Couche Contrôleur

December 22, 2024

1 Comprendre la Couche Contrôleur et les Tests d'Intégration dans Spring Boot

1.1 Introduction à la Couche Contrôleur

La *couche contrôleur* est une partie essentielle d'une application Spring Boot. Elle gère les requêtes HTTP des clients et les associe aux services ou à la logique métier appropriés. Grâce aux contrôleurs, nous définissons les *endpoints* de notre API REST, permettant aux clients d'effectuer des opérations comme la création, la lecture, la mise à jour ou la suppression de ressources.

1.1.1 Rôle de la Couche Contrôleur

- Agit comme une *passerelle* vers l'application.
- Transforme les requêtes client en appels de service appropriés.
- Gère à la fois les données entrantes (corps des requêtes) et les réponses sortantes (renvoyées au client).
- Garantit que l'application respecte les principes de l'architecture RESTful.

1.1.2 Annotations Clés dans les Contrôleurs

- `@RestController` : Marque une classe comme un contrôleur où chaque méthode renvoie un corps de réponse.
- `@RequestMapping` : Associe des requêtes HTTP à des méthodes ou des classes spécifiques.
- `@PostMapping`, `@GetMapping`, `@PutMapping`, `@DeleteMapping` : Associent des méthodes HTTP (POST, GET, PUT, DELETE) à des opérations spécifiques.
- `@RequestBody` : Lie le corps d'une requête client à un paramètre de méthode.
- `@PathVariable` : Extrait une valeur de l'URI pour l'utiliser dans la méthode.
- `@Autowired` : Injecte des dépendances comme des services dans le contrôleur.

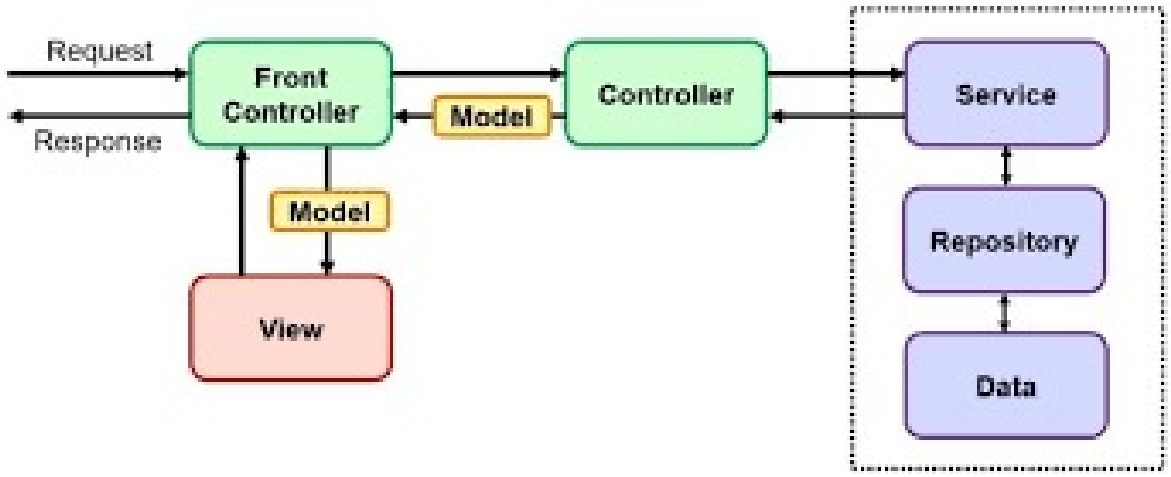

Figure 1: Diagramme de la Couche Contrôleur interagissant avec d'autres composants tels que les Services, les Référentiels et les Clients.

2 Les Contrôleurs que Nous Avons Construits

2.1 Contrôleur Auteur

Objectif : Gérer les auteurs avec des détails comme `nom` et `âge`.

