The Back-end for Front-end Pattern (BFF)

Sep 18, 2015

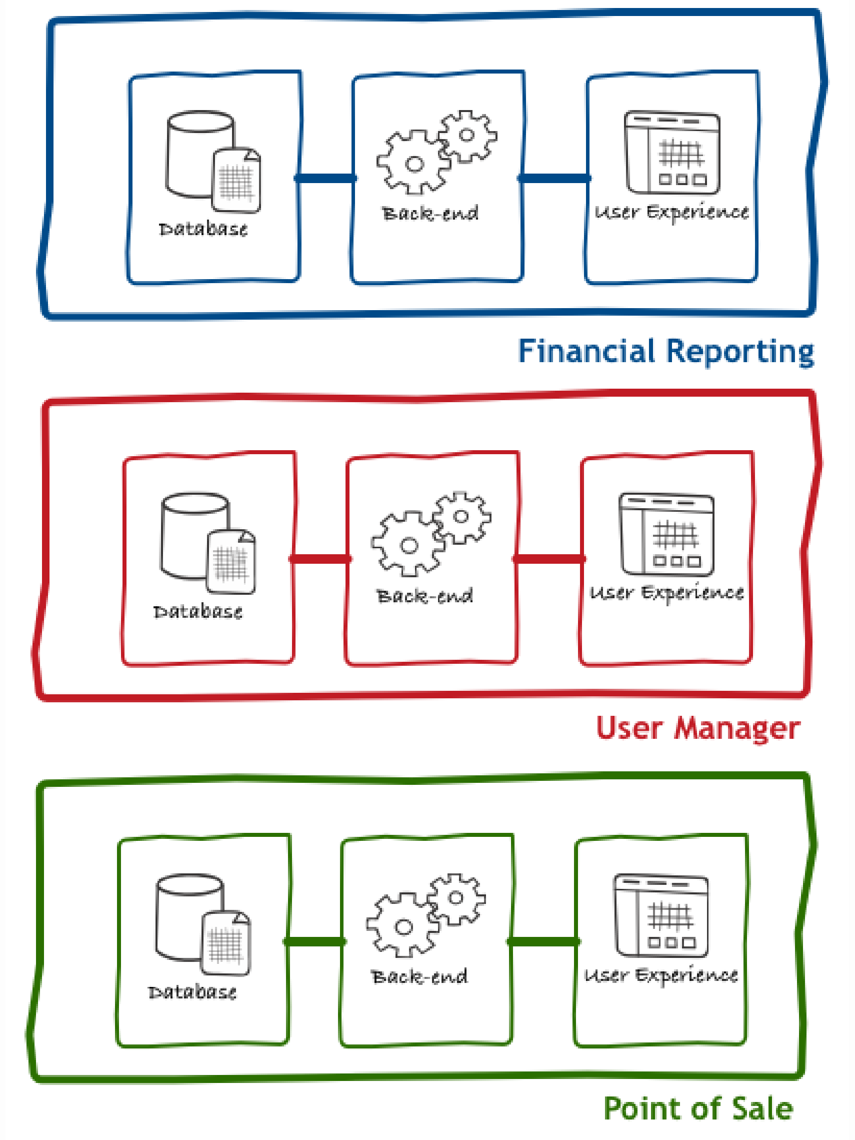
• Microservices • SoundCloud • Front-end • Edge • BFF • Patterns •

When I was at SoundCloud, being transparent about our architecture evolution was part of our technology strategy. Something we've talked about on countless occasions but never really described in detail was our application of the *Back-end for Front-end* architecture pattern, or BFF. This post documents my understanding of how we developed and applied this technique.

My understanding of the evolution of software components

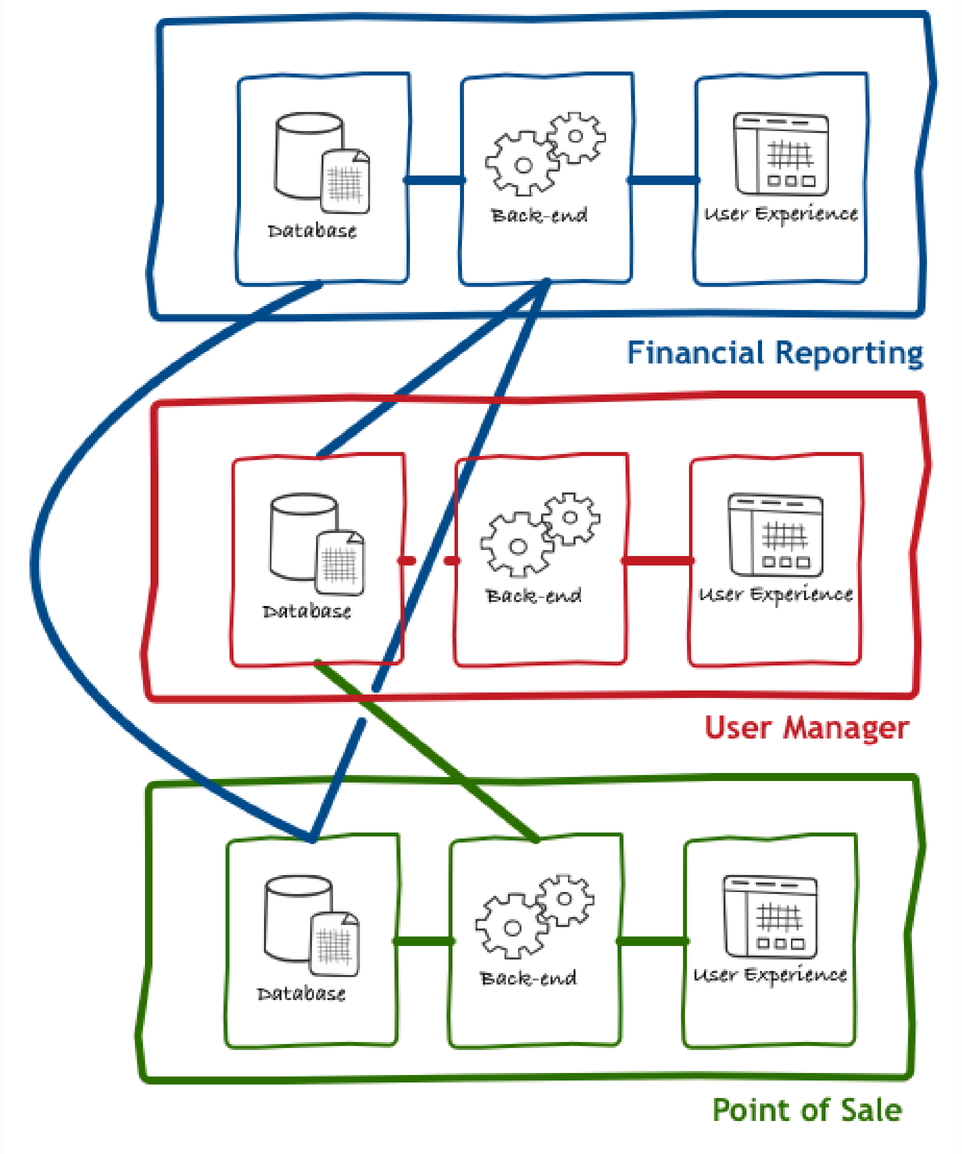
Before fully distributed architectures became feasible, organisations would usually build an application in one or more *tiers*. A tier was a highly-coupled, but fairly independent component of an application. It was *coupled* in the sense that, as opposed to services, it was meant to be used by one application only. It was *independent* in how it didn't run as part of the same process, often not even in the same machine.

