Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

LEONARDO MANARIN VEZZONI

ARQUITETURA ORIENTADA A SERVIÇOS E MICROSSERVIÇOS: UM ESTUDO DE CASO

ASSIS/SP
2022

Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

LEONARDO MANARIN VEZZONI

ARQUITETURA ORIENTADA A SERVIÇOS E MICROSSERVIÇOS: UM ESTUDO DE CASO

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação do Instituto Municipal de Ensino Superior de Assis – IMESA e a Fundação Educacional do Município de Assis – FEMA, como requisito parcial à obtenção do Certificado de Conclusão.

Orientando(a): Leonardo Manarin Vezzoni
Orientador(a): Dr. Almir Rogério Camolesi

ASSIS/SP
2022

FICHA CATALOGRÁFICA

V597a Vezzoni, Leonardo Manarin
Arquitetura Orientada a Serviços e Microsserviços: um estuo de caso / Leonardo Manarin Vezzoni. Fundação Educacional do Município de Assis - FEMA - Assis, 2022.
35 f.
Orientador: Prof. Dr. Almir Rogério Camolesi
Trabalho de Conclusão de Curso - Fundação Educacional do Município de Assis - Fema, curso de Ciência da Computação, Assis, 2022.

1.Arquitetura de Software. 2.SOA. 3.Microsserviços
CDD: 005. 1

RESUMO

Tendo em vista o recente aumento na diversidade dos dispositivos eletrônicos, a adoção de uma arquitetura de software escalável e capaz de servir os mais variados dispositivos, independente do sistema operacional ou da interface utilizada, se faz necessária. Assim sendo, o presente trabalho busca realizar um estudo de caso acerca da Arquitetura Orientada a Serviços (SOA) e a Arquitetura de Microsserviços, avaliando em quais situações uma sobressai sobre a outra.

Palavras-chave: Arquitetura de software; Arquitetura Orientada a Serviços; Microsserviços.

ABSTRACT

In view of the recent increase in diversity of electronic devices, the adoption of a software architecture that is scalable and capable of serving the most varied devices, independently of the operating system or the used interface, is necessary. Therefore, the present work seeks to carry out a case study on Service Oriented Architecture (SOA) and microservices architecture, evaluating under which circumstance one architecture might be better than the other.

Keywords: Software architecture; Service Oriented Architecture; Microservices.

LISTA DE ILUSTRAÇÕES

Figura 1 - Diagrama de Classes da Aplicação………………… ……………………………..22
Figura 2 - Diagrama de Caso de Uso …………………………………………………………23
Figura 3 - Configuração de um deployment Kubernetes ……………………………………25
Figura 4 - Configuração de um Serviço para Load Balancer e Configuração do Ingress-NGINX ………………………………………………………………………………….. 25
Figura 5 - Organização das Pastas dos Projetos …………………………………………… 26
Figura 6 - Organização de um Domínio…………………………………………………… 27
Figura 7 - Barras de Navegação………………………………………………………………28
Figura 8 - Tela de Cadastro……………………………………………………………………28
Figura 9 - Tela de Autenticação………………………………………...………………...29
Figura 10 - Tela de Fornecedores cadastrados………………………………………………29
Figura 11 - Tela de Produtos de um Fornecedor.……………………………………………30
Figura 12 - Tela de Detalhes do Produto.……………………………………………………..30
Figura 13 - Tela de Registro de Produto………………………………………………………31
Figura 14 - Tela de Ordens……………………………………………………………………31

SUMÁRIO

1. Introdução……..8
1.1. Objetivo……..8
1.2. Justificativa……9
1.3. Motivação……9
1.4. Perspectivas de Contribuição……9
1.5. Estrutura do Trabalho……..10
2. Arquitetura Orientada a Serviços - SOA……11
2.1. Web Services……..11
2.2. REST……..12
2.2.1. Modelo Cliente-Servidor……12
2.2.2. Client-Stateless-Server……12
2.3. Protocolo HTTP……..13
2.3.1. Mensagens HTTP……..13
2.3.2. Verbos e Status HTTP……14
3. Revisão Bibliográfica de Ferramentas e Tecnologias Utilizadas no Desenvolvimento do Trabalho……15
3.1. NodeJs……15
3.1.1. Gerenciador De Pacotes Node……15
3.1.2. Express……..15
3.1.3. Mongoose……16
3.1.4. TypeScript……16
3.2. Docker……17
3.2.1. Kubernetes……17
3.3. Mensageiros e Comunicação em Microsserviços……18
3.3.1. Comunicação e Persistência de Dados ……18
3.3.2. Soluções Mensageiros ……..19
4. Proposta e Desenvolvimento do Trabalho……20
4.1. Proposta do Trabalho ……..20
4.2. Desenvolvimento do Trabalho……..20
4.3. Diagrama de Classes……..20
4.4. Diagrama de Caso de Uso……22
4.5. Tecnologias Utilizadas……..22
4.6. Infraestrutura da Aplicação……23
4.7. Estrutura do Projeto……..25
4.8. Estrutura dos Domínios……...25
4.9. Telas……26
4.9.1. Barra de Navegação……26
4.9.2. Tela de Cadastro……27
4.9.3. Tela de Autenticação……28
4.9.4. Tela de Fornecedores Cadastrados……..28
4.9.5. Tela de Produtos do Fornecedor……29

4.9.6. Tela de Detalhes do Produto……………………………………..29
4.9.7. Tela de Novo Produto …………………………………………….30
4.9.8. Tela de Ordens …………………………………………………...30
4.10. Comparação das Arquitetura s …………………………..………...31
5. Conclusão ………………. ……………………………………..…………..32
Referências Bibliográficas……………………………………………………33

1 – INTRODUÇÃO

Com o aumento na diversidade e utilização dos diversos dispositivos eletrônicos, desde o tradicional computador de mesa até os *smartphones*, se tornou necessária a adoção de uma arquitetura de software que unificasse as regras de negócio de uma aplicação, com a finalidade de oferecer os mesmos serviços a diferentes clientes (dispositivos), este tipo de arquitetura foi denominada de arquitetura orientada a serviços (SOA).

No momento de se escolher o padrão de arquitetura que será utilizado em uma aplicação, é comum pensar que existe uma solução que resolverá todos os problemas da construção de um *software*, mas a realidade é que existem cenários específicos em que uma se sobressai sobre as outras. Segundo Richardson (2018, p.19), “Se a arquitetura de microsserviços é adequada ou não para sua aplicação, isto dependerá de diversos fatores, portanto, aconselhar que uma aplicação utilize sempre a arquitetura de microsserviços ou que a nunca utilize, são conselhos igualmente ruins”.

Desta forma, pretende-se, realizar um estudo de caso onde serão construídas duas aplicações com o mesmo propósito, porém, utilizando arquiteturas diferentes, assim sendo possível medir os pontos fortes e fracos de cada uma, servindo como material de referência para futuras aplicações.

