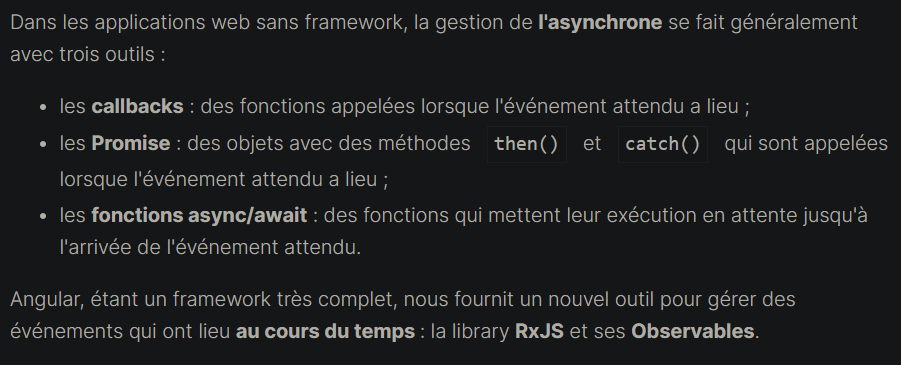
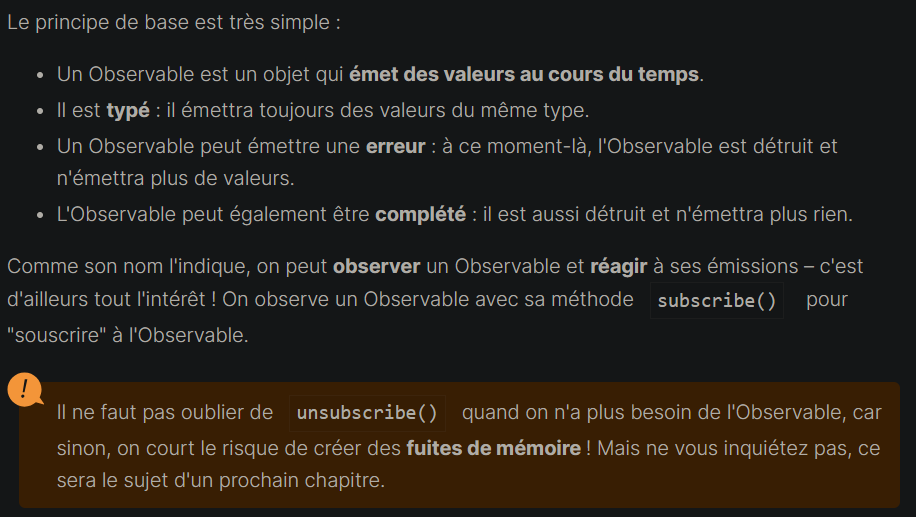
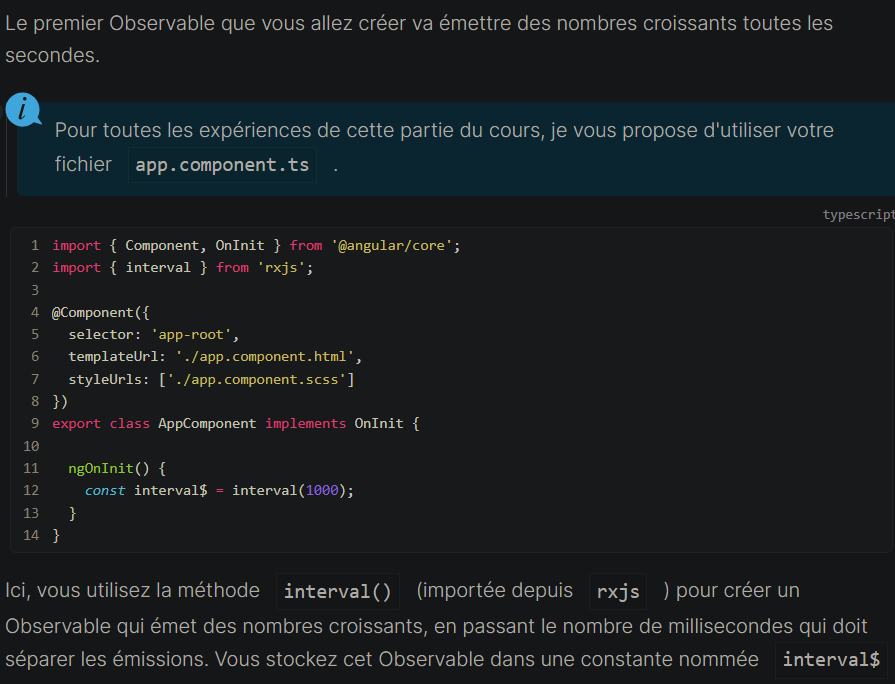
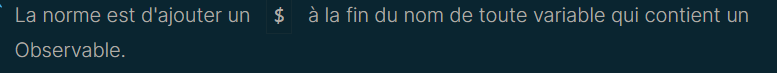
**Les observables RxJS :**

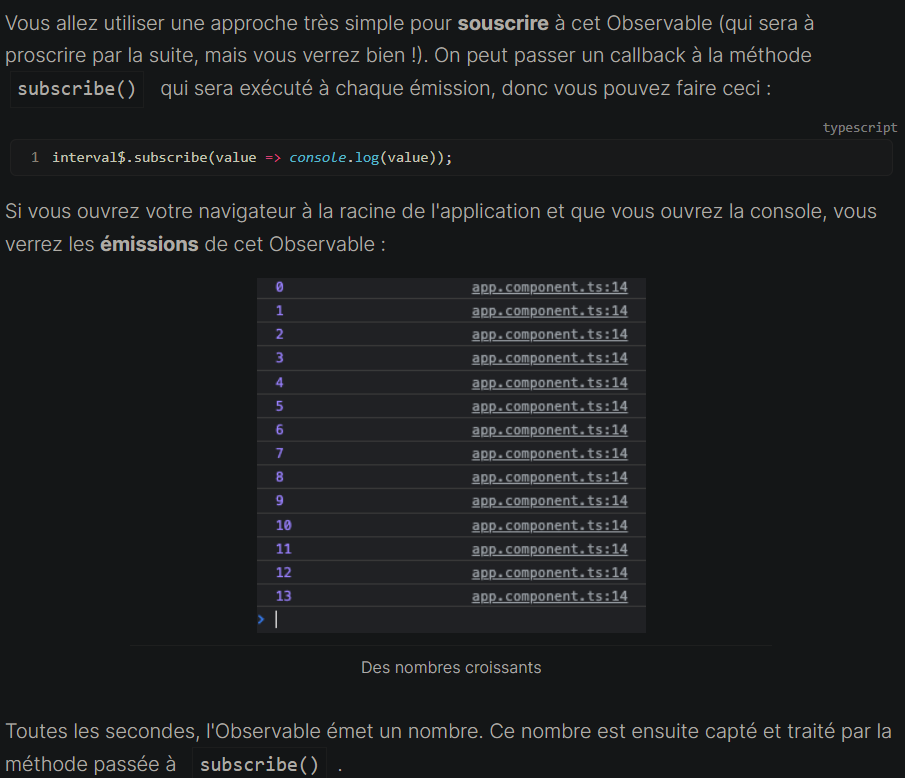
****

****

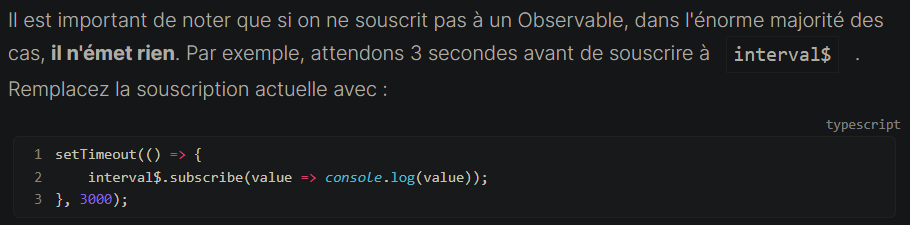
Create an observable

****

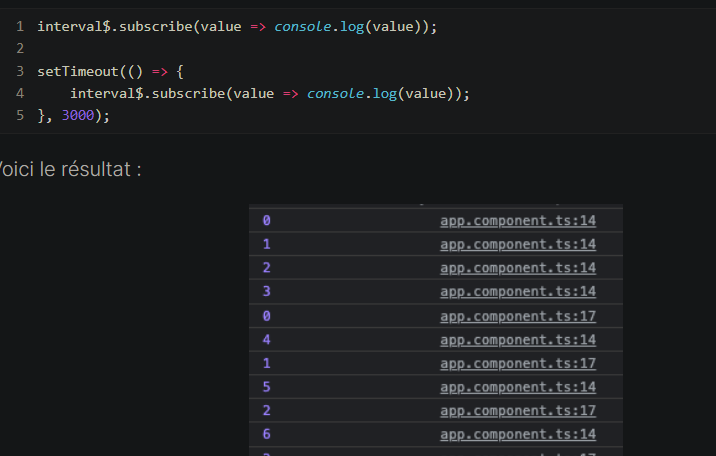
****

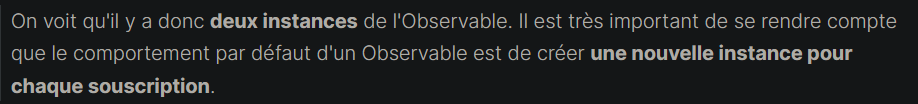
****

If we don’t subscribe to an observable, il n’émet rien !

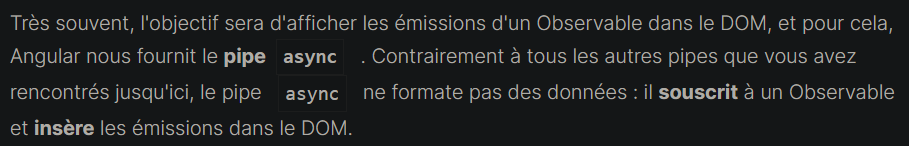
****

Si on souscrit 2 fois à un observable, on crée 2 instances de l’objet qui sont indépendants !

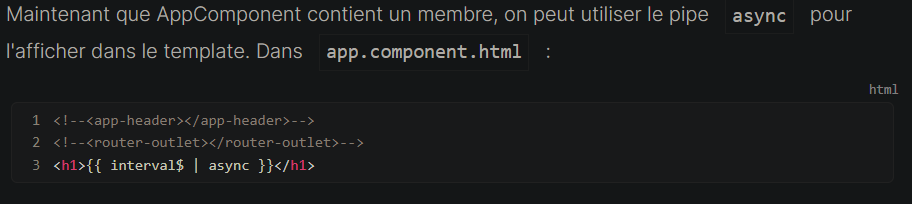
****

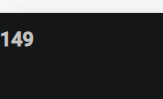
****

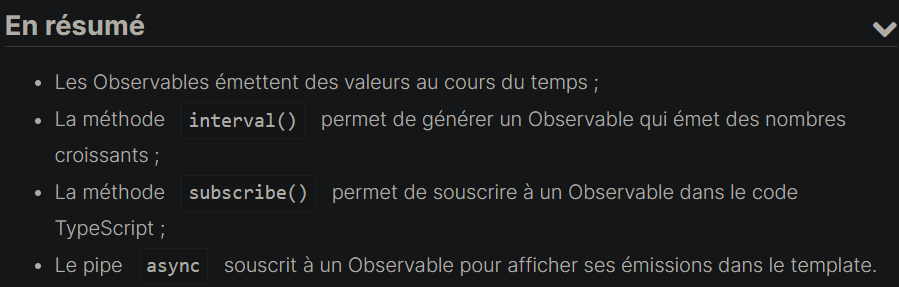
**Pipe async =>** permet d’inscrire à un observable et émettre ses valeurs dans le dom.

****

****

****

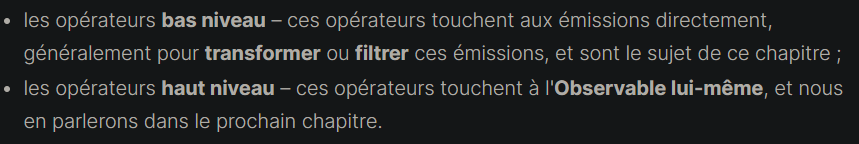
**Au bout de 149s.**

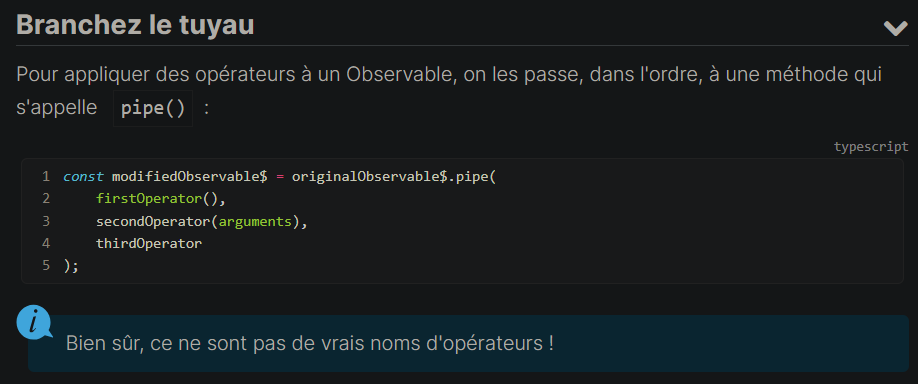
****

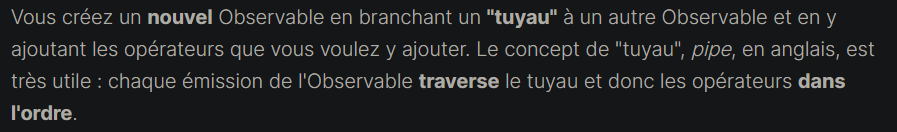
Les observables occupent une place très importante dans le framework d’Angular, et nous permettent de créer des fonctionnalités très avancées et complexes beaucoup plus facilement.

