BFF (Backends For Frontends)

- Pattern: Backends For Frontends

Pattern: Backends For Frontends

https://samnewman.io/patterns/architectural/bff/

Sam Newman

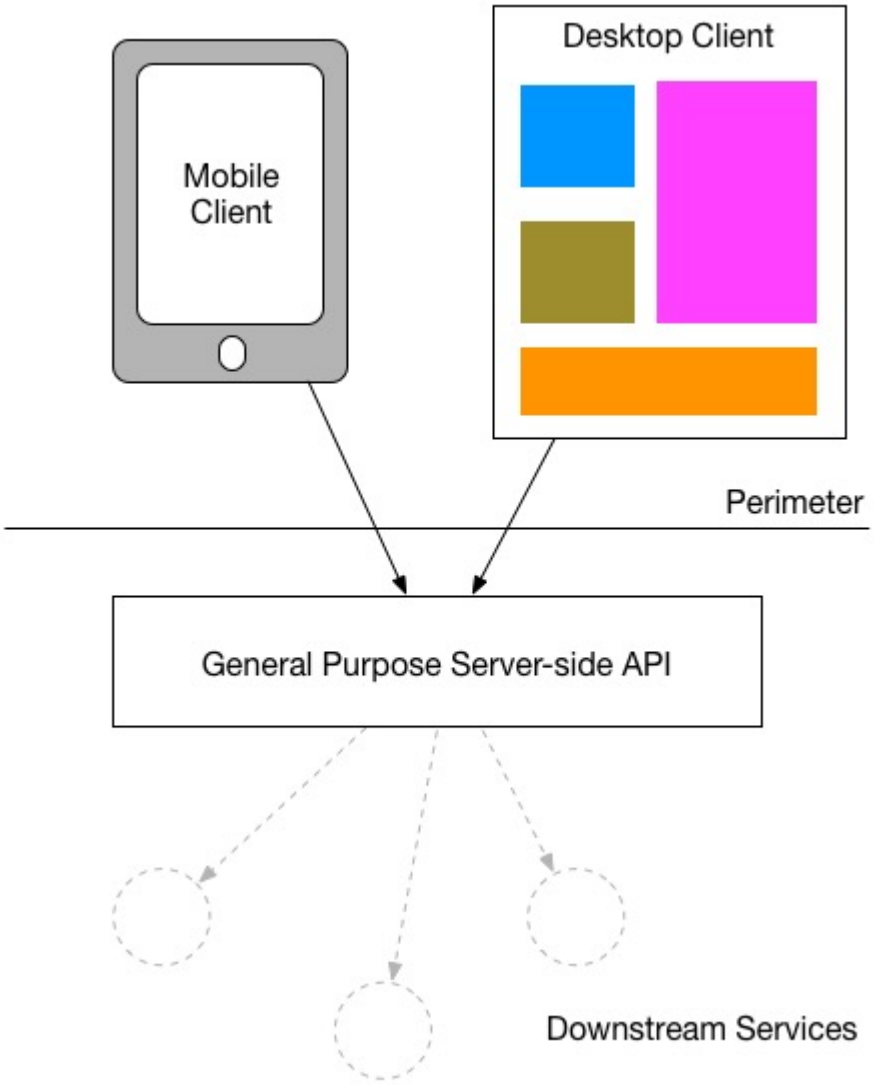
Introduction

웹의 출현과 성공으로 인해 사용자 인터페이스를 제공하는 방식은 두꺼운 클라이언트 애플리케이션에서 웹을 통한 인터페이스 제공으로 사실상 전환되었으며, 이는 전반적인 SaaS 기반 솔루션의 성장도 촉진하는 계기가 되었습니다. 웹을 통해 사용자 인터페이스를 제공하는 것은 큰 이점을 가져왔습니다. 가장 중요한 것은 클라이언트 측 설치 비용이 대부분의 경우 완전히 제거됨에 따라 새로운 기능을 출시하는 비용이 크게 감소했다는 점입니다.

하지만 이러한 단순한 환경은 오래가지 못했습니다. 곧이어 모바일 시대가 도래했기 때문입니다. 이제 우리는 새로운 문제에 직면하게 되었습니다. 서버 측 기능을 데스크톱 웹 UI뿐만 아니라 하나 이상의 모바일 UI를 통해서도 노출해야 했습니다. 초기에는 데스크톱 웹 UI를 염두에 두고 시스템을 개발했기 때문에, 이러한 새로운 유형의 사용자 인터페이스를 수용하는 데 어려움을 겪는 경우가 많았습니다. 이는 종종 데스크톱 웹 UI와 백엔드 서비스 간의 강한 결합(tight coupling)으로 인해 발생하는 문제였습니다.

The General-Purpose API Backend

다양한 유형의 UI를 수용하는 첫 번째 단계는 일반적으로 단일 서버 사이드 API를 제공하는 것이며, 시간이 지나면서 새로운 유형의 모바일 상호작용을 지원하기 위해 필요한 기능을 추가하는 것입니다.



A general purpose API backend

만약 서로 다른 UI들이 동일하거나 매우 유사한 호출을 수행하고자 한다면, 이러한 범용 API는 성공적으로 작동하기 쉬울 것입니다. 그러나 모바일 경험의 특성은 데스크톱 웹 경험과 크게 다를 수 있습니다.

우선, 모바일 기기의 제공 기능(affordance)은 데스크톱과 매우 다릅니다. 화면 공간이 제한적이므로 표시할 수 있는 데이터의 양이 적습니다. 또한, 서버 측 리소스에 다수의 연결을 여는 것은 배터리 소모를 가속화하고 제한된 데이터 요금제를 빠르게 소진할 수 있습니다.

둘째로, 모바일 기기에서 제공하고자 하는 상호작용 방식도 데스크톱과는 크게 다를 수 있습니다. 예를 들어, 전형적인 오프라인 소매업체를 생각해보겠습니다. 데스크톱 애플리케이션에서는 사용자가 판매 중인 상품을 검색하고, 온라인 주문을 하거나 매장에서 예약하는 기능을 제공할 수 있습니다. 하지만 모바일 기기에서는 바코드를 스캔하여 가격 비교를 하거나 매장 내에서 상황에 맞는 할인 혜택을 제공하는 기능을 원할 수도 있습니다.

