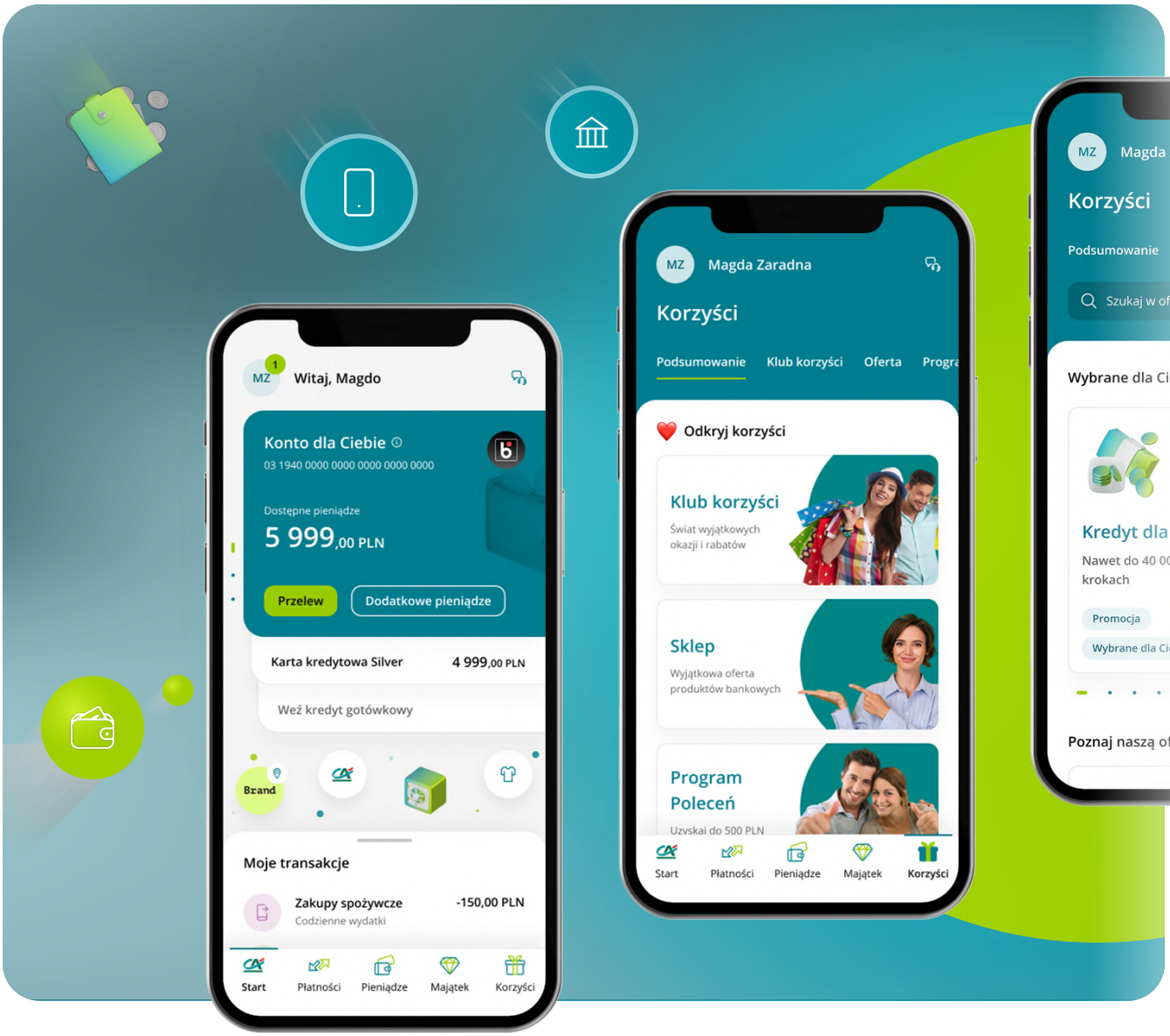
LeanCode

Building Mobile Banking Apps with Flutter

Lessons Learned From Delivering the Credit Agricole Project



LeanCode

Table of Contents

Editorial note

This ebook was created based on the experience gathered while building banking application for Credit Agricole Bank Polska and other financial institutions.

Involving around 30 Flutter Developers and 250 people of any role makes it probably one of the biggest Flutter applications in the world developed by LeanCode and other partners - with almost 6,000 Merge Requests (yes, we use GitLab), yielding over 16,000 commits.

"CA24 Mobile" is the native mobile banking application built with Flutter framework for retail clients of Credit Agricole Bank Polska, Polish branch of the globally known French international Banking Group Crédit Agricole.

MVP of the project took 12 months to accomplish with around 30 Flutter Developers and it was definitely meeting the definition of the large-scale project.

The project was divided into 12 business squads and 4 technical squads. Each developer was assigned to a squad, sometimes more than one.

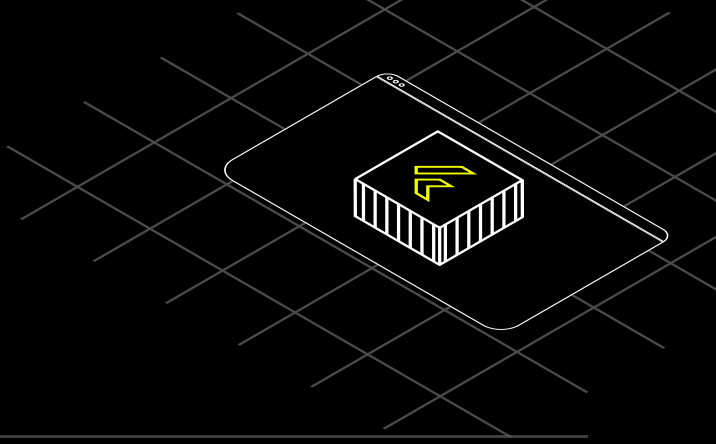
Some of the most noteworthy ones are:

- The Overall Design squad – responsible for maintaining UI components for the whole application, global appearance-related things (like themes or accessibility stuff), and also UX research and the core of cooperation designer-developer.
- The STP squad – responsible for all authentication and security-related things in the app. It is them who integrated Runtime Application Self-Protection and authorized API requests.
- The Framework squad – responsible for our core framework consisting of stuff such as the application's navigator, project's architecture, its infrastructure with all CI/CD, and other meta stuff.

The rest of the squads mainly focused on a specific business domain, like money lending, checking bank accounts, onboarding, and so on.

This ebook serves the purpose of sharing some of the learnings and pain points behind developing and maintaining such a large-scale banking app project, as well as proving that Flutter is an enterprise-ready technology.

Business perspective on Flutter



By Katarzyna Tomczyk-Czykier, Tribe Owner in the "CA24 Mobile" project and Managing Director in Channel Excellence and Omnichannel Orchestration.

The mobile banking app is called "CA24 Mobile – app full of benefits". That's how it was communicated to the market and the customers. We want to show you what exactly does full of benefits mean.

We focus on the strategy of being "100% digital and 100% human" for our customers all the time. We keep that in mind while delivering our projects, products, and service solutions. Hence the connection between the customer and the adviser, with technology in the background.

Some of the stated goals:

- top 3 among mobile banking applications in Poland;
- + 100K increase in accounts per year
- process automation with the use of bots up to 45% by the end of 2023;
- shortening the time-to-market, resulting from our idea of conducting both our projects, Seamless and Omnichannel, in the Agile model;
- top 1 in CRI (Cash Return on Investment);
- -4 percentage point of churn indicator;
- +5 percentage point of account activation ratio.

In the ever-changing world of technology and mobile devices, we needed a new mobile application, the best one available on the market. We knew about the changing customer behavior. We knew customers visit local bank branches less and less often and that we had to integrate our channels to meet customers' expectations. Our goal is to be in the narrow range of choice, so customers want to choose our offer.

Therefore we decided to create an entirely new application, starting from a blank page. The new mobile app would meet our business needs, realize the bank's strategy, allow flexible growth, integrate with other components, and lead us to the top of the market.

We created a product, a mobile banking application, and introduced it in our MVP1, expecting it'll achieve 6th place. Our final goal is to develop our CA24 app further, so the MVP2 we plan to release by the end of this year will reach the top 3. Then we'll start a significant acquisition of new customers.

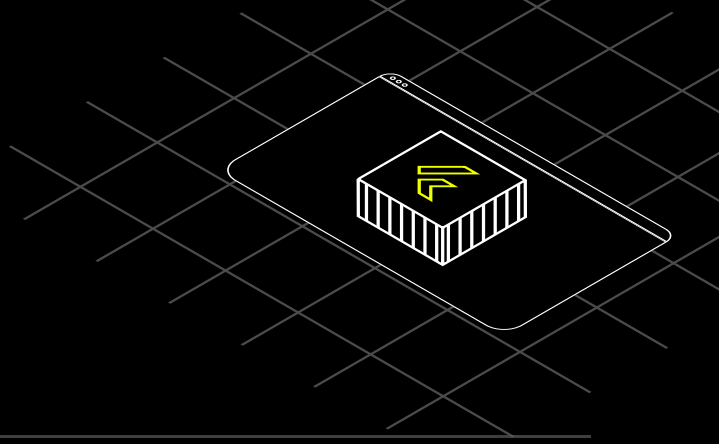
To choose our design priorities, we asked ourselves: "Which areas should we start with?" This type of product, the best product in the world, as experts would always like to offer to the customer, can't be made all at once.

We decided first to achieve that fantastic, seamless customer experience, make the customer marvel at the app, and offer those benefits. We achieved these first two goals in MVP1. We still need to deliver new processes in the new application, but it's something that requires constant development and will be included in the following MVPs.

From the project organization perspective, we first worked on a design of the application and the agency that would meet the challenge of designing something very modern. Then we searched for the technology that would allow us to deliver such an outstanding design and necessary functionalities.

Key areas: Opening an Account, Security, Product Services, Full Range of Payments and Transfers, Buy By Click, Benefit Club, Marketing Preferences.

IT perspective on Flutter



By Tomasz Czerwiński, Strategy and Architecture Department Director and Technical Tribe Owner in the "CA24 Mobile" project.

We were looking for a technology to support us in the business context. We wanted to know if using a native solution and developing it for every platform was better. Or was a cross-platform solution a more suitable choice for our case?

We had some negative experiences with building an application in native mode. Our maintenance costs started to increase. Apps started diverging because of the different platforms. It was also challenging to manage an appropriate time-to-market. The application was basically “dead” in terms of development and improvements. We knew our application required building something from the ground up.

Back then, React Native was considered a more mature and popular solution when we made the decision. However, we wanted to bet on something more innovative. More modern. Something that would accelerate our product and strategy development and offer a shorter way, so we started considering cross-platform solutions.

We decided to do a PoC (Proof of Concept). We did three PoCs with three different companies, including LeanCode. The PoCs showed us it was both an efficient and effective solution and that the development was fast – the 120 FPS was available to us. An operational demo version of the app answered all of our requests and needs.

We would say what we needed on Friday, and on Monday, we’d have a demo version. That showed what we meant – the effectiveness and high performance on the two platforms (iOS and Android).

Start of our decision path – Validation criteria:

- Product maturity
- Future vision
- UX/UI opportunities
- Development and maintenance costs
- Resource availability and potential
- Security

Finally, despite various recommendations not necessarily supporting our choice of Flutter, we chose the best technology for our case. It’s Flutter, obviously, and we’re happy about this decision. It was a hard one involving various risks, but we decided to risk it, considering the good risk analysis.

We also think that another factor of our success was that it was an innovative project, considering Flutter at scale and that we decided to deliver a mobile banking app in such a short time with such an investment. The Agile approach was something new for us. On such a scale, it would be new for many organizations.

This project was organized in the Agile model of work. There were over 200 people involved. We’re talking about a tribe, a concept made by Spotify. In our case, it was a “village” of more than 200 people. About 100 of them were in IT. The Flutter team had more than 20 people, sometimes even around 30. We also had 20+ backend developers.

Introduction to the project

When we talk about scale, we have to define what it really means in the context of a mobile application. There are projects at LeanCode that we call either “small” or “large”. It happens that we take a completely different approach when it comes to those.

By developers working in the 13 Framework squad at the "CA24 Mobile" project:

Mateusz Wojtczak,
Senior Flutter Developer &
Head of Mobile at LeanCode

Jakub Fijałkowski,
Senior Backend Developer &
Head of Backend at LeanCode

Robert Odrowąż-Sypniewski,
Flutter Developer at LeanCode.

