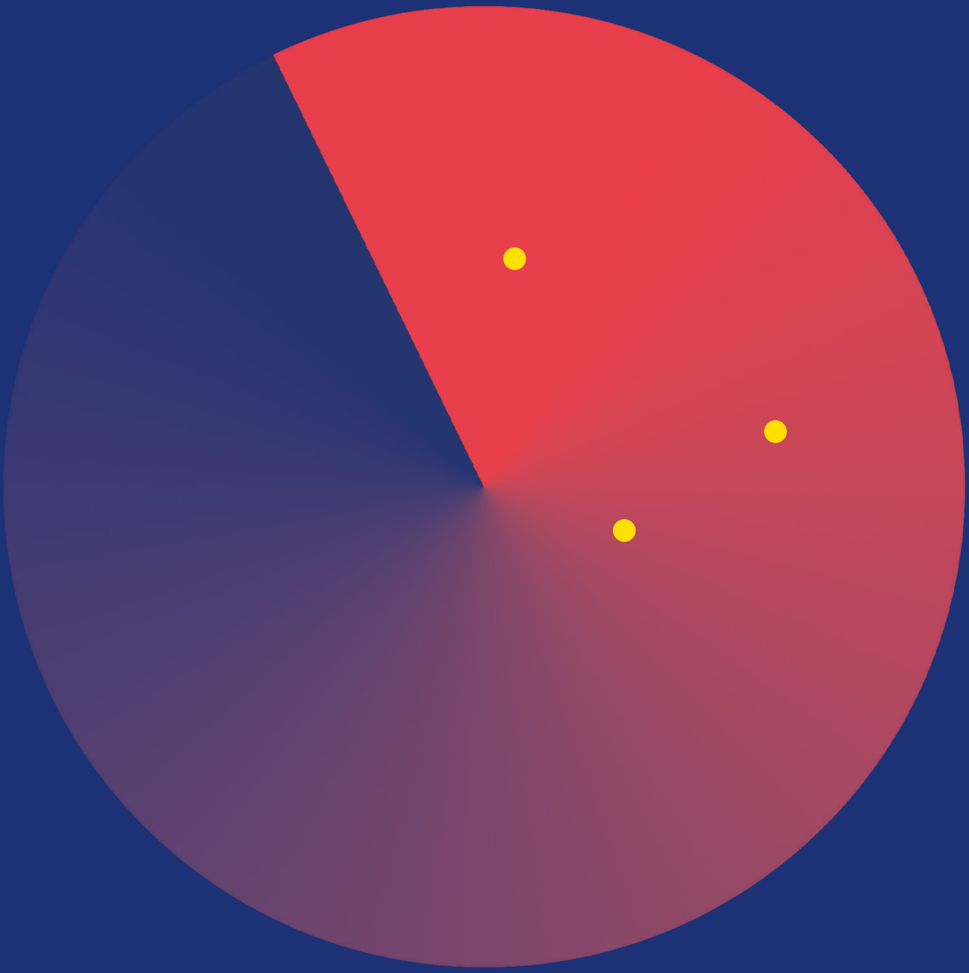
Tech Radar

#3

NOVEMBER 2024



theodo. Apps

MOBILE

Inspired by your daily feedback and co-created by our content committee of 13 experts, the first 100% mobile focused Tech Radar. Take a look at our third edition!

Introduction

While the mobile ecosystem has reached a certain level of maturity, it remains extremely dynamic. Increase in technology, frameworks, and SDKs presents complex choices for any mobile team. In addition, multi-platform support (web, TV, desktop) is becoming more extensive and robust, encouraging code sharing between different front-end applications. For years, we've been building a knowledge base that we've decided to open so that the community can benefit from our experience and insights.

This Tech Radar, co-created over the course of several workshops, is a snapshot of the topics driving discussions within Theodo Apps' 3 tech tribes. The 13 members of the Content Committee have been exploring the technologies and techniques that make up the 53 Blips selected by our tech team. We invite you to discover and respond to their points of view.

With the goal of publishing our vision of the mobile ecosystem, we made certain editorial choices that strongly influenced the content:

- **pass on our expertise:** by talking about the technologies we use daily, that we have experimented with or that we have been following for some time.
- **not addressing obvious choices:** certain approaches or technologies are clearly adopted by a large majority of the community, sometimes even mentioned in a technology's official documentation. If we feel that our opinion will not add value to a technology, you won't find it on the radar.
- **be clear about our recommendations:** to provide you with the best possible guidance, we have chosen to adopt clear, assertive positions.

This approach allows us to present our vision divided into 4 levels of recommendation, based on our interpretation:

- **Adopt:** don't hesitate, in our opinion it's the best choice to date.
- **Trial:** we invite you to test this technology, as it has great potential for meeting your needs. It can be integrated into a production project, if you have properly assessed the risks and alternatives.
- **Assess:** we think this technology is worth keeping an eye on, as it is likely to gain in importance over the coming months. It may be worthwhile to read up on the subject or to carry out a "proof of concept".
- **Hold:** we'd advise against this technology at this stage, either because we don't consider it mature enough, or because we think it's less relevant than its competitors.

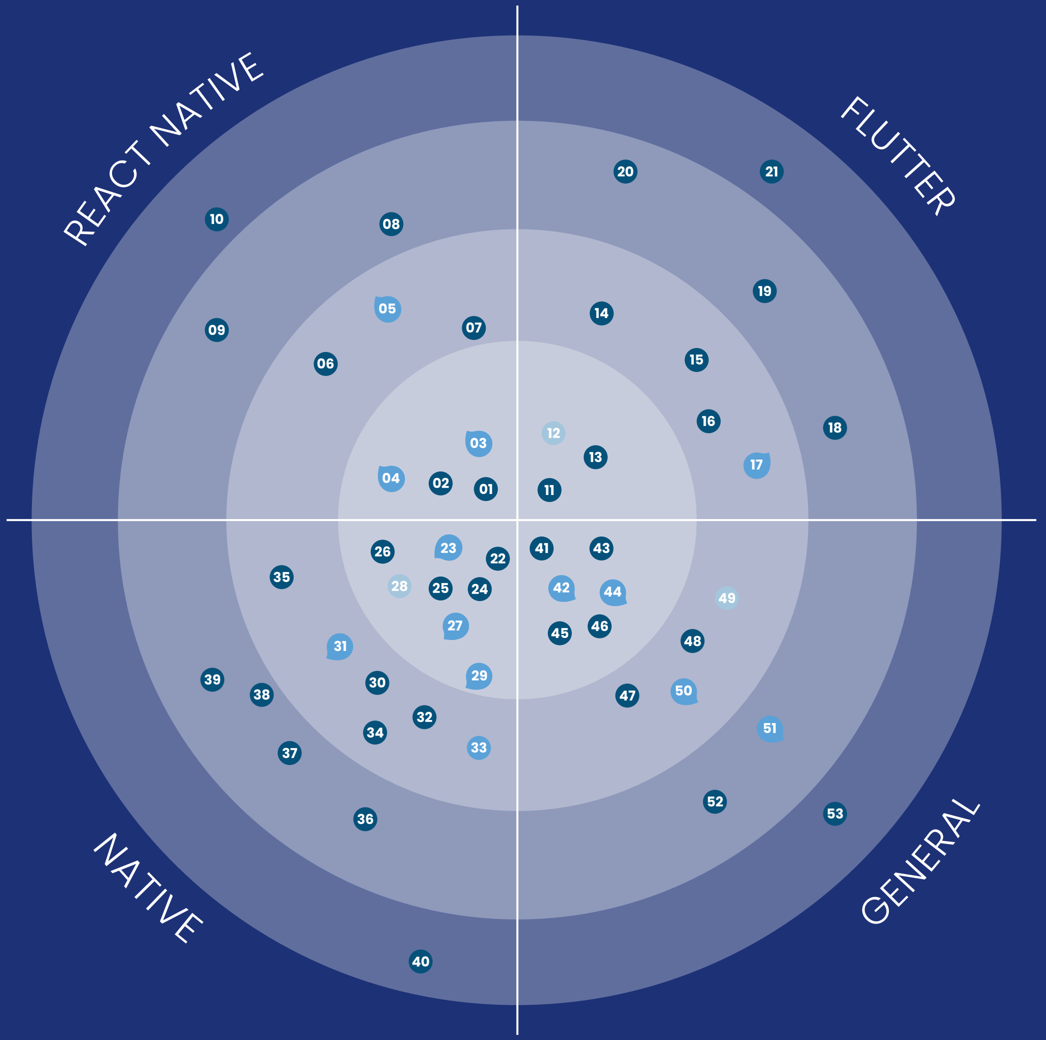
Although future developments may change the situation, we think it's best not to go down this road for the time being.

We'd love to hear from you and look forward to exchanging our visions and technical expertise!

SOMMAIRE

Our Tech Radar

In this 100% Mobile Tech Radar, we share our expert opinion on the techniques, platforms, tools, languages and frameworks associated with the main technologies we use every day: React Native, Flutter, and native technologies.



● New ● Move ● No change

Blips

REACT NATIVE

1. Migrate to 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

NATIVE

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 in Kotlin with Arrow
36. Amper
37. Compose Multiplatform (CMP)
38. Swift Perception
39. Swift Testing
40. SwiftUI Hot Reload

GENERAL

41. Feature Toggles
42. Flashlight
43. Generate an API client
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

Which alternative to choose?

You may notice that our radar doesn't make a clear recommendation between Flutter, React Native, and Native technologies. These 3 technologies define the choices, but we don't compare them directly.

When we launched BAM, now Theodo Apps, in 2014, the choice of technology was a very risky one. We were convinced that cross-platform was the future of mobile and we chose a technology stack consisting of Cordova and Ionic. But this was not an obvious choice, given the many competitors to Cordova, such as Xamarin (backed by Microsoft) or Titanium (which uses native UI components).

Each of these solutions had distinct strengths and significant drawbacks. In 2015, the landscape changed with the emergence of React Native, which solved most of the problems of the other frameworks. We adopted it as early as October 2015. Two years later, Flutter was created with a technically different approach, but with the same level of quality. Over time, the Flutter solution has proven itself. At the same time, the emergence and growing popularity of Swift and Kotlin brought a lot of freshness and modernity to native development, to the detriment of cross-platform development.

So, we've gone from a phase where the choice was not obvious (before summer 2015), to a phase where the choice was clear (2015-2018), before becoming very complex again. This is why we choose a specific solution for each project and this is part of the first discussions we have with our customers.

These discussions must consider:

- **product strategy:** what functions, what design, who will be the users?
- **tech vision:** what is the company's technical vision, who is going to work on this project, what are the existing teams, what is the recruitment strategy?
- **the budget:** how much can we invest, over what period and what are the project's financing conditions?

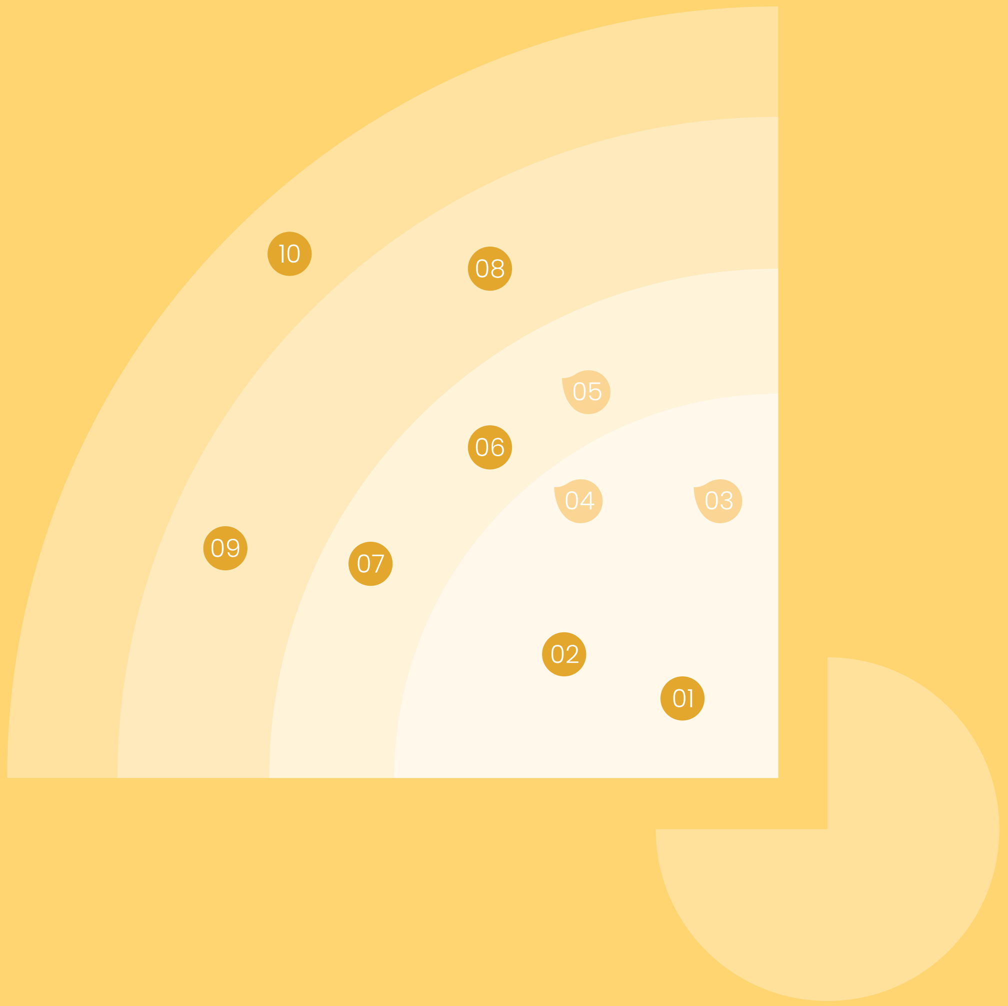
This exchange allows us to evaluate the project in relation to the 3 technologies and to make a recommendation, strong depending on the constraints.

