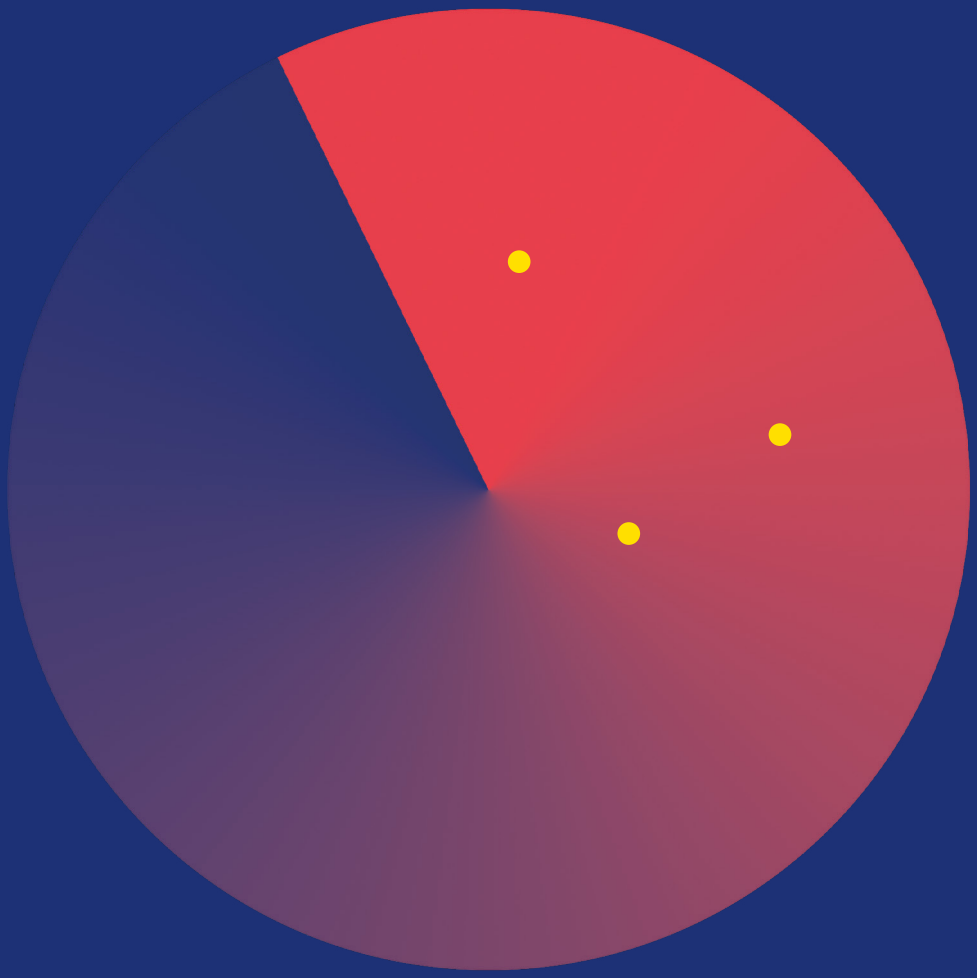
Tech Radar

#3

NOVEMBRE 2024



theodo. Apps

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 troisième édition !

Introduction

Bien que l'écosystème mobile ait atteint une certaine maturité, il demeure extrêmement dynamique. La multitude de technologies, de frameworks et de SDKs confronte chaque équipe mobile à des choix complexes. À cela s'ajoute un support multiplateforme de plus en plus étendu (web, TV, desktop) et stable, ce qui encourage le partage de code entre différentes applications front. Depuis des années, nous construisons une base de connaissances que nous avons décidé d'ouvrir, afin que la communauté puisse bénéficier de nos expériences et 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 de Theodo Apps. 13 membres du comité de contenu se sont penchés sur les technologies et techniques qui constituent les 53 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 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 un "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

Notre Tech Radar 8

Blips 9

Quelle alternative choisir ? 10

Les cadrans

— **React Native** 12

— **Flutter** 26

— **Natif** 42

— **Transverse** 66

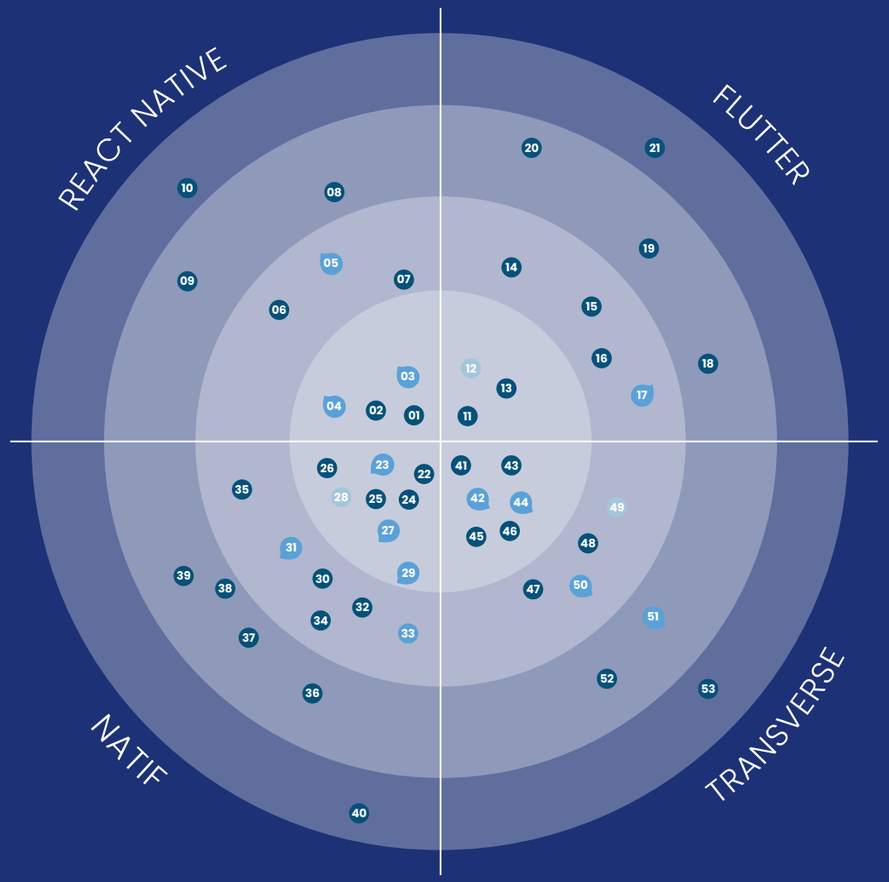
Autres technos adoptées 85

Contributeurs 87

À propos de Theodo Apps 88

Notre 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, Flutter et les technologies natives.



Blips

REACT NATIVE

1. Migrer vers Expo
2. Solito
3. Suspense for data fetching
4. React Native Web
5. Expo Router
6. React Native Keyboard Controller
7. React TV Space Navigation
8. React Server Components for mobile
9. react-native-unistyles
10. react-native-svg as a default

FLUTTER

11. checks
12. Riverpod
13. sliver_tools
14. flutter_map
15. mmkv
16. Reactive Forms
17. Shorebird
18. Dart Macros
19. Patrol
20. Signals
21. isar

NATIF

22. Koin
23. Gradle Version Catalog
24. KSP
25. SF Symbols
26. Swift 6 Migration
27. The Composable Architecture
28. Tuist 4
29. µ-Features Architecture
30. Compose Stability Configuration File
31. Kotlin Multiplatform
32. Factory
33. Room
34. Swift Dependencies
35. Typed Errors en Kotlin avec Arrow
36. Amper
37. Compose Multiplatform (CMP)
38. Swift Perception
39. Swift Testing
40. SwiftUI Hot Reload

TRANSVERSE

41. Feature Toggles
42. Flashlight
43. Générer son client API
44. Rive
45. Supabase
46. UI Snapshot testing
47. Identifying Defect Detection Stage
48. Maestro
49. MASVS 2.1
50. Ship Show Ask
51. Weekly Engineering Review
52. Fleet
53. PWA-first

Quelle alternative choisir ?

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

Lorsque nous avons lancé Theodo Apps (anciennement BAM), en 2014, le choix technologique était très risqué. Nous étions convaincus que le multiplateforme était l'avenir du mobile et avons choisi une 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 ?

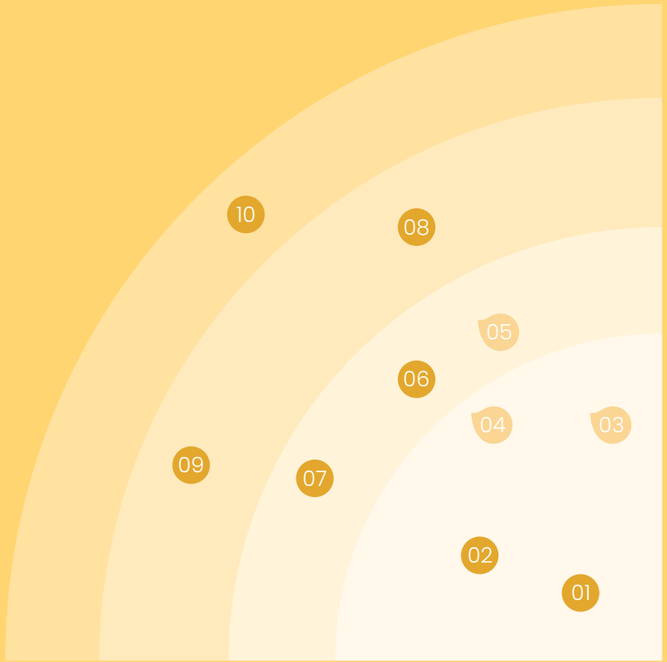
Cet échange nous permet d'évaluer le projet par rapport aux 3 technologies et d'émettre une recommandation, plus ou moins forte en fonction des contraintes.

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 les 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.

CADRAN

React Native



10 BLIPS | 4 ADOPT | 3 TRIAL | 2 ASSESS | 1 HOLD

Nouveau Changement Pas de changement

Dans cette édition du Tech Radar, nous mettons en avant deux tendances majeures dans l'écosystème React Native : **l'adoption croissante d'Expo** et l'essor des **applications universelles**.

Expo, désormais recommandé par Meta pour le développement des nouvelles applications React Native, s'impose comme une solution incontournable. Pour de nombreux projets, la migration vers Expo représente un gain significatif en termes de **qualité**, de **productivité** et de **maintenance**, rendant l'usage de ce framework essentiel pour tout projet React Native.

Parallèlement, les applications universelles connaissent un véritable essor grâce à des outils comme Solito et Expo Router. Ces solutions permettent de **partager une base de code entre les plateformes web et mobile**, réduisant ainsi les coûts et la complexité du développement. Le partage de code peut également s'étendre aux plateformes **TV et Desktop** qui bénéficient d'un écosystème React Native de plus en plus riche.

Avec l'évolution continue de ces outils, le slogan de React Native, "Learn once, write anywhere", n'a jamais été aussi pertinent, facilitant le développement d'applications performantes et multiplateformes.

PAR **CYRIL BONACCINI**
Staff Engineer

ADOPT
1/10 blips

1 Migrer vers Expo

La maintenance de la partie "Native" des applications React Native peut être chronophage. Les mises à jour de versions majeures de React Native prennent souvent plusieurs jours et introduisent des bugs en raison d'incompatibilités avec les autres packages natifs utilisés. La mise à jour de ces derniers est également un processus fastidieux.

Expo est la solution pour simplifier le développement et la maintenance des applications React Native. Le système de prebuild d'Expo, appelé Continuous Native Generation (CNG), automatise la génération du code natif à partir de fichiers de configuration. Les config plugins intègrent des modifications natives spécifiques, tandis que l'API des Expo Modules simplifie la création de modules natifs. Expo Application Services (EAS) permet de builder et déployer des applications mobiles facilement. De plus, Expo Updates offre une solution de mise à jour over-the-air (OTA) pour envoyer des mises à jour sans passer par les stores.

Meta recommande officiellement l'utilisation d'Expo pour les nouvelles applications React Native, comme indiqué sur le site de React Native : "To build a new app with React Native, we recommend a framework like Expo". De plus, avec la fermeture programmée d'AppCenter par Microsoft le 31 mars 2025, **Expo Updates reste la seule solution viable pour les mises à jour OTA**.

De nombreux projets n'ont pas démarré avec Expo car la solution n'a pas toujours été aussi complète et stable qu'aujourd'hui. Heureusement, **il est possible de migrer progressivement vers Expo**. D'abord en configurant une nouvelle application Expo avec l'ensemble des dépendances de l'application existante pour que le code JavaScript puisse être exécuté. Puis en mettant à jour les processus de déploiement pour utiliser EAS et Expo Updates. Enfin, en migrant vers les modules Expo pour bénéficier de modules de meilleure qualité, mieux maintenus et mis à jour automatiquement lors de la montée de version du SDK Expo.

Chez Theodo, nous avons observé **des gains significatifs en termes de temps de développement et de maintenance** en utilisant Expo dans nos projets. La simplification des mises à jour, l'amélioration de la productivité et la robustesse de l'écosystème Expo en font une solution incontournable pour les projets React Native. Malgré quelques défis liés à la migration et aux limitations du support de certaines plateformes, l'**adoption d'Expo est une décision stratégique bénéfique à long terme.**

NOTRE POINT DE VUE

Nous recommandons fortement de planifier la migration de votre projet vers Expo pour simplifier les mises à jour, augmenter la productivité et améliorer la maintenance des projets React Native.

2 Solito

ADOPT
2/10 blips

Les applications universelles gagnent en popularité, permettant aux développeurs de cibler iOS, Android et le web avec une base de code commune. Cette tendance pose la question de la meilleure technologie pour gérer cette approche multi-plateforme. À Theodo nous utilisons en fonction des projets deux frameworks : Expo Router et Solito.

Comme l'indique sa documentation, Solito est à la fois :

- une **librairie** qui sert de **pont entre React Navigation et Next.js** ;
- une **CLI** permettant de créer un projet avec un **monorepo** contenant une application Expo et une application Next.js, où Solito est utilisé pour la navigation dans le code partagé.

Le principal avantage de Solito par rapport à Expo Router est qu'il permet d'utiliser les fonctionnalités avancées de Next.js. Cela permet d'activer le **rendu côté serveur (SSR)**, d'utiliser des **server components**, d'améliorer la **gestion des polices et des images**, ainsi que d'offrir un meilleur **support pour l'internationalisation**.

Pour des applications pour lesquelles la performance et le SEO sont critiques, c'est un avantage incontestable.
Le monorepo permet en outre d'avoir une **séparation claire entre le code spécifique au web ou au mobile et le code partagé**, ce qui est particulièrement intéressant si certaines features de votre application ne sont que sur le web ou que sur mobile. Un autre use case intéressant est celui où l'on a déjà une application Expo et une application NextJS et que l'on souhaite partager du code entre ces applications. On peut alors regrouper les applications sous un monorepo et partager les composants mobile progressivement.

Toutefois, l'utilisation d'un monorepo avec une application Expo et une application NextJS comporte quelques défis par rapport à une unique application sous Expo Router. Bien que cela puisse introduire une certaine complexité, comme la **nécessité de maintenir deux systèmes de navigation distincts**, cette approche permet également une flexibilité accrue. La création de pages reste rapide, mais pour assurer une expérience cohérente entre l'application mobile et le site web, il est important de bien synchroniser la structure des pages entre les deux routeurs. Solito facilite grandement la navigation dans le code partagé, même si l'**absence de typage et d'autocomplétion pour les URLs** nécessite une attention particulière pour éviter les erreurs.

