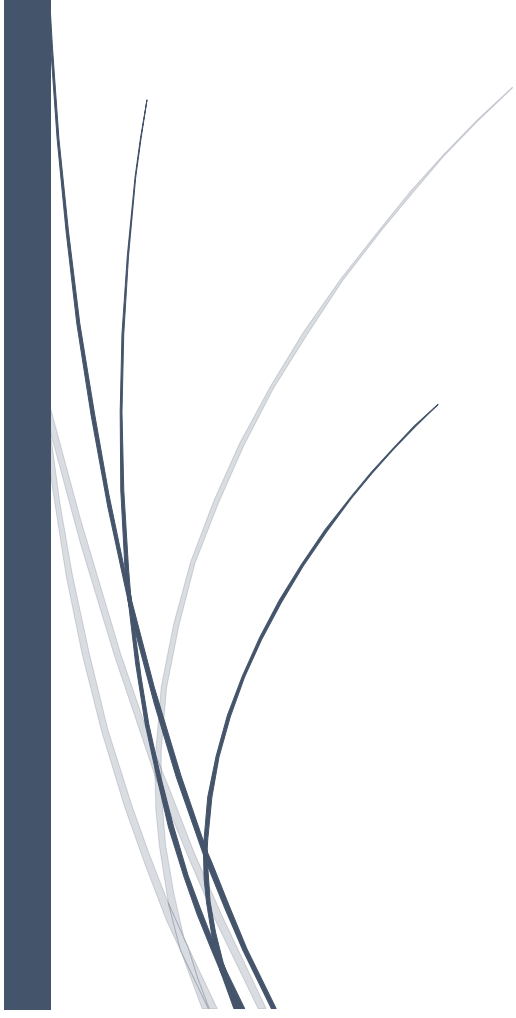
TP : Développer un service web Rest avec Spring BOOT et Spring Data Rest



Architecture des composants d'entreprise

Table des matières

I. Objectif du TP

- Utiliser le starter spring-boot-starter-data-rest pour exposer une modèle via l'architecture Rest.
- Utiliser la base de données H2.

NB : Le code source du TP est disponible sur GITHUB :

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

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 2023.1.2 (Community Edition).*

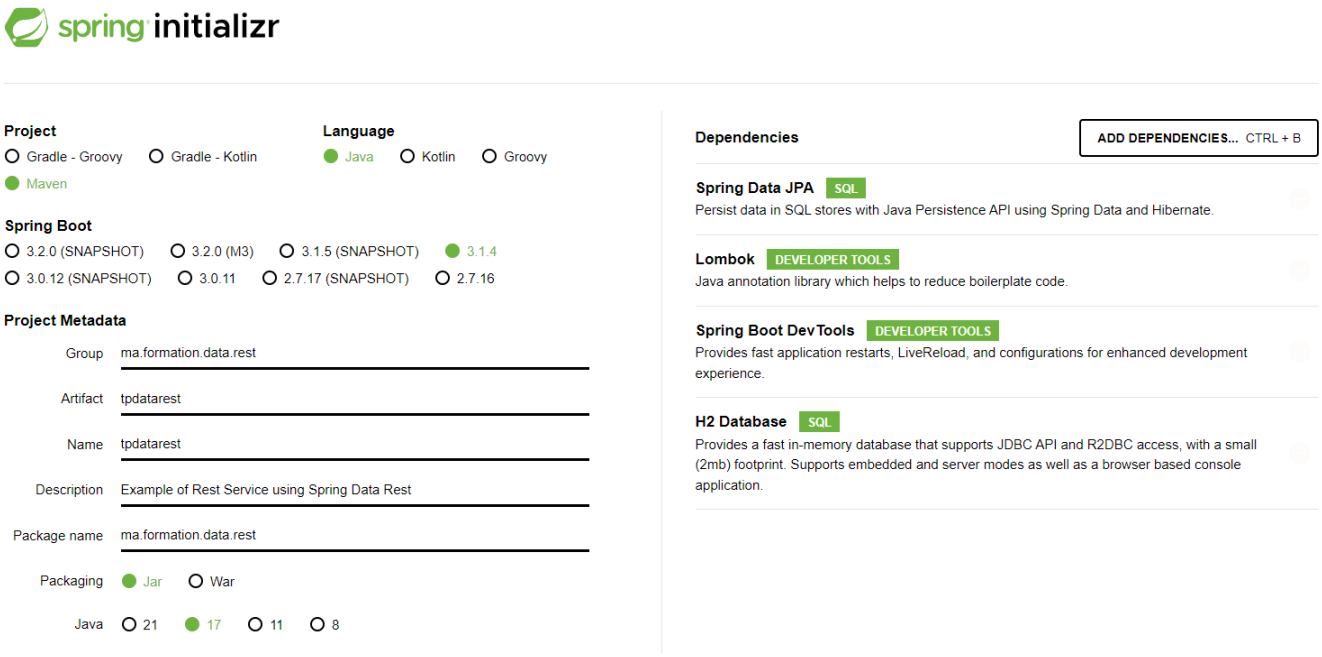
III. C'est quoi Spring Data Rest

- *Spring Data REST* fait partie du projet *Spring Data* 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 hypermédia (hypermedia-driven 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* comme *media type*.
    - 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. Développement du SW avec Spring Data Rest

