FACULDADE DE TECNOLOGIA DE SÃO PAULO

JULIO DA SILVA CRUZ

MODELAGEM DE MICROSSERVIÇOS E ARQUITETURAS DE APIS

SÃO PAULO

2021

JULIO DA SILVA CRUZ

MODELAGEM DE MICROSSERVIÇOS E ARQUITETURA DE APIS

Trabalho de conclusão de curso de graduação apresentado como requisito para obtenção do grau de Tecnólogo em Análise e Desenvolvimento de Sistemas.
Professora: Grace Anne Pontes Borges

SÃO PAULO

2021

RESUMO

A arquitetura em microsserviços propõe maneiras de facilitar o trabalho em equipe em grandes projetos, porém sua implementação requer padrões arquiteturais complexos. Como toda arquitetura ela possui padrões estabelecidos e debatidos na comunidade, uma vez que a utilização da arquitetura independe da tecnologia utilizada para sua construção, possibilitando uso de *.Net*, *Java* e outras. As convenções desta arquitetura, quando má interpretadas podem trazer sérios problemas sistêmicos, com a aparição de antipadrões. Nesta arquitetura podemos utilizar modelos de *APIs* como *gRPC*, *REST, SOAP* e *GraphQL*. A escolha entre os modelos de construção de *APIs* aplicados na arquitetura de microsserviços traz ao desenvolvedor a necessidade de conhecer esses padrões para saber onde utilizá-los e como utilizá-los, pois cada implementação traz características diferentes que são mais vantajosas em alguns cenários do que em outros.

Palavras-chave: Microsserviços; *.Net; REST; GraphQL; SOAP*

ABSTRACT

Microservice architecture proposes ways to facilitate teamwork in large projects, but its implementation requires complex architectural patterns. Like any architecture, it has established and debated standards in the community, since the use of the architecture does not depend on the technology used for its construction, enabling the use of .Net, Java and others. The conventions of this architecture, when misinterpreted, can bring serious systemic problems, with the appearance of anti-patterns. In this architecture we can use API models such as gRPC, REST, SOAP and GraphQL. Choosing between the API construction models applied in microservices architecture brings the developer to know these patterns in order to know where to use them and how to use them, as each implementation brings different characteristics that are more advantageous in some scenarios than in others.

Keywords: Microservices; .Net; REST; GraphQL; SOAP

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

Seja uma aplicação monolítica em cascata ou D*evOps* com microsserviços, uma aplicação passa por mudanças sejam elas em suas regras ou nas tecnologias utilizadas em alguma parte do sistema. Essas mudanças podem ocorrer em qualquer fase do projeto, e um ponto importante em qualquer alteração é garantir que a aplicação como um todo permaneça funcionando após a sua realização.

Esse tipo de preocupação envolve a modelagem de uma aplicação, onde termos como estabilidade, escalabilidade, performance e segurança aparecem com frequência e estão relacionados com a facilidade de *deploy* da aplicação.

1.1 CONTEXTO ATUAL

Conforme as aplicações corporativas crescem acompanhando o mercado, é natural que a complexidade relacionada ao escopo e tamanho do projeto dificulte, em aplicações monolíticas, a manutenção e a implantação da aplicação. Escalabilidade, eficácia na implantação, entrega contínua tornam-se preocupações comuns para as empresas, este fato tem feito as empresas migrarem sua arquitetura monolítica para arquiteturas baseadas em microsserviços.

A maior motivação desta migração é garantir o desacoplamento da aplicação, ou seja, criar serviços apartados. Os serviços apartados funcionam como partes independentes do sistema, representadas por *APIs* (*Application Programming Interface*) que quando integradas com os demais serviços compõem o sistema. Esse desacoplamento traz grandes vantagens em termos de implantação facilitando-a, também assegura que o sistema não fique *offline* por causa de uma implantação que afetaria apenas determinado escopo do negócio. Recentemente a própria Netflix passou por essa transição, que conforme descreveu Adrian Cockroft, arquiteto de sistemas na empresa, que descreve que a alteração se fez necessária para construir sistemas e fazer mudanças o mais rápido possível.

Como podemos ver temos grandes vantagens ao aplicar os padrões de microsserviços em aplicações corporativas, mas uma questão interessante é sua usabilidade contra os custos envolvidos em implantar um padrão complexo como este. Arquiteturas de pequeno, médio ou grande porte enfrentam esse debate durante a elaboração de uma proposta arquitetural que envolva microsserviços, então conhecer a fundo o que o padrão exige e considerar a adoção de um sistema distribuído baseado em microsserviços também é uma abordagem interessante para aqueles que analisando prós e contras desta distribuição muito dispendiosa queiram dispor de um meio de sair da arquitetura monolítica adotando algumas práticas desta arquitetura.

Nesse ponto de análise dos requisitos para a implantação de uma arquitetura em microsserviços, o estudo equivocado pode levar a anti-padrões que trarão consequências negativas a aplicação, debater sobre essas abordagens focando nos principais equívocos cometidos na estrutura trará um maior entendimento da arquitetura.

Dentro da arquitetura podemos encontrar diversas tecnologias seguindo as formas do padrão estabelecido e um ponto crucial a ser respondido são os padrões de construção das APIs por trás dos nossos serviços e suas aplicabilidades dentro da arquitetura. As arquiteturas das *APIs* mais conhecidas *SOAP*, *GraphQL*, *REST* e *gRPC* possuem vantagens e desvantagens e para além disso temos abordagens que podem ser adotadas dentro de nossa arquitetura de microsserviços onde sair do padrão *REST* é um caminho vantajoso.

1.2 HIPÓTESES

Esse trabalho tem como hipótese que uma arquitetura baseada em microsserviços possui estabilidade, escalabilidade e segurança, a partir de suas características, como por exemplo:

- Sistema composto por serviços;
- Independência tecnológica entre os serviços;

- Entrega contínua sem grandes riscos de impactar áreas não envolvidas com a alteração feita no sistema;

1.3 OBJETIVO

O objetivo deste projeto é através de pesquisa dos padrões e anti-padrões de microsserviços, trazer informações compiladas sobre sua aplicabilidade e discorrer sobre os detalhes da arquitetura e a sua abordagem em C#.

Alinhado também a este objetivo, convém a análise das propostas da Microsoft para a arquitetura acompanhando os padrões de desenvolvimento apontados como referência na elaboração do modelo dentro do *framework,* também outras ferramentas adjacentes ao desenvolvimento na plataforma.

Outro objetivo é a análise do resultado da utilização de outros padrões de arquitetura de API como *SOAP*, *GraphQL*, *gRPC* e *REST* para elaboração de soluções dentro dos padrões de microsserviços.

1.4 JUSTIFICATIVA

Tendo em vista as arquiteturas já consolidadas no mercado e as necessidades de crescimento das aplicações corporativas é de grande importância o conhecimento da arquitetura de microsserviços e para quem já está inserido na plataforma de desenvolvimento da Microsoft utilizando C# conhecer os padrões e as ferramentas utilizadas na elaboração desta arquitetura é fundamental. Também conhecer os principais anti-padrões do desenvolvimento da arquitetura nos permite evitá-los e assim fazer as escolhas certas durante o desenvolvimento. Conhecer outras formas de construir os serviços representados por *APIs* utilizando outras arquiteturas além da *REST* no contexto correto traz um leque maior de possibilidades para construção de uma aplicação melhor.

O valor científico deste projeto está em ajudar a propor soluções práticas aplicáveis no mercado para uma aplicação mais estável e resiliente, propondo as melhores práticas dentro do dotnet para implantação de APIs e a aplicabilidade delas dentro de aplicações que utilizam a arquitetura de microsserviços.

1.5 METODOLOGIA DA PESQUISA

Através da documentação dos padrões de desenvolvimento com arquitetura de microsserviços serão elaborados rascunhos arquiteturais de casos reais de negócios fazendo menção a abordagem que podemos utilizar na construção da *API* seja utilizando arquitetura *GraphQL*, *gRPC* ou *REST* com experimentações comparando o comportamento, desempenho, manutenibilidade, custo e complexidade.

2 MICROSSERVIÇOS

A arquitetura de microsserviços é um estilo de arquitetura de sistemas onde diversos serviços compõem sua estrutura. Seu conceito possui semelhanças ao apresentado por outra arquitetura mais antiga, a Arquitetura Orientada a Serviços (*SOA)*, com a diferença que a arquitetura de microsserviços utiliza tecnologias menos custosas como *REST* e *gRPC* e os serviços representam pedaços menores da aplicação diferentemente da arquitetura *SOA* (RICHARDSON, 2018).

Os serviços dentro da arquitetura de microsserviços são fracamente acoplados, ou seja, cada serviço dentro dela funciona independentemente de outras partes da aplicação exprimindo uma necessidade do sistema. De acordo com as requisições da arquitetura o desacoplamento dos serviços deve ser tão forte que cada serviço tem seu próprio banco de dados garantindo, em tempo de execução, que um serviço não bloqueie o funcionamento de outro, por mau desempenho ou por ter causado algum bloqueio no banco de dados, por exemplo (RICHARDSON, 2018).

O desacoplamento através de serviços possibilita a facilitação da implantação de grandes aplicações e entregas contínuas da aplicação como proposto nas boas práticas da metodologia *DevOps* (Microsoft, 2021). Como os serviços são um pequeno pedaço da aplicação, sua manutenção e até mesmo experimentação de novas tecnologias são facilmente elaboradas sem impactar o sistema e outros times trabalhando em conjunto com estas atividades.

Construir um sistema dentro do estilo de arquitetura de microsserviços requer atender aos seus padrões, que serão discutidos nos tópicos seguintes, primeiramente para a construção de uma aplicação nesta arquitetura vamos ter que refletir sobre os serviços que irão suprir as necessidades do sistema. Nesta etapa alguns conceitos são muito importantes e iremos discorrer sobre eles ao longo deste capítulo.

2.1 SERVIÇOS

A primeira etapa na construção de uma arquitetura em microsserviços é a decomposição da aplicação em serviços (JR; SCHMELMER, 2016), mas primeiramente devemos entender melhor sobre a essência de um serviço no contexto da arquitetura. Para tal, podemos abordar a seguinte definição, Richardson (2019, p 41) "*A service is a standalone, independently deployable software component that implements some useful funtionality.*", desta maneira podemos entender que um serviço é um componente independente que deve implementar funcionalidades que não apenas sejam úteis, mas que também possam ser moduladas e encapsuladas dentro do componente.

A arquitetura requer que o serviço seja expresso através de uma *API*, estas *APIs* possuem sua própria arquitetura e pode abordar tecnologias próprias independente das tecnologias abordadas no restante da aplicação (RICHARDSON, 2018), como veremos na sessão 3 arquiteturas que podem ser empregadas nas *APIs*.

Uma *API* é uma interface que possibilita a comunicação entre múltiplos componentes em um sistema ou até mesmo com outros sistemas exteriores (BUNA, 2021).

A *API* tem que ser capaz de ser lançada independentemente do restante da aplicação e o serviço deve conter seu próprio banco de dados, que não deve ser compartilhável com os demais serviços da aplicação (TORRE; WAGNER; ROUSOS, 2021). Isso evita a quebra da aplicação por causa de um mal funcionamento de um serviço específico que possa impactar na queda do banco de dados. Isso também facilita a dinâmica do desenvolvimento de alterações no banco de dados visto que o time responsável pelo serviço não teria que ter a preocupação de consultar outros times responsáveis pelos demais serviços da aplicação para realizar esta operação.

Estas restrições trazidas pela arquitetura para os serviços possibilitam o alto desacoplamento que debatemos aqui e esta é a principal característica que deve ser seguida na elaboração de um serviço dentro da arquitetura.

