

Application Web de Gestion Scolaire : Maîtrise de l'Écosystème Spring Boot

Une exploration des architectures robustes et des patterns de persistance avancés.

Réalisé par : ABOU-EL KASEM Kenza

Encadré par : Pr. EL HADDAD Mohammed
École Nationale des Sciences Appliquées

Le Défi : Modéliser un Système Scolaire Complet et Fiable

Objectifs

- Gestion de 4 entités : Élève, Filière, Cours, Dossier Administratif.
- Implémentation de relations JPA complexes : `@OneToOne`, `@ManyToOne`, `@ManyToMany`.
- Logique métier spécifique : Génération automatique du numéro d'inscription (format : FILIERE-ANNEE-CODE).
- Développement d'interfaces CRUD complètes et réactives.

Arsenal Technique

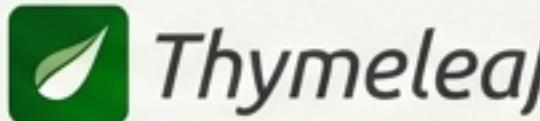


Spring boot



HIBERNATE

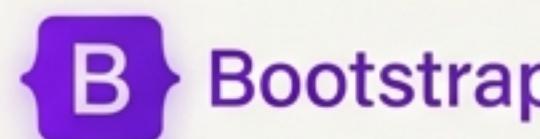
Spring Data JPA



Thymeleaf



MySQL

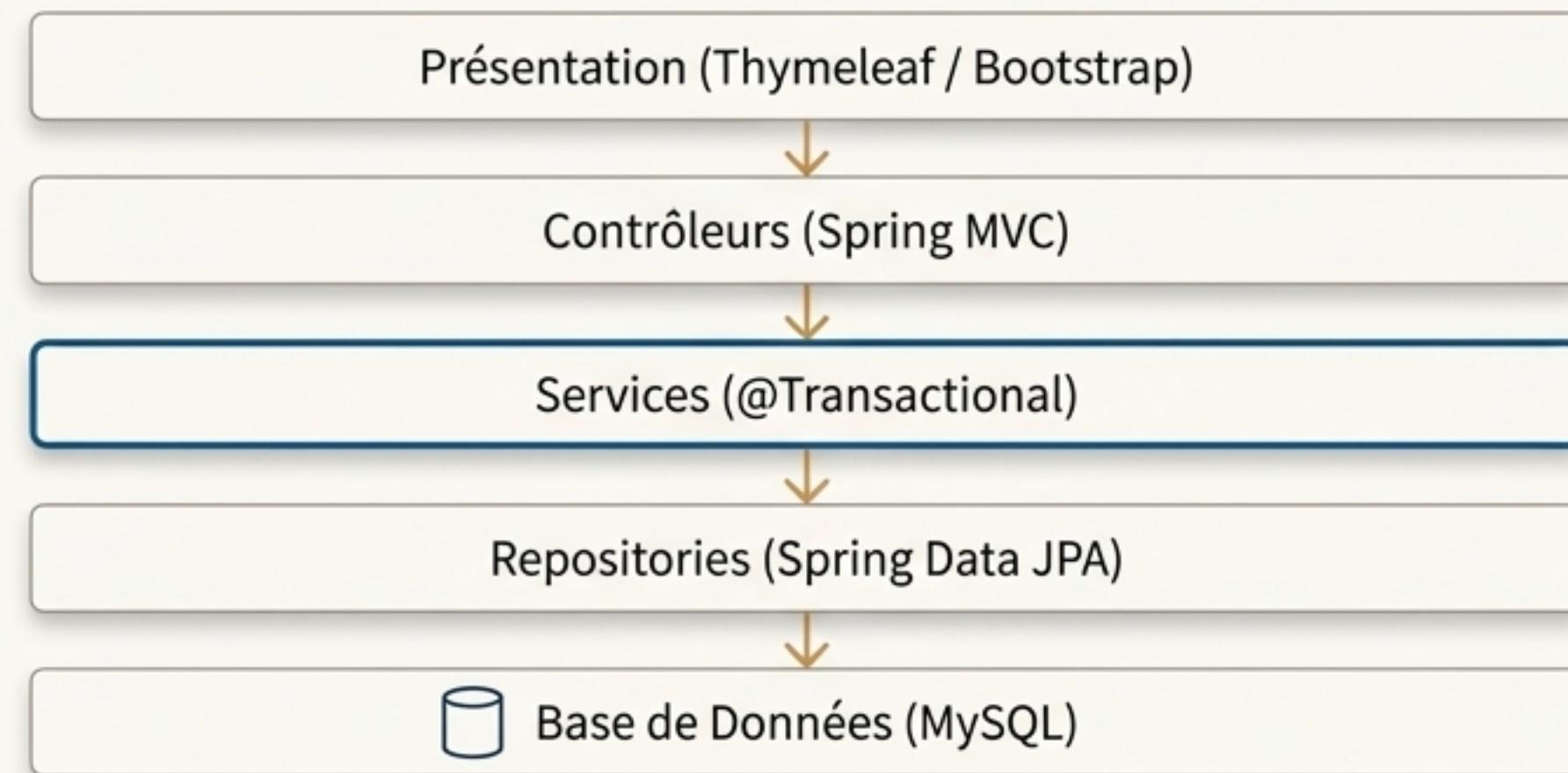


Bootstrap



Lombok

Le Plan Directeur : La Séparation des Responsabilités (SoC)



Section des Bénéfices



Maintenabilité

Les modifications sont isolées dans une seule couche, ce qui empêche les régressions en cascade.



Testabilité

Chaque couche peut être testée indépendamment en utilisant des mocks, garantissant la robustesse.



Réutilisabilité

La couche Service est agnostique du protocole (HTTP, REST, etc.) et contient la logique métier pure.

Vaincre le Couplage Fort avec l'Inversion de Contrôle (IoC)

✗ Code Rigide & Intestable

```
// Anti-pattern : Couplage Fort
public class EleveService {
    private EleveRepository repo = new
EleveRepositoryImpl();
    private FiliereRepository filiereRepo = new
FiliereRepositoryImpl();
    // ...
}
```

- ✗ Impossible de mocker
- ✗ Fuites mémoire potentielles
- ✗ Dépendance à l'implémentation

