

# Utilisation de Redux

# Table des matières

<b>I. Contexte</b>	<b>3</b>
<b>II. Gérer le state global</b>	<b>3</b>
<b>III. Exercice : Appliquez la notion</b>	<b>6</b>
<b>IV. Redux</b>	<b>7</b>
<b>V. Exercice : Appliquez la notion</b>	<b>13</b>
<b>VI. Essentiel</b>	<b>14</b>
<b>VII. Auto-évaluation</b>	<b>14</b>
A. Exercice final .....	14
B. Exercice : Défi.....	15
<b>Solutions des exercices</b>	<b>16</b>

## I. Contexte

**Durée** : 1 h

**Environnement de travail** : Visual Studio Code

**Pré-requis** : React basique, composants

### Contexte

Nous avons vu que, pour créer des composants d'interface un peu plus complexes dans leur logique d'affichage, il était souvent nécessaire de passer par la création d'un state lié au composant. Ce state nous permet de stocker des données, dont la nature est temporaire et non-persistée. Ces données servent à gérer les aspects de logique le temps du cycle de vie du composant et sont ensuite détruites. Quand nous avons parlé des composants, nous avons évoqué le fait que ces composants s'organisent comme des briques de construction, qu'il conviendra de mettre ensemble pour former une interface complète et cohérente. La communication entre les composants étant assurée par les props, nous sommes en mesure de gérer un state qui dépasserait les limites du composant pour maintenir un état global de l'application. Mais la gestion d'un state global, armé seulement de props et de state local, est périlleuse et d'autres outils ont été pensés pour mieux répondre à cette problématique.

## II. Gérer le state global

### Objectifs

- Différencier le state global du state local d'un composant
- Voir l'architecture Flux et ses composantes

### Mise en situation

Le state « global » fait donc allusion aux données maintenues, non pas par un composant en particulier (comme pourrait l'être « l'onglet actif » dans un composant d'onglets), mais plutôt qui concernent l'application au sens global. Ces données peuvent être par exemple le thème sélectionné par l'utilisateur pour un blog (*Light* ou *Dark*), les erreurs affichées à l'utilisateur suite à une action, le fait que le menu de l'application soit ouvert ou pas, etc.

Toutes ces données vont nous servir à maintenir l'état global de l'application durant son fonctionnement. Une fois de plus, ces données sont volatiles et non persistées. Elles n'ont d'importance que pendant le cycle de vie de l'application.

Facebook a mis au point une architecture spécifique pour gérer le state global d'une application, le pattern Flux. Il existe de<sup>1</sup> - très<sup>2</sup> - nombreuses<sup>3</sup> - implémentations<sup>4</sup> du pattern Flux, quand on parle d'application front-end SPA.

Flux repose sur plusieurs entités, que nous allons détailler.

---

1 <https://redux.js.org/>

2 <https://vuex.vuejs.org/>

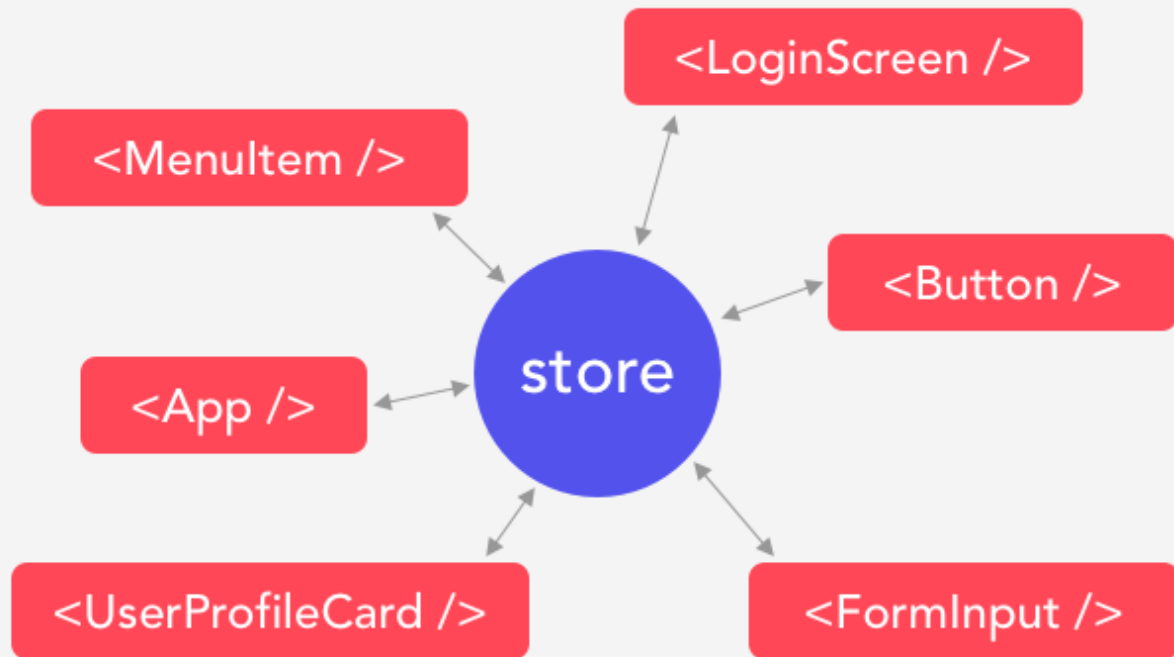
3 <https://mobx.js.org/README.html>

4 <https://medium.com/social-tables-tech/we-compared-13-top-flux-implementations-you-won-t-believe-who-came-out-on-top-1063db32fe73>

## Définition Store

Là où le state représente les données, le concept de *store* ou *datastore* représente la structure qui va nous permettre d'interagir avec nos données. Une notion très importante à prendre en compte avec le state global de l'application est celle du point de vérité unique.

Le store expose les données aux composants de l'application. Nous pourrions le représenter de cette manière :



Au centre de l'application, il offre un accès en lecture au state par n'importe quel composant qui en ferait la demande. Cependant, il ne permet la mutation des données du state que via l'appel à une action du dispatcher (nous allons en donner la définition dans un instant). On dit que le store agit comme un point de vérité unique, car il est souverain des données de l'application. Ces données peuvent être lues et traitées par plusieurs composants, mais la mutation de ces données passera obligatoirement par le store qui reste maître. Le fonctionnement des props en React est similaire : un composant parent qui passe une prop à un enfant reste maître de la prop, et agit comme le seul point de vérité pour cette prop. L'enfant pourra éventuellement muter la prop de son côté (c'est une mauvaise pratique), mais l'origine ne sera pas modifiée et tout événement qui provoquerait le *re-render* du composant écraserait la valeur locale de la prop dans le composant enfant par la valeur détenue par le point de vérité.

Une fois l'action de mutation effectuée sur le state, le store va agir comme un *event emitter* en prévenant tous les composants qui l'écoutent de la mise à jour d'une donnée.

