**Architecture des composants d'entreprise**

Professeur : M. Oussama Hamal

# Rapport des Travaux Pratiques (TP1 - TP4)

**Réalisé par :**
Amine Icame / Salma Benomar
Miage

Date : 04 Décembre 2025

# Table des matières

# Chapitre 1

# TP 1 : Implémenter le Design Pattern IOC avec Spring IOC

## Introduction du TP

**Objectif :** Se familiariser avec le framework Spring IOC et l'injection de dépendances pour réduire le couplage fort.

**Compétences visées :**

— Configuration d'un projet Maven Spring.
— Injection par modificateur (Setter), constructeur et Factory (@Bean).
— Tests unitaires avec JUnit 5.

## 1.1 Déroulement du TP

### 1.1.1 Description

Nous avons créé une architecture modulaire (DAO, Service, Modèle) pour gérer des articles. Nous avons d'abord utilisé le fichier `pom.xml` pour les dépendances, puis nous avons implémenté les différentes méthodes d'injection offertes par Spring.

### 1.1.2 Prises d'écran

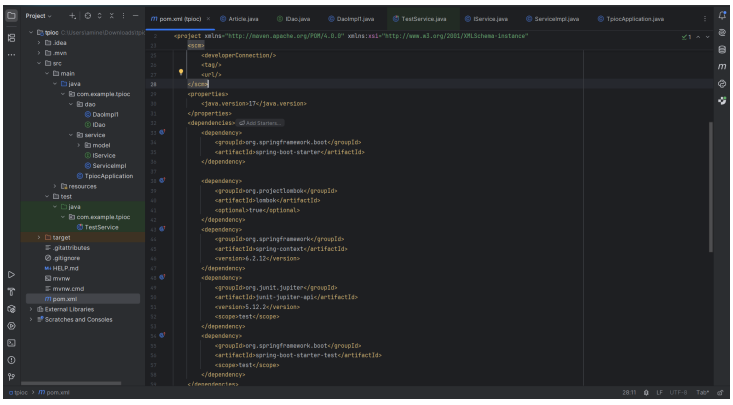

FIGURE 1.1 – Arborescence du projet

**Commentaire :** Structure classique Maven respectant la séparation des couches (dao, service, model).

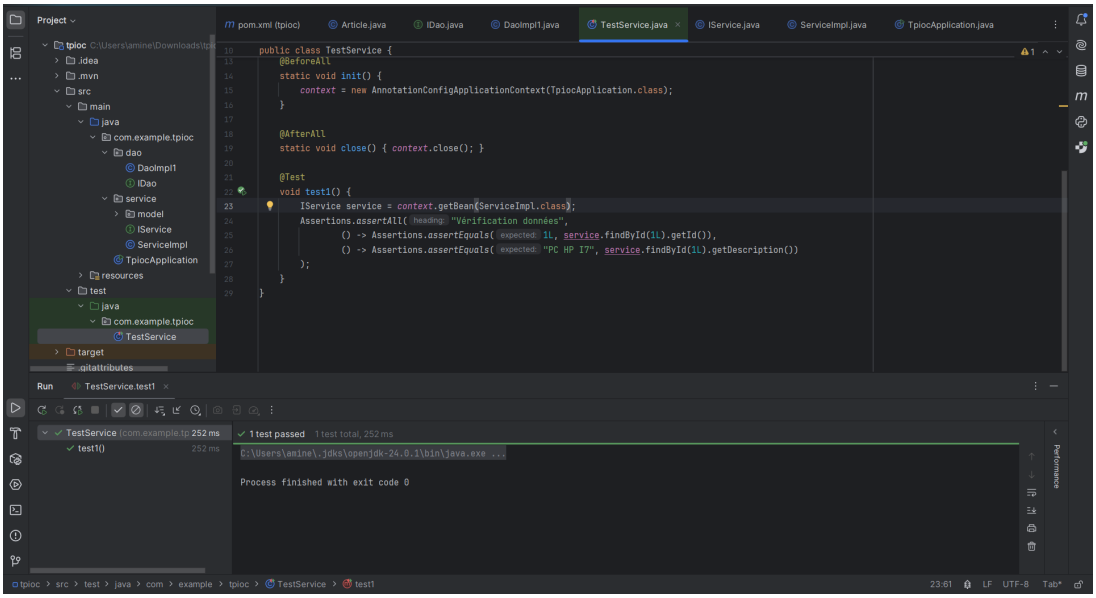


FIGURE 1.2 – Injection par Setter

