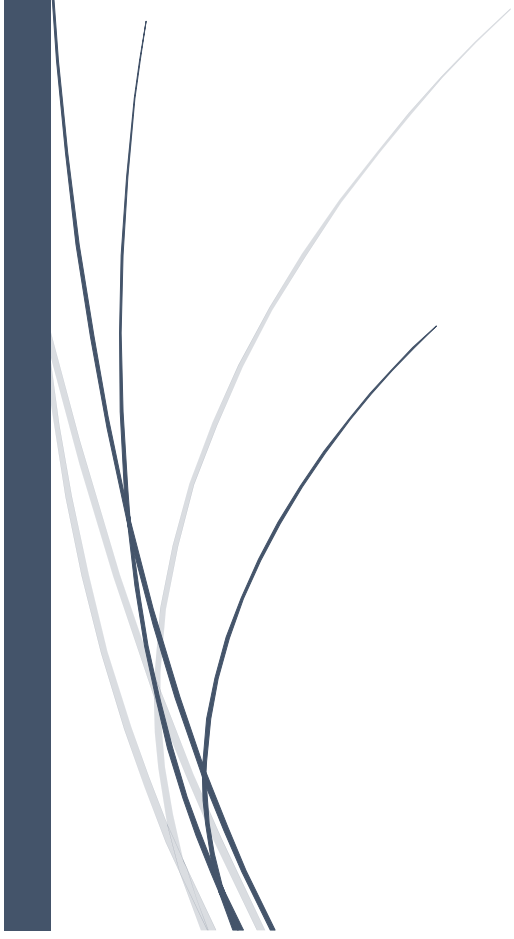
TP : Développer un service web Rest avec Spring Data Rest et Spring BOOT



Architecture des composants d'entreprise

Table des matières

I. Objectif du TP

- Utiliser le starter *Spring Data Rest* pour exposer une modèle via l'architecture Rest.
- Utiliser Spring Data JPA pour générer les méthodes CRUD.
- Utiliser la base de données H2.
- Configurer l'api *OpenAPIi version 3* pour rendre votre service web compréhensible à tout le monde.
- Utiliser l'interface de Swagger pour tester votre service web.

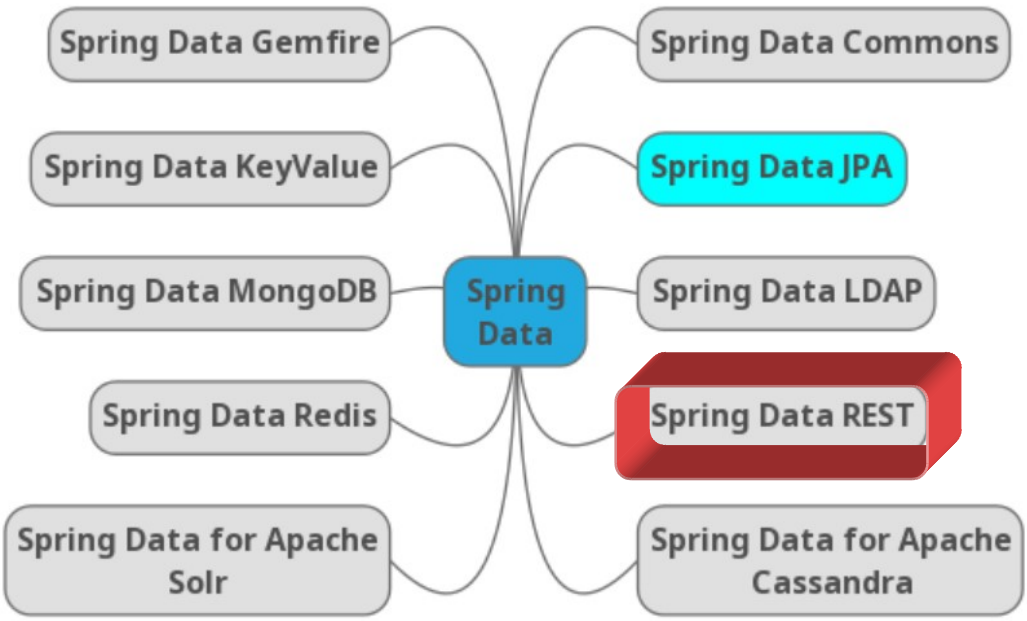
II. Prérequis

- IntelliJ IDEA ;
- JDK version 17 ;
- Une connexion Internet pour permettre à Maven de télécharger les librairies.
- Postman pour effectuer les tests.

NB : *Ce TP a été réalisé avec IntelliJ IDEA 2023.2.3 (Ultimate Edition).*

III. C'est quoi Spring Data Rest

- *Spring Data REST* fait partie du projet *Spring Data « Umbrella Project »* et facilite la création des services Web « hypermedia-driven REST ».



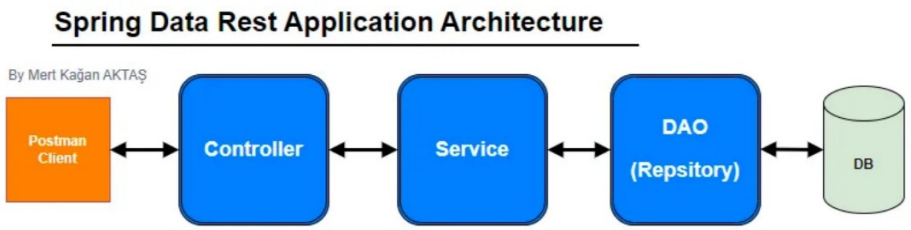
- Spring Data REST s'appuie sur les référentiels Spring Data, analyse le modèle de domaine de votre application et expose les ressources HTTP pour les agrégats contenus dans le modèle.
- Parmi les fonctionnalités fournies par Spring Data Rest :
    - Expose une API REST pour votre classe modèle en utilisant `HAL (Json Hypertext Application Language)` comme *media type*.

- Resource Discoverability (ou découverte des ressources) : Par défaut, Spring Data REST utilise HAL pour restituer les réponses. HAL définit les liens à contenir dans une propriété du document renvoyé.
- Expose les collections et les associations représentant votre modèle.
- Supporte la pagination via des liens de navigation.
- Supporte actuellement JPA, MongoDB, Neo4j et Cassandra.

- Le site officiel de *Spring Data Rest* est https://spring.io/projects/spring-data-rest.
  - Plusieurs exemples sont disponibles sur le site officiel : https://spring.io/projects/spring-data-rest#samples.

IV. Pourquoi Spring Data Rest ?

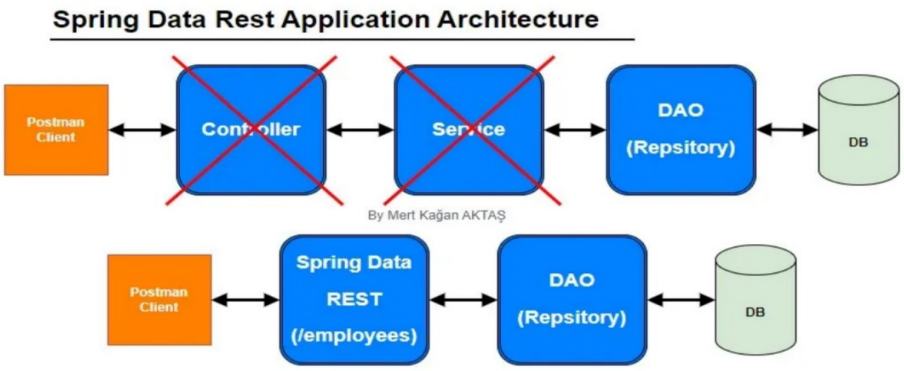
- Dans une application typique basée sur Spring, nous créons une architecture à trois couches comme ci-dessous :



- Quel est le problème ?
  Si nous voulons créer une API REST CRUD pour une entité, nous devons créer une classe de contrôleur et écrire manuellement les API REST (créer, mettre à jour, supprimer, obtenir, paginer, trier, etc.).
  Si nous voulons créer une API REST CRUD pour une autre entité : par exemple , Département , Projet , Société , Utilisateur , etc. Ensuite, nous devons créer un contrôleur pour toutes ces entités et créer des API REST.

  ➔ Constat : Nous répétons encore le même code….

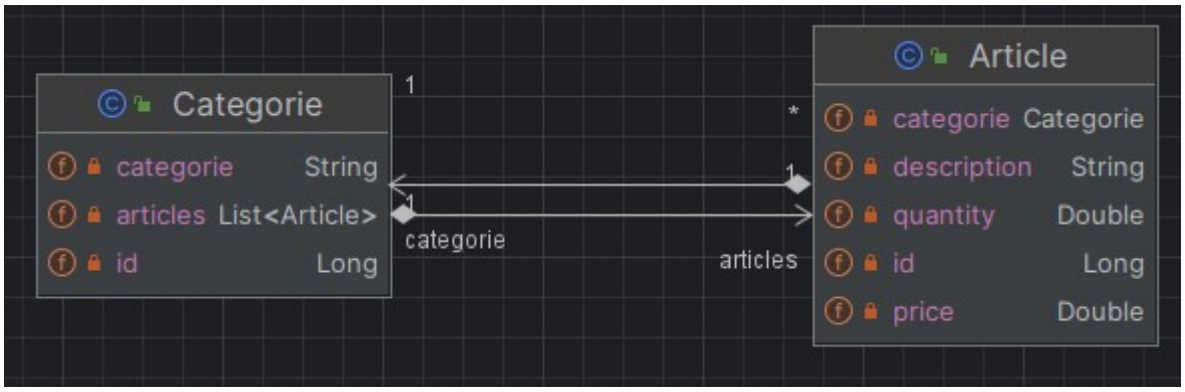
- Solution :
  Spring Data REST est la solution. il utilise des interfaces qui étendent JpaRepository et fournit des API REST CRUD pour les entités avec un minimum d'efforts. Cela permet de minimiser le code de la couche contrôleur standard.



V. Développement du WS avec Spring Data Rest

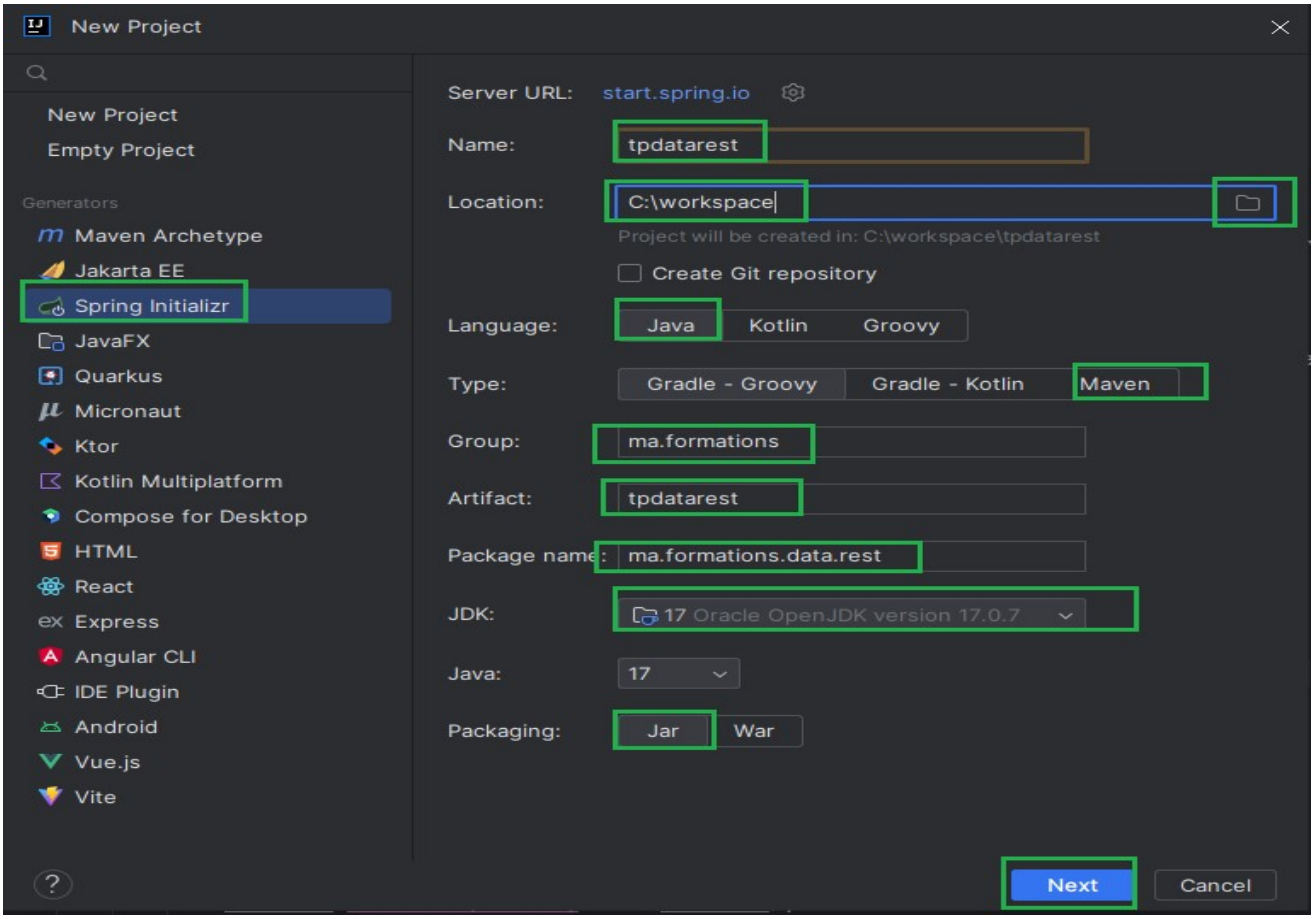
a. Objet de l'atelier

- Nous allons développer un service web qui permet d'effectuer des opérations CRUD sur les articles contenus dans une base de données H2. Le diagramme de classe est le suivant :

