



RxJS : Les bases

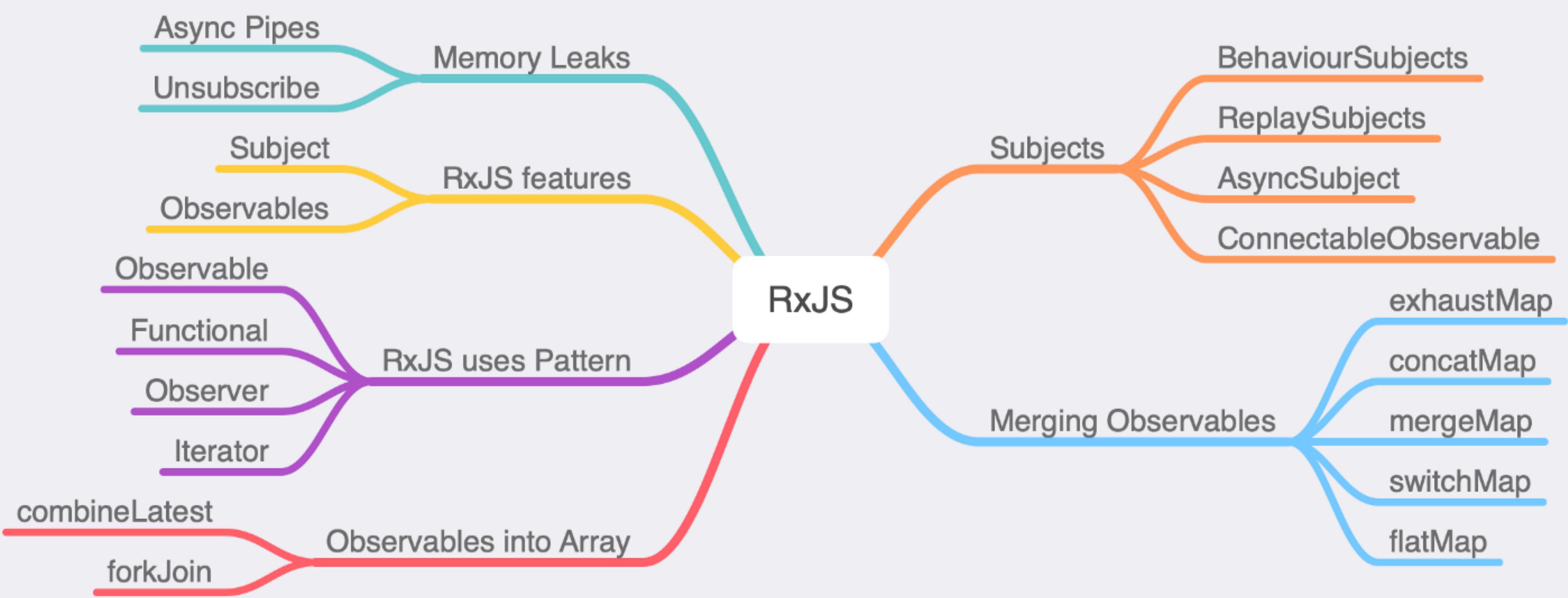
Préparé par NDAO Mame Coumba

odigo

Qu'est-ce que RxJS?

RxJS est une bibliothèque de programmation réactive utilisant des Observables, pour faciliter la composition de code asynchrone ou basé sur des rappels.





COLLECTIONNER

différentes sources de données et ses différents attributs pour former une séquence d'observable

