UNIVERSIDADE FEDERAL DE OURO PRETO

INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS

DEPARTAMENTO DE COMPUTAÇÃO

VINÍCIUS SOUZA ALMEIDA

Orientador: Prof. Dr. Carlos Frederico M. C. Cavalcanti

PLATAFORMA DE E-COMMERCE INTEGRADA A MICRO SERVIÇOS

Ouro Preto, MG

2022

UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

VINÍCIUS SOUZA ALMEIDA

PLATAFORMA DE E-COMMERCE INTEGRADA A MICRO SERVIÇOS

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Carlos Frederico M. C. Cavalcanti

Ouro Preto, MG
2022

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

Vinícius Souza Almeida

PLATAFORMA DE E-COMMERCE INTEGRADA A MICROSERVIÇOS

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal
de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 12 de Janeiro de 2022.

Membros da banca

Carlos Frederico M. da Cunha Cavalcanti (Orientador) - Doutor - Universidade Federal de Ouro Preto
Fernando Cortez Sica (Examinador) - Doutor - Universidade Federal de Ouro Preto
Ricardo Augusto Rabelo Oliveira (Examinador) - Doutor - Universidade Federal de Ouro Preto

Carlos Frederico M. da Cunha Cavalcanti, Orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 12/01/2022.

Documento assinado eletronicamente por **Carlos Frederico Marcelo da Cunha Cavalcanti**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 13/01/2022, às 12:41, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0267929** e o código CRC **5F00DCEE**.

Referência: Caso responda este documento, indicar expressamente o Processo nº 23109.013468/2021-94 SEI nº 0267929

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35400-000
Telefone: 3135591692 - www.ufop.br

Agradeço aos meus pais e meu irmão por toda paciência e ajuda até aqui.

Agradecimentos

Primeiramente gostaria de agradecer a Universidade Federal de Ouro Preto, por todo suporte e ensino de qualidade e ao meu orientador Carlos Frederico, por sempre acreditar em mim.

"You are dividing the services such that you are developing them independently, deploying them independently, and maintaining them independently. The very basic idea of Divide and Conquer."(SHAIKH, 2021)

Resumo

Neste trabalho a concepção de uma plataforma de um e-commerce foi efetuada por completa, usando a arquitetura de microserviços, contendo três diferentes *API's REST*, uma para cliente, uma para vendedor e uma para efetuar a transação entre os dois, na confecção do *frontend* foi usado TypeScript, ReactJS e NextJS para a escrita do código. Já os três *backends* para os diferentes microserviços citados anteriormente o NodeJS foi escolhido, pois o mesmo possui fácil integração, velocidade e bom acoplamento. O *deploy* foi feito em um servidor próprio onde foram abertas as portas e toda a configuração de rede e DNS e propagação feita manualmente.

Palavras-chave: Microserviços. Backend. FrontEnd. Deploy.

Abstract

In this work, the conception of an e-commerce platform was carried out completely, using the microservices architecture, containing three different REST API's, one for the client, one for the seller and one to carry out the transaction between the two, in the creation of the frontend TypeScript, ReactJS and NextJS were used to write the code. For the three backends for the different microservices mentioned above, NodeJS was chosen, because it has easy integration, speed and good coupling. The deployment was done on its own server where ports were opened and all the network and DNS configuration and propagation were done manually.

Keywords: Microservices, Backend. FrontEnd. Deploy.

Lista de Ilustrações

Figura 3.1 – Modelo relacional banco de dados de usuário 8
Figura 3.2 – Modelo relacional banco de dados de vendedor 9
Figura 3.3 – Modelo relacional banco de dados de compras 9
Figura 3.4 – MVC, model view controller, por (GACKENHEIMER, 2015) 13
Figura 3.5 – Hierarquia de componentes, por (FEDOSEJEV, 2015) 14
Figura 3.6 – Configurações do servidor 17
Figura 3.7 – Tabela do Heroku 18
Figura 3.8 – Topologia 19
Figura 4.1 – Tela inicial com vendedor logado 22
Figura 4.2 – Usuário comum sendo logado independente de outras API's na API de usuário 23
Figura 4.3 – Vendedor comum sendo logado independente de outras API's na API de Vendedor 23
Figura 4.4 – Retorno de uma imagem de um determinado produto de uma venda com o id 19 24
Figura 4.5 – Configuração do banco de dados PostgreSQL no heroku 25
Figura 4.6 – Detalhes do banco de dados PostgreSQL Hobby dev no heroku 25
Figura 4.7 – Detalhes do banco de dados PostgreSQL Private 2 no heroku 26
Figura 4.8 – Testes de inserção de usuário em milissegundos 26
Figura 4.9 – Média de inserção de usuário em milissegundos 27
Figura 4.10–Testes de login de usuário em milissegundos 27
Figura 4.11–Média de login de usuário em milissegundos 27
Figura A.1 – Inserção 1 no servidor Heroku 32
Figura A.2 – Inserção 2 no servidor Heroku 32
Figura A.3 – Inserção 3 no servidor Heroku 32
Figura A.4 – Inserção 4 no servidor Heroku 33
Figura A.5 – Inserção 5 no servidor Heroku 33
Figura A.6 – Inserção 1 no servidor local 33
Figura A.7 – Inserção 2 no servidor local 33
Figura A.8 – Inserção 3 no servidor local 34
Figura A.9 – Inserção 4 no servidor local 34
Figura A.10–Inserção 5 no servidor local 34

Sumário

1 Introdução **1**

1.1 Justificativa 1

1.2 Objetivos 1

1.3 Fundamentação Teórica 2

1.4 Organização do Trabalho 4

1.4.1 Estrutura da Monografia 4

2 Revisão Bibliográfica **5**

2.1 Trabalhos Relacionados 6

3 Desenvolvimento **8**

3.1 Banco de dados 8

3.2 *back-ends* 9

3.2.1 KnexJS 9

3.2.2 *Endpoints* 10

3.2.3 *Middlewares* 10

3.2.4 *Express* 11

3.2.5 CORS 11

3.2.6 Node.js 12

3.3 *front-end* 12

3.3.1 TypeScript 12

3.3.2 React 13

3.3.2.1 Componentes 14

3.3.2.2 Contextos 14

3.3.2.2.1 AuthContext 15

3.3.2.2.2 AppContext 15

3.3.3 NextJS 15

3.3.4 Serviços 15

3.3.4.1 Axios 15

3.4 *Deploy* 16

3.4.0.1 Panorama na área de TI 16

3.4.0.2 Virtualização 16

3.4.0.3 *Datacenters* e Nuvem 17

3.4.0.4 O aspecto *PERFORMANCE* 17

3.4.0.5 Aspectos indiretos consideráveis 18

3.4.0.6 Topologia e *Deploy* Local 19

4 Resultados **22**

4.1 Estudos em bancos de dados com servidor local e online 24

4.1.1 Testes de performance 26
5 Considerações Finais 28
5.1 Conclusão 28

Referências 29

Apêndices 31
APÊNDICE A Chamadas aos servidores 32
A.1 Heroku - Inserção 32
A.2 Local - Inserção 33

1 Introdução

Microsserviços são um método arquitetônico para a criação de aplicativos. A diferença entre a arquitetura de microsserviço e a abordagem monolítica tradicional é como ela decompõe o aplicativo em funções básicas que se interagem entre si. Cada função é chamada de serviço e pode ser criada e implantada de forma independente (HENRIQUE, 2021). Isso significa que cada serviço individual pode funcionar ou falhar sem afetar outros serviços (REDHAT, 2021) e a aplicação como um todo.

Neste trabalho apresentaremos a concepção completa de sistema e-commerce desenvolvido usando microsserviços e também as principais tecnologias *front-end* que viabilizaram o seu desenvolvimento, principalmente React, NextJS e TypeScript.

Usamos três microserviços diferentes, um para vendedores do nosso e-commerce, outro para usuários finais e, por fim, um para as transações entre os dois.

Também fizemos um estudo sobre virtualização das máquinas e o comparativo entre o *deployment*[1] em máquinas virtuais e *deployment* em servidor local.

1.1 Justificativa

Softwares monolíticos dominaram as duas últimas décadas. Com a popularidade de empresas de IaaS (*Infrastructure as a Service*, também conhecido como "Infraestrutura na Nuvem") onde a AWS (Amazon Web Service), Google Cloud e Microsoft Azure são os provedores de infraestrutura mais populares na atualidade.

Este trabalho perpassa todas as partes da concepção de um software e, em nosso caso específico, concentrado na arquitetura em microserviços que atualmente é a arquitetura mais indicada para softwares escaláveis[2] e que sejam executados isto é, que "rodem"na "nuvem". Também foi utilizado um *deployment* diferente do usual, em um servidor próprio e não em serviços de um provedor de IaaS.

1.2 Objetivos

O objetivo geral deste trabalho é apresentar o desenvolvimento de um software baseado em microsserviços. Objetiva-se também mostrar como é a interação entre os serviços através das

[1] *deployment* é o termo usado quando se faz o *deploy*, isto é "colocar no ar", "em funcionamento", o sistema ou módulo desenvolvido.

[2] A escalabilidade do software é uma medida de quão fácil é aumentar ou diminuir um pedaço de software. Em muitos casos, refere-se à capacidade do software de lidar com cargas de trabalho aumentadas enquanto usuários são adicionados e quando são removidos com impacto de custo mínimo.

tecnologias elencadas de tal forma que o sistema funcione perfeitamente tanto em um ambiente profissional (também chamado "de produção") ou acadêmico. Iremos mostrar o desempenho das tecnologias ReactJS e NextJS para o *front-end* e NodeJS para os *back-ends*. O segundo objetivo é verificar o desempenho de um servidor local a um servidor alugado Amazon, isto é um servidor no modelo IaaS, onde o mesmo código será testado em ambos ambientes.

1.3 Fundamentação Teórica

Iremos, nesta seção, conceituar alguns termos usados em engenharia de software que são importantes no contexto deste documento

- *Framework*: *Framework* é um termo em inglês que, em sua tradução direta, significa estrutura. De modo geral, essa estrutura é projetada para resolver um problema específico. Na programação, uma estrutura é um conjunto de códigos comuns que podem combinar várias partes de um projeto de desenvolvimento. É como um quebra-cabeça que pode ser colocado em muitos lugares diferentes e conectar todas as linhas de código de uma maneira quase perfeita. Assim como suas funções e aplicativos parecem tão simples, os desenvolvedores precisam ter um bom entendimento do tipo de *framework* que está usando ou será usado em seu projeto. Um exemplo de *Framework* é o NextJS.

- *back-end*: É a parte do software responsável por trabalhar atrás dos panos, do lado do servidor. Quem processa os dados enviados pelo usuário, responde as requisições do front-end, lida com o banco de dados e quem se preocupa com a segurança da aplicação, o servidor entende, processa e, então, envia para o front-end a resposta. No nosso caso possuímos três diferentes e eles usam a linguagem JavaScript pelo NodeJs

- *front-end*: É o responsável por trabalhar o que o usuário enxerga, onde cada elemento se encaixa e é apresentado, o lado do cliente. Quem colhe os dados enviados pelo usuário e responde por sua experiência.

- *Deploy*: O verbo *to deploy*, em inglês, significa implantar. Em programação, seu significado está intimamente relacionado à sua tradução literal: implantação, na verdade, significa colocar alguns aplicativos que foram desenvolvidos online.

