

Cours Inf3522 - Développement d'Applications N tiers

Lab_4 : Création de services web RESTful avec Spring Boot

Dans ce lab, nous allons d'abord créer un service web RESTful. Ensuite, nous allons démontrer comment utiliser Spring Data REST pour créer un service web RESTful qui fournit également toutes les fonctionnalités CRUD automatiquement. Nous utiliserons l'application de base de données que nous avons créée dans le lab précédent comme point de départ.

Dans ce lab, il est permis d'utiliser des IDEs comme IntelliJ IDEA ou Eclipse (y installer Spring Tool Suite : aller dans help puis eclipse marketplace puis rechercher spring tool suite et l'installer).

Exercice 1

Les services web sont des applications qui communiquent sur Internet en utilisant le protocole HTTP. Il existe de nombreux types d'architectures de services web, mais l'idée principale derrière toutes les conceptions est la même.

Les bases d'un service web RESTful

Representational State Transfer (REST) est un style architectural permettant de créer des services web. **REST** n'est pas une norme, mais définit un ensemble de contraintes établies par *Roy Fielding*. Les six contraintes sont les suivantes :

- **Stateless** : Le serveur ne stocke aucune information sur l'état du client.
- **Client-serveur** : Le client et le serveur agissent de manière indépendante. Le serveur ne renvoie pas d'informations sans demande du client.
- **Cacheable** : De nombreux clients demandent souvent les mêmes ressources ; il est donc utile de stocker en cache les réponses pour améliorer les performances.
- **Interface uniforme** : Les requêtes provenant de différents clients ont la même apparence. Les clients peuvent inclure, par exemple, un navigateur, une application Java et une application mobile.
- **Système en couches** : REST nous permet d'utiliser une architecture en couches.
- **Code à la demande** : Il s'agit d'une contrainte optionnelle.

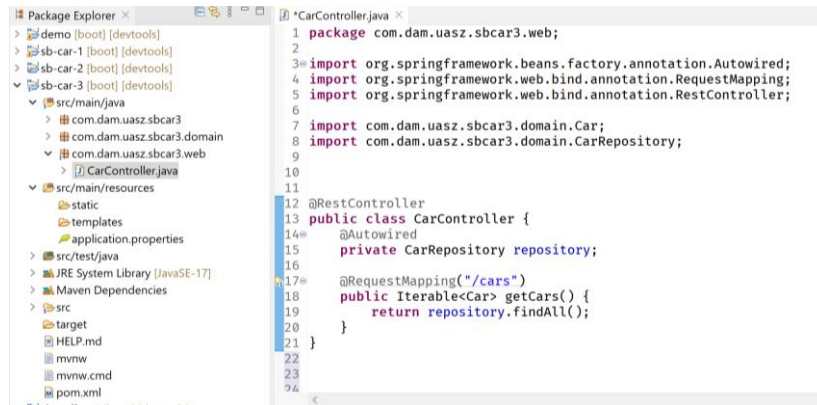
L'**interface uniforme** est une contrainte importante, ce qui signifie que chaque architecture **REST** devrait comporter les éléments suivants :

- **Identification des ressources** : Il y a des ressources avec des identifiants uniques, par exemple, des **URIs** dans les services **REST** basés sur le web. Les **ressources REST** doivent exposer des **URIs** de structure de répertoire facilement compréhensibles. Par conséquent, une bonne stratégie de nommage des ressources est très importante.
- **Manipulation des ressources par représentation** : Lorsque vous faites une demande à une ressource, le serveur répond avec une représentation de la ressource. En général, le format de la représentation est **JSON** ou **XML**.
- **Messages auto-descriptifs** : Les messages doivent contenir suffisamment d'informations pour que le serveur sache comment les traiter.
- **L'hypermédia comme moteur de l'état de l'application (HATEOAS)** : Les réponses peuvent contenir des liens vers d'autres zones du service.

Le **service web RESTful** que nous allons développer dans ce lab suit les principes architecturaux **REST** décrits ci-dessus.

Création d'un service web RESTful avec Spring Boot

Il faut créer un nouveau package **com.dam.uaszbcar3.web**, et à l'intérieur nous aurons une nouvelle classe **CarController** définie comme ci-dessous.



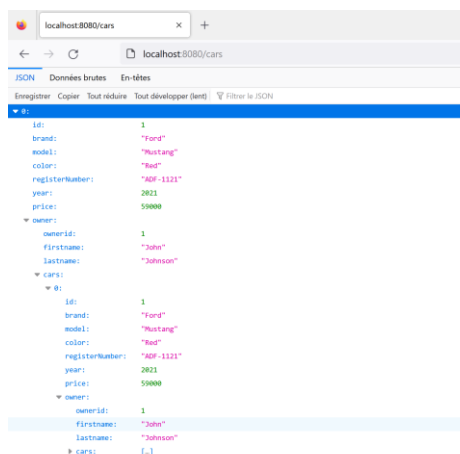
L'annotation **@RestController** identifie cette classe comme étant le contrôleur du service web RESTful !!! La méthode est annotée avec l'annotation **@RequestMapping**, qui définit le point de terminaison auquel la méthode est associée. Lorsque l'utilisateur accède au point de terminaison **/cars**, la méthode **getCars()** est exécutée.

La méthode **getCars()** renvoie tous les objets de voiture, qui sont ensuite convertis en objets JSON par la bibliothèque Jackson (<https://github.com/FasterXML/jackson>).

