16. Concevoir une application avec Redux

16.1. Qu'est-ce que Redux ?

En 2013, Facebook a dit que AngularJS de Google est lent et lourd, donc en cette année-là, il a introduit ReactJS à la communauté des développeurs. Pour le ReactJS était seulement une bibliothèque pour créer des Component et rendre ces Component sur l'interface. Le ReactJS n'avait pas la capacité de gérer l'état des applications. Peu de temps après, Facebook a introduit une bibliothèque Javascript nommé **Flux** pour gérer l'état des applications et elle est une bibliothèque créée pour soutenir le React.

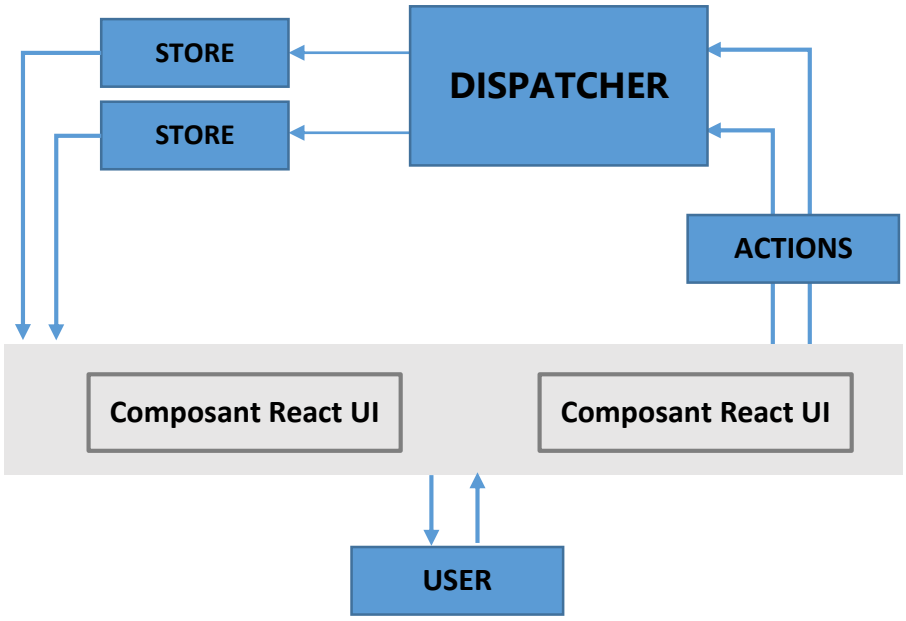
En ce temps-là, Dan Abramov a étudié le Flux de Facebook et le langage ELM. Il était influencé par l'architecture ELM, et a trouvé la complexité de Flux. En mai 2015 Dan Abramov a annoncé une nouvelle bibliothèque appelée **Redux**, elle est basée sur l'architecture de ELM et éliminé la complexité de Flux.

Après sa naissance, le Redux a suscité un vif écho et a immédiatement attiré l'attention de la communauté React et même Facebook a également invité Dan Abramov à travailler. Actuellement, le Redux et le Flux existent en parallèle, mais le Redux est populaire et plus largement utilisé.

16.2. L'architecture de Flux

L'architecture de Flux était introduit pour la première fois par Bill Fisher et Jing Chen à la conférence Facebook F8 en 2014. L'idée est redéfini le modèle MVVM (Model View - View Model) qui était largement utilisé auparavant avec le concept de "flux de données unidirectionnel" (unidirectional data flow).

Des actions (actions) et des événements (events) dans le Flux vont passer par un "circuit fermé" avec la forme ci-dessous :



Les parties dans l'architecture de FLUX :

