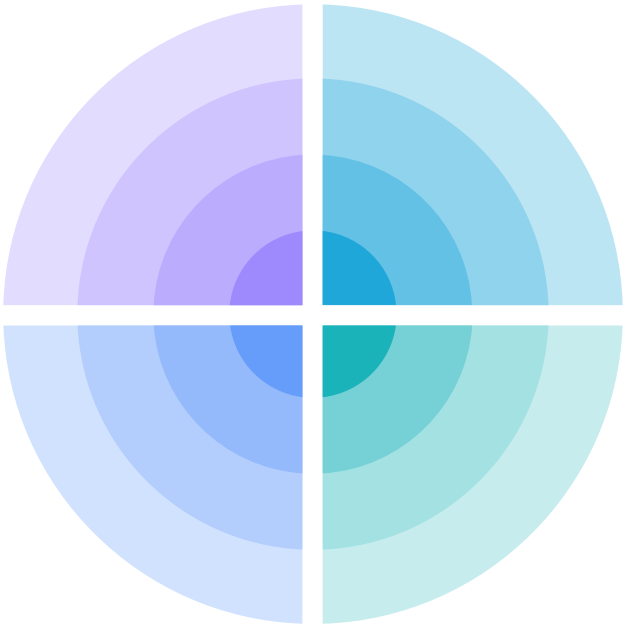
FOCUS 100% MOBILE

Édition été 2022

Tech Radar

by



REACT NATIVE FLUTTER NATIVE TRANSVERSE

Introduction

Co-construit à l'occasion de plusieurs ateliers, notre Tech Radar présente un focus 100 % mobile, inspiré du retour d'expérience de nos 3 tribes BAM et retranscrit par 15 contributeurs.

Vous y retrouverez 54 blips sélectionnés par notre équipe tech. Nous qualifions de "blip" une technologie ou une technique présente sur notre Tech Radar, qui joue un rôle dans le développement des apps mobiles. La double ambition portée par ce projet est à la fois d'apporter un regard critique pour vous guider dans vos choix de transition de technos, tout en vous invitant à une discussion ouverte : contactez-nous, nous serions ravis d'avoir vos retours et d'échanger sur nos visions et expertises tech !

Sommaire

01. Les origines

En préparant notre Tech Radar, j'ai retrouvé une newsletter du radar de ThoughtWorks dans ma boite mail, datée du 17/12/2014. Deux jours avant l'inauguration officielle de l'entreprise, j'étais déjà abonné ! J'ai en effet découvert le Tech Radar bien avant de créer BAM. J'ai l'impression qu'il était avec moi pendant toute ma carrière de Dev. **Ce Tech Radar a toujours été une source d'inspiration et d'apprentissage. Et depuis que je l'ai découvert, l'objectif de construire une équipe capable de produire une telle source de connaissances ne m'a jamais quitté !**

C'est la quatrième fois que nous faisons un Tech Radar chez BAM. Les éditions précédentes étaient toutes très différentes. L'une d'entre elles a pris la forme d'un whiteboard, alimenté par des post-its de toute l'équipe Tech. Une autre utilisait l'outil open source de ThoughtWorks, couplé à un fichier Excel créé pendant la pause lunch un vendredi. Une autre a été élaborée par un comité scientifique, créé ad hoc et composé des techs les plus expérimentés de l'équipe. Aucune de ces éditions n'a été une grande réussite. L'énergie n'était pas là, les résultats étaient mitigés. Cela nous a permis d'avoir quelques discussions intéressantes autour des technos, mais au bout de trois éditions j'ai arrêté de pousser.

Cette année, en lisant les différents retours de ThoughtWorks sur leur radar (la rétrospective, le podcast sur comment ça se passe réellement), j'ai compris ce qui ne fonctionnait pas : notre radar était toujours très orienté vers une communication interne, où le besoin n'était pas présent. Un radar c'est un investissement important. Un offsite de quelques jours s'impose, avec une préparation, un bon cadre et une organisation en mode projet. Je me suis dit que c'était peut-être le bon moment pour réessayer. Depuis un certain temps, BAM a réussi à devenir l'une des plus grandes agences mobiles en France. Nous avons aussi choisi une approche agnostique à la technologie. **En partant de notre stack historique en React Native, nous avons commencé à livrer des projets en Flutter et en natif, ce qui nous permet aujourd'hui de toucher les trois approches les plus importantes.** Au-delà des projets que nous réalisons, plusieurs personnes nous consultent pour connaître notre avis : sur les technologies à choisir, la méthodologie de travail et les difficultés à anticiper. J'espère que la première édition publique de notre radar nous permettra de **mieux diffuser nos apprentissages, d'amorcer une discussion avec la communauté mobile sur ces sujets-là et de contribuer à la communauté dans un esprit open source.** Je vous invite à découvrir le travail de notre équipe et je vous souhaite une excellente lecture !"

Marek / **CTO et co-fondateur**

02. Atelier de création

En nous fixant l'objectif de publier notre vision de l'écosystème mobile, nous avons fait quelques choix éditoriaux qui ont fortement influencé le contenu :

• **Transmettre notre expertise,** en parlant des technologies que nous utilisons au quotidien, que nous avons expérimentées ou que nous suivons depuis un certain moment.

• **Ne pas aborder des choix évidents :** certaines approches ou technologies sont clairement adoptées par une grande majorité de la communauté, parfois même mentionnées dans la documentation officielle d'une technologie. Si nous estimons que notre avis n'apportera pas de plus-value sur une technologie, vous ne la retrouverez pas sur le radar.

• **Être clair sur nos recommandations :** pour mieux vous guider, nous avons opté pour des prises de position assumées.

À titre d'exemple, pour une technologie en « adopt », allez-y les yeux fermés; à l'inverse, si le niveau de recommandation d'une technologie est « hold » ce n'est pas un jugement de valeur. Cela signifie que nous ne l'utilisons pas et nous vous recommandons de ne pas l'employer pour l'instant. Cela n'exclut pas la nécessité de tenir compte de certaines subtilités, mais cela nous permet d'être plus clairs. Vous ne retrouverez pas deux solutions équivalentes en « adopt ». Lorsque nous n'arrivons pas à trancher, nous les positionnons en «trial» et vous recommandons de considérer les deux !

D'ailleurs, s'il peut vous paraître surprenant de ne pas retrouver quelques blips (technologies ou techniques), nous en avons délibérément écarté certains : soit parce qu'ils présentent selon nous un niveau d'adoption trop évident et avéré pour le moment, soit parce que nous ne les avons pas suffisamment expérimentés à ce stade.

Notre Tech Radar a été grandement inspiré du travail de ThoughtWorks et du contenu créé autour de ce modèle (articles de blog, podcasts, interviews). Nous avons également adopté une approche en 3 étapes :

• **Une phase de recommandations,** émises par les Heads Of Tribe de React Native, Flutter et Native. Ces derniers ont consulté leurs équipes pour réunir des recommandations plus proches du terrain.

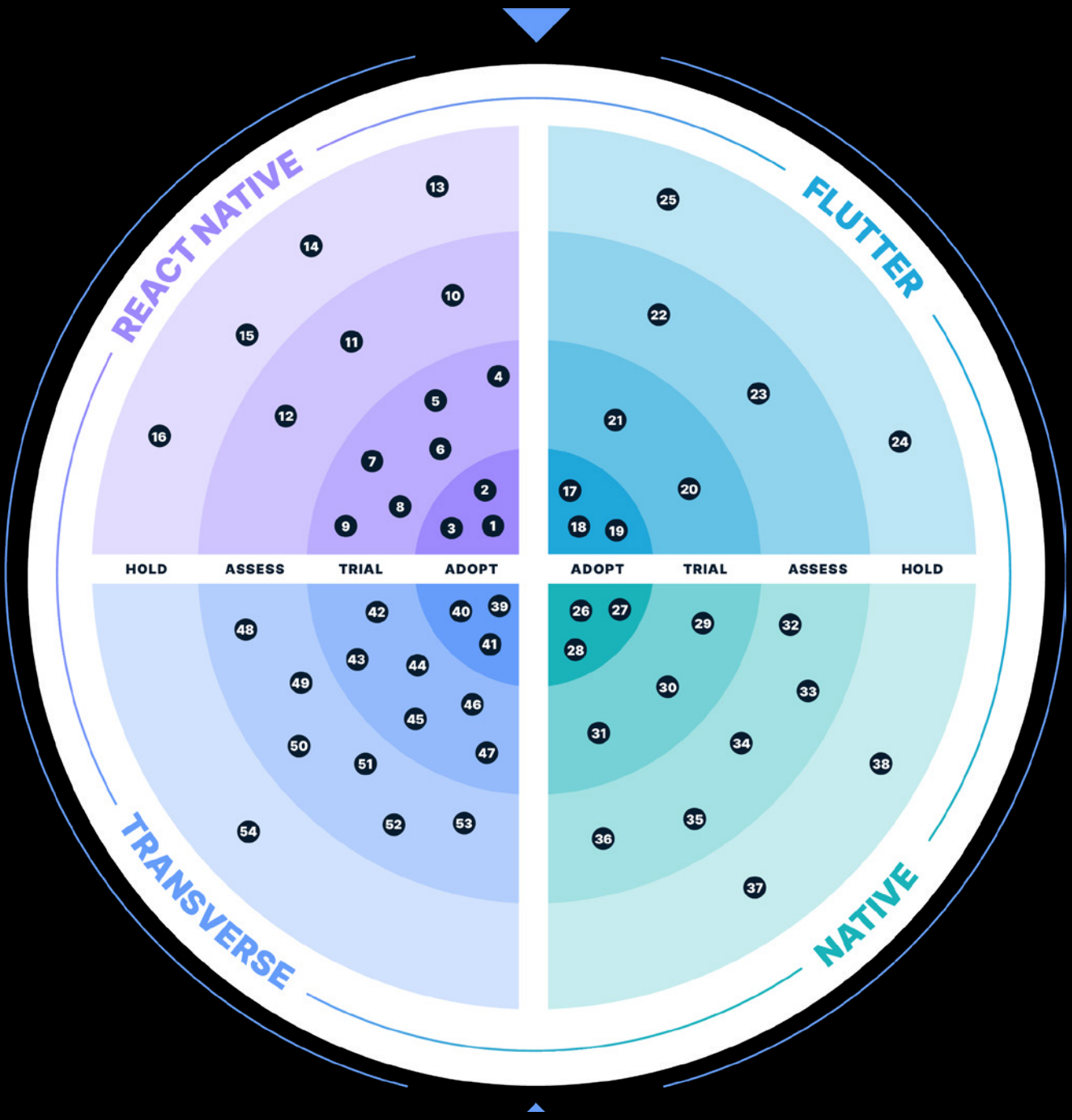
• **Une phase exploratoire,** avec deux jours d'offsite aux côtés des leaders techniques de BAM. Au programme, pour chaque blip : clarifier le contenu, pitcher aux membres d'équipes qui ne connaissent pas la techno et décider si le blip mérite sa place dans le radar, avant d'organiser un débat pour s'aligner sur le niveau de recommandation. Le niveau de recommandation de certains blips a fait tout un voyage de « adopt » à « hold » et inversement ! Cette démarche a été répétée plus de 100 fois, pour traiter toutes les recommandations émises par nos Tribes.

• **Une dernière phase** a été dédiée à la **création de contenu :** avec une rédaction des blips par les membres de la Tribe techno, ensuite challengés par au moins 4 personnes.

En quelques chiffres, ça donne quoi ? 135 blips discutés, 54 blips retenus dans le radar, 14 personnes impliquées dans la rédaction sur environ 200h de travail et encore 50h pour la mise en forme. Nous avons ainsi à la fois très hâte, tout en étant un peu anxieux en mettant ce radar entre vos mains !

03. Le Tech Radar

Dans ce tech radar 100% mobile, nous vous partageons notre avis d'expert sur les techniques, plateformes, outils, langages et frameworks associés aux principales technologies que nous utilisons au quotidien chez BAM : React Native, Native et Flutter.



4 NIVEAUX D'ADOPTION

ADOPT
nous recommandons l'usage de ces technos.

TRIAL
ces technos sont prêtes à l'emploi, mais nous pensons qu'elles pourraient encore être optimisées.

ASSESS
nous vous invitons à vous documenter sur ces technos si elles répondent à un besoin spécifique.

HOLD
nous ne sommes pas convaincus par ces technos et ne recommandons pas leur utilisation à ce stade.

04. Les Blips

REACT NATIVE

ADOPT

1. Flipper
2. Hermes Engine
3. React Query

TRIAL

4. Expo
5. Create React Native modules using JSI
6. React Native Web
7. Reanimated v2
8. API validation with runtypes
9. Storybook

ASSESS

10. Jotai
11. React Native Skia
12. WatermelonDB

HOLD

13. Enable Fabric (on React Native apps)
14. Build React Native forms with no libs
15. Redux Saga by default
16. Styled Components

FLUTTER

ADOPT

17. Mocktail
18. OpenAPI Generator for Flutter
19. Pigeon

TRIAL

20. BLoC
21. Riverprod

ASSESS

22. Alchemist
23. Melos

HOLD

24. GraphQL with Flutter
25. Provider

TECHNOLOGIES NATIVES

ADOPT

26. Jetpack compose
27. Koin
28. Swift UI

TRIAL

29. Resolver
30. Tuist
31. VIPER

ASSESS

32. App Code
33. Composable Architecture
34. µ-Features Architecture
35. Using relation db as state management
36. Texture

HOLD

37. KMM
38. Litho

TRANSVERSE

ADOPT

39. ADR
40. BPMN
41. QRQC

TRIAL

42. Appium
43. Clean archi / Onion architecture
44. CVSS Security audit
45. Instabug
46. 4 Key metrics
47. Observable

ASSESS

48. App Auth
49. Cognito
50. CRDT (yJs, Automerge)
51. Ghost Programming
52. Github Copilot
53. Libsodium

HOLD

54. Firebase Authentication

05. Le choix de la technologie mobile

Vous allez sûrement constater que notre radar n'émet pas de recommandation claire entre Flutter vs React Native vs Native. Les 3 technologies définissent les cadrans, mais on ne les compare nulle part.

Quel est le meilleur choix ?

Lorsque nous avons lancé BAM en 2014, le choix technologique était très risqué. Nous étions convaincus que le multiplateforme représentait le futur du mobile et nous avons choisi notre stack technologique, composé de Cordova et Ionic. Mais cette décision n'était pas évidente en raison des nombreux concurrents de Cordova, comme Xamarin (soutenu par Microsoft) ou Titanium (approche qui utilise des composants natifs en UI). Quelqu'un se souvient encore de Famo.us ?

Chacune de ces solutions avait quelques points forts bien distincts et beaucoup d'inconvénients. Le paysage a changé avec React Native en 2015 qui est parvenu à adresser la majorité des désavantages d'autres frameworks. Nous l'avons adopté dès octobre 2015. Flutter, sorti deux ans plus tard, a embrassé une approche techniquement différente, mais avec le même niveau de qualité. Après un certain temps, la solution a su montrer sa valeur. En parallèle, l'émergence et la montée en popularité de Swift et Kotlin ont apporté beaucoup de fraîcheur et de modernité dans le développement natif, ce qui a fait perdre certains arguments au développement multiplateforme.

Nous sommes donc passés d'une phase où le choix n'était pas évident (avant l'été 2015), suivie d'une phase où le choix était plutôt clair (2015 - 2018) avant de devenir à nouveau très complexe. Pour chacun de nos projets, nous prenons une décision spécifique pour choisir la technologie.