Il est donc essentiel de maîtriser la manipulation de ces observables. Cette manipulation se fait avec les opérateurs.

Il existe deux types principaux d’opérateurs :



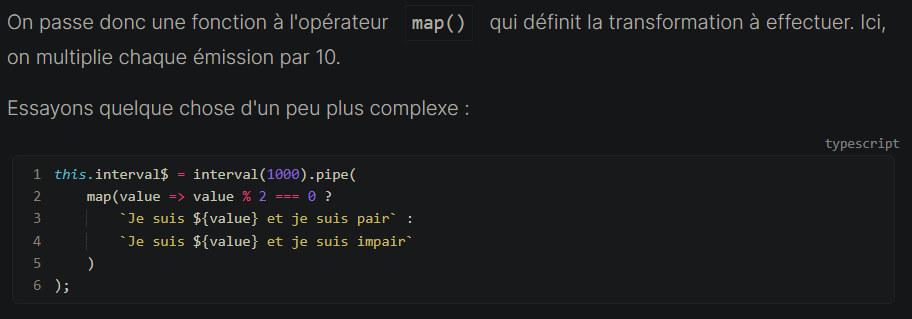




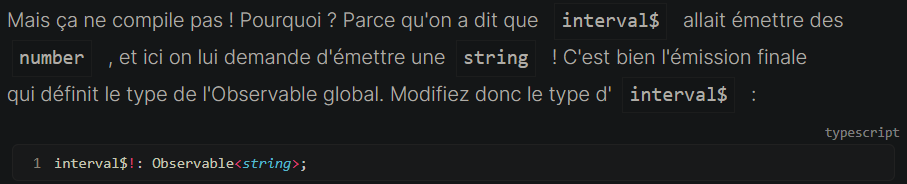
Obsevable.pipe(

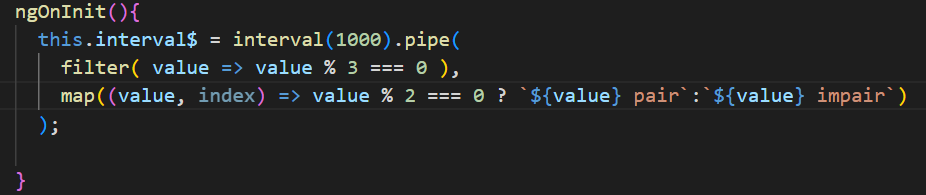
Map(element => element\*5),

) ;



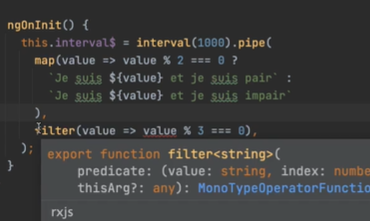
Là on a changé le type de retour en string, il faut donc changer le type de l’observable.





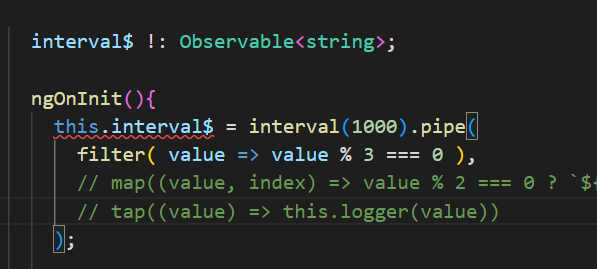
On peut ajouter un 2eme opérateur : filter.

L’ordre est très important, si on met le filter après map :

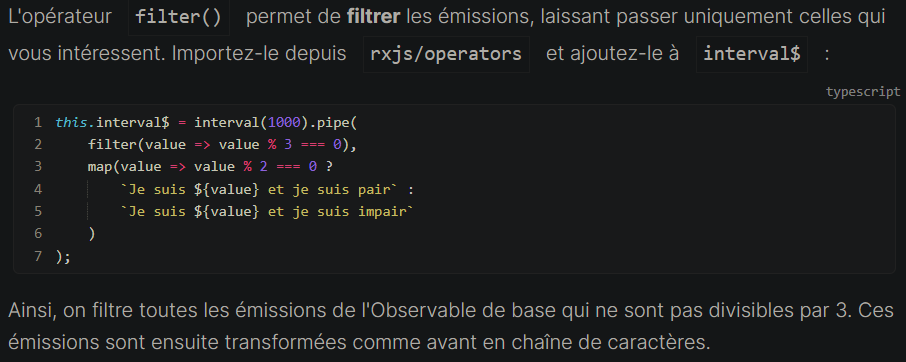


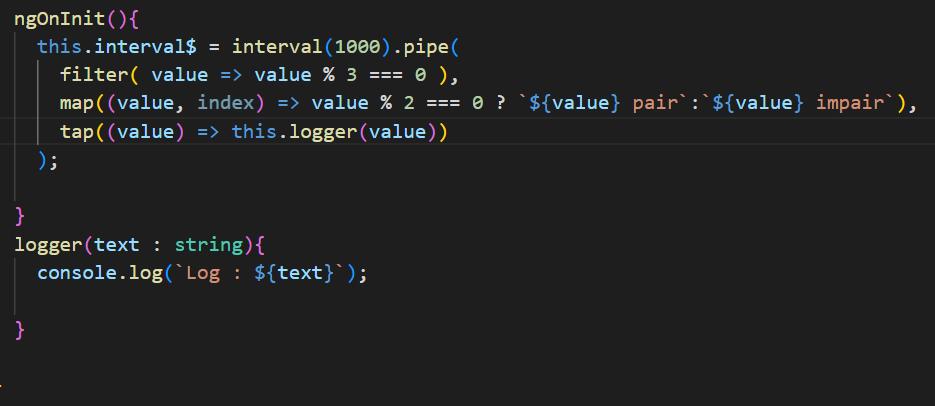
Ça ne marche pas car le type de retour de map est string, et string divisé par number est impossible !

Il faut bien comprendre que observable<string> veut dire que le renvoie final est string, l’observable peut contenir des numbers mais à la fin il doit à tout coup return des strings



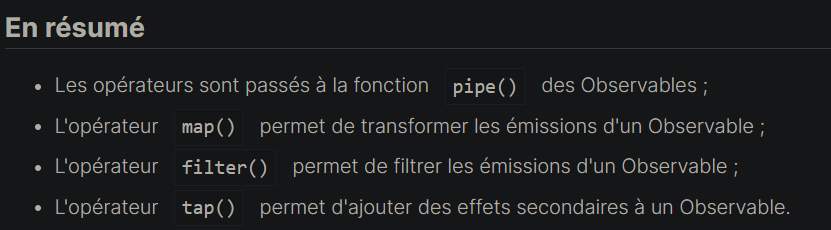
Si on enlève le renvoie final qui retourne des strings, l’intervalle contient des numbers et non pas des strings !





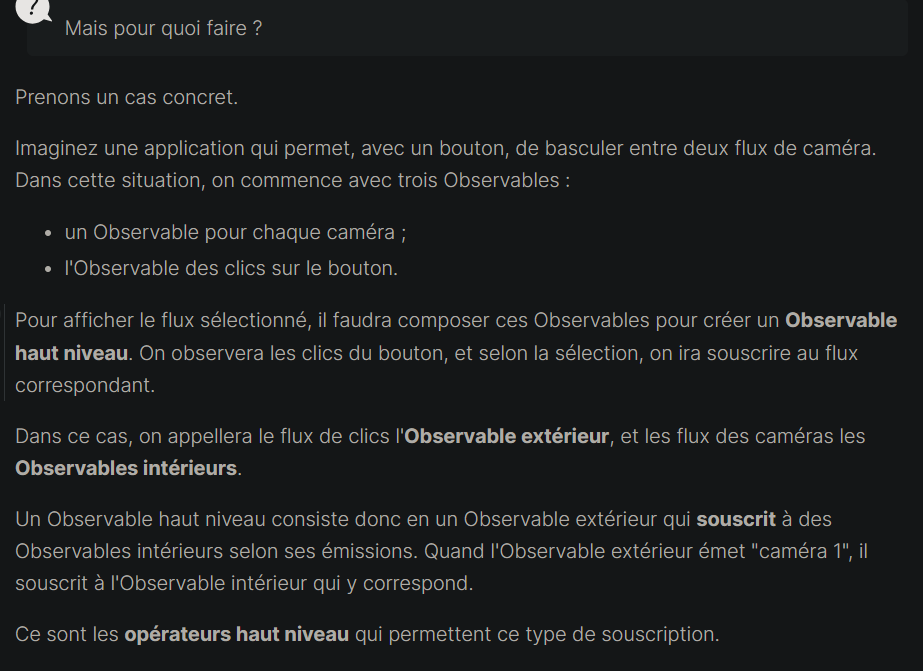
L’opérateur tap permet de réagir aux différentes émissions de l’observable sans toucher à l’observable lui-même.

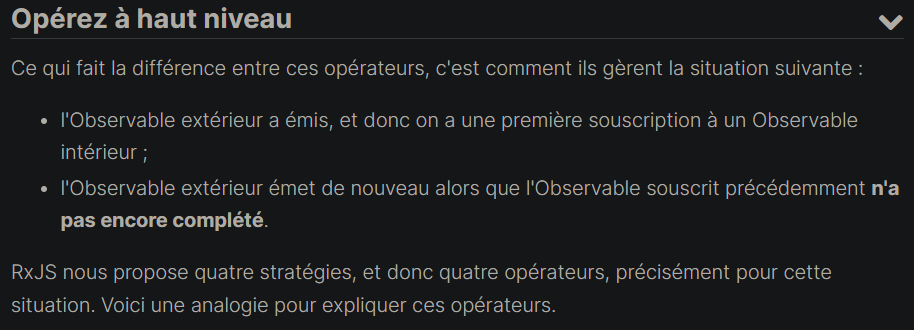


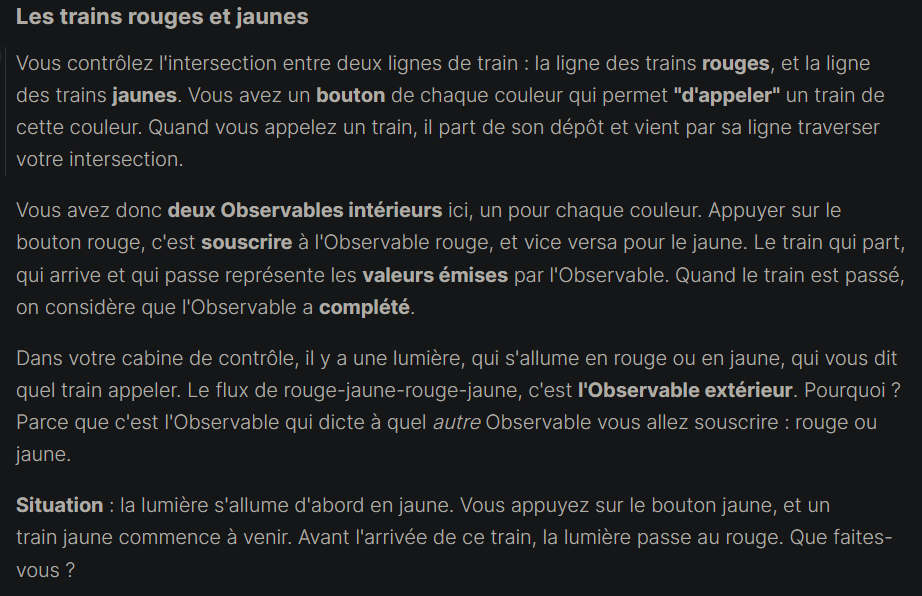


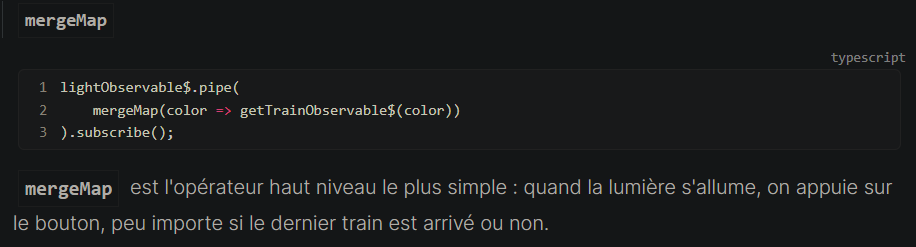
Observable de haut niveau !

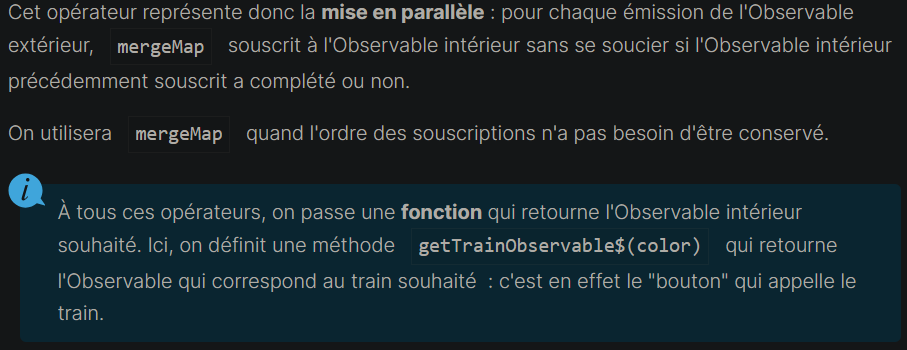
C’est un observable qui souscrit à d’autres observables !

