## 💡 Projet : Librairie en ligne (Web + Mobile)

### 🛍️ **Fonctionnalités principales**

#### Côté utilisateur :

* 🔐 Inscription / Connexion (JWT)
* 🔎 Rechercher un livre par titre, auteur, ou catégorie
* 📖 Voir les détails d’un livre (titre, description, prix, image)
* ➕ Ajouter un livre au panier
* 🧾 Passer une commande
* 📜 Consulter ses commandes

#### Côté admin :

* ➕ Ajouter / modifier / supprimer des livres
* 📦 Voir les commandes passées
* 🧑‍💼 Gérer les utilisateurs

### 👥 **Utilisateurs**

* **Client** (ROLE\_USER)
* **Admin** (ROLE\_ADMIN)

### 📱 Interfaces

#### Web (Angular)

* Admin : tableau de bord, gestion des livres, commandes
* Client : navigation dans la boutique, panier, commande

#### Mobile (Flutter)

* Même fonctions pour le **client uniquement** :
  + Voir les livres
  + Ajouter au panier
  + Commander

### ⚙️ **Backend - Spring Boot**

* Authentification JWT (Spring Security)
* API REST pour :
  + Auth (login/register)
  + Livres (CRUD admin, GET public)
  + Panier (ajouter, retirer, vider)
  + Commandes (créer, consulter)

### 🧱 **Base de données MySQL (simplifiée)**

sql

CopierModifier

User(id, username, email, password, role)

Book(id, title, author, price, description, imageUrl)

Cart(id, user\_id)

CartItem(id, cart\_id, book\_id, quantity)

Order(id, user\_id, date, total)

OrderItem(id, order\_id, book\_id, quantity, price)

### 🛠️ **Technologies**

| **Partie** | **Stack** |
| --- | --- |
| Frontend | Angular + Angular Material |
| Mobile | Flutter (ou React Native) |
| Backend | Spring Boot + Spring Security + JWT |
| DB | MySQL |
| Hébergement | Render (backend), Firebase (frontend) |

### ✅ Ce que tu vas apprendre :

* Authentification sécurisée avec JWT
* Communication Angular / Flutter avec Spring Boot API
* Création de panier et gestion de commandes
* Déploiement d’apps web et mobile
* Modélisation de base de données relationnelle
* Séparation des rôles (admin/user)

### 🚀 Bonus possibles

* Système de paiement fictif (pas besoin de carte réelle)
* Upload d’image pour les livres
* Gestion des stocks
* Système de recherche + filtres par catégories

## 

## 🧱 1. ****Conception****

## 1. ****Définir les rôles****

* **Admin**
  + Gère les livres (ajout, modification, suppression)
  + Consulte les commandes
  + Gère les utilisateurs (optionnel)
* **Client**
  + S’inscrit / Se connecte
  + Parcourt les livres
  + Ajoute des livres au panier
  + Passe des commandes
  + Consulte ses commandes

## 2. ****Définir les entités principales****

| **Entité** | **Description** | **Principaux attributs clés** |
| --- | --- | --- |
| User | Utilisateur du système | id, username, email, password, role |
| Book | Livre disponible à la vente | id, title, author, price, description, imageUrl, stock |
| Cart | Panier d’un utilisateur | id, user\_id |
| CartItem | Articles dans un panier | id, cart\_id, book\_id, quantity |
| Order | Commande passée par un utilisateur | id, user\_id, date, total |
| OrderItem | Articles dans une commande | id, order\_id, book\_id, quantity, price |

## 3. ****Concevoir le schéma de base de données****

Voici un schéma relationnel simple :

User

----

id (PK)

username

email

password

role (ADMIN, USER)

Book

----

id (PK)

title

author

price

description

imageUrl

stock

Cart

----

id (PK)

user\_id (FK)

CartItem

--------

id (PK)

cart\_id (FK)

book\_id (FK)

quantity

Order

-----

id (PK)

user\_id (FK)

date

total

OrderItem

---------

id (PK)

order\_id (FK)

book\_id (FK)

quantity

price

Relations clés :

* Un **User** a un **Cart** (1-1)
* Un **Cart** contient plusieurs **CartItems** (1-N)
* Un **Order** appartient à un **User** (N-1)
* Un **Order** contient plusieurs **OrderItems** (1-N)
* Chaque **OrderItem** et **CartItem** réfère à un **Book**

