# *Mise en place d'une API Rest avec Spring*

## 1 Créer un client en Vue.js

Nous avons déjà créé des clients en ligne de commande ( `curl` ) et en Java ( `RestTemplate` ). Nous allons maintenant créer une application WEB côté client en **Vue.js**[1] et Axios (pour le lancement des requêtes Rest).

### 1.1 Préalables

- Reprenez le projet du TP précédent.
- Ajoutez au fichier `pom.xml` les dépendances suivantes :

```
<!-- Pour Vue.js -->
<dependency>
    <groupId>org.webjars.npm</groupId>
    <artifactId>vue</artifactId>
    <version>3.2.19</version>
</dependency>
<!-- Pour Axios -->
<dependency>
    <groupId>org.webjars.npm</groupId>
    <artifactId>axios</artifactId>
    <version>0.22.0</version>
</dependency>
```

- Ajoutez au fichier `headers.jsp` les clauses suivantes (pour que nos pages JSP soient équipées de ces deux nouvelles librairies) :

```
<c:url var="vue_js" value="/webjars/vue/3.2.19/dist/vue.global.js" />
<c:url var="axios_js" value="/webjars/axios/0.22.0/dist/axios.min.js" />

...
<script src="${vue_js}"></script>
<script src="${axios_js}"></script>
...
```

- Modifiez le fichier `application.properties` en ajoutant la ligne ci-dessous. Nous allons utiliser des ressources statiques (fichier Javascript) et il est nécessaire que le navigateur ne conserve pas de cache (afin de toujours utiliser la dernière version).

```
spring.resources.cache.cachecontrol.max-age=0
```

- Préparez un fichier vide afin d'accueillir notre code javascript :

src/main/resources/static/app.js

```
// pour le code Javascript
```

- Préparez le fichier JSP ci-dessous :

1. https ://vue3-fr.netlify.app/

src/main/webapp/WEB-INF/jsp/app.jsp

```
<%@ include file="/WEB-INF/jsp/header.jsp"%>

<c:url var="home" value="/aaa" />
<c:url var="app" value="/app.js" />

<div id="myApp">
    <div class="container">
        <h1>My application</h1>
        <p>Hello.</p>
    </div>
</div>
<script src="${app}"></script>

<%@ include file="/WEB-INF/jsp/footer.jsp"%>
```

- Préparez un contrôleur pour l'accès à cette page :

```
package myboot.app4.web;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller()
public class VueAppController {

    @RequestMapping(value = "/app")
    private ModelAndView hello() {
        return new ModelAndView("app");
    }

}
```

- Vérifiez que la ressource `/app` est accessible ainsi que le fichier javascript ( `/app.js` ).

## 1.2 Très rapide introduction à Vue.js

Modifions nos fichiers :

app.js

```
const myApp = {

    // Préparation des données
    data() {
        console.log("data");
        return {
            counter: 1,
            message: "Hello",
            list: [10, 20, 30],
            axios: null,
        }
    },

    // Mise en place de l'application
    mounted() {
        console.log("Mounted␣");
        this.axios = axios.create({
            baseURL: 'http://localhost:8081/api/',
            timeout: 1000,
            headers: { 'Content-Type': 'application/json' },
        });
    },

    methods: {
        // Place pour les futures méthodes
    }
}

Vue.createApp(myApp).mount('#myApp');
```

L'application Vue.js est créée par la dernière ligne. Elle va gérer l'élément identifié par `myApp`. Elle prépare des données `data()`, elle s'initialise `mounted()`, et elle prévoie des méthodes.

app.jsp

```
...
<div id="myApp">
    <div class="container">
        <h1>My application</h1>
        <p>{{ message }}</p>
        <p>list = <span v-for="element in list">{{element}} - </span></p>
        <p>counter = {{counter}}</p>
    </div>
</div>
...
```

La page HTML (générée par la page JSP) une fois chargée par le client va initialiser Vue.js et ce dernier va remplacer `{{ message }}` par le contenu de la donnée en question. La mise à jour sera automatique si la donnée est modifiée. Vous remarquerez la facilité avec laquelle nous pouvons itérer sur une collection afin de générer plusieurs éléments HTML (affichage de la liste).

Vérifiez dans la console Javscript de votre navigateur les messages générés.

Ajoutons un bouton (en HTML) et une méthode (en JS) pour incrémenter le compteur :

app.js

```
...
    methods: {
        // Place pour les futures méthodes
        incCounter: function() {
            console.log("incremente␣le␣compteur␣");
            this.counter++;
        }
    }
...
```

app.jsp

```
...
<div id="myApp">
    <div class="container">
        <h1>My application</h1>
        <p>{{ message }}</p>
        <p>counter = {{counter}}</p>
        <p>list = <span v-for="element in list">{{element}} - </span></p>
        <button v-on:click="incCounter">Plus un</button>
    </div>
</div>
...
```

Nous pouvons donc capter les évènements JS et leur associer des méthodes Vue.js qui agissent sur les données (ce qui provoque une mise à jour). À titre d'exemple, testez le traitement ci-dessous de l'évènement `mouseover`.

app.jsp

```
...
    <span v-on:mouseover="incCounter">Il faut me survoler</span>
...
```

**À faire** : Prévoir un argument `step` à la méthode `incCounter` et mettez en place deux boutons pour augmenter le counter de 1 ou de 2.

