Développement d'applications Java

Lab_2 : Premières pas avec Spring-boot

Dans ce lab, nous allons créer une petite application avec Spring boot. Nous allons mettre en place un petit service web avec Spring boot, en créant une api de type REST. Nous aurons également besoin d'un outil comme postman afin de tester l'api ainsi créée.

Exercice 1

Créer une nouvelle application *Spring Boot* nommée **sb-coffee-1**.

Création d'un simple domaine : créer une classe **Coffee,** dans le fichier principal de l'application comme suit :

SbCoffee1Application.java

```
import java.util.UUID;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SbCoffee1Application {

    public static void main(String[] args) {
        SpringApplication.run(SbCoffee1Application.class, args);
    }
}


class Coffee {
    private final String id;
    private String name;

    public Coffee(String id, String name) {
        this.id = id;
        this.name = name;
    }

    public Coffee(String name) {
        this(UUID.randomUUID().toString(), name);
    }
```

La classe **Coffee** dispose de trois constructeurs :

- Un constructeur sans arguments (dont le corps est vide)
- Un constructeur avec tous les champs renseignés
- Un constructeur qui ne renseigne que le nom du café, l'id sera généré par un UUID.

La suite de cette classe, avec les getters et setters, est renseignée ci-dessous :

```
class Coffee {
    private final String id;
    private String name;

    public Coffee(String id, String name) {
        this.id = id;
        this.name = name;
    }

    public Coffee(String name) {
        this(UUID.randomUUID().toString(), name);
    }

    public String getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Créer, toujours dans le même fichier, une classe nommée **RestCoffeeController** qui va permettre de définir nos différentes routes associées aux différents verbes http : **GET, POST, PUT, DELETE**. Ces méthodes ou verbes http permettent de facilement mettre en œuvre des opérations **CRUD** pour **CREATE, RETRIEVE, UPDATE, DELETE**.

- Le verbe **GET** est utilisé dans pour effectuer des recherches, correspondant à **R : RETRIEVE**
- Le verbe **POST** est utilisé dans pour effectuer la création, correspondant à **C : CREATE**
- Le verbe **PUT** est utilisé dans pour effectuer une maj, correspondant à **U : UPDATE**
- Le verbe **DELETE** est utilisé dans pour effectuer une suppression, correspondant à **D : DELETE**.

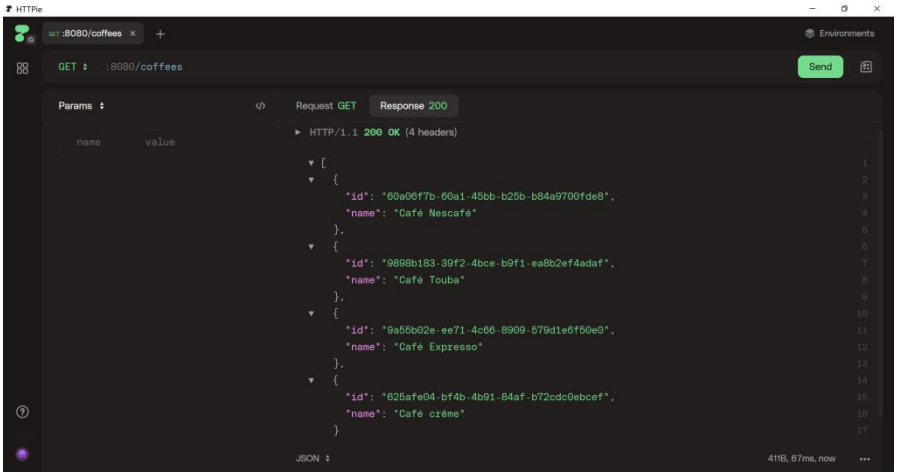
Nous verrons plus en détail ces aspects par la suite.

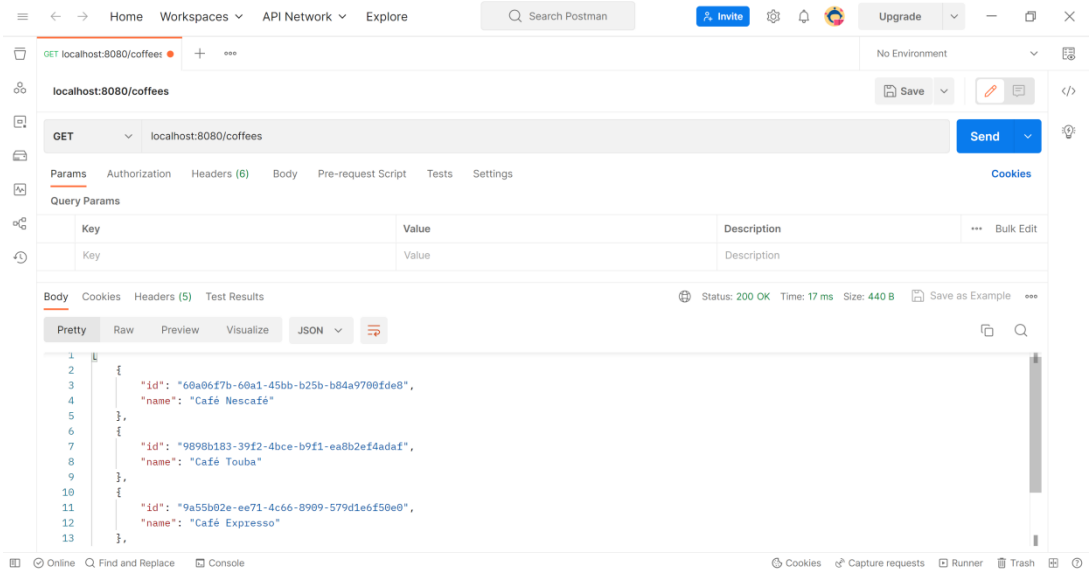
Dans ce bout de code, on stocke un ensemble de café en utilisant une **ArrayList**. Après, dans le constructeur, nous ajoutons un ensemble de café, ceci nous permettra d'avoir des résultats lorsque l'on effectue des recherches.

1) RECHERCHER : tous