MODIFIER

les attributs à partir de cette séquence dans un pipeline grâce aux opérateurs

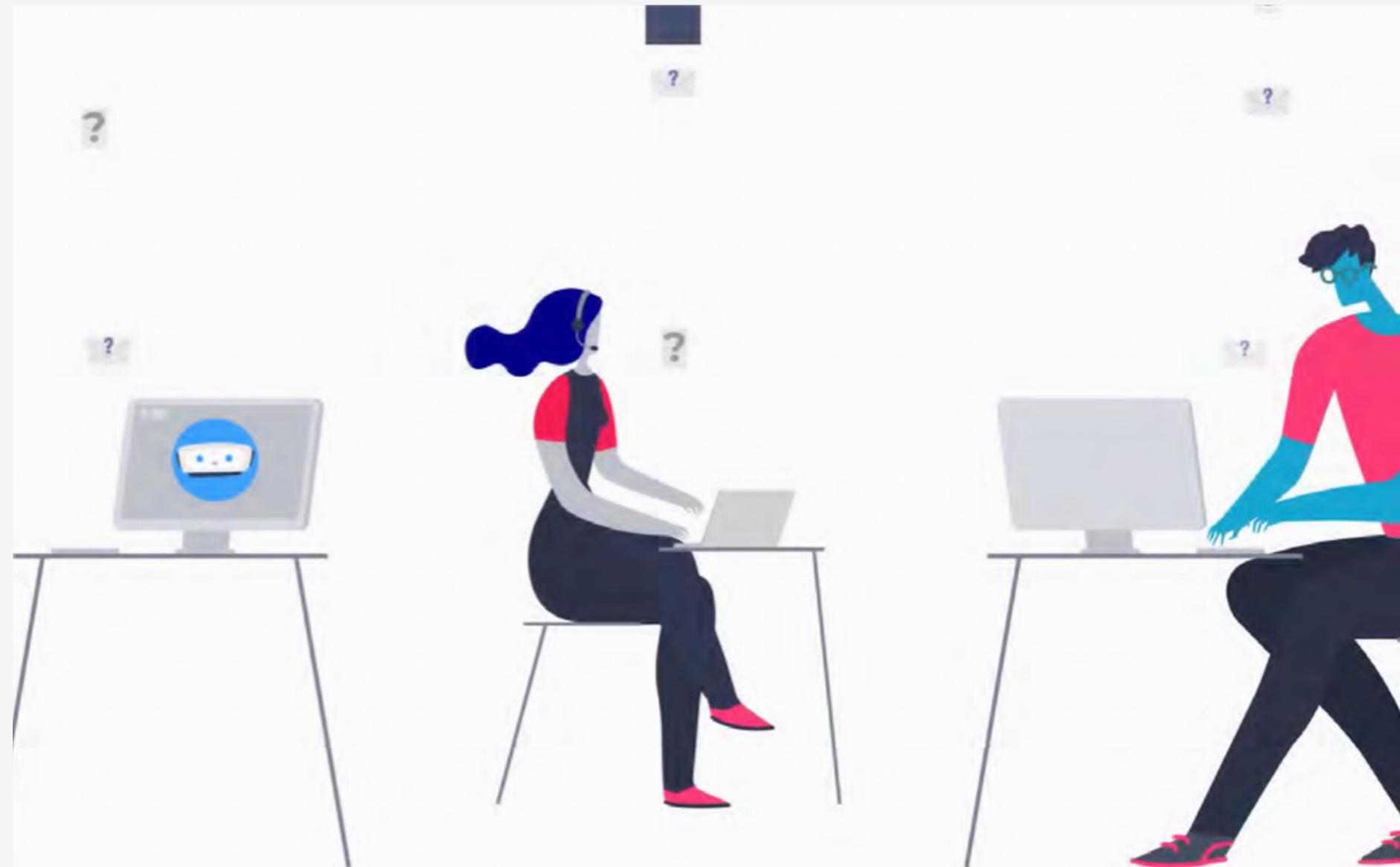
COMBINER

les attributs pour avoir un ensemble de données avec chacun un attribut différent ou un ensemble de données avec les mêmes attributs

FILTRER

les données à partir de leurs attributs ou d'autres attributs à définir.

Foctionnement de RxJS



Pour réagir à des événements ou à des données de manière asynchrone (c'est-à-dire ne pas devoir attendre qu'une tâche, par exemple un appel HTTP, soit terminée avant de passer à la ligne de code suivante), il y a eu plusieurs méthodes depuis quelques années. Il y a le système de callback, par exemple, ou encore les Promise. Avec l'API RxJS, fourni et très intégré dans Angular, la méthode proposée est celle des Observables.

Les Observables et les observers

QU'EST-CE QU'UN OBSERVABLE?

Un Observable est un objet qui émet des informations auxquelles on souhaite réagir. Ces informations peuvent venir d'un champ de texte dans lequel l'utilisateur rentre des données, ou de la progression d'un chargement de fichier, par exemple. Elles peuvent également venir de la communication avec un serveur : le client HTTP, que vous verrez dans un chapitre ultérieur, emploie les Observables.