NOTRE POINT DE VUE

Solito est une excellente option pour les projets d'application universelle pour qui le SEO et la performance sur le web sont critiques. Pour des applications qui n'auraient pas besoin de ces features, il sera plus facile et moins coûteux d'utiliser Expo Router.

3 Suspense
for data fetching

ADOPT
3/10 blips

Nous avons déjà parlé de Suspense lors du dernier Tech Radar, et nous avions déjà recommandé son activation.

L'intérêt majeur de **Suspense avec une solution de data fetching** comme React Query est la **simplification des composants**, dont nous n'avons plus besoin de préciser les états de chargement et le traitement des erreurs.

C'est un **gain en maintenabilité, mais aussi un gain en UX** puisque cela encourage le placement de loaders et d'encarts d'erreur.

Il y a toutefois un aspect sur lequel il faut être vigilant lors du passage à Suspense. Si plusieurs hooks de data fetching utilisant Suspense sont mis bout-à-bout, les tâches asynchrones s'exécuteront les unes après les autres, même si elles sont indépendantes, ce qu'on nomme communément des appels en waterfall.

La solution que nous préconisons est de **répartir au maximum les hooks dans les composants les plus bas possibles de l'arbre React**, pour que le composant étant suspendu soit le plus petit possible. Les composants étant suspendus en parallèle, cela permet de bien avoir des appels asynchrones en parallèle. Pour améliorer les délais avant interactivité (TTI), il est également possible de **pre-fetcher les requêtes**.

Récemment, l'équipe React a publié un changement cassant totalement React Query[1] avec Suspense (tous les appels devenaient waterfall, même en appliquant l'approche ci-dessus).

Ce qu'il faut retenir, c'est que le changement a été annulé, ce qui a entraîné un report de la sortie de React 19. Cela nous rend donc particulièrement confiants dans le fait que cette façon d'utiliser Suspense est effectivement une **pratique reconnue et validée par l'équipe React**.

NOTRE POINT DE VUE

Avec tous les avantages en UX et en maintenabilité apportés par Suspense pour du data fetching (avec React Query ou d'autres librairies le supportant), nous recommandons fortement le passage à Suspense. Nous recommandons également de migrer les projets existants.

[1] https://tkdodo.eu/blog/react-19-and-suspense-a-drama-in-3-acts

4 React Native Web

ADOPT
4/10 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.

Nous avons pu partager entre 75% et 95% du code entre des applications React web et des applications React Native en utilisant React Native Web sur plusieurs projets, et notre retour d'expérience est très positif. Les variations de la quantité de code partagé dépendent du volume du fonctionnalités spécifiques web et/ou mobile, notamment la mise en page.

Dans notre précédent radar, nous remontions des problèmes de performances et d'accessibilité. Beaucoup de travail a été fait depuis, à la fois sur react-native et react-native-web mais aussi avec des librairies de la communauté comme Tamagui et Expo Router.

React Native Web n'est pas une solution magique pour résoudre tous les problèmes de partage de code, les points restants sont davantage liés à la manière de travailler qu'à la technologie elle-même :

- 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.

NOTRE POINT DE VUE

Nous utilisons maintenant React Native Web sur tous nos projets nécessitant le partage de code entre le web et le mobile. Cette adoption confirme notre confiance dans la technologie, que nous recommandons désormais comme une solution fiable pour développer des applications sur ces plateformes.

5 Expo Router

TRIAL
5/10 blips

Le développement d'applications web nécessite de prendre en compte des contraintes qui sont en général peu connues des développeurs mobiles : taille de bundle, accès direct à n'importe quelle page via une URL, rendu côté serveur…

Pour un framework, offrir les meilleures performances vis-à-vis de ces contraintes implique de gérer directement le processus de bundling, la navigation, et éventuellement le data-fetching. C'est ce que font par exemple Next.js, Astro, et Remix.

Expo-router est un framework de navigation universel pour les apps React Native qui, grâce à son intégration aux autres outils expo, promet une expérience optimisée à la fois sur iOS/Android et sur le Web. Concrètement, expo-router propose du "**filesystem-based routing**" (créer un fichier dans le dossier `app` définit une nouvelle route) et est une surcouche à React Navigation. La plateforme Web est officiellement supportée avec la gestion des urls, du bundle splitting pour optimiser les performances, ou encore la possibilité d'inclure du CSS. Il s'agit du principal avantage d'expo-router.

À l'usage, **la jeunesse d'Expo Router se fait encore sentir**. Par exemple, le typage de la navigation est expérimental et incomplet (il ne couvre pas les search params), et l'API, en apparence simple, complique l'implémentation de certains scénarios qui ne posaient pas de problème aux développeurs maîtrisant React Navigation. Le choix d'un dossier `app` dont tous les fichiers deviennent des routes oblige à maintenir 2 arbres parallèles dans la codebase, alors que d'autres frameworks reviennent en arrière sur cette contrainte voire sur l'idée même de "filesystem-based routing".

NOTRE POINT DE VUE

Nous avons décidé de mettre Expo Router **pour les universal apps** en Trial car, il s'agit du meilleur choix pour déployer sur le Web une App React Native. Les performances ne sont néanmoins pas encore "best-in-class" (pas de server-side-rendering, pas de tree-shaking à ce jour), donc dans certains scénarios, une surcouche, par exemple avec Next.js et Solito, reste nécessaire.

6 Keyboard Controller

TRIAL
6/10 blips

Presque toutes les applications mobiles permettent à l'utilisateur d'entrer du texte et sont donc amenées à manipuler le clavier virtuel d'iOS et Android. Même dans les applications les plus populaires, des bugs ou une expérience utilisateur sub-optimale sont souvent présents autour de cette gestion du clavier.

En effet, il s'agit d'un problème plus complexe qu'il n'y parait. Et dans le cas de React Native, les différences entre les plateformes ne nous facilitent pas la tâche.

Le package react-native lui-même fournit des **API limitées** pour faire face à ce problème, un développeur a donc décidé de s'y attaquer et a publié React Native Keyboard Controller.

React Native Keyboard Controller offre une API unifiée entre iOS et Android qui inclut toutes les fonctionnalités nécessaires : ouverture/fermeture du clavier avec animation de l'UI (intégré avec reanimated), gestion des champs de texte, placement de boutons fixes au-dessus du clavier…

React Native Keyboard Controller est une bibliothèque récente et peut présenter des instabilités, mais il s'agit déjà de **la solution la plus complète dans son domaine**, d'où sa position dans notre cadran Trial.

NOTRE POINT DE VUE

Nous avons commencé à utiliser cette bibliothèque sur nos projets et encourageons les autres développeurs à faire de même.

7 React TV

Space Navigation

TRIAL
7/10 blips

L'un des défis majeurs lors du développement d'une **application pour TV** est la **gestion du focus à la télécommande**. Les solutions ne sont pas uniformes entre plateformes. React Native tvOS propose une implémentation, mais elle n'est pas totalement prédictible entre tvOS et AndroidTV, et n'existe pas encore pour le web. Cela complique la création d'applications universelles avec React Native.

Chez Theodo, nous avons développé React TV Space Navigation, une librairie qui réimplémente la navigation dans React sans utiliser le focus natif, **assurant un comportement uniforme sur toutes les plateformes**. Elle offre une API "react-friendly" avec des composants purement déclaratifs et inclut des optimisations comme la virtualisation et le défilement CSS pour **améliorer les performances**, notamment sur les appareils moins puissants.

Toutefois, contourner le focus natif a des limitations : **le support de l'accessibilité est incomplet**, et certaines fonctionnalités spécifiques, comme le "ploc" sur Android, sont absentes. Malgré ces inconvénients, nous avons déployé avec succès une application de streaming multiplateforme importante utilisant cette librairie. Cette application fait preuve d'une grande fluidité à l'utilisation.

Nous suivons l'évolution de `react-native-tvos`, qui pourrait un jour rendre notre librairie obsolète. Avec un bon découpage de composants, il est facile de revenir à la gestion par défaut de `react-native-tvos`, mais cela supprimerait le support du web, qui ne bénéficie d'aucune gestion de focus intégrée.

NOTRE POINT DE VUE

Pour l'instant, nous recommandons d'essayer notre librairie sur vos projets TV multiplateformes en ayant en tête les limites de cette solution qui vient avec certains compromis d'accessibilité et de feeling natif.

8 React Server

Components for mobile

ASSESS
8/10 blips

La "server-driven UI" consiste à envoyer depuis le serveur une description précise de ce qui doit être affiché à l'écran, plutôt que des données brutes. Certaines applications font appel à cette approche pour répondre à des contraintes comme la nécessité d'itérer rapidement, une forte personnalisation de l'UI, ou les limitations de taille des apps.

L'équipe React a annoncé en mars 2024 un **mécanisme officiel de server-driven UI, les "React Server Components"** (RSC). Ces composants peuvent être "composés" avec les "client components". React, et le framework utilisé, se charge de gérer les interactions entre les deux mondes (créer les bons bundle javascript, réconcilier l'arbre de composants côté client, etc). L'application mobile a alors pour seul rôle d'afficher l'UI transmise.

Les server components s'exécutant sur un serveur, ils peuvent accéder directement à des secrets, à des bases de données, ou encore au système de fichiers. Cela promet de radicalement simplifier le développement de certaines fonctionnalités. De plus, les allers-retours client-serveur sont évités, ce qui améliore les performances.

À ce jour, **aucune intégration des Server Components dans un framework React Native** n'existe, mais Expo travaille sur le sujet. Une implémentation React Native des Server Components devra faire face à plusieurs défis :

- il faudra s'assurer que les composants natifs générés côté serveur sont bien disponibles côté natif, sous peine de **plantage** ;
- les applications mobiles doivent souvent fonctionner, au moins en partie, **hors-ligne** ;
- la composition "transparente" des composants client et serveur est en fait limitée par de subtiles règles qui définissent quel composant peut importer et/ou recevoir en prop quel autre composant (cette limitation s'applique aussi sur le web).

NOTRE POINT DE VUE

Nous vous invitons à surveiller comment l'écosystème des Server Components en React Native évolue pour pouvoir les utiliser dès que possible, dans les cas où c'est pertinent.

9 Unistyles

ASSESS
9/10 blips

Beaucoup de librairies de style ont émergé pour répondre à la popularité croissante des applications universelles. Parmi ces solutions, on retrouve Tamagui, Nativewind et Unistyles. Avec le web sont apparues de nouvelles problématiques de performance et de server-side rendering (SSR), complexifiant les solutions et rendant leur comparaison plus difficile.

Unistyles propose une API proche de celle de StyleSheet, mais avec des fonctionnalités supplémentaires : thème accessible dans le style, **variants**, **styles dynamiques**, **breakpoints**, accès aux insets et taille de l'écran. Le cœur de la bibliothèque est écrit en C++, offrant **des performances proches de StyleSheet**.

L'avantage de Unistyles est la simplicité de son API. La définition d'un thème personnalisé est simple, car la structure n'est pas imposée, contrairement à Tamagui. La **proximité avec StyleSheet** facilite la prise en main pour les développeurs en React Native, tandis que ceux habitués à Tailwind seraient plus familiers avec Nativewind.

La bibliothèque est compatible sur de nombreuses plateformes : web, Windows, macOS, visionOS, et TV. Cependant, **le support pour le SSR n'est pas encore complet**, car les styles responsives n'utilisent pas de media queries. La version 3 devrait intégrer un compiler, comme Nativewind ou Tamagui, pour **générer le CSS au build time**.

Unistyles est récent mais connaît un développement très actif. Le projet bénéficie de l'engagement de son contributeur principal, Jacek Pudysz, ce qui lui permet d'avancer rapidement. Cependant, cette dépendance à une seule personne peut susciter des questions quant à sa pérennité.

NOTRE POINT DE VUE

En résumé, Unistyles est une solution extrêmement prometteuse pour des applications universelles. Facile d'utilisation, elle offre un grand nombre de fonctionnalités pratiques. Elle manque cependant encore de maturité, notamment pour le support web, qui n'est pas encore au niveau de celui de Nativewind. Ces lacunes devraient toutefois être comblées avec la sortie de la V3.

10 react-native-svg as a default

HOLD
10/10 blip

Un SVG est un format d'image vectoriel où l'image est dessinée dynamiquement. Ce format a des propriétés intéressantes : il n'est jamais flou et peut hériter de variations de couleurs ou de formes. Cependant, dessiner dynamiquement a un coût en performance, surtout si le moteur de rendu n'est pas adapté ou si l'image est complexe.

La librairie `react-native-svg` est souvent utilisée pour dessiner des SVG, en recréant chaque forme SVG en vue React, ce qui permet de les **modifier ou les animer dynamiquement**. Cependant, cette transformation peut être lente, particulièrement pour les icônes, qui sont souvent nombreuses sur un même écran et ralentissent le rendu.

Il existe des solutions plus adaptées pour les icônes ou les SVG statiques. Pour les icônes, on peut utiliser des polices d'icônes (icon fonts). Cela signifie convertir les SVG en lettres dans une police. La contrainte de cette solution est d'avoir un SVG mono-couleur, une caractéristique typique des icônes standards. Une méthode couramment utilisée est de recourir à IcoMoon, en combinaison avec vector-icons d'Expo.

Pour d'autres illustrations, **`expo-image`** est une bonne alternative. Cette librairie gère les SVG et les affiche de manière plus performante. Mais il est alors impossible d'animer le SVG et de personnaliser ses couleurs (à moins d'appliquer une teinte sur tout le conteneur SVG, mais cela modifie toute l'image).

Ces alternatives couvrent une grande partie des besoins en SVG, car il est rare de devoir modifier partiellement la couleur des SVG.

NOTRE POINT DE VUE

Nous recommandons de ne pas utiliser react-native-svg par défaut pour tous les SVG. Il est préférable d'adopter des solutions comme les icon fonts pour les icônes, ou expo-image pour les SVG statiques afin d'améliorer les performances. Toutefois, il reste pertinent d'utiliser react-native-svg dans le cas de SVG dynamiques.

CADRAN

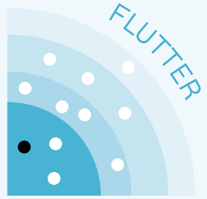
Flutter

11 BLIPS | 3 ADOPT | 4 TRIAL | 3 ASSESS | 1 HOLD

Nouveau Changement Pas de changement

Pour cette troisième édition de notre radar, ce cadran présente la nouvelle génération de technologies Flutter. Sur des problématiques déjà adressées par des solutions parfois bien ancrées dans l'écosystème telles que **`flutter_test`**, les state managers ou le stockage **`hive`**, Flutter prend un nouveau souffle. De nouvelles solutions pour afficher des cartes performantes, maîtriser la gestion des formulaires ou persister les données, émergent et améliorent l'expérience de développement et la robustesse des applications. Ce cadran propose une vue d'ensemble de ces petites évolutions aux grands impacts.

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



11 checks

ADOPT
1/11 blips

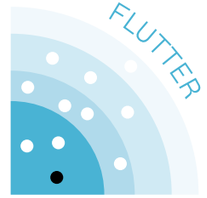
Dans les tests unitaires en dart, la solution de base pour vérifier qu'une variable a bien la valeur attendue est l'utilisation de la fonction `expect` de `flutter_test` qui prend une variable et un matcher. Le problème principal de cette fonction est le **typage dynamique de ces arguments**. Si le type de la variable ne correspond pas à celui du matcher, **une erreur remonte seulement lors de l'exécution** du test sans aucune indication au developpeur pendant l'écriture de test.

Le package `checks` expose plusieurs méthodes pour vérifier la valeur et le type de variable dans la fonction équivalente, check, et propose un support pour l'égalité profonde des collections. Les méthodes disponibles sont automatiquement filtrées selon le type de variable dans la fonction.