Let's illustrate this with three fictional applications that any larger company would develop back then:

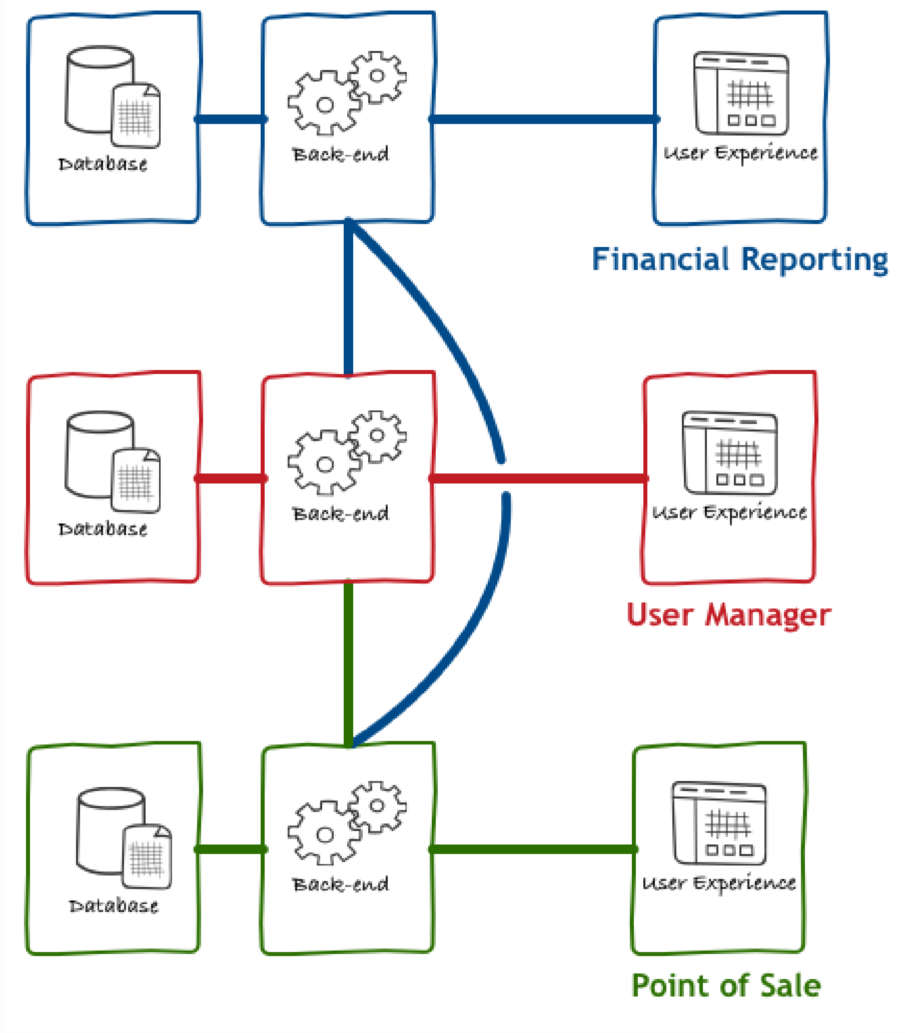


These architectures could get very complicated, but overall it was very easy to draw a line between the different applications, clearly demarcating where one starts and the other ends.

Back then, each application had its own copy of data and duplicated implementation of common business processes. Over time, as organisations acquired or built more and more applications, we realised that we needed something different. We needed applications to share data and reuse logic, and our once simple architecture became a bit more complicated:



With the need for more reuse and consolidation, the collective mindset of the software industry settled on a quite abstract concept called *services*. In practical terms, this means that the diagram above was changed to something akin to this:

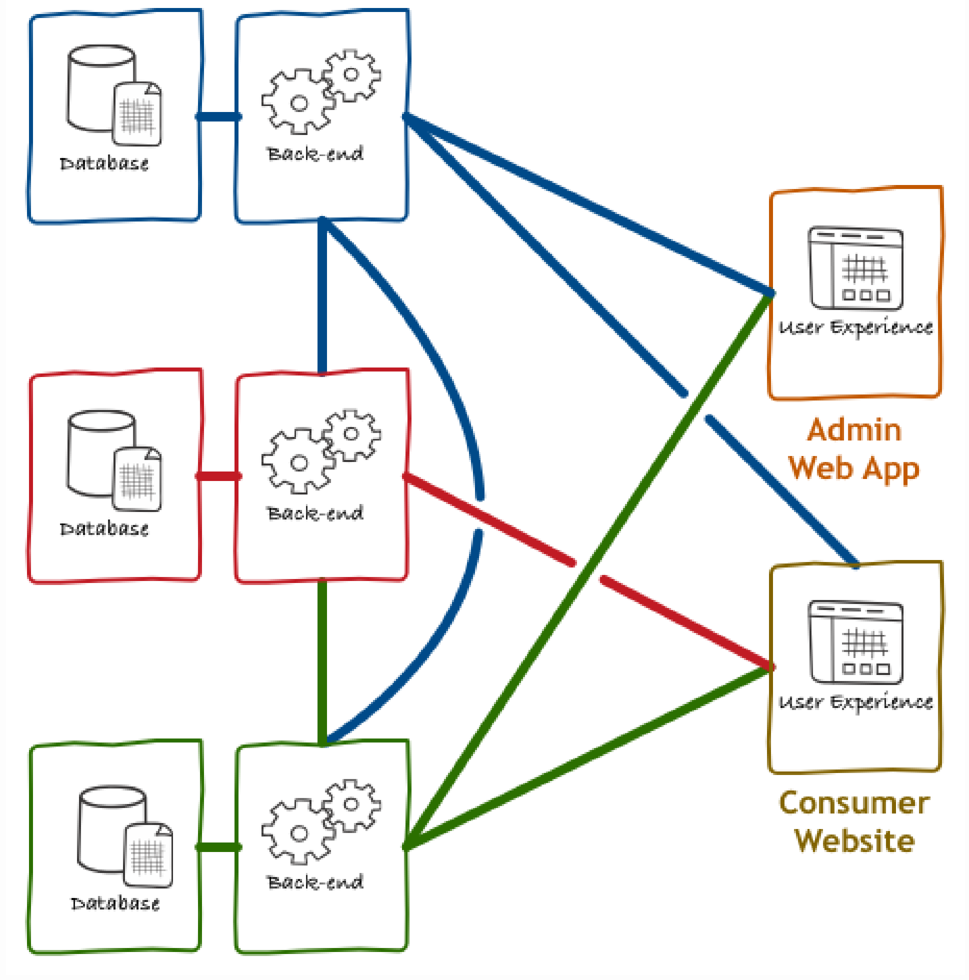


The selling point for the architecture above is the flexibility that those reusable services offer. In theory, building an application on this platform is now a matter of:

1. Selecting which services you'll need
2. Write some glue code that calls these services
3. Merge the data you get back from them into something more familiar to the end-user
4. Renders this data in a way the end-user can consume

At the same time, computers and the Internet were becoming more popular. Customers who used to interact with a clerk or system operator started directly interacting with the applications themselves. Design thinking and user experience research have moved us away from complicated user interfaces focused on making expert users more efficient to richer, more usable experiences that would be understood by customers—nobody reads the manual of a website. Richer experiences require rich data, and this means aggregating information from various sources.

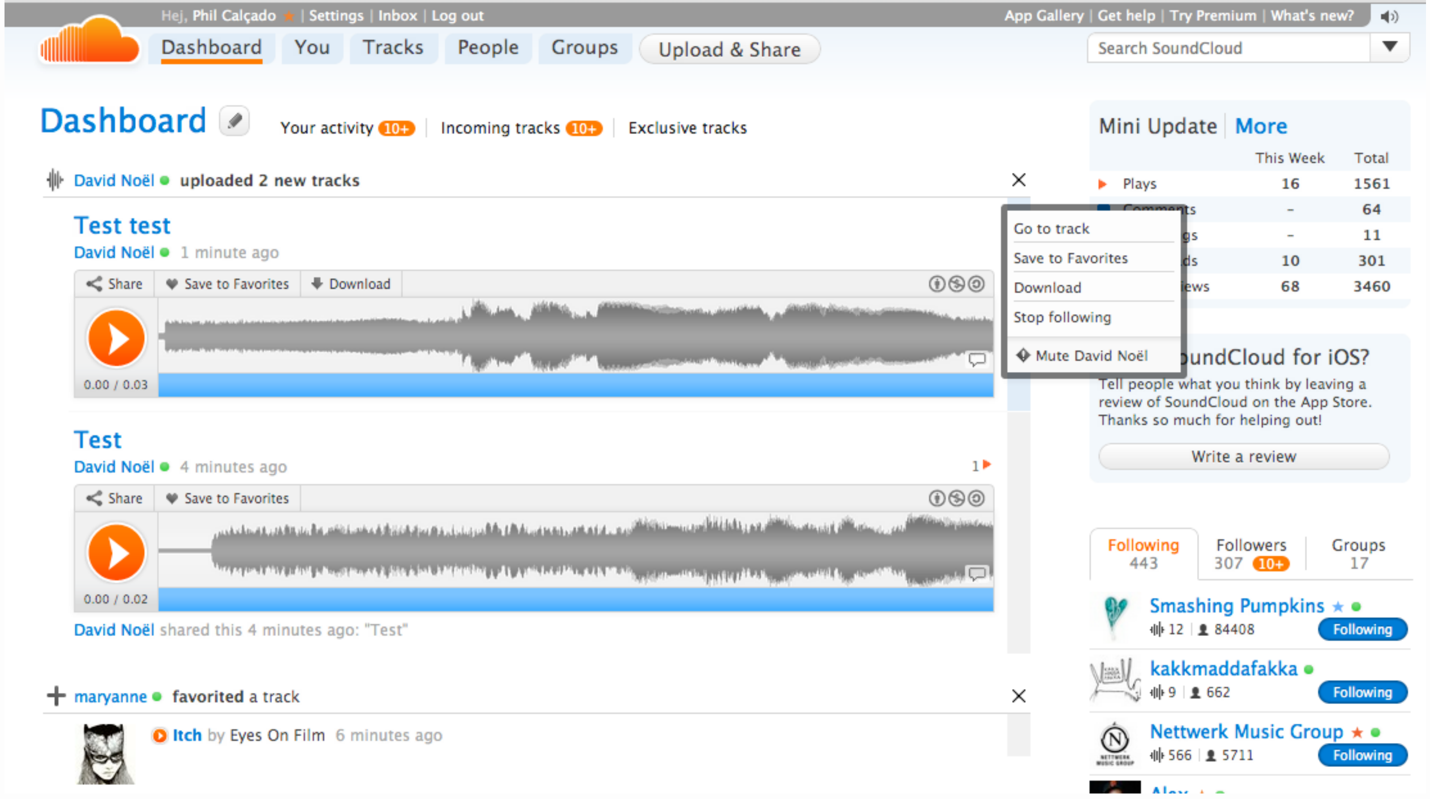
Following up with our example, we end up with something like the diagram below.