✓ Code Flexible & Prêt pour les Tests

```
// Solution : Injection par constructeur
@Service
@RequiredArgsConstructor
public class EleveService {
    private final EleveRepository eleveRepository;
    private final FiliereRepository
filiereRepository;
    // ... Spring injecte les dépendances
}
```

- ✓ Immutabilité (`final`)
- ✓ Tests unitaires faciles
- ✓ Erreur à la compilation si une dépendance manque

Garantir l'Atomicité des Opérations avec `@Transactional`

Le Scénario Catastrophe

Un élève est ajouté, mais le serveur crashe avant la création de son dossier administratif.



Le Résultat: Une donnée orpheline. L'élève existe en base, mais son dossier est manquant, créant une incohérence critique dans le système.

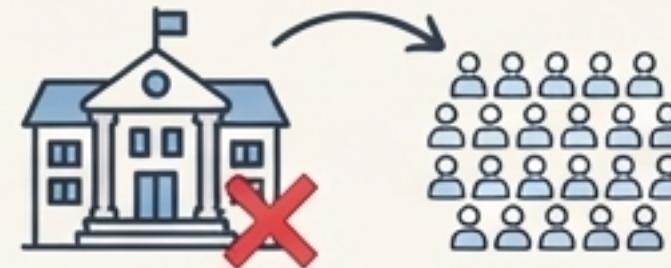
La Solution Élégante

```
@Transactional  
public Eleve ajouterEleve(Eleve eleve, Long idFiliere) {  
    // 1. Lier l'élève à sa filière  
    // 2. Générer le code Apogée  
    // 3. Créer le Dossier Administratif  
    // 4. Lier et sauvegarder l'élève et le dossier  
    return eleveRepository.save(eleve);  
}
```

Tout ou Rien.

L'opération réussit entièrement, ou elle est entièrement annulée en cas d'erreur. Les propriétés ACID sont garanties.

Le Dilemme de la Suppression : "Cascade" vs. Logique Métier



Le Problème Métier

Que faire des 50 élèves d'une filière 'Informatique' que l'on décide de fermer ?

Analyse des Options

`CascadeType.REMOVE`



Simple, mais supprime définitivement les 50 dossiers élèves.

****Risque :** Perte de données historiques.**

Bloquer la Suppression



Sécurisé, mais rigide.
La filière ne peut jamais être fermée.

Détachement Manuel (`Set Null`)



L'élève devient 'orphelin' mais son historique est préservé pour une réaffectation future.