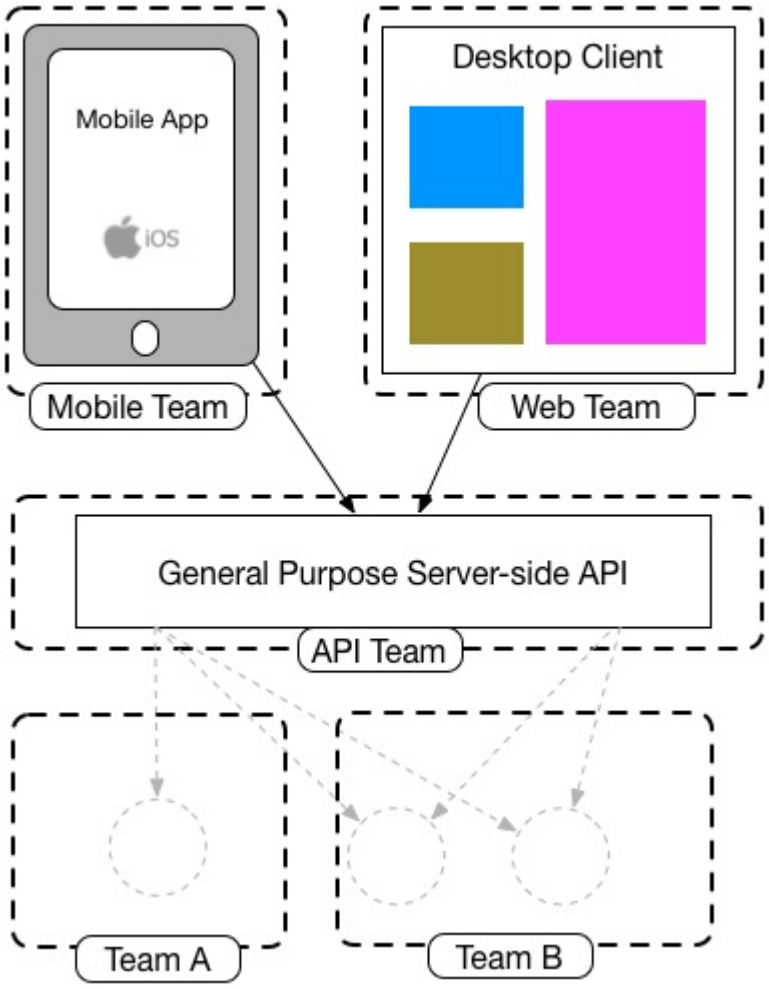
더 많은 모바일 애플리케이션을 개발하면서, 우리는 사람들이 모바일 앱을 매우 다르게 사용한다는 사실을 깨닫게 되었습니다. 따라서 노출해야 할 기능 또한 달라질 수밖에 없습니다.

실제로 모바일 기기는 데스크톱보다 다른 방식으로, 더 적은 호출을 수행하며, 다른(그리고 아마도 더 적은) 데이터를 표시하고자 합니다. 이는 모바일 인터페이스를 지원하기 위해 API 백엔드에 추가적인 기능을 제공해야 한다는 것을 의미합니다.

범용 API 백엔드의 또 다른 문제는 기본적으로 여러 개의 사용자 중심 애플리케이션에 기능을 제공한다는 점입니다. 이는 하나의 API 백엔드가 새로운 기능을 배포하는 과정에서 병목 현상이 발생할 가능성이 높아지며, 많은 변경 사항이 동일한 배포 가능한 아티팩트(artifact)에서 이루어져야 한다는 부담을 초래할 수 있습니다.

범용 API 백엔드는 여러 책임을 떠안게 되는 경향이 있으며, 이로 인해 많은 작업이 필요해지면서, 결국 이 코드베이스를 전담하는 팀이 구성되기도 합니다. 그러나 이는 문제를 더욱 악화시킬 수 있습니다. 이제 프런트엔드 팀이 변경 사항을 적용하려면 별도의 팀과 협업해야 하며, 이 팀은 서로 다른 클라이언트 팀의 우선순위를 조율하는 동시에 새로운 API를 소비하는 여러 다운스트림 팀과도 협력해야 합니다.

이 시점에서, 우리는 특정 비즈니스 도메인에 집중하지 않는 스마트 미들웨어를 아키텍처 내에 만들어버린 셈입니다. 이는 많은 사람들이 합리적인 서비스 지향 아키텍처(Service Oriented Architecture)라고 생각하는 방향과는 어긋날 수 있습니다.



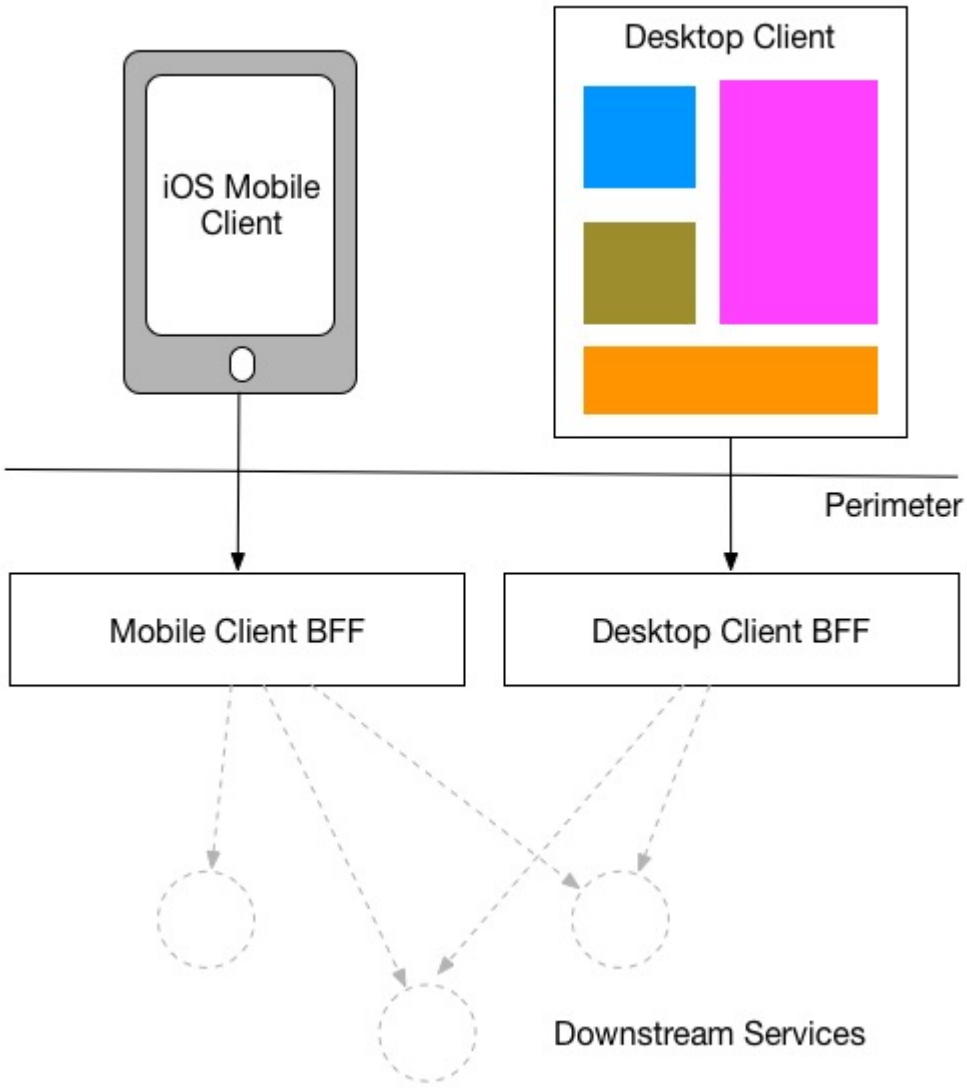
Common Team Structures When Using A Generic Backed API

Introducing The Backend For Frontend

이 문제에 대한 한 가지 해결책으로, REA와 SoundCloud에서 사용된 방법이 있습니다. 범용 API 백엔드를 두는 대신, 각 사용자 경험마다 별도의 백엔드를 두는 방식입니다. 전(前) SoundCloud 개발자인 Phil Calçado는 이를 **Backend For Frontend(BFF)**라고 명명했습니다.

