devoteam | M Applications

Spring Data REST: exposez rapidement une ressource à l'aide de l'API REST

02 juin 2020

Java REST Spring Articles

Dans cet article, nous allons présenter, utiliser et expliquer les principales caractéristiques du projet *Spring Data REST* de Spring, dont l'objectif principal est de simplifier autant que possible l'exposition d'une ressource grâce à l'utilisation de l'API REST.

Pour pouvoir suivre et tirer parti du contenu de cet article, le lecteur doit avoir une connaissance préalable de base de Java, Maven et du framework Spring, ainsi que du fonctionnement de base d'une API REST .

Afin de montrer comment Spring Data REST fonctionne, cet article développera un petit exemple d'API REST.

(Vous pouvez retrouver le projet ici : https://github.com/DevoteamModernApplications/spring-data-rest-example)

Introduction à Spring Data REST

Spring Data REST est un projet – parmi tant d'autres – de Spring Framework qui vise à accélérer et simplifier l'exposition d'une ressource à l'aide d'une API REST.

Étant donné que Spring Data REST est un projet inclus dans le framework Spring, il vous suffit d'ajouter une dépendance dans le POM (le *starter spring-boot-starter-data-rest*)

devoteam | M Applications

Recherche...

```
ndencies>
dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-rest</artifactId>
</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>
    <exclusions>
        <exclusion>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
        </exclusion>
    </exclusions>
</dependency>
</dependencies>
```

En plus de la dépendance Spring Data REST, il faut ajouter des dépendances pour JPA et pour H2, puisque nous allons utiliser une base de données en mémoire pour simplifier l'application d'exemple. Finalement, nous avons aussi ajouté *Lombok*, un *toolbox* de développement Java permettant d'automatiser lors de la compilation des taches comme l'ajout des *getters* et *setters*, *constructors* et beaucoup d'avantage.

Création d'un projet avec Spring Data Rest

Après avoir installé les dépendances expliquées dans la section précédente, il nous suffit d'effectuer deux étapes supplémentaires pour disposer d'une API fonctionnelle utilisant Spring Data REST.

devoteam | M Applications

e à créer une entité: une classe *POJO* qui nous permettra de mapper les objets
ents de la base de données que nous allons utiliser comme support pour notre

```java
package edu.devoteam.geo.domain;

import lombok.Data;

import javax.persistence.*;

@Entity
@Data
public class Country {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String name;

    private Integer population;
}
```

Pour notre petit exemple d'API, nous commencerons par l'entité *Country* . L'entité possède 3 champs et pour l'implémentation des méthodes héritées d' *Object* nous utilisons l'annotation Lombok @Data, qui génère le code au moment de la compilation. Cela nous permet d'avoir un code source plus propre et de ne pas passer de temps à realiser des tâches répétitives.

La seconde étape consiste à terminer la création du *repository* Spring Data pour l'entité que nous venons de créer. Le *repository* est généré sous la forme d'une *interface* et s'étend à partir de l'une des interfaces proposées pour Spring Data (il existe plusieurs niveaux d'implémentation tels que *CrudRepository* ou *PageAndSortingRepository* qui offrent des différentes fonctionnalités telles que la pagination): dans notre cas, nous étendons à partir de *JpaRepository*

```java
package edu.devoteam.geo.repository;

import edu.devoteam.geo.domain.Country;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

//@RepositoryRestResource(collectionResourceRel = "country", path = "country")
public interface CountryRestRepository extends JpaRepository<Country, Long> {
}
```

Comme vous pouvez le voir, avec l'utilisation d'un *repository* normal, lors de l'exécution de l'application (*mvn spring-boot: run*), Spring Data REST détecte le *repository* et expose la ressource via un point de *endpoint* créé par défaut.

devoteam | M Applications

int ainsi que les différentes méthodes HTTP de l'API, Spring Data REST s'appuie
REST et du **HATEOAS** (*Hypermédia le moteur d'état de l' application*), un
e les principales directives de l'architecture REST, mais fournit également
des éléments d'information supplémentaires -les liens vers les ressources- pour faciliter l'utilisation de l'API.

Recherche...

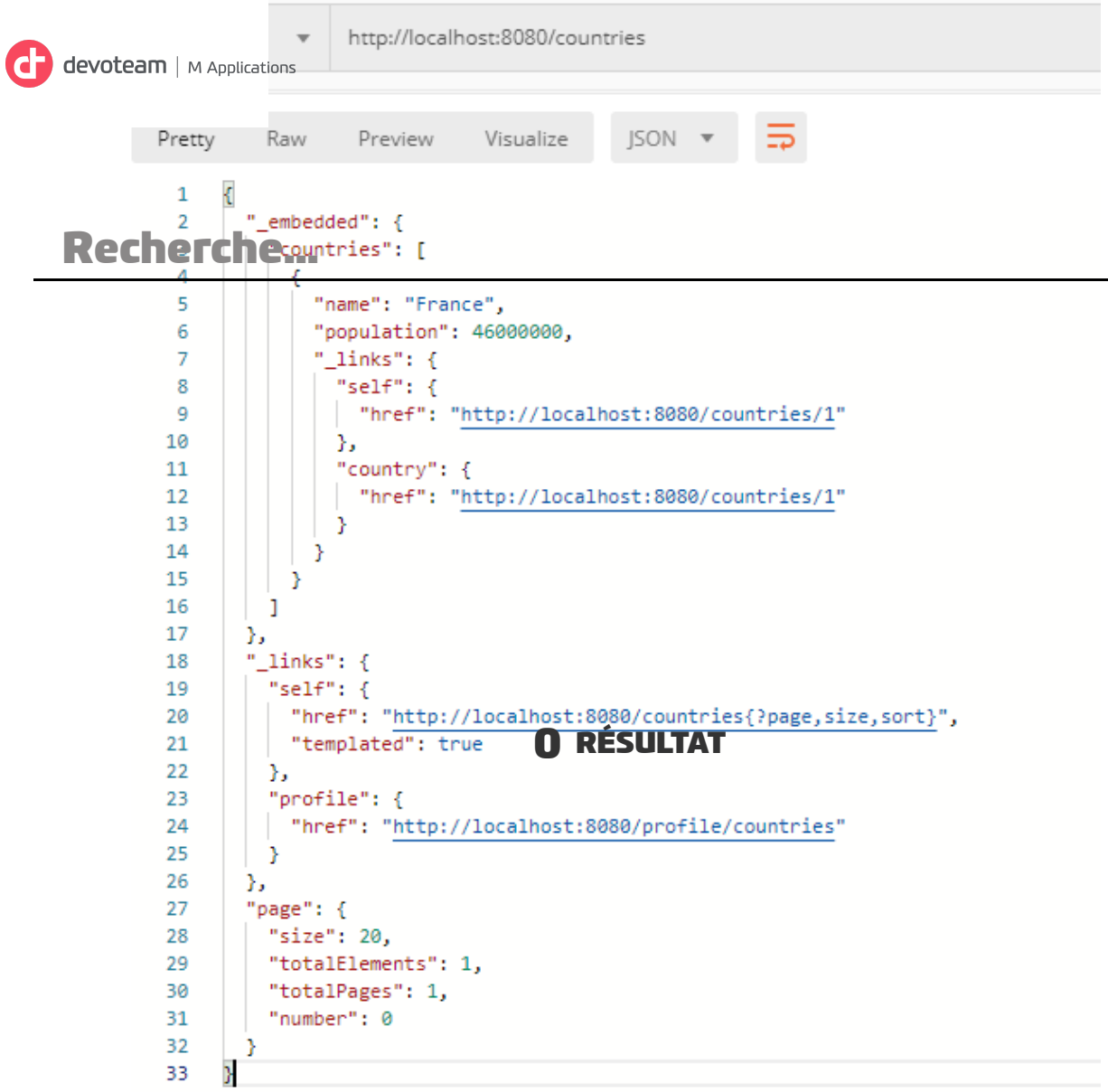
Avant de terminer cette section, il reste à expliquer l'annotation ***@RepositoryRestResource***. Le but de cette annotation est de modifier les propriétés qui sont automatiquement déduites par Spring Data REST lors de l'exposition de notre ressource: des éléments tels que le *path* à une ressource ou une collection de ressources peuvent être modifiés en utilisant les attributs de cette annotation.

