Redux Saga

1. Redux-Saga là gì?

Redux-Saga *là một thư viện redux middleware, giúp quản lý những side effect trong ứng dụng redux trở nên đơn giản hơn. Bằng việc sử dụng tối đa tính năng Generators (function*) của ES6, nó cho phép ta viết async code nhìn giống như là synchronous.*

1.1. Side effect là gì??

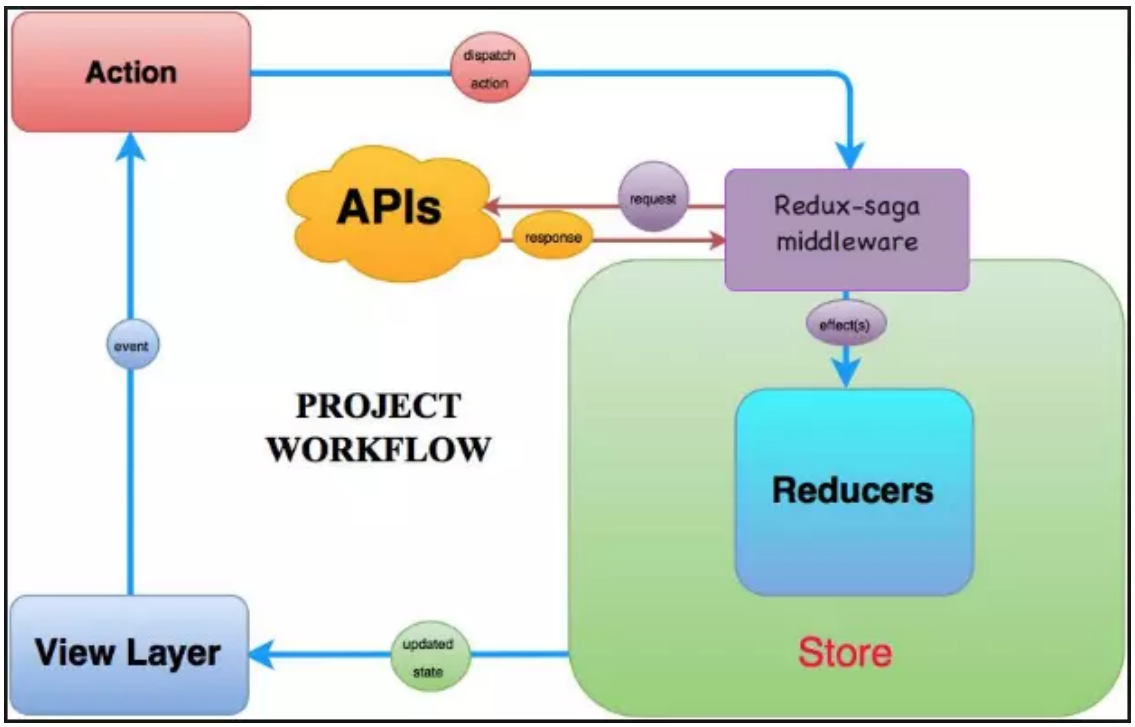
Ta đã biết tất cả những xử lý ở REDUCER đều phải là synchronous và pure tức chỉ là xử lý đồng bộ. Nhưng trong ứng dụng thực tế thì cần nhiều hơn vậy ví dụ như asynchronous (thực hiện một số việc như gọi một hàm AJAX để fetch dữ liệu về nhưng cần đợi kết quả chứ kết quả không trả về ngay được) hoặc là impure (thực hiện lưu, đọc dữ liệu ra bên ngoài như lưu dữ liệu ra ổ cứng hay đọc cookie từ trình duyệt… đều cần đợi kết quả). Những việc như thế trong lập trình hàm gọi nó là ***side effects****.*

1.2. Generator function là gì?

Khác với function bình thường là thực thi và trả về kết quả, thì ***Generator function*** *có thể thực thi, tạm dừng trả về kết quả và thực thi tiếp. Từ khóa để làm được việc đấy là "YIELD". Generator được đưa ra cách đây mấy chục năm nhưng đến ES2015 mới được bổ sung, các ngôn ngữ khác đã được bổ sung tính năng này như C#, PHP, Ruby, C++, R…*