- **Endpoints** :
  - **Créer** : `POST /api/authors/save` - Ajoute un nouvel auteur au système.
  - **Lire** :
    - `GET /api/authors/all` : Récupère tous les auteurs.
    - `GET /api/authors/one/{id}` : Récupère un auteur spécifique par ID.
  - **Mettre à jour** : `PUT /api/authors/update/{id}` - Modifie les détails d'un auteur.
  - **Supprimer** : `DELETE /api/authors/delete/{id}` - Supprime un auteur par ID.

Exemple de Code : Contrôleur Auteur L'exemple suivant montre l'implémentation d'une couche contrôleur (Contrôleur Auteur). Il contient les annotations et toutes les fonctionnalités requises pour cette couche.

Le code complet est disponible dans le dépôt GitHub du projet :

`https://github.com/DOesNOtM1tter/spring_boot_project-/blob/master/supporting_project/supporting_project/src/main/java/com/example/demo/controllers/AuthorControlle` `java`.

Explication :

- La classe est annotée avec `@RestController`, indiquant qu'elle gère les requêtes HTTP et fournit des réponses au format JSON.

- L'annotation `@PostMapping("/save")` spécifie l'endpoint pour enregistrer un auteur.
- `AuthorService` est injecté en utilisant `@Autowired` pour gérer la logique métier.
- La méthode `create()` prend un objet `AuthorDto`, appelle la couche service pour l'enregistrer, et renvoie une réponse avec le statut `201 CREATED`.

2.2 Contrôleur Livre

Objectif : Gérer les livres avec des attributs comme `titre`, `auteur`, `catégorie` et `date de publication`.

- **Endpoints** :
  - **Créer** : `POST /api/books/save` - Ajoute un nouveau livre.
  - **Lire** :
    - `GET /api/books/all` : Récupère tous les livres.
    - `GET /api/books/one/{id}` : Récupère un livre spécifique par ID.
  - **Mettre à jour** : `PUT /api/books/update/{id}` - Met à jour les détails d'un livre.
  - **Supprimer** : `DELETE /api/books/delete/{id}` - Supprime un livre.

Exemple de Code : Tests d'intégration du Contrôleur Livre

L'exemple suivant démontre l'implémentation d'un test d'intégration pour la couche contrôleur. Il inclut les annotations nécessaires ainsi que toute la logique JUnit et MockMvc requise pour cette couche.

Le code complet est disponible sur le dépôt GitHub du projet :

`https://github.com/D0esN0tM1tter/spring_boot_project-/blob/master/supporting_project/supporting_project/src/test/java/com/example/demo/services/BookServicesIntegr` `java`.

3 Contrôleur Catégorie

Objectif : Gérer les catégories qui classifient les livres.

- **Points de terminaison** :
  - **Créer** : `POST /api/category/save` - Ajoute une nouvelle catégorie.
  - **Lire** :
    - `GET /api/category/all` : Liste toutes les catégories.
    - `GET /api/category/one/{id}` : Récupère une catégorie spécifique par ID.
  - **Mettre à jour** : `PUT /api/category/update/{id}` - Met à jour les détails d'une catégorie.
  - **Supprimer** : `DELETE /api/category/delete/{id}` - Supprime une catégorie.

4 Tests d'Intégration : MockMvc et Transformation JSON-DTO-Entity

4.1 Pourquoi les tests sont importants

Les tests permettent de vérifier que :

- Chaque contrôleur gère les requêtes comme prévu.
- Les données circulent correctement entre les contrôleurs et les services.
- Le système répond correctement aux erreurs ou aux entrées invalides.

4.2 Options de tests et pourquoi nous avons choisi MockMvc

Lors du développement d'une application web, plusieurs options de tests sont disponibles. Par exemple, des tests peuvent être effectués côté serveur avec des technologies telles que JPA, Thymeleaf, ou en utilisant des frameworks frontend comme Angular pour tester l'interaction entre le client et le serveur.