**Commentaire :** Utilisation de `@Autowired` sur le setter. Spring injecte le bean DAO correspondant.

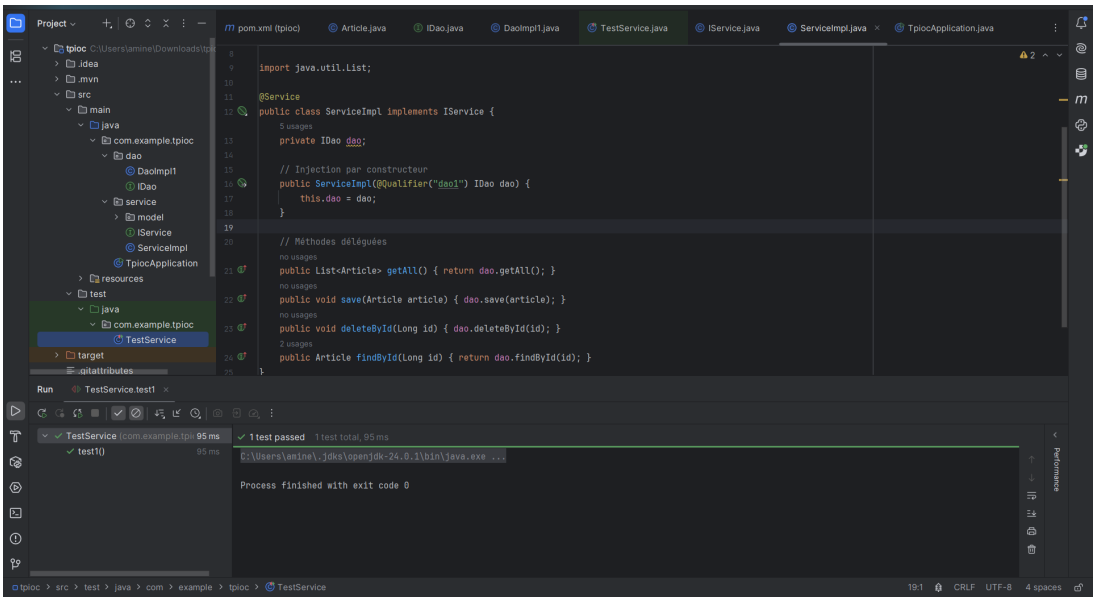


FIGURE 1.3 – Injection par Constructeur

**Commentaire :** Injection recommandée pour garantir l'état de l'objet à la création.

### 1.1.3 Résultats & observations

**Résultats obtenus :** L'exécution des tests unitaires (JUnit) prouve que les dépendances sont correctement injectées. Le test passe au vert à chaque étape.

**Difficultés rencontrées :** Comprendre la différence entre l'injection par défaut (Singleton) et Prototype.

**Solutions adoptées :** Lecture de la documentation et utilisation de `@Qualifier` lorsque deux implémentations de la même interface existent.

### 1.1.4 Code source

Exemple de la classe de test :

```
@Test
void test1() {
    IService service = context.getBean(ServiceImpl.class);
    Assertions.assertEquals(1L, service.findById(1L).getId());
}
```

**Explication :** On récupère le bean Service depuis le contexte Spring et on vérifie que le DAO injecté fonctionne en récupérant un article.

# Conclusion

J'ai appris que l'IOC permet de déléguer la gestion du cycle de vie des objets au framework, rendant le code plus modulaire et testable.

