PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Pós-graduação em Arquitetura de Software Distribuído

Samuel Martins da Silva

COMPARATIVO ENTRE ARQUITETURA DE MICROSSERVIÇOS E NANOSSERVIÇOS

Belo Horizonte
2018

Samuel Martins da Silva

COMPARATIVO ENTRE ARQUITETURA DE MICROSSERVIÇOS E NANOSSERVIÇOS

Dissertação apresentada na Pós-graduação em Arquitetura de Software Distribuído da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de Arquiteto de software distribuído .

Orientador: Prof. Marco Aurélio de Souza Mendes
Coordenador: Prof. Tadeu Faria

Área de concentração: DevOps

Belo Horizonte
2018

Samuel Martins da Silva

COMPARATIVO ENTRE ARQUITETURA DE MICROSSERVIÇOS E NANOSSERVIÇOS

Dissertação apresentada ao Pós-graduação em Arquitetura de Software Distribuído da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de Arquiteto de software distribuído .

Área de concentração: DevOps

Prof. Marco Aurélio de Souza Mendes(Orientador) – PUC Minas

Prof. Tadeu Faria(Coordenador) – PUC Minas

Belo Horizonte, 30 de Agosto de 2018

RESUMO

O presente trabalho avaliou e comparou a implantação e desenvolvimento de duas arquiteturas: microsserviços e nanosserviços. Para isso, foi contextualizado o atual cenário de desenvolvimento de softwares escaláveis, bem como a evolução dos padrões arquiteturais monolíticos para padrões de microsserviços. Para tal avaliação, foi desenvolvido um nanoserviço e um microserviço, ambos implantados na Azure. Tanto o microserviço quanto o nanoserviço desenvolvido fazem a mesma tarefa: cadastrar um registro em um banco de dados CosmosDB, da Azure.

Palavras-chave: Azure. Microsserviços. Nanosserviços. Serverless.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Grandes empresas na internet como Netflix, Amazon e Linkedin estão utilizando o padrão arquitetural de microsserviços para o *deploy* de grandes aplicações na nuvem. Entretanto, em alguns cenários, além de ganhar agilidade, escalabilidade e manutenibilidade, o custo da infraestrutura para manter esse tipo de padrão é um fator crítico. Sendo assim, novas abordagens de desenvolvimento de software surgiram nos últimos anos para tentar amenizar este problema, como é o caso do padrão arquitetural de nanosserviços (serverless). Grandes empresas entraram nesse ramo oferecendo soluções para esse tipo de arquitetura, como é o caso da Amazon (*AWS Lambda*), Google (*Cloud Platform*), IBM (*Bluemix's OpenWhisk*) e Microsoft (*Azure function*).

1.1 Objetivos

O objetivo desse trabalho é fazer uma análise comparativa entre duas estratégias de arquiteturas de software: a arquitetura de microsserviços e nanosserviços.

1.1.1 Objetivos específicos

- Fazer uma prova de conceito escrita em NodeJs, demonstrando uma aplicação em um caso de uso real utilizando a arquitetura de microsserviços, e a mesma prova de conceito utilizando uma arquitetura de nanosserviços;
- Mensurar o custo de configuração e manutenção das duas arquiteturas;
- Avaliar a dificuldade de desenvolvimento;
- Avaliar a escalabilidade;

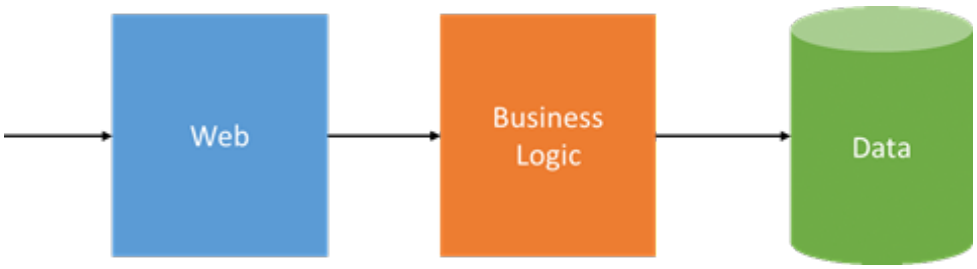
2 REVISÃO DA LITERATURA

Neste capítulo é feita a revisão da literatura que serviu de base para a análise e construção do trabalho. Inicialmente, é contextualizado o cenário de desenvolvimento de software utilizando uma arquitetura monolítica e, sem seguida, são abordadas as arquiteturas de microsserviços e nanosserviços.