Par ailleurs, checks **améliore l'expérience de développement** en proposant, dans les IDEs modernes, une auto-complétion filtrée directement sur les méthodes proposées. Toutes les méthodes de test peuvent être appelées à la chaîne, une **approche déclarative** qui favorise la clarté et la lisibilité du code. La librairie propose aussi une possibilité de s'attendre à des valeurs de `Futures` ou des `Streams`. Enfin, **la possibilité de personnaliser** les tests efface les limitations en permettant de personnaliser son utilisation en combinant des tests existants ou en créant des logiques entièrement propre à son contexte.

NOTRE POINT DE VUE

Bien que peu connue, la librairie `checks` offre des avantages significatifs qui rendent **l'expérience de test plus explicite et lisible**. C'est pourquoi nous en faisons notre solution par défaut dans les projets Flutter et recommandons vivement son adoption.



12 Riverpod

ADOPT
2/11 blips

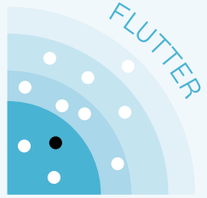
En Flutter, la gestion du state global se fait via les `InheritedWidget` et `ChangeNotifier`, mais ces APIs présentent plusieurs défauts, comme la verbosité, la complexité, la difficulté à tester ou l'impossibilité de créer plusieurs states d'un même type.

Riverpod est une **librairie de caching réactif**, qui permet de résoudre ces problèmes. Inspirée de `react-query`, elle propose de servir la donnée via des providers, déclarés en dehors du cycle de vie des widgets, et pouvant automatiquement rebuild les widgets qui les écoutent. Les providers peuvent **servir de la donnée asynchrone** venant par exemple d'un call API ou une base de donnée locale, **gèrent les erreurs** et le **cache** et permettent de facilement définir d'autres fonctionnalités telles que le debounce ou le pull-to-refresh.

Riverpod est aussi **compile-safe**, propose une API déclarative, est activement maintenue, et est soutenue par une vaste communauté (6k stars sur github, 98% de popularité sur pub.dev). Depuis l'année dernière, nous avons pu expérimenter ses outils de génération de code, qui permettent aux hot reloads de mettre à jour un provider et de **réduire encore la verbosité et la complexité de l'API**.

NOTRE POINT DE VUE

Nous recommandons d'utiliser Riverpod qui nous accompagne depuis maintenant 3 ans sur nos projets Flutter de toutes envergures. L' annonce de **riverpod 3**, qui devrait permettre de définir et réutiliser des providers ayant en paramètres des types génériques, ne fait que renforcer notre enthousiasme.



13 sliver_tools

ADOPT
3/11 blips

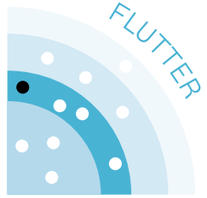
En Flutter, les slivers sont un type de widget qui s'intègre dans les vues scrollables et réagissent au scroll pour permettre de confectionner des écrans scrollables animés et complexes. Si les slivers ne présentent pas de difficulté particulière dans leur utilisation, ils sont cependant des widgets bas niveau et leur écriture est complexe.

Si le framework propose un certain nombre de widgets slivers de très haut niveau tel que le `SliverAppBar` ou la `SliverList` il peut être **difficile de personnaliser ses vues scrollables** en dehors des classiques proposés.
La librairie `sliver_tools` propose une **collection de widgets slivers** prêt à l'emploi qui enrichissent ceux présents nativement dans le framework en apportant un niveau de flexibilité intermédiaire entre les widgets haut niveau et les `RenderObjects` bas niveau.

Parmi les plus utilisés, on retrouve `MultiSliver` qui permet de combiner plusieurs slivers en un seul afin d'améliorer la qualité de code en découpant les responsabilités ainsi que le `SliverPinnedHeader` qui permet de créer des éléments scrollable qui s'accrochent en haut de la vue scrollable pour les maintenir en visibilité et donner un effet de navigation plaisant.

NOTRE POINT DE VUE

Notre utilisation de `sliver_tools` sur les projets est très concluante et n'a pas remonté de limitations. Cette boite à outil fait désormais partie de notre stack standard chez Theodo pour nous permettre de donner vie aux vues scrollables originales imaginées en collaboration avec nos designers.



14 flutter_map

TRIAL
4/11 blips

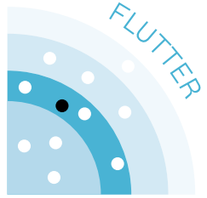
La **cartographie** dans les applications mobiles est un sujet qui peut s'avérer complexe en raison des performances et des limitations imposées par certaines librairies pour afficher divers éléments.
Traditionnellement, les développeurs se sont tournés vers des solutions comme Mapbox et Google Maps, qui utilisent un moteur graphique en C++ pour rendre les cartes.
Ces solutions sont robustes, mais elles **ne s'intègrent pas toujours de manière fluide avec Flutter**, en particulier pour personnaliser les éléments à afficher sur la carte.

La librairie `flutter_map` a été entièrement écrite en Dart avec **une API déclarative et composable** pour les éléments UI, associée à une **approche impérative pour le contrôle manuel,** tel que les animations.
Cette approche permet aux développeurs de profiter pleinement des avantages de Flutter, notamment la possibilité d'intégrer facilement des widgets sur la carte.
Elle bénéficie également d'un **écosystème varié d'extensions open source**.
Nous avons utilisé `flutter_map` pour créer des cartes raster très poussées en terme de personnalisation, et nous avons été impressionnés par **sa facilité d'utilisation et ses performances pour les cartes raster**.

`flutter_map` présente cependant des **limitations importantes**. L'extension pour les cartes vectorielles souffre de **problèmes de performances majeurs**, le rendant impraticable pour ce type de cartes. Les transitions entre les niveaux de zoom ne sont pas aussi fluides que celles des SDK comme Mapbox ou Google Maps. Cette limitation nous a poussés à privilégier des **solutions alternatives pour les projets nécessitant des cartes vectorielles**. Pour résoudre ces problèmes, la communauté travaille activement sur ces problèmes avec une solution utilisant les dernières avancées de Flutter comme Impeller ou `flutter_gpu`.

NOTRE POINT DE VUE

Nous vous recommandons d'essayer `flutter_map` pour intégrer des cartes raster à vos projets Flutter, en raison de sa simplicité d'utilisation et de ses performances. Nous vous invitons cependant à rester ouverts aux autres solutions disponibles, notamment dans les cas d'utilisation de cartes vectorielles.



15 mmkv

TRIAL
5/11 blips

Il existe plusieurs solutions de bases de données locales en Flutter, notamment shared_preferences, Isar, Hive, ObjectBox, SQLite, etc. Chacune de ces solutions a ses propres avantages et inconvénients en termes de performance, de facilité d'utilisation et de fonctionnalités. Dans ce contexte, nous avons intégré MMKV pour Flutter sur l'un de nos projets. MMKV est une **bibliothèque de stockage clé/valeur performante et facile à utiliser**, développée par Tencent. Utilisée dans l'application WeChat, MMKV est conçue pour offrir des performances optimales grâce à l'utilisation de `mmap` et de `protobuf`, permettant de synchroniser la mémoire avec les fichiers et d'encoder/décoder les valeurs efficacement.

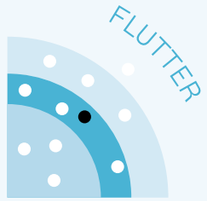
L'utilisation de Dart FFI pour les opérations de lecture et d'écriture synchrones permet de tirer pleinement profit de ces performances en Flutter. Nous avons mesuré que l'ouverture d'une base de données MMKV est **beaucoup plus économe en ressources CPU** que d'autres solutions, comme par exemple Isar, qui peut bloquer un thread pendant plusieurs centaines de millisecondes. L'API de MMKV est **claire et minimaliste**, ce qui simplifie son intégration et son utilisation dans les projets Flutter. Depuis la migration vers une architecture de plugin fédérée, l'expérience de développement avec MMKV s'est nettement améliorée. De plus, MMKV supporte le **chiffrement des données**, offrant ainsi une couche supplémentaire de sécurité.

Cependant, MMKV pour Flutter présente quelques limitations. Actuellement, **seules les plateformes iOS et Android sont supportées**, bien que MMKV en lui-même soit disponible sur iOS, Android, Linux, macOS et Windows. La prise en charge future du web pourrait s'avérer complexe à mettre en œuvre. De plus, une mise à jour mineure récente a supprimé le support pour les architectures Android ARM7 et x86, impactant environ 2% de nos utilisateurs en production.

Malgré ces défis, nous trouvons que MMKV est une solution très intéressante. Elle est simple et extrêmement performante, développée par Tencent, ce qui lui confère une grande fiabilité, et une maintenance à long terme. Bien que très populaire dans les communautés React Native et native, MMKV reste encore méconnue dans l'écosystème Flutter, probablement en raison de la présence de bases de données locales synchrones déjà établies.

NOTRE POINT DE VUE

Nous recommandons de tester MMKV dans vos projets Flutter. Cependant, il est essentiel de réaliser une **évaluation approfondie des plateformes et des appareils** utilisés par vos utilisateurs afin d'assurer une intégration réussie et une compatibilité avec les exigences spécifiques de votre projet.



16 Reactive Forms

TRIAL
6/11 blips

La gestion des formulaires et la validation des entrées utilisateur sont des aspects cruciaux mais complexes du développement d'applications. Reactive Forms, une bibliothèque pour Flutter, propose **une gestion des formulaires basée sur le modèle**, inspirée par Angular. Chez Theodo, nous l'utilisons pour les projets nécessitant des formulaires, comme ceux de connexion, d'inscription ou de paiement.

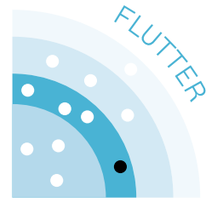
Reactive Forms se distingue par son **riche écosystème de validateurs prédéfinis, asynchrones et personnalisables**. Cette flexibilité facilite la gestion des règles de validation complexes et spécifiques à chaque projet. Cependant, la prise en main peut être difficile et le code requis pour définir un formulaire est **parfois verbeux**. Le système de typage pourrait également être amélioré. Une extension utilisant la génération de code est en cours de développement pour simplifier cette fonctionnalité.

Reactive Forms s'intègre bien avec les outils de gestion d'état comme Riverpod ou Bloc, permettant de réagir efficacement aux changements d'état du formulaire. Il permet aussi de **tester unitairement chaque règle de validation**, garantissant qu'aucune régression n'est introduite lors des mises à jour. Cela **améliore la productivité et la maintenabilité des applications**.

La documentation et le support communautaire de Reactive Forms sont excellents, avec une communauté active et des mises à jour régulières. Le package compte 458 étoiles sur GitHub et 839 likes sur Pub, et l'écosystème de validateurs créés par la communauté est un atout majeur.

NOTRE POINT DE VUE

Nous recommandons d'essayer Reactive Forms pour vos projets Flutter en raison de **ses capacités de validation robustes**, de sa flexibilité et de son intégration fluide avec les outils de gestion d'état. Toutefois, étant donné la complexité initiale et la verbosité du code, il convient de le faire **avec précaution**.



17 Shorebird

TRIAL
7/11 blips

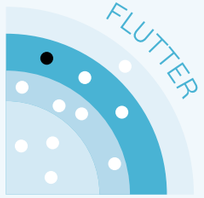
Les délais de mise à jour constituent un défi majeur en développement mobile, car chaque version doit être validée par les stores, retardant ainsi la disponibilité des correctifs urgents pour les utilisateurs. De plus, il faut attendre que chaque utilisateur mette à jour l'application. Shorebird, annoncé début 2024 par Éric Seidel, créateur de Flutter, est une solution open-source pour **déployer des mises à jour d'applications Flutter over-the-air** (OTA) sans passer par les stores. Cette approche permet de déployer des mises à jour mineures directement via leurs serveurs, **accélérant ainsi le cycle de développement et de déploiement**.

L'intégration de Shorebird dans un projet Flutter est simple mais présente certains inconvénients. Le coût peut être un frein pour certaines équipes, et la commande shorebird patch est plus lente que flutter build, ce qui **peut ralentir le processus de mise à jour**, surtout dans un environnement de QA actif plusieurs fois par jour. Actuellement, Shorebird **supporte uniquement iOS et Android**, excluant les applications desktop. De plus, l'application doit être redémarrée pour exécuter le code mis à jour, ce qui peut affecter l'expérience utilisateur.

Malgré ces inconvénients, Shorebird offre une **fonctionnalité précieuse** en étant la **seule solution de mise à jour OTA pour Flutter**. Cet outil répond à un besoin crucial en développement mobile, et respecte les règles des stores en ne mettant à jour que du code interprété. Bien que la CLI de Shorebird ait récemment atteint la version 1, elle présente encore quelques instabilités, mais ces problèmes sont rapidement résolus, l'**équipe de développement étant attentive aux retours de la communauté**.

NOTRE POINT DE VUE

Shorebird montre **des résultats prometteurs** et a le potentiel de transformer la gestion des mises à jour dans les applications Flutter. Nous recommandons de tester cette technologie pour évaluer son impact sur vos cycles de développement et de déploiement.



18 Dart Macros

ASSESS
8/11 blip

Lors de nos développements, nous nous appuyons sur des outils de génération de code tels que **`freezed`**, **`json_serializable`** et **`build_runner`**. Ces outils, bien qu'essentiels pour adresser des problématiques récurrentes telles que la désérialisation, l'immutabilité ou l'égalité profonde, introduisent des inefficacités et nuisent à l'expérience développeur en **imposant des générations de code manuelles fréquentes**. Ils tendent également à encombrer nos projets avec des fichiers générés.

Les macros Dart, actuellement en beta, promettent de transformer cette dynamique en intégrant **la métaprogrammation directement dans le compilateur**. Cette méthode permet aux développeurs de générer du code à mesure qu'ils écrivent. Elles **améliorent ainsi la productivité en réduisant les générations répétitives** et en maintenant une base de code plus propre.

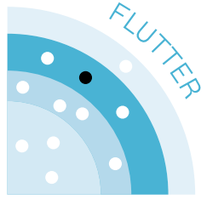
Les macros pourraient cependant compromettre la simplicité et la lisibilité de Dart, qui sont essentielles pour sa facilité d'apprentissage. Dart est réputé pour être un langage "ennuyeux" dans le bon sens du terme : prévisible et stable. L'introduction des macros, par contraste, **introduit une dimension de complexité** qui n'est pas traditionnellement associée à Dart.

Si les macros semblent menacer cette transparence, le système d'augmentation proposé permet aux développeurs de **visualiser le code augmenté d'un simple clic** dans leur environnement de développement, maintenant ainsi une compréhension claire du code exécuté.

Notons que les macros seront surtout utilisées par les mainteneurs de librairies, comme **`freezed`** ou **`json_serializable`**, et non au quotidien par tous les développeurs Flutter. Nous serons attentifs à la manière dont les mainteneurs et la communauté accueilleront et mettront en œuvre cette nouvelle fonctionnalité.

NOTRE POINT DE VUE

Nous sommes optimistes sur le potentiel des macros Dart pour simplifier et améliorer la génération de code dans nos applications Flutter. Bien que les macros soient **actuellement en phase expérimentale**, l'équipe Dart prévoit sur sa roadmap une version stable pour début 2025.



