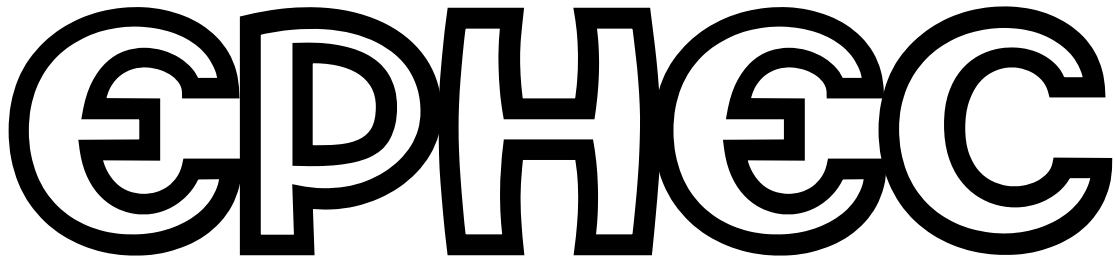


Ecole Pratique des Hautes Etudes Commerciales



Institut d'Enseignement Supérieur Economique de Type Court

**PROGRAMMATION RÉACTIVE ET DONNÉES TEMPS-
RÉEL: LE CAS D'UNE APPLICATION E-COMMERCE**

Samih ALKEILANI

**Rapporteur(s)
J. HECQUET**

Travail de Fin d'Études présenté
en vue de l'obtention du diplôme de
BACHELIER EN INFORMATIQUE DE GESTION

Année académique 2017-2018

Avant-propos

Je remercie le professeur Jean-Paul Hecquet, mon rapporteur, pour sa disponibilité et ses conseils qui m'ont permis de fournir un travail que j'espère de qualité.

Je tiens également à remercier Davy Engone de la société Hackages pour nos échanges d'idées et ses recommandations liées à la programmation réactive.

Je remercie Gianni Tagliarino pour ses conseils et nos discussions très intéressantes, ainsi que tous mes collègues de chez Delhaize et mi8 qui ont contribué de près ou de loin à la réalisation de ce travail.

Enfin, merci à ma famille et à mes amis pour leurs remarques et commentaires.

Table des matières

1 Chapitre I : Introduction	3
1.1 Contexte et objectifs du travail de fin d'études	3
1.2 Structure du travail de fin d'études	5
2 Chapitre II : Programmation réactive	7
2.1 Origines	7
2.2 Définitions	13
2.3 Propriétés	16
3 Chapitre III : Architecture générale	17
3.1 Systèmes basés sur les événements	17
3.1.1 Origines	17
3.1.2 Notions	19
3.1.3 Avantages et inconvénients	19
3.1.4 Comparaison avec le modèle Requête/Réponse	20
3.1.5 Systèmes à envoi direct	23
3.1.6 Systèmes à envoi indirect	24
3.1.7 Comparaison	25
3.2 Le patron de conception Observer	26
3.2.1 Cas concret	26
3.2.2 Fonctionnement	27
3.2.3 Propriétés	28
3.2.4 Points clés	30
3.3 Reactive Extensions	31
3.3.1 Généralités	31
3.3.2 Observable	31
3.3.3 Opérateurs	34
3.4 Flux de données unidirectionnel	36
3.4.1 Concepts	36
3.4.2 Store, single source of truth	37
3.4.3 Comparaison avec Model-View-Controller	38
3.4.4 Points clés	40

4 Chapitre IV : Technologies back end	42
4.1 Introduction au NoSQL	42
4.1.1 Origines	42
4.1.2 La capacité de monter à l'échelle ou scalability	44
4.1.3 Bases de données distribuées	45
4.1.4 Le mouvement NoSQL	50
4.2 Introduction à l'authentification	53
4.3 Backend as a Service (BaaS) et la plateforme Firebase	57
4.3.1 Introduction	57
4.3.1.1 Backend as a Service	57
4.3.1.2 Choix d'utilisation de Firebase	59
4.3.2 Firebase Realtime Database	60
4.3.2.1 Généralités	60
4.3.2.2 Monter à l'échelle avec Firebase Realtime Database	61
4.3.2.3 Structures de données imbriquées et plates	62
4.3.2.4 Relations entre documents	66
4.3.2.5 Temps-réel	67
4.3.3 Firebase Authentication	68
5 Chapitre V : Application multi- plateformes réactive avec Xamarin	69
5.1 Introduction	69
5.1.1 Choix d'utilisation de Xamarin	69
5.2 Environnement de programmation	71
5.3 Architecture réactive envisagée	72
5.3.1 Légende des interactions entre composants	72
5.3.2 Interaction entre l'application et le back-end	73
5.3.3 Architecture réactive de l'application	74
5.3.3.1 Schématisation	74
5.3.3.2 Dépendances	76
5.3.4 Programmation réactive	77
5.3.4.1 Du back-end aux services	77
5.3.4.2 Des services au store	78
5.3.4.3 Du store aux vues	79
5.3.4.4 Dans les vues	81
5.3.4.5 Cohérence du flux des données	82
5.3.4.6 Résumé	83
6 Chapitre VI : Cas de l'application BdOccaz	84
6.1 Présentation	84
6.1.1 Contexte	84

6.1.2 Solution	84
6.1.3 Remarques	85
6.2 Fonctionnalités	86
6.2.1 Consultation des articles mis en vente	86
6.2.2 Détail d'une vente	87
6.2.3 Recherche, article indisponible	88
6.2.4 Recherche, détails vente	89
6.2.5 Recherche, achat de la bande dessinée en vente	90
6.2.6 Vendre, identification de la bande dessinée	91
6.2.7 Vendre, confirmation	92
6.2.8 Transactions, achats en cours	93
6.2.9 Transactions, détail achat	95
6.2.10 Transactions, ventes en cours	97
6.2.11 Transactions, détail vente	99
6.2.12 Favoris	101
6.2.13 Connexion	102
6.2.14 Création de compte BdOccaz	103
6.2.15 Compte	104
6.2.16 Profil utilisateur	106
6.3 Règles concernant les données	107
6.3.1 Correspondance avec le NoSQL	107
6.3.2 Règles de structure	107
6.3.3 Règles de validation	109
6.4 Schéma entités-associations	110
6.5 Base de données	111
6.5.1 Création de tables intermédiaires pour les relations many-to-many	111
6.5.2 Schéma de la base de données	113
7 Chapitre VII : Conclusion	114
8 Bibliographie	116
9 Annexes	124
9.1 Stack technologique	124
9.1.1 Back-end Firebase	124
9.1.2 Front-end Xamarin	124
9.2 Description des associations du schéma entité-associations	125

1 Chapitre I : Introduction

1.1 Contexte et objectifs du travail de fin d'études

*"(...) If we build our system to be event-driven, we can more easily achieve scalability and failure tolerance (...). A scalable, decoupled, and error-proof application is fast and responsive to users."*¹

Les applications d'aujourd'hui deviennent de plus en plus interactives, répondant à toutes sortes d'événements provenant de leur environnement interne et externe. Elles traitent ainsi des événements et exécutent les tâches qui y sont liées, comme modifier l'état de l'application ou afficher des données².

L'interface utilisateur représente généralement la partie la plus interactive de l'application et doit réagir à (et coordonner) une multitude d'événements de sources différentes : appui sur un bouton, notification venant d'une plateforme *cloud*, affichage du résultat d'une opération asynchrone exécutée en arrière-plan...

Ces événements contrôlent l'exécution de l'application, qui a la responsabilité de réagir à ces derniers. Ce principe est parfois appelé inversion de contrôle, ou *the Hollywood Principle*³.

La programmation réactive a récemment gagné en popularité, proposée comme solution adaptée au développement d'applications modernes basées les événements.⁴

Individus et sociétés communiquent leur propre vision de la programmation réactive à la communauté⁵, ce qui entraîne une incompréhension dans le domaine. Notons que ces

¹ Nickolay Tsvetinov, [Learning Reactive Programming with Java 8](#)

² "Reactive programming in Standard ML - IEEE Conference Publication."
<https://ieeexplore.ieee.org/document/674156/>. Accessed 28 Aug. 2018.

³ "Don't call us, we'll call you"

⁴ "MASTER THESIS Reactive Programming with Events - Tomas Petricek."
<http://tomasp.net/academic/theses/events/events.pdf>. Accessed 28 Aug. 2018.

⁵ "The Essence of Reactive Programming - TU Delft Repositories."
<https://repository.tudelft.nl/islandora/object/uuid:bd900036-40f4-432d-bfab-425cdebc466e/datastream/OBJ/download>. Accessed 29 Aug. 2018.

définitions diffèrent souvent au niveau de l'accent mis sur certains aspects ou du détail technique employé.

Dans ce travail de fin d'études, nous tenterons de faire converger des définitions tirées de plusieurs travaux universitaires vers une description générale de la programmation réactive. Nous étudierons également ses grands principes, ainsi que son utilisation dans une application de vente et d'achat de bandes dessinées nommée *BdOccaz*, développée avec la technologie *Xamarin*⁶.

Xamarin est une plateforme permettant de concevoir des applications multi-plateformes (*iOS* et *Android* et *Windows*) avec le langage de programmation *C#*. À l'heure actuelle et à notre connaissance, il n'existe en *Xamarin* aucune librairie proposant une architecture applicative réactive alliée à la notion de flux de données unidirectionnel (les données de l'application suivent le même sens lors de leur traitement, ce qui rend la logique plus prévisible et plus facile à suivre)⁷, bien que ces pratiques soient largement répandues dans bon nombre de technologies web⁸. Nous citerons par exemple l'utilisation de librairies *JavaScript* comme *Flux*⁸ ou *Redux*⁹.

Nous envisagerons une telle architecture au [point 5.3](#) en nous inspirant des patrons de conception utilisés dans les librairies précédemment mentionnées et la mettrons en place dans l'application *BdOccaz*.

Enfin, nous présenterons l'application et donnerons des indications quant aux bénéfices de la programmation réactive au sein de celle-ci.

⁶ "Xamarin App Development with Visual Studio | Visual Studio." 21 Aug. 2018, <https://visualstudio.microsoft.com/xamarin/>. Accessed 29 Aug. 2018.

⁷ "Reactivity, state and a unidirectional data flow | TamTam." <https://www.tamtam.nl/en/reactivity-state-and-a-unidirectional-data-flow/>. Accessed 29 Aug. 2018.

⁸ "Flux | Application Architecture for Building User Interfaces." <https://facebook.github.io/flux/docs/in-depth-overview.html>. Accessed 29 Aug. 2018.

⁹ "Data Flow - Redux." <https://redux.js.org/basics/dataflow>. Accessed 29 Aug. 2018.

1.2 Structure du travail de fin d'études

Nous introduirons, au chapitre 2, la programmation réactive, ses avantages et ses grands principes pour finalement tenter d'apporter une définition de ce terme aguicheur.

Les notions de base acquises, nous examinerons (au chapitre 3) différentes techniques et patrons de conception utilisés en programmation réactive. L'objectif est de fournir une compréhension des différents concepts utilisés durant la conception de l'application de bandes dessinées *BdOccaz*.

Dans le chapitre 4, nous étudierons les technologies choisies côté serveur pour la conception de notre application. Dans ce cadre, nous ferons un détour par une introduction aux bases de données *NoSQL*¹⁰ et à l'authentification, ces concepts étant utilisés au sein de notre back end. Nous comparerons par exemple les systèmes *NoSQL* aux *SGBRs*¹¹, analyserons certaines de leurs propriétés et parlerons du standard *OAuth 2.0*.

Le chapitre 5 concernera les technologies front end employées. Nous décortiquerons au [point 5.3](#) un élément fondamental de ce travail, l'architecture réactive pensée pour notre application multi-plateformes *Xamarin*, qui établit un lien pratique entre les différentes techniques vues au chapitre 3.

Enfin, dans le chapitre 6, nous présenterons des mock-ups de l'application *BdOccaz* et remplacerons l'utilisation de la programmation réactive dans le contexte de certains écrans de l'application. Nous exposerons finalement les règles de structure ayant conduit au schéma entité-relations et présenterons le schéma relationnel de la base de données.

¹⁰ "What is NoSQL? | Nonrelational Databases, Flexible Schema Data"
<https://aws.amazon.com/nosql/>. Accessed 30 Aug. 2018.

¹¹ Systèmes de gestion de bases de données relationnelles

Première partie : Théorie

Le terme programmation réactive est très à la mode et est souvent utilisé de manière contextuelle, ce qui laisse place à une certaine confusion quant à son essence. Dans cette première partie, notre but est d'apporter des éclaircissements sur des notions telles que les systèmes réactifs, interactifs, la programmation réactive...

Nous parlerons des grands principes et des avantages de cette dernière, ainsi que des techniques permettant d'en bénéficier. Plus généralement, nous étudierons les concepts derrière les techniques que nous mettrons en pratique dans la deuxième partie de ce travail de fin d'études.

2 Chapitre II : Programmation réactive

2.1 Origines

Il y a dix ans, les interactions avec les pages web étaient limitées à soumettre un long formulaire au serveur et effectuer un simple rendu de la réponse du côté client. Les applications modernes ont évolué et possèdent une caractéristique temps-réel beaucoup plus importante. Par exemple, modifier un champ dans un formulaire peut automatiquement déclencher une sauvegarde en *back-end*, des *likes* sur un article sont directement visibles par les autres utilisateurs connectés... Les applications traitent une abondance d'événements en tout genre, offrant à l'utilisateur une expérience hautement interactive.

La programmation réactive se présente en tant qu'outil pour gérer cette complexité et cette dimension temps-réel efficacement.¹²

Nous verrons dans le [chapitre 3](#) que la programmation réactive possède de multiples implémentations, mais concentrons-nous d'abord sur ce qu'elle est réellement.

La plupart de la recherche sur la programmation réactive vient de *Fran*, un langage de programmation développé vers la fin des années 90 visant à faciliter la conception d'animations multimédias interactives.

Le premier article [Elliott and Hudak, 1997]¹³ introduit les notions de *behaviours* et d'*events* afin de représenter des valeurs de type continues et discrètes.

¹² "The introduction to Reactive Programming you've been ... - GitHub Gist."
<https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>. Accessed 29 Aug. 2018.

¹³ "Functional Reactive Animation - Conal Elliott." <http://conal.net/papers/icfp97/>. Accessed 1 Sep. 2018.

Behaviors ou valeurs *continues* ([FIGURE I](#)) : changent dans le temps, mais ont toujours une valeur. Ex. les coordonnées du pointeur d'une souris.

Events ou valeurs *discrètes* ([FIGURE II](#)) : événements qui se produisent périodiquement mais qui n'ont pas toujours une valeur. Ex. nous ne pouvons demander "quelle est la valeur actuelle d'un clic de souris ?".

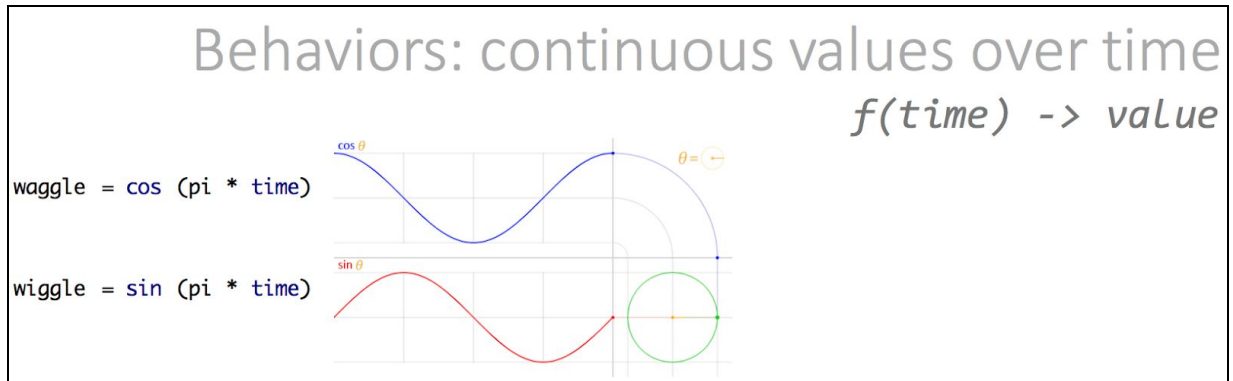


FIGURE I : Behaviors, continuous values¹⁴

Reactive Programming: origins & ecosystem. Jonas Chapuis, 2017

"Waggle" et "wiggle" désignent la valeur continue du déplacement horizontal et vertical d'une chose quand elle est oscillée ou secouée.

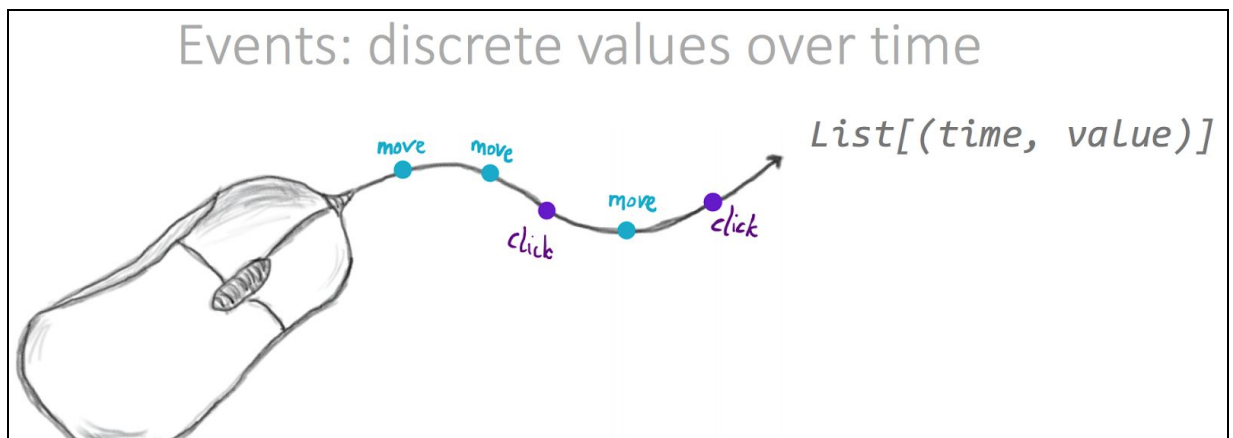


FIGURE II : Events, discrete values¹⁵

Reactive Programming: origins & ecosystem. Jonas Chapuis, 2017

L'action de cliquer sont des événements se produisant à un instant T . À chaque valeur est associée une indication de temps (time, value).

¹⁴ "Reactive programming: origins & ecosystem - Declarative programming."
<http://jonaschapis.com/wp-content/uploads/2017/09/ReactiveProgrammingOriginsAndEcosystem.pdf>. Accessed 1 Sep. 2018.

¹⁵ "Reactive programming: origins & ecosystem - Declarative programming."
<http://jonaschapis.com/wp-content/uploads/2017/09/ReactiveProgrammingOriginsAndEcosystem.pdf>. Accessed 1 Sep. 2018.

Le second texte [Wan and Hudak, 2000]¹⁶ étudie les applications de la programmation réactive au domaine de la robotique. Les auteurs classifient les robots comme étant de bons candidats à la programmation réactive. En effet, ils possèdent des notions *continues* (behaviors) comme la vitesse, l'orientation, la distance d'un mur mais également des notions *discrètes* (events) comme heurter un autre objet, recevoir un message ou atteindre un objectif.

Notons la similitude conceptuelle avec les applications actuelles qui travaillent également avec des valeurs variables dans le temps et des séquences ordonnées d'événements.

Nous constaterons que certains travaux parlent de programmation "fonctionnelle" réactive. Il s'agit juste de la combinaison de deux paradigmes de programmation.

Les deux propriétés principales de la programmation fonctionnelle sont la pureté et l'immuabilité. Une fonction pure signifie qu'elle ne modifie aucun état (ex. données en dehors de sa portée ou *scope*) et que son retour (*output*) dépend uniquement de ses paramètres (*input*). Une donnée immuable signifie que sa valeur ne peut être modifiée (ex. les variables se transforment en constantes). La programmation fonctionnelle repose également sur d'autres concepts tels que celui des fonctions d'ordre supérieur, qui signifie qu'une fonction peut être un paramètre ou une valeur de retour d'une autre fonction.^{17 18}

La plupart des langages actuels sont qualifiés de multi-paradigmes¹⁹, ou hybrides, et permettent aux programmeurs de travailler avec le style de programmation le plus adapté au problème rencontré. Il est donc très courant de retrouver un mélange de ces modèles de programmation dans une application web ou mobile typique.

C'est le cas de l'application *BdOccaz*, présentée au [chapitre 6](#), où nous combinerons programmation impérative, orientée objet, fonctionnelle, et évidemment, réactive. Dans ce travail, l'accent sera cependant mis sur la programmation réactive, étant donné sa pertinence dans le contexte des données temps-réel et l'étendue importante des sujets déjà couverts.

¹⁶ "Arrows, Robots, and Functional Reactive Programming - Springer Link."
https://link.springer.com/chapter/10.1007/978-3-540-44833-4_6. Accessed 1 Sep. 2018.

¹⁷ "What is 'Functional Programming'?" | alvinalexander.com."
<https://alvinalexander.com/scala/fp-book/what-is-functional-programming>. Accessed 1 Sep. 2018.

¹⁸ "Functional programming - Haskell Wiki - Haskell.org." 24 Dec. 2014,
https://wiki.haskell.org/Functional_programming. Accessed 1 Sep. 2018.

¹⁹ "The design of a multiparadigm programming language - Science Direct."
<https://www.sciencedirect.com/science/article/pii/0165607493900411>. Accessed 1 Sep. 2018.

Le populaire patron de conception *MVC (Model-View-Controller)*²⁴ constitue un bon exemple. Le *Model* désigne la représentation des données de notre système de stockage (ex. base de données), la *View* et le *Controller* se chargent d’afficher, de modifier ces données et de les enregistrer à nouveau.

La question suivante est alors soulevée : est-il, toujours et en toutes circonstances, adapté de structurer nos applications de cette façon ?

Il est courant en programmation impérative de demander explicitement la récupération d’un ensemble de données. Cette approche se veut entièrement valide et appropriée dans certains cas, mais essayons à présent de voir une application en tant qu’ensemble d’*unités de traitement* qui ont chacune pour objectif d’effectuer une opération sur les données qui y circulent continuellement.

Nous pourrions alors organiser ces *unités de traitement en pipeline*²⁵ (FIGURE IV). Un *pipeline*, par analogie à un *pipeline* physique, consiste en une chaîne d’éléments de traitement (processus, composants, fonctions, etc), arrangée de telle sorte que la valeur de sortie de chaque élément est passée au suivant. Dans notre cas, les éléments suivants *écoutent* l’élément précédent de la chaîne afin de recevoir les données.



FIGURE IV : Geomajas pipeline architecture²⁶

Prenons comme exemple le modèle de l’application *Twitter*. Plutôt que de récupérer les tweets à chaque demande explicite, l’application écouterait les tweets pertinents, recevrait les changements liés à ceux-ci pour enfin *réagir* en conséquence (effectuer une opération comme mettre à jour les données affichées...). C’est dans cette même ligne d’idées que se profile la programmation réactive.

En tant que paradigme, la programmation réactive n’apporte pas nécessairement de principes fondamentalement nouveaux. Elle lie plutôt un ensemble de pratiques, mettant l’accent sur certains éléments clés, ce qui nous amène à repenser la façon dont nous modélisons nos applications. Nous verrons dans le [chapitre 3](#) qu’au fil du temps, des

²⁴ "MVC Framework Introduction - Tutorialspoint."

https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm. Accessed 2 Sep. 2018.

²⁵ "Pipelines in Computing and Software Engineering - JavaBrahman." 14 Oct. 2015,

<https://www.javabrahman.com/programming-principles/pipelines-in-computing-and-software-engineering/>. Accessed 2 Sep. 2018.

²⁶ "3.2. Pipelines - Geomajas."

<http://files.geomajas.org/maven/1.8.0/geomajas/devuserguide/html/arch-pipeline.html>. Accessed 2 Sep. 2018.

abstractions (ex. librairies) aujourd'hui largement employées comme les *Reactive Extensions*²⁷ sont venues s'ajouter aux concepts inhérents à la programmation réactive.

Notons finalement que le modèle réactif se veut abouti, utilisé par de grands acteurs tels que *Facebook*, *Netflix* ou encore *Microsoft*.^{28, 29}

²⁷ "ReactiveX.io." <http://reactivex.io/>. Accessed 2 Sep. 2018.

²⁸ "Reactive Programming at Netflix – Netflix TechBlog – Medium." 16 Jan. 2013, <http://techblog.netflix.com/2013/01/reactive-programming-at-netflix.html>. Accessed 2 Sep. 2018.

²⁹ "Reactive Programming - Facebook for Developers." <https://developers.facebook.com/docs/camera-effects/docs/reactive-programming/>. Accessed 2 Sep. 2018.

2.2 Définitions

Programmes réactifs et interactifs

Penchons-nous d'abord sur l'essence d'un programme réactif. En 1991, *G. Berry* fournit une définition perspicace en mettant en relation les programmes réactifs avec leurs homologues, dits interactifs :

*“Interactive programs interact at their own speed with users or with other programs; from a user point of view, a time-sharing system is interactive.
Reactive programs also maintain a continuous interaction with their environment, but at a speed which is determined by the environment, not by the program itself.”*³⁰

Les programmes interactifs concrétisent l'idée d'un modèle de calcul basé sur la méthode *pull*, où le programme a le contrôle de la vitesse à laquelle les données seront demandées et traitées. Un exemple parfait de programme interactif est une structure telle qu'une boucle itérant sur une collection de données : le programme contrôle la vitesse à laquelle les données sont extraites de la collection et demandera l'élément suivant seulement après avoir fini de manipuler l'actuel.

Les programmes réactifs, au contraire, incarnent un modèle de calcul reposant sur la méthode *push*, basée sur les événements, où la vitesse à laquelle le programme interagit avec l'environnement est déterminée par l'environnement plutôt que par le programme lui-même. En d'autres termes, le rôle du programme est celui d'un observateur qui réagira à la réception d'événements. Des exemples de tels systèmes sont des applications avec une interface graphique traitant de divers événements provenant d'une entrée d'utilisateur (ex. clics de souris, appuis sur les boutons du clavier), programmes traitant des marchés boursiers, médias sociaux...³¹

C'est sur ces derniers que se penche notre travail de fin d'études.

Lien entre données temps-réel et réactivité

Ce lien entre les notions de données temps-réel et de réactivité dont nous avons déjà parlé au point [2.1 Origines](#) est également établi par *G. Berry* dans ce même article.

Il mentionne que les programmes en temps réel sont habituellement réactifs mais que dans certains cas (processus lents), les programmes écrits sans attention particulière (ex. non-réactifs) peuvent fonctionner en temps-réel. Il précise ensuite que cela représente une minorité des cas, les contraintes de temps nécessitant de prendre soin de tous les détails du code exécuté.

³⁰ "Real time programming : special purpose or general purpose ... - Inria."
<http://www-sop.inria.fr/members/Gerard.Berry/Papers/Berry-IFIP-89.pdf>. Accessed 3 Sep. 2018.

³¹ "The Essence of Reactive Programming - TU Delft Repositories."
<https://repository.tudelft.nl/islandora/object/uuid:bd900036-40f4-432d-bfab-425cdebc466e/datastream/OBJ/download>. Accessed 3 Sep. 2018.

Programmation réactive

- (1) “Reactive programming is a programming paradigm that is built around the notion of continuous time-varying values and propagation of change.”³²
- (2) “Reactive Programming (RP) is a recent programming paradigm that supports the development of reactive applications through dedicated language abstractions. It is based on concepts like time-varying values (a.k.a. signals or behaviors), events streams to model discrete updates, automatic tracking of dependencies, and automated propagation of change.”³³

Ces deux définitions ont retenu notre attention, chacune proposant un niveau de détail différent. L'article de la *Vrije Universiteit Brussel* (1) de 2012 se base sur les concepts clés de valeurs changeant dans le temps et de propagation du changement. Ils exposent également cet exemple :

```
var1 = 1
var2 = 2
var3 = var1 + var2
```

En programmation impérative séquentielle classique, la valeur de la variable `var3` contiendra toujours 3, qui est la somme des valeurs initiales des variables `var1` et `var2` même si on attribue ultérieurement une nouvelle valeur à `var1` ou `var2` (sauf si le programmeur assigne explicitement une nouvelle valeur à la variable `var3`).

En programmation réactive, la valeur de la variable `var3` est automatiquement recalculée au fil du temps lorsque la valeur de `var1` ou `var2` change. Les valeurs changent avec le temps et lorsqu'elles changent, tous les calculs dépendants sont ré-exécutés.

Le travail *Reactive Programming: A Walkthrough* (2) réalisé en 2015 met plus précisément en avant les notions d'abstractions de langage, de *behaviors* et d'*events*.

*Ben Lesh*³⁴ approche la programmation réactive de manière plus informelle dans un récent podcast³⁵ :

Nous parlons de programmation réactive dès qu'un composant³⁶ d'application réagit à l'arrivée de données suite au déclenchement d'un événement, les traitant et les passant via un évènement au prochain composant.

Cette description (paraphrasée) nous semble également pertinente de par sa simplicité.

³² "A Survey on Reactive Programming - Software Languages Lab - VUB."
<http://soft.vub.ac.be/Publications/2012/vub-soft-tr-12-13.pdf>. Accessed 3 Sep. 2018.

³³ "(PDF) Reactive Programming: A Walkthrough - ResearchGate."
https://www.researchgate.net/publication/308807238_Reactive_Programming_A_Walkthrough. Accessed 3 Sep. 2018.

³⁴ Senior Software Engineer chez Google menant le projet RxJS (bibliothèque Reactive Extensions en JavaScript)

³⁵ "VoV 020: Reactive Programming With Vue With Tracy Lee ... - Player FM." 17 Jul. 2018,
<https://player.fm/series/views-on-vue/vov-020-reactive-programming-with-vue-with-tracy-lee>. Accessed 3 Sep. 2018.

³⁶ Tout morceau de logique : classe, ensemble de fonctions...

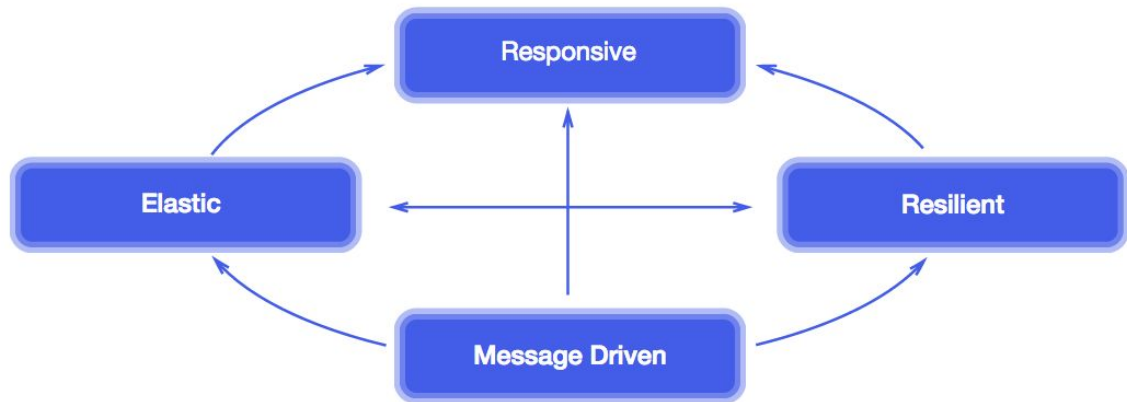
Sur base de ces différents éléments, tentons à notre tour d'apporter une définition que nous voudrions simple et représentative :

DÉFINITION

La programmation réactive est un paradigme de programmation facilitant la gestion des données variant dans le temps et la propagation de ces changements au moyen d'événements.

2.3 Propriétés

Selon le Reactive Manifesto³⁷, voici les quatre grands principes d'un système réactif :



Responsive : Le système doit être rapide et fournir des informations utiles aux interactions des utilisateurs et aux conditions d'erreur.

Resilient : Un système résilient tombe rarement en panne et reste réactif même après un échec.

Elastic : Ayant la capacité de gérer une quantité variable de travail.

Message Driven : Dans un système réactif, des messages ou notifications sont transmis entre les composants lorsqu'un événement se produit. Ils contiennent des informations utiles comme des nouvelles données, une erreur...

³⁷ "The Reactive Manifesto." <https://www.reactivemanifesto.org/>. Accessed 10 Sep. 2018.

3 Chapitre III : Architecture générale

*“Architecture is the fundamental **organisation** of a system, embodied in its **components**, their **relationship** to each other and the environment, and the **principles** governing its **design** and **evolution**.”³⁸*

Nous avons dans le chapitre 2 posé les bases de la programmation réactive, positionnant la notion d'événements en tant que pierre angulaire. Nous allons à présent approfondir ce mécanisme et décrire différentes techniques pour modéliser une application de manière réactive en utilisant les événements.

Ce chapitre traite d'architecture applicative, nous ferons donc abstraction du fonctionnement interne de chaque partie³⁹ individuelle pour mieux nous concentrer sur la manière de les interconnecter.

Nous utiliserons également les termes “module” et “composant” pour désigner une partie d'application.

3.1 Systèmes basés sur les événements

Note : En supplément des différentes sources référencées, les explications ci-après sont inspirées de l'ouvrage « *Event-Based Programming : Events to the Limit* » de *Ted Faison* (le lien se trouve au point [9 Bibliographie](#)).

3.1.1 Origines

Les événements existent depuis le début des années 80, le patron de conception *Model-View-Controller* de *Smalltalk* étant l'un des premiers à les utiliser afin de garder différentes parties d'un système synchronisées. Les systèmes d'exploitation munis d'une interface graphique tels que *Microsoft Windows* reposent également sur ce modèle.

Autrefois, le flux d'exécution d'un programme était souvent déterminé depuis son environnement interne, plus précisément sa routine principale (pensons à la fonction *main*

³⁸ ANSI/IEEE Std 1471-2000

³⁹ Tout morceau de logique : classe, ensemble de fonctions, composant...

d'une application traditionnelle⁴⁰). Un programme était simplement exécuté puis se terminait de lui-même. Cette pratique était très courante durant les débuts de l'informatique et est encore utilisée, par exemple pour les programmes pilotés par lignes de commande. Tous les paramètres sont alors définis à l'avance et passés lorsque le programme démarre. Des sous-routines sont invoquées à partir de la routine principale et le programmeur est responsable de gérer explicitement le flux d'exécution du programme. Nous parlons ici de programmation procédurale⁴¹, un type de programmation impérative.

Dans une application moderne typique, basée sur les événements, ce flux global n'est pas perceptible par le programmeur. En effet, la routine principale se présente souvent sous la forme de boucle d'événements, ou *event-loop*, généralement fournie par les bibliothèques de la plateforme utilisée. Son rôle est d'attendre qu'un événement se produise pour le gérer de manière appropriée grâce à son répartiteur d'événements ou *event-dispatcher*. Ce mécanisme est illustré de manière simplifiée dans le point [FIGURE V](#).

