

TP3 Full-Stack

Formation : Master 2 Génie Informatique Logiciel

Université : Université de Rouen Normandie

Étudiants :

Mahad MOUSSA ABDILLAHI

Encadrante : Hugo Mochet

Dépôt GitHub :

<https://github.com/mahad253/tp3-fullstack>

Année universitaire : 2025–2026

Table des matières

1	Introduction	2
2	Montée de version Spring Boot et Java	2
2.1	Modification du fichier pom.xml	2
3	Migration Javax vers Jakarta	3
4	Mise en place Docker	3
5	Problème du séparateur décimal et Structure de la base de données	4
6	Reconstruction complète de la base	4
7	Premiers tests API	4
8	Gestion des horaires d'ouverture	5
8.1	Annotation NoOverlappingOpeningHours	5
8.2	Valdateur NoOverlappingOpeningHoursValidator	5
9	Intégration d'Elasticsearch pour la recherche plein texte	7
9.1	Ajout des dépendances Hibernate Search	7
9.2	Ajout du service Elasticsearch dans Docker	7
9.3	Configuration Spring Boot	8
9.4	Indexation de l'entité Shop	8
9.5	Service ElasticSearchService	8
9.6	Reindexation automatique au démarrage	9
9.7	Intégration dans ShopService	9
9.8	Visuel de l'Elasticsearch	9
10	Problème rencontré	10
10.1	Correction	10
10.2	Architecture finale	10
11	Frontend – Interface responsive	10
11.1	Principe général	11
11.2	Gestion des marges	11
11.3	Adaptation de l'affichage	12
12	Conclusion	12

1 Introduction

Dans ce TP3 Full-Stack, j'ai travaillé sur une application backend Spring Boot fournie au départ, permettant de gérer des boutiques, des produits, des catégories ainsi que des horaires d'ouverture.

Le projet utilise PostgreSQL comme base de données et Docker pour l'orchestration des services.

Mon objectif principal était d'adapter le projet aux nouvelles versions techniques, corriger les problèmes existants, respecter les contraintes métier et stabiliser entièrement le backend.

J'ai commencé par lancer le projet tel quel afin de comprendre son fonctionnement général, puis j'ai progressivement appliqué les différentes exigences.

2 Montée de version Spring Boot et Java

Le projet initial utilisait Java 11 et une ancienne version de Spring Boot. J'ai décidé de migrer vers :

- Spring Boot 3.5
- Java 21

Cette étape était nécessaire pour travailler avec un environnement moderne et compatible avec Jakarta.

2.1 Modification du fichier pom.xml

J'ai commencé par modifier le parent Spring Boot :

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.5.0</version>
</parent>
```

Puis j'ai mis à jour la version Java :

```
<properties>
  <java.version>21</java.version>
</properties>
```

Ensuite j'ai vérifié les dépendances principales :

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.7.5</version>
</dependency>
```

Après ces changements, le projet ne compilait plus à cause des imports Javax.

3 Migration Javax vers Jakarta

Spring Boot 3 n'utilise plus javax mais jakarta.

J'ai donc remplacé tous les imports dans l'ensemble du projet.

Exemple dans Shop.java :

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Column;
import jakarta.validation.constraints.NotNull;
```

Cette modification concernait toutes les entités, services et validateurs.

Sans cette étape, le backend refusait de démarrer.

4 Mise en place Docker

Le projet est lancé avec Docker Compose.

Les services principaux sont :

- PostgreSQL
- Backend Spring Boot
- Frontend Angular

Je lance le projet avec :

```
docker compose up --build
```

Lorsque des erreurs apparaissaient, je supprimais entièrement les volumes :

```
docker compose down -v
docker compose up --build
```

Cela permet de repartir sur une base propre.

5 Problème du séparateur décimal et Structure de la base de données

Les données sont injectées via le fichier `fill_tables.sql`.

Une exigence importante était la non-utilisation de la virgule dans la base de données.

Au début, lorsque j'entrais 12,22 le système interprétait 12.22.

J'ai corrigé cela afin que le backend respecte le format de l'exigence :

- En divisant par 100 et à ne plus mettre des chiffres décimaux dans le backend.
- Et en modifiant les prix des produits dans `fill_tables.sql` car dans mon code il les divise désormais par 100.

Exemple d'insertion :

