CENTRO UNIVERSITÁRIO UNICARIOCA

ALEX GABRIEL MENEZES TORRES

UTILIZAÇÃO DO PADRÃO API GATEWAY COM MICROSSERVIÇOS

RIO DE JANEIRO

2021

ALEX GABRIEL MENEZES TORRES

UTILIZAÇÃO DO PADRÃO API GATEWAY COM MICROSSERVIÇOS

Trabalho de Conclusão de Curso submetido ao Centro Universitário Carioca, como pré-requisito para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Manuel Martins Filho

RIO DE JANEIRO

2021

T 688u Torres, Alex Gabriel Menezes.
Utilização do padrão API Gateway com microsserviços/ Alex Gabriel Menezes Torres. Rio de Janeiro, 2021.
36 f.

Orientador: Manuel Martins.
Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) – Centro Universitário Unicarioca. Rio de Janeiro, 2021.

1. API Gateway. 2. Microsserviço. I. Martins, Manuel. II. Título.

CDD 004

ALEX GABRIEL MENEZES TORRES

UTILIZAÇÃO DO PADRÃO API GATEWAY COM MICROSSERVIÇOS

Trabalho de Conclusão de Curso submetido ao Centro Universitário Carioca, como pré-requisito para a obtenção do grau de Bacharel em Ciência da Computação

Rio de Janeiro, 20 de novembro de 2021.

BANCA EXAMINADORA

André Luiz Avelino Sobral – MSc.

Coordenador do curso

_____

Almir Fernandes dos Santos – MSc.

Convidado

_____

Manuel Martins Filho – DSc.

Orientador

_____

AGRADECIMENTOS

Meu agradecimento nunca vai ser o suficiente por tudo que a minha avó, dona Alice, batalhou por mim para que eu pudesse alcançar meus objetivos, pois tenho certeza que sem ela em minha vida eu jamais teria chegado aonde cheguei.

Esta conquista também não seria possível sem meus tios que sempre estiverem por mim em todos os momentos e me incentivaram a sempre querer mais, ter ambição para o mundo e a vontade de vencer.

Também gostaria de agradecer minha parceira, minha melhor amiga e o amor da minha vida, Andreza Rodrigues, por todo apoio e suporte durante minha graduação, por acreditar em mim e me impulsionar até quando mesmo estivesse extremamente desacreditado.

Alex Gabriel Menezes Torres

"Se custar a minha paz, já custou caro demais" (Black Alien)

RESUMO

Para quem está inserido no mundo da computação, mais especificamente no nó do desenvolvimento de software, nota-se a demanda crescente por sistemas escaláveis, mais robustos e com a utilização de padrões e tecnologias mais recentes, buscando o equilíbrio perfeito entre a fácil manutenção, a baixa complexidade e a performance dos serviços. Com isto em mente, não podemos deixar de mencionar a preferência pela utilização da arquitetura de microsserviços, que por sua vez atende todos os pontos aqui mencionados, podendo ser utilizado em conjunto com API (application programming interface) que fornece fácil acesso e de forma abstraída a estes serviços. O objetivo deste trabalho é relacionar a utilização do serviço de API gateway com a arquitetura de microsserviço, definindo seu papel, suas características, vantagem e a aplicabilidade.

Palavras-chave: API Gateway; Microsserviço; Padrão de Projeto;

ABSTRACT

For those in the computing world, more specifically in the software development node, there is a growing demand for scalable, more robust systems that use the latest standards and technologies, seeking the perfect balance between easy maintenance, the low complexity and also the performance of the services. With this in mind, we cannot fail to mention the preference for using the microservices architecture, which in turn meets all the points mentioned here, and can be used in conjunction with API (application programming interface) that provides easy access and in an abstract way to these services. The objective of this work is to relate the use of the API gateway service with the microservice architecture, defining its role, characteristics, advantage and applicability.

Keywords: API Gateway; microservice; Design Pattern;

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
B2B	Business-to-business
B2C	Business-to-consumer
CLI	Command Line Interface
DB	Database
ER	Entidade Relacionamento
DNS	Domain Name System
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
I/O	Iput/Output
IP	Internet Protocol
JSON	JavaScript Object Notation
JWT	JSON Web Token
OAUTH	Open Authorization
OO	Orientação a objetos
ORM	Object-Relational Mapping
XML	eXtensible Markup Language

SUMÁRIO

1 **INTRODUÇÃO**

Com o avanço da tecnologia e a demanda por sistemas mais flexíveis, a adoção da arquitetura de microsserviço se tornou quase que indispensável atualmente, justamente pela sua escalabilidade e o baixo acoplamento com outras partes do sistema, ao contrário de uma arquitetura monolítica, por exemplo. Nessa arquitetura os sistemas são desenvolvidos para serem independentes, mas considerando sempre a regra de negócio da organização como alvo principal.

Como a base desta arquitetura é um sistema distribuído, deve-se levar em consideração a melhor forma de diminuir a complexidade de cada serviço implementado, pois para o usuário final não é interessante saber a quantidade de serviços isolados que está sendo consumido, mas sim apenas parecer como um sistema unificado. Dito isso, a abordagem do serviço de API Gateway é um dos mais convenientes para essas situações, pois é possível abstrair o acesso aos serviços do *backend* por uma única entrada, como também redirecionar e proteger esses serviços de forma específica, não tendo que lidar com a complexidade de ter que projetar uma entrada diferente para cada serviço desejado.

Este trabalho tem como objetivo identificar e detalhar o uso do *API Gateway* atualmente, visando igualmente demonstrar as nuances da arquitetura de microsserviço, seu comparativo em relação à sistemas monolíticos e por fim justificar a utilização em conjunto dessas duas metodologias.

2 **FUNDAMENTAÇÃO TEÓRICA**

2.1 **MICROSSERVIÇO**

A concepção de desenvolvimento de um sistema é um leque de possibilidades, visto que é possível adotar, por exemplo, dois tipos de arquiteturas: a primeira sendo menos complexa e com alto nível de acoplamento e unificação ou o segundo tipo, em microsserviço, que parte para o princípio onde toda a regra de negócio do projeto da organização será composta por sistemas independentes, com o nível de acoplamento baixo, porém com o nível da complexidade maior.

Para Martin Fowler e James Lewis (2014) a arquitetura de microsserviço é definida como: "um conjunto de pequenos serviços, cada um rodando em um processo próprio e se comunicando por mecanismos leves, comumente sendo uma API rodando através do protocolo HTTP/HTTPS." Por esta definição, conseguimos extrair a ideia e fazer a analogia com uma fábrica de construção de automóveis: como um todo a fábrica possui um propósito e uma finalidade, tendo o seu escopo bem definido. Além disso tem-se o porquê de sua existência, onde cada setor possui sua própria finalidade e a sua autonomia, podendo fabricar, por exemplo, um material que é utilizado por outro setor. No entanto, por mais que dois setores sejam diferentes se analisarmos o que os mesmos fazem, levando em consideração um contexto geral, os dois setores que não se conversam necessitam existir para o funcionamento geral do processo em que os mesmos estão funcionando e integrados. Por essa analogia podemos concluir o mesmo sobre sistema distribuídos, onde cada serviço construído faz parte de um sistema maior, sem a necessidade do serviço A ou B se conhecer, compartilhar a mesma base de dado ou até a mesma linguagem.

Partindo do princípio de que o sistema principal é modularizado em componentes/serviços apartados, podemos facilmente isolar os riscos da aplicação, pois com componentes isolados conseguimos ter controle de cada serviço e do que o mesmo é composto. Assim, por exemplo, imaginando que um componente ficasse indisponível, a aplicação inteira não seria afetada, ficando apenas com indisponibilidade em uma *feature* neste caso. Pode-se imaginar também o cenário onde é necessário realizar uma manutenção ou a correção de um bug identificado no

serviço. Se as aplicações estão bem distribuídas e com seus princípios e deveres bem definidos é possível resolver o problema sem afetar o restante do sistema, pois não seria necessário rever toda a complexidade de um grande monolito para identificar uma falha ou ainda ter que rever todos os detalhes e métodos que são ativados internamente.

Conforme vamos destrinchando as razões de uma implementação de um sistema em microsserviço, é possível começar a entender as vantagens de se utilizar esta arquitetura, assim como também visualizar seus potenciais problemas. Estar ciente desses benefícios e malefícios é crucial para a escolha ou não do caminho a ser seguido, portanto é necessário estar atento não só ao custo-benefício, como também a identificar os *pattern* da arquitetura, sabendo utilizar bem e identificar, principalmente, quando uma escolha não faz parte do ciclo de vida de uma implementação em microsserviço.

2.1.1 **Princípios**

Como mencionado no tópico anterior, é preciso saber identificar as características que tornam um sistema um microsserviço, pois identificando cada particularidade, é possível obter um panorama dos requisitos para a implementação. Para Newman (2015), é preciso identificar os seguintes princípios:

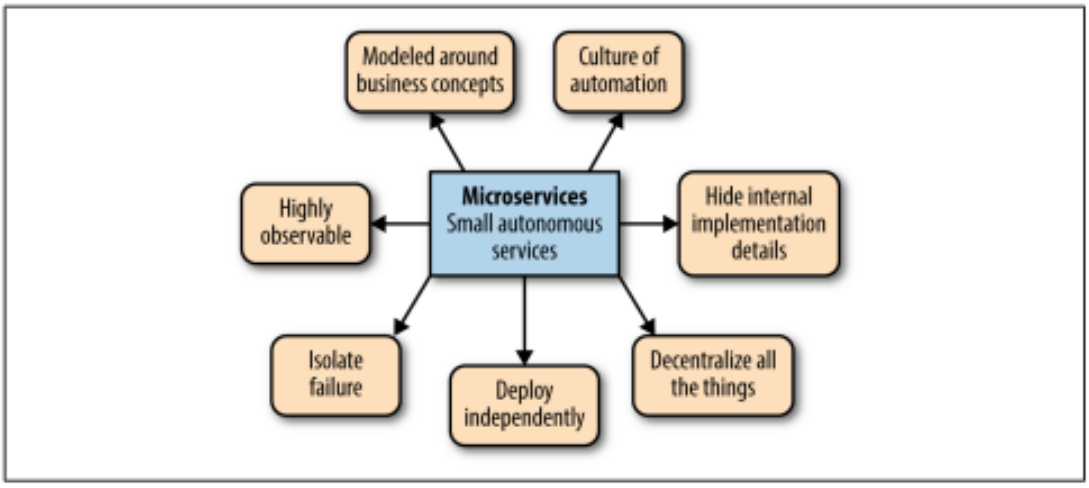
Abstrair e encapsular a implementação

- Isolamento de falhas
- Ter *deploys* independentes
- Os serviços devem estar voltados para a regra do negócio
- Utilizar ao máximo os serviços de monitoramento
- Encapsular as regras e detalhes de implementação
- Cultura da automatização
- Descentralizar ao máximo os serviços

De um modo geral, a utilização dos princípios pontuados acima visa ao máximo desacoplar não só equipes, mas também a responsabilidade do sistema, sugerindo a quem for implementar o uso de boas práticas da arquitetura de microsserviço, tornando ao máximo possível menos complexos os componentes dos sistemas. É

importante ressaltar que, ao escolher não implementar um dos princípios pontuados, é preciso ter noção do que está se abrindo mão no contexto geral e a médio e longo prazo do sistema da organização.