# Chapitre 2

# TP 2 : Développer un service web REST avec Spring Boot

## Introduction du TP

**Objectif :** Créer une API REST complète gérant les méthodes HTTP et la validation des données.

**Compétences visées :**

— Création de contrôleurs REST.
— Utilisation des DTOs et Converters.
— Gestion globale des exceptions avec `@ControllerAdvice`.

## 2.1 Déroulement du TP

### 2.1.1 Description

Nous avons développé une API pour gérer des articles (CRUD). Nous avons configuré Spring Boot, créé les couches métier et exposé les endpoints via un RestController.

### 2.1.2 Prises d'écran

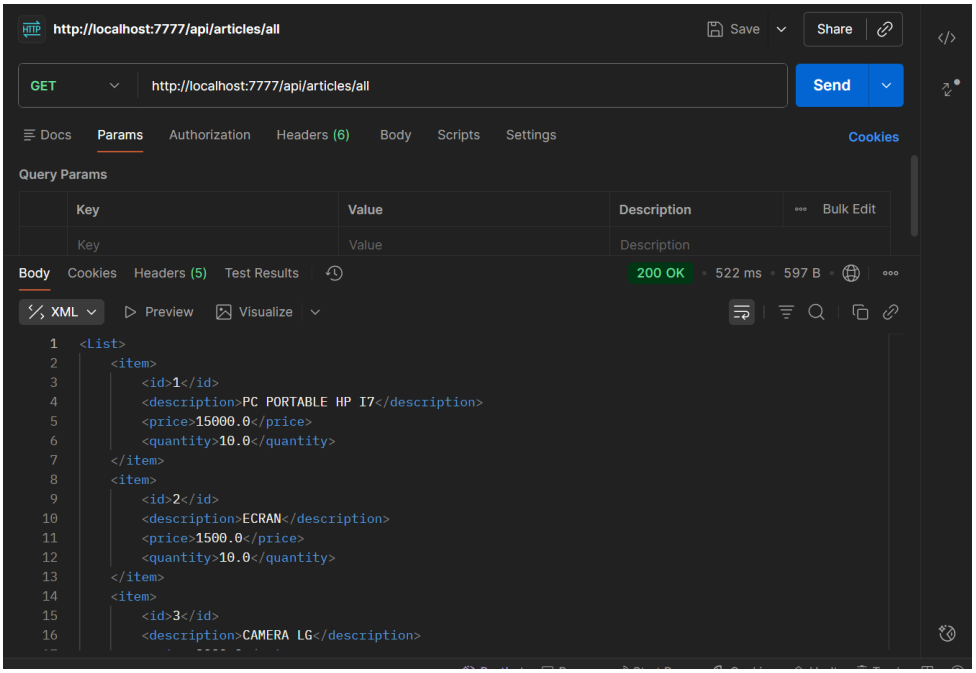


FIGURE 2.1 – Test GET avec Postman (XML)