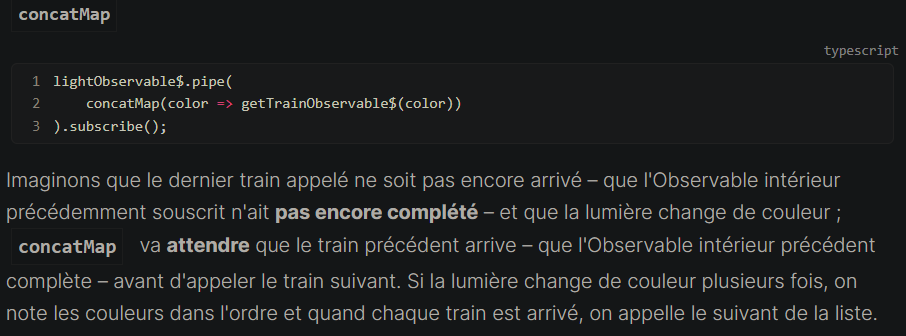


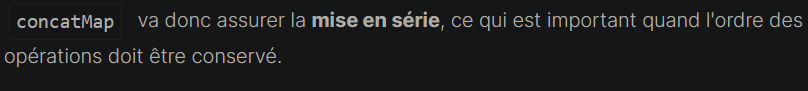


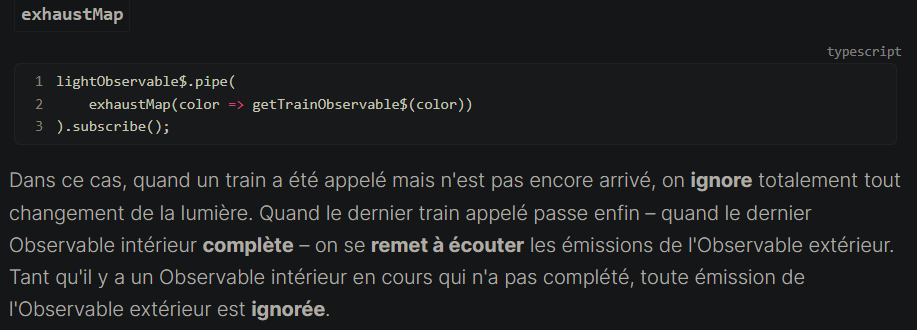


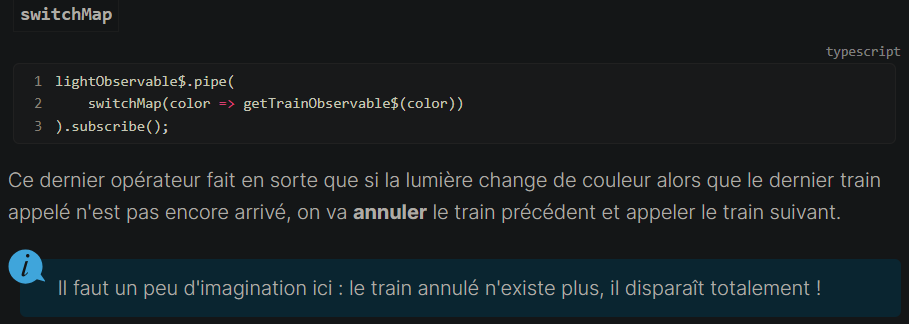
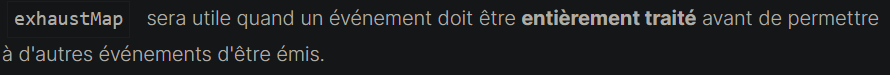


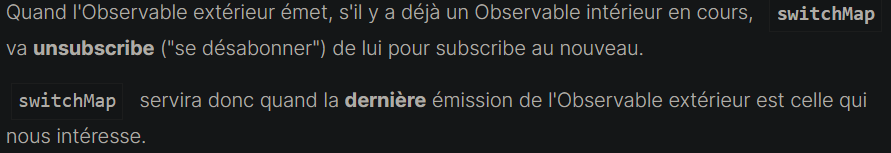


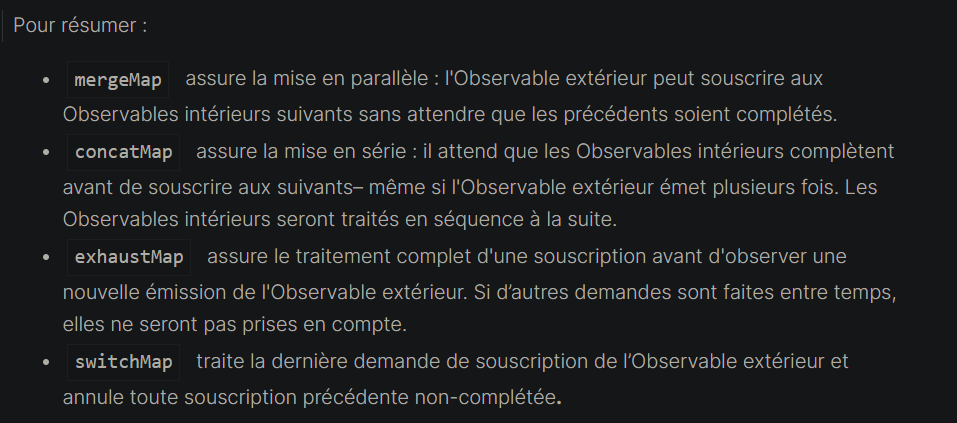


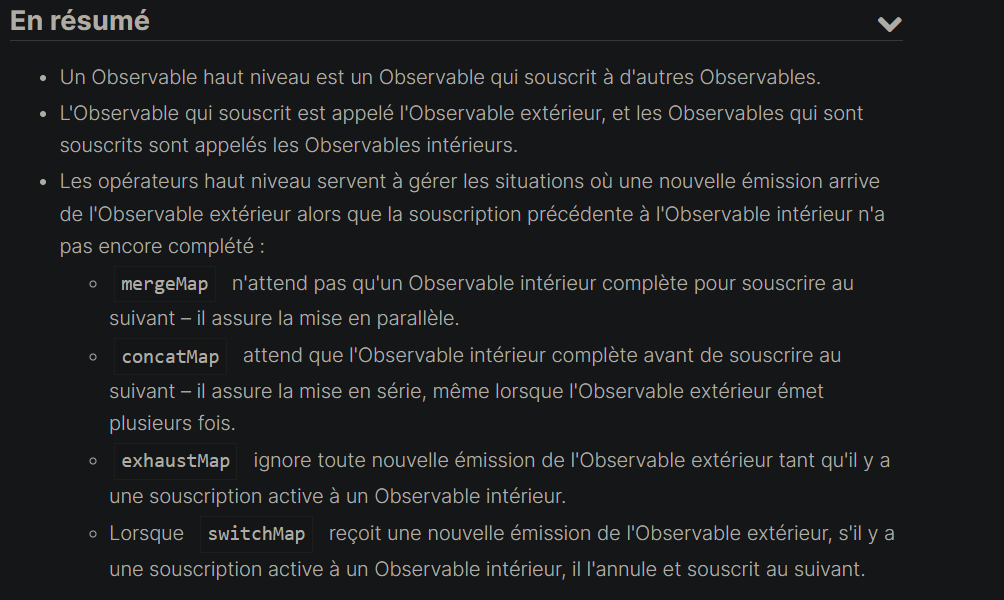




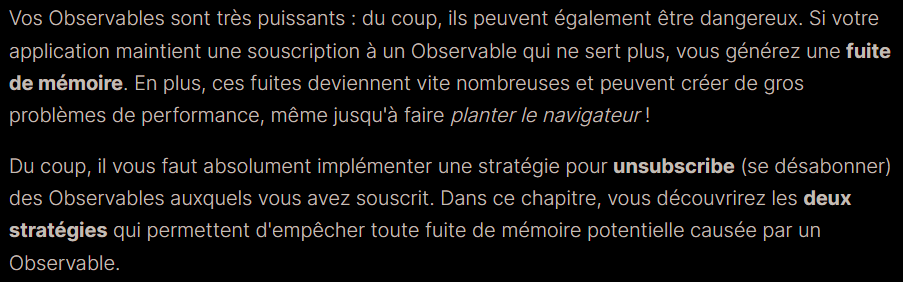


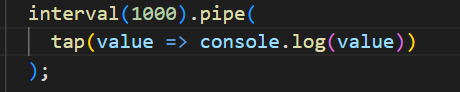




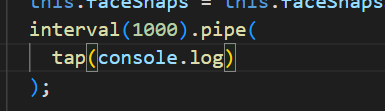


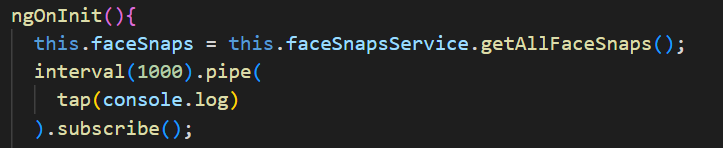
**Memory leak**

****

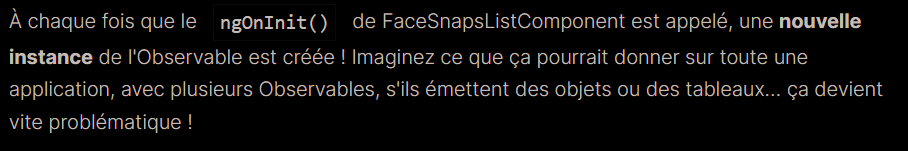
****

**Or we can do :**

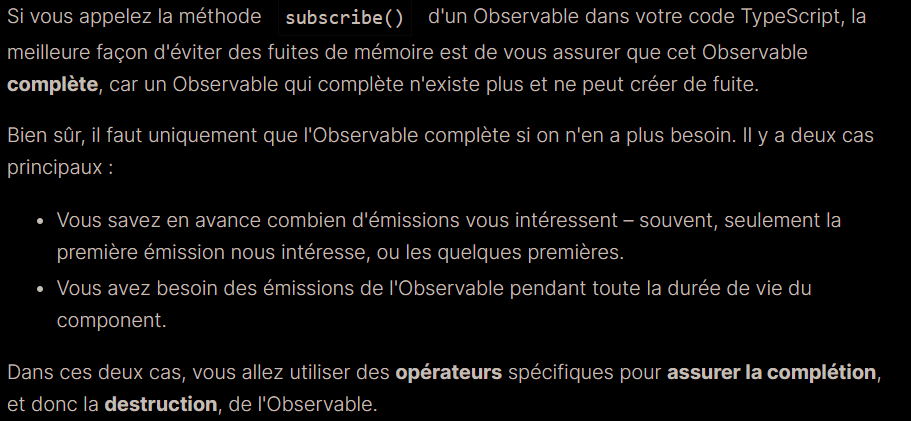
****



ngOnInit is called every time we call the component, so it will create an observable and subscribe to it.

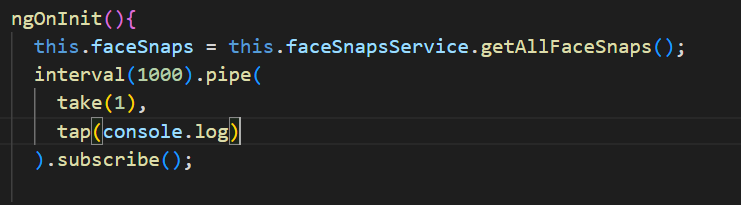


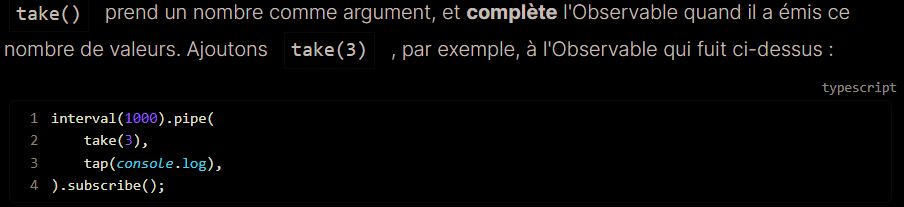
To prevent this, unsubscribe :



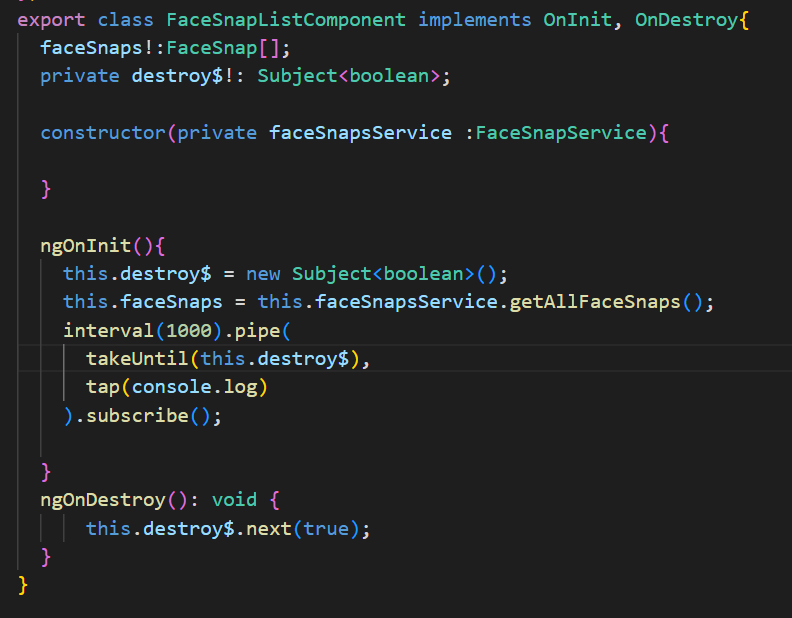
Once an observable complete all its emission, it gets destroyed !

