ReactiveX JS

Alvin Berthelot

Version 1.1.0

Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons.

Attribution - Partage dans les Mêmes Conditions 3.0 non transposé.

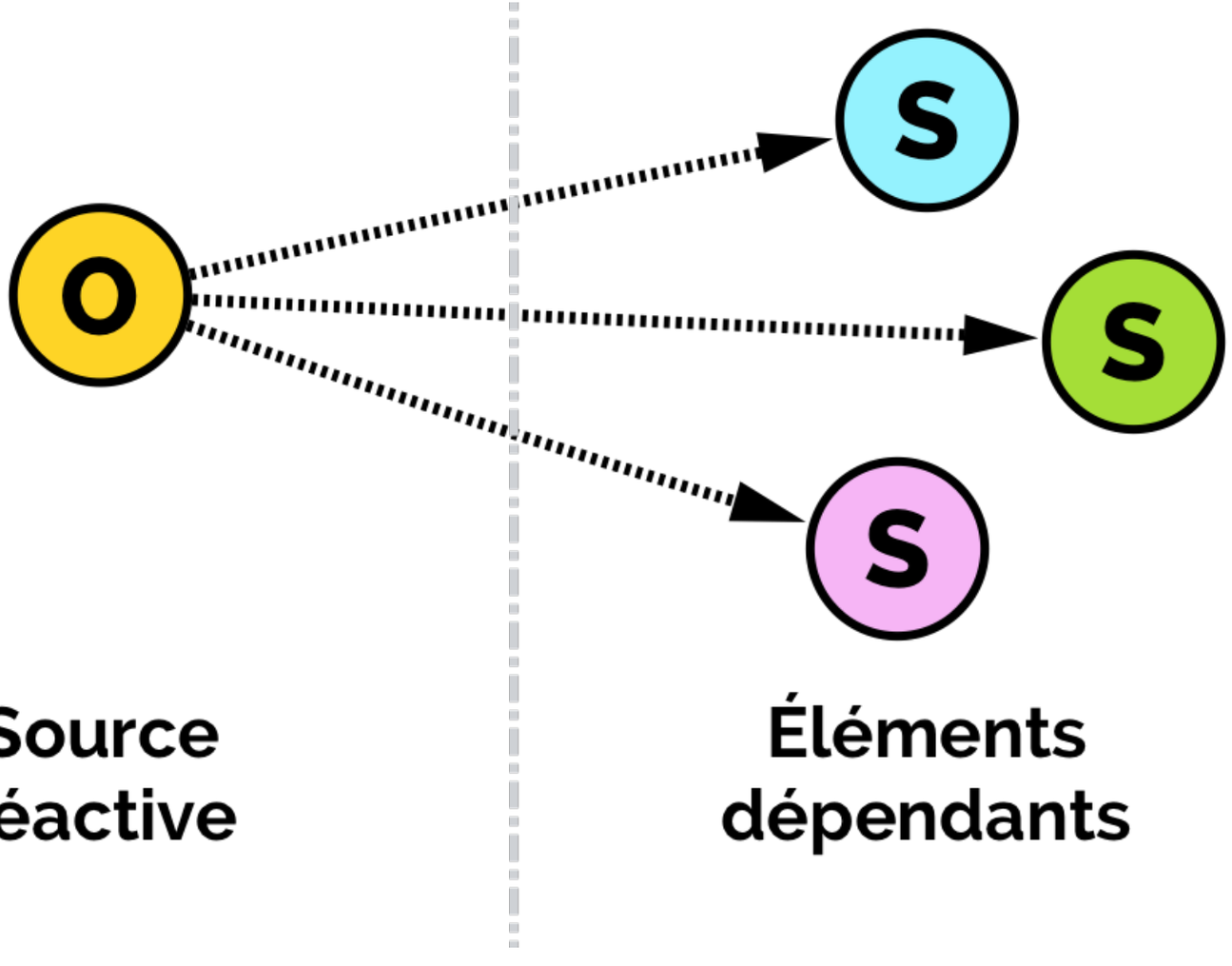
La licence, ses explications ainsi que les moyens de contribution et réappropriation sont détaillés à la fin.

