International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 09 Issue: 06 | Jun 2022 www.irjet.net p-ISSN: 2395-0072

Backend for Frontend in Microservices

Hariom Sharma[1], Dr. Nagaraj Bhat[2]

[1]Student, Department of Electronics and Communication engineering, RVCE, Bengaluru, Karnataka, India
[2]Assistant Professor, Department of Electronics and Communication engineering, RVCE, Bengaluru, Karnataka, India

---

Abstract - *Web based application or desktop based application has quite distinctive requirements as that of mobile based applications moreover with the evolution in technologies and tools, and the constant shift in end users' requirements, the overall process to serve both mobile and work-station based applications using same back-end micro services has also complicated over times. Amidst individual teams working on their corresponding interfaces of front end application, it requires an exclusive narrowing at the back-end to detail the development if it fails to meet the requirement of the application. So this complexity is a kind of loophole further any misalignment and error can cause atrocious experience for the end-users. This paper focuses on overcoming these challenges and improve the user experience.*

***Key Words*: API, API gateway, Monolithic, Request, Response .**

1. INTRODUCTION

Earlier most of the application were targeted to meet requirement of desktop web UI. To meet the requirement of web user interface backend services was made in parallel. As the technology evolved backend services are used to serve both web and mobile users. But the requirements of desktop user interface are different from the requirement of mobile user interface as screen size, performance, and display differ. Any changes done in the backend services to meet the requirement of a user interface will impact the working of other UI and due to this conflicting requirement there are separate teams for different interfaces working on a shared backend which results in unnecessary use of resources, efforts and money. To get rid of this tight coupling, additional layer for different user interface is developed between backend and frontend which act as a gateway between frontend and backend.

An effective way is to identify the business boundaries first then separate the API gateway on this basis and utilize an API gateway per client. As allocation of multiple API gateways, one per client will help to fulfill demands of each client. This procedure is called the "back-end for front-end" or BFF pattern. Decoupling of frontend and backend eliminates the chances of conflicting upgrade requirements and helps to improve performance, reliability and consistency

LITERATURE REVIEW

[1] gave detailed analysis on decisions models for choosing patterns and approaches when it comes to selecting microservices architecture. Detailed analysis on decision in microservices models pattern selection and strategies was done. It also identified that there is still a lack in terms of having an apt decision models that can be used to leverage patterns and strategies as employable comprehension to have relevant design in microservices based systems. They further narrowed down to four decision model which could be used based upon the requirements .

[2] gave detailed analysis on DevOps, cloud and virtualization as an important factors in the microservice ecosystem and gave analysis on the role of these factors. It also covered the areas in which research on microservices to be conducted. It also explored the relationship of microservices with Service-Oriented Architecture and Domain-driven design which are highly used to develop microservices. It also identified that to overcome the hardware limitation containerization as an effective method apart from speeding up the delivery process.

[3] gave a data driven approach to compare microservices and monolithic architectures. There are many reports, research papers and studies which contradicts one another when it comes to making a comparison between microservices architecture and monolithic architecture. So a detailed comparison is being discussed and key performance is being analyzed. While comparing the load testing scenario both architectures performed almost equally well. And in concurrency testing scenarios monolithic showed enhanced performance in terms of latency. Additionally examining microservices applications built with distinct services discovery techs such as Eureka and Consul showed that applications built with Consul have better throughput.

[4] highlighted API gateway as the one of key component for the working of application. It gave detailed analysis on API gateway management for a microservice based architecture. It also gave analysis on the common functions API gateway bypasses which is needed in the microservices and also summarized the certain interior implementation and interface of the system as the exclusive entrance for microservices.

They further wind up by giving a latest resolution for the obstructions in managing API gateway, flow control and reverse proxy function, API gateway gives resolution to the issue of how a client can be designated to an exclusive service and hence enhancing the development competence.

[5] highlighted the evolution of cloud computing in information technology domain and a lot has has been changed and improved when it comes to follow standards, rules and regulations. They gave analysis about how microservices architecture is the preferred and apt choice for on-demand memory, horizontally and vertically scalable, flexible, elastic, and rapidly evolving cloud applications. They followed systematic mapping study for microservices to find out the current trends related to microservices.

