Création de Services REST avec Spring

Introduction

Dans ce cours, nous allons explorer comment créer une API REST avec Spring, en utilisant le framework Spring Boot. Nous nous baserons sur un exemple de contrôleur pour une entité Personne.

Qu'est-ce que REST ?

Avant de commencer à créer un serveur API, je souhaite parler un peu de REST. REST est un ensemble de conventions tirant parti du protocole HTTP pour fournir un comportement CRUD (Créer, Lire, Mettre à jour et Supprimer) sur des données et des collections de données. Le comportement CRUD correspond bien aux verbes des méthodes du protocole HTTP, comme suit :

Action	Méthode HTTP	Description
Créer	POST	Créer un nouvel objet
Lire	GET	Lire les informations d'un objet existant
Mettre à Jour	PUT	Mettre à jour les informations sur un objet existant
Supprimer	DELETE	Supprimer un objet

Vous pouvez effectuer ces actions sur une ressource. Il est utile de penser à une ressource comme à un objet auquel un identifiant unique peut s'appliquer : une personne, une commande, l'adresse d'une personne, par exemple. Cette conception de la ressource s'aligne bien avec une URL (Uniform Resource Locator). Une URL doit identifier une ressource unique sur le Web. Avec ces deux concepts, nous disposons d'un moyen d'entreprendre des actions d'application communes sur quelque chose qui est identifié de manière unique sur le Web.

Création d'un service REST

Le service REST que nous allons créer donnera accès à la base de données contenant la table personnes à travers des URL. Avec Spring, l'utilisation d'une base de données implique souvent l'usage d'Hibernate. Il est donc nécessaire de créer un projet avec les dépendances suivantes : Spring Web, MySQL Driver et Spring Data JPA.

Comme avec Hibernate, nous avons besoin d'une entité, qui représente les données en mémoire, et d'un repository qui fait le lien avec la base de données (selon le pattern DAO).

Selon les actions CRUD à exécuter, nous devrons créer des instances d'objets Personne. Il nous faut un constructeur qui initialise tous les attributs. Cependant, pour que Spring puisse gérer la classe avec ses beans, il est également nécessaire de coder un constructeur par défaut.

L'Entité

```
@Entity
@Table(name = "personnes")

public class Personne {
    private @Id @GeneratedValue Long id;
    private String nom;
    private String email;



    //Constructeur par défaut pour SPRING
    public Personne(){

    }

    //Constructeur surchargé pour initialiser les attributs
    public Personne(String nom, String email){
        this.nom = nom;
        this.email = email;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

Le Repository

```
public interface PersonneRepository extends JpaRepository<Personne, Long> {

}
```

Création d'un Contrôleur REST avec Spring

Spring Boot simplifie la création d'applications basées sur des microservices et des API REST. Examinons les composants clés de notre exemple :

@RestController

@RestController est une annotation Spring qui indique que la classe est un contrôleur où chaque méthode renvoie un objet de domaine au lieu d'une vue. Elle combine @Controller et @ResponseBody.

Le Contrôleur de Service

```
@RestController
public class ServiceController {

    private final PersonneRepository repo;

    public ServiceController(PersonneRepository pr){
        repo = pr;
    }
```

Dans cet exemple, ServiceController est un contrôleur REST qui utilise PersonneRepository pour interagir avec la base de données.

Opérations CRUD

Lire les Ressources (GET)

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

    @GetMapping("/rest/personnes/{id}")
    public Personne getPersonne(@PathVariable long id){
        return repo.findById(id).orElse(new Personne("Inconnu", "Inconnu"));
    }
```

Ces méthodes gèrent les requêtes GET pour récupérer toutes les personnes ou une personne spécifique par son ID.

Les "variables path" dans un contrôleur Spring font référence à une technique utilisée pour extraire des valeurs directement à partir de l'URI (Uniform Resource Identifier) d'une requête HTTP. Ces valeurs sont souvent utilisées pour identifier une ressource spécifique sur laquelle une opération doit être effectuée. L'annotation @PathVariable dans Spring MVC est utilisée pour capturer ces valeurs.

Créer une Ressource (POST)

```
    @PostMapping("/rest/personnes")
    public Personne addPersonne(@RequestBody Personne newPersonne){
        return repo.save(newPersonne);
    }
```

Cette méthode gère la création d'une nouvelle personne. @RequestBody indique que le paramètre doit être lié au corps de la requête.

Mettre à jour une Ressource (PUT)

```
    @PutMapping("/rest/personnes/{id}")
    public Personne updateFruit(@RequestBody Personne newPersonne, @PathVariable
long id){
        return repo.findById(id)
                .map(personne -> {
                    //classe personne créée avec un id
```

```
                    personne.setNom(newPersonne.getNom());
                    personne.setEmail(newPersonne.getEmail());

                    return repo.save(personne);
                })
                .orElseGet(() -> {
                    newPersonne.setId(id);
                    return repo.save(newPersonne);
                });
    }
```

Cette méthode gère la mise à jour d'une personne existante en base de données.

