Spring Boot : services web 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

Plan

Spring Boot

Service web (WS pour Web Service) ?

- Un programme (ensemble de fonctionnalités exposées en temps réel et sans intervention humaine)
- Accessible via internet, Intranet, réseaux privés...
- Indépendant de tout système d'exploitation
- Indépendant de tout langage de programmation
- Utilisant un système standard d'échange (**XML** ou **JSON**), ces messages sont généralement transportés par des protocoles internet connus **HTTP** (ou autres comme **FTP**, **SMTP**...)
- Pouvant communiquer avec d'autres WS

Spring Boot

Les WS peuvent utiliser les technologies web suivantes :

- **HTTP** (Hypertext Transfer Protocol) : le protocole, connu, utilisé par le World Wide Web et inventé par Roy Fiedling.
- **REST** (Representational State Transfer) : une architecture de services Web, créée aussi par Roy Fielding en 2000 dans sa thèse de doctorat.
- **SOAP** (Simple object Access Protocol) : un protocole, défini par Microsoft et IBM ensuite standarisé par **W3C**, permettant la transmission de messages entre objets distants (physiquement distribués).
- **WSDL** (Web Services Description Language) : est un langage de description de service web utilisant le format **XML** (standardisé par le **W3C** depuis 2007).
- **UDDI** (Universal Description, Discovery and Integration) : un annuaire de WS.

HTTP, **REST** (Representational State Transfer) et **RESTful** ?

- Les API REST sont basées sur le protocole **HTTP** (architecture client/serveur) et utilisent le concept de ressource.
- Une ressource est identifiée par une URI unique.
- L'API REST utilise donc des méthodes suivantes pour l'échange de données entre client et serveur
  - `GET` pour la récupération,
  - `POST` pour l'ajout,
  - `DELETE` pour la suppression,
  - `PUT` pour la modification,
  - ...
- Plusieurs formats possibles pour les données échangées : texte, **XML**, **JSON**...
- **RESTful** est l'adjectif qui désigne une API REST.

Spring Boot

Rôle du contrôleur dans une application **MVC**

- Le contrôleur reçoit une requête **HTTP** et communique avec modèle, service, constructeur de formulaires pour retourner une réponse **HTTP** contenant une page **HTML**.
- Le contrôleur peut aussi retourner une réponse **HTTP** ne contenant pas de vue.
- Il retourne des données sous format **JSON**, **XML**...

Spring Boot

Rôle du contrôleur dans une application **MVC**

- Le contrôleur reçoit une requête **HTTP** et communique avec modèle, service, constructeur de formulaires pour retourner une réponse **HTTP** contenant une page **HTML**.
- Le contrôleur peut aussi retourner une réponse **HTTP** ne contenant pas de vue.
- Il retourne des données sous format **JSON**, **XML**...

Ceci est l'objet de ce chapitre.

Spring Boot

Création de projet Spring Boot

- Aller dans `File > New > Other`
- Chercher `Spring`, dans `Spring Boot` sélectionner `Spring Starter Project` et cliquer sur `Next >`
- Saisir
  - `cours-spring-rest` dans `Name`,
  - `com.example` dans `Group`,
  - `cours-spring-rest` dans `Artifact`
  - `com.example.demo` dans `Package`
- Cliquer sur `Next`
- Chercher et cocher les cases correspondantes aux `Spring Data JPA`, `MySQL Driver`, `Lombok`, `Spring Web` et `Spring Boot DevTools`
- Cliquer sur `Next` puis sur `Finish`

Spring Boot

Explication

- Le package contenant le point d'entrée de notre application (la classe contenant le `puclic static void main`) est `com.example.demo`
- Tous les autres packages `dao`, `model`... doivent être dans le package `demo`.

Spring Boot

Explication

- Le package contenant le point d'entrée de notre application (la classe contenant le `puclic static void main`) est `com.example.demo`
- Tous les autres packages `dao`, `model`... doivent être dans le package `demo`.