## 4. ****Préparer les cas d’utilisation****

### Cas d’utilisation principaux

| **Fonctionnalité** | **Utilisateur** | **Description** |
| --- | --- | --- |
| S’inscrire | Client | Créer un compte utilisateur |
| Se connecter | Client/Admin | Authentification avec JWT |
| Ajouter un livre | Admin | Créer un nouveau livre |
| Modifier/Supprimer un livre | Admin | Mettre à jour ou retirer un livre |
| Voir la liste des livres | Client | Consulter les livres disponibles |
| Ajouter un livre au panier | Client | Ajouter un ou plusieurs exemplaires d’un livre au panier |
| Modifier quantité dans panier | Client | Changer la quantité d’un livre dans le panier |
| Passer commande | Client | Finaliser l’achat, créer une commande |
| Voir l’historique des commandes | Client | Consulter ses commandes passées |
| Voir les commandes | Admin | Visualiser toutes les commandes passées |

## 🔧 2. ****Mise en place du backend (Spring Boot + MySQL)****

## ⚙️ 1. Créer le projet Spring Boot

1. Ouvre **Spring Tool Suite 4**
2. Va dans File > New > Spring Starter Project
3. Configure le projet :
   * **Name** : bookstore-backend
   * **Packaging** : Jar
   * **Java version** : 17 (ou 11 si ton PC est limité)
   * **Dependencies** :
     + Spring Web
     + Spring Data JPA
     + MySQL Driver
     + Spring Security
     + Spring Boot DevTools
4. Clique sur Finish

## 🛢️ 2. Connecter à la base de données MySQL

Dans src/main/resources/application.properties (ou .yml), ajoute :

properties

CopierModifier

spring.datasource.url=jdbc:mysql://localhost:3306/bookstore\_db?useSSL=false&serverTimezone=UTC

spring.datasource.username=root

spring.datasource.password=ton\_mot\_de\_passe

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect

spring.jpa.properties.hibernate.format\_sql=true

Ensuite, crée la base dans **MySQL** :

sql

CopierModifier

CREATE DATABASE bookstore\_db;

## 🧱 3. Créer les entités JPA

Tu vas créer des classes dans le package com.bookstore.model

package com.bookstore.model;

import jakarta.persistence.\*;

import java.util.Set;

@Entity

@Table(name = "users") // évite conflit avec mot réservé SQL "user"

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(nullable=false, unique=true)

private String username;

@Column(nullable=false, unique=true)

private String email;

@Column(nullable=false)

private String password;

@Column(nullable=false)

private String role; // ex: ROLE\_USER, ROLE\_ADMIN

// Relation 1 User => 1 Cart

@OneToOne(mappedBy = "user", cascade = CascadeType.ALL)

private Cart cart;

// Relation 1 User => N Orders

@OneToMany(mappedBy = "user", cascade = CascadeType.ALL)

private Set<Order> orders;

// Getters / Setters (omise ici pour la lisibilité)

// Constructeurs, equals, hashCode, toString (à ajouter)

}

Tu fais pareil pour les autres entités :

* Book.java
* Cart.java
* CartItem.java
* Order.java
* OrderItem.java

| **Classe** | **Rôle principal** |
| --- | --- |
| **User** | Gère les infos utilisateur et les rôles |
| **Book** | Stocke les données sur les livres |
| **Cart** | Contient les livres en cours d’achat |
| **CartItem** | Détaille chaque livre et sa quantité dans le panier |
| **Order** | Représente une commande validée |
| **OrderItem** | Détaille les livres achetés dans une commande |

## 🛒 1. C’est quoi un Cart et un CartItem ?

* **Cart (panier)** : C’est **le panier** du client, où il ajoute les produits **avant** de passer la commande.
* **CartItem (élément du panier)** : C’est **chaque produit** ajouté dans le panier, avec sa **quantité**.

## 🧾 2. C’est quoi une Order et un OrderItem ?

* **Order (commande)** : C’est la commande finale validée par l’utilisateur. Elle représente le **panier confirmé et payé**.
* **OrderItem (élément de commande)** : C’est chaque produit dans la commande avec la quantité, prix, etc.

## 💾 4. Créer les Repositories

Dans com.bookstore.repository, tu ajoutes des interfaces comme :

java

CopierModifier

public interface UserRepository extends JpaRepository<User, Long> {

Optional<User> findByEmail(String email);

}

Pareil pour :

* BookRepository
* CartRepository
* CartItemRepository
* OrderRepository
* OrderItemRepository

## 🔐 5. Authentification avec JWT (Spring Security)

Tu dois :

1. Créer les classes suivantes :
   * JwtUtil : pour générer et valider les tokens
   * UserDetailsServiceImpl
   * JwtFilter
   * SecurityConfig
2. Ajouter un contrôleur d’authentification :
   * /register : créer un nouvel utilisateur
   * /login : renvoyer un JWT

Tu veux que je t’envoie le code de l'authentification complète avec JWT ? (config + contrôleur + utilitaires) ?

