

Fiche d'investigation de fonctionnalité

Fonctionnalité : Recherche principale	Fonctionnalité de recherche générale #1
<p>Problématique : Afin de pouvoir donner la meilleure expérience de recherche aux utilisateurs, nous cherchons à rendre le plus fluide possible les séquences de requêtes de la barre principale dans le but d'afficher les recettes le plus rapidement possible de manière pertinente. Le but de cette fonctionnalité est de se démarquer de la concurrence pour proposer aux utilisateurs une séquence de recherche très réactive et pertinente. Il est prioritaire de prendre en compte les différents besoins des utilisateurs pour les fidéliser afin qu'ils n'aillent pas naviguer sur d'autres sites concurrents.</p> <p>En effet il est primordial d'avoir une recherche principale très performante pour offrir aux usagers le meilleur outil possible pour leur utilisation. Le site doit pouvoir être compatible avec tous les navigateurs et doit être responsive.</p>	

<p>Option 1 : On effectue un algorithme utilisant la programmation fonctionnelle avec les méthodes de l'objet array (foreach, filter, map, reduce etc....) On utilise la méthode filter pour générer un tableau de recettes. (Figure1)</p>	
<p>Avantages :</p> <ul style="list-style-type: none"> - Recherche rapide, fluide et pertinente - Compatibilité avec tous les navigateurs - Moins de répétitions, moins de lignes de code - Code plus maintenable (ESLINT, structure) - Plus d'opérations par seconde dans un temps imparti 	<p>Inconvénients :</p> <ul style="list-style-type: none"> - Lisibilité du code moins claire
<p>Nombre de caractères à remplir pour déclencher la recherche : 3</p>	

<p>Option 2 : Utilisation de boucles natives (while,for etc..) On utilise une boucle For pour l'ensemble des recettes pour voir s'il y a un match avec la valeur de la recherche. (Figure 2)</p>	
<p>Avantages :</p> <ul style="list-style-type: none"> - Lisibilité du code plus claire 	<p>Inconvénients En cours</p> <ul style="list-style-type: none"> - Moins d'opération par seconde pour le navigateur dans un temps imparti - Boucles moins performantes - Plus de lignes de codes
<p>Nombre de caractères à remplir pour déclencher la recherche : 3</p>	

<p>Solution retenue :</p> <p>En testant les 2 algorithmes dans un benchmark nous nous sommes rendu compte que la solution la plus rapide et pertinente était l'option 1 du filter qui effectue plus d'opérations par seconde après analyse des performances sur différents outils de benchmark. Nous avons donc retenu la solution de programmation fonctionnelle (filter).</p>
--

