**FACULDADE DE TECNOLOGIA DE SÃO JOSÉ DOS CAMPOS**

**FATEC PROFESSOR Jessen Vidal**

**JONATHAS HENRIQUE DE MORAES**

**APLICAÇÃO DE MICROSSERVIÇOS EM SISTEMA DE E-COMMERCE**

São José dos Campos

2021

**JONATHAS HENRIQUE DE MORAES**

**APLICAÇÃO DE MICROSSERVIÇOS EM SISTEMA DE E-COMMERCE**

Trabalho de Graduação apresentado à Faculdade de Tecnologia de São José dos Campos, como parte dos requisitos necessários para a obtenção do título de Tecnólogo em Banco de Dados.

**Orientador: Me. Giuliano Araújo Bertoti**

São José dos Campos

2021

**Dados Internacionais de Catalogação-na-Publicação (CIP)**

**Divisão de Informação e Documentação**

MORAES, Jonathas Henrique de

Aplicação de Microsserviços em Sistema de E-commerce.

São José dos Campos, 2021.

60f.

Trabalho de Graduação – Curso de Tecnologia em Banco de Dados.

FATEC de São José dos Campos: Professor Jessen Vidal, 2021.

Orientador: Me., Giuliano Araújo Bertoti

1. Microsserviços. 2. Docker. 3. Apache Kafka. I. Faculdade de Tecnologia. FATEC de São José dos Campos: Professor Jessen Vidal. Divisão de Informação e Documentação. II. Aplicação de Microsserviços em Sistema de E-commerce

**REFERÊNCIA BIBLIOGRÁFICA**

MORAES, Jonathas Henrique de. **Aplicação de Microsserviços em Sistema de E-commerce.** 2021. 60f. Trabalho de Graduação - FATEC de São José dos Campos: Professor Jessen Vidal.

**CESSÃO DE DIREITOS**

NOME DO AUTOR: Jonathas Henrique de Moraes

TÍTULO DO TRABALHO: Aplicação de Microsserviços em Sistema de E-commerce

TIPO DO TRABALHO/ANO: Trabalho de Graduação/2021.

É concedida à FATEC de São José dos Campos: Professor Jessen Vidal permissão para reproduzir cópias deste Trabalho e para emprestar ou vender cópias somente para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte deste Trabalho pode ser reproduzida sem a autorização do autor.

|  |  |
| --- | --- |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Jonathas Henrique de Moraes  R. Icatu, 390. Cj. Res. Trinta e Um de Março  12237-010, São José dos Campos – SP |  |

**JONATHAS HENRIQUE DE MORAES**

**APLICAÇÃO DE MICROSSERVIÇOS EM SISTEMA DE E-COMMERCE**

Trabalho de Graduação apresentado à Faculdade de Tecnologia de São José dos Campos, como parte dos requisitos necessários para a obtenção do título de Tecnólogo em Banco de Dados.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Me., José Walmir Gonçalves Duque - FATEC**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Tecnólogo, Samuel Pereira Machado Alves da Silva - FATEC**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Me., Giuliano Araújo Bertoti - FATEC**

**\_\_\_\_\_/\_\_\_\_\_/\_\_\_\_\_**

**DATA DA APROVAÇÃO**

Dedico este trabalho a minha família, pelo total apoio e compreensão durante esse tempo, e a meu orientador, pelo suporte constante na sua elaboração.

**AGRADECIMENTOS**

Agradeço ao professor Me. Giuliano Bertoti pela confiança depositada em mim, pelas dicas e materiais disponibilizados para que seja possível finalizar este trabalho.

Agradeço a todos os professores da Faculdade de Tecnologia de São José dos Campos - Professor Jessen Vidal.

Agradeço a todos os colegas e amigos que contribuíram de alguma forma para a realização deste trabalho.

**RESUMO**

A manutenção de um sistema disponível na internet pode se tornar um desafio no caso de um aumento repentino da demanda de acessos. Com a pandemia, muitos sistemas se depararam com este aumento repentino e enfrentam a necessidade de escalar seus serviços e aumentar a disponibilidade para seus clientes. Uma alternativa para uma maior agilidade ao escalar um sistema é construí-lo em componentes e separar cada um deles em diferentes servidores, possibilitando a redundância de instâncias de um mesmo recurso do sistema. Desta forma, o objetivo deste trabalho é construir um sistema de microsserviços exemplificando a aplicação do padrão em um serviço de *e-commerce*. Neste trabalho desenvolveu-se um sistema que separa funcionalidades em componentes desacoplados e executados em contêineres Docker, possibilitando a escalabilidade por meio da adição de novos serviços, além de permitir a redundância das instâncias de cada funcionalidade. Para que não haja concorrência no acesso ao banco de dados do sistema, cada funcionalidade foi desenvolvida com seu próprio banco de dados, e na eventual necessidade de replicar dados de um serviço para outro, foram utilizados eventos do Apache Kafka. Para atender o objetivo proposto, o sistema se mostrou eficaz, replicando os dados quando necessário e disponibilizando todos os recursos planejados. Entretanto, para um maior controle das instâncias de cada funcionalidade ativa, será necessária a adição de um orquestrador de contêineres Docker, que oferecem gerenciamento de rede, estado atual e escala dos serviços em execução.

**Palavras-Chave**: Microsserviços, Docker, Apache Kafka, Arquitetura Orientada a Serviços, Arquitetura Orientada a Eventos, GraphQL.

**ABSTRACT**

Maintaining an internet available system can become a challenge on a sudden access increase demand. With the pandemic, many systems were faced with this sudden increase and faced the need to scale their services and increase availability to their customers. An alternative for greater agility when scaling a system is to build it in components and separate each one of them in different servers, allowing the redundancy of instances from the same resource. Thereby, the objective of this work is to build a microservices system exemplifying the pattern application in an e-commerce service. This work has developed a system that separates functionalities into decoupled components that run in Docker containers, allowing scalability through the addition of new services, also allowing the redundancy of each functionality instance. To avoid concurrence in database accessing, each functionality was developed with its database and in the eventual need to replicate data from one service to another, Apache Kafka events were used. To comply with the proposed objective, the system proved to be effective, replicating data when necessary and providing all planned resources. However, for greater control over the instances of each active functionality, it will be necessary to add one docker container Orchestrator, which allows you to manage the network, current state, and scale of running services.

**Keywords**: Microservices, Docker, Apache Kafka, Service-Oriented Architecture, Event-Driven Architecture, GraphQL.

**LISTA DE FIGURAS**

Figura 1 - Comunicação entre produtor e consumidor por meio de tópicos. 23

Figura 2 – Arquitetura do sistema de Microsserviços. 26

Figura 3 - Classes do serviço de usuário. 27

Figura 4 - Tipos do serviço de usuário. 28

Figura 5 - *Queries* do serviço de usuário. 28

Figura 6 - *Inputs* do serviço de usuário. 29

Figura 7 - *Mutations* do serviço de usuário 30

Figura 8 - Entidades do serviço de estoque. 31

Figura 9 - *Types* do serviço de estoque 32

Figura 10 - *Queries* do serviço de estoque. 32

Figura 11 - *Inputs* do serviço de estoque. 33

Figura 12 - *Mutations* do serviço de estoque. 34

Figura 13 - Entidades do serviço de compras. 35

Figura 14 - DTOs do serviço de compras. 36

Figura 15 - *Controllers* do serviço de compras. 37

Figura 16 - Entidades do serviço de pagamentos. 38

Figura 17 - Tipos de dados do serviço de pagamentos. 39

Figura 18 - *Queries* do serviço de pagamentos. 40

Figura 19 - *Inputs* do serviço de pagamentos. 41

Figura 20 - *Mutations* do serviço de pagamentos. 42

Figura 21 - Declaração da classe gerenciadora de requisições e método de GET da classe 43

Figura 22 - Método de requisição POST 44

Figura 23 - Método de requisição PATCH 44

Figura 24 - Método de requisição DELETE 45

Figura 25 - Requisição de criação de usuário 46

Figura 26 - Constantes para o mapeamento do retorno da API GraphQL 46

Figura 27 - Dockerfile da API Gateway 47

Figura 28 – Configurações para os serviços do Kafka e Zookeeper 48

Figura 29 - Configurações para o serviço de usuário e seu banco de dados 49

Figura 30 - Configurações do serviço de API *Gateway* 50

Figura 31 - Requisição para criação de usuário. 54

Figura 32 - Resposta da requisição de criação de usuário. 55

Figura 33 - Envio de mensagem Kafka ao tópico *USER\_CREATED* após criação de usuário. 55

Figura 34 - Consumo de mensagem do tópico *USER\_CREATED.* 55

Figura 35 - Dados inseridos no banco do serviço de usuários. 56

Figura 36 - Dados inseridos no banco do serviço de pagamentos 56

**LISTA DE TABELAS**

[Tabela 1 - Exemplo de recursos e operações - API REST 21](#_Toc88060132)

**LISTA DE ABREVIATURAS E SIGLAS**

API Interface de Programação de Aplicativos (*Application Programming Interface*)

REST Transferência Representacional de Estado (*Representational State Transfer*)

HTTP Protocolo de Transferência de Hipertexto (*Hypertext Transfer Protocol*)

DTO Objeto de Transferência de Dados (*Data Transfer Object*)

ESB *Enterprise Service Bus*

SOA Arquitetura Orientada a Serviços (*Service-Oriented Architecture*)

URI Identificador Uniforme de Recurso (*Uniform Resource Identifier*)

JSON Notação de Objeto JavaScript (*JavaScript Object Notation*)

**SUMÁRIO**

[1. INTRODUÇÃO 15](#_Toc88238381)

[1.1. Objetivos do Trabalho 15](#_Toc88238382)

[1.2. Conteúdo do Trabalho 15](#_Toc88238383)

[2. FUNDAMENTAÇÃO TÉCNICA 17](#_Toc88238384)

[2.1. Monólito e Microsserviços 17](#_Toc88238385)

[2.2. Virtualização 17](#_Toc88238386)

[2.3. Máquinas Virtuais e Docker 18](#_Toc88238387)

[2.4. Arquitetura Orientada a Serviços (*Service-Oriented Architecture* - SOA) 18](#_Toc88238388)

[2.5. Event-driven Architecture 19](#_Toc88238389)

[2.6. Comunicação entre Microsserviços 19](#_Toc88238390)

[2.7. Armazenamento de Dados 20](#_Toc88238391)

[2.8. REST 21](#_Toc88238392)

[2.9. GraphQL 22](#_Toc88238393)

[2.10. Apache Kafka 22](#_Toc88238394)

[3. DESENVOLVIMENTO 25](#_Toc88238395)

[3.1. Arquitetura do Sistema 25](#_Toc88238396)

[**3.1.1. User Service** 26](#_Toc88238397)

[**3.1.1.1. Entidades** 26](#_Toc88238398)

[**3.1.1.2. *Types*** 27](#_Toc88238399)

[**3.1.1.3. *Queries*** 28](#_Toc88238400)

[**3.1.1.4. *Inputs*** 29](#_Toc88238401)

[**3.1.1.5. *Mutations*** 29](#_Toc88238402)

[3.1.2. Store Service 30](#_Toc88238403)