1.1 – Objetivo

O objetivo do presente trabalho é construir uma mesma aplicação utilizando duas arquiteturas diferentes, a fim de entender melhor as vantagens e desvantagens de cada proposta, contribuindo para a área de engenharia de software, mais especificamente com conhecimentos sobre arquitetura de software, a primeira aplicação será construída utilizando a arquitetura SOA monolítica, já a segunda será com a arquitetura de microsserviços.

1.2 – Justificativa

Seguindo Breivold, Crnkovic e Larsson (2010), a arquitetura de um software é sua fundação e contém praticamente todos seus atributos qualitativos. Com isso, é possível saber como esse software pode evoluir no futuro, dando a base para que este se mantenha competitivo no mercado por um longo período de tempo.

Desta forma, ao se explorar com maiores detalhes as duas arquiteturas, o trabalho se justifica ao poder servir de referência para futuros pesquisadores ou até mesmo arquitetos de software

1.3– Motivação

O presente trabalho tem por motivação, um aperfeiçoamento em aplicações que o autor ajudou a construir e notou que caso uma arquitetura mais específica tivesse sido utilizada, a aplicação teria tido um desenvolvimento mais rápido e eficiente. Deste modo, espera-se que a construção de uma aplicação simples utilizando arquiteturas diferentes, possa contribuir para a área.

1.4 – Perspectiva de Contribuição

Dentro da grande área da engenharia de software, a arquitetura de software é uma subárea explorada a um longo tempo, cuja importância é cada vez maior, como afirmado no artigo de Garlan (2014). Apesar desta subárea conter diversos livros e artigos que abordam o tema, não existem numerosas abordagens práticas, desta forma, o artigo que será produzido, poderá servir como um material de referência com uma abordagem mais didática sobre o assunto.

1.5 – Estrutura do Trabalho

O presente trabalho está organizado em 5 capítulos.

- Capítulo 1 - trata da Introdução, onde é apresentado o tema da pesquisa.
- Capítulo 2 - apresenta uma revisão sobre web services e arquitetura REST.
- Capítulo 3 - realiza uma revisão bibliográfica das ferramentas e tecnologias utilizadas no desenvolvimento do trabalho
- Capítulo 4 - apresenta a proposta e o desenvolvimento do trabalho
- Capítulo 5 - apresenta a conclusão.

2.0 – Arquitetura Orientada a Serviços - SOA

Seguindo Victorino e Bräscher (2009), a arquitetura orientada a serviços é uma arquitetura fracamente acoplada, que consegue decompor os processos de uma organização em serviços (módulos), sendo que os serviços conseguem comunicar-se entre si independente da plataforma e da linguagem em que foram construídos. Sendo um de seus benefícios, prover uma linguagem comum entre o analista de negócio e os desenvolvedores de sistemas de informações. A implementação de uma aplicação que utilize SOA, é possível através de *Web Services*. Sendo que *Web Services*, são componentes de um software que representa um conjunto de funções que podem ser utilizadas através da internet. (Lee, Siew. Chan, Lai. Lee, Eng. 2006).

2.1 – Web Services

Um único web service consiste de um serviço e a descrição deste serviço. O serviço é um módulo de um software que está disponível através da web. Já a descrição do serviço, contém detalhes de sua interface, implementação, categoria, localização e outras propriedades. Os modos mais comuns para a implementação de web services, são através de dois padrões, *Simple Object Acess Protocol* (SOAP) e *Representional State Transfer* (REST) (Tihomirovs, Juris. Grabis Jānis, 2016).

O SOAP, é um protocolo mais antigo que o padrão REST, sendo mais comum, atualmente, encontrá-lo em sistemas legado, o mesmo utiliza predominantemente o XML[1] para formatar os dados e assim, através do HTTP ou outros protocolos, realiza a troca de informações entre os sistemas (Liu, Yan et al, 2008). O REST, é uma abordagem mais recente que utiliza o protocolo HTTP para transmitir informação, sendo que os dados podem estar formatados em XML, JSON, entre outras alternativas.

Para o presente trabalho, optou-se por construir *web services* utilizando REST, já que as tecnologias utilizadas para construir as aplicações possuem maior suporte para o mesmo, facilitando o desenvolvimento.

---

[1] Linguagem de marcação que define como um documento deve ser escrito, utiliza marcadores e espaçamentos para transmitir dados.

2.2 – REST

Web Services baseados no estilo arquitetural REST, surgem, em geral, como uma alternativa mais leve e simples do que o SOAP para se construir sistemas distribuídos na web (Tihomirovs, Juris. Grabis Jānis, 2016). Aplicações que utilizam esta arquitetura, são denominadas de RESTful, entre os princípios, seguindo o trabalho que propõe a arquitetura original (Fielding, Roy. 2000), destacam-se os seguintes.

2.2.1 – Modelo Cliente-Servidor

Este modelo arquitetural é comumente encontrado em aplicações que necessitam realizar troca de informações através da internet, o cliente é sempre o responsável por iniciar a comunicação, e o servidor apenas responde, podendo negar a requisição ou aceitar e enviar uma resposta (Fielding, Roy. 2000, capítulo 3.4.1) . Este conceito é de grande importância na separação de responsabilidade de uma aplicação, já que fica bem definido que as regras de negócio pertencem ao servidor que disponibiliza o mesmo serviço a múltiplos clientes.

2.2.2 – Client-Stateless-Server

Este conceito complementa o modelo cliente-servidor, e define que o servidor não deve possuir uma sessão que preserve algum tipo de estado anterior, deste modo as requisições de um cliente devem sempre conter todos os dados necessários para realizar a requisição sendo que a gerência de estado, caso necessária, deve ser feita pelo cliente (Fielding, Roy. 2000, capítulo 3.4.3) . Este princípio garante diversas vantagens, como a visibilidade de um serviço, já que é possível conhecer a natureza do mesmo de forma modular e um aumento na robustez da aplicação, uma vez que falhas podem ser isoladas por serviços.

2.3 – Protocolo HTTP

Dentro de aplicações RESTful, a troca de informações entre o cliente e o servidor, acontece utilizando o protocolo HTTP (protocolo de transferência de hipertexto). (Fielding, Roy, et al. Hypertext Transfer Protocol, 1999). Este protocolo segue o modelo cliente-servidor sendo que para a web, o cliente costuma ser os navegadores. Para realizar a transferência de dados, é utilizado o protocolo tcp/ip em conjunto com *sockets* que ligam ambas as partes e servem como um meio para realizar a transferência de dados.

2.3.1 – Mensagens HTTP

As mensagens HTTP são divididas em duas categorias, requisições (cliente) e respostas (servidor) essas mensagens compartilham muitos dos atributos porém com nomes e propósitos específicos para cada um dos envolvidos, a primeira linha da mensagem descreve o tipo de requisição (verbos) e o status de conclusão, em seguida vem o cabeçalho, que contém informações descritivas sobre a requisição/resposta, como por exemplo, o host/server, o tipo de conteúdo aceito (text/html, JSON, XML…etc.), entre outras informações que foram necessárias para realizar a comunicação, por último, tem-se o *body*, o body de uma requisição, são os dados enviados, por exemplo, no momento de se enviar um formulário para o servidor, já para a resposta, o body contém as informações obtidas após a conclusão da chamada (Fielding, Roy, et al. Hypertext Transfer Protocol, 1999, capítulo 4).