2. BACKGROUND OVERVIEW

1. Existing System

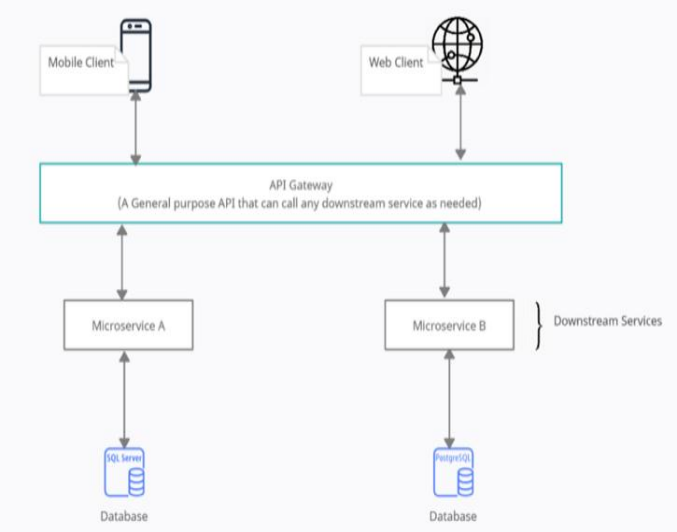

Figure-1 Single API gateway

In microservices based applications, the user interface usually connects with multiple microservices. And this type of interaction can become complex if having many microservices. These microservices could be invoked which in turn makes interaction more complex. So there might be a need to process scissoring concerns in an exclusive or inter-medial place. This is correct place to use API gateway. API gateway serves as an alternate proxy between backend microservices and client applications. The requests initiating from the client apps to the corresponding microservices are being redirected by API gateway and at the same time it manages the scissoring features(reliability, safety, logging and caching).

By aggregation of numerous microservices, delay or lag can also be reduced. This method is suitable for exclusive client systems.

2. Drawback

Some of the drawbacks of a single API gateway are :-

- It's not an effective and efficient approach of having a single API gateway that handles the requests and responds to all microservices for various user interface. Because this has a tendency to make API gateway service inflated over time which in turn can make it inflexible or monolithic.
- Single API gateway acts as single point of entry and failure in it will bring the entire system down.
- A single API Gateway also impacts the speed and reliability of the system since all the device users request goes to same API gateway and response also comes from the same gateway
- One of the most principal drawbacks is that when an API gateway is implemented, then that particular tier is being coupled with the internal microservices. And this might lead to some challenging difficulties for the application with passage of time.
- Along with API Gateway an additional network call also occurs which in turn can cause increased response time. But, this additional call has less implications than having a client interface as this client interface directly calls microservices.
- API Gateway must be scaled out properly if not then it can act as a bottleneck to derail the progress made in case it fails to meet application's requirements.
- There are possibilities of having development obstruction in case API Gateway is being developed by an exclusive team.

3. BFF ARCHITECTURE

To improve and enhance user experience, BFF plays an important role. Regardless of the platform the frontend application is running on, it gives seamless user interaction which is one the main advantages of BFF pattern. It consists of multiple backends to address the requirements of different frontend user interface, like desktop, browser, and native-mobile apps. It enhances the overall performance of the system as the browser resources are utilized efficiently. It allows user to have a seamless interaction as there are well defined API's for specific uses.

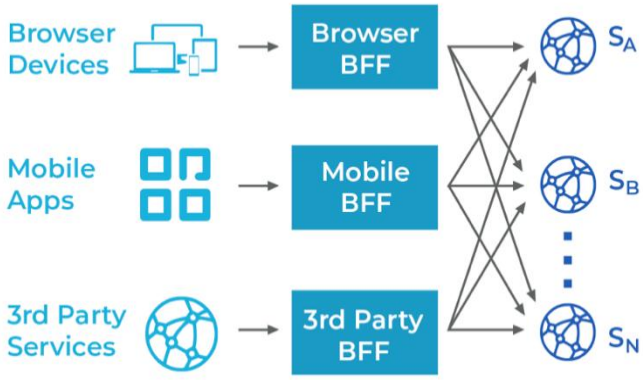

Fig - 2: BFF Architecture

A simple solution to this is to create a intermediate layer for all types of user interface and depending on the needs of the user interface write the logic in the BFF layer. It can be optimized regularly as per our requirements as every type of interface is connected to a specific BFF layer. It is faster than a single API gateway which is shared by all kinds of interfaces. It allows interface team flexibility in terms of language selection while implementing the BFF layer for any interface

4. BFF Working

The user interface layer contains the essential logic to suitably organize the data coming from the backend microservices as the data delivered by backend may not be structured or filtered as per the requirement of user interface. Implementing this logic in the user interface layer has a damaging effect on system's performance as it consumes a lot of browser's resources and eventually affects performance of system.