2.1 Arquitetura de microsserviços

Para um entendimento mais claro da arquitetura de microsserviços, será feita uma comparação com um modelo bastante conhecido e amplamente utilizado, que é a arquitetura monolítica. Uma arquitetura monolítica é o tipo de aplicação onde todo o código-fonte está contido em um único lugar. Geralmente, é dividido em três camadas: **web**, **negócios** e **dados**, conforme mostrado na Figura 1

Figura 1 – Arquitetura de três camadas de uma aplicação monolítoca



Fonte: Russinovich, 2016

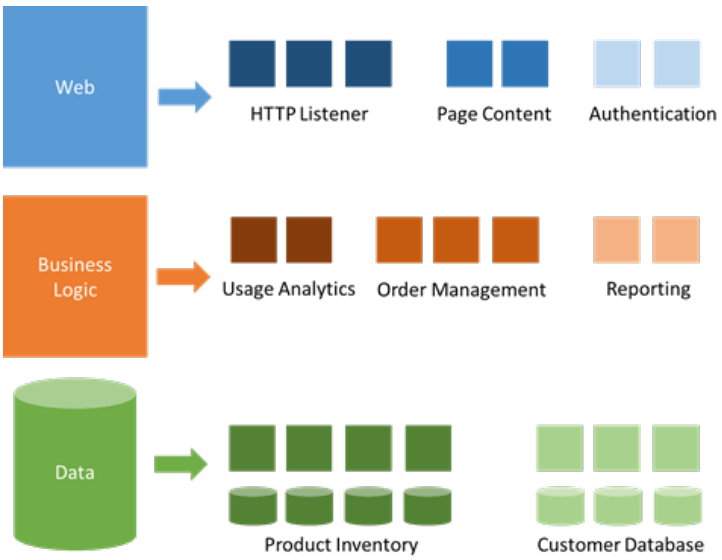
A camada web é responsável pela apresentação ao usuário. Sendo assim, nela é contida os arquivos JavaScript, HTML e CSS da aplicação. A camada de negócios é encarregada de armazenar toda a lógica da aplicação e todas as regras para que os fluxos aconteçam. Normalmente, nessa camada são encontradas, além da inteligência do sistema, chamadas a APIs (internas ou externas) e demais regras para apresentação na camada web. Por último, a camada de dados, onde são encontradas apenas as classes responsáveis pela comunicação direta com as bases de dados (Oracle, MySql, SQL Server e etc).

Com a evolução dos conceitos e paradigmas arquiteturais e o grande avanço de empresas da internet, as aplicações começaram a ter uma complexidade cada vez maior, as equipes de desenvolvimento foram crescendo e os papéis na carreira de T.I ficaram mais bem divididos. Daí surge uma necessidade constante de evolução, manutenção e escalabilidade de aplicações, o que fez com que novas abordagens de arquitetura fossem sendo introduzidas no mercado, como é o caso da arquitetura de microsserviços. Diferentemente dos softwares chamados de monolíticos, onde toda a base está contida em um único lugar, a aplicação desenvolvida em microsserviços é dividida em vários serviços que são independentes. Embora não haja uma definição precisa desse estilo arquitetural, há algumas características comuns, como o *deploy* automatizado e independente por serviço, testes independentes e utilização livre de linguagens para cada serviço (FOWLER; LEWIS, 2014).

2.1.1 Componentização por meio de serviços

O termo microsserviços enfatiza que a aplicação em questão precisa, necessariamente, ser uma composição de serviços. Sendo assim, a arquitetura definida na Figura 1 decomposta em microsserviços ficaria como mostrado na Figura 2.

