

Compte Rendu - Travail à Rendre :

**DEV BACK END
WITH SPRING**

Réalisée par :

- KARA MARIAM

Encadré par :

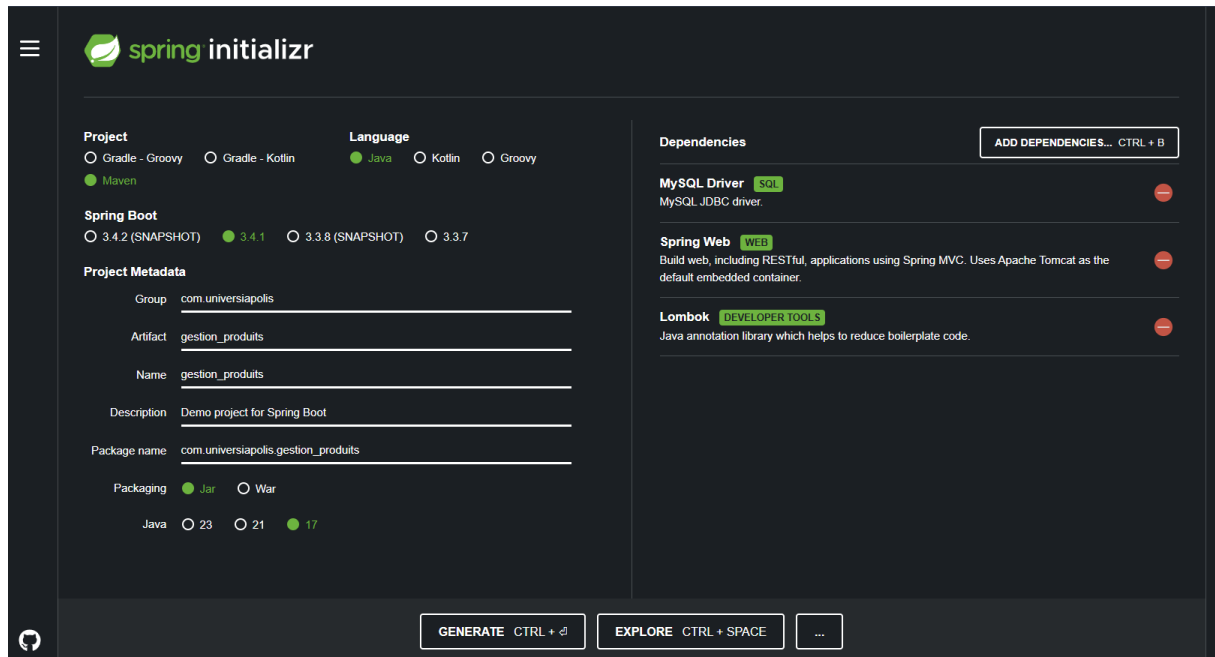
- M. ACHTCHTY

Générer un projet Spring Boot

Spring Initializr est un outil web qui permet de générer un projet Spring Boot avec les dépendances et la configuration de base nécessaires pour démarrer rapidement le développement d'une application. Il fournit une interface simple pour sélectionner les composants que vous souhaitez inclure dans votre projet, comme Spring Web, Spring Data JPA, et bien d'autres.

1. Accéder à Spring Initializr

Allez sur le site de Spring Initializr : <https://start.spring.io/>



The screenshot shows the Spring Initializr web interface. It has a dark theme. On the left, there's a sidebar with a hamburger menu icon. The main area is divided into sections: 'Project' with radio buttons for 'Gradle - Groovy', 'Gradle - Kotlin', 'Maven' (selected), and 'Language' with radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'. Below that is 'Spring Boot' with radio buttons for '3.4.2 (SNAPSHOT)', '3.4.1' (selected), '3.3.8 (SNAPSHOT)', and '3.3.7'. The 'Project Metadata' section has input fields for 'Group' (com.universiapolis), 'Artifact' (gestion_produits), 'Name' (gestion_produits), 'Description' (Demo project for Spring Boot), and 'Package name' (com.universiapolis.gestion_produits). There are also radio buttons for 'Packaging' ('Jar' selected, 'War' unselected) and 'Java' ('23' unselected, '21' unselected, '17' selected). On the right, the 'Dependencies' section has a button 'ADD DEPENDENCIES... CTRL + B' and a list of selected dependencies: 'MySQL Driver' (SQL), 'Spring Web' (WEB), and 'Lombok' (DEVELOPER TOOLS). Each dependency has a red minus button to remove it. At the bottom, there are buttons for 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and a three-dot menu.