[3.1.2.1. Entidades 30](#_Toc88238404)

[3.1.2.2. *Types* 31](#_Toc88238405)

[3.1.2.3. *Queries* 32](#_Toc88238406)

[3.1.2.4. *Inputs* 33](#_Toc88238407)

[3.1.2.5. *Mutations* 33](#_Toc88238408)

[3.1.3. Order Service 34](#_Toc88238409)

[3.1.3.1. Entidades 34](#_Toc88238410)

[3.1.3.2. *Data Transfer Objects* (DTOs) 35](#_Toc88238411)

[3.1.3.3. *Controllers* 36](#_Toc88238412)

[3.1.4. Payment Service 37](#_Toc88238413)

[3.1.4.1 Entidades 37](#_Toc88238414)

[3.1.4.2. *Types* 38](#_Toc88238415)

[3.1.4.3. *Queries* 39](#_Toc88238416)

[3.1.4.4. *Inputs* 40](#_Toc88238417)

[3.1.4.5. *Mutations* 41](#_Toc88238418)

[3.1.5. API Gateway 42](#_Toc88238419)

[3.1.5.1. Requisições GraphQL 45](#_Toc88238420)

[3.1.6. Configuração Docker 47](#_Toc88238421)

[3.1.7. Eventos Kafka 50](#_Toc88238422)

[4. RESULTADOS 53](#_Toc88238423)

[4.1. Composição do sistema 53](#_Toc88238424)

[4.3. Acesso a dados 53](#_Toc88238425)

[4.2. Atuação do Apache Kafka 54](#_Toc88238426)

[4.2.1. Replicação dos dados 54](#_Toc88238427)

[5. CONSIDERAÇÕES FINAIS 57](#_Toc88238428)

[5.1. Contribuições 57](#_Toc88238429)

[5.2. Trabalho Futuros 58](#_Toc88238430)

[REFERÊNCIAS 59](#_Toc88238431)

# 1. INTRODUÇÃO

Com o surgimento e expansão da pandemia por mais de um ano, muitas lojas e comércios precisaram se modernizar e se adaptar para continuar atendendo a população, ocasionando em um grande crescimento na modalidade de *e-commerce*.

Esse crescimento aumenta também a necessidade de escalabilidade dos sistemas para comportar este grande aumento de acessos simultâneos. Além do aprimoramento de servidores e infraestrutura necessária para manter o sistema disponível para os clientes, o próprio sistema precisa estar preparado para eventuais quedas de alguns servidores, se mantendo disponível o máximo possível.

Os microsserviços permitem a componentização do *backend* do sistema e sua hospedagem em diversos servidores diferentes. Com isso, em uma eventual queda de servidor, é possível manter disponíveis outros elementos do conjunto para que os clientes possam continuar acessando o serviço e o *e-commerce* não tenha uma perda substancial de acessos e receita.

A proposta deste trabalho é explicar como funciona um sistema baseado em microsserviços e exemplificar por meio de um e-commerce.

## 1.1. Objetivos do Trabalho

O objetivo geral deste trabalho é o desenvolvimento de uma aplicação de e-commerce para exemplificar o funcionamento de um sistema baseado em microsserviços.

Para cumprir o objetivo geral, se fez necessário definir os seguintes objetivos específicos:

* Entender e aplicar a arquitetura SOA
* Combinar SOA com Arquitetura Orientada a Eventos (*Event-driven Architeture*)
* Organizar rotas para direcionamento dos microsserviços
* Configurar um serviço para comunicação entre os microsserviços
* Resolver problemas de múltiplos acessos aos Bancos de Dados e de informações compartilhadas entre microsserviços

## 1.2. Conteúdo do Trabalho

O presente trabalho está estruturado em cinco Capítulos, cujo conteúdo é sucintamente apresentado a seguir:

No Capítulo 2 é feita a fundamentação das tecnologias relacionadas ao objetivo geral deste trabalho.

O Capítulo 3 apresenta o desenvolvimento da solução, suas diferentes abordagens no mercado de trabalho e todas as dificuldades encontradas durante o desenvolvimento.

No Capítulo 4 são apresentados os resultados da pesquisa

O Capítulo 5 apresenta as considerações finais deste trabalho a partir da análise dos resultados obtidos e sugestões para passos futuros a partir deste estudo.

# 2. FUNDAMENTAÇÃO TÉCNICA

Este capítulo descreve as tecnologias utilizadas no desenvolvimento deste trabalho para auxiliar o leitor na compreensão da solução proposta.

## 2.1. Monólito e Microsserviços

Uma aplicação considerada monolítica (ou um monólito), é uma aplicação *server-side*, chamada aqui de *backend*, que é compilada em uma única unidade e é responsável por todos os processos. Neste caso, a aplicação disponibiliza a interface para interação com o cliente e recebe suas requisições, executa validações de regra de negócio e acessa dados no banco de dados em uma única instância, sem depender de outros módulos para executar tais tarefas (FOWLER, 2014).

Esta abordagem funciona bem em aplicações pequenas por ser de fácil desenvolvimento, testagem e manutenção. Entretanto, da mesma forma que a aplicação cresce, sua complexidade também aumenta, dificultando manutenções futuras e obrigando ao desenvolvedor compilar a aplicação toda ao corrigir algum erro ou adicionar alguma funcionalidade ao projeto total (FOWLER, 2014).

A arquitetura de microsserviços permite a componentização da aplicação via serviços que funcionam de forma independente e respondem requisições web realizadas pelos clientes. Esta componentização, além de possibilitar a separação do desenvolvimento de cada componente por uma equipe pequena focada em apenas uma parte do sistema, facilita os testes e a implementação de funcionalidades novas, pois permite a compilação apenas do componente modificado, mantendo os demais intactos e funcionando enquanto o serviço em questão é atualizado ou corrigido (FOWLER, 2014).

## 2.2. Virtualização

Para que seja possível explicar o que são containers e fazer o paralelo desta abordagem para a que utiliza máquinas virtuais, é preciso antes ambientar o conceito de virtualização.

Virtualização é uma combinação das engenharias de software e hardware que cria um ambiente virtual, permitindo que múltiplos sistemas operacionais funcionem na mesma plataforma física. É chamada de máquina hospedeira (*host machine*) a máquina física onde é instalado e executado o sistema de virtualização, e máquinas hóspedes (*guest machines*) são as máquinas virtuais ou containers criados nesta máquina hospedeira (KUMAR, 2015).

Utilizando esta abordagem, é possível maximizar os recursos físicos da máquina hospedeira ao dividi-la em diversas máquinas hóspedes, cada uma com seu respectivo sistema operacional (KUMAR, 2015).

## 2.3. Máquinas Virtuais e Docker

Máquinas virtuais permitem a criação de uma espécie hardware virtual, onde recursos da máquina hospedeira são utilizados para criar cópias virtuais em cada máquina hóspede. Desta forma, uma ferramenta de virtualização que utiliza máquinas virtuais permite o acesso ao hardware da máquina hospedeira apenas, fazendo com que seja necessária a instalação completa do sistema operacional desejado e consumindo boa parte dos recursos físicos da máquina hospedeira (MERKEL, 2014).

Já containers Docker efetivamente virtualizam um sistema operacional, levantando apenas os serviços necessários para o funcionamento desejado. Cada container não sabe da existência dos demais existentes no sistema operacional hospedeiro, pois as camadas de rede, processos e demais recursos são abstraídas e virtualizadas pelo Docker (MERKEL, 2014).

Simplificando a comparação entre as abordagens, containers Docker virtualizam em nível de Sistema Operacional, enquanto máquinas virtuais virtualizam em nível de hardware (MERKEL, 2014).

## 2.4. Arquitetura Orientada a Serviços (*Service-Oriented Architecture* - SOA)

SOA é um conceito de arquitetura que busca disponibilizar funcionalidades de um sistema como serviços. Neste conceito, cada serviço possui baixo acoplamento entre os demais que compõem o sistema e pode ser desenvolvido utilizando tecnologias e linguagens de programação distintas, precisando apenas utilizar a mesma técnica de comunicação entre os demais, para que todos possam se comunicar sem problemas (MARÉCHAUX, 2005).

Em SOA, os componentes podem ser considerados consumidores ou fornecedores de um serviço, mas um fornecedor pode também ser um consumidor ao realizar requisições para outro componente do sistema. A comunicação entre os componentes pode ser síncrona ou assíncrona e os consumidores podem encontrar dinamicamente os fornecedores disponíveis no sistema. Podem também existir componentes auxiliares que intermediam a comunicação entre os consumidores e os fornecedores, denominados ESB (*Enterprise Service Bus*) (BIANCO, 2007).

## 2.5. Event-driven Architecture

Arquitetura orientada a eventos ou *Event-driven architecture* é uma arquitetura de desenvolvimento de software onde eventos assíncronos são transmitidos entre componentes e serviços desacoplados (MARÉCHAUX, 2005).

Sua estrutura é composta por emissores de eventos e seus consumidores, onde os emissores não possuem conhecimento de nenhum outro elemento além do conteúdo do evento emitido por eles. Uma vez emitido o evento, o emissor não sabe o que ocorre com o sistema, nem quem o recebe (MICHELSON, 2011).

Eventos são quaisquer ocorrências ou mudanças de estado significativas no sistema e podem ser originadas internamente, como o carregamento de um programa no sistema, ou externamente, como a interação do usuário com a aplicação (REDHAT, 2021).

A vantagem desta arquitetura é que, por ser altamente desacoplada, sua comunicação é totalmente assíncrona e pode ocorrer para diversos consumidores com um único evento, promovendo um ambiente com alta escalabilidade (MICHELSON, 2011).

## 2.6. Comunicação entre Microsserviços

Os serviços que compõem uma aplicação baseada em microsserviços podem se comunicar por requisições HTTP síncronas, requisições HTTP assíncronas e por meio de serviços de mensageria. Entretanto, a escolha do método de comunicação entre os serviços deve levar em consideração a necessidade e situação, pois cada método funciona de uma forma distinta e nem sempre é o apropriado dependendo da necessidade (GONZAGA, 2019).

Requisições HTTP síncronas bloqueiam a aplicação até que haja uma resposta e cria um acoplamento entre os serviços, pois dependerão exclusivamente desse *endpoint* que faz a comunicação entre eles. Já as requisições assíncronas resolvem o problema da espera da resposta do serviço, porém adicionam uma complexidade aos consumidores dos serviços, pois passa a existir a necessidade de verificar o status da ação requisitada (RABELO, 2019).

Serviços de mensageria permitem a comunicação por meio de mensagens enviadas a um elemento intermediário, chamado de “*message broker*”, que realiza o balanceamento de carga das mensagens recebidas e as encaminha aos serviços por meio de canais ou tópicos específicos. Cada serviço observa um ou mais canais/tópicos, recebe a mensagem enviada pelo broker e a processa de acordo com sua regra de negócio (GONZAGA, 2019).

## 2.7. Armazenamento de Dados

O armazenamento de dados em sistemas baseados em microsserviços pode ser realizado de diversas formas, porém um dos princípios de microsserviços descritos por (FOWLER, 2014) é de que se deve evitar comunicações indiretas entre microsserviços em um banco de dados.