A comunicação entre os serviços pode ser feita tanto com a utilização de *APIs* como também utilizando mensageria. Sendo assim dentro de um serviço temos operações características, como comandos, consultas e eventos. Os comandos são responsáveis por realizar alterações, enquanto as consultas obtêm informações e os eventos são responsáveis por, por exemplo, ecoar a ação realizada, por um comando, nos outros serviços da aplicação utilizando, por exemplo, um *event bus* que é um artefato de *software* onde as aplicações conseguem comunicar com as demais aplicações do sistema através de notificações que são recebidas pelas demais aplicações caso endereçado a elas (RICHARDSON, 2018).

2.2 IDENTIFICANDO OS SERVIÇOS DENTRO DA APLICAÇÃO

Para definir os serviços de nossa aplicação temos que nos lembrar que não existe apenas uma forma de se chegar a uma abstração correta do sistema, sendo assim, não existe uma receita para chegar a esse resultado.

Estes serviços são compostos de operações, cujas quais são invocadas pelos clientes que a consomem ou outros serviços acessando-as através de eventos. Estas operações são nomeadas e possuem parâmetros que devem ser passados ao consumi-las e retornam alguma informação (TORRE; WAGNER; ROUSOS, 2021).

Duas estratégias tomam destaque para destilar a aplicação em serviços, uma delas é através dos contextos obtidos na modelação de domínio utilizando os conceitos de modelagem do *DDD* (*Domain Driven Design*) que é um padrão utilizado para arquitetura do sistema de acordo com o negócio que desejamos abstrair. Outra estratégia é através das capacidades de negócio estabelecer os serviços da aplicação, onde cada caso de uso é interpretado como um serviço (TORRE; WAGNER; ROUSOS, 2021).

Veremos na sequência a implementação destas duas metodologias.

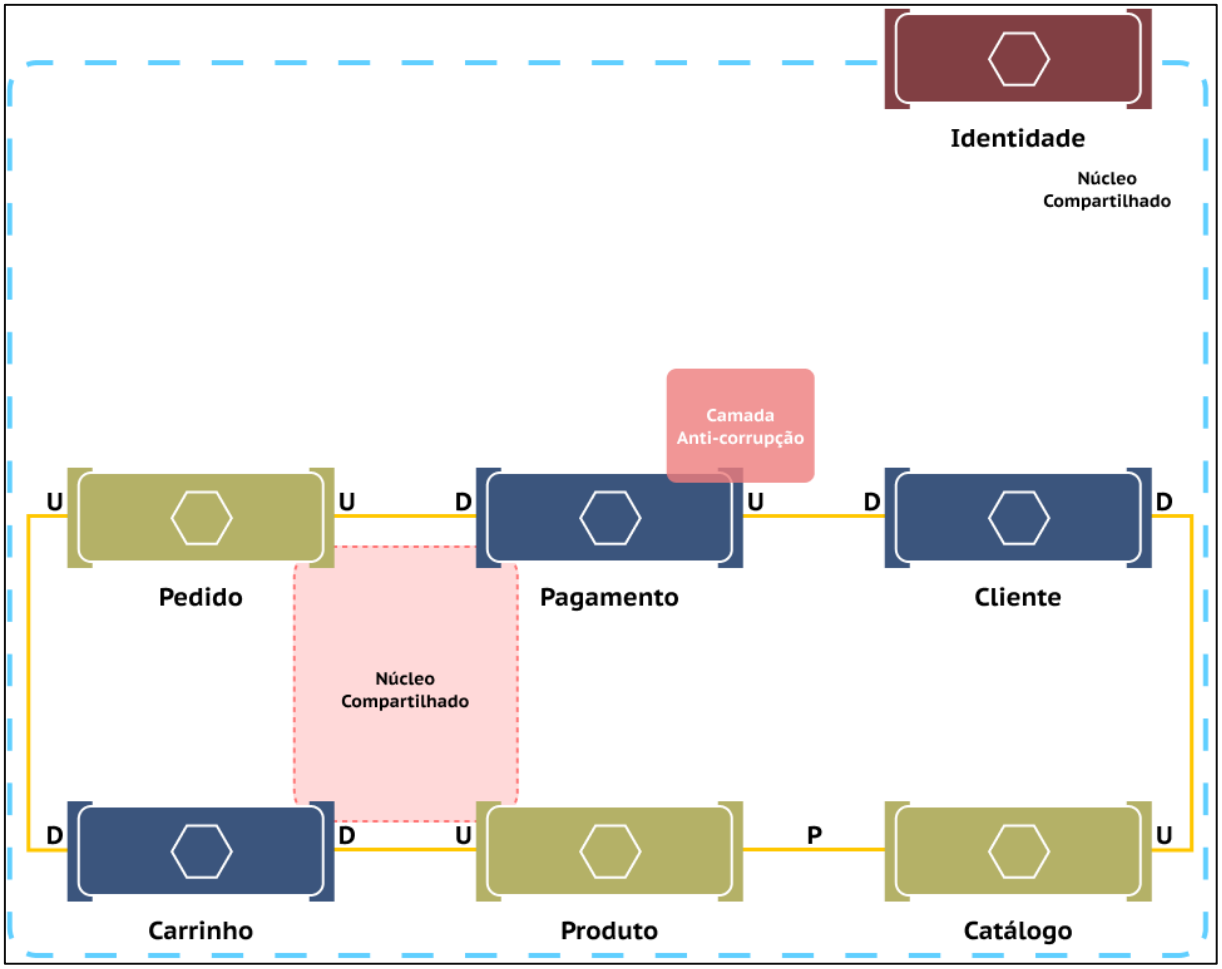
2.2.1 Identificação através do contexto delimitado

A modelagem através do *DDD* colocar em prática as exigências de seus padrões. Para isto devemos entender o domínio da aplicação seguindo uma série de etapas repassadas aqui com o objetivo de obter o domínio e especificar o serviço através dos contextos delimitados obtidos. Para tal, assim como disse Evans (2003, p 49) "*The model focuses requirements analysis. It intimately interacts with programming and design.*", devemos salientar a importância da análise de requisitos para expor através do sistema as necessidades do negócio, para tal utilizaremos a seguir alguns conceitos trazidos da obra de Erick Evans, *Domain-Driven Design: Tacking Complexity in heart of Software*.

Para demonstrar este processo, vamos utilizar como exemplo um contexto de estudo de um sistema para uma loja virtual, e para elaborar os serviços da aplicação vamos utilizar a abordagem com os contextos delimitados da aplicação através do mapa contextual exibido na Figura 1.

.

Figura 1 - Mapa contextual aplicação de loja virtual, os contextos em amarelo representam os domínios principais, enquanto os em azul representam os domínios genéricos e o vermelho é o domínio auxiliar. Relativo as relações “U” representa upstram, “D” downstream e “P” parceiros.



Fonte: autoria própria.

Nesta relação cada contexto se transformaria em um serviço, sendo assim teríamos um serviço responsável por pedido, pagamento, cliente, carrinho, produto, catálogo e identidade.

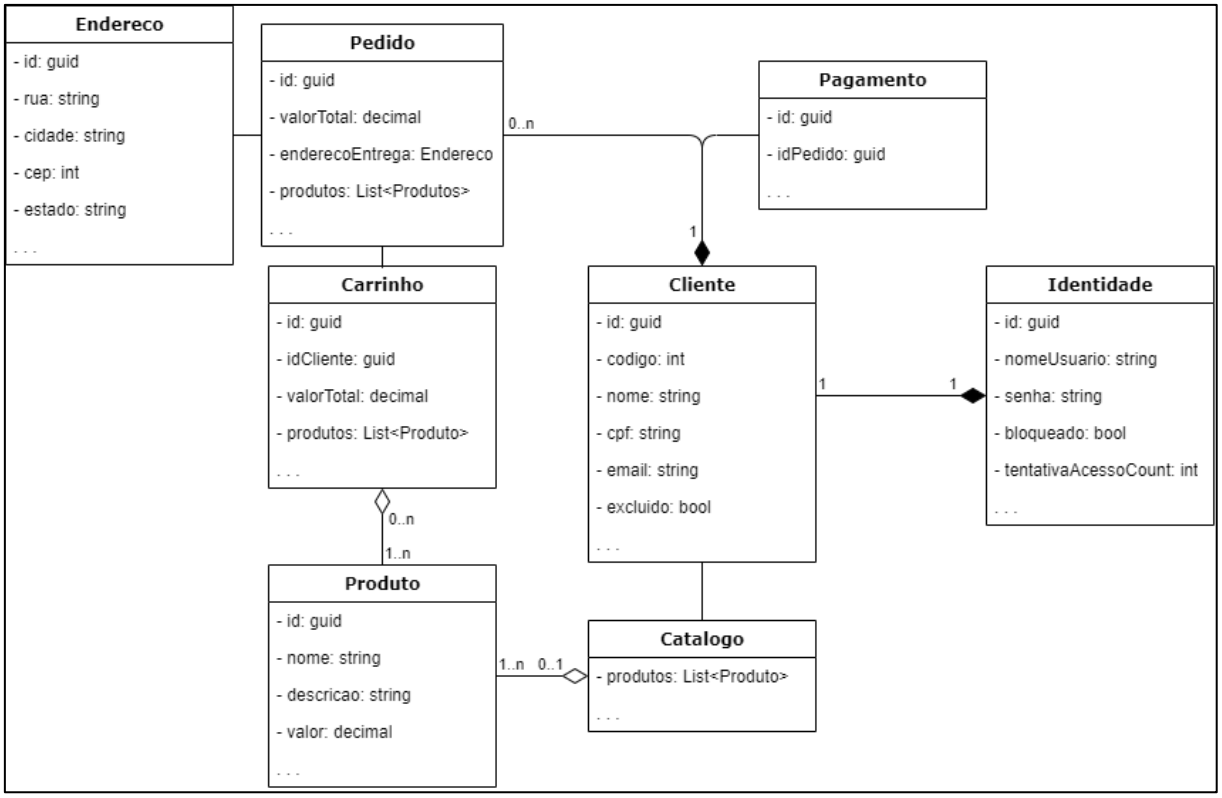
Para chegar ao mapa contextual da aplicação é necessário conversar com o *domain expert* e documentar as histórias relatadas, a partir destas histórias observamos os substantivos para obter as entidades da aplicação e então delimitar os contextos (EVANS, 2003). Por exemplo, foi extraído o modelo da figura anterior a partir da seguinte história:

- Como cliente quero navegar na loja virtual e visualizar o catálogo de produtos;

- Como cliente quero acessar o produto a partir da lista de produtos no catálogo;
- Como cliente, ao abrir um produto e adicioná-lo ao carrinho, necessitarei fazer login na loja virtual;
- Como cliente, ao finalizar a compra, quero realizar o pedido a partir dos produtos no carrinho se o mesmo contiver ao menos um produto;
- Como cliente, ao finalizar pedido, devo efetuar o pagamento.

Com os requerimentos funcionais da história observamos alguns substantivos importantes que nos levam a captar entidades da aplicação, como cliente, catálogo, produtos, carrinho, pedido e pagamento.

Figura 2 – Algumas classes chave em esboço de diagrama de classes.



Fonte: autoria própria.

Após identificar as classes chave da aplicação podemos, a partir dos requisitos funcionais, identificar também as operações do sistema. Estas operações sistêmicas são manipuladas pela aplicação e identificá-las nos ajuda a entender os *endpoints* de nossa *API* e para descrevê-las devemos observar os verbos do requisito funcional e então anotar os comportamentos relevantes da ação (EVANS, 2003).

Endpoints são os pontos de acesso disponibilizados em nossa *API*, estes pontos delimitam como nossa requisição devem ser codificadas e quais parâmetros de cabeçalho e corpo são requiridos. Uma *API* pode ser composta de diversos *endpoints* que são as portas de entrada das requisições feitas pelos clientes que pode tanto ser uma aplicação do frontend, por exemplo um aplicativo *mobile*, quanto outro serviço no *backend* como a *API* de um terceiro.

Podemos identificar dois tipos de operações *command* e *queries*. Os commands ou comandos, são responsáveis por quase todas as responsabilidades que temos em um *CRUD* menos a leitura, sendo assim fazemos a criação, atualização e deleção de dados através deles. Já as *queries* ou buscas são responsáveis apenas pela leitura (EVANS, 2003).