2. Configuration du projet

Sur la page d'accueil de Spring Initializr, vous pouvez configurer les options suivantes :

✓ **Project** :

Choisissez le type de projet :

- Maven Project : Si vous utilisez Maven comme outil de gestion de dépendances.

✓ **Language** :

Choisissez le langage de programmation :

- Java : Le langage principal pour Spring Boot.

✓ **Spring Boot Version** :

Sélectionnez la version de Spring Boot que vous souhaitez utiliser.

✓ **Project Metadata** :

- **Group** : Le groupe d'identification du projet (par exemple, com.universiapolis).
- **Artifact** : Le nom de l'artefact (par exemple, gestion-produits).
- **Name** : Le nom du projet.
- **Description** : Une brève description de votre projet.
- **Package Name** : Le nom du package Java principal.

✓ **Packaging :**

Choisissez entre Jar ou War. Pour une application Spring Boot classique, vous utilisez généralement Jar.

✓ **Java Version :**

Sélectionnez la version de Java que vous souhaitez utiliser pour votre projet.

3. Ajouter des dépendances

Sous la section Dependencies, vous pouvez ajouter les dépendances nécessaires pour votre projet. Quelques exemples courants :

- Spring Web : Pour construire des applications web avec Spring MVC.
- Spring Data JPA : Pour la gestion des bases de données relationnelles avec JPA.
- MySQL Driver : Si vous utilisez MySQL comme base de données.
- Spring Boot DevTools : Pour le rechargement automatique en développement.
- Spring Boot Starter Test : Pour les tests unitaires.

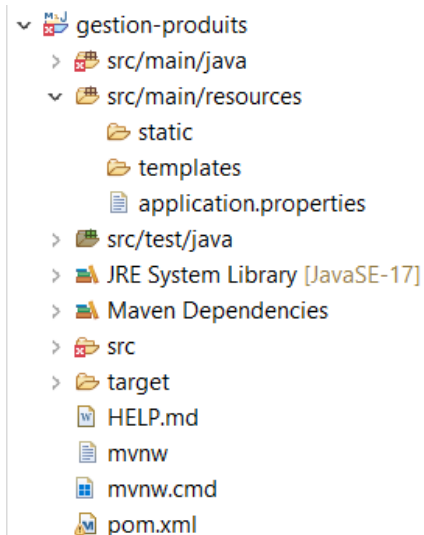
4. Générer le projet

Une fois que vous avez configuré votre projet et sélectionné les dépendances, cliquez sur le bouton Generate. Cela va télécharger un fichier .zip contenant la structure du projet avec les fichiers nécessaires.

5. Importer dans votre IDE

Décompressez le fichier .zip téléchargé et importez-le dans votre IDE (par exemple, IntelliJ IDEA, Eclipse, ou VS Code). Si vous utilisez Maven, l'IDE reconnaîtra automatiquement le fichier pom.xml et téléchargera les dépendances.

Le fichier pom.xml est le fichier de configuration principal pour un projet Maven. Il contient des informations sur le projet, ses dépendances, ses plugins, et d'autres paramètres de configuration.



Le fichier **application.properties** contient les configurations spécifiques à l'application Spring Boot.

```
application.prop... × Product.java ProductController.j... ProductRepository.j... GestionProduitsAp... IPro
1 spring.application.name=gestion-produits
2 spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
3 spring.datasource.url=jdbc:mysql://localhost:3306/gestion_produits?createDatabaseIfNotExist=true
4 spring.datasource.username=root
5 spring.datasource.password=
6 server.port=8085
7 spring.jpa.generate-ddl=true
8 spring.jpa.hibernate.ddl-auto=create
9 spring.jpa.show-sql=true
```

- ✓ **spring.application.name** : Le nom de l'application.
- ✓ **spring.datasource.driver-class-name** : Le driver JDBC pour MySQL.
- ✓ **spring.datasource.url** : L'URL de connexion à la base de données MySQL.
- ✓ **spring.datasource.username et spring.datasource.password** : Les informations d'identification pour la connexion à la base de données.
- ✓ **server.port** : Le port sur lequel l'application Spring Boot sera lancée (ici, 8085).
- ✓ **spring.jpa.generate-ddl** : Permet de générer automatiquement le schéma de la base de données (utile en développement).
- ✓ **spring.jpa.hibernate.ddl-auto** : Spécifie l'action de gestion du schéma (ici, create pour créer le schéma au démarrage).
- ✓ **spring.jpa.show-sql** : Affiche les requêtes SQL générées par Hibernate.

