;
    }

    customer.add(order);

    customerRepository.save(customer);

    return new PurchaseResponse(orderTrackingNumber);
}
```

VI. Controller layer

Clasa CheckoutController este un controller REST responsabil pentru gestionarea cererilor legate de procesul de checkout (plasarea unei comenzi). Această clasă este punctul de intrare în backend pentru cererile HTTP legate de plasarea comenzilor.

```java
@RestController
@RequestMapping("/api/checkout")
public class CheckoutController {

    2 usages
    private CheckoutService checkoutService;

    public CheckoutController(CheckoutService checkoutService) { this.checkoutService = checkoutService; }

    @PostMapping("/purchase")
    public PurchaseResponse placeOrder(@RequestBody Purchase purchase) {
        PurchaseResponse purchaseResponse = checkoutService.placeOrder(purchase);
        return purchaseResponse;
    }
}
```

Explicația codului frontend (Angular)

I. App.module.ts

Fișierul **app.module.ts** este fișierul central într-o aplicație Angular, deoarece configurează modulul principal al aplicației. Acesta definește componentele, directivele, pipe-urile, modulele externe și serviciile necesare pentru funcționarea aplicației.

Această secțiune reprezintă configurația rutelor în aplicația Angular, definită în obiectul routes de tip Routes (navigarea între diferitele pagini ale aplicației) . Fiecare element din acest array definește o rută specifică din aplicație, împreună cu componenta asociată și alte opțiuni (cum ar fi protecția rutelor cu canActivate).

```typescript
const routes: Routes= [
  {path: 'order-history', component:OrderHistoryComponent, canActivate: [OktaAuthGuard],
                  data: {onAuthRequired: sendToLoginPage}},


  {path: 'members', component:MembersPageComponent, canActivate: [OktaAuthGuard],
                  data: {onAuthRequired: sendToLoginPage}},//if authenticated give access to route else send to login page

  {path: 'login/callback', component:OktaCallbackComponent},
  {path: 'login', component:LoginComponent},

  {path: 'checkout', component:CheckoutComponent},
  {path: 'cart-details', component:CartDetailsComponent},
  {path: 'products/:id', component:ProductDetailsComponent},
  {path: 'search/:keyword', component:ProductListComponent},
  {path: 'category/:id', component: ProductListComponent},
  {path: 'category', component: ProductListComponent},
  {path: 'products', component: ProductListComponent},
  {path: '', redirectTo: '/products', pathMatch: 'full'},
  {path: '**', redirectTo: '/products', pathMatch: 'full'}
```

II. Service:

Product.service.ts:

Serviciul ProductService, este responsabil pentru interacțiunea cu backend-ul RESTful pentru a obține date legate de produse și categorii de produse.

Funcția general ProductService:

- Gestionează accesul la API-urile backend pentru produse și categorii.
- Oferă metode pentru:
  - Obținerea detaliilor unui produs.
  - Listarea produselor pe categorii sau prin cuvinte-cheie.
  - Paginarea rezultatelor.
  - Obținerea categoriilor de produse.

De asemenea contine interfetele auxiliare:

a. GetResponseProducts Definește structura răspunsului JSON pentru produse. Conține două părți: _embedded.products(Lista de produse) si page (Informații despre paginare) (dimensiunea paginii, numărul total de elemente etc.).

b. GetResponseProductCategory Definește structura răspunsului JSON pentru categoriile de produse. _embedded.productCategory(Lista de categorii de produse).

```
82  interface GetResponseProducts{
83    _embedded:{
84      products: Product[];
85    },
86    page:{
87      size: number,
88      totalElements: number,
89      totalPages: number,
90      number: number
91    }
92  }
93  //unwraps the json from spring data rest _embedded entry
94  interface GetResponseProductCategory{
95    _embedded:{
96      productCategory: ProductCategory[];
97    }
98  }
```

Order-history.service.ts

Serviciul **OrderHistoryService** este utilizat pentru a obține istoricul comenzilor unui client specific, pe baza adresei sale de email. Este un serviciu Angular care interacționează cu backend-ul RESTful pentru a efectua această operațiune.

Funcția generală

- Serviciul interacționează cu API-ul backend pentru a obține datele legate de comenzile unui client.
- Se bazează pe Spring Data REST, utilizând interogarea personalizată findByCustomerEmailOrderByDateCreatedDesc.
- Returnează datele într-un format specific pentru a fi utilizate în componentele Angular.

```typescript
@Injectable({
  providedIn: 'root'
})
export class OrderHistoryService {

  private orderUrl =environment.luv2shopApiUrl + '/orders';
  constructor(private httpClient: HttpClient) { }

  getOrderHistory(theEmail: string):Observable<GetResponseOrderHistory>
  {
    //neet to build url based on the customer email
    const orderHistoryUrl = `${this.orderUrl}/search/findByCustomerEmailOrderByDateCreatedDesc?email=${theEmail}`;
    return this.httpClient.get<GetResponseOrderHistory>(orderHistoryUrl);
  }
}

```

Luv2-shop-form.service.ts

Serviciul **Luv2ShopFormService** este responsabil pentru furnizarea de date necesare populării formularelor utilizate precum liste derulante pentru țări, state (provincii), luni și ani pentru cardurile de credit. Acesta interacționează cu backend-ul REST pentru a obține listele de țări și state și construiește programatic listele de luni și ani pentru validarea cardurilor de credit.

Funcția generala Luv2ShopFormService:

- Obține date necesare pentru formulare (țări, state).
- Generează programatic liste pentru dropdown-uri (luni și ani pentru cardurile de credit).
- Simplifică gestionarea datelor dinamice necesare pentru formularele de checkout sau înregistrare.

```typescript
  getCreditCardMonths(startMonth:number): Observable<number[]>{
    let data: number[] = [];

    //build an array for "Month" dropdown list
    //- start at current month and loop until

    for(let theMonth = startMonth; theMonth <=12; theMonth++)
    {
      data.push(theMonth);
    }

    return of(data);
  }


  getCreditCardYears():Observable<number[]>{
    let data: number[] = [];

    //build an array for "Year" downlist list
    //-start at current year and loop for the next 10 years

    const startYear: number = new Date().getFullYear();
    const endYear: number = startYear + 10;

    for(let theYear = startYear; theYear <= endYear; theYear++)
    {
      data.push(theYear);
    }

    return of(data);//wrap object as an Observable

  }
```

Cart.service.ts

Serviciul **CartService** este responsabil pentru gestionarea coșului de cumpărături, acesta gestionează articolele din coș, calculează totalurile (preț și cantitate) și persistă datele în **localStorage** sau **sessionStorage** pentru a oferi o experiență de utilizare continuă.

Funcția general cart.service.ts**:** Gestionează articolele din coșul de cumpărături.

- Calculează totalul prețului și cantității din coș.
- Persistă datele coșului în **localStorage** (sau alte medii de stocare).
- Permite adăugarea, modificarea cantității și ștergerea articolelor din coș.