Pour la suite, nous considérons

- une entité `Personne` à définir dans `com.example.demo.model`
- une interface DAO `PersonneRepository` à définir dans `com.example.demo.dao`
- un contrôleur REST `PersonneRestController` à définir dans `com.example.demo.controller`

Spring Boot & REST

Créons une entité `Personne` **dans** `com.example.demo.model`

```
package com.example.demo.model;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;

@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@RequiredArgsConstructor
public class Personne {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long num;
    @NonNull
    private String nom;
    @NonNull
    private String prenom;
}
```

Spring Boot & REST

Préparons notre interface DAO `PersonneRepository`

```
package com.example.demo.dao;

import org.springframework.data.jpa.repository.JpaRepository;

import com.example.demo.model.Personne;

public interface PersonneRepository extends JpaRepository<Personne,
   Long> {

}
```

Créons le contrôleur REST suivant

```
@CrossOrigin
@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) {
    return personneRepository.save(personne);
  }
}
```

Spring Boot

Dans `application.properties`, ajoutons les données permettant la connexion à la base de données et la configuration de `Hibernate`

```
server.servlet.context-path=/ws
spring.datasource.url = jdbc:mysql://localhost:3306/cours_rest?createDatabaseIfNotExist=true
spring.datasource.username = root
spring.datasource.password = root
spring.jpa.hibernate.ddl-auto = update
spring.jpa.show-sql = true
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.naming.physical-strategy = org.hibernate.boot.model.naming.
  PhysicalNamingStrategyStandardImpl
```


L'ajout de la propriété `spring.jpa.hibernate.naming.physical-strategy` permet de forcer **Hibernate** à utiliser les mêmes noms pour les tables et les colonnes que les entités et les attributs.

Spring Boot

Commençons par modifier le point d'entrée (qui implémentera l'interface `ApplicationRunner`) pour ajouter des tuples dans la base de données avant d'appeler nos services web REST

```
@SpringBootApplication
public class CoursSpringRestApplication implements ApplicationRunner {

  @Autowired
  private PersonneRepository personneRepository;

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

  @Override
  public void run(ApplicationArguments args) throws Exception {
    Personne personne1 = new Personne("wick", "john");
    Personne personne2 = new Personne("dalton", "jack");
    personneRepository.save(personne1);
    personneRepository.save(personne2);
  }
}
```

Spring Boot

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 frontend tel que **Angular**, **Vue.js** ou autre

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 long id) {
    return personneRepository.findById(id).orElse(null).toString();
  }

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

Spring Boot

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 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 Boot

Pour tester

allez à l'URL

- `localhost:8080/ws/personnes`
- Ou `localhost:8080/ws/personnes/1`

Spring Boot

Pour tester

allez à l'URL

- `localhost:8080/ws/personnes`
- Ou `localhost:8080/ws/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**

Spring Boot

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 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 Boot

Comment préciser le format (**JSON** ou **XML**) souhaité ?

- Utilisez **Postman** en choisissant la méthode `GET`
- Dans `Headers`, ajoutez les deux clés `Accept` et `Content-Type` avec la valeur `application/json` (ou `application/xml`)
- Envoyez

Spring Boot

On peut aussi indiquer avec l'attribut `produces` les formats acceptés

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

  @GetMapping(value = "/personnes", produces = { MediaType.APPLICATION_JSON_VALUE, MediaType.
     APPLICATION_XML_VALUE })
  @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 Boot

Pour ajouter une personne

- Utilisez **Postman** en précisant la méthode `POST`
- Dans `Headers`, ajoutez les clés `Accept` et `Content-Type` avec la valeur `application/json`
- Dans `Body`, cochez `raw` et sélectionnez `JSON`
- Ajoutez l'objet **JSON** suivant

```
{
    "nom": "maggio",
    "prenom": "carol"
}
```

- Envoyez

Spring Boot

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 Boot

Pour récupérer l'objet 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 Boot

On peut aussi indiquer avec l'attribut `consumes` les formats acceptés

```
@PostMapping(value = "/personnes", consumes = { MediaType.
   APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE })
@ResponseBody
public Personne addPersonne(Personne personne) {
    System.out.println(personne);
    return personneRepository.save(personne);
}
```

Spring Boot

Constats

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

Spring Boot

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 Boot

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 Boot

Exercice 1

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

Spring Boot

Exercice 2

Développer une application utilisant les frameworks **Angular**, **React.js** ou **Vue.js**, offrant à l'utilisateur des interfaces graphiques pour la gestion des données relatives aux personnes. Ces fonctionnalités incluent l'ajout, la modification, la suppression, la consultation et la recherche des informations, en se basant sur le service web existant.