In short, we can't say with certainty that "technology A is better than technology B". The decision must be made on a case-by-case basis, with input from all stakeholders and mobile experts who know the different solutions.

We'd love to sit down with you over coffee and give you a personalized recommendation for your app and your business.

QUADRANT

React Native



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

● New ● Move ● No change

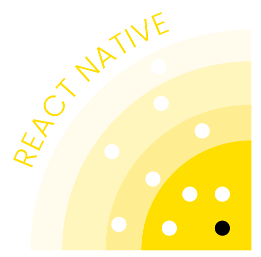
In this edition of Tech Radar, we highlight two major trends in the React Native ecosystem: the **growing adoption of Expo** and the rise of **universal applications**.

Now recommended by Meta for the development of new React Native applications, Expo has become a must-have solution. For many projects, migrating to Expo represents a significant gain in terms of **quality**, **productivity** and **maintenance**, making the use of this framework essential for all React Native projects.

At the same time, universal applications are booming thanks to tools like Solito and Expo Router. These solutions make it possible to **share a code base between web and mobile platforms**, reducing development costs and complexity. Code sharing can also be extended to **TV and desktop** platforms, which are benefiting from an increasingly rich React Native ecosystem.

As these tools continue to evolve, React Native's "learn once, write anywhere" tagline has never been more relevant, facilitating the development of high-performance, cross-platform applications.

BY **CYRIL BONACCINI**
Staff Engineer



ADOPT
1/10 blips

❶ Migrate to Expo

Maintaining the "native" part of React Native applications can be time consuming. Major releases of React Native often take several days and introduce bugs due to incompatibilities with other native packages in use. Updating the latter is also a tedious process.

Expo is the solution that makes it easier to build and maintain React Native applications. Expo's prebuild system, called Continuous Native Generation (CNG), automates the generation of native code from configuration files. Config plugins integrate specific native modifications, while the Expo Modules API simplifies the creation of native modules. Expo Application Services (EAS) makes it easy to build and deploy mobile applications. In addition, Expo Updates offers an overthe-air (OTA) update solution for sending updates without having to go through the stores.

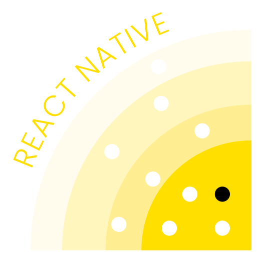
Meta officially recommends the use of Expo for new React Native applications, as stated on the React Native website: "To build a new app with React Native, we recommend a framework like Expo". What's more, with the scheduled closure of AppCenter by Microsoft on March 31, 2025, **Expo Updates remains the only viable solution for OTA updates**.

Many projects never left the ground with Expo because the solution wasn't always as complete and stable as it is today. Fortunately, **it's possible to migrate to Expo gradually**. First, by configuring a new Expo application with all the dependencies of the existing application so that the JavaScript code can be executed. Then, by updating the deployment processes to use EAS and Expo updates. Finally, by migrating to Expo modules to benefit from higher quality modules that are better maintained and automatically updated when the Expo SDK is upgraded.

At Theodo, we've seen **significant gains in development and maintenance time** by using Expo in our projects. The ease of updates, improved productivity, and robustness of the Expo ecosystem make it a must-have solution for React Native projects. Despite some migration challenges and support limitations on certain platforms, **adopting Expo is a strategic decision that will pay off in the long run**.

OUR PERSPECTIVE

We strongly recommend planning to migrate your project to Expo to simplify upgrades, increase productivity, and improve maintenance of React Native projects.



ADOPT
2/10 blips

2 Solito

Universal applications are gaining in popularity, allowing developers to target iOS, Android and the web with a common code base. This trend raises the question of the best technology to handle this multi-platform approach. At Theodo, we use two frameworks depending on the project: Expo Router and Solito.

As its documentation indicates, Solito is both:

- A **library** that **bridges the gap between React Navigation and Next.js**.
- A **CLI** for creating a project with a **monorepo** containing an Expo application and a Next.js application, where Solito is used to **navigate the shared code**.

The main advantage of Solito over Expo Router is the ability to use the advanced features of Next.js. **These include enabling serverside rendering (SSR)**, using **server components**, improved **font and image management**, and **better support for internationalization**. For applications where performance and SEO are critical, this is an undeniable advantage. Monorepo also provides a clear separation between web- or mobile-specific code and shared code, which is especially interesting if certain features of your application are web- or mobile-only. Another interesting use case is if you already have an Expo application and a NextJS application, and you want to share code between them. In this case, the applications can be grouped under a single repository and the mobile components can be shared incrementally.

However, using a monorepo with an Expo application and a NextJS application presents some challenges compared to a single application under Expo Router. While this can introduce some complexity, such as **the need to maintain two separate navigation systems**, this approach also allows for greater flexibility. Page creation is still fast, but to ensure a consistent experience between the mobile app and the website, it's important to synchronize the page structure between the two routers.

Solito makes it much easier to navigate the shared code, although **the lack of typing and autocompletion for URLs** requires special attention to avoid errors.

OUR PERSPECTIVE

Solito is an excellent option for universal application projects where SEO and web performance are critical. For applications that don't need these features, it's easier and less expensive to use Expo Router.

ADOPT
3/10 blips

❸ Suspense for data fetching

We talked about Suspense in the last Tech Radar and recommended its activation. The major advantage of using **Suspense with a data fetching solution** like React Query is the **simplification of the components**, whose loading states and error handling we no longer need to specify. This **not only improves maintainability, but also UX**, since it encourages the placement of loaders and error inserts.

There is, however, one aspect to be aware of when switching to Suspense. If several data fetching hooks using Suspense are placed end-to-end, the asynchronous tasks will execute one after the other, even if they are independent, a phenomenon commonly referred to as waterfall calls. The solution we recommend is to **distribute hooks as far down the React tree as possible**, so that the component being suspended is as small as possible. As the components are suspended in parallel, this ensures asynchronous parallel calls. To improve time-to-interactivity (TTI), it is also possible to **prefetch requests**.

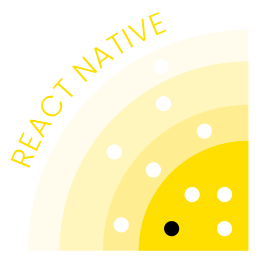
Recently, the React team released a change that completely broke React Query* with Suspense (all calls became waterfall, even when applying the above approach).

However, that change was cancelled and it delayed the release of React 19. This makes us particularly confident that this way of using Suspense is indeed **a practice recognized and validated by the React team**.

OUR PERSPECTIVE

With all the UX and maintainability benefits offered by Suspense for data fetching (with React Query or other libraries supporting it), we strongly recommend switching to Suspense. We also recommend migrating existing projects.

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



ADOPT
4/10 blips

❹ React Native Web

Since an app is often available on both web and mobile, it's natural to want to share code between these platforms, which is possible for business logic, UI state, and API calls. However, UI components cannot be shared by default.

React Native Web is a React DOM-compliant implementation of React Native components and APIs.

We've been able to share between 75% and 95% of the code between React Web and React Native applications using React Native Web on several projects, and the feedback has been very positive. Variations in the amount of code shared depend on the amount of web and/or mobile-specific functionality, such as page layout.

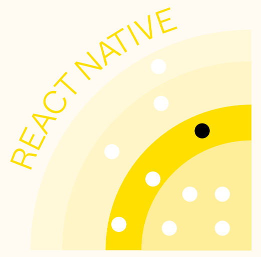
In our previous Radar, we reported on performance and accessibility issues. A lot of work has been done since then, both on react-native and react-native-web, as well as with community libraries like Tamagui and Expo Router.

React Native Web is not a magic solution to all codesharing problems, and the remaining points have more to do with the way we work than the technology itself:

- Page layout can differ between web and mobile, so it's often better **to share certain components** of a page rather than the entire page;
- It's important to consider code sharing **at the design stage** to maximize the benefits.

OUR PERSPECTIVE

We are now using React Native Web on all of our projects that require code sharing between web and mobile. This adoption confirms our confidence in the technology and we now recommend it as a reliable solution for developing applications for these platforms.



TRIAL
5/10 blips

5 Expo Router

Developing web applications requires taking into account constraints that are generally unknown to mobile developers: bundle size, direct access to any page via a URL, server-side rendering...

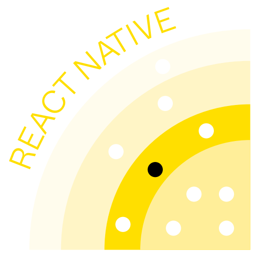
In order for a framework to offer the best performance with respect to these constraints, it must directly manage the bundling process, navigation and, if necessary, data fetching. This is what Next. js, Astro, and Remix do.

Expo-router is a universal navigation framework for React Native apps that, thanks to its integration with other Expo tools, promises an optimized experience on both iOS/Android and the web. Specifically, exporouter provides **file system based routing** (creating a file in the `app` folder defines a new route) and is an overlay for React navigation.
The web platform is officially supported, with URL management, bundle splitting to optimize performance, and the ability to include CSS. This is the main advantage of Expo Router.

In use, **Expo Router's youth is still noticeable**.
For example, the navigation typing is experimental and incomplete (it doesn't cover search parameters), and the seemingly simple API complicates the implementation of certain scenarios that weren't a problem for developers familiar with React Navigation. The choice of an `app` folder where all files become routes means that 2 parallel trees must be maintained in the codebase, whereas other frameworks go back to this constraint and even the very idea of "filesystem-based routing".

OUR PERSPECTIVE

We have decided to put **Expo Router for universal apps** in Trial because it's the best choice for deploying a React Native app on the web. However, the performance is not yet "best-inclass" (no server-side rendering, no treeshaking yet), so in certain scenarios an overlay, for example with Next.js and Solito, is still necessary.



TRIAL
6/10 blips

6 Keyboard Controller

Almost all mobile applications allow the user to enter text and therefore need to manipulate the iOS and Android virtual keyboard. Even the most popular apps often have bugs, or a suboptimal user experience related to this keyboard management.
In fact, it's a more complex problem than it seems. And in the case of React Native, platform differences don't make it any easier.

The React Native package itself **provides limited APIs** to deal with this problem, so one developer decided to tackle it and released the React Native Keyboard Controller.

This library provides **a unified API between iOS and Android** that includes all the necessary functionality: keyboard open/close with UI animation (integrated with reanimated), text field management, placement of fixed buttons above the keyboard...

React Native Keyboard Controller is a recent library and may have some instabilities, but it's already the **most complete solution in its field**, hence its position on our trial dial.

OUR PERSPECTIVE

We've started using this library in our projects and encourage other developers to do the same.

TRIAL
7/10 blips

❼ React TV

Space Navigation

One of the biggest challenges in developing a **TV application is managing the focus on the remote control**. Solutions are not consistent across platforms. React Native tvOS provides an implementation, but it's not completely predictable between tvOS and AndroidTV, and it doesn't yet exist for the web. This makes it difficult to build universal applications with React Native.

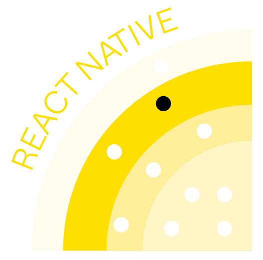
At Theodo, we've developed React TV Space Navigation, a library that reimplements navigation in React without using the native focus, **ensuring consistent behavior across platforms**. It provides a React-friendly API with purely declarative components and includes optimizations like virtualization and CSS scrolling to **improve performance**, especially on less powerful devices.

However, bypassing native focus has limitations: **accessibility support is incomplete**, and some specific features, such as "ploc" on Android, are missing. Despite these drawbacks, we have successfully deployed a large cross-platform streaming application using this library. This application is very smooth to use.

We are watching the development of **`react-native-tvos`**, which may one day make our library obsolete. With proper component isolation, it's easy to revert to the default **`react-native-tvos`** management, but this would remove support for the web, which has no builtin focus management.

OUR PERSPECTIVE

For now, we recommend trying out the library for your cross-platform TV projects, keeping in mind the limitations of this solution, which comes with certain compromises in terms of accessibility and native feel.



ASSESS
8/10 blips

❽ React Server

Components for mobile

In server-driven UI, the server sends a precise description of what should appear on the screen, rather than raw data. Some applications use this approach to respond to constraints such as the need for rapid iteration, heavy UI customization, or application size limitations.

In March 2024, the React teamannounced **an official serverdriven UI mechanism called "React Server Components" (RSC)**. These components can be "composed" with "client components". React and the framework it uses are responsible for managing the interactions between the two worlds (creating the right javascript bundles, reconciling the client-side component tree, etc.).

The only role of the mobile application is to display the transferred UI.

Because server components run on a server, they can directly access secrets, databases, and the file system. This promises to radically simplify the development of certain functionalities. It also avoids client-server round trips, which improves performance. So far, there is **no integration of Server Components in a React Native framework**, but Expo is working on it. A React Native implementation of Server Components will face several challenges:

- You'll need to make sure that the native components generated on the server side are available on the native side, otherwise you'll **crash**.
- Mobile applications often need to operate, at least in part, **offline**.
- The "transparent" composition of client and server components is in fact limited by subtle rules that define which component can import and/or receive in prop which other component (this limitation also applies to the web).