If we know how many emission we need from the observable we can specify it by using **operator take(numberOfEmission) ,**



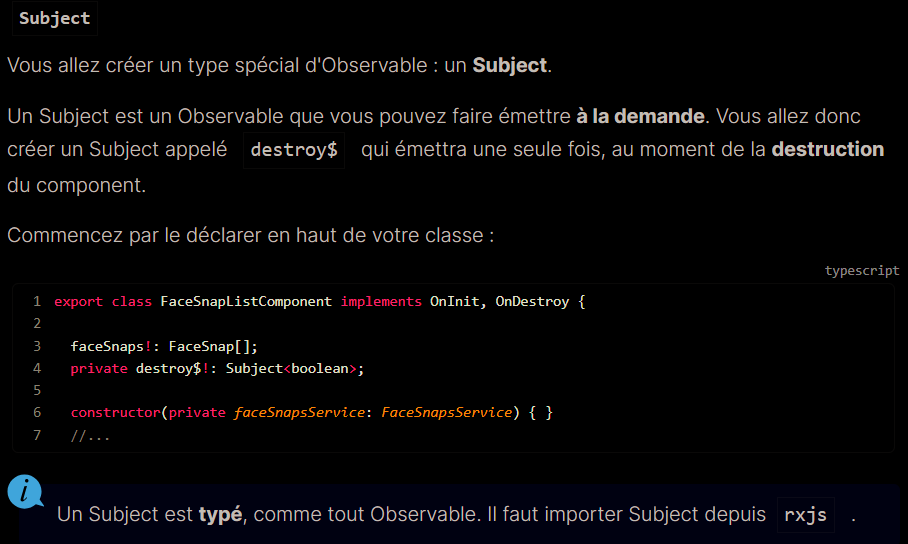
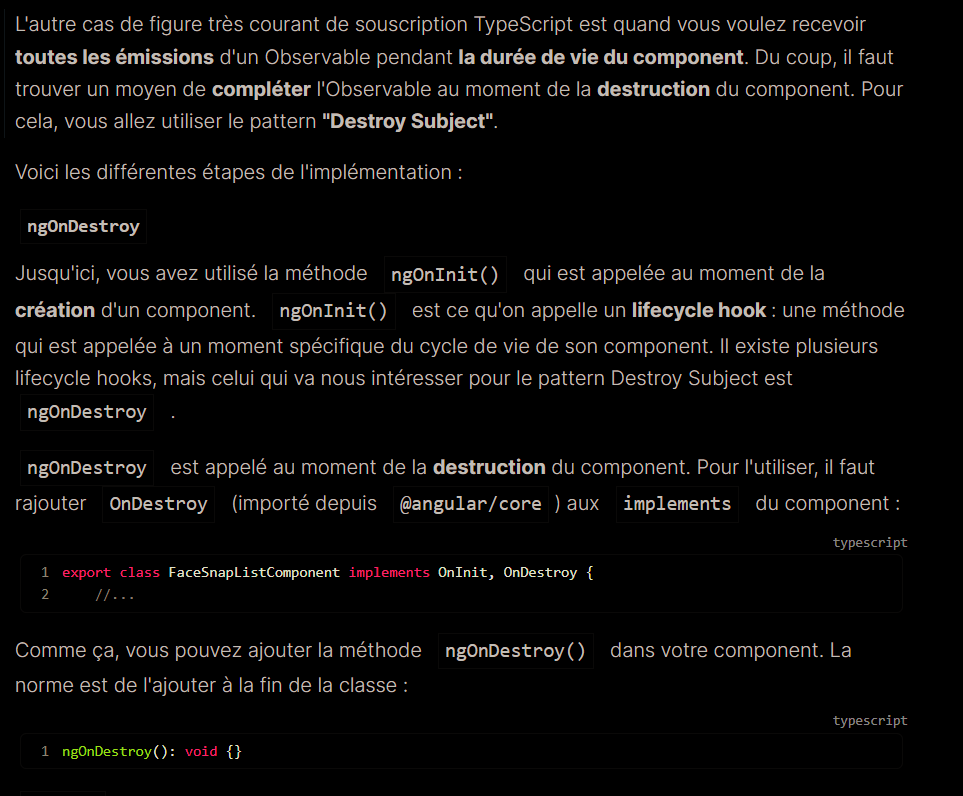


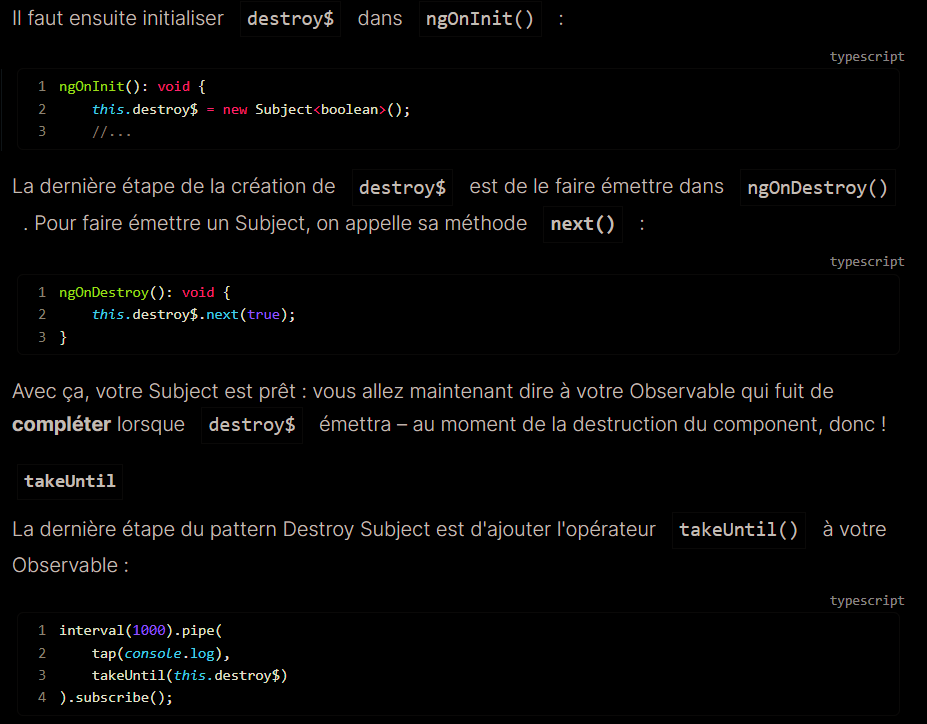
If we need the observable to keep emitting as long as the component is rendered, and once we destroy the component, the observable should get destroyed too !



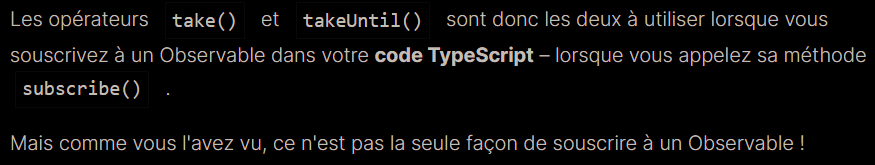
A subjet is an observable that we can control, we create a subject, and on destroy the subject created emets true.

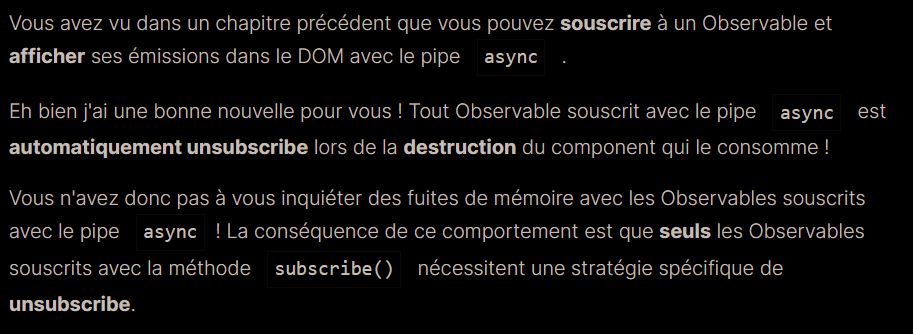
In our observable we use takeUntil, which means that i twill emets until the destroy subject emets which will happen onDestroy of the component !



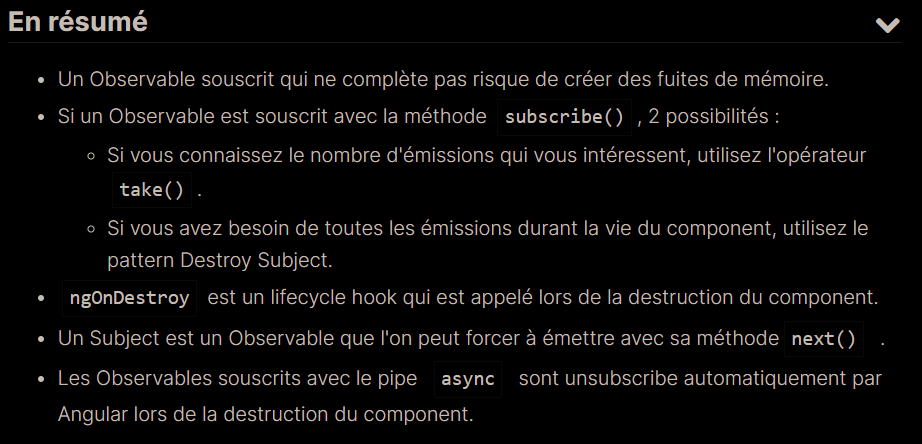


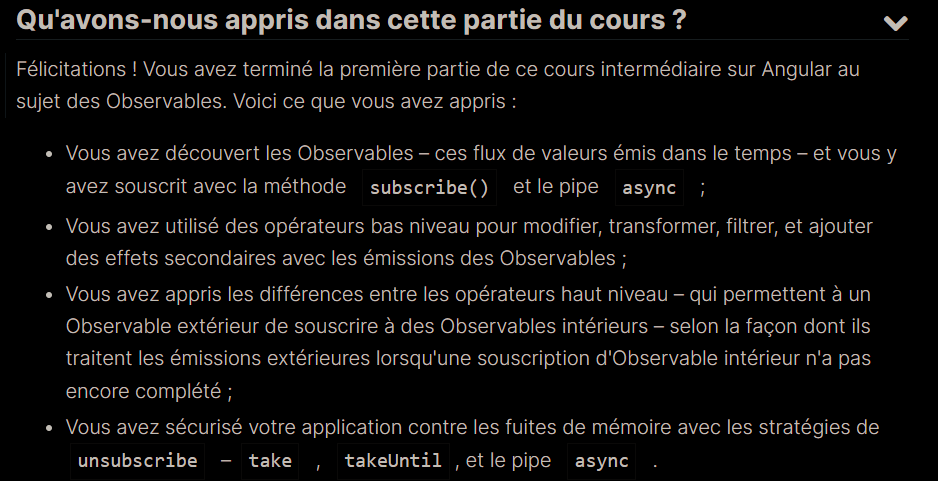






Using the pipe async to subscribe to an observable assure its completion therefore destruction once the component is destroyed !

