

Compt Rendu

5ème année
Ingénieur Informatique & Réseaux

TP3

SPRING DATA REST & SPRING BOOT

Réalisé par : IMANE Chakrellah
YASSINE Ech-chaoui

Prof : Pr.Oussama
Classe: 5IIR-11

2025-2026

Table des matières

1. Objectif du TP	2
2. Plan du compte rendu.....	Erreur ! Signet non défini.
3. Architecture du projet.....	2
4. Notions importantes dans le code	3
5. Explication des tests JUnit (Test Unitaire).....	4
6. Explication de la console H2	5
7. Tests Postman (Explications des Captures)	6
1) GET /ecommerce	6
2) GET /ecommerce/1	6
3) GET /ecommerce/1?projection=articleDTO	7
4) GET /ecommerce/search/byCategorie?categorie=CATEGORIE_1	7
5) POST /ecommerce - Crédation d'un article.....	8
6) POST /categories - Crédation d'une catégories.....	8
7) PUT /ecommerce/6/categorie - Associer article 6 à catégories 4.....	9
8. Tests Swagger UI.....	10
9. HAL Explorer.....	11
10. Conclusion	11
11. Questions	12

Tout les TPs Réalisés ce trouve dans le lien suivant qui contient le projet + CR :

<https://github.com/CHAKRELLAH44/JEE-ARCHITECTURE.git>

1. Objectif du TP

L'objectif de ce TP est de développer un **service web RESTful** basé sur :

- **Spring Boot**
- **Spring Data JPA**
- **Spring Data REST**
- **Base de données H2 (en mémoire)**
- **OpenAPI + Swagger UI**

Ce service expose automatiquement des endpoints REST grâce aux Repository Spring Data REST, sans écrire de contrôleur.

Le but est de comprendre :

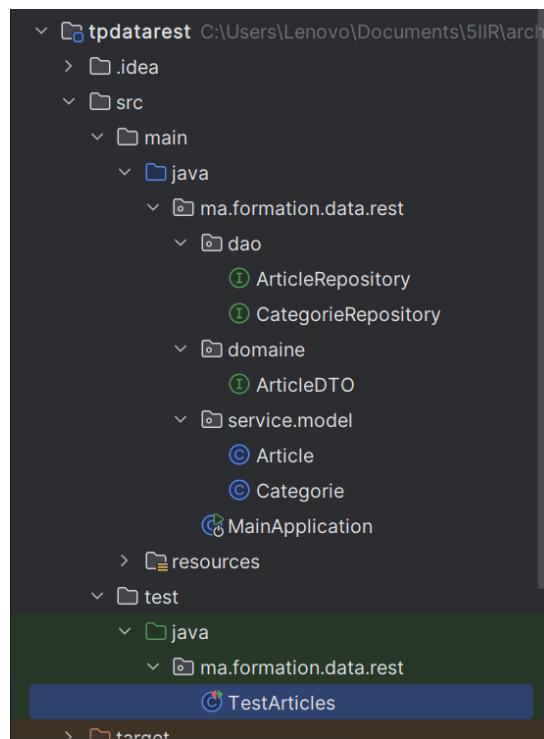
Comment exposer une API REST sans controller

Comment utiliser Spring Data REST pour générer automatiquement :

- GET
- POST
- PUT
- Search
 - Utiliser la base de données **H2 en mémoire**.
 - Tester l'API avec **Postman / Swagger UI**
 - Découvrir les **projections** (DTO) pour personnaliser les réponses JSON.
 - Comment associer des entités via REST (relation ManyToOne)
 - Tester avec des **Tests unitaires MockMvc**

Le TP permet de comprendre comment gagner énormément de temps dans le développement des API REST grâce à Spring Data REST.

2. Architecture du projet



4. Notions importantes dans le code

Spring Data REST

- Génère automatiquement les endpoints REST à partir des Repository
- Plus besoin de @RestController
- Exemple :
/ecommerce expose ArticleRepository
/categories expose CategorieRepository

JpaRepository

- Fournit :
findAll()
findById()
save()
delete()
- Spring Data Rest expose automatiquement ces méthodes via HTTP

Projection (ArticleDTO)

Permet de personnaliser les données retournées.