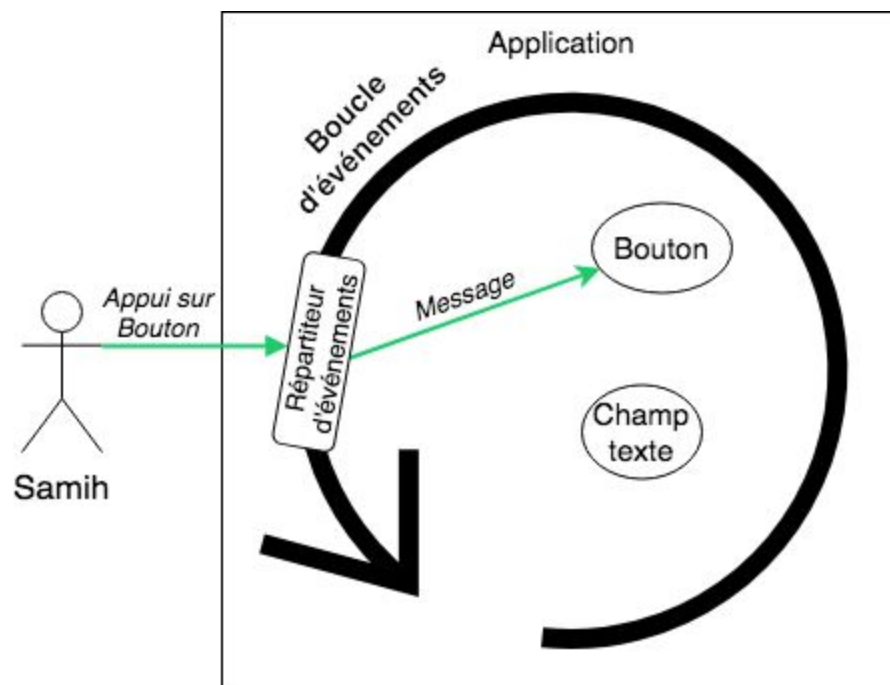


FIGURE V : Boucle d'événements ou *event-loop*
Inspiré de "The Android Event Loop"⁴². Samih Alkeilani, 2018

Le changement d'horizon de la programmation procédurale à la programmation événementielle a été accéléré par l'introduction des interfaces graphiques, largement adoptées dans les systèmes d'exploitation et les applications utilisateur.⁴³

⁴⁰ "public static void main(String[] args) - Java main method - JournalDev."
<https://www.journaldev.com/12552/public-static-void-main-string-args-java-main-method>. Accessed 5 Sep. 2018.

⁴¹ "Use of procedural programming languages for controlling production"
<http://ieeexplore.ieee.org/document/120848/>. Accessed 5 Sep. 2018.

⁴² "The Android Event Loop - mattias - Niklewski." 15 Sep. 2012,
http://mattias.niklewski.com/2012/09/android_event_loop.html. Accessed 7 Sep. 2018.

⁴³ "Event Driven Programming - TechnologyUK."
<http://www.technologyuk.net/software-development/designing-software/event-driven-programming.shtml>. Accessed 5 Sep. 2018.

3.1.2 Notions

Un événement désigne une action reconnue et gérée par un programme. Il peut être déclenché par le système, l'utilisateur ou d'autres sources.⁴⁴ Citons par exemple la fin d'une tâche par un logiciel, des frappes sur le clavier par un utilisateur, une minuterie...

La programmation événementielle est un paradigme dans lequel le flux d'un programme est déterminé par des événements⁴⁵. Nous pensons qu'il est raisonnable d'affirmer que toutes les applications graphiques actuelles bénéficient de la programmation événementielle.

Un système est dit basé sur les événements ou *event-driven* quand ses différents composants interagissent principalement au moyen de notifications d'événement. Nous élaborerons une telle architecture dans le point [5.3 Architecture réactive envisagée](#).

Les notifications d'événement sont des signaux envoyés d'un composant à un autre pour le prévenir qu'un événement a été déclenché. Le but d'un tel processus est de permettre aux différentes parties de l'application de réagir à la réception d'une notification d'événement afin de, par exemple, rester synchronisées.

Dans la suite de ce travail, nous choisirons par souci de clarté de remplacer le terme "notification d'événement" par "notification".

3.1.3 Avantages et inconvénients

La première question soulevée est : en quoi un système basé sur les événements se veut-il utile ? Pour y répondre, commençons par préciser que l'idée derrière une telle approche est de faire communiquer plusieurs parties d'application entre elles au moyen de notifications, mais également de les rendre indépendantes.

Le problème adressé ici est bien connu dans le monde du logiciel, il s'agit du couplage ou *coupling*. Le couplage est le degré d'interdépendance entre des composants, ou la force de la relation entre deux modules⁴⁶. Quand une application est développée, changer une partie étroitement couplée peut provoquer un dysfonctionnement dans d'autres parties, voire dans le reste du système. Les programmeurs travaillant sur des modules couplés doivent synchroniser leurs activités, et la maintenance du programme peut également s'avérer compliquée étant donné que tout changement peut provoquer des conséquences inattendues.

Une approche orientée événements réduit la complexité globale du programme, rendant chaque partie individuellement plus simple. En contrepartie, leurs opérations peuvent devenir plus difficile à comprendre sans voir le reste du système. En effet, la réduction du couplage implique que le composant expéditeur d'une notification ne connaisse pas directement le(s) destinataire(s). Il n'est donc pas possible de savoir vers quel(s) composant(s) sont envoyées les notifications en scrutant la logique de l'expéditeur.

⁴⁴ "What is an Event? Webopedia Definition." <https://www.webopedia.com/TERM/E/event.html>. Accessed 5 Sep. 2018.

⁴⁵ "Event-driven programming" <http://www.technologyuk.net/software-development/designing-software/event-driven-programming.shtml>

⁴⁶ "ISO/IEC/IEEE 24765:2010 - Systems and software engineering" <https://www.iso.org/standard/50518.html>. Accessed 6 Sep. 2018.

Illustrons à l'aide d'un exemple ce compromis et les différences entre les systèmes orientés événement et le modèle requête/réponse⁴⁷ habituel.

3.1.4 Comparaison avec le modèle Requête/Réponse

Note : En supplément des différentes sources référencées, les explications ci-après sont inspirées de la conférence « *GOTO 2017 - The Many Meanings Of Event-Driven Architecture* » de *Martin Fowler* (le lien se trouve au point [9 Bibliographie](#)).

Imaginons une société d'assurances habitation, voir [FIGURE VI](#). Les clients ont la possibilité de changer leur adresse, ce qui affecte le prix de leur assurance. Lors d'un changement d'adresse :

- Le service *Customer Management* prévient le service *Insurance Quoting* afin qu'il émette un nouveau devis.
- (Le service *Insurance Quoting* envoie un email au client avec le nouveau devis.)

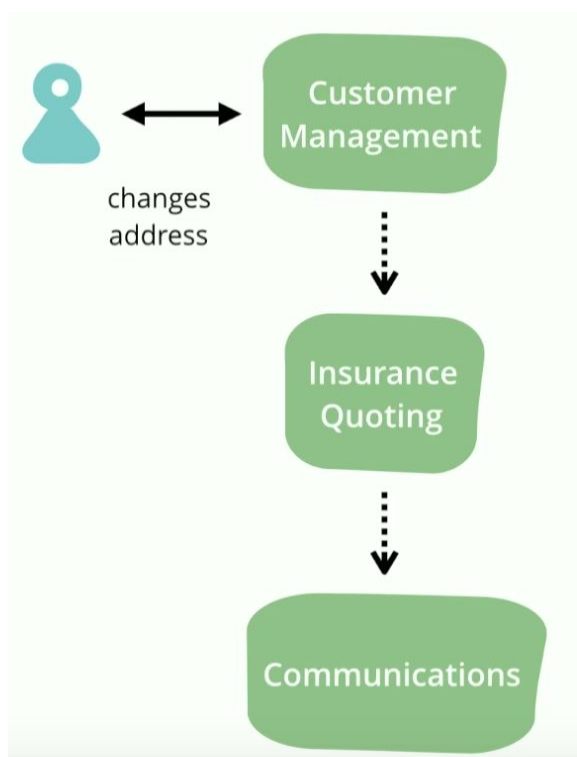


FIGURE VI : Assurance habitation, requête/réponse
The Many Meanings Of Event-Driven Architecture. Martin Fowler, 2017

En utilisant un modèle requête/réponse, le service *Customer Management* invoque ici de la logique appartenant au service *Insurance Quoting* (ex. appelle une de ses fonctions de mise à jour) pour le prévenir d'un changement d'adresse. Ceci introduit la notion de couplage

⁴⁷ "Evented Systems vs Request-Response Systems - Pico Labs." 10 Jul. 2017, <https://picolabs.atlassian.net/wiki/spaces/docs/pages/1189842/Evented+Systems+vs+Request-Response+Systems>. Accessed 6 Sep. 2018.

précédemment mentionnée, car nous importons la connaissance du fonctionnement du service *Insurance Quoting* au sein du service *Customer Management* (ex. pour qu'il puisse appeler une de ses fonctions).

La [FIGURE VII](#) montre comment approcher ce problème grâce aux événements. Le service *Customer Management* ne connaît plus le service *Insurance Quoting* car la relation de dépendance est maintenant inversée.

- (1) Le service *Insurance Quoting* prévient qu'il est intéressé par l'événement *address changed* : c'est à présent lui qui connaît *Customer Management* (car il souscrit à son événement), et plus l'inverse.
- (2) Quand l'adresse du client change, le service *Customer Management* émet une notification d'événement et le service *Insurance Quoting* décide s'il veut y réagir.

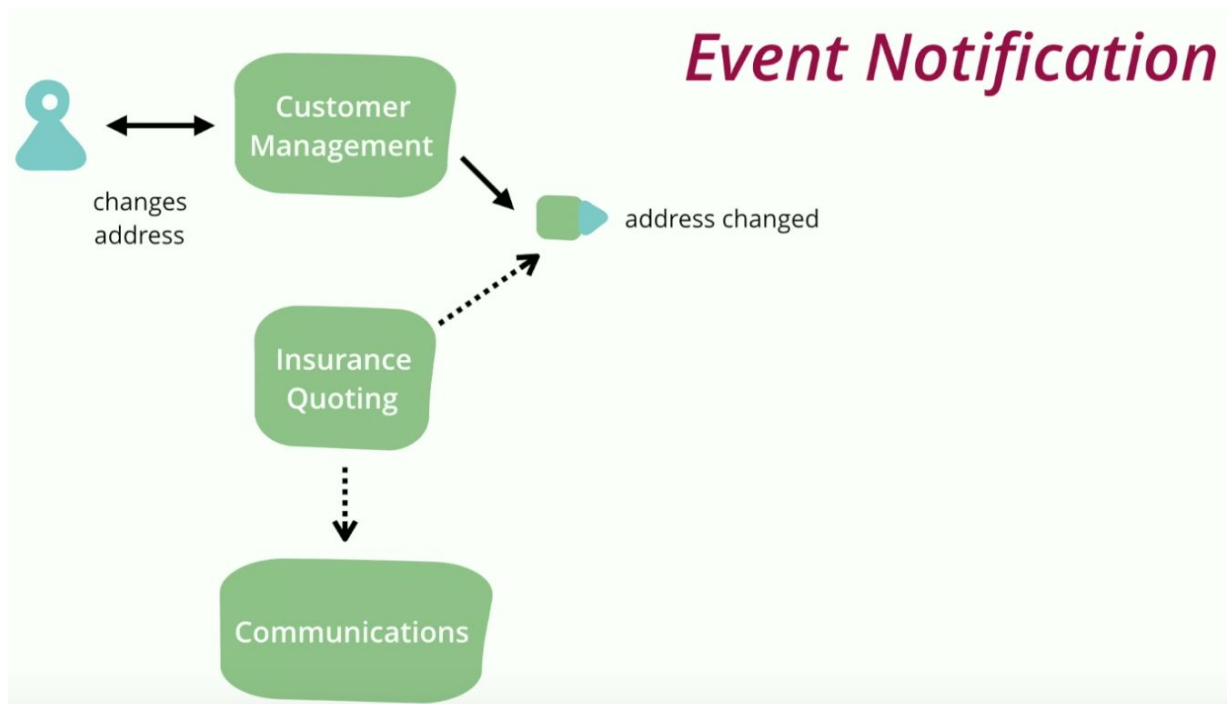


FIGURE VII : Assurance habitation, événement

The Many Meanings Of Event-Driven Architecture. Martin Fowler, 2017

Nous utilisons donc ici les événements comme mécanisme de notification entre plusieurs services. Comme déjà indiqué, il constitue aussi une façon classique de gérer les interactions entre les interfaces graphiques et le reste de la logique d'une application.

Par exemple, quand un utilisateur entre une information dans un *champ de formulaire*, cela a-t-il du sens que notre composant *champ de formulaire* (très générique) connaisse le reste nos services (ex. *Customer Management*, *Insurance Quoting*...) et qu'il invoque leur logique ?

Cela peut en tout cas rapidement augmenter la complexité de notre *champ de formulaire*, le rendant plus difficile à maintenir.

Au même titre que dans la [FIGURE VII](#), nous préférons que notre composant *champ de formulaire* émette une notification d'événement lorsque le texte change, en laissant la possibilité à nos services de signaler qu'ils sont intéressés par ce dernier et d'y réagir. Il suffit alors que chacun de nos services possède une certaine connaissance de notre *champ de formulaire* afin de réagir à ses notifications.

Nous avons décrit de manière générale le fonctionnement d'un système basé sur les événements, mais notons qu'il existe plusieurs manières de le mettre en place dans nos applications. Nous allons à présent exposer deux grands types de systèmes basés sur les événements.

3.1.5 Systèmes à envoi direct

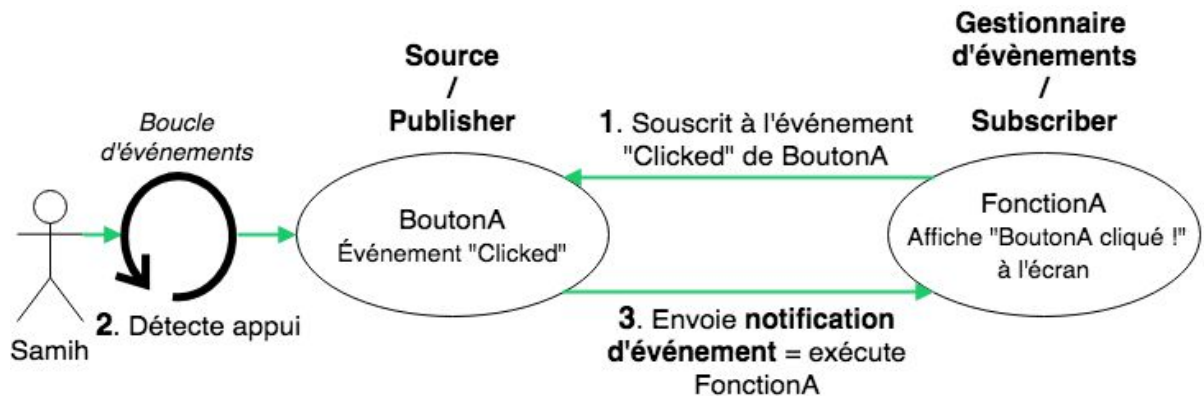


FIGURE VIII : Systèmes à envoi direct

Inspiré de "[Event-Based Programming : Events to the Limit](#)". Samih Alkeilani, 2018

L'idée est que, comme illustré dans le point [FIGURE VIII](#), la source de l'événement (ex. bouton) gère *directement* la liste de gestionnaires d'événements (ex. liste de fonctions). Quand l'événement est déclenché, la source de l'événement le détecte et envoie une notification aux gestionnaires d'événements qui y sont souscrits. Voici les différents intervenants :

Source ou *publisher*

Expose l'événement et envoie une *notification* ([FIGURE VIII](#) ; 3) quand ce dernier a eu lieu ([FIGURE VIII](#) ; 2). La source possède et notifie une liste de gestionnaires d'événements.

- Nous pouvons ici voir l'expression *envoyer une notification* comme synonyme d'*exécuter* la fonction souscrite ([FIGURE VIII](#) ; 1).
- Si la liste de gestionnaires d'événements est vide (aucun gestionnaire n'est enregistré), aucune notification ne sera envoyée.

Gestionnaire d'événement ou *subscriber*

Typiquement, il s'agit d'une fonction (ou d'une suite d'instructions) souscrite ([FIGURE VIII](#) ; 1) à l'événement et exécutée en réponse à une notification d'événement reçue ([FIGURE VIII](#) ; 3).

Notification d'événement

Ensemble de données utiles concernant un événement, envoyé aux différents gestionnaires d'événements ([FIGURE VIII](#) ; 3).

- Nous pouvons ici voir les données utiles comme des valeurs passées en paramètres de la fonction exécutée.

3.1.6 Systèmes à envoi indirect

*“Indirect communication is defined as communication between entities (...) through an **intermediary** with **no direct coupling** between the sender and the receiver(s).”*⁴⁸

Les systèmes à envoi indirect impliquent que les gestionnaires d'événements (ex. fonction(s) affichant “bouton cliqué !”) ne souscrivent plus *directement* à la source de l'événement (ex. bouton) mais passent par un module intermédiaire. Ce dernier agit en tant que source (ex. remplace bouton) pour les gestionnaires d'événements, les gère et les notifie lorsqu'un événement survient.

Étant donné qu'il n'existe plus de connexion entre la source et les gestionnaires d'événements, ces derniers utiliseront un modèle de souscription différent. Ils pourront exprimer au module intermédiaire leur intérêt pour certaines notifications en fonction de leur sujet (*topic-based model*), contenu (*content-based model*) ou encore leur type (*type-based model*).⁴⁹

Nous pouvons voir le module intermédiaire comme une couche d'abstraction qui permet d'implémenter de la logique lors du traitement des notifications qu'il reçoit avant de les transférer aux différents gestionnaires. Il utilisera typiquement un mécanisme de tampon et délivrera les notifications de manière asynchrone, de sorte à ce que la vraie source (ex. bouton) ne soit alors pas affectée par la durée du traitement des gestionnaires d'événements.

Ce fonctionnement offre une *scalability* supérieure, ou la capacité d'un produit à s'adapter à une montée en charge, en particulier sa capacité à maintenir ses fonctionnalités et ses performances en cas de forte demande (ex. nombre de notifications simultanées).⁵⁰ Il faut toutefois évaluer, en fonction du cas d'utilisation, si la propriété asynchrone est adéquate.

L'avantage et le désavantage des systèmes à envoi indirect relèvent tous deux du découplage entre la source de l'événement et les gestionnaires d'événements.

Nous avons mentionné que deux modules découplés pouvaient facilement travailler de manière indépendante. Cependant, dans ce cas, il faut veiller à ce que le système puisse assurer la livraison des notifications aux gestionnaires d'événements. En effet, une faille dans le module intermédiaire pourrait compromettre l'envoi de notifications.

La différence entre les deux types de systèmes est schématisée au point [FIGURE IX](#).

⁴⁸ "Indirect communication."

http://www.inf.fu-berlin.de/inst/ag-se/teaching/V-NETZPR-2015/07_Indirect_Communication%20-%20I.pdf. Accessed 10 Sep. 2018.

⁴⁹ "Content-based Publish/Subscribe Systems - Semantic Scholar."

<https://pdfs.semanticscholar.org/4221/21c6edcfd7445cfc2d447c55e8e87115cd4.pdf>. Accessed 8 Sep. 2018.

⁵⁰ "Characteristics of scalability and their impact on ... - ACM Digital Library."

<https://dl.acm.org/citation.cfm?id=350432>. Accessed 9 Sep. 2018.

3.1.7 Comparaison

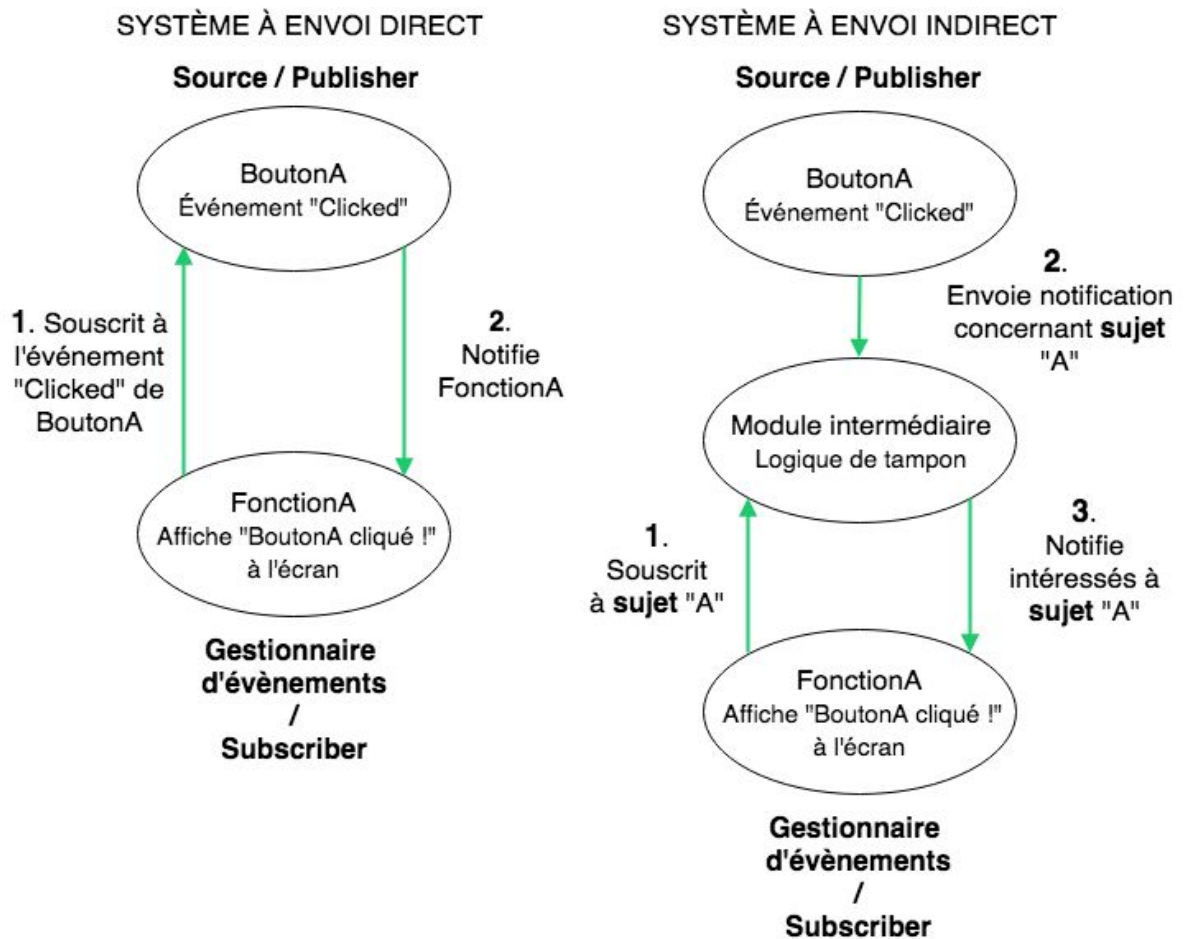


FIGURE IX : Systèmes à envoi direct et indirect

Inspiré de "[Event-Based Programming : Events to the Limit](#)" & "[JMS - Publish/Subscribe messaging example](#)⁵¹". Samih Alkeilani, 2018

La grande majorité des systèmes à envoi indirect sont connus sous le nom *Publish-subscribe* et sont également utilisés dans des réseaux distribués à grande échelle. Ces systèmes quittent malheureusement le cadre ce travail de fin d'études.

Notre travail se concentre sur les systèmes à envoi direct, plus particulièrement le patron de conception *Observer*. Sa simplicité et son caractère synchrone poussent la plupart des langages de programmation à l'implémenter dans leur système de gestion d'événements.

⁵¹ "JMS - Publish/Subscribe messaging example using ActiveMQ and" 26 Nov. 2014, <https://www.codenotfound.com/jms-publish-subscribe-messaging-example-activemq-maven.html>. Accessed 8 Sep. 2018.

3.2 Le patron de conception *Observer*

Nous allons à présent exposer le patron de conception *Observer* qui se veut être le système à envoi direct le plus connu. Soulignons que le patron de conception *Observer* est une application du mécanisme que nous avons vu dans le point [3.1.5 Systèmes à envoi direct](#) : les principes sont identiques, seul le vocabulaire diffère.

Ce patron de conception est couramment utilisé dans les langages de programmation modernes (*C#*, *JavaScript*, *Java*...) et est également intégré dans une multitude de frameworks et de bibliothèques connus comme *.NET*.

L'idée principale est qu'un composant (subject) maintient une liste d'abonnés (observers) qu'il notifie de tout changement d'état. En analogie avec [3.1 Systèmes basés sur les événements](#), les abonnés (observers) sont les gestionnaires d'événements et le subject correspond à la source de l'événement.

3.2.1 Cas concret

Laissons la [FIGURE X](#) clarifier ce concept. Chaque enchérisseur (observer) possède une palette numérotée utilisée pour indiquer son offre et observe l'évolution du prix de l'enchère. Le commissaire-priseur (subject) commence l'enchère et lorsqu'il accepte une offre, il modifie le prix de l'enchère et *notifie* tous les enchérisseurs (observers) du changement.

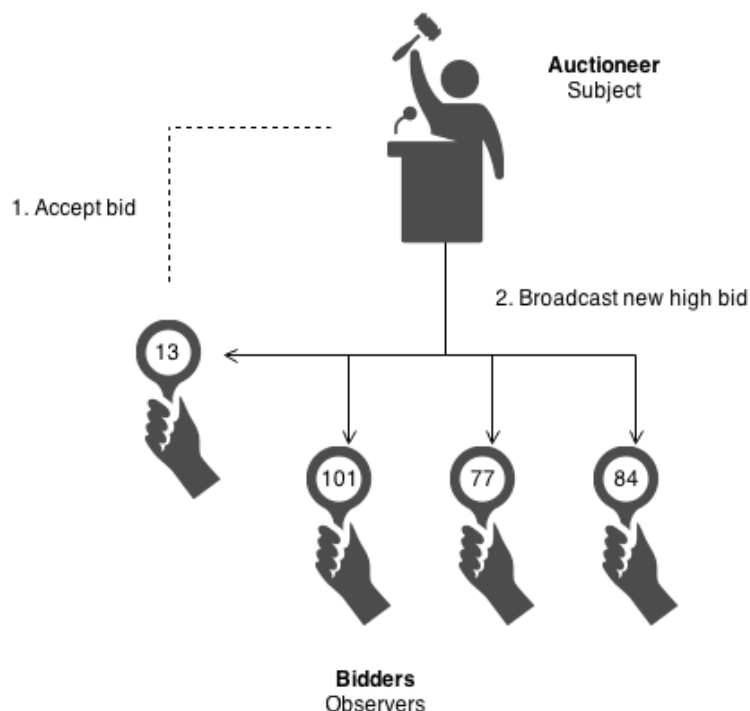


FIGURE X : Observer Design Pattern Example⁵²
SourceMaking, 2018

⁵² "Observer Design Pattern - SourceMaking." https://sourcemaking.com/design_patterns/observer. Accessed 9 Sep. 2018.

3.2.2 Fonctionnement

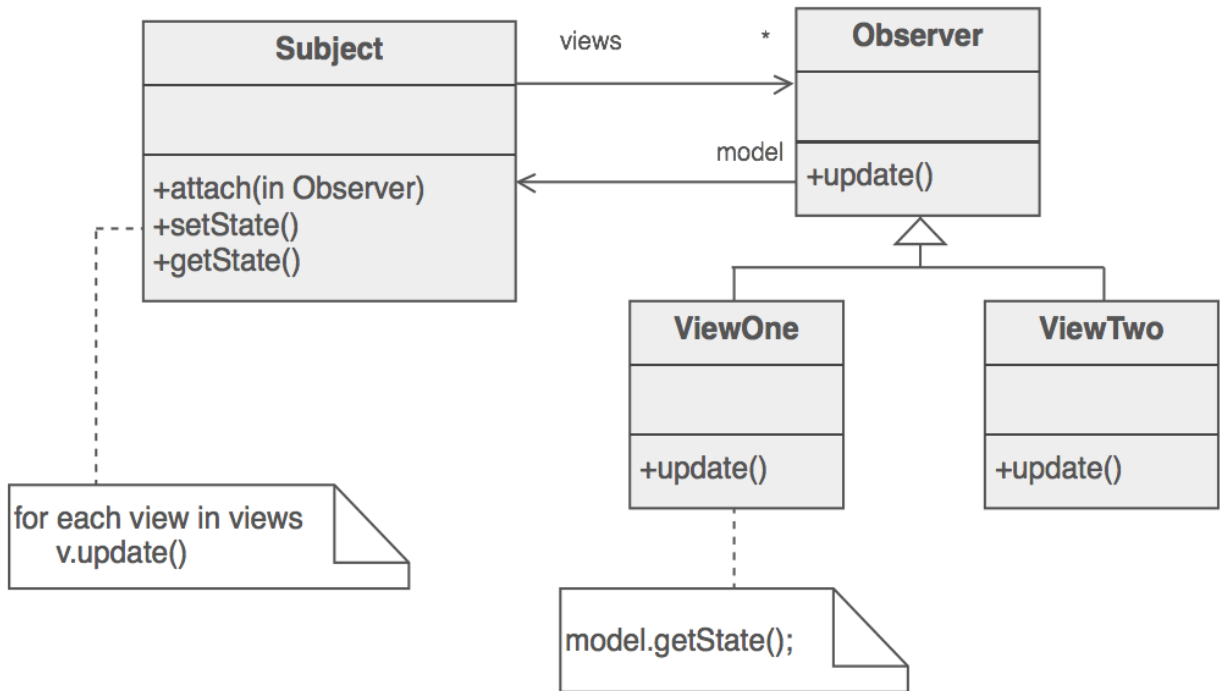


FIGURE XI : Observer Design Pattern Structure⁵³
SourceMaking, 2018

Le Subject (model) offre aux observers (Views) la possibilité de s'enregistrer à ses changements grâce à la fonction `attach(in Observer)`.

Quand un changement d'état a lieu via `setState()`, le Subject (model) notifie tous ses observers (Views) en appelant leur fonction `update()`.

Au sein de la fonction `update()`, l'observer (View) réagit au changement. Il peut alors récupérer le nouvel état du Subject (model) grâce à la fonction `getState()` du Subject et l'utiliser à son tour pour se mettre à jour.

Quand l'observer récupère le nouvel état du subject, nous avons à faire à un modèle d'interaction nommé *pull*, où les observers sont responsables de récupérer les données qui les intéressent après un changement.

Avec le modèle *push*, le subject inclut les données ayant changé dans la notification qu'il envoie à ses observers, ces derniers n'ayant alors pas besoin de les récupérer ensuite. Concrètement, dans la [FIGURE XI](#), le modèle *push* reviendrait à ajouter un paramètre dans la fonction utilisée pour notifier les observers : `update(in DataChanged)`.

⁵³ "Observer Design Pattern - SourceMaking." https://sourcemaking.com/design_patterns/observer. Accessed 9 Sep. 2018.

3.2.3 Propriétés

Synchrone

Dans patron de conception *Observer* le subject envoie les notifications de manière synchrone à ses observers. Autrement dit, la fonction `update()` (FIGURE XI) est appelée pour chaque observer, l'une après l'autre, dans l'ordre de leur souscription.

Analysons ce comportement à la FIGURE XII où nous pouvons voir que les observers sont simplement enregistrés dans une collection membre du subject.

Lorsqu'un changement se produit, une itération se produit sur la collection `observerCollection` et la fonction `update()` est appelée pour chaque observer (`notifyObservers()`).

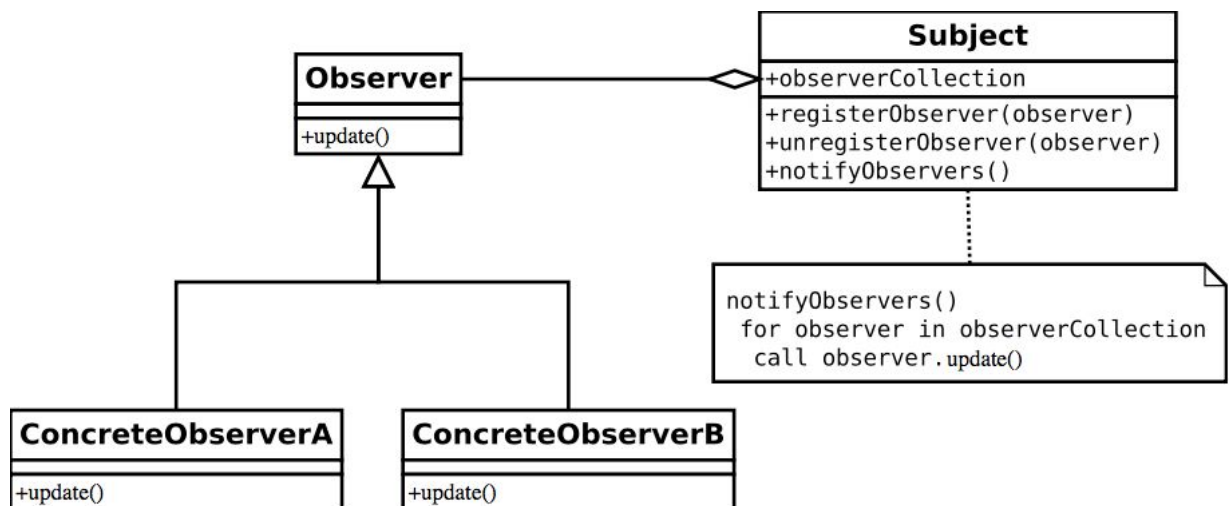


FIGURE XII : Observer Design Pattern⁵⁴
Peer-reviewed answer, 2017

⁵⁴ "The observer pattern is two modules or is one module?" <https://stackoverflow.com/a/47697564>. Accessed 9 Sep. 2018.

Lapsed listener problem

Apercevons-nous à la [FIGURE XII](#) qu'au sein de la classe `Subject` se trouve une fonction `unregisterObserver(observer)`. Il est essentiel que le `Subject` propose à ses `Observers` un moyen de se désabonner des changements d'états auxquels ils ont souscrit au préalable.

Si le `Subject` ne permet pas ce mécanisme, des fuites de mémoire ou *memory leaks* peuvent avoir lieu. Dans le cadre du patron de conception *Observer*, ce problème est appelé *lapsed listener problem*. Ce dernier se produit quand un `Observer` ne parvient pas à se désabonner des notifications dont il n'a plus besoin - le `Subject` maintient alors, tout au long de sa durée de vie, une référence vers l'`Observer`.

Ceci ne causera pas seulement des *memory leaks* mais également une dégradation de performances, car l'`Observer` n'étant plus intéressé par les notifications les recevra toujours et le contenu de sa fonction `update()` sera exécuté.

Un moyen courant de mettre en place ce processus constitue en l'implémentation du patron de conception *Dispose*. Il permet au développeur d'explicitement libérer les ressources non-gérées (*unmanaged resources*) par le *Garbage Collector* en appelant la fonction `dispose()` sur ces dernières.⁵⁵ Pensons notamment aux fichiers ouverts, connexions réseau/base de données ouvertes...

Cette nécessité dans notre cas s'explique par le fait que le `Subject` maintient des références *fortes* envers les `Observers`, ce qui empêche le *Garbage Collector* de les collecter même s'ils deviennent `null` (car ils sont toujours dans la liste du `Subject`).

Une autre façon de pallier au problème est de faire en sorte que le `Subject` englobent ses `Observers` avec des références *faibles*. Imaginons une référence faible comme le conteneur de l'`Observer` (ou tout autre objet) qui permet de le désinscrire/supprimer de la liste du `Subject`, et ce même s'il est `null`. À l'envoi d'une notification, le `Subject` vérifie alors si l'`Observer` est `null` et le supprime de sa collection le cas échéant. Une fois désinscrit, le *Garbage Collector* peut alors faire son travail et le collecter.^{56, 57, 58, 59}

⁵⁵ "Cleaning Up Unmanaged Resources | Microsoft Docs." 29 Mar. 2017, <https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/unmanaged>. Accessed 10 Sep. 2018.