When it comes to small projects, it’s easier to keep the entire process of their development under control. We can assume that we are aware of pretty much everything project-related, and all development practices are dictated by the greater good of the client and care for the budget. These projects tend to hold tight even if we don’t keep to the strict rules of project management.

We can see our development sins, but we’re confident that if there’s the right time, it’ll be fairly easy to refactor them. Of course, we always want to keep our code simple, clean, and easy to change. We would like it to be loosely coupled and well structured, so we do constant refactoring and always keep it top-notch.

However, we all know that it’s a pursuit of perfection that can become pretty frustrating if we don’t take any measures to build some safety net around it.

When we’re talking about large projects, we mean a project spread across different teams, companies, and professions because development is only a small part of the whole.

We often know every developer working with us on small projects. There’s even a huge possibility that a backend dev that serves us some API sits next to us the whole time.

It’s much easier to ignore organizational issues if you don’t see them because the team sits in a single room. We don’t benefit so much from building powerful abstractions when we can go get some coffee with the team and discuss everything ad-hoc.

So, in large projects, we just want to help ourselves – to deal with multiple obstacles that we cannot directly resolve - obstacles that are not only related to code. Let’s see how we can wrap our heads around this level of complexity not only in terms of code but, most of all – in terms of communication.

In this study, we would like to share our thoughts and experiences from our almost-a-year-long journey of developing a large banking mobile application with Flutter.

Perception of code ownership in large projects

Let's talk about code ownership. In a small project by a single team, there isn't too much to speak of (or is there?). The team wrote it; the team owns it and is responsible for it. However, what if we are talking about large project?

In a small project, a team may split the responsibility for the code further internally, but it doesn't change the external perception – the team owns the code. The reality might be slightly different, but it will boil down to this variation. If the team doesn't own the code, it will deteriorate quickly.

In the case of the CA24 Mobile app, at the highest point, 30 mobile devs were working on the project split into 12 business teams (also known as squads). We might say that this is just a simple extrapolation – each team owns the code they create. That would be true. Mostly.

The problem arises when we realize that the teams work on a single app and single codebase. A. Single. Codebase. So, by another extension, if every team creates the code in the same codebase, then is every team responsible for everything? If that's the case, then no one will be responsible for anything.

There might be places where some people will feel responsible for their piece, but that will be an exception. If you find a bug, nobody will be willing to fix it because that will always be someone else's problem. If you need another feature, you will need to write it yourself because no one cares.

We need to follow the limit and set boundaries exactly where they should be. In a project of this size, these will not come out naturally. We need to put them down and abide by the rules explicitly. If we do this right from the beginning, it will not feel artificial and won't be questioned.

After all, the rule is natural. Everyone feels responsible for their work, but we cannot allow a situation where too many people claim ownership of the same piece. That is the problem, and we need to overcome it by introducing explicit code ownership.

Let's put some ground rules on how we define code ownership at LeanCode in a Flutter project of this size.

First and foremost, every line of code is owned by a team, not by a single person. Even if the team consists of a single developer, it is still the whole team's responsibility to maintain the code. Developers are rarely working autonomously, and someone else (usually "business people") always prioritizes the work.

That's why a single person should never be responsible for a piece of code. If that happens, introducing any code changes will require getting approvals from the developer's supervisors that have much different priorities than we have. Teams are part of a structure at a level that gives them enough power to correctly prioritize the work around the code and pay for it by investing hours. A single person cannot do that.

Even if a team has a single dev and they switch teams (or leave), the responsibility for the code stays within the said team. A situation where a team gets dismissed or the whole staff of a team leaves happens extremely rarely. There will always remain a person inside a team who owns a piece of code (even from a business perspective).

And by the responsibility of the code, we mean that the team maintains the code. If a bug is found in a piece of code owned by the team, their obligation is to reserve time for a fix and do it. They also must ensure that the code adheres to the always - changing coding standards. It doesn't necessarily mean that only they are allowed to add new code lines there. If there is a need, someone else outside the team can edit the code – but that does not change the one responsible; it's still the team.

And although we are rather pragmatic when it comes to all the rules, meaning that we follow the "there can always be exceptions" motto, this is the one that we believe should be followed with religious devotion. Otherwise, problems arise.

The "no exceptions" rule is critical here. If we don't abide, no matter if we allow some piece of code to be owned by anyone or just a couple of different teams, the ownership will get blurred and lost. The quality of the code will deteriorate.

Then some bugs will emerge, or there will appear a need to do a technical change (there probably won't be a need for business change as no one owns the code; hence no one cares about its business value), and there will be no one willing to take care of it. Effectively the code will be abandoned (or one of the teams will be forced to own it).

We know we need to preserve ownership, but how should we define who owns which part of the code?

When a whole application is developed by a single team (be it a small mobile app or a microservice on the backend), the rule is simple: they own everything they create. There will be some shared code in internal libraries, but these will be created and maintained by a different team, and we probably won't see the code at all – we will consume it.

An extensive Flutter application that is developed at a rapid pace is not that simple. Effectively everyone works on a single codebase, sometimes with not-so-clean business responsibilities.

Of course, we could introduce very clean boundaries and physically separate the teams, but that would introduce so much friction and tedious work for the person responsible for putting it all together that the development would grind to a halt. We need to find a different way.

And, to be honest, the answer is not that far away – let's split the project into packages, the same way we would do when separating physically, but keep everything in the same repository and reference (& compose) using just regular path references.

This will make the boundaries clean, but the walls won't be so high – you can always look into other teams' code without any fuss because it's right there, just a click away. You also compile everything at once, so provided that you enforce that on CI (and you should!), everything will work more or less at all times. You will also, just by random chance, test other people's code.

Nevertheless, code ownership is not a silver bullet. This approach has problems that need to be overcome. For once, because the walls are not that high, there is the possibility that someone will modify the code behind the owners' back. This might be a problem when your team does not want to adhere to set rules, but it is primarily an organizational issue – and I would say that if that happens, it means that you have other, more critical problems.

In every project, there always is a shared folder for files that are used across the teams. A place where the local packages are bootstrapped and combined to create the final application binary. We don't want these kinds of packages because they don't have clear ownership.

Everyone writes code there (directly or indirectly), but no one feels responsible for it. We also can't ditch these completely because we need to have some shared ground. Otherwise, we would re-implement the same thing repeatedly (which will happen, but with the common folder – at a much smaller scale).

We also need an "app" package that brings everything together. This one is even more important than the first one - we can't replicate it. We are required to have a single entry point to the app. And everyone needs to contribute to the app.

We cannot make an exception from our main rule for these two cases. Not abiding will be far more disastrous than sticking to our guns. It means that we need to find a way for us to make it work.

We found out that finding some team that naturally fits to be the owner of this kind of code is not the best idea. What worked for us was to create an "artificial" technical team. This team was responsible for every left-out piece of code.

Don't mistake it for a team of exiles, a team where you end up doing only the work that no one else wants to do. They have a much more important role.

At first, it might seem like a bad idea. This "artificial" technical team doesn't bring any clear business value. It looks like a team that no one needs – no business Product Owner, and they don't create any end-user features. You can't easily plan their work in sprints.

However, their work is vital. You can't possibly allow 30+ people to work on a single codebase and expect that they will come up with a coherent, maintainable architecture. That people will never stomp on each other's toes. They will spontaneously combine everything into a single application.

That won't happen just like that. Natural-born leaders might be willing to do this work because they are best suited for it, but leaving that to chance will make the process longer and more painful.

Every project of this size requires a team of "architects" that overlooks the development, puts the basics in place, and sets the ground rules for development. This team is responsible for the code that no one needs, and they are responsible for making sure that everything works together.

They put in the ground rules and ensure they are followed. They design how to (technically) communicate and try to satisfy every developer's need that is not directly related to the business.

Keeping up with every developer's needs can be exhausting. And that is not the only responsibility of the technical team. As said before, they are responsible for the overall architecture and keeping everything in proper order.

Although it might not seem as much, that is a lot of work. Every minor technical bug, every small optimization, and every feature required to make the app work and be maintainable will ultimately be their responsibility. Although not at all times, the technical team will have plenty of work.

Because at the end of the day, someone has to be responsible. And if no one wants to be responsible, the technical team will be.

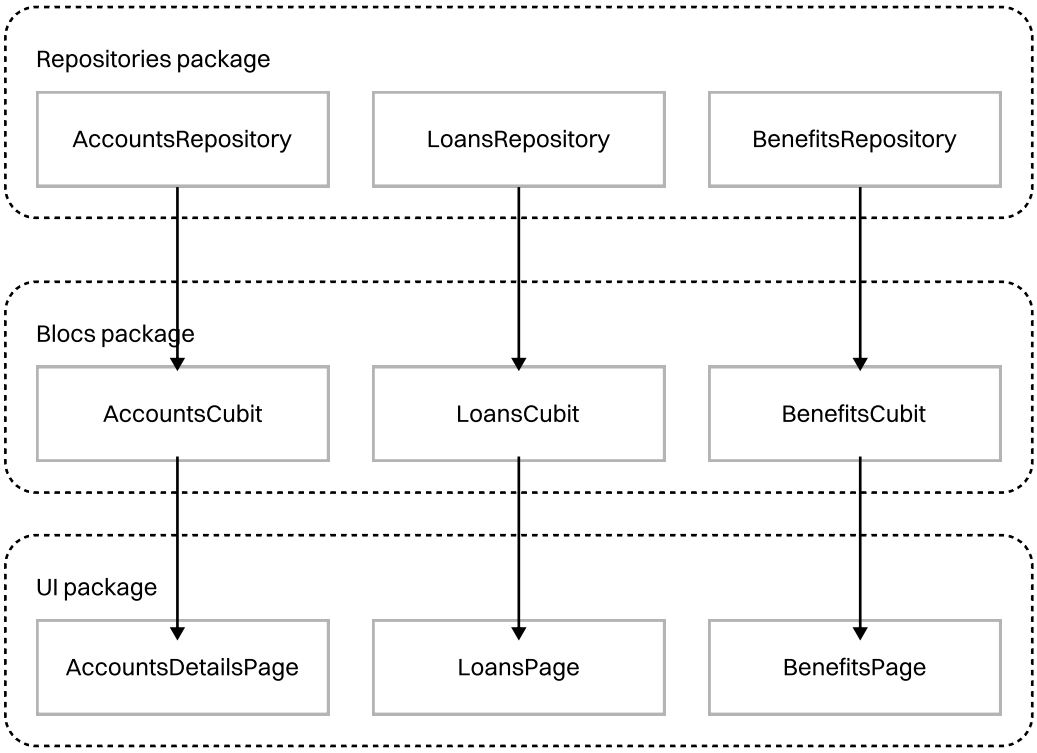
Feature-based project structure

Starting the project often begins with us thinking about how to structure this one better than the previous one. There are multiple recommendations for structuring code. We have a lot of architectural patterns that enforce or promote some, but these can be summed up into two main approaches – horizontal and vertical structures.

You may have heard about layering your code. It's a common thing that people propose to divide code into layers that handle different abstractions. The layers often go from the end-user (UI layer) through some view & application logic until they reach pure data – meaning some domain logic or data contracts. That's what we call a horizontal approach.

As we can see in the diagram, in this example, we have a UI layer with some Flutter widgets, an application logic layer with some Blocs/Cubits, and a data layer with repositories representing our data sources. While the horizontal approach may not seem so bad, it can corrupt our project in the long run. Let's see what that architecture means.

Why do we structure the code? To organize it in a better way. Why do we organize? To communicate better. That's something that we can build our code structure on. As said in previous chapters, we need code ownership. So, let's try to answer this question: Who owns the repository package? Who owns the widgets package?



