Get unlimited access to the best of Medium for less than $1/week. Become a member

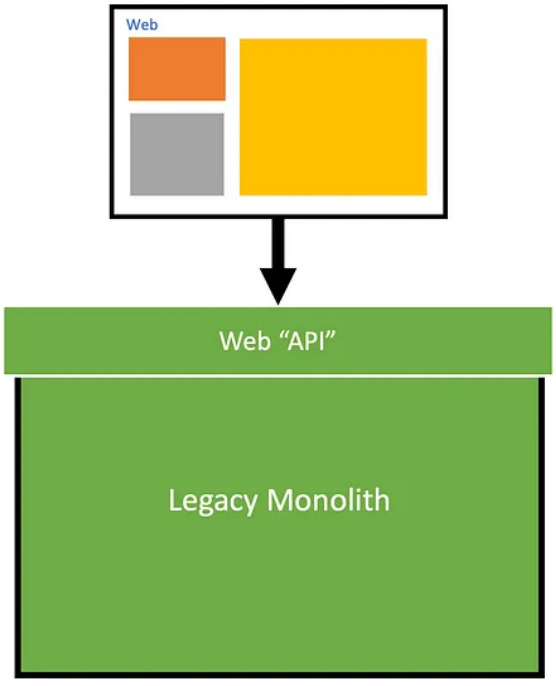
MobilePeople

Backend for frontend (BFF) pattern — why do you need to know it?


Michael Szczepanik Follow 6 min read · Oct 25, 2021

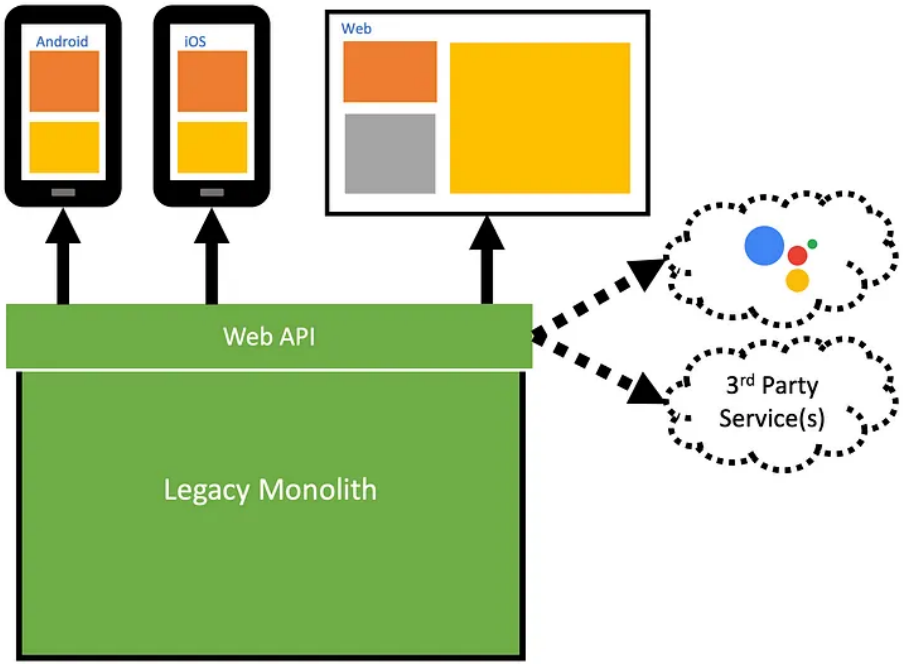
1K 7

Our typical issue starts when we need to integrate some API to our mobile app. Let's imagine the case when you need to create a mobile app for an existing system. The system was one monolithic solution that exposed an API serving only the web client.



The existing system

The idea from the client is not only limited to new mobile apps, but he also considers voice assistants and 3rd party services which will use our API. So the issue is that one API needs to support all these types of clients, and we need to take care of their requirements and maintenance.