```
55 @RestController
56 @RequestMapping("/coffees")
57 class RestApiCoffeeController {
58     private List<Coffee> coffees = new ArrayList<>();
59
60     public RestApiCoffeeController() {
61         coffees.addAll(List.of(
62             new Coffee("Café Nescafé"),
63             new Coffee("Café Touba"),
64             new Coffee("Café Expresso"),
65             new Coffee("Café Crême")
66         ));
67     }
68
69     @GetMapping
70     Iterable<Coffee> getCoffees() {
71             return coffees;
72     }
73 }
74
```

Après nous définissons une route au niveau de la racine **/coffees** de notre API REST. Ci-dessous, nous testons cette route en utilisant un client **httpie** et un client **postman**.





2) RECHERCHER : une

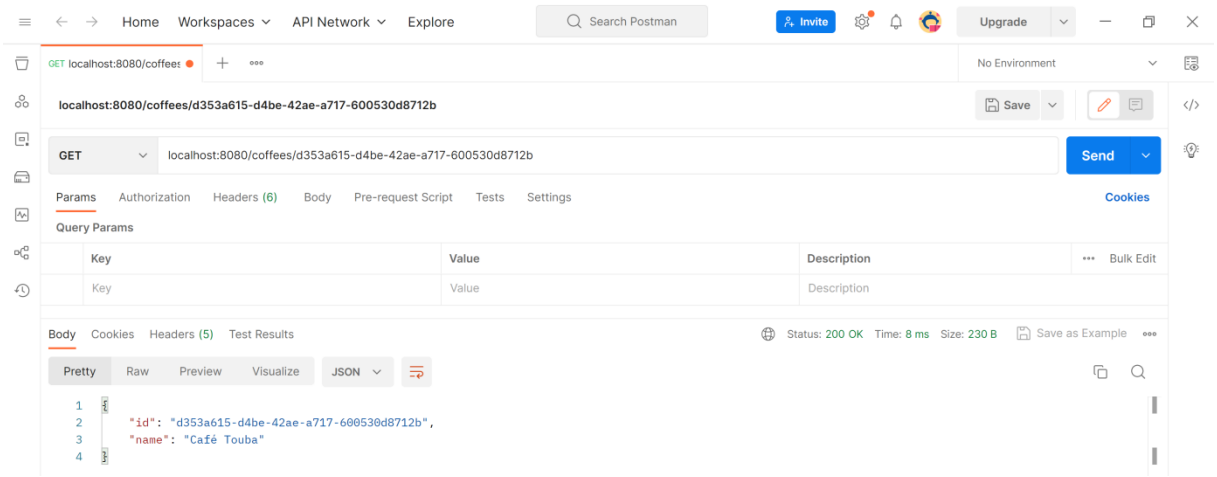
Après cela nous allons modifier notre contrôleur en ajoutant une nouvelle route. On invoque toujours une méthode **GET** mais avec cette fois comme URL, l'**ID** du café recherché :