Desta forma, uma das abordagens mais utilizadas e mais fáceis de se implementar sistemas de microsserviços é a de um banco de dados para cada componente. Para que os dados cresçam em conjunto com os serviços, normalmente é utilizada uma base de dados não relacional, permitindo o armazenamento dos dados no mesmo formato que são enviados para os clientes da aplicação, utilizando um objeto JSON (BROWN, 2016).

Esta abordagem torna o banco de dados da aplicação independente, pois trabalha sem a influência de bancos de outros serviços. Além disso, torna mais seguro o acesso aos dados, pois nenhum microsserviço pode acessar o banco de dados de outro e eventualmente corrompê-lo (TAIBI, 2018).

Outro padrão para armazenamento de dados utilizado é o de cluster de bancos de dados. Este padrão separa um esquema para cada microsserviço que o utiliza, preservando a consistência dos dados de cada microsserviço e deixando os demais esquemas invisíveis para o microsserviço em questão. Esta abordagem traz vantagens de escalabilidade, pois dedica um hardware para os bancos de dados, porém traz como desvantagens a complexidade de implementação de cluster e aumenta o risco de falhas, pois apresenta um elemento distribuído no esquema de microsserviços existente (RICHARDSON, 2014).

Dentre as demais, a abordagem mais simples de se utilizar ao migrar um monolito para uma arquitetura de microsserviços é a de um banco de dados compartilhado entre os componentes. O padrão é semelhante ao cluster, mas neste caso todos os componentes acessam concorrentemente o mesmo banco de dados e não existe isolamento de dados entre os serviços. A vantagem desta abordagem é a simplicidade ao de migrar de um monolito, pois não será necessário modificar nenhum esquema do monolito original. Como desvantagem, este padrão aumenta o risco de falhas pois a concorrência de acesso ao banco pode causar atrasos ou mesmo erros no acesso de um serviço, enquanto outro o acessa (RICHARDSON, 2014).

## 2.8. REST

Um dos estilos arquiteturais para a construção de APIs no *backend* é o REST. Foi originado na tese de Doutorado de Roy Fielding, que também foi coautor do HTTP, um dos protocolos mais utilizados no mundo atualmente. Desta forma, é notável que os conceitos do REST são alinhados a boas práticas do HTTP como o uso adequado de seus métodos, URI’s, códigos de retorno e cabeçalhos (SAUDATE, 2013).

Uma API REST descreve uma relação de recursos e operações que podem ser acessadas através deles, utilizando qualquer cliente HTTP, como uma aplicação JavaScript executada no navegador da máquina cliente. Em nível de classes, um *controller* é a classe que armazena os recursos disponíveis (SURWASE, 2016).

A URI, que é o identificador universal de um recurso, é utilizada para representar e diferenciar recursos e APIs dentro de um mesmo servidor. O caminho base pode ser utilizado para isolar diferentes APIs no mesmo endereço do servidor ou mesmo diferentes versões da mesma API. Por exemplo, a primeira versão de uma API de usuários pode ser representada como “usuariosV1” e uma segunda versão como “usuariosV2” (SAUDATE, 2013; SURWASE, 2016).

Os recursos são representados por caminhos relativos ao caminho base. Cada recurso possui uma lista de operações que podem ser executadas pelo cliente. Cada operação possui um nome, um método HTTP (como POST, GET e PUT) e deve possuir um nome único em relação aos demais recursos na mesma API. A combinação de método HTTP e caminho relativo da API identifica o recurso e operação executada na requisição (SURWASE, 2016).

A Tabela 1 ilustra o esquema de recursos e operações de uma API REST.

Tabela 1 - Exemplo de recursos e operações - API REST

|  |  |  |  |
| --- | --- | --- | --- |
| Método HTTP | Recurso | Nome da Operação | Descrição |
| GET | /usuario | listaUsuarios | Lista todos os usuários cadastrados. |
| POST | /usuario | criaUsuario | Cria um usuário na aplicação. |
| PUT | /usuario/123 | atualizaUsuario | Atualiza o usuário com o id 123. |
| DELETE | /usuario/123 | excluiUsuario | Exclui o usuário com o id 123. |

Fonte: Autor (2021).

## 2.9. GraphQL

GraphQL é uma linguagem de consulta para APIs e um ambiente de execução para o recebimento de dados existentes através desta linguagem de consulta. É uma alternativa ao REST e se difere por retornar exatamente o que os clientes solicitam e nada mais (REDHAT, 2021; GRAPHQL, 2021).

O desenvolvimento de uma API em GraphQL requer três elementos específicos: esquemas (*schemas*), consultas (*queries/mutations*) e resolvedores (*resolvers*).

Os esquemas são a descrição dos dados que serão enviados como parâmetros nas consultas, chamados de Inputs, e os dados que serão retornados em uma consulta ou após uma modificação de dados, chamados de Types (REDHAT, 2021).

As consultas são podem ser para obter dados ou para modificá-los. Consultas de obtenção de dados são chamadas de *queries* e englobam quaisquer requisições que apenas retornam dados. Já consultas para modificação de dados são chamadas de *mutations* e são utilizadas para criação, modificação e exclusão de dados no sistema (LANDEIRO, 2019).

Resolvedores são funções atreladas as consultas e possuem o papel de buscar ou modificar os dados requeridos de acordo com a requisição do usuário. São responsáveis por indicar qual é a classe e método do projeto que será incumbida de fornecer os dados ou modificá-los (REDHAT, 2021).

## 2.10. Apache Kafka

O Apache Kafka é um sistema de mensageria e uma plataforma de transmissão de dados de alto desempenho que vem sendo amplamente utilizada para pré-processamento e transporte de quantidades massivas de dados entre aplicações. Sua implementação segue um modelo “produtor-consumidor” em que mensagens são geradas por “produtores” e enviadas a um “*broker*”, que categoriza e armazena as mensagens em “tópicos” e as direciona aos “consumidores” (CORREA, 2020).

Os servidores chamados “*brokers*” são servidores Kafka que recebem mensagens das aplicações chamadas “produtoras” e os armazena em disco. As mensagens recebidas são armazenadas em tópicos, que são a maneira de categorizar e classificar mensagens dentro do Kafka. Produtores são aplicações que escrevem mensagens e as enviam em tópicos aos *brokers*. Os brokers as armazenam e encaminham aos “consumidores”. Consumidores são aplicações que recebem as mensagens dos *brokers*, e para que isso seja possível, se inscrevem em determinados tópicos do *broker* e recebem apenas as mensagens dos tópicos inscritos (LAWLOR, 2018). A Figura 1 ilustra a comunicação entre produtor e consumidor com a ajuda do *cluster* Kafka.

Figura 1 - Comunicação entre produtor e consumidor por meio de tópicos.



Fonte: SEYMOUR (2021).

É importante ressaltar que os produtores não enviam mensagens para consumidores específicos, mas enviam para tópicos nos brokers sem o conhecimento dos consumidores inscritos em tal categoria. O mesmo ocorre no caminho inverso, onde os consumidores não possuem conhecimento dos produtores que enviaram a mensagem que está sendo recebida. Este padrão aumenta a escalabilidade do sistema, pois facilita a adição e remoção de produtores ou consumidores (SHARMA, 2018).

A implementação do Kafka consiste em um *cluster* de múltiplos *brokers*, que são numerados e armazenam dados atribuídos aos tópicos. Um *cluster* é a conexão entre duas ou mais máquinas que trabalham em conjunto para o processamento de alguma atividade com o propósito de melhorar o desempenho. Os tópicos podem ser divididos em partições, que são unidades de paralelização. Se um tópico for configurado para ter N partições, então N consumidores podem ler independentemente e paralelamente o mesmo tópico. Além disso, as partições dos tópicos podem ter um fator de replicação R configurado no Kafka, permitindo um número R de cópias destas partições a serem gerenciadas pelo cluster. Assim, caso R - 1 de máquinas do cluster falhem, a informação não será perdida (HESSE, 2021; LAWLOR, 2018).

Outro recurso do Kafka é o conceito de grupos de consumidores. Como o Kafka não possui o conceito de filas em sua implementação, não é possível que uma mensagem enviada a um tópico específico seja recebia por apenas um consumidor dentre os demais que se inscrevem neste determinado tópico. Para alcançar tal funcionalidade, é possível criar grupos de consumidores dentro do tópico desejado. Mensagens transmitidas a um grupo de consumidores são lidas apenas por um de seus membros, diferentemente de tópicos padrão do Kafka, que permitem a leitura das mensagens por todos que se inscrevem (LAWLOR, 2018).

# 3. DESENVOLVIMENTO

Neste capítulo é descrita a metodologia de desenvolvimento do sistema incluindo a arquitetura do sistema, os elementos, tecnologias específicas utilizadas e todo o processo de desenvolvimento.

## 3.1. Arquitetura do Sistema

A aplicação a ser desenvolvida como exemplo é um *e-commerce* e será fragmentada em microsserviços divididos de acordo com a necessidade, com a estratégia de aumentar a disponibilidade do sistema no caso de uma eventual falha em um dos servidores que hospeda os serviços.

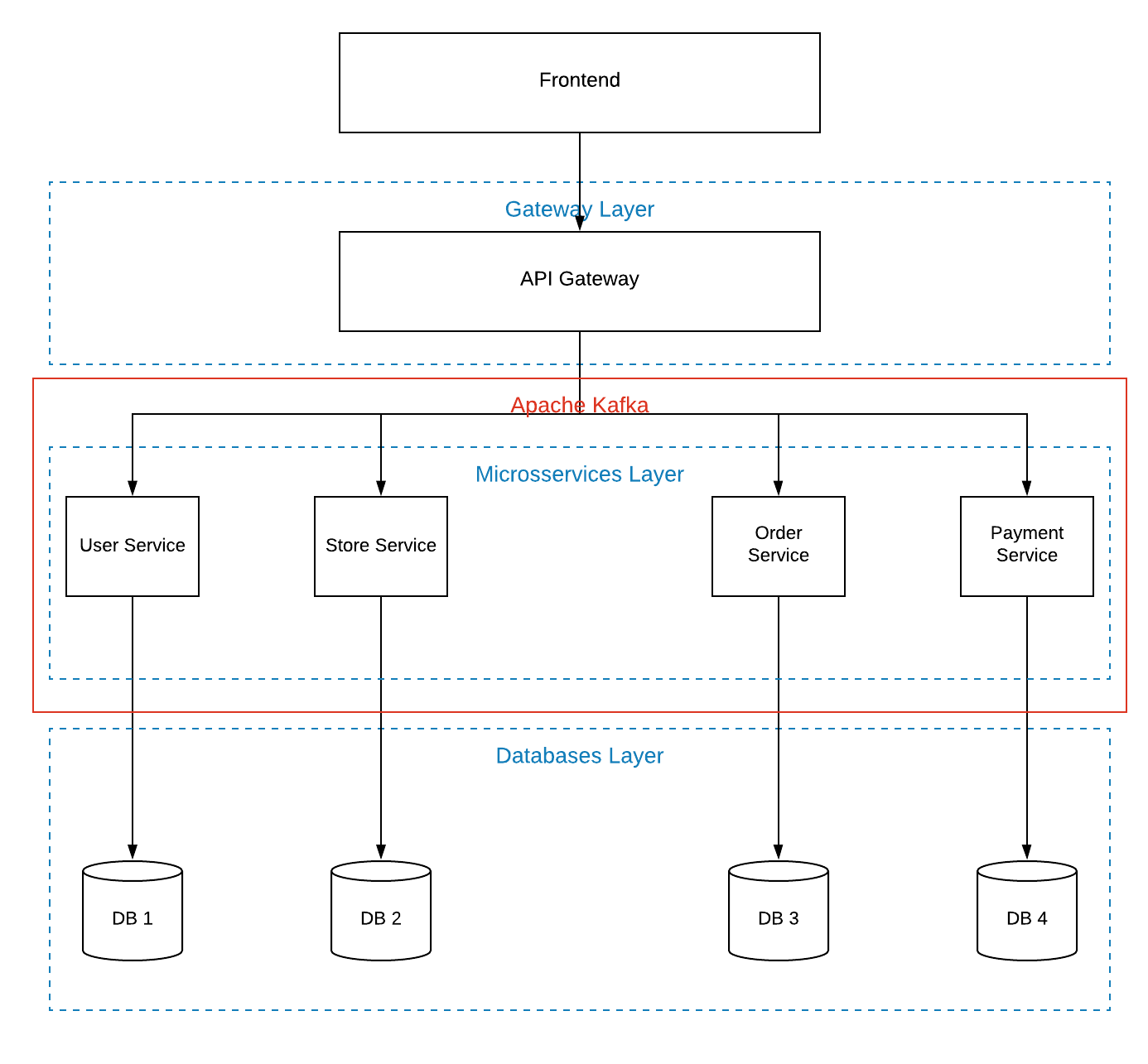
Os serviços planejados foram: *User, Store, Payment e Order*. Cada um deles será executado em um container Docker e possuirá um banco de dados dedicado.