- DOM: É acrônimo para Document Object Model, DOM é a interface entre a linguagem Javascript e os objetos HTML. O DOM foi criado com o objetivo de desenvolver um padrão de linguagem de script para navegadores, pois no passado cada navegador tinha sua própria maneira de manipular objetos, o que causava muitas incompatibilidades e obrigava os desenvolvedores a escrever versões de script para todos os navegadores. Quando a página da web é carregada, o navegador cria o DOM, a árvore de elementos HTML.

- Arquitetura de software: Segundo (BALDISSERA, 2021), "Um padrão arquitetural é uma solução já estudada, testada e documentada de um problema recorrente. O modelo ajuda na tomada de decisões do projeto de software, como qual será sua utilidade e as funções e relacionamento de cada subsistema. É ele que define a estrutura fundamental do programa. Os modelos arquiteturais foram descritos pela primeira vez por Christopher Alexander, no final da década de 1970. Em dois livros, o autor descreve um método de documentação de padrões, que, apesar de ter sido pensado para a arquitetura, foi adaptado para a área de software e se popularizou na década de 1990. Desde então, os padrões se tornaram indispensáveis no trabalho de arquitetos de software."

- Banco de dados relacional: Segundo (ORACLE, 2021),"Um banco de dados relacional é um banco de dados usado para armazenar e fornecer acesso a pontos de dados relacionados entre si. Os bancos de dados relacionais são baseados no modelo relacional, que é uma forma intuitiva e direta de representar dados em tabelas."

- UI: é tudo aquilo que é percebido visualmente em alguma plataforma e leva o usuário a uma interação. Pode ser um botão, um menu diferente ou até mesmo som.

- *Benchmarking*: que traduzido para o português é "avaliação comparativa", portanto consiste no processo de busca das melhores práticas para melhorar o desempenho.

- Token de acesso: Segundo (FACEBOOK,), "um token de acesso é uma cadeia de caracteres opaca que identifica um usuário, aplicativo ou Página. Ele pode ser usado pelo aplicativo para fazer chamadas", no nosso caso as chamadas são nos nossos *backend*.

- *Query builder*: Segundo (DBFORGESTUDIO, 2021), é um projeto feito para aumentar a produtividade e simplificar as tarefas de construção de consultas SQL.

- *API RESTful*: Por *doglio2015pro*, simplificando, REST (abreviação de Representational State Tranfer) é um estilo arquitetônico definido para ajudar a criar e organizar sistemas distribuídos. A palavra-chave dessa definição deve ser estilo, porque um aspecto importante do REST é que é um estilo arquitetônico, não uma diretriz, um padrão ou algo que seria implica que há um conjunto de regras rígidas a serem seguidas, para acabar com uma arquitetura RESTful

- *Thread*: Um thread é um contexto de execução, que consiste em todas as informações de que uma CPU precisa para executar um fluxo de instruções.

- *Dead-lock*: É uma situação em que um grupo de processos é bloqueado porque outro processo que contém o recurso necessário para o mesmo ser executado está esperando que outros processos recebam os recursos de outros processos.

1.4 Organização do Trabalho

Este trabalho foi dividido em cinco diferentes capítulos, o primeiro é uma introdução, no segundo, temos uma revisão bibliográfica e o embasamento teórico de alguns termos necessários para o entendimento do trabalho, no capítulo três o desenvolvimento do trabalho, dividido em 4 etapas, banco de dados, *back-ends*, *front-end* e *deployment*. Já no capítulo quatro temos os resultados e discussões e no capítulo cinco a conclusão.

1.4.1 Estrutura da Monografia

Capítulo 1: Introdução, onde iremos apresentar o contexto de nosso trabalho, seus objetivos e como será feito o desenvolvimento;

Capítulo 2: Revisão bibliográfica, nesta seção explanarmos os principais conceitos e elementos usados nesta monografia;

Capítulo 3: Desenvolvimento, nesta seção iremos mostrar o projeto da aplicação e as decisões tomadas;

Capítulo 4: Resultados, nesta seção iremos demostrar o sistema pronto e as avaliações de performance feitas;

Capítulo 5.1: Conclusão, nesta seção iremos explorar sinteticamente as conclusões tiradas.

2 Revisão Bibliográfica

No início do desenvolvimento de softwares para web, a arquitetura comumente usada era a monolítica, que basicamente é uma aplicação de software em os diferentes componentes estão ligados a um único programa, e um dos grandes problemas dessa arquitetura, segundo (GOUIGOUX; TAMZALIT, 2017), era o acoplamento massivo entre os seus componentes. Isto resultava em um alto custo tanto computacional, quanto financeiro, porém principalmente temporal, para garantir a qualidade pois uma pequena modificação no código poderia ter impacto em qualquer recurso do aplicativo. Isso potencialmente atrapalha o surgimento de novas *features* (funcionalidades) em softwares que já estão prontos e também a manutenção do mesmo.

Neste cenário, a demanda de uma arquitetura de software com menor acoplamento entre os componentes e maior facilidade de manutenção foi criada. Com isso surgiu a arquitetura por microsserviços. Segundo (DMITRY; MANFRED, 2014), "a arquitetura em microsserviços é uma abordagem para desenvolver uma aplicação como um conjunto de pequenos serviços independentes. Cada um dos serviços é executado em seu próprio processo independente"

Portanto, dessa forma, o software que foi desenvolvido e mostrado desse documento foi dividido em módulos funcionais usando a arquitetura de microsserviços. Dessa forma, o *front-end* (parte visual e interação com o usuário) está desacoplado ao *back-end* (servidor), que foi separado em três *back-ends* diferentes que serão listados no corpo deste documento.

Para o *back-end* desse trabalho foi usado o Node.js. Segundo (GONZALEZ, 2016), "O Node.js é o candidato perfeito para arquiteturas microsserviços por uma série de razões, conforme indicado na lista a seguir:

- Fácil de aprender (embora possa ser difícil de dominar)
- Fácil de escalar
- Altamente testável
- Fácil de implantar
- Gerenciamento de dependências por meio de npm [1]
- Existem centenas de bibliotecas para integrar com a maioria dos padrões protocolos"

Portanto Node.js se mostra uma ótima escolha para a concepção do software de e-commerce.

[1] npm: Gerenciador de Pacotes do nodeJS (Node Package Manager) que vem junto com o nodeJS e que é muito útil ao desenvolvimento

Já na parte do *front-end* da aplicação foi usado o React junto com o Next.js usando o typescript para que o código esteja devidamente tipado e mais facilmente entendível.

O React, criado pela equipe do Facebook em 2011, tem como objetivo otimizar a atualização e sincronização das atividades em *feeds* (alimentadores) de notícias de redes sociais, incluindo chat, status, etc. Utiliza a linguagem de programação Javascript para tornar mais fácil e simplificada a ligação entre HTML, CSS e todos os componentes da página.

Javascript é uma das linguagens de programação mais comuns e populares do mundo, com um grande número de bibliotecas e outros usuários que a utilizam. Entre eles, Node.js, Angular, VueJS, jQuery, Ember.js.

Os chamados "componentes"facilitam o funcionamento da página, como a atualização das variáveis de uma determinada página. Existem atributos ou props no componente. Esses props são objetos com várias informações, que podem ser de diferentes tipos, como funções, números ou strings. Assim como atributos, states ou estados, eles também são objetos ou informações, mas em vez de serem passados para o componente, eles são criados dentro do componente. Ao contrário dos props, o estado é mutável, como variáveis declaradas dentro de uma função.

O DOM ou VDOM virtual é a representação da memória do DOM (Document Object Model) real da interface. Na prática, o DOM real representa a estrutura da camada visual da página. Para tornar isso mais claro, imagine o seguinte processo: Para atualizar a página, o React primeiro salva suas alterações na memória, em um ambiente chamado DOM virtual. Isso ocorre porque manipular o DOM virtual é muito mais rápido do que manipular o DOM real - ou seja, a página é atualizada. No *front-end*, a atualização do DOM é muito comum, pois é por meio deles que efetivamente atualizam as páginas de um site ou aplicativo. Após criar esta estrutura virtual, o React a converte para a tela real com o mínimo de processos possível, tornando as atualizações mais ágeis. Este processo é chamado de reconciliação.

Além do React, também usamos o NextJS, que é um framework do React. O que isto significa? Que NextJS adiciona alguns recursos no topo do React, como renderização estática, e do lado do servidor. Ele tem suporte para Typescript e um serviço de processamento de roteamento muito interessante, pois é simples de implementar e de fácil entendimento. Muitos recursos são mais perceptíveis em nossos aplicativos de produção porque o NextJS tem dois pontos principais em seu objetivo: fazer com que nossos aplicativos React tenham um desempenho melhor e o problema dos mecanismos de pesquisa indexarem o conteúdo da página.

2.1 Trabalhos Relacionados

De acordo com (GONZALEZ, 2016) "as arquiteturas orientadas a microsserviços têm algumas particularidades que os tornam desejável para qualquer empresa de médio / grande porte que deseja manter seus sistemas de TI resilientes e em status pronto para aumentar ou diminuir,

porém, eles não são o "Santo Graal"da engenharia de software mas, quando manuseados com cuidado, eles se tornam a abordagem perfeita para resolver a maioria dos grandes problemas enfrentados por empresas dependentes de tecnologia.

É importante manter os princípios-chaves da arquitetura orientada a microsserviços em mente, como resiliência, capacidade de composição, elasticidade e assim por diante; de outra forma, você pode acabar com um aplicativo monolítico dividido em diferentes máquinas que produz problemas em vez de uma solução elegante.

3 Desenvolvimento

3.1 Banco de dados

Para o desenvolvimento, foram escolhidos bancos de dados relacionais, neste caso o PostgreSQL que segundo (DRAKE; WORSLEY, 2002), é um projeto *open source*. *Open source*, por definição, significa que você pode obter o código-fonte, usar o programa e modificá-lo livremente, sem os limites do proprietário do software. No mundo do banco de dados, *Open source* significa que você tem acesso aprimorado a números de *benchmarking* e estatísticas de desempenho, que empresas como a Oracle proíbem". Dessa forma o primeiro passo foi a criação do modelo relacional nas figuras 3.1, 3.2 e 3.3 dos três diferentes bancos, pois cada *back-end* precisa possuir um banco próprio e sem conexões com os outros, para assim garantirmos o conceito de microserviços, pois caso algum dos *back-end*s caia, outra parte do sistema deve conseguir funcionar perfeitamente.