Observable est l'objet de base de la programmation réactive. C'est lui qui va nous permettre de créer des observables.

QU'EST-CE QU'UN OBSERVER

À un observable, on associe un Observer qui est un bloc de code qui sera exécuté à chaque fois que l'Observable émet une information. L'Observable émet trois types d'information : des données, une erreur, ou un message "complete". Donc, tout Observer peut avoir trois fonctions : une pour réagir à chaque type d'information. Il s'agit en résumé d'une collection de fonctions de rappel ou callback qui savent comment et à quel moment écouter les valeurs qui vont être délivrées à travers un observable. On distingue un subscriber() en plus de l'observer.

Les fonctions de rappel

En JS, une fonction de rappel ou callback est la fonction que l'on va créer et qui va passer en paramètre au travers d'une autre fonction pour pouvoir effectuer un certain nombre d'opérations.

- **NEXT()** : Elle fait appel à une prochaine valeur suivante si celle-ci existe
- **ERROR()** : Elle gère les erreurs qui peuvent paraître pendant que nous appelons une sequence d'observable
- **COMPLETE()** : C'est un signal qui nous exprime que l'ensemble des éléments que nous attendons à travers une sequence est arrivé complètement.

Elle vont nous permettre de créer notre observable et par conséquent notre séquence d'observables.

En pratique

```
next: (item: unknown) => console.log(`Une valeur va être émise ${item}`),
```

→ next prend en paramètre item qui est la valeur qui va être émise à travers l'observable et va le retourner dans notre exemple en console avec un message "une valeur va être émise" et afficher l'item ensuite.

```
error: (err: unknown) => console.log(`Oups il ya une erreur ${err}`),
```

→ error va recevoir un message de type erreur, prend un paramètre appelé err et va le retourner en console avec le message "Oups il y a une erreur"

```
complete: () => console.log('terminé...plus aucunes valeurs')
```

→ complete est la fonction qui ne prend aucun paramètre et va juste nous envoyer un signal pour nous rappeler que les évènements ou la liste de valeur attendue dans notre sequence est complète.

Création d'un Observer

```
ngOnInit() {  
  const observer = {  
    next: (item: unknown) => console.log(`Une valeur va etre emise ${item}`),  
    error: (err: unknown) => console.log(`Oups il ya une erreur ${err}`),  
    complete: () => console.log('terminé...plus aucunes valeurs')  
  };  
}
```

Nous avons créer une constante appelée Observer qui est un objet js contenant nos trois fonctions (next(), error(), complete())

Chacune de ces fonctions va recevoir une notification de différents types.

avec TypeScript dans nos paramètre item ou err, on peut ajouter un type à notre valeur attendue si le on connaît sinon on met any ou unknown

Pour avoir accès aux Observables il faut ajouter deux imports : observable depuis rxjs et OnInit depuis angular/core

```
import { Component, OnInit } from '@angular/core';  
import { Observable } from 'rxjs';
```

On implemente OnInit et on crée l'observable dans ngOnInit

```
export class AppComponent implements OnInit {
```



Séquence d'observables



Une séquence d'observable est un ensemble d'objets ou éléments de différents types (chaine de caractere, ensemble de chaine de caractère, nombres, évènements, ou simplement des objets) composés de différents types de paramètre et de variable, une réponse à une requête http que nous avons envoyée, une combinaison d'autres observables arrivant à différents moments.

Dans notre exemple, c'est un ensemble de nouvelles valeurs qui vont être émises.



A savoir

Techniquement, un observable n'est pas toujours asynchrone, il peut être:

- synchrone : Si dans notre chaîne d'observable nous entrons les valeurs 1, 2 et 3, celles-ci vont arriver toujours dans l'ordre de façon synchrone
- Ou bien asynchrone : Si notre séquence est composée de différents éléments ou différentes sources d'observables alors ceux peuvent terminer à différents moments donnés

Une séquence d'observable peut émettre une certain nombre de valeurs de façon finie dans le cas de 1,2,3 ou de façon infinie si nous avons par exemple une intervalle de seconde entre chaque valeur émise au fil du temps sans jamais s'arrêter jusqu'à ce qu'on le fasse nous même.