⁵⁶ "Andy Mc's .NET Framework FAQ - Andy McMullan." <http://www.andymcm.com/dotnetfaq.htm>. Accessed 10 Sep. 2018.

⁵⁷ "8.8. weakref — Weak references — Python 3.7.0 documentation." <https://docs.python.org/3/library/weakref.html>. Accessed 10 Sep. 2018.

⁵⁸ "Observer Pattern and Lapsed Listener Problem." 17 Apr. 2016, <http://ilkinulas.github.io/development/general/2016/04/17/observer-pattern.html>. Accessed 10 Sep. 2018.

⁵⁹ "Design Patterns: Observer and Publish-Subscribe - YouTube." 26 Jul. 2017, <https://www.youtube.com/watch?v=72bdaDI4KLM>. Accessed 10 Sep. 2018.

Typiquement, les patron de conception *Observer* et *Dispose* iront de pair et deux solutions se profilent :

- (1) La fonction `registerObserver(observer)` retourne la souscription de l'observer. Nous pouvons ensuite fonction `dispose()` quand nous désirons désabonner notre observer.
- (2) La fonction `registerObserver(observer)` ne retourne rien, la souscription reste au sein de notre subject. Pour la désinscription, nous appellerons la fonction `unregisterObserver(observer)` qui se chargera elle-même d'appeler la fonction `dispose()`.

3.2.4 Points clés

Le patron de conception *Observer* est principalement utilisé la gestion d'événements dans des [systèmes basés sur les événements](#).

Il est implémenté dans la plupart des langages modernes, comme par exemple le *delegate model* (ex. délégué `EventHandler`) dans le langage C#, permettant de gérer et de lever des événements.

“The delegate model follows the observer design pattern, which enables a subscriber to register with, and receive notifications from, a provider. An event sender pushes a notification that an event has happened, and an event receiver receives that notification and defines a response to it.”⁶⁰

La programmation réactive étant basée sur les événements, ces derniers eux-mêmes fondés sur le patron de conception *Observer*, nous serons très souvent amenés à l'utiliser et le mentionner tout au long de ce travail.

⁶⁰ "Handling and Raising Events | Microsoft Docs." 29 Mar. 2017, <https://docs.microsoft.com/en-us/dotnet/standard/events/>. Accessed 10 Sep. 2018.

3.3 Reactive Extensions

Les explications ci-dessous sont inspirées du site officiel des *Reactive Extensions*⁶¹.

3.3.1 Généralités

La librairie *Reactive Extensions*, ou *ReactiveX*, est de loin la plus populaire en programmation réactive - il est d'ailleurs courant d'entendre parler des *Reactive Extensions* pour désigner la programmation réactive et inversement. Elle est disponible dans une multitude de langages et *frameworks* tels que *Java*, *JavaScript*, *.NET*, *Scala*, *Swift* et d'autres.

Les *Reactive Extensions* étendent le patron de conception *Observer* vu [au point précédent](#). Nous y retrouvons donc les notions d'observers, d'événements et de subject. Pour rappel, lors d'un événement (ex. *clic* de souris), le subject envoie une notification aux observers qui y sont souscrits afin qu'ils puissent réagir à cet événement.

La librairie expose un nouveau type, nommé observable. Les observers souscriront la plupart du temps à un observable à la place d'un subject (malgré qu'il existe certaines différences entre les deux⁶²). Nous pouvons le voir en tant producteur de nouvelles données.

3.3.2 Observable

Un observable est un *wrapper* autour d'un *stream* de données, ou encore une *collection arrivant au fil du temps*⁶³.

Événements, requêtes asynchrones, animations peuvent être transformés en observables (voir [EXTRAIT DE CODE I](#)). Pour les événements qui se veulent le cas d'utilisation le plus courant, les [notifications d'événement](#) seront envoyées aux observers grâce à leur fonction `onNext()`, équivalente à la fonction `update()` vue au point [3.2.2](#).

- Un *stream* est une suite d'éléments mis à disposition au fil du temps. Imaginons la lecture d'un *stream* comme des articles sur un tapis roulant, étant traités un à la fois^{64,65}. Notons que cette idée rejoint l'approche discutée [plus tôt](#) où nous envisagions une « application en tant qu'ensemble d'unités de traitement agissant sur des données qui y circulent continuellement ».

⁶¹ "ReactiveX - Intro." <http://reactivex.io/>. Accessed 11 Sep. 2018.

⁶² "On The subject Of subjects (in RxJS) – Ben Lesh – Medium." 9 Dec. 2016, <https://medium.com/@benlesh/on-the-subject-of-subjects-in-rxjs-2b08b7198b93>. Accessed 12 Sep. 2018.

⁶³ "Introducing the Observable from @jhusain on @eggheadio." <https://egghead.io/lessons/rxjs-introducing-the-observable>. Accessed 13 Sep. 2018.

⁶⁴ "SRFI 41 - Scheme Requests for Implementation - Schemers.org." <https://srfi.schemers.org/srfi-41/srfi-41.html>. Accessed 11 Sep. 2018.

⁶⁵ "A Correspondence Between ALGOL 60 and Church's Lambda-Notation." https://fi.ort.edu.uy/innovaportal/file/20124/1/22-landin_correspondence-between-algol-60-and-churchs-lambda-notation.pdf. Accessed 11 Sep. 2018.

EXTRAIT DE CODE I

Transformation d'un événement en observable avec ReactiveX en C#

```
var obs = Observable.FromEventPattern(ev => Button.Clicked += ev, ev =>
Button.Clicked -= ev);
```

Posons-nous deux questions relatives à cette notion de collection au fil du temps : pourquoi ne pourrions-nous pas agir sur des *streams* de données comme nous le ferions sur des tableaux ? En y pensant bien, quelle différence y a-t-il entre recevoir des notifications d'un événement (patron de conception *Observer*) et itérer sur un tableau d'éléments (patron de conception *Iterator*⁶⁶) ?

L'ouvrage populaire *Design Patterns: Elements of Reusable Object-Oriented Software*⁶⁷ publié en 1994 présente les deux patrons de conception séparément :

- *Observer* (modèle *push*) : les nouvelles données sont poussées vers les observers en invoquant leur fonction `update()`.
- *Iterator* (modèle *pull*) : utilisé pour traverser un tableau et accéder à ses éléments grâce aux fonctions `next()` et `hasNext()` (afin de savoir s'il y en a encore). Autrement dit, les éléments sont "tirés" de la liste lors de l'itération.

EXTRAIT DE CODE II

*Itération sur une collection implémentant le patron de conception *Iterator* en C#*

```
var primes = new List<int>{ 2, 3, 5, 7, 11, 13, 17, 19};
foreach (var p in primes)
    ... // p est 'tiré' de la liste
```

Remarquons qu'il s'agit tous deux des moyens de consommer des éléments, l'un après l'autre.

La différence réside dans le fait que les données sont poussées dans un *stream* avec le patron *Observer*, permettant aux observers d'y réagir (réactif). Dans ce processus, c'est l'observable, ou producteur de données, qui contrôle l'envoi.

Quand des données sont 'tirées' d'une liste avec le patron *Iterator*, le consommateur des données contrôle le flux. C'est lui qui tire les données sur de l'itérable (tableau) grâce à un iterator. Dans l'[EXTRAIT DE CODE II](#), le tableau `primes` renvoie un iterator qui permet au code client (`foreach`) de la parcourir et d'en récupérer les éléments.⁶⁸

Nous pouvons aussi stipuler qu'avec un tableau, toutes les données sont stockées en mémoire et sont prêtes à apparaître de façon synchrone. Avec un observable, aucune donnée

⁶⁶ "Design Patterns Iterator Pattern - Tutorialspoint."

https://www.tutorialspoint.com/design_pattern/iterator_pattern.htm. Accessed 13 Sep. 2018.

⁶⁷ "Design Patterns: Elements of Reusable Object-Oriented Software [Book]."

<https://www.oreilly.com/library/view/design-patterns-elements/0201633612/>. Accessed 13 Sep. 2018.

⁶⁸ "Reactive Programming at Netflix – Netflix TechBlog – Medium." 16 Jan. 2013, <http://techblog.netflix.com/2013/01/reactive-programming-at-netflix.html>. Accessed 13 Sep. 2018.

n'est stockée en mémoire et les éléments arrivent dans le temps (ex. des *clics* de souris), de manière asynchrone. Représentons-nous cette idée grâce à la [FIGURE XIII](#) & [FIGURE IX](#).

- Asynchrone signifie ici que le fait d'attendre de nouvelles données venant d'un observable ne bloque pas l'exécution du code client⁶⁹.

An array of values



A stream of values



FIGURE XIII : An array and a stream of values⁷⁰

An Animated Intro to RxJS, 2017

Voici la représentation d'une suite de notifications d'événements *clic* au fil du temps :

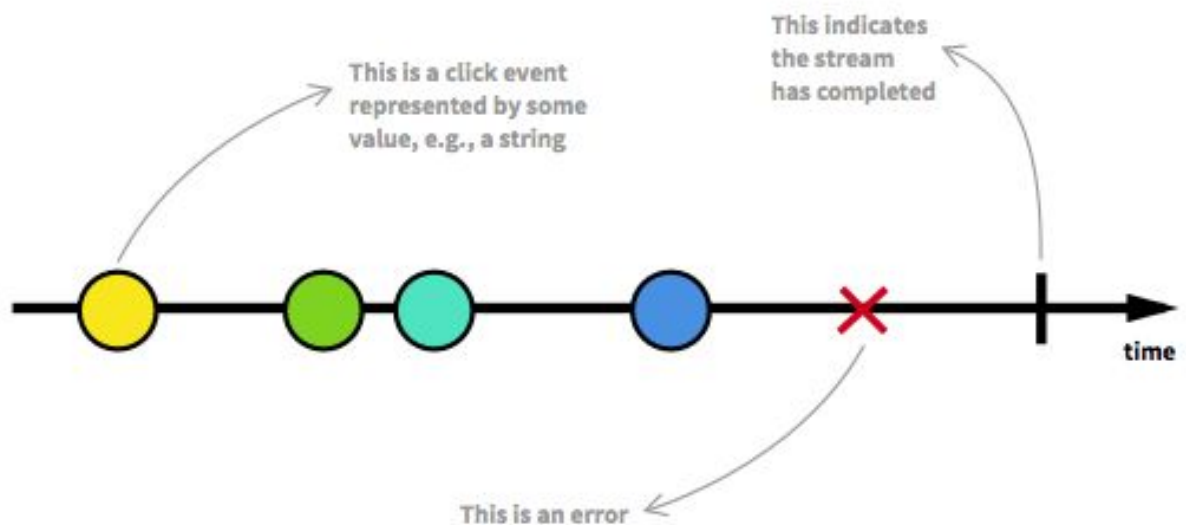


FIGURE IX : Marble Diagram of an Event Stream⁷¹

The introduction to Reactive Programming you've been missing

⁶⁹ "Glossary - The Reactive Manifesto." <https://www.reactivemanifesto.org/glossary>. Accessed 12 Sep. 2018.

⁷⁰ "An Animated Intro to RxJS | CSS-Tricks." 24 Feb. 2017, <https://css-tricks.com/animated-intro-rxjs/>. Accessed 12 Sep. 2018.

⁷¹ "The introduction to Reactive Programming you've been ... - gists · GitHub." <https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>. Accessed 12 Sep. 2018.

3.3.3 Opérateurs

Nous avons vu que dans le patron de conception *Observer*, seule la fonction `update()` était invoquée par le subject pour notifier ses observers. Par contre, quand nous itérons sur un tableau d'éléments, le pattern *Iterator* propose deux fonctions supplémentaires importantes :

- Un moyen pour le producteur de données (tableau) de signaler qu'il n'y a plus de données (fin du tableau) : `hasNext()`.
- Un moyen pour le producteur de données (tableau) de signaler qu'une erreur s'est produite : `next()` qui retourne une erreur.

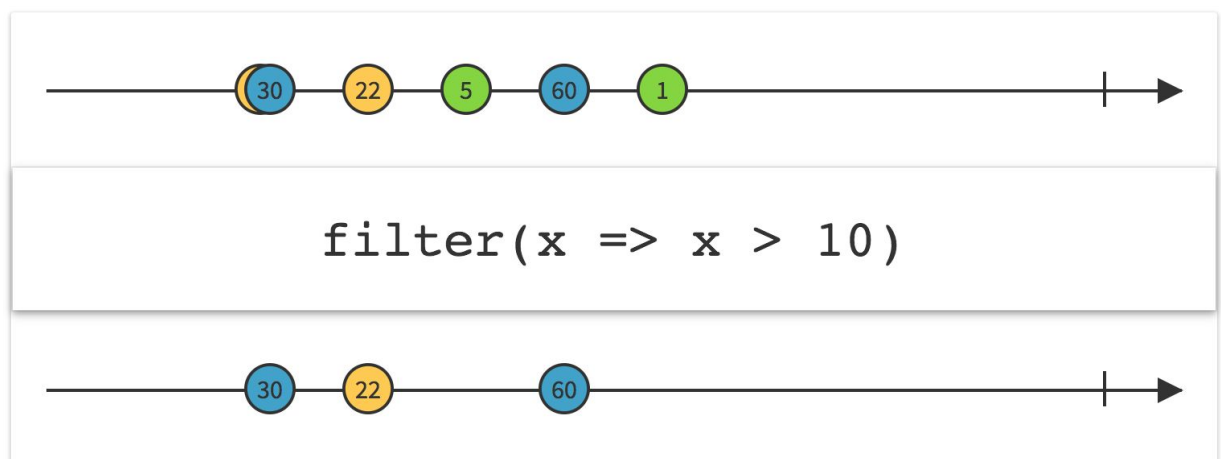
Les *Reactive Extensions* ajoutent ces deux fonctions manquantes au patron de conception *Observer*. Un observer aura alors trois fonctions qui pourront être invoquées par l'observable : `update()`, `onNext()`, `onError()` et `onComplete()`.

*“The Observer pattern done right
ReactiveX is a combination of the best ideas from
the Observer pattern, the Iterator pattern, and functional programming”⁷²*

Cette amélioration offre la possibilité d'appliquer sur les observables des opérateurs tels que ceux utilisés sur les tableaux (*Filter/Where, Map/Select...*). Grâce à ces derniers, il est donc possible de filtrer, de transformer ou de combiner les données au fur et à mesure qu'elles arrivent, en temps-réel. Imaginons par exemple que ces données sont poussées d'un serveur dans notre observable.

Il est à noter que chaque opérateur retourne un nouvel observable sur lequel il est à nouveau possible d'opérer. En voici quelques uns à titre d'exemple :

Filter : filtre un observable pour n'accepter seulement les éléments qui passeront le test spécifié (ex. supérieurs à 10). La seconde ligne représente le nouvel observable. Concrètement, cela signifie que si le test n'est pas passé, la fonction `onNext()` de l'observer ne sera pas appelée (pas de réaction).



⁷² "ReactiveX." <http://reactivex.io/>. Accessed 17 Sep. 2018.

FIGURE XV : Filter operator⁷³
Filter - ReactiveX

Map : transforme les éléments émis par un observable en appliquant une fonction sur chaque élément (ex. multipliés par 10). La seconde ligne représente le nouvel observable.

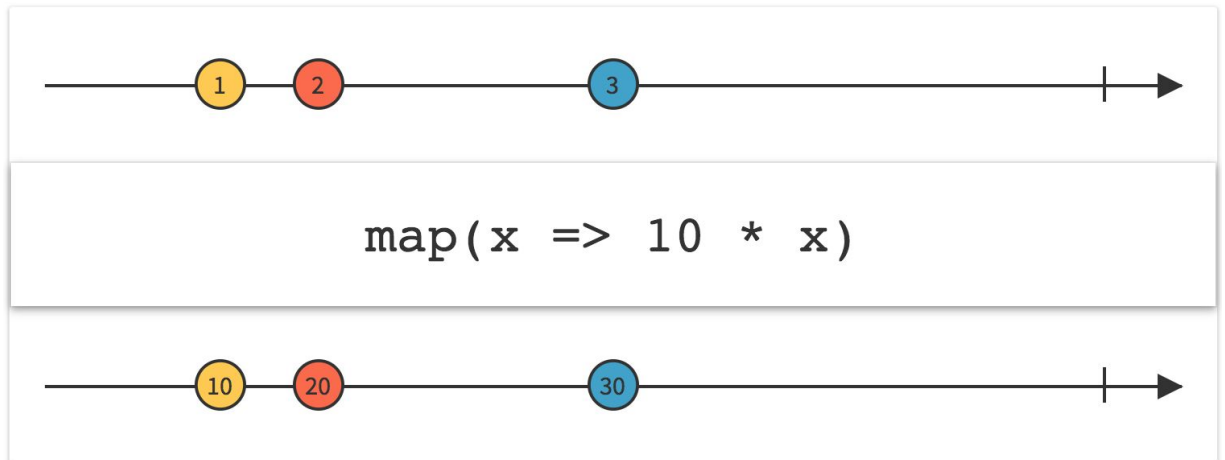


FIGURE XVI : Map operator⁷⁴
Map - ReactiveX

Zip : combine les émissions de plusieurs observables en appliquant une fonction (spécifiée). Tout comme une fermeture éclair sur un vêtement ou un sac, la méthode `Zip` regroupe deux séquences de valeurs; deux par deux.

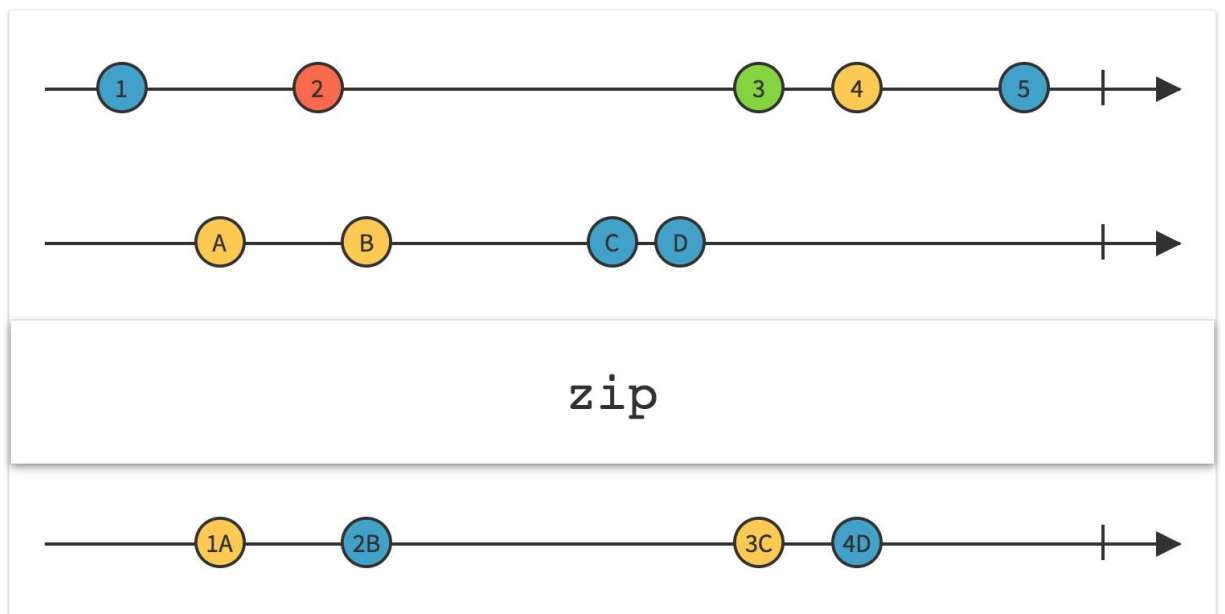


FIGURE XVII : Zip operator⁷⁵
Zip - ReactiveX

⁷³ "ReactiveX - Filter operator." <http://reactivex.io/documentation/operators/filter.html>. Accessed 13 Sep. 2018.

⁷⁴ "ReactiveX - Map operator." <http://reactivex.io/documentation/operators/map.html>. Accessed 13 Sep. 2018.

⁷⁵ "Zip - ReactiveX." <http://reactivex.io/documentation/operators/zip.html>. Accessed 13 Sep. 2018.

3.4 Flux de données unidirectionnel

3.4.1 Concepts

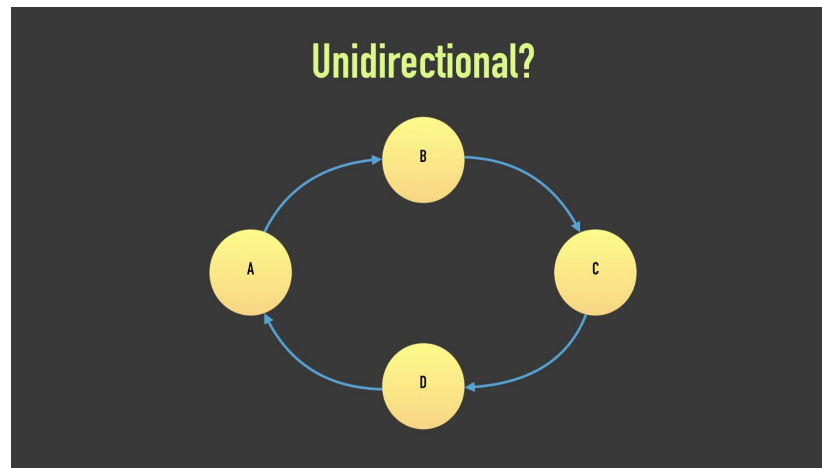


FIGURE XVIII : Unidirectional data flow⁷⁶
Unidirectional data flow on Android using Kotlin, 2018

Le concept de flux de données unidirectionnel est une pratique d'architecture logicielle qui est la plupart du temps utilisée dans les systèmes basés sur les événements.

Nous utiliserons cette approche dans le point [5.3 Architecture réactive envisagée](#).

Il a pour but principal de rendre la logique et les changements de données d'une application plus prévisibles et faciles à suivre⁷⁷. En effet, les données suivent toutes le même chemin et sens, nous permettant de les traiter de manière constante peu importe leur origine.

Un flux de données unidirectionnel se montre particulièrement utile dans une application où⁷⁸ :

- Les données changent dans le temps.
- Les changements doivent être reflétés immédiatement dans l'interface utilisateur.
 - Rappelons qu'il s'agit là d'un principe inhérent à la programmation réactive et aux données temps-réel, sujet de ce travail de fin d'études. Nous souhaitons éviter de devoir re-synchroniser à l'aide d'un bouton pour voir apparaître les changements.
- Les données affichées peuvent être modifiées de différents endroits (ex. action sur la vue, appel à un service), les changements d'états ainsi que le flux général de l'application devenant difficiles à suivre.

⁷⁶ "Unidirectional data flow on Android: The blog post (part 1)." 14 Mar. 2018, <https://proandroiddev.com/unidirectional-data-flow-on-android-the-blog-post-part-1-cadcf88c72f5>. Accessed 13 Sep. 2018.

⁷⁷ "Data Flow - Redux." <https://redux.js.org/basics/dataflow>. Accessed 13 Sep. 2018.

⁷⁸ "The Case for Flux – The Startup – Medium." 18 Feb. 2015, <https://medium.com/swlh/the-case-for-flux-379b7d1982c6>. Accessed 13 Sep. 2018.

Jing Chen, ingénieur logiciel chez Facebook et créatrice de l'architecture *Flux* (qui repose sur ce concept de flux de données unidirectionnel), décrit les bénéfices d'une telle approche lors d'une conférence⁷⁹ (paraphrasé) :

- Si une contrainte unidirectionnelle est imposée à la structure d'une application lors de sa conception, elle n'explose pas en complexité lorsqu'elle évolue.

Flux a été introduit en 2014 et depuis lors, un grand nombre de bibliothèques proposant un flux de données unidirectionnel ont émergé, citons par exemple *Redux*, *MobX* ou *Akita*⁸⁰. Il n'est pas utile d'expliquer le fonctionnement de *Flux* dans le cadre de ce travail, mais remarquons la cohérence que l'architecture propose, les flèches suivant toujours la même direction :

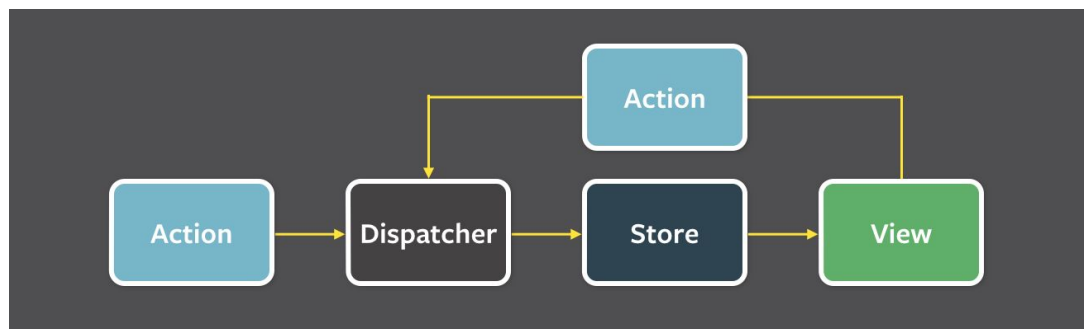


FIGURE XIX : Flux flow of data⁸¹
Flux concepts, 2017

3.4.2 Store, *single source of truth*

*"When in doubt, ask the store."*⁸²

Il existe des différences entre les bibliothèques précédemment énoncées, mais toutes sont alignées sur les principes généraux du *store*.

Le *store*, parfois nommé *state*, désigne l'état d'une application ou plus concrètement le seul endroit où se trouvent les données affichées dans les différentes vues.

Pour cette raison, il est considéré en tant que source unique de vérité, ou *single source of truth*.

Il est conseillé de veiller à maîtriser au maximum les changements dans le *store* et d'être prudent quant à sa modification pour éviter les altérations d'états non désirées.⁸³

⁷⁹ "React and Flux: Building Applications with a Unidirectional ... - YouTube." 28 Aug. 2014, https://www.youtube.com/watch?v=i_969noyAM. Accessed 13 Sep. 2018.

⁸⁰ "GitHub - datorama/akita: Simple and Effective State Management for" <https://github.com/datorama/akita>. Accessed 13 Sep. 2018.

⁸¹ "flux/examples/flux-concepts at master · facebook/flux · GitHub." <https://github.com/facebook/flux/tree/master/examples/flux-concepts>. Accessed 13 Sep. 2018.

⁸² "Single Source of Truth – Emmanuel Fleurine – Medium." 7 Aug. 2017, <https://medium.com/@EmmaFleurine/single-source-of-truth-3e83001159c0>. Accessed 13 Sep. 2018.

⁸³ "Single Source of Truth – Emmanuel Fleurine – Medium." 7 Aug. 2017, <https://medium.com/@EmmaFleurine/single-source-of-truth-3e83001159c0>. Accessed 13 Sep. 2018.

3.4.3 Comparaison avec Model-View-Controller

Jing Chen explique également comment Facebook a pu corriger un de leur problème les plus ennuyeux (la pastille de messages non-lus apparaissant intempestivement) sur leur messagerie instantanée en abandonnant le traditionnel patron de conception *Model-View-Controller* (*MVC*) en faveur de l'architecture *Flux*⁸⁴.

La complexité croissante rencontrée avec les patrons de conception de type *MVC* peut être représentée comme ceci :

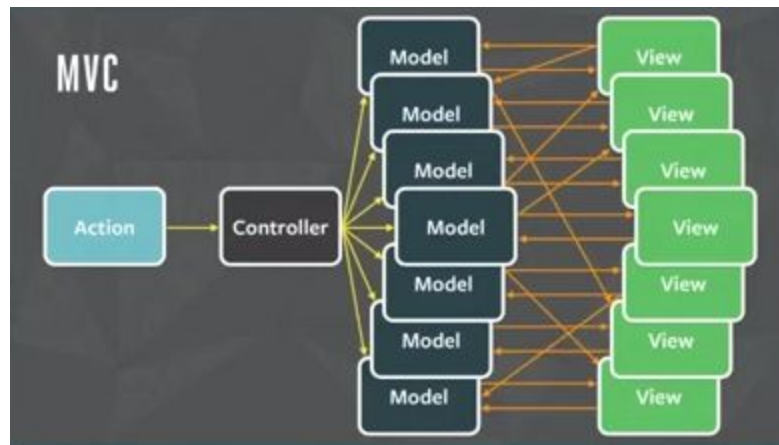


FIGURE XX : Model-View-Controller *scalability*⁸⁵
Hacker Way: Rethinking Web App Development at Facebook, 2014

Notons qu'une application suivant le patron de conception *MVC* dans les règles de l'art ne devrait pas évoluer de cette manière, mais que ce cas se présente suite au manque de contraintes imposées par la structure.

Le patron de conception *Model-View-ViewModel*⁸⁶ ou *MVVM*, largement utilisé avec la technologie *Xamarin* (avec laquelle notre application e-commerce est développée), travaille avec un mode de *binding* (liaison) *TwoWay* (bidirectionnel) entre la vue et le *ViewModel*.

La vue se met à jour (réactivité) sur base du *ViewModel* qui contient les données affichées, mais une action de l'utilisateur (sur la vue) peut également modifier le *ViewModel*. Le *ViewModel* manipule lui les modèles suite à l'appel à un service ou un changement qu'un utilisateur effectue à partir de la vue. Nous avons clairement un flux de données bidirectionnel dans ce cas.

⁸⁴ "Hacker Way: Rethinking Web App Development at Facebook - [https](https://www.youtube.com/watch?v=nYkdrAPrdcw)"
<https://www.youtube.com/watch?v=nYkdrAPrdcw>. Accessed 13 Sep. 2018.

⁸⁵ "Hacker Way: Rethinking Web App Development at Facebook - [https](https://www.youtube.com/watch?v=nYkdrAPrdcw)"
<https://www.youtube.com/watch?v=nYkdrAPrdcw>. Accessed 13 Sep. 2018.

⁸⁶ "The Model-View-ViewModel Pattern - Xamarin | Microsoft Docs." 6 Aug. 2017,
<https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm>. Accessed 13 Sep. 2018.

La [FIGURE XXI](#) illustre cette différence de manière simplifiée.

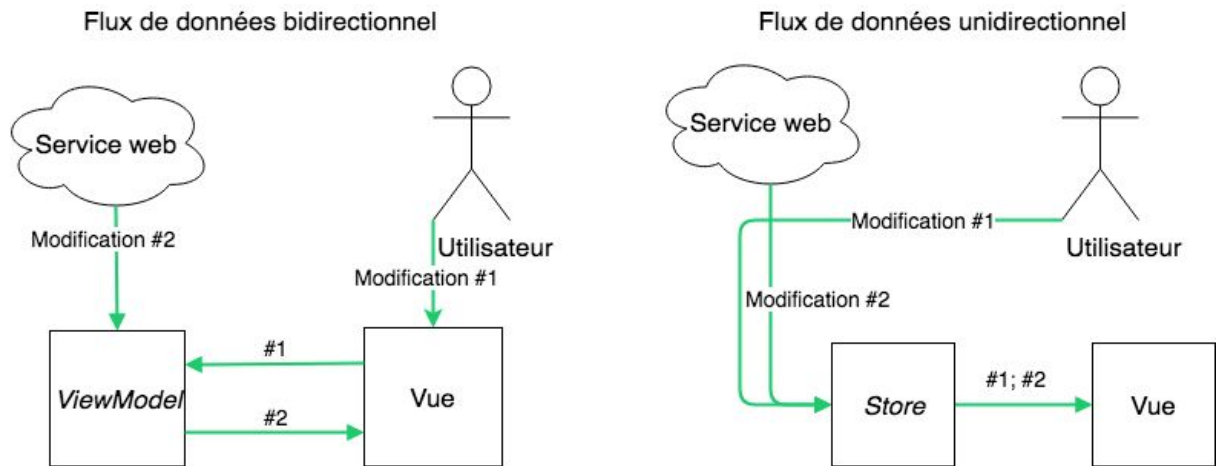


FIGURE XXI : Flux de données uni et bidirectionnels

Samih Alkeilani, 2018

3.4.4 Points clés

À titre d'exemple, voici le schéma de fonctionnement d'*Akita*, qui se rapproche le plus de notre manière d'implémenter une architecture réactive avec un flux de données unidirectionnel (présentée au point [5.3 Architecture réactive envisagée](#)).

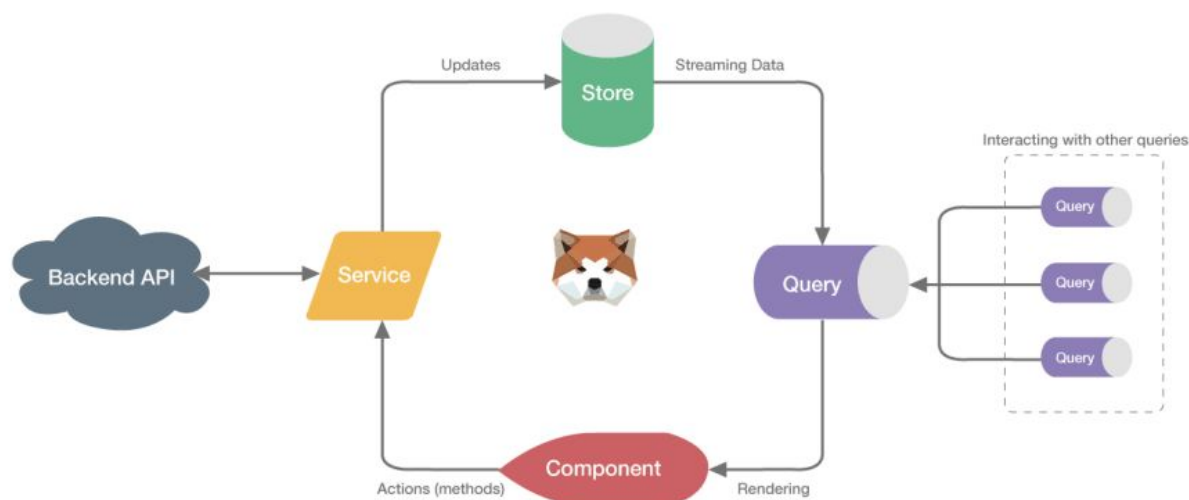


FIGURE XXII : Akita Architecture⁸⁷

Akita, Simple and Effective State Management for Angular Applications

Regardons brièvement la [FIGURE XXII](#) et mettons en avant les éléments que nous exploiterons dans notre architecture :

- Le *store* ou source unique de vérité : contient l'état et données de l'application.
- Les composants : ce sont nos vues - elles affichent les données du *store*. Nous verrons que, grâce au patron de conception *Observer*, ces dernières sont mises à jour quand le *store* est modifié.
- Les services : modifient les données du *store*. Notons qu'aucune autre partie de l'application ne possède ce pouvoir. Ces modifications ont typiquement lieu suite à un appel à un service web.

Pouvoir plus facilement suivre le flux des données et changements prend tout son sens dans le contexte d'une application réactive qui réagit à des données provenant de plusieurs sources (serveur, interface utilisateur...), potentiellement simultanément.

⁸⁷ "GitHub - datorama/akita: Simple and Effective State Management for"
<https://github.com/datorama/akita>. Accessed 13 Sep. 2018.

Deuxième partie : Pratique

Nous abordons à présent la partie pratique de ce travail de fin d'études. Dans un premier temps, nous ferons un détour par une introduction aux bases de données *NoSQL* et à l'authentification avant de présenter la plateforme *back-end Firebase* où nous utiliserons ces deux techniques travers de produits tels que *Firebase Realtime Database* et *Firebase Authentication*.

