International Journal of Science and Research (IJSR)
ISSN: 2319-7064
ResearchGate Impact Factor (2018): 0.28 | SJIF (2019): 7.583

Decomposition Techniques in Distributed Service Architecture

Mallanna S D

[1]Assistant Professor, Siddaganga Institute of Technology, Tumakuru, Karnataka, India

Abstract: ***In this modern era of cloud computing, distributed service design in web applications is quintessential to harness the benefits of scalability and elasticity. Such design can be achieved either by implementing the service-oriented architecture (SOA) or a fine-grained micro service architecture (MSA).Defining the boundaries of the services and decomposing them into separate units is a challenging task in the web service design. Single responsibility principle (SRP) and common closure principle (CCP) are the 2 major guiding principles which drive the decomposition of the backend into micro services. Services can be logically segregated either by (i) delineating the sub-domains of the business served and converting each sub domain into a bunch of services grouped together as a microservice, or (ii) by laying out the major entity model and then building the services related to the capabilities of each entity grouped together as a microservice. Both these approaches have their own benefits and issues. This paper discusses these approaches by taking the real-world examples and explains which pattern is suitable under which circumstances.***

Keywords: Microservices, Service oriented architecture, Service Decomposition, Single responsibility principle, common closure principle

1. Introduction

The modern-day web application backend design entails distributed service architecture. Not long ago all the applications were built using a monolithic design pattern. Monolithic systems were easy and convenient to build when client-server model was one of the most popular designs for the distributed systems, as explained by Tasneem Salah and other in their research work [1]. As change is key to the success of any business, the web applications need to be changed quickly to handle the needs of the business. Things started to escalate quickly, and monolithic applications started to fade. Monolithic systems get more complex to handle as the size of the application grows as all the code is stacked in a single unit. They got difficult to debug the production issues and test the incremental changes. Build cycles were also significantly longer due to the bulk of code stacked up as a single unit. These issues have given rise to the newer design patterns which involved breaking down the monolith to a more loosely coupled services which was known as Service Oriented Architecture (SOA). In this pattern, a web front end, mobile or other third-party callers can make calls to the distributed backend services. These calls are handled by Enterprise Service Bus (ESB) which integrates various application services together over a Bus-like infrastructure. ESB comes with a embedded service registry which keeps track of the backend services. ESB translates the calls coming to it and translates them to the suitable message type understood by the relevant service in the backend. Though SOA was a great improvement from the monolith design, it had it's own share of issues. Though the services were delineated in SOA, they needed to be deployed as a single unit in the form fat application services. On top of it as examined by researchers like Chaitanya K Rudrabhatla [2].SOA performs the service routing, orchestration and business validations at a single central hub called ESB, which becomes a cumbersome layer as services grow. To handle these draw backs Micro Service Architecture (MSA) came into light.

The microservice architecture structures an application as a set of loosely coupled services. This design greatly helps in accelerating the software development lifecycle by enabling the continuous integration, development and deployment (CI/CD). The biggest advantage of micro services comes from the fact that it enables the components to be deployed independently. This greatly simplifies the development, testing and deployment cycles as the changes are limited to a smaller region rather than the entire monolith. When this design is paired up with the cloud environment and distributed using the smaller containers, the benefits are enormous. The smaller containers can start and stop quickly, thus enabling the auto scaling seamlessly. This gives the elasticity and horizontal scaling capabilities efficiently. But these benefits are not automatically realized. Instead, they can only be attained by the careful functional decomposition of the application into services. Rest of this paper discusses the techniques to decompose the web application services in an optimal way.

2. Decomposition of Microservices

While designing the micro services, it should be ensured that a service must be small enough to be easily developed and tested. To design the smaller services, the backend functionality needs to be decomposed into services in an efficient and reliable way. There are 2 major guiding principles which can drive the decomposition of services.

Single Responsibility Principle (SRP) is a guiding principle which defines a responsibility of a class as a reason to change, and states that a class should only have one reason to change. It is highly beneficial to apply SRP and design services that are cohesive and implement a small set of strongly related functions [3].

The backend web services also be decomposed in a way so that most new and changed requirements only affect a single service. That is because changes that affect multiple services require coordination and more testing, which slows down development. This is the essence of Common Closure

Licensed Under Creative Commons Attribution CC BY

Principle (CCP), which is a second guiding principle for service decomposition which states that classes that change for the same reason should be in the same package. Perhaps, for instance, two classes implement different aspects of the same business rule. This principle states that only one package should be impacted when a business rule changes. The section below describes the techniques to implement the SRP and CCP techniques while implementing the MSA.

3. Domain Driven Design

3.1 Key considerations

Domain Driven Design is an architectural pattern used to decompose the services by following the common closure principle (CCP). As per this, all the classes which get impacted by a chance should be packaged together as a microservice. Some of the key considerations for this implementation are –

1) Services must be designed to be cohesive in nature. A service should comprise of a small set of strongly related business activities.
2) Each service should be autonomous and be loosely coupled. Which means in case of a change, it should be possible to code and deploy the service alone without impacting anything else.

