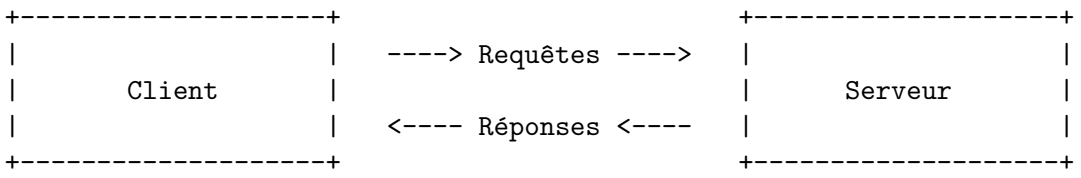
Mise en place d'une API Rest avec Spring

1 Les services WEB

Nous avons maintenant besoin de généraliser cette connexion et d'améliorer **l'hétérogénéité de nos solutions**. Pour ce faire, nous allons introduire la notion de **Services WEB** (**Web Services**) qui consiste à utiliser des documents XML pour coder les requêtes et les réponses qui transitent entre le client et serveur via le protocole HTTP :



Ces WS peuvent publier des données et/ou permettre aux clients de modifier les données métiers. Ces WS échangent souvent des documents XML sur la base du protocole **Soap**. Le protocole SOAP est géré dans JEE par l'API **JAX-WS**.

2 Les services REST

Depuis quelques années, les WS se démocratisent et se simplifient. Afin d'éviter le recours à XML, des auteurs ont suggéré d'utiliser toute la **puissance du protocole HTTP** pour coder les requêtes en utilisant à la fois :

- la méthode HTTP (`GET`, `POST`, `PUT`, `DELETE`, ...) pour coder la nature de la requête,
- les paramètres HTTP placés dans l'entête ou même dans l'URI pour préciser les arguments de la requête. Par exemple

  ```
  GET http://monservice.fr/ws/produit/A255/prix/euros
  ```

  pour obtenir le prix en euros du produit identifié par le code `A255`. Le résultat peut être codé en mode texte, en XML ou par le langage **JSON**. Nous venons de définir les services **REST** (representational state transfer). Ils sont gérés dans JEE par l'API **JAX-RS**.

3 Préalables

⏩ Travail à faire :

- Reprenez le projet du TP précédent.
- créez le package `myboot.app3.model` : pour les données.
- créez le package `myboot.app3.dao` : pour les dépôts.
- créez le package `myboot.app3.web` : pour les contrôleurs.
- créez le package `myboot.app3.test` dans le répertoire `test`.

4 Un premier service WEB

Commencons par un service REST très simple :

```java
package myboot.app3.web;

import java.util.Arrays;
import java.util.List;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api")
public class HelloRestController {

    @GetMapping("/hello")
    public String hello() {
        return "Hello";
    }

    @GetMapping(value = "/list")
    public List<Integer> list() {
        return Arrays.asList(10, 20, 30);
    }

}
```

Travail à faire :

- Testez ce service dans votre navigateur (**Chrome** et **Firefox**).
- Dans **Chrome**, ajoutez l'extension **APIs Hub Rest Client** afin de tester plus facilement votre API.
- Testez cette API avec le client HTTP `curl` :

```
curl -X "GET" "http://localhost:8081/api/hello"
```

- Ajoutez la fonction suivante afin de traiter les paramètres placés dans l'URI :

```java
@GetMapping("/hello/{message}")
public String helloWithMessage(@PathVariable String message) {
    return "Hello␣" + message;
}
```

- Ajoutez la fonction suivante afin de tester l'utilisation du `ResponseEntity`. Explorer la JavaDoc de cette classe.

```java
@GetMapping(value = "/hello2")
public ResponseEntity<String> hello2() {
    return ResponseEntity.ok("Hello");
}
```

- Construisez une entrée `/api/notFound` qui renvoie une code http `NOT_FOUND` (avec la méthode `notFound` de `ResponseEntity`).
- Faites la même chose pour `/api/noContent`.

4.1 Tester une API REST en créant un client

⏩ Travail à faire :

- Avec les `RestTemplate`, construisez un test unitaire pour ces **six points** d'entrée.
- Aidez-vous des informations sur la récupération des listes.