a. Création du projet Maven

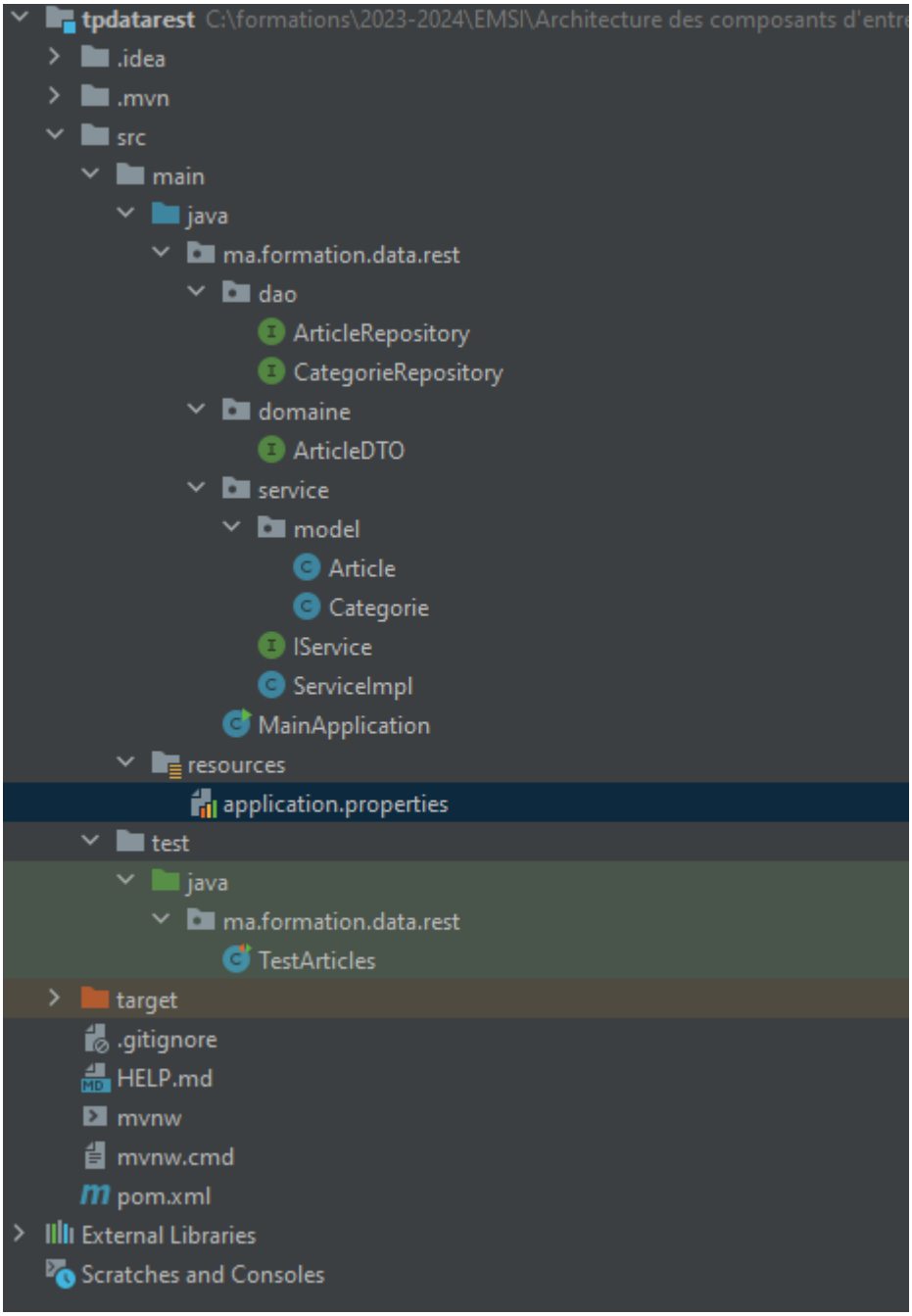
1. Aller au lien https://start.spring.io/. L'interface suivante s'affiche :



- Cocher java et Maven.
- Cocher la version 3.1.4 dans Spring Boot.
- Entrer le Group, l'artifact et le nom du package.
- Cocher jar dans Packaging.
- Cocher 17 dans Java.
- Ajouter les dépendances :
  - Spring Boot Dev Tools ;
  - Lombok ;
  - Spring Data JPA ;
  - H2.
- Enfin, cliquer sur le bouton « GENERATE » pour générer le projet Maven.
- Décompresser le ZIP et copier le projet dans votre workspace.

b. L'arborescence du projet

Créer l'arborescnce suivante :



c. Le fichier pom.xml

- Modifier le fichier pom.xml comme illustré ci-après :

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://maven.apache.org/POM/4.0.0"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
```

```
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.1.4</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>ma.formation.data.rest</groupId>
  <artifactId>tpdatarest</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>tpdatarest</name>
  <description>Example of Rest Service using Spring Data Rest</description>
  <properties>
    <java.version>17</java.version>
  </properties>
  <dependencies>
    <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>
  </dependencies>
```

```
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <configuration>
          <excludes>
            <exclude>
              <groupId>org.projectlombok</groupId>
              <artifactId>lombok</artifactId>
            </exclude>
          </excludes>
        </configuration>
      </plugin>
    </plugins>
  </build>

</project>
```

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*.

- La dépendance :

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

- Il s'agit de la base de données virtuelle H2. Il s'agit d'une base de données embarquée qui sera démarrée et arrêtée avec l'application.

d. Le fichier application.properties

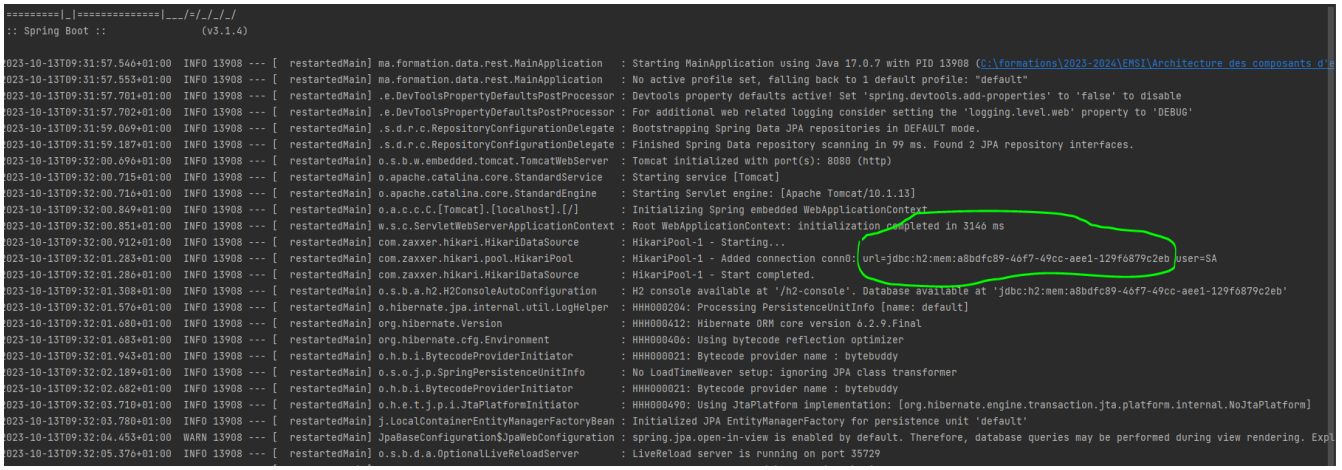
- Copier les lignes suivantes au niveau du fichier *application.properties* :