**À faire** : Ajoutez le code suivant après l'incrémentation et vérifiez le résultat (la lecture asynchrone du résultat de la requête `GET /movies/1` provoque la mise à jour de l'affichage).

app.js

```
this.axios.get('/movies/1')
    .then(r => {
        console.log("read␣movie␣1␣done");
        this.message = r.data;
    });
```

### 1.3 Une application de gestion des films

**Travail à faire** :

- En ajoutant une donnée `movies`, en utilisant l'initialisation (`mounted()`), l'itération (`element in list`) et la récupération de données (méthode `get` d'Axios), préparez un affichage des films (inspirez vous de la présentation de la page `movies.jsp`).
- Dans cet affichage, préparez des boutons pour supprimer, éditer et visualiser les films en leur associant des méthodes vides pour l'instant. Vous pouvez prévoir un argument pour ces méthodes et émettre un message sur la console pour vérifier le bon fonctionnement.

- Implémentez la fonction de suppression en appelant l'API `DELETE /movies/ID` et en rafraîchissant la liste des films.
- Implémentez la fonction de visualisation détaillée d'un film :
  - ▷ prévoir une donnée `movie` (film à visualiser),
  - ▷ prévoir dans la page JSP/HTML un affichage de ce film,
  - ▷ prévoir dans la méthode de visualisation un chargement de ce film.

  À cette étape vous pouvez visualiser la liste des films et, en même temps, les détails d'un film. Afin d'éviter ce mélange, vous pouvez conditionner l'affichage d'un élément avec l'attribut `v-if` :

```
<div v-if="(movie != null)">
    ... détails du film ...
</div>
```

- Prévoyez un barre de navigation pour faciliter l'accès aux actions :

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
    <a class="navbar-brand" href="${home}">Movies</a>
    <a class="navbar-brand" href="#" v-on:click="populateMovies()">Populate</a>
    <a class="navbar-brand" href="#" v-on:click="listMovies()">List of movies</a>
</nav>
```

- Mettez en place la route `PATCH /movies/` pour peupler la BD avec trois nouveaux films et exploitez cette route au sein de la méthode `populateMovies`.

### Modifier les données

Nous allons maintenant éditer un film :

- prévoir une donnée `editable` (film à modifier),
- prévoir une donnée `errors` (les erreurs à afficher initialisées à `[]`),
- prévoir dans la page JSP/HTML la mise en place d'un formulaire (si `editable` n'est pas `null`) :

```
<form id="app" method="post" novalidate="true">

    <div class="form-group">
        <label>Name :</label>
        <input v-model="editable.name" class="form-control"
            v-bind:class="{'is-invalid':errors.name}" />
        <div v-if="(errors.name)" class="alert alert-warning">
            {{errors.name}}
        </div>
    </div>
    <div class="form-group">
        <label>Year :</label>
        <input v-model="editable.year" class="form-control"
            v-bind:class="{'is-invalid':errors.year}" number />
        <div v-if="(errors.year)" class="alert alert-warning">
            {{errors.year}}
        </div>
    </div>
    <div class="form-group">
        <label>Description :</label>
        <textarea v-model="editable.description" rows="5" cols="50"
            class="form-control"></textarea>
    </div>
    <div class="form-group">
        <button v-on:click.prevent="submitMovie()" class="btn btn-primary">
            Save</button>
        <button v-on:click="listMovies()" class="btn btn-primary">
            Abort</button>
    </div>
</form>
```

Cet exemple introduit le *binding* pour les attributs avec `v-bind` ainsi que l'initialisation des données du formualire avec `v-model`. [plus d'information [2]]

- Construire la méthode `submitMovie()` qui va utiliser la route `PUT /movies/` de notre API. À ce stade, les données invalides doivent provoquer le blocage du formulaire.
- Ajouter à la méthode `submitMovie()` un code JS de validation des données.

### Utiliser la validation côté serveur

Afin d'éviter une double validation (côté JS et côté serveur), nous allons utiliser la validation côté API REST. Pour ce faire, il est nécessaire de

- revoir la route `PUT /movies/` afin de maitriser la phase de validation :
  - ▷ demander l'injection de l'outil de validation :

```
@Autowired
LocalValidatorFactoryBean validationFactory;
```

  - ▷ l'utiliser pour valider le film (méthode `validate`)
  - ▷ renvoyer les erreurs de validation sur la forme :

```
{ "champ1": "message d'erreur 1", "champ2":"message d'erreur 2" }
```

- récupérer ces erreurs côté JS afin de les afficher (grâce à la donnée `errors`). Vous pouvez utiliser l'expression `(Object.keys(this.errors).length==0)` afin de vérifier l'absence d'erreur.

### Ajouter des films

2. https://vue3-fr.netlify.app/guide/template-syntax.html#interpolations

Modifiez votre application afin d'offrir la fonction d'ajout d'un nouveau film (à ajouter dans le menu).