Par défaut, **@RequestMapping** gère toutes les méthodes HTTP (GET, PUT, POST, etc.) requises. Vous pouvez définir la méthode acceptée en utilisant le paramètre suivant : **@RequestMapping(value="/cars", method=GET)**. Maintenant, cette méthode ne gère que les demandes GET provenant du point de terminaison **/cars**. Vous pouvez également utiliser l'annotation **@GetMapping** à la place, et seules les demandes GET sont alors associées à la méthode **getCars()**. Il existe d'autres annotations pour les différentes méthodes HTTP, telles que **@PostMapping**, **@DeleteMapping**, etc.

Pour pouvoir renvoyer les voitures à partir de la base de données, nous devons injecter **CarRepository** dans le contrôleur. Ensuite, nous pouvons utiliser la méthode **findAll()** que le référentiel fournit pour récupérer toutes les voitures. Grâce à l'annotation **@RestController**, les données sont maintenant **sérialisées au format JSON** dans la réponse.

Maintenant, nous sommes prêts à exécuter notre application et à naviguer vers **localhost:8080/cars**.



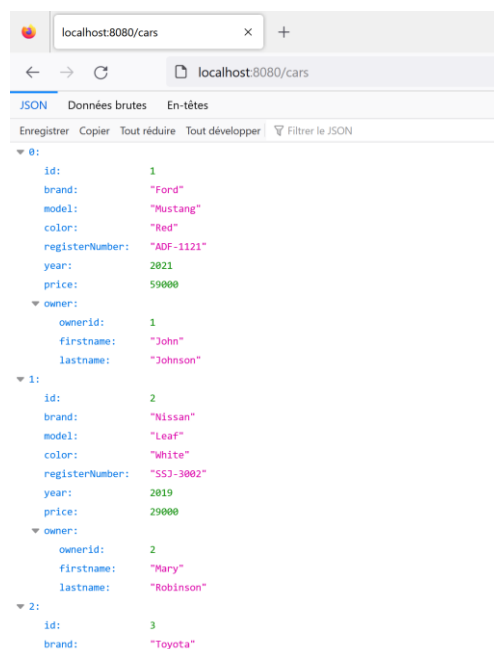
Nous pouvons voir qu'il y a quelque chose qui ne va pas, et que l'application semble être dans une boucle infinie. Cela se produit en raison de notre relation un-à-plusieurs entre les tables de voiture et de propriétaire. Que se passe-t-il en pratique ? Tout d'abord, la voiture est sérialisée, et elle contient un propriétaire qui est ensuite sérialisé, et cela, à son tour, contient des voitures qui sont alors sérialisées, et ainsi de suite. Pour éviter cela, nous pouvons utiliser différentes solutions. Une façon est d'utiliser l'annotation **@JsonIgnore** pour le champ **cars** dans la classe **Owner**, qui ignore le champ de voitures dans le processus de sérialisation. Nous utiliserons également l'annotation **@JsonIgnoreProperties** pour ignorer les champs générés par Hibernate (lignes 11, 12, 14, 30) :

```

CarController.java  *Owner.java x
6 import jakarta.persistence.Id;
7 import jakarta.persistence.OneToMany;
8 import java.util.List;
9 import jakarta.persistence.CascadeType;
10
11 import com.fasterxml.jackson.annotation.JsonIgnore;
12 import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
13
14 @JsonIgnoreProperties({"hibernateLazyInitializer","handler"})
15 @Entity
16 public class Owner {
17     private long ownerId;
18     private String firstname, lastname;
19
20     public Owner() {}
21
22     public Owner(String firstname, String lastname) {}
23
24     @JsonIgnore
25     @OneToMany(cascade=CascadeType.ALL, mappedBy="owner")
26     private List<Car> cars;
27
28     public List<Car> getCars() {
29         return cars;
30     }
31 }

```

Maintenant, lorsque vous exécutez l'application et naviguez vers **localhost:8080/cars**, tout devrait se dérouler comme prévu et vous obtiendrez toutes les voitures de la base de données au format JSON, comme indiqué dans la capture d'écran suivante :



Nous avons écrit notre premier service web RESTful, qui renvoie toutes les voitures. Spring Boot propose une manière beaucoup plus puissante de créer des services web RESTful et nous allons l'explorer dans ce qui suit. Vous pouvez supprimer la classe CarController pour la suite.

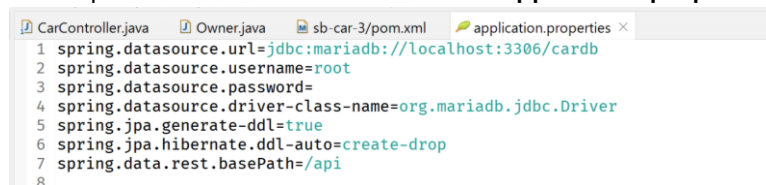
Utilisation de Spring Data REST

Spring Data REST (<https://spring.io/projects/spring-data-rest>) fait partie du projet **Spring Data**. Il offre une manière facile et rapide de mettre en œuvre des **services web RESTful avec Spring**. Pour commencer à utiliser Spring Data REST, vous devez ajouter la dépendance suivante au fichier **pom.xml** :

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

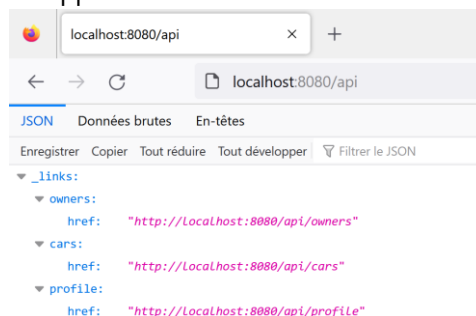
Par défaut, Spring Data REST trouve tous les répertoires publics de l'application et crée automatiquement des services web RESTful pour vos entités. Dans notre cas, nous avons deux répertoires : **CarRepository** et **OwnerRepository**, donc *Spring Data REST crée automatiquement des services web RESTful pour ces répertoires*.

