

RxJS

Introduction

RxJS est une librairie qui est présente dans Angular et qui permet de faire de la programmation réactive. Angular utilise cette librairie dans son module HttpClient qui gère les requêtes HTTP.

RxJS permet de manière générale de gérer des flux de données asynchrones ou synchrones en utilisant le design pattern Observable.

RxJS est constitué de deux éléments fondamentaux : les observables et opérateurs.

Qu'est-ce qu'un Observable

Derrière ce dernier il y a évidemment le pattern Observable que vous connaissez déjà.

Un Observable est un objet qui émettra des informations, par exemple cela peut être des données d'un serveur HTTP, ou un événement dans le DOM, ou des données provenant d'une WebSocket, ...

On l'associe à un Observer. Si l'Observable émet de l'information l'Observer s'exécutera.

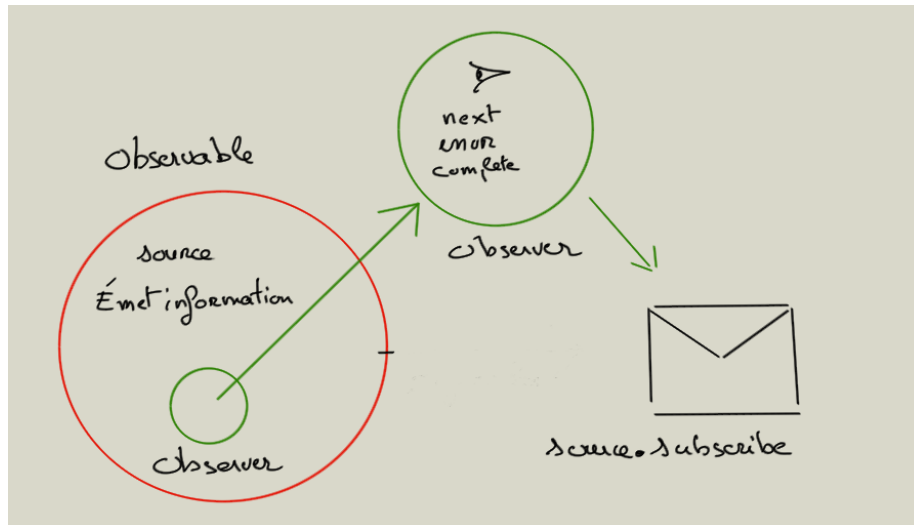
Un Observer est constitué de 3 méthodes : next(), error() et complete()

La méthode next « pushera » des données à émettre, la méthode error gère les exceptions et la méthode complete termine le flux.

Caractéristiques

- Un Observable s'appelle lorsque vous décidez de l'appeler, il faut souscrire (subscribe) à ce dernier pour récupérer les données/flux.
- Vous pouvez stopper un Observable pendant son exécution.
- Vous pouvez préparer en amont les données que vous allez recevoir avant la souscription. Des fonctions comme map, reduce, filter, ... permettent de préparer les données. Ce sont les opérateurs (operators) de RxJS.
- Il ne produira des valeurs que si vous y souscrivez, dans ce cas il est dit froid (cas général) et sinon il peut produire des valeurs (nombre de cliques dans une page) il sera dit chaud. Dans tous les cas il faudra souscrire à ce dernier pour récupérer les valeurs (principe lazy loading).

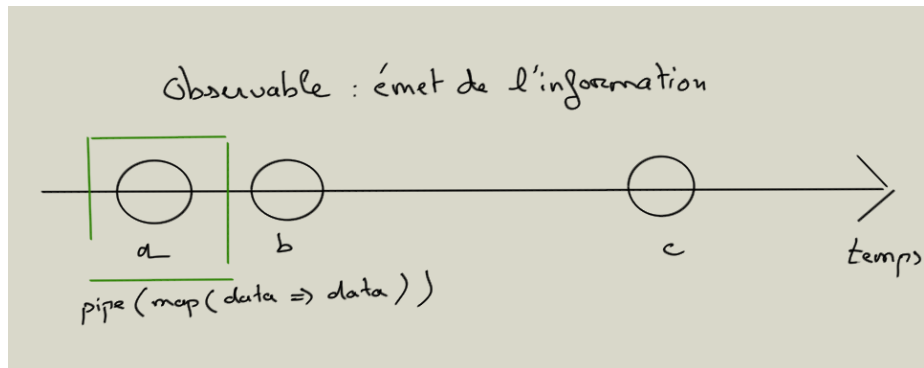
Représentation d'un Observable



Les opérateurs

RxJS propose des opérateurs, c'est une fonction qui se placera entre l'Observable et la souscription et qui permet de filtrer ou modifier les données reçues avant la souscription. Voici quelques opérateurs de cette librairie :

- `map()` permet de modifier les valeurs reçues, faire des calculs sur des chiffres modifier des objets.
- `filter()` : filtre les valeurs reçues en fonction d'un critère logique, si c'est un nombre par exemple filtrer les nombres pairs.
- `scan()` permet de faire des calculs de proche en proche comme la somme de valeurs numériques : `scan((acc, current) => acc + current, 0)`. La valeur 0 est la valeur initiale pour acc.
- `reduce()` est identique à `scan()`, mais retourne le calcul total à la fin.
- `max()` permet de déterminer le maximum dans une liste d'objet ou numérique, elle prend deux valeurs : `max((a,b) => a > b ? 1: -1)` pour trouver la valeur numérique la plus grande.