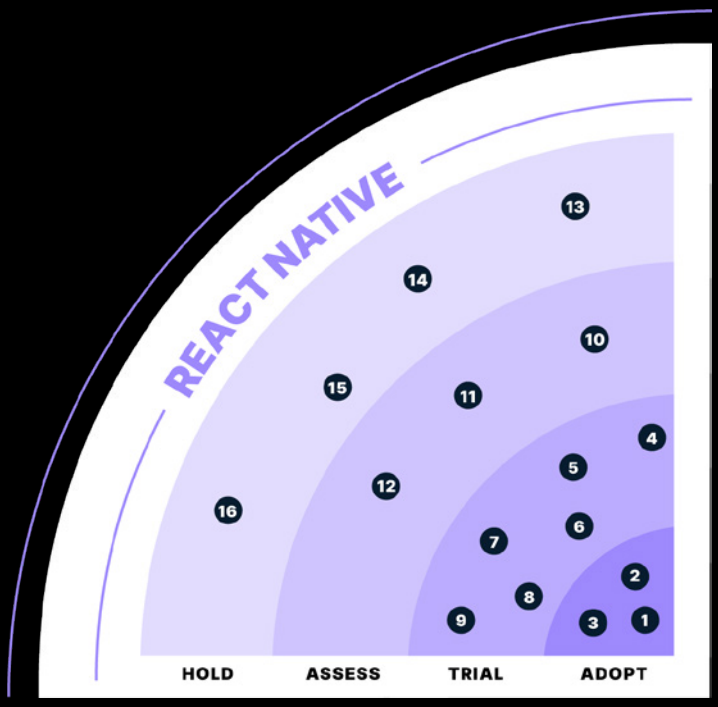
Une discussion qui doit prendre en compte :

- **La stratégie produit :** quelles fonctionnalités, quel design, qui seront les utilisateurs ?
- **La vision Tech :** quelle vision technique de l'entreprise, qui va travailler sur ce projet, quelles sont les équipes existantes, quelle stratégie de recrutement ?
- **Le budget :** quel est le montant qu'on peut investir, sur quelle durée et quelles sont les conditions de financement du projet ?

Cet échange nous permet d'évaluer le projet par rapport aux 3 technologies et de faire une recommandation, plus ou moins forte en fonction des contraintes. Pour résumer, nous estimons qu'il n'est pas possible de vous dire avec certitude " la technologie A est mieux que la technologie B ". La décision doit se faire au cas par cas avec des inputs de toutes parties prenantes et des experts mobiles qui connaissent les différentes solutions.

Discutons autour d'un café, si vous voulez notre recommandation personnalisée pour votre app et votre entreprise.

06. React Native



16 BLIPS

3 ADOPT
6 TRIAL
3 ASSESS
4 HOLD

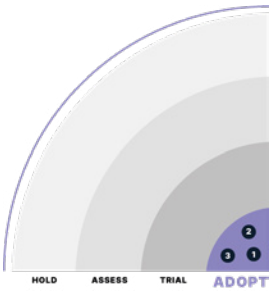
À retenir

Coté React Native, nous avons choisi de présenter les innovations apportées dans l'écosystème. **React Native continue à être une solution de choix pour le développement d'applications multiplateformes.** Elles couvrent notamment :

- Les nouveautés "opt-in" de React Native qui rende l'app plus performante
- Les librairies et architectures pour gérer le state
- Les librairies d'UI et de graphiques
- Les outils de déploiement ou de debugging

Les innovations présentées dans ce cadran font aujourd'hui partie de notre stack technique de référence."

Cyril / **Staff Engineer**



1 Flipper

Dans une application développée en React Native, le code JavaScript est exécuté par un moteur à l'intérieur du téléphone. Pour débogger une application, l'approche historique était d'activer le mode debug de React Native. Ce mode permet de déporter l'exécution du code Javascript du téléphone vers un debugger (comme React Native Debugger ou l'inspecteur JS sur Chrome) afin de prendre le contrôle du code exécuté.

Cela présente deux inconvénients majeurs : les mêmes problèmes ne sont pas toujours reproduits avec ou sans le mode debug et l'exécution est fortement ralentie.

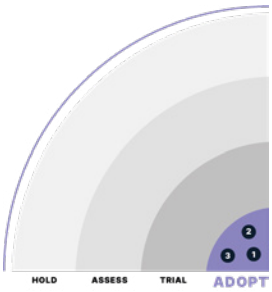
Flipper est un outil d'inspection et de débogage d'application mobile créé par Facebook. Il peut être utilisé pour du React Native ou pour des applications natives. **Chez BAM, nous l'employons majoritairement sur nos applications React Native.**

Avec Flipper et Hermès (un moteur javascript optimisé pour React Native), **le debugging s'opère directement au niveau du téléphone. Cela rend l'opération plus rapide et surtout plus fiable.** Flipper utilisant Hermès pour son debugger, l'activation d'Hermès est obligatoire sur l'application à déboguer.

Au-delà du debugging JavaScript, Flipper présente d'autres avantages. **L'outil propose différents plugins d'inspection de réseau, de layout et de mémoire.** Il peut se brancher à des librairies comme react navigation ou react query, qui possèdent un plugin Flipper.

De plus, **Flipper étant très extensible, il est facile d'ajouter des plugins faits maison.** Cela permet notamment d'avoir des mesures spécifiques à l'application développée, ou encore des interactions spéciales pouvant aider les développeurs. A titre d'exemple, on peut imaginer un plugin pour activer/désactiver quelque chose dans l'application. **Chez BAM, nous avons notamment développé un plugin pour** inspecter la performance de nos applications**.**

Nous recommandons aujourd'hui Flipper comme debugger par défaut sur nos projets React Native. Si son installation était anciennement compliquée, en raison de dépendances natives, ce n'est plus le cas aujourd'hui. Flipper répond bien mieux à nos besoins d'outillage que React Native debugger.



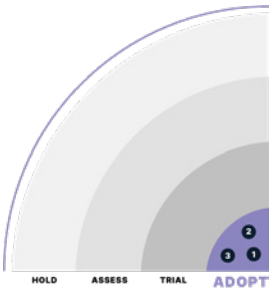
2 Hermes Engine

Le temps de démarrage des apps a longtemps été un point noir de React Native, surtout sur des téléphones Android bas de gamme. À titre d'exemple, l'une de nos apps, qui contenait un code JS conséquent, démarrait en 12.9s sur un téléphone bas de gamme.

Pour remédier à ce problème, Facebook a implémenté Hermes, un nouveau moteur JavaScript. Au lieu de parser et compiler le JS en code binaire au lancement de l'app (comme un moteur JS traditionnel), Hermes le fait pendant le «build time» de l'app.

Ainsi, une app sous Hermes n'a plus de fichier JS embarqué, mais un fichier de code binaire. Les résultats sont frappants : en utilisant Hermes, l'app mentionnée précédemment démarre maintenant en 3.9s, et le temps de démarrage de toutes nos apps a été divisé au moins par 2. Passer enableHermes à true dans la configuration et ajouter quelques polyfills (ex: i18n, regexp) vous permettront de profiter de cette amélioration.

Chez BAM, nous activons Hermes sur tous nos projets et nous vous recommandons de faire de même. Les bénéfices surpassent les coûts, car sans Hermes, le temps de démarrage de vos apps Android ne sera probablement pas à la hauteur des recommandations de Google.



3 React Query

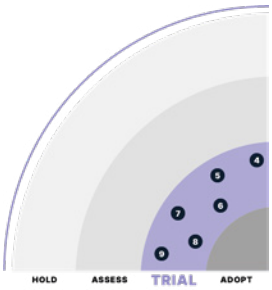
Si de nombreux développeurs sous-estiment la complexité associée aux appels asynchrones, ces derniers constituent en réalité un problème très vaste et complexe (gestion de l'état de chargement, de l'état d'erreur, du cache, du refetch). Une implémentation simple et performante peut s'avérer très complexe.

React Query est présentée comme une librairie de hooks pour charger, mettre en cache et modifier des données asynchrones en React. Elle offre un large éventail de fonctionnalités : une simple requête avec un état d'erreur et de chargement, des requêtes dépendantes ou en parallèle, des requêtes paginées. Elle intègre également une gestion simple et efficace du cache et assure une compatibilité avec du rendu côté serveur et notamment NextJS.

En revanche, le choix a été fait de ne pas normaliser le cache. Cette action doit donc être réalisée manuellement si nécessaire.

Chez BAM, cela fait bientôt deux ans que nous avons commencé à utiliser React Query. Cette librairie est très simple à prendre en main. Nous vous conseillons tout de même de porter attention à la configuration par défaut, car la durée du cache et le nombre de tentatives des requêtes en cas d'erreur peuvent varier selon les projets.

Nous avons constaté que cette solution nous a permis d'implémenter beaucoup plus rapidement nos appels asynchrones, avec une meilleure qualité, ce qui nous a permis de concentrer nos efforts sur des fonctionnalités métier. Nous l'utilisons maintenant sur tous nos projets qui n'ont pas d'API Graphql, et nous vous conseillons de faire de même.



4 Expo

L'univers mobile n'est pas toujours évident à prendre en main pour des développeurs qui n'ont pas d'expérience dans le domaine. La nécessité de comprendre les spécificités de chaque OS, leurs APIs et la manière de signer et déployer les applications peuvent constituer une véritable barrière à l'entrée.

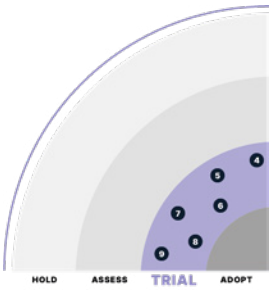
Expo est une plateforme qui permet de développer des applications React Native en atténuant grandement cette barrière. Elle permet de générer une application en « managed workflow » qui abstrait toute la partie native et dont il est possible de s'éjecter (à la manière de create-react-app). Cette plateforme présente de nombreuses fonctionnalités, qui simplifient le quotidien des développeurs, comme :

- La gestion automatique de la signature des applications : les connaissances sur le système de signature sur iOS et Android ne sont donc pas prérequises
- Un système de mise à jour « Over The Air »
- Un service de déploiement continu (Expo EAS) intégré
- Une multitude de packages permettant d'utiliser les APIs natives
- L'intégration de React Native Web : les applications développées avec Expo peuvent être déployées sur le web

Nous avons eu une bonne expérience avec cette plateforme, utilisée sur plusieurs de nos projets en production.

Un point de vigilance cependant : en utilisant Expo, on devient très dépendant du service. Il nous est arrivé de ne pas pouvoir déployer notre application pendant plusieurs heures, car cette plateforme était indisponible. De plus, il est maintenant possible de modifier le code natif même en « managed workflow » via des patchs moyennement faciles à maintenir.

Ce n'est donc pas une solution idéale pour des projets qui requièrent beaucoup de modifications du code natif. Nous conseillons plutôt Expo aux équipes qui travaillent sur un projet de petite à moyenne taille et qui n'ont pas ou peu de connaissances dans le domaine du mobile.



5 Create React Native modules using JSI

React Native est un framework basé sur React, qui permet d'implémenter des applications natives iOS et android à partir d'une seule base de code. Pour ce faire, deux pans techniques sont nécessaires :

- Le pan technique javascript, où l'interface graphique de l'application est définie à l'aide de composants génériques
- Le pan technique natif, où ces mêmes composants génériques sont associés en composants natifs, spécifiques à chaque plateforme

En informatique, il existe deux grandes stratégies pour faire communiquer deux pans techniques :

- La sérialisation de messages (ex: le protocole HTTP avec des messages au format XML ou JSON)
- Les Foreign Function Interface (FFI) qui permettent à partir d'un langage donné d'appeler des méthodes implémentées dans un autre langage (ex: l'objet DOM et ses APIs comme document. getElementById() sont disponibles en Javascript mais sont implémentés en C++ au niveau des navigateurs)

Pendant des années, React Native a utilisé des messages JSON sérialisés, envoyés à travers un bridge pour faire communiquer les pans techniques JS et natif.

Ce système a ses limites, car lorsque le débit d'information en transit entre les deux pans techniques est trop important (animations, évènements de scroll sur de longues listes...), le bridge se congestionne. **On constate alors des problèmes de réactivité de l'application.**

Pour pallier ces problématiques, la nouvelle architecture React Native dispose d'une brique appelée JSI, une Foreign Function Interface qui relie le runtime JS au domaine natif.

Cette interface standardisée est déjà implémentée dans certains moteurs JavaScript comme JSC et Hermes. Son utilisation permet d'améliorer de manière significative la performance. Certaines librairies ont déjà tiré profit de JSI :

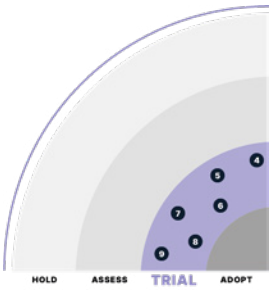
- React-native-mmkv, qui offre une api synchrone et 20 fois plus performante que react-native-async-storage.
- React-native-bignumber, qui rend disponible au monde JS l'implémentation C++ de bn, une librairie d'arithmétique sur des grands nombres entiers, proposée par OpenSSL. Cette implémentation a été mesurée comme 300 fois plus performante que son implémentation équivalente en JS.

Cependant, JSI est également source de complexité, car la liaison entre le pan technique natif et le pan technique JS de votre module devra être implémentée en C++. De plus, la documentation est encore peu fournie et le développement d'API asynchrones est complexe, ce qui peut freiner son déploiement à grande échelle.

Si le module que vous souhaitez implémenter possède des interfaces simples (manipulant des types sérialisables), des outils comme codegen permettent de générer le code C++ lié à JSI, à partir d'une interface de votre module écrite en Typescript ou en Flow.

Sinon il vous faudra créer vos fichiers de liaison en C++, utiliser les API de JSI et créer vos propres hostObjects pour pouvoir implémenter vos modules natifs.

Aujourd'hui, nous expérimentons la migration de notre module le plus connu, react-native-image-resizer, vers JSI, et nous vous conseillons de l'étudier pour vos modules natifs.



6 React Native Web

Une application étant souvent disponible à la fois sur le web et sur le mobile, il est naturel de chercher à partager du code entre ces plateformes. Il est possible de partager de la logique métier, du state d'UI et des call API entre une application web React et une application mobile React Native, mais il n'est pas possible de partager des composants UI par défaut.

React Native Web est une implémentation des composants et APIs React Native, compatible avec React DOM.

L'installation est très simple : il suffit d'installer react-native-web en dépendance de l'application web, et d'ajouter un alias de react-native vers react-native-web dans la configuration du bundler. Il est alors possible d'importer tout composant codé en React Native dans son application web React.

Nous avons utilisé React Native Web sur plusieurs projets, ce qui nous a permis de partager entre 75% et 95% de code entre le web et le mobile. Notre retour d'expérience est très positif. Il est toutefois important de garder deux choses en tête :

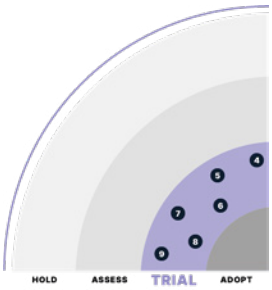
- Le code produit par React Native Web est plus complexe que du code React pur. Cela peut impacter la performance ainsi que la facilité de débogage, même si nous avons constaté que l'impact sur la performance est en général imperceptible pour l'utilisateur.
- La mise en page est souvent assez différente entre le web et le mobile, et il est souvent plus pertinent de partager certains composants d'une page, plutôt qu'une page dans son intégralité. Certains comportements, très spécifiques au web ne sont pas pertinents à partager.

Il peut être intéressant d'utiliser cet outil à condition de définir une stratégie claire, qui permet de dissocier quels sont les composants qui l'utilisent de ceux qu'il sera nécessaire de coder manuellement pour chaque plateforme.

React Native Web est utilisée par de nombreuses entreprises, comme Twitter (dont faisait partie le créateur de la librairie) et Uber. Cette librairie est également supportée par Expo, qui l'inclut par défaut dans ses applications.

En revanche, cette solution a été essentiellement développée par un seul développeur, ce qui présente un risque au niveau de la maintenance.

Bien qu'il soit raisonnable de penser que les entreprises l'utilisant pourraient reprendre la maintenance, ce risque nous fait positionner React Native Web dans la section trial.



7 Reanimated v2

La création d'animations en React Native est parfois difficile. Bien que l'API Animated, fournie avec React Native soit simple, le code est impératif et a tendance à complexifier les composants. Le coût d'implémentation est élevé et par conséquent, les animations ne sont pas priorisées dans le développement de nos applications.

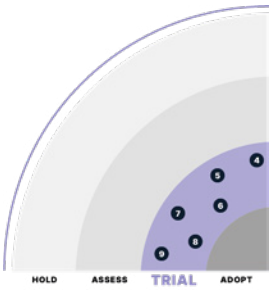
La librairie *Reanimated* change la donne à partir de la version 2. Grâce à son approche plus déclarative, les animations deviennent très simples à implémenter. Le code d'animation ne nuit pas à la lisibilité du reste du composant. En cas d'animation plus complexe, l'approche par les hooks de *Reanimated* permet d'extraire facilement le bloc complexe dans un hook isolé. Les hooks permettent également d'extraire les animations en blocs réutilisables.

De plus, les *Layout Transitions* de *Reanimated* rendent les animations d'apparition, de disparition ou de changement d'un élément dans une liste peu complexes. Il s'agirait de tâches très complexes sans librairie.