Nous présenterons ensuite les technologies *front-end* choisies pour le développement de l'application *BdOccaz*, pour enfin exposer des mock-ups de l'application au chapitre 6, en passant par la conception de sa base de données.

4 Chapitre IV : Technologies back end

4.1 Introduction au NoSQL

Note : En supplément des différentes sources référencées, les explications ci-après sont inspirées de la conférence « *GOTO 2012 - Introduction to NoSQL* » de *Martin Fowler* (le lien se trouve au point [9 Bibliographie](#)).

4.1.1 Origines

Les bases de données relationnelles ou *RDBMS*⁸⁸ ont été le premier choix pour le stockage d'informations financières, de fabrication et de logistique, les données du personnel et d'autres applications depuis les années 1980.

“Un schéma relationnel est essentiellement un groupe de tables représentant des objets et des relations entre ceux-ci. Ces tables, faites de lignes et colonnes, peuvent être interrogées à l'aide d'un langage d'interrogation structuré (SQL).

*Dans un schéma relationnel, les tables sont normalisées dans le but d'éviter les doublons et de consolider les données. Chaque entité ou relation possède une représentation minimaliste qui peut s'étendre grâce à des jointures des tables.”*⁸⁹

Avec ces systèmes fiables viennent certains problèmes. Observons la [FIGURE XXIII](#) afin d'en explorer un des plus connus.

⁸⁸ Relational Database Management System

⁸⁹ "L'élasticité des bases de données sur le cloud computing - CoDE" 18 Apr. 2011, <http://code.ulb.ac.be/dbfiles/Deg2011mastersthesis.pdf>. Accessed 16 Sep. 2018.

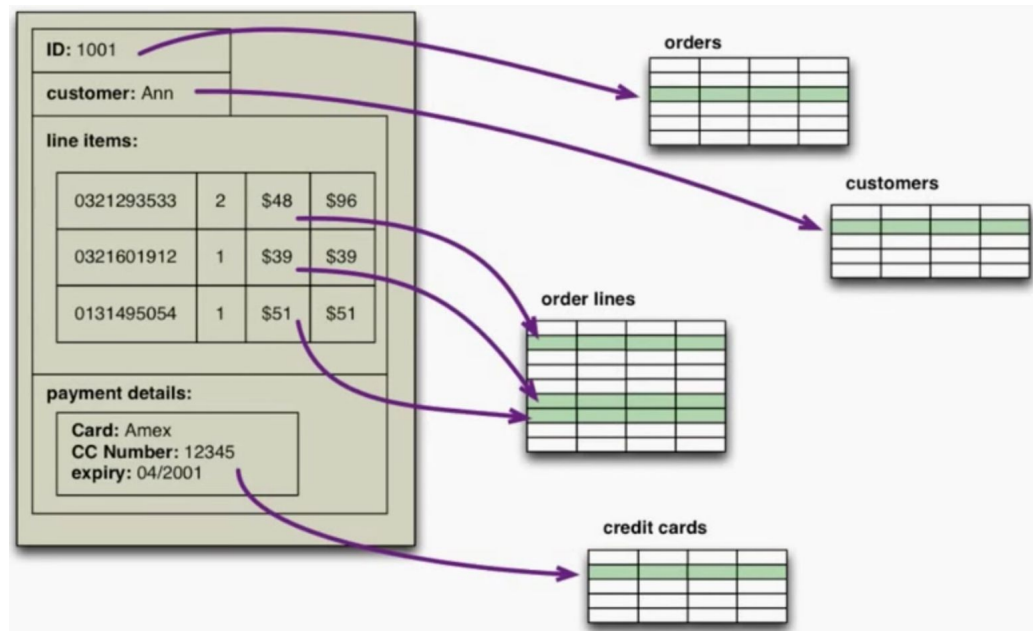


FIGURE XXIII : Relational databases impedance mismatch problem

GOTO 2012 - Introduction to NoSQL, Martin Fowler

En tant que développeurs applicatifs, nous assemblons des structures d'objets en mémoire dans nos applications (souvent en tant qu'ensemble cohésif, ex. un objet "commande" contient toutes les informations liées à la commande).

Cependant, quand vient le moment de sauvegarder ces données, nous sommes souvent amenés à devoir les séparer en morceaux afin qu'ils correspondent aux différentes lignes et tables de notre base de données relationnelle (la [FIGURE XXIII](#) illustre ce cas). Une seule structure logique (objet) pour une interface utilisateur se retrouve alors séparée dans notre base de données.

Ce problème est connu sous le nom d'*impedance mismatch problem*, ou *object-relational impedance mismatch problem*. Il désigne les difficultés rencontrées lorsqu'il s'agit de faire correspondre ces deux modèles (objet et relationnel) différents⁹⁰.

C'est en partie pour cette raison que sont apparues les bases de données objet (non-relationnelles) dans le courant des années 90. Avec ces dernières, il devenait possible de sauvegarder directement une structure objet sur disque, répondant au problème précédemment énoncé. Cela a poussé une partie du secteur à croire que les bases de données objet allaient complètement remplacer les *RDBMS* classiques.

Cela n'a toutefois pas été le cas, et c'est seulement entre 2000 et 2010 que les bases de données objet ont connu une explosion en popularité, se positionnant en tant qu'alternatives aux systèmes relationnels⁹¹.

⁹⁰ "The Object-Relational Impedance Mismatch - Agile Data."

<http://www.agiledata.org/essays/impedanceMismatch.html>. Accessed 16 Sep. 2018.

⁹¹ "NoSQL databases eat into the relational database market" 4 Mar. 2015,

<https://www.techrepublic.com/article/nosql-databases-eat-into-the-relational-database-market/>. Accessed 16 Sep. 2018.

Quel élément a-t-il donc provoqué ce changement ?

Une des raisons principales est probablement la croissance exponentielle des données et du trafic sur internet. Afin de répondre à ce volume important, nous rencontrons le besoin de monter à l'échelle, ou de *scalability*.

4.1.2 La capacité de monter à l'échelle ou *scalability*

Nous avons brièvement abordé le principe de *scalability* [plus tôt dans ce travail](#). Nous allons maintenant l'approfondir et le replacer dans le contexte des bases de données. Les définitions ci-dessous sont tirées du mémoire "L'élasticité des bases de données sur le cloud computing" écrit par Nicolas Degroodt pour l'Université Libre de Bruxelles.⁹²

- (1) "La capacité à **monter à l'échelle** d'un service est sa capacité à pouvoir assumer une production constante lorsque le nombre de requêtes augmente."

Afin qu'un serveur de base de données puisse monter à l'échelle lorsque le nombre de requêtes augmente, nous pourrions ajouter des ressources verticalement (*scale up*) ou horizontalement (*scale out*).⁹³

- (2) "Considérant un service, (...) monter à l'échelle **verticalement** est la propriété qui décrit l'évolution (...) lorsqu'on **augmente** ses **ressources** (CPU, mémoire, etc.)."
- (3) "Considérant un service, (...) monter à l'échelle **horizontalement** est la propriété qui décrit l'évolution (...) lorsqu'on augmente le **nombre d'instances**."

En résumé, nous dirons que :

- Le *scaling up* correspond à l'ajout de ressources (mémoire, processeurs) à un seul noeud. Cette approche va le plus souvent de pair un système centralisé conventionnel ([FIGURE XXIV](#)) muni d'une base de données relationnelle.
- Le *scaling out* vise à ajouter un noeud à un *cluster*, technique propre aux architectures distribuées.

- (4) "Un *cluster* d'ordinateur est un parc d'ordinateurs **interconnectés** (on parle souvent de grappe) afin de **partager** leurs **ressources** dans un but commun."

En tant qu'utilisateur, nous pouvons nous représenter un *cluster* comme une seule unité de traitement logique.

Outre les différentes analogies que nous ferons, l'accent sera porté sur les architectures distribuées dans ce travail car il s'agit du moyen le plus adapté de monter à l'échelle avec une bases de données *NoSQL*.

⁹² "L'élasticité des bases de données sur le cloud computing - CoDE" 18 Apr. 2011, <http://code.ulb.ac.be/dbfiles/Deg2011mastersthesis.pdf>. Accessed 14 Sep. 2018.

⁹³ "Scale-up x Scale-out: A Case Study using Nutch/Lucene - IEEE Xplore." <https://ieeexplore.ieee.org/document/4228359/>. Accessed 14 Sep. 2018.

4.1.3 Bases de données distribuées

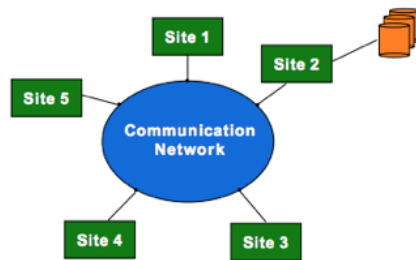
*"A distributed database is a collection of multiple, **logically** interrelated databases distributed over a computer network."*⁹⁴

Dans une base de données distribuée, les données sont stockées sur plusieurs ordinateurs ou noeuds (l'ensemble formant un *cluster*)^{95,96} et un processus de réplication maintient généralement les données à jour entre les différents noeuds. Stocker les données sur un seul serveur relèverait plutôt d'une approche centralisée.

Ces dernières constituent un moyen de monter à l'échelle adapté aux bases de données objet *NoSQL* que nous rencontrerons dans notre cas pratique. Elles tendent également à être financièrement de plus en plus avantageuses à vitesse et disponibilité comparables, grâce à :

- L'utilisation de *commodity hardware* (matériel à bas prix) : Le besoin pour des serveurs très puissants n'est plus réellement présent dans un système distribué étant donné qu'il s'agit un grand nombre d'ordinateurs (moins puissants) qui travaillent ensemble.
- L'accessibilité des connexions à haut débit, qui rendent la communication aisée entre les différents ordinateurs du *cluster*.⁹⁷

Centralized DBMS on a Network



Distributed DBMS Environment

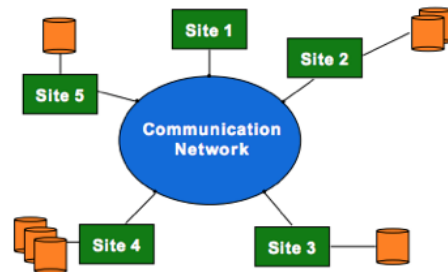


FIGURE XXIV : Systèmes centralisés et distribués⁹⁸
What is a Distributed Database System. University of Waterloo

⁹⁴ "What is a Distributed Database System."

<https://cs.uwaterloo.ca/~tozsu/courses/cs856/F02/lecture-1-ho.pdf>. Accessed 14 Sep. 2018.

⁹⁵ Un ordinateur ou un serveur

⁹⁶ "Definition: distributed database." https://www.its.bldrdoc.gov/fs-1037/dir-012/_1750.htm. Accessed 14 Sep. 2018.

⁹⁷ "Cluster Computing: Applications - Georgia Tech College of Computing." 25 Jul. 2004, <https://www.cc.gatech.edu/~bader/papers/ijhpc.html>. Accessed 16 Sep. 2018.

⁹⁸ "What is a Distributed Database System." <https://cs.uwaterloo.ca/~tozsu/courses/cs856/F02/lecture-1-ho.pdf>. Accessed 14 Sep. 2018.

Théorème CAP ou de Brewer

La spécificité d'une base de données est qu'elle stocke des données dont nous devons pouvoir garantir une certaine intégrité. Dans le contexte des systèmes distribués, ce problème peut être résumé par une conjecture connue sous le nom de théorème CAP ou théorème de Brewer.⁹⁹ Elle est énoncée comme suit :

*“Parmi les trois propriétés suivantes : **consistance**, **haute disponibilité** et **tolérance à la partition**, tout système de données **distribué** ne peut respecter, au plus, que deux d'entre elles.”¹⁰⁰*

CAP introduit des propriétés relatives aux systèmes distribués qui nous sont inconnues. Nous allons maintenant les décrire.

Consistance des données : les données récupérées au même moment de différents noeuds d'un système distribué doivent toujours être équivalentes. Autrement dit, les données auxquelles nous accédons doivent toujours être à jour.

- Si cette propriété est laissée de côté, il n'est pas garanti que les données récupérées contiennent la dernière écriture.



FIGURE XXV : Représentation de la consistance des données¹⁰¹
CAP Theorem: Revisited, 2014

Haute disponibilité : les données doivent être récupérées, même si certains noeuds du cluster ne sont pas fonctionnels (hors service).

- Si cette propriété est laissée de côté, nous pouvons recevoir un time-out/erreur lors d'une requête.

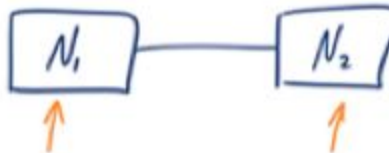


FIGURE XXVI : Représentation de la haute disponibilité
CAP Theorem: Revisited, 2014

⁹⁹ "Brewer's CAP Theorem in Simple Words - HowToDoInJava." <https://howtodoinjava.com/hadoop/brewers-cap-theorem-in-simple-words/>. Accessed 16 Sep. 2018.

¹⁰⁰ "L'élasticité des bases de données sur le cloud computing - CoDE" 18 Apr. 2011, <http://code.ulb.ac.be/dbfiles/Deg2011mastersthesis.pdf>. Accessed 16 Sep. 2018.

¹⁰¹ "CAP Theorem: Revisited - Robert Greiner." 14 Aug. 2014, <http://robertgreiner.com/2014/08/cap-theorem-revisited/>. Accessed 16 Sep. 2018.

Pour des raisons de performance et de sécurité (réplication des données), les systèmes distribués tirent profit du fait de pouvoir s'étirer sur plusieurs instances. Répartir un service sur de nombreuses instances augmente le risque de défaillance d'une partie du service, donnant naissance à une partition. Pour expliquer le phénomène de partition, imaginons notre service distribué sur deux boîtes noires ; un phénomène de partition arrive lorsque le câble reliant ces deux boîtes noires est débranché (les deux sous-systèmes ne sont plus capables de communiquer).¹⁰²

Tolérance à la partition : Un service distribué tolérant à la partition est un système tel qu'un [phénomène de partition](#) n'altère pas son bon fonctionnement. Concrètement, le cluster (entier) doit continuer à fonctionner même s'il y a une erreur de réseau entre certains noeuds.¹⁰³ Contrairement à la haute disponibilité, les deux noeuds sont ici fonctionnels mais la liaison entre les deux est coupée (voir [FIGURE XXVII](#)).

La seule exception à cette règle serait une erreur de réseau mettant en péril l'entière du réseau. L'idée est ici que les données doivent être suffisamment répliquées à travers les combinaisons de noeuds pour que le système puisse continuer à fonctionner lors d'erreurs réseau partielles.

- Si cette propriété est laissée de côté, le système peut ne plus répondre lorsqu'une erreur réseau survient.



FIGURE XXVII : Représentation de la tolérance à la partition

CAP Theorem: Revisited, 2014

Notons que l'interconnexion de noeuds (ou ordinateurs) est l'essence des systèmes distribués. Ces derniers sont pas à l'abri d'une erreur réseau, donc la tolérance à la partition se veut une propriété élémentaire qui doit être garantie.

¹⁰² "L'élasticité des bases de données sur le cloud computing - CoDE" 18 Apr. 2011, <http://code.ulb.ac.be/dbfiles/Deg2011mastersthesis.pdf>. Accessed 16 Sep. 2018.

¹⁰³ "Brewer's Conjecture and the Feasibility of Consistent ... - CMU (ECE)." <https://users.ece.cmu.edu/~adrian/731-sp04/readings/GL-cap.pdf>. Accessed 16 Sep. 2018.

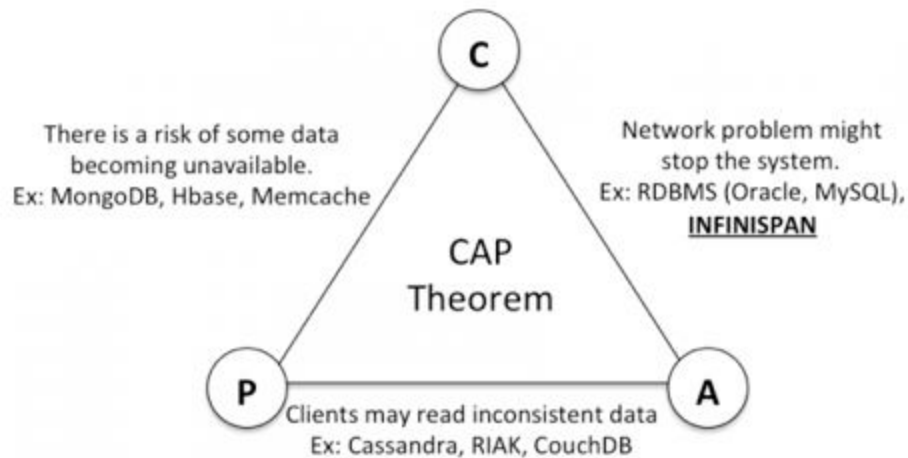


FIGURE XXVIII : CAP Theorem¹⁰⁴
Brewer's CAP Theorem in Simple Words - HowToDoInJava

- Les systèmes dits *CA* (consistants et hautement disponibles) désignent les systèmes de bases de données relationnelles.
 - Étant donné qu'ils fonctionnent généralement sur une seule instance, la tolérance à la partition n'existe pas et un problème de réseau peut effectivement causer une non-réponse du noeud (et donc un échec du système).
- Les systèmes dits *AP* (hautement disponibles et tolérants à la partition) garantissent la haute disponibilité au prix de la consistance des données.
 - Lors d'une erreur réseau (partition) ou d'une baisse de performances¹⁰⁵, ces derniers privilégient l'envoi d'une réponse au client, même si elle ne contient pas les dernières données (rappelons qu'un choix doit être fait entre ces deux dernières propriétés car les trois ne peuvent être garanties).
- Les systèmes dits *CP* (consistants et tolérants à la partition) garantissent la consistance des données au prix de la disponibilité lorsqu'une [partition](#) est présente.
 - En cas d'erreur réseau, de tels systèmes privilégient l'envoi de données à jour et s'ils ne peuvent le garantir, une erreur sera renvoyée.

Notons qu'il s'agit d'une classification théorique, et qu'en pratique, la différence entre certains systèmes est assez difficile à discerner.

Il existe à présent également des bases de données relationnelles distribuées, pensées dans l'effort d'être adaptées aux *clusters*. Ces dernières sont souvent classifiées en *CA* d'un point de vue *CAP*, comme montré dans la [FIGURE XXVIII](#).

¹⁰⁴ "Brewer's CAP Theorem in Simple Words - HowToDoInJava."

<https://howtodoinjava.com/hadoop/brewers-cap-theorem-in-simple-words/>. Accessed 16 Sep. 2018.

¹⁰⁵ "DBMS Musings: Problems with CAP, and Yahoo's little known NoSQL" 23 Apr. 2010, <http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html>. Accessed 16 Sep. 2018.

Propriétés ACID des bases de données relationnelles

L'objectif de ce point est de comprendre la raison pour laquelle les bases de données relationnelles sont potentiellement moins adaptées à une architecture distribuée que les bases de données objet (*NoSQL*).

Les bases de données relationnelles (*SQL*) reposent sur la notion de transactions. Une transaction est un groupe d'opérations (récupérations, insertions, modification) qui respectent les propriétés *ACID*¹⁰⁶. Penchons-nous sur ces dernières :

- Atomicité : l'ensemble des opérations est une suite indissociable c'est-à-dire qu'en cas d'échec d'une opération, tout l'ensemble sera annulé (y compris les opérations précédentes).
- Consistance : le résultat de l'ensemble des opérations doit être cohérent selon un ensemble de règles.
- Isolation : lorsque deux transactions ont lieu en même temps, les modifications effectuées par la première ne sont visibles par la seconde que lorsque la première est terminée.
- Durabilité : une transaction terminée ne peut être annulée.

¹⁰⁷

Le respect de ces propriétés engendrent une complexité supplémentaire dans le contexte d'un système distribué.

Par exemple, une transaction distribuée sur plusieurs noeuds demanderait également une annulation sur ces différents noeuds en cas d'erreur (afin de respecter la propriété d'atomicité).

C'est pour cette raison que les bases de données relationnelles sont généralement mises en place dans un environnement non-distribué (ou qu'elles sacrifient la tolérance à la partition). Par opposition, avec les bases de données *NoSQL*, l'entièreté des données d'une requête seront souvent stockées sur un seul noeud (dû aux structures objets riches) et rendent le travail en *cluster* plus aisé.

¹⁰⁶ "What is a Transaction? | Microsoft Docs." 30 May. 2018, <https://docs.microsoft.com/en-us/windows/desktop/ktm/what-is-a-transaction>. Accessed 16 Sep. 2018.

¹⁰⁷ "L'élasticité des bases de données sur le cloud computing - CoDE" 18 Apr. 2011, <http://code.ulb.ac.be/dbfiles/Deg2011mastersthesis.pdf>. Accessed 16 Sep. 2018.

4.1.4 Le mouvement NoSQL

(Bases de données NoSQL)

*“Next Generation Databases mostly addressing some of the points: being **non-relational, distributed, open-source and horizontally scalable**.”¹⁰⁸*

Nous allons maintenant passer à la présentation du mouvement *NoSQL*, devenu populaire suite à l’adoption des systèmes distribués que nous venons de décrire et de comparer avec les architectures centralisées (type relationnel). Remarquons que sa définition met en avant le terme *horizontally scalable*, concept inhérent à l’approche distribuée.

Comme déjà mentionné, les quantités considérables de données et le [besoin de monter à l’échelle](#) qui en découle ont rendu le *NoSQL* populaire. L’arrivée du *Web 2.0* en 2009 a également joué son rôle en amenant le stockage de contenu non-structuré (ex. contenu généré par l’utilisateur de type articles de blog).

Nous l’aurons compris, les bases de données *NoSQL* proposent un mécanisme pour le stockage et la récupération de données, avec lequel les données sont modélisées différemment des bases de données relationnelles. Analysons à présent ces différences.

Comparaison avec le *SQL*

Le consensus général dans le domaine est que les bases de données *NoSQL* proposent une meilleure flexibilité de par leur schéma beaucoup moins strict (parfois appelé sans schéma, ou *schema-less*), les bases de données *SQL* offrant en revanche une intégrité des données supérieure, notamment grâce à leurs propriétés [ACID](#).¹⁰⁹

Les bases de données *NoSQL* ont pour but le stockage de large quantités de données sans trop se soucier de leur structure.

Elles enregistrent, par exemple, les données sous forme de collections (ex. tables) de documents *JSON*¹¹⁰ où un champ (ex. colonne) peut typiquement être ajouté à un document (ex. ligne) sans modifier la structure de ceux déjà enregistrés.

Nous dirons que leur schéma est dynamique, approprié aux données non-structurées.

¹⁰⁸ "NoSQL Databases." <http://nosql-database.org/>. Accessed 16 Sep. 2018.

¹⁰⁹ "SQL vs NoSQL: The Differences Explained - Panoply Blog." 9 Mar. 2017, <https://blog.panoply.io/sql-or-nosql-that-is-the-question>. Accessed 16 Sep. 2018.

¹¹⁰ Il existe plusieurs types de base de données NoSQL : orientées document, clé-valeur, orientées graphes...

Schema-less signifie que deux documents *NoSQL* n'ont pas besoin d'avoir des champs communs et qu'ils peuvent stocker des types de données différents (voir [EXTRAIT DE CODE III](#)).

EXTRAIT DE CODE III

Documents ayant des structures différentes au sein d'une collection

```
[
  {
    ISBN: 9780992461225,
    title: "JavaScript: Novice to Ninja",
    author: "Darren Jones",
    price: 29.00
  },
  {
    ISBN: 9780994182654,
    title: "Jump Start Git",
    author: "Shaumik Daityari",
    format: "ebook",
    price: 29.00
  }
]
```

Une pratique connue en *NoSQL* est celle de dénormalisation des données. Par opposition aux bases de données SQL normalisées (formes normales), il est courant dans une base de données *NoSQL* de dupliquer des informations afin d'obtenir toutes les données liées à un document lors de sa récupération.

La normalisation de base de données est une technique qui garantit que les données sont structurées de manière à ce que des modifications de données n'entraînent pas d'incohérence. Par contre, la dénormalisation de la base de données consiste à optimiser les accès en lecture dans la base de données en créant des données redondantes.¹¹¹

Par exemple, en *NoSQL*, les informations d'un livre pourraient très bien se retrouver dans une collection "livres" mais également dans la collection "ventes" s'il s'avère souvent nécessaire dans notre application de récupérer une vente avec le livre qui y est lié. Cela permet d'améliorer le temps de lecture car nous avons toutes les données à portée de main en une seule requête (aucune agrégation n'est nécessaire), mais il faudrait cependant modifier le livre à deux endroits différents si ce dernier venait à changer. Si cela n'est pas fait, nous aurons de l'inconsistance dans nos données (la technique de mise à jour multi-chemins¹¹² est proposée en réponse à ce problème).

¹¹¹ "Building Scalable Databases: Denormalization, the NoSQL Movement" 10 Sep. 2009, <http://www.25hoursaday.com/weblog/2009/09/10/BuildingScalableDatabasesDenormalizationTheNoSQLMovementAndDigg.aspx>. Accessed 16 Sep. 2018.

¹¹² "Firebase Multi-path Updates — Updating Denormalized Data in" 24 Feb. 2017, <https://medium.com/@danbroadbent/firebase-multi-path-updates-updating-denormalized-data-in-multiple-locations-b433565fd8a5>. Accessed 18 Sep. 2018.

Dans cette ligne d'idée, une base de données *NoSQL* est en fait souvent modélisée en fonction des différents écrans du front end. Durant sa conception, il est souvent question de jongler entre l'approche dénormalisée (gain en temps de lecture, potentielle perte de consistance) et normalisée (gain potentiel en consistance, perte en temps de lecture) selon le problème rencontré.

"Structure your data according to your view."¹¹³

Notons que même si les bases de données *NoSQL* ne garantissent pas forcément la réussite d'une série d'opérations (transactions), le besoin est moins présent qu'en relationnel.

Par opposition aux tables qui tendent à être minimalistes de par leur normalisation, les documents sont des structures riches (dont les données se trouveraient typiquement dans plusieurs tables *SQL*) et les solutions *NoSQL* offrent généralement l'atomicité au niveau du document (A des propriétés [ACID](#)).

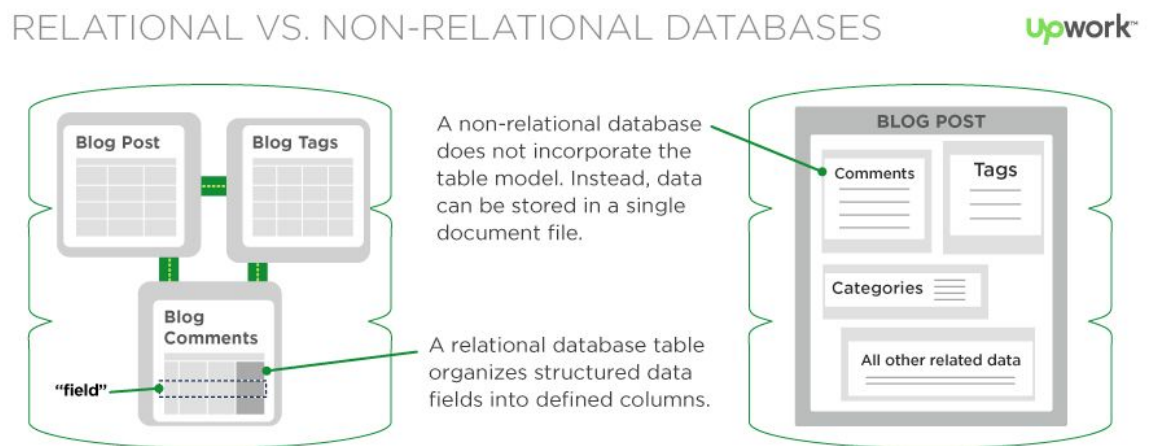


FIGURE XXIV : Relational vs. non-relational databases¹¹⁴

SQL vs. NoSQL: What's the difference?, Upwork

Nous pouvons constater dans la [FIGURE XXIV](#) que le modèle de données proposé par le *NoSQL* se rapproche à celui des objets dans nos applications (ex. tout est contenu dans un "BLOG POST").

Il existe actuellement plus de 225 bases de données *NoSQL* différentes sur le marché. Nous utiliserons dans le cadre de l'application *BdOccaz* la base de données *NoSQL Firebase Realtime Database*, présentée au [point 4.3.2](#).

¹¹³ "Denormalization: How, When and Why (part2) | Codementor." 4 Oct. 2017, <https://www.codementor.io/nwankwo.c.michael/denormalization-how-when-and-why-part2-ciu7f6vni>. Accessed 18 Sep. 2018.

¹¹⁴ "SQL vs. NoSQL: What's the difference? - Upwork." <https://www.upwork.com/hiring/data/sql-vs-nosql-databases-whats-the-difference/>. Accessed 16 Sep. 2018.

4.2 Introduction à l'authentification

OAuth est un standard pour l'autorisation et la délégation d'accès, et chaque serveur peut l'implémenter.

La délégation d'accès signifie concrètement que l'accès à une ressource (ex. des données dans une base de données) n'est pas autorisé par le serveur qui la possède, mais par un serveur d'autorisation prévu à cet effet. Cette approche est utilisée comme moyen pour les applications d'accéder à des informations possédées par d'autres sites (ex. informations de compte *Google*, *Facebook*, *Twitter*...), sans avoir le mot de passe de l'utilisateur.¹¹⁵

OAuth répond à la question :

“Comment pourrions-nous autoriser une application à accéder à certaines de nos données sans lui donner un mot de passe ?”

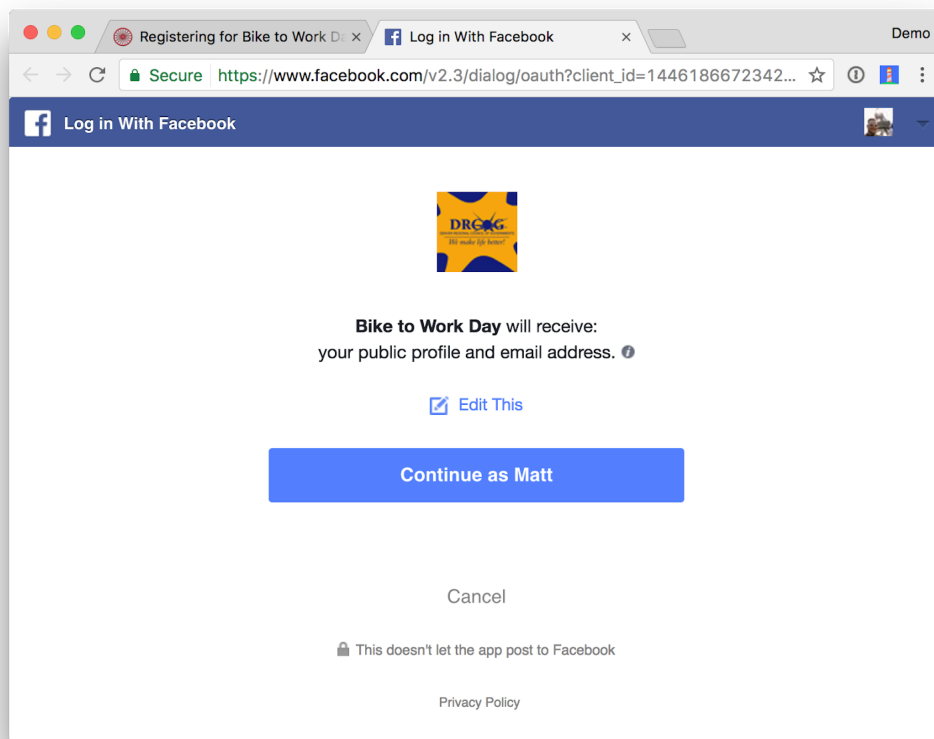


FIGURE XXX : OAuth

What the Heck is OAuth? Matt Raible, 2017

Dans la [FIGURE XXX](#), il est demandé à *Matt* de se connecter avec son compte *Facebook* et d'autoriser l'application *Bike to Work Day* à accéder à son profil et son adresse e-mail.

¹¹⁵ "Understanding OAuth: What Happens When You Log Into a Site with" 13 Jun. 2012, <https://lifehacker.com/5918086/understanding-oauth-what-happens-when-you-log-into-a-site-with-google-twitter-or-facebook>. Accessed 15 Sep. 2018.

Même s'il accepte, le mot de passe *Facebook* de *Matt* ne sera en aucun cas connu par l'application *Bike to Work Day*.

Il existe deux versions d'*OAuth*, qui sont des spécifications complètement différentes. La version 2.0 est la plus utilisée de nos jours, c'est donc sur celle-ci que nous nous concentrerons.¹¹⁶

Avant *OAuth*, les sites web demandaient d'entrer le nom d'utilisateur et mot de passe directement dans un formulaire. *OAuth* a été créé en tant que réponse à ce type d'authentification, nommée authentification basique. Dans ce scénario, le client envoie des requêtes *HTTP* avec un en-tête contenant le mot "Basic" suivi d'un espace et d'une chaîne de caractères contenant "username:password" encodée en *base64*¹¹⁷.

Décrivons sommairement les trois intervenants principaux du processus d'autorisation avec *OAuth*.

- *Application* : il s'agit de l'application utilisateur ([FIGURE XXX](#), application *Bike to Work Day*). C'est elle qui demande l'accès à certaines ressources.
- *Serveur de ressources* : il possède les ressources auxquelles l'application utilisateur veut avoir accès ([FIGURE XXX](#), serveur *Facebook* contenant le profil et l'adresse e-mail de *Matt*).
- *Serveur d'autorisation* : c'est vers ce serveur que l'autorisation est déléguée (au lieu du [serveur de ressources](#)). Il a comme fonction d'autoriser l'*Application* à accéder au *Serveur de ressources* via un processus d'autorisation ([FIGURE XXX](#), serveur d'autorisation *Facebook*).

Éclaircissons à présent le fonctionnement d'*OAuth* à l'aide d'un exemple concret, tiré (et remanié) de la vidéo *OAuth 2.0 : An Overview*.¹¹⁸

MyBucks est une application de finances personnelles. Les utilisateurs accèdent via *MyBucks* aux données de leur compte bancaire *Memorial Bank*.

¹¹⁶ "What the Heck is OAuth? | Okta Developer." 21 Jun. 2017, <https://developer.okta.com/blog/2017/06/21/what-the-heck-is-oauth>. Accessed 15 Sep. 2018.

¹¹⁷ "Basic Authentication | Swagger." <https://swagger.io/docs/specification/authentication/basic-authentication/>. Accessed 15 Sep. 2018.

¹¹⁸ "OAuth 2.0 : An Overview." <https://www.youtube.com/watch?v=CPbvxxslDTU>. Accessed 15 Sep. 2018.