개념적으로 보면, 사용자 중심 애플리케이션은 두 개의 구성 요소로 이루어져 있다고 생각할 수 있습니다. 하나는 경계(perimeter) 바깥에 위치하는 클라이언트 사이드 애플리케이션이고, 다른 하나는 경계 안쪽에 위치하는 서버 사이드 구성 요소인 **BFF**입니다.

BFF는 특정한 사용자 경험에 맞춰 긴밀하게 결합(tightly coupled)되어 있으며, 일반적으로 해당 사용자 인터페이스를 담당하는 동일한 팀이 관리합니다. 이를 통해 UI가 필요로 하는 API를 정의하고 조정하는 과정이 쉬워지며, 클라이언트와 서버 구성 요소를 동시에 릴리스하는 과정도 단순해집니다.

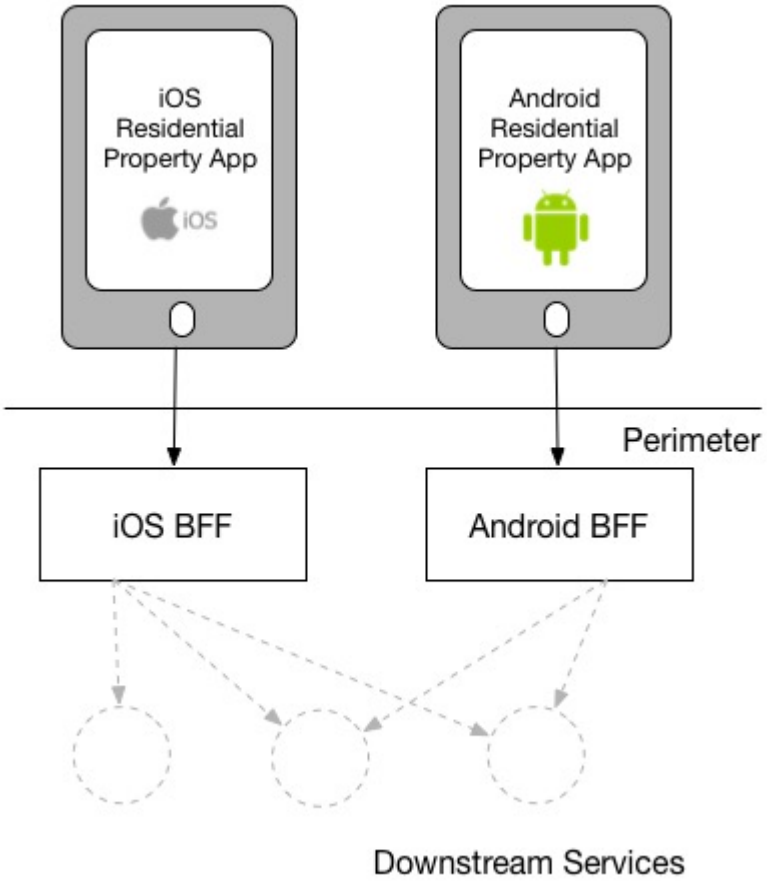


Using one server-side BFF per user interface

How Many BFFs?

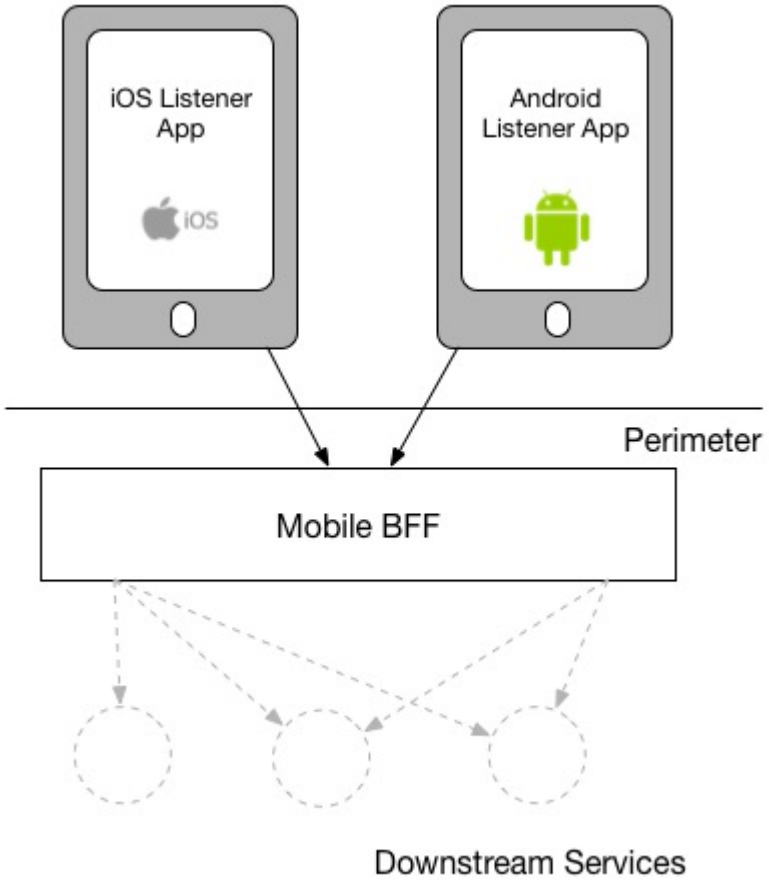
서로 다른 플랫폼에서 동일한(혹은 유사한) 사용자 경험을 제공하는 방식에는 두 가지 접근법이 있습니다.

제가 선호하는 모델은 **각각의 클라이언트 유형마다 하나의 BFF(Backend For Frontend)를 엄격하게 유지하는 방식**입니다. 이 모델은 REA에서 사용된 사례를 본 적이 있습니다.



Different mobile platform, different BFF, as used at REA

다른 모델은 SoundCloud에서 사용된 방식으로, **사용자 인터페이스 유형별로 하나의 BFF를 두는 것**입니다. 즉, Android와 iOS용 네이티브 리스너(listener) 애플리케이션이 동일한 BFF를 사용하는 방식입니다.