Package Entity

Le **package entity** contient les classes qui représentent les entités de votre base de données, c'est-à-dire les objets qui seront mappés sur les tables de la base de données.

La **classe Product** est une entité JPA qui représente un produit dans une application de gestion de produits. Elle est mappée à une table dans la base de données, où chaque instance de la classe correspond à une ligne dans cette table.

```
@Entity
@AllArgsConstructor @NoArgsConstructor @Data
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String description;
    private double price;
}
```

Points clés :

✓ Annotations JPA :

- **@Entity** : La classe est une entité JPA, ce qui signifie qu'elle sera persistée dans la base de données.
- **@Id** : Le champ id est la clé primaire de l'entité.
- **@GeneratedValue(strategy = GenerationType.IDENTITY)** : L'ID est généré automatiquement par la base de données (auto-incrément).

✓ Lombok :

- **@Data** : Génère automatiquement les méthodes getter, setter, toString(), equals(), et hashCode().
- **@AllArgsConstructor** : Génère un constructeur avec tous les champs comme paramètres.
- **@NoArgsConstructor** : Génère un constructeur sans arguments.

✓ Attributs :

- **id** : Identifiant unique du produit (clé primaire).
- **name** : Nom du produit.
- **description** : Description du produit.
- **price** : Prix du produit.

En résumé, cette classe définit la structure d'un produit avec ses attributs et les annotations nécessaires pour l'intégrer à un système de gestion de base de données relationnelle via JPA.

Package Metier

Le package metier dans une application Spring Boot représente la couche métier ou logique métier, qui contient les services responsables de la gestion des opérations principales liées aux entités de l'application.

Interface IProduit

```
6 public interface IProduit {
7
8     public Product addProduct(Product product);
9     public List<Product> getAllProducts();
10    public Product getProductById(Long id) ;
11    //public List<Product> searchProducts(String keyword);
12    public Product updateProduct(Long id, Product productDetails);
13    public void deleteProduct(Long id) ;
14 }
```

L'interface IProduit définit les méthodes que votre service métier doit implémenter pour gérer les produits. Ces méthodes incluent l'ajout, la récupération, la mise à jour et la suppression des produits.

- **addProduct** : Ajoute un produit à la base de données.
- **getAllProducts** : Récupère tous les produits de la base de données.
- **getProductById** : Récupère un produit spécifique par son identifiant.
- **updateProduct** : Met à jour un produit existant en fonction de son identifiant.
- **deleteProduct** : Supprime un produit de la base de données.

Implémentation de l'interface IProduit (classe IProductImpl)

La classe IProductImpl implémente l'interface IProduit. Elle utilise le ProductRepository pour interagir avec la base de données via Spring Data JPA.

```
@Service
public class IProductImpl implements IProduit{

    @Autowired
    private ProductRepository productRepository;

    @Override
    public Product addProduct(Product product) {
        return productRepository.save(product);
    }

    @Override
    public List<Product> getAllProducts() {
        return productRepository.findAll();
    }

    @Override
    public Product getProductById(Long id) {
        return productRepository.findById(id).orElse(null);
    }
}
```

Explication des méthodes

- ✓ **addProduct** : Utilise `productRepository.save(product)` pour ajouter un produit à la base de données. Cette méthode retourne l'objet `Product` après l'avoir enregistré.
- ✓ **getAllProducts** : Récupère tous les produits à l'aide de `productRepository.findAll()`. Cela retourne une liste de tous les produits présents dans la base de données.
- ✓ **getProductById** : Utilise `productRepository.findById(id)` pour récupérer un produit spécifique en fonction de son identifiant. Si le produit n'existe pas, il retourne `null`.
- ✓ **updateProduct** : Récupère un produit existant avec `getProductById(id)` et met à jour ses propriétés (`name`, `description`, `price`) avec les nouvelles valeurs fournies. Si le produit est trouvé, il est sauvegardé à l'aide de `productRepository.save(product)`. Sinon, `null` est retourné.
- ✓ **deleteProduct** : Utilise `getProductById(id)` pour récupérer un produit et, s'il est trouvé, le supprime de la base de données avec `productRepository.deleteById(id)`.