Figura 1 — Princípios de um sistema Microsserviço



Fonte: Building Microservice (Newman, 2015)

2.1.2 **Vantagem**

Abaixo estão listados alguns dos pontos fortes deste tipo de arquitetura

● Alta resiliência: como já mencionado anteriormente e ao contrário de sistemas monolíticos, um dos principais pontos desta arquitetura é o desacoplamento e a autonomia. No caso de falhas, apenas o componente X ou Y estará indisponível, não todo o sistema. Para o usuário final é melhor, pois dificilmente irá notar a falta do serviço, dependendo somente e exclusivamente se o serviço em questão possui alta demanda de requisição ou não. Para o desenvolvedor também é uma interessante forma de tratar os erros para o usuário final sem ter que fazer a famosa gambiarra.

● Escalabilidade: é possível escalar determinada funcionalidade da aplicação, como nova funcionalidade ou melhorias, sem ter que subir para todo o escopo geral da aplicação, indisponibilizando o sistema por conta de uma parte.

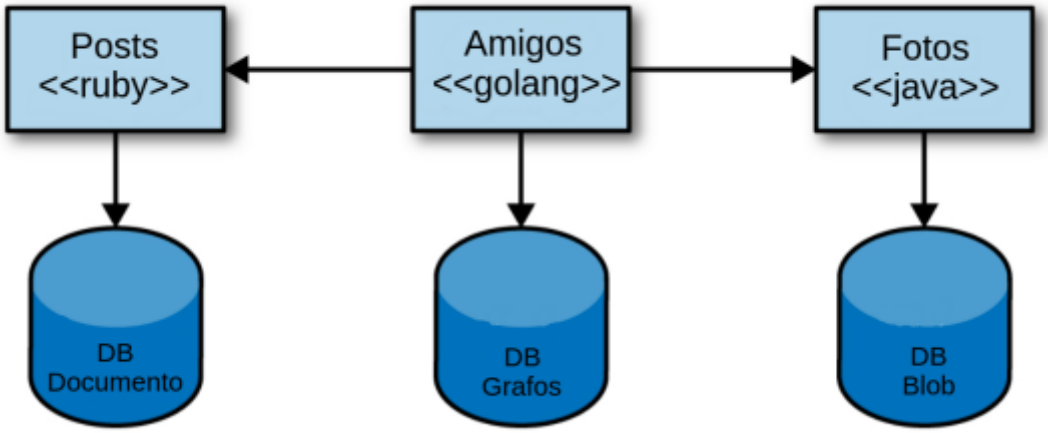
● Alta disponibilidade: seja um componente crítico para o negócio da organização ou até mesmo, um serviço que necessita de mais disponibilidade devido a uma demanda maior, é possível alocar o(s) serviço(s) em mais

servidores e tornar o desempenho e a velocidade de *throughput* maior para quem ou o que está consumindo o componente.

● Heterogeneidade de ferramentas e linguagem por serviço: esse ponto, talvez, seja o mais claro para quem está procurando entender a diferença da arquitetura em microsserviço versus arquitetura monolítica. Diferentemente da monolítica, não é preciso ficar preso a uma língua e a serviços concebidos e imaginados no início do projeto. Para cada serviço pensado e oferecido é possível e inclusive altamente recomendado, construí-lo com as mais variadas tecnologias e diferentes linguagens que atendam ao tipo de serviço a ser construído e disponibilizado, seja para o B2B ou B2C.

Como no exemplo da figura 2 abaixo, é possível visualizar cada serviço integrado com os demais, porém contendo diferentes tipos de DB com suas fontes específicas de dados, e diferentes linguagens direcionadas para cada tipo de componente disponibilizado.

Figura 2 — Sistema heterogêneo e seus serviços



Fonte: Building Microservice (Newman, 2015)

2.1.3 **Desvantagens a serem consideradas e superadas**

Independente do tipo de arquitetura que seja adotada, existirá uma série de fatores que façam com que o caminho escolhido seja mais tranquilo ou mais árduo. Do ponto de vista do microsserviço, esse desafio pode ser ainda mais doloroso se a

organização que o implementar não estiver com alguns paradigmas bem descontruídos e em mente durante a transição de um monolito legado para uma arquitetura em microsserviço.

Nesse caso, não se deve pesar apenas de forma técnica, mas também de forma cultural, como a implementação da cultura de Devops para se ter mais comunicação e estar em sincronia como uma equipe ou squads. Como Jennifer Davis e Ryn Daniels (2016, p. 64) sugerem, não é possível se conseguir comunicação adicionando, por exemplo, mais uma equipe. Isso não fará com que sua comunicação melhore, mas sim piore.

> "Criar uma equipe chamada ‘devops’ ou renomear uma existente não é o suficiente para criar uma cultura devops. Se sua organização está em um estado onde o desenvolvimento e a equipe de operação não se comunica um com o outro, uma equipe a mais causará mais ruído de comunicação, não ao contrário."

Abaixo veremos mais alguns pontos a serem considerados durante a criação de um projeto ou durante uma transição para o microsserviço:

- **Monitoramento realmente eficaz e obrigatório**: com uma aplicação monolítica conseguimos atrelar um sistema de monitoramento suficientemente capaz de analisar e gerenciar os I/O (input e output) de forma geral, tendo que fazê-lo só uma vez, pois toda a aplicação estará coberta com os logs e triggers configurados. Do ponto de vista do microsserviço, entretanto, é preciso gerenciar e pensar que tipo de monitoramento será implementado, pois cada componente possui sua própria linguagem, serviços e endpoints. No meio disso processo, deve existir um poderoso monitoramento no que diz respeito à infraestrutura que irá monitorar os requests e responses para diversos componentes. A não abordagem deste ponto pode fazer com que sua aplicação fique quase que impossível de rastrear problemas, sem a descrição do porquê e quando ocorreu a falha.
- **Mais atenção aos testes**: seguindo a premissa já abordada sobre as múltiplas linguagens e ferramentas que compõe múltiplos serviços, o teste escrito para garantir a integridade e pleno funcionamento do serviço necessita ser construído de forma diferente de uma tradicional aplicação monolítica, por exemplo. É preciso levar em consideração diversos fatores e ter em mente que,

se tratando de funcionalidades modularizadas, as mesmas podem ter seus problemas específicos, como por exemplo uma falha ao acesso ao banco que o serviço utiliza ou uma indisponibilidade temporária. Além disso, diferente da arquitetura monolítica, não é possível realizar testes de forma direta, por exemplo: como um sistema monolito se comporta de forma única, é possível identificar e querer realizar o teste de unidade em um método em específico, e é mais fácil de realizar esse tipo de teste, pois o código se encontra todo num mesmo escopo e interligado com outras partes também do mesmo sistema, o que torna mais fácil de identificar exatamente o que precisa ser validado e como, pois se tem conhecimento de todas as rotas e chamadas, subcamadas de um método desejado para o teste. Já pensando em microsserviço, esse exemplo não se enquadra tão bem, pois como é desacoplado de outros serviços, é preciso desenvolver um teste mais direcionado à tolerância a falhas e resiliência do serviço.

- **Custo**: até este ponto dá para se imaginar como a forma de modularização do sistema em componentes pode acarretar problemas, principalmente na área de complexidade, financeira e recursos. Como os recursos e a área financeira praticamente andam de mãos dadas, podemos citar que facilmente diversos serviços distribuídos podem trazer impacto nos recursos de memória, de banda e custo, pois para cada serviço existe seu próprio ecossistema e o que o faz funcionar. Como mencionado no ponto forte, podemos ter serviços em linguagens e utilizando ferramentas diferentes o que de fato é um ótimo ponto, porém sendo utilizado de forma estratégica com os recursos e limites financeiros e de infraestrutura da organização. Por exemplo: podemos ter vários serviços que consumam muito processamento de uma máquina caso a linguagem escolhida não seja a mais adequada, por exemplo. O mesmo pode ser dito das ferramentas utilizadas, pois não é preciso ir muito a fundo para imaginar um serviço A utilizando ferramentas menos custosas, mas um serviço B, por exemplo, utilizando ferramentas de alto nível e, portanto, com o valor de despesa alto. Por isso é preciso avaliar que funcionalidade será construída e pensar nos detalhes das ferramentas atreladas ao serviço, avaliar o custo-benefício, demanda pela funcionalidade e sempre pensando a longo prazo.

- **Equipe**: aproveitando o tópico anterior sobre custo, aqui também é um ponto importantíssimo, pois é preciso não só saber gerenciar e alocar o recurso humano correto à determinado serviço, mas também, dependendo da organização e o serviço oferecido, ter profissionais de diferentes domínios de linguagem e conhecimentos de diversas ferramentas.

Como podemos perceber nestes pontos, estar pensando na implementação de um microsserviço como arquitetura para sua organização ou projeto é mais do que apenas ser conveniente, é preciso estar ciente de ter uma equipe que esteja alinhada. Como abordado por Davis e Daniels (2016) é necessário extrema comunicação entre as equipes e squads, que as funcionalidades tenham seus requisitos e ferramentas bem definidas para evitar o gasto desnecessário e otimizar e gerenciar o uso de recursos de forma a visar o lucro e não o desperdício. Finalmente, mas definitivamente não menos importante, estar com a parte técnica alinhada, a infraestrutura preparada para suportar serviços diferenciados com todas as suas particularidades.