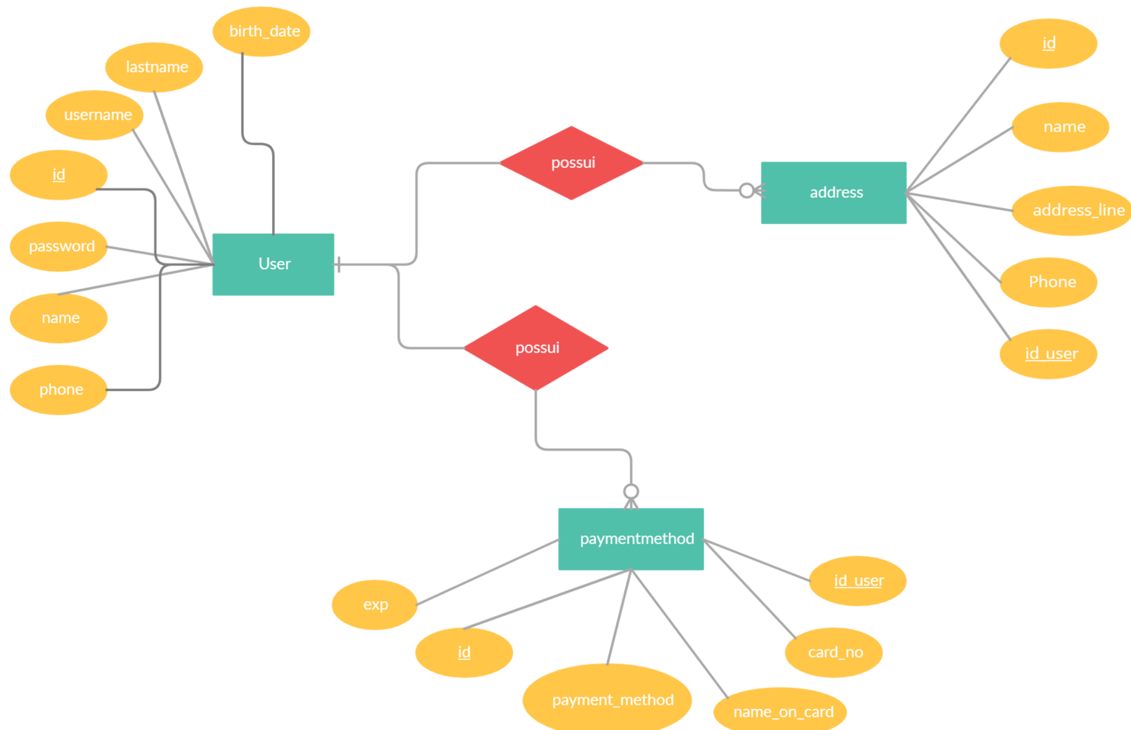

Figura 3.1 – Modelo relacional banco de dados de usuário

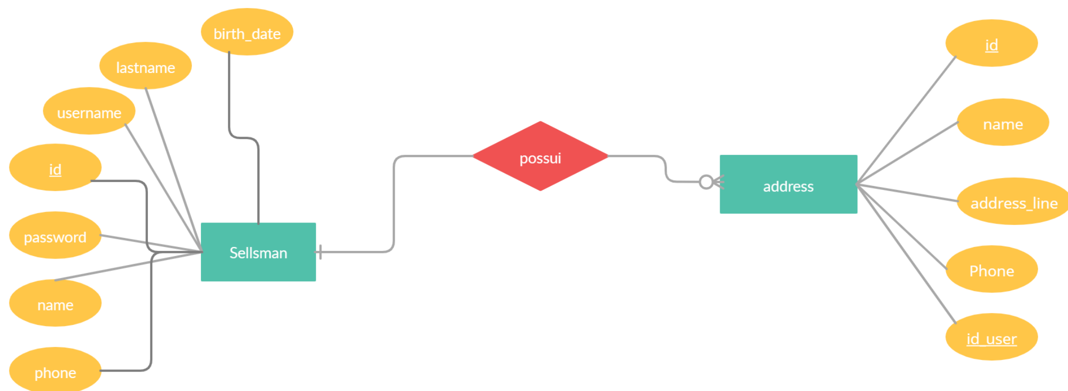

Figura 3.2 – Modelo relacional banco de dados de vendedor

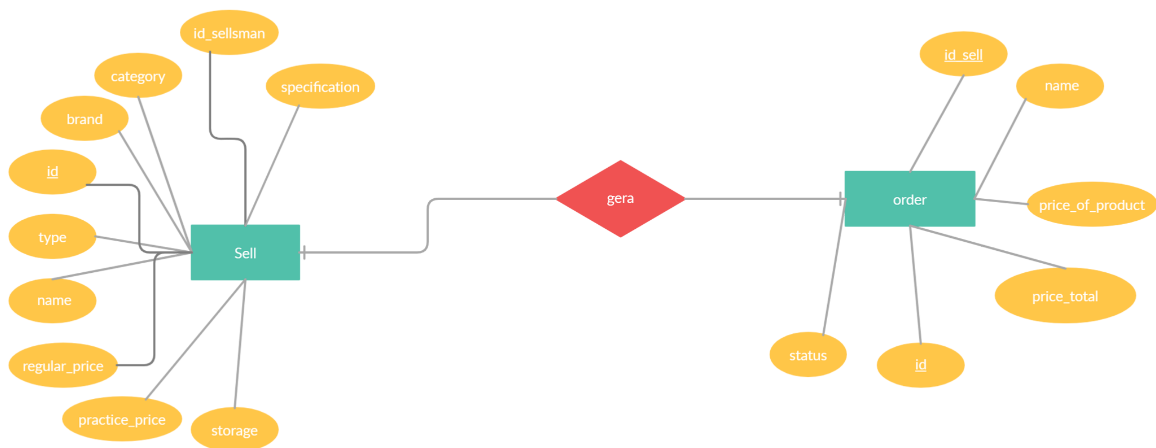

Figura 3.3 – Modelo relacional banco de dados de compras

3.2 *back-ends*

3.2.1 KnexJS

Para a modelagem de dados, seleções, modificações, inserções e exclusões do banco, integramos ao NodeJS um *query builder* chamado KnexJS que por meio do qual um código javascript unifica a forma de fazer consultas *(queries)* em bancos que usam a linguagem SQL, implementando uma camada de tal forma que as consultas sejam independentes de um banco específico, trazendo a facilidade da fácil troca. Isto implicará que a aplicação será funcionará independente do banco usando sem alterar o código.

Pelo KnexJS, foram criados *migrations* que, de acordo com o (KNEXJS, 2021), permitem que seja definido conjuntos de alterações de esquema do banco que é facilmente refletido no banco de dados, com esta ferramenta.

Dentro da pasta do *back-end* em NodeJS, um arquivo knexfile.js é criado para configurar

qual banco ele irá se conectar quando for chamado.

3.2.2 *Endpoints*

Endpoints é um termo em inglês que pode ser traduzido literalmente, como “pontos de extremidade”, desta forma o termo pode ser aplicado a diferentes contextos. Em protocolos de comunicação, por exemplo, os *endpoints* fazem referência aos terminais de conexão entre uma API/*back-end* e o *client side* que no caso seria o *front-end*.

Cada um dos três *back-ends* possuem *endpoints* de tal forma que o *front-end* faz a requisição e o *back-end* responde com informações. Por exemplo, os produtos que deverão aparecer na página inicial são informações que o *front-end* recebe do *back-end*. Outro exemplo são os *endpoints* das rotas de *login* para que os usuários ou o vendedor efetuarem o mesmo e receberem de volta o *token* de acesso as informações de quem está logado no sistema.

3.2.3 *Middlewares*

Como um e-commerce possui áreas onde o usuário ou vendedor devem estar logados ao sistema, ou seja, alguns *endpoints* só podem ser chamados caso o mesmo esteja logado e com um *token* de acesso válido para receber a resposta do *back-end*. O *middleware* nada mais é que um código de verificação que toda requisição precisa passar antes de entrar no código do *endpoint* específico, sabendo disso, criamos um *middleware* chamado *auth*, que verifica se um usuário está logado ou não pelo seu token de acesso enviado pelo *header* de cada requisição, portanto dessa forma sabemos se o usuário esta autorizado ou não a realizar uma determinada chamada ao *back-end* ou não.

Segundo (LEARNING, 2021), uma requisição de API REST permite que dados sejam recuperados ou enviados através de um serviço que está sendo executado em algum *host* na nuvem. Servidores da Web executam serviços que são acessados através de uma interface RESTfull que dão suporte às operações necessárias para a correta execução dos aplicativos clientes. Cada solicitação de API REST usa um método HTTP. Os métodos mais comuns são GET, POST, PATCH, PUT e DELETE, e ao enviar tal requisição a uma API privada (apenas para usuários logados) deverá ser enviado com um *token* de acesso. Em nosso caso, os *tokens*s são gerados pela biblioteca *jsonwebtoken*, que usa uma palavra secreta para gerar o *token* aleatoriamente e o mesmo dura 24h após a autenticação.

O *middleware* de verificação confere se o *token* é valido ou não e, a partir disso, perfaz a operação solicitada. Caso contrário, ele retorna que o usuário não está autorizado a perfazer a operação solicitda.

3.2.4 *Express*

Express é uma *framework* de aplicativos Web Node.js mínima e flexível, que fornece um conjunto poderoso de recursos para o desenvolvimento de aplicativos da Web e móveis. Promove o rápido desenvolvimento de aplicações web baseadas em Node.js, pois permite configurar *middlewares* para responder a solicitações HTTP, permite a criação de tabelas de roteamento para executar diferentes operações baseadas em métodos HTTP / HTTPS e URLs, e até mesmo permitir a renderização de páginas HTML dinamicamente com base na passagem de parâmetros para o modelo.

No nosso caso, o mais importante são o uso de rotas HTTP, criadas para serem respondidas por entradas específicas e diferentes parâmetros provenientes do *front-end*.

3.2.5 CORS

CORS é a abreviatura de *Cross-Origin Resource Sharing* que é um mecanismo que permite ou restringe os recursos solicitados no servidor Web, dependendo de onde é feita a solicitação HTTP.

Esta política é usada para proteger um servidor Web específico de ser acessado por outros sites ou domínios. Por exemplo, apenas domínios permitidos podem acessar arquivos hospedados no servidor, como folhas de estilo, imagens ou *scripts*.

Cada solicitação HTTP possui um cabeçalho denominado *origin*. Ele define a origem da solicitação de domínio. Podemos usar as informações do cabeçalho para restringir ou permitir que as funções do nosso servidor web os protejam.

Por exemplo, quando você ainda está desenvolvendo, se usar para o *front-end* como React, seu aplicativo de *front-end* será servido em http://localhost:3000. Ao mesmo tempo, seu servidor *Express* pode estar sendo executado em uma porta diferente, como http://localhost:3030.

Usamos três diferentes portas para os diferentes *back-end*s e as chamadas se diferem pela porta e o endereço de cada rota. Por exemplo, a rota de *login* de usuário é http://localhost:3030/login, enviando no corpo da requisição o usuário e a senha e recebendo de volta os dados do usuário logado (menos a senha, por motivos de segurança) e o *token* de acesso, que dura, como dito anteriormente, 24 horas. Esse *token* no *front-end* é guardado dentro dos *cookies*, que segundo (KASPERSKY, 2021) são arquivos de texto com pequenos pedaços de dados - como nome de usuário e senha - que são usados para identificar seu computador quando você usa uma rede de computadores. *Cookies* específicos, conhecidos como *cookies* HTTP, são usados para identificar usuários específicos e melhorar sua experiência de navegação na web e usado usando o *interceptors*, que é como o nome diz são interceptadores, que no Axios conseguimos usa-los para executar algo antes do Request e/ou Response seja iniciada, em todas as requisições do *front-end* para o *back-end*.

3.2.6 Node.js

O Node.js foi usado para o desenvolvimento do projeto pois, segundo (PEREIRA, 2014), o mesmo possui uma boa performance com relação ao consumo de memória, já que usa ao máximo e de forma eficiente o poder de processamento dos servidores, principalmente em sistema que produzem uma alta carga de processamento. Sistemas Node.js não sofrem de *dead-locks*, porque o Node.js trabalha apenas em *single-thread* (única *thread* por processo). Portanto, desenvolver sistemas nesse paradigma é simples, segundo os mesmos autores.

