# CHAPTER 4
# ANALYSIS AND DESIGN

## 4.1. Analysis

When a developer starts building a web application, the process usually started with building applications with monolithic architecture that contain many services. Monolith architecture chose because it is easier to develop and maintain. While the application is still small, the monolith application works perfectly with a small number of the team. The problem with monolith starts surfacing when the application starts getting bigger, the number of developers increases, and features growth. Since monolith offers no scalability, many companies chose to migrate to microservices. When migrating to microservices, the first step done is to split the monolith services into smaller services. The next step is to divide the data from a single database into separated databases. The database separation is done so that the service could run independently. Even it seems that microservice have so many advantages, they also have drawbacks.

To find out the better architecture performance-wise, API provided by both application need to be tested. The test was conducted by hitting the API several times with many threads concurrently. The test result is latency and request time, success rate needs to be monitored as well since the error occurred during load testing. With the data collected it could be shown which architecture performs better.

## 4.2. Design

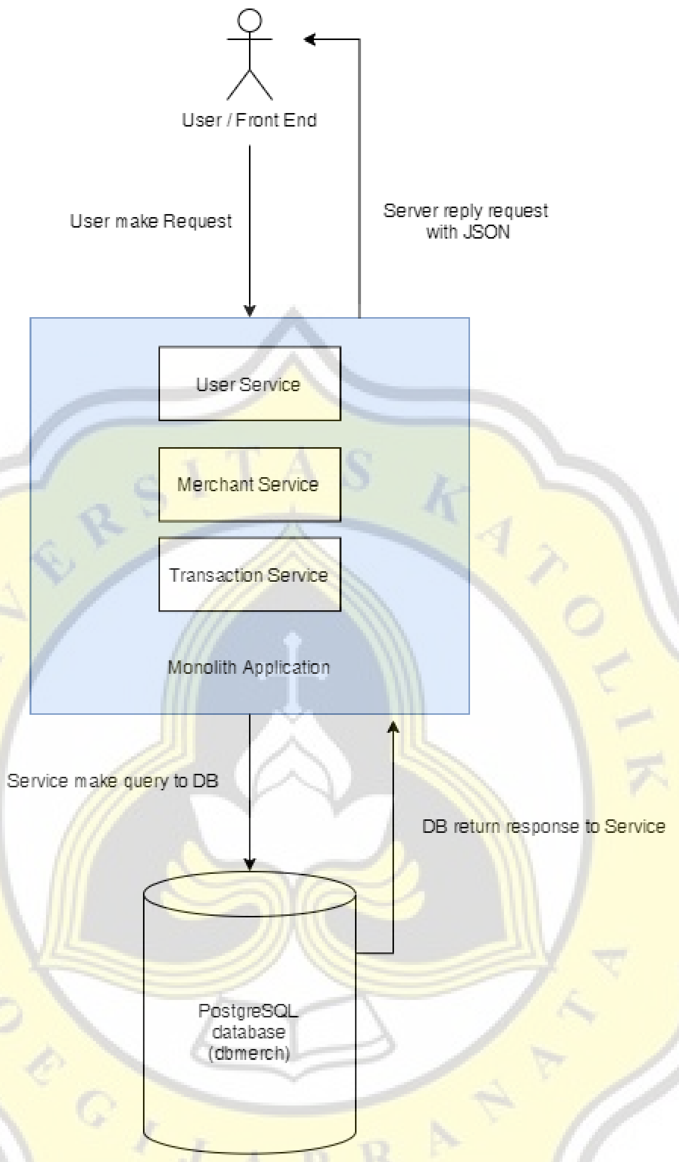


**Figure 4.1** Monolith Architecture Design

Monolith application design based on Go Echo framework where service separated as some package. Endpoint part or usually coded on main package handle HTTP request and routing the request. Usecase package handle the business logic. Every logic that goes through on the application is handled on the use-case part. The repository package handles every communication to the database. In order to achieve smaller latency, the repository part can't have any business logic outside query to the database. The application consists of many services such as merchant service, user service, and transaction service. Every service has its own character and features.