## 🔄 6. Développer les Endpoints REST API

### 📌 AuthentificationController :

* POST /register
* POST /login

### 📌 BookController :

* GET /books
* POST /books (admin)
* PUT /books/{id}
* DELETE /books/{id}

### 📌 CartController :

* GET /cart
* POST /cart/add
* DELETE /cart/remove/{itemId}

### 📌 OrderController :

* POST /order
* GET /orders (utilisateur connecté)
* GET /admin/orders (admin)

RQ :  
Pour une relation **One-to-One** entre User et Cart, où **chaque User a un Cart**, et chaque Cart appartient à un seul User, voici comment tu peux coder **les classes modèles** avec JPA.

## Exemple simple d’une relation 1-to-1 bidirectionnelle entre User et Cart

### 1. Classe User.java

@Entity

@Table(name = "users")

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String username;

private String email;

private String password;

private String role;

**// Relation OneToOne inverse (User n’est pas propriétaire)**

**@OneToOne(mappedBy = "user", cascade = CascadeType.ALL)**

private Cart cart;

// getters et setters

public Long getId() { return id; }

public void setId(Long id) { this.id = id; }

// autres getters/setters

public Cart getCart() { return cart; }

public void setCart(Cart cart) {

this.cart = cart;

if(cart != null) {

cart.setUser(this);

}

}

}

### 2. Classe Cart.java

@Entity

@Table(name = "carts")

public class Cart {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

**// Relation OneToOne propriétaire (avec la clé étrangère user\_id)**

**@OneToOne**

**@JoinColumn(name = "user\_id", nullable = false)**

private User user;

// getters et setters

public Long getId() { return id; }

public void setId(Long id) { this.id = id; }

public User getUser() { return user; }

public void setUser(User user) { this.user = user; }

}

## Explications

* **Propriétaire de la relation** : La classe qui possède la clé étrangère en base, ici c’est Cart. C’est pour ça que Cart a l’annotation @JoinColumn.
* **mappedBy** dans User indique que cette classe est côté inverse, la relation est déjà gérée par Cart.
* Le cascade = CascadeType.ALL dans User permet que si tu sauvegardes un utilisateur avec un panier, le panier soit aussi sauvegardé automatiquement.
* La méthode setCart dans User est codée pour assurer la cohérence bidirectionnelle.

### ✅ Raisons pour faire de Cart le propriétaire :

* 💾 **Clé étrangère dans la table cart** : on stocke souvent le user\_id dans cart, car chaque panier appartient à un seul utilisateur. Donc, c’est Cart qui référence User.
* 🧭 **Responsabilité logique** : un panier ne vit que pour un seul utilisateur. Donc on attache le panier à l'utilisateur en mettant la FK dans la table cart.

### 📋 Résultat en base :

* Table cart aura une colonne user\_id (clé étrangère).
* Table user n’a pas besoin de colonne spéciale pour cart.

### ⚖️ Et si on voulait l’inverse ?

On pourrait faire de User le propriétaire, mais :

* Ce serait moins logique.
* On stockerait la clé étrangère dans la table user, ce qui n’est pas pratique si on a d’autres relations 1:1 ou 1:N dans le futur.

### **🔄 Pourquoi cascade = CascadeType.ALL dans une relation entre User et Cart ?**

### ✅ **1. Définition de** cascade

cascade signifie que **les opérations effectuées sur l’entité principale** (ex: User) seront **automatiquement propagées** à l'entité liée (ex: Cart).

### ✅ **2. Que fait** CascadeType.ALL **?**

Cela inclut **tous ces types de cascades** :

| **Type** | **Effet** |
| --- | --- |
| PERSIST | Sauvegarder automatiquement le Cart quand on sauvegarde le User. |
| MERGE | Fusionner les changements du Cart quand on met à jour le User. |
| REMOVE | Supprimer le Cart automatiquement si on supprime le User. |
| REFRESH | Mettre à jour le Cart quand on rafraîchit le User depuis la base. |
| DETACH | Détacher le Cart aussi si on détache le User du contexte de persistence. |

### 🧠 **Pourquoi l’utiliser ici ?**

Parce qu’on considère que :

* Un **panier (Cart) ne vit pas sans son utilisateur** (User).
* S’il y a création d’un User, on veut **automatiquement créer son Cart**.
* S’il y a suppression d’un User, on veut **supprimer son Cart**.

