Couche Persistance

December 22, 2024

1 Introduction

La couche **persistance** agit comme un intermédiaire entre la logique métier de l'application et les systèmes de stockage des données. Ses principaux rôles incluent :

- **Stockage des données** : Elle sauvegarde les données générées par l'application afin qu'elles puissent être récupérées plus tard, même après la fermeture de l'application.
- **Récupération des données** : Elle extrait les données stockées lorsque cela est nécessaire, permettant à l'application d'afficher ou de manipuler ces informations.

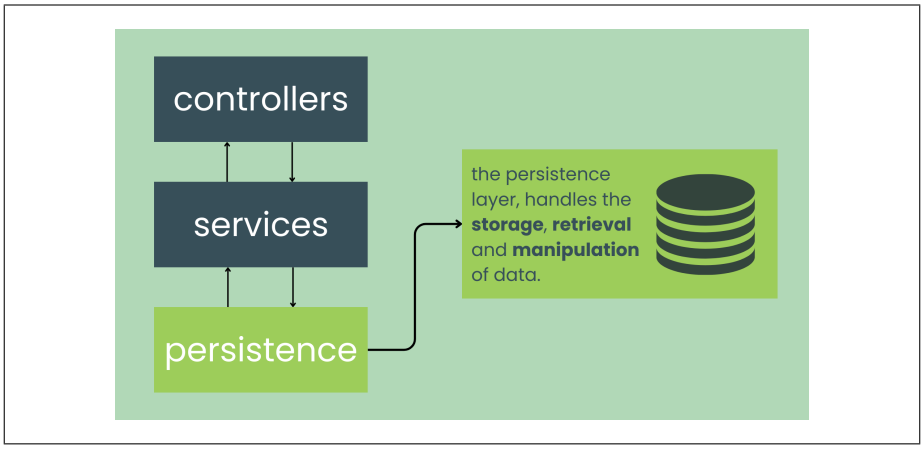

Figure 1: Couches de l'application

Dans les sections suivantes, nous allons explorer les différentes approches pour configurer la couche de persistance dans une application multi-couches.

2 Approche classique avec JDBC

2.0.1 API JDBC

- **JDBC** (Java Database Connectivity) est une API (Interface de Programmation d'Applications) qui permet aux applications Java d'interagir avec des **bases de données relationnelles**. Elle fournit un ensemble standard d'interfaces et de classes pour se connecter aux bases de données, exécuter des requêtes SQL et gérer les résultats.
- **API de bas niveau** : JDBC est une API de bas niveau, ce qui signifie que les développeurs doivent gérer manuellement les connexions à la base de données, traiter les exceptions SQL et fermer les ressources.

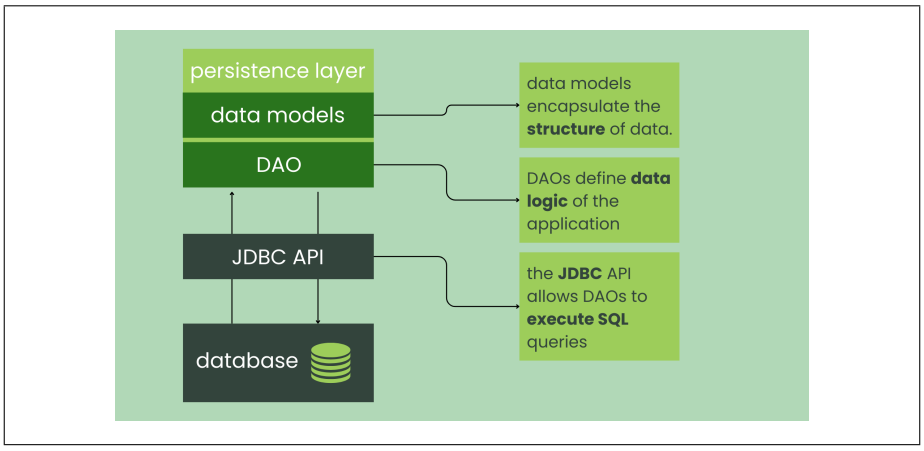

Figure 2: API JDBC

2.1 Configuration de la connexion

La première étape pour configurer la couche de persistance est d'inclure la dépendance JDBC dans le classpath de l'application, soit manuellement, soit en ajoutant la balise de dépendance appropriée dans Maven.