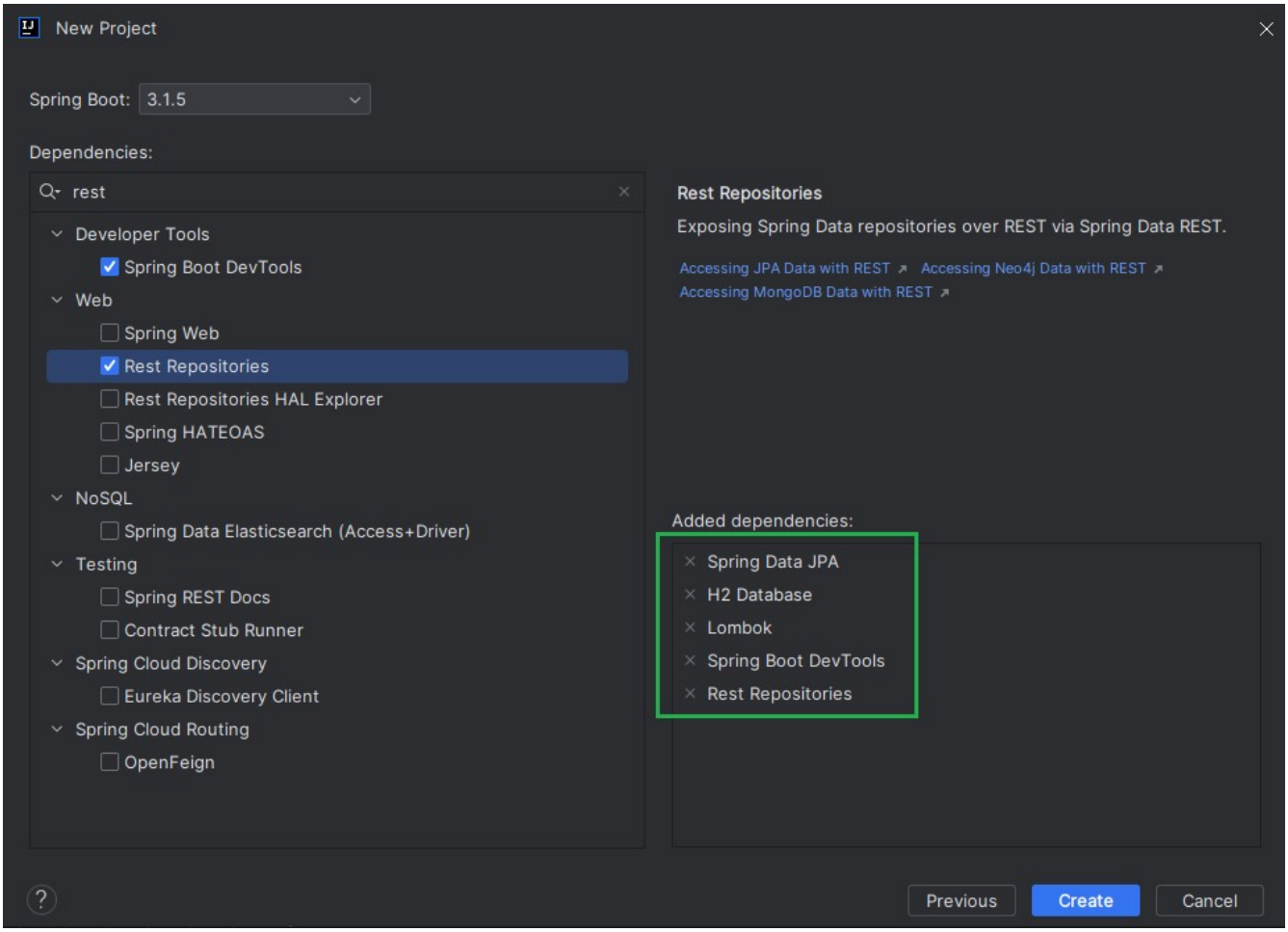


b. Création du projet Maven

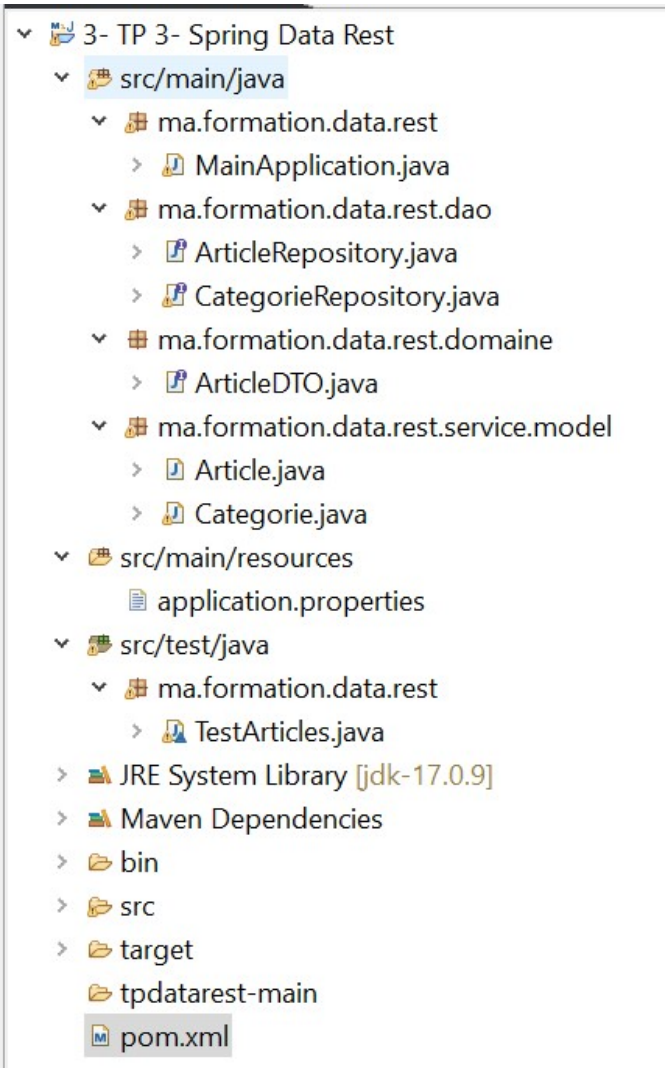
- Créer un nouveau projet comme le montre la fenêtre suivante :



1. Entrer le nom de votre projet.
2. Dans Location, préciser le dossier danslequel le projet sera crée (par exemple : c:\workspace).
3. Dans Language, cliquer sur Java.
4. Dans Type, cliquer sur Maven.
5. Entrer le Group et l'Artifact.
6. Entrer le nom du package.
7. Choisir JDK 17.
8. Dans Packaging, cliquer sur Jar. Enfin, cliquer sur Next. La fenêtre suivante s'affiche :



- Ajouter les dépendances suivantes : Spring Data JPA, H2 database, Lombok, Spring Boot DevTools et Rest Repositories et cliquer ensuite sur Create.
- Ou bien via Eclispe :



c. Le fichier pom.xml

- Vérifier que votre fichier pom.xml généré contient les dépendances suivantes :

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

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

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

```
  <scope>runtime</scope>
  <optional>true</optional>
</dependency>

<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>

<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <optional>true</optional>
</dependency>

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

Explications :

- La dépendance :

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

✓ Il s'agit du starter de *Spring Data Rest*. Ce dernier permet d'ajouter toutes les dépendances nécessaires et permet également d'auto-configurer le Framework *Spring Data Rest*.

- La dépendance :

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

✓ Il s'agit du starter de *Spring Data JPA*. Ce dernier permet de d'ajouter toutes les dépendances nécessaires et permet également d'auto-configurer l'api *JPA*. Le starter *spring-boot-starter-data-jpa* utilise comme implémentation par défaut de l'api *JPA*, le Framework *Hibernate*. Pour rappel, l'api JPA est implémentée par 03 Framework : Hibernate, Toplink et EclipseLink.

- La dépendance :

```
<dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
   <scope>runtime</scope>
</dependency>
```

✓ Il s'agit de la base de données virtuelle H2. H2 est une base de données embarquée qui se démarre et s'arrête avec l'application.

d. Le fichier application.properties

- Copier les lignes suivantes au niveau du fichier *application.properties* :

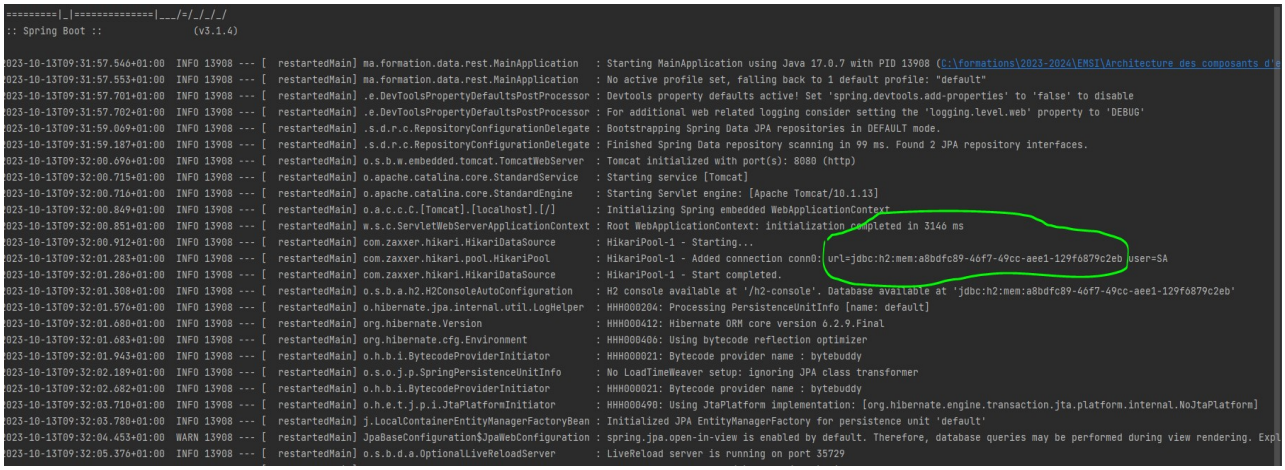
```
# The name of the H2 database :
spring.datasource.url=jdbc:h2:mem:testdb
# The H2 Driver :
spring.datasource.driverClassName=org.h2.Driver
spring.data.jpa.repositories.bootstrap-mode=default
spring.datasource.username=sa
spring.datasource.password=
# The Dialect : java => SQL compatible with H2
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
# automatic creation and modification of tables
spring.jpa.hibernate.ddl-auto=update
# Activate the H2 console :
spring.h2.console.enabled=true
# For customizing the console URL
spring.h2.console.path=/h2
```

Explications :

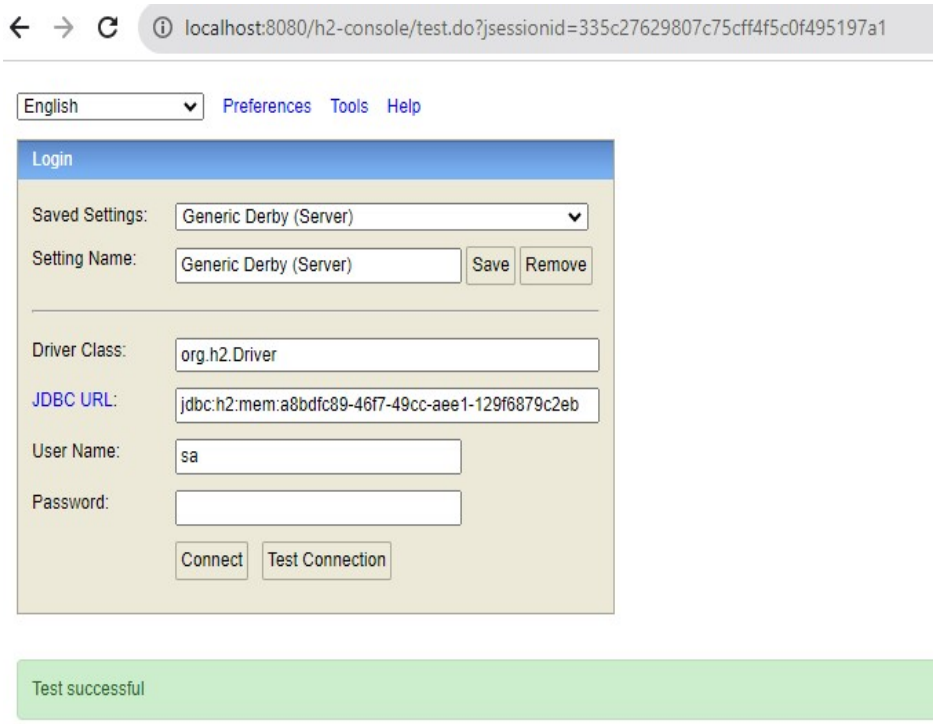
✓ Au niveau de ce fichier, nous avons configuré la base de données H2 : le driver, l'URL, le compte utilisateur, le dialecte permettant la transformation du code java vers SQL et l'activation de la création et de la modification automatique des tables.

✓ Vous pouvez ne rien préciser au niveau de ce fichier. En effet, par défaut *Spring Boot* configurera automatiquement la base de données H2. Dans ce cas :

➢ Le nom de la base de données sera généré aléatoirement par *Spring Boot* comme illustré ci-après :



- Pour l'accès à la console de H2, Spring Boot utilisera le lien : */h2-console* comme illustré ci-après :



e. Les classes modèles « Article » et « Categorie »

- La classe **Article** :