La programmation réactive

Définition de la programmation réactive

> La programmation réactive est un paradigme de programmation visant à conserver une cohérence d'ensemble en propageant les modifications d'**une source réactive** aux **éléments dépendants de cette source**.
>
> — définition Wikipedia

Source réactive / Éléments dépendants



Une source réactive peut être définie comme **un élément génèrant un flux de données dans un laps de temps**, quel que soit le type de données.

La notion d'observable

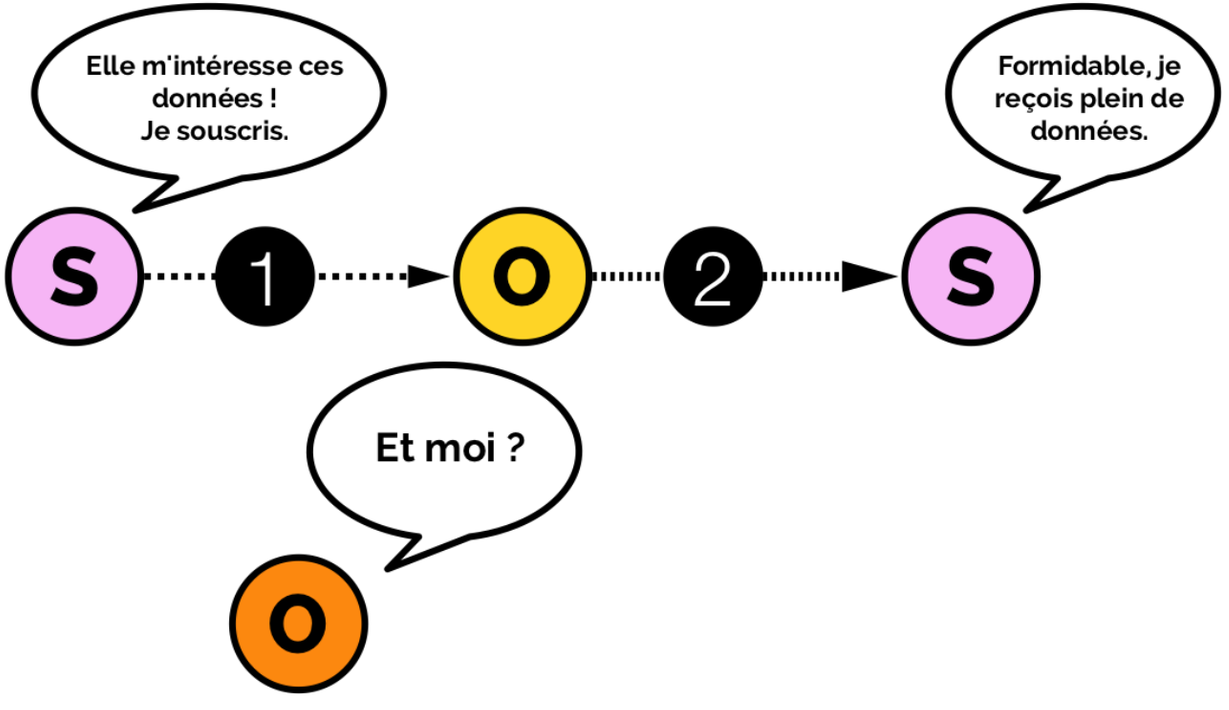
Si l'on prend en considération que tout élément peut générer des données dans un laps de temps, alors quasiment tout peut se définir comme une source réactive :

- Un simple tableau de valeurs.
- Un service retournant des données en AJAX.
- Une connexion en Web Sockets.
- Une IHM Web avec un bouton, ou un champ de saisie ou les mouvements du curseur.
- etc.

Un élément qui génère des données sur un laps de temps peut être observé, d'ailleurs à l'avenir on emploiera le terme d'**observable** au lieu de source réactive.

La notion de souscripteur (ou observateur)

Pour qu'un élément en observe un autre (et devienne **observateur**) il faut qu'il souscrive à ce dernier.



Il faut bien appréhender le fait qu'un élément observable n'est pas nécessairement observé. Par exemple des éléments d'une application n'ont pas nécessairement besoin d'observer les coordonnées x et y du curseur.

La notion de laps de temps

Pourquoi indiquer **"dans un laps de temps"** lorsque l'on parle des observables ?

