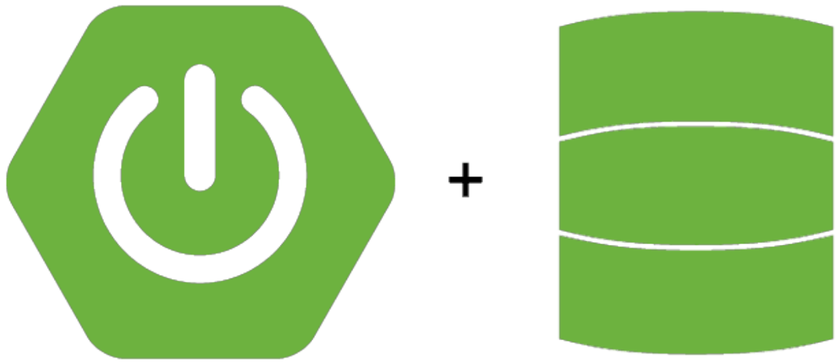
SPRING DATA JPA



Philémon Globléhi
Développeur Back-end

01- Définition

Un framework est, un “cadre de travail“. L’objectif est généralement de simplifier le travail des développeurs informatiques , en leur offrant une architecture “prête à l’emploi” et qui leur permette de ne pas repartir de zéro à chaque nouveau projet.

Un ORM (Object-Relational Mapping), sont des frameworks qui permettent de créer une correspondance entre un modèle objet et un modèle relationnel de base de données.

En Java, les frameworks ORM se basent sur une spécification nommée JPA (Java Persistance API). Cette spécification est une API qui décrit les comportements nécessaires au bon fonctionnement d’un framework ORM.
Grâce à ces spécifications, de nombreuses implémentations ont vu le jour, comme :

- EclipseLink ;
- Spring Data JPA ;
- OpenJPA ;
- Hibernate.

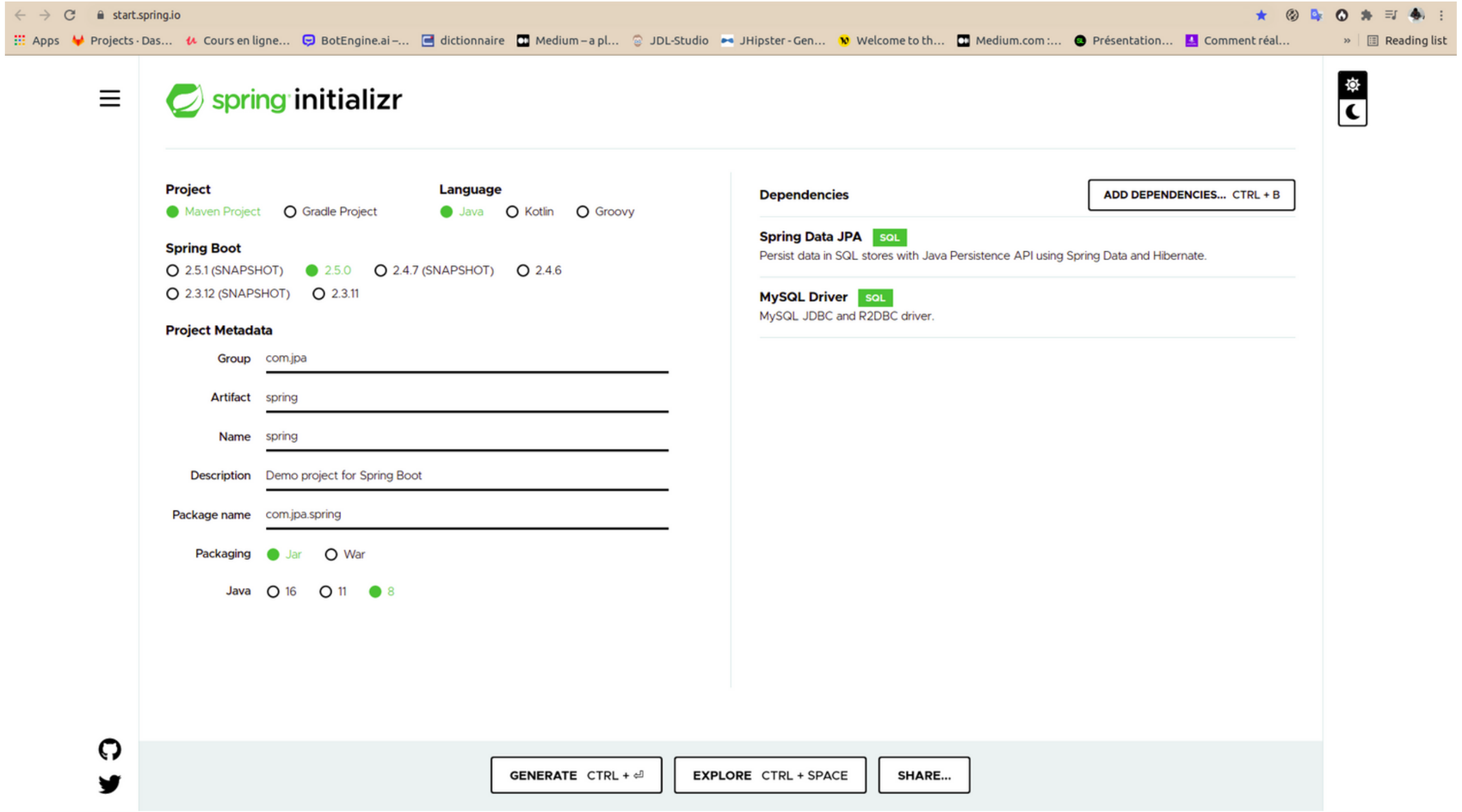
02 - Pourquoi Spring Data JPA ?

- Sa simplicité : Assez simple d'utilisation
- C’est un composant Spring, et le framework Spring est un excellent framework pour implémenter des applications Java robustes et performantes.
- Il n’y a qu’un pas après avoir appris Spring Data JPA pour apprendre Hibernate, un autre framework ORM très utilisé sur le marché.

Spring Data JPA est un framework ORM performant qui vous permet d’interagir avec une base de données.

03 - Configuration

Nous allons créer la structure de notre projet Spring Boot sur https://start.spring.io/, que nous importerons ensuite dans notre IDE.



Lors de la génération du projet, le répertoire /src/main/resources contient un fichier par défaut nommé applications.properties

Ce fichier est considéré par Spring comme étant une source de propriétés. Au démarrage de l'application, ce fichier sera lu, et les informations présentes pourront être exploitées par le code de l'application.

Voici le code nécessaire à la connexion de la base de données. Il s'agit d'ouvrir une connexion JDBC (Java DataBase Connectivity) vers notre base de données.

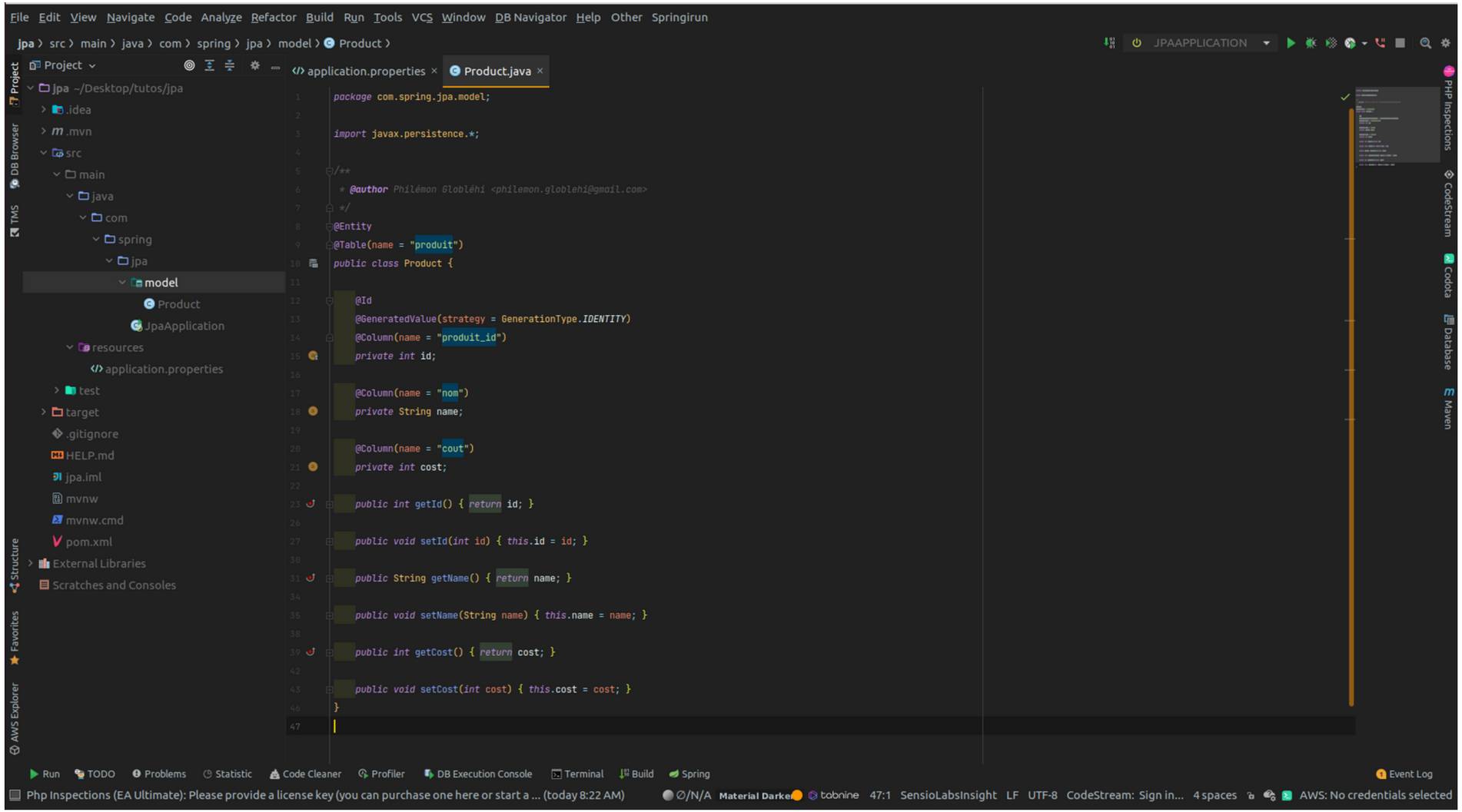
application.properties

```
1
2   spring.datasource.url=jdbc:mysql://localhost:3306/assuranceAuto?serverTimezone=UTC
3   spring.datasource.username=globlehi
4   spring.datasource.password=globlehi
5   spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect
6
```

- spring.datasource.url : indique l'URL de connexion à la base de données. Cette URL possède plusieurs informations clés :
    - jdbc:mysql : le protocole utilisé.
    - localhost : la machine où la base de données est installée.
    - 3306 : le port de la base de données sur votre poste.
    - /assuranceAuto : le nom de la base de données.
    - ?serverTimeZone=UTC : ce paramètre permet de résoudre une erreur courante avec le driver MySQL.
- spring.datasource.username : Nom de l'utilisateur pour se connecter à la base de données.
- spring.datasource.password : Mot de passe de l'utilisateur
- spring.jpa.properties.hibernate.dialect : Spécifie au framework ORM le dialecte SQL à utiliser, cela permet d'optimiser les traitements du framework.