Exemple

```
const { Observable } = require('rxjs');
const { map, filter } = require('rxjs/operators');

// Création de l'observable
// on passe un objet observer
const numbers = Observable.create((observer) => {
  let count = 1;
  const interval = setInterval(() => {
    observer.next(count);
    count++;
  }, 1000);

  // cette méthode sera lancée si on se désinscrit
  return () => clearInterval(interval);
});

// mapping data
const pipeNumbers = numbers.pipe(
  map(number => number + 10),
  filter(number => number % 2 === 0)
);

// s'inscrire
const subscribeOne = pipeNumbers.subscribe(
  num => console.log(`data : ${num}`)
);

// se désinscrire au bout de 10 secondes
setTimeout(() => {
  subscribeOne.unsubscribe();
}, 10000);
```

Exercice 1

Dans la suite des exercices ils sont écrits en Javascript, pour exécuter le code utilisez node : `node monfichier.js`

Dans le dossier Observable récupérez la source `ex1_points.js` dans le dossier `chap12_sources` et placez ce fichier dans ce dossier. Tapez la ligne de code suivante pour installer rxjs :

```
npm install rxjs
```

Dans le fichier vous avez une liste de points à deux coordonnées, à l'aide de l'opérateur `max` déterminer le point le plus éloigné du point $O(0,0)$, rappelons que `Math.sqrt(x2 + y2)` vous donnera la distance d'un point par rapport à au point $O(0,0)$.

Vous afficherez votre résultat dans un `console.log`

remarques : l'objet `of` émet une série de valeurs, c'est un Observable.

Documentation max RxJS

Exercice 2

Récupérez la source suivante `ex2_game.js` et mettez là dans le dossier Observable. Vous n'avez pas à réinstaller RxJS.

Nous disposons d'une liste d'utilisateurs `users`. Vous allez souscrire à l'Observable `Users`.

Envoyez les `users` un à un et arrêtez l'envoi lorsque la liste des `users` est vide. Utilisez la méthode `observer.complete()`; elle s'exécutera et arrêtera l'envoi.

Avant l'envoi pensez à ordonner les résultats en fonction du score par ordre de score décroissant.

Dans la souscription à l'observable vous pouvez utiliser l'objet `observer` lui-même de la manière suivante :

```
source.subscribe( { next() { /* du code */ }, complete() { /* du code */ } } )
```

Filtrez les scores supérieur à 100 à l'aide du filtre **`filter`**; affichez les résultats avec `console.log`.

Mettez également la première lettre des noms en majuscule, pour se faire utiliser le pipe `map`, qui mapperà chacune des données envoyée et donc vous permettra de modifier sa valeur avant la récupération dans la souscription.

Subject

Un Subject est à la fois un Observable et un Observer. On peut donc comme un Observable souscrire à ce dernier pour recevoir le flux de la souscription et également lui envoyer un flux de données.

```
const { Subject } = require('rxjs');
let subject = new Subject();

// Un subject peut être observé (observable) et on peut recevoir le flux de données émit.
let subscription = subject.subscribe(
  (n) => { console.log(`next : ${n}`); }, // next
  (e) => { console.log('error'); }, // error
  () => { console.log('completed'); } // completed
);

// Mais c'est également un observer et on peut donc lui envoyer des données qui seront obser
subject.next(1);
subject.next(2);

subject.complete();
```

Exercices opérateurs approfondissement

Dans la suite des exercices créez une page HTML et utilisez RxJS en CDN directement dans le fichier :

```
<script src="https://unpkg.com/rxjs@6.3.3/bundles/rxjs.umd.min.js">
</script>
```

Exercice pgcd

En utilisant l'opérateur mergeMap et deux champs de formulaire donnez le pgcd de deux nombres entiers. Vous utiliserez fromEvent pour récupérer les valeurs de chaque champ. Faites une page HTML et affichez le résultat sous les deux champs de saisis.

```
// Dans le fichier js pour importer opérateurs et observables utilisez
// la syntaxe suivante :
const { map, mergeMap, pluck } = rxjs.operators;
const { fromEvent, interval, timer } = rxjs;

// Pour récupérer l'événement sur le champ text id="num1"
const ob1$ = fromEvent(num1, 'input');
```