Spring Boot

Exercice 2

Développer une application utilisant les frameworks **Angular**, **React.js** ou **Vue.js**, offrant à l'utilisateur des interfaces graphiques pour la gestion des données relatives aux personnes. Ces fonctionnalités incluent l'ajout, la modification, la suppression, la consultation et la recherche des informations, en se basant sur le service web existant.

Pour régler le problème **CORS** : deux solutions

- Annoter les contrôleurs eu les méthodes par `@CrossOrigin`
- Définir une classe de configuration dans laquelle on précise ce qu'on autorise

Spring Boot

Première solution avec `@CrossOrigin`

```
@CrossOrigin
@RestController
public class PersonneRestController {

    // contenu précédent

}
```

Spring Boot

Deuxième solution avec une classe de configuration

```
package com.example.demo.configuration;

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry
        .addMapping("/**")
        .allowedMethods("GET", "POST", "PUT", "UPDATE", "DELETE", "OPTIONS")
        .allowedOrigins("http://localhost:4200");
    }
}
```

Spring Boot

Pour la suite

- supposons qu'une personne peut avoir une ou plusieurs adresses
- commençons par créer une entité `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 Boot

La classe `Adresse`

```
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@RequiredArgsConstructor
public class Adresse {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @NonNull
    private String rue;
    @NonNull
    private String codePostal;
    @NonNull
    private String ville;

}
```

Spring Boot

Le nouveau contenu de la classe `Personne`

```
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@RequiredArgsConstructor
public class Personne {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long num;
    @NonNull
    private String nom;
    @NonNull
    private String prenom;
    @NonNull
    @ManyToMany(cascade = { CascadeType.MERGE, CascadeType.REFRESH}, fetch = FetchType.EAGER)
    private List<Adresse> adresses;

}
```

Spring Boot

Le nouveau contenu de la classe `Personne`

```
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@RequiredArgsConstructor
public class Personne {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long num;
    @NonNull
    private String nom;
    @NonNull
    private String prenom;
    @NonNull
    @ManyToMany(cascade = { CascadeType.MERGE, CascadeType.REFRESH}, fetch = FetchType.EAGER)
    private List<Adresse> adresses;

}
```

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

Spring Boot

Contenu de `AdresseRepository`

```
package com.example.demo.dao;

import org.springframework.data.jpa.repository.JpaRepository;

import com.example.demo.model.Adresse;

public interface AdresseRepository extends JpaRepository<Adresse, Long> {

}
```

Spring Boot

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

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

Spring Boot

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 Boot

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);
  adresseRepository.saveAll(personne.getAdresses())
  return personneRepository.saveAndFlush(personne);
}
```

Spring Boot

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);
  adresseRepository.saveAll(personne.getAdresses())
  return personneRepository.saveAndFlush(personne);
}
```


Sans oublier

```
@Autowired
private AdresseRepository adresseRepository;
```

Spring Boot

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 Boot

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

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

Spring Boot

Et si on voulait 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. sauvegardée.

Spring Boot

En renvoyant l'objet précédent, la réponse est :

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

Spring Boot

Exercice

Créer le contrôleur `AdresseRestController` avec les 5 actions principales.

Spring Boot

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

```
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@RequiredArgsConstructor
public class Adresse {

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

  // N'oublions pas les getter et setter
}
```

Spring Boot

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

```
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@RequiredArgsConstructor
public class Adresse {

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

  // N'oublions pas les getter et setter
}
```

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

Spring Boot

Remarque

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

Spring Boot

Remarque

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

Solution

Pou arrêter la boucle infinie, on peut utiliser l'annotation `@JsonIgnore` ou `@JsonIgnoreProperties`.

Spring Boot

Définitions

- **Sérialisation** : convertir un objet **Java** en une chaîne **JSON** pour l'envoyer via une **API REST**.
- **Désérialisation** : convertir une chaîne **JSON** reçue via une **API REST** en un objet **Java** utilisable.

Spring Boot

Quelques bibliothèques pour la sérialisation et la désérialisation en **Java**

- **Jackson** (développé par **FasterXML**)
- **Gson** (de **Google**) : plus simple à utiliser pour des cas de base et offre moins de fonctionnalités avancées que **Jackson**
- **JSON-B** (standard basé sur la spécification **JSR 367**) : Moins flexible et moins riche en termes de fonctionnalités que **Jackson**.
- **Moshi** (développée par **Square**, cofondée par **Jack Dorsey** (cofondateur de **Twitter**) et **Jim McKelvey**) : très proche de **Gson**, mais avec un meilleur support pour les annotations **Java** modernes et plus léger que **Jackson**.
- ...