19 Patrol

ASSESS
9/11 blip

Chez Theodo, nous pensons que les tests automatiques sont une des meilleures manières de prévenir les régressions fonctionnelles ou visuelles sur les applications mobiles. Cependant, les solutions de test existantes en Flutter, telles que les tests unitaires et les tests de widgets, ne couvrent pas toujours de manière exhaustive les scénarios de bout en bout (E2E).

Patrol[1] est une librairie Flutter qui **simplifie l'écriture et l'exécution des tests E2E**. Elle intègre une API qui **permet d'interagir très simplement avec les fonctionnalités natives** de l'appareil de test telles que les permissions, les notifications ou les paramètres. La librairie patrol_finders[2] qui est désormais indépendante de Patrol, offre en outre du sucre syntaxique rendant l'écriture des tests plus intuitive et moins sujette aux erreurs.

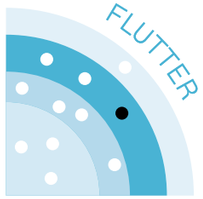
Chez Theodo, nous abordons les tests automatiques en utilisant plutôt des tests unitaires, ainsi que des tests d'UI adaptatif (également connus sous le nom de golden tests) grâce à des outils comme adaptive_test[3]. Ils garantissent une couverture fonctionnelle complète tout en vérifiant visuellement l'interface utilisateur.

Pour les tests E2E, **Patrol offre une API extensive**, mais ne fonctionnera qu'avec des applications Flutter. Pour ce besoin-là notre choix actuel se porte sur Maestro, une technologie qui a fait ses preuves et dont nous pouvons mutualiser les apprentissages avec les équipes React Native et natives iOS/Android. Nous trouvons cependant que **Patrol résout le problème des tests E2E en Flutter de manière intéressante et intuitive**.

NOTRE POINT DE VUE

Pour les développeurs Flutter qui veulent améliorer leurs tests E2E, Patrol mérite d'être exploré, en particulier si vous voulez utiliser un outil qui s'intègrera parfaitement à vos tests Flutter existants. Même si notre choix actuel se porte sur Maestro, nous continuerons de suivre l'évolution de Patrol.

[1] https://pub.dev/packages/patrol
[2] https://pub.dev/packages/patrol_finders
[3] https://pub.dev/packages/adaptive_test



20 Signals

ASSESS
10/11 blip

La gestion des états globaux d'une application est cruciale pour sa maintenabilité et ses performances. Diverses solutions ont été proposées pour Flutter, et une nouvelle option mérite notre attention : Signals.

Signals est une bibliothèque innovante pour Flutter, **facilitant la gestion des états avec des signaux réactifs**. Inspirée par les concepts de réactivité de l'écosystème JavaScript, tel que preact, elle propose une réactivité fine où chaque signal représente une valeur encapsulée dans une coque réactive. Les signaux peuvent être des états simples ou des calculs dérivés d'autres états, formant un graphe acyclique de dépendances.

Signals offre plusieurs avantages clés :

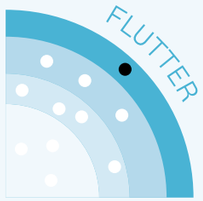
- **Réactivité fine** : les signaux suivent automatiquement les dépendances et les libèrent lorsqu'elles ne sont plus nécessaires.
- **Évaluation "lazy"** : les signaux ne sont calculés que lorsqu'ils sont lus, optimisant ainsi les performances.
- **API flexible** : permet une composition multiple des signaux avec une surface API réduite.
- **Rendu chirurgical** : seules les parties nécessaires de l'arbre des widgets sont mises à jour, améliorant les performances.
- **Compatibilité Dart Native** : supporte Dart JS, Shelf Server, CLI, VM, Flutter (Web, Mobile et Desktop).

L'API des Signals est similaire à celle de Riverpod, mais avec un système push-pull plus avancé. Contrairement à Riverpod, qui gère également l'injection de dépendances réactives, Signals **se concentre exclusivement sur la gestion des états.** Pour faire scaler une application Flutter, il faut donc utiliser une solution d'injection de dépendances comme InheritedWidget, Provider ou GetIt.

Chez Theodo, nous explorons les avantages potentiels de Signals, une technologie prometteuse qui pourrait unifier les pratiques de gestion d'état dans la communauté Flutter, actuellement fragmentée. Bien que Signals soit encore **relativement nouvelle et peu testée** dans des projets de production à grande échelle, **sa popularité croissante dans d'autres frameworks web** indique une convergence vers une technique standard de programmation réactive.

NOTRE POINT DE VUE

Nous vous encourageons à expérimenter avec Signals, tout en gardant à l'esprit qu'elle **n'a pas encore atteint le niveau d'adoption** de solutions établies comme Riverpod ou BLoC. Gardez un œil sur ce package pour ses évolutions futures et son potentiel d'intégration.



21 isar

HOLD
11/11 blips

La gestion locale des données est particulièrement importante pour les applications qui doivent fonctionner hors ligne ou qui nécessitent un haut niveau de confidentialité et de sécurité. Une base de données efficace permet de gérer les données de manière optimale, d'assurer une rapidité d'accès suffisante et de garantir une expérience utilisateur fluide.

Isar est une **librairie de base de données NoSQL rapide et facile à utiliser**, spécialement développée pour les applications Flutter. Conçue pour remplacer Hive, une base de données clé/valeur largement utilisée dans l'écosystème Flutter, Isar promet des performances élevées grâce à son moteur écrit en Rust. Elle offre des fonctionnalités avancées telles que les index composites, les opérations asynchrones et le support multiplateforme (iOS, Android, Desktop). Cependant, nous avons relevé plusieurs points qui justifient une approche prudente.

La dernière période d'activité significative sur le dépôt GitHub d'Isar remonte à un an, ce qui soulève des **préoccupations quant à son développement et à sa maintenance**. De plus, des tests internes réalisés dans le cadre de notre initiative Kaizen ont montré que **le temps d'ouverture des données chiffrées avec Isar pouvait ralentir l'ouverture d'une application** de plusieurs centaines de millisecondes. Cette limitation qui survient à un moment critique de l'expérience utilisateur, est absente d'autres solutions plus matures comme MMKV.

En dépit des promesses de performances élevées garanties par son moteur Rust, Isar présente donc des **limitations en termes de rapidité et de fiabilité** qui sont actuellement bloquantes pour une application d'envergure en production.

NOTRE POINT DE VUE

Nous recommandons de choisir d'autres solutions qu'Isar pour gérer le stockage de données locales, comme MMKV ou ObjectBox par exemple. Bien que la librairie Isar soit prometteuse, il est préférable d'attendre qu'elle atteigne un niveau de maturité et de stabilité plus élevé avant de l'adopter pour des projets de production.

CADRAN

Natif

19 BLIPS | 8 ADOPT | 6 TRIAL | 4 ASSESS | 1 HOLD

Nouveau Changement Pas de changement

Les technologies natives se distinguent par leur maturité, tout en restant un terrain fertile pour l'innovation. Nous notions l'an dernier **une volonté de tous les acteurs de réduire leurs inconvénients historiques**, comme les temps de compilation élevés ou les fichiers de configuration complexes. **Cette transformation semble aujourd'hui largement terminée**. L'expérience développeur est enrichie et fluide, ce qui permet de maximiser la valeur délivrée aux utilisateurs finaux.

De plus, ce n'est pas la fin des innovations : **l'écosystème Android se lance à l'assaut des apps universelles** avec ses outils Multiplatform, tandis que l'écosystème iOS semble décidé à se renouveller par grands pans entiers avec **Swift Concurrency** et **Swift Testing**.

Le futur des applications mobiles natives est prometteur et excitant, et c'est un horizon que nous vous proposons de découvrir dans ce cadran.

PAR **LOUIS PRUD'HOMME**
Tech Lead

ADOPT
1/19 blips

22 Koin

Le pattern d'injection de dépendance est largement utilisé pour appliquer le principe d'inversion de contrôle. Effectivement, ce pattern contribue à une plus grande modularité du code, améliore la maintenabilité et facilite les tests. Toutefois, l'implémentation de l'injection de dépendances peut s'avérer ardue.

Plusieurs frameworks d'injection de dépendances sont disponibles pour Kotlin, parmi lesquels Koin, Dagger-Hilt, Kodein et kotlin-inject se distinguent. Koin, en particulier, semble offrir de nombreux avantages. Explorons pourquoi en le comparant à Hilt, la librairie recommandée par Google pour Android, selon différents critères.

Hilt utilise des annotations qui spécifient les attributs à injecter et la manière de les construire. Bien que Koin supporte également les annotations, il permet aussi la configuration des dépendances via un langage spécifique de domaine (DSL) qui est **intuitif et simple à maîtriser**. Cette configuration est isolée dans un fichier séparé du code métier, facilitant ainsi la séparation des responsabilités.

Koin et Hilt utilisent des approches différentes pour gérer les dépendances. Koin procède à l'injection à l'exécution en tant que service-locator, instanciant toutes les dépendances et fournissant les références nécessaires aux classes. À l'opposé, Hilt réalise l'injection directement lors de la compilation, et ses annotations sont compatibles avec le Kotlin Symbol Processing (KSP) depuis novembre 2024.

Puisque Koin doit résoudre les dépendances à l'exécution, cela a un impact sur les performances, mais la différence de performance est négligeable.

Par ailleurs, Hilt, par son fonctionnement, détecte les erreurs dès la phase de compilation. Toutefois, Koin qui ne les détecte qu'à l'exécution n'a pas dit son dernier mot, car il est possible de vérifier la configuration de Koin dans des **tests unitaires**, évitant ainsi le déploiement d'une configuration erronée.

Concernant Kotlin Multiplatform (KMP), Hilt n'est compatible qu'avec Android natif, tandis que **Koin est multi-plateformes**, offrant un avantage notable. Depuis la Google I/O, il semble que la compatibilité des bibliothèques Google avec KMP soit seulement une question de temps. Reste à voir si Hilt sera facilement adaptable à KMP.

NOTRE POINT DE VUE

En conclusion, bien que Koin ne soit pas la librairie d'inversion de contrôle préconisée par Google, sa simplicité d'utilisation et sa compatibilité multi-plateforme en font un choix privilégié pour nos nouveaux projets. Pour un projet existant utilisant Hilt, une migration vers Koin devrait être envisagée seulement si un passage à KMP est prévu dans un futur proche.

ADOPT
2/19 blips

23 Gradle Version Catalog

Dans les projets multi-modules Gradle, la **gestion des dépendances** et de leurs versions est un **défi** soulevant trois principaux problèmes. Premièrement, la mise à jour des dépendances doit être effectuée sur chaque module. Deuxièmement, il est difficile de vérifier si une **dépendance** est **obsolète**. Enfin, l'**autocomplétion** des dépendances est inexistante. Créer une solution maison est une option, vous en trouverez plusieurs, mais il est peu probable qu'elles atteignent la qualité du Gradle Version Catalog. Cette dernière crée un catalogue unique et centralisé dans fichier minimal basé sur **TOML** : vous pouvez y stocker toutes vos dépendances et leurs versions, y compris les plugins. Il est facile de les mettre à jour au besoin.

En effet, le catalogue de versions offre un avantage : **votre IDE vous avertit des nouvelles versions** de bibliothèques et peut les mettre à jour pour vous. Nous recommandons d'automatiser ces mises à jour avec Dependabot ou une solution similaire.

Tous les modules peuvent référencer les dépendances de manière sûre, permettant à votre IDE de vous aider avec l'**autocomplétion**, même si vous utilisez encore des fichiers Gradle en Groovy au lieu des nouveaux fichiers KotlinScript.

La migration vers le catalogue de versions est **simple et bien documentée**. C'est aussi l'occasion idéale de passer vos fichiers Gradle de Groovy vers Kotlin (.kts).

NOTRE POINT DE VUE

Bien que RefreshVersion soit une autre solution viable, Gradle Version Catalog a été standardisé par Gradle et Google comme **solution officielle de gestion des dépendances**. Nous recommandons de passer à la solution présentée pour tout projet qui n'utilise pas déjà une des deux solutions viables.

ADOPT
3/19 blips

24 KSP

Les développeurs utilisent souvent la génération de code pour éviter les répétitions et **améliorer la lisibilité du code**. En Android, les annotations sont essentielles. Les développeurs peuvent s'appuyer sur des annotations prédéfinies de librairies comme Room, Hilt ou Glide, ou créer des annotations personnalisées pour générer du code spécifique. Historiquement, le "Kotlin Annotation Processing Tool" (KAPT) **était la norme** dans la communauté Android. Il permet l'utilisation d'annotations Java dans le code Kotlin, assurant ainsi la compatibilité avec les librairies Java.

Le Kotlin Symbol Processing (**KSP**), développé par JetBrains et Google, vise à surmonter les limitations de KAPT en traitant directement les annotations Kotlin, **sans les convertir en bytecode Java**. Cette approche est **deux fois plus rapide que KAPT**, car elle élimine des étapes de compilation. En outre, KSP s'adapte mieux aux spécificités du langage Kotlin, telles que les paramètres par défaut, les coroutines et les data class.

De nombreuses librairies, auparavant dépendantes de KAPT, migrent désormais vers KSP, la plupart ont déjà fait cette transition. Une aubaine, car **KSP prend en charge Kotlin Multiplatform**. Les développeurs souhaitant migrer leurs projets d'Android vers Kotlin Multiplatform doivent donc adopter KSP au lieu de KAPT.

Si vous n'utilisez pas d'annotations personnalisées, la **migration est simple**. Toute librairie nécessitant KAPT fournira des instructions détaillées pour passer à KSP.

NOTRE POINT DE VUE

Bien que KAPT soit toujours une option viable, KSP le surpasse dans tous les aspects. Il offre un meilleur support pour les fonctionnalités spécifiques à Kotlin et fonctionne deux fois plus vite. Nous recommandons fortement d'adopter KSP.

ADOPT
4/19 blips

25 SF Symbols

Un point de frustration récurrent que rencontrent les développeurs avec les images matricielles, comme les PNG, réside dans leur manque de flexibilité : redimensionnement manuel, création de versions multiples pour chaque contexte ou couleur. Les images vectorielles, bien que supérieures, demandent également des manipulations ou des versions spécifiques pour chaque utilisation.

Les SF Symbols d'Apple offrent une solution élégante à ces problèmes. Il s'agit d'icônes vectorielles qui s'intègrent **parfaitement** avec le texte de notre application, ce qui améliore grandement l'expérience de développement et la maintenabilité. Leur flexibilité est notable : on peut ajuster l'épaisseur du trait, modifier leur couleur et les rendre multicolores, ce qui simplifie le travail des designers et assure une meilleure intégration dans le design system de l'application. De plus, grâce au moteur de rendu partagé avec les typographies, **leur performance est optimale**.

L'un des avantages majeurs des SF Symbols est leur harmonie avec SwiftUI, notamment en ce qui concerne les animations et l'intégration aux styles hiérarchiques. Pour les développeurs, cela signifie une **réduction du temps de développement** et pour les utilisateurs, une amélioration de la qualité visuelle des applications.

Il est possible de créer ses propres SF Symbols avec l'outil Sketch[1] directement ou en partant d'une base existante. Cependant, la gestion des marges n'étant pas la même par rapport à celle des SVG, il est préférable d'utiliser un convertisseur spécialisé comme SfSymbolConverter[2], surtout si vous avez un set d'icônes déjà existant. Alternativement, les 6 000 icônes disponibles par défaut offrent une solution rapide et efficace pour réduire le lead time des fonctionnalités.

NOTRE POINT DE VUE

Nous vous recommandons l'adoption de cette technologie pour tous vos projets natifs. Malgré quelques subtilités dans la création de symboles personnalisés, les SF Symbols présentent des avantages indéniables en termes de rapidité d'intégration, de confort, de flexibilité et de performance.