3.2 Real world example

This architectural pattern can be explained by taking a real-world industry example. For ex –If you consider an ecommerce application, the various business functions can be listed at a high level as –

1) Inventory management.
2) Order management.
3) Payment system.
4) Shipping management.

As per this principle the services should be segregated in such a way that all the functionality related to the business module should be packaged an independent micro service as shown in Figure 1 below. This design helps because the changes can be segregated and deployed individually. However, it has its own issues as listed below.

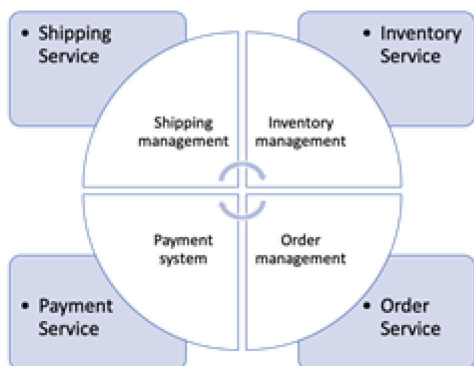

Figure 1: Domain driven design for e-commerce application

3.3 Challenges with Domain Driven model

Though the domain driven model helps in reducing the impact of change percolating into services dealing with other business functionality, it has some challenges which need to be carefully dealt with. Here are some of the key challenges-

1) Sometimes the microservice dealing with a sub domain might be too heavy depending on the functionality involved. So further break down of the modules might be necessary. Decomposing a single sub domain to multiple services might complicate the data queries and also pose challenges in persisting the transactions.
2) When the services become heavy it might have a negative impact on throughput and latency of the service [4]
3) When the sub domain is broken into micro domains, it calls for a decentralized framework for microservice coordination [5] [6]
4) Though a database per service model can be followed to maintain isolation of services, there still would be a need to pass the state of entities across services. This might complicate the database and network design.

4. Entity Driven Design

As an alternative to the above pattern Event driven design is presented. Entity driven model is a design pattern where services are based on the entities involved in the business transaction. In this design, the services are designed around the activities of the entities. This would let a web application to be broken down into as many micro services as the number of major entities involved in the business flow. This might prove advantageous as the entity states are persisted in one place and a reactive event driven approach can be taken to propagate the transactions to other microservices. However, this has become more of an anti-pattern due to the following drawbacks.

4.1 Challenges with Entity Driven model

Listed below are some of the shortfalls of the entity driven model for the decomposition of microservices –

1) It might lead to a granular micro service model which might become complex to handle[7]
2) Inter service communication becomes a major issue as this is needed for almost all use cases in this design.
3) It would need a number of orchestrator pattern services to handle the business logics which span across multiple entities. Thus, complicating the design further[8]
4) Asynchronous event choreographies might be needed for transaction propagation even for the flows which deal with the same business sub-domain.

5. Conclusion

Table 1: Domain driven vs entity driven models

Domain Driven	*Entity Driven*
Clear segregation of services based on business needs	Business functionality is spread across orchestrator services.
A business change is limited to a single service	Small change might cause a change across multiple services
May cause latency issues in fat services [9]	Services are fine grained.
Transactions are mostly bound to the service	Transactions are not bound to the service
Lower need of interservice communication	Greater need of interservice calls.

References

Licensed Under Creative Commons Attribution CC BY

[1] Salah, Tasneem & Zemerly, Jamal & Yeob Yeun, Chan & Al-Qutayri, Mahmoud & Al-Hammadi, Yousof. (2016). The evolution of distributed systems towards microservices architecture. 318-325. 10.1109/ICITST.2016.7856721.
[2] Chaitanya K Rudrabhatla. A Systematic Study of Micro Service Architecture Evolution and their Deployment Patterns. International Journal of Computer Applications 182(29):18-24, November 2018.
[3] C. Richardson, "Pattern: monolithic architecture", Microservices. io, 2019.
[4] Jayasinghe M., Chathurangani J., Kuruppu G., Tennage P., Perera S. (2020) An Analysis of Throughput and Latency Behaviours Under Microservice Decomposition. In: Bielikova M., Mikkonen T., Pautasso C. (eds) Web Engineering. ICWE 2020. Lecture Notes in Computer Science, vol 12128. Springer, Cham
[5] S Newman, Building microservices: designing fine-grained systems, O'Reilly Media, Inc, 2015.
[6] D. Goel and A. Nayak, "Reactive Microservices in Commodity Resources," 2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, 2019, pp. 3658-3665, doi: 10.1109/BigData47090.2019.9006584.
[7] S. Newman, Building microservices: designing fine-grained systems . O'Reilly Media, Inc., 2015
[8] V. F. Pacheco, "Microservice patterns and best practices: Explore patterns like cqrs and event sourcing to create scalable, maintainable, and testable microservices," 2018
[9] A. Akbulut and H. G. Perros, "Performance Analysis of Microservice Design Patterns," in IEEE Internet Computing, vol. 23, no. 6, pp. 19-27, 1 Nov.-Dec. 2019, doi: 10.1109/MIC.2019.2951094.

Licensed Under Creative Commons Attribution CC BY