Merchant services have basic create, update and delete or so-called CRUD features. User services have login features that return JSON web tokens or are more popular as JWT for authorization purposes. Transaction services have bulk insert and transaction logic features. Every service is made different in order to compare architecture performance while handling such features. Understanding how the application work, in general, is explained in figure 4.2.

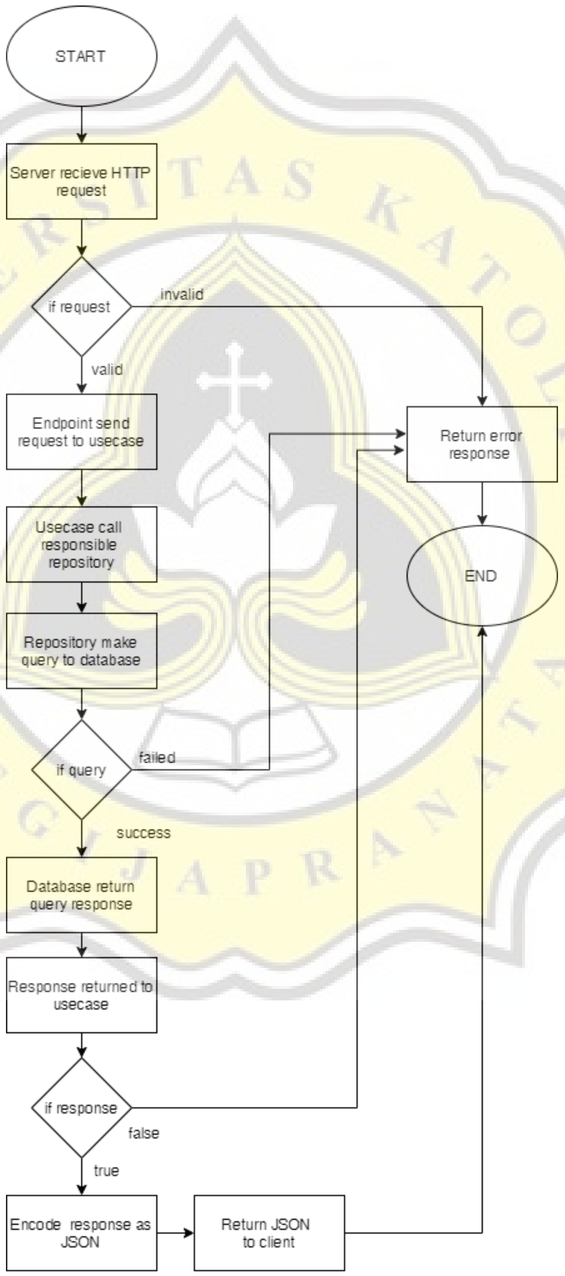

START
Server recieve HTTP request
if request
invalid
valid
Endpoint send request to usecase
Usecase call responsible repository
Repository make query to database
if query
failed
success
Database return query response
Response returned to usecase
if response
false
true
Encode response as JSON
Return JSON to client
Return error response
END

**Figure 4.2** Flowchart Monolith Application Handling Single HTTP Request

The monolith application starts with the user, in this case, could be the front end or directly to the user making HTTP request to the server. When the user hits the API provided, the endpoint layer calls the use-case function as requested. Usecase call repository function to make a query to the shared database which consists of all the data stored by every service. Upon receiving a response from the database, the repository returns the data to the use case function, use-case function sends the data to the endpoint layer to encode data as JSON response. Meanwhile, it is possible to face errors while the program is running. When a layer face error or even the database sends error messages, the server will respond to the request with an error message. The service count is finished when the server sends a return to the client-side whether it is a success or an error.

