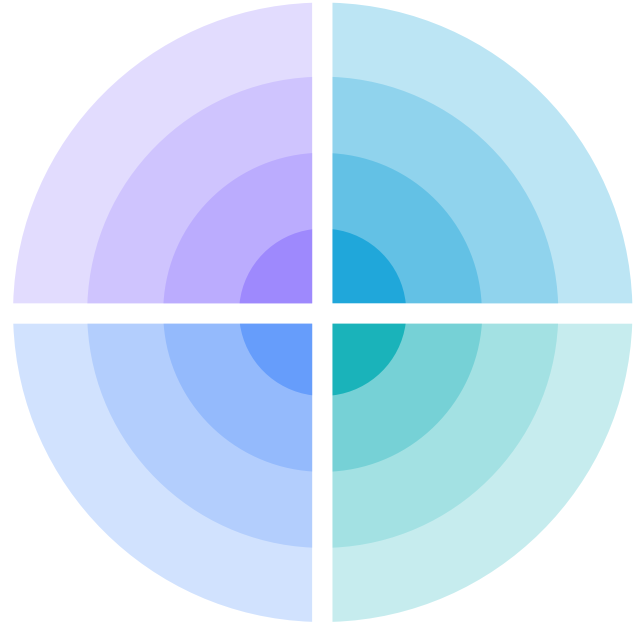
100% MOBILE FOCUS

Summer 2022

Tech Radar

by



Introduction

Co-constructed and co-written over several workshops, our Tech Radar presents an entirely mobile-focused subject matter inspired by the feedback from our 3 tribes BAM and transcribed by 15 contributors.

Our tech team has selected 54 blips that you will discover in the document below. Is considered a « blip » a technology or technique present on our Tech Radar, that plays a key role in the development of mobile apps. Our ambition is to bring a critical outlook in order to guide the transition of technologies, while inviting you to freely discuss with us : contact us, we would be delighted to have your feedback and exchange on our visions and tech expertise!

Summary

01.	Tech Radar Origins	4
02.	Creative Workshop	5
03.	Our Tech Radar	6
04.	Blips	7
05.	Choosing the right stack	8
06.	React Native	9
07.	Flutter	26
08.	Native Technologies	36
09.	Transverse	50
10.	Contributors	67
11.	About us	68

01. Tech Radar Origins

While preparing our Tech Radar, I found a ThoughtWorks radar newsletter in my mailbox, dated 17/12/2014, 2 days before the company's official launch; I was already subscribed! I did, in fact, discover the Tech Radar long before I created BAM. I feel like it's been with me throughout my dev career. **This Tech Radar has always been a source of inspiration and learning. And since I discovered its existence, the ambition to build a team capable of producing such knowledge has never left me!**

This is the fourth time we've done a Tech Radar at BAM. The previous editions were all very different. One of them took the form of a whiteboard with post-its from the entire tech team. Another used the ThoughtWorks open source tool and a spreadsheet file created during a Friday lunch break. Another was drawn up by a Scientific Committee, created ad hoc and composed of the most experienced techs from our team. Neither of these editions was a great success. The energy was not there, and the results were mixed. It produced some interesting discussions within the techs, but after 3 editions, I stopped pushing it.

This year, when I read feedback from ThoughtWorks on their radar (in retrospective, the podcast on how it's going), I understood what was not working. Our radar was still too focused on internal communication; the need was not there. A radar is a significant investment. A few days offsite is necessary, lots of preparation, a good framework, and a project-based organisation. I thought it might be a good time to try again.

BAM has been one of France's largest mobile agencies for some time now. We also chose a wide approach to technology. **Starting from our historical stack in React Native, we began to deliver projects in Flutter and Native, which allows us to cover the 3 most important themes.** Beyond the projects we deliver, many people seek our opinion: on the technologies to choose the work methods and the anticipated difficulties. **I hope our radar's first public edition will allow us to share what we've learned, engage the mobile community, and contribute in an open-source spirit.** This is an open invitation to discover our team's work. I wish you an excellent reading!»"

Marek / **CTO and co-founder**

02. Creative Workshop

When publishing our vision of the mobile ecosystem, we made a few editorial choices that strongly influenced the content:

- **Adopt an expert position :** talk about the technologies we use daily, technologies we've experimented with, or that we've been following for a while.
- **We also didn't want to address the obvious choices:** a large majority of the community has already adopted some approaches and technologies, sometimes even mentioned in the 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.
- **We want to be very clear about our recommendations :** to guide you in the best possible manner, therefore we've opted to take a stand - for example, if a technology is adopted, it's safe to go ahead with your eyes closed; on the other hand, if a technology is on hold, we're not diminishing its value it just means «we're not using it, and we recommend that you not use it for the time being». There are always subtleties to consider, but we want to give explicit opinions. You will not find two competing solutions in adoption if the choice isn't obvious. When we can't decide, we clearly state our opinion through a trial and recommend you consider both!

By the way, you might be surprised that some blips are missing. We've deliberately left some out because we believe they're too obvious and well-integrated at this stage or simply because we don't have enough experience using them!

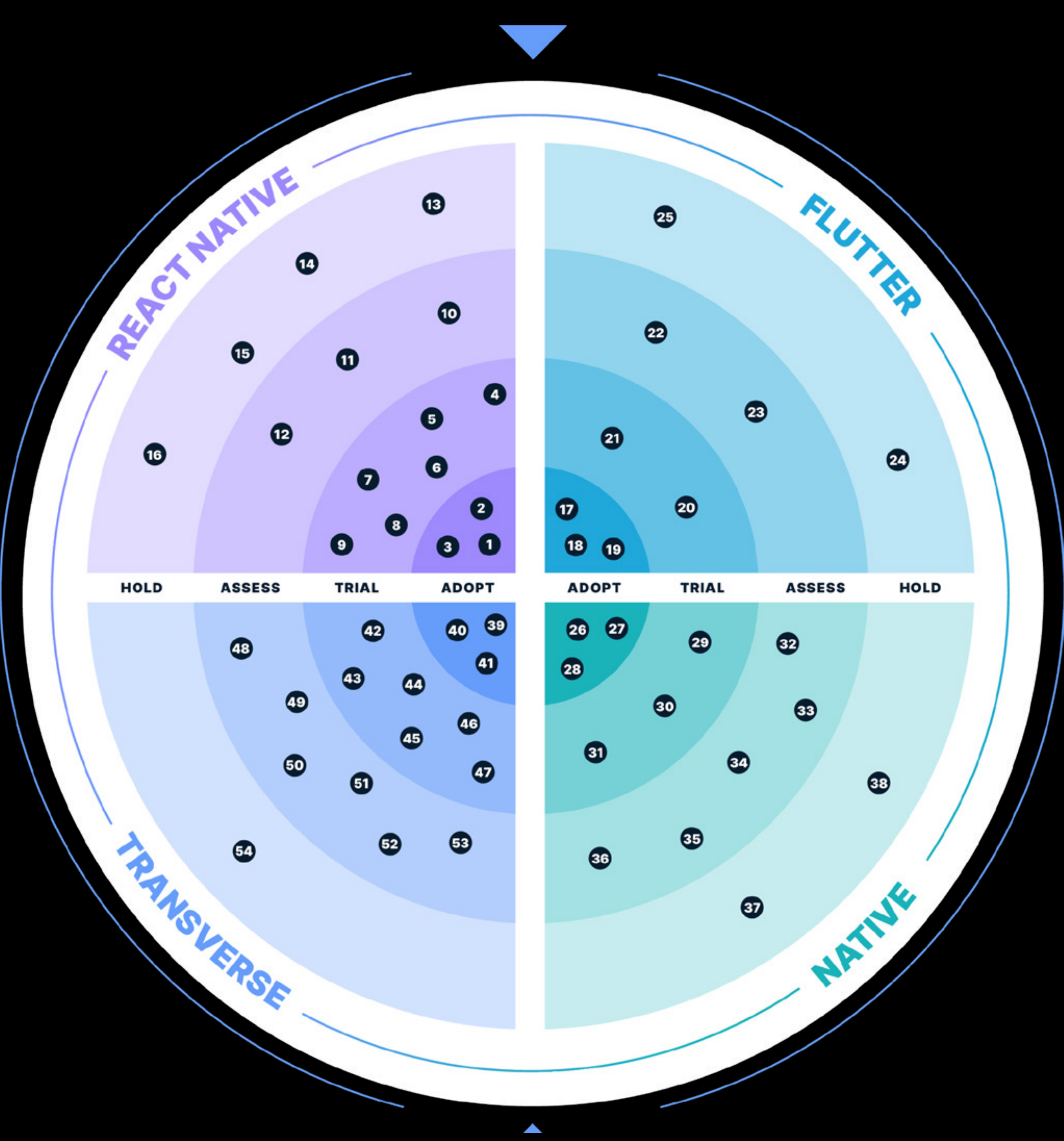
Our Tech Radar has been greatly inspired by the work of ThoughtWorks and the content created around this model (blog posts, podcasts, interviews). We've also adopted a 3-stage approach:

- **A recommandation phase :** issued by the Heads of our various tribes regarding each technology we use: React Native, Flutter and Native.
- The Heads of Tribes consulted their teams to gather recommendations close to field **experience— an exploratory phase** with 2 days of offsite work alongside BAM's technical leaders. Each blip includes clarifying the content, pitching to team members who were not familiar with the technology and, most importantly, deciding whether the blip deserved to be on the radar or not. This was followed by a debate to align with the level of recommendation we would share. Some blips have made the journey from adopt to hold and vice versa! This was repeated over 100 times to process all the recommendations from our Tribes.
- **A final phase is dedicated to content creation :** with blips written by members of the technology tribe, then challenged by at least 4 people.

In numbers, what did this give us? 135 blips discussed 54 retained blips on the radar, 15 people involved in the drafting process, approximately 200 hours of work and another 50 hours for the editing. All of this makes us very excited, but at the same time, we're a little anxious to hand over this radar to you!

03. Our Tech Radar

In this 100% mobile tech radar, we share our expert opinion on techniques, platforms, tools, languages, and frameworks associated with the leading technologies we use daily at BAM: React Native, Native and Flutter.



4 LEVELS OF ADOPTION

ADOPT
we recommend using these technologies

TRIAL
these technologies are ready to use, but we believe they could be optimised

ASSESS
we recommend you read up on these technologies if they meet a specific need

HOLD
we're not convinced by these technologies and don't recommend their use at this stage

04. Blips

REACT NATIVE

ADOPT

1. Flipper
2. Hermes Engine
3. React Query

TRIAL

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

ASSESS

10. Jotai
11. React Native Skia
12. WatermelonDB

HOLD

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

FLUTTER

ADOPT

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

TRIAL

20. BLoC
21. Riverprod

ASSESS

22. Alchemist
23. Melos

HOLD

24. GraphQL with Flutter
25. Provider

NATIVE TECHNOLOGIES

ADOPT

26. Jetpack compose
27. Koin
28. Swift UI

TRIAL

29. Resolver
30. Tuist
31. VIPER

ASSESS

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

HOLD

37. KMM
38. Litho

TRANSVERSE

ADOPT

39. ADR
40. BPMN
41. QRQC

TRIAL

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

ASSESS

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

HOLD

54. Firebase Authentication

05. Choosing the right stack

You will probably notice that our radar doesn't make a clear recommendation between Flutter vs RN vs Native. All 3 technologies define the scope but never are they compared.

Which is the best choice ?

When we launched BAM in 2014, the technology we chose was risky. We believed that cross-platform was the future of mobile and chose our technology stack, Cordova + Ionic. But the choice was far from obvious, Cordova's competitors being Xamarin (backed by Microsoft), Titanium (an approach that uses native UI components) and many others. Does anyone still remember Famo.us? Each of these solutions had some distinct strengths but many drawbacks.

The landscape changed with React Native in 2015, which addressed most of the drawbacks of other frameworks. That's why we adopted it in October 2015. Flutter, released 2 years later, took a technically different approach but with the same level of quality. After some time, the solution proved its value. On the other hand, the introduction and rise in popularity of Swift and Kotlin brought a lot of freshness and innovation to native development, which took away some of the arguments for cross-platform development. We've gone from a phase where the choice was not obvious (before the summer of 2015), through a stage where the choice was relatively straightforward (2015 - 2018), to a stage where the choice is once again very complex.