Interaction avec l'API

Une fois que nous avons configuré et démarré l'application (mvn spring-boot: run), l'utilisation de l'API est guidée par les réponses de l'API et est très intuitive.

0 RÉSULTAT

- GET */ countries*

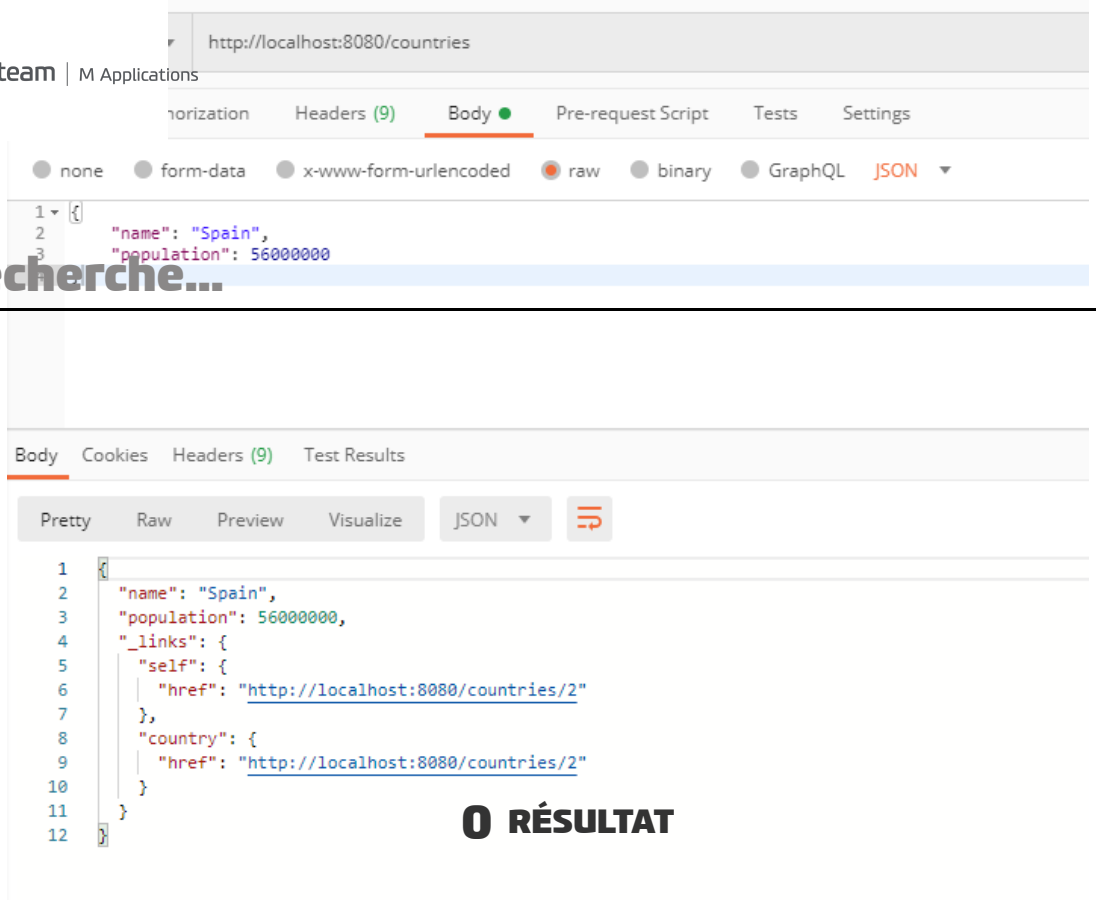



```
{
  "_embedded": {
    "countries": [
      {
        "name": "France",
        "population": 46000000,
        "_links": {
          "self": {
            "href": "http://localhost:8080/countries/1"
          },
          "country": {
            "href": "http://localhost:8080/countries/1"
          }
        }
      }
    ]
  },
  "_links": {
    "self": {
      "href": "http://localhost:8080/countries{?page,size,sort}",
      "templated": true
    },
    "profile": {
      "href": "http://localhost:8080/profile/countries"
    }
  },
  "page": {
    "size": 20,
    "totalElements": 1,
    "totalPages": 1,
    "number": 0
  }
}
```