Spring Boot

Quelques annotations **Jackson**

- `@JsonIgnore` : utilisé pour ignorer un champ spécifique dans une classe (lors de la sérialisation ou de la désérialisation).
- `@JsonIgnoreProperties` : utilisé pour ignorer plusieurs champs d'une classe (lors de la sérialisation ou de la désérialisation).

Spring Boot

Nouveau contenu de la classe `Adresse`

```
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@RequiredArgsConstructor
public class Adresse {

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

Spring Boot

Problématique

- En demandant la liste de personnes à `PersonneRestController`, on obtient une liste des personnes avec leurs adresses.
- En demandant la liste d'adresses à `AdresseRestController`, on obtient une liste des adresses sans les personnes.

Spring Boot

Problématique

- En demandant la liste de personnes à `PersonneRestController`, on obtient une liste des personnes avec leurs adresses.
- En demandant la liste d'adresses à `AdresseRestController`, on obtient une liste des adresses sans les personnes.

Solution

Utiliser `@JsonIgnoreProperties`.

Spring Boot

Nouveau contenu de la classe `Adresse`

```
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@RequiredArgsConstructor
public class Adresse {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @NonNull
    private String rue;
    @NonNull
    private String codePostal;
    @NonNull
    private String ville;

    @JsonIgnoreProperties("adresses")
    @ManyToMany(fetch = FetchType.EAGER, mappedBy = "adresses")
    private List<Personne> personnes = new ArrayList<Personne>();
}
```

Spring Boot

Nouveau contenu de la classe `Personne`

```
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@RequiredArgsConstructor
public class Personne {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long num;
    @NonNull
    private String nom;
    @NonNull
    private String prenom;
    @NonNull
    @ManyToMany(cascade = { CascadeType.MERGE, CascadeType.REFRESH },
       fetch = FetchType.EAGER)
    private List<Adresse> adresses;
}
```

Spring Boot

Autres annotations **Jackson**

- `@JsonProperty` : permet de spécifier le nom d'une propriété **JSON** lors de la sérialisation et de la désérialisation (utile lorsque le nom de la propriété dans l'objet Java diffère du nom dans le **JSON**).
- `@JsonAlias` : permet de spécifier plusieurs noms possibles pour une propriété lors de la désérialisation.
- `@JsonFormat` : utilisée pour spécifier le format de sérialisation/désérialisation des champs, en particulier pour les dates et heures.
  Exemple (`@JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "dd-MM-yyyy HH:mm:ss")`)
- ...

Spring Boot

curl : **C**lient for **URL**s

- Commande permettant le transfert de données
- Utilisant plusieurs protocoles

Spring Boot

Pour demander la liste de personnes

```
curl --request GET http://localhost:8080/ws/personnes
```

Spring Boot

Pour demander la liste de personnes

```
curl --request GET http://localhost:8080/ws/personnes
```

Par défaut, la méthode `HTTP` est `GET`

```
curl http://localhost:8080/ws/personnes
```

Spring Boot

Pour demander la liste de personnes

```
curl --request GET http://localhost:8080/ws/personnes
```

Par défaut, la méthode `HTTP` est `GET`

```
curl http://localhost:8080/ws/personnes
```

Pour récupérer seulement l'entête de la réponse

```
curl -I http://localhost:8080/ws/personnes
```

Spring Boot

Pour avoir un résultat au format XML (Attention aux espaces non-nécessaires)

```
curl --header "Accept:application/xml" http://localhost:8080/ws/
  personnes
```

Spring Boot

Pour avoir un résultat au format XML (Attention aux espaces non-nécessaires)

```
curl --header "Accept:application/xml" http://localhost:8080/ws/
  personnes
```

Le raccourci

```
curl -H "Accept:application/xml" http://localhost:8080/ws/personnes
```

Spring Boot

Pour avoir un résultat au format XML (Attention aux espaces non-nécessaires)

```
curl --header "Accept:application/xml" http://localhost:8080/ws/
  personnes
```

Le raccourci

```
curl -H "Accept:application/xml" http://localhost:8080/ws/personnes
```

Pour sauvegarder le résultat dans un fichier