```
# Le nom de la base de données.
spring.datasource.url=jdbc:h2:mem:testdb
# Le Driver de H2
spring.datasource.driverClassName=org.h2.Driver
spring.data.jpa.repositories.bootstrap-mode=default
spring.datasource.username=sa
spring.datasource.password=
# Le Dialect : java => SQL compatible avec H2
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
# Création et modification automatique des tables
spring.jpa.hibernate.ddl-auto=update
# Pour activer la console
spring.h2.console.enabled=true
# Pour personnaliser l'URL de la console H2
spring.h2.console.path=/h2
```

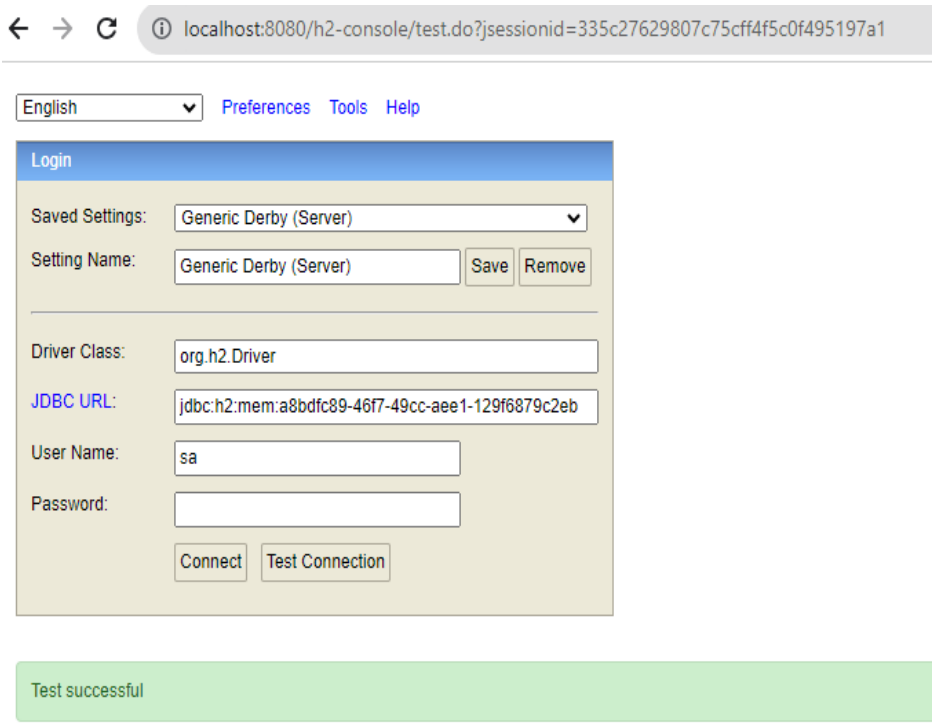
Explications :

- 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.Data;
import lombok.NoArgsConstructor;

@Entity
```

```java
@Data
@NoArgsConstructor
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;

   public Article(String description, Double price, Double quantity) {
      this.description = description;
      this.price = price;
      this.quantity = quantity;
   }

}
```

- La classe **Categorie** :

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

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

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

@Entity
@Data
@NoArgsConstructor
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 Categorie(String categorie) {
        this.id = id;
        this.categorie = categorie;
    }
}
```

f. L'interface ArticleDTO

- 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 = "articleDAO", 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* 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*.

g. Les interfaces DAO : ArticleRepository et CategorieRepository

- L'interface **ArticleRepository** :

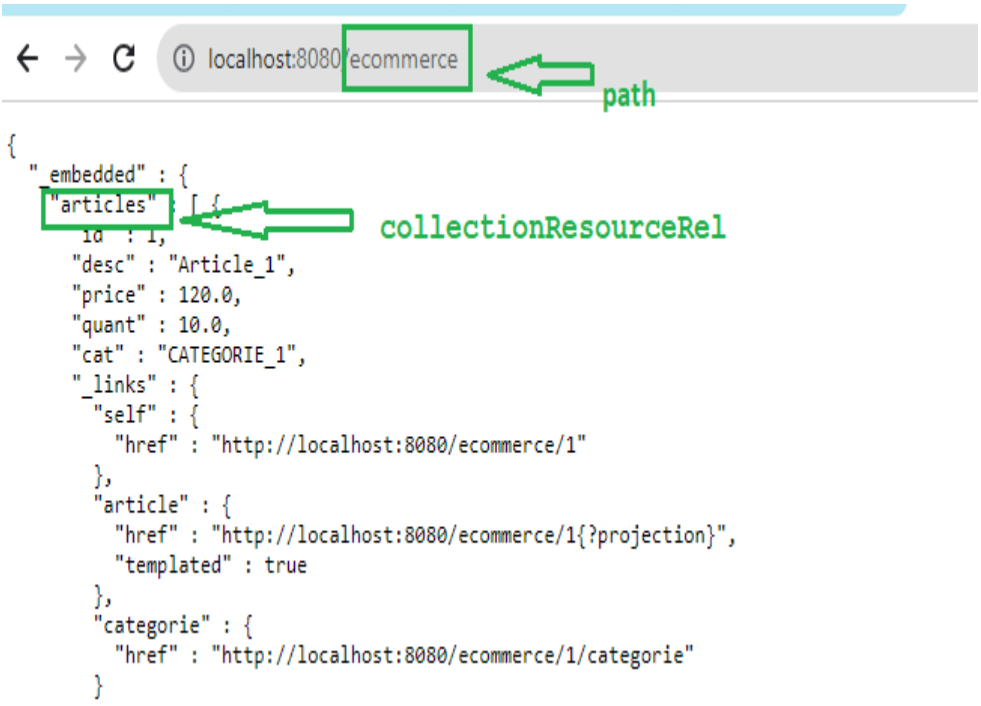
```
package ma.formation.data.rest.dao;