04 - Entité

Nous allons implémenter la classe Product dans un package nommé model.

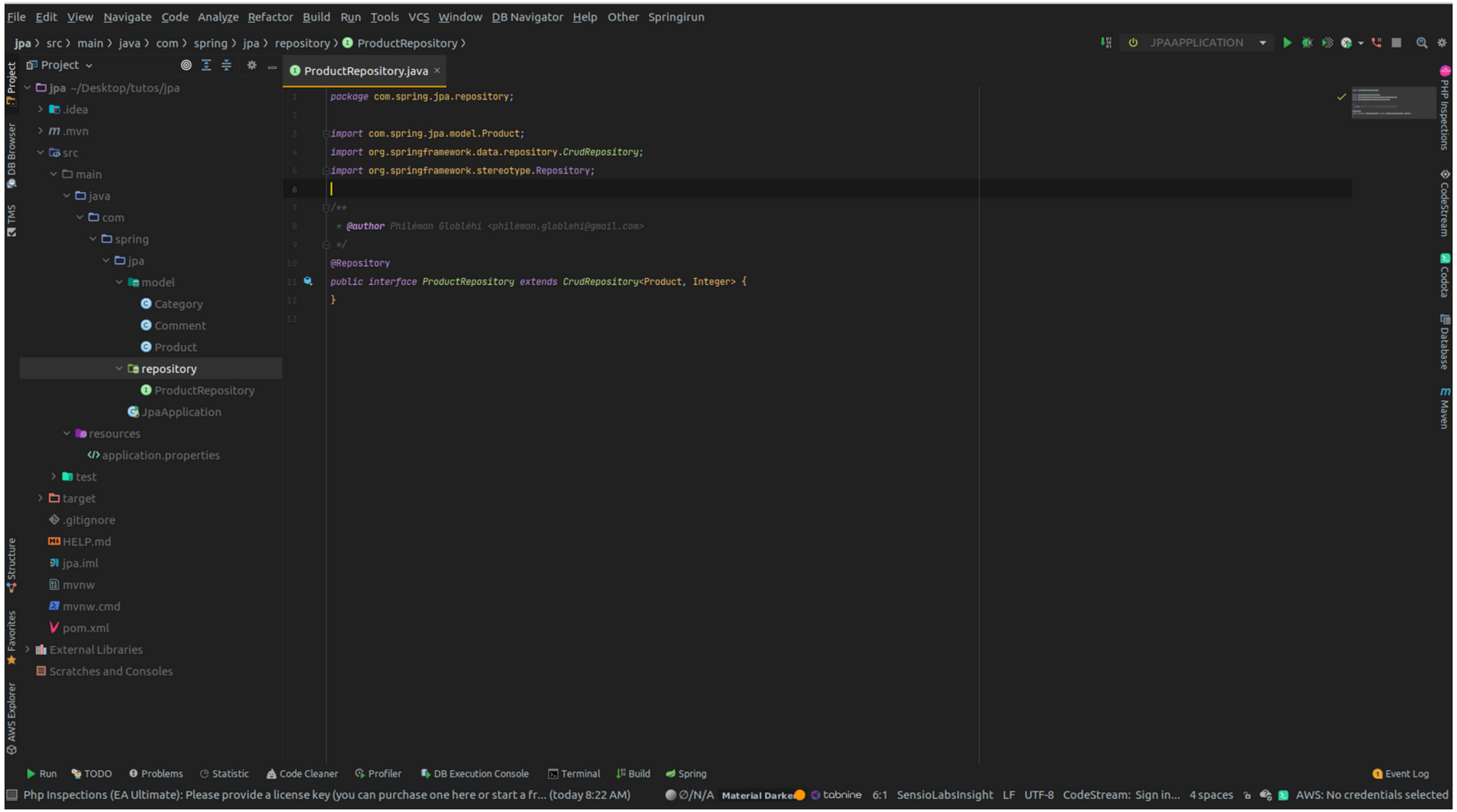


- @Entity est une annotation qui indique que la classe correspond à une table de la base de données.
- @Table(name=”produit”) est une annotation qui indique le nom de la table associée.
- @Id est une annotation qui indique la clé primaire de la table .
- id étant auto-incrémenté, on ajoute l’annotation @GeneratedValue(strategy = GenerationType.IDENTITY).
- @Column est une annotation qui permet de faire le lien entre les attributs de la classe avec le nom du champ de la table dans la base de données.

Il est important de noter que Spring Data JPA est capable de faire l’association automatique entre le nom de l’entité et le nom de la table, ainsi qu’entre le nom des attributs et le nom des colonnes. Donc, si les noms sont similaires, les annotations @Table et @Column deviennent facultatives.

05 - Repository

Nous allons implémenter l'interface ProductRepository dans un package nommé repository.



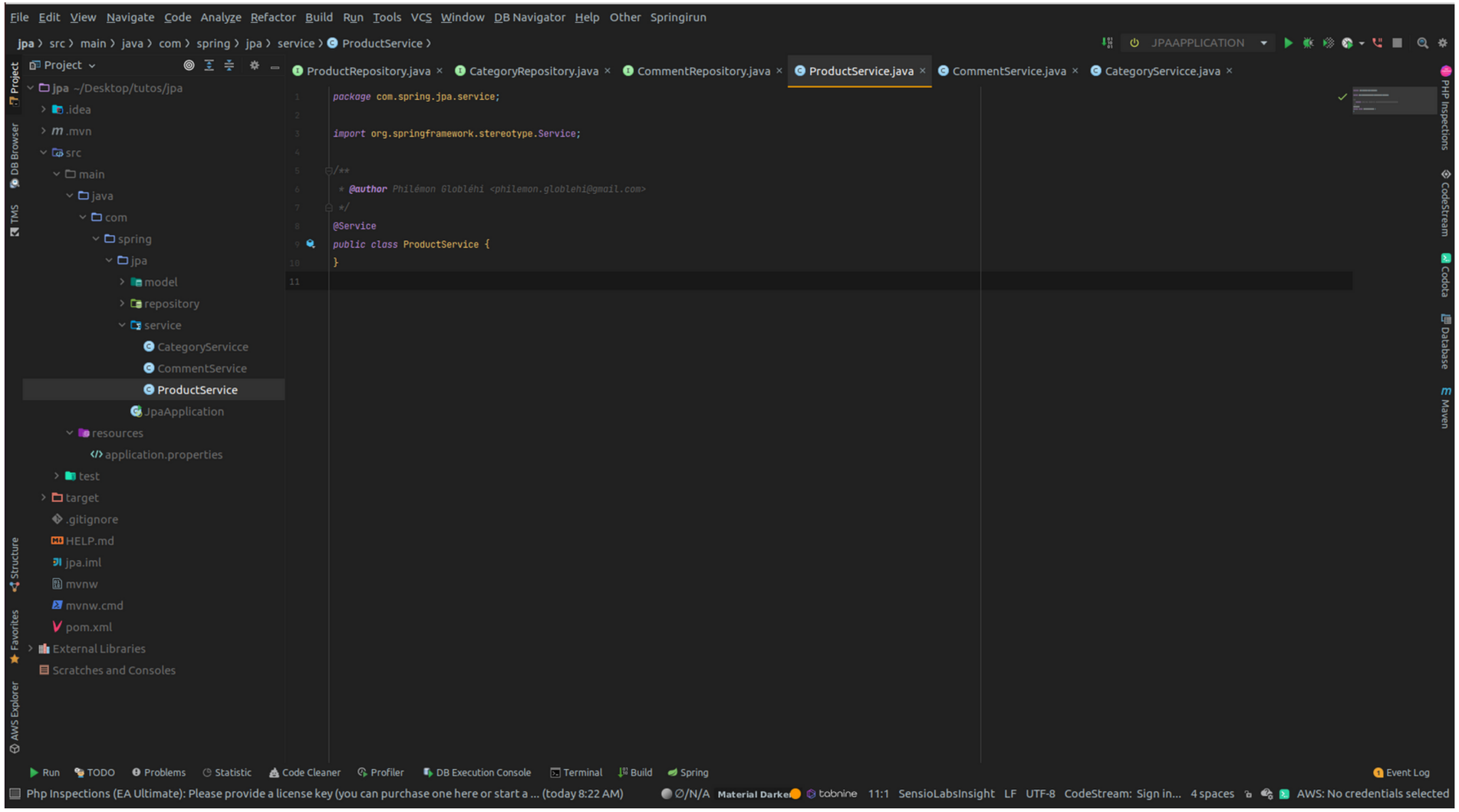
- @Repository est une annotation Spring pour indiquer que la classe a pour rôle de communiquer avec une source de données (en l'occurrence la base de données).

L’interface étend CrudRepository.
C’est une interface qui est fournie par Spring. Elle spécifie des méthodes pour manipuler notre entité. Elle utilise la généricité pour que son code soit applicable à n’importe quelle entité, d’où la syntaxe “CrudRepository<Product, Integer>”.

@Repository est une spécialisation de l’annotation @Component. Elle permet de déclarer auprès du Context d'Application Spring qu’une classe est un bean à exploiter. Par son nom, elle fournit une indication sémantique sur le rôle de la classe.

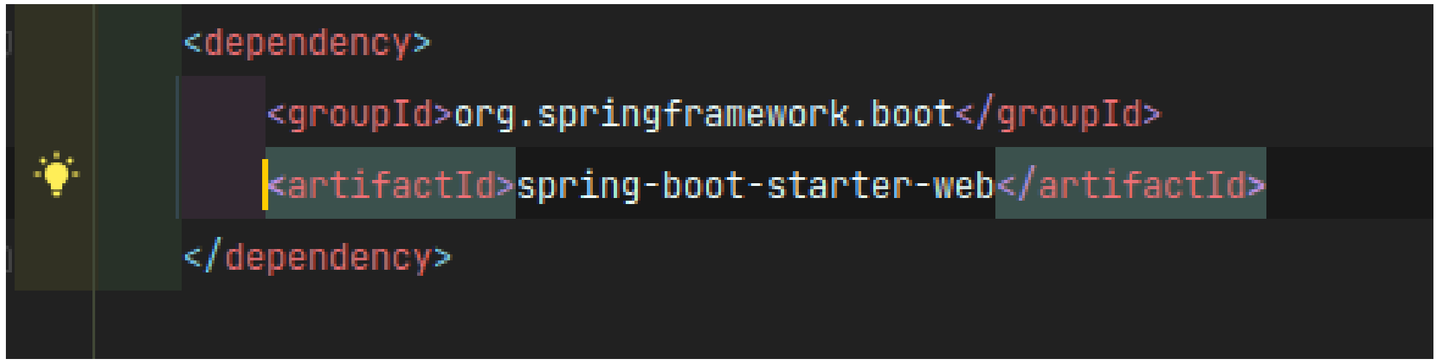
06 - Service

Pour suivre l'architecture en couche, la couche métier est mise en place. Elle permet d'appliquer des traitements spécifiques sur les données avant l'interaction avec la base de données.
Chaque classe aura l'annotation @Service (une autre spécialisation de @Component).

