TP 1 : Création d'une application Web de Gestion de Produits

Spring Boot (Backend) & Angular (Frontend)

Auteur : Saoudi Haythem

Matière : JEE2

Classe : 5ème Année Génie Logiciel

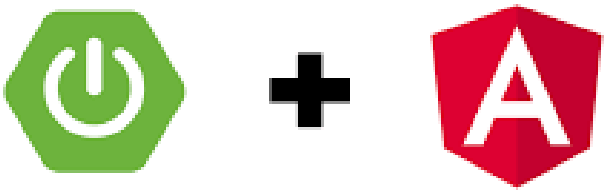

Table des matières

Première partie

Partie I : Développement du Backend avec Spring Boot

1 Introduction aux Frameworks

1.1 Présentation de Spring et Spring Boot

Spring Framework est l'un des frameworks les plus populaires de l'écosystème Java pour le développement d'applications d'entreprise robustes et performantes. Il fournit un support complet pour le développement d'applications, de la couche de persistance des données à la couche de présentation.

Un concept fondamental dans Spring est **l'Inversion de Contrôle (IoC)**. Au lieu que le développeur gère manuellement la création et le cycle de vie des objets (les dépendances), le framework Spring s'en charge. Cela signifie que le développeur peut se concentrer sur la logique métier de l'application, tandis que les aspects techniques, comme l'instanciation des composants et l'injection des dépendances, sont délégués au framework.

Spring Boot est une extension du framework Spring qui simplifie radicalement le processus de création et de déploiement d'applications basées sur Spring. Son objectif principal est de réduire la configuration manuelle et d'accélérer le développement grâce à :

- **La configuration automatique :** Spring Boot configure intelligemment l'application en fonction des dépendances présentes dans le projet.
- **Les "Starters" :** Des dépendances pré-configurées pour des besoins spécifiques (web, data, sécurité, etc.).
- **Le serveur embarqué :** Plus besoin de déployer un fichier WAR, l'application peut être lancée comme une simple application Java.

En résumé, la formule est simple : **Spring Boot = Spring Framework + Configuration Automatique + Outils de Développement**.

1.2 Spring vs Spring Boot : Tableau comparatif

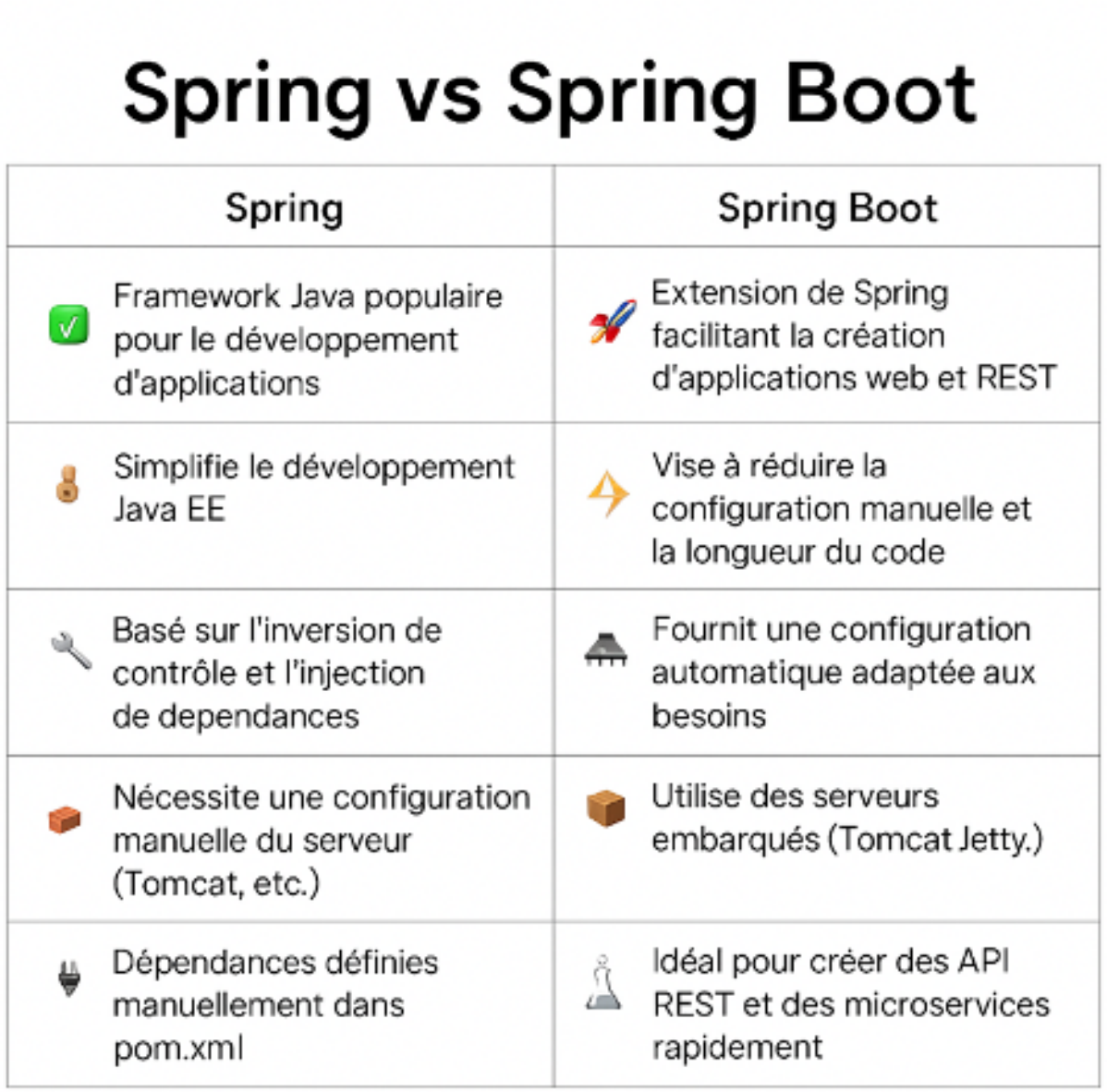

FIGURE 1 – Comparaison des caractéristiques entre Spring et Spring Boot.

2 Objectif du Travail Pratique

L'objectif de ce TP est de créer une application web *full-stack* de gestion de produits et de catégories. Pour cela, nous utiliserons :

— **Spring Boot** pour le développement du **backend** (API RESTful).

— **Angular** pour le développement du **frontend** (interface utilisateur).

2.1 Backend (Spring Boot)

— **API RESTful :** Développer une API RESTful pour la gestion complète des produits et des catégories, permettant les opérations CRUD (Créer, Lire, Mettre à jour, Supprimer).

— **Modélisation des données :** Concevoir une base de données relationnelle (ex : MySQL) et établir une relation *one-to-many* (une catégorie peut avoir plusieurs produits).

2.2 Frontend (Angular)

— **Interface Utilisateur :** Créer une interface utilisateur intuitive pour interagir avec le backend, permettant de gérer les catégories et les produits.

— **Interaction avec l'API :** Utiliser le module `HttpClient` d'Angular pour communiquer avec l'API Spring Boot.

— **Composants et Routage :** Structurer l'application avec des composants dédiés et un système de routage pour une navigation fluide.

3 Architecture de l'Application

L'application suivra une architecture en couches, une bonne pratique dans le développement logiciel, qui sépare clairement les responsabilités.

