



Cooking

Compte Rendu Client Riche : API Spoonacular

Par Antoine PETIT et Lucas VIGIER

Sommaire :

Présentation du projet :	1
Consignes :	1
Apparence du projet :	2
Accueil	2
Header	2
Filtres	2
Détails	2
Cartes de recettes	2
Organisation avec Vue.Js :	2
Index.html :	3
Différents composants :	3
search.js :	3
filtres.js :	3
app.js :	5
recipe.js :	5
detail.js :	6
Consommation de l'API :	6
omdb.js :	6
Conclusion :	7
Répartition des tâches :	7

1. Présentation du projet :

Consignes :

Vous devez proposer un client riche portant sur <https://spoonacular.com/food-api>.

Vous devrez dans ce projet utiliser la persistance locale. La richesse de votre application et sa robustesse seront évaluées.

Voici un exemple de scénario possible, je souhaite pouvoir choisir une recette en fonction de mes intolérances et de mes choix, pour la recette choisie j'aimerais avoir des informations sur les vins proposés. Le choix de la technologie JavaScript est votre.

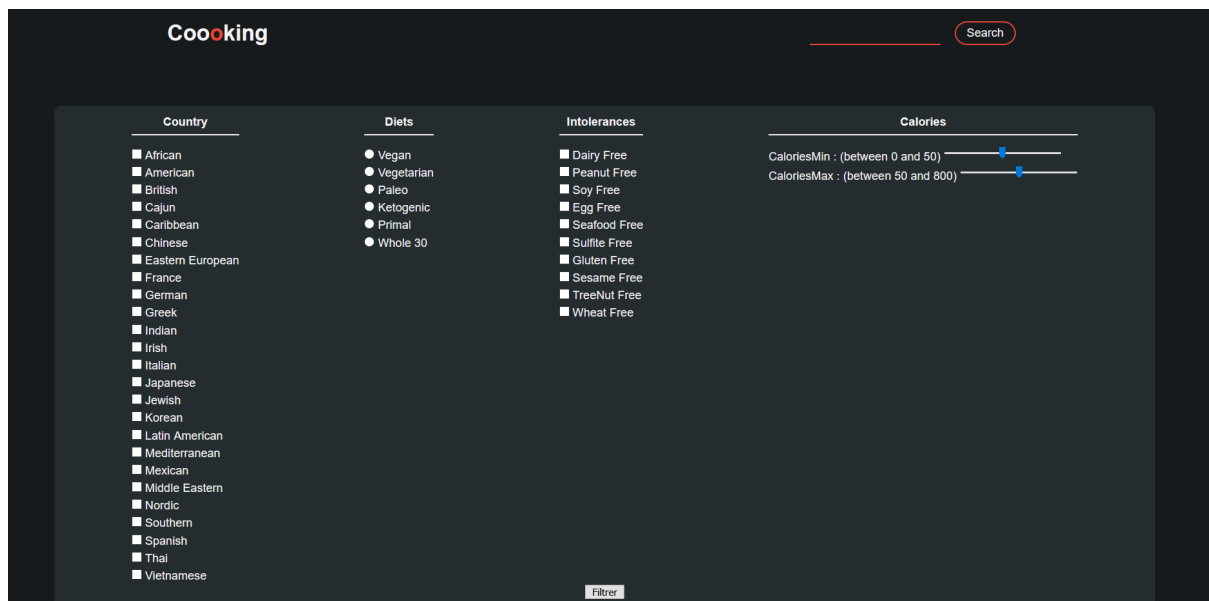
Contraintes du projet :

- Utiliser l'API Spoonacular
- Utiliser Vue Js

Lors de notre enseignement M4103 Programmation Web - Client Riche, nous avons été amené à programmer un site web utilisant un API. Nous avons donc décidés de créer le site web Cooking, ce site vous permet à l'aide de l'api Spoonacular de rechercher des recettes de cuisine selon leurs nom ou leurs caractéristiques comme leurs pays d'origine, ou selon vos régimes (vegan, végétarien, ...) mais encore selon vos intolérances.

a. Apparence du projet :

i. Accueil



The screenshot shows the 'Cooking' application interface. At the top, there is a 'Search' button. Below it, there are four main filter sections: 'Country', 'Diets', 'Intolerances', and 'Calories'. Each section contains a list of options with checkboxes or radio buttons. The 'Country' list includes African, American, British, Cajun, Caribbean, Chinese, Eastern European, France, German, Greek, Indian, Irish, Italian, Japanese, Jewish, Korean, Latin American, Mediterranean, Mexican, Middle Eastern, Nordic, Southern, Spanish, Thai, and Vietnamese. The 'Diets' list includes Vegan, Vegetarian, Paleo, Ketogenic, Primal, and Whole 30. The 'Intolerances' list includes Dairy Free, Peanut Free, Soy Free, Egg Free, Seafood Free, Sulfite Free, Gluten Free, Sesame Free, TreeNut Free, and Wheat Free. The 'Calories' section has two sliders: 'CaloriesMin : (between 0 and 50)' and 'CaloriesMax : (between 50 and 800)'. At the bottom right, there is a 'Filtrer' button.

