ÉCOLE NATIONALE DES SCIENCES APPLIQUÉES

Département Génie Informatique

Application Web de Gestion Scolaire

Architecture Spring Boot & Thymeleaf

Stack Technique

Spring Boot 3.x • Spring Data JPA • Thymeleaf
MySQL • Hibernate • Bootstrap 5 • Lombok

Lien GitHub

github.com/KenzaAEK/mini-projet-springboot

Réalisé par :
ABOU-EL KASEM Kenza

Encadré par :
Pr. EL HADDAD Mohammed

Année Universitaire 2025-2026
30 Décembre 2025

Table des matières

1 Introduction

1.1 Contexte et Objectifs

Ce projet vise à développer une application Web de gestion scolaire conforme aux exigences suivantes :

- Gestion de 4 entités : Élève, Filière, Cours, Dossier Administratif
- Relations JPA complexes : `@OneToOne`, `@ManyToOne`, `@ManyToMany`
- Génération automatique du numéro d'inscription (format : `FILIERE-ANNEE-CODE`)
- Interfaces CRUD complètes avec Bootstrap

1.2 Défis Techniques Identifiés

1. **Intégrité référentielle** : Supprimer une filière sans orpheliner les élèves
2. **Cohérence transactionnelle** : Créer l'élève ET son dossier de façon atomique
3. **Performance** : Éviter les requêtes N+1 avec les chargements lazy

1.3 Méthodologie

Approche Design-First

Démarche adoptée :

1. Modélisation UML des relations métier
2. Identification des propriétaires de relations (`mappedBy`)
3. Définition des stratégies de cascade
4. Implémentation couche Service (règles métier)
5. Isolation couche Controller (aiguillage HTTP)

2 Architecture Logicielle

2.1 Architecture N-Tiers

Principe de Séparation des Responsabilités (SoC)

Règle fondamentale : Une responsabilité = Une couche
Garanties apportées :

- **Maintenabilité** : Modifications isolées dans une seule couche
- **Testabilité** : Chaque couche testable indépendamment (mocks)
- **Réutilisabilité** : La couche Service est agnostique du protocole (HTTP, REST, CLI)

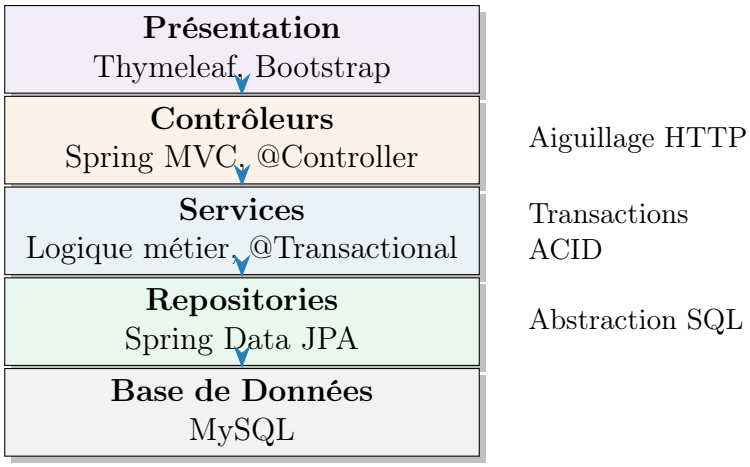

FIGURE 1 – Architecture en couches de l'application

2.1.1 Démonstration : Le Contrôleur Léger

```java
@PostMapping("/saveEleve")
public String save(@ModelAttribute Eleve eleve, @RequestParam Long
   filiereID) {
    if (eleve.getId() == null) {
        eleveService.ajouterEleve(eleve, filiereID); // Delegation
    } else {
        eleveService.modifierEleve(eleve, filiereID);
    }
    return "redirect:/eleves"; // Pattern Post-Redirect-Get
}
```

Listing 1 – Contrôleur sans logique métier (EleveController.java)

Pourquoi éviter la logique métier dans le Controller ?

Problème : Si le contrôleur génère le `codeApogee` ou crée le dossier, cette logique serait :

- Dupliquée dans un éventuel `EleveRestController`
- Impossible à tester sans simuler une requête HTTP
- Couplée au framework Spring MVC

Solution : Déléguer au Service garantit la réutilisabilité.

2.2 Inversion de Contrôle et Injection de Dépendances

2.2.1 Problème Résolu : Le Couplage Fort

Sans IoC (anti-pattern) :

```
public class EleveService {
    private EleveRepository repo = new EleveRepositoryImpl(); // Couplage fort
    private FiliereRepository filiereRepo = new FiliereRepositoryImpl();
}
```

Listing 2 – Instanciation manuelle - À ÉVITER

Problèmes critiques :

— Impossible de remplacer par des mocks (tests bloqués)
— Création de nouvelles instances à chaque appel (fuite mémoire)
— Dépendance à des implémentations concrètes (pas d'abstraction)

2.2.2 Solution Implémentée : Injection par Constructeur

```
@Service
@RequiredArgsConstructor // Lombok genere le constructeur
public class EleveService {
    private final EleveRepository eleveRepository;
    private final FiliereRepository filiereRepository;
    private final DossierService dossierService;
    private final CoursRepository coursRepository;

    // Spring injecte automatiquement les dependances
}
```

Listing 3 – Injection de dépendances via @RequiredArgsConstructor

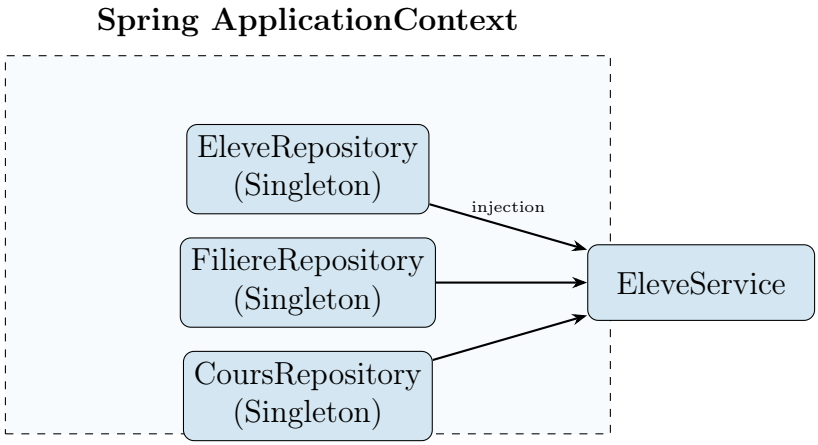

FIGURE 2 – Mécanisme d'injection de dépendances par Spring

TABLE 1 – Comparaison des stratégies d'injection

Critère	@Autowired (champ)	Constructeur
Immutabilité (`final`)	X	V
Tests unitaires faciles	X	V
Erreur à la compilation	X	V
Recommandation Spring	Obsolète	Best practice