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 = "byDescription", rel = "customFindByDescription")
   List<Article> findByDescription(@Param("description") String description);
}
```

Explications :

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

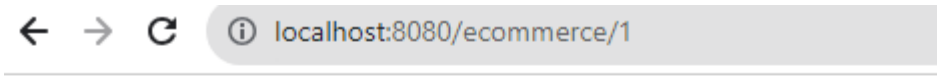
- L'attribut *collectionResourceRel* sert à configurer le nom du champ qui représentera la collection des articles dans le message JSON ou XML.

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



- L'attribut *excerptProjection* 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 :




```
{
  "description" : "Article_1",
  "price" : 120.0,
  "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"
    }
  }
}
```


- 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 par exemple http://localhost:8080/ecommerce/search/byDescription?description=Article_1 permet d'accéder au service findByDescription :

- 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> {
   Categorie findByCategorie(@Param("categorie") String categorie);
}
```

h. L'interface IService

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

import ma.formation.data.rest.service.model.Article;
import ma.formation.data.rest.service.model.Categorie;

public interface IService {
   void save(Categorie cat, Article... articles);
}
```

Explications :

- Le type ***Article…*** s'appelle ***varargs***. Càd, la méthode save(..) prends aucun paramètre ou bien 1 ou plusieurs paramètre de type Article.

i. La classe ServiceImpl

- La classe **ServiceImpl** :

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

import lombok.AllArgsConstructor;
import ma.formation.data.rest.dao.CategorieRepository;
import ma.formation.data.rest.service.model.Article;
import ma.formation.data.rest.service.model.Categorie;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
```

```java
@Service
@Transactional
@AllArgsConstructor
public class ServiceImpl implements IService {
   private CategorieRepository categorieRepository;

   @Override
   public void save(Categorie cat, Article... articleList) {
      for (Article a : articleList) {
         a.setCategorie(cat);
         cat.getArticles().add(a);
      }
      categorieRepository.save(cat);
   }
}
```

Explications :

- L'annotation ***@Transactional*** appliquée sur la classe permet de rendre toutes les méthodes de cette classe transactionnelles. Càd, soit la méthode s'exécute dans sa globalité ou non.

j. La classe MainApplication

- Modifier la classe de démarrage de *Spring Boot* comme suit :

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

import lombok.AllArgsConstructor;
import ma.formation.data.rest.service.IService;
import ma.formation.data.rest.service.model.Article;
import ma.formation.data.rest.service.model.Categorie;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
@AllArgsConstructor
public class MainApplication {

   private IService service;

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

```
  @Bean
  public CommandLineRunner initDatabase() throws Exception {
    return (a) -> {
      var categorie1 = new Categorie("CATEGORIE_1");
      var categorie2 = new Categorie("CATEGORIE_2");
      var categorie3 = new Categorie("CATEGORIE_3");

      var article1 = new Article("Article_1", 120d, 10d);
      var article2 = new Article("Article_2", 6000d, 30d);
      var article3 = new Article("Article_3", 7000d, 44d);
      var article4 = new Article("Article_4", 12000.0, 5d);
      var article5 = new Article("Article_4", 12000.0, 5d);

      service.save(categorie1, article1, article2);
      service.save(categorie2, article3, article4);
      service.save(categorie3, article5);
    };
  }
}
```

Explications :

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

k. La classe de test : TestArticles

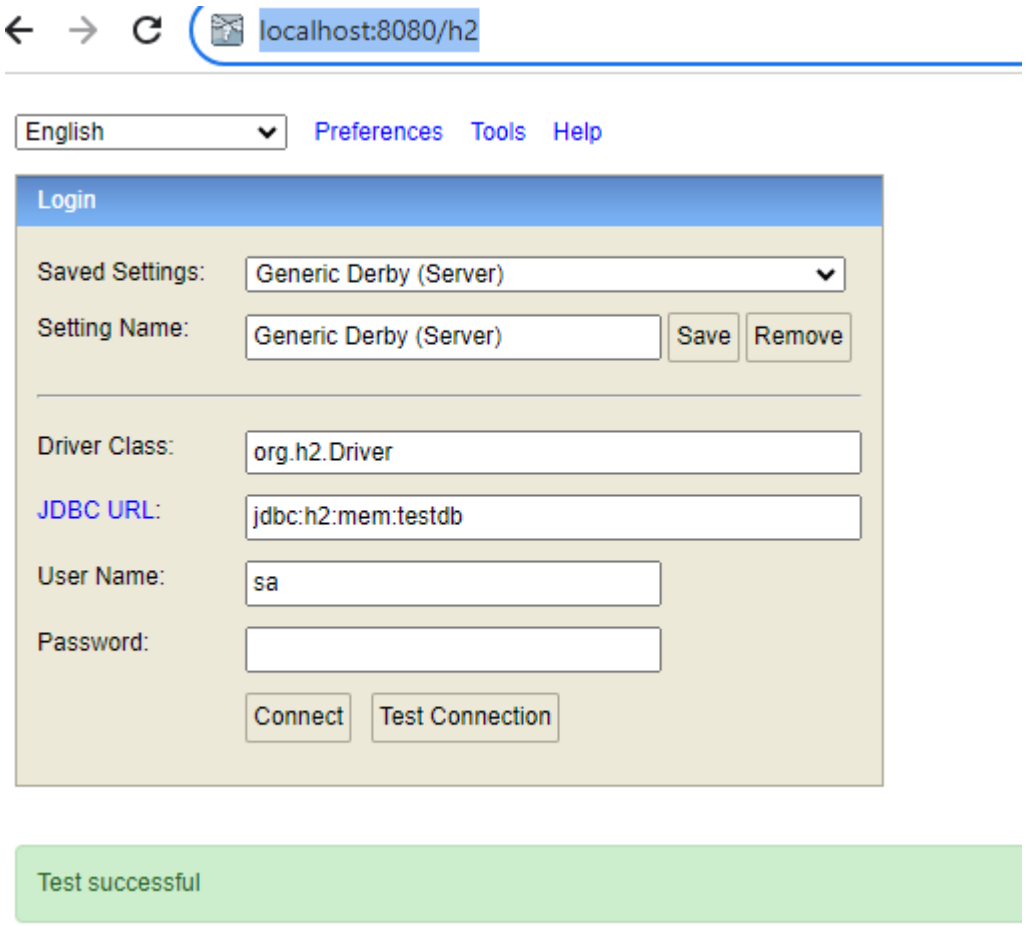
- Pour les tests unitaires, développer la classe **TestArticles** suivante :

```
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)
```

