Développement Web Full Stack

TP1 : Spring Boot

API REST pour une boutique en ligne

Ibrahim ALAME

Durée : 4 heures

Spring Boot - API REST

Entités JPA • Repositories • Services • Controllers
Relations • Validation • Exceptions

Consignes importantes

- Ce TP contient des **exercices à compléter** : vous devez écrire le code manquant
- Les zones marquées `// TODO` ou `_____` sont à compléter par vous
- Des **questions de compréhension** sont posées tout au long du TP
- Testez votre code régulièrement avec curl ou Postman
- **N'oubliez pas de rendre votre travail à la fin de la séance**

Barème :

- Partie 1 (Mise en place) : 2 points
- Partie 2 (Entité et CRUD) : 6 points
- Partie 3 (Service et Relations) : 6 points
- Partie 4 (Validation et Exceptions) : 4 points
- Questions de compréhension : 2 points

Total : 20 points

Contents

Part I
Mise en place du projet

1 Création du projet Spring Boot

1.1 Rappel : Architecture Spring Boot

Spring Boot simplifie le développement d'applications Spring :

- **@SpringBootApplication** : Active l'auto-configuration
- **Starters** : Dépendances pré-configurées (spring-boot-starter-web, etc.)
- **Serveur embarqué** : Tomcat intégré, port 8080 par défaut

Starters utilisés dans ce TP :

- `spring-boot-starter-web` : Spring MVC, REST controllers, Tomcat
- `spring-boot-starter-data-jpa` : JPA/Hibernate, accès aux données
- `spring-boot-starter-validation` : Bean Validation (JSR-380)

Exercice 1.1 : Créez un projet Spring Boot sur https://start.spring.io avec :

- Group : `com.boutique`
- Artifact : `shop-api`
- Dépendances : Spring Web, Spring Data JPA, H2 Database, Validation

Question 1 : Quelle est la différence entre les dépendances `spring-boot-starter-web` et `spring-boot-starter-data-jpa` ? À quoi sert chacune ?

2 Configuration de la base de données

Configuration H2 Database :

- `jdbc:h2:file:./data/db` : Base de données persistante (fichier)
- `jdbc:h2:mem:db` : Base de données en mémoire (perdue au redémarrage)
- `AUTO_SERVER=TRUE` : Permet plusieurs connexions simultanées (application +

outil externe)

Options ddl-auto (création du schéma) :

- `create-drop` : Crée les tables au démarrage, les supprime à l'arrêt
- `update` : Met à jour le schéma sans perdre les données existantes
- `validate` : Vérifie que le schéma correspond aux entités (ne modifie rien)
- `none` : Aucune action automatique sur le schéma

Exercice 1.2 : Complétez le fichier `application.properties` ci-dessous.

```
# Nom de l'application
spring.application.name=shop-api

# Configuration H2 - Completez les valeurs manquantes
spring.datasource.url=jdbc:h2:_________:./data/shopdb;AUTO_SERVER=TRUE
spring.datasource.driverClassName=_________
spring.datasource.username=_________
spring.datasource.password=

# Configuration JPA
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=_________
# (quelle valeur pour creer/mettre a jour les tables
    automatiquement ?)
spring.jpa.show-sql=_________
# (true ou false pour voir les requetes SQL ?)
```

Listing 1: application.properties - À compléter

Question 2 : Expliquez la différence entre `ddl-auto=create-drop`, `ddl-auto=update` et `ddl-auto=validate`. Quelle option choisiriez-vous en production ?

Question 3 : À quoi sert le paramètre `AUTO_SERVER=TRUE` dans l'URL de connexion H2 ?

Part II
Entité Produit et CRUD

3 Création de l'entité Produit

3.1 Rappel : Annotations JPA

Annotations essentielles pour les entités :

- `@Entity` : Marque la classe comme entité JPA
- `@Table(name = "...")` : Nom de la table (optionnel)
- `@Id` : Clé primaire
- `@GeneratedValue(strategy = GenerationType.IDENTITY)` : Auto-incrément
- `@Column(nullable = false, length = 100)` : Contraintes de colonne

Règle importante : Une entité JPA doit avoir un **constructeur sans argument** (vide). JPA utilise la réflexion pour créer des instances lors du chargement depuis la base de données.