2.2 **API GATEWAY**

Com a ideia e a implementação de uma arquitetura voltada para microsserviços, também vem desafios, como por exemplo, manter o nível de complexidade que é ter vários serviços granularizados, com seu próprio ciclo de vida, autenticação, logs e especificações sobre configuração de rotas. A necessidade da utilização do serviço de API Gateway se mostra justificável, principalmente pensando em uma aplicação com serviços distribuídos e modularizados, visto que a premissa central do serviço é definir um ponto único, como um middleware, de I/O entre cliente e o backend, sendo o segundo uma coleção de microsserviços que podem se relacionar ou não. Para melhor entendimento, uma boa analogia seria o facade utilizado dentro do âmbito de design pattern focado em OO, onde temos um centralizador de interfaces que possui a capacidade de abstrair o máximo possível classes complexas para quem o está consumindo.

Como dito, o gateway fica responsável por fazer o papel intermediário, desta forma o usuário final ou o sistema parceiro não conseguem ter acesso aos endpoints do serviço X, Y ou Z. Para que isso aconteça basta apenas realizar a configuração de um DNS/IP único, que possui um gateway centralizador, e apenas mudar a rota desejada que foi exposta para utilização de acordo com o serviço desejado. Com isso temos o backend com sua coleção de serviços encapsulados e protegidos, tornando sua localização e acesso privado para o tráfego da internet. Podemos também ter uma lista vasta de serviços de apoio atrelado ao gateway que pode ser do mais simples monitoramento e relatório de I/O como até a autenticação e bloqueio de request por requisitante, por exemplo.

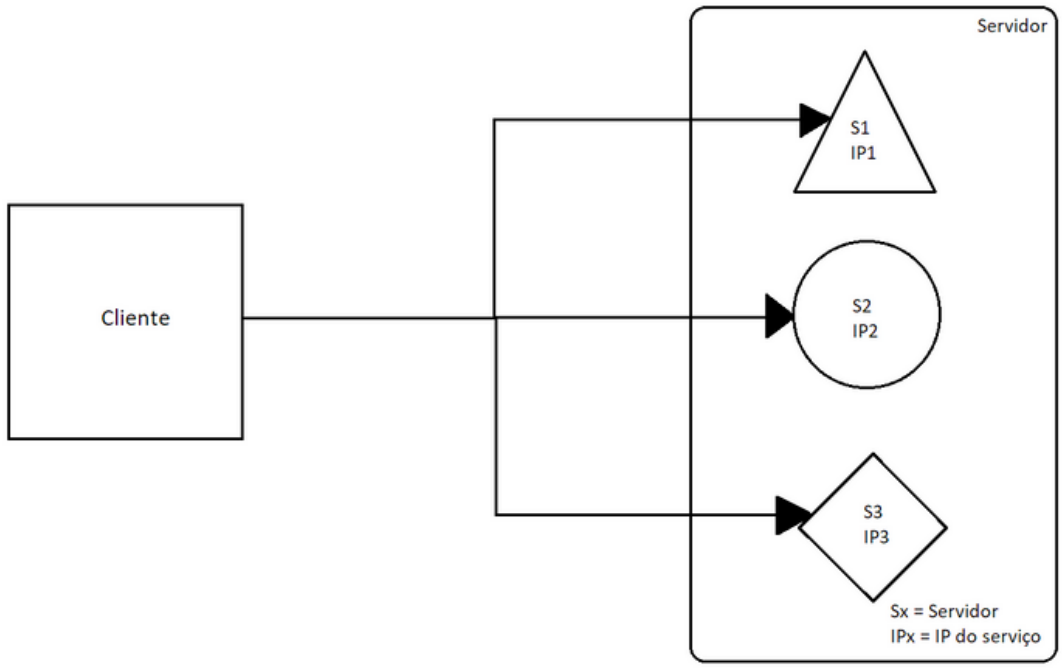
2.2.1 **Cliente, Gateway e Servidor**

Como um gateway consegue encapsular, abstrair e, principalmente, realizar roteamento de tráfego, pode-se ter como exemplificação a arquitetura de cliente-servidor como base para melhor compreensão de como o roteamento de uma aplicação se comporta sem e com a utilização de um gateway de API.

No cenário ilustrado abaixo temos um cliente que realiza uma requisição a um backend que possui três serviços modularizados e distribuídos em máquinas diferentes. Observando do ponto de vista do frontend responsável por renderizar as informações obtidas, pode-se pontuar duas complexidades: se preocupar com o DNS de cada serviço, visto que estão em hosts diferentes e lidar com diferentes tipos de autenticação por serviço e os seus protocolos de acesso.

Apesar do exemplo ser um cenário básico, onde um cliente qualquer consome apenas três serviços distintos, numa aplicação real implementar e manter chamadas distribuídas pode se tornar complexo sem a ajuda de um gateway, podendo até causar problemas de otimização e performance de I/O, por exemplo, se escalarmos os acessos e o número de serviços disponíveis para um valor muito maior que apenas três. A complexidade, o *throughput* e os problemas passariam a ser escalados verticalmente conforme a demanda e implementações fossem aumentando.

Diagrama 1 — Cliente-servidor sem api gateway



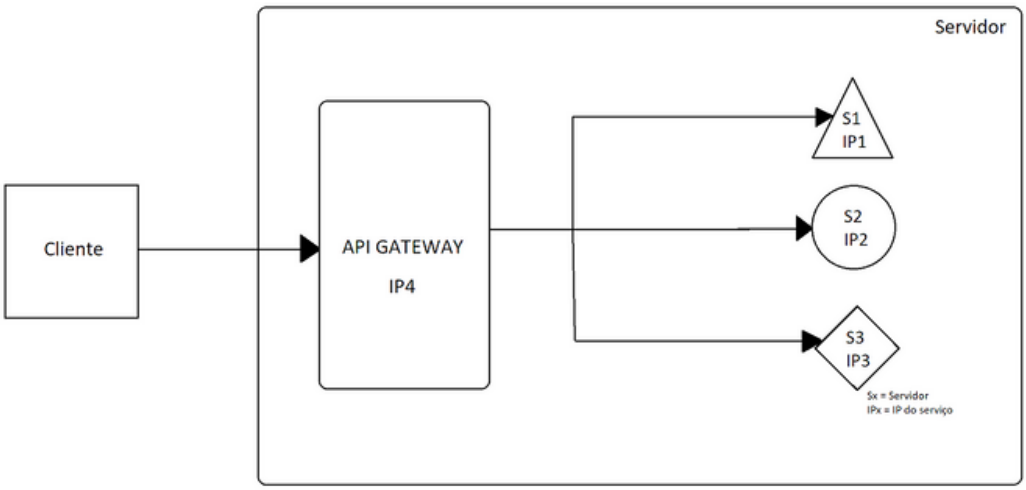
Fonte: Autoria pessoal

Já no segundo cenário, representado no diagrama abaixo, a mesma requisição vinda do lado do cliente atinge um *DNS* único de um provedor qualquer, apenas especificando o serviço que quer consumir (por exemplo **GET/api.gateway.com/produto/{{id}}**) que dispõe do serviço de API Gateway, que por sua vez fica responsável por abstrair o acesso aos três serviços. Tanto para o cliente, quanto para o servidor, é uma forma agradável e conveniente de lidar com a requisição.

Do ponto de vista do cliente, o mesmo não precisa conhecer quais serviços está acessando para obter o payload do response da aplicação, muito menos realizar, no mínimo, três requisições diferentes para conseguir os dados desejados, pois o gateway realiza as consultas necessárias dado que foi configurado pelo desenvolvedor no momento da configuração do endpoint da API “produto”. Dessa forma em apenas numa chamada é retornado um payload completo referente ao produto desejado pelo cliente, montado a partir de três chamadas diferentes sendo transparente para quem solicitou o serviço, pois parece apenas uma requisição. Já do ponto de vista do servidor, é conveniente pelo simples fato dos serviços não estarem

publicamente e diretamente acessível através da internet, precisando passar por um gateway.

Diagrama 2 — Cliente-servidor com api gateway



Fonte: Autoria pessoal

2.2.2 **Recursos oferecidos por um API Gateway**

Por mais que a função principal de um gateway seja o de roteamento de requisições, também é oferecido outras *features*, como por exemplo:

- Autenticação: ao receber uma requisição, é possível, para um endpoint em específico, definir qual autenticação será utilizada para validar o acesso a coleção de serviços internos, tais como jwt, oauth, http basic. Também é possível realizar o roteamento para um serviço externo, um outro endpoint que dispõe de uma autenticação própria, por exemplo.
- Métricas: como o gateway está disposto no meio do tráfego entre cliente servidor é o melhor local para se medir o I/O e gerar relatórios, alertas ou bloqueios baseado no início e fim de uma requisição. É possível realizar levantamentos individuais de quantidade de chamadas de um usuário em específico, como também a quantidade de requests recebidos por microsserviço.
- Response: um problema comum quando se trata de um sistema que não utiliza o serviço de api gateway é de ter que lidar com múltiplos tipos de responses com formatos diferentes. No entanto, passando por um gateway, o cliente não precisa

se preocupar com um microsserviço X devolvendo uma resposta em JSON e o microsserviço Y devolvendo um XML, pois já é feito a tradução do tipo de resposta vindo do servidor. No cenário onde um serviço devolve um payload no formato XML e o requisitante deseja em JSON, o gateway consegue lidar com a tradução fazendo um parse para o formato desejado.

3 **APLICAÇÃO**

Para fins de exemplificação, uma aplicação utilizando o padrão de arquitetura em microsserviço e o serviço de API gateway foi construído e será aqui detalhado de forma a compreender na prática alguns dos pontos apresentados no capítulo anterior.

3.1 **TECNOLOGIAS UTILIZADAS**

De forma geral, tanto o microsserviço quanto o serviço de API gateway utilizaram do **docker** e **docker compose** para orquestrar todos os *containers* criados e utilizados nas aplicações.

3.1.1 **Microsserviço**

O microsserviço implementado possui apenas a responsabilidade de gerenciar as categorias, produtos e consequentemente o estoque de um catálogo qualquer.

O PHP versão 7.4 foi utilizado como meio para a regra de negócio acima apresentada. Para as abstrações de implementação e sintaxe, o mini framework "Lumen" também foi utilizado e escolhido justamente por ser mais leve em comparação ao Laravel framework e pela aplicação apenas conter APIs.

A persistência utilizada é relacional e foi utilizado o MariaDB (versão 10.4) para este serviço e por fim o nginx é utilizado como servidor HTTP.