```
curl -H "Accept:application/xml" -o personnes.xml http://localhost
  :8080/ws/personnes
```

Spring Boot

Pour ajouter une personne

```
curl --request POST -H "Content-Type:application/json" http://localhost
  :8080/ws/personnes --data "{\"nom\":\"wick\",\"prenom\":\"john\"}"
```

Spring Boot

Pour ajouter une personne

```
curl --request POST -H "Content-Type:application/json" http://localhost
  :8080/ws/personnes --data "{\"nom\":\"wick\",\"prenom\":\"john\"}"
```

Notons que `--request POST` est facultatif si nous utilisons `-d`, car ce dernier implique une requête `POST`.

```
curl -H "Content-Type:application/json" http://localhost:8080/ws/
  personnes -d "{\"nom\":\"wick\",\"prenom\":\"john\"}"
```

Spring Boot

Pour ajouter une personne

```
curl --request POST -H "Content-Type:application/json" http://localhost
  :8080/ws/personnes --data "{\"nom\":\"wick\",\"prenom\":\"john\"}"
```

Notons que `--request POST` est facultatif si nous utilisons `-d`, car ce dernier implique une requête `POST`.

```
curl -H "Content-Type:application/json" http://localhost:8080/ws/
  personnes -d "{\"nom\":\"wick\",\"prenom\":\"john\"}"
```

Le mode mono-ligne ne facilite pas la lecture, on peut utiliser le mode multi-lignes avec ^ sous Windows ou \ sous Linux (N'oublions pas l'espace avant)

```
curl -H "Content-Type:application/json" ^
http://localhost:8080/ws/personnes ^
-d "{\"nom\":\"wick\",\"prenom\":\"john\"}"
```

Spring Boot

Pour avoir de l'aide

```
curl -h
```

Spring Boot

Problématique

- Si nous envoyons une requête **HTTP** de type **GET** à l'URL `http://localhost:8080/ws/personnes/1`, nous obtiendrons un objet **JSON** contenant des informations sur la personne ayant le numéro `1`.
- Si nous demandons des informations sur une personne qui n'existe pas (par exemple numéro `10000`), nous n'obtiendrons pas de réponse mais un statut **200 OK**.
- Ce type de retour peut avoir des graves conséquences au récepteur.
- Nous préférons générer une exception pour un tel cas.

Spring Boot

Commençons par créer notre classe `PersonneNotFoundException`

```
package com.example.demo.exception;

public class PersonneNotFoundException extends RuntimeException {
        public PersonneNotFoundException() {
                super("Personne introuvable");
        }
}
```

Spring Boot

Utilisons l'exception dans notre méthode `GET`

```
public Personne getPersonne(@PathVariable("id") long id) {
        var personne = personneRepository.findById(id).orElse(null);
        if (personne == null) {
                throw new PersonneNotFoundException();
        }
        return personne;
}
```

Spring Boot

En envoyant une requête `GET` à `http://localhost:8080/personnes/10000`, la réponse est :

```
état HTTP 500 – Erreur interne du serveur


Type: Rapport d'exception

message: Request processing failed; nested exception is com.
   example.demo.exception.PersonneNotFoundException: Personne
   introuvable

description: Le serveur a rencontré une erreur interne qui l'a
   empêché de satisfaire la requête.
```

Spring Boot

Pour modifier le code de 500 à 404, modifions notre classe d'exception

```
package com.example.demo.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.NOT_FOUND)
public class PersonneNotFoundException extends RuntimeException {
        public PersonneNotFoundException() {
                super("Personne introuvable");
        }
}
```

Spring Boot

Principaux Status Codes **HTTP**

- **200 OK** : La requête a réussi.
- **201 Created** : La requête a abouti et une nouvelle ressource a été créée.
- **202 Accepted** : La requête a été acceptée pour traitement, mais celui-ci n'est pas encore terminé.
- **203 Non-Authoritative Information** : La requête a réussi, mais les informations renvoyées peuvent ne pas être fiables.
- **204 No Content** : La requête a réussi, mais il n'y a pas de contenu à renvoyer.
- **400 Bad Request** : La requête est mal formée ou invalide.
- **401 Unauthorized** : L'authentification est requise pour accéder à la ressource.
- **403 Forbidden** : Accès interdit à la ressource, même avec authentification.
- **404 Not Found** : La ressource demandée est introuvable.
- **405 Method Not Allowed** : La méthode de requête n'est pas autorisée pour cette ressource.
- **500 Internal Server Error** : Une erreur interne du serveur a empêché le traitement de la requête.

Spring Boot

Pour plus d'informations sur les Status Codes **HTTP**

https://restfulapi.net/http-status-codes/

Spring Boot

En renvoyant la requête précédente, la réponse est :