L'architecture décrite ci-dessous représente la base pour établir une connexion dans l'application.

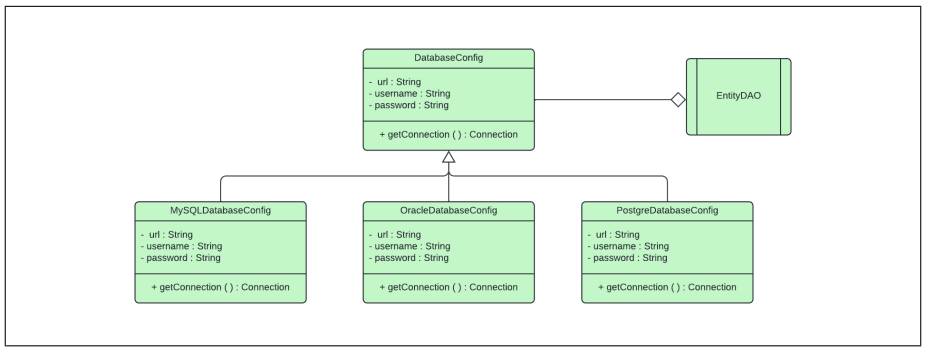

Figure 3: Configuration de la connexion à la base de données

- **Configuration de la base de données** : Cette classe abstraite définit le comportement commun pour toutes les classes de configuration de base de données. La méthode `getConnection()` de cette classe retourne la connexion à un SGBDR (Système de Gestion de Base de Données Relationnelle) spécifié.
- **Classes enfants** : Chaque classe enfant est responsable de l'établissement d'une connexion à un SGBDR spécifique, en héritant de la classe de base `DatabaseConfiguration`.
- **Classe DAO** : La classe DAO (Data Access Object) gère l'implémentation de la logique de données, exécute les requêtes SQL et utilise la connexion fournie par la classe de configuration de la base de données pour interagir avec cette dernière.

2.2 Logique métier des données

La logique métier des données repose sur deux composants principaux.

L'architecture ci-dessous décrit la structure générale de la couche de persistance dans une application **CRUD**.

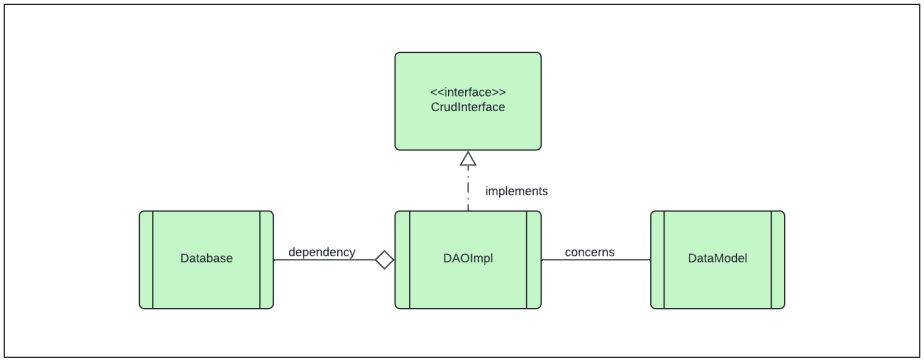

Figure 4: Architecture de la couche de persistance dans une application CRUD

Composants clés :

- **Modèle de données** : Une classe qui définit le comportement commun pour toutes les classes concrètes de modèle de données, représentant la structure des données gérées par l'application.
- **Interface CRUD** : Une interface qui définit les signatures des méthodes pour les opérations `CRUD` principales : Créer, Lire, Mettre à jour et Supprimer.
- **Implémentation DAO** : Classes concrètes qui implémentent les méthodes CRUD pour un modèle de données spécifique, en gérant la logique d'interaction avec la base de données.

