Tech Radar

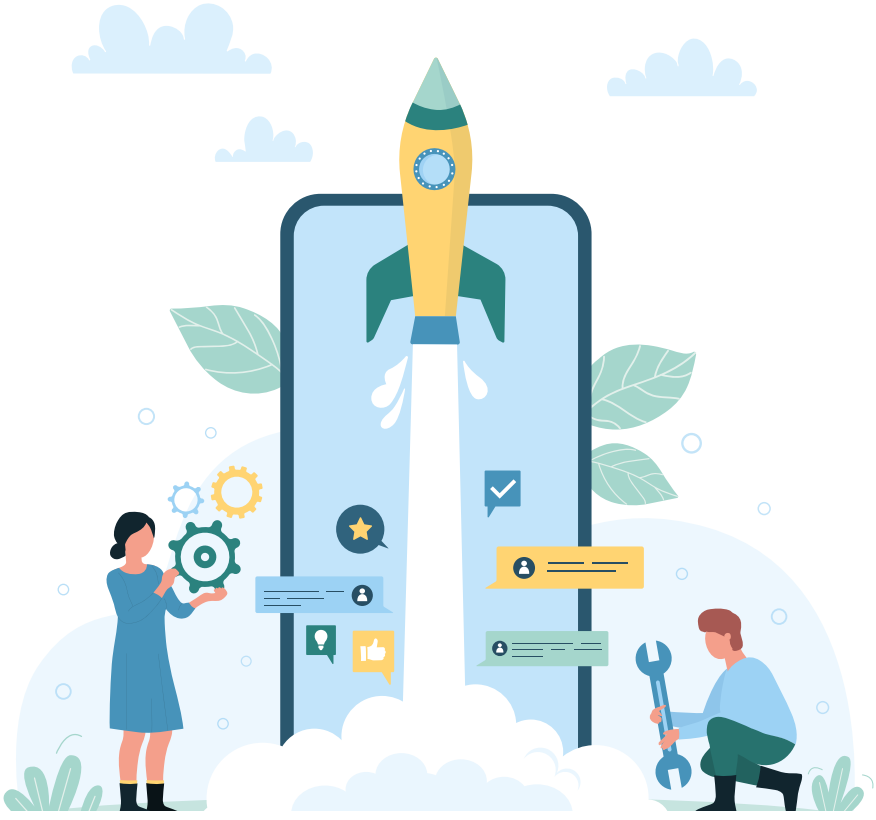
#2

JUIN 2023

BAM Part of Theodo Group

MOBILE

Inspiré du retour d'expérience de notre quotidien et co-construit par notre comité de contenu composé de 13 experts, notre Tech Radar est le premier à présenter un focus 100% mobile. Découvrez notre deuxième édition !



Introduction

Malgré une certaine maturité, l'écosystème mobile reste extrêmement dynamique. La multitude de technologies, de frameworks et de SDKs place chaque équipe mobile face à des choix complexes. Depuis des années, nous construisons une base de connaissances que nous avons décidé d'open-sourcer. Nous espérons que cela permettra à la communauté de bénéficier de nos expériences et de nos apprentissages.

Co-construit à l'occasion de plusieurs ateliers, ce Tech Radar est un snapshot des sujets qui animent les discussions au sein des 3 tribes tech BAM. 13 membres du comité de contenu se sont penchés sur les technologies et techniques qui constituent les 61 blips sélectionnés par notre équipe tech. Nous vous invitons à découvrir leur point de vue et à y réagir.

En nous fixant l'objectif de publier notre vision de l'écosystème mobile, nous avons opéré certains 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 clairs sur nos recommandations :** afin de vous guider au mieux, nous avons choisi d'adopter des prises de position claires et assumées.

Cette approche nous permet de présenter notre vision découpée en 4 niveaux de recommandation, sur la base de notre interprétation :

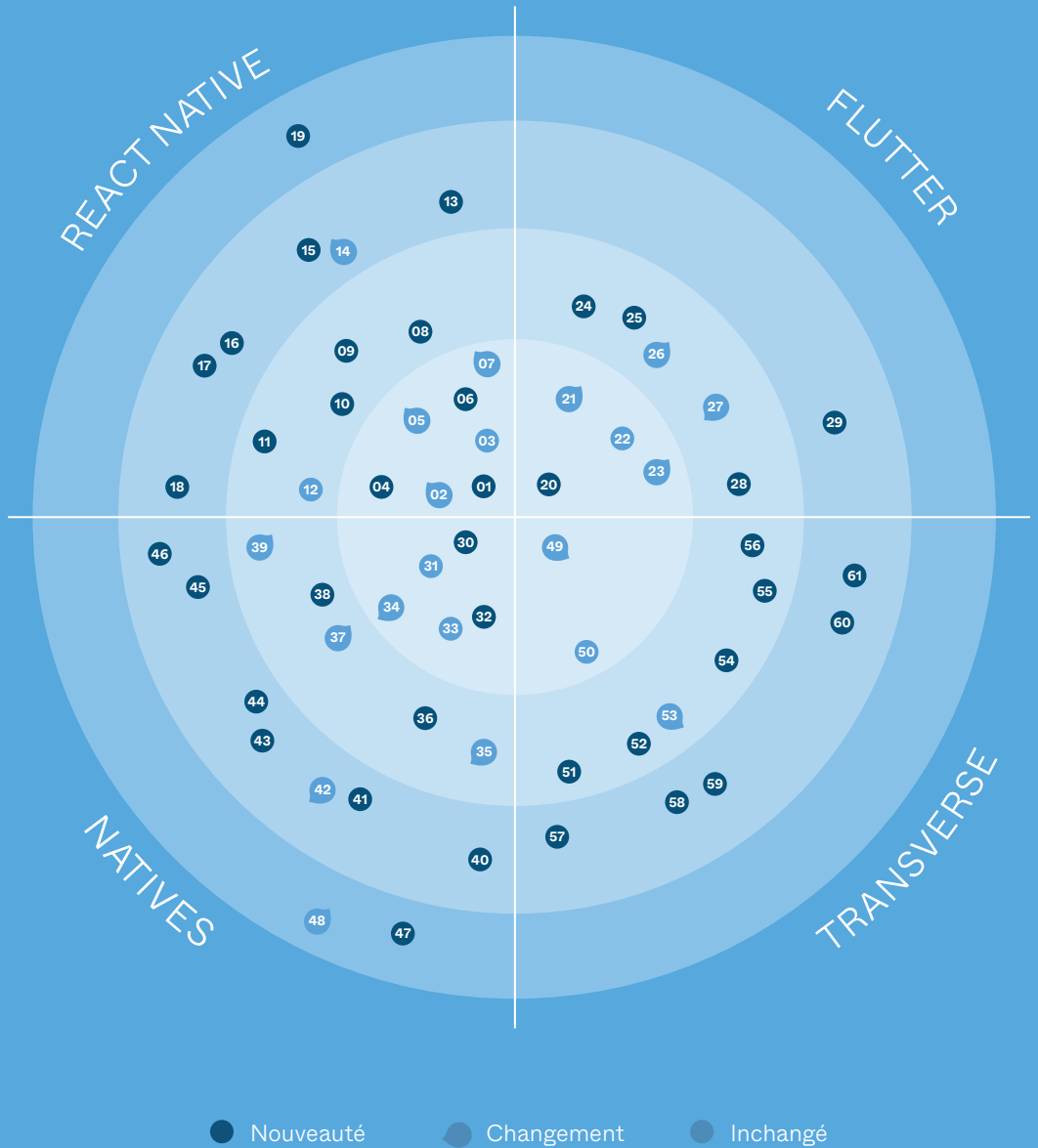
- **Adopt :** n'hésitez pas, c'est selon nous le meilleur choix à date ;
- **Trial :** nous vous invitons à tester cette technologie, car elle présente un fort potentiel pour répondre à vos besoins. Elle peut être intégrée dans un projet en production, si vous avez bien évalué les risques et les alternatives ;
- **Assess :** nous pensons que cette technologie mérite d'être surveillée car elle devrait gagner en importance dans les mois à venir. Il peut être intéressant de se documenter sur le sujet ou de réaliser une "proof of concept" ;
- **Hold :** nous vous déconseillons cette technologie à ce stade, soit parce que nous ne la jugeons pas assez mature, soit parce que nous l'estimons moins pertinente que ses concurrents. Bien que les évolutions futures puissent changer la donne, nous estimons qu'il est préférable de ne pas s'engager dans cette voie pour le moment.

Vos retours sont les bienvenus et nous serions ravis d'échanger sur nos visions et expertises techniques !

SOMMAIRE

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 : React Native, Native et Flutter.



Les blips

REACT NATIVE

1. EAS
2. Expo
3. Hermes Engine
4. MMKV
5. React Native Reanimated
6. Yarn 3
7. Zod
8. Enabling suspense with React Query
9. FlashList
10. NativeWind
11. React Native bundled for brownfield
12. React Native for Web
13. Custom Bundlers
14. Enabling the new RN architecture
15. Expo router
16. Reassure
17. Tamagui
18. tRPC
19. RN Context for state management

FLUTTER

20. Fast Immutable Collection
21. Melos
22. Pigeon
23. Riverpod
24. custom_lint
25. GoRouter
26. graphql_flutter
27. OpenAPI Generator for dart
28. Impeller for iOS
29. Shorebird

TECHNOLOGIES NATIVES

30. Gradle KTS
31. Jetpack Compose
32. KotlinX Serialization
33. SwiftUI
34. Tuist
35. The Composable Architecture
36. Hilt
37. Koin
38. Room
39. µ-Features Architecture
40. Factory
41. Glance
42. KMM
43. Media3
44. Paparazzi
45. RefreshVersions
46. Swift snapshot testing
47. JUnit 5 for Android
48. Texture

TRANSVERSE

49. 4 Key Metrics
50. Technical-Functional Design Diagram
51. 100% of coverage
52. Flashlight
53. Github copilot
54. Maestro
55. Right First Time
56. Rive
57. Co-writing software with ChatGPT
58. MASVS 1.5
59. No-code mobile
60. Ship, show, ask
61. TestPlan

Quelle alternative choisir ?

Vous remarquerez sans doute que notre radar n'émet pas de recommandation claire entre Flutter, React Native et Native. Ces 3 technologies définissent les cadrans, mais nous ne les comparons nulle part.

Lorsque nous avons lancé BAM en 2014, le choix technologique était très risqué. Nous étions convaincus que le multiplateforme était l'avenir du mobile et nous avons choisi notre stack technologique composée 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 (qui utilise des composants natifs en UI).

Chacune de ces solutions présentait à la fois des points forts bien distincts et des inconvénients importants. En 2015, le paysage a changé avec l'émergence de React Native, qui a permis de résoudre la plupart des problèmes rencontrés par les autres frameworks.

Nous l'avons adopté dès octobre 2015. Créé deux ans plus tard, Flutter a embrassé une approche techniquement différente, mais avec le même niveau de qualité. Au fil du temps, la solution Flutter a démontré sa valeur. En parallèle, l'émergence et la popularité croissante de Swift et Kotlin ont apporté beaucoup de fraîcheur et de modernité au développement natif, au détriment du développement multiplateforme.

Nous sommes donc passés d'une phase où le choix n'était pas évident (avant l'été 2015), à une phase où le choix était plutôt clair (2015-2018) avant de devenir à nouveau très complexe. C'est pourquoi nous choisissons une solution spécifique pour chaque projet et cela fait partie des premières discussions que nous engageons avec nos clients.

Ces discussions doivent 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 ?

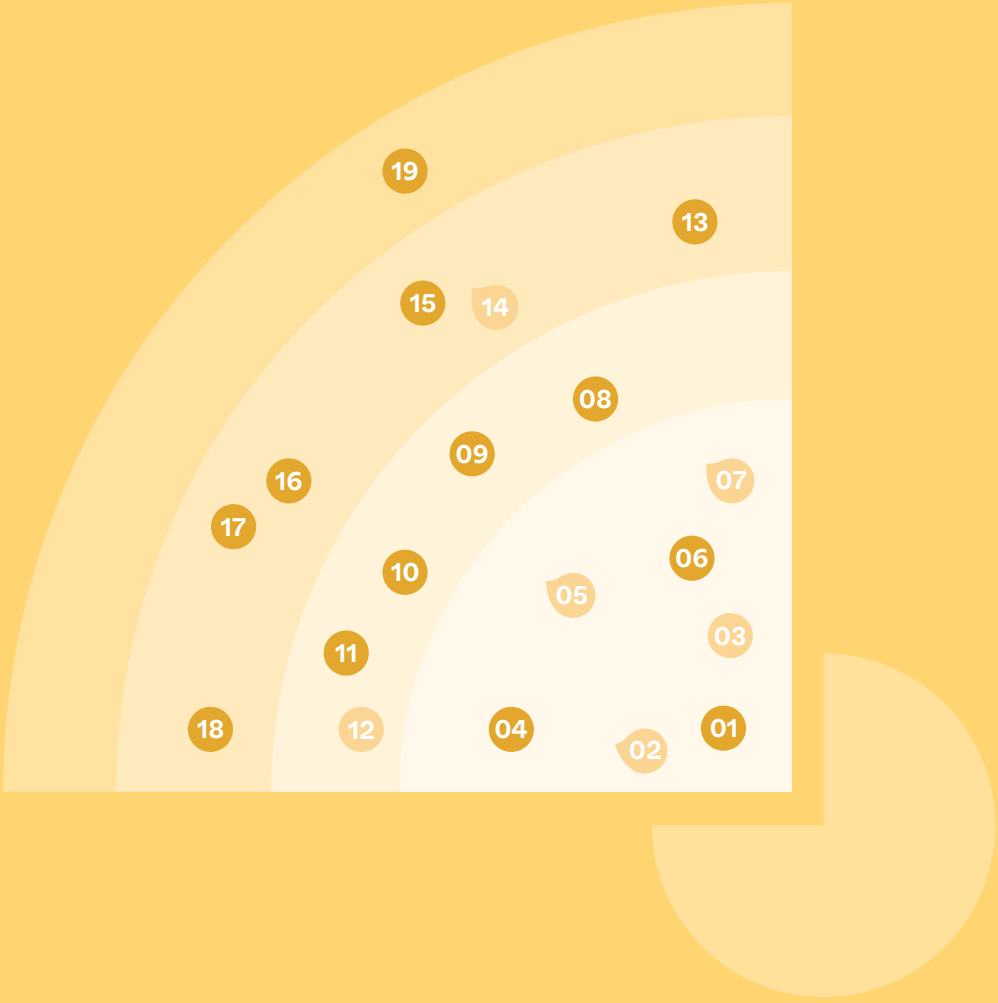
Cet échange nous permet d'évaluer le projet par rapport aux 3 technologies et d'émettre une recommandation, plus ou moins forte en fonction des contraintes.

Pour résumer, nous ne pouvons pas affirmer avec certitude que "la technologie A est meilleure que la technologie B". La décision doit être prise au cas par cas, avec des inputs de toutes parties prenantes et des experts mobiles qui connaissent les différentes solutions.

Nous serions ravis de discuter avec vous autour d'un café pour vous proposer une recommandation personnalisée pour votre app et votre entreprise.

LE CADRAN

React Native



19 BLIPS | 7 ADOPT | 5 TRIAL | 6 ASSESS | 1 HOLD

Nouveauté Changement Inchangé

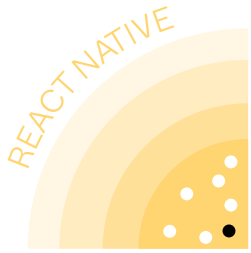
Expo et sa suite d'outils sont les grands gagnants de cette nouvelle édition du Tech Radar.

Au fil des années, ils sont devenus une brique essentielle de notre pile technique, ayant un impact significatif sur la livraison et la qualité de nos projets. Aujourd'hui, ils sont devenus la norme pour tous nos nouveaux projets.

Nous sommes parfaitement conscients des débats intenses qui entourent la performance et l'animation en React Native. Cependant, l'écosystème évolue rapidement et de plus en plus d'outils sont disponibles pour répondre à ces problématiques. Nous avons fait de la quête de la meilleure performance l'un de nos principaux combats. Vous découvrirez ici tous nos outils préférés pour atteindre et pérenniser des performances au plus haut standard du marché, même sur les appareils les plus modestes.

Enfin, nous sommes convaincus que le multiplateforme est une meilleure solution pour la grande majorité des cas. Nous contribuons au développement de l'expertise dans la migration de projets natifs existants en React Native, ainsi que dans le partage de code entre les bases de code web et mobile. Il n'y a qu'un pas pour passer du mobile à une application pour votre Smart TV !

PAR **CYRIL BONACCINI**
Staff Engineer



1 EAS

ADOPT
1/7 blips

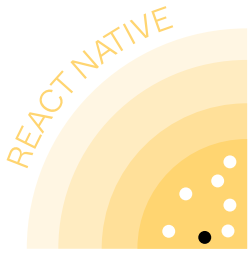
La route vers la mise en production d'une application React Native est longue et semée d'embûches : mise en place du build, signature de l'application, déploiement sur les stores, et ce pour chaque plateforme ! Plusieurs outils sont requis pour ce système de déploiement : fastlane, une CI avec des runners macOS, firebase, chacun exigeant sa propre configuration. Cette configuration, qui demande plusieurs jours ainsi que des connaissances pointues en développement mobile, ne garantit pas automatiquement un système de déploiement performant : la durée de build sur iOS est élevée, pouvant atteindre une demi-heure, sans optimisation de build complexe. Pour simplifier la mise en place d'un système de déploiement efficace, Expo a créé *Expo Application Services.* **EAS réunit plusieurs outils en un, permettant de builder et de déployer des applications React Native plus rapidement, plus facilement et à moindre coût :**

- EAS a été optimisé pour réduire drastiquement la durée de build, ce qui rend son système de déploiement continu très performant ;
- Expo a simplifié la configuration d'EAS en abstrayant le déploiement et la signature de certificats. Ces améliorations nous ont permis de réduire le temps de configuration sur nos projets, d'une moyenne de deux jours à seulement deux heures ;
- pour les modifications ne contenant pas de code natif, EAS offre un système d'update OTA (Over-The-Air) permettant de partager les mises à jour sans avoir à télécharger une nouvelle application.

Nous estimons que EAS coûte en moyenne 100$ par mois pour un projet de taille moyenne, mais reste également disponible gratuitement avec un temps d'attente.