2.3.2 – Verbos e status HTTP

Dentro das mensagens HTTP, se faz necessário especificar a natureza da requisição, e um número que identifique o resultado final da resposta, deixando mais claro a finalidade do serviço e o resultado obtido, respectivamente. Destacam-se quatro verbos.

- GET: indica a solicitação de um recurso, como por exemplo, os dados de um usuário ou uma lista de restaurantes, aceita o que é chamado de *query string* para realizar filtros específicos, indicados sempre por "?" e em seguida os argumentos, por exemplo, <URL>?order=desc&search=abc;
- POST: indica em geral, a inserção de novos dados dentro de um recurso da aplicação ou utilizado quando existem dados sensíveis que não podem ser exibidos na URL, os dados a serem enviados ficam no body da requisição, sendo utilizado por exemplo, para inserir um novo usuário no sistema, ou logar um usuário existente;
- PUT: utilizado para atualizar ou substituir dados ligado a um recurso, um exemplo seria a atualização dos dados de um usuário dentro do sistema;
- DELETE: utilizado para remover um recurso, um exemplo seria a remoção de um usuário do sistema.

Já para os status de uma resposta, destaca-se os seguintes grupos, de 200 a 299, sucesso de 300-399, redirecionamento, de 400 a 499, erros causados pelo cliente e de 500-599 erros do servidor.

3 – Revisão Bibliográfica de Ferramentas e Tecnologias Utilizadas no Desenvolvimento do Trabalho

3.1 – NodeJs

Programa que permite a execução de código JavaScript (JS) *server-side*, não mais limitando JS aos navegadores, foi construído utilizando como base a V8 Engine do Chrome, sendo que seus principais módulos são escritos em C e C++, *Node* foi construído de modo que suas funcionalidades sejam assíncronas e não bloqueantes, utiliza a biblioteca libuv, construída em c que se baseia em loops de evento para gerenciar Entradas e Saídas (E/S) assíncronas. Existe um único *thread* que em conjunto com o loop de eventos, lida com a execução do código, a junção destes conceitos permite que uma aplicação construída em Node tenha uma alta escalabilidade, podendo servir múltiplos clientes sem interrupção (Sobre Node.js, 2022).

3.1.1 – Gerenciador De Pacotes Node

Um dos principais motivos para a crescente utilização do Node é seu gerenciador de pacotes (NPM), que atualmente (2022), conta com mais de um milhão de pacotes *open source* registrados (npm: pacotes, 2022). O grande número de pacotes disponíveis através da linha de interface de comando do Node, concede um desenvolvimento mais rápido, todos os pacotes são instalados em uma pasta cujo o nome é node_modules, já a lista de dependência de um projeto pode ser conferida no arquivo package.json, que contém informações sobre a versão atual do pacote, regras de atualização entre outras informações relacionadas ao pacote, a instalação de um novo pacote é relativamente simples, o comando segue o formato: npm install <nomeDoPacote> –<argumentos>.

3.1.2 – Express

Framework de código aberto para Node.js que permite de maneira simples e robusta configurar uma aplicação web e sua API. Suas principais funções são, gerenciamento de rotas já com os verbos HTTP e expressões regulares para as rotas, middlewares construídos pelo próprio desenvolvedor para autenticação, autorização, tratamento

de exceções entre outras necessidades (express.js, 2022).

3.1.3 – Mongoose

Uma das principais características de bancos de dados orientados a documentos como o MongoDB, é que uma coleção de documentos (uma tabela em bancos relacionais), não possui um esquema fixo, agregando grande flexibilidade no armazenamento, apesar deste comportamento ser vantajoso em alguns casos, ao se construir uma aplicação com entidades bem definidas, é necessário algum tipo de modelo que valide os dados que estão entrando. Deste modo, o mongoose é uma biblioteca para mapeamento de um objeto com um documento, esse mapeamento é feito através de um esquema que utiliza o conceito par nome-valor, em que é definido o nome do dado e seu tipo, por exemplo: {nome: string, valor: number…}. Além desta funcionalidade essencial, esta biblioteca fornece diversas maneiras de se criar métodos customizados para uma maior abstração e padronização do código.

3.1.4 – TypeScript

A linguagem JS originalmente foi criada para rodar scripts simples dentro do browser, sendo fracamente tipada e não possuindo diversas funcionalidades de linguagens mais robustas, agora com o Node e a utilização da linguagem fora do browser, se fez necessário a adição de novas funcionalidades que deixassem a linguagem mais completa, desta forma, surgiu o TypeScript (TS). TS, é uma linguagem fortemente tipada que é transpilada para JS no momento de sua execução, esse *superset* do JS adiciona: tipagem para variáveis, objetos e funções, interfaces, classes abstratas, módulos, entre outras utilidades. Ao se construir aplicações mais complexas, e com múltiplas pessoas trabalhando no mesmo projeto, TS é mais adequado, uma vez que provém um código com maior nível de abstração que em conjunto com testes unitários, aumenta a qualidade do código, servindo quase como uma documentação.

3.2 - Docker

Plataforma open source que facilita a construção de aplicações distribuídas, sendo que essas aplicações dentro Docker recebem o nome de *container*, que seria a imagem de uma VM que contém todas as dependências necessárias. Esses *containers* rodam de forma isolada no kernel do sistema operacional, facilitando a criação e controle de dos mesmos, além disso, são extremamente leves, uma vez que contém apenas o essencial para executar a aplicação, e portáteis, não havendo necessidade de uma instalação manual em máquinas diferentes, Rad et al. (2017) .

3.2.1 – Kubernetes

Plataforma para orquestração de containers, foi construído para facilitar a construção, escalabilidade e automação de sistemas distribuídos orientados a containers, lidando com problemas como balanço de carga entre réplicas, restart automático de deployments em caso de falha, updates sem tempo de inatividade e entre outros, Karim, Lebre (2020).

Kubernetes é essencial quando se trata de microsserviços, já que a orquestração de vários serviços pode ficar complexa, desse modo, seus principais componentes são, *Kubernetes Cluster*, coleção de node e um master para manejar os nodes, *Node*, uma *virtual machine* (VM) que roda os containers, *Pod*, algo como um *container* em execução, sendo que um *pod* pode rodar múltiplos contêiner, *Deployment*, responsável por monitorar um certo número de pods (idênticas), averiguando se estão rodando corretamente e *Service*, que provém uma url para acessar containers que estão rodando.

3.3 – Mensageiros e Comunicação em Microsserviços

Ao se desenvolver uma aplicação com microsserviços, é necessário pensar em um método para que esses serviços se comuniquem, procurando sempre manter um fraco acoplamento para que mantenham suas premissas de serem resistentes a falhas, escaláveis e de fácil mantenabilidade, deste modo, a presença de um software mensageiro que auxilia na emissão e distribuição de eventos é essencial.

