Pattern: Backends For Frontends

Written on Nov 18 2015

Single-purpose Edge Services for UIs and external parties

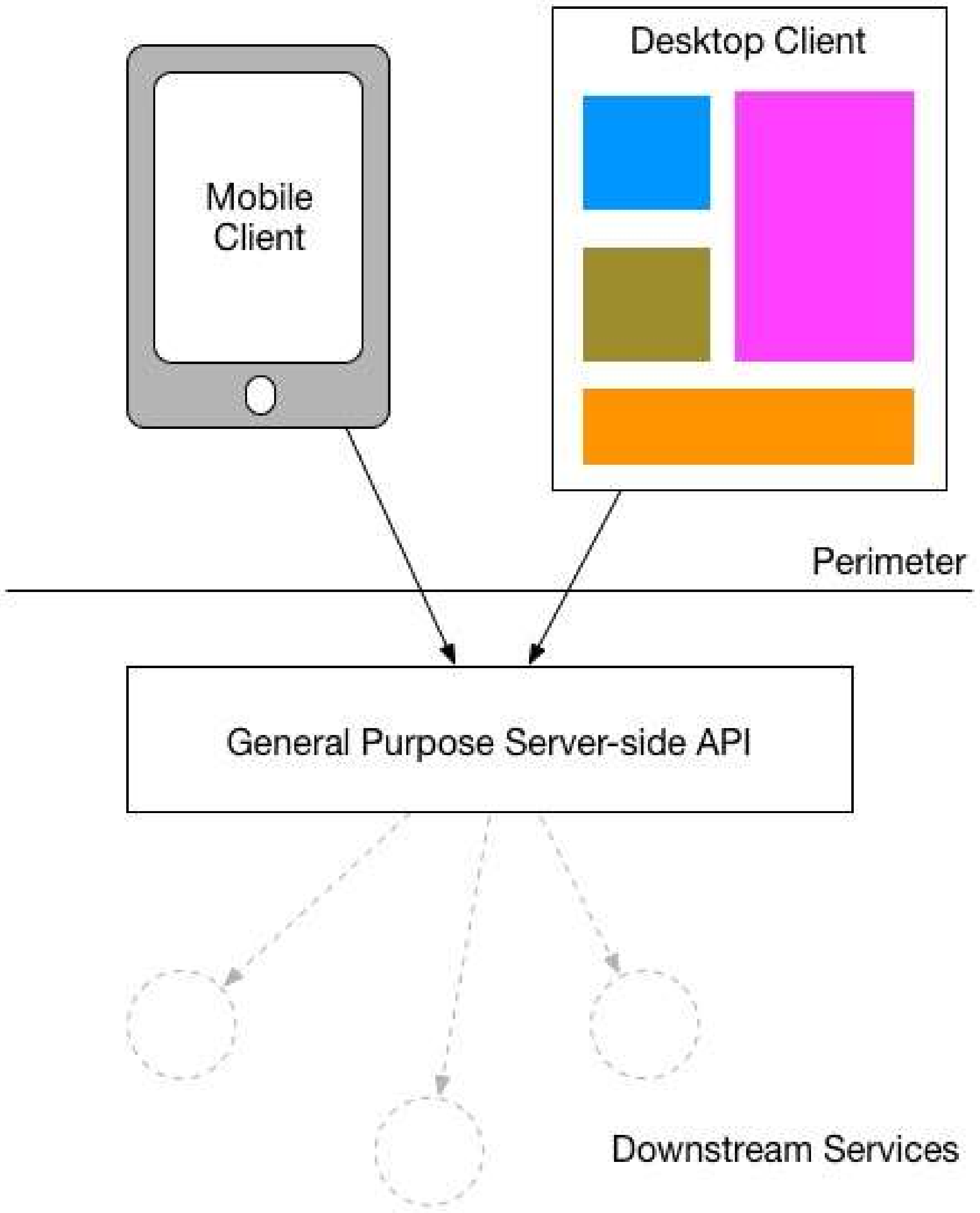
Introduction

With the advent and success of the web, the de facto way of delivering user interfaces has shifted from thick-client applications to interfaces delivered via the web, a trend that has also enabled the growth of SAAS-based solutions in general. The benefits of delivering a user interface over the web were huge - primarily as the cost of releasing new functionality was significantly reduced as the cost of client-side installs was (in most cases) eliminated altogether.

This simpler world didn't last long though, as the age of the mobile followed shortly afterwards. Now we had a problem. We had server-side functionality which we wanted to expose both via our desktop web UI, and via one or more mobile UIs. With a system that had initially been developed with a desktop-web UI in mind, we often faced a problem in accommodating these new types of user interface, often as we already had a tight coupling between the desktop web UI and our backed services.

The General-Purpose API Backend

A first step in accommodating more than one type of UI is normally to provide a single, server-side API, and add more functionality as required over time to support new types of mobile interaction:



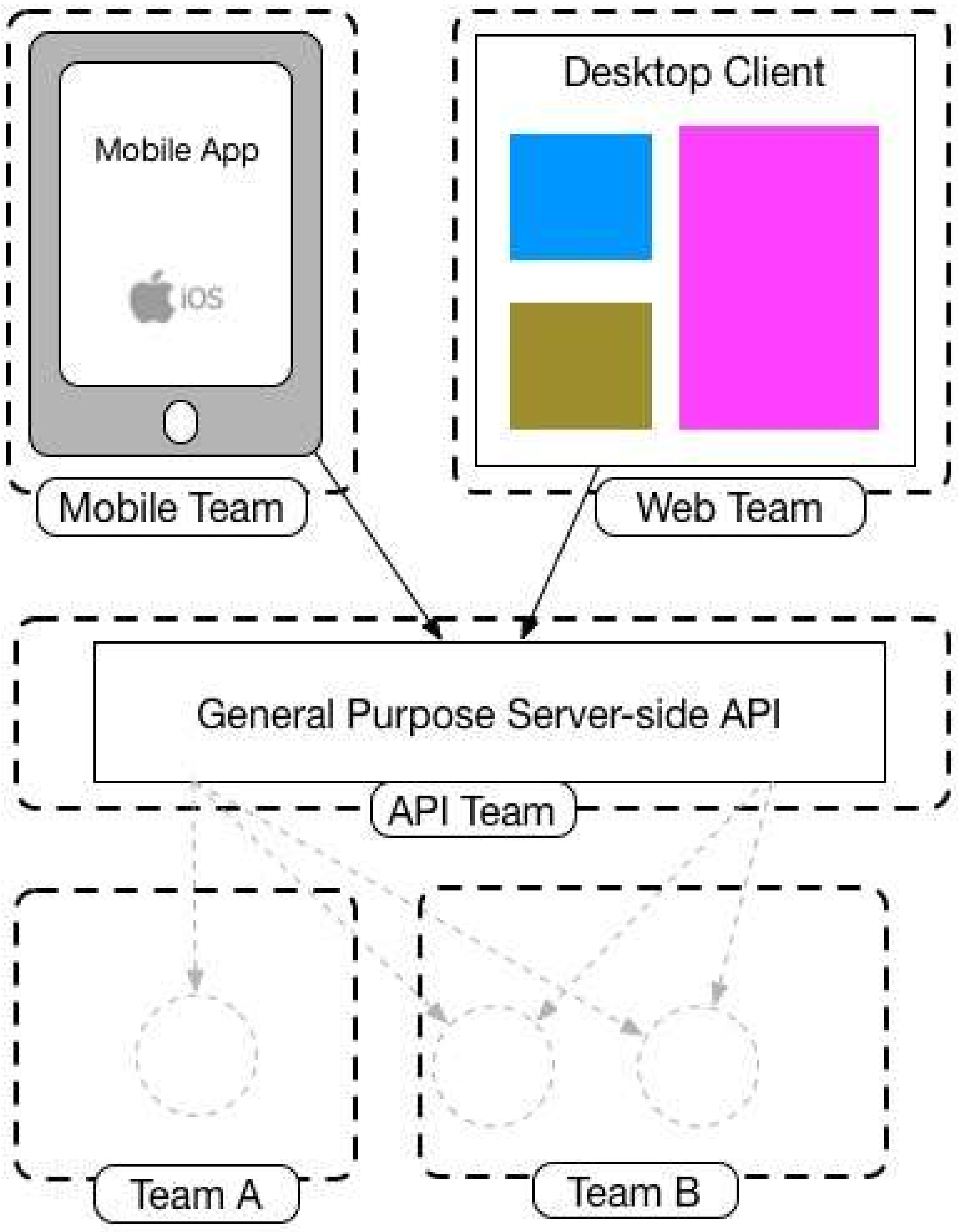
A general purpose API backend

If these different UIs want to make the same or very similar sorts of calls, then it can be easy for this sort of general-purpose API to be successful. However the nature of a mobile experience often differs drastically from a desktop web experience. Firstly, the affordances of a mobile device are very different. We have less screen real estate, which means we can display less data. Opening lots of connections to server-side resources can drain battery life and limited data plans. And secondly, the nature of the interactions we want to provide on a mobile device can differ drastically. Think of a typical bricks-and-mortar retailer. On a desktop app I might allow you to look at the items for sale, order online or reserve in store. On the mobile device though I might want to allow you scan bar codes to do price comparisons or give you context-based offers while in store. As we've built more and more mobile applications we've come to realise that people use them very differently and therefore the functionality we need to expose will differ too.

So in practice, our mobile devices will want to make different calls, fewer calls, and will want to display different (and probably less) data than their desktop counterparts. This means that we need to add additional functionality to our API backend to support our mobile interfaces.

Another problem with the general-purpose API backend is that they are by definition providing functionality to multiple, user-facing applications. This means that the single API backend can become a bottleneck when rolling out new delivery, as so many changes are trying to be made to the same deployable artifact.

The tendency for the general-purpose API backend to take on multiple responsibilities, and therefore require lots of work, often results in a team being created specifically to handle this code base. This can make the problem much worse, as now front-end teams have to interface with a separate team to get changes made - a team which will have to balance both the priorities of the different client teams, and also work with multiple downstream teams to consume new APIs as they become available. It could be argued that at this point we have just created a smart-piece of middleware in our architecture, something which is not focused on any particular business domain - something which goes against many people's views of what sensible Service Oriented Architecture should look like.

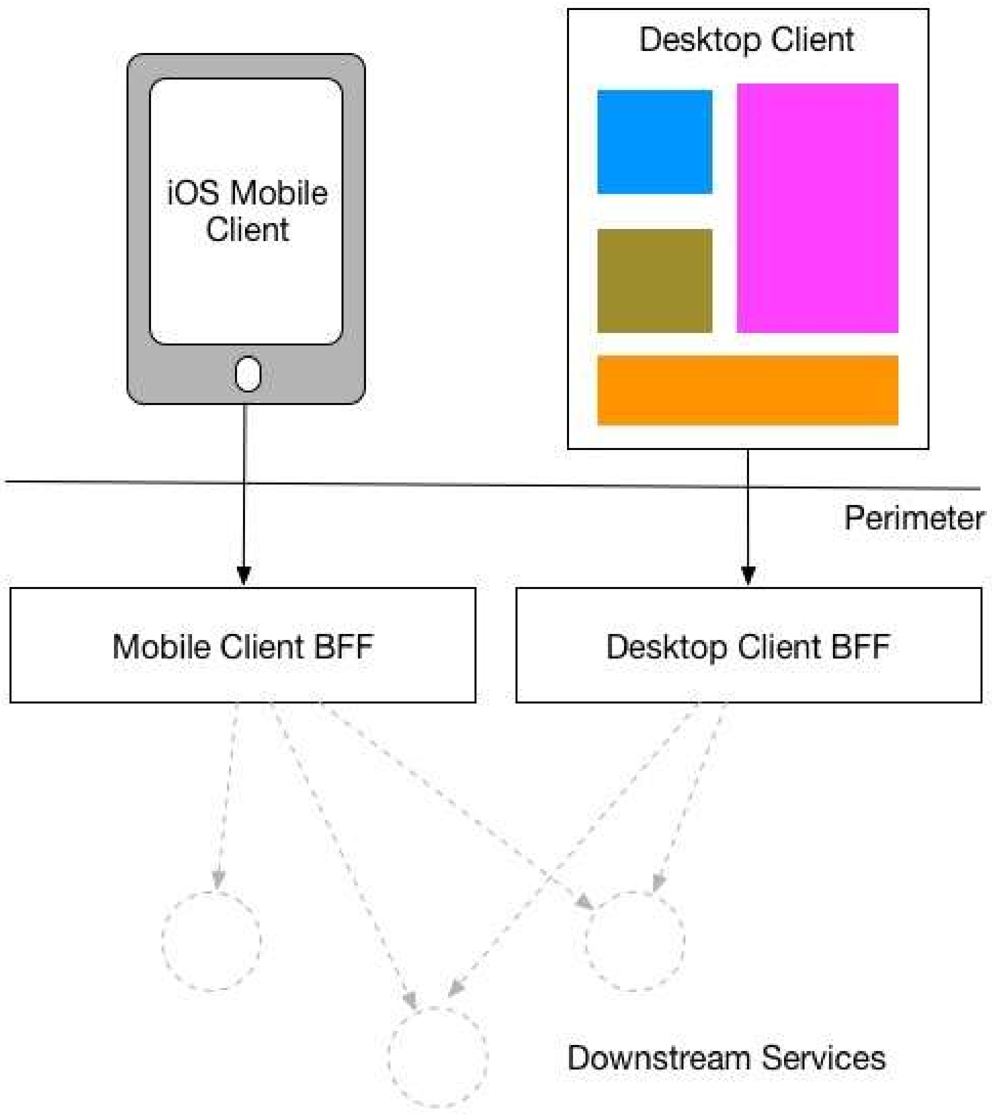


Common Team Structures When Using A Generic Backed API

Introducing The Backend For Frontend

One solution to this problem that I have seen in use at both REA and SoundCloud is that rather than have a general-purpose API backend, instead you have one backend per user experience - or as (ex-SoundClouder) Phil Calçado (http://philcalcado.com) called it a Backend For Frontend (BFF). Conceptually, you should think of the user-facing application as being two components - a client-side application living outside your perimeter, and a server-side component (the BFF) inside your perimeter.