OUR PERSPECTIVE

We invite you to keep an eye on how the React Native Server Components ecosystem evolves, so that you can use them as soon as possible, where relevant.

REACT NATIVE

ASSESS
9/10 blips

9 Unistyles

Many style libraries have emerged in response to the growing popularity of universal applications. These include Tamagui, Nativewind, and Unistyles. With the Web came new performance and server-side rendering (SSR) issues, making solutions more complex and difficult to compare.

Unistyles provides an API close to that of StyleSheet, but with additional features: accessible theme in style, **variants, dynamic styles, breakpoints**, access to insets and screen size. The core of the library is written in C++ and **offers similar performance to StyleSheet**.

The advantage of Unistyles is the **simplicity of its API**. Defining a custom theme is easy because the structure is not imposed, unlike Tamagui.

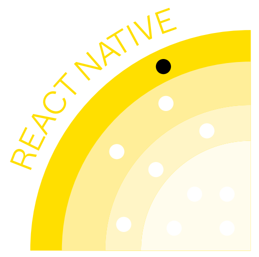
Its proximity to StyleSheet makes it easier for React Native developers to get started, while those used to Tailwind will be more familiar with Nativewind.

The library is compatible across many platforms: web, Windows, MacOS, visionOS, and TV. However, **support for SSR is not yet complete**, since responsive styles don't use media queries. Version 3 should include a compiler, such as Nativewind or Tamagui, **to generate CSS at build time**.

Unistyles is a new project, but it's very active. The project benefits from the commitment of its main contributor, Jacek Pudysz, which allows it to move forward quickly. However, this dependence on a single individual may raise questions about its sustainability.

OUR PERSPECTIVE

Unistyles is a very promising solution for universal applications. It is easy to use and offers many practical features. However, it still lacks maturity, especially in terms of web support, which is not yet on par with Nativewind. These shortcomings should be fixed with the release of V3.



HOLD
10/10 blip

10 react-native-svg as a default

SVG is a vector-based image format in which the image is dynamically drawn. This format has some interesting properties: it is never blurred and can inherit color or shape variations. However, drawing dynamically comes at a performance cost, especially if the rendering engine is not tuned or if the image is complex.

The `react-native-svg` library is often used to draw SVGs, recreating each SVG shape in the React view so that it can be **dynamically modified or animated**. However, this transformation can be slow, especially for icons, which are often numerous on a single screen and slow rendering.

There are better solutions for icons and static SVGs. For icons, icon fonts can be used. This means converting SVGs to characters in a font. The limitation of this solution is that the SVG must be monochrome, a typical feature of standard icons. A common method is to use IcoMoon in combination with @expo/vector-icons.

For other illustrations, **`expo-image`** is a good alternative. This library handles SVGs and displays them better. But it's impossible to animate the SVG and adjust its colors (unless you apply a tint to the whole SVG container, but that changes the whole image).

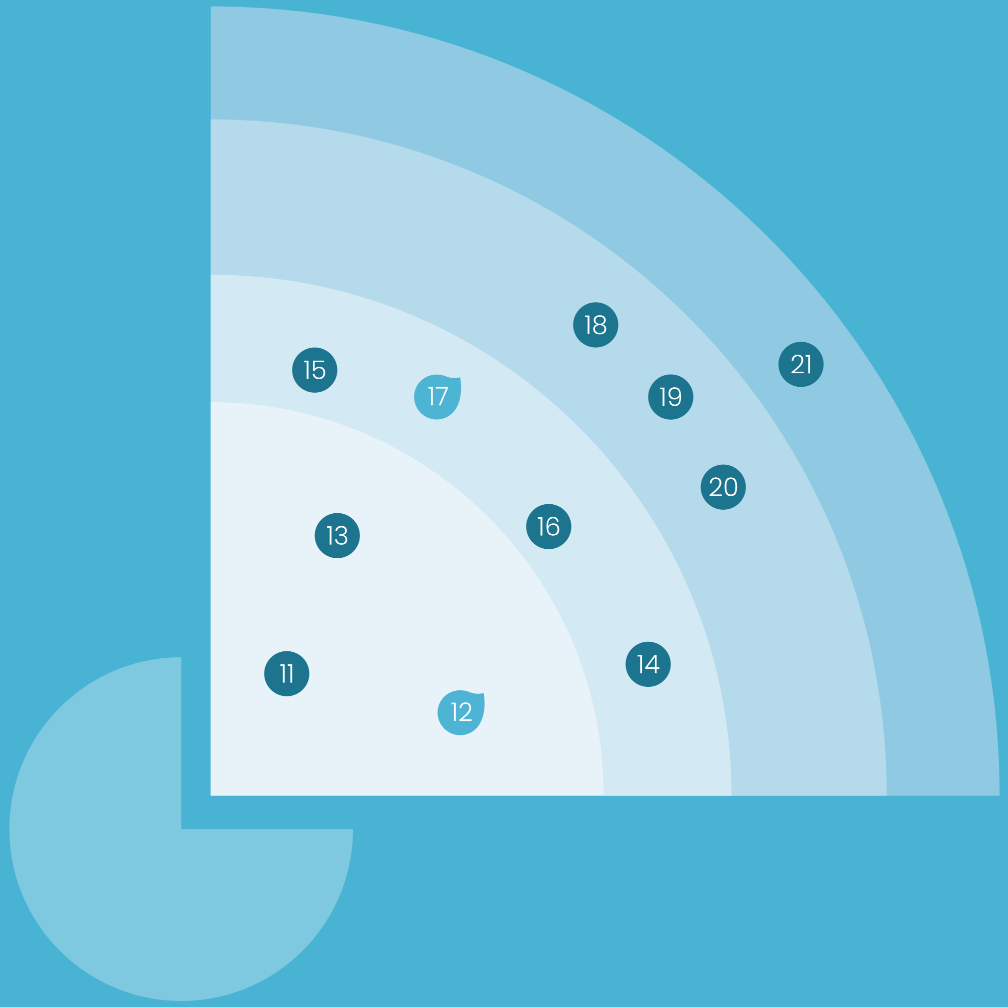
These alternatives cover a large part of SVG needs, since it's rare to need to partially change the color of SVGs.

OUR PERSPECTIVE

We recommend not using react-nativesvg by default for all SVGs. It is preferable to use solutions like icon fonts for icons or expo-image for static SVGs to improve performance. However, it is still important to use react-native-svg for dynamic SVGs

QUADRANT

Flutter

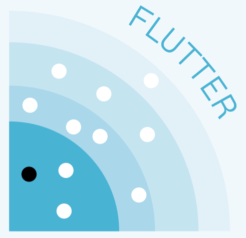


11 BLIPS | 3 ADOPT | 4 TRIAL | 3 ASSESS | 1 HOLD

● New ● Move ● No change

In this third edition of our radar, we present the new generation of Flutter technologies. Flutter breathes new life into issues already addressed by some of the more established solutions in the ecosystem, such as **`flutter_test`**, state managers, and **`hive`** storage. New solutions for rendering high-performance maps, managing forms, or persisting data are emerging, improving the development experience and the robustness of applications. This handout provides an overview of these small developments with big impact.

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



ADOPT
1/11 blips

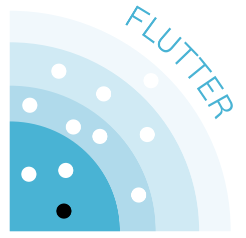
⓫ checks

When unit testing in dart, the basic solution for checking that a variable has the expected value is to use the `flutter_test expect` function, which takes a variable and a matcher. The main problem with this function is the **dynamic typing of its arguments**. If the type of the variable does not match the type of the matcher, **an error is raised only during test execution**, without any indication to the developer during test writing.

The `checks` package exposes several methods for checking the value and type of variables in the equivalent function and provides support for deep equality of collections. The available methods are automatically filtered by the type of variable in the function. In addition, Checks **enhances the development experience** by providing filtered autocompletion directly on the provided methods in modern IDEs. All check methods can be called in a chain, a **declarative approach** that promotes code clarity and readability. The library also provides the ability to expect `Futures` or `Streams` values. Finally, **the ability to customize** tests removes limitations by allowing you to combine existing tests or create logic entirely specific to your context.

OUR PERSPECTIVE

Although little known, the `checks` library offers significant benefits **that make testing more explicit and readable**. That's why we make it our default solution for Flutter projects and strongly recommend its adoption.



ADOPT
2/11 blips

⓬ Riverpod

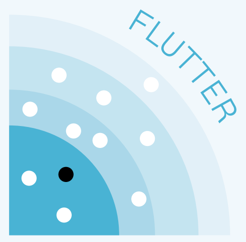
In Flutter, global state management is achieved through `InheritedWidget` and `ChangeNotifier`, but these APIs have several shortcomings, including verbosity, complexity, difficulty in testing and the impossibility of creating multiple states of the same type.

Riverpod is a **reactive caching library** that solves these problems. Inspired by `react-query`, it proposes to serve data via providers, which are declared outside the widget lifecycle and can automatically rebuild widgets that listen to them. Providers can serve **asynchronous data** coming, for example, from a call API or a local database, **handle errors** and **caching**, and easily define other functionalities such as debounce or pull-to-refresh.

Riverpod is also **compilesafe**, offers a declarative API, is actively maintained, and is supported by a vast community (6k stars on github, 98% popularity on pub.dev). Since last year, we've been able to experiment with its code generation tools, which enable hot reloads to update a provider and **further reduce API verbosity and complexity**.

OUR PERSPECTIVE

We recommend using Riverpod, which we've been using for 3 years on Flutter projects of all sizes. The announcement of riverpod 3, which should make it possible to define and reuse providers with generic types as parameters, only reinforces our enthusiasm.



13 sliver_tools

ADOPT
3/11 blips

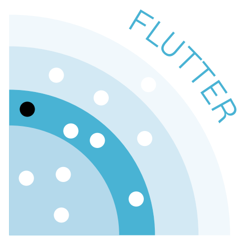
In Flutter, slivers are a type of widget that integrate into scrollable views and react to scrolling to create complex, animated scrollable screens. While slivers are not particularly difficult to use, they are low-level widgets and their writing is complex.

While the framework offers several very high-level slivers, such as the `SliverAppBar` and the `SliverList`, it can be **difficult to customize scrollable views** beyond the classic ones on offer. The `sliver_tools` library offers a **collection of** ready-to-use **slivers** that enhance the framework's native slivers, providing **an intermediate level of flexibility** between high-level widgets and low-level `RenderObjects`.

Among the most frequently used are `MultiSliver`, which lets you combine several slivers into one to improve code quality by cutting out responsibilities, and `SliverPinnedHeader`, which lets you create scrollable elements that snap to the top of the scrollable view to keep them visible and give a pleasant navigation effect.

OUR PERSPECTIVE

Our use of `sliver_tools` on projects has been very conclusive, with no reported limitations. This toolbox is now part of our standard stack at Theodo, enabling us to **bring to life the original scrollable views** imagined in collaboration with our designers.



14 flutter_map

TRIAL
4/11 blips

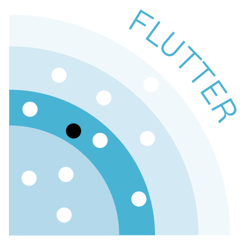
Mapping in mobile applications is a complex subject due to the performance and limitations imposed by certain libraries for displaying various elements. Traditionally, developers have turned to solutions such as Mapbox and Google Maps, which use a C++ graphics engine to render maps. While these solutions are robust, they **don't always integrate seamlessly** with Flutter, particularly when it comes to customizing the elements to be displayed on the map.

The `flutter_map` library has been written entirely in Dart, with **a declarative and composable API** for UI elements, combined with **an imperative approach for manual control** (e.g. animations). This approach enables developers to take full advantage of Flutter's benefits, notably by easily integrating widgets on the map. It also benefits from **a varied ecosystem of open-source extensions**. We've used `flutter_map` to create highly customized raster maps and have been impressed by **its ease of use and performance for raster maps**.

`flutter_map` does, however, have some **important limitations**. The extension for vector maps suffers from **major performance problems**, making it impractical for this type of map. Transitions between zoom levels also lack fluidity (compared with the Mapbox or Google Maps SDKs). This limitation has prompted us to opt for **alternative solutions for projects requiring vector maps**. To solve these problems, the community is actively working on a solution using Flutter's latest advances, such as Impeller or `flutter_gpu`.

OUR PERSPECTIVE

We recommend that you try `flutter_map` for integrating raster maps into your Flutter projects, due to its ease of use and performance. However, we invite you to remain open to other available solutions, especially when using vector maps.



TRIAL
5/11 blips

15 mmkv

There are several local database solutions available in Flutter, including SharedPreferences, Isar, Hive, ObjectBox, SQLite and others. Each of these solutions has its own advantages and disadvantages in terms of performance, ease of use and functionality. In this context, we have integrated MMKV for Flutter on one of our projects. MMKV is a **high-performance, easy-to-use key/value storage library** developed by Tencent. Used in the WeChat application, MMKV is designed to offer optimum performance using **`mmap`** and **`protobuf`**, enabling memory to be synchronized with files and values to be encoded/decoded efficiently.

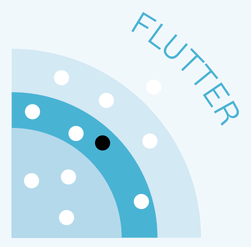
By using Dart FFI for synchronous read and write operations, we can take full advantage of this Flutter performance. We have measured that opening an MMKV database is much more **CPU-efficient** than other solutions, such as Isar, which can block a thread for several hundred milliseconds. MMKV's API is **clear and minimalist**, making it easy to integrate and use in Flutter projects. Since the migration to a federated plugin architecture, the development experience with MMKV has improved significantly. In addition, MMKV supports **data encryption**, providing an extra layer of security.

