

Building a Backend for Frontend (BFF) For Your Microservices

(<https://nordicapis.com/building-a-backend-for-frontend-shim-for-your-microservices/>)



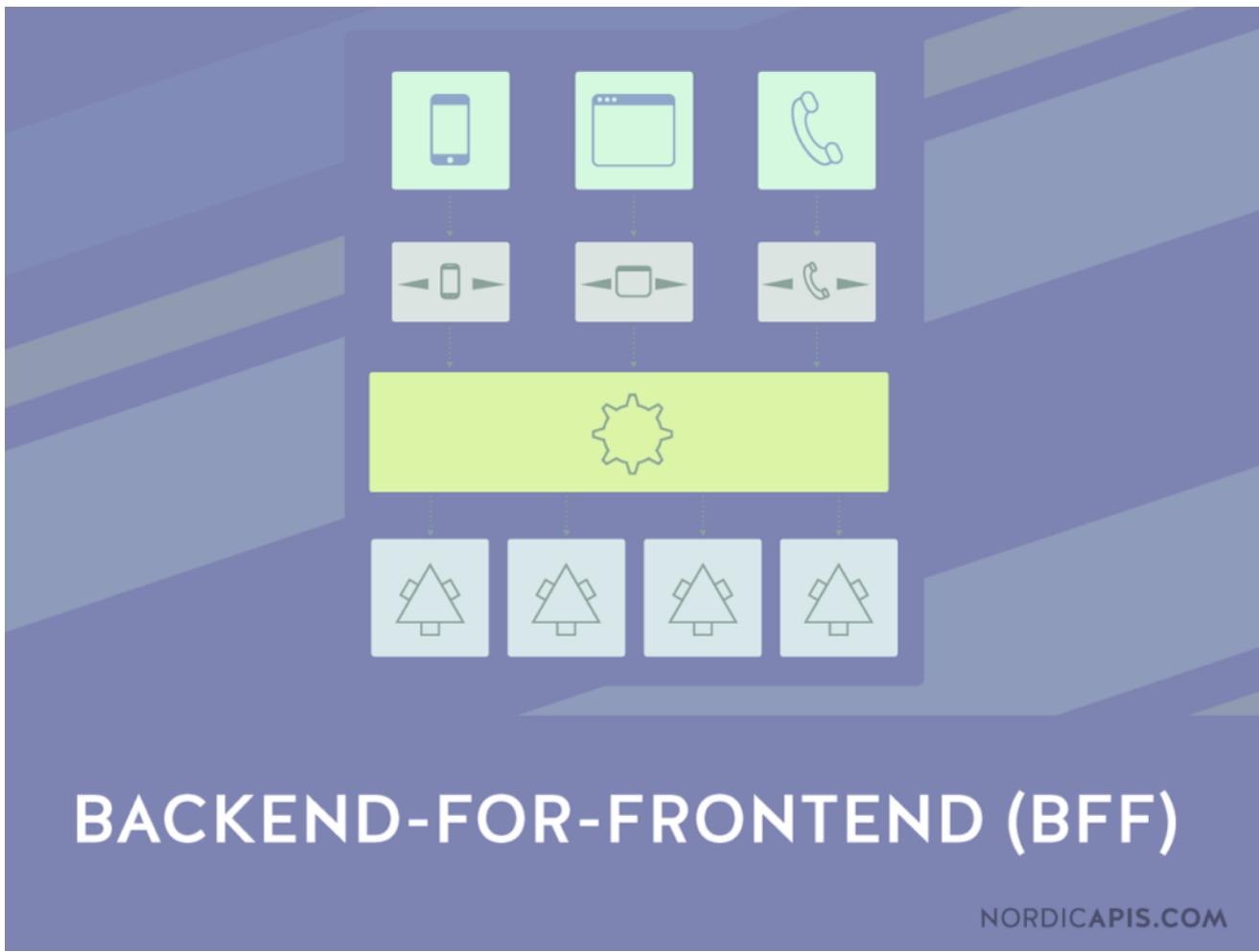
Kristopher Sandoval(<https://nordicapis.com/author/sandovaleffect/>)