BFF layer doesn't run on browser's server so the logic executed in the BFF layer doesn't take up browser's resources. To take advantage of this, the logic implemented in user interface layer previously is moved to BFF layer. So, when the user interface calls an API to retrieve the data, the call first goes to the intermediate layer and the intermediate layer invokes the relevant backend microservices. The data transferred by the backend microservices is passed to BFF layer where data is formatted or filtered as per the requirement of the user interface. BFF layer for mobile and web UI users are different and all the layers are connected to same backend services.

5. Implementation of BFF

Implementation is decided based on several factors. There is not an absolute solution for this. Generally either Java or NodeJs is being preferred. Mostly it's based on the technical stack of the organization, skill set of employees, and what enhancements are being focused on like(development expenses, production, performance, cache, memory, security etc). Comparing these two working language :

1. Java

- It is highly appropriate when there is a limitation in terms of CPU.
- Java has been in the industry for quite so long now and it comes with huge potential for mathematical computation and others. It has quite developed IDE and remote debugging features has its own advantages.
- Java runs as an exclusive process based on threads. This thread is responsible for managing each request.

2. NodeJs

- It is highly appropriate when there is a limitation in terms of IO.
- It provides features that minimizes complexity and at the same time enhances development speed.
- NodeJs works on one principal thread which in turn utilizes background threads for tasks. Here the principal/head thread is responsible for changing all your data so it cut short the issues like threading, locks & consistency of data

Because of the asynchronous performance of NodeJs it is being said that it is quite faster but Java is also not behind. Both asynchronous and non-blocking things can be done at a time with the spring reactor after choosing the correct server(Tomcat NIO or other servers developed on top of the NIO connector).

6. Results

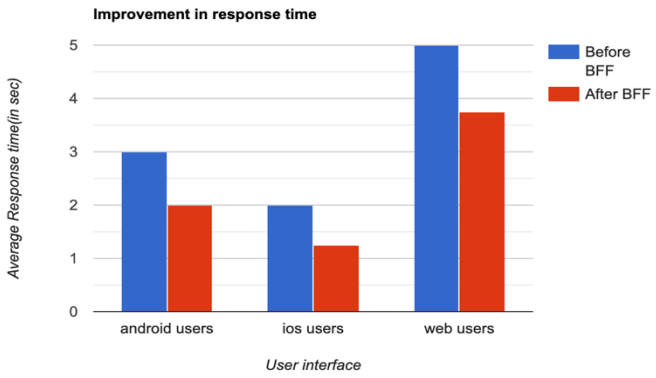

Fig – 3 : Average response time in different UIs

Figure-3 shows different user interface average response time before and after the implementation of BFF when frontend calls to backend services. For android users average response time is 3 seconds before BFF layer was introduced in the system and it improved by 1 second after the implementation of BFF. Similarly for ios users average response time improved by 750 milliseconds and for desktop web users it improved from 5 seconds to 3.75 seconds before and after the implementation of BFF respectively.

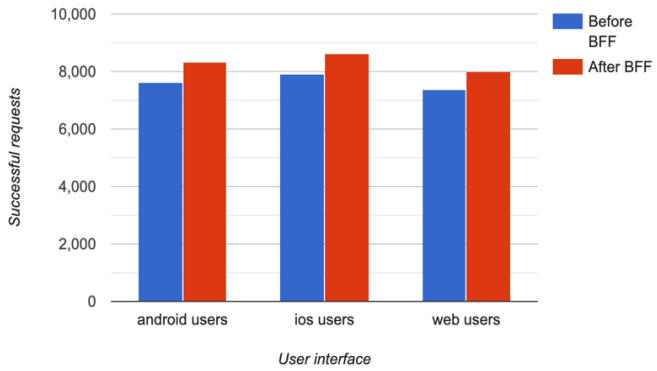

Fig – 4 : Successful requests for different UIs

Figure-4 shows total number of successful requests when 10000 requests are made to backend services for each user interface before and after the implementation of BFF. For android users the number of successful requests increased from 7645 to 8324 before and after implementation of BFF. Similarly for ios users total number of successful requests increased from 7913 to 8637 and for desktop web users successful requests increased to around from 7362 to 8012.