Création d'une séquence d'observables

Pour créer un cas concret simple, nous allons créer un Observable dans appcomponent qui va va emmettre 3 valeurs (Valeur 1, Valeur 2, Valeur 3) avec chacune la methode next() et ensuite complete() pour mettre fin à notre sequence d'observable.

```
const stream = new Observable(myObserver => {
    myObserver.next('Valeur 1');
    myObserver.next('Valeur 2');
    myObserver.next('Valeur 3');
    myObserver.complete();
```

Nous avons appeler le constructeur d'observable avec RxJS newObservable, qui prend en paramètre myObserver et à l'intérieur nous avons créer notre séquence (stream).

Démarrer une séquence d'observables

Maintenant que nous avons créer notre séquence d'observable, il faut obligatoirement la démarrer.

Pour ce faire, nous allons appeler la méthode subscribe() sinon il nous sera impossible de travailler avec.

Il suffit d'appeler notre séquence stream: stream.subscribe() et à l'intérieur mettre l'observable créé .

Cela suffit pour démarrer notre séquence d'observable.

Cette méthode de création de séquence permet de bien comprendre comment s'effectue une creation de séquence d'observable mais il y a une méthode moins longue.

Autre méthode de création de séquence d'observables

A l'intérieur du subscribe, appeler item et l'affecter à un console.log. Copier/coller console.log et afficher item à l'intérieur pour éviter de créer à chaque fois un objet. Ensuite afficher err, complete sans avoir à mettre de paramètre. Cela évite de créer séparément un objet de type observer.

```
const subscription = stream.subscribe(  
  item => console.log(`Une valeur va être émise ${item}`),  
  err => console.log(`Oups il ya une erreur ${err}`),  
  () => console.log('terminé...plus rien')  
);
```

On peut donc sauvegarder tout ceci dans une souscription. Ce pendant, il arrive qu'une séquence d'observable soit composée d'éléments finis ou finis. Il faut donc se détacher de la souscription.

Comment arrêter une séquence d'observables?

Complete()

Cette méthode arrête automatiquement notre séquence

Error()

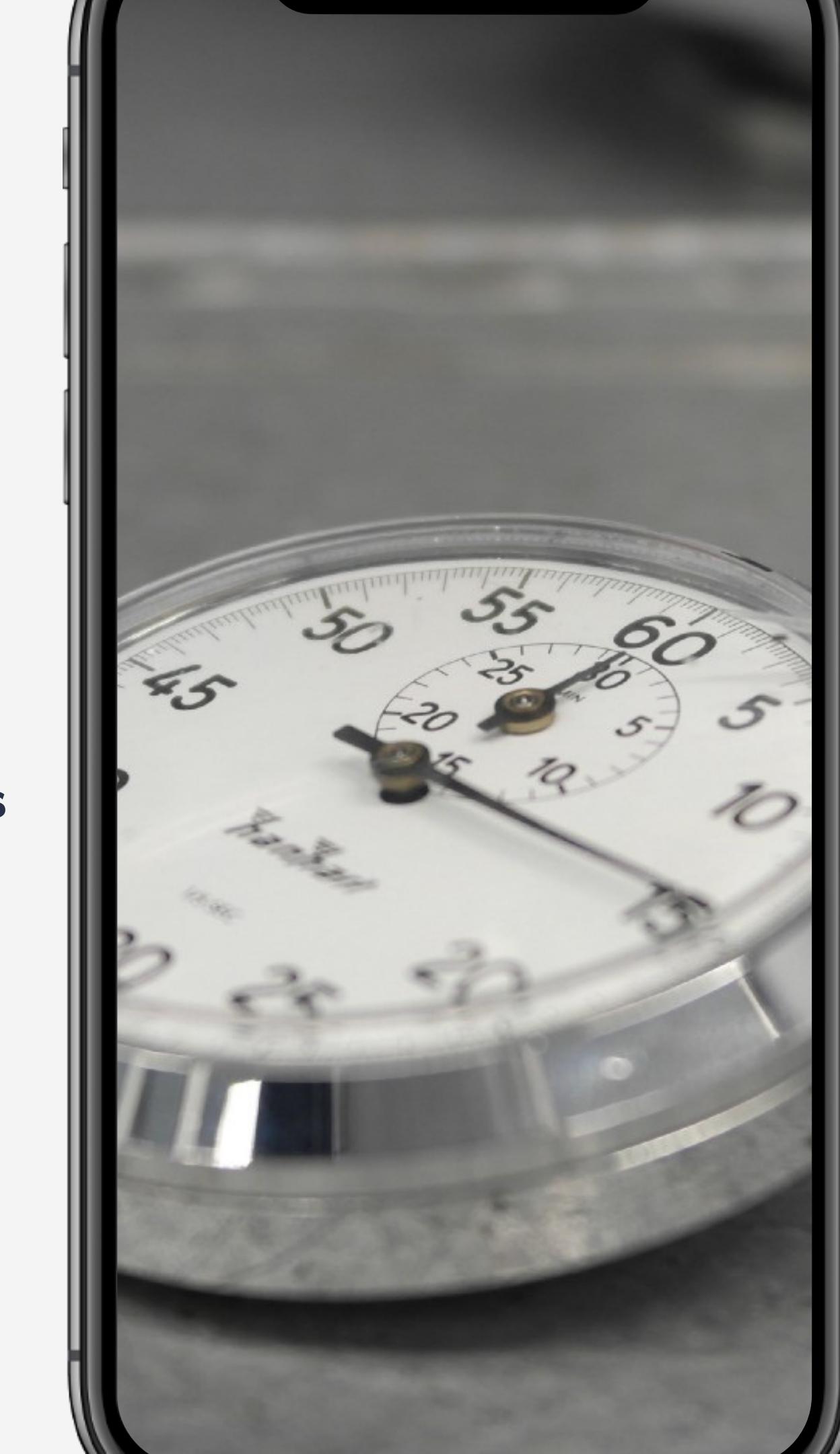
Elle peut arrêter automatiquement notre séquence à cause d'une erreur