The BFF is tightly coupled to a specific user experience, and will typically be maintained by the same team as the user interface, thereby making it easier to define and adapt the API as the UI requires, while also simplifying process of lining up release of both the client and server components.

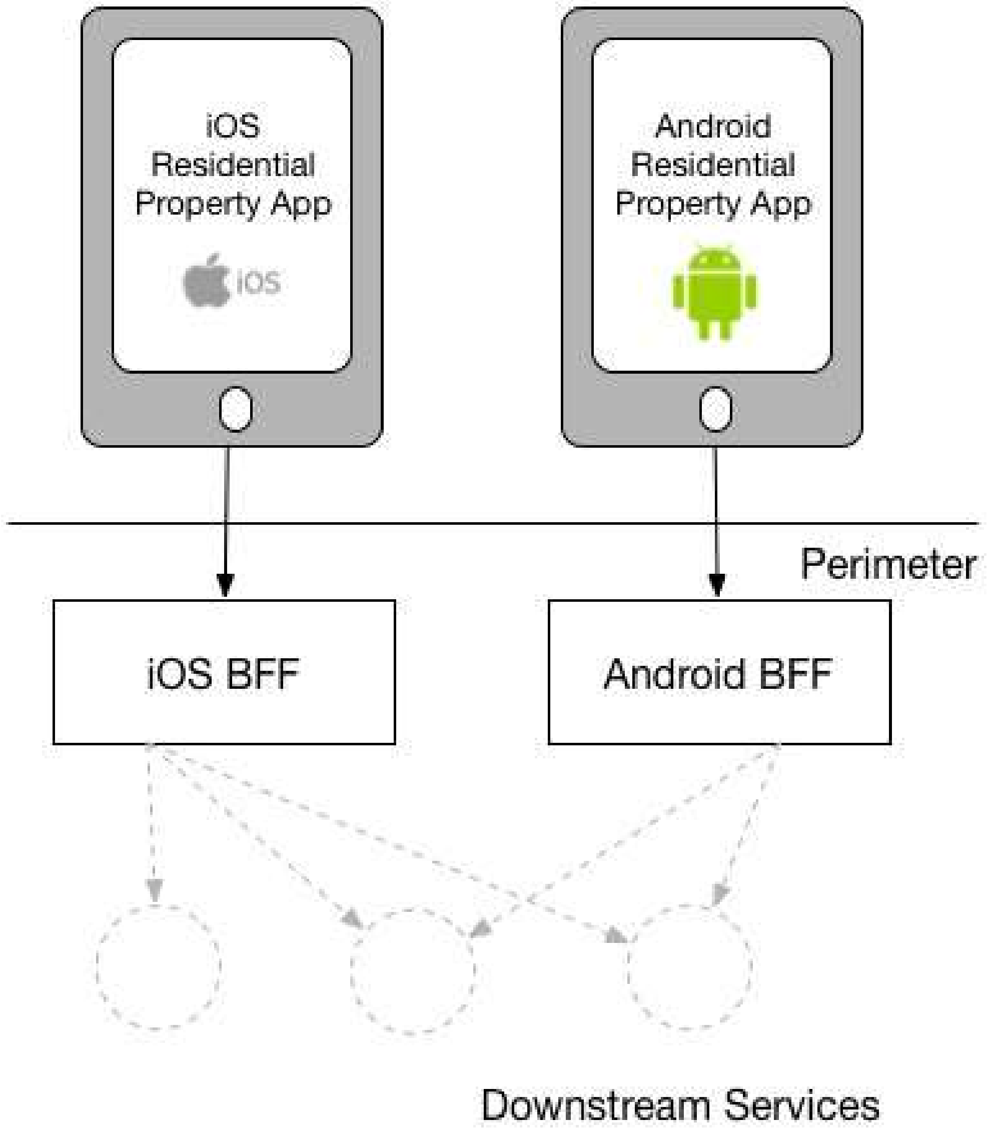


Using one server-side BFF per user interface

The BFF is tightly focused on a single UI, and just that UI. That allows it to be focused, and will therefore be smaller.

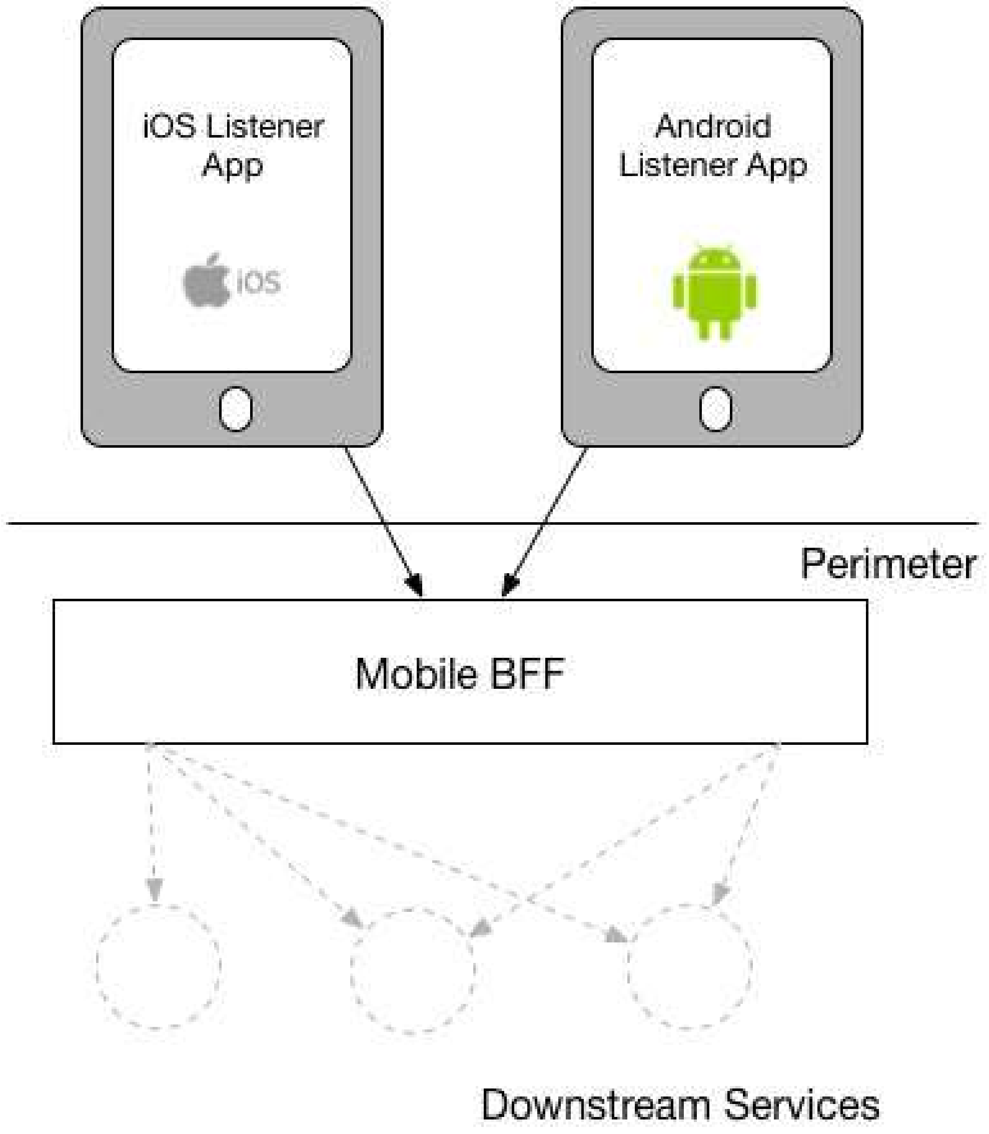
How Many BFFs?