```
package ma.formation.data.rest.service.model;

import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;
```

```java
@Entity
@Data
@NoArgsConstructor
@Builder
@AllArgsConstructor
public class Article {
    @Id
    @GeneratedValue
    private Long id;
    private String description;
    private Double price;
    private Double quantity;

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

- La classe **Categorie** :

```java
package ma.formation.data.rest.service.model;

import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.ArrayList;
import java.util.List;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Categorie {
    @Id
    @GeneratedValue
    private Long id;
    @Column(unique = true)
    private String categorie;

    @OneToMany(mappedBy = "categorie", cascade = CascadeType.ALL)
    private List<Article> articles = new ArrayList<>();
```

```
    public void addArticle(Article article) {
        article.setCategorie(this);
        articles.add(article);
    }
}
```

f. L'interface ArticleDTO

```
package ma.formation.data.rest.domaine;

import ma.formation.data.rest.service.model.Article;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.data.rest.core.config.Projection;

@Projection(name = "articleDTO", types = Article.class)
public interface ArticleDTO {
    Long getId();

    @Value("#{target.description}")
    String getDesc();

    Double getPrice();

    @Value("#{target.quantity}")
    Double getQuant();

    @Value("#{target.categorie.categorie}")
    String getCat();

}
```

Explications :

- ✓ L'annotation *@Projection* de *Spring Data Rest* permet de préciser les projections des données à transférer via Rest. Dans cet exemple, nous allons transmettre : l'identifiant de l'article (champ : *id*), la description (champ : *desc*), le prix (champ *price*), la quantité (champ : quant) et la catégorie (champs : cat). Par défaut, Spring Data Rest n'envoie pas l'id dans la réponse Rest.

- ✓ Par défaut, *Spring Data Rest* injectera la valeur du champ ayant le même nom au niveau de la classe Article (classe précisée dans l'attribut *types* de l'annotation *@Projection*).

- ✓ Si aucun champ ayant le même nom ne se trouve dans la classe *Article*, il faut annoter ce dernier par *@Value* en précisant le nom du champ correspondant au niveau de la classe *Article* en utilisant le mot clé: *target*

g. Les interfaces DAO : ArticleRepository et CategorieRepository

- L'interface **ArticleRepository** :

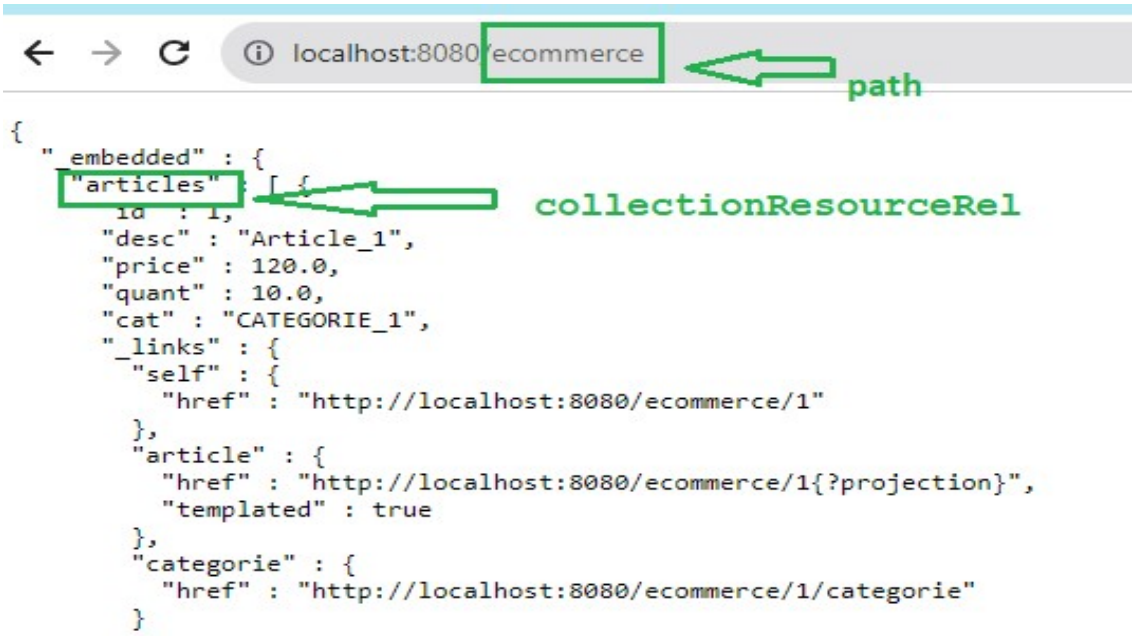
```
package ma.formation.data.rest.dao;

import ma.formation.data.rest.domaine.ArticleDTO;
import ma.formation.data.rest.service.model.Article;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;
import org.springframework.data.rest.core.annotation.RestResource;

import java.util.List;

@RepositoryRestResource(collectionResourceRel = "articles", path = "ecommerce",
excerptProjection = ArticleDTO.class)
public interface ArticleRepository extends JpaRepository<Article, Long> {
   @RestResource(path = "byCategorie", rel = "customFindByDescription")
   List<Article> findByCategorie_Categorie(@Param("categorie") String description);
}
```

Explications :

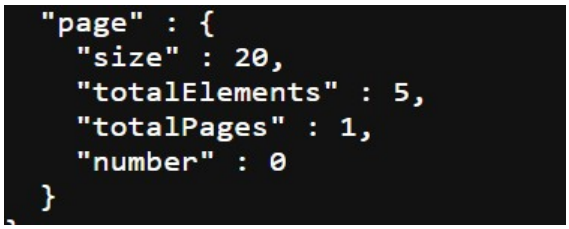
- ✓ L'annotation *@RepositoryRestResource* permet d'exposer le Repository *ArticleRepository* sous forme d'Api Rest.

- ✓ L'attribut *collectionResourceRel* sert à configurer le nom du champ qui représentera la collection des articles dans le message JSON ou XML.

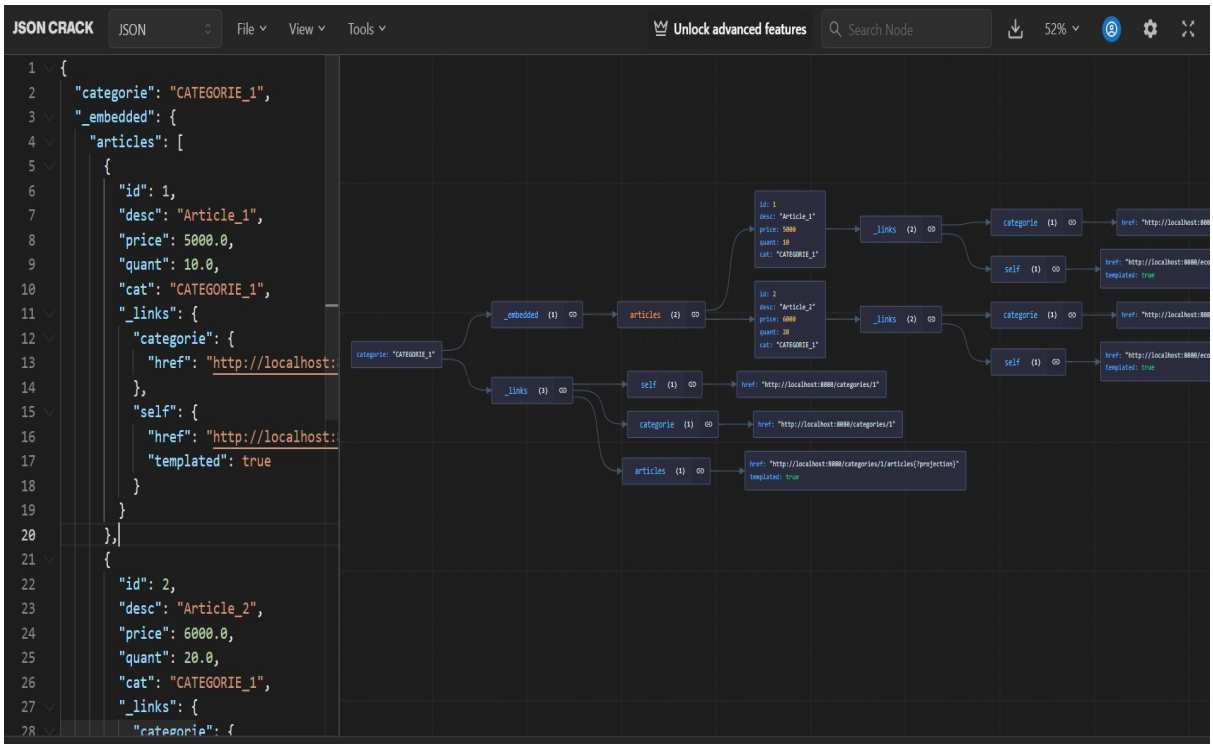
- ✓ L'attribut *path* sert à configurer le nom de l'Endpoint dans l'URL. Voir ci-dessous pour plus de précisons :



✓ Remarquer la pagination en fin de page



✓ Vous pouvez utiliser le lien https://jsoncrack.com/editor pour une meilleure visualisation de l'arborescence json :



✓ L'attribut *excerptProjection ou (l'extrait de la projection)* permet de préciser les projections que l'api peut exposer aux clients. Dans l'exemple ci-dessus, c'est les données de l'interface *ArticleDTO* qui seront transmises aux clients :

➢ Cette projection est utilisée par défaut dans la consultation de l'ensemble des articles. Par ailleurs, cette même projection n'est pas utilisée si vous consulter un article par son id :