Annexes

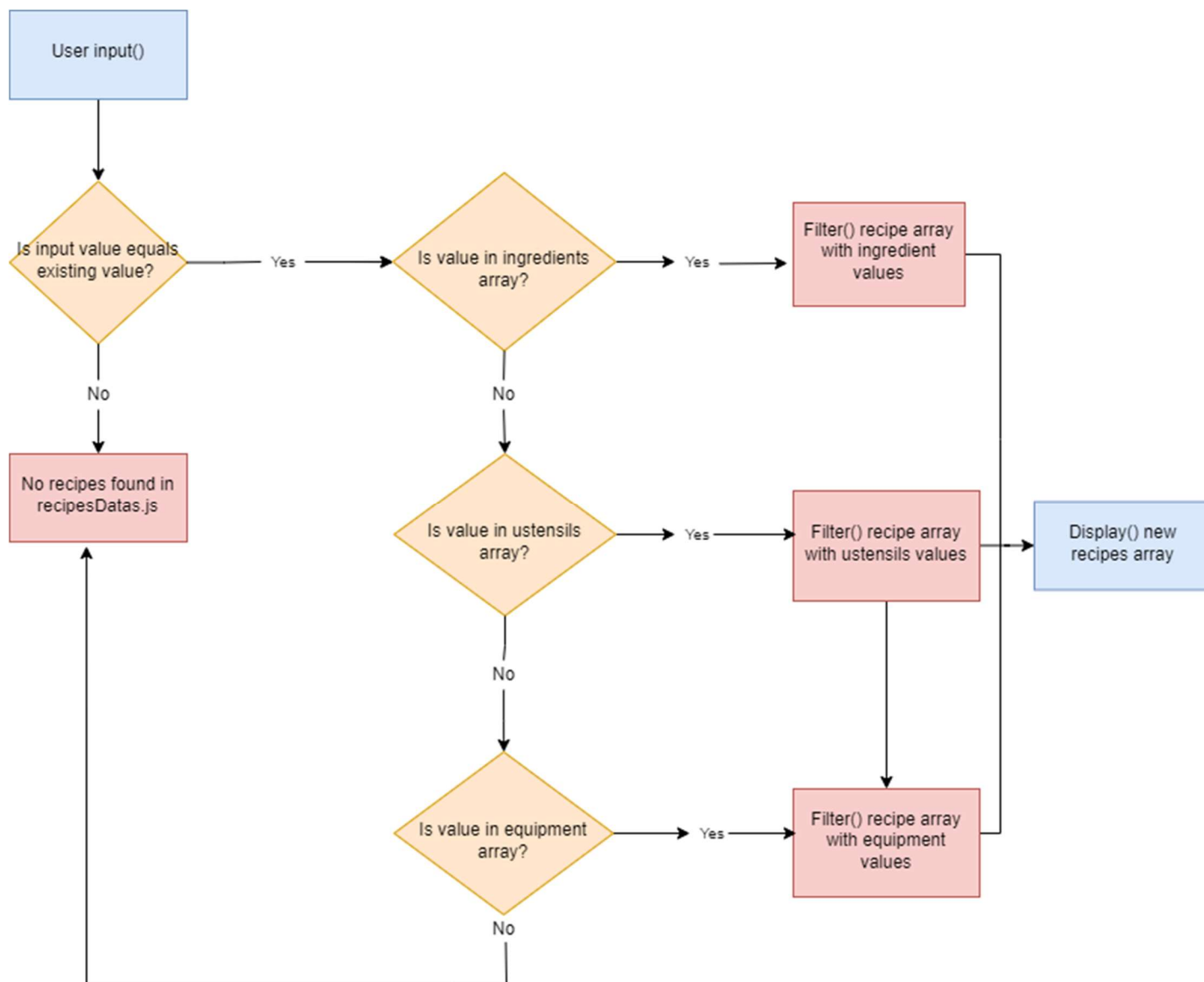


Figure 1 Algorithme – Utilisation de la méthode filter pour créer un nouveau tableau si la valeur de la recherche correspond à une valeur liée aux ingrédients, ustensiles, appareils d'une recette.