Tabela 1 - Comandos da aplicação obtidos a partir dos requisitos funcionais nas histórias

Ator	História	Comando	Descrição
Cliente	Cadastrar cliente	cadastrar()	Cadastro de cliente Cria novo usuário dentro do sistema.
Cliente	Adicionar produto no carrinho	addCarrinho(produto)	Adição de produto no carrinho Adiciona um novo produto dentro do carrinho.
Cliente	Remover produto carrinho	remCarrinho(produtoId)	Remove produto do carrinho Remove produto existente no carrinho
Cliente	Fazer pedido	anotarPedido()	Fazer pedido Cria um pedido de acordo com produtos no carrinho
Cliente	Fazer pagamento	pagar()	Fazer pagamento Faz o pagamento do pedido em aberto

Fonte: Richardson (2018).

Podemos especificar cada comando a partir de seu comportamento, é uma boa prática observar as pré-condições e pós-condições que o comando implementa também é importante observar seu retorno, caso exista.

Tabela 2 - Descrição dos comandos

Operação	Retorno	Pré-condições	Pós-condições
Cadastrar cliente	idCliente	Não deve existir no sistema; Informar CPF válido; Informar senha; . . .	Um novo cliente é cadastrado.
Adicionar produto carrinho		O cliente deve estar autenticado no sistema; O produto deve possuir quantidade em estoque; . . .	Adiciona produto no carrinho; . . .

Fonte: Richardson (2018).

Tão importante quanto os *commands* as *queries* são necessárias para obter as informações que preencherão as telas do sistema.

Tabela 3 - Mapeando queries através dos requisitos funcionais.

Cenário	*Query*	Retorno
Cliente acessa página de catálogo	buscarProdutos()	Retorna os produtos para o cliente.
Cliente abre produto	buscarProduto(idProduto)	Retorna dados do produto.

Fonte: Richardson (2018).

Através destes itens conseguimos identificar os contextos delimitados da aplicação e esboçar nosso mapa contextual.

2.2.2 Identificação através da capacidade de negócios

A capacidade de negócios é determinada através das atribuições funcionais da empresa para realizar atividades; elas são imutáveis ou raramente sofrem mudanças, pois focam no que é o negócio da empresa e não em como é feito (RICHARDSON, 2018).

De acordo com o caso utilizado anteriormente podemos delimitar algumas capacidades de negócio.

- Gerenciamento de clientes
- Gerenciamento de produtos
- Gerenciamento de pedidos
- Gerenciamento de pagamentos

Estas são algumas das capacidades de negócio da aplicação, cada uma delas é transformada em uma *API*, porém conforme a necessidade de negócio podemos encontrar sub capacidades de negócio fazendo a composição de uma capacidade de negócio e caso sendo mais de uma o ideal pode ser dividir em mais serviços que por sua vez são representados em *APIs*.

2.3 DEFININDO *APIS*

Definir qual estilo de *API* implementar é um processo em que deve ser levado em consideração a estratégia que será utilizada, vai muito além de apenas escolher o arquivo de transição que estamos mais familiarizados *JSON*, *XML* ou *Protocol Buffers*.

A estratégia por trás destes estilos nos permite implementar *APIs* para obter os mesmos dados de maneiras diferentes que advêm com o contexto da implementação, e os padrões para aplicações destes estilos serão discutidos a frente, mas antes de tratarmos sobre estes assuntos é importante discorrer sobre as características de uma *API* dentro da arquitetura de microsserviços.

Porém, como descrito na obra *Microsservices from day one* de *Jr. Carneiro, Cloves; Schmelmer Tim (2016, p.40),* "*In our experience, organizations that let each developer and/or development team build APIs as they see fit will end up with a ton of inconsistencies in inputs, outputs, error treatment, and response envelopes.*", ou seja, escolher um padrão para as *APIs* dos nossos serviços é fundamental para manter a coerência para os clientes que as consomem.

Implementar uma interface (*Interface Definition Language – IDL*) para cada *API* é uma regra válida para qualquer estilo de *API* que seja implementada dentro da arquitetura, sendo o padrão *REST* o mais comum nas arquiteturas, porém assim como o *SOAP,* por ser um padrão antigo, não contamos com uma funcionalidade nativa para o *pattern*, mas contamos com o projeto *Open API Specification* que é uma evolução do projeto Swagger, que por sua vez conta com uma série de funcionalidades para documentar *APIs REST.*

Cada serviço será representado em uma *API*, conforme já falamos anteriormente, e cada *API* será consumida por algum cliente que no caso de uma requisição síncrona deverá aguardar o retorno da *API* de serviço para dar uma ação ao usuário e nesses casos na arquitetura é recomenda a aplicação de alguns padrões como *Retry* e *Circuit breaker*. O *Circuit Breaker* consiste em quando obtivermos uma determinada quantidade de falhas consecutivas nas requisições síncronas da aplicação então a aplicação não fara mais requisições por um período de tempo pré-estabelecido, após o término deste período poderão ser feitas requisições novamente e se ocorrerem falhas consecutivas o sistema entra novamente neste período de "quarentena", este padrão é utilizado com frequência a fim de evitar falhas em cascata que podem afetar outros serviços da aplicação. O padrão *Retry* garante que caso a chamada síncrona a um *endpoint* retorne uma falha, então serão feitas novas tentativas para acessar esse *endpoint* de acordo com uma quantidade pré-estabelecida (RICHARDSON, 2018).

Basicamente nas requisições de *APIs* podemos encontrar dois padrões: assíncrono e síncrono. Requisições assíncronas são utilizadas quando o cliente não deseja receber um retorno sobre aquela ação, ou pelo menos, não imediatamente. Já em uma requisição síncrona é desejável ter uma resposta em tempo real para o cliente.

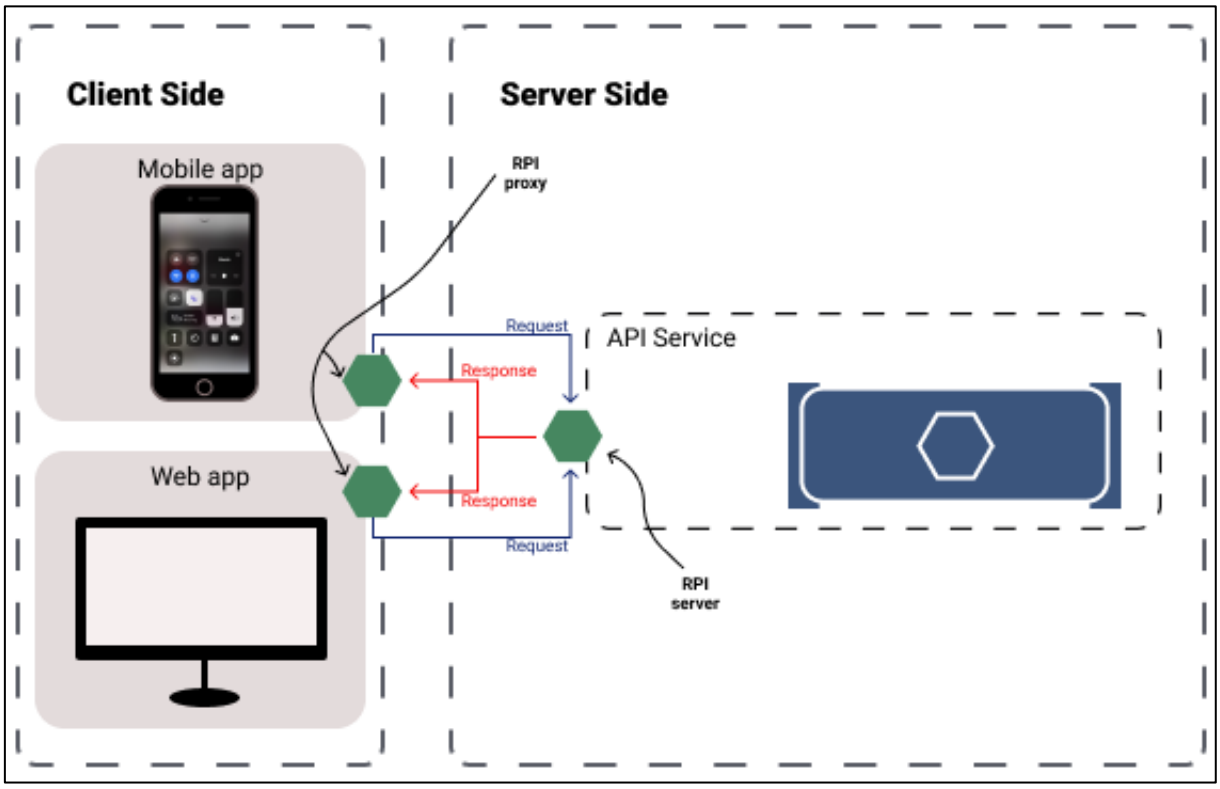
2.3.1 Comunicação síncrona

Os meios de comunicação síncronos podem ser construídos através de *APIs REST*, *gRPC*, *SOAP* e etc. Neles como dito anteriormente o cliente tem o comportamento de enviar uma requisição e aguardar uma resposta do servidor, como veremos posteriormente no capítulo de modelagem de *APIs* estas abordagens podem ser feitas de maneiras diferentes e trabalhadas com diferentes tipos de arquivos de troca de informações (TORRE; WAGNER; ROUSOS, 2021).

Como benefício esse padrão traz a facilidade de sua implementação, porém não permite uma série de abordagens assíncronas muitas vezes necessárias nas aplicações conforme descreveremos na seção 2.3.2.

Este padrão não é utilizado apenas nas comunicações do cliente com o servidor, mas pode também ser utilizado na comunicação entre *APIs* no próprio servidor como veremos no caso das *APIs Gateway* que podem fazer requisições para serviços através de uma comunicação síncrona.

Figura 3 - Comunicação síncrona de múltiplos clientes com um serviço



Fonte: autoria própria.

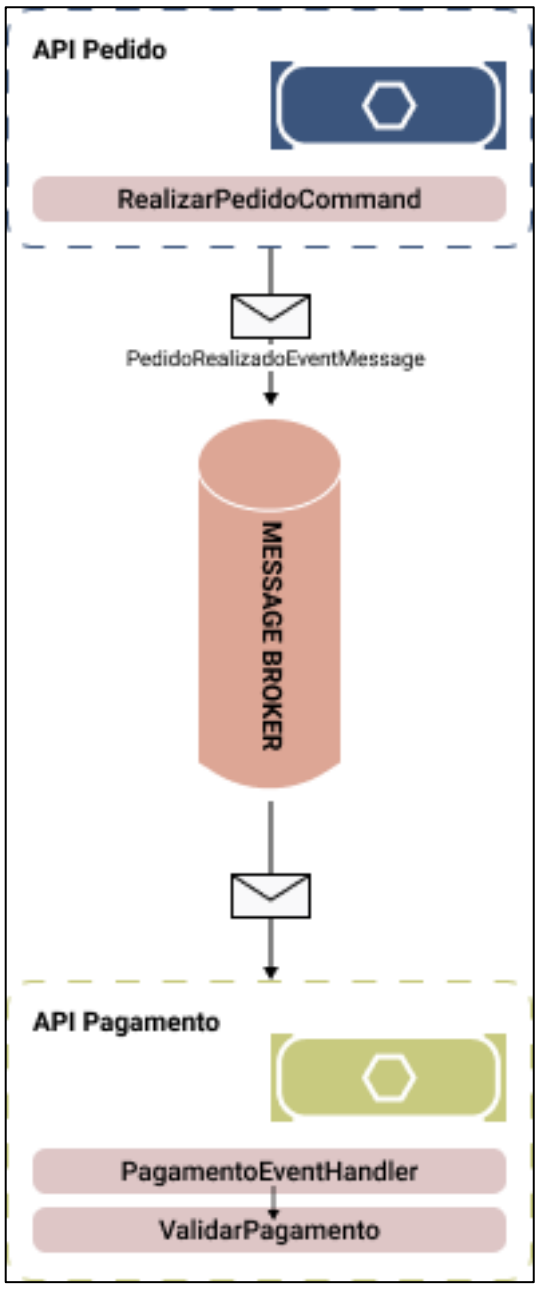
2.3.2 Comunicação assíncrona

A comunicação assíncrona remete a utilização de mensageria para este processo as aplicações que utilizam esse padrão de comunicação podem utilizar tanto *message broker* quanto uma arquitetura *brokerless*, sendo a primeira opção comumente utilizada.