Well, everyone and no one, of course. We could bring an iconic software aphorism in here - Conway's Law. It states that:

Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.
— Melvin E. Conway

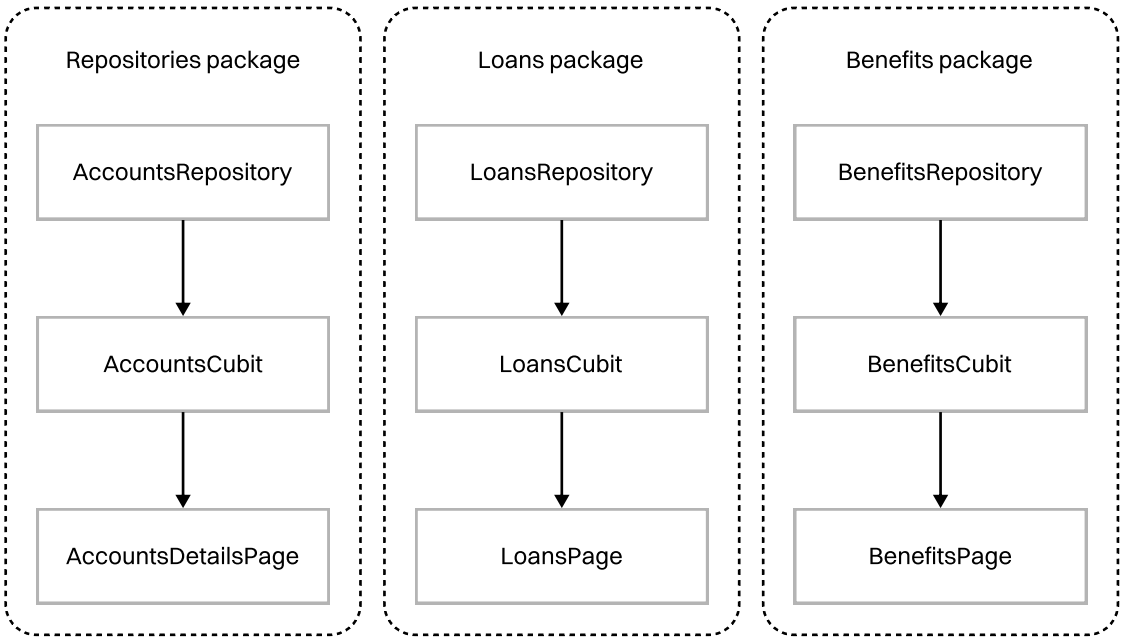
What does it mean for us? It means that no matter how we structure the code, it would still be shaped similarly to the organization. That means we would like to project our organization's communication structure onto our code so that we don't have to artificially divide our code and do it as we do with the rest of our work.

In that case, let's ask ourselves: how do we communicate? Well, is there a "Repository Daily Meeting"? Is there a "Widgets Daily Meeting"? Not at all.

The team is most likely divided into business squads with responsibilities according to some business domains. In that way, there are people in squads that are crucial to progress but have nothing to do with software development.

Ok, so we have our "Loans Squad Daily Meeting." We can talk about business requirements, and as developers, we can extract the business knowledge from other people and process it so that we know how to build the software around that.

In this domain-driven workflow, let's come back to code structure. Let's try to project our communication model onto that:



Now we can see that if the same developers work on anything related to loans, they don't have to work outside their packages. This way, we also minimize code conflicts between squads.

Our daily work is going to be revolving around those packages that are directly owned by our squad. That's what we call a vertical approach, and it allowed us to maintain a large project code structure.

Also, this example shows a 1:1 relation between squads and packages. It's a bit of a different rule. As with previous code ownership assumptions, we allow multiple packages per squad. Usually, some larger business domains are still strictly related to the squad, but devs need another package for that.

For example, the "Onboarding Squad" can have their onboarding customer journey and a KYC (Know Your Customer) module that is a separate thing. It can be organized in a different package. The important thing is to remember that one squad can own many packages, but every package should have only one owner.

However, this doesn't exhaust the subject of code structure.

What about the actual code inside the package?

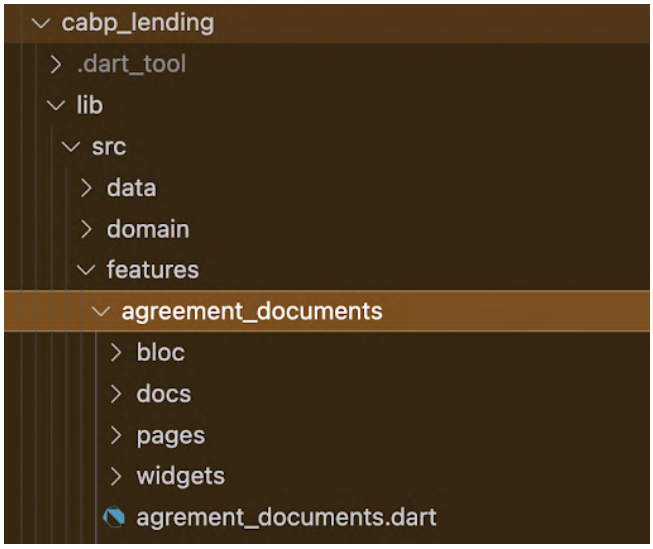
It's also essential to have some guidelines here. That's because you want your code structure to be predictable. When one developer swaps squads with another, you don't want them to spend weeks studying the code before anything happens.

How can we organize it inside the package?

Let's use the same assumptions as with package layering. Let's try to make a vertical approach inside packages. To do that, we need to define the core unit of our work: in our example, it's a feature.

In agile development, a developer works on a user story enabling users to do something related to the business. If we want to let users see their loan agreement documents, we have to do everything from UI to data sources because otherwise, the feature doesn't make sense.

So, we can define our features inside a package, and inside every feature, we can split our work into blocs, widgets, pages, data classes, etc. What happens inside a feature directory stays there. For other devs, it's a black box as long as they don't need to go in there.



Running tasks in monorepo

The next crucial thing in large projects is simplicity and consistency in daily work. What we mean by that is that we don't want to have trouble restoring dependencies in our package.

A developer would wish to have a simple way of fulfilling their daily tasks, running tests just in their packages, running formatting.

We need a task runner like "Make" or "*insert your favorite language*ake" to run all these things in a multi-package environment. It would be convenient if we had a task runner that is somehow aware of Dart/Flutter packages and is adjusted to the ecosystem we work in.

Fortunately, we found out that there already are tools for that purpose. One that we have selected is Melos. It's officially described as *"A tool for managing Dart projects with multiple packages"*. That's pretty much what we need.

Melos provides a couple of scripts itself, such as:

- Automatic versioning & changelog generation.
- Automated publishing of packages to pub.dev.
- Local package linking and installation.
- Executing simultaneous commands across packages.
- Listing of local packages & their dependencies.

```yaml
name: banking_app
packages:
  - packages/**
ignore:
  - synthetic_package
scripts:
  analyze: melos exec -- dart analyze
  get:
    run: melos exec -- flutter pub get
  test:
    description: Run tests in a specific package.
    run: melos exec -- flutter test --reporter expanded
    select-package:
      dir-exists:
        - "test/"
      ignore:
        - "flutter_test"
  generate:
    run: melos exec -c 1 -- exec flutter pub run build_runner build --delete-conflicting-
outputs
    select-package:
      depends-on:
        - "build_runner"
  l10n:
    run: melos exec -- flutter gen-l10n && melos exec -- flutter format .
    select-package:
      scope:
        - "banking_app_l10n"
  format:
    run: melos exec -- exec flutter format .
    select-package:
```

A typical Melos default file structure looks something like this:

```
my-melos-repo/
 melos.yaml
 packages/
  package-1/
   pubspec.yaml
  package-2/
   pubspec.yaml
```

The location of your packages can be configured via the melos.yaml configuration file if the default is unsuitable.

It helps maintain many popular packages like FlutterFire (Firebase libraries for Flutter), Flame (game engine), or Flutter Community Plus plugins. However, the most important thing is that you can run any shell script in each package directory and filter what packages it should consider.

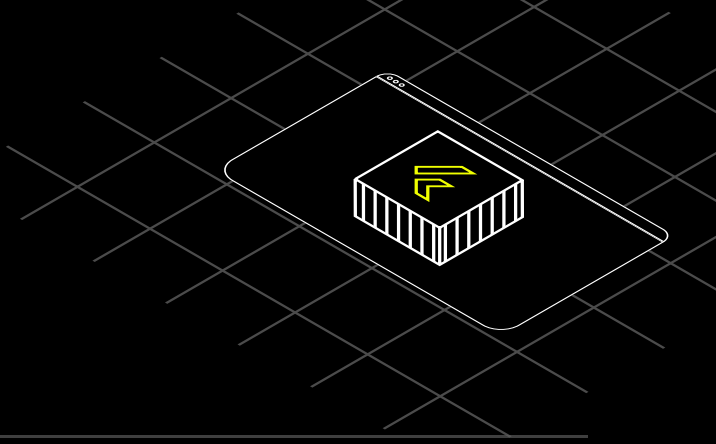
It's helpful because you can define such scripts and filters once, write some simple docs (although the melos.yaml file is already pretty convenient to read), and help new developers on your team be quickly onboarded in the environment. A task runner also simplifies building a CI/CD pipeline because some tasks are already defined there.

Last but not least, you can use concurrency for your scripts. But beware of that because your Flutter scripts may fail if they are run in parallel. In some cases, there will be locks, so even if you start it concurrently, the subsequent tasks still have to wait before the other ends.

Also, Melos's command output can end up messy when many packages are being considered since it doesn't maintain the order – simply speaking, the output is mangled from all the executions.

Still, it's a great tool, and you probably should use it, especially when working on large Flutter projects.

Getting started with Melos



By Majid Hajian, Head of DevRel at Invertase and a Google Developer expert, an award-winning book author, Flutter, PWA, perf enthusiast, and a passionate software developer with years of developing and architecting complex web and mobile applications.

Melos requires a few one-off steps to be completed before you can start using it. Melos can be installed as a global package via pub.dev:

```dart pub global activate melos```

To set up your project to use Melos, create a melos.yaml file in the root of the project. Within the melos.yaml file, add name and packages fields:

```
name: my_project
Packages:
 - packages/**
```

The packages list should contain paths to the individual packages within your project.
Each path can be defined using the glob pattern expansion format.

Consider setting usePubspecOverrides to true, if you are using Dart 2.17.0 or greater:

```
command:
 bootstrap:
  usePubspecOverrides: true
```

This enables a new mechanism for linking local packages, which integrates better with other tooling (e.g. dart tool, flutter tool, IDE plugins) than the mechanism currently being used by default.

Please read the documentation for usePubspecOverrides before enabling this feature.

Once installed & setup, Melos needs to be bootstrapped. Bootstrapping has 2 primary roles:

1. Installing all package dependencies (internally using pub get).
2. Locally linking any packages together.

Once successfully bootstrapped, you can develop your packages side-by-side with changes to a single package immediately reflecting across other dependent packages.

Melos also provides other helpful features such as running scripts across all packages.

For example, to run dart analyzer in each package, add a new script item in your melos.yaml:

```
name: my_project

packages:
 - packages/**

scripts:
 analyze:
  exec: dart analyze .
```

Then execute the command by running melos run analyze.
```

Cross-squad communication

The physical split into (local) packages does not solve all the problems. Unfortunately, it even amplifies other issues. Life is not simple; although you might have a "clean-cut" when it comes to team responsibilities, you must assume that there will be a point when a person might need someone else's help.

With twelve squads and a banking app, some features depend on data from other squads, and some invalidate data in others. You can't possibly have an app that has 12 completely separate features.

Take, for example, the accounts in a banking app – it is the most basic feature to see a list of accounts you own. Every feature depends on it this way or another.

For example, when you see a transactions list (a basic feature for a bank app), you must see it in the context of an account overview. On the other hand, when a payment is made, it changes the account balance, which is yet another account property.

As you can see, there always will be dependencies between teams. What is worse, the dependencies will not be unidirectional. There might be cases where two teams need to cooperate with each other in both directions.

An example: you need some data (must be synchronous), then you do something and expect someone else to update the data mentioned above (might be asynchronous).

But that is just one example and arguably simpler (to maintain, not to develop) because you can quickly invert the control by, e.g., using events, resulting in effectively unidirectional dependence.

But there are cases where you need access in both directions. A squad worked on providing basic authorization functionality, which was later used by arguably every squad.

On the other hand, when displaying some auth-related configurations, we needed the information from accounts. This couldn't be made asynchronous – everything needs to be provided now. And we have a cyclic dependency that, although solvable without too much fuss, is another thing to maintain.

And because of the plethora of options, we need to put some constraints there so we can do everything easily. And to make it last.

Our solution to that is quite simple. We divide the communication into two groups – when we need some data or we need to make an action "right now."
It covers, for example, the authorization mentioned above or getting the accounts list; and asynchronous, where we invert the dependency and expose that some action occurred as an event. The other squad can subscribe to it and react, updating their data.
It covers updating the account (a reaction in the accounts team) to a new transaction (event raised by the payments squad).