NOTRE POINT DE VUE

Nous vous recommandons vivement d'utiliser EAS. Sa configuration est rapide et facile à mettre en place, ce qui en fait une excellente option, notamment pour les développeurs moins expérimentés dans le domaine mobile. En optant pour EAS, vous économiserez du temps et pourrez vous concentrer pleinement sur l'essentiel : le développement de votre application.



ADOPT
2/7 blips

2 Expo

Le monde du développement mobile peut être intimidant pour les développeurs novices dans ce domaine. En effet, ils devront faire face à des problèmes de build, de signature et de déploiement, qui requièrent des connaissances spécifiques au mobile. React Native apporte également son lot de difficultés, avec des mises à jour complexes et des bridges compliqués à implémenter.

Depuis quelques années, **le framework Expo apporte des solutions aux problèmes rencontrés par les développeurs React Native.** Ce dernier est constitué de plusieurs parties pouvant être utiles selon les besoins de vos projets :

- si vous débutez en React Native, utilisez Expo Go, un client mobile permettant de faire tourner votre code Javascript en fournissant les librairies nécessaires à la plupart des projets, éliminant ainsi les étapes de build, de signature et de déploiement qui sont les plus grands obstacles pour les nouveaux développeurs mobile ;
- si vous avez besoin de librairies natives qui ne sont pas incluses dans Expo go, utilisez le système Prebuild d'Expo, qui génère le code natif de votre application, dont vous n'avez pas à vous soucier. Les mises à jour React Native sont également grandement simplifiées, la difficulté résidant principalement dans les modifications natives ;
- si vous avez des modules natifs à créer, faîtes le à l'aide de l'API Expo Modules. Elle abstrait la complexité d'implémentation des bridges React Native, supporte Kotlin et Swift et facilite le support pour la nouvelle architecture ;

- si vous avez besoin de faire quelques modifications sur vos fichiers natifs, intégrez-les dans des config plugins.

Toutefois, si les modifications natives de votre projet sont complexes ou nombreuses, utiliser les systèmes de custom plugins et de Prebuild risque de rendre ces modifications plus coûteuses à développer et à maintenir. Dans ce cas, nous recommandons de versionner vos fichiers natifs, et d'adopter un développement en "bare workflow" ; Quel que soit votre cas, vous pourrez tirer parti des librairies du SDK Expo et de la CLI.

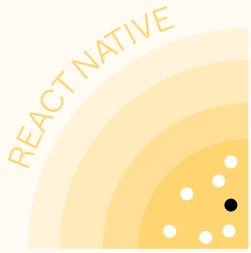
À l'issue de notre dernier Tech Radar, nous avions émis deux avertissements relatifs à l'utilisation d'Expo :

- d'abord, la difficulté pour modifier du code natif, qui a été diminuée par les Expo modules ;
- ensuite, l'impossibilité de déployer une application Expo en cas d'inaccessibilité du système de déploiement d'Expo (EAS).

En solution de secours, il est désormais possible de builder et de déployer son application en local. Ces améliorations nous permettent de passer Expo en "Adopt". Ainsi, **tout projet React Native peut tirer profit des différentes parties d'Expo en fonction de ses besoins.**

NOTRE POINT DE VUE

Pour les nouveaux projets, nous recommandons de démarrer en utilisant toutes les parties d'Expo répondant aux besoins de vos projets, pour abstraire un maximum de complexité le plus longtemps possible. Si à un moment un outil d'Expo ne répond plus à votre besoin, vous pourrez l'éjecter.



ADOPT
3/7 blips

3 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 de 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.**

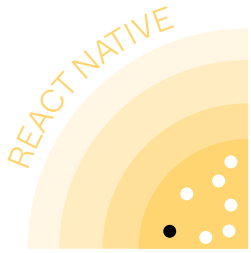
Ainsi, une app sous Hermes n'inclut plus de fichier JS embarqué, mais plutôt un fichier de code binaire. Les résultats sont frappants : grâce à l'utilisation d'Hermes, l'application mentionnée précédemment se lance désormais en seulement 3.9 secondes et le temps de démarrage de toutes nos apps a été réduit d'au moins la moitié. Passer enableHermes à true dans la configuration et ajouter quelques polyfills (ex : i18n, regexp) vous permettront de profiter de cette amélioration.

Certaines fonctionnalités sont encore manquantes (comme certaines APIs d'internationalisations) et pourront nécessiter l'ajout d'un polyfill. Mais la liste se restreint : par exemple, les regex capture group mentionnés dans la précédente édition de ce tech radar sont désormais supportés.

Hermes est désormais activé par défaut sur iOS et Android depuis la version 0.70.0. Cependant, d'après l'étude de Software Mansion "State of React Native 2022", Hermes n'est encore adopté que par 60% des équipes.

NOTRE POINT DE VUE

Pour les projets démarrés sans Hermes, nous vous conseillons de faire la migration pour l'activer. 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.



4 MMKV

ADOPT
4/7 blips

Les navigateurs web modernes permettent aux développeurs de stocker localement des données de type clé-valeur sur l'ordinateur de l'utilisateur grâce à l'API `localStorage`**. React Native s'est inspiré de cette API pour fournir une API similaire, appelée** `AsyncStorage`.

Cependant, contrairement à son homologue web qui est synchrone, `AsyncStorage` possède une API asynchrone en raison de l'utilisation du bridge React Native. Toutefois, dans la plupart des cas d'utilisation de `AsyncStorage` dans nos applications, la synchronisation serait préférable. `AsyncStorage` nous oblige à utiliser des Promises JavaScript, ce qui complexifie la persistance et l'hydratation de l'état au démarrage de l'application.

Heureusement, une alternative sérieuse à `AsyncStorage` existe : `react-native-mmkv`.

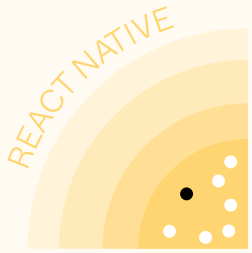
Cette bibliothèque est un module natif qui exploite la puissante technologie MMKV, un framework natif de persistance des données clé-valeur, développé par Tencent.

Les avantages de `react-native-mmkv` sont nombreux :

- une API synchrone grâce à l'utilisation de [JSI](./jsi.md) ;
- des méthodes 20 fois plus performantes que celles d'`AsyncStorage` d'après un benchmark effectué par le créateur de react-native-mmkv ;
- le chiffrement du stockage pour les données sensibles ;
- un ensemble de hooks pour synchroniser les composants React avec les changements de valeurs des clés.

NOTRE POINT DE VUE

Nous vous recommandons donc d'utiliser `react-native-mmkv` comme solution de persistance de vos données de type clé/valeur pour vos projets React Native, et nous vous invitons à prévoir un plan de migration de l'utilisation d'`AsyncStorage` vers `react-native-mmkv`.



5 React Native Reanimated

ADOPT
5/7 blips

Faire des animations en React Native n'est pas trivial. Bien que l'API Animated (fournie avec React Native) soit simple, le code est impératif et complexifie les composants. Le coût d'implémentation étant élevé, les animations ne sont pas souvent prioritaires dans le développement d'applications.

La bibliothèque Reanimated change la donne en offrant une approche plus déclarative pour les animations. Les animations sont simples à implémenter et n'affectent pas la lisibilité du code du composant. Les animations sont exécutées dans le thread UI, ce qui garantit des animations fluides et de bonnes performances.

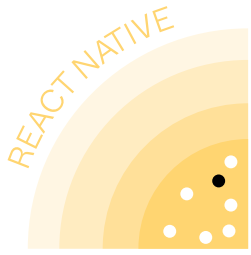
Pour des animations plus complexes, les hooks de Reanimated permettent d'extraire facilement des blocs de code d'animation dans des hooks séparés, permettant une réutilisation et une maintenance simplifiée.

Mieux encore, les Layout Transitions de la bibliothèque rendent les animations d'apparition/disparition ou de changement plus simples et intuitives. Ces tâches auraient été très complexes à réaliser sans l'utilisation de cette bibliothèque.

Reanimated améliore considérablement la qualité du code d'animation par rapport à Animated de React Native, tout en incitant les développeurs à implémenter des animations. La bibliothèque est activement maintenue et sa stabilité s'est nettement améliorée au fil des mois. Sa version 3 inclut des améliorations de performances et de qualité, ainsi que le support de la nouvelle architecture de React Native et les transitions d'éléments partagés. Reanimated est aujourd'hui un pilier de la stack technique d'un projet React Native.

NOTRE POINT DE VUE

Nous vous conseillons l'utilisation de sa version 2 si vos dépendances sont incompatibles avec la version 3 (utilisation de l'api legacy de la V1), sa version 3 sinon.



6 Yarn 3

ADOPT
6/7 blips

Les projets React Native ont besoin, comme tous les projets JavaScript, d'un package manager pour installer et mettre à jour les dépendances qu'ils utilisent.

La version 2 de yarn a été lancée en 2020 avec un objectif ambitieux : rendre possible les "zero-installs", avec lesquelles un "checkout" git suffit à avoir les bons packages installés, sans étape d'installation. Cette fonctionnalité remplace le dossier `node_modules` par un mécanisme appelé `plug-n-play` (PnP).

Ce mécanisme a en pratique été peu adopté car il nécessite une intégration spécifique dans chaque outil (nodejs, bundlers, IDE, TypeScript, etc...). Par exemple : metro, le bundler React Native, ne gère pas à ce jour le PnP, ce qui a donc bloqué l'adoption de yarn 2 par les équipes React Native. Néanmoins, yarn 3 a ré-introduit un mode de fonctionnement "classique" avec un dossier `node_modules`. Cela permet d'utiliser yarn 3 avec React Native (en ajoutant une ligne de configuration `nodeLinker: node-modules`) et de bénéficier de ses autres avantages :

- cette version est maintenue et plus rapide ;
- la version de yarn est obligatoirement fixée dans le projet, donc les installations sont plus déterministes ;
- des fonctionnalités qui nécessitaient des packages supplémentaires sont intégrées, comme la possibilité de patcher certaines dépendances, ou `yarn dedupe` qui permet de réduire la taille du bundle en respectant les contraintes de version fixées dans le `package.json` ;
- des plugins pratiques tels que celui qui installe automatiquement les packages `@types` pour TypeScript sont disponibles.

NOTRE POINT DE VUE

Avec le support des symlinks qui est arrivé récemment dans metro, le package manager pnpm pourrait devenir une bonne alternative, mais yarn 3 reste la meilleure option pour les projets React Native aujourd'hui. Nous l'utilisons sur nos projets et la transition s'est déroulée sans aucune difficulté.

7 Zod

ADOPT
7/7 blips

Au cours des cinq dernières années, TypeScript a considérablement modifié le paysage JavaScript, se hissant même parmi les cinq langages les plus utilisés en 2022, selon le sondage stack-overflow. La promesse de TypeScript est de garantir que si les données en entrée sont correctes, les données en sortie le seront également. Cependant, cela soulève plusieurs questions :

- comment vérifier que les données en entrée sont correctes lorsqu'elles proviennent d'un environnement non contrôlé (résultat d'API, formulaire...) ?
- comment intégrer cette vérification à l'exécution de notre application, étant donné que la vérification du typage a lieu lors de la phase de compilation ?

La librairie Zod offre une solution à cette problématique en permettant de générer des types et des validateurs associés à partir d'un schéma de données. Ces validateurs permettent de s'assurer que les données reçues correspondent bien au type attendu par l'application et protègent celle-ci contre les données mal formatées.

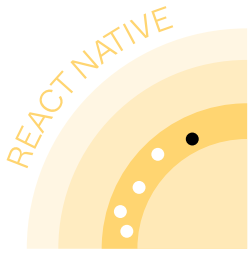
Les cas d'utilisation de Zod sont multiples :

- validation des variables d'environnement ;
- validation des retours d'API ;
- validation des formulaires.

Cependant, l'utilisation de Zod présente un inconvénient : la librairie requiert beaucoup de connaissances pour être utilisée efficacement. Bien que les développeurs seniors puissent l'utiliser efficacement, cela peut compliquer l'intégration des développeurs juniors. Néanmoins, la librairie peut être adoptée de manière incrémentale, ce qui facilite son adoption au sein des équipes.

NOTRE POINT DE VUE

Dans notre précédent radar, nous recommandions l'utilisation de Runtypes, une librairie concurrente offrant des fonctionnalités similaires. Toutefois, nous avons rencontré des problèmes de compatibilité avec le moteur JavaScript Hermès. De plus, nous avons constaté que Zod était plus complet et mieux maintenu que son concurrent. C'est pourquoi Zod est devenue la librairie que nous utilisons par défaut sur nos projets, et nous vous recommandons de faire de même.



TRIAL
1/5 blips

8 Enabling suspense with React Query

React Query s'est progressivement imposée comme une solution de référence pour la gestion des appels asynchrones et de l'état serveur dans les applications React. Nous l'avions d'ailleurs positionnée en "Adopt" dans notre précédente édition du radar.

Néanmoins, même avec React Query, le code peut devenir complexe lorsque les appels asynchrones se multiplient au sein d'une même page et que l'on souhaite garder une gestion fine des états de chargement et d'erreur. Ces appels doivent être faits (avec des `useQuery`) là où l'on souhaite afficher le loader (ex : au niveau d'un composant de page) plutôt que dans le composant qui a besoin des données. Les composants deviennent moins réutilisables.

Depuis React 18, Suspense n'est plus une fonctionnalité expérimentale et résout ce problème. On remplace les conditions `if(isLoading)` par un composant `<Suspense>`, positionné n'importe où, plus haut dans l'arbre de composants. React gère ensuite l'affichage du loader ou du contenu. Une fonctionnalité similaire existe pour les erreurs avec les "Error Boundaries".

React Query propose en expérimental, la compatibilité avec Suspense. Il suffit de configurer le client React Query avec `suspense: true`.

Activer cette option permet de bénéficier dès aujourd'hui des avantages de Suspense, de l'organisation du code au découpage des composants.

À noter que :

- le support de Suspense par React Query est encore expérimental, et des "breaking changes" sont à prévoir ;
- les APIs côté React vont probablement encore évoluer (avec le nouveau hook `use`) ;
- l'autre bénéfice majeur de Suspense, les améliorations de performance, nécessite l'utilisation de la nouvelle architecture de React Native.

NOTRE POINT DE VUE

Il est tout à fait possible d'activer Suspense dès aujourd'hui sur de nouveaux projets (le coût de migration des APIs actuelles vers les futurs changements sera plus faible que sans Suspense), mais pour des projets existants, il est préférable d'attendre avant de lancer une migration.

9 FlashList

TRIAL
2/5 blips

Pour afficher une longue liste d'éléments comme un fil d'actualités, React Native propose `FlatList`, qui permet de virtualiser l'affichage : seule une partie de la liste est rendue à l'écran et se met à jour dynamiquement au fur et à mesure que l'utilisateur scroll, garantissant ainsi une meilleure performance.

Cependant, par défaut, les `FlatList` rendent assez d'éléments pour remplir 21 écrans (oui, c'est énorme !), ce qui est coûteux. De plus, au scroll, la `FlatList` va parcourir tous les éléments affichés pour les re-rendre s'ils sont en effet dans la vue. Une mémorisation agressive permet de limiter l'impact en performance, qui restera cependant en deçà des composants natifs comme la `RecyclerView` d'Android.

`FlashList` de Shopify est une solution alternative qui répond à ces problèmes de performance :

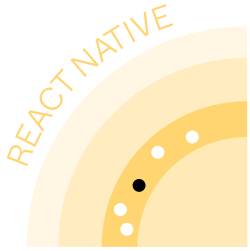
- elle est basée sur le composant JS `recylerlistview` de Flipkart qui permet le "recyclage" d'éléments. Au scroll, le recylage va réutiliser les éléments hors de portée et changer leurs props, ce qui diminue le temps de rendu ;
- cette technique permet également à `FlashList` d'optimiser le rendu et de rendre assez d'éléments pour remplir moins de 2 écrans.

Les gains sont flagrants : la consommation CPU au scroll est réduite de moitié sur la majorité de nos projets ! Migrer de FlatList à FlashList semble simple en théorie : il suffit de modifier deux lettres !

En pratique, les éléments de la liste peuvent nécessiter une adaptation dans des cas complexes. Comprendre le fonctionnement du recyclage, ajoute de la complexité (ex : il est déconseillé d'avoir un `state` propre aux éléments de la liste. De plus, nous avons rencontré quelques problèmes avec le support RTL, ce qui peut être un frein à l'utilisation de FlashList. À noter également, que le support web est encore en beta.

NOTRE POINT DE VUE

Nous recommandons de mettre en place la migration de FlatList vers FlashList pour profiter des gains de performance conséquents, tout en effectuant des tests de non-régression pour vous éviter de tomber dans les cas limites non supportés par la bibliothèque.



TRIAL
3/5 blips

10 NativeWind

La mise en forme dans React Native peut être lente et fastidieuse. Utiliser des stylesheets ou une librairie comme Styled Components peut être lent et ne garantit pas la cohérence des styles dans l'ensemble du code. Sur le web, Utility First Styling est devenue une méthode courante pour accélérer et améliorer la mise en forme, Tailwind étant l'exemple le plus populaire de ce concept.

L'idée de style Utility First est d'abstraire la stylisation courante que vous feriez dans des "utility classes". Par exemple, si vous souhaitez ajouter un padding de 4 rem, vous aurez une classe p-4 pour gérer cette fonctionnalité. Un autre exemple pourrait être d'arrondir complètement les angles d'une div, donc vous auriez une classe rounded-full pour gérer cette fonctionnalité. Utiliser le style de type "Utility First" signifie que vous devez combiner ces classes pour styliser complètement vos composants.

Étant donné que Tailwind est une solution d'abord conçue pour le web, quelques bibliothèques portent Tailwind pour React Native :

1. `tailwind-rn`
2. `twrnc`
3. `NativeWind`

Nous avons choisi d'inclure Tailwind dans notre cadran "trial" cette année, car il est largement utilisé sur le web et a été testé et approuvé au cours des dernières années. **Du côté mobile, nous pensons que NativeWind a atteint un niveau de maturité suffisant pour l'utiliser sereinement en tant qu'implémentation de Tailwind pour React Native.**

NOTRE POINT DE VUE

Nous vous recommandons d'utiliser NativeWind, car il offre la plus faible divergence par rapport à la bibliothèque originale Tailwind et un plugin Babel pour reproduire l'implémentation de Tailwind React. NativeWind est activement maintenu, prend en charge RN pour le web et offre de nombreuses fonctionnalités telles que le mode sombre, les classes arbitraires, les requêtes média, les thèmes, les valeurs personnalisées et les plugins.