2.3 Exemple de code

Voici un exemple simple illustrant comment implémenter la couche de persistance pour une entité `User` en utilisant JDBC. Cela inclut la configuration de la base de données, un modèle de données, une interface CRUD et une implémentation DAO. Le code complet est disponible sur le dépôt GitHub du projet :
`https://github.com/D0esN0tM1tter/spring_boot_project-/tree/master/supporting_project/persistence_examples/jdbc_classic_approach`.

2.3.1 Configuration de la base de données

La classe **DatabaseConfiguration** est une classe utilitaire abstraite qui centralise la logique de connexion à la base de données. Cela garantit que les détails de connexion (comme l'URL, le nom d'utilisateur et le mot de passe) sont gérés en un seul endroit, favorisant la réutilisabilité et la maintenabilité.

Explication :

- `DriverManager.getConnection` : Établit une connexion à la base de données en utilisant l'URL, le nom d'utilisateur et le mot de passe spécifiés.
- Les constantes comme `URL`, `USER` et `PASSWORD` stockent les détails de configuration de manière sécurisée et les rendent facilement modifiables.
- `Connection` : Représente une connexion active à la base de données, nécessaire pour exécuter des requêtes SQL.

2.3.2 Modèle User

La classe **User** sert de modèle de données pour représenter la structure d'un enregistrement dans la base de données. Chaque instance correspond à une ligne de la table `users`.

Explication :

- Les champs comme `id`, `name` et `email` correspondent directement aux colonnes de la table de la base de données.
- Les méthodes getter et setter permettent l'encapsulation, offrant un accès contrôlé à ces champs.

2.3.3 Interface CRUD

L'interface **UserDAO** définit le contrat pour effectuer des opérations **CRUD** sur les données `User`.

Explication :

- `createUser(User user)` : Ajoute un nouvel enregistrement utilisateur à la base de données.
- `getUserById(int id)` : Récupère un enregistrement utilisateur basé sur son ID.
- `getAllUsers()` : Récupère tous les enregistrements utilisateurs de la base de données.
- `updateUser(User user)` : Met à jour un enregistrement utilisateur existant avec de nouvelles informations.
- `deleteUser(int id)` : Supprime un enregistrement utilisateur basé sur son ID.

2.3.4 Implémentation DAO

La classe **UserDAOImpl** implémente l'interface `UserDAO`. Elle fournit la logique concrète pour interagir avec la base de données.

Explication :

- `PreparedStatement` : Utilisé pour exécuter des requêtes paramétrées de manière sécurisée, empêchant les attaques par injection SQL.

- `ResultSet` : Stocke le résultat des requêtes `SELECT`, permettant une itération sur les lignes retournées.
- Chaque méthode utilise un bloc `try-with-resources` pour s'assurer que les connexions, instructions et autres ressources sont automatiquement fermées.

3 Approche JDBC template

3.1 JDBC template

Le **JdbcTemplate** de Spring est une API de haut niveau dans le framework Spring qui simplifie les interactions avec la base de données en gérant les tâches courantes telles que la gestion des connexions, l'exécution des instructions SQL et la gestion des exceptions. Cela permet de rendre le code d'accès aux données plus concis et de réduire le code répétitif associé à l'API standard **JDBC**.
REMARQUE : Pour utiliser le JDBC template, il est nécessaire de le déclarer comme dépendance dans le fichier `pom.xml`.