- POST / *countries*

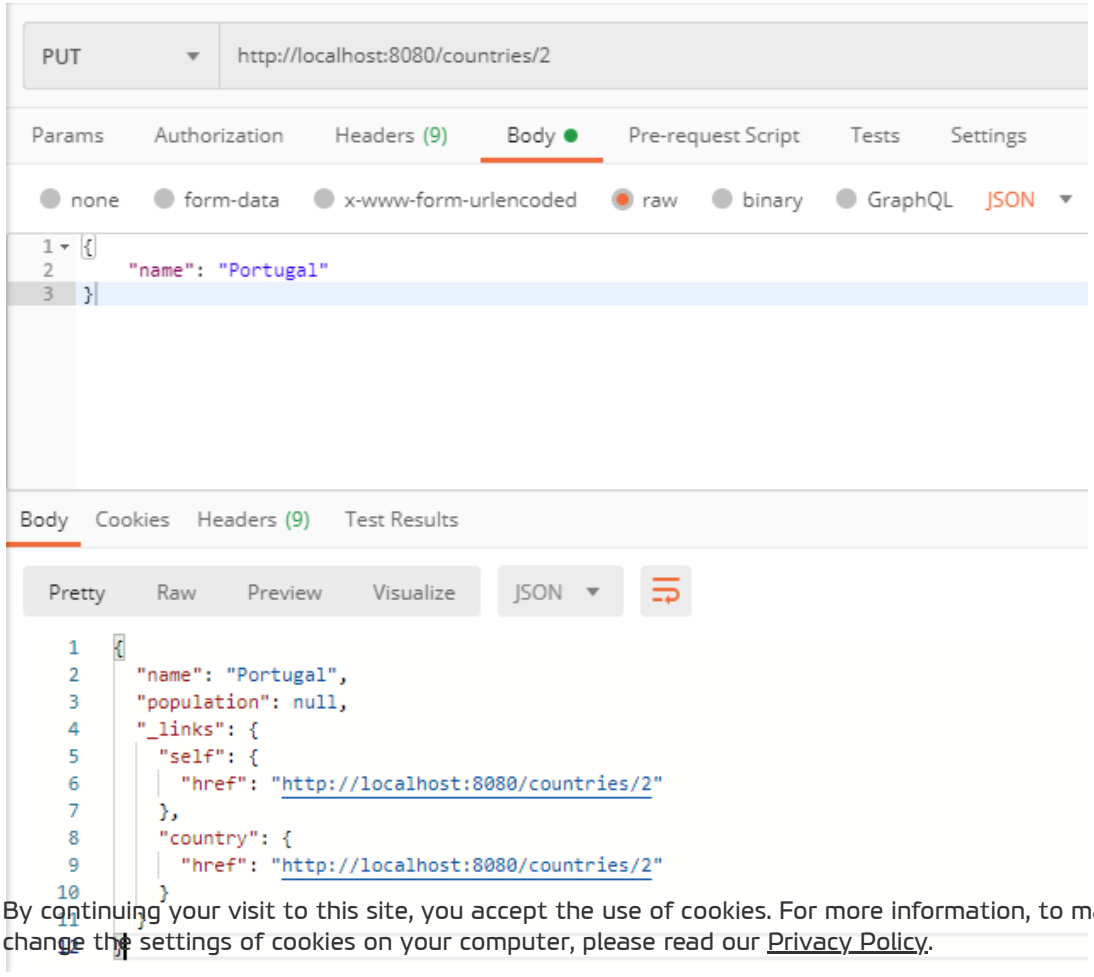
devoteam | M Applications

Recherche...



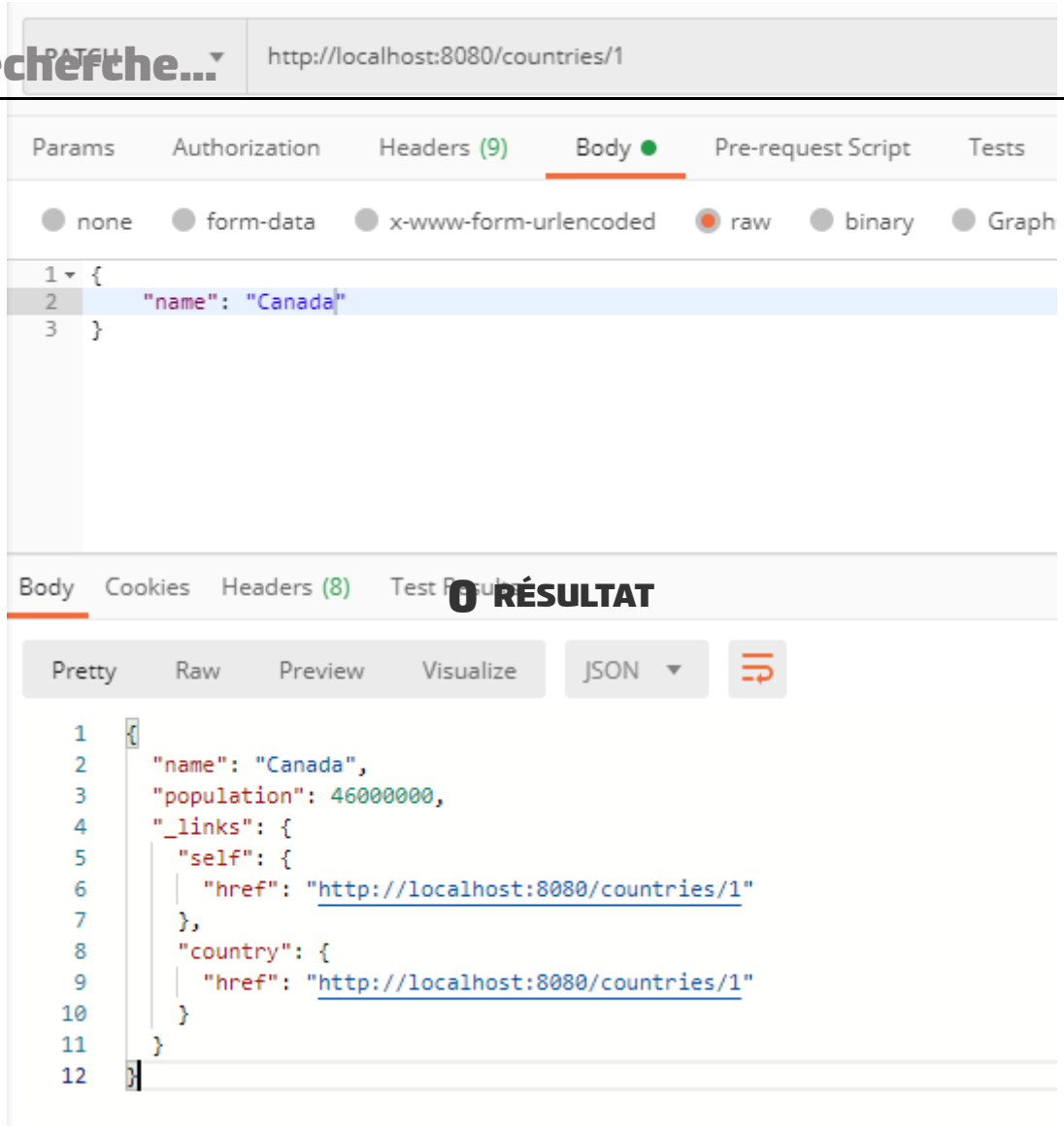
0 RÉSULTAT

- PUT / *countries* / ID



devoteam | M Applications

- PATCH / countries / *ID*



Comme vous pouvez le voir, l'utilisation de l'API est assez simple et intuitive. De plus, les informations HATEOAS fournies par Spring Data REST vous permettent de «cibler» et d'automatiser l'utilisation de l'API par les clients.