Reanimated va exécuter le code Javascript des animations dans le thread UI. Cela va permettre d'obtenir des animations fluides, sans problème de performance.

Globalement, *Reanimated* améliore la qualité du code d'animation par rapport à *Animated* de React Native. De plus, cette librairie améliore la qualité du produit en incitant à implémenter des animations.

Malgré les bénéfices apportés par les animations pour le produit, nous ne positionnons Reanimated qu'en trial car nous observons encore quelques problèmes. A titre d'exemple, sur certains projets, nos équipes ont observé des crashes ou des animations qui ne se déclenchent pas sur les builds release. Cependant, la librairie évolue vite : les nouvelles features arrivent assez vite et des bugs sont résolus fréquemment.



8 API validation with Runtypes

Sur les cinq dernières années, Typescript a changé le paysage javascript, en accédant même au top-5 des langages les plus utilisés en 2022 dans le sondage stack-overflow.

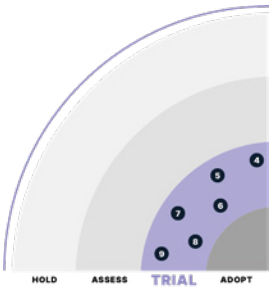
Sa promesse est de vérifier que si les données en entrée sont bonnes, les données en sortie le seront également. Cela débouche sur plusieurs interrogations :

Qui vérifie que les données en entrée sont correctes et comment intégrer cette vérification pendant l'exécution de notre application, alors que la vérification du typage se fait pendant la phase de compilation ?

La librairie Runtypes permet de générer des types et validateurs associés à partir d'un schéma de la donnée. Ces validateurs permettent de vérifier que les données reçues sont bien conformes au type attendu par une application et de la protéger d'une donnée mal formatée.

Nous vous recommandons de mettre en place un système de validation de types afin d'assurer la sécurité du typage de bout en bout. La bibliothèque Runtypes est très puissante, parfois trop (Contracts, Branded Types, Constraints) ce qui peut être à double tranchant : si les développeurs séniors peuvent s'en servir à juste titre, cela complexifie l'onboarding des développeurs juniors.

Nous vous conseillons d'introduire Runtypes petit à petit, en limitant son utilisation au strict nécessaire pour commencer.



9 Storybook

Les technologies web et mobile actuelles (Flutter, Jetpack Compose, SwiftUI, React, Vue) nous encouragent à pratiquer le design atomique. Cette méthode de conception consiste à construire nos UI en partant des composants plutôt que des pages.

Storybook est un outil de développement d'interfaces graphiques en isolation disponible en web et React Native, qui répond à la dynamique du design atomique. Il met à disposition des développeurs un environnement (les stories) dans lequel les composants peuvent être développés indépendamment de l'application dans laquelle ils doivent être utilisés.

Storybook permet ainsi d'améliorer la parallélisation des développements et d'augmenter la robustesse des composants, avec un focus "component first". Cet outil permet également de documenter et de structurer les composants du design system. Malheureusement le tooling storybook en React Native n'est pas aussi avancé qu'en web (Storybook server, chromatic).

Addon React Native Web permet de configurer Storybook pour afficher les projets React Native en utilisant React Native Web.

C'est une solution très intéressante si votre projet React Native a vocation à être compatible web. Dans le cas contraire, elle constitue un coût de développement et de maintenance conséquent.

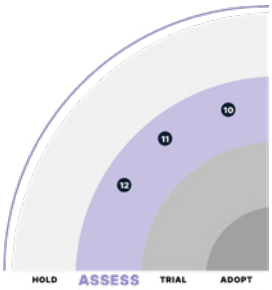
Contrairement à l'environnement web qui vous permet de publier votre storybook en site statique, React Native requiert de builder et déployer une application dédiée sur une plateforme de distribution (par exemple AppCenter) pour rendre votre storybook accessible à l'ensemble des parties prenantes du projet.

Enfin, des versions de Storybook/React Native posent des problèmes de compatibilité avec Storybook, rendant la mise à jour de la dépendance compliquée. A titre d'exemple, à l'heure où nous rédigeons ce document, Storybook/React Native n'est pas compatible avec la version 6.4 de Storybook.

Chez BAM, les retours d'expérience sur cette librairie varient d'un projet à l'autre. Pour certains projets, nos équipes ont rencontré des problématiques de déploiement liées à storybook, ce qui les a contraint à concevoir un système de gestion de bibliothèque de composants intégralement.

À l'inverse, d'autres projets utilisant React Native et React Native Web ont pu tirer profit des avantages du tooling de chacune des plateformes pour améliorer le développement et la validation de leur design system.

Pour cette raison, nous avons positionné Storybook en trial. Une chose est sûre : l'écosystème autour de Storybook évolue rapidement. L'écart entre le tooling React Native et web devrait donc réduire avec le temps.



10 Jotai

Un state global est important pour éviter des incohérences UX, comme un counter de messages non lu pas synchrone avec les messages. Cependant, React ne propose pas de gestion de state global : il est possible de propager un state local à travers l'arbre de composant via les «props» mais c'est très lourd et peu maintenable.

Redux propose une première solution à ce problème. Via une approche *flux unidirectionnel* il permet de concevoir un state global. Nous l'avons adopté en 2016, au point de l'utiliser par défaut sur tous nos projets. Nous avons cependant constaté deux problèmes :

- D'une part, un boilerplate très conséquent, même avec « redux-toolkit ».
- D'autre part, une approche « centrale » de la donnée, qui implique une bien meilleure conception en amont.

Entre-temps, l'écosystème React a évolué. À titre d'exemple, les hooks simplifient la syntaxe pour gérer le state local. Et React-Query ou Apollo, qui permettent de gérer les appels API, facilitent la gestion des données distantes.

Dans ce contexte, Redux est moins utile et des alternatives plus simples et plus modernes ont émergé. Plutôt qu'un grand state monolithique, nous recommandons une approche atomique telle que le hook useState de React : dans un fichier partagé on définit un atom (exemple const userAtom = atom(null)) et on peut ensuite remplacer les useState locaux par useAtom(userAtom).

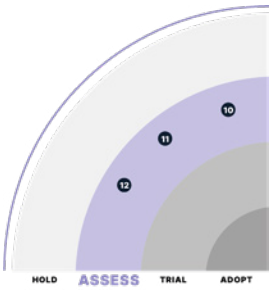
Cette alternative est plus simple, plus proche de l'API des hooks de state react, et demande moins de conception en amont pour designer le state.

Sur nos projets, nous avons utilisé Recoil et Jotai qui partagent une approche similaire.

Nous préférons Jotai pour plusieurs raisons :

- La stabilité de son API et sa documentation
- Ses utilitaires pour simplifier l'expérience développeur
- La performance qui optimise les re-rendus
- Sa simplicité d'utilisation avec Typescript
- Sa compatibilité avec Suspense et le mode concurrent de React
- Son intégration avec d'autres projets comme XState, react-query et Immer.

Nous pensons que l'approche de Jotai est complémentaire avec l'utilisation de React-Query ou d'Apollo pour gérer le state distant et que 95% des projets n'ont plus besoin de Redux. Nous vous conseillons d'essayer la combinaison «Jotai» et «React-Query»/»Apollo» pour votre prochain projet.



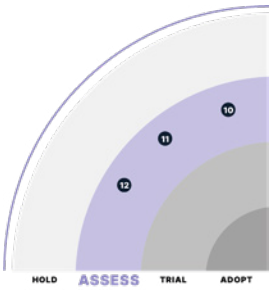
11 React Native Skia

React Native Skia est une librairie de rendu graphique 2D haute performance. Elle se base sur Skia, le moteur de rendu graphique utilisé par Flutter et Jetpack Compose.

Cette librairie permet notamment d'afficher des images vectorielles, des graphes ou encore des filtres, du floutage et des shaders sur vos applications. Enthousiasmant d'observer tout ce que la communauté React Native réalise avec cette librairie, n'est-ce pas ?

Seul bémol : React Native Skia est en version «alpha» et la documentation actuelle mentionne que son utilisation doit faire l'objet de précautions. Ainsi, nous ne recommandons pas l'usage de cette librairie pour un projet en production à ce stade.

Cette librairie nous semble cependant être une solution prometteuse et performante qui pourrait, à terme, remplacer d'autres librairies comme react-native-svg. Nous avons en effet identifié des problèmes de performance relatifs à cette librairie.



12 WatermelonDB

Comme détaillé dans « Using relational db as state management », utiliser une base de données relationnelle dans une application mobile présente de nombreux avantages : dénormalisation, consistance, contraintes du schéma, transactions, index optimisant l'accès ou encore recherche textuelle...

Quelles sont les solutions les plus adéquates en React Native ?

Watermelon est un module natif pour React Native qui permet de lire, sauvegarder et rechercher de la donnée de façon réactive depuis la bibliothèque SQLite, qui constitue la base de données la plus utilisée au monde. Très utilisée dans l'éco-système embarqué et desktop, elle est également de plus en plus exploitée en mobile avec Room sur Android et Core Data sur iOS.

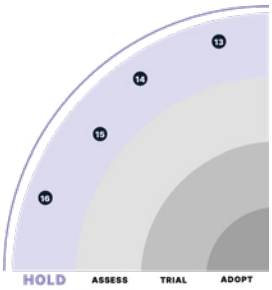
WatermelonDB fonctionne avec le paradigme réactif : en d'autres termes, lorsqu'une ligne change dans la base de données, les composants qui en dépendent sont automatiquement re-rendus.

Enfin, la base de données Watermelon permet la synchronisation avec un serveur distant : pour implémenter une meilleure expérience hors ligne, il suffit de fournir deux routes côté serveur pour tirer et pousser les changements en local.

Ce module présente cependant deux freins :

- WatermelonDB est actuellement en version 0.24. La librairie est stable et complète, mais l'API pourrait changer d'ici la version 1.0.
- Le support des hooks est en pause. Pour bien faire un hook qui peut lancer et interrompre un rendu React basé sur un changement d'état extérieur (base de données), il est nécessaire de disposer du composant React Suspense et du mode concurrent. Et ce dernier pointe juste le bout de son nez. En attendant, utiliser Watermelon peut avoir une saveur de 2017.

Nous vous invitons donc à surveiller les futures évolutions de cette libraire pour l'intégrer lorsqu'elle sera stabilisée.



13 Enable Fabric

Annoncée depuis 2018, la nouvelle architecture de React Native est enfin disponible et activable. Elle permet ce qui était jusqu'alors impossible, notamment :

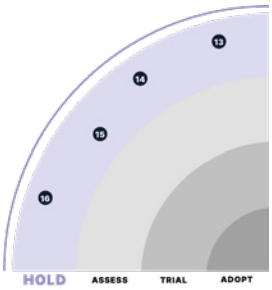
- Le support des nouvelles fonctionnalités de concurrence de React (Suspense pour le chargement des données, startTransition...) à partir de la version 0.69. Celles-ci permettent d'améliorer l'expérience utilisateur (ex: startTransition permet de garder l'app réactive même pendant une transition UI coûteuse)
- De mesurer et de rendre les vues natives de manière synchrones, ce qui permet d'éviter un effet de «saut» du layout
- De partager certaines optimisations jusqu'alors propres à une plateforme, grâce au partage de code en C++ (ex: le « View Flattening » qui était jusqu'à présent uniquement disponible sur Android)

Pour migrer vers cette nouvelle architecture, il suffit de changer un paramètre de configuration dans votre app. À noter qu'il faut tout de même que toutes les librairies de vos projets supportent cette nouvelle architecture.

Certaines d'entre elles, comme reanimated ou react-navigation ont déjà migré ou entamé la migration, mais cela ne s'est pas encore étendu à la plupart des librairies.

De plus, le fait d'activer la nouvelle architecture rallonge considérablement les temps de build natif (jusqu'à x5 en version 0.69), ce qui peut ralentir l'équipe. Nos tests traduisent également une dégradation de la performance sur certains composants, comme la FlatList, qui dans notre expérience en interne, consomme 20% de CPU en plus avec Fabric. Flashlist, le nouveau plugin de listes hautes performances par Shopify, pourrait pallier ces problèmes lorsque le support pour la nouvelle architecture y est ajouté.

Notre recommandation : cette nouvelle architecture n'est pas encore prête à être activée. Meta travaille cependant d'arrache pied pour améliorer la documentation et faciliter la migration des librairies et projets React Native. On espère donc pouvoir passer Fabric en « Adopt » dans notre prochain Tech Radar.



14 Build React Native forms with no libs

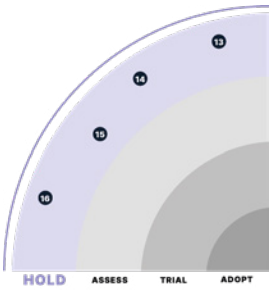
Les formulaires font partie des fonctionnalités présentes dans la quasi-totalité des applications. Conçus et pensés de manière très simple, il peut être tentant de vouloir les développer sans utiliser de librairie.

Pourtant, un formulaire n'est jamais simple. Il y a énormément d'états et de cas limites à prendre en compte pour chaque champ tels que : leur valeur, leur état de remplissage, leur validation, leur état d'erreur ou encore leur formatage.

Nous observons régulièrement des projets, pour lesquels les formulaires ont été délibérément développés sans librairie. Ces projets sont sujets à présenter des défauts de qualité, et à des formulaires dont la complexité va exploser avec le temps.

Pourtant, il existe aujourd'hui plusieurs librairies React qui facilitent grandement leur implémentation. Nous avons par exemple l'habitude d'utiliser les librairies Formik et React Hook Form, avec lesquelles nous avons eu de bonnes expériences. Ces librairies présentent plusieurs avantages : elles sont faciles à prendre en main grâce à une bonne documentation, et leur compatibilité avec des librairies comme Yup facilite la validation des formulaires.

Pour ces raisons, nous déconseillons fortement le choix de développer des formulaires sans librairie.



15 Redux Saga by Default

Pas si simple d'implémenter des appels APIs : à titre d'exemple, comment gérer les cas d'erreurs ou de chargement et ajouter un timeout ou un retry ?

Pendant plusieurs années, nous avons utilisé la librairie redux-saga pour répondre à ces problématiques. Depuis 2017, elle s'est distinguée à la fois pour la simplicité d'écriture de ses cas d'usage alors que la syntaxe async/await n'était pas encore répandue, mais aussi pour sa testabilité face à redux-thunk.

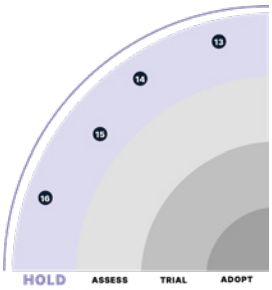
Cette solution reste cependant difficile à maîtriser pour plusieurs raisons :

- Elle requiert une familiarité avec la syntaxe des générateurs ES6, qui reste peu utilisée
- La création de sagas demande un boilerplate conséquent
- Son API très complète demande un effort non négligeable d'apprentissage (comprendre la différence entre spawn et fork par exemple n'est pas instantané)

De plus, React a évolué depuis 2017. Avec l'apparition des hooks en 2019 et l'arrivée de Suspense, de nouvelles solutions comme react-query commencent à émerger. Plus simples d'utilisation, elles répondent également à la majorité des besoins en asynchrones de nos projets.

En parallèle, redux-saga reste néanmoins un outil très puissant pour gérer une logique asynchrone complexe, avec toutes les bases pour en faire un système résilient : notamment avec spawning et le redémarrage automatique des sagas en cas d'erreur. A l'instar de techniques utilisées dans le monde des télécommunications (ex: Erlang, Akka stream). Elles requièrent néanmoins des connaissances techniques pointues et ces cas restent rares en pratique.

À l'heure actuelle, nous sommes d'avis de chercher une alternative plus simple que cette librairie, que nous ne recommandons pas d'installer par défaut.



16 Styled Components

En React Native, le style des composants se fait par défaut avec des StyleSheets. Cependant, l'utilisation de StyleSheets n'est pas adaptée lorsqu'on utilise un thème dynamique ou des styles conditionnés.

Styled-components est une solution de CSS-in-JS bien connue de l'écosystème web, qui inclut notamment une implémentation React Native. Son utilisation permet de profiter de nombreuses fonctionnalités telles que le theming, les styles en tant qu'objets ou encore la configuration du style via les props.