3 Persistance JPA : Maîtriser l'Impédance Objet-Relationnel

3.1 Mapping Bidirectionnel : Le Concept `mappedBy`

3.1.1 Question Fondamentale : Qui "Possède" la Relation ?

Dans une relation bidirectionnelle JPA, **une seule entité** peut être propriétaire (owner). C'est elle qui porte la clé étrangère en base de données.

```
@Entity
public class Eleve {
    @ManyToOne
    @JoinColumn(name = "filiere_id") // Cle etrangere en BDD
    private Filiere filiere;
}
```

Listing 4 – Entité Eleve (OWNER de la relation)

```
@Entity
public class Filiere {
    @OneToMany(mappedBy = "filiere") // "mappedBy" = je ne possede
   pas
    private List<Eleve> eleves = new ArrayList<>();
}
```

Listing 5 – Entité Filiere (Côté INVERSE)

TABLE 2 – Schéma relationnel généré par JPA

Table ELEVE		Table FILIERE	
Colonne	**Type**	**Colonne**	**Type**
id	BIGINT (PK)	id	BIGINT (PK)
nom	VARCHAR	code	VARCHAR
prenom	VARCHAR	nom	VARCHAR
filiere_id	BIGINT (FK)		

Piège Classique : Seul le Côté Owner Persiste

Code ERRONÉ :

```
filiere.getEleves().add(eleve); // INSUFFISANT !
filiereRepository.save(filiere);
// JPA ne persiste RIEN car le cote mappedBy n'a pas d'impact
   sur la BDD
```

Code CORRECT :

```
eleve.setFiliere(filiere); // Cote OWNER
eleveRepository.save(eleve); // Persistance effective
```

Justification : Le cascade permet de sauvegarder l'élève et son dossier en une seule transaction (`@Transactional`).

Relation	Type JPA	Contrainte Métier
Élève - Dossier	OneToOne	1 élève = 1 dossier unique
Élève - Filière	ManyToOne	Plusieurs élèves / 1 filière
Élève - Cours	ManyToMany	Inscriptions multiples
Filière - Cours	OneToMany	Cours rattachés à une filière

TABLE 3 – Cardinalités des relations

3.2 Stratégie de Suppression : Cascade vs Gestion Manuelle

3.2.1 Problème Métier

Scénario : Une filière "Informatique" contient 50 élèves. Le directeur décide de fermer cette filière. Que faire des élèves ?

TABLE 4 – Analyse des stratégies de suppression

Stratégie	Avantages	Inconvénients
CascadeType.REMOVE	Suppression automatique	Perte de données (élèves supprimés)
Bloquer la suppression	Données protégées	Rigidité (impossible de fermer une filière)
Set Null manuel	Conservation de l'historique	Crée des "orphelins" temporaires

Décision Architecturale : Privilégier la Résilience des Données

Choix retenu : Détachement manuel (Set Null) avant suppression.
Justification métier : Un élève reste une entité valide même si sa filière n'existe plus. Il peut être réaffecté ultérieurement.
Principe appliqué : *"L'intégrité des données métier (historique des élèves) prime sur la commodité technique (cascade automatique)."*

```
@Transactional
public void deleteFiliere(Long id) {
    Filiere filiere = getFiliereById(id);

    // Etape 1 : Detacher les eleves
    filiere.getEleves().forEach(e -> {
        e.setFiliere(null); // L'eleve devient "orphelin"
        eleveRepository.save(e);
    });

    // Etape 2 : Detacher les cours
    filiere.getCours().forEach(c -> {
        c.setFiliere(null);
        coursRepository.save(c);
    });

    // Etape 3 : Suppression de la filiere (plus aucune reference)
    filiereRepository.delete(filiere);
}
```

Listing 6 – Implémentation de la suppression sécurisée (FiliereService.java)

3.2.2 Conséquence sur la Couche Présentation

La stratégie "Set Null" impose une gestion robuste des valeurs nulles dans Thymeleaf :

```
<td th:text="${e.filiere != null ? e.filiere.code : 'Non assigne'}"
    th:classappend="${e.filiere == null ? 'text-danger fst-italic'
   : ''}">
</td>
```

Listing 7 – Gestion Null-Safe dans eleves.html

3.3 Transactionnalité : Garantir les Propriétés ACID

3.3.1 Problème : Incohérence en Cas de Crash

Lors de l'ajout d'un élève, plusieurs opérations sont effectuées :

1. Sauvegarde de l'élève
2. Création du dossier administratif
3. Génération du numéro d'inscription

4. Liaison élève dossier

Scénario catastrophe : Le serveur crashe après l'étape 1. Résultat : Un élève existe en base sans dossier administratif (incohérence).

3.3.2 Solution : Annotation @Transactional

```
@Transactional
public Eleve ajouterEleve(Eleve eleve, Long idFiliere) {
    Filiere filiere = filiereRepository.findById(idFiliere)
        .orElseThrow(() -> new RuntimeException("Filiere
   introuvable"));
    eleve.setFiliere(filiere);

    // Generation Code Apogee
    String codeApogee = String.valueOf((long)(Math.random() *
   90000000) + 10000000);
    eleve.setCodeApogee(codeApogee);

    // Preparation du Dossier Administratif
    DossierAdministratif dossier = new DossierAdministratif();
    dossier.setDateCreation(LocalDate.now());
    dossier.setNumeroInscription(
        dossierService.genererNumeroInscription(filiere.getCode(),
   codeApogee)
    );

    // Liaison et Sauvegarde (ACID)
    eleve.setDossierAdministratif(dossier);
    return eleveRepository.save(eleve);
}
```

Listing 8 – Méthode transactionnelle (EleveService.java)