Dans ce chapitre, il est nécessaire de souligner qu'il existe le chemin */profile* (et ses chemins dérivés) qui fournissent des informations sur les points de terminaison. Ce n'est pas un élément de documentation , comme pourrait l'être *Swagger* , mais il peut servir à guider les développeurs qui mettent en œuvre des solutions avec Spring Data REST.

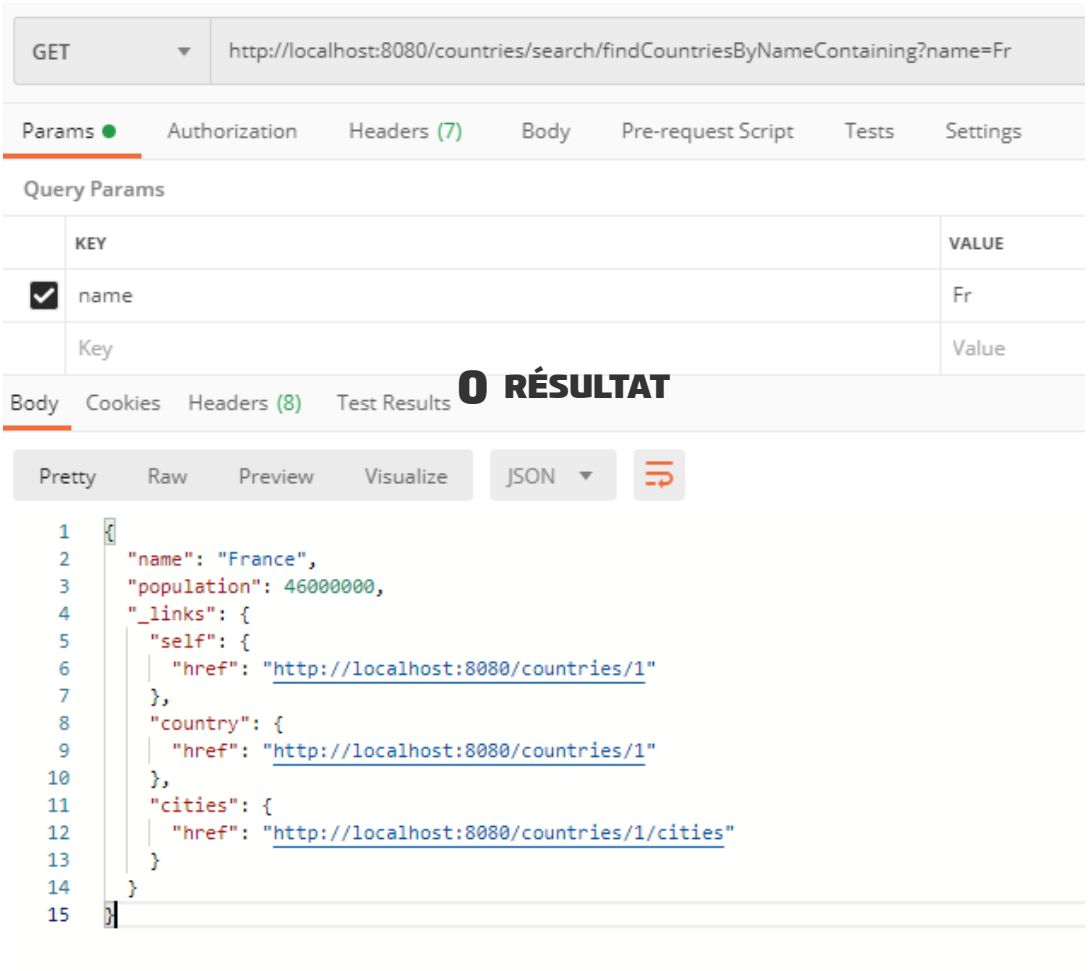
Finalement, nous allons voir qu'il est aussi possible d'appeler des méthodes de recherche crées dans les *repository* pour réaliser des recherches dans la base de données. Pour invoquer ces méthodes, il faut ajouter au path */countries/search*, le nom de la méthode invoquée : dans l'exemple, *findCountriesByNameContaining* suivi du paramètre *name* (Notez bien que dans ce cas le *name* est passé comme paramètre de la requête HTTP)

```
RestResource(collectionResourceRel = "country", path = "country")
face CountryRestRepository extends JpaRepository<Country, Long> {

    //@RestResource(path = "name")
    Optional<Country> findCountriesByNameContaining(String name);
}
```

Recherche...

Le résultat de faire l'appel comment décrit précédemment est le suivant.



Notez bien que l'annotation @RestResource peut être utiliser pour changer le *path* à utiliser pour faire appel à la méthode,

Ajout de logique à l'API

Une fois que nous avons réussi à démarrer l'application et à effectuer les premiers appels d'API pour récupérer, insérer ou mettre à jour des objets *Country* , il est temps de voir comment nous pouvons ajouter une logique métier à notre API.

devoteam | M Applications

ıs Spring Data REST, la logique est ajoutée à travers d' *événements* , qui à leur la programmation orientée aspect (AOP en anglais). L'événement central autour s événements sont créés est la transaction avec la base de données.

Pour gérer ces événements, vous devez déclarer un composant Spring et l'annoter avec l'annotation @RepositoryEventHandler. Notez que vous pouvez déclarer un *bean* à l''aide d'annotations (de manière déclarative) ou en créant le bean dans une méthode d'une classe de configuration (annotée avec @Configuration) et annoter la méthode avec @Bean.

