PARTIE 4: RxJS

RxJS

- La programmation réactive est une façon de construire une application basée sur des événements et d'y réagir.
- `RxJS` (Reactive Extensions for JavaScript) permet la gestion de flux d'événements synchrones ou **asynchrones**. Elle est intégrée dans Angular, notamment pour les requêtes HTTP (`HttpClient`), les formulaires réactifs, le routeur et d'autres modules qui exposent des **observables**.

OBSERVABLE / OBSERVER

- Un **observable** retourne un flux d'événements au fil du temps. Ces événements peuvent correspondre à:
  - des valeurs de même type (de 1 à des milliers). Ex: une réponse HTTP
  - une erreur. L'observable est détruit et n'émettra plus de valeur.
  - une terminaison. L'observable est détruit et n'émettra plus de valeur.
- Un **observable** est similaire à une collection de valeurs, comme un tableau, mais dont les valeurs arrivent de façon asynchrone.
- Ces flux sont écoutés (consommés) par les *observers* (abonnés)

EXÉCUTEZ LES EXEMPLES

Pour tester le code du cours:

- Ouvrez le projet TypeScript de test dans `stackBlitz`, un IDE web basé sur la technologie WebAssembly développée par le W3C.

CRÉER UN OBSERVABLE

- Instancier la classe *Observable* de RxJS et spécifier le type des valeurs du flux
- La fonction passée en paramètre du constructeur est en charge d'émettre les valeurs et les éventuelles erreurs, ainsi que d'éventuellement terminer le flux.
- Utiliser `next()` pour émettre une nouvelle valeur
- Utiliser `complete()` pour arrêter l'émission de nouvelles valeurs

```
const messages$ = new Observable<string>((observer) => {
    observer.next('Ceci est une info'),
    observer.next('Ceci est un warning'),
    observer.complete()
  })
```

CRÉER UN *OBSERVER*

- Un *observer* peut être un objet littéral qui définit quoi faire des émissions d'un **observable**. Il définit trois fonctions de *callbacks*: chacune correspond à un type d'événements : `next`, `error` (facultatif) et `complete` (facultatif).
- Un *observer* peut être *juste* une fonction de *callback*. Dans ce cas, elle s'applique pour les événements `next`.

```
observerObj = {
  next: (value: string) => console.log(value),
  error: (error: Error) => console.log(`une erreur est survenue: ${error}`),
  complete: () => console.log("fin")
};
```

```
observerFn =  (value: string) => console.log(value);
```

PATTERN OBSERVER

- Il faut s'abonner à un **observable** en utilisant sa méthode `subscribe()`, en lui passant en paramètre un *observer*, pour recevoir et traiter les valeurs qu'il émet.

VISUALISEZ LE RÉSULTAT

```
messages$.subscribe(observerObj);
// dans la console avec l'objet observerObj
//Ceci est une info
//Ceci est un warning
//fin
```

```
messages$.subscribe(observerFn);
// dans la console avec la fonction observerFn
//Ceci est une info
//Ceci est un warning
```

OPÉRATEURS DE CRÉATION RxJS

- Les opérateurs RxJS sont des fonctions de deux types:
  - celles qui permettent de créer un observable (`interval`, `of`, `from`...)

```
// exemples d'opérateurs qui créent des observables
import { of, from } from 'rxjs';

of(1, 4, 9)
  .subscribe((v) => console.log(`value: ${v}`));
from([1, 4, 9])
  .subscribe(v => console.log(`value: ${v}`));
// les deux observables affichent
// value: 1
// value: 4
// value: 9
concat(of(1, 4, 9), from([11, 44, 99]))
  .subscribe(v => console.log(`value: ${v}`));
// value: 1
// value: 4
// value: 9
// value: 11
// value: 44
// value: 99
```

OPÉRATEURS RxJS *PIPEABLE*

- La majorité des opérateurs RxJS (`map`, `filter`, ...) servent à transformer la collection de valeur émise par un **observable**
  - Ils s'utilisent dans la méthode *pipe* des **observables**
  - Ils prennent en entrée un observable
  - Ils retournent en sortie un observable qui émettra les valeurs transformées si on y souscrit
  - Il s'agit d'opérations pures (n'affectent pas l'observable en entrée).
  - Il est possible de les enchaîner dans l'ordre dans la méthode *pipe*.

```
interval(1000).pipe(
      map((x) => x * 2),
      filter((x) => x>10),
      take(4) // émet 4 valeurs et s'arrête
    ).subscribe((x) => console.log(`value: ${x}`));
// Après 6 secondes
// value: 12 // value: 14
// value: 16 // value: 18
```

EXEMPLES D'OPÉRATEURS RxJS

- `tap` retourne un observable identique et permet d'ajouter des effets de bord

interval(1000).pipe(map((x) => x * 2), filter((x) => x>10), take(4), // émet 4 valeurs et s'arrête tap(() => this.messagesService.add ('les 4 premiers entiers paires supérieurs à 10.', 'validation'))).subscribe((x) => console.log(`value: ${x}`));