Exemple :

```
@Projection(name = "articleDTO", types = Article.class)
```

Appel :

```
GET /ecommerce/1?projection=articleDTO
```

H2 Database

- Base embarquée en mémoire
- Recharge les données à chaque démarrage
- Console disponible via /h2

CommandLineRunner

Initialise la base automatiquement au lancement :

- Catégories
- Articles
- Relations entre les deux

Relations JPA

Article → Categorie

ManyToOne

→ Un article appartient à une catégorie

Categorie → Article

OneToMany

→ Une catégorie contient plusieurs articles

5. Explication des tests JUnit (Test Unitaire)

The screenshot shows a terminal window displaying JUnit test results. At the top, it says "TestArticles (ma.formation.data.rest) 1 sec". Below that, there are two test cases: "testGetArticleById()" and "test GetAllArticles()". Both tests are marked with a green checkmark, indicating they passed. The execution time for "testGetArticleById()" is listed as "1 sec 339 ms", and for "test GetAllArticles()", it is "262 ms".

Explication :

- `test GetAllArticles ()`
- `testGetArticleById ()`

Les deux tests passent, donc :

- L'API GET `/ecommerce` retourne bien la liste des articles au format JSON HAL
- L'API GET `/ecommerce/1` renvoie correctement l'article avec id=1

Les assertions MockMvc ont validé :

- `description`
- `price`
- `quantity`
- La structure JSON renvoyée par Spring Data REST

Cela confirme que Spring Boot + Spring Data REST sont correctement configurés.

6. Explication de la console H2

6.1. Connexion réussie à H2

Cela confirme :

- Que Spring Boot a bien démarré H2
- Que la BD en mémoire testdb est accessible

The screenshot shows the H2 Console login interface. At the top, there's a dropdown for 'English' and links for 'Preferences', 'Tools', and 'Help'. Below that is a 'Saved Settings' dropdown set to 'Generic H2 (Embedded)'. A 'Setting Name' input field also contains 'Generic H2 (Embedded)'. There are 'Save' and 'Remove' buttons. The 'Driver Class' is set to 'org.h2.Driver'. The 'JDBC URL' is 'jdbc:h2:mem:testdb'. The 'User Name' is 'sa' and the 'Password' field is empty. At the bottom are 'Connect' and 'Test Connection' buttons. A green status bar at the bottom says 'Test successful'.

6.2. Affichage de la table ARTICLE

The screenshot shows the H2 Console interface with the 'ARTICLE' table selected. On the left is a tree view of the database schema, including 'ARTICLE' with columns 'ID', 'DESCRIPTION', 'PRICE', 'QUANTITY', and 'CATEGORIE_ID', and 'Indexes'. Other tables like 'CATEGORIE' and 'INFORMATION_SCHEMA' are also listed. On the right, a SQL query 'SELECT * FROM ARTICLE;' is entered, and the results are displayed in a table:

ID	DESCRIPTION	PRICE	QUANTITY	CATEGORIE_ID
1	Article_1	5000.0	10.0	1
2	Article_2	6000.0	20.0	1
3	Article_3	7000.0	30.0	2
4	Article_4	8000.0	40.0	2
5	Article_5	9000.0	50.0	3

Below the table, it says '(5 rows, 4 ms)' and there's an 'Edit' button.

On voit :

- Tables : ARTICLE et CATEGORIE
- Requête exécutée : SELECT * FROM ARTICLE;