```
11  cartItems: CartItem[] = [];
12
13  totalPrice: Subject<number> = new BehaviorSubject<number>(0);
14  totalQuantity: Subject<number> = new BehaviorSubject<number>(0);
15
16
17  //storage: Storage = sessionStorage;
18  storage:Storage = localStorage;//data is persisted even if you close the browser tab
19
20  constructor() {
21    //read data from storage
22    let data = JSON.parse(this.storage.getItem('cartItems'));
23
24    if(data !=null)
25    {
26      this.cartItems = data;
27
28      //compute totals based on the data that is read from storage
29      this.computeCartTotals();
30    }
31  }
32
```

```
87    persistCartItems(){ //adaug elem in browser
88      this.storage.setItem('cartItems', JSON.stringify(this.cartItems));
89    }
90
```

Auth-interceptor.service.ts

Serviciul **AuthInterceptorService** este un interceptor HTTP utilizat pentru gestionarea autentificării în cererile HTTP dintr-o aplicație Angular care folosește **Okta** pentru autentificare și autorizare. Acesta interceptează cererile HTTP și adaugă un **token de acces** în antetul cererii dacă URL-ul acesteia corespunde unui endpoint securizat.

Funcția generală

- **Interceptare cereri HTTP**: Interceptează toate cererile HTTP înainte de a fi trimise la backend.
- **Adăugare de token pentru autentificare**: Adaugă un token de acces (Bearer token) în antetul cererilor HTTP către endpoint-urile securizate.

```
export class AuthInterceptorService implements HttpInterceptor{

  constructor(@Inject(OKTA_AUTH)private oktaAuth:OktaAuth) { }
  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    return from(this.handleAccess(request,next));
  }
  private async handleAccess(request: HttpRequest<any>, next: HttpHandler): Promise<HttpEvent<any>> {
    const theEndpoint = environment.luv2shopApiUrl+'/orders';
    //Only add an access token for secured endpoints
    const securedEndpoints = [theEndpoint];

    if(securedEndpoints.some(url => request.urlWithParams.includes(url))){
      //get access token
      const accessToken = this.oktaAuth.getAccessToken();
      //clone the request and add new header with access token
      request = request.clone({
        setHeaders:{
          Authorization:'Bearer ' + accessToken
        }
      });
    }
    return await lastValueFrom(next.handle(request));
  }
}
```

III. Components:

Search.components

Această componentă, SearchComponent, oferă funcționalitatea de căutare a produselor. Este utilizată pentru a captura un cuvânt-cheie introdus de utilizator, apoi redirecționează utilizatorul către o pagină care afișează rezultatele căutării pe baza acelui cuvânt-cheie.

Flow: Căutarea prin Enter:

- Utilizatorul introduce un cuvânt-cheie în câmpul de text și apasă Enter.
- Se declanșează evenimentul (keyup.enter), care: apelează metoda doSearch si navighează către URL-ul /search/{value}, unde {value} este cuvântul introdus.

Product-list.component

Componenta **ProductListComponent** este responsabilă pentru afișarea listei de produse. Aceasta permite utilizatorului să vizualizeze produse în funcție de categorii, să caute produse pe baza unui cuvânt-cheie și să adauge produse în coșul de cumpărături. De asemenea, implementează funcționalități de paginare și actualizare a dimensiunii paginii pentru o experiență de utilizare mai eficientă.

Flow:

1. **Inițializare (ngOnInit)**: Abonează componenta la parametrii rutei (paramMap), astfel încât să detecteze schimbările de categorie sau căutare si apelează metoda listProducts() pentru a încărca produsele relevante.
2. **Determinarea modului de afișare**:
   - **Mod de căutare**: Dacă URL-ul conține un parametru keyword, este activat modul de căutare si se apelează handleSearchProducts() pentru a obține rezultatele căutării.
   - **Mod de categorie**: Dacă URL-ul conține un parametru id, produsele sunt afișate pentru categoria specificată si se apelează handleListProducts().
3. **Obținerea datelor**: ProductService este utilizat pentru a obține datele din backend iar datele sunt prelucrate prin metoda processResult() pentru a popula proprietățile componentei.
4. **Interacțiunea utilizatorului**:

- **Paginare**: Utilizatorul poate schimba pagina folosind ngb-pagination si apelarea metodei listProducts() reîncarcă produsele pentru pagina curentă.
- **Adăugare în coș**: Utilizatorul poate adăuga produse în coș prin clic pe butonul "Add to cart" si CartService gestionează adăugarea produsului.

Product-details.component

Componenta **ProductDetailsComponent** este utilizată pentru afișarea detaliilor unui produs specific. Aceasta permite utilizatorului să vizualizeze informații detaliate despre un produs (cum ar fi imaginea, prețul și descrierea) și să adauge produsul în coșul de cumpărături.

Fluxul componentei

1. **Inițializarea componentei**: ngOnInit() detectează parametrul id din URL prin ActivatedRoute si apelează handleProductDetails() pentru a încărca informațiile despre produsul corespunzător.
2. **Obținerea datelor despre produs**: handleProductDetails() preia id-ul produsului din URL, dupa folosește ProductService pentru a obține detaliile produsului din backend si stochează detaliile produsului în proprietatea product.
3. **Afișarea informațiilor în interfață**:Detaliile produsului sunt afișate în șablonul HTML utilizând binding-ul Angular.
4. **Adăugarea în coș**: La clic pe butonul "Add to cart", metoda addToCart() este declanșată si produsul este adăugat în coș folosind CartService.
5. **Navigarea înapoi**:Link-ul "Back to Product List" permite utilizatorului să revină la pagina cu lista de produse.

```
22  ngOnInit(): void {
23    this.route.paramMap.subscribe(()=>{
24      this.handleProductDetails();
25    })
26  }
27    handleProductDetails() {
28      //get the id param string. convert string to a number using the + symbol
29      const theProductId: number = +this.route.snapshot.paramMap.get('id')!;
30
31      this.productService.getProduct(theProductId).subscribe(
32        data =>{
33          this.product = data;
34        }
35      )
36    }
37
38    addToCart(){
39      console.log(`Adding to cart: ${this.product.name}, ${this.product.unitPrice}`);
40      const theCartItem = new CartItem(this.product);
41      this.cartService.addToCart(theCartItem);
42    }
43  }
```

Product-category-menu.component

Componenta **ProductCategoryMenuComponent** este responsabilă pentru afișarea unui meniu lateral care listează categoriile de produse disponibile într-o aplicație eCommerce. Fiecare categorie este un link care permite utilizatorului să navigheze și să vadă produsele asociate acelei categorii.

Fluxul componentei

1. **Inițializarea componentei**:Metoda ngOnInit() este apelată la inițializarea componentei si aceasta apelează metoda listProductCategories().
2. **Obținerea datelor din backend**: listProductCategories() utilizează serviciul ProductService pentru a obține categoriile de produse,iar rezultatul este stocat în productCategories.
3. **Construirea meniului**: HTML-ul iterează prin productCategories utilizând *ngFor si pentru fiecare categorie, creează un link dinamic către pagina acelei categorii.
4. **Navigarea utilizatorului**:
    - Când utilizatorul face clic pe o categorie:
        - Navighează către URL-ul corespunzător (/category/{id}).
        - Componenta responsabilă de afișarea produselor (ex. ProductListComponent) gestionează afișarea produselor din categoria selectată.

Order-history.component

Componenta **OrderHistoryComponent** este utilizată pentru afișarea istoricului comenzilor unui utilizator într-o aplicație eCommerce. Aceasta recuperează comenzile plasate de utilizator și le afișează într-un tabel, oferind detalii precum numărul de urmărire al comenzii, prețul total, cantitatea totală și data plasării comenzii. Dacă nu există comenzi, se afișează un mesaj corespunzător.

Fluxul componentei

1. **Inițializarea componentei**: Metoda ngOnInit() este apelată la încărcarea componentei iar aceasta apelează metoda handleOrderHistory() pentru a recupera datele despre comenzi.
2. **Recuperarea datelor despre comenzi**: Adresa de email a utilizatorului este citită din sessionStorage. Serviciul OrderHistoryService este utilizat pentru a obține comenzile asociate acelei adrese de email. Rezultatele sunt stocate în orderHistoryList.
3. **Afișarea datelor**: Tabelul afișează comenzile utilizând *ngFor iar dacă lista este goală, se afișează un mesaj de avertizare.