Exercice 2.1 : Complétez l'entité `Produit` avec les annotations JPA appropriées.

L'entité doit avoir :

- `id` : Long, clé primaire auto-générée
- `nom` : String, obligatoire, max 100 caractères
- `description` : String, optionnel, max 500 caractères
- `prix` : Double, obligatoire
- `stock` : Integer, obligatoire, défaut 0
- `actif` : boolean, défaut true

```
1  package com.boutique.shopapi.model;
2
3  import jakarta.persistence.*;
4
5  _______ // TODO: Annotation pour marquer comme entite
6  _______ // TODO: Annotation pour nommer la table "produits"
7  public class Produit {
8
9      _______ // TODO: Annotation cle primaire
10     _______ // TODO: Annotation generation automatique
11     private Long id;
```

```
12
13      _______ // TODO: Colonne obligatoire, max 100 caracteres
14      private String nom;
15
16      _______ // TODO: Colonne optionnelle, max 500 caracteres
17      private String description;
18
19      _______ // TODO: Colonne obligatoire
20      private Double prix;
21
22      _______ // TODO: Colonne obligatoire
23      private Integer stock = 0;
24
25      private boolean actif = true;
26
27      // Constructeur vide (requis par JPA)
28      public Produit() {
29      }
30
31      // TODO: Creer un constructeur avec parametres (nom,
            description, prix, stock)
32
33
34
35
36      // TODO: Generer tous les getters et setters
37      // Utilisez Alt+Insert dans IntelliJ
38
39
40
41  }
```

Listing 2: model/Produit.java - À compléter

Question 4 : Pourquoi JPA exige-t-il un constructeur vide (sans paramètres) dans les entités ?

4 Création du Repository

4.1 Rappel : Spring Data JPA

Spring Data JPA génère automatiquement les requêtes à partir du nom des méthodes :

- `findByNom(String nom)` → `WHERE nom = ?`

- `findByPrixLessThan(Double prix)` → `WHERE prix < ?`
- `findByNomContainingIgnoreCase(String mot)` → `WHERE UPPER(nom) LIKE %MOT%`
- `findByActifTrueAndStockGreaterThan(Integer n)` → `WHERE actif = TRUE AND stock > ?`

Exercice 2.2 : Créez l'interface `ProduitRepository` et ajoutez les méthodes de recherche demandées.

```java
package com.boutique.shopapi.repository;

import com.boutique.shopapi.model.Produit;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import java.util.List;

@Repository
public interface ProduitRepository extends
    JpaRepository<_______, _______> {
    // TODO: Quels types mettre dans JpaRepository<?, ?> ?

    // TODO: Methode pour trouver les produits dont le nom
        contient un mot-cle
    // (insensible a la casse)
    List<Produit> _______________________________(String
        motCle);

    // TODO: Methode pour trouver tous les produits actifs
    List<Produit> _______________________________();

    // TODO: Methode pour trouver les produits avec un prix <=
        prixMax
    List<Produit> _______________________________(Double
        prixMax);

    // TODO: Methode pour trouver les produits actifs ayant du
        stock (stock > 0)
    List<Produit> _______________________________(Integer
        stock);

    // TODO: Methode pour trouver les produits avec un prix
        entre min et max
    List<Produit> _______________________________(Double min,
        Double max);
}
```

Listing 3: repository/ProduitRepository.java - À compléter

Question 5 : Sans écrire de code SQL, comment Spring Data JPA génère-t-il les requêtes ? Quel est l'avantage de cette approche ?