Exercice fibonnaci

Un générateur est un objet sur lequel il faut itérer. Le mot réservé `yield` garde en mémoire la position de l'itération, ainsi dans l'exemple ci-dessous, vous pouvez afficher tous les nombres entiers en partant de 1 en itérant sur la fonction `generator`. L'intérêt des générateurs est essentiellement porté sur l'empreinte mémoire, en effet comme il n'y a pas de "return" dans la fonction génératrice on a pas besoin de stocker le résultat dans une variable, on peut directement exploiter la valeur dans le script au fur et à mesure que le générateur renvoie une valeur :

```
function* generator(i = 0) {
  while(true){
    i++;
    yield i;
  }
}

// itération
for(let num of generator()) console.log(num);
```

Définissez un générateur qui permet d'afficher les termes numériques entiers de la suite de Fibonacci.

Puis à l'aide de l'opérateur `switchMap` et de l'observable `interval` créez un script avec un bouton HTML permettant à chaque fois que l'on clique sur ce dernier de relancer la génération des termes de la suite de Fibonacci.

Vous utiliserez également l'opérateur `take` pour stopper l'intervall au bout d'un certain nombre de fois.

Précision sur SwitchMap pour l'exercice

Cet opérateur permet d'éviter la génération à l'infini d'observables. Dans l'exemple qui suit à chaque fois que l'on clique on crée en mémoire un nouvel Observable `interval` :

```
const click$ = fromEvent(generate, 'click');
const interval$ = interval(1000);

// à chaque clique on crée un observable interval à l'infini
const $start = click$.subscribe(
  x => interval$.subscribe(y => console.log(y))
);
```

Une autre approche plus intéressante est la suivante à chaque clique on switch sur un nouvel Observable à l'aide de l'opérateur `switchMap` :

```
const click$ = fromEvent(generate, 'click');
```

```

const interval$ = interval(1000);

// à chaque clique on crée un observable interval à l'infini
const start$ = click$.pipe(
  switchMap(event => interval$)
);

/*
à chaque clique on retourne un nouvel Observable
interval qui se relance en supprimant le précédent
Observable :
*/
start$.subscribe(x => console.log(x))

```

Exercice nombre de cliques

Créez un page avec un bouton nous allons contrôler le nombre de clique(s) de l'utilisateur, si celui-ci fait trop de cliques, c'est-à-dire un nombre supérieur à 3 dans un délais de 250ms nous lui signalerons.

Faites une approche purement JS pour faire cet exercice dans un premier temps.

Version RxJS

Pour ce même exercice vous allez maintenant réfléchir à la problématique ci-dessus en vous aidant de RxJS et de certains opérateurs :

Dès que l'événement click a eu lieu attendre 250ms si rien ne c'est passé alors faire un buffer de clicks, sinon si quelque chose se passe un click rapide attendre avant de bufferiser les clicks.

Filtrez les résultats pour garder que les résultats contenant au moins 3 cliques.

Affichez un message si le nombre de clique est supérieur ou égale à 3.

Voici les opérateurs que l'on peut utiliser pour résoudre ce problème classique :

```
debounce(() => timer(250))
```

Exercice 4

En utilisant un observable créer un compteur dans l'application.

Créez le script dans le TypeScript du component suivant : app.component.ts

Search Albums :

Previous **1** 2 3 4 Next

Page principe Albums Music

1 Pop

consequat excepteur

Cillum proident commodo do non esse cillum incididunt officia qui occaecat.

Importez Observable et interval de RxJS :

```
import { Component } from '@angular/core';

import { interval, Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  title = 'app-music';
}
```

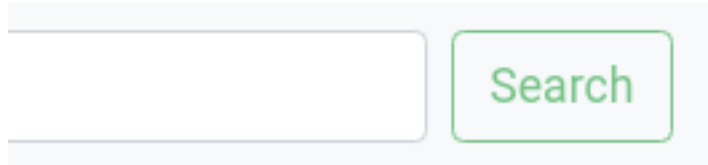
La classe interval génère une suite de nombres entiers incrémentés à intervalle de temps fixé.

```
const count = interval(1000); // toutes les secondes n+1
```

Souscrivez à cet Observable et affichez le nombre de secondes qui défilent dans l'application.

Améliorez le rendu du compteur en affichant les heures/minutes et secondes.

- time : 0 h 4 min 15 s



Indications : utilisez l'opérateur map pour mettre la logique avant souscription du compteur demandé.

Utilisez l'opérateur take de RxJS afin de stopper le flux au bout de 12heures (12*3600 secondes).

Exercice 5 (Subject)

Le but de cet exercice est de créer un player avec une barre de progression audio. Elle indiquera la durée de défilement des morceaux. Pour simplifier chaque morceau dure exactement 2 minutes. Cette durée est précisée dans le fichier album-mock.ts avec la clé duration.

