Journal of Information Technology and Computer Science
Volume 5, Number 3, December 2020, pp. 338-347
Journal Homepage: www.jitecs.ub.ac.id

Mobile Application Architecture Restructuring with Microservice Approach

Ardiono Roma Nugraha[1], Aini Suri Talita[2]
[1]Master of Information System Management, [2]Faculty of Industrial Technology
Gunadarma University, Indonesia
{[1]ardionoroma, [2]ainikrw}@gmail.com

Received 01 September 2020; accepted 30 December 2020

Abstract. Microservice is an architecture that can solve many problems in a monolithic architecture. One of the problems is the ability to handle many concurrent users. The existing monolithic application can be restructured into microservices to increase robustness in handling a lot of users, without exception native mobile application. This study aimed to restructure the existing native mobile application named TemanBisnis into microservices. The restructuring process can be done by splitting the application features according to its business domain into one service. Two microservice architecture designs were proposed in this study, named 3-1 architecture and 2-1-1 architecture. Both architectures can handle up to 100 concurrent users, although they start to produce errors. By performance, the 3-1 architecture is better than the 2-1-1 architecture. In the end, an existing native mobile application can be restructured into microservices. The 3-1 architecture should be adopted to achieve the best results between these two architectures.

Keyword: Microservices, Software Architecture, Architecture Restructuring, Native Mobile Application, SME

1 Introduction

Monolithic is one of the most popular software architecture. It is an architecture that combines all of its components into one thing (monolith) [1]. Monolithic allows software architects to deploy the software once, and all of the system components will be in production. To accompany the system needs, new features are added into the system. Now the system grows, much more complex and slower than before.

This problem can be solved by microservices. Microservices is an architecture that structures a system into collection of services that highly maintainable, loosely coupled, independently deployable, and organized around the business capabilities [2]. All the functionalities inside the monolith system will be arranged into a subsystem that small enough to maintain. If we want to upgrade one of the services, it will not affect the whole system because they are not directly dependent. If there is an error to one of the services, user will not consider it because the user interface is still displaying as usual.

TemanBisnis is a mobile application that enables SME (*small, medium-sized enterprises*) entrepreneurs to record their financial transactions through their smartphone, rather than manually recording them. The current architecture of TemanBisnis is a native mobile application combined with a simple monolithic web

service. Its main business process are mainly done in the user's smartphone and the web service will handle the report export and payment features. This architecture creates a problem when a user uses a low specification smartphone and they have a lot of transactions recorded, TemanBisnis performance will be much slower. When there are a lot of users access the export feature concurrently, sometime the web service also fails to generate the report.

In 2023, TemanBisnis has the vision to get more than one million users from SME entrepreneurs. With their current architecture, it seems difficult and high risk to handle that number of users with million transactions. This study will discuss how to restructure the current architecture of TemanBisnis into microservices and what kind of microservice architecutre that suitable with their need.

Related to this study, in 2019 Rizki Mufrizal and Dina Indarti have been restructured the existing monolithic application into microservices [3]. They used the *strangler pattern* strategy to restructure the application. Chen-Yuan Fan and Shang-Pin Ma in 2017 also successfully migrated a mobile application named EasyLearn into a microservice architecture [4]. Bucchiarone et al. researched in 2018 about how reimplementing a monolithic architecture into microservices can improve scalability [5].

This study will adopt the Waterfall model as a research method. By splitting current TemanBisnis features into services according to its business domain, the microservice architecture is designed. The current TemanBisnis features that will be restructured into microservice architecture are only transaction recording, business contact, custom category, inventory management, and inventory transaction recording. The microservice architecture then will be implemented and tested to find out how its performance, whether it can fulfill the TemanBisnis' need or not.

Usually, the microservice architecture is designed by splitting the monolithic architecture by its business domain [6, 7]. Each service is deployed individually and can communicate with each others. This splitting cause the architecture to has branch-like form. This form is commonly found in the real world. [3, 4] use this and this form is also explained in many books [8, 9, 10]. We also use this form in this study and later we will call it as *3-1 architecture*.