Outra grande vantagem é que o Node.js é altamente escalável e de baixo nível. Assim, o programa interage diretamente com diversos protocolos de rede e internet usando bibliotecas que acessam recursos do sistema operacional, principalmente em sistemas baseados em Unix.

Outra coisa interessante que podemos ver em (PEREIRA, 2014), é que o Node.js é orientado a eventos e segue a mesma filosofia de orientação de eventos do Javascript *client-side*. A única diferença é que não existem eventos de *click* do mouse ou afins. Na verdade, trabalhamos com eventos de I/O do servidor, como *connect* de um banco de dados. Portanto, o Node.js possui um *event-loop* que é o agente responsável por escutar e emitir eventos no sistema. Na prática ele é um *loop* infinito que, a cada iteração verifica em sua fila de eventos se um determinado foi emitido, levando o mesmo para a fila de executados.

Ainda em (PEREIRA, 2014), observamos que o Node.js possui o *design event-driver*, que foi insipirado nos *frameworks Event Machine* (do Ruby) e *Twisted* (do Python), porém o *event-loop* é mais performático, porque seu mecanismo é nativamente executado no modo "não bloqueante".

O modelo de microsserviço permite que os desenvolvedores possam dividir os componentes de uma infraestrutura de aplicativo maior. Como cada componente é executado de forma independente, o desenvolvedor pode atualizar ou modificar o componente sem afetar o aplicativo maior. Cada componente expõe uma interface para usuários externos que não conhecem nenhuma lógica interna do serviço.

No nosso caso, decidimos fazer três microsserviços, que funcionam independentes. Esse método não é o único para esta arquitetura, porém, foi o que acreditamos ser mais eficiente para esse projeto.

3.3 *front-end*

3.3.1 TypeScript

Segundo (BIERMAN; ABADI; TORGERSEN, 2014), TypeScript é uma extensão do JavaScript destinada a permitir o desenvolvimento mais fácil de aplicativos JavaScript em grande escala.

O Typescript como ferramenta de desenvolvimento tem várias vantagens. Um dos maiores problemas é encontrar erros no processo de implementação, pois pode ser utilizado o *Intellisense* do IDE, permitindo a visualização de pontos de melhoria e problemas de compilação.

O foco principal do Typescript é trazer a tipagem estática para Javascript, enquanto também adiciona alguns recursos para promover a aplicação de conceitos OOP também conhecido como programação orientada a objetos que é uma metodologia ou paradigma para projetar códigos usando classes e objetos simulando abstrações da vida real.

Podemos pensar no Typescript como um intensificador da linguagem Javascript. Ele permite o uso dessa linguagem para construir sistemas grandes e complexos sem quaisquer obstáculos.

3.3.2 React

Segundo (GACKENHEIMER, 2015), React é um *framework* javascript. React foi originalmente criado por engenheiros do Facebook para resolver os desafios envolvidos ao desenvolver interfaces de usuário complexas com conjuntos de dados que mudam com o tempo.

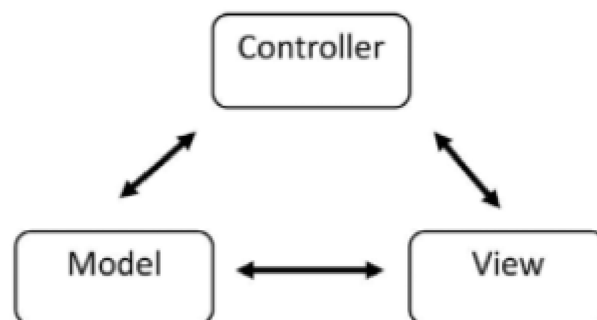

Figura 3.4 – MVC, model view controller, por (GACKENHEIMER, 2015)

O React funciona no modo MVC, *Model View Controller*. A figura 3.4 (GACKENHEIMER, 2015) mostra os fundamentos de cada um dos componentes em uma arquitetura de MVC. O modelo lida com o estado do aplicativo e envia eventos de mudança de estado para a *view*. A *view* é a aparência voltada para o usuário e a interface de interação para o usuário final. A *view* pode enviar eventos para o *controller* e, em alguns casos, para o *model*. O *controller* é o despache principal de eventos, que podem ser enviados para o *model*, para atualizar o estado, e a *view* para atualizar a apresentação.

Sobre o desempenho do ReactJS, segundo (AGGARWAL, 2018), o mesmo é conhecido por ser um executor altamente eficiente. Este é um dos principais fatores que tornam as estruturas destacam-se entre dezenas de *frameworks* existentes no mundo competitivo. A razão para um desempenho altamente eficiente do *framework* é essencialmente o recurso virtual DOM. O que acontece é que o ReactJS mantém um modelo de objeto de documento virtual dentro da memória.

Sempre que uma mudança deve ser refletida no página da web exibida atualmente, em vez de atualizar instantaneamente o DOM de navegação, as primeiras alterações na virtual DOM são feito. Depois que as alterações no DOM virtual são feitas, um algoritmo `diff()` é aplicado que compara o já salvo, o virtual DOM e o DOM de navegação e apenas os nós relevantes e desejados da árvore DOM do navegador são atualizados, o que resulta no desempenho extremamente rápido do aplicativo. Ou seja, apenas o que foi modificado é renderizado novamente, a parte que não há alteração, apenas é repetido o código.

3.3.2.1 Componentes

Os componentes React são como funções JavaScript. Eles aceitam entradas como propriedades e retornam novos elementos React, os chamados JSX.

Os componentes permitem que a UI seja dividida em partes independentes e reutilizáveis, ou seja, cada parte do aplicativo é tratada como um bloco independente.

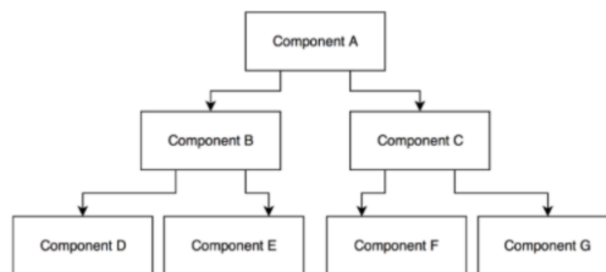

Figura 3.5 – Hierarquia de componentes, por (FEDOSEJEV, 2015)

Como visto na figura 3.5, também é possível fazer uma hierarquia de componentes para criar diferentes componentes provenientes de um componente pai, apenas passando o atributo *children*, dessa forma criando um bom reuso do código.

3.3.2.2 Contextos

Os contextos, segundo (PHAN, 2020), no React, tem o escopo que contém todos os dados do componente pai que os componentes filhos podem obter e usar.

O React segue um fluxo de dados unilateral. Isso significa que, ao projetar aplicativos React, os desenvolvedores geralmente aninham componentes filho em componentes-pai. Como os dados fluem apenas em uma direção, é mais fácil depurar erros e manter-se atualizado sobre onde está o problema no aplicativo, portanto as informações que ficam alocadas nos contextos é o que

O e-commerce possui três estados para usuários e vendedores, um com usuário logado, um com vendedor logado e um com usuário deslogado, desta forma sendo criados o AppContext

e o AuthContext que são contextos usados para o controle do que é mostrado ao usuário/vendedor ou convidado dentro da aplicação.

Quando o usuário ou vendedor é logado, guardamos o identificador do usuário logado e o *token* de acesso nos *cookies* e o passamos para o AuthContext. Dessa forma, todas as páginas passam a ter as informações de quem está logado e o *token* de acesso do mesmo.

3.3.2.2.1 AuthContext

O AuthContext, nada mais é que um contexto, como dito anteriormente, que guarda informações do usuário e de seu estado. Por exemplo, se está logado ou não, se é vendedor ou usuário, caso esteja logado, o Id, que é a chave primária da entidade "User"ou da entidade "Vendor". Esse Id é sempre incluído nas requisições ao *back-end* para receber as informações sobre o usuário ou vendedor que está logado.

3.3.2.2.2 AppContext

O AppContext salva as informações do usuário que esta navegando pelo e-commerce, dessa forma salvando no carrinho de compras os itens que o usuário deslogado deseja efetuar a compra.

3.3.3 NextJS

(CHU, 2020) afirma que Next.js é um framework React com uma estrutura que permite a construção de um React *front-end* aplicativo e lida de forma transparente com a renderização do lado do servidor. Isso resolve o problema de aplicativos React, que normalmente carregam o conteúdo depois que o javascript é carregado, resultando em SEO e experiência do cliente ruins. Next.js fornece renderização de servidor, permitindo o aplicativo para exibir seu estado inicial antes que qualquer script javascript seja carregado.

Dessa forma o *leak* de informações de renderização do lado de servidor para o *front-end* se torna transparente para o usuário, deixando a experiência mais limpa e agradável.

3.3.4 Serviços

Serviços é uma separação de arquivos de código que ajudam na não repetição de código, dessa forma como *service*, criamos um arquivo para cada API/*back-end*s, um para API de usuário, uma para API de vendedor e uma para API de compras e vendas.

3.3.4.1 Axios

Segundo (AXIOS, 2021), AXIOS é um projeto de código aberto e um cliente HTTP baseado em *Promisses* para fazer solicitações. Ele pode ser usado em navegadores e no NodeJS.

Dessa forma, todas as requisições feitas a qualquer um dos três *back-end*s são feitas através do Axios pelo *front-end*, onde cada url retorna as informações vigentes de determinados *endpoints*.

3.4 *Deploy*

Nos últimos anos e de forma ainda mais acelerada durante a pandemia que teve inicio em 2020, tem acontecido o "êxodo digital" por grande parte das empresas e usuários. Cada vez uma parcela maior dos negócios ganha espaço no mundo digital. Termos como"*Everything as Service* ou *Product as a Service*" ganham cada vez mais significado e peso no mercado. Existem alguns fatores que contribuem para isso, entre os quais podemos pontuar o maior acesso da população à Internet de alta velocidade e a variedade de equipamentos que funcionam com componentes conectados e controlados pela Internet. Computadores de bordo de carros, *smart houses* e *smart TV* são exemplos da Internet em nosso dia-a-dia.

Na área de TI não é diferente e as opções de *deploy* de serviços e produtos ganha cada vez mais opções. Algumas considerações devem ser observadas, no entanto, e entraremos um pouco nelas a seguir.

3.4.0.1 Panorama na área de TI

Entre as pessoas de TI (estudantes ou profissionais) que estão mais bem informadas sobre as novas tecnologias e possibilidades, também aconteceu o êxodo digital. Em especial entre os mais jovens e graduandos, os serviços em nuvem são tão presentes e práticos, que muitos sequer consideram a possibilidade de fazer o *deployment* em estrutura física própria. Embora os serviços em nuvem ofereçam uma enorme gama de opções e simplicidade para a realização de projetos dos mais variados escopos, eles no geral contam com algumas limitações. A mais simples de se observar é o aspecto da *performance*.

3.4.0.2 Virtualização

A fundação de toda a computação em nuvem é a virtualização de hardware. Foi o que possibilitou a expansão que vemos na última década. A medida que a tecnologia do hardware avançou, ficou evidente que havia desperdício de recursos computacionais. A virtualização entrou em cena, usando os mesmos recursos (processador, RAM, armazenamento, conexões de rede, etc...) para múltiplas instâncias de sistemas operacionais virtualmente separados, cada um executando tarefas variadas simultaneamente e otimizando os custos de operação e investimento de hardware.