Package Repository

Le **package repository** est essentiel dans une application Spring Boot car il fournit une abstraction de l'accès aux données, permettant de manipuler les entités sans avoir à écrire de SQL ou de requêtes complexes.

Structure du package repository

1. Interfaces de Repository :

- Ces interfaces sont responsables de l'accès aux données pour des entités spécifiques. Elles étendent généralement des interfaces comme `JpaRepository`.
- Ces interfaces fournissent des méthodes par défaut pour effectuer des opérations de base telles que la création, la lecture, la mise à jour et la suppression des entités.

2. Exemple de Repository (`ProductRepository`) :

```
package com.universiapolis.gestion_produits.repository;

import com.universiapolis.gestion_produits.entity.Product;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Component;
import java.util.List;

@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {

}
```

Explication du `ProductRepository` :

- **@Repository** :
 - Cette annotation marque l'interface comme un **repository Spring**, ce qui permet à Spring de la gérer comme un bean et d'assurer la gestion des exceptions liées à la persistance des données.
- **extends JpaRepository<Product, Long>** :
 - `JpaRepository` est une interface de Spring Data JPA qui fournit des méthodes pour effectuer des opérations de persistance de manière simple et rapide.
 - **Product** : L'entité sur laquelle ce repository opérera.
 - **Long** : Le type de la clé primaire de l'entité `Product`.
- **Méthodes héritées** :
 - **save(Product product)** : Enregistre un produit dans la base de données (ajout ou mise à jour).
 - **findById(Long id)** : Recherche un produit par son identifiant.
 - **findAll()** : Récupère tous les produits.
 - **deleteById(Long id)** : Supprime un produit par son identifiant.

Package Controller

Le **package controller** contient les classes qui gèrent les requêtes HTTP entrantes. Ces classes sont annotées avec `@RestController` ou `@Controller` pour définir des points d'entrée d'API REST.

La **classe ProductController** est un contrôleur Spring MVC qui gère les requêtes HTTP pour l'entité **Product**. Elle utilise le service métier **IProduit** pour effectuer les opérations de gestion des produits. Voici une explication détaillée de cette classe :

1. Annotations

```
@RestController
@RequestMapping("/universiapolis")
public class ProductController {
```

- **@RestController** : Cette annotation combine **@Controller** et **@ResponseBody**, ce qui signifie que cette classe est un contrôleur Spring et que les méthodes renverront directement des réponses au format JSON ou XML (au lieu de retourner une vue HTML).
- **@RequestMapping("/universiapolis")** : Définit un préfixe pour toutes les URL mappées dans ce contrôleur. Par exemple, toutes les méthodes de ce contrôleur commenceront par `/universiapolis`.

2. Injection du service métier (IProduit)

```
@Autowired
private IProduit ProductMetier;
```

- **@Autowired** : Cette annotation permet à Spring d'injecter automatiquement une instance du service métier **IProduit** dans le contrôleur. Le service métier est responsable de la logique métier et de l'accès aux données via le repository.

3. Méthodes de gestion des produits

- **@PostMapping("/products")** :

```
@PostMapping("/products")
public Product addProduct(@RequestBody Product product) {
    return this.ProductMetier.addProduct(product);
}
```

Cette méthode gère les requêtes HTTP POST pour ajouter un produit. Elle reçoit un objet **Product** dans le corps de la requête (`@RequestBody`) et appelle la méthode `addProduct()` du service métier pour ajouter ce produit à la base de données.

- **@GetMapping("/products") :**

```
@GetMapping("/products")
public List<Product> getAllProducts() {
    return this.ProductMetier.getAllProducts();
}
```

Cette méthode gère les requêtes HTTP GET pour récupérer tous les produits. Elle appelle la méthode `getAllProducts()` du service métier pour récupérer la liste des produits.

- **@GetMapping("/products/{id}") :**

```
@GetMapping("/products/{id}")
public Product getProductById(@PathVariable("id") Long id) {
    return this.ProductMetier.getProductById(id);
}
```

Cette méthode gère les requêtes HTTP GET pour récupérer un produit par son identifiant. L'identifiant du produit est passé dans l'URL via **@PathVariable**.