We also put some more concrete constraints there. For example, the synchronous facade should expose the data as streams (backed by `BehaviorSubject`) so that integration and auto-update (after an event) are easily doable.

Because of that, the data should be automatically provided (when requested), and every failure should be automatically reattempted. The error should not, in most cases, be visible through a facade because you can't react to it sensibly – and if you allow reacting to it, you can easily react too much. Multiple teams will be shown the same error message.

The other kind of interaction, where you don't need data but need to do something, is even simpler - you expose a properly named & self-contained method that does what you need, the way you need. Making more rules here is unnecessary.

The other kind of facade, the asynchronous one, is mostly similar, except it does even less. If you do some sort of action, you expose an event that describes what occurred. For example, when the user makes a payment, you expose a data package that tells you the source and destination accounts, the title, and what amount it covers. Nothing more is revealed.

You also don't store the events – we're not doing event sourcing or any persistent-event architecture. After all, every event we publish is meant to update some UI or trigger data reload. The heavy lifting is done on the backend.

Navigation

Another form of cross-squad communication is cross-squad navigation. In such a vast domain split between twelve different business squads, navigating through many pages in the applications will be prepared by another squad.

For example, from the page presenting details of a bank account that is owned by the accounts squad, there must be a way to seamlessly navigate to bank transfer which is owned by the payments squad. Such a navigation use-case needs to be addressed while adhering to the code ownership principles and code separation between multiple dart packages.

To achieve those objectives, we implemented custom navigation based on the Flutter standard navigation with the following objectives:

- Separation of navigation targets from the page implementation
- Allowing for passing the context data between pages
- Separation of a navigator from Flutter

Separation from page implementation

To allow for navigation between different squads without exposing the specific page implementation between pages and squads, the page definition has been split between the globally visible target and package-private builder, which builds the page based on data from the target.

The targets are somewhat similar to intents known from the android development world. A target is a class representing a specific page in the app. It can also contain additional context data that needs to be provided to build the page.

For example, the target for the bank account details page will contain the bank account number of the account that is meant to be displayed.

```dart
class AccountDetailsPageTarget extends DialogTarget {
  const AccountDetailsPageTarget({this.nrbNumber});

  final String? nrbNumber;
}
```

The target is an identifier necessary to execute the navigation; it must be available to all squads who might need to navigate the page. Thus, the targets are defined in the central navigation package.

Separation from Flutter

Navigation is part of business logic; because of this, it should be available from the bloc/cubit business logic code. However, usually, we avoid introducing a dependency on flutter in business code.

This constraint means we have to somehow trigger the navigation on the view side. It is usually achieved by creating some pseudo state or introducing an additional stream for navigation events.

In our solution, because the navigator state is independent of Flutter, we have no issue passing it to the business logic, and we can execute navigation right from the bloc/cubit.

```dart
void _navigateOnResult(ApiResponse<void> result) {
  result.maybeMap(
    success: (_) => _appNavigator.push(
      const AgreementDocsSuccessPageTarget(),
    ),
    systemError: (_) {},
    orElse: () => _appNavigator.push(
      const AgreementDocsUploadErrorPageTarget(),
    ),
  );
}
```

Localization & Translation Management Systems

The next important thing when developing large Flutter applications is being available to people from different cultures, locales, and countries. Thus, we need a way to set up localization (often put as l10n).

Localization "is the process of adapting a product's translation to a specific country or region." It's a part of a more extensive process called internationalization (or i14n).

Hundreds or thousands of string values are used across the whole app, and they have to be translated according to the current user locale on their device or personal setting within the app. In our case, we needed over 5500 string values in two languages for the first release, but we also needed the possibility of adding more languages later at a low cost.

That's why you need a localization solution.

Flutter already has a preferred solution for l10n. There are two main packages from the Flutter team: one called flutter_localizations (for l10n) and another called intl (for i14n). These two are connected and even work together to make your app available to everyone. The most popular approach is to use those packages along with .arb format files that contain "key-value" information about each localized string.

```
"aboutAppPageApplicationAdditionalInfoSection": "Additional information",
"@aboutAppPageApplicationAdditionalInfoSection": {
  "type": "text"
},
"aboutAppPageApplicationDataSection": "App data",
"@aboutAppPageApplicationDataSection": {
  "type": "text"
},
```

The problem is that information about what to localize and how to do it doesn't come from the development. In most cases, it's business knowledge that has been processed by many people in the organization: marketing people, product owners, translators, and others. Developers are only one small part of that.

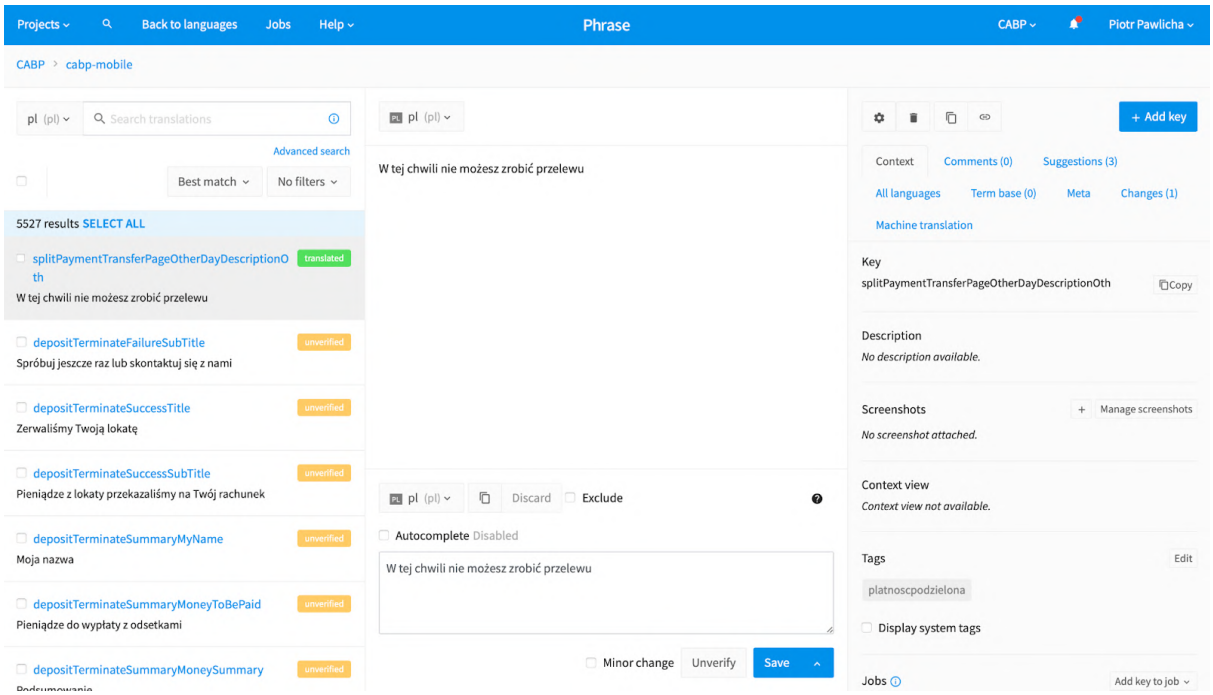
It gives us a reason to think that localization is no longer a development thing. It's a product thing. Thus, localization files stored in code repositories should not be a single source of truth. Referring back to the rule of code ownership, localization should also be owned following the same rule, and that ownership should be transferred to business people.

Fortunately, there already are tools meant for that purpose. They are called Translation Management Systems and typically are web apps optimized for localization workflows.

In our project, we used Phrase as a TMS tool because it supports the .arb format and has many convenient features like comments, activity tracking, roles, and advanced translation workflows.

TMS tools can help you with things like:

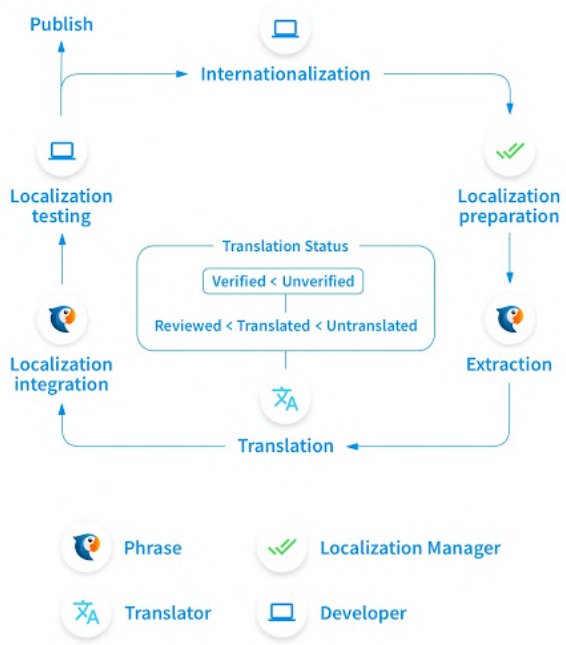
- Tracking the history of translation terms
- Discuss localization through comments
- Manage glossary of business domain-related terms that could be uncertain for translators
- Tagging terms
- Importing keys and values from l10n files
- Exporting translations to l10n files
- Versioning translations (like Git in the development world)



Another important thing is to remember that translation workflow behaves differently than the development process; therefore, we need to have other relevant tools.

We have business people who make initial localization terms in a large project. Then it goes to translators (probably more than one) that also send some translations to native speakers to consult and confirm. After that, localizations can be accepted by translators or changed again.

That means we need a way to verify some already translated terms. It's hard to visualize this process, so let's take a look at the Phrase workflow diagram:



As we can see, localization has a specifically designed workflow. TMS tools are crucial to have in your project (even if it's not large) because using them is simple, and many l10n-specific things are being taken care of for us.

Last but not least, check the trial version of your TMS tool first. It's crucial because many TMS tools have different features and handle some specific file formats in different ways. Especially with Flutter .arb files that are not so popular in the l10n world, you'd better be sure that the tool you will pay for is compatible with your development.

Automatic UI tests

I think we all can agree that UI tests are great. They can simplify testing tremendously and lessen the number of regressions. They could give you feedback if you broke something right away. They might be hard to maintain, and you probably need a dedicated team just for writing UI tests, but they're still worth it.

There is one problem, though – they're painful to work within the Flutter app. You have all the building blocks, Flutter Driver to test Flutter parts, and you can use Appium for the native parts, which is cumbersome. It gets even worse if you have two separate teams: one writing the app, the other writing the tests.

As some of you probably know, to find a Flutter element in the app, you either have to find it by traversing the widget tree, which is rather complicated and error-prone, or look for it using an attribute. The one used the most is the `key` attribute – you basically provide an id that both teams know.

The only problem is that someone has to add it, and it has to be the developer. And you can't add keys to every element because it will be no different from traversing the tree.

And since the team doing the tests is the one that requests keys (they only know what they need to find), they need to ask the dev team to do it politely. That might work if you have a single team developing the app and a single team developing the tests.

If that's not the case: the testers' team must know who owns this particular element (which might not be so obvious from the outside) and direct the request to the owner. Otherwise, it will surely get lost. You have to have a definitive procedure for requesting keys.

Doing the procedure & pinpointing the owner takes time, although you can't start writing the tests without it. But the best person for the job will be someone from the inside - a dev. And dev time is precious.

And that makes a problem:

- Automatic versioning & changelog generation.
- It takes developers' time,
- Which, if not accounted for, might be considered "wasted",
- So it is not in line with PO's goals,
- Hence they don't want to do it,
- So developers can't provide necessary things for people doing UI tests,
- And they waste their time waiting on blockers.

The only solution is to make the UI tests a normal part of the SCRUM user stories. UI tests should probably be a part of the acceptance criteria for each user story. Only then will UI tests not cannibalize developer time, and everyone will want to write them – not only devs & testers but also business people.

Everyone will have a stake in creating them. Since UI tests will be a part of the user stories and be worked on simultaneously with features, they will change with them because most of the time, the changes in-app and in tests would be symmetric), which will ensure that they will not deteriorate.

In the long run, this is the only solution if you have a big, multi-team project where different powers pull in different directions.

Also, let's not forget that UI tests are still tests – and should be treated as such.

It would be best to run UI tests with your standard development workflow. I agree that they might take too long to run, and it might not be feasible to run them on every build, but at least a minimal viable subset should be run that way. And everything else should be run periodically (a couple of times a day at least).