3.1.2 **API Gateway**

O Gateway utilizado nessa implementação, que também é *open source*, foi o **Kong** e ele que é responsável por realizar o roteamento para o IP do microsserviço de "gerenciamento de catálogo" e suas interfaces disponíveis.

O **Konga** também está sendo utilizado como GUI para atuar em conjunto com o **Kong** e realizar as configurações gerais do API gateway.

A persistência utilizada é o PostgreSQL, também *open* source.

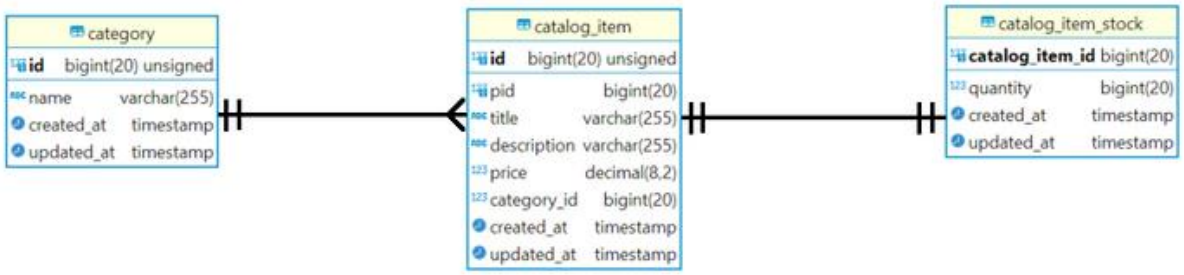
3.2 **CENÁRIO**

3.2.1 **Modelagem ER do microsserviço**

O microsserviço responsável por gerenciar um catálogo possui três entidades:

- "***category***", responsável por gerenciar as categorias que os produtos estarão contidos
- "***catalog_item***", responsável por gerenciar os produtos propriamente dito, correlacionando uma categoria ao mesmo, como também valor, descrição e nome
- "***catalog_item_stock***", responsável por gerenciar a quantidade do estoque de um determinado produto da entidade "catalog_item"

Diagrama 3 — Modelagem ER do microsserviço de gerenciamento de catálogo

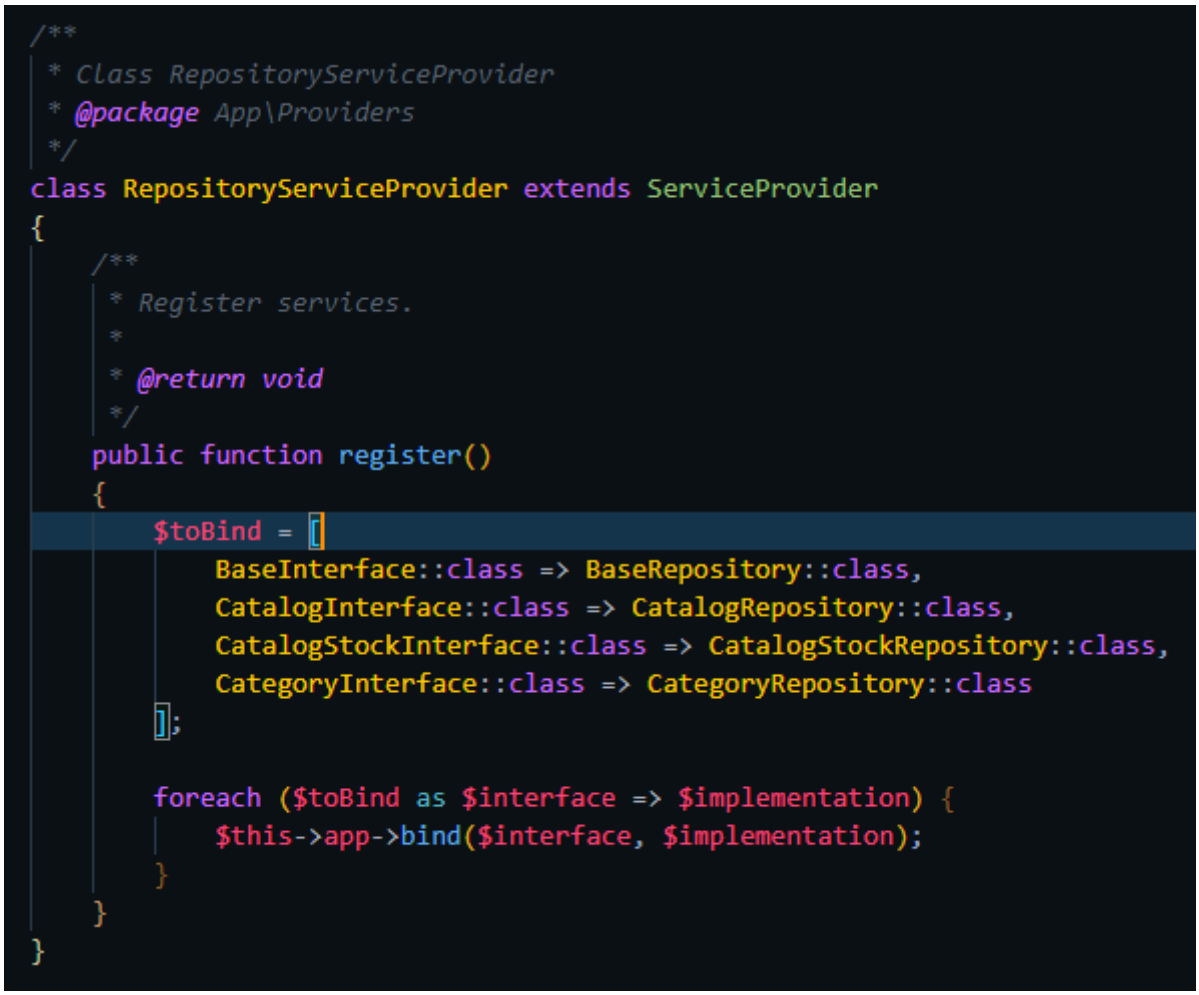


Fonte: Autoria pessoal

3.2.2 **Modelagem da ER na aplicação**

As entidades estão representadas em classes do tipo de modelo, repository e interface. A utilização da interface como parte do processo de acesso a entidade se dá, pois foi utilizado neste microsserviço o ***repository pattern***, que é uma forma de encapsular totalmente o acesso direto a um modelo por outras classes/métodos, ficando apenas acessível através de um *repository,* que por sua vez somente esta classe fica responsável pelo acesso a manipulação dos dados de uma entidade. A interface, por sua vez, é utilizada para dar *bind* a um *repository* com a técnica de *service cointainer* provido pelo Laravel/Lumen.

Imagem 1 — Utilização da técnica de service container do laravel

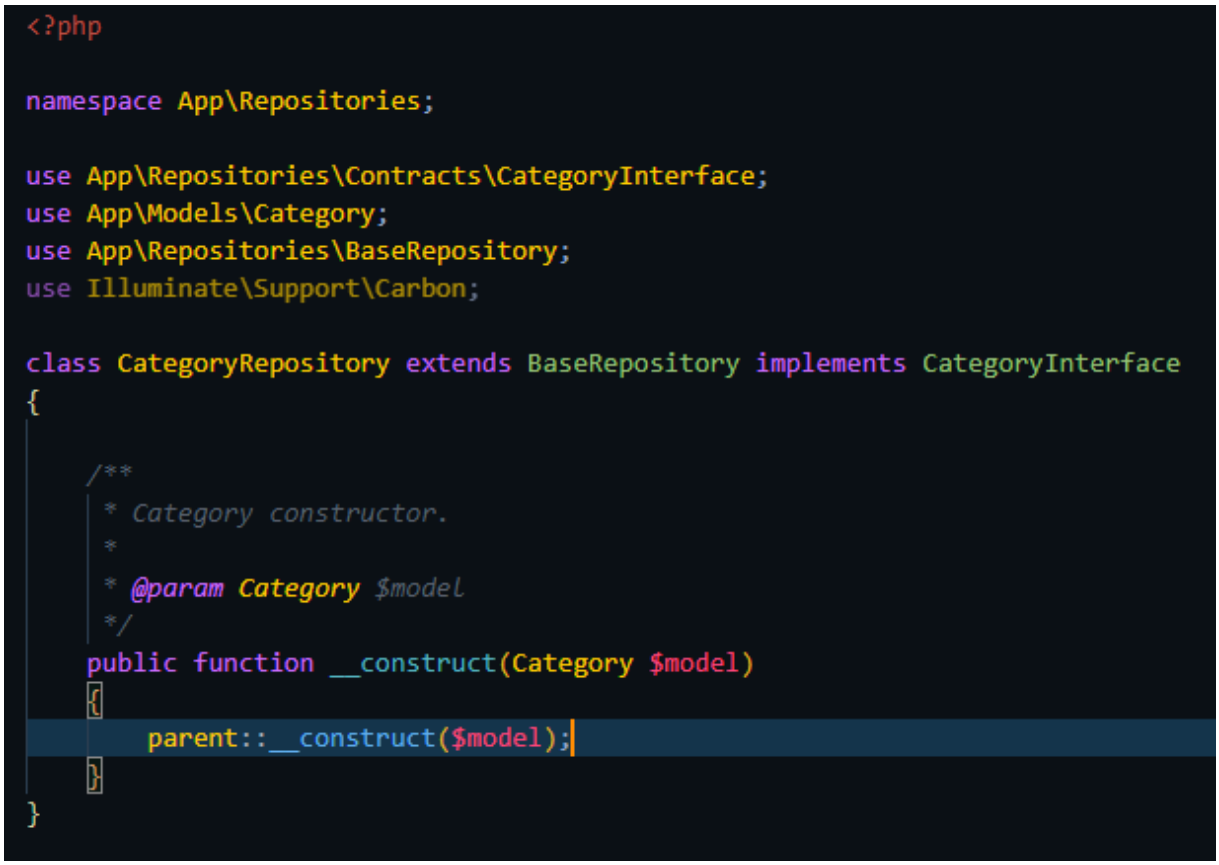


```php
/**
 * Class RepositoryServiceProvider
 * @package App\Providers
 */
class RepositoryServiceProvider extends ServiceProvider
{
    /**
     * Register services.
     *
     * @return void
     */
    public function register()
    {
        $toBind = [
            BaseInterface::class => BaseRepository::class,
            CatalogInterface::class => CatalogRepository::class,
            CatalogStockInterface::class => CatalogStockRepository::class,
            CategoryInterface::class => CategoryRepository::class
        ];

        foreach ($toBind as $interface => $implementation) {
            $this->app->bind($interface, $implementation);
        }
    }
}
```

Fonte: Autoria pessoal

Desta forma, em termos práticos, só é necessário declarar a interface em um construtor de uma classe qualquer, por exemplo um *controller*, e ela já terá acesso a classe *repository* responsável por lidar com o acesso a entidade, tornando então a ER encapsulada e abstrata.