Figura 2 – Arquitetura de uma aplicação em microsserviços



Fonte: Russinovich, 2016

A componentização da aplicação ocorre por meio de serviços, que podem comunicar entre si por meio de APIs REST. Uma característica desses serviços é que eles possuem um baixo nível de acoplamento e interdependência. Sendo assim, o desenvolvedor deve ser capaz de fazer o deploy de um serviço sem a necessidade de alterar um outro serviço. "O padrão Arquitetura de microsserviços impõe um nível de modularidade que, na prática, é extremamente difícil de obter com uma base de código monolítica. Consequentemente, serviços individuais são muito mais rápidos de desenvolver e muito mais fáceis de entender e manter."(RICHARDSON, 2015)

2.1.2 Smart endpoints e dumb pipes

Um ponto importante a ser considerado na arquitetura de microsserviços é o conceito de *Smart endpoints & Dumb pipes*, que em tradução livre seria "Endpoints inteligentes e canos burros". Em um microserviço, a comunicação entre os componentes/serviços é geralmente feita por meio do padrão *request/response*, utilizado no protocolo HTTP. Nessa abordagem, a inteligência do processamento acontece somente dentro dos serviços, enquanto a comunicação utiliza mecanismos simples (REST ou algum protocolo leve de troca mensagens) (SCHUSTER et al., 2015)

2.1.3 Tolerância a falhas

Como dito no início deste capítulo, microsserviços devem ser independentes de outras partes da aplicação, além de possuir um baixo acoplamento e boa autonomia. Uma consequência dessa arquitetura é que a aplicação passa a ter uma boa tolerância a falhas. Supondo que em uma aplicação de uma loja online exista diversos microsserviços, sendo

um deles o serviço de carrinho de compras. Se em algum momento o serviço de login falhar, o serviço de carrinho de compras deve continuar operando normalmente sem que suas funcionalidades sejam afetadas. Esse padrão arquitetural resolve um outro problema, que consiste na monitoração e identificação rápida de falhas em funcionalidades. "Dado que os serviços podem falhar a qualquer momento, é importante ser capaz de detectar falhas rapidamente e, se possível, restaurar o serviço automaticamente"(FOWLER; LEWIS, 2014).

2.2 Arquitetura de nanosserviços

A constante busca por redução de custos de infraestrutura e ganhos em performance nas aplicações web fez com que novos conceitos arquiteturais fossem difundidos, a ponto de virarem produtos e serviços oferecidos por grandes empresas da internet. A arquitetura de nanosserviços veio com o propósito de reduzir preocupações com custos e configurações de infraestrutura e servidores, divergindo da arquitetura de microsserviços, onde a infraestrutura ainda é mantida pelo desenvolvedor. A partir dessa necessidade, alguns serviços como a Azure Function ou AWS Lambda oferecem a possibilidade de construir uma aplicação ainda menor que um microserviço, e sem a necessecidade de um servidor, o que é chamado de aplicações serverless.

O termo ou *serverless* se refere a um assunto totalmente emergente no mercado de computação em nuvens. Esse conceito pode ser definido como aplicações em que a lógica do lado do servidor ainda é escrita pelo desenvolvedor mas, ao contrário das arquiteturas tradicionais, é executada em contêineres efêmeros de computação sem estado, acionados por evento e totalmente gerenciada por terceiros (ROBERTS, 2018). Contêineres efêmeros podem ser vistos como contêineres com pouco tempo de duração, que são ligados ou acionados apenas para a execução de uma tarefa e desligados logo após o seu término. A definição da Amazon ainda diz que "[...] As aplicações sem servidor não exigem que você assuma o provisionamento, a escalabilidade e o gerenciamento de servidores". (AMAZON, 2018).

Esse novo conceito de arquitetural ainda pode ser chamado de FaaS (*Function as a Serice* ou Função como Serviço), pois cada tarefa executada em um serviço é desmembrada em uma função, pequena o suficiente para rodar em um contêiner efêmero.

Na Figura 3, é exemplificado uma arquitetura baseada em eventos, onde uma função é disparada através de um hook do *Github*. A cada novo *commit*, uma nova mensagem é enviada a um canal no *Slack*.

Figura 3 – Fluxo de chamada de uma função no Google Cloud Functions

Fonte: Serverless. . . , 2018

Embora este seja um conceito muito novo, existem algumas características comuns em arquiteturas de nanosserviços em contêineres serverless, como ausência de estado,

limite do tempo de execução de uma função e latência de inicialização.

2.2.1 Ausência de estados

Funções como serviço possuem algumas limitações quanto a persistência de dados em memória após múltiplas chamadas. Esse tipo de serviço é também nomeado de *stateless*, justamente pelo fato das funções não armazenarem estado de uma chamada para outra. Dados que devem ser persistidos precisam ser tratados fora da instância da função, utilizando algum outro serviço próprio como *Redis* ou *Amazon Step Functions*.