Contracts

We all know that somewhere there is a regular JSON (or XML)-based API. Someone will probably call it a REST API, REST-ful, REST-ish, or just "an" API. Or GraphQL one. That does not matter. No matter what API style you use, it still requires a non-negligible amount of work.

First and foremost, you must manually manage it. You need to design every endpoint deliberately.

You need to consider every aspect of the API: will it be a problem to compose a request? Do I require some other request to be done before this one? Or is a request done after this one? How do we ensure that the same data is passed to a couple of different requests? Or how can we tell that the data format will be the same across a single "area" of the API?

Although it seems simple, it's not that simple; it's not easy to express that in a structured way.

Yes, we have OpenAPI that allows us to express that, and yes, although manually, we will probably be using some tooling to manage everything. But that won't be fully automatic. And there will be some OpenAPI-code impedance mismatch.

Especially that you will either have both backend and frontend contracts generated out of the OpenAPI schema, and both generators will work slightly differently.

And this mismatch is not the only thing that will cause problems. The clients will work the way the tool wants them to, which might not make sense to you.

You will have to tweak them and then maintain them manually. And if you use some exotic feature or design something slightly different than the tool authors assumed, your code will break. Sometimes very subtly.

And this will push you towards manually writing a client for this single request, and then another, and then another, and you will have a set of manually constructed requests that are unmaintainable, probably broken, or altogether wrong.

Because of these, contract testing is a must – API breakage will be too common to ignore. And finding what broke will be very difficult without a robust set of tests. Or when the backend allows for some ambiguity.

So, our solution to that is "strongly typed contracts." A concept that is now widely used in the banking app and at LeanCode.

The idea is simple: since backend devs are the ones that serve the requests, and they have the most "business" knowledge when it comes to how the things should move underneath, let them write everything – both the request schema, the request handlers and clients for these requests.

But instead of using OpenAPI to design that, let's use a language native to the backend that allows us to express everything I was talking about earlier. The backend language is probably some general-purpose one, so parsing it & generating a client that is based on it will probably be a not-so-hard task if we limit the feature set to only the vital things.

This approach has several benefits:

1. A single source of truth – the backend team dictates how everything should look like. Of course, they do so after agreeing with the mobile team on what it should be. ;)
2. They know the flows and the data they need to manage, so they can preserve the meaning and communicate that to clients using the same language. Plus, a sprinkle of in-code documentation.
3. Since they write regular, readily usable code, they don't waste time doing documentation (that will, in the end, be thrown away because the requirements change).
4. Since everything is code and code that is generated, there isn't a place for technical omissions. Basic types are the same everywhere, so if contracts want an `int,` you will not be able to put a `String` there. If everything compiles (both on the backend and client-side), there is a high chance that it is correct.
5. Since the client is a regular Dart (/JS) code, its discoverability is the same as the rest of the project - you open "the API" in VSCode or your other favorite editor. You control how the client works, so you can make it readable without sacrificing functionality.
6. The contracts are also easily versionable regarding schema – you use git for that and point to a particular commit/tag to use a specific version. Do not confuse this with API versioning. It would be best if you still did that.

What's best is that it's Dart all the way, and every mobile developer is/needs to be comfortable with it. Of course, it's not a silver bullet.

It does not solve every problem possible and introduces several issues on its own. You have a strongly-typed client, so if something changes in the types, you need to account for that when updating the contracts.
And it won't matter that it changed far away, in a request you don't use.

They don't make the API versioning & supporting easier. You still need to do that, although particular versioning requests are now slightly easier. You can also enforce that the API is backward compatible with proper linting.

**And since you operate on high-level types and you have a limited number of types there (because, after all, it is JSON), you can't express everything.
And everything is typed, so there is less room for ambiguity, and sometimes you need to make things awkward.**

Although there are problems, some of them minor, some of them amplified, all in all, it is a gigantic net plus that simplifies the development and understanding of the project.

Taming legacy

When we develop a large-scale project, we often make decisions that become outdated over time. Also, those decisions will have to be made in the future, so it's inevitable. Our code is legacy from the moment we wrote it.

If we wrote it again, there would probably be other things that we'd consider. So it's okay. We have to accept it. We can't remove it, but we can tame it. What can we do as developers?

First, we can deprecate things. If some function, widget, class, or whole module is being used around the code, a new way of doing things must be introduced since business assumptions change, then deprecate. In Dart (like other languages), there is an annotation @deprecated that comes in handy. It's helpful because when 20+ developers are working on the project, they must be aware that this method or class shouldn't be used and what is the correct way to prevent that.

However, make sure you deprecate. The worst thing about deprecation is not obeying it. It introduces even more corruption to our code and communication since both ways of doing one thing are still being used, and new developers don't know what to choose. Also, we have to remember about broken windows in our code.

A broken window in the code means a code smell or something that should obviously be done any other way, but there probably was no time for that or any other unpredictable reason. That code will be emerging as more and more corrupted because people wouldn't bother to refactor things.

That's why we want to be clear about deprecation. Because the objective is to encourage refactoring, we want to minimize concepts that discourage it – such as broken windows.

To ensure policies in our large organization, we must maintain a technical squad because there must still be an owner to secure things. The technical squad owns everything that's not related to any business domain – CI/CD, tooling, cross-squad big picture things, and one of those responsibilities is taming legacy.

That squad has to organize cycle status events where we can monitor the progress of refactoring deprecated code. That's why it's essential to synchronize across squads weekly or so. We don't want the inertia to grow silently as we come to the release. If we control it, it's not so much of inertia.

Design System

Last but not least, from our experience, we should apply similar constraints when it comes to design. The design of our app means UI from the code side, and to make our user experience top-notch, we have to be consistent. What does it mean?

It means that good app design should have consistent behaviors, a consistent color palette, consistent approaches to similar user stories, and so on. Moreover, this should not come from the code because the UX/UI and design don't come from us – developers.

We merely implement the thoughts and concepts of designers that we work with. That's why we have to maintain a design system and do it in tight collaboration with designers. Developers should always use design system components, and one squad should be responsible for the design system itself.

In our organization, the Overall Design Squad is a kind of a "master squad" for all designers working on the project. It also has developers that, while continuously interacting with UX/UI people, are developing design system concepts in the codebase.

Project Owner's perspective on Flutter

By Marcin Olech, Product Owner at Tribe Seamless in the 07 Payments squad in the "CA24 Mobile" project.

I'm a Product Owner at Tribe Seamless in the 07 Payments squad, responsible for our mobile app's payments area. These include various types of transfers as well as functionalities relating to the BLIK service. As for a banking app, the convenience of use and reliability and security of these processes are of key importance.

I previously worked on a project for our previous mobile app, which was a native app: initially, it was available on Android and iOS, and then additionally on Huawei. While native apps obviously have their advantages, the increased labor intensity related to each functionality was the complication.

Business analysis, testing, and then maintenance and development of the app required a tripling of efforts each time due to the specifics of each of the systems. Moreover, costs were also significantly higher for this reason.

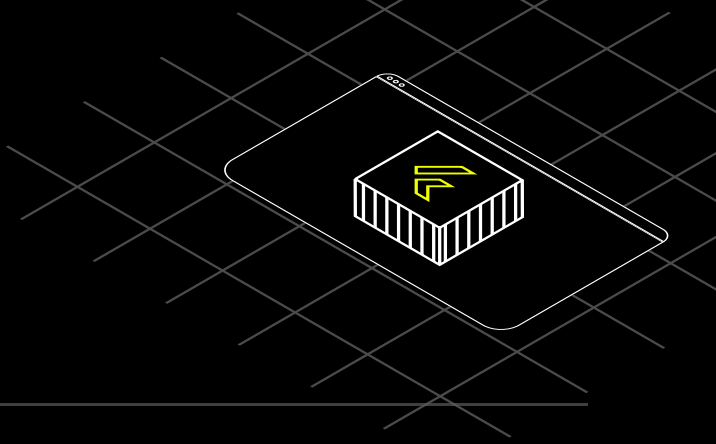
As part of our work on the new mobile app, we decided to choose Flutter as the technology we would use. This was my first contact with this framework, and of course, I initially had some concerns knowing that it was "something new." While working with the developers and seeing the application running even in its early stages, my concerns quickly disappeared.

I already know that the community is also growing faster and faster. Flutter supports all our business concepts and eliminates the problems related to the native apps I wrote about. We tested the app on different systems and devices, but it worked identically in most cases.

This made our work much easier and resulted in building the app available quickly from the start of the project. In the case of functionalities that may work differently on different operating systems, such as PUSHes, for example, the developers always pointed this out to us to pay special attention to it.

In conclusion - I consider the choice of Flutter as the framework for our application to be a brave but very good decision by the bank.

Project Owner's perspective on Flutter



By Katarzyna Skiba, Product Owner at Tribe Seamless in the 05 Accounts squad in the "CA24 Mobile" project.

In the project, I have been acting as a Product Owner of one of the squads from the beginning. My squad provided a very important part of the new application "CA24 Mobile – full of benefits":

1. A large part of the dashboard, including its key element, the so-called accordion, on which we present the products that the customer uses most often along with offers in which they may be interested, in an innovative way.
2. Access to the customer's most important daily banking products (accounts, foreign currency accounts, account cards, credit cards, savings).
3. Possibility of performing active operations on the above-mentioned products (card activation, change of card PIN, temporary card blocking, opening of a foreign currency account, opening of savings products, and many more).

The main challenges I faced were managing a large backlog and a large team, including developers. Moreover, I handled these challenges in a working methodology previously not practiced at the Credit Agricole Bank, i.e., SCRUM.

We had to (not only me but the whole team) learn to work in sprints, where the pace is very fast, and these 2-week periods pass in a flash, one after another. But it was worth the effort and gave us a huge satisfaction when after these two weeks, we could present the result of your work and boast about it in the Review.

In my opinion, it has also been possible due to Flutter, which simply makes coding functionalities easier to implement. You don't have to create the code for each operating system separately. You don't have to test the functionalities on each system separately because, in most cases, they work the same.

What other benefits can I see from using Flutter in our app?

- It's easier and cheaper to maintain the app.
- The ability to react quickly to changes in the agile way of working is enormous.
- The possibility to create outstanding animations that make our application stand out, including our avatar or a river full of benefits, is excellent. Although, we faced some of Flutter's limitations regarding animations. It turned out that on some of the users' older devices, some animations did not work to our requirements and had to be modified.
- Programming in Flutter also made managing and delegating tasks between developers easier. I didn't have to think about who was doing a particular functionality on iOS and who was on Android. And each developer could take over a colleague's task if necessary.
- In my opinion, it is also a relatively easy language to learn, broadening one's skills. In the project, I met at least two developers (including one in my team) who started as typically backend developers, and in the course of the project, they also learned programming in Flutter.

To conclude, I want to say that it is amazing that in one year, we have developed and made available to our customers a completely new application, rich in many functionalities that were missing in our previous app. This was also possible because of Flutter technology.

Introduction to the Design System

By Albert Wolszon, A Flutter Developer at LeanCode, responsible for the UI development of the "CA24 Mobile" app.

The design team should have already prepared the foundations of their design system at this point. Most primitive of which is a style guide that contains the very-very basic elements such as:

- fonts used throughout the application,
- brand colors and styles for specific content,
- titles, captions or hints,
- and many other concepts that are defined at the very beginning and then reused as a variable – rather than hardcoded – in all components.

The design team – from the Efigence company – used Figma for the CA24 Mobile app designs.

Those basics are reused in Figma as well as in Flutter widgets. Deciding on how to define those styles must be well-thought-out. Well, all those 30 Flutter developers will use it later, right?

Colors. There are around 40 colors in our style guide. All those colors come from Figma. They are validated in terms of contrast and brand compliance.

We created two structures that help use our defined colors. A `CAColor` class, which is a child class of Flutter's `Color` class. The only difference is that it has its constructor private so that only the `cabp_common_ui` can instantiate new colors.

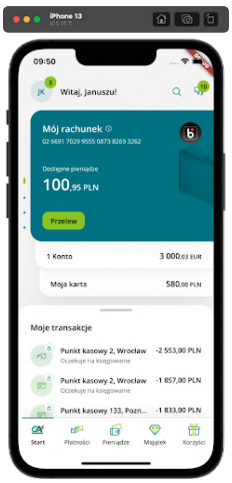
```
class CAColor extends Color {
 const CAColor._(int value) : super(value);

 CAColor._color(Color color) : super(color.value);
}
```

The second structure is our color palette.
A `CAColors` class holds all instances of `CAColor` and is used by all other widgets or screens. It should be an inherited widget, so widgets that use colors from the palette get rebuilt when it changes.
For that, we also have a `lerp` static method, which helps when changing themes.