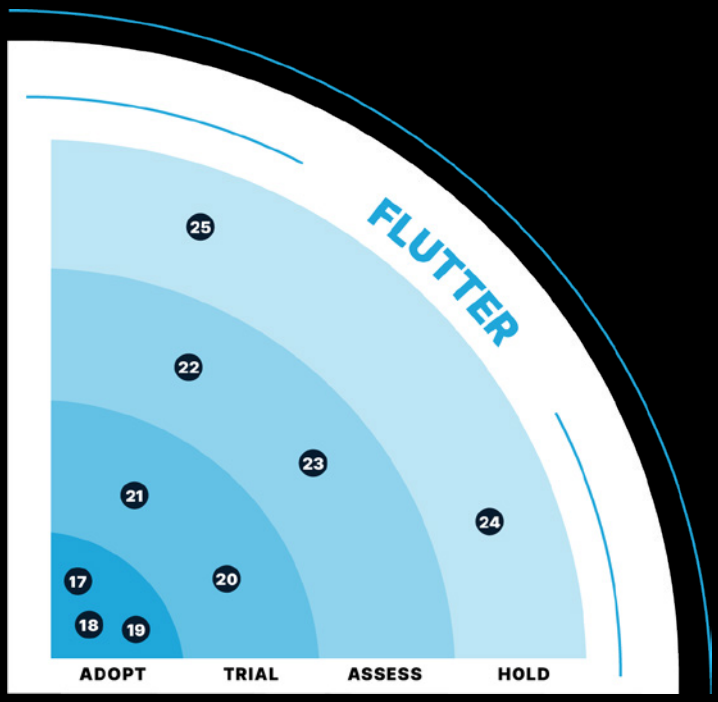
Styled-components nous encourage à séparer nos composants d'UI de nos smart components, favorisant ainsi la séparation des responsabilités.

Cependant, la librairie présente plusieurs inconvénients liés au typage. Nous avons observé des problèmes de performance rédhibitoires sur l'auto-complétion au niveau de l'IDE.

Ce problème a été identifié et corrigé à plusieurs reprises dans le passé. Malheureusement, il reste toujours d'actualité, ce qui n'est pas rassurant sur le court terme.

Nous recommandons de privilégier la bibliothèques CSS-in-JS Emotion, qui possède une api très similaire à Styled-components, ce qui simplifie grandement la migration. Emotion est une solution plus légère et plus rapide que Styled-components, et nous n'avons constaté aucun problème de typage à ce stade.

07. Flutter



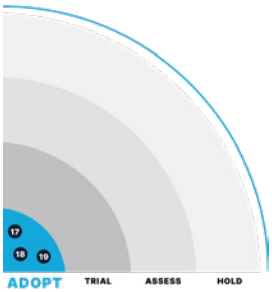
9 BLIPS

3 ADOPT
2 TRIAL
2 ASSESS
2 HOLD

À retenir

Côté Flutter, nous avons choisi de présenter des blips qui couvrent les problématiques principales de nos équipes depuis 1 an sur des grandes thématiques de l'écosystème Flutter : state management, appels API, tests et bridges natifs. Il s'agit de technologies que nous avons eu l'occasion d'expérimenter sur des projets, souvent jusqu'en production, et sur lesquels nous trouvons pertinent de donner notre retour d'expérience.»

Guillaume / **Head of Flutter**



17 Mocktail

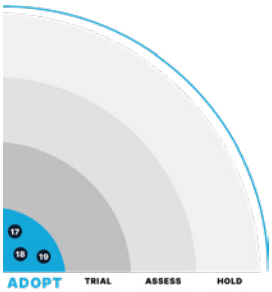
Dans une démarche de test unitaires, l'un des enjeux principaux consiste à mocker les dépendances des fonctions/méthodes testées, c'est-à-dire à passer en paramètre des objets qui imitent le comportement des variables attendues, sans pour autant les implémenter réellement. Dans l'écosystème Flutter et Dart, la bibliothèque la plus populaire est mockito.

Cette dernière propose une API très accessible pour mocker les classes et surcharger leur comportement en fonction du besoin de nos tests.

L'une des limitations de cette approche est qu'elle repose sur de la génération de code via le build_runner de Dart. Elle requiert de relancer la génération de code à chaque modification de nos classes, afin de mettre à jour l'ensemble des mocks. Bien qu'il soit très répandu dans l'écosystème Dart, Le principe de code génération est une approche qui reste assez controversée dans la communauté. En effet, elle ajoute une étape de friction dans les tâches de développement et introduit un code non «maîtrisé» dans la base de code.

Afin de pallier ce problème, Felix Angelov propose une solution alternative : Mocktail. **Mocktail est basée sur une API très similaire à celle de Mockito.** Cependant l'implémentation est différente car elle n'utilise pas la génération de code. Celà permet de mocker toutes les dépendances de nos tests, en bénéficiant de la nullsafety. L'expérience de développement devient ainsi bien plus fluide.

Cette approche a rapidement conquis à l'unanimité les experts Flutter de chez BAM pour devenir le standard de mock sur tous nos projets.



18 OpenAPI Generator

Toute application cliente s'interface avec un ou plusieurs serveurs, qui permettent de récupérer les données qu'elle affiche, et d'y envoyer les informations que l'utilisateur renseigne. Avec Flutter, la définition de clients API qui permet ces interactions peut être fastidieuse et nécessiter une grande quantité de code boilerplate. C'est notamment le cas pour créer une (dé)sérialisation JSON/dart des données échangées.

La solution OpenAPI Generator permet de générer automatiquement le code dart responsable de l'interfaçage, avec une API conforme au standard Open API 3.

Cela implique de générer un fichier .yaml de contrat d'API depuis le code back, afin de l'utiliser pour générer le client. Il est aussi possible de créer ce fichier manuellement si l'API respecte la norme.

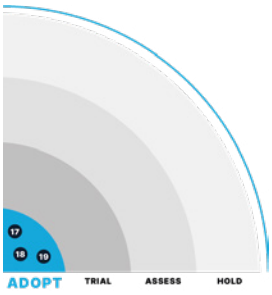
Cette approche présente de nombreux bénéfices :

- Garantir la qualité du contrat d'API en imposant le respect du standard OpenAPI 3 (ex : envoyer la valeur null plutôt qu'une String vide (''))
- Simplifier la (dé)sérialisation JSON/dart
- Isoler la définition des entités et faciliter les mocks et donc la testabilité de l'app

Après avoir employé cette approche sur plusieurs projets en production, nous avons cependant identifié **2 freins majeurs** à la mise en place de cette génération :

- Possible manque d'outil pour la génération automatique du contrat d'API .yaml (par exemple, en Go)
- Respect partiel ou non respect de la norme Open API sur certains projets

Si ces 2 freins sont bien adressés, on peut réduire la complexité de l'écriture d'un client API dart complet, en un unique fichier .yaml.



19 Pigeon

Lors du développement d'un projet Flutter, qu'il s'agisse d'une application ou bien d'un package, nous pouvons rapidement être amenés à nous interfacer avec du code natif mobile : iOS ou Android. Pour permettre aux développeurs de répondre à ce besoin, Flutter propose le système de PlatformChannel : des canaux de communication entre le code natif de la plateforme et le code dart de Flutter.

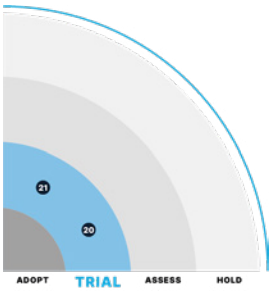
Cette approche, bien documentée par Flutter, présente 2 limitations :

- La quantité de code boilerplate à écrire est importante
- Le typage entre les données passées d'un côté du PlatformChannel et celles récupérées de l'autre côté n'est pas possible. Une fois qu'une donnée est envoyée dans le PlatformChannel, il faut en vérifier le type au runtime de l'autre côté à chaque fois.

Pour pallier ces limitations, Flutter propose le package Pigeon. Cette solution de génération de code permet de créer automatiquement l'ensemble des briques constitutives d'un PlatformChannel, en assurant le typage des données transférées. Cette approche permet donc de gagner un temps considérable sur la génération du code, nécessaire au fonctionnement du PlatformChannel. En parallèle, elle vient garantir le typage des données échangées de part et d'autre, prévenant ainsi de nombreux bugs.

On note cependant quelques limitations telles que l'absence de support des events channels ou de la gestion des enums. La librairie n'est pas encore compatible avec des plateformes desktop.

En dépit de ces quelques limitations, nous recommandons fortement l'usage de Pigeon. Notre recommandation s'appuie notamment sur son appartenance aux packages officiels Flutter ainsi que sur son dynamisme, comme en témoigne le grand nombre d'évolutions sur les derniers mois.



20 BLoC

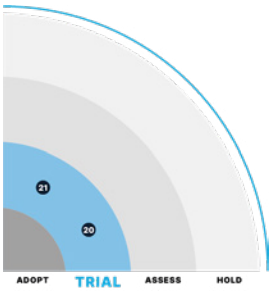
Comme tout framework réactif, Flutter repose sur l'existence d'un State dont la mise à jour déclenche une modification de l'UI. Depuis sa création, les solutions de State Management dédiées à Flutter se multiplient.

BLoC est la solution développée par Félix Angelov pour le State Management des applications Flutter. Créée fin 2018, elle apparaît aujourd'hui, comme beaucoup d'autres, dans la liste des solutions de State Management recommandées pour Flutter, mais elle s'est également installée de manière pérenne dans la communauté avec : plus de 400 releases, une 8e version sortie cette année et des sponsors très impliqués dans la communauté Flutter (ex : Very Good Ventures). Nous avons eu l'occasion de tester BLoC sur plusieurs projets en production chez BAM.

BLoC présente une approche structurée et prédictible avec des objets aux responsabilités clairement définies (ex : Event, State et Bloc) et elle se combine très bien avec le package *Freezed* pour limiter l'écriture manuelle de code boilerplate. Par ailleurs, elle propose une très bonne intégration dans les éditeurs de code avec des snippets permettant d'instancier les widgets un clin d'œil.

Sa documentation très complète et alimentée par de nombreux exemples, ainsi que sa communauté active, présentent deux de ses atouts majeurs. Enfin, l'aisance avec laquelle il est possible de tester et de mocker le code écrit en utilisant BloC a permis à nos équipes d'atteindre et de maintenir une couverture de test de 100% sur nos derniers projets.

Bien que nous soyons convaincus par cette solution et que nous la recommandons pour gérer le State d'une application Flutter, nous maintenons son statut en Trial car d'autres solutions telle que Riverpod (également présente dans ce Tech Radar) sont également recommandables. A ce jour, nous n'avons pas identifié de grand vainqueur sur le terrain du State Management associé à Flutter.



21 Riverprod

Au même titre que BLoC et Provider, Riverprod est une solution de State Management dédiée à Flutter. Elle apparaît en 2ème position dans la documentation "List of state management" de Flutter, juste derrière son prédécesseur Provider et avant les solutions « Native » de Flutter (ex : setState et InheritedWidget).

Cet excellent positionnement est un premier indicateur de la qualité et de la stabilité de la solution. Riverpod a été créée pour répondre à 2 limitations de Provider :

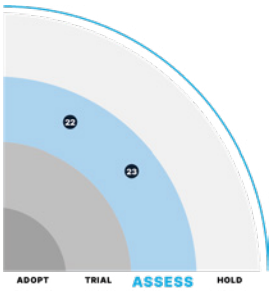
- Être indépendant de Flutter
- Être compile safe

Par ailleurs, elle présente de nombreuses qualités telles que sa syntaxe minimaliste et sa capacité à être facilement mockable et testable.

Enfin, Riverpod se rapproche aujourd'hui de la solution homologue react : react-query en proposant une excellente gestion de l'asynchrone avec des états de loading, error, data et refreshing présents out of the box, qui permettent d'afficher l'UI correspondant à chaque état de chargement d'une donnée.

Si cette solution nous a convaincus après l'avoir utilisée sur plusieurs projets en production, et que nous la recommandons pour gérer le state d'une application Flutter, nous avons décidé de maintenir son statut en trial.

En effet, d'autres solutions telles que BLoC (présente dans ce Tech Radar) sont également recommandables. A date, nous n'avons pas identifié de grand vainqueur sur le terrain du « State Management » associé à Flutter.



22 Alchemist

En Flutter, les tests d'éléments visuels (widgets, pages) s'appuient sur le système de *golden tests*, qui consiste à générer un rendu du widget testé au format image.

Lorsque les tests sont lancés à nouveau, la nouvelle image est comparée à la version précédente. Si une différence de pixels est détectée, le test échoue et met en garde contre les régressions. Il est possible de configurer la tolérance d'un pourcentage d'erreurs, cependant, dans la réalité, la donnée varie fortement en fonction du design et de l'écran ciblé. Il est donc difficile de déterminer un seuil de tolérance pertinent et commun à tous les types de designs.

Dans la vie d'un projet, les suites de tests sont lancées par des développeurs sur leurs machines locales, qui opèrent sous Mac OS (contrainte des développeurs mobiles), mais également sur les machines d'intégration continue, que nous configurons pour opérer sous Linux (les machines d'intégration continue sous MacOS sont ~ 5 à 10 fois plus onéreuses).

Cette différence d'OS entre les machines de développement et les machines d'intégration continue peut paraître anodine, mais en cas de *golden tests* sur des éléments textuels, elle devient problématique.

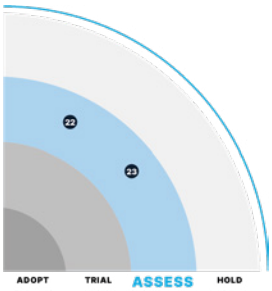
En effet, les *golden tests* affichant du texte apparaissent différents sur les plateformes MacOS et Linux, causant des faux négatifs dans les tests lancés en CI.

La solution Alchemist propose un ensemble d'outils permettant de faciliter l'utilisation des *golden tests* en Flutter. L'une de ses fonctionnalités phares vise à résoudre le problème évoqué ci-dessus, en générant des images goldens différentes, pour les tests en local et ceux de la CI.

Par ailleurs, Alchemist permet de générer plusieurs goldens au sein d'une même page, pour plus de concision.

Enfin, en attribuant un *label golden* aux tests goldens, elle permet de lancer indépendamment et parallèlement les tests goldens et les tests classiques afin d'optimiser la vitesse d'exécution de la suite de tests.

Nous avons testé cette librairie chez BAM et les résultats étaient encourageants. Néanmoins son adoption reste encore marginale avec moins de 200 stars sur Github et une popularité < 80% sur pub.dev. Nous vous recommandons donc de surveiller ce projet pour voir s'il devient plus adopté dans la communauté.



23 Melos

Certaines architectures projets se basent sur la notion de multi-packages. En d'autres termes, un seul répertoire va contenir plusieurs packages, correspondant chacun à une brique indépendamment développable, testable et déployable.

Ces packages sont souvent interdépendants et il devient difficile pour le package manager (quelle que soit la techno: npm, pub, composer, pip, gem ...) de résoudre les dépendances de chaque package. Et ce, à la fois dans un environnement de production et dans un environnement local de développement.

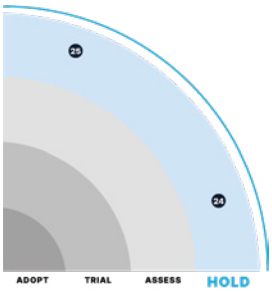
Les dépendances liées à un environnement de production doivent être résolues depuis le site du provider (pub.dev, npmjs.com ...). Côté développement, le point de départ est le code source local de chaque package.

En Flutter, et plus généralement en Dart, ce problème est résolu par Melos, qui propose un outil en ligne de commande pour ces «mono-repos». Il résout les dépendances avec un linking local, permet le lancement de commandes sur l'ensemble des packages (analyze, test...) et propose automatiquement le versioning et le déploiement de chaque package pour s'intégrer facilement sur une CI.

Après avoir utilisé Melos plusieurs mois sur un projet de SDK Flutter, nous avons rencontré des difficultés pour débogger certaines erreurs de CI causées par cet outil, et perdu du temps parfois précieux.

Par ailleurs, la documentation indique que le package est en développement actif et non pas stabilisé, bien qu'utilisé par des grands noms tels que FlutterFire.

Si nous vous invitons à garder un œil sur Melos qui, porté par l'entreprise Invertase (fer de lance de l'open source Flutter), promet de devenir la référence de son domaine, nous le positionnons aujourd'hui dans la catégorie Assess. Au regard des éléments évoqués, cette solution n'a pas atteint une maturité suffisante pour être adoptée les yeux fermés.



24 GraphQL with Flutter

La capacité à récupérer les données exposées par un serveur, via une API, constitue une fonctionnalité clé pour toute application mobile.