Nous pouvons résumer le processus d'autorisation *OAuth* en six étapes :

Workflow of OAuth 2.0

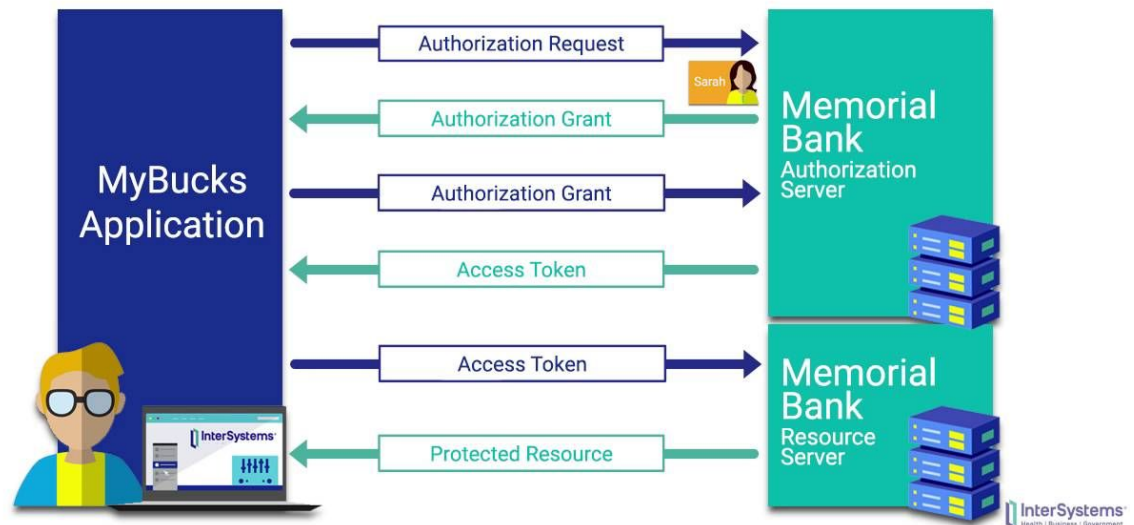


FIGURE XXXI : Workflow of OAuth 2.0

OAuth 2.0: An Overview, 2016

- (1) L'application *MyBucks* demande l'autorisation à l'utilisateur d'accéder à certaines données ([FIGURE XXXI](#)). Notons que dans bien des cas, comme dans la [FIGURE XXX](#), *Google* ou *Facebook* se retrouvent à la place de *Memorial Bank*.

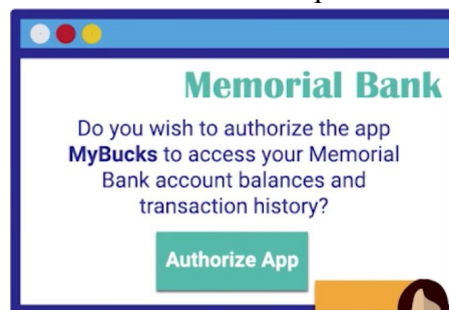


FIGURE XXXII : Autorisation demandée par l'application MyBucks

OAuth 2.0 : An Overview

- (2) L'utilisateur a autorisé l'application *MyBucks*, une preuve de cette autorisation est créée et renvoyée par le serveur d'autorisation à l'application *MyBucks*.
- (3) L'application *MyBucks* présente cette preuve au serveur d'autorisation afin d'obtenir un jeton d'accès pour les ressources désirées (soldes du compte et historique des transactions - [FIGURE XXXII](#)).
- (4) Le jeton d'accès est renvoyé par le serveur d'autorisation à l'application *MyBucks*.
- (5) Une requête est envoyée au serveur de ressources pour récupérer les soldes du compte et historique des transactions. Le jeton d'accès reçu à l'étape 4 est inclus dans cette requête.
- (6) Les ressources protégées sont envoyées à l'application *MyBucks*.

Pour que tout ceci fonctionne avec *OAuth*, le développeur devait au préalable enregistrer l'application *MyBucks* auprès du serveur d'autorisation de la *Memorial Bank*. En effet, le serveur de la *Memorial Bank* doit être informé du nom de l'application (*MyBucks*) et de l'*URL* (ex. *www.mybucks.com/home*) vers laquelle il doit rediriger l'utilisateur après qu'il ait autorisé l'application.

Notons qu'après l'étape illustrée dans la [FIGURE XXXIII](#), le serveur de la *Memorial Bank* renverra un *Client ID* et un *Client Secret* à *MyBucks* que l'application utilisera plus tard pour demander un jeton d'accès à une ressource (troisième flèche de la [FIGURE XXXI](#)).

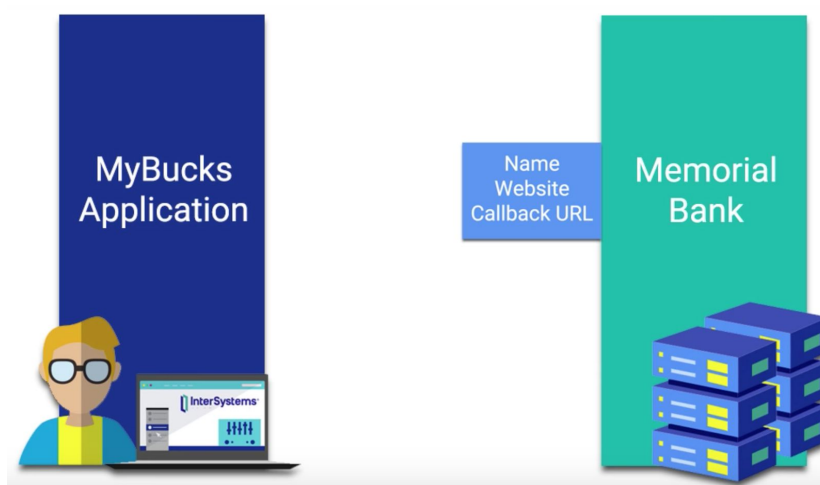


FIGURE XXXIII : Enregistrement de l'application MyBucks auprès du serveur
OAuth 2.0 : An Overview

Avec *OAuth* seul, nous parlions d'octroyer à l'utilisateur l'accès à une ressource. Il est tentant de voir cette autorisation (avec un jeton d'accès) comme de l'authentification car les deux processus ont généralement lieu en même temps.

Cependant, il est important de savoir que l'autorisation à une ressource (*OAuth*) et l'authentification de l'utilisateur (le fait de confirmer son identité) utilisent un mécanisme distinct.¹¹⁹ L'authentification est un concept étroitement lié à l'utilisateur, et *OAuth* n'envoie aucune information utilisateur à l'application - il n'est question que de jetons d'accès.

En effet, l'authentification se fait via des jetons d'identification qui sont passés avec les jetons d'accès (*Access Token*, [FIGURE XXXI](#)). Ces jetons d'authentification sont émis par un protocole d'authentification, pouvant être vu une couche au-dessus d'*OAuth*. Pour ce faire, *Firebase* utilise *OpenID Connect*.¹²⁰

"(...) This has led many developers and API providers to incorrectly conclude that OAuth is itself an authentication protocol and to mistakenly use it as such."

OAuth et *OpenID Connect* sont donc tous deux utilisés par le module *Firebase Authentication*.

¹¹⁹ "End User Authentication with OAuth 2.0 — OAuth." <https://oauth.net/articles/authentication/>. Accessed 16 Sep. 2018.

¹²⁰ "OpenID Connect | OpenID." <https://openid.net/connect/>. Accessed 15 Sep. 2018.

4.3 Backend as a Service (BaaS) et la plateforme Firebase

Note : En supplément des différentes sources référencées, les explications ci-après sont basées sur le site officiel de *Firebase*¹²¹ et sa documentation¹²².

4.3.1 Introduction

Dans un modèle de développement classique, lors de la conception d'une application mobile ou web, il est courant de développer un service web qui expose un ensemble d'*APIs*¹²³ à cette application.

En outre, il est souvent nécessaire de proposer dans ce service web des fonctionnalités telles que le stockage des données dans une base de données, les notifications ou encore la connexion avec les réseaux sociaux. Pour ce faire, chacune de ces fonctionnalités doit normalement être individuellement incorporée dans notre service web, un processus qui peut être coûteux en temps et compliqué pour les développeurs.¹²⁴

4.3.1.1 Backend as a Service

*“These platforms, often called mBaaS (mobile backend as a service) or mPaaS (mobile platform as a service) typically solve for the scalability and security concerns specific to enterprise deployment, freeing development teams to **focus on front end** user experience.”*¹²⁵

Backend as a Service (BaaS), ou *serverless* (ex. plus de serveurs, machines virtuelles à gérer), est un modèle qui répond au problème précédemment énoncé et vise à remplacer la conception d'un service web traditionnel, permettant un gain de temps et d'argent potentiel.

BaaS est un développement relativement récent dans le domaine du *cloud*, la plupart des startups *BaaS* datant de 2011 ou plus tard.¹²⁶ Les tendances indiquent cependant que ces services gagnent en popularité auprès des entreprises.¹²⁷

¹²¹ "Firebase." <https://firebase.google.com/>. Accessed 15 Sep. 2018.

¹²² "Documentation | Firebase - Google." <https://firebase.google.com/docs/>. Accessed 15 Sep. 2018.

¹²³ Application programming interface ou ensemble de fonctions exposées par un service.

¹²⁴ "Kinvey Raises \$5 Million For Mobile And Web App Backend As A" 11 Jul. 2012, <https://techcrunch.com/2012/07/11/kinvey-raises-5-million-for-mobile-and-web-app-back-end-as-a-service/>. Accessed 15 Sep. 2018.

¹²⁵ "Pando: AnyPresence partners with Heroku to beef up its enterprise" 24 Jun. 2013, <https://pando.com/2013/06/24/anypresence-partners-with-heroku-to-beef-up-its-enterprise-mbaas-offering/>. Accessed 15 Sep. 2018.

¹²⁶ "FatFractal ups the ante in backend-as-a-service market - Techgoondu" 30 Sep. 2012, <https://www.techgoondu.com/2012/09/30/fatfractal-ups-the-ante-in-backend-as-a-service/>. Accessed 15 Sep. 2018.

¹²⁷ "Mobile Backend as a Service Market Size, Share, Market Intelligence" 1 Jun. 2018, <https://www.marketwatch.com/press-release/mobile-backend-as-a-service-market-size-share-market-intelligence-company-profiles-and-trends-forecast-to-2023-2018-06-01>. Accessed 15 Sep. 2018.

Les plateformes *BaaS* proposent aux développeurs un accès simplifié et unifié aux fonctionnalités backend souvent utilisées¹²⁸ (qui seraient normalement intégrées une par une dans notre service web) en mettant à disposition :

- Des *APIs* permettant aux front ends d'accéder aux différentes fonctionnalités.
- Des *SDKs*¹²⁹, packages à intégrer dans les front ends qui facilitent les requêtes à ces *APIs*.
- Un moyen de gérer les différentes fonctionnalités back end offertes, typiquement au travers d'un site web.

Concrètement, dans nos applications web et mobiles, nous intégrerons des *SDKs* qui faciliteront l'envoi de requêtes *HTTP* (*GET*, *POST*) aux *APIs* exposées par ces plateformes *BaaS*. De cette manière, nous pourrions par exemple stocker des données dans une base de données *cloud*, nous authentifier ou encore envoyer des notifications.

La FIGURE XXXIV expose le principe de *Backend as a Service* ainsi que différents fournisseurs. Remarquons que l'avantage d'unification des *APIs* est mentionné.

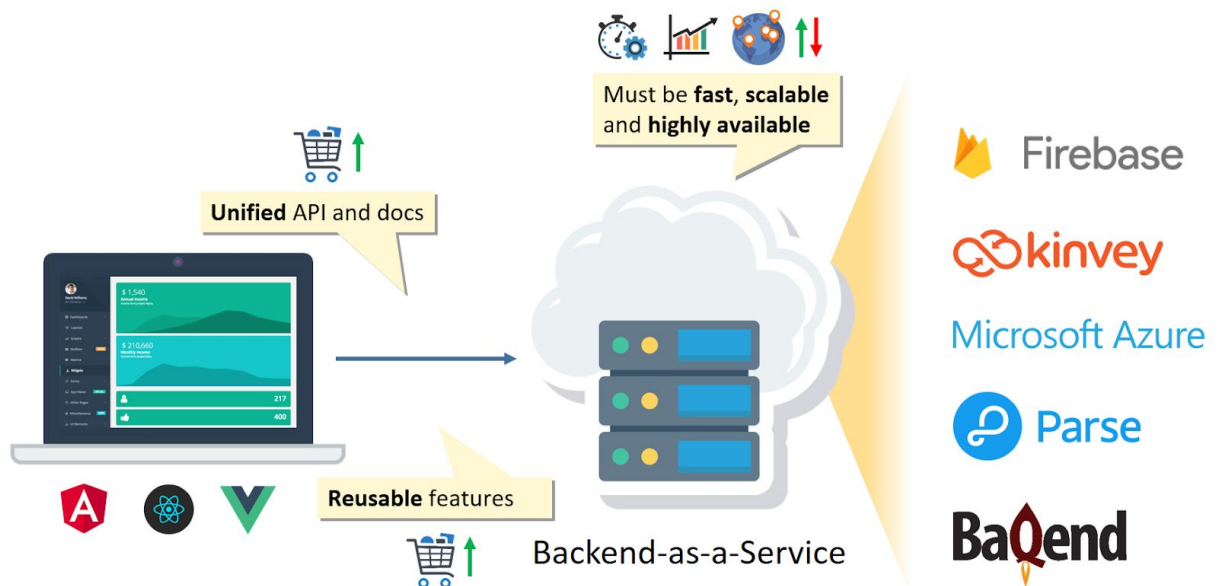


FIGURE XXXIV : Backend-as-a-Service¹³⁰

Lessons Learned Building a Backend-as-a-Service: A Technical Deep Dive, 2017

¹²⁸ "Rise of Mobile Backend as a Service (MBaaS) API Stacks." 3 Jun. 2012, <https://apievangelist.com/2012/06/03/rise-of-mobile-backend-as-a-service-mbaas-api-stacks/>. Accessed 15 Sep. 2018.

¹²⁹ "SDK (software development kit) - Gartner IT Glossary." <https://www.gartner.com/it-glossary/sdk-software-development-kit>. Accessed 15 Sep. 2018.

¹³⁰ "Lessons Learned Building a Backend-as-a-Service: A ... - Baqend Blog." 17 May. 2017, <https://medium.baqend.com/how-to-develop-a-backend-as-a-service-from-scratch-lessons-learned-a9fac618c2ce>. Accessed 15 Sep. 2018.

4.3.1.2 Choix d'utilisation de Firebase

"Build apps fast, without managing infrastructure."

Firebase est une plateforme *BaaS* de développement d'applications mobiles et Web développée par *Firebase, Inc.* en 2011, puis acquise par *Google* en 2014¹³¹. *Firebase* se positionne en tant qu'un des acteurs principaux de l'approche *BaaS*, son but est donc d'offrir aux clients la possibilité se concentrer sur la conception des front ends de leurs applications mobiles ou web.

Dans cette optique, *Firebase* propose des produits comme la *Firebase Realtime Database* ou encore *Firebase Authentication* que nous avons choisi d'élaborer dans ce rapport.

Notre choix s'est porté sur *Firebase* de par sa simplicité d'utilisation, sa croissante popularité^{132,133} et son utilisation très courante avec des *frameworks* web comme *Angular*¹³⁴, mais surtout car les fonctionnalités fournies par la *Firebase Realtime Database* correspondent totalement à notre cas d'utilisation. En effet, ces dernières facilitent la mise à profit des données temps-réel (d'où le terme "*Realtime*") et font de notre application un très bon candidat à la programmation réactive, tous deux piliers de ce travail de fin d'études.

La gestion des différentes fonctionnalités offertes par *Firebase* a lieu à partir de la console.

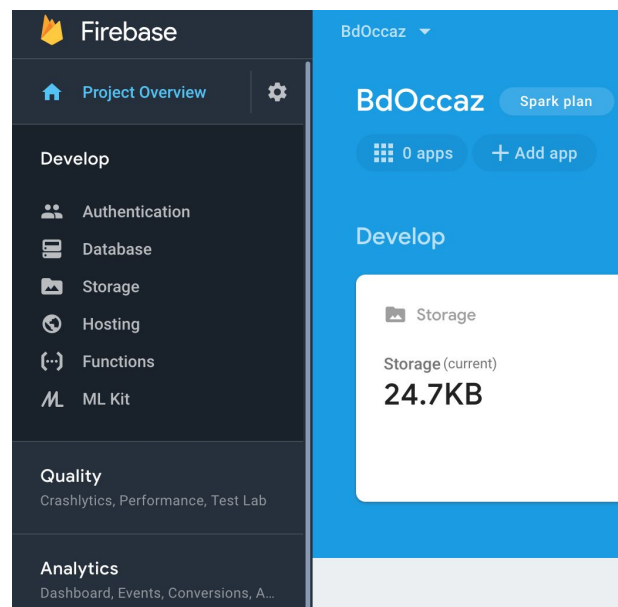


FIGURE XXXV : Console Firebase
Samih Alkeilani, 2018

¹³¹ "The Firebase Blog: Firebase expands to become a unified app platform." 18 May. 2016, <https://firebase.googleblog.com/2016/05/firebase-expands-to-become-unified-app-platform.html>. Accessed 15 Sep. 2018.

¹³² "Google Owns 100% Of 2017's Top 10 Fastest-Growing SDKs ... - Forbes." 19 Dec. 2017, <https://www.forbes.com/sites/johnkoetsier/2017/12/19/google-owns-100-of-2017s-top-10-fastest-growing-sdks-on-android/>. Accessed 15 Sep. 2018.

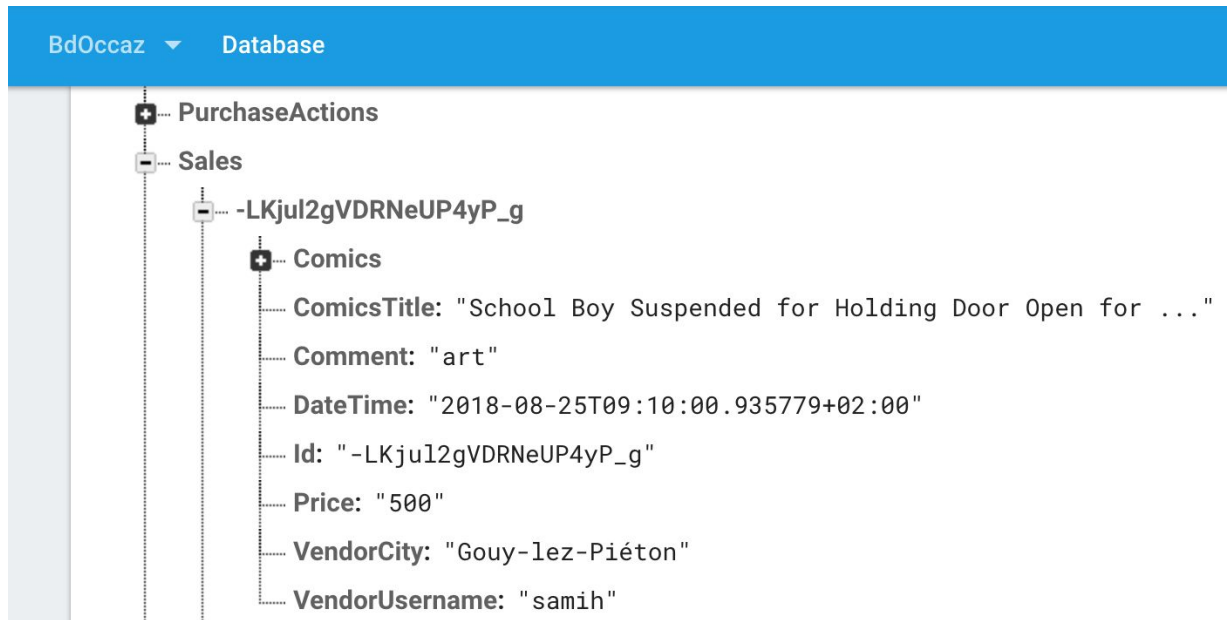
¹³³ "DB-Engines Ranking - Trend of Firebase Realtime Database Popularity." https://db-engines.com/en/ranking_trend/system/Firebase+Realtime+Database. Accessed 15 Sep. 2018.

¹³⁴ "AngularFirebase: Learn Angular and Firebase with Videos and" <https://angularfirebase.com/>. Accessed 15 Sep. 2018.

4.3.2 Firebase Realtime Database

4.3.2.1 Généralités

La *Firebase Realtime Database* est une base de données *NoSQL* de type document hébergée dans le *cloud* et administrée via la [console Firebase](#). Les données y sont stockées sous forme de documents *JSON*¹³⁵.



**FIGURE XXXVI : Structure de données JSON représentée dans la console
Firebase**

Samih Alkeilani, 2018

Lors de l'ajout de données, un nouveau noeud est créé avec celles-ci dans la structure existante. Comme vu dans le [point 4.1.4](#) sur la dénormalisation, quand il s'agit du paradigme *NoSQL*, nous adopterons des pratiques différentes à celles mises en place dans une base de données relationnelle.

Précisons qu'à défaut d'écrire un service web, *Firebase* propose les *Cloud Functions*, permettant de réagir à un événement ayant eu lieu dans la base de données. Ainsi, nous pouvons exécuter automatiquement du code *back-end* en réponse à des événements et des requêtes *HTTPS*. Le code des *Cloud Functions* est stocké dans le *cloud* de *Google* et s'exécute dans cet environnement - il n'est pas nécessaire de gérer et d'adapter nos propres serveurs.¹³⁶

¹³⁵ "JSON." <https://www.json.org/>. Accessed 15 Sep. 2018.

¹³⁶ "Cloud Functions for Firebase - Google." <https://firebase.google.com/docs/functions/>. Accessed 16 Sep. 2018.

4.3.2.2 Monter à l'échelle avec Firebase Realtime Database

Nous avons vu dans notre introduction au *NoSQL*, plus spécifiquement quand nous parlions des bases de données distribuées, que les solutions *NoSQL* ont souvent tendance à monter à l'échelle de manière horizontale quand le nombre de requête augmente (ajouter d'autres instances).

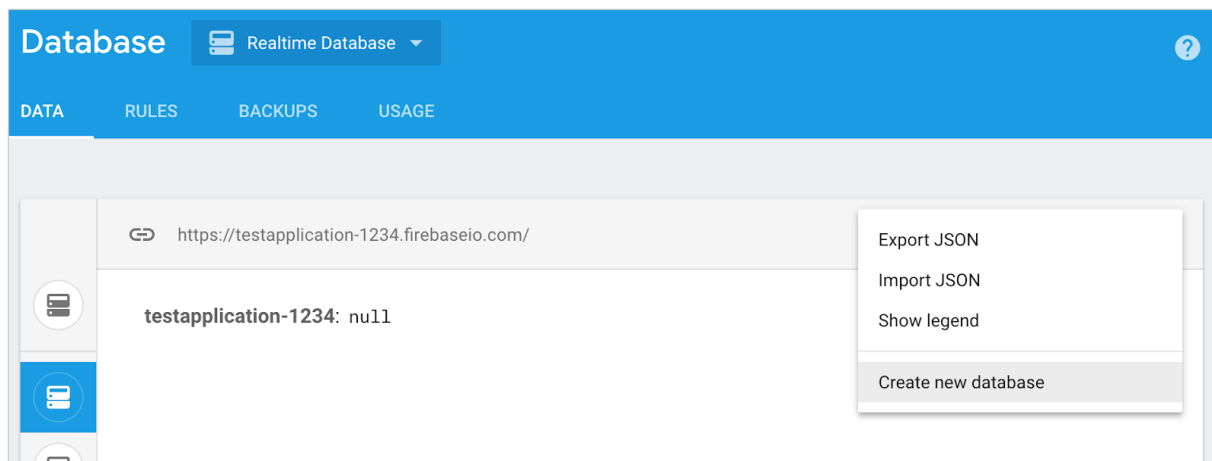
La *Firebase Realtime Database* ne fait pas exception à la règle, adoptant un tel modèle. Avec une seule instance de la base de données, nous sommes par exemple, limités à 100.000 connections simultanées (une connection est équivalente à un appareil mobile connecté) et à ~100.000 réponses simultanées par seconde venant de cette base de données.¹³⁷

Pour aller plus loin, il est nécessaire monter à l'échelle de manière en séparant nos données sur plusieurs instances de la *Firebase Realtime Database*, également connu sous le nom de partitionnement de base de données (*database sharding*).¹³⁸ Le *database sharding* nous offre en effet la flexibilité nécessaire pour évoluer au-delà des limites que nous venons d'énoncer.

Lors de la mise en place de cette technique, nous veillerons à satisfaire les conditions suivantes :

- Chaque requête ne s'exécute que sur une seule instance de base de données.
- Pas de partage ou de duplication des données entre les instances de la base de données (ou minimal).
- Chaque instance d'application se connecte uniquement à une base de données à un moment donné.

Nous pouvons voir dans l'illustration suivante qu'il est possible de créer plusieurs instances à partir de la console :



Firebase Realtime Database, création de plusieurs instances

¹³⁷ "Realtime Database Limits - Firebase - Google."

<https://firebase.google.com/docs/database/usage/limits>. Accessed 18 Sep. 2018.

¹³⁸ "Scale with Multiple Databases | Firebase Realtime Database ... - Google."

<https://firebase.google.com/docs/database/usage/sharding>. Accessed 18 Sep. 2018.

4.3.2.3 Structures de données imbriquées et plates

Dans le contexte de la *Realtime Database*, il est préférable d'éviter l'imbrication de données, car récupérer les données d'un noeud particulier implique également la récupération de tous ses noeuds enfants. Au même titre, lorsqu'un accès (en lecture ou en écriture) est octroyé sur un noeud, il l'est aussi pour tous les noeuds enfants.

Considérons l'exemple suivant :

EXTRAIT DE CODE IV

Exemple de structure de données imbriquée

```
{
  "chats": {
    "one": {
      "title": "Historical Tech Pioneers",
      "messages": {
        "m1": { "sender": "ghopper", "message": "Relay. Cause: moth." },
        "m2": { ... },
        // Liste très longue de messages
      }
    },
    "two": { ... }
  }
}
```

Avec cette structure, l'itération sur les enfants du noeud “chats” provoque la récupération de tous les messages de chaque “chat”, même si nous n'étions intéressés que par leur “title”.

Voici à présent une séparation des données de manière plus plate, comme recommandé par la documentation de *Firebase* concernant la structure des données¹³⁹ :

EXTRAIT DE CODE V

Exemple de structure de données plate

```
{
  "chats": {
    "one": {
      "title": "Historical Tech Pioneers",
      "lastMessage": "ghopper: Relay malfunction found. Cause: moth.",
      "timestamp": 1459361875666
    },
    "two": { ... },
    "three": { ... }
  },
  "messages": {
    "m1": {
      "name": "eclarke",
      "message": "The relay seems to be malfunctioning.",
      "timestamp": 1459361875337,
      "chatId": "one"
    },
    "m2": { ... }
  }
}
```

Comme nous pouvons le constater, chaque “chat” ne contient maintenant plus que les informations relatives à la conversation en question, les messages étant stockés dans un noeud séparé. Il est dorénavant possible d’itérer rapidement à travers les enfants du noeud “chats” et de récupérer le noeud “title” de chaque “chat” sans devoir récupérer les “messages” de chaque conversation.

¹³⁹ "Structure Your Database | Firebase Realtime Database ... - Google." 16 Aug. 2018, <https://firebase.google.com/docs/database/ios/structure-data>. Accessed 15 Sep. 2018.

La portée des noeuds

Au delà de ce problème, avec l'[EXTRAIT DE CODE IV](#), il nous serait impossible d'effectuer une requête sur les "messages" séparément du noeud "chats" (ex. récupérer tous les messages où le "message" est...), tout simplement car les messages n'existent pas en dehors du contexte de chaque "chat". Cette notion est également nommée la portée d'un noeud.

Dans l'[EXTRAIT DE CODE V](#), nous avons changé la portée du noeud "messages", la positionnant en tant que collection à la racine, ou *root collection*.¹⁴⁰ Cela rend la requête précédemment mentionnée (ex. récupérer tous les messages où le "message" est...) possible étant donné que les "messages" existent maintenant indépendamment des "chats".

Afin de savoir s'il est nécessaire de placer une collection à la racine, la question à se poser est : *allons-nous avoir besoin d'interroger ces données sur des documents parents ?*

Autrement dit, si dans l'[EXTRAIT DE CODE IV](#), nous devons récupérer "tous les messages où le sender est...", il est nécessaire d'en faire une collection à la racine.

Validation des données

La base de données *Firebase Realtime Database* est dite [schema-less](#). Cela nous offre la flexibilité d'en changer la structure au fur et à mesure de notre développement, mais il est aussi important que nos données restent consistantes. Il nous est alors possible d'écrire des règles de validation prévues à cet effet.

EXTRAIT DE CODE VI

*Règle imposant que les données écrites dans /foo/
soient une chaîne de moins de 100 caractères*

```
{
  "rules": {
    "foo": {
      ".validate": "newData.isString() && newData.val().length < 100"
    }
  }
}
```

¹⁴⁰ "Model Relational Data in Firestore NoSQL - YouTube." 9 Feb. 2018,
<https://www.youtube.com/watch?v=jm66TSIVtcc>. Accessed 15 Sep. 2018.

Indexes

Afin de pouvoir exécuter des requêtes sur la base de données où un filtre est appliqué (clause “where” - ex. récupérer tous les noeuds “Sales” OÙ le “ComicsTitle” est...), nous le renseignerons à *Firebase* en plaçant un index (`.indexOn`) sur ce noeud à partir de la console.

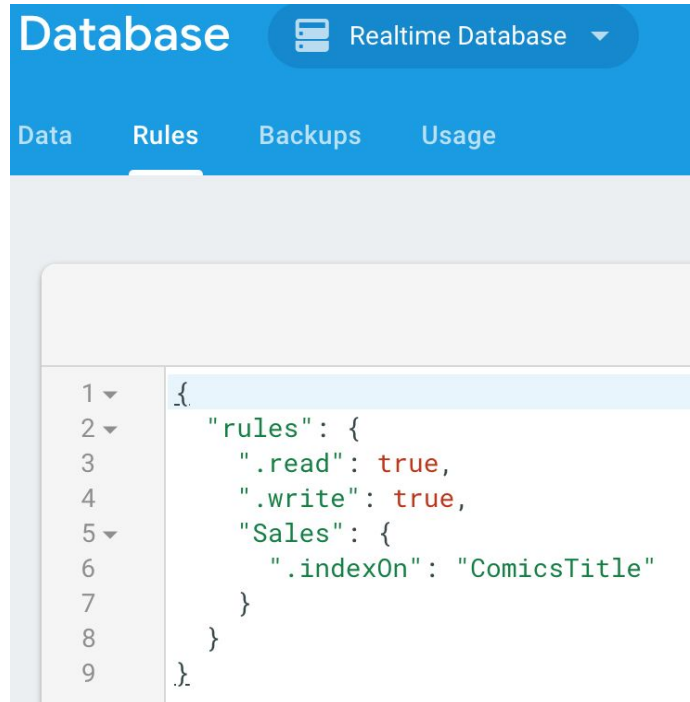


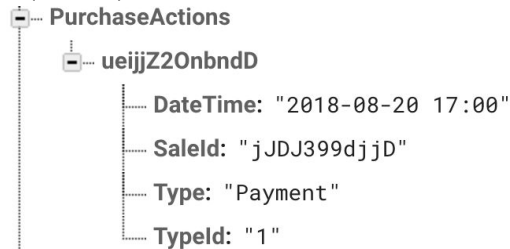
FIGURE XXXVII : Index sur un noeud dans la console Firebase

Samih Alkeilani, 2018

4.3.2.4 Relations entre documents

Bien que celles-ci soient gérées différemment des bases de données relationnelles, il est possible d'exprimer avec la *Firestore Realtime Database (NoSQL)* des relations entre plusieurs documents. Il existe plusieurs manières de procéder, voici quelques exemples tirés de notre cas pratique présenté au chapitre 6 :

Une vente est liée à plusieurs actions d'achat (l'utilisateur paye, l'article est livré...), et une action d'achat ne peut être liée qu'à une et une seule vente. Une action d'achat reprend alors l'identifiant de la vente (*SaleId*).



Représentation d'une relation one-to-many

Un utilisateur peut mettre en favoris plusieurs ventes, et une vente peut être mise en favoris par plusieurs utilisateurs. Une collection (ex. table intermédiaire) *Favorites* est donc créée et reprend l'identifiant de l'utilisateur (*Username*) ainsi que celui de la vente (*SaleId*).



Représentation d'une relation many-to-many

Terminons par illustrer la notion de dénormalisation. Les *Comics* existent dans une collection à part mais nous voyons ici que les données sont dupliquées au sein de *Sales* : en effet, lors de la récupération d'une *Sale* dans notre interface utilisateur, nous avons très souvent besoin de la *Comics* qui y est liée (en dupliquant, nous évitons un appel supplémentaire pour récupérer les données de *Comics*).



Dénormalisation

4.3.2.5 Temps-réel

La *Firebase Realtime Database* possède un mécanisme intégré pour le stockage et la synchronisation des données en temps réel.

L'idée ici est que notre application réagisse à une notification d'événement envoyée par *Firebase* (signalant que de nouvelles données sont disponibles). Cela va de pair avec le coeur de ce travail et signifie que nous disposons d'un moyen pour récupérer les dernières données (suite à une modification...) de notre *Firebase Realtime Database*. Nous exploiterons ensuite ces dernières dans notre application grâce aux techniques vues dans le [chapitre 3](#).

Soulignons enfin l'importance de cette dimension temps-réel en mentionnant le nombre important d'applications sur le marché à l'heure actuelle. Au vu de ce panel de choix étendu, il est de notre point de vue essentiel que l'utilisateur puisse interagir avec les dernières données à leur apparition, car c'est à cet instant qu'elles ont généralement le plus de pertinence.

En bref, afin de bénéficier du mécanisme temps-réel de la *Firebase Realtime Database*, notre application observe les changements qui ont lieu dans la base de données et y réagit en utilisant la programmation réactive.

4.3.3 Firebase Authentication

Introduction

La plupart des applications doivent connaître l'identité d'un utilisateur. Connaître l'identité d'un utilisateur permet à une application d'enregistrer de manière sécurisée les données utilisateur dans le *cloud* et pour, par exemple, offrir la même expérience personnalisée sur tous les appareils de l'utilisateur.

Firebase Authentication fournit des *SDKs* pour authentifier les utilisateurs d'une application. Il prend en charge l'authentification à l'aide de mots de passe, de numéros de téléphone et d'autres fournisseurs d'identités populaires tels que *Google*, *Facebook* et *Twitter*, etc.

Firebase Authentication s'intègre avec les autres services *Firebase* (*Realtime Database*,...) et s'appuie sur les normes du secteur : *OAuth 2.0* et *OpenID Connect* (discutés au point [4.2 Introduction à l'authentification](#)).

Fonctionnement

Pour effectuer la connexion d'un utilisateur dans notre application, nous obtiendrons d'abord les informations d'authentification de l'utilisateur.

Ces informations d'identification peuvent être l'adresse électronique et le mot de passe de l'utilisateur, ou un jeton *OAuth* provenant d'un fournisseur d'identité. Ensuite, nous transmettons ces informations d'identification au *Firebase Authentication SDK*.

Les services back-end de *Firebase* vérifieront alors ces identifiants et retourneront une réponse. Après une connexion réussie, nous pourrions accéder aux informations de base de l'utilisateur et contrôler ses accès aux données stockées dans d'autres produits *Firebase*.

Par défaut, les utilisateurs authentifiés peuvent lire et écrire dans la *Firebase Realtime Database*. Nous pouvons cependant modifier ce comportement en changeant les règles de la base de données depuis la console *Firebase*, comme exposé à la [FIGURE XXXVII](#).

5 Chapitre V : Application multi-plateformes réactive avec Xamarin

5.1 Introduction

Xamarin est un logiciel et *framework* utilisé pour développer des applications multi-plateformes natives iOS, Android et Windows à l'aide du langage de programmation C#.