2.2.2 Latência de inicialização

Um outro fator comum em contêineres serverless é a latência no tempo de inicialização. Como dito no início deste capítulo, os contêineres efêmeros são inicializados e destruídos logo ao término da execução de uma função. Isso significa que para cada chamada de função é necessário inicializar o contêiner. "Essa abordagem conhecida como início "frio", requer a inicialização do contêiner com as bibliotecas necessárias, que podem gerar uma latência de inicialização indesejada para a execução da função". (AKKUS et al., 2018).

2.2.3 Tempo de execução

O tempo de execução de cada função é um fator importante a ser levado em conta no desenvolvimento de um nanoserviço, pois o custo da aplicação passa diretamente pelo tempo gasto na execução de uma função. Quanto mais demorada a função, maior o custo. Os provedores desse tipo de serviço também possuem alguns limites de execução. No caso da AWS Lambda, o *timeout* é de cinco minutos. Segundo Roberts (2018), por esse fator, certas tarefas de longa duração não são adequadas para uma FaaS function sem uma grande reestruturação no sistema.

2.3 Trabalhos Relacionados

Villamizar et al. (2016) fizeram um trabalho onde foram feitas análises comparativas de custos de uma aplicação web desenvolvida e entregue utilizando o mesmo cenário com três abordagens diferentes: arquitetura monolítica, arquitetura de microsserviços com infraestrutura gerenciada pelo cliente e arquitetura de microsserviços gerenciada por um provedor de serviços na nuvem (no caso, a *AWS Lambda*). Neste trabalho foi descrito os desafios do desenvolvimento e deploy de uma arquitetura de microsserviços em comparação com uma arquitetura monolítica, apontando os custos de desenvolvimento e manutenção das arquiteturas.

Para avaliar as implicações da utilização de uma arquitetura de microsserviços, foi utilizado um caso de uma aplicação real onde o software ajuda nos processos de negócios de uma empresa, gerando e consultando planos de pagamento de empréstimos de dinheiro entregues por uma instituição aos seus clientes. Para efeitos de comparação, foram considerados dois serviços de uma lista de diversos outros na aplicação original.

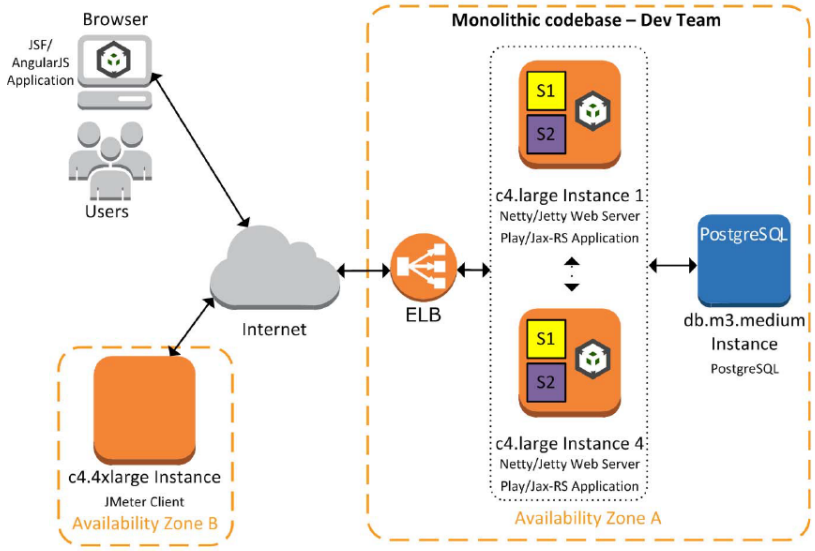
Para a arquitetura monolítica, um serviço foi desenvolvido utilizando o padrão MVC, sendo o *back end* responsável por uma API Rest, e o front end sendo responsável por consumir essa API. O importante nessa arquitetura foi simular um cenário de uma

arquitetura monolítica onde toda a base de código fosse encontrada em um único lugar, com o deploy sendo feito de uma só vez. Nessa abordagem a aplicação pôde ser entregue em um único ambiente onde escalabilidade de qualquer parte do projeto fica restrita aos recursos de um único servidor ou em um ambiente multi-servidor.