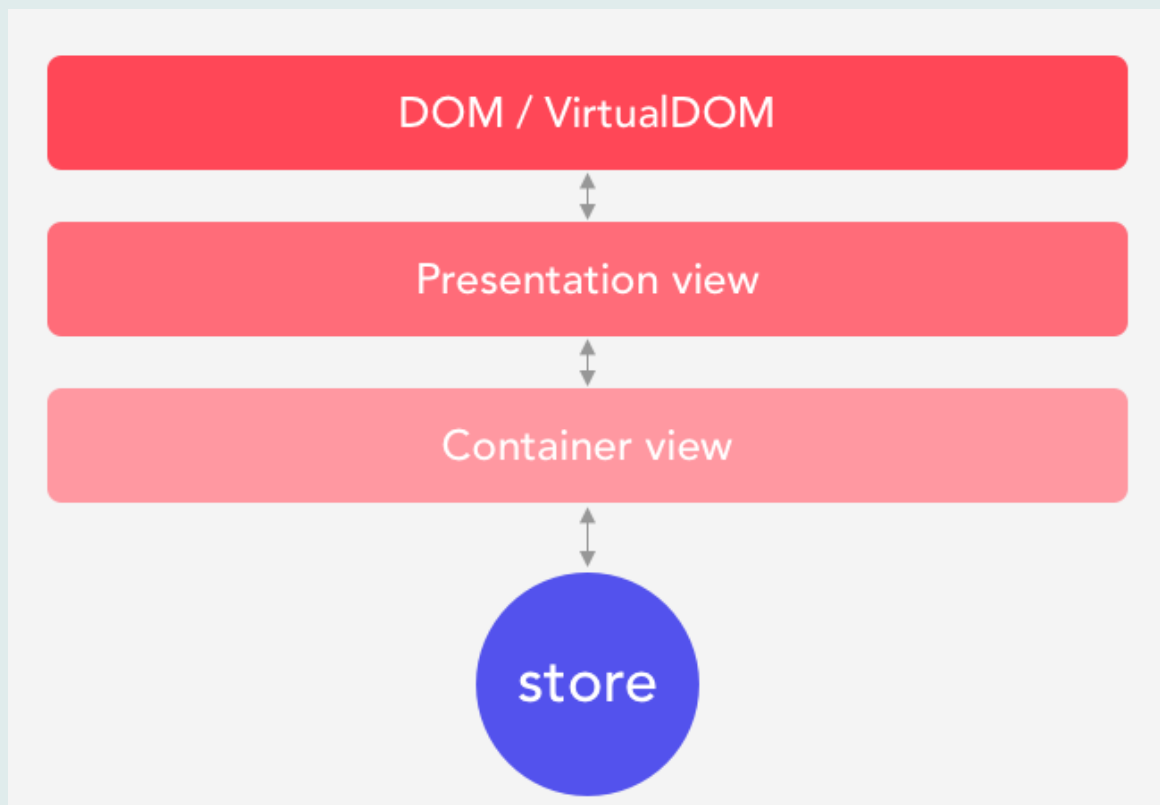
Il est possible d'avoir plusieurs stores dans l'application. Une telle pratique permet alors d'obtenir des states qui ne vont gérer les données que d'une seule entité (par exemple, uniquement les données utilisateurs). Les composants peuvent alors souscrire à plusieurs stores en fonction des données dont ils auront besoin pour fonctionner.

**Définition**    **Dispatcher**

Le **dispatcher** est une entité qui va gérer les appels des composants à des actions de mutation du state global. L'appel à l'action est ensuite transmis au(x) store(s) enregistré(s) dans l'application, afin qu'elle puisse être traitée.

**Définition**    **Views**

Les **views** représentent les composants de notre application React. Ils écoutent les mutations provenant du store et mettent à jour leur rendu en fonction de ces mises à jour. Ils sont connectés au dispatcher, qui va se charger de transmettre au store les appels aux actions. Jusqu'à présent, nous n'avons géré qu'un seul type de composant, ce que Flux définit comme des *presentation views*, responsables de l'affichage seulement. Flux définit un second type de composants, les *container views*, chargés de s'interfacer avec le store.



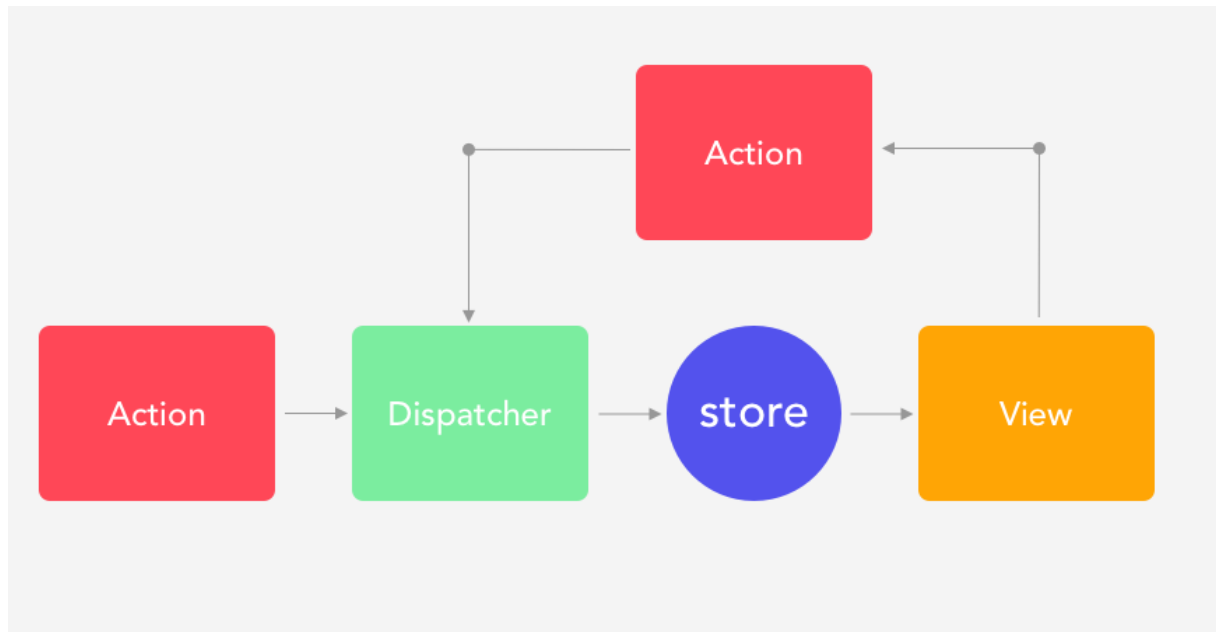
Nous allons voir un peu plus tard que React gère la communication entre la *container view* et la *presentation view* simplement via des props, les deux étant des composants React.

**Définition**    **Actions**

Une action est un objet contenant toute l'information nécessaire pour réaliser une mutation du state, en particulier le state impacté, et éventuellement un *payload* de données qui vont venir remplacer les données existantes.

**Le pattern Flux**

En prenant en compte toutes ces entités, voici le schéma de fonctionnement du pattern Flux. Il est important de noter que les données circulent toujours de manière uni-directionnelle, du dispatcher vers la view (notre composant). Ce qui n'empêche pas notre view d'émettre des actions vers le dispatcher.