Um *container* hospedará o *broker* do Kafka para enviar as mensagens enviadas pelos serviços em eventos específicos como criação de usuário ou criação de um carrinho de compras. Estes eventos do Kafka serão explicados com mais detalhes posteriormente.

Por fim, existirá um *container* Docker com uma aplicação que servirá como o foco da comunicação entre o *backend* e o *frontend*. Esta aplicação é chamada de “API Gateway” e será responsável pelas chamadas diretas aos microsserviços responsáveis pelas ações da loja.

A Figura 2 retrata a arquitetura da aplicação, ilustrando todos os serviços que a compõem e a separação de suas camadas.

Figura 2 – Arquitetura do sistema de Microsserviços.



Fonte: Autor (2021).

**3.1.1. *User Service***

Serviço que centraliza dados relacionados aos usuários do sistema. Nele estão armazenados e manipulados dados do usuário que efetuará a compra no e-commerce. Estes dados auxiliarão no processo de autenticação da conta, personalização de perfil e entrega dos produtos.

A API para comunicação com este serviço foi desenvolvida utilizando GraphQL e seus detalhes estão descritos com mais detalhes a seguir.

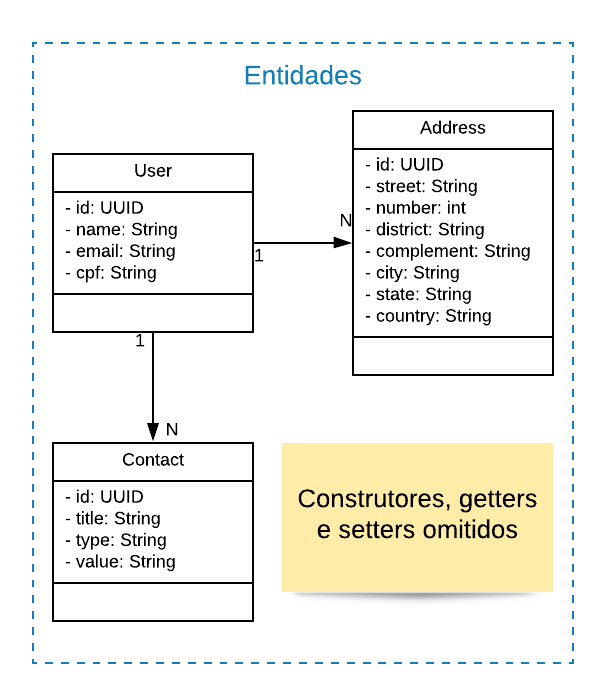
**3.1.1.1. Entidades**

Para o módulo que lida com informações dos usuários, foram criadas três entidades:

* *User*: expressa um usuário da aplicação e possui informações como nome, CPF, e-mail, uma lista de contatos e uma lista de endereços que podem ser cadastrados na aplicação para a realização das compras.
* *Address*: expressa um endereço para cobrança ou entregas para as compras na aplicação. Pode ser cadastrado mais de um endereço, para que haja mais flexibilidade do usuário para a escolha de entregas.
* *Contact*: Informações de contato com o usuário para auxiliar no processo de cobrança e entrega dos itens adquiridos na aplicação.

A Figura 3 retrata todas as entidades do serviço de usuário.

Figura 3 - Classes do serviço de usuário.

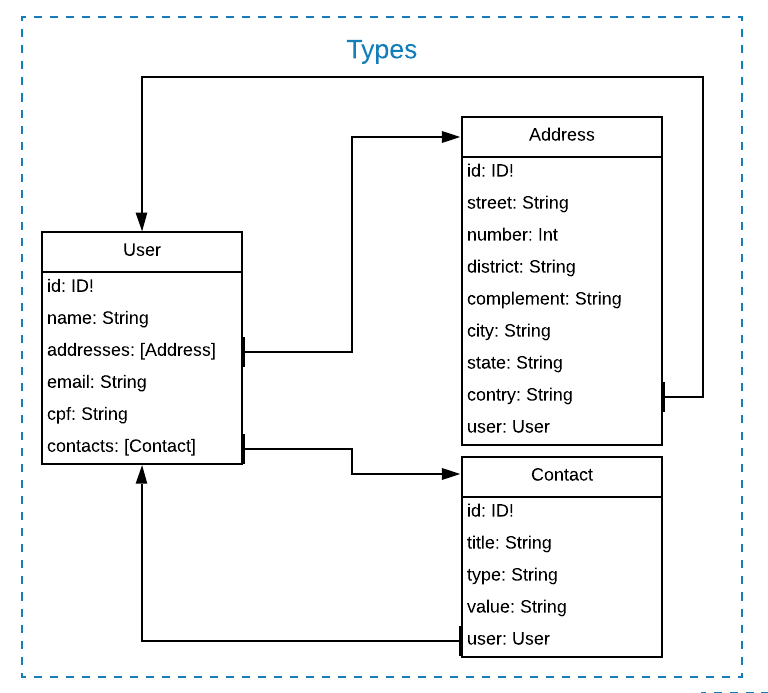


Fonte: Autor (2021).

**3.1.1.2. *Types***

*Types* são os tipos de dados utilizados para as respostas das requisições das *queries* e *mutations* do serviço de usuário. A Figura 4 retrata os tipos de dados do serviço de usuário.

Figura 4 - Tipos do serviço de usuário.

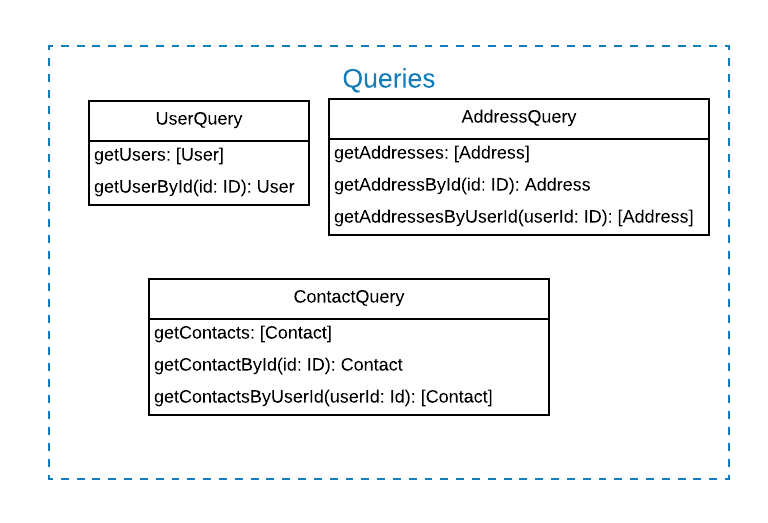


Fonte: Autor (2021).

**3.1.1.3. *Queries***

Trata-se das requisições de seleção de dados no banco da aplicação. A Figura 5 retrata as *queries* do serviço de usuário.

Figura 5 - *Queries* do serviço de usuário.

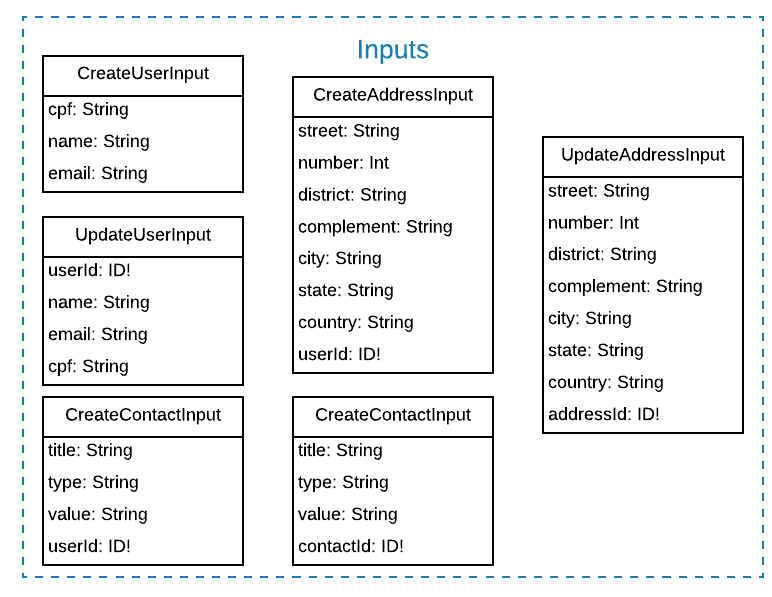


Fonte: Autor (2021).

**3.1.1.4. *Inputs***

*Inputs* são os modelos para inserção ou modificação de dados no banco da aplicação e utilizados como parâmetros de *mutations*. A Figura 6 retrata os *inputs* do serviço de usuário.

Figura 6 - *Inputs* do serviço de usuário.

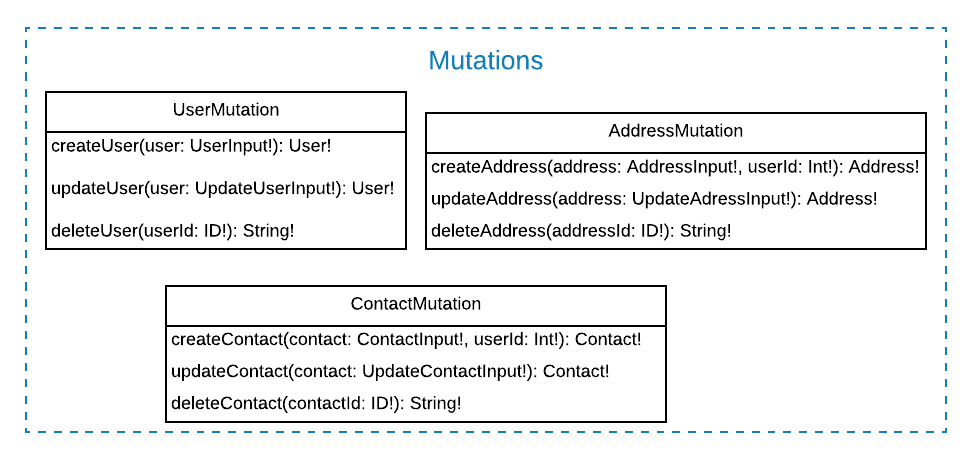


Fonte: Autor (2021).