The branch-like or *3-1 architecture* form is not an obligation when creating microservice. We can modify the architecture form according to what we need. There are many patterns that we can consider while designing the microservice architecture, like aggregator pattern, chained pattern, branch microservice pattern, API gateway pattern, etc [8, 13]. Each pattern has its own pros and cons, resulting in different architecture form. In paper [5], they modified the microservice architecture to circular-like form because they need to use Redis and RabbitMQ. In this study, we later will also modify the *3-1 architecture* to compare the modified architecture performance (later we will call it as *2-1-1 architecture*) with the usual *3-1 architecture* form.

2 Research Method

In this study, the Waterfall model is adopted as a research method with an adjustment. Waterfall model is used in this study because it is focused on its each stage and relative statically defined at the first stage [16]. This helps TemanBisnis as a small company to plan the time of developing this architecture without distract their other developments. Because this study only provide prototype of the microservice architecture, the maintenance stage in the Waterfall model will be replaced with *results and discussion* process to get a conclusion from the

microservice architecture that has been created. In general, the research method stages of this study can be seen in Figure 1.

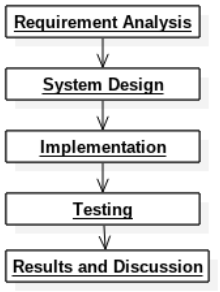

Figure 1. Research Method Stages

2.1 Requirement Analysis

In the first stage, requirement analysis process is done by collecting data from an interview with the CTO of TemanBisnis. The data collected from the requirement analysis stage will be used for designing system architecture in the next stage. To create a microservice, a monolithic application is usually splitted by its business domain [6, 7]. In this study, the existing TemanBisnis application is splitted into three services, namely Accounting, Inventory, and User service. Accounting service is responsible for features that related with accounting domain, that are transactions recording, business contact, and custom category. Inventory service is responsible for features that related with inventory domain, like inventory management and inventory transactions recording. User service is responsible for features that related with user management, like register, login, logout and profile edit.

2.2 System Design

For this study, two architecture designs were proposed. The first one is called "3-1 form" that is usually used and explained in many microservice books [8, 9, 10]. The second one is called "2-1-1 form". The reason about proposal of these two designs is because the researcher wants to know the performance of the commonly used "3-1 form", and what will happen to its performance if a slight change is done to that architecture by using "2-1-1 form". Figure 2 and 3 show the comparison between the 3-1 and 2-1-1 architecture design.

As shown in the Figure 2, the 3-1 architecture deployed all of its services including databases and API gateway into the Google Cloud Platform (GCP) server. All services (Accounting, Inventory, and User service) are placed parallel an each wrapped in a Docker container. All databases are grouped into one database Docker container, and this container can interact with the services container vice-versa. All of these services are guarded by an API gateway which can determine what request can be processed into the system and what request must be rejected. To add more security, each request also protected by a JWT token that must be verified and valid before allowed to access the service.

The 2-1-1 architecture is slightly different with the 3-1 architecture. As shown in the Figure 3, all services are still deployed into GCP server. Instead of placing services parallely, only the Accounting and Inventory service that placed parallel, while the User service guarding them at the front. The reason of moving the User

service in front of Accounting and Inventory service is the researcher expecting more security because the request need to pass the API gateway and User service, before proceed to the Accounting or Inventory service.