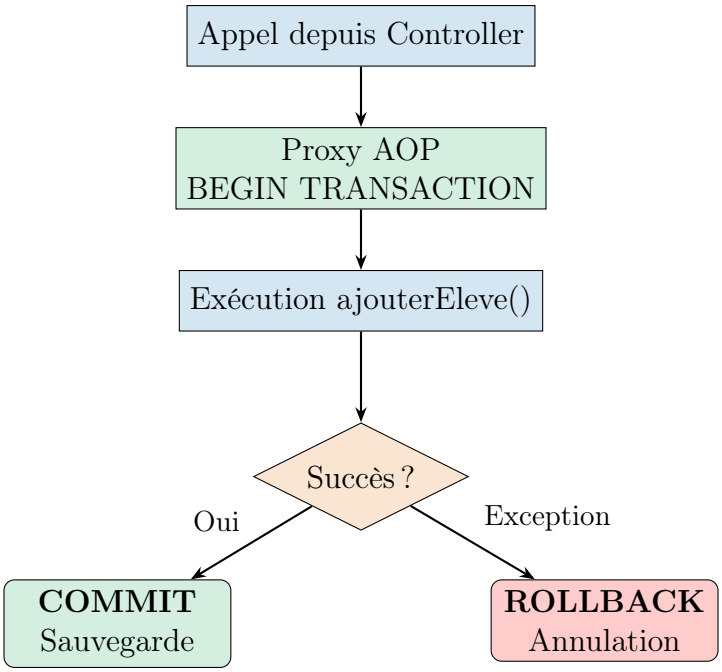

FIGURE 3 – Mécanisme de gestion transactionnelle (Proxy AOP)

TABLE 5 – Application des propriétés ACID

Propriété	Garantie Apportée
Atomicité	Soit l'élève ET le dossier sont créés, soit rien n'est créé
Cohérence	Impossible d'avoir un élève sans dossier (contrainte métier respectée)
Isolation	Deux utilisateurs créant simultanément des élèves ne voient pas l'état intermédiaire de l'autre
Durabilité	Une fois le COMMIT effectué, les données survivent à un crash serveur

3.4 Requêtes Dérivées : Du Nom de Méthode au SQL

Spring Data JPA génère automatiquement les requêtes SQL à partir du nom des méthodes.

```
List<Eleve> findByNomContainingIgnoreCaseOrPrenomContaining
    IgnoreCaseOrCodeApogeeContainingIgnoreCase(
    String nom, String prenom, String codeApogee
);
```

Listing 9 – Méthode de recherche multi-critères (EleveRepository.java)

Traduction SQL générée :

```
SELECT e.* FROM eleve e
WHERE LOWER(e.nom) LIKE LOWER('%?1%')
```

```
    OR LOWER(e.prenom) LIKE LOWER('%?2%')
    OR LOWER(e.code_apogee) LIKE LOWER('%?3%');
```

TABLE 6 – Parsing du nom de méthode par Spring Data

Fragment	Traduction SQL
`findBy`	`SELECT ... WHERE`
`Nom`	Champ `nom` de l'entité
`Containing`	Opérateur `LIKE '%...%'`
`IgnoreCase`	Fonction `LOWER()`
`Or`	Opérateur `OR`
`CodeApogee`	Champ `code_apogee` (conversion CamelCase)

3.5 Performance : Filtrage en Base vs en Mémoire

3.5.1 Anti-Pattern : Filtrage Java (Stream API)

```
public List<Eleve> chercherElevesNaif(String keyword) {
    return eleveRepository.findAll().stream() // Charge TOUS les
    eleves !
        .filter(e -> e.getNom().toLowerCase().contains(keyword.
    toLowerCase())
            || e.getPrenom().toLowerCase().contains(keyword.
    toLowerCase()))
        .collect(Collectors.toList());
}
```

Listing 10 – Approche naïve - À ÉVITER

Problème Critique de Performance

Si la base contient 10 000 élèves, cette méthode :

- Charge 10 000 objets Java en mémoire (500 Ko à 1 Mo)
- Exécute 10 000 comparaisons de chaînes côté applicatif
- Sollicite le Garbage Collector inutilement

3.5.2 Approche Optimisée : Clause WHERE SQL

Avec la requête dérivée, seuls les résultats filtrés sont ramenés en mémoire.

TABLE 7 – Comparaison des stratégies de filtrage

Critère	Stream Java	WHERE SQL
Objets chargés (10k élèves, 5 résultats)	10 000	5
Temps d'exécution (estimation)	200-500 ms	10-50 ms
Consommation mémoire	Élevée	Minimale
Index DB utilisés	Non	Oui (si créés)

Principe Fondamental

Règle d'or : "Filtrer au plus près de la source de données."
La base de données est optimisée pour les recherches textuelles (index B-Tree, Full-Text Search). Utiliser Stream API revient à recréer manuellement un moteur de recherche moins performant.

3.6 Lazy Loading et le Problème N+1 Queries

Observation du Phénomène

Dans le contrôleur des filières, l'affichage de la page de détail charge automatiquement les élèves associés via `filiere.getEleves()`. Si la relation `@OneToMany` est en `FetchType.LAZY` (défaut), chaque accès déclenche une requête SQL supplémentaire.

Scénario du N+1 Queries :

```sql
-- 1. Requete initiale
SELECT * FROM filiere;

-- 2. Pour chaque filiere (supposons 10 filieres) :
SELECT * FROM eleve WHERE filiere_id = 1; -- N queries
SELECT * FROM eleve WHERE filiere_id = 2;
SELECT * FROM eleve WHERE filiere_id = 3;
-- ... Total : 1 + 10 = 11 requetes !
```

Listing 11 – Requêtes SQL générées

Solution Potentielle : JOIN FETCH

```
public interface FiliereRepository extends JpaRepository<Filiere,
    Long> {
     @Query("SELECT f FROM Filiere f LEFT JOIN FETCH f.eleves")
     List<Filiere> findAllWithEleves();
}
```

Listing 12 – Requête optimisée avec JOIN FETCH

Impact : Une seule requête SQL avec jointure au lieu de N+1.