Imagem 2 — Classe do tipo repository que contém acesso a entidade Category



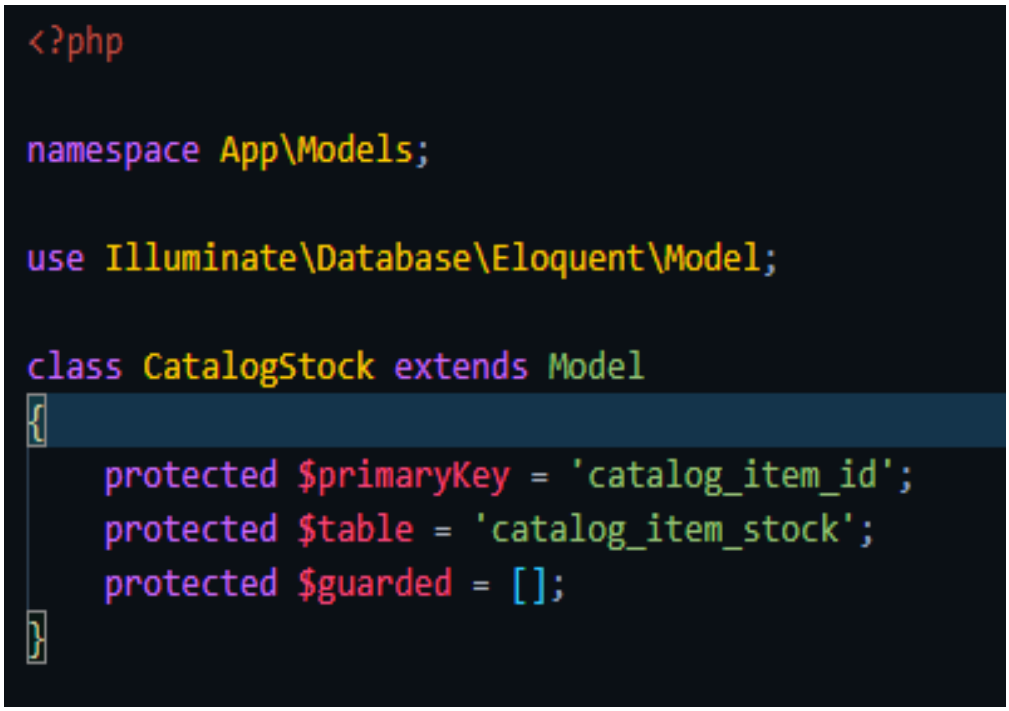
```php
<?php

namespace App\Repositories;

use App\Repositories\Contracts\CategoryInterface;
use App\Models\Category;
use App\Repositories\BaseRepository;
use Illuminate\Support\Carbon;

class CategoryRepository extends BaseRepository implements CategoryInterface
{

    /**
     * Category constructor.
     *
     * @param Category $model
     */
    public function __construct(Category $model)
    {
        parent::__construct($model);
    }
}
```

Fonte: Autoria pessoal

No exemplo acima, a entidade *Category*, do banco relacional, é referenciada no construtor da classe *CategoryRepository* que é repassada ao seu pai, *BaseRepository*, que possui métodos genéricos de manipulação de dados de uma entidade qualquer. Demais métodos específicos de uma entidade devem ficar somente concentrado na classe *repository* específica da entidade, por exemplo *CategoryRepository*.

A classe do tipo model, por sua vez, fica responsável apenas pelas configurações de acesso a entidade, como por exemplo a especificação de quais atributos são necessários durante um *update* e a tabela a ser utilizada como referência para o *ORM* do laravel/lumen poder atuar sobre.

Imagem 3 — Entidade catalog_stock sendo referenciada e configurada em um model



Fonte: Autoria pessoal

E por fim, como mencionado anteriormente, só é preciso referenciar a interface, que nos garante acesso ao ORM encapsulado através de um repository, em um construtor qualquer e a partir disto está pronto para ser manipulado durante seu processo de *runtime.*

Imagem 4 — Controller responsável por conter as interfaces da aplicação de Catalog

Fonte: Autoria pessoal

3.2.3 **Rotas para acesso às interfaces**

As rotas foram configuradas contendo agrupamentos, dividindo diferentes tipos de APIs contidas no mesmo controller central, o CatalogController. O controller é responsável por prover acesso a manipulação de um produto, categoria e estoque.

Imagem 5 — Rotas e agrupamentos responsáveis por lidar com as requisições ao CatalogController

Fonte: Autoria pessoal

3.2.4 **Konga, o GUI do Kong**

O **Kong** foi configurado através do serviço ***Konga***, que serve de interface gráfica para as *APIs restful* de configuração de um **Kong**. No ***Konga*** é possível realizar qualquer tipo de configuração, sem necessidade de uma requisição direta por *CLI*, por exemplo. Assim como é possível ter uma visualização de conexões abertas e *requests* recebidos, por exemplo, no *dashboard* principal da interface.

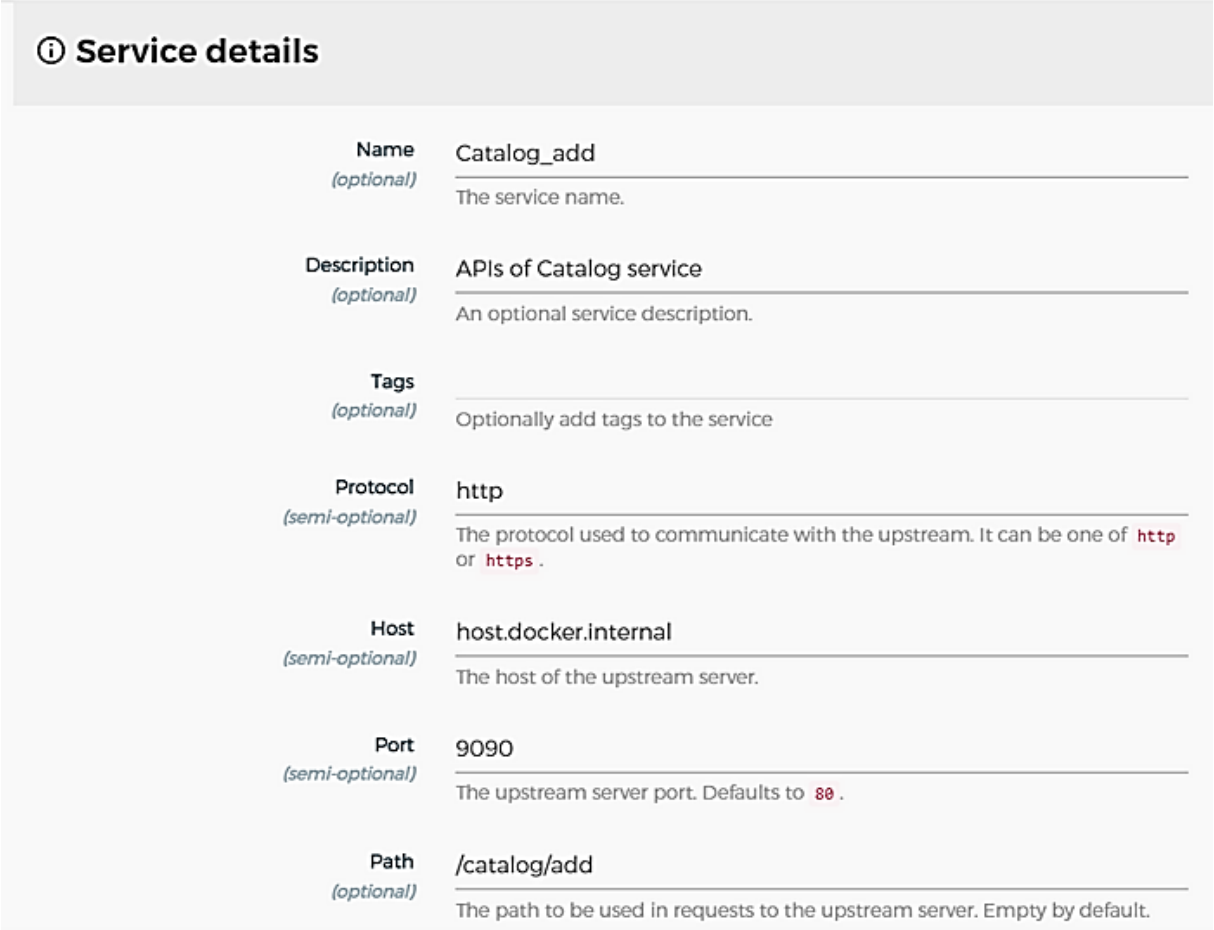
3.2.4.1 **Serviços**

A configuração e criação de um serviço dentro do kong representa um serviço referenciado através do seu IP e porta. O serviço é responsável por encapsular a rota do microsserviço a ser conectado, protegendo a aplicação provedora dos dados.

O microsserviço construído, responsável por gerenciar um catálogo, atende e responde requisições através da porta 9090 e o localhost como host, que é onde está configurado o nginx da aplicação.

A imagem abaixo representa a criação e a configuração de um novo serviço, no caso referenciando o serviço de criação e persistência de um novo produto pro microsserviço de catálogo. Como mostrado na imagem 5 deste capítulo, a rota que foi configurada agora está sendo referenciada na propriedade "path" dentro desse serviço do kong.