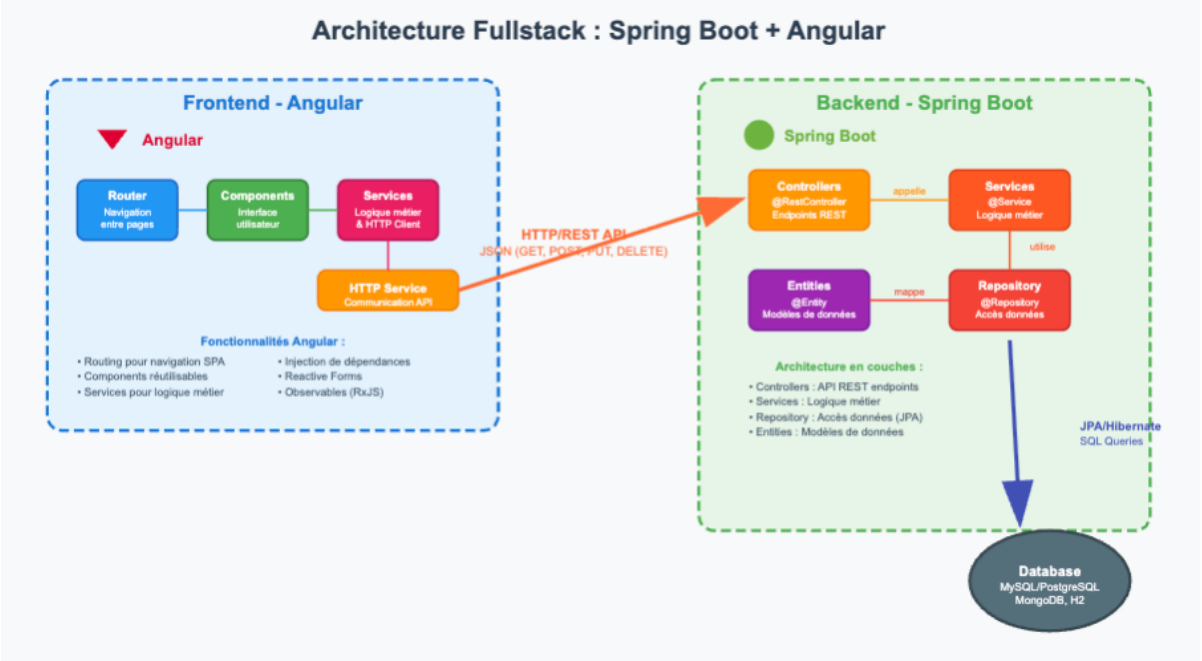

FIGURE 2 – Architecture Fullstack : Angular (Frontend) + Spring Boot (Backend).

— **Couche de Présentation (Frontend - Angular) :** Gère l'interaction avec l'utilisateur.

— **Couche Contrôleur (Backend - Spring Boot) :** Expose les points d'entrée de l'API REST (*endpoints*).

— **Couche Service (Backend) :** Contient la logique métier de l'application.

— **Couche d'Accès aux Données (Backend - Repository) :** Gère la communication avec la base de données.

— **Couche Entités (Backend) :** Représente les tables de la base de données.

4 Mise en Place du Projet Backend

4.1 Création du projet avec Spring Initializr

La première étape consiste à générer le squelette de notre projet Spring Boot via l'outil en ligne **Spring Initializr**.

1. Rendez-vous sur https://start.spring.io/.
2. Configurez le projet avec les métadonnées suivantes :
   - **Project :** Maven
   - **Language :** Java
   - **Group :** com.poly
   - **Artifact :** gestioncatalogue
3. Ajoutez les dépendances nécessaires :
   - **Lombok :** Pour réduire le code répétitif (getters, setters, etc.).
   - **Spring Web :** Pour créer des applications web et des API RESTful.
   - **Spring Data JPA :** Pour simplifier l'accès aux données.
   - **MySQL Driver :** Le pilote JDBC pour communiquer avec une base de données MySQL.
4. Cliquez sur "GENERATE" pour télécharger le projet.

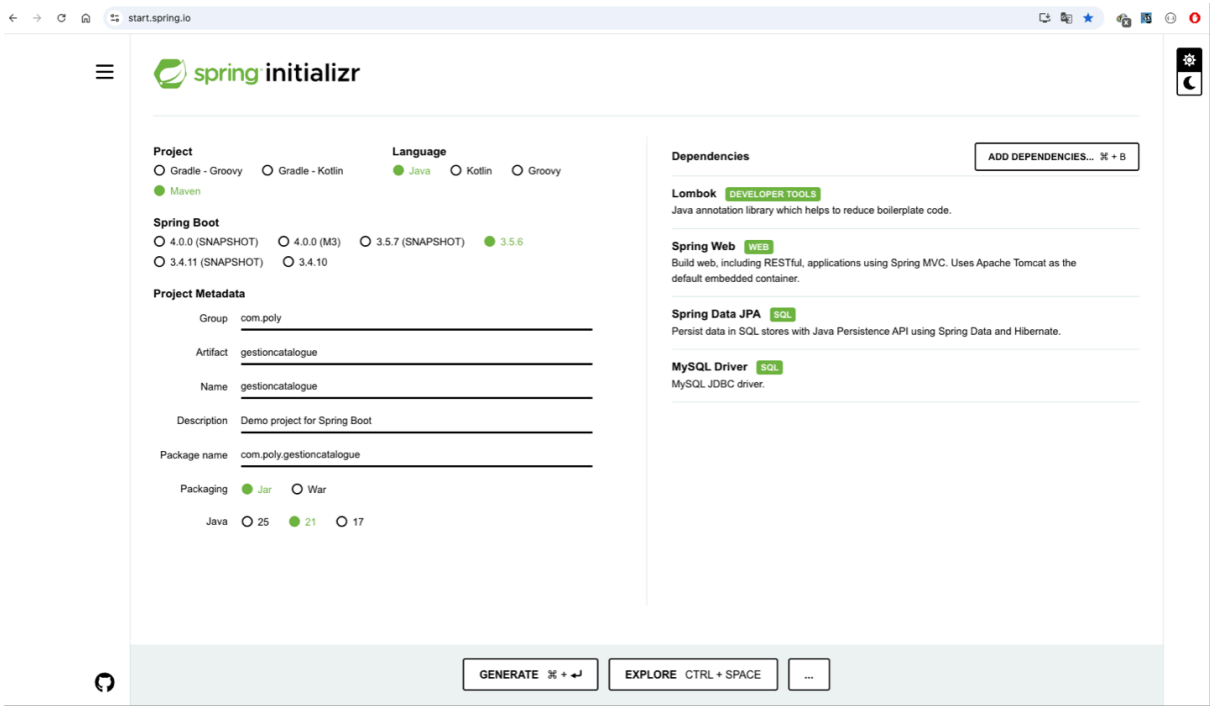

FIGURE 3 – Configuration du projet sur Spring Initializr.

4.2 Structure du Projet

Une fois le projet importé dans votre IDE (Eclipse ou IntelliJ), vous obtiendrez une arborescence de projet bien définie, comme décrit dans la figure ci-dessous.

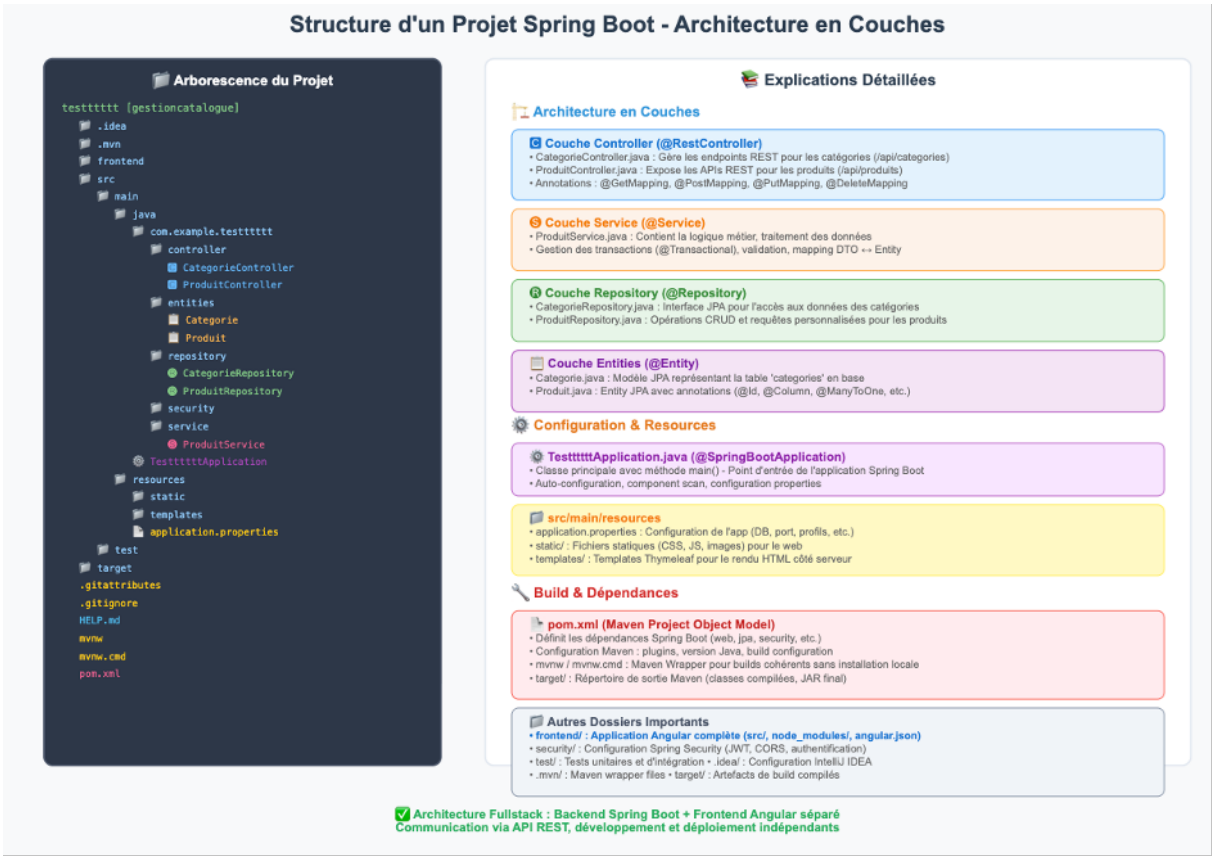

FIGURE 4 – Structure d'un projet Spring Boot avec une architecture en couches.