As mensagens utilizadas para troca de informações, elas possuem um corpo que pode ser escrito em binário ou em algum tipo de texto formatado, e possuem um cabeçalho. No cabeçalho temos diversos pares de nome e valor que descrevem a identificação da mensagem, se haverá alguma resposta para algum endereço que a esteja escutando (RICHARDSON, 2018).

É muito comum encontrarmos comandos e eventos como tipos de mensagens; basicamente os comandos descrevem ações que desejamos realizar como uma operação, já os eventos descrevem reações que serão acionadas a partir de outras ações que os chamam (RICHARDSON, 2018).

Figura 4 - Realização de pedido chamando evento de validação de pagamento, comunicação entre APIs através de um message broker



Fonte: autoria própria.

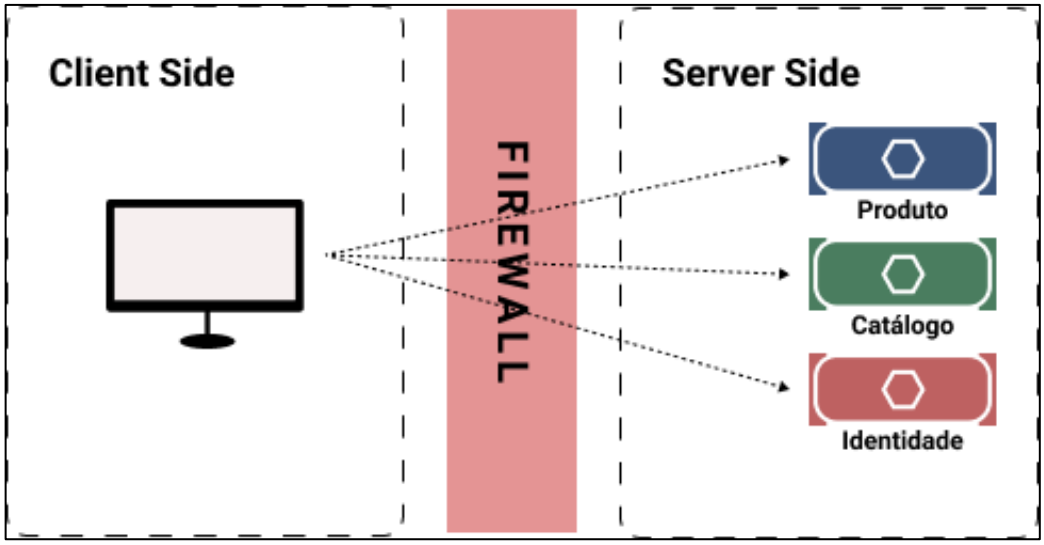
Neste exemplo o evento é chamado após a realização de pagamento e é responsável por validar o pagamento de um pedido realizado, essas situações são bem comuns também para tratamento de notificações após algum comando, nestes casos também é possível fazer a utilização de eventos.

2.4 *APIS GATEWAY*

Podemos encontrar alguns problemas nas requisições feitas para as nossas *APIs*, como existem diversas *APIs* dentro da arquitetura de microsserviços, cada uma representando um serviço todas as requisições que provém de dispositivos diferentes acabam por requisitá-las. Estes dispositivos, como por exemplo uma aplicação mobile, costuma exibir menos dados do que aplicações *web* e contam com menos poder de conexão com *internet*.

Outro problema encontrado é em relação a latência causada pelas verificações de *firewall* dos clientes que executam chamadas para os serviços que já não trazem respectivamente todas as informações necessárias, uma vez que é necessário realizar chamadas para outros serviços para obter todas as informações que trarão os dados necessários para montar uma tela.

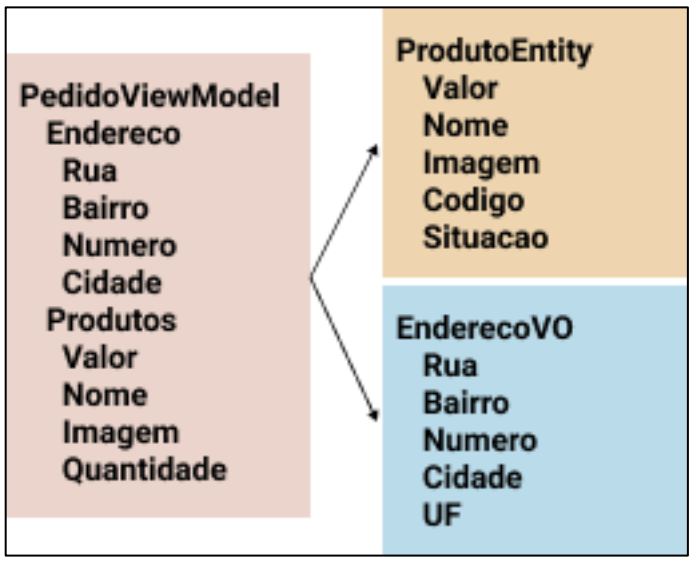
Figura 5 - Lentidão causada pelas diversas requisições para montagem de uma tela onde os dados estão divididos entre APIs de diferentes serviços



Fonte: autoria própria.

Para nos ajudar com este problema o padrão de *APIs Gateway* podem nos ajudar muito, como diz Richardson (2019, p 259) "*An API gateway is a service that's the entry point into the application from the outside world.*" para além disso as *APIs gateways* conseguem fazer a criação de *Data Transfer Objects* (DTO) que nos auxiliam a transformar as entidades que possuímos em cada *API* de serviço em um objeto composto até mesmo de dados de outras *APIs* restritos apenas ao escopo do que será exibido em tela, uma vez que as *DTOs* não são fiéis as entidades e são apenas utilizadas para fazer a transferência dos dados das *APIs* de serviço para o necessário para consumo em telas.

Figura 6 - Exemplo de dois objetos sendo misturados em um para exibição em classe



Fonte: autoria própria.

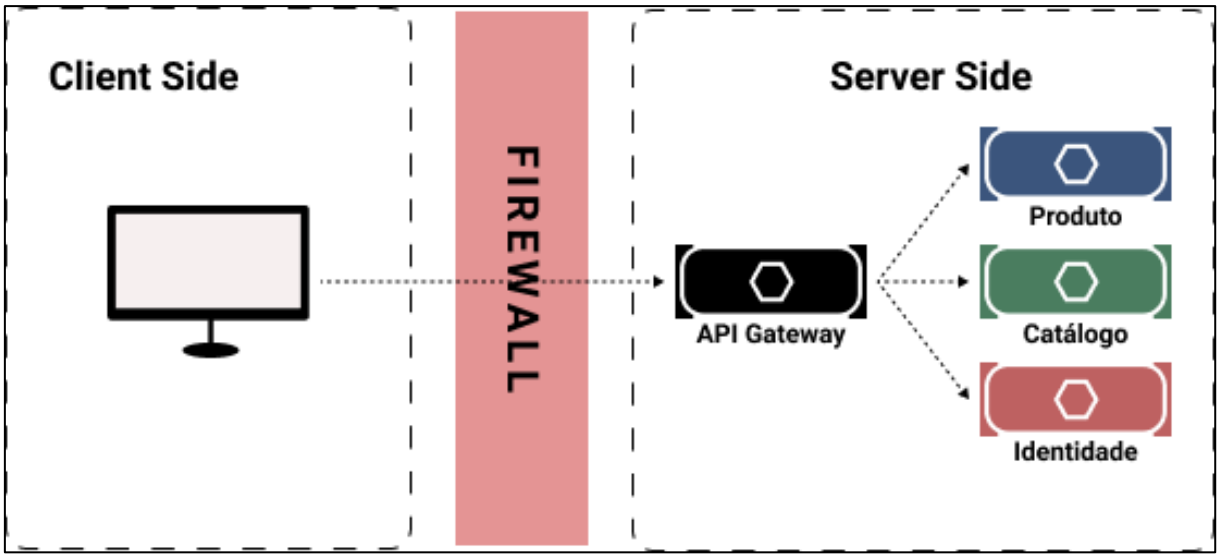
Com a *API Gateway* podemos fazer apenas uma requisição para obter os dados que serão consumidos por nossos clientes. Esta *API* será responsável por fazer as requisições para os serviços a fim de trazer junção das informações dos serviços dentro de um determinado contexto que atenda aos requisitos de negócio estabelecidos para comunicar-se com os clientes que o acessam.

Basicamente todas as requisições dos clientes chamam a *API gateway* que por sua vez é responsável por fazer o roteamento para requisições das *APIs* de serviço apropriadas fazendo a agregação dos resultados destas *APIs* e retornando estas informações para o cliente (RICHARDSON, 2018).

A operação de agregar os resultados de requisições feitas de diferentes *APIs* e retornar em um objeto único para um cliente é conhecido como composição de *APIs* e normalmente a *API gateway* pode ficar responsável pelo processo de autenticação.

É comum encontrarmos *APIs gateway* fazendo a passagem de uma requisição *REST* para *gRPC*, sendo disponibilizada de forma externa através de requisições *REST* e nas transições internas utilizar *gRPC*, mas é claro, de acordo com a necessidade de implementação.

Figura 7 - Comunicação através de API gateway



Fonte: autoria própria.

Como cada cliente necessita de quantidades de dados diferentes para a mesma operação, como o exemplo dado pelo consumo de dados de uma aplicação *web* e outra *mobile*, um padrão recomendado é criar dentro do gateway *APIs* específicas para cada cliente, assim o *subset* de dados requeridos por cada cliente pode ser atendido de acordo com a sua necessidade por uma *API* que não trará mais que o necessário.

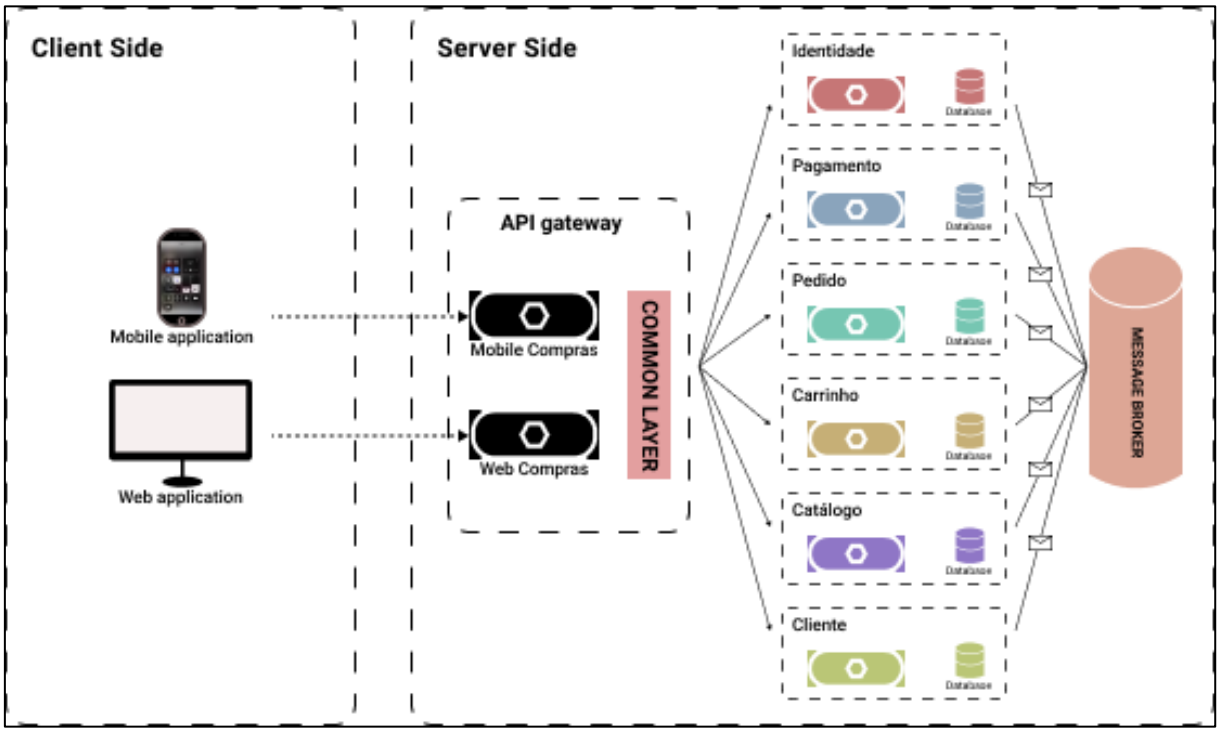
2.5 CONCLUSÃO

Como vimos a implementação de microsserviços requer a utilização de diversas práticas e no geral ela desacopla a aplicação e permite que seja possível delimitar de forma independente times para trabalhar com cada área que envolve a entrega do produto e ainda possibilita a entrega contínua da aplicação. Porém como dito no início esta abordagem requer uma série de implementações que devem ser seguidas e adotá-las requer aumento na complexidade estrutural em relação as estruturas monolíticas, por exemplo.