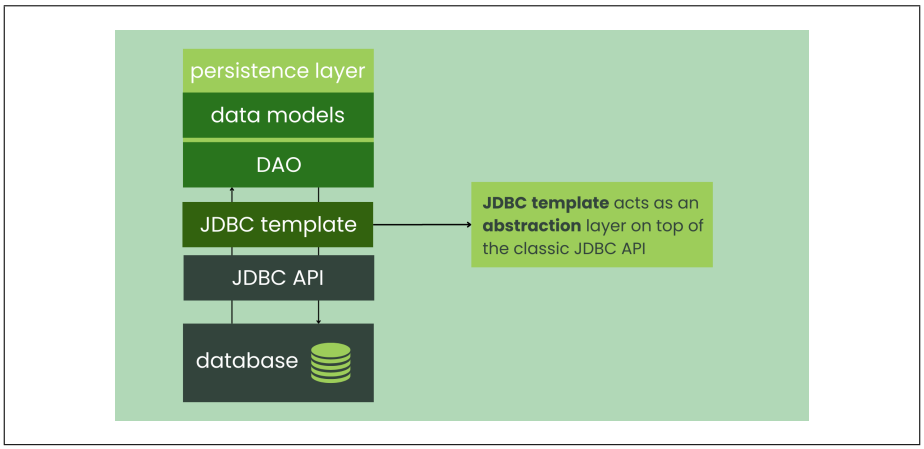

Figure 5: JDBC template

3.2 Configuration de la connexion

Dans une application Spring Boot, les détails de la connexion sont définis dans le fichier `application.properties`. Sur la base de cette configuration, le conteneur IoC (Inversion de Contrôle) de Spring génère automatiquement un objet `DataSource`, qui est ensuite injecté dans l'objet `JdbcTemplate` pour permettre les opérations sur la base de données.

```
# Configuration de la base de donn es
spring.datasource.url=jdbc:mysql://localhost:3306/
    mydatabase
spring.datasource.username=myuser
spring.datasource.password=mypassword
spring.datasource.driver-class-name=com.mysql.cj.
    jdbc.Driver
```

Listing 1: Configuration de la base de données MySQL

3.3 Logique métier des données

Dans le cadre de l'utilisation du JDBC template, la couche de persistance suit une architecture similaire à l'approche précédente. Chaque modèle de données dispose d'une classe DAO correspondante responsable de la gestion de ses opérations sur la base de données.
Cependant, au lieu d'utiliser des objets `Connection` pour interagir avec la base de données, les DAOs utilisent maintenant le `JdbcTemplate`.

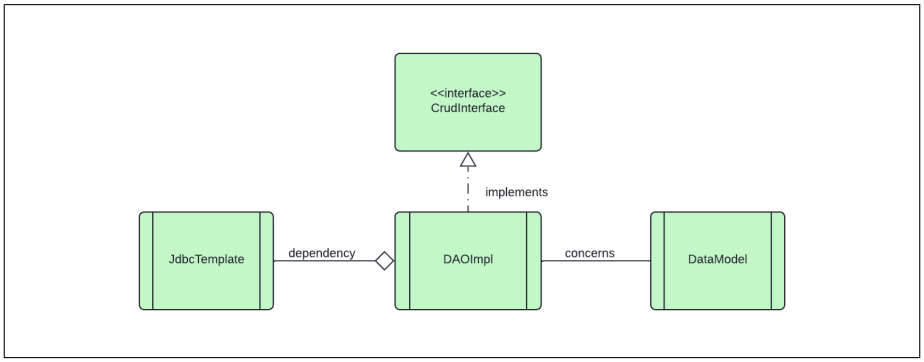

Figure 6: JDBC template

La mise en place de la couche de persistance dans ce contexte implique plusieurs étapes clés :

1. **Configurer JdbcTemplate comme un Bean** : Dans une classe `@Configuration`, définissez `JdbcTemplate` comme un bean Spring. Cela permet à l'instance `JdbcTemplate` d'être injectée comme dépendance dans les classes DAO pour leur permettre d'effectuer des opérations sur la base de données.

2. **Créer des modèles de données** : Définissez des classes de modèle de données qui représentent les entités dans la base de données. Ces classes

seront utilisées pour mapper les enregistrements de la base de données aux objets Java.