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

두 번째 모델에서 제가 가장 우려하는 부분은 **하나의 BFF를 여러 유형의 클라이언트가 사용할수록, 여러 가지 책임을 처리하면서 점점 비대해질 가능성이 커진다는 점**입니다. 하지만 여기서 중요한 것은, **BFF를 공유하는 경우에도 동일한 유형의 사용자 인터페이스를 위한 것이라는 점**입니다. 예를 들어, SoundCloud의 iOS 및 Android용 리스너(listener) 네이티브 애플리케이션은 동일한 BFF를 사용하지만, 다른 네이티브 애플리케이션(예: 신규 크리에이터 애플리케이션 Pulse)은 별도의 BFF를 사용합니다.

또한, 저는 **Android와 iOS 애플리케이션을 동일한 팀이 관리하면서 BFF도 함께 운영하는 경우, 이 모델을 사용하는 것에 대해 비교적 유연한 입장**입니다. 하지만 두 애플리케이션이 서로 다른 팀에서 유지보수된다면, 더 엄격한 모델을 추천하는 편입니다. 결국, **조직 구조가 어떤 모델이 가장 적합한지를 결정하는 핵심 요인이 될 수 있으며, 이는 다시 한 번 Conway의 법칙(Conway's Law)의 영향을 받습니다**.

제가 대화했던 SoundCloud 엔지니어들은 **iOS와 Android 리스너 애플리케이션을 위한 단일 BFF를 사용하는 것이, 지금 다시 결정한다면 재고해볼 부분이라고 언급**하기도 했습니다.

Stewart Gleadow(그는 다시 Phil Calçado와 Mustafa Sezgin을 인용했습니다)의 한 가지 가이드라인이 특히 인상적이었습니다. **"하나의 경험, 하나의 BFF(One experience, one BFF)"**라는 원칙입니다. 즉, iOS와 Android의 사용자 경험이 매우 유사하다면 단일 BFF를 유지하는 것이 더 쉬운 선택이 될 수 있습니다. 하지만 두 플랫폼의 경험이 크게 다르다면, 별도의 BFF를 두는 것이 더 합리적입니다.

Pete Hodgson은 **BFF는 팀의 경계를 기준으로 구성될 때 가장 효과적**이라고 언급했습니다. 즉, 팀 구조에 따라 BFF의 개수가 결정되어야 한다는 것입니다. 예를 들어, **모바일 팀이 하나라면 BFF도 하나가 적절하지만, iOS 팀과 Android 팀이 분리되어 있다면 BFF도 각각 나누는 것이 바람직합니다**.

그러나 **팀 구조는 시스템 설계보다 더 유동적이라는 점이 문제**입니다. 예를 들어, 모바일을 위한 단일 BFF가 있는 상태에서 팀이 iOS와 Android 전담팀으로 분리된다면, **BFF도 분리해야 할까요?** 반면, 처음부터 별도의 BFF

를 두었다면 팀이 분리될 때도 더 쉽게 운영될 것입니다. **BFF 구조와 팀 구조의 상호작용은 매우 중요한 요소이며, 이를 더 깊이 탐구할 예정입니다**.

BFF 개수를 줄이려는 주된 동기는 **서버 사이드 기능을 재사용하고 중복을 최소화하는 것**입니다. 하지만 이를 해결할 다른 방법들도 있으며, 이에 대해서도 곧 다룰 예정입니다.

And Multiple Downstream Services (Microservices!)

BFF는 **백엔드 서비스의 수가 적은 아키텍처에서는 유용한 패턴**이 될 수 있습니다. 하지만 **많은 수의 서비스를 사용하는 조직에서는 필수적**이기도 합니다. 이는 **사용자 기능을 제공하기 위해 여러 다운스트림 호출을 집계해야 할 필요성이 크게 증가하기 때문**입니다.

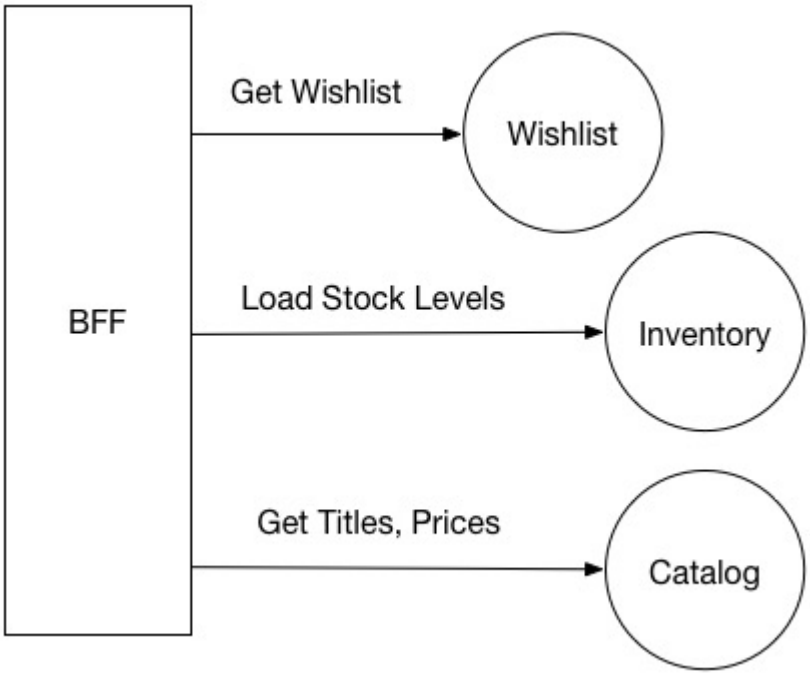
이러한 상황에서는 **BFF에 대한 단일 요청이 여러 개의 마이크로서비스 호출로 이어지는 것이 일반적**입니다. 예를 들어, **전자상거래(e-commerce) 기업을 위한 애플리케이션**을 생각해 보겠습니다. 사용자의 **위시리스트(wish list)**에 있는 상품 목록을 불러와 **재고 수준(stock levels)과 가격(price)을 표시**해야 한다고 가정해봅시다.

The Brakes - Give Blood	In Stock! (14 items re - maining)	$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

우리가 필요한 정보는 여러 개의 서비스에서 관리하고 있습니다.

- **Wishlist 서비스**는 위시리스트의 정보와 각 상품의 ID를 저장합니다.
- **Catalog 서비스**는 각 상품의 이름과 가격 정보를 저장합니다.
- **Inventory 서비스**는 재고 수준(stock levels) 정보를 저장합니다.

따라서, BFF에서는 **전체 위시리스트를 가져오는 메서드를 제공해야 하며, 최소한 3번의 서비스 호출이 필요합니다**.



Making multiple downstream calls to construct a view of a wishlist

효율성 관점에서 보면, 가능한 한 많은 호출을 **병렬(parallel)로 실행하는 것이 훨씬 더 스마트한 방법**입니다.

위시리스트(Wishlist) 서비스에 대한 초기 호출이 완료된 후, **다른 서비스들(Catalog 및 Inventory)에 대한 호출을 동시에 실행하면 전체 응답 시간을 줄일 수 있습니다**. 그러나 **병렬로 실행할 호출과 순차적으로 실행해야 할 호출을 조합하는 것은 관리가 어려울 수 있으며**, 특히 복잡한 시나리오에서는 더욱 그렇습니다.