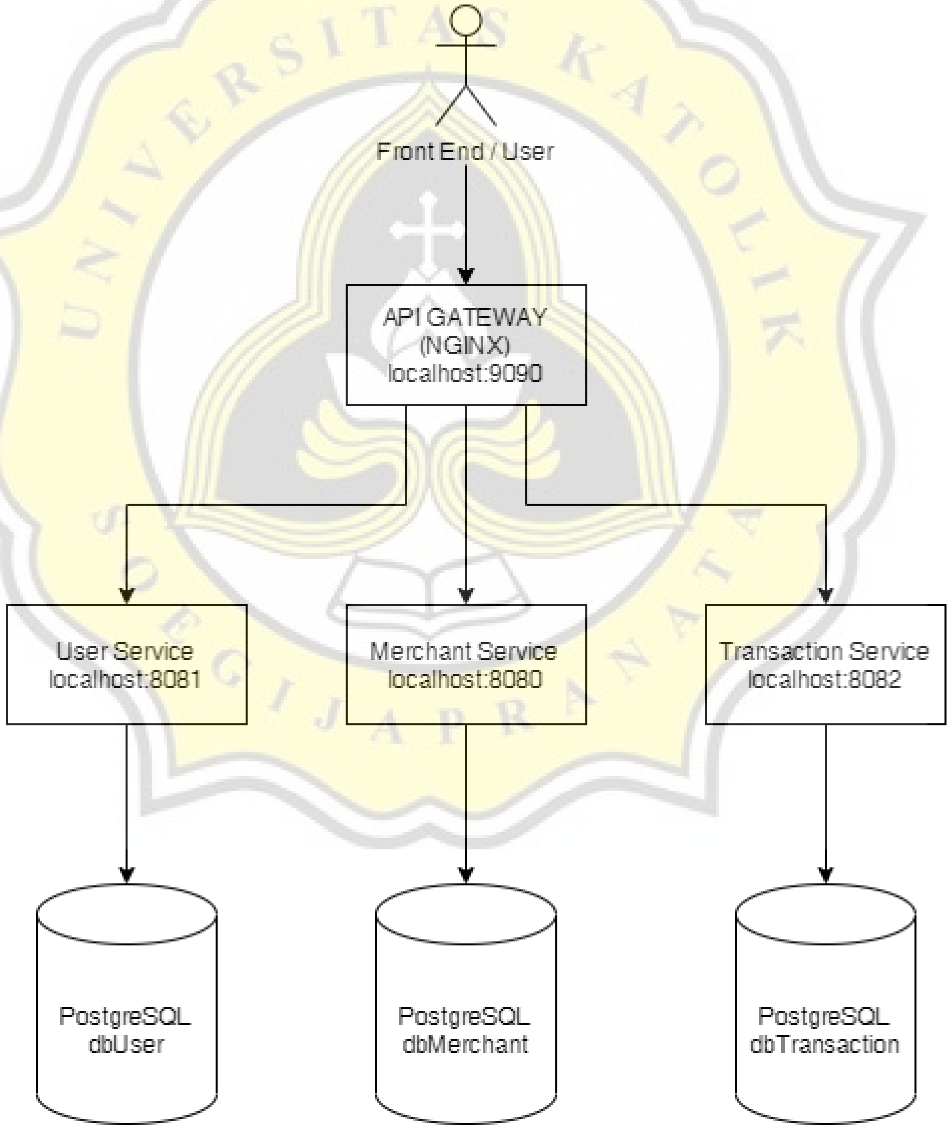

Front End / User
API GATEWAY (NGINX) localhost:9090
User Service localhost:8081
Merchant Service localhost:8080
Transaction Service localhost:8082
PostgreSQL dbUser
PostgreSQL dbMerchant
PostgreSQL dbTransaction