```
insert into products (id, price, shop_id) values (122, 199, 1);
```

Les prix sont stockés en centimes.

199 correspond donc à 1,99€.

6 Reconstruction complète de la base

Après toutes les modifications SQL, j'ai recréé entièrement la base :

```
docker compose down -v  
docker compose up --build
```

Cela garantit que toutes les données sont propres et cohérentes.

7 Premiers tests API

Une fois le backend fonctionnel, j'ai testé les endpoints avec le navigateur et Postman.

J'ai vérifié :

- création boutique
- suppression boutique
- filtres
- tris
- relations produits

À ce stade, le backend fonctionnait correctement.

8 Gestion des horaires d'ouverture

Une contrainte importante du TP est d'empêcher le chevauchement des horaires d'ouverture d'une boutique sur un même jour.

Pour cela, j'ai ajouté une validation personnalisée basée sur Bean Validation.

J'ai commencé par créer une annotation dédiée.

8.1 Annotation NoOverlappingOpeningHours

```
package fr.fullstack.shopapp.validation;

import jakarta.validation.Constraint;
import jakarta.validation.Payload;
import java.lang.annotation.*;

@Documented
@Constraint(validatedBy = NoOverlappingOpeningHoursValidator.class)
@Target({ ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
public @interface NoOverlappingOpeningHours {

    String message() default "Chevauchement d horaires d tect ";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

}
```

Cette annotation est placée directement sur la liste des horaires dans l'entité Shop.

Elle permet d'appeler automatiquement le validateur associé lors de la création ou modification d'une boutique.

8.2 Validateur NoOverlappingOpeningHoursValidator

Ensuite, j'ai implémenté la classe de validation qui vérifie concrètement les horaires.

```
package fr.fullstack.shopapp.validation;

import fr.fullstack.shopapp.model.OpeningHoursShop;
import jakarta.validation.ConstraintValidator;
import jakarta.validation.ConstraintValidatorContext;

import java.util.List;

public class NoOverlappingOpeningHoursValidator
    implements ConstraintValidator<NoOverlappingOpeningHours,
        List<OpeningHoursShop>> {

    @Override
    public boolean isValid(List<OpeningHoursShop> hours,
        ConstraintValidatorContext context) {
```

```
    if (hours == null || hours.isEmpty()) {
        return true;
    }

    for (int i = 0; i < hours.size(); i++) {
        OpeningHoursShop a = hours.get(i);

        if (a.getOpenAt() == null || a.getCloseAt() == null) {
            continue;
        }

        for (int j = i + 1; j < hours.size(); j++) {
            OpeningHoursShop b = hours.get(j);

            if (b.getOpenAt() == null || b.getCloseAt() == null) {
                continue;
            }

            if (a.getDay() == b.getDay()) {

                boolean overlap =
                    !(a.getCloseAt().isBefore(b.getOpenAt())
                    || a.getOpenAt().isAfter(b.
                        getCloseAt()));

                if (overlap) {
                    return false;
                }
            }
        }
    }

    return true;
}
```

Le validateur parcourt tous les horaires deux par deux.

Pour chaque jour identique, il vérifie si les plages horaires se recouvrent.

Si un chevauchement est détecté, la validation échoue immédiatement.

9 Intégration d'Elasticsearch pour la recherche plein texte

Dans cette partie, j'ai intégré Elasticsearch au projet afin d'ajouter une fonctionnalité de recherche plein texte sur les boutiques.

PostgreSQL reste la base principale, mais Elasticsearch est utilisé uniquement pour la recherche rapide et tolérante aux fautes.

L'objectif est de pouvoir rechercher une boutique par son nom, même avec des erreurs de frappe.

9.1 Ajout des dépendances Hibernate Search

J'ai commencé par ajouter les dépendances nécessaires dans le fichier pom.xml :

```
<dependency>
  <groupId>org.hibernate.search</groupId>
  <artifactId>hibernate-search-mapper-orm</artifactId>
  <version>7.2.1.Final</version>
</dependency>

<dependency>
  <groupId>org.hibernate.search</groupId>
  <artifactId>hibernate-search-engine</artifactId>
  <version>7.2.1.Final</version>
</dependency>