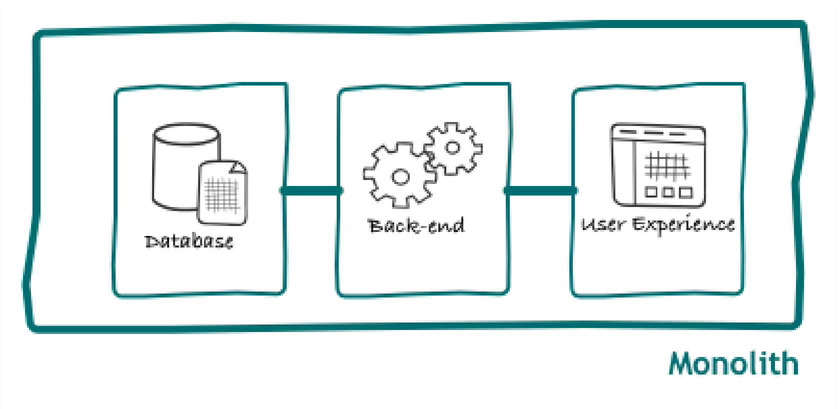
Instead of having what were just user interfaces for *Line-of-Business* systems, more and more we ended up with user interfaces that were applications in their own right. These applications were often written in JSP, PHP, or ASP, and their code contained both the user interface and application-specific back-end logic.

Breaking down monoliths

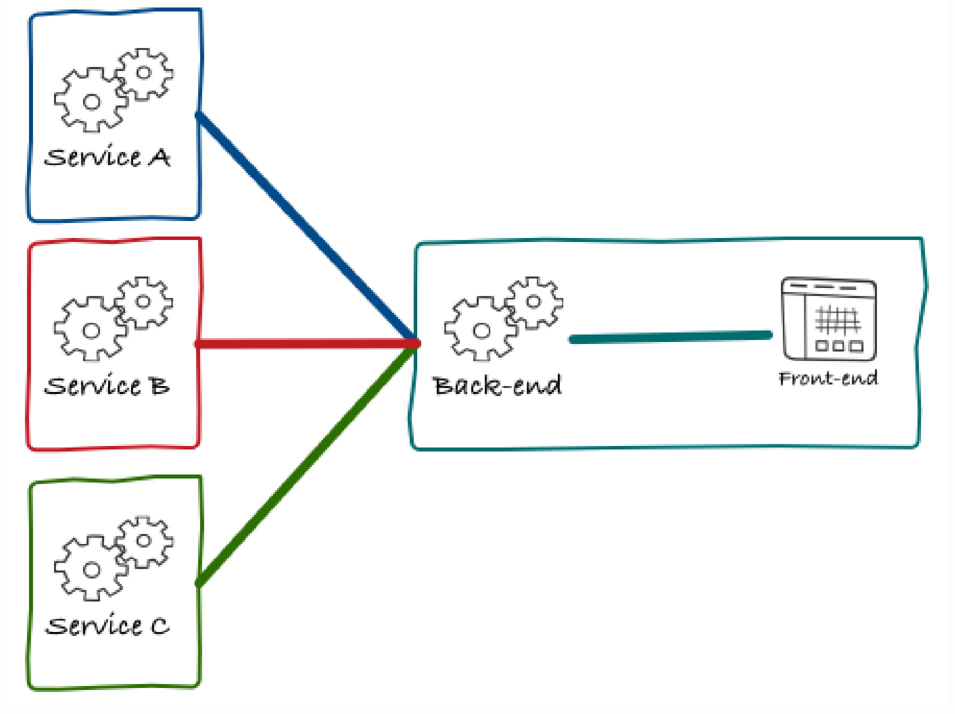
The oversimplified example above isn’t that different from how a lot of modern tech organisations evolved their architectures. In 2011, SoundCloud’s website looked like this:



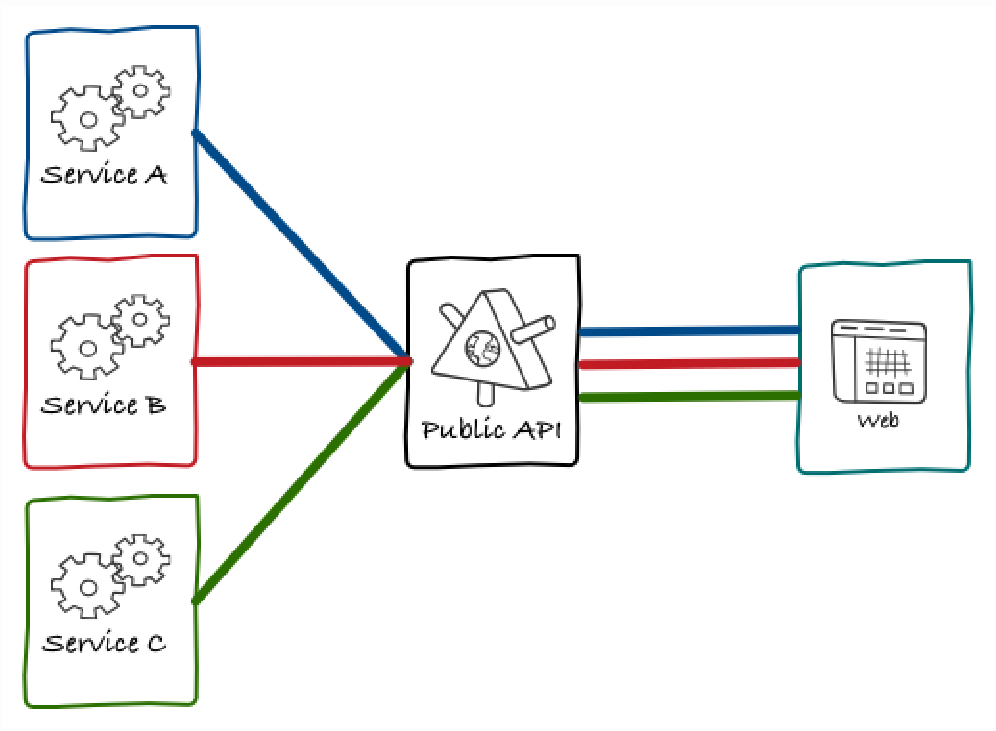
Logic and All logic was in one place. There was *one* system, and this system was *the* application.