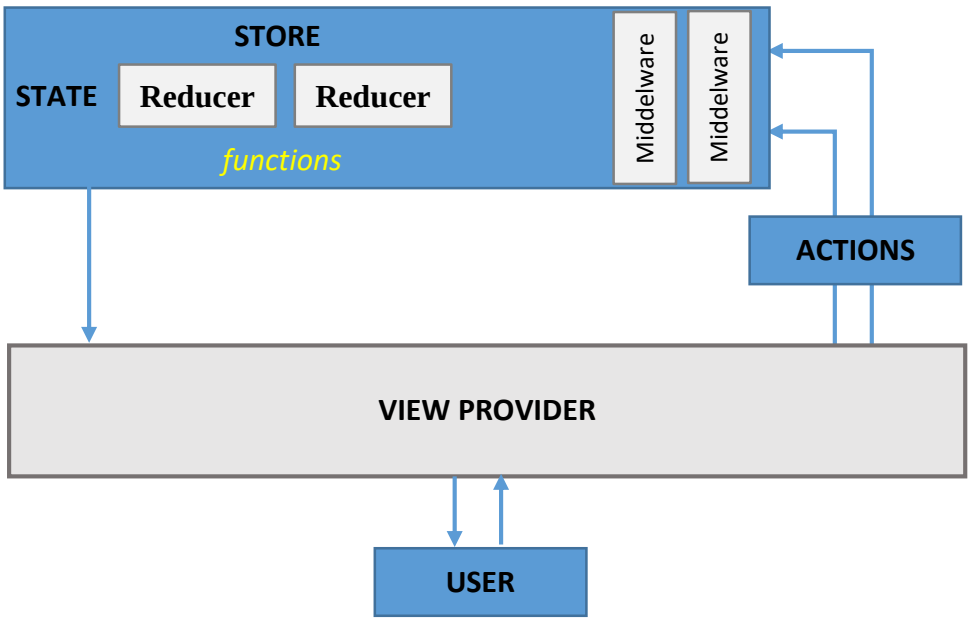
- **VIEW** : une composition hiérarchique (hierarchical composition) des React Component.
- **ACTION** : Est un objet pur créé pour stocker les informations relatives à un événement de l'utilisateur (Clique sur l'interface,..), il comprend les informations telles que : type de l'action, temps et location, ses coordonnées et quel state qu'il vise à changer.
- **DISPATCHER** : Le seul point de l'application à recevoir des objets Action à gérer.
- **STORE** : Store écoute des Action, gère des données et l'état de l'application. Les Store basent sur les objets action pour répondre au USER INTERFACE correspondant.

16.3. L'architecture de Redux

Redux apprend l'architecture de Flux mais il omit la complexité inutile.

- Redux n'a pas de concept DISPATCHER.
- Redux n'a que STORE au lieu de plusieurs STORE comme le Flux.
- Les objets Action sera recus et gérés directement par STORE.

Ci-dessous l'illustration de l'architecture de REDUX :



Les parties dans l'architecture de REDUX :

- **VIEW PROVIDER** : Représente un View Framework pour inscrire avec STORE. Dans lequel, View Framework peut être React ou Angular,...
- **ACTION** : un objet pur créé pour stocker les informations relatives à l'événement d'un utilisateur (cliquez sur l'interface, ...). Il inclut les informations telles que: le type d'action, l'heure de l'événement, l'emplacement de l'événement, ses coordonnées et quel state qu'il vise à modifier.

- **STORE** : Gère l'état de l'application et a fonction dispatch (action).
- **MIDDLEWARE** : (Logiciel intermédiaire) Fournit un moyen d'interagir avec les objets Action envoyés à STORE avant leur envoi à REDUCER. A Middleware, vous pouvez effectuer des tâches telles que la rédaction de journaux, la génération d'erreurs, la création de requêtes asynchrones (asynchronous requests) ou la distribution (dispatch) de nouvelles actions, ...
- **REDUCER** : (Modificateur) Une fonction pure pour renvoyer un nouvel état à partir de l'état initial. Remarque : REDUCER ne modifie pas l'état de l'application. Au lieu de cela, il créera une copie de l'état d'origine et le modifiera pour obtenir un nouvel état.

Store

Redux permet de stocker dans un seul objet appelé "the store" tous les états (state) de l'application. Cet objet est "la seule source de vérité" (single source of truth) et il est accessible par tous les composants.

L'objet store possède trois méthodes :

- **subscribe** qui permet a tout écouteur (listener) d'être notifié en cas de modification du store. Les gestionnaires de vues (comme React) vont souscrire au store pour être notifié des modification et effectuer mettre à jour l'interface graphique en conséquence.
- **dispach** qui prend en paramètre une action et exécute le reducer qui va, à son tour, mettre à jour le store avec un nouvel état.
- **getState** qui retourne l'état courant du store. L'objet retourné ne doit pas être modifié.

Reducer

Pour mettre à jour le **store**, il n'est pas possible de le faire directement puisque redux s'inscrit dans le paradigme de la programmation fonctionnelle et donc de l'immuabilité.

On va utiliser la fonction appelée "**reducer**" qui reçoit en arguments le "state" et une "action" et qui retourne le nouveau "state" en fonction de l'"action". C'est au programmeur de décider quelle technique il emploie pour assurer l'immuabilité : spread operator, assign, immer (produce) ....

Il y aura en général plusieurs "reducer" dans une application (un par composant).

Action

Action est un **objet littéral** (plain object) qui représente ce qu'il vient de se passer. On peut l'assimiler à un événement. Cet objet (événement) est envoyé en argument à la méthode "**dispatch**" de l'objet "**store**". La méthode "dispach" est l'unique point d'entrée du "store" ce qui va permettre de contrôler plus facilement les actions des utilisateurs. C'est grâce à cela que l'on va pouvoir "loger" toutes les modifications de l'interface (cf Redux dev tools) ou que l'on va pouvoir facilement mettre en place des mécanismes de "défaire et refaire".