4.3 Approches possibles pour les tests

- **Tests côté serveur (JPA, Thymeleaf)** :
  - Les tests côté serveur impliquent de tester la logique métier dans le backend. Par exemple, JPA (Java Persistence API) peut être utilisé pour tester les interactions avec la base de données, en s'assurant que les entités sont correctement persistées et récupérées.
  - Thymeleaf, intégré avec Tomcat, peut être utilisé pour tester la génération côté serveur de vues en utilisant des modèles HTML dynamiques. Ces tests vérifient que les données sont correctement injectées dans les pages HTML avant leur rendu.
- **Tests avec des frameworks Front-End (Angular)** :
  - Un autre type de test consiste à tester les interactions entre le frontend et le backend. Des frameworks comme Angular permettent de simuler des requêtes HTTP et de vérifier que le serveur répond correctement aux actions des utilisateurs. Ces tests vérifient non seulement l'intégration du frontend avec le backend mais aussi l'interface utilisateur et les interactions avec l'API.

4.4 Pourquoi nous avons choisi MockMvc

Bien que d'autres méthodes de test puissent être utilisées pour tester l'application de bout en bout, nous avons choisi d'utiliser `MockMvc` pour nos tests d'intégration. Cette approche présente plusieurs avantages, notamment :

- **Test simplifié du Backend** : `MockMvc` nous permet de simuler des requêtes HTTP directement sur le contrôleur sans nécessiter un serveur réel comme Tomcat. Cela permet de tester l'intégration des couches contrôleur et service sans avoir besoin d'une configuration complète de serveur ou d'un client frontend comme Angular.

- **Vérification de la logique métier** : Avec `MockMvc`, nous pouvons vérifier que la logique métier, le routage des requêtes et la transformation des données (via les DTO, entités, etc.) fonctionnent comme prévu, tout en évitant les complexités supplémentaires liées à la configuration d'un serveur web complet ou d'un framework frontend.
- **Isolation et concentration sur l'API** : Cette méthode permet de tester uniquement l'API, isolant le backend sans avoir besoin d'un client frontend. Cela rend les tests plus rapides et plus ciblés, en nous concentrant sur les composants essentiels côté serveur.

En résumé, bien que des alternatives telles que JPA, Thymeleaf ou des tests avec Angular existent pour tester les interactions entre différents composants, l'utilisation de `MockMvc` nous permet de tester efficacement le backend en isolation et de nous concentrer sur la logique backend, tout en simplifiant le processus de test.

4.5 Rôle de MockMvc

4.5.1 Qu'est-ce que MockMvc ?

MockMvc est un outil de Spring qui permet de tester des contrôleurs sans démarrer un serveur. Il simule des requêtes HTTP et fournit des réponses comme dans une véritable interaction client-serveur.

4.5.2 Pourquoi MockMvc ?

- Évite la nécessité d'un serveur en fonctionnement.
- Simule le processus de requête et réponse HTTP.
- Idéal pour les tests d'intégration puisqu'il garantit que le contrôleur interagit correctement avec la couche de service et les autres composants.

4.5.3 Étapes pour tester avec MockMvc :

- Configurer MockMvc avec `@AutoConfigureMockMvc` dans votre classe de test.
- Utiliser MockMvc pour simuler des requêtes HTTP (par exemple, POST, GET).
- Vérifier les codes de statut, les en-têtes et le corps des réponses attendues.

4.6 Exemple MockMvc : Tests du BookController

4.6.1 Test pour la création d'un livre

Simule une requête POST pour créer un nouveau livre. Convertit un objet BookDto en JSON, envoie la charge utile JSON via MockMvc, et vérifie le code de statut (201 Créé) et le contenu de la réponse (par exemple, titre, auteur).

```
@Test
public void testCreateBook() throws Exception {
    AuthorDto author = new AuthorDto("Sami", 21);
    CategoryDto category = new CategoryDto("Info");
    BookDto book = new BookDto("Java", author, category,
       parseDate("2024-11-20"));

    String bookJson = objectMapper.writeValueAsString(book);

    mockMvc.perform(MockMvcRequestBuilders.post("/api/books/save"
       )
            .contentType(MediaType.APPLICATION_JSON)
            .content(bookJson))
            .andExpect(MockMvcResultMatchers.status().isCreated()
               )
            .andExpect(MockMvcResultMatchers.jsonPath("$.title").
               value("Java"))
            .andExpect(MockMvcResultMatchers.jsonPath("$.author.
               name").value("Sami"));
}
```