#### Syntaxe À retenir

- Le state global permet de gérer les données qui concernent toute une application SPA. Il serait possible de gérer ce state global avec les outils que nous avons à notre disposition actuellement, props et state local. Mais cela pourrait s'avérer fastidieux et d'autres outils sont plus adaptés à la situation, tels que diverses implémentations du pattern Flux mis au point par Facebook. Ce pattern définit une manière de gérer le state local à travers plusieurs entités reliées entre elles par un flux de données uni-directionnel.

## Exercice : Appliquez la notion

[solution n°1 p.17]

### Exercice

Dans le pattern Flux, les données transitent entre les entités...

- ☐ De manière uni-directionnelle
- ☐ De manière bi-directionnelle

### Exercice

Flux désigne deux types de views, lesquelles ?

- ☐ Templating view et Wrapper view
- ☐ Presentation view et Container view
- ☐ Style view et Data view

### Exercice

Le rôle du dispatcher est...

- ☐ De créer plusieurs stores
- ☐ De gérer l'aspect routing de l'application
- ☐ De transmettre les actions émises par les views au store

## IV. Redux

### Objectifs

- Voir ce qu'est Redux et comment l'installer
- Tracer les parallèles et différences entre Redux et le pattern Flux dont il est issu
- Comprendre le rôle central que joue le `reducer` dans Redux
- Voir comment définir plusieurs `reducers` dans un seul store

### Mise en situation

Redux est un outil de gestion de state global basé sur le pattern Flux. C'est une librairie très utilisée dans l'écosystème React et qui possède plusieurs extensions qui lui sont propres.

Nous allons voir comment Redux implémente chaque élément du pattern Flux que nous avons décrit au travers d'un exemple portant sur la gestion des erreurs dans une application React.

#### Exemple Installation et initialisation du store

À partir d'une simple application React générée avec `create-react-app`, nous allons commencer par installer Redux dans le projet. Pour cela, il suffit de saisir la commande :

```
1 npm install redux
```

Une fois Redux installé, nous allons pouvoir créer un nouveau fichier, `store.js`, qui sera placé à la racine du répertoire `src`.

Dans ce fichier, nous allons commencer par initialiser notre store :

```
1 import { createStore } from 'redux';
2
3 const store = createStore();
4 export default store;
```

En l'état, il ne se passe pas grand chose : nous allons stocker dans ce store les données qui vont représenter des messages d'erreurs à afficher à l'utilisateur. Nous avons décidé qu'il pouvait y avoir plusieurs erreurs simultanément dans l'application, c'est pourquoi notre state `errors` sera initialisé sous la forme d'un tableau vide. Pour décrire ce state (le tableau vide `errors`) à notre store, nous allons devoir passer par une entité intermédiaire qui s'appelle un `reducer`. Redux pousse à n'avoir qu'un seul store par application, mais nous avons vu qu'il était possible de diviser notre gestion du state global en plusieurs stores, qui porte chacun un type de données particulier. Avec Redux, ces stores sont remplacés par des `reducers`. Il est alors possible de déclarer autant de `reducers` qu'on le souhaite, qui seront responsables d'un seul type de données et seront rattachés à notre store unique.

Concrètement, un `reducer` Redux est simplement une fonction pure (qui n'a pas d'effets de bord) avec la signature suivante :

```
1 (state, action) => state
```

C'est-à-dire qu'un `reducer` prend deux paramètres : le state de l'application et une action de mutation. Le `reducer` décrit les transformations à effectuer en fonction du type d'action passée en paramètre et renvoie le state modifié par l'une des mutations.

Nous allons créer notre `reducer errorsReducer` :

```
1 const errorsReducer = (state = { errors: [] }, action) => {
2   switch(action.type) {
3     case 'ADD_ERROR':
4       return {errors: [...state.errors, action.payload]};
5     case 'RESET_ERRORS':
6       return {errors: []};
7     default:
8       return state;
9   };
10};
```

On considère que nos actions possèdent une propriété `type` qui va nous permettre de les différencier dans le `switch...case`.

Nous prévoyons deux actions dans ce `reducer` : `ADD_ERROR` qui nous permet d'ajouter une nouvelle erreur dans le tableau d'erreurs, et `RESET_ERRORS` qui fera en sorte de réinitialiser nos erreurs.

Si on observe le retour de nos `case`, on constate quelque chose de particulier : plutôt que de réaliser une simple mutation, on renvoie systématiquement un nouvel objet de `state` dans son entièreté. Prenons l'exemple de l'action `ADD_ERROR`. Si on considère que `action.payload` correspond à une « erreur », pour ajouter cette nouvelle erreur dans le tableau d'erreurs, nous aurions pu écrire quelque chose comme :

```
1 state.errors.push(action.payload);
```

Mais nous avons écrit :

```
1 return {errors: [...state.errors, action.payload]};
```

## Fondamental Immutabilité du state

Avec Redux, le `state` est présumé immuable. Il est capital de prendre cela en compte, sans quoi **vous allez rencontrer des bugs de synchronisation du state**. C'est extrêmement important et, pour être honnête, c'est probablement l'élément le plus perturbant pour les développeurs qui apprennent Redux. Il faut absolument renvoyer l'objet de `state` en entier systématiquement à chaque mutation. On ne doit pas simplement modifier une valeur à l'intérieur. Pour cette raison, le `spread-operator` est le meilleur ami du développeur Redux : dans la ligne de code ci-avant, on fait en sorte de récupérer tous les éléments du tableau `errors` pour les réinjecter dans le tableau avec la nouvelle valeur. Redux met à disposition une page dédiée<sup>1</sup> sur sa documentation, spécifiquement pour fournir des patterns de mutation immutables, tels que celui que nous venons de voir un peu plus haut.

## Liaison avec les composants

Maintenant que nous avons vu comment initialiser un store avec un `reducer`, qui porte à la fois le `state` et les actions réalisables sur ce `state`, voyons maintenant comment Redux gère la liaison avec nos composants. Si on se réfère au pattern Flux, il s'agit des couches *dispatcher* et *view*.

Dans sa forme la plus simple, il va nous être possible de souscrire au store et à ses éventuelles mutations en utilisant la fonction :

```
1 store.subscribe(() => console.log(store.getState()));
```

À l'initialisation du store et à chaque mutation qui suivra, la fonction anonyme contenue à l'intérieur de `store.subscribe()` s'exécutera. En l'occurrence, nous sommes dans un cas très simple et nous souhaitons juste afficher le contenu du `state`. Dans la réalité, nous verrons un peu plus tard que la fonction `subscribe()` s'utilise très rarement telle quelle : nous ferons plutôt appel à une librairie supplémentaire pour lier le cycle de vie de nos composants avec le processus de souscription au store Redux.