**3.1.1.5. *Mutations***

*Mutations* são as requisições de criação ou modificação de dados no banco. E utilizam como parâmetro os *Inputs* previamente declarados. A Figura 7 retrata as *mutations* do serviço de usuário.

Figura 7 - *Mutations* do serviço de usuário



Fonte: Autor (2021).

### 3.1.2. *Store Service*

Este serviço é destinado aos dados dos produtos da loja. Nele estão armazenados os produtos, seus valores e seus atributos.

A API para comunicação com este serviço foi desenvolvida utilizando GraphQL e seus detalhes estão descritos com mais detalhes a seguir.

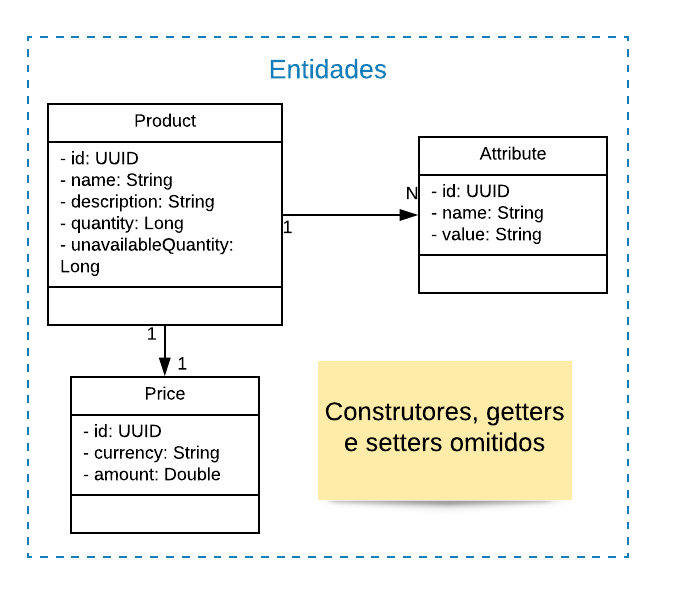
#### 3.1.2.1. Entidades

Para o módulo que lida com produtos do estoque da loja, foram criadas três entidades:

* *Product*: expressa um produto da loja e possui informações como nome, descrição e quantidade do item disponível em estoque. O atributo “*unavailableQuantity*” indica a quantidade de itens do produto que foram comprados, porém o pagamento ainda não foi efetuado.
* *Price*: expressa o valor do produto na loja e possuía apenas duas informações principais: moeda e quantia. O intuito de indicar a moeda é tornar a loja escalável a ponto de poder oferecer itens a serem vendidos por uma moeda diferente do real e permitir esta mudança de forma fácil, apenas criando um produto indicando seu valor e o código da moeda com três letras.
* *Attribute*: entidade genérica para indicar informações específicas de diferentes tipos de produtos como cores, dimensões, materiais e informações técnicas.

A Figura 8 retrata todas as entidades do serviço de estoque.

Figura 8 - Entidades do serviço de estoque.



Fonte: Autor (2021).

#### 3.1.2.2. *Types*

Referem-se aos tipos de dados utilizados para as respostas das requisições GraphQL. A Figura 9 retrata todas os tipos de dados utilizados pelo GraphQL no serviço de estoque.

Figura 9 - *Types* do serviço de estoque

Uma imagem contendo Texto

Descrição gerada automaticamente

Fonte: Autor (2021).

#### 3.1.2.3. *Queries*

*Queries* são requisições de seleção de dados no banco da aplicação. A Figura 10 retrata todas as *queries* de GraphQL no serviço de estoque.

Figura 10 - *Queries* do serviço de estoque.

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Fonte: Autor (2021).

#### 3.1.2.4. *Inputs*

*Inputs* são modelos para inserção ou modificação de dados no banco da aplicação e utilizados como parâmetros de *mutations*. A Figura 11 retrata os *inputs* utilizados no serviço de estoque.

Figura 11 - *Inputs* do serviço de estoque.

Texto

Descrição gerada automaticamente

Fonte: Autor (2021).

#### 3.1.2.5. *Mutations*

São requisições de criação ou modificação de dados no banco e utilizam como parâmetro os *Inputs* previamente declarados. A Figura 12 retrata as *mutations* do GraphQL utilizadas no serviço de estoque.

Figura 12 - *Mutations* do serviço de estoque.

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

Fonte: Autor (2021).

### 3.1.3. *Order Service*

Serviço reservado ao carrinho de compras dos clientes no sistema. Armazena os itens selecionados pelo usuário para uma compra, seus valores e o valor total, que é a soma do valor de todos os itens selecionados.

A API para comunicação com este serviço foi desenvolvida utilizando REST e seus detalhes estão descritos com mais detalhes a seguir.

#### 3.1.3.1. Entidades

Para o módulo que lida com o carrinho de compras da loja, foram criadas três entidades:

* *Cart*: expressa o carrinho de compras do usuário da loja. Recebe o id do cliente no serviço de usuário para associar as compras ao comprador. Além disso, possui uma lista de produtos e um preço final, que deve ser igual a soma do valor dos produtos adicionados.
* *OrderProduct*: expressa um produto adicionado no carrinho de compras e possui todos os campos do produto situado no serviço de estoque, com exceção do campo “*unavailableQuantity*”, que existe apenas para controle de estoque. O campo “*quantity*” indica a quantidade do produto de um mesmo tipo adicionada no carrinho de compras.
* *Price*: expressa o valor do produto no carrinho e o valor total do carrinho de compras.

A Figura 13 retrata todas as entidades do serviço de compras.

Figura 13 - Entidades do serviço de compras.

Texto

Descrição gerada automaticamente com confiança média

Fonte: Autor (2021).

#### 3.1.3.2. *Data Transfer Objects* (DTOs)

São objetos utilizados para transferência de informações entre o serviço de compras e os demais serviços do sistema. Fazem o mesmo papel dos *Types* e *Inputs* dos serviços que utilizam GraphQL, portanto podem ser utilizados tanto como parâmetros de métodos dos *controllers*, quanto para retorno das requisições HTTP realizadas no serviço. A Figura 14 retrata todos os DTOs utilizado no serviço de compras.

Figura 14 - DTOs do serviço de compras.

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

Fonte: Autor (2021).

#### 3.1.3.3. *Controllers*

*Controllers* são classes que disponibilizam a API REST para a comunicação entre o serviço de compras com os demais serviços do sistema. Neste caso, fazem o mesmo papel dos *resolvers* dos serviços desenvolvidos com GraphQL e são classificados por entidade do sistema. A Figura 15 retrata os *controllers* do serviço de compras.

Figura 15 - *Controllers* do serviço de compras.

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Fonte: Autor (2021).

### 3.1.4. *Payment Service*

Trata-se do serviço de pagamentos do sistema e faz integração com sistemas externos de pagamento e armazena dados do cliente, métodos de pagamento, contatos e pedidos dos clientes. Nesta aplicação, o serviço de pagamento simulará compras finalizadas com sucesso e canceladas.

A API para comunicação com este serviço foi desenvolvida utilizando GraphQL e seus detalhes estão descritos com mais detalhes a seguir.

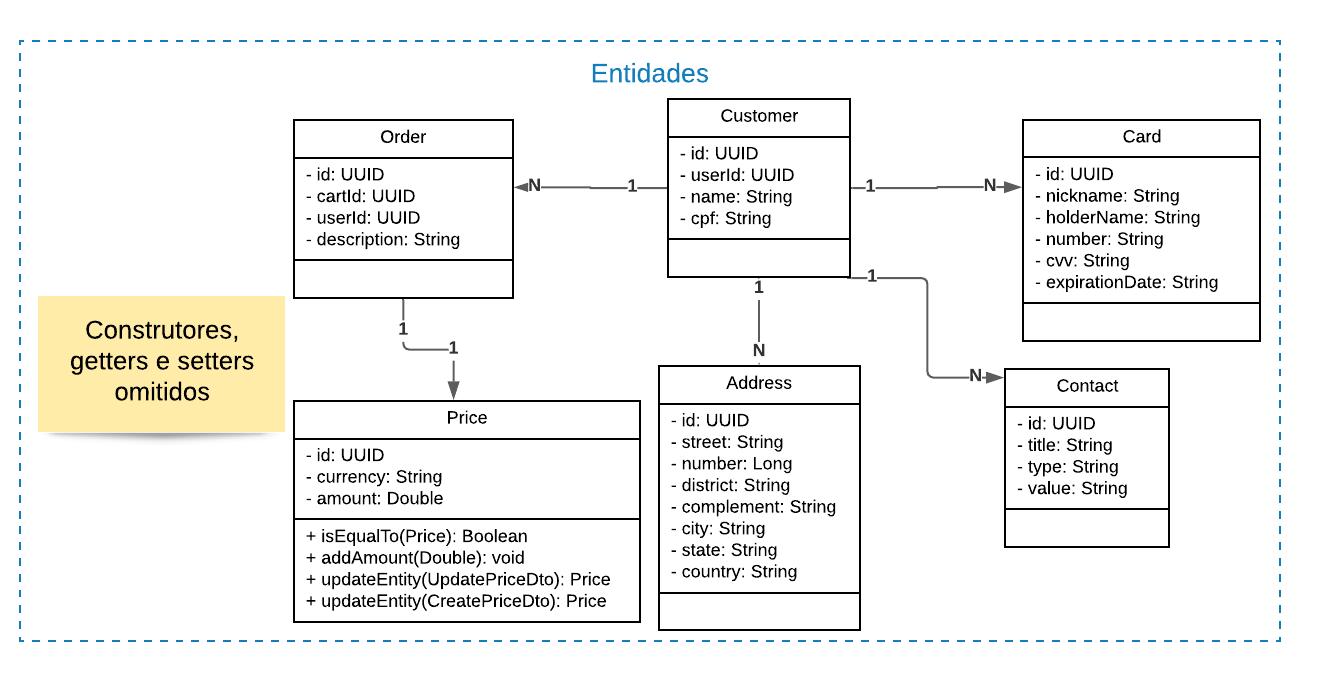
#### 3.1.4.1 Entidades

Para o módulo de pagamentos, foram criadas cinco entidades:

* *Customer*: expressa o cliente da loja e é uma cópia dos dados do usuário no serviço de usuários, para identificação no pagamento.
* *Order*: entidade que indica um pedido vindo do serviço de compras para pagamento.
* *Price*: expressa o valor total do pedido do usuário;
* *Address*: carrega informações de endereço para pagamentos do pedido na loja.
* *Card*: expressa cartões para pagamento das compras na loja.
* *Contact*: possui informações de contato para comunicação entre a loja e o cliente que efetuou a compra.