5 Étape 1 : Définition des Entités

Les entités sont des classes Java qui représentent les tables de notre base de données. Nous utiliserons les annotations de **JPA (Java Persistence API)** pour le mappage objet-relationnel et **Lombok** pour réduire le code standard.

5.1 Optimisation avec Lombok

Lombok est une bibliothèque qui se branche à votre processus de build et génère automatiquement du code Java.

- `@Data` : Génère les getters, setters, `toString()`, `equals()` et `hashCode()`.
- `@NoArgsConstructor` : Génère un constructeur sans arguments.
- `@AllArgsConstructor` : Génère un constructeur avec tous les champs comme arguments.
- `@Builder` : Implémente le design pattern Builder pour une construction d'objets plus lisible.

5.2 Entité `Produit`

Cette entité représente un produit dans notre catalogue.

```
1 package com.example.entities;
2
3 import jakarta.persistence.*;
4 import lombok.*;
```

```java

@Entity
@Table(name = "produits")
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class Produit {

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

    private String nom;
    private double prix;
    private int quantite;
    private String photo;

    @ManyToOne
    @JoinColumn(name = "categorie_id")
    private Categorie categorie;
}
```

Explication des annotations

- `@Entity` : Indique que cette classe est une entité JPA.
- `@Table(name = "produits")` : Spécifie le nom de la table dans la base de données.
- `@Id` : Marque le champ `id` comme clé primaire.
- `@GeneratedValue` : Configure la stratégie de génération de la clé primaire (auto-incrément).
- `@ManyToOne` : Définit une relation plusieurs-à-un. Plusieurs produits peuvent appartenir à une catégorie.
- `@JoinColumn` : Spécifie la colonne de jointure (clé étrangère) dans la table `produits`.

5.3 Entité `Categorie`

Cette entité représente une catégorie de produits.

```java
package com.example.entities;

import jakarta.persistence.*;
import lombok.*;
import java.util.List;
import com.fasterxml.jackson.annotation.JsonIgnore;

@Entity
@Table(name = "categories")
@Data
@Builder
@AllArgsConstructor
```

```java
@NoArgsConstructor
public class Categorie {

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

    private String nom;
    private String description;

    @JsonIgnore
    @OneToMany(mappedBy = "categorie", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    private List<Produit> produits;
}
```

Explication des annotations

— `@OneToMany` : Définit une relation un-à-plusieurs. Une catégorie peut avoir plusieurs produits.

— `mappedBy = "categorie"` : Indique que la relation est gérée par le champ `categorie` de l'entité `Produit`. Cela évite la création d'une table de jointure.

— `cascade = CascadeType.ALL` : Propage les opérations (création, mise à jour, suppression) de la catégorie à ses produits associés.

— `fetch = FetchType.LAZY` : Spécifie que la collection des produits ne doit pas être chargée de la base de données en même temps que l'entité `Categorie`. La collection ne sera récupérée que lors du premier accès explicite.

— `@JsonIgnore` : Empêche la sérialisation de la liste des produits lors de la conversion de l'objet `Categorie` en JSON, afin d'éviter les boucles de sérialisation infinies.

Stratégie de chargement : LAZY vs EAGER

L'attribut `fetch` définit *quand* les données d'une association sont récupérées depuis la base de données. Il existe deux stratégies principales :

— **FetchType.LAZY (Chargement Paresseux)** : C'est la stratégie la plus performante dans la majorité des cas. Les données de la collection (ici, la liste des produits) ne sont chargées que lorsque l'application y accède pour la première fois.

  — **Avantage** : Le chargement initial de l'entité parente (`Categorie`) est très rapide et consomme moins de mémoire, car il évite de charger des centaines de produits inutilement.

  — **Inconvénient** : Peut conduire à une erreur de type `LazyInitializationException` si l'on tente d'accéder à la collection alors que la session de la base de données est déjà fermée.

— **FetchType.EAGER (Chargement Empressé)** : Cette stratégie charge immédiatement l'entité parente `Categorie` **et** toute la collection de ses `Produits` associés, souvent via une seule requête SQL avec jointure.

  — **Avantage** : La collection est toujours disponible, ce qui peut simplifier le code car il n'y a pas de risque de `LazyInitializationException`.

  — **Inconvénient** : Peut causer de graves problèmes de performance. Si une catégorie contient des milliers de produits, ils seront tous chargés en mémoire, même si seule une information de la catégorie était nécessaire.

Comportement par défaut pour `@OneToMany`

Pour une relation `@OneToMany`, la spécification JPA (Java Persistence API) définit que la stratégie de chargement par défaut est **FetchType.LAZY**.
C'est un choix délibéré et sécuritaire, car les relations "vers plusieurs" (`@OneToMany`, `@ManyToMany`) sont susceptibles de contenir un grand nombre d'entités. Il est donc fortement recommandé de conserver ce comportement par défaut, sauf pour des raisons très spécifiques et justifiées.

6 Étape 2 : Définir les paramètres d'accès à la BD

Pour que notre application puisse communiquer avec la base de données MySQL, nous devons configurer la source de données (datasource) dans le fichier `application.properties`, situé dans

le dossier `src/main/resources`.

```
# Configuration du port du serveur
server.port=8081

# --- DATABASE ---
spring.datasource.url=jdbc:mysql://localhost:3306/catalogue_db?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password= # Mettez votre mot de passe ici

# --- JPA / HIBERNATE ---
# Montrer les requêtes SQL générées dans la console
spring.jpa.show-sql=true

# Stratégie de mise à jour du schéma de la base de données
# 'update' : met à jour le schéma sans supprimer les données
spring.jpa.hibernate.ddl-auto=update

# Spécifier le dialecte SQL pour MySQL
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

Explication des propriétés

- `spring.datasource.url` : L'URL de connexion JDBC vers la base de données. L'option `createDatabaseIfNotExist=true` est très pratique en développement car elle crée la base de données si elle n'existe pas au premier lancement.
- `spring.jpa.hibernate.ddl-auto=update` : C'est une instruction puissante. Elle demande à Hibernate (l'implémentation JPA utilisée par Spring) de comparer les entités Java avec les tables de la base de données et de mettre à jour le schéma si des différences sont détectées (ajout de colonnes, etc.), sans perdre les données existantes.

Après avoir ajouté cette configuration, vous pouvez lancer l'application. Spring Boot se connectera à la base de données et créera les tables `produits` et `categories` automatiquement.

7 Étape 3 : Création de la Couche d'Accès aux Données (Repository)

La couche Repository est une abstraction qui gère la communication avec la base de données. Avec **Spring Data JPA**, la création de cette couche est incroyablement simplifiée. Il suffit de créer une interface qui hérite de `JpaRepository`.

`JpaRepository<T, ID>` prend deux paramètres génériques :

- `T` : Le type de l'entité à gérer (ex : `Produit`).
- `ID` : Le type de la clé primaire de l'entité (ex : `Long`).

En héritant de `JpaRepository`, nos interfaces obtiennent instantanément des méthodes CRUD complètes (`save()`, `findById()`, `findAll()`, `deleteById()`, etc.) sans avoir à écrire la moindre ligne d'implémentation.