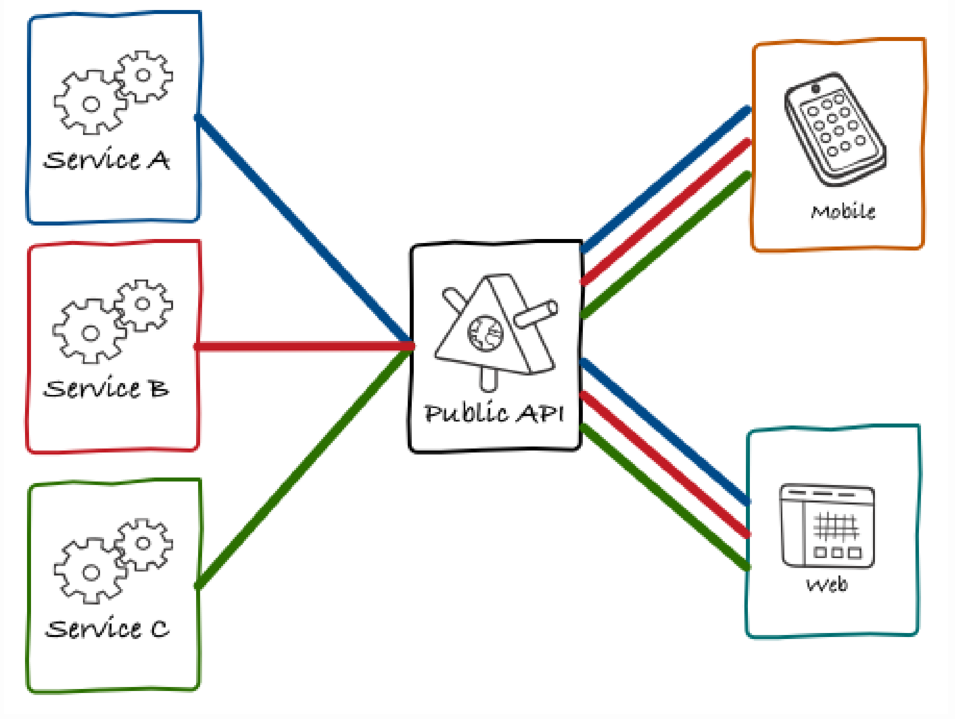
As described in a previous article, we have found many problems with this architecture and decided to extract logic into microservices. As successful as we were in extracting back-end services, for the longest time the *mothership* was still on the critical path for every single request.



The main motivation behind the architecture changes we were making was reducing our time-to-market for new features, and we have detected that our worst bottleneck was in any change that had to touch the monolith. Considering how often user interface changes, extracting its code from the monolith was an intuitive way to boost productivity. We then extracted our UI layer in its own component, and made it fetch data from our public API:



Back in 2011, when these architecture changes were happening, the vast majority of our users were on the web. As people like Fred Wilson have predicted, eventually this changed and our user base started using mobiles apps way more often than the web interface. SoundCloud has had mobile clients for both Android and iOS for a very long time and, similarly to our new web application, they talked directly to our public API.

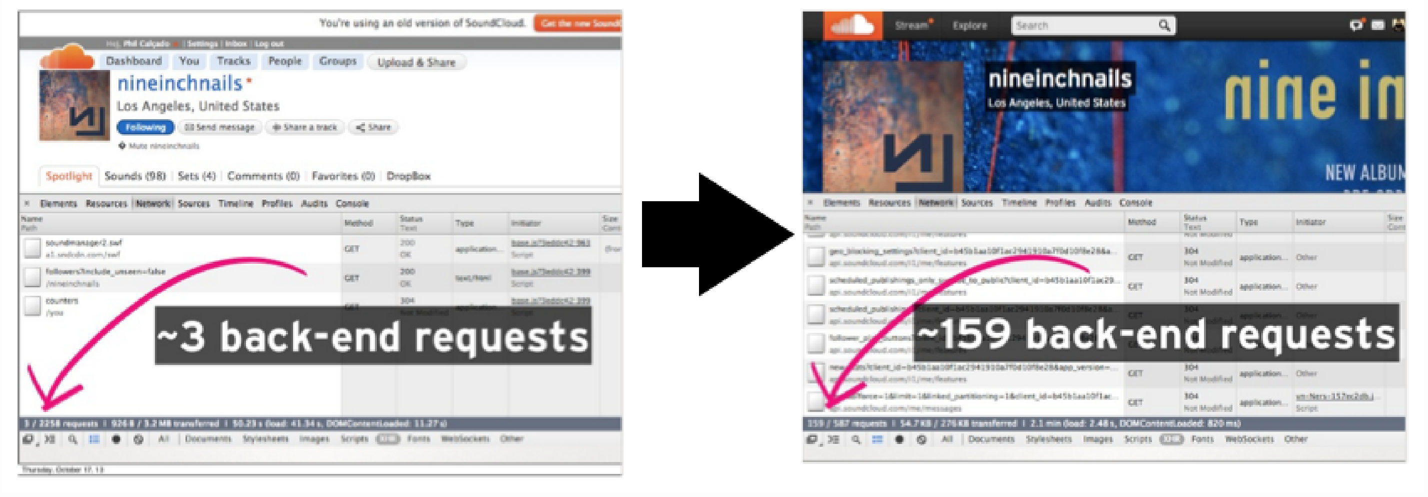


The challenges with dogfooding

In modern software engineering, dogfooding is usually considered a good thing. Building our products on top of our own API was perceived as the best way to make sure that our API had high-quality and was always up-to-date. In practice, we have experienced several problems with this approach.

The first issue we had was not necessarily related with technology, but a fundamental challenge for product development. If we were to use the public API only, there was nothing that we could offer in our platform that wouldn't be available for third-party API clients. As much as we wanted a thriving ecosystem of SoundCloud integrations, we were an advertisement business and as such we needed to make sure people were using our properties, not just our data. Creating features exclusive to our own applications meant that we had to constantly check for OAuth scopes in many places and make it very hard for people to spoof our *"official app"* keys.

On a more technical problem, our public APIs almost by definition are very generic. To empower third-party developers to build interesting integrations, you need to design an API that makes no assumptions about how the data is going to be used. This results in very fine-grained endpoints, which then require a large number of HTTP requests to multiple different endpoints to render even the simplest experiences. Below you can see how many requests we used to make in the monolithic days versus the number of those we make for the new web application:



To generate that single profile page, we would have to make many calls to different API endpoints, e.g.:

- `GET /tracks/1234.json` (the author of the track)
- `GET /tracks/1234/related.json` (the tracks to recommend as related)
- `GET /users/86762.json` (information about the track's author)
- `GET /users/me.json` (information about the current user)
- …