For each of our projects, we select a specific technology. A discussion that must take into account :

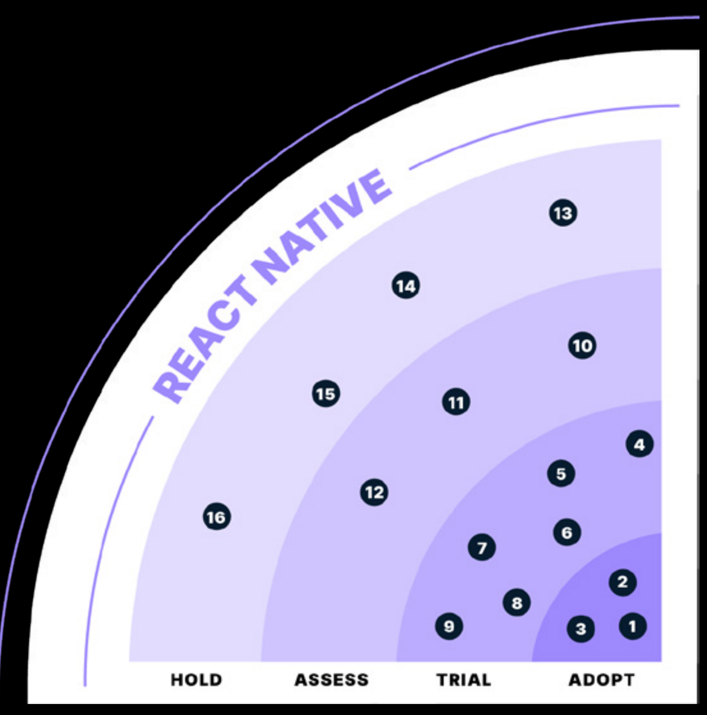
- **Product Strategy :** what features, what design, who will be the users?
- **The Tech vision :** what is the company's technical vision, who will work on this project, what are the existing teams, and what is the recruitment strategy?
- **The budget :** how much can we invest? What constraints do investors bring in?

This deliberation allows us to evaluate the project alongside these 3 technologies and to make a more or less solid recommendation, depending on the limitations.

To sum up, we believe that it's not possible to tell you with certainty «technology A is better than technology B» in the current landscape. The decision must be made on a case-by-case basis with input from all stakeholders and mobile experts with a clear understanding of the different solutions.

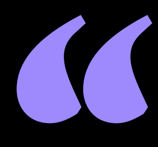
If you want our personalised recommendation for your App and your business, let's have a coffee together!

06. React Native



16 BLIPS

3 ADOPT
6 TRIAL
3 ASSESS
4 HOLD



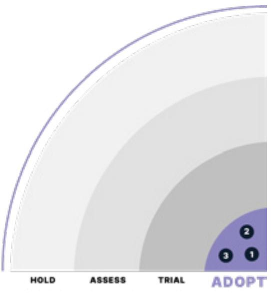
To remember

On the React Native side, we've chosen to present the innovations that make us say that **React Native continues to be a solution of choice for developing cross-platform Apps.** They cover many areas like:

- "Opt-in" news from React Native, which create more powerful apps
- Libraries and architectures for managing the state
- UI and graphics libraries
- Deployment or debugging tools

They are now part of our reference technical stack."

/ Staff Engineer



1 Flipper

In a React Native application, the JavaScript code runs on phone's JS engine. The usual debugging approach was to activate React Native Debug Mode. This would run the code on an external JS engine like React Native Debugger or Chrome's JavaScript inspector. By running the code externally, the developer can take control over the code execution. execution.

This has two major drawbacks: the same problems are not always reproduced with or without debug mode, and execution is significantly slowed.

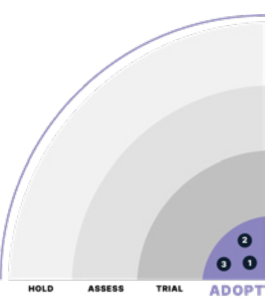
Flipper is a mobile App inspection and debugging tool. It can be used for React Native or native Apps. At BAM, we use it mostly on our React Native Apps.

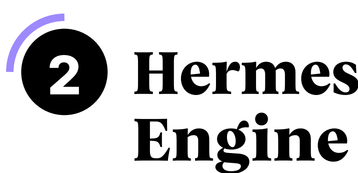
With Flipper and Hermes (a JavaScript engine optimised for React Native), **debugging is done directly on the phone. This makes the operation faster and more reliable.** As Flipper uses Hermes for its debugger, the activation of Hermes is mandatory for the App to be debugged.

Beyond JavaScript debugging, Flipper has other advantages. **The tool offers various network, layout, and memory inspection plugins.** It can connect to libraries such as react-navigation or react Query, which has a Flipper plugin.

Furthermore, as **Flipper is very extensible, it's easy to add homemade plugins.** This makes it possible to set specific measures in the developed App or special interactions that can help developers. For example, one can imagine a plugin to activate/deactivate something in the App. **At BAM, we've developed a plugin to inspect the performance of our Apps**.

We now recommend Flipper as the default debugger on our React Native projects. While it used to be complicated to install, due to native dependencies, this is no longer the case. Flipper meets our needs in terms of tools much better than React Native debugger.





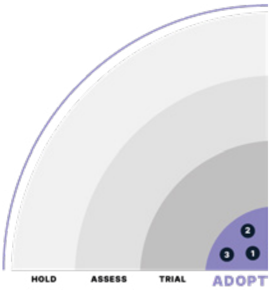
2 Hermes Engine

App start-up time has long been a problem with React Native, especially on low-end Android phones. For example, one of our apps, which contained many JS code, started in 12.9s on a low-end phone.

Facebook has integrated Hermes, a new JavaScript engine, to solve this problem. Instead of parsing and compiling the JS into binary code at app launch (like a traditional JS engine), Hermes does it during the App's build time.

Thus, an app underHermes no longer has an embedded JS file but a binary code file. The results are striking when using Hermes, the App now starts in 3.9s, and the start-up time of all our apps has been cut by at least 2 by simply setting the enableHermes boolean to true in the settings. A few polyfills (eg. i18n, regexp) will allow you to use this implementation even in projects that are not compatible with current JS coverage.

At BAM, we enable Hermes on all our projects and recommend you do the same. The advantages outweigh the costs; without Hermes, the start-up time of your Android apps will probably not be up to Google's recommendations.



3 React Query

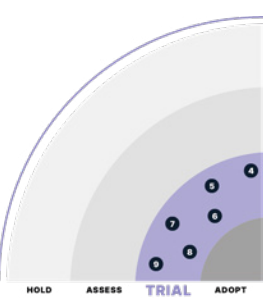
While many developers underestimate the complexity associated with asynchronous calls, they are a very large and complex problem (handling load state, error state, cache, refetch). A simple, high-performance integration can be very tricky.

React Query is presented as a library of hooks for loading, caching, and modifying asynchronous data in React. It offers many features: a simple query with error and load status, dependent or parallel queries, and paged queries. It also includes simple and efficient cache management and ensures compatibility with server-side rendering, notably NextJS.

However, the choice was made to not normalise the cache. This action must therefore be carried out manually if necessary.

At BAM, we started using React Query almost two years ago. This library is very easy to use. However, we recommend that you pay attention to the default setting, as the length of the cache and the number of query attempts in case of error can vary from project to project

We've found that this solution has allowed us to integrate our asynchronous calls much faster and with better quality, allowing us to focus our efforts on business features. We now use it on all our projects that don't have the graphql API, and we recommend you do the same.



4 Expo

The mobile world isn't always easy to get to grips with for developers who have no experience in the field. The need to understand the specifics of each OS, their APIs and how to sign and deploy Apps can be a real entry barrier.

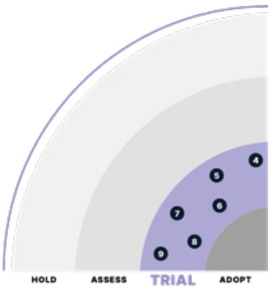
Expo is a platform that allows you to develop React Native Apps by considerably reducing this barrier. You can create a managed workflow App that removes the entire native part from which it's possible to eject (just like the create-react-app). This platform has many features which simplify developers' daily tasks, such as :

- Automatic App signing management: knowledge of the iOS and Android signing system isn't required
- An «Over the Air» update system
- An integrated Continuous Deployment Service (Expo EAS)
- A multitude of packages allowing the use of native APIs
- React Native Web integration: Apps developed with Expo can be deployed on the web

We've substantial experience working with this platform, which we've used on several production projects.

One cautionary note, however: when using Expo, you become very dependent on their service. There have been times when we could not deploy our App for several hours because the platform was unavailable. Moreover, it's now possible to modify the native code even in «managed workflow» using easy-to-main patches.

Therefore, it's not the best solution for projects requiring many modifications to the native code. We recommend Expo for teams working on small to medium-sized projects with little or no knowledge of mobile technology.



5 Create React Native modules using JSI

React Native is a framework based on React to integrate native iOS and Android Apps from a single code base. To do this, two technical elements are necessary:

- on the JavaScript technical side, where the App's graphical interface is defined using generic components
- the native technical side, where these same generic components are combined into native components specific to each platform

In computer science, there are two main strategies for making two technical areas communicate:

- message serialization (e.g., HTTP protocol with messages in XML or JSON format)
- the Foreign Function Interface (FFI) allows to call integrated methods from one language into another language (e.g., the DOM object and its APIs like document.getElementById() are available in JavaScript but are integrated into C++ at the browser level)

For years, React Native has used serialised JSON messages sent across a bridge to communicate between the JS and native technical panels.

This system has limits because when the flow of information between the two technical areas is too great (animations, scrolling events on long lists, etc.), the bridge becomes congested. **This causes problems with the App's responsiveness**.

To overcome these problems, the new React Native architecture has a brick called JSI, a Foreign Function Interface that links the JS runtime to the native domain. This standardised interface is already integrated into JavaScript engines such as JSC and Hermes. Its use significantly improves performance. Some libraries have already taken advantage of JSI :

- react-native-mmkv, which offers a synchronous API, is 20 times more powerful than react-native-async-storage.

- react-native-big number, which makes available to the JS world the C++ integration of bn, a library for arithmetic on large integers, proposed by OpenSSL. This integration has been measured as 300 times more efficient than its equivalent integration in JS.

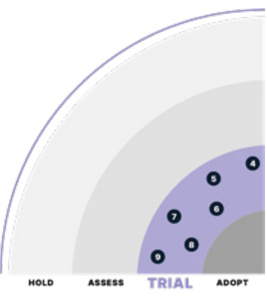
.

However, JSI is also a source of complexity, as the binding between your module's native and JS technical elements will have to be integrated into C++. In addition, documentation is still sparse, and the development of asynchronous APIs is complex, which may slow down its large-scale deployment.

If the module you wish to integrate has simple interfaces (handling serial types), tools such as Codegen allow you to generate the C++ code linked to JSI from your module's interface, written in Typescript or Flow.

Otherwise, you'll have to create your link files in C++, use the JSI APIs and create your own hostObjects to integrate your native modules.

Today we're experimenting with migrating our most popular module, react-native-image-resizer, to JSI. We recommend that you do your research for your native modules.



6 React Native Web

As an App is often available on both web and mobile, it's natural to look for ways to share code between these platforms. It's possible to share business logic, UI state and calls APIs between a React web App and a React Native mobile App. Still, it's impossible to share UI components by default.

React Native Web is an integration of React Native components and APIs, compatible with React DOM.

Installation is straightforward: install react-native-web as a web app's dependency and add an alias from react-native to react-native-web in the bundler setting. You can import any component coded in React Native into its React web App.

We've used React Native Web on several projects, which has allowed us to share between 75% and 95% of our code between web and mobile. Our feedback is very positive. However, it's essential to keep two things in mind:

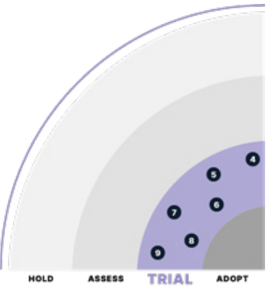
- The code produced by React Native Web is more complex than pure React code. This can impact performance and ease of debugging. However, we've found that the performance impact is generally imperceptible to the user.
- The layout is often quite different between web and mobile, and it's often more relevant to share specific page components rather than the whole page. Some behaviours are particular to the web and are not relevant to share.

It's interesting to use this tool if you define a clear strategy, which makes it possible to dissociate the components that use it from those that need to be coded manually for each platform.

React Native Web is used by many companies, such as Twitter (of which the library's creator was a member) and Uber. The library is also supported by Expo, which includes it by default in its Apps.

On the other hand, this solution was essentially developed by a single developer, which presents a maintenance risk.

Although it's reasonable to think that companies using it could take over maintenance, the risk regarding maintenance means we've placed React Native Web in the trial section.



7 Reanimated v2

Creating animations in React Native can sometimes be challenging. Although the Animated API with React Native is simple, the code is mandatory. It tends to make the components more complex. The integration cost is high; therefore, animations are not prioritised in our App development.