```
// ProduitRepository.java
package com.example.repository;
```

```java

import com.example.entities.Produit;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ProduitRepository extends JpaRepository<Produit, Long> {}
```

```java
// CategorieRepository.java
package com.example.repository;

import com.example.entities.Categorie;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface CategorieRepository extends JpaRepository<Categorie, Long> {}
```

7.1 Requêtes Personnalisées

Spring Data JPA permet de créer facilement des requêtes personnalisées de deux manières principales.

7.1.1 Requêtes dérivées (Derived Queries)

Il suffit de déclarer une méthode dans l'interface en respectant une convention de nommage stricte. Spring Data JPA analyse le nom de la méthode et génère automatiquement la requête SQL correspondante. C'est simple et puissant pour les requêtes courantes.

7.1.2 Requêtes JPQL avec `@Query`

Pour des requêtes plus complexes ou pour avoir un contrôle total, on peut utiliser l'annotation `@Query` pour écrire directement la requête en JPQL (Java Persistence Query Language).

```java
// Ajouts dans l'interface ProduitRepository.java

import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import java.util.List;

public interface ProduitRepository extends JpaRepository<Produit, Long> {

    // Derived Query: Cherche les produits dont le nom contient une chaîne de caractères
    List<Produit> findByNomContains(String keyword);

    // Requête JPQL: Cherche les produits par ID de catégorie
    @Query("SELECT p FROM Produit p WHERE p.categorie.id = :idCat")
    List<Produit> getProduitsByCatId(@Param("idCat") Long idCat);
}
```

7.2 Ajout de Données de Test au Démarrage

Spring Boot offre un moyen très pratique d'exécuter du code juste après le démarrage de l'application grâce à l'interface `CommandLineRunner`. Nous allons l'utiliser pour créer et sauvegarder quelques catégories et produits.

Cette logique s'ajoute directement dans la classe principale de l'application (celle annotée avec `@SpringBootApplication`).

Le Rôle de `@Bean` et `CommandLineRunner`

- `CommandLineRunner` est une interface fonctionnelle qui possède une seule méthode : `run(String... args)`. Tout composant (Bean) qui implémente cette interface verra sa méthode `run` exécutée automatiquement après le chargement du contexte Spring.
- `@Bean` est une annotation que l'on place sur une méthode dans une classe de configuration. Elle indique à Spring que cette méthode produit un objet (un "bean") qui doit être géré par le conteneur Spring. Ici, nous créons un bean de type `CommandLineRunner`.

```java
// Dans la classe principale, ex: TesttttttApplication.java

package com.example.testtttttt;

import com.example.testtttttt.entities.Categorie;
import com.example.testtttttt.entities.Produit;
import com.example.testtttttt.repository.CategorieRepository;
import com.example.testtttttt.repository.ProduitRepository;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class TesttttttApplication {

    public static void main(String[] args) {
        SpringApplication.run(TesttttttApplication.class, args);
    }

    @Bean
    CommandLineRunner initDatabase(CategorieRepository categorieRepository,
                                   ProduitRepository produitRepository) {
        return args -> {
            // Créer et sauvegarder des catégories
            Categorie catInformatique = Categorie.builder()
                .nom("Informatique")
                .description("Matériel et logiciels informatiques")
                .build();
            categorieRepository.save(catInformatique);

            Categorie catElectronique = Categorie.builder()
                .nom("Électronique")
                .description("Appareils électroniques grand public")
```

```
                .build();
        categorieRepository.save(catElectronique);

        // Créer et sauvegarder des produits en les liant aux catégories
        produitRepository.save(Produit.builder()
                .nom("PC Portable Gamer")
                .prix(1200.50)
                .quantite(15)
                .categorie(catInformatique)
                .build());

        produitRepository.save(Produit.builder()
                .nom("Imprimante Laser")
                .prix(350.00)
                .quantite(30)
                .categorie(catInformatique)
                .build());

        produitRepository.save(Produit.builder()
                .nom("Machine à laver")
                .prix(499.99)
                .quantite(10)
                .categorie(catElectronique)
                .build());

        System.out.println("---- Base de données initialisée avec des données de test ---
    };
  }
}
```

Maintenant, à chaque fois que vous (re)démarrerez l'application, ces données seront insérées dans vos tables, vous assurant d'avoir toujours un jeu de données frais pour vos tests.

8 Étape 4 : Création de la Couche Service (Métier)

La couche Service contient la logique métier de l'application. Elle agit comme un intermédiaire entre la couche Controller et la couche Repository. Cette séparation des préoccupations rend le code plus modulaire, plus propre et plus facile à tester.

8.1 Interfaces des Services

Définir des interfaces est une bonne pratique en programmation orientée objet. Cela permet de découpler l'implémentation de la définition du service, facilitant ainsi les tests et d'éventuels changements d'implémentation. Voici l'interface complétée avec toutes les méthodes CRUD (Create, Read, Update, Delete) et de recherche.

```
// IProduitService.java
package com.example.service;
import com.example.entities.Produit;
import java.util.List;
public interface IProduitService {
```

```java
// Méthode pour ajouter ou mettre à jour un produit
Produit saveProduit(Produit p);
// Méthode pour mettre à jour un produit existant
Produit updateProduit(Produit p);

// Méthode pour supprimer un produit par son ID
void deleteProduitById(Long id);

// Méthode pour récupérer un produit par son ID
Produit getProduit(Long id);

// Méthode pour récupérer la liste de tous les produits
List<Produit> getAllProduits();

// Méthode pour rechercher des produits par nom
List<Produit> findByNomContains(String nom);

// Méthode pour rechercher des produits par l'ID de leur catégorie
List<Produit> findByCategorieId(Long idCat);
}
```

8.2 Implémentation des Services

Grâce à **l'injection de dépendances** de Spring, nous n'avons pas besoin d'instancier nous-mêmes nos Repositories. Le conteneur Spring s'en charge pour nous. L'annotation `@Service` déclare cette classe comme un composant Spring, et `@AllArgsConstructor` de Lombok crée un constructeur qui permet l'injection automatique.

Comprendre l'Injection de Dépendances

L'injection de dépendances est un patron de conception où un objet reçoit ses dépendances (c'est-à-dire les autres objets avec lesquels il travaille) d'une source externe, au lieu de les créer lui-même. Dans Spring, le "conteneur IoC" est cette source externe. Cela permet d'éviter un couplage fort entre les classes, rendant le code plus flexible, plus facile à tester et à maintenir.

Voici l'implémentation complète de l'interface IProduitService.

```java
// ProduitServiceImpl.java
package com.example.service;
import com.example.repository.ProduitRepository;
import com.example.entities.Produit;
import lombok.AllArgsConstructor;
import org.springframework.stereotype.Service;
import java.util.List;
@Service
@AllArgsConstructor // Injection de dépendances via le constructeur grâce à Lombok
public class ProduitServiceImpl implements IProduitService {

private final ProduitRepository produitRepository;

@Override
```

```java
public Produit saveProduit(Produit p) {
    return produitRepository.save(p);
}

@Override
public Produit updateProduit(Produit p) {
    // La méthode save() de JpaRepository gère à la fois la création et la mise à jour
    return produitRepository.save(p);
}

@Override
public void deleteProduitById(Long id) {
    produitRepository.deleteById(id);
}

@Override
public Produit getProduit(Long id) {
    // findById retourne un Optional, .get() lève une exception si non trouvé
    return produitRepository.findById(id).orElse(null); // ou .orElseThrow(...)
}

@Override
public List<Produit> getAllProduits() {
    return produitRepository.findAll();
}

@Override
public List<Produit> findByNomContains(String nom) {
    return produitRepository.findByNomContains(nom);
}

@Override
public List<Produit> findByCategorieId(Long idCat) {
    // Suppose que la méthode findByCategorieId(Long id) existe dans ProduitRepository
    return produitRepository.getProduitsByCatId(idCat);
}
}
```

9 Étape 5 : Création de la Couche Contrôleur (API REST)

Le contrôleur, maintenant refactorisé, n'interagit plus directement avec les Repositories. Il délègue toutes les opérations à la couche de service (IProduitService), ce qui constitue une architecture plus propre et plus maintenable.

Annotations clés du Controller

- `@RestController` : Combine `@Controller` et `@ResponseBody`. Indique que les méthodes retournent des données (JSON) directement dans le corps de la réponse HTTP.
- `@RequestMapping("/api")` : Définit un préfixe d'URL pour tous les endpoints de ce contrôleur.
- `@CrossOrigin("*")` : Autorise les requêtes provenant de n'importe quelle origine. Essentiel pour le développement, car le frontend Angular sera servi sur un port différent (par défaut 4200).
- `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping` : Mappent les requêtes HTTP GET, POST, PUT, DELETE aux méthodes Java correspondantes.
- `@PathVariable` : Lie une variable de la méthode à une partie de l'URL (ex : `/products/id`).
- `@RequestParam` : Lie un paramètre de la requête URL (ex : `?nom=PC`) à une variable de la méthode.
- `@RequestBody` : Lie le corps de la requête HTTP (généralement du JSON) à un objet Java.

9.1 ProduitController

Le contrôleur injecte maintenant IProduitService et chaque méthode appelle la logique métier correspondante depuis le service.