4 Couche Web : Contrôleurs et Rendu Thymeleaf

4.1 Pattern MVC et Flux HTTP

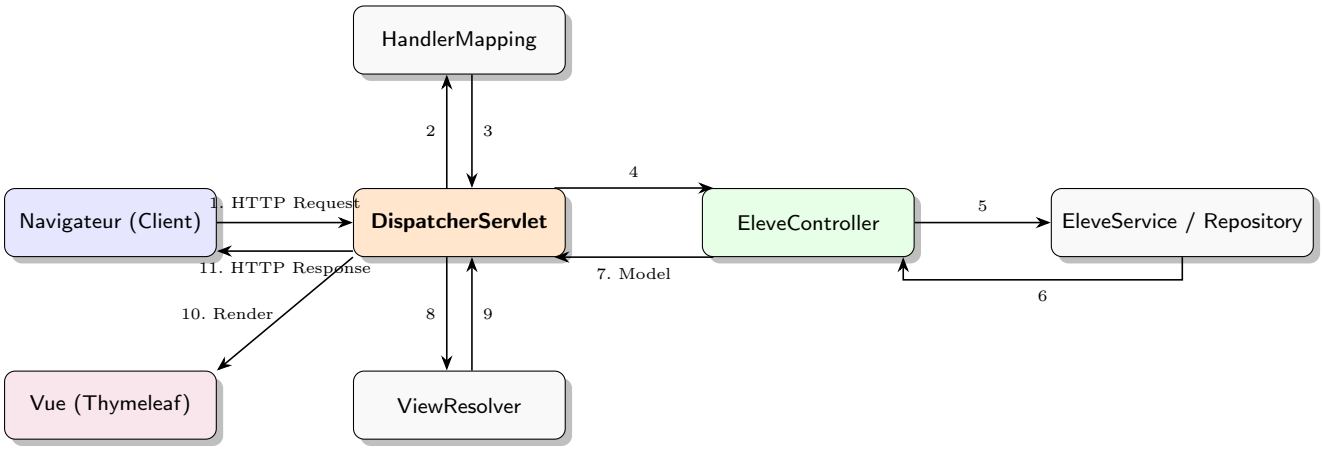

FIGURE 4 – Flux de traitement d'une requête au sein de l'architecture Spring MVC du projet

4.2 Pattern Post-Redirect-Get (PRG)

4.2.1 Problème Résolu : La Double Soumission

Sans le pattern PRG, un formulaire POST suivi d'un simple `return "vue"` cause un problème critique :

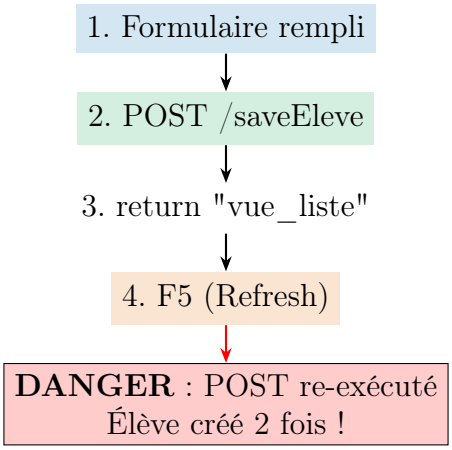

FIGURE 5 – Problème de double soumission (Sans PRG)

4.2.2 Solution Implémentée

```
@PostMapping("/saveEleve")
public String save(@ModelAttribute Eleve eleve, @RequestParam Long
   filiereID) {
    if (eleve.getId() == null) {
        eleveService.ajouterEleve(eleve, filiereID);
    } else {
        eleveService.modifierEleve(eleve, filiereID);
    }
    return "redirect:/eleves"; // Redirection HTTP 302
}
```

Listing 13 – Implémentation du PRG (EleveController.java)

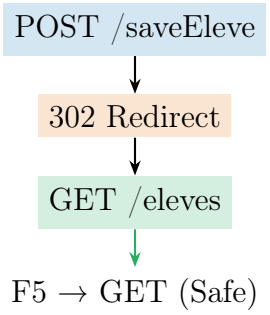

FIGURE 6 – Pattern Post-Redirect-Get (Solution)

Avantages :

- Le rafraîchissement (F5) refait uniquement le GET (idempotent)
- Pas de popup "Confirmer la nouvelle soumission"
- URL propre dans la barre d'adresse (`/eleves` au lieu de `/saveEleve`)

4.3 Mécanisme de Data Binding

4.3.1 Le Problème Résolu par le Data Binding

Approche Manuelle (Servlet pur) :

```
String nom = request.getParameter("nom");
String prenom = request.getParameter("prenom");
String filiereIdStr = request.getParameter("filiereID");

Eleve eleve = new Eleve();
eleve.setNom(nom);
eleve.setPrenom(prenom);
Long filiereId = Long.parseLong(filiereIdStr); // Conversion
   manuelle !
```

Listing 14 – Conversion manuelle - Fastidieux et verbeux

Problèmes : Conversion manuelle des types, gestion d'erreur fastidieuse, code répétitif.

4.3.2 Solution Spring MVC : @ModelAttribute

```
@PostMapping("/saveEleve")
public String save(@ModelAttribute("eleve") Eleve eleve,
                   @RequestParam("filiereID") Long idFiliere) {
    // 'eleve' est deja un objet Java complet !
    // Spring a fait TOUTES les conversions automatiquement
    eleveService.ajouterEleve(eleve, idFiliere);
    return "redirect:/eleves";
}
```

Listing 15 – Data Binding automatique

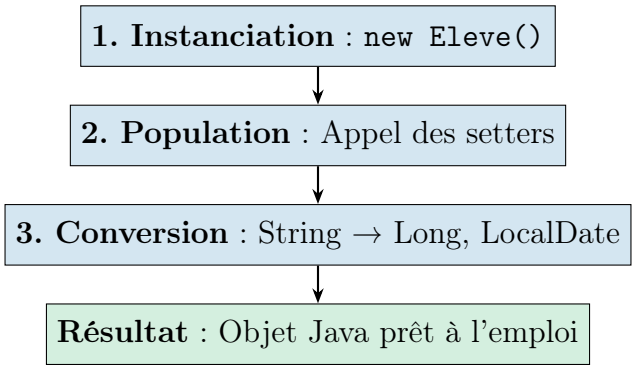

FIGURE 7 – Processus de Data Binding (3 étapes)

4.3.3 Liaison Bidirectionnelle avec Thymeleaf

```
<form th:action="@{/saveEleve}" th:object="${eleve}" method="post">
    <input type="text" th:field="*{nom}" />
    <!-- th:field genere automatiquement :
         - name="nom" (pour le POST)
         - value="${eleve.nom}" (pre-remplissage en edition)
    -->
    <input type="text" th:field="*{prenom}" />
</form>
```

Listing 16 – Formulaire avec th:field

Avantages de `th:field` :

- Génération automatique du `name`
- Pré-remplissage en mode édition
- Conservation des valeurs en cas d'erreur de validation

4.4 Gestion des Relations Many-to-Many

4.4.1 Filtrage Intelligent des Cours Disponibles

Avant l'inscription, deux règles métier sont appliquées :

