Redux, une intro.

- C'est un système de gestion d'état qui se comporte de manière consistante : avec les mêmes inputs, les mêmes outputs sont produites, à chaque fois
- Peut s'exécuter dans différents environnements : client, serveur et natif(mobile etc)
- Indépendant de React
- Le cœur de la librairie est assez léger, 2ko dépendances comprises
- Un peu embêtant à bien configurer

Redux, une intro.

- Créer une app tout en configurant tout ce qu'il faut comme il faut :

npx create-react-app my-app --template redux

Redux, une intro.

Cas de base :

- L'état global, dans son entièreté, est stocké en un seul objet dans une seul 'store'

- La seule façon de changer l'état est de créer une action (objet décrivant le changement à opérer) et de le *dispatcher* au store

- Pour spécifier la façon avec laquelle l'état doit changer dû à l'action, on devra écrire une fonction pure (sans effets de bord) qui calcule le nouvel état à partir de l'ancien. Ce sera un reducer.

- Tout comme pour un state géré via useReducer ou useState, on ne modifiera jamais directement l'objet représentant l'état. Aucune mutation permise.

Redux, une intro.

Voir le fichier exemple1.js

Redux, une intro.

Dans une application Redux typique, il n'y a qu'un seul store avec une seule fonction reducer.

Au fur et à mesure que l'application grandit, on éclatera le reducer en réducteurs plus petits fonctionnant indépendamment sur les différentes parties de l'arborescence d'état.

C'est exactement comme s'il n'y a qu'un seul composant racine dans une application React, mais tout en étant unique à la racine, il sera composé de nombreux petits composants.

Redux, une intro.

Cette architecture peut sembler trop compliquée pour un compteur, mais la beauté de ce modèle réside dans sa capacité à s'adapter à des applications volumineuses et complexes.

Il permet également à des outils de développement très puissants de fonctionner comme il le faut, car il est possible de retracer chaque mutation jusqu'à l'action qui l'a provoquée.

Vous pouvez enregistrer les sessions utilisateur et les reproduire simplement en rejouant chaque action.

Exemple avec redux toolkit

Redux Toolkit simplifie le processus d'écriture de la logique Redux et de configuration du store. Avec Redux Toolkit, cette même logique ressemble à :

Voir exemple2.js

Redux Toolkit nous permet d'écrire la logique métier de façon plus concise et plus facile à lire, tout en suivant le même comportement et le même data flow Redux.

Redux, une intro.

Dans une application Redux typique, il n'y a qu'un seul store avec une seule fonction reducer.

Au fur et à mesure que l'application grandit, on éclatera le reducer en réducteurs plus petits fonctionnant indépendamment sur les différentes parties de l'arborescence d'état.

C'est exactement comme s'il n'y a qu'un seul composant racine dans une application React, mais tout en étant unique à la racine, il sera composé de nombreux petits composants.

Redux, les concepts fondamentaux

Imaginons que l'état d'une application soit décrit comme un simple objet. Par exemple, l'état d'une application todo peut ressembler à ceci:

```
{
  todos: [{
    text: 'Eat food',
    completed: true
  }, {
    text: 'Exercise',
    completed: false
  }],
  visibilityFilter: 'SHOW_COMPLETED'
}
```

Redux, les concepts fondamentaux

Cet objet est comme un «modèle» sauf qu'il n'y a pas de setters.

C'est ainsi fait pour ne pas laisser différentes parties du code changer l'état arbitrairement, provoquant des bogues difficiles à reproduire.

Pour modifier quelque chose dans l'état, on doit envoyer une action. Une action est un simple objet JavaScript décrivant ce qui s'est passé. Voici quelques exemples d'actions:

Redux, les concepts fondamentaux

Une action est un simple objet JavaScript décrivant ce qui s'est passé. Voici quelques exemples d'actions:

```
{ type: 'ADD_TODO', text: 'Go to swimming pool' }
{ type: 'TOGGLE_TODO', index: 1 }
{ type: 'SET_VISIBILITY_FILTER', filter: 'SHOW_ALL' }
```

Redux, les concepts fondamentaux

Faire en sorte que chaque changement soit décrit comme une action nous permet d'avoir une compréhension claire de ce qui se passe dans l'application.