<dependency>
  <groupId>org.hibernate.search</groupId>
  <artifactId>hibernate-search-backend-elasticsearch</artifactId>
  <version>7.2.1.Final</version>
</dependency>
```

Ces bibliothèques permettent à Hibernate de communiquer directement avec Elasticsearch.

9.2 Ajout du service Elasticsearch dans Docker

J'ai ensuite ajouté Elasticsearch dans docker-compose.yml :

```
elasticsearch:
  image: docker.elastic.co/elasticsearch/elasticsearch:7.17.25
  environment:
    discovery.type: single-node
  ports:
    - "9200:9200"
```

L'API Spring Boot démarre uniquement lorsque PostgreSQL et Elasticsearch sont prêts grâce à :

```
depends_on:
  db:
    condition: service_healthy
  elasticsearch:
    condition: service_healthy
```


Cela évite les erreurs de connexion au démarrage.

9.3 Configuration Spring Boot

Dans `application.properties`, j'ai configuré Hibernate Search :

```
spring.jpa.properties.hibernate.search.backend.protocol=http
spring.jpa.properties.hibernate.search.backend.hosts=\${ES_URL}

elasticsearch.reindex-on-startup=false
```

Ainsi, Spring communique avec Elasticsearch via Docker, et une réindexation est automatiquement déclenchée au démarrage.

9.4 Indexation de l'entité Shop

Pour rendre une entité recherchable, j'ai annoté Shop :

```
@Indexed(index = "idx_shops")
public class Shop
```

Puis j'ai marqué le champ `name` comme champ plein texte :

```
@FullTextField
private String name;
```

Cela permet à Elasticsearch d'indexer le nom des boutiques.

9.5 Service ElasticSearchService

J'ai créé un service dédié.

Réindexation complète

```
@Transactional
public void reindexAll() throws InterruptedException {

    Session session = entityManager.unwrap(Session.class);
    SearchSession searchSession = Search.session(session);

    searchSession.massIndexer(Shop.class)
        .threadsToLoadObjects(5)
        .batchSizeToLoadObjects(25)
        .startAndWait();
}
```

Cette méthode synchronise toutes les boutiques PostgreSQL vers Elasticsearch.

Recherche plein texte

```
List<Shop> results = searchSession.search(Shop.class)
    .where(f -> f.match()
        .field("name")
        .matching(searchQuery.trim())
        .fuzzy(2))
    .fetch(...)
```

L'option `fuzzy(2)` permet de supporter les fautes de frappe.

9.6 Reindexation automatique au démarrage

J'ai créé une classe `ElasticStartupLoader` :

```
@Component
public class ElasticStartupLoader implements ApplicationRunner {

    @Override
    public void run(ApplicationArguments args) throws Exception {
        indexService.reindexAll();
    }
}
```

À chaque lancement du backend :

PostgreSQL → Elasticsearch

Cela garantit que l'index est toujours à jour.

9.7 Intégration dans ShopService

Dans le service principal :

```
if (label.isPresent()) {
    return elasticSearchService.searchShops(label.get(), pageable);
}
```

Si un label est fourni, la recherche passe par Elasticsearch.

Sinon, les requêtes utilisent PostgreSQL normalement.

9.8 Visuel de l'Elasticsearch

Afin de vérifier le bon fonctionnement d'Elasticsearch et la présence des données indexées par l'application, j'ai utilisé la commande suivante :

```
curl localhost:9200/_cat/indices?v
```

Cette commande permet d'afficher la liste des index présents dans le cluster Elasticsearch ainsi que leur état (health), leur statut, le nombre de documents indexés et l'espace de stockage utilisé.