Vous pouvez définir l'endpoint du service dans votre fichier **application.properties** : (ligne 7)



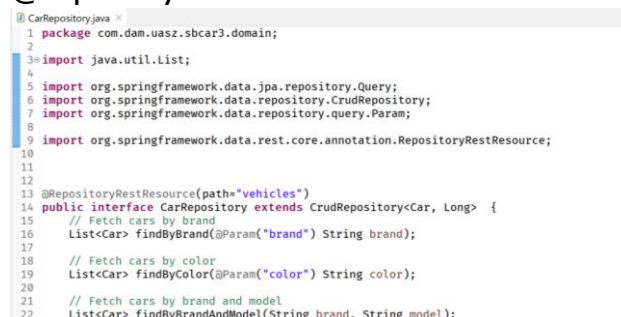
```
1 spring.datasource.url=jdbc:mariadb://localhost:3306/cardb
2 spring.datasource.username=root
3 spring.datasource.password=
4 spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
5 spring.jpa.generate-ddl=true
6 spring.jpa.hibernate.ddl-auto=create-drop
7 spring.data.rest.basePath=/api
8
```

Maintenant, vous pouvez accéder au service web RESTful depuis l'endpoint **localhost:8080/api**. En appelant l'endpoint racine du service, il renvoie les ressources disponibles. Spring Data REST renvoie des données JSON au format **Hypertext Application Language (HAL)**. Le format **HAL** fournit un ensemble de conventions pour exprimer des hyperliens en **JSON** et rend votre service web RESTful plus facile à utiliser pour les développeurs frontend :

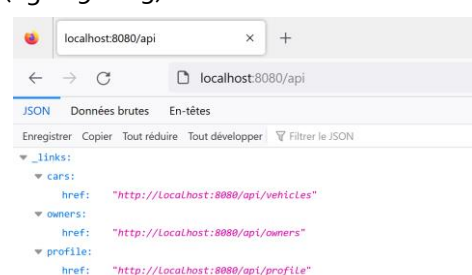


```
{
  "_links": {
    "owners": {
      "href": "http://localhost:8080/api/owners"
    },
    "cars": {
      "href": "http://localhost:8080/api/cars"
    },
    "profile": {
      "href": "http://localhost:8080/api/profile"
    }
  }
}
```

Nous pouvons voir qu'il y a des liens vers les services d'entités **Car** et **Owner**. Le nom du chemin de service Spring Data REST est dérivé du nom de la classe d'entité. Le nom sera ensuite pluriel et non capitalisé. Par exemple, le nom du chemin du service d'entité **Car** sera **cars**. Le lien de **profile** est généré par Spring Data REST et contient des métadonnées spécifiques à l'application. Si vous souhaitez utiliser un nom de chemin différent, vous pouvez utiliser l'annotation **@RepositoryRestResource** dans votre classe de répertoire (ligne 9 et 13).



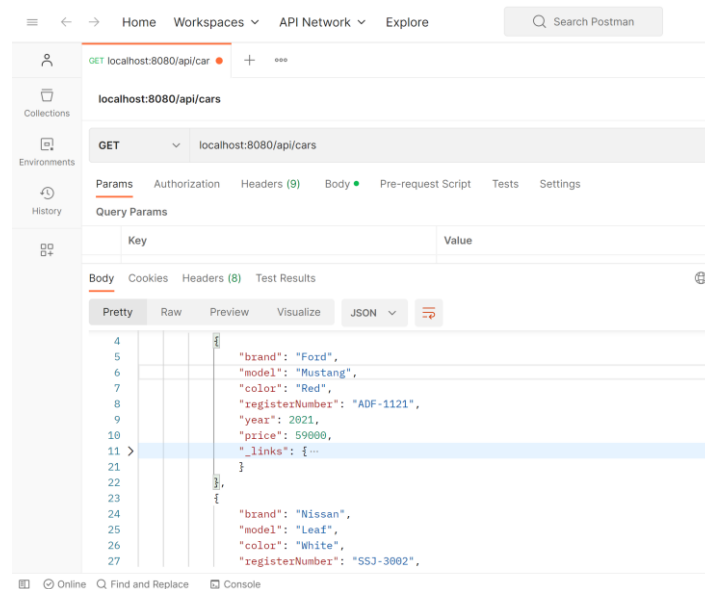
```
1 package com.dam.usz.sbcar3.domain;
2
3 import java.util.List;
4
5 import org.springframework.data.jpa.repository.Query;
6 import org.springframework.data.repository.CrudRepository;
7 import org.springframework.data.repository.query.Param;
8
9 import org.springframework.data.rest.core.annotation.RepositoryRestResource;
10
11
12
13 @RepositoryRestResource(path="vehicles")
14 public interface CarRepository extends CrudRepository<Car, Long> {
15     // Fetch cars by brand
16     List<Car> findByBrand(@Param("brand") String brand);
17
18     // Fetch cars by color
19     List<Car> findByColor(@Param("color") String color);
20
21     // Fetch cars by brand and model
22     List<Car> findByBrandAndModel(String brand, String model);
23 }
```