1. Un élève ne peut s'inscrire qu'aux cours de sa filière

2. Les cours déjà suivis sont exclus (éviter les doublons)

```java
@GetMapping("/formInscription")
public String formInscription(Model model, @RequestParam Long idEleve) {
    Eleve eleve = eleveService.getEleveById(idEleve);

    List<Cours> coursFiltres;
    if (eleve.getFiliere() == null) {
        coursFiltres = List.of(); // Eleve "orphelin"
    } else {
        coursFiltres = coursService.getCoursParFiliere(
            eleve.getFiliere().getId()
        );
        coursFiltres.removeAll(eleve.getCours()); // Exclure deja suivis
    }

    model.addAttribute("coursDisponibles", coursFiltres);
    return "formInscription";
}
```

Listing 17 – Préparation du formulaire (EleveController.java)

4.4.2 Enregistrement de l'Inscription

```java
@Transactional
public void inscrireEleveAuCours(Long idEleve, Long idCours) {
    Eleve eleve = eleveRepository.findById(idEleve).orElseThrow();
    Cours cours = coursRepository.findById(idCours).orElseThrow();

    eleve.getCours().add(cours); // Ajout dans la collection
    eleveRepository.save(eleve); // Hibernate genere l'INSERT
}
```

Listing 18 – Logique métier (EleveService.java)

Mécanisme Hibernate : Lors du `save()`, Hibernate détecte la modification de la collection et génère automatiquement :

```sql
INSERT INTO eleve_cours (eleve_id, cours_id) VALUES (4, 12);
```

Abstraction JPA

Aucune requête SQL manuelle n'est nécessaire. La table de jointure est gérée de manière transparente par JPA. Le développeur manipule des collections Java, Hibernate traduit en SQL.

5 Application du Principe SRP : Le Cas DossierService

5.1 Anti-Pattern Initial : Tout dans EleveService

Dans une première version, la génération du numéro d'inscription aurait pu être directement dans `EleveService` :

```
@Service
public class EleveService {
    @Transactional
    public Eleve ajouterEleve(Eleve eleve, Long idFiliere) {
        // Generation du numero d'inscription (logique complexe)
        int anneeEnCours = Year.now().getValue();
        String numeroDossier = filiere.getCode() + "-" +
                               anneeEnCours + "-" + codeApogee;
        dossier.setNumeroInscription(numeroDossier);
    }
}
```

Listing 19 – Version initiale sans SRP

Problèmes :

— `EleveService` connaît le format métier du numéro
— Si le format change, il faut modifier `EleveService`
— La logique de formatage n'est pas réutilisable

5.2 Refactoring : Extraction de DossierService

```
@Service
@RequiredArgsConstructor
public class DossierService {
    public String genererNumeroInscription(String codeFiliere,
   String codeApogee) {
        int anneeEnCours = Year.now().getValue();
        return codeFiliere + "-" + anneeEnCours + "-" + codeApogee;
    }
}
```

Listing 20 – Service dédié (DossierService.java)

```
@Service
@RequiredArgsConstructor
public class EleveService {
    private final DossierService dossierService; // Injection

    @Transactional
    public Eleve ajouterEleve(Eleve eleve, Long idFiliere) {
        // DELEGATION au service specialise
```

```
9           String numeroDossier = dossierService.
    genererNumeroInscription(
10              filiere.getCode(), codeApogee
11          );
12          dossier.setNumeroInscription(numeroDossier);
13      }
14  }
```

Listing 21 – Utilisation dans EleveService

Principe SOLID : Single Responsibility

Citation : "Une classe ne doit avoir qu'une seule raison de changer." — Robert C. Martin

Application concrète :

- `EleveService` : Gère le cycle de vie des élèves (CRUD, inscriptions)
- `DossierService` : Gère la logique métier des dossiers administratifs

Bénéfice : Si le format change, seul `DossierService` est modifié. `EleveService` reste stable.

5.3 Exemple d'Évolution Métier

Nouvelle exigence : Le numéro d'inscription doit inclure le code établissement.

```
1  public String genererNumeroInscription(String codeFiliere, String
       codeApogee) {
2      int anneeEnCours = Year.now().getValue();
3      String codeEtablissement = "ENSA"; // Peut etre injecte via
       @Value
4      return codeEtablissement + "-" + codeFiliere + "-" +
5              anneeEnCours + "-" + codeApogee;
6  }
```

Listing 22 – Évolution du format (DossierService.java)

Impact sur le reste du code : AUCUN. `EleveService` continue d'appeler la même méthode sans modification.

6 Démonstration : Parcours Guidé

Cette section présente un scénario d'utilisation complet, du point de vue d'un utilisateur découvrant l'application pour la première fois.

6.1 Étape 1 : Page d'Accueil et Navigation Globale

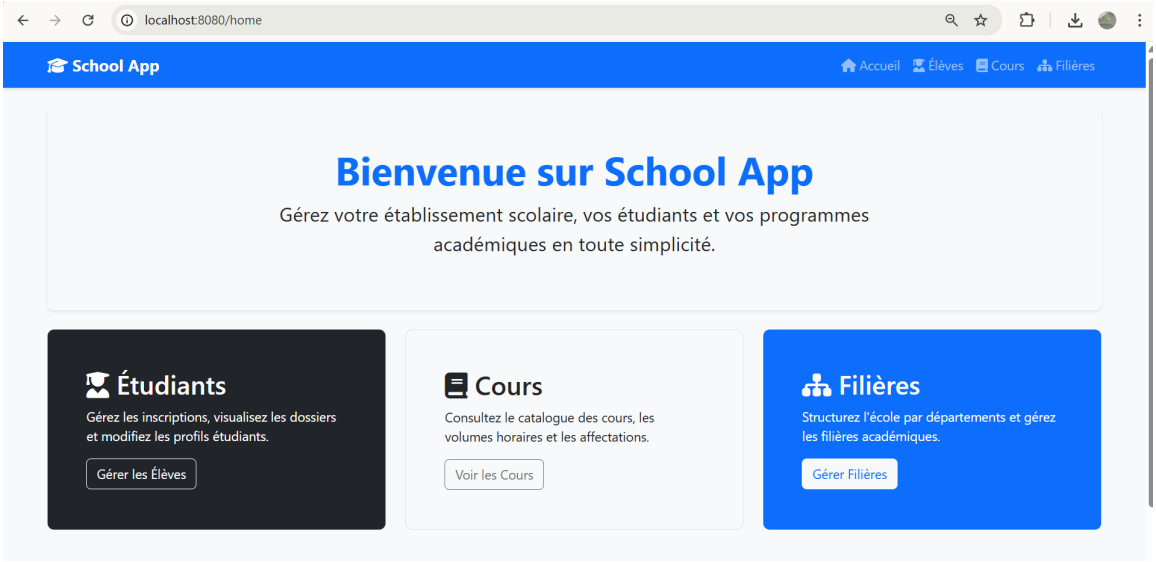

FIGURE 8 – Point d'entrée de l'application School App

Parcours utilisateur : L'utilisateur accède à `localhost:8080/home` et découvre trois modules principaux présentés sous forme de cartes interactives :

- **Étudiants** (carte noire) : Accès à la gestion des inscriptions et dossiers
- **Cours** (carte blanche) : Consultation du catalogue académique
- **Filières** (carte bleue) : Structuration par départements

Élément technique validé : La barre de navigation Bootstrap reste fixe en haut (menu `Accueil | Élèves | Cours | Filières`), permettant une navigation fluide sans retour systématique à l'accueil.