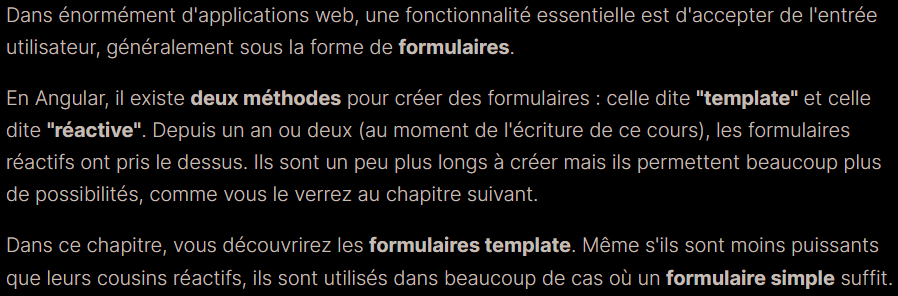


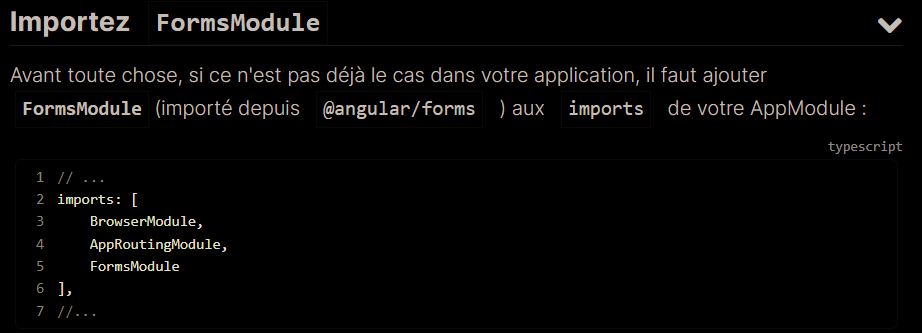


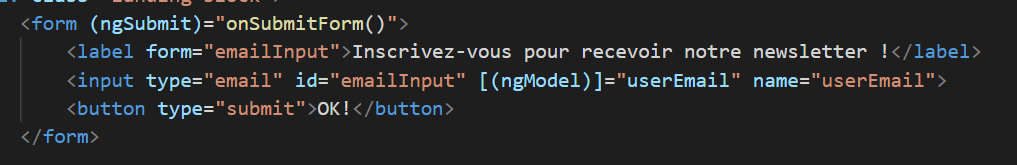
<https://www.learnrxjs.io/>

**Formulaires :**

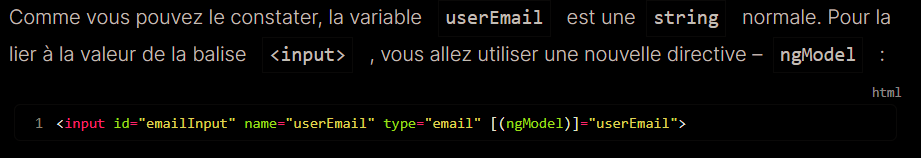
**Formulaire template**

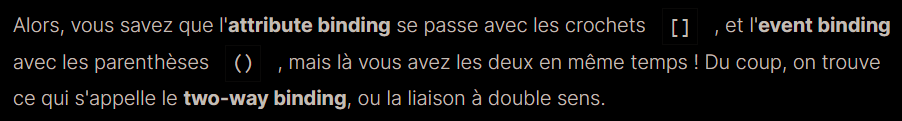




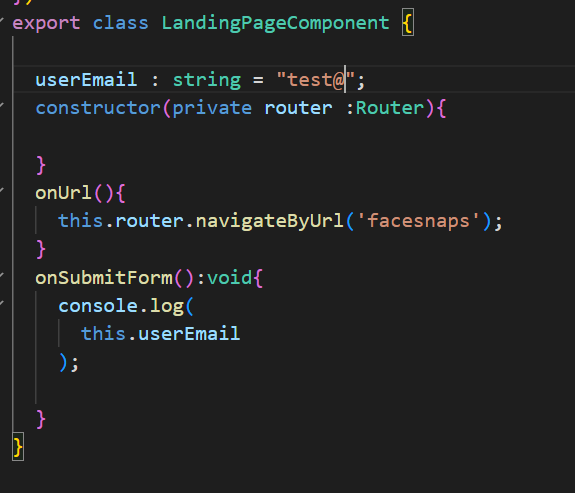


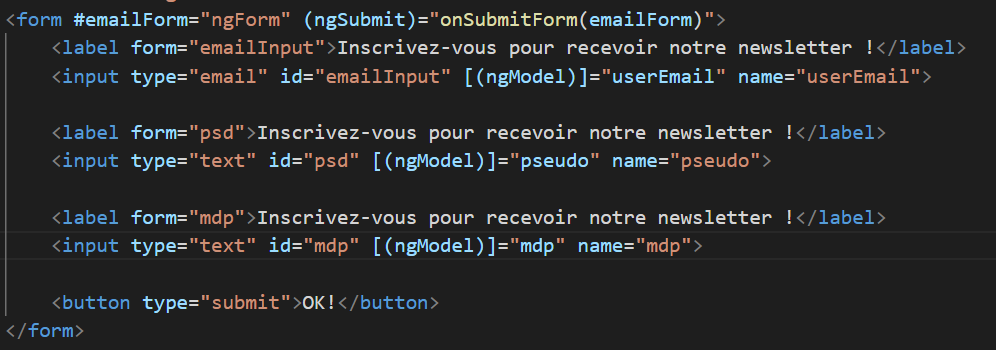
Two way binding [(ngModel)] = ‘’variable’’.

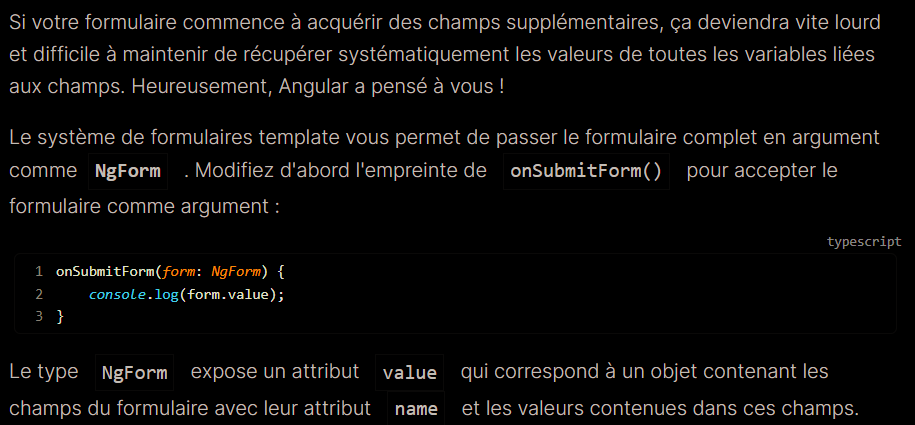


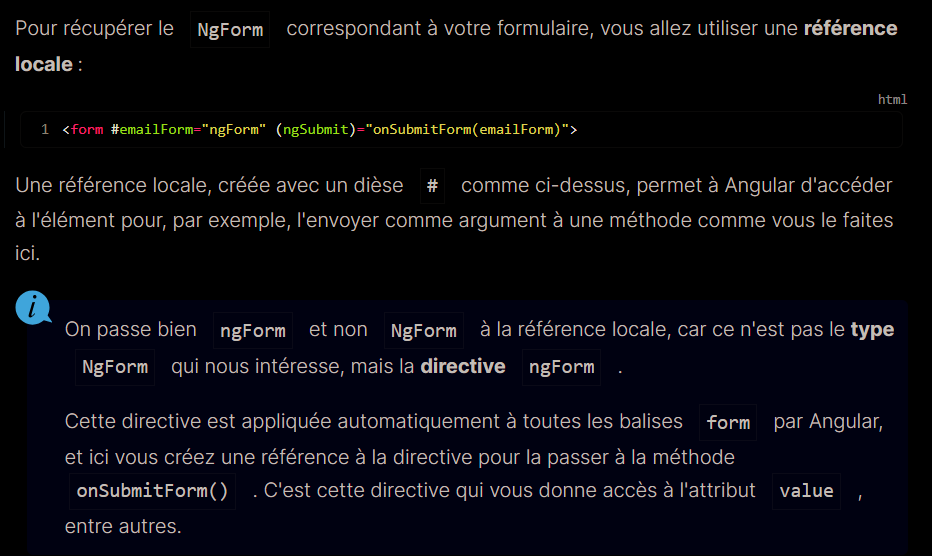


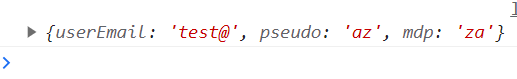










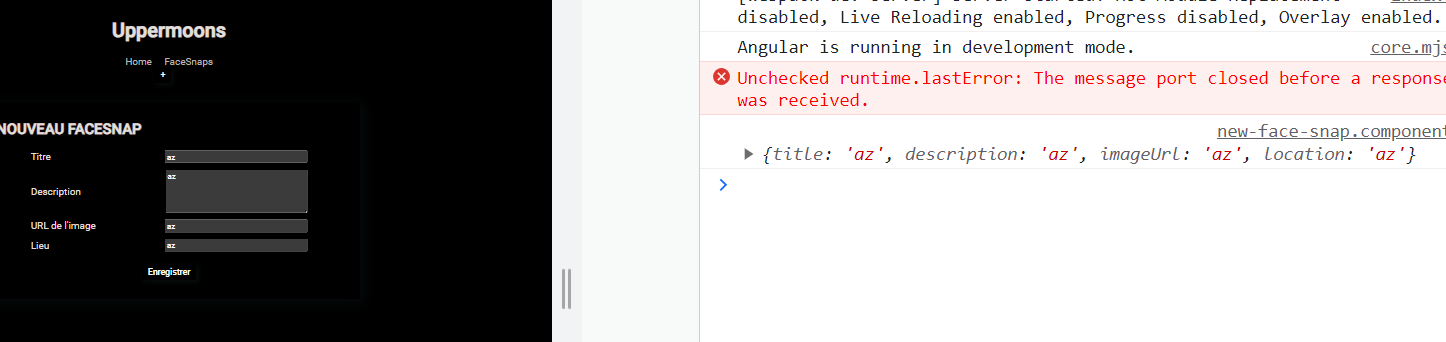


Formulaire réactive.

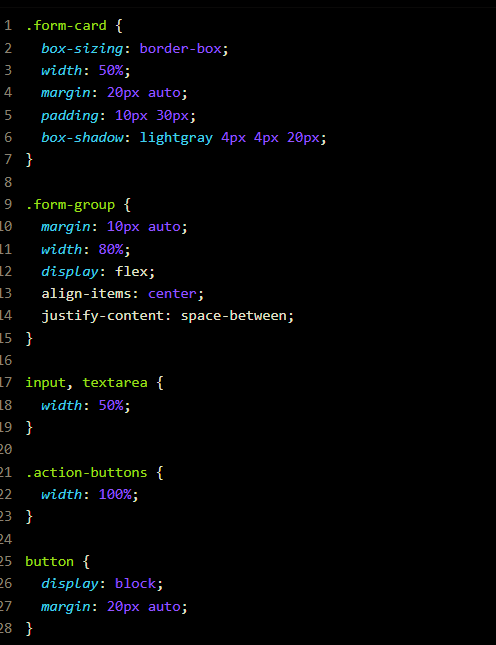


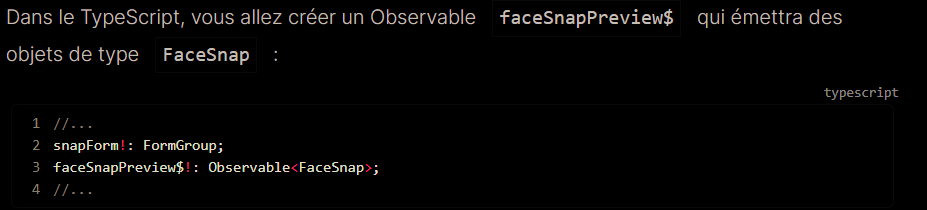




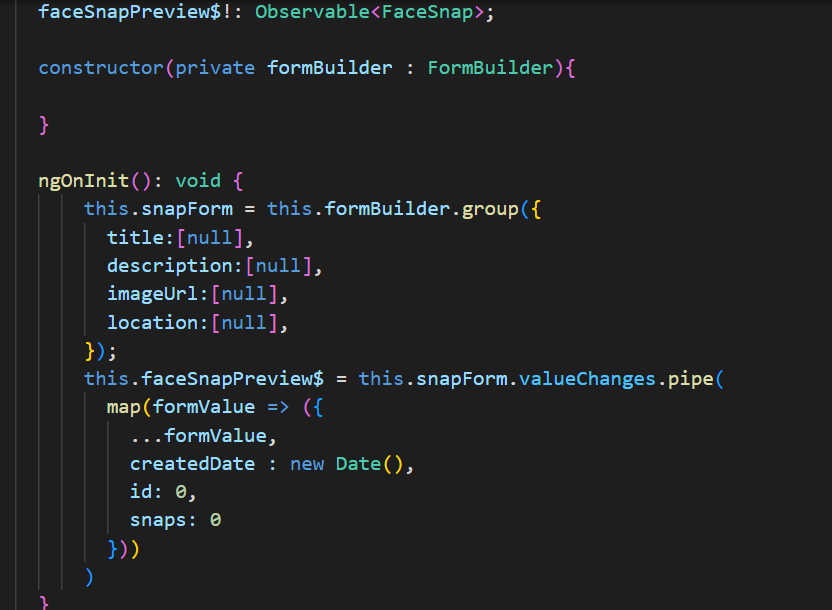


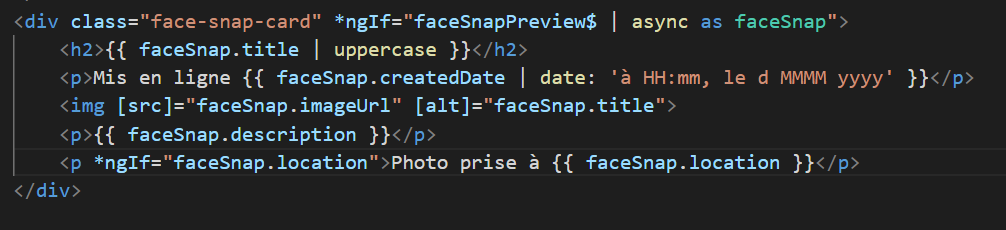


Style : 