11 React Native bundled for brownfield

TRIAL
4/5 blips

Un des plus grands avantages de React Native est de pouvoir partager le même code entre 2 voire 3 plateformes (Android, iOS, Web...). Mais si vous avez une application native existante avec une codebase importante, comment en tirer parti ?

Tout réécrire de zéro serait trop risqué : on ne peut tester en production qu'à la fin d'une longue refonte. Les applications dites "brownfield" sont des applications qui, tout en conservant leur codebase "legacy", ici leur codebase native, migrent progressivement vers la nouvelle technologie, React Native, permettant ainsi de mettre en production petit à petit. React Native dispose maintenant d'une documentation à cet effet. Mais si des développeurs natifs continuent à maintenir la partie native de l'app en parallèle de la migration en React Native, la configuration que cela implique sera disruptive pour eux. En effet, Node est alors nécessaire pour lancer l'application et metro doit tourner en parallèle de l'app native pour servir le JS (comme une app React Native habituelle). De plus, comme React Native et les libs associées (comme Reanimated ou react-native-webview) intègrent un code natif important, le temps de build natif est rallongé puisque ces dépendances seront souvent recompilées. Sur notre dernier projet Brownfield, le temps de build iOS était augmenté de 30 à 50% selon les machines des développeurs. L'alternative est d'empaqueter React Native, les libs nécessaires et le code JavaScript dans un SDK tiers : distribué sous la forme d'un .aar pour Android ou d'un `.xcframework` sur iOS. Ainsi l'intégration dans l'app native est simple : la partie RN s'active et s'utilise comme n'importe quelle lib tierce native.

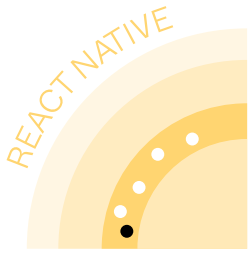
De plus, le code étant pré-compilé, l'impact sur les temps de build est négligeable. À noter que Walmart avait développé Electrode pour industrialiser ce processus, mais la solution est aujourd'hui peu maintenue et potentiellement lourde pour des équipes plus réduites que celles de Walmart.

En contrepartie, il devient plus compliqué de développer en même temps sur la partie native et la partie React Native de l'app. Des contrats d'interfaçage entre la partie native et le SDK React Native doivent donc être clairement définis au préalable.

Cette intégration est par ailleurs plus complexe que celle recommandée par la documentation et demande une véritable expertise native, nous avons dû par exemple modifier le plugin Gradle de RN pour assurer l'activation d'Hermes.

NOTRE POINT DE VUE

Nous vous recommandons donc de vous assurer d'avoir une équipe native de taille suffisante et l'expertise nécessaire avant de choisir cette solution.



12 React Native Web

TRIAL
5/5 blips

Une app étant souvent disponible à la fois sur le web et sur le mobile, il est naturel de vouloir partager du code entre ces plateformes, ce qui est possible pour la logique métier, le state d'UI et les calls API. Cependant, les composants UI ne peuvent pas être partagés par défaut.

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

Son installation est simple, il suffit d'installer react-native-web en dépendance de l'app web et et d'ajouter un alias de react-native vers react-native- web dans la configuration du bundler. Cela permet d'importer des composants React Native dans une app web React.

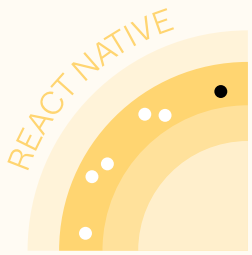
Nous avons pu partager entre 75% et 95% du code entre le web et le mobile en utilisant React Native Web sur plusieurs projets, et notre retour d'expérience est très positif.

Dans notre précédent radar, nous remontions un risque lié à la maintenance de la librairie par un seul développeur : Nicolas Gallagher. Embauché par Meta depuis, il a initié un travail d'unification des APIs web et native.
En échangeant avec nos équipes ou d'autres équipes ayant choisi d'utiliser RN for Web, nous avons cependant noté des points souvent sous-estimés :

- le code peut être plus complexe que du code React pur, ce qui peut affecter le débogage et les performances ;
- l'accessibilité n'est pas encore parfaitement gérée (ex : le composant FlatList n'est pas traduit par des balises ul et li) mais des optimisations sont en cours ;
- la mise en page peut différer entre le web et le mobile, il est donc souvent préférable de partager certains composants d'une page plutôt que sa totalité ;
- il est important de considérer le partage de code dès la conception des fonctionnalités pour maximiser les avantages ;
- l'utilisation de React Native Web requiert des compétences plus avancées en architecture dès le début du projet pour assurer un code maintenable.

NOTRE POINT DE VUE

Bien que React Native Web évolue dans la bonne direction, les compromis sur l'accessibilité et les difficultés rencontrées par certaines équipes nous empêchent de le recommander pleinement à ce stade.



ASSESS
1/6 blips

13 Custom Bundlers

Metro est le bundler par défaut pour React Native depuis 2017. Il fournit des fonctionnalités intéressantes (Live Reload, Hot Reload, build optimisé pour React Native) et répond aux besoins d'un grand nombre de projets. Toutefois, il présente certaines limites :

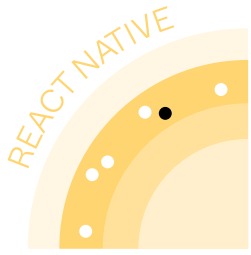
- les liens symboliques ne sont pas supportés, ce qui empêche l'utilisation de pnpm et complexifie la configuration des monorepos ;
- le découpage dynamique du code n'est pas supporté ce qui empêche la modularisation du front comme le fait webpack avec les modules fédératifs sur le web ;
- le tree shaking (suppression automatique du code non utilisé) n'est pas officiellement supporté ;
- les temps de build sont relativement longs par rapport à d'autres bundlers.

React Native profite des innovations de l'écosystème web et Metro est remis en question par d'autres bundlers :

- ESbuild : grâce à react-native-esbuild, un bundler implémenté en Go, il est possible de réduire le temps de build de notre bundle JS jusqu'à 10 fois et de diminuer la taille du bundle d'environ 20% grâce à l'implémentation du tree-shaking ;
- Repack : une surcouche de webpack proposée par Callstack qui amène le support du code splitting et des modules fédératifs ;
- rnx-kit : une suite d'outils proposée par Microsoft qui permet d'augmenter les possibilités offertes par Metro, comme le support des liens symboliques et du tree shaking.

NOTRE POINT DE VUE

Ces outils et librairies bousculent le monopole établi par Metro, mais restent encore très expérimentaux et requièrent des connaissances avancées en bundling JavaScript pour être configurés et utilisés sur des gros projets en production. Ils peuvent cependant répondre à certains de vos besoins incompatibles avec Metro.



ASSESS
2/6 blips

14 Enabling the new RN architecture

Annoncée depuis 2018, la nouvelle architecture de React Native est enfin disponible et activable.
Elle permet désormais :

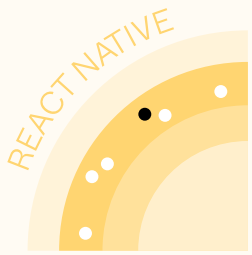
- de supporter de nouvelles fonctionnalités de concurrence de React, qui améliorent l'expérience utilisateur (ex : `startTransition` permet de garder l'app réactive même pendant une transition UI coûteuse). On bénéficie également des améliorations de performance liées à `Suspense` ;
- de mesurer et de rendre les vues natives de manière synchrones, pour 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 disponible uniquement sur Android).

Pour migrer vers cette nouvelle architecture, il suffit de changer un paramètre de configuration dans votre app. A condition que toutes les librairies de vos projets supportent cette nouvelle architecture. La liste de librairies manquantes a diminué depuis notre dernière édition, mais certaines librairies majeures comme `react-native-vision-camera` ou `flashlist` demandent encore du travail.

Dans notre précédent Tech Radar, nous avons constaté des builds longs (5 fois plus que la durée sans la nouvelle architecture), ce qui est fixé depuis la version 0.71. Hermes et React Native ne sont désormais plus compilés depuis le code source, mais distribués sous forme d'artefacts pré-compilés sur Maven Central. Dans la version 0.72.0, Meta a corrigé des régressions de performance que nous avions signalées. De plus, des améliorations ont été apportées à la documentation pour faciliter la migration des apps et de librairies existantes.

Dans la version 0.70.0, la fonctionnalité "auto-linking" a été ajoutée pour les librairies. En revanche, certaines fonctionnalités de concurrence de React 18 comme `useDeferredValue` semblent ne pas impacter la performance aussi positivement que sur React Web.

Notre recommandation : cette nouvelle architecture n'est pas encore prête à être activée sur vos apps, mais constitue le futur de React Native. Nous vous recommandons donc d'analyser la liste de librairies que vous utilisez et de vérifier qu'un plan de migration est en cours pour chacune d'entre elles, afin de préparer l'avenir. On espère d'ailleurs pouvoir passer la nouvelle architecture en "Adopt" dans notre prochain Tech Radar.



15 Expo Router

ASSESS
3/6 blips

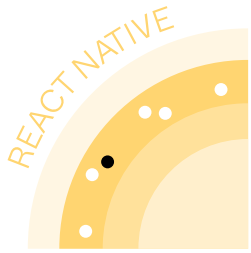
La navigation dans React Native a constitué un défi majeur pour les développeurs. Bien que React Navigation se soit imposée comme la bibliothèque de navigation de référence dans RN, elle peut être délicate à utiliser et entraîner des problèmes de performances. Les meilleures pratiques en termes de routage sont difficiles à définir, en particulier pour les apps universelles qui fonctionnent à la fois sur les plateformes natives et web.

Expo Router vise à simplifier ce processus en isolant certaines complexités de React Navigation et en standardisant la navigation à l'aide d'un routage basé sur le système de fichiers (FSR), devenu un standard dans le développement web grâce à des frameworks comme Next.js Cela diffère de l'approche déclarative adoptée dans React Navigation, où vous définissez les navigateurs dans votre code. Ex : là où vous auriez `<Stack.Screen name="Home" component={HomeScreen} />` dans React Navigation, pour ajouter un écran d'accueil, vous créeriez un fichier `home.ts` à l'intérieur du répertoire `app` avec Expo Router.

Expo Router présente plusieurs avantages, dont : la création automatique des deep-links, l'application des meilleures pratiques grâce au FSR et une compatibilité automatique avec les universal apps (native et web). Cela a un impact important sur la résolution de certains défis traditionnels, liés à la navigation. Globalement, les fonctionnalités d'Expo Router simplifient le processus de deep linking et de transition fluide entre le web et le natif.

NOTRE POINT DE VUE

Compte tenu de sa relative nouveauté, il y a encore de la place pour la maturité et l'amélioration de composants tels que la documentation et la stabilité du support TypeScript. Nous avons laissé Expo Router dans la catégorie "assess". Par conséquent, nous recommandons de garder un œil sur Expo Router car au fil du temps, il peut devenir une solution préférable à React Navigation.



16 Reassure

ASSESS
4/6 blips

La performance des applications mobiles est un enjeu majeur pour améliorer la rétention utilisateur. Pour les applications React Native, environ 80% des problèmes de performances proviennent du code JavaScript. Les régressions de performance sont parmi les plus difficiles à détecter, et les utilisateurs finaux sont souvent les premiers à les signaler.

Les tests E2E peuvent être utilisés pour répondre à cette problématique. Ils sont cependant coûteux à mettre en place, sont longs à exécuter et ne permettent pas de tester la performance des composants en isolation.

Callstack nous propose une nouvelle solution avec sa librairie Reassure. Reassure fonctionne en utilisant react-native-testing-library pour effectuer des tests de performance sur des composants React Native. Il mesure et compare les temps de rendu des composants entre différentes branches git afin de détecter les régressions.

Reassure présente des avantages clés :

- il s'appuie sur la bibliothèque de référence pour tester une app React Native. La configuration est simplifiée pour tout projet utilisant cette librairie et la courbe d'apprentissage pour rédiger des tests est quasi nulle ;
- l'exécution des tests est rapide ;
- l'adoption d'une approche plus unitaire des tests de performance ;
- il permet d'intégrer les tests de performance dans votre processus d'intégration continue. Vous pourrez ainsi bloquer le merge de toute pull request introduisant des régressions de performances sur les composants suivis par des tests de performance.

Malgré ses nombreux avantages, Reassure présente quelques limitations. Il n'évalue pas de façon absolue la performance du composant, un problème de performance dès la création du composant ne sera donc pas détecté. Il n'est pas compatible avec `jest-expo`, et la technologie est encore jeune et évolue rapidement.

NOTRE POINT DE VUE

Après une première phase de preuve de concept, nous avons décidé de déployer Reassure sur certains de nos projets.
Nous pensons que cette technologie va continuer à gagner en popularité et en efficience et nous vous conseillons de la tester.

17 Tamagui

ASSESS
5/6 blips

Les apps universelles qui fonctionnent à la fois sur le web et sur mobile, ont gagné en popularité. Toutefois, bon nombre des bibliothèques existantes (ex : React Native for Web) dans ce domaine ne traitent que les problèmes basiques. Les développeurs doivent donc composer avec les différences d'UI plus complexes, comme les comportements et mises en page des composants. Par exemple, la gestion des états des boutons sur le web vs. le mobile peut nécessiter une double pression sur les boutons en mode natif. Les apps mobiles utilisent des piles et des onglets pour les mises en page, tandis que les apps web utilisent des mises en page à plusieurs colonnes pour exploiter pleinement l'espace réel de l'écran.

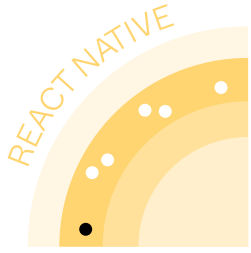
Tamagui est un kit d'interface UI complet, qui offre un style cohérent tout en partageant des tokens, des thèmes, des polices et bien plus encore sur le web et les plateformes natives. C'est un moteur de styles léger, avec des thèmes, des animations, du responsive, des pseudo-styles et des optimisations de performances, pour aplatir l'arborescence des composants avec une évaluation partielle. Il produit un minimum de CSS et de styles dans RN. De plus, il fournit des composants pré-construits pouvant être utilisés sur le web et le mobile, aussi bien pour des éléments courants (ex : les boutons) ou plus complexes (ex : mises en page) avec des implémentations spécifiques au web ou au mobile selon la cible.

Tamagui sépare les implémentations tout en unifiant les API, ce qui permet aux composants d'apparaître et de se comporter différemment sur le web et sur le mobile.

C'est une approche puissante, qui offre une expérience native selon la plateforme, bien qu'imparfaite : certains composants et éléments de la bibliothèque d'UI gardent un aspect et un comportement "web-first" ou "mobile-first". Cependant, dans la plupart des cas, les compromis effectués sont judicieux et pragmatiques.

Nous recommandons de considérer Tamagui pour une app d'abord conçue pour le mobile et pour laquelle la gestion d'une app web séparée est impossible. Tamagui peut constituer un puissant outil pour créer des apps universelles à partir d'une codebase unique.

À ce stade, nous avons choisi de le placer dans la catégorie "assess", car il n'est pas encore utilisé en production dans l'un de nos projets. Bien qu'il soit soutenu par Vercel et qu'il compte un grand nombre de stars et de contributeurs, il s'accompagne de peu de guides accessibles aux débutants, pour toucher un public plus large.



18 tRPC

ASSESS
6/6 blips

Les appels d'API posent un défi important en termes de typage, ce qui rend difficile la détection des breaking changes. Diverses solutions ont été proposées, telles que le partage de types entre frontend et backend, l'utilisation de GraphQL, OpenAPI avec codegen, ou gRPC avec Protobuf. Cependant, ces solutions peuvent ajouter une complexité à une codebase ainsi qu'une charge de travail significative.

tRPC adopte une approche intéressante pour obtenir la typesafety entre le le front et le back, en utilisant TypeScript. Avec tRPC, vous pouvez créer et consommer des API entièrement sécurisées par des types, sans avoir besoin de schémas ou de génération de code, ce qui élimine les problèmes liés aux contrats d'API.

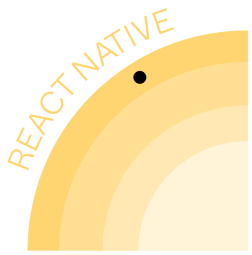
Parmi les principales caractéristiques de tRPC figurent la sécurité statique des types et l'autocomplétion pour les entrées du client.

Il est devenu bien connu dans la communauté pour son excellente expérience de développement. De plus, de nombreuses ressources et tutoriels sont disponibles en ligne, grâce à la popularité de la bibliothèque, ce qui facilite son apprentissage et son utilisation.

tRPC est une bibliothèque largement adoptée avec environ 25 000 étoiles sur GitHub et une équipe de cinq mainteneurs actifs, ce qui garantit la longévité du projet. Nous avons constaté que l'utilisation de tRPC a un effet très positif sur notre vitesse de développement et facilite notre cycle de développement full-stack. Il s'intègre bien avec React Query, ce qui signifie qu'il peut être complémentaire avec notre stack des appels API. De plus, son intégration avec les Edge Runtimes (comme Cloudflare Workers) permet un déploiement limitant les besoins en infrastructure.

NOTRE POINT DE VUE

Il existe encore des domaines dans lesquels tRPC est en retard par rapport à d'autres solutions, tels que le versionning d'API, l'internationalisation, ainsi qu'un manque de prise en charge de l'upload de fichiers. En conséquence, nous avons décidé de classer tRPC dans la catégorie "assess" à mesure que la bibliothèque évolue.



19 RN Context
for state management

HOLD
1/1 blip

La gestion de l'état global d'une app est un enjeu crucial pour assurer sa maintenabilité et ses performances. Concernant React, diverses solutions ont été proposées, `redux` s'imposant comme le choix par défaut. Toutefois, son utilisation pour la gestion de l'état global peut rapidement devenir complexe et difficile à maintenir.

