Immer

Qu'est-ce que l'immuabilité ?

Tout d'abord, immuable est l'opposé de mutable et mutable signifie changeable, modifiable... susceptible d'être modifié.

Donc, quelque chose d'immuable est quelque chose qui ne peut pas être modifié.

Poussé à l'extrême, cela signifie qu'au lieu d'avoir des variables traditionnelles, vous seriez constamment en train de créer de nouvelles valeurs et de remplacer les anciennes. JavaScript n'est pas aussi extrême, mais certains langages ne permettent pas du tout la mutation (Elixir, Erlang et ML viennent à l'esprit).

Bien que JavaScript ne soit pas un langage purement fonctionnel, il peut parfois prétendre l'être. Certaines opérations sur les tableaux en JS sont immuables (ce qui signifie qu'elles renvoient un nouveau tableau, au lieu de modifier l'original). Les opérations sur les chaînes sont toujours immuables (elles créent une nouvelle chaîne avec les modifications). Vous pouvez également écrire vos propres fonctions qui sont immuables. Il vous suffit de respecter quelques règles.

Un exemple de code avec la mutabilité

Prenons un exemple pour voir comment fonctionne la mutabilité. Nous allons commencer par cet objet user :

```
var user={
    name:"user1",
    adress:{
        sreet:"ste 123",
        city:'NJ'
    },
    age:45}
```

Imaginons que nous ayons l'état de base ci-dessus, et que nous ayons besoin de mettre à jour les propriétés ***nom*** et ***street***. Cependant, nous ne voulons pas modifier l'état de base original, et nous voulons également éviter le clonage profond (pour préserver le premier user).

Copier un objet

```
    var newuser=user
    newuser.name="user2"
    console.log('old user:'+JSON.stringify(user))
    console.log('new user:'+JSON.stringify(newuser))
```

output :

old user:{"name":"user2","adress":{"sreet":"ste 123","city":"NJ"},"age":45}

new user:{"name":"user2","adress":{"sreet":"ste 123","city":"NJ"},"age":45}

C’est étrange, le code ne fait pourtant aucune modification sur user ! Pourquoi le JavaScript se comporte de cette façon ?

Parce que les chaînes de caractères sont des valeurs primitives, alors que les tableaux et les objets utilisent des références.

Méthode 1 : Utiliser le destructuring

```
var newuser={...user,adress:{...user.adress}}
  newuser.name="user2"
  newuser.adress.city="NY"
    console.log('old user:'+JSON.stringify(user))
    console.log('new user:'+JSON.stringify(newuser))
```

output

old user:{"name":"user1","adress":{"sreet":"ste 123","city":"NJ"},"age":45}

new user:{"name":"user2","adress":{"sreet":"ste 123","city":"NY"},"age":45}

Le **destructuring** est une notion essentielle en JavaScript mais dans la pratique, la syntaxe peut être un peu frustrante à utiliser, surtout lorsqu'il y a plusieurs niveaux. Heureusement pour nous, il existe une librairie JavaScript qui permet de faire changer un state sans faire de mutations : Immer !

Méthode 2 : Immer

```
const newUser=produce(user,draft=>{
    draft.name="user2"
    draft.adress.city="NY"
})
     console.log('old user:'+JSON.stringify(user))
   console.log('new user:'+JSON.stringify(newUser))
```

output :

old user:{"name":"user1","adress":{"sreet":"ste 123","city":"NJ"},"age":45}

new user:{"name":"user2","adress":{"sreet":"ste 123","city":"NY"},"age":45}

Dans ce cas, le baseState reste inchangé, tandis que le nextState est mis à jour pour refléter les modifications apportées au draftState. Vous pouvez en savoir plus sur Immer en consultant son site officiel

Immer

Immer est l'une des nombreuses bibliothèques d'immuabilité que vous pouvez utiliser dans votre application. Selon son site officiel, Immer est basé sur le mécanisme de copie sur écriture. L'idée est d'appliquer des changements à un draftState temporaire, qui sert de proxy à l'état actuel. Immer vous permettra d'interagir facilement avec vos données tout en conservant tous les avantages liés à l'immuabilité.