Imagem 6 — Registrando um novo serviço que referência a rota /catalog/add do microsserviço



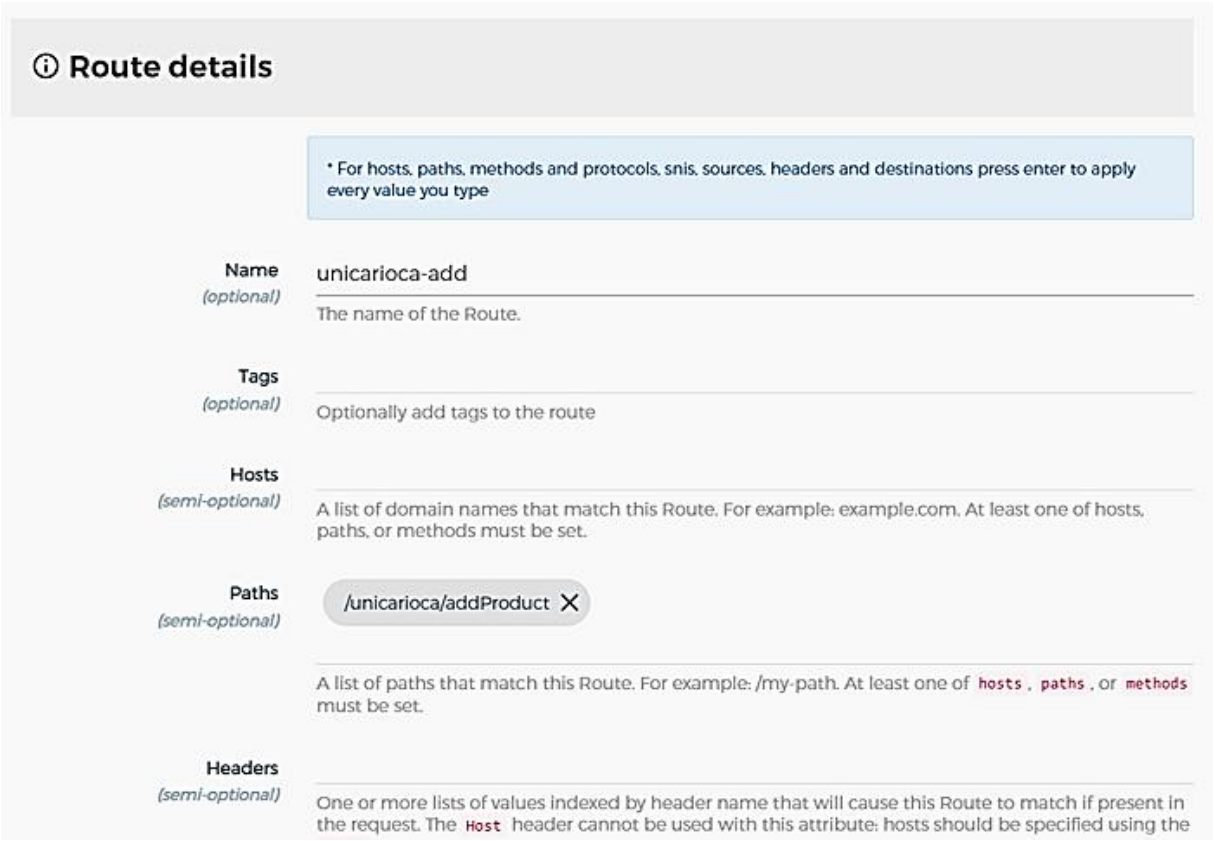
Fonte: Autoria pessoal

3.2.4.2 **Rotas**

As rotas, por sua vez, possuem o papel de integrar com um serviço e ditar regras para atender a determinada requisição. Uma rota contém apenas um serviço e

um serviço pode conter múltiplas rotas. As possibilidades de configuração de uma rota para um serviço permitem que o administrador crie rotas diferentes para determinada necessidade, com suas próprias autenticações, aliases e demais opções que o kong oferece.

Imagem 7 — Rota personalizada para a API /catalog/add do microsserviço



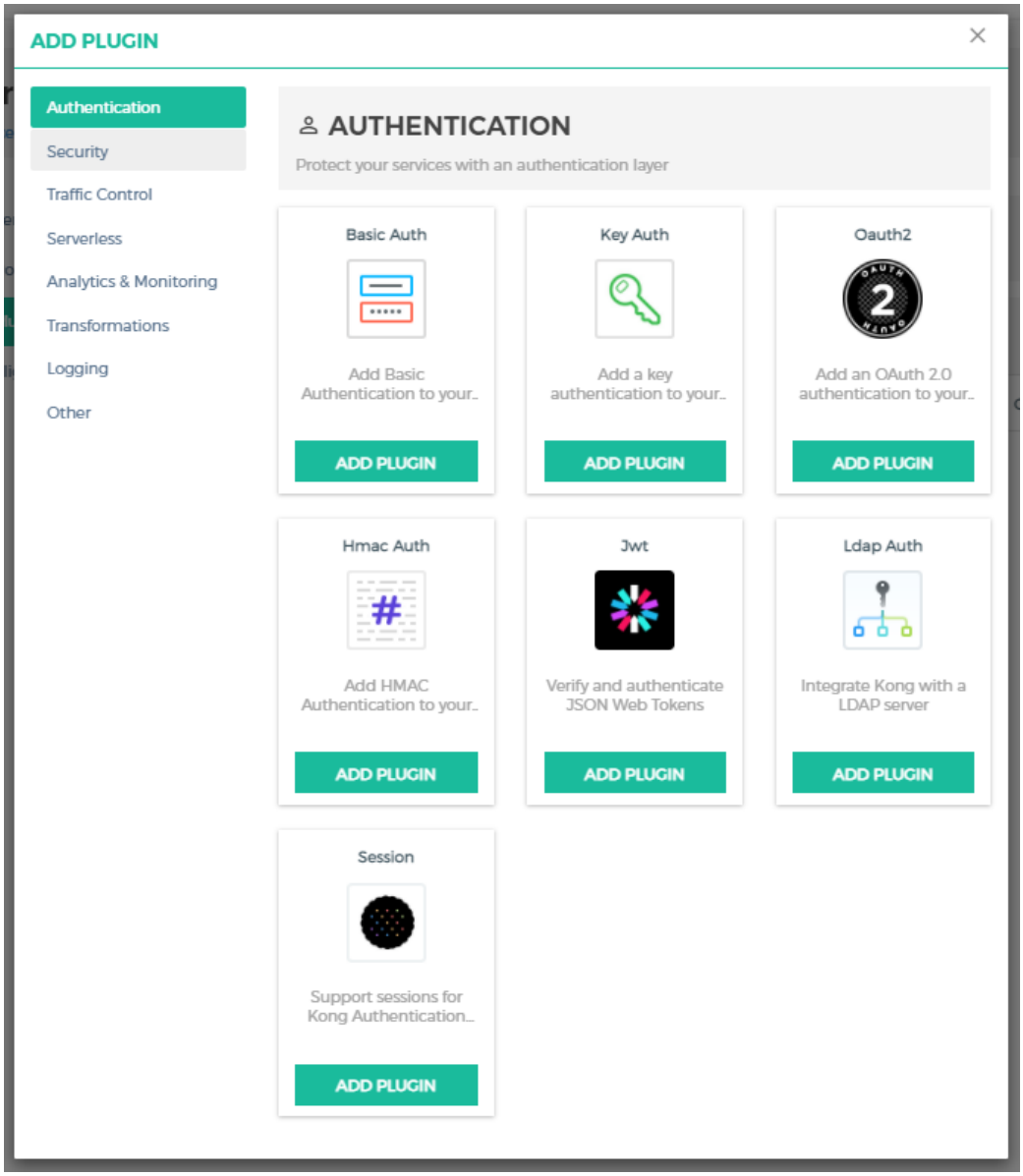
Fonte: Autoria pessoal

3.2.4.3 **Plugins**

Os plugins são responsáveis por atuarem dentro do API gateway, auxiliando o administrador do sistema com os requests e os responses das aplicações conectadas ao gateway. Como mencionado na parte teórica, o melhor local para se ter disposto ferramentas de análises, de bloqueio, limitação ou autenticação é justamente no API gateway.

Um serviço pode conter múltiplos plugins, o que torna geral, independente das rotas, para os clientes que realizarem o request ou pode ser configurado plugins para cada rota criada para um determinado serviço, o que possibilita, por exemplo, que para uma rota em específico exista um conjunto de plugins apartado de uma outra rota que também encaminha o request para o mesmo serviço.

Imagem 8 — Lista de plugins disponíveis para uso



Fonte: Autoria pessoal

3.2.5 **Resultados**

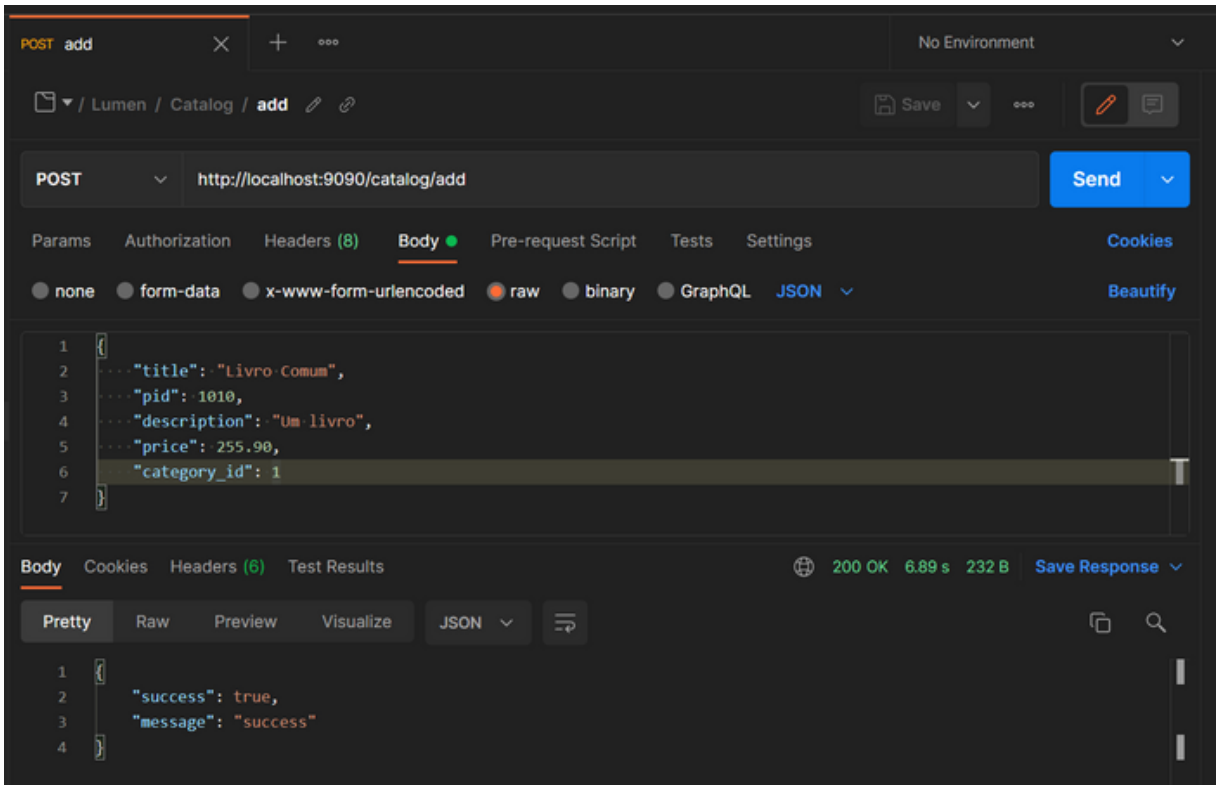
Dado as exemplificações dos tipos de configurações disponíveis dentro do gateway do **Kong**, os resultados serão demonstrados neste tópico utilizando cada tipo de configuração apresentada.