5.1.1 Choix d'utilisation de Xamarin

Le choix de *Xamarin* repose sur d'abord le fait qu'il s'agit à l'heure actuelle d'une des solutions les plus abouties dans le domaine. Nous souhaitons deuxièmement exposer la programmation réactive à travers de multiple plateformes, possibilité que nous offre l'approche *Xamarin* avec une ré-utilisation intéressante du code existant. Nous avons donc :

- La possibilité de développer des applications pour plusieurs plateformes à partir d'un seul langage. Cela nécessiterait normalement l'utilisation du langage *Swift* pour *iOS*, *Java/Kotlin* pour *Android* et *C#* pour *Windows*.
- La réutilisation et partage d'un maximum de code entre les différentes plateformes. Concrètement, tout ce qui concerne la logique métier et les interfaces utilisateur¹⁴¹ peut être facilement réutilisé (voir *Shared C#* - [FIGURE XXXVIII](#)). Certains accès systèmes (connexion à une imprimante Bluetooth...) ou la personnalisation de certains composants nécessitent parfois d'écrire ce code pour chaque plateforme mais il s'agit toujours du C# (voir *Platform-Specific C#* - [FIGURE XXXVIII](#)).
- La réduction du temps de développement par rapport à une approche où il serait nécessaire de réécrire totalement l'application pour chaque plateforme.

¹⁴¹ La librairie *Xamarin.Forms* expose une trousse à outil complète pour développer des interfaces utilisateur en C# et XAML.

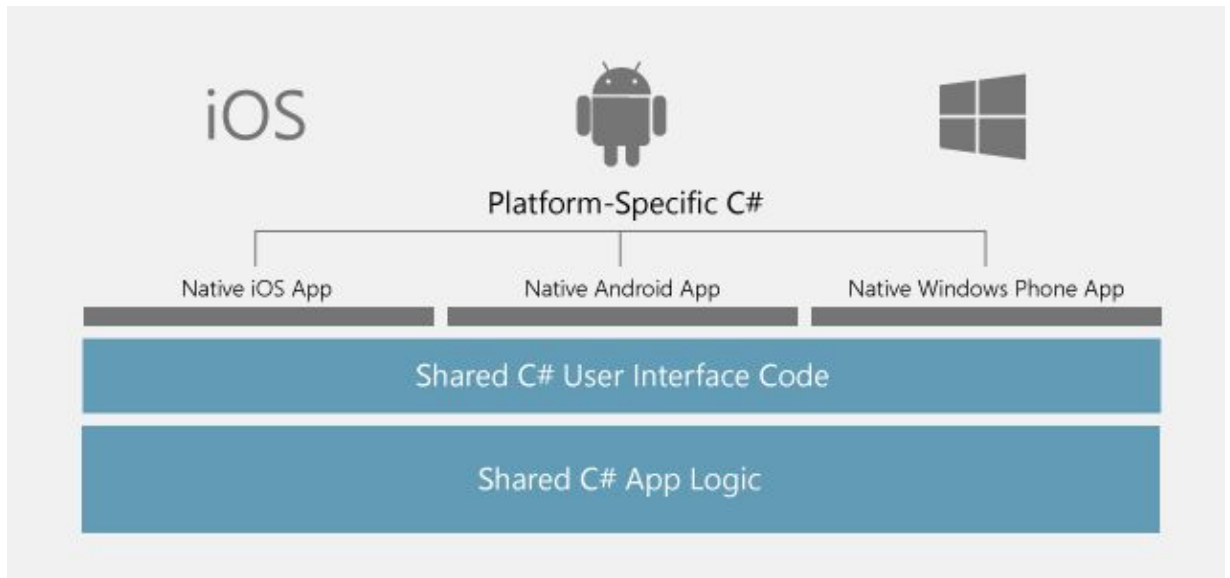


FIGURE XXXVIII : Single shared codebase for Android, iOS and Windows¹⁴²
Microsoft, 2018

¹⁴² "Xamarin - Visual Studio - Microsoft." 29 Aug. 2018, <https://visualstudio.microsoft.com/xamarin/>. Accessed 17 Sep. 2018.

5.2 Environnement de programmation

L'éditeur de code utilisé est *Visual Studio* pour *Mac*.

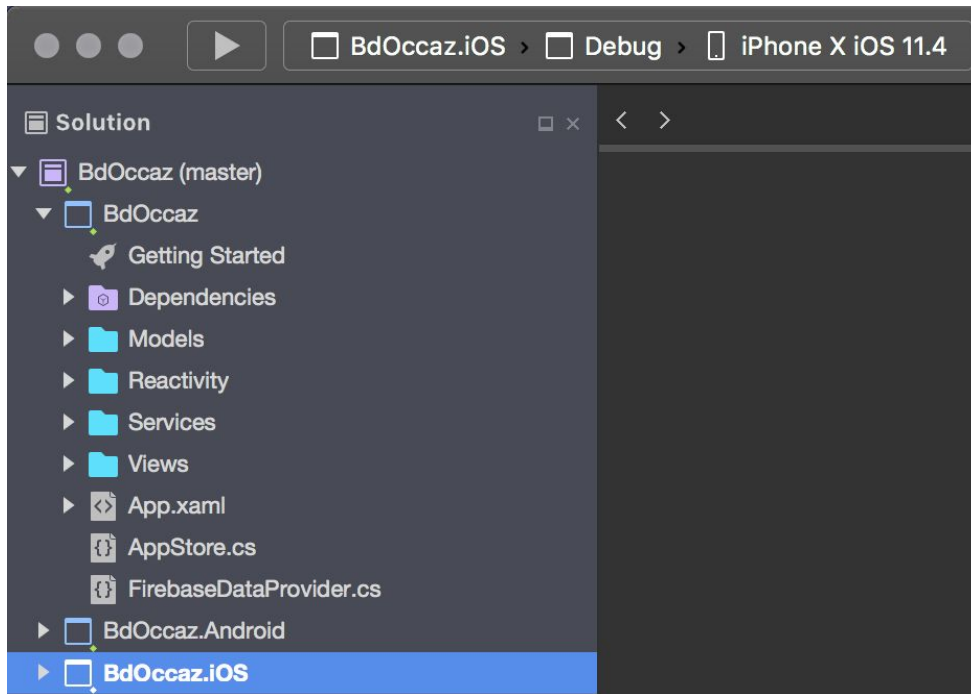


FIGURE XXXIX : Solution de l'application BdOccaz sur Visual Studio

Samih Alkeilani, 2018

À la [FIGURE XXXIX](#), nous pouvons voir trois projets au sein de la solution applicative *BdOccaz*. Ces derniers sont respectivement *BdOccaz*, *BdOccaz.Android* et *BdOccaz.iOS*.

Le projet *BdOccaz* (librairie *.NET Standard*) contient l'entièreté du code partagé et réutilisable entre les différentes plateformes (*iOS*, *Android* et *Windows* dans notre cas). C'est dans ce projet que nous définirons notre logique métier (dossiers *Models* et *Services* à la [FIGURE XXXIX](#)) et nos interfaces utilisateur (dossier *Views*, [FIGURE XXXIX](#)).

Les autres projets contiendront la logique spécifique aux plateformes qu'il n'est pas possible de partager.

Pour plus de détails sur les librairies utilisées, voir [9.1 Stack technologique](#).

5.3 Architecture réactive envisagée

C'est ici que nous allons assembler les différents concepts de programmation réactive vus au [Chapitre 3 - Architecture générale](#) et examiner leur implémentation. Pour rappel, ces derniers sont, du plus haut niveau (interaction entre les différents composants) au plus bas niveau (programmation au sein d'un composant) :

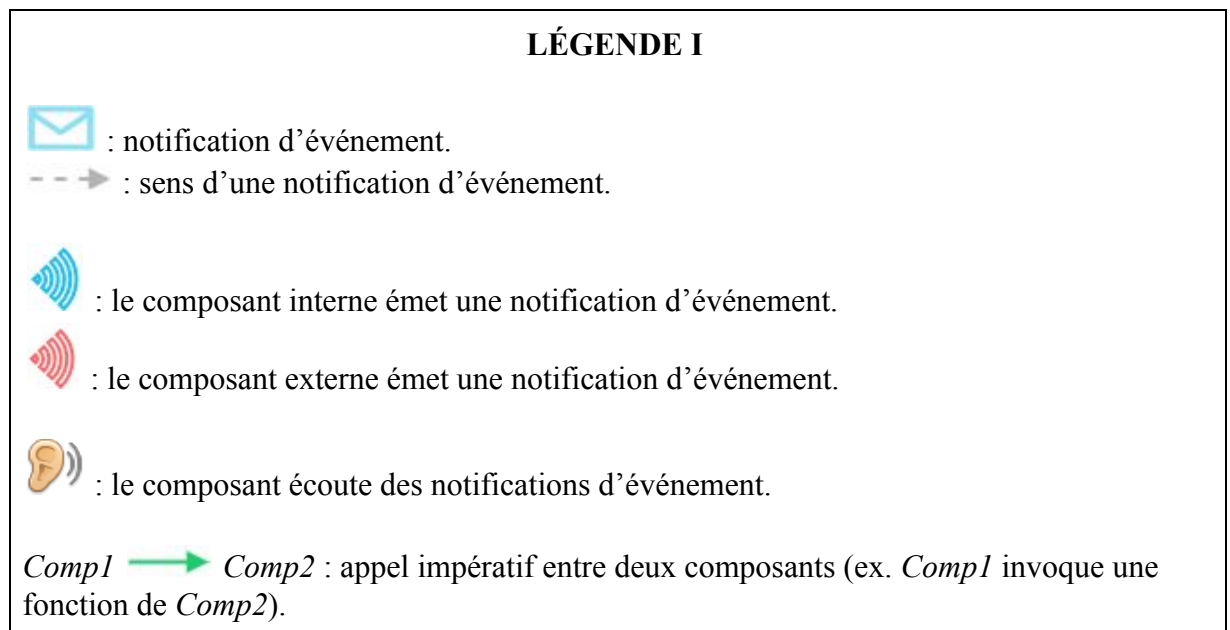
- La notion architecturale de flux de données unidirectionnel abordée au [point 3.4](#).
- La programmation basée sur les événements, présentée au [point 3.1](#).
- Les *Reactive Extensions*, au [point 3.3](#).
- Le patron de conception *Observer* au [point 3.2](#).

Notons que les différents extraits de code ci-après sont tirés de l'application *Xamarin BdOccaz*, dans les langages *C#* et *XAML* (conception les interfaces utilisateur).

Commençons par représenter schématiquement l'interaction de l'application *BdOccaz* avec notre base de données *Firebase Realtime Database* à la [FIGURE XL](#). Afin de bénéficier du mécanisme temps-réel, notre application "écoute" les changements qui ont lieu dans la base de données.

5.3.1 Légende des interactions entre composants

Les composants sont des parties d'applications, techniquement il s'agit le plus souvent de *classes*.



5.3.2 Interaction entre l'application et le back-end

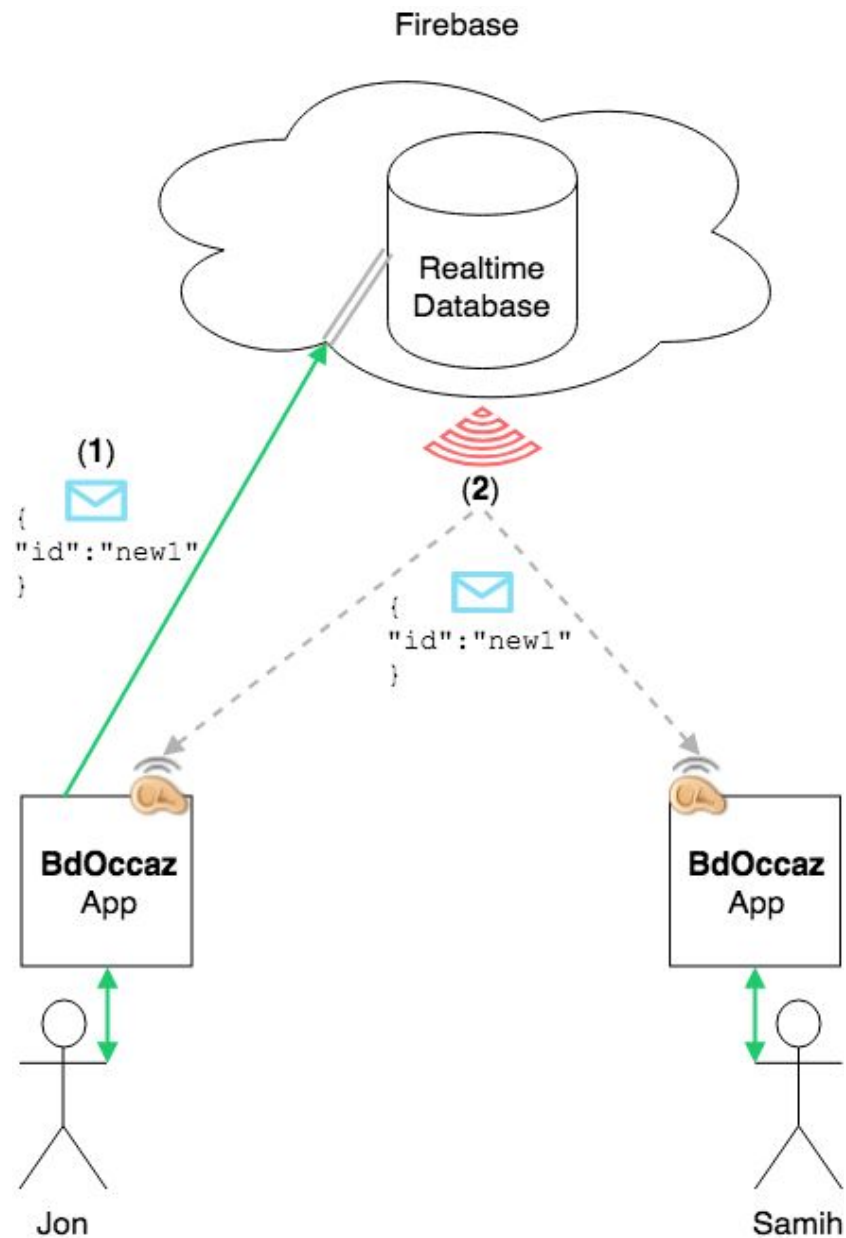


FIGURE XL : BdOccaz “écoute” Firebase
Samih Alkeilani, 2018

- (1) Un nouvel élément est inséré par *Jon* dans la base de données.
- (2) Les abonnés aux changements (*Samih* et *Jon* dans ce cas) reçoivent la modification en temps réel.

5.3.3 Architecture réactive de l'application

5.3.3.1 Schématisation

Après l'interaction entre l'application *BdOccaz* et notre back-end (*Firebase*), regardons de plus près l'architecture applicative de *BdOccaz* en nous appuyant sur la [FIGURE XLI](#). Nous pouvons voir comment notre architecture propage une notification d'événement du composant *Service* qui écoute notre back-end à la *View*.

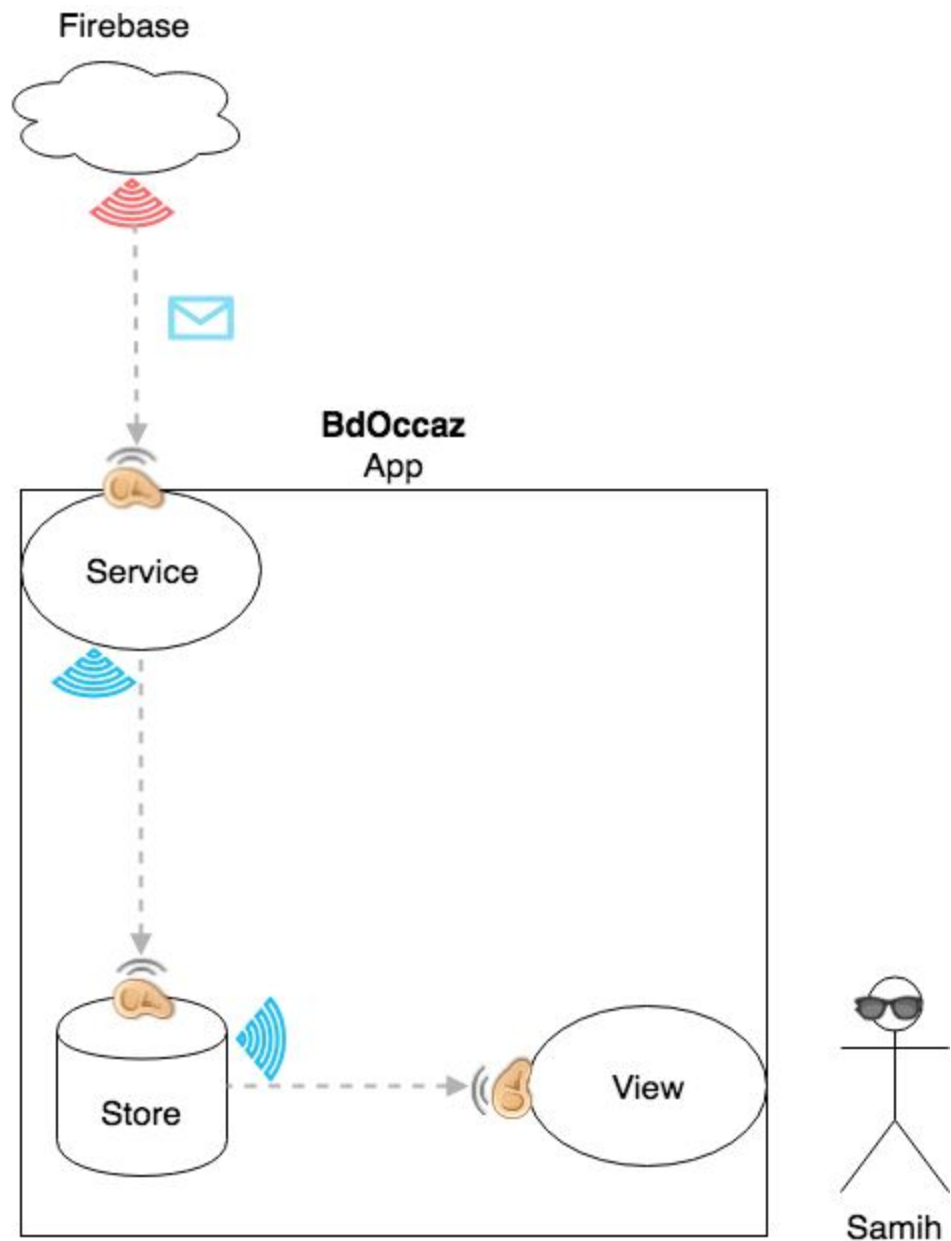


FIGURE XLI : Propagation de la notification d'événement dans l'application
Samih Alkeilani, 2018

Passons à la description des trois grands intervenants de la [FIGURE XLI](#) en constatant certaines similitudes avec l'architecture web *Akita* présentée à la [FIGURE XXII](#).

Store ou source unique de vérité

- Contient l'état et données de l'application.
- Écoute les changements des *services* (agit en tant qu'observer) et met à jour les données (qui sont d'ailleurs les siennes) suite à ces derniers.
 - Le *store* est le seul composant qui peut modifier ses données, et il encapsule cette logique de mise à jour grâce à des *setters* privés.
- Notifie les *Views* suite aux changements (agit en tant qu'observable).

Views

- Gèrent les interactions avec l'utilisateur.
- Écoutent les changements du *store* (agit en tant qu'observer) pour les refléter.

Services

- Contiennent la logique métier, ex. appel à des services web...
- Écoutent des changements externes à l'application (agit en tant qu'observer).
- Notifient le *store* des changements reçus (agit en tant qu'observable).

Ajoutons à la [FIGURE XLII](#) l'élément manquant au schéma précédent en représentant la manière dont notre architecture gère l'action d'un utilisateur.

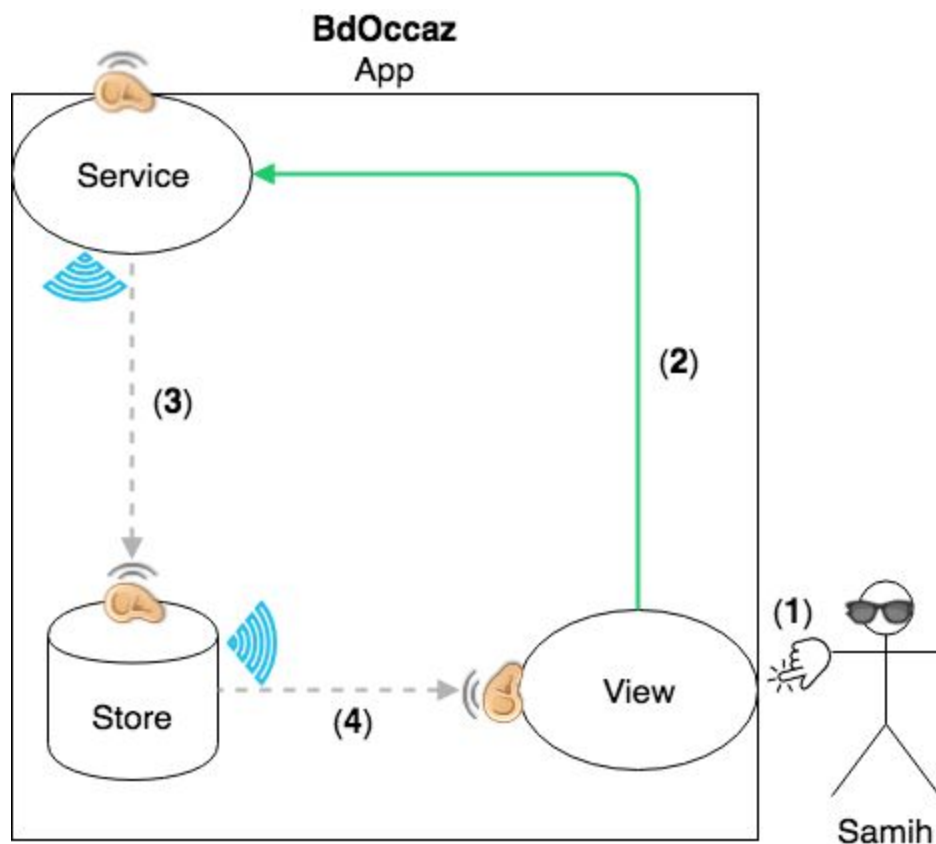


FIGURE XLII : Flux de l'application suite à une action utilisateur
Samih Alkeilani, 2018

Notons une certaine cohérence dans la façon de gérer les actions de différentes sources.

- Si une notification d'événement provient du back-end, cette dernière est propagée jusqu'à la *View* par l'intermédiaire du *Service*.
- Si une notification d'événement provient de la vue (*View*), une fonction du service est invoquée et tout le reste du flux est identique : le *service* peut par exemple insérer une nouvelle donnée et il recevra ensuite la notification du changement qui sera propagée de la même manière.

Cette architecture nous offre un flux de données unidirectionnel (sens des flèches), ce qui facilite le suivi des différents changements dans le code.

Le second avantage est que tous les changements d'états de notre application ont lieu à un seul endroit, plus précisément là où le *store* écoute les changements des *services* et y réagit. Concrètement, cela signifie que si une vue affiche des données incorrectes ou inconsistantes, nous saurons que le problème vient exclusivement de cet emplacement.

5.3.3.2 Dépendances

Nous avons introduit les notions de couplage et de dépendances entre modules [au début de ce travail](#), lors de l'étude des événements. Nous allons maintenant décrire les relations entre les différents composants de l'architecture.

Dépendances du *store*

- *Services* : le *store* connaît les différents services car il réagit à leurs notifications d'événements pour se mettre à jour.

Dépendances des *views*

- *Service(s)* : une vue possède la référence du/des service(s) avec qui elle traite, car c'est par eux que passent toutes les modifications de données.
- *Store* : les vues reflètent les données du *store* (mais n'ont pas la possibilité de les modifier).

Dépendances des *services*

- Service(s) externe(s) : les *services* ne possèdent aucune référence envers le *store* ou les *views* - leur rôle est simplement d'exécuter de la logique et d'émettre des notifications auxquelles le *store* réagit. Précisons cependant qu'il aura une dépendance envers un potentiel service externe qui fournit les nouvelles données (dans notre cas, *Firebase*).

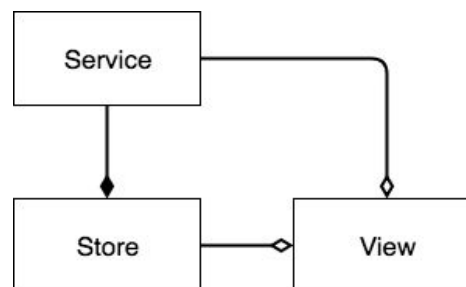


FIGURE XLIII : Dépendances entre les composants principaux
Samih Alkeilani, 2018

5.3.4 Programmation réactive

5.3.4.1 Du back-end aux services

Analysons l'[EXTRAIT DE CODE VII](#) qui permet à l'application d'écouter les changements venant du back-end en temps réel :

EXTRAIT DE CODE VII

Souscription à l'événement de Firebase

```
var subscription = FirebaseService.Client?  
    .Child("Sales")  
    .AsObservable<Sale>()  
    .Subscribe(OnSalesChangedFromFirebase);
```

- Nous rencontrons les termes *Observable*, *Subscribe* et pouvons en déduire que nous avons affaire au patron de conception *Observer* décrit [dans le point 3.2](#).
- Nous avons vu qu'il était possible de créer des observables à partir de [différentes sources](#). En l'occurrence, grâce à la fonction *AsObservable<T>*, un observable est créé à partir d'un objet de type *FirebaseEvent<T>* qui représente simplement le type de l'événement de *Firebase*. Nous pouvons ensuite traiter cet événement comme une *collection arrivant au fil du temps* et y appliquer des opérateurs (filtres...) en utilisant les *Reactive Extensions* comme montré dans l'[EXTRAIT DE CODE VIII](#).
- L'appel à la fonction *Subscribe* provoque le début de l'écoute (création du *stream*) des modifications en temps réel dans notre *Firebase Realtime Database*.
- La fonction *OnSalesChangedFromFirebase* passée en paramètre à la fonction *Subscribe* de l'observable, est notre [observateur](#). C'est cette dernière qui sera exécutée lors de la réception d'une notification de *Firebase*.

EXTRAIT DE CODE VIII

Souscription à l'événement de Firebase et utilisation des Reactive Extensions pour filtrer les notifications d'événements reçues

```
var subscription = FirebaseService.Client?  
    .Child("Sales")  
    .AsObservable<Sale>()  
    .Where(saleEvent => saleEvent.Key == "MyFilter") //Filtre  
    .Subscribe(OnSalesChangedFromFirebase);
```

L'opérateur *Where* est l'équivalent de *Filter*, décrit au point [3.3.3 Opérateurs](#).

5.3.4.2 Des services au store

Ici, le *store* souscrit à l'événement **Changed** du service `salesService` pour en recevoir les notifications :

EXTRAIT DE CODE IX

Souscription du store à l'événement du service

```
var subscription =
Observable.FromEventPattern<EventHandler<CollectionChangedEventArgs>,
CollectionChangedEventArgs>(
    ev => salesService.Changed += ev,
    ev => salesService.Changed -= ev).Subscribe(x =>
{
    ...
    UpdateCollectionFromChanges(x); //Le store met à jour ses données
});
```

Dans l'[EXTRAIT DE CODE IX](#), nous utilisons les *Reactive Extensions* pour transformer l'événement **Changed** de `salesServices` en objet de type observable grâce à la fonction `FromEventPattern`. Ceci nous permet, si nous le souhaitons, d'utiliser divers opérateurs sur les notifications d'événements (comme dans l'[EXTRAIT DE CODE VIII](#)), mais notons également l'avantage suivant :

Dans notre cas, il est probable que des notifications de *Firebase* et de l'interface utilisateur arrivent parfois en simultané. Les *Reactive Extensions* fournissent également une gestion de la concurrence qui constitue une raison supplémentaire d'ajouter la 'couche observable' par dessus nos événements.

Le *service* notifie ensuite ses observers (dans notre cas, le *store*) suite à un changement reçu de *Firebase* en déclenchant son événement **Changed** :

EXTRAIT DE CODE X

Le service réagit au changement de Firebase déclenchant son événement Changed, notifiant ainsi ses observers

```
//Cette fonction est notre observer, invoquée quand changement arrive de Firebase
void OnSalesChangedFromFirebase(FirebaseEvent<Sale> newData)
{
    ...
    switch (newData.EventType)
    {
        case FirebaseEventType.InsertOrUpdate:
            OnChanged(newData); //Envoi de la notification avec nouvelles données
            break;
        case FirebaseEventType.Delete:
            ...
    }
}
```

5.3.4.3 Du store aux vues

En ce qui concerne la propagation des changements du *store* à la vue, nous tirerons profit du système de **Binding** fourni par la librairie *Xamarin.Forms*. Ce mécanisme est largement utilisé en *Xamarin* et d'autres librairies comme *WPF*.

Le mécanisme nous permet de donner un contexte à notre vue (**BindingContext**) et à partir de celui-ci, de renseigner les propriétés pour lesquelles elle doit écouter les changements (**Binding**).

Notre compréhension des concepts sous-jacents nous permet de réaliser que ce mécanisme est également une encapsulation (*C# .NET*) du patron de conception *Observer*. En effet, nous venons de décrire la façon dont la vue (observer) souscrit aux changements d'un observable (dans notre cas, le *store*).

Pour en revenir aux termes spécifiques, nous implémenterons dans notre *store* l'interface **INotifyPropertyChanged** qui dans son contrat possède l'événement **PropertyChanged**. Nous déclencherons cet événement lorsqu'un membre du *store* est modifié, ce qui aura pour effet d'envoyer une notification aux composants (vues) souscrits à ce changement.

Premièrement, les vues s'abonnent aux changements du store.

EXTRAIT DE CODE XI

Paramétrage du contexte dans le constructeur de la vue

```
public SalesListPage(SalesService salesService, AppStore store)
{
    ...
    BindingContext = store;
    ...
}
```

EXTRAIT DE CODE XII

*Souscription de la vue à différents éléments du store
avec le mot-clé **Binding***

```
<ListView ItemsSource="{Binding Sale}" ...>
    ...
</ListView>
```

Ensuite, quand un membre du *store* est modifié, ce dernier déclenche l'événement `PropertyChanged` fourni par l'interface `IPropertyChanged` pour notifier la vue du changement (souscrite grâce au mécanisme de binding [EXTRAIT DE CODE XII](#)) :

EXTRAIT DE CODE XIII

Le store notifie à son tour ses observers du changement

```
public class Store : INotifyPropertyChanged
{
    ...
    private Sale sale;
    public Sale Sale
    {
        ...
        private set
        {
            sale = value;
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Sale"));
        }
    }
}
```

5.3.4.4 Dans les vues

Voici comment utiliser les *Reactive Extensions* et leurs opérateurs dans le contexte de l'implémentation d'une barre de recherche (`SalesSearchBar`) :

EXTRAIT DE CODE XIV

Utilisation des Reactive Extensions avec une SearchBar

```
var subscription =
Observable.FromEventPattern<EventHandler<TextChangedEventArgs>,
TextChangedEventArgs>(ev => SalesSearchBar.TextChanged += ev, ev =>
SalesSearchBar.TextChanged -= ev)
    .Select(evt => evt.EventArgs.NewTextValue)
    .Throttle(TimeSpan.FromMilliseconds(1000))
    .DistinctUntilChanged()
    .ObserveOn(SynchronizationContext.Current)
    .Subscribe(async searchTerm =>
    {
        await salesService.FetchByComicsTitleAsync(searchTerm);
    }));
```

Premièrement, comme déjà montré dans les extraits de code précédents, nous transformons l'événement `TextChanged` de l'objet `SearchBar` afin de pouvoir traiter les notifications reçues comme une collection (y appliquer des opérateurs).

Les exemples ci-dessous font référence à l'[EXTRAIT DE CODE XIV](#).

Select : équivalent de l'opérateur `Map`, abordé dans le [point sur les opérateurs](#). Il permet de transformer la valeur arrivant dans l'observable.

- Exemple : nous transformons l'objet de notification d'événement (`evt`) en son texte (`evt.EventArgs.NewTextValue`) - voyons-le comme une extraction. Nous verrons pourquoi cela s'avère utile.

Throttle : cet opérateur est appliqué sur l'observable qu'a retourné l'opérateur `Select` (avec le texte de l'événement en tant que valeur). Il permet d'ignorer les frappes trop rapides de l'utilisateur endéans un certain délai afin d'éviter un grand nombre d'appels au serveur.

- Exemple : si l'utilisateur entre rapidement "jfdj" en 1 seconde, la fonction (observer) passée à `Subscribe` ne sera exécutée qu'une fois et le serveur ne sera appelé qu'une fois avec ces 4 lettres (au lieu de l'être normalement à chaque frappe).

DistinctUntilChanged : grâce à cet opérateur, la fonction (observer) passée à `Subscribe` ne sera pas rappelé si la valeur qu'il reçoit est identique à la précédente.

- Exemple : si nous entrons "Samih" dans la barre de recherche et supprimons le "ih" pour ensuite le réécrire très rapidement, la fonction passée à `Subscribe` serait normalement ré-exécutée après 1 seconde :
"Samih" (appel serveur) => "Sam" => "Samih" => 1 seconde... => appel serveur.

Grâce à `DistinctUntilChanged`, ce deuxième appel serveur sera évité. En effet, nous obtiendrons la même valeur (vu que nous avons entré “*Samih*” à nouveau) et le second appel sera ignoré.

C’est ici que l’opérateur `Select` appliqué précédemment prend tout son sens : si nous n’avions pas transformé l’événement `TextChanged` en texte (la valeur du nouveau texte de l’événement), la valeur qu’aurait reçu l’opérateur `DistinctUntilChanged` aurait été un objet de type événement qui aurait toujours été différent du précédent. En effet, il y aurait une comparaison des références des deux objets. Après la transformation du `Select`, nous avons juste affaire à une comparaison de deux chaînes de caractères.

ObserveOn : permet de spécifier le contexte sur lequel nous désirons que l’observable observe les changements. Dans le cas d’une barre de recherche, nous désirons que cela soit fait sur le *thread principal* (et pas en background) car il s’agit d’un élément de l’interface utilisateur.

Grâce aux *Reactive Extensions* et à leurs opérateurs, nous avons pu en quelques lignes de code écrire certaines fonctionnalités qui seraient plus ardues à mettre en place sans leur utilisation.

5.3.4.5 Cohérence du flux des données

Enfin, démontrons que le flux des données de l’application reste identique, qu’il s’agisse d’un événement reçu de *Firebase* ou issu d’une interaction utilisateur comme le cas échéant avec la fonction `FetchByComicsTitleAsync`.

EXTRAIT DE CODE XV

Fonction de recherche par titre - appel à Firebase et notification du changement

```
public async Task FetchByComicsTitleAsync(string searchTerm)
{
    var result = await FirebaseService.Client?.Child("Sales")
        .OrderBy("ComicsTitle")
        .StartAt(searchTerm)
        .EndAt(searchTerm)
        .OnceAsync<Sale>();

    OnChanged(result);
}
```

Quand une notification provient du back-end, nous avons vu dans l’[EXTRAIT DE CODE X](#) que le *service* notifie ses observateurs (le *store* dans notre cas) d’un changement grâce à son événement `OnChanged`.

Poursuivons le scénario où un changement est émis à partir de la vue, comme c’est le cas avec la fonction `FetchByComicsTitleAsync` (appelée à partir de la vue dans l’[EXTRAIT DE CODE XIII](#)).