Fonctions de création de séquence d'observables

of(), from() : ces fonctions appellent toujours la complete() si on les utilise pour créer notre séquence.

Les opérateurs

Take(), take until()



Comment arrêter une séquence d'observables?

La méthode `Unsubscribe()`

Cette méthode est la plus connue pour arrêter une séquence d'observables. Elle permet en effet de se désabonner d'une séquence laquelle on a été abonnée.

Pour pouvoir l'utiliser nous pouvons directement appeler le nom de l'observable et ensuite appeler `unsubscribe()`

Dans notre exemple nous aurons alors
`myObservable.unsubscribe.`



En pratique

```
myObserver.next('Valeur 1');
myObserver.error(new Error());//error() arrete une sequence d'observable à cause d'une erreur
myObserver.next('Valeur 2');
myObserver.next('Valeur 3');
myObserver.complete();//complete() permet de stopper automatiquement notre sequence d'observable quand elle est terminée
);
```

```
const subscription = stream.subscribe(
  item => console.log(`Une valeur va etre emise ${item}`),
  err => console.log(`Oups il ya une erreur ${err}`),
  () => console.log('terminé...plus rien')
);
subscription.unsubscribe(); //appel de la methode unsubscribe qui nous permet de nous detacher de la chaine
// et de pouvoir stopper notre sequence d'observable
```

Lorsqu'on fait un subscribe, cette souscription n'est pas stockée dans une variable : on ne peut donc plus y toucher une fois qu'elle est lancée, et cela peut nous causer des bugs.

En effet, une souscription à un Observable qui continue à l'infini continuera à recevoir les données, que l'on s'en serve ou non, et nous pouvons en subir des comportements inattendus. Afin d'éviter tout problème, quand on utilise des Observables personnalisés, il est vivement conseillé de stocker la souscription dans un objet Subscription que l'on importe depuis RxJS

```
ngOnDestroy() {
  this.subscription.unsubscribe();
}
```

Le code fonctionne de la même manière qu'avant, mais il est recommandé pour un composant Angular ajouter le code qui évitera les bugs liés aux Observables. On va se servir du niveau de vie OnDestroy avec sa fonction ngOnDestroy() et pour cela il faut l'implémenter comme le OnInit pour la souscription. Ainsi toutes les souscriptions sont détruites et on évitera ainsi une fuite de mémoire ou memoryleacks

Création d'observables avec les fonctions of() et from()