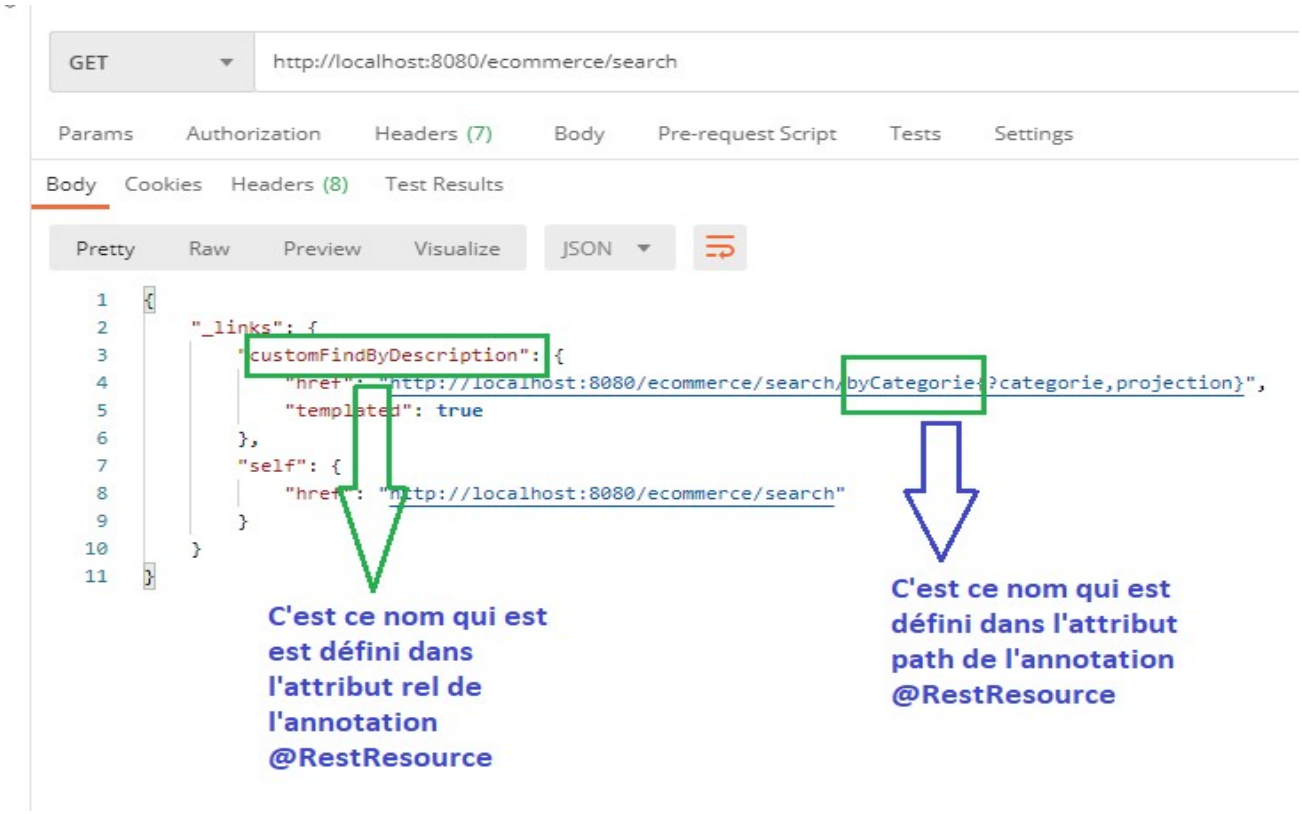
http://localhost:8080/ecommerce/1

➢ Pour utiliser la projection *ArticleDTO*, il faut utiliser le lien http://localhost:8080/ecommerce/1?projection=articleDTO :

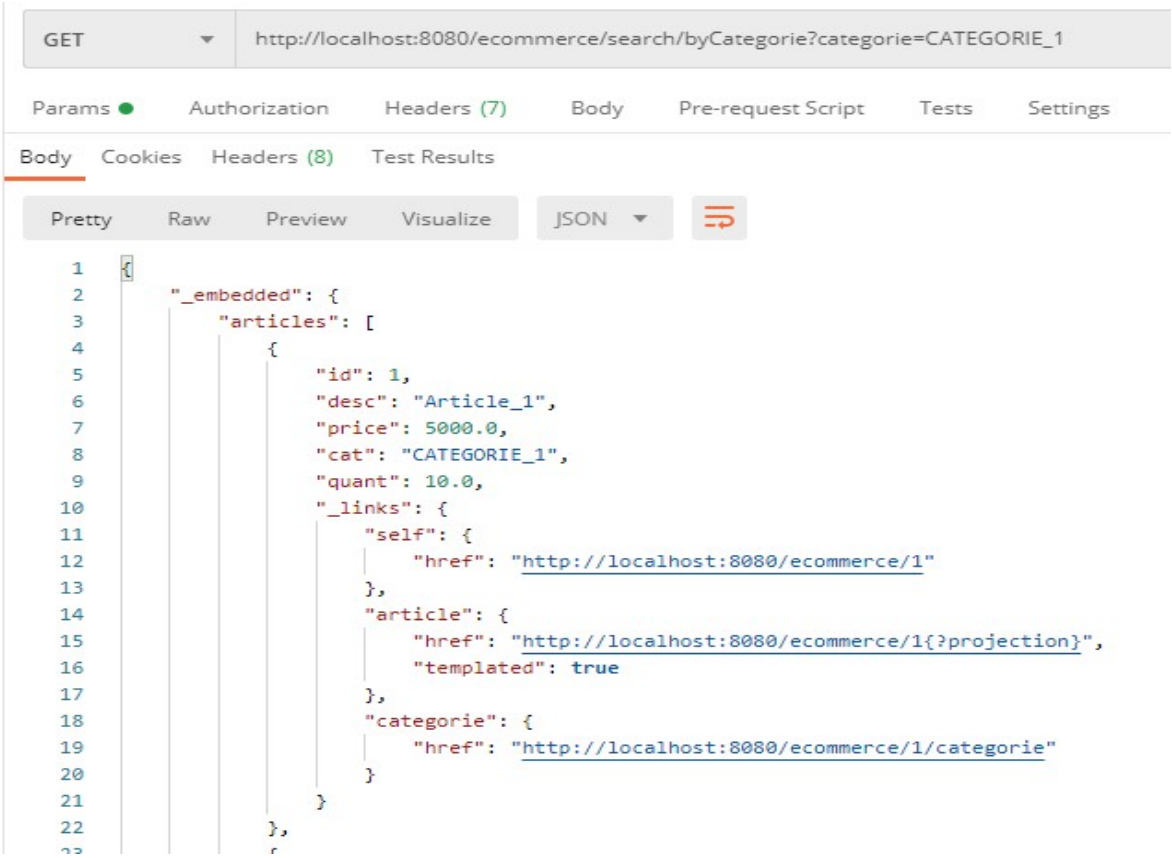
```
{
  "id" : 1,
  "desc" : "Article_1",
  "price" : 120.0,
  "cat" : "CATEGORIE_1",
  "quant" : 10.0,
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/ecommerce/1"
    },
    "article" : {
      "href" : "http://localhost:8080/ecommerce/1{?projection}",
      "templated" : true
    },
    "categorie" : {
      "href" : "http://localhost:8080/ecommerce/1/categorie"
    }
  }
}
```


✓ Les services définis dans le Repository sont accessibles via le lien : http://localhost:8080/ecommerce/search.

✓ Pour chaque service défini dans l'interface Repository, l'annotation *@RestResource* permet de personnaliser le nom du champ qui sera produit au niveau du message JSON/XML ainsi que le nom de l'URL pour accéder au service en question. Voir ci-dessous pour plus de précisions :



✓ Le lien http://localhost:8080/ecommerce/search/byCategorie?categorie=CATEGORIE_1 permet d'exécuter le service *findByCategorie_Categorie* :



- L'interface **CategorieRepository** :

```
package ma.formation.data.rest.dao;

import ma.formation.data.rest.service.model.Categorie;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

@RepositoryRestResource(collectionResourceRel = "categories", path = "categories")
public interface CategorieRepository extends JpaRepository<Categorie, Long> {
}
```

h. La classe MainApplication

- Au niveau de votre classe de démarrage de Spring BOOT, ajouter la méthode suivante :

```
@Bean
public CommandLineRunner initDatabase(CategorieRepository categorieRepository,
                                      ArticleRepository articleRepository) {
  return (a) -> {
    var categorie1 = Categorie.builder().categorie("CATEGORIE_1").build();
    var categorie2 = Categorie.builder().categorie("CATEGORIE_2").build();
    var categorie3 = Categorie.builder().categorie("CATEGORIE_3").build();
```

```
    var article1 = Article.builder().description("Article_1").price(5000.0).quantity(10.0).build();
    var article2 = Article.builder().description("Article_2").price(6000.0).quantity(20.0).build();
    var article3 = Article.builder().description("Article_3").price(7000.0).quantity(30.0).build();
    var article4 = Article.builder().description("Article_4").price(8000.0).quantity(40.0).build();
    var article5 = Article.builder().description("Article_5").price(9000.0).quantity(50.0).build();



    categorie1 = categorieRepository.save(categorie1);
    categorie2 = categorieRepository.save(categorie2);
    categorie3 = categorieRepository.save(categorie3);

    article1=articleRepository.save(article1);
    article2=articleRepository.save(article2);
    article3=articleRepository.save(article3);
    article4=articleRepository.save(article4);
    article5=articleRepository.save(article5);

    categorie1.setArticles(new ArrayList<>());
    categorie1.addArticle(article1);
    categorie1.addArticle(article2);

    categorie2.setArticles(new ArrayList<>());
    categorie2.addArticle(article3);
    categorie2.addArticle(article4);

    categorie3.setArticles(new ArrayList<>());
    categorie3.addArticle(article5);

    articleRepository.save(article1);
    articleRepository.save(article2);
    articleRepository.save(article3);
    articleRepository.save(article4);
    articleRepository.save(article5);

  };
}
```

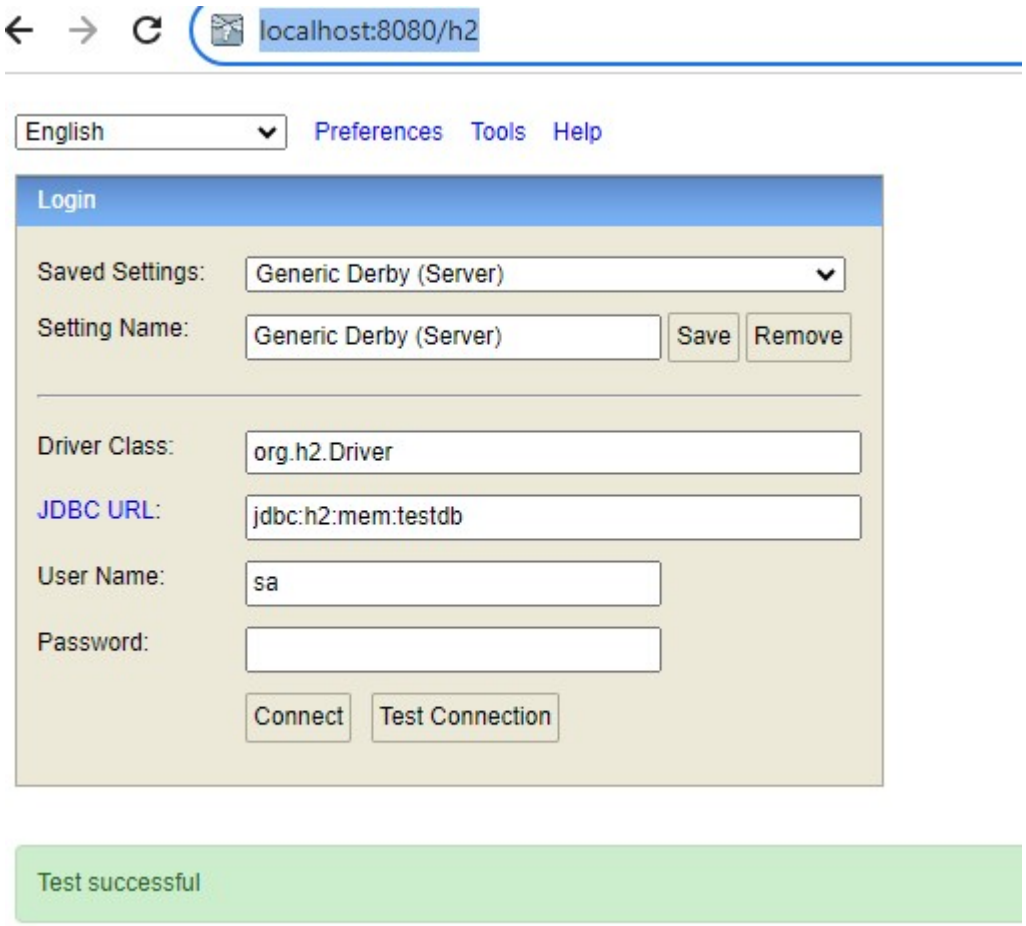
Explications :

- ✓ La méthode *initDatabase* est annotée par *@Bean* et retourne une instance de type *CommandLineRunner :* sera exécutée après le démarrage de l'application.