E como pudemos ver, uma parte muito importante da arquitetura é a construção das *APIs* que são denominadas pelos nossos serviços, pois não apenas estruturamos o nosso serviço com base na escolha da estrutura da *API,* mas também definimos a intercomunicação das *APIs*.

Para descrever o caso de exemplo de uma aplicação de loja virtual de acordo com os apontamentos que foram feitos até aqui podemos descrevê-lo da seguinte forma conforme a Figura 8.

Figura 8 - Arquitetura microsserviços loja virtual



Fonte: autoria própria.

A partir das diversas abordagens descritas até este capítulo é possível abstrair a partir da arquitetura de microsserviços a aplicação desta forma. Como descrito cada ciente tem sua própria *API gateway* a fim de trazer apenas os dados necessários para o cliente, fazendo o roteamento entre a requisição e as *APIs* de serviço alvo, também um *common layer* para aplicação de regras comuns no *gateway* como por exemplo autenticação. Cada serviço foi representado em uma *API* que possui seu próprio banco de dados e a comunicação entre os serviços é feita através do *message broker* desacoplando as *APIs* de serviço.

Assim conseguimos atender as necessidades de uma aplicação em microsserviços e agora seguimos o debate sobre a modelagem de *APIs* dentro da arquitetura de microsserviços.

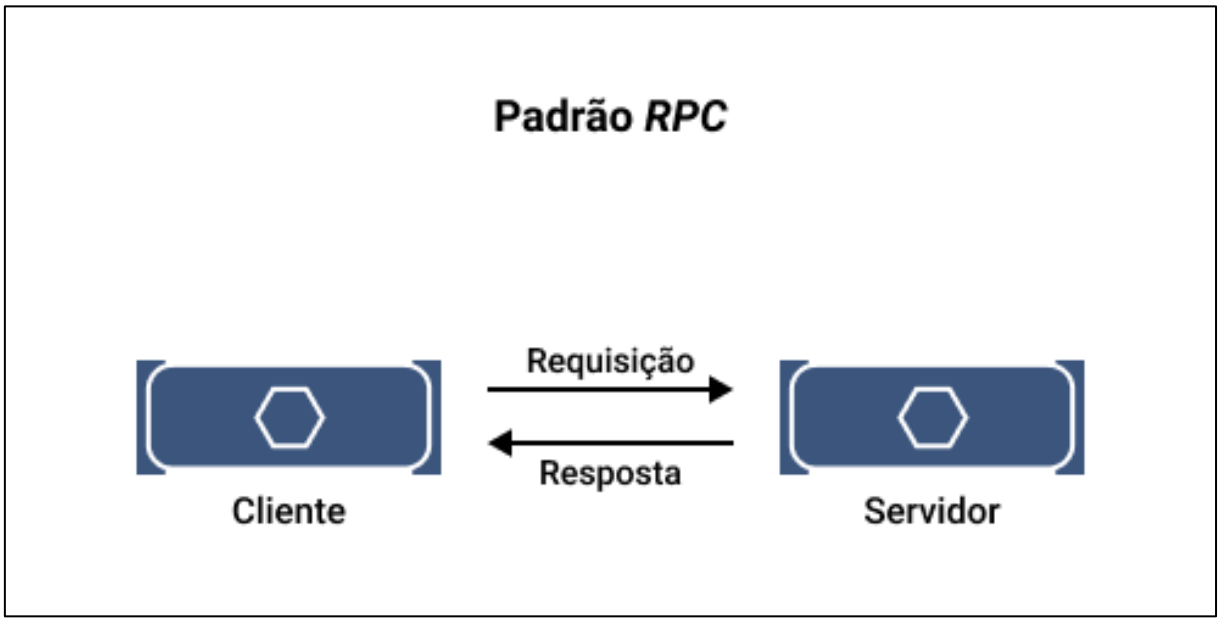
3 MODELAGEM DE *APIS* E INTERCOMUNICAÇÃO DE SERVIÇOS

A comunicação entre *APIs* pode ser feita de maneira assíncrona ou síncrona. A comunicação síncrona pressupõe a chamada de um serviço através de uma requisição de um cliente que deverá aguardar o retorno podendo ficar bloqueado até que obtenha uma resposta, já a comunicação assíncrona pressupõe um tratamento através de mensageria (RICHARDSON, 2018).

As comunicações síncronas possuem quatro principais estilos de arquiteturas de desenvolvimento *SOAP*, *REST*, *graphQL* e *gRPC*, sendo o *REST* a arquitetura mais popular (RICHARDSON, 2018).

Neste capítulo será abordado cada uma das quatro arquiteturas descrevendo o processo de implementação e motivação do uso da arquitetura, ressaltando que estas estruturas fazem comunicações através de *Remote Procedure Call* (*RPC*), ou seja, como dito no início do capítulo haverá sempre cliente enviando requisições para o servidor e aguardando uma resposta, sendo que cada arquitetura propõe uma forma de realizar estas operações e possui vantagens e desvantagens que serão discutidas posteriormente.

Figura 9 - Padrão de comunicação através de *RPC*



Fonte: autoria própria.

3.1 *REST*

Podemos definir a arquitetura *REST* a partir da afirmação do próprio criador do estilo de arquitetura FIELDING (2000, p 76) “*The design rationale behind the Web architecture can be described by an architectural style consisting of the set of constraints applied to elements within the architecture*.” onde essas restrições ou limites são aplicados sobre os clientes e serviços através de verbos que expressam ações.

A arquitetura *REST* tem como característica a utilização de verbos para delimitar as requisições sobre os serviços. Os principais verbos *HTTP* utilizados são *GET*, *POST*, *PUT*, *PATCH* e *DELETE*. A utilização destes verbos traz anotação semântica para cada função utilizada na *API* (SUBRAMANIAN; RAJ, 2019).

Tabela 4 - Descrição dos verbos *HTTP*

Verbo	Descrição
GET	Utilizado para a obter representação dos recursos.
POST	Utilizado para criação dos recursos
PUT	Utilizado para atualizar os recursos, sendo necessário o envio de todas as informações do recurso
PATCH	Utilizado para atualizar os recursos, sendo necessário apenas o envio da informação a ser atualizada
DELETE	Utilizado para a deleção de recursos

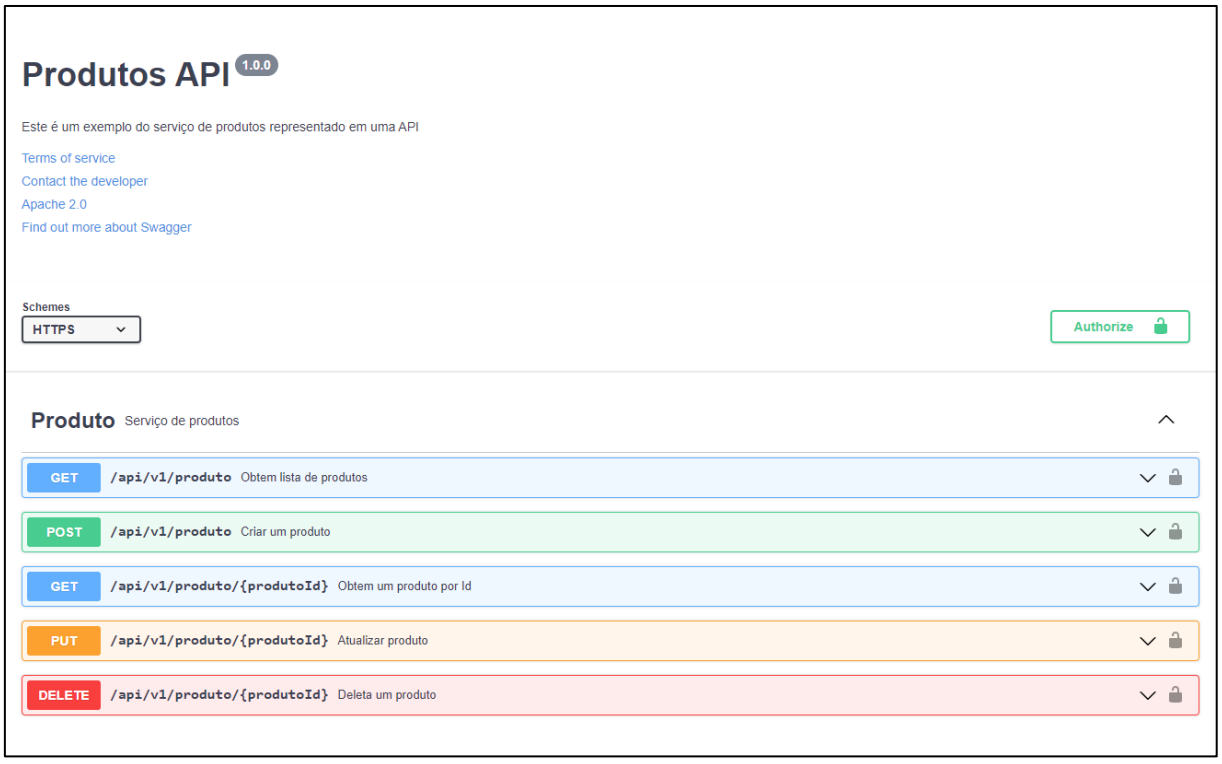
Fonte: Subramanian (2019).

Na utilização deste padrão temos alguns conceitos importantes como as *URI* e *queries*, *Uniform Resource Identifier* (*URI*) é utilizado para identificar o recurso que vai ser tratado na *API*, para identificar a função a ser executada é necessário tanto saber sua *URI* quanto o verbo da requisição. A URI segue uma série de regras de boas práticas como a não utilização de caracteres maiúsculas e separação de palavras através de hífen (SUBRAMANIAN; RAJ, 2019).

Para comunicação o tipo de mídia utilizado mais comum é o *JSON* e o *XML*, mas também encontramos a disponibilidade de outros tipos de mídia como o binário. Essas mídias são incorporadas no corpo das requisições, porém por padrão alguns conceitos são aplicados nos verbos e no corpo das requisições que utilizam estes verbos, como por exemplo as requisições com o verbo *GET* jamais devem apresentar um corpo (SUBRAMANIAN; RAJ, 2019).

Toda *API* deve ser representada através de uma interface, também conhecida como *Interface Definition Language* (*IDL*), porém o *REST* não tem uma *IDL* por padrão. Tendo este ponto em vista foram criadas *IDLs* para a representação do *REST* e a mais utilizada é a *Open API Specification*, encapsulada pela biblioteca do *Swagger* (RICHARDSON, 2018).