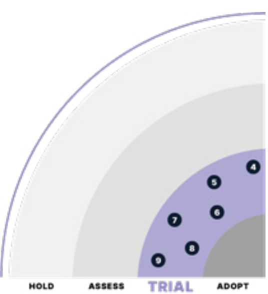
The Reanimated library changes the game from version 2 onwards. Thanks to its more declarative approach, animations become very simple to integrate.

The animation code doesn't interfere with the readability of the rest of the components. Reanimated's hooks approach makes it easy to extract the problematic block in an isolated hook in the case of more complex animations. The hooks also allow animations to be extracted into reusable blocks.

In addition, Reanimated's Layout Transitions make the animation of appearing, disappearing, or changing an item in a list simple. These would be very complex tasks without a library. Reanimated will run the JavaScript code for the animations in the UI thread. This allows smooth animations with no performance issues.

Overall, reanimated improves the quality of animation code compared to React Native's Animated. In addition, this library enhances product quality by encouraging animation integrations.

Despite the advantages of animations for the product, we're only positioning Reanimated as a «trial» because we still see some issues. For example, on some projects, our teams have observed crashes or animations that don't trigger on release builds. However, the library is evolving fast: new features are arriving quickly, and bugs are being fixed frequently.



8 API validation with Runtypes

Over the past five years, Typescript has changed the JavaScript landscape, reaching the top 5 most used languages in 2022 in the stack-overflow survey.

It promises to verify that the output will be sound if the data input is good. This leads to several questions:

Who checks that the input data is correct, and how can we integrate this check during the execution of our App, whereas the typing check is done during the compilation phase?

The Runtypes library allows types and associated validators to be generated from a data schema. These validators make it possible to check that the data received conforms with the type expected by an App and to protect it from poorly formatted data.

We recommend that you set up a validation system to ensure the safety of the end-to-end typing. The Runtypes library is very powerful, sometimes too much so (Contracts, Branded Types, Constraints), which can be double-edged: while senior developers can rightly use it, it makes onboarding junior developers more complex.

We advise introducing Runtypes gradually, limiting its use to the strictly necessary.

9 Storybook

Current web and mobile technologies (Flutter, Jetpack Compose, SwiftUI, React, Vue) encourage us to practise atomic design. This design method consists of building our UI from components rather than pages.

Storybook is an isolated GUI development tool available on the web and React Native, which addresses the dynamics of atomic design. It provides developers with an environment (stories) in which components can be developed independently of the App.

Storybook makes it possible to improve the simultaneous development and to strengthen components with a «component first» focus.

This tool also allows to document and structure the design system's components. Unfortunately, the storybook tooling in React Native isn't as advanced as on the web (Storybook server, chromatic).

Addon React Native Web allows you to configure Storybook to display React Native projects using React Native Web.

This is a very interesting solution if your React Native project is intended to be web compatible. Otherwise, it represents significant development and maintenance costs.

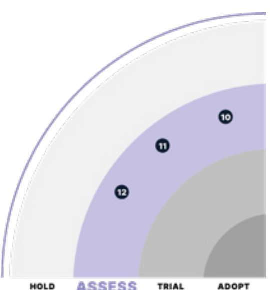
Unlike the web environment, which allows you to publish a static site, React Native requires you to build and deploy a dedicated App on a distribution platform (e.g. AppCenter) to make your Storybook accessible to all project stakeholders.

Finally, some Storybook/React Native versions have compatibility issues with Storybook, making it difficult to update the dependency. For example, at the time of writing, Storybook/React Native isn't compatible with Storybook 6.4.

At BAM, feedback on this library varies from project to project. For some projects, our teams have encountered deployment issues related to Storybook, which has forced them to design an entire component library management system.

Conversely, other projects using React Native and React Native Web could take advantage of each platform's tools to improve the development and validation of their design system.

Because of this we put Storybook in the trial quadrant. One thing is sure: the ecosystem around Storybook is evolving rapidly. Therefore, the gap between React Native and web tooling should narrow.



10 Jotai

A global state is vital to avoid UX inconsistencies, such as a counter of unread messages that aren't synchronous with the messages. But react doesn't offer global state management: it's possible to propagate a local state through the component tree via «props", but this is cumbersome and not very maintainable.

Redux offers a solution to this problem. Through a one-way flow approach, you can design a global state. We adopted it in 2016, to the point of using it by default on all our projects. However, we've noticed two problems:

- On the one hand, a very large boilerplate, even with «redux-toolkit».
- On the other hand, a «central» approach to data implies a much better upstream design.

Meanwhile, the React ecosystem has evolved. For example, hooks simplify the syntax for managing the local state. And React-Query or Apollo allows API call management and makes remote data management easier.

In this context, Redux is less useful and complicated; more modern alternatives have emerged. Rather than a large monolithic state, we recommend an atomic approach such as React's useState hook: in a shared file, we define an atom (e.g., const userAtom = atom(null)), and then we can replace local useStates with useAtom(userAtom).

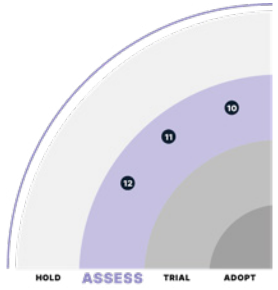
This alternative is more straightforward, closer to the state react hooks API, and requires less upstream state design.

On our projects, we've used Recoil and Jotai, which share a similar approach.

We prefer Jotai for several reasons:

- stability of its API and documentation
- features that simplify the developer experience
- performance that optimises returns
- ease of use with Typescript
- compatibility with Suspense and React's synchronised mode
- integration with other projects such as XState, react-query and Immer.

We believe that Jotai's approach is complementary to using React-Query or Apollo to manage the remote state and that 95% of projects no longer need Redux.
For your next project, we recommend you try combining «Jotai» and «React-Query»/»Apollo».



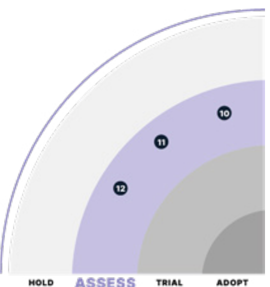
11 React Native Skia

React Native Skia is a high-performance 2D graphics rendering library. It's based on Skia, the graphics rendering engine used by Flutter and Jetpack Compose.

This library allows you to display vector images, graphs, filters, blurring and shaders on your Apps. It's exciting to see what the React Native community is doing with this library, don't you think?

The only downside is that React Native Skia is in the «alpha» version and the current documentation mentions that you must tread lightly when using it. Thus, we don't recommend using this library for a project in production at this stage.

However, this library is a promising and efficient solution which could, in the long term, replace other libraries such as react-native-SVG. Of course, we've identified performance problems with this library.



12 WatermelonDB

As detailed in « [Using relational db as state management]() », using a relational database in a mobile App has many advantages: denormalisation, consistency, schema constraints, transactions, indexes to optimise access, and text search.

What are the most appropriate solutions in React Native?

Watermelon is a native module for React Native that allows you to read, save and search data reactively from the SQLite library, the most widely used database in the world. Commonly used in the embedded and desktop ecosystems, it's also increasingly used in mobile with Room on Android and Core Data on iOS.

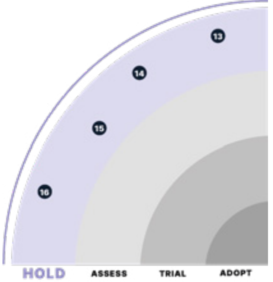
WatermelonDB works with the reactive paradigm: in other words, when a row changes in the database, the components that depend on it are automatically re-rendered.

Finally, the Watermelon database means you can synchronise with a remote server: to integrate a better offline experience, just provide two server-side routes to pull and push changes locally.

However, this module has two limitations:

- WatermelonDB is currently in version 0.24. The library is stable and complete, but the API may change by version 1.0.
- Hook support is paused. You need the React Suspense component and concurrent mode to properly make a hook that can start and stop a React rendering based on an external (database) state change. And the latter is just now coming to completion. In the meantime, using Watermelon may leave you with a slight taste of 2017.

We suggest following this library closely and incorporate it in your toolset once it has been marked as stable.



13 Enable Fabric

Announced back in 2018, the new React Native architecture is finally available and can be activated. It makes the previously impossible possible; it:

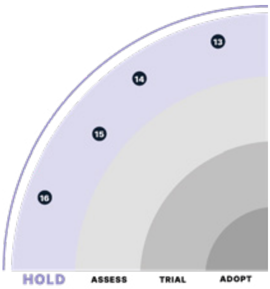
- Supports new React competitive features (Suspense for data loading, startTransition...) from version 0.69. These features improve the user experience (e.g., startTransition keeps the App responsive even during an expensive UI transition)
- Measures and synchronously renders native views, thus avoiding a «jumping» layout effect
- Shares specific optimisations that were previously platform-specific, thanks to the sharing of C++ code (e.g., «View Flattening», which was previously only available on Android)

To migrate to this new architecture, you must change a setting in your App. Please note that all libraries in your projects must support this new architecture.

Reanimated or react-navigation have already migrated or started the migration, but this has not yet spread to most libraries.

In addition, enabling the new architecture significantly increases native build times (up to x5 in version 0.69), which can slow down the team. Our tests also show reduced performance on some components, such as FlatList, which in our experience, consumes 20% more CPU with Fabric. FlashList, a new plugin for high-performance lists made by Spotify, may solve this problem once the new architecture has been integrated.

Our recommendation: this new architecture isn't ready to be activated. However, Meta is working hard to improve the documentation and make the React Native libraries and projects migration process more accessible. In our next Tech Radar, we hope to move Fabric to «Adopt».



14 Build React Native forms with no libs

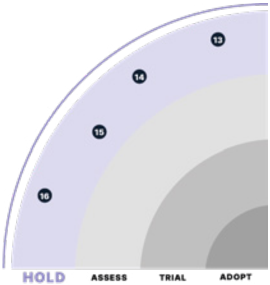
Forms are part of almost all App functionalities. Designed and thought in a very simple way, it can be tempting to want to develop them without using a library.

However, a form is never simple. There are a lot of states and edge cases to consider for each field, such as their value, filling state, validation, error state, or formatting.

We regularly see projects where forms have been deliberately developed without libraries. These projects are prone to quality defects and forms whose complexity will explode over time. However, several React libraries now make it much easier to integrate. For example, we use Formik and React Hook Form libraries, with which we've had an overall satisfying experience.

These libraries have several advantages: they are easy to learn thanks to good documentation, and their compatibility with libraries such as Yup makes validating forms easier.

For these reasons, we strongly advise against developing forms without libraries.



15 Redux Saga by Default

Not so easy to integrate API calls: for example, how to handle error or load cases and add a timeout or a retry?

We've used the redux-saga library for several years to address these issues. Since 2017, it has set itself apart for its simplicity of writing, its use cases when the async/await syntax was not yet widespread, and its testability against redux-thunk.

However, this solution is still challenging to master for several reasons:

- it requires familiarity with the syntax of ES6 generators, which isn't widely used
- the creation of sagas requires a substantial boilerplate
- its API requires a significant amount of learning (understanding the difference between spawn and fork, for example, isn't instantaneous)

In addition, react has evolved since 2017. The appearance of hooks in 2019 and the arrival of Suspense, new solutions such as react-query are starting to emerge. They are easier to use and meet most of our projects' asynchronous needs.

Simultaneously, redux saga remains a very powerful tool to manage complex asynchronous logic, with all the basics to make it a resilient system: notably with spawning and the automatic restart of sagas in case of error, like techniques used in telecommunications (e.g., Erlang, Akka stream). However, they require specialised technical knowledge, and such cases are still relatively rare in practice.

We suggest you look for a simpler alternative to this library. We don't recommend installing this library by default.

16 Styled Components

In React Native, components are styled by default with StyleSheets. However, using StyleSheets isn't suitable when using a dynamic theme or conditional styles.

Styled-components is a CSS-in-JS solution well known in the web ecosystem, which includes a React Native implementation. This means you get many features such as theming, styles as objects or style setting via props.