**Commentaire :** Négociation de contenu : en demandant du XML via le header Accept, l'API s'adapte.

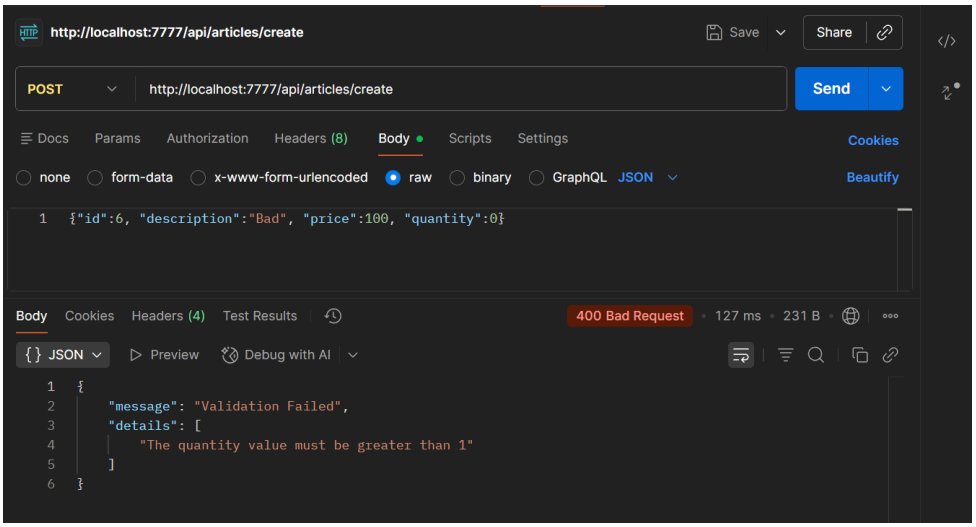


FIGURE 2.2 – Gestion d'erreur de validation

**Commentaire :** Tentative de création avec une quantité invalide. L'API retourne une erreur 400 personnalisée.

### 2.1.3 Résultats & observations

**Résultats obtenus :** L'API répond correctement aux requêtes GET, POST, PUT, DELETE en JSON et XML.

**Difficultés rencontrées :** Gérer correctement les messages d'erreurs pour qu'ils soient lisibles par le client.

**Solutions adoptées :** Implémentation d'un `ExceptionHandlerController` qui capture les exceptions de validation.

### 2.1.4 Code source

```
@PostMapping("/create")
public ResponseEntity<String> create(@Valid @RequestBody ArticleDTO dto)
    {
    service.create(dto);
    return new ResponseEntity<>("Created", HttpStatus.CREATED);
}
```

**Explication :** L'annotation `@Valid` déclenche la validation automatique des champs du DTO.

## Conclusion

Ce TP m'a permis de maîtriser la chaîne complète de traitement d'une requête REST, de la réception à la réponse, en passant par la validation et la gestion des erreurs.

# Chapitre 3

# TP 3 : Développer un service web avec Spring Data REST

## Introduction du TP

**Objectif :** Générer automatiquement une API REST HATEOAS à partir des repositories JPA.

**Compétences visées :**

— Utilisation de Spring Data REST.
— Configuration des Projections.
— Documentation avec Swagger (OpenAPI).

## 3.1 Déroulement du TP

### 3.1.1 Description

Nous avons utilisé Spring Data REST pour exposer directement nos entités `Article` et `Categorie` sans écrire de contrôleur manuel. Nous avons ajouté Swagger pour visualiser l'API.

### 3.1.2 Prises d'écran

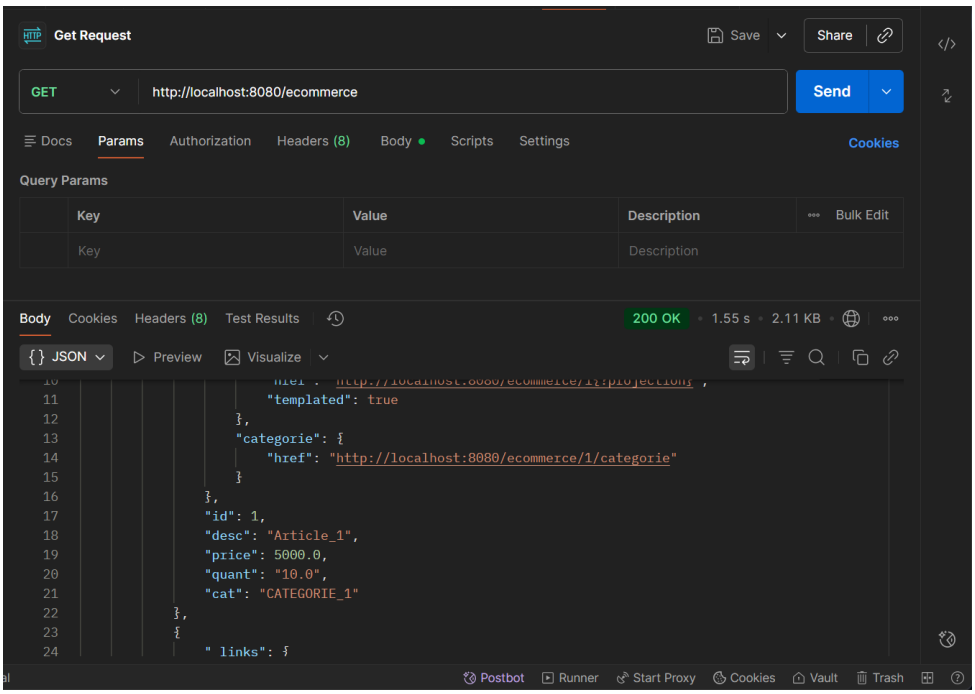


FIGURE 3.1 – Liste des articles (Format HAL)