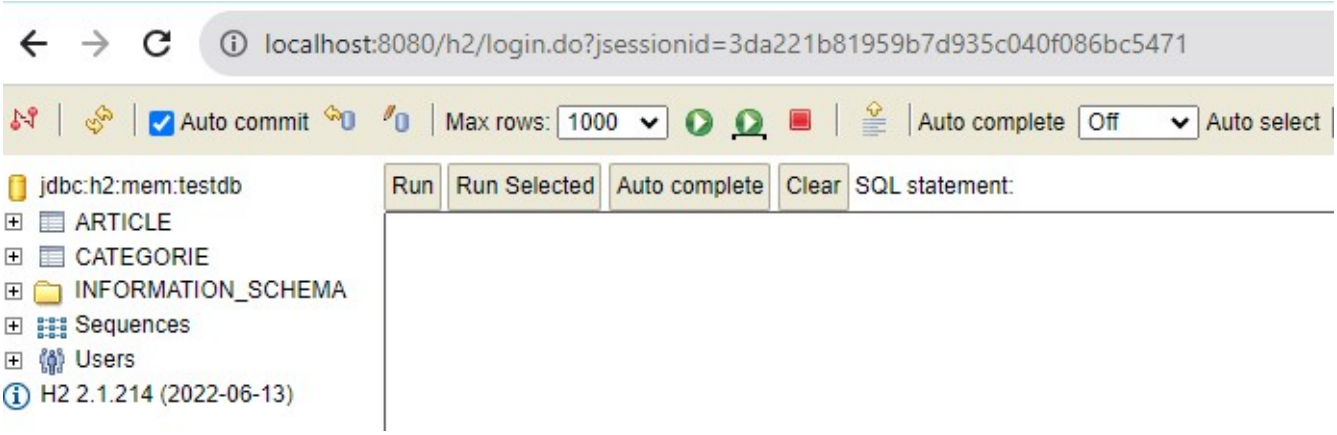
- ✓ Spring injectera les deux objets *articleRepository* et *categorieRepository* étant donné que ces derniers ont été passés en paramètre de la méthode initDatabase().

VI. Premier test

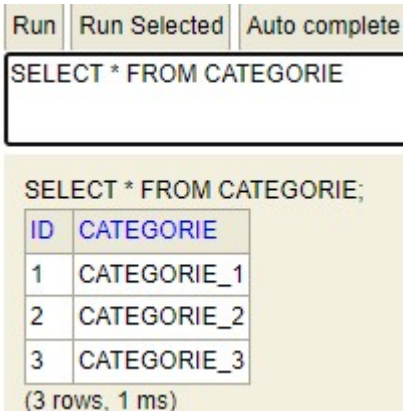
- Démarrer la méthode main de votre classe de démarrage de Spring Boot.
- Accéder à la console via le lien http://localhost:8080/h2 :

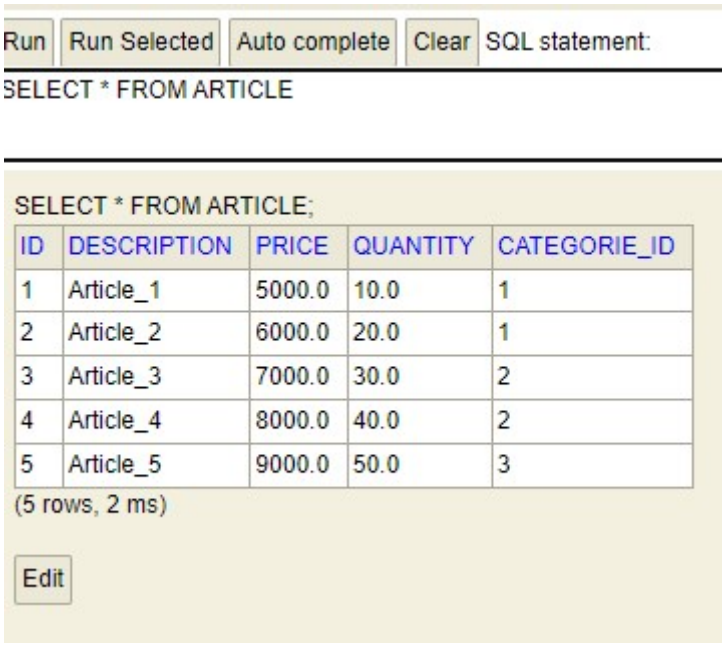


- Cliquer sur le bouton Connect pour accéder à la base de données :



- Vérifier que les deux tables Article et Categorie ont été bien crées et initialisées :



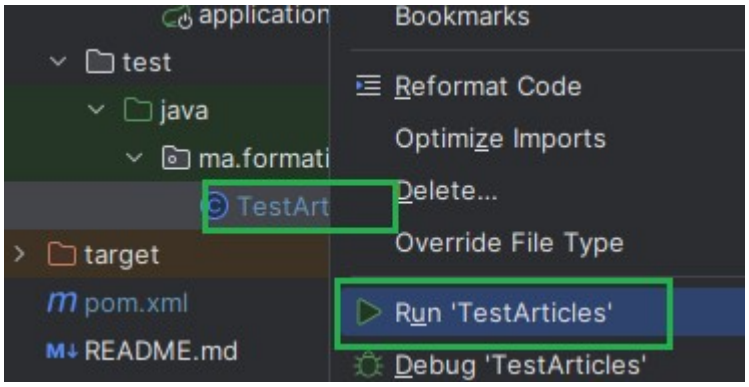


- Dans test/java, créer la classe **TestArticles** suivante pour effectuer les tests unitaires des deux méthodes GET (findAll et findById) du Repository *ArticleRepository* :

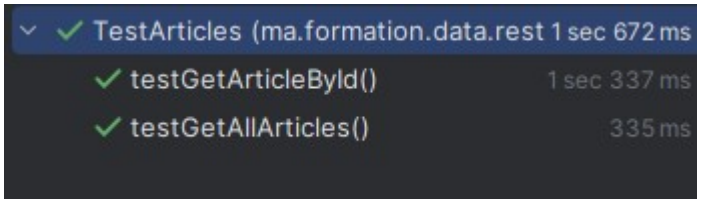
```
package ma.formation.data.rest;

import ma.formation.data.rest.service.model.Article;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;


@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class TestArticles {

   @Autowired
   WebApplicationContext context;

   private MockMvc mvc;

   @BeforeEach
   void setUp() {
      this.mvc = MockMvcBuilders.webAppContextSetup(context).build();
   }
```

```
    @Test
    void testGetAllArticles() throws Exception {


mvc.perform(get("/ecommerce").contentType(MediaType.APPLICATION_JSON).accept(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$._embedded.articles[0].desc").value("Article_1"))
            .andExpect(jsonPath("$._embedded.articles[0].price").value(5000))
            .andExpect(jsonPath("$._embedded.articles[0].quant").value(10));
    }

    @Test
    void testGetArticleById() throws Exception {


mvc.perform(get("/ecommerce/1").contentType(MediaType.APPLICATION_JSON).accept(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.description").value("Article_1"))
            .andExpect(jsonPath("$.price").value(5000))
            .andExpect(jsonPath("$.quantity").value(10));
    }

}
```

- Lancer le test comme montré ci-après :



- Vérifier que le résultat est concluant :



VII. Tests avec Postman

i. Tester la méthode GET pour toute la liste

- Le lien est http://localhost:8080/ecommerce :

```
{
    "_embedded": {
        "articles": [
            {
                "id": 1,
                "desc": "Article_1",
                "price": 5000.0,
                "cat": "CATEGORIE_1",
                "quant": 10.0,
                "_links": {
                    "self": {
                        "href": "http://localhost:8080/ecommerce/1"
                    },
                    "article": {
                        "href": "http://localhost:8080/ecommerce/1{?projection}",
                        "templated": true
                    },
                    "categorie": {
                        "href": "http://localhost:8080/ecommerce/1/categorie"
                    }
                }
            },
            {
                "id": 2,
                "desc": "Article_2",
                "price": 6000.0,
                "cat": "CATEGORIE_1",
```


Remarquer que la projection ArticleDTO qui est utilisée

j. Tester la méthode GET pour un article donnée et une catégorie donnée

- Le lien est http://localhost:8080/ecommerce/1 :


```
{
    "description": "Article_1",
    "price": 5000.0,
    "quantity": 10.0,
    "_links": {
        "self": {
            "href": "http://localhost:8080/ecommerce/1"
        },
        "article": {
            "href": "http://localhost:8080/ecommerce/1{?projection}",
            "templated": true
        },
        "categorie": {
            "href": "http://localhost:8080/ecommerce/1/categorie"
        }
    }
}
```

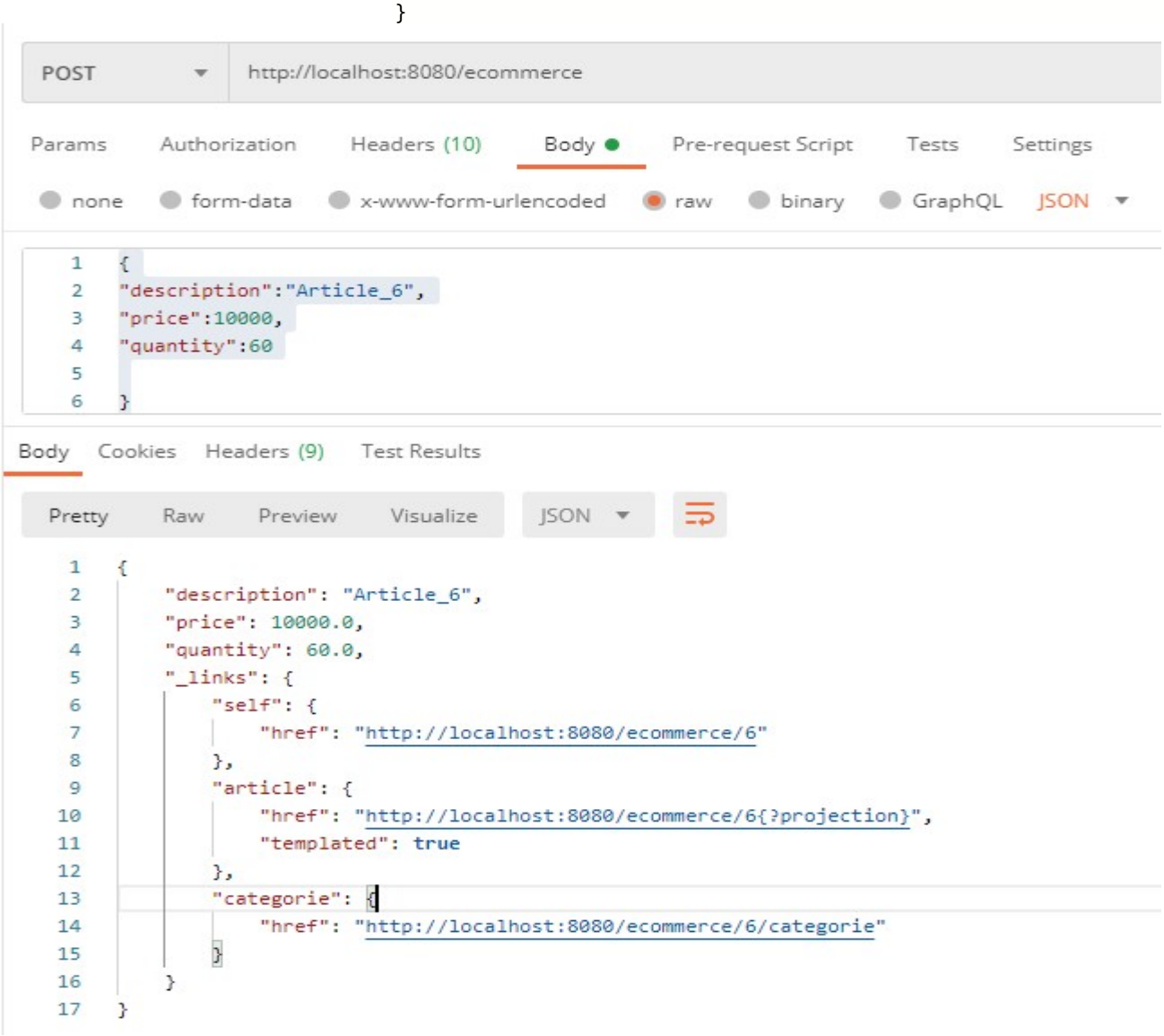

Remarquer que lorsque vous consultez un article par son ID, la projection ArticleDTO n'est pas utilisée.

k. Tester la méthode POST pour un article donnée et une catégorie donnée

Créer un nouvel article :

- Pour créer un nouveau article, voici le lien : http://localhost:8080/ecommerce.
- Dans le *header* ajouter le paramètre *Accept* avec comme valeur application/json.
- Dans le *body*, préciser le message JSON suivant :

```
{
"description":"Article_6",
"price":10000,
"quantity":60
}
```