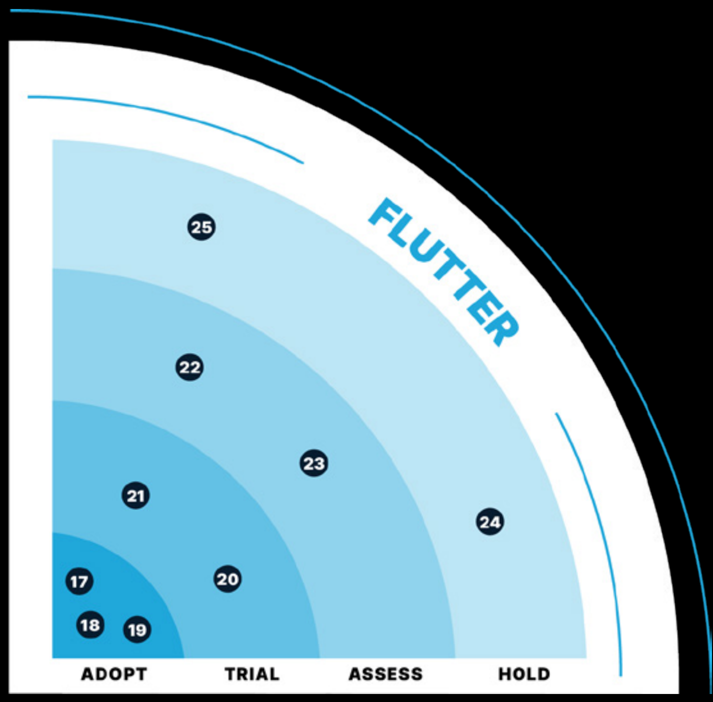
Styled components encourage us to separate our UI components from our smart components, which promotes responsibility separation.

However, the library has several drawbacks related to typing. We've observed prohibitive performance problems with auto-completion in the IDE.

This problem has been identified and corrected several times in the past. Unfortunately, it's still present, which isn't reassuring in the short term.

We recommend that you use the CSS-in-JS Emotion library, which has an API very similar to Styled components, significantly simplifying migration. Emotion is a lighter and faster solution than Styled components, and we've not seen any typing problems at this stage.

07. Flutter



9 BLIPS

3 ADOPT
2 TRIAL
2 ASSESS
2 HOLD

To remember

On the Flutter side, we've chosen to present blips that cover the main issues our teams have encountered over the last year regarding the Flutter ecosystem's main topics: state management, API calls, tests and native bridges. We've experimented with these technologies on projects, often up to production, and on which we believe it's important for us to give feedback.»

Guillaume / **Head of Flutter**

17 Mocktail

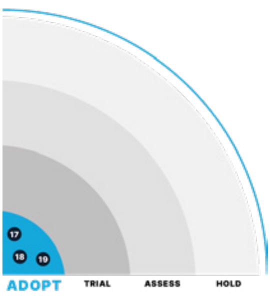
In a unit testing approach, one of the main issues is to mock the dependencies of the features/methods under test, i.e., to pass as object settings that mimic the behaviour of the expected variables without integrating them. In the Flutter and Dart ecosystem, the most popular library is mockito.

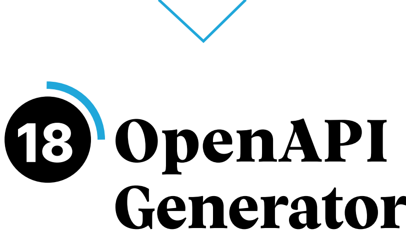
It offers a very accessible API for mocking classes and overloading their behaviour according to the needs of our tests.

One of this approach's limitations is that it relies on code generation via the Dart build_runner. It requires us to rerun the code generation every time we change our classes to update all the mocks. Although it's widespread in the Dart ecosystem, the principle of code generation is an approach that remains somewhat controversial in the community. It adds a step of friction to the development tasks and introduces unchecked code in the code base.

To overcome this problem, Felix Angelov provides an alternative solution: Mocktail. **Mocktail is based on an API very similar to Mockito.** However, the integration is different because it doesn't use code generation. This allows us to mock all our test dependencies that use null safety. This makes the development experience much smoother.

This approach quickly won over the Flutter experts at BAM and became the standard mock on all our projects.





18 OpenAPI Generator

Any client App interfaces with one or more servers, which can retrieve the data it displays and send the information the user puts into it. With Flutter, the definition of API clients that allow these interactions can be tedious and require a large amount of boilerplate code. This occurs when creating a JSON/dart exchanged data (de) serialisation.

The OpenAPI Generator solution allows you to automatically generate the dart code responsible for interfacing with an API that complies with the standard Open API 3.

This involves generating an API contract .yaml file from the back-end code used to generate the client. It's also possible to create this file manually if the API complies with the standard.

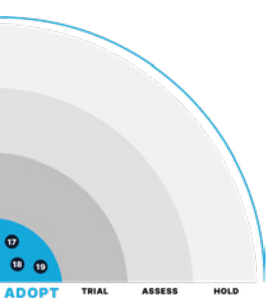
This approach has many advantages:

- guarantees the API contract quality by imposing compliance with the OpenAPI 3 standard (e.g., send the null value rather than an empty String (''))
- simplify JSON/dart (de)serialization
- isolate the definition of entities, makes mocks and the App's testability easier

After using this approach on several projects in production, we've identified **2 significant obstacles** to this integration::

- no tool for automatic generation of .yaml API contract (e.g., in Go)
- partial compliance or non-compliance with the Open API standard on some projects

If these two obstacles are well addressed, the complexity of writing a complete dart API client can be reduced to a single .yaml file.



19 Pigeon

When developing a Flutter project, whether an App or a package, we may have to interface with native mobile code: iOS or Android. For developers to meet this need, Flutter offers a PlatformChannel system: communication channels between the platform's native code and Flutter's dart code.

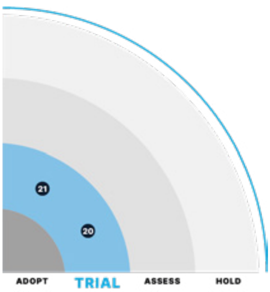
This well-documented Flutter approach has 2 limitations:

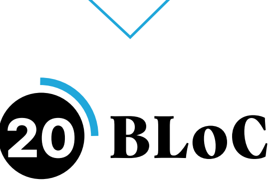
- The amount of boilerplate code to be written is significant
- Typing between data passed on one side of the PlatformChannel and data retrieved from the other isn't possible. Once data is sent to the PlatformChannel, its type must be checked at runtime on the other side each time.

To overcome these limitations, Flutter offers the Pigeon package. This code generation solution automatically creates all PlatformChannel building blocks while ensuring the transferred data typing. This approach saves considerable time in generating the code required for the PlatformChannel to work. At the same time, it guarantees the data exchanged on both sides, which prevents multiple bugs.

However, some limitations exist, such as the lack of support for event channels or enum management. The library isn't yet compatible with desktop platforms.

Despite these limitations, we strongly recommend the use of Pigeon. Our recommendation is based on the official Flutter package membership and its dynamic features, proven in many developments over the last few months.





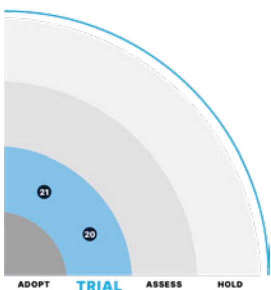
Like any reactive framework, Flutter relies on the existence of a state; when updated, it triggers the UI modification. Since its creation, State Management solutions for Flutter have multiplied.

Felix Angelov developed the BLoC solution for State Managing Flutter Apps. Created at the end of 2018, it now appears, like many others, in the list of recommended State Management solutions for Flutter. Still, it has continuously established itself in the community with more than 400 releases. An 8th version was released this year with sponsors still very involved in the Flutter community (e.g., Very Good Ventures). At BAM, we've had the opportunity to test BLoC on several projects currently in production.

BLoC presents a structured and predictable approach with objects and clearly defined responsibilities (e.g., Event, State and Block). It perfectly synchronises with the Freezed package to limit manual writing of boilerplate code. Furthermore, it offers good code editor integration with snippets to instantiate widgets in the blink of an eye.

Two of its significant assets are its complete documentation with many examples and its active community. Finally, the ease with which test and mock-up code is written using BloC has enabled our teams to achieve and maintain 100% test coverage on our latest projects.

Although we're convinced by this solution and recommend it for managing the state of a Flutter App, we pace it in Trial status because other solutions such as Riverpod (also present in this Tech Radar) are also recommendable. To date, we've not identified a clear winner in the field of state management for Flutter.



Like BLoC and Provider, Riverprod is a state management solution dedicated to Flutter. It appears in 2nd position in Flutter's «List of state management» documentation, just behind its predecessor Provider and before Flutter's «Native» solutions (e.g., setState and InheritedWidget). This excellent positioning is the first indicator of the solution's quality and stability.

This excellent positioning is the first indicator of the solution's quality and stability. Riverpod was created to address 2 provider limitations:

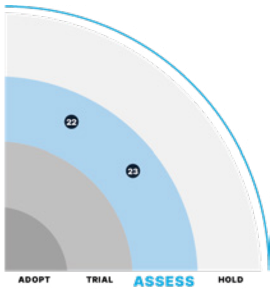
- Being independent of Flutterr
- Be compile safe

In addition, it has many qualities, such as its minimalist syntax and ability to be easily mockable and testable.

Finally, Riverpod is now closing the gap with its react react-query, offering excellent asynchronous management with loading, error, data and refreshing states out of the box, which means the UI matches each data loading state to be displayed.

While this solution won us over after several production projects, and we recommend it for managing the state of a Flutter App, we've decided to maintain its status as a trial.

Other solutions such as BLoC (present in this Tech Radar) are also recommendable. To date, we've not identified a clear winner in the field of state management for Flutter.



In Flutter, visual element testing (widgets, pages) is based on the golden test system, which consists of rendering a widget as a test in image format.

The new image is compared to the previous version when the tests are rerun. If a pixel difference is detected, the test fails and indicates a regression. It's possible to set a percentage error tolerance; however, the figure varies greatly depending on the design and the target screen. It's therefore difficult to determine a relevant tolerance threshold common to all types of design.

During a project's lifecycle, test suites are run by developers on their local machines, which run on Mac OS (a constraint for mobile developers), but also on the continuous integration machines, which we configure to run on Linux (continuous integration machines on MacOS are ~ 5 to 10 times more expensive).

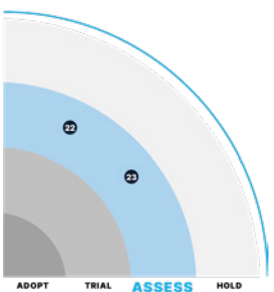
This difference in OS between development and continuous integration machines may seem trivial. Still, it becomes problematic with golden tests on textual elements.

Indeed, golden tests displaying text appear different on macOS and Linux platforms, causing false negatives in tests run in CI.

The Alchemist solution offers tools that make running golden tests in Flutter easier. One key feature is to solve the above mentioned issue by generating different golden images for local and IC tests.

Moreover, Alchemist generates several goldens within the same page for more conciseness. Finally, assigning a golden label to the golden tests allows the golden tests and the classical tests to be run independently and simultaneously to optimise the test suite's execution speed.

At BAM, we tested this library, and the results were encouraging. Nevertheless, its adoption is still marginal, with less than 200 stars on GitHub and a popularity of < 80% on pub.dev. We, therefore, recommend you keep an eye on this project to see if it continues to be adopted by the community.



23 Melos

Some project architectures are based on the concept of multi-packages. In other words, a single directory will contain several packages, each corresponding to an independently developable, testable, and deployable brick.

These packages are often interdependent, and it becomes difficult for the package manager (whatever the technology: npm, pub, composer, pip, gem ...) to resolve the dependencies of each package. And this, in a production environment and in a local development environment.

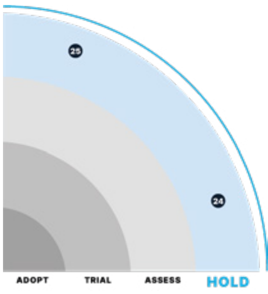
The dependencies linked to a production environment must be resolved from the Provider's site (pub.dev,npmjs.com ...). On the development side, the starting point is the local source code of each package.

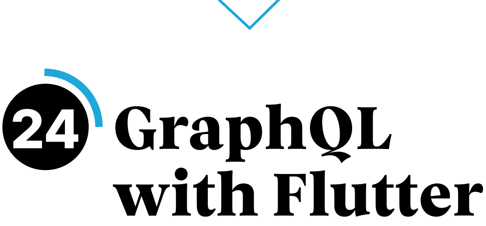
In Flutter, and more generally in Dart, this problem is solved by Melos, which offers a command line tool for these «mono-stores». It resolves dependencies with local linking, allows commands to be run on all packages (analyse, test...) and automatically provides versioning and deployment for each package for easy CI integration.

After using Melos for several months on a Flutter SDK project, we encountered difficulties debugging some CI errors caused by this tool and lost valuable time.

Furthermore, the documentation indicates that the package is under active development and not stabilised. However, it's used by big names such as FlutterFire.

Carried by the company Invertase (spearheading the open-source Flutter), we advise you to keep an eye on Melos as it's set to become a reference in its field. We're positioning it in the «Assess» category. Given the elements mentioned, this solution has not reached sufficient maturity to be blindly adopted.





The ability to retrieve data exposed by a server via an API is a crucial feature for any mobile App.