3.4.0.3 *Datacenters* e Nuvem

Hoje, temos muitos grandes *players* no mercado de virtualização e serviços "em nuvem". Entre eles podemos citar a Microsoft com o serviço Azure, Amazon com o *AWS Web Services*, Google e seu *Google Cloud* e *Digital Ocean*, entre outros. Quase todos esses serviços contam com créditos iniciais gratuitos ou período de testes. Algumas outras plataformas oferecem serviços gratuitos. Da perspectiva de teste e desenvolvimento, o uso de *datacenters* "na nuvem" é uma abordagem flexível e cumpre a promessa de colocar um *app*, site ou serviço *online* e funcionando rapidamente visto que muitos desses serviços já tem suas API's embarcadas e serviços prontos para serem alimentados e compilados *on the fly*.

3.4.0.4 O aspecto *PERFORMANCE*

É importante observar as principais diferenças entre os servidores e computadores tradicionais para aspecto de comparação. Enquanto um computador tradicional moderno é direcionado para celeridade em tarefas pouco paralelas, servidores são direcionados para grande paralelismo em cargas mais complexas de trabalho. Além das quantidades de memória RAM disponíveis nesses sistemas, os processadores de servidores atuais têm muitos mais núcleos internos operando a uma frequência mais baixa.

Processador	Segmento	Cores/Threads	Clock Base / Boost	Max RAM	Soquetes
Ryzen 5950x	Doméstico	16/32	3.40Ghz/4.90Ghz	128GB	1P
Epyc 7763	Servidor	64/128	2.45Ghz/3.5Ghz	4 TB	1P/2P

Figura 3.6 – Configurações do servidor

Acima, os processadores da AMD atuais mais rápidos para *Desktop* e *Server*. Os *cores* são processadores físicos internos à estrutura do chip e *threads* são filas de processamento simultâneo que implementa um processador lógico. Assim, por exeplo, o Ryzen possui 12 Cores e 32 CPUs lógicas. Ou seja, cada *core* se "responsabiliza" por duas filas paralelas de processamento (CPUs lóigcas). A última coluna mostra quando soquetes de processadores são instaláveis em uma única máquina. O processador *Epyc* permite a instalação de 2 Processadores em uma única máquina, alcançando 128 *cores* e 256 *threads*. Considerando a ideia de dividir os recursos de uma máquina física entre múltiplas instâncias virtuais simultâneas, é natural que individualmente a performance das instâncias sofra uma queda de performance mediante cargas de trabalho mais intensas. As informações não são fáceis de serem encontradas nas plataformas, e costumam usar métricas e termos internos dessas empresas.

Usando essa tabela da imagem 3.7 como exemplo, podemos observar.

Dyno Type	Memory (RAM)	CPU Share	Compute	Dedicated	Sleeps
free	512 MB	1x	1x-4x	no	yes
hobby	512 MB	1x	1x-4x	no	no
standard-1x	512 MB	1x	1x-4x	no	no
standard-2x	1024 MB	2x	4x-8x	no	no
performance-m	2.5 GB	100%	12x	yes	no
performance-l	14 GB	100%	50x	yes	no

Figura 3.7 – Tabela do Heroku

Embora *CPU Share* e *Compute* fique em um conceito vago que pode significar qualquer métrica interna da plataforma, RAM, *Dedicated* e *Sleeps* são conceitos universais. Nas atuais tecnologias, 512MB de RAM serão limitantes de performance em uma variedade de novos compiladores e linguagens. Se assumirmos que *CPU Share 1x* significa 1 *thread* de um processador com *Dedicated* "no", consideramos que todo o processamento da tarefa estará sujeito a processamento sequencial dividido em fila única, em um processador típico de servidor que costumeiramente não tem *clock* elevado. Nas outras plataformas, as escalas diferem, mas o conceito em si é o mesmo. Na plataforma Netlify temos 3GB de RAM e 1 CPU (não especifica se seria um *thread* ou um *core*). Novamente, embora haja mais cache em RAM e processos, sem dúvida, um único *thread* de processamento é pouco para tarefas mais complexas.

3.4.0.5 Aspectos indiretos consideráveis

Outras questões a serem observadas são:

- Segurança dos dados
- *Uptime* de conexão e serviço.

Todas as plataformas de computação em nuvem têm investimentos constantes em segurança geral da plataforma, *firewalls* e outros mecanismos para evitar qualquer tipo de invasão ou exposição dos dados nela hospedados.

Como o "negócio" (*core business*) dessas plataformas é a venda da infra estrutura como serviço, toda a estrutura de *datacenters* conta com múltiplos *links* dedicados de Internet para mitigar quedas e equilibrar a carga e para manter tudo *online* o máximo possível.

Algumas dessas plataformas contam com *datacenters* em locais diferentes separados por centenas de kilômetros de distância, mantendo cópias de tudo em mais de um lugar. Bancos de

nobreaks e geradores sempre ficam a disposição para assumir a geração em cenário de interrupção de energia.

Em um ambiente local, mesmo que medidas sejam tomadas, muito dificilmente pode-se alcançar indicadores similares de redundância e segurança das alcançadas por *datacenters* dedicados a esses segmentos.

3.4.0.6 Topologia e *Deploy* Local

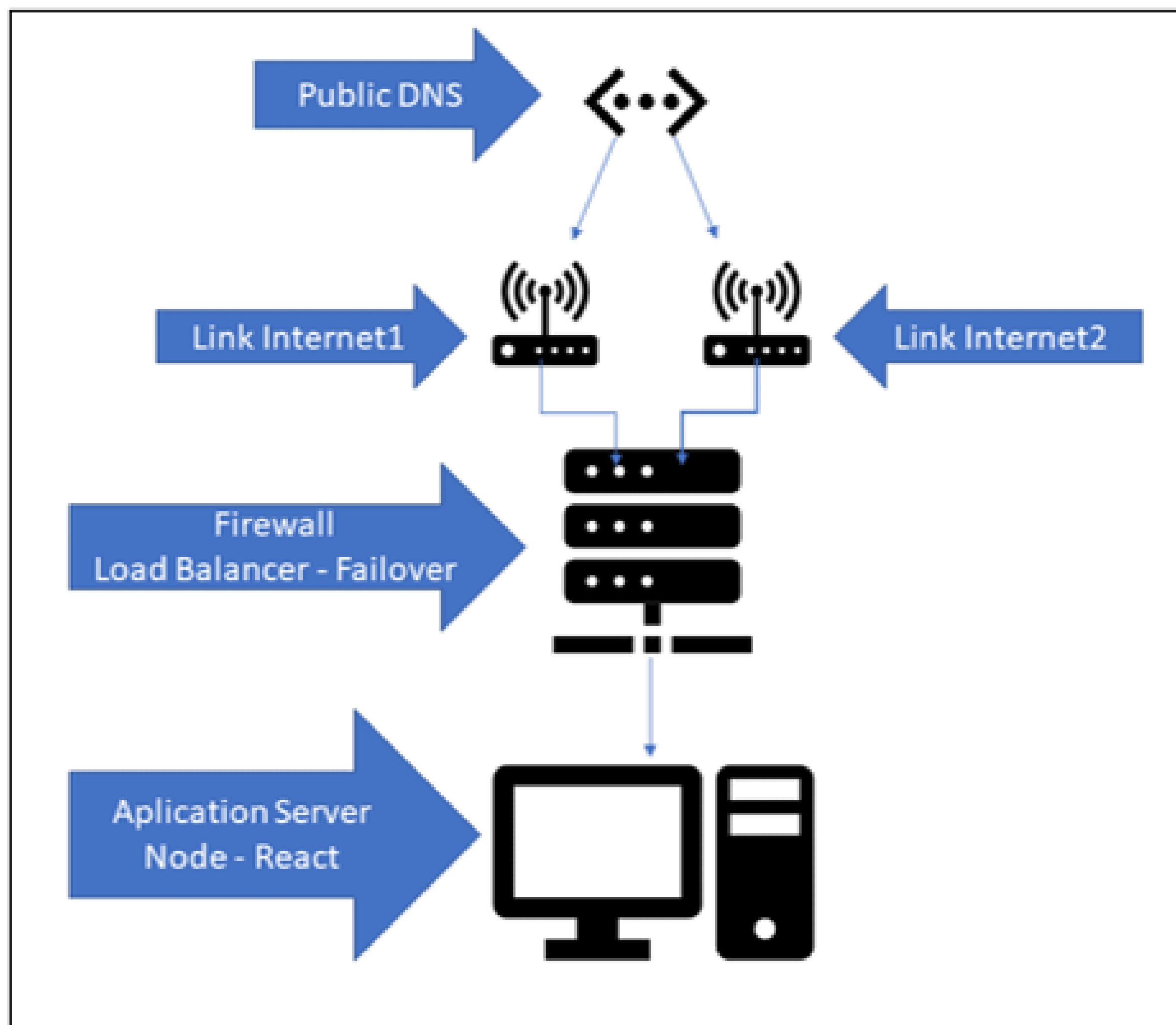

Figura 3.8 – Topologia

O cenário local segue diagrama acima. São 2 provedores de internet com IPs Fixos de internet apontados através de DNS Público colocados manualmente. Em caso de queda do primeiro, precisa-se alterar manualmente para o secundário.

Existem ferramentas para fazer verificação de queda e troca, mas como nessa proposta o *uptime* 24/7 não é indispensável, optamos por não arcar com esse investimento.

O *Firewall* está configurado adequadamente seguindo diretiva de todas as portas fechadas para entrada exceto as necessárias para funcionamento do serviço configurado no *Deploy*. O Windows Server do servidor de aplicação segue diretiva de bloqueio de usuário após 3 tentativas incorretas de logon, virtualmente impedindo tentativa de quebra de senha por *brute force*.

Conforme observamos nos recursos de hardware alocados em nuvem, não é preciso muito para que localmente tenhamos mais recursos/performance disponível para a aplicação. Qualquer coisa acima de 2 núcleos de processador e 4gb de RAM bastaria para melhorar em métrica de *performance*.

,

4 Resultados

Os resultados obtidos do acoplamento entre todas essas tecnologias foi satisfatório, pois todas as funções do e-commerce funcionam e também funcionam independentes, dessa forma, garantindo o conceito de microserviços.

O gargalo maior do *deploy* em uma máquina só, onde todos os serviços de *back-end*s e de *front-end* estão em um mesmo *host*, é que, caso a mesma caia por qualquer que seja o motivo, fará todo os serviço cair. Portanto, se faz necessário separar os serviços de *front-end*s e de *back-end*s em diferentes máquinas fazendo que, caso algum serviço de *back-end* caia, os demais serviços continuem funcionando.

