|  |
| --- |
|  |
| Magazin on-line |
| Proiect JSF-JPA Mureșan Horea  Rîștei Anamaria Andra  Grupa 244, Sisteme distribuite în Internet |

# Introducere

Tema aleasă presupune implementarea unei soluții pentru un magazin on-line folosind JavaServer Pages și JavaServer Faces.

Proiectul implementat este menit sa servească o interfata posibililor clienți ai unui magazin on-line și cuprinde următoarele funcționalități principale:

* clientul poate să vizualizeze produsele și să le filtreze după categoria de care aparțin, precum și după cuvinte cheie din nume și descriere;
* clientul poate să adauge un produs în coșul de cumpărături, specificând și cantitatea de produse dorită;
* clientul poate să vizualizeze conținutul coșului de cumpărături și să șteargă produsele pe care nu le mai dorește;
* clientul poate să anuleze comanda sau să o finalizeze, această acțiune necesitând completarea unui formular de comandă pentru a specifica anumite detalii de livrare.

Implementarea realizată de noi este centrată pe cele două tehnologii principale cerute în cadrul acestei teme, dar sunt folosite și alte concepte specifice programării orientată obiect, precum: șablonul MVC pentru o structură modulară, clară și organizată a aplicației, conceptul de single responsability, prin care fiecare metodă este responsabilă de o singură operație, dar și aspecte de programare web precum CSS și JavaScript aplicate paginilor JSF. Proiectul este arhivat sub forma unei arhive war exploded și deployed pe un server de aplicații GlassFish.

În continuare vor fi prezentate straturile aplicației în conformitate cu șablonul MVC.

# Model

Obiectele folosite pentru modelarea conceptelor reale în contextul unui magazin sunt următoarele:

1. Product, reprezentat în baza de date ca înregistrare în tabelul Products, având coloanele:

* productId - un număr întreg, generat automat la inserarea în baza de date
* productName - un text de maxim 255 caractere
* productDescription - un text cuprinzând descrierea produsului sub forma unor perechi de tipul "atribut:valoare;" (s-a ales această forma de reprezentare pentru a putea sparge ulterior informația stocată în grupuri și a o servi clientului într-o formă mai organizată și mai ușor de citit)
* productCategory - text reprezentând categoria de care aparține produsul (Laptop, Telefon, etc.)
* productPrice - un număr real
* productQuantity - un număr întreg, actualizat de fiecare dată când se finalizează o comandă pentru produsul respectiv
* productImageFileName - text conținând numele fișierului imagine corespunzător produsului

Astfel, clasa java corespunzătoare acestui obiect respectând conceptele JPA este următoarea:

@Entity

@Table(name = "Products")

public class Product {

@Id

@GeneratedValue

private int productId;

private String productName;

private String productDescription;

private String productCategory;

private Float productPrice;

private int productQuantity;

private String productImageFileName;

//constructors, getters, setters

}

1. Order, obiect care conține datele necesare pentr a face livrarea la persoana corectă și la adresa corectă este persistat în baza de date în tabelul Orders:

* orderId - număr întred, generat automat
* clientName - text reprezentând numele complet al clientului
* clientAddress - text reprezentând adresa clientului
* orderTotal - număr real reprezentând suma totală pe care clientul trebuie să o plătească pentru comandă (plata se va face la primirea coletului)
* orderDate - data (inclusiv minute) la care a fost plasată comanda

Clasa java, împreună cu adnotările specifice JPS, corespunzătoare acestui obiect este următoarea:

@Entity

@Table(name = "Orders")

public class Order {

@Id

@GeneratedValue

private int orderId;

private String clientName;

private String clientAddress;

private float orderTotal;

private Date orderDate;

//constructors, getters, setters

}

1. ProductToOrder ilustrează legătra dintre datele introduse de client în formularul de comandă (salvate în obiectul Order) și produsele comandate, reprezentând astfel o mapare a unui obiect Order la mai multe obiecte Product:

* ptoId - un număr întreg, generat automat
* productId - identificatorul produsului comandat de client (deja existent în tabela Products)
* orderId - identificatorul datelor de comandă introduse de client (deja existent în tabela Orders)
* productQuantity - număr întreg reprezentând cantitatea de produse dorite de client (nu poate depăși cantitatea disponibilă)

Clasa java și adnotările JPA corespunzătoare acestui obiect sunt:

@Entity

@Table(name = "ProductToOrder")

public class ProductToOrder {

@Id

@GeneratedValue

private int ptoId;

private int productId;

private int orderId;

private int productQuantity;

//constructors, getters, setters

}

Tot la nivelul acesta sunt implementate și două clase de validare a obiectelor Product și Order, pentru a asigura faptul ca valorile câmpurilor lor sunt corespunzătoare cu logica de business. Mai exact, se verifică dacă se respectă lungimea anumitor câmpuri (clientName să fie între 1 și 100 de caractere, clientAddress să fie între 1 și 255 caractere), sau dacă unele câmpuri au valori relevante pentru a se putea continua operația (orderTotal să fie mai mare decât zero).

