Université de Montréal

Du bon usage de la programmation réactive avec Rx.JS

par
Patrick Chartrand

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Travail présenté à Antonio Tavares
dans le cadre du cours IFT 1142,
programmation côté client

21 avril 2020

Table des matières

Résumé.....3

Introduction.....4

Le flux de données.....4

La nature asynchrone.....6

Le concept des observables.....8

Conclusion.....10

Bibliographie.....11

Résumé

Rx.JS est actuellement l'une des technologies de la programmation Web parmi les plus utilisées et les plus efficaces au niveau des possibilités et de l'ingéniosité des fonctions. Entre autres, elle permet un contrôle aisé des flux de données aussi bien de manière synchrone que asynchrone via des opérations notamment ergonomiques et réactives grâce aux observables.

Rx.JS, ou les extensions réactives pour JavaScript, constitue une bibliothèque réactive qui est la plus documentée et la plus fournie au niveau des opérateurs disponibles à l'heure actuelle. C'est qu'elle est une implémentation dans le langage JavaScript par les *Reactives Extensions*, à savoir un vaste projet d'API mieux connu sous le nom de ReactiveX (2020). Cette bibliothèque, notamment arrivée à sa cinquième version, peut être utilisée dans le cadre de plusieurs travaux aussi bien du côté client – *Front-End* – dans des environnements Angular que du côté serveur – *Back-End* – via des applications avec Nodes.js. Dans le cadre de cette recherche, nous chercherons à mieux comprendre ce qu'est la technologie Rx.JS par de la théorie sur les flux de données, par la nature essentiellement asynchrone du logiciel et, enfin, par le concept fondamental des observables.

Le flux de données

Tout d'abord, l'une des structures théoriques qui est essentielle à la bibliothèque Rx.JS est la notion de flux. C'est que les extensions réactives pour JavaScript permettent de manipuler des séquences ordonnées de données qui circulent dans le temps. Ce faisant, le flux de données, pouvant prendre la forme d'informations brutes du type JavaScript, peut subir des modifications et des mises à jour sans être arrivé à la fin de son émission (Farhi, 2017, p. 6). Dans un modèle d'encodage synchrone, c'est-à-dire de base par sa linéarité, le suivi des données et l'exécution de leur variable sont clairement lisibles, ou identifiables,

puisque chaque étape du flux est déterminée et prévue en un lieu précis dans le temps et dans l'espace. En effet, le schéma (figure 1) suivant illustre bien le déroulement d'un flux de données basique, à savoir synchrone :

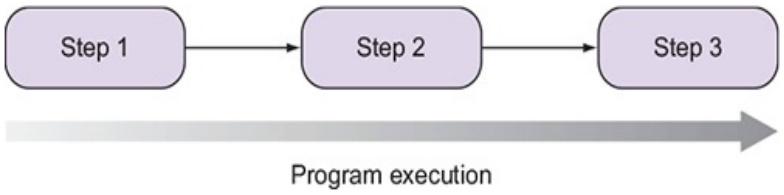

Figure 1. Exécution de programme synchrone (Daniels et Atencio, 2017, p. 32).

Dans cette illustration, sont aisément observables les différents états du flux de données dans la durée, ou la ligne temporelle, ce qui permet à la gestion du programme de ne rencontrer aucune contrainte. Par exemple, dans le cas d'une programmation JavaScript, l'étape première pourrait signaler une variable, entre autres, du type « string » ou « number » représentée par un évènement « onClick »; l'étape deuxième, une erreur dans l'exécution si elle a lieu; l'étape troisième, l'arrivée à terme du flux et, ce faisant, la complétion du transfert de données. Toutefois, là où tout se complique et où la programmation réactive telle que Rx.JS intervient, c'est dans un système qui doit appréhender des changements de données dans le flux à n'importe quel moment dans la durée : Rx.JS peut alors réagir par une notification à toutes les étapes du processus afin qu'elles puissent s'adapter à tout asynchronisme.

La nature asynchrone

Ensuite, si un système doit effectivement récupérer des données de façon dite asynchrone, Rx.JS est en mesure d'exécuter une fonction n'interrompant pas le déroulement du script et qui ne sera retournée qu'à la fin du processus. En effet, les flux de données asynchrones sont composés à partir de sources multiples telles que divers évènements du DOM : les opérations Rx.JS offrent alors la possibilité à l'utilisateur d'interagir avec les données dès leur disponibilité, à savoir en tout temps (Vidal, 2018, p. 29). L'image qui suit (figure 2) montre bien des asynchronismes à l'œuvre :

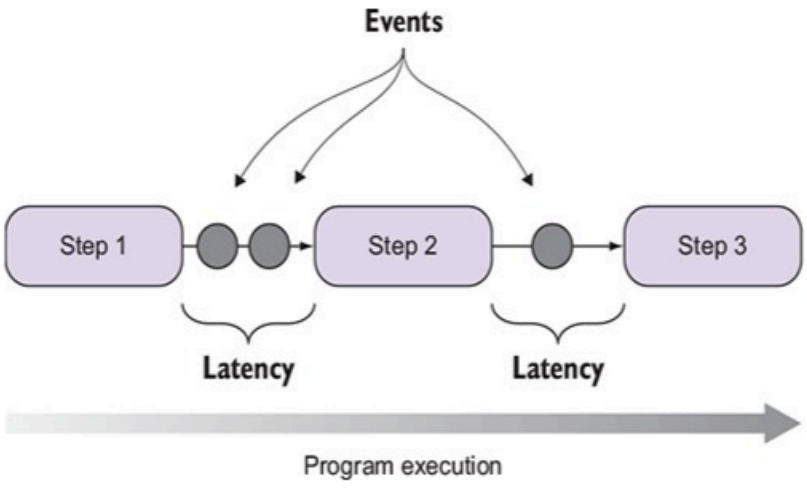

Figure 2. Exécution de programme asynchrone (Daniels et Atencio, 2017, p. 32).