The GraphQL standard has gradually become an alternative to REST in recent years. Apps developed in Flutter can also exchange their data with a GraphQL API, thanks to 3 available solutions: graphql_flutter, ferry and artemis.

Among these 3 solutions, graphql_flutter is the most popular (with 3k GitHub stars to date). This approach, inspired by Apollo (the reference for GraphQL clients in the React ecosystem), is complete and has many advantages.

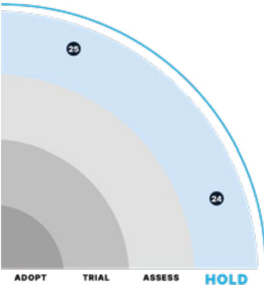
We highlight the offline management feature with cache integration, a polling/redistribution system, and optimistic result support.

However, there are some limitations regarding the .graphql files and code generation support. The documentation is relatively light and requires the use of hooks (a pattern that has not yet matured in Flutter).

The more recent ferry solution offers another approach by insisting on robust code generation integration, which allows strong typing on manipulated objects. Used by our teams on one of our latest projects and now in production, this solution has given rise to mixed feedback. The reason for this was a maintenance bottleneck (no response on several issues created by our teams and 85% of commits made by its creator).

As we've still not had the opportunity to test the Artemis solution on a project, it seems premature to offer any evaluation at this stage. Furthermore, our recent experience with the various GraphQL clients in the Flutter ecosystem doesn't allow us to highlight a solid solution to recommend.

For the time being, we're sceptical about this approach's maturity.



Like BLoC, Provider is one of the most popular State Management solutions.

Provider was designed early on by Rémi Rousselet as a wrapper around Flutter's native solution: InheritedWidget, to make it easier to use and more scalable with an easy-to-understand Provider/Consumer API.

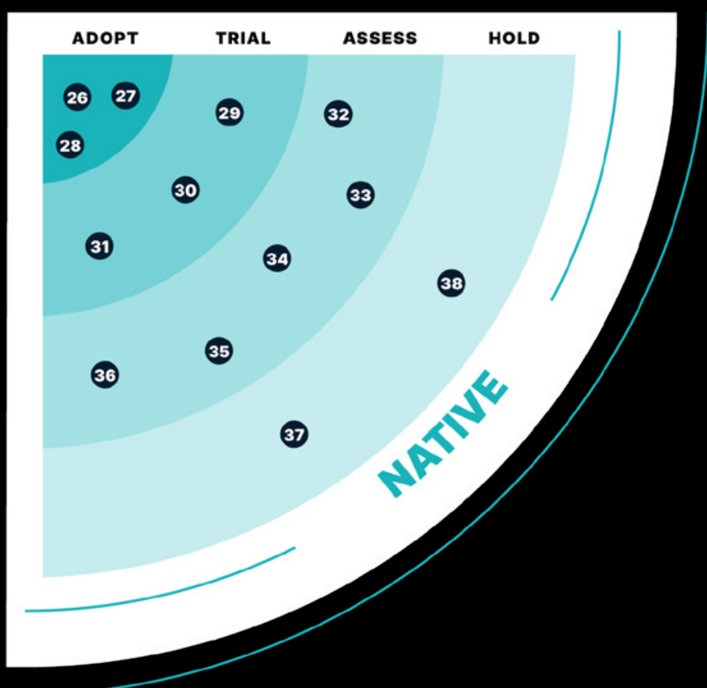
This solution is now the standard approach to state management and dependency injection for Flutter apps, as proven by the Flutter Favorite label.

Suppose Provider still appears to be a stable and solid approach to managing the state of a Flutter App. In that case, this solution has some limitations, such as the asynchronous loading of providers, the support of several providers of the same type or the runtime exceptions.

The emergence of more modern solutions such as BLoC or Riverpod, also published by the author of Provider and built essentially to solve the shortcomings of Provider, make it a second-rate solution.

In other words, if you want to use Provider, our position can be summed up into one sentence: «Use Riverpod; it's the same, but better».

08. Native Technologies



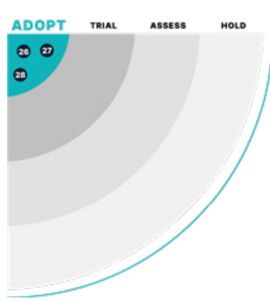
13 BLIPS

3 ADOPT
3 TRIAL
5 ASSESS
2 HOLD

To remember

On the Native side, we've focused on all the technologies we've used on our production projects. Our recommendations are based on BAM's native tribe expertise (with iOS and Android experts). We've identified three main aspects: the visual/layout area, the state and the architecture (and related DevX tools) which is an even bigger differentiating element of a good App in React Native."

Tycho / **Principal Technologist**



26 Jetpack Compose

UI frameworks are often debated: external templates or programmatic integration, graphical design tool or handwriting? And Android's historical visual layout technology is no exception. It's characterised by a life cycle that's difficult to master and cumbersome to create new graphic components.

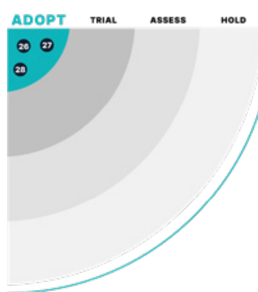
Furthermore, designed in the mid-2000s, the API graphics considered restrictions obsolete in today's smartphones and limited Android's evolution.

Therefore, Google's teams started from scratch to create Jetpack Compose, adopting a declarative and reactive API inspired by the React framework. The code is more readable, with a code structure close to what is displayed and more concise: the complexity of updating the UI state is hidden.

Jetpack Compose is complex technology. Integration relies on a Kotlin compiler plugin, a different calculation engine and numerous component libraries. All the Material design system's components have been reintegrated for Compose.

The framework's maturity is impressive. Everyday use cases are documented and easily integrated, and a solution can easily be integrated for borderline cases. However, some features may be missing compared to historical components, such as animations or navigation. There's also much enthusiasm in the developer community. The community regularly writes articles and additional libraries.

Jetpack Compose is now a safe choice for developing a new Android app. However, be careful when migrating an existing app: while Jetpack Compose easily integrates with reactive data sources (e.g., a ViewModel with LiveData, RxJava or Flow), integration on a monolithic and imperative code base could require significant modification.



The SOLID principles, especially the dependency inversion principle, are not always rigorously applied in mobile development.

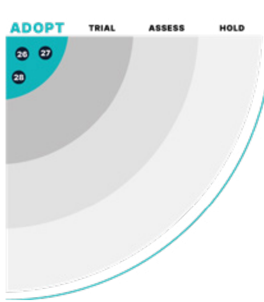
We encountered several existing projects that did not use them. For many stakeholders, mobile Apps are wrongly considered to be smaller. As a result, programming practices tend to favour writing speed over solid code.

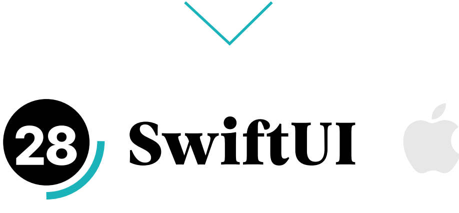
Applying these principles is sometimes worsened by a lack of good practices and tools.
Join is one of the solutions that has emerged as an answer to these issues.

This framework has a very intuitive DSL, which allows you to master the necessary settings to make an ID compatible with the basic concepts of Android in just a few minutes.

It's also more flexible and easier to learn than its main competitor, Dagger-Hilt, which is part of Jetpack.

Despite its verbosity and a slight impact on performance, Koin has become our default Kotlin DI library.





28 SwiftUI

Before the arrival of SwiftUI, there were two alternatives for doing UI in iOS:

- use UIKit, the default framework created by Apple. However, it only offers an imperative and not very readable API..
- use third-party declarative frameworks, such as Texture. However, these libraries are poorly maintained and lack documentation.

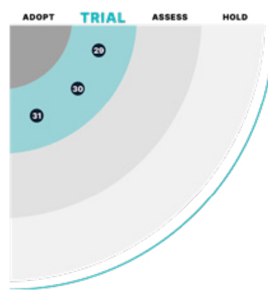
In 2020, Apple released SwiftUI, a new declarative framework to follow the trend of Jetpack Compose, React-Native and Flutter, called SwiftUI. This framework is also required for some new features, such as widgets.

It has many advantages: on the one hand, the syntax is clear, simple, and well documented, which makes it easier to learn, and on the other hand, the ecosystem is very complete, with many libraries that integrate with SwiftUI.

If it's impossible to code a specific part of the App in SwiftUI, this framework is easily interoperable with UIKit. Finally, it interfaces well with Combine, Apple's reactive programming framework.

On the other hand, this framework still has shortcomings, such as the lack of backwards compatibility with older OSes. For example, we can't use SwiftUI v2 if our users have iOS 13. Some APIs are inaccessible under SwiftUI, and you must bridge to UIKit to use them. Some bugs in the framework have been fixed in recent releases, but some bugs and DX issues, such as previews, make the experience unpredictable.

Therefore, we recommend the SwiftUI framework, but with some reservations since it's dependent on the iOS version supported by your App and sometimes requires you to code the solution in UIKit and create a bridge.

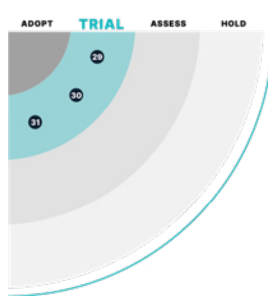


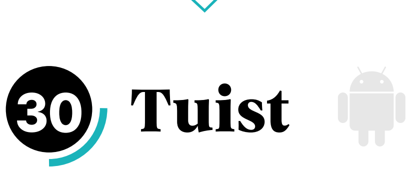
Dependency injection determines which integration will be used at runtime in a dynamic and configurable manner to make code maintainable and testable.

Among the different libraries allowing dependency injection in iOS, we recommend you use Resolver. With the Property Wrapper in your Swift files, this library will enable you to inject dependencies at runtime and is available from Swift 5.1.

Resolver doesn't generate additional files at compile time, unlike Swinjectn, for example, which makes the code more readable and allows you to use weak var to avoid introducing RetainCycle. Moreover, it offers very good testability. Finally, it's more efficient than its competitors. For example, it's 800% faster at resolving dependencies than Swinject.

However, this solution is still relatively new and lacks traction in the community. At this stage, it has only 1.8k stars vs Swinject, which has 5.3k. However, in practice, we use it successfully on projects in production.





The iOS development tools are outdated: for the launch of a new project, the «xcodeproj» format can easily reach 3000 lines or more, quickly making the content incomprehensible. It's impossible to edit manually, so you can't review it. And with multiple developers, git conflicts are inevitable.

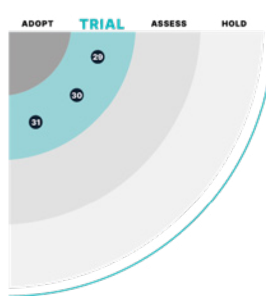
Tuist suggests not to version the «xcodeproj» project but to version a setting in Swift, which generates it on the fly.

For example, with one of our projects, we go from 1788 lines of code to 19. As a result, the file is much more readable, and conflicts disappear.

Tuist has several advantages:

- Firstly, it helps with modularisation. For a project in micro framework architecture, it automates link creation between projects in the same workspace, generates a dependency graph to document the project and provides a cache to optimise compilation.
- Secondly, it speeds up development: a page's skeleton or an API call is generated in one command that establishes the file architecture.
- Tuist also generates typed access to assets (images, videos, sounds, etc.) to avoid crashing because of a file name error.

At BAM, we use Tuist on all our new projects. The API is stable, and the community is active and always on hand to resolve issues.
The only downside is that documentation has become less consistent with the move from version 2.x to 3.x. However, this isn't a barrier (clear source code and active community).
Nevertheless, Tuist is well worth a try!



In terms of architecture, the iOS ecosystem is fragmented: there's no clearly defined reference standard at this stage.

This is because iOS documentation often advocates a straightforward architecture, which gives a lot of responsibility to the ViewController and isn't adapted to a more extensive App (example).

Have you ever seen a 1000+ line ViewController? Unfortunately, we have...

VIPER is an architecture that allows the user interface, which is part of an iOS App, to be broken down into different layers of responsibilities. Unlike «Clean architecture», it doesn't provide a flow for data integration and business logic.