**Commentaire :** Réponse automatique de Spring Data REST incluant les liens de navigation (HATEOAS).

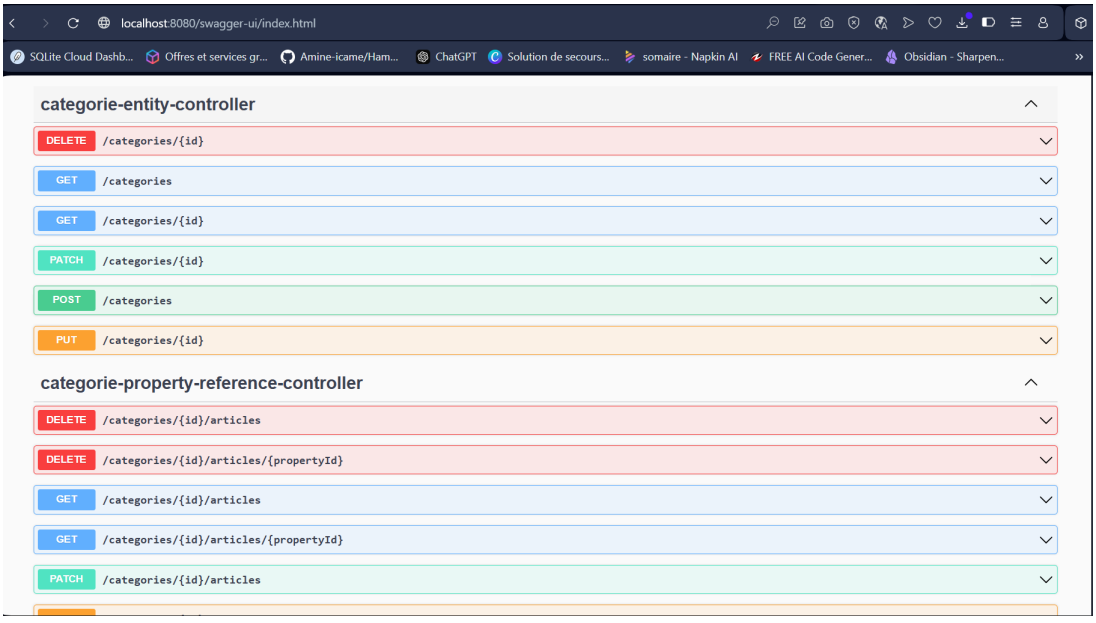


FIGURE 3.2 – Interface Swagger UI

**Commentaire :** Documentation interactive générée automatiquement permettant de tester les endpoints.

### 3.1.3 Résultats & observations

**Résultats obtenus :** Une API complète et documentée obtenue très rapidement.
**Difficultés rencontrées :** Par défaut, les ID ne sont pas exposés dans le JSON.
**Solutions adoptées :** Création d'une interface de `Projection` (ArticleDTO) pour forcer l'affichage de l'ID et personnaliser les champs.

### 3.1.4 Code source

```
1 @RepositoryRestResource(path = "ecommerce", excerptProjection =
      ArticleDTO.class)
2 public interface ArticleRepository extends JpaRepository<Article, Long>
      { ... }
```

**Explication :** Cette seule annotation suffit pour exposer l'entité via REST sur le chemin /ecommerce.

## Conclusion

J'ai appris que Spring Data REST est un outil puissant pour le prototypage rapide, permettant de se concentrer sur le modèle de données plutôt que sur la "plomberie" HTTP.