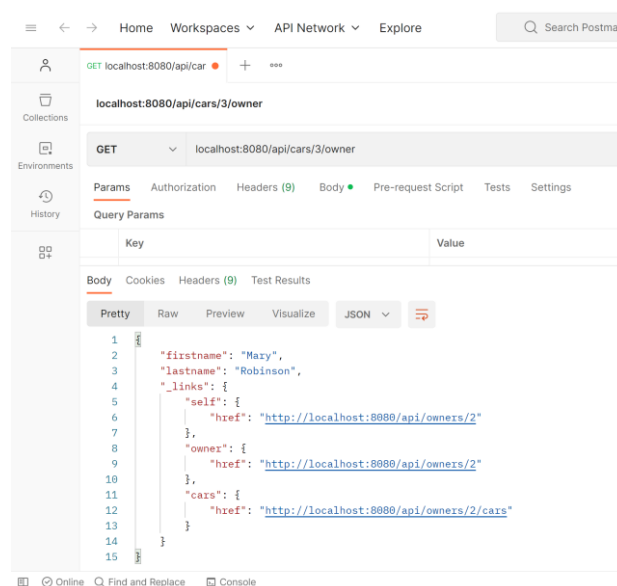
```
{
  "_links": {
    "cars": {
      "href": "http://localhost:8080/api/vehicles"
    },
    "owners": {
      "href": "http://localhost:8080/api/owners"
    },
    "profile": {
      "href": "http://localhost:8080/api/profile"
    }
  }
}
```

Vous pouvez supprimer l'annotation précédente et nous continuerons avec le nom d'endpoint par défaut, **/cars**. Maintenant, nous allons commencer à examiner plus attentivement différents services en utilisant **Httpie**, **Curl** ou **Postman**. Pour ce lab, nous allons utiliser **Postman**.

Si vous faites une demande à l'endpoint **/cars**, **http://localhost:8080/api/cars**, en utilisant la méthode **GET** (notez que vous pouvez utiliser un navigateur web pour les requêtes **GET**), vous obtiendrez une liste de toutes les voitures, comme indiqué dans la capture d'écran suivante :



Dans la réponse **JSON**, vous pouvez voir qu'il y a un tableau de voitures et que chaque voiture contient des données spécifiques à la voiture. Toutes les voitures ont également l'attribut « **_links** », qui est une collection de liens, et avec ces liens, vous pouvez accéder à la voiture elle-même ou obtenir le propriétaire de la voiture. Pour accéder à une voiture spécifique, le chemin sera **http://localhost:8080/api/cars/{id}**. La requête **GET** à **http://localhost:8080/api/cars/3/owner** retourne le propriétaire de la voiture avec l'identifiant 3. La réponse contient maintenant les données du propriétaire, un lien vers le propriétaire et des liens vers les autres voitures du propriétaire :

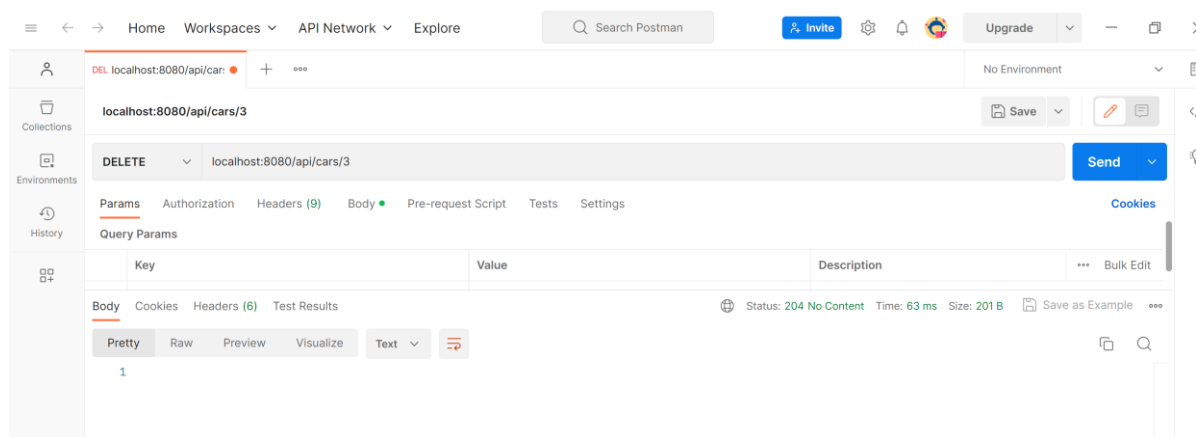


Le service **Spring Data REST** fournit toutes les opérations **CRUD**. Le tableau suivant montre les méthodes HTTP que vous pouvez utiliser pour différentes opérations **CRUD** :