<sup>1</sup> <https://redux.js.org/recipes/structuring-reducers/immutable-update-patterns>



Enfin, pour déclencher des actions sur le store, là où le pattern Flux définit une entité précise, le *dispatcher*, Redux expose une fonction `store.dispatch()` qui va permettre d'envoyer une action sous la forme d'un objet qui sera transmise au store via le *reducer* associé.

Si nous souhaitons faire appel à l'action `ADD_ERROR`, nous allons pouvoir écrire :

```
1 store.dispatch({
2   type: 'ADD_ERROR',
3   payload: {
4     id: Math.random(),
5     message: 'Oops... Something bad happened 🤖',
6     severity: 'critical'
7   }
8 });
```

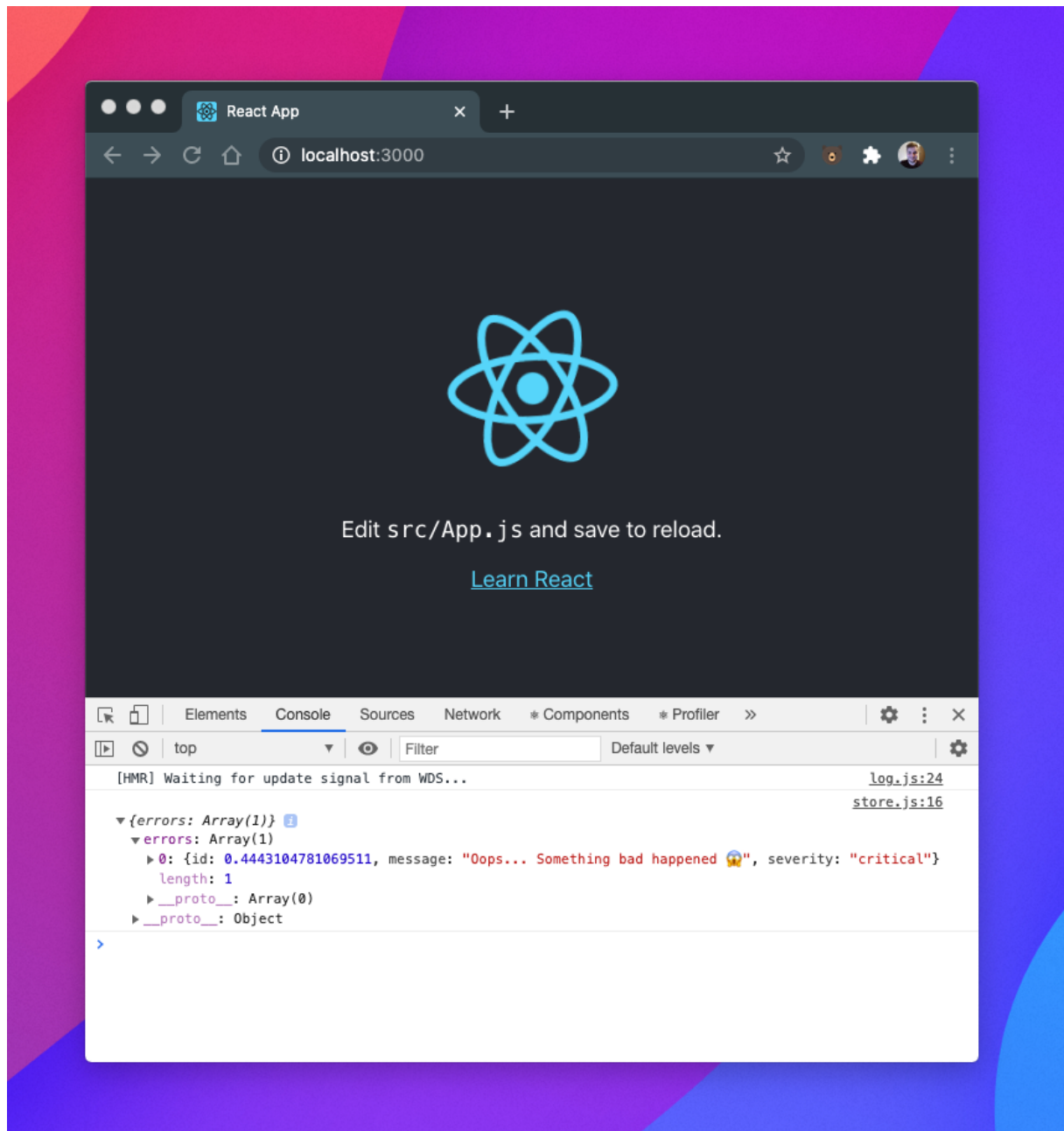
Le format de l'objet passé à la fonction `dispatch` est laissé à la discrétion du développeur : cependant, la propriété `type` est presque toujours nommée de cette manière. Nous créons ici une nouvelle erreur que nous qualifions de « critique » avec un message et un identifiant unique. Si on consolide toutes les notions vues jusqu'à présent, nous obtenons le code suivant :

```
1 import { createStore } from 'redux';
2
3 const errors = (state = { errors: [] }, action) => {
4   switch(action.type) {
5     case 'ADD':
6       return {errors: [...state.errors, action.payload]};
7     case 'RESET':
8       return {errors: []};
9     default:
10      return state;
11   };
12 };
13
14 const store = createStore(errors);
15
16 store.subscribe(() => console.log(store.getState()));
17
18 store.dispatch({
19   type: 'ADD',
20   payload: {
21     id: Math.random(),
22     message: 'Oops... Something bad happened 🤖',
23     severity: 'critical'
24   }
25 });
26
27 export default store;
```

Importons maintenant notre fichier `store.js` dans `index.js` pour qu'il soit exécuté :

```
1 import './store';
```

Dans Chrome Dev Tools, le résultat affiché est alors :



Au chargement de l'application, celle-ci a souscrit aux mutations du store : quand l'action `ADD_ERROR` a été appelée, la fonction anonyme dans `store.subscribe` a été exécutée, ce qui a provoqué l'affichage console du state dans lequel nous pouvons retrouver l'erreur qui a été ajoutée au tableau d'erreurs.

### Plusieurs reducers

Nous avons déjà pu le voir, Redux pousse les développeurs à ne définir qu'un seul store dans leur application, mais qui permet de gérer plusieurs domaines de données via des `reducers`. Pour le moment, nous avons simplement déclaré un `reducer`, nommé `errors`, dans notre store. Voyons maintenant comment ajouter un second `reducer`, pour gérer cette fois-ci les données qui concernent l'utilisateur connecté et l'authentification de celui-ci.

### Remarque

Nous parlons jusqu'ici de state « global » de l'application. Nous aurons l'occasion de voir dans une prochaine section que de plus en plus de développeurs font la distinction entre le `client state` et le `server state`, qui forment ensemble ce qu'on a nommé jusqu'ici le `global state`. Les erreurs de l'application peuvent être catégorisées comme du `client state` (local, non persisté). À l'inverse, les données relatives à l'utilisateur (acquises et persistées de manière asynchrone sur une source de données sur laquelle nous n'avons pas forcément la pleine souveraineté) font partie de ce qu'on appelle le `server state`. Ces dernières années, les développeurs ont eu tendance à utiliser Redux pour gérer ces deux types de state, et c'est également ce que nous allons faire ici. Gardez cependant en mémoire que gérer du `server state` avec Redux pourrait ne pas être complètement optimal. Nous verrons un peu plus tard une solution plus appropriée pour gérer ce type de données.