L'arrivée des hooks et des contextes React a changé la donne, incitant de nombreux développeurs à les adopter pour gérer l'état global. Les contextes React offrent une solution pratique pour partager des données à tous les enfants appartenant à une branche de l'arbre de nos composants, et ce, peu importe leur position. Ils nous permettent ainsi d'éviter le "prop drilling". Si la donnée change, l'ensemble des enfants qui écoutent le contexte se re-rendent. Couplé aux hooks d'état, on obtient une implémentation possible d'état global. Cette implémentation pose néanmoins quelques problèmes :

- des problèmes de performance liés à des re-rendus excessifs, qui nécessitent l'implémentation de mesures de prévention comme la mémorisation qui seraient implémentées par une librairie spécialisée en gestion d'État ;
- le state n'est pas accessible en dehors de l'arbre de composants, ce qui peut être utile quand on souhaite le mettre à jour à la suite d'événements externes (ex : réception d'une notification push) ;
- les implémentations deviennent plus rapidement complexes du fait de l'utilisation d'API bas niveaux et de la présence de code boilerplate ;
- les outils de debugging sont plus restreints. On ne peut par exemple pas se reposer sur les redux devtools (compatibles avec d'autres solutions comme zustand) qui permettent d'analyser toutes les mises à jour de notre état et de remonter dans le temps.

Pour éviter ces problèmes, nous vous conseillons d'utiliser des bibliothèques spécialisées et éprouvées pour gérer les différents types d'états :

- cache serveur : `react-query` ou `apollo` ;
- état des formulaires : `react-hook-form` ;
- état global : `zustand` si l'accès au state en dehors de l'arbre de composant est nécessaire, `jotai` dans le cas contraire ;
- état local : les primitives React sont suffisantes.

NOTRE POINT DE VUE

En conclusion, nous vous recommandons l'utilisation d'APIs optimisées pour vos besoins en state management qui faciliteront l'intégration de nouvelles fonctionnalités tout en assurant de très bonnes performances. Réservez l'utilisation des contextes pour fournir des informations qui changent peu (thème, préférences utilisateur...) à travers l'arborescence de vos composants sans avoir à passer manuellement les props à chaque niveau.

LE CADRAN

Flutter

10 BLIPS | 4 ADOPT | 5 TRIAL | 1 ASSESS | 0 HOLD

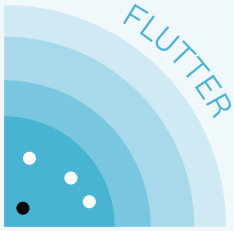
Nouveauté Changement Inchangé

Flutter s'affirme depuis quelques années comme un acteur majeur du développement d'applications pour le mobile et le multi-plateforme.

Cet écosystème très dynamique est en évolution constante avec de nombreux outils et librairies mis à jour régulièrement par une communauté en forte croissance.

Pour cette édition, le cadran Flutter présente un état de l'art optimiste de notre stack technique. Nos recommandations fortes de technologies éprouvées ainsi que celles présentant des limitations importantes que nous vous invitons à utiliser avec prudence. Nous y présentons également quelques technologies prometteuses que nous considérons comme clés pour l'écosystème Flutter et que nous espérons voir évoluer dans les mois à venir.

PAR **GUILLAUME DIALLO-BOISGARD**
Head of Tribe Flutter



ADOPT
1/4 blips

20 Fast Immutable Collection

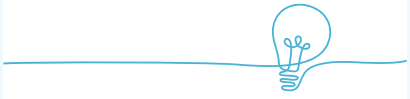
Travailler avec des collections mutables est toujours risqué en raison de leurs possibles effets secondaires. Dart propose `UnmodifiableListView` comme solution intégrée, mais elle n'est pas 100% fiable car elle peut échouer lors de l'exécution.
À la place, nous avons choisi d'utiliser **le package** `fast_immutable_collections`, qui **offre une liste immuable sécurisée lors de la compilation appelée IList** (ainsi que ISet pour les ensembles et IMap pour les cartes).

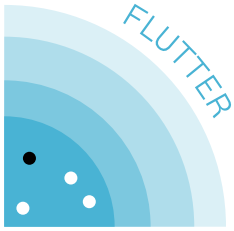
Nous utilisons actuellement IList pour toutes nos apps, car il s'agit d'un remplacement simple de List. De plus, il présente peu de dépendances et prend en charge une large gamme de versions de Dart, ce qui nous permet de l'utiliser également dans nos plugins sans risque de conflits de dépendances.

Non seulement IList est moins sujet aux erreurs, mais il offre également une syntaxe plus propre lorsqu'il est utilisé avec d'autres états immuables (comme les classes de `freezed`) en évitant le besoin de copie défensive. De plus, IList inclut des méthodes pratiques telles que `removeDuplicates` et `intersectsWith`. Le seul inconvénient que nous avons trouvé est que la syntaxe pour créer une liste constante (`const IListConst([])`) est légèrement laborieuse.

NOTRE POINT DE VUE

Avec très peu d'inconvénients et un impact positif important sur la qualité et la maintenabilité, nous avons intégré `fast_immutable_collections` à tous nos projets Dart (à la fois les applications et les plugins) et nous vous recommandons vivement de faire de même.





ADOPT
2/4 blips

21 Melos

Afin de réduire la complexité d'un projet de SDK, il est possible de le découper en plusieurs packages indépendants. Ces packages sont développés et versionnés ensemble, on parle alors souvent de "mono-repos". Cette architecture permet de séparer le développement, les tests et le déploiement de chaque brique en processus indépendants, mais cette modularité ajoute **une problématique majeure : comment gérer les interdépendances entre packages, à la fois en développement local et une fois publiés ?**

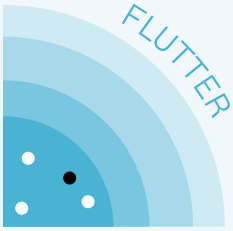
En Flutter, et plus généralement en Dart, ce problème est résolu par Melos, qui propose un outil en ligne de commande pour ces "mono-repos". Il résout les dépendances avec un linking local, permet le lancement de commandes sur l'ensemble des packages (analyze, test, scripts customs...) et propose automatiquement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

L'an dernier, nous adoptions une posture réservée sur Melos. Depuis, la documentation s'est grandement améliorée,de nombreux bugs ont été résolus et la version 3 a amélioré la syntaxe utilisée pour la configuration de Melos. Pour le debugging, un système de logging vous permet facilement de savoir dans quel package vos scripts ont rencontré un problème.

Melos étant développé par Invertase, un acteur solide de l'écosystème open source Flutter et Dart, nous avons confiance dans le fait que la librairie sera continuellement améliorée et mise à jour pour répondre aux besoins de la communauté. Les développeurs d'Invertase utilisent d'ailleurs Melos pour développer leurs propres outils, comme FlutterFire.

NOTRE POINT DE VUE

La facilité de développement garantie par Melos et l'absence d'alternative mature, nous ont décidés à adopter cette librairie pour nos projets de SDK Dart et Flutter. Nous recommandons Melos à tous les développeurs de packages, en particulier ceux qui travaillent sur des projets complexes avec plusieurs packages et équipes de développement.



ADOPT
3/4 blips

22 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. 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 deux problématiques :

- la quantité de code boilerplate à écrire est importante ;
- le typage sécurisé 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.

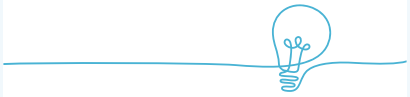
Pour pallier ces limitations, Flutter propose le package Pigeon. **Cette solution de génération de code permet de créer automatiquement l'ensemble des briques constitutives d'un PlatformChannel, en assurant le typage des données transférées.** Cette approche permet donc de gagner un temps considérable 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.

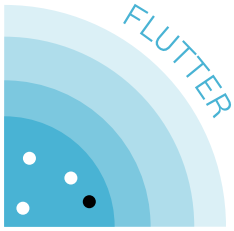
On note quelques limitations comme l'absence de support des events channels et l'incompatibilité avec des plateformes desktop. Des améliorations sont également possibles sur la syntaxe un peu lourde pour la gestion des enums ou le support des collections d'objets non nuls.

Ces limitations sont également présentes sans utilisation de pigeon et nous n'observons pas d'alternative plus efficace pour répondre à ce besoin. Par ailleurs, certaines évolutions à venir, comme l'utilisation de ffigen permettant d'appeler les langages natifs directement depuis le code dart, promettent d'offrir une solution encore plus performante et complète.

NOTRE POINT DE VUE

Après le développement de deux packages et près d'un an et demi d'utilisation, nous recommandons fortement Pigeon.





ADOPT
4/4 blips

23 Riverpod

En Flutter, la gestion du state global est réalisée grâce aux InheritedWidget et ChangeNotifier, mais ces APIs sont très verbeuses et bas niveau. C'est pour simplifier leur utilisation que Provider a été créé, et naturellement, la communauté Flutter a largement adopté cette solution. Plus récemment, le créateur de Provider a introduit un nouveau package de state management : Riverpod.

Riverpod a été créé pour répondre à 3 limitations de Provider :

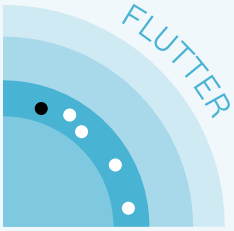
- être indépendant de Flutter pour pouvoir faire de l'injection de dépendances avant même l'instanciation de l'arbre de widgets et permettre de tester les responsabilités de manière isolée ;
- pouvoir injecter plusieurs variables du même type sans conflits ;
- être compile safe : les erreurs n'interviennent plus au runtime, mais dès la compilation pour détecter les problèmes le plus tôt possible.

Par ailleurs, Riverpod présente de nombreuses qualités telles que sa syntaxe minimaliste et sa capacité à être facilement mockable et testable. Riverpod évolue constamment avec le support de nouvelles fonctionnalités tel que le support du state asynchrone, des outils de refactoring intégrés, des règles de lint adaptées.

Enfin l'auteur du package a annoncé de belles évolutions pour le futur, notamment la persistance du state et le support du static meta programming.

NOTRE POINT DE VUE

Après l'avoir utilisée sur plusieurs projets en production, Riverpod est devenue la solution de *state management* que nous choisissons par défaut pour nos projets d'application mobile chez BAM.



TRIAL
1/5 blips

24 custom_lint

Le langage Dart mise beaucoup sur son excellente expérience de développement (DevX), assurée par sa syntaxe familière et ses puissantes intégrations aux IDE les plus connus comme VSCode ou Android Studio. Avec plus de 220 règles de lints créées par les teams Dart et Flutter, le linter est une des fonctionnalités les plus remarquables de ces intégrations. Cependant, il arrive que toutes ces règles ne suffisent pas pour mettre en application certaines conventions d'équipe particulières.

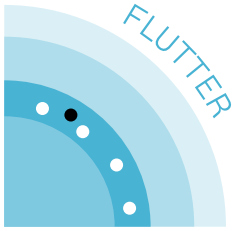
Dans cette situation, il est possible de recourir à la librairie custom_lint. **Grâce à une API relativement simple d'utilisation, cette librairie rend possible l'écriture de règles personnalisées pour l'analyzer Dart.** Ces règles peuvent ensuite être ajoutées à l'analyzer dans le fichier analysis_options.yaml, de la même manière que les règles classiques. Comme chaque règle créée avec custom_lint est un package Dart, il est possible de les publier sur pub.dev et de les partager ainsi avec l'ensemble de la communauté.

Créée très récemment, custom_lint ne peut pas être adoptée les yeux fermés. Plusieurs limitations restent gênantes dans l'intégration de la librairie à certains projets. Elle est particulièrement difficile, voire impossible, pour les mono-repos qui contiennent plusieurs packages Dart ou Flutter ne partageant pas les mêmes dépendances.

Le package étant activement en développement, ses dépendances sont très souvent mises à jour et imposent des contraintes parfois difficiles à suivre, sur les projets aux nombreuses dépendances qui ont souvent un cycle de mise à jour plus long. Pour tirer le plein potentiel de custom_lint, il est nécessaire d'avoir une connaissance intermédiaire de la librairie analyzer. Cette dernière est de très bas niveau et présente un coût d'entrée important.

NOTRE POINT DE VUE

Après l'avoir utilisée pendant plusieurs mois sur un projet d'application Flutter, et sur un projet de SDK, nous pensons que custom_lint est une librairie prometteuse, qui devrait rapidement devenir une référence incontournable dans la communauté. En raison de l'absence d'alternative fiable et des limitations mentionnées précédemment, nous avons décidé de placer custom_lint en “trial”. Nous avons l'intention de suivre de très près son évolution dans les mois à venir.



TRIAL
2/5 blips

25 GoRouter

La première API de navigation en Flutter, appelée Navigator, est relativement simple. Pour naviguer vers une page, il faut la pousser sur la pile de navigation. Pour revenir en arrière, il faut la retirer de la pile. Malgré sa simplicité et sa popularité, le Navigator comporte des lacunes et des limitations :

- le support des URLs et des deeplinks ;
- le support des actions "suivant" et "précédent" ;
- la possibilité de pousser plusieurs pages dans la pile de navigation.

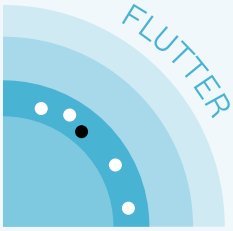
Avec le support du web, l'équipe Flutter a introduit une nouvelle API de navigation : Router. Contrairement à son prédécesseur qui fonctionne de manière impérative, le Router est une API déclarative.

Cependant, cette API est d'un niveau extrêmement bas. Pour s'en servir, l'utilisation d'une librairie est nécessaire.

Plusieurs options sont disponibles, comme RouteMaster et Beamer, labellisées Flutter Favorite. Sur nos projets, nous utilisons GoRouter. Initialement indépendant, GoRouter est depuis maintenu par l'équipe interne de Flutter. Cette librairie supporte la navigation par URL, les deeplinks sans configuration supplémentaire et permet d'écrire facilement des tests de navigation. Il existe même une extension, basée sur la génération de code, qui permet de typer les différentes routes. Notre expérience avec GoRouter est globalement positive, bien que nous relevons des problèmes lors de l'utilisation de navigateurs imbriqués.

NOTRE POINT DE VUE

Nous vous recommandons d'utiliser GoRouter mais avec une certaine précaution tout en gardant un œil sur les alternatives sérieuses disponibles, d'où un positionnement en "trial".



TRIAL
3/5 blips

26 graphql_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 avec le package graphql_flutter (avec 3.1k stars github à ce jour).

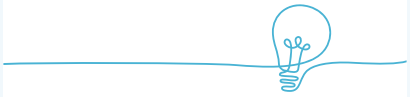
Le package graphql_flutter fournit une intégration transparente avec GraphQL, permettant aux développeurs de définir des requêtes et des mutations en utilisant des chaînes de caractères GraphQL. De plus, il offre la possibilité de faire du cache, du polling/rediffusion et du support des résultats optimistes, qui sont des fonctionnalités très utiles pour améliorer les performances et l'expérience utilisateur de l'application.

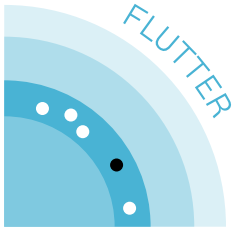
Le package graphql_flutter est également compatible avec le package graphql_codegen, qui permet la génération de code pour le typage fort des objets manipulés et le support des fragments. Cette fonctionnalité est particulièrement utile pour les grands projets qui nécessitent un code robuste ainsi que facilement maintenable et réutilisable.

Bien que graphql_flutter pousse à faire les calls apis directement dans vos widgets ou via des hooks Flutter, il est tout à fait possible de l'utiliser avec d'autres solutions de state management sans trop de difficultés. Cependant, l'utilisation des streams du client GraphQL dans ces solutions de state management peut se révéler complexe.

NOTRE POINT DE VUE

Notre expérience de l'utilisation de GraphQL en Flutter s'est nettement améliorée depuis l'année dernière et nous sommes désormais plus confiants pour l'utiliser sur des projets.





TRIAL
4/5 blips

27 OpenAPI Generator for dart

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 Dart, 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 OpenAPI 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.

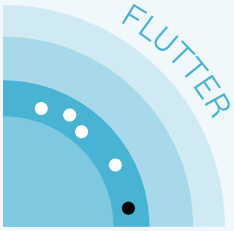
Cette approche présente de nombreux bénéfices :

- 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 éprouvé cette approche sur plusieurs projets en production, nous avons constaté un frein majeur à la mise en place de cette génération : certaines spécifications OpenAPI ne sont pas supportées nativement par le langage Dart, spécifiquement les objets qui peuvent être de plusieurs types (`oneOf` et `allOf`). Un client dart généré via le générateur OpenAPI ne supportera donc pas ces types. Il est alors nécessaire de définir les modèles API avec une librairie basée sur la code generation comme Freezed.

NOTRE POINT DE VUE

Même si nous utilisons un générateur OpenAPI sur chacun de nos projets en production, cette limitation importante, à laquelle nous avons été confrontés à plusieurs reprises sur nos projets, nous pousse à rétrograder cette solution en "trial".



TRIAL
5/5 blips

28 Impeller for iOS

Flutter utilise Skia comme bibliothèque de rendu cross plateforme pour dessiner les interfaces utilisateur. Skia est une bibliothèque performante, optimisée pour les processeurs graphiques modernes, offrant à Flutter des performances élevées sur les plateformes mobiles. C'est également le moteur de rendu graphique de chrome.

En utilisant des shaders, des programmes permettant de manipuler les graphismes et les effets visuels, Skia peut produire des effets sophistiqués et des transitions fluides au sein de l'interface utilisateur.

Néanmoins, le processus de compilation des shaders, effectué au runtime, peut entraîner des problèmes de jank (ralentissements et saccades) lors de l'exécution d'une application Flutter, notamment sur iOS.

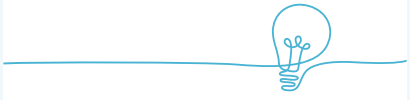
Pour remédier à ces problèmes, une solution courante consiste à pré-compiler les shaders au build time. Cette approche est actuellement limitée à iOS. Sur Android, cette méthode n'est pas compatible avec tous les modèles de smartphone qui utilisent différents pilotes graphiques. Cette étape de pré-compilation est chronophage car elle nécessite de parcourir l'ensemble de l'application sur un appareil physique pour "enregistrer" les shaders pré-compilés, ce qui la rend difficilement scalable.

