# SHOPCONNECT – STRUCTURE DU CODE

## Architecture de l'application

L'application ShopConnect que je vais développer suit une architecture Java avec Spring Boot, en appliquant le modèle MVC (Modèle-Vue-Contrôleur). Cette architecture permet une séparation claire des responsabilités et garantit une maintenance facile et évolutive.

Mon application sera divisée en quatre couches principales :

* Controller (la couche de présentation)
* Service (la couche de logique métier)
* Repository (la couche d'accès aux données)
* Model (la couche de représentation des données)

L'objectif de cette architecture est de structurer le code de manière à ce que chaque couche ait un rôle bien défini, favorisant ainsi la clarté et la maintenabilité du projet.

## 1. Controller : La couche de présentation

Le Controller est le point d'entrée de l'application. Il gère toutes les requêtes HTTP provenant du frontend ou d'une API et retourne les réponses sous forme de JSON ou XML.

### Rôles du Controller :

Recevoir les requêtes : Le Controller intercepte les requêtes HTTP entrantes (GET, POST, PUT, DELETE) depuis le frontend (PHP) ou une API.

Appeler les services : Il redirige ces requêtes vers la couche Service, qui est chargée d'appliquer la logique métier nécessaire pour traiter les données.

Envoyer des réponses : Une fois que le traitement est terminé, le Controller renvoie une réponse HTTP au frontend (souvent des données JSON).

### Exemple concret dans ShopConnect :

Dans ShopConnect, un ProductController pourrait recevoir une requête pour obtenir la liste des produits disponibles. Le Controller va transmettre cette requête au ProductService, qui se chargera de récupérer les données via le ProductRepository. Ensuite, le Controller renverra ces données au client.

## 2. Service : La couche de logique métier

La couche Service est l'endroit où se trouve toute la logique métier de l'application. Le Service orchestre les appels aux repositories et applique les règles métier avant de renvoyer les résultats aux Controllers.

### Rôles du Service :

Appliquer la logique métier : Le Service exécute les règles spécifiques à ShopConnect, telles que la vérification de stock ou la validation des commandes.

Accéder aux données : Il appelle les méthodes des repositories pour interagir avec la base de données (lecture, écriture).

Assurer l'intégrité des données : Avant de persister ou de manipuler les données, le Service s'assure que les règles métier sont respectées, comme par exemple la vérification de la disponibilité des produits avant de permettre une commande.

### Exemple concret dans ShopConnect :

Le ProductService pourrait avoir une méthode getAllProducts() qui appelle le ProductRepository pour récupérer tous les produits disponibles dans la base de données. Avant de renvoyer les produits, il pourrait également appliquer certaines règles, comme filtrer les produits en fonction de leur disponibilité.

## 3. Repository : La couche d'accès aux données

La couche Repository est responsable de l'accès aux données. Elle interagit directement avec la base de données (dans ce cas, PostgreSQL) pour effectuer des opérations CRUD (Create, Read, Update, Delete).

### Rôles du Repository :

Interaction avec la base de données : Les repositories utilisent Spring Data JPA pour récupérer, insérer, mettre à jour ou supprimer des données dans la base de données.

Pas de logique métier : Le Repository n'applique aucune logique métier, il se contente de gérer la persistance des entités.

Simplification de l'accès aux données : Grâce à Spring Data JPA, les méthodes de base comme findAll(), save(), et delete() sont automatiquement générées sans avoir à écrire de requêtes SQL.

### Exemple concret dans ShopConnect :

Le ProductRepository pourrait utiliser JpaRepository pour faciliter la gestion des produits dans la base de données. Il permettra de récupérer la liste des produits, de les ajouter, de les modifier ou de les supprimer, sans avoir à écrire des requêtes SQL complexes.

## 4. Model : La couche de représentation des données (Entités)

La couche Model (ou entités) représente les objets métier de ShopConnect. Chaque entité correspond à une table de la base de données. Par exemple, une entité Product représenterait la table products dans PostgreSQL.

### Rôles du Model :

Représenter les données : Les entités définissent les données sous forme d’objets Java.

Mapping avec la base de données : Grâce à JPA, chaque entité est mappée à une table de la base de données, et chaque champ de l'entité correspond à une colonne de la table.

Gérer les relations : Les entités peuvent définir des relations entre elles, comme des relations @OneToMany, @ManyToOne pour représenter les liens entre les différentes tables (ex. : une boutique possède plusieurs produits).

### Exemple concret dans ShopConnect :

L'entité Product définira les champs tels que id, name, price, description, etc., correspondant aux colonnes de la table products. De même, une relation @ManyToOne peut être définie pour indiquer qu'un produit appartient à une boutique.

## 5. Relations entre les couches

Le Controller agit comme l'intermédiaire entre les utilisateurs (ou le frontend PHP) et la logique métier (Service). Il reçoit des requêtes et retourne des réponses. Le Service orchestre les actions et les appels au Repository, qui interagit directement avec la base de données pour persister ou récupérer des Models (entités).

Voici un schéma représentant les interactions :

#### [Controller] <---> [Service] <---> [Repository] <---> [Database]

#### ↑

#### [Model]

### Dans cette structure :

Le Controller interagit uniquement avec le Service pour déléguer la logique métier.

Le Service centralise toute la logique et délègue l'accès aux données au Repository.

Le Repository se charge des opérations sur la base de données via JPA.

Le Model représente les entités qui sont mappées sur les tables de la base de données.

## 6. Avantages de cette architecture

### Séparation des responsabilités :

Chaque couche a une responsabilité distincte : le Controller gère les requêtes, le Service applique la logique métier, et le Repository accède aux données. Cela facilite la compréhension du code et sa maintenance.

### Facilité de maintenance :

Si une logique métier doit être modifiée, je peux la changer directement dans le Service sans affecter le Controller ou le Repository.

### Réutilisation du code :

Les Services et les Repositories peuvent être réutilisés à plusieurs endroits dans l'application, réduisant ainsi la duplication de code.

### Testabilité :

Cette architecture est idéale pour les tests unitaires. Chaque couche peut être testée indépendamment : je pourrai tester les Services sans avoir besoin d’appeler les Controllers ou la base de données directement.