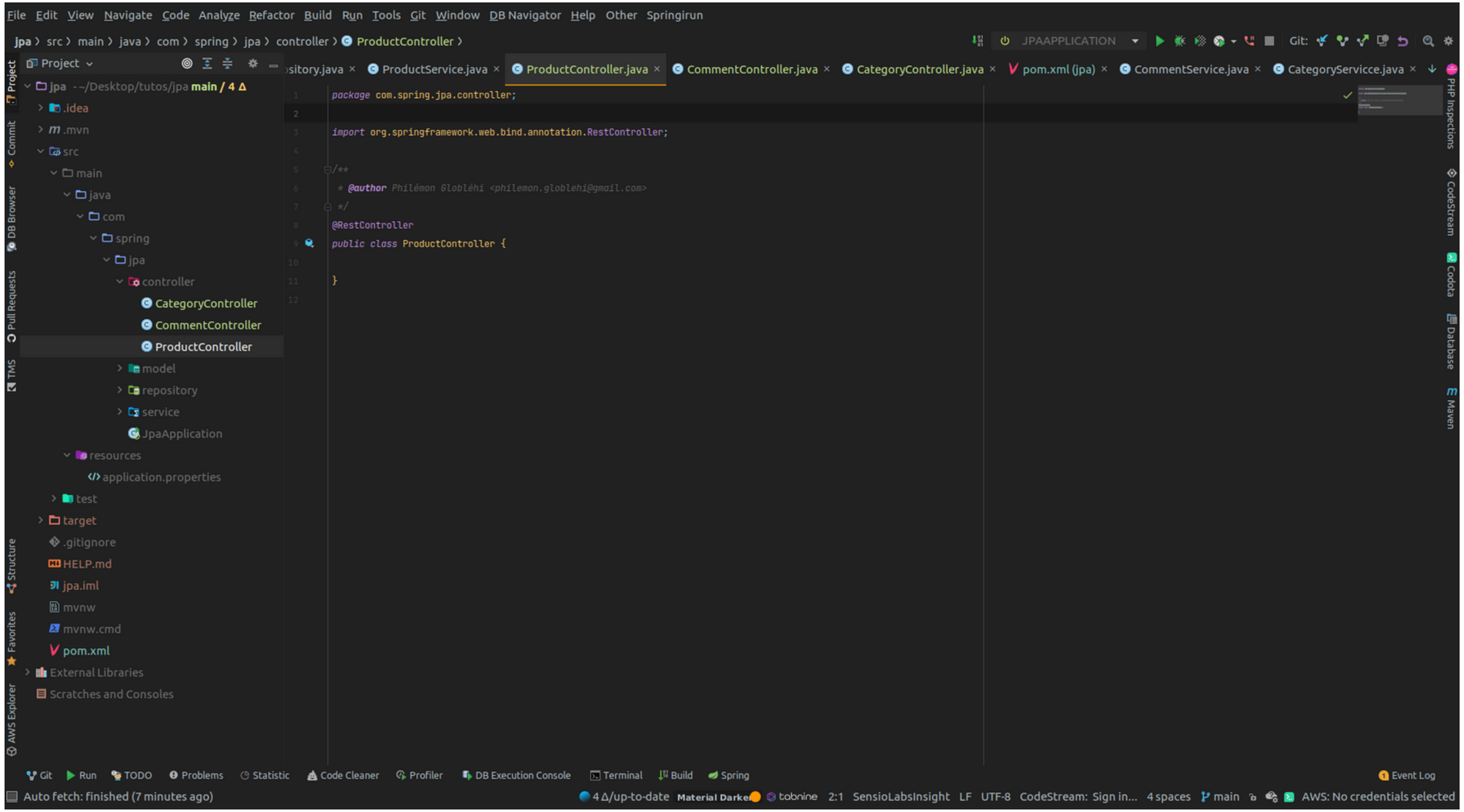


07 - Controlleur

Pour exposer le contenu de nos différentes ressources, nous allons créer les controlleurs dans le package controller. Déjà ajouter dans le pom.xml la dépendance Spring Web dans la section <dependencies>...</dependencies>
Ensuite lancer la build de l'application avec la commande mvn install

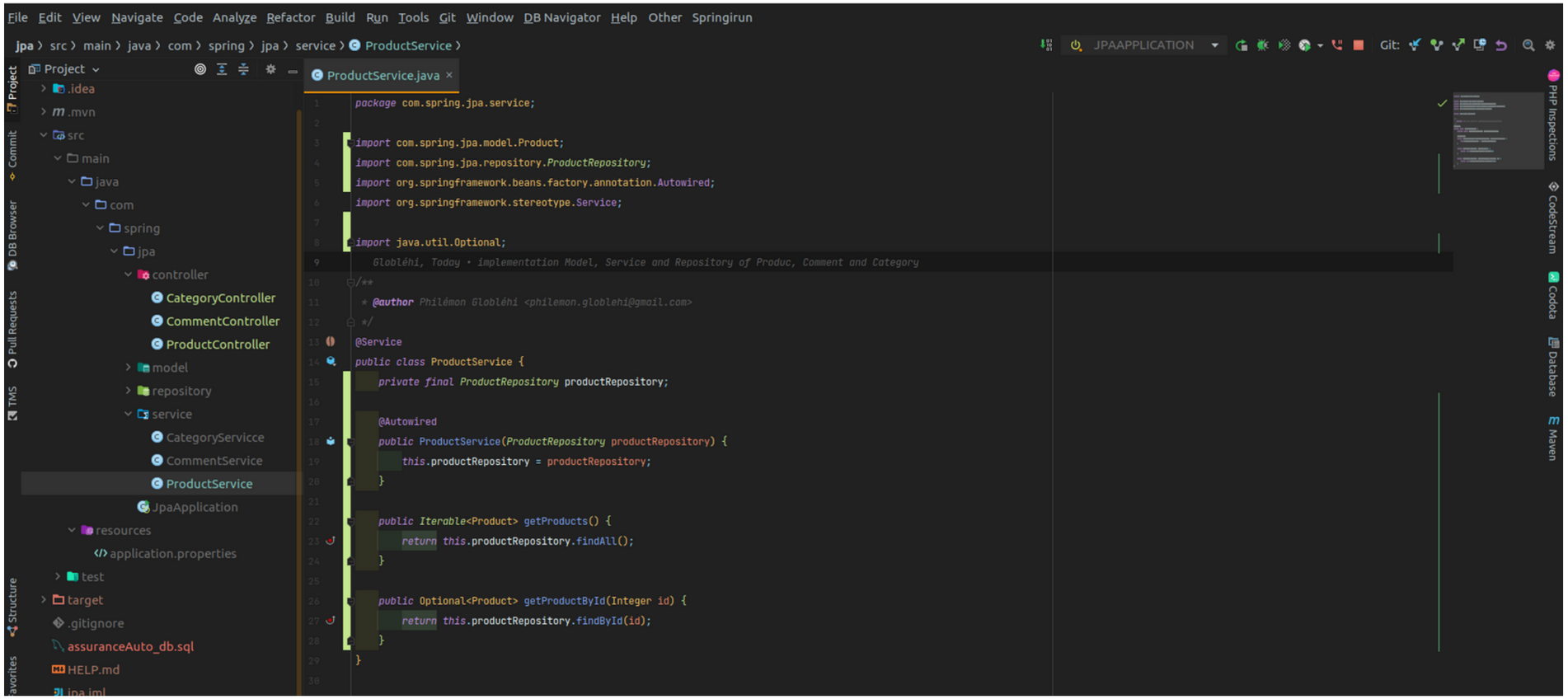


Chaque classe aura l'annotation @RestController (une autre spécialisation de @Component).
Elle indique à Spring d'insérer le retour de la méthode au format JSON dans le corps de la réponse HTTP.
Grâce à cela, les applications qui vont communiquer avec votre API accéderont au résultat de leur requête en parsant la réponse HTTP.

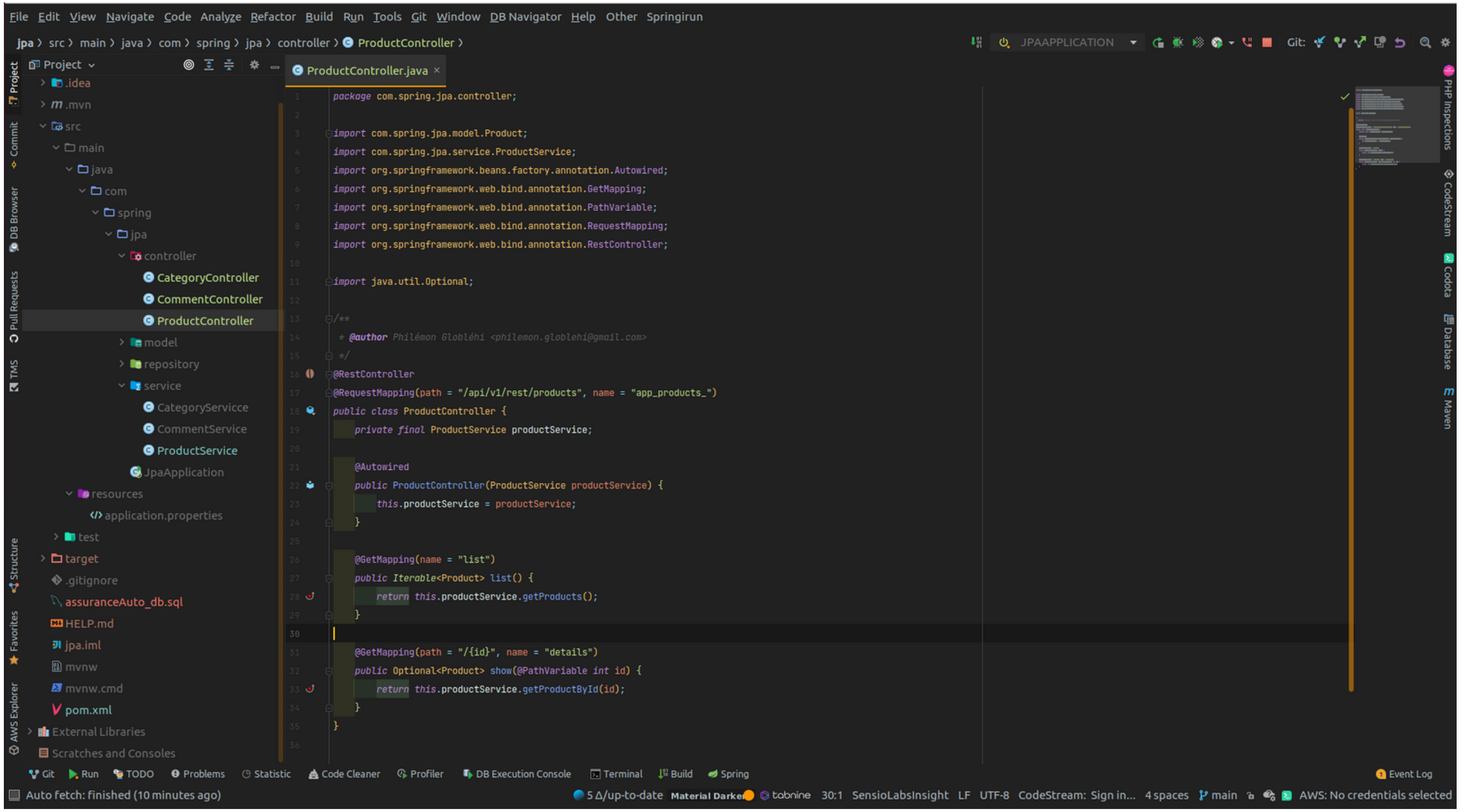


08 - Récupérer une ressource

- L'interface CrudRepository donne accès à de nombreuses méthodes pour interagir avec la base de données.
- Pour récupérer un ensemble de données, nous avons utilisé findAll() qui renvoie une liste d'objets correspondant à toutes les données de la table associée à l'entité concernée.
- Pour récupérer une donnée précise, nous avons utilisé findById(Integer id), qui renvoie un unique objet correspondant à l'ID demandé.