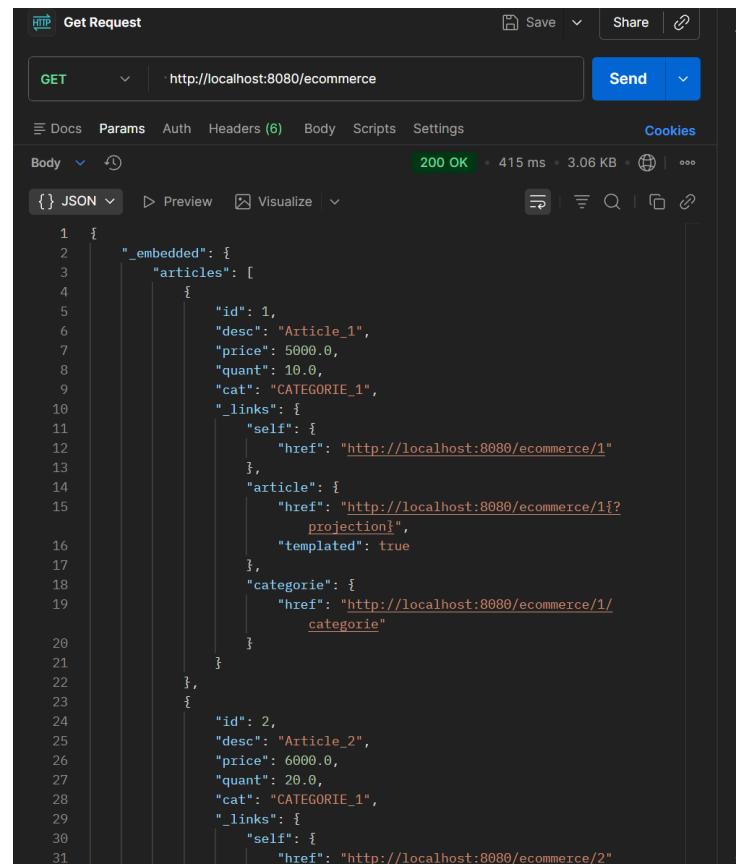
7. Tests Postman

1) GET /ecommerce

Explication :

- Retourne une **liste paginée** de tous les articles
- Le champ `_embedded.articles` contient les articles
- Les champs proviennent de la **projection ArticleDTO**
→ `id`, `desc`, `price`, `quant`, `cat`
- Les liens `_links` permettent d'accéder aux ressources liées (`self`, `categorie...`)

Spring Data REST applique **automatiquement la projection par défaut.**



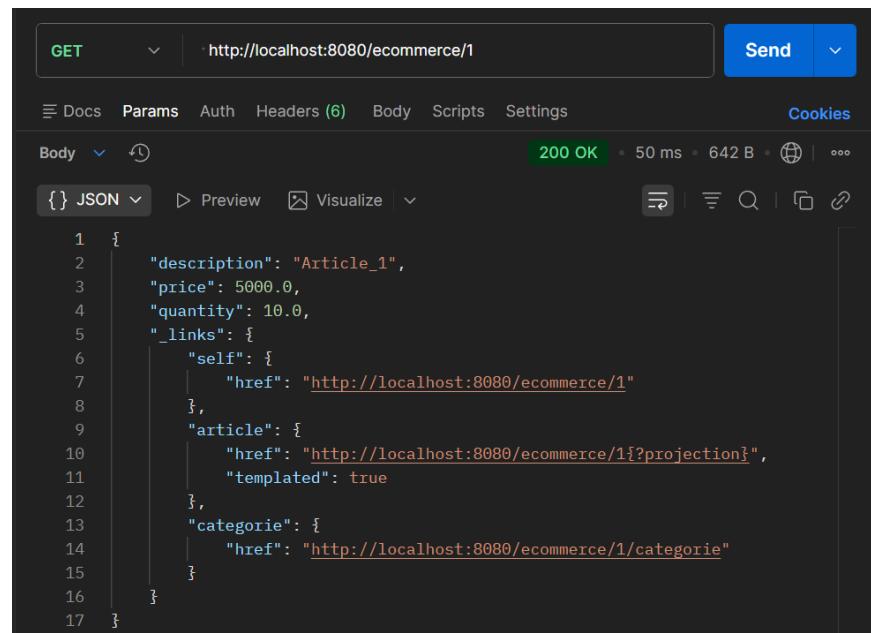
```
1 {  
2   "_embedded": {  
3     "articles": [  
4       {  
5         "id": 1,  
6         "desc": "Article_1",  
7         "price": 5000.0,  
8         "quant": 10.0,  
9         "cat": "CATEGORIE_1",  
10        "_links": {  
11          "self": {  
12            "href": "http://localhost:8080/ecommerce/1"  
13          },  
14          "article": {  
15            "href": "http://localhost:8080/ecommerce/1{?projection}",  
16            "templated": true  
17          },  
18          "categorie": {  
19            "href": "http://localhost:8080/ecommerce/1/categorie"  
20          }  
21        },  
22      },  
23      {  
24        "id": 2,  
25        "desc": "Article_2",  
26        "price": 6000.0,  
27        "quant": 20.0,  
28        "cat": "CATEGORIE_1",  
29        "_links": {  
30          "self": {  
31            "href": "http://localhost:8080/ecommerce/2"  
32          }  
33        }  
34      }  
35    ]  
36  }  
37 }
```