```java
// ProduitController.java
package com.example.controller;
import com.example.entities.Produit;
import com.example.service.IProduitService;
import lombok.AllArgsConstructor;
import org.springframework.web.bind.annotation.*;
import java.util.List;
@RestController
@RequestMapping("/api")
@CrossOrigin("*")
@AllArgsConstructor
public class ProduitController {

private final IProduitService produitService;

@GetMapping("/products")
public List<Produit> getAll() {
    return produitService.getAllProduits();
}

@GetMapping("/products/{id}")
public Produit getOne(@PathVariable Long id){
    return produitService.getProduit(id);
}

@GetMapping("/products/byCat/{id}")
public List<Produit> getByCat(@PathVariable Long id){

```

```
    return produitService.findByCategorieId(id);
}

@GetMapping("/products/byNom")
public List<Produit> getByNom(@RequestParam String nom){
    return produitService.findByNomContains(nom);
}

@PostMapping("/products/add")
public Produit addProduit(@RequestBody Produit produit){
    return produitService.saveProduit(produit);
}

@PutMapping("/products/update")
public Produit updateProduit(@RequestBody Produit produit){
    return produitService.updateProduit(produit);
}

@DeleteMapping("/products/{id}")
public void deleteProduit(@PathVariable Long id){
    produitService.deleteProduitById(id);
}
}
```

9.2 CategorieController

Ce contrôleur reste inchangé, car la demande concernait principalement la logique des produits. Il continue d'utiliser directement le CategorieRepository pour des opérations simples.

```
// CategorieController.java
package com.example.controller;
import com.example.entities.Categorie;
import com.example.repository.CategorieRepository;
import lombok.AllArgsConstructor;
import org.springframework.web.bind.annotation.*;
import java.util.List;
@RestController
@RequestMapping("/api")
@CrossOrigin("*")
@AllArgsConstructor
public class CategorieController {

private final CategorieRepository categorieRepository;

@GetMapping("/categories")
public List<Categorie> getCategories() {
    return categorieRepository.findAll();
}
}
```

10 Étape 6 : Initialisation des Données et Test de l'API

Avant de construire notre interface utilisateur, il est crucial de s'assurer que notre API REST fonctionne comme prévu. Pour cela, nous allons d'abord insérer quelques données de test dans notre base de données au démarrage de l'application, puis nous utiliserons un outil appelé Swagger UI pour visualiser et interagir avec nos endpoints.

10.1 Test de l'API avec Swagger UI

Swagger (maintenant connu sous le nom d'OpenAPI Specification) est un standard pour documenter les API REST. Swagger UI est un outil qui génère une interface web interactive à partir de cette documentation, vous permettant de voir tous vos endpoints et de les tester directement depuis votre navigateur.

10.1.1 Ajouter la dépendance

Pour intégrer Swagger UI à notre projet Spring Boot, il suffit d'ajouter une seule dépendance dans le fichier `pom.xml`.

```
<!-- Dépendance pour Swagger UI / OpenAPI 3 -->
        <dependency>
            <groupId>org.springdoc</groupId>
            <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
            <version>2.5.0</version>
        </dependency>
```

10.1.2 Utilisation

Après avoir ajouté la dépendance, suivez ces étapes :

1. **Rechargez votre projet Maven** pour qu'il télécharge la nouvelle dépendance.
2. **Redémarrez votre application Spring Boot**.
3. **Ouvrez votre navigateur** et accédez à l'URL suivante :
   http://localhost:8081/swagger-ui/index.html

Vous devriez voir une page web listant vos deux contrôleurs (`produit-controller` et `categorie-controller` et tous les endpoints que vous avez créés.

Depuis cette interface, vous pouvez :

— **Voir la structure** de chaque endpoint (méthode HTTP, URL, paramètres requis).

— **Dérouler un endpoint** pour voir plus de détails et un bouton "Try it out".

— **Tester l'endpoint** en remplissant les paramètres et en cliquant sur "Execute". La réponse du serveur (code de statut, corps de la réponse JSON) s'affichera directement sur la page.

Cet outil est indispensable en développement pour valider le comportement du backend de manière isolée, avant même d'écrire la première ligne de code frontend.

FIGURE 5 – Interface de Swagger UI montrant les endpoints de l'API générés automatiquement.

Deuxième partie

Partie II : Développement du Frontend avec Angular

Maintenant que notre API backend est fonctionnelle et testée, nous allons construire l'interface utilisateur qui permettra de consommer cette API. Nous utiliserons Angular, un framework puissant pour construire des applications web monopages (Single Page Applications).

11 Étape 1 : Mise en Place du Projet Angular

1. **Créer un dossier** `frontend` à la racine de votre projet Spring Boot.
2. **Ouvrir un terminal** dans ce nouveau dossier `frontend`.
3. **Créer le projet Angular** avec la commande Angular CLI. Assurez-vous d'avoir Node.js et Angular CLI installés sur votre machine.
   `ng new catalogue-frontend`
   Puis, naviguez dans le dossier du projet nouvellement créé :
   `cd catalogue-frontend`
4. **Ajouter Bootstrap** pour le style.
   `npm install bootstrap`
   Ensuite, ouvrez le fichier `src/styles.scss` et ajoutez la ligne suivante au tout début :

   ```
   @import "bootstrap/dist/css/bootstrap.min.css";
   ```

5. **Lancer le serveur de développement.**
   `ng serve -open`
   L'application sera alors accessible dans votre navigateur à l'adresse http://localhost:4200.

12 Étape 2 : Création des Briques de Base

12.1 Modèle de Données (Interface)

En TypeScript, on utilise des **interfaces** pour définir la structure des données. Utilisez la commande suivante pour générer le fichier :

`ng generate interface models/produit`

Cela crée un fichier `produit.ts` dans le dossier `src/app/models`. Remplissez-le avec le contenu suivant :

```
// src/app/models/produit.ts
export interface Produit {
    id: number;
    nom: string;
    prix: number;
    quantite: number;
    categorie: any; // Pour simplifier, on utilise le type 'any'
}
```

12.2 Service de Communication (ProductService)

Un service est une classe dédiée à une tâche spécifique, comme la communication avec une API. Créez le service avec cette commande :

`ng generate service services/product`

Concepts Clés du Service Angular

- `@Injectable(providedIn: 'root')` : Ce décorateur indique qu'Angular peut injecter cette classe comme une dépendance. `providedIn: 'root'` en fait un service "singleton" disponible dans toute l'application.
- `HttpClient` : Le service d'Angular pour effectuer des requêtes HTTP. Il doit être importé dans le fichier `app.module.ts`.
- `Observable` : Les méthodes de `HttpClient` retournent des Observables (RxJS). C'est un flux de données asynchrone auquel on peut "s'abonner" (`.subscribe()`) pour recevoir les données une fois qu'elles arrivent du serveur.

Voici le code complet pour le `product.service.ts` :

```
// src/app/services/product.service.ts
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { Produit } from '../models/produit';

@Injectable({
  providedIn: 'root'
})
export class ProductService {
  private apiUrl = 'http://localhost:8081/api'; // URL de base de notre API

  constructor(private http: HttpClient) { }
```

```
14
15    public getProducts(): Observable<Produit[]> {
16      return this.http.get<Produit[]>(`${this.apiUrl}/products`);
17    }
18
19    public getProductsByNom(nom: string): Observable<Produit[]> {
20      return this.http.get<Produit[]>(`${this.apiUrl}/products/byNom?nom=${nom}`);
21    }
22
23    public getProductsByCat(idcat: string): Observable<Produit[]> {
24      return this.http.get<Produit[]>(`${this.apiUrl}/products/byCat/${idcat}`);
25    }
26
27    public addProduct(produit: Produit): Observable<void> {
28      return this.http.post<void>(`${this.apiUrl}/products/add`, produit);
29    }
30
31    public deleteProduct(id: number): Observable<void> {
32      return this.http.delete<void>(`${this.apiUrl}/products/${id}`);
33    }
34  }
```

Important : Activer HttpClient dans une Application Standalone

Dans une architecture moderne basée sur les composants `standalone`, la configuration des fournisseurs (*providers*) de l'application, comme `HttpClient`, est centralisée dans le fichier `app.config.ts`.

Pour activer `HttpClient`, il suffit d'utiliser la fonction `provideHttpClient()` et de l'ajouter au tableau des `providers`.

12.3 Service de Communication (CategorieService)

Ce service est dédié à la récupération des catégories depuis l'API. Créez-le avec :
`ng generate service services/categorie`
Remplissez le fichier `src/app/services/categorie.service.ts` :