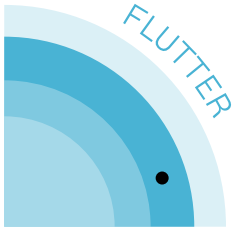
C'est pourquoi Flutter a introduit Impeller, un nouveau moteur graphique, conçu pour remplacer Skia et résoudre les problèmes de jank. Impeller est capable de compiler un ensemble plus petit et plus simple de shaders. Il peut également s'appuyer sur les APIs graphiques natives comme Vulcan sur Android et Metal sur iOS, pour tirer parti de la puissance du processeur graphique du téléphone.

Récemment devenu disponible sur iOS, son utilisation permet d'obtenir des performances graphique bluffantes. Il est désormais activé par défaut sur les apps iOS depuis la version 3.10 de Flutter. Sur nos applications mobiles, nous avons observé une nette amélioration de la fluidité des animations. Nous avons cependant constaté quelques bugs visuels, qui sont néanmoins rapidement fixés par l'équipe Flutter.

NOTRE POINT DE VUE

Impeller apparait clairement comme le futur moteur graphique de Flutter. Nous vous recommandons de l'activer sur vos projets, tout en restant vigilants aux régressions visuelles qu'il pourrait introduire.





ASSESS
1/1 blip

29 Shorebird

En développement mobile, les cycles d'itérations sont bien plus longs qu'en web :

- chaque nouvelle version d'application mobile doit être validée par les stores (Apple et Google) rallongeant ainsi le temps entre le moment où un correctif urgent est implémenté et le moment où il est disponible pour les utilisateurs ;
- il faut attendre que chaque utilisateur décide de mettre à jour l'application pour bénéficier du correctif ;
- dans les environnements de staging et QA, qui peuvent être déployés plusieurs fois par jour, les temps de build natif mobile et le fait de devoir télécharger les nouvelles versions de l'application ajoutent de la friction dans les processus de déploiement.

Pour répondre à ce problème et déployer des changements mineurs sans passer par le store, les applications développées en React Native bénéficient depuis plusieurs années maintenant de solution du type: "over-the-air" update CodePush, Expo Updates qui permettent d'envoyer des mises à jour de l'application développées en javascript, sans passer par les stores. Cette solution est très appréciée par les développeurs et les utilisateurs, mais elle n'est pas disponible pour les applications développées en Flutter.

Début 2023, le créateur de Flutter, Éric Seidel a annoncé la création d'un projet open source nommé Shorebird, qui permettra de déployer des mises à jour des applications Flutter sans passer par les stores. Pour développer cette solution, il a été rapidement rejoint par Felix Angelov, dont le nom est familier aux développeurs Flutter puisqu'il est l'auteur des packages BLoC et Mason.

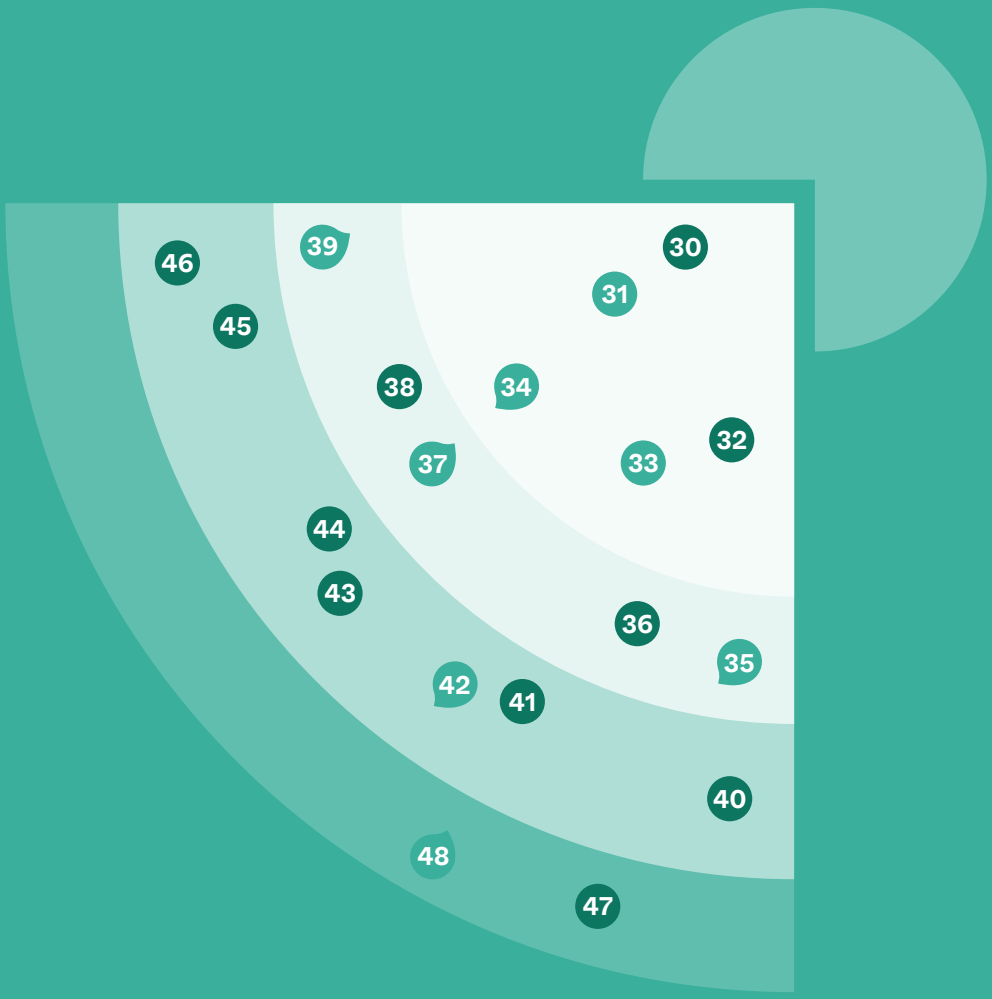
À l'écriture de ce blip, la solution Shorebird est toujours en phase early stage de développement et ne fonctionne que pour Android. Il est donc trop tôt pour se prononcer sur son adoption.

NOTRE POINT DE VUE

Notre expérience de CodePush nous donne envie de bénéficier d'une solution similaire pour nos applications Flutter. De plus, la renommée des deux développeurs impliqués dans le projet nous rassure quant à la qualité de la solution qui sera proposée.

LE CADRAN

Technologies Natives



19 BLIPS | 5 ADOPT | 5 TRIAL | 7 ASSESS | 2 HOLD

Nouveauté Changement Inchangé

Dans notre tribu dédiée au développement natif sur mobile, nous sommes fermement convaincus que les technologies natives jouent un rôle crucial dans la création des produits les plus innovants et la création d'une expérience utilisateur hors pair.

Les plateformes iOS et Android connaissent actuellement une période d'innovation très active, avec l'émergence de nombreux outils qui modernisent les plateformes et réduisent leurs pains points historiques. Des librairies telles que GradleKTS, Tuist, et refreshVersion illustrent comment la communauté améliore la productivité et l'expérience des développeurs. C'est cette dynamique que nous avons souhaité mettre en avant dans l'élaboration de notre tech radar.

PAR **ARTHUR LEVOYER**
Head of Tribe Native

ADOPT
1/5 blips

30 Gradle KTS

Avec les premières versions du SDK, les plus anciens développeurs Android se souviennent qu'il fallait utiliser l'IDE Eclipse et le système de build Ant pour développer son app. Ce dernier étant basé sur des fichiers XML, il fallait passer par des scripts externes dès qu'il fallait ajouter un peu de logique à la construction de l'app. En 2014, l'expérience de développement connaît une avancée majeure avec l'arrivée d'Android Studio et le système de build Gradle. Basé sur le langage Groovy, Gradle permet de coder une logique de build aussi complexe que nécessaire et a engendré tout un écosystème de plugins.

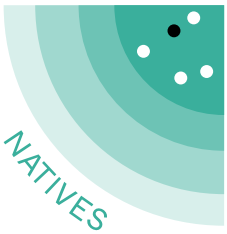
Cependant, Groovy étant un langage à typage dynamique, il est compliqué d'y implémenter une auto-complétion pertinente. De plus, certains problèmes ne sont visibles qu'à l'exécution du programme. **Depuis la sortie d' Android Gradle Plugin 4.1 en 2020, il est possible d'écrire les scripts Gradle directement en Kotlin.**

On peut donc écrire ses scripts de build avec la même syntaxe que le reste de l'application et on a accès à toute la puissance de Kotlin pour le faire. Le typage fort permet de découvrir la syntaxe possible en fonction du contexte dans lequel on se trouve, même si les concepts de Gradle se passent difficilement d'avoir à consulter la documentation.

Sur le plan technique, l'interopérabilité est assurée. Cependant, l'intégration d'un plugin non documenté pour cette nouvelle syntaxe, exploitant la dynamique de Groovy, peut poser quelques difficultés. De plus, il est regrettable que les applications générées avec Android Studio continuent d'utiliser des scripts en Groovy, qu'il faudra convertir à la main en KTS.

NOTRE POINT DE VUE

Malgré ces légères contraintes, il est important de noter que de plus en plus de projets adoptent chaque jour KTS, qui devient le nouveau standard vers lequel migrer, sans réelle urgence, mais si l'occasion se présente.



ADOPT
2/5 blips

31 Jetpack Compose

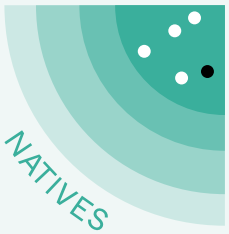
Les frameworks UI suscitent souvent des débats : templates externes ou implémentation programmatique, outil graphique pour la conception ou écriture manuelle ? La technologie historique de mise en page visuelle d'Android ne fait pas exception. Elle présente un cycle de vie délicat à maîtriser et peut être lourde lors de la création de nouveaux composants graphiques.
De plus, conçues au milieu des années 2000, les APIs graphiques tenaient compte de contraintes obsolètes pour les smartphones actuels, ce qui limitait l'évolution d'Android.

Les équipes de Google sont 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 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-implémentée pour Compose.

La maturité du framework est impressionnante. Les cas d'usage courants sont documentés, facilement implémentables, tout comme les solutions pour les cas limites. Il peut toutefois manquer certaines fonctionnalités par rapport aux anciens composants (ex : sur les animations des listes ou la sûreté de typage de la navigation). Il est regrettable de devoir dépendre de versions non finales des bibliothèques de support, dont les APIs peuvent changer rapidement, pour utiliser les dernières version des composants et bénéficier des dernières fonctionnalités. Cependant, la communauté des développeurs reste enthousiaste et produit régulièrement des articles et des bibliothèques complémentaires.

NOTRE POINT DE VUE

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.



ADOPT
3/5 blips

32 KotlinX Serialization

La sérialisation d'objets est cruciale en développement logiciel pour la gestion de systèmes distribués, des services web et l'échange d'informations entre différentes plateformes.

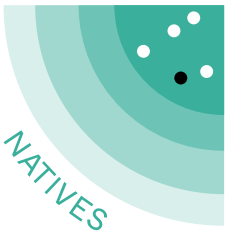
Des solutions populaires comme Gson, Moshi ou Jackson, fonctionnent bien sur la JVM et permettent de tirer parti de la reflexion. **Développé par JetBrains, KotlinX Serialization se distingue par sa simplicité, sa compatibilité multiplateforme et sa prise en charge des types Kotlin, y compris la nullabilité et les sealed class**.

KotlinX Serialization supporte d'autres formats que JSON, comme Protobuf, CBOR, Hocon et Properties. La librairie est performante, légère, sans dépendance sur la reflexion et utilise un plugin de compilation pour générer les sérialiseurs/désérialiseurs. On regrette cependant que l'adapter de KotlinXSerialization pour Retrofit2, développé par Jake Wharton, soit encore externe au repository de Retrofit, mais celui-ci vient de se stabiliser, il va donc rejoindre les autres adapter officiels.

Bien que Gson, Moshi ou Jackson soient populaires et puissants, KotlinX Serialization est performant, léger, offre une meilleure intégration avec Kotlin et est compatible multiplateforme.

NOTRE POINT DE VUE

Si vous êtes satisfait de votre solution actuelle, inutile de changer immédiatement. Cependant, si vous démarrez un nouveau projet Kotlin ou si votre projet nécessite de tirer parti de ces avantages, KotlinX Serialization est aujourd'hui la bibliothèque que BAM utilise sur tous ses nouveaux projets.



ADOPT
4/5 blips

33 SwiftUI

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

- utiliser UIKit, le framework par défaut, créé par Apple. Néanmoins, celui-ci ne propose qu'une API impérative et verbeuse, malgré l'outil Interface Builder ;
- utiliser des frameworks tiers déclaratifs, 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 dans la même tendance que Jetpack Compose. Ce framework est également requis pour certaines nouveautés ou améliorations, comme les widgets ou les animations iOS.

Il présente de nombreux avantages : d'une part, la syntaxe est claire et simple, ce qui facilite la montée en compétence ; d'autre part, l'écosystème bénéficie d'un soin particulier d'Apple, avec des outils dédiés directement intégrés à XCode. Il permet également de profiter de l'écosystème préexistant. En effet, SwiftUI est aisément interopérable avec UIKit et Combine, facilitant la transition vers ce nouveau framework.

En revanche, SwiftUI présente encore quelques lacunes. Certaines APIs ne sont pas encore accessibles sous SwiftUI et nécessitent un bridge vers UIKit. De plus, le framework étant récent, des bugs apparaissent régulièrement et doivent être contournés dans l'attente d'un correctif. Ceux-ci n'affectent généralement pas la production, mais peuvent offrir une expérience frustrante aux développeurs.

Cependant, le défaut majeur de SwiftUI est bien l'absence assumée de rétrocompatibilité avec les anciennes versions des OS. Cette stratégie pousse à abandonner le support de versions plus anciennes mais surtout limitera l'adoption du framework tant qu'utiliser des nouveautés sera synonyme de laisser des utilisateurs sur le carreau. Certaines APIs comme les lazy grids ou les previews stables que nous considérons essentielles ne sont disponibles qu'à partir d'iOS 14, ce qui nous pousse à considérer cette version d'iOS comme le minimum requis pour un projet SwiftUI.

NOTRE POINT DE VUE

SwiftUI est un framework que nous recommandons si la version minimale supportée par votre application est iOS 14, ce qui permet de couvrir 93% de la population française.

ADOPT
5/5 blips

34 Tuist

Les outils de développement iOS sont vétustes : le format "xcodeproj", pour un simple nouveau projet induit 344 lignes incompréhensibles (cela peut monter à 3000 lignes ou plus). Impossible de l'éditer à la main et donc de le review. Et à plusieurs développeurs, les conflits git sont inévitables. Il est donc impossible de l'éditer à la main et de le review. Aussi, 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. Sur un de nos projets, nous passons par exemple de 1788 lignes de code à 19 lignes. Le fichier est bien plus lisible, et les conflits disparaissent.

Tuist 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 afin de documenter le projet et propose un cache afin d'optimiser la compilation.

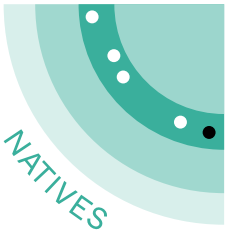
Tuist accélère le développement. En une commande qui échafaude l'architecture de fichier, on génère le squelette d'une page, d'un appel API... Tuist permet également de générer un accès typé aux assets (images, vidéos, sons, etc.) afin d'éviter un crash due à une erreur de nom de fichier.

La communauté de Tuist est très active et les développeurs sont réactifs, ce qui permet de résoudre rapidement les problèmes et de contribuer facilement.

NOTRE POINT DE VUE

La migration d'un projet existant vers Tuist est assez simple : nous avons migré un de nos plus gros projets (comprenant 1600 fichiers et 50 frameworks) en quelques jours.

Nous recommandons l'adoption de Tuist pour tous les nouveaux projets et la migration des projets existants.



TRIAL
1/5 blips

35 The Composable Architecture

SwiftUI introduit de nouvelles fonctionnalités pour l'UI par rapport à UIKit, ainsi que pour la gestion de l'état et du cycle de vie des composants grâce à son approche réactive.

Il offre des ObservableObjet et des EnvironmentObject pour externaliser le state. Ces primitives de code sont très puissants, mais nécessitent une architecture bien pensée pour tirer pleinement parti de leur potentiel.

C'est pourquoi nous nous sommes tournés vers The Composable Architecture (TCA) : **une architecture très inspirée du "one-way data flow" de Redux, qui vise à structurer le state, le rendre testable et facilement réutilisable. Son objectif est également d'améliorer l'expérience de développement en liant tous les changements de state à des actions utilisateur.**

Contrairement à des bibliothèques concurrentes comme ReSwift, TCA est conçu pour SwiftUI. Il propose également des mécanismes d'injection de dépendances intégrés qui facilitent le développement. Les créateurs de TCA ont bâti une communauté active et ont produit de nombreux tutoriels ainsi qu'une doc complète, ce qui permet de lisser la courbe d'apprentissage. Elle reste cependant volumineuse, ce qui nécessite une exploration préalable, et son accès est payant. De plus, TCA est très opiniâtre dans sa gestion du state, ce qui complique parfois son intégration avec le reste de l'écosystème iOS.

Bien que ces problèmes puissent être contournés facilement, la bibliothèque est encore récente et présente certaines lacunes :

- l'absence de prise en charge complète de la navigation, oblige à réinventer la roue et à utiliser des expressions complexes ;
- le compilateur Swift interprète difficilement ces expressions complexes, ce qui peut entraîner des lenteurs ou des erreurs de compilation.

Pour l'instant, nous utilisons TCA avec prudence dans nos projets, en attendant de voir si ce framework souffre des mêmes problèmes de scalabilité que Redux.Nous avons cependant hâte de la sortie de la première version stable, qui embarquera une meilleure gestion de la navigation ainsi.

NOTRE POINT DE VUE

Chez BAM, nous utilisons The Composable Architecture sur des projets en production et nous vous recommandons de regarder cette bibliothèque pour la gestion du state sur vos prochaines apps en SwiftUI.