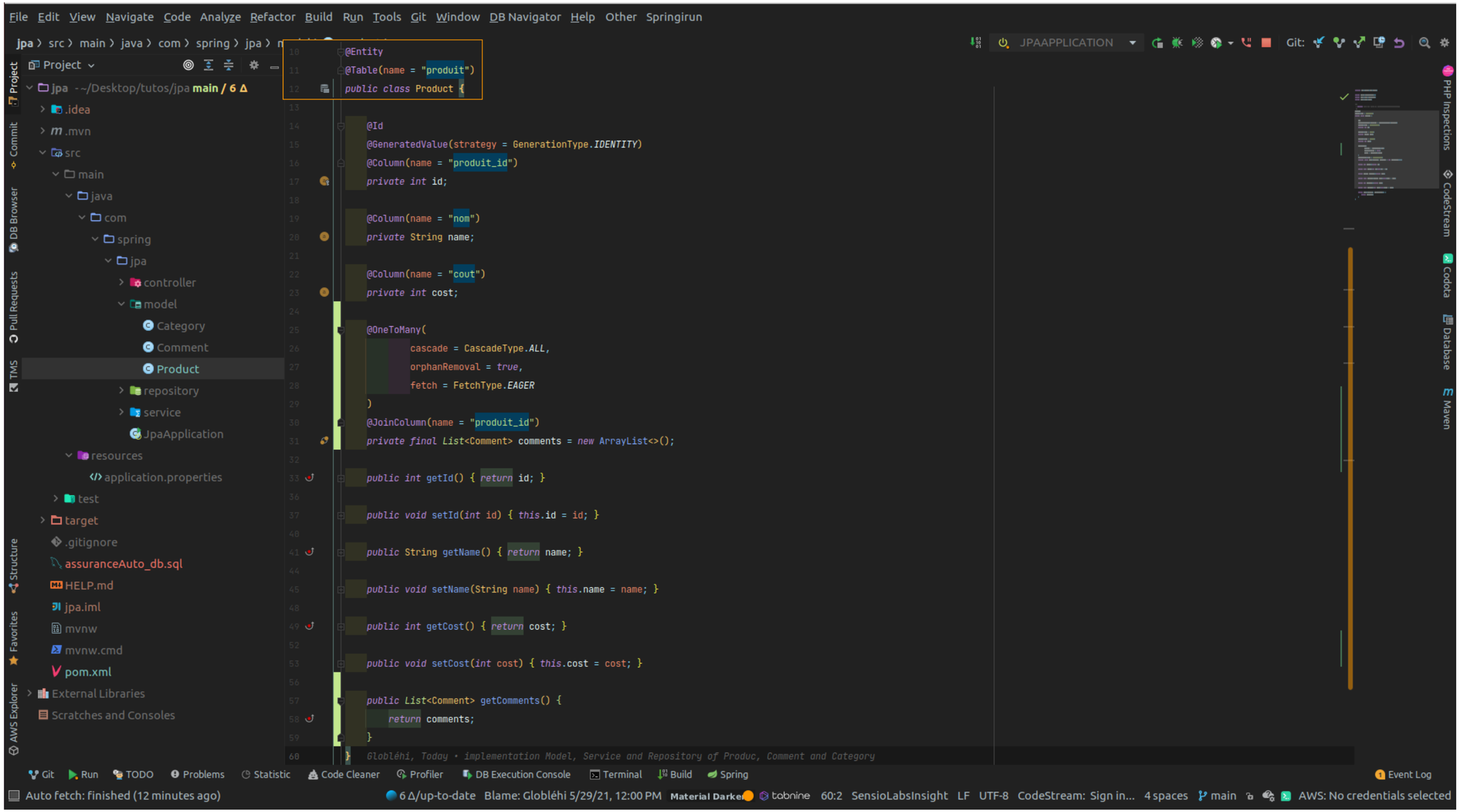


- Exposer la ressource via le controlleur

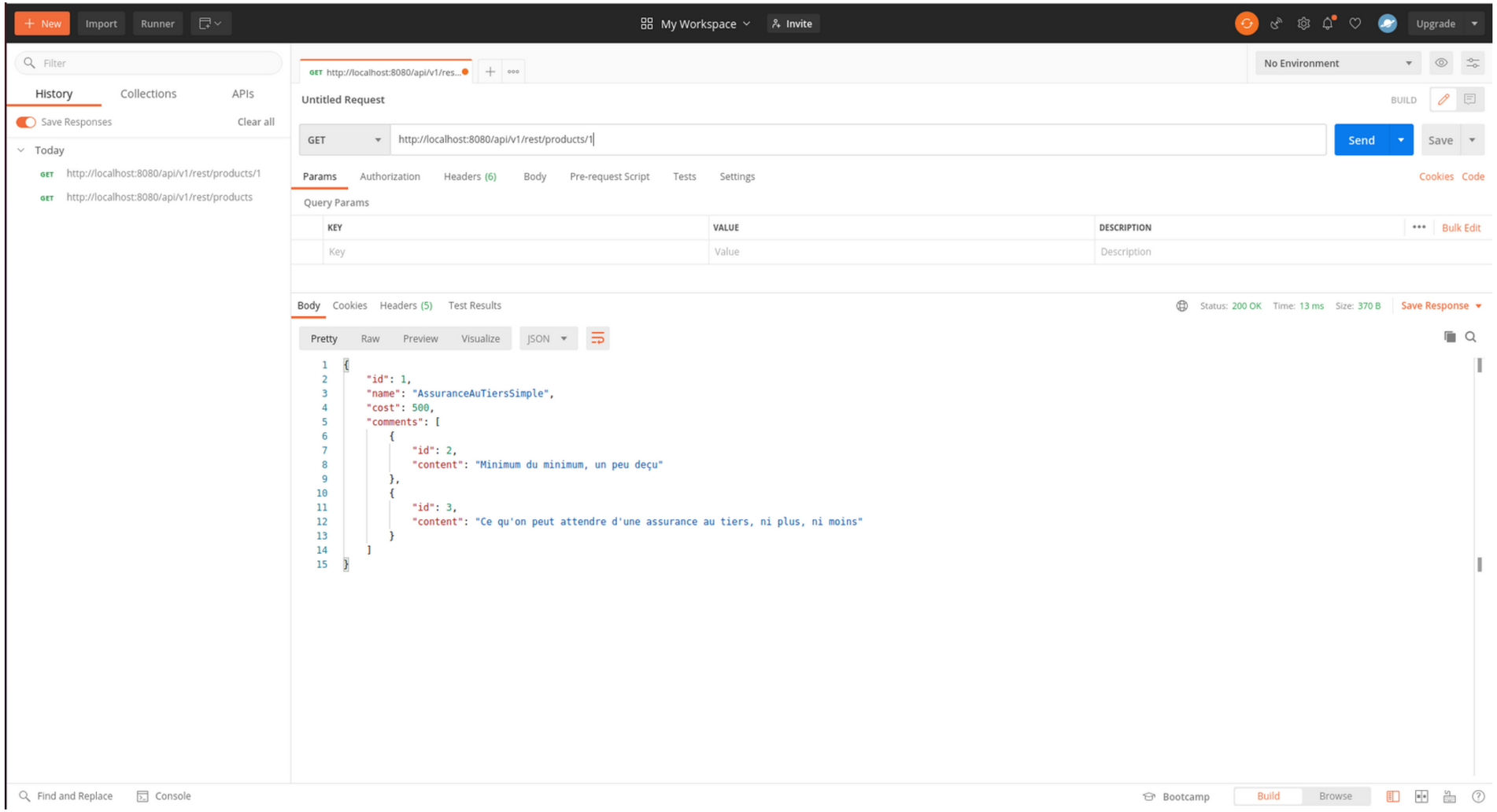


10 - Relations unidirectionnelles(OneToMany)

- @OneToMany : cette annotation permet de spécifier une relation ‘une à plusieurs’. Pour un produit, il y a plusieurs commentaires possibles.
- La propriété ‘cascade’ permet de définir quel impact l’action sur une entité aura sur son entité associée. Le type ‘ALL’ signifie que toutes les actions sur l’entité Produit seront propagées sur l’entité Commentaires. Exemple : si on supprime le produit, les commentaires associés seront également supprimés.
- La propriété orphanRemoval=true permet d’activer un mécanisme qui garantit la non-existence de commentaire orphelin de son produit. Si on supprime un commentaire de la liste des commentaires du Product, alors le commentaire devient orphelin, et il est supprimé de la base de données.
- La propriété fetch possède la valeur EAGER, et cela signifie qu’à la récupération du produit, tous les commentaires seront également récupérés.
- @JoinColumn : Cette annotation permet d’indiquer le nom de la clé étrangère dans la table de l’entité concernée.