May 19, 2017

 (<https://www.facebook.com/sharer.php?u=https%3A%2F%2Fnordicapis.com%2Fbuilding-a-backend-for-frontend-shim-for-your-microservices%2F>)

 (<https://twitter.com/intent/tweet?url=https%3A%2F%2Fnordicapis.com%2Fbuilding-a-backend-for-frontend-shim-for-your-microservices%2F&text=Building+a+Backend+for+Frontend+%28BFF%29+For+Your+Microservices>)

 (<https://www.linkedin.com/s/in/backend-for-frontend-shim-for-your-microservices%2F&title=Building>)



APIs are, by design, vastly interconnected. In order to effectively function and carry out the purposes for their creation, APIs are crafted to connect easily to other services, both internal and external. While this certainly is beneficial from a user perspective, developers often find the result of this type of development to have some problematic elements.

Chief of these elements is that when **microservices** become more numerous, they become **more reliant on each other**. Each new microservice introduced into the ecosystem has its own methods of communication and work, and must be designed specifically to interact with every other dependent service.

Some strategies can be adopted to negate much of the negative points of this aspect of microservice architecture design. One method is to create a **Backend for Frontend (BFF)** shim to help organize microservice architectures and coordinate functionality across a diverse, wide system.

Today, we're going to define exactly what a Backend for Frontend shim is, and how it does what it does. We'll highlight some best practices to keep in mind while developing, and discuss some potential points of failure that must be carefully avoided.

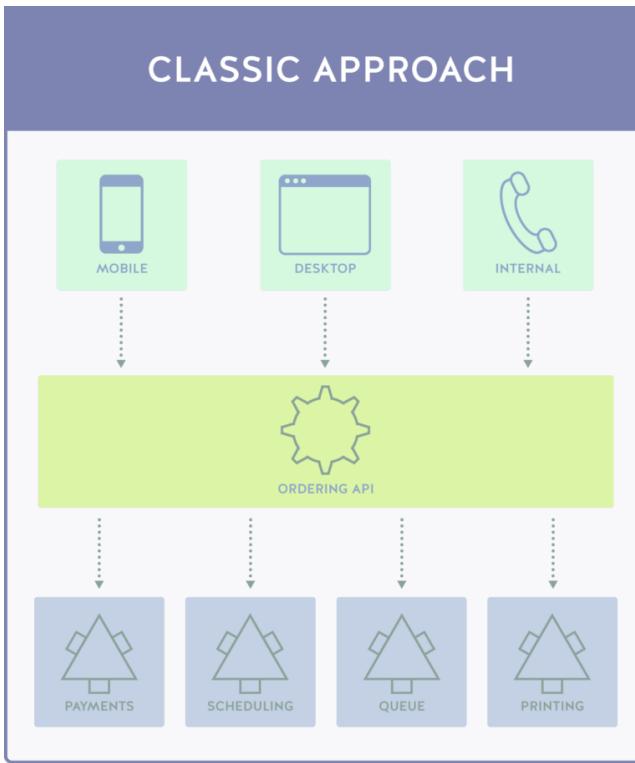
What is a Backend for Frontend?

APIs are essentially a vast number of parts functioning in concert with one another toward a single goal. But the API designer often creates systems that, over time, reveal cracks or weaknesses that need to have a small piece of code to align everything the way it should be. The **Backend for Frontend design pattern**, as described by Phil Calçado (http://philcalcado.com/2015/09/18/the_back_end_for_front_end_pattern_bff.html), refers to the concept of developing niche backends for each user experience.

A Backend for Frontend is a unique type of shim that fills a design gap that is inherent in the API process. A BFF is, in simple terms, a **layer** between the user experience and the resources it calls on. When a mobile user requests data, in a BFF situation, their request is translated through the BFF and into a general layer below it. What this functionally means can be demonstrated in the following example.

3D Printing: Classic Approach

Let's say you have a 3D printing business that takes orders and prints material on demand. In the traditional design, you would be looking at the following API flow:



This type of architecture has several user types, each with different user experiences and API needs. The mobile user utilizes an application, the desktop user uses a web client, and internal orders are placed by customers calling into support, who manually adds the orders to the queue using an internal program.