TRIAL
2/5 blips

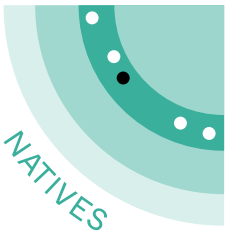
36 Hilt

L'injection de dépendances (DI) est un design pattern très utilisé en développement, qui permet de faciliter les tests, de rendre le code plus modulaire et d'améliorer la maintenabilité. **Hilt est un framework d'injection de dépendances pour Android qui a été publié par Google en 2020. Il est basé sur Dagger 2, la célèbre bibliothèque d'injection de dépendances, et fournit une mise en œuvre simplifiée de l'injection de dépendances dans les apps Android.**
L'un des problèmes clés résolu par Hilt concerne le code boilerplate nécessaire pour configurer Dagger, pour l'injection de dépendances dans les apps Android. Dagger, bien qu'il soit puissant, peut être complexe à configurer. Hilt simplifie ce processus en fournissant un ensemble d'annotations, qui peuvent être utilisées pour marquer les classes à injecter et générer automatiquement le code Dagger sous-jacent. Cela permet aux développeurs de se concentrer sur l'écriture de la logique métier plutôt que de se soucier des subtilités de Dagger. Hilt et Dagger résolvent les dépendances lors de la compilation, tandis que Koin est un localisateur de services, qui le fait là l'exécution. Cette différence a un certain impact sur les performances de l'app à l'exécution. Cependant, les bibliothèques d'injection de dépendances au moment de la compilation peuvent complexifier la création d'une architecture micro-frontend pour votre app, tandis que les localisateurs de services peuvent faciliter cette tâche.

Nous avons déjà mentionné Koin dans notre précédent Radar. Nous déconseillons de tenter de remplacer Koin par Hilt dans une codebase existante et volumineuse.

NOTRE POINT DE VUE

Lorsque vous démarrez un nouveau projet, essayez Hilt si vous souhaitez éviter les erreurs à l'exécution et suivre les conseils de Google. Sinon, optez pour Koin si vous souhaitez une bibliothèque légère, simple d'utilisation et compatible avec les projets Kotlin Multiplatform. Nous vous conseillons d'ajuster votre choix en fonction de votre équipe, de votre environnement et de vos plateformes.



TRIAL
3/5 blips

37 Koin

Le pattern d'injection de dépendance est largement utilisé pour appliquer le principe d'inversion de contrôle, lors du développement d'applications Android. En effet, cela permet de rendre le code plus modulaire, maintenable et facile à tester. Cependant, la mise en place de l'injection de dépendance peut être fastidieuse.

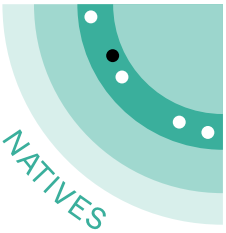
Koin répond à tous ces enjeux avec une configuration des dépendances directement dans le code via un DSL très intuitif et simple à utiliser. Cependant, il répond au même besoin que le framework recommandé par Google pour Android : Dagger-Hilt.

Les deux fonctionnent très différemment. Koin gère l'injection des dépendances au runtime, alors que Hilt se configure via les annotations standard de Java pour l'injection, génère du code et injecte les dépendances directement lors de la compilation.

Hilt sera donc plus performant au runtime et pourra détecter les erreurs dès la compilation, tandis que Koin n'impactera pas la compilation mais risque d'émettre des exceptions au runtime s'il est mal configuré. En pratique, l'impact de Koin au runtime est minime si ce n'est négligeable.

NOTRE POINT DE VUE

Maintenant que Hilt a gagné en maturité, le choix entre les deux n'est pas évident. En fonction des besoins, il est recommandé de privilégier Koin pour sa simplicité d'utilisation, son intégration parfaite au sein de l'écosystème Kotlin et sa compatibilité avec Kotlin Multiplatform, ou Hilt pour ses performances élevées au runtime, son respect des standards et la détection des erreurs à la compilation. Néanmoins, le choix final doit être adapté aux préférences de l'équipe et aux besoins spécifiques du projet.



TRIAL
4/5 blips

38 Room

Le stockage des données dans l'application revêt une grande importance.

Afin d'améliorer l'expérience utilisateur et d'éviter les temps d'attente pour les résultats de requêtes, il est recommandé de mettre en place un système de cache. Les bases de données relationnelles offrent de réels avantages pour cet usage :

- format de données structuré et prévisible ;
- normalisation des données ;
- accès plus simple ;
- propriétés avancées de cohérence (ACID) ;
- gestion des transactions...

Sur mobile, SQLite s'est imposée comme la solution pour faire des bases relationnelles. Cependant, il est fortement déconseillé d'utiliser directement cette technologie que ce soit pour des raisons de complexité ou de typage.

Il est alors courant d'utiliser un ORM (Object-Relational Mapping) : une librairie faisant l'interface entre une base de données et le reste du code.

Dans le contexte d'Android, Google a créé et intégré Room dans Android Jetpack, un ensemble de bibliothèques visant à simplifier la tâche des développeurs. Avec Room, il est possible de déclarer facilement toute la structure de la base de données et les différentes requêtes en quelques lignes de codeKotlin, avec des annotations.

Room est une librairie mature, existant depuis 2018, qui bénéficie d'une documentation complète et est facile à utiliser. Elle s'intègre facilement en Kotlin et offre la possibilité de faire des opérations réactives. Par exemple, lorsqu'une table est modifiée (ajout, modification ou suppression de lignes), Room émet automatiquement une nouvelle valeur, permettant aux composants de se mettre à jour.

Cette fonctionnalité est compatible avec les bibliothèques réactives les plus courantes, telles que :

- les Flows des coroutines ;
- les Observables de Rx ;
- ou les LiveData.

Il existe des concurrents sérieux à Room, tels que SQL Delight ou Realm, qui proposent une version compatible avec Kotlin Multiplatform, ce que Room ne propose pas.

NOTRE POINT DE VUE

Vous devriez prendre le temps de comparer Room avec ses concurrents, mais l'utilisation de Room sur votre projet nous semble sans risque, à moins que vous envisagiez un projet Kotlin Multiplatform.

TRIAL
5/5 blips

39 µ-Features Architecture

Il y a quinze ans, Adrian Cockcroft, architecte chez Netflix, a introduit 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 (ex : un appel API, du stockage, des notifications...) 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).

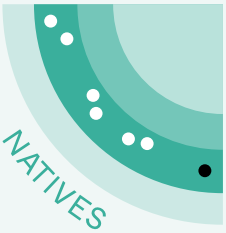
Mais un bon découpage des modules implique une bonne expertise du domaine métier. À titre d'exemple, il faut particulièrement travailler l'interface pour augmenter la cohérence et diminuer le couplage de chaque module avec les autres. Cela demande également une connaissance de l'architecture ainsi qu'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

NOTRE POINT DE VUE

Chez BAM, les micro-features sont devenues un choix de référence dans les nouveaux projets iOS. Nous avons pu tester à quel point cette architecture permettait de simplifier les évolutions ou refonte partielle d'une grosse application dans le domaine de la santé.



ASSESS
1/7 blips

40 Factory

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.

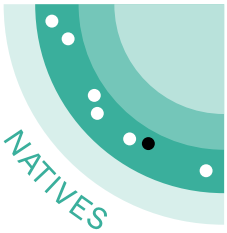
Dans la précédente édition de ce radar, nous vous conseillions d'utiliser Resolver pour l'injection de dépendances, un framework "moderne" créé par Michael Long et utilisant les Property Wrapper de Swift 5.1.

L'écosystème de l'injection de dépendances a évolué depuis, et **Michael Long a créé Factory une nouvelle librairie qui remplace Resolver. Elle est d'après l'auteur, plus rapide, plus petite et plus simple à utiliser. Mais surtout elle devient safe au compile time :** alors que Resolver s'attend à ce que les dépendances soient injectées au "runtime" et émet une erreur si ce n'est pas le cas, Factory permet de vérifier au "compile time" que toutes les dépendances sont bien injectées. C'est un gros plus pour la qualité du code, mais cela complexifie la mise en place d'une architecture en micro framework.

Récemment, l'auteur a sorti une version 2.0 de Factory qui permet de déclarer des containers d'injection de dépendance. La notion est subtile, mais elle permet de réaliser l'inversion de contrôle, non plus avec le pattern "Service Locator" comme "Resolver", mais avec le vrai pattern "Dependency Injection". Utiliser une vraie injection de dépendances par container diversifie théoriquement les cas d'usages : s'il est toujours possible de faire ce que faisaient Resolver ou Factory 1.x, d'autres cas d'usages sont possibles, en particulier pour l'injection de vue SwiftUI.

NOTRE POINT DE VUE

Il y a donc beaucoup de changements sur Resolver / Factory mais la direction que prend l'auteur est claire : rendre Factory plus adapté pour des projets en SwiftUI et plus documenté.
Le contrecoup : une API plus complexe, en particulier dans le cadre d'une architecture micro-frontend. Nous vous conseillons donc de suivre les évolutions de Factory pour vos projets en SwiftUI, mais de rester sur Resolver pour vos applications UiKit.



ASSESS
2/7 blips

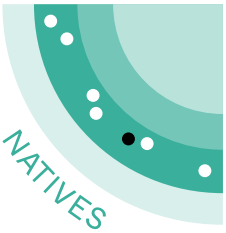
41 Glance

Anciennement, le développement de widgets sur Android excluait l'utilisation de Jetpack Compose, obligeant les développeurs à recourir à RemoteViews (une interface utilisateur légère et modifiable depuis un processus externe à l'application principale) et un code XML complexe. Ce processus rendait la création et la maintenance des widgets laborieuse et moins intuitive.

S'appuyant sur le runtime de Jetpack Compose, Glance offre une API déclarative pour faciliter la création d'interfaces utilisateur de widgets. Parmi les fonctionnalités clés de Glance figurent notamment la gestion du state et une intégration simplifiée avec d'autres bibliothèques Jetpack, telles que Jetpack DataStore pour la gestion des données. De plus, Glance permet l'interopérabilité avec RemoteViews en les enveloppant dans AndroidRemoteViews. Il permet enfin de définir comment un widget doit réagir lors du redimensionnement, grâce à l'introduction de SizeMode, qui résout une importante difficulté que l'on rencontrait sur les projets avant Glance, en proposant trois options : Single, Exact et Responsive.

NOTRE POINT DE VUE

Il convient de souligner que Jetpack Glance est toujours en version Alpha et n'est pas complètement stable. Les développeurs souhaitant utiliser cette bibliothèque devraient l'essayer sur des projets non critiques afin d'évaluer sa maturité et sa pertinence pour leurs besoins spécifiques.



ASSESS
3/7 blips

42 KMM

Les plateformes mobiles iOS et Android ont toujours été incompatibles entre elles : Elles n'utilisent ni le même langage (Kotlin vs 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 complexe 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 dédié à l'application, sur lequel il ne reste qu'à implémenter le code propre à chaque plateforme (en particulier l'UI).

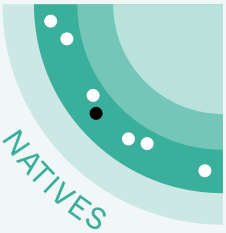
Nous avons consacré plusieurs semaines à cette technologie pour l'évaluer et avons également publié

plusieurs bibliothèques l'utilisant. Avec un peu de maîtrise on peut créer un cœur commun aux deux plateformes et consommer la même API. Si on est prêt à quelques concessions (quelques features de Swift ne sont pas supportées et il faudra faire appel à des bibliothèques externes pour l'interopérabilité des coroutines par exemple) Kotlin Multiplatform répond aux attentes.

Depuis Octobre 2022, KMM est maintenant en bêta, et se dote d'un nouveau modèle mémoire pour la partie native pour remplacer l'ancien, qui était très controversé car compliqué à prendre en main pour les développeurs habitués à Kotlin/JVM. Kotlin Multiplatform permet aux applications adressant nativement iOS et Android de partager du code entre elles. Cela peut être fait de manière graduelle, et même une application complètement écrite en KMM permet d'implémenter les API natives de ces OS si nécessaire. Cependant, l'éco-système KMM est encore incomplet et manque de stabilité, ce qui nous bloquerait pour démarrer un projet entièrement basé sur cette technologie.

NOTRE POINT DE VUE

Nous avons fait évoluer KMM de “hold” à “assess” et allons continuer d'expérimenter régulièrement avec cette technologie. Cependant, le support de Jetpack Compose pour iOS qui vient d'être annoncé ne serait-il pas l'élément manquant pour que cette technologie décolle réellement ?



ASSESS
4/7 blips

43 Media3

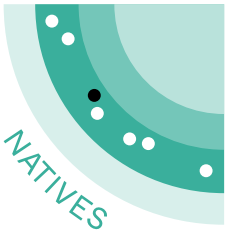
Les APIs de lecture de média sur Android telles que Jetpack Media, Jetpack Media2 et ExoPlayer, ont été développées indépendamment, avec des objectifs différents et des fonctionnalités qui se chevauchent. Les développeurs Android devaient non seulement choisir la bibliothèque à utiliser, mais aussi écrire des adaptateurs lorsque des fonctionnalités de plusieurs API étaient nécessaires.

Jetpack Media3 résout ce problème en fusionnant et en affinant les fonctionnalités communes de ces API existantes, notamment l'interface utilisateur, la gestion de la lecture et les sessions médias, en une seule API plus cohérente et facile à utiliser, tout en tirant parti des fonctionnalités améliorées et des meilleures pratiques de développement.

L'interface de lecture d'ExoPlayer a également été mise à jour, améliorée et rationalisée pour servir d'interface de lecture commune pour Media3.

NOTRE POINT DE VUE

Media3 est actuellement en version bêta, nous conseillons donc d'envisager de l'adopter pour les nouveaux projets et de préparer la migration pour les projets existants utilisant les API de gestion des médias précédentes. Chez BAM, nous avons attendu cette migration avec impatience, car nous avons éprouvé des difficultés avec les anciennes APIs et nous allons pouvoir la tester sur le prochain projet média.



ASSESS
5/7 blips

44 Paparazzi

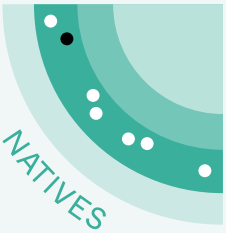
Les tests de snapshot sont de plus en plus répandus sur les projets Frontend et en particulier mobiles. Ils permettent une forme de tests unitaires sur une partie du code qui est souvent difficilement testable unitairement. C'est en particulier le cas sur Android où le code utilisant le framework est peu ou pas utilisable dans le contexte d'un test unitaire (et donc pas sur un device). Le framework Android propose des méthodes pour réaliser des tests d'UI, mais elles nécessitent de faire tourner ces tests sur des devices (réels ou émulés) ce qui consomme beaucoup de ressources en CI et est complexe à mettre en œuvre.

Paparazzi propose une solution élégante et inédite à ce dernier problème : Instancier des View Android sans nécessiter de device ou d'émulateur. En interne, Paparazzi utilise notamment une partie du code qu'Android Studio met en œuvre pour proposer des prévisualisations de Layout. On obtient des rendus bitmap pour chacune des configurations sélectionnées. Une première utilité est la prévisualisation du rendu des écrans sans avoir à tester le code sur différents devices. Mais le gros intérêt est de pouvoir facilement dérouler des tests de non-régression visuelle. Paparazzi propose un framework complet pour dérouler ce type de tests, générer les écrans de référence et détecter les écarts par rapport à celles-ci.

Nous avons intégré cette bibliothèque à plusieurs de nos projets Android et c'est vraiment le bon outil pour nos tests de non-régression visuelle (sans pour autant remplacer les tests unitaires). Cependant comme elle repose sur un détournement de l'API interne d'Android Studio elle peut être sujette à régressions et problèmes, comme ce fut le cas avec Jetpack Compose 1.2 ou Android API 33 qui n'ont pas été supportés pendant plusieurs mois par la version release de Paparazzi. Sur les écrans complexes, on peut aussi rencontrer des problèmes en cas de recomposition (par exemple avec des composants qui modifient leur état provoquant une recomposition) sur le même écran.

NOTRE POINT DE VUE

Ce genre de problème pourrait empêcher une montée de version majeure de la version d'Android ou de Jetpack Compose, et c'est pourquoi nous vous conseillons d'intégrer cette bibliothèque avec précaution.



ASSESS
6/7 blips

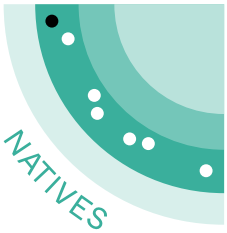
45 Refresh Versions

Que ce soit pour optimiser le temps de compilation ou pour organiser un projet de taille importante, il est maintenant très courant d'avoir des projets modulaires en Android. Cependant et jusqu'à peu, Gradle ne proposait pas de méthode standard pour centraliser la déclaration des dépendances et de leurs versions. Plusieurs pratiques co-existaient dans la communauté à ce sujet, mais aucune n'était vraiment satisfaisante pour ce qui est de faire monter les versions des dépendances et beaucoup empêchent de fonctionner les suggestions de l'IDE quant aux nouvelles versions. On se retrouve alors à naviguer sur le site de chaque dépendance ou bien Maven pour vérifier si une nouvelle version est disponible.

RefreshVersions propose une solution élégante à ce problème : vérifier sur Maven si de nouvelles versions de chaque dépendance sont disponibles et l'indiquer en commentaire dans un fichier qui centralise les versions des dépendances. Libre au développeur de sélectionner la dernière version disponible d'une dépendance, en commitant une modification de ce fichier central. refreshVersions propose également un catalogue intégré de la plupart des dépendances courantes ce qui permet de les découvrir via l'auto-complétion lorsque l'on ajoute une dépendance à un module. Saluons également l'outil de migration intégré qui permet d'adopter l'outil en quelques minutes si le projet a une structure classique. Dommage cependant que l'outil commence tout juste à supporter les fichiers Gradle Catalog qui est la nouvelle solution poussée par Gradle pour centraliser la déclaration des dépendances et ne permet pas de se baser uniquement sur celui-ci.

NOTRE POINT DE VUE

Ce projet mérite d'accroître sa notoriété et doit encore évoluer pour s'intégrer dans un maximum de projets existants. Cependant si l'outil de migration fonctionne pour votre projet, il vous fera déjà gagner un temps précieux. En phase avec la philosophie du projet (c'est au développeur de sélectionner les bibliothèques à mettre à jour), nous avons adopté cet outil sur tous nos nouveaux projets Android.



ASSESS
7/7 blips

46 Swift snapshot testing

Lorsque l'on teste un programme, il est difficile de tester plus que la logique unitaire sans faire de test end-to-end. Ces derniers sont souvent coûteux en temps et peuvent se révéler difficiles à maintenir lorsque le produit change beaucoup.