3.3.1 – Comunicação e Persistência de Dados

Existem diversas abordagens sobre como arquitetar a comunicação de microsserviços apesar de não existir o melhor método, existem duas principais. A primeira é sem a presença de um mensageiro, sendo então síncrona, em que os domínios (serviços) da aplicação se comunicam diretamente através de chamadas HTTP, esta abordagem contém quase todas as desvantagens de uma aplicação monolítica, como forte acoplação e não resistente a falhas, em conjunto com a complexidade de serviços distribuídos. A segunda é com a ajuda de uma fila de eventos em conjunto com algum modo de distribuir estes eventos, normalmente através de um mensageiro, sendo assíncrono; neste método, cada domínio da aplicação tem seu próprio banco de dados, contendo somente as tabelas necessária para seu próprio funcionamento, sendo então, totalmente independente de outros serviços (fracamente acoplado), a parte complexa deste método, é que em uma aplicação os serviços contem sempre algum tipo de interdependência, a solução para este problema, é a implementação de uma fila de eventos, sempre que um domínio realiza uma operação que adicione ou modifique seus dados, é emitido um evento para um canal do mensageiro, e todos os outros domínios cadastrados neste canal específico recebem o evento e com base em suas próprias regras de negócio utilizam o evento para modificar seu banco de dados, sendo possível criar serviços adicionais para geração de relatórios e outras necessidade. As principais vantagens dessa abordagem são, resistência a falhas, uma vez que os serviços são autossuficientes e escalabilidade por necessidade de cada domínio, as desvantagens são, alta complexidade de implementação, já que todos os envolvidos no projeto precisam conhecer estes conceitos, assim como um especialista para manter o projeto no

caminho correto e duplicação de dados já que diferentes serviços podem ter classes em comum para seu funcionamento.

3.3.2 – Soluções Mensageiros

Existem diversas soluções open source para softwares mensageiros, como o RabbitMQ, ApacheKafka entre outros, para a aplicação desenvolvida neste trabalho foi escolhido o NATS Streaming Server. Assim como outros mensageiros, NATS utiliza o paradigma publicar/inscrever, em que o responsável por publicar a mensagem especifica o canal e todos os clientes inscritos recebem a mensagem, sem a necessidade de o cliente estar disponível no momento da publicação (assíncrono), além disso, essas mensagens são armazenadas em uma fila e são enviadas apenas uma vez para cada cliente. NATS também conta características específicas como o desligamento automático de inscritos que não estão conectados e inscrições persistentes que garantem que clientes com tempo de inatividade recebam suas mensagens (T Sharvari, K Sowmya. 2019).

4 – Proposta e Desenvolvimento do Trabalho

4.1 - Proposta do Trabalho

A proposta do presente trabalho foi construir uma aplicação simples com duas arquiteturas distintas, SOA monolítico e microsserviços, o aplicativo permite a compra e venda de produtos alimentícios, assim como em propostas semelhantes, o cliente pode escolher o fornecedor e realizar a compra dos produtos desejados, ao passo que o fornecedor consegue cadastrar seus produtos e preços.

Desenvolveu-se um pequeno número de serviços, construídos em ambas as arquiteturas, sendo capaz de medir os pontos positivos e negativos de cada proposta, assim como as dificuldades de mudar da arquitetura SOA monolítica para a arquitetura de microsserviços.

4.2 – Desenvolvimento do Trabalho

A apresentação do desenvolvimento da aplicação foi dividida nos seguintes tópicos: modelagem das classes, diagrama de caso de uso, tecnologias utilizadas, infraestrutura, padronizações na codificação das classes, as telas e por fim uma comparação das especificidades de cada arquitetura utilizada.

4.3 – Diagrama de Classes

Ao se construir uma aplicação, é necessário definir os dados que serão armazenados, uma possível abordagem é utilizando conceitos de *Domain Drive Design* (DDD), esta abordagem divide a aplicação em domínios (classes), para cada classe, através da linguagem ubíqua, é definido seus atributos, métodos e área de atuação dentro da aplicação.

Para aplicação desenvolvida, foram identificados três domínios, o primeiro é o de *users*, responsável por autenticar e autorizar usuários, optou-se por utilizar o conceito de cargos para separar clientes de fornecedores, fazendo com que chamadas para recuperar dados de usuário fossem mais abstraídas, além de facilitar uma possível adição de mais cargos; o segundo é o de *products*, responsável por permitir que fornecedores cadastrem seus produtos; o terceiro é o *orders*, responsável pelas ordens criadas por clientes, associando sempre um cliente a um fornecedor e aos produtos que a serem comprados, na figura 1, é possível visualizar o diagrama de classes com mais detalhes sobre os atributos das classes e seus métodos.