Dans l’[EXTRAIT DE CODE XV](#), nous voyons à l’intérieur de la fonction qu’il s’agit bien du même cheminement qu’énoncé précédemment (événement `OnChanged` déclenché par le *service*), que ce changement vienne de la vue ou de *Firebase*.

5.3.4.6 Résumé

Les différents composants de notre application communiquent effectivement en envoyant des notifications d'événements pour propager les changements, ce qui satisfait notre définition de la programmation réactive au [chapitre 2](#).

Notre programmation met à profit les *Reactive Extensions* afin de pouvoir traiter les événements comme des collections et appliquer des opérateurs (filtres, transformations...) sur ces derniers.

Finalement, notre architecture intègre la notion de flux de données unidirectionnel, rendant les changements et le flux général de l'application plus facile à suivre.

6 Chapitre VI : Cas de l'application BdOccaz

6.1 Présentation

6.1.1 Contexte

Il n'y a actuellement à notre connaissance aucune application offrant la possibilité de vendre et d'acheter simplement des bandes dessinées de seconde main en Belgique.

Le potentiel client est toute personne, du particulier au libraire, qui possède des bandes dessinées dont elle n'a plus utilité ou qui est intéressée d'en obtenir à prix avantageux (réduction approximative de 50 %).

6.1.2 Solution

La solution proposée est une application *Android*, *iOS* et *Windows* d'achat et de vente de bandes dessinées de seconde main. Le client peut rechercher, acheter ou encore mettre en ventes des bandes dessinées.

Lors de la recherche, l'utilisateur entre le titre de la bande dessinée. Si l'article recherché est en vente, il pourra procéder à son achat. S'il ne l'est pas, le client pourra être notifié lors de la prochaine apparition de l'article recherché.

Pour la vente, l'utilisateur sera invité à scanner le code-barres de la bande dessinée à l'aide de l'appareil photo de son smartphone. Si l'article est trouvé dans notre base de données, il pourra le mettre en vente directement, en choisissant (ou pas) de modifier le prix de vente par défaut proposé par nos services.

L'utilisateur pourra également consulter ses transactions en cours (achats, ventes), l'état de son compte...

La base de données articles contenant les informations sur les bandes dessinées sera remplie par le gestionnaire de la plateforme. Grâce à une catégorisation détaillée et une base

de données étoffée, la plateforme propose à l'utilisateur des facilités pour l'achat et la vente de bande dessinées.

6.1.3 Remarques

Les mock-ups d'écran exposés au point [6.2 Fonctionnalités](#) sont des prototypes et ont pour but d'aider à la visualisation de l'application et de donner une idée concrète de son design. Le design réel de l'application différera donc des images dans ce document, tout en adoptant la même structure générale.

En raison de l'accent mis sur la programmation réactive dans ce travail, toutes les fonctionnalités énoncées ci-dessus et exposées dans les mock-ups ne seront pas toutes implémentées dans la première version de l'application.

Nous pouvons voir celles-ci comme des possibilités d'améliorations futures.

6.2 Fonctionnalités

6.2.1 Consultation des articles mis en vente

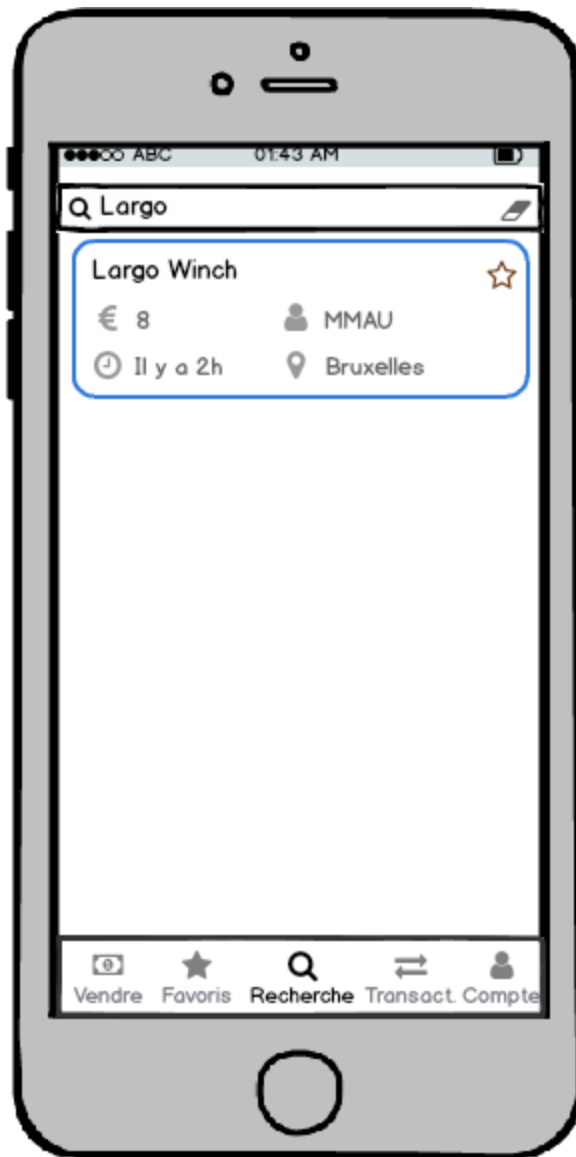


La page d'accueil ou de recherche est la première page affichée lorsqu'un utilisateur ouvre l'application *BdOccaz*. Elle permet à l'utilisateur de voir les dernières ventes mises en ligne. Nous pouvons sélectionner une vente pour voir son détail, ajouter une vente à nos favoris en tapant l'étoile dans le coin supérieur droit de la vente, ou encore rechercher un article de notre choix à l'aide de la barre de recherche en haut de l'écran.

NOTE PROGRAMMATION RÉACTIVE I

Notons que la mise à profit de la programmation réactive a du sens dans ce scénario. En observant la liste de ventes en base de données, nous serions notifiés de tout changement (ex. le prix d'une vente change, un article est vendu) et pourrions le refléter sur l'écran en temps réel.

6.2.2 Détail d'une vente



La liste des dernières ventes est filtrée en fonction de ce que nous entrons dans la barre de recherche. En l'occurrence, nous avons cherché "*Largo Winch*" ce qui réduit la liste à une seule vente. Nous pouvons ensuite, par exemple, sélectionner la vente (et afficher son détail).

Effacer le texte entré permet de revenir à la liste non-filtrée, état décrit dans le point [6.2.1 Consultation](#).

6.2.3 Recherche, article indisponible

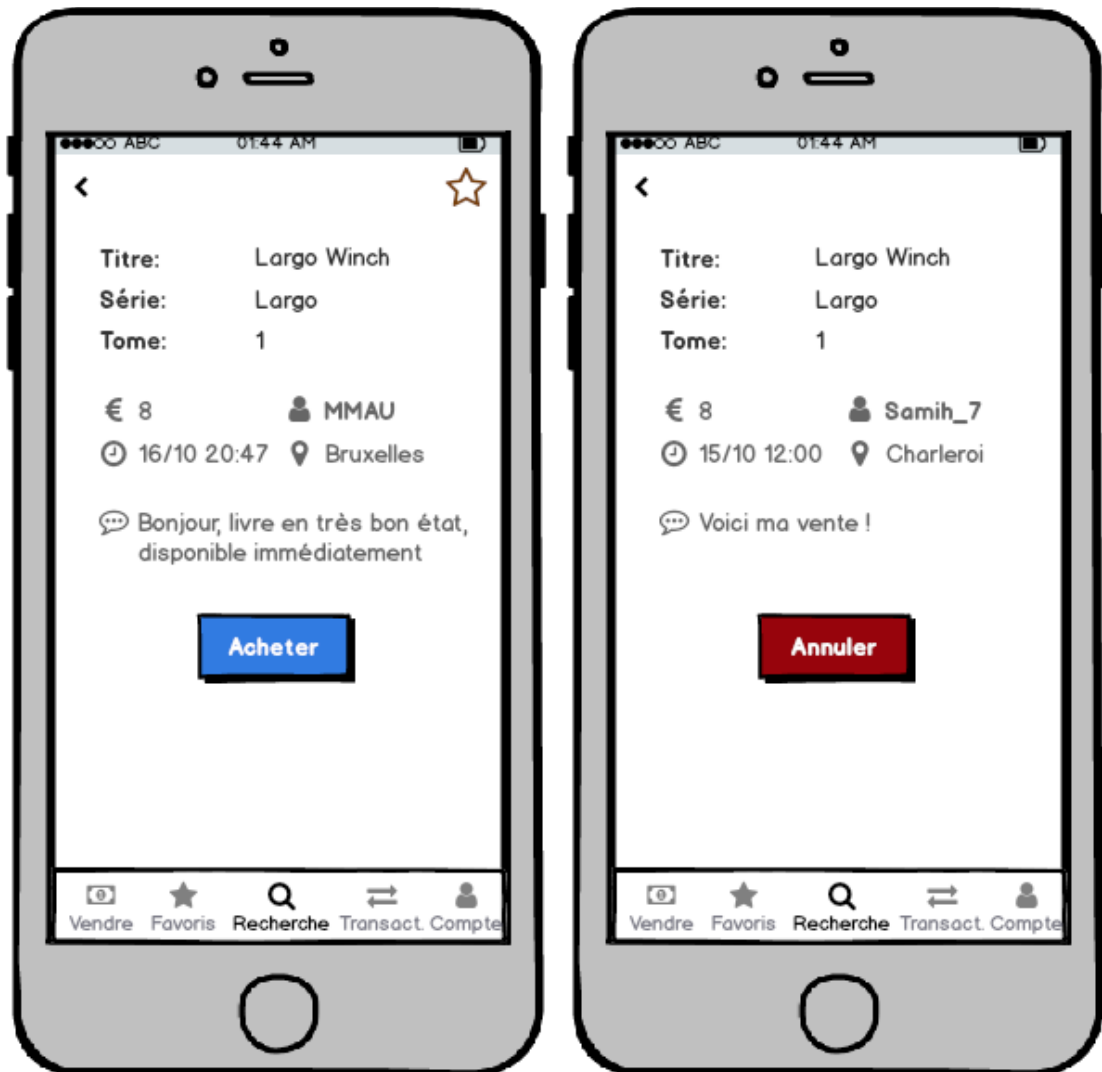


Lorsque nous recherchons un article qui n'est pas en vente actuellement, il nous est proposé d'être notifié à l'apparition de l'article voulu.

Après affichage d'un message de confirmation, nous serons re-dirigés vers la page d'accueil.

Effacer le texte entré permet de revenir à la la liste non-filtrée, état décrit dans le point [6.2.1 Consultation](#).

6.2.4 Recherche, détails vente



Nous consultons ici le détail de l'article choisi et pouvons entreprendre la première étape vers l'achat de ce dernier grâce au bouton "Acheter", ou revenir à notre recherche (bouton "Back"). Il est toujours possible d'ajouter la vente en tant que favorite.

- L'écran de droite correspond à une de nos propres ventes.

Si nous sélectionnons le nom d'utilisateur du vendeur, nous serons re-dirigés vers son profil (voir [6.2.16. Profil utilisateur](#)).

Si nous poursuivons sans être connecté, nous serons invités à nous connecter / créer un compte, comme illustré dans [6.2.13. Connexion](#).

NOTE PROGRAMMATION RÉACTIVE II

Notons que l'utilisation de la programmation réactive a du sens dans ce scénario. Par exemple, si le prix d'une vente change nous pourrions voir cette modification en temps réel avant de procéder à l'achat de la bande dessinée.

6.2.5 Recherche, achat de la bande dessinée en vente



Place à la confirmation de l'achat après sélection du mode de livraison et du mode de paiement. Lors du paiement par la poste, un supplément de 2.6 € nous sera demandé pour payer au vendeur le prix de l'envoi du colis.

Les modes de paiement seront affichés en fonction de ce que le vendeur aura autorisé lors de la mise en vente de l'article (nous y reviendrons au point [6.2.7 Ventes, confirmation](#)).

6.2.6 Vendre, identification de la bande dessinée



Nous nous situons ici dans la partie “*Vendre*” de l’application. Nous pouvons scanner le code-barres (EAN) de l’article à vendre à l’aide de l’appareil photo de notre mobile, ou choisir de l’entrer manuellement.

Si l’utilisateur poursuit et qu’il n’est pas connecté, il sera invité à se connecter / créer un compte, comme illustré dans [6.2.13. Connexion](#).

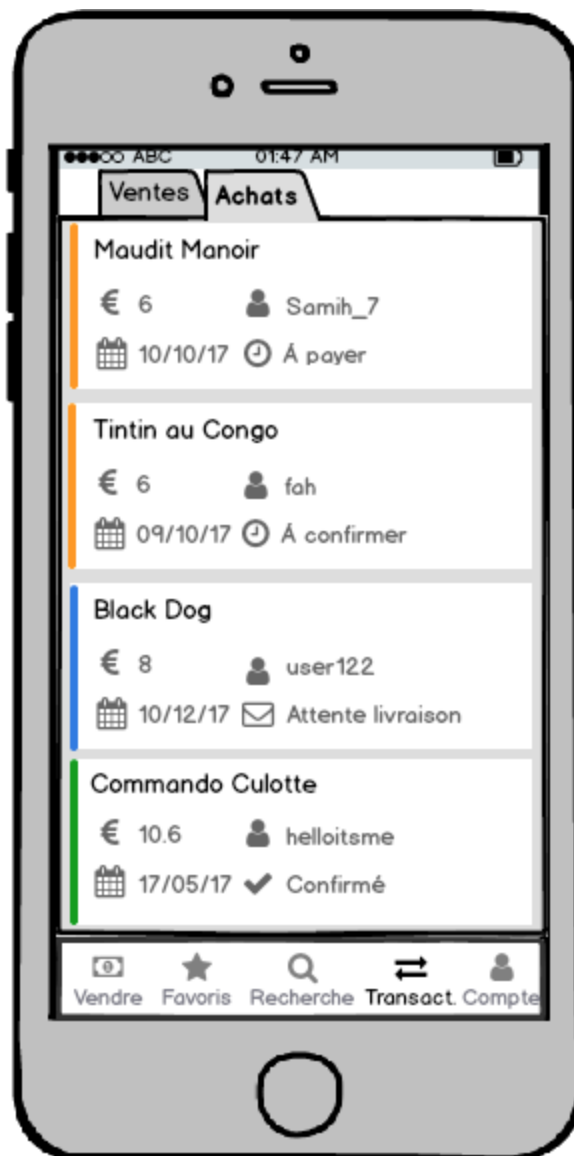
6.2.7 Vendre, confirmation



Voici les détails de la bande dessinée scannée, les champs étant pré-remplis. Il est recommandé d'utiliser le prix conseillé qui reste modifiable en raison d'une éventuelle détérioration ou particularité de l'article.

Le vendeur sélectionne également le type de paiement accepté pour cette vente.

6.2.8 Transactions, achats en cours



Voici la liste des achats en cours qui permet de consulter nos achats et leurs états.

- Couleur **orange** : une action est requise de notre côté.
- Couleur **verte** : la transaction est terminée et aucune action n'est requise.
- Couleur **bleue** : la transaction n'est pas terminée mais aucune action n'est requise (attente du vendeur).

Les achats pour lesquels une action est requise seront affichés en premier lieu, suivis de ceux pour lesquels aucune action n'est requise mais qui ne sont pas terminés, et enfin les achats terminés.

Dans ce contexte (écran), nous nous situons en tant qu'acheteur et voici la signification des états d'un l'achat selon les modes de paiement et de livraison définis :

Pour les achats par virement bancaire (TRANSF) livrés par la poste (MAIL)

- À payer : nous devons confirmer que nous avons payé.
- Attente livraison : nous avons payé et devons attendre que le vendeur nous livre l'article.
- À confirmer : le vendeur a signalé avoir livré l'article et nous devons en confirmer la réception.
- Confirmé : nous avons confirmé la bonne réception de l'article livré par le vendeur.

Pour les achats par virement bancaire (TRANSF) en main propre (HAND)

- À payer : nous devons confirmer que nous avons payé.
- Attente livraison : nous avons payé et nous devons rencontrer le vendeur pour la livraison de l'article.
- À confirmer : le vendeur a signalé avoir livré l'article et nous devons en confirmer la réception.
- Confirmé : l'échange est fait et nous avons signalé que la transaction était terminée.

Pour les achats en liquide (CASH) en main propre (HAND)

- À payer : nous devons confirmer que nous avons payé.
- Attente livraison : nous avons payé en main propre.
- À confirmer : le vendeur a signalé nous avoir donné l'article et nous devons en confirmer la réception.
- Confirmé : l'échange est fait et nous avons signalé que la transaction était terminée.

6.2.9 Transactions, détail achat



Nous visualisons à présent les détails de l'achat choisi dans la liste. Nous pouvons voir plus d'informations sur notre achat : son statut ainsi que les possibles actions qui s'y rapportent, le nom du vendeur et son adresse e-mail afin de le contacter en cas de besoin...

En tant qu'acheteur, deux actions sont possibles : signaler que nous avons payé et confirmer la bonne réception du colis, ce qui clôturera la transaction.

Si nous sélectionnons le nom d'utilisateur du vendeur, nous serons redirigés vers son profil (voir [6.2.16. Profil utilisateur](#)).

6.2.10 Transactions, ventes en cours



Voici la liste des ventes en cours qui permet de consulter nos ventes et leurs états.

- Couleur **orange** : une action est requise de notre côté.
- Couleur **verte** : la transaction est terminée et aucune action n'est requise.
- Couleur **bleue** : la transaction n'est pas terminée mais aucune action n'est requise (attente de l'acheteur).

Les ventes pour lesquelles une action est requise seront affichées en premier lieu, suivies de celles pour lesquelles aucune action n'est requise mais qui ne sont pas terminées, et enfin les ventes terminées.

Dans ce contexte (écran), nous nous situons en tant que vendeur et voici la signification des états de la vente en fonction des modes de paiement et de livraison pour cette dernière :

Pour les ventes par virement bancaire (TRANSF) livrées par la poste (MAIL)

- Attente paiement : nous attendons le paiement de l'acheteur.
- À livrer : l'acheteur a signalé avoir fait le virement et nous devons envoyer le colis par la poste.
- Attente confirmation : nous avons indiqué avoir envoyé le colis et l'acheteur doit en confirmer la bonne réception.
- Confirmé : l'acheteur a confirmé la bonne réception de l'article et la transaction est clôturée.

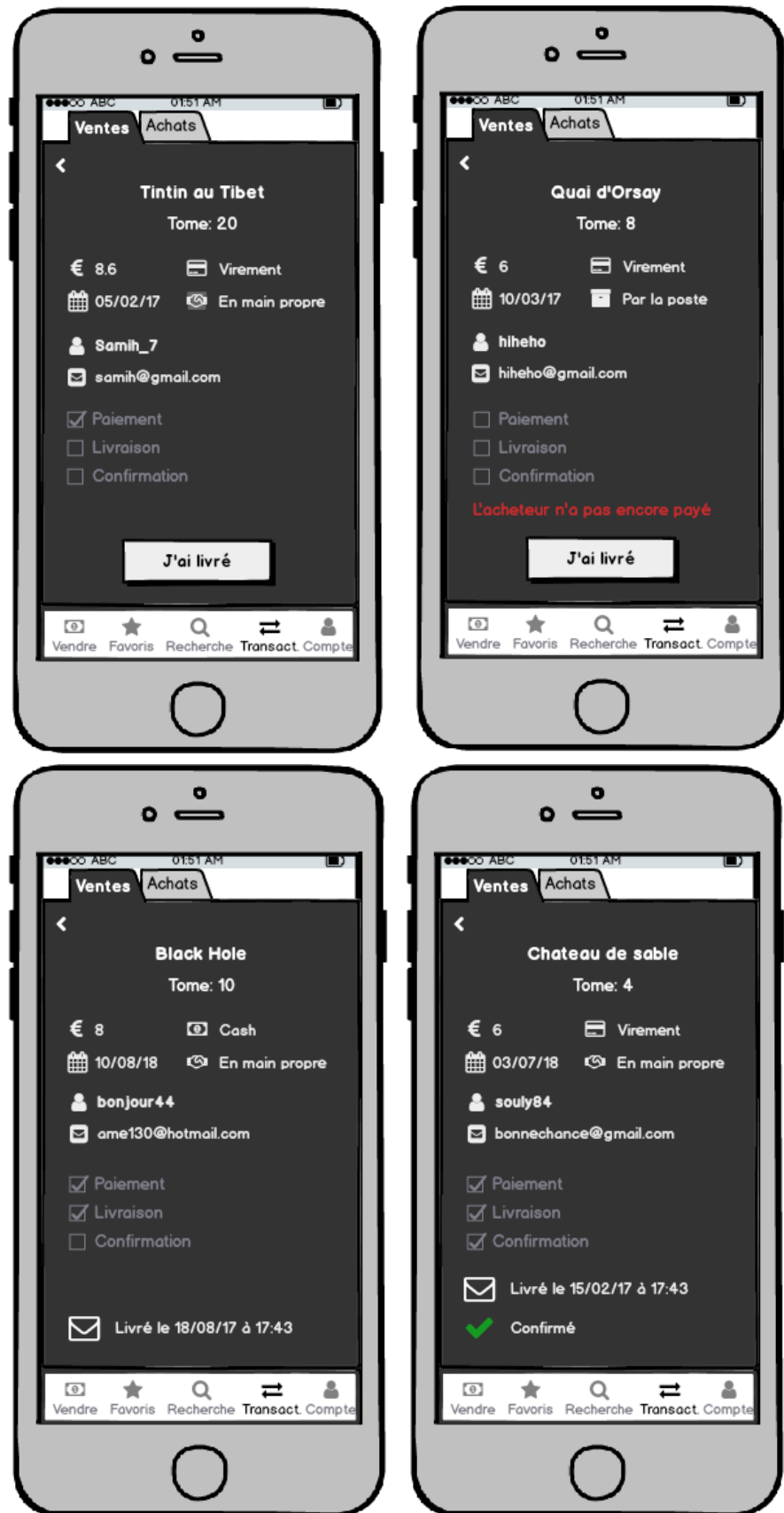
Pour les ventes par virement bancaire (TRANSF) en main propre (HAND)

- Attente paiement : nous attendons le paiement de l'acheteur.
- À livrer : l'acheteur a signalé avoir fait le virement et nous devons le rencontrer pour lui donner le colis.
- Attente confirmation : nous avons indiqué avoir donné le colis à l'acheteur qui doit en confirmer la bonne réception.
- Confirmé : l'acheteur a confirmé la bonne réception de l'article et la transaction est clôturée.

Pour les ventes en liquide (CASH) en main propre (HAND)

- Attente paiement : l'article a été acheté et nous devons rencontrer l'acheteur pour être payé et lui donner l'article.
- À livrer : l'acheteur a signalé nous avoir payé en main propre et nous devons à notre tour indiquer que nous lui avons donné le colis.
- Attente confirmation : nous avons indiqué avoir donné le colis à l'acheteur qui doit en confirmer la bonne réception.
- Confirmé : l'acheteur a confirmé la bonne réception de l'article et la transaction est clôturée.

6.2.11 Transactions, détail vente



Nous visualisons à présent les détails de la vente choisie dans la liste. Nous pouvons voir plus d'informations sur notre achat : son statut ainsi que les possibles actions qui s'y rapportent, le nom de l'acheteur et son adresse e-mail afin de le contacter en cas de besoin...

En tant que vendeur, une action est possible : signaler que nous avons livré le colis (soit par la poste, soit en main propre). La transaction sera clôturée quand l'acheteur accusera de la bonne réception de ce dernier (V Confirmé).

Si l'utilisateur choisit de ne pas mettre d'adresse postale, elle lui sera alors demandée lorsqu'il voudra acheter un article, en le redirigeant vers [6.2.15. Compte](#).

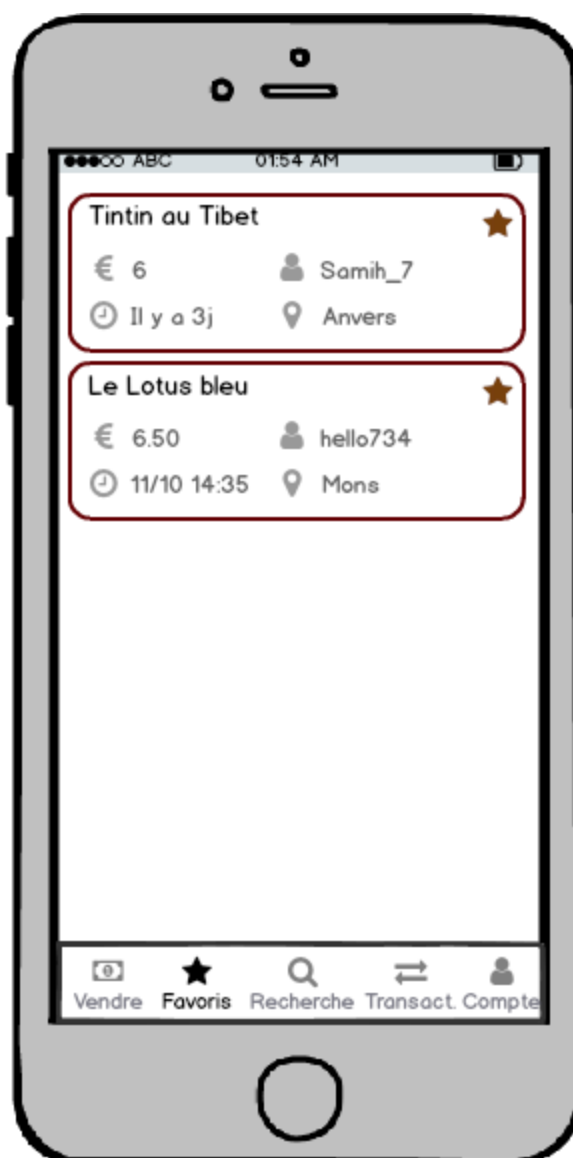
Si nous sélectionnons le nom d'utilisateur de l'acheteur, nous serons redirigés vers son profil (voir [6.2.16. Profil utilisateur](#)).

Notons que l'utilisation de la programmation réactive a du sens dans ce scénario. Par exemple, si le statut de notre vente change (ex. l'acheteur paie) nous pourrions voir cette modification reflétée en temps réel.

NOTE PROGRAMMATION RÉACTIVE III

L'utilisation de la programmation réactive a du sens dans ce scénario. En observant notre vente en base de données, nous serions notifiés de tout changement (ex. l'acheteur paie) et pourrions le refléter sur l'écran en temps réel.

6.2.12 Favoris



Sur cette page seront affichés les ventes que nous aurons renseignées en tant que favorites en sélectionnant l'★ dans le coin supérieur droit.

En sélectionnant un favoris, nous arrivons sur le détail de la vente — voir [6.2.4 Recherche, détails vente](#).

Si l'article a été vendu, il disparaîtra de nos favoris.

6.2.13 Connexion



Cette page sera affichée si nous tentons de vendre ou d’acheter un article sans être connecté. Pour créer un compte *BdOccaz*, nous devons sélectionner “[Créer un compte](#)”.

6.2.14 Création de compte BdOccaz

The image shows a smartphone screen with the 'Création compte' (Create account) form. The form is titled 'Création compte' and has a back arrow icon. It contains the following fields:

- Adresse e-mail (with an envelope icon)
- Nom d'utilisateur (with a person icon)
- Mot de passe (with a magnifying glass icon)
- Nom (with a person icon)
- Prénom (with a person icon)
- Rue, numéro (livraison) (with a location pin icon)
- CP (with a location pin icon)
- Commune (with a location pin icon)
- BExx xxxx xxxx xxxx (with a card icon)

Below the fields is a blue button with a white checkmark. At the bottom of the screen is a navigation bar with the following icons and labels: 'Vendre' (camera icon), 'Favoris' (star icon), 'Recherche' (magnifying glass icon), 'Transact.' (double arrows icon), and 'Compte' (person icon).



Nous avons la possibilité de créer un compte *BdOccaz* afin de pouvoir acheter et vendre sur la plateforme. Cet écran est affiché suite à la sélection de “Créer un compte” comme illustré au point [6.2.13. Connexion](#).

6.2.15 Compte



Description

La fonctionnalité “Compte” nous permet de consulter notre profil, nos informations de paiement ainsi que notre historique. Il sera ultérieurement possible de les modifier.

L'historique personnel reprendra nos achats et ventes. Les achats sont symbolisés par un  (nous obtenons une bande dessinée), et les ventes par un  (nous obtenons de l'argent).

- Sélectionner un achat nous redirigera vers [6.2.9. Transactions, détail achat.](#)
- Sélectionner une vente nous redirigera vers à [6.2.11. Transactions, détail vente.](#)

NOTE PROGRAMMATION RÉACTIVE IV

Bien que l'architecture de l'application reste réactive (basée sur les événements...), cette vue ne nécessite pas d'effort particulier en terme de réactivité. Les données sont relativement statiques (nous sommes le seul à pouvoir les changer), nous pouvons nous contenter de les récupérer une fois arrivé sur l'écran. L'observation continue des changements liés à ces données n'est pas nécessaire dans ce cas.

6.2.16 Profil utilisateur



Nous consultons ici le profil d'un utilisateur après avoir sélectionné son nom d'utilisateur sur le détail d'une vente (point [6.2.4 Recherche, détails vente](#)). Nous pouvons y revenir grâce au bouton 'Back' dans le coin supérieur gauche de l'écran.

L'historique des ventes nous permet de voir si l'utilisateur a déjà vendu un article, ainsi que la date et le prix de la vente.

6.3 Règles concernant les données

6.3.1 Correspondance avec le NoSQL

Pour l'application *BdOccaz*, nous utilisons une base de données *NoSQL*. Les règles ci-après sont donc adaptées à la manière dont les relations sont modélisées avec la *Firebase Realtime Database* (voir point [4.3.2.4 Relations entre documents](#)).

6.3.2 Règles de structure

Légende

- Les termes entre crochets [Entité] représentent les différentes entités.
- Les bullet points sont des définitions de concepts.
- Les bullet points accompagnés de (1), (2)... sont des relations entre les concepts ((1), (2) étant les identifiants des relations).

Règles

Vente [Sale]

- Une vente [Sale] doit posséder une date/heure, un prix, zéro ou un commentaire
- Une vente [Sale] favorite doit posséder un statut (*actif ou pas*)

ONE-TO-ONE, ONE-TO-MANY

- Une vente [Sale] doit être liée à une et une seule bande dessinée [Comics] (1)
- Une vente [Sale] doit être liée à un et un seul vendeur [User] (2)
- Une vente [Sale] doit être liée à zéro ou un seul achat [Purchase] (8)

MANY-TO-MANY

- Une vente [Sale] doit être liée à zéro ou plusieurs vendeurs [User] (*définie comme favorite par plusieurs vendeurs*) (3)
- Une vente [Sale] doit être liée à un ou plusieurs modes de livraison [DeliveryMode] (5)
- Une vente [Sale] doit être liée à un ou plusieurs modes de paiement [PaymentMode] (6)

Bande dessinée [Comics]

- Une bande dessinée [Comics] doit posséder un titre, un tome, un auteur, zéro ou une série, un identifiant unique qui est la valeur de son code-barres (EAN)

ONE-TO-ONE, ONE-TO-MANY

- Une bande dessinée [Comics] doit être liée à zéro ou plusieurs ventes [Sale] (1)

MANY-TO-MANY

- Une bande dessinée [Comics] doit être liée à zéro ou plusieurs utilisateurs [User] (*être le sujet de notification pour plusieurs utilisateurs*) (4)

Utilisateur [User]

- Un vendeur [User] doit posséder un nom d'utilisateur, une adresse e-mail, zéro ou un compte IBAN, un nom, un prénom, zéro ou une photo

ONE-TO-ONE, ONE-TO-MANY

- Un vendeur [User] doit être lié à zéro ou plusieurs ventes [Sale] (2)
- Un acheteur [User] doit être lié à zéro ou plusieurs achats [Purchase] (9)
- Un utilisateur [User] doit être lié à zéro ou une adresse postale [Address] (13)

MANY-TO-MANY

- Un utilisateur [User] doit être lié à zéro ou plusieurs ventes [Sale] (3)
- Un utilisateur [User] doit être lié à zéro ou plusieurs bandes dessinées [Comics] (*être notifié pour plusieurs bandes dessinées*) (4)

Adresse postale [Address]

- Une adresse postale [Address] doit posséder un code postal, une rue et une ville

ONE-TO-ONE, ONE-TO-MANY

- Une adresse postale [Address] doit être liée à un et un seul utilisateur [User] (13)

Mode de livraison [DeliveryMode]

- Un mode de livraison [DeliveryMode] doit posséder un type, zéro ou un prix
Le type d'un mode de livraison [DeliveryMode] peut être par la poste = 'MAIL', ou en main propre = 'HAND' (*d'autres types pourront être ajoutés*)

ONE-TO-MANY

- Un mode de livraison [DeliveryMode] doit être lié à zéro ou plusieurs achats [Purchase] (11)

MANY-TO-MANY

- Un mode de livraison [DeliveryMode] doit être lié à zéro ou plusieurs ventes [Sale] (5)

Mode de paiement [PaymentMode]

- Un mode de paiement [PaymentMode] doit posséder un type
Le type d'un mode de paiement [PaymentMode] peut être en virement bancaire = 'TRANSF' ou en liquide = 'CASH' (*d'autres types pourront être ajoutés*)

ONE-TO-MANY

- Un mode de paiement [PaymentMode] doit être lié à zéro ou plusieurs achats [Purchase] (10)

MANY-TO-MANY

- Un mode de paiement [PaymentMode] doit être lié à zéro ou plusieurs ventes [Sale] (6)

Achat [Purchase]

- Un achat [Purchase] doit posséder une date, un prix total

ONE-TO-ONE, ONE-TO-MANY

- Un achat [Purchase] doit être liée à une et une seule vente [Sale] (8)
- Un achat [Purchase] doit être lié à un et un seul acheteur [User] (9)
- Un achat [Purchase] doit être lié à un et un seul moyen de paiement [PaymentMode] (10)
- Un achat [Purchase] doit être lié à un et un seul moyen de livraison [DeliveryMode] (11)
- Un achat [Purchase] doit être lié à zéro ou plusieurs actions [PurchaseAction] (12)

Action sur achat [PurchaseAction]

- Une action [PurchaseAction] doit posséder une date, un type
- Le type de l'action d'un achat peut être 1 = 'Païement', 2 = 'Livraison' ou 3 = 'Confirmation'.