Depuis quelques années, le standard GraphQL s'impose progressivement comme une alternative à REST, au sein de la communauté. Aujourd'hui, les applications développées en Flutter peuvent également échanger leurs données avec une API GraphQL, notamment grâce à 3 solutions disponibles : graphql_flutter, ferry et artemis.

Parmi ces 3 solutions, graphql_flutter est la plus populaire (avec 3k stars github à ce jour). Cette approche, très inspirée d'Apollo (référence des clients GraphQL de l'écosystème React) est très complète et présente de nombreux atouts.

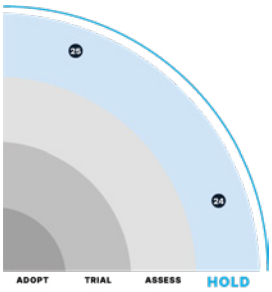
Parmi eux, on note la gestion du hors ligne avec l'intégration d'un cache, d'un système de polling/rediffusion et le support des résultats optimistes.

On observe cependant quelques limitations relatives au support des fichiers .graphql et à la génération de code, dont la documentation reste légère et qui nécessite l'utilisation des hooks (un pattern qui n'est pas arrivé à maturité en Flutter).

La solution ferry, plus récente, propose une autre approche en insistant sur une intégration solide de la génération de code, qui permet un typage fort des objets manipulés. Utilisée par nos équipes sur l'un de nos derniers projets et aujourd'hui en production, cette solution a donné lieu à un retour d'expérience mitigé. Pour cause, un goulot d'étranglement au niveau de la maintenance (absence de réponses sur plusieurs issues créées par nos équipes et 85% des commits réalisés par le créateur).

N'ayant pas encore eu l'occasion de tester la solution Artemis sur un projet, il nous semble prématuré de vous proposer une évaluation quelconque à ce stade. Par ailleurs, nos expériences récentes avec les différents clients GraphQL de l'écosystème Flutter ne nous permettent pas de faire ressortir nettement une solution solide à recommander.

Nous restons pour l'instant sceptiques sur la maturité de cette approche.



25 Provider

Au même titre que BLoC, Provider est l'une des solutions de State Management les plus populaires.

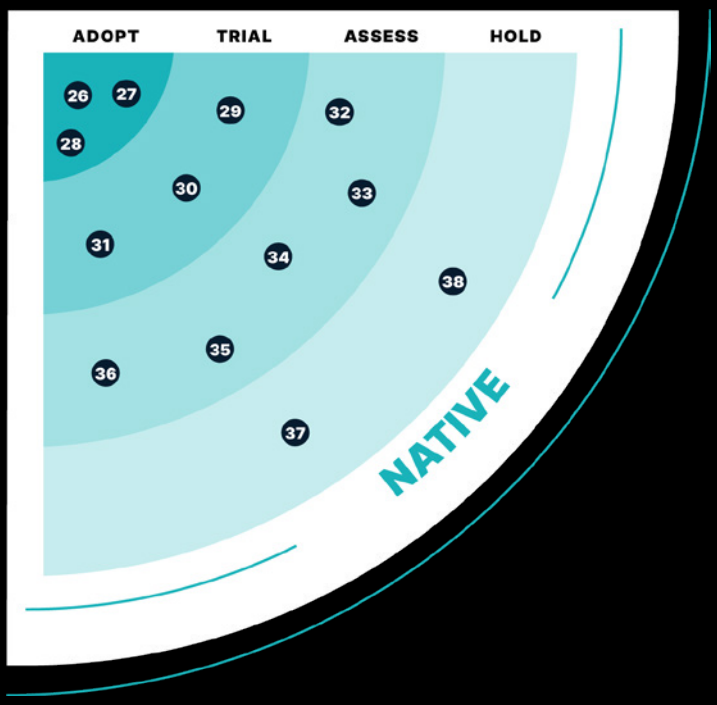
Provider a été conçu très tôt, par Rémi Rousselet, comme un wrapper autour de la solution native de Flutter: InheritedWidget, pour le rendre plus simple d'utilisation et plus scalable avec une API Provider/Consumer simple à appréhender. Cette solution constitue aujourd'hui l'approche standard pour le State Management et l'injection de dépendances des apps Flutter, comme en témoigne le label *Flutter Favorite* qui lui est attribué.

Si Provider apparaît toujours comme une approche stable et solide pour gérer le State d'une application Flutter, cette solution présente certaines limites telles que le chargement asynchrone des providers, le support de plusieurs providers de même type ou encore les exceptions au runtime.

L'émergence de solutions plus modernes telles que BLoC ou encore de Riverpod, également publiée par l'auteur de Provider, et construite essentiellement pour résoudre les défauts de Provider, en font aujourd'hui une solution de second plan.

En d'autres termes, si vous souhaitez utiliser Provider, notre discours pourrait être simplifié en une phrase : " Utilisez Riverpod, c'est pareil, mais en mieux "

08. Technologies Natives



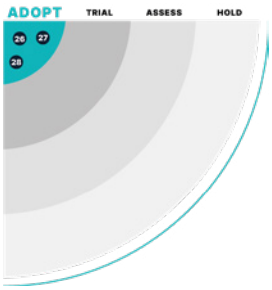
13 BLIPS

3 ADOPT
3 TRIAL
5 ASSESS
2 HOLD

À retenir

Côté Native, nous nous sommes concentrés sur l'ensemble des technologies expérimentées sur nos projets en production. Vous y retrouverez nos recommandations, basées sur l'expertise de la tribe native chez BAM, qui réunit des experts iOS et Androïd. Nous relevons trois pans principaux : le pan visuel/layout, le pan relatif au state et le pan relatif à l'architecture (ainsi que les outils de DevX associés) qui est encore plus qu'en React Native un élément différenciant d'une bonne application. ”

Tycho / **Principal Technologist**



26 Jetpack Compose

Les frameworks UI font souvent l'objet de débats : templates externes ou implémentation programmatique, outil graphique pour la conception ou écriture à la main ? La technologie de layouting visuel historique d'Android n'en fait pas l'exception. Elle se caractérise par un cycle de vie délicat à maîtriser et/ou par une lourdeur pour créer de nouveaux composants graphiques.

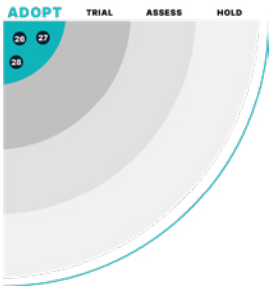
De plus, les APIs graphiques ayant été conçues au milieu des années 2000, elles tenaient compte de contraintes obsolètes pour les smartphones d'aujourd'hui et limitaient l'évolution d'Android.

Les équipes de Google sont donc reparties d'une feuille blanche pour créer Jetpack Compose, en adoptant une API déclarative et réactive, inspirée par le framework React. Le code est plus lisible, avec une structure du code proche de ce qui est affiché, et plus concis : la complexité de la mise à jour de l'état de l'UI est masquée.

Jetpack Compose est une technologie complexe : l'intégration s'appuie sur un plugin du compilateur Kotlin, un moteur de calcul de différences et de nombreuses bibliothèques de composants. En effet, l'intégralité des composants du design system Material a été ré-implémenté pour Compose.

La maturité du framework est impressionnante. Les cas d'usage courants sont documentés, facilement implémentables et une solution peut facilement être implémentée pour les cas limites. Il peut toutefois manquer certaines fonctionnalités par rapport aux composants historiques, à titre d'exemple sur les animations ou la navigation. On note également l'engouement au sein de la communauté des développeurs. Des articles et des bibliothèques complémentaires sont régulièrement écrits par la communauté.

Jetpack Compose est aujourd'hui un choix sûr pour développer une nouvelle application sur Android. Attention en revanche à la migration d'une app existante : si Jetpack Compose s'intègre très bien avec des sources de données réactives (ex : un ViewModel avec LiveData, RxJava ou Flow), l'intégration sur une base de code monolithique et impérative pourrait nécessiter d'importantes modifications.



27 Koin

Les principes SOLID, notamment le principe d'inversion des dépendances, ne sont pas toujours appliqués rigoureusement dans le développement mobile.

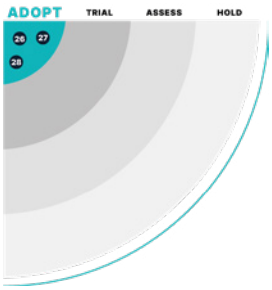
Nous avons rencontré plusieurs projets existants qui ne les appliquaient pas. Pour de nombreux stakeholders, les applications mobiles sont considérées à tort comme plus petites. Ainsi, les pratiques de programmation tendent à privilégier la vitesse d'écriture à la robustesse du code.

Cette situation est parfois accentuée par un manque de pratiques et outils permettant d'appliquer ces principes facilement.

Koin est l'une des solutions qui a émergé en réponse à ces problématiques : ce framework dispose d'un DSL très intuitif, qui permet de maîtriser les configurations nécessaires pour faire une DI compatible avec les concepts de base sous Android en quelques minutes.

Cette solution est par ailleurs plus flexible et facile à prendre en main que son principal concurrent, Dagger-Hilt, qui fait partie de Jetpack.

Malgré sa verbosité et son léger impact sur la performance, Koin est devenue notre librairie de DI en Kotlin par défaut.



28 SwiftUI

Avant l'arrivée de SwiftUI, il existait deux alternatives pour faire de la UI en iOS :

- Utiliser UIKit, le framework par défaut, créé par Apple. Néanmoins, celui-ci ne propose qu'une API impérative et peu lisible.
- Utiliser des frameworks tierces déclaratives, comme Texture. Ces bibliothèques sont néanmoins peu maintenues, et marquées par un manque de documentation.

En 2020, Apple a publié SwiftUI, un nouveau framework déclaratif pour suivre la tendance de Jetpack Compose, React-Native et Flutter, appelé SwiftUI. Ce framework est également requis pour certaines nouveautés, comme les widgets.

Il présente de nombreux avantages : d'une part, la syntaxe est claire, simple et très documentée ce qui facilite la montée en compétence, d'autre part, l'écosystème est très complet, avec de nombreuses librairies qui s'intègrent avec SwiftUI.

Dans le cas où il est impossible de coder une certaine partie de l'app en SwiftUI, ce framework est facilement interopérable avec UIKit. Enfin il s'interface très bien avec Combine, le framework de programmation reactive d'Apple.

En revanche, ce framework présente encore quelques lacunes, comme l'absence de rétrocompatibilité sur les anciens OS. A titre d'exemple, on ne peut pas utiliser SwiftUI v2 si nos utilisateurs disposent de la version iOS 13. Quelques APIs ne sont pas encore accessibles sous SwiftUI et il faut faire un bridge vers UIKit pour les utiliser. Certains bugs du framework ont été corrigés dans les récentes versions, mais certains bugs et problèmes DX persistent, comme les previews, qui rendent l'expérience imprévisible.

SwiftUI est donc un framework que nous recommandons, mais avec certaines réserves puisqu'il est dépendant de la version d'iOS supportée par votre app et qu'il requiert, dans certaines cas, de coder sa solution en UIKit et d'effectuer un bridge.

29 Resolver

Pour rendre un code maintenable et testable, l'injection de dépendance permet de déterminer de manière dynamique et configurable quelle implémentation concrète sera utilisée à l'exécution.

Parmi les différentes librairies permettant l'injection de dépendances en iOS, nous vous conseillons l'utilisation de Resolver. Avec l'utilisation de Property Wrapper dans vos fichiers Swift, cette librairie vous permet d'injecter des dépendances au runtime et est disponible à partir de la version Swift 5.1. Resolver ne génère pas de fichiers supplémentaires à la compilation, à la différence par exemple de Swinject, ce qui rend le code plus lisible et permet d'utiliser des weak var pour ne pas introduire de RetainCycle. De plus, elle offre une très bonne testabilité. Enfin, elle est plus performante que ses concurrents. A titre d'exemple, elle est 800% plus rapide à résoudre les dépendances que Swinject.

Néanmoins, cette solution est encore relativement récente et manque d'adhérence dans la communauté. A ce stade, elle ne compte que 1,8k stars vs. Swinject qui en compte 5,3k. En pratique, nous l'utilisons cependant avec succès sur des projets en production.

30 Tuist

Les outils de développement iOS sont vétustes : pour le lancement d'un nouveau projet, le format xcodeproj peut facilement atteindre jusqu'à 3000 lignes ou plus, rendant son contenu facilement incompréhensible. L'impossibilité de l'éditer à la main empêche également sa review. Et à plusieurs développeurs, les conflits git sont inévitables.

Tuist propose de ne plus versionner ce projet xcodeproj, mais de versionner une configuration en Swift qui le génère à la volée.

À titre d'exemple, sur l'un de nos projets, nous passons de 1788 lignes de code à 19 lignes. Par conséquent, le fichier est bien plus lisible, et les conflits disparaissent.

Tuist présente plusieurs avantages :

- Premièrement, il aide à la modularisation. Pour un projet en architecture micro framework, il automatise la création de liens entre les projets dans un même espace de travail, génère un graphe de dépendances pour documenter le projet et propose un cache afin d'optimiser la compilation.
- Ensuite, il accélère le développement : en une commande qui établit l'architecture de fichier, on génère le squelette d'une page ou d'un appel API.
- Tuist permet également de générer un accès typé aux assets (images, vidéos, sons, etc.) afin d'éviter un crash dû à une erreur de nom de fichier.

Chez BAM, nous adoptons Tuist sur tous nos nouveaux projets. L'API est devenue stable et la communauté, très active, est toujours prête à apporter son aide en cas de problème. Seul bémol, la documentation est devenue moins conséquente avec le passage de la version 2.x à la version 3.x. bien que ça ne soit pas bloquant (code source clair et communauté active). Néanmoins, Tuist vaut le détour !

31 Viper

En termes d'architecture, l'écosystème iOS est très fragmenté : il n'existe à ce stade aucun standard de référence clairement défini.

Pour cause, la documentation iOS prône souvent une architecture très simple, qui donne beaucoup de responsabilités au ViewController et n'est pas adaptée à une application plus conséquente (exemple).

Avez-vous déjà vu des ViewController de 1000+ lignes ? Nous, malheureusement, oui.

VIPER est une architecture qui permet de découper la partie interface utilisateur d'une application iOS en différentes couches de responsabilités. À la différence de "Clean architecture", elle ne fournit pas un flux quant à l'implémentation des données et de la logique métier.

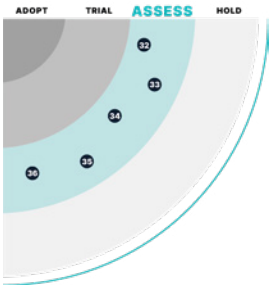
Dans VIPER, on retrouve 5 éléments architecturaux :

- La vue (V) qui gère l'interface,
- Le présenteur (P) qui relie les données à la vue
- Les données, qui viennent de l'entité (E)
- Et sont filtrées par l'intéracteur (I)
- Enfin, le routeur (R) qui s'occupe de la navigation

Ce modèle peut sembler complexe, mais en pratique, il présente 3 avantages :

- Les responsabilités sont clairement découpées ce qui améliore la maintenance
- Chaque fichier est de taille moyenne : plus de UIViewController interminable
- La gestion de la mémoire est claire : le système maintient une référence forte à la vue, la vue au présenteur, et le présenteur au routeur et à l'intéracteur. Il n'y a plus aucune fuite de mémoire par cycles de retenue.

Pour votre prochain projet, nous vous invitons à passer au dessus de la complexité associée à VIPER et de tester son architecture. Le jeu en vaut la chandelle.



32 App Code

La communauté iOS peine à trouver un éditeur de code pouvant répondre correctement à tous ses besoins.

Solution entièrement dédiée à l'écosystème, Xcode répond à des besoins de base du développement, simplifie le débugging et offre un système de build très complet.

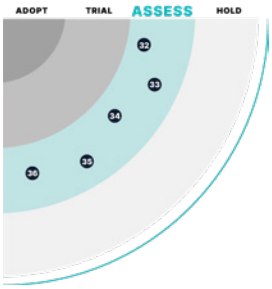
Néanmoins, son interface se révèle assez particulière en termes d'organisation et de visuel, à la différence des autres éditeurs. Cela rend la transition difficile pour des développeurs d'autres technologies.

Xcode offre également une auto-complétion et un linting temps réel très lent et, à titre d'exemple, il ne permet pas d'ajouter des plugins de type vim.

C'est pourquoi nous utilisons AppCode sur nos projets, en complément de Xcode.