2) GET /ecommerce/1

Explication :

- Retourne l'article complet sans projection
- Champs originaux : `description`, `price`, `quantity`
- Inclus les liens HAL (`_links.self`)

Spring Data REST n'utilise pas la projection pour `findById`.

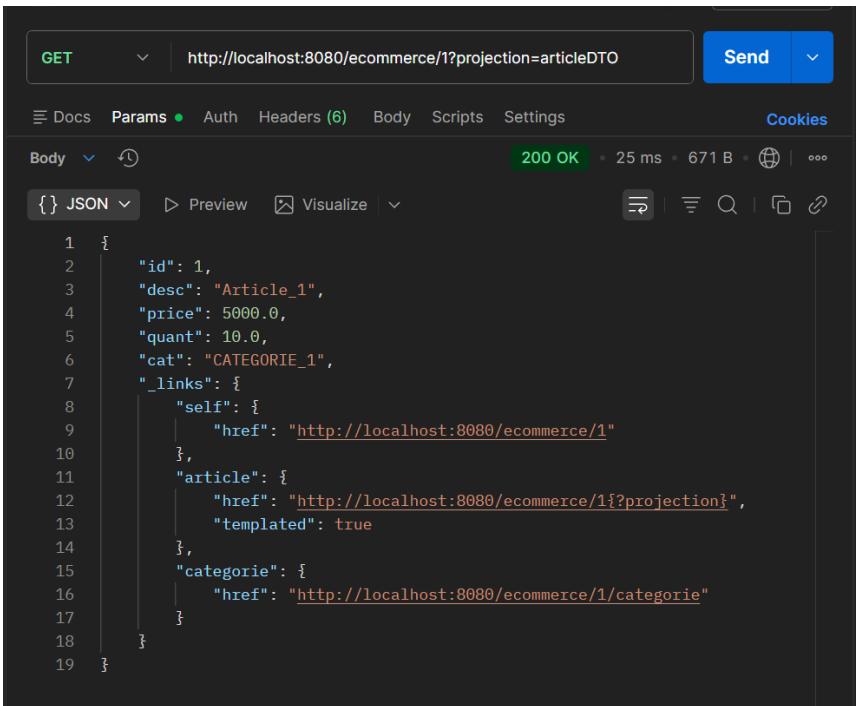


```
1 {  
2   "description": "Article_1",  
3   "price": 5000.0,  
4   "quantity": 10.0,  
5   "_links": {  
6     "self": {  
7       "href": "http://localhost:8080/ecommerce/1"  
8     },  
9     "article": {  
10       "href": "http://localhost:8080/ecommerce/1{?projection}",  
11       "templated": true  
12     },  
13     "categorie": {  
14       "href": "http://localhost:8080/ecommerce/1/categorie"  
15     }  
16   }  
17 }
```

3) GET /ecommerce/1?projection=articleDTO

Explication :

- Ici, Spring applique la projection ArticleDTO
- Les champs sont transformés :
→ desc, quant, cat
- Idéal pour exposer uniquement les données nécessaires