When it comes to delivering the same (or similar) user experience on different platforms, I have seen two different approaches. The model I prefer is to strictly have a single BFF for each different type of client - this is a model I saw used at REA:


iOS Residential Property App
iOS
Android Residential Property App
Perimeter
iOS BFF
Android BFF
Downstream Services

Different mobile platform, different BFF, as used at REA

The other model, which I have seen in use at SoundCloud, uses one BFF per type of user interface. So both the Android and iOS versions of the listener native application use the same BFF:



Having one BFF for different mobile backends, as used at SoundCloud

My main concern with the second model is just that the more types of clients you have using a single BFF, the more temptation there may be for it to become bloated by handling multiple concerns. The key thing to understand here though is that even when sharing a BFF, it is for the same class of user interface - so while SoundCloud's listener Native applications for both iOS and Android use the same BFF, other native applications would use different BFFs (for example the new Creator application Pulse uses a different BFF). I'm also more relaxed about using this model if the same team owns both the Android and iOS applications and own the BFF too - if these applications are maintained by different teams, I'm more inclined to recommend the more strict model. So you can see your organisation structure as being one of the main drivers to which model makes the most sense (Conway's Law (https://en.wikipedia.org/wiki/Conway%27s_law) wins again). It's worth noting that the SoundCloud engineers I spoke to suggested that having one BFF for both Android and iOS listener applications was something they might reconsider if making the decision again today.

One guideline that I really like from Stewart Gleadow (who in turn credited Phil Calçado and Mustafa Sezgin) was 'one experience, one BFF'. So if the iOS and Android experiences are very similar, then it is easier to justify having a single BFF. If however they diverge greatly, then having separate BFFs makes more sense.

Pete Hodgson made the observation that BFFs work best when aligned around team boundaries, so team structure should drive how many BFFs you have. So that if you have a single mobile team, you should have one BFF, but if you had separate iOS and Android teams, you'd have separate BFFs. My concern is that team structures tend to be more fluid than our system design. So if you have a single BFF for mobile, then split the team into iOS and Android specialisations, do you then have to split the BFF too? If the BFFs were already separate, then splitting the team would be easier as you can reassign ownership of the already independent asset. The interplay of BFF and team structure is important though, something we'll explore more shortly.

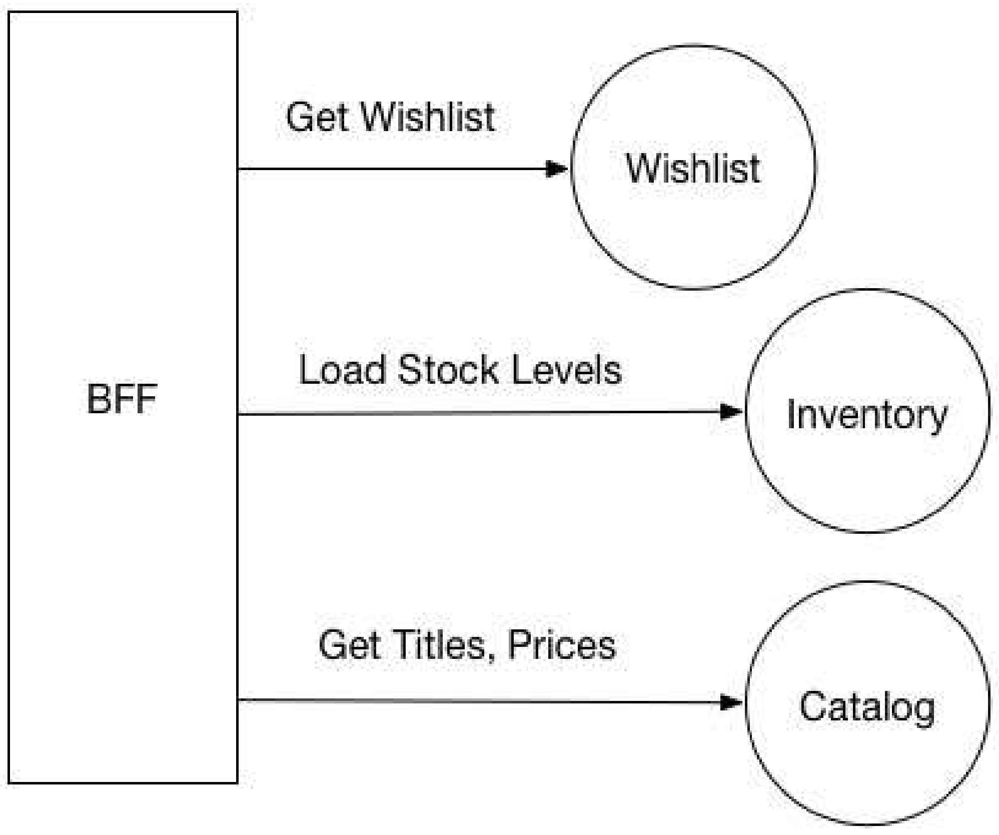
Often the driver towards having a smaller number of BFFs is around reusing server-side functionality to avoid too much duplication, but there are other ways to handle this which we'll cover shortly.

And Multiple Downstream Services (Microservices!)