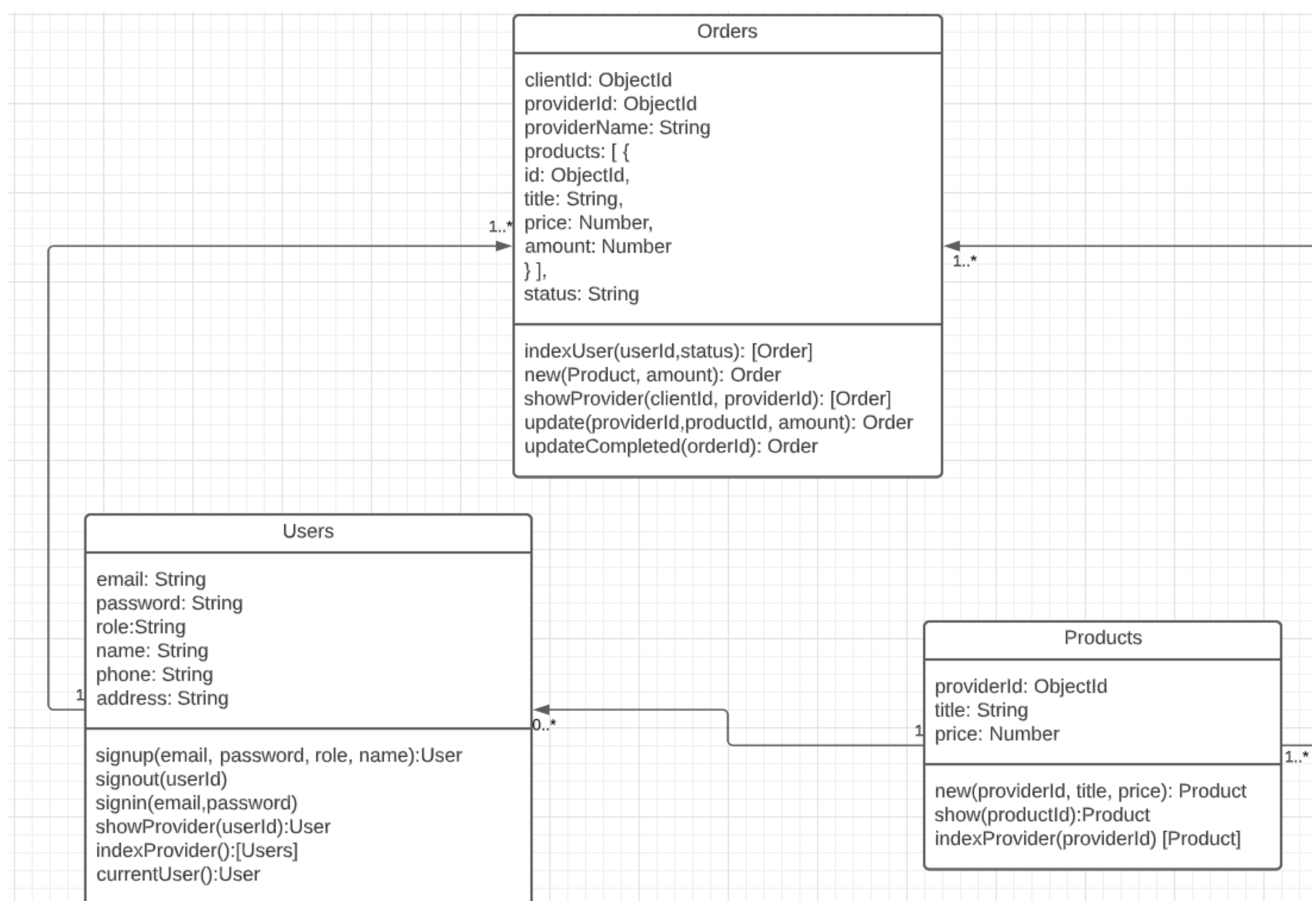

Figura 1: Diagrama de Classes da Aplicação

4.4 – Diagrama de Caso de Uso

Este diagrama é útil para se ter uma visão geral de como os usuários interagem com o sistema, assim como suas permissões dentro da aplicação. Para a presente aplicação, foram identificados três atores: cliente, fornecedor e usuário não registrado. Na figura 2 abaixo é possível visualizar quais funcionalidades cada autor tem acesso.

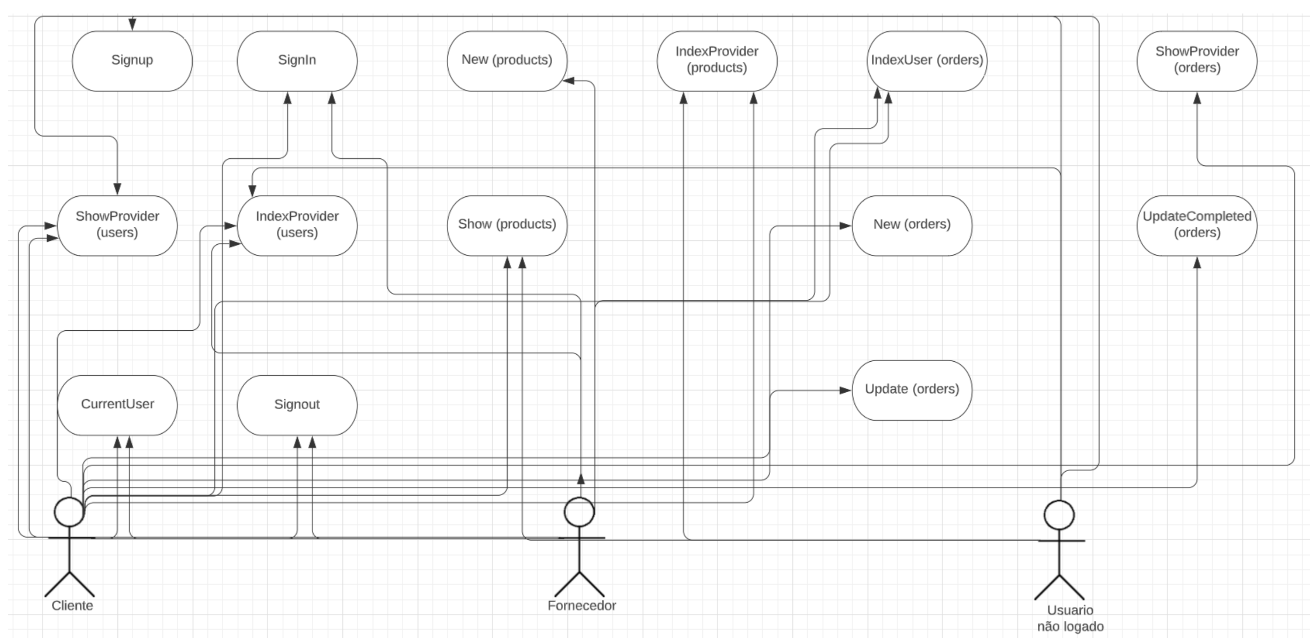

Figura 2: Diagrama de Caso de Uso

4.5 – Tecnologias Utilizadas

Para o back-end da aplicação, em ambas arquiteturas, a fim de construir os serviços e o servidor web, foi utilizado TS em conjunto com Node.js, MongoDB para armazenar os dados e exclusivamente para a arquitetura de microsserviços, foi utilizado o NATS streaming server como mensageiro, distribuindo eventos. Já o front-end, em ambas as arquiteturas foi utilizado React com JS, conseguindo aproveitar a mesma imagem Docker. Por fim, foi utilizado o Kubernetes (K8s) para orquestrar os containers, facilitando a automação do desenvolvimento da aplicação, com integração contínua e entregas contínuas (CI/CD), permitindo por exemplo que na aplicação desenvolvida com a arquitetura de microsserviços, apenas os domínios afetados fossem reiniciados para entregar novas funcionalidades, sem tempo de inatividade de outros serviços.

4.6 – Infraestrutura da Aplicação

Em ambos projetos, foi utilizado o GitHub[1] como repositório de código, quando se trata de arquitetura de microsserviços, uma possível abordagem é utilizar o conceito de multi-repo, em que se tem múltiplos repositórios, um para cada domínio ou serviço da aplicação, no entanto, devido a complexidade de se administrar este tipo de abordagem, em ambos os projetos, optou-se pelo conceito de mono-repo, em que se utiliza apenas um repositório para todos os serviços. Apesar de não ter sido implementado nesta aplicação específica, o GitHub oferece o GitHub Actions, uma plataforma que ajuda no CI/CD, permitindo por exemplo, que todo novo código enviado passe por testes automatizados no próprio GitHub, e somente se obtiver sucesso será integrado ao ambiente destinado.

O ambiente de desenvolvimento foi construído levando em conta a integração contínua com o K8s, foi utilizado a ferramenta *Skaffold* que facilita o CI/CD, sendo configurado através de um arquivo .yaml, onde é declarado as imagens que serão utilizadas e seus respectivos locais, o *Skaffold* detecta quando ocorre uma mudança no código e automaticamente atualiza o deployment do K8s, praticamente sem tempo de inatividade. Para a aplicação com arquitetura SOA, a configuração foi feita com um K8s rodando na própria máquina, ao passo que a de MS, foi configurada com uma máquina virtual do google cloud, uma vez que existem muitas VMs rodando simultaneamente, consumindo memória RAM em excesso, na figura 3 abaixo pode ser visualizada o arquivo de configuração de um deployment K8s.