### Exemple

Pour gérer ce nouveau `reducer`, nous allons commencer par importer dans le fichier `store.js` le module `combineReducers` venant de Redux :

```
1 import { combineReducers, createStore } from 'redux';
```

À ce state, si notre application requiert plusieurs `reducers`, c'est vraisemblablement qu'il devient intéressant de déplacer chaque `reducer` dans son propre fichier. Pour plus de clarté dans notre exemple, nous allons déclarer les `reducers` directement dans le fichier `store.js`.

```
1 import { combineReducers, createStore } from 'redux';
2
3 const userInitialState = {
4   id: null,
5   username: null,
6   authToken: null
7 };
8
9 const userReducer = (state = userInitialState, action) => {
10   switch(action.type) {
11     case 'LOGIN_USER':
12       return {...action.payload};
13     case 'LOGOUT_USER':
14       return {...userInitialState};
15     default:
16       return state;
17   };
18 };
19
20 const errorsReducer = (state = { errors: [] }, action) => {
21   switch(action.type) {
22     case 'ADD_ERROR':
23       return [...state.errors, action.payload];
24     case 'RESET_ERRORS':
25       return {errors: []};
26     default:
27       return state;
28   };
29 };
```

Nous avons défini ici deux `reducers` : `userReducer` et `errorsReducer`. Chaque `reducer` possède son propre `state`, celui de `userReducer` est d'ailleurs externalisé dans une constante `userInitialState` plutôt que d'être défini comme valeur par défaut du premier paramètre du `reducer`.

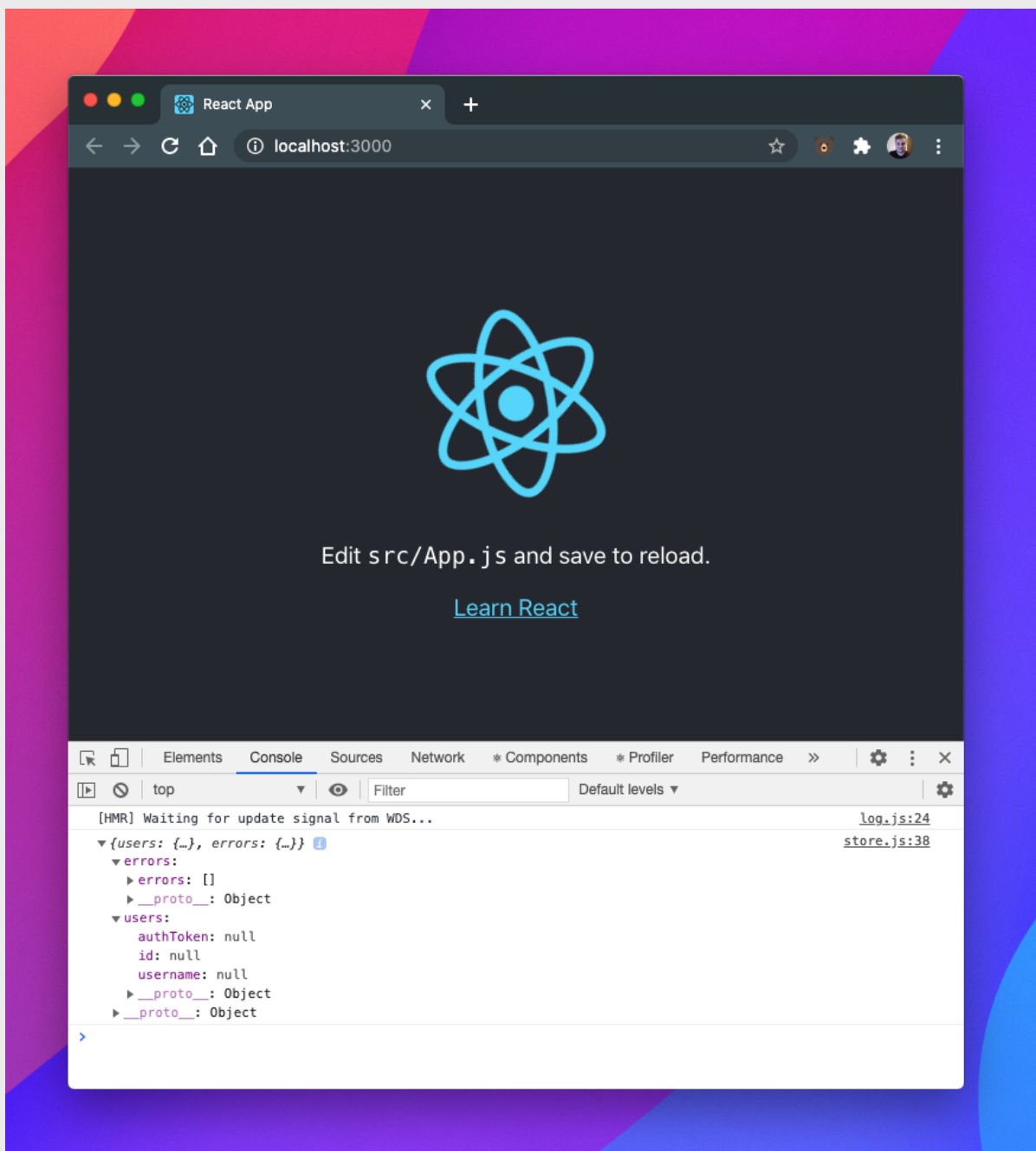
Le reducer `userReducer` expose deux nouvelles actions : `LOGIN_USER` et `LOGOUT_USER`, qui servent respectivement à remplir l'objet de state suite à la connexion de l'utilisateur et à le réinitialiser en cas de déconnexion.

Nous allons maintenant faire usage de `combineReducers` en créant un nouveau reducer racine, `rootReducer`, qui sera passé au store.

```
1 const rootReducer = combineReducers({
2   users: userReducer,
3   errors: errorsReducer
4 });
5
6 const store = createStore(rootReducer);
```

Notre store est maintenant prêt, si on affiche le contenu du state :

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



On observe que le state est maintenant un objet dont les propriétés de premier niveau sont celles définies dans l'objet passé en paramètre de la fonction `combineReducers`. Pour cette raison, même s'il serait possible grâce à la syntaxe ES6 object literal<sup>1</sup> d'écrire notre objet de cette manière, ce n'est pas recommandé.

```
1 const rootReducer = combineReducers({  
2   userReducer,  
3   errorsReducer  
4 });
```

Nous nous retrouverions alors avec un objet de state contenant les propriétés `userReducer` et `errorsReducer` : ce n'est pas un très bon choix de nommage et on préférera renommer les propriétés à la déclaration de l'objet sous leur forme plus concise, `user` et `errors`.