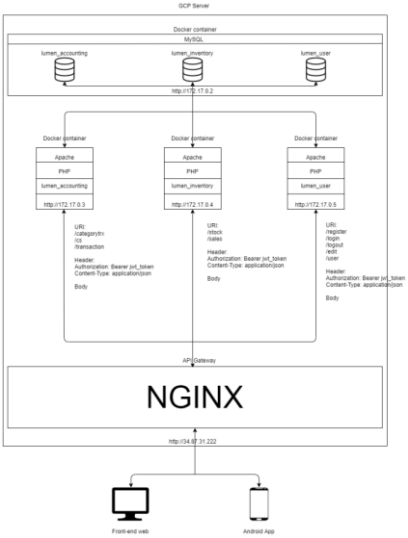

Figure 2. The 3-1 architecture design

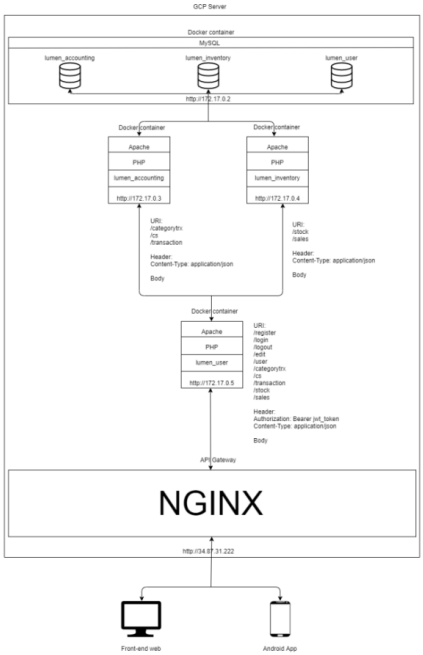

Figure 3. The 2-1-1 architecture design

Besides the architecture design, the database also needs to be refactored. The *database per service* concept is used in this study [11, 12]. Figure 4 shows the database architecture before refactoring.

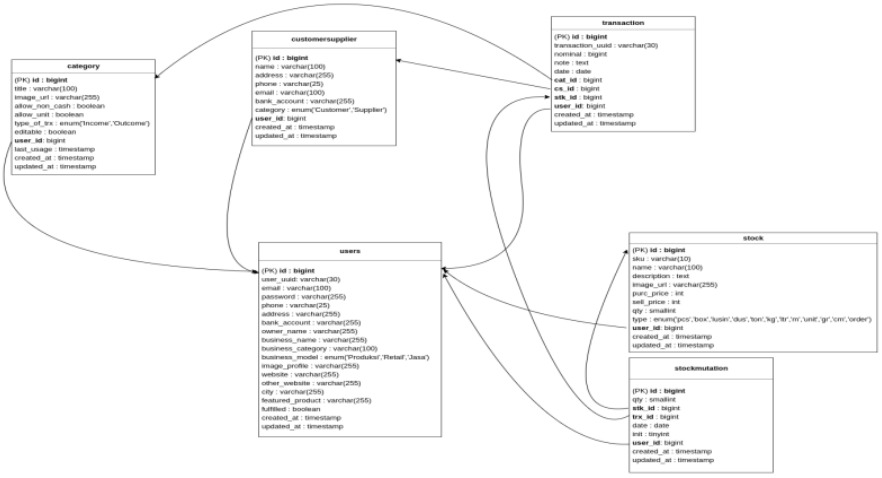

Figure 4. The database architecture before refactoring

After refactoring, the database is now splitted into three databases, each responsible for its service. The accounting database is responsible for the accounting service, consists of the transaction table for transactions recording feature, customersupplier table for the business contact feature, and category table for the custom category feature. The inventory database is responsible for the inventory service, consists of the stock table for inventory management feature, and stockmutation table for inventory transactions recording feature. The user database is responsible for the user service, consists of the users table for user management feature. The Figure 5-7 show the accounting, inventory, and users table after refactoring, respectively.