Les événements disponibles sont les suivants:

- ***CreateEvent*** : événement qui implique l'action de créer un nouvel enregistrement dans la base de données (équivalent à la méthode HTTP POST).
- ***SaveEvent***: événement qui implique l'action de modifier un enregistrement existant dans la base de données. En général, cet événement est lancé en cas de mise à jour des informations (en utilisant les méthodes PUT et PATCH de HTTP)
- ***DeleteEvent****:* événement qui, comme son nom l'indique, implique l'action de supprimer un enregistrement de la base de données (équivalent aux appels REST avec la méthode DELETE).
- ***LinkSaveEvent****:* événement qui implique l' action de modifier un objet lié. Il est généralement utilisé pour les relations d'objet.

Pour chaque événement expliqué ci-dessus, vous pouvez ajouter de la logique avant et après l' exécution (*before* et *after*) de différentes manières.

Après avoir vu des différents points et événements que nous pouvons écouter et où nous pouvons intervenir, comme on peut le voir en tant que gestionnaire applique à notre projet.

devoteam | M Applications

Recherche...

```
ponent
ositoryEventHandler
4j
public class CountryEventHandler {

    @HandleBeforeCreate
    public void handleCountryBeforeCreate(Country country){
        log.info("Intercept Country Before Create it....");
        log.info(country.toString());
    }

    @HandleAfterCreate
    public void handleCountryAfterCreate(Country country){
        log.info("Intercept Country After Create it....");
        log.info(country.toString());
    }
```

Dans cette image, vous pouvez voir la déclaration de la classe et des méthodes pour exécuter la logique avant et après la création de la ressource.

```
    @HandleBeforeSave
    public void handleCountryBeforeSave(Country country){
        log.info("Intercept Country Before Save it....");
        log.info(country.toString());
    }

    @HandleAfterSave
    public void handleCountryAfterSave(Country country){
        log.info("Intercept Country After Save it....");
        log.info(country.toString());
    }
```

Dans cette image, nous pouvons voir les méthodes du gestionnaire pour intercepter la modification des informations ou des ressources existant auparavant dans l'application.

```
    @HandleBeforeDelete
    public void handleAuthorBeforeDelete(Country country){
        log.info("Intercept Country Before Delete it ....");
        log.info(country.toString());
    }

    @HandleAfterDelete
    public void handleAuthorAfterDelete(Country country){
        log.info("Inside Author After Delete it ....");
        log.info(country.toString());
    }
```

devoteam | M Applications

Enfin, dans cette image se trouvent les *gestionnaires* qui permettent d'intercepter et d'ajouter de la logique avant et après la suppression d'une ressource, p ou support r une requête SUPPRIMER.

Recherche...

Ajouter des validations aux entités

Pour ajouter des validations aux entités à créer ou à mettre à jour, l'approche la plus simple consiste à utiliser les validations que la spécification JSR 380 met à disposition pour vérifier les cas les plus courants.

Pour commencer, il faut ajouter les annotations de validation dans les champs que nous voulons valider. Les annotations par défaut se trouvent dans le package *javax.validation.constrains.*

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

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @NotEmpty
    private String name;

    @Positive
    private Integer population;
}
```

Bien que cela n'entre pas dans le cadre de cet article, nous pouvons également créer nos validateurs personnalisés en implémentant l'interface *Validator* (ou d'autres interfaces plus spécifiques) de l'API Spring (il est important de ne pas la confondre avec l'interface Java Validator).

Ensuite, nous devons déclarer l'utilisation de validateurs dans la configuration Spring Data REST. Pour ce faire, nous devons créer une classe, appliquer l'annotation @Configuration de Spring et étendre *RepositoryRestConfigurer.*

Ensuite, nous devons implémenter la méthode configureValidatingRepositoryEventListener pour ajouter l'interface Spring *Validator* que, précédemment, nous aurons injectée dans la classe, pour tous les points que nous voulons (normalement, *beforeCreate et beforeSave*).

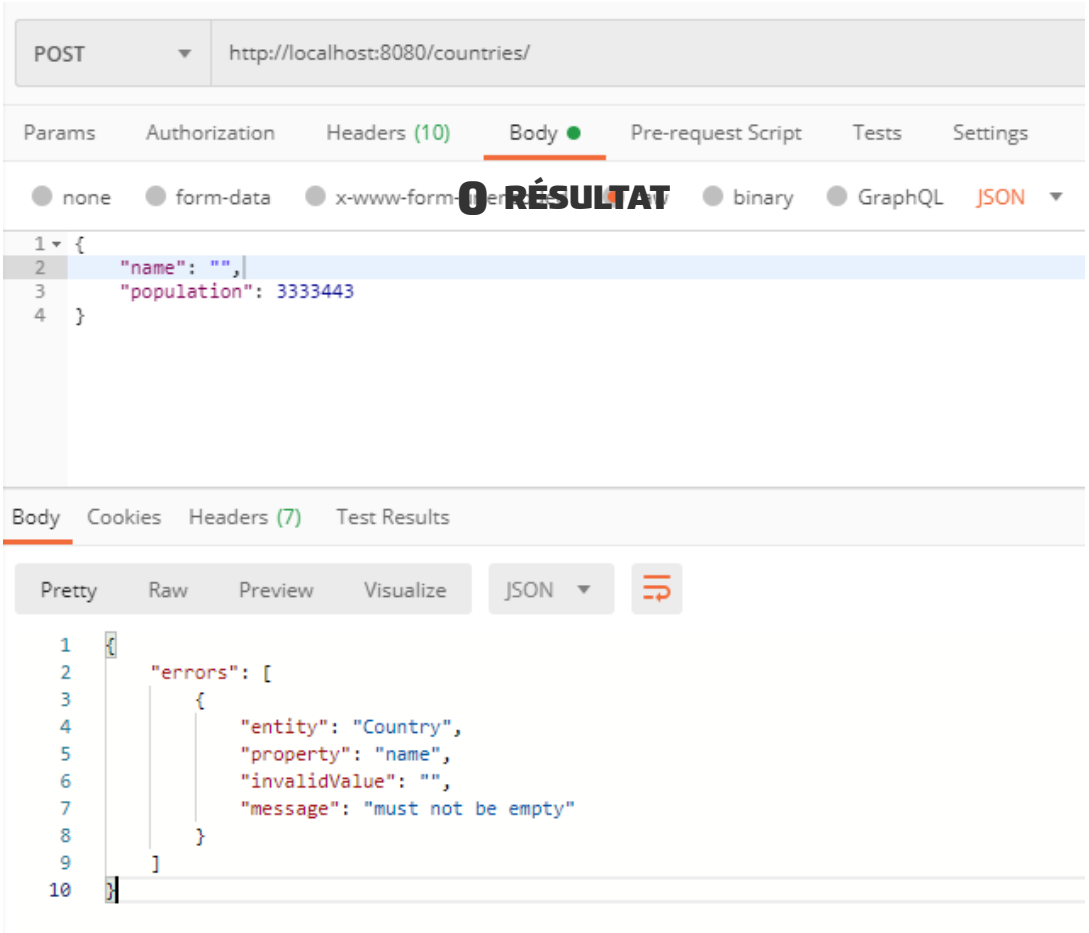
```
or
RestConfiguration implements RepositoryRestConfigurer {

ator validator;