3. **Définir l'interface CRUD** : Créez une interface qui spécifie les opérations CRUD (Créer, Lire, Mettre à jour, Supprimer). Cette interface sera implémentée par les DAOs pour gérer l'accès aux données de manière cohérente.

4. **Implémenter les DAOs** : Développez des classes DAO qui fournissent des implémentations concrètes des opérations CRUD définies dans l'interface. Ces classes utiliseront `JdbcTemplate` pour interagir avec la base de données.

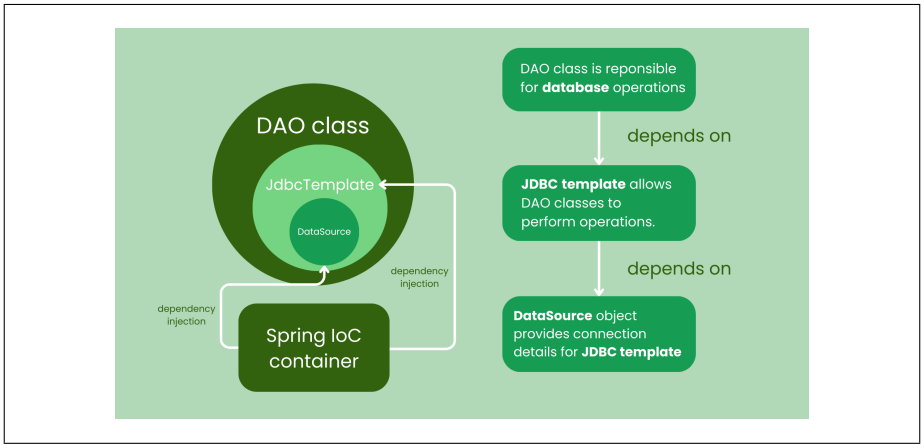

Figure 7: Injection de dépendance dans la couche de persistance

3.4 Exemple de code

L'exemple suivant montre l'implémentation d'une couche de persistance pour la gestion d'une entité `Product` à l'aide de Spring JDBC. Il comprend la configuration de la base de données, un modèle de données, une interface DAO et une implémentation DAO. Le code complet est disponible sur le dépôt GitHub du projet :
`https://github.com/DOesNOtM1tter/spring_boot_project-/tree/master/supporting_project/persistence_examples/jdbctemplate_example`.

3.4.1 Configuration de la base de données

La classe **DatabaseConfig** définit le bean `JdbcTemplate`, qui est essentiel pour exécuter des opérations SQL dans les applications Spring.

Explication :

- `JdbcTemplate` : Un utilitaire fourni par Spring pour interagir avec la base de données de manière simple et efficace.
- `DataSource` : Injecté dans le `JdbcTemplate`, il fournit la connexion à la base de données.

3.4.2 Modèle de données Product

La classe **Product** sert de modèle de données pour l'entité `Product`, représentant un enregistrement dans la base de données.

Explication :

- Les annotations `@Data`, `@AllArgsConstructor` et `@NoArgsConstructor` de Lombok génèrent automatiquement le code standard comme les getters, setters et constructeurs.
- Les champs comme `id`, `label` et `price` sont directement liés aux colonnes de la table de la base de données.

3.4.3 Interface DAO Product

L'interface **ProductDao** définit les opérations CRUD pour l'entité `Product`.

Explication :

- Des méthodes comme `save`, `update` et `deleteById` gèrent la logique de persistance pour l'entité `Product`.
- `findById` et `findAll` permettent de récupérer un ou plusieurs produits, respectivement.

3.4.4 Implémentation DAO

La classe **ProductDaoImpl** implémente l'interface `ProductDao`, fournissant des méthodes concrètes pour interagir avec la base de données.

Explication :

- `RowMapper` : Permet de mapper chaque ligne du jeu de résultats à un objet `Product`.
- `KeyHolder` : Capture l'ID généré automatiquement lorsqu'un nouveau produit est inséré dans la base de données.
- Chaque méthode utilise `JdbcTemplate` pour exécuter les requêtes SQL de manière efficace.