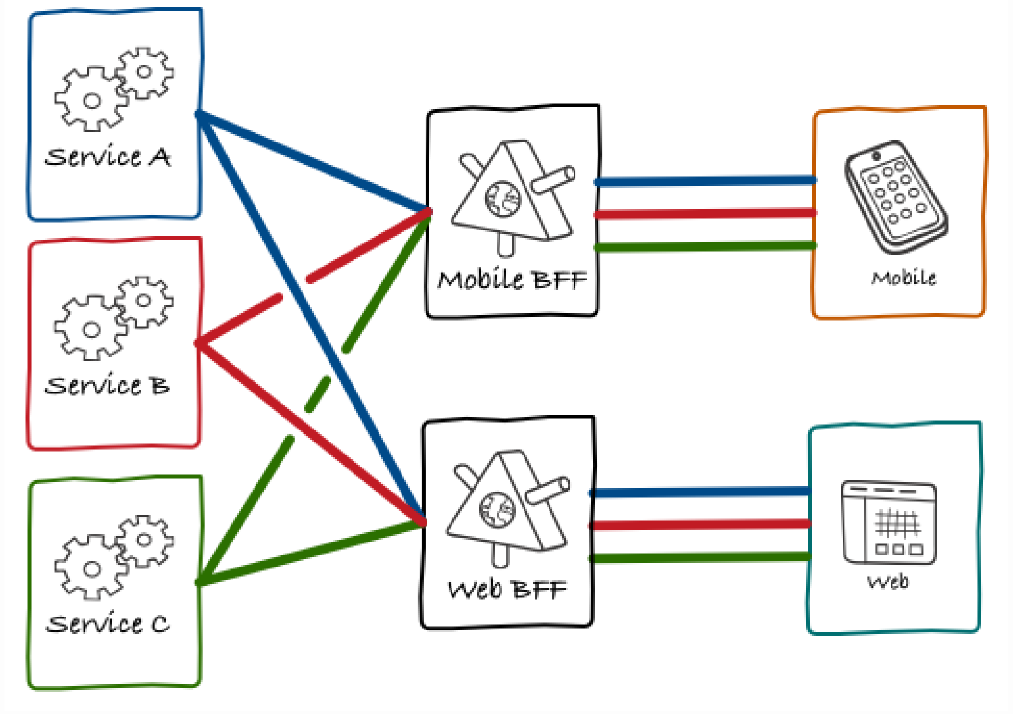
…which the web application would then merge to create the user profile page. While this problem exists on all platforms, it was even worse for our growing mobile user base that often used unreliable and slow wireless networks.

A third and even more annoying problem we had with the architecture above is that, even without the monolith, we still had a bottleneck on the API. Every time a team needed to change an existing endpoint we needed to make sure that the changes would not only not break any of our existing clients (including important third-party integrations). Whenever we added something new, we had to invest a lot of time in making sure that the new endpoint wasn't over-specialised for a specific app, that all clients could easily use them. All this coordination made our day-to-day much harder than it should be, and also made it almost impossible for us to do A/B testing and slow rollouts of new features.

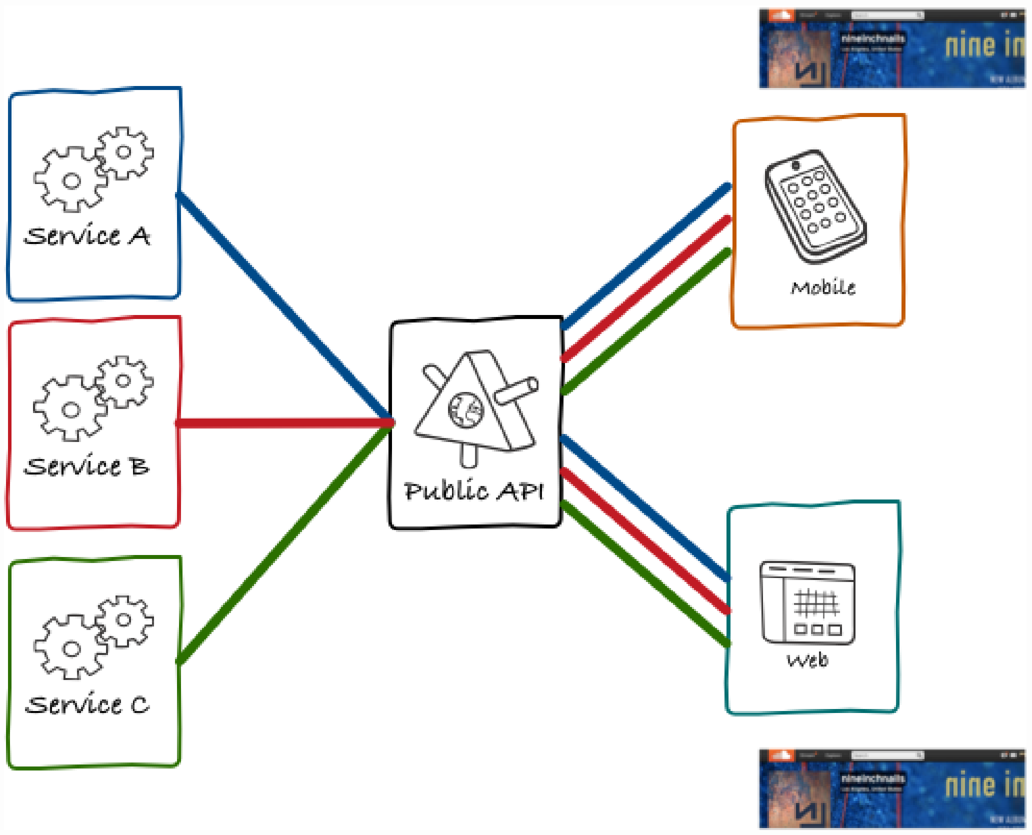
The Back-end for Front-end Pattern (BFF)

Almost one year after the debut of the architecture above, we started gearing up to develop what would be our new iOS application. This was a massive project which would ultimately change the user experience across all properties. With such high stakes, experimentation and iteration during development were crucial. As the engineering team started thinking about the application's architecture, we saw that the challenges described above would become a blocker for the project, we needed to re-think the way we were doing things.