A Figura 16 retrata todas asentidadesdo serviço de pagamentos.

Figura 16 - Entidades do serviço de pagamentos.

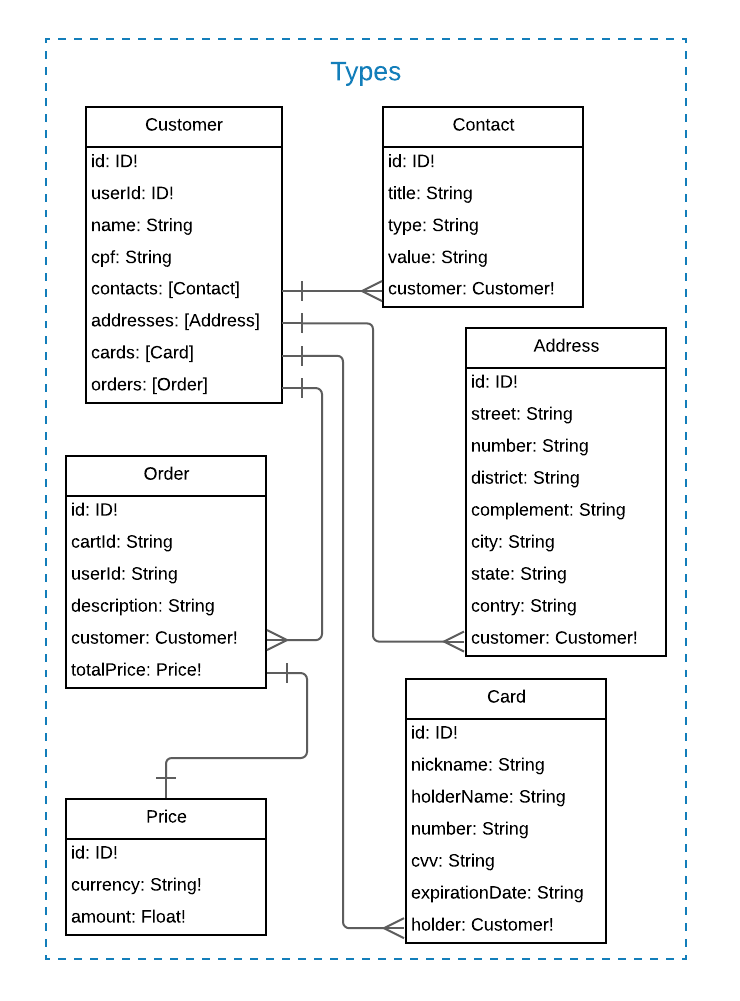


Fonte: Autor (2021).

#### 3.1.4.2. *Types*

Referem-se aos tipos de dados utilizados para as respostas das requisições das *queries* e *mutations*. A Figura 17 retrata todos os tipos de dados do serviço de pagamentos.

Figura 17 - Tipos de dados do serviço de pagamentos.

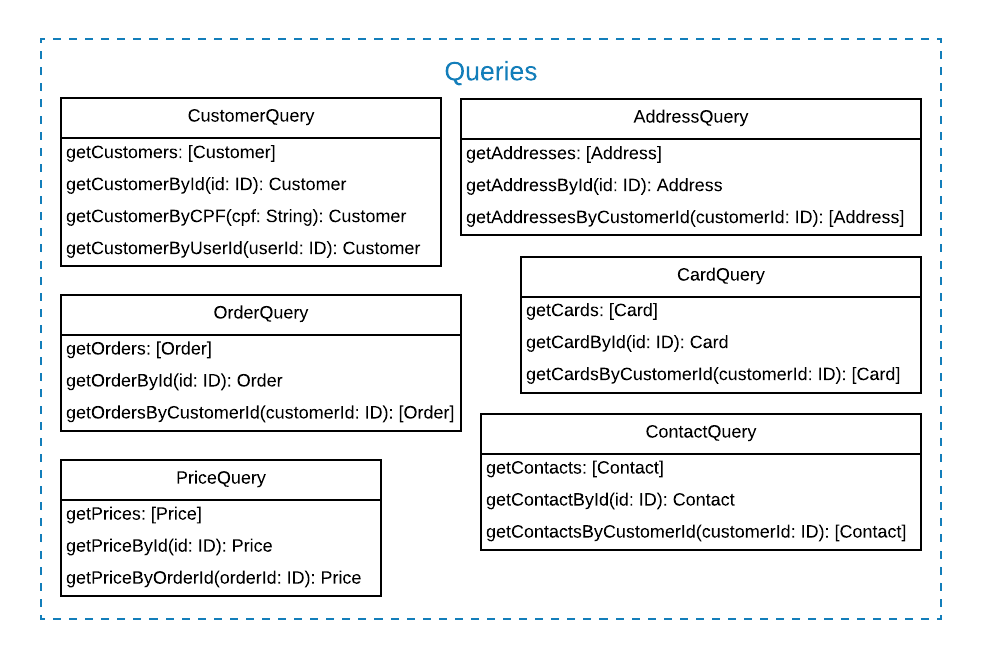


Fonte: Autor (2021).

#### 3.1.4.3. *Queries*

São requisições de seleção de dados no banco da aplicação. A Figura 18 retrata todas as *queries* de GraphQL do serviço de pagamentos.

Figura 18 - *Queries* do serviço de pagamentos.



Fonte: Autor (2021).

#### 3.1.4.4. *Inputs*

São modelos para inserção ou modificação de dados no banco da aplicação e utilizados como parâmetros de *mutations*. A Figura 19 retrata todos os modelos de *inputs* de GraphQL do serviço de pagamentos.

Figura 19 - *Inputs* do serviço de pagamentos.

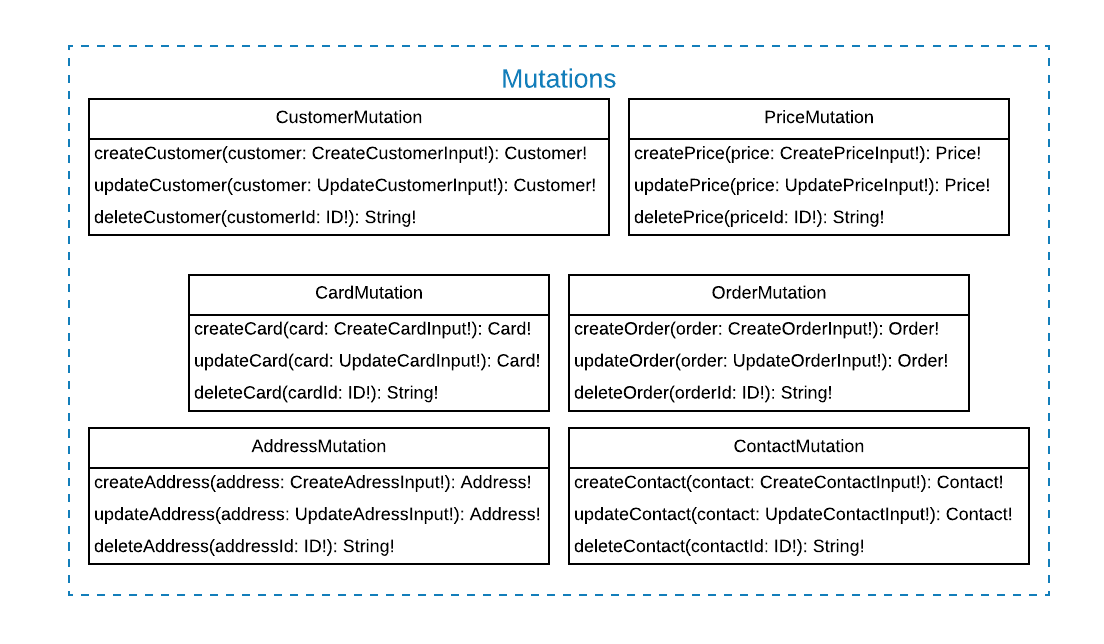


Fonte: Autor (2021).

#### 3.1.4.5. *Mutations*

São requisições de criação ou modificação de dados no banco e utilizam como parâmetro os Inputs previamente declarados. A Figura 20 retrata todas as *mutations* do serviço de pagamentos.

Figura 20 - *Mutations* do serviço de pagamentos.



Fonte: Autor (2021).

### 3.1.5. API *Gateway*

O API *Gateway* é uma aplicação intermediária que faz a interface entre os serviços do *backend* e as aplicações que o consomem. Todas as requisições para o sistema são direcionadas inicialmente a ela e redirecionadas para os microsserviços correspondentes. Portanto, nela podem estar todas as validações e serviços de autenticação e autorização para acesso a rotas e recursos.

Com base no aumento da demanda de acessos ao sistema, novas instâncias desta aplicação poderão ser adicionadas ao projeto para suprir o número de acessos.

O serviço de *gateway* foi desenvolvido com esta finalidade utilizando o *framework* Spring Boot. Todos os seus recursos estão disponíveis em REST e fazem interface com os sistemas disponíveis em GraphQL e com os serviços disponíveis em REST.

Foi criada uma classe de gerenciamento de requisições HTTP para centralizar a lógica de envio e retorno de dados. Como esta é uma classe única, ela deve ser genérica para aceitar todo tipo de dado tanto na requisição, quanto no retorno.

Para que seja possível, todos os métodos foram criados utilizando o recurso de tipagem genérica do Java, permitindo a tipagem de todos os retornos desejados no momento da requisição sem a necessidade de múltiplos métodos com a mesma finalidade. As Figuras 21, 22, 23 e 24 ilustram como a classe e seus métodos estão organizados.

Figura 21 - Declaração da classe gerenciadora de requisições e método de GET da classe

Texto

Descrição gerada automaticamente

Fonte: Autor (2021).

A classe gerenciadora é um serviço do Spring Boot e possui como atributo uma instância de um *WebClient*, que é uma interface de requisições HTTP do Spring Boot. Este atributo é responsável por realizar todas as requisições HTTP da classe gerenciadora.

Na classe gerenciadora foram criados métodos para requisições GET, POST, PUT e PATCH. O método de requisições GET do gerenciador recebe a URL a ser requisitada e uma classe genérica para a tipagem do retorno da requisição. O tipo da classe é inferido utilizando a notação ‘T’, e todas as operações que manipulam dados deste tipo são declaradas com esta notação, como é possível observar em “ResponseEntity<T>”.

Caso ocorra algum erro na requisição, é realizada uma verificação do código do erro retornado pela API requisitada e, caso seja igual a 404, uma *exception* customizada é enviada ao cliente que fez a requisição ao *Gateway*. Caso o código do erro seja diferente de 404, um erro genérico será enviado. Este tratamento de erros é realizado em todos os métodos de requisição.

Figura 22 - Método de requisição POST

Texto

Descrição gerada automaticamente

Fonte: Autor (2021).

O método de requisições POST do gerenciador recebe, além da URL e do tipo genérico T, uma instância de um tipo genérico B para o corpo da requisição, e retorna um ResponseEntity do tipo genérico T.

Figura 23 - Método de requisição PATCH

Texto

Descrição gerada automaticamente

Fonte: Autor (2021).