3.2.5.1 **Rota personalizada**

Foi utilizado como base de exemplificação o endpoint "/catalog/add" configurado nas rotas do microsserviço, demonstrado na imagem 5 deste capítulo.

A primeira chamada realizada foi diretamente ao IP do microsserviço de catálogo, como demonstra a imagem abaixo.

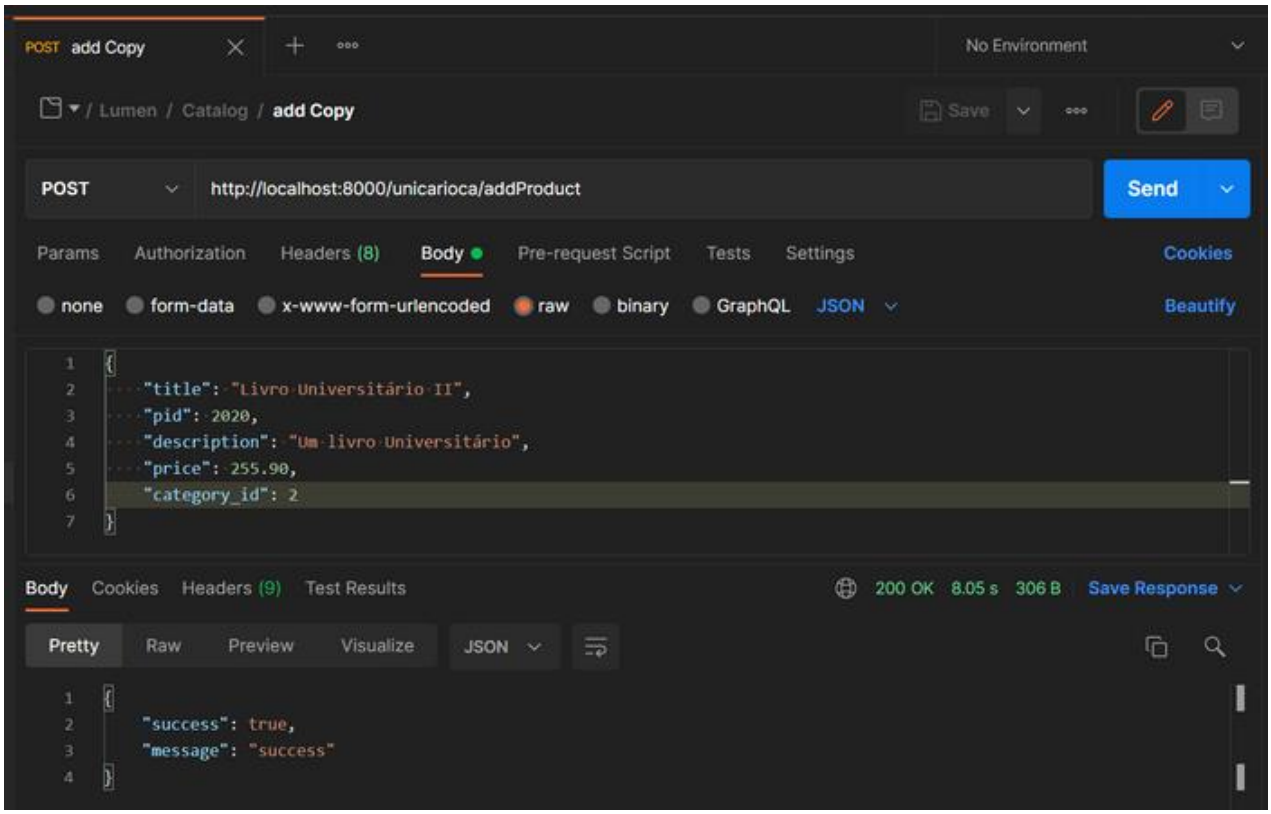
Imagem 9 — Request realizado diretamente a rota do microsserviço sem o api gateway



Fonte: Autoria pessoal

Já no segundo exemplo da imagem abaixo, é possível realizar o mesmo tipo de request, porém de forma que o IP do serviço final fique encapsulado ao cliente.

Imagem 10 — Request realizado indiretamente à rota do microsserviço com o Kong API Gateway



Fonte: Autoria pessoal

Como nesse cenário o kong gateway utiliza a porta 8000, apenas é preciso referenciá-la juntamente à rota personalizada criada a partir de um serviço.

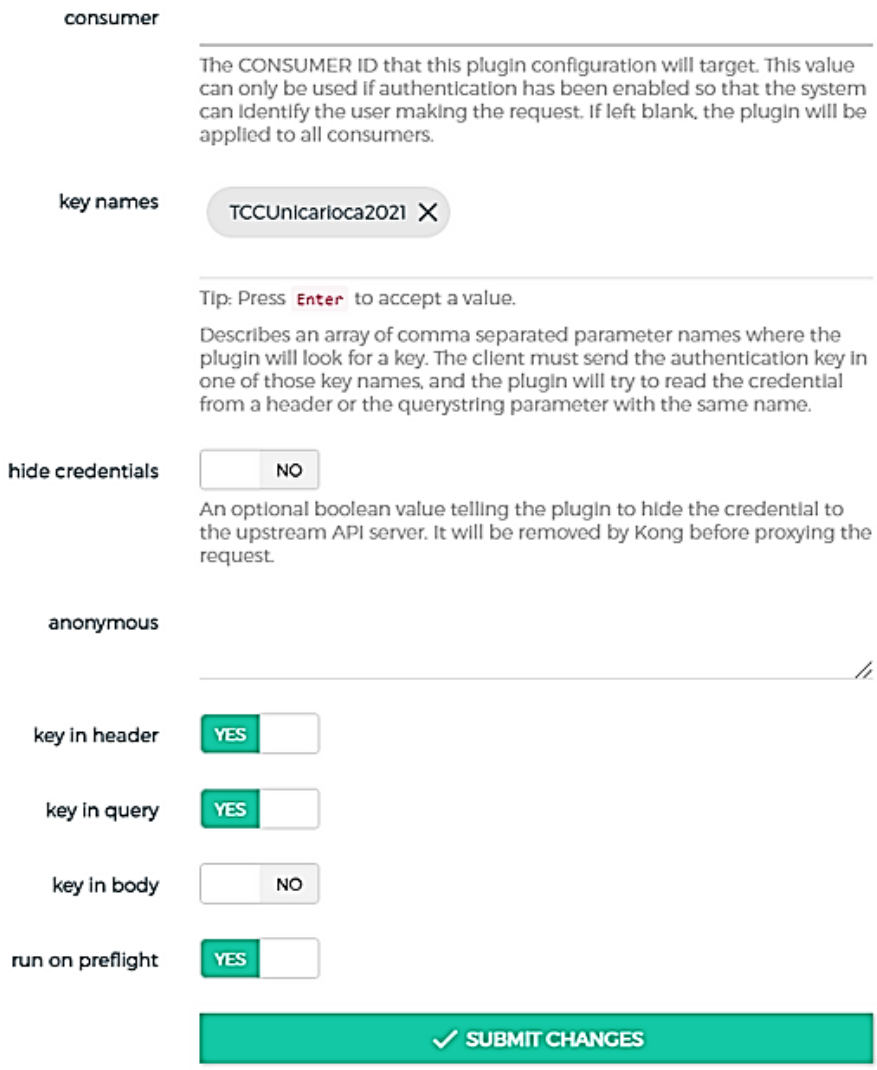
3.2.5.2 **Plugin na prática**

A demonstração será com base na rota configurada na imagem 7 e o payload do request da imagem 10 deste capítulo.

3.2.5.2.1 ***Key Auth***

A chave de acesso configurada para a rota "/unicarioca/addProduct" é demonstrada na imagem abaixo.

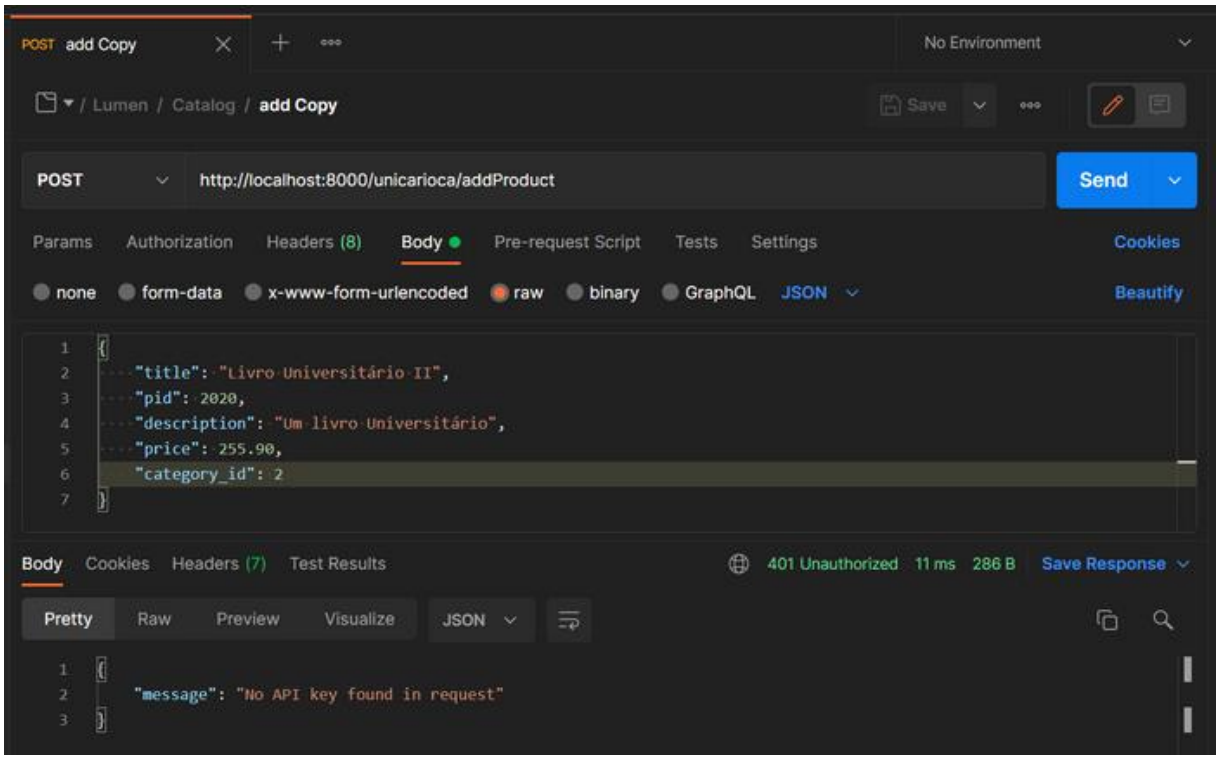
Imagem 11 — Criação da autenticação via key auth



Fonte: Autoria pessoal

Após aplicar a configuração, ao tentar realizar a mesma chamada para a mesma rota, é retornado uma exceção devido à falta da chave de acesso registrada no header do request.