Parce que généralement **on ne sait pas prédire à quel moment celui-ci va générer des données**.

- Un simple tableau de valeurs : tout de suite.
- Un service retournant des données en AJAX : ça dépend du serveur (et du réseau).
- Le bouton d'une IHM Web : ça dépend quand l'utilisateur aura envie de cliquer.

Si on ne récupère pas les valeurs tout de suite, il s'agit en fait d'un autre moyen de parler d'**asynchronisme**.

"Observable" Vs "Promise"

En JavaScript, le fonctionnement de programmation asynchrone a été couvert par le biais des callbacks mais surtout depuis ES2015 par le biais des "promises".

Il existe plusieurs différences entre les observables et les promises :

- Une "promise" une fois appelée renverra forcément quelque chose et une seule fois. Alors qu'un observable peut ne jamais rien retourner ou retourner de multiple données.
- Une "promise" est forcément asynchrone alors qu'un observable peut être soit synchrone, soit asynchrone ; tout dépend de sa source de données.
- Une "promise" est "eager" alors qu'un observable est "lazy".

Les observables vont donc nous permettre de **travailler sur une collection** et ainsi de manipuler/transformer facilement les données avec des fonctions d'ordre supérieur, le tout de **manière asynchrone** !

La programmation fonctionnelle réactive

Certains éléments sont définis pour produire des données (**les observables**) sans se soucier de qui les consommera.

Certains éléments sont définis pour consommer des données (**les observateurs**) sans se soucier de qui les a produit.

Ainsi un maximum d'éléments peuvent être responsables de leurs propres comportements **sans se soucier du contexte**.

On retrouve ici le paradigme de la programmation fonctionelle qui veut qu'on ne soucie jamais du contexte d'exécution d'une fonction.

Les paradigmes de programmation fonctionnelle et de programmation réactive sont tout à fait compatibles, leur combinaison est appelée **la programmation fonctionnelle réactive** (FRP : Fonctional Reactive Programming).

ReactiveX

Librairie ReactiveX

ReactiveX est une librairie disposant d'API pour la programmation asynchrone reposant sur **la notion d'observable**.

L'utilisation des observables va permettre de construire des applications selon le paradigme de **programmation réactive**.

Cette librairie est implémentée pour différents langages : Java, JavaScript, .NET, Swift, ainsi que pour de nombreux autres langages.

Par la suite, nous nous focaliserons bien évidemment sur ReactiveX pour JavaScript, communément appelée RxJS.

Déclarer un observable avec RxJS

Il existe de nombreux opérateurs pour créer un observable avec RxJS, nous verrons cela avec l'API et les opérateurs de création.

Pour l'instant créons notre propre flux de données en déclarant nos données (la suite de Fibonacci) à envoyer une par une.

```
let fibonacciObservable = Rx.Observable.create(function(observer) {
  observer.next(0)
  observer.next(1)
  observer.next(1)
  observer.complete()
})
```

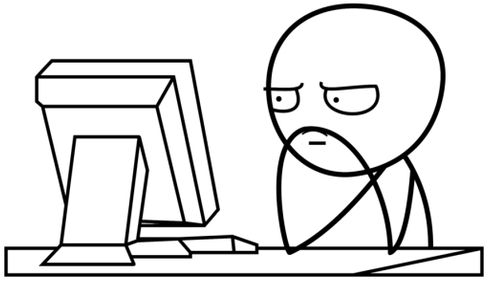
Un observable dispose de 3 callbacks sur l'observateur : `next`, `error` et `complete` (Cf. documentation) pour lui indiquer le traitement de son flux.

Déclarer un observateur avec RxJS

Un observable en soit n'a que très peu d'intérêt, cela devient intéressant lorsqu'un (ou plusieurs) observateur(s) consomme(nt) son flux de données.

```
fibonacciObservable.subscribe(value => {
  console.log('Value', value)
})
// Value 0
// Value 1
// Value 1

fibonacciObservable.subscribe(
  value => {
    console.log('Fibonacci', value)
  },
  error => {
    console.error(error)
  },
  () => {
    console.log('Fibonacci c\'est fini')
  })
// Fibonacci 0
// Fibonacci 1
// Fibonacci 1
// Fibonacci c'est fini
```



Exercices

Vous allez déclarer un tableau de valeurs. Avec vous devez afficher la somme du tableau (uniquement les nombres) dans la console.

```
var source = [0, 1, 1, 'foo', 2, 3, 5, 'bar', 8, 13];
```

1. En programmation fonctionnelle.
2. En utilisant les observables et les observateurs de RxJS, de manière synchrone.
3. Indiquer une erreur lorsque le flux contient une valeur qui n'est pas un nombre.