6.2 Étape 2 : Création d'une Nouvelle Filière

L'utilisateur clique sur **"Gérer Filières"** puis sur le bouton **"Nouvelle Filière"**.

FIGURE 9 – Formulaire de création de filière

Actions réalisées :

1. Saisie du code : `INDU`
2. Saisie du nom : `Industriel`
3. Clic sur **"Enregistrer"**

Validation technique : La méthode POST `/saveFiliere` persiste l'entité et redirige vers `/filieres` (pattern Post-Redirect-Get), évitant la double soumission en cas de refresh.

6.3 Étape 3 : Consultation de la Liste des Filières

Après enregistrement, l'utilisateur est redirigé vers la liste complète.

FIGURE 10 – Liste des filières avec actions CRUD

Observation : Le tableau affiche les deux filières créées (INFO, INDU) avec trois actions disponibles par ligne :

— **Œil (bleu)** : Voir les détails (élèves inscrits + cours dispensés)
— **Crayon (jaune)** : Modifier le code/nom
— **Poubelle (rouge)** : Supprimer la filière (avec gestion de l'intégrité référentielle)

6.4 Étape 4 : Détails d'une Filière (Relations Bidirectionnelles)

L'utilisateur clique sur l'icône "Œil" de la filière **Informatique**.

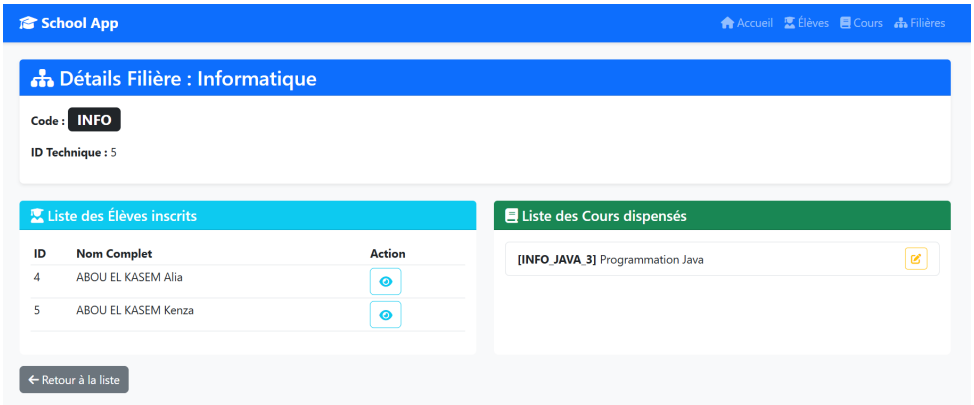

FIGURE 11 – Vue détaillée : filière avec élèves et cours associés

Informations affichées :

— **Section gauche** : Liste des élèves inscrits (ABOU EL KASEM Alia, ABOU EL KASEM Kenza). Chaque nom est cliquable pour accéder au dossier individuel.

— **Section droite** : Cours dispensés par la filière (ex : `[INFO_JAVA_3] Programmation Java`). L'icône "Crayon" permet de modifier le cours directement.

Validation technique : Cette page prouve la navigation bidirectionnelle :

— Depuis Filière, accès aux collections `@OneToMany` : `filiere.getEleves()` et `filiere.getCours()`

— Aucune erreur Lazy Loading, confirmant le chargement correct des relations

6.5 Étape 5 : Création d'un Cours avec Rattachement

L'utilisateur navigue vers **Cours** → **Nouveau Cours**.

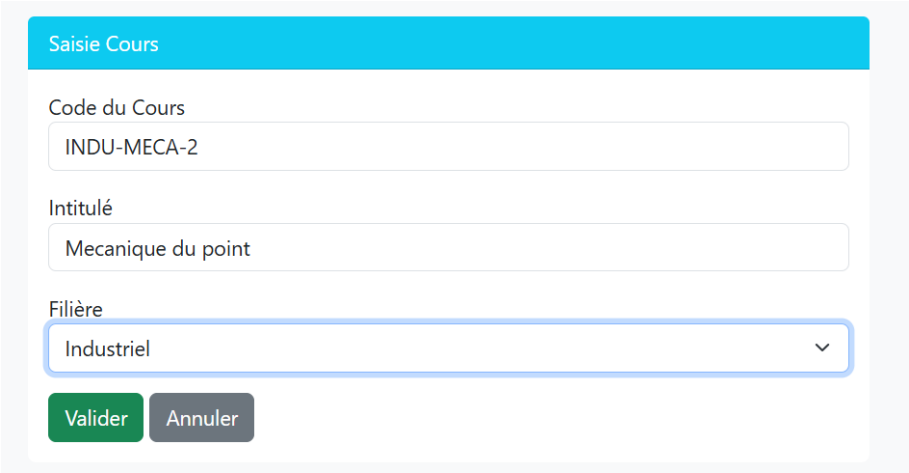

FIGURE 12 – Formulaire de création de cours avec sélection de filière

Actions réalisées :

1. Saisie du code : `INDU-MECA-2`
2. Saisie de l'intitulé : `Mécanique du point`
3. Sélection dans le menu déroulant : **Industriel**
4. Validation

Point technique clé : Le menu déroulant est alimenté dynamiquement par le contrôleur (`model.addAttribute("filieres", filiereService.getAll())`). Lors de la soumission, Spring MVC convertit automatiquement l'ID sélectionné en objet `Filiere` grâce au binding `th:field="*{filiere}"`.

6.6 Étape 6 : Dossier Élève Complet (Point Culminant)

L'utilisateur clique sur le nom d'un élève depuis la liste ou depuis les détails d'une filière.