ii. Header

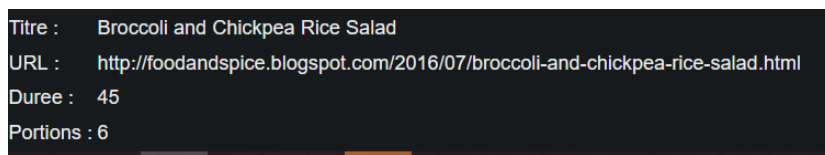


Nous avons opté pour une présentation simple, sérieuse et intuitive. Ce sont pour nous les caractéristiques qui rendent un site professionnel et simple d'utilisation.

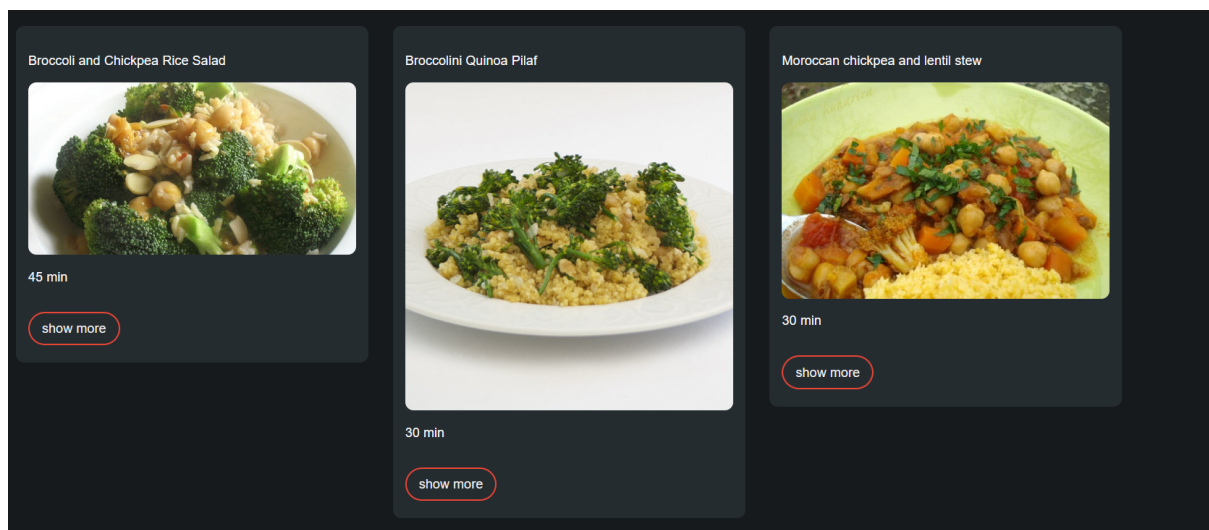
iii. Filtres

La liste des filtres étant très longue sur Spoonacular, nous avons décidé de sélectionner les filtres principaux. Ajouter d'autres filtres aurait allongé encore plus la liste sans y apporter un réel intérêt.

iv. Détails



v. Cartes de recettes



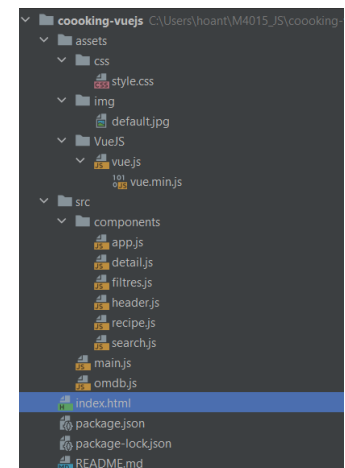
Le site affiche autant de cartes de recettes qu'il y a de résultat. les cartes sont composé de :

- La photo de la recette
- Le nom de la recette
- Le temps de préparation de la recette
- Un bouton SHOW MORE qui permet d'aller sur le site proposant la recette

b. Organisation avec Vue.Js :

Pour l'organisation des fichiers, nous avons le choix entre utiliser Vuejs CLI ou Vuejs via une balise script. Nous avons opté pour la deuxième solution car notre enseignante, nous avait toujours appris Vuejs via la balise script. Même si cela ne nous a pas empêché d'essayer la création du projet via Vuejs CLI.