However, MMKV for Flutter does have a few limitations. Currently, **only iOS and Android platforms are supported**, although MMKV itself is available on iOS, Android, Linux, macOS and Windows. Future web support could prove complex to implement. In addition, a recent minor update removed support for Android ARM7 and x86 architectures, impacting around 2% of our production users.

Despite these challenges, we find MMKV to be a very interesting solution. It's simple and extremely powerful, developed by Tencent, which makes it highly reliable and easy to maintain over the long term. Although very popular in the React Native and Native communities, MMKV is still relatively unknown in the Flutter ecosystem, probably due to the presence of already established synchronous local databases.

OUR PERSPECTIVE

We recommend testing MMKV in your Flutter projects. However, **a thorough assessment of the platforms and devices** used by your users is essential to ensure successful integration and compatibility with your project's specific requirements.



TRIAL
6/11 blips

⓰ Reactive Forms

Forms management and user input validation are crucial but complex aspects of application development. Reactive Forms, a library for Flutter, offers **model-based form management** inspired by Angular. At Theodo, we use it for projects requiring forms, such as login, registration or payment forms.

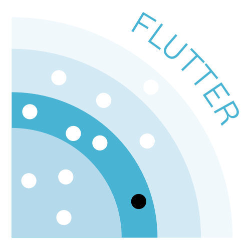
Reactive Forms boasts a **rich** ecosystem of **predefined, asynchronous and customizable validators**. This flexibility makes it easy to manage complex, project-specific validation rules. However, it can be difficult to get to grips with, and the code required to define a form is **sometimes verbose**. The typing system could also be improved. An extension using code generation is currently under development to simplify this functionality.

Reactive Forms integrates well with state management tools such as Riverpod or BLoC, enabling efficient reaction to form state changes. It also enables **unit testing of each validation rule**, ensuring that no regression is introduced during updates. This **improves application productivity and maintainability**.

Reactive Forms' documentation and community support are excellent, with an active community and regular updates. The package has 458 stars on GitHub and 839 likes on Pub, and the ecosystem of community-created validators is a major asset.

OUR PERSPECTIVE

We recommend trying out Reactive Forms for your Flutter projects because of **its robust validation capabilities**, flexibility and seamless integration with state management tools. However, given the initial complexity and verbosity of the code, you should proceed **with caution**.



TRIAL
7/11 blips

⓱ Shorebird

Update times are a major challenge in mobile development, as each version must be validated by the stores, delaying the availability of urgent patches to users. What's more, you must wait for each user to update the application. Shorebird, announced in early 2024 by Flutter creator Eric Seidel, is an open-source solution for **deploying Flutter application updates over-the-air** (OTA) without going through the stores. This approach enables minor updates to be deployed directly via their servers, **accelerating the development and deployment cycle**.

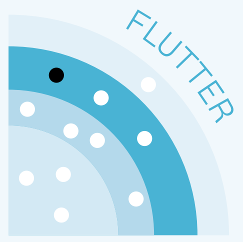
Integrating Shorebird into a Flutter project is straightforward but has its drawbacks. Cost can be a barrier for some teams, and the shorebird patch command is slower than flutter build, which can **slow down the update process**, especially in a QA environment that's active several times a day. Currently, Shorebird **only supports iOS and Android**, excluding desktop applications. In addition, the application has to be restarted to run the updated code, which can affect the user experience.

Despite these drawbacks, Shorebird offers a **valuable feature** as the **only OTA update solution for Flutter**. This tool meets a crucial need in mobile development, and respects store rules by updating only interpreted code.

Although Shorebird's CLI has recently reached version 1, it still presents a few instabilities, but these are quickly resolved as the **development team is attentive to feedback from the community**.

OUR PERSPECTIVE

Shorebird shows **promising results** and has the potential to transform release management in Flutter applications. We recommend testing this technology to assess its impact on your development and deployment cycles.



18 Dart Macros

ASSESS
8/11 blip

During development, we rely on code generation tools such as **`freezed`**, **`json_serializable`** and **`build_runner`**. These tools, while essential for addressing recurring issues such as deserialization, immutability or deep equality, introduce inefficiencies and detract from the developer experience by **imposing frequent manual code generation**. They also tend to clutter up our projects with generated files.

Dart macros, currently in beta, promise to transform this dynamic by integrating **metaprogramming directly into the compiler**. This allows developers to generate code as they write. They **thus improve productivity by reducing repetitive generations** and maintaining a cleaner code base.

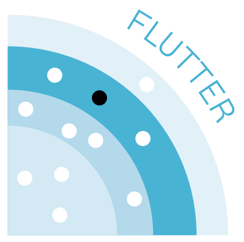
Macros, however, could compromise Dart's simplicity and readability, which are essential to its ease of learning. Dart is renowned for being a "boring" language in the good sense of the word: predictable and stable. The introduction of macros, by contrast, **introduces a dimension of complexity** not traditionally associated with Dart.

If macros seem to threaten this transparency, the proposed augmentation system allows developers to **visualize augmented code with a single click** in their development environment, thus maintaining a clear understanding of the code being executed.

Note that macros will be used mainly by library maintainers, such as **`freezed`** or **`json_serializable`**, and not by all Flutter developers on a day-to-day basis. We'll be keeping a close eye on how maintainers and the community welcome and make use of this new feature.

OUR PERSPECTIVE

We're optimistic about the potential of Dart macros to simplify and improve code generation in our Flutter applications. Although the macros are **currently in the experimental phase**, the Dart team's roadmap calls for a stable version in early 2025.



19 Patrol

ASSESS
9/11 blip

At Theodo, we believe that automated testing is one of the best ways to prevent functional or visual regressions in mobile applications. However, existing Flutter testing solutions, such as unit tests and widget tests, do not always comprehensively cover endto-end (E2E) scenarios.

Patrol[1] is a Flutter library that **simplifies the writing and execution of E2E tests**. It integrates an API that **makes it very easy to interact with the test device's native functionalities**, such as permissions, notifications or parameters. The patrol_finders[2] library, which is now independent of Patrol, also offers syntactic sugar, making test writing more intuitive and less error prone.

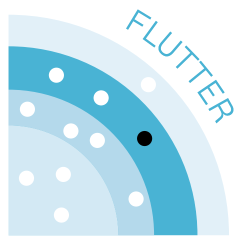
At Theodo, we approach automated testing by using unit tests instead, as well as adaptive UI tests (also known as golden tests) thanks to tools like adaptive_test[3]. They guarantee complete functional coverage while visually verifying the user interface.

For E2E testing, **Patrol offers an extensive API** but will only work with Flutter applications. For this need, our current choice is Maestro, a proven technology whose learning we can share with the React Native and native iOS/Android teams. However, we find that **Patrol solves the problem of E2E testing in Flutter in an interesting and intuitive way**.

OUR PERSPECTIVE

For Flutter developers looking to improve their E2E testing, Patrol is worth exploring, especially if you want to use a tool that integrates seamlessly with your existing Flutter tests. Although our current choice is Maestro, we'll continue to follow Patrol's development.

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



ASSESS
10/11 blip

20 Signals

Managing the overall state of an application is critical to its maintainability and performance. Several solutions have been proposed for Flutter, and one new option deserves our attention: Signals.

Signals is an innovative library for Flutter **that simplifies state management with reactive signals**. Inspired by reactivity concepts from the JavaScript ecosystem, such as preact, it provides fine-grained reactivity, where each signal represents a value encapsulated in a reactive shell. Signals can be simple states or computations derived from other states, forming an acyclic graph of dependencies.

Signals offers several key advantages:

- **Fine-tuned reactivity**: signals automatically track dependencies and release them when they are no longer needed.
- **"Lazy" evaluation**: signals are calculated only when they are read, thus optimizing performance.
- **Flexible PLC**: Allows multiple signal composition with reduced PLC surface area.
- **Surgical rendering**: Only the necessary parts of the widget tree are updated, improving performance.
- **Dart Native compatibility**: Supports Dart JS, Shelf Server, CLI, VM, Flutter (Web, Mobile and Desktop).

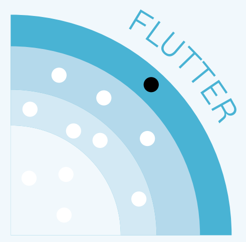
The Signals API is like that of Riverpod, but with a more advanced push-pull system. Unlike Riverpod, which also handles reactive dependency injection, **Signals focuses exclusively on state management**.

To scale a Flutter application, you need to use a dependency injection solution such as InheritedWidget, Provider or GetIt.

At Theodo, we're exploring the potential benefits of Signals, a promising technology that could unify state management practices in the currently fragmented Flutter community. Although Signals is still **relatively new and not widely used** in largescale production projects, its **growing popularity in other web frameworks** indicates a convergence towards a standard reactive programming technique.

OUR PERSPECTIVE

We encourage you to experiment with Signals, bearing in mind that it **hasn't yet reached the level of adoption** of established solutions like Riverpod or BLoC. Keep an eye on this package for future developments and integration potential.



HOLD
11/11 blips

21 isar

Local data management is especially important for applications that need to operate offline or that require a high level of confidentiality and security. An efficient database ensures optimal data management, fast access, and a smooth user experience.

Isar is a **fast, easy-to-use NoSQL database library** designed specifically for Flutter applications. Designed to replace Hive, a key/value database widely used in the Flutter ecosystem, Isar promises high performance thanks to its engine written in Rust. It offers advanced features such as composite indexes, asynchronous operations, and crossplatform support (iOS, Android, desktop). However, we have identified a few issues that warrant a cautious approach.

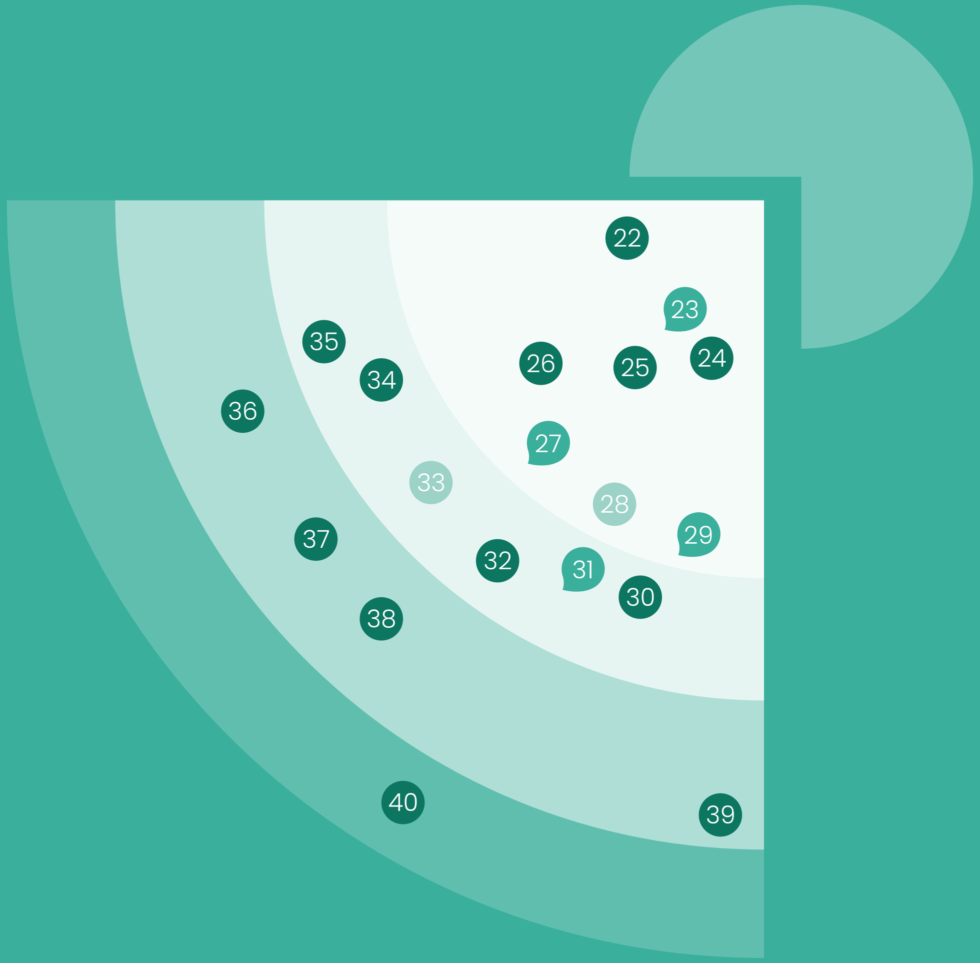
The last period of significant activity on Isar's GitHub repository was a year ago, **raising concerns about its development and maintenance**. In addition, internal testing conducted as part of our Kaizen initiative has shown that **the time required to open encrypted data with Isar can slow the opening of an application** by several hundred milliseconds. This limitation, which occurs at a critical moment in the user experience, is not present in other more mature solutions such as MMKV. Therefore, despite the promise of high performance guaranteed by its Rust engine, Isar has speed and reliability **limitations that currently block a large-scale production application**.

OUR PERSPECTIVE

We recommend choosing solutions other than Isar to manage local data storage, such as MMKV or ObjectBox. Although the Isar library is promising, it's best to wait until it reaches a higher level of maturity and stability before using it for production projects.

