SUHANA Sutra 51701858
FIDALGO Alex 71600135

PROGRAMMATION COMPARÉE

FLUTTER

vs

REACT NATIVE

vs

SWIFT & KOTLIN

Sous la direction de Guatto Adrien, Letouzey Pierre & Férée Hugo

SOMMAIRE

1 Introduction

Dans ce rapport, nous aborderons la programmation mobile sous plusieurs aspects. Nous aborderons brièvement la programmation mobile native (en **Swift** & en **Kotlin**) à titre de comparateur mais notre attention sera principalement portée sur la comparaison de deux frameworks que sont **React Native** et **Flutter**.
En effet, la comparaison entre **React Native** et **Flutter**, frameworks de programmation mobile multiplateforme, nous à semblée plus pertinente et intéréssante.

2 Langage de programmation & Structure de code

Tout d'abord parlons des différents langages de programmation utilisés pour développer une application mobile. Il existe aujourd'hui deux catégories d'applications mobiles : les applications pûrement **natives** et les applications **multiplateformes**.

2.1 Langage de programmation

Un application multiplateforme se base en grande partie sur une même base de code, et comme son nom l'indique est déstinée à plusieurs plateformes (**Web, iOS, Android**) : c'est le rôle des frameworks **React Native** et **Flutter**.

Comme mentionné précédemment, il existe les langages **Swift** et **Kotlin** pour développer une application mobile native (sous entendu **iOS** et **Android**), respectivement sortis en **2014** par **Apple** et **2011** par **JetBrains**.

React Native, mis en production par **Facebook** en **2015**, nous permet quant à lui de développer une application multiplateforme en utilisant le langage **JavaScript** : langage multi-paradigmes (script, orienté objet, impératif & fonctionnel) **interprété** assez populaire sorti en **1995**.

Flutter quant à lui, à vu le jour en **2018** de par **Google** et est basé sur le langage de programmation **Dart** : un langage de programmation orienté objet (classes essentiellement) **compilé**, sorti en **2011**. Il permet également de développer des applications multiplateformes, tout comme **React Native**.

2.2 Structure de code

En **React Native**, tout comme en **Flutter**, l'interface utilisateur se construit de la même façon, à quelques détails près. En **React Native**, tous les éléments sont des appelés des **composants react**. Ces composants se construisent simplement à partir d'une fonction renvoyant une instance de l'objet **JSXComponent**.

En **Flutter**, c'est la même chose, seulement ici ce sont des **Widgets** et ils sont construits via des classes et non des fonctions. En effet, en **Flutter** il existe deux types de **widgets** : les widgets **Stateful** (avec état) et les widgets **Stateless** (sans état). Ces deux **widgets** sont des classes, il est par conséquent possible de les étendrent. La création d'un nouveau **Widget** se fait simplement par l'extension d'une des deux classes mentionnéés ci-dessus et de l'**override** de la fonction **build** (renvoyant le **widget** en question) de celle-ci.

En fait, une application utilisant l'un de ces deux frameworks partage la même structure en terme de code : il peut être vu comme un arbre de **composants**.

Terminons cette partie sur une petite comparaison entre ces deux frameworks sur l'aspect que nous venons d'aborder. La maintenabilité du code par exemple peut-être plus complexe à partir d'un certain moment en **Flutter** car un code bien "aérer" voudrait dire "beaucoup" de classes (une pour chaque widget en réalité), alors qu'il suffirait simplement d'écrire différentes fonctions en **React Native**. En ce qui concerne les langages de programmation, **JavaScript** "semble" être plus simple à appréhender car plus populaire que **Dart**, évidemment celà est subjectif. Pour terminer, ces deux frameworks diposent tous deux d'un avantage certain : il est possible de faire du rechargement à chaud. En d'autres termes, lors du processus de développement d'une application, il est possible d'observer les modifications apportées en temps réel, un vrai bénéfice en terme de temps de développement.

3 Architecture

Si l'architecture des langages de programmation comme **Swift** où encore **Kotlin** (permettant de développer des applications pûrement natives) ne présente rien de très particulier, c'est un point crucial à aborder lorsque l'on parle des frameworks **React Native** et **Flutter**. En effet, pouvoir développer une application à partir d'une même base de code nécessite une architecture particulière que nous allons approfondir dans cette partie.