Por último, para que os diferentes servidores dentro do cluster K8s comuniquem com o mundo exterior, é necessário criar um serviço para load balancer na configuração do deployment, este serviço provê uma url para acessar o contêiner enquanto está em execução. Em conjunto com este serviço foi utilizado o kubernetes/ingress-nginx, que serve como um load balancer e um proxy reverso, utilizando um arquivo de configuração .yaml, é possível especificar o caminho (path) de cada service do K8s, a figura 4 contém um exemplo de configuração do ingress.

---

[1] Link para o github com o código de cada projeto: https://github.com/apollxx; tcc_ms e tcc_soa.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: users-depl
spec:
  replicas: 1
  selector:
    matchLabels:
      app: users
  template:
    metadata:
      labels:
        app: users
    spec:
      containers:
        - name: users
          image: us.gcr.io/tcc-ms-353211/users
          env:
            - name: MONGO_URI
              value: mongodb://users-mongo-srv:27017/users
            - name: JWT_KEY
              valueFrom:
                secretKeyRef:
                  name: jwt-secret
                  key: JWT_KEY
```

Figura 3: Configuração de um deployment Kubernetes

```yaml
apiVersion: v1
kind: Service
metadata:
  name: users-srv
spec:
  selector:
    app: users
  ports:
    - name: users
      protocol: TCP
      port: 3000
      targetPort: 3000
```

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-srv
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/use-regex: "true"
spec:
  rules:
    - host: ms.dev
      http:
        paths:
          - path: /api/users/?(.*)
            pathType: Prefix
            backend:
              service:
                name: users-srv
                port:
                  number: 3000
```

Figura 4: Configuração de um Serviço para Load Balancer (esquerda) e Configuração do Ingress-NGINX

(direita).

4.7 – Estrutura do Projeto

De modo geral, optou-se por padronizar organização das pastas da seguinte maneira: pasta *Infra* onde estão as configurações do K8s, pasta Client onde está o front-end da aplicação, pastas de domínio, para SOA é somente o server enquanto para MS cada domínio tem sua própria pasta, pasta *Common*, exclusivo para MS, esta pasta é publicado no NPM para compartilhar definições comuns entre os diferentes domínios da aplicação e por fim, na raiz do projeto o arquivo de configuração do Skaffold.

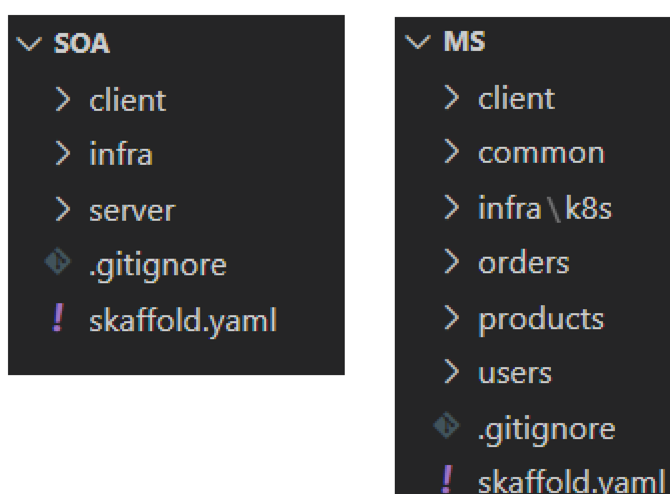

Figura 5: Organização das Pastas dos Projetos

4.8 – Estrutura dos Domínios

Na raiz de uma pasta de domínio (ou server), tem um package.json, onde estão todas as dependências do projeto e o Dockerfile que especifica como deve ser montada a imagem de um container. Dentro da pasta src, está todo o código específico de um domínio, sendo dividido da seguinte maneira: na raiz da pasta src, o arquivo app.ts contém toda a configuração do express, o arquivo index.ts é o responsável por iniciar o servidor e validar se todas as variáveis de ambiente estão devidamente configuradas, dentro da pasta models, estão todas as classes mongoose e TS que definem os tipos de dados do domínio, dentro da pasta routes, estão todas as funcionalidades (serviços), regras de negócio e rotas do servidor (caminho da api), dentro da pasta test, estão as configurações para os teste unitários (não implementado nas aplicações) e por fim,

exclusivamente para MS, a pasta events contém todos os *publishers* e *listeners* do domínio.

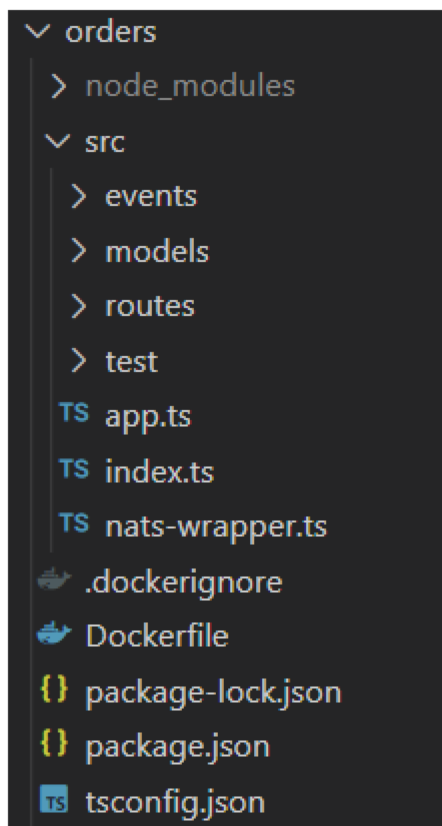

Figura 6: Organização de um Domínio

4.9 – Telas

A fim de prover uma interface para as funções construídas no Back-end da aplicação foram construídas um total de sete telas e um menu para a navegação dos usuários, sendo que algumas destas telas são exclusivas para um dos tipos de usuário, a seguir será detalhado o objetivo de cada tela e as funções utilizados.

4.9.1 – Barra de Navegação

Para que um usuário consiga acessar as principais telas da aplicação, um menu para navegação é essencial. Para esta aplicação, foi construído uma barra de navegação que de acordo com o tipo de usuário é modificada, para um usuário não logado, estão disponíveis as funções de se cadastrar e a de se logar; para um cliente logado, a possibilidade de ver suas ordens aberta, suas ordens completadas e de se deslogar;

por fim, para um fornecedor, é possível registrar um produto, visualizar as ordens completadas e se deslogar.