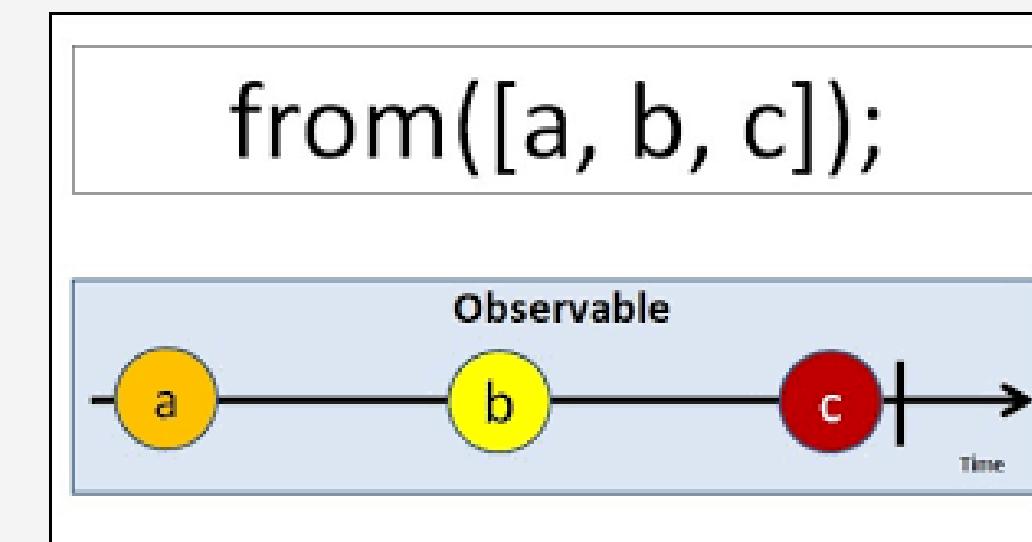
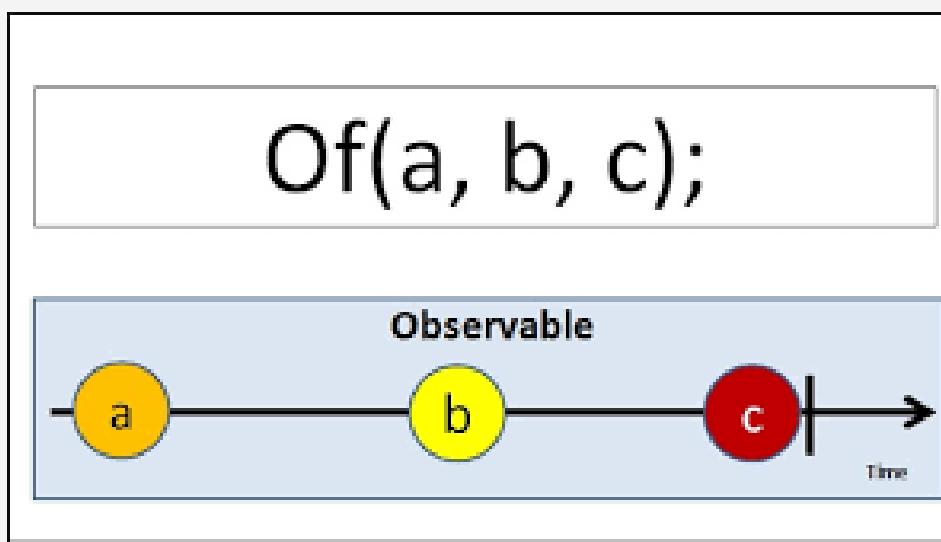


Création d'observable grace aux fonctions of() et from()

RxJS met à notre disposition des fonctions de création d'observable pour créer une séquence d'observables. Parmi celles-ci nous avons: Intervalle, fromEvent et of(), from() qui sont les plus utilisées.

Ces deux fonctions appellent automatiquement la méthode complete() lorsqu'on passe les valeurs et nous n'avons pas besoin d'appeler le constructeur NewObservable à chaque fois, d'appeler complete() et unsubscribe() pour se détacher de notre chaîne d'observable.

Ces fonctions sont à importer depuis RxJS.



EN PRATIQUE

```
of(1,2,3,4).subscribe(console.log)
```

of retourne l'arrêt complete() tel que nous l'avions passé initialement en paramètre.

```
from([12,13,14,15]).subscribe(  
  (item: number) => console.log(`ma valeur ${item}`),  
  (err: any) => console.error(err),  
  () => console.log('Termine')  
)
```

Avec from() nous avons besoin d'une structure de données, dans ce cas nous allons prendre plusieurs éléments 12,13,14,15 et ajouter subscribe comme avec of() ou utiliser notre méthode précédente en définissant le type de la valeur que l'on souhaite avoir .