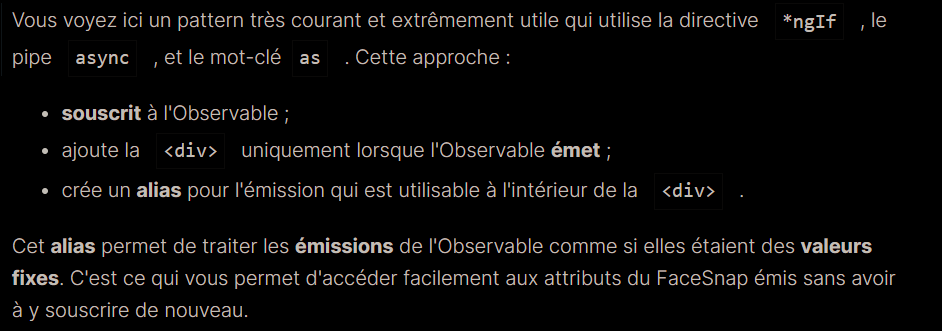
****

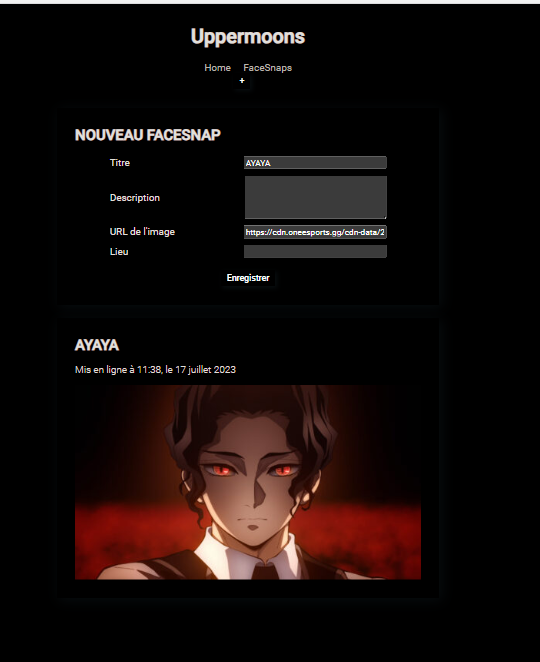
****

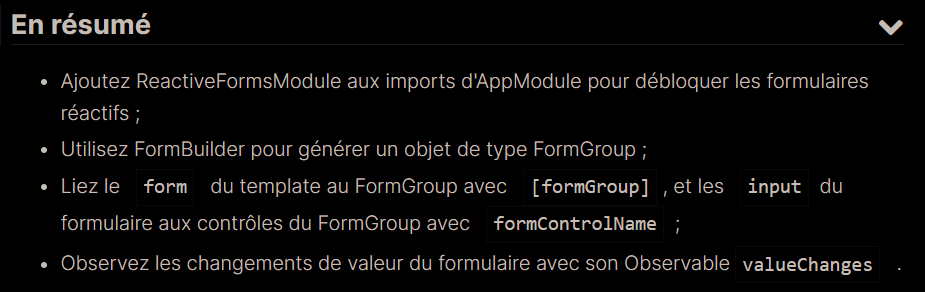
****

**Pipe async as faceSnap allows us to take the value of the observable and treats it as a fixed value !**

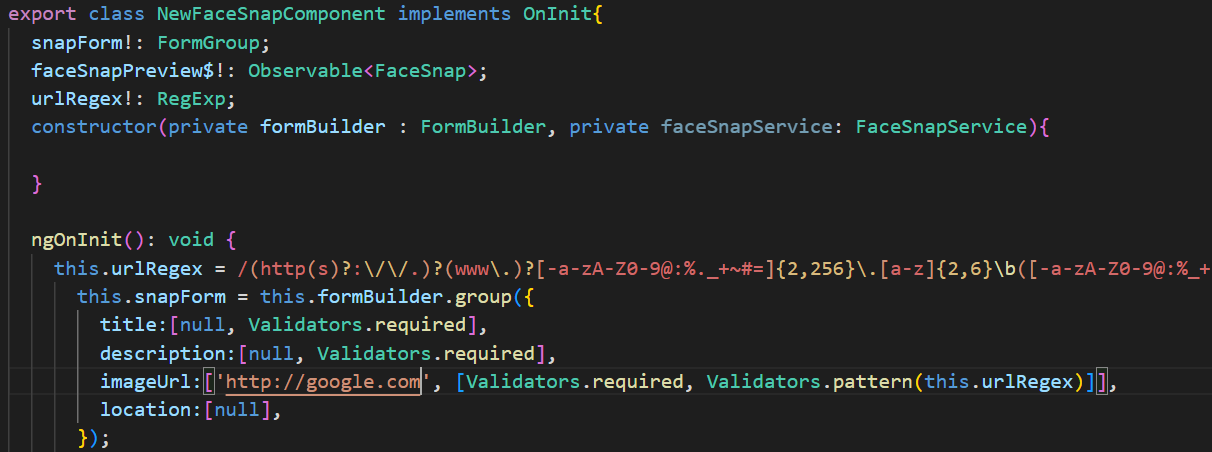
****

****

****

****

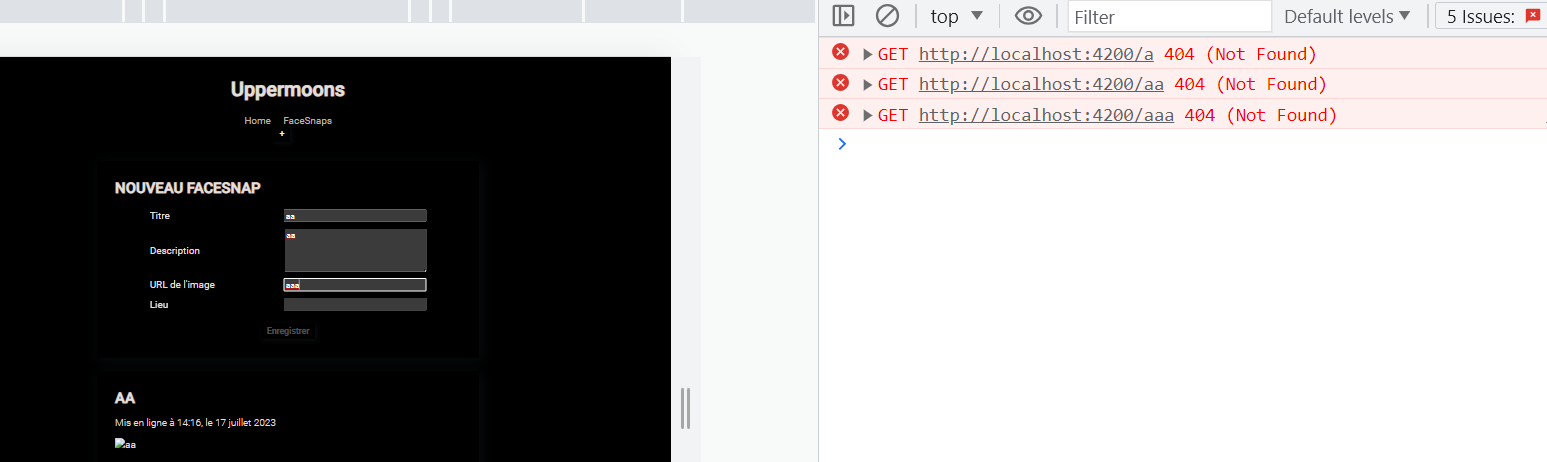
**Validators :**

****

when we instanciate, we pass the validators.

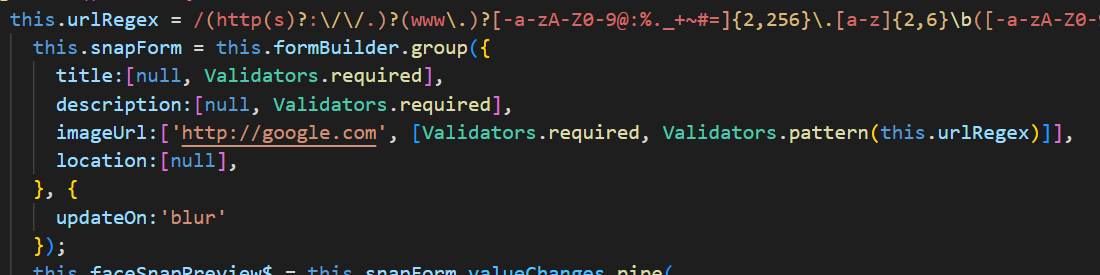
We can disable a button as long as the validators are not met :

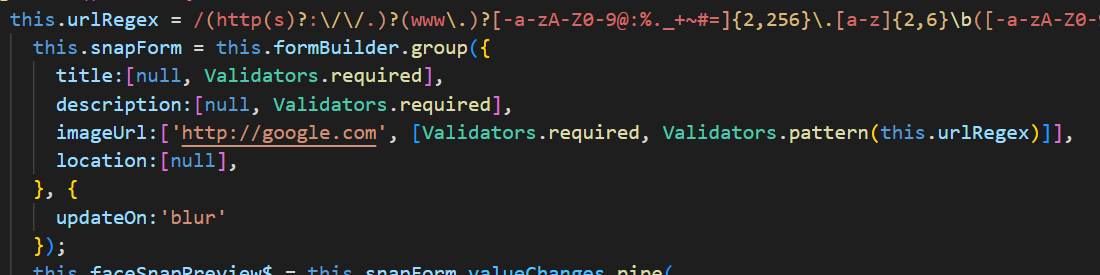


****

When completing the url field, we get error, cuz it s trying to get the image even tho the url is not finished yet !!

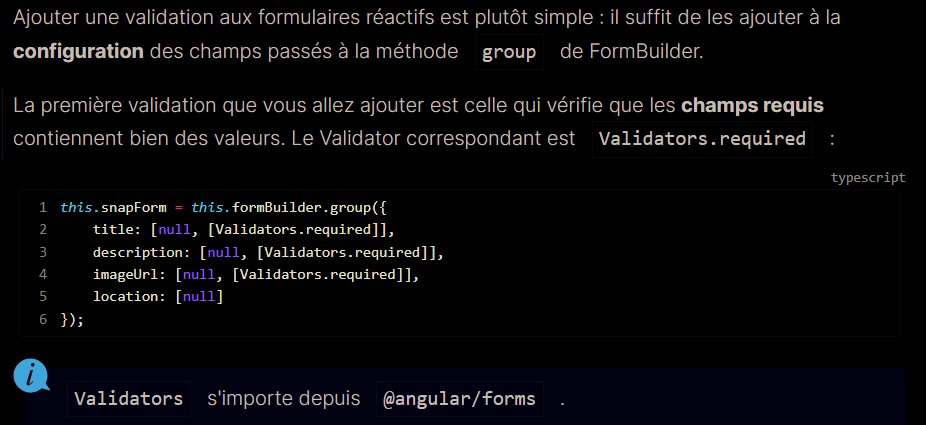
To correct this we can make it so that the observable emets a value only if we change a field, therefore after we finish completing it .

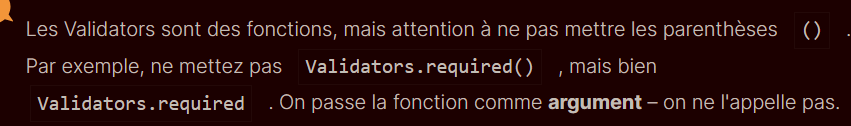


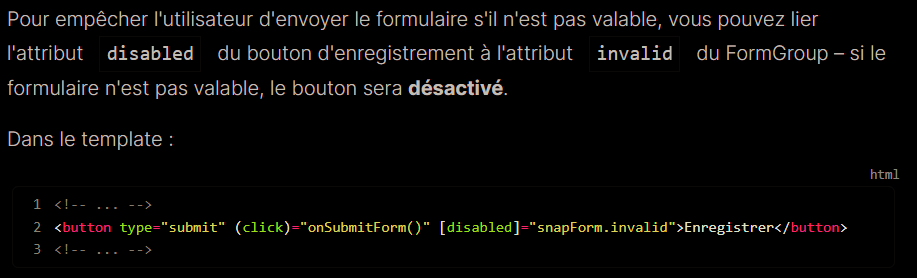


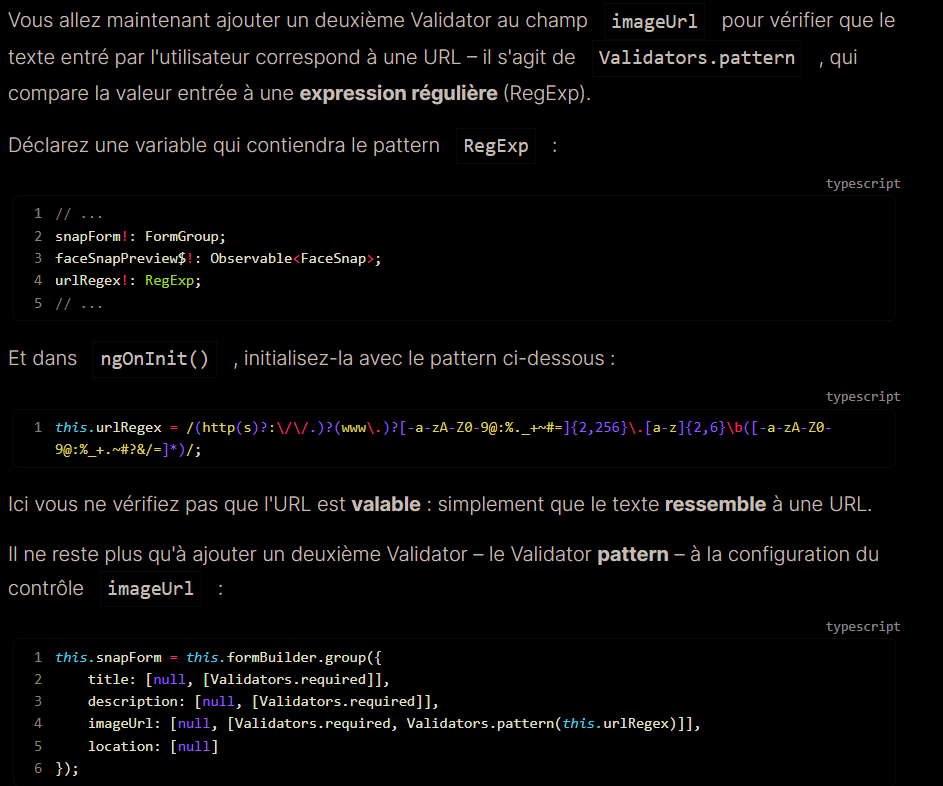
Le formulaire n’est mis à jour que quand on change de champs !