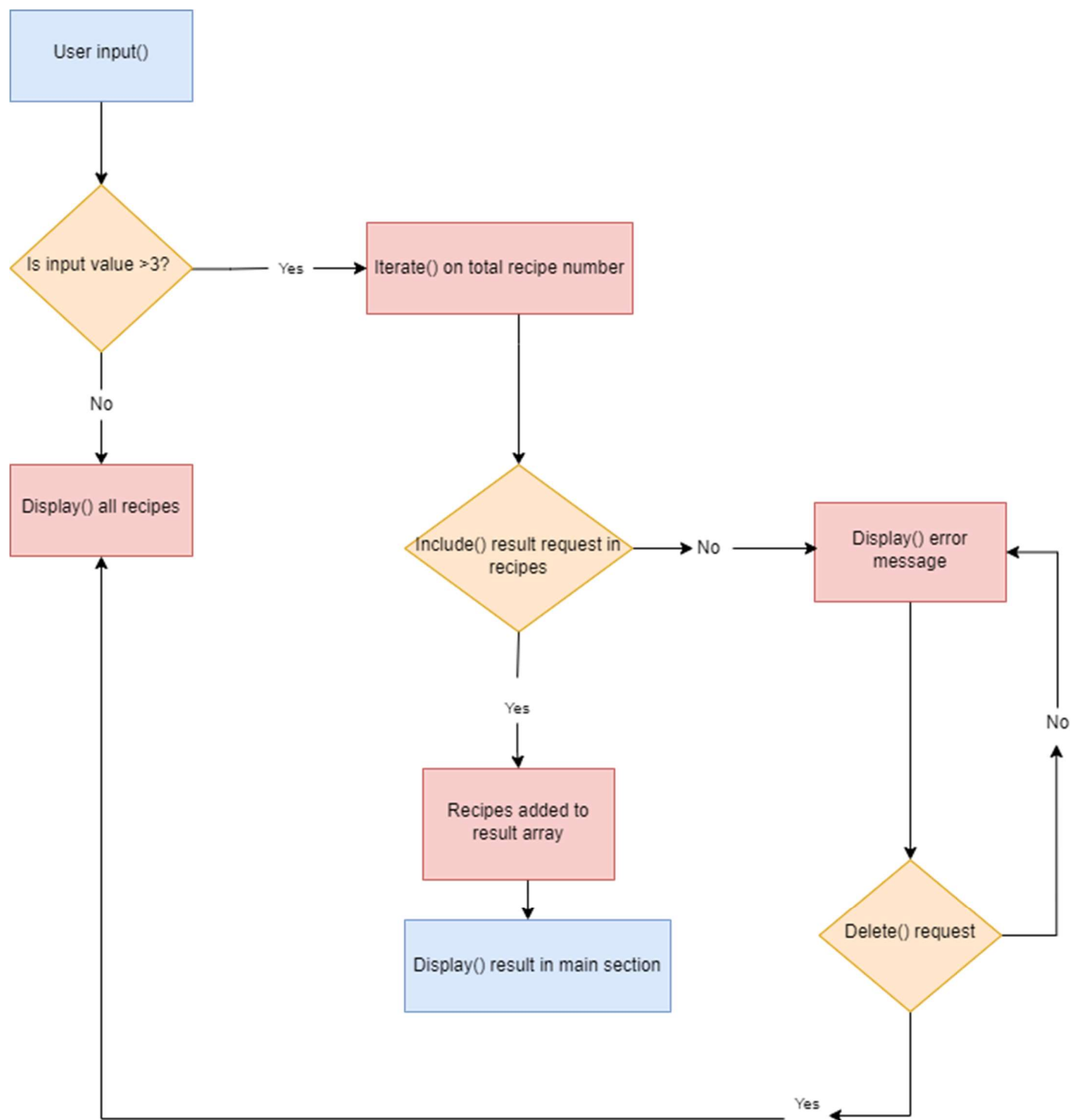
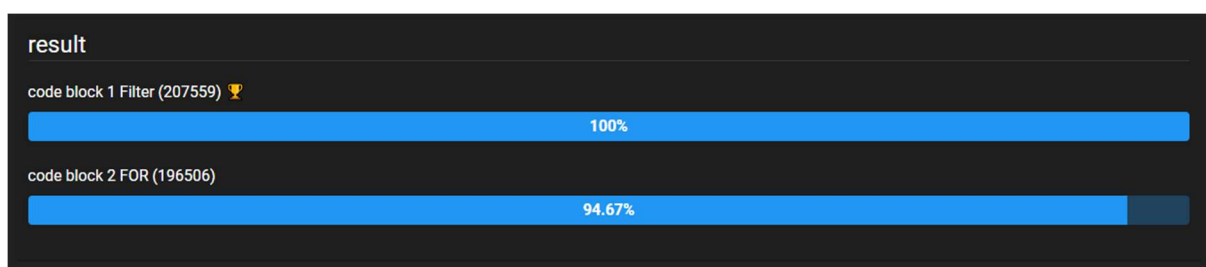


Figure 2 Algorithme : Utilisation d'une boucle for sur l'ensemble des recettes : utilisation de la méthode push si la recette remplit une des conditions de la recherche.

Analyse des Performances JavaScript

1. Test de performance avec l'outil de benchmark en ligne JSBen.ch

Nous obtenons pour l'algorithme avec Filter 207 559 opérations par seconde contre 196 506 opérations par seconde pour l'algorithme avec For sur un test data de 50 recettes. On remarque ici que l'algorithme avec Filter est plus performant que l'algorithme avec la boucle native For. En effet plus la nombre d'opérations par seconde est élevé plus l'algorithme est performant :



Lien vers la comparaison: <https://jsben.ch/ZD9id>

On test les performances JS suivant plusieurs scénarios prédéfinis. Les nombres du tableau indiquent le nombre total d'opérations/seconde dans un temps imparti.