5 Création du Contrôleur REST

5.1 Rappel : API REST avec Spring MVC

Correspondance HTTP / CRUD :

HTTP	CRUD	Annotation	Code retour
GET	Read	@GetMapping	200 OK
POST	Create	@PostMapping	201 Created
PUT	Update	@PutMapping	200 OK
DELETE	Delete	@DeleteMapping	204 No Content

Codes d'erreur courants :

- **404 Not Found** : Ressource non trouvée (ID inexistant)
- **400 Bad Request** : Données invalides

Annotations importantes :

- `@PathVariable` : Extrait une variable de l'URL (`/api/produits/{id}`)
- `@RequestBody` : Convertit le JSON du corps en objet Java
- `@RequestParam` : Extrait un paramètre de requête (`?nom=value`)

Exercice 2.3 : Complétez le contrôleur REST avec les opérations CRUD.

```
1  package com.boutique.shopapi.controller;
2
3  import com.boutique.shopapi.model.Produit;
4  import com.boutique.shopapi.repository.ProduitRepository;
5  import org.springframework.beans.factory.annotation.Autowired;
6  import org.springframework.http.HttpStatus;
7  import org.springframework.http.ResponseEntity;
8  import org.springframework.web.bind.annotation.*;
9  import java.util.List;
10
11 @RestController
12 @RequestMapping("/api/produits")
13 public class ProduitController {
14
```

```
15      @Autowired
16      private ProduitRepository produitRepository;
17
18      // GET /api/produits - Recuperer tous les produits
19      @_______  // TODO: Quelle annotation ?
20      public List<Produit> getAllProduits() {
21          return ________________________; // TODO: Appel au
                repository
22      }
23
24      // GET /api/produits/{id} - Recuperer un produit par ID
25      @GetMapping("/{id}")
26      public ResponseEntity<Produit> getProduitById(_______ Long
            id) {
27          // TODO: Quelle annotation pour extraire {id} de l'URL ?
28
29          return produitRepository.findById(id)
30                  .map(produit -> ResponseEntity.ok(produit))
31                  .orElse(________________________); // TODO: Que
                       retourner si non trouve ?
32      }
33
34      // POST /api/produits - Creer un nouveau produit
35      @_______  // TODO: Quelle annotation ?
36      @ResponseStatus(HttpStatus._______)  // TODO: Quel code
            HTTP pour une creation ?
37      public Produit createProduit(_______ Produit produit) {
38          // TODO: Quelle annotation pour deserialiser le JSON ?
39          return ________________________; // TODO: Sauvegarder le
                produit
40      }
41
42      // PUT /api/produits/{id} - Modifier un produit
43      @PutMapping("/{id}")
44      public ResponseEntity<Produit> updateProduit(
45              @PathVariable Long id,
46              @RequestBody Produit produitDetails) {
47
48          return produitRepository.findById(id)
49                  .map(produit -> {
50                      // TODO: Mettre a jour tous les champs du
                            produit
51                      produit.setNom(________________________);
52                      ________________________
53                      ________________________
54                      ________________________
55                      ________________________
56
57                      Produit updated =
                            produitRepository.save(produit);
58                      return ResponseEntity.ok(updated);
```

```
59                  })
60                  .orElse(ResponseEntity.notFound().build());
61      }
62
63      // DELETE /api/produits/{id} - Supprimer un produit
64      @_______("/{id}")  // TODO: Quelle annotation ?
65      public ResponseEntity<Void> deleteProduit(@PathVariable
           Long id) {
66          if (produitRepository.existsById(id)) {
67              ______________________  // TODO: Supprimer le
                   produit
68              return ResponseEntity.noContent().build();
69          }
70          return ______________________; // TODO: Que retourner
               si non trouve ?
71      }
72  }
```

Listing 4: controller/ProduitController.java - À compléter

Exercice 2.4 : Ajoutez les endpoints de recherche suivants au contrôleur (écrivez le code complet vous-même) :

1. `GET /api/produits/recherche?nom=xxx` : Recherche par nom
2. `GET /api/produits/disponibles` : Produits actifs en stock
3. `GET /api/produits/prix?min=x&max=y` : Produits par gamme de prix