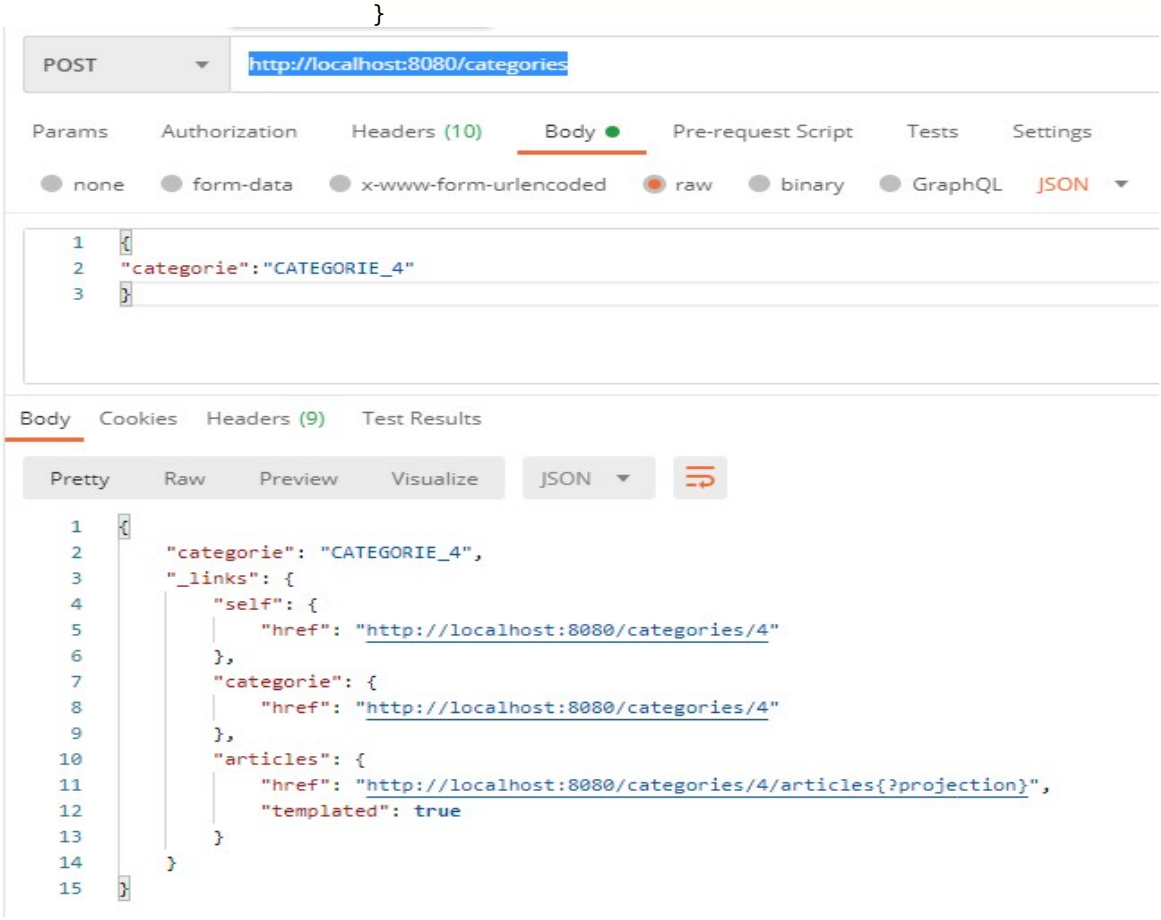


Créer une nouvelle catégorie :

- Pour créer une nouvelle catégorie, voici le lien : http://localhost:8080/categories.
- Dans le header ajouter le paramètre *Accept* avec comme valeur application/json.
- Dans le *body*, préciser le message JSON suivant :

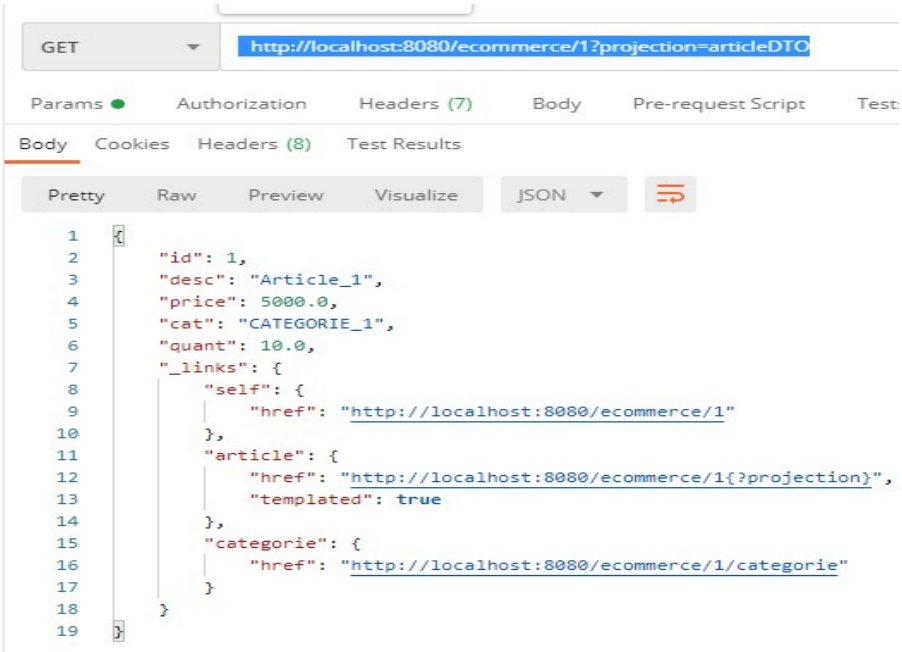
```
{
"categorie":"CATEGORIE_4"
```

}



I. Tester la méthode GET pour consulter un article par son ID en utilisant la projection

- Le lien est http://localhost:8080/ecommerce/1?projection=articleDTO :



m. Tester le service byCategorie

- Le lien est http://localhost:8080/ecommerce/search/byCategorie?categorie=CATEGORIE_1 :


```
{
    "_embedded": {
        "articles": [
            {
                "id": 1,
                "desc": "Article_1",
                "price": 5000.0,
                "cat": "CATEGORIE_1",
                "quant": 10.0,
                "_links": {
                    "self": {
                        "href": "http://localhost:8080/ecommerce/1"
                    },
                    "article": {
                        "href": "http://localhost:8080/ecommerce/1{?projection}",
                        "templated": true
                    },
                    "categorie": {
                        "href": "http://localhost:8080/ecommerce/1/categorie"
                    }
                }
            },
            {
                "id": 2,
```

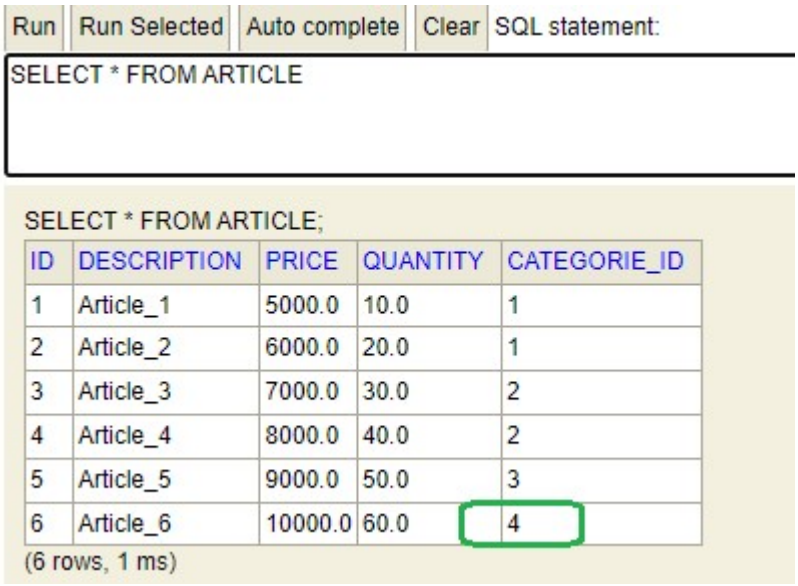

n. Tester la méthode PUT pour associer une catégorie à un article

Pour associer la catégorie CATEGORIE_4 (ID=4) à l'article Article_6 (ID=6), suivre les étapes suivantes :

- La méthode est PUT.
- Le lien est http://localhost:8080/ecommerce/6/categorie.
- Dans *Headers*, ajouter le paramétre ***Content-Type*** avec comme valeur ***text/uri-list*** :

- Dans *Body*, entrer le lien http://localhost:8080/categories/4

- Exécuter la requête et vérifier que le code Status est 204 et que la catégorie a été bien associée à l'article :



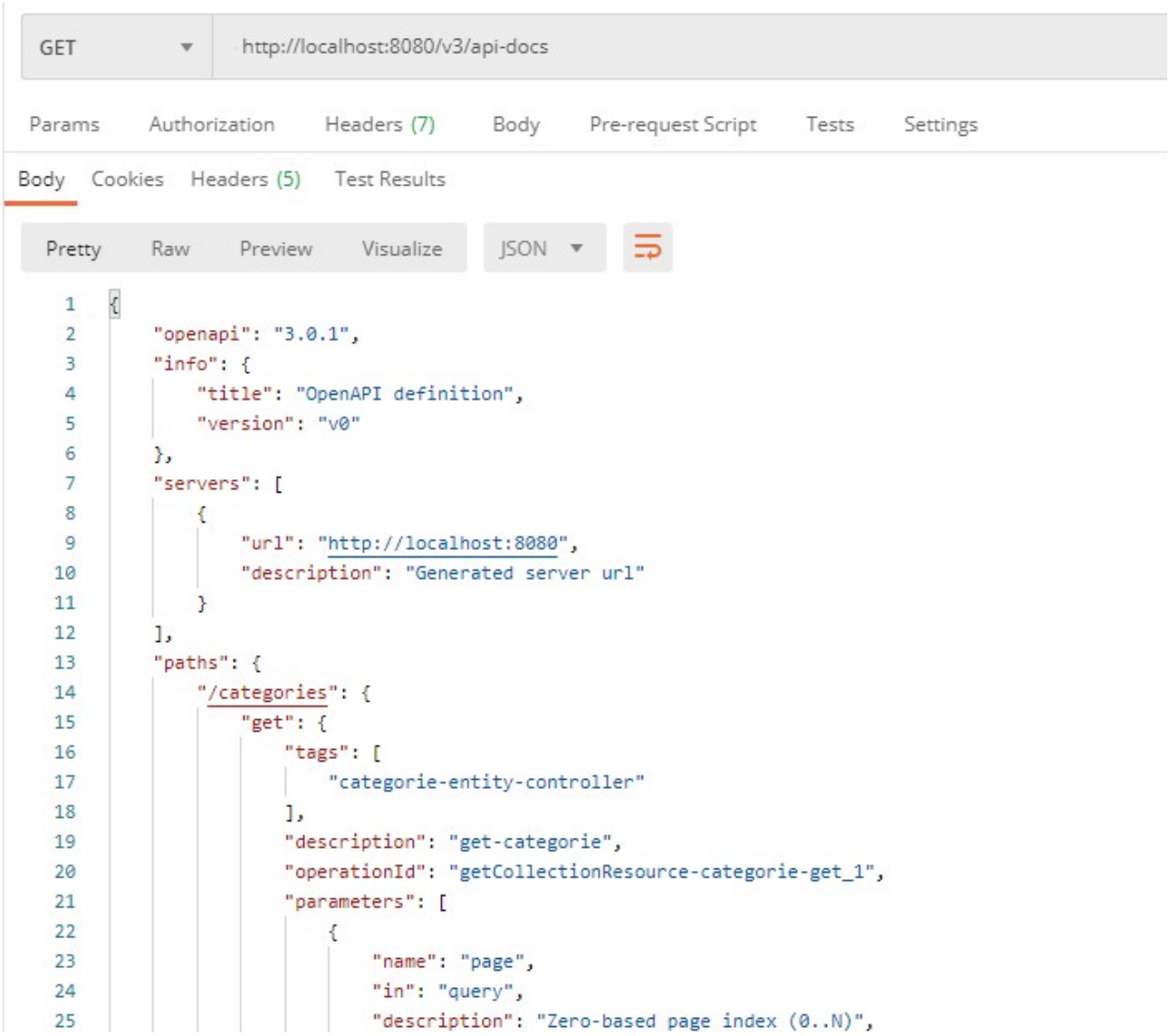
VIII. Configuration de l'api OpenAPI et Swagger

1. Configuration de l'api OpenAPI

- Ajouter la dépendance suivante au niveau du fichier pom.xml :

```
<dependency>
   <groupId>org.springdoc</groupId>
   <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
   <version>2.2.0</version>
</dependency>
```

- Démarrer votre application et vérifier que la documentation en formation JSON est accessible via le lien suivant : http://localhost:8080/v3/api-docs (n'oublier pas de changer votre port au cas où vous n'avez pas utilisé le port 8080) :



- Vous pouvez personnaliser le lien pour accéder à la documentation générée par OpenAPI en ajoutant la clé suivante au niveau du fichier *application.properties* :

```
springdoc.api-docs.path=/api-docs
```