On remarque sur l'ensemble des scénarios 1 à 5 (pas d'occurrence entré au préalable, plus de datas en entrée) que l'algorithme Filter obtient plus d'opérations par seconde que pour l'algorithme For, ce qui confirme une nouvelle fois son avantage de rapidité et de performance.

En effet le scénario 1 comme constaté précédemment obtient 207 559 opérations par seconde pour l'algorithme Filter contre 196 506 opérations par seconde pour l'algorithme For. Le scénario 2 indique pour une occurrence vide au départ 1 561 076 opérations par seconde sont effectués pour l'algorithme Filter contre 332 013 opérations par seconde pour l'algorithme For. Les scénarios 3 à 5 indiquent que pour respectivement 500, 1000 et 5000 datas en entrées on obtient 16896, 9894 et 6821 pour l'algorithme Filter contre 16007, 9670 et 6612 pour l'algorithme For.

	Algo1 Filter	Algo2 For
Scenario1 (occurrence: ex: poulet) data: 50	207559	196506
Scenario2 (occurrence: ex: "") data: 50	1561076	332013
Scenario3 (occurrence: ex: poulet) data: 500	16896	16007
Scenario4 (occurrence: ex: poulet) data: 1000	9894	9670
Scenario5 (occurrence: ex: poulet) data: 5000	6821	6612

Tableau de résultats des différents scénarios en opérations/seconde

JSBEN.CH

Comparaison benchmark Filter vs For

Setup block (useful for function initialization, it will be run before every test, and is not part of the benchmark.)


```

5      "servings" : 1,
6 *    "ingredients": [
7 *      {
8 *        "ingredient" : "Lait de coco",
9 *        "quantity" : 400,
10 *       "unit" : "ml"
11 *     },
12 *     {
13 *       "ingredient" : "Jus de citron",
14 *       "quantity" : 2
15 *     },
16 *   ]

```

[ADD LIBRARY](#)


boilerplate block (code will executed before every block and is part of the benchmark, use it for data initializing.)

code block 1 Filter 

```

1      let value = "Poulet";
2      let recipesMatched = [];
3 *     recipes.filter((recipe) => {
4 *       if (
5 *         recipe.name.includes(value) ||
6 *         recipe.description.includes(value) ||
7 *         recipe.ingredients.some((elt) => elt.ingredient.includes(value))
8 *       ) {
9 *         recipesMatched.push(recipe);
10 *       }
11 *     });
12 *     return {

```

code block 2 FOR 

```

1      let value = "Poulet";
2      let recipesMatched = [];
3 *     for (let i = 0; i < recipes.length; i++) {
4 *       const { name, ingredients, description } = recipes[i];
5 *       const includesInName = name.includes(value);
6 *       const includesInDescription = description.includes(value);
7 *       let includesInIngredients = false;
8 *       for (let y = 0; y < ingredients.length; y++) {
9 *         if (ingredients[y].ingredient.includes(value)) {
10 *           includesInIngredients = true;
11 *         }
12 *       }

```

[Codes block datas recettes, filter et for \(JSBen.ch\)](#)

2. Test de performance avec l'outil de benchmark JSBench.me

On observe que l'observation précédente avec JSBen.ch se confirme avec l'outil JSBench.me. Cet outil fonctionne de la même manière que JSBench.me car il s'agit d'un outil en ligne de type benchmark Javascript dépendant de la connexion coté client. On obtient 102 599 opérations par seconde pour la méthode filter alors qu'on a 101946 opérations par seconde pour la méthode for. Le résultat précédent se confirme même si la différence est moins flagrante que pour JSBen.c ; la méthode de programmation fonctionnelle avec filter est plus performante et plus rapide que la méthode native For. On obtient plus d'opérations par secondes pour filter comme indiqué sur l'impression d'écran ci-dessous :