6 Tests de l'API

Exercice 2.5 : Testez votre API en exécutant les commandes curl suivantes. Notez les résultats.

```
# 1. Creer un produit
curl -X POST http://localhost:8080/api/produits \
  -H "Content-Type: application/json" \
  -d '{"nom": "Smartphone", "description": "Test", "prix":
     599.99, "stock": 10}'

# 2. Lister tous les produits
```

```
curl http://localhost:8080/api/produits

# 3. Recuperer le produit avec ID 1
curl http://localhost:8080/api/produits/1

# 4. Tester un ID inexistant
curl -v http://localhost:8080/api/produits/999
```

Listing 5: Commandes de test

Question 6 : Quel code de statut HTTP recevez-vous pour chacune de ces requêtes ? Expliquez pourquoi.

- Création (POST) : _____
- Liste (GET tous) : _____
- ID existant (GET) : _____
- ID inexistant (GET) : _____

Part III
Architecture en couches

7 Couche Service

7.1 Rappel : Séparation des responsabilités

Architecture en 3 couches :

- **Controller** : Gère les requêtes HTTP (pas de logique métier)
- **Service** : Contient la logique métier et les règles de gestion
- **Repository** : Accès aux données uniquement

Annotations importantes :

- `@Service` : Marque une classe comme service Spring (composant métier)
- `@Transactional` : Gère les transactions (rollback automatique en cas d'erreur)
- `@Autowired` : Injection de dépendances automatique

Gestion des erreurs métier :

- `IllegalArgumentException` : Argument invalide (ex: quantité négative)
- `IllegalStateException` : État invalide (ex: stock insuffisant pour retrait)

Exercice 3.1 : Créez la classe `ProduitService` avec la logique métier suivante :

- Vérifier que le prix est positif avant de créer un produit
- Méthode `ajouterStock(id, quantite)` : ajoute du stock
- Méthode `retirerStock(id, quantite)` : retire du stock (avec vérification)

```
1  package com.boutique.shopapi.service;
2
3  import com.boutique.shopapi.model.Produit;
4  import com.boutique.shopapi.repository.ProduitRepository;
5  import org.springframework.beans.factory.annotation.Autowired;
6  import org.springframework.stereotype.Service;
7  import org.springframework.transaction.annotation.Transactional;
8  import java.util.List;
9  import java.util.Optional;
10
11 @_______ // TODO: Quelle annotation pour un service ?
12 @Transactional
13 public class ProduitService {
14
15     @Autowired
16     private ProduitRepository produitRepository;
17
18     public List<Produit> getAllProduits() {
19         return produitRepository.findAll();
20     }
21
22     public Optional<Produit> getProduitById(Long id) {
23         return produitRepository.findById(id);
24     }
25
26     public Produit createProduit(Produit produit) {
27         // TODO: Verifier que le prix est positif, sinon lever
               une exception
28         if (________________________) {
29             throw new IllegalArgumentException("Le prix doit
                   etre positif");
30         }
31
32         // TODO: Initialiser le stock a 0 si null
33         if (________________________) {
34             ________________________
35         }
36
```

```
37          return produitRepository.save(produit);
38      }
39
40      /**
41       * Ajouter du stock a un produit
42       */
43      public Produit ajouterStock(Long id, Integer quantite) {
44          // TODO: Verifier que la quantite est positive
45
46
47          // TODO: Recuperer le produit ou lever une exception si
                non trouve
48          Produit produit = ________________________
49
50          // TODO: Ajouter la quantite au stock actuel
51
52
53          // TODO: Sauvegarder et retourner le produit
54
55      }
56
57      /**
58       * Retirer du stock (pour une vente)
59       */
60      public Produit retirerStock(Long id, Integer quantite) {
61          // TODO: Verifier que la quantite est positive
62
63
64          // TODO: Recuperer le produit
65
66
67          // TODO: Verifier que le stock est suffisant, sinon
                lever IllegalStateException
68
69
70          // TODO: Retirer la quantite du stock
71
72
73          // TODO: Sauvegarder et retourner
74
75      }
76  }
```

Listing 6: service/ProduitService.java - À compléter

Question 7 : Quelle est la différence entre `IllegalArgumentException` et `IllegalStateException` ? Dans quels cas utiliser chacune ?