    @Override
    public void configureValidatingRepositoryEventListener(ValidatingRepositoryEventListener validatingListener) {
        validatingListener.addValidator( event: "beforeCreate", this.validator);
        validatingListener.addValidator( event: "beforeSave", this.validator);
    }

}
```

Si nous essayons maintenant de faire une demande POST avec le nom du pays vide, cela devrait retourner une erreur.



La langue du message d'erreur, lorsqu'il s'agit du message par défaut, change en fonction des paramètres régionaux envoyés dans l'en-*tête Accept-language*.

Ajout de relations entre des entités à l'API

La dernière étape de ce guide est de travailler avec les relations entre les entités. Il existe plusieurs façons de lier deux entités de base de données à l'aide de JPA.

devoteam | M Applications

but de cet article n'est pas d'expliquer en détail chaque type de relation et, ; REST s'intègre très bien avec les relations dans JPA, dans cet article, nous ...iple avec une relation bidirectionnelle @ManyToOne – @OneToMany .

Pour commencer, nous avons créé une entité: dans notre exemple, nous avons créé l'entité City.

Recherche...

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

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String name;

    private Boolean isCapital = Boolean.FALSE;

    @ManyToOne
    @JoinColumn(name="country_id")
    private Country country;
}
```

0 RÉSULTAT

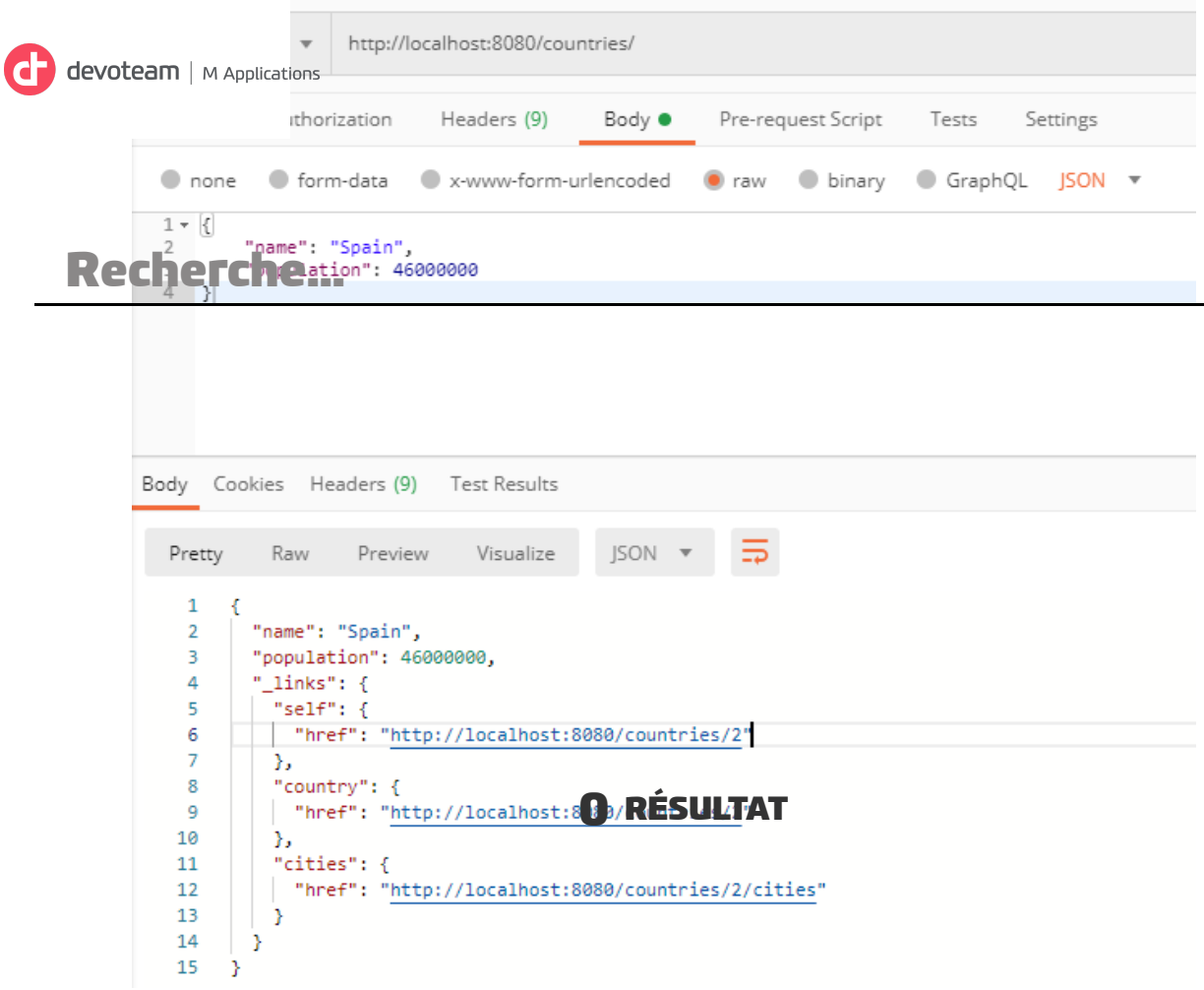
Nous créons une **relation 1 à N** entre le pays (Country) et la nouvelle entité City. Dans cet exemple, la relation bidirectionnelle sera pilotée par l'entité Pays.

```
@OneToMany
private List<City> cities;
```