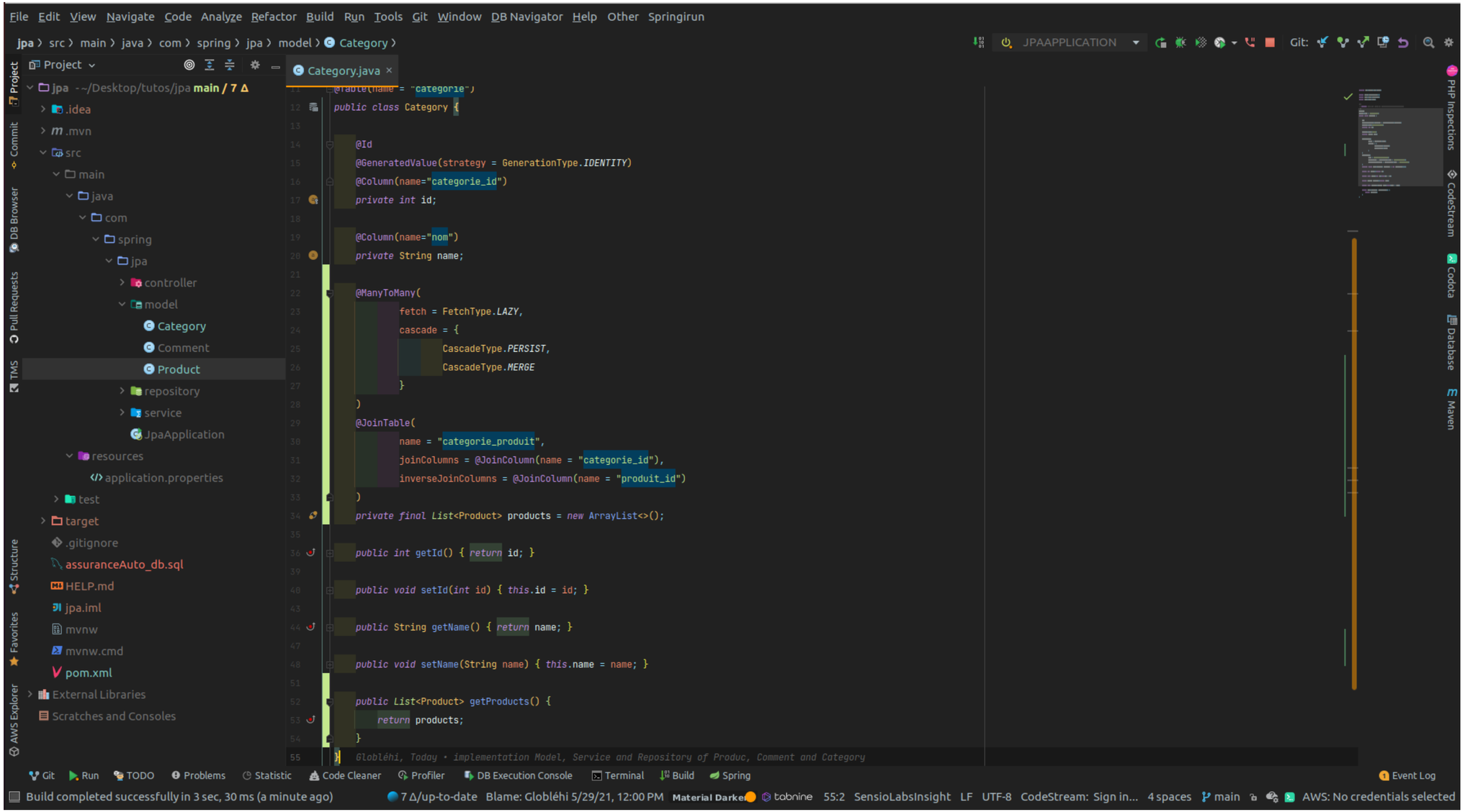

File Edit View Navigate Code Analyze Refactor Build Run Tools Git Window DB Navigator Help Other Springirun
jpa › src › main › java › com › spring › jpa
JPAAPPLICATION
Git:
Project
jpa ~/Desktop/tutos/jpa main / 6 Δ
.idea
.mvn
src
main
java
com
spring
jpa
controller
model
Category
Comment
Product
repository
service
JpaApplication
resources
application.properties
test
target
.gitignore
assuranceAuto_db.sql
HELP.md
jpa.iml
mvnw
mvnw.cmd
pom.xml
External Libraries
Scratches and Consoles
@Entity
@Table(name = "produit")
public class Product {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "produit_id")
private int id;
@Column(name = "nom")
private String name;
@Column(name = "cout")
private int cost;
@OneToMany(
cascade = CascadeType.ALL,
orphanRemoval = true,
fetch = FetchType.EAGER
)
@JoinColumn(name = "produit_id")
private final List<Comment> comments = new ArrayList<>();
public int getId() { return id; }
public void setId(int id) { this.id = id; }
public String getName() { return name; }
public void setName(String name) { this.name = name; }
public int getCost() { return cost; }
public void setCost(int cost) { this.cost = cost; }
public List<Comment> getComments() {
return comments;
}
Globléhi, Today • implementation Model, Service and Repository of Produc, Comment and Category
PHP Inspections
CodeStream
Codota
Database
Maven
Git
Run
TODO
Problems
Statistic
Code Cleaner
Profiler
DB Execution Console
Terminal
Build
Spring
Event Log
Auto fetch: finished (12 minutes ago)
6 Δ/up-to-date
Blame: Globléhi 5/29/21, 12:00 PM
Material Darker
tabnine
60:2
SensioLabsInsight
LF
UTF-8
CodeStream: Sign in...
4 spaces
main
AWS: No credentials selected

Test via postman

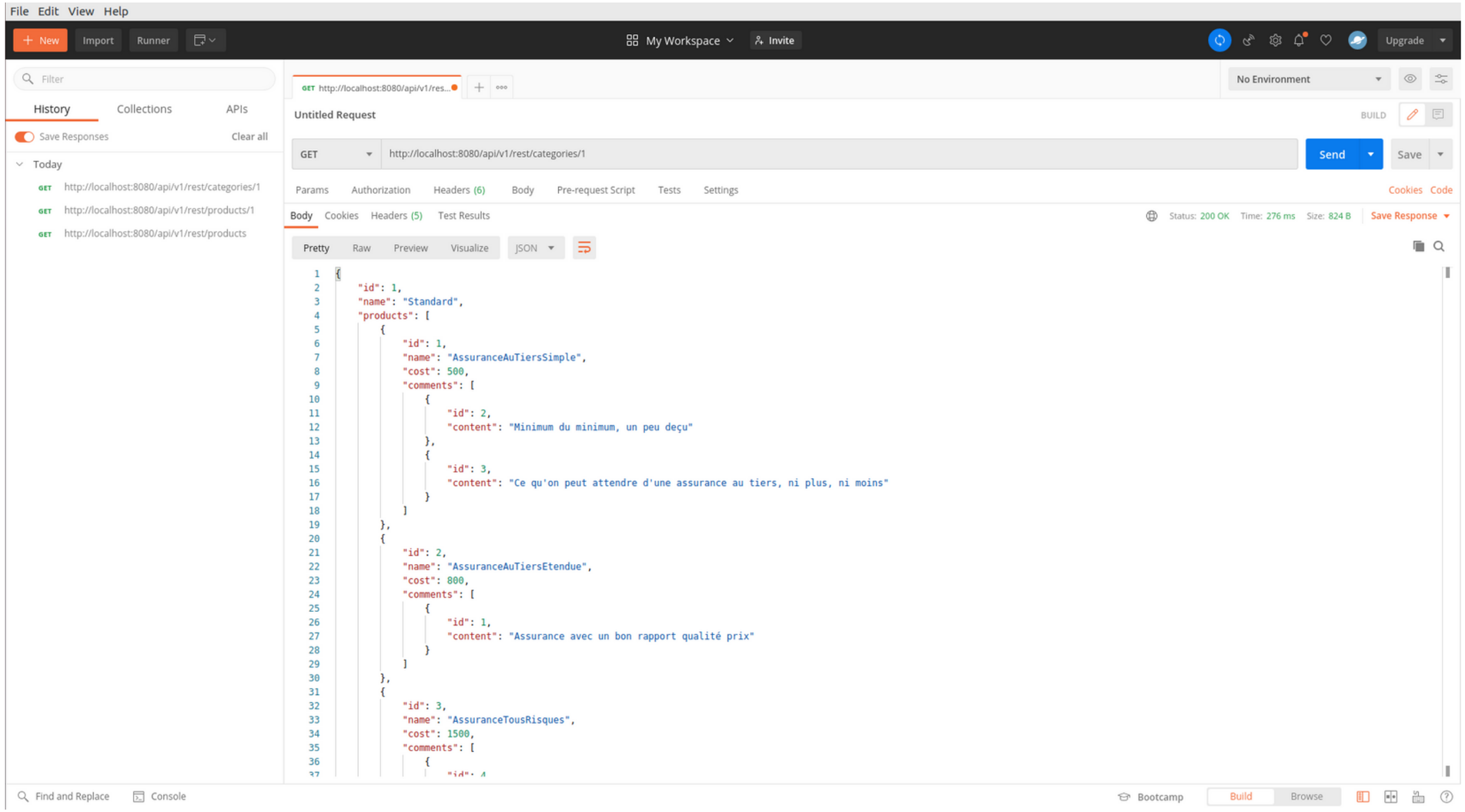


11 - Relations unidirectionnelles(ManyToMany)

- @ManyToMany : cette annotation permet de spécifier une relation ‘plusieurs à plusieurs’. Pour un produit, il y a plusieurs catégories possibles et pour une catégorie, il y a plusieurs produits possibles.
- La propriété fetch possède la valeur LAZY, et cela signifie qu’à la récupération de la catégorie, les produits ne sont pas récupérés. Par voie de conséquence, les performances sont meilleures (la requête est plus légère)
- Cascade : Je spécifie donc uniquement PERSIST et MERGE, la cascade s’applique donc tant en création qu’en modification.
- Il est également nécessaire d’utiliser l’annotation @JoinTable avec les propriétés suivantes :
    - name : correspond au nom de la table de jointure en base de données ;
    - joinColumns correspond à la clé étrangère dans la table de jointure ;
    inverseJoinColumns correspond à la clé étrangère dans la table de jointure de la seconde entité concernée par la relation.