HTTP Method	CRUD
GET	Read
POST	Create
PUT/PATCH	Update
DELETE	Delete

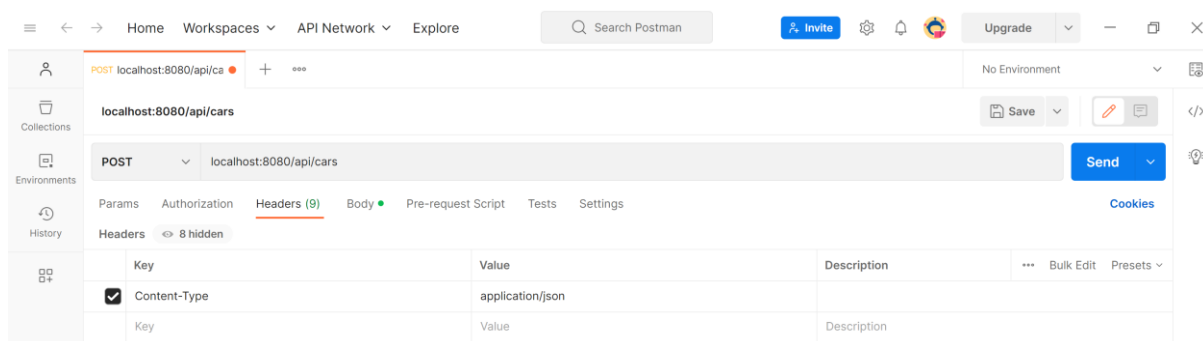
Ensuite, nous verrons comment supprimer une voiture de la base de données en utilisant notre service web RESTful. Dans une opération de suppression, vous devez utiliser la méthode **DELETE** et le lien vers la voiture qui sera supprimée (<http://localhost:8080/api/cars/{id}>).

La capture d'écran suivante montre comment vous pouvez supprimer une voiture avec l'identifiant 3 en utilisant l'application de bureau Postman (notez que vous devez vérifier l'identifiant d'une voiture dans votre base de données et l'utiliser à la place). Dans Postman, vous devez sélectionner la méthode HTTP appropriée dans la liste déroulante, entrer l'URL de la requête, puis cliquer sur le bouton Envoyer :

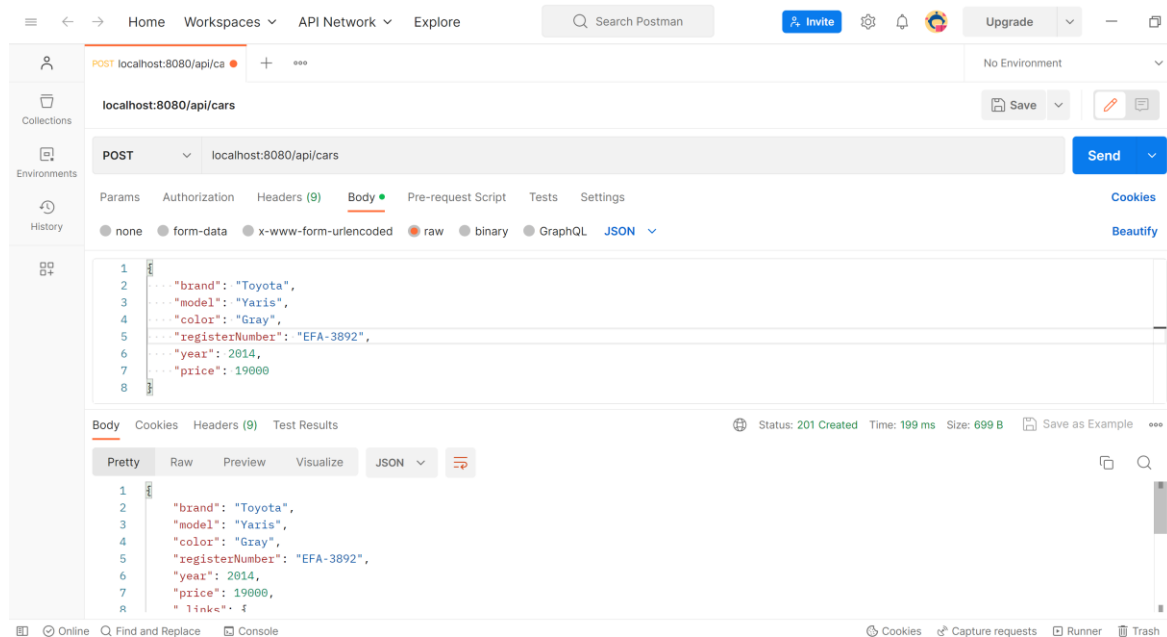


Si tout s'est bien passé, vous verrez le code de réponse **204 No Content** dans Postman. Après la demande de suppression réussie, vous verrez également qu'il ne reste plus que deux voitures dans la base de données si vous faites une requête **GET** à l'endpoint <http://localhost:8080/api/cars/>. Si vous avez reçu le code de réponse **404 Not Found**, vérifiez que vous utilisez un ID de voiture qui existe dans la base de données.