이러한 문제를 해결하는 데 도움이 되는 것이 **반응형(reactive) 프로그래밍 스타일**입니다. 예를 들어 **RxJava**나 **Finagle의 futures 시스템**을 사용하면 **여러 개의 호출을 조합하는 작업을 더 쉽게 관리**할 수 있습니다.

하지만 **장애(Failure) 처리 방식**을 명확히 이해하는 것도 중요합니다.

위의 예시에서, **모든 다운스트림 호출이 성공적으로 응답해야만 클라이언트에 데이터를 반환하도록 강제하는 것이 과연 올바른 접근일까요?**

- **Wishlist 서비스가 다운된 경우**, 우리는 아무것도 할 수 없습니다.
- 하지만 **Inventory 서비스만 다운된 경우**, 전체 요청을 실패시키기보다는 **재고(stock level) 표시 기능을 제외하고 나머지 데이터를 반환하는 것이 더 나은 선택**일 수 있습니다.

이러한 장애 처리 로직은 **우선적으로 BFF에서 관리해야 하지만**, 동시에 **BFF에 요청을 보내는 클라이언트도 부분 응답(partial response)을 해석하고 올바르게 렌더링할 수 있어야 합니다**.

Reuse and BFFs

사용자 인터페이스마다 단일 BFF를 사용하는 것에 대한 우려 중 하나는 **BFF들 간에 많은 중복이 발생할 수 있다는 점**입니다. 예를 들어, 각 BFF가 동일한 유형의 집합 작업을 수행하거나, 다운스트림 서비스와 상호작용하는 데 있어 유사한 코드가 반복될 수 있습니다. 이러한 중복을 피하려는 일부 사람들은 이를 하나로 통합하려고 하여, **범용 집계(aggregating) Edge API 서비스**를 만들고자 할 수 있습니다. 하지만 이 모델은 **여러 번에 걸쳐 매우 비대해지고 여러 가지 책임이 얽힌 코드**를 초래하는 경우가 많았습니다.

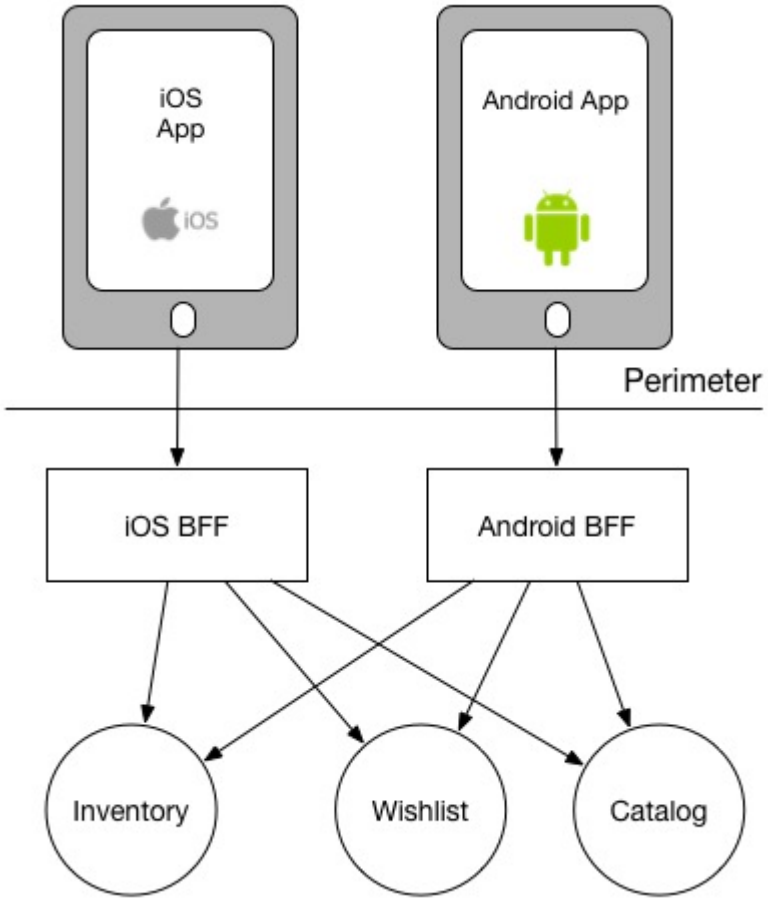
저는 **서비스 간 중복 코드에 대해서는 비교적 유연한 입장**입니다. 즉, **단일 프로세스 경계를 넘지 않는 한 중복을 적절한 추상화로 리팩토링하려고 노력하지만**, **서비스 간 중복에 대해서는 같은 반응을 보이지 않습니다**. 이는 **공유 코드를 추출하는 과정이 서비스 간 긴밀한 결합을 초래할 가능성**을 우려하기 때문입니다. **저는 일반적인 중복보다는 이 결합이 더 큰 문제라고 생각합니다**. 그럼에도 불구하고, **일부 경우에는 중복을 제거하는 것이 타당한 상황**이 있을 수 있습니다.

제 동료인 Pete Hodgson은 **BFF가 없는 경우**, 종종 **'공통' 로직이 각 클라이언트에 내장되게 된다고 지적했습니다.** 다양한 기술 스택을 사용하는 클라이언트들에서는 **이 중복이 발생하는 사실을 인식하는 것이 어려울 수 있습니다**. 그러나 **서버 측 컴포넌트에 공통 기술 스택을 사용하는 조직에서는**, 여러 BFF들 간의 중복을 쉽게 인식하고 이를 분리할 수 있는 경우가 많습니다.

공유 코드를 추출해야 할 때가 온다면, 두 가지 명백한 옵션이 있습니다. 첫 번째는 **공유 라이브러리를 추출하는 방식**인데, 이 방식은 **비용이 적게 들지만 위험이 큰 경우**가 많습니다. 그 이유는 **공유 라이브러리가 결합을 일으킬 수 있기 때문**입니다, 특히 **다운스트림 서비스와 상호작용하기 위한 클라이언트 라이브러리를 생성할 때 더욱 그렇습니다**. 그럼에도 불구하고, **서비스 내에서 추상화해야 할 코드가 매우 특정한 문제일 경우에는 이 방법이 적합할 수 있습니다**.

