Spring MVC : services REST

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en Programmation par contrainte (IA)
Ingénieur en Génie logiciel

elmouelhi.achref@gmail.com

Plan

Spring & REST

Rappel

- Un des composants indispensable d'une application Spring MVC est le contrôleur
- Le contrôleur reçoit une requête HTTP de la part d'un utilisateur et communique avec le modèle et la vue pour retourner une réponse
- Le contrôleur peut aussi recevoir une requête HTTP de la part d'une autre application Back-end ou Front-end, communiquer avec le modèle et puis retourner une réponse sans construire une vue

Spring & REST

Rappel

- Un des composants indispensable d'une application Spring MVC est le contrôleur
- Le contrôleur reçoit une requête HTTP de la part d'un utilisateur et communique avec le modèle et la vue pour retourner une réponse
- Le contrôleur peut aussi recevoir une requête HTTP de la part d'une autre application Back-end ou Front-end, communiquer avec le modèle et puis retourner une réponse sans construire une vue

Ceci est l'objet de ce chapitre.

Considérons le contrôleur `PersonneRestController`

```
@Controller
public class PersonneRestController {
  @Autowired
  private PersonneRepository personneRepository;

  @GetMapping("/personnes")
  public String getPersonnes(Model model) {
    List<Personne> personnes = personneRepository.findAll();
    model.addAttribute("personnes", personnes);
    return "showPersonnes";
  }
  @GetMapping("/personnes/{id}")
  public String getPersonne(@PathVariable("id") long id, Model model) {
    Personne personne = personneRepository.findById(id).orElse(null);
    model.addAttribute("personne", personne);
    return "showPersonne";
  }
  @PostMapping("/addPersonne")
  public String addPersonne(Personne personne, Model model) {
    Personne personne2 = personneRepository.save(personne);
    model.addAttribute("personne", personne2);
    return "showPersonne";
  }
}
```

Spring & REST

Explication

- Le contrôleur `PersonneRestController` contient trois méthodes permettant soit de récupérer une ou plusieurs personnes de la base de données et d'afficher le résultat dans la vue, soit d'ajouter une nouvelle personne dans la base de données et l'afficher dans la vue.
- Le contrôleur retourne deux vues (soit `showPersonnes`, soit `showPersonne`) qui servent principalement à afficher des données récupérées de la base de données

Spring & REST

Remarques

- Comment fait-on si le contrôleur doit récupérer et retourner les données sans les afficher dans une vue (Notre projet **Spring** devient donc un service web) ?
- L'affichage sera accordé à un framework Front-end tel que **Angular** ou autre

Spring & REST

Nouveau contenu du contrôleur `PersonneRestController`

```
@Controller
public class PersonneRestController {

  @Autowired
  private PersonneRepository personneRepository;

  @GetMapping("/personnes")
  public String getPersonnes() {
    return personneRepository.findAll().toString();
  }

  @GetMapping("/personnes/{id}")
  public String getPersonne(@PathVariable("id") long id) {
    return personneRepository.findById(id).orElse(null).toString();
  }

  @PostMapping("/personnse")
  public String addPersonne(Personne personne) {
    System.out.println(personne);
    return personneRepository.save(personne).toString();
  }
}
```

Spring & REST

Problématique

- Si on saisit l'URL `localhost:8080/personnes` dans la barre d'adresse, la méthode `getPersonnes` sera exécutée
- La liste de personnes sera récupérée de la base de données
- Comme le type de la valeur de retour de cette méthode est `String`, le contrôleur va chercher à afficher la vue dont le nom est précisé après `return`
- Mais cette valeur ne correspond pas à une vue, elle correspond plutôt à des valeurs récupérées de la base de données.

Pour corriger ça, on peut utiliser l'annotation @ResponseBody

```
@Controller
public class PersonneRestController {

  @Autowired
  private PersonneRepository personneRepository;

  @GetMapping("/personnes")
  @ResponseBody
  public String getPersonnes() {
    return personneRepository.findAll().toString();
  }
  @GetMapping("/personnes/{id}")
  @ResponseBody
  public String getPersonne(@PathVariable("id") long id) {
    return personneRepository.findById(id).orElse(null).toString();
  }

  @PostMapping("/personnes")
  @ResponseBody
  public String addPersonne(Personne personne) {
    System.out.println(personne);
    return personneRepository.save(personne).toString();
  }
}
```

Spring & REST

Pour tester

- Aller sur `localhost:8080/FirstSpringMvc/personnes`
- Ou sur `localhost:8080/FirstSpringMvc/personnes/1`

Spring & REST

Pour tester

- Aller sur `localhost:8080/FirstSpringMvc/personnes`
- Ou sur `localhost:8080/FirstSpringMvc/personnes/1`