QUADRANT

Natif



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

● New ● Move ● No change

Native technologies are characterized by maturity while remaining a fertile ground for innovation. Over the past year, we have seen a **determination by all players to reduce their historical disadvantages**, such as long compile times and complex configuration files. **Today, that transformation seems largely complete**. The developer experience is richer and more fluid, maximizing the value delivered to end users.

And the innovation doesn't stop there: **the Android ecosystem is launching an assault on universal apps** with its multiplatform tools, while the iOS ecosystem seems determined to innovate in large chunks with **Swift Concurrency** and **Swift Testing**.

The future of native mobile is promising and exciting, and it's a horizon we invite you to explore in this issue.

BY **LOUIS PRUD'HOMME**
Tech Lead

ADOPT
1/19 blips

22 Koin

The dependency injection pattern is widely used to apply the principle of control inversion. Indeed, this pattern contributes to greater code modularity, improves maintainability, and facilitates testing. However, implementing dependency injection can be tricky.

There are several dependency injection frameworks available for Kotlin, including Koin, Dagger-Hilt, Kodein, and kotlin-inject. Koin seems to offer many advantages. Let's see why by comparing it to Hilt, the library recommended by Google for Android.

Hilt uses annotations to specify the attributes to be injected and how to construct them. Although Koin also supports annotations, it also allows dependencies to be configured using a domain-specific language (DSL) that is **intuitive and easy to master**. This configuration is isolated in a file separate from the business code, facilitating the separation of responsibilities.

Koin and Hilt use different approaches to dependency management. Koin performs injection at runtime as a service locator, instantiating all dependencies and providing the necessary references to classes. In contrast, Hilt performs injection directly at compile time, and its annotations have been compatible with Kotlin Symbol Processing (KSP) since November 2024.

Since Koin must resolve dependencies at runtime, this has an impact on performance, but the difference in performance is negligible.

Hilt, on the other hand, detects errors at compile time. However, Koin, which only detects them at runtime, has not yet said its last word, as it is possible to check Koin's configuration in **unit tests** and thus avoid deploying a faulty configuration. As far as Kotlin Multiplatform (KMP) is concerned, Hilt is only compatible with native Android, while **Koin is multi-platform**, offering a notable advantage. Since Google I/O, it seems that the compatibility of Google libraries with KMP is only a matter of time. It remains to be seen whether Hilt will be easily adaptable to KMP.

OUR PERSPECTIVE

In conclusion, although Koin is not the inversion of control library recommended by Google, its ease of use and multi-platform compatibility make it a preferred choice for our new projects. For an existing project using Hilt, a migration to Koin should only be considered if a switch to KMP is planned soon.

ADOPT
2/19 blips

23 Gradle Version Catalog

In multi-module Gradle projects, **managing dependencies** and their versions is a **challenge** that raises three main issues. Firstly, dependencies need to be updated module by module. Secondly, it's difficult to check whether a **dependency is obsolete**. Finally, there is no dependency **autocompletion**. Creating a custom solution is one option. You'll find several, but they're unlikely to match the quality of the Gradle version catalog. This approche creates a single, centralized catalog in a minimal file based on **TOML**: here you can store all your dependencies and their versions, including plugins. It's easy to update them as needed.

In fact, the version catalog has one advantage: **your IDE will alert you to new library** versions and can update them for you. We recommend automating these updates with Dependabot or a similar solution. All modules can safely reference dependencies, so your IDE can help you with **autocompletion**, even if you're still using Gradle files in Groovy instead of the new KotlinScript files.

Migrating to the version catalog is **easy and well documented**. It's also the perfect opportunity to switch your Gradle files from Groovy to Kotlin (.kts).

OUR PERSPECTIVE

Although RefreshVersion is another viable solution, Gradle Version Catalog has been standardized by Gradle and Google as the **official dependency management solution**. We recommend switching to this solution for any project that is not already using one of the two viable solutions.

ADOPT
3/19 blips

24 KSP

Developers often use code generation to reduce repetition and **improve code readability**. In Android, annotations are essential. Developers can rely on predefined annotations from libraries such as Room, Hilt, or Glide, or create custom annotations to generate specific code. Historically, the Kotlin Annotation Processing Tool (KAPT) **has been the standard** in the Android community. It allows Java annotations to be used in Kotlin code, ensuring compatibility with Java libraries.

Kotlin Symbol Processing **(KSP)**, developed by JetBrains and Google, aims to overcome the limitations of KAPT by processing Kotlin annotations directly **without converting them to Java bytecode**. This approach is **twice as fast as KAPT** because it eliminates compilation steps. What's more, KSP adapts better to the specifics of the Kotlin language, such as default parameters, coroutines, and data classes.

Many libraries that previously depended on KAPT are now migrating to KSP, and most have already done so. This is a godsend because **KSP supports Kotlin multiplatform**. So, developers who want to migrate their projects from Android to Kotlin Multiplatform should use KSP instead of KAPT.

If you don't use custom annotations, **the migration is straightforward**. Each library that requires KAPT provides detailed instructions for switching to KSP.

OUR PERSPECTIVE

Although KAPT is still a viable option, KSP surpasses it in every way. It provides better support for Kotlin-specific features and is twice as fast. We strongly recommend switching to KSP

ADOPT
4/19 blips

25 SF Symbols

A recurring frustration for developers with raster images such as PNGs is their lack of flexibility: manual resizing, creating multiple versions for each context or color. Vector images, while superior, also require specific manipulation or versions for each use.

Apple's SF icons provide an elegant solution to these problems. They are vector icons that **seamlessly** integrate with our application's text, greatly improving the development experience and maintainability. Their flexibility is remarkable: you can adjust the thickness of the stroke, change the color, and make them multicolored, which simplifies the work of designers and ensures better integration into the application's design system. What's more, thanks to the rendering engine shared with the fonts, **their performance is optimal**.

One of the major advantages of SF Symbols is their harmony with SwiftUI, especially when it comes to animations and integration with hierarchical styles. For developers, this means a **reduction in development time**, and for users, an improvement in the visual quality of applications.

It's possible to create your own SF symbols using the Sketch[1] tool, either directly or from an existing base. However, since the management of edges is not the same as with SVGs, it is preferable to use a specialized converter such as SfSymbolConverter[2], especially if you already have an existing icon set. Alternatively, the 6,000 icons available by default offer a quick and efficient solution to reduce the lead time of functionalities.

OUR PERSPECTIVE

We recommend using this technology for all your native projects. Despite some subtleties in the creation of custom icons, SF Symbols offers undeniable advantages in terms of integration speed, convenience, flexibility and performance.

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

ADOPT
5/19 blips

26 Swift 6 Migration

Despite its advantages, Swift suffers from problems that affect developer productivity. On the one hand, there is a major problem of **slow compilation**, particularly pronounced in large projects, where typing causes the compiler to fail with the message "The compiler is unable to type-check this expression in a reasonable time". On the other hand, **the validity of concurrent code is delegated** to the developers who write it.

Swift 6 provides a solution to these problems. In addition to several new features and syntaxes, one of the most significant improvements in this new version is **the increased compilation speed**. This improvement alone more than justifies a migration, as it can significantly reduce developer frustration and improve productivity.

But Swift 6 also highlights (statically) compiler-tested **concurrent code**, allowing developers to reduce the number of these sometimes hard-to-detect bugs. However, this feature is not without its drawbacks. Static validation implies major changes that disrupt existing code bases, requiring significant refactoring and customization efforts. **This transition can be difficult and resource-intensive**, especially for large projects, despite the gradual migration made possible by the feature flags available in Swift 5.

OUR PERSPECTIVE

Given these considerations, we recommend migrating to Swift 6. Although major changes require careful evaluation of the investment, the benefits of improved compile speed and verified concurrency are substantial.

ADOPT
6/19 blips

27 The Composable Architecture

SwiftUI provides powerful code primitives for externalizing state but requires a welldesigned architecture to take full advantage of their potential and not create gigantic monolithic states. In addition, with a **developer community split between** UIKit, SwiftUI, @Observable, @ObservableObject, and different architectures, it's not easy to find your way around an existing project.

The Composable Architecture (TCA) is a framework designed to solve these problems in iOS applications. Inspired by the **one-way** Flux **architecture** popularized by Redux, TCA adapts this approach to Swift, making state management more intuitive and less verbose than in JavaScript. Unlike libraries like ReSwift, TCA is designed for SwiftUI while remaining compatible with UIKit.

TCA structures state around user actions, ensuring a **clear**, **traceable** flow of changes, facilitating testing, and enabling modular architecture. Functionality can be developed independently and then integrated into a complete application. This formalization of the code improves the development experience and **reduces the learning curve**. What's more, the community[1] around TCA is robust, offering extensive documentation[2] and tutorials to support developers.

The addition of navigation management, as well as the backporting of certain features that were otherwise unavailable on earlier versions of iOS, such as state observability, attest to the library's transformation this year and propel it to the **forefront** of its field.

OUR PERSPECTIVE

At Theodo, we've successfully implemented The Composable Architecture in production projects and highly recommend it for state management in SwiftUI applications.

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

ADOPT
7/19 blips

28 Tuist 4

We've featured Tuist in our previous Tech Radar for its ability to improve iOS project management. Using it simplifies build configuration. Better **management** of build **caches** speeds up compilation times, especially for projects based on modular architecture. However, because of the major changes introduced, this new version may prove divisive.

The first major change is the end of Carthage support: each project will now have to manage the fetching of Carthage dependencies, which may require significant adjustments to existing workflows.

What's more, Tuist 4 no longer supports application signing, forcing developers who previously used this feature to completely rethink their signing workflow.

Another new feature of Tuist 4 is the use of Swift Package Manager (SPM) files instead of a specific format. This transition allows for better integration with other **tools in the ecosystem** (e.g. Xcode or dependabot).

Perhaps this is really the **maturity** phase, as Tuist focuses on what it does well to do it better. Finally, the challenges mentioned above are fortunately made easier thanks to the extensive migration documentation and good community support; this makes it possible to take advantage of the significant improvements in build performance brought about by Tuist 4. The syntax is also even **more intuitive**, which makes the tool easier to use and speeds up development.

OUR PERSPECTIVE

We always recommend using Tuist for new projects. We also recommend migrating to Tuist 4, as the long-term performance and project management benefits far outweigh the initial transition challenges.

ADOPT
8/19 blips

29 µ-Features
Architecture

The growing complexity of mobile applications poses a number of challenges, including the maintainability of a growing code base, increasing build times and difficulties in complying with the test pyramid. To address these issues, companies such as SoundCloud and JustEat have popularized **micro-feature architecture**, which divides the monolith into smaller, more specialized modules.

This architecture is composed of four types of modules:

- Main application: coordinates the application's various functions.
- Functionality modules: manage visual components, navigation and user interaction.
- Business logic modules: responsible for domain-specific services and entities.
- Core modules: interface with external functionalities (e.g. API calls, storage, logs).

The micro-feature architecture offers significant advantages: **it facilitates testing of business logic, reduces build times** thanks to caching, and enables common **code to be shared** when teams expand, or new products are developed. This approach also simplifies upgrades and partial overhauls of complex applications.

However, its implementation requires good domain expertise for efficient slicing, a solid understanding of software architecture principles, and good initial design. Tools like Tuist for iOS* can facilitate this slicing, improve caching, and make the transition to this architecture smoother.

OUR PERSPECTIVE

At Theodo, micro-features have become a benchmark choice for new native projects. We've been able to test just how much this architecture can simplify the upgrading or partial redesign of complex applications.

*https://docs.tuist.io/guides/develop/projects/tma-architecture

TRIAL
9/19 blips

30 Compose
Stability Configuration File

With Jetpack Compose, when the parameter of a composable changes, only the affected part is updated, provided the rest is stable. If a parameter is considered unstable, the entire view will be re-rendered even if its value has not changed. This can **slow down rendering** and increase the load on the UI thread, affecting the performance of complex screens.

A major problem is that **any class originating from a different module is considered unstable by the Compose compiler**. Model classes are often separated from views to respect the separation of responsibilities, and this degrades the stability of composables. You can detect these stability problems with tools such as the layout inspector or the compose compiler report[1]. To solve this problem, three main strategies[2] were used:

- Wrap the template class in a stable local class using the @Stable annotation.
- Map the object to a copy of your model class.
- Add the compose-compile dependency to the model layer to annotate models as stable, which undermines the separation of responsibilities.

Jetpack Compose recently introduced a stability configuration file[3], a declarative approach to managing stability **without modifying code**. This file enables the Compose compiler to treat listed classes as stable.

Using this file is **faster, more elegant and less restrictive** than other solutions. Developers can set up stable classes in a root file or in separate files for each module.

OUR PERSPECTIVE

At Theodo, we see great potential in this solution. Although it is new and may have yet unknown limitations, we recommend a gradual implementation to ensure project compatibility.

[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) tackles a central problem of modern development: avoiding code duplication and the associated costs, such as the multiplication of bugs and inconsistencies between platforms. There are many cross-platform solutions on the market, such as React Native or Flutter. These solutions are frameworks that offer a single code base for creating complete applications that target multiple platforms.

KMP takes a **more flexible** approach. It's not a framework, but a technology enabling the Kotlin language to **compile on platforms** other than the JVM used for native Android development.

This more modular approach makes KMP ideal for **sharing code** between different platforms (in the form of a library) **without imposing** anything.

On iOS, for example, Kotlin code is compiled and generates a library that can be used in iOS projects just like any other library native to this platform. This flexibility means that code can be shared according to need: a simple function, a specific layer of the application (network, business, etc.), a specific feature, or even the entire code. **You only share what you want to share**.