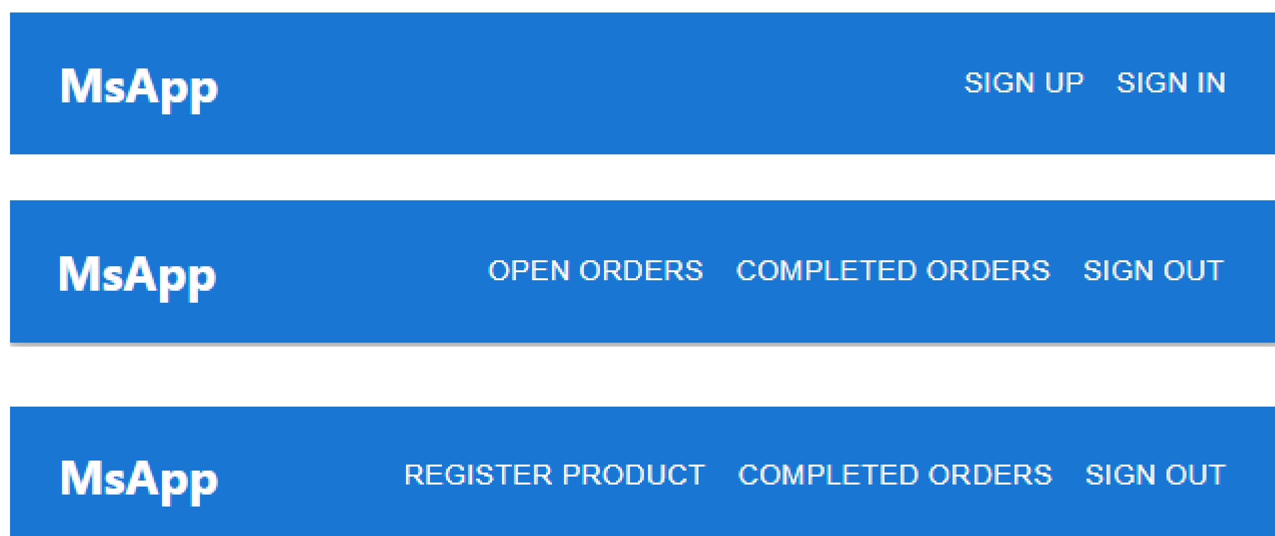

Figura 7: Barras de Navegação

4.9.2 – Tela de Cadastro

Esta tela permite que usuário se cadastre na aplicação, sendo necessário informar o nome, email, senha e escolher o tipo de usuário. Usuários do tipo cliente conseguem comprar produtos e abrir ordens, enquanto usuários do tipo fornecedor conseguem adicionar produtos à sua loja.

Sign Up

name

email

password

Role

Client

Provider

SIGN UP

Figura 8: Tela de Cadastro

4.9.3 – Tela de Autenticação

Esta tela permite que usuários cadastrados acessem suas contas, sendo necessário informar o email e a senha.

Figura 9: Tela de Autenticação

4.9.4 – Tela de Fornecedores Cadastrados

Esta tela é o index da aplicação, sempre que um usuário é autorizado ele é redirecionado para esta tela, onde estão listados todos os fornecedores cadastrados na aplicação. Todo tipo de usuário consegue visualizar esta tela e ao clicar em um fornecedor, o usuário é redirecionado a uma página onde pode visualizar os produtos do fornecedor.

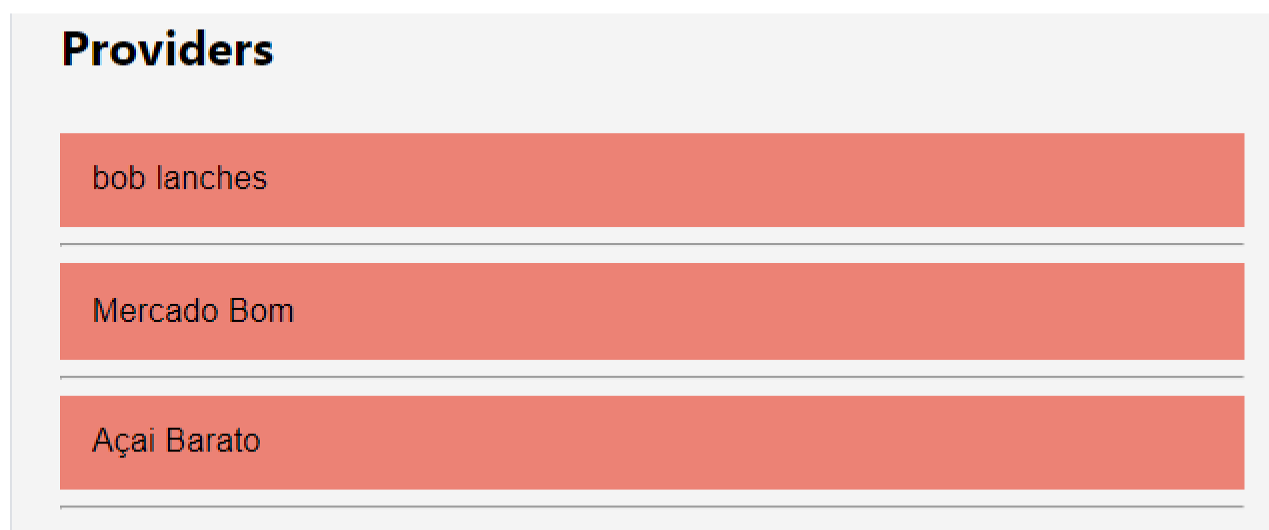

Figura 10: Tela de Fornecedores Cadastrados

4.9.5 – Tela de Produtos do Fornecedor

Um usuário consegue acessar essa tela através do index da aplicação, ao clicar em um fornecedor. Ao clicar em um dos itens, o usuário será redirecionado aos detalhes deste item, onde poderá associá-lo a uma ordem.

Products

Title	Price
Lanchao do Bob	15
Lanche Normal	10
Super Lanche	18

Figura 11: Tela de Produtos de um Fornecedor

4.9.6 – Tela de Detalhes do Produto

Esta tela pode ser acessada através da tela de produtos de um fornecedor, ao clicar em um dos produtos disponíveis, sendo exclusiva para usuários do tipo cliente. O objetivo desta tela é fornecer detalhes do produto para o cliente, permitindo que o mesmo escolha a quantidade que quer deste produto e o adicione a sua ordem.

Lanchao do Bob

Price: 15

Amount: 0 + -

ADD

Figura 12: Tela de Detalhes do Produto

4.9.7 – Tela de Novo Produto

Esta tela pode ser acessada através da barra de navegação, apenas por usuários do tipo fornecedor, provendo uma interface para que fornecedores consigam cadastrar seus produtos ao informar o nome e o preço.

Figura 13: Tela de Registro de Produto

4.9.8 – Tela de Ordens

Esta tela pode ser acessada através da barra de navegação, por usuários do tipo cliente e fornecedor, clientes podem consultar suas ordens abertas e finalizá-las ou consultar suas ordens finalizadas, fornecedores podem somente visualizar suas ordens finalizadas.