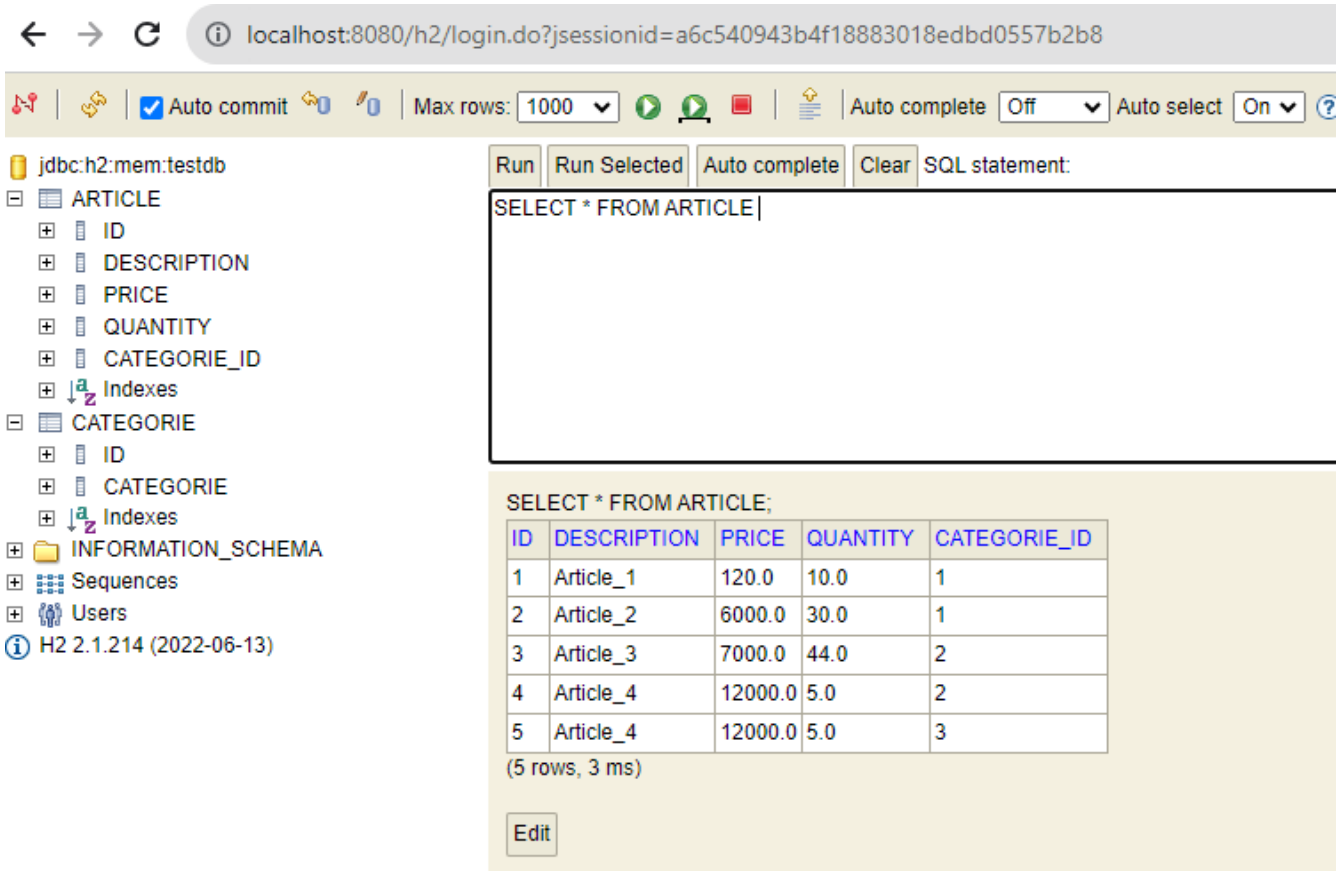
```
class TestArticles {

   @Autowired
   WebApplicationContext context;

   private MockMvc mvc;

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


   @Test
   void testGetAllArticles() throws Exception {

      mvc.perform(get("/articles")
         .contentType(MediaType.APPLICATION_JSON)
         .accept(MediaType.APPLICATION_JSON))
           .andExpect(status().isOk())
           .andExpect(jsonPath("$._embedded.articles[0].description").value("Article_1"))
           .andExpect(jsonPath("$._embedded.articles[0].price").value(120))
           .andExpect(jsonPath("$._embedded.articles[0].quantity").value(10));
   }

   @Test
   void testGetArticleById() throws Exception {

      Article article = new Article("Article_1", 120d, 10d);

      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(120D))
           .andExpect(jsonPath("$.quantity").value(10D));
   }
}
```