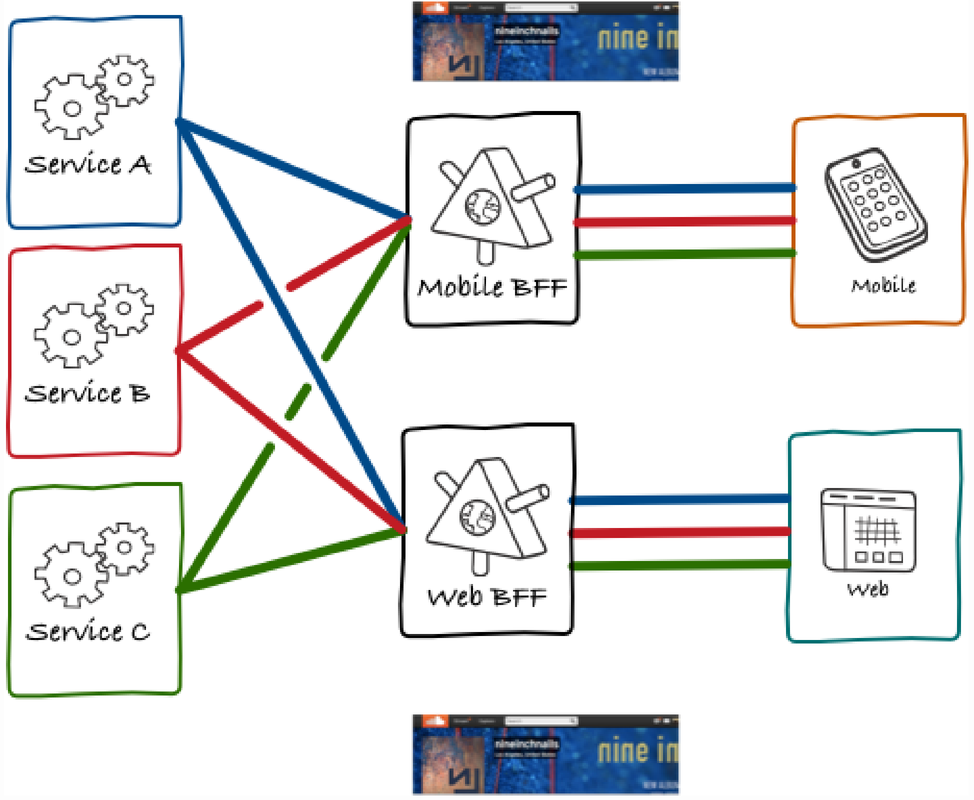
Our first proposed solution would be to have different APIs for mobile and web. The idea was that having the team working on the client own the API would allow for them to move much quicker as it required no coordination between parts. Our original idea was to have different *back-ends* for different *front-ends*. The term BFF was coined by our Tech Lead for web, Nick Fisher (my initial suggestion was *BEFFE*, but our Dutch-speaking team mates vetoed that option).



In its first incarnation, these back-ends still looked very much look like the public API, with many generic endpoints that required many calls from the client to render a single screen. Over time, though, we saw something interesting happening. Using the user profile page as an example, previously this as a concept that only existed on the client side. The web or mobile application would fetch data from various endpoints and use it to create an object that we called user profile. It was an *application-specific* object.



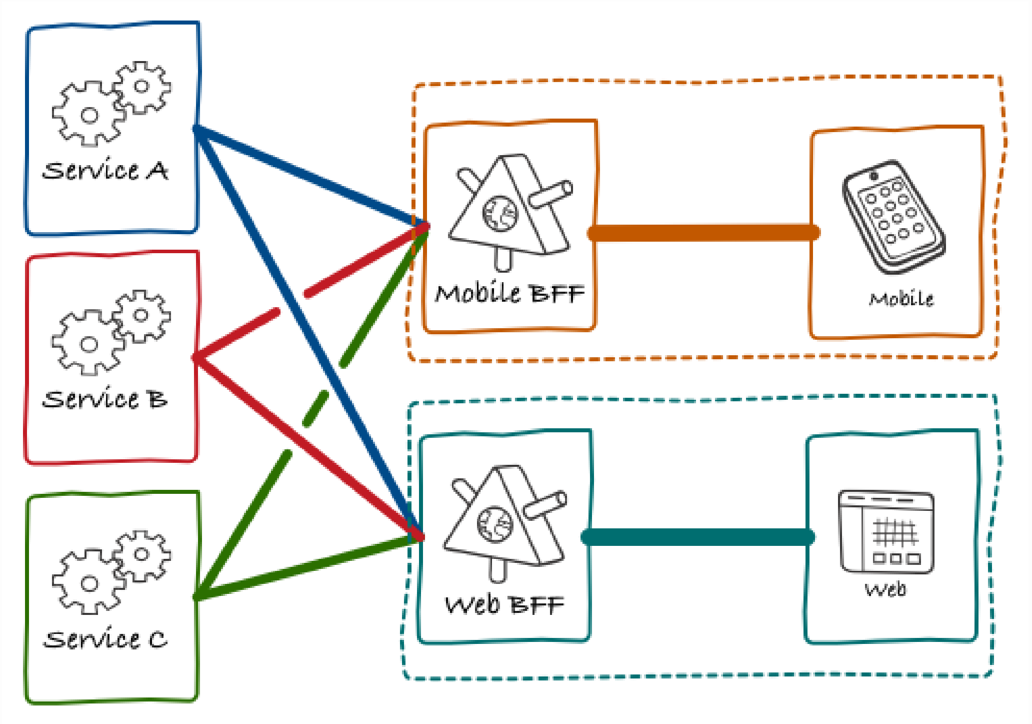
At some point our client teams realised that, since they owned the API, they could push this object down the API. They could extract all the logic that made many calls to different services and mashed them together into the user profile in their back-end.



This would ultimately both simplify the code and improve performance. Instead of making the multiple different calls to many endpoints described above all the client needed to request was a single resource:

- `GET /user-profile/123.json`

As we further experimented with this model, we found ourselves writing much of the Presentation Model in the BFF. At this stage we realised that the BFF wasn't an API *used* by the application. **The BFF was part of the application.**