3.1 Architecture de React Native

Tout d'abord parlons de l'architecture de **React Native**. Lorsqu'une application **React Native** démarre, le **Main Thread** est lancé. Ce **Main Thread** va ensuite lancer deux autres threads : le **Shadow Thread** et le **JavaScript Thread**. Le **Main Thread** est responsable du rendu de l'interface graphique composée de composants natifs, le **JavaScript Thread** lui permet de gérer la partie logique de l'application et le **Shadow Thread** quant à lui permet de faire une liaison entre ces deux derniers.

En effet, dans une application **React Native**, chacun des composants **React** va être transmis au **Main Thread**, qui aura pour rôle de charger les composants natifs selon la plateforme. Cette liaison est rendu possible par le **pont React Native**. Par exemple le composant **View** de **React Native** va être transformé en **UIView** pour **iOS**, en **Android.View** pour **Android** où encore en **div** pour du **Html5**.

Cette transmission de composants est effectué via un processus de sérialisation, en effet chacun des composants est transformé en fichier **JSON** qui sera ensuite transmis au **Main Thread**, qui lui sera responsable de le dé-sérialiser afin de produire le composant natif associé. Pour les interactions utilisateur, le même procédé est effectué. Le composant est envoyé au **Main Thread** qui va gérer un évènement (click sur un bouton par exemple) puis va le renvoyer au **JavaScript Thread** responsable de la gestion logique.
Voir ICI pour une illustration de communication entre les deux threads mentionnés précedemment, via le processus de sérialisation/désérialisation **JSON**.

3.2 Architecture de Flutter

Parlons maintenant de l'architecture que **Flutter** implique. **Flutter** est (contrairement à **React Native**) basé sur ses propres moteurs graphiques avec **SKIA** notamment. Pour l'affichage de ses composants, **Flutter** s'en remet à ses moteurs graphiques, qui seront responsables de dessiner chaque pixels à l'écran. Il est évidemment possible de communiquer via des **API** avec les interfaces natives lorsqu'il s'agit d'avoir accès à certaines propriétés du système. Pour générer une application **Flutter**, il faut évidemment quelques étapes de compilation. En effet, les moteurs graphiques de **Flutter** - étant écrit en **C** où bien en **C++** - peuvent être compilé avec **Android NDK** (le kit de développement natif d'Android) où bien avec **LLVM** lorsqu'il s'agit de générer une application **iOS**. Le code **Dart** est lui compilé via un processus de compilation **Ahead of Time** vers du code natif **ARM/X86**. Finalement, un fichier **APK** est généré pour **Android** et son équivalent **IPA** pour **iOS**. Pour terminer sur l'architecture de **Flutter**, il faut savoir que tout se passe au niveau du code compilé : interactions utilisateur, entrée etc ..

4 Performances

L'architecture décrite précédemment amène implicitement à ses poser des questions sur les performances de ces frameworks. De plus, si ces frameworks permettent en effet de gagner un temps considérable sur le développement d'une application, il n'en reste pas moins qu'un utilisateur final souhaite une expérience fluide et agréable. Nous allons donc dans cette partie analyser deux benchmarks afin d'éclaircir ce point.

4.1 Performances - Calculs mathématiques

Cette partie résumera le benchmark que l'on peut retrouver ICI. Ces tests de performances sont effectués sur deux algorithmes **Gauss-Legendre** et **Borwein** permettant d'effectuer des calculs sur le nombre PI. Chaque test à eté effectué sur **100** calculs du nombre PI avec une précision à **10 millions** de chiffres.

En prenant comme comparateur une application écrite en **Kotlin**, on observe que **Flutter** est environ **2** fois moins rapide que l'application native et que **React Native** est quant à lui beaucoup plus lent, de **6** à **15** fois plus lent.

Même constat sur **iOS** en prenant comme comparateur une application native écrite en **Swift**. Néamoins, petite surprise sur l'algorithme **Gauss-Legendre** où l'application écrite avec **Flutter** performe un peu mieux l'application native, de l'ordre de **15%**.

4.2 Performances - Interface utilisateur

Cette partie résumera le benchmark que l'on peut retrouver ICI. Celui-ci regroupe trois tests de performance différents : du **"caching" d'images**, **affichage de plusieurs animations** et **surchagement d'animations**.