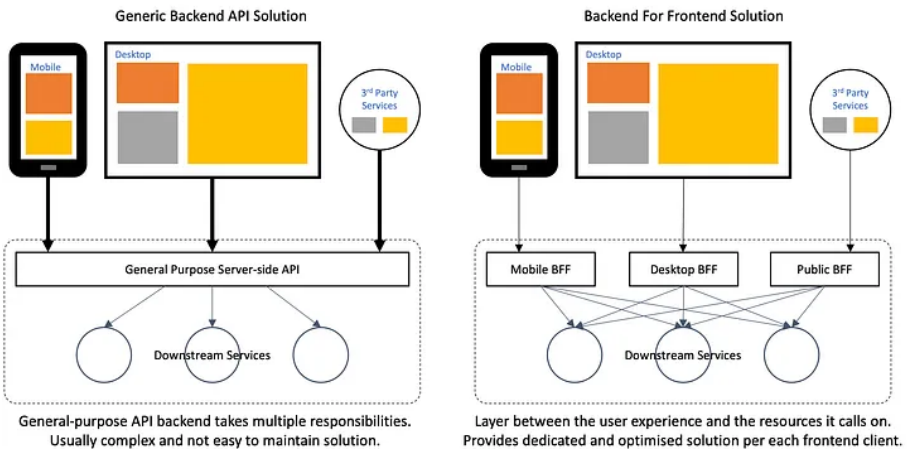


The target (?)

The solution which can help in this case is the **Backend For Frontend** pattern.

Backend for Frontend (BFF) design pattern

You need to think of the user-facing application as being two components — a client-side application living outside your perimeter and a server-side component (BFF) inside your perimeter. BFF is a variant of the **API Gateway** pattern, but it also provides an additional layer between microservices and each client type separately. Instead of a single point of entry, it introduces multiple gateways. Because of that, you can have a tailored API that targets the needs of each client (mobile, web, desktop, voice assistant, etc.), and remove a lot of the bloat caused by keeping it all in one place. The below image describes how it works.

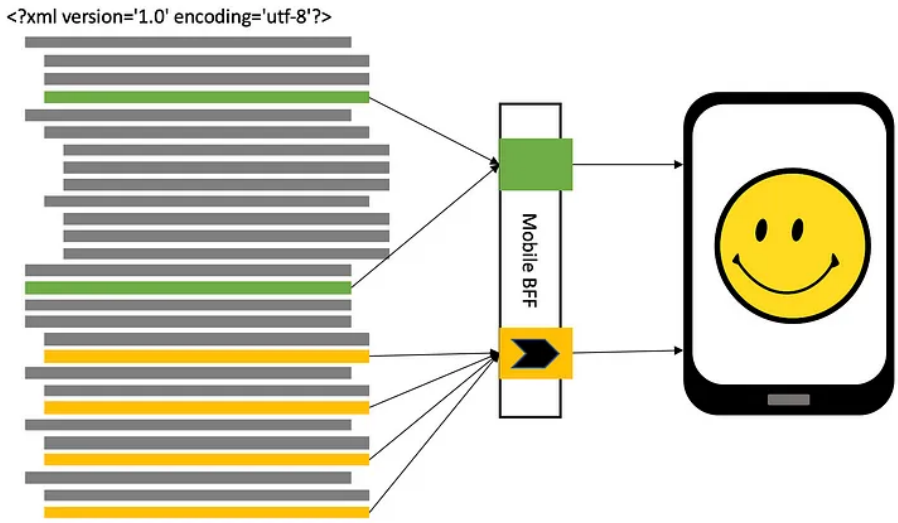


General-purpose API vs BFF

How to create the best API for your BFF

Let's back to our client and his Web API, which can be a legacy XML-based

solution in which only a few parts of shared data are important from a mobile perspective. Using BFF, we can extract important data and convert them to a better format (for example, JSON). It is also an excellent time to create endpoints dedicated to specific app screens or features and data required by them. For example, the below image shows us that green and yellow parts can be extracted separately from legacy API because different mobile app screens need them.



BFF extracts only required data from the main API and converted them to two new mobile BFF endpoints

Here you can find some recommendations from me for that step:

- Focus on UI/UX and data needed by them.
- Don't try to make everything generic from the beginning; this may cause that component to be used across organizations and many people will want to contribute.
- Use *particular feature first over generic usage* strategy. This is the best approach to keep clean API dedicated to one client.
- Use the gold rule of three.

Why BFF?

Decoupling of Backend and Frontend for sure gives us faster time to market as frontend teams can have dedicated backend teams serving their unique needs. The release of new features of one frontend does not affect the other.

We can much easier maintain and modify APIs and even provide API versioning dedicated for specific frontend, which is a big plus from a mobile app perspective as many users do not update the app immediately.

Top highlight

Simplify the system from Frontend and Backend perspectives as we don't need any compromise.