Décision Architecturale

"L'intégrité des données métier (historique des élèves) prime sur la commodité technique (cascade automatique). Nous avons opté pour un détachement manuel avant suppression."

Qui 'Possède' la Relation ? Le Rôle Clé de `mappedBy`

Dans une relation bidirectionnelle JPA, une seule entité est responsable de la colonne de jointure en base de données. `mappedBy` délègue cette responsabilité.



Le Piège à Éviter

Code ERRONÉ

```
filiere.getEleves().add(eleve);
filiereRepository.save(filiere);
```

✗ Ne fonctionne pas. Seul le côté propriétaire de la relation peut persister la clé étrangère.

Code CORRECT

```
eleve.setFiliere(filiere);
eleveRepository.save(eleve);
```

✓ La modification doit être faite sur l'entité qui possède la relation (ici, 'Eleve').

L'Anti-Pattern N+1 et le Filtrage Inefficace

1 Problème 1 : Filtrage en Mémoire (Anti-Pattern)

- Le Scénario: "Une recherche doit trouver 5 élèves parmi 10 000 enregistrements."
- L'approche naïve:

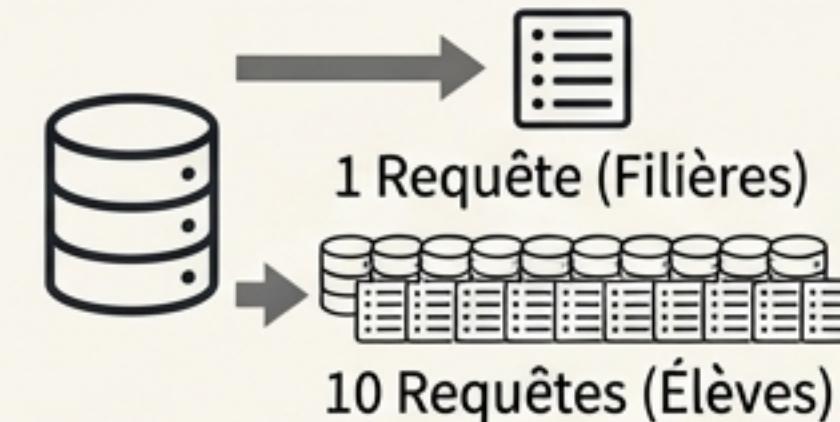
```
eleveRepository.findAll().stream().filter(...)
```



Charge **10 000** objets Java en mémoire pour n'en garder que **5**. Inefficace et non scalable.

2 Problème 2 : Le Problème N+1 Queries

- Le Scénario: "Afficher une liste de 10 filières et le nombre d'élèves dans chacune."
- L'approche naïve: "Une requête pour les filières, puis 10 requêtes pour les élèves de chaque filière."