[1] https://www.sketch.com/
[2] https://github.com/tychota/SfSymbolConverter

ADOPT
5/19 blips

26 Swift 6 Migration

Swift, malgré ses avantages, souffre de problèmes affectant la productivité des développeurs. D'une part, un problème majeur de **lenteur de compilation**, particulièrement prononcé dans les projets de grande taille, où le typage provoque des échecs du compilateur, avec le message « The compiler is unable to type-check this expression in reasonable time ». D'autre part, **la validité du code concurrent est déléguée** aux développeurs qui l'écrivent.

Swift 6 propose une solution à ces problèmes. Outre une gamme de nouvelles fonctionnalités et de syntaxes, l'une des améliorations les plus significatives de cette nouvelle version est sa **vitesse de compilation accrue**. Cette amélioration à elle seule justifie largement la migration, car elle peut considérablement réduire la frustration des développeurs et améliorer leur productivité.

Mais Swift 6 met également en avant le **code concurrent vérifié** par le compilateur (statiquement), ce qui permet aux développeurs de réduire le nombre de ces bugs parfois difficiles à détecter. Cependant, cette fonctionnalité n'est pas sans inconvénients. Cette validation statique implique des changements majeurs qui perturberont les bases de code existantes, nécessitant ainsi des efforts significatifs de refactoring et d'adaptation. **Cette transition peut être difficile et coûteuse en ressources**, en particulier pour les projets de grande envergure, malgré la migration progressive rendue possible par les feature flags disponibles en Swift 5.

NOTRE POINT DE VUE

Compte tenu de ces considérations, nous recommandons de migrer vers Swift 6. Bien que les changements majeurs nécessitent une évaluation attentive de l'investissement, les avantages de la vitesse de compilation améliorée et de la concurrence vérifiée sont substantiels.

ADOPT
6/19 blips

27 The Composable Architecture

SwiftUI offre des primitives de code puissantes pour externaliser le state, mais nécessitent une architecture bien pensée pour tirer pleinement parti de leur potentiel et ne pas créer de gigantesques states monolithiques. De plus, avec une **communauté de développeurs fracturée** entre UIKit, SwiftUI, les @Observable, les @ObservableObject, différentes architectures, il n'est pas facile de trouver ses marques dans un projet existant.

“The Composable Architecture” (TCA) est un framework conçu pour résoudre ces problèmes dans les applications iOS. Inspiré de l’**architecture unidirectionnelle** Flux popularisée par Redux, TCA adapte cette approche à Swift, rendant la gestion de l’état plus intuitive et moins verbeuse qu'en JavaScript. Contrairement à des bibliothèques comme ReSwift, TCA est conçu pour SwiftUI, tout en restant compatible avec UIKit.

TCA structure le state autour des actions des utilisateurs, assurant un flux de changements **traçable et clair**, ce qui facilite les tests et permet une architecture modulaire. Les fonctionnalités peuvent être développées indépendamment, puis intégrées pour former une application complète. Cette formalisation du code améliore l’expérience de développement et **réduit la courbe d’apprentissage**. En outre, la communauté[1] autour de TCA est robuste et offre une documentation exhaustive[2] et des tutoriels pour accompagner les développeurs.

L'arrivée de la gestion de la navigation, ainsi que le "back-port" de certaines fonctionnalités autrement indisponibles sur des versions antérieures d'iOS, comme l'observabilité du state, témoignent de la transformation de la librairie cette année, la propulsant au **sommet** de son domaine.

NOTRE POINT DE VUE

Chez Theodo, nous avons mis en œuvre The Composable Architecture avec succès dans des projets en production et nous le recommandons vivement pour la gestion de state dans les applications SwiftUI.

[1] https://www.sketch.com/
[2] https://pointfreeco.github.io/swift-composable-architecture/main/documentation/composablearchitecture/

ADOPT
7/19 blips

28 Tuist 4

Nous avons déjà adopté Tuist dans notre précédent Tech Radar en raison de ses capacités à améliorer la gestion des projets iOS. Son utilisation simplifie la configuration des builds. Une meilleure **gestion du cache** des builds permet l'accélération les temps de compilation, en particulier pour les projets basés sur une architecture modulaire. Cependant, en raison des changements majeurs introduits, cette nouvelle version peut s'avérer clivante.

Le premier changement majeur est l’arrêt de la prise en charge de Carthage : chaque projet devra désormais gérer le fetch des dépendances Carthage, ce qui peut nécessiter des ajustements significatifs dans un workflow existant.

De plus, Tuist 4 ne gère plus la signature des applications, obligeant ainsi les développeurs qui utilisaient cette fonctionnalité à revoir entièrement leur workflow de signature.

Une autre nouveauté de Tuist 4 est l’utilisation des fichiers de SPM (Swift Package Manager) plutôt qu’un format spécial. Cette transition permet de mieux s’intégrer dans les **outils de l’écosystème** (par exemple Xcode ou dependabot).

Cette version est peut-être en réalité celle de la **maturité**, car Tuist se concentre sur ce qu’il fait bien afin de le faire mieux. Enfin, les défis cités sont fort heureusement facilités grâce à une documentation de migration fournie ainsi qu'un bon support de la communauté ; cela permet de profiter des améliorations significatives en termes de performance de build apportées par Tuist 4. La syntaxe est également encore **plus intuitive**, ce qui simplifie l’utilisation de l’outil et accélère le développement.

NOTRE POINT DE VUE

Nous vous recommandons toujours d'adopter Tuist pour de nouveaux projets. Nous vous recommandons également la migration vers Tuist 4. Les avantages à long terme en termes de performance et de gestion des projets surpassent largement les défis initiaux de la transition.

29 µ-Features

Architecture

ADOPT
8/19 blips

La complexité croissante des applications mobiles pose des défis, notamment sur la maintenabilité d'une base de code grandissante, le temps de build croissant et les difficultés à respecter la pyramide des tests. Pour y remédier, des entreprises comme SoundCloud et JustEat ont popularisé l'**architecture micro-features**, qui divise le monolithe en modules plus petits et plus spécialisés.

Cette architecture se compose de quatre types de modules :

- Application principale : coordonne les différentes fonctionnalités de l'application.
- Modules de fonctionnalité : gèrent les composants visuels, la navigation et les interactions avec l'utilisateur.
- Modules de logique métier : responsables des services et entités spécifiques à un domaine.
- Modules cœur : interfacent avec des fonctionnalités externes (ex. appels API, stockage, logs).

L'architecture micro-features offre des avantages notables : elle **facilite les tests** des logiques métiers, **réduit le temps de build** grâce au cache, et permet de **mutualiser le code** commun lors de l'agrandissement d'équipe ou de développement de nouveaux produits. Cette approche simplifie également les évolutions et les refontes partielles d'applications complexes.

Cependant, sa mise en œuvre requiert une bonne expertise du domaine pour un découpage efficace, une solide compréhension des principes d'architecture logicielle, et une bonne conception initiale. Des outils comme Tuist pour iOS[1], peuvent faciliter ce découpage, améliorer le cache, et rendre la transition vers cette architecture plus fluide.

NOTRE POINT DE VUE

Chez Theodo, les micro-features sont devenues un choix de référence dans les nouveaux projets natifs. Nous avons pu tester à quel point cette architecture permettait de simplifier les évolutions ou refonte partielle d'applications complexes.

[1] https://docs.tuist.io/guides/develop/projects/tma-architecture

30 Compose

Stability Configuration File

TRIAL
9/19 blips

Avec Jetpack Compose, lorsque le paramètre d'un composable change, seule la partie impactée est mise à jour, à condition que le reste soit stable. Si un paramètre est considéré instable, toute la vue sera re-rendue même si sa valeur n'a pas changée, cela peut **ralentir le rendu** et augmenter la charge du thread UI, affectant les performances des écrans complexes.

Un problème majeur est que **toute classe provenant d'un module différent est considérée comme instable par le compilateur Compose**. Les classes modèles sont souvent séparées des vues pour respecter la séparation des responsabilités et cela dégrade la stabilité des composables. Vous pouvez détecter ces problèmes de stabilité avec des outils comme le layout inspector ou le compose compiler report[1]. Pour résoudre cela, trois stratégies principales[2] ont été utilisées :

- envelopper la classe modèle dans une classe locale stable avec l'annotation @Stable ;
- mapper l'objet sur une copie de votre classe modèle ;
- ajouter la dépendance compose-compiler à la couche modèle pour annoter les modèles comme stables, ce qui nuit à la séparation des responsabilités ;

Jetpack Compose a récemment introduit un fichier de configuration de stabilité[3], une approche déclarative pour gérer la stabilité **sans modifier le code**. Ce fichier permet au compilateur Compose de traiter les classes listées comme stables.

L'utilisation de ce fichier est plus **rapide, élégante et moins contraignante** que les autres solutions. Les développeurs peuvent configurer des classes stables dans un fichier à la racine ou dans des fichiers séparés pour chaque module.

NOTRE POINT DE VUE

Chez Theodo, nous voyons un grand potentiel dans cette solution. Bien qu'elle soit nouvelle et puisse avoir des limites encore inconnues, nous recommandons une implémentation progressive pour assurer la compatibilité des projets.

[1] https://developer.android.com/develop/ui/compose/performance/stability/
[2] https://apps.theodo.com/article/jetpack-compose-toute-classe-dun-autre-module-est-elle-instable
[3] https://developer.android.com/develop/ui/compose/performance/stability/fix#configuration-file

TRIAL
10/19 blips

31 Kotlin

Multiplatform (KMP)

Kotlin Multiplatform (KMP) s'attaque à un problème central du développement moderne : éviter la duplication de code et les coûts associés, tels que la multiplication des bugs et les incohérences entre les plateformes.

Il existe de nombreuses solutions multiplateformes sur le marché, comme React Native ou Flutter. Ces solutions sont des frameworks qui offrent une base de code unique pour créer des applications complètes qui ciblent plusieurs plateformes.

KMP adopte une approche **plus flexible**. Ce n'est pas un framework, mais une technologie permettant au langage Kotlin de **compiler sur d'autres plateformes** que la JVM utilisée pour le développement natif Android.

Cette approche, plus modulaire, rend KMP idéal pour **mutualiser du code** entre différentes plateformes (sous forme d'une librairie) **sans imposer** quoique ce soit.

Sur iOS, par exemple, le code Kotlin est compilé et génère une librairie utilisable dans des projets iOS exactement comme une autre librairie native de cette plateforme. Cette flexibilité permet de partager du code selon les besoins : une simple fonction, une couche spécifique de l'application (réseau, métier, etc.), une fonctionnalité spécifique, ou même l'intégralité du code. **On partage uniquement ce que l'on veut partager**.

Grâce à KMP, Kotlin bénéficie d'une **interopérabilité bidirectionnelle** : on peut compiler du code Kotlin et l'intégrer dans du code natif, tout en utilisant des bibliothèques natives existantes dans du code Kotlin spécifique à chaque plateforme, simplifiant ainsi les ponts souvent nécessaires pour exploiter les fonctionnalités propres à **chaque plateforme**. L'écosystème de KMP est en plein essor, avec Google travaillant activement à rendre ses bibliothèques natives Android compatibles avec KMP. D'un autre côté, la communauté est également **très active**, et les bibliothèques les plus populaires de l'univers Android sont déjà compatibles avec KMP ou très proches de l'être, telles que Room, Retrofit, Coil, Koin, etc.

KMP est désormais **stable** sur presque toutes les plateformes, y compris Android, iOS, Desktop (Windows, Mac, Linux), et même le web en transpilant vers JavaScript/TypeScript. La compilation vers le WebAssembly est quant à elle encore en alpha. De plus, la compilation pour iOS se fait pour l'instant vers de l'Objective-C bien que Jetbrains ait annoncé l'arrivée prochaine de Kotlin-To-Swift lors de la Kotlin Conf 2024 à Copenhague.

KMP s'impose comme une solution **idéale** pour partager du code entre différentes plateformes. Nous vous encourageons à l'essayer : ajouter un module KMP à un projet Android est assez simple (quelques configurations dans Gradle suffisent) et vous pouvez commencer à partager du code. KMP est parfait pour créer une bibliothèque multiplateforme. Cependant, si vous souhaitez également partager l'interface utilisateur, il faudra utiliser Compose Multiplatform (CMP), qui n'est pas encore aussi stable que KMP.

NOTRE POINT DE VUE

Nous vous encourageons à démarrer vos nouveaux projets Android directement avec KMP pour garantir leur évolutivité. Cela n'affecte en rien le projet lui-même, mais vous permettra de partager du code plus facilement à l'avenir.

32 Factory

TRIAL
11/19 blips

Dans le domaine du développement, il est souvent difficile de rendre le code testable et de maintenir une base de code évolutive en raison du fort couplage des dépendances. Une solution efficace pour résoudre ce problème est l'injection de dépendances, qui permet de découpler les composants du code et d'améliorer ainsi sa testabilité et sa maintenabilité.

Factory[1] est une bibliothèque moderne qui résout ces problèmes de manière efficace pour les développeurs Swift. Elle offre des avantages clés tels que la **vérification de l'injection** des dépendances au moment de la compilation, assurant ainsi qu'aucune erreur ne survienne à l'exécution en raison d'une dépendance manquante ou incorrectement injectée. Factory permet également l'utilisation de **scopes d'injections**, facilitant ainsi la mise en place d'une véritable **inversion de contrôle** grâce au pattern Dependency Injection.

L'année dernière, nous recommandions de conserver Resolver pour les projets utilisant UIKit ; cependant, en raison de sa dépréciation, nous préconisons désormais de changer de bibliothèque d'injection de dépendances. Pour ceux qui souhaitent conserver une syntaxe similaire et plus de fonctionnalités, Factory est une **excellente** alternative. Du fait de la proximité de sa syntaxe et de son fonctionnement, la migration de Resolver vers Factory se fait **facilement**. La documentation de Factory est également améliorée, offrant des instructions claires et détaillées pour une utilisation efficace dans les projets SwiftUI et UIKit.

NOTRE POINT DE VUE

Nous recommandons d'essayer Factory dans vos projets car elle présente un potentiel significatif pour améliorer la qualité du code et la performance des applications iOS. Bien que des compromis existent, ses avantages justifient une évaluation sérieuse pour les projets futurs.

[1] https://github.com/hmlongco/Factory

33 Room

TRIAL
12/19 blips

Le stockage des données dans les applications mobiles est essentiel pour améliorer l'expérience utilisateur et éviter les temps d'attente. Un système de cache permet de fonctionner en mode offline-first, en affichant d'abord les données en cache avant de les rafraîchir via un appel réseau.

SQLite est la solution standard pour les bases relationnelles sur mobile, mais il est souvent préférable d'utiliser un ORM (Object-Relational Mapping) pour simplifier les interactions et le typage : 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 code Kotlin, 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 via des méthodes réactives comme des flows (coroutines), des observables (Rx) ou des Livedata.

La nouveauté en 2024 est que, depuis mai, Room est **compatible avec Kotlin Multiplatform**. Bien que cette version soit encore en alpha, Google a annoncé son engagement envers KMP, ce qui laisse présager une version **stable** avant la fin de l'année.

NOTRE POINT DE VUE

Il existe des concurrents sérieux à Room, comme SQL Delight ou Realm, qui offrent également une compatibilité avec KMP. Quoiqu'il en soit, Room demeure relativement simple à utiliser : c'est une valeur sûre pour Android et bientôt pour KMP aussi. Vous pouvez l'intégrer sans risque à votre application Android et envisager son utilisation dans une future version Multiplatform.

