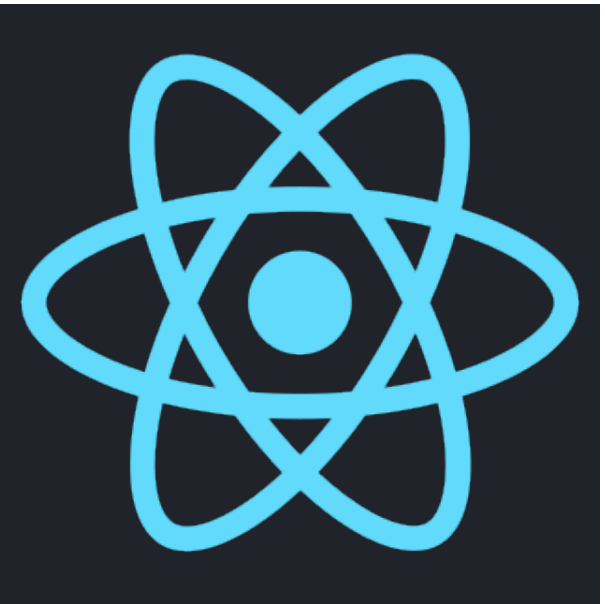
Technologies Front-End

Framework JS



Module « React »

Made by Valentin AMATHIEU

Redux-saga

SMS et asynchronicité

» Un des principaux intérêt de redux consiste en la séparation de la partie logique et données, de la partie vue.

» Mais lorsque l'on introduit des opérations asynchrones (temporisation, récupération de données en ligne, ...), il semble difficile de les gérer directement au niveau de notre store.

» En effet, un reducer est une fonction synchrone !

» On se retrouve donc avec une logique fracturée, une partie synchrone étant gérée par notre store (reducers) et l'autre asynchrone gérée au niveau de la partie vue.

» Le plus pratique et logique serait de faire en sorte de pouvoir traiter des opérations asynchrones, au niveau de notre store, d'une manière ou d'une autre.

» C'est là que le middleware redux-saga intervient !

Installation

» Pour utiliser redux-saga, vous devez déjà avoir installé redux dans votre application.

» Installer ensuite le package : npm install redux-saga

» La dernière diapositive vous montrera comment brancher ce middleware sur votre store redux.

Les sagas

» Redux-saga est un middleware permettant de gérer des "effets de bords" dans une application gérées avec Redux.

» L'idée est de brancher ce middleware au niveau de notre store, et de le faire démarrer des processus, appelées sagas.

» Une saga est une fonction de type générateur.

» Les générateurs sont des structures assez bien adaptées pour réaliser des traitements asynchrones.

» Ainsi, il va être possible pour nous de créer un nouveau genre d'action, pouvant être gérées, non pas directement par le store, mais par nos sagas !

» Ces sagas pourront ainsi communiquer avec le middleware (grâce au mot-clé yield), par l'intermédiaire d'effets, tout cela de manière asynchrone.

Exemple de saga

» Voici un générateur pouvant être utilisé en tant que saga

```
function* helloSaga() {
    console.log('Hello Sagas !');
}
```

» Ici, un générateur plus complexe, communiquant des effets au middleware

```
function* incrementAsync() {
    yield call(delay, 1000);
    yield put({ type: 'INCREMENT' });
}
```

Les effets

- Ces effets sont des objets représentant des instructions pour le middleware.
- Parmi ces instructions, certaines permettent au middleware de communiquer avec le store (dispatch d'action, ...)
- Le principe est simple :
  - Un effet est envoyé au middleware depuis la saga
  - L'appel à cet effet peut ou non être bloquant :
    - Soit la saga est mise en pause, en attendant le résultat de l'exécution des instructions décrites par l'effet
    - Soit l'exécution de la saga continue en parallèle
  - Lorsqu'un un effet bloquant est terminé, le middleware reprends le cours de l'exécution de la saga, jusqu'au prochain effet à envoyer (prochain yield !)

Types d’effet

- Un effet peut être une promesse que vous créez vous-même ou bien une fonction prédéfinie (factory) de l'API redux-saga. En voici une liste non exhaustive :
  - Promesse : Quand une promesse est communiquée au middleware, la saga courante est suspendue jusqu'à la complétion de cette promesse. Une fois la promesse complétée, le middleware reprend la saga en exécutant son code jusqu'au prochain yield.
  - put(action) : Effet non bloquant. Informe le middleware qu'il doit dispatcher une action au store.
  - call(fn, ...args) : Effet bloquant. Informe le middleware qu'il doit exécuter la fonction avec les arguments fournis.