Functionally, this is done by separating each of the functionalities into an application or web service, each of which call into a general API. The general API has instances of each object and call as needed by each service, and responds to the specific call type (which is denoted via a source entry) with the data required.

While this is certainly useable, it has some serious issues. Prime of course is the **massive size** of the Ordering API that is being called. The Ordering API not only has to contain all of the processes for the desktop and internal solutions, but has to handle each call from the mobile application with their own specific call functions, security systems, and media functionalities.

That's not to say the combination of calls is necessarily the problem – the problem is specifically the nature of how these calls are handled. A single, large API navigating each call and calling to the same monolithic API is problematic and slow. Data needs to be presented in a specific format and language for the mobile solution, which is entirely separated from the desktop solution, resulting in multiple duplicate functions with slightly modified "tweaks".

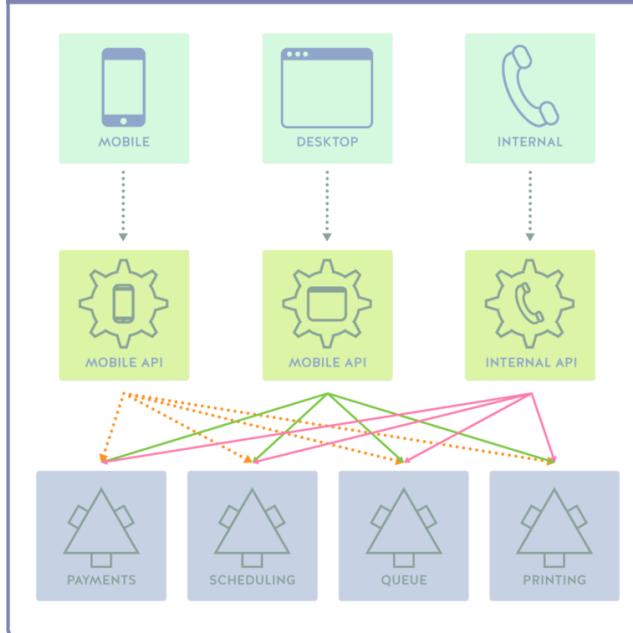
Another big problem is the fact that a single monolithic team is going to be responsible for the user experience of three diverse users, each with their own specific needs and experiences. Clumping everything together like this is messy, and has the effect of a **bloated** internal team.

Related: Asynchronous APIs in Choreographed Microservices (<http://nordicapis.com/asynchronous-apis-in-choreographed-microservices/>)

3D Printing: Microservices Design

We can solve many of these issues through the following change in flow:

MICROSERVICES DESIGN



In this design, each application client type has its own **dedicated API**, effectively making a series of microservices specifically tailored to each user experience, unlike the first example, which relied on a general API. In this case, each call is pushed through a specific, isolated API, and then taps into the resources on the backend. So far so good, right? Not so fast. While many of the problems have been resolved, an entirely new system of issues have arisen.

The big issue here is **microservice continuity**. While we've separated a lot of functionality out in meaningful ways according to the user experience that will use them, we have three disparate APIs calling the same backend functions, which means a lot of **duplication** with code and functionality. Most importantly, as the API is updated and evolved, this will mean **greater overhead** both in terms of team economic and time cost and in the size and speed of the API.