Désormais, le lien de la documentation est http://localhost:8080/api-docs

- Le format de la documentation générée par l'api OpenAPI est par défaut JSON. Vous pouvez consulter la documentation en format YAML en accédant au lien suivant :

  http://localhost:8080/api-docs.yaml

2. Intégration avec Swagger UI

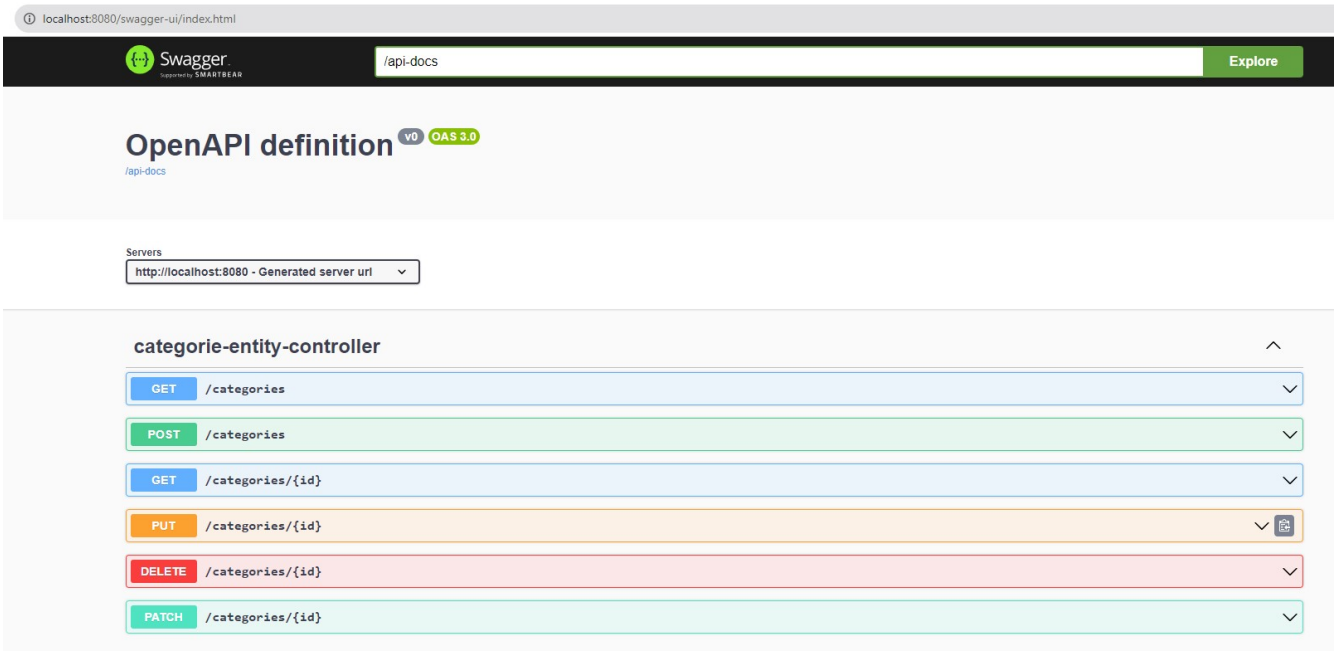
- En plus de générer la spécification OpenAPI 3, nous pouvons intégrer *springdoc-openapi* à *Swagger UI* pour interagir avec notre spécification API et tester les Endpoints.

- La dépendance *springdoc-openapi* inclut déjà *Swagger UI*, cette dernière est accessible via le lien suivant : http://localhost:8080/swagger-ui/index.html.

- Vous pouvez également personnaliser le lien pour accéder à *Swagger UI* en ajoutant la clé suivante au niveau du fichier *application.properties* :

```
springdoc.swagger-ui.path=/docs-ui
```

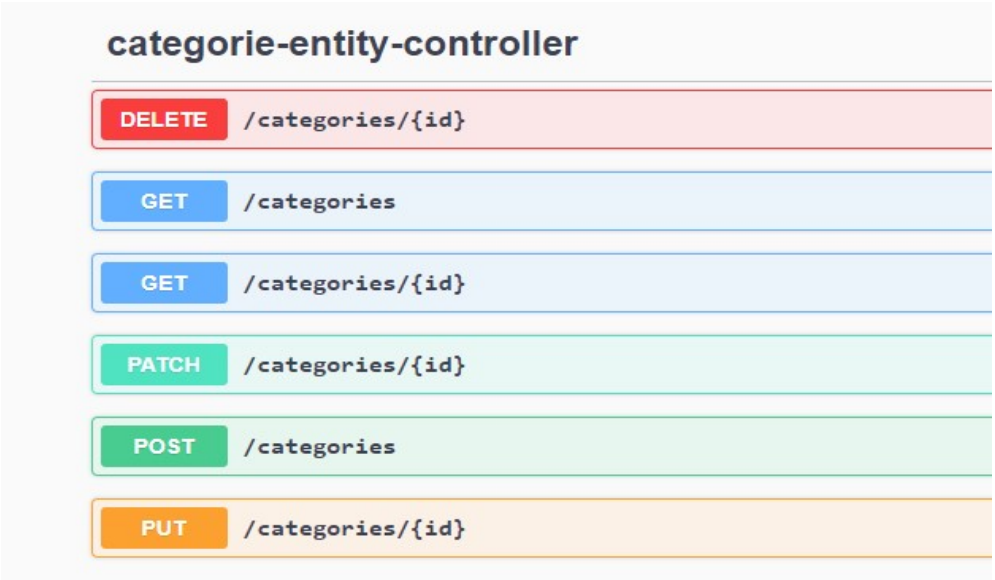
- Démarrer votre application et accéder au lien http://localhost:8080/docs-ui :



- Vous pouvez trier les méthodes au niveau de Swagger UI en ajoutant la clé suivante au niveau du fichier application.properties :

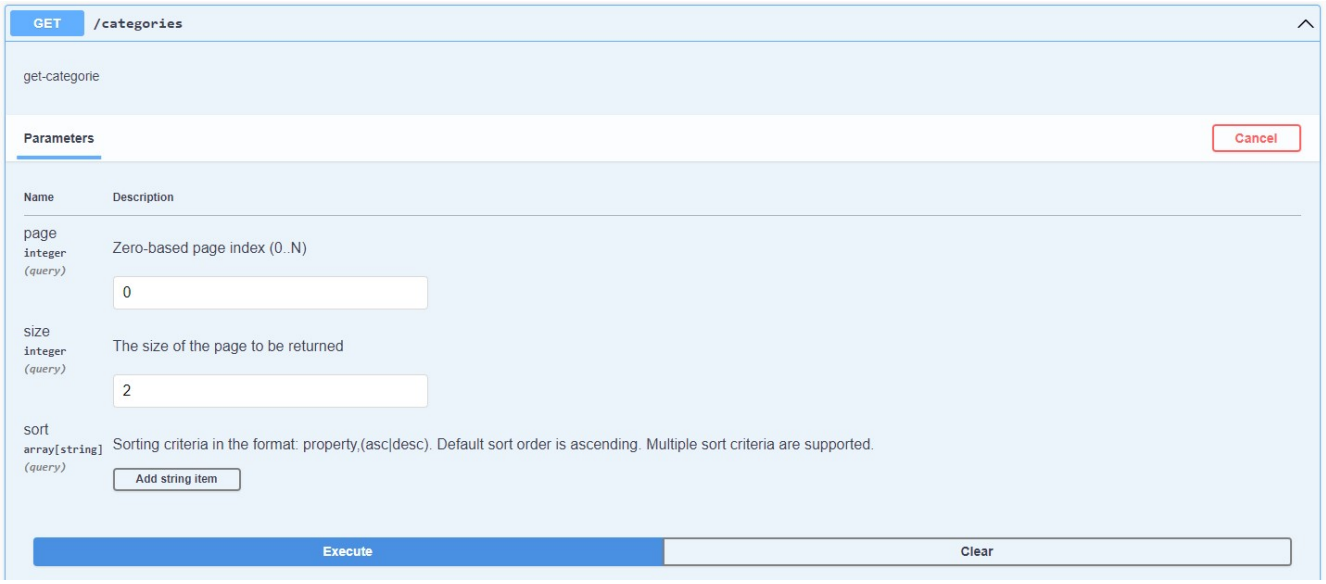
```
springdoc.swagger-ui.operationsSorter=method
```

- Démarrer votre application et accéder au lien http://localhost:8080/docs-ui :

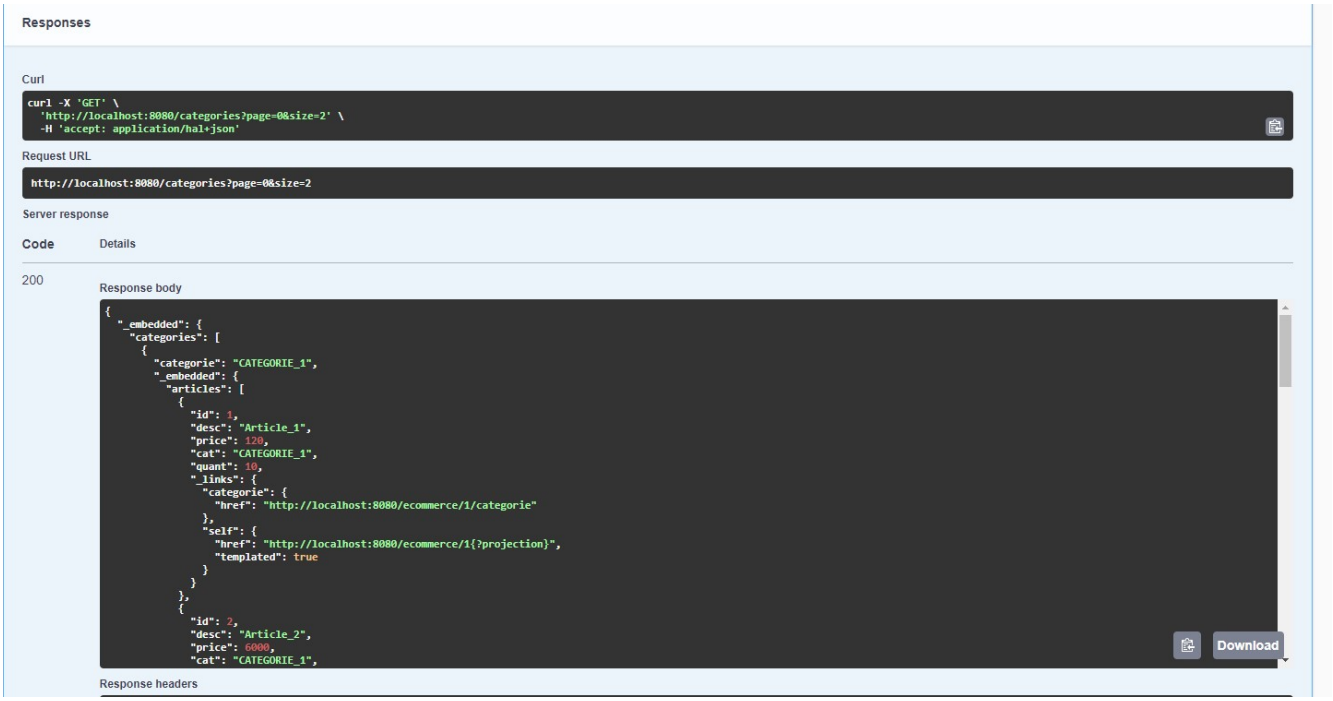


3. Tester la méthode Get avec Swagger UI

- Vous pouvez tester par exemple l'Endpoint « *Get /categories* » en suivant les étapes suivantes :
  - ✓ Cliquer sur ***Get***.
  - ✓ Cliquer ensuite sur le bouton ***Try it out*** :