# Chapitre 4

# TP 4 : Service web avec GraphQL et Spring Boot

## Introduction du TP

**Objectif :** Implémenter une API GraphQL permettant au client de demander spécifiquement les données souhaitées.

**Compétences visées :**

— Définition de schémas GraphQL (.graphqls).

— Implémentation de Queries et Mutations.

— Test avec GraphiQL.

## 4.1 Déroulement du TP

### 4.1.1 Description

Nous avons créé un service bancaire (Clients, Comptes, Transactions). Contrairement à REST, nous avons défini un schéma typé et un endpoint unique pour traiter les demandes flexibles.

### 4.1.2 Prises d'écran

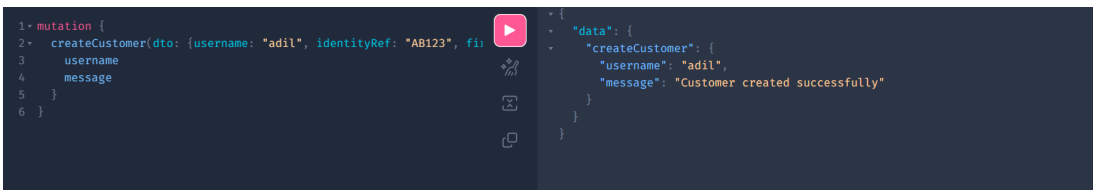


FIGURE 4.1 – Mutation : Création d'un client

**Commentaire :** Envoi d'une mutation pour créer une donnée. On choisit de ne recevoir en retour que le username et le message.

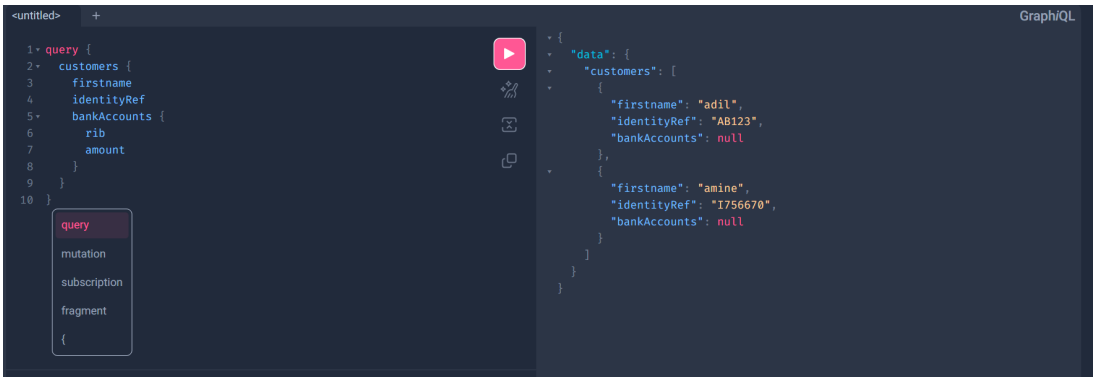


FIGURE 4.2 – Query : Récupération Clients et Comptes

**Commentaire :** En une seule requête, nous récupérons le client et la liste de ses comptes bancaires, ce qui éviterait plusieurs appels API en REST.

### 4.1.3 Résultats & observations

**Résultats obtenus :** Le serveur GraphQL répond précisément aux demandes, renvoyant uniquement les champs sollicités.

**Difficultés rencontrées :** La syntaxe du fichier de schéma (`.graphqls`) est stricte et doit correspondre exactement aux DTO Java.

**Solutions adoptées :** Vérification minutieuse des types dans le fichier schéma et redémarrage de l'application à chaque modification.

### 4.1.4 Code source

```
@QueryMapping
public List<CustomerDto> customers() {
    return customerService.getAllCustomers();
}
```

**Explication :** Le contrôleur mappe la requête GraphQL "customers" vers la méthode Java correspondante.

## Conclusion

Ce TP m'a montré la flexibilité de GraphQL qui résout les problèmes de sur-chargement (over-fetching) de données typiques des API REST classiques.

### 4.1.5 Lien GitHub

Lien GitHub : `https://github.com/amine-icame/architecture_de_composant`