Figura 10 - Exemplo de representação da *API* de produtos através da *IDL swagger*

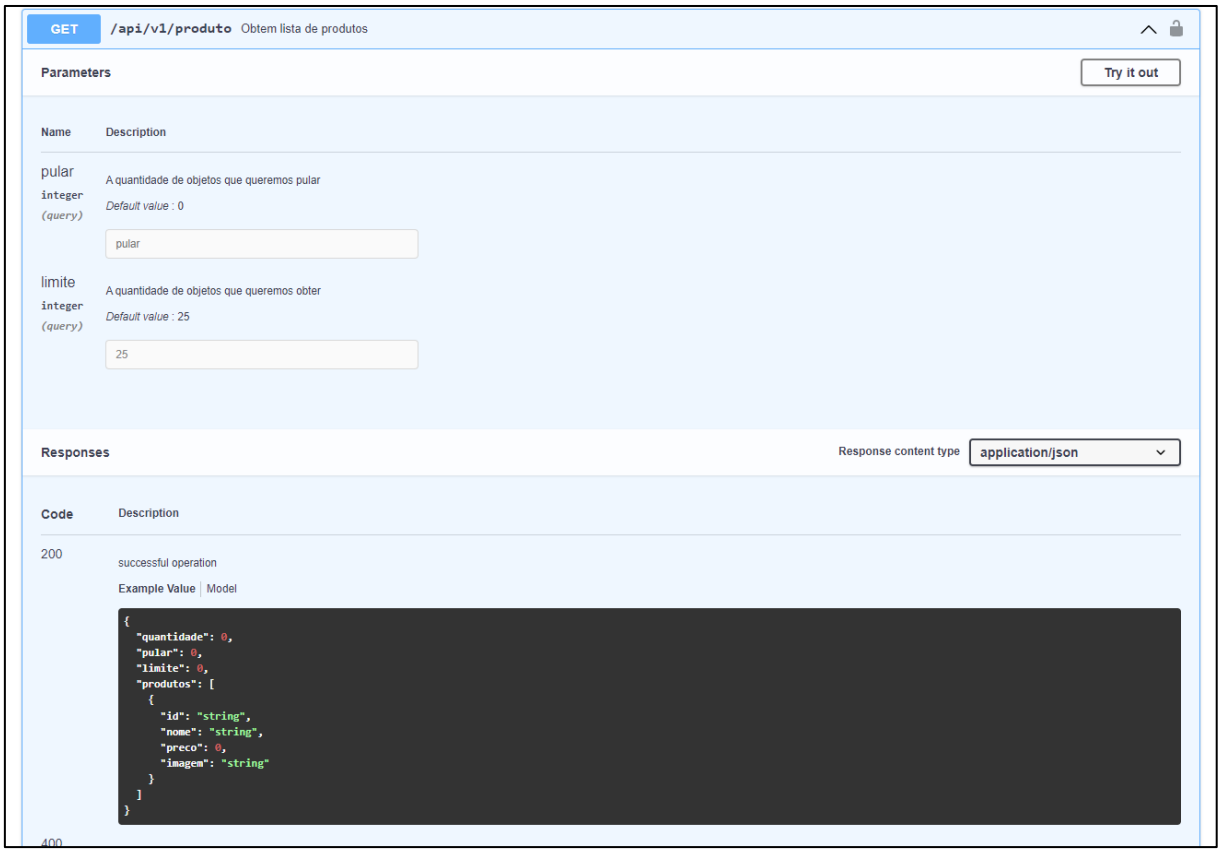


Fonte: Autoria própria.

Como podemos observar na Figura 10 temos a representação de um serviço em uma *API* no estilo *REST* é evidente a utilização dos verbos para delimitar as ações realizadas neste serviço conforme discutido anteriormente neste capítulo.

Também é notável a utilização do versionamento, que é outra boa prática na utilização de *APIs* que possibilita a evolução das *APIs* sem correr o risco de impactar aplicações que fazem determinado consumo a versão antiga da *API* que está sendo atualizada. Nos monolitos uma atualização desse tipo traria erros de compilação que seriam facilmente mapeados para que se fizesse as alterações necessária, já em um sistema com arquitetura em microsserviços esses erros são gerados apenas em tempo de execução, sendo necessário uma análise mais detalhada de cada caso de uso da aplicação que será impactado com a mudança. Por este motivo o versionamento das *APIs* é um fator extremamente importante, pois permite um crescimento de forma orgânica da aplicação evoluindo degrau por degrau em pequenos passos, porém constantes (RICHARDSON, 2018).

Figura 11 - Paginação em requisição de leitura



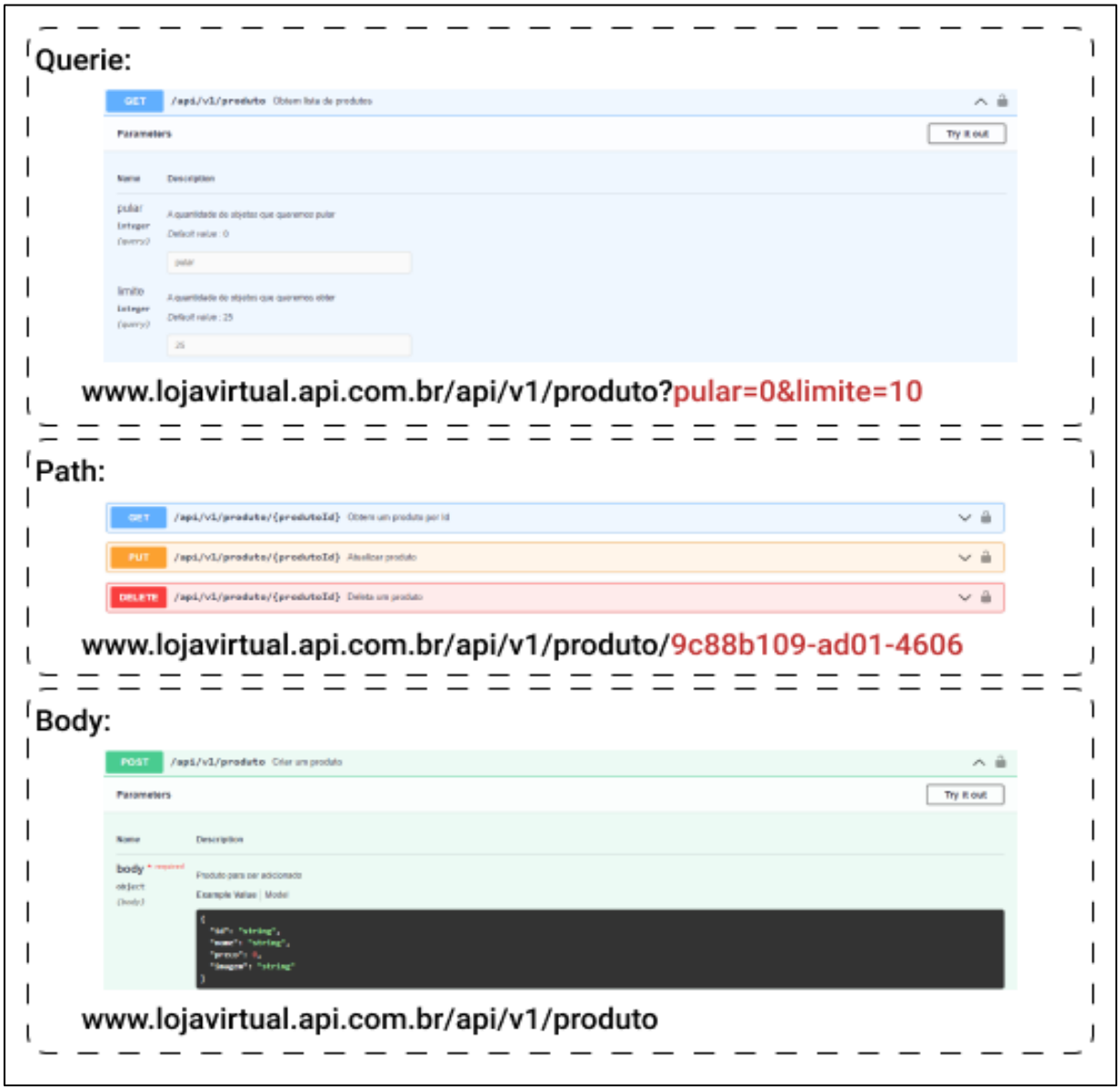
Fonte: Autoria própria.

Na Figura 11 podemos observar a utilização de outro padrão importante, que é a paginação. Através dela evitamos a transmissão massiva de dados, obtendo os dados por “fatias”.

Para paginar fazemos a utilização de parâmetros, nesse caso “pular” indica quantos registros almejamos saltar para começar a partir de determinado índice e “limite” indica a quantidade máxima de registros que almejamos obter. Assim imaginando o cenário de consumo de uma tabela com milhares de registros, aprimoramos a velocidade e a performance da aplicação como um todo, pois temos a redução da quantidade de registros utilizados a cada resposta do serviço (SUBRAMANIAN; RAJ, 2019).

Neste exemplo as *URLs* são utilizadas de diversas formas, a passagem de informações ocorre tanto através do *PATH*, quanto por *Query* e pelo *Body*.

Figura 12 - *URLs* dos *endpoints* da *API* e transmissão de dados



Fonte: Autoria própria.

Temos na Figura 12 a exposição das passagens de conteúdo na requisição, a funcionalidade do serviço no caso das *URLs* deste exemplo que utilizam o *path* para a passagem de parâmetros e não possuem body são alcançadas através do verbo da requisição, *GET*, *PUT* ou *DELETE*. Quando a requisição é feita através do *body* temos um tipo de mídia utilizado para trazer estes dados, neste caso, a mídia utilizada foi o *JSON* (SUBRAMANIAN; RAJ, 2019).

Este padrão de requisição oferece suporte para o protocolo *HTTP*/1.1, sendo que hoje temos protocolos mais vantajosos como o *HTTP*/2, sendo a principal diferença a velocidade de transição pois no protocolo *HTTP*/1.1 temos a abertura de diversas requisições e no *HTTP*/2 temos a conexão multiplex onde podemos numa mesma requisição *TCP* receber e enviar dados do cliente e do servidor, ou seja, utilizamos apenas uma única requisição durante toda a operação do cliente com o servidor (MIRANDA, 2016).

3.2 *SOAP*

Assim como o *REST* o *SOAP* é um padrão de modelagem de *APIs* que utiliza mídias não binárias, utiliza protocolo *HTTP*/1 e tem como característica a utilização de arquivos *XML* para transmissão de dados. Este padrão possui algumas camadas que podem ser descritas da seguinte forma (ENGLANDER, 2002):

- Envelope: que é caracterizada por possuir o conteúdo da mensagem e detalhes do processamento desse conteúdo;
- Regras: esta camada possui as regras para codificação dos tipos de dados customizados;
- Aplicação: esta camada especifica a aplicação do envelope e as regras de codificação de dados para as chamadas e respostas.

O envelope engloba todas as mensagens do *SOAP* em um arquivo *XML*, ele deve estar presente para que a mensagem seja válida, em resumo podemos concluir que o envelope contém os dados da mensagem da requisição *SOAP* (ENGLANDER, 2002).