4.6.2 Test de la recherche d'un livre par ID

- Simule une requête GET pour récupérer un livre par ID.
- Vérifie le statut de la réponse (200 OK) et le contenu (par exemple, le titre du livre, l'auteur).

4.6.3 Test de la recherche de tous les livres

- Simule une requête GET pour récupérer tous les livres.
- Vérifie la liste des livres dans la réponse.

4.6.4 Intégration de MockMvc dans la classe de test

- `@SpringBootTest` : Charge l'ensemble du contexte de l'application pour les tests.
- `@AutoConfigureMockMvc` : Active MockMvc pour les tests d'intégration.
- `MockMvc.perform` : Exécute les requêtes HTTP.
- `ObjectMapper` : Sérialise les objets Java en JSON et vice versa.

4.7 3.3.2 Pourquoi des tests d'intégration ?

Les tests d'intégration valident l'interaction entre :

- **Les couches Contrôleur et Service**.
- **Les DTO, Entités et JSON**.

5 Utilité de la transformation JSON-DTO-Entité dans un projet en couches

5.1 Transformation JSON-DTO-Entité

Dans notre projet en couches, la **transformation JSON-DTO-Entité** est un patron de conception essentiel qui garantit un flux de données propre entre les différentes couches de l'application.

- **JSON** :
  - Représente les données échangées entre le client et le serveur.
  - Utilisé comme format standard pour les corps des requêtes et des réponses HTTP en raison de sa simplicité et de sa lisibilité.
- **DTO (Data Transfer Object)** :
  - Sert d'intermédiaire entre le client et les couches internes de l'application.
  - Permet la validation et la structuration des données entrantes et sortantes sans exposer la structure de la base de données.
  - Prévient le couplage étroit entre les API externes et les modèles internes du domaine.
- **Entité** :
  - Représente la structure réelle de la base de données utilisée pour la persistance.
  - La couche Service convertit les DTO en Entités pour les opérations sur la base de données et inversement.

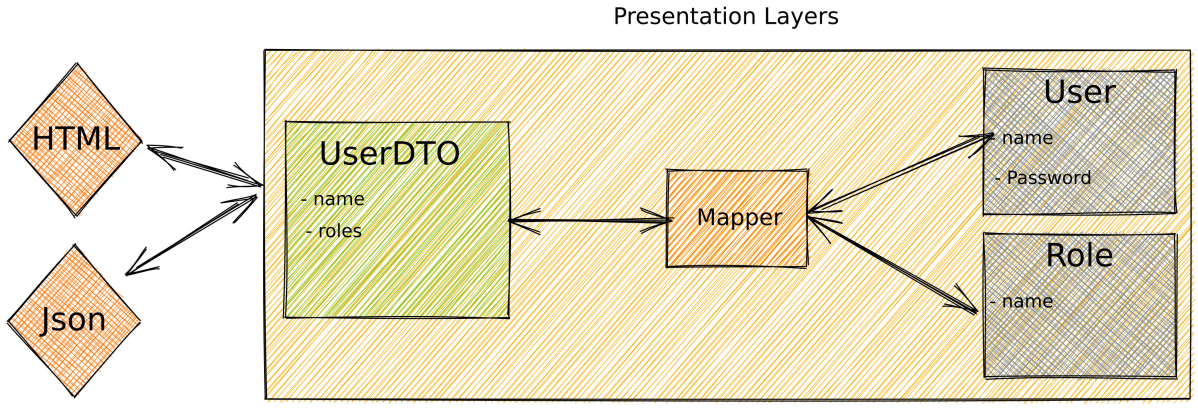

Figure 2: Ajoutez un diagramme montrant la transformation de JSON → DTO → Entité.