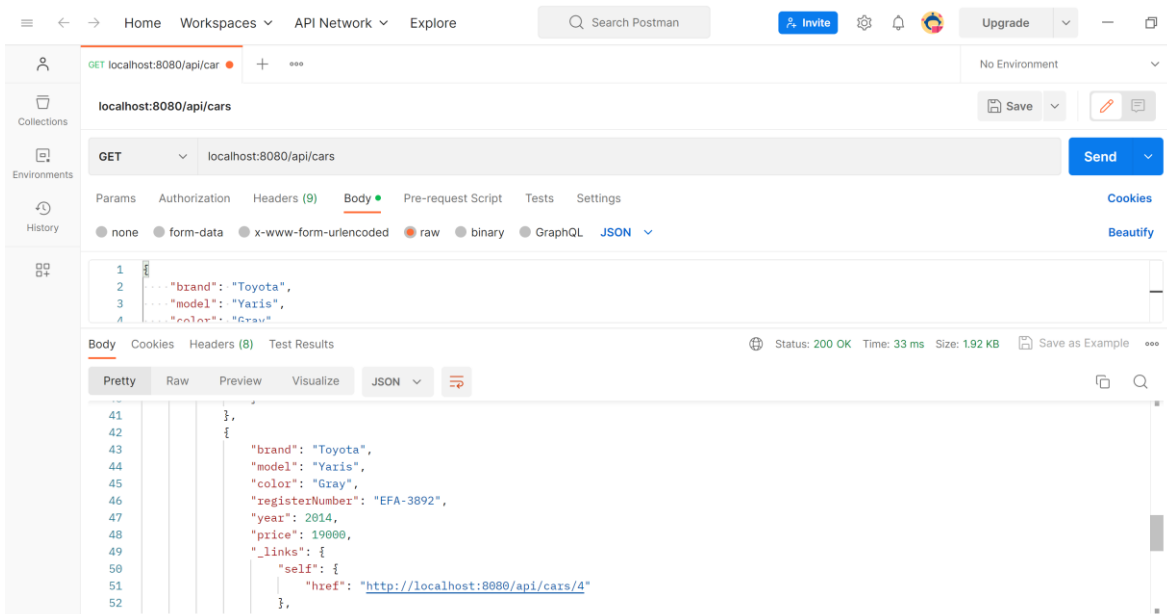
Lorsque nous voulons ajouter une nouvelle voiture à la base de données, nous devons utiliser la méthode **POST** et l'URL de la requête est <http://localhost:8080/api/cars>. L'en-tête doit contenir le champ **Content-Type** avec la valeur **application/json** et le nouvel objet de voiture sera incorporé dans le corps de la requête au format **JSON**.



Si vous cliquez sur l'onglet **Body** et sélectionnez **raw** dans **Postman**, vous pouvez taper une nouvelle chaîne JSON de voiture dans l'onglet **Body** comme illustré dans la capture d'écran suivante :

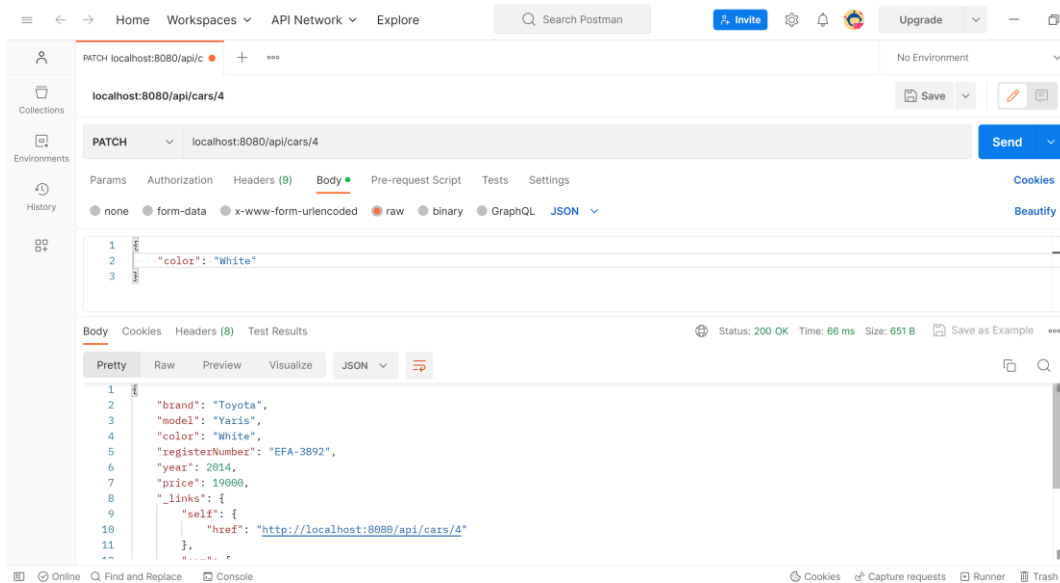


La réponse renverra un nouvel objet de voiture créé et le code de statut de la réponse sera **201 Created** si tout s'est bien passé. Maintenant, si vous faites à nouveau une requête **GET** à l'adresse **`http://localhost:8080/api/cars`**, vous verrez que la nouvelle voiture existe dans la base de données.