Exercice 3.2 : Modifiez `ProduitController` pour utiliser `ProduitService` au lieu de `ProduitRepository` directement. Ajoutez aussi les endpoints :

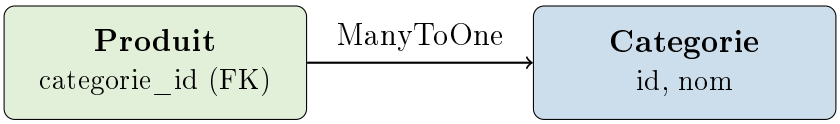
- `PATCH /api/produits/{id}/stock/ajouter?quantite=X`
- `PATCH /api/produits/{id}/stock/retirer?quantite=X`

8 Entité Catégorie avec relation

8.1 Rappel : Relations JPA

Relation ManyToOne / OneToMany :
Un produit appartient à **une** catégorie, une catégorie contient **plusieurs** produits.

- Côté "Many" (Produit) : `@ManyToOne` + `@JoinColumn`
- Côté "One" (Catégorie) : `@OneToMany(mappedBy = "categorie")`
- `@JsonIgnore` : Évite la récursion infinie lors de la sérialisation JSON



Exercice 3.3 : Créez l'entité `Categorie` avec les champs : id, nom (unique), description.

Exercice 3.4 : Ajoutez la relation ManyToOne dans `Produit` vers `Categorie`.

```
// TODO: Ajouter dans la classe Produit

_______ // Annotation pour relation Many-to-One
_______ // Annotation pour la colonne de cle etrangere "categorie_id"
private Categorie categorie;

// + Getter et Setter
```

Listing 7: Ajout dans Produit.java

Exercice 3.5 : Créez `CategorieRepository`, `CategorieService` et `CategorieController` (CRUD complet). Écrivez le code vous-même en vous inspirant des classes Produit.

Question 8 : Si on ne met pas `@JsonIgnore` sur la liste des produits dans Catégorie, que se passe-t-il lors de la sérialisation JSON ? Pourquoi ?

Part IV
Validation et Exceptions

9 Validation avec Bean Validation

9.1 Rappel : Annotations de validation

Annotations de validation (jakarta.validation.constraints) :

`@NotNull`	Ne doit pas être null
`@NotEmpty`	Non null et taille > 0 (String, Collection)
`@NotBlank`	Non null, non vide, au moins 1 caractère non-blanc (String)
`@Size(min, max)`	Taille entre min et max
`@Min(value)` / `@Max(value)`	Valeur min/max
`@Positive`	Strictement positif
`@PositiveOrZero`	Positif ou zéro
`@Email`	Format email valide

Pour activer : ajoutez `@Valid` devant `@RequestBody` dans le contrôleur.

Exercice 4.1 : Ajoutez les annotations de validation sur l'entité Produit :

- nom : obligatoire, entre 2 et 100 caractères
- prix : obligatoire et strictement positif
- stock : obligatoire et positif ou zéro

```
// TODO: Ajoutez les imports necessaires
import jakarta.validation.constraints.*;

// Dans la classe Produit :

_______(message = "Le nom est obligatoire")
_______(min = 2, max = 100, message = "Le nom doit faire entre
    2 et 100 caracteres")
@Column(nullable = false, length = 100)
private String nom;

_______(message = "Le prix est obligatoire")
_______(message = "Le prix doit etre positif")
@Column(nullable = false)
private Double prix;

_______(message = "Le stock est obligatoire")
_______(message = "Le stock ne peut pas etre negatif")
@Column(nullable = false)
private Integer stock = 0;
```

Listing 8: Validation dans Produit.java

Exercice 4.2 : Activez la validation dans le contrôleur en ajoutant `@Valid`.