Tout comme le benchmark précedent, le résultat est sans appel. Les applications utilisant **React Native** et **Flutter** sont une nouvelle fois moins performantes. **Flutter** consomme en moyenne **2** fois plus de mémoire et de **CPU** que les applications natives. **React Native** quant à lui reste plus lent que ce dernier et se fait largement distancer par les applications natives.

4.3 Peformances - Conclusion

D'après ces benchmarks, les résultats sont assez clairs. En effet, on observe sans grande suprise que les applications développées avec les frameworks **React Native** et **Flutter** sont moins performantes que les applications natives. **Flutter** reste néamoins plus rapide que **React Native** mais se voit produire des applications plus conséquentes (en terme de taille) que ce dernier en raison des différents moteurs graphiques devant être disponible dans le code compilé pour réaliser tous les traitements graphiques. De plus, **Flutter** est bien plus performant lorsqu'il s'agit de consommation **mémoire** et **CPU**. Néamoins, **React Native** proposant des composants pûrement natifs profite sans doute de certaines optimisations natives, pouvant le rendre sur certains points meilleur (en terme de **FPS** notamment).

Ces différences de performances s'expliquent d'un côté par la nature des langages utilisés : **JavaScript** étant interpreté, il est forcément plus lent que **Dart**, lui étant compilé. Pour terminer sur ce point, les échanges permanents entre les deux threads en **React Native** expliquent cette perte de performance certaine. Il se peut que, à certains moments, comme le montre le deuxième benchmark, le pont soit surchargé, menant à des chutes de performances assez importantes. Un désavantage certain que **Flutter** n'a pas à subir, les moteurs graphiques étant bien plus rapide que ce processus de communication.

5 Temps de développement | Fléxibilité | Maintenabilité

Si la performance est un point crucial lorsque l'on parle d'applications mobiles, il ne faut pas oublier d'autres aspects tels que **le temps de développement**, **la fléxibilité** et la **maintenabilité**, aspects qui s'avèrent être tout aussi important tout au long du développement de celles-ci. **Flutter** et **React Native** permettent tous deux par exemple de partager une base de code à **100%** et à **80-90%** respectivement, entre les différentes plateformes, ce qui confère à ces applications une maintenabilité accrue par rapport aux applications native. Néamoins, ces approches restent moins flexibles que leurs concurrences natives, ces frameworks ne mettant pas forcément toutes les implémentations natives à disposition du développeur. **React Native** peut quant à lui être touché par des problèmes de compatibilité lors de mises à jour de composants natifs. Le coût des applications natives est forcément plus élévé également. En effet avoir deux à trois équipes de développeurs pour une même application entraîne forcément des coûts non négligeables, point positif pour les applications utilisant ces frameworks de développement multiplateforme qui peuvent donc être développées à moindre coût. Pour terminer, l'ennemi de tout développeur, le **temps**. Ici, ces frameworks sont encore une fois loin devant leur concurrence native.

6 Communauté

Pour terminer, la **communauté** présente autour d'un projet logiciel / framework est un point crucial à prendre en compte lorsque l'on souhaite le manipuler de façon concrète.

Voici donc quelques chiffres concernant **React Native**, **Flutter**, **Swift** & **Kotlin** :

	Étoiles sur Github	**Contributeurs**	**Librairies**
Flutter	138 000	977	18 000
React Native	102 000	2 291	37 545
Swift	59 000	922	X
Kotlin	40 000	524	X

7 Conclusion

On peut conclure de façon assez positives sur ces deux frameworks de développement multiplateforme. En effet, si les applications qui en résultent restent moins performantes que des applications natives, ils restent très intéréssant sur d'autres aspects comme la maintenabilité, le temps et le coût de développement. Une évolution constante des smartphones appuie le fait que les performances soient négligeables devant les bénéfices de ces frameworks. Sauf dans certains cas très spécifiques, (applications nécéssitant de lourds calculs, beaucoup d'animations) l'approche multiplateforme reste à prendre en compte très sérieusement.

8 Questions et suggestions

8.1 Questions

→ Question : **Sawssen** : Aviez vous déjà utilisé **React Native** où bien **Flutter** dans un projet personnel ?

→ Réponse : J'ai pour ma part déjà utilisé **React Native** pour développer une application et je n'ai pas observer de grandes différences au niveau performances en comparaison avec une application native.

8.2 Suggestions

→ **Mr. Férée** : Il aurait été pertinent de rajouter des transparents sur le benchmark au niveau de l'interface utilisateur et pas seulement sur des calculs pûrement mathématiques.