```
1   // src/app/services/categorie.service.ts
2   import { Injectable } from '@angular/core';
3   import { HttpClient } from '@angular/common/http';
4   import { Observable } from 'rxjs';
5
6   @Injectable({
7     providedIn: 'root'
8   })
9   export class CategorieService {
10    private apiUrl = 'http://localhost:8081/api'; // URL de base de notre API
11
12    constructor(private http: HttpClient) { }
13
14    public getAllCategories(): Observable<any[]> {
15      return this.http.get<any[]>(`${this.apiUrl}/categories`);
16    }
17  }
```

13 Étape 3 : Composant d'Affichage des Produits

Ce composant affichera la liste des produits et permettra à l'utilisateur d'interagir avec. Générez-le avec la commande suivante :

```
ng generate component components/products
```

13.1 Logique du Composant (products.component.ts)

Dans une architecture standalone, le composant doit déclarer lui-même ses dépendances. Il n'y a plus de `NgModule` pour le faire à sa place. Pour ce composant, les changements se situent uniquement dans le décorateur `@Component` :

1. On ajoute la propriété `standalone: true`.
2. On ajoute un tableau `imports` pour importer les modules et directives nécessaires.
   - `CommonModule` : Indispensable pour les directives structurelles comme `*ngFor` et les pipes comme `currency`.
   - `FormsModule` : Requis pour utiliser le *two-way data binding* avec `[(ngModel)]`.

La logique à l'intérieur de la classe (`constructor`, `ngOnInit`, méthodes...) reste **strictement identique**.

```
// Décorateur original (basé sur les modules)
@Component({
  selector: 'app-products',
  templateUrl: './products.component.html',
  styleUrls: ['./products.component.css']
})

// Décorateur modifié pour une architecture Standalone
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-products',
  standalone: true,
  imports: [CommonModule, FormsModule], // Déclaration des dépendances
  templateUrl: './products.component.html',
  styleUrls: ['./products.component.css']
})
```

Code complet du composant (la logique ne change pas) :

```
import { Component, OnInit } from '@angular/core';
import { ProductService } from '../../services/product.service';
import { Produit } from '../../models/produit';
import { Router } from '@angular/router';
import { CategorieService } from '../../services/categorie.service';
// Pour standalone, il faudrait ajouter les imports suivants :
// import { CommonModule } from '@angular/common';
// import { FormsModule } from '@angular/forms';
```

```
9
10 @Component({
11   selector: 'app-products',
12   // standalone: true, // A décommenter pour le mode standalone
13   // imports: [CommonModule, FormsModule], // A décommenter pour le mode
     ↪  standalone
14   templateUrl: './products.component.html',
15   styleUrls: ['./products.component.css']
16 })
17 export class ProductsComponent implements OnInit {
18   produits?: Produit[];
19   searchNom: string = '';
20   selectedCategorie: string = ''; // L'ID de la catégorie
21   categories: any[] = []; // Doit être chargé depuis CategorieService
22
23   constructor(
24     private serviceproduit: ProductService,
25     private serviceCategorie: CategorieService, // Injecter le service
26     private router: Router
27   ) {}
28
29   ngOnInit(): void {
30     this.loadAllProducts();
31     this.loadCategories(); // Appeler une méthode pour charger les catégories
       ↪  ici
32   }
33
34   loadAllProducts(): void {
35     this.serviceproduit.getProducts().subscribe(data => this.produits = data);
36   }
37
38   loadCategories(): void {
39     this.serviceCategorie.getAllCategories().subscribe(data => this.categories =
       ↪  data);
40   }
41
42   onSearchChange(): void {
43     if (this.searchNom) {
44       this.serviceproduit.getProductsByNom(this.searchNom)
45         .subscribe(data => this.produits = data);
46     } else {
47       this.loadAllProducts();
48     }
49   }
50
51   onCategorieChange(): void {
52     if (this.selectedCategorie) {
53       this.serviceproduit.getProductsByCat(this.selectedCategorie)
54         .subscribe(data => this.produits = data);
55     } else {
56       this.loadAllProducts();
57     }
```

```
  }

  supprimer(p: Produit): void {
    let conf = confirm("Êtes-vous sûr de vouloir supprimer ce produit ?");
    if (!conf) return;

    this.serviceproduit.deleteProduct(p.id).subscribe(() => {
      alert("Produit supprimé avec succès !");
      this.loadAllProducts(); // Recharger la liste pour refléter la suppression
    });
  }

  ajouterNouveauProduit(): void {
    this.router.navigate(['/add-product']);
  }
}
```

13.2 Template HTML (products.component.html)

```
<div class="container mt-4">
    <div class="card">
        <div class="card-header d-flex justify-content-between
        ↪  align-items-center">
            <h3>Liste des Produits</h3>
            <button class="btn btn-primary" (click)="ajouterNouveauProduit()">
                Ajouter un produit
            </button>
        </div>
        <div class="card-body">
            <!-- Section des filtres -->
            <div class="row mb-3">
                <div class="col-md-6">
                    <input type="text" class="form-control"
                           placeholder="Rechercher par nom..."
                           [(ngModel)]="searchNom"
                           (input)="onSearchChange()">
                </div>
                <div class="col-md-6">
                    <select class="form-select"
                            [(ngModel)]="selectedCategorie"
                            (change)="onCategorieChange()">
                        <option value="">Toutes les catégories</option>
                        <option *ngFor="let cat of categories" [value]="cat.id">
                            {{ cat.nom }}
                        </option>
                    </select>
                </div>
            </div>

            <!-- Tableau des produits -->
            <table class="table table-striped table-bordered">
```

```
                <thead class="thead-dark">
                    <tr>
                        <th>ID</th>
                        <th>Nom</th>
                        <th>Prix</th>
                        <th>Quantité</th>
                        <th>Catégorie</th>
                        <th>Actions</th>
                    </tr>
                </thead>
                <tbody>
                    <tr *ngFor="let p of produits">
                        <td>{{ p.id }}</td>
                        <td>{{ p.nom }}</td>
                        <td>{{ p.prix | currency:'EUR' }}</td>
                        <td>{{ p.quantite }}</td>
                        <td>{{ p.categorie.nom }}</td>
                        <td>
                            <button (click)="supprimer(p)" class="btn btn-danger
                            ↪ btn-sm">
                                Supprimer
                            </button>
                        </td>
                    </tr>
                </tbody>
            </table>
        </div>
    </div>
</div>
```

14 Étape 4 : Composant d'Ajout de Produit

Ce composant contiendra le formulaire pour créer un nouveau produit. Générez-le avec la commande :

`ng generate component components/add-product`

14.1 Logique du Composant (add-product.component.ts)

Similaire au composant précédent, la transformation en standalone se fait dans le décorateur. Ici, le formulaire piloté par le template (*template-driven form*) est au cœur du composant, rendant l'import de `FormsModule` essentiel.

1. On ajoute `standalone: true`.
2. Dans le tableau `imports`, on inclut `FormsModule` (pour `NgForm`, `ngModel`, etc.) et `CommonModule` (pour `*ngFor` dans le select).

Encore une fois, la logique de la classe reste inchangée.