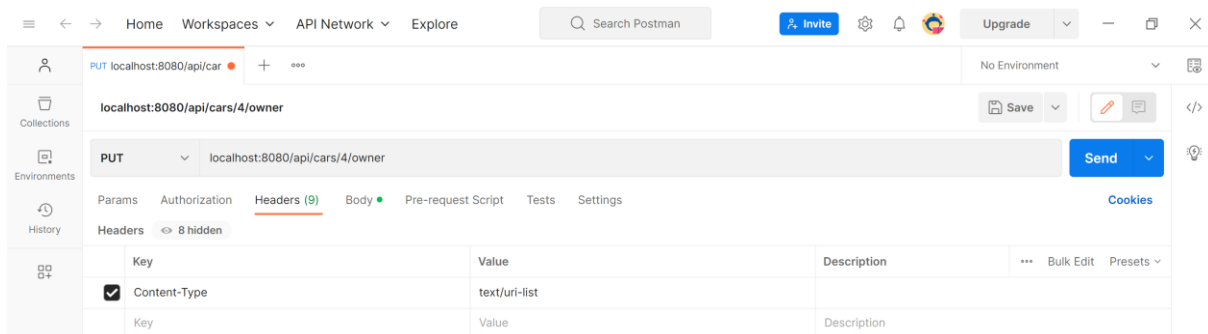
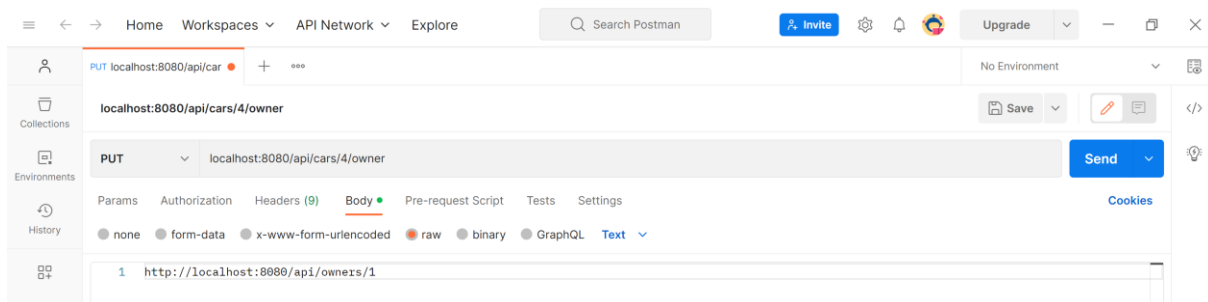
Pour mettre à jour les entités, nous pouvons utiliser la méthode **PATCH** et le lien vers la voiture que nous voulons mettre à jour (**`http://localhost:8080/api/cars/{id}`**). L'en-tête doit contenir le champ **Content-Type** avec la valeur **`application/json`** et l'objet de la voiture avec les données éditées sera donné à l'intérieur du corps de la requête. Si vous utilisez **PATCH**, vous devez envoyer uniquement les champs qui sont mis à jour. Si vous utilisez **PUT**, vous devez inclure tous les champs dans le corps de la requête.

Modifions la voiture que nous avons créée dans l'exemple précédent et changeons la couleur en blanc. La requête Postman est présentée dans la capture d'écran suivante (notez que nous avons défini l'en-tête comme dans l'exemple **POST** et que nous utilisons l'identifiant de la voiture dans l'URL) :

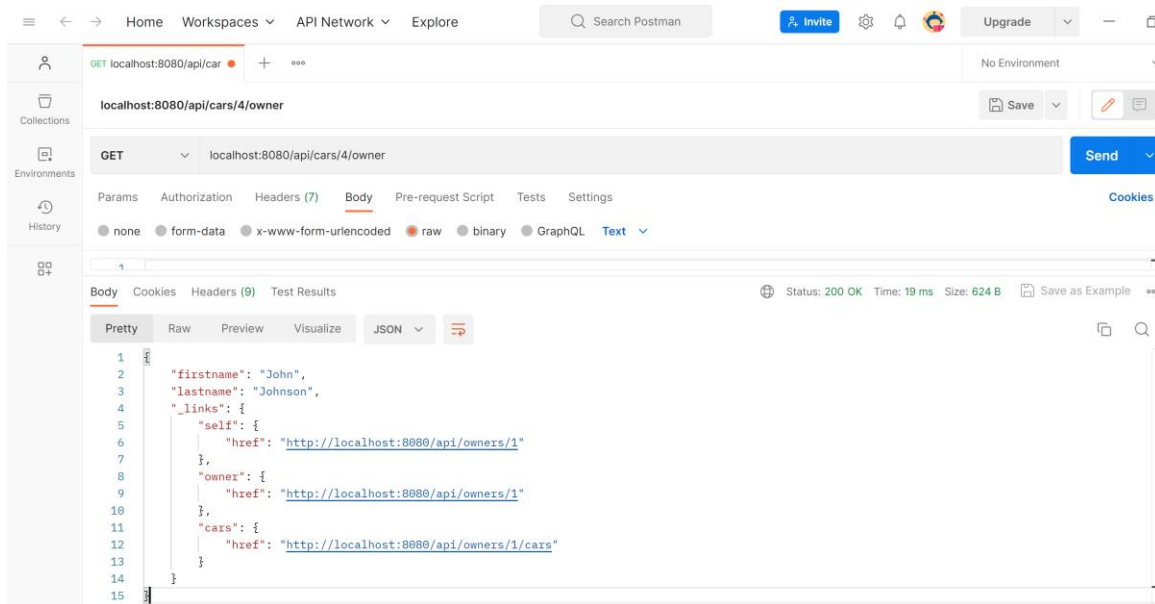


Si la mise à jour a réussi, le statut de réponse est **200 OK**. Si vous récupérez maintenant la voiture mise à jour en utilisant la requête **GET**, vous verrez que la couleur est mise à jour.