EXEMPLES D'OPÉRATEURS RxJS

- `distinctUntilChanged` retourne un observable qui émet une valeur quand elle change de la précédente émise.

```
of(1, 4, 4, 4, 9, 4).pipe(
  distinctUntilChanged()
).subscribe(v => console.log(`value: ${v}`));
// value : 1
// value : 4
// value : 9
// value : 4
```

EXEMPLES D'OPÉRATEURS RxJS

- `mergeMap` permet de retourne un observable qui combine les valeurs émises par deux observables.

```
const obs2$ = of('a', 'b', 'c');
obs2$.pipe(
  mergeMap((x) => interval(1000).pipe(map(i => i+x)))
  ).subscribe(v => console.log(`value: ${v}`));
// value: 0a
// value: 0b
// value: 0c
// value: 1a
// value: 1b
// value: 1c
//...
```

ENTRAINEMENT RxJS-fruits

SUBJECT

- Si l'on veut faire émettre programmatiquement un nouvel événement à un observable, il faut utiliser un **observable spécial**: un `Subject` car les observables classiques sont en lecture seule. On ne peut que les écouter.
- Comme un `Subject` est `multicast`, tous ses abonnés sont avisés de ce nouvel événement.
- Ces obervables `Subject` permettent de déclencher des actions suite à un événement dans différents composants.

ENTRAINEMENT RxJS-cars

RxJS ET COMPOSANTS

- Dans les composants, les souscriptions se font classiquement dans le constructeur ou dans la méthode `ngOnInit()` (nécessaire si ses entrées et son DOM ont besoin d'être initialisés).

```
...
export class MyComponent implements OnInit {
  private readonly messages$ = new Observable<string>((observer) => {
    observer.next('Ceci est une info'),
    observer.next('Ceci est un warning'),
    observer.complete()
  })
  ngOnInit(): void {
    this.messages$.subscribe(observerObj);
    //ou this.messages$.subscribe(observerFn);
  }
}
```

EXERCICE DANS MONPREMIERPROJET

- Utilisez la fonction `interval` de RxJs, qui permet de créer un observable qui émet séquentiellement des nombres, pour afficher dans la console depuis combien de temps en secondes chaque alerte est affichée sur la page.
- Regardez ce qui s'affiche sur la console.
- Supprimez les alertes en cliquant sur les croix.
- Que constatez-vous dans la console?

FUITES DE MÉMOIRE

- A chaque nouveau composant créé, le passage dans `ngOnInit` provoque une nouvelle souscription à l'observable
- le fait d'enlever un composant d'un template détruit le composant mais la souscription à l'observable subsiste => **fuites de mémoire**
- il faut donc se désabonner des observables qui continueraient d'émettre si nous ne le faisons pas

SE DÉSABONNER DES OBSERVABLES (1/2)

- A la destruction du composant:

```
import { ..., OnDestroy, OnInit } from '@angular/core';
import { interval, Subscription } from 'rxjs';
...
export class Alert implements OnInit, OnDestroy {
  ...
  private subscription!: Subscription;

  ngOnInit(): void {
    this.subscription = interval(1000)
      .subscribe((x)
        => console.log(`Alerte n°${this.message.id} depuis ${x} secondes`));
  }
  ngOnDestroy(): void {
      this.subscription.unsubscribe();
    }
  }
```

SE DÉSABONNER DES OBSERVABLES (2/2)

- A la destruction du composant:

```
import { ..., OnDestroy, OnInit } from '@angular/core';
import { interval, Subscription } from 'rxjs';
...
export class Alert implements OnInit, OnDestroy {
  ...
  subscriptions: Subscription[] = [];

  ngOnInit(): void {
    const subscription = interval(1000)
      .subscribe((x)
        => console.log(`Alerte n°${this.message.id} depuis ${x} secondes`));
    this.subscriptions.push(subscription);
  }
  ngOnDestroy(): void {
    for (const subscription of this.subscriptions){
      subscription.unsubscribe();
    }
  }
```

LE PIPE async

- Ce pipe est utilisable quand les valeurs émises par un observable sont seulement utilisées dans le *template*
- Il fonctionne pour toutes les données obtenues de façon asynchrone (promesse ou observable).
- Il a l'avantage de gérer la souscription et le désabonnement aux observables.
- Comme les signaux, la réception d'une nouvelle valeur à traiter par un pipe `async` déclenche une détection de changements qui met à jour le *template*.

```
export class Alert{
  protected readonly  interval2$: Observable<number>= interval(1000);
  ...
}
```

```
...Cette alerte est affichée depuis {{ interval2$ | async }} secondes</p>
```

ALLER PLUS LOIN: RXJS

Présentation vidéo de André Staltz sur RxJS à NgEurope 2016