4.2 Tester une API REST avec MockMvc

Nous ne sommes pas obligé de lancer le serveur pour tester une API et nous pouvons utiliser la classe `MockMvc`. Essayez l'exemple ci-dessous :

```
package myboot.app3.test;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.servlet.MockMvc;

@SpringBootTest
@AutoConfigureMockMvc
public class TestHelloRestApiMockMvc {

    @Autowired
    private MockMvc mvc;

    @Test
    public void testHello() throws Exception {
        mvc.perform(get("/api/hello")).andDo(print())//
                .andExpect(status().isOk())//
                .andExpect(content().string("Hello"));
    }

    @Test
    public void testHelloJohn() throws Exception {
        mvc.perform(get("/api/hello/john")).andDo(print())//
                .andExpect(status().isOk())//
                .andExpect(content().string("Hello␣john"));
    }

    @Test
    public void testList() throws Exception {
        mvc.perform(get("/api/list")).andDo(print())//
                .andExpect(status().isOk())//
                .andExpect(content().json("[10,20,30]"));
    }

}
```

4.3 Utiliser les headers

Voila un exemple de traitement des entêtes dans des requêtes REST :

```java
@GetMapping(value = "/headers")
public ResponseEntity<String> headers(@RequestHeader String myHeader) {
    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.set("resultHeader", myHeader.toUpperCase());
    var res = ResponseEntity.ok()//
            .headers(responseHeaders)//
            .header("xx", "yy")//
            .body("HEADER␣" + myHeader);
    return res;
}
```

⏩ **Travail à faire :** Préparez un test unitaire avec la méthode `exchange` d'un `RestTemplate` et le code ci-dessous pour préparer, envoyer et récupérer des entêtes :

```java
    HttpHeaders headers = new HttpHeaders();
    headers.add("myHeader", "myHeaderValue");
    HttpEntity entity = new HttpEntity(headers);
    ResponseEntity<String> response = restTemplate.exchange(url,
        HttpMethod.GET, entity, String.class);
```

5 Utiliser des javaBeans

Nous allons maintenant construire une API REST autour de l'entité `Movie` utilisée dans le premier TP :

```java
package myboot.app3.web;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import myboot.app1.dao.MovieRepository;
import myboot.app1.model.Movie;

@RestController
@RequestMapping("/api")
public class MovieRestController {

    @Autowired
    MovieRepository repo;

    @GetMapping("/movies")
    public Iterable<Movie> getMovies() {
        return repo.findAll();
    }

    @GetMapping("/movies/{id}")
    public Movie getMovie(@PathVariable int id) {
        return repo.findById(id).get();
    }

}
```

⏩ Travail à faire :

- Préparez un test unitaire pour vérifier ces deux routes.
- Préparez un test unitaire pour `/movies/ID` sur un film inconnu.
- Faites les mêmes tests avec le client `curl` en ligne de commande afin de visualiser le code `JSON`.

5.1 Contrôler les données renvoyés

⏩ Travail à faire :

- Avec ces explications, ajoutez des annotations `@JsonView` afin d'éviter que le champ `description` ne soit renvoyée. Vous devrez également déclarer une annotation `@JsonView` sur le contrôleur pour indiquer quelle est la vue à utiliser pour sérialiser l'instance de `Movies`.
- Vous pouvez également explorer cette présentation afin de mieux comprendre les annotations JSON. Utilisez notamment `@JsonIgnore` pour supprimer une information. Explorez (au travers de test), quelques autres possibilités.

5.2 Supprimer des données

Ajoutez la nouvelle route ci-dessous :

```
@DeleteMapping("/movies/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
void deleteMovie(@PathVariable int id) {
    repo.deleteById(id);
}
```

⏩ Travail à faire :Vérifiez par un test unitaire son bon fonctionnement.

5.3 Ajouter des données

Ajoutez la nouvelle route ci-dessous :

```
@PostMapping("/movies")
public Movie postMovie(@RequestBody Movie m) {
    repo.save(m);
    return m;
}
```

⏩ Travail à faire :

- Vérifiez par un test unitaire son bon fonctionnement.
- Essayez d'ajouter un film avec `curl` (exemple ci-dessous à adapter).

```
curl -X POST -H "Content-Type:␣application/json" \
    -d '{"name":␣"John",␣"email":␣"john@example.com"}' \
    https://example/contact
```

- Ajoutez l'annotation `@Valid` à l'argument `Movie`. Les films à ajouter doivent maintenant être valides (en fonction des annotations de validation). Tentez, dans un TU, d'enregister un film invalide.

5.4 Modifier des données

En respectant la même forme, traitez la requête `PUT /movies` qui va mettre à jour un film. Il faut traiter le cas de la mise à jour d'un film qui n'existe pas en BD (je vous conseille d'utiliser la méthode `orElseThrow` du résultat `Optional` renvoyé par la méthode Spring Boot `findById` - plus d'informations).

⏩ **Travail à faire :**Ajoutez un test unitaire pour valider cette modification.

5.5 Améliorer le traitement des erreurs

Pour l'instant, les requêtes `GET /movies/ID` et `PUT /movies` sur des ID erronés provoquent une erreur interne qui n'est pas très lisible. Définissez l'exception ci-dessous et utilisez-là (avec la clause `orElseThrow`) pour que ces deux routes provoquent des erreurs `NOT_FOUND`.