V. Tests avec Postman

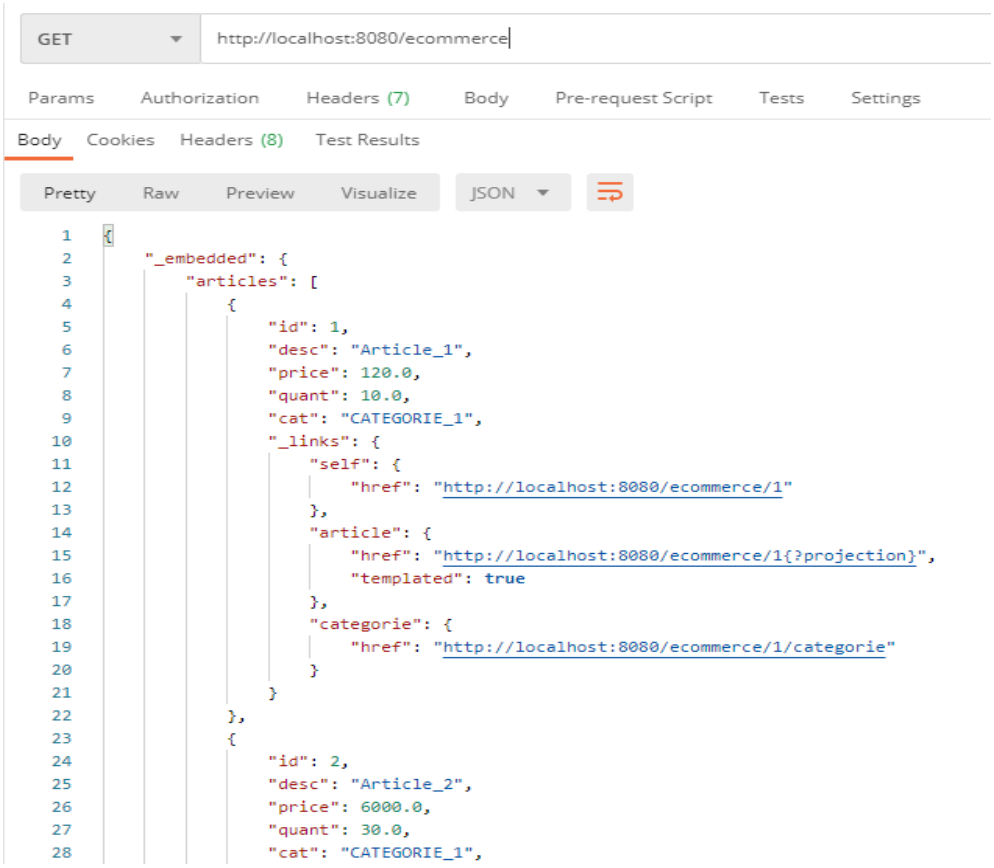
- Démarrer la méthode **MainApplication**.
- Accéder à la console via le lien http://localhost:8080/h2 :



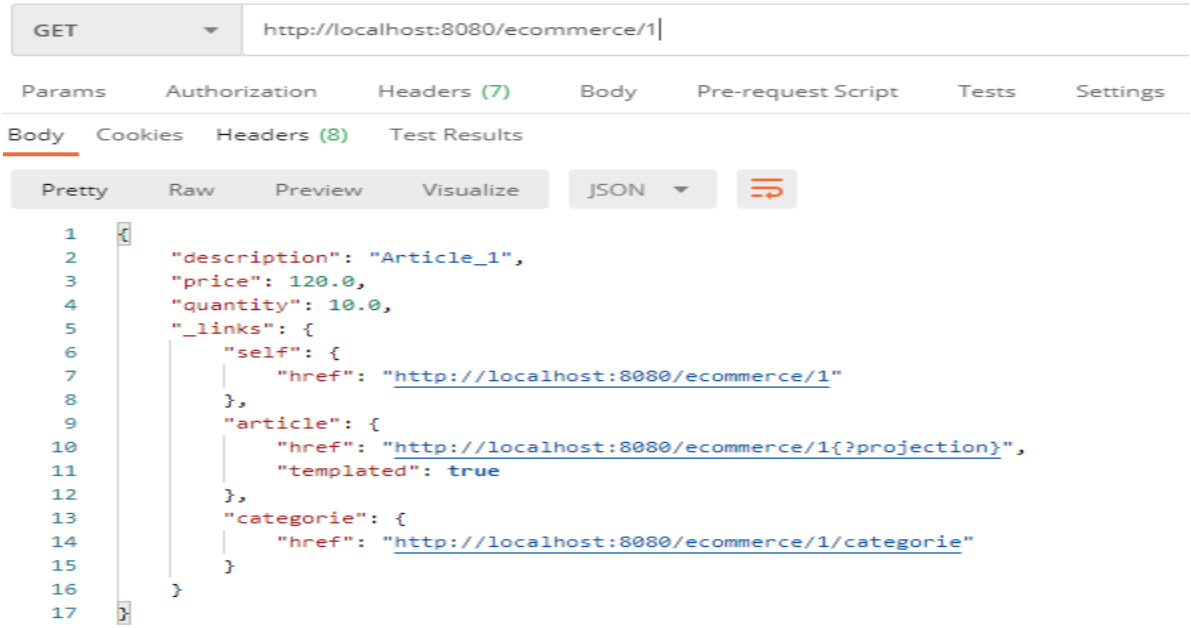
- Cliquer sur le bouton Conect pour accéder à la base de données :