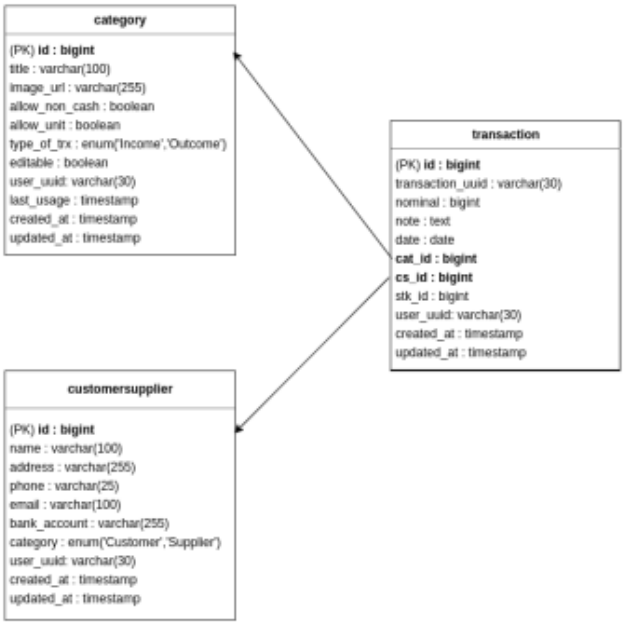

Figure 5. The Accounting service database architecture after refactoring

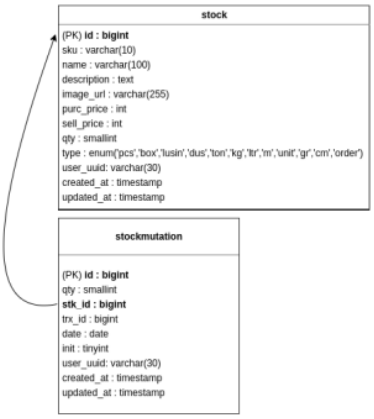

Figure 6. The Inventory service database architecture after refactoring

Figure 7. The User service database architecture after refactoring

2.3 Implementation

The implementation stage is done by creating prototypes that implement the microservice architecture designs that created in the Figure 2 and Figure 3. The prototypes then deployed into the Google Cloud Platform (GCP) server. Table 1 shows the tools used in the implementation stage.

Table 1. System Development Tools

Type	Tools Name
Programming Language	PHP 7.2
Framework	Lumen 5.8
Database	MySQL 5.7
Web Server	Apache 2.4.29
Containerization	Docker 19.03.2
API Gateway	NGINX 1.10.3
Server Platform	Google Cloud Platform – Google Cloud Engine
Operating System	Ubuntu Server 16.04 LTS

2.4 Testing

For the testing purpose, API performance testing is done. This testing aims to determine the performance of both architectures in handling concurrent users. The testing will be done in the GCP server for all services using Gatling framework. Gatling is a highly capable load testing framework. It is suitable to test the

performance of a variety of services, especially web applications that using HTTP protocol like in this study [15]. Table 2 shows the testing scenario in this study. The *login* case is for testing the User service, *get all transactions* is for testing the Accounting service, and *create an inventory transaction* is for testing the Inventory service. The results and discussion will be discussed in Section 3.

Table 2. Testing Scenario

Case	Concurrent Users	Testing Duration
Login	25 users	5 minutes
	50 users	
	100 users	
Get all transactions	25 users	
	50 users	
	100 users	
Create an inventory transaction	25 users	
	50 users	
	100 users	

3 Results and Discussion

This study was conducted by restructuring TemanBisnis architecture from a native mobile application into microservices. In general, the processes of this study are collecting data, designing the microservice architecture of the system, refactoring the database, creating the prototype, deploying to the server, testing, and evaluating the testing results. This section discusses the testing results and concludes which architecture has the better performance and should be adopted.

3.1 Results

The testing flow in this study is by testing the 3-1 architecture for 25 concurrent users, then testing the 2-1-1 architecture for 25 concurrent users, and so on. Table 3-5 show the testing result for each case.