**Figure 4.3** Microservice Application Architecture

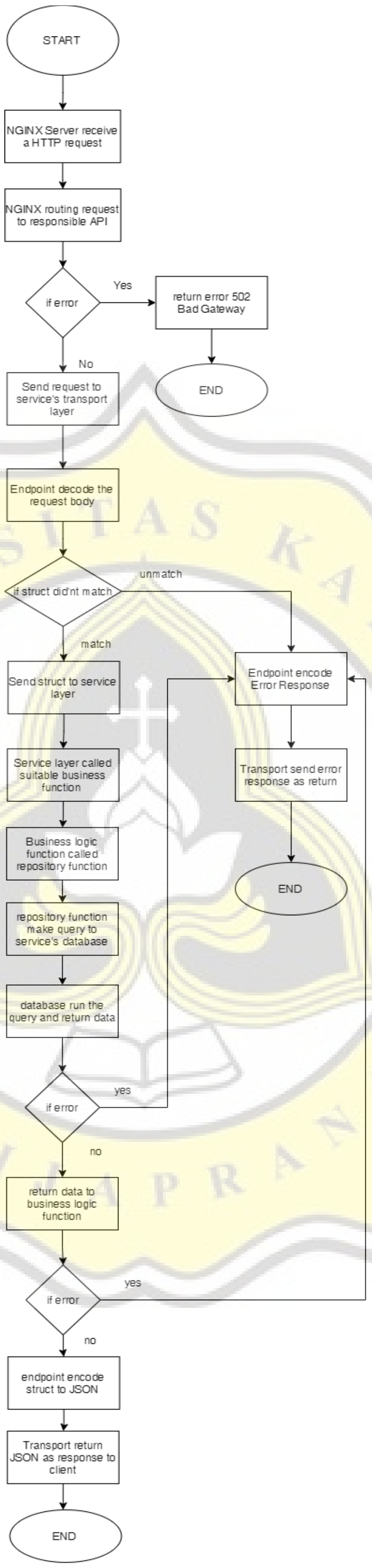

START
NGINX Server receive a HTTP request
NGINX routing request to responsible API
if error
Yes
return error 502 Bad Gateway
END
No
Send request to service's transport layer
Endpoint decode the request body
if struct did'nt match
unmatch
match
Send struct to service layer
Endpoint encode Error Response
Service layer called suitable business function
Transport send error response as return
Business logic function called repository function
END
repository function make query to service's database
database run the query and return data
if error
yes
no
return data to business logic function
if error
yes
no
endpoint encode struct to JSON
Transport return JSON as response to client
END

**Figure 4.4** Microservice Flowchart Handling Single HTTP Request

Microservice application designed following microservices principles where an application runs independently, distributed, and scalable. As explained on figure 4.3, in a microservice architecture, every service act independent while having their own port and database. Even though all the services share the same relational database management system (RDBMS), all of the services have their own unique database. Since the service is distributed, to make a fair test all the services need to be accessible from one port. NGINX server used to reverse proxy services port and combined them so that the services' API listening on a single port.

On this project, a microservice was built based on Go Kit as a microservices toolkit. In the Go Kit principles, services are separated into three main layers such as transport layer, endpoint layer, and service layer. The transport layer is the part where transport processes are done since some cases need more than HTTP API to transport. In this project, the transport layer handles HTTP transport. The endpoint layer is often described as a controller where safety logic is coded. Since the logic functions need to be exposed externally, the endpoint layer is the one that receives the request and converts it to the struct needed. Not only that, but the endpoint layer is also the one that calls the service layer to get the return struct. The service layer is where the business logic lives. Monolith-wise, the service layer is the use-case where all the logic functions are coded.

The microservice application starts when the NGINX server receives a request from a client as drawn in figure 4.4. Upon receiving a request, NGINX forwards the request to a suitable service. While forwarding the request, NGINX errors may occur. If such scenarios happen, NGINX will return an error such as 502 Bad Gateway. On the other hand, when the request success to be forwarded, the transport layer of the service receives the request. From the transport layer, the request will be decoded as a struct. While decoding the request body, it may occur that the struct which is desired doesn't match. If such a scenario happens, the service will return an error message encoded by an endpoint as a response. When the struct match, the service layer will start the work by calling the business logic function. It consists of a logic function and repository function. The logic function will handle all the business logic while the repository handles the communication to the database. Upon getting a response from the database, if no errors are faced, the service layer returns the response to the endpoint layer so that the struct could be encoded to JSON format which later will be used by the transport layer as a response to the client.

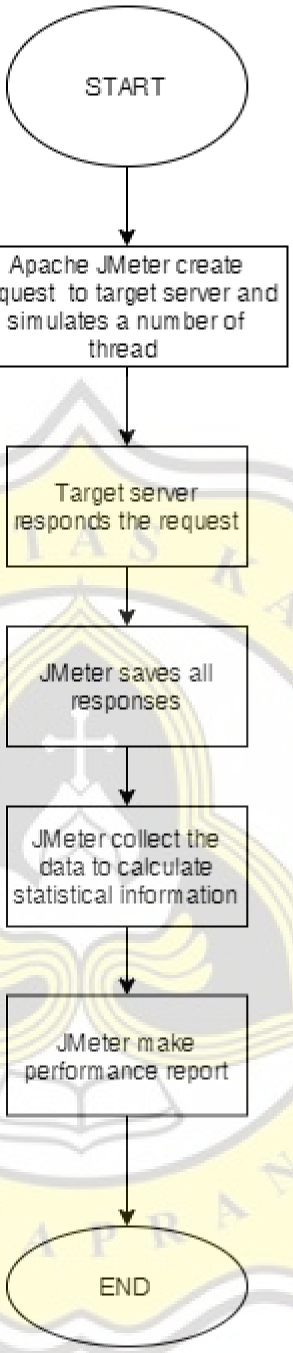
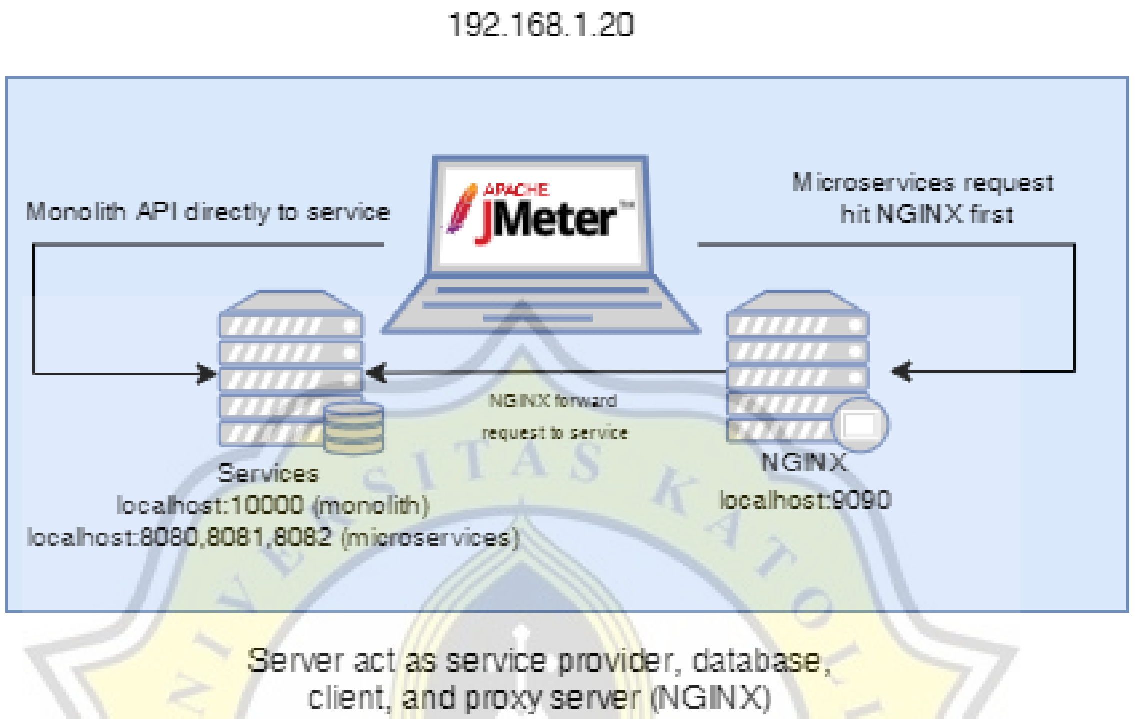