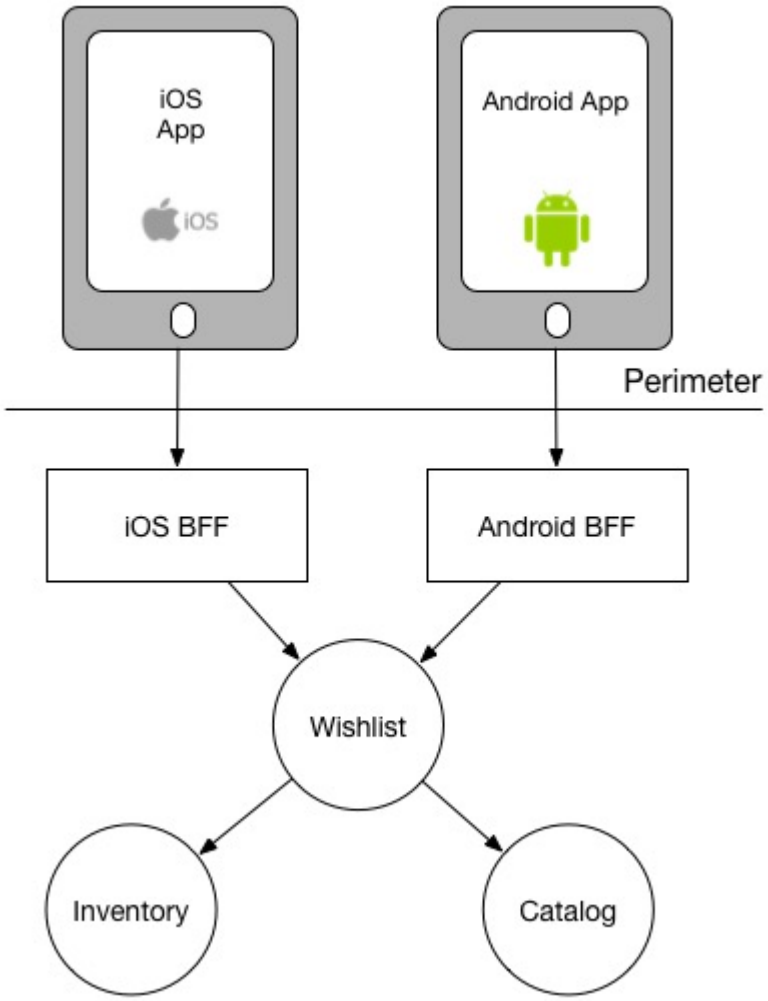
다른 옵션은 **새로운 서비스를 만들어 공유 기능을 추출하는 것**입니다. 이 방식은 **추출된 서비스가 해당 도메인에 맞춰 모델링된 경우** 잘 작동할 수 있습니다.

이 접근 방식의 변형으로는 **집계 책임을 더 내려가 있는 서비스로 위임하는 방법**도 있을 수 있습니다. 예를 들어, **위시리스트를 렌더링하는 예시**에서 위시리스트를 **Android, iOS, Web에서 렌더링한다고 가정해봅시다**. 각 BFF가 동일한 **세 개의 호출**을 한다고 했을 때,



Multiple BFFs performing the same tasks

대신, **Wishlist 서비스**가 **다운스트림 호출을 대신 처리하도록 변경**함으로써, **호출하는 측의 작업을 간소화**할 수 있습니다.



Pushing aggregation duties further downstream to remove duplication in BFFs

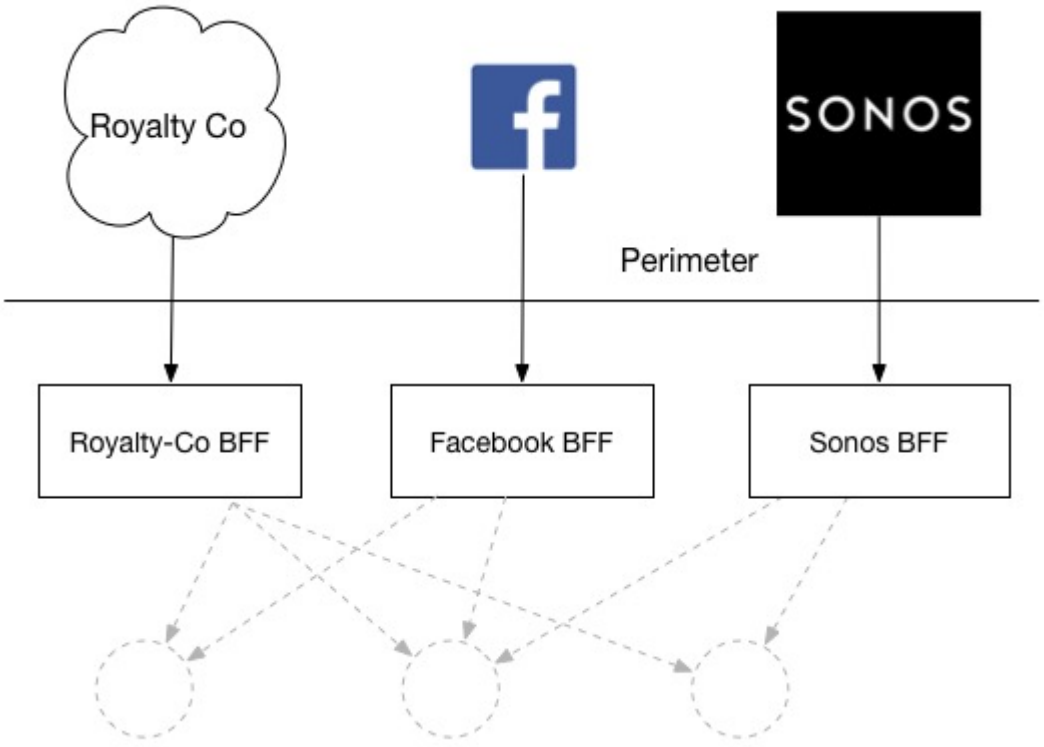
저는 **동일한 코드가 두 군데에서 사용된다고 해서** 꼭 **새로운 서비스를 추출하고자 하는 마음이 생기지는 않습니다**. 하지만 **새 서비스를 만드는 거래 비용이 충분히 낮거나, 여러 군데에서 그 서비스를 사용하게 된다면** (예: 데스크탑 웹에서) **서비스를 추출하는 것을 고려할 수 있습니다**. 저는 여전히 **무언가를 세 번째로 구현할 때 추상화를 고려하라는 예전의 격언**이 **서비스 레벨에서도 좋은 규칙**이 된다고 생각합니다.

BFFs for Desktop Web and Beyond

BFF는 **모바일 장치의 제약을 해결하는 데 유용한 도구**로 생각할 수 있습니다. 데스크탑 웹 경험은 일반적으로 더 강력한 장치와 더 나은 연결성을 갖춘 환경에서 제공되며, 이 경우 여러 개의 다운스트림 호출을 하는 비용이 관리 가능합니다. 이러한 경우에는 웹 애플리케이션이 BFF 없이 **직접 다운스트림 서비스에 여러 번의 호출을 하는 것이 가능**합니다.

하지만 **웹에서도 BFF를 사용하는 것이 유용할 수 있는 경우**도 있습니다. 예를 들어, **서버 측 템플릿을 사용하여 웹 UI의 큰 부분을 생성할 때**, BFF는 이를 처리하는 **자연스러운 장소**가 됩니다. 또한 **캐싱을 단순화**하는 데도 도움이 될 수 있습니다. 왜냐하면 BFF 앞에 리버스 프록시를 배치하여 **집계된 호출의 결과를 캐시할 수 있기 때문**입니다. (단, 집계된 콘텐츠의 만료 기간을 적절히 설정하여 **가장 신선한 콘텐츠**가 만료되는 시간에 맞추어야 합니다.) 실제로 **BFF라고 부르지 않고 사용되는 경우도 많았으며, 범용 API 백엔드**도 종종 이러한 방식에서 발전하게 됩니다.