Fibonacci en mode `Function`

Pour arriver à un résultat similaire sans `Observable` nous aurions très certainement écrit cela.

```
let fibonacciArray = () => [0, 1, 1, 2, 3, 5, 8, 13]

fibonacciArray().forEach(value => {
  console.log('Value', value)
})
// Value 0
// Value 1
// ...

fibonacciArray().forEach(value => {
  console.log('Fibonacci', value)
})
// Fibonacci 0
// Fibonacci 1
// ...
```

`Function` Vs `Observable`

Quelles sont les ressemblances et les différences entre ces 2 approches ?

La ressemblance principale est que nous demandons explicitement d'avoir accès aux données, soit en invoquant la fonction `fibonacciArray()`, soit en souscrivant à l'observable `fibonacciObservable.subscribe(⋯)`.

Avec un observable l'itération est implicite puisque potentiellement nous pouvons récupérer une multitude de données.

Avec un observable nous pouvons récupérer les données une par une sur un laps de temps alors qu'avec une fonction "classique" toutes les données sont récupérées en une seule fois. Cela ne se perçoit pas dans notre exemple.

Déclarer un observable asynchrone avec RxJS

Déclarons désormais notre observable à partir d'un tableau avec l'opérateur `from` pour gagner en similitude.

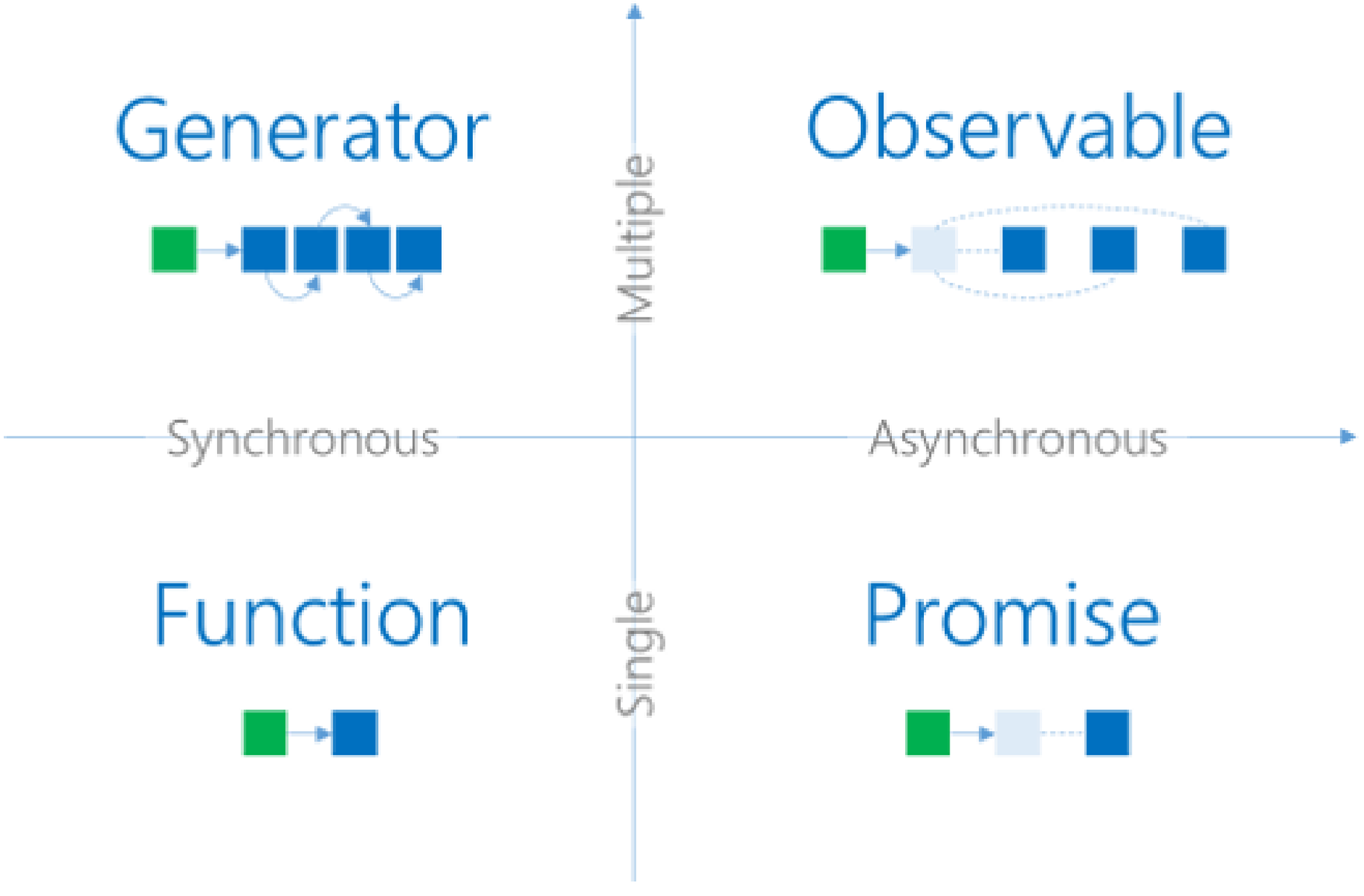
```
let fibonacciObservable = Rx.Observable.from([0, 1, 1, 2, 3, 5, 8, 13])
```