4 Approche Java Persistence API

Cette section décrit l'utilisation de l'API Java Persistence (JPA) et Hibernate pour implémenter une couche de persistance dans les applications Java. Elle couvre une introduction à JPA, le processus de configuration pour établir une connexion à la base de données, ainsi que l'intégration de la logique métier.

4.1 JPA et Hibernate

L'API de persistance Java (JPA) est une spécification standard pour le mappage objet-relationnel (ORM) en Java. Elle offre une approche indépendante de la plate-forme pour interagir avec les bases de données relationnelles en mappant les objets Java sur des tables de base de données. Hibernate est une implémentation populaire de JPA qui propose des fonctionnalités ORM avancées et une intégration transparente avec JPA.

Principales caractéristiques :

- Élimine le code redondant pour la gestion des opérations de base de données.
- Fournit des annotations pour mapper les classes Java aux tables de base de données.
- Offre un langage de requête (JPQL) pour interagir avec la base de données de manière orientée objet.
- Gère automatiquement les connexions et transactions de base de données.

Avantages de JPA et Hibernate :

- Simplifie les opérations sur la base de données grâce à la gestion des entités.
- Réduit le risque d'injection SQL en utilisant des requêtes paramétrées.
- Améliore la portabilité entre différentes bases de données.

4.2 Configuration de la connexion

Pour utiliser JPA et Hibernate dans un projet, certaines configurations sont nécessaires pour établir une connexion à la base de données et gérer les entités. Dans une application Spring Boot, les paramètres de connexion et de JPA peuvent être configurés dans le fichier `application.properties`.

Étapes pour la configuration :

1. Ajouter les dépendances pour JPA et Hibernate dans le fichier `pom.xml` (Maven) ou `build.gradle` (Gradle).
2. Créer un fichier de configuration (`application.properties`) pour définir les propriétés de la base de données telles que l'URL, le nom d'utilisateur, le mot de passe et les paramètres spécifiques à Hibernate.
3. Annoter les classes Java avec des annotations JPA (`@Entity`, `@Table`, etc.) pour les définir comme des entités mappées aux tables de la base de données.
4. Utiliser `EntityManagerFactory` pour créer un `EntityManager`, qui fournit une interface pour interagir avec le contexte de persistance.

Configuration des propriétés de l'application : Pour configurer la connexion à la base de données MySQL dans une application Spring Boot, les propriétés suivantes doivent être ajoutées au fichier `application.properties` :