Les opérateurs

Un opérateur est une fonction qui se place entre l'Observable et l'Observer (la Subscription, par exemple), et qui peut filtrer et/ou modifier les données reçues avant même qu'elles n'arrivent à la Subscription. C'est une fonction qui est utilisée pour manipuler les éléments d'une séquence d'observable.

Nous allons voir quelques exemples.





RXJS OPERATORS

MAP()

modifie les valeurs reçues et peut effectuer des calculs sur des chiffres, transformer du texte, créer des objets...

FILTER()

comme son nom l'indique, filtre les valeurs reçues selon la fonction qu'on lui passe en argument.

THROTTLETIME()

impose un délai minimum entre deux valeurs — par exemple, si un Observable émet cinq valeurs par seconde, mais ce sont uniquement les valeurs reçues toutes les secondes qui vous intéressent, nous pouvons passer `throttleTime(1000)` comme opérateur.

SCAN() ET REDUCE()

permettent d'exécuter une fonction qui réunit l'ensemble des valeurs reçues selon une fonction qu'on lui passe — par exemple, on peut faire la somme de toutes les valeurs reçues. La différence basique entre les deux opérateurs : `reduce()` nous retourne uniquement la valeur finale, alors que `scan()` retourne chaque étape du calcul.



LE .PIPE : LA CLE POUR SE SERVIR DES OPERATEURS

Fonctionnement

Si on crée par exemple notre séquence d'observable avec `of(1, 2, 3)` nous allons utiliser `.pipe` pour pouvoir nous servir des opérateurs. On peut alors introduire de nouveaux opérateurs à l'intérieur de la séquence d'observable et chaque élément de la séquence d'observable va passer à travers chaque opérateur que l'on va inscrire dans le pipe.

Illustration:

```
of(1, 2, 3 )
```

```
    .pipe(
```

```
        map(elem=>elem*2),
```

-- `map()` va transformer l'élément reçu (`elem`) de notre séquence d'observable et le multiplier par 2

```
        tap(el=>console.log),
```

-- `tap()` permet de récupérer l'élément qui a été transformé

```
        take(2)
```

-- `take()` permet d'effectuer une logique prédéfinie avec RxJS

```
        ).subscribe(console.log)
```

-- on ajoute `.subscribe` pour terminer la séquence avec `.pipe` et `console.log` pour pouvoir récupérer les transformations qui ont été effectuées dans le pipe

Le .pipe

Donc en pratique, à travers l'implémentation d'un simple opérateur, il va s'agir pour cet opérateur de récupérer l'élément de l'observable, de souscrire à cet élément et d'effectuer des transformations de façon interne ensuite de retourner un autre nouvel observable qui va être réutilisé dans la séquence suivante ou alors dans l'opérateur suivant .

Sources

- https://www.youtube.com/channel/UC1Snl7_ciBy9A5jAHUdRm0g
- <https://openclassrooms.com/fr/courses/4668271-developpez-des-applications-web-avec-angular/5089331-observez-les-donnees-avec-rxjs>
- <https://rxjs-dev.firebaseio.com/guide/overview>
- sources Images: https://images.indepth.dev/images/2020/05/RxJS-in-Angular_-When-To-Subscribe_.jpg
- https://miro.medium.com/max/1007/1*hj71wy_fVD1qIG8q9mnPjg.png
- <https://s3.amazonaws.com/coursetro/posts/150-full.png>
- https://miro.medium.com/max/1400/1*HeOwc9fteR4oiVheE7qnAg.png
- [https://1.bp.blogspot.com/-uEhhOTI7d10/YKPCwqU0HzI/AAAAAAAAG4/3wU98nsLdbQqzNwYqmPHwKQoHa4Pym1uACNcBGAsYHQ/s1920/operators%2bin%2Bangular.jpg](https://1.bp.blogspot.com/-uEhhOTI7d10/YKPCwqU0HzI/AAAAAAAAG4/3wU98nsLdbQqzNwYqmPHwKQoHa4Pym1uACNcBGAsYHQ/s1920/operators%2Bin%2Bangular.jpg)
- <https://www.tektutorialshub.com/wp-content/uploads/2020/01/Observable-Of-Operator.jpg>
- canva.com