The BFF can benefit from hiding unnecessary or sensitive data before

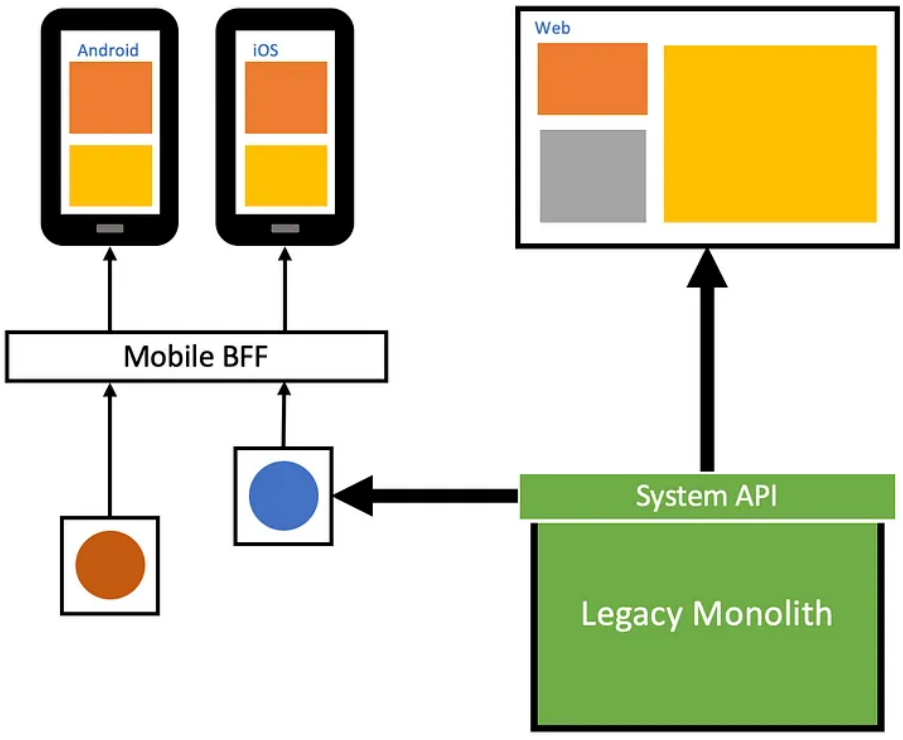
transferring it to the frontend application interface, so keys and tokens for 3rd party services can be stored and used from BFF.

Allows to send formatted data to frontend and because of that can minimalize logic on it.

Additionally, give us possibilities for performance improvements and good optimization for mobile.

How to introduce BFF into the project

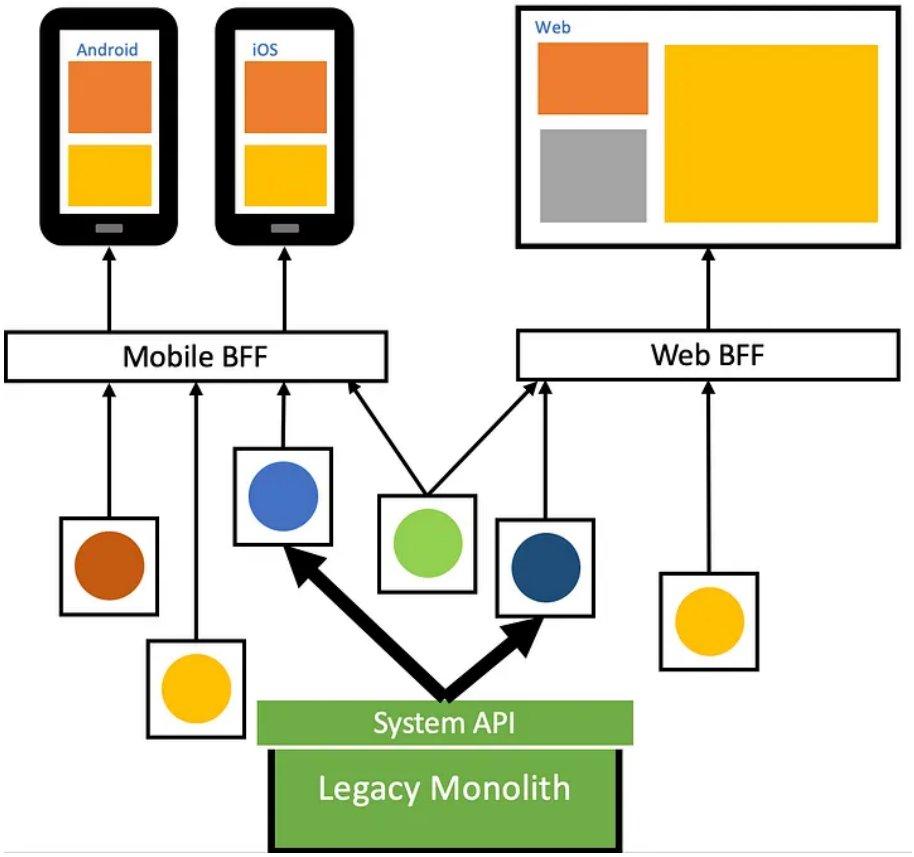
In our example, we have started by creating BFF only for the mobile clients, and in the first version, the web client used the same legacy API. This step is usually tiny as it covers only one BFF without refactoring of the existing system. After its release, you can show the customer and other teams (backend, other frontends teams) the benefits of it, mostly from a maintenance and development perspective. In the presented case, the whole part related to legacy API preprocessing and conversion was extracted to separate microservice from a performance perspective. The future refactoring needs of the legacy monolith system and API were another reason for this extraction.