```
73
74      @GetMapping("/{id}")
75      Optional<Coffee> getCoffeeById(@PathVariable String id) {
76          for (Coffee c: coffees) {
77              if (c.getId().equals(id)) {
78                  return Optional.of(c);
79              }
80          }
81          return Optional.empty();
82      }
83  }
84
```

On recherche un café en passant son ID. Cette méthode utilise **Optional**[1] de java8, qui est une meilleure à l'option **null**.

Le test de cette route est donné ci-dessous avec comme client **httpie** et **postman**

[1] https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html



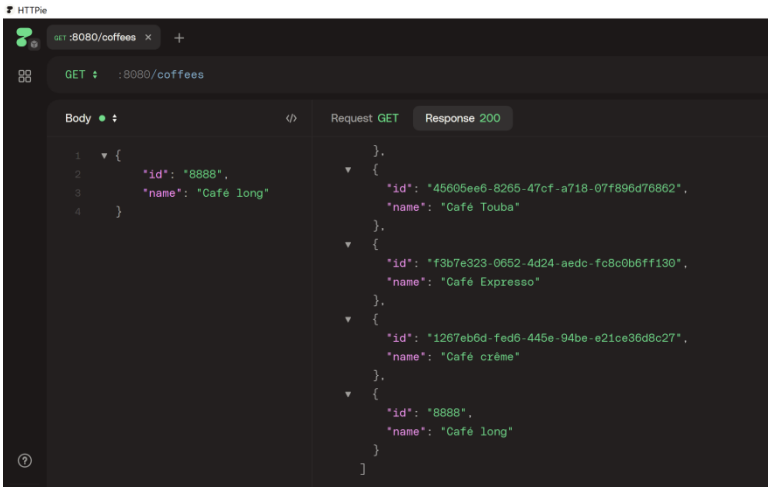
3) CREER

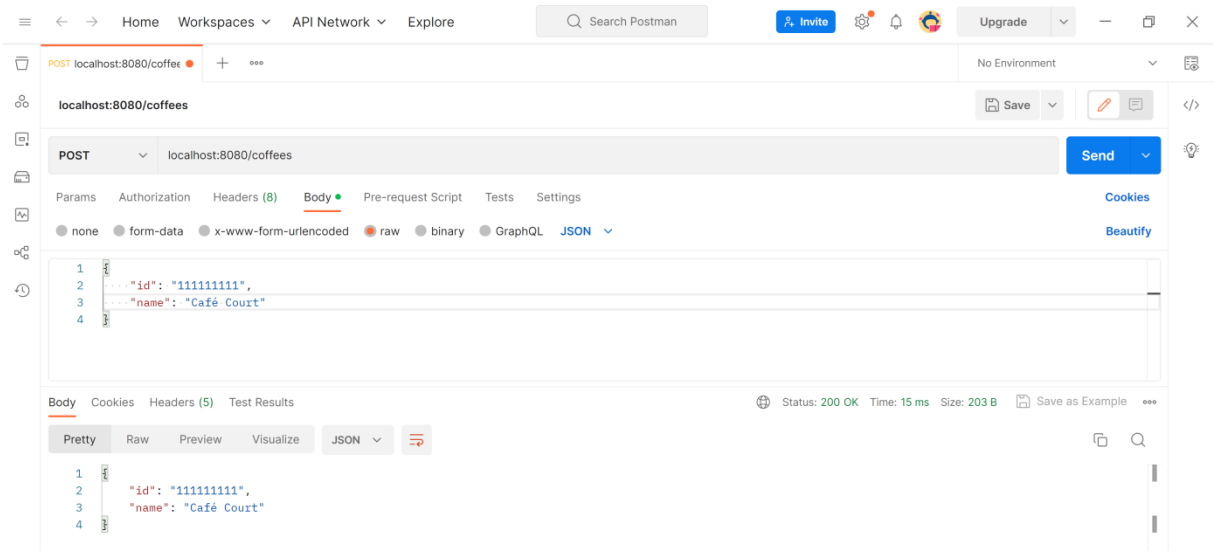
Maintenant nous allons voir comment créer un nouveau café, en utilisant la méthode **POST**. Cette méthode à la différence de **GET**, a ce qu'on appelle un corps qui permettra de renseigner les informations sur la nouvelle ressource à créer.