START
Apache JMeter create request to target server and simulates a number of thread
Target server responds the request
JMeter saves all responses
JMeter collect the data to calculate statistical information
JMeter make performance report
END

**Figure 4.5** Flowchart JMeter

In order to compare both architectures, testing software is needed. Since this project tests HTTP server performance, Jmeter is the most suitable testing application. JMeter is capable to make numerous requests concurrently hitting the server's API. Upon getting a response from the server, JMeter saves all of them and collects them so that the data could be calculated and displayed as a performance report. The data used in this project are latency, success rate, and response time.



192.168.1.20
Monolith API directly to service
APACHE JMeter
Microservices request hit NGINX first
NGINX forward request to service
Services
localhost:10000 (monolith)
localhost:8080,8081,8082 (microservices)
NGINX
localhost:9090
Server act as service provider, database, client, and proxy server (NGINX)

**Figure 4.6** Testing Design Where JMeter, Services, Database, and Proxy Server (NGINX) Deployed On a Single Device.

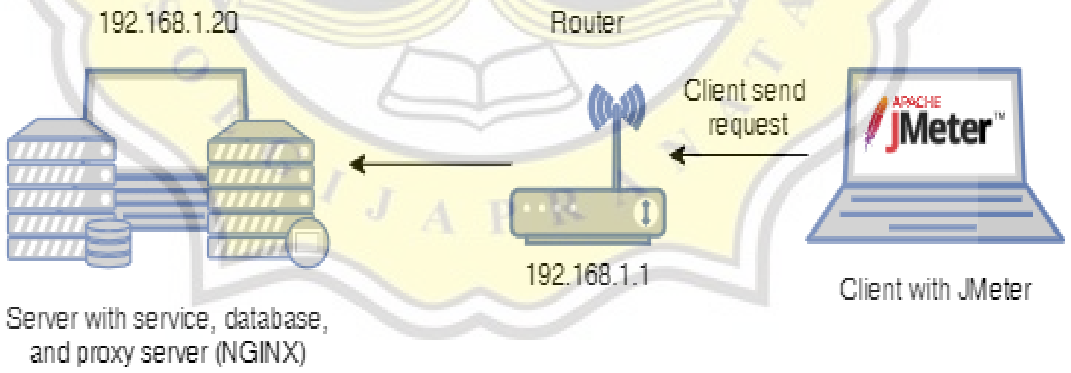


**Figure 4.7** Testing Design Where JMeter Separated From Main Server

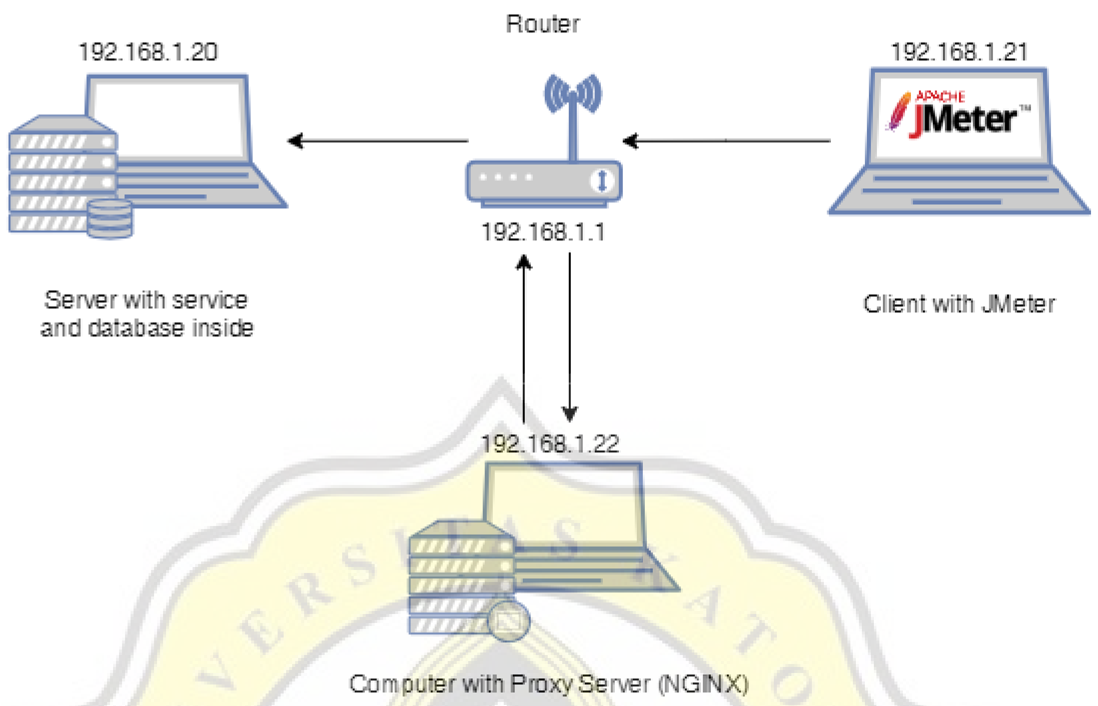


**Figure 4.8** Testing Design Where Services Server, JMeter, NGINX Are Separated

In this project, there are several testing designs conducted. The first testing design is shown in figure 4.6. The design where services, database, JMeter, and NGINX all deployed together in a single device only. In the first design, every process is done locally, when JMeter tests the monolith application, it creates multiple requests, those requests are sent to localhost where services and their database are deployed. On the other hand, when testing microservice applications, to hit the API through a single port, NGINX deployed. The NGINX server deployed on localhost as well. This means that when JMeter creates a request to a microservice application the request is sent to the NGINX server and after that NGINX hits the service's API.