Pentru că aplicația este implementată folosind JPS pentru conectarea la baza de date, interogarea acesteia, modificarea și inserarea obiectelor în tabele, s-a definit un fișier persistence.xml în care este configurată conexiunea la serverul MYSQL, împreună cu numele bazei de date și al credențialelor necesare conectării la aceasta:

<persistence-unit name="mysql">

<!-- connection source for GlassFish -->

<jta-data-source>jdbc/mysql</jta-data-source>

<!-- \*\*\* -->

<properties>

<property name="showSql" value="true"/>

<property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect"/>

<property name="hibernate.hbm2ddl.auto" value="create"/>

<property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>

<property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/onlineshopdb"/>

<property name="javax.persistence.jdbc.user" value="root"/>

<property name="javax.persistence.jdbc.password" value=""/>

</properties>

</persistence-unit>

# Repository

La nivelul repository au fost implementate două clase diferite de repository, fiecare responsabilă de operații corespunzătoare produselor, respectiv persistării comenzilor plasate de client.

Clasa ProductRepo implementează operațiile CRUD necesare gestionării obiectelor Product(getAll, getBy\*, update).

Clasa OrderRepo este responsabilă pentru persistarea comenzilor valide depuse de către client. Pentru aceasta, implementează operații de inserare atât a obiectelor Order, cât și a obiectelor ProductToOrder, după ce acestea au fost validate folosind instanțe ale clasei de validare corespunzătoare.

Pentru a accesa tabelele necesare din baza de date se folosește o instanță a EntityManager asociată configurațiilor din persistence.xml:

@PersistenceContext(unitName = "mysql")

private EntityManager entityManager;

Acest obiect este folosit pentru:

* a citi înregistrări din baza de date:

TypedQuery<Product> query = entityManager.createQuery(

"SELECT p FROM Product p WHERE p.productQuantity > 0", Product.class);

productList = query.getResultList();

* a modifica înregistrări deja existente: entityManager.merge(product);
* a insera înregistrări noi: entityManager.persist(order);

# Controller

Clasa controller este responsabilă pentru delegarea responsabilităților metodelor din repository, în funcție de operațiile care sunt necesar de îndeplinit. De asemenea, este responsabilă pentru invocarea metodelor corecte din nivelul view în funcție de datele primite de la nivelul repository.

Clasa numită ClientController conține o instanță a clasei ProductRepo și o instanță a clasei OrderRepo, pentru a putea gestiona request-urile primite de la stratul superior în ceea ce privește vizualizarea produselor, depunerea unei comenzi, introducerea informațiilor despre un client.

# View

La acest nivel sunt implementate paginile web care sunt prezentate clientului în cadrul aplicației. Acestea sunt scrise folosind tehnologia JSF, astfel că pentru fiecare obiect folosit în pagină, există un bean corespunzător.

Spre exemplu, pagina principală conține un formular pentru filtrarea produselor după cuvinte cheie din nume sau descriere, precum și un tabel în care acestea sunt afișate. Această pagina numită searchPage.xhtml folosește două bean-uri pentru a gestiona detaliile afișate despre produse, respectiv pentru a afișa categoriile de care produse aparțin.

Bean-ul SearchBean este responsabil pentru a lua categoriile produselor și a le afișa într-un combo box:

<f:selectItems value="#{searchBean.getCategoryMap()}"/>

SearchBean este definit ca un bean dependent de sesiune, în care sunt injectate folosind EJB instanțe ale ProductRepo și OrderRepo:

@ManagedBean(name = "searchBean")

@SessionScoped

public class SearchBean implements Serializable {

@EJB

private ProductRepo productRepo;

@EJB

private OrderRepo orderRepo;

//...

public Map<String, Object> getCategoryMap() {...}

}

Metoda getCategoryMap() apelată pentru a popula lista de categorii instanțiază clasa ClientController și apelează o metodă get care întoarce toate categoriile găsite în baza de date.