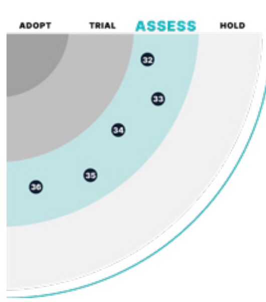
In VIPER, there are 5 architectural elements:

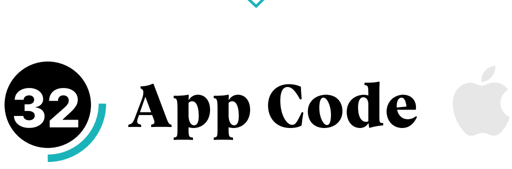
- the view (V), which manages the interface,
- the presenter (P), which links the data to the view
- the data, which comes from the entity (E)
- and are filtered out by input (I)
- finally, the router (R), which takes care of the navigation

This model may seem complex, but in practice, it presents 3 advantages:

- responsibilities are clearly divided, which improves maintenance
- each file is of average size: no more endless UIViewController
- Memory management is well-defined: the system maintains a strong reference to the view, the view to the presenter, and the presenter to the router and the inter-operator. There's no more memory leakage through hold cycles.

For your next project, we suggest you try to overcome the complexity associated with VIPER and test its architecture. It's well worth it!





The iOS community struggles to find a code editor that can adequately meet all its needs.

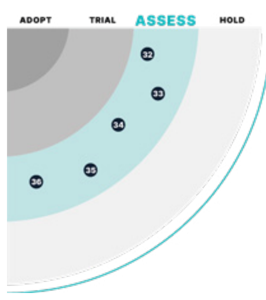
Xcode is entirely dedicated to the ecosystem, meets basic development needs, simplifies debugging, and offers a complete build system. Nevertheless, unlike other editors, its interface is quite particular regarding organisation and visuals.

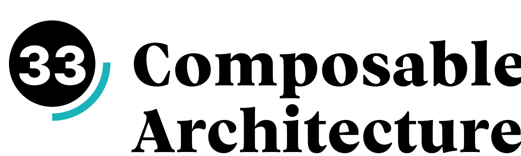
This makes the transition difficult for developers using different technologies. Xcode also offers very slow auto-completion and real-time linting; for example, it doesn't allow the addition of vim-type plugins.

Therefore, we use AppCode on our projects, in addition to Xcode. AppCode is a code editor developed by JetBrains.

It offers a superior developer experience thanks to a better thought-out interface closer to other editors, better auto-completion and linting, and extensive support, especially regarding refactoring.

However, given Apple's historical lock on its ecosystem, a risk to AppCode's sustainability will persist until Apple shows some form of approval for this kind of tool.





33 Composable Architecture

The SwiftUI framework has brought new features to UI and state management due to its declarative approach, unlike UIKit. SwiftUI provides ObservableObjects and EnvironmentObjects to externalise the state. These primitive codes are very powerful but don't guarantee architectural quality.

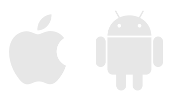
The Composable Architecture, inspired by the one-way data flow popularised by Redux in 2017, structures the global state and makes it testable and easily reusable.

In addition, the developer experience is improved because it's easier to know which event causes a given state change.

On the other hand, this makes the code sometimes verbose, especially when using non-equatable types in the state. The library is new: although it's very well documented, its integration with other libraries in the community isn't optimal. The API surface is significant; OptionalPath handles a convenient nullable state but is hard to find in the documentation

At BAM, we're starting to use Composable Architecture on some production projects. We recommend you look at this library for state management on your following SwiftUI apps.

34 µ-Features Architecture



Fifteen years ago, Adrian Cockcroft, software architect at Netflix, introduced microservices. The aim: to minimise friction within the «server» teams. The plan: separate responsibilities, optimise storage and make the developer teams more agile.

Mobile apps are becoming more and more like server Apps: local relational storage, features are increasingly complex, and there are interconnections between multiple services.

Development problems are also very similar: growing code base, increasing build time, and difficulty respecting the test pyramid.

Engineers from many companies, such as Soundcloud and JustEat, have popularised the micro-feature approach, replacing a giant bloc with different types of smaller modules:

- The main application which brings together and coordinates the UX flows.
- UX flows manage a visible part of the App (e.g., using VIPER architecture), referring to the components and navigation between pages. UX flows transfer the logic to the business logic modules.
- Business logic modules that manage a set of services and entities dedicated to the same scope.
- If needed, these modules use a core module. Core modules provide an interface for external features (such as an API call, storage, notifications, etc.) or tooling (logging module, debugging module, etc.)

There are many advantages to this. If the modules are well divided, it's possible to develop on a smaller scale. This means that business logic can be tested easily, and build time is reduced (thanks to the cache).

Furthermore, suppose the team grows, or the company develops other products. In that case, sharing a standard code between the different projects (authentication modules) is possible.

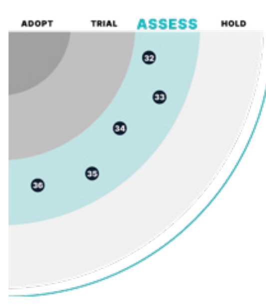
But knowing how to split modules successfully implies having significant knowledge when it comes to business logic. For example, the interface must be improved to increase consistency and reduce connecting modules. This also requires knowledge of architecture, good design, and initial investment.

And as with microservices, specific tooling is necessary to prevent the dream from becoming a nightmare.

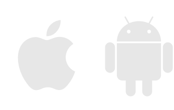
Suppose the architecture is poorly executed or the team is not well trained. In that case, the architecture can slow down the project instead of speeding it up.

One advantage of mobile App development is the ability to package micro-features into a single binary, unlike web-based microservices. Furthermore, tooling's becoming more and more stable on iOS, for example, Tuist help.

At BAM, micro-features have become a major trend when developing new iOS projects.»



35 Using relational db as state management



On the server side, the industry has been using relational databases (like Postgres and MySQL) to store data for decades.

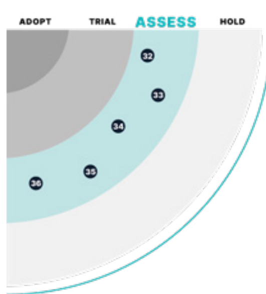
This has many advantages: structured and predictable data format, data normalisation, simplified access, advanced consistency properties (ACID), transaction management, etc.

However, on the mobile client side, there's a lag in this area. Many projects are initiated with an unstructured approach (document in Redux) and end up reintegrating structure constraints, denormalization and transactions as bugs occur, which leads to high software complexity.

We've closely followed Facebook's rewrite of the Messenger App on iOS (lightspeed project). At the heart of lightspeed is SQLite relational database: it takes care of storage, filters, and transactions. Thus, software complexity is reduced.

We're also seeing the emergence of high-level libraries based on SQLite, such as Room (Android) or Watermelon (React Native). This means it's essential to train teams on database fundamentals.

We believe that the type of data storage is an important question to ask, and we encourage you to consider using a relational database for your mobile Apps.



36 Texture

UI formatting on iOS isn't an easy task. Between written constraint code amateurs and other profiles that favour visual UI builders, no single method has been unanimously accepted until now.

React Native, however, quickly overturned the status quo by initiating the splitting of self-contained components and the use of flexboxes. The approach was later integrated for layout and inspired SwiftUI and JetpackCompose.

But how do you take advantage of these advantages on a compatible iOS App? Today, many projects support a fleet of non-W SwiftUI compatible phones or have invested too much time in a UIKit project, which explains their reluctance to change their approach drastically.

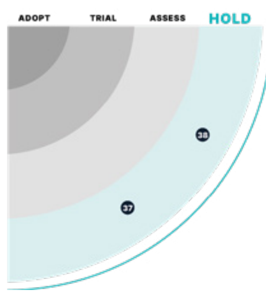
Formerly called AsyncDisplayKit, Texture is a framework created by Facebook and now developed by Pinterest.

It defines visual components and arranges them using flexboxes. The framework is compelling and concise compared to UIKit, and it's easy to connect with existing UIKit components. Furthermore, an ecosystem connects it to observable libraries like RxSwift.

The library could be documented better, but it's still ahead of Litho, for example.

Also, although Texture has little activity at this stage, it has an active Slack community to help. Texture has already proven itself on big Apps such as Pinterest, New York Times, NFL or Auxy, and we've used it successfully at BAM.

If SwiftUI constraints prevent you from using this framework now, Texture can be an excellent transitional solution.



iOS and Android mobile platforms have never been compatible: they don't use the same language (Kotlin and Swift), the same execution environment (Virtual Machine vs native execution on iOS) or the same APIs (frameworks specific to each OS).

Although it has always been technically possible to share code between the two platforms via Foreign Function Interface (FFI), these are only used for specific library codes (e.g., SQLite is usable on iOS and Android) and rarely for the App code.

An App's business code often exposes a complex data model and numerous functionalities that are tedious to interface with FFIs.

Kotlin Multiplatform Mobile promises to write business code and data layers (e.g., JSON serialisation or local database DTOs) in Kotlin and to expose these integrations through a library that can be used by each platform (a .aar for Android and a .framework for iOS)

The framework translates calls and data structures from Swift to Kotlin and vice versa. It builds a kind of SDK specific to the App; you only need to integrate each platform's specific code (the UI).

We've already integrated this technology on several projects, spending weeks evaluating it.

In each case, we decided not to continue the integration beyond the evaluation. This is because there are several drawbacks to using KMM:

- Increase in build time on the iOS App.
- Model impairment for iOS (e.g., KMM doesn't expose sealed classes with associated values, a somewhat idiomatic and popular feature of Swift)
- Limits App modularisation
- An ecosystem that isn't very dynamic (few new libraries, problems that have existed for several years without being resolved)

There are several obstacles to developing the project beyond the basic templates:

- Adding dependencies isn't always easy
- The maintenance of Gradle files is problematic: in documentation and tutorials, several formats and conventions coexist with no real harmony
- Cryptic errors, especially when integrating on iOS

Finally, some teams may have psychological barriers to integrating Kotlin code into an iOS App.

Companies that report using this technology successfully, such as CashApp in the US and Deezer in France, have highly experienced technical teams and complex products.

For smaller projects with less experienced teams, these complications may not be worth the effort and may ultimately impact maintainability.

Because of these various trade-offs and limitations, we don't recommend massively adopting this technology on projects. However, Kotlin Multiplatform can be of genuine interest if the project is suitable and if the technical teams are experienced and accept certain compromises.

:

38 Litho

Interface development on Android isn't easy. Using XML Layout is not the most intuitive. ReactNative's Flexbox-like approaches allow us to build a UI using and efficiently arranging components.

In 2017, JetpackCompose did not exist: Facebook developed Litho. This declarative framework uses a component approach to speed up interface development. This framework has the advantage of using Yoga, Facebook's layout engine, and flexboxes. Th

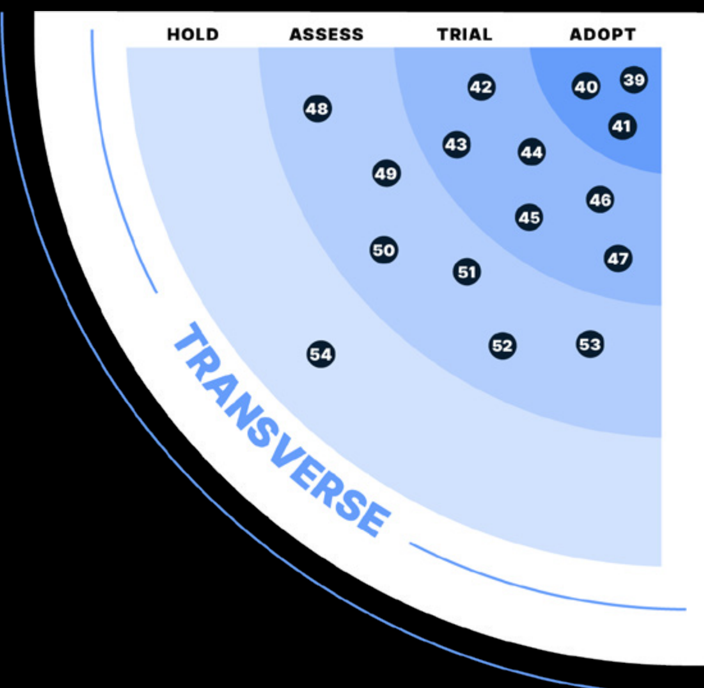
rough the ViewFlatening principle, Litho optimises component rendering performance. Litho promises to make visual interface integration easier.

However, we don't recommend using this interface on your projects, as it lacks documentation and has a very small community at this stage. Furthermore, it's not optimised for use with Kotlin, which slows development.

Facebook has indicated that it wants to change this and is working on a new version of the documentation.

JetpackCompose s'impose désormais pour gérer les interfaces sur vos projets Android.