```
// Décorateur original (basé sur les modules)
@Component({
  selector: 'app-add-product',
  templateUrl: './add-product.component.html',
```

```
  styleUrls: ['./add-product.component.css']
})

// Décorateur modifié pour une architecture Standalone
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-add-product',
  standalone: true,
  imports: [CommonModule, FormsModule], // FormsModule est crucial ici
  templateUrl: './add-product.component.html',
  styleUrls: ['./add-product.component.css']
})
```

Code complet du composant (la logique ne change pas) :

```
import { Component, OnInit } from '@angular/core';
import { NgForm } from '@angular/forms';
import { ProductService } from '../../services/product.service';
import { Router } from '@angular/router';
import { CategorieService } from '../../services/categorie.service';
// Pour standalone, il faudrait ajouter les imports suivants :
// import { CommonModule } from '@angular/common';
// import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-add-product',
  // standalone: true, // A décommenter pour le mode standalone
  // imports: [CommonModule, FormsModule], // A décommenter pour le mode standalone
  templateUrl: './add-product.component.html',
  styleUrls: ['./add-product.component.css']
})
export class AddProductComponent implements OnInit {
  categories: any[] = []; // Doit être chargé depuis CategorieService

  constructor(
    private prodservice: ProductService,
    private router: Router,
    private servicecat: CategorieService // Injecter
  ) { }

  ngOnInit(): void {
    this.servicecat.getAllCategories().subscribe(data => this.categories = data);
  }

  onSubmit(productForm: NgForm): void {
    if (productForm.invalid) {
      return; // Ne pas soumettre si le formulaire est invalide
    }

```

```
    // La valeur du formulaire est directement un objet Produit
    this.prodservice.addProduct(productForm.value).subscribe(() => {
      alert("Produit ajouté avec succès !");
      this.router.navigate(['/products']);
    });
  }

  retourListe(): void {
    this.router.navigate(['/products']);
  }
}
```

14.2 Template HTML (add-product.component.html)

```
<div class="container mt-4">
    <div class="d-flex justify-content-between align-items-center mb-4">
        <h2 class="text-primary">Ajouter un nouveau produit</h2>
        <button class="btn btn-outline-secondary" (click)="retourListe()">
            <i class="fas fa-arrow-left"></i> Retour à la liste
        </button>
    </div>

    <div class="card">
        <div class="card-body">
            <form #productForm="ngForm" (ngSubmit)="onSubmit(productForm)">
                <div class="row">
                    <!-- Nom du produit -->
                    <div class="col-md-6 mb-3">
                        <label for="nom" class="form-label">Nom du produit
                        ↪  *</label>
                        <input type="text" id="nom" name="nom"
                        ↪  class="form-control"
                               ngModel #nom="ngModel" required
                               [class.is-invalid]="nom.invalid && nom.touched">
                    </div>

                    <!-- Prix -->
                    <div class="col-md-6 mb-3">
                        <label for="prix" class="form-label">Prix *</label>
                        <input type="number" id="prix" name="prix"
                        ↪  class="form-control"
                               ngModel #prix="ngModel" required
                               [class.is-invalid]="prix.invalid &&
                               ↪  prix.touched">
                    </div>
                </div>

                <div class="row">
                    <!-- Quantité -->
                    <div class="col-md-6 mb-3">
```

```
                        <label for="quantite" class="form-label">Quantité *</label>
                        <input type="number" id="quantite" name="quantite" class="form-control"
                               ngModel #quantite="ngModel" required
                               [class.is-invalid]="quantite.invalid && quantite.touched">
                    </div>

                    <!-- Catégorie -->
                    <div class="col-md-6 mb-3">
                        <label for="categorie" class="form-label">Catégorie *</label>
                        <select id="categorie" name="categorie" class="form-select"
                                ngModel #categorie="ngModel" required
                                [class.is-invalid]="categorie.invalid && categorie.touched">
                            <option [ngValue]="null" disabled>
                                Sélectionnez une catégorie
                            </option>
                            <option *ngFor="let cat of categories" [ngValue]="cat">
                                {{ cat.nom }}
                            </option>
                        </select>
                    </div>
                </div>

                <!-- Boutons d'action -->
                <div class="mt-3">
                    <button type="submit" class="btn btn-success"
                            [disabled]="productForm.invalid">
                        Enregistrer
                    </button>
                </div>
            </form>
        </div>
    </div>
</div>
```

15 Étape 5 : Routage et Configuration Finale

Cette étape finale consiste à connecter tous nos composants et modules pour que l'application soit navigable. Nous allons explorer deux approches : la méthode moderne avec les composants **standalone** et l'approche traditionnelle basée sur les **NgModules**.

15.1 Approche Moderne (Standalone)

L'architecture standalone vise à simplifier la configuration en la centralisant. Les fichiers `app-routing.module.ts` et `app.module.ts` disparaissent au profit d'une configuration plus di-

recte.

15.1.1 Définition des Routes (app.routes.ts)

Les routes sont définies dans un simple tableau constant, généralement dans un nouveau fichier `app.routes.ts`.

```
// src/app/app.routes.ts
import { Routes } from '@angular/router';
import { ProductsComponent } from './components/products/products.component';
import { AddProductComponent } from './components/add-product/add-product.component';

export const routes: Routes = [
  { path: 'products', component: ProductsComponent },
  { path: 'add-product', component: AddProductComponent },
  { path: '', redirectTo: 'products', pathMatch: 'full' } // Route par défaut
];
```

15.1.2 Configuration de l'Application (app.config.ts)

Ce fichier est le nouveau centre de configuration. On y utilise la fonction `provideRouter` pour activer le routage avec les routes que nous venons de définir.

```
// src/app/app.config.ts
import { ApplicationConfig } from '@angular/core';
import { provideRouter } from '@angular/router';
import { routes } from './app.routes';
import { provideHttpClient } from '@angular/common/http';

export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes), // Activation du routage pour l'application
    provideHttpClient()    // Activation de HttpClient
  ]
};
```

15.1.3 Mise à jour du Composant Principal (AppComponent)

Le composant racine (`AppComponent`) doit lui-même devenir standalone. Il doit alors importer les directives `RouterOutlet` et `RouterLink` pour que son template puisse les utiliser.

```
// src/app/app.component.ts
import { Component } from '@angular/core';
import { RouterLink, RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, RouterLink], // Importe les directives pour le template
  templateUrl: './app.component.html'
```

```
})
export class AppComponent {}
```

Le template `app.component.html` reste identique, car les directives sont maintenant fournies directement par l'import du composant.

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark mb-4">
  <div class="container-fluid">
    <a class="navbar-brand" routerLink="/products">Catalogue de Produits</a>
  </div>
</nav>

<main class="container">
  <!-- Les composants routés s'afficheront ici -->
  <router-outlet></router-outlet>
</main>
```

Astuce : Distinguer l'ancienne approche (basée sur les NgModules) La section suivante décrit l'approche traditionnelle d'Angular, qui était la norme avant la version 15. Elle repose sur une organisation en modules (NgModule). Bien qu'elle soit toujours fonctionnelle, elle est considérée comme plus verbeuse et a été remplacée par l'architecture **standalone** comme méthode par défaut dans les nouvelles applications Angular (depuis la v17). Cette partie est conservée ici à des fins de comparaison et pour la maintenance de projets plus anciens.

15.2 Approche Traditionnelle (Héritée des anciennes versions)

Cette approche est basée sur la séparation des responsabilités en modules. Le routage a son propre module (`AppRoutingModule`), et le module principal (`AppModule`) importe tous les autres modules et déclare les composants.

15.2.1 Table de Routage (app-routing.module.ts)

Ce fichier définit la navigation entre les composants et encapsule la logique de routage.

```
// src/app/app-routing.module.ts
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { ProductsComponent } from './components/products/products.component';
import { AddProductComponent } from './components/add-product/add-product.component';
const routes: Routes = [
{ path: 'products', component: ProductsComponent },
{ path: 'add-product', component: AddProductComponent },
{ path: '', redirectTo: 'products', pathMatch: 'full' } // Route par défaut
```

```
];
@NgModule({
imports: [RouterModule.forRoot(routes)],
exports: [RouterModule]
})
export class AppRoutingModule { }
```

15.2.2 Composant Principal (app.component.html)

Le template reste le même. Les directives `<router-outlet>` et `routerLink` fonctionnent car `AppRoutingModule` (qui exporte `RouterModule`) est importé dans le module principal `AppModule`.

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark mb-4">
<div class="container-fluid">
<a class="navbar-brand" routerLink="/products">Catalogue de Produits</a>
</div>
</nav>
<main class="container">
<!-- Les composants routés s'afficheront ici -->
<router-outlet></router-outlet>
</main>
```

15.2.3 Module Principal (app.module.ts)

C'est le cœur de l'application. Il déclare les composants et importe tous les modules nécessaires à leur fonctionnement.