Similar, ProductBean este definit ca un bean dependent de sesiunea curentă și primește instanțe ale claselor ProductRepo si OrderRepo prin injectare EJB. Acest bean este folosit în pagină pentru a fi apelat în formularul de filtrare a produselor, dar și pentru afișarea produselor în pagină (în cadrul unui tabel cu 10 rânduri) și paginarea rezultatelor, dar și pentru a redirecta clientul la o pagină în care sunt afișate toate detaliile despre produsul selectat. Beanul este definit astfel:

@ManagedBean(name = "productBean")

@SessionScoped

public class ProductBean implements Serializable {

@EJB

private ProductRepo productRepo;

@EJB

private OrderRepo orderRepo;

private List<Product> products = null;

private String keyword;

private String category;

//other methods and fields

}

Pentru a filtra produsele după anumite cuvinte cheie, se completează formularul și se apelează metoda search() cu un keyword și o categorie selectată:

public void search() {

if("".equals(keyword)) {

products = controller.getAllProducts();

} else {

products = controller.getFilteredProducts(keyword);

}

recreateModel();

}

Tot în această pagină sunt apelate metode din clasa Product pentru a se afișa anumite detalii despre produse. De exemplu, pentru a afișa prețul fiecărui produs afișat în tabelul de produse, se apelează metoda getPrice() din clasa Product pentru instanța curentă de produs:

<h:column>

<f:facet name="header">

Product price

</f:facet>

<h:outputText value="#{product.productPrice}"/>

</h:column>

Numele fiecărui produs afișat în tabel este un link către o pagină de detalii (details.xhtml) despre acel produs. Pagina se numește details și folosește două bean-uri pentru afișarea detaliilor despre produs, detailsBean, respectiv pentru formularul de adăugare a produsului la coșul de cumpărături, cartBean. Acestea sunt definite astfel:

@ManagedBean(name = "detailsBean")

@SessionScoped

public class DetailsBean {

static Product product;

//...

}

Bean-ul DetailsBean conține instanța selectată de client a cărei detalii sint afișate în pagină.

@ManagedBean(name = "cartBean")

@SessionScoped

public class CartBean {

private Map<Product, Integer> cart;

//...

}

Bean-ul CartBean este responsabil de gestionarea listei de produse comandate și de afișarea acesteia în pagina corespunzătoare coșului de cumpărături numită cart.xhtml.

Pentru finalizarea comenzii este necesar să fie completate informațiile despre client și adresa de livrare. Aceste informații sunt furnizate prin intermediul formularului din pagina orderDetails.xhtml. Operația este îndeplinită de către bean-ul OrderBean care preia numele clientului, adresa, telefonul și emailul și le transmite clasei controller pentru a fi validate și stocate în baza de date alături de produsele comandate.

@ManagedBean(name = "orderBean")

@SessionScoped

public class OrderBean {

private String name;

private String address;

private String phone;

private String email;

private List<String> order;

private float totalPrice;

@EJB

private ProductRepo productRepo;

@EJB

private OrderRepo orderRepo;

private ClientController controller;

//...

}

Acțiunea de salvare a comenzii este declanșată de butonul următor, din formularul de comandă:

<h:commandButton actionListener="${orderBean.completeOrder()}" action="searchPage" value="Send"/>

Metoda completeOrder() este definită în OrderBean astfel:

public void completeOrder() throws Exception {

if(controller == null) {

controller = new ClientController(productRepo, orderRepo);

}

Order orderInfo = new Order(name, address, totalPrice, new Date());

controller.saveOrder(order, orderInfo);

FacesContext.getCurrentInstance().getExternalContext().invalidateSession();

}

# Testare

Pentru a asigura funcționarea corectă a aplicației, au fost definite cazuri de testare menite să evidențieze eventualele probleme. Pe baza acestora au fost implementate ulterior o serie de teste în clasa TestBean pentru a verifica funcționalitățile principale ale aplicației.

1. Vizualizarea produselor (inclusiv filtrare)

Vizualizarea tuturor produselor este testată în cadrul metodei testGetAllProducts(). Metoda apelează controller-ul pentru a primi produsele existente în baza de date și verifică dacă rezultatele primite au cantitatea strict pozitivă:

// Calls ctrl to return all the products from the db.

// Test passed if: all the products have quantity > 0.

...

List<Product> list = controller.getAllProducts();  
boolean result = true;  
if (list != null && list.size() > 0) {  
 for (Product p : list) {  
 if (p.getProductQuantity() <= 0) result = false;  
 }  
} else result = false;  
  
if (result == true)  
 rez = "Pass";  
else rez = "Failed";

...

Pentru a testa filtrarea produselor după un cuvânt cheie se folosește metoda testGetFilteredProducts(String keyword). Metoda mai verifică dacă primește rezultate corecte pentru filtrarea după anumite cuvinte cheie:

// Calls ctrl to return only the products that contain a given keyword in their name or description.

// Test case 1: filter by empty => test passed if all products are returned

// Test case 2: filter by an existing keyword => test passed if all the products contain the given keyword

// Test case 3: filter by a nonexistent keyword => test passed if nothing is returned

...