Supprimer une Ressource (DELETE)

```
    @DeleteMapping("/rest/personnes/{id}")
    public void delPersonne(@PathVariable long id){
        repo.deleteById(id);
    }
```

Cette méthode permet de supprimer une personne par son ID.

Test du service

Pour tester le service, il faut, une fois de plus, compiler et exécuter le JAVA.

Le service est accessible à l'URL **http://localhost:8080/**.

Pour tester un service REST, des outils comme Postman sont extrêmement utiles. Postman est une application populaire qui permet de tester les API en envoyant des requêtes HTTP et en examinant les réponses. Pour cela, il faut tester des URL tel que **http://localhost:8080/rest/personnes** avec les bonnes méthodes.

Création d'un Client avec Spring

Ce client va consommer le service REST développé précédemment. Il n'y aura donc pas d'accès direct à la base de données, puisque toutes les interactions se feront à travers les URL du service REST.

Pour le nouveau projet, nous n'aurons besoin que des dépendances suivantes : *Thymeleaf et Spring Web*.

Afin d'éviter tout conflit de port avec le service, nous devrons modifier le numéro de port dans le fichier application.properties du client.

```
server.port=8081
```

Nous aurons également besoin de modéliser les entités Personne sous forme d'objets Java. Nous allons donc créer une classe Personne, mais sans les annotations JPA/Hibernate, car elle ne sera pas utilisée pour l'accès direct à la base de données mais seulement pour représenter les données reçues du service REST.

Le contrôleur du Client

RestTemplate est une classe fournie par Spring qui simplifie la communication avec les services HTTP et facilite la consommation des services RESTful. Elle offre une série de méthodes pratiques pour interagir avec des services web, notamment pour envoyer des requêtes HTTP et recevoir des réponses.

Dans notre ClientController, nous commençons par déclarer une instance de RestTemplate :

```
@Controller
public class ClientController {
   private RestTemplate restTemplate;
```

Après avoir initialisé notre ClientController, nous allons maintenant approfondir comment effectuer différentes opérations CRUD à travers notre client REST.

Envoyer une Requête GET

```
    @GetMapping("/")
    public String index(Model model){
        this.restTemplate = new RestTemplate();
        String url = "http://localhost:8080/rest/personnes";
        ResponseEntity<List<Personne>> response = restTemplate.exchange(url,
HttpMethod.GET, null, new ParameterizedTypeReference<List<Personne>>() {});

        List<Personne> personnes = response.getBody();
        model.addAttribute("personnes", personnes);

        return "index";
    }
```

Dans cet exemple, une requête GET est envoyée à l'URL spécifiée. La méthode exchange est utilisée ici pour plus de flexibilité, permettant de spécifier le type de la méthode HTTP, les en-têtes éventuels, et le type de l'objet de réponse.
La réponse retournée par RestTemplate est traitée pour obtenir les données souhaitées. Dans l'exemple ci-dessus, response.getBody() renvoie la liste des objets Personne récupérés.

Afficher les Détails d'une Personne

La méthode getPersonne récupère les détails d'une personne spécifique en utilisant son identifiant. L'annotation @PathVariable est utilisée pour extraire l'ID de l'URL.

```
    @GetMapping("/personnes/{id}")
    public String getPersonne(Model model, @PathVariable long id){
        this.restTemplate = new RestTemplate();
        String url = "http://localhost:8080/rest/personnes/{id}";
        ResponseEntity<Personne> response = restTemplate.getForEntity(url,
Personne.class, id);

        Personne personne = response.getBody();
        model.addAttribute("personne", personne);

        return "personne";
    }
```

Pour récupérer des données, RestTemplate offre plusieurs méthodes. L'une des plus utilisées est getForObject ou getForEntity.