Table 3. Login Testing Result

Architecture Name	Concurrent Users	Total Requests	Response Time			
			t < 800 ms	800 ms < t < 1200 ms	t > 1200 ms	Failed
3-1 Architecture	25 users	2669	16	12	2641	0
	50 users	2623	4	6	2613	0
	100 users	2483	0	10	2473	0
2-1-1 Architecture	25 users	2566	0	14	2552	0
	50 users	2648	7	27	2614	0
	100 users	2202	0	10	2192	0

Table 4. Get All Transactions Testing Result

Architecture Name	Concurrent Users	Total Requests	Response Time			
			t < 800 ms	800 ms < t < 1200 ms	t > 1200 ms	Failed
3-1 Architecture	25 users	3456	349	677	2420	10
	50 users	2872	7	16	2844	5
	100 users	2795	1	5	2781	8
2-1-1 Architecture	25 users	1565	4	1	1556	4
	50 users	1238	1	0	1237	0
	100 users	6602	4044	295	998	1265

Table 5. Create New Inventory Transaction Testing Result

Architecture Name	Concurrent Users	Total Requests	Response Time			
			t < 800 ms	800 ms < t < 1200 ms	t > 1200 ms	Failed
3-1 Architecture	25 users	3456	90	198	3258	0
	50 users	2872	0	10	2712	0
	100 users	2795	2434	117	2354	1275
2-1-1 Architecture	25 users	1565	315	472	3092	5
	50 users	1238	17	21	3445	19
	100 users	6602	1801	597	2273	2821

From the testing results, both architectures can handle up to 50 concurrent users without error. When handling 100 concurrent users, the system starts to unstable. In the Table 3 and 4, the failed requests is quite small compared to the Table 5. This is because the flow tested in Table 3 and 4 is a simple read process from the database that does not need to processed further by the application. Most errors happened in the Inventory service. This because to create a new inventory transaction, the Inventory service needs to access the Accounting service to create a new transaction record there. This complex and quite long process cause the system to produce errors when accessed by 100 concurrent users. However, the 2-1-1 architecture caused more errors than the 3-1 architecture, especially while handling 100 concurrent users because of its quite complex flow inside the architecture. Most requests also done in more than 1200ms may be caused by the busy Apache web server that still processes the incomplete previous requests.

3.2 Discussion

From the results obtained in Subsection 3.1, the microservice architecture designed in this study has been successfully implemented as a prototype and can implement the current TemanBisnis features. Both 3-1 or 2-1-1 architecture can still fulfill the TemanBisnis' needs, that are capable to handle many users and not too rely on user's smartphone. The architectures start to reach its limit and produce many errors while handling 100 concurrent users that access the system simultaneously. For the 2-1-1 architecture, the errors may caused by the architecture form that require the request to pass the User service first before continue to the Accounting or Inventory service, so the flow become more complex than the 3-1 architecture. But in the 3-1 architecture, the similar errors are still happened, although the flow is more simple.

The errors that happened may caused by the limitation of the tools used in this study. This study uses Lumen, a PHP framework. PHP is an interpreted language that translates the program in every execution that causes a slower process [12]. Besides that, this study also uses Apache as a web server that runs the program inside the docker container. Apache is a process-driven web server that creates a new thread for every request [14]. This may cause huge memory consumption in the server. The server used in this study only has 1.7 GB of RAM and all these limits can cause the server being unstable while handling more than 100 concurrent users.

From this study, based on performance the 3-1 architecture is better and more stable than the 2-1-1 architecture because of the simpler flow. In microservice practice, it is also not recommended to build architecture similar to the 2-1-1 architecture. It is because all services are still dependent to User service. If the User service is down, the

whole system will be down and the system can do nothing. This is against the microservice principle that the architecture should be loosely coupled [10].

From this discussion we can conclude that a native mobile application can change their architecture into microservices. Both architectures are capable of handling many users, although the number of maximum users that can be handled depends on the server specification and the tools used. From the Results, the 3-1 architecture is better than the 2-1-1 architecture because of its simpler flow and each service inside is more independent than the 2-1-1 architecture has. If a native mobile application, especially TemanBisnis want to achieve the best performance between these two architectures, they may adopt the 3-1 architecture as their new microservice architecture.