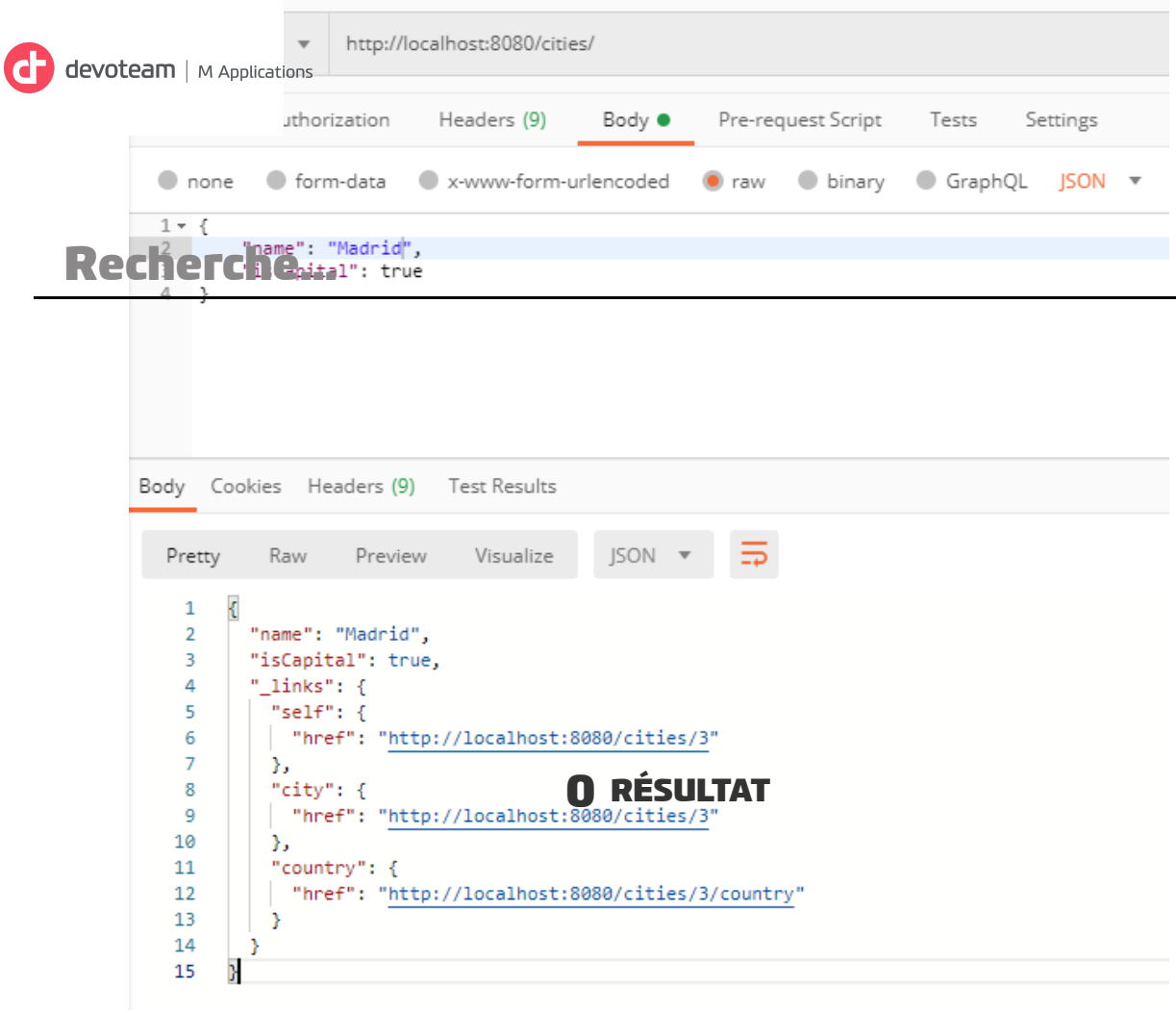
Ensuite, pour activer Spring données REST sur l'entité nouvellement créée, nous devons créer un *repository* JPA (comme nous l' avons fait pour Country et d' étendre *JpaRepository*).

Avant de tester l'association, il est à noter que, comme dans la définition des *repository* REST, il est également possible de modifier le chemin de la relation ou le nom de la relation en utilisant l'annotation @RestResource et ses attributs.

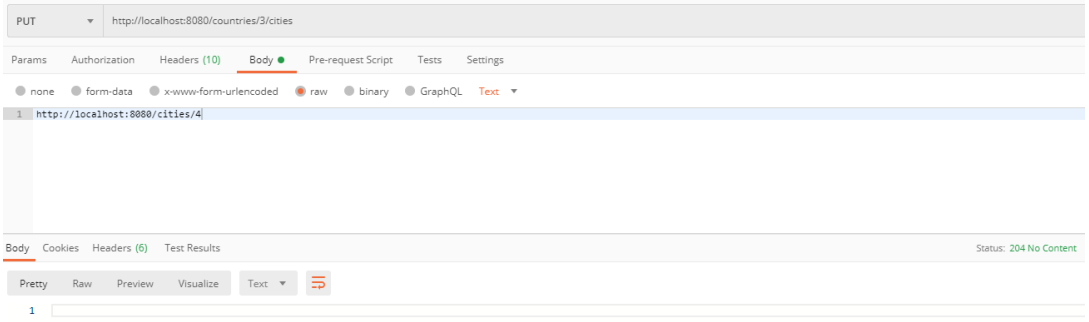
Pour tester notre association, nous devons d'abord créer un objet de l'entité Country.