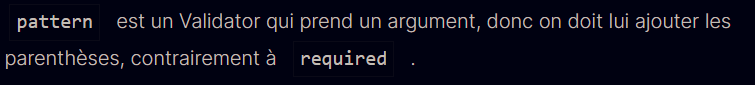
updateOn :’blur’





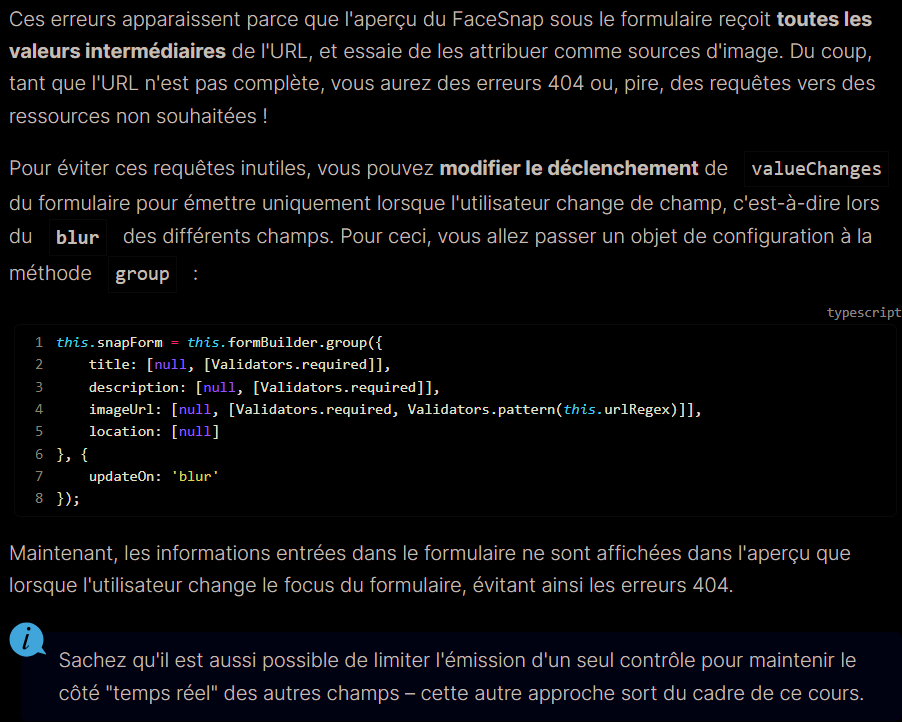






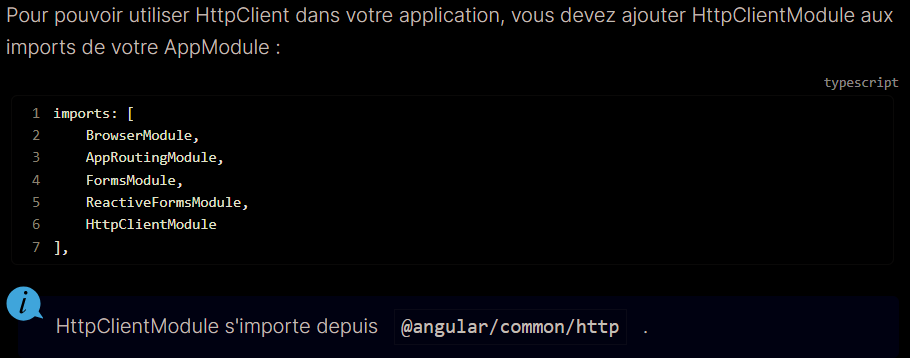


Changez le rythme des émissions !

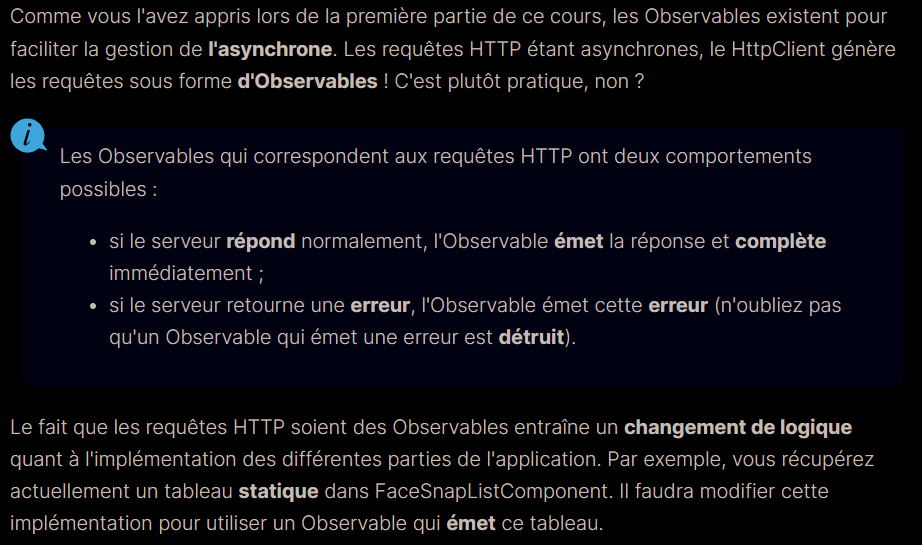


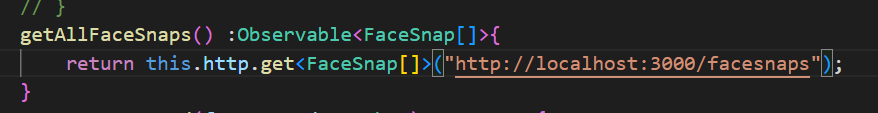


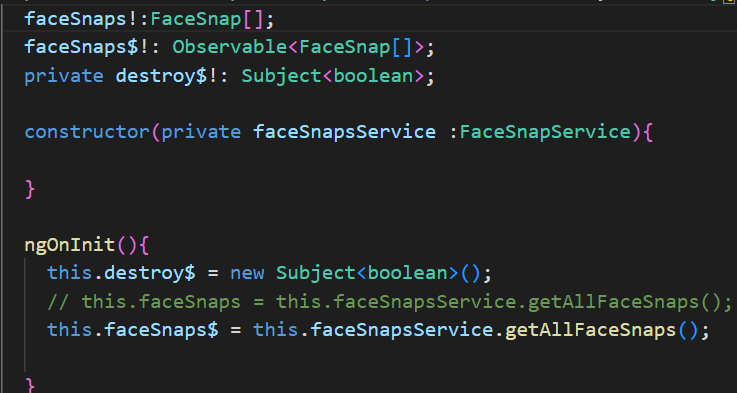
HttpClient : http requests



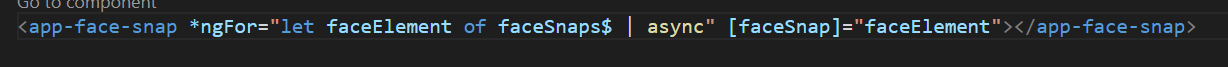




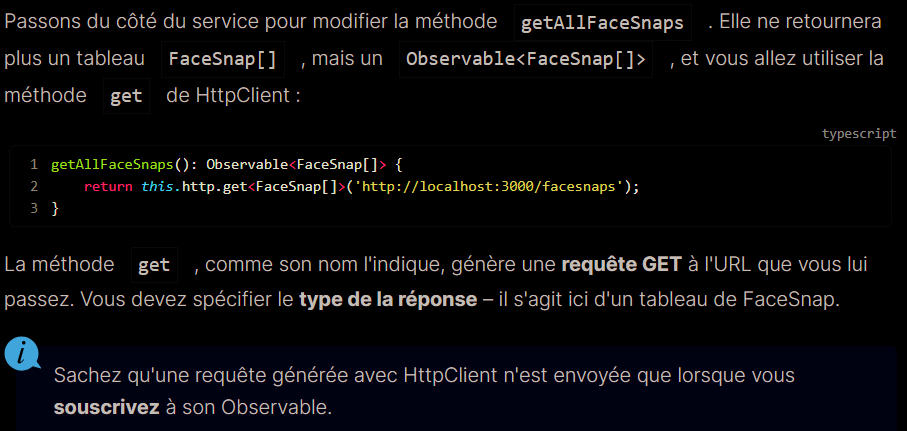


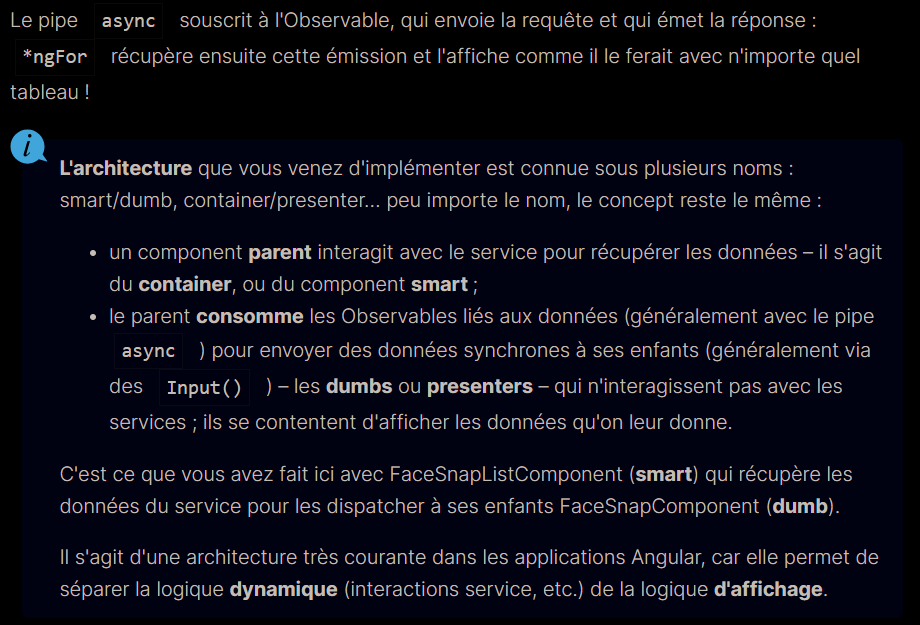


And in the template



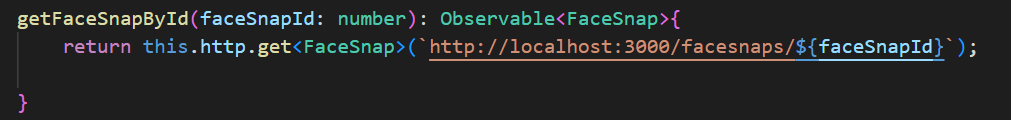
We subscribe to our observable by using the pipe async

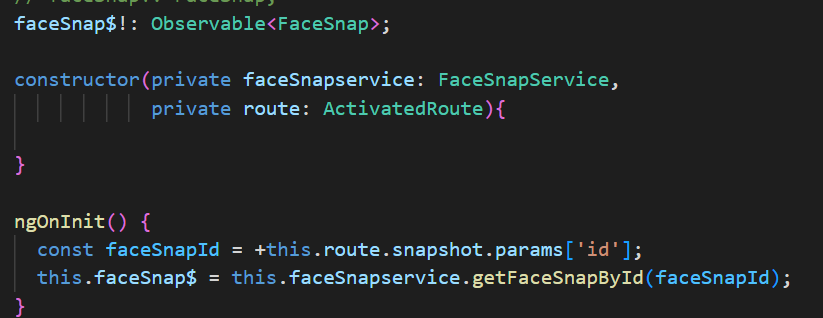




To get a single faceSnap :

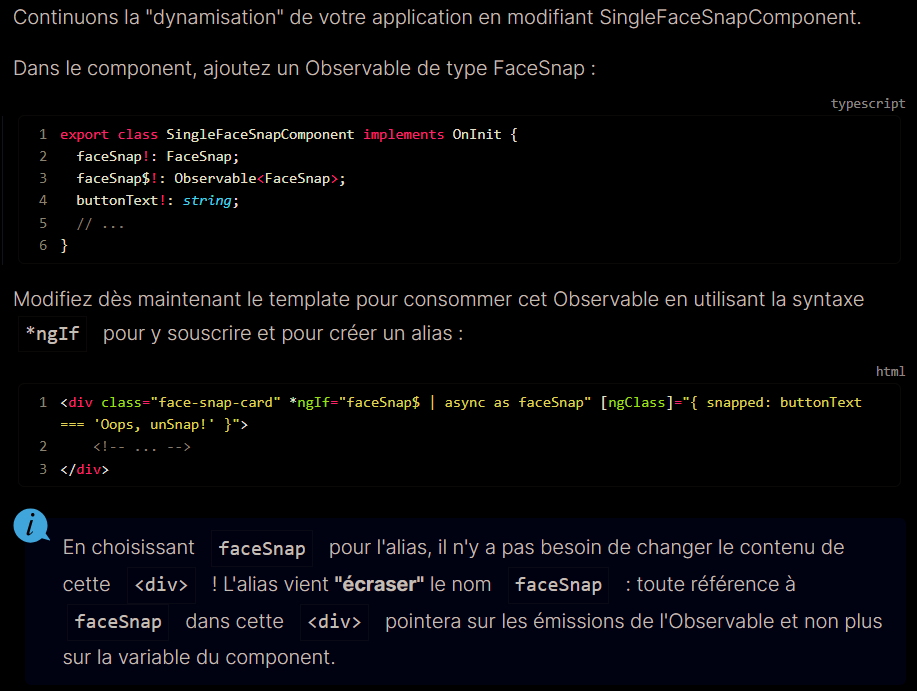
In the server layer :

in the single face snap component



In our template :

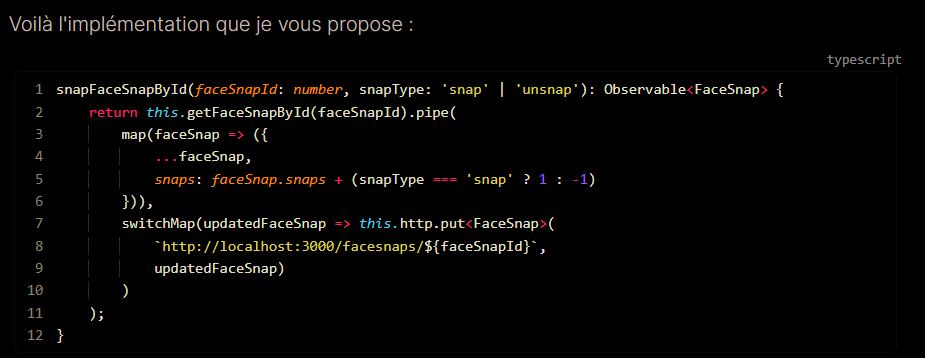
ngIf : if the observable is not null, we subscribe to it with async pipe and we generate an alias called faceSnap.