```
9   export class OrderHistoryComponent implements OnInit{
10
11    orderHistoryList: OrderHistory[] = [];
12
13    storage: Storage = sessionStorage;
14
15    constructor(private orderHistoryService:OrderHistoryService){
16
17    }
18
19    ngOnInit(): void {
20      this.handleOrderHistory();
21    }
22
23    handleOrderHistory(){
24      //read the user's email address from browser storage
25      const theEmail = JSON.parse(this.storage.getItem('userEmail')!);
26
27      //retrieve data from the service
28      this.orderHistoryService.getOrderHistory(theEmail).subscribe(
29        data=>{
30          this.orderHistoryList = data._embedded.orders;
31        }
32      );
33    }
```

Login-status.component

Componenta **LoginStatusComponent** este utilizată pentru a afișa starea autentificării utilizatorului într-o aplicație Angular care folosește **Okta** pentru autentificare și autorizare. Aceasta gestionează logarea și delogarea utilizatorului, afișează numele utilizatorului autentificat și oferă acces la funcționalități suplimentare, cum ar fi vizualizarea istoricului comenzilor sau accesul la o zonă de membri.

Fluxul componentei

1. **Inițializarea componentei**: ngOnInit() se abonează la modificările stării de autentificare utilizând OktaAuthStateService.
2. **Obținerea stării de autentificare**: Când starea de autentificare se actualizează isAuthenticated.Dacă utilizatorul este autentificat, apelează getUserDetails() pentru a prelua numele și email-ul utilizatorului.
3. **Afișarea informațiilor utilizatorului**: Dacă utilizatorul este autentificat, afișează numele său și linkuri către funcționalități suplimentare.Dacă utilizatorul nu este autentificat, afișează un buton de logare.
4. **Delogarea utilizatorului**: La clic pe butonul "Logout", metoda logout() este apelată pentru a încheia sesiunea curentă cu **Okta**.

```
export class LoginStatusComponent implements OnInit {
  isAuthenticated: boolean= false;
  userFullName: string = '';

  storage: Storage = sessionStorage;

constructor(private oktaAuthService: OktaAuthStateService, @Inject(OKTA_AUTH) private oktaAuth: OktaAuth){

}

ngOnInit(): void {
  //subscribe to authentication state changes
  this.oktaAuthService.authState$.subscribe(
   (result) =>{
     this.isAuthenticated = result.isAuthenticated!;
     this.getUserDetails();
   }
  );
   getUserDetails(){
    if( this.isAuthenticated){
      //fetch the logged in user details(user's claims)

      //user full name is exposed as a property name
      this.oktaAuth.getUser().then(
        (res)=>{
          this.userFullName = res.name as string;

          //retrieve the user's email from authentication response
          const theEmail= res.email;

          //store the email in browser storage
          this.storage.setItem('userEmail', JSON.stringify(theEmail));

        }
      );
   }
   }
   logout(){
    //terminates the session with Okta and removes current tokens
    this.oktaAuth.signOut();
   }
```

Login.component

Componenta **LoginComponent** este utilizată pentru autentificarea utilizatorilor într-o aplicație Angular folosind widget-ul de autentificare **Okta Sign-In Widget**. Aceasta gestionează afișarea widget-ului în interfață, configurarea acestuia cu parametrii necesari și redirecționarea utilizatorului după autentificare reușită.

Fluxul componentei

1. **Inițializarea componentei**: ngOnInit() este apelat iar Widget-ul Okta este eliminat (dacă există) și apoi renderizat din nou în containerul #okta-sign-in-widget.
2. **Afișarea widget-ului Okta Sign-In**: Widget-ul permite utilizatorului să introducă acreditivele (nume de utilizator și parolă).
3. **Autentificare reușită**: Dacă autentificarea este reușită, callback-ul response.status === 'SUCCESS' este apelat si utilizatorul este redirecționat către pagina specificată în redirectUri.
4. **Gestionarea erorilor**: Dacă există o eroare la autentificare sau configurare, aceasta este gestionată prin aruncarea unei excepții.

```
  oktaSignin: any;
constructor(@Inject(OKTA_AUTH) private oktaAuth:OktaAuth){
  this.oktaSignin = new OktaSignIn(
    {
      logo: 'images/logo.png',
      baseUrl: myAppConfig.oidc.issuer.split('/oauth2')[0],
      clientId: myAppConfig.oidc.clientId,
      redirectUri: myAppConfig.oidc.redirectUri,
      authParams:{
        pkce: true,
        issuer: myAppConfig.oidc.issuer,
        scopes: myAppConfig.oidc.scopes
      }
    }
  );
}

ngOnInit(): void {
  this.oktaSignin.remove();