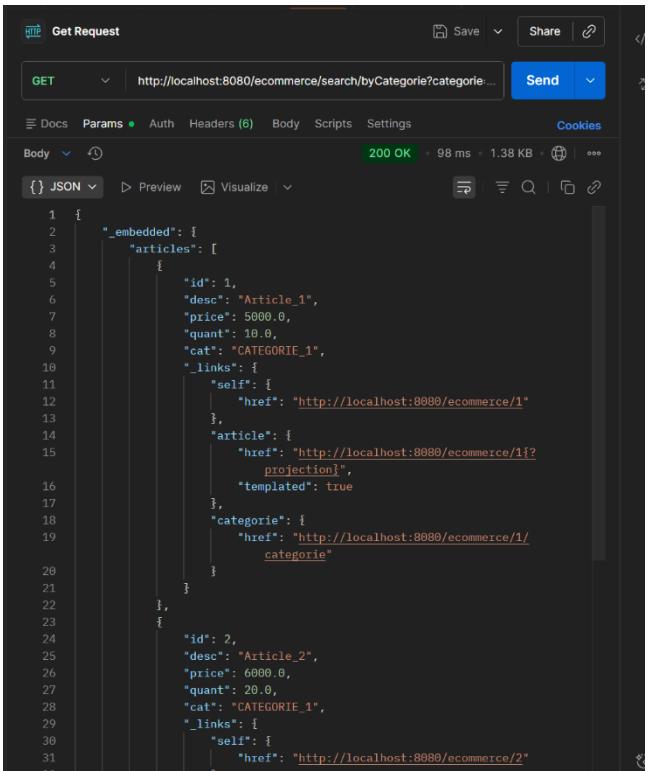
```
1 {  
2   "id": 1,  
3   "desc": "Article_1",  
4   "price": 5000.0,  
5   "quant": 10.0,  
6   "cat": "CATEGORIE_1",  
7   "_links": {  
8     "self": {  
9       "href": "http://localhost:8080/ecommerce/1"  
10    },  
11    "article": {  
12      "href": "http://localhost:8080/ecommerce/1?projection",  
13      "templated": true  
14    },  
15    "categorie": {  
16      "href": "http://localhost:8080/ecommerce/1/categorie"  
17    }  
18  }  
19 }
```

4) GET /ecommerce/search/byCategorie?categorie=CATEGORIE 1

Explication :

- Retourne tous les articles dont la catégorie = CATEGORIE_1
- Le nom du service byCategorie est défini par :

```
@RestResource(path = "byCategorie")
```



```
1 {  
2   "_embedded": {  
3     "articles": [  
4       {  
5         "id": 1,  
6         "desc": "Article_1",  
7         "price": 5000.0,  
8         "quant": 10.0,  
9         "cat": "CATEGORIE_1",  
10        "_links": {  
11          "self": {  
12            "href": "http://localhost:8080/ecommerce/1"  
13          },  
14          "article": {  
15            "href": "http://localhost:8080/ecommerce/1?projection",  
16            "templated": true  
17          },  
18          "categorie": {  
19            "href": "http://localhost:8080/ecommerce/1/  
20            categorie"  
21          }  
22        },  
23        {  
24          "id": 2,  
25          "desc": "Article_2",  
26          "price": 6000.0,  
27          "quant": 20.0,  
28          "cat": "CATEGORIE_1",  
29          "_links": {  
30            "self": {  
31              "href": "http://localhost:8080/ecommerce/2"  
32            }  
33          }  
34        }  
35      ]  
36    }  
37 }
```

5) POST /ecommerce – Cr eation d'un article

Explication :

- Spring Data REST cr e automatiquement un nouvel article
- Le statut HTTP 201 Created indique que la cr eation est r ussie
- `_links.self` pointe vers `/ecommerce/6`

Par d faut, l'article n'a pas de cat gorie.

```
POST http://localhost:8080/ecommerce
```

Body raw JSON

```
1 {
2   "description": "Article_6",
3   "price": 10000,
4   "quantity": 60
5 }
```

Body JSON Preview Visualize

```
1 {
2   "description": "Article_6",
3   "price": 10000.0,
4   "quantity": 60.0,
5   "_links": {
6     "self": {
7       "href": "http://localhost:8080/ecommerce/6"
8     },
9     "article": {
10      "href": "http://localhost:8080/ecommerce/6?projection",
11      "templated": true
12    },
13    "categorie": {
14      "href": "http://localhost:8080/ecommerce/6/categorie"
15    }
16  }
17 }
```

6) POST /categories – Cr eation d'une cat gorie

Explication :