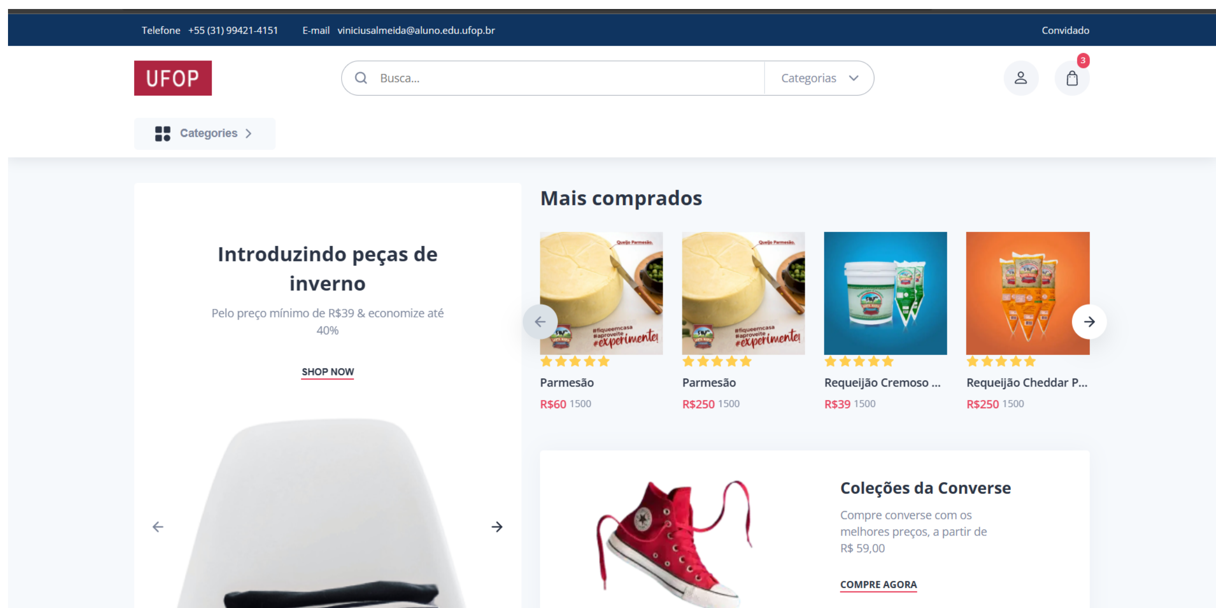

Figura 4.1 – Tela inicial com vendedor logado

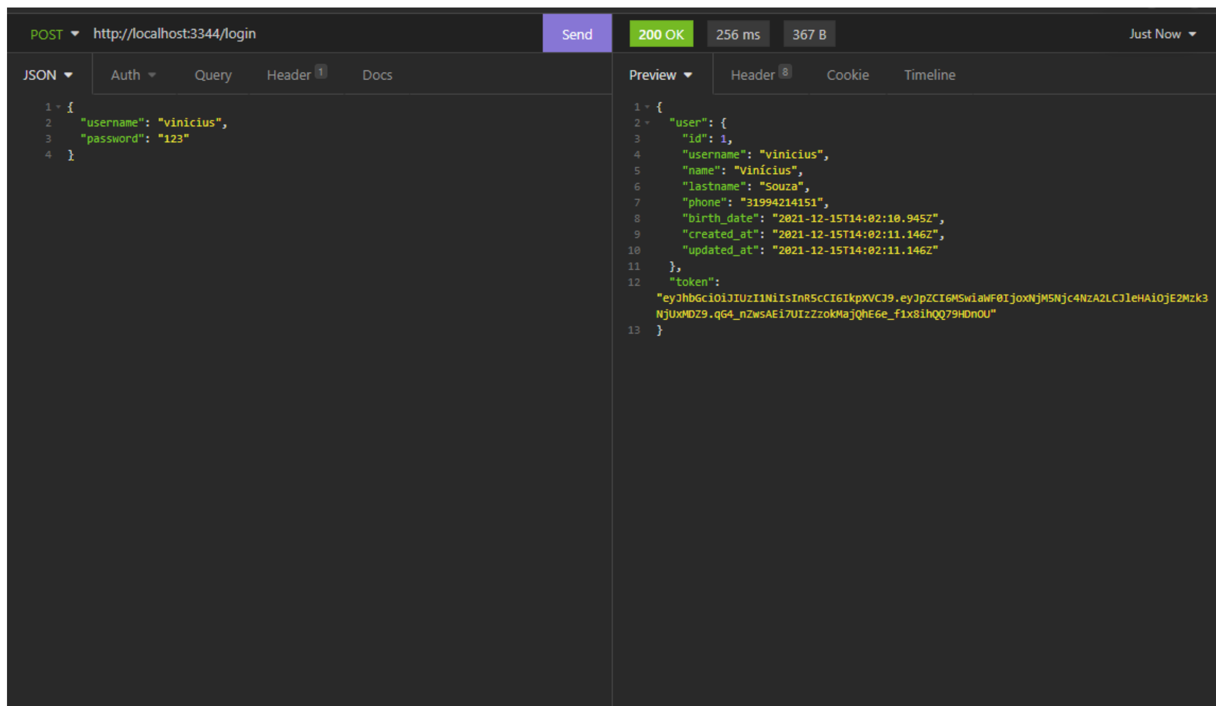

Figura 4.2 – Usuário comum sendo logado independente de outras API’s na API de usuário

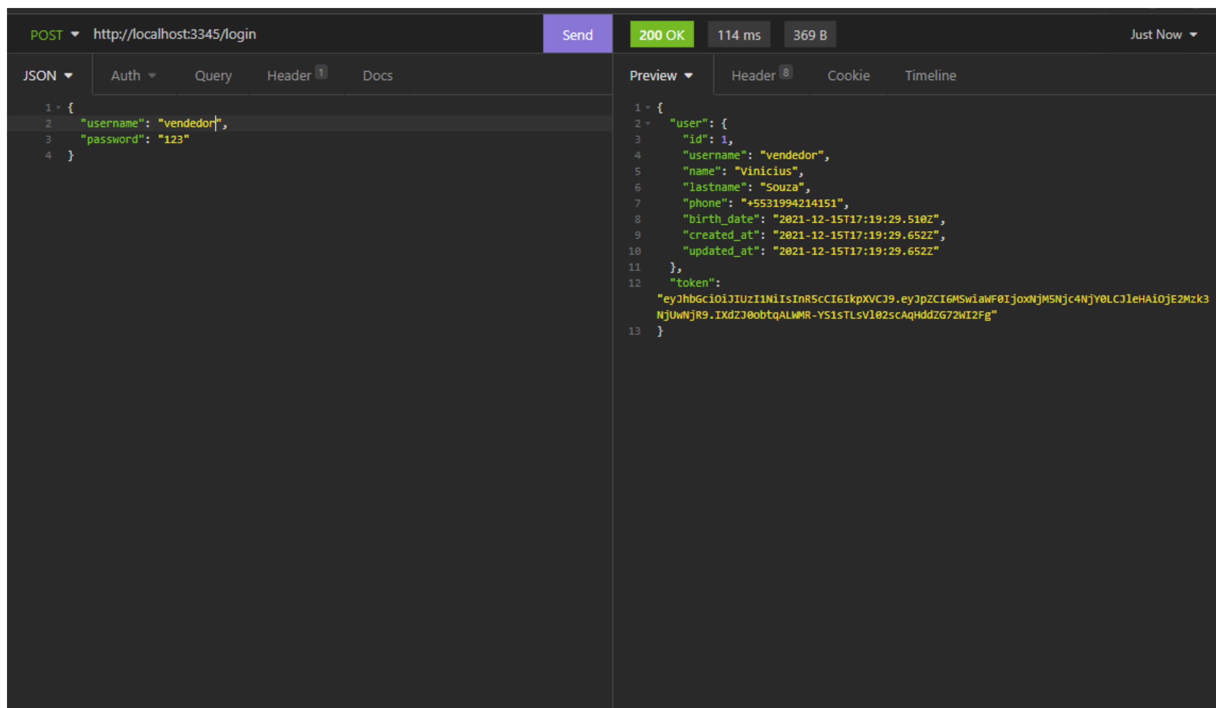

Figura 4.3 – Vendedor comum sendo logado independente de outras API’s na API de Vendedor

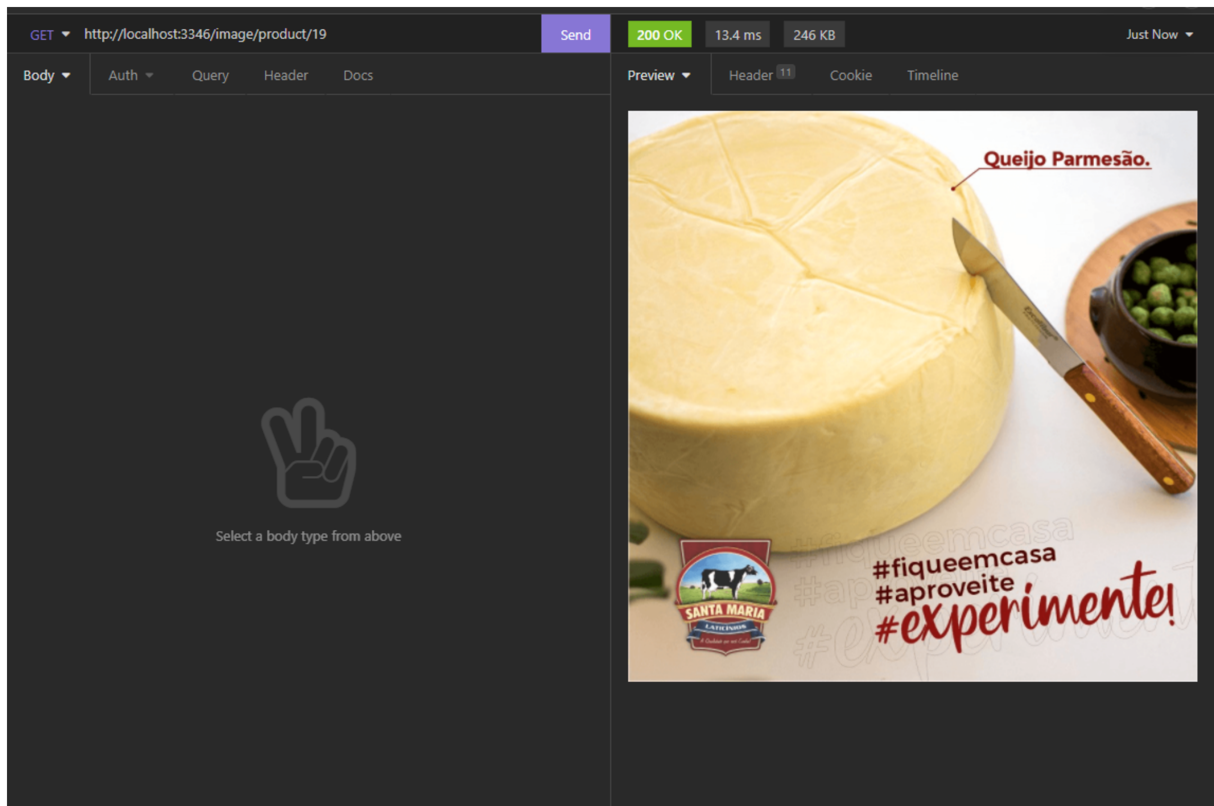

Figura 4.4 – Retorno de uma imagem de um determinado produto de uma venda com o id 19

No quesito *deploy*, o mesmo foi feito em diferentes ambientes. Neste documento iremos mostrar alguns pontos do *deploy* feito em ambiente Amazon Web Services, para fins de comparação, ao *deploy* em servidor local. Em ambas situações, o *deploy* foi bem sucedido, ou seja, as funcionalidades se manteram, tanto com ambiente virtualizado amazon, como em servidor local.

4.1 Estudos em bancos de dados com servidor local e online

Fizemos também um estudo sobre a *performance* dos microserviços hospedados em um servidor local, como descrito no capítulo anterior, e um servidor de serviços online. Para exemplificar, o *deploy* foi feito no Heroku, onde está hospedado o banco de dados e o *backend* nesse ambiente, com a seguinte configuração:

Para fins acadêmicos, usamos o plano *Hobby Free*, que limita o tamanho do banco, a quantidade de colunas e a quantidade de conexões simultâneas além de ter as seguintes configurações.