  this.oktaSignin.renderEl({
    el: '#okta-sign-in-widget'},//this name shoul be same as div tag in login.compoent.html
    (response: any)=>{
      if( response.status === 'SUCCESS'){
        this.oktaAuth.signInWithRedirect();
      }
    },
    (error: any) =>{
      throw error;
    }
  );
```

Checkout.component

Componenta **CheckoutComponent** reprezintă un formular avansat de checkout. Aceasta gestionează validarea datelor introduse, adunarea informațiilor de livrare și facturare, detaliile cardului de credit și trimiterea comenzii către backend. Componenta integrează mai multe servicii și funcționalități pentru a oferi o experiență completă de checkout.

Metode principale:

- **reviewCartDetails()**:Se abonează la modificările din CartService pentru a actualiza totalul prețului și cantității.
- **onSubmit()**:Validează formularul și, dacă este valid:
  - Creează obiecte pentru comandă (Order) și articolele comenzii (OrderItem).
  - Populează informațiile pentru livrare, facturare și card de credit.
  - Trimite comanda către backend folosind CheckoutService.
  - Resetează coșul și formularul după plasarea comenzii.

- **copyShippingAddressToBillingAddress()**: Copiază adresa de livrare în adresa de facturare, dacă este bifată opțiunea.
- **handleMonthsAndYears()**: Actualizează lista lunilor disponibile pe baza anului selectat.
- **getStates(formGroupName)**: Populează lista de state pentru țara selectată (atât pentru livrare, cât și pentru facturare).

Fluxul componentei

1. **Inițializare**: Formularele și listele dinamice sunt populate si detaliile coșului sunt actualizate.
2. **Validare**: Formularul validează câmpurile pe măsură ce utilizatorul introduce date.
3. **Submit**: Utilizatorul completează formularul și apasă "Purchase".Dacă formularul este valid: creează și trimite comanda către backend si resetează coșul și formularul.
4. **Resetare**: După plasarea comenzii, utilizatorul este redirecționat la pagina de produse.

Cart-status.component

Componenta **CartStatusComponent** este utilizată pentru a afișa starea curentă a coșului de cumpărături. Aceasta afișează numărul total de articole din coș și prețul total al acestora. Componenta este reactivă, actualizându-se automat atunci când se modifică conținutul coșului.

Fluxul componentei

1. **Inițializare**: ngOnInit() este apelat la inițializarea componentei. Se invocă metoda updateCartStatus() pentru a începe ascultarea modificărilor din coș.
2. **Actualizarea datelor**: CartService emite modificări ale prețului total (totalPrice) și cantității totale (totalQuantity). Componenta se actualizează automat pe baza acestor modificări.
3. **Afișarea datelor**: Totalul prețului și numărul total de articole sunt afișate în interfața utilizatorului.
4. **Navigare**: La clic pe zona coșului, utilizatorul este redirecționat către /cart-details.

```
updateCartStatus() {

  //subscribe to the cart totalPrice
  this.cartService.totalPrice.subscribe(
    data => this.totalPrice = data
  );
  //subscribe to the cart totalQuantity
  this.cartService.totalQuantity.subscribe(
    data => this.totalQuantity = data
  );

}
```

Cart-details.component

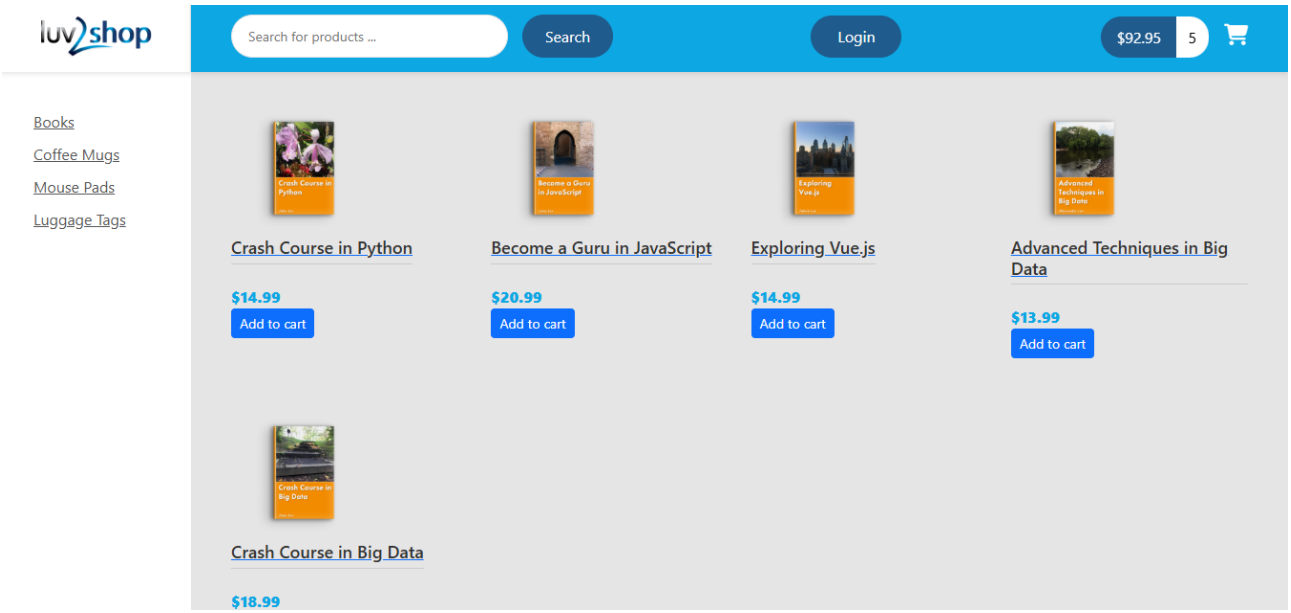
Componenta **CartDetailsComponent** este utilizată pentru a afișa detalii despre coșul de cumpărături. Aceasta permite utilizatorilor să vadă produsele din coș, să actualizeze cantitățile, să elimine produse și să navigheze către procesul de checkout.

1. **Fluxul componentei**
2. **Inițializare**: La inițializare, listCartDetails() preia produsele din coș și calculează totalurile.
3. **Actualizări în timp real**: Componenta se actualizează automat când:
    - Se adaugă un produs în coș.
    - Se modifică cantitatea unui produs.
    - Se elimină un produs din coș.

4. **Acțiuni utilizator**:Utilizatorul poate:
   - Crește/scădea cantitatea unui produs.
   - Elimina un produs complet.
   - Naviga la checkout pentru finalizarea comenzii.

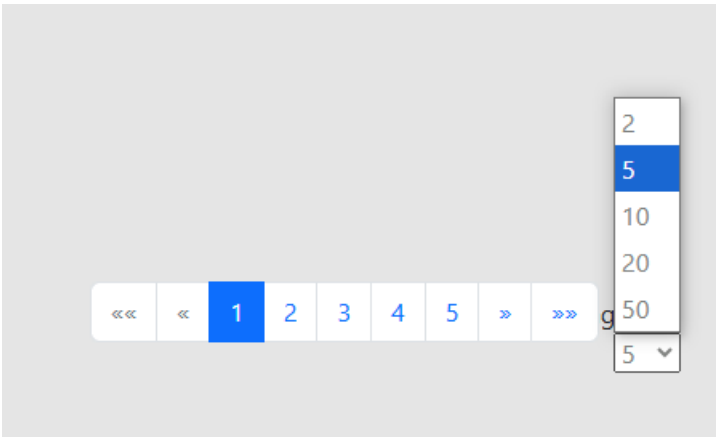
```
listCartDetails() {
  //get a handle to the cart items
  this.cartItems = this.cartService.cartItems;

  //subscribe to the cart totalPrice
  this.cartService.totalPrice.subscribe(
    data => this.totalPrice =data
  );