34 Swift

Dependencies

TRIAL
13/19 blips

Dans le développement iOS, le découplage des dépendances est un défi récurrent, car aucune solution officielle n'existe. Les projets ont donc souvent un code fortement couplé, rendant les tests et la maintenance difficiles.

Swift Dependencies[1] est une librairie d'injection de dépendances conçue pour **solutionner** ces problèmes dans les applications Swift. Initialement pensée pour The Composable Architecture (TCA), il est tout à fait possible de l'utiliser de façon **autonome**.

Comme avec tous les mécanismes d'injection de dépendances, découpler votre code de ses dépendances et permettre d'injecter des mocks pendant les tests le rend plus facilement **testable et maintenable**. De plus, elle fournit une méthode pour injecter des stubs pour les previews. Cette librairie utilise des mécanismes **similaires à** `@Environment` de SwiftUI pour la gestion des dépendances, ce qui **facilite son apprentissage**. Elle inclut également une macro pour implémenter rapidement la plupart des dépendances, accélérant leur définition et réduisant le code boilerplate.

Swift Dependencies est **légère** et compatible avec une adoption progressive, permettant aux équipes de l'intégrer à **leur propre rythme**. Cependant, cette légèreté se fait au prix de certaines fonctionnalités comme les weak injections et les scopes d'injection, qui doivent être réimplémentées au cas par cas.

NOTRE POINT DE VUE

Swift Dependencies offre donc une manière convaincante gérer les dépendances, que ce soit dans un projet TCA ou en migration incrémentale dans un projet sans injection, même si l'absence de fonctionnalités importantes peut être un frein à son adoption.

[1] https://github.com/pointfreeco/swift-dependencies

35 Typed Errors

en Kotlin avec Arrow

TRIAL
14/19 blips

En programmation, on distingue les erreurs métier (un mot de passe trop court) des erreurs techniques (une requête réseau échouée). Les erreurs métier sont des cas normaux d'exécution et doivent être gérées au même titre que les succès, tandis que les erreurs techniques sont des détails d'implémentation.

Historiquement, Java utilise des exceptions **vérifiées** (checked) **obligatoires**. Mais leur gestion est difficile et elles polluent souvent le code métier, le rendant dépendant de l'implémentation. De plus, elles sont régulièrement dévoyées pour gérer les erreurs métier, ce qui est **coûteux** en termes de performance car elles génèrent une stacktrace.

Kotlin a répondu à ce problème avec des exceptions non vérifiées, rendant la gestion des erreurs **optionnelle**. Cela réduit la robustesse du code mais permet de découpler le code métier de l'implémentation technique. Arrow, une bibliothèque pour Kotlin, propose une approche inspirée de la **programmation fonctionnelle** pour gérer les erreurs métier : les Typed Errors. Cela permet de **distinguer** les erreurs techniques, gérées par les exceptions non vérifiées, des erreurs métier, avec deux méthodes principales :

- Méthode explicite avec `Either` : un type similaire à Result, permettant de spécifier un type d'erreur précis sans hériter de Throwable.
- Méthode implicite avec `Raise` : un DSL similaire aux exceptions vérifiées de Java, fonctionnant parallèlement au système d'exceptions de Kotlin. Actuellement implémentée avec des extensions de méthode, Raise pourrait utiliser les context parameters de Kotlin à l'avenir, bien que ces derniers soient encore expérimentaux et prévus pour Kotlin 2.2.

Ces méthodes évitent try-catch et les stacktraces, **améliorant ainsi performance et flexibilité**.

NOTRE POINT DE VUE

Arrow est un outil puissant pour rendre le code Kotlin plus robuste et lisible. Either est déjà incontournable, mais Raise, bien que prometteur, reste limité par sa dépendance à des fonctionnalités expérimentales.

36 Amper

ASSESS
15/19 blips

La configuration Gradle est incontournable pour tout projet Android ou Kotlin Multiplatform, mais elle peut rapidement devenir complexe si l'on sort des sentiers battus. C'est notamment le cas avec des projets multi-modules, des flavors variées, et l'intégration de plugins tiers. Cette complexité, souvent source de frustration, nécessite une courbe d'apprentissage raide, même pour les développeurs expérimentés.

JetBrains a pris en compte ces défis et a développé **Amper**, un nouveau système de build conçu pour simplifier la configuration. **Intuitif et puissant**, Amper s'intègre parfaitement à l'écosystème existant. Il existe en version standalone pour les projets simples et en tant que plugin Gradle pour ceux qui nécessitent l'écosystème de Gradle.

Amper simplifie la configuration des projets Kotlin Multiplatform avec un fichier par module, **réduisant ainsi le boilerplate** de Gradle. En tant que plugin pour Gradle, Amper est totalement **interopérable**, offrant un fallback Gradle pour toute fonctionnalité non supportée.

Amper supporte la création et l'exécution d'applications JVM, Android, iOS, Desktop et Web, ainsi que la création de **bibliothèques Kotlin Multiplatform**. Il permet également de mixer du code Kotlin, Java et Swift, tout en supportant les projets multi-modules et l'utilisation de Compose Multiplatform.

Cependant, Amper est encore en phase de preview et n'est pas recommandé pour les projets en production : il est encore jeune, avec une API susceptible de changer et des fonctionnalités manquantes.

C'est néanmoins le **bon moment pour expérimenter** et fournir des retours à JetBrains, reconnus pour leur **écoute des utilisateurs**.

NOTRE POINT DE VUE

Amper est spécifiquement conçu pour faciliter le développement de projets Kotlin Multiplatform et pourrait devenir une solution intéressante à explorer pour les développeurs souhaitant simplifier la gestion et la maintenance de leurs builds.

37 Compose Multiplatform

ASSESS
16/19 blips

Kotlin Multiplatform (KMP) permet de partager du code écrit en Kotlin entre diverses plateformes (Android, iOS, Desktop, Web, etc.) mais pas de faire des interfaces utilisateurs. Ainsi, même en partageant la logique métier, il est nécessaire d'utiliser des solutions natives (comme Compose sur Android ou SwiftUI pour iOS).

Pour remédier à cela, JetBrains développe Compose Multiplatform (CMP) qui permet d'écrire une UI **commune à toutes les plateformes** en utilisant l'API de Compose éprouvée sur Android et le moteur de rendu **Skia** éprouvé par Flutter.

CMP est une technologie très prometteuse, mais elle est encore jeune et les niveaux de stabilité **varient** selon les plateformes :

- Android - Stable : intégré nativement ;
- iOS - Beta : bien que déjà **suffisamment stable** pour être utilisée, quelques limitations existent et des changements d'API sont à prévoir ;
- Desktop - Stable : **solidement intégré** sur Mac, Windows et Linux ;
- Web - Alpha : la transpilation en JS/TS fonctionne, mais ses performances laissent à désirer.

C'est la raison pour laquelle JetBrains la déprécie en faveur de sa remplaçante : la version **compilée vers WebAssembly**. Cependant, celle-ci requiert des versions très récentes des navigateurs et n'est pas encore compatible avec Safari. Les deux solutions nuisent au référencement (SEO) et, étant encore en phase alpha, sont particulièrement instables et inadaptées à un projet destiné au grand public.

NOTRE POINT DE VUE

CMP est **très prometteur** et nous croyons en son potentiel : pour les projets ciblant Android et Desktop, il peut être adopté sans hésitation. Pour iOS, il est également recommandable malgré son statut bêta. Cependant, pour le web, il est encore trop tôt pour utiliser CMP en production à large échelle. Les progrès rapides réalisés par JetBrains et la communauté sont encourageants et c'est pourquoi nous surveillons cette technologie de près.

ASSESS
17/19 blips

38 Swift Perception

L'annotation `@Observable`, introduite avec iOS 17, est une avancée majeure pour la réactivité des applications iOS. Cependant, cette fonctionnalité n'est disponible que sur iOS 17 et au-delà. Cela pose un problème pour les développeurs qui doivent maintenir la compatibilité avec des versions antérieures d'iOS (iOS 14 à 16), les privant ainsi de cette fonctionnalité essentielle.

La librairie `swift-perception`[1] vise à résoudre ce problème en **backportant** `@Observable` pour les versions d'iOS 14 à 16 rendant son utilisation possible sans nécessiter de mise à jour de l'OS des utilisateurs ou la création de plus de déchets électroniques.

Cependant, l'utilisation de swift-perception présente quelques inconvénients. D'abord, bien que cette solution émule le comportement de `@Observable`, elle n'est pas aussi bien intégrée que l'implémentation native. La syntaxe fournie est **similaire**, mais pas identique, à `@Observable` : il est nécessaire d'envelopper vos Views dans un composant spécifique, ce qui ajoute du bruit dans le code. Celui-ci devient moins agréable à relire et cela augmente la charge de travail pour supprimer la librairie si elle devient obsolète.

En outre, comme toutes les librairies utilisant les macros avancées de Swift, swift-perception allonge **considérablement** le temps de compilation. Il existe des solutions pour mitiger ce problème, comme la précompilation de swift-syntax[2], mais le cœur du problème demeure.

NOTRE POINT DE VUE

En conclusion, swift-perception est une solution **précieuse** pour les développeurs souhaitant utiliser `@Observable` sur des versions d'iOS antérieures à 17. Malgré ses inconvénients, cela permet de **préparer le code pour le futur** sans se priver d'utilisateurs. Chez Theodo, nous l'intégrons dans notre R&D et nous vous recommandons de l'essayer pour vos projets compatibles avec iOS 14 à 16.

[1] https://github.com/pointfreeco/swift-perception
[2] https://github.com/sjavora/swift-syntax-xcframeworks

ASSESS
18/19 blips

39 Swift Testing

XCTest a incarné les tests iOS depuis qu'il a remplacé OCUnit, mais il est connu pour sa verbosité et son manque de flexibilité. En effet, le code de test croît exponentiellement en quantité et n'est pas facilement lisible. Cela a poussé les développeurs à chercher des alternatives comme Quick. Mais bien que ces outils aient une syntaxe plus expressive, ils ne répondent pas au problème de la verbosité, tout en imposant quand même un apprentissage plus long.

Swift Testing[1] apparaît **prometteur** face à ces défis. Ce framework propose une syntaxe qui tire parti de la puissance des macros et des annotations, afin d'être plus intuitif et moins verbeux. On notera par exemple la **paramétrisation des tests**, permettant aux développeurs d'exécuter le même test avec différentes entrées de manière transparente.

De plus, Swift Testing **améliore l'organisation** des tests grâce à une hiérarchie basée sur les types Swift et une catégorisation basée sur des tags. Il exploite également Swift 6 pour améliorer la sûreté des tests concurrents mais surtout pour lancer les tests en parallèle par défaut, ce qui améliore grandement leur **vitesse d'exécution.**

Enfin, l'intégration aux côtés de tests existants basés sur XCTest est simple, permettant une **migration facile** sans revoir toute l'infrastructure de test.

NOTRE POINT DE VUE

Chez Theodo, nous surveillons de près Swift Testing. Son approche innovante et ses fonctionnalités en font un candidat sérieux pour une adoption future. Cependant, étant donné sa nouveauté, il est actuellement en phase d'assess. Nous pensons que Swift Testing a le potentiel de devenir un outil clé dans notre stratégie de test à mesure qu'il prouve sa fiabilité et s'enrichit de fonctionalités (comme les tests UI ou de performances).

[1] https://developer.apple.com/documentation/testing/

HOLD
19/19 blips

40 Hot Reload

en SwiftUI

Le hot reload est la capacité de voir les changements apportés à une application en cours d'exécution sans avoir à attendre sa recompilation. Cette fonctionnalité accélère le développement en permettant aux développeurs de constater les effets de leurs modifications en temps réel et dans un contexte plus proche du réel que les previews.

Cette capacité est généralement considérée comme inaccessible aux technologies compilées comme SwiftUI. Cependant, une librairie permet de rendre cela possible en recompilant puis en injectant le code modifié grâce à des outils très bas niveau : InjectIII[1]. Cette librairie prend également en charge UIKit, Vapor et même Bezel.

Cependant, l'utilisation d'InjectIII implique de lourdes concessions. La première installation peut être longue et frustrante. Par la suite, il faut faire preuve de beaucoup de patience face aux limitations techniques : il peut être nécessaire de changer le chemin vers votre application ou de compromettre la sécurité de votre Mac pour activer le hot reload. De plus, certaines fonctionnalités de SwiftUI, comme `.onChange`, ne sont **pas compatibles** avec le rechargement.

Le principal inconvénient de cette librairie réside toutefois dans la nécessité de modifier **tous les composants** de votre codebase pour effacer leurs types vers `AnyView`. Ces adaptations considérables peuvent interférer avec les mécanismes d'identification SwiftUI, affectant les animations et les performances de votre application lors du développement.
Il faut également un workflow spécifique pour retirer toute trace de la librairie en production, sans quoi la sécurité et les performances de l'application seraient **compromises**.

NOTRE POINT DE VUE

En **dépit** de l'exploit technique réalisé, nous déconseillons l'utilisation de cette librairie en raison de ses coûts élevés. Néanmoins, cela démontre qu'il est possible de mettre en œuvre le hot reload et certains ingénieurs chez Apple pourraient s'en inspirer.

[1] https://github.com/johnno1962/InjectionIII

CADRAN

Transverse

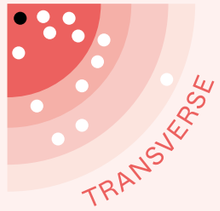
13 BLIPS | 6 ADOPT | 4 TRIAL | 2 ASSESS | 1 HOLD

Nouveau Changement Pas de changement

Cette année BAM, désormais sous marque de Theodo Apps, va marquer son 10-ème anniversaire et le paysage des outils et méthodes de développement dans le mobile a bien changé. Notre partie "Transverse" en témoigne. Il y a 10 ans, les développeurs mobiles n'avaient même pas d'outil d'intégration continue digne de ce nom à leur disposition. Aujourd'hui, nous parlons des techniques d'évaluation de sécurité, d'un nouvel éditeur de code, et d'une intégration encore plus forte entre le code serveur et le code front.

Dans les précédentes éditions, nous avons aussi utilisé cette section pour partager nos pratiques de développement issues du Lean. Cette année marque aussi la sortie du livre "Lean Tech Manifesto" écrit par Fabrice Bernhard et Benoît Charles-Lavauzelle, les fondateurs de Theodo. Nous continuons de vous partager nos bonnes pratiques Lean Tech et en même temps, nous vous invitons à les approfondir avec la lecture de ce livre.

PAR **MAREK KALNIK**
CTO & Co-founder



41 Feature Toggle

ADOPT
1/14 blips

Chez Theodo, nous appliquons une approche Lean avec des déploiements continus et des tâches découpées pouvant être terminées en une journée, évitant les effets de batching. Le **trunk-based development**, où les développeurs intègrent fréquemment leurs modifications sur la branche principale, est au cœur de notre processus, mais reste difficile à appliquer dans le mobile à cause **des cycles de validation des stores**.

Pour mettre en place un flux de travail efficace et avoir un contrôle fin des applications en production, les feature toggles sont essentiels. Il existe plusieurs types de feature toggles, notamment **les permissions toggles et les A/B testing toggles**, qui ont une utilité métier spécifique. Ce sont les **release toggles** et les **operational toggles** qui facilitent les pratiques Lean de développement.

Les release toggles, intégrés directement dans le code, permettent de **merger des fonctionnalités en cours de construction sans les activer** pour les utilisateurs finaux. Cela facilite l'intégration continue et limite le nombre de branches et d'environnements nécessaires au développement.