적어도 한 조직에서는 **외부 파트너가 호출을 해야 하는 경우에도 BFF를 사용**하는 모습을 본 적이 있습니다. 다시 말해, **음악 상점**의 예시를 가져와 보면, **3자에게 로열티 정보 추출, Facebook 통합, 또는 다양한 셋톱박스 장치에 대한 스트리밍**을 제공하기 위해 BFF를 노출하는 방식입니다.



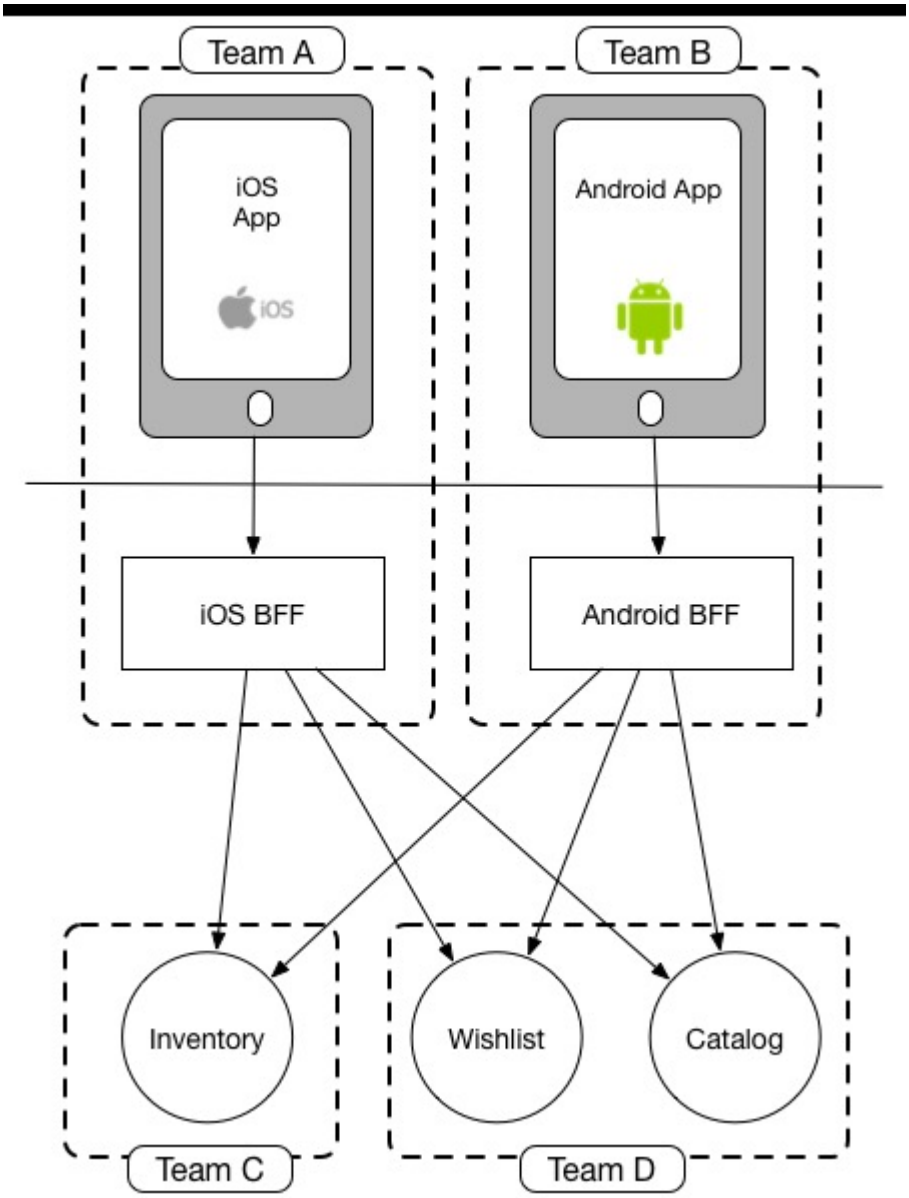
Exposing APIs to 3rd Parties using a BFF

이 접근 방식은 **외부 파트너가 자신이 사용하는 API 호출을 변경하거나 사용할 능력(또는 의지)이 제한적일 때** 특히 효과적일 수 있습니다. **범용 API 백엔드**의 경우, **외부 파트너 중 일부가 변경을 할 수 없거나 원하지 않아서 이전 버전의 API를 유지해야 할 수도 있습니다**. 하지만 **BFF**를 사용하면 이러한 문제는 **상당히 줄어듭니다**.

And Autonomy

자주 발생하는 상황은 **하나의 팀이 프론트엔드를 작업하고, 다른 팀은 백엔드 서비스를 만드는 경우**입니다. 일반적으로 **비즈니스 수직에 맞춰 마이크로서비스로 전환**하여 이를 피하려고 노력하지만, **이 경우에도 이를 피하기 어려운 상황**이 존재합니다. 첫째, **규모나 복잡성의 일정 수준에 도달하면 여러 팀이 참여해야 할 필요**가 있습니다. 둘째, **좋은 Android 또는 iOS 경험을 구현하려면 전문화된 팀의 기술이 필요**한 경우가 많습니다.

그래서 **사용자 인터페이스를 만드는 팀은, 다른 팀이 관리하는 API를 호출하는 상황에 직면하게 됩니다**, 그리고 종종 **이 API는 사용자 인터페이스가 개발되는 동안 계속 진화**합니다. 이때 **BFF**가 도움이 될 수 있습니다. 특히 **BFF를 사용자 인터페이스를 만드는 팀이 소유하는 경우**, **프론트엔드를 만들면서 동시에 BFF의 API를 발전**시킬 수 있습니다. 이렇게 하면 **두 가지를 모두 빠르게 반복**할 수 있습니다. **BFF 자체는 여전히 다른 다운스트림 서비스들을 호출해야 하지만**, **사용자 인터페이스 개발에 방해가 되지 않도록** 이를 수행할 수 있습니다.