09. Transverse



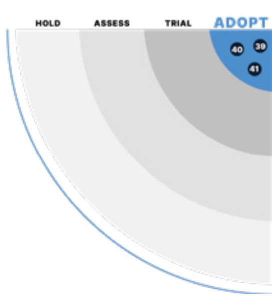
16 BLIPS

3 ADOPT
6 TRIAL
6 ASSESS
1 HOLD

To remember

This category is a way for us to share our experience, particularly the methods and approaches we've implemented at an architectural level. Some issues may be related to the 3 radar technologies, and others are more or less complex depending on the ecosystem's maturity. We consider that an ecosystem's most prominent topic is its architecture. Applications are becoming increasingly complicated and are constantly evolving. Therefore, good architecture is key to maintaining innovation at a good pace. Whether it's Facebook's Lightspeed architecture, discussions around Clean mobile Architecture or the rise of functional programming, every day, there's something new in mobile programming. We suggest you keep an eye on it very closely."

Marek / **CTO & Co-founder**



39 ADR

Very often, developers and architects must make choices on projects: languages, technologies, external libraries, architectures, etc.

We raised three issues associated with these choices:

- On the one hand, the team generally doesn't agree on choices as the same person often makes them
- On the other hand, even if the choice is consensual, it's not necessarily the best
- Finally, as the choice isn't documented, the teams who will take over the code will not understand it

An Architecture Decision Record (ADR) is a document that addresses these three issues.

It's made in a team and stored with the documentation. It aims first to explain the problem and the context and then to list all possible solutions. Finally, it seeks to define discriminating criteria and to score each solution according to each criterion. The solution with the highest total score is selected.

We've generalised the use of ADR in our teams to justify all choices that are out of the ordinary. We already have some success stories: for example, we pushed the Django server side in a project, which reduced the workload by half.

In practice, the time spent doing the ADR is more than worth it, and the teams gain skills thanks to the tool. We suggest you try it too.

40 BPMN

Many Scrum integrations have fallen into the same trap by focusing on too many user stories and by leaving the concept of a feature on the shelf for too long. As a result, a team can regularly be led to chain user stories together without having an overall vision. The effects are detrimental to productivity:

- This results in the oversight of edge cases, which lead to bugs and incomplete functionalities
- The macro estimates become unreliable, as there's a lot of feedback from one sprint to another

The core of the feature's development is done in one sprint but managing edge cases, or certain sub-courses may require several sprints. This is due to a lack of shared vision and alignment between product, design, and tech. Indeed, POs often lack the tech skills to specify the epic and find the edge cases.

At BAM, a few years ago, we introduced a one-hour workshop at the launch of every feature. This workshop brings together all the stakeholders to draw a technical-functional diagram.

Using a visual language inspired by BPMN, we draw up all the user behaviours and their technical impacts. This document then forms the basis for writing the user stories.

A shared system between teams makes it possible to :

- Agree on the exact scope of the functionality and improve common understanding
- Make alerting easier by highlighting grey areas
- Visualising functional and technical complexity
- Enable better QA and facilitate debugging

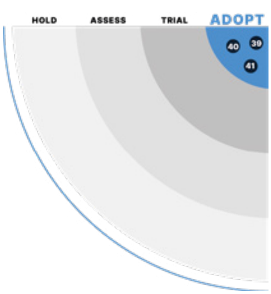
The advantages are obvious:

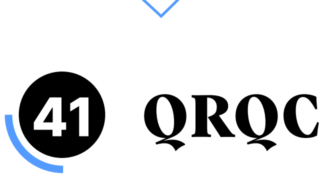
- Fewer bugs are found
- The lead time required to release a feature is decreasing
- There is a better alignment between the teams

Better estimate reliability

For example, on one of our projects, this workshop allowed us to detect a functional misunderstanding that would have cost the team a 5-week delay had it been seen during integration.

We use this approach for all complex functionalities, and we suggest you test it on all your projects.





41 QRQC

How do you immediately react to defect detection and prevent them from happening again?

For each learning, we used to add a box to be checked for our «definition of ready», which made management cumbersome. With QRQC, we discover why a process led to defect X by person Y to create the standard and provide the necessary training.

First developed in an industrial context, the Toyota Production System has enabled Toyota to reach the world's highest quality, stability, and productivity level.

Toyota's continuous improvement methods inspire Quick Response Quality Control (QRQC). As soon as a quality defect is detected, it's immediately analysed by the Tech Leader.

- Which line of code introduced this problem?
- Who? Under what actual circumstances?
- What are the root causes that led to this problem? (e.g., a lack of standards, knowledge?)
- Why was this defect not detected earlier (quality of wall)
- What could have prevented this problem from being introduced?

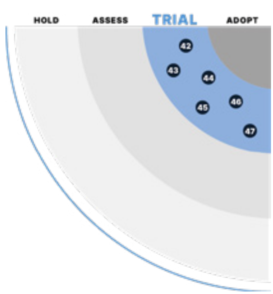
Regardless of their severity, all defects are opportunities for the team to learn, standardise work, improve tools and DevX, communicate more clearly, etc.

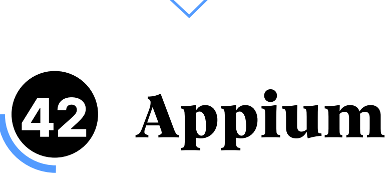
We have a QRQC animation approach: training dojos, and feedback presentations, through a standard template that makes interactions easier.

This accelerates developer and Tech Lead training and capitalises on each person's experience by sharing what they have learned. Furthermore, each new defect is an opportunity to improve our working methods as we go along.

If it's not mastered, the difficulty with this method is the tendency to integrate actions that weigh down the development process. This common mistake should be avoided.

We advocate that every Tech Lead should do at least 2 QRQCs per week - this is the best way for us to maintain continuous training and a high level of quality. Should you aim for one QRQC for every defect? Perhaps you need to find the right balance between the level of analysis, size, nature of countermeasures, etc., to maintain a good training rhythm and team motivation.





42 Appium

For e2e tests in React Native, Detox is generally widely acclaimed. At the same time, Appium is often left behind, even though it has been used for 8 years by the native community.

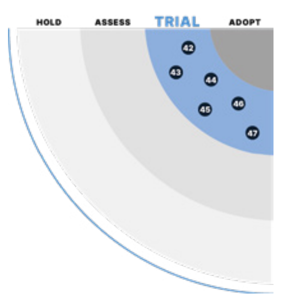
We've identified several main reasons:

- Appium has encountered compatibility issues with React Native, notably in retrieving an element by test-id (crucial). These problems have been fixed by React Native update.
- Its documentation seems obscure and complex, primarily because Appium supports many languages and platforms (mobile, web, desktop...)

However, once you know how to use Appium with Webdriver to perform the essential operations for e2e tests (scroll, click, wait for elements to appear), writing tests becomes like Detox. Appium also has significant advantages: it performs tests in black box mode. In other words, you can run an Appium test script on any app without any prior setup. This includes your release candidate or even your production app from the stores. The execution speed is supposed to be slower, but a thorough study contradicts this information.

We find that both work on our projects. If your developers already have experience with one technology, we encourage you to continue using it.

If your QAs write automated tests, Appium can allow them to do so without building the App. You may want to consider Detox's excellent documentation or its focus on mobile Apps.



43 Clean / Onion Architecture

On average, an App is rewritten every 4 years. So, how can we architect an App so that it's easily testable, maintainable, and deployable while ensuring rapid integration of future functional needs without increasing the time to market?

Each platform (iOS, Android, Flutter, React Native) uses a different approach to managing an App's view. However, the App's core is still the business logic. To isolate it from integration details and make it testable, it's necessary to :

- Segment the code base into small domains (see micro features architecture)
- Manage the business rules within these domains

Robert C. Martin, Jeffrey Palermo and other leading architects of onion architecture and clean architecture make several recommendations:

- Separate the business rules from the used technologies, which don't change in the same context. While the code related to the business rules must be understandable by someone in the business and will only change if the functional requirements change, the code related to the technologies is less stable as frameworks evolve regularly.
- Use dependency injection (e.g. via Koin or Resolver), so business rules are not dependent on technologies. This improves testability and makes infrastructure changes faster.

We've applied these architectures in the field for our iOS and Android projects, which has allowed us to make several observations:

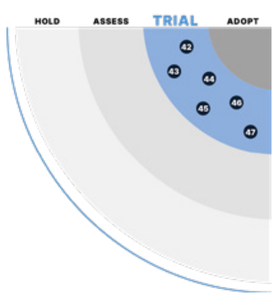
- A clearer and more understandable business code
- Simplification for unit testing of business code
- Shorter files

The ability to modify functionality without
- introducing regression
- Easier to push back the choice of frameworks/libraries to use. This prototypes an App with less complex infrastructure choices (e.g., persisting data in a .txt file rather than mounting a MySQL database).

At this stage, our teams are still studying the clean architecture App on our Flutter and React Native projects. The complexity isn't always justifiable on short projects and with more junior teams. Moreover, injecting dependencies in JavaScript isn't simple.

However, you can separate into business domains and isolate business rules using a layered architecture.

Integrating clean architecture isn't necessary if your project is simple and doesn't contain any business logic. But if your project has many complex business rules, we recommend you test the clean architecture!



44 CVSS Security Audit

Security is a core concern for every mobile App. So, how do you make a development team aware of this, so they don't create vulnerabilities through lack of knowledge or negligence?

One of the first steps is to have a clear architecture diagram, such as a level 2 C4 diagram (container).

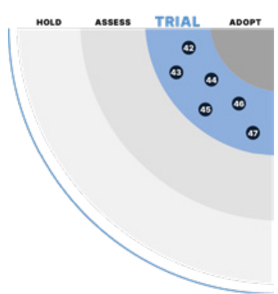
Next, teams are encouraged to ask the following questions about the main features of each block::

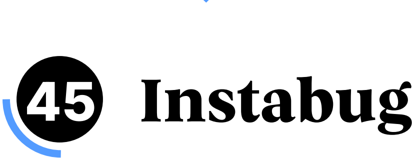
- Who: who could be the threatening agent?
- How: what would be its attack vector (direct access to the device or accessible online), and what preconditions are needed (installed malware, phishing...)?
- What impact: does it affect data confidentiality (read access to unauthorised data), data integrity (write access or unauthorised data deletion) or availability (preventing access by others)?

These questions form the basis of the Central Vulnerability Scoring System (CVSS): entries in a tool such as the CVSS calculator, produce a score ranging from 0 (no risk) to 10 (critical vulnerability). This has two positive impacts:

- These questions give teams a critical view of their App without requiring too much prior knowledge. In addition, OWASP's ranking of the most common security vulnerabilities helps teams prevent vulnerabilities from being deployed in production.
- The results also provide a stable and uniform rating system. This system is also used for the many security flaws that may be present in the downloaded dependencies. This gives a consistent rating of the project's internal and external flaws. This scoring system gives teams the tools to prioritise or arbitrate a technical decision.

We recommend that you use CVSS to evaluate the App's level of security.





45 Instabug

When a user reports a problem to customer service, the information is often incomplete or incorrect. There are several possible scenarios:

- The situation isn't straightforward, and as a developer, it's impossible to know what happened at the time of the problem
- There is also no context to the issue, such as App logs or sent requests.
- On the other hand, monitoring services don't detect all lousy user experiences

Instabug monitors and prioritises performance and stability problems in an App.

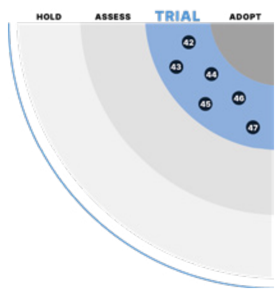
The end user can report a problem on the App by providing context (logs, requests, reproduction steps, video). Users can report a problem that isn't visible on a classic monitoring tool or for which they would not have called customer service. Users can display the reporting mode by shaking the phone, taking a screenshot, or clicking a button.

From this mode, the user can then specify the issue. This also avoids the problem being interpreted by two people (the user and customer service).

We had the opportunity to use Instabug in production. This has allowed us to get accurate feedback on problems and fix them much faster. Instabug has native SDKs, React-Native, Flutter, Cordova, Xamarin, Unity and Web. However, there's no Expo integration at this stage.

Integration is fast, and it's easy to switch services. The tool offers a priority and assignment system and integrations with classic platforms like Trello or Jira.

However, Instabug logs all HTTP payloads by default: we recommend filtering sensitive data to comply with the RGPD regulation.



4 Key Metrics

Today, software development is the engine of the world's most successful companies. For example, what do Amazon, Tesla and ING have in common?

The research (DORA) has been exploring development practices since 2013, which includes over 2000 organisations and covers all software App areas. The findings and research have been published in the book Accelerate: The Science of DevOps.