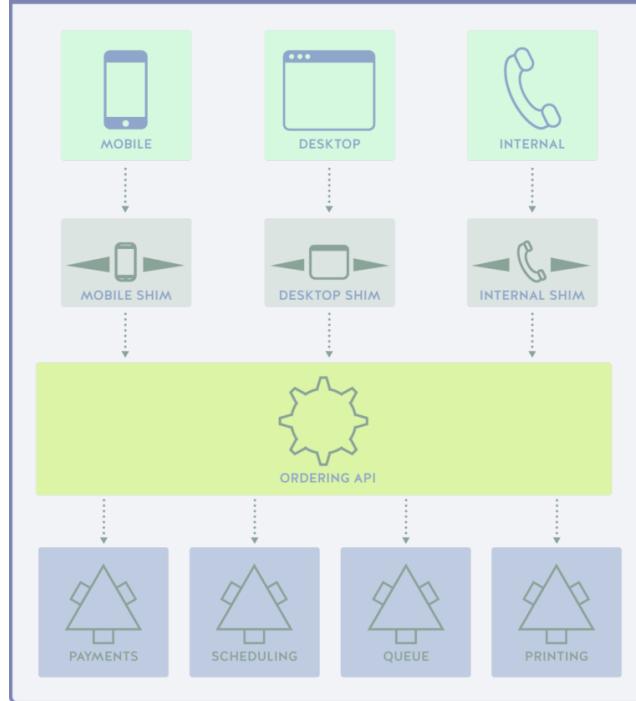
We've traded one world of extremes for another. This is, in many ways, the current API landscape for most providers. Due to this, a wide variety of strategies have been used. One of them, and the one we're discussing today, is the idea of a Backend for Frontend shim.

Also read: 4 Design Tweaks to Improve API Operations (<http://nordicapis.com/4-design-tweaks-to-improve-api-operations/>)

3D Printing: Backend for Frontend

What we're conceptually talking about is a “translator” layer, a shim that takes disparate calls and allows it to be converted into a common form. While we're adding a lot of conceptual layers in this example, do keep in mind the “shims” are rather small pieces of code, much smaller than the API as a whole.

BACKEND-FOR-FRONTEND



While this kind of design might look like a carbon copy rebranding of the microservices architecture, what you're actually seeing here is a layer of **translation services**. Each shim is maintained in concert with another by a "shim team", with the Ordering API being managed by a single API team. Shouldn't this cause the same problems we saw earlier?

The key difference here is in the handling of calls. In the "General API" design above, we had a single API containing variations on the initial data set triggered by source identifications. The problem was that each source had to have its own specialty call, and the data had to be manipulated by the API before being passed along.

In the microservices solution, we got around this by having individual APIs for each service, but problems persisted in the form of lacking continuity and the resultant team scope and size creep that followed.

In the shim approach, what we have is a group of disparate calls being translated into a single call. The different data type translations and systems are done during data handling from the API to the shim, essentially translating the *service specific call into a general call**.

Not only does this alleviate the code base size and scope issues of both the general API and the teams that develop them, this also shifts processing power and responsibility to multiple services which can be designed and maintained by a single shim team.

We're essentially creating a highway, and avoiding all of the surface and toll roads – which means better, **more efficient throughput**.

Best Practices

With all of this in mind, let's discuss some best practices and general pitfalls to avoid when developing this kind of system.

Avoid Shim-to-API Conversion

One of the largest pitfalls a developer can fall into is to think of a shim in the same way one would think of a microservice. These two things are vastly different, and need to be coded as such.

When we talk about a translation shim, what we're really talking about it something that simply states "when this data is returned, transform it from this type of data into the data type specified by the application." We're not talking about a self-contained, all inclusive API, we're talking about a translation layer.

Shim Duplication

The entire point of creating a shim is for increased efficiency in the code base and to eliminate duplication, so it might seem strange that this is a pitfall to watch for. The reality is, however, that those new to shim development might consider a shim necessary for every unique user experience. But how granular should you be?

The missing piece here is that each user experience should be grouped by **experience type**, not by the source of the experience. For instance, a developer may want to develop separate shims for iOS, Android, Windows Mobile, and 3rd Party Mobile operating systems, thinking this would result in better performance.

The truth is that most of these mobile experiences can accept certain common data types, as the user experience is so similar. The application experience between operating systems is alike enough to not warrant their own individual shims.

Over Reliance on Shims

Related to the idea of Shim-to-API conversion, there is a definite threat of a shim becoming too much of a crutch. While it's very tempting to use the shim concept as a front-line for security, for extended functionality, and more, a shim is only meant to be a shim – it's not meant to serve as anything more than a layer of translation.

As such, try to avoid over-relying on the shim. The main API codebase should still handle all functionality and should still do all the things a properly API should do – the shim is supposed to fill a gap, not add functionality and services.