Thanks to KMP, Kotlin benefits from **bi-directional interoperability**: Kotlin code can be compiled and integrated into native code, while existing native libraries can be used in platform-specific Kotlin code, simplifying the bridges often needed to exploit **platform-specific** functionalities.

The KMP ecosystem is booming, with Google actively working to make its native Android libraries compatible with KMP. On the other hand, the community is also very **active**, and the most popular libraries in the Android universe are already compatible with KMP or very close to it, such as Room, Retrofit, Coil, Koin, etc.

KMP is now **stable** on almost all platforms, including Android, iOS, Desktop (Windows, Mac, Linux), and even the web by transpiling to JavaScript/TypeScript. Compilation to WebAssembly is still in alpha. Compilation for iOS is currently based on Objective-C, although Jetbrains announced the imminent arrival of KotlinTo-Swift at Kotlin Conf 2024 in Copenhagen.

KMP is an **ideal** solution for sharing code between different platforms. We encourage you to give it a try: adding a KMP module to an Android project is straightforward (just a few configurations in Gradle) and you can start sharing code. KMP is perfect for creating a cross-platform library. However, if you also want to share the user interface, you'll need to use Compose Multiplatform (CMP), which is not yet as stable as KMP.

OUR PERSPECTIVE

We encourage you to start your new Android projects directly with KMP to guarantee their scalability. This in no way affects the project itself but will make it easier for you to share code in the future.

32 Factory

TRIAL
11/19 blips

In development, it is often difficult to make code testable and maintain a scalable code base due to the strong coupling of dependencies. An effective solution to this problem is dependency injection, which allows components to be decoupled from the code, thus improving its testability and maintainability.

Factory* is a modern library that effectively solves these problems for Swift developers. Key benefits include dependency **injection checking** at compile time, ensuring that no errors occur at runtime due to missing or incorrectly injected dependencies. Factory also allows the use of **injection scopes**, facilitating the implementation of true **inversion of control** thanks to the Dependency Injection pattern.

Last year, we recommended keeping Resolver for projects using UIKit; however, due to its depreciation, we now recommend changing dependency injection libraries. For those wishing to retain a similar syntax and more functionality, Factory is an **excellent alternative**. Because of the similarity of its syntax and operation, migration from Resolver to Factory is **straightforward**. Factory documentation has also been improved, offering clear, detailed instructions for efficient use in SwiftUI and UIKit projects.

OUR PERSPECTIVE

We recommend you give Factory a try in your projects, as it offers significant potential for improving the code quality and performance of iOS applications. Although there are trade-offs, its benefits warrant serious evaluation for future projects.

*https://github.com/hmlongco/Factory

33 Room

TRIAL
12/19 blips

Data storage in mobile applications is essential to enhance the user experience and avoid waiting times. A caching system makes it possible to operate in offline-first mode, displaying cached data first before refreshing it via a network call.

SQLite is the standard solution for relational databases on mobiles, but it's often preferable to use an ORM (Object-Relational Mapping), a library that interfaces a database with the rest of the code, to simplify interactions and typing.

In the context of Android, Google has created and integrated Room into Android Jetpack, a set of libraries designed to simplify the task of developers. With Room, it's possible to **easily** declare the entire database structure and various queries in **a few lines** of Kotlin code, with annotations.

Room is a mature library, in existence since 2018, which benefits from comprehensive documentation and is easy to use. It **integrates easily** with Kotlin and offers reactive operations. For example, when a table is modified (adding, modifying or deleting rows), Room automatically emits a new value, enabling components to update themselves via reactive methods such as flows (coroutines), observables (Rx) or Livedata.

What's new in 2024 is that, since May, Room has been **compatible with Kotlin Multiplatform**. Although this version is still in alpha, Google has announced its commitment to KMP, which points to a stable release before the end of the year.

OUR PERSPECTIVE

There are serious competitors to Room, such as SQL Delight or Realm, which also offer compatibility with KMP. In any case, Room remains relatively simple to use it's a safe bet for Android and soon for KMP too. You can safely integrate it into your Android application and envisage its use in a future Multiplatform version.

TRIAL
13/19 blips

34 Swift Dependencies

In iOS development, decoupling dependencies is a recurring challenge because there is no official solution. As a result, projects often have highly coupled code, making testing and maintenance difficult.

Swift Dependencies* is a dependency injection library designed to **address** these issues in Swift applications. Initially designed for The Composable Architecture (TCA), it can also be used as a **standalone library**.

Like all dependency injection mechanisms, it decouples your code from its dependencies and allows mocks to be injected during testing, making it **easier to test and maintain**. It also provides a method for injecting stubs for previewing.

This library uses **similar** dependency management mechanisms to SwiftUI's `@Environment`, making it **easy to learn**. It also includes a macro to quickly implement most dependencies, speeding up their definition and reducing boilerplate code.

Swift Dependencies is **lightweight** and compatible with incremental adoption, allowing teams to integrate at their **own pace**. However, this lightness comes at the cost of certain features, such as weak injections and injection scopes, which must be reimplemented on a case-by-case basis.

OUR PERSPECTIVE

Swift Dependencies offers a compelling way to manage dependencies, whether in a TCA project or as an incremental migration in a non-injection project, although the lack of important functionality may be a barrier to adoption.

*https://github.com/pointfreeco/swift-dependencies

TRIAL
14/19 blips

35 Typed Errors in Kotlin avec Arrow

In programming, we distinguish between business errors (a password that's too short) and technical errors (a failed network request). Business errors are normal cases of execution and must be handled in the same way as successes, whereas technical errors are implementation details.

Historically, Java has used **mandatory checked** exceptions. However, they are difficult to manage and often pollute the business code, making it implementation dependent. What's more, they are regularly abused to handle business errors, which is **costly** in terms of performance because they generate stacktraces.

Kotlin has responded to this problem with unchecked exceptions, making error handling **optional**. This reduces the robustness of the code but decouples the business code from the technical implementation.

Arrow, a library for Kotlin, provides a **functional-programming-inspired** approach to handling business errors: Typed Errors. **This distinguishes** technical errors, which are handled by unchecked exceptions, from business errors, with two main methods:

- Explicit method with **`Either`**: a typesimilar to Result, allowing you to specify an exact error type without inheriting Throwable.
- Implicit method with **`Raise`**: a DSL similar to Java's checked exceptions that runs in parallel with Kotlin's exception system. Currently implemented with method extensions, Raise may use Kotlin's context parameters in the future, although these are still experimental and planned for Kotlin 2.2.

These methods avoid trycatch and stacktraces, **improving performance and flexibility**.

OUR PERSPECTIVE

Arrow is a powerful tool for making Kotlin code more robust and readable. Both are already must-haves, but Raise, while promising, remains limited by its reliance on experimental features.

ASSESS
15/19 blips

36 Amper

Gradle configuration is a must for any Android or Kotlin multiplatform project, but it can quickly become complex if you think outside the box. This is especially true for multi-module projects, multiple flavors, and third-party plugin integration. This complexity, often a source of frustration, requires a steep learning curve even for experienced developers.

In response to these challenges, JetBrains has developed **Amper**, a new build system designed to simplify configuration. **Intuitive and powerful**, Amper integrates seamlessly with your existing ecosystem. It is available as a standalone version for simple projects and as a Gradle plugin for those who need the Gradle ecosystem.

Amper simplifies the configuration of Kotlin multiplatform projects with one file per module, **reducing Gradle's boilerplate**. As a Gradle plugin, Amper is fully **interoperable** and provides a Gradle fallback for any unsupported functionality.

Amper supports building and running JVM, Android, iOS, desktop and web applications, as well as building **Kotlin multiplatform libraries**. It also lets you mix Kotlin, Java and Swift code, while supporting multi-module projects and the use of Compose Multiplatform.

However, Amper is still in preview and not recommended for production projects: it's still young, with an API that's likely to change, and missing features. Nevertheless, this is **a good time to experiment** and provide feedback to JetBrains, who are known for **listening to their users**.

OUR PERSPECTIVE

Amper is specifically designed to facilitate the development of Kotlin multiplatform projects and could become an interesting solution to explore for developers who want to simplify the management and maintenance of their builds.

ASSESS
16/19 blips

37 Compose Multiplatform

Kotlin Multiplatform (KMP) allows you to share code written in Kotlin between different platforms (Android, iOS, desktop, web, etc.), but not to create user interfaces. So even when you're sharing business logic, you have to use native solutions (like Compose on Android or SwiftUI for iOS).

To remedy this, JetBrains has developed Compose Multiplatform (CMP), which allows you to write **a common UI for all platforms** using Compose's Android-proven API and Flutter's proven **Skia** rendering engine.

CMP is a very promising technology, but it is still in its infancy and stability levels **vary** from platform to platform:

- Android - Stable: natively integrated
- iOS - Beta: although already **stable enough** to use, some limitations exist, and API changes are to be expected.
- Desktop - Stable: **firmly integrated** on Mac, Windows and Linux
- Web - Alpha: the JS/TS transpilation works, but its performance leaves something to be desired. This is why JetBrains has deprecated it in favor of its replacement: the version **compiled to WebAssembly**. However, this version requires very recent browser versions and is not yet compatible with Safari. Both solutions are detrimental to search engine optimization (SEO) and, being still in the alpha phase, are particularly unstable and unsuitable for a project aimed at the general public.

OUR PERSPECTIVE

CMP is **very promising**, and we believe in its potential: for projects targeting Android and desktop, it can be adopted without hesitation. For iOS, it's also highly recommended, despite its beta status. For the web, however, it's still too early to use CMP in largescale production. The rapid progress made by JetBrains, and the community is encouraging, so we'll be keeping a close eye on this technology.

ASSESS
17/19 blips

38 Swift Perception

The **`@Observable`** annotation, introduced with iOS 17, is a big step forward for the responsiveness of iOS applications. However, this feature is only available in iOS 17 and later. This poses a problem for developers who need to maintain compatibility with earlier versions of iOS (iOS 14 through 16), depriving them of this important feature.

The **`swift-perception`**[1] library aims to solve this problem by **backporting** `@Observable` for iOS versions 14 to 16, allowing it to be used without requiring users to update their OS or create more e-waste.

However, there are a few drawbacks to using swift perception. First, while this solution emulates the behavior of **`@Observable`**, it is not as well integrated as the native implementation. The syntax provided is **similar** but not identical to **`@Observable`**: you must wrap your views in a special component, which adds noise to the code. This makes the code less pleasant to read, and increases the workload involved in deleting the library when it becomes obsolete.

What's more, like all libraries that use Swift's advanced macros, swift perception adds **significantly** to compilation time. There are solutions to mitigate this, such as precompiling swift-syntax[2], but the core of the problem remains.

OUR PERSPECTIVE

In conclusion, swift perception is a **valuable** solution for developers who want to use **`@Observable`** on iOS versions prior to 17. Despite its drawbacks, it allows you to **prepare your code for the future** without losing users. At Theodo, we're integrating it into our R&D and recommend that you try it for your iOS 14 to 16 compatible projects.

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

ASSESS
18/19 blips

39 Swift Testing

XCTest has been the epitome of iOS testing since it replaced OCUnit, but it's notoriously verbose and inflexible. In fact, the amount of test code grows exponentially and is not easy to read. This has led developers to look for alternatives such as Quick. But while these tools have a more expressive syntax, they don't address the problem of verbosity, and they require a longer learning curve.

Swift Testing* is a **promising** answer to these challenges. This framework has a syntax that leverages the power of macros and annotations to be more intuitive and less verbose. For example, **test parameterization** allows developers to seamlessly run the same test with different inputs.

In addition, Swift Testing **improves test organization** with a Swift type-based hierarchy and tag-based categorization. It also takes advantage of Swift 6 to improve the safety of concurrent tests, but most importantly, to launch tests in parallel by default, greatly improving their **execution speed**.

Finally, integration with existing XCTest-based tests is straightforward, enabling **easy migration** without overhauling the entire test infrastructure.

OUR PERSPECTIVE

At Theodo, we're keeping a close eye on Swift Testing. Its innovative approach and features make it a serious candidate for future adoption. However, due to its newness, it is currently in the evaluation phase. We believe that Swift Testing has the potential to become a key tool in our testing strategy as it proves its reliability and adds new features (such as UI or performance testing).

* https://developer.apple.com/documentation/testing/

HOLD
19/19 blips

40 Hot Reload
en SwiftUI

Hot Reload is the ability to see changes made to a running application without waiting for it to recompile. This feature speeds development by allowing developers to see the effects of their changes in real time and in a context that is closer to the real world than previews.

This capability is generally considered inaccessible to compiled technologies like SwiftUI. However, there is a library that makes this possible by recompiling and then injecting the modified code using very low-level tools: InjectIII*. This library also supports UIKit, Vapor, and even Bezel.

However, using InjectIII comes with some serious concessions. The initial installation can be long and frustrating. After that, you'll have to be very patient in the face of technical limitations: you may have to change the path to your application or compromise your Mac's security to enable hot reloading. What's more, some SwiftUI features, such as `.onChange`, are **not compatible** with hot reloading.

The main drawback of this library, however, is that it requires you to modify **all components** in your codebase to clear their types to `AnyView`. This significant customization can interfere with SwiftUI's identification mechanisms, affecting your application's animations and performance during development. A special workflow is also required to remove all traces of the library in production, otherwise the security and performance of the application will **be compromised**.