Donc :

@OneToOne(mappedBy = "user", cascade = CascadeType.ALL)

private Cart cart;

→ permet que **toutes les actions sur User soient propagées à Cart**, ce qui simplifie beaucoup la gestion.

### ⚠️ Attention :

* Si tu mets CascadeType.ALL, **supprimer un User supprimera automatiquement le Cart associé**.
* Si ce comportement n’est **pas souhaité**, choisis uniquement les types nécessaires :  
  Par exemple :

cascade = { CascadeType.PERSIST, CascadeType.MERGE }

RQ :  
class cart{  
@OneToMany(mappedBy = "cart", cascade = CascadeType.ALL, orphanRemoval = true)   
**private** Set<CartItem> items;  
}

|  |
| --- |
| **\*)**Set<...> |
| Une **collection** sans doublons (contrairement à une List). | |

### 🔁 Pourquoi Set ?

* On utilise souvent Set quand on ne veut **pas avoir deux fois le même article** (même livre) dans le panier.
* Il garantit **l’unicité** des objets.

## \*) 📌 ****Quand mettre**** orphanRemoval = true ****?****

Tu l’utilises **dans une relation parent-enfant**, quand tu veux que **les enfants soient automatiquement supprimés de la base de données dès qu’ils ne sont plus liés au parent**.

### 🧩 Exemple classique :

* **Un Panier (Cart) contient plusieurs Articles (CartItem).**
* Si tu enlèves un article du panier (cart.getItems().remove(article)), **tu veux que cet article disparaisse aussi de la base**, car il ne sert plus à rien tout seul. (car il n’est plus lié à aucun panier.)

### ✋ Sans orphanRemoval = true

* L’article est simplement “détaché” du panier (plus dans la collection).
* Mais il reste **dans la base** (une ligne orpheline).
* Tu dois le supprimer manuellement (avec delete).

### ✔️ Avec orphanRemoval = true

* L’article est supprimé automatiquement de la base dès qu’il n’est plus dans la collection.
* Tu n’as pas besoin de faire de suppression manuelle.

### RQ : 🔍 @JoinColumn(name = "cart\_id", nullable = false) signifie :

C’est une **annotation de JPA** utilisée pour configurer la **clé étrangère** (foreign key) dans une relation @ManyToOne ou @OneToOne.

### 🧱 Concrètement :

Supposons que tu sois dans la classe CartItem :

@ManyToOne

@JoinColumn(name = "cart\_id", nullable = false)

private Cart cart;

Cela veut dire :

* 🔗 Le champ cart dans CartItem **fait référence à un Cart** (panier).
* 🛠️ Dans la base de données, une colonne **cart\_id** sera créée dans la table cart\_item.
* 🔐 Cette colonne **représente une clé étrangère** pointant vers la table cart.
* 🚫 nullable = false signifie que **cart\_id ne peut pas être NULL** : un CartItem doit **toujours être rattaché à un panier**.

### 📌 Exemple SQL généré :

CREATE TABLE cart\_item (

id BIGINT PRIMARY KEY,

cart\_id BIGINT NOT NULL,

FOREIGN KEY (cart\_id) REFERENCES cart(id)

);

### RQ : 📦 1. **Relation entre** OrderItem **et** Book

@ManyToOne

@JoinColumn(name = "book\_id", nullable = false)

private Book book;

👉 Cela signifie que **chaque OrderItem est lié à un seul Book**.  
Mais **un même livre peut apparaître dans plusieurs OrderItem**, car plusieurs commandes peuvent contenir le même livre.

C’est une relation **ManyToOne** du point de vue de OrderItem → Plusieurs OrderItem peuvent référencer le **même** Book.

### 📦 2. **Relation entre** Order **et** OrderItem

@ManyToOne

@JoinColumn(name = "order\_id", nullable = false)

private Order order;

👉 Cela signifie que chaque OrderItem appartient à une seule Order.  
Mais une Order peut contenir **plusieurs** OrderItem.

### 📘 Donc oui :

✅ Une commande (Order) peut contenir **plusieurs livres**, mais **via** la table OrderItem.  
Chaque OrderItem représente un livre commandé (avec quantité, prix à ce moment-là).

C'est ce qu’on appelle une **relation "many-to-many" avec attributs** (comme quantity, price) décomposée en une entité intermédiaire : OrderItem.