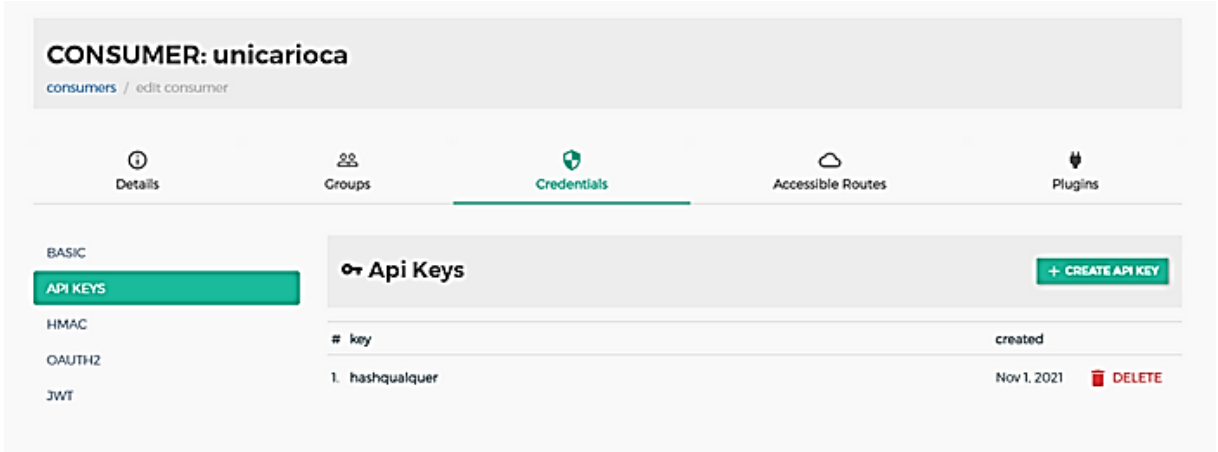
Imagem 12 — Falha no request após não inserir a nova chave para acesso



Fonte: Autoria pessoal

Após a criação da chave, é preciso criar o valor a ser associado. Dentro do serviço de "consumers", é possível criar um consumidor qualquer e especificar o tipo de autenticação do mesmo, o que no cenário da exemplificação é o key auth. O valor definido para a chave "TCCUnicarioca2021" é "hashaleatorio", como demonstrado na imagem abaixo.

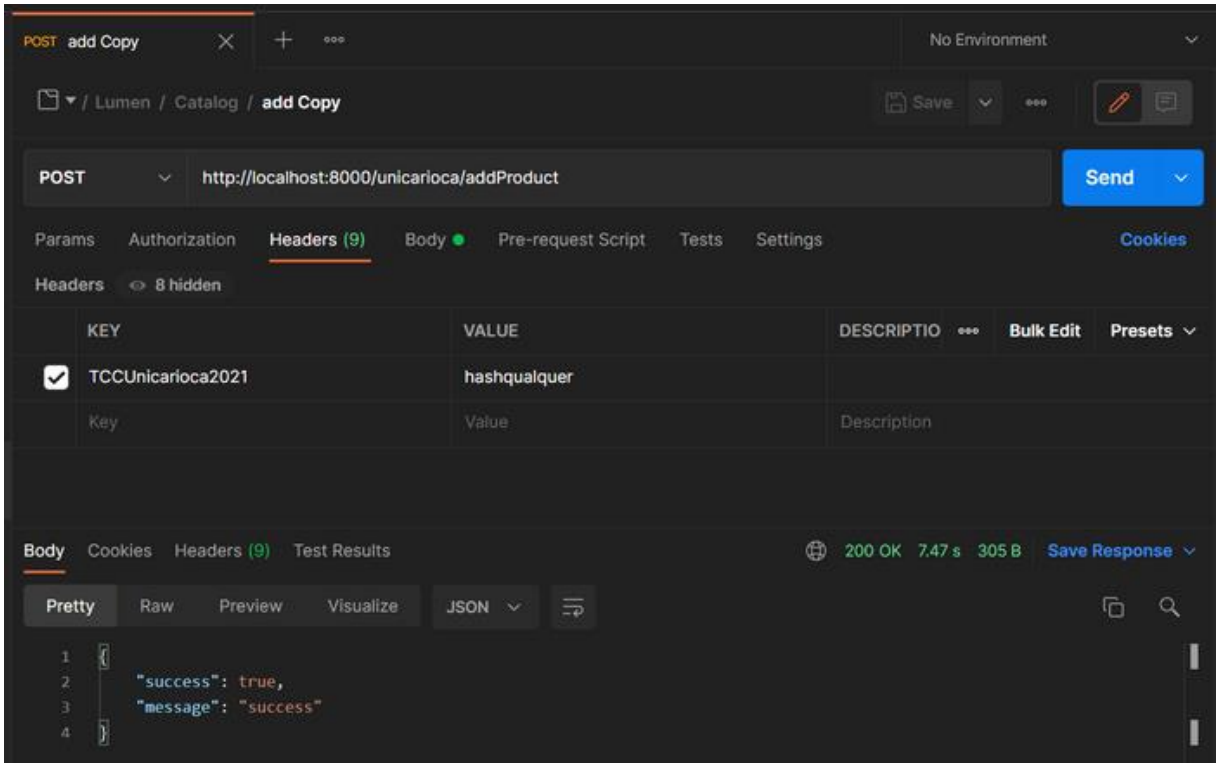
Imagem 13 — Definindo valor de key auth para o consumidor "unicarioca"



Fonte: Autoria pessoal

Ao realizar novamente o request, agora passando no header do payload do request a chave e valor respectivamente sendo "TCCUnicarioca2021" e "hashaleatorio", é possível retornar o acesso ao endpoint de cadastro de produto.

Imagem 14 — Request bem-sucedido após utilizar o key auth esperado



Fonte: Autoria pessoal

4 **CONCLUSÃO**

Neste trabalho acadêmico foi apresentado a utilização do serviço de API Gateway, sendo o **Kong** o provedor, com a arquitetura em microsserviço. O projeto usou PHP como linguagem base e o framework Lumen para facilitar as abstrações das APIs e o acesso às entidades da aplicação.

Os dois conceitos foram detalhados, seja destrinchando a diferença entre a arquitetura monolítica e a de microsserviço, como também o serviço de API Gateway, que durante este trabalho percebe-se como a utilização deste serviço facilita nos pontos de escalabilidade, encapsulamento, abstração e a manutenção dos serviços como um todo de uma organização.

Dado o cenário e os escopos das aplicações apresentadas neste trabalho, fica aberta a possibilidade de projetos futuros implementarem o uso de diferentes tipos de serviços de API Gateways disponíveis no mercado, como também a utilização de outros tipos de subserviços, sejam eles de autenticação, de *rate limit* ou o que for necessário para o sistema beneficiado com o gateway.

REFERÊNCIAS

DAVIS, Jennifer; DANIELS, Ryn. **Effective DevOps**: Building a Culture of Collaboration, Affinity, and Tooling at Scale. "O'Reilly Media, Inc.", v. 3, f. 205, 2016. 410 p. Disponível em: https://www.amazon.com.br/Effective-DevOps-Building-Collaboration-Affinity-ebook/dp/B01GGQKXOE/ref=tmm_kin_swatch_0?_encoding=UTF8&qid=&sr=. Acesso em: 20 ago. 2021.

FOWLER, Martin. **Microservices**. Martin Fowler. 2014. Disponível em: https://martinfowler.com/articles/microservices.html. Acesso em: 25 set. 2021.

FOWLER, Susan J.. **Microsserviços prontos para a produção**: Construindo sistemas padronizados em uma organização de engenharia de software. Novatec Editora, v. 2, f. 112, 2019. 224 p. Disponível em: https://www.amazon.com.br/Microsservi%C3%A7os-prontos-para-produ%C3%A7%C3%A3o-padronizados-ebook/dp/B07QMWDMC2/ref=tmm_kin_swatch_0?_encoding=UTF8&qid=&sr=. Acesso em: 20 set. 2021.

NEWMAN, Sam. **Migrando sistemas monolíticos para microsserviços**: Padrões evolutivos para transformar seu sistema monolítico. Novatec Editora, f. 144, 2020. 288 p. Disponível em: https://play.google.com/store/books/details/Sam_Newman_Migrando_sistemas_monol%C3%ADticos_para_micr?id=eLfZDwAAQBAJ. Acesso em: 29 ago. 2021.

SAUDATE, Alexandre. **APIs REST**: Seus serviços prontos para o mundo real. Casa do Código, v. 3, f. 168, 2021. 335 p. Disponível em: https://www.casadocodigo.com.br/products/livro-apis-rest. Acesso em: 1 out. 2021.

TEAM, Documentation. **Amazon API Gateway Developer Guide**, f. 358. 2018. 716 p. Disponível em: https://www.amazon.com/Amazon-API-Gateway-Developer-Guide-ebook/dp/B078YH6445. Acesso em: 3 out. 2021.

TEAM, Documentation. **AWS Serverless Application Model: Developer Guide**. Disponível em: https://www.amazon.com/gp/product/B07P7K9VZB/ref=dbs_a_def_rwt_bibl_vppi_i6. Acesso em: 14 out. 2021.