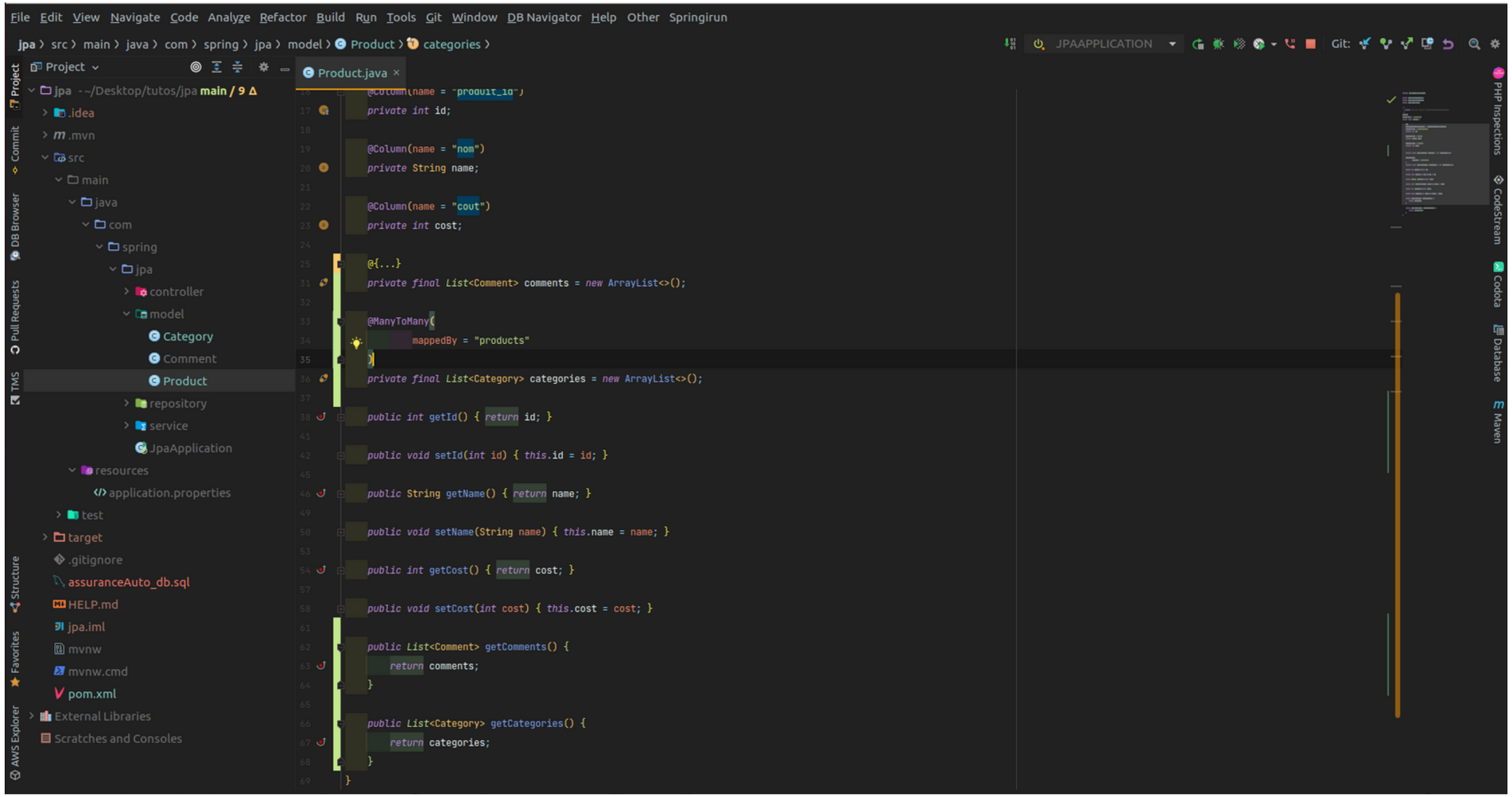

File Edit View Navigate Code Analyze Refactor Build Run Tools Git Window DB Navigator Help Other Springirun
jpa › src › main › java › com › spring › jpa › model › Category
JPAAPPLICATION
Git:
Project
jpa ~/Desktop/tutos/jpa main / 7 Δ
.idea
.mvn
src
main
java
com
spring
jpa
controller
model
Category
Comment
Product
repository
service
JpaApplication
resources
application.properties
test
target
.gitignore
assuranceAuto_db.sql
HELP.md
jpa.iml
mvnw
mvnw.cmd
pom.xml
External Libraries
Scratches and Consoles
Category.java
@Table(name = "categorie")
public class Category {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name="categorie_id")
private int id;
@Column(name="nom")
private String name;
@ManyToMany(
fetch = FetchType.LAZY,
cascade = {
CascadeType.PERSIST,
CascadeType.MERGE
}
)
@JoinTable(
name = "categorie_produit",
joinColumns = @JoinColumn(name = "categorie_id"),
inverseJoinColumns = @JoinColumn(name = "produit_id")
)
private final List<Product> products = new ArrayList<>();
public int getId() { return id; }
public void setId(int id) { this.id = id; }
public String getName() { return name; }
public void setName(String name) { this.name = name; }
public List<Product> getProducts() {
return products;
}
}
Globléhi, Today • implementation Model, Service and Repository of Produc, Comment and Category
PHP Inspections
CodeStream
Codota
Database
Maven
Git
Run
TODO
Problems
Statistic
Code Cleaner
Profiler
DB Execution Console
Terminal
Build
Spring
Event Log
Build completed successfully in 3 sec, 30 ms (a minute ago)
7 Δ/up-to-date
Blame: Globléhi 5/29/21, 12:00 PM
Material Darker
tabnine
55:2
SensioLabsInsight
LF
UTF-8
CodeStream: Sign in...
4 spaces
main
AWS: No credentials selected

Test via postman



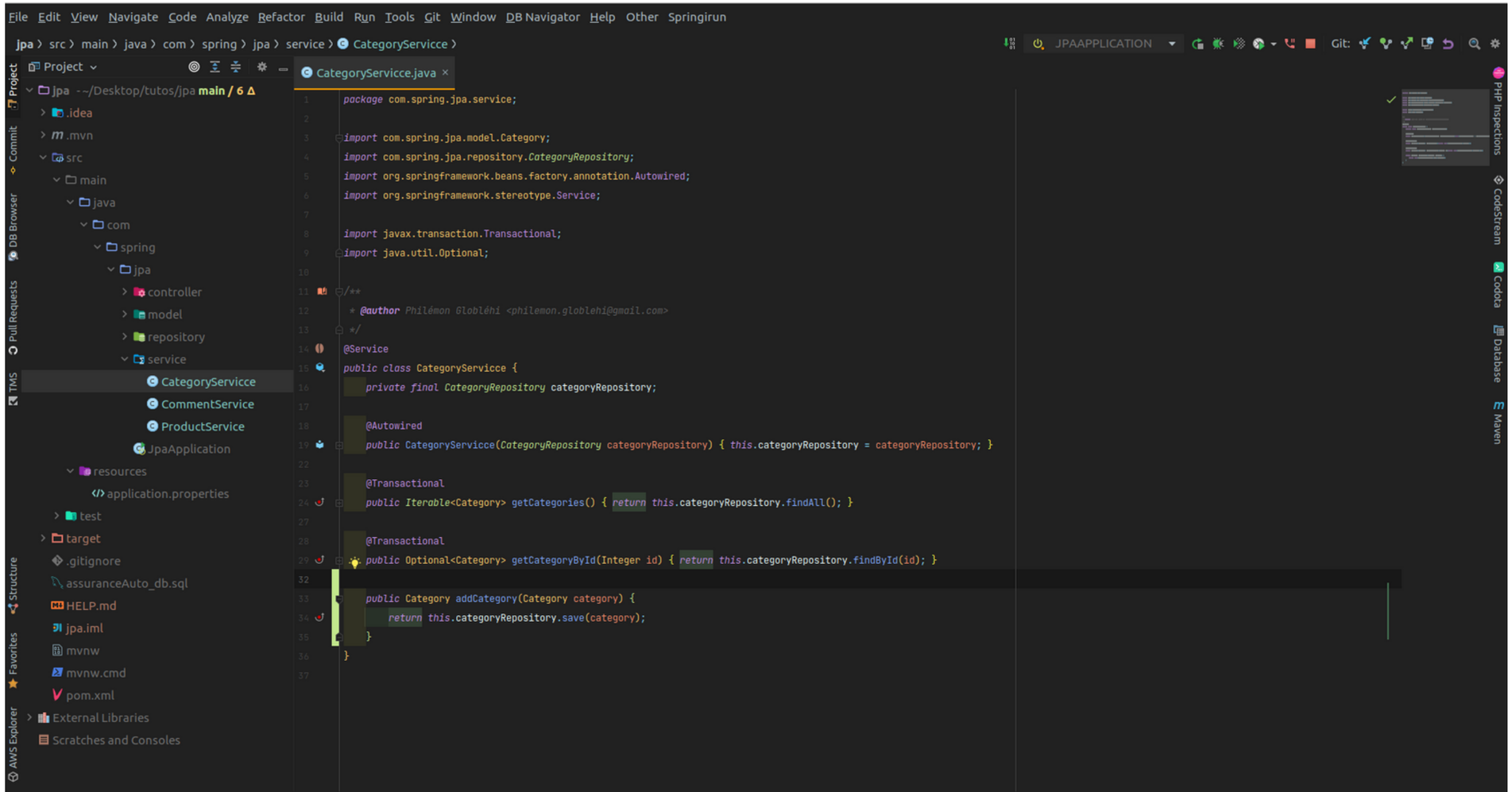
12 - Relations bidirectionnelles(ManyToMany)

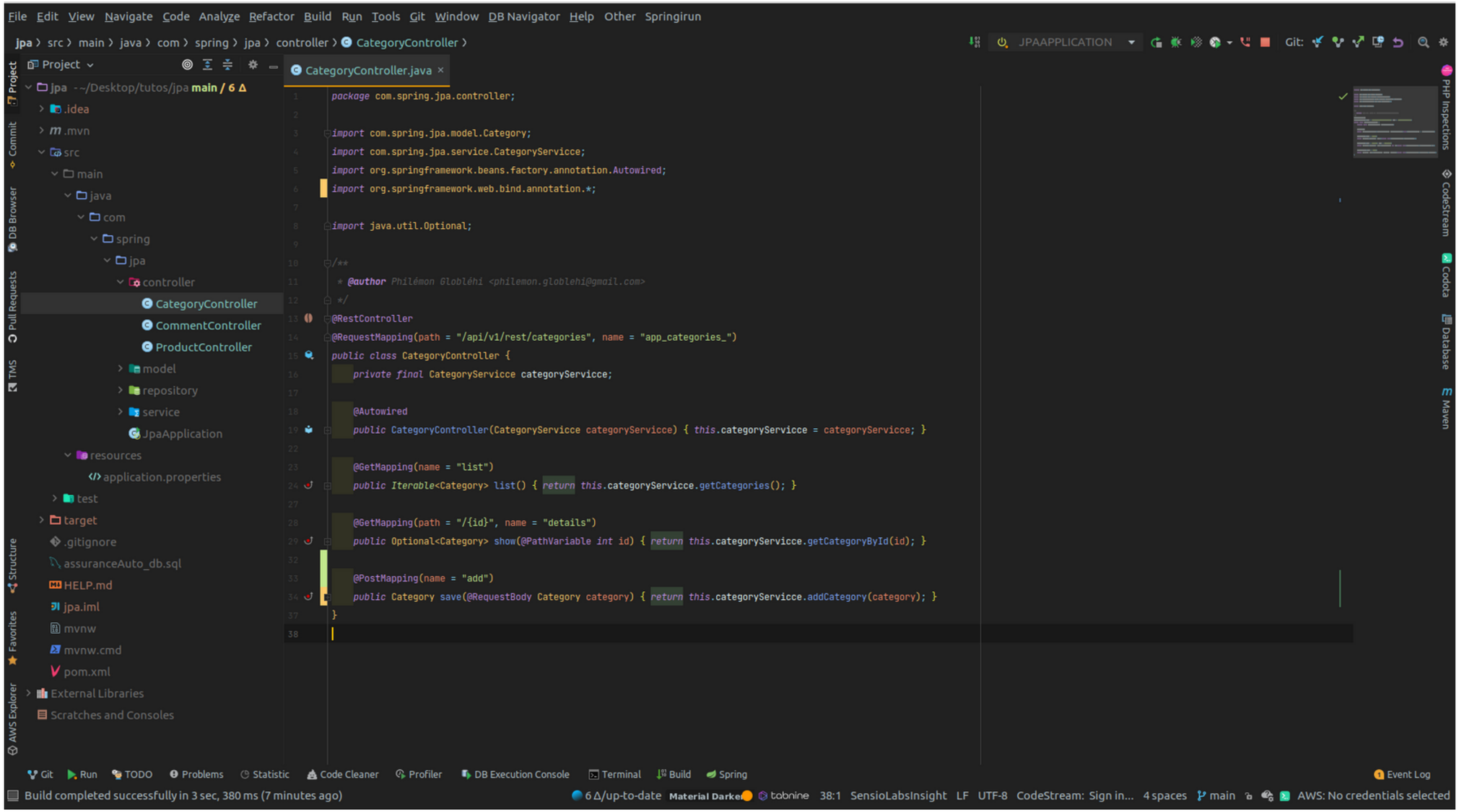
@ManyToMany est de rigueur, et l'unique propriété à utiliser est mappedBy. Cette propriété est valide dans le contexte de la bidirectionnalité, car elle indique le nom de l'attribut dans l'autre entité.
Étant donné que dans la classe Category, l'annotation @ManyToMany est apposée sur l'attribut nommé "products", alors la valeur de la propriété mappedBy est "products".