The first version of the system

The next step was the whole BFF structure for existing frontends. Please check the figure below which shows different types of microservices and their usage:
The blue ones are the layers on top of the monolith system which is under refactoring to microservices.
The green one is the service that is used by both BFFs at the same time.
Yellow ones, on the other hand, are duplicated from a deployment perspective and

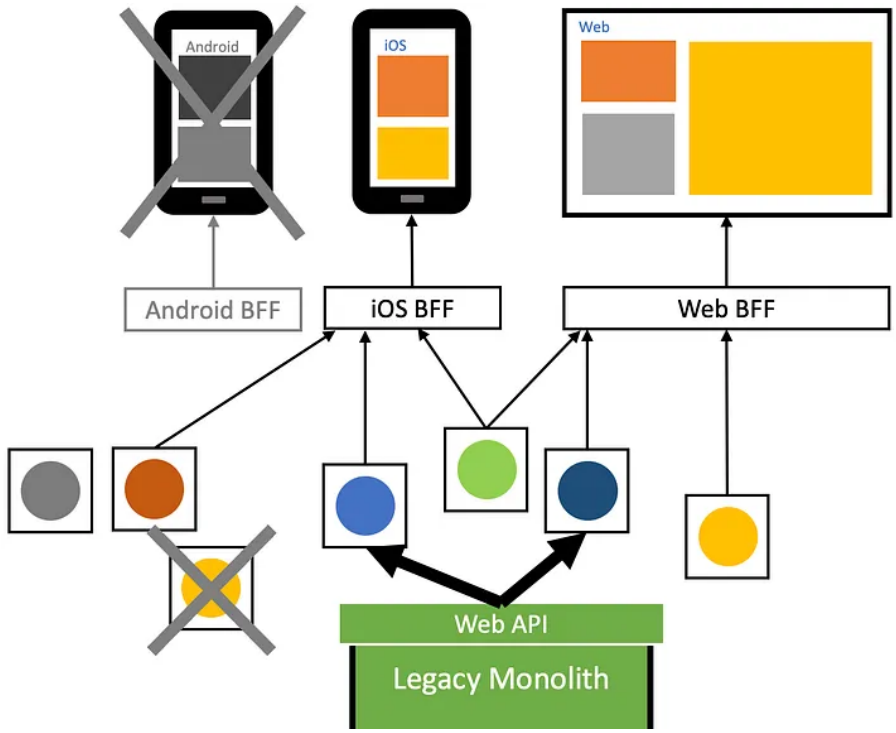
dedicated for each BFF — better performance.
The red one is dedicated only to mobile BFF and provides some mobile-specific solutions, for example, notifications, message queues, etc.



BFF system for Web and Mobile frontends

Fault tolerance in BFF

Fault tolerance is the most significant power of BFF allows us to handle most of the problems on the BFF layer instead of the client (frontend). In case of any microservice change, it can be controlled by all BFFs without emergency deployment of frontend clients. We all know that update of the mobile application on both stores is not an easy operation. It requires some additional time like review, which is not easy to estimate, and sometimes even its output can be a surprise — rejection. With BFF solution, we can cover versioning and backward compatibility separately per each client (frontend). Whole fault tolerance and strategies for that can be covered and managed in the BFF layer. For example, in our system, we can introduce separate BFF for each mobile client and avoid problems when one of them has some issues that affect the whole system, such as doing some self DDoS. In that case, we can disconnect this BFF from the system and investigate the problem inside of it without affecting the rest of the system. This is for sure a good strategy for BFFs dedicated for 3rd party services.



Fault tolerance in BFF

When use BFF?

Like many other patterns, using the BFF in your application depends on the context and the architecture you plan to follow.

For an application that only provides one front-end, I suspect Backend For Frontend will only make sense if and when you have a significant amount of aggregation required on the server-side. Still, even then, the concept of API gateway can be used. BFF style can be considered when you plan to extend the number of frontend types.