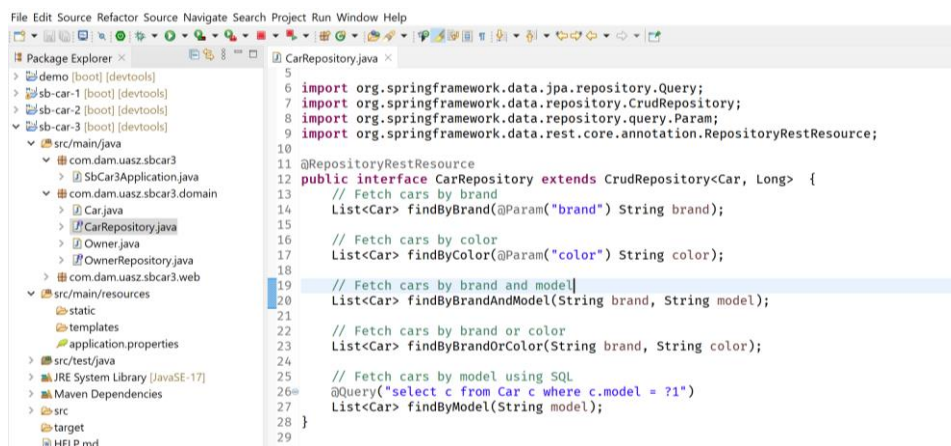
Ensuite, nous allons ajouter un propriétaire à la nouvelle voiture que nous venons de créer. Nous pouvons utiliser la méthode **PUT** et le chemin `http://localhost:8080/api/cars/{id}/owner`. Dans cet exemple, l'ID de la nouvelle voiture est 4, donc le lien est `http://localhost:8080/api/cars/4/owner`. Le contenu du corps est maintenant lié à un propriétaire, par exemple, <http://localhost:8080/api/owners/1>. La valeur **Content-Type** des en-têtes doit être **text/uri-list** dans ce cas.



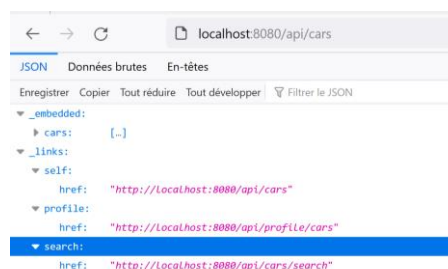
Enfin, vous pouvez effectuer une requête **GET** pour le propriétaire de la voiture et vous devriez voir que le propriétaire est maintenant lié à la voiture, comme indiqué dans la capture d'écran suivante :



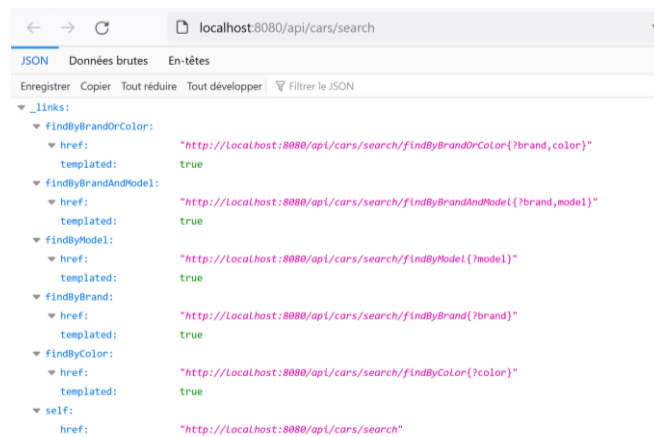
Dans le lab précédent, nous avons créé des requêtes pour notre repository. Ces requêtes peuvent également être incluses dans notre service. Pour inclure des requêtes, vous devez ajouter l'annotation **@RepositoryRestResource** à la classe repository. Les paramètres de requête sont annotés avec l'annotation **@Param**. Le code source suivant montre **CarRepository** avec ces annotations:



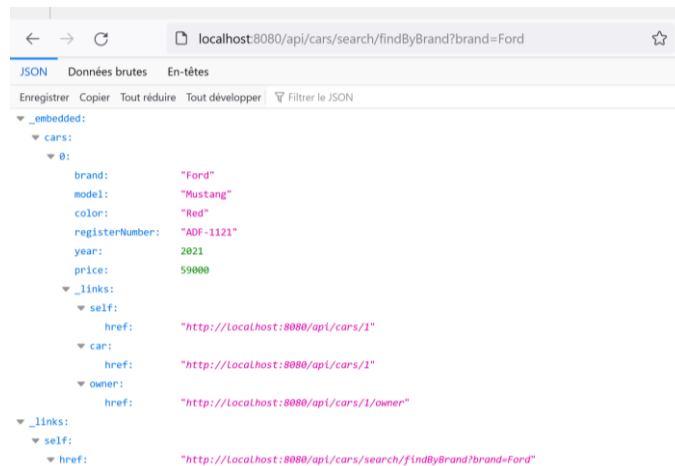
Maintenant, lorsque vous effectuez une requête **GET** sur le chemin **http://localhost:8080/api/cars**, vous pouvez voir qu'il y a un nouvel endpoint appelé **/search**.



Appeler le chemin **http://localhost:8080/api/cars/search** renvoie la réponse suivante :



À partir de la réponse, vous pouvez voir que les deux requêtes sont maintenant disponibles dans notre service. L'URL suivante démontre comment récupérer des voitures par marque : <http://localhost:8080/api/cars/search/findByBrand?brand=Ford>.



Nous avons maintenant créé l'**API RESTful de notre backend**, que nous consommerons plus tard avec notre frontend **React**.

Résumé

Dans ce lab, nous avons créé un service web **RESTful avec Spring Boot**. Tout d'abord, nous avons créé un contrôleur et une méthode qui renvoie toutes les voitures au format **JSON**.

Ensuite, nous avons utilisé **Spring Data REST** pour obtenir un service web entièrement fonctionnel avec toutes les fonctionnalités **CRUD**. Nous avons couvert différents types de requêtes nécessaires pour utiliser les fonctionnalités **CRUD** du service que nous avons créé. Enfin, nous avons également inclus nos requêtes dans le service web **RESTful**.