Na arquitetura de microsserviços operada na nuvem pelo cliente, cada um dos dois serviços deveria ser desenvolvido num padrão de três camadas de forma independente, utilizando diferentes *stacks* de desenvolvimento. O serviço 1 foi responsável apenas por gerar novos planos de pagamento, sem a necessidade de persistência e o segundo serviço, retorna os dados completos de pagamento, precisando assim de uma camada de dados.

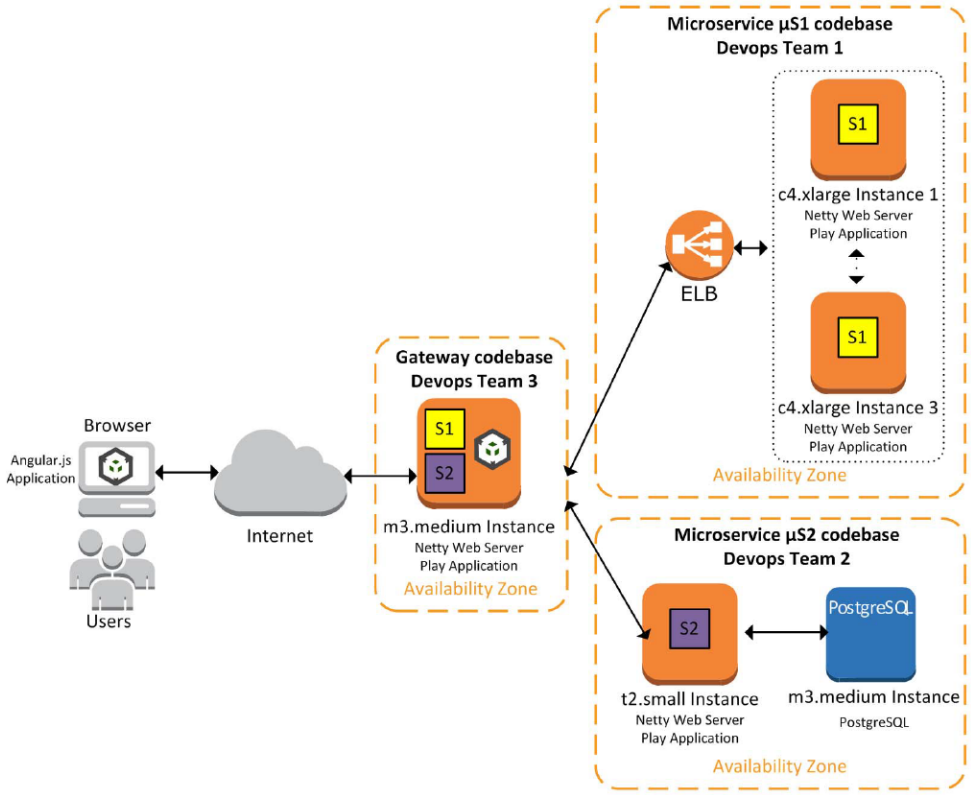
Na arquitetura operada pela *AWS Lambda*, dois *gateways* de serviços precisaram ser implementados como funções separadas na *Amazon*. Ambas as funções eram responsáveis por receber as requisições do navegador, consumir microsserviços via REST, recuperar dados de outras funções e retornar resultados para os usuários finais. A infraestrutura de build e deploy da arquitetura monolítica, microsserviços operada pelo cliente e microsserviços operada pela amazon são mostradas nas Figuras 4, 5 e 6, respectivamente.