```
(base) mahad@mahad-Latitude-5540:~/Bureau/tp3-fullstack$ curl localhost:9200/_cat/indices?v
health status index      uuid                                pri rep docs.count docs.deleted store.size pri.store.size
green  open   .geoip_databases Hz65PHQ-T8m0iSQzxCGJ2g  1   0         41           0      39.4mb      39.4mb
yellow open   idx_shops-000001 W0SzJY-uSdiltaowq4I0pg  1   1          10           0      16.1kb      16.1kb
```

Dans notre cas, on observe notamment l'index `idx_shops-000001` correspondant aux boutiques de l'application. La présence de cet index ainsi que le nombre de documents associés confirme que les données sont correctement indexées dans Elasticsearch.

L'état `yellow` de l'index est normal dans un environnement local, car le cluster ne comporte qu'un seul nœud et ne peut donc pas allouer le shard de réplica. En environnement de production avec plusieurs nœuds, cet état passerait à `green`.

10 Problème rencontré

Lors des tests, j'ai rencontré une erreur intermittente :

```
Transaction silently rolled back because it has been marked as
rollback-only
```

Après analyse, le problème venait du fait que les données étaient insérées manuellement via SQL. Hibernate ne connaissait donc pas la dernière valeur des IDs, ce qui provoquait parfois des conflits de clé primaire.

10.1 Correction

La solution a été d'utiliser une génération d'ID basée sur PostgreSQL :

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

Ainsi, Hibernate s'appuie directement sur la séquence PostgreSQL.

Depuis cette correction, les créations des boutiques avec horaires sont devenues stables.

10.2 Architecture finale

L'architecture obtenue est :

Frontend → Spring Boot → Hibernate Search → Elasticsearch

PostgreSQL reste la base principale, Elasticsearch est utilisé uniquement pour la recherche.

11 Frontend – Interface responsive

Dans cette partie, je présente le travail réalisé sur le rendu responsive de l'interface front-end. L'objectif était de permettre une utilisation correcte de l'application aussi bien sur ordinateur que sur tablette ou téléphone mobile.

Pour cela, je me suis appuyé sur des règles CSS simples, principalement à l'aide de *media queries*, afin d'adapter les marges et l'organisation du contenu en fonction de la taille de l'écran.

11.1 Principe général

Sur les écrans larges (ordinateur), j'ai choisi de conserver des marges importantes afin d'améliorer la lisibilité du contenu. En revanche, lorsque la taille de l'écran diminue, ces marges sont progressivement réduites afin d'optimiser l'espace disponible et d'éviter les débordements.

Cette logique est définie dans le fichier `App.css`, qui gère la mise en page globale de l'application.

11.2 Gestion des marges

Par défaut, sur un écran de type desktop, l'application est centrée avec des marges importantes :

```
.App {  
  margin: 7% 20%;  
}  
  
.header {  
  margin: 0 20%;  
}
```

Pour les écrans de type tablette, j'ai réduit ces marges afin d'occuper une plus grande surface d'affichage :

```
@media (max-width: 1024px) {  
  .App {  
    margin: 5% 10%;  
  }  
  
  .header {  
    margin: 0 10%;  
  }  
}
```

Sur les écrans mobiles, l'interface adopte une disposition presque plein écran, ce qui améliore la lisibilité et l'ergonomie :

```
@media (max-width: 768px) {  
  .App {  
    margin: 5% 5%;  
  }  
  
  .header {  
    margin: 0 5%;  
  }  
}
```

Enfin, pour les très petits écrans, j'ai encore réduit les marges afin d'exploiter au maximum l'espace disponible :

```
@media (max-width: 480px) {  
  .App {
```

```
        margin: 4\% 3\%;  
    }  
  
    .header {  
        margin: 0 3\%;  
    }  
}
```

11.3 Adaptation de l’affichage

En complément de la gestion des marges, l’affichage des composants (cartes de boutiques, produits et filtres) s’adapte automatiquement sur les écrans mobiles. Les éléments sont affichés de manière plus verticale, ce qui facilite la navigation tactile et améliore l’expérience utilisateur.

12 Conclusion

Ce TP3 m’a permis de renforcer mes compétences en développement Full-Stack, notamment sur la partie backend avec Spring Boot, la gestion de PostgreSQL via Docker et l’intégration d’Elasticsearch pour la recherche plein texte.

J’ai obtenu une application fonctionnelle et conforme au sujet, avec une architecture claire et stable. Le code source est disponible sur GitHub :

<https://github.com/mahad253/tp3-fullstack>.

Le léger retard de livraison (un jour) est dû à un problème technique lié à la gestion des horaires d’ouverture des boutiques, ainsi qu’à des contraintes personnelles. L’ensemble des TP2 et TP3 a été réalisé intégralement seul.