```
84      @PostMapping
85      Coffee postCoffee(@RequestBody Coffee coffee) {
86          coffees.add(coffee);
87          return coffee;
88      }
89  }
```

Ensuite nous effectuons les tests dans httpie et dans postman

Après en faisant un **GET**, on retrouve notre nouveau café :





4) MODIFIER

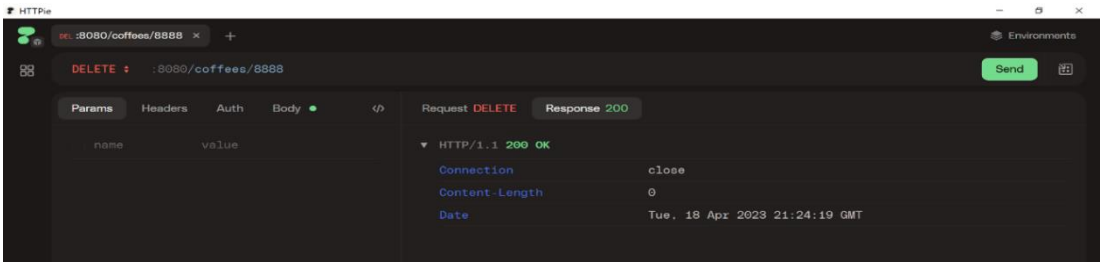
Maintenant nous allons essayer de modifier un café en utilisant la méthode **PUT**. Tout comme **POST**, cette méthode requiert un corps et faudra préciser en plus l'URL correspondant à la ressource en question à modifier. **PUT** va modifier la ressource si cette dernière est présente, sinon va créer une nouvelle ressource. En d'autres termes, peut être utilisé pour créer une nouvelle ressource, et donc jouer le rôle de **POST**.

```
89
90      @PutMapping("/{id}")
91      ResponseEntity<Coffee> putCoffee(@PathVariable String id, @RequestBody Coffee coffee) {
92          int coffeeIndex = -1;
93          for (Coffee c: coffees) {
94              if (c.getId().equals(id)) {
95                  coffeeIndex = coffees.indexOf(c);
96                  coffees.set(coffeeIndex, coffee);
97              }
98          }
99          return (coffeeIndex == -1) ?
100                 new ResponseEntity<>(postCoffee(coffee), HttpStatus.CREATED) :
101                 new ResponseEntity<>(coffee, HttpStatus.OK);
102     }
103 }
```

5) SUPPRIMER

Enfin il reste à implémenter l'opération de suppression avec la méthode **DELETE**.

```
03
04    @DeleteMapping("/{id}")
05    void deleteCoffee(@PathVariable String id) {
06        coffees.removeIf(c -> c.getId().equals(id));
07    }
08
09 }
10
```



NB

Le fait que les données soient sauvegardées entraine que ces dernières sont réinitialisées à chaque lancement du serveur. Par la suite, nous utiliserons différents types de BD afin de rendre plus pérennes nos données.

Exercice 2

Créer une application spring boot nommée **sb-car-1** qui va mettre en place une API de type REST, qui donc devra implémenter un CRUD. La ressource à gérer, est une collection de voitures, est présentée ci-dessous (ajouter un attribut id comme dans l'exercice 1) :

```
{
  "brand": "Toyota",
  "model": "Corolla",
  "color": "silver",
  "registerNumber": "BBA-3122",
  "year": 2021,
  "price": 32000
}
```