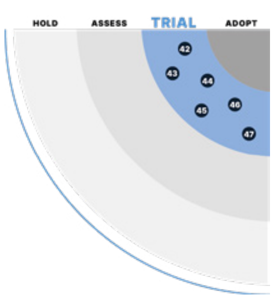
This scientific study, based on raw data, shows a strong correlation between success (productivity, profitability, growth) and four key metrics: the frequency of release, the rate of error-free releases, the lead time for a code to reach production, and the lead time to correct a production defect.

In our Lean practice, we've found genuine importance in these four key metrics, all of which promote rapid feedback loops and continuous improvement that benefit customers and teams.

This framework's strength is its rigorous approach based on many company statistics that support it. This plays a significant part in convincing decision-makers. Its weakness is that it's very high level and lacks practical examples.

We are currently working on creating a method that would help improve on these metrics. Another source of inspiration for our method is Sadao Nomuara's Dantotsu method.

Our advice: read the Accelerate book and test within your environment. We recommend that everyone apply the theory and make small changes step by step.



47 Observable

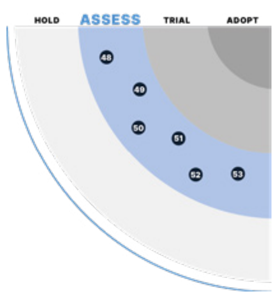
Two models are available to manage data flows in our apps: the imperative and the declarative.

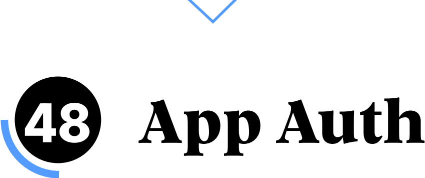
The requirement is to call the data from the server. This involves determining when the data fetch should be performed, managing the return (success, error cases etc.) and then updating the UI accordingly. This translates into heavy and longwinded code.

The declarative approach consists of subscribing to one or more data and then telling our UI to update itself if a specific data item is modified. This data to which we will subscribe represents an observable, i.e., a pipe in which events circulate. It's possible to combine these pipes by using operators, and managing complex problems then become simpler.

This model's main drawback is getting to grips with it since the mental model is very complicated and different from the imperative model.

We use this method on our projects for complex problems using libraries such as RxSwift, RxJava or Combine.





48 App Auth

Faced with increasing IT security risks in recent years, mobile App security has also become a major concern.

It’s up to the development teams to integrate existing and modern standards by using simple but secure «high-level» libraries.

Among the complex issues associated with security, authentication is a significant challenge, especially when using an external provider (social network, cloud).

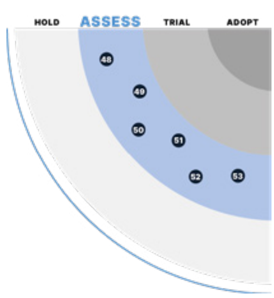
It’s essential to follow specific standards such as OAuth2 or OpenIDConnect carefully. However, there are still many grey areas in the integration details.

For example, RFC 8252 prohibits using «webviews» in section 8.12, as they present security risks and serious usability problems.

In the field, we see many projects using webviews with a deep link, universal link, or webviews with communication via the postMessage() API. These approaches have security holes.

AppAuth offers a package to make the secure integration of RFC 8252 easier. It’s easy to access via an iOS and Android library and a React Native and Flutter bridge to develop a more secure solution faster.

We want to support mobile development security awareness through library recommendations. AppAuth seems to be a fitting solution to explore for securing authentication.



49 Cognito

Integrating user authentication on your apps can be a time-consuming investment when your teams develop it from scratch.

You need to include a set of functional mechanisms from the start, such as confirmation emails or forgotten passwords, and ensure a high level of security from start to finish.

On many of our projects, we use Cognito, the AWS solution for authentication.

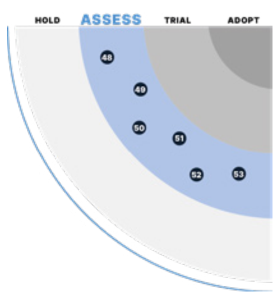
Very simple to set up, it integrates various simple use cases (such as those mentioned above) but also more complex ones adapted to your needs by interfacing with AWS Lambda or AWS Gateway.

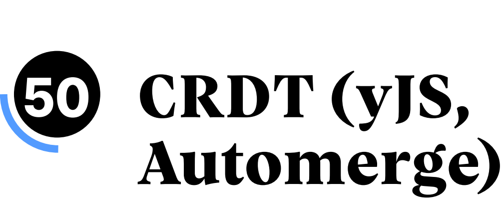
Cognito also allows intrinsic role management on your App, which gives you different access levels depending on the user. It manages strong data encryption, which is necessary for Apps with high-security stakes (e.g., health sector). Finally, it easily integrates 2FA or social login and an SSO service.

Using Amplify, Cognito also integrates UX components into your Apps. With a free tier of 50,000 users, it’s also inexpensive.

On the other hand, we’ve encountered some problems with the Flutter Cognito SDK. AWS documentation is sometimes confusing and challenging to navigate, unlike very clear documentation from more expensive solutions, such as Auth0.

We advise you to evaluate this solution’s relevance to your needs. For example, a custom soluion may suit you better if your authentication process is very basic.





50 CRDT (yJS, Automerge)

One of mobile technology's promises is the ability to collaborate while on the move. For example, how can collective contributions be managed when something is missing from the network? How to resolve network conflicts caused by competing editions? This complex issue has been researched since the 1990s.

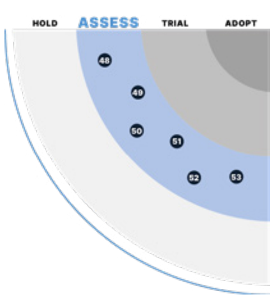
Since then, many algorithms have been developed, culminating in the CRDTs.

A CRDT (Conflict Free Resolve Datatype) is a structure for holding data and changes so that it automatically resolves the most direct conflicts and supports the resolution of complex disputes (JSON CRDT, Kleppmann 2017).

Research papers are often very complex to understand, let alone integrate. But there are ready-made libraries. Here are two in JavaScript:

- Automerge, written by Martin Kleppmann, the author of the book «Designing data-intensive Apps»
- yJS integrates a different algorithm and performs better on large documents

In any case, if you have asynchronous collaboration needs, we advise you to look at CRDTs.



51 Ghost Programming

To simplify the sequencing of technical tasks in a user story for novice developers, we encourage you to use Ghost Programming.

This technique involves inviting Ghost Programming to break down the user story into micro-tasks before development and estimate the time each task will take (5-10 min each). The Tech Leaders are called upon if the actual task duration exceeds the forecast, then this estimate is challenged. This programming practice has many advantages for juniors:

- Prepares development beforehand and anticipates blockages
- Avoids tunnel effect through better project visibility
- Quickly identifies difficulties, to react and call in the team if necessary
- Makes code commits easier and simplifies proofreading

Presenting the advantages of this technique before its integration will get your employees on board and avoid overcontrolling the project.

After experimenting with Ghost Programming at BAM, we've observed faster development time and increased autonomy from junior developers regarding technical solution writing.

We suggest you make this standard practice and share this powerful training and productivity tool with your teams.

52 Github Copilot

GitHub Copilot is a development assistant that helps you write code faster by providing autocompletion with ready-to-use code snippets. Based on OpenAI Codex, GitHub Copilot analyses comments and function or variable names to suggest integration.

We've been following this too closely. Its launch went viral and has given way to mixed reviews. In testing at BAM, we've received great feedback on several use cases, including :

- Integration of simple algorithms, e.g., division, loop, iteration, sorting
- Making teams more efficient in a language they don't master (e.g., Fastlane setting in Ruby for a TypeScript developer)
- Recommendations for typical interfaces or templates (e.g., Country, Person)
- Proposal of standard test cases in unit tests

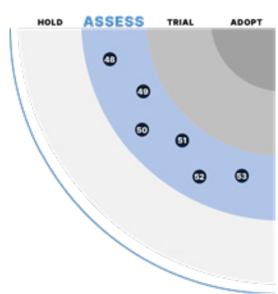
However, the positive feedback from these use cases is proven by recommendations that are not always relevant or even wrong. One of our staff felt that: «50% of the recommendations are ok, 10% are perfect, but the rest are not relevant (e.g., not the right tech, strange syntax)». Feedback from other team members was mixed, revealing a high variability in the results.

Some risks are well identified by the community but not always controlled:

- The legal risk relates to the potential use of copied code from a project with a closed licence. This would lead to intellectual property infringement. Opinions differ, but the situation isn't clear until there is a law case on this in the US.
- The other risk is related to developers working with a tendency to accept a recommendation without really checking if the code is correct (especially among junior developers) but also a feeling of loss of ownership when writing the code.

Our recommendation on this tool is a bit mixed: in some cases, Copilot can save you time, but the long-term effects are unclear. Maybe a more accurate tool using the same technology could avoid the massive use of librarian lines, like left-pad?

For the moment, we suggest you test Copilot with caution and analyse the results. Note also that some competitors are coming on the market, like Tabnine or Amazon Whisperer.



53 Libsodium

Cryptography has become a significant issue in recent years. Increased communications and continuous improvement in computing power (and soon the quantum computer) have led to new algorithms..

The first rule isn't to develop or integrate your encryption algorithms and to only use the algorithms developed by recognised experts and audited by bodies such as NIST.

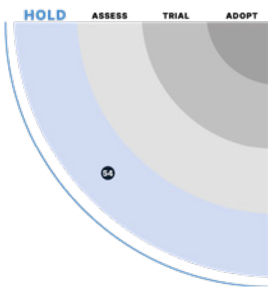
In 2021, OWASP ranked the misuse of cryptography as second highest vulnerability. And in practice, we can confirm this ranking.

So, what to do? How to choose the correct algorithm? How to use it properly?

Libsodium is an open-source project that provides an encryption library. For each use (encryption, authenticity, password backup, etc.), there's a unique algorithm documented on libsodium.gitbook.io.

For example, password backup uses the argon2i algorithm and is accessible using the sodium_crypto_pwhash function. The algorithm is chosen for its security and performance, especially on less powerful embedded hardware.

We recommend evaluating Libsodium and comparing it to other solutions. For example, Libsodium is preferable to CryptoKit (iOS) or Cryptography (Android) because it has a good algorithm for a use case, and its library prevents misuse. Having the same library on the mobile and the server is also a guarantee of compatibility. And if your needs don't allow you to choose Libsodium, you take inspiration from the algorithm choices. Libsodium is a C++ library, but React Native, Flutter, iOS, and Android packages are also available



54 Firebase Authentication

An App must be able to authenticate its users. This is a critical technical choice, which will significantly impact security, user experience and development cost. Thus, it may be tempting to use a mature third-party authentication solution that has worked through all these common issues in depth and terms of quality but also offers a solution at a competitive price.

Firebase Authentication is an attractive solution for your App to quickly solve all these problems. We've used it on several projects, some of which have high security and data privacy issues.

This solution's main advantages are its integration speed and its publisher's reputation, Google. In addition, it offers a free trial version. However, while using it, we discovered several drawbacks:

- The authentication system is complicated to replace once it's in place (vendor lock-in)
- This solution is free to a certain number of users, but after that, it becomes expensive. This should be considered depending on your ambitions for the App in terms of growth
- Particular attention must be given to managing user backups, as this data doesn't live with the rest of the App data; it's hosted on other servers. It's essential to keep this in mind and to make backups that are consistent with the rest of the datas

Firebase doesn't currently work in China. We've not yet been able to clarify whether the personal data management in Firebase is fully compliant with the GDPR.

To sum up, we believe it's best to refrain from using Firebase for user authentication. We suggest you study the long-term implications and consider another alternative (e.g., Amazon Cognito) or local integration in your project technology (e.g., in SSO, AppAuth, or OAuth).
See also : Ory Hydra ; OpenIDConnect

10. Contributors

Alexandre M. / App Performance Expert

Anaïs S. / Marketing Lead

Antoine D. / Head of React Native

Arthur L. / Head of Native

Cyril B. / Staff Engineer

François G. / VP Engineering

Guillaume D. / Head of Flutter

Louis K. / Tech Lead

Marek K. / CTO & Co-founder

Maxence L. / Tech Lead

Morgane J. / Head of Marketing

Nicolas H. / Tech Lead

Pierre C. / Tech Lead

Pierre P. / Tech Lead

Tycho T. / Principal Technologist

11. About BAM

2014 With **8 years of expertise**, BAM is the startup within the M33 group specialized in mobile app design and development.

+120 our workforce is now over **120 BAMers**, all passionate about mobile technologies and their impact on the world around us.
`

+200 **Mobile apps :** BAM has developed a real multi-sector mobile expertise by working alongside major groups and scale-ups such as TF1, Drouot, Bouygues, Promod, Urgo and Colas (...)

+15M **Users** on apps designed with our clients.