2. Redux-Saga hoạt động như thế nào?

Khi View Layer thực hiện một Action, thì Action này sẽ được đưa tới Reducers, sẽ có các đoạn logic xử lý cho các Action tương ứng. Sau khi xử lý xong sẽ được cập nhật ra Store với các đoạn logic mà chúng ta viết bên Reducers. Ta thấy Redux-saga là một middleware nằm giữa Action và Reducers. Diễn giải là khi View Layer thực hiện một Action, trước khi Action được đưa tới Reducers thì nó sẽ được thực hiện các side effect (call API), sau khi có dữ liệu trả về từ API thì nó sẽ tiếp tục đưa dữ liệu này vào trong Reducers và tiến hành cập nhật dữ liệu ở trong Store



Đối với logic của saga, ta cung cấp một hàm cho saga, chính hàm này là hàm đứng ra xem xét các action trước khi vào store, nếu là action quan tâm thì nó sẽ thực thi hàm sẽ được thực thi, nếu bạn biết khái niệm hook thì hàm cung cấp cho saga chính là hàm hook. Điều đặc biệt của hàm hook này nó là một generator function, trong generator function này có `yield` *và mỗi khi* `yield` *ta sẽ trả về một plain object. Object trả về đó được gọi Effect object. effect object này đơn giản chỉ là một object bình thường nhưng chứa thông tin đặc biệt dùng để chỉ dẫn middleware của Redux thực thi các hoạt động khác ví dụ như gọi một hàm async khác hay put một action tới store. Để tạo ra effect object đề cập ở trên thì ta gọi hàm từ thư viện của saga là* `redux-saga/effects`.

3. Tại sao tôi phải sử dụng Saga?

Trích dẫn trong document của redux-saga:

Contrary to redux thunk, you don't end up in callback hell, you can test your asynchronous flows easily and your actions stay pure. Tạm dịch: trái với redux thunk, bạn không cần phải phát dồ lên với các callback trong mỗi action, đến với saga đi, bạn có thể test các async action với một quy trình dễ dàng mà không làm hư các action đó 😛

4. BASIC HELPERS

Là các helpers hoạt động ở tầng thấp trong saga APIS.
Dưới đây là những helpers thông dụng nhất mà bạn có thể dùng để chạy các effects của mình:

- `takeEvery()`
- `takeLatest()`
- `take()`
- `put()`
- `call()`

4.1. takeEvery()

```
import { takeLatest } from 'redux-saga/effects'

function* watchMessages() {
  yield takeEvery('ADD_MESSAGE', postMessageToServer)
}
```

Generator `watchMessages` *sẽ tạm dừng cho đến khi action* `ADD_MESSAGE` *được thực thi, và mỗi khi* `ADD_MESSAGE` *được thực thi, nó sẽ gọi đến function* `postMessageToServer`*(số lần được gọi có thể là vô hạn và* `postMessageToServer` *sẽ chạy mà không cần biết trước đó function* `postMessageToServer` *đã chạy xong hay chưa).*

4.2. takeLatest()

Một helper thông dụng và nổi tiếng khác chính là `takeLatest()`*, nó gần như giống hệt người anh em* `takeEvery()` *của mình, chỉ có cái khác là nó chỉ cho phép một function được chạy trong một thời điểm. Tức là nếu như trước đó có một function đang chạy, nó sẽ hủy function đó và chạy lại lần nữa với dữ liệu mới nhất.*

Cũng giống như `takeEvery()`*, generator sẽ không bao giờ ngừng lại và tiếp tục chạy cho đến khi một action chỉ định được diễn ra.*

4.33. take()

`take()` *khác biệt ở chỗ nó chỉ đợi đúng một lần duy nhất. Khi action nó đợi được diễn ra,* `promise` *sẽ được* `resolve` *và vòng lặp sẽ tiếp tục.*

4.4. put()

Khi muốn dispatch một action trong redux store, thay vì phải truyền vào redux store hoặc là dispatch action đến saga, bạn có thể dùng luôn ***put:***

```
yield put({ type: 'INCREMENT' })
yield put({ type: "USER_FETCH_SUCCEEDED", data: data })
```

`put()` *sẽ trả về một plain object, sau này khi viết test cho redux sẽ đơn giản hơn rất nhiều cho chúng ta.*

4.5. call()

Khi bạn muốn gọi một số function ở trong saga, bạn có thể làm thế bằng cách viết một function trả về promise dạng như:

```
delay(1000)
```

Nhưng viết như trên sẽ rất khó cho chúng ta khi viết test. Thay vào đó, `call()` *cho phép bạn bọc function trên và trả về một object dễ làm việc hơn:*

```
call(delay, 1000)
```

Đây là kết quả trả về:

```
{ CALL: {fn: delay, args: [1000]}}
```