- **@PutMapping("/products/{id}") :**

```
@PutMapping("/products/{id}")
public Product updateProduct(@PathVariable("id") Long id, @RequestBody Product p) {
    return this.ProductMetier.updateProduct(id,p);
}
```

Cette méthode gère les requêtes HTTP PUT pour mettre à jour un produit existant. L'identifiant du produit est passé dans l'URL via **@PathVariable**, et les nouvelles données du produit sont passées dans le corps de la requête via **@RequestBody**.

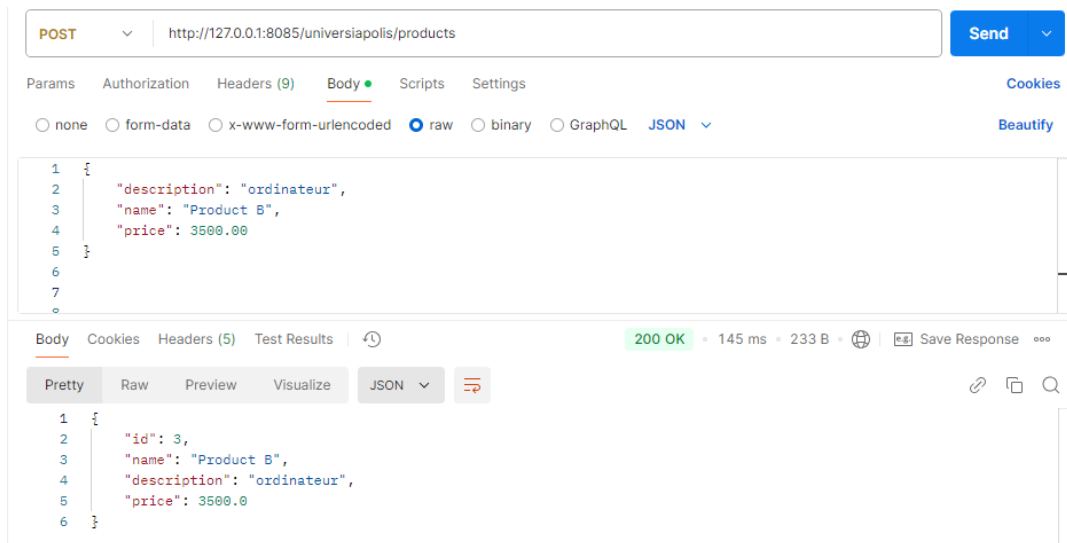
- **@DeleteMapping("/products/{id}") :**

```
@DeleteMapping("/products/{id}")
public void deleteProduct(@PathVariable("id") Long id) {
    this.ProductMetier.deleteProduct(id);
}
```

Cette méthode gère les requêtes HTTP DELETE pour supprimer un produit en fonction de son identifiant, passé dans l'URL via **@PathVariable**.