BFFs can be a useful pattern for architectures where there are a small number of backend services. For organisations using a large number of services however they can be essential, as the need to aggregate multiple downstream calls to deliver user functionality increases drastically. In such situations it will be common for a single call in to a BFF to result in multiple downstream calls to microservices. For example, imagine an application for an e-commerce company. We want to pull back a list of items in a user's wish list, displaying stock levels, and price:

The Brakes - Give Blood	In Stock! (14 items remaining)	$5.99	**Order Now**
Blue Juice - Retrospectable	Out Of Stock	$17.50	**Pre Order**
Hot Chip - Why Make Sense?	Going fast (2 items left)	$9.99	**Order Now**

Multiple services hold the pieces of information we want. The Wishlist service stores information about the list, and IDs of each item. The Catalog service stores the name and price of each item, and the Stock levels are stored in our inventory service. So in our BFF we'd expose a method for retrieving the full playlist, which would consist of at least 3 calls:



Making multiple downstream calls to construct a view of a wishlist

From an efficiency point of view, it would be much smarter to run as many calls in parallel as possible. Once the initial call to the Wishlist service completes, ideally we'd like to then run the calls to the other services at the same time to reduce the overall call time. This need to mix calls that we want to run in parallel vs those that run in sequence can quickly become painful to manage, especially for more complex scenarios. This is one area where a reactive style of programming can help (such as that provided by RxJava (https://github.com/ReactiveX/RxJava) or Finagle's futures (https://twitter.github.io/scala_school/finagle.html) system) as the composition of multiple calls becomes easier to manage.

Failure modes though become important to understand. In our example above, we could insist that all downstream calls have to return in order for us to return a payload to our client. However is this sensible? Obviously we can't do anything if the Wishlist service is down, but if only the Inventory service was down, wouldn't it be better to just degrade the functionality we pass back to the client, perhaps just by removing the stock level indicator? These concerns have to be managed by the BFF itself in the first instance, but we also need to make sure that the client making the call to the BFF can interpret a partial response and render it correctly.

Reuse and BFFs

One of the concerns of having a single BFF per user interface is that you can end up with lots of duplication between the BFFs themselves. For example they may end up performing the same types of aggregation, have the same or similar code for interfacing with downstream services etc. Some people react to this by wanting to merge these back together, and so have a general-purpose aggregating Edge API service. This model has proven time and again to lead to highly bloated code with multiple concerns squashed together.

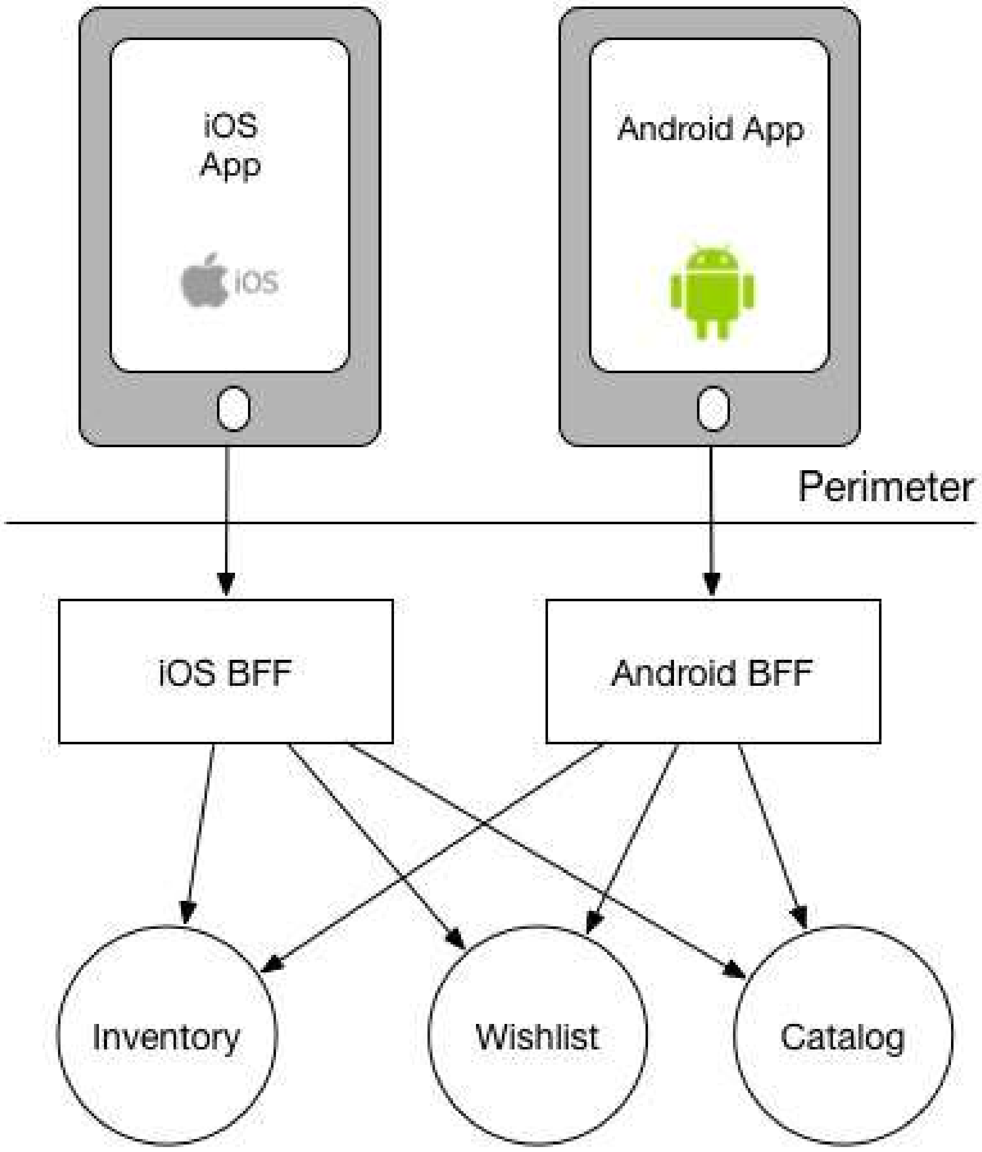
As I have said many times before, I am fairly relaxed about duplicated code across services. Which is to say that while in a single process boundary I will typically do whatever I can to refactor out duplication into suitable abstractions, I don't have the same reaction when confronted by duplication across services. This is mostly as I am often more worried about the potential for extracting shared code to lead to tight coupling between services - something I am more worried about than duplication in general. That said, there are certainly cases where this is warranted.

My colleague Pete Hodgson has pointed out that when you don't have BFFs, then often the 'common' logic ends up being baked into the different clients themselves. Due to the fact that these clients use very different technology stacks, identifying the fact that this duplication is occurring can be difficult. With organisations tending to have a common technology stack for server-side components, having multiple BFFs with duplication may be easier to spot and factor out.