```
état HTTP 404 – Non trouvé


"status": 404,
"error": "Not Found",
"trace": "com.example.demo.exception.
   PersonneNotFoundException: Personne introuvable
   ... "
"message": "Personne introuvable",
```

Spring Boot

La classe d'exception peut être simplifiée en remontant le message d'erreur au niveau de l'annotation

```
package com.example.demo.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(value = HttpStatus.NOT_FOUND, reason = "Personne introuvable")
public class PersonneNotFoundException extends RuntimeException {

}
```

Spring Boot

En renvoyant la requête précédente, on reçoit la même réponse

```
état HTTP 404 – Non trouvé


"status": 404,
"error": "Not Found",
"trace": "com.example.demo.exception.
   PersonneNotFoundException: Personne introuvable
   ... "
"message": "Personne introuvable",
```

Spring Boot

On peut aussi utiliser l'exception prédéfinie `ResponseStatusException`

```
@GetMapping("/personnes/{id}")
public Personne getPersonne(@PathVariable long id) {
    var personne =  personneRepository.findById(id).orElse(null);
    if (personne == null) {
        throw new ResponseStatusException(
                HttpStatus.NOT_FOUND,
                "Personne avec id = " + id + " est introuvable"
        );
    }
    return personne;
}
```

Spring Boot

En renvoyant la requête précédente, on reçoit la réponse suivante

```
état HTTP 404 – Non trouvé


"status": 404,
"error": "Not Found",
"trace": "org.springframework.web.server.ResponseStatusException: 404 NOT_FOUND \"Personne avec
    id = 100 est introuvable\" ... "
"message": "Personne avec id = 100 est introuvable"
```

Spring Boot

Pour les méthodes qui n'ont qu'un seul code serveur, on peut utiliser l'annotation `@ResponseStatus` pour retourner le code `HTTP` 201 (`CREATED`)

```
@ResponseStatus(HttpStatus.CREATED)
@PostMapping("/personnes")
public Personne addPersonne(@RequestBody Personne personne) {
        adresseRepository.saveAll(personne.getAdresses());
        return personneRepository.save(personne);
}
```

Spring Boot

On peut aussi utiliser le constructeur de la classe `ResponseEntity` qui prend deux paramètres : la valeur de retour et le code serveur

```
@DeleteMapping("/personnes/{id}")
public ResponseEntity<Boolean> deletePersonne(@PathVariable Long id) {

    var personne = personneRepository.findById(id).orElse(null);
    if (personne == null) {
       return new ResponseEntity<Boolean>(false, HttpStatus.NOT_FOUND);
    }
    personneRepository.deleteById(id);
    return new ResponseEntity<Boolean>(true, HttpStatus.NO_CONTENT);
}
```

Spring Boot

On peut aussi combiner `ResponseEntity` avec les exceptions précédentes

```
@DeleteMapping("/personnes/{id}")
public ResponseEntity<Boolean> deletePersonne(@PathVariable Long id) {

    var personne = personneRepository.findById(id).orElse(null);
    if (personne == null) {
        throw new ResponseStatusException(
            HttpStatus.NOT_FOUND,
            "Personne avec id = " + id + " est introuvable"
        );
    }
    personneRepository.deleteById(id);
    return new ResponseEntity<Boolean>(true, HttpStatus.NO_CONTENT);
}
```

Spring Boot

Pour contrôler les valeurs reçues avant insertion dans la base de données

- ajouter le module de validation de **JPA**/**Hibernate** dans `pom.xml`
- modifier l'entité `Personne` en rajoutant lea validateurs (annotations **JPA**)
- modifier le contrôleur pour vérifier la validité des données et pouvoir retourner les messages d'erreur

Spring Boot

Pour ajouter la dépendance de validation

- Faire clic droit sur le projet et aller dans `Spring > Edit Starters`
- Cocher la case `Validation` dans `I/O`

Spring Boot

Pour ajouter la dépendance de validation

- Faire clic droit sur le projet et aller dans `Spring > Edit Starters`
- Cocher la case `Validation` dans `I/O`

Ou ajouter les dépendances suivantes

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

Spring Boot

Définissons nos règles de validation dans le modèle (l'entité)