  //subscribe to the cart totalQuantity
  this.cartService.totalQuantity.subscribe(
    data => this.totalQuantity = data
  );
  //compute cart total price and quantity
  this.cartService.computeCartTotals();
}

incrementQuantity(theCartItem: CartItem){
  this.cartService.addToCart(theCartItem);
}
decrementQuantity(theCartItem: CartItem){
  this.cartService.decrementQuantity(theCartItem);
}

remove(theCartItem: CartItem){
  this.cartService.remove(theCartItem);
}

```

Prezentare generală a interfeței aplicației



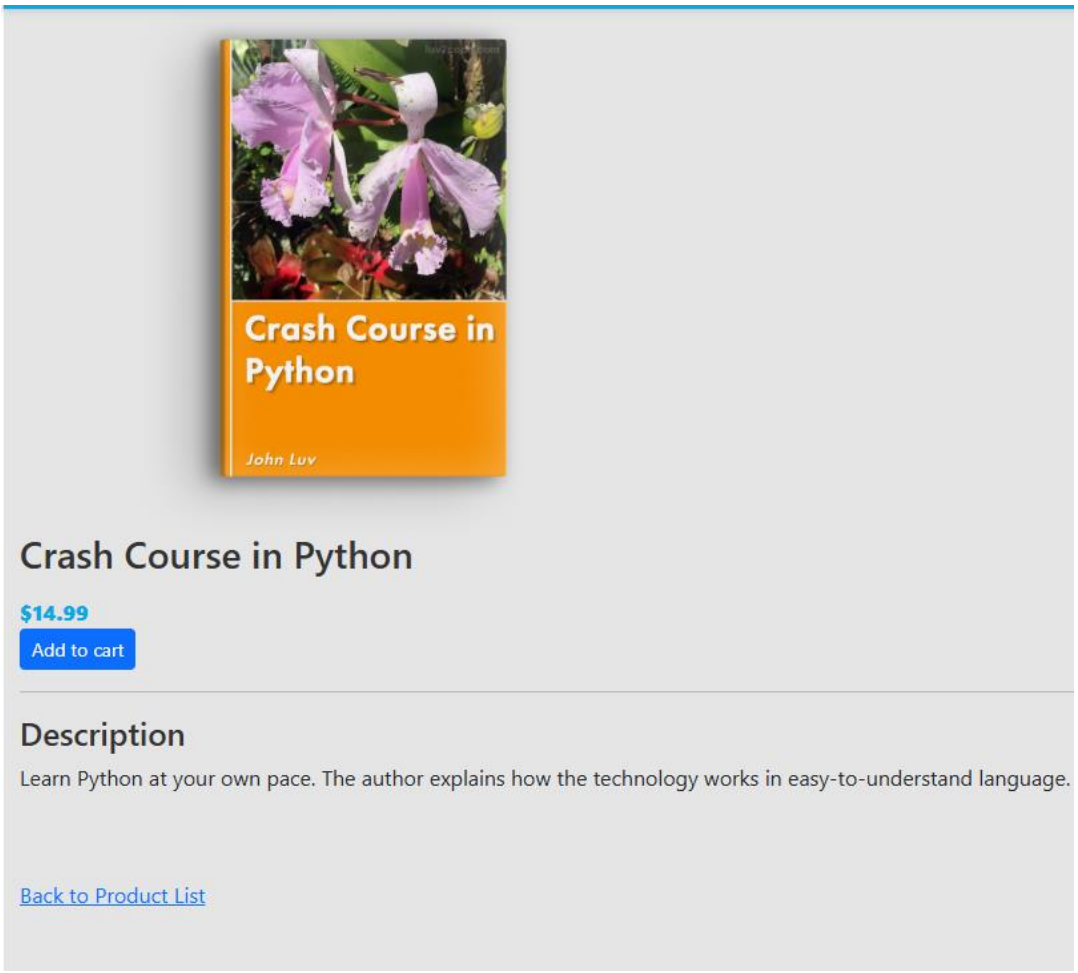
Imaginea prezintă pagina principală a aplicației eCommerce **Luv2Shop**, incluzând:

1. **Bară de navigare superioară**:
2. **Categorii de produse**:
3. **Produse afișate**:



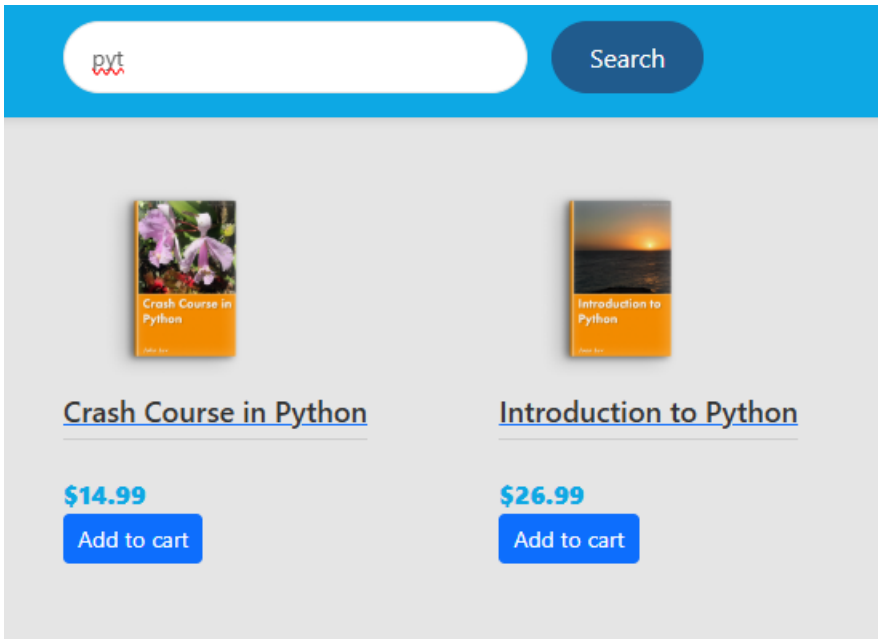
Imaginea prezintă **funcționalitatea de paginare** a aplicației eCommerce:

1. **Navigarea între pagini**:
   - Butoane numerotate permit utilizatorului să navigheze între pagini.
   - Butoanele „<<” și „>>” oferă acces rapid la prima și ultima pagină.
2. **Selectarea dimensiunii paginii**:
   - Un meniu dropdown permite utilizatorului să aleagă câte produse să fie afișate pe o pagină (2, 5, 10, 20, 50).



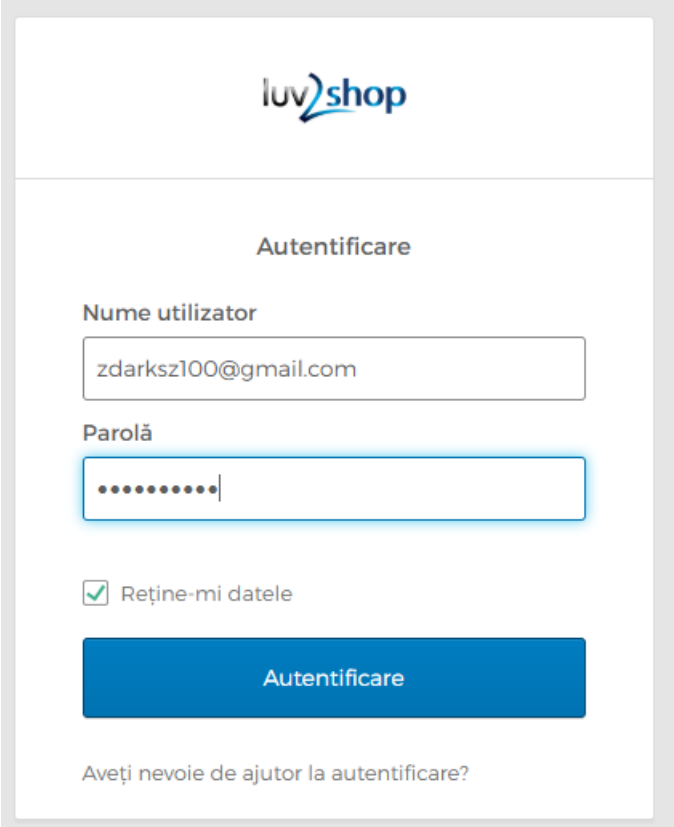
Imaginea prezintă **pagina de detalii a unui produs** din aplicația eCommerce:

1. **Informații despre produs**:
2. **Acțiune disponibilă**: Buton „Add to cart”: Permite utilizatorului să adauge produsul în coșul de cumpărături.
3. **Descrierea produsului**: Oferă detalii suplimentare despre produs, explicând ce învață utilizatorul din acest curs.
4. **Navigare înapoi**:



Imaginea prezintă **pagina de rezultate pentru o căutare** în aplicația eCommerce:

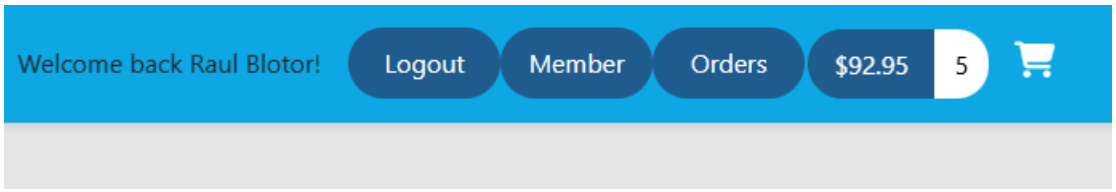