When the time does arise to extract shared code, there are two obvious options. The first, which is often cheapest but more fraught, is to extract a shared library of some sort. The reason this can be problematic is that shared libraries are a prime source of coupling, especially when used to generate client-libraries for calling downstream services. Nonetheless there are situations where this feels right - especially when the code being abstracted is purely a concern inside the service.

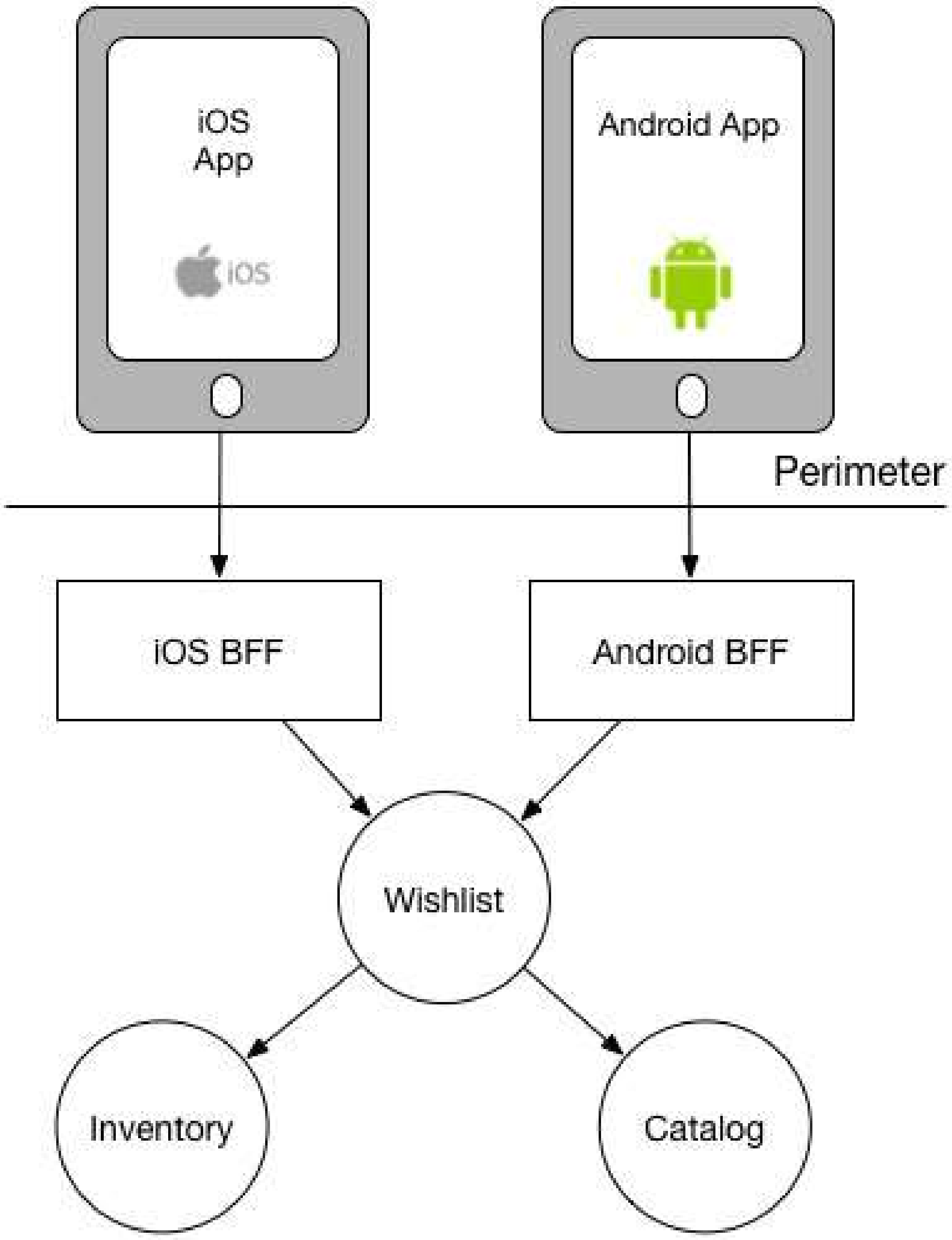
The other option is to extract out the shared functionality in a new service, which can work well if you can conceptualise the new service has something modeled around the domain in question.

A variation of this approach might be to push aggregation responsibilities to services further downstream. Take the example above where we discussed rendering of a wish list. Let's imagine we are rendering a wishlist in two places - on Android, iOS Web. Each of our BFFs are making the same three calls:



Multiple BFFs performing the same tasks

Instead, we could change the Wishlist service to make the downstream calls for us, thereby simplifying the job for the callers:



Pushing aggregation duties further downstream to remove duplication in BFFs

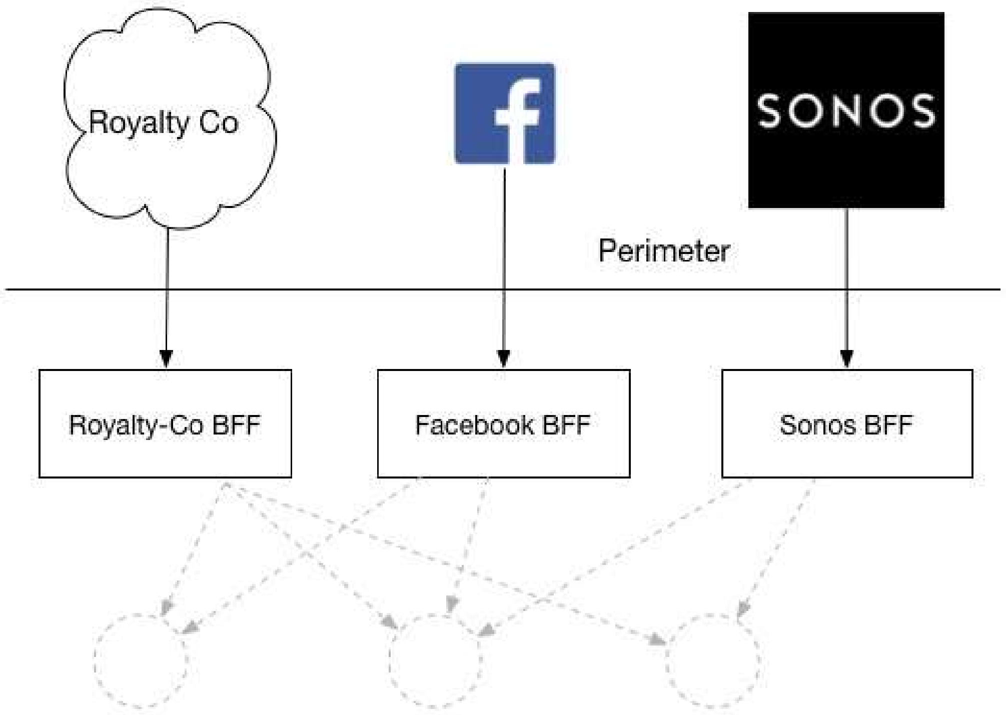
I have to say that the same code being used in two places wouldn't necessarily cause me to want to extract out a service in this way, but I'd be certainly considering it if the transaction cost of creating a new service was low enough, or I was using it in more than a couple of places (for example maybe on the desktop web). I think the old adage of creating an abstraction when you're about to implement something for the 3rd time still feels like a good rule of thumb, even at the service level.