Formulaire pour Ajouter ou Mettre à Jour une Personne

Pour ajouter ou mettre à jour une personne, nous fournissons un formulaire. La méthode formPersonne affiche un formulaire vide pour créer une nouvelle personne, tandis que majPersonne charge les détails d'une personne existante dans le formulaire pour la mise à jour.

```
    @GetMapping("/personnes/form/add")
    public String formPersonne(Model model)
    {
```

```
        Personne personne = new Personne();
        model.addAttribute("personne", personne);

        return "form";
    }

    @GetMapping("/personnes/maj/{id}")
    public String majPersonne(Model model, @PathVariable long id)
    {
        this.restTemplate = new RestTemplate();
        String url = "http://localhost:8080/rest/personnes/{id}";
        ResponseEntity<Personne> response = restTemplate.getForEntity(url,
Personne.class, id);

        Personne personne = response.getBody();
        model.addAttribute("personne", personne);

        return "form";
    }
```

Supprimer une Personne

La méthode delPersonne supprime une personne spécifique. La méthode delete de RestTemplate est utilisée ici.

```
    @GetMapping("/personnes/del/{id}")
    public String delPersonne(Model model, @PathVariable long id)
    {
        this.restTemplate = new RestTemplate();
        String url = "http://localhost:8080/rest/personnes/{id}";
        restTemplate.delete(url, id);

        return "redirect:/";
    }
```

Mettre à Jour ou Ajouter une Personne

Les méthodes updatePersonne et addPersonne gèrent la soumission du formulaire pour la mise à jour ou l'ajout d'une personne. Elles utilisent respectivement exchange pour la requête PUT et postForEntity pour la requête POST.

```
    @PostMapping("/personnes/maj/{id}")
    public String updatePersonne(@ModelAttribute("personne") Personne personne,
@PathVariable long id){
        this.restTemplate = new RestTemplate();
        String url = "http://localhost:8080/rest/personnes/{id}";
        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_JSON);
        HttpEntity<Personne> request = new HttpEntity<>(personne, headers);

        ResponseEntity<Personne> response = restTemplate.exchange(url,
HttpMethod.PUT, request, Personne.class, id);

        return "redirect:/";
    }

    @PostMapping("/personnes/form/add")
    public String addPersonne(@ModelAttribute("personne") Personne personne){
        this.restTemplate = new RestTemplate();
        String url = "http://localhost:8080/rest/personnes";
```

```
        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_JSON);

        HttpEntity<Personne> request = new HttpEntity<>(personne, headers);

        ResponseEntity<Personne> response = restTemplate.postForEntity(url,
request, Personne.class);

        return "redirect:/";
    }
```

Les HttpHeaders permettent également d'ajouter divers en-têtes HTTP personnalisés à la requête. Ces en-têtes peuvent inclure des informations d'authentification (comme les jetons JWT), des directives de cache, des informations sur le client, etc.

Les pages Web

Il n'y a que 3 pages web : l'index, le formulaire et la fiche personne.

L'index

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Liste des personnes</title>
</head>

<body>
      <h1>Liste des personnes</h1>
      <ul>
          <li th:each="personne:${personnes}" ><a
th:href="@{/personnes/{id}(id=${personne.id})}" th:text="${personne.nom} + ' - ' +
${personne.email}"></a></li>
      </ul>

      <a href="./personnes/form/add">Nouvelle personne</a>
</body>
</html>
```

th:href="@{/personnes/{id}(id=${personne.id})}" créée des url dynamiquement à partir des id des objets.

La ficher personne

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>personne</title>
</head>

<body>
  <section th:object="${personne}">
    <h1 th:text="${personne.nom}"></h1>
    <p th:text="${personne.email}"></p>

    <p><a th:href="@{/personnes/maj/{id}(id=${personne.id})}">modifier</a> </p>
    <p><a th:href="@{/personnes/del/{id}(id=${personne.id})}">supprimer</a> </p>
```

```
  </section>
</body>
</html>
```

Affiche tout simplement les données de la personne.

Le formulaire

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>formulaire</title>
</head>
<body>
  <h1> Formulaire personne</h1>

  <form method="post" th:object="${personne}">
      <label>nom</label>
      <input type="text" name="nom" id="nom" th:field="${personne.nom}">

      <label>origine</label>
      <input type="text" name="email" id="email" th:field="${personne.email}">

      <button type="submit">Valider</button>
  </form>

    <a href="/">Accueil</a>

</body>
</html>
```

Il est commun à tous les appels. C'est le contrôleur qui se charge de le préremplir en cas de modification. Notez que le formulaire n'a pas d'attribut action, ce qui permet de le renvoyer vers l'URL qui l'affiche, distinguant ainsi simplement l'ajout de la modification.

Conclusion

La création d'une API REST avec Spring Boot est un processus relativement simple et direct. Spring Boot fournit de nombreux outils et annotations pour gérer les requêtes HTTP, interagir avec la base de données et structurer votre application de manière efficace.

De plus, nous avons vu comment créer un client REST en utilisant Spring et RestTemplate pour interagir avec un service REST. Nous avons exploré comment envoyer des requêtes GET, POST, PUT et DELETE pour réaliser des opérations CRUD sur les ressources du service.

Ce cours vous a donné un aperçu de la création d'une API REST simple avec Spring. Vous pouvez étendre ces concepts pour créer des applications plus complexes et robustes.