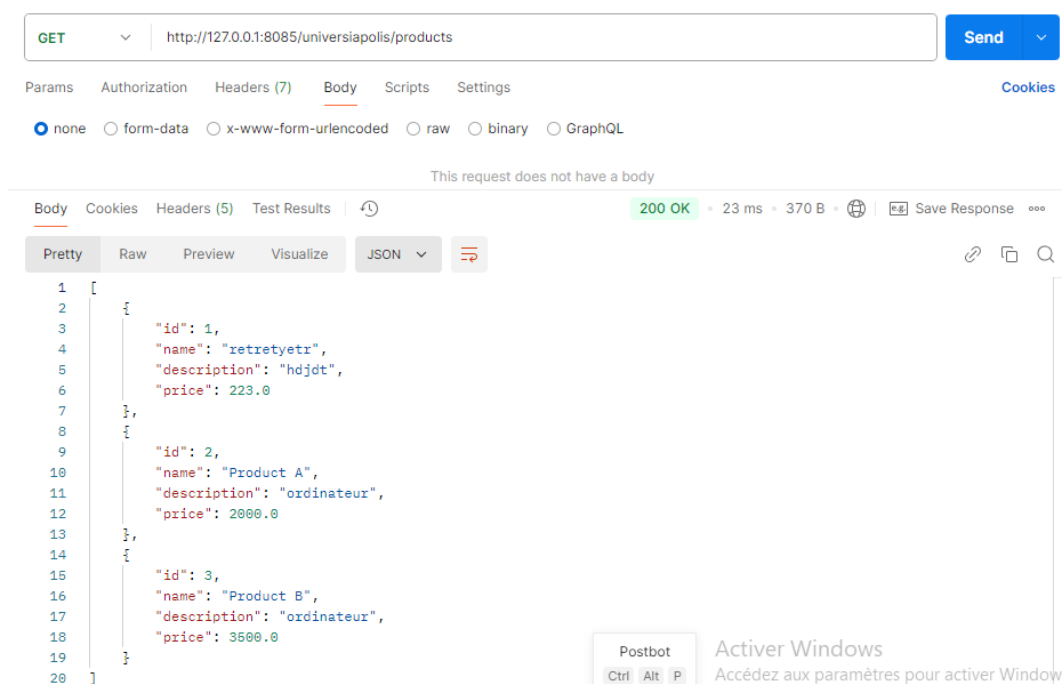
Test avec Postman

1. Ajouter un produit (POST)



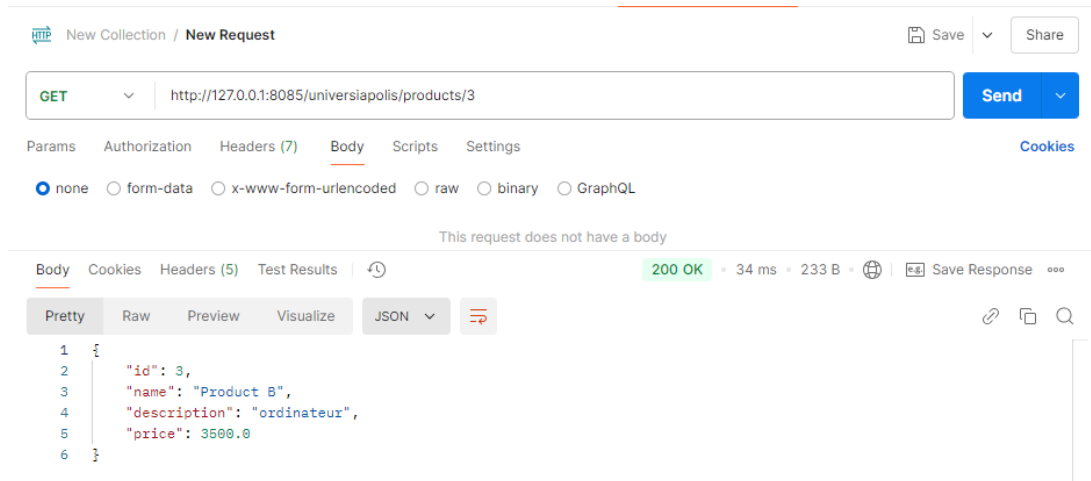
- **URL** : `http://localhost:8085/universiapolis/products`
- **Méthode HTTP** : POST
- **Description** : Ajoute un nouveau produit à la base de données.
- **Corps de la requête (Body)** : Utilisez l'option **raw** et sélectionnez **JSON** comme format, puis ajoutez un objet JSON représentant le produit à ajouter.
- **Réponse attendue** : Un objet JSON représentant le produit ajouté, avec un identifiant généré automatiquement.