List<Product> list = controller.getFilteredProducts(keyword);

if (!keyword.isEmpty()) {

if (keyword.equals("stuff"))

if (list != null && list.size() == 0) rez = "Pass";

else rez = "Failed";

else if (keyword.equals("Laptop"))

if (list != null && list.size() > 0) {

String ok = "Pass";

for (Product p : list) {

if (!(p.getProductName().contains(keyword) || p.getProductDescription().contains(keyword)))

ok = "Failed";

}

rez = ok;

} else rez = "Failed";

} else if (list != null && list.size() > 0) rez = "Pass";

else rez = "Failed";

...

1. Vizualizarea detaliilor unui produs

Vizualizarea detaliilor unui produs se rezumă la operația getById() implementată la nivel de repository, în clasa ProductRepo. Pentru a o testa, s-a implementat metoda testGetProductDetails():

// Requests the product with id 1 and checks to see if the name corresponds with the returned result.

// Expected result: the product with id 1 is returned with all the correct detail fields (name, description, price, quantity)

...

String rez = "";

Product p = productRepo.getProductById(1);

String name = "Laptop ASUS A540LJ-XX612D Procesor Intel® Core™ i3-5005U 2.00GHz, Broadwell™, 15.6\", 4GB, 500GB, DVD-RW, nVIDIA® GeForce® 920M 2GB, Free DOS, Chocolate Black";

if (p.getProductName().equals(name)) rez = "Pass";

else rez = "Failed";

...

1. Vizualizarea coșului de cumpărături

Coșul de cumpărături conține lista produselor comandate, alături de cantitatea în care au fost comandate. Pentru a testa modul în care este returnată lista aceasta, se folosește metoda testViewCart(Product p) care verifică dacă se returnează o listă goală când produsul trimis ca și parametru este null, respectiv dacă se returnează lista cu produsul trimis ca și parametru corect adăugat:

// Requests the contents of the shopping cart

// Test case 1: if the cart is empty, an empty list is returned, no exception is thrown

// Test case 2: if the cart contains something, the complete and correct contents are returned

...

if (p == null)

if (cartBean.getCart().size() != 0) rez = "Failed";

else rez = "Pass";

else {

cartBean.addToCart(p, 1);

if (cartBean.getCart().size() != 1) rez = "Failed";

else rez = "Pass";

}

...

1. Adăugarea unui produs în coșul de cumpărături

Adăugarea unui produs în coșul de cumpărături este influențată de cantitatea de produs comandată. Dacă se comandă un număr ce depășește cantitatea valabilă, sau un număr negativ, sau un caracter care nu este număr, produsul nu se adaugă în coș.

// Adds a product to the shopping cart.

// Test case 1: adds an existing number of products (between 1 and available quantity) => test passed if product is added

// Test case 2: adds a negative or 0 number of products => test fails and nothing is added

// Test case 3: adds more than the available quantity for the product => test fails and nothing is added

// Test case 4: adds a char as the product quantity => test fails and nothing is added

...

cartBean.addToCart(p, quantity);

if (quantity > 0 && quantity <= p.getProductQuantity())

if (cartBean.getCart().size() == 1) rez = "Pass";

else rez = "Failed";

else if (cartBean.getCart().size() != 0) rez = "Failed";

else rez = "Pass";

...

1. Ștergerea unui produs din coșul de cumpărături

Dacă un produs adăugat în coșul de cumpărături nu mai este dorit, acesta se poate șterge din coș. Metoda testDeleteFromCart() testează dacă operația este efectuată corect, asupra produsului corespunzător.

// Deletes a product from the shopping cart.

// Test passed if the selected product and only the selected product is removed from the cart.

...

CartBean cartBean = new CartBean();

Product p = productRepo.getProductById(1);

cartBean.addToCart(p, 1);

cartBean.addToCart(productRepo.getProductById(2), 1);

cartBean.remove(p);

if (cartBean.getCart().size() != 1) rez = "Failed";

else rez = "Pass";

...

1. Finalizarea unei comenzi

Finalizarea unei comenzi constă în introducerea datelor despre client (nume, adresă, telefon, email) și persistarea acestor date în tabela Orders, a produselor comandate în tabela ProductToOrder, precum și în modificarea cantității disponibile pentru produsele comandate. Metoda care testează această funcționalitate primește ca și parametri id-ul produsului, cantitatea dorită, precum și informațiile despre client în obiectul Order.

// Finalize order by filling in client details and saving the order in the database.

// Also updates the available quantity for the ordered products.

// Test case 1: all the user information given is correct => test passed and insert and update operations are completed correctly

// Test case 2: string values are either null, or the floating point value is null or a char=> test fails and the info is requested again

...

try {

controller.saveOrder(Collections.singletonList(productId + "#" + quantity), order);

rez = "Pass";

} catch() {

rez = "Failed";

}

...