Si quelque chose a changé, nous savons pourquoi cela a changé.

Enfin, pour lier l'état et les actions ensemble, nous écrivons une fonction appelée réducteur.

Redux, les concepts fondamentaux

Encore une fois, rien de magique à ce sujet: c'est juste une fonction qui prend l'état et l'action comme arguments, et renvoie l'état suivant de l'application.

Il serait difficile d'écrire une unique fonction reducer pour une grande application, nous écrivons donc des fonctions plus petites gérant des parties de l'état

Redux, les concepts fondamentaux

```
function visibilityFilter(state = 'SHOW_ALL', action) {
  if (action.type === 'SET_VISIBILITY_FILTER') {
    return action.filter
  } else {
    return state
  }
}

function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return state.concat([{ text: action.text, completed: false }])
    case 'TOGGLE_TODO':
      return state.map((todo, index) =>
        action.index === index
          ? { text: todo.text, completed: !todo.completed }
          : todo
      )
    default:
      return state
  }
}
```

Redux, les concepts fondamentaux

Et nous écrivons un autre réducteur qui gère l'état complet de notre application en appelant ces deux réducteurs pour les clés d'état correspondantes:

```
function todoApp(state = {}, action) {
  return {
    todos: todos(state.todos, action),
    visibilityFilter: visibilityFilter(state.visibilityFilter, action)
  }
}
```

Redux, les concepts fondamentaux

C'est fondamentalement toute la base de Redux. Notez que nous n'avons utilisé aucune API Redux.
Redux vient avec quelques utilitaires pour faciliter l'implémentation de ce modèle.
Mais l'idée principale est que vous décrivez comment votre état est mis à jour au fil du temps en réponse aux objets d'action, et 90% du code que vous écrivez est simplement du JavaScript, sans utiliser Redux lui-même, ses API ou toute magie.

La trinité sacrée en Redux

1- Une seule source de vérité : l'état global de l'app est stocké dans une seule arborescence d'objet au sein d'un seul store (appelle le magasin si tu préfères).

Cela facilite la création d'applications universelles, car l'état du serveur peut être sérialisé et passé dans le client sans effort de codage supplémentaire.

La trinité sacrée en Redux

1- Une seule source de vérité : l'état global de l'app est stocké dans une seule arborescence d'objet au sein d'un seul store (appelle le magasin si tu préfères).

Une arborescence d'état unique facilite également le débogage ou l'inspection d'une application; cela permet également de persister(sauvegarder) l'état de l'application pendant le développement, pour un cycle de développement plus rapide.

La trinité sacrée en Redux

1- Une seule source de vérité : l'état global de l'app est stocké dans une seule arborescence d'objet au sein d'un seul store (appelle le magasin si tu préfères).

Certaines fonctionnalités qui ont été traditionnellement difficiles à implémenter - Annuler / Rétablir, par exemple - peuvent soudainement devenir triviales à implémenter, si tout l'état est stocké dans une seule arborescence.

La trinité sacrée en Redux

1- Une seule source de vérité : l'état global de l'app est stocké dans une seule arborescence d'objet au sein d'un seul store (appelle le magasin si tu préfères).

```
console.log(store.getState())

/* Prints
{
  visibilityFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Consider using Redux',
      completed: true,
    },
    {
      text: 'Keep all state in a single tree',
      completed: false
    }
  ]
}
*/
```

La trinité sacrée en Redux

2- L'état est en lecture seule : la seule façon de changer le state est d'émettre une action (description de ce qui doit se passer)

Cela garantit que ni les vues ni les callbacks réseau n'auront jamais la possibilité de modifier directement l'état sur place.