Types d’effet

» Suite :

- » takeEvery(action, workerSaga) : Effet non bloquant. Indique au middleware d'intercepter les dispatch sur une "action" et de déléguer sa prise en charge à une saga dédiée, communément appelée une Worker saga. On appelera, par convention, la saga utilisant ce genre d'effet, une Watcher saga
- » all([saga1, saga2, ...]) : Afin de pouvoir lancer plusieurs sagas en parallèle dans notre application, on peut utiliser la fonction "all", qui prendra en paramètre un tableau contenant une liste de saga à démarrer. Une saga utilisant "all" est, par convention, appelée une Root saga.

Testabilité

» Certains de ces effets peuvent sembler superflu.

» Par exemple, "put" permet dispatcher de dispatcher une action au store.

» Mais n'aurait-on pas pu directement faire appel à dispatch que l'on fournirait à notre sage en paramètre ?

» Si, mais pour des questions de testabilité, il est plus intéressant de travailler avec des objets représentants les informations des actions de nos sagas, plutôt que de vouloir tester le résultats de ces actions.

Démarrer une saga

» Pour démarrer une saga :

- » Le middleware de redux-saga doit être appliqué au store
- » La saga (généralement une root saga) doit être démarrée avec la méthode "run" du middleware redue-saga

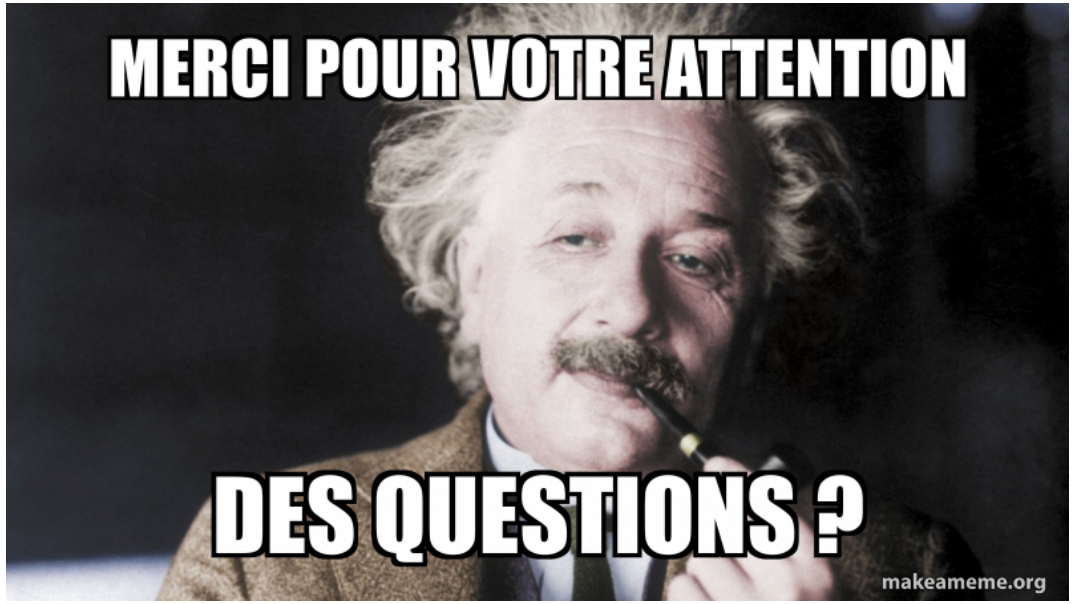
Démarrage d’une saga

» Exemple d’une saga démarré après la création du store

```
import { createStore, applyMiddleware } from 'redux';
import createSagaMiddleware from 'redux-saga';

import reducer from ‘./reducers';
import { rootSaga } from './sagas';

const sagaMiddleware = createSagaMiddleware();

const store = createStore(reducer, applyMiddleware(sagaMiddleware));

sagaMiddleware.run(rootSaga);
```

Question Time

?



Atelier Time

Exercice

« Ping Pong »

Fin du mon.. module !