### Syntaxe À retenir

- Redux est une implémentation React du pattern Flux qui est extrêmement populaire. Redux est très inspiré du pattern Flux, mais ne reprend cependant pas toutes ses entités. Nous retrouvons le store et les actions, mais ces actions sont définies dans un élément spécifique à Redux, les `reducers`. Ce sont également ces `reducers` qui exposent le state.
- Un `reducer` est une fonction pure, qui prend en paramètres le state original, ainsi que l'action, et retourne un nouvel objet de state contenant la mutation effectuée. Il est fortement conseillé (c'est-à-dire nécessaire) avec Redux de faire en sorte que le state soit immuable : chaque action renverra un nouvel objet de state contenant les mises à jour.
- Le dispatcher Flux est intégré à la librairie sous la forme de la fonction `store.dispatch()` qui permet d'émettre une action vers le ou les `reducer(s)`. De la sorte, il est possible de définir plusieurs `reducers` qui seront combinés dans un `reducer` racine, passé à son tour en paramètre au store. Le state qui découlera de plusieurs `reducers` combinés est un objet dont les propriétés de premier niveau seront les `reducers`, chacune de ces propriétés portera ensuite le state du `reducer` associé.

## V. Exercice : Appliquez la notion

### Question

[solution n°2 p.17]

Vous souhaitez prototyper la gestion du state global de Stopify, votre plateforme de streaming musical.

Vous représenterez les données de l'application en deux `reducers` et vous écrirez les actions possibles sur chaque `reducer` :

- **songs** - Lire une chanson (augmente un compteur du nombre de lectures - `played`)
- **playlists** - Créer une playlist nommée, ajouter une chanson à une playlist, supprimer une playlist

Enfin, vous déclencherez les actions dans l'ordre suivant et ferez en sorte que le résultat de chaque action soit affiché en console :

- Lire une chanson (augmente le nombre de lectures - `played`)
- Créer une playlist nommée
- Ajouter une chanson à une playlist
- Supprimer une playlist

Vous pourrez vous appuyer sur les données présentes dans le fichier JSON suivant pour ajouter les trois chansons au state initial du `reducer` `songs`. Il suffira d'importer le fichier pour avoir accès aux données : `import data from './stopify.json';`

<sup>1</sup> [https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Op%C3%A9rateurs/Initialisateur\\_objet](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Op%C3%A9rateurs/Initialisateur_objet)

[cf. stopify.json]

## VI. Essentiel

## VII. Auto-évaluation

### A. Exercice final

#### Exercice 1

[solution n°3 p.20]

Exercice

Dans le pattern Flux, comment s'appelle l'entité qui déclenche les mutations de state ?

- ☐ Une action
- ☐ Une mutation
- ☐ Un setter

Exercice

Le state global...

- ☐ Représente les données qui concernent toute l'application SPA
- ☐ Représente les données stockées dans le `local storage` du navigateur
- ☐ Représente les données servant à la logique d'affichage d'un composant

Exercice

Le state global ne peut être géré que par un outil implémentant le pattern Flux.

- ☐ Vrai
- ☐ Faux

Exercice

Redux est un outil...

- ☐ Qui est basé sur les contextes React
- ☐ Qui s'apparente à Apollo GraphQL
- ☐ Qui reprend l'implémentation Flux

Exercice

La méthode `createStore()` prend en paramètre...

- ☐ Un objet de state
- ☐ Un reducer
- ☐ L'instance du dispatcher

Exercice

Un reducer...

- ☐ Est un objet contenant les propriétés `state` et `action`
- ☐ Est un tableau listant les clés de `state`
- ☐ Est une fonction pure

Exercice

Une action est le plus souvent...

- ☐ Appelée dans un `switch...case` sur la valeur de `action.type`
- ☐ Contenue dans la clé `mutate` de l'objet de `state`
- ☐ Dotée d'un `payload`

Exercice

La meilleure manière d'ajouter un nouvel élément dans un tableau `users` depuis une action est...

- ☐ `return state.users.push(action.payload);`
- ☐ `return state.users[users.length + 1] = action.payload;`
- ☐ `return {...state, users: [...state.users, action.payload]};`

Exercice

La fonction permettant de souscrire aux mises à jour du `state` est...

- ☐ `store.subscribe()`
- ☐ `store.on()`
- ☐ `store.listen()`

Exercice

L'entité dispatcher Flux est remplacée en Redux par...

- ☐ `store.commit()`
- ☐ `store.dispatch()`
- ☐ `store.mutate()`

## B. Exercice : Défi

Vous allez devoir construire le store Redux d'une application de gestion de *TODO*.

### Question

[solution n°4 p.22]

Les données qu'il faudra gérer dans le `state` global sont :

- La liste des *TODO*

Un *TODO* correspond au format de données suivant :

```
1 {  
2   id: string(unique),  
3   content: string,  
4   done: boolean  
5 }
```

Les données seront réparties dans deux `reducers` : le premier servira à gérer le thème, le second les *TODO*.

Vous devrez écrire un composant simple pour afficher la liste des *TODO* et permettre d'ajouter un nouveau *TODO*. Enfin, un bouton permettra de réinitialiser la liste des *TODO*.

Nous considérons pour cet exercice qu'il n'y aura pas de persistance des *TODO*, la liste de *TODO* n'est constituée que des éléments saisis durant le cycle de vie de l'application.

**Indice :**


Nous n'utiliserons pas React Redux ici, qui facilite la liaison avec les composants React. Par conséquent, il faudra trouver un moyen de souscrire au store à la création du composant.

## Solutions des exercices



**Exercice p. 6 Solution n°1****Exercice**

Dans le pattern Flux, les données transitent entre les entités...

- ☒ De manière uni-directionnelle
- ☐ De manière bi-directionnelle
-  Le schéma de fonctionnement forme une boucle, mais les données ne vont toujours que dans un seul sens : **action -> dispatcher -> store -> view.**

**Exercice**

Flux désigne deux types de views, lesquelles ?

- ☐ Templating view et Wrapper view
- ☒ Presentation view et Container view
- ☐ Style view et Data view

**Exercice**

Le rôle du dispatcher est...