Exemple de configuration de `application.properties` :

```
# Configuration de la base de donn es MySQL
spring.datasource.url=jdbc:mysql://localhost:3306/
    testdb?useSSL=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-name=com.mysql.cj.
    jdbc.Driver
spring.jpa.database-platform=org.hibernate.dialect.
    MySQL5InnoDBDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Cette configuration garantit que Spring Boot utilise Hibernate comme fournisseur JPA et établit la connexion à la base de données MySQL avec les identifiants et paramètres nécessaires.

4.3 Logique métier des données

La couche de logique métier interagit avec la couche de persistance pour effectuer des opérations CRUD et appliquer des règles métier.

Responsabilités de la couche logique métier :

- Valider et transformer les données avant la persistance.
- Coordonner entre la couche de persistance et les autres couches de l'application (par exemple, les contrôleurs).
- Gérer les transactions de base de données, garantissant la cohérence des données.
- Abstraire la logique de persistance de la couche de présentation.

Utilisation des Repositories : Les repositories dans JPA sont des interfaces qui permettent de définir et de gérer les interactions avec la base de données avec un minimum de code standard. Hibernate peut étendre ces repositories pour fournir des requêtes personnalisées et des fonctionnalités avancées.

Gestion du cycle de vie des entités : JPA fournit différents états pour les entités tels que *transient*, *persistent*, et *detached*, permettant aux développeurs de gérer efficacement le cycle de vie des entités.

4.4 Exemple de code

Dans cette section, nous allons passer en revue un exemple simple montrant comment configurer et utiliser JPA avec Hibernate dans une application Spring Boot. Le code complet est disponible sur le dépôt GitHub du projet : `https://github.com/DOesNOtM1tter/spring_boot_project-/tree/master/supporting_project/persistence_examples/jpa_examples`.

Étape 1 : Classe d'entité

Pour définir une entité JPA, nous devons annoter la classe avec `@Entity` et la mapper à une table dans la base de données en utilisant `@Table`. De plus, nous utilisons `@Id` pour marquer le champ clé primaire et `@GeneratedValue` pour la génération automatique de l'ID.

Explication :

- `@Entity` : Marque la classe comme une entité JPA.
- `@Id` : Spécifie la clé primaire de l'entité.
- `@GeneratedValue` : Configure la stratégie de génération de la clé primaire (par exemple, `IDENTITY` pour les IDs auto-incrémentés).
- `@Column` : Mappe les champs aux colonnes de la base de données, avec des options comme `nullable` pour définir les contraintes des colonnes.

4.4.1 Repository du Produit

L'interface du repository étend `JpaRepository`, offrant des opérations CRUD intégrées.

Explication :

- `JpaRepository` : Fournit des méthodes telles que `save()`, `findById()`, `findAll()` et `deleteById()` pour les opérations de base de données sans avoir besoin de mises en œuvre personnalisées.
- Des méthodes de requête personnalisées peuvent être ajoutées si nécessaire.

4.4.2 Couche Service

La couche service interagit avec le repository pour gérer la logique métier.

Explication :

- La couche service est utilisée pour abstraire la logique métier, rendant la couche contrôleur plus claire.
- L'annotation `@Autowired` injecte le repository, permettant une injection de dépendances facile.
- Les méthodes telles que `saveProduct()` et `getProductById()` offrent un accès aux méthodes du repository pour créer, récupérer et supprimer des produits.

4.4.3 Couche Contrôleur

Le contrôleur expose les méthodes du service via des points de terminaison RESTful pour l'interaction avec le front-end.

Explication :

- `@RestController` : Marque la classe comme un contrôleur REST qui gère les requêtes HTTP.
- `@RequestMapping` : Spécifie l'URL de base pour les points de terminaison du contrôleur.
- `@PostMapping`, `@GetMapping` et `@DeleteMapping` : Gèrent respectivement les requêtes HTTP POST, GET et DELETE.
- `@RequestBody` et `@PathVariable` : Gèrent les charges utiles des requêtes et les paramètres d'URL dynamiques.

Étape 5 : Configuration de l'application

Assurez-vous que l'application est correctement configurée pour exécuter une application Spring Boot avec JPA. Les détails de la configuration peuvent être définis dans le fichier `application.properties`, comme décrit dans la section précédente.

Cette configuration crée un flux complet pour gérer les entités dans une application Spring Boot avec JPA et Hibernate. Avec une configuration minimale, Spring Boot gère la plupart du code redondant, permettant aux développeurs de se concentrer sur la logique métier.

5 Conclusion

Dans ce chapitre, nous avons exploré différentes approches pour gérer la couche de persistance dans une application Spring Boot. Nous avons couvert l'implémentation de la persistance avec JDBC, détaillant la configuration de la base de données, les modèles de données et les interfaces CRUD, ainsi que l'utilisation de `JdbcTemplate` pour interagir avec la base de données. En parallèle, nous avons également abordé la configuration de la persistance avec JPA et Hibernate, la configuration de la connexion à la base de données et la définition des entités à l'aide des annotations JPA. Ces deux approches offrent des avantages différents en fonction des besoins du projet : JDBC pour des opérations plus manuelles et contrôlées, et JPA/Hibernate pour une gestion automatisée et un mappage objet-relationnel simplifié. En combinant ces techniques, nous pouvons construire une couche de persistance robuste, flexible et maintenable.