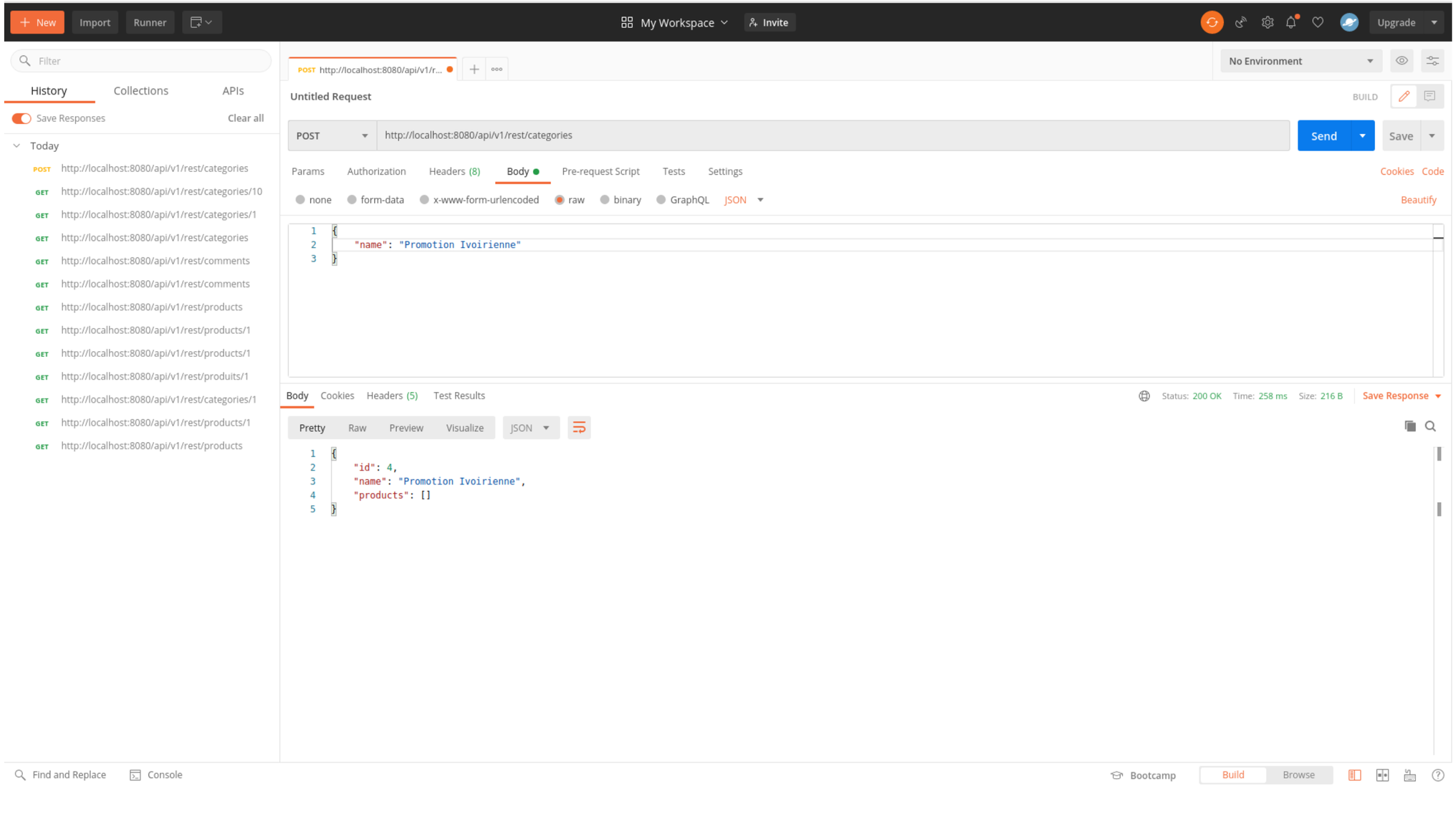
13 - Persister des données

Spring Data JPA va grandement nous simplifier la tâche, et ainsi nous faire gagner du temps !
Tout comme pour la récupération de données, l'interface CrudRepository nous offre des méthodes déjà implémentées.
Les méthodes save et saveAll permettent de sauvegarder (ou persister) en base de données



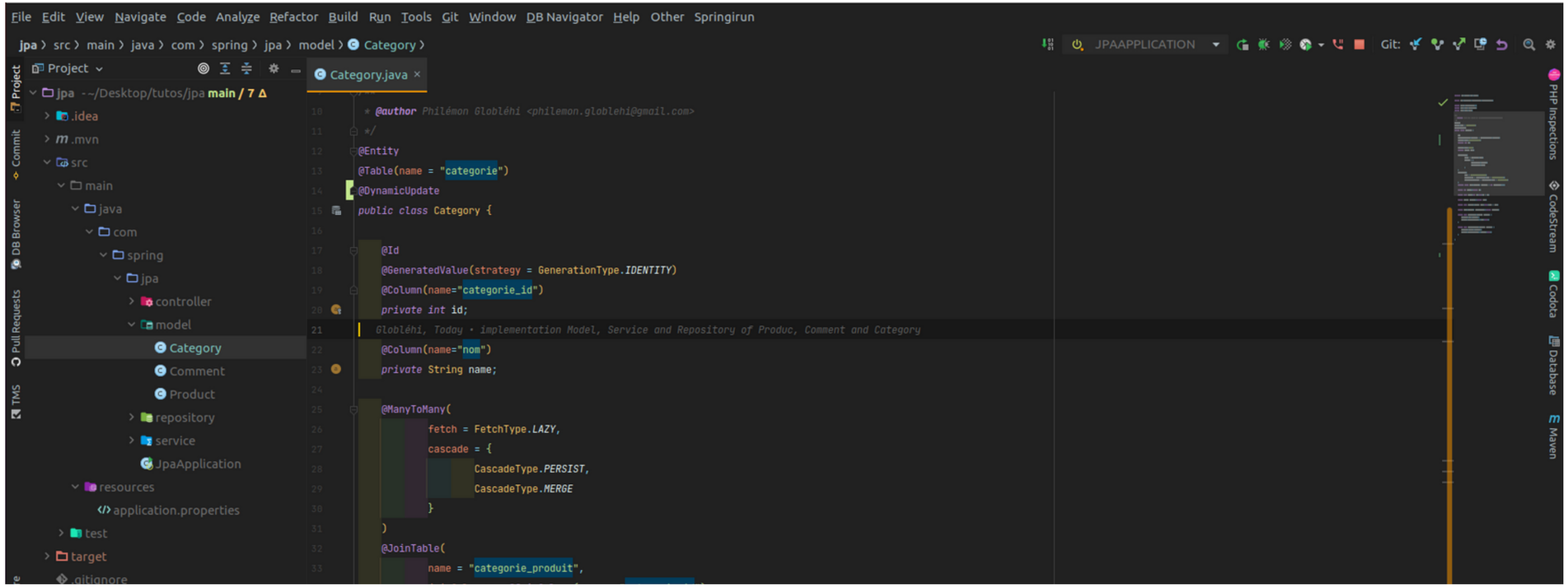

File Edit View Navigate Code Analyze Refactor Build Run Tools Git Window DB Navigator Help Other Springirun
jpa › src › main › java › com › spring › jpa › controller › CategoryController
JPAAPPLICATION
Git:
Project
jpa ~/Desktop/tutos/jpa main / 6 Δ
.idea
.mvn
src
main
java
com
spring
jpa
controller
CategoryController
CommentController
ProductController
model
repository
service
JpaApplication
resources
application.properties
test
target
.gitignore
assuranceAuto_db.sql
HELP.md
jpa.iml
mvnw
mvnw.cmd
pom.xml
External Libraries
Scratches and Consoles
CategoryController.java
package com.spring.jpa.controller;
import com.spring.jpa.model.Category;
import com.spring.jpa.service.CategoryServicce;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.Optional;
/**
* @author Philémon Globléhi <philemon.globlehi@gmail.com>
*/
@RestController
@RequestMapping(path = "/api/v1/rest/categories", name = "app_categories_")
public class CategoryController {
private final CategoryServicce categoryServicce;
@Autowired
public CategoryController(CategoryServicce categoryServicce) { this.categoryServicce = categoryServicce; }
@GetMapping(name = "list")
public Iterable<Category> list() { return this.categoryServicce.getCategories(); }
@GetMapping(path = "/{id}", name = "details")
public Optional<Category> show(@PathVariable int id) { return this.categoryServicce.getCategoryById(id); }
@PostMapping(name = "add")
public Category save(@RequestBody Category category) { return this.categoryServicce.addCategory(category); }
}
PHP Inspections
CodeStream
Codota
Database
Maven
Git
Run
TODO
Problems
Statistic
Code Cleaner
Profiler
DB Execution Console
Terminal
Build
Spring
Event Log
Build completed successfully in 3 sec, 380 ms (7 minutes ago)
6 Δ/up-to-date
Material Darker
tabnine
38:1
SensioLabsInsight
LF
UTF-8
CodeStream: Sign in...
4 spaces
main
AWS: No credentials selected

Test via postman



14 - Modifier des données

La création et la mise à jour d'un objet sont gérées par les mêmes méthodes save(S entity) et saveAll(Iterable<S> entities).
Spring Data JPA va inspecter le contenu de l'objet que nous lui fournissons, et pourra identifier s'il est déjà existant en base ou non, grâce à son identifiant unique (la clé primaire).
Ajoutons l'annotation @DynamicUpdate à nos models. Cette annotation doit être appliquée sur l'entité (dans notre exemple, la classe Category). Si elle est présente, alors la requête SQL update exécutera un set uniquement sur les valeurs qui ont changé et permet de booster vos performances