Notre architecture se compose donc avec index.html, des composants js équipés de template, un fichier style.css, et des fichiers js permettant le bon fonctionnement de vuejs et de la consommation de l'api.



c. Index.html :

Le fichier Index.html contient tout le code html composant le projet, nous y avons réunis tous les templates afin de créer une certaine organisation dans nos fichiers. Ces templates sont dans des script type="text/template" id="nom du composant". C'est grâce à l'id que nous récupérons le template dans nos composants js.

2. Différents composants :

Les composants Vuejs sont composés de plusieurs parties :

- props : pour la validation des conditions pour la bonne création du composant
- template : contient le html
- data : contient les variables que nous allons utiliser
- methods : contient les fonctions du composant

a. search.js :

Le composant search contient le bouton search ainsi que la barre de recherche.

```
<form @submit.prevent="handleSubmit">
  <input type="text" v-model="recipetitle"/>
  <input type="submit" value="Search" class="btn"/>
</form>
```

L'input type text est la barre de recherche, quand le bouton search sera appuyé, nous prenons le texte de la recherche que nous mettons dans une variable recipetitle. Avec le submit.prevent, nous nous dirigeons ensuite vers la fonction handleSubmit de search.js.

```
handleSubmit: function () {
  useRecipeApi.bySearch(this.recipetitle).then(data =>
    this.$emit('search-done', data.results));
}
```

Cette fonction permet d'appeler une autre fonction contenue dans omdb.js qui nous permet de faire nos requêtes. le then data=>this.emit('search-done', data.result)); nous permet d'envoyer nos résultats de requêtes dans app.js avec @search-done="searchCompleted" qui nous permet par la suite de créer notre tableau de recette.

b. filtres.js :

filtres.js a la même fonction que search.js, ce composant permet également d'effectuer une recherche et à la fin de constituer un tableau de recette. Cependant, ce composant fonctionne différemment. En effet, chaque filtre à sa propre variable qui lui est attribuée.

```
<div class="filter">
  <input type="checkbox" id="Chinese" name="pays" v-model="Chinese">
  <label for="Chinese">Chinese</label>
</div>

<div class="filter">
  <input type="radio" id="vegetarian" name="diet" v-model="vegetarian">
  <label for="vegetarian">Vegetarian</label>
</div>
```

La variable Chinese est contenu dans filtres.js: `Chinese: ''`,

Puis pour chaque pression sur le bouton filtrer, nous allons regarder si la variable Chinese a été rempli ou non :

```
if (this.Chinese !== ''){
  pays += 'Chinese,';
```

Cette condition if fait partie des nombreuses conditions if contenues dans createRequest. Cette fonction va nous permettre de constituer au fur et à mesure notre URL pour notre requête.

```
if (pays !== ''){
  request += '&cuisine=' + pays;
```

La variable pays, nous permet de faire la liste de tous les pays sélectionnés et de les ajouter tous en même temps à l'url. Nous avons procédé de la même façon pour les intolérances.

Le deuxième cas était pour les régimes (diet), pour ce cas, nous ne pouvons pas avoir plusieurs diet dans la même requête. Chaque Diet a sa variable, cependant, nous ajoutons directement la diet à l'url et les relations entre les conditions ne sont pas des if mais des else if ce qui ne nous permet d'avoir qu'un seul ajout en sortant des conditions.

```
if (this.vegan !== ''){
  request += '&diet=vegan'
} else if (this.vegetarian !== ''){
  request += '&diet=vegetarian'
}
```

Une fois les filtres ajoutés à l'url, nous ajoutons la clé de l'api (le nombre n'est pas obligatoire) et on fait le return en donnant à this.request contenu dans data la valeur de notre url constitué de nos filtres.

```
request += '&apiKey=b51ad72a29914fb4b0ce97a4be312061&number=9'
this.request = request;
```

Quand nous appuyons sur le boutons filtrer trois fonction sont appelées :

```
handleSubmit: function () {
  this.createRequest();
  this.deletefilter();
  useRecipeApi.byFiltre(this.request).then(data =>
    this.$emit('search-done', data.results));
```

createRequest pour constituer notre url en fonction de nos filtres.
 deletefilter pour remettre nos filtres à 0 et effacer les valeurs contenues dans notre variable.
 Et la consommation de notre api via la recherche byFiltre qui va ensuite nous renvoyer nos réponses à notre requête.

c. app.js:

app.js est le composant le plus important de tous. En effet, c'est lui qui va gérer une partie de la recherche des recettes :

```
@search-done="searchCompleted"
```

```
searchCompleted: function (data) {
  if (data != undefined) {
    this.recipes = data;
    this.selectedId = null;
```

Si l'on clique sur le bouton Search ou sur le bouton filtrer, nous allons effectuer un searchCompleted sur search-done, ce qui va nous permettre de mettre toutes les réponses à nos requêtes dans le tableau recipes

```
<recipe v-for="recipe in recipes" v-bind:key="recipe.imdbID"
:recipe=recipe v-on:recipe-selected="recipeSelected($event)"></recipe>
```

v-for est un outil qui va servir à parcourir des recipes qui est un tableau de recipe (recipe = recette). Et pour chaque recipe, si la recette a le profil d'une recette acceptable, nous allons la mettre en recipe-selected, ce qui déclenche l'évènement recipeSelected.

```
recipeSelected: function (id) {
  this.selectedId = id;
```

recipeSelected permet de récupérer l'ID de notre recipe.