Cependant, un changement de paradigme s'impose ici, c'est-à-dire que la logique synchronique du « Do this; then immediately do that » passe à une autre

se résumant à « Do this (wait for an indeterminate period of time); then do that » (Daniels et Atencio, 2017, p. 13); ce temps d'attente, ou de latence, entre une et plusieurs requêtes n'est néanmoins pas envisageable pour des raisons de performance du réseau. Du fait que le système ne puisse alors rester inactif en attendant le retour d'une requête, Rx.JS permet de mettre à profit les périodes de latence par un programme toujours réactif via la continuation du traitement d'autres données. Par exemple, une fonction Rx.JS de rappel remet à plus tard la gestion d'une tâche pour continuer avec les autres, soit le code suivant :

```
ajax('/data',
  items  => {
     items.forEach(item => {
        // process each item
     });
});
beginUiRendering();
```

(Daniels et Atencio, 2017, p. 25).

Dans cette fonction, l'exécution reporte ultérieurement la gestion d'information afin de donner libre cours à la complétion du programme, ce qui permet d'entamer d'autres tâches plutôt que de stagner. Donc, le rôle de Rx.JS est considérable, et même à ne pas négliger, dans la conception d'applications optimales, notamment asynchrone.

Le concept des observables

Mais la conception d'applications Rx.JS repose sur une autre composante non négligeable, à savoir l'un des principaux concepts du logiciel qui est celui des observables. C'est qu'un observable contient les données qui peuvent éventuellement être mises à jour ou portées à changer au fil du temps. Ainsi, cet objet permet de suivre, ou littéralement d'observer, la progression du flux pour aviser de tout changement au fur et à mesure de la durée (Farhi, 2017, p. 8). C'est l'idée d'une collection de valeurs ou d'évènements futurs qui serait appelée que représente un observable, soit le code qui suit :

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

const foo = new Observable(subscriber => {
  console.log('Hello');
  subscriber.next(42);
  subscriber.next(100); // "return" another value
  subscriber.next(200); // "return" yet another
  setTimeout(() => {
    subscriber.next(300); // happens asynchronously
  }, 1000);
});

console.log('before');
foo.subscribe(x => {
  console.log(x);
});
console.log('after');
```

(2020).

Dans cet exemple, l'opération Rx.JS rend possible non seulement le retour de

multiples valeurs via une fonction propre à la bibliothèque telle que *observable.subscribe()*, mais aussi l'affichage asynchrone de la donnée ayant la valeur numérique « 300 », et ce, simplement en quelques lignes de code. De plus, sont à distinguer deux catégories des objets en question ici qui sont les observables froids et les observables chauds (Mezzalira, 2018, p. 72). La différence entre les deux réside, entre autres, leur pratique : un observable froid attend une action de l'usager en exécutant un flux de données depuis le début, dont la mécanique a un aspect synchronique, tandis qu'un observable chaud s'active tout seul – du moins de manière programmée – et exécute un flux à partir d'une source, c'est-à-dire d'une donnée, à chaque souscription; c'est une mécanisme multi-source pour le second. Ainsi, un observable de la catégorie froide se construirait de la façon suivante :

```
import Rx from "rxjs";
const source = Rx.Observable.interval(2000).startWith(123)
source.subscribe(value => console.log("first observer", value))
setTimeout(_ =>{
   source.subscribe(value => console.log("second observer", value))
}, 5000);
setTimeout(_ =>{
   source.subscribe(value => console.log("third observer", value))
}, 8000)
```

(Mezzalira, 2018, p. 78)

Par cet exemple, l'exécution du flux de données est clairement indiquée comme débutant à un moment précis via l'opérateur *startWith*. Or, dans une

programmation de la catégorie chaude, l'opérateur précédant sera suivi d'autres tels que *.publish()* et *.refCount();*, dont l'un et l'autre permettent respectivement de ne retourner une valeur que lorsqu'elle est appelée, puis de ne pas définir un déclencheur. Donc, les observables chauds offrent la possibilité d'élargir une utilisation Rx.JS, notamment du fait que la catégorie froid est la mise en forme par défaut.

Conclusion

En somme, une compréhension de la technologie Rx.JS passe notamment à travers un cadre théorique qui revisite la notion même de flux, ainsi que par la mise en relief de la nature asynchrone qui est essentielle à la conception du logiciel. Le concept clé qui est celui des observables constitue, de surcroit, l'un des points fondamentaux pour bien comprendre l'usage des opérateurs Rx.JS. D'ailleurs, se pratiquant naturellement dans un contexte de programmation réactive, ces opérations tendent à assouplir les flux de données par un contrôle plus grand et plus libre de leur source, notamment par leur multiplication. Les opérateur Rx.JS rendent également possible la manipulation des données au sein de la linéarité du temps d'exécution, ce qui permet l'optimisation d'un programme par des transformations dans la chaine même et non dans la source des données.

Bibliographie

Daniels, Paul P. et Atencio, L. (2017). *RxJS in Action*. O'Reilly.

Farhi, O. (2017). *Reactive Programming with Angular and ngrx*. Apress.

Mezzalira, L. (2018). *Front-End Reactive Architectures*. Apress.

RxJS. (2020). *Overview*. https://rxjs-dev.firebaseapp.com/guide/overview

Vidal, C. (2018). *Programmation web réactive* [thèse de doctorat, Université Côte d'Azur]. NNT. https://tel.archives-ouvertes.fr/tel-01900619