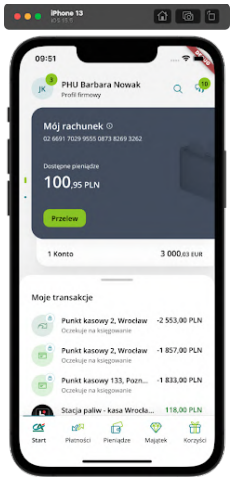
It is very handy in our case.

The CA24 Mobile application has four themes: light & dark theme Cartesian multiplied with Detal & SOHO themes (which are fancy names for retail and business account themes).

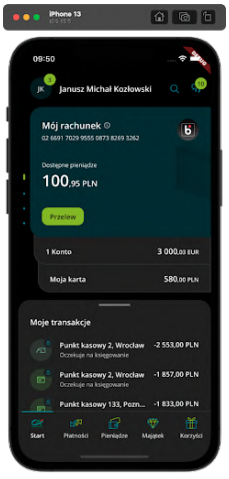
Detal light



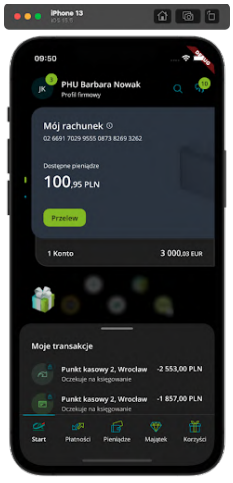
Soho Light



Detal dark



Soho dark



The second most important basic for us was text style. We also used a custom class for that purpose, making the constructor private.

```dart
abstract class CATextStyles {
 static const _parent = TextStyle(
   fontFamily: 'Open Sans',
   package: cabpCommonUiPackage,
   fontFeatures: [
     FontFeature.tabularFigures(),
   ],
 );

 // General use

 static final headline1 = CATextStyle._style(
   _parent.copyWith(
     debugLabel: 'Headline1',
     fontWeight: FontWeight.w600,
     fontSize: 24,
     height: 36 / 24,
   ),
 );

 // ...
}
```

What we found is that keeping text style and its color separate makes things much, much more manageable. The color usually isn't strictly related to the text style, and it's more convenient to have `style` and `color` parameters in `CAText` instead of using `copyWith` on style or other wild constructs. It has also enabled us only to accept our constrained types for text and other widgets:

```dart
class CAText extends StatefulWidget {
 CAText(
   String data, {
   Key? key,
   this.style,
   this.color,
   this.textAlign,
   this.maxLines,
   this.semanticsLabel,
   this.scalingStrategy,
 })  : data = [CATextSpan(data)],
       assert(maxLines == null || maxLines > 0),
       super(key: key);


 final CATextStyle? style;
 final CAColor? color;
 // ...
}
```

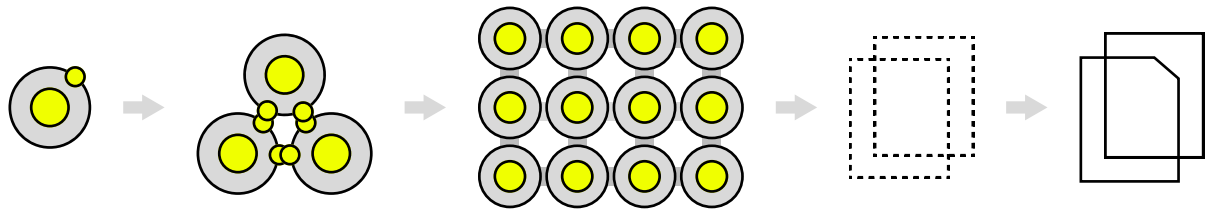
Box shadows (which we call elevations), curves, durations, and map styles are solved similarly.

Atomic Design

Once we have the basics set up, it's time to create our first widget! Flutter already gives us a collection of many customizable widgets.

From the most trivial ones like `SizedBox,` `Container` or `Text` to Material's `ContainedButton,` `AlertDialog` or `Scaffold.`

Those widgets are put into another. Small parts are reused in more extensive pieces of UI. Brad Frost formalized the process of placing UI pieces together in a methodology better known as Atomic Design.

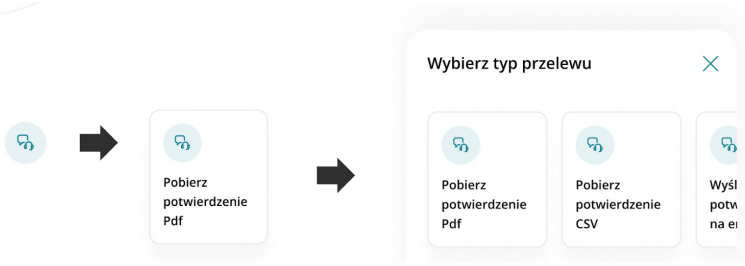


It's a methodology of designing oriented around components, not whole screens. It defines how the minor independent parts – atoms – are built and reused in other places, such as molecules. It can be a simple thing like an icon that we always put on a small circular background.

A molecule is a group of UI elements that together can deliver some data to the user and simply add the purpose to a screen fragment. We have a card where we put a title next to that icon and add some border and box-shadow around it.

Organisms group molecules together, creating fully-meaningful parts of the user interface. If you see an organism somewhere, it will be entirely understandable and function as a container with some context that you can safely put next to other "contexts," like a carousel of mentioned cards with a heading and a close icon on a bottom sheet.

If we put it on top of anything, it will still be visually understandable to the user.



Templates help glue everything together. Pages are simply... screens.

I highly recommend reading more about the methodology in "Atomic Design" by Brad Frost, a free online ebook.

The Overall Design squad was developing mainly atoms and molecules, but there were also some complex organisms and templates.

Ambiguity

One of the biggest pain points was the frustration when different people had different ideas on how something should work. Naturally, developers came to designers with questions regarding components they develop.

Be it how something should behave in certain conditions, what should be clickable in this and that edge case, and how those parts should animate on the screen.

We usually called each other for ad-hoc calls and discussed that. The problems arose when some time had passed, and others started asking questions about why something works that way and not the other.

Those small decisions led to scratching our heads, searching through conversations with designers on Teams, or revisiting Git history for that component to remind ourselves why we introduced such change in the first place.

At some point, it became a frustration that needed to be addressed ASAP. From then on, we always update the specifications on Figma or documentation in code straight away so that all knowledge and changes with their motives persist.

On a higher level, this is known as Architecture Decision Log.

Future-proofing

During the development of this common UI library, there were a few situations where we were worried that we would need to rewrite many components because of some changes.

That was the case when we were introducing drastic changes to how the increased text scale accessibility feature was influencing our components or when there were other changes to how everything should animate with the finger gestures on our custom scaffolds.

You can't see some of those changes, but preparing for some of them is possible.

When we were first developing more complex organisms, there was no motion design prepared at that point, unfortunately. If we completely ignored the transitions when changing the currently selected tab in the tab bar or carousel physics, adding them later would be a refactor.

It would cost us not only the time needed for introducing changes in the widget but also the time wasted in migrating all of the code that already used it.

The same principle applies when introducing new variants to components. Let's say you have a card that, until now, had only one style. The designer prepared another state for this card.

Let's say that's a card that describes a debit bank card. The new state is for a case when a bank blocks the card. You can add a boolean `blocked` parameter to that card or introduce an enum that describes its states, like `normal,` `blocked,` and probably some other like `shipped` or `expired` in the future.

The dark mode was one of the things we knew we'd be implementing at some point in the future, but we didn't have anything close to its specification. If we did not create this construct of color palettes mentioned at the beginning of the article, we would spend a horrendous amount of time replacing all colors with their dark mode counterparts. It paid itself back doubly when we had to introduce another dimension of color themes (retail/business account).

The most recent challenge we encountered was fixing the application's appearance with an increased text scale. You'd be surprised how many people reported issues related to that.

Thankfully, all text content, labels, and paragraphs in the app used our custom `CAText` widget. So we could address the vast majority of those issues by remapping how the Flutter's text scale factor influenced font size and its line-height, as we agreed with the design team.

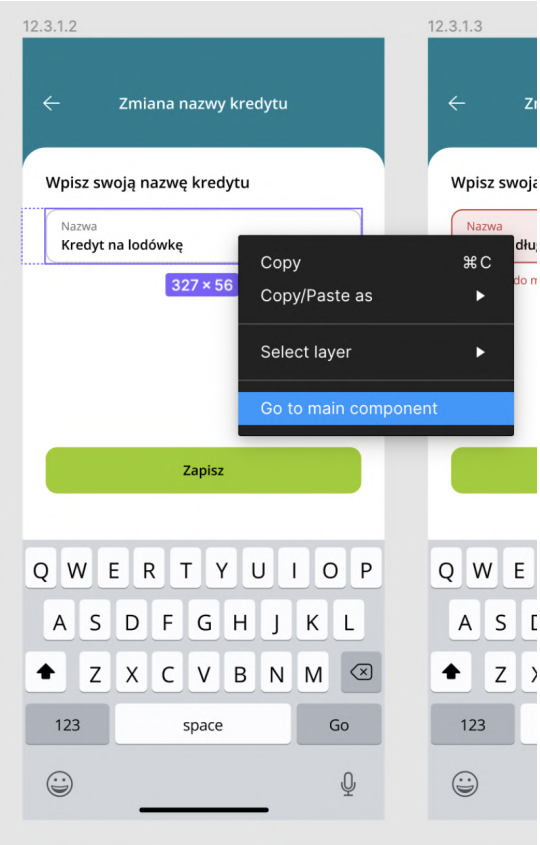
Think ahead. It saves time and helps you stay sane.

Navigation

Other developers develop screens using our UI components, based on Figma designs.

Figma allows navigating to component definitions from their instances conveniently. There, developers are welcomed with all available variants for a component, its specifications, and its Flutter widget name, which they can use straight away in their Dart code.

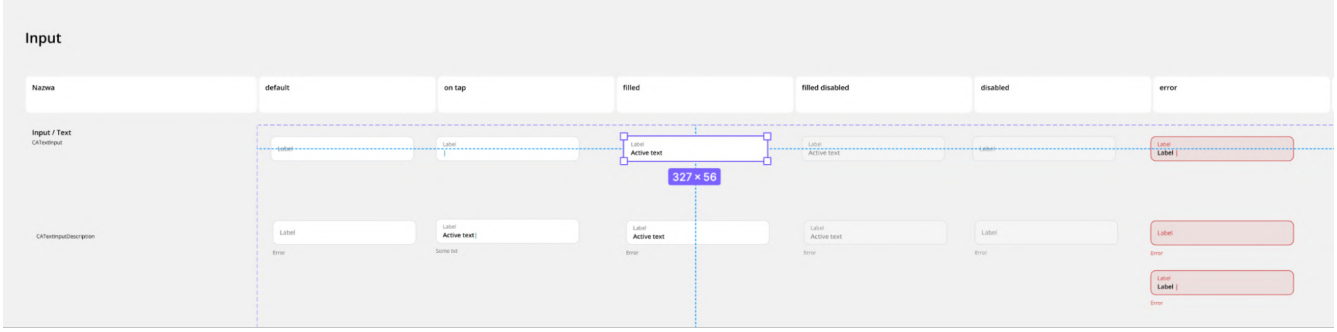
Or, in other cases – an enum value, a named constructor, a few class names, or a widget's name with some parameter value. Whatever a developer needs to use the component.



Developers from the Overall Design squad sometimes need to investigate how and why a component is used in a certain way. It generally comes down to finding the screen's source code, using Flutter Inspector, or searching for localized strings on a screen.

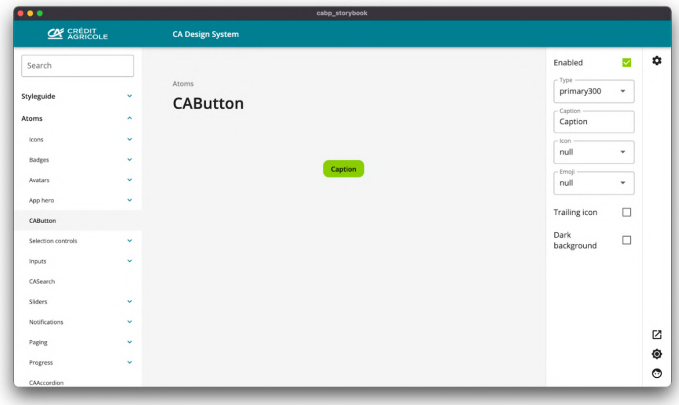
Ideally, all screens have a one-line comment on top of their class that describes where this screen could be found in Figma.