7. The Challenges in BFF

It is evident that a BFF layer has many advantages but it's important to know about the challenges before implementing BFF. These are :-

- Fan Out: A breakdown of the single service will impact the users who have the same type of device interface to access backend microservices. Fig-5 represents breakdown in the BFF layer for web users which affects all the users who uses desktop browser.

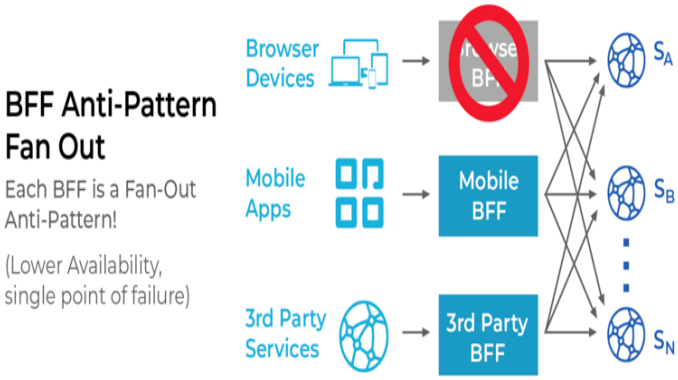

Fig – 5 : BFF Anti-pattern Fan Out

- Fuse: Any failure in the microservices that responds to requests of multiple BFFs can bring down the whole system. Fig-6 shows failure in the microservices which will impact all the users who want to access microservices.

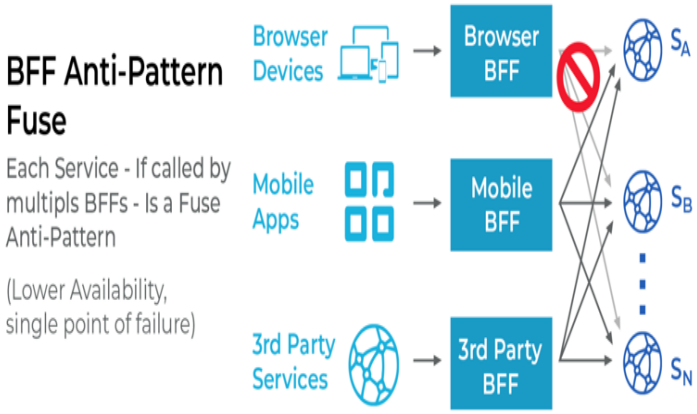

Fig – 6 : BFF Anti-pattern Fuse

- Duplication and Lower Reuse: The cost of development will be more since there will be deployment of multiple BFFs having similar capabilities with different teams. Faster response time and increased consistency may decree this interruption.

8. Overcoming challenges

Resolve issue of Fan out:- Fault isolation needs to be implemented as there shouldn't be any exclusive corresponding services which BFF synchronizes that would take it down entirely. Preferably each and every posterior services will have their own termination point of BFF for each module. This will highly improve availability and fault isolation with a little cost in terms of increased number of deployments. Between all downstream components if we need interaction and coordination then we have to rethink

the cause of splitting each corresponding service as in when to split components.

Rectify fuses:- In order to resolve this we need a dedicated service to each BFF interface. The possibility of dedicating services is only feasible if corresponding services do not need to share a database because in that case the database itself will become a fuse. Henceforth if a service requires a database then it is preferable to have separate deployments to enhance availability. In case databases are needed by the service then there will be technical debt that will be only partially remediated by eliminating fan out.

Reuse:- Depending upon the implementation this problem may or may not occur. However if there are possibilities that there could be overlapping of functionalities between different modules there it is preferable to ensure that teams are recognising larger efforts which should be shared. If these larger requirements are implemented in reusable libraries it will help to lower down the development expenses and at the same time there will be reduction in time to market for other features.

Multiplication:- In the above points we have already discussed if the teams own their services via the service lifecycle and executes easier releases and communications using automation then it will rectify all the problems of large number of deployable services.

9. Conclusion

Backend For Frontend is design pattern created to improve the user experience. With time every application needs to be upgraded as the requirements of customers changes rapidly. So, BFF as an intermediate layer is an efficient way to solve various user interface conflicting upgrade requirements and also provide consistency to the application. The smaller size, extensibility and re-usability of microservices architectures add scalability, flexibility power to development as well as operations team. This paper focuses on problems developers face while working on a single API gateway and how multiple gateway helps to tackle the challenges and problems developers might face and how different language can be used to implement the intermediate layer depending upon the requirements of user interface and backend microservices.