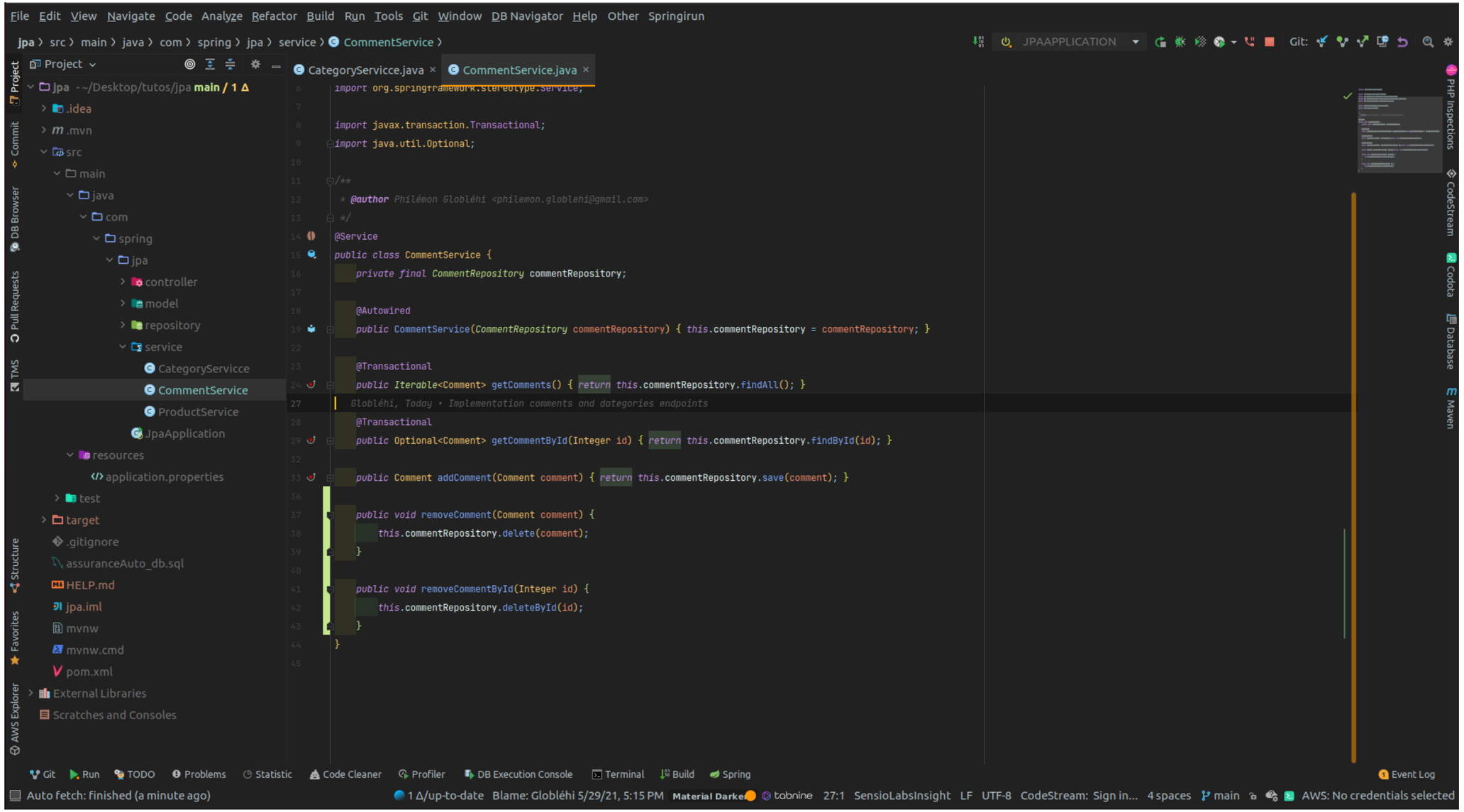
15 - Supprimer des données

Spring Data JPA tient vraiment à nous faciliter la tâche pour la suppression

Pas moins de 4 méthodes sont à notre disposition :

- delete(S entity)
- deleteAll()
- deleteAll(Iterable<S> entities)
- deleteById(Integer id)

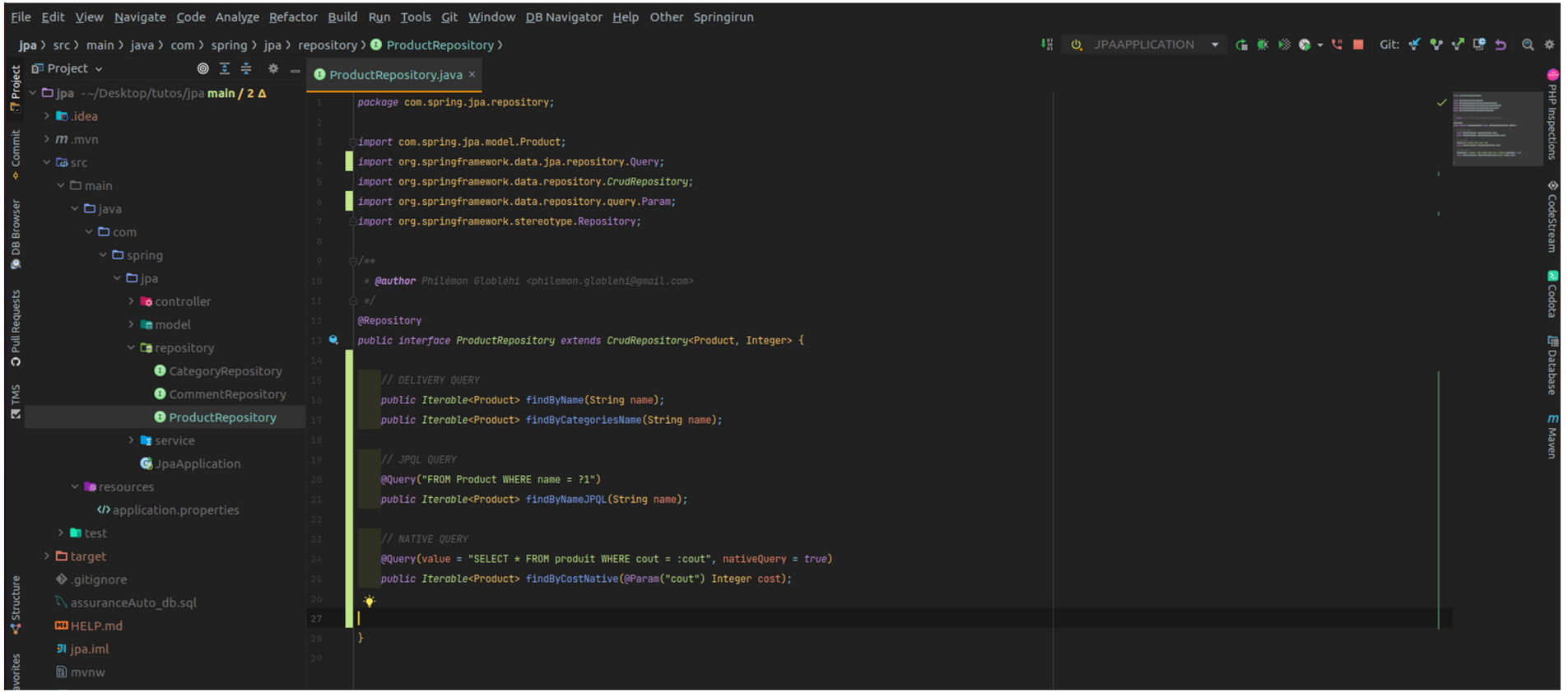
Il est important, dans le cas de la suppression, de bien tenir compte du type de CASCADE qui a été spécifié au niveau des relations. Un CascadeType.ALL ou un CascadeType.REMOVE déclenche une suppression en cascade en fonction des associations. Et cela peut avoir de graves conséquences !


File Edit View Navigate Code Analyze Refactor Build Run Tools Git Window DB Navigator Help Other Springirun
jpa › src › main › java › com › spring › jpa › service › CommentService ›
JPAAPPLICATION
Git:
Project
jpa ~/Desktop/tutos/jpa main / 1 Δ
.idea
.mvn
src
main
java
com
spring
jpa
controller
model
repository
service
CategoryServicce
CommentService
ProductService
JpaApplication
resources
application.properties
test
target
.gitignore
assuranceAuto_db.sql
HELP.md
jpa.iml
mvnw
mvnw.cmd
pom.xml
External Libraries
Scratches and Consoles
CategoryServicce.java
CommentService.java
import org.springframework.stereotype.Service;
import javax.transaction.Transactional;
import java.util.Optional;
/**
* @author Philémon Globléhi <philemon.globlehi@gmail.com>
*/
@Service
public class CommentService {
private final CommentRepository commentRepository;
@Autowired
public CommentService(CommentRepository commentRepository) { this.commentRepository = commentRepository; }
@Transactional
public Iterable<Comment> getComments() { return this.commentRepository.findAll(); }
Globléhi, Today • Implementation comments and dategories endpoints
@Transactional
public Optional<Comment> getCommentById(Integer id) { return this.commentRepository.findById(id); }
public Comment addComment(Comment comment) { return this.commentRepository.save(comment); }
public void removeComment(Comment comment) {
this.commentRepository.delete(comment);
}
public void removeCommentById(Integer id) {
this.commentRepository.deleteById(id);
}
}
PHP Inspections
CodeStream
Codota
Database
Maven
Git Run TODO Problems Statistic Code Cleaner Profiler DB Execution Console Terminal Build Spring
Event Log
Auto fetch: finished (a minute ago)
1 Δ/up-to-date Blame: Globléhi 5/29/21, 5:15 PM Material Darker tabnine 27:1 SensioLabsInsight LF UTF-8 CodeStream: Sign in... 4 spaces main AWS: No credentials selected

16 - Requêtes personnalisées

Les requêtes personnalisées sont des requêtes qui ne sont pas incluses par défaut dans l'interface CrudRepository, on a :

- les requêtes dérivées;
- les requêtes JPQL ;
- les requêtes natives SQL.



Les requêtes dérivées

À la ligne 14, j'ai rajouté le prototype d'une nouvelle méthode, public Iterable<Product> findByName(String name);
Le nom de cette méthode findByName n'a pas été construit au hasard :

- Le mot find correspond à la récupération de données.
- Le mot By est un mot clé utilisé par Spring Data JPA pour indiquer que la requête aura un paramètre.
- Le mot Name correspond à l'attribut de l'entité qui est notre critère de recherche.

Cette méthode est une requête dérivée (Derived Query) car nous n'allons pas en écrire l'implémentation. Spring Data JPA est capable de générer le code nécessaire à partir du prototype de la méthode.

les requêtes dérivées ont la capacité de traverser les relations .
La ligne 15 définit une nouvelle méthode findByCategoriesName :

- find pour la récupération de données ;
- By pour introduire des critères ;
- Categories car l'attribut au sein de l'entité Product se nomme categories ;
- Name car au sein de l'entité Category il y a l'attribut name.

Les requêtes JPQL

JPQL signifie Java Persistence Query Language. Ce langage de requête est défini au sein de l'API JPA, et permet d'exécuter des requêtes via les entités et non via les tables directement.
La ligne 21 définit une méthode findByNameJPQL utilise l'annotation @Query avec une requête JPQL.
On note qu'il n'y a pas de clause SELECT, et ce n'est pas le nom de la table mais le nom de l'entité qui est indiqué après le FROM.

Les requêtes natives

Les requêtes SQL natives permettent d'exécuter des requêtes SQL classiques en manipulant les tables directement.
La ligne 25 définit une méthode findByNameJPQL utilise l'annotation @Query avec une requête JPQL
La méthode findByCostNative utilise l'annotation @Query avec une requête SQL native.
3 aspects particuliers :

1. La requête est dans une propriété value de l'annotation.
2. La propriété nativeQuery de l'annotation doit être à true.
3. Le paramètre au sein de la méthode doit être annoté @Param afin que l'association soit faite entre le paramètre et la requête SQL.

Merci

Philémon Globléhi, Back-end Developer & DevOps Lover
Github: https://github.com/philemongloblehi
Linkedin: https://www.linkedin.com/in/philemon-globlehi/