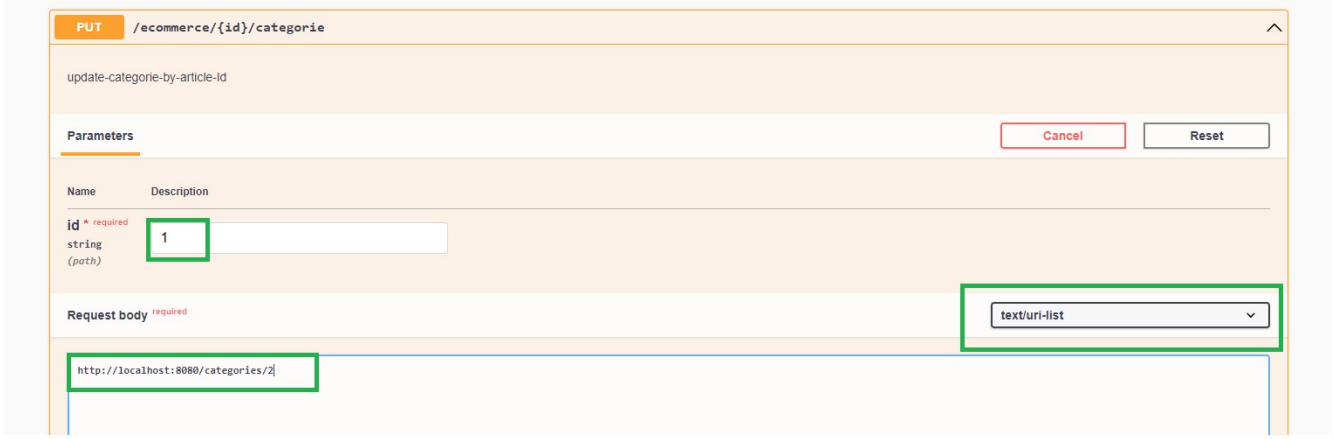
  - ✓ Entrer le numéro de la page et le size (ici on souhaite consulter les deux premières catégories).
  - ✓ Cliquer ensuite sur ***Execute***, le résultat est le suivant :



4. Tester la méthode Put avec Swagger UI

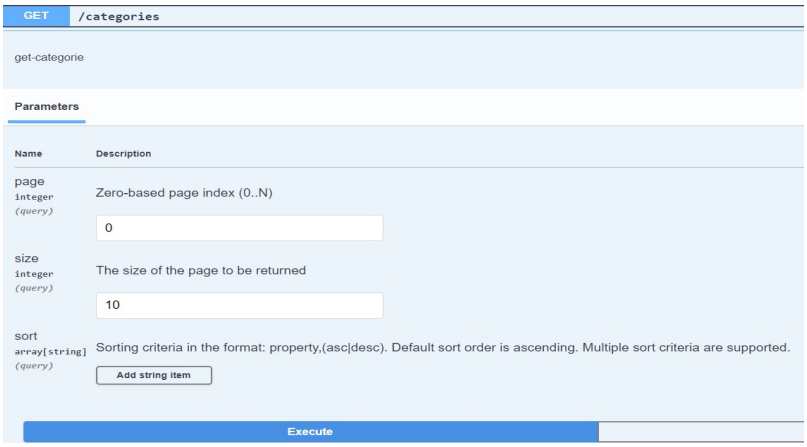
- Pour modifier la catégorie de l'article dont l'id est égal à 1, suivre les étapes suivantes :
  - ✓ Cliquer sur « ***PUT ecommerce/{id}/categorie*** » suivante :

  - ✓ Cliquer ensuite sur ***Try it out :***



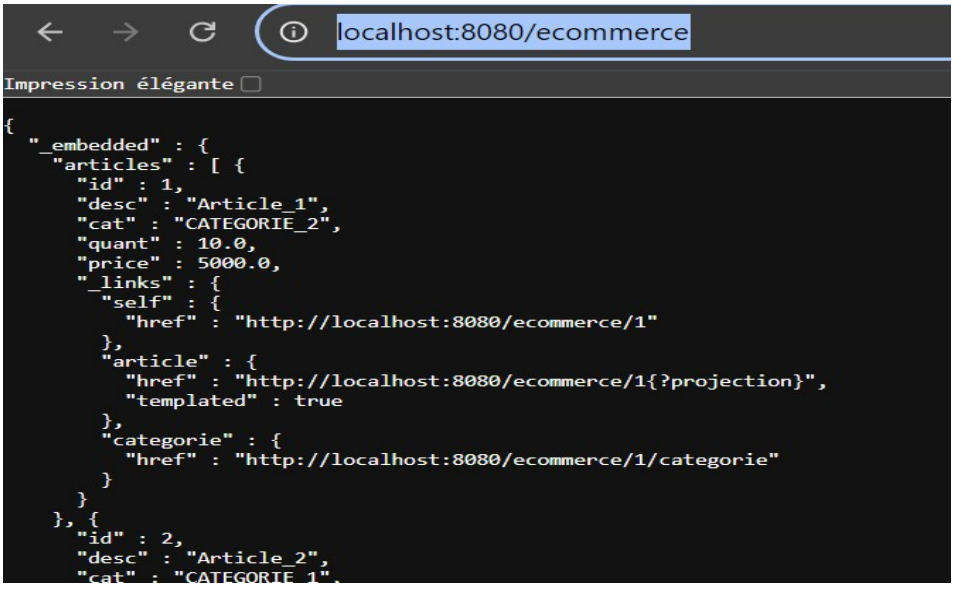
  - ✓ Entrer l'id de l'article.
  - ✓ Dans ***Request body***, choisir ***text/uri-list***
  - ✓ Entrer le lien http://localhost:8080/categories/2
  - ✓ Ensuite cliquer sur **Exécute**.

✓ Puis vérifier que l'update s'est bien passé :




```
{
  "categorie": "CATEGORIE_2",
  "_embedded": {
    "articles": [
      {
        "id": 1,
        "desc": "Article_1",
        "cat": "CATEGORIE_2",
        "quant": 10,
        "price": 5000,
        "_links": {
          "categorie": {
            "href": "http://localhost:8080/ecommerce/1/categorie"
          },
          "self": {
            "href": "http://localhost:8080/ecommerce/1{?projection}",
            "templated": true
          }
        }
      },
```


Ou bien : http://localhost:8080/ecommerce




```
{
  "_embedded" : {
    "articles" : [ {
      "id" : 1,
      "desc" : "Article_1",
      "cat" : "CATEGORIE_2",
      "quant" : 10.0,
      "price" : 5000.0,
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/ecommerce/1"
        },
        "article" : {
          "href" : "http://localhost:8080/ecommerce/1{?projection}",
          "templated" : true
        },
        "categorie" : {
          "href" : "http://localhost:8080/ecommerce/1/categorie"
        }
      }
    }, {
      "id" : 2,
      "desc" : "Article_2",
      "cat" : "CATEGORIE_1",
```

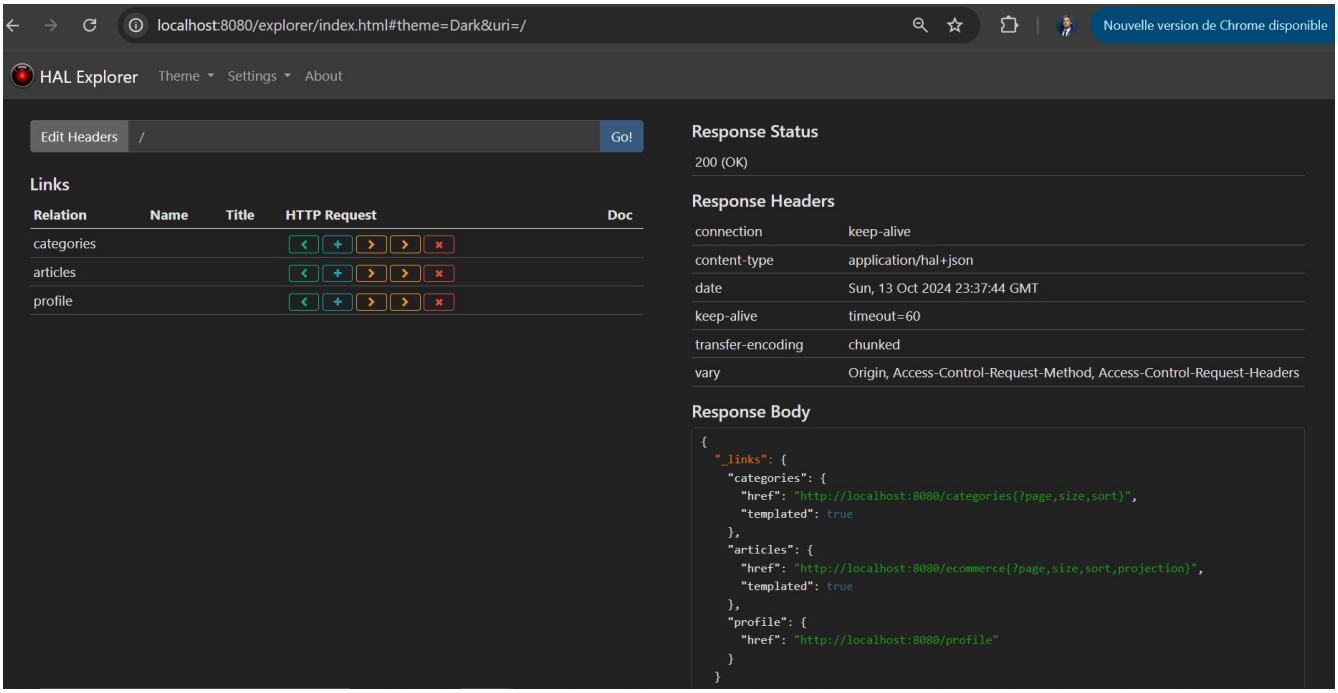
IX. HAL Explorer

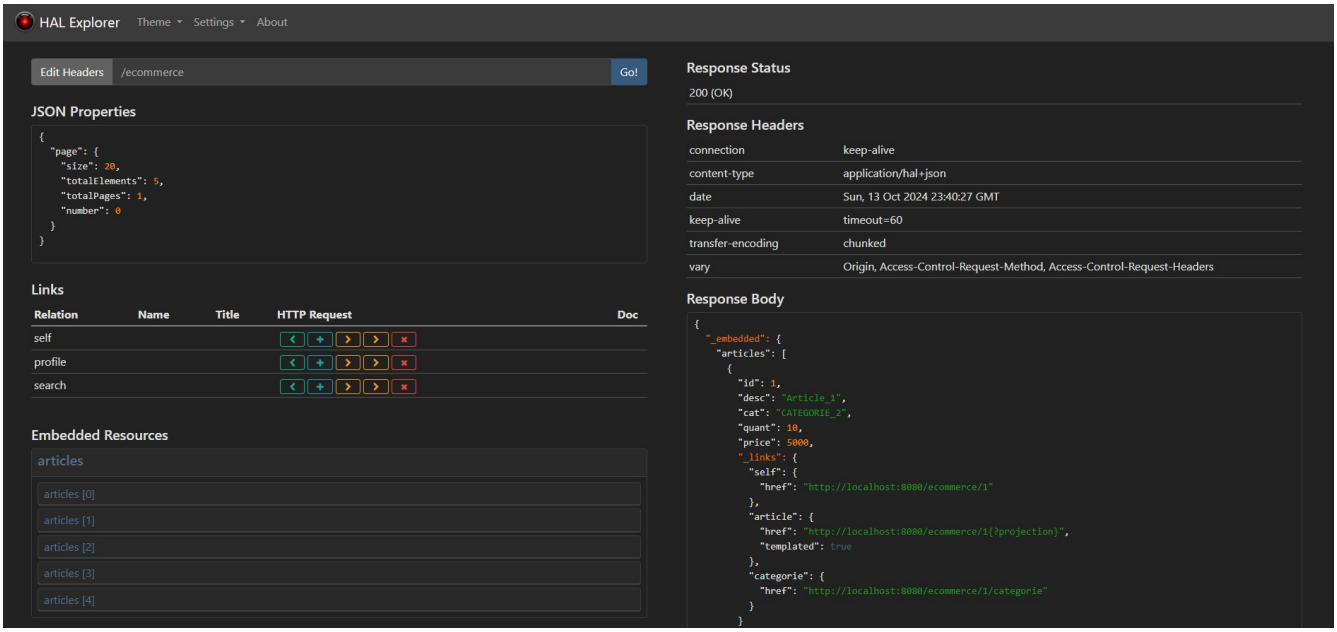
Kai Tödter a créé une application utile : HAL Explorer. Il s'agit d'une application Web basée sur Angular qui vous permet d'explorer facilement les réponses HTTP basées sur HAL et HAL-FORMS. Elle prend également en charge les profils Spring générés par Spring Data REST. Vous pouvez la pointer vers n'importe quelle API Spring Data REST et l'utiliser pour naviguer dans l'application et créer de nouvelles ressources

La liste suivante montre comment ajouter la dépendance dans Maven :

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-rest-hal-explorer</artifactId>
</dependency>
```

Si vous utilisez Spring Boot ou Spring Data, vous n'avez pas besoin de spécifier la version.
Cette dépendance configure automatiquement HAL Explorer pour qu'il soit servi lorsque vous visitez l'URI racine de votre application dans un navigateur. (REMARQUE : http://localhost:8080/
a été connecté au navigateur et a été redirigé vers l'URL affichée dans l'image suivante.)





Documentation officielle de HAL : https://datatracker.ietf.org/doc/html/draft-kelly-json-hal

Conclusion

Le code source du TP est disponible sur GITHUB :

https://github.com/abbouformations/tpdatarest.git.