```
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@RequiredArgsConstructor
public class Personne {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long num;
    @NonNull
    @Size(min = 2)
    @NotEmpty(message = "Le champ nom est obligatoire")
    private String nom;
    @NonNull
    @NotEmpty(message = "Le champ prénom est obligatoire")
    @Size(min = 2)
    private String prenom;
    @NonNull
    @ManyToMany(cascade = { CascadeType.MERGE, CascadeType.REFRESH }, fetch = FetchType.EAGER)
    private List<Adresse> adresses;
}
```

Spring Boot

Les annotations de validation utilisées

- `@Size` : pour indiquer la taille (le nombre de caractère) min et/ou max d'un champ (on peut aussi utiliser `@Min` et `@Max`)
- `@NotEmpty` : pour préciser qu'un champ ne peut pas être vide (à ne pas confondre avec `@NotNull` qui concerne plutôt le champ d'une table et non pas d'un formulaire)

Spring Boot

Autres validateurs pour chaînes de caractères

- `@NotNull` : Le champ ne doit pas être `null`.
- `@NotBlank` : Le champ ne doit pas être vide et doit contenir des caractères non blancs.
- `@NotEmpty` : Le champ ne doit pas être vide (longueur > 0).
- `@Size(min, max)` : Contrainte de taille minimale et maximale.
- `@Pattern(regex)` : Le champ doit correspondre à une expression régulière.
- `@Email` : Valide si le champ respecte le format d'un email.
- `@Length(min=, max=)` : Vérifie si la longueur de la chaîne de caractères est dans l'intervalle

Spring Boot

Autres validateurs pour types numériques

- `@Min(value)` : Le champ doit être supérieur ou égal à une valeur donnée.
- `@Max(value)` : Le champ doit être inférieur ou égal à une valeur donnée.
- `@Range(min=, max=)` : La valeur du champ entier doit être comprise entre les valeurs `min` et `max` spécifiées.
- `@DecimalMin(value)` : Pour les nombres à virgule flottante, minimum inclus.
- `@DecimalMax(value)` : Pour les nombres à virgule flottante, maximum inclus.
- `@Positive` : Le champ doit être strictement positif.
- `@Negative` : Le champ doit être strictement négatif.
- `@Digits(integer, fraction)` : Limitation du nombre de chiffres dans la partie entière et fractionnaire.

Spring Boot

Autres validateurs pour types date/temps

- `@Past` : Le champ doit représenter une date dans le passé.
- `@PastOrPresent` : Le champ doit être une date passée ou la date actuelle.
- `@Future` : Le champ doit représenter une date dans le futur.
- `@FutureOrPresent` : Le champ doit être une date future ou la date actuelle.

Spring Boot

Autres annotations

- `@AssertTrue` : Le champ doit être `true`.
- `@AssertFalse` : Le champ doit être `false`.
- `@ISBN` : Le champ doit être un numéro **ISBN** valide.
- `@Pattern` : Le champ doit respecter une expression régulière.
- `@URL(protocol=, host=, port=, regexp=, flags=)` : Le champ doit correspondre à une **URL**.
- `@CreditCardNumber` : Le champ doit correspondre à un numéro de carte bancaire.
- ...

Spring Boot

Pour plus de détails

https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/#section-validating-bean-constraints

Spring Boot

Essayons d'ajouter un nouvel objet de type `Personne`

- Utilisez **Postman** en précisant la méthode `POST`
- Dans `Headers`, ajoutez les clés `Accept` et `Content-Type` avec la valeur `application/json`
- Dans `Body`, cochez `raw` et sélectionnez `JSON`
- Ajoutez l'objet **JSON** suivant

```
{
    "nom": "w",
    "prenom": "j",
    "adresses": [
        {
            "rue": "paradis",
            "codePostal": "13015",
            "ville": "marseille"
        }
    ]
}
```

- Envoyez

Spring Boot

Constat

Les champs `nom` et `prénom` contenaient des valeurs invalides, mais l'objet a bien été inséré dans la base de données.

Spring Boot

Pour rectifier le problème précédent, on utilise l'annotation `@Valid`