References

[1] Muhammad Waseem, Peng Liang, Aakash Ahmad, Mojtaba Shahin, Arif Ali Khan, Gastón Márquez "Decision Models for Selecting Patterns and Strategies in Microservices Systems and their Evaluation by Practitioners" in 2022 44th International Conference on Software Engineering (ICSE) SEIP Track, 2022

[2] Mohammad Sadegh Hamzehloui, Shamsul Sahibuddin, and Ardavan Ashabi "A Study on the Most Prominent Areas of Research in Microservices" in 2019 International Journal of Machine Learning and Computing, 2019

[3] Omar Al-Debagy, Peter Martinek "A Comparative Review of Microservices and Monolithic Architectures" in 2018 18th IEEE International Symposium on Computational Intelligence and Informatics, 2018

[4] J T ZHAO S Y JING and L Z Jiang "Management of API gateway in microservices architecture" in 2018 IOP Conference Series: Journal of Physics, 2018

[5] Hulya Vural, Murat Koyuncu, and Sinem Guney "A Systematic Literature review on Microservices" in 2017 International Conference on Computational Science and Its Applications, 2017

[6] H. M. Ayas, P. Leitner, and R. Hebig. 2021. Facing the giant: A grounded the-ory study of decision-making in microservices migrations. In Proc. of the 15thACM/IEEE Int. Symp. on Empirical Software Engineering and Measurement (ESEM).ACM, 1–11

[7] R. Chen, S. Li, and Z. Li "From Monolith to Microservices:- A Data-flow-Driven Approach," in 2017 24th Asia-Pacific Software Engineering Conference (APSEC), 2017.

[8] Tan Yiming. Design and Implementation of Platform Service Framework Based on Microservice Architecture [D]. Beijing Jiaotong University , 2017.

[9] Tan Yiming. Design and Implementation of Platform Service Framework Based on Microservice Architecture [D]. Beijing Jiaotong University , 2017.

[10] S. Li, H. Zhang, Z. Jia, Z. Li, C. Zhang, J. Li, Q. Gao, J. Ge, and Z. Shan. 2019. Adata□ow-driven approach to identifying microservices from monolithic applica-tions. Journal of Systems and Software 157 (2019), 110380.

[11] R. Matt. 2020. Security Patterns for Microservice Architectures. https://tinyurl.com/zs85z9as accessed on 2021-07-05.

[12] M. Villamizar, "Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures," Jun. 2017.

[13] N. Dragoni, I. Lanese, S.T. Larsen, M. Mazzara, R. Mustafin, and L. Safina. 2017.Microservices: How to make your application scale. In Proc. of the 11th Int. AndreiErshov Memorial Conf. on Perspectives of System Informatics (PSI). Springer, 95–104.

[14] H. Harms, C. Rogowski, and L. Lo Iacono. 2017. Guidelines for adopting frontendarchitectures and patterns in microservices-based systems. In Proc. of the 11thJoint Meeting on Foundations of Software Engineering (ESEC/FSE). ACM, 902–907.

[15] S. Haselböck and R. Weinreich. 2017. Decision guidance models for microservicemonitoring. In Proc. of the 14th Int. Conf. on Software Architecture Workshops(ICSAW). IEEE, 54–61.

[16] S. Haselböck, R. Weinreich,and G. Buchgeher. 2017. De cision guidance models formicroservices: Service discovery and fault tolerance. In Proc. of the 5th EuropeanConf. on the Engineering of Computer-Based Systems (ECBS). ACM, 1–10.

[17] P. Raj, H. Subramanian, and A. C. Raman. 2017. Architectural Patterns: UncoverEssential Patterns in the Most Indispensable Realm of Enterprise Architecture. PacktPublishing Ltd.

[18] G.A. Lewis, P. Lago, and P. Avgeriou. 2016. A decision model for cyber-foragingsystems. In Proc. of the 13th Working IEEE/IFIP Conf. on Software Architecture(WICSA). IEEE, 51–60.

[19] Balalaie, Armin, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables DevOps: migration to a cloud-native architecture. IEEE Software, 2016. 33(3) 42-52.