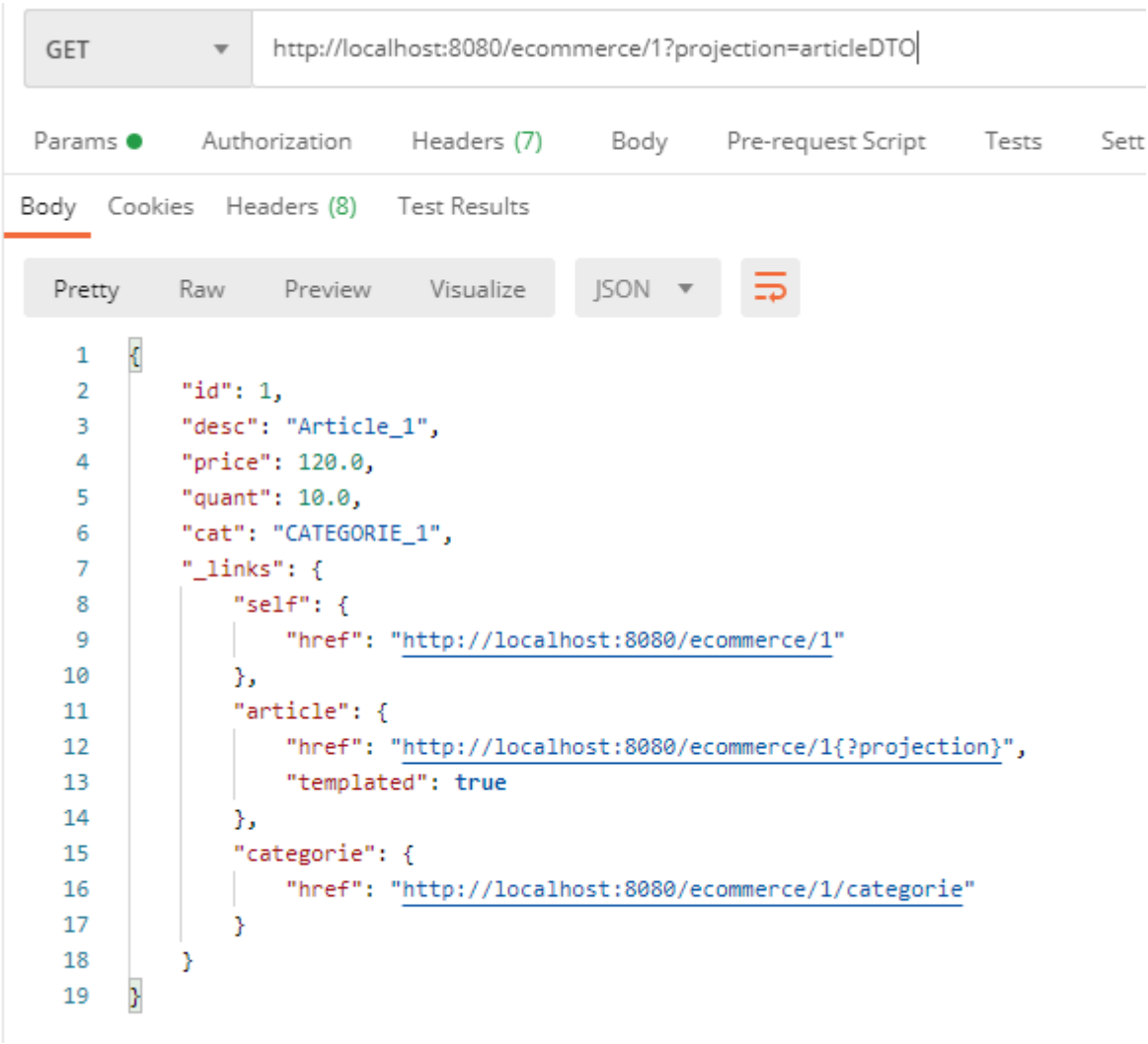
- Vérifier que les deux tables Article et Categorie ont été bien crées et initialisées.
- Tester la resource **ArticleRepository** :
  - Tester la méthode GET pour l'ensemble des articles :



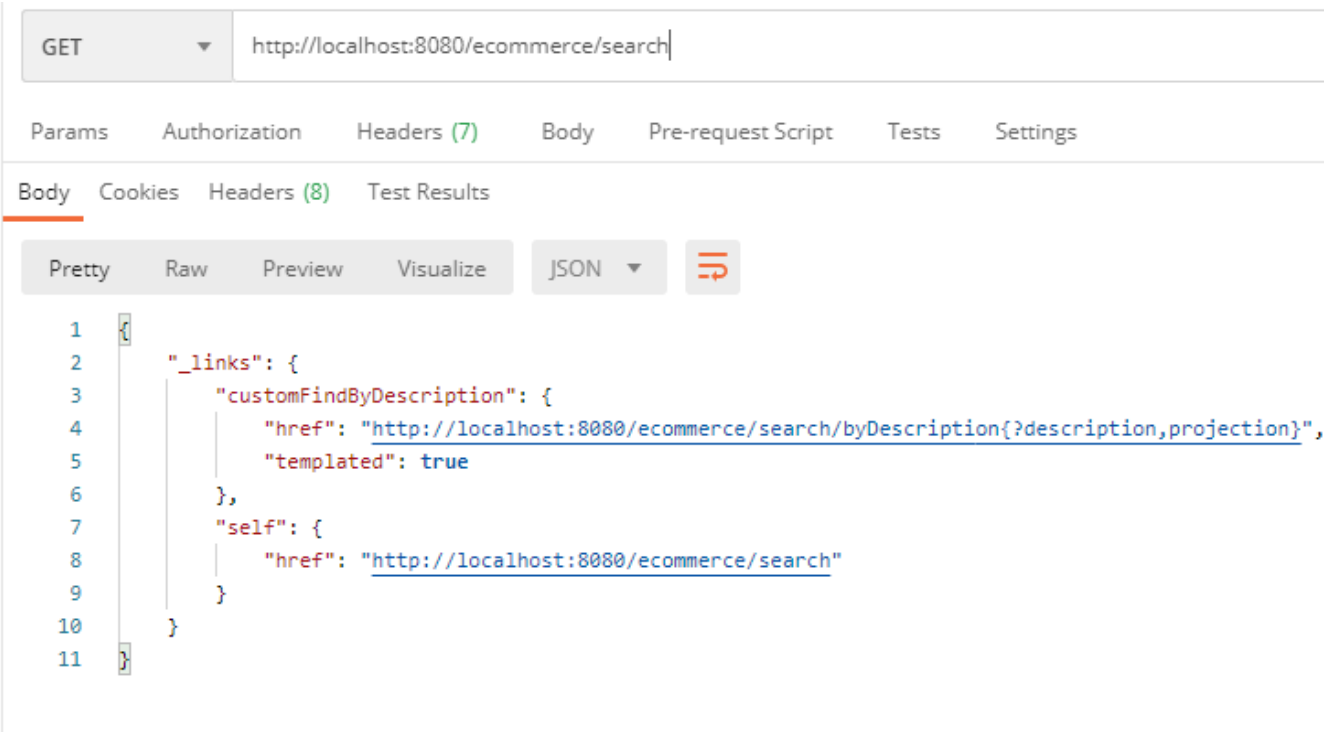
- Tester la méthode GET pour consulter un article par son ID :