4 Conclusion

The process of restructuring a native mobile application architecture from an application named TemanBisnis to microservice architecture has been successfully done. It can be done by collecting data about the existing architecture as much as possible and used it as a benchmark for designing the microservice architecture. The current features of the existing application then splitted into services based on its domain. The current single database is also redesigned using *database per service* concept.

The prototype of the new architecture also successfully implemented the current features of TemanBisnis. The transaction recording, business contact, custom category, inventory stock management, and inventory transaction recording feature has been implemented in the microservice architecture prototype. The architectures that proposed in this study are the 3-1 architecture and the 2-1-1 architecture. Based on performance, the 3-1 architecture is better than the 2-1-1 architecture.

Based on testing conducted in this study, both architectures can handle up to 50 concurrent users without produce any error. When the user number is increasing until 100 concurrent users, the microservice architecture especially the 2-1-1 architecture starts to become unstable and produces many errors. The errors may caused by the limitation of the tools used in this study.

From the Conclusion, some things can be improved from this study. The tools used in this study can be changed or upgraded to improve the performance of the microservice architectures proposed. As a recommendation for the future researches, the microservice architecture designs proposed in this study can be developed by modifying or combining the other principles or patterns in microservice. This research can be used as a reference for other future researches that want to restructure the native mobile application or monolithic application into microservices.

References

1. Fowler, S. J.: Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization, p. 20. O'Reilly Media, Inc., California (2017)
2. Newman, S.: Building Microservices: Designing Fine-Grained System, p. 2. O'Reilly Media, Inc., California (2015)
3. Mufrizal, R., Indarti, D.: Refactoring Arsitektur Microservice pada Aplikasi Absensi PT. Graha Usaha Teknik. Jurnal Nasional Teknologi dan Sistem Informasi vol. 05 no. 01, 57-68 (2019)

4. Fan, C. Y., Ma, S. P.: Migrating Monolithic Mobile Application to Microservice Architecture: An Experiment Report. IEEE 6th International Conference on AI & Mobile Services, 109-112 (2017)
5. Bucchiarone, A., et. al.: From Monolithic to Microservices: An Experience Report from the Banking Domain. IEEE Software vol. May/June, 50-55 (2018)
6. Nadareishvili, I., et. al.: Microservice Architecture: Aligning Principles, Practices, and Culture, pp. 62-64. O'Reilly Media, Inc., California (2016)
7. Carneiro Jr., C., Schmelmer, T.: Microservices from Day One: Build Robust and Scalable Software from the Start, pp. 25-26. Florida, Apress (2016)
8. Torre, C. D. L., et. al.: .NET Microservices: Architecture for Containerized .NET Applications, p. 45. Microsoft Corporation, Washington (2019)
9. Daya, S., et. al.: Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservice Approach, p. 26. IBM Redbooks, New York (2015)
10. Pacheco, V. F.: Microservices Patterns and Best Practices, p. 280. Packt Publishing, Mumbai (2018)
11. Indrasiri, K., Siriwardena, P.: Microservices for Enterprise: Designing, Developing, and Deploying. Apress, Florida (2018)
12. Sánchez, C. P., Vilariño, P. S.: PHP Microservices. Packt Publishing, Mumbai (2017)
13. Tutorialspoint.: Microservice Architecture. Tutorials Point (I) Pvt. Ltd., Madhapur (2017)
14. Garrett, O.: NGINX vs Apache: Our View of a Decade-Old Question. https://www.nginx.com/blog/nginx-vs-apache-our-view. NGINX (2015)
15. Gatling Corp.: Gatling Documentation. https://www.gatling.io/docs/current. Gatling Corp (2020)
16. Gallagher, A., et. al.: The Waterfall Model: Advantages, Disadvantages, and When You Should Use It. https://developer.ibm.com/technologies/devops/articles/waterfall-model-advantages-disadvantages. IBM Developer (2019)