filter vs for	
enter test suite description	
Setup HTML	Setup HTML code goes here... It will be injected into BODY of the test suite HTML d
Setup JavaScript	<pre>const recipes = [{ "id": 1, "name": "Limonade de Coco", "servings": 1, "ingredients": [{ "ingredient": "Lait de coco", "quantity": 400, "unit": "ml" }] }];</pre>
FILTER	<pre>let value = "Poulet"; let recipesMatched = []; recipes.filter((recipe) => { if (recipe.name.includes(value) recipe.description.includes(value) recipe.ingredients.some(elt => elt.ingredient.includes(value))) { recipesMatched.push(recipe); } }); return f</pre>
finished	
102599.83 ops/s ± 0.87%	
Fastest	
FOR	<pre>let value = "Poulet"; let recipesMatched = []; for (let i = 0; i < recipes.length; i++) { const { name, ingredients, description } = recipes[i]; const includesInName = name.includes(value); const includesInDescription = description.includes(value); let includesInIngredients = false; for (let y = 0; y < ingredients.length; y++) { if (ingredients[y].ingredient.includes(value)) { includesInIngredients = true; } } }</pre>
finished	
101946.8 ops/s ± 0.51%	
0.64 % slower	

3. Test de performance avec l'outil interne console.time() et console.timeEnd()

Enfin, j'ai mis en place l'outil interne de performance directement dans le code Javascript avec console.time() et console.timeEnd(). Console.time doit être placé dans le fichier principal main.js, et après que la recherche débute à 3 caractères (Utils.Valid) et console.TimeEnd doit être placé juste après l'obtention du résultat :

```
// Construit avec la recherche Input
document.getElementById("searchBarInput").addEventListener("keyup", (key) => {
  let valueSearch = key.target.value;
  if (Utils.Valid(valueSearch)) {
    console.time();

    let result = Search.searchMainInput(valueSearch);
    if (result.recipesMatched.length === 0) {
      return Messages.buildResultMessageWithNoResult();
    }
    Utils.clearRecipes();
    Builder.initSearch(result);
    console.timeEnd();

    return;
  }
});
```

Avec méthode filter :

```
default: 2.176025390625 ms  
>
```

Avec la méthode for :

```
default: 2.298095703125 ms    main.js:22  
ms  
>
```

Il faut noter que le test a été effectué sur la recherche principale avec le mot clé ingrédient « ail ».

Comme on l'observe sur les impressions d'écran, avec la méthode Filter le temps d'exécution de la recherche principale se fait en 2,17ms contre 2,29 pour la méthode For. Ce résultat confirme et est cohérent avec les résultats obtenus pour les outils en ligne JSBen.ch et JSbench.me. Même si la différence n'est pas grande entre les deux temps d'exécution des deux méthodes, Filter reste plus performant et plus rapide que For.

4. Recommandation d'algorithme à garder suite à mon analyse et mes tests :

Pour conclure on peut indiquer que la méthode Filter est plus performante que la méthode For même si cela reste assez marginal (100% pour la méthode Filter contre 95% pour la méthode For en termes de performance d'opérations/seconde). L'ensemble des tests effectués avec les outils Jsben.ch, Jsbench.me et console.Time() démontre une cohérence de résultats avec un avantage pour la méthode Filter.

La méthode Filter possède de nombreux avantages comme une recherche plus performante (plus d'opérations par seconde dans un temps imparti), plus rapide, plus pertinente et plus fluide. Elle est compatible avec plus de navigateurs. Il y a aussi moins de lignes de codes et donc moins de répétitions. Le code est également plus maintenable (ESLINT, structure). En effet à l'opposé, la méthode For qui certes peut apporter une lisibilité de code plus intéressante, a de nombreux inconvénients comme des boucles moins performantes, plus de lignes de code et obtient moins d'opérations par seconde dans un temps imparti (moins fluide et moins performant)

Cela est confirmé par de nombreux tests, benchmarks d'outils externes et internes et de reviews documentés en ligne que sur un contexte identique de recherche principale la programmation fonctionnelle est plus performante que la programmation de boucles natives

Comme indiqué en conclusion du document d'investigation, la solution retenue pour la démonstration de l'application concerne la méthode de programmation fonctionnelle Filter.