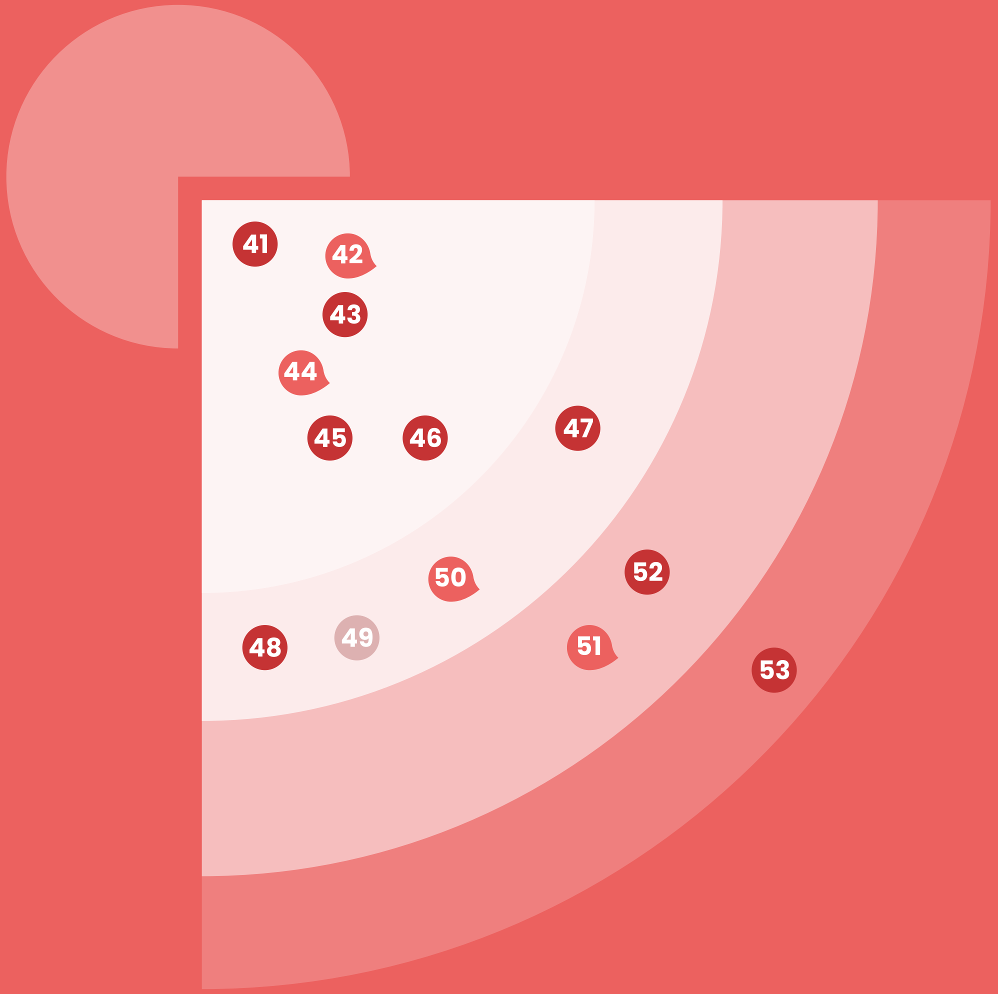
OUR PERSPECTIVE

In spite of the technical feat, we don't recommend using this library due to its important drawbacks. Nevertheless, it shows that hot reloading is possible, and some Apple engineers might be inspired by it.

* https://github.com/johnno1962/InjectionIII

QUADRANT

General



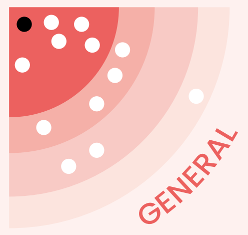
13 BLIPS | 6 ADOPT | 4 TRIAL | 2 ASSESS | 1 HOLD

● New ● Move ● No change

This year, BAM, now under the Theodo Apps brand, celebrates its 10th anniversary, and the landscape of mobile development tools and methodologies has changed considerably. Our "Transverse" section is a testament to this. 10 years ago, mobile developers didn't even have a continuous integration tool worthy of the name. Today, we're talking about security assessment techniques, a new code editor, and even tighter integration between server and front-end code.

In previous issues, we've also used this section to share our lean development practices. This year also marks the release of the book "Lean Tech Manifesto" written by Fabrice Bernhard and Benoît CharlesLavauzelle, the founders of Theodo. We continue to share our Lean Tech best practices with you and invite you to take a closer look at them by reading this book

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



❹ 41 Feature Toggle

ADOPT
1/13 blips

At Theodo, we use a lean approach with continuous deployments and chunked tasks that can be completed in a day, avoiding batch effects. **Trunk-based development**, where developers frequently merge their changes into the main branch, is at the heart of our process, but remains difficult to apply in the mobile space due to store **validation cycles**.

Feature toggles are essential for efficient workflow and finegrained control of applications in production. There are several types of feature toggles, including **authorization toggles and A/B test toggles**, which have a specific business purpose. **Release toggles and operational toggles** facilitate lean development practices.

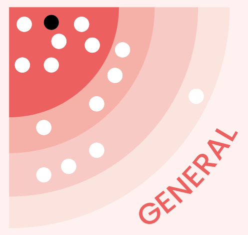
Release toggles, which are built directly into code, **allow features under construction to be** merged without being activated for end users. This facilitates continuous integration and limits the number of branches and environments required for development. **Operational toggles**, which can be configured remotely and retrieved at runtime, allow **a problematic or obsolete feature to be quickly disabled** without requiring a new deployment. This increased flexibility is critical for responding quickly to production incidents and maintaining application stability.

Feature toggles add complexity to code, especially when they are interdependent, multiplying possible test scenarios. To prevent this from becoming unmanageable, it's important to follow certain best practices:

- To avoid the interdependence of toggles, it is advisable **not to associate several toggles with the same functionality**. In addition, it is essential to unit-test the different paths that toggled code can take, to ensure that every possible combination works correctly. During deployment, it's also a good idea not to change the state of too many toggles at the same time, to limit the risk of conflicts and regressions.
- To ensure efficient continuous integration, it's a good idea **to change a release toggle to an operational toggle when a feature is ready to be deployed**. This maintains the toggle's functional continuity, facilitating tests, updates and deployments without major disruption.
- **Documenting toggles** is essential for longterm maintenance. The lifespan of toggles can vary from a few weeks to several years, and without proper documentation it can be difficult for developers to understand their use and condition.
- **Knowing the status of toggles in production** is also critical. To this end, it is important to version release toggles to ensure deterministic application builds. An interactive operational toggle console, such as Firebase Remote Config, can be used to monitor and modify toggle status in real time.
- Finally, **choosing an appropriate update frequency** on the application side for dynamic toggles helps maintain the ability to respond quickly in the event of an incident.

OUR PERSPECTIVE

Despite these complexities, we see feature toggles as an essential practice for maintaining a fast and agile development pace, especially in a mobile development context, while ensuring control of applications in production.



ADOPT
2/13 blips

42 Flashlight

How do you know if your application is performing well? This is a complex question, not least because of the multiplicity of metrics (FPS, TTI, RAM...) and the non-determinism of measurements.

Flashlight, which we are developing, aims to be the answer to this question for Android. Flashlight gives your app a real-time performance score, **without any preconfiguration in the app**. So, unlike most existing tools, **even production apps are supported**, regardless of their technology.

To go one step further, Flashlight can run your e2e tests multiple times, aggregating different metrics and averaging the results into a report with an assigned score. Advantages over other tools: this score provides an easy-to-understand overview of performance, and the comparison view lets you assess the impact of a change in the application. At its core, Flashlight is open source. That means you can take measurements locally with your own device, but there's also a cloud version that runs on a real Android device and can be integrated with a CI to get a score on a regular basis. However, you'll have to provide it with your own E2E tests (only Maestro is supported), which can be tedious.

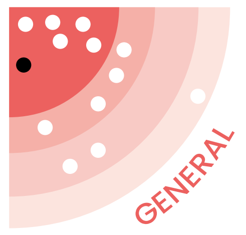
We use Flashlight **as an indicator in performance improvement tasks**, which has allowed us to improve the scrolling fluidity of a React Native app or reduce the startup time of a Flutter app.

Flashlight has also allowed **us to avoid performance regressions in our projects**, for example by detecting abnormal CPU consumption after installing an SDK.

Finally, Flashlight is used in conjunction with Meta to monitor the performance of the React Native framework.

OUR PERSPECTIVE

We now recommend Flashlight as an "Adopt". We invite you to use it to assess the impact of major technology decisions, or as an indicator to help you with a performance improvement.



ADOPT
3/13 blips

43 Generate an API client

Writing a REST API client by hand is a time-consuming task that adds little value to users. This time, which is not spent on more creative or useful pursuits, is often a source of frustration for developers. What's more, the repetitive nature of this task can lead to distractions and errors in translating API specifications into working code.

Tools such as OpenAPI Generator* help solve these problems. This approach automatically generates client code from API specifications without any manual intervention.

This practice has a few limitations:

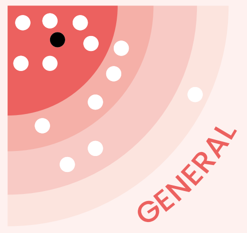
- Generation tools don't always support the entire OpenAPI specification, as in the case of **`oneOf/allOf`** when generating Dart clients.
- The generated code may be incorrect or incomplete, particularly when the generator is confronted with specifications that are partial or out of phase with the actual implementation of the specified API.

However, this latter limitation actually paves the way for better collaboration and understanding between development teams. Knowing that their specifications will be used to automatically generate client code, back-end developers are encouraged to adhere strictly to OpenAPI standards and provide clear, concise specifications. This improves the quality of generated code, documentation, and **API maintainability**, and fosters a culture of collaboration and quality within teams.

OUR PERSPECTIVE

We strongly recommend automatic API client generation for any organization that wants to streamline its development processes and produce superior applications. Keep in mind, however, that this practice requires a joint investment by front-end and back-end teams.

* https://github.com/OpenAPITools/openapi-generator/



ADOPT
4/13 blips

44 Rive

Designers are increasingly incorporating animation into their interfaces to enhance the perceived quality of a product and **evoke emotion in users**.

However, creating animations with code is challenging for mobile developers due to the complexity of synchronizing multiple elements and managing a large amount of code. The variety of platforms and frameworks also makes it difficult to create consistent, reusable animations.

Another approach is to integrate animated assets, but most existing solutions **don't allow direct code interaction** with animations, making it impossible (or very difficult) to respond to user interactions and changes in application state. Rive is a solution that responds to these needs by providing a detailed understanding of the production process for digital interfaces, by proposing a unified production pipeline between designers and developers.

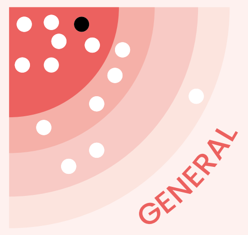
- On the design side, Rive eliminates the need to use multiple tools such as Illustrator or After Effect. The editor allows you to design and animate within the same interface and facilitates the transfer of assets.
- On the technical side, Rive **reduces the number of roundtrips with designers**, eliminates the problem of corrupted files and simplifies the integration of interactive animations with integrated state machines. Rive makes collaboration between designers and developers smoother, reduces errors and speeds up the production process.

Animations created with Rive can **respond to clicks**, **movement**, **or state changes** based on the data they receive, providing a dynamic and immersive user experience. Rive optimizes its files to be up to **10 times lighter than Lottie files**. What's more, it limits the impact on the codebase by integrating the state machine directly into the file.

The Rive tool positions itself as a strong competitor to other animation creation solutions thanks to its better performance and more developed APIs. Because the rendering engine and players are open source, they offer **greater flexibility and adaptability** to meet specific project needs.

OUR PERSPECTIVE

After more than a year of use, we now recommend adopting Rive to **add motion** to your products through highquality interactive animations. The financial and technical constraints mentioned last year now seem to us to be largely offset by the simplification of production workflows.



45 Supabase

ADOPT
5/13 blips

We're always looking for tools that simplify backend development while providing robust functionality. Supabase caught our eye as an open-source backendas-a-service (BaaS) platform that **simplifies the integration of complex authentication systems and the management of backend infrastructure**.

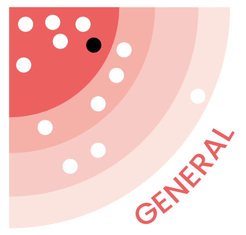
Supabase stands out for its use of PostgreSQL, which provides flexibility for complex queries and data management. Its authentication system is highly configurable, supporting email/passwords, magic links, and third-party providers. **The quality of its SDKs ensures smooth development on multiple platforms**, and its integration with frontend frameworks improves developer accessibility.

Supabase makes it possible to write serverless edge functions and fine-tune security management with Denobased RLS. This improves **performance**, **security and scalability**, making it ideal for the needs of modern applications. Supabase's self-hosting capabilities potentially offer total control over data and infrastructure, **addressing the need for confidentiality and customization**.

Given these strengths, we place Supabase in the Adopt category. Its **robust** functionality, **ease of use**, and **open-source** nature make it an excellent choice for developers. In addition to simplifying back-end development, Supabase offers advanced features that are essential for largescale applications.

OUR PERSPECTIVE

We highly recommend adopting Supabase to improve development efficiency and project scalability.



46 UI Snapshot testing

ADOPT
6/13 blips

The challenge of ensuring the consistency of a mobile application's user interface is a headache for all developers. This difficulty is compounded by the **variety of devices** (phones, tablets, TVs, ...), **orientations, and screen sizes**, making manual regression testing nearly impossible. Snapshotting UI provides a modern solution to these challenges.

UI Snapshotting **captures snapshots of your application's UI** in various states and stores them in your codebase. When a change is made, these snapshots are compared to the current UI to **detect any unintentional changes**.