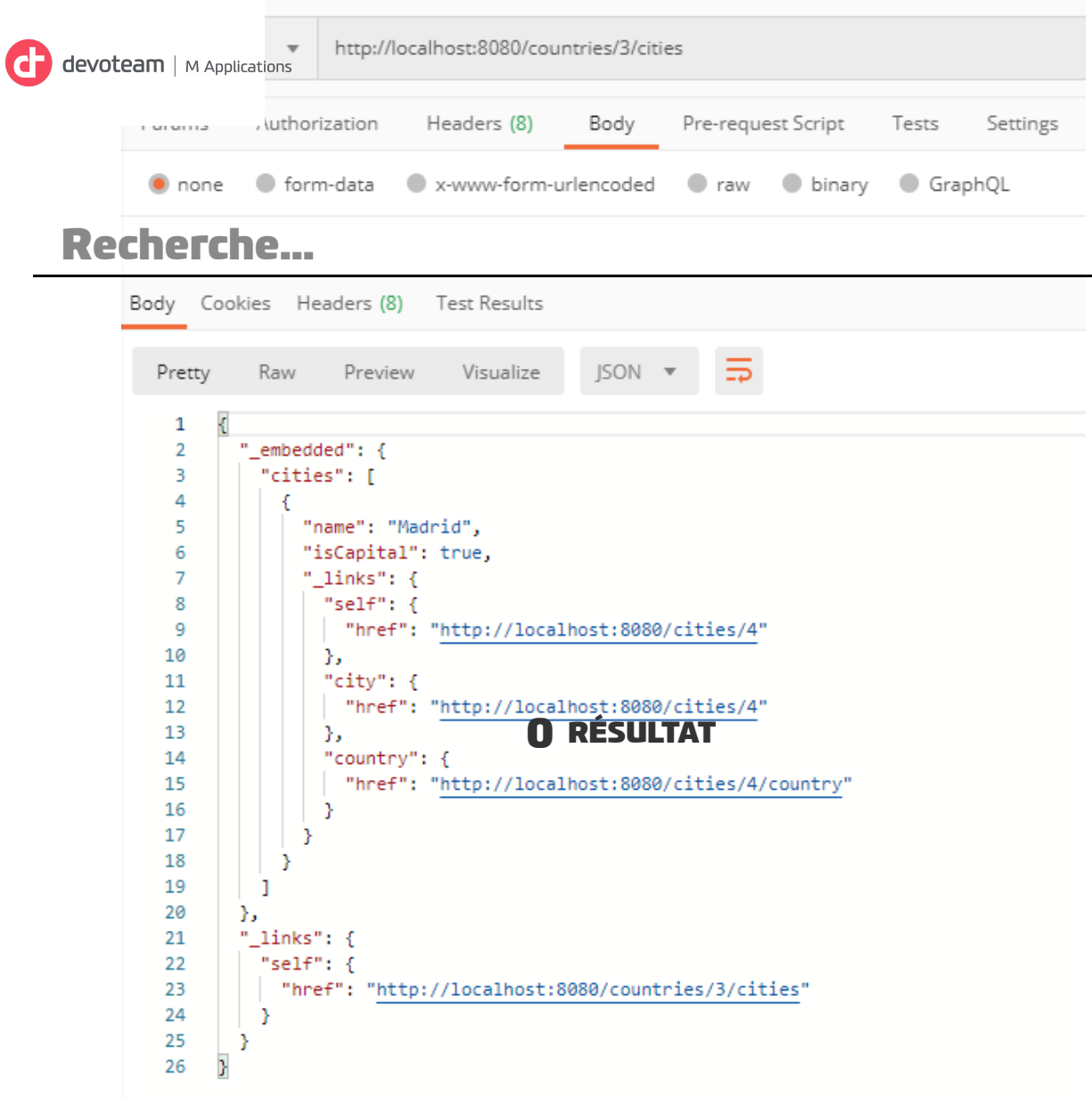
Ensuite, vous devez créer un objet de l'entité City. Il est important de le créer d'abord, puis de créer la relation entre les objets.



Lorsque nous avons créé les deux objets, il est temps de les relier. Pour cela, nous allons faire une requête PUT avec un Content-Type: "text / uri-list" dans lequel nous spécifierons le chemin de la relation (/countries/{id}/cities) et, dans le corps de la requête , l'url de l'objet que nous voulons mettre à jour.



Si la relation réussit, nous devrions obtenir un code de réponse 204 – Pas de contenu. Si nous allons maintenant vérifier les *cities* répertoriées dans le *country* que nous avons créé, nous devrions voir qu'une première ville a été ajoutée.



Comme vous pouvez le voir, l'intégration avec les relations JPA est automatique avec Spring Data REST: la seule chose importante à retenir est que **vous devez d'abord créer les objets séparément puis les joindre à l'aide d'une demande PUT**.

Conclusions

Comme nous l'avons vu tout au long de cet article, le projet Spring Data REST est une option très intéressante pour créer rapidement des applications serveur avec un certain niveau de logique associée et de connexion à la base de données pour la persistance des informations. Ensuite, nous allons voir les principaux avantages de Spring Data REST.

devoteam | M Applications

es exposées par défaut . L'implémentation Spring Data REST permet automatiquement les ressources (ou entités) à l'aide d'une API utomatiquement. Il n'est pas nécessaire de créer la couche contrôleur que nous aurions créée avec une application conventionnelle et la couche service (où nous ajoutons la logique) est considérablement réduite.

Recherche...

- **Abstraction de la couche d'accès aux données.** L'utilisation des packages Spring Data vous permet d'abstraire l'application de la couche d'accès aux données. En d'autres termes: implémenté Spring Data JPA, l'application fonctionnera avec n'importe quelle base de données relationnelle, sans avoir besoin de changer le code (PostgreerSQL, MySQL, MariaDB, Oracle SQL, etc.). De plus, les changements à faire pour travailler avec d'autres technologies de stockage d'informations (bases de données non relationnelles ou systèmes de mise en cache) sont considérablement réduits.
- ***Découverte automatique* et documentation.** Par défaut, l'API créée avec Spring Data REST implémente les directives HATEOAS: cela signifie que, avec les ressources et les collections mises à disposition, l'application attache les liens vers les ressources créées ou disponibles, permettant aux clients de mieux utiliser API simple et automatisée. De plus, bien que nous ne l'ayons pas vu dans cet article, il est possible d'ajouter Swagger à l'application pour obtenir, de manière simplifiée et par le biais d'annotations, une documentation détaillée des endpoints exposés.

0 RÉSULTAT

- **Intégration avec d'autres packages Spring.** L'utilisation de Spring Data REST est parfaitement intégrée aux autres packages Spring Framework. Avec Spring Web, pour l' intégration avec des applications Web conventionnelles, avec Spring Security, pour la protection des itinéraires, etc.

Pour ces raisons et bien d'autres, l'utilisation de Spring Data REST est un excellent outil pour la création rapide d'applications exposées via des API REST, par exemple, la création de micro-services , la réalisation de *POC* (*Proof Of Concept*) ou Même la création de domaines (lorsque nous travaillons dans une application basée sur un domaine) dans lesquels une couche logique importante n'est pas nécessaire.

Ricard

devoteam | M Applications

NEWS SUIVANTE

Native Script

JS Native Script Articles

M-applications

Conditions d'utilisation
Legal Notice
Personal Data

M-APPLICATIONS

73 rue Anatole France
92300 Levallois Perret
France
+33 1 41 49 48 48

SUIVEZ-NOUS

Discover all information about Devoteam group here:

Devoteam Group