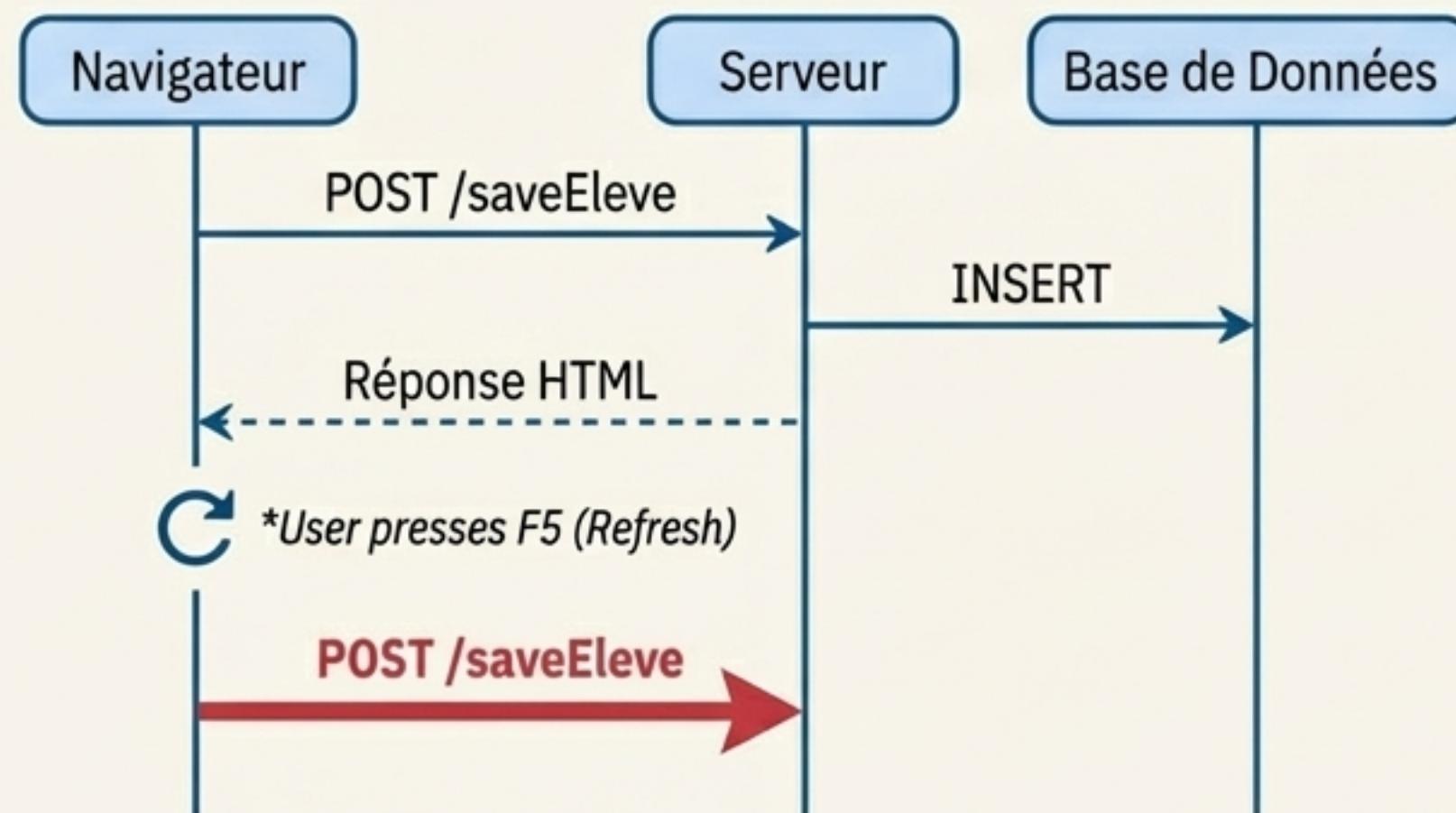
11 requêtes SQL pour une seule vue. Inacceptable.

Filtrer à la source

Les requêtes dérivées de Spring Data JPA (`findByNomContaining...`) et les requêtes `@Query` avec `JOIN FETCH` interrogent la base de données intelligemment, ne retournant que les données strictement nécessaires.