1. **Căutare realizată**: Cuvântul „pyt” a fost introdus în bara de căutare, iar aplicația a returnat produsele relevante.
2. **Produsele afișate**: Fiecare produs are un buton „Add to cart” pentru a fi adăugat în coș.
3. **Navigare intuitivă**: Bara de căutare rămâne vizibilă, permițând utilizatorului să efectueze alte căutări.



Imaginea prezintă **pagina de autentificare** din aplicația eCommerce **Luv2Shop**:

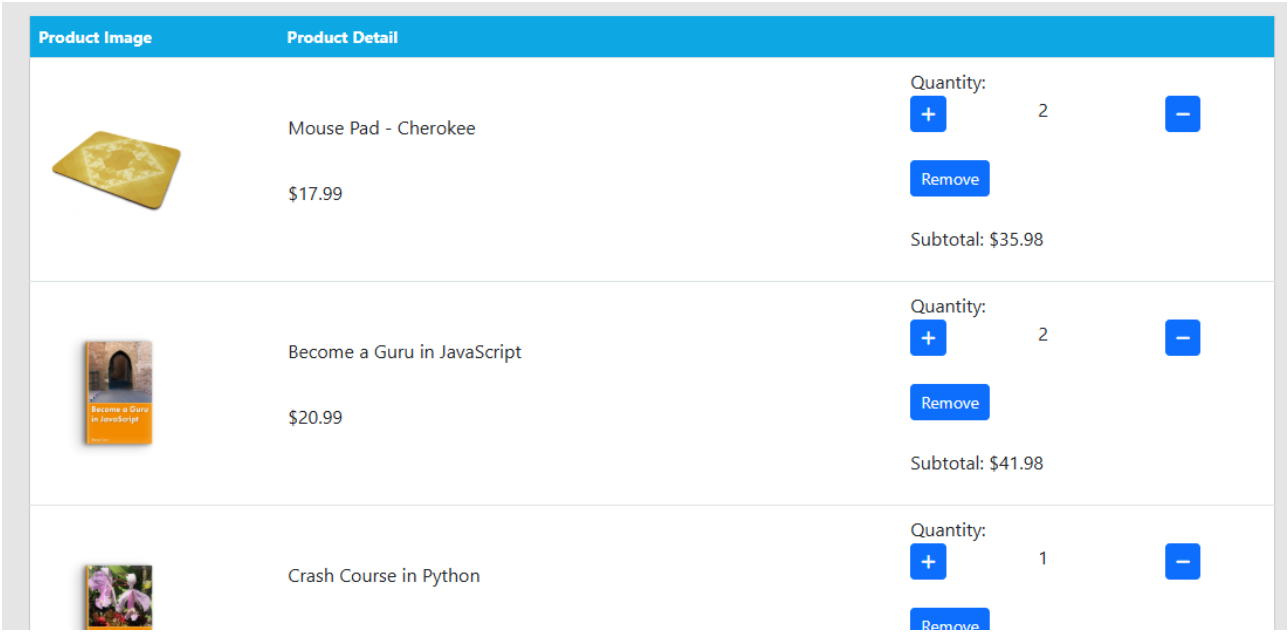
1. **Câmpuri pentru autentificare**:

2. **Opțiuni suplimentare**: **Checkbox „Reține-mi datele”**: Permite utilizatorului să rămână conectat.
3. **Buton de acțiune**: **„Autentificare”**: Declanșează procesul de logare în aplicație.
4. **Link pentru asistență**:



Imaginea prezintă **bara de navigare** afișată după ce utilizatorul s-a autentificat cu succes în aplicația eCommerce **Luv2Shop**:

1. **Mesaj de bun venit**: „Welcome back Raul Blotor!” indică faptul că utilizatorul este autentificat.
2. **Funcționalități disponibile**:
   - **„Logout”**: Permite utilizatorului să se deconecteze din cont.
   - **„Member”**: Acces către secțiunea dedicată utilizatorilor autentificați.
   - **„Orders”**: Navighează către istoricul comenzilor utilizatorului.
3. **Starea coșului**:
   - **Preț total al coșului**: $92.95.
   - **Numărul de produse din coș**: 5.
4. **Iconița coșului**: Permite acces rapid către pagina cu detaliile coșului de cumpărături.

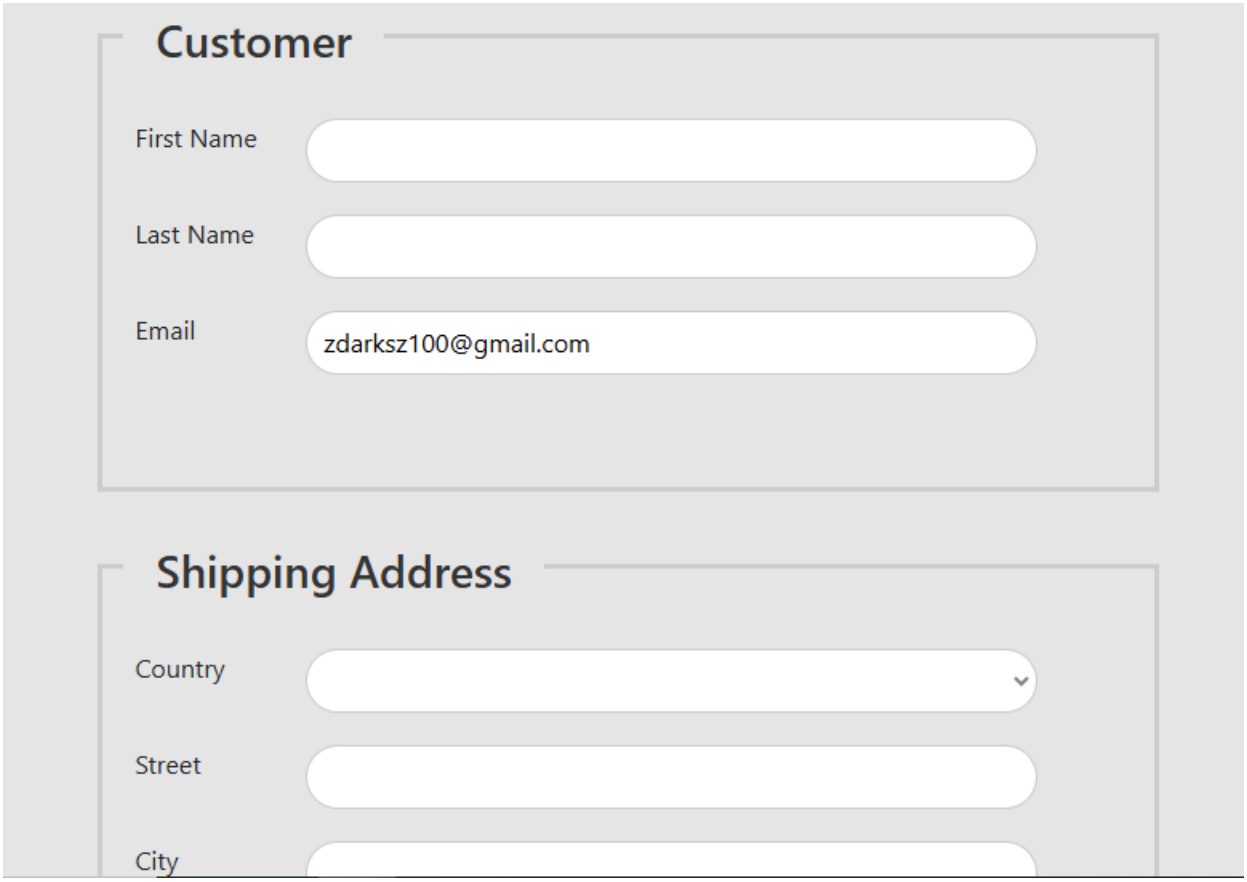


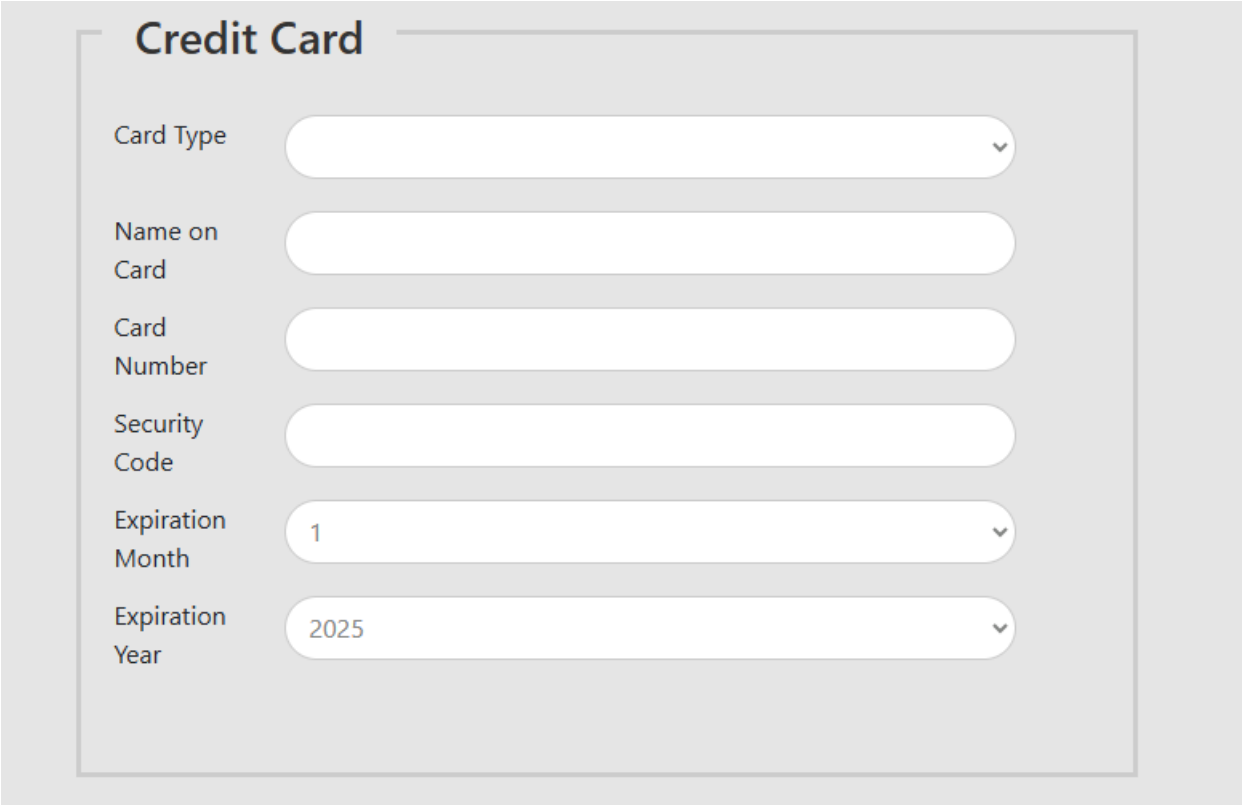
Imaginea prezintă **pagina de coș de cumpărături**, care permite utilizatorilor să gestioneze articolele adăugate înainte de finalizarea comenzii.

Detalii afișate: Tabel cu produse din coș:

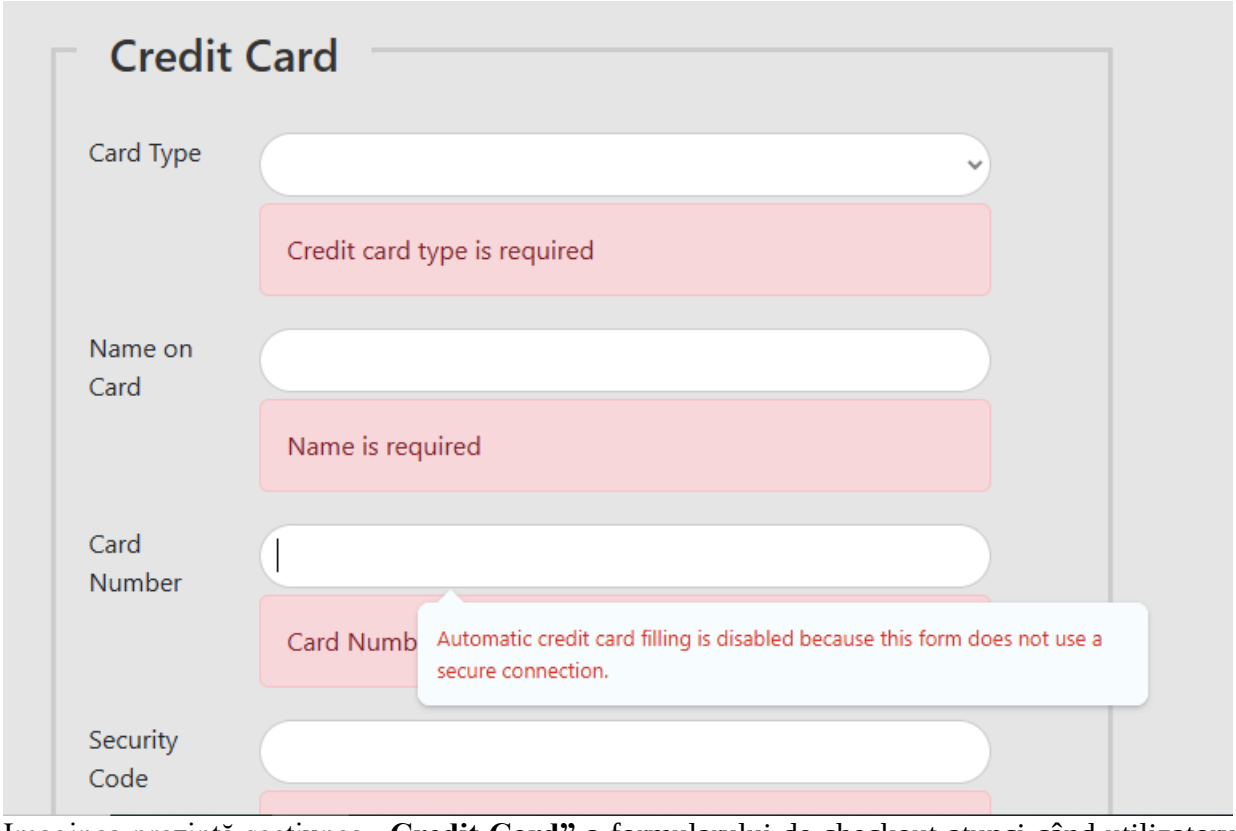