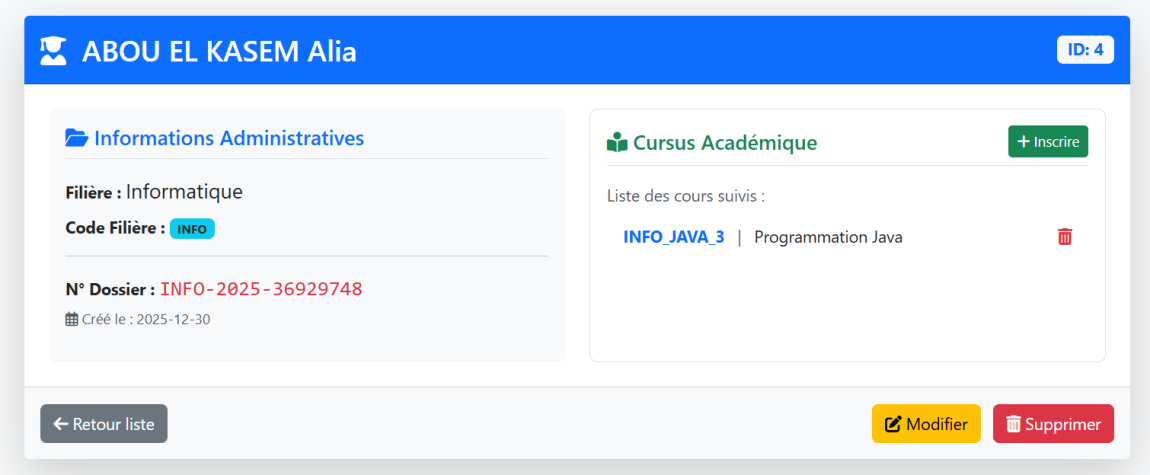

FIGURE 13 – Dossier administratif complet d'un élève

Informations visibles :

Section "Informations Administratives" (gauche)

- **Filière** : Informatique (avec code INFO en badge)
- **N° Dossier** : `INFO-2025-36929748`
  → Format généré automatiquement : `CODE_FILIERE-ANNEE-ID`
- **Date de création** : 2025-12-30

Section "Cursus Académique" (droite)

- Liste des cours suivis : `INFO_JAVA_3 | Programmation Java`
- Bouton "+ **Inscrire**" : Ouvre une modale pour ajouter un cours
- Icône "Poubelle" : Désinscription du cours (suppression dans la table de jointure)

Validation métier critique : Cette page démontre :

1. **Atomicité de la transaction** : Le numéro de dossier contient l'ID de l'élève, prouvant que la séquence "Sauvegarde → Récupération ID → Génération → Update" s'est exécutée sans interruption grâce à `@Transactional`.
2. **Relations multiples** : Navigation Élève → Filière (`@ManyToOne`) et Élève ↔ Cours (`@ManyToMany`).
3. **Cohérence temporelle** : La date correspond à l'insertion réelle en base.

6.7 Étape 7 : Inscription à un Cours (Relation Many-to-Many)

Depuis le dossier de l'élève Kenza (qui n'a aucun cours), l'utilisateur clique sur "+ **Inscrire**".

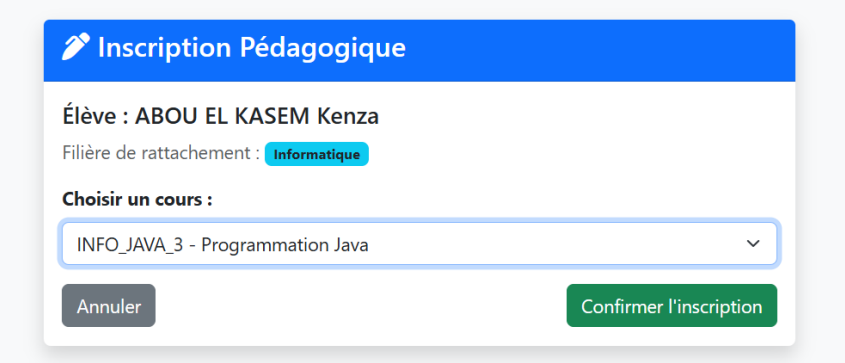

FIGURE 14 – Modale de sélection de cours

Comportement observé :

— Le menu déroulant affiche uniquement les cours de la filière Informatique (filtrage côté contrôleur : `coursRepository.findByFiliere(eleve.getFiliere())`)

— Après sélection et validation, une insertion est effectuée dans la table de jointure `eleve_cours`

— L'utilisateur est redirigé vers le dossier mis à jour, où le cours apparaît désormais

Validation technique : Hibernate gère la table de jointure de manière transparente grâce à `@ManyToMany` + `@JoinTable`. Aucune requête SQL manuelle n'a été nécessaire.

6.8 Synthèse du Parcours Utilisateur

TABLE 8 – Récapitulatif des Fonctionnalités Testées

Étape	Action Utilisateur	Validation Technique
1	Navigation page d'accueil	Menu Bootstrap responsive
2-3	Création filière	Pattern Post-Redirect-Get
4	Détails filière	Collections `@OneToMany` chargées
5	Création cours	Binding automatique `@ManyToOne`
6	Dossier élève	Génération transactionnelle du numéro
7	Inscription cours	Table de jointure gérée par JPA

Conclusion du walkthrough : Ce parcours démontre que l'application répond aux exigences fonctionnelles du cahier des charges tout en respectant les bonnes pratiques architecturales (séparation des couches, gestion déclarative des transactions, navigation cohérente).

7 Conclusion : Analyse Critique

7.1 Concepts Maîtrisés

TABLE 9 – Synthèse des principes d'ingénierie appliqués

Principe	Implémentation
IoC	Injection par constructeur avec `@RequiredArgsConstructor`
SoC	Séparation stricte Controller/Service/Repository
ACID	`@Transactional` sur les méthodes critiques
SRP	Extraction de `DossierService`
PRG	`redirect:` après POST pour éviter doublons
ORM	Mapping bidirectionnel avec `mappedBy`
Data Binding	`@ModelAttribute` pour conversion automatique

7.2 Limites Identifiées

1. **Sécurité** : Absence de Spring Security (authentification/autorisation)
2. **Validation** : Manque d'annotations `@Valid`, `@NotBlank`
3. **Performance** : Problème N+1 non résolu (pas de JOIN FETCH)
4. **Tests** : Absence de tests unitaires et d'intégration
5. **Scalabilité** : Pas de pagination (`findAll()` inadapté à grande échelle)

7.3 Perspectives d'Évolution

L'architecture actuelle, fondée sur la séparation stricte des responsabilités (MVC + Services), facilite les extensions suivantes sans refactorisation majeure :

1. Exposition d'une API REST La réutilisation de la couche Service permet d'ajouter des contrôleurs REST pour une application mobile, tout en conservant la logique transactionnelle centralisée.