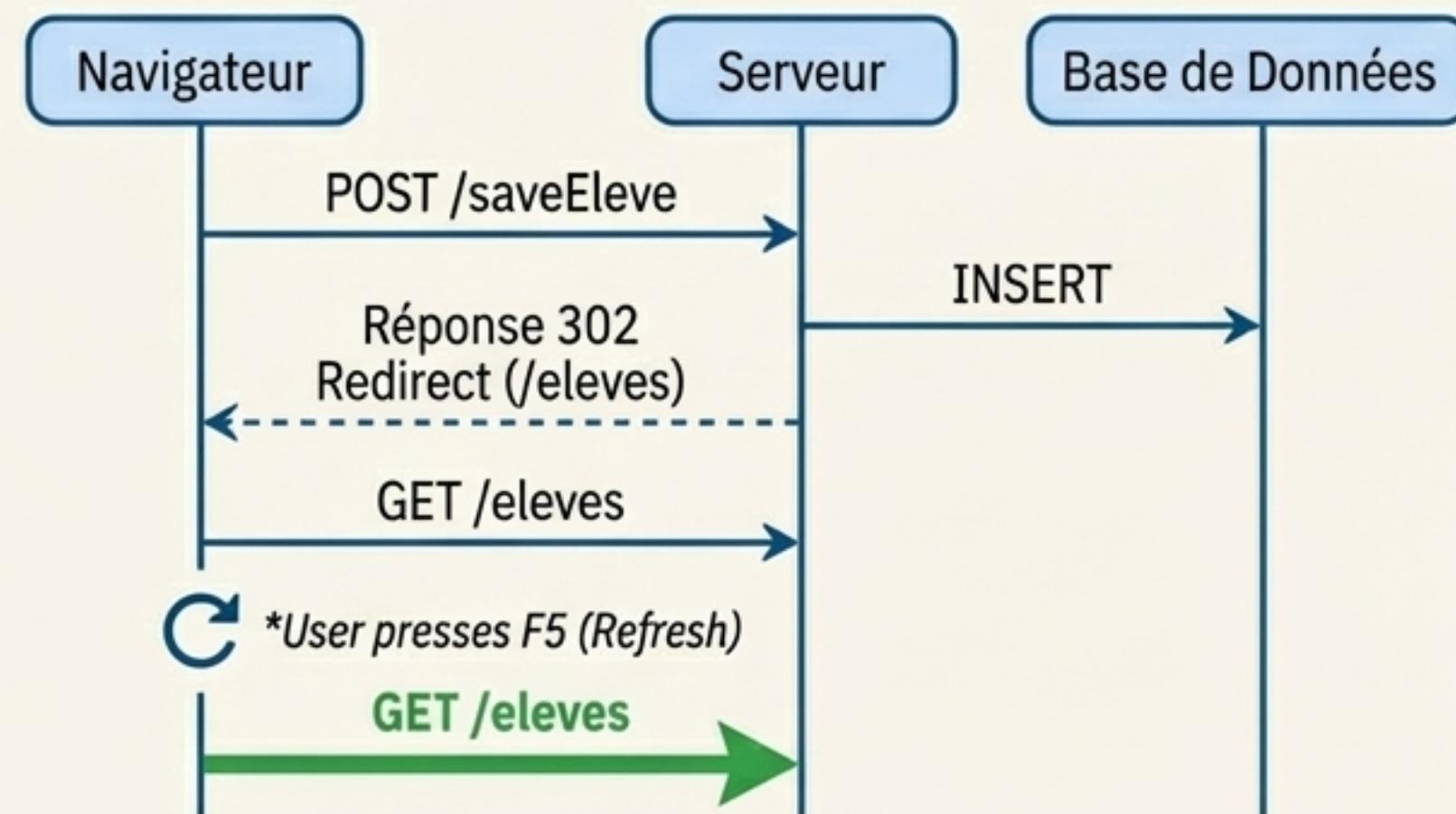
Éliminer la Double Soumission avec le Pattern Post-Redirect-Get (PRG)

✗ Sans PRG



DANGER : Le formulaire est soumis une seconde fois, créant un doublon.