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
class MovieNotFoundException extends RuntimeException {
    private static final long serialVersionUID = 1L;
}
```

⏩ **Travail à faire :**Modifiez en conséquence les tests unitaires.

ℹ **Note :Nouveautés 2024.** Pour générer un code `NOT_FOUND` si le film n'existe pas, il maintenant préférable que les méthodes `getMovie()/putMovie()` renvoient une instance de `ResponseEntity <Movie>`. Ainsi, il est facile d'utiliser la méthode statique `of` de `ResponseEntity` pour renvoyer l'`Optional` obtenu via Spring-Data.

5.6 Filtrer des données

Modifiez le traitement de la route `GET /movies` afin d'ajouter des fonctions de filtrage sur des paramètres optionnels comme

```
GET /movies?name=Fred&year=1999
```

⏩ **Travail à faire :**Prenez en compte la présence de l'un des critères ou aucun des deux. Je vous conseille de prévoir des valeurs par défaut pour les paramètres.

6 Traitement des relations

6.1 Préparer des données

Nous allons nous baser sur les entités ci-dessous :

```java
package myboot.app3.model;

import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

import jakarta.persistence.Basic;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.OneToMany;
import jakarta.persistence.OrderColumn;

import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.ToString.Exclude;

@Entity
@Data
@NoArgsConstructor
public class Writer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;

    @Basic(fetch = FetchType.LAZY)
    private String description;

    @OneToMany(fetch = FetchType.LAZY, mappedBy = "writer")
    @Exclude
    private List<Post> posts = Arrays.asList();

    @OneToMany(fetch = FetchType.LAZY, mappedBy = "writer")
    @OrderColumn(name = "position")
    @Exclude
    private List<Comment> comments = new LinkedList<>();

    public Writer(String name, String email, String description) {
        this.name = name;
        this.email = email;
        this.description = description;
    }

}
```

```java
package myboot.app3.model;

import java.util.LinkedList;
import java.util.List;

import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.OneToMany;

import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.ToString.Exclude;

@Entity
@Data
@NoArgsConstructor
public class Post {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String subject;

    @OneToMany(mappedBy = "post", fetch = FetchType.LAZY)
    @Exclude
    private List<Comment> comments = new LinkedList<>();

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn
    @Exclude
    private Writer writer;

    public Post(String subject, Writer writer) {
        super();
        this.subject = subject;
        this.writer = writer;
    }

}
```

```java
package myboot.app3.model;

import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;

import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.ToString.Exclude;

@Entity
@Data
@NoArgsConstructor
public class Comment {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String reply;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn
    @Exclude
    private Post post;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn
    @Exclude
    private Writer writer;

    public Comment(String reply, Post post, Writer writer) {
        super();
        this.reply = reply;
        this.post = post;
        this.writer = writer;
    }

}
```

Nous allons également définir les dépôts nécessaires :

```java
package myboot.app3.dao;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import myboot.app3.model.Writer;

@Repository
@Transactional
public interface WriterRepository extends CrudRepository<Writer, Long> {

}
```

```java
package myboot.app3.dao;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import myboot.app3.model.Post;

@Repository
@Transactional
public interface PostRepository extends CrudRepository<Post, Long> {

}
```

```java
package myboot.app3.dao;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import myboot.app3.model.Comment;

@Repository
@Transactional
public interface CommentRepository extends CrudRepository<Comment, Long> {

}
```

6.2 Traiter les relations

Faites en sorte de créer des données qui comportent des boucles (`Writer1 -> Post1 -> Comment1 --> Writer1`) et d'autres pas (`Writer2 -> Post2 -> Comment2 -> Writer3`).

- Commencez par prévoir un contrôleur pour traiter la route `GET /writers/ID`.
- Quel résultat obtenez-vous ? Pouvez-vous récupérer les publications de tous les utilisateurs ?
- Utilisez les annotations `@JsonIgnore` pour régler le problème des boucles.
- Recommencez en utilisant `@JsonBackReference` (sur les liaisons inverses) et `@JsonManagedReference` (sur les collections) pour bien traiter les relations. Vérifiez par un test unitaire que la récupération d'un utilisateur, entraine aussi les publications et les commentaires.
- Les boucles peuvent être gérées par l'ajout d'une annotation qui précise l'identifiant d'une instance. Testez la sérialisation/de-sérialisation de la classe ci-dessous (avec `L1->L2->L1`). Inspirez vous de ces exemples sans forcément mettre en place un service REST.

```java
package myboot.app3.model;

import com.fasterxml.jackson.annotation.JsonIdentityInfo;
import com.fasterxml.jackson.annotation.ObjectIdGenerators;

import lombok.Data;
import lombok.NoArgsConstructor;

@JsonIdentityInfo(//
        generator = ObjectIdGenerators.PropertyGenerator.class, //
        property = "name"//
)
@Data
@NoArgsConstructor
public class Loop {

    private String name = "";

    private Loop loop;

    public Loop(String name) {
        super();
        this.name = name;
    }

}
```