Cet ID nous permet de récupérer les informations de la recipe dans détail.

```
<detail v-bind:recipeId="selectedId" :key="selectedId"></detail>
```

d. recipe.js :

Le composant recipe contient l'affichage des recettes sur notre site. Chaque recipe contenu dans recipes est traité une par une grâce à l'appel de ce composant dans app.js.

Le traitement d'une recipe contient la récupération de l'image, du nom, du temps de préparation et de l'urlsorce (l'url du site qui propose la recette).

Pour le nom le temps de préparation ainsi que l'urlsorce, la commande est simple :

```
<span class="card-title"> {{recipe.title}} </span>
<p> {{recipe.readyInMinutes}} </p><br>
<a :href="recipe.sourceUrl" >show more</a>
```

Pour l'image c'est différent, en effet il faut prendre le cas où il n'y a pas d'image.

```
recipe.image == 'N/A'? 'assets/img/default.jpg'
```

Mais nous avons été confrontés à un autre problème. Quand nous faisons une requête avec des filtres, nous obtenions un URL d'image complet mais quand nous le faisons par search, nous devons rajouter <https://spoonacular.com/recipeImages/> au début de recipe image. C'est pourquoi le :src de notre image est égal à :

```
:src="recipe.image == 'N/A'? 'assets/img/default.jpg' :
recipe.image.startsWith('https://spoonacular.com/recipeImages/') ==
true? recipe.image : 'https://spoonacular.com/recipeImages/' +
recipe.image"
```

Dans un premier temps, on demande à recipe.image s'il contient quelque chose. Ensuite, on regarde si recipe.image commence par <https://spoonacular.com/recipeImages/> si oui on ne touche pas recipe.image et si non on lui rajoute <https://spoonacular.com/recipeImages/>.

Pour l'url de l'image, la différence vient des réponses à nos requêtes. En effet, si l'on effectue une requête avec Search dans l'url, l'url est donné sans

<https://spoonacular.com/recipeImages/> mais la requête contient beaucoup plus d'informations que si l'on effectue la recherche avec complexSearch.

e. detail.js :

```
detail : {
  title: null,
  sourceUrl: null,
  readyInMinutes: null,
  servings: null
},
```

La partie détail permet de stocker le titre, l'url, le temps de préparation, Props permet de vérifier la création de recipeId et Mounted se lance dès que la condition est valide. Nous récupérons donc recipe.id dans recipe.js puis la fonction effectue une requête d'information via useRecipeApi. Les informations sont ensuite envoyées dans detail.

```
props: ["recipeId"],

mounted(){
  console.log("test");
  if (this.recipeId !== undefined) {
    useRecipeApi.byId(this.recipeId).then(data => {
      this.detail = data;
      console.log(data)
    });
  }
}
```

3. Consommation de l'API :

a. omdb.js :

Pour la consommation de l'api, nous avons vu précédemment comment nous récupérons les urls ou les éléments qui vont constituer nos requêtes. Passons à présent aux envois des requêtes.

Pour la partie Search, nous récupérons dans un premier temps notre paramètre search (recipeTitle) et notre clé d'api.

```
bySearch: (search) => new Promise((resolve, reject) => {
  const RECIPE_API_URL =
`https://api.spoonacular.com/recipes/search?apiKey=${API_KEY}&number=9&q
uery=${search}`;
  fetch(RECIPE_API_URL)
    .then(response => response.json())
    .then(jsonResponse => resolve(jsonResponse))
    .catch((err) => reject(err))
})
```

Notre URL étant constituée, nous effectuons un fetch afin de récupérer la réponse de notre API dans un format json. Pour notre demande byfilter, le code est quasiment similaire sauf que nous récupérons en paramètres notre URL déjà constituée.

4. Conclusion :

Nous avons trouvé que l'utilisation de Vuejs pour un projet utilisant une API était intelligente. En effet, la documentation de Vuejs et les tutoriels étant extrêmement complets et en français, tout cela m'a grandement aidé. L'API aussi était très bien documentée, et nous avons bien compris l'utilisation des API. Vuejs est plus intuitif que d'autres frameworks vu dans d'autres projets comme Angular par exemple.

5. Répartition des tâches :

Antoine : partie JS, HTML et compte rendu

Lucas : partie CSS et HTML

Merci d'avoir pris le temps de lire tout notre compte rendu.