BFFs for Desktop Web and Beyond

You can think of BFFs as just having a use in solving the constraints of mobile devices. The desktop web experience is typically delivered on more powerful devices with better connectivity, where the cost of making multiple downstream calls is manageable. This can allow your web application to make multiple calls directly to downstream services without the need for a BFF.

I have seen situations though where the use of a BFF for the web too can be useful. When you are generating a larger portion of the web UI on the server-side (e.g using server-side templating), a BFF is the obvious place where this can be done. It can also simplify caching somewhat as you can place a reverse proxy in front of the BFF, allowing you to cache the results of aggregated calls (although you have to make sure you set your cache controls accordingly to ensure that the aggregated content's expiry is as short as the freshest piece of content in the aggregation needs it to be). I've seen it used multiple times in fact without calling it a BFF - in fact the general-purpose API backend often grows from such a beast.

I've seen at least one organisation use BFFs for other external parties that need to make calls. Coming back to my perennial example of a music shop, I might expose a BFF to allow 3rd parties to extract royalty payment information, provide Facebook integration or allow streaming to a range of set-top box devices:



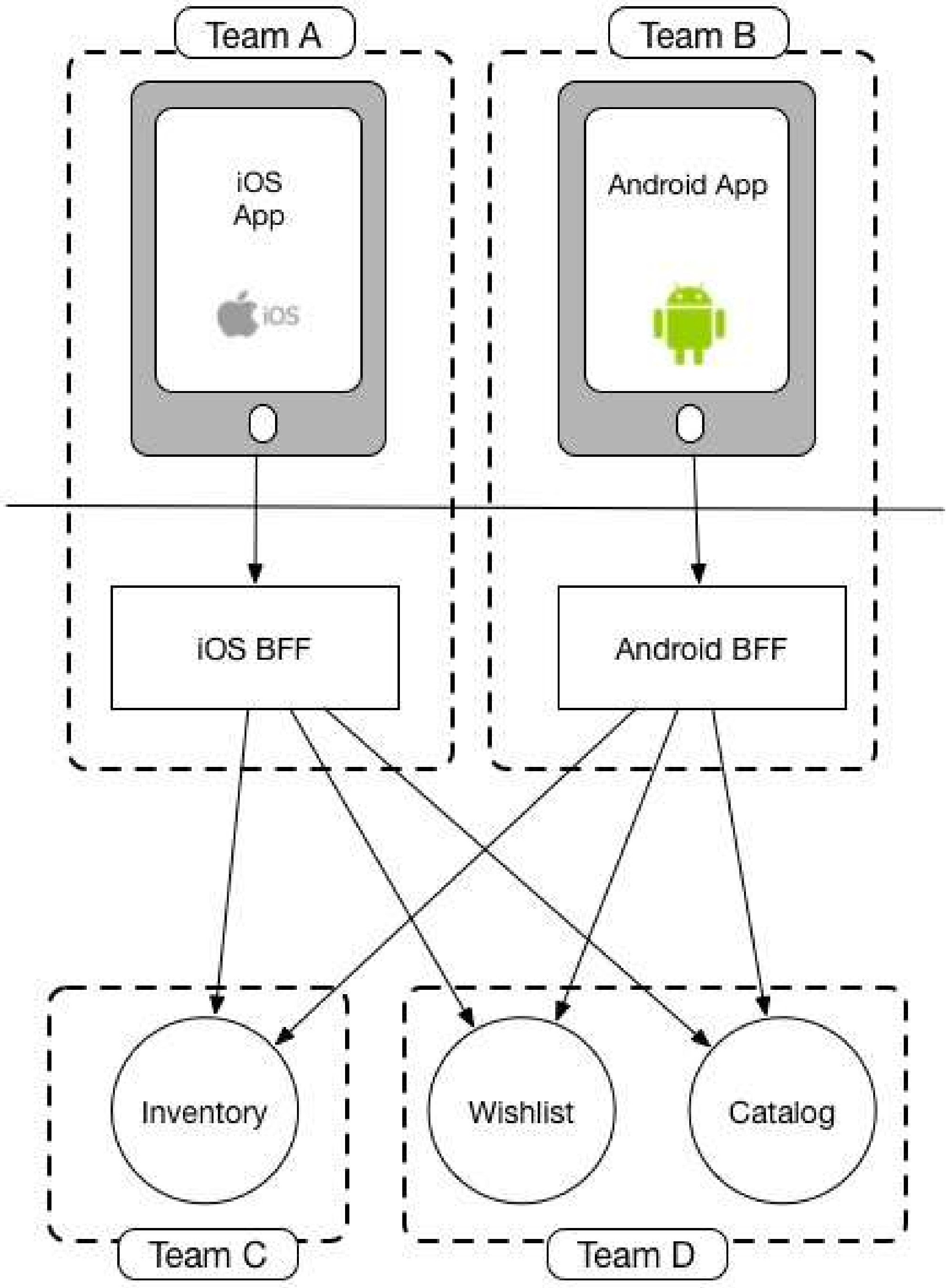
Exposing APIs to 3rd Parties using a BFF

This approach can be especially effective as third-parties often have limited to no ability (or desire) to use or change the API calls they make. With a general-purpose API backend, you may have to keep old versions of the API around just to satisfy a small subset of your outside parties unable to make a change - with BFF this problem is substantially reduced.

And Autonomy

Quite often we see situation where one team is working on a frontend, and a different team is creating the backend services. In general, we're trying to avoid this by moving to microservices which are aligned around business verticals, but even then there are situations where this is hard to avoid. Firstly, at a certain level of scale or complexity, multiple teams need to get involved. Secondly, the depth of technical skills required to execute a good Android or iOS experience often need specialised teams.

So teams building user interfaces are confronted with the situation that they are calling an API which another team is driving, and often than API is evolving while the user interface is being developed. The BFF can help here, especially if it is owned by the team creating the user interface. They evolve the API of the BFF at the same time as creating the front end. They can iterate both quickly. The BFF itself still needs to call the other downstream services, but this can be done without having to interrupt development of the user interface.