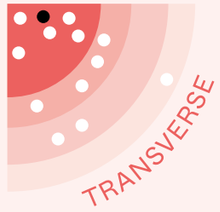
Les operational toggles, configurés de manière distante et récupérés au runtime, permettent de **désactiver rapidement une fonctionnalité problématique ou obsolète** sans nécessiter un nouveau déploiement. Cette flexibilité accrue est cruciale pour réagir rapidement aux incidents en production et maintenir la stabilité de l'application.

Les feature toggles ajoutent de la complexité au code, surtout lorsqu'ils sont interdépendants, multipliant les scénarios de test possibles. Pour éviter que cela ne devienne ingérable, il est essentiel de suivre certaines bonnes pratiques :

- Pour se prévenir de l'interdépendance des toggles, il est recommandé de **ne pas associer plusieurs toggles à la même fonctionnalité**. De plus, il est essentiel de tester unitairement les différents chemins que peut prendre le code togglé, ce qui permet de s'assurer que chaque combinaison possible fonctionne correctement. Lors d'un déploiement, il est également judicieux de ne pas changer l'état de trop de toggles en même temps, afin de limiter les risques de conflits et de régressions.
- Pour garantir une intégration continue efficace, il est pertinent de **changer un release toggle en operational toggle lorsqu'une fonctionnalité est prête à être déployée**. Cela permet de maintenir une continuité fonctionnelle du toggle, facilitant ainsi les tests, les mises à jour et les déploiements sans interruption majeure.
- **Documenter les toggles** est essentiel pour une maintenance à long terme. La durée de vie des toggles peut varier de quelques semaines à plusieurs années, et sans une documentation adéquate, il peut être difficile pour les développeurs de comprendre leur utilisation et leur état.
- **Connaître l'état des toggles en production** est également crucial. Pour cela, il est important de versionner les release toggles, ce qui permet de garantir des builds d'application déterministes. Une console interactive, pour les operational toggles, telle que Firebase Remote Config, peut être utilisée pour surveiller et modifier l'état des toggles en temps réel.
- Enfin, choisir une fréquence de mises à jour adéquate côté applicatif des toggles dynamiques permet de maintenir la capacité de réagir rapidement en cas d'incident.

NOTRE POINT DE VUE

Malgré cette complexité, nous considérons les feature toggles comme une pratique essentielle pour maintenir une cadence de développement rapide et agile, surtout dans un contexte de développement mobile, tout en garantissant un contrôle des applications en production.



42 Flashlight

ADOPT
2/14 blips

Comment savoir si votre app a de bonnes performances ? Cette question est complexe, notamment à cause de la multiplicité des métriques (FPS, TTI, RAM...), et du non déterminisme des mesures.

Flashlight, que nous développons, se veut être une réponse à cette question pour Android. Flashlight donne en temps réel un score de performance à votre app, **sans aucune mise en place préliminaire dans l'app**. Ainsi, à la différence de la plupart des outils existants, **même les apps de production sont supportées**, quelle que soit leur technologie.

Pour aller plus loin, Flashlight peut lancer vos tests E2E plusieurs fois, agréger différentes métriques et moyenner les résultats dans un rapport avec un score attribué. Avantages par rapport à d'autres outils : ce score donne une vue d'ensemble sur la performance facile à comprendre et la vue de comparaison permet d'évaluer l'impact d'un changement dans l'app.

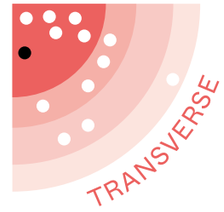
Le cœur de Flashlight est open source. Vous pouvez donc effectuer vos mesures en local avec votre propre device, mais il existe également une version cloud qui tourne sur un vrai device Android et peut être intégrée à une CI pour récupérer un score régulièrement. Néanmoins, il faudra lui fournir vos propres tests E2E (seul Maestro est supporté), ce qui peut s'avérer fastidieux.

Nous utilisons Flashlight comme **indicateur dans des tâches d'amélioration de performance**, qui nous a permis d'améliorer la fluidité du scroll d'une app React Native ou de réduire le temps de démarrage d'une app Flutter.

Flashlight nous a aussi permis d'**éviter des régressions de performance** sur nos projets, en détectant par exemple une consommation CPU anormale suite à l'installation d'un SDK. Enfin, Flashlight est utilisé en collaboration avec Meta pour le suivi des performances du framework React Native.

NOTRE POINT DE VUE

Nous recommandons désormais Flashlight en "Adopt". Nous vous invitons notamment à l'utiliser pour évaluer l'impact de décisions technologiques majeures, ou comme indicateur pour vous aider dans une tâche d'amélioration de performances.



43 Générer son client API

ADOPT
3/14 blips

L'écriture manuelle d'un client d'API REST est une tâche chronophage, qui apporte peu de valeur ajoutée aux utilisateurs. Ce temps qui n'est pas passé à des occupations plus créatives ou utiles est souvent **source de frustration** pour les développeurs. De plus, la répétitivité de cette tâche peut entraîner des **pertes d'attention et des erreurs** dans la traduction des spécifications d'API en code fonctionnel.

Des outils comme OpenAPI Generator[1] permettent de résoudre ces problèmes. Cette approche permet de **générer automatiquement le code client à partir de spécifications d'API** en se passant de toute intervention manuelle.

Cette pratique présente quelques limitations :

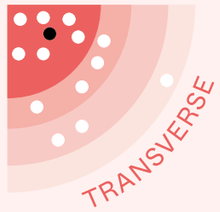
- les outils de génération ne supportent pas toujours l'entièreté des spécifications OpenAPI, c'est le cas par exemple des `oneOf`/`allOf` lors de la génération de client Dart ;
- le code généré peut être incorrect ou incomplet, notamment lorsque le générateur est confrontée à des spécifications partielles ou déphasées par rapport à l'implémentation réelle de l'API spécifiée.

Cependant, cette dernière limitation ouvre en réalité la voie à une **meilleure collaboration** et compréhension entre les équipes de développement : en sachant que leurs spécifications seront utilisées pour générer automatiquement du code client, les développeurs backend sont incités à respecter rigoureusement les normes OpenAPI et à fournir des spécifications claires et précises.
Cela améliore la qualité du code généré, la documentation et **la maintenabilité des API** en plus de favoriser une culture de collaboration et de qualité au sein des équipes.

NOTRE POINT DE VUE

Nous recommandons vivement la génération automatique de client API pour toute organisation cherchant à optimiser ses processus de développement et à produire des applications de qualité supérieure. Il faut toutefois garder à l'esprit que cette pratique nécessite un **investissement conjoint des équipes frontend et backend**.

[1] https://github.com/OpenAPITools/openapi-generator/



44 Rive

ADOPT
4/14 blips

Les designers incorporent de plus en plus d'animations dans leurs interfaces pour augmenter la qualité perçue d'un produit et **générer de l'émotion chez les utilisateurs**.

Cependant, créer des animations 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.

Une autre approche consiste à intégrer des assets animés, mais la plupart des solutions existantes **ne permettent pas d'interagir directement avec les animations via le code**, rendant impossible (ou très compliqué) 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 par une fine compréhension du processus de production des interfaces digitales, en proposant notamment un **pipeline de production unifiée entre designers et développeurs**.

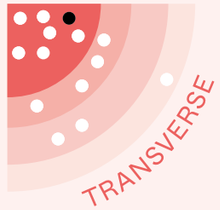
- Côté design, Rive élimine la nécessité d'utiliser plusieurs outils comme illustrator ou After Effect. L'éditeur permet de concevoir et d'animer au sein d'une même interface et facilite la transmission d'assets.
- Côté technique, Rive **réduit les allers-retours avec les designers**, élimine les problématiques de fichiers corrompus et simplifie l'intégration d'animations intéractives avec des state-machines intégrées.

Rive rend la collaboration entre designers et développeurs plus fluide, réduit les erreurs et accélère le processus de production. 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. Rive optimise ses fichiers pour les rendre jusqu'à **10 fois plus légers que les fichiers Lottie**. De plus, il limite l'impact sur la codebase en intégrant la state-machine directement dans le fichier.

L'outil Rive se positionne comme un compétiteur solide face aux autres solutions de création d'animations grâce à ses meilleures performances et API 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

Après plus d'un an d'utilisation, nous recommandons désormais d'adopter Rive pour **apporter du mouvement** à vos produits grâce à des animations interactives de haute qualité.
Les contraintes financières et techniques évoquées l'année dernière nous semblent aujourd'hui largement compensées par la simplification des flux de production.



45 Supabase

ADOPT
6/14 blips

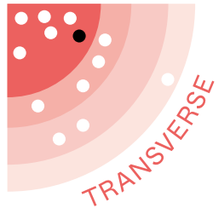
Nous sommes constamment à la recherche d'outils qui simplifient le développement backend tout en offrant des fonctionnalités robustes. Supabase a retenu notre attention en tant que plateforme open-source de backend-as-a-service (BaaS) qui **facilite considérablement l'intégration de systèmes d'authentification complexes et la gestion de l'infrastructure backend**.

Supabase se distingue en utilisant PostgreSQL, offrant une flexibilité pour des requêtes complexes et la gestion des données. Son système d'authentification est hautement configurable, prenant en charge les emails/mots de passe, les liens magiques et les fournisseurs tiers. **La qualité de ses SDK garantit un développement fluide sur diverses plateformes**, et son intégration avec les frameworks front-end améliore l'accessibilité pour les développeurs. Supabase permet d'écrire des edge functions serverless basées sur Deno et de gérer la sécurité de manière fine avec les RLS. Cela améliore les **performances**, la **sécurité** et la **scalabilité**, ce qui le rend adapté aux exigences des applications modernes. Les capacités d'auto-hébergement de Supabase offrent potentiellement un contrôle total sur les données et l'infrastructure, **répondant aux besoins de confidentialité et de personnalisation**.

Compte tenu de ces atouts, nous plaçons Supabase dans la catégorie Adopt. Ses fonctionnalités robustes, sa facilité d'utilisation et son caractère open-source en font un choix exceptionnel pour les développeurs. Supabase simplifie non seulement le développement backend, mais offre également des fonctionnalités avancées essentielles pour les applications à grande échelle.

NOTRE POINT DE VUE

Nous recommandons fortement d'adopter Supabase pour améliorer l'efficacité du développement et la scalabilité des projets.



46 UI Snapshot testing

ADOPT
7/14 blips

Le défi d'assurer la cohérence de l'UI d'une application mobile est un casse-tête pour tous les développeurs. Cette difficulté est accentuée par la **multitude d'appareils** (téléphones, tablettes, téléviseurs, ...), **d'orientations et de tailles d'écran**, rendant les tests de régression manuels presque impossibles. Le Snapshotting UI offre une solution moderne à ces défis. Le Snapshotting UI consiste à capturer des snapshots de l'UI de votre application dans divers états et à les stocker dans votre codebase. Lorsqu'un changement est apporté, ces snapshots sont comparés à l'interface actuelle pour détecter toute modification non intentionnelle.

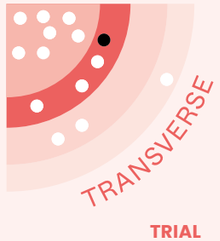
Nous avons utilisé le Snapshotting UI sur plusieurs plateformes, chacune ayant un degré de maturité différent :

- **Flutter** : la plus avancée, offrant des tests de snapshots intégrés avec Flutter goldens. Cela ne nécessite pas d'émulateurs, rendant les tests extrêmement rapides.
- **Android natif** : nous utilisons la bibliothèque Paparazzi qui capture des snapshots d'UI et teste les régressions sans utiliser d'émulateurs.
- **iOS natif** : bien que la bibliothèque swift-snapshot-testing repose sur des simulateurs, elle est suffisamment rapide pour avoir un impact minime sur notre CI.
- **React Native** : ce framework manque d'une solution mature. Les tests de snapshots de l'UI nécessitent une configuration supplémentaire et s'appuient généralement sur des tests end-to-end sur émulateurs associés à jest-image-snapshot.

Un défi majeur que nous avons rencontré est l'incohérence des snapshots entre les différentes architectures CPU, simulateurs et systèmes d'exploitation, ce qui peut entraîner des écarts dans l'environnement de CI. Pour atténuer cela, nous pouvons réduire la précision de la configuration afin d'obtenir des snapshots cohérents sur la CI.

NOTRE POINT DE VUE

Adopter le Snapshotting UI améliore la productivité et garantit une expérience utilisateur cohérente et fiable à travers les mises à jour. Malgré les défis liés aux incohérences entre les environnements et au potentiel ralentissement des tests qui tournent sur émulateur, les avantages l'emportent largement sur ces inconvénients.



TRIAL
8/14 blips

47 Identifying Defect Detection Stage

Dans Dantotsu Radical Quality, Sadao Nomura propose de classer les défauts selon leur **étape de détection**, plutôt que leur gravité ou leur urgence. Nous avons développé ses idées dans The Lean Tech Manifesto, en suggérant cinq étapes :

A - Détecté avant la mise en production par le développeur

B - Détecté avant d'atteindre un client interne (revues de code, intégration continue)

C - Détecté avant la mise en production (revues fonctionnelles)

D - Détecté après la mise en production mais avant une plainte (déploiement continu, monitoring)

E - Détecté par un utilisateur

Cette approche contraste fortement avec la pratique courante de l'industrie, où les bugs sont classés par gravité, et où seuls les plus critiques sont traités et analysés. Détecter les défauts tôt et en discuter présente plusieurs avantages :

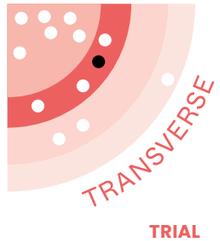
- Les bugs récents sont frais dans l'esprit du créateur, **facilitant l'analyse et l'apprentissage**.
- On évite que la personne ayant introduit le bug soit partie, empêchant tout apprentissage.
- **Les bugs mineurs** révèlent souvent des méconnaissances pouvant causer des problèmes plus graves ; **les traiter améliore la stabilité globale**.

Chez Theodo, nous avons progressivement inclus les étapes A à C dans notre analyse et visons à réduire les bugs détectés tardivement (D et E) en les identifiant plus tôt. Nous pensons qu'une augmentation des détections aux premières étapes est positive, si cela permet de réduire celles des étapes ultérieures. Cela présente plusieurs avantages :

- L'équipe traite les obstacles plus tôt, **améliorant la productivité globale**.
- Nous réduisons le **coût de correction des défauts**.
- L'équipe **génère des connaissances** plus rapidement.
- **La qualité du produit final s'améliore.**

NOTRE POINT DE VUE

Nous recommandons cette méthode, mais sa mise en pratique nécessite une **bonne culture générative**. Se concentrer sur l'**apprentissage** et **éviter de blâmer** est essentiel, et certaines équipes peuvent ne pas être prêtes à adopter pleinement cette méthodologie.



TRIAL
9/14 blips

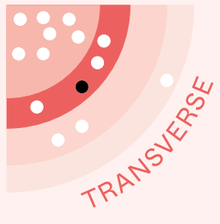
48 Maestro

Faire des tests E2E pour des applications mobiles, en particulier pour React Native et Flutter, est fastidieux. Appium (et Detox) ont leurs atouts, mais ils sont souvent complexes à utiliser, peu intuitifs et peuvent être difficiles à automatiser sur la CI/CD.