✓ Avec PRG

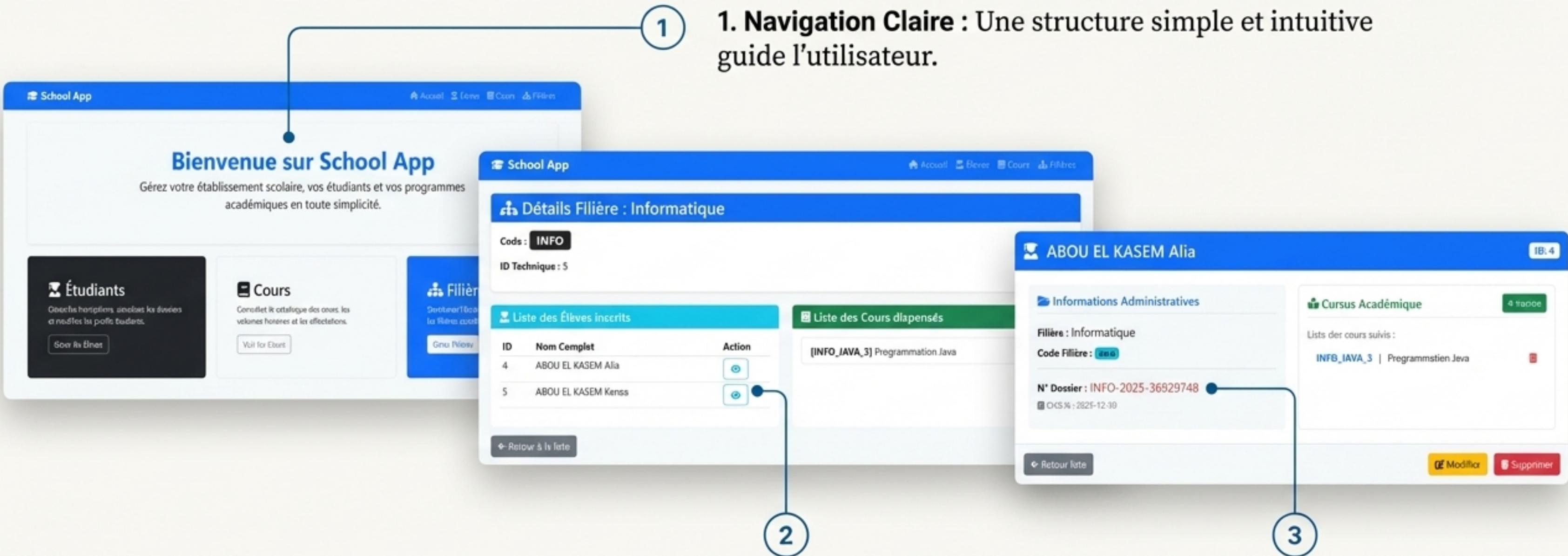


SÉCURISÉ : Le rafraîchissement recharge la page de liste sans resoumettre les données.

```
return "redirect:/eleves";
```

Une seule ligne pour une expérience utilisateur robuste et prévisible.

De l'Architecture à l'Expérience : Le Parcours Utilisateur



1. Navigation Claire : Une structure simple et intuitive guide l'utilisateur.

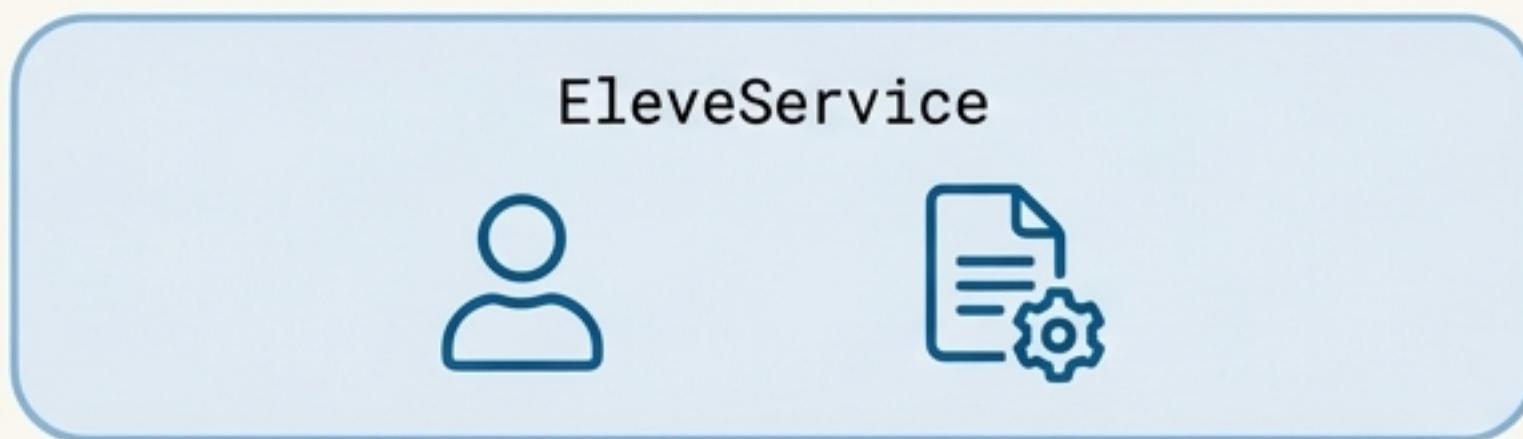
2. Relations Bidirectionnelles
Visibles : `mappedBy` permet de naviguer facilement de la filière vers ses élèves et cours.

3. Dossier Unifié : Le N° de dossier est généré et sauvegardé de manière atomique avec l'élève grâce à `@Transactional`.

Un Code Maintenable : Le Principe de Responsabilité Unique (SRP)

Le Cas d'Étude : Le refactoring de la génération du numéro de dossier.