Portanto, para ter um banco de dados dedicado com os valores próximos do que o local tem a oferecer seria cerca de $600 por mês, para ter as configurações da figura 4.7. Claro que, o banco de dados *Private 2* do Heroku dificilmente teria alguns tipos de quedas, pois estes serviços possuem vários links de internet e geradores para o caso de queda de energia ou de internet, dessa forma, sempre que formos criar ambientes, seja online, ou local para fazermos os *deploys* das

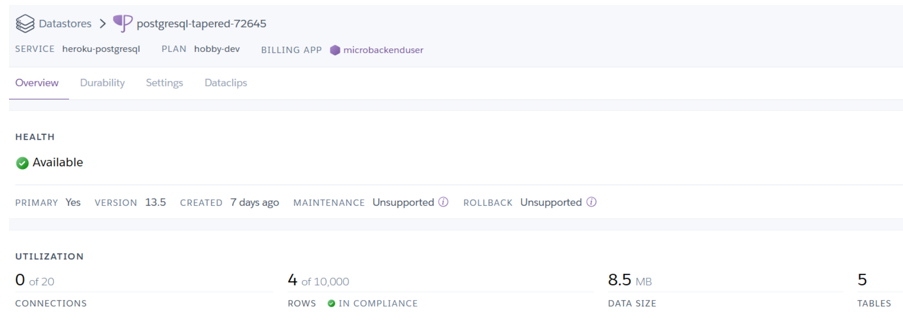

Figura 4.5 – Configuração do banco de dados PostgreSQL no heroku

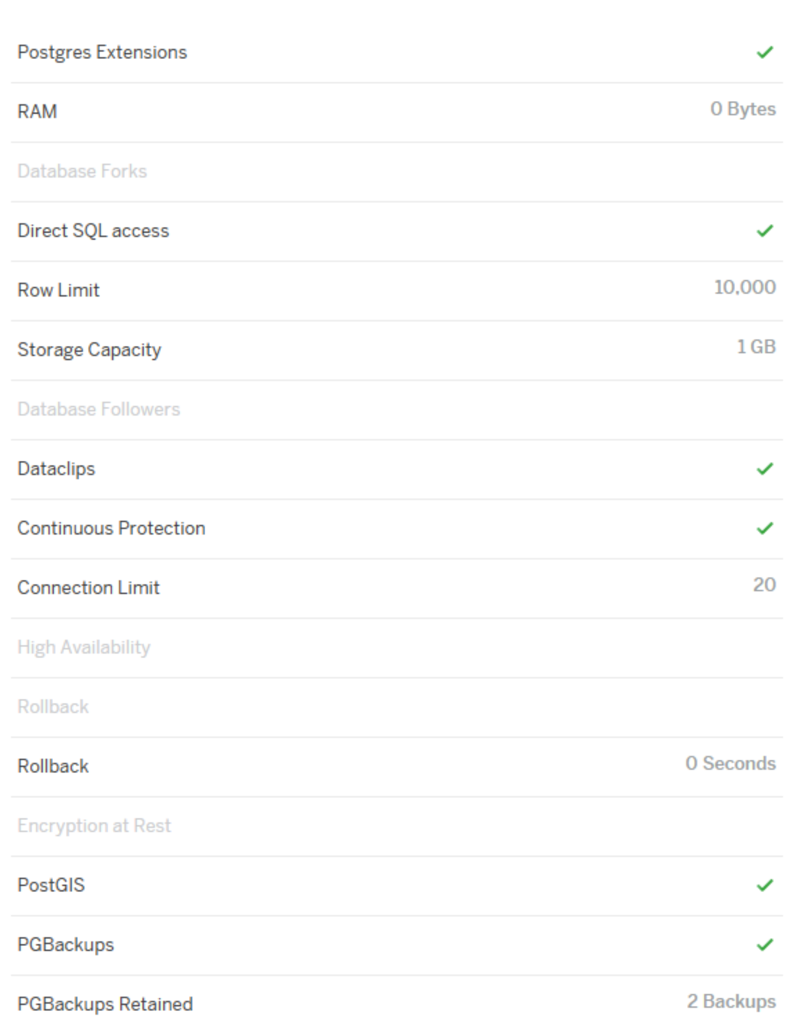

Figura 4.6 – Detalhes do banco de dados PostgreSQL Hobby dev no heroku

aplicações, devemos pensar nas redundâncias de funcionamento, que nesse caso seriam links diferentes de internet e geradores de energia, e no custo benefício delas.

Portanto a partir disso, podemos ver como é importante o profissional que faz o *deploy*, pois o mesmo, consegue otimizar a performance, fazendo com que seja necessário menos *hardware*.

Postgres Extensions	✓
RAM	8 GB
Database Forks	✓
Direct SQL access	
Row Limit	
Storage Capacity	256 GB
Database Followers	✓
Dataclips	✓
Continuous Protection	✓
Connection Limit	400
High Availability	✓
Rollback	✓
Rollback	7 Days
Encryption at Rest	✓
PostGIS	✓
PGBackups	✓
PGBackups Retained	50 Backups
Heroku Private Spaces	✓
Burstable Performance	
Heroku Postgres via PrivateLink	✓
Heroku Postgres via mutual TLS	✓
HIPAA compliance	
BYOK (Bring Your Own Key)	✓

Figura 4.7 – Detalhes do banco de dados PostgreSQL Private 2 no heroku

4.1.1 Testes de performance

Nas figuras abaixo mostram algumas diferenças de performance entre os dois servidores em chamadas idênticas e muito pequenas, portanto, quanto menor o retorno da chamada, menos diferença em performance são percebidos.

Tempo de resposta por inserção de usuários		
Testes	Servidor local	Servidor Heroku
Usuário 1	275ms	708ms
Usuário 2	84.5ms	242ms
Usuário 3	86.5ms	666ms
Usuário 4	92.8ms	248ms
Usuário 5	84.7ms	256ms

Figura 4.8 – Testes de inserção de usuário em milissegundos

Média do tempo de resposta por inserção de usuários		
Testes	Servidor local	Servidor Heroku
Média	124.7ms	424ms

Figura 4.9 – Média de inserção de usuário em milissegundos

Tempo de resposta por login de usuários		
Testes	Servidor local	Servidor Heroku
Usuário 1	76ms	265ms
Usuário 2	73.1ms	267ms
Usuário 3	78.5ms	234ms
Usuário 4	74.1ms	231ms
Usuário 5	81.8ms	269ms

Figura 4.10 – Testes de login de usuário em milissegundos

Média do tempo de resposta por login de usuários		
Testes	Servidor local	Servidor Heroku
Média	76.7ms	253.2ms

Figura 4.11 – Média de login de usuário em milissegundos

5 Considerações Finais

5.1 Conclusão

Deste ponto, podemos concluir que, o funcionamento geral de um e-commerce, feito a partir de três microserviços distintos é funcional e dessa forma, atingimos nosso objetivo, todas as funcionalidades para o funcionamento em ambiente puramente acadêmico esta funcionando, pois não foi vinculado nenhuma plataforma de pagamento online, apesar do acoplamento para o mesmo não ser complexo em NodeJS ou React.

Além de que, nesse caso, podemos ver que mesmo em chamadas pequenas, com retornos muito rápidos ainda há uma diferença bem considerável de *performance*, porém, é bem difícil criar redundâncias em ambiente local, pois o valor para a compra dos hardwares necessários para que a redundância seja bem considerável a curto prazo, entretanto, se o projeto ficar grande demais e ser necessário mais hardware para o bom funcionamento, os servidores próprios são a opção mais viável em termos financeiros.

Referências

AGGARWAL, S. Modern web-development using reactjs. *International Journal of Recent Research Aspects*, v. 5, n. 1, p. 133–137, 2018.

AXIOS. *Introdução*. 2021. Disponível em:<https://axios-http.com/ptbr/docs/intro>. Acesso em: 28 de novembro 2021.

BALDISSERA, O. *Quais são os tipos de arquitetura de software e como escolher o melhor para seu projeto*. 2021. Disponível em:<https://posdigital.pucpr.br/blog/tipos-de-arquitetura-de-software>. Acesso em: 29 de novembro 2021.

BIERMAN, G.; ABADI, M.; TORGERSEN, M. Understanding typescript. In: SPRINGER. *European Conference on Object-Oriented Programming*. [S.l.], 2014. p. 257–281.

CHU, S. Shopify upsell app: Using next. js, react. js to boost sale. 2020.

DBFORGESTUDIO. *Query Builder Overview*. 2021. Disponível em:<https://docs.devart.com/studio-for-mysql/building-queries-with-query-builder/query-builder-overview.html>. Acesso em: 02 de dezembro 2021.

DMITRY, N.; MANFRED, S.-S. On micro-services architecture. *International Journal of Open Information Technologies*, . …, v. 2, n. 9, 2014.

DRAKE, J. D.; WORSLEY, J. C. *Practical PostgreSQL*. [S.l.]: "O'Reilly Media, Inc.", 2002.

FACEBOOK, D. *Tokens de acesso*. Disponível em:<https://developers.facebook.com/docs/facebook-login/access-tokens/>. Acesso em: 03 de dezembro 2021.

FEDOSEJEV, A. *React. js essentials*. [S.l.]: Packt Publishing Ltd, 2015.

GACKENHEIMER, C. *Introduction to React*. [S.l.]: Apress, 2015.

GONZALEZ, D. *Developing microservices with node. js*. [S.l.]: Packt Publishing, 2016.

GOUIGOUX, J.-P.; TAMZALIT, D. From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture. In: IEEE. *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. [S.l.], 2017. p. 62–65.

HENRIQUE, M. *Microsserviços vs SOA*. 2021. Disponível em:<https://medium.com/tecnologia-e-afins/microsserviços-vs-soa-79c0b288207>. Acesso em: 25 de novembro 2021.

KASPERSKY. *What are Cookies?* 2021. Disponível em:<https://www.kaspersky.com/resource-center/definitions/cookies>. Acesso em: 01 de dezembro 2021.

KNEXJS. *Migrations*. 2021. Disponível em:<https://knexjs.org/#Migrations>. Acesso em: 02 de dezembro 2021.

LEARNING, P. *Postman Learning Center*. 2021. Disponível em:<https://learning.postman.com/docs/getting-started/sending-the-first-request/>. Acesso em: 02 de dezembro 2021.

ORACLE. *What is a relational database*. 2021. Disponível em:<https://www.oracle.com/br/database/what-is-a-relational-database>. Acesso em: 26 de novembro 2021.

PEREIRA, C. R. *Aplicações web real-time com Node. js*. [S.l.]: Editora Casa do Código, 2014.

PHAN, H. D. React framework: concept and implementation. 2020.

REDHAT. *O que são microsserviços?* 2021. Disponível em:<https://www.redhat.com/pt-br/topics/microservices/what-are-microservices>. Acesso em: 27 de novembro 2021.

SHAIKH, I. *THE 3 STAGES OF MICROSERVICE: DIVIDE, CONQUER, AND CHAOS*. 2021. Disponível em:<https://coderstea.in/post/microservices/microservices-introduction-divide-conquer-and-chaos/>. Acesso em: 31 de dezembro 2021.