Seulement notre exemple n'est toujours pas pertinent avec RxJS puisqu'il utilise toujours un observable synchrone.

Désormais **déclarons un observable asynchrone** en utilisant les opérateurs `interval` et `zip`.

```
let fibonacciObservable = Rx.Observable
  .fromArray([0, 1, 1, 2, 3, 5, 8, 13])
  .zip(Rx.Observable.interval(1000), (value, index) => value)
```

Comparatif


Generator
Observable
Multiple
Synchronous
Asynchronous
Function
Promise
Single

Licence

CC BY-SA 3.0

Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons. Attribution - Partage dans les Mêmes Conditions 3.0 non transposé.

Copyright © 2017 Alvin Berthelot.

Pour toutes questions, réclamations ou remarques, merci d'envoyer un message à alvin.berthelot@webyousoon.com.

Explications licence CC BY-SA 3.0

Cette licence permet aux autres de remixer, arranger, et adapter votre œuvre, même à des fins commerciales, tant qu'on vous accorde le mérite en citant votre nom et qu'on diffuse les nouvelles créations selon des conditions identiques.

Cette licence est souvent comparée aux licences de logiciels libres, "open source" ou "copyleft".

Toutes les nouvelles œuvres basées sur les vôtres auront la même licence, et toute œuvre dérivée pourra être utilisée même à des fins commerciales.

C'est la licence utilisée par Wikipédia ; elle est recommandée pour des œuvres qui pourraient bénéficier de l'incorporation de contenu depuis Wikipédia et d'autres projets sous licence similaire.

Contribution et réappropriation

Ce fichier PDF est généré avec Asciidoctor à partir d'un dépôt Git se trouvant sous GitHub.

https://github.com/alvinberthelot/slides-rx-js

Cela signifie que vous n'avez pas besoin de vous battre avec un fichier binaire (le PDF) pour **contribuer**, **vous réapproprier le contenu** ou **modifier le thème** de présentation.

Contribution

Vous voulez **contribuer au contenu** car :

- Il y a une erreur (ça arrive à tout le monde), de typographie, de compréhension, ou tout autre chose.
- Vous souhaitez apporter une précision.

Il vous suffit de contribuer au projet via Git par le moyen d'une "pull request" sur le dépôt Git.

Réappropriation

N'oubliez pas les conditions de la licence.

Vous voulez vous **réapproprier le contenu** car :

- Vous souhaitez donner un style différent.
- Vous souhaitez enlever/ajouter/modifier des sections dans votre contexte.

Il vous suffit de "forker" le dépôt Git et d'y apporter vos propres modifications, puis de générer par vous même le nouveau PDF.