Maestro est un nouveau framework de test d'interface utilisateur mobile très efficace. Bien qu'il ne dispose peut-être pas d'un ensemble de fonctionnalités aussi complet qu'Appium, c'est un **choix idéal pour ceux qui débutent dans les tests E2E** grâce à son setup minimal et à sa mise en œuvre rapide. Vous pouvez faire tourner votre premier test en seulement **15 minutes**. Maestro comprend aussi une interface graphique (Maestro Studio) qui vous permet de **sélectionner visuellement des éléments de l'interface utilisateur** et propose des suggestions sur la manière d'interagir avec ces éléments dans vos tests. De plus, Maestro possède son propre service cloud intégré appelé Maestro Cloud. Pas besoin de configurer des simulateurs, il vous suffit d'uploader votre application et vos tests, et ils se chargent du reste (rapports, exécution parallèle, prévention des erreurs intermittentes, enregistrements d'écran, logs, etc.).

NOTRE POINT DE VUE

Nous sommes encore en train d'intégrer cet outil à certains projets. Trouver la bonne configuration adaptée à chacun de ces projets peut être complexe. Reste à déterminer si les aspects tels que la tarification, la facilité d'utilisation, la fiabilité et le retour sur investissement en feront un choix judicieux, mais nous plaçons de grands espoirs sur Maestro !



TRIAL
10/14 blips

49 MASVS 2.1

En 2024, la sécurité des applications mobiles est plus que jamais une priorité, face à des menaces en constante évolution. Avec 7 milliards de souscriptions aux réseaux mobiles[1] et un tiers des entreprises[2] subissant des interruptions ou des pertes de données dues à des compromissions sur des appareils mobiles, il est essentiel pour les développeurs d'**intégrer la sécurité dès la phase de conception de leurs applications**.

Le Mobile Application Security Verification Standard (MASVS) d'OWASP, un standard incontournable pour la sécurité des applications mobiles, a récemment évolué avec les versions 2.0 et 2.1. MASVS 1.5 se composait de 84 points de contrôle répartis en 7 catégories couvrant des domaines critiques tels que le stockage des données, l'authentification, et les communications réseau. La version 2.0 de MASVS a apporté des simplifications significatives, éliminant les redondances et adoptant des terminologies standardisées comme OSCAL. Le résultat est une **liste rationalisée de 22 points de contrôle**, offrant une approche plus claire et concise.

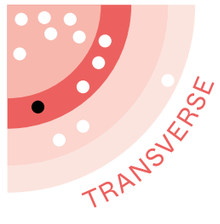
L'une des innovations majeures de MASVS 2.0 est l'introduction des profils MAS. Ces profils configurent les tests de sécurité en fonction des exigences spécifiques de l'application, en tenant compte de son niveau de sensibilité. Les tests associés sont décrits en détail dans le **guide de test (MASTG)**, qui fournit des conseils techniques pratiques pour valider les 22 points de contrôles de sécurité.

La version 2.1 a enrichi le standard avec la catégorie MASVS-PRIVACY, essentielle pour aider les développeurs à se conformer aux réglementations de protection des données, comme le RGPD.

NOTRE POINT DE VUE

Chez Theodo, nous avons adopté MASVS comme **framework principal pour sécuriser nos applications.** Ce standard offre une solution complète et flexible, parfaitement adaptée aux besoins variés de nos projets. Son approche structurée facilite non seulement la mise en œuvre des contrôles de sécurité, mais renforce également notre collaboration avec les auditeurs sécurité.

[1] https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/
[2] https://www.verizon.com/business/resources/executivebriefs/2022-mobile-security-index-report-executive-summary.pdf



TRIAL
11/14 blips

50 Ship Show Ask

Pendant le développement d'une fonctionnalité, la relecture du code est cruciale pour garantir sa qualité et sa conformité aux normes de l'équipe. Toutefois, un contrôle systématique peut ralentir les développeurs.

Pour éviter cela, Rouan Wilsenach propose une approche intitulée "Ship, Show and Ask". Elle consiste pour chaque changement à choisir parmi ces trois options :

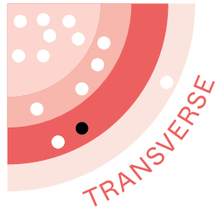
- **Ship** : créer une branche poussée sans relecture pour les changements mineurs et standardisés ;
- **Show** : créer une branche avec une pull-request et merger sans attendre la relecture pour les changements qui ne sont pas critiques, mais nécessitent une relecture ultérieure ; ainsi que pour les changements faits en équipe (pair 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.

Nous avons parlé de cette pratique l'année dernière et elle fait partie désormais de nos standards. Nous avons diminué le temps passé sur les code reviews sans pour autant diminuer la qualité du code.

Tout comme la relecture de code, le pair / ensemble programming, Ship / Show / Ask est un moyen de communication entre un manager et un managé. C'est au Tech Lead d'utiliser cette méthode à bon escient pour garantir la cohérence du code, la progression des développeurs et la capacité de l'équipe à délivrer des fonctionnalités rapidement. Il lui revient la responsabilité de définir des règles pour améliorer la qualité.

NOTRE POINT DE VUE

"Ship, Show and Ask" permet d'accélérer le développement tout en maintenant la qualité du code, mais une stratégie de qualité de code globale reste primordiale pour garantir la robustesse et la cohérence des livrables.



51 Weekly
Engineering Review

ASSESS
12/14 blips

Gérer une équipe technique en croissance est un défi. À mesure qu'elle se développe, avec plusieurs projets, différentes technologies et différents contextes, il devient difficile de s'assurer que tout le monde est aligné sur le plus important, et encore plus difficile d'assurer la collaboration et le partage des connaissances inter équipes. Chaque Engineering Manager travaille dur avec son équipe, tandis que d'autres rencontrent les mêmes problèmes de l'autre côté de l'open space.

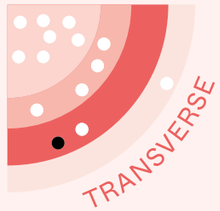
Le Weekly Engineering Review est un rituel qui rassemble tous les Engineering Managers et Staff Engineers de l'entreprise. Ce principe est fortement inspiré par le **Weekly Business Review** d'Amazon. L'objectif est de garantir un **alignement fort de tous les leaders techniques** sur des indicateurs de performance communs et de favoriser le partage des connaissances et la collaboration entre les différentes équipes. La réunion est structurée autour d'un **ensemble de KPIs** qui agrègent la performance de toutes les équipes techniques dans différents domaines : livraison, qualité, santé de l'équipe… Les KPIs dépendent du contexte, ils peuvent donc et doivent évoluer au fil du temps pour refléter les défis actuels et s'aligner avec les enjeux business de l'entreprise. L'aspect **hebdomadaire** permet à chaque leader technique de maintenir un focus fort sur les objectifs et les aide à aligner le travail quotidien de leurs équipes avec ces derniers. Un **responsable est assigné à chaque KPI** pour s'assurer que les données sont collectées, que le KPI est mis à jour chaque semaine en amont de la réunion et commenter les variations. Enfin, la WER est structurée par un **facilitateur** qui veille à ce que chaque KPI soit préparé et que la réunion se déroule efficacement.

Chez Theodo Apps, nous avons mis en place la WER depuis 6 mois. Nous avons suivi des KPIs tels que le nombre de bugs non résolus, le coût de chaque écran développé dans nos projets, le nombre de CFP envoyés par nos ingénieurs pour des conférences, le nombre de partages d'apprentissages inter équipes ou encore le nombre d'ingénieurs ayant participé à des bootcamps ou formations.

Après quelques mois de fonctionnement de la WER, nous avons constaté un **impact mitigé**. Certains KPIs ont été efficacement impactés à la hausse, comme la participation aux formations ou le nombre de défauts analysés par les chefs d'équipe pour mieux comprendre l'origine de bugs. D'autres KPIs, comme le nombre de CFPs envoyés, n'ont pas connu d'évolution significative. Indépendamment de son impact direct, la WER se révèle être un moyen efficace pour créer une **véritable équipe de leaders techniques** qui opéraient auparavant dans des silos. Le retour de tous les participants est très positif quant au sentiment d'appartenance à une équipe générée.

NOTRE POINT DE VUE

Il est encore trop tôt pour mesurer l'impact de la WER sur la performance de l'entreprise ou constater des améliorations tangibles au sein des équipes. Nous apprenons encore comment exécuter correctement ce rituel et considérons cette pratique comme une **expérience en cours sur le management et le leadership d'ingénierie** en attendant de stabiliser sa pratique.



52 Fleet

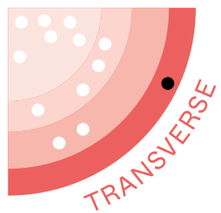
ASSESS
13/14 blips

En développant pour Kotlin Multiplatform (KMP), les développeurs doivent jongler entre Android Studio pour Kotlin et Xcode pour Swift, ce qui perturbe le flux de travail, augmente les **changements de contexte** et peut entraîner des **conflits de synchronisation** lors de l'édition de fichiers dans l'un des éditeurs. Fleet, un IDE de JetBrains, a été créé pour **faciliter l'utilisation de KMP**. Il résout ces problèmes en permettant aux développeurs de gérer le code des deux langages dans un seul IDE sans avoir besoin d'utiliser d'autres éditeurs.

En tant que fournisseur de confiance d'outils comme IntelliJ et Android Studio, JetBrains garantit une meilleure efficacité pour les développeurs multiplateformes avec Fleet. L'IDE offre des fonctionnalités telles que l'auto-complétion et une **navigation dans le code qui surpassent même celle de Xcode**. Cependant, Fleet est encore en version préliminaire et présente certaines limitations, notamment l'absence de fonctionnalités et des problèmes de stabilité. Par exemple, il n'est pas possible d'ajouter des plugins, ce qui signifie que **vous ne pouvez pas utiliser Copilot**, mais JetBrains propose une alternative d'auto-complétion basée sur l'IA.

NOTRE POINT DE VUE

Fleet montre un potentiel prometteur, mais il est encore immature. Nous recommandons de l'essayer sur des projets nécessitant un engagement minimal. Il constitue un allié précieux pour le développement d'une application KMP. À l'avenir, il pourrait également être utile pour un bridge React Native ou un plugin Flutter. Chez Theodo, nous continuons à suivre son développement et à expérimenter avec cet outil pour optimiser notre productivité.



53 PWA-first

Mobile strategy

HOLD
14/14 blips

Il y a quelques années, les Progressive Web Apps (PWA) étaient très en vogue. Certains allaient même jusqu'à prédire la fin des applications mobiles, et quelques études de cas ont présenté des arguments convaincants en faveur des PWA.

Cependant, **les applications universelles ont mûri entre-temps**, tandis que les PWA n'ont jamais vraiment décollé. Leur adoption a été freinée par un **manque de support sur iOS**, des **problèmes de découvrabilité**, des difficultés à gagner la confiance des utilisateurs et des **fonctionnalités manquantes** pour les développeurs.

Le fait que le PWA Summit n'ait eu lieu que deux fois et n'ait pas été reconduit en 2024 est un signe que les PWA perdent de leur élan. Certains acteurs les utilisent encore avec succès, et de nouvelles PWA sont régulièrement déployées, mais nous pensons qu'elles ne constituent plus une stratégie mobile solide.

NOTRE POINT DE VUE

Les idées proposées par les PWA restent globalement pertinentes, mais dans leur état actuel, les applications universelles (que ce soit avec React Native ou Flutter) offrent bien plus d'avantages, tout en intégrant certaines fonctionnalités des PWA, comme le manifeste web et l'aptitude à être installées. Nous vous recommandons de vous concentrer sur les applications universelles et d'y intégrer des fonctionnalités PWA, plutôt que de partir directement d'une PWA.

NOTRE STACK ACTUELLE

Autres technos adoptées

Depuis que nous avons lancé notre Tech Radar en 2022, nous continuons à explorer et à partager des technologies innovantes et des choix techniques audacieux. Au fil des éditions, vos retours et vos questions nous ont encouragés à approfondir nos recommandations pour aider à démarrer de nouveaux projets. Cette édition s'inscrit dans cette démarche, et nous avons le plaisir de vous présenter une mise à jour de notre stack technologique actuelle, incluant les technologies les plus couramment utilisées pour lancer un projet from scratch.

STACK REACT NATIVE

- Expo + EAS
- Typescript + ESLint + Prettier
- Jest + React Native Testing Library
- React Query / Apollo
- Zod
- Jotai / Zustand
- Yarn4
- React Navigation
- Luxon
- Reanimated
- React Native Unistyles
- React Native MMKV
- React Hook Form
- React Native Vision Camera
- For TV : React Tv Space Navigation

STACK NATIVE ANDROID

- Kotlin
- Flow + Coroutines
- Jetpack Compose
- Clean Architecture + MVI
- Koin
- Retrofit + KotlinX Serialization
- Jetpack DataStore
- Room
- KtLint
- Slack Compose Linter
- MockK
- MockWebServer
- Paparazzi
- Maestro
- Rive
- Flipper

STACK FLUTTER

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

STACK NATIVE IOS

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

NOS CONTRIBUTEURS

Le comité de contenu

ALEXANDRE MOUREAUX
App Performance Expert

ARTHUR LEVOYER
Head of Native

CYRIL BONACCINI
Staff Engineer

DENNIS BORDET
Tech lead

GUILLAUME DIALLO-BOISGARD
Head of Flutter

LOUIS DACHET
Tech Lead

LOUIS PRUD'HOMME
Tech lead

MAREK KALNIK
CTO & Co-founder

MATTHIEU GICQUEL
Staff Engineer

MATTHIEU PERNELLE
Tech Lead

MAXIME ROUGIEUX
Tech lead

NICOLAS ACART
Developer

PIERRE POUPIN
Tech Lead

UN GRAND MERCI À NOS CONTRIBUTEURS DE L'OMBRE :

Alexis Ego / *Solutions Architect* • Antoine Cottineau / *Developer* • Antoine Doubovetzky / *Head of React Native* • Hugues Baratgin / *Developer* • Julien Calixte / *Engineering Manager* • Louis Giboin / *Developer* • Mathieu Fedrigo / *Tech Lead* • Micha Dyatlov / *Developer* • Mo Khazali / *Head Of Mobile - Theodo UK* • Paul Briand / *Developer* • Pierre Zimmermann / *Tech Lead* • Rémi Bougaud / *Lead Designer* • Skander Ellouze / *Developer* • Thomas Coumau / *Developer* • Tanguy Moisson / *Developer*

QUI SOMMES-NOUS ?

À propos de Theodo

Theodo est une entreprise internationale de conseil en technologie. Nous accompagnons les entreprises dans la conception, le développement et le déploiement de produits digitaux ingénieux qui transforment la vie de leurs utilisateurs.

Pour aider nos clients à chaque étape du cycle de vie de leurs produits, nous avons développé des entités expertes, spécialisées dans les technologies clés de la conception et du développement produit ou dans certains secteurs de l'économie.

- **Theodo Cloud** : Modernisation des infrastructures IT
- **Theodo Apps** : Applications multiplateformes
- **Theodo Data & IA** : Plateformes de données et Solutions IA
- **Theodo HealthTech** : Systèmes et services de santé
- **Theodo FinTech** : Services financiers
- **Theodo GovTech** : Solutions pour le secteur public

LES CHIFFRES CLÉS

- 15 ans d'existence
- 700 Theodoers passionnés
- Des équipes en France, au Royaume Uni et au Maroc
- Des clients dans plus de 12 pays

À PROPOS DE THEODO APPS

Theodo Apps est l'entité experte technologique du groupe Theodo spécialisée dans la conception et le développement d'applications mobiles et multiplateformes. Nous travaillons en étroite collaboration avec les différentes entités du groupe pour offrir à nos clients des solutions sur mesure.

10 ans
d'expérience

250
produits développés

120
experts

CONCEPTION & RÉALISATION

Pauline Gaillard & Ariane de Bélizal / *Cheffes de projet*
Stéphanie Landrein / *Direction artistique*

MOBILE

theodo.
Apps