Apêndices

APÊNDICE A – Chamadas aos servidores

Nesta apêndice iremos mostrar algumas telas de inserção no servidor Heroku e Local]

A.1 Heroku - Inserção

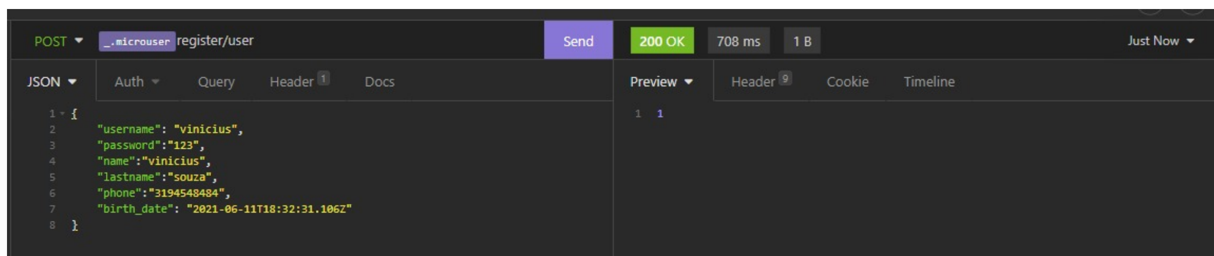

Figura A.1 – Inserção 1 no servidor Heroku

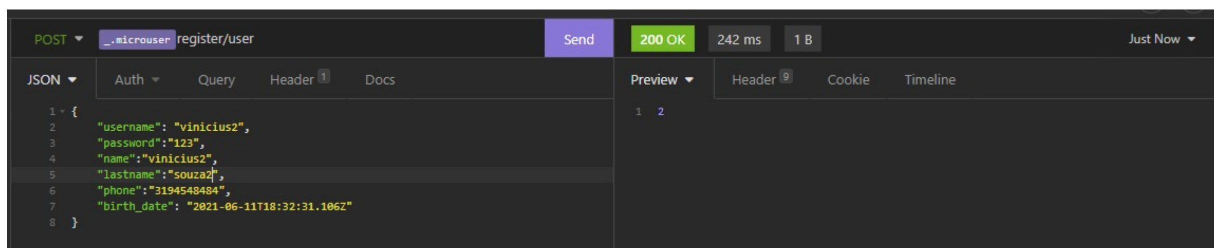

Figura A.2 – Inserção 2 no servidor Heroku

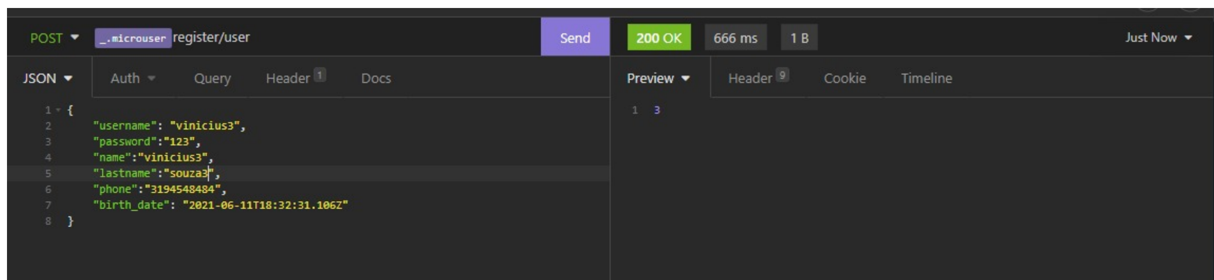

Figura A.3 – Inserção 3 no servidor Heroku

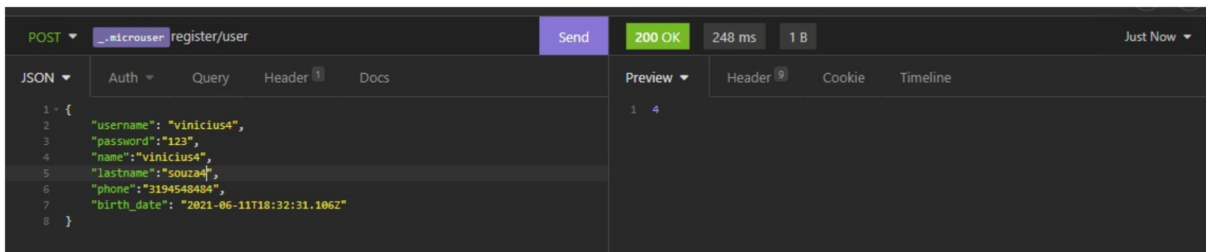

Figura A.4 – Inserção 4 no servidor Heroku

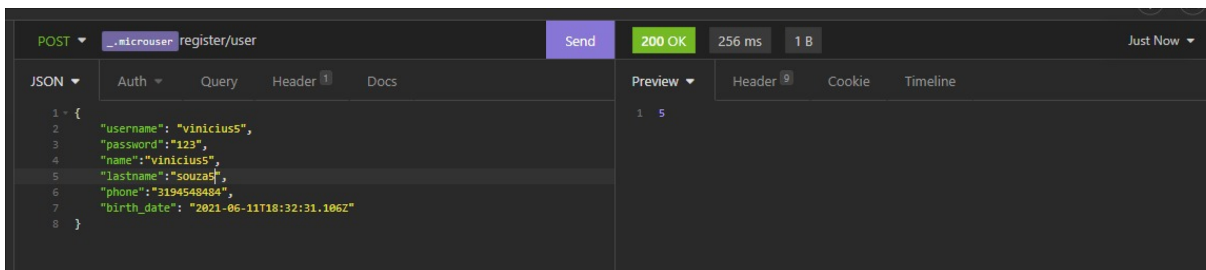

Figura A.5 – Inserção 5 no servidor Heroku

A.2 Local - Inserção

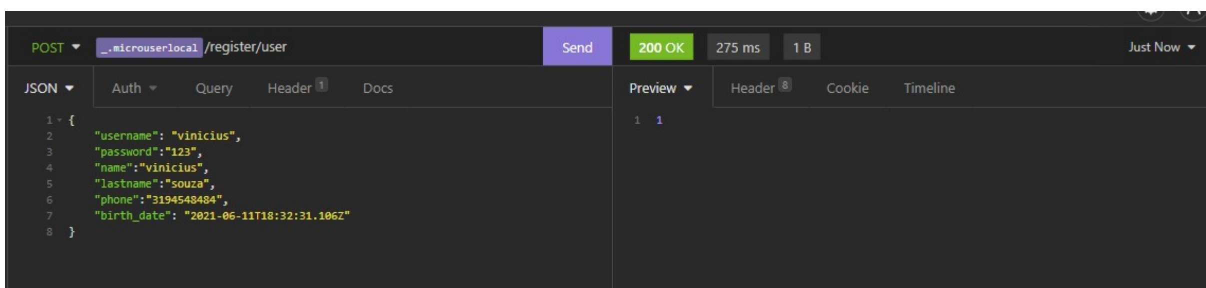

Figura A.6 – Inserção 1 no servidor local

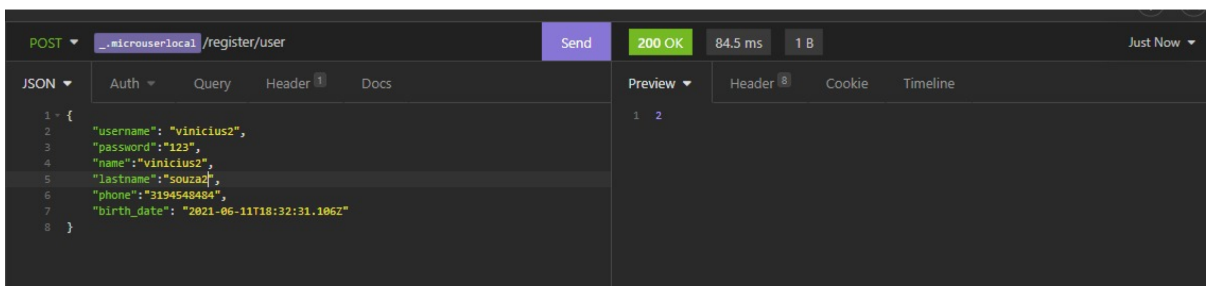

Figura A.7 – Inserção 2 no servidor local

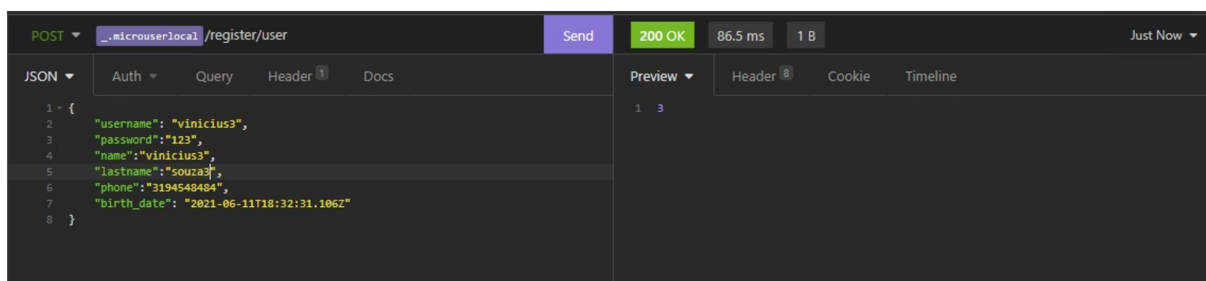

Figura A.8 – Inserção 3 no servidor local

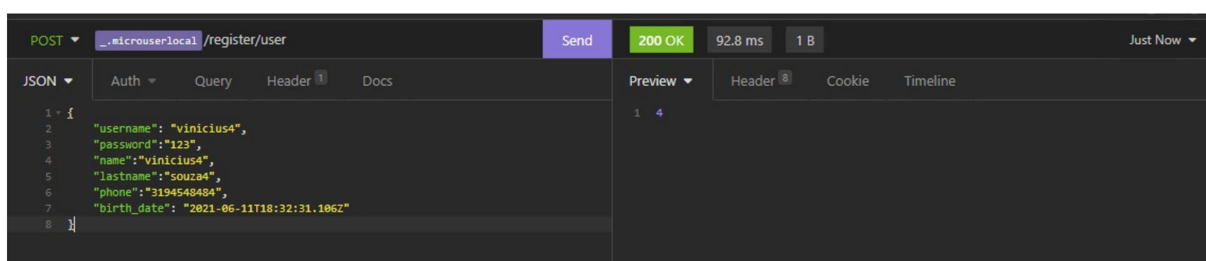

Figura A.9 – Inserção 4 no servidor local

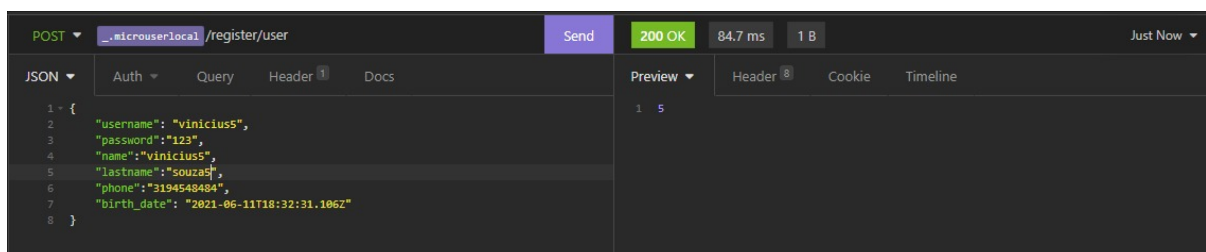

Figura A.10 – Inserção 5 no servidor local