Vous allez tout d'abord créer un component `AudioPlayerComponent`. Récupérez le code HTML dans le fichier `audio-player.component.html` du dossier source `Ex5_source`, vous trouverez une progress bar en HTML.

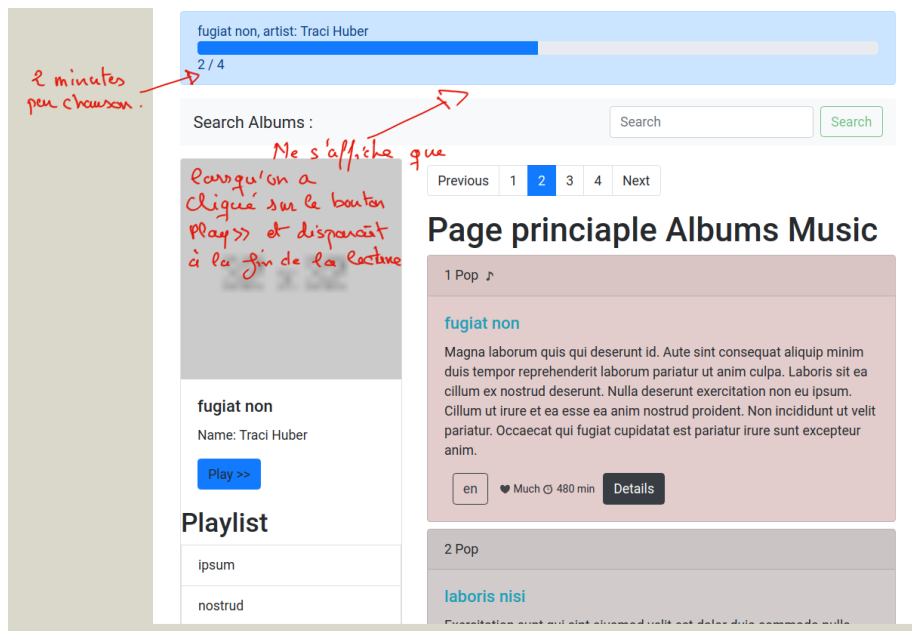
La communication entre les différents composants se fera de la manière suivante :

- Le component `AlbumDetailsComponent` possède le bouton "play" et envoie l'information de lecture au component `AlbumsComponent`.
- Le service `AlbumService` reçoit l'information et communique celle-ci au component `AudioPlayerComponent` qui se déclenche.
- Toute la logique de lecture se fera dans le component `AudioPlayerComponent`.

C'est dans le service que le sujet `SubjectAlbum` envoie l'information au component `AudioPlayerComponent`.

ng g c audio-player

Une fois cliqué sur le bouton « play » dans le détail d'un album affichez sous la bannière du site une information de progression de lecture des morceaux de l'album :



Placez le sélecteur de ce composant dans le composant HTML `app.component.html`

Création du Subject dans le service AlbumService

Importez tout d'abord la classe Subject de RxJS dans le service lui-même :

```
import { Subject } from 'rxjs';
```

Définissez l'objet suivant, ce sera notre Subject :

```
subjectAlbum = new Subject<Album>();
```

Rappelons qu'un Subject est à la fois un Observable et un Observer.

Vous allez créer deux méthodes : `switchOn` et `switchOff`, respectivement la première méthode mettra le statut de l'album à « on » et l'autre à « off » pour l'écoute d'un album de musique.

La mise à jour d'un album se fera de la manière suivante :

```
this.subjectAlbum.next(album);
```

Faites la même chose pour la méthode `switchOff`. Cette dernière n'aura pas besoin d'émettre quoi que se soit. Elle changera uniquement le statut dans la propriété `status` de l'objet de type `Album`.

Dans le composant `AlbumComponent` dans la méthode `playParent` modifiez le code comme qui suit :

```
playParent($event){
```

```

    this.status = $event.id; // identifiant unique
    console.log($event);
    // méthode dans le service
    this.ablumService.switchOn($event);
  }

```

Dans le component `AudioPlayerComponent` :

Dans le component `AudioPlayerComponent` vous allez écrire le code de l'Observable. Souscrivez à ce dernier afin de recevoir les modifications de l'album et gérez la progression de la barre d'écoute.

Les chansons durent 2 minutes et la propriété « `duration` » de l'objet `Album` donne le total des secondes de toutes les chansons, faites une proportion pour afficher la barre de progression dynamiquement.

Dans le code HTML vous devez utiliser une directive d'attribut Angular pour passer la valeur en pourcentage de la barre de progression, utilisez le code suivant. Le `ratio` devra être une propriété de votre component `AudioPlayerComponent` :

```

<div class="progress-bar" role="progressbar" [style.width]="ratio + '%" aria-valuenow=" ar

```