AppCode est un éditeur de code développé par JetBrains, qui nous offre une expérience de développeur supérieure grâce à une interface mieux pensée et plus proche des autres éditeurs, une auto-completion et un linting plus performant, ainsi qu'un support poussé, notamment pour le refactoring.

Néanmoins, étant donné qu'Apple verrouille historiquement son écosystème, un risque sur la pérennité d'AppCode persistera tant qu'Apple n'aura pas montré une forme d'approbation sur ce genre d'outil.



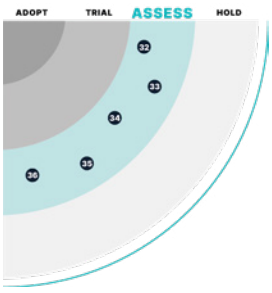
33 Composable Architecture

Le framework SwiftUI a apporté des nouveautés sur la UI, mais également sur la gestion du state de par son approche déclarative à la différence de UIKit. SwiftUI fournit des ObservableObjet et des EnvironmentObject pour externaliser le state. Ces primitives de code sont très puissantes, mais ne permettent pas de garantir une qualité d'architecture en tant que tel.

La Composable Architecture, très inspirée du *one-way data flow* popularisé par Redux en 2017, permet de structurer le state global mais aussi de le rendre testable et facilement réutilisable. De plus, la developer experience est améliorée car il est plus facile de savoir quel événement provoque un changement de state donné.

En revanche, cela rend le code parfois verbeux en particulier quand il s'agit d'utiliser des types non *Equatable* dans le state. La bibliothèque est récente : bien qu'elle soit très bien documentée, son intégration à d'autres bibliothèques de la communauté n'est pas optimale. La surface d'API est grande : à titre d'exemple, OptionalPath permet de gérer un state nullable très pratique, mais est difficile à trouver dans la documentation.

Chez BAM, nous commençons à utiliser Composable Architecture sur certains projets en production et nous vous recommandons de regarder cette bibliothèque pour la gestion du state sur vos prochaines apps en SwiftUI.



34 µ-Features Architecture

Il y a quinze ans Adrian Cockcroft, architecte chez Netflix, introduisait la notion d'architecture microservice. Le but : minimiser la friction des équipes « serveur ». Au programme : une séparation des responsabilités, un stockage optimisé et une plus grande agilité du côté des équipes de développement.

Aujourd'hui, les applications mobiles rappellent de plus en plus les applications serveurs par leur stockage relationnel en local, la complexité croissante de leur features ou encore les interconnexions avec de multiples services.

Les problèmes de développement relatifs aux applications mobiles et serveurs sont également similaires : base de code grandissante, temps de build croissant, difficulté à respecter la pyramide des tests.

C'est ainsi que des ingénieurs de plusieurs entreprises, telles que Soundcloud et JustEat ont popularisé l'approche de micro-features (ou micro-applications) en remplaçant un monolithe par de plus petits modules de différents types :

- L'application principale, qui réunit et coordonne les flux UXs
- Les flux UXs qui gèrent un pan visuel de l'application (par exemple en utilisant une architecture VIPER), c'est-à-dire à la fois les composants, ainsi que la navigation entre les pages. Les flux UXs délèguent la logique aux modules de logique métier
- Les modules de logique métier qui gèrent un ensemble de services et d'entités et sont dédiés à un même domaine d'application. Ces modules utilisent, si nécessaire, un module cœur
- Les modules cœur qui servent d'interface vers des fonctionnalités externes (comme un appel API, du stockage, des notifications, etc.) ou de l'outillage (module de journalisation, de déboggage, etc.)

Les avantages sont multiples. Si les modules sont bien découpés, il est possible de développer sur un périmètre plus petit. Cela implique des logiques métiers plus facilement testables ainsi qu'un temps de build diminué grâce au cache.

Par ailleurs, si l'équipe s'agrandit ou que l'entreprise développe d'autres produits, il sera possible de mutualiser du code commun entre les différents projets (par exemple, les modules d'authentification).

Mais un bon découpage des modules implique une bonne expertise du domaine métier. A titre d'exemple, il faut particulièrement travailler l'interface pour augmenter la cohérence et diminuer le couplage de chaque module avec les autres. Cela demande également une connaissance de l'architecture ainsi qu' une bonne conception, en plus d'un investissement initial.

Et comme pour les micro-services, un outillage spécifique devient nécessaire pour éviter que le rêve ne tourne en cauchemar. Il suffit que l'architecture soit mal implémentée, ou l'équipe non formée, pour que l'architecture ralentisse le projet au lieu de l'accélérer.

Un avantage du développement mobile est le packaging des micro-features en un seul binaire, contrairement aux micro-services en web. Par ailleurs l'outillage devient de plus en plus stable : par exemple, Tuist aide sur iOS.

Chez BAM, les micro-features sont devenues un choix de référence dans les nouveaux projets iOS.

35 Using relational db as state management

Côté serveur, l'industrie utilise des bases de données relationnelles (comme Postgres, MySQL) depuis des décennies pour stocker les données.

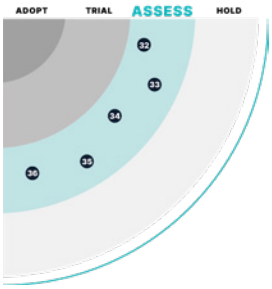
Cela présente de nombreux avantages : format de données structuré et prévisible, normalisation des données, accès simplifié, propriétés avancées de cohérence (ACID), gestion des transactions, etc...

Pourtant, côté client mobile, on observe un retard sur ce sujet. De nombreux projets sont initiés avec une approche non structurée (document dans Redux) et finissent, au fil des bugs, par réimplémenter les contraintes de structure, la dénormalisation et les transactions, ce qui conduit à une forte complexité logicielle.

Nous avons suivi de près la réécriture de l'application Messenger sur iOS par Facebook (projet lightspeed). Au cœur de *Lightspeed*, on retrouve l'utilisation de la base de données relationnelle SQLite : cette dernière s'occupe du stockage, des filtres et des transactions. Ainsi, la complexité logicielle est réduite.

Nous observons également l'émergence de bibliothèques haut niveau, basées sur SQLite, comme Room (Android) ou Watermelon (React Native). Cela implique de former les équipes sur les fondamentaux d'une base de données.

Nous pensons que le type de stockage de données est une question primordiale à se poser et nous vous invitons à considérer l'utilisation d'une base de données relationnelle pour vos applications mobiles.



36 Texture

La mise en forme de l'interface sur iOS n'est pas une tâche facile : entre les amateurs des contraintes écrites sous forme de code et les profils qui privilégient les constructeurs d'interface visuelle, aucune méthode n'a fait l'unanimité jusqu'à présent.

React Native a cependant rapidement bouleversé le status quo en initiant le découpage en composants autonomes et l'utilisation de flexbox. L'approche implémentée plus tard pour l'agencement et en inspirant SwiftUI et JetpackCompose.

Mais comment profiter de ces bénéfices sur une application iOS compatible ? Aujourd'hui, un grand nombre de projets supportent un parc de téléphones non compatibles avec SwiftUI ou ont investi trop de temps dans un projet en UIKit, ce qui explique leur réticence à changer drastiquement d'approche.

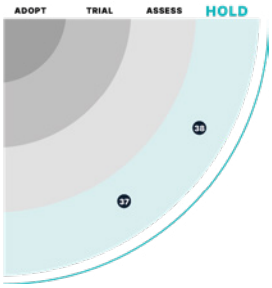
Anciennement appelé AsyncDisplayKit, Texture est un framework créé par Facebook et dont le développement est aujourd'hui assuré par Pinterest.

Il permet de définir des composants visuels et de les agencer avec des flexboxs. Le framework est très performant et succinct par rapport à UIKit, et il est facile de le connecter avec des composants UIKit existants. De plus, il existe un écosystème pour le relier à des librairies d'observable comme RxSwift.

La librairie pourrait être mieux documentée, mais reste par exemple en avance sur Litho. Aussi, bien que Texture présente peu d'activités à ce stade, elle dispose d'une communauté Slack assez active pour dépanner.

Texture a déjà fait ses preuves sur de grosses applications telles que Pinterest, New York Times, NFL ou Auxy et nous l'avons utilisé avec succès chez BAM.

Si les contraintes liées à SwiftUI vous empêchent d'utiliser ce framework en ce moment, Texture peut constituer une bonne solution de transition.



37 KMM

Les plateformes mobiles iOS et Android ont toujours été incompatibles entre elles :

Elles n'utilisent ni le même langage (Kotlin et Swift), ni le même environnement d'exécution (Machine Virtuelle vs exécution native sur iOS) ni les mêmes APIs (frameworks propre à chaque OS).

Bien qu'il ait toujours été techniquement possible de partager du code entre les 2 plateformes via les Foreign Function Interface (FFI) des langages, ces dernières ne sont utilisées que pour le code de certaines bibliothèques (ex : SQLite est utilisable sur iOS et Android) et très rarement pour le code applicatif.

En effet le code métier d'une application expose souvent un modèle de données complexes et de nombreuses fonctions qu'il est très fastidieux d'interfacer via les FFIs.

La promesse de Kotlin Multiplatform Mobile est de pouvoir écrire le code métier et les couches de données (ex : la sérialisation JSON ou les DTOs d'une base de données locale) en Kotlin et d'exposer ces implémentations via une bibliothèque consommable par chaque plateforme (un .aar pour Android et un .framework pour iOS).

Le framework assure la traduction des appels et des structures de données de Swift vers Kotlin et vice-versa. On bâtit ainsi une sorte de SDK propre à l'application, sur lequel il ne reste qu'à implémenter le code propre à chaque plateforme (en particulier l'UI).

Nous avons déjà intégré cette technologie sur plusieurs projets en y consacrant des semaines d'évaluation. À chaque fois, nous avons décidé de ne pas poursuivre l'intégration au-delà de l'évaluation. En effet, l'utilisation de KMM présente plusieurs inconvénients :

- Une augmentation du temps de build sur l'app iOS
- Une dégradation du modèle pour iOS (ex: KMM n'expose pas d'enum avec des associated values, feature idiomatique et populaire de Swift)
- Des limitations dans la modularisation de l'application
- Un écosystème assez peu dynamique (peu de nouvelles bibliothèques, des problèmes présents depuis plusieurs années sans être réellement résolus)

On retient plusieurs freins pour faire évoluer le projet au-delà des templates de base :

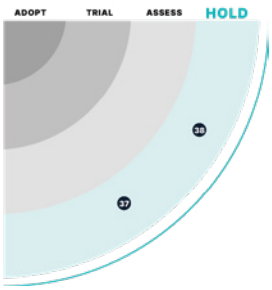
- Un ajout de dépendances qui n'est pas toujours simple
- La maintenance des fichiers Gradle est problématique : dans les documentations et les tutoriels, plusieurs formats et conventions coexistent sans réel consensus
- Des erreurs parfois cryptiques, en particulier lors de l'intégration sur iOS

Notons enfin qu'il peut exister dans certaines équipes, une barrière psychologique à l'intégration de code en Kotlin dans une application iOS.

Les entreprises qui rapportent utiliser cette technologie avec succès, telles que CashApp aux États-Unis et Deezer en France, disposent d'équipes techniques très expérimentées et d'un produit dont le domaine est complexe.

Pour des projets de plus petite envergure et menés par des équipes moins stables, ces complications n'en valent peut-être pas la peine et elles risquent, à terme, d'impacter la maintenabilité.

En raison de ces différents compromis et limitations, nous ne recommandons pas l'adoption massive de cette technologie sur les projets. Kotlin Multiplatform peut cependant présenter un réel intérêt, si le projet s'y prête et à condition que les équipes techniques soient expérimentées et acceptent certains compromis.



38 Litho

Le développement d'interface sur Android n'est pas chose facile. L'utilisation de XML Layout n'est peut-être pas la plus intuitive.

Les approches du type Flexbox de React Native nous offrent la possibilité de construire une UI en utilisant et en agençant facilement des composants.

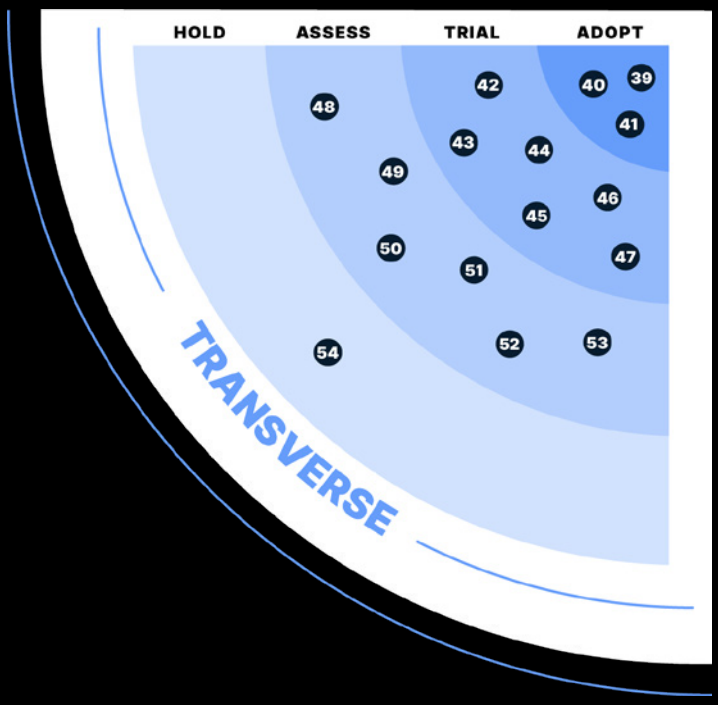
En 2017, JetpackCompose n'existait pas : pour accélérer le développement d'interface, Facebook a développé Litho, un framework déclaratif qui utilise une approche composants.

Ce framework a l'avantage d'utiliser Yoga, le moteur d'agencement de Facebook, et donc d'utiliser les flexbox. Via le principe de ViewFlatening, Litho optimise la performance dans le rendu des composants. Litho constitue une belle promesse pour faciliter la mise en place d'interface visuelle.

Néanmoins, nous ne conseillons pas d'utiliser cette interface sur vos projets, car elle dispose à ce stade d'une documentation lacunaire et d'une communauté très réduite. De plus, elle n'est pas optimisée pour l'utilisation avec Kotlin, ce qui ralentit les développements.

Facebook a cependant indiqué vouloir changer cet état de fait et travaille actuellement sur une nouvelle version de la documentation. JetpackCompose s'impose désormais pour gérer les interfaces sur vos projets Android.

09. Transverse



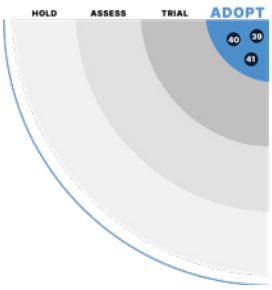
16 BLIPS

3 ADOPT
6 TRIAL
6 ASSESS
1 HOLD

À retenir

Cette catégorie est pour nous une façon de partager nos expériences sur la méthode et les approches au niveau d'architecture. Certaines des problématiques peuvent bien se décliner dans les 3 technologies du radar, d'autres sont plus ou moins complexes en fonction de la maturité de l'écosystème. Nous considérons que le thème majeur de l'écosystème aujourd'hui est la question d'architecture. Les applications deviennent de plus en plus complexes, elles évoluent d'une manière constante et une bonne architecture est clé pour maintenir le rythme d'innovation. Entre l'architecture Lightspeed de Facebook, les discussions autour de Clean Architecture mobile ou une nouvelle popularité de programmation fonctionnelle, cet aspect de programmation mobile est très riche en nouveautés et nous vous invitons à le surveiller de très près."

Marek / **CTO & Co-founder**



39 ADR

Très souvent, les développeurs et architectes sont amenés à faire des choix sur les projets : langages, technologies, bibliothèques externes, architectures...

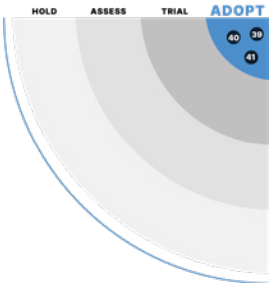
Nous avons remarqué 3 problématiques associées à ces choix :

- Les choix ne font pas nécessairement consensus au sein de l'équipe, car ils sont souvent portés par une même personne
- Même si le choix est consensuel, ce n'est pas forcément le meilleur
- Le choix n'étant pas documenté, les équipes qui reprendront le code ne le comprendront pas

Un *Architecture Decision Record* (ADR) est un document qui vient adresser ces trois problèmes. Il est fait en équipe et stocké avec la documentation. **Il vise d'abord à expliciter le problème et le contexte, puis à lister toutes les solutions possibles.** Enfin, il vise à définir des critères discriminants et à attribuer une note à chaque solution, par rapport à chaque critère. Le choix retenu est la solution qui obtient la meilleure note totale.