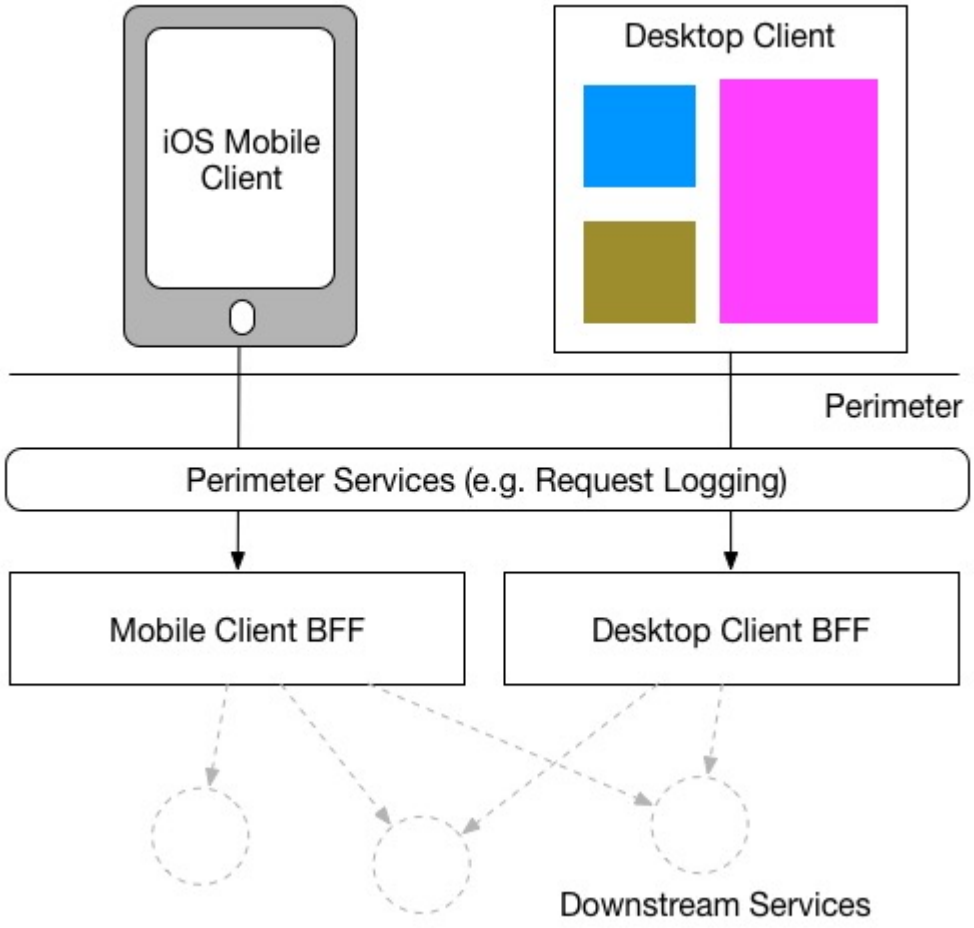


Example team ownership boundaries when using BFFs

이렇게 **팀 경계에 맞춰 BFF를 사용하는** 또 다른 장점은 **인터페이스를 만드는 팀이 기능이 어디에 위치할지에 대해 훨씬 더 유연하게 생각할 수 있다는 점**입니다. 예를 들어, **기능을 서버 측으로 옮겨서 향후 재사용을 촉진하고 네이티브 모바일 애플리케이션을 간소화**하거나, **새로운 기능을 더 빠르게 출시할 수 있도록 (앱 스토어 검토 절차를 우회하여)** 할 수 있습니다. 이러한 **결정은 팀이 모바일 애플리케이션과 BFF를 모두 소유하고 있다면** 팀 단독으로 내릴 수 있으며, **다른 팀과의 조정이 필요하지 않습니다**.

General Perimeter Concerns

일부 사람들은 **BFF를 사용하여 인증/인가 또는 요청 로깅**과 같은 **일반적인 경계 문제를 구현**합니다. 이에 대해 저는 다소 고민이 있습니다. 한편으로는, **이 기능이 너무 일반적이기 때문에 이를 별도의 계층으로 구현하는 것이 더 적합**하다고 생각합니다. 예를 들어, **Nginx**나 **Apache 서버**와 같은 더 상위 계층을 사용하는 방식입니다. 반면, **이러한 추가 계층은 지연(latency)을 불가피하게 증가시킬 수 있습니다**. BFF는 **마이크로서비스 환경**에서 자주 사용되며, 이 환경에서는 **많은 네트워크 호출로 인해 지연에 매우 민감**해집니다. 또한, **프로덕션처럼 운영되는 스택을 만들기 위해 배포해야 하는 계층이 많을수록 개발 및 테스트가 더 복잡해질 수 있습니다**. 이런 이유로, **모든 이러한 문제를 BFF 내부에서 더 자립적인 해결책으로 처리하는 것이 매력적일 수 있습니다**.



Using a network appliance to implement generic perimeter concerns

앞서 논의한 것처럼, 이 중복을 해결하는 또 다른 방법은 **공유 라이브러리**를 사용하는 것입니다. **BFF들이 동일한 기술을 사용한다고 가정하면**, 이는 그리 어렵지 않을 것입니다. 다만, **마이크로서비스 아키텍처에서 공유 라이브러리를 사용할 때의 일반적인 주의사항**은 여전히 적용됩니다.

When To Use

웹 UI만 제공하는 애플리케이션의 경우, **서버 측에서 상당한 양의 집합이 필요할 때**에만 BFF가 의미가 있을 것이라고 생각합니다. 그렇지 않으면 **추가적인 서버 측 구성 요소 없이도 다른 UI 구성 기법들이 잘 작동할 수 있습니다** (이 부분에 대해 곧 이야기할 수 있기를 바랍니다).

하지만 **모바일 UI나 제3자에게 특정 기능을 제공해야 할 때**는 **처음부터 각 당사자를 위한 BFF를 사용하는 것**을 강력히 고려할 것입니다. **추가 서비스 배포 비용이 높다면 재고할 수도 있겠지만**, **BFF가 가져오는 관심사의 분리**는 대부분의 경우 꽤 설득력 있는 제안입니다. **UI를 만드는 사람들과 다운스트림 서비스 간에 큰 분리가 있을 때**는 위에서 설명한 이유들로 **BFF를 사용하는 것에 더 끌리게 됩니다**.

Conclusion

는 마이크로서비스를 사용할 때 모바일 개발에서 중요한 문제를 해결합니다. 또한 **일반적인 API 백엔드에 대한 매력적인 대안**을 제공하며, 많은 팀들이 **모바일 개발 외의 다른 목적으로도 BFF를 사용**합니다. BFF는 **지원하는 소비자 수를 제한**하는 간단한 작업만으로도 훨씬 더 쉽게 작업하고 변경할 수 있게 하며, **고객-facing 애플리케이션을 개발하는 팀들이 더 많은 자율성을 유지할 수 있도록 돕습니다**.

이 기사를 연구하는 데 도움을 준 Matthias Käppler, Michael England, Phil Calçado, Lukasz Plotnicki, Jon Eaves, Stewart Gleadow, Kristof Adriaenssens에게 감사드리며, Giles Alexander, Ken McCormack, Sriram Viswanathan, Kornelis Sietsma, Hany Elemary, Martin Fowler, Vladimir Sneblic, Pete Hodgson에게도 일반적인 피드백에 감사드립니다. 추가 피드백도 정말 감사드리며, 아래에 댓글을 남겨주세요!

from : https://samnewman.io/patterns/architectural/bff/