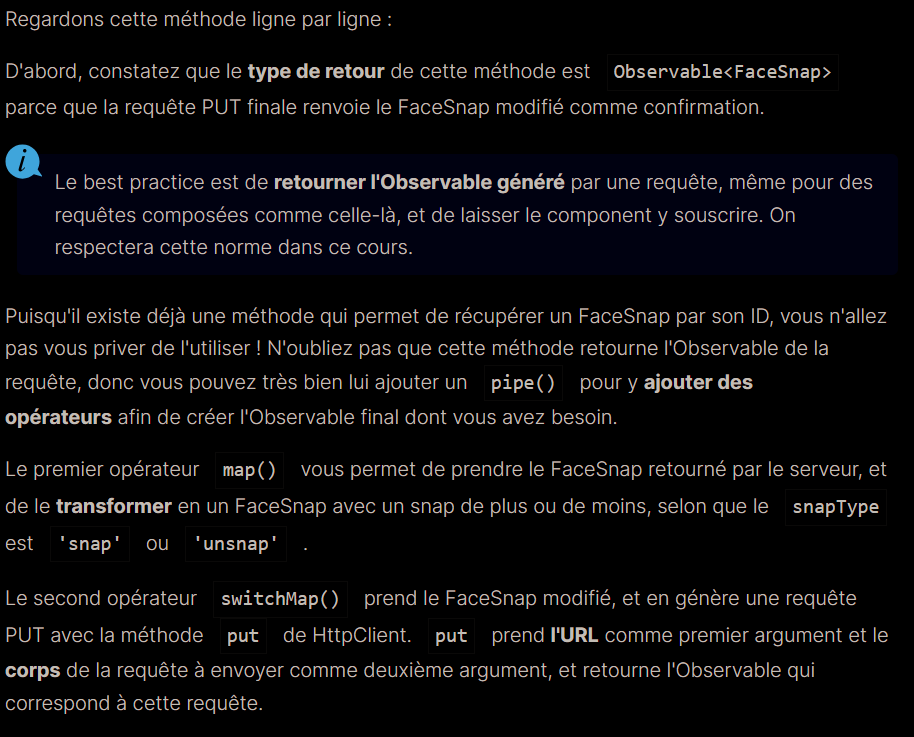


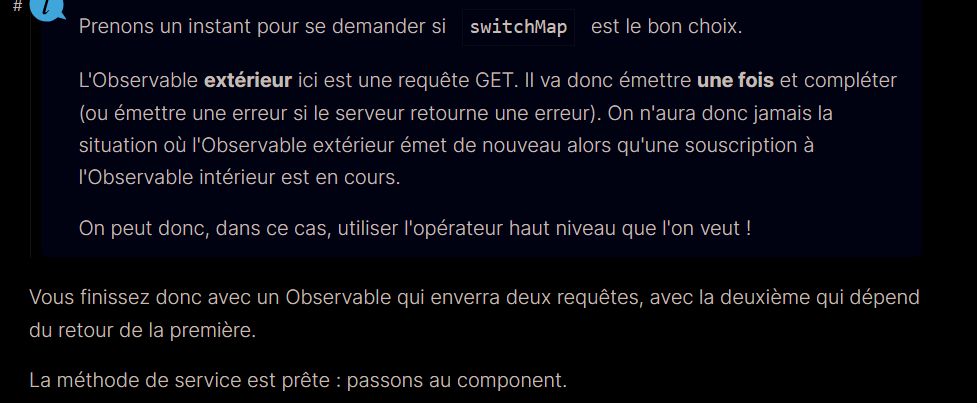




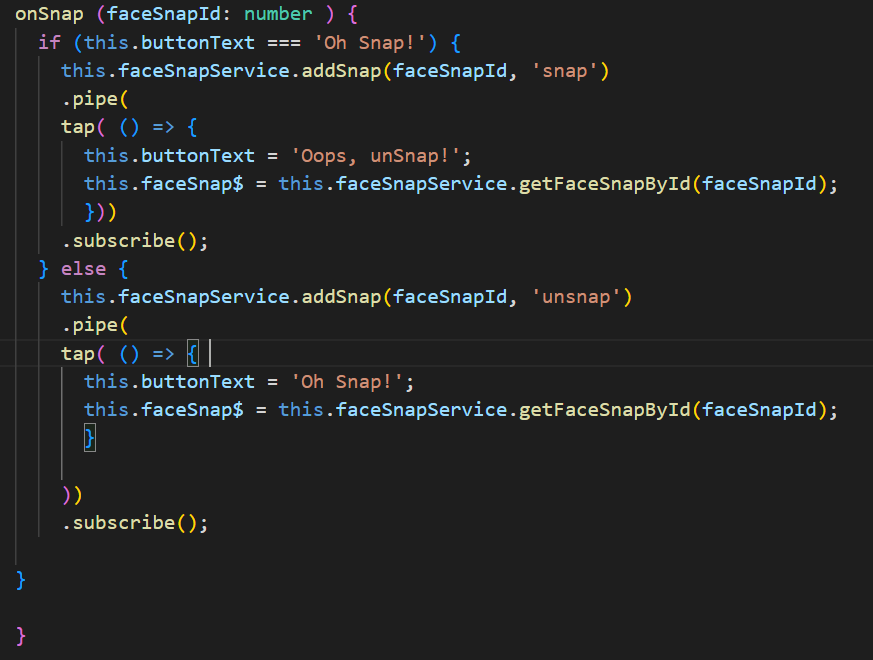
Put request:



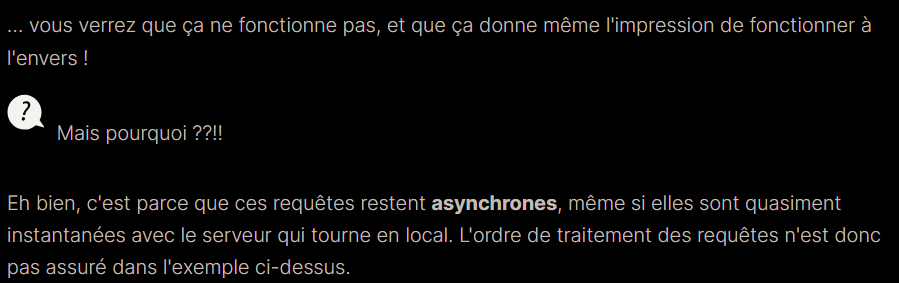


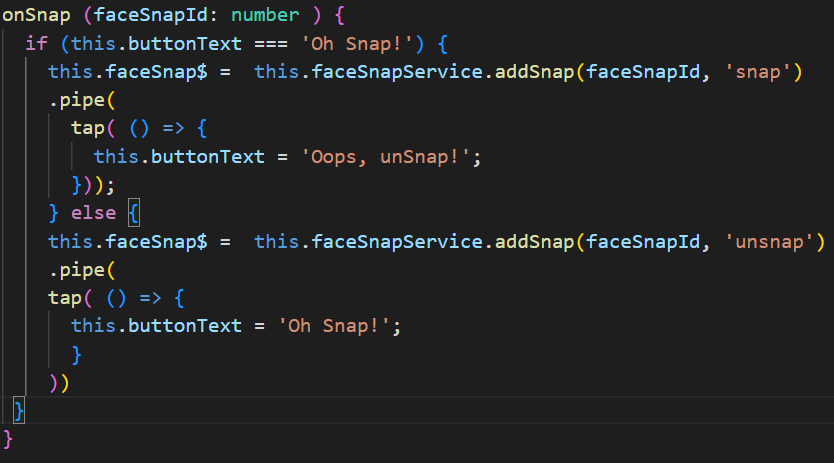


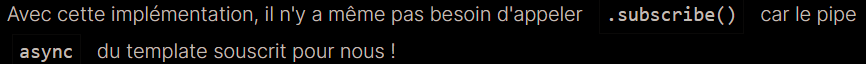
On the component :

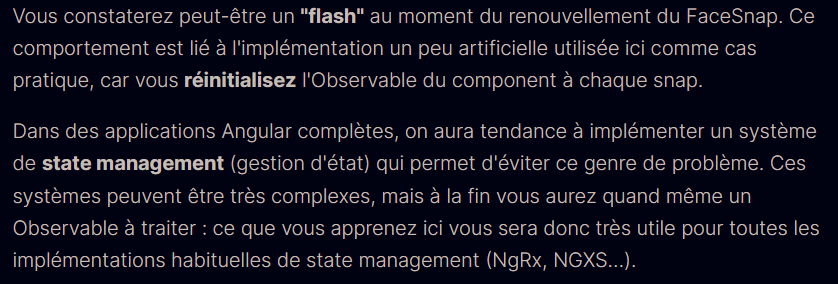


addSnap() return a observable so we have to subscribe to it : the problem is that it s asynchrone, so we use .pipe with tap, which ensures that the observable has emet value before executing the code within.

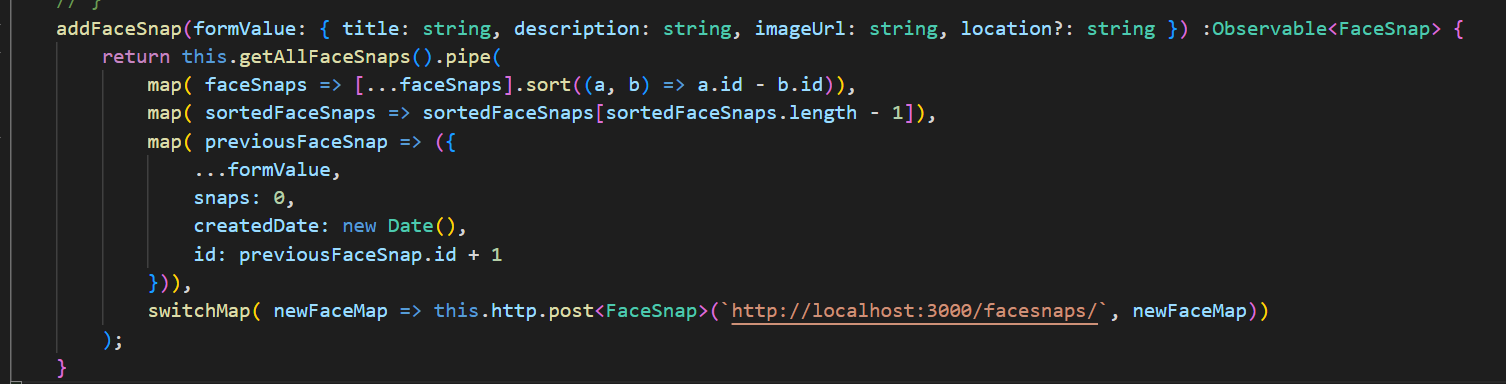




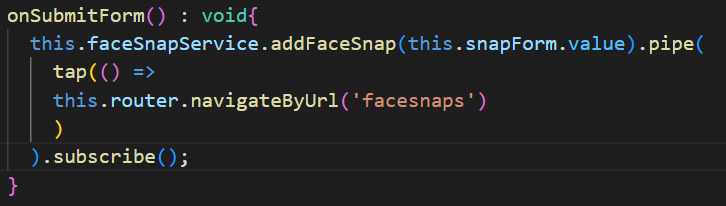




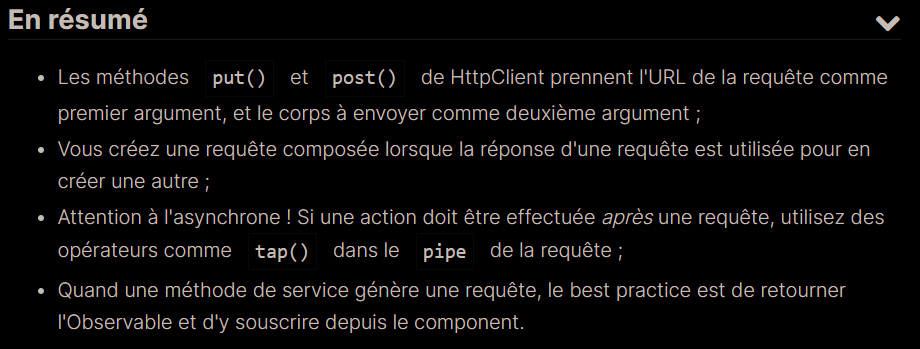
Post method :



On the component :







Sécuriser vos requêtes !

interceptors