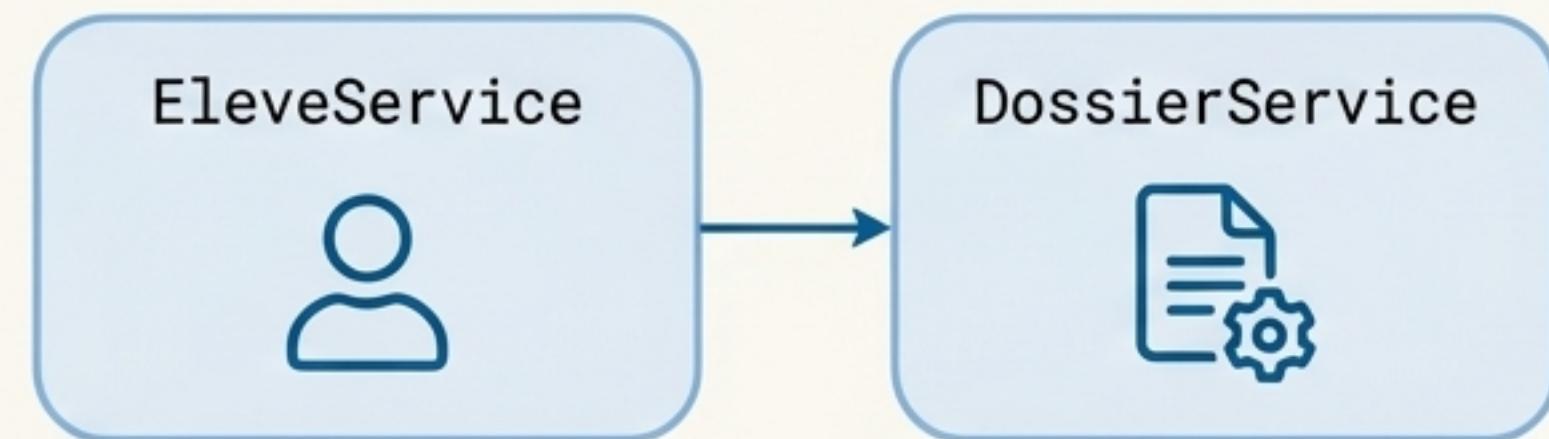
Avant (Logique Couplée)



‘EleveService’ est responsable de la logique des élèves ET du formatage du numéro de dossier.

✗ Problème: Si la règle de formatage change, ‘EleveService’ doit être modifié, même si la logique de gestion des élèves n'a pas changé.

Après (Logique Découplée)



‘EleveService’ délègue la génération du numéro à ‘DossierService’, un expert dédié à cette seule tâche.

✓ Bénéfice: La modification du format n'impacte que ‘DossierService’. ‘EleveService’ reste stable et non modifié.

“Une classe ne doit avoir qu'une seule raison de changer.”

— Robert C. Martin

Bilan et Analyse Critique : Une Fondation Solide

Points Forts

-  **SoC Maîtrisé** : Séparation stricte des couches Controller, Service, et Repository.
-  **Injection de Dépendances** : Utilisation systématique de l'injection par constructeur pour un code testable.
-  **Gestion des Transactions** : Garantie de la cohérence des données avec `@Transactional`.
-  **Patterns Web Robustes** : Implémentation du pattern Post-Redirect-Get pour une UX fiable.
-  **ORM Avancé** : Maîtrise des concepts `mappedBy` et des stratégies de suppression.

Axes d'Amélioration

-  **Sécurité** : Absence de Spring Security pour l'authentification et les autorisations.
-  **Validation des Données** : Manque de validation des entrées avec les annotations `@Valid`.
-  **Tests Automatisés** : Absence d'une suite de tests unitaires (JUnit/Mockito) et d'intégration.
-  **Performance à Grande Échelle** : Problème N+1 non résolu globalement et absence de pagination.

Perspectives d'Évolution : Bâtir sur des Bases Saines

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



Exposer une API REST

La couche Service, contenant toute la logique métier, est immédiatement réutilisable pour ajouter des RestController pour une application mobile.

Intégrer un Cache

L'ajout de Spring Cache (@Cacheable) sur les services retournant des données de référence (filières, cours) peut drastiquement améliorer la performance.

Découpler avec des Événements Asynchrones

Des actions comme l'envoi d'emails de confirmation peuvent être découplées du flux principal en utilisant Spring Eveing Events (@EventListener).

Conclusion et Ressources

Ce projet démontre la construction d'une application web robuste et maintenable grâce à une application rigoureuse des principes d'architecture et des patterns de l'écosystème Spring. La fondation est saine, prête pour des évolutions futures.

Explorez le code source sur GitHub :
github.com/KenzaAEK/mini-projet-springboot