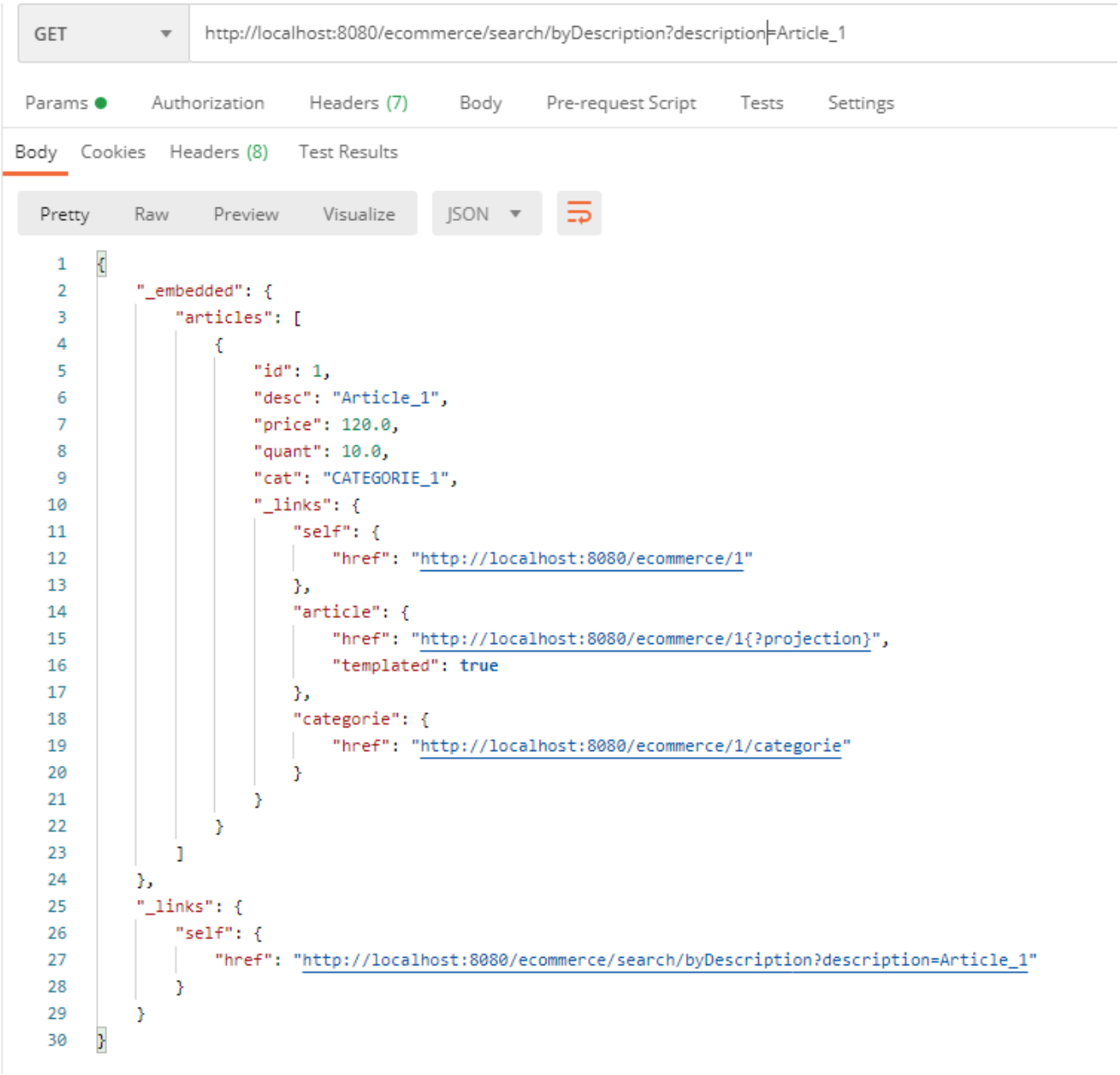


- Tester la méthode GET pour consulter un article par son ID en utilisant la projection articleDTO :



- Tester le service byDescription :


GET
http://localhost:8080/ecommerce/search
Params
Authorization
Headers (7)
Body
Pre-request Script
Tests
Settings
Body
Cookies
Headers (8)
Test Results
Pretty
Raw
Preview
Visualize
JSON
{
"_links": {
"customFindByDescription": {
"href": "http://localhost:8080/ecommerce/search/byDescription{?description,projection}",
"templated": true
},
"self": {
"href": "http://localhost:8080/ecommerce/search"
}
}
}


GET
http://localhost:8080/ecommerce/search/byDescription?description=Article_1
Params
Authorization
Headers (7)
Body
Pre-request Script
Tests
Settings
Body
Cookies
Headers (8)
Test Results
Pretty
Raw
Preview
Visualize
JSON
{
"_embedded": {
"articles": [
{
"id": 1,
"desc": "Article_1",
"price": 120.0,
"quant": 10.0,
"cat": "CATEGORIE_1",
"_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"
}
}
}
]
},
"_links": {
"self": {
"href": "http://localhost:8080/ecommerce/search/byDescription?description=Article_1"
}
}
}