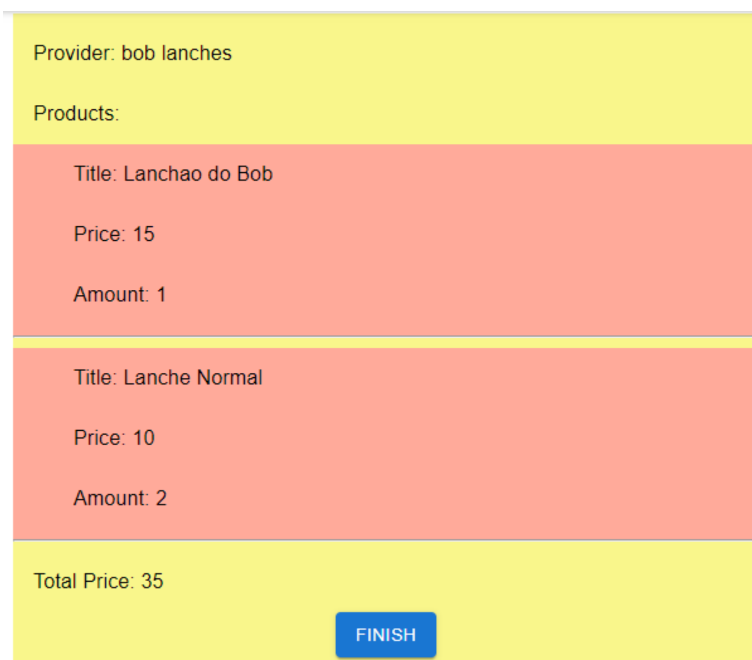

Figura 14: Tela de Ordens

4.10 – Comparação das Arquiteturas

Com a construção da aplicação com arquitetura SOA monolítica e a com arquitetura de microsserviços, foi possível observar as dificuldades da transição de uma arquitetura para a outra, assim sendo, foram analisados os seguintes aspectos: modelagem da aplicação e infraestrutura .

A modelagem da aplicação, seria a definição dos domínios, tipos de dados que serão armazenados e as funcionalidades esperadas. A definição dos domínios independe das tecnologias utilizadas, portanto, não há mudanças neste tópico. Na arquitetura de microsserviços, cada domínio tem seu próprio banco de dados, quando um domínio necessita de informações de outro, é necessário adaptar o banco de dados do domínio em específico para conter apenas os dados necessários para a informação, além disto, para implementar esta ideia, é necessário utilizar um mensageiro para distribuir os eventos, portanto há uma grande mudança neste tópico. Por fim, as funcionalidades da aplicação permanecem as mesmas uma vez que não há necessidade de modificar as funções existentes já que cada serviço contém os dados que necessita. Como para as aplicações construídas foi utilizado o K8s para orquestrar os múltiplos containers, as mudanças na infraestrutura foram pequenas, bastando apenas adicionar mais arquivos de configuração e o cluster do K8s gerência automaticamente.

5 – Conclusão

Com o aumento no número e diversidade de dispositivos eletrônicos com acesso a internet, fez com que os sistemas distribuídos ficassem mais complexos, surgindo então, novas arquiteturas que suprisse essas novas necessidades.

O presente trabalho teve como objetivo, comparar duas arquiteturas utilizadas em sistemas distribuídos, a SOA moníltica e a de microsserviços, essa comparação foi feita através da construção de uma mesma aplicação que utilizasse as diferentes arquiteturas, assim sendo capaz de entender as dificuldades e vantagens de se utilizar cada uma. As tecnologias utilizadas para a construção das aplicações foram: TypeScript com Node.js como linguagem para construir os servidores, MongoDB para persistência de dados, Docker e Kubernetes para a infraestrutura, facilitando a transição da arquitetura SOA para a de Microsserviços e exclusivamente para a arquitetura de microsserviços NATS streaming server para mensageria.

Desta forma, conclui-se que os objetivos especificados no presente trabalho foram completados, ambas as proposta provaram ser uma boa escolha dentro de suas próprias especificidades, a arquitetura SOA monolítica provê uma construção mais rápida, acelerando o desenvolvimento de projetos e tendo uma maior flexibilidade ao passo que a arquitetura de microsserviços se destaca em projetos mais estabelecidos fornecendo escalabilidade por domínio da aplicação, além de uma melhor definição dos domínios. Para propostas futuras, poderiam ser feitas comparações quantitativas entre as arquiteturas como custo, desempenho entre outras características, além disto, poderiam sem explorados mais a fundo processos de DevOps e testes automatizados.

Referências Bibliográfica

Breivold, Crnkovic, Larsson. A systematic review of software architecture evolution research. Disponível em: https://www.sciencedirect.com/science/article/abs/pii/S0950584911001376. Acesso em 7 nov. 2021.

Express.js. Disponível em: https://expressjs.com/. Acesso em 3 jul. 2022.

Fielding, Roy. Architectural Styles and the Design of Network-based Software Architectures. Disponível em: https://www.ics.uci.edu/~fielding/pubs/dissertation/abstract.htm. acesso em 9 mar 2022.

Fielding, Roy, Et al. Hypertext Transfer Protocolo - - HTTP/1.1 .Disponível em: https://www.hjp.at/doc/rfc/rfc2616.html. Acesso em 13 mar 2022.

Garlan. Software architecture: a travelogue. Disponível em: https://dl.acm.org/doi/abs/10.1145/2593882.2593886. Acesso em: 7 nov. 2021.

Lee, Siew. Chan, Lai. Lee, Eng. Web Services Implementation Methodology for SOA Application. Disponível em https://ieeexplore.ieee.org/document/4053410. Acesso em: 8 mar. 2022.

Liu, Yan et al. Reengineering Legacy Systems with RESTful Web Service. Disponível em: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.414.7936&rep=rep1&type=p df. Acesso em: 9 mar. 2022.

Manaouil, Karim; Lebre, Adrien. Kubernetes and the Edge. Disponível em: https://hal.inria.fr/hal-02972686v2/document. Acesso em 13 jul. 2022.

npm: pacotes. Disponível em https://blog.rocketseat.com.br/npm-pacotes-pacotes-e-pacotes/. Acesso em: 24 jun. 2022.

Sobre Node.js. Disponível em: https://nodejs.org/pt-br/about/. Acesso em: 24 jun. 2022.

T, Sharvari. K Sowmya. A study on Modern Messagin System - Kafka RabbitMQ and

NATS Streaming. Disponível em: https://arxiv.org/ftp/arxiv/papers/1912/1912.03715.pdf. Acesso em 17jul. 2022.

Tihomirovs, Juris. Grabis Jānis. Comparison of SOAP and REST Based Web Services Using Software Evaluation Metrics. disponível em: https://itms-journals.rtu.lv/article/view/itms-2016-0017. Acesso em 8 mar. 2022.

RAD, Babak Bashari; BHATTI, Harrison John; AHMADI, Mohammad: An Introduction to Docker and Analysis of its Performance. Dispnível em: https://www.researchgate.net/profile/Harrison-Bhatti/publication/318816158_An_Introduction_to_Docker_and_Analysis_of_its_Performance/links/61facc0c007fb504472fd6c7/An-Introduction-to-Docker-and-Analysis-of-its-Performance.pdf. Acesso em 11 jul 2022.

Richardson, Chris. Microsservices Pattern: with examples in javas. 1. ed.

Victorino, Marcio; Bräscher, Marisa. Organização da Informação e do Conhecimento, Engenharia de Software e Arquitetura Orientada a Serviços: uma Abordagem Holística para o Desenvolvimento de Sistemas de Informação Computadorizados. Disponível em: https://www.brapci.inf.br/_repositorio/2010/08/pdf_739cdbe8fd_0011617.pdf.

Acesso em:

6 mar. 2022.