We have so many screens and designs that we have over a dozen Figma files for different squads with all screens numbered, so it's easy to give a file's name and the screen number to find it. It saves you the hassle of asking the developer or designer for directions when you can't find the design yourself.



Storybook

If we were to name one thing that improved our productivity in the project the most, we would scream storybook without a second of thought.



- Everyone else. Product Owners – they need to know whether a component is already developed or not to plan their squad's sprint more effectively. Social media people – they can set up and screenshot a card component with the travel credit information for their vacation marketing campaign.

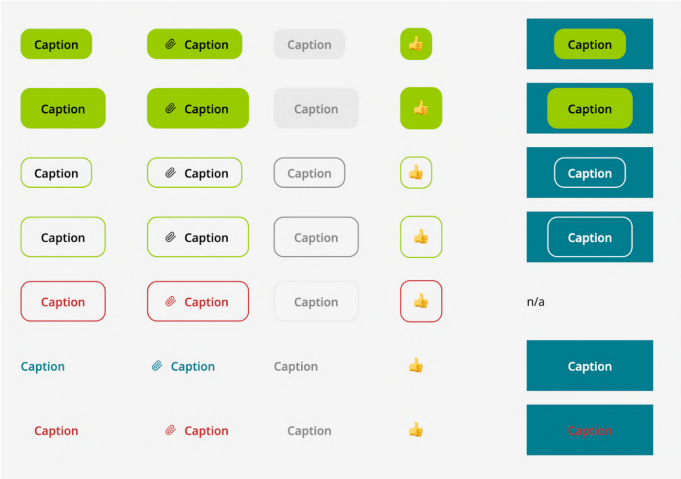
It serves:

- The Overall Design squad of developers as our magic workplace. We develop widgets in isolation and check all edge cases, strings of different lengths, states, and parameter toggle straight away without needing to sign in to the application. We don't have to wait for the emulator/simulator to boot or click through complex processes to reach this specific widget in a chosen state. It's also much quicker to compile and launch the desktop storybook application than wait for the mobile one. It's just faster.

- Other developers as a place to explore widgets before putting them inside a screen, checking if it suffices their needs even without writing code and hot reloading the app.

- Hosted on the Web as a place to display increments on Sprint Reviews, where we can open each new or modified component in a new tab and simply go through all of them prepared.

- As an environment to validate the components by the designers if everything looks and works correctly, as they wanted.



All those buttons are easily accessible through the storybook via its knobs. Do you want to see how a button looks in different sizes? Different type? With a leading icon? With a trailing one? On dark background? Sure thing! Adjust the knobs in the sidebar, and you're good to go!

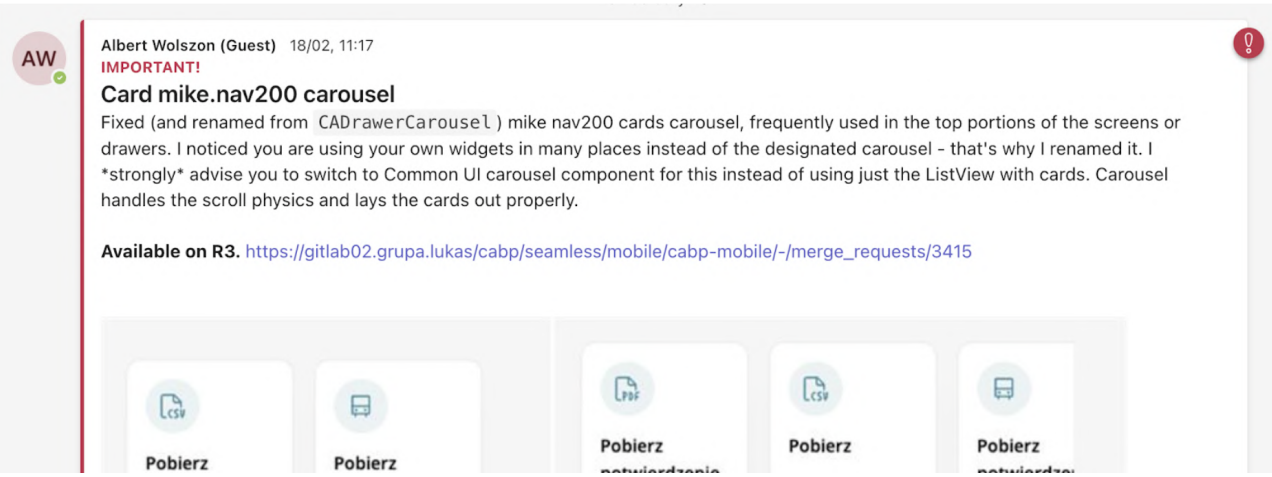
Communication

Being the code owner of a package used by everyone else makes you hold and create knowledge that needs to be shared with other developers.

When it is information that is crucial at the moment – an announcement of a new component that some squads were awaiting or a deprecation notice with some hints on what to replace the component with – we were using a simple means of a message on our dedicated Teams channel for Common UI announcements. It was marked as important, so everyone got this annoying push notification and wouldn't miss the news.

Apart from the developer-developer communication, as the Overall Design's developers, we also had the developer-designer one. There were three significant learnings for us:

- **Formalizing each decision, as mentioned in Ambiguity.**



We used Markdown documents in the repository for higher-level documentation and other communications that should stay in some place and be accessed when necessary. It is where we store steps necessary for newly onboarded developers or people who need to use some part of the framework or some shared functionality for the first time.

It is a place with help not specifically on how something works but what component or solution to choose and why when designing screens and business logic.

- **Process of receiving updates from designers.**

A vast part of our daily work was developing components our designers have designed. Or we were updating the already created ones. But we need to be notified of those components that have changed. It is something that would usually be resolved simply by using the JIRA board, where designers drop a tile concerning their component onto the next column on a board where we notice it and begin to work on it. For security reasons, designers could not access the JIRA back then, so we made our kanban board in Figma.

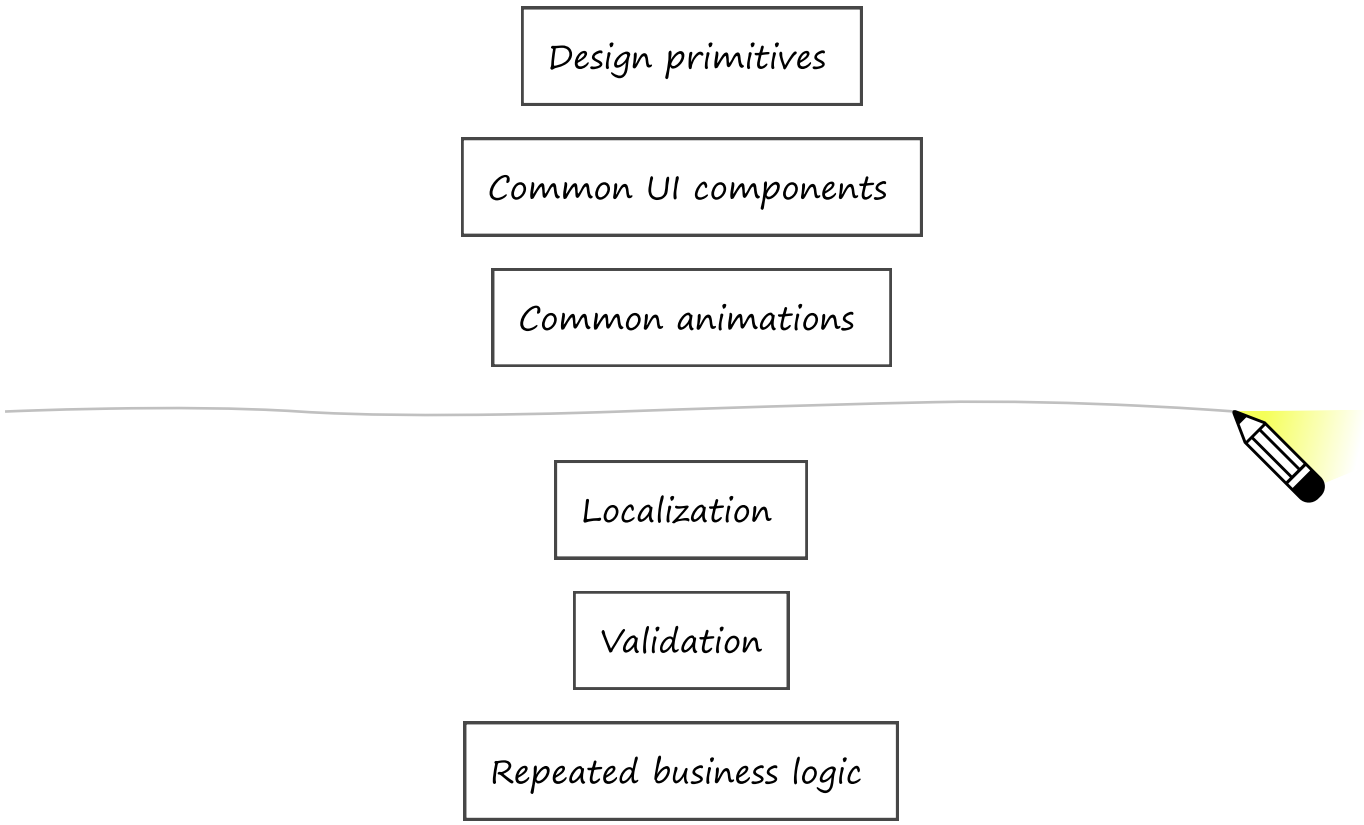
- **Components' changelog would be very handy.**

Sadly Figma doesn't support something like that, and we didn't maintain such changelogs by hand. Figma does indeed have file revisions history, but it's the specific component's changes that we usually wanted to check, not the whole file.

Responsibility

What is the responsibility of the Overall Design squad? It needs to be decided. Clear code ownership is crucial for maintaining such a big codebase. Otherwise, the code without clear ownership will be lost.

Here are some of the things you may wish to be responsible for as a squad or not:



You need to decide where you're going to draw a line.

Technical challenges

During the development, we stumbled upon many challenges, both small and big. The first significant decision involved choosing the correct tool for a storybook. As you already know, the outcome was relatively successful.

When we had this dilemma, there were only two open-source players on pub available – `storybook_flutter` and `dashbook.` After doing some research and creating an MVP, we decided to go with `storybook_flutter.` Later, we refactored it to accommodate our requirements, such as tree structure for the components, two theme dimensions, and Credit Agricole branding.

At the moment of writing, two more solutions are available: `widgetbook` and `flutterbook`, both of which seem promising.

The application's biggest and most complex component is the root bottom drawer located on the start page, below the accordion and benefits river, connected to the screen it shows when expanding. When you play with the application, you can see how the drag gesture slowly reveals the second screen, which is a separate route.

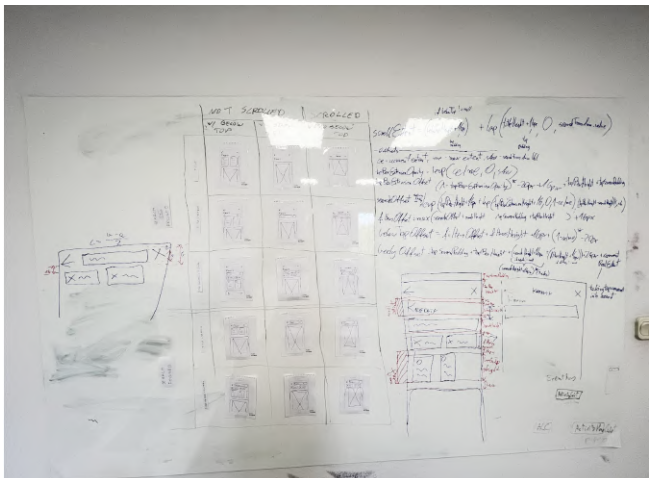
Apart from that, on this second screen, you can scroll the page, typical behavior, with the header elements folding themselves with the paging zip on the side. When you focus on the search bar, it also animates itself to the app bar. And if you happen to be on top of the page, you may drag the body down to collapse the drawer from the start page.