Figura 4 – Implantação da arquitetura monolítica na Amazon Web Services



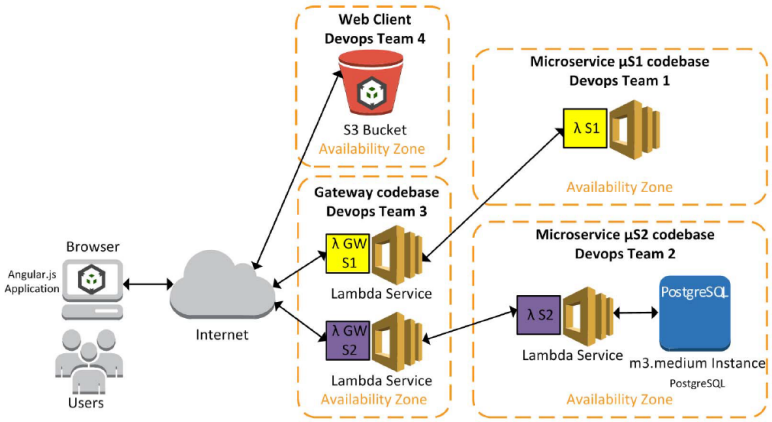
Fonte: Villamizar et al., 2016

Figura 5 – Implantação da arquitetura de microsserviço na Amazon Web Services



Fonte: Villamizar et al., 2016

Figura 6 – Implantação da arquitetura na Amazon Web Services Lambda



Fonte: Villamizar et al., 2016

Por meio do cenário proposto, foi comprovado que a arquitetura de microsserviços conseguiu reduzir em cerca de 13% dos custos de manutenção. No entanto, o uso de serviços em nuvem como o AWS Lambda, projetado exclusivamente para implantar microsserviços em um nível mais granular (por solicitação / função HTTP), teve um custo de infraestrutura reduzido em até 77,08%.

Um outro trabalho pertinente ao tema deste artigo foi a análise de ambientes de produção sem servidores, feita por Lee, Satyam e Fox (2018). Nesta análise, foi comparado a taxa de transferência (*throughput*), uso de rede e entrada/saída de arquivos entre os principais provedores de soluções serverless no mercado, sendo eles o *Google Functions*, *AWS Lambda*, *Azure Functions* e *IBM OpenWhisk*. Após todos os testes, a conclusão foi de que a Amazon Lambda possui uma maior elasticidade em relação aos seus concorrentes no tocante a performance da CPU, largura de banda e taxa de transferência em chamadas concorrentes. Um outro ponto destacado pelos autores foi de que a computação serverless pode ser vantajosa para casos de processamento de dados distribuídos caso as tarefas sejam pequenas o bastante para executar em uma instância de função com 1.5GB a 3GB de limite de memória e limite de execução entre 5 a 10 minutos.

3 METODOLOGIA

Para alcançar os objetivos definidos este trabalho se apoiou numa revisão de literatura relacionada a desenvolvimento web, padrões arquiteturais e alguns conceitos de implantação. Foi necessário mostrar a evolução dos padrões arquiteturais e conceituar as diferenças entre arquitetura de Microsserviços e Nanosserviços. Para auxiliar a comparação entre as duas arquiteturas, foi desenvolvido um microserviço em NodeJs, utilizando o CosmosDB da Azure e o Mongoose para modelagem e conexão ao banco. O microserviço é baseado nos padrões REST, portanto foi utilizado também o express para o gerenciamento de rotas da api. O trecho de código a seguir demonstra um exemplo de como os dados são inseridos no banco por meio de uma requisição POST na rota */api/project/*:

Listing 3.1 – Código fonte da rota /api/project/

```
import express from 'express'
import { Project } from './models/project'
const projectRouter = express.Router()

projectRouter.post('/', async (req, res) => {
    const { name } = req.body

    try {
        const newProject = await new Project({ name }).save()
        res.send({ project: newProject })
    } catch (error) {
        res.send({ error })
    }
})
```

O microsserviço foi implantado em um container docker, utilizando o Dockerfile como mostrado a seguir:

Listing 3.2 – Código fonte do Dockerfile

```
FROM node:8

# Create app directory
WORKDIR /usr/application

COPY package*.json ./
RUN npm install

COPY . .
EXPOSE 8080

CMD [ "npm", "start" ]
```

Foi desenvolvido também um nanoserviço do tipo *httpTrigger*, implantado no azure function, que executa a mesma tarefa do microserviço mostrado anteriormente. O trecho de código a seguir mostra a implementação de um nanoserviço que adiciona um projeto em um banco de dados CosmosDB utilizando o mongoose como intermediador da conexão entre a aplicação e o serviço da azure:

Listing 3.3 – Código fonte do nanoserviço

```
module.exports = function (context, req) {
    mongoose.connect(connectionString).then(function(){
        var Project = mongoose.model('Project', {
            name: String
        })

        new Project(req.body.name).save().then(function(newProject) {
            context.res = {
                body: newProject,
                headers: {
                    'Content-Type': 'application/json'
                }
            }

            context.done();
        })
    })
};
```

Com isso, têm-se as duas provas de conceito de arquiteturas implantadas na nuvem conforme foi proposto o objetivo deste trabalho.