C'est avec l'objectif de trouver un juste milieu entre ces deux approches que les tests de snapshots ont été imaginés puis appliqués aux technologies web. Leur usage se répand de plus en plus et c'est justement ce que promet la librairie swift-snapshot-testing. **Elle permet** en effet **de prendre des captures d'écran ou de l'arborescence de rendu d'une application iOS native (SwiftUI ou UIKit).**

Cela permet de fournir des tests d'affichages et de non-régression efficaces à moindre frais, la librairie étant facile à installer et configurer, tandis que les tests eux-mêmes sont rapides à mettre en place. C'est d'autant plus vrai qu'il est possible de paramétrer le test pour prendre des captures de plusieurs formats différents automatiquement, ce qui est utile pour s'assurer que le rendu est bien celui auquel on s'attend.

Également, nous observons que swift-snapshot-testing n'introduit pas de perte de performance lors des phases de compilation ou de test.

En revanche, la librairie souffre de plusieurs défauts, notamment de différences de rendu entre les architectures Intel et ARM. Cela pousse à baisser la précision des tests ou à ne développer l'application que sur une seule architecture du processeur (ce qui s'applique également aux fournisseurs de CI). Il est également assez difficile de tester certaines librairies tierces en fonction de leur recours aux animations, bien que cela soit de mieux en mieux supporté.

NOTRE POINT DE VUE

Chez BAM, nous avons utilisé swift-snapshot-testing sur plusieurs projets, en lancement ou existants, avec succès. Cela nous a permis et nous permet encore de nous prémunir de régressions visuelles ou de nous assurer que nos designs correspondent aux attentes sur tous les appareils cibles. Cependant, bien qu'ils ne soient pas majeurs, ces divers problèmes nous empêchent de recommander cette librairie aveuglément.

HOLD
1/2 blips

47 JUnit 5 for Android

JUnit est le framework de test unitaire le plus répandu pour Java, et par extension pour les autres langages de la JVM, dont Kotlin.

Introduit en 2017, **JUnit 5 propose une nouvelle API plus lisible et permettant d'écrire des tests paramétrés de manière efficace.** Malheureusement cette version n'est toujours pas supportée officiellement par la plateforme Android. Des ports non-officiels existent, par exemple android-junit5. Prise en isolation, cette bibliothèque fonctionne très bien et permet de profiter de la nouvelle API sur votre projet.

Cependant la plupart des autres bibliothèques utilisables pour les tests unitaires (Jetpack testing et Paparazzi par exemple) ne supportent pas l'API de JUnit 5.

Il y a ainsi de grandes chances qu'au cours de votre projet vous ayez à utiliser ce genre de dépendances, ce qui vous forcera à utiliser JUnit 4 pour les tests qui en dépendent, entraînant une complexification de la configuration du projet.

NOTRE POINT DE VUE

Pour ces raisons évoquées, nous ne vous conseillons pas d'utiliser JUnit 5 tant que Google n'en propose pas un support officiel pour Android.

HOLD
2/2 blips

48 Texture

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

Il permet de définir des composants visuels et de les agencer avec des flexbox. Le framework est très performant et succinct par rapport à UIKit, et il est facile de le connecter avec des composants UIKit existants. Lorsque Swift UI était encore récent (arrivé sur iOS 13 et plus abouti à partir d'iOS 14), Texture était une bonne alternative pour avoir des flexbox sans sacrifier la compatibilité des devices les plus anciens. Nous avons alors choisi de l'utiliser chez BAM.

Toutefois, Texture possède plusieurs inconvénients. Le code nécessaire pour l'affichage est certes moins verbeux que UIKit, mais il reste plus long et complexe qu'avec Swift UI. La documentation disponible est très lacunaire et il peut arriver de ne pas comprendre pourquoi quelque chose ne marche pas. Enfin, l'activité de cette librairie est désormais très faible : la dernière mise à jour date de septembre 2021 et peu de sites hors la documentation officielle proposent des ressources.

Swift UI étant désormais disponible sur un grand parc d'appareils (iOS 14 et supérieurs étant utilisés par 94% des utilisateurs en mars 2023), nous vous conseillons de privilégier cette librairie graphique. Si toutefois, vous voulez assurer une forte compatibilité, nous vous recommandons de rester sur UIKit, l'effort pour passer à Texture étant trop important.

NOTRE POINT DE VUE

Si vous êtes actuellement sur un projet Texture, il n'est pas forcément nécessaire d'effectuer le changement dès aujourd'hui. C'est un compromis à trouver entre maintenir une uniformité dans le code et limiter les problèmes dus à Texture.

LE CADRAN

Transverse

13 BLIPS | 2 ADOPT | 6 TRIAL | 5 ASSESS | 0 HOLD

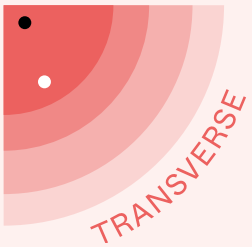
Nouveauté Changement Inchangé

Aujourd'hui, de nombreux outils comme l'IA générative ou le no code sont à la disposition des développeurs pour produire du code rapidement. Quelle que soit la technologie avec laquelle vous construisez votre produit, votre étoile du nord reste de livrer vite un produit sécurisé et sans défaut de qualité.

C'est pourquoi ce quadrant propose des indicateurs globaux de vitesse, de qualité et de sécurité, ainsi que des conseils pour permettre à vos équipes de garantir la meilleure qualité au quotidien.

Mais la performance durable d'équipe ne repose pas uniquement sur des outils et indicateurs. Nous sommes, par exemple, convaincus de l'importance de la formation de vos experts. Il est, pour cela, primordial d'encourager une prise de recul, une réelle compréhension en profondeur des outils utilisés afin d'éviter que l'expert ne devienne esclave d'une aide dont il ne maîtrise pas le fonctionnement. C'est ce qui garantit la production de livrables de qualité à une vitesse compétitive sur le long terme.

PAR **MARION VALENTIN**
Head of BAM Nantes



ADOPT
1/2 blips

49 4 Key Metrics

Aujourd'hui, le développement logiciel est le moteur des entreprises les plus performantes du monde. À 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 :

- fréquence de mise en production ;
- taux de mise en production sans erreur ;
- lead time pour qu'un code arrive en production ;
- lead time pour corriger un défaut en production.

L'étude elle-même, les KPIs ainsi que les mécanismes sous-jacents ont été décrits en profondeur dans un livre de Nicole Forsgren, Jez Humble et Gene Kim, intitulé Accelerate.

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.

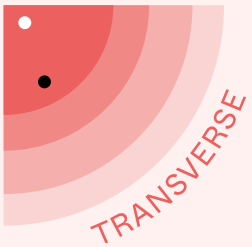
La force principale de ce framework est l'approche rigoureuse basée sur les statistiques de nombreuses entreprises qui le soutiennent et qui donne une grande légitimité pour convaincre les décideurs.

Notre application de ce framework nous a permis de constater sa puissance, comme un point de départ des changements organisationnels. Plusieurs équipes l'ont utilisé avec succès pour impliquer des stakeholders dans les discussions visant l'amélioration du processus de dév. Nous avons observé des impacts positifs sur les temps de déploiements, leurs fréquences et une implication plus importante des équipes dans le run et monitoring de leurs apps.

Le point difficile est la mise en pratique, qui nécessite souvent de challenger l'organisation en profondeur. Les KPI ne donnent pas forcément des outils pour mener ces changements, mais l'étude de cas dans le chapitre 16 de Accelerate donne une grille d'analyse qui est très utile. Ce schéma liste les pratiques à adopter par les équipes, le management et la direction pour améliorer la culture, la structure de l'organisation, l'innovation, les stratégies de déploiement et l'amélioration du flux via le Lean.

NOTRE POINT DE VUE

Notre conseil, lisez Accelerate et testez-le dans votre contexte, mettez en place les indicateurs et utilisez la partie "Transformation" comme guide pour atteindre le prochain palier.



ADOPT
2/2 blips

50 Technical-Functional Design Diagram

Plusieurs implémentations de Scrum sont tombées dans le même travers en se concentrant sur les user stories et en laissant au second plan la notion de fonctionnalité. Dans ces conditions, une équipe peut régulièrement être amenée à enchaîner des user stories, sans avoir de vision globale du sujet dans lequel elles s'insèrent. Les résultats de ce biais 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 la fonctionnalité et trouver les cas limites seuls.

Chez BAM, nous avons introduit depuis quelques années un atelier d'une heure, au lancement d'une fonctionnalité. 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 commune pour écrire les user stories.

Un schéma partagé entre tous les membres de l'équipe permet à la fois de :

- se mettre d'accord sur le scope exact de la fonctionnalité et d'en améliorer la compréhension commune ;
- faciliter la levée d'alerte en mettant en avant des zones d'ombre ;
- visualiser la complexité fonctionnelle et technique ;
- améliorer la QA et faciliter le débogage ;
- Accélérer l'onboarding fonctionnel des nouveaux membres de l'équipe.

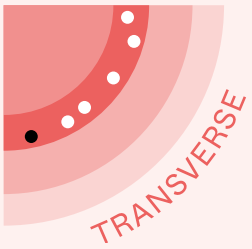
Les bénéfices sont flagrants :

- 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 ;
- un partage de connaissance entre les équipes de différents projets plus rapide.

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.

NOTRE POINT DE VUE

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



TRIAL
1/6 blips

51 100% of coverage

Pour s'assurer qu'une unité de code, telle qu'une fonction ou une classe, fonctionne comme prévu, les développeurs peuvent écrire du code qui vérifie son comportement, c'est ce qu'on appelle un test unitaire. À partir des ces tests, on peut mesurer quelle proportion du code fonctionnel est testée, c'est la couverture de test.

Écrire des tests unitaires permet d'améliorer la qualité du code d'un projet, de documenter et de faciliter les modifications du code existant. Cependant, cela peut prendre du temps, en particulier au démarrage d'un nouveau projet, ainsi que pour leur maintenance tout au long du développement.

Chez BAM, nous encourageons nos équipes d'ingénieurs à écrire des tests, au fur et à mesure des développements. Certaines équipes ont relevé le défi de tester 100% de la base de code.

Voici nos apprentissages :

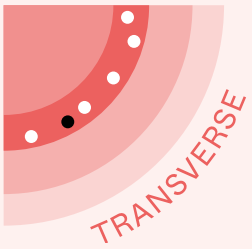
Premièrement, tester 100% du code n'est pas toujours un indicateur de qualité. Nous n'avons pas observé une réduction significative du nombre de bugs entre les projets qui testent environ 80% de leur code, et ceux qui le testent à 100%.

Ensuite, en début de projet, les équipes sont ralenties par l'écriture des tests. Mais au bout d'un mois ou deux, les tests permettent aux équipes d'itérer rapidement sur les fonctionnalités à développer et de modifier le code existant en toute confiance.
Enfin, tester 100% du code nous a ouvert des opportunités d'apprentissage :

- chaque fois qu'un membre de l'équipe développe un module, il doit se poser la question suivante : "Comment est-ce que je vais le tester ?". Le code doit être testable, ce qui incite l'équipe à mieux découper les composants, à ré-utiliser leur code et à injecter proprement les dépendances. Cela nous a également permis d'identifier et de supprimer les lignes de code inaccessibles ;
- pour tester une fonctionnalité, il est nécessaire de comprendre comment son implémentation fonctionne. En écrivant des tests, les développeurs approfondissent leurs connaissances sur les outils qu'ils utilisent ;
- en testant 100% du code, on découvre plus de cas à la marge, notamment dans les bibliothèques que l'on utilise. Ce sont des occasions idéales pour participer à l'open source. Sur nos projets, c'est grâce aux tests que plusieurs développeurs ont fait leurs premières contributions.

NOTRE POINT DE VUE

Nous vous conseillons d'essayer de tester à 100% votre code sur vos projets de moyen ou long terme. Même si cette stratégie n'est pas suffisante pour garantir la qualité, elle permet de former efficacement les équipes aux bonnes pratiques de développement.



TRIAL
2/6 blips

52 Flashlight

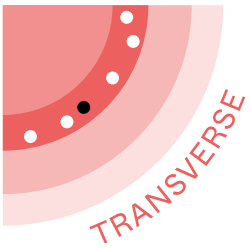
Comment savoir si votre app a une bonne performance ? Cette question est complexe pour plusieurs raisons, notamment car diverses métriques entrent en jeu : FPS, CPU, TTI, RAM...

- les mesures de performance ne sont pas déterministes ;
- les mesures sont coûteuses en temps et fastidieuses.

Flashlight se veut être une réponse à cette question (disclaimer : c'est un outil que nous développons). Flashlight donne un score de performance à votre app. Un peu comme Lighthouse mais pour les apps Android (iOS n'est pas supporté). Avec Flashlight, aucun setup dans l'app n'est requis pour mesurer sa performance. Ainsi, même les apps de production sont supportées, et ce, quelle que soit leur technologie (natif, React Native, Flutter...)

En revanche, Flashlight ne peut pas explorer l'app seul à ce stade. Par défaut, il délivrera facilement un score qui concerne le démarrage de l'app. Mais pour aller plus loin, vous aurez besoin de lui partager vos propres tests e2e. Flashlight va lancer ces tests plusieurs fois, agréger différentes métriques-clés et moyenner les résultats dans un rapport de performance.

Si vous n'avez pas de tests e2e, tirer profit de Flashlight peut s'avérer plus compliqué, mais nous recommandons l'usage de Maestro pour en mettre en place rapidement. Le cœur de Flashlight est open source. Ainsi, vous pouvez l'utiliser sur votre propre device, mais une version cloud qui tourne sur un device bas de gamme Android est disponible en beta, pour simuler la réalité du marché. En quelques clics, Flashlight Cloud donne un score de performance à votre app et peut être intégré à une CI pour récupérer ce score régulièrement. La durée des tests peut être longue (plus de 10 min), ainsi nous ne conseillons pas de l'intégrer sur chaque pull request, mais plutôt de vérifier l'évolution du score sur le démarrage et quelques parcours critiques chaque semaine ou avant chaque release. Nous vous invitons également à utiliser Flashlight pour évaluer l'impact de décisions technologiques majeures, ou comme indicateur pour vous aider dans une tâche d'amélioration de performances.



TRIAL
3/6 blips

53 Github copilot

Cet assistant de développement permet d'écrire du code plus vite, en proposant une autocompletion avec des bouts de code prêts à être utilisés.
En se basant sur OpenAI Codex, il analyse les commentaires et le code pour suggérer une implémentation.

Nous observons dernièrement une forte émulation autour de l'IA générative, notamment avec ChatGPT. Copilot fait partie de cette vague. L'utilisation de Copilot est très efficace dans certains cas, notamment :

- l'implémentatio d'algorithmes simples (ex : division, boucle, itération, tri) ;
- les tâches répétitives (ex : la création des serializers ou appels API) ;
- Les recommandations des interfaces ou modèles typiques (ex : Country, Person) ;
- la proposition des test cases standards dans les tests unitaires.

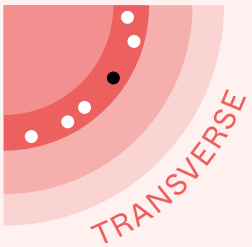
Depuis son apparition, nous évaluons ses impacts sur le quotidien des développeurset tirons plusieurs apprentissages. Ce produit assureune forte rétention des utilisateurs. Parmi les développeurs interrogés, 80% ont attribué une note NPS de 9 ou plus sur 10. Parmi les retours obtenus, on relève un gain de temps sur les tâches récurrentes. 72% des interrogés ont attribué la note 5/5 à la question : "Trouves-tu que copilot t'aide à écrire plus vite ?". On note également 3 retombées positives sur l'apprentissage :

- Copilot facilite la productivité dans un environnement méconnu ;
- il favorise la découverte de nouvelles solutions, bénéfiques aux personnes déjà expérimentées pour leur progression ;
- il agit comme un premier lecteur de code, évaluant le naming grâce aux suggestions générées.

Le ralentissement de la formation des profils intermédiaires est observé car il est plus facile d'obtenir un code fonctionnel sans une réelle compréhension. Malgré cela, nous croyons aux impacts positifs de Copilot sur la productivité. Les risques peuvent être gérés efficacement avec l'aide d'un Tech Leader.

NOTRE POINT DE VUE

Nous vous recommandons d'explorer l'utilisation de Copilot dans votre organisation. Il est important de noter que ce secteur dynamique compte plusieurs concurrents, comme Tabnine, qui méritent également d'être évalués.



TRIAL
4/6 blips

54 Maestro

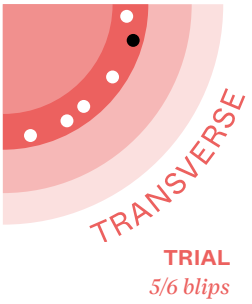
Nous avons mentionné Appium dans notre précédent Tech Radar, comme framework de tests e2e. Maestro se présente comme une nouvelle solution alternative qui tire profit des apprentissages de ses prédécesseurs. Et contrairement à une simple promesse, Maestro garde les avantages majeurs d'Appium sans ses inconvénients. En effet, contrairement à Appium, la documentation de Maestro est excellente, ce qui permet de créer un test e2e assez poussé en moins de 30 minutes. L'API est bien pensée et fournit de base les fonctionnalités essentielles à l'écriture de tests e2e (scroll, click, attendre une apparition d'élément). De plus, Maestro fonctionne (à l'instar d'Appium) en mode boîte noire. En d'autres termes, vous pouvez lancer un script de tests Maestro sur n'importe quelle app, sans aucune setup préalable. Cela inclut votre release candidate ou même votre app de production depuis les stores.

`maestro` dispose également d'une version `cloud`, qui permet d'avoir rapidement et simplement une CI avec tests, enregistrements d'écrans et logs

Maestro est encore jeune, et nous avons parfois rencontré quelques bugs dans certaines versions, ainsi que des lenteurs sur iOS. Maestro est par ailleurs moins facilement extensible qu'Appium pour rajouter des comportements qui sortent de l'ordinaire (comme lancer une commande `adb` ou simuler un scroll ultra-rapide).

NOTRE POINT DE VUE

Maestro pourrait rapidement devenir le standard du marché car l'équipe ajoute constamment des nouvelles fonctionnalités qui révolutionnent l'écriture de tests. Notre nouveauté préférée ? `maestro studio` qui permet quasiment d'écrire les tests à notre place ! Nous vous recommandons donc son utilisation si vous n'avez pas déjà de tests e2e mis en place. C'est d'ailleurs la solution recommandée par Flashlight pour effectuer des tests e2e de performance.