Question 9 : Quelle est la différence entre `@NotNull`, `@NotEmpty` et `@NotBlank` ?

10 Gestion globale des exceptions

Gestion centralisée des exceptions avec Spring :

- `@RestControllerAdvice` : Classe qui intercepte les exceptions de tous les contrôleurs
- `@ExceptionHandler(TypeException.class)` : Méthode qui gère un type d'exception spécifique

Codes HTTP recommandés :

- **400 Bad Request** : Données invalides (validation, argument incorrect)
- **404 Not Found** : Ressource non trouvée
- **409 Conflict** : Conflit d'état (ex: stock insuffisant)
- **500 Internal Server Error** : Erreur serveur inattendue

Exercice 4.3 : Créez une classe `GlobalExceptionHandler` qui :

- Gère `IllegalArgumentException` → retourne 400 Bad Request
- Gère `IllegalStateException` → retourne 409 Conflict
- Gère les erreurs de validation → retourne 400 avec la liste des erreurs

```java
package com.boutique.shopapi.exception;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import
   org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.*;
import java.util.*;

_______ // TODO: Quelle annotation pour un gestionnaire global ?
public class GlobalExceptionHandler {

    // Gestion des IllegalArgumentException
    @ExceptionHandler(_______________________) // TODO: Quelle
       classe ?
    public ResponseEntity<Map<String, Object>> handleBadRequest(
            IllegalArgumentException ex) {

        Map<String, Object> body = new HashMap<>();
        body.put("status", _______); // TODO: Code HTTP
        body.put("error", "Bad Request");
        body.put("message", ex.getMessage());

        return ResponseEntity.status(_______).body(body);
    }

    // TODO: Ajouter la gestion de IllegalStateException (409
       Conflict)









    // Gestion des erreurs de validation
```

```java
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, Object>> handleValidation(
            MethodArgumentNotValidException ex) {

        Map<String, String> errors = new HashMap<>();
        // TODO: Parcourir les erreurs et les ajouter a la map
        // ex.getBindingResult().getFieldErrors() retourne la
           liste des erreurs



        Map<String, Object> body = new HashMap<>();
        body.put("status", 400);
        body.put("error", "Validation Error");
        body.put("errors", errors);

        return ResponseEntity.badRequest().body(body);
    }
}
```

Listing 9: exception/GlobalExceptionHandler.java - À compléter

11 Tests finaux

Exercice 4.4 : Testez les cas d'erreur et vérifiez les réponses JSON.

```bash
# Test validation : prix negatif
curl -X POST http://localhost:8080/api/produits \
  -H "Content-Type: application/json" \
  -d '{"nom": "Test", "prix": -10, "stock": 5}'

# Test validation : nom vide
curl -X POST http://localhost:8080/api/produits \
  -H "Content-Type: application/json" \
  -d '{"nom": "", "prix": 100, "stock": 5}'

# Test stock insuffisant
curl -X PATCH
   "http://localhost:8080/api/produits/1/stock/retirer?quantite=9999"
```

Listing 10: Tests des erreurs

Question 10 : Pourquoi est-il important d'avoir des messages d'erreur clairs et structurés dans une API REST ? Donnez un exemple de bonne et mauvaise pratique.

Part V
Extension (Bonus)

Extension : Si vous avez terminé en avance, implémentez un système de commandes :

- Entité `Commande` : id, dateCommande, statut (enum : EN_ATTENTE, CONFIRMEE, EXPEDIEE, LIVREE, ANNULEE)
- Entité `LigneCommande` : id, quantite, prixUnitaire, produit, commande
- Endpoint pour créer une commande avec plusieurs lignes
- Lors de la création, vérifier le stock et le décrémenter

Bon courage !

Fin du TP1