If your application needs to develop an optimized backend for a specific frontend interface or your clients need to consume data that require aggregation on the backend, BFF is for sure a suitable option. Of course, you might reconsider if the cost of deploying additional BFFs' services is high, but even then the separation of concerns that a BFF can bring make it a fairly compelling proposition in most cases.

Conclusion

Backend For Frontend is a design pattern created with the developer and, more importantly, the user and their experience in mind. It is the answer to the ever-growing adoption of applications to meet, interact, and serve customers, ensuring consistency while still meeting their diverse and evolving needs.

Video about BFF from Droidcon Berlin

Backend for Frontend - The secret of a great mobile project

"We have API for web, you can reuse it" - sounds familiar? You know this problem which we usually have with API which...

www.droidcon.com

Next article

https://medium.com/mobilepeople/bff-backend-is-a-friend-for-frontend-pros-and-cons-71857725fe7f

Backend Frontend Mobile iOS Android

Published in MobilePeople

427 followers · Last published Oct 19, 2025

Follow

We're the community of people in love with Mobile technologies both native and cross-platform ones, in engineering and testing. We design, conduct & support Mobile sharing experience events, activities and conferences where you can watch, listen and talk about mobile technologies

Written by Michael Szczepanik

513 followers · 25 following

Follow

Solution Architect @ EPAM | Android and Flutter enthusiast

Responses (7)

Ishu

What are your thoughts?

Peeratchai Petpadriew
Jul 2, 2022

Hi Michael,

Thank you for sharing this.

In what circumstances would you consider GraphQL with single API for all clients over this BFF pattern?

7 2 replies Reply

Ryan J. Peterson
May 24, 2023

Would you be interested in contributing to a project I am working with on some friends to add more details around BFF Patterns and their use cases at https://bff-patterns.com/ ?

7 1 reply Reply

Sudhanshu
Jun 6, 2022 (edited)

Michael, I have this question from the Implementation w.r.t. roles in the team.

Who should be designing the BFF in a project? Should it be the Backend developers since they are expert in this role, or should it be the FrontEnd developer?

5 1 reply Reply

See all responses

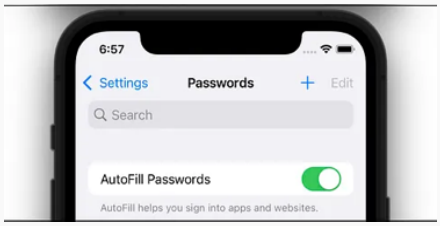
More from Michael Szczepanik and MobilePeople

In MobilePeople by Michael Szczepanik

A Deep Dive into Google Pay and Apple Pay

As mobile payment platforms continue to evolve, the battle between Google Pay and...

Apr 13, 2023 126



In MobilePeople by Elina Semenko

iOS Password AutoFill—how and why?

iOS Password Autofill magic tricks

Oct 13, 2022 203 5

In MobilePeople by Vasiliy Nikitin

Stop using Dispatchers.IO

Why withContext(Dispatchers.IO) is usually not a good idea

Apr 4 507 19

In MobilePeople by Michael Szczepanik

BFF—Backend (as a friend) For Frontend: Pros and Cons

From Tailored APIs to Backend Complexities. Understand how BFF bridges the gap...

Oct 26, 2023 102

See all from Michael Szczepanik

See all from MobilePeople

Recommended from Medium

In ITNEXT by Animesh Gaitonde

Solving Double Booking at Scale: System Design Patterns from Top...

Learn how Airbnb, Ticketmaster, and booking platforms handle millions of concurrent...

Oct 8 2K 24

In Vibe Coding by Alex Dunlop

99% of Developers are Using Claude Wrong (How to Be The 1%)

The Claude techniques most developers never discover

Jun 20 3.6K 71

Abhinav

Docker Is Dead—And It's About Time

Docker changed the game when it launched in 2013, making containers accessible and...

Jun 9 7.1K 200

In Level Up Coding by Fareed Khan

Building an Agentic Deep-Thinking RAG Pipeline to Solve Complex...

Planning, Retrieval, Reflection, Critique, Synthesis and more

Oct 20 1.4K 17

The Latency Gambler

I Interviewed 20+ Engineers. Here's Why Most Can't Code

Over the past year as a Senior Software Engineer at a B2B SaaS company, I've...

Sep 9 2.9K 95

Vladislav Todorov

Level Up Your Mobile System Design: Architecting Feeds &...

Alright, so I've been really diving into mobile system design lately—trying to wrap my...

Apr 30 60

See more recommendations