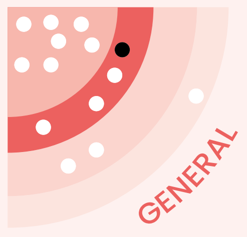
We have used Snapshotting UI on several platforms, each with a different level of maturity:

- **Flutter**: The most advanced, offering integrated snapshot testing with Flutter Goldens. This does not require emulators, making testing extremely fast.
- **Native Android**: We use the Paparazzi library, which captures UI snapshots and tests regressions without using emulators.
- **Native iOS**: Although the swift-snapshot-testing library relies on simulators, it's fast enough to have minimal impact on our CI.
- **React Native**: This platform lacks a mature solution. UI snapshot testing requires additional configuration and generally relies on end-to-end testing on emulators in conjunction with jest-imagesnapshot.

A major challenge we've encountered is the **inconsistency of snapshots between different CPU architectures**, **simulators**, and **operating systems**, which can lead to inconsistencies in the CI environment. To mitigate this, we can reduce the precision of the configuration to get consistent snapshots on the CI.

OUR PERSPECTIVE

Adopting a snapshotting UI improves productivity and ensures a **consistent**, **reliable user experience** across updates. Despite the challenges of inconsistencies between environments and the potential slowdown of tests running on emulators, the benefits far outweigh these drawbacks.



TRIAL
7/13 blips

47 Identifying Defect Detection Stage

In the book Dantotsu Radical Quality, Sadao Nomura suggests classifying defects according to their **stage of detection** rather than their severity or urgency. We expanded on his ideas in The Lean Tech Manifesto, suggesting five stages of defect detection:

A - Detected before release by the developer

B - Detected before reaching an internal customer (code reviews and continuous integration)

C - Detected before release (functional reviews)

D - Detected after release but before receiving a complaint (continuous deployment and monitoring)

E - Detected by a user

This approach contrasts sharply with current industry practice, where defects are ranked by severity and only the most critical are handled and analyzed. Early detection and discussion of bugs has several advantages:

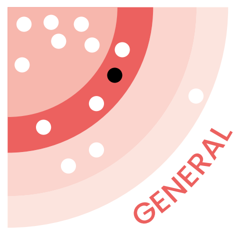
- Bugs caught early are fresh in the minds of those who introduced them, making it easier for them to explain their **misunderstandings and learn from their mistakes**.
- We avoid situations where the person who introduced the bug has already left, preventing any learning
- Even **minor bugs** are often associated with misunderstandings that can lead to more serious problems. Addressing the causes of small bugs **improves the overall stability of the application**.

At Theodo, we have gradually expanded our analysis from D and E bugs to include earlier detection stages. We have also focused on reducing the total number of late discovered bugs (D and E) to detect them earlier. We believe that an increase in early-stage detections is positive if it helps reduce the number of late-stage detections. This has several advantages:

- The team addresses obstacles earlier in the project, **improving overall productivity**.
- We reduce the cost of fixing defects.
- The team generates knowledge faster.
- The quality of the final product is improved.

OUR PERSPECTIVE

We recommend this method but putting it into practice requires a **good generative culture**. Focusing on **learning** and **avoiding blame** is essential, and some teams may not be ready to fully adopt this methodology.



TRIAL
8/13 blips

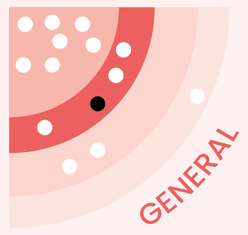
48 Maestro

Performing E2E testing for mobile applications, especially for React Native and Flutter, is tedious. Appium (and Detox) have their strengths, but they are often complex to use, not very intuitive, and can be difficult to automate in CI/CD.

Maestro is a new, highly effective mobile UI testing framework. While it's not as comprehensive as Appium, its minimal setup and quick implementation make it an **ideal choice for those new to E2E testing**. You can run your first test in as little as **15 minutes**. Maestro also includes a graphical interface (Maestro Studio) that lets you **visually select UI elements** and provides suggestions on how to interact with those elements in your tests. What's more, Maestro has its own built-in cloud service called Maestro Cloud. No need to set up simulators, just upload your application and tests and they take care of the rest (reports, parallel execution, intermittent error prevention, screen recordings, logs, etc.).

OUR PERSPECTIVE

We're still in the process of integrating this tool into certain projects. Finding the right configuration for each of these projects can be complex. It remains to be seen if price, ease of use, reliability and return on investment will make it a wise choice, but we have high hopes for Maestro!



49 MASVS 2.1

TRIAL
9/13 blips

In 2024, mobile application security is more of a priority than ever, in the face of constantly evolving threats. With 7 billion subscriptions to mobile networks[1], and a third of businesses[2] experiencing downtime or data loss due to compromises on mobile devices, it's essential for developers to **integrate security right from the design phase of their applications**.

OWASP's Mobile Application Security Verification Standard (MASVS), a musthave standard for mobile application security, has recently evolved into versions 2.0 and 2.1. MASVS 1.5 consisted of 84 checkpoints divided into 7 categories covering critical areas such as data storage, authentication, and network communications.

MASVS version 2.0 has made significant simplifications, eliminated redundancies and adopted standardized terminology such as OSCAL. The result is a **streamlined list of 22 control points** that provides a clearer, more concise approach.

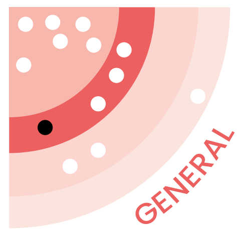
One of the major innovations in MASVS 2.0 is the introduction of MAS profiles. These profiles configure security tests according to the specific requirements of the application, considering its level of sensitivity. The associated tests are described in detail in the **Testing Guide (MASTG)**, which provides practical technical advice for validating the 22 security checkpoints.

Version 2.1 adds the MASVS-PRIVACY category to the standard, which is essential to help developers comply with privacy regulations such as the RGPD.

OUR PERSPECTIVE

At Theodo, we have adopted MASVS as our **main framework for securing our applications**. This standard offers a comprehensive and flexible solution, perfectly adapted to the diverse needs of our projects. Its structured approach not only facilitates the implementation of security controls, but also strengthens our collaboration with security auditors.

[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



50 Ship Show Ask

TRIAL
10/13 blips

During feature development, code review is critical to ensuring quality and adherence to team standards. However, systematic review can slow developers down.

To avoid this, Rouan Wilsenach suggests an approach called "ship, show, ask". For every change, you have three options:

- **Ship**: create an advanced branch without proofreading for minor, standardized changes.
- **Show**: create a branch with a pull-request and merge without waiting for proofreading for changes that are not critical but require proofreading later; also, for changes made in a team (pair or mob programming).
- **Ask**: create a branch with a pull-request and wait for proof-reading before merging for critical or uncertain changes requiring a colleague's opinion. We talked about this practice last year, and it's now part of our standards. We have reduced the time spent on code reviews without reducing the quality of the code.

Like code review, pair/ mob programming, ship/show/ask is a means of communication between a manager and an employee. It's up to the Tech Lead to use this method wisely to ensure code consistency, developer progress, and the team's ability to deliver features quickly. It's the Tech Lead's responsibility to define rules to improve quality.

OUR PERSPECTIVE

"Ship, show, ask" speeds development while maintaining code quality, but an overall code quality strategy remains essential to ensure robust, consistent deliverables.

GENERAL

ASSESS
11/13 blips

51 Weekly Engineering Review

Managing a growing technical team can be challenging. As it grows, with multiple projects, different technologies, and different contexts, it becomes difficult to ensure that everyone is focused on what's most important, and even more difficult to ensure cross-team collaboration and knowledge sharing. Each engineering manager works hard with his or her team, while others face the same problems on the other side of the open space.

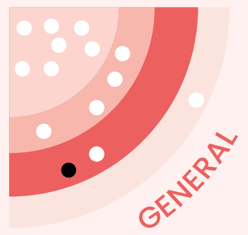
The Weekly Engineering Review is a ritual that brings together all the company's engineering managers and staff engineers. The principle is strongly inspired by Amazon's **Weekly Business Review**. The goal is to ensure strong **alignment of all technical leaders** around common performance metrics and to foster knowledge sharing and collaboration across teams. The meeting is structured around a **set of KPIs** that aggregate the performance of all technical teams in different areas: delivery, quality, team health... The KPIs are contextual, so they can and should evolve over time to reflect current challenges and align with the company's business interests. The **weekly** aspect allows each technical leader to maintain a strong focus on the goals and helps them align the day-to-day work of their teams with them. A **manager is assigned to each KPI** to ensure that data is collected, that the KPI is updated each week before the meeting, and to comment on deviations. Finally, the WER is structured by a **facilitator** who ensures that each KPI is prepared and that the meeting runs smoothly. At Theodo Apps, we implemented the WER 6 months ago. We've been tracking KPIs such as the number of unresolved bugs, the cost of each screen developed in our projects, the number of CFPs sent by our engineers to conferences, the number of inter-team learning exchanges, and the number of engineers who attended bootcamps or training courses.

After a few months of WER operation, we have seen a **mixed impact**. Some KPIs, such as training attendance or the number of defects analyzed by team leaders to better understand the root cause of defects, have increased. Other KPIs, such as the number of CFPs sent, have not changed significantly.

Regardless of its direct impact, the WER is proving to be an effective means of creating a **true team of technical leaders** who previously worked in silos. Feedback from all participants has been very positive about the sense of team spirit created.

OUR PERSPECTIVE

It's too early to measure the impact of the WER on business performance or to see tangible improvements within the teams. We're still learning how to execute this ritual properly, and we see this practice as an **ongoing experiment in engineering management and leadership** until we stabilize the practice.



ASSESS
12/13 blips

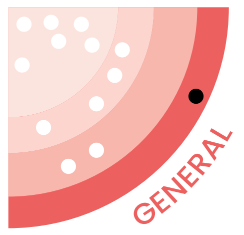
52 Fleet

When developing for Kotlin Multiplatform (KMP), developers must juggle between Android Studio for Kotlin and Xcode for Swift, which disrupts workflow, increases co**ntext switching**, and can lead to **synchronization conflicts** when editing files in both editors. Fleet, an IDE from JetBrains, was created to **facilitate the use of KMP**. It solves these problems by allowing developers to manage code from both languages in a single IDE without the need to use other editors.

As a trusted provider of tools such as IntelliJ and Android Studio, JetBrains ensures greater efficiency for cross-platform developers with Fleet. The IDE offers features like auto-completion and **code navigation that surpass even Xcode**. However, Fleet is still in its early stages and has several limitations, including a lack of features and stability issues. For example, you can't add plugins, which means you **can't use Copilot**, but JetBrains offers an AI-based auto-completion alternative.

OUR PERSPECTIVE

At Theodo, we have adopted MASVS as our **main framework for securing our applications**. This standard offers a comprehensive and flexible solution, perfectly adapted to the diverse needs of our projects. Its structured approach not only facilitates the implementation of security controls, but also strengthens our collaboration with security auditors.



HOLD
13/13 blips

53 PWA-first

Mobile strategy

A few years ago, Progressive Web Apps (PWAs) were all the rage. Some even went so far as to predict the end of mobile apps, and some case studies made a compelling case for PWAs.

But **while universal apps have matured**, PWAs have never really taken off. Their adoption has been hampered by **a lack of support on iOS**, **discoverability issues**, difficulty gaining user trust, and **a lack of functionalities** for developers.

The fact that the PWA Summit has only been held twice, and not again in 2024, is a sign that PWAs are losing momentum. Some players are still using them successfully, and new PWAs are being introduced regularly, but we believe they no longer represent a solid mobile strategy.

OUR PERSPECTIVE

The ideas proposed by PWAs remain relevant overall, but in their current state, universal applications (whether with React Native or Flutter) offer far more benefits while incorporating some of the features of PWAs, such as web manifest and installability. We recommend focusing on universal applications and integrating PWA functionality into them, rather than starting with a PWA.

OUR CURRENT STACK

Other adopted technologies

Since we launched our Tech Radar in 2022, we have continued to explore and share innovative technologies and bold technical choices. Throughout the editions, you feedback and questions have encouraged us to deepen our recommendations for launching a new project. In this vein, we're pleased to provide you with an update on our current technological stack, featuring the most commonly used technologies for launching a project 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

OUR CONTRIBUTORS

The content committee

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

A BIG THANK TO OUR SHADOW CONTRIBUTORS :

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*

WHO WE ARE

About Theodo

Theodo is a leading international technology consultancy that empowers and supports innovative companies in designing, developing and deploying ingenious tech products that transform the lives of their users. To support our clients through every stage of their product lifecycle, we have established practices specialized on key product technologies or specific industry sectors.

- **Theodo Cloud**: Modernizing IT infrastructures
- **Theodo Apps**: Cross-platform applications
- **Theodo Data & IA**: Data platforms and AI solutions
- **Theodo HealthTech**: Health systems and services
- **Theodo FinTech**: Financial services
- **Theodo GovTech**: Public sector

KEY FIGURES

- 15 years of existence
- 700 dedicated Theodoers
- Offices in France, the UK, and Morocco
- Clients in 12+ countries

ABOUT THEODO APPS

Theodo Apps is our practice specialized in mobile and cross-platform applications. We partner with clients to define and execute their mobile and cross-platform strategy, designing and developing award-winning custom applications.

10 **years of experience**

250 **products developed**

120 **experts**

DESIGN & PRODUCTION

Pauline Gaillard & Ariane de Bélizal / *Project Managers*

Stéphanie Landrein / *Art Direction*

theodo.
Apps