- **Product Image**: Afișează imaginea fiecărui produs.
- **Product Detail**: Numele și prețul produsului sunt afișate.
- **Quantity**: Utilizatorul poate ajusta cantitatea folosind butoanele:
  - +: Crește cantitatea.
  - -: Scade cantitatea (elimină produsul dacă cantitatea devine 0).

- **Remove**: Buton pentru a elimina complet produsul din coș.
- **Subtotal**: Afișează costul total pentru fiecare produs (cantitate × preț).





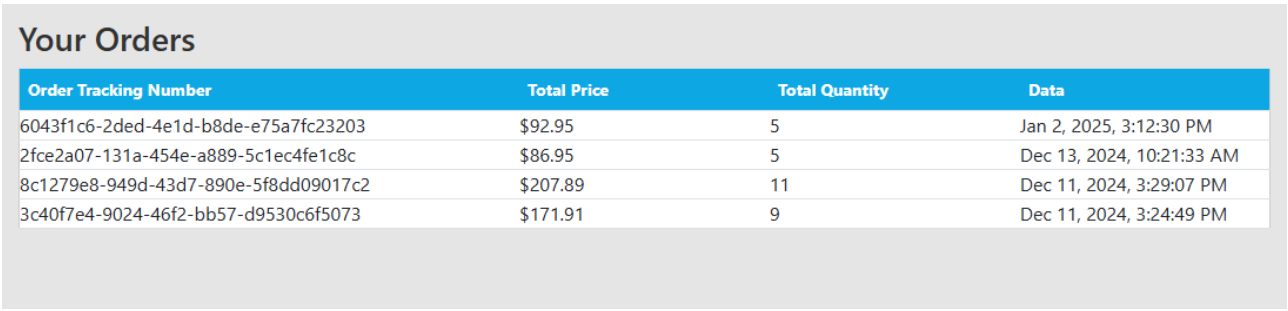
Imaginile prezintă **formularul de checkout** din aplicația eCommerce, care permite utilizatorului să finalizeze comanda prin completarea datelor necesare.



Imaginea prezintă secțiunea **„Credit Card”** a formularului de checkout atunci când utilizatorul nu completează câmpurile obligatorii, iar validarea formularului afișează erori.

Mesaje de eroare afișate:

- **Card Type**: „Credit card type is required” – Utilizatorul trebuie să selecteze tipul cardului.
- **Name on Card**: „Name is required” – Numele titularului cardului este obligatoriu.
- **Card Number**: „Card Number” – Lipsa unui număr de card valid.
- **Security Code**: Mesaj similar pentru completarea codului de securitate.



Imaginea prezintă **pagina de istoricul comenzilor** din aplicația eCommerce, unde utilizatorul poate vedea detalii despre comenzile plasate anterior.

1. **Detalii afișate**
2. **Titlu**:
    - „Your Orders” – indică faptul că secțiunea afișează comenzile utilizatorului autentificat.
3. **Tabel cu detalii despre comenzi**:
    - **Order Tracking Number**: Numărul unic de identificare al fiecărei comenzi, utilizat pentru urmărirea acesteia.
    - **Total Price**: Costul total al comenzii.
    - **Total Quantity**: Numărul total de produse din comandă.