- Nouvelle cat gorie CATEGORIE_4 bien cr e e
- Spring Data REST renvoie les liens automatiques :
`/categories/4,`
`/categories/4/articles`

```
POST http://localhost:8080/categories
```

Body raw JSON

```
1 {
2   "categorie": "CATEGORIE_4"
3 }
```

Body JSON Preview Visualize

```
1 {
2   "categorie": "CATEGORIE_4",
3   "_links": {
4     "self": {
5       "href": "http://localhost:8080/categories/4"
6     },
7     "categorie": {
8       "href": "http://localhost:8080/categories/4"
9     },
10    "articles": {
11      "href": "http://localhost:8080/categories/4/articles{?projection}",
12      "templated": true
13    }
14  }
15 }
```

7) PUT /ecommerce/6/categorie – Associer article 6 à catégorie 4

Explication :

- Méthode HAL standard
- Content-Type: text/uri-list obligatoire
- Le body contient l'URL de la catégorie :

http://localhost:8080/categories/4

- Résultat **204 No Content** = association réussie

The screenshot shows a POSTMAN interface. The method is set to PUT, and the URL is http://localhost:8080/ecommerce/6/categorie. The 'Body' tab is selected, and the raw text contains the URL 'http://localhost:8080/categories/4'. The response section shows a status of 204 No Content.

Vérification H2 – Article 6 a bien CATEGORIE_ID = 4

Explication :

La BD montre maintenant :

L'association a bien fonctionné.

The screenshot shows the H2 database interface. On the left, there is a tree view of the schema with nodes like jdbc:h2:mem:testdb, ARTICLE, CATEGORIE, INFORMATION_SCHEMA, Sequences, and Users. The timestamp is H2 2.1.214 (2022-06-13). On the right, there is a SQL editor with the query 'SELECT * FROM ARTICLE;' and a results table. The table has columns ID, DESCRIPTION, PRICE, QUANTITY, and CATEGORIE_ID. There are six rows of data, with the last row (Article_6) having CATEGORIE_ID = 4, which is highlighted with a red border. The results are labeled '(6 rows, 1 ms)'.

ID	DESCRIPTION	PRICE	QUANTITY	CATEGORIE_ID
1	Article_1	5000.0	10.0	1
2	Article_2	6000.0	20.0	1
3	Article_3	7000.0	30.0	2
4	Article_4	8000.0	40.0	2
5	Article_5	9000.0	50.0	2
6	Article_6	10000.0	60.0	4

8. Tests Swagger UI

The screenshot shows the Swagger UI interface with two requests for the `/categories` endpoint.

Request 1:

- Path Parameters:** `id`: 1
- Request URL:** `http://localhost:8080/categories/1`
- Server response:**
 - Code:** 200
 - Response body:**

```
{
  "_embedded": {
    "categories": [
      {
        "categorie": "CATEGORIE_1",
        "embedded": {
          "articles": [
            {
              "id": 1,
              "desc": "Article_1",
              "price": 1000,
              "quant": 1000,
              "cat": "CATEGORIE_1",
              "link": "http://localhost:8080/categories/1/categorie"
            }
          ],
          "self": {
            "href": "http://localhost:8080/categories/1?projection"
          }
        }
      },
      {
        "id": 2,
        "desc": "Article_2",
        "price": 2000,
        "quant": 2000
      }
    ]
  }
}
```
 - Response headers:**
 - connection: keep-alive
 - content-type: application/hal+json

Request 2:

- Path Parameters:** `page`: 0, `size`: 2
- Request URL:** `http://localhost:8080/categories?page=0&size=2`
- Server response:**
 - Code:** 200
 - Response body:**

```
{
  "page": 0,
  "size": 2,
  "totalPages": 1,
  " totalPages": 1,
  "totalElements": 2,
  "totalElements": 2,
  "content": [
    {
      "categorie": "CATEGORIE_1",
      "embedded": {
        "articles": [
          {
            "id": 1,
            "desc": "Article_1",
            "price": 1000,
            "quant": 1000
          }
        ]
      }
    },
    {
      "categorie": "CATEGORIE_2",
      "embedded": {
        "articles": [
          {
            "id": 2,
            "desc": "Article_2",
            "price": 2000,
            "quant": 2000
          }
        ]
      }
    }
  ]
}
```
 - Response headers:**
 - connection: keep-alive
 - content-type: application/hal+json

Explication :

- Swagger UI lit automatiquement l'API générée par Spring Data REST
- On peut tester les méthodes :
 - GET /categories
 - GET /ecommerce
 - PUT association
- Pagination visible → page, size

Swagger documente même les endpoints générés automatiquement.