Trivial stuff, eh? No. The first iteration of this artifact took 2 weeks, heavy research, and deep-diving into the internals of scrolling, gestures, and routes.

Looking inside `DraggableScrollableSheet` was very helpful, as it did more or less the same things, just a magnitude of complexity lesser. But after the design team reiterated and reiterated more on how this screen should behave, the following updates were harder and harder to introduce until, at some point.

We finally couldn't introduce the change as the code was such an incomprehensible mess.

It was a time for a clean and thought-out refactor. We started by drawing all available states on paper and laying them out on the floor to find how each state relates to the others. Then we knew what we were dealing with. We could start with looking for the correct math for concrete translations and applying them.



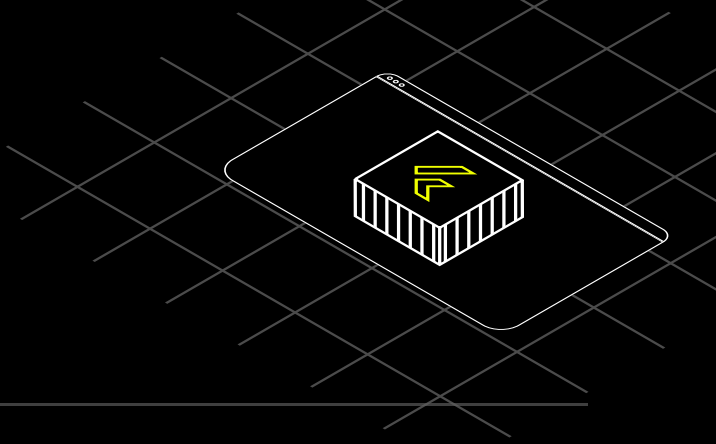
The code behind it was so clean it shined.

On the contrary, we also had some minor technical problems. One of them was the `VisualDensity` in a few of the Material components we used, like `TextField` and `TextButton.` At first, we didn't notice it. Still, the buttons on the mobile were bigger than on the desktop – in the storybook. Once we found the problem, it was a quick override in those widgets, and voila!

This project was a huge opportunity to dive into some Flutter internals.

Some of the other things we dove into were inputs, input decorators, and the backend connecting Flutter with native input logic. Dropdowns, how they work, and how they use `CompositedTransformFollower`s and `Target`s. Scaffolds with all their insets. Routes, how they animate and handle popping, `Overlay.`

UX perspective on Flutter



By Marcin Kinderman, User Experience & Service Design Director at Credit Agricole's Online and Mobile Banking department.

Our job was to create a UX strategy based on our goals and aspirations for the future. It meant that we had to analyze the main ways our product would be used.

Starting with the discovery phase, we have examined all the materials and prerequisites including User Journeys gathered and created by the CX Team. Then we started conducting IDIs (In-depth interviews) with potential customers.

Equipped with this knowledge, we organized UX workshops to extract the main needs and issues our users might have. Such analysis has paved the way to such featues as the "River of benefits" where users get big discounts when doing shopping at well-known stores. A combination of CA partners' discount offers paired with smooth animation of bubbles with icons and logos floating on the dashboard.

During the Envision Phase, we have iterated plenty of ideas. Before the first sketch was drawn, we had to check the constrains of chosen development technology. Fortunately for us, Flutter is a very “design-friendly” framework.

UI-wise, we were able to discover Flutter's potential by closely examining the proof of concept. This has led us to believe the constraints we had to consider wouldn't compromise the app's UX and UI of the new CA24 Mobile.

As soon as our project leaders accepted the concept, we started the design and delivery phase. We could expand our team and enable designers to work within squads. But at the same time, we began creating the first UI frames ready for development.

And that's why we also had to begin working on our Design System, which was coordinated by Squad 01: Overall Design. Initially, this task was hardly trivial because even with the Atomic Design approach, at this stage, we had to decide the final shape of the main Molecules quickly and the Organisms our app was built.

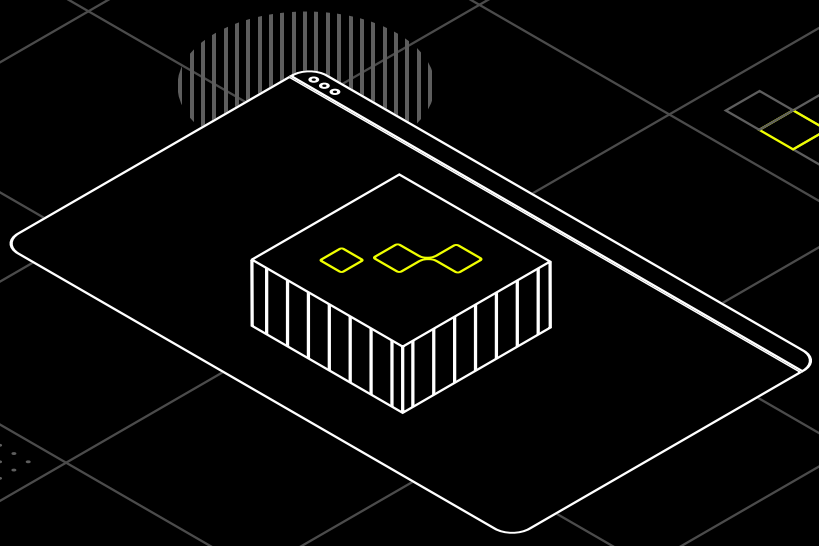
With every sprint, we kept growing our library of fully developed components. That was also one of the most significant benefits or Flutter working across platforms: we were able to have just one instance of each component in our Design System.

Key takeaways:

- Flutter did not limit us in any significant way in terms of design.
- Using a Design System was crucial for consistency and significantly reduced the time needed to work on new functionalities.
- With a project comprising more than 2k frames, we could easily manage a broader scope than our previous mobile app was offering.
- Both designers and developers must have time to support each other during components development and creation.
- Flutter-based Design System allowed us to easily update components on the go after we have discovered different bugs or, at times, lack of consistency in designs.
- Flutter enabled the full potential of animations, including the implementation of “Dito” our app's animated mascot.
- Major UI animations are usually better when a professional animator first creates an example of precise motion and transitions of components.
- Until now, we have conducted nine UX user tests with 90 participants. Each test provides valuable information about user flows and how users interact with individual Design System components.

In the end, User Experience should be "technology-agnostic," as users should not be able to spot the difference in how something is implemented, no matter which platform they use. In this aspect, Flutter saved us a lot of time.

Apart from minor hardware incompatibilities, a spectrum of screens and viewport sizes, we didn't have to work on separate solutions for Android, iOS, or Huawei platforms. That also gave us an edge in delivering new features, placing us ahead of the rest in the banking market.



Wrap up

We're confident that the knowledge we share in this ebook will help you draw your conclusions on using Flutter and give you insights on organizing work in large Flutter teams.

Credit Agricole Bank Polska faced a decision of whether to update its current application or build a completely new one to meet customer expectations and become one of the leaders in mobile banking.

They needed a solution that would allow them to create a coherent application for both platforms. But at the same time, meeting its requirements not only during the hottest phase of project development but also in the longer term.

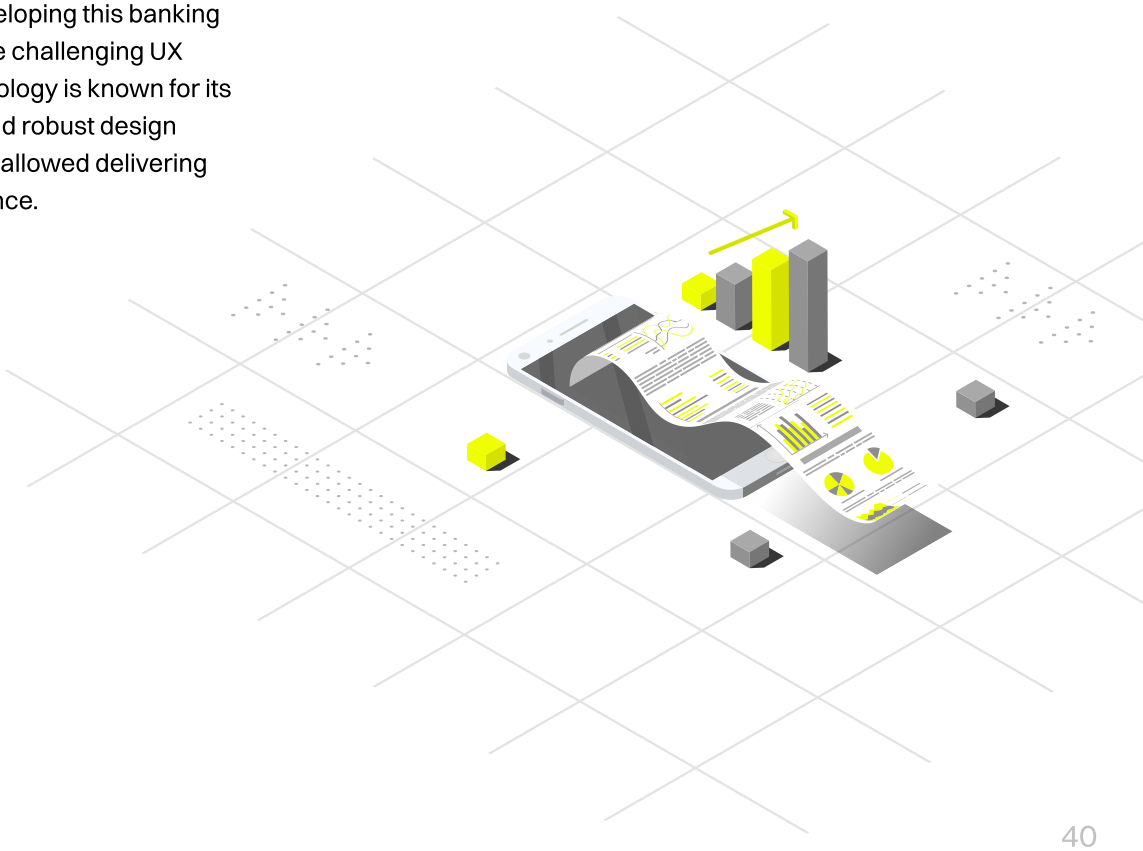
They were aware that the chosen technology had to ensure the quick addition of new functions in the future and ease of maintaining the application. Flutter technology meets all these needs, making it a perfect fit for building the mobile banking application.

Another crucial area of developing this banking application was meeting the challenging UX requirements. Flutter technology is known for its seamless UI/UX features and robust design elements. So, in this case, it allowed delivering a unique customer experience.

In our opinion, this project proves that Flutter technology is ready for large-scale enterprise projects. "CA24 Mobile" app will enable Credit Agricole Bank Polska to advance in Poland's highly competitive banking market thanks to this technology.

A final word of thanks to all the developers and business partners who contributed to this ebook: without your effort, time, and support, it would not have been possible.

The team of 200+ members from different companies, including 20+ Flutter Developers, worked together for a year on this mobile banking project to deliver probably one of the biggest Flutter applications in the world.



About LeanCode

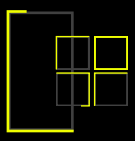
LeanCode is a Software House from Warsaw, Poland, and a leading provider of the native mobile applications built with the Flutter Framework.

We have a team of 60+ developers, designers, product owners, scrum masters, and QA engineers who support the development of mobile and web applications using Flutter, .NET, React, and other technologies.

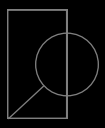
The majority of our clients represent the Banking and Fintech industry, but we also develop products for Marketplaces, Logistics companies, SportTech, MedTech startups, and others.

We work with clients from all over the world, including the USA, UK, Germany, and Australia.

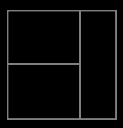
Our services include:



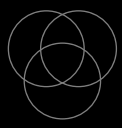
Mobile App Development



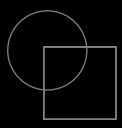
Mobile Apps Audit



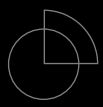
Web development



DevOps Service



Product Design



IT Consulting

You can find out more about our core technologies, services, and delivered applications on our website:

leancode.co

If you have any project in mind, we are always open to discuss your needs and possibilities. The best is to reach us via:

Get an estimate form

See what's new at LeanCode on:

LeanCode