6 Comprendre l'architecture REST

Le *Representational State Transfer* (REST) est un style architectural utilisé pour concevoir des applications en réseau. REST repose sur les principes de l'absence d'état, des interactions orientées ressources et de l'utilisation des méthodes HTTP standard.

Les applications RESTful exposent leurs ressources via des Identifiants Uniformes de Ressources (URI) et permettent l'interaction avec ces ressources à l'aide des verbes HTTP suivants :

- **GET** : Récupérer une ressource ou une liste de ressources.
- **POST** : Créer une nouvelle ressource.
- **PUT** : Mettre à jour une ressource existante.
- **DELETE** : Supprimer une ressource.

Un système RESTful communique en utilisant un protocole sans état, généralement HTTP, ce qui assure évolutivité et simplicité. Les données sont souvent échangées dans des formats comme JSON ou XML, ce qui les rend légères et compatibles avec différents types de clients.

REST joue un rôle essentiel dans le développement des applications web modernes en permettant des interactions flexibles et efficaces entre les clients et les serveurs.

6.1 REST dans le contexte du projet

Dans ce projet, REST a été choisi pour concevoir des API facilitant la communication entre le client et le serveur. Ces API gèrent des ressources telles que les données des utilisateurs, les tâches ou les transactions, en suivant les principes REST. En adoptant REST, l'application bénéficie d'une meilleure modularité, d'une intégration facilitée avec les systèmes tiers et d'interactions client-serveur simplifiées.

6.2 Tester les API RESTful avec MockMvc

Les API RESTful mises en œuvre dans ce projet ont été testées à l'aide de `MockMvc`, un framework fourni par Spring pour tester les applications web. En utilisant `MockMvc`, nous avons assuré que les API respectaient les principes REST et renvoyaient les résultats attendus pour toutes les méthodes HTTP.

Les tests se sont concentrés sur :

- Vérification de la correcte implémentation des méthodes HTTP (GET, POST, PUT, DELETE).
- Vérification de l'accessibilité des ressources via les URI.
- Validation des formats de réponse (par exemple, JSON) et des codes de statut HTTP.
- Assurer un traitement adéquat des cas limites, tels que les entrées invalides ou les accès non autorisés.

Cette approche garantit la fiabilité et la précision des API RESTful, qui sont essentielles pour l'architecture et le fonctionnement de l'application.

7 Conclusion

Dans ce rapport, nous avons exploré les aspects clés de la couche Contrôleur dans une application Spring Boot, en soulignant son rôle dans la gestion des requêtes HTTP et en servant de point d'entrée à la logique métier de l'application. En détaillant la construction et l'objectif de divers contrôleurs, notamment les contrôleurs Author, Book et Category, nous avons montré l'implémentation pratique des principes REST.

L'intégration d'annotations telles que `@RestController`, `@RequestMapping` et `@Autowired` permet un développement simplifié et le respect des normes web modernes. De plus, l'incorporation des DTO permet une séparation propre entre les demandes du client et le modèle de données de l'application.

Pour valider la fonctionnalité de ces contrôleurs, nous avons utilisé `MockMvc` pour les tests d'intégration. Cette approche nous a permis de tester efficacement les points de terminaison API sans la surcharge d'une configuration complète de serveur, en fournissant des retours rapides sur la précision de la logique du contrôleur et ses interactions avec la couche service.

En combinant des principes de conception robustes avec des stratégies de test efficaces, ce rapport met en évidence une méthodologie structurée pour construire et valider des API RESTful dans Spring Boot. Ces connaissances posent une base solide pour développer des applications web évolutives et maintenables tout en garantissant une performance fiable grâce à des tests approfondis.