Constat

- Le résultat obtenu est une chaîne de caractère ou un tableau de chaîne de caractère
- Sachant que **Spring** nous offre la possibilité de récupérer le résultat sous format JSON

Pour corriger ça, on peut utiliser l'annotation @ResponseBody

```
@Controller
public class PersonneRestController {

  @Autowired
  private PersonneRepository personneRepository;

  @GetMapping("/personnes")
  @ResponseBody
  public List<Personne> getPersonnes() {
    return personneRepository.findAll();
  }
  @GetMapping("/personnes/{id}")
  @ResponseBody
  public Personne getPersonne(@PathVariable("id") long id) {
    return personneRepository.findById(id).orElse(null);
  }

  @PostMapping("/personnes")
  @ResponseBody
  public Personne addPersonne(Personne personne) {
    System.out.println(personne);
    return personneRepository.save(personne);
  }
}
```

Spring & REST

Avant de tester, il faut ajouter une dépendance `jackson` pour assurer la conversion d'objets Java en objets JSON ou XML

```
<!-- https://mvnrepository.com/artifact/com.fasterxml.
   jackson.dataformat/jackson-dataformat-xml -->
<dependency>
  <groupId>com.fasterxml.jackson.dataformat</groupId>
  <artifactId>jackson-dataformat-xml</artifactId>
  <version>2.9.5</version>
</dependency>
```

Spring & REST

Pour ajouter une personne

- utiliser Postman en précisant la méthode `POST` (Utiliser le format JSON car Spring est configuré par défaut au format JSON)
  - dans `Headers`, préciser la clé `Accept` et la valeur `application/json`
  - dans `Body`, cocher `raw` et sélectionner `JSON(application/json)`

Spring & REST

Pour ajouter une personne

- utiliser Postman en précisant la méthode `POST` (Utiliser le format JSON car Spring est configuré par défaut au format JSON)
  - dans `Headers`, préciser la clé `Accept` et la valeur `application/json`
  - dans `Body`, cocher `raw` et sélectionner `JSON(application/json)`

Résultat

- Erreur et la personne n'a pas été ajoutée dans la base de données
- Message affiché dans la console : `{"num":null, "nom":null, "prenom":null}`

Spring & REST

Pour récupérer l'objet envoyé défini dans le corps de la requête HTTP et assurer le binding avec l'objet défini comme paramètre de la méthode `addPersonne()`, on doit utiliser l'annotation `@RequestBody`

```
@PostMapping("/personnes")
@ResponseBody
public Personne addPersonne(@RequestBody Personne personne) {

  System.out.println(personne);
  return personneRepository.save(personne);

}
```

Spring & REST

Constats

- Toutes les méthodes du contrôleur ne retournent pas de vue
- Toutes ces méthodes sont annotées par `@ResponseBody`

Spring & REST

Constats

- Toutes les méthodes du contrôleur ne retournent pas de vue
- Toutes ces méthodes sont annotées par `@ResponseBody`

On peut optimiser

- On peut utiliser l'annotation `@RestController`

Spring & REST

Remplaçons `@ResponseBody` **par** `@RestController`

```
@RestController
public class PersonneRestController {

  @Autowired
  private PersonneRepository personneRepository;

  @GetMapping("/personnes")
  public List<Personne> getPersonnes() {
    return personneRepository.findAll();
  }

  @GetMapping("/personnes/{id}")
  public Personne getPersonne(@PathVariable("id") long id) {
    return personneRepository.findById(id).orElse(null);
  }

  @PostMapping("/personnes")
  public Personne addPersonne(@RequestBody Personne personne) {
    System.out.println(personne);
    return personneRepository.save(personne);
  }
}
```

Spring & REST

Exercice 1

Écrire puis tester les deux méthodes **HTTP** `put` et `delete` qui permettront de modifier ou supprimer une personne.

Spring & REST

Pour la suite

- supposons qu'une personne peut avoir une ou plusieurs adresses
- commençons par créer une classe `Adresse` avec 4 attributs `id`, `rue`, `ville` et `codePostal`
- déclarons dans `Personne` une liste de `Adresse`
- utilisons **Postman** pour ajouter des personnes dans la base de données avec leurs adresses

Spring & REST

La classe Adresse

```
@Entity
public class Adresse {

  @Id
  @GeneratedValue
  private Long id;
  private String rue;
  private String codePostal;
  private String ville;

  // N'oublions pas les getters / setters / toString
      / Constructeur sans paramètre
}
```

Spring & REST

Le nouveau contenu de la classe `Personne`

```
@Entity
@Table(name="personnes")
public class Personne {

  @Id
  @GeneratedValue
  private Integer num;
  private String nom;
  private String prenom;

  @ManyToMany(cascade = { CascadeType.MERGE, CascadeType.REFRESH},
     fetch = FetchType.EAGER)
  private List<Adresse> adresses = new ArrayList<Adresse>();

  // N'oublions pas les getters / setters / toString / constructeur
     sans paramètre / méthodes déléguées add et remove pour adresses
}
```

Seules les propriétés `MERGE` et `REFRESH` sont demandées.