TRIAL
5/6 blips

55 Right First Time

Notre démarche de post-mortems QRQC nous a permis de constater que les bugs sont détectés bien trop tard. Les problèmes simples à détecter sont adressés dans le processus de validation et la QA, mais les défauts formateurs sont souvent découverts après quelques mois ou années. Cela limite les opportunités d'apprentissage et d'amélioration.

Nous avons été inspirés par le livre The Toyota Way of Dantotsu Radical Quality Improvement de Sadao Nomura. Il propose une classification des défauts selon le stade de leur détection allant de A (détecté pendant une tâche) à D (détecté par un utilisateur en production).

Alors que nous ne focalisions notre approche QRQC que sur les défauts de type D, la démarche Dantotsu propose de réagir sur les défauts détectés à chaque étape : plus il est détecté tard dans le flux, plus il devient coûteux de le corriger et d'empêcher la mauvaise pratique de se diffuser dans le code.

Nous avons adopté cette classification et expérimenté une démarche d'analyse des défauts de type A, que nous avons nommé "Right First Time". Elle se compose de 3 étapes clés :

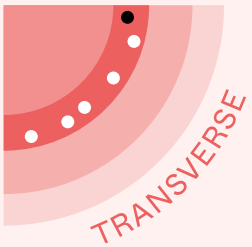
- préparer la stratégie technique avant de coder ;
- écrire tout le code, préférablement en TDD ;
- lancer le simulateur pour tester la fonctionnalité une fois la stratégie technique implémentée.

Si le code ne fonctionne pas du premier coup, il s'agit d'un défaut de type A. Nous notons ce défaut et le nombre d'essais requis pour valider les besoins fonctionnels attendus. À la fin du ticket, la personne qui l'a implémenté identifie les apprentissages à en tirer (ex : en analysant les hypothèses erronées de la stratégie initiale). Les premiers résultats sont encourageants :

- les projets présentent moins de bugs ;
- l'équipe progresse plus vite ;
- elle délivre les fonctionnalités plus vite en limitant le context switching.

La principale limite constatée est l'extrême rigueur requise pour appliquer cette méthode, ce qui complique son application quotidienne par les équipes. Nous étudions comment la simplifier et quels supports créer pour faciliter son adoption.

Malgré ces freins, nous vous encourageons à mener des tests chez vous sur des défauts de type A et observer les discussions et apprentissages que cela génère. Pour en savoir plus sur cette méthode, vous pouvez lire le livre de Sadao Nomura et regarder la conférence de Fabrice Bernhard.



TRIAL
6/6 blips

56 Rive

Créer des animations uniquement avec du code représente un défi pour les développeurs mobiles, en raison de la complexité liée à la synchronisation de nombreux éléments et à la gestion d'un important volume de code. La variété des plateformes et frameworks complique également la création d'animations cohérentes et réutilisables. De plus, la plupart des solutions existantes ne permettent pas d'interagir directement avec les animations via le code, rendant impossible les réactions aux interactions de l'utilisateur et aux éventuels changements d'état de l'application.

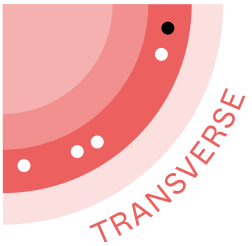
Rive est une solution qui répond à ces besoins et simplifie l'intégration d'animations pour les développeurs, compatible avec la plupart des frameworks et plateformes. De plus, l'éditeur propose une interface utilisateur intuitive et supporte l'import d'animations provenant d'autres logiciels tels que Lottie.

Les animations créées avec Rive peuvent réagir aux clics, aux mouvements ou aux changements d'état en fonction des données reçues, offrant ainsi une expérience utilisateur dynamique et immersive. Un autre avantage de Rive est l'optimisation de la taille des animations, qui peuvent être 10 fois plus légères qu'un fichier Lottie, garantissant des performances améliorées pour les applications mobiles.

L'outil Rive se positionne comme un compétiteur solide face aux autres solutions de création d'animations grâce à ses meilleures performances et APIs plus développées. Le moteur de rendu et les lecteurs étant open source, ils offrent une flexibilité et adaptabilité accrues pour répondre aux besoins spécifiques des projets.

NOTRE POINT DE VUE

Malgré certains aspects financiers et techniques liés à l'utilisation de Rive, tels que l'abonnement de 14$ par utilisateur par mois et la compatibilité avec iOS 14 minimum pour les applications natives, nous vous recommandons d'utiliser Rive pour créer des animations interactives de haute qualité.



ASSESS
1/5 blips

57 Co-writing software with ChatGPT

Les IA génératives ont connu une adoption croissante ces derniers mois.
En tête du peloton : ChatGPT.
Il existe de nombreux cas d'utilisation, les plus répandus sur internet sont les suivants : co-création d'articles marketing, écriture et synthèse automatique d'emails ou encore création de code. ChatGPT n'est pas un outil de code à proprement parler, il y a des modèles spécialement conçus pour cela :

A titre d'exemple, OpenAI Codex qui est derrière GitHub Copilot. De nombreuses personnes ont néanmoins réussi à obtenir des résultats très intéressants.

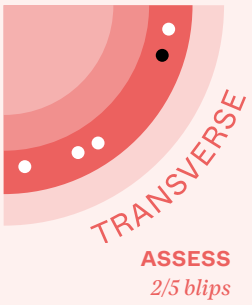
Pour citer quelques cas : un plugin Chrome qui permet d'afficher ChatGPT ou un bot de trading codé en Python. Tous les deux codés grâce à ChatGPT.

Les réussites partagées sont néanmoins sujettes au biais de sélection : nous voyons les quelques résultats impressionnants, mais nous ne voyons pas forcément tous les échecs. Dans nos tests, nous avons observé que plusieurs solutions ne sont tout simplement pas fonctionnelles, ou alors qu'elles contiennent des bugs subtils. Toute solution donnée par ChatGPT doit donc être relue et validée attentivement. Sans parler de la pertinence de la réponse, étant donné que le contexte de la codebase actuelle n'est pas forcément facile à inclure. L'amélioration de la productivité est donc limitée à des cas d'usage bien précis. Les cas qui s'y prêtent bien sont soit des solutions standards dans une technologie (formulaire de login, image centrée) soit des questions haut niveau sur les pratiques adoptées par la communauté.

Nous pensons que ces problèmes seront à terme résolus car l'écosystème évolue rapidement. Certaines solutions pourraient par exemple composer ChatGPT avec de l'IA spécialisée dans le code ou ajouter des systèmes de vérification automatiques. L'utilisation des outils tels que ChatGPT est donc sûrement une tendance à observer attentivement.

NOTRE POINT DE VUE

Entre-temps, nous vous conseillons de chercher à renforcer vos compétences en Prompt Engineering et fine-tuning, les deux compétences clés pour profiter pleinement de cette vague de l'IA.



ASSESS
2/5 blips

58 MASVS 1.5

En 2022, on enregistrait 6.6 milliards de souscriptions aux réseaux de téléphonies mobiles. Sur cette même période, un tiers des entreprises ont subi des temps d'arrêt importants ou des pertes de données en raison d'une compromission liée à un appareil mobile. Face à ce constat, il est primordial pour les développeurs d'applications mobiles d'intégrer la sécurisation de leurs apps dès la phase de conception.

L'an dernier, nous avons présenté CVSS, une norme évaluant la gravité des vulnérabilités sur une échelle de 0 à 10. Toutefois, cette norme ne fournit pas d'indications sur les éléments à rechercher. Sans une solide expérience en matière de sécurité, il est difficile de savoir par où commencer.

Le Mobile Application Security Verification Standard (MASVS) d'OWASP est une norme de sécurité pour les applications mobiles. Sa version 1.5 est séparée en 2 niveaux, chacun couvrant des contrôles de sécurité de plus en plus avancés :

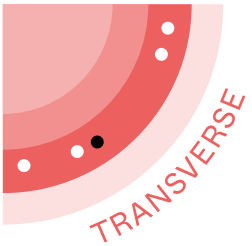
- Niveau L1 : Sécurité Standard ;
- Niveau L2 : Défense en profondeur destinée aux apps manipulant des données hautement sensibles.

Ces niveaux peuvent être complétés par un ensemble de contrôles (le niveau R) visant à renforcer la résilience face à la rétro-ingénierie, la manipulation et le modage de votre application. On obtient ainsi 4 niveaux de sécurisation : MASVS-L1, MASVS-L1+R, MASVS-L2 et MASVS-L2+R.

Ils regroupent au total 84 points de contrôle de sécurité touchant différents domaines, comme le stockage des données, l'authentification et les communications réseau. Pour chaque point de contrôle, des guides et des outils sont proposés pour explorer et corriger les éventuelles failles de sécurité. Actuellement en cours de déploiement, la version 2.0 simplifie les points de contrôle et aligne la définition des niveaux de sécurité avec le format OSCAL.

NOTRE POINT DE VUE

Nous avons introduit MASVS v1.5 par le biais d'un projet où la sécurité est un enjeu majeur. Cette solution permet notamment de mieux collaborer avec les auditeurs sécurité. Cet essai nous a convaincu de la pertinence du standard et nous sommes en train de le déployer à plus grande échelle afin de renforcer la sécurité de l'ensemble de nos applications.



ASSESS
3/5 blips

59 No-code mobile

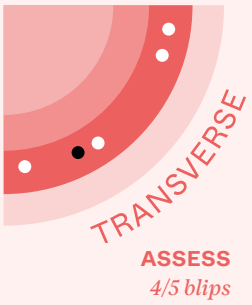
Il y a encore quelques années, no-code mobile était une fausse piste. Très souvent basées sur du web, les solutions laissaient à désirer en termes de qualité et de performance. Elles étaient appropriées surtout pour des POC ou des outils internes à faibles enjeux où l'expérience utilisateur n'était pas la priorité.

L'écosystème a récemment connu une belle avancée avec de nouveaux outils basés sur Flutter (comme FlutterFlow) ou React Native (comme draftbit).

Même s'ils ne sont pas encore très matures, nous avons déjà eu des expériences très positives avec, comparé à la génération précédente. Les interfaces offrent une flexibilité suffisante pour les cas d'usage standards et la possibilité d'exporter le code permet d'éviter le lock-in.

NOTRE POINT DE VUE

Avoir des alternatives à des solutions basées sur des WebViews permet de mieux répondre aux attentes des utilisateurs. Nous vous recommandons de les regarder et essayer pour des projets à petit budget, cela vaut le détour.



ASSESS
4/5 blips

60 Ship, show, ask

Dans le processus de développement d'une fonctionnalité, la relecture du code est une étape clé pour garantir sa qualité et sa conformité aux normes de l'équipe.

Toutefois, ajouter une étape de contrôle pour chaque relecture bloque le développeur et nuit à la productivité. Pour éviter cela, **Rouan Wilsenach propose une approche intitulée "Ship, Show and Ask". Elle consiste pour chaque changement à choisir parmi ces trois options :**

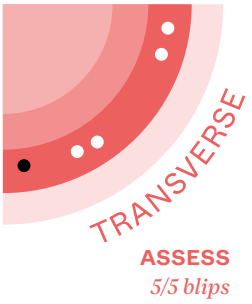
- Ship : créer une branche poussée sans relecture pour les changements mineurs et standardisés ;
- Show : créer une branche avec une pull-request et merger sans attendre la relecture pour les changements qui ne sont pas critiques, mais nécessitent une relecture ultérieure ; ainsi que pour les changements faits en équipe (duo ou ensemble programming) ;
- Ask : créer une branche avec une pull-request et attendre la relecture avant de merger pour les changements critiques ou incertains qui nécessitent l'avis d'un collègue.

Récemment, nous avons testé cette approche sur trois projets avec des équipes expérimentées. Elle a permis de réduire le temps de relecture des pull-requests, de favoriser une réflexion plus approfondie sur le sens de chaque pull-request en redonnant la responsabilité au développeur et de chercher le consensus de la solution avant de coder plutôt qu'après. Les demandes de relecture Ask systématiques sont descendues à près de 50% depuis cette mise en place. Nous n'avons pas constaté de baisse de la qualité du code.

Cette expérimentation est limitée par le manque de recul de par son utilisation récente chez BAM, la qualité déjà élevée du code et de la conception technique ainsi que la maturité de l'outillage (CI, tests unitaires) mis en place.

NOTRE POINT DE VUE

La relecture de code est une étape intéressante pour aligner l'équipe sur le code à fournir. En revanche nous vous encourageons à réfléchir à la qualité de façon globale et sélectionner les outils (code review, tests, intégration continue, conception, pair programming, ensemble programming, etc.) les plus aptes à garantir la qualité de chaque changement selon la maturité de vos équipes et de vos process.



61 TestPlan

Dans le cycle de développement d'une feature, l'étape de relecture (ou code review) est une étape essentielle. Elle permet de s'assurer que le code produit est de bonne qualité, et qu'il est conforme aux standards de l'équipe.

Une dérive courante est de pousser cette standardisation à l'extrême, et d'instaurer une liste toujours plus longue de règles à respecter. Les développeurs finiront par ne plus lire les règles, cocher machinalement les cases et ne plus réfléchir aux impacts de leur code.
Pour éviter cette dérive, nous avons testé de remplacer la liste de règles par une section "test plan" dans laquelle les développeurs doivent décrire les tests qu'ils ont effectués pour valider leur code. Cette section est inspirée des pratiques de "test plan" popularisées par Meta pour contribuer aux projets open source.

Comme la section "test plan" est libre, les développeurs peuvent décrire ce qui fait le plus de sens dans le contexte du projet et du ticket : en pratique, nous avons vu par exemple des études de query plan PostgreSQL pour assurer la performance des index, des tests d'accessibilité d'une page mobile à un lecteur d'écran ou encore des tests de performance de la page web. Ceci crée des discussions intéressantes entre le développeur et le reviewer lors de l'étape de code review d'une part, et permet d'autre part d'apprendre plus lors d'un post-mortem sur un bug (QRQC).

Mais, comme la documentation, la section "test plan" demande un effort et une rigueur supplémentaires. Au fil du temps, la qualité de cette section est devenue de plus en plus variable sur nos projets.

NOTRE POINT DE VUE

Nous pensons néanmoins que cette section est un bon compromis entre la liste de règles obligatoires et l'absence de “check qualité” documenté dans la pull-request. Nous recommandons donc d'essayer cette approche sur vos projets, et de l'adapter à vos besoins.

Autres technos adoptées chez BAM

Dans notre Tech Radar, nous avons choisi de mettre en lumière des technologies innovantes ainsi que des choix audacieux. À la suite de notre première édition, publiée en 2022, vous avez été plusieurs à nous demander nos recommandations pour lancer un nouveau projet. Dans cette lignée, nous avons le plaisir de vous présenter une liste de technologies qui sont les plus couramment utilisées sur un projet from scratch.

STACK REACT NATIVE

- Expo + EAS
- Typescript + ESLint + Prettier
- Jest + React Native Testing Library
- React Query / Apollo
- Zod
- Jotai / Zustand
- Yarn3
- React Navigation
- Luxon
- Reanimated
- Emotion
- React Native MMKV
- React Hook Form

STACK FLUTTER

- Riverpod
- GoRouter
- Melos
- Mocktail
- Custom_lint
- Golden Tests
- Rive
- graphql_flutter
- Dio
- Open API Generator
- fast_immutable_collection

STACK NATIVE IOS

- Swift
- SwiftUI
- TCA
- Combine
- swift-snapshot-testing
- AnyCodable
- XCTest Dynamic Overlay

STACK NATIVE ANDROID

- Kotlin
- Flow + Coroutines
- Jetpack Compose
- Clean Architecture + MVI
- Koin
- Retrofit + KotlinX Serialization
- Jetpack DataStore
- Room
- KtLint
- MockK
- MockWebServer

Le comité de contenu

CYRIL BONACCINI
Staff Engineer

GUILLAUME DIALLO-BOISGARD
Head of Flutter

ANTOINE DOUBOVETZKY
Head of React Native

MATTHIEU GICQUEL
Tech Lead

NICOLAS HAAN
Tech Lead

CHARLOTTE ISAMBERT
Tech Lead

MAREK KALNIK
CTO & Co-founder

MO KHAZALI
Head of Mobile Theodo UK

ARTHUR LEVOYER
Head of Native

ALEXANDRE MOUREAUX
App Performance Expert

MATTHIEU PERNELLE
Tech Lead

TYCHO TATITSCHEFF
Principal Technologist

MARION VALENTIN
Head of BAM Nantes

UN GRAND MERCI À NOS CONTRIBUTEURS DE L'OMBRE

Nicolas Acart / *Développeur* • Dennis Bordet / *Développeur* • Julien Calixte / *Engineering Manager* • Thomas Coumau / *Développeur* • Louis Dachet / *Tech Lead* • Lucas Delsol / *Développeur* • Rémi Deronzier / *Développeur* • Maxence Leroy / *Architecte Développeur* • Valentin Marquis / *Développeur* • Louis Prud'homme / *Développeur* • Maxime Rougieux / *Développeur* • Clément Taboulot / *Architecte Développeur* • Pierre Zimmermann / *Architecte Développeur*

Quelques chiffres clés

Experts mobiles du Groupe Theodo depuis 9 ans, nous accompagnons nos clients de la phase de discovery et de conception produit à la phase de design et de développement en React Native, Flutter ou en technologies natives.

+250 Apps mobiles : BAM a développé une véritable expertise mobile pluri-sectorielle en travaillant aux côtés de grands groupes, PMEs et startups, parmi lesquels : The Fork, Renault, Bouygues, Colas, ekWateur, Biogen, TF1, Carrefour, Compagnie des Alpes, Urgo...

+130 Nous sommes aujourd'hui plus de 130 BAMers, répartis au sein des pôles business développement, produit, design et tech, tous convaincus que le smartphone n'est que le premier support technologique qui bouleverse nos usages et nos modes de vie. C'est pourquoi nous construisons une entreprise capable de relever les défis d'aujourd'hui et de demain.

CONCEPTION & RÉALISATION

Chantal Khin & Yohan Nicolleau / *Direction Artistique*

Anaïs Silvano / *Marketing Lead*

Ninon Narcy / *Coach Agile*

MOBILE

BAM

Part of Theodo Group