Provide What is Requested

As part of the BFF/shim development concept, each experience will only require certain items of data. Developers might consider the shim to be a **filter** as well as a **translator**, but that's not what we want – the API should only provide the data requested by the shim.

Imagine a shopping front that catalogues items, and collates them with data including manufacturing date, type, frequency of purchase, etc. The mobile application might only need 10 of the 50 data objects that are contained in each entry, whereas the desktop application will need 40 of them, and the internal application will need all 50.

It's tempting to have the shim itself strip the unneeded data out in transit, but this starts to push it towards being a functional API instead of a shim. This type of **data handling should be done on the API level itself**, using source triggers. Each shim should notify the API of the client type, and the API should respond to only requests for specific data sets, rather than supplying an entire data object.

This is specifically done as an approach to the initial code base – objects should be separated with the concept of shims and disparate user experiences in mind. Doing so will increase overall speed, reduce bandwidth, and increase experience satisfaction.

Conclusion

What's important to keep in mind throughout all of this is that a shim is simply one solution to an almost universal architectural problem. While there are indeed other strategies, a shim is very effective when faced with a complex backend and a widely varying range of user experiences. To summarize the types of architectures we've covered:

- In the **classic design**, we have a *mobile call*, a *desktop call*, and an *internal call*. All three of these calls require their own entries or at least a code set to handle conversion inside the General API.
- In the **microservices design**, we have a *mobile call*, a *desktop call*, and an *internal call*. These calls individually interact with their own APIs, the *Mobile API*, the *Desktop API*, and the *Internal API*.
- In the **BFF shim design**. We have a *mobile call*, a *desktop call*, and an *internal call*. These calls are each intercepted by the *mobile shim*, the *desktop shim*, and the *internal shim*. The shim converts the individual calls into a *general call*, which is then returned and translated by the *calling shim*, that is the shim that made the request.

Whether or not it can be properly implemented is an answer subject to the specific use case, backend architecture, and business approach for each individual service. That being said, a BFF can be incredibly powerful when used as it is meant to be used – a translative layer between disparate API services and experiences.

🔗 api (<https://nordicapis.com/tag/api/>), API design (<https://nordicapis.com/tag/api-design/>), APIs (<https://nordicapis.com/tag/apis/>), approach (<https://nordicapis.com/tag/approach/>), architecture (<https://nordicapis.com/tag/architecture/>), backend (<https://nordicapis.com/tag/backend/>), Backend-for-Frontend (<https://nordicapis.com/tag/backend-for-frontend/>), BFF (<https://nordicapis.com/tag/bff/>), Design (<https://nordicapis.com/tag/design/>), frontend (<https://nordicapis.com/tag/frontend/>), layer (<https://nordicapis.com/tag/layer/>), microservice (<https://nordicapis.com/tag/microservice/>), microservices (<https://nordicapis.com/tag/microservices/>), shim (<https://nordicapis.com/tag/shim/>), theory (<https://nordicapis.com/tag/theory/>), translation (<https://nordicapis.com/tag/translation/>), translative (<https://nordicapis.com/tag/translative/>)

4 Comments (https://nordicapis.com/building-a-backend-for-frontend-shim-for-your-microservices/#disqus_thread)



Kristopher Sandoval

(<https://nordicapis.com/author/sandovaleffect/>)

Kristopher is a web developer and author who writes on security and business. He has been writing articles for Nordic APIs since 2015.

 (<https://www.linkedin.com/in/kristophersandoval/>)

⟵ 10+ Awesome Tools and... (<https://nordicapis.com/10-awesome-tools-and-extensions-for-graphql-apis/>)

The Role of APIs in Genomics ⟶ (<https://nordicapis.com/the-role-of-apis-in-genomics/>)

[Recommend 3](#) [Tweet](#) [Share](#)

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name

**Dušan Dimitrić** • 4 years ago

Great post but the concept is a little too ephemeral to (mentally) grasp. Some more concrete examples would be great.