```
// src/app/app.module.ts
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';
import { FormsModule } from '@angular/forms'; // Important pour ngModel
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { ProductsComponent } from './components/products/products.component';
import { AddProductComponent } from './components/add-product/add-product.component';
@NgModule({
declarations: [
AppComponent,
ProductsComponent,
AddProductComponent
],
imports: [
BrowserModule,
AppRoutingModule, // Importe la configuration du routage
HttpClientModule, // Pour les requêtes HTTP
FormsModule // Pour les formulaires (ngModel et ngForm)
],
providers: [],
bootstrap: [AppComponent]
})
```

```
export class AppModule { }
```

Troisième partie
Partie III : Bonus

16 Gestion de l'Upload d'Images

Pour enrichir notre application, nous allons ajouter la possibilité d'uploader une image lors de la création d'un produit. Cela implique des modifications à la fois dans le backend et le frontend pour gérer un type de requête spécifique : `multipart/form-data`.

16.1 Modifications du Backend (Spring Boot)

Notre backend doit être capable de recevoir un fichier en plus des données JSON du produit, de le sauvegarder sur le disque du serveur, et de stocker son nom dans la base de données.

16.1.1 Mise à jour de la Couche Service

Nous ajoutons deux méthodes dans `IProduitService` et leur implémentation dans `ProduitServiceImpl`.

1. **Une méthode pour sauvegarder le fichier :** Cette méthode prend un `MultipartFile`, génère un nom unique pour éviter les conflits, et sauvegarde le fichier dans un dossier `photos` situé dans le répertoire de l'utilisateur.

```
// Dans ProduitServiceImpl.java
import org.springframework.web.multipart.MultipartFile;
import java.io.IOException;
import java.nio.file.*;

// ...

public String uploadFile(MultipartFile file) throws IOException {
    // Génère un nom de fichier unique en préfixant avec le timestamp
    String newName = System.currentTimeMillis() + "_" + file.getOriginalFilename();

    // Définit le chemin de sauvegarde en utilisant la concaténation de chaînes
    String uploadDir = System.getProperty("user.home") + "/photos";
    Path uploadPath = Paths.get(uploadDir);

    // Crée le dossier de destination s'il n'existe pas
    if (!Files.exists(uploadPath)) {
        Files.createDirectories(uploadPath);
    }

    // Crée le chemin complet vers le nouveau fichier (dossier + nom du fichier)
    Path filePath = uploadPath.resolve(newName);

    // Écrit tous les octets du fichier sur le disque
    Files.write(filePath, file.getBytes());

```

```
27      // Retourne le nouveau nom du fichier pour référence future (ex: sauvegarde en BDD)
28      return newName;
29  }
```

2. **Une méthode pour récupérer l'image :** Cette méthode lira les octets (bytes) d'une image depuis le disque pour pouvoir l'afficher dans le frontend.

```
1   // Dans ProduitServiceImpl.java
2
3   public byte[] getImage(Long id) throws IOException {
4       Produit produit = produitRepository.findById(id).orElse(null);
5       String photoName = produit.getPhoto();
6
7       Path imagePath = Paths.get(System.getProperty("user.home"), "photos", photoName);
8
9       return Files.readAllBytes(imagePath);
10  }
```

16.1.2 Mise à jour du Contrôleur

Le `ProduitController` doit être modifié pour gérer les requêtes `multipart/form-data`.

— L'endpoint d'ajout est modifié pour consommer `MULTIPART_FORM_DATA_VALUE`.

— Il utilise `@RequestParam` pour recevoir séparément les données du produit (sous forme de chaîne JSON) et le fichier.

— `ObjectMapper` est utilisé pour convertir la chaîne JSON en objet Java `Produit`.

— Un nouvel endpoint `/image/{id}` est créé pour servir les images.

```
1   // Dans ProduitController.java
2   import com.fasterxml.jackson.databind.ObjectMapper;
3   import org.springframework.http.MediaType;
4   import org.springframework.web.multipart.MultipartFile;
5   import java.io.IOException;
6
7   // ...
8
9   // On modifie l'endpoint POST
10  @PostMapping(value = "/products/add", consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
11  public void addProduit(
12          @RequestParam("produit") String produitJson,
13          @RequestParam(value = "file", required = false) MultipartFile file) throws IOExceptio
14
15      // Désérialiser la chaîne JSON en objet Produit
16      Produit produit = new ObjectMapper().readValue(produitJson, Produit.class);
17
18      if (file != null && !file.isEmpty()) {
19          String photoName = produitService.uploadFile(file);
20          produit.setPhoto(photoName);
21      }
22
23      produitService.saveProduit(produit);
```

```
}

// On ajoute un endpoint pour récupérer l'image
@GetMapping(value = "/image/{id}", produces = MediaType.IMAGE_JPEG_VALUE)
public byte[] getImage(@PathVariable Long id) throws IOException {
    return produitService.getImage(id);
}
```

16.2 Modifications du Frontend (Angular)

Le frontend doit maintenant envoyer un objet `FormData` au lieu d'un simple JSON pour pouvoir inclure le fichier.

16.2.1 Mise à jour du Service Produit

Nous ajoutons une nouvelle méthode dans `ProductService` qui construit un objet `FormData`.

```
// Dans product.service.ts

// NOTE: Cette méthode remplace 'addProduct'. Il faudrait renommer ou surcharger.
public addProductWithImage(produit: Produit, file: File): Observable<void> {
    const formData = new FormData();

    // Clé 'produit' : doit correspondre au @RequestParam("produit") du backend
    formData.append('produit', JSON.stringify(produit));

    // Clé 'file' : doit correspondre au @RequestParam("file") du backend
    if (file) {
      formData.append('file', file, file.name);
    }

    return this.http.post<void>(`${this.apiUrl}/products/add`, formData);
}
```

16.2.2 Mise à jour du Composant d'Ajout

Le composant `AddProductComponent` est modifié pour gérer la sélection du fichier.

— **Logique (add-product.component.ts)** : On ajoute une propriété pour stocker le fichier sélectionné et une méthode pour gérer l'événement de sélection. La méthode `onSubmit` est mise à jour pour appeler la nouvelle méthode du service.

```
// Dans add-product.component.ts
export class AddProductComponent implements OnInit {
  // ...
  selectedFile?: File;

  // Méthode appelée lorsque l'utilisateur sélectionne un fichier
  onImageSelect(event: any): void {
    if (event.target.files.length > 0) {
      this.selectedFile = event.target.files[0];
```

```
    }
  }

  onSubmit(productForm: NgForm): void {
    if (productForm.invalid || !this.selectedFile) {
        alert("Veuillez remplir tous les champs et sélectionner une image.");
        return;
    }

    this.prodservice.addProductWithImage(productForm.value, this.selectedFile)
      .subscribe(() => {
        alert("Produit ajouté avec succès !");
        this.router.navigate(['/products']);
      });
  }
  // ...
}
```

— **Template (add-product.component.html)** : On ajoute un champ de type `file` dans le formulaire.

```
<!-- Dans add-product.component.html, à l'intérieur de la balise <form> -->
<div class="row">
    <!-- Champ pour l'upload de l'image -->
    <div class="col-md-12 mb-3">
        <label for="photo" class="form-label">Image du produit *</label>
        <input type="file" id="photo" class="form-control"
               (change)="onImageSelect($event)" required>
    </div>
</div>
```

16.2.3 Affichage des Images dans la Liste

Pour que l'image de chaque produit apparaisse dans le tableau, nous modifions le template du composant d'affichage.

— **Mise à jour du modèle de données (produit.ts)** : Assurez-vous que le modèle de données côté frontend inclut le champ pour le nom de la photo.

```
// Dans src/app/models/produit.ts
export interface Produit {
    id: number;
    nom: string;
    prix: number;
    quantite: number;
    categorie: any;
    photo?: string; // Propriété pour le nom du fichier image
}
```

— **Mise à jour du template (products.component.html)** : Ajoutez une colonne au tableau pour l'image. L'attribut `src` de la balise `<img>` pointera directement vers l'endpoint de l'API qui renvoie les données de l'image.

```html
<!-- Dans products.component.html, modifier le contenu de la balise <table> -->
<thead class="thead-dark">
    <tr>
        <th>Image</th>
        <th>ID</th>
        <th>Nom</th>
        <th>Prix</th>
        <th>Quantité</th>
        <th>Catégorie</th>
        <th>Actions</th>
    </tr>
</thead>
<tbody>
    <tr *ngFor="let p of produits">
        <td>
            <!-- L'URL est construite dynamiquement avec l'ID du produit -->
            <img [src]="'http://localhost:8081/api/image/' + p.id"
                 alt="{{ p.nom }}"
                 width="80"
                 class="img-thumbnail">
        </td>
        <td>{{ p.id }}</td>
        <td>{{ p.nom }}</td>
        <td>{{ p.prix | currency:'EUR' }}</td>
        <td>{{ p.quantite }}</td>
        <td>{{ p.categorie.nom }}</td>
        <td>
            <button (click)="supprimer(p)" class="btn btn-danger btn-sm">Supprimer</butt
        </td>
    </tr>
</tbody>
```