9. HAL Explorer

The screenshot shows the HAL Explorer interface for a Spring Data REST API. At the top, there's a navigation bar with 'HAL Explorer', 'Theme', 'Settings', and 'About'. Below it, a header bar shows 'Edit Headers' and the URL '/ecommerce'. A 'Go!' button is on the right.

JSON Properties: A code block displays a JSON object representing a page:

```
{  
    "page": {  
        "size": 20,  
        "totalElements": 5,  
        "totalPages": 1,  
        "number": 0  
    }  
}
```

Links: A table lists three links:

Relation	Name	Title	HTTP Request	Doc
self			< > + >> >>> >>>>	
profile			< > + > >> >>> >>>>	
search			< > + > >> >>> >>>>	

Embedded Resources: A section titled 'articles' lists five items:

- articles [0]
- articles [1]
- articles [2]
- articles [3]
- articles [4]

Response Status: Shows '200 (OK)'.

Response Headers: A table lists various headers:

connection	keep-alive
content-type	application/hal+json
date	Mon, 01 Dec 2025 13:27:15 GMT
keep-alive	timeout=60
transfer-encoding	chunked
vary	Origin, Access-Control-Request-Method, Access-Control-Request-Headers

10. Conclusion

Ce TP a permis de :

- Comprendre comment exposer automatiquement une API REST avec Spring Data REST
- Manipuler des entités JPA et leurs relations
- Générer des projections personnalisées
- Tester l'API via Postman, Swagger UI et HAL Explorer
- Vérifier les données via H2 Console
- Initialiser automatiquement la BD via CommandLineRunner
- Écrire des tests MockMvc pertinents

Spring Data REST permet d'accélérer énormément le développement en supprimant la nécessité d'écrire des contrôleurs pour les opérations CRUD.

Quelque question que je me suis posée :

1. C'est quoi un test MockMvc ?

MockMvc est un outil de test intégré à Spring qui permet de **tester les endpoints REST sans lancer un vrai serveur.**

Il simule des requêtes HTTP (GET, POST, PUT...)

Il vérifie la réponse (statut, JSON, contenu...)

Il est utilisé dans les tests unitaires JUnit

Résumé :

MockMvc teste l'API en interne, sans Postman, sans navigateur, sans serveur réel.

2. Différence entre Postman et Swagger

Postman

- Outil externe
- Permet de tester n'importe quelle API
- Plus puissant pour les scénarios complexes (auth, collections, automatisation)

Swagger UI

- Généré automatiquement par Spring
- Affiche la documentation de l'API
- Permet aussi de tester les endpoints, mais surtout pour la **découverte** de l'API

Résumé :

Postman = outil de test complet

Swagger = documentation + test simple intégré au projet

3. À quoi sert HAL Explorer ?

HAL Explorer est une interface graphique pour naviguer dans les API Spring Data REST utilisant HAL.

Il permet de :

- Explorer toutes les ressources REST
- Voir les liens _links générés (self, relation, search...)
- Tester les endpoints très rapidement
- Comprendre la structure HAL générée automatiquement

Résumé :

HAL Explorer = navigateur d'API Spring Data REST.

4. Différence entre base H2 et fichier SQL manuel

Base H2 (en mémoire)

- Base de données embarquée
- Videra son contenu à chaque redémarrage
- Très utile pour les tests et le développement rapide
- Création automatique par Spring (tables + données via Java)

Fichier SQL manuel

- Script SQL placé dans `/resources` (ex: `data.sql`)
- Spring exécute ce fichier au démarrage
- Permet de créer des données fixes, toujours les mêmes
- Reproductible, même si la BD s'efface

Résumé :

H2 = base temporaire pour tester

SQL manuel = données contrôlées via scripts persistants.