O método de requisições PATCH recebe os mesmos parâmetros do método POST e possui o mesmo padrão de requisição do método, apenas se diferenciando pelo método HTTP utilizado.

Figura 24 - Método de requisição DELETE

Texto

Descrição gerada automaticamente

Fonte: Autor (2021).

O método de exclusão tem a estrutura semelhante ao método de requisições GET da classe, portanto não envia um corpo de requisição. A diferença entre os dois métodos é apenas o método HTTP utilizado.

#### 3.1.5.1. Requisições GraphQL

As requisições para serviços GraphQL poderiam ser realizadas com a ajuda de bibliotecas e pacotes que encapsulam todo o processo e oferecem uma interface mais amigável para requisições. Entretanto, para que o desenvolvimento do API *Gateway* siga o mesmo padrão e centralize todas as requisições em um ponto, a mesma classe gerenciadora foi utilizada para tais requisições.

Como essencialmente todas as requisições GraphQL são um POST com um body que segue um padrão de formato, as chamadas do *gateway* aos serviços GraphQL são todas realizadas através do método chamado “doPOST”. A Figura 25 retrata a chamada de uma das requisições utilizando a classe gerenciadora.

Figura 25 - Requisição de criação de usuário

Texto

Descrição gerada automaticamente

Fonte: Autor (2021).

A estrutura do corpo da requisição não é exatamente um JSON, então foi preciso criar uma *string* seguindo o padrão do corpo das requisições GraphQL e montá-la de acordo com o tipo de requisição e retorno de cada recurso dos serviços que utilizam este padrão de comunicação.

Para organizar e manter a estrutura de retorno das requisições no padrão desejado, foi criada uma classe de *string* constantes, evitando assim reescrita de código e erro na grafia das palavras. A Figura 26 ilustra algumas constantes criadas na classe e utilizadas no mapeamento do retorno da API GraphQL na criação de usuário.

Figura 26 - Constantes para o mapeamento do retorno da API GraphQL

Texto

Descrição gerada automaticamente

Fonte: Autor (2021).

### 3.1.6. Configuração Docker

O Docker foi utilizado no desenvolvimento da aplicação para a execução das aplicações em um ambiente controlável e portátil, evitando assim a incompatibilidade de alguma dependência.

Cada microsserviço que compõe o sistema possui um *Dockerfile*, que é um arquivo de automatização de compilação do projeto. A figura 27 retrata a estrutura do *Dockerfile* do serviço de API *Gateway*, que segue a mesma estrutura dos demais arquivos Docker.

Figura 27 - Dockerfile da API Gateway

Texto

Descrição gerada automaticamente

Fonte: Autor (2021).

A partir de um contêiner Docker com Java 8 e Maven pré-instalados, todas as dependências do projeto são instaladas e um pacote do projeto é criado com a extensão “.jar” para a execução posterior. Após a criação do pacote, os arquivos contidos nele são extraídos com o comando “jar -xf” e executados com o comando contido nos parâmetros do “ENTRYPOINT” do *Dockerfile.*

Para a execução de todos os contêineres necessários na aplicação e para controle de variáveis e credenciais dos serviços, foi criado um arquivo “docker-compose.yaml”. Este arquivo mapeia as conexões entre serviços e seus bancos de dados; a ordem de inicialização dos contêineres para evitar conflito; as portas de rede expostas para a máquina hospedeira do contêiner, variáveis de ambiente com credenciais de banco de dados e configurações do sistema que sobrescrevem as contidas no arquivo “application.yaml” do Spring Boot. As Figuras 28, 29 e 30 retratam alguns exemplos destas configurações contidas no arquivo “docker-compose.yaml”.

Figura 28 – Configurações para os serviços do Kafka e Zookeeper

Texto

Descrição gerada automaticamente

Fonte: Autor (2021).

O serviço de Zookeeper é responsável pela sincronização de configurações e dados entre os diferentes clusters Kafka existentes em um sistema. Desta forma, foi necessário criar um contêiner com o Zookeeper para o controle do cluster configurado para o exemplo.

Utilizando o contêiner do Zookeeper como dependência de execução, o serviço do Kafka é descrito logo abaixo com fator de replicação de apenas 1 para que seja mais simples a exemplificação do sistema.

Figura 29 - Configurações para o serviço de usuário e seu banco de dados

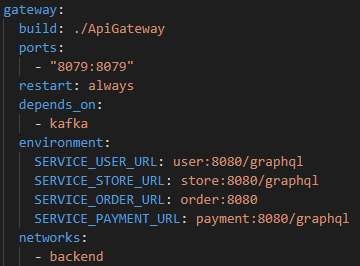
Texto

Descrição gerada automaticamente

Fonte: Autor (2021).

A imagem do serviço de usuário depende do seu banco de dados e do apache kafka e define credenciais, endereço, e dialeto para comunicação com o banco de dados. Todas estas configurações sobrescrevem as já existentes no arquivo de configuração da aplicação contido no arquivo “application.yaml”.

Figura 30 - Configurações do serviço de API *Gateway*



Fonte: Autor (2021).

O serviço de API *Gateway* depende apenas do Kafka e mapeia o endereço de cada microsserviço que compõem o sistema do exemplo. Em cada URL base, no lugar do endereço foi utilizado o nome do serviço declarado dentro da rede do Docker, fazendo com que “http://localhost:8080” seja declarado como “user:8080”.

### 3.1.7. Eventos Kafka

Os eventos do sistema de mensageria foram utilizados para a propagação de dados em determinadas ações do usuário no sistema.

Alguns serviços do sistema podem enviar dados para o Kafka, os categorizando com um dos tópicos abaixo e outros podem se inscrever nos tópicos para receberem os dados e os armazenarem de acordo com a necessidade.

A seguir são descritos os tópicos trafegados na aplicação e suas respectivas requisições no sistema:

* **USER\_CREATED:** Tópico enviado pelo serviço de usuário após um novo cadastro. O serviço de pagamentos se inscreve neste tópico e consome os dados do usuário cadastrado, criando um “*customer*”, que identifica o autor das compras a serem pagas no sistema.
* **USER\_DELETED:** Tópico enviado pelo serviço de usuário após a exclusão de um usuário do sistema. O serviço de pagamentos se inscreve neste tópico e consome os dados do usuário excluído, removendo também o “*customer*” vinculado ao usuário em questão.
* **USER\_UPDATED:** Tópico enviado pelo serviço de usuário após a atualização de um usuário do sistema. O serviço de pagamentos se inscreve neste tópico e consome os dados do usuário atualizado, modificando também o “*customer*” vinculado ao usuário em questão.
* **ADDRESS\_CREATED:** Enviado pelo serviço de usuários após a criação de um endereço do usuário, este tópico é consumido pelo serviço de pagamentos para o cadastro de um endereço para entrega e cobrança, vinculando ao cadastro prévio do cliente no serviço.
* **ADDRESS\_DELETED:** Enviado pelo serviço de usuários após a exclusão de um endereço do usuário. O tópico é consumido pelo serviço de pagamentos para a exclusão do endereço em questão.
* **ADDRESS\_UPDATED:** Enviado pelo serviço de usuários após a atualização de um endereço do usuário, este tópico é consumido pelo serviço de pagamentos e atualiza o endereço em sua base de dados.
* **CONTACT\_CREATED:** Tópico enviado pelo serviço de usuário na criação de um novo contato do cliente. É consumido pelo serviço de pagamentos, que cadastra os mesmos dados em sua base de dados.
* **CONTACT\_DELETED:** Tópico enviado pelo serviço de usuário na exclusão de um contato do cliente. É consumido pelo serviço de pagamentos, que exclui o contato relacionado de sua base de dados.
* **CONTACT\_UPDATED:** Tópico enviado pelo serviço de usuário na atualização de um contato do cliente. É consumido pelo serviço de pagamentos, que atualiza os mesmos dados em sua base de dados.
* **ORDER\_STARTED:** Tópico enviado pelo serviço de compras ao adicionar um item no carrinho de compras. É consumido pelos serviços de estoque e pagamentos. O serviço de estoque aumenta o valor do atributo “*unavailableQuantity*”, previamente indisponibilizando a quantidade do item selecionado até que a compra seja finalizada e paga sem nenhum problema. Já o serviço de pagamentos faz um pré-cadastro do carrinho de compras criado, para que seja identificada a quantia a ser paga no fim da compra.
* **ORDER\_PAID:** Enviado pelo serviço de pagamentos, este tópico indica que o pedido foi pago com sucesso. O serviço de compras consome este tópico para atualizar os dados relacionados. Após consumo, o carrinho relacionado ao pedido pago é atualizado e armazenado para futuras verificações.
* **ORDER CANCELED:** Tópico enviado pelo serviço de pagamento ao cancelamento de um pedido pelo usuário. O serviço de compras o consome e atualiza suas informações. Após consumo, o carrinho relacionado ao pedido é finalizado.
* **CART\_PAID:** Enviado pelo serviço de compras após consumir o tópico “ORDER\_PAID”, este tópico indica que o carrinho de compras foi pago com sucesso. O serviço de estoque consome este tópico para atualizar os dados relacionados. Após consumo, o atributo “*unavailableQuantity*” do produto é zerado, indisponibilizando definitivamente a quantidade do produto comprado.
* **CART\_CANCELED:** Tópico enviado pelo serviço de compras após consumir o tópico “ORDER\_CANCELED”. O serviço de estoque o consome e atualiza suas informações. Após consumo, o atributo “*unavailableQuantity*” do produto é zerado e a quantidade do produto é somada novamente.

# 4. RESULTADOS

Neste capítulo serão apresentados os resultados do desenvolvimento do sistema de microsserviços como exemplos de replicação de dados através do Apache Kafka, o papel do Kafka na aplicação desenvolvida e a solução para o acesso múltiplo ao banco de dados.

## 4.1. Composição do sistema

O sistema desenvolvido aplica conceitos da arquitetura SOA ao separar funções distintas da aplicação final em diferentes serviços desacoplados, e combina esta arquitetura com a arquitetura orientada a eventos quando utiliza um serviço de mensageria para a replicação dos dados entre os serviços e está dividido em três camadas: serviços, *gateway* e mensageria.

A camada de serviços acomoda os microsserviços *User, Store, Payment e Order* e seus respectivos bancos de dados. Cada serviço está em um contêiner Docker e se conecta com seu respectivo banco de dados através da rede interna do Docker.

O *Gateway* é uma camada que intermedia as requisições aos serviços que compõem o sistema de *e-commerce*. Ele recebe as requisições de outras aplicações e redireciona para os serviços desejados.

O serviço de mensageria recebe dados dos serviços produtores e encaminha aos respectivos tópicos, onde os serviços consumidores que escutam estes tópicos armazenam uma cópia destes dados em seus respectivos bancos de dados.

## 4.3. Acesso a dados

Um dos problemas que ocorrem em aplicações que utilizam arquiteturas de microsserviços é o acesso ao banco de dados. A concorrência no acesso decorrente da utilização de um único banco pode causar inconsistências no armazenamento e recuperação dos dados contidos neste banco.