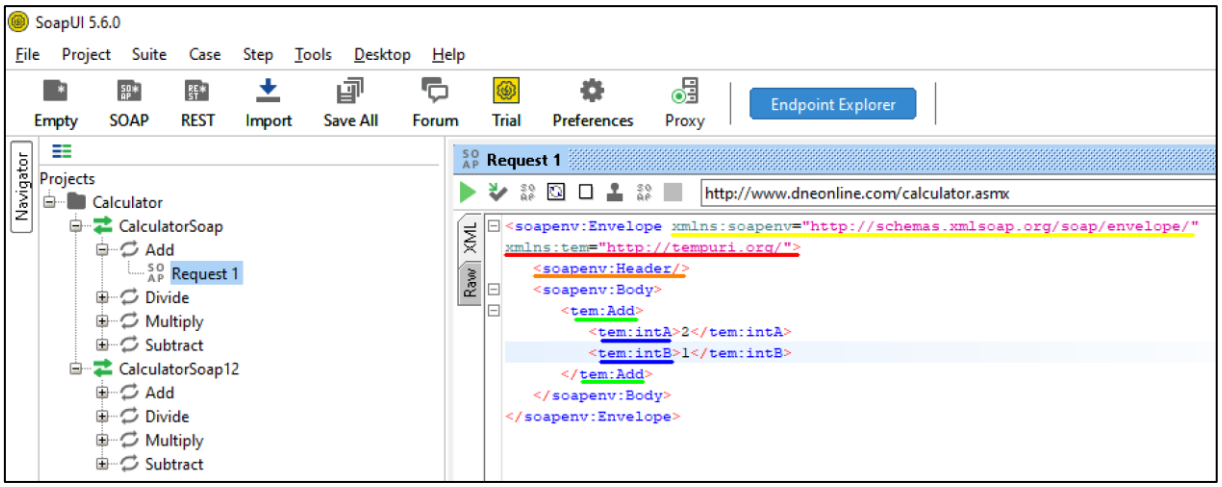
Este envelope pode conter um elemento, o *header*, que se presente deve ser o primeiro subelemento do envelope. Independentemente de haver um subelemento *header*, o elemento deve conter um subelemento *body* onde caso não haja a presença de um *header* então o *body* deve ser o primeiro subelemento do envelope. O *header* é utilizado para codificar os elementos utilizados para autenticação ou outros processos relacionados com o processamento da mensagem, esta característica é semelhante ao *REST,* porém apenas o tráfego e o formato das informações que se diferenciam (ENGLANDER, 2002).

Alguns atributos podem ser passados dentro do *header*, como a *action* que representa aplicações que a requisição almeja atingir antes de passar para a requisição final, dentro deste conseguimos passar informações que vão ser consumidas apenas neste intermediário e não serão passados para a requisição final (ENGLANDER, 2002).

O padrão *SOAP* não tem especificado uma regra padrão para serialização de dados, isso é feito através de um atributo *encodingStyle* que pode ser encontrado tanto junto da *tag* envelope quanto do *body*. Com ele conseguimos definir as regras de codificação e decodificação através da *URL* especificada no atributo, que é denominada como um namespace, que pode conter um arquivo *XML*. Um *namespace* é um mecanismo para eliminar ambiguidade entre os elementos ou atributos do XML nos ajudando a entender o contexto do elemento (ENGLANDER, 2002).

Agora veremos alguns casos de requisições a *APIs SOAP* abertas.

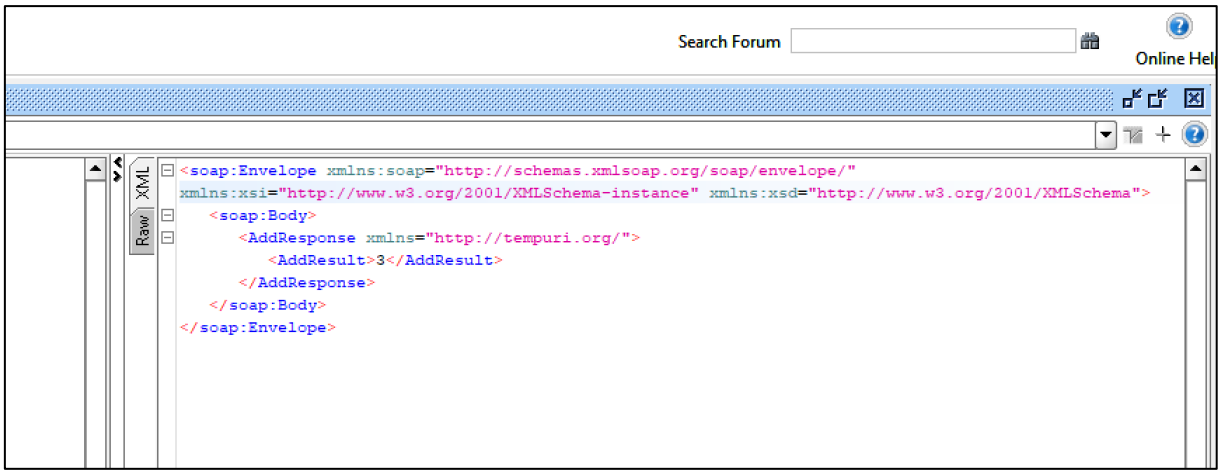
Figura 13 - Requisição para *API SOAP* calculadora, corpo da requisição no SOAPUI



Fonte: Autoria própria.

Na Figura 13 foi feita requisição para uma *API SOAP* aberta cujo endereço pode ser encontrado em http://www.dneonline.com/calculator.asmx?WSDL (data do acesso: 27/11/2021), como podemos observar em amarelo temos a definição do envelope, em vermelho podemos ver a declaração do *namespace* “tem” cujo qual corresponde a funcionalidade que será acionada, em laranja temos um header vazio, sublinhado em verde podemos ver a função que será acionada e por fim em azul verificamos as variáveis utilizadas.

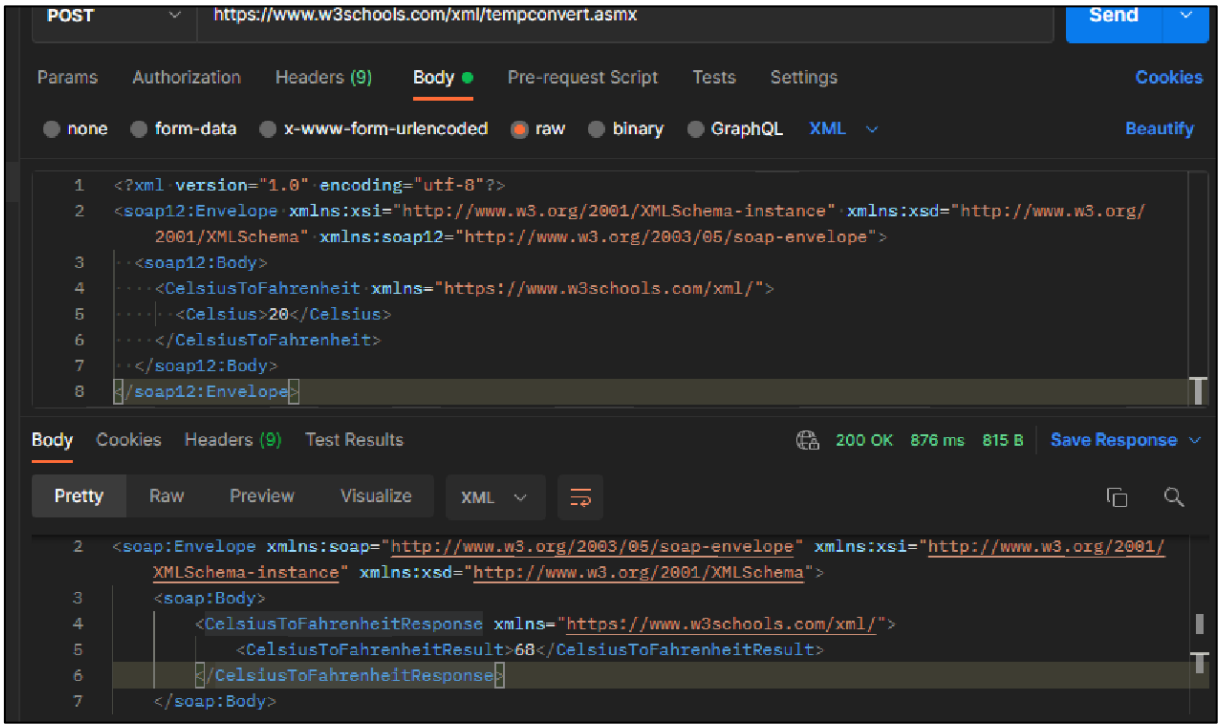
Figura 14 - Retorno da requisição *API SOAP* no SOAPUI



Fonte: Autoria própria.

Conforme podemos observar na Figura 14 obtivemos o retorno esperado da *API* sendo a operação de soma entre os números 2 e 1. A ferramenta utilizada para esta requisição foi o SoapUI em sua versão gratuita, porém também podemos utilizar outra ferramenta totalmente gratuita para fazer requisições para *APIs SOAP*, o Postman. Conforme demonstrado na Figura 15 onde foi feita requisição para *API* de conversão de temperatura Celsius para Fahrenheit.

Figura 15 - Requisição SOAP API de conversão de temperatura através do Postman



Fonte: Autoria própria.

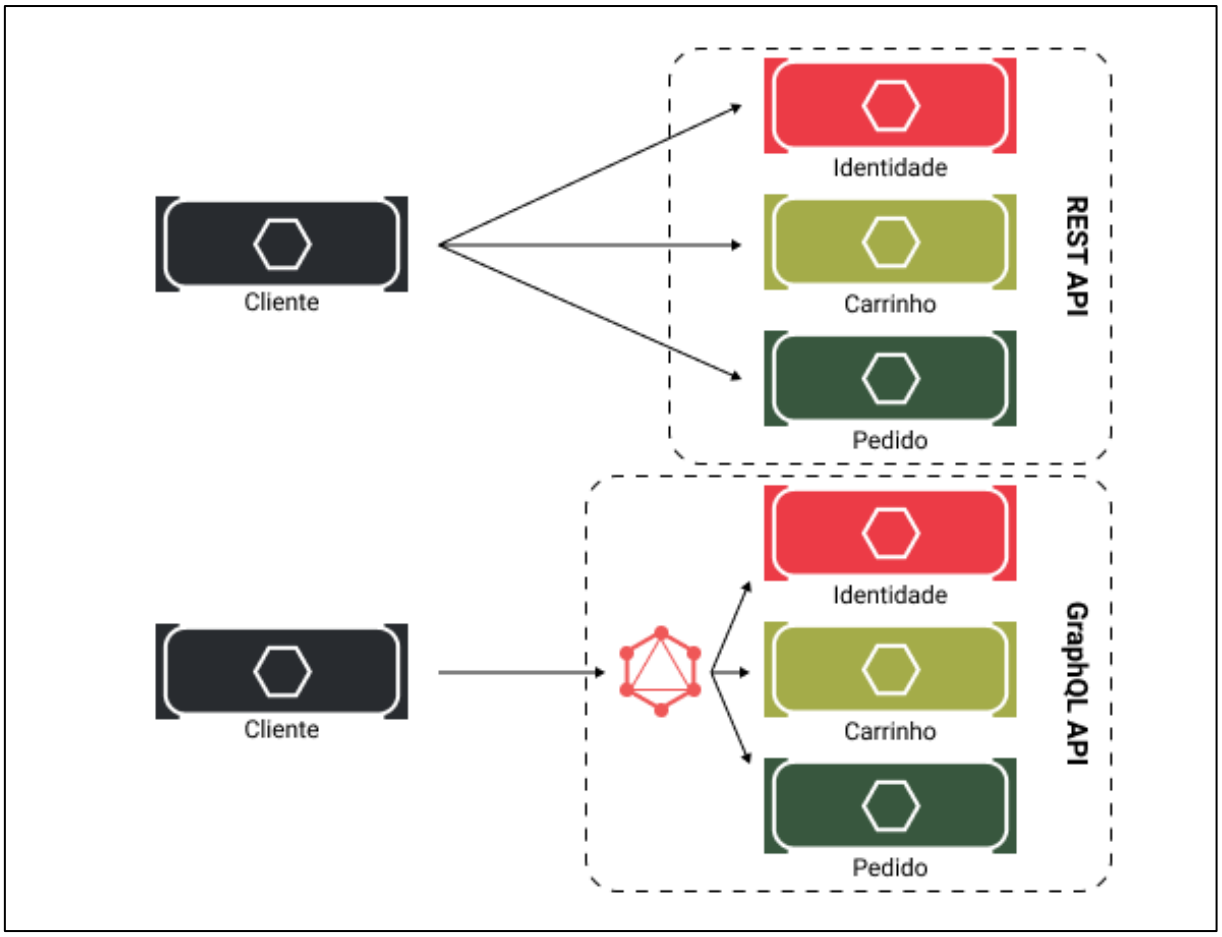
3.3 *GRAPHQL*

GraphQL nos traz outra modelagem de *API* com funcionalidade diferente dos vistos até aqui. Criado pelo Facebook em 2012 posteriormente lançado em 2015 parte do princípio da representação de dados através de uma estrutura de gráficos por isso o nome *graph*, que utiliza *query language* nas requisições para as *APIs*. A interpretação destas *queries* feitas nas requisições é traduzida em tempo de execução e processada a fim de trazer os dados requisitados. Por este motivo, *GraphQL* torna-se muito flexível e eficiente facilitando a composição de dados (BUNA, 2021).

