FICHE D'INVESTIGATION DE FONCTIONNALITE

Fonctionnalité : Filtrage des recettes	**Fonctionnalités #1**
Problématique : Créer un système efficace de filtrage des recettes à partir d'une liste de recettes initiales, en tenant compte à la fois des mots-clés (saisis dans une barre de recherche) et des tags (ingrédients, appareils ou ustensiles sélectionnés). L'objectif est de maximiser la performance et la maintenabilité tout en garantissant une bonne lisibilité du code.	

Option 1 : Approche avec **boucles natives**

Le filtrage est implémenté en utilisant des boucles for natives pour parcourir et comparer les éléments des tableaux. L'intersection des recettes filtrées par mots-clés et par tags est réalisée manuellement avec des boucles imbriquées.

Avantages :

- **Contrôle explicite :** Les boucles permettent un suivi précis des itérations, utile pour des ajustements ou des optimisations spécifiques.
- **Compatibilité universelle :** Fonctionne dans tous les environnements JavaScript sans dépendance aux fonctionnalités avancées.

Inconvénients :

- **Complexité élevée :** Le code devient rapidement long et difficile à lire, en particulier avec des boucles imbriquées.
- **Moins performant :** Les boucles imbriquées augmentent la complexité temporelle.
- **Difficulté de maintenance :** En cas d'ajout de nouvelles conditions ou de changement, il est plus complexe de modifier le code.

Exemple d'utilisation :

- Scénarios où le volume de données reste limité et où une approche simple est acceptable.

Option 2 : Approche avec programmation fonctionnelle

Le filtrage est implémenté à l'aide de méthodes fonctionnelles comme filter, map, et every. Ces méthodes permettent de travailler de manière déclarative avec des itérations implicites.

Avantages :

- **Lisibilité accrue :** Le code est plus concis et expressif.
- **Performances optimisées :** Les méthodes fonctionnelles sont souvent optimisées en interne dans les moteurs JavaScript modernes.
- **Maintenance facilitée :** Les modifications sont plus faciles à intégrer grâce à l'approche modulaire.
- **Evite les erreurs classiques :** Pas besoin de gérer manuellement les index ou les conditions d'arrêt des boucles.

Inconvénients :

- **Apprentissage nécessaire :** Les développeurs non familiers avec les méthodes fonctionnelles pourraient avoir besoin d'un temps d'adaptation.
- **Dépendances implicites :** Nécessite des fonctionnalités modernes, ce qui pourrait limiter son usage dans des environnements très anciens.

Exemple d'utilisation :

- Projets où la maintenabilité et la lisibilité du code sont prioritaires, notamment dans des équipes de développement collaboratif.

Solution retenue :

Nous avons retenu **l'approche avec programmation fonctionnelle** pour les raisons suivantes :

1. **Lisibilité et clarté :** Les méthodes comme filter et every rendent le code plus facile à lire et à comprendre.
2. **Performances générales :** Bien que l'impact puisse être marginal sur des petits ensembles de données, l'approche fonctionnelle est souvent plus rapide dans les implémentations modernes.
3. **Maintenance et extensibilité :** En cas d'évolution des besoins, l'approche modulaire facilite les ajouts ou modifications de logique.

Cette solution est plus adaptée à un environnement de développement moderne avec des exigences en termes de qualité du code et de collaboration en équipe.