Para que os serviços permaneçam desacoplados e independentes, além de solucionar o problema de concorrência de acesso, cada serviço possui seu próprio banco de dados dedicado e situado em um contêiner Docker. Desta forma, todas as informações que precisam ser compartilhadas entre dois ou mais serviços são enviadas a um tópico do Kafka para replicação posterior.

## 4.2. Atuação do Apache Kafka

Situado na camada de mensageria, o Apache Kafka foi a ferramenta utilizada para comunicação assíncrona entre os microsserviços. Este serviço recebe certas mensagens, as categoriza em tópicos e redireciona para replicação de dados entre os microsserviços do sistema.

## 4.2.1. Replicação dos dados

A replicação de dados entre os serviços ocorre se faz necessária por conta do método de armazenamento utilizado na arquitetura de microsserviços. Como cada serviço possui seu banco de dados dedicado, caso haja necessidade de compartilhamento de dados entre dois ou mais serviços, se faz necessária a replicação destes dados de alguma forma.

O método de replicação dos dados utilizado neste sistema foi intermediado pelo Apache Kafka. Em pontos específicos do sistema, após o armazenamento dos dados no banco do serviço, uma mensagem é enviada a um tópico do Kafka para que os serviços que o consomem possam armazenar uma cópia dos dados para uso futuro.

As Figuras 31, 32, 33, 34, 35 e 36 ilustram um exemplo da replicação dos dados em ação no sistema desenvolvido.

Figura 31 - Requisição para criação de usuário.



Fonte: Autor (2021).

Primeiro é criada uma requisição GraphQL para o microsserviço de usuários a criação de um novo usuário no sistema.

Figura 32 - Resposta da requisição de criação de usuário.

Texto

Descrição gerada automaticamente

Fonte: Autor (2021).

Após a inserção de dados, o serviço retorna uma confirmação com os dados enviados, e o ID gerado automaticamente pela aplicação antes do envio ao banco de dados.

Figura 33 - Envio de mensagem Kafka ao tópico *USER\_CREATED* após criação de usuário.



Fonte: Autor (2021).

Após a confirmação, um evento do Apache Kafka é enviado pelo serviço de usuário ao broker com o rótulo “USER\_CREATED” e com os principais dados do usuário criado como corpo da mensagem.

Figura 34 - Consumo de mensagem do tópico *USER\_CREATED.*



Fonte: Autor (2021).

A mensagem enviada ao broker é consumida por todos os serviços que se inscrevem para consumir o tópico específico. Neste caso, o microsserviço de pagamentos recebe os dados enviado pelo serviço de usuários e usa estes dados para realizar um pré-cadastro de um customer, que armazena os dados de um usuário cadastrado para pagamento posterior na plataforma.

Figura 35 - Dados inseridos no banco do serviço de usuários.

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Fonte: Autor (2021).

Utilizando uma ferramenta de acesso ao banco de dados, é possível verificar que o usuário foi cadastrado na base de dados do serviço que recebeu a requisição inicialmente, e foi replicado para o serviço de pagamentos, conforme ilustra a Figura 35.

Figura 36 - Dados inseridos no banco do serviço de pagamentos

Interface gráfica do usuário, Texto, Aplicativo, chat ou mensagem de texto

Descrição gerada automaticamente

Fonte: Autor (2021).

Nos dados de um customer temos o cpf e o nome cadastrados anteriormente, além do campo “id\_user”, que é o id do usuário cadastrado no serviço de usuários e replicado para o serviço de pagamentos. Por fim, temos o campo “id\_customer”, que é gerado automaticamente pelo sistema.

# 5. CONSIDERAÇÕES FINAIS

Este capítulo apresenta as contribuições e conclusões deste trabalho, assim como os trabalhos futuros.

## 5.1. Contribuições

As contribuições apresentadas neste trabalho são:

* Aplicação combinando SOA e arquitetura orientada a eventos;
* Rotas de acesso externo a aplicação utilizando um API *Gateway*;
* Solução para problema de concorrência de acessos ao servidor único de banco de dados e
* Comunicação entre serviços com Apache Kafka.

O momento pandêmico vivenciado atualmente no mundo fez com que os serviços migrassem temporariamente ou definitivamente suas atividades para serviços na internet, aumentando o acesso destes serviços e criando a necessidade de recursos para escalabilidade e contenção de possíveis indisponibilidades.

Uma proposta de contenção para este aumento de acessos foi a utilização de microsserviços, pois permitiram uma escalabilidade mais ágil do sistema, além de proporcionar facilidade na solução de uma possível indisponibilidade do servidor que hospeda a aplicação, pois cada serviço do sistema foi idealmente hospedado em uma máquina diferente.

Neste trabalho foi apresentada uma aplicação baseada em microsserviços exemplificando um sistema de *e-commerce* para ilustrar os conceitos contidos em um sistema com tal arquitetura e algumas ferramentas utilizadas no desenvolvimento deste sistema.

Um dos objetivos deste trabalho foi combinar o conceito de SOA com a arquitetura orientada a eventos. Um sistema que implementa SOA separa suas funcionalidades em diferentes serviços que são desacoplados entre si e podem ser desenvolvidos em diferentes linguagens e tecnologias. Já a arquitetura orientada a eventos faz a comunicação entre seus serviços desacoplados através de um sistema de mensageria assíncrona.

O projeto desenvolvido neste trabalho combinou as duas arquiteturas pois, além de separar funcionalidades em serviços diferentes, fez a comunicação entre os serviços através do Apache Kafka, que é uma ferramenta de mensageria.

Em relação ao problema de concorrência no acesso ao banco de dados da aplicação, foram criados serviços de bancos de dados para cada microsserviço com um contêiner dedicado, isolando os dados acessados por cada serviço. Os eventos Kafka permitiram a replicação dos dados nos diferentes bancos.

Quanto ao API *gateway*, a vantagem de utilizá-lo foi fornecer um caminho central para requisições de aplicações móveis ou *web* que consumirão este sistema. Além disso, este serviço central pode se encarregar de proteção de acesso a rotas do sistema, evitando acesso indevido de certos recursos.

Durante o desenvolvimento, ocorreram problemas de performance devido à falta de memória RAM necessária para executar os 11 contêineres Docker que compõem o sistema. Foi necessário aumentar de 8GB para 16GB para que seja possível executar os contêineres e as demais aplicações para teste das APIs em paralelo.

Por fim, o sistema atendeu ao objetivo proposto que consistiu em exemplificar um sistema de microsserviços e manter atualizados os dados armazenados em cada banco de dados de acordo com a necessidade. Entretanto, para que haja um maior controle das instâncias em execução, será necessária a adição de um orquestrador de contêineres Docker no sistema, permitindo um gerenciamento da rede e estado atual dos serviços de forma ágil e robusta.

## 5.2. Trabalho Futuros

Este trabalho não encerra as contribuições no tema de um E-commerce com Microsserviços, mas abre oportunidade para os seguintes trabalhos futuros:

* Autenticação e autorização de rotas na camada do gateway e
* Orquestração de contêineres Docker.

# REFERÊNCIAS

BIANCO, P.; KOTERMANSKI, R.; MERSON, P. **Evaluating a Service-Oriented Architecture**. Carnegie Mellon University: Software Architecture Technology Initiative, 2007. 91p. (CMU/SEI-2007-TR-015).

BROWN, K. WOOLF, B. **Implementation Patterns for Microservices Architectures**. In: HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 22, 2016, Monticello, Illinois. 2016. 35p.

CORRÊA, I. C.; BARCELOS, P. P. Otimização do Mecanismo de Confirmação de Entrega de Mensagens no Apache Kafka. In: ESCOLA REGIONAL DE ALTO DESEMPENHO DA REGIÃO SUL (ERAD-RS), 20. , 2020, Santa Maria. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2020 . p. 145-146.

FOWLER, M. LEWIS, J. **Microservices: A Definition of a new Architectural Term**. 2014. Disponível em: http://martinfowler.com/articles/microservices.html. Acesso: 27/04/2021.

GONZAGA, R. **Comunicação entre microsserviços: Async**. 2019. Disponível em: https://medium.com/@rafaelgss/comunicacao-entre-microservices-async-ed3e5897ba6. Acesso: 25/05/2021.

GRAPHQL. **A Query Language for your API**. Disponível em: https://graphql.org/. Acesso em: 01//06/2021

HESSE, G.; MATTHIES, C.; UFLACKER, M. **How Fast Can We Insert? An Empirical Performance Evaluation of Apache Kafka**. In: IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS), 2020, Hong Kong, 2021. 8f.

KUMAR, R.; CHARU S. **An Importance of Using Virtualization Technology in Cloud Computing**. Global Journal of Computers & Technology. Vol. 1, n. 2, p. 56-60, 2015.

LANDEIRO, M. **Analysis of GraphQL performance: a case study**. 2019. 134 f. Tese (Mestrado em Engenharia Informática) - Instituto Superior de Engenharia do Porto, Porto, 2019.

LAWLOR, B. et. al. Field of genes: using Apache Kafka as a bioinformatic data repository. **GigaScience**, Vol 7, n. 4, p. 9-8, 2018.

MARÉCHAUX, J. Combining Service-Oriented Architecture and Event-Driven

Architecture using an Enterprise Service Bus. **IBM Developer works**, Vol. 1, p. 1269-1275, 2006.

MERKEL, D. Docker: Lightweight Linux Containers for Consistent Development and

Deployment. **Linux journal**. Vol. 2014, N. 239, 2 p., 2014.

MICHELSON, B. M. **Event-Driven Architecture Overview**. Boston, MA: Elemental Links. 2011. 12 p.

RABELO, E. **Micro Serviços: 3 métodos de comunicação entre serviços**, 2019. Disponível em: https://oieduardorabelo.medium.com/micro-servi%C3%A7os-3-m%C3%A9todos-de-comunica%C3%A7%C3%A3o-entre-servi%C3%A7os-5bc49021c991. Acesso em 25/05/2021.

REDHAT. “**O que é arquitetura orientada a eventos?”**. Disponível em: https://www.redhat.com/pt-br/topics/integration/what-is-event-driven-architecture. Acesso em: 18/05/2021.

REDHAT. “**GraphQL - O que é e para que serve?”**. Disponível em: https://www.redhat.com/pt-br/topics/api/what-is-graphql. Acesso em: 01/06/2021.

RICHARDSON, C. **Microservice Architecture.** 2014. Disponível em: http://microservices.io. Acesso em: 01/06/2021.

SAUDATE, A**. REST Construa APIs Inteligentes de Maneira Simples**. São Paulo: Casa do Código, 2013.

SEYMOUR, M. **Mastering Kafka Streams and ksqlDB: Building Real-Time Data Systems by Example**. Sebastopol, CA: Oreilly, 2021.

SHARMA, A.; GOMEZ, R. Apache Kafka: The truth is the log. INFO-H-415: Advanced Databases, Université Libre de Bruxelles - École Polytechniquee, 2018.

SURWASE V. REST API Modeling Languages - A Developer’s

Perspective. **IJSTE - International Journal of Science Technology & Engineering.** Vol 2, Issue 10, 2016.

TAIBI, D. LENARDUZZI, V. PAHL, C. **Architectural Patterns for Microservices: A Systematic Mapping Study**. In Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018), pages 221-232. ISBN: 978-989-758-295-0.