RxJS : les clefs pour comprendre les observables

Speakers : Thierry Chatel
Format : Conférence
Date : 20 avril 2018
Slides : https://tchatel.github.io/conf-rxjs/RxJS.pdf

Qu'est-ce qu'un observable ?

Flux de données qui arrive successivement dans le temps.
Permet de récupérer les résultats d'une opération asynchrone.
Un observable peut récupérer :

- qu'une seule donnée : réponse d'un serveur
- plusieurs données : via les websockets par exemple

Fonctionne également pour les opérations synchrones

3 types de notification :

1. next : arrivée d'une nouvelle valeur
2. error : terminaison en erreur. A la première error, c'est terminé.
3. complete : terminaison en succès. C'est terminé pour l'observable.

Sur les flux de données (les observables), on peut appliquer des opérations.
On peut voir les observables comme une suite de notes de musique.
On va pouvoir brancher les opérateurs sur un observable pour entendre quelque chose.
Autre image : assemblage de tuyaux.

Utilisation du pipe :

```
return notif$.pipe(
        filter(notif => notif.type === 'Alert'),
        map(notif => notif.code + ': ' + notif.message)
) ;
```

Opérateurs sur un tableau :

- [0, 30, 22, 5, 60, 1].filter(x => x > 10) // 30, 22, 60
- every
- map, reduce

Opérateurs similaires sur un observable : filter, map, reduce

Opérateurs RxJS

RxJS 6 vient avec 105 opérateurs.
Webpak est capable de ne prendre que les fonctions utilisées et non pas les 105.

Un opérateur RxJS :

- Renvoie toujours un nouvel observable sans modifier celui d'origine
- Fonction pure

Grosse différence avec les tableaux : nécessité de s'abonner.
Pour s'abonner, on appelle la méthode subscribe() de l'observable. On peut lui passer 2 types d'objets.
On peut annuler la souscription par un subscription.unsubscribe() => permet d'éviter parfois des memory leaks ou de récupérer des données qui ne nous intéressent pas.

Observables Cold et Hot

2 types d'observables : hot & cold

- Cold : lors de la lecture, on commence au début. Exemple du fichier MP3 => **unicast**
- Hot : écoute d'un morceau en cours de route. Une seule source est diffusée à toutes les souscriptions. Exemple de la radio => **multicasted**

Pas de moyen de savoir si le morceau de musique est chaud ou froid depuis un observable.

Création d'un observable cold avec la fonction create():

```
const observable: Observable =
        create(observer =>
                { observer.next(1);
                  observer.next(2);
                  setTimeout(() =>
                          { observer.next(3);
                           observer.complete(); }, 1000); });
```

La création d'un observable hot passe par l'utilisation d'un Subject :

```
const subject = new Subject();
subject.subscribe(v => console.log('myObserver: ' + v));
subject.next(1);
```

RxJs propose différentes variante de sujets :

- le BehaviorSubject conserve un état (valeur courante). Au moment de la souscription, on reçoit cette valeur.
- ReplaySubject : dernières valeurs d'un buffer
- AsyncSubject : résultat d'une opération asynchrone

Fonctions utilitaires : fromPromise, fromEvent …

Possibilité de transformer un cold en hot avec la fonction shareReplay.

Observables d'ordre 2

Observable d'observable : observable de données dont chaque donnée est un observable.
La fonction groupBy regroupe des données au sein d'observable
En général, on souhaite aplatir ces observables. Des opérateurs sont prévus pour cela.
Exemple : switchMap => map() puis switch()
Le switchMap est couramment utilisé pour de l'auto-complétion : on ne souhaite pas connaître la réponse d'un résultat lorsque l'utilisateur a déjà ressaisi quelque chose. On ignore le résultat.

How-to : services asynchrones

- Toujours renvoyer un observable
- Ne pas souscrire pour fournir les vraies données : on ne saura pas quand elles arrivent
- Dans les services, on va enchaîner les observables. Ce n'est que lors de l'affichage qu'on souscrit au résultat.

Comment gérer les erreurs ?

- Remonter les erreurs au niveau où on peut les traiter (comme les exceptions)
- Pour déclencher une erreur : throw 'fail' ; ou throwError('fail')
- Intercepter une erreur : catchError
- Réessayer : retry ou retryWhen

Exemple : détecter un double click

```
buildDoubleClicksObservable(button: HTMLButtonElement,
                            delay: number): Observable<MouseEvent> {

  const clicks$: Observable<MouseEvent> =
    fromEvent(button, 'click');

  const doubleClicks$: Observable<MouseEvent> =
    clicks$.pipe(
      buffer(clicks$.pipe(debounceTime(delay))),
      filter(clickGroup => clickGroup.length === 2),
      map(clickGroup => clickGroup[0])
   );

  return doubleClicks$;
}
```

Opérateur dinstinctUntilChanged : fournit une valeur seulement si elle est différente de la précédente

Convention : mette un $ à la fin du nom des variables qui contiennent un observable