Sendo assim com G*raphQL* não precisamos de diversos *endpoints* para fazer, por exemplo, nossas *queries* (consultas) com os dados que queremos ou então adotar a abordagem de ter apenas um *endpoint* para obter todos os dados da nossa entidade com o filtro destes dados sendo feito nos próprios clientes, o que não seria uma boa prática, pois estaria trazendo dados desnecessários para a aplicação cliente que por fim não utilizaria todos os dados obtidos pela requisição (BUNA, 2021).

No *GraphQL* o cliente pode fazer suas requisições em na linguagem *GraphQL* que permite trazer exatamente aquilo que o cliente necessita consumir (BUNA, 2021).

Figura 16 - Comunicação *GraphQL*



Fonte: Autoria própria.

O cliente envia uma requisição através de um protocolo de transporte, como o *HTTP*. A *API* em *GraphQL* interpreta a requisição e se comunica com o banco de dados para fazer exatamente o que está sendo solicitado na requisição e por fim envia uma resposta para o cliente. A mídia utilizada para comunicação geralmente é o *JSON*, assim como no *REST*, porém tanto em relação a mídia quanto ao protocolo de transporte e a base de dados no *GraphQL* temos a liberdade de utilizar a tecnologia que melhor se adeque ao nosso escopo (BUNA, 2021).

Quadro 1 - Consulta através do *SQL*

```
SELECT ID, CODIGO, DESCONTO,TOTAL
FROM PEDIDOS
WHERE PED_GID_ID = 'c50611fd-c999-4303-b9e3-e2f512a9f489'
```

Fonte: Autoria própria.

Como vemos no Quadro 1 o comando comum utilizado para consulta no *SQL* (*Structured Query Language*) possui semântica parecida com a utilizada nas consultas *GraphQL* conforme o Quadro 2, porém a sintaxe é mais simples.

Quadro 2 - Consulta através de *GraphQL*

```
{
  pedidos (id: “c50611fd-c999-4303-b9e3-e2f512a9f489”) {
    id,
    codigo,
    desconto,
    total
  }
}
```

Fonte: Buna (2021).

Além do Facebook, conhecido hoje como Meta, outras empresas como o Github estão utilizando *GraphQL*. Apesar do foco dado as consultas podemos realizar comandos dentro das *APIs GraphQL*, quando fazemos consultas chamamos de *queries* e quando estamos executando comandos chamamos esta ação de *mutations* (BUNA, 2021).

Como vimos as consultas são semelhantes no *GraphQL* com a utilização de um *Select* no *SQL* e o mesmo ocorre com as *mutations* que são semelhantes aos comandos *SQL Insert, Update e Delete*. O *GraphQL* também suporta um terceiro tipo de requisição chamado *subscription* que é usado para monitorar as entidades em tempo real como uma consulta contínua em determinada entidade (BUNA, 2021).

GraphQL utiliza tipagem forte, ou seja, seus dados devem estar tipificados. Possui *schemas* que definem todas as operações que uma *API GraphQL* realiza. As *APIs* possuem uma *IDL* nativa da arquitetura chamada *GraphiQL* e seus *schemas* são expostos através desta *IDL*, cuja qual os clientes utilizam para enviar as requisições para os serviços em *GraphQL* (BUNA, 2021).

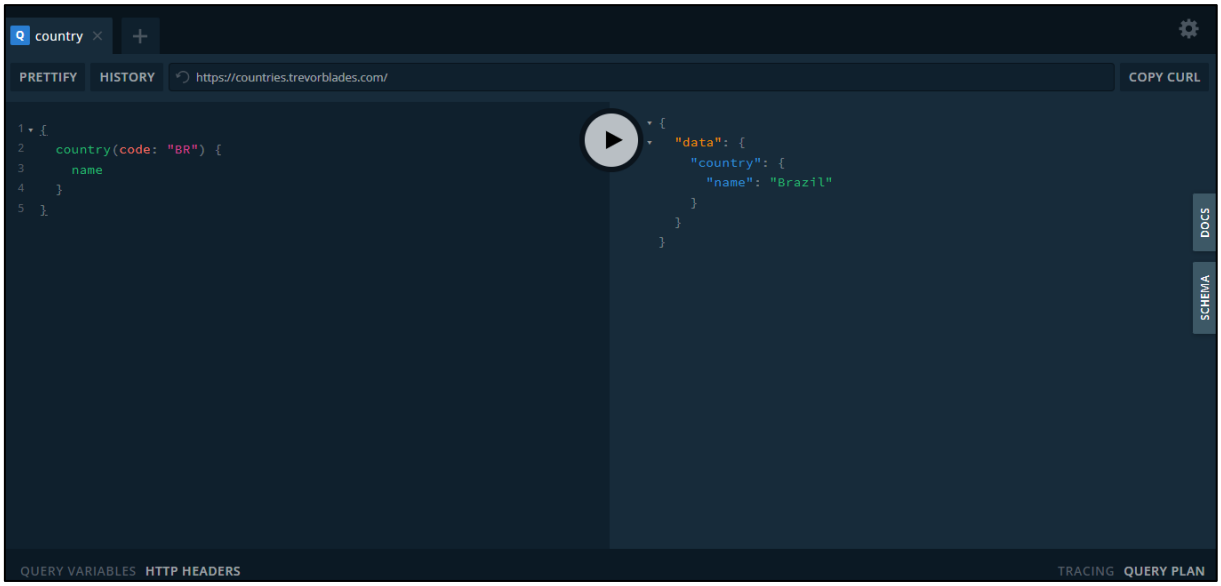
Quadro 3 - *GraphQL* exemplo de *schema*

```
type pedido (id: String!) {
  codigo: Int!
  desconto: Float
  total: Float!
  itens: [PedidoItem]
}
```

Fonte: Buna (2021)

Como vemos no Quadro 3 um *schema* possui tipagem forte definindo cada um de seus atributos. A exclamação delimita a obrigatoriedade do atributo, podemos referenciar outros tipos definidos no *schema*, como encontramos um tipo entre colchetes significa a representação de uma lista, sendo assim o atributo itens representa uma lista de PedidoItem (BUNA, 2021).

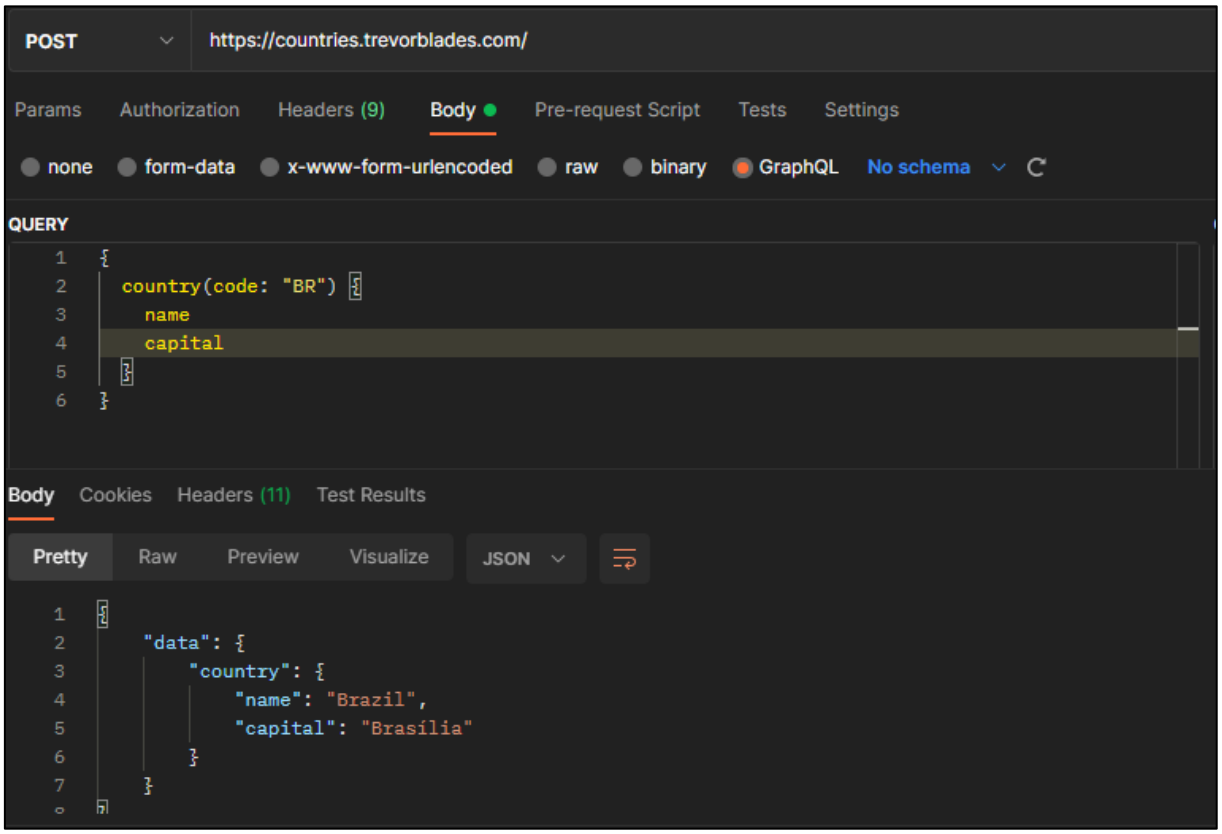
Figura 17 - Consulta *GraphQL API* aberta de países



Fonte: Autoria própria.

Na Figura 17 vemos uma requisição a uma *API* aberta para consulta de países, vemos também a utilização da *IDL GraphiQL*.

Figura 18 - Consulta *GraphQL* adicionando outro campo em requisição a *API* aberta



Fonte: Autoria própria.

Na Figura 18 vemos o dinamismo e flexibilidade das requisições *GraphQL*, sendo utilizado desta vez o Postman.

3.4 *GRPC*

O gRPC foi criado pelo Google, uma das intenções era usar protocolo *HTTP*/2 e esta é uma característica padrão desta arquitetura. Outro ponto característico é a utilização de mídia binária, tudo isto traz vantagens em termos de desempenho e velocidade, uma vez que a mídia já está em linguagem de máquina, portanto seu processamento será mais rápido. Pode ser utilizado tanto em plataformas de desenvolvimento de aplicativos, quanto em navegadores e requisições *backend* de serviços externos (RENDLE; STEINER, 2021).

As mensagens em binário utilizadas pelo padrão gRPC são estruturadas em um arquivo chamado *protocol buffer*, também chamados de *protobuf* é uma linguagem neutra criada pelo Google utilizada para serializar dados. O *protobuf* é muito mais leve que um arquivo JSON e requer menos processamento durante o processo de serialização, cujo qual é compilado através do compilador protoc que é mantido pelo Google (RENDLE; STEINER, 2021).

Quadro 4 - Demonstração de utilização de arquivo *protobuf*

```
syntax = "proto3";

option csharp_namespace = "Pedido";

import "google/protobuf/wrappers.proto"

message Pedido {
  string id = 1;
  int32 codigo = 2;
  google.protobuf.DoubleValue desconto = 3;
  double total = 4;
  repeated PedidoItem itens = 5;
}
```