Spring & REST

Le contenu de `AdresseRepository`

```
public interface AdresseRepository extends
   JpaRepository<Adresse, Long> {

}
```

Spring & REST

Exemple d'objet JSON à persister (en utilisant Postman)

```
{
  "nom": "el mouelhi",
  "prenom": "achref",
  "adresses": [
    {
      "rue": "paradis",
      "ville": "marseille",
      "codePostal": "13015"
    }
  ]
}
```

Message d'erreur parlant d'un objet faisant référence à une instance non sauvegardée

Spring & REST

Modifions `PersonneRestController` afin qu'il puisse attacher les entités inverses avant de les persister

```
@PostMapping("/personnes")
public Personne addPersonne(@RequestBody Personne personne) {

  System.out.println(personne);
  List <Adresse> adresses = personne.getAdresses();
  for (Adresse adresse : adresses) {
    Adresse adr = null;
    if (adresse.getId() != null) {
      adr = adresseRepository.findById(adresse.getId()).orElse(
        null);
      adresses.set(adresses.indexOf(adresse), adr);
    }  else {
      adr = adresseRepository.save(adresse);
    }
  }
  return personneRepository.saveAndFlush(personne);
}
```

Spring & REST

La réponse de la requête précédente (avec les clés primaires)


```
{
  "num": 1,
  "nom": "el mouelhi",
  "prenom": "achref",
  "adresses": [
    {
      "id": 1,
      "rue": "paradis",
      "ville": "marseille",
      "codePostal": "13015"
    }
  ]
}
```

Spring & REST

Et si on veut ajouter une personne en lui affectant l'adresse précédente

```
{
    "nom": "wick",
    "prenom": "john",
    "adresses": [
        {
            "id": 1,
            "rue": "paradis",
            "codePostal": "13015",
            "ville": "marseille"
        }
    ]
}
```

Spring & REST

Et si on veut ajouter une personne en lui affectant l'adresse précédente

```
{
    "nom": "wick",
    "prenom": "john",
    "adresses": [
        {
            "id": 1,
            "rue": "paradis",
            "codePostal": "13015",
            "ville": "marseille"
        }
    ]
}
```

Pour l'adresse, seule la clé primaire compte, les autres attributs sont facultatifs.

Spring & REST

En renvoyant l'objet précédent, la réponse est :

```
{
    "num": 2,
    "nom": "wick",
    "prenom": "john",
    "adresses": [
        {
            "id": 1,
            "rue": "paradis",
            "codePostal": "13015",
            "ville": "marseille"
        }
    ]
}
```

Spring & REST

Modifions la classe `Adresse` pour rendre son association avec la classe `Personne` bidirectionnelle

```
@Entity
public class Adresse {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  private String rue;
  private String codePostal;
  private String ville;
  @ManyToMany(fetch = FetchType.EAGER,mappedBy="adresses")
  private List<Personne> personnes = new ArrayList<Personne>();

  // N'oublions pas les getters / setters / méthodes déléguées
     add et remove pour personnes
}
```

Relancer le projet, en cas d'erreur, supprimer la base de données.

Spring & REST

Remarque

En essayant de consulter la liste de personnes (avec leurs adresses respectives), on a une boucle infinie car l'association est désormais bidirectionnelle.

Spring & REST

Remarque

En essayant de consulter la liste de personnes (avec leurs adresses respectives), on a une boucle infinie car l'association est désormais bidirectionnelle.

Solution

Pou arrêter la boucle infinie, on peut ajouter l'annotation `@JsonIgnore` dans la classe `Adresse`

Spring & REST

Nouveau contenu de la classe `Adresse`

```
import com.fasterxml.jackson.annotation.JsonIgnore;

//autres imports

@Entity
public class Adresse {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  private String rue;
  private String codePostal;
  private String ville;
  @JsonIgnore
  @ManyToMany(fetch = FetchType.EAGER,mappedBy="adresses")
  private List<Personne> personnes = new ArrayList<Personne>();

  // N'oublions pas les getters / setters / méthodes déléguées add et
     remove pour personnes
}
```

Spring & REST

Exercice 2

Créer une application **Angular** qui permet à un utilisateur, via des interfaces graphiques) la gestion de personnes (ajout, modification, suppression, consultation et recherche) en utilisant les web services définis par **Spring**.

Spring & REST

Exercice 2

Créer une application **Angular** qui permet à un utilisateur, via des interfaces graphiques) la gestion de personnes (ajout, modification, suppression, consultation et recherche) en utilisant les web services définis par **Spring**.

Pour régler le problème CORS

Il faut ajouter l'annotation `@CrossOrigin` au contrôleur.