ONE-TO-ONE, ONE-TO-MANY

- Une action [PurchaseAction] doit être lié à un et un seul achat [Purchase] (12)

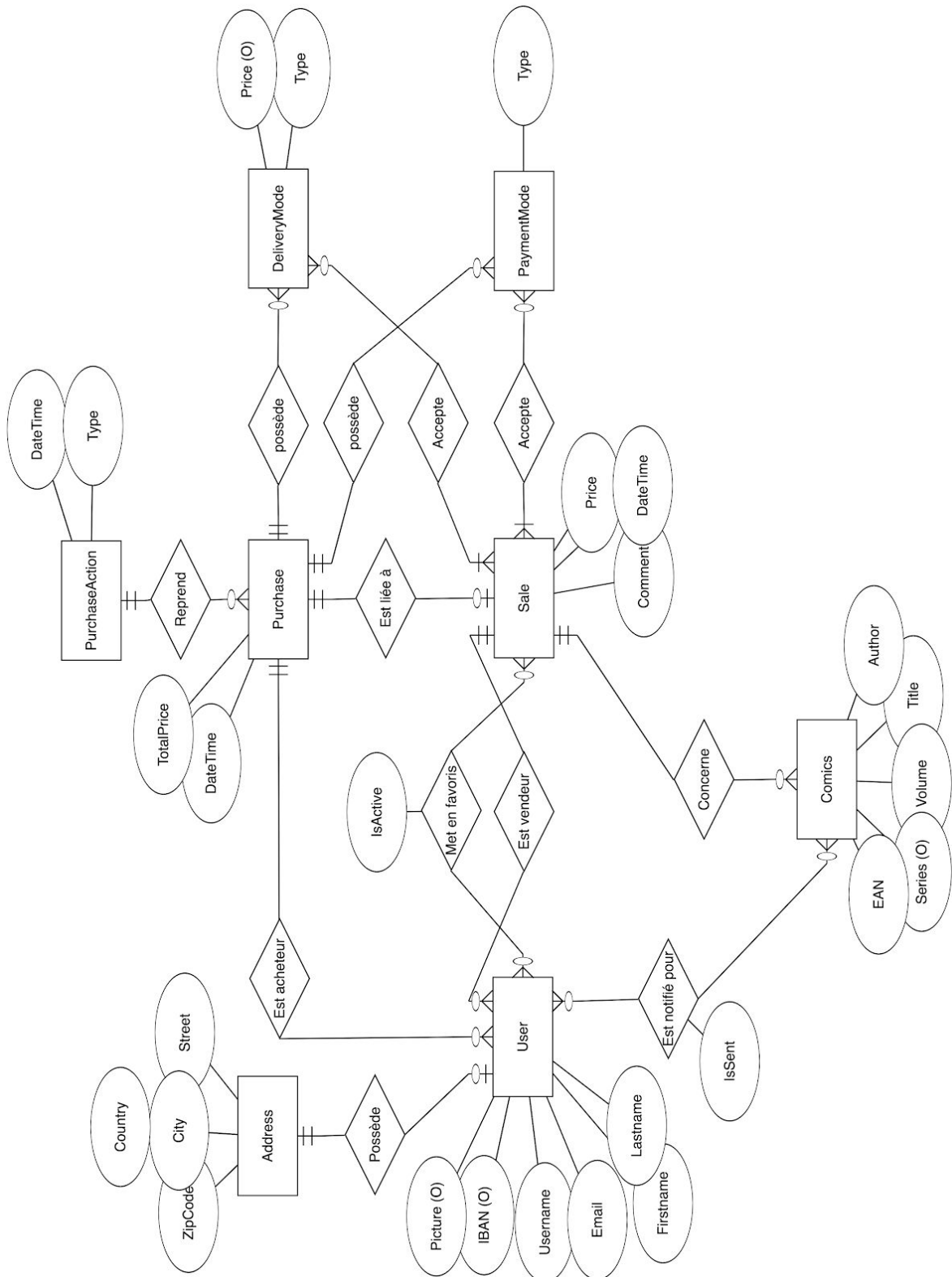
6.3.3 Règles de validation

- Une vente [Sale] favorite peut être retirée des favoris via son statut
- Un vendeur [User] peut supprimer sa propre vente s'il n'y a pas encore d'acheteur [User]
- Si le mode de paiement est 'TRANSF' et que le mode de livraison est 'MAIL', l'achat peut être confirmé
- Si le mode de paiement est 'TRANSF' et que le mode de livraison est 'HAND', l'achat peut être confirmé
- Si le mode de paiement est 'CASH' et que le mode de livraison est 'MAIL', l'achat ne peut pas être confirmé
- Si le mode de paiement est 'CASH' et que le mode de livraison est 'HAND', l'achat peut être confirmé
- La bande dessinée [Comics] ne pourra pas être mise en vente si elle n'est pas trouvée
- Si un achat [Purchase] ne possède pas encore d'action [PurchaseAction], il sera considéré comme étant dans son état initial.
- Si le nom d'utilisateur/e-mail n'est pas trouvé ou ne correspond pas au mot de passe entré, la connexion sera refusée
- Si le nom d'utilisateur/e-mail est trouvé et qu'il correspond au mot de passe entré, la connexion sera acceptée

6.4 Schéma entités-associations

Légende (voir [Annexe 9.2](#) pour les symboles des associations)

- Un rectangle représente une entité.
- Un losange représente une relation.
- Un ovale représente un attribut. L'annotation (O) représente un attribut optionnel.



6.5 Base de données

6.5.1 Création de tables intermédiaires pour les relations many-to-many

Relation “*User > Met en favoris > Sale*” [Favorite]

- Un favoris [Favorite] doit posséder un statut (actif ou pas)

ONE-TO-MANY

- Un utilisateur [User] doit être lié à zéro ou plusieurs ventes favorites [Favorite]
- Une vente favorite [Favorite] doit être liée à un et un seul utilisateur [User]
- Une vente [Sale] doit être liée à zéro ou plusieurs ventes favorites [Favorite]
- Une vente favorite [Favorite] doit être liée à une et une seule vente [Sale]

Relation “*User > Est notifié pour > Comics*” [Notification]

- Une notification [Notification] doit posséder un statut (envoyée ou pas)

ONE-TO-MANY

- Une bande dessinée [Comics] doit être liée à zéro ou plusieurs notifications [Notification]
- Une notification [Notification] doit être liée à une et une seule bande dessinée [Comics]
- Un utilisateur [User] doit être liée à zéro ou plusieurs notifications [Notification]
- Une notification [Notification] doit être liée à un et un seul utilisateur [User]

Relation “*Sale > Accepte > DeliveryMode*”

[AcceptedDeliveryMode]

- Une vente [Sale] doit être liée à un ou plusieurs modes de livraison acceptés [AcceptedDeliveryMode]
- Un mode de livraison accepté [AcceptedDeliveryMode] doit être lié à une et une seule vente [Sale]
- Un mode de livraison [DeliveryMode] doit être lié à un ou plusieurs modes de livraison acceptés [AcceptedDeliveryMode]
- Un mode de livraison accepté [AcceptedDeliveryMode] doit être lié à un et un seul mode de livraison [DeliveryMode]

Relation “*Sale > Accepte > PaymentMode*”

[AcceptedPaymentMode]

- Une vente [Sale] doit être liée à un ou plusieurs modes de livraison acceptés [AcceptedPaymentMode]

- Un mode de livraison accepté [AcceptedPaymentMode] doit être lié à une et une seule vente [Sale]
- Un mode de livraison [DeliveryMode] doit être lié à un ou plusieurs modes de livraison acceptés [AcceptedPaymentMode]
- Un mode de livraison accepté [AcceptedPaymentMode] doit être lié à un et un seul mode de livraison [DeliveryMode]

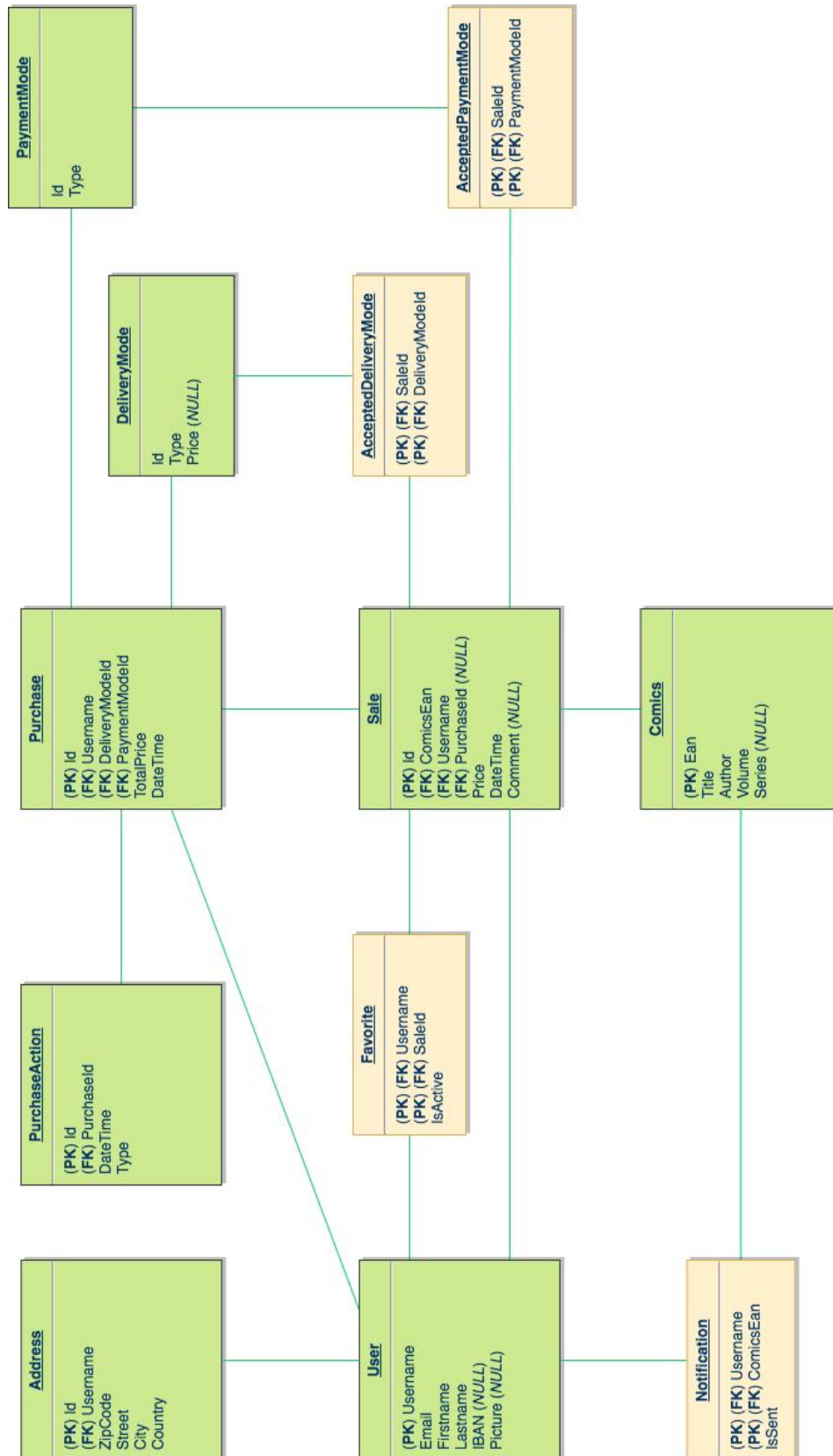
Règles de validation (suite aux nouvelles tables)

- Si une vente [Sale] est ajoutée et que la bande dessinée [Comics] liée à cette vente est également liée à une ou plusieurs notifications, l'utilisateur lié à chaque notification sera notifié

6.5.2 Schéma de la base de données

Légende

- Les tables intermédiaires créées dans le cadre de relations many-to-many sont schématisées en couleur jaune.



7 Chapitre VII : Conclusion

Dans ce travail, nous avons approché la programmation réactive d'un point de vue théorique et nous avons décrit sa mise en place dans une application e-commerce.

Plus spécifiquement, dans la première partie de ce travail, nous avons étudié la programmation réactive en commençant par ses origines et l'avons définie en nous appuyant sur plusieurs définitions existantes. Nous avons apporté des éclaircissements quant aux bénéfices de la programmation réactive en établissant un lien avec le traitement des données en temps réel. Nous avons ensuite exposé des pratiques courantes en programmation réactive, par exemple les systèmes basés sur les événements et les patrons de conception qui en sont dérivés. Nous avons attaché une attention particulière au patron *Observer*, largement mis à profit par les librairies de programmation actuelles.

Nous avons débuté la deuxième partie par une introduction au paradigme *NoSQL* et à l'authentification. Dans ce contexte, nous avons abordé des technologies telles que les bases de données distribuées et le standard d'autorisation *OAuth*. Nous avons ensuite identifié leur utilisation dans notre solution *back-end*, la plateforme *Firebase*. Nous avons justifié le choix de cette plateforme en mentionnant le mécanisme temps-réel que propose la *Firebase Realtime Database*.

Nous avons poursuivi en présentant la technologie *Xamarin*, utilisée pour le *front-end* de notre application e-commerce. Nous sommes dès lors passés à l'étude de l'architecture réactive *Xamarin* imaginée et implémentée dans le cadre de ce travail. Notre architecture rassemble les pratiques de programmation réactive vues dans la première partie. Nous avons procédé à une description schématique de la relation entre les différentes parties du système pour finalement illustrer à l'aide d'exemples concrets la programmation de chaque partie.

Enfin, nous avons détaillé notre cas pratique, l'application de vente et d'achat de bande dessinées d'occasion *BdOccaz*. Nous avons présenté ses différents écrans et avons évalué la pertinence de la programmation réactive pour certains d'entre eux. Sur base des fonctionnalités de l'application, nous avons établi un ensemble de règles structurelles pour notre modèle de données que nous avons amené vers un schéma entité-associations. Dans l'optique de fournir une analyse complète, nous avons également conçu un schéma de base de données, finalement adapté à la base de données *NoSQL Firebase Realtime Database* manipulée dans le cadre de l'application *BdOccaz*.

Nous concluons en soulevant que, selon nous, la considération de solutions *Backend as a Service* telles que *Firebase* peut engendrer un gain de temps grâce aux fonctionnalités qu'elles proposent.

L'approche *NoSQL* vient avec son lot d'avantages, d'inconvénients et un modèle de pensée totalement différent. De par sa croissante popularité et son affinité pour le temps-réel, nous voyons cette dernière en tant qu'alternative viable aux bases de données relationnelles.

Nous pensons également que *Xamarin* se veut une technologie aboutie, dotée d'une vaste communauté et adaptée au développement d'applications multi-plateformes.

Finalement, nous pensons que la programmation réactive s'avère être un modèle efficace, cohabitant aisément avec d'autres paradigmes de programmation. Les plus grandes sociétés de logiciels appliquent ses concepts et tirent parti de bibliothèques telles que les *Reactive Extensions*, ce qui, pour nous, confirme la validité de l'approche. Il est nécessaire de consacrer du temps à la familiarisation avec la programmation réactive et ses outils. Cependant il s'agit, selon nous, d'un bon investissement étant donné la réutilisation possible de ces derniers dans une multitude d'écosystèmes.

8 Bibliographie

"8.8. weakref — Weak references — Python 3.7.0 documentation."

<https://docs.python.org/3/library/weakref.html>. Accessed 10 Sep. 2018.

"An Animated Intro to RxJS | CSS-Tricks." 24 Feb. 2017,

<https://css-tricks.com/animated-intro-rxjs/>. Accessed 12 Sep. 2018.

"Andy Mc's .NET Framework FAQ - Andy McMullan."

<http://www.andymcm.com/dotnetfaq.htm>. Accessed 10 Sep. 2018.

"AngularFirestore: Learn Angular and Firebase with Videos and"

<https://angularfirebase.com/>. Accessed 15 Sep. 2018.

"Arrows, Robots, and Functional Reactive Programming - Springer Link."

https://link.springer.com/chapter/10.1007/978-3-540-44833-4_6. Accessed 1 Sep. 2018.

"A Survey on Reactive Programming - Software Languages Lab - VUB."

<http://soft.vub.ac.be/Publications/2012/vub-soft-tr-12-13.pdf>. Accessed 28 Aug. 2018.

"Basic Authentication | Swagger."

<https://swagger.io/docs/specification/authentication/basic-authentication/>. Accessed 15 Sep. 2018.

"Books by Venkat Subramaniam (Author of Practices of an ... - Goodreads."

https://www.goodreads.com/author/list/2817.Venkat_Subramaniam. Accessed 1 Sep. 2018.

"Brewer's CAP Theorem in Simple Words - HowToDoInJava."

<https://howtodoinjava.com/hadoop/brewers-cap-theorem-in-simple-words/>. Accessed 16 Sep. 2018.

"Brewer's Conjecture and the Feasibility of Consistent ... - CMU (ECE)."

<https://users.ece.cmu.edu/~adrian/731-sp04/readings/GL-cap.pdf>. Accessed 16 Sep. 2018.

"Building Scalable Databases: Denormalization, the NoSQL Movement" 10 Sep. 2009,

<http://www.25hoursaday.com/weblog/2009/09/10/BuildingScalableDatabasesDenormalizationTheNoSQLMovementAndDigg.aspx>. Accessed 16 Sep. 2018.

"CAP Theorem: Revisited - Robert Greiner." 14 Aug. 2014,

<http://robertgreiner.com/2014/08/cap-theorem-revisited/>. Accessed 16 Sep. 2018.

"Characteristics of scalability and their impact on ... - ACM Digital Library."

<https://dl.acm.org/citation.cfm?id=350432>. Accessed 9 Sep. 2018.

"Cleaning Up Unmanaged Resources | Microsoft Docs." 29 Mar. 2017,
<https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/unmanaged>. Accessed 10 Sep. 2018.

"Cloud Functions for Firebase - Google." <https://firebase.google.com/docs/functions/>. Accessed 16 Sep. 2018.

"Content-based Publish/Subscribe Systems - Semantic Scholar."
<https://pdfs.semanticscholar.org/4221/21c6edcfd7445cfc2d447c55e8e87115cd4.pdf>. Accessed 8 Sep. 2018.

"Data Flow - Redux."
<https://redux.js.org/basics/dataflow>. Accessed 29 Aug. 2018.

"DB-Engines Ranking - Trend of Firebase Realtime Database Popularity."
https://db-engines.com/en/ranking_trend/system/Firebase+Realtime+Database. Accessed 15 Sep. 2018.

"Denormalization: How, When and Why (part2) | Codementor." 4 Oct. 2017,
<https://www.codementor.io/nwankwo.c.michael/denormalization-how-when-and-why-part2-ci-u7f6vnj>. Accessed 18 Sep. 2018.

"Design Patterns: Elements of Reusable Object-Oriented Software [Book]."
<https://www.oreilly.com/library/view/design-patterns-elements/0201633612/>. Accessed 13 Sep. 2018.

"Design Patterns Iterator Pattern - Tutorialspoint."
https://www.tutorialspoint.com/design_pattern/iterator_pattern.htm. Accessed 13 Sep. 2018.

"Design Patterns: Observer and Publish-Subscribe - YouTube." 26 Jul. 2017,
<https://www.youtube.com/watch?v=72bdaDI4KLM>. Accessed 10 Sep. 2018.

"Documentation | Firebase - Google." <https://firebase.google.com/docs/>. Accessed 15 Sep. 2018.

"Event-Based Programming: Taking Events to the Limit."
https://books.google.be/books?id=9CL446IzhuAC&pg=PA253&dq=event+source&hl=en&sa=X&ved=0ahUKEwjHnN_hwKPdAhUJYVAKHYQEAXIQ6AEIKjAA#v=onepage&q&f=false. Accessed 05 Sep. 2018.

"End User Authentication with OAuth 2.0 — OAuth."
<https://oauth.net/articles/authentication/>. Accessed 16 Sep. 2018.

"Event Driven Programming - TechnologyUK."
<http://www.technologyuk.net/software-development/designing-software/event-driven-programming.shtml>. Accessed 5 Sep. 2018.

"FatFractal ups the ante in backend-as-a-service market - Techgoindu" 30 Sep. 2012, <https://www.techgoindu.com/2012/09/30/fatfractal-ups-the-ante-in-backend-as-a-service/>. Accessed 15 Sep. 2018.

"Firebase." <https://firebase.google.com/>. Accessed 15 Sep. 2018.

"Firebase Multi-path Updates — Updating Denormalized Data in" 24 Feb. 2017, <https://medium.com/@danbroadbent/firebase-multi-path-updates-updating-denormalized-data-in-multiple-locations-b433565fd8a5>. Accessed 18 Sep. 2018.

"Firebase Realtime Database | Store and sync data in real time | Firebase." <https://firebase.google.com/products/realtime-database/>. Accessed 15 Sep. 2018.

"Flux | Application Architecture for Building User Interfaces." <https://facebook.github.io/flux/docs/in-depth-overview.html>. Accessed 29 Aug. 2018.

"flux/examples/flux-concepts at master · facebook/flux · GitHub." <https://github.com/facebook/flux/tree/master/examples/flux-concepts>. Accessed 13 Sep. 2018.

"Functional Reactive Animation - Conal Elliott." <http://conal.net/papers/icfp97/>. Accessed 1 Sep. 2018.

"Functional programming - Haskell Wiki - Haskell.org." 24 Dec. 2014, https://wiki.haskell.org/Functional_programming. Accessed 1 Sep. 2018.

"Cluster Computing: Applications - Georgia Tech College of Computing." 25 Jul. 2004, <https://www.cc.gatech.edu/~bader/papers/ijhpc.html>. Accessed 16 Sep. 2018.

"GitHub - datorama/akita: Simple and Effective State Management for" <https://github.com/datorama/akita>. Accessed 13 Sep. 2018.

"GitHub - reflex-frp/reflex: Reflex FRP is a composable, cross-platform" <https://github.com/reflex-frp/reflex>. Accessed 2 Sep. 2018.

"Glossary - The Reactive Manifesto." <https://www.reactivemanifesto.org/glossary>. Accessed 12 Sep. 2018.

"Google Owns 100% Of 2017's Top 10 Fastest-Growing SDKs ... - Forbes." 19 Dec. 2017, <https://www.forbes.com/sites/johnkoetsier/2017/12/19/google-owns-100-of-2017s-top-10-fastest-growing-sdks-on-android/>. Accessed 15 Sep. 2018.

"GOTO 2012 - Introduction to NoSQL" https://www.youtube.com/watch?v=qI_g07C_Q5I. Accessed 16 Sep 2018.

"GOTO 2017 - The Many Meanings Of Event-Driven Architecture" <https://www.youtube.com/watch?v=STKCRSUsyP0>. Accessed 6 Sep 2018.

"Hacker Way: Rethinking Web App Development at Facebook - <https://www.youtube.com/watch?v=nYkdrAPrdcw>. Accessed 13 Sep. 2018.

"Indirect communication."
http://www.inf.fu-berlin.de/inst/ag-se/teaching/V-NETZPR-2015/07_Indirect_Communication%20-%20I.pdf. Accessed 10 Sep. 2018.

"Introducing the Observable from @jhusain on @eggheadio."
<https://egghead.io/lessons/rxjs-introducing-the-observable>. Accessed 13 Sep. 2018.

"Handling and Raising Events | Microsoft Docs." 29 Mar. 2017,
<https://docs.microsoft.com/en-us/dotnet/standard/events/>. Accessed 10 Sep. 2018.

"ISO/IEC/IEEE 24765:2010 - Systems and software engineering"
<https://www.iso.org/standard/50518.html>. Accessed 6 Sep. 2018.

"JMS - Publish/Subscribe messaging example using ActiveMQ and" 26 Nov. 2014,
<https://www.codenotfound.com/jms-publish-subscribe-messaging-example-activemq-maven.html>. Accessed 8 Sep. 2018.

"JSON." <https://www.json.org/>. Accessed 15 Sep. 2018.

"Kinvey Raises \$5 Million For Mobile And Web App Backend As A" 11 Jul. 2012,
<https://techcrunch.com/2012/07/11/kinvey-raises-5-million-for-mobile-and-web-app-back-end-as-a-service/>. Accessed 15 Sep. 2018.

"L'élasticité des bases de données sur le cloud computing - CoDE" 18 Apr. 2011,
<http://code.ulb.ac.be/dbfiles/Deg2011mastersthesis.pdf>. Accessed 16 Sep. 2018.

"Lessons Learned Building a Backend-as-a-Service: A ... - Baqend Blog." 17 May. 2017,
<https://medium.baqend.com/how-to-develop-a-backend-as-a-service-from-scratch-lessons-learned-a9fac618c2ce>. Accessed 15 Sep. 2018.

"Managing the Data Base Environment - James Martin - Google Books."
https://books.google.com/books/about/Managing_the_Data_Base_Environment.html?id=nMgmAAAAAAAJ. Accessed 2 Sep. 2018.

"MASTER THESIS Reactive Programming with Events - Tomas Petricek."
<http://tomasp.net/academic/theses/events/events.pdf>. Accessed 28 Aug. 2018.

"Model Relational Data in Firestore NoSQL - YouTube." 9 Feb. 2018,
<https://www.youtube.com/watch?v=jm66TSIVtcc>. Accessed 15 Sep. 2018.

"Mobile Backend as a Service Market Size, Share, Market Intelligence" 1 Jun. 2018,
<https://www.marketwatch.com/press-release/mobile-backend-as-a-service-market-size-share-market-intelligence-company-profiles-and-trends-forecast-to-2023-2018-06-01>. Accessed 15 Sep. 2018.

"MVC Framework Introduction - Tutorialspoint."
https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm. Accessed 2 Sep. 2018.

Nickolay Tsvetinov, [Learning Reactive Programming with Java 8](#).

"NOSQL Databases." <http://nosql-database.org/>. Accessed 16 Sep. 2018.

"NoSQL databases eat into the relational database market" 4 Mar. 2015,
<https://www.techrepublic.com/article/nosql-databases-eat-into-the-relational-database-market/>. Accessed 16 Sep. 2018.

"OAuth 2.0 : An Overview." <https://www.youtube.com/watch?v=CPbvxxslDTU>. Accessed 15 Sep. 2018.

"Observer Design Pattern - SourceMaking."
https://sourcemaking.com/design_patterns/observer. Accessed 9 Sep. 2018.

"Observer Pattern and Lapsed Listener Problem." 17 Apr. 2016,
<http://ilkinulas.github.io/development/general/2016/04/17/observer-pattern.html>. Accessed 10 Sep. 2018.

"On The subject Of subjects (in RxJS) – Ben Lesh – Medium." 9 Dec. 2016,
<https://medium.com/@benlesh/on-the-subject-of-subjects-in-rxjs-2b08b7198b93>. Accessed 12 Sep. 2018.

"OpenID Connect | OpenID." <https://openid.net/connect/>. Accessed 15 Sep. 2018.

"Pando: AnyPresence partners with Heroku to beef up its enterprise" 24 Jun. 2013,
<https://pando.com/2013/06/24/anypresence-partners-with-heroku-to-beef-up-its-enterprise-mbaas-offering/>. Accessed 15 Sep. 2018.

"public static void main(String[] args) - Java main method - JournalDev."
<https://www.journaldev.com/12552/public-static-void-main-string-args-java-main-method>. Accessed 5 Sep. 2018.

"(PDF) Reactive Programming: A Walkthrough - ResearchGate."
https://www.researchgate.net/publication/308807238_Reactive_Programming_A_Walkthrough. Accessed 3 Sep. 2018.

"React and Flux: Building Applications with a Unidirectional ... - YouTube." 28 Aug. 2014,
https://www.youtube.com/watch?v=i__969noyAM. Accessed 13 Sep. 2018.

"Reactive Programming at Netflix – Netflix TechBlog – Medium." 16 Jan. 2013,
<http://techblog.netflix.com/2013/01/reactive-programming-at-netflix.html>. Accessed 2 Sep. 2018.

"Reactive Programming, changing the world at Netflix ... - YouTube." 6 May. 2016,
<https://www.youtube.com/watch?v=yEeDbHvg1vQ>. Accessed 12 Sep. 2018.

"Reactive Programming by Venkat Subramaniam - YouTube." 9 Nov. 2016,
<https://www.youtube.com/watch?v=weWSYIUdX6c>. Accessed 1 Sep. 2018.

"Reactive programming in Standard ML - IEEE Conference Publication."
<https://ieeexplore.ieee.org/document/674156/>. Accessed 28 Aug. 2018.

"Reactive programming: origins & ecosystem - Declarative programming."
<http://jonaschapuis.com/wp-content/uploads/2017/09/ReactiveProgrammingOriginsAndEcosystem.pdf>. Accessed 1 Sep. 2018.

"ReactiveX.io." <http://reactivex.io/>. Accessed 2 Sep. 2018.

ReactiveX - Filter operator." <http://reactivex.io/documentation/operators/filter.html>. Accessed 13 Sep. 2018.

"ReactiveX - Map operator." <http://reactivex.io/documentation/operators/map.html>. Accessed 13 Sep. 2018.

"Realtime Database Limits - Firebase - Google."
<https://firebase.google.com/docs/database/usage/limits>. Accessed 18 Sep. 2018.

"Zip - ReactiveX." <http://reactivex.io/documentation/operators/zip.html>. Accessed 13 Sep. 2018.

"Real time programming : special purpose or general purpose ... - Inria."
<http://www-sop.inria.fr/members/Gerard.Berry/Papers/Berry-IFIP-89.pdf>. Accessed 3 Sep. 2018.

"Rise of Mobile Backend as a Service (MBaaS) API Stacks." 3 Jun. 2012,
<https://apievangelist.com/2012/06/03/rise-of-mobile-backend-as-a-service-mbaas-api-stacks/>. Accessed 15 Sep. 2018.

"Scale with Multiple Databases | Firebase Realtime Database ... - Google."
<https://firebase.google.com/docs/database/usage/sharding>. Accessed 18 Sep. 2018.

"SDK (software development kit) - Gartner IT Glossary."
<https://www.gartner.com/it-glossary/sdk-software-development-kit>. Accessed 15 Sep. 2018.

"Single Source of Truth – Emmanuel Fleurine – Medium." 7 Aug. 2017,
<https://medium.com/@EmmaFleurine/single-source-of-truth-3e83001159c0>. Accessed 13 Sep. 2018.

"SQL vs NoSQL: The Differences Explained - Panoply Blog." 9 Mar. 2017,
<https://blog.panoply.io/sql-or-nosql-that-is-the-question>. Accessed 16 Sep. 2018.

"Structure Your Database | Firebase Realtime Database ... - Google." 16 Aug. 2018,
<https://firebase.google.com/docs/database/ios/structure-data>. Accessed 15 Sep. 2018.

"The Android Event Loop - mattias - Niklewski." 15 Sep. 2012,
http://mattias.niklewski.com/2012/09/android_event_loop.html. Accessed 7 Sep. 2018.

"The design of a multiparadigm programming language - Science Direct."
<https://www.sciencedirect.com/science/article/pii/S0165607493900411>. Accessed 1 Sep. 2018.

"The Essence of Reactive Programming - TU Delft Repositories."
<https://repository.tudelft.nl/islandora/object/uuid:bd900036-40f4-432d-bfab-425cdebc466e/datastream/OBJ/download>. Accessed 29 Aug. 2018.

"The Firebase Blog: Firebase expands to become a unified app platform." 18 May. 2016,
<https://firebase.googleblog.com/2016/05/firebase-expands-to-become-unified-app-platform.html>. Accessed 15 Sep. 2018.

"The introduction to Reactive Programming you've been ... - GitHub Gist."
<https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>. Accessed 29 Aug. 2018.

"The Model-View-ViewModel Pattern - Xamarin | Microsoft Docs." 6 Aug. 2017,
<https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm>. Accessed 13 Sep. 2018.

"The Object-Relational Impedance Mismatch - Agile Data."
<http://www.agiledata.org/essays/impedanceMismatch.html>. Accessed 16 Sep. 2018.

"The observer pattern is two modules or is one module?"
<https://stackoverflow.com/a/47697564>. Accessed 9 Sep. 2018.

"The Case for Flux – The Startup – Medium." 18 Feb. 2015,
<https://medium.com/swlh/the-case-for-flux-379b7d1982c6>. Accessed 13 Sep. 2018.

"The principal programming paradigms."
<https://www.info.ucl.ac.be/~pvr/paradigmsDIAGRAMeng108.pdf>. Accessed 1 Sep. 2018.

"Understanding OAuth: What Happens When You Log Into a Site with" 13 Jun. 2012,
<https://lifehacker.com/5918086/understanding-oauth-what-happens-when-you-log-into-a-site-with-google-twitter-or-facebook>. Accessed 15 Sep. 2018.

"Unidirectional data flow on Android: The blog post (part 1)..". 14 Mar. 2018,
<https://proandroiddev.com/unidirectional-data-flow-on-android-the-blog-post-part-1-cadc72f5>. Accessed 13 Sep. 2018.

"VoV 020: Reactive Programming With Vue With Tracy Lee ... - Player FM." 17 Jul. 2018,
<https://player.fm/series/views-on-vue/vov-020-reactive-programming-with-vue-with-tracy-lee>. Accessed 3 Sep. 2018.

"What is an Event? Webopedia Definition." <https://www.webopedia.com/TERM/E/event.html>. Accessed 5 Sep. 2018.

"What is 'Functional Programming'? | alvinalexander.com." <https://alvinalexander.com/scala/fp-book/what-is-functional-programming>. Accessed 1 Sep. 2018.

"What is NoSQL? | Nonrelational Databases, Flexible Schema Data" <https://aws.amazon.com/nosql/>. Accessed 30 Aug. 2018.

"What is a Transaction? | Microsoft Docs." 30 May. 2018, <https://docs.microsoft.com/en-us/windows/desktop/ktm/what-is-a-transaction>. Accessed 16 Sep. 2018.

"What the Heck is OAuth? | Okta Developer." 21 Jun. 2017, <https://developer.okta.com/blog/2017/06/21/what-the-heck-is-oauth>. Accessed 15 Sep. 2018.

"Xamarin App Development with Visual Studio | Visual Studio." 21 Aug. 2018, <https://visualstudio.microsoft.com/xamarin/>. Accessed 29 Aug. 2018.

9 Annexes

9.1 Stack technologique

9.1.1 Back-end Firebase

Base de données : *Firebase Realtime Database*¹⁴³.

Authentification : *Firebase Authentication*¹⁴⁴.

Logique : *Cloud Functions for Firebase*¹⁴⁵.

9.1.2 Front-end Xamarin

Librairie(s) UI : *Xamarin.Forms*¹⁴⁶.

Programmation réactive : *Reactive Extensions*¹⁴⁷.

SDKs Firebase : *FirebaseDatabase.net*¹⁴⁸, *FirebaseAuthentication.net*¹⁴⁹.

Conteneur d'injection de dépendances : *AutoFac*¹⁵⁰.

Données fictives : *GenFu*¹⁵¹.

¹⁴³ "Firebase Realtime Database - Google." <https://firebase.google.com/docs/database/>. Accessed 17 Sep. 2018.

¹⁴⁴ "Firebase Authentication | Firebase." 16 Aug. 2018, <https://firebase.google.com/docs/auth/>. Accessed 17 Sep. 2018.

¹⁴⁵ "Cloud Functions for Firebase - Google." <https://firebase.google.com/docs/functions/>. Accessed 17 Sep. 2018.

¹⁴⁶ "Xamarin.Forms - Xamarin | Microsoft Docs." <https://docs.microsoft.com/en-us/xamarin/xamarin-forms/>. Accessed 17 Sep. 2018.

¹⁴⁷ "GitHub - dotnet/reactive: The Reactive Extensions for .NET." <https://github.com/dotnet/reactive>. Accessed 17 Sep. 2018.

¹⁴⁸ "GitHub - step-up-labs/firebase-database-dotnet: C# library for" <https://github.com/step-up-labs/firebase-database-dotnet>. Accessed 17 Sep. 2018.

¹⁴⁹ "GitHub - step-up-labs/firebase-authentication-dotnet: C# library for" <https://github.com/step-up-labs/firebase-authentication-dotnet>. Accessed 17 Sep. 2018.

¹⁵⁰ "Autofac: Home." <https://autofac.org/>. Accessed 17 Sep. 2018.

¹⁵¹ "GitHub - MisterJames/GenFu: GenFu is a library you can use to" <https://github.com/MisterJames/GenFu>. Accessed 17 Sep. 2018.

9.2 Description des associations du schéma entité-associations

Examples:

One A is associated with one B:



One A is associated with one or more B's:



One or more A's are associated with one or more B's:



One A is associated with zero or one B:



One A is associated with zero or more B's:

