Cloud Deployments Using Micro-Services for Digital Monetization Application for Customers

AMALNA JOSE
Student M.Tech. CSE, R. V College of Engineering, Bengaluru

Dr. RAJASHREE SHETTAR
Professor, CSE, R. V College of Engineering, Bengaluru

Abstract--**The market today is heading towards cloud. Market demand is growing starting with customer experience (CX) and business applications heading towards "pay-as-you-go" subscription pricing model for the software requiring a unified and personalized experience. The traditional cloud deployments take the portfolio step wise and fail to unify the portfolio in the right way. This paper proposes to develop a new breed of cloud services providing digital monetization experience to the customer, where the customer can rapidly turn their ideas into new revenue streams at web pace and web scale. The use of microservice architecture drastically reduced the build and test time for the application to minutes when compared to the on-premise deployment which is typically in hours. The debugging of the solution would not result in shutting the entire application rather only the microservice module that has issue this helps to provide service with ideally zero downtime.**

Keywords—**Microservices; Docker; Monetization;**

I. INTRODUCTION

Cloud computing is becoming the cornerstone of the digital economy now. Industries around the globe now use cloud private, public or a combination of the two to deliver their products and services. According to 451 Research which is a leading information technology research and advisory company based out of New York, 40% of all enterprise workloads are running on private or public cloud. Some of the leading companies like Netflix, LinkedIn [1], Facebook, Amazon are built around cloud native applications wherein a distributed microservice based architecture is used to quickly deliver new products and services which are highly cost effective and responsive to the market demands.

By breaking up applications into distinct single-purpose services or micro-services [2] that are loosely coupled on dependency and identified through an explicit service endpoint, the overall agility and maintainability of the application would be improved to gain competitive benefit in the market today. Microservice[2] based applications allows to circulate work over numerous groups such that each group can take a shot at individual application areas without forcing extra work on others. A Microservice model allows the decay of an application into freely executing administrations [3]. Updating of a single microservice can be done more effortlessly and the resulting update can be production enabled without the requirement for long integration work over the different development teams [4].

A microservice based design normally authorizes a secluded structure. It fits to a continuous delivery software development process. A change to a little piece of the application just requires one or few administrations to be revamped and redeployed. Major advantage of microservice is that it sticks to standards, such as fine-grained interfaces, business-driven advancement, polyglot programming and persistence, lightweight container deployment, ideal cloud application structures, and DevOps (DEVlopment OPerationS) [2] with all-encompassing administration monitoring. This paper focuses on building a monetization system based on microservices architecture to enable the tenant of the system to be able to generate revenue and web pace and web scale.

II. OVERVIEW OF MICROSERVICES ARCHITECTURE

The most ideal approach to describe microservices is to contrast it with the old monolithic method for building applications. In the customary three-tier design, the server-side application plays the part of the center layer, preparing business logic and serving information from database level to customers (web browsers, portable applications, web of things, and so on.). The application is composed as a solitary, brought together code base and everything keeps running in a similar procedure. Scaling is finished by imitating the same solid application on different servers [6].

In microservices architecture [7], the monolithic is disintegrated into various little, granular, freely deployable administrations [8]. The way that these administrations are autonomously deployable is critical; it empowers some of microservices most vital advantages. These services can be created in parallel by various groups, utilizing diverse technology stack that are most appropriate for their purpose. Likewise, as they are autonomously deployed, they can be freely scaled. For instance, an administration that is CPU-substantial, however, need not bother for much memory as it can be scaled on servers furnished with effective processor but a low memory. The services that are required only need to be scaled and not all of them.

In the event that the services are so free and segregated, by what method it would be advisable to share code between them. Indeed, this is an issue of finding the adjust point [8]. While sharing code between services permits reusing existing functionalities and keeping the DRY(Don't Repeat Yourself) rule, it likewise expands the coupling of the services. One arrangement is to share just the specialized libraries, and the basic functionalities can be made into remain solitary services that different services can call to which leads to communication between services.

Correspondence between the microservices should be possible in two primary ways, HTTP and message line (Azure Service Bus, RabbitMQ, Apache Kafka, and so forth) [4]. Fundamentally, HTTP is immediate correspondence and ought to be utilized when there is a need for a quick reaction from the other service. But, the publish/subscribe mechanism of the message queue is more like produce and overlook way.

At long last, as the services are exceptionally granular, customer applications, as a rule, need to interface with different services to get the information they require. To permit changes in the services without influencing the customers, an API Gateway is utilized [9]. The API Gateway is a conceptual layer that covers up away all the microservices, leaving a solitary endpoint for customers to impart. Demands going to the entryway will be proxies/directed to the proper services. The entryway can likewise help to effortlessly screen the use of the services. Figure 1 below shows a model for the microservices architecture.

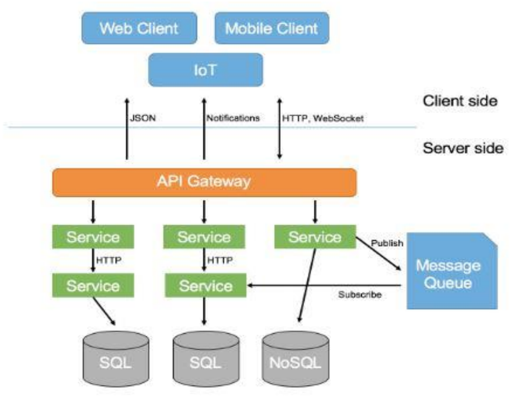

Figure 1 A model for microservices architecture

While the old monolithic architecture had functioned admirably previously, in the present world where applications need to deploy new elements all the time and should be work unceasingly, it's just outdated. Modification in a little part requires testing, rebuilding and redeploying the whole application. Furthermore, since everything keeps running in the same process, an unhandled exception can cut down the entire framework. Microservices architecture, on the other hand is a great deal more adaptable and flexible. The services themselves are extremely simple, concentrating on doing just a single thing admirably so they're easier to test and guarantee a higher quality. Each service can be worked with the most appropriate tools and technologies, permitting polyglot persistence and such. Different developers and teams can deliver autonomously under this architecture. This is helpful for continuous delivery, permitting frequent releases while keeping whatever remains of the framework stable. On the off chance that a service goes down, it will just influence the parts that specifically rely on upon it (if there are such parts). Alternate parts will keep on functioning well.

III. OVERVIEW OF DOCKER CONTAINERS

Docker is a device intended to make it simpler to develop, deploy, and run applications by utilizing containers. Containers enable a developer to bundle up an application with the greater part of the parts it needs, for example, libraries and different dependencies, and ship everything out as one bundle. On account of the container, the engineer can rest guaranteed that the application will keep running on some other Linux machine paying little heed to any modified settings that machine may have that could vary from the machine utilized for composing and testing the code.

Docker is somewhat similar to a virtual machine. In any case, not at all like a virtual machine, instead of making an entire virtual working framework, Docker enables applications to utilize an indistinguishable Linux piece from the framework that they're running on and just requires applications be dispatched with things not officially running on the host PC. This gives a critical execution support and diminishes the measure of the application.

Docker is an instrument that is intended to profit both designers and framework heads, making it a piece of numerous DevOps tool chains. For developers, it implies that they can concentrate on composing code without agonizing over the framework that it will at last be running on. It likewise enables them to get a head begin by utilizing one of thousands of projects effectively intended to keep running in a Docker compartment as a piece of their application. For operations staff, Docker gives adaptability and conceivably lessens the quantity of frameworks required due to its little impression and lower overhead.

A container is a lightweight, executable and stand-alone bundle of a bit of programming that incorporates everything expected to run it: code, runtime, framework instruments, framework libraries, settings. Accessible for both Linux and Windows based applications; containerized programming will dependably run the same, paying little respect to the earth. Holders separate programming from its environment, for instance contrasts amongst advancement and organizing situations and help decrease clashes between groups running distinctive programming on a similar framework.