→ **Mr. Férée & Mr. Letouzy** : Apporter plus d'éléments techniques sur le processus de communication du pont **React Native (sérialisation/désérialisation JSON)**

Ressources

→ https://www.thedroidsonroids.com/blog/flutter-vs-react-native-what-to-choose-in-2021#whatisflutter

→ https://medium.com/swlh/flutter-vs-native-vs-react-native-examining-performance-31338f081980

→ https://railsware.com/blog/react-native-vs-native-app-development-ios-and-android-in-one-go/

→ https://www.youtube.com/watch?v=lHhRhPV--G0

→ https://www.youtube.com/watch?v=gvkqT_Uoahw

→ https://www.youtube.com/watch?v=TU_kTuz2i9Y

→ https://www.youtube.com/watch?v=TU_kTuz2i9Y

→ https://mobidev.biz/blog/how-react-native-app-development-works

→ https://medium.com/swlh/flutter-vs-react-native-vs-native-deep-performance-comparison-990b90c11433

Illustration représentant la communication entre les deux threads via le pont React Native : (N->JS : Transmission de natif vers JavaScript | JS->N : Transmission de JavaScript vers React Native).

```
N>JS : RCTEventEmitter.receiveTouches(["topTouchStart",[      {"identifier":0,"locationY":32.17039108276367,"locationX":49.83349609375,"pageY":287.8370666503906,"timestamp":45361688,"ta
               rget":7,"pageX":179.83349609375}],[0]]) // click-down
JS>N : UIManager.measure([9,875]) // asking the location of the view on the screen
JS>N : NativeAnimatedModule.startAnimatingNode([9,3,{"type":"frames","frames":[1],"toValue":0.2,"iterations":1},877]) // start click-down animation
JS>N : Timing.createTimer([191,500,1543053811369,false]) // start timer to recognize long press
JS>N : UIManager.setJSResponder([9,false])
N>JS : <callback for UIManager.measure>([0,0,100,83,130,255.6666717529297]) // response to #2
N>JS : <callback for NativeAnimatedModule.startAnimatingNode>([{"finished":true}]) // response to #3 — end click-down animation
N>JS : RCTEventEmitter.receiveTouches(["topTouchEnd",[{"identifier":0,"locationY":32.17039108276367,"locationX":49.83349609375,"pageY":287.8370666503906,"timestamp":45361831,
      "target":7,"pageX":179.83349609375}],[0]]) // click-up
JS>N : UIManager.clearJSResponder([])
JS>N : Timing.deleteTimer([191]) // stop long-press timer
JS>N : NativeAnimatedModule.startAnimatingNode([10,3,{"type":"frames","frames":[0,0.008888888888888889,0.035555555555555556,0.08000000000000002,0.14222222222222222,0.2222
               2222222222227,0.3200000000000001,0.4355555555555557,0.5644444444444444,0.6799999999999999,0.7777777777777777,0.857777777777
               7778,0.9199999999999999,0.9644444444444443,0.991111111111111,1,1],"toValue":1,"iterations":1},887])
JS>N : UIManager.playTouchSound([]) // telling native code to play a sound
JS>N : NativeAnimatedModule.createAnimatedNode([12,{"type":"style","style":{"opacity":3}}])
JS>N : NativeAnimatedModule.connectAnimatedNodes([3,12])
JS>N : NativeAnimatedModule.createAnimatedNode([13,{"type":"props","props":{"style":12}}])
JS>N : NativeAnimatedModule.connectAnimatedNodes([12,13])
JS>N : NativeAnimatedModule.disconnectAnimatedNodeFromView([11,9])
JS>N : NativeAnimatedModule.disconnectAnimatedNodes([10,11])
JS>N : NativeAnimatedModule.disconnectAnimatedNodes([3,10])
JS>N : NativeAnimatedModule.dropAnimatedNode(10])JS>N : NativeAnimatedModule.dropAnimatedNode([11])
JS>N : UIManager.updateView([7,"RCTImageView",                {"src":[{"uri":"http://10.0.3.2:8081/assets/img/like.png?platform=android&hash=5211eac8ed90175c56858673e2e47b4d"}]}])
JS>N : NativeAnimatedModule.connectAnimatedNodeToView([13,9]) // finally we set the image
N>JS : <callback for NativeAnimatedModule.startAnimatingNode>([{"finished":true}]) // response to #11
```