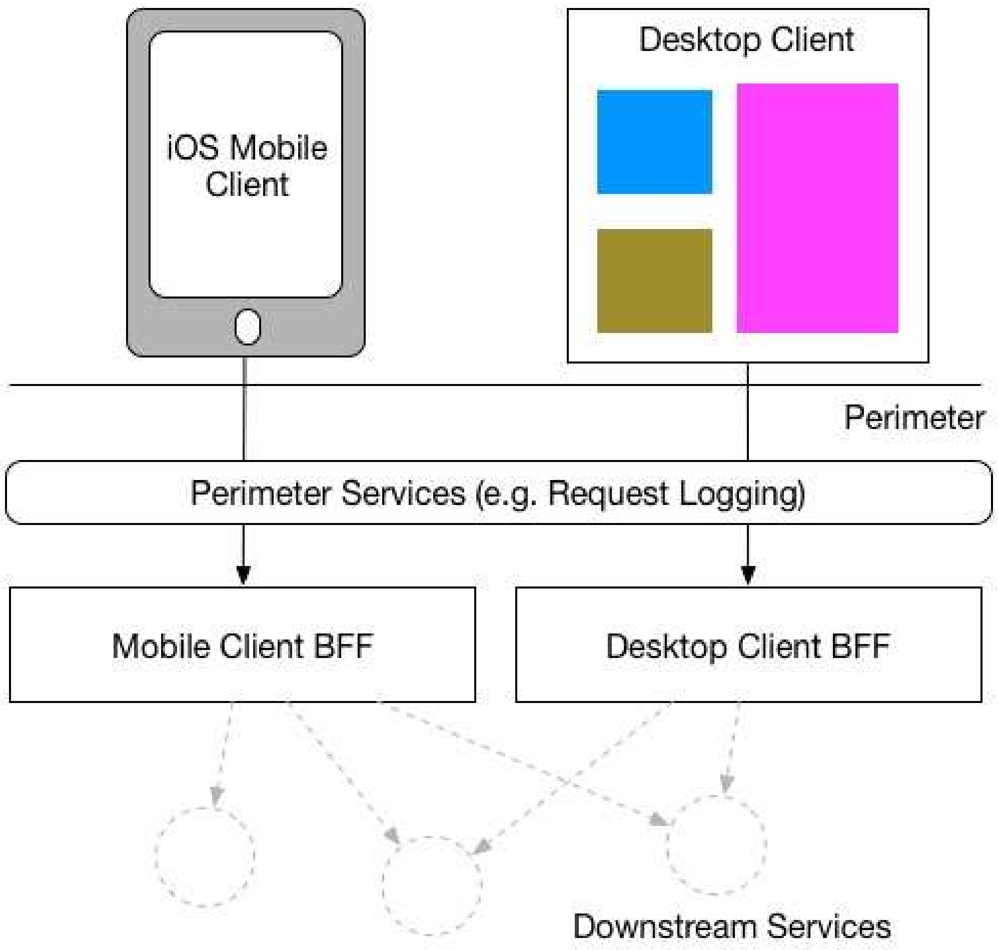


Example team ownership boundaries when using BFFs

The other benefit of using a BFF aligned along team boundaries like this is that the team creating the interface can be much more fluid in thinking about where functionality lives. For example they could decide to push functionality on to the server-side to promote reuse in the future and simplify a native mobile application, or to allow for the faster release of new functionality (as you can bypass the app store review processes). This decision is one that can be made by the team in isolation if they own both the mobile application and the BFF - it doesn't require any cross-team coordination.

General Perimeter Concerns

Some people use BFFs to implement generic perimeter concerns, such as authentication/authorisation or request logging. I'm torn about this. On the one hand, much of this functionality is so generic that I'd be inclined to implement it using another layer sitting further upstream, perhaps using something like a tier of Nginx or Apache servers. On the other hand, such an additional layer can't help but add latency. BFFs are often used in microservice environment where we are already very sensitive about latency due to the high number of network calls being made. Also, the more layers you have to deploy to make a production-like stack can make development and test more complex - having all of these concerns inside the BFF as a more self-contained solution can be attractive as a result:



Using a network appliance to implement generic perimeter concerns

As we discussed earlier, another way to factor out this duplication could be to use a shared library. Assuming your BFFs are using the same technology, this shouldn't be too difficult, although the usual caveats about shared libraries in a microservice architecture apply.

When To Use

For an application which is only providing a web UI, I suspect a BFF will only make sense if and when you have a significant amount of aggregation required on the server-side. Otherwise, I think other UI composition techniques can work just as well without requiring an additional server-side component (I'll hopefully talk about those soon).

The moment that you need to provide specific functionality for a mobile UI or third party though, I would strongly consider using a BFFs for each party from the outset. I might reconsider if the cost of deploying additional services is high, but the separation of concerns that a BFF can bring make it a fairly compelling proposition in most cases. I'd be even more inclined to use a BFF if there is a significant separation between the people building the UI and downstream services, for reasons outlined above.

Further Reading (And Viewing)

- Since I wrote this piece, Lukasz Plotnicki from ThoughtWorks has published a great article (https://www.thoughtworks.com/insights/blog/bff-soundcloud) on SoundCloud's use of the BFF pattern
- Lukasz being interviewed (http://softwareengineeringdaily.com/2016/02/04/moving-to-microservices-at-soundcloud-with-lukasz-plotnicki/) about the pattern (and other things) on a recent episode of the Software Engineering Podcast.
- Bora Tunca from SoundCloud also goes into more detail during a talk (https://www.youtube.com/watch?v=jfN6HOgURXM) at microxchg 2016.

Conclusion

Backends For Frontends solve a pressing concern for mobile development when using microservices. In addition they provide a compelling alternative to the general-purpose API backend, and many teams make use of them for purposes other than just mobile development. The simple act of limiting the number of consumers they support makes them much easier to work with and change, and helps teams developing customer-facing applications retain more autonomy.

Thanks go to Matthias Käppler, Michael England, Phil Calçado, Lukasz Plotnicki, Jon Eaves, Stewart Gleadow and Kristof Adriaenssens for their help in researching this article, and Giles Alexander, Ken McCormack, Sriram Viswanathan, Kornelis Sietsma, Hany Elemary, Martin Fowler, Vladimir Sneblic, and Pete Hodgson for general feedback. I'd really appreciate any further feedback too, so feel free to leave a comment below!

→ **See more patterns (/patterns/)**.

ALSO ON **SAMNEWMAN.IO**

8 years ago · 1 comment
New Online Training

9 years ago · 5 comments
Magpie Talkshow Episode 10

2 y
**D
Pl**

81 Comments 1 Login ▾

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

92 Share Best Newest Oldest

Ahmed Ramadan
8 years ago

Hi Sam,
I have one question here , if i took the approach to add the perimeter layer (api management layer) to be used only for Authentication , Throttling , routing) and BFF for orchestration and optimization for mobile needs.
If i have one Mobile BFF exposing multiple application services like products, user details, etc.. would it make sense to divide the BFF per service call and make each call a deployable unit and maintained separately.
having the perimeter layer to handle the routing by mapping each BFF service to one API. or the otherwise to