Nous avons généralisé l'usage d'ADR dans nos équipes pour justifier tous les choix qui sortent de l'ordinaire et nous avons déjà quelques success stories : par exemple, nous avons poussé *django* server side dans un projet, qui a permis de réduire de moitié la charge de travail.

En pratique, le temps passé à faire l'ADR est largement rentabilisé et les équipes montent en compétences grâce à l'outil. Nous vous invitons à l'adopter également.



40 BPMN

Plusieurs implémentations de Scrum sont tombées dans le même travers, en faisant un focus trop important sur les *user stories*, et en faisant passer au second plan la notion d'une fonctionnalité. Par conséquent, une équipe peut régulièrement être amenée à enchaîner des *user stories*, sans avoir une vision globale. Les résultats nuisent beaucoup à la productivité :

- Cela se traduit par l'oubli de cas limites, qui va conduire à des bugs et/ou à des fonctionnalités incomplètes
- Les macro estimations qui en résultent sont peu fiables, car il y a beaucoup de retours d'un sprint à l'autre

Le cœur du développement de la fonctionnalité est réalisé au cours d'un sprint, mais la gestion des cas limites ou de certains sous-parcours peut nécessiter plusieurs sprints. Cela est dû à un manque de vision commune et d'alignement entre le produit, le design et la tech. En effet, les POs manquent souvent de compétences tech pour spécifier l'epic et trouver les cas limites. Chez BAM, nous avons introduit depuis quelques années un atelier d'une heure, au lancement d'une epic. Cet atelier réunit toutes les parties prenantes pour dessiner un schéma technico-fonctionnel. Avec un langage visuel inspiré de BPMN, nous dessinons tous les comportements utilisateurs, ainsi que leurs impacts techniques. Ce document constitue ensuite une base de départ pour écrire les user stories.

Un schéma partagé entre toutes les équipes permet à la fois de :

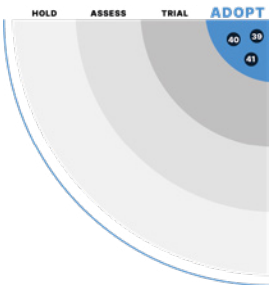
- Se mettre d'accord sur le scope exact de la fonctionnalité et d'améliorer la compréhension commune
- Faciliter la levée d'alerte en mettant en avant des zones d'ombre
- Visualiser la complexité fonctionnelle et technique
- Permettre une meilleure QA et faciliter le débogage

Les bénéfices sont flagrants :

- On constate moins de bugs
- Le lead time nécessaire pour sortir une fonctionnalité diminue
- On remarque un meilleur alignement entre les équipes
- Une meilleure fiabilité des estimations

À titre d'exemple, sur l'un de nos projets, cet atelier nous a permis de détecter une mécompréhension fonctionnelle qui aurait coûté 5 semaines de retard à l'équipe, si elle avait été détectée au cours de l'implémentation.

Nous utilisons cette approche pour toute fonctionnalité complexe et nous vous invitons à la tester sur tous vos projets



41 QRQC

Comment réagir immédiatement à la détection de défauts, pour empêcher qu'ils ne se reproduisent à l'avenir ?

Dans le passé, pour chaque apprentissage nous ajoutions une case à cocher dans notre « definition of ready », ce qui alourdissait la gestion. Avec le QRQC, nous trouvons pourquoi un processus a conduit à l'introduction du défaut X par la personne Y, pour créer le standard et la formation manquante.

D'abord développé dans un contexte industriel, le Toyota Production System a permis à Toyota de se hisser au plus haut niveau mondial de qualité, stabilité, et de productivité.

Le Quick Response Quality Control (QRQC) est inspiré des méthodes d'amélioration continue de Toyota. Dès qu'un défaut de qualité est détecté, il est immédiatement analysé par le Tech Leader.

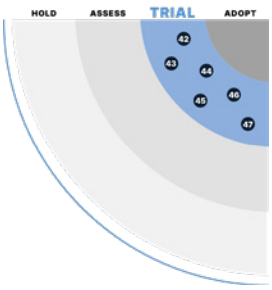
- Quelle ligne de code a introduit ce problème ?
- Qui ? Dans quelles circonstances réelles ?
- Quelles sont les causes profondes qui ont amené ce problème ? (ex: un manque de standard, de connaissances ?)
- Pourquoi on n'a pas détecté ce défaut plus tôt ? (mur de qualité)
- Qu'est-ce qui aurait pu empêcher d'introduire ce problème ?

Quelle que soit leur gravité, tous les défauts constituent pour l'équipe des occasions d'apprendre, de standardiser le travail, d'améliorer les outils et la DevX, de communiquer plus clairement, etc.

Nous avons une démarche d'animation autour du QRQC : dojos de formation, présentations de retours d'expérience, par le biais d'un template commun qui facilite les échanges. Cela nous permet d'accélérer la formation des développeurs et Tech Leads et de capitaliser sur l'expérience de chacun en diffusant les apprentissages.

Par ailleurs, chaque nouveau défaut est une occasion d'améliorer nos méthodes de travail petit à petit. La difficulté qui découle de cette méthode, si elle n'est pas maîtrisée, est la tendance à mettre en place des actions qui alourdissent le processus de développement. Il faut éviter cette erreur type.

Nous préconisons que tout Tech Lead fasse au moins 2 QRQC par semaine : c'est pour nous la meilleure méthode pour maintenir la formation en continu et la qualité à très haut niveau. Est-ce qu'il faut viser un QRQC pour chaque défaut ? Peut-être, mais il faut trouver le bon équilibre dans la profondeur d'analyse, la taille et la nature des contre-mesures, etc. pour maintenir un bon rythme de formation et la motivation de l'équipe.



42 Appium

Pour les tests e2e en React Native, Detox est en général largement plébiscité. En parallèle, Appium est souvent laissé pour compte, bien qu'il soit utilisé depuis 8 ans par la communauté native.

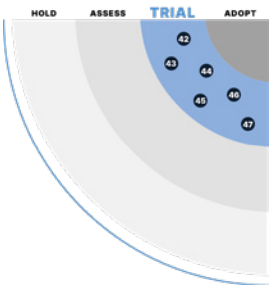
Nous avons identifié plusieurs raisons principales :

- Appium a rencontré des problèmes de compatibilité avec React Native, notamment pour récupérer un élément par test-id (crucial) : ces problèmes ont été corrigés par une mise à jour de React Native
- Sa documentation paraît obscure et labyrinthique, notamment parce qu'Appium supporte de nombreux langages et plateformes (mobile, web, desktop...)

Pourtant, une fois que l'on sait utiliser Appium avec Webdriver pour effectuer les opérations essentielles aux tests e2e (scroll, click, attendre une apparition d'éléments), l'écriture de tests devient similaire à Detox. De plus, cela vous permet de bénéficier de l'atout majeur d'Appium : cet outil effectue les tests en mode boîte noire. En d'autres termes, vous pouvez lancer un script de tests Appium sur n'importe quelle app, sans aucun setup préalable. Cela inclut votre release candidate ou même votre app de production depuis les stores. La vitesse d'exécution est supposée être plus lente, mais une étude très poussée contraste cette information.

Nous constatons sur nos projets que les deux fonctionnent. Si vos développeurs ont déjà de l'expérience avec une techno, nous vous encourageons à maintenir son utilisation.

Si vos QAs écrivent les tests automatisés, Appium peut leur permettre de le faire sans builder l'app. Vous pouvez privilégier l'excellente documentation de Detox ou son focus sur les applications mobiles.



43 Clean / Onion Architecture

En moyenne une application est ré-écrite tous les 4 ans. Ainsi, comment architecturer une application pour qu'elle soit facilement testable, maintenable et déployable tout en assurant une intégration rapide de nos futurs besoins fonctionnels sans augmenter le time to market ?

Chaque plateforme (iOS, Android, Flutter, React Native) utilise une approche différente pour gérer la vue d'une application, mais le cœur de l'app reste la logique métier. Dans l'optique de l'isoler pour la protéger des détails d'implémentation et la rendre testable, il convient de :

- Segmenter la code base en petits domaines (voir micro features architecture)
- Gérer les règles métiers au sein de ces domaines

Robert C. Martin, Jeffrey Palermo et d'autres architectes référents de l'*onion architecture* et la *clean architecture* apportent plusieurs recommandations :

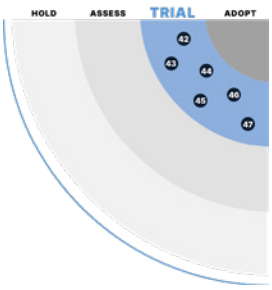
- Séparer les règles métiers des technologies utilisées, qui ne changent pas dans le même contexte. Tandis que le code lié aux règles métiers doit être compréhensible par quelqu'un du métier et ne changera qu'en cas de modifications des besoins fonctionnels, le code lié aux technologies est plus instable, les frameworks évoluant régulièrement.
- Utiliser l'injection de dépendances (ex: via koin ou resolver) pour que les règles métiers ne dépendent pas des technologies. Cela permet d'améliorer la testabilité et de rendre les changements d'infrastructure plus rapides.

Nous avons appliqué ces architectures sur le terrain pour nos projets iOS et Android, ce qui nous a permis d'émettre plusieurs constats :

- Un code métier plus clair et compréhensible
- Une simplification pour tester unitairement le code métier
- Des fichiers plus courts
- La possibilité de modifier une fonctionnalité sans introduire de régression
- Plus de facilité à repousser les choix de frameworks / librairies à utiliser. Cela permet de prototyper une app avec des choix d'infrastructures moins complexes (ex: persister de la donnée dans un fichier .txt plutôt que monter une base de données mySQL).

À ce stade, nos équipes étudient encore l'application de la clean architecture sur nos projets Flutter et React Native. La complexité apportée n'est pas toujours justifiable sur des projets courts et avec des équipes plus juniors. De plus, l'injection de dépendances en Javascript n'est pas simple. On peut cependant tirer profit d'une séparation en domaines métiers et d'une isolation de nos règles métiers via une architecture en couches.

Si votre projet est simple et qu'il ne contient pas de logique métier, l'implémentation d'une clean architecture n'est pas nécessaire. Mais si votre projet contient de nombreuses règles métiers complexes, nous vous recommandons de tester la clean architecture !



44 CVSS Security Audit

La sécurité est une préoccupation au cœur de chaque application mobile. Ainsi, comment sensibiliser une équipe de développement pour éviter qu'elle ne crée des failles inconsciemment, par manque de connaissance ou par inattention ?

L'une des premières étapes est d'avoir un schéma clair de l'architecture, comme un diagramme C4 de niveau 2 (container).

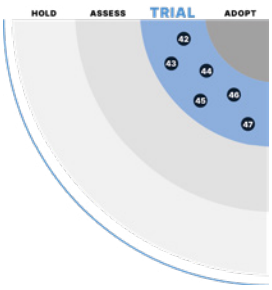
Ensuite, les équipes sont encouragées à se poser les questions suivantes pour les grandes fonctionnalités de chaque bloc :

- Qui : qui pourrait être l'agent de menace ?
- Comment : quel serait son vecteur d'attaque (accès direct à l'appareil ou accessible en ligne) et quelles préconditions sont nécessaires (malware installé, hameçonnage...) ?
- Quel impact : est-ce que cela affecte la confidentialité des données (accès en lecture à des données non autorisées), l'intégrité des données (accès en écriture ou suppression à des données non autorisées) ou la disponibilité (empêcher l'accès à d'autres) ?

Ces questions constituent la base du Central Vulnerability Scoring System (CVSS) : entrées dans un outil comme CVSS calculator, elles produisent une note allant de 0 (pas de risque) à 10 (vulnérabilité critique). Cela présente deux impacts positifs :

- Ces questionnements permettent aux équipes d'avoir un avis critique sur leur application, sans demander trop de connaissances au préalable. De plus, le classement des failles de sécurité les plus fréquentes OWASP aide les équipes à éviter qu'une faille ne soit déployée en production
- Les résultats permettent également d'avoir un système de notation stable et uniforme. Ce système est également utilisé pour les nombreuses failles de sécurité qui peuvent être présentes dans les dépendances téléchargées. Ceci permet donc d'avoir une notation homogène entre les failles internes et externes au projet. Ce système de notation donne des outils aux équipes pour prioriser ou arbitrer une décision technique

Nous vous conseillons d'utiliser CVSS pour évaluer le niveau de sécurité de vos applications.



45 Instabug

Lorsqu'un utilisateur remonte un problème au service client, les informations sont souvent incomplètes ou erronées. Plusieurs cas de figures sont possibles :

- La situation n'est pas claire : en tant que développeur, il est alors impossible de savoir ce qu'il s'est passé au moment du problème
- Il n'y a pas non plus d'éléments de contexte du problème, comme les logs applicatifs ou les requêtes envoyées
- D'un autre côté, les services de monitoring ne permettent pas de détecter toutes les mauvaises expériences utilisateur

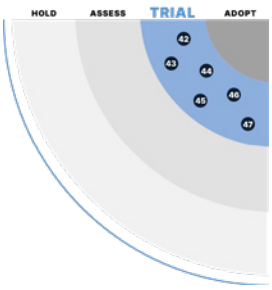
Instabug est un outil qui permet de monitorer et de prioriser les problèmes de performance et de stabilité d'une application. Il permet à l'utilisateur final de remonter un problème sur l'application en fournissant tout le contexte associé à celui-ci (logs, requêtes, étapes de reproduction, vidéo).

L'avantage principal est que l'utilisateur peut remonter un problème qui n'est pas visible sur un outil de monitoring classique, ou pour lequel il n'aurait pas appelé le service client.

Il peut afficher la modale de reporting en secouant le téléphone, en prenant une capture d'écran ou cliquant sur un bouton. Depuis cette modale, il peut alors préciser le problème rencontré. Cela permet aussi d'éviter l'interprétation du problème par deux interlocuteurs (l'utilisateur et le service client).

Nous avons eu l'occasion d'utiliser Instabug en production. Cela nous a permis d'obtenir des remontées précises des problèmes rencontrés et de les corriger plus rapidement. Instabug dispose de SDK natifs, React-Native, Flutter, Cordova, Xamarin, Unity et Web. Il n'existe cependant pas d'intégration Expo à ce stade. L'intégration est rapide et il est facilement possible de changer de service. L'outil propose un système de priorité et d'assignation ainsi que des intégrations avec les plateformes classiques comme Trello ou Jira.

Attention cependant, Instabug log tous les payloads https par défaut : nous recommandons de filtrer les données sensibles pour être conforme à la réglementation RGPD.



46 Key 4 Metrics

Aujourd'hui, le développement logiciel est le moteur des entreprises les plus performantes du monde. A titre d'exemple, que retrouvons nous de commun entre Amazon, Tesla et ING ?

La recherche (DORA) mène depuis 2013 une exploration des pratiques de développement, qui recense aujourd'hui plus de 2000 organisations et couvre tous les domaines d'application logiciel. Les découvertes et la recherche ont été publiées dans le livre *Accelerate: The Science of Devops.*

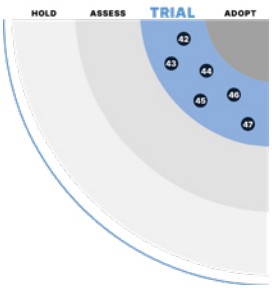
Cette étude scientifique, fondée sur la donnée brute, montre une corrélation forte entre le succès (productivité, profitabilité, croissance) et quatre métriques clés : la fréquence de mise en production, le taux de mise en production sans erreur, le Lead time pour qu'un code arrive en production, et le Lead time pour corriger un défaut en production.

Dans notre pratique du Lean, nous avons trouvé une vraie résonance dans ces quatre métriques clés, qui promeuvent toutes les boucles de feedback rapide et l'amélioration continue au service du client et des équipes.

La force principale de ce framework est l'approche rigoureuse basée sur les statistiques de nombreuses entreprises qui le soutiennent, et qui donnent une grande légitimité pour convaincre les décideurs. Le point faible du livre, c'est qu'il reste haut niveau et manque d'exemples pratiques.