    - **Data**: Data și ora plasării comenzii.

Concluzie

Această documentație detaliază funcționalitățile și structura aplicației **Luv2Shop**, un magazin online dezvoltat cu **Spring Boot**, **Angular**, și **MySQL**. Aplicația oferă o experiență completă pentru utilizatori, de la navigarea și căutarea produselor până la plasarea comenzilor și vizualizarea istoricului acestora.

Puncte cheie ale documentației:

1. **Backend robust și bine organizat:**Straturile aplicației sunt structurate logic: entități, DAO, servicii și controlere REST.Folosirea JPA și a Spring Data simplifică interacțiunea cu baza de date, iar serviciile oferă logică clară pentru procese precum checkout-ul sau gestionarea comenzilor.
2. **Frontend modern și intuitiv:**Angular este utilizat pentru a crea o interfață dinamică, responsivă și ușor de navigat.Componenta de gestionare a stării coșului și mecanismele de paginare oferă o experiență de utilizator fluidă.Formulare validate pentru datele de plată și livrare asigură acuratețea informațiilor introduse de utilizatori.
3. **Funcționalități principale detaliate:** Pagina principală prezintă produse, categorii și funcționalitatea de căutare.Coșul de cumpărături permite adăugarea, modificarea și eliminarea produselor, iar checkout-ul finalizează procesul prin plasarea comenzii.Sistemul de autentificare cu Okta oferă securitate utilizatorilor, iar istoricul comenzilor permite consultarea tranzacțiilor anterioare.
4. **Design centrat pe utilizator:**Interfața grafică este simplă și clară, oferind un flux logic al utilizatorului.Funcționalitățile de validare și erorile clar marcate îmbunătățesc experiența de utilizator și asigură acuratețea datelor.