4 CONCLUSÃO

Há uma grande diferença de custo de implantação e manutenção das duas arquiteturas. Isso devido ao fato das propostas de ambas as arquiteturas serem diferentes, como descrito no capítulo 2 deste trabalho. Com um plano básico para suportar a arquitetura de microsserviços proposta, o custo ficou em R$275.92 em um período de 30 dias. O pacote de serviços inclui os seguintes produtos: *Container Registry*, *Azure Cosmos DB* e o *App Service*.

Para a manuteção da arquitetura de nanosserviços proposta, o custo é consideravelmente menor pois a cobrança em cima da Azure Function é feita por número de requisições/tempo de processamento da requisição. Portanto, em uma simulação feita no Azure Princing Calculator onde eram feitas 10 milhões de requisições com tempo de processamento de 1 segundo cada, o custo total foi de R$128.68 (já incluso o Azure Cosmos DB).

A comunicação entre serviços na arquitetura baseada em microsserviços pode ser feita de diversas formas dentro de um mesmo contexto. Pode acontecer por meio de mensagens, por meio de HTTP ou qualquer outro protocolo. Comunicar entre nanosserviços já não é uma tarefa tão fácil. O fato de funções serem *stateless* faz com que seja necessário um gerenciador de estados externo a função escrita. Isso pode dificuldade o desenvolvimento de uma aplicação em alguns cenários.

No quesito escalabilidade, a arquitetura de microsserviços torna-se um pouco mais complexa e cara no contexto proposto, pois muitas vezes é necessário manter uma infraestrutura com recursos que não estão sendo utilizados. Um outro fato é que, configurar um ecossistema de microsserviços contêinerizados e escaláveis não é uma tarefa fácil, visto que seria necessário configurar um orquestrador como Docker Swarm ou Kubernetes. Já na arquitetura de nanosserviços, a escalabilidade acontece de forma automática, visto que a cobrança fica restrita apenas ao uso ou não da função implantada.

REFERÊNCIAS

AKKUS, I. E. et al. ***SAND: Towards High-Performance Serverless Computing***. 2018. Acesso em: 19 ago. 2018. Disponível em: <https://www.usenix.org/system/files/conference/atc18/atc18-akkus.pdf>. 10

AMAZON. *Aplicativos e computação sem servidor*. 2018. Acesso em: 19 ago. 2018. Disponível em: <https://aws.amazon.com/pt/serverless/>. 9

FOWLER, M.; LEWIS, J. ***Microservices: a definition of this new architectural term***. 2014. Acesso em: 15 ago. 2018. Disponível em: <https://martinfowler.com/articles/microservices.html>. 7, 9

LEE, H.; SATYAM, K.; FOX, G. C. ***Evaluation of Production Serverless Computing Environments***. 2018. Acesso em: 19 ago. 2018. Disponível em: <https://www.researchgate.net/publication/324362882_Evaluation_of_Production_Serverless_Computing_Environments>. 13

RICHARDSON, C. ***Introduction to Microservices***. 2015. Acesso em: 17 ago. 2018. Disponível em: <https://www.nginx.com/blog/introduction-to-microservices/>. 8

ROBERTS, M. ***Serverless Architectures***. 2018. Acesso em: 19 ago. 2018. Disponível em: <https://martinfowler.com/articles/serverless.html>. 9, 10

RUSSINOVICH, M. ***Microservices: An application revolution powered by the cloud***. 2016. Acesso em: 16 ago. 2018. Disponível em: <https://azure.microsoft.com/pt-br/blog/microservices-an-application-revolution-powered-by-the-cloud/>. 7, 8

SCHUSTER, T. et al. ***Microservice Based Tool Support for Business Process Modelling***. 2015. Acesso em: 17 ago. 2018. Disponível em: <https://www.researchgate.net/publication/282571308_Microservice_Based_Tool_Support_for_Business_Process_Modelling>. 8

SERVERLESS application backends. 2018. Acesso em: 19 ago. 2018. Disponível em: <https://cloud.google.com/functions/use-cases/serverless-application-backends>. 9

VILLAMIZAR, M. et al. ***Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures*** . 2016. Acesso em: 18 ago. 2018. Disponível em: <https://www.researchgate.net/publication/304320076_Infrastructure_Cost_Comparison_of_Running_Web_Applications_in_the_Cloud_Using_AWS_Lambda_and_Monolithic_and_Microservice_Architectures>. 10, 11, 12