IV. METHODOLOGY

The product is divided into different functional areas like billing, care, payments, accounts-receivable etc. Each functional area is developed as an individual self-contained micro-service. Functionality required for each microservice is identified and the appropriate design is developed for attaining each of the required functionality. The communication among the microservices is with the use of REST based calls or Kafka messaging events. As per the design depending on the priority of the functionalities the required modules are developed. Each microservice is then run in dockerised containers. So the approach for development is to initially develop the basic needed module. So the approach is to develop modules for configuring a payment gateway which allows the administrator to add, modify and delete the payment gateway provider. Next the modules necessary for adding a card is developed. The next module required is to perform a sale operation for buying a product by the user of the product. Once the microservices with basic functionality are up and running the next step is to integrate the developed components to complete the complete buying and billing journey. The integration can take place either via the REST calls or by using a messaging queue. The approach adopted is to follow REST calls when only two microservices are involved but if there are more services then the approach of messaging is followed.

V. INFERENCE AND RESULT

Based on the results obtained it can be inferred that the applications built in a microservice approach cater to fast development, testing and deployment when compared to the monolithic architecture. The build time for each microservice is around 10 minutes in case of first time build as the necessary packages needs to be downloaded but the later builds have a reduced time varying from 5-10 minutes when compared to the build time of monolithic application which ranges from 4-6 hours. The build for the microservices include the running of various tests like unit tests, integration test and so on. But the monolithic application testing is a separate task which typically takes few hours to complete. The on premise deployment of the product will also involve lot of tasks and is time consuming to set up the product. From the perspective of a customer the overall deployment cost is reduced as it is a SaaS based product. Since each component is developed as individual microservice the customer can opt to the services that are essential to him rather than opting for the entire package in case of the on-premise deployment.

VI. CONCLUSION AND FUTURE WORK

This work mainly aimed at developing a cloud native solution for the billing and revenue management of the tenant organization. With the aim of providing the communication service provider or any billing and charging system to quickly try and launch their product service.

The current development does not support a notification system to the customer hence a correspondence service is required to provide the notification or messaging support for the system. The ability to support refund and adjustment is not accommodated. Currently the system supports only sale operation it can be extended to support the authorization capture and void operation of payment.

VII. ACKNOWLEDGEMENT

The authors thank Dr S Sridhar Director RVCT RVCE Bangalore for communicating this paper for publication in this journal.

VIII. REFERENCES

[1] Alex Feinberg, Phanindra Ganti, Lei Gao, “Data Infrastructure at LinkedIn”, IEEE 28th International Conference on Data Engineering, Washington DC,2012, pp. 1370 – 1381.

[2] Johannes Thönes, “Microservices”, IEEE Software, vol. 31, (1), 2015, pp. 116 – 116.

[3] David Jaramillo, Duy V Nguyen, Robert Smart, “Leveraging microservices architecture by using Docker technology”, Southeast Con, Hursley, United Kingdom, 2016, pp. 1-5.

[4] Rami Bahsoon, Sara Hassan, “Microservices and Their Design Trade-offs: A Self-Adaptive Roadmap”, IEEE International Conference on Services Computing, Birmingham, UK, 2016, pp.813 – 818.

[5] Armin Balalaie and Abbas Heydarnoori, “Microservices Architecture Enables DevOps: Migration to a Cloud-NativeArchitecture”, IEEE Software, vol. 33, (3), 2016, pp. 42-52.

[6] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara, “Workload Characterization for Microservices”, IEEE International Symposium on Workload Characterization (IISWC), Tokyo, 2016, pp. 1-10.

[7] Maria Fazio, Antonio Celesti, Rajiv Ranjan, “Open Issues in Scheduling Microservices in the Cloud”, Cloud Computing, vol. 3, (5), 2016, pp. 81- 88.

[8] Alan Sill, “The Design and Architecture of Microservices”, Cloud Computing, vol. 3, (5), 2016, pp. 76-80.

[9] Mazin Yousif, “Microservices”, Cloud Computing, vol. 3, (5), 2016, pp. 4-5.