Au lieu de cela, ils expriment une intention de transformer l'État. Parce que tous les changements sont centralisés et se produisent un par un dans un ordre strict, il n'y a pas de conditions de course (race condition) subtiles à surveiller.

La trinité sacrée en Redux

2- L'état est en lecture seule : la seule façon de changer le state est d'émettre une action (description de ce qui doit se passer)

Comme les actions ne sont que des objets simples, elles peuvent être consignées, sérialisées, stockées et ensuite relues à des fins de débogage ou de test.

La trinité sacrée en Redux

2- L'état est en lecture seule : la seule façon de changer le state est d'émettre une action (description de ce qui doit se passer)

```
store.dispatch({
  type: 'COMPLETE_TODO',
  index: 1
})

store.dispatch({
  type: 'SET_VISIBILITY_FILTER',
  filter: 'SHOW_COMPLETED'
})
```

La trinité sacrée en Redux

3- Les changements ne se font qu'à l'aide de fonctions pures : pour spécifier comment le state doit être modifier par une action, on écrira des reducers « purs »

Pureté = 0 effet de bord et 0 dépendance à quelque chose qui peut changer dans l'environnement, seules les entrées déterminent la sortie

La trinité sacrée en Redux

3- Les changements ne se font qu'à l'aide de fonctions pures : pour spécifier comment le state doit être modifier par une action, on écrira des reducers « purs »

Les réducteurs ne sont que de pures fonctions qui prennent l'état précédent et une action, et renvoient l'état suivant.

Ne pas oublier de return les nouveaux objets d'état, au lieu de muter l'état précédent.

On commence avec un seul réducteur et, à mesure que l'app grandit, on divise en réducteurs plus petits qui gèrent des parties spécifiques de l'arborescence d'état.

La trinité sacrée en Redux

3- Les changements ne se font qu'à l'aide de fonctions pures : pour spécifier comment le state doit être modifier par une action, on écrira des reducers « purs »

Étant donné que les réducteurs ne sont que des fonctions, on peut contrôler l'ordre dans lequel ils sont appelés, transmettre des données supplémentaires ou même créer des réducteurs réutilisables pour des tâches courantes telles que la pagination.

La trinité sacrée en Redux

3- Les changements ne se font qu'à l'aide de fonctions pures : pour spécifier comment le state doit être modifier par une action, on écrira des reducers « purs »

```
function visibilityFilter(state = 'SHOW_ALL', action) {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER':
      return action.filter
    default:
      return state
  }
}
```

La trinité sacrée en Redux

3- Les changements ne se font qu'à l'aide de fonctions pures : pour spécifier comment le state doit être modifier par une action, on écrira des reducers « purs »

Voir exemple3.js

Et c'est tout! Tu connaît maintenant maintenant ce qu'il y a de plus fondamental à connaître à propos de Redux.

Redux, peut être que t'en as pas besoin

Dan Abramov, créateur de Redux, est pour moi le Christiano Ronaldo du dev web moderne, on l'aime ou on le déteste, mais on est obligés de le respecter si on comprend un tant soit peu son art.

Allons lire ce que lui même dit de ce point précis : Redux, p't'être que t'en as pas du tout besoin.

https://medium.com/@dan_abramov/you-might-not-need-redux-be46360cf367

Redux, peut être que t'en as pas besoin

Dan Abramov, créateur de Redux, est pour moi le Christiano Ronaldo du dev web moderne, on l'aime ou on le déteste, mais on est obligés de le respecter si on comprend un tant soit peu son art.

Allons lire ce que lui même dit de ce point précis : Redux, p't'être que t'en as pas du tout besoin by the way.

https://medium.com/@dan_abramov/you-might-not-need-redux-be46360cf367

Redux, peut être que t'en as pas besoin

Et si on confortait d'abord notre « react thinking » ?
https://reactjs.org/docs/thinking-in-react.html

Avant d'aller plus loin dans Redux