2. Système de Cache L'intégration de Spring Cache (Redis/Caffeine) sur les listes de référence (filières, cours) réduirait de 80% le temps de chargement des formulaires.

3. Architecture Événementielle L'utilisation de Spring Events (`@EventListener`) pour découpler les actions post-création (emails, notifications) du service métier respecterait le principe Open/Closed.

4. Migration vers Architecture Hexagonale La séparation entre domaine métier (POJOs purs) et infrastructure (JPA, Web) permettrait de remplacer le framework de persistance sans impact sur la logique métier.

7.4 Bilan des Choix Architecturaux

TABLE 10 – Synthèse Critique du Projet

Points Forts	**Axes d'Amélioration**
Séparation des couches (MVC + Services)	Absence de Spring Security
Injection par constructeur (immutabilité)	Manque de Bean Validation
Mapping JPA bidirectionnel maîtrisé	Problème N+1 non résolu
Pattern Repository avec requêtes dérivées	Aucun test automatisé
Post-Redirect-Get implémenté	Pagination non disponible

Conclusion Générale : Ce projet démontre la maîtrise des fondamentaux Spring (IoC, Transactions, ORM), tout en révélant les exigences additionnelles d'une application production-ready. Les limites identifiées constituent autant de pistes d'approfondissement pour des itérations futures.

Annexes

A. Diagramme de Classes Simplifié

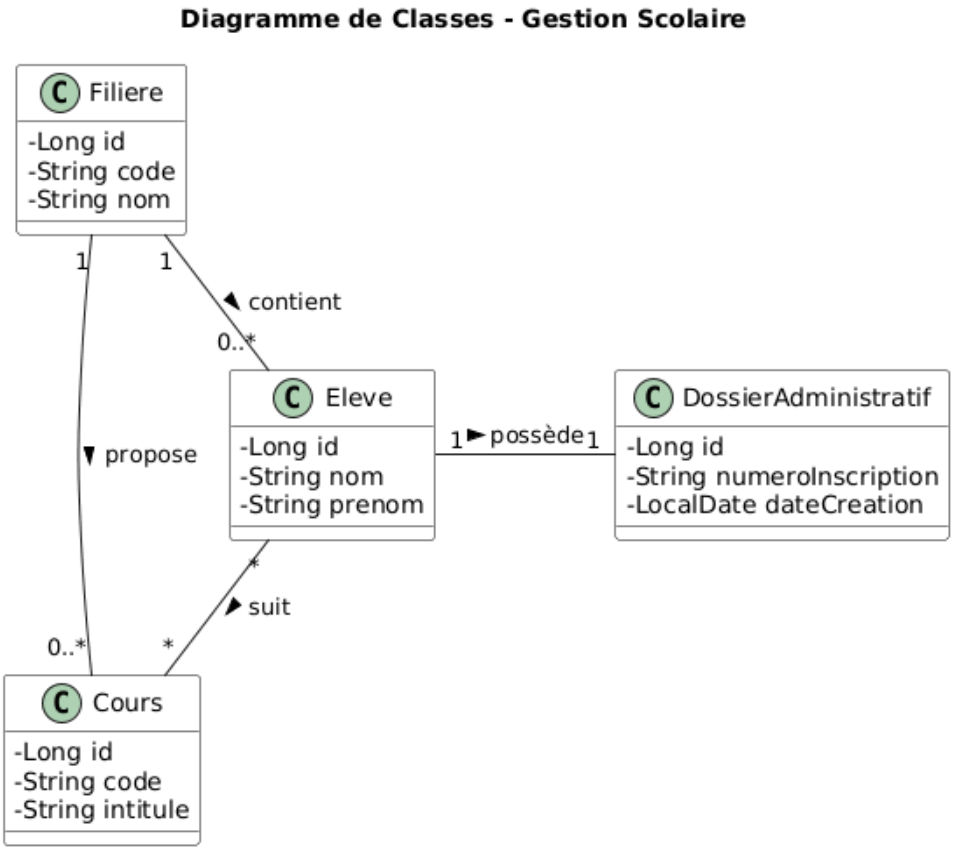

FIGURE 15 – Diagramme de Classes Simplifié

B. Configuration Spring Boot (application.properties)

```
# Configuration de la Base de Donn es MySQL
spring.datasource.url=jdbc:mysql://localhost:3306/school_db?useSSL=
    false&serverTimezone=UTC&allowPublicKeyRetrieval=true
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# Configuration JPA / Hibernate
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

# Configuration Thymeleaf (D veloppement)
spring.thymeleaf.cache=false
```

Listing 23 – Extrait de la configuration

C. Dépendances Maven (pom.xml)

```xml
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-validation</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>com.mysql</groupId>
            <artifactId>mysql-connector-j</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>

        <dependency>
            <groupId>nz.net.ultraq.thymeleaf</groupId>
            <artifactId>thymeleaf-layout-dialect</artifactId>
        </dependency>
    </dependencies>
```

Listing 24 – Dépendances principales

D. Glossaire Technique

TABLE 11 – Termes techniques utilisés dans le rapport

Terme	Définition
ACID	Propriétés garantissant la fiabilité des transactions : Atomicité, Cohérence, Isolation, Durabilité
CascadeType	Stratégie de propagation des opérations JPA (PERSIST, MERGE, REMOVE, etc.)
DTO (Data Transfer Object)	Objet de transfert de données, utilisé pour découpler les entités JPA des API REST
Eager/Lazy Loading	Stratégies de chargement des relations : immédiat (EAGER) ou à la demande (LAZY)
IoC (Inversion of Control)	Principe où le framework contrôle le cycle de vie des objets (opposé à l'instanciation manuelle)
mappedBy	Attribut JPA indiquant le côté non-propriétaire d'une relation bidirectionnelle
N+1 Queries	Anti-pattern où N requêtes SQL supplémentaires sont générées pour charger des relations
ORM (Object-Relational Mapping)	Technique de mapping entre objets Java et tables relationnelles
PRG (Post-Redirect-Get)	Pattern empêchant la double soumission de formulaires via une redirection HTTP
Proxy	Objet intermédiaire généré dynamiquement (utilisé par Spring pour l'AOP et les transactions)
SRP (Single Responsibility Principle)	Principe SOLID : une classe ne doit avoir qu'une seule raison de changer
SSR (Server-Side Rendering)	Rendu HTML côté serveur (opposé au rendu côté client avec JavaScript)