The second testing design is where JMeter is separated on another device as shown in figure 4.7. In this design, a test conducted with JMeter with IP address 192.168.1.21 hit the local server on IP address 192.168.1.20. When JMeter tests the monolith application, JMeter directly makes a request to the monolith application's API. Meanwhile, when testing microservices, JMeter makes a request to the NGINX server port first before NGINX forwards the request to the responsible service. NGINX server in the second design is deployed together with the services and database. These devices can communicate through a local network and are routed using a router.

For the final testing design, not only JMeter, the NGINX proxy server is separated onto another device as shown in figure 4.8. This mean, testing conducted using three devices which every device have their responsibilities. The first device contains services and a database, this device act as the service provider. Unlike the other design, the server device no longer contains an NGINX proxy server. This first device runs with IP address 192.168.1.20. This address will be the address hit by JMeter for the monolith application. The second device was deployed with the NGINX proxy server. This device serves only a single purpose as a reverse proxy. The second device registered with static IP on 192.168.1.22. The third device's sole purpose is to be the client. On this device, JMeter was deployed for testing. This device registered on static IP address 192.168.1.21. When testing the monolith application, JMeter on the third device directly sends the request to 192.168.1.20:10000. On the other hand, when testing the microservice application, JMeter sends the request to the second device first, meaning the request is sent to NGINX server port which is later forwarded to service API where located on the first device. NGINX on the second device role is to forward the request from a client to service.