2. Récupérer tous les produits (GET)



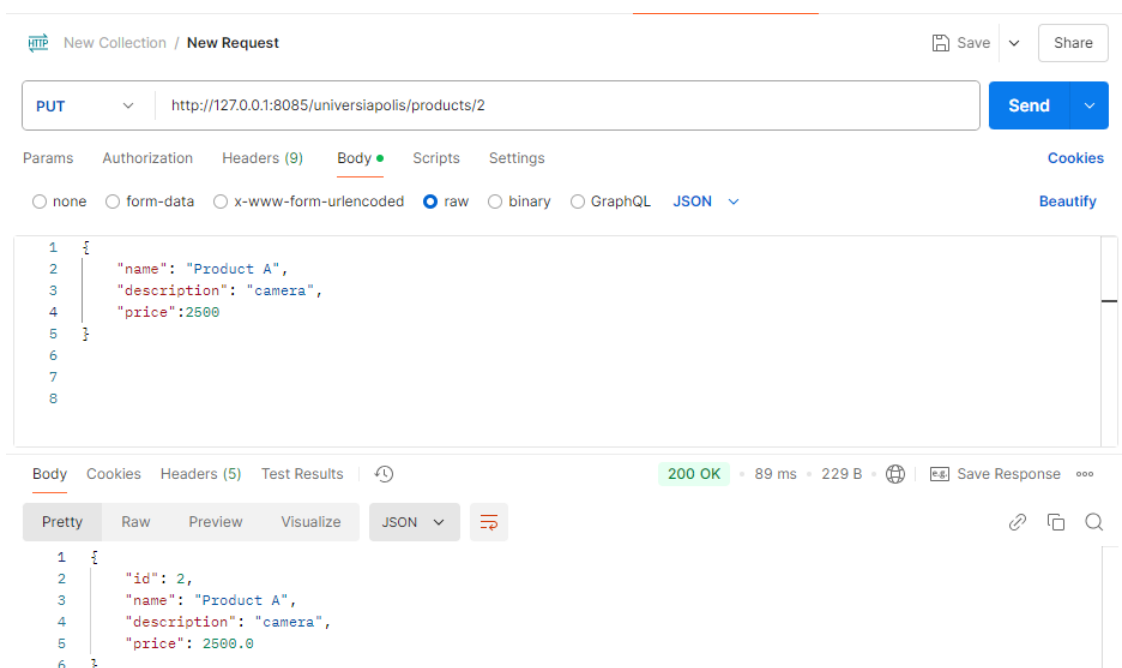
- **URL** : `http://localhost:8085/universiapolis/products`
- **Méthode HTTP** : GET
- **Description** : Récupère tous les produits stockés dans la base de données.
- **Corps de la requête** : Aucun corps requis.
- **Réponse attendue** : Une liste d'objets JSON représentant les produits.

3. Récupérer un produit par ID (GET)



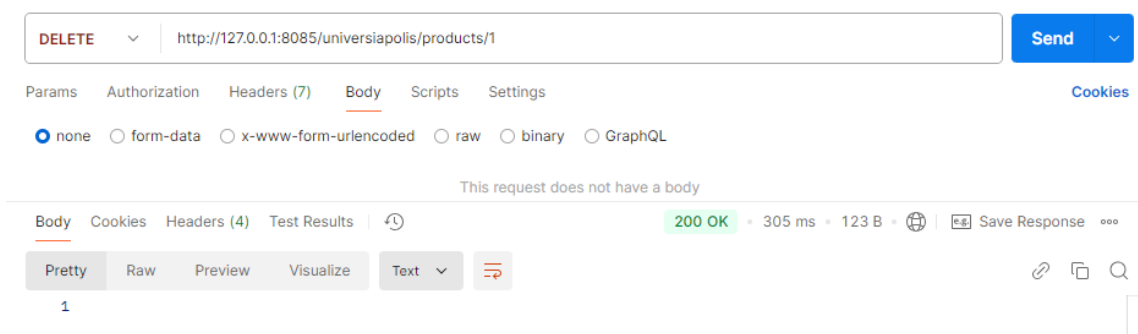
- **URL** : `http://localhost:8085/universiapolis/products/{id}`
- **Méthode HTTP** : GET
- **Description** : Récupère un produit spécifique en fonction de son identifiant.
- **Paramètre** : Remplacez `{id}` par l'identifiant du produit que vous souhaitez récupérer.
- **Exemple d'URL** : `http://localhost:8085/universiapolis/products/1`
- **Réponse attendue** : Un objet JSON représentant le produit demandé.

4. Mettre à jour un produit (PUT)



- **URL** : `http://localhost:8085/universiapolis/products/{id}`
- **Méthode HTTP** : PUT
- **Description** : Met à jour les informations d'un produit existant en fonction de son identifiant.
- **Paramètre** : Remplacez `{id}` par l'identifiant du produit à mettre à jour.
- **Corps de la requête (Body)** : Utilisez l'option **raw** et sélectionnez **JSON** comme format, puis ajoutez un objet JSON représentant les nouvelles données du produit.
- **Exemple d'URL** : `http://localhost:8085/universiapolis/products/1`
- **Réponse attendue** : Un objet JSON représentant le produit mis à jour.

5. Supprimer un produit (DELETE)



- **URL** : `http://localhost:8085/universiapolis/products/{id}`
- **Méthode HTTP** : DELETE
- **Description** : Supprime un produit de la base de données en fonction de son identifiant.
- **Paramètre** : Remplacez `{id}` par l'identifiant du produit à supprimer.
- **Exemple d'URL** : `http://localhost:8085/universiapolis/products/1`
- **Réponse attendue** : Aucune réponse spécifique, le produit est supprimé avec succès.