```
@ResponseStatus(HttpStatus.CREATED)
@PostMapping("/personnes")
public Personne addPersonne(@Valid @RequestBody  Personne personne) {
    adresseRepository.saveAll(personne.getAdresses());
    return personneRepository.save(personne);
}
```

Spring Boot

Pour rectifier le problème précédent, on utilise l'annotation `@Valid`

```
@ResponseStatus(HttpStatus.CREATED)
@PostMapping("/personnes")
public Personne addPersonne(@Valid @RequestBody  Personne personne) {
    adresseRepository.saveAll(personne.getAdresses());
    return personneRepository.save(personne);
}
```


Renvoyer la requête précédente et vérifiez qu'une erreur a été retournée (videz le cache si nécessaire)

Spring Boot

Remarque

Nous avons également la possibilité de définir notre propre validateur.

Spring Boot

Remarque

Nous avons également la possibilité de définir notre propre validateur.

Exemple

Définissons un validateur `Uppercase` qui vérifie qu'une chaîne de caractères est en majuscule.

Spring Boot

Étapes

- Créez une annotation personnalisée `@Uppercase` (une `@interface`)
- Implémentez la logique de validation dans une classe `UppercaseValidator` (qui implémente `ConstraintValidator` qui applique cette annotation

Spring Boot

Définissons l'annotation `@Uppercase`

```
package com.example.demo.validators;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import jakarta.validation.Constraint;
import jakarta.validation.Payload;

@Constraint(validatedBy = UppercaseValidator.class)
@Target({ ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER })
@Retention(RetentionPolicy.RUNTIME)
public @interface Uppercase {
    String message() default "Le texte doit être en majuscules.";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Spring Boot

Explication

- `@Constraint` permet de lier l'annotation à la classe `UppercaseValidator`.
- `@Target` définit les éléments auxquels le validateur peut s'appliquer (champs, méthodes, paramètres).
- `@Retention` définit que l'annotation sera accessible à l'exécution.
- `Class<?>[] groups() default {}` : obligatoire mais vide par défaut (**JSR** 303 et **JSR** 380).
- `Class<? extends Payload>[] payload() default {}` : obligatoire mais vide par défaut (**JSR** 303 et **JSR** 380).

Spring Boot

Et la classe `UppercaseValidator`

```
package com.example.demo.validators;

import jakarta.validation.ConstraintValidator;
import jakarta.validation.ConstraintValidatorContext;

public class UppercaseValidator implements ConstraintValidator<Uppercase, String> {

    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        if (value == null) {
            return true;
        }
        return value.equals(value.toUpperCase());
    }
}
```

Spring Boot

Utilisons `@Uppercase` dans l'entité `Personne`

```
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@RequiredArgsConstructor
public class Personne {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long num;
    @NonNull
    @Size(min = 2)
    @NotEmpty(message = "le champ nom est obligatoire")
    @Uppercase
    private String nom;
    @NonNull
    @NotEmpty(message = "le champ prénom est obligatoire")
    @Size(min = 2)
    private String prenom;
    @NonNull
    @ManyToMany(cascade = { CascadeType.MERGE, CascadeType.REFRESH }, fetch = FetchType.EAGER)
    private List<Adresse> adresses;
}
```

Spring Boot

Ou avec un texte plus précis

```
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@RequiredArgsConstructor
public class Personne {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long num;
    @NonNull
    @Size(min = 2)
    @NotEmpty(message = "le champ nom est obligatoire")
    @Uppercase(message = "le champ nom doit être en majuscule")
    private String nom;
    @NonNull
    @NotEmpty(message = "le champ prénom est obligatoire")
    @Size(min = 2)
    private String prenom;
    @NonNull
    @ManyToMany(cascade = { CascadeType.MERGE, CascadeType.REFRESH }, fetch = FetchType.EAGER)
    private List<Adresse> adresses;
}
```

Spring Boot

Vérifiez l'ajout de cette nouvelle personne déclenche une erreur

- Utilisez **Postman** en précisant la méthode `POST`
- Dans `Headers`, ajoutez les clés `Accept` et `Content-Type` avec la valeur `application/json`
- Dans `Body`, cochez `raw` et sélectionnez `JSON`
- Ajoutez l'objet **JSON** suivant

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

- Envoyez