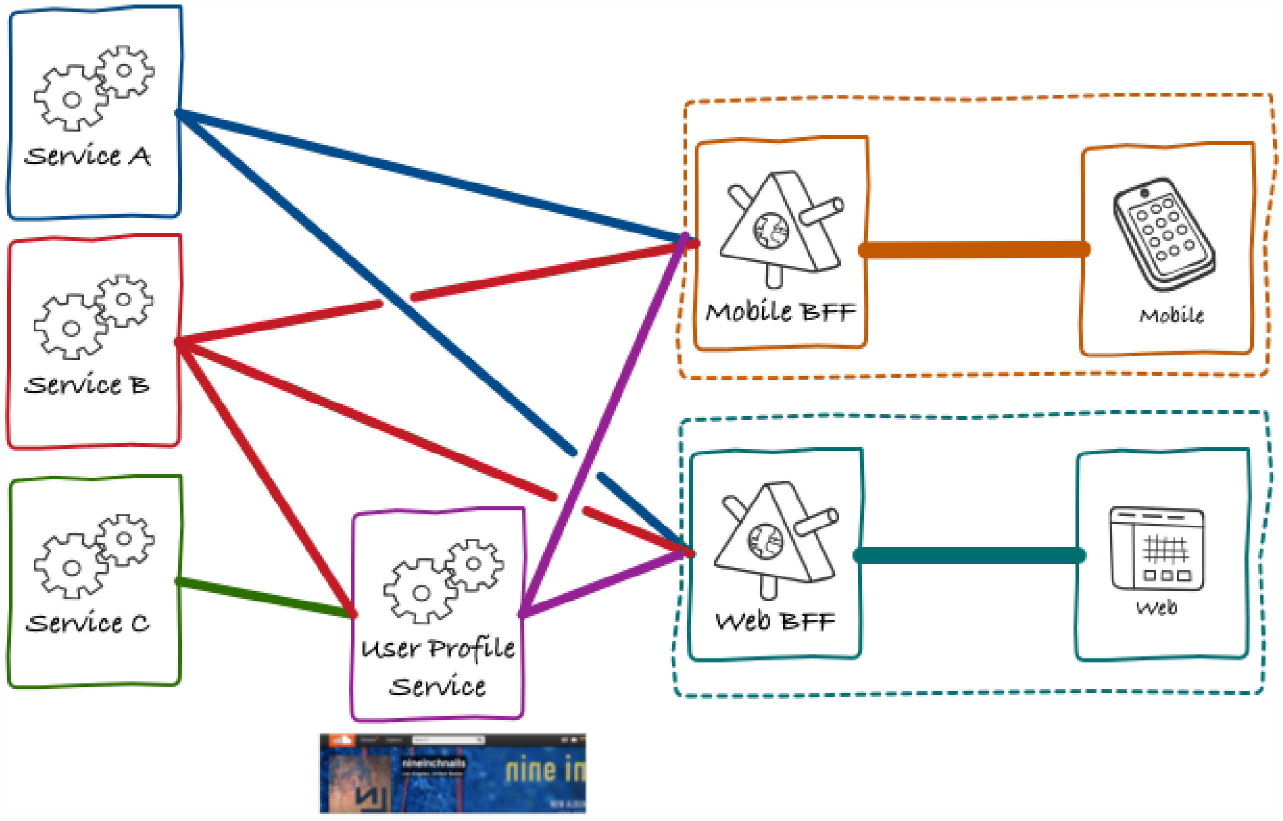


Eventually all of our properties, including APIs, started following this pattern.

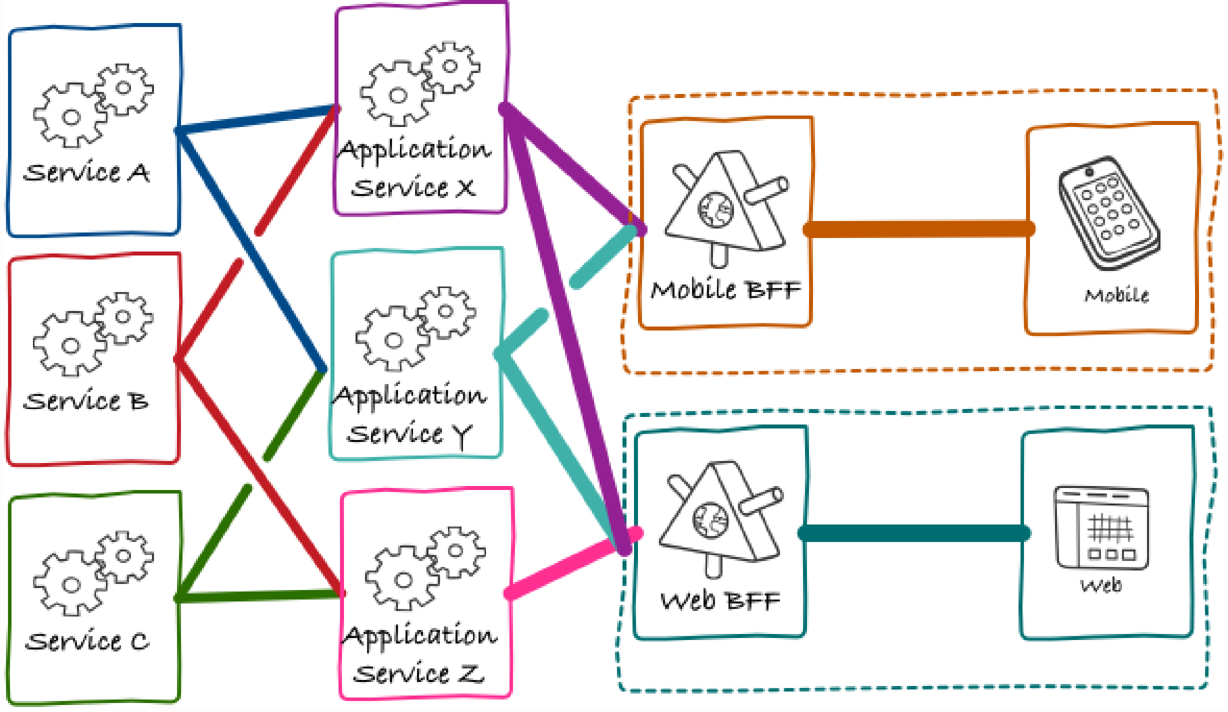
All the way down

At some point we had about five different BFFs in production, and we have begun looking at how to increase our productivity even further. Following with our user profile example, something that became obvious to us is that, given every single application had an equivalent of a user profile page, there was a lot of duplicated code across all BFFs fetching and merging data for them.

The duplication wasn't exact, larger screens like a web browser would have much more information on their user profile page than tiny mobile apps. Nevertheless, we saw the duplication as a *bad smell* indicating that we were missing an object in our domain model. To fix that, we created a `UserProfileService` that would deal with this duplicated logic.



Over time, we found more and more situations like these. We started consciously moving towards an architecture where most of the core objects understood by users had their own microservice backing them.



Acknowledgements

During my time at SoundCloud, the BFF was maintained by the *Core Engineering* team composed of Kristof Adriaenssens, Bora Tunca, Rahul Goma Phulore, Hannes Tydén, Vitor Pellegrino, Sam Starling, and I.

Fabio Kung, Paul Osman, Emerson Macedo, and Caio Filipini gave feedback on drafts of this article.

Phil Calçado

Phil Calçado

pcalcado
pcalcado