3 ⤵ | ⤴ • Reply • Share >

**Anshul Wagadre** • 3 years ago

Angular Services might be an example of this architecture

1 ⤵ | ⤴ • Reply • Share >

**Ricardo G** • a year ago

Forgot to mention the biggest reason, the jwt orchestrator

^ | ⤴ • Reply • Share >

**Jia Hau Wong** • 2 years ago

Nice

^ | ⤴ • Reply • Share >

[Subscribe](#) [Add Disqus to your site](#) [Add Disqus](#) [Do Not Sell My Data](#)

Latest Posts

Overview of APICheck

Kristopher Sandoval

[\(https://nordicapis.com/overview-of-apicheck/\)](https://nordicapis.com/overview-of-apicheck/)

March 25, 2021

13 Important Metrics for API Companies

Derrick Gilling

[\(https://nordicapis.com/13-important-metrics-for-api-companies/\)](https://nordicapis.com/13-important-metrics-for-api-companies/)

March 24, 2021

How Service Mesh Prevents a Microservices Mess

Art Anthony

[\(https://nordicapis.com/how-service-mesh-prevents-a-microservices-mess/\)](https://nordicapis.com/how-service-mesh-prevents-a-microservices-mess/)

March 23, 2021



API-AS-A-PRODUCT

Creating an API-centric SaaS business

Wednesday, April 28, 2021

(<https://nordicapis.com/events/livecast-api-as-a-product/>)

(<https://youtu.be/2Ud0JwC3JNl>)



SECURING OPEN BANKING

WATCH NOW



Love APIs?

Contribute your thoughts to our blog!



(<https://nordicapis.com/create-with-us/>)

CALL FOR SPEAKERS

DO YOU HAVE A COOL PROJECT
TO SHARE?



SUBMIT A TALK!

(<https://nordicapis.com/call-speakers/>)

Smarter Tech Decisions Using APIs



High impact blog posts and eBooks on API business models, and tech advice



Connect with market leading platform creators at our events



Join a helpful community of API practitioners

API Insights Straight to Your Inbox!

Can't make it to the event? Signup to the Nordic APIs newsletter for quality content. High impact blog posts on API business models and tech advice.

tim@apple.com

Subscribe

Join Our Thriving Community

Become a part of the world's largest community of API practitioners and enthusiasts. Share your insights on the blog, speak at an event or exhibit at our conferences and create new business relationships with decision makers and top influencers responsible for API solutions.



Write

(<https://nordicapis.com/create-with-us/>)

Speak

(<https://nordicapis.com/call-speakers/>)

Sponsor

(<https://nordicapis.com/about/contact-us/>)

Events

Best Public API of 2020 (<https://nordicapis.com/best-public-api-of-2020/>)

Platform Summit 2020 (<https://nordicapis.com/events/platform-summit-2020/>)

Austin API Summit 2020 (<https://nordicapis.com/events/austin-api-summit-2020/>)

Blog

Blog (/blog)

Business Models (<https://nordicapis.com/category/business-models/>)

Marketing (<https://nordicapis.com/category/marketing/>)

Platforms (<https://nordicapis.com/category/platforms/>)

Security (<https://nordicapis.com/category/security/>)

Strategy (<https://nordicapis.com/category/strategy/>)

Design (<https://nordicapis.com/category/design/>)

Resources

eBooks (/api-ebooks/)

Blog Submission Guidelines (<https://nordicapis.com/create-with-us/>)

Call for Speakers (<https://nordicapis.com/call-speakers/>)

Code of Conduct (<https://nordicapis.com/code-of-conduct/>)

About

About (<https://nordicapis.com/about/>)

Press (<https://nordicapis.com/about/press/>)

Privacy Policy (<https://nordicapis.com/nordic-apis-privacy-policy/>)

Volunteer (<https://nordicapis.com/student-volunteer/>)

Social



(<https://twitter.com/nordicapis>)



(<https://www.linkedin.com/company/nordic-apis>)



(<https://www.facebook.com/NordicAPIs>)



(<https://www.youtube.com/user/nordicapis>)

© 2013-2021 Nordic APIs AB | Supported by  CURITY (<https://curity.io>) | Website policies (/policies/)