Nous creusons actuellement le sujet afin de vous fournir une méthode plus concrète à l'avenir, pour optimiser ces métriques. Par exemple, l'approche de Sadao Nomura portée par Dantotsu nous guide beaucoup aussi.

Notre conseil : lire le livre Accelerate, et le tester dans votre contexte. Nous recommandons à tout le monde de s'approprier la théorie, et d'amener les changements par touches.



47 Observable

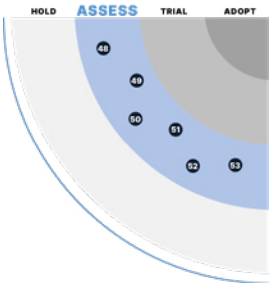
Pour gérer les flux de données dans nos applications, deux paradigmes s'offrent à nous : l'impératif et le déclaratif.

L'impératif consiste à appeler la donnée depuis le serveur. Cela implique de déterminer à quel moment le fetch des données doit être réalisé, de gérer le retour (succès, cas d'erreur etc.) puis de mettre à jour l'UI en fonction. L'impératif se traduit par un code lourd et verbeux.

Le déclaratif consiste à s'abonner à une ou plusieurs données, puis à indiquer à notre UI de se mettre à jour si une donnée spécifique est modifiée. C'est cette donnée à laquelle on va venir s'abonner qui représente un observable, c'est-à-dire un tuyau dans lequel des événements circulent. Il est possible de combiner ces tuyaux en utilisant des opérateurs, et la gestion de problèmes complexes devient alors plus simple.

Le principal inconvénient de ce paradigme est sa complexité à être pris en main, puisque le modèle mental est très différent du paradigme impératif.

Nous utilisons cette méthode sur nos projets pour des problèmes complexes en nous appuyant sur des librairies telles que RxSwift, RxJava ou Combine.



48 App Auth

Face à des risques grandissants en sécurité informatique sur les dernières années, la sécurité des applications mobiles est devenue elle aussi un sujet de préoccupation majeur.

Aussi, il appartient aux équipes de développement de se baser sur des standards existants et modernes en utilisant des librairies simples mais sécurisées dites de «haut niveau».

Parmi les sujets complexes associés à la sécurité, l'authentification représente un enjeu majeur, d'autant plus lorsqu'on passe par un fournisseur externe (réseau social, cloud).

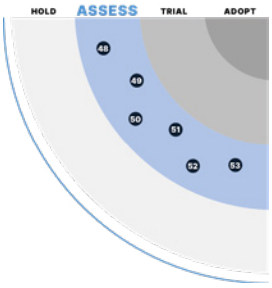
Il est donc important de suivre minutieusement certains standards tels que OAuth2 ou OpenIDConnect, bien qu'il existe encore à ce jour de nombreuses zones d'ombre dans les détails d'implémentation.

À titre d'exemple, le RFC 8252 interdit l'usage des «webviews» dans sa section 8.12, car elles présentent des risques de sécurité et de sérieux problèmes d'ergonomie.

Sur le terrain, nous observons de nombreux projets utiliser des webviews avec un deeplink ou un universal link, ou bien des webviews avec une communication via l'API postMessage(). Ces approches présentent des failles de sécurité.

AppAuth propose un package pour faciliter l'implémentation sécurisée du RFC 8252. Il est facile d'accès : via une librairie iOS et Android ainsi qu'un bridge React Native et Flutter, et permet ainsi de développer plus rapidement une solution plus sécurisée.

Nous souhaitons accompagner la prise de conscience de la sécurité côté développement mobile avec des recommandations de librairies et AppAuth nous semble être une solution intéressante à explorer pour sécuriser l'authentification.



49 Cognito

L'implémentation de l'authentification des utilisateurs sur vos apps peut représenter un investissement coûteux en temps, lorsqu'elle est développée de A à Z par vos équipes.

Il faut inclure dès le départ un ensemble de mécanismes fonctionnels de type mail de confirmation ou mot de passe oublié et assurer un niveau de sécurité élevé de bout en bout.

Sur nombre de nos projets, nous utilisons Cognito, la solution d'AWS pour l'authentification.

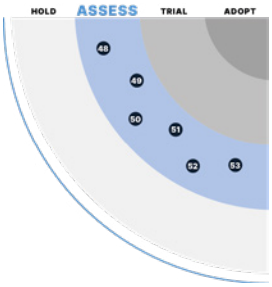
Très simple à mettre en place, elle permet d'implémenter différents cas d'usage simples (comme ceux mentionnés ci-dessus) mais également plus complexes et adaptés à vos besoins, en s'interfaçant avec AWS Lambda ou AWS Gateway.

Cognito autorise également une gestion intrinsèque des rôles sur votre application, qui permet de donner différents niveaux d'accès selon l'utilisateur. Elle gère l'encryption forte des données, nécessaire pour les applications avec fort enjeux de sécurité (ex: secteur de la santé). Enfin, elle permet une implémentation de la 2FA ou du social login ainsi que l'intégration avec un service SSO, en toute simplicité.

Via l'utilisation d'Amplify, Cognito propose également des composants UX à intégrer dans vos applications. Avec un «free tier» à 50 000 utilisateurs, elle est également peu onéreuse.

En revanche, nous avons rencontré quelques problèmes avec le SDK Flutter Cognito. De plus, de manière générale, la documentation d'AWS est parfois confuse et difficile à parcourir, à la différence de documentations très claires, issues de solutions plus onéreuses, comme Auth0.

Nous vous conseillons donc d'évaluer la pertinence de cette solution pour vos besoins. À titre d'exemple, si vos processus d'authentification sont très basiques, une autre solution peut être plus adaptée.



50 CRDT (yJS, Automerge)

L'une des promesses du mobile est de pouvoir collaborer tout en se déplaçant. Ainsi, comment gérer les contributions collectives lorsque le réseau laisse à désirer ? Comment résoudre les conflits réseaux, occasionnés lors d'une édition concurrente ? Ce questionnement complexe nourrit la recherche depuis les années 1990.

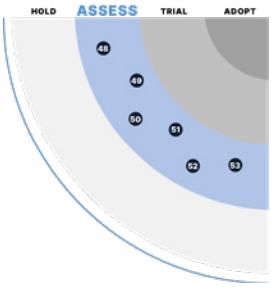
Depuis, de nombreux algorithmes ont vu le jour; à leur apogée: les CRDTs.

Une CRDT (Conflict Free Resolve Datatype) est une structure permettant de contenir données et modifications, de sorte à résoudre automatiquement les conflits les plus simples et à accompagner la résolution des conflits complexes (JSON CRDT, Kleppmann (2017)).

Les papiers de recherche sont souvent très complexes à comprendre, et encore plus à implémenter. Mais, il existe des librairies prêtes à l'emploi. En voici deux en Javascript:

- Automerge, écrite par Martin Kleppmann, l'auteur du livre «Designing data intensive applications»
- yJS qui implémente un autre algorithme et est plus performant sur des gros documents

Quoi qu'il en soit, si vous avez des besoins de collaboration asynchrone, nous vous conseillons de regarder du côté des CRDTs.



51 Ghost Programming

Pour simplifier le séquencement des tâches techniques d'une user story auprès de développeurs débutants, nous encourageons l'utilisation de Ghost Programming.

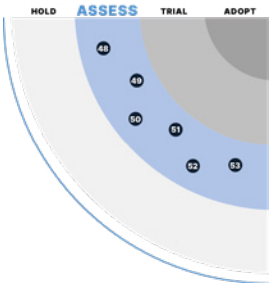
Cette technique consiste à les inviter à découper la user story en micro-tâches en amont du développement et à estimer le temps que prendra chaque tâche (5-10 min chacune). Cette estimation est ensuite challengée par les Tech Leaders, qui doivent être sollicités si la durée effective d'une tâche dépasse l'estimation prévue. Cette pratique de programmation présente de nombreux avantages pour des juniors :

- Préparer le développement en amont et anticiper tout blocage
- Éviter l'effet tunnel grâce à une meilleure visibilité sur le projet
- Identifier rapidement une difficulté pour réagir et solliciter son équipe en cas de besoin
- Faciliter le découpage en commits du code et simplifier la relecture

Présenter les avantages de cette technique en amont de son implémentation vous permettra d'embarquer vos collaborateurs, tout en évitant un sentiment de contrôle du travail.

Après avoir expérimenté le Ghost Programming chez BAM, nous avons observé une accélération de la vitesse de développement ainsi qu'une hausse d'autonomie dans la rédaction de solutions techniques par nos développeurs débutants.

Nous vous invitons à généraliser cette pratique qui constitue un puissant instrument de formation et de productivité au sein de vos équipes.



52 Github Copilot

Github Copilot est un assistant de développement qui permet d'écrire du code plus rapidement, en proposant une autocomplétion avec des bouts de code prêts à être utilisés. En se basant sur OpenAI Codex, Github Copilot analyse les commentaires et les noms de fonctions ou variables, pour suggérer une implémentation.

Nous avons regardé avec intérêt les opportunités portées par cet outil, dont l'arrivée a fait le buzz et a suscité des avis très partagés. En le testant chez BAM, nous avons eu de très bons retours sur plusieurs cas d'usage, notamment :

- Implémentation d'algorithmes simples, ex. : division, boucle, itération, tri
- Rendre les équipes plus efficaces dans un langage que l'on maîtrise peu (ex: configuration Fastlane en Ruby pour un développeur TypeScript)
- Recommandations des interfaces ou modèles typiques (ex: Country, Person)
- Proposition de test cases standard dans les tests unitaires

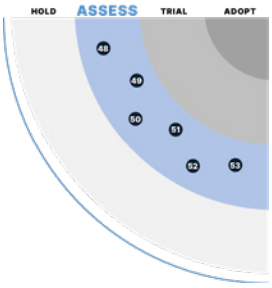
Le retour d'expérience positif sur ces cas d'usage est toutefois nuancé par des recommandations pas toujours pertinentes, voire erronées. Un de nos collaborateurs estime que : *« 50% des recommandations sont pas mal, 10% sont parfaites, mais que le reste n'est pas pertinent (ex: pas la bonne tech, syntaxe étrange) ».* Les retours d'autres membres d'équipe divergent, ce qui traduit une forte variabilité des résultats obtenus.

Quelques risques sont bien identifiés par la communauté, mais pas toujours maîtrisés :

- Le risque légal est lié à une utilisation potentielle du code copié d'un projet, avec une licence fermée. Cela entraînerait donc une entrave à la propriété intellectuelle. Les avis divergent, mais la situation n'est pas vraiment claire tant qu'il n'y a pas une jurisprudence à ce sujet aux US
- L'autre risque est plus lié au travail des développeurs avec une tendance à accepter une recommandation sans vraiment vérifier si le code est correct (surtout chez des développeurs juniors) et un sentiment de perte de sentiment d'ownership du code écrit

Sur cet outil, notre recommandation est un peu mitigée : dans certains cas de figure, Copilot peut vous faire gagner du temps mais les effets long-terme ne sont pas clairs. Peut-être qu'un outil plus précis utilisant la même techno pourrait nous éviter une utilisation massive des libraires de quelques lignes, comme left-pad ?

Pour le moment, nous vous invitons à tester Copilot avec prudence, et à étudier les résultats. A noter aussi que quelques concurrents arrivent sur le marché, comme Tabnine ou Amazon Whisperer.



53 Libsodium

La cryptographie est devenue un enjeu majeur ces dernières années. La hausse du nombre de communications et l'amélioration continue de la puissance de calcul des ordinateurs (et bientôt l'ordinateur quantique) ont conduit aux développements de nouveaux algorithmes.

La première règle à suivre est de ne pas élaborer ni d'implémenter ses propres algorithmes de chiffrement et d'utiliser uniquement les algorithmes développés par les des experts reconnus et audités par des organismes comme NIST.

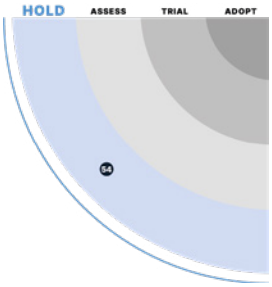
En 2021, l'OWASP classe la mauvaise utilisation de la cryptographie comme la faille numéro deux. Et sur le terrain, nous confirmons ce classement.

Alors que faire ? Comment choisir le bon algorithme ? Comment bien l'utiliser ?

Libsodium est un projet open source qui propose une bibliothèque de chiffrement. Pour chaque usage (chiffrement, authenticité, sauvegarde de mots de passe, etc.), il existe un algorithme unique, documenté sur le site libsodium.gitbook.io.

Par exemple, la sauvegarde de mot de passe utilise l'algorithme argon2i et est accessible via la fonction sodium_crypto_pwhash. L'algorithme est choisi pour sa sécurité et sa performance, en particulier sur du matériel embarqué moins puissant.

Nous recommandons d'évaluer Libsodium, en la comparant à d'autres solutions. A titre d'exemple, Libsodium est préférable à CryptoKit (iOS) ou Cryptography (Android) car elle présente un bon algorithme pour un cas d'usage, et que sa bibliothèque fait barrage à une mauvaise utilisation. Disposer d'une même bibliothèque sur le mobile et sur le serveur est aussi une garantie de compatibilité. Et si vos besoins ne vous permettent pas de choisir Libsodium, nous vous invitons tout de même à vous inspirer des choix d'algorithme. Libsodium est une bibliothèque C++ mais il existe également des packages pour React Native, Flutter, iOS et Android.



54 Firebase Authentication

Il est crucial pour une application de pouvoir authentifier ses utilisateurs. C'est un choix technique critique, qui aura un impact de taille sur la sécurité, l'expérience utilisateur et le coût de développement. Ainsi, il peut être tentant d'utiliser une solution d'authentification tierce mature, qui a travaillé en qualité et en profondeur tous les problèmes types et propose sa solution à un tarif intéressant.

Firebase Authentication est une solution attrayante pour permettre à votre application de résoudre rapidement tous ces problèmes. Nous l'avons utilisé à plusieurs reprises sur différents projets, dont certains à très fort enjeu de sécurité et de confidentialité des données.

Les principaux avantages de cette solution sont sa rapidité d'intégration ainsi que la réputation de son éditeur Google. De plus, elle permet de bénéficier d'une version d'essai gratuite. En l'utilisant, nous avons cependant découvert plusieurs points noirs :

- Le système d'authentification est extrêmement difficile à remplacer une fois mis en place (vendor lock-in)
- Cette solution est gratuite jusqu'à un certain nombre d'utilisateurs, mais représente ensuite une charge importante du projet, à prendre en compte dans les ambitions de croissance de l'application
- Une attention particulière doit être portée à la gestion des backups utilisateurs, car cette donnée ne vit pas avec le reste des données applicatives, hébergées sur d'autres serveurs. Il est important de garder ce point en tête, et d'en faire des sauvegardes cohérentes avec le reste des données

À noter également qu'à ce jour, Firebase ne fonctionne pas en Chine. De plus, nous ne sommes pas encore parvenus à clarifier si la gestion des données personnelles dans Firebase est pleinement conforme au RGPD.

En conclusion, nous pensons qu'il est préférable de s'abstenir d'utiliser Firebase pour l'authentification des utilisateurs. Nous vous invitons à étudier les répercussions sur le long terme et à considérer une autre alternative (ex: Amazon Cognito) ou une implémentation locale, dans la technologie de votre projet (ex: en SSO, AppAuth, ou OAuth). Voir aussi : Ory Hydra ; OpenIDConnect

10. Les contributeurs

Alexandre M. / App Performance Expert

Anaïs S. / Marketing Lead

Antoine D. / Head of React Native

Arthur L. / Head of Native

Cyril B. / Staff Engineer

François G. / VP Engineering

Guillaume D. / Head of Flutter

Louis K. / Tech Lead

Marek K. / CTO & Co-founder

Maxence L. / Tech Lead

Morgane J. / Head of Marketing

Nicolas H. / Tech Lead

Pierre C. / Tech Lead

Pierre P. / Tech Lead

Tycho T. / Principal Technologist

11. À Propos de BAM

2014 Avec **8 années d'expertise,** BAM est la startup spécialisée dans la conception et le développement d'apps mobiles du groupe M33.

+120 Nous sommes aujourd'hui plus de 120 **BAMers**, tous passionnés par les technologies mobiles et les impacts qu'elles ont sur le monde qui nous entoure.

+200 **Apps mobiles :** BAM a développé une véritable expertise mobile pluri-sectorielle en travaillant aux côtés de grands groupes et scale-ups comme TF1, Drouot, Bouygues, Promod, Urgo et Colas (...)

+15M **d'utilisateurs** sur les apps conçues avec nos clients.