- ☐ De créer plusieurs stores
- ☐ De gérer l'aspect routing de l'application
- ☒ De transmettre les actions émises par les views au store

**p. 13 Solution n°2**

Fichier `store.js` contenant la définition de notre store Redux :

```
1 import { combineReducers, createStore } from 'redux';
2 import data from './stopify';
3
4 /**
5  * Songs reducer, initial state comes from the imported JSON file
6  * @param {array} state
7  * @param {object} action
8  */
9 const songsReducer = (state = data, action) => {
10   switch(action.type) {
11     case 'PLAY_SONG':
12       /**
13        * Find the target song...
14        */
15       const targetSong = state.find((song) => song.id === action.payload.id);
16       /**
17        * ...then spread everything that's not the target song
18        * finally add back the target song with the "played" prop incremented
19        */
20       return [
21         ...state.filter((song) => song.id !== action.payload.id),
22         {...targetSong, played: targetSong.played + 1}
```

```

23   ];
24   default:
25     return state;
26   }
27 };
28
29 /**
30  * Playlists reducer, starts as an empty array
31  * @param {array} state
32  * @param {object} action
33  */
34 const playlistsReducer = (state = [], action) => {
35   switch(action.type) {
36     case 'MAKE_PLAYLIST':
37       /**
38        * Spread the existing playlists, add the new playlist at the end.
39        * New playlist name's the name prop passed in the action's payload.
40        * New playlist is initialized with an empty "songs" array.
41        */
42       return [
43         ...state,
44         {
45           id: Math.random(),
46           name: action.payload.name,
47           songs: []
48         }
49       ];
50     case 'ADD_SONG_TO_PLAYLIST':
51       /**
52        * Find the target playlist...
53        */
54       const targetPlaylist = state.find((playlist) => playlist.id ===
55       action.payload.playlistId);
56       /**
57        * ...then spread everything that's not the target playlist
58        * finally add back the target playlist.
59        * Spread any song that would already be in the target playlist,
60        * then add the new song at the end.
61        */
62       return [
63         ...state.filter((playlist) => playlist.id !== action.payload.playlistId),
64         {
65           ...targetPlaylist,
66           songs: [
67             ...targetPlaylist.songs,
68             action.payload.song
69           ]
70         }
71       ];
72     case 'DELETE_PLAYLIST':
73       /**
74        * Return everything that's not matching the playlist id
75        * passed as prop in the payload object.
76        */
77       return [...state.filter((playlist) => playlist.id !== action.payload.id)];
78     default:
79       return state;
80   }
81 }

```

```

80 };
81
82 /**
83  * Combining our reducers into a rootReducer
84  * ready to be injected in the store.
85  */
86 const rootReducer = combineReducers({
87   songs: songsReducer,
88   playlists: playlistsReducer
89 });
90
91 /**
92  * Creating store.
93  */
94 const store = createStore(rootReducer);
95
96 /**
97  * For each global state mutation
98  * print the whole state in the console.
99  */
100 store.subscribe(() => console.log(store.getState()));
101
102 /**
103  * Play the song with id: 1
104  * 🎵 Halcyon - Elder 🎵
105  */
106 store.dispatch({ type: 'PLAY_SONG', payload: { id: 1 } });
107
108 /**
109  * Make a new playlist named "Coolest songs !" - Yea that's right.
110  */
111 store.dispatch({ type: 'MAKE_PLAYLIST', payload: { name: 'Coolest songs !' } });
112
113 /**
114  * Trying to get the first playlist's id.
115  * "?" syntax is called conditional chaining
116  *
117  * https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Op%C3%A9rateurs/Optional\_chaining
118  * It's a pretty neat feature added in ES2020
119  * At any point in the object call chain, if a prop is undefined, the call
120  * will gracefully assign "undefined" to the left operand,
121  * and will not break like it does in a normal object call chain.
122  */
123
124 const playlistId = store.getState()?.playlists[0]?.id;
125
126 /**
127  * If we found a playlist id...
128  */
129 if (playlistId) {
130   /**
131    * Add a new song to the target playlist
132    */
133   store.dispatch({
134     type: 'ADD_SONG_TO_PLAYLIST',
135     payload: {
136       playlistId,
137       song: {
138         id: Math.random(),

```

```

137     title: "Born in winter",
138     artist: "Gojira",
139     played: 0
140   }
141 }
142 })
143
144 /**
145  * Then proceed to delete the target playlist
146  */
147 store.dispatch({
148   type: 'DELETE_PLAYLIST',
149   payload: { id: playlistId }
150 })
151 }
152
153 export default store;

```

[HMR] Waiting for update signal from WDS...

```

▼ {songs: Array(3), playlists: Array(0)} ⓘ
  ▼ playlists: Array(0)
    length: 0
    ▶ __proto__: Array(0)
  ▼ songs: Array(3)
    ▶ 0: {id: 2, title: "In Waves", artist: "Trivium", played: 0}
    ▶ 1: {id: 3, title: "Sign of the Cross", artist: "Iron Maiden", played: 0}
    ▶ 2: {id: 1, title: "Halcyon", artist: "Elder", played: 1}
    length: 3
    ▶ __proto__: Array(0)
    ▶ __proto__: Object

```

PLAY\_SONG

```

▼ {songs: Array(3), playlists: Array(1)} ⓘ
  ▼ playlists: Array(1)
    ▶ 0: {id: 0.6131934970364741, name: "Coolest songs !", songs: Array(0)}
    length: 1
    ▶ __proto__: Array(0)
  ▼ songs: Array(3)
    ▶ 0: {id: 2, title: "In Waves", artist: "Trivium", played: 0}
    ▶ 1: {id: 3, title: "Sign of the Cross", artist: "Iron Maiden", played: 0}
    ▶ 2: {id: 1, title: "Halcyon", artist: "Elder", played: 1}
    length: 3
    ▶ __proto__: Array(0)
    ▶ __proto__: Object

```

MAKE\_PLAYLIST

```

▼ {songs: Array(3), playlists: Array(1)} ⓘ
  ▼ playlists: Array(1)
    ▼ 0:
      id: 0.6131934970364741
      name: "Coolest songs !"
      ▼ songs: Array(1)
        ▶ 0: {id: 0.9072023292457985, title: "Born in winter", artist: "Gojira", played: 0}
        length: 1
        ▶ __proto__: Array(0)
        ▶ __proto__: Object
      length: 1
      ▶ __proto__: Array(0)
  ▼ songs: Array(3)
    ▶ 0: {id: 2, title: "In Waves", artist: "Trivium", played: 0}
    ▶ 1: {id: 3, title: "Sign of the Cross", artist: "Iron Maiden", played: 0}
    ▶ 2: {id: 1, title: "Halcyon", artist: "Elder", played: 1}
    length: 3
    ▶ __proto__: Array(0)
    ▶ __proto__: Object

```

ADD\_SONG\_TO\_PLAYLIST

```

▼ {songs: Array(3), playlists: Array(0)} ⓘ
  ▼ playlists: Array(0)
    length: 0
    ▶ __proto__: Array(0)
  ▼ songs: Array(3)
    ▶ 0: {id: 2, title: "In Waves", artist: "Trivium", played: 0}
    ▶ 1: {id: 3, title: "Sign of the Cross", artist: "Iron Maiden", played: 0}
    ▶ 2: {id: 1, title: "Halcyon", artist: "Elder", played: 1}
    length: 3
    ▶ __proto__: Array(0)
    ▶ __proto__: Object

```

DELETE\_PLAYLIST

### Exercise p. 14 Solution n°3

**Exercice**

Dans le pattern Flux, comment s'appelle l'entité qui déclenche les mutations de state ?

- ☒ Une action
- ☐ Une mutation
- ☐ Un setter

**Exercice**


Le state global...

- ☒ Représente les données qui concernent toute l'application SPA
- ☐ Représente les données stockées dans le `local storage` du navigateur
- ☐ Représente les données servant à la logique d'affichage d'un composant

**Exercice**

Le state global ne peut être géré que par un outil implémentant le pattern Flux.

- ☐ Vrai
- ☒ Faux

 Le state global peut être géré de plusieurs manières, y compris depuis un composant. Mais dans une application volumineuse, il sera plus simple de se munir d'un outil spécialisé.

**Exercice**

Redux est un outil...

- ☐ Qui est basé sur les contextes React
- ☐ Qui s'apparente à Apollo GraphQL
- ☒ Qui reprend l'implémentation Flux


**Exercice**

La méthode `createStore()` prend en paramètre...

- ☐ Un objet de state
- ☒ Un reducer
- ☐ L'instance du dispatcher

**Exercice**

Un reducer...

- ☐ Est un objet contenant les propriétés `state` et `action`
  - ☐ Est un tableau listant les clés de state
  - ☒ Est une fonction pure
-  C'est une fonction pure avec la signature `(state, action) => state`.


### Exercice

Une action est le plus souvent...

- ☒ Appelée dans un `switch...case` sur la valeur de `action.type`
- ☐ Contenue dans la clé `mutate` de l'objet de state
- ☐ Dotée d'un `payload`

### Exercice

La meilleure manière d'ajouter un nouvel élément dans un tableau `users` depuis une action est...

- ☐ `return state.users.push(action.payload);`
- ☐ `return state.users[users.length + 1] = action.payload;`
- ☒ `return {...state, users: [...state.users, action.payload]};`
-  Le state est immutable quand on utilise Redux, un reducer doit être une fonction pure qui renvoie un nouvel objet de state.

### Exercice

La fonction permettant de souscrire aux mises à jour du state est...

- ☒ `store.subscribe()`
- ☐ `store.on()`
- ☐ `store.listen()`

### Exercice

L'entité dispatcher Flux est remplacée en Redux par...

- ☐ `store.commit()`
- ☒ `store.dispatch()`
- ☐ `store.mutate()`

## p. 15 Solution n°4

Store redux - `store.js`:

```
1 import {createStore } from 'redux';
2
3 const todosReducer = (state = [], action) => {
4   switch(action.type) {
5     case 'ADD_TODO':
6       /**
7        * We spread the existing todos, and add the new at the end
8        */
9       return [...state, action.payload];
10    case 'SET_DONE':
11      /**
12       * The start by finding the todo matching with the id
13       * passed as action.payload.
14       * We then display that item with the modified "done" property
15       * followed by the rest of the todos.
```

```

16      */
17      const item = state.find((todo) => todo.id === action.payload);
18      return [
19        {...item, done: true},
20        ...state.filter((todo) => todo.id !== action.payload)
21      ];
22      case 'RESET_TODOS':
23        /**
24         * Resetting the todos array to it's original form: empty
25         */
26        return [];
27      default:
28        return state;
29    };
30  };
31
32  const store = createStore(todosReducer);
33  export default store;

```

Composant `<Todo />` - Récupère la liste des *TODO* depuis le store redux pour les afficher, permet d'ajouter un *TODO*, de marquer un *TODO* comme « terminé » ou de réinitialiser la liste des *TODO*.

```

1  import React, { useState, useEffect } from 'react';
2  import store from '../store';
3  import styled from 'styled-components';
4
5  const Todo = () => {
6    /**
7     * Defining two local states to store
8     * - The newTodo text (from input)
9     * - The list of todos coming from the store
10   */
11   const [newTodo, setNewTodo] = useState('');
12   const [todos, setTodos] = useState([]);
13
14   /**
15    * At component mounting, subscribe to the store
16    * then call the syncStore() method
17   */
18   useEffect(() => {
19     store.subscribe(() => syncStore())
20   }, []);
21
22   /**
23    * Set local todos state with global state coming from the store
24   */
25   const syncStore = () => {
26     setTodos(store.getState());
27   };
28
29   /**
30    * Set local newTodo state with current input value
31    * @param {Object} event
32   */
33   const handleChange = (event) => {
34     setNewTodo(event.target.value);
35   };
36
37   /**

```

```

38  * Handle the "create todo" button
39  * reset the input value then
40  * dispatch ADD_TODO action with the new todo as it's payload
41  * @param {Object} event
42  */
43  const handleCreate = (event) => {
44    setNewTodo('');
45    store.dispatch({
46      type: 'ADD_TODO',
47      payload: {
48        id: Math.random(),
49        content: newTodo,
50        done: false
51      }
52    })
53  };
54
55  /**
56   * Handle the "done" button click
57   * dispatch the SET_DONE action that will toggle the "done"
58   * property on the todo whose "done" button is being clicked
59   * @param {number} id
60   */
61  const handleDone = (id) => {
62    store.dispatch({
63      type: 'SET_DONE',
64      payload: id
65    });
66  };
67
68  /**
69   * Handle the "reset todos" button
70   * dispatch the RESET_TODOS action that will empty the global state
71   */
72  const handleReset = () => {
73    store.dispatch({ type: 'RESET_TODOS' });
74  }
75
76  return (
77    <Wrapper>
78      <NewTodo>
79        <Input
80          type="text"
81          value={newTodo}
82          onChange={handleChange}
83        />
84        <Button onClick={handleCreate}>Cr  er le todo</Button>
85      </NewTodo>
86
87      <ul>
88        {todos.map((todo) => (
89          <TodoItem
90            key={todo.id}
91            todo={todo}
92          >
93            {todo.content}
94            <Button onClick={() => handleDone(todo.id)}>Fait</Button>
95          </TodoItem>

```



```

96     })}
97   </ul>
98
99   <ResetButton onClick={handleReset}>Réinitialiser les TODO</ResetButton>
100 </Wrapper>
101 );
102 };
103
104 const Wrapper = styled.div`
105   width: 48rem;
106   background: #fff;
107   padding: 1rem;
108   border-radius: 6px;
109 `;
110
111 const NewTodo = styled.div`
112   display: flex;
113   align-items: center;
114   justify-content: space-between;
115
116   & > * {
117     margin: 0 0.5rem;
118   }
119 `;
120
121 const Input = styled.input`
122   height: 40px;
123   padding: 0 0.5rem;
124   border: 1px solid #ccc;
125   border-radius: 4px;
126   flex: 1;
127 `;
128
129 const Button = styled.button`
130   height: 40px;
131   border: 0;
132   background: #5352ed;
133   color: #fff;
134   font-weight: bold;
135   font-size: 0.8rem;
136   border-radius: 6px;
137   padding: 0 2rem;
138   margin-left: 1rem;
139 `;
140
141 const ResetButton = styled(Button)`
142   background: #ff4757;
143   margin: 2rem 0;
144 `;
145
146 const TodoItem = styled.li`
147   display: flex;
148   align-items: center;
149   justify-content: space-between;
150   width: 100%;
151   margin: 1rem 0;
152   padding: 1rem;
153   background: #f5f5f5;

```

```

154 color: ${props => props.todo.done ? '#ccc' : '#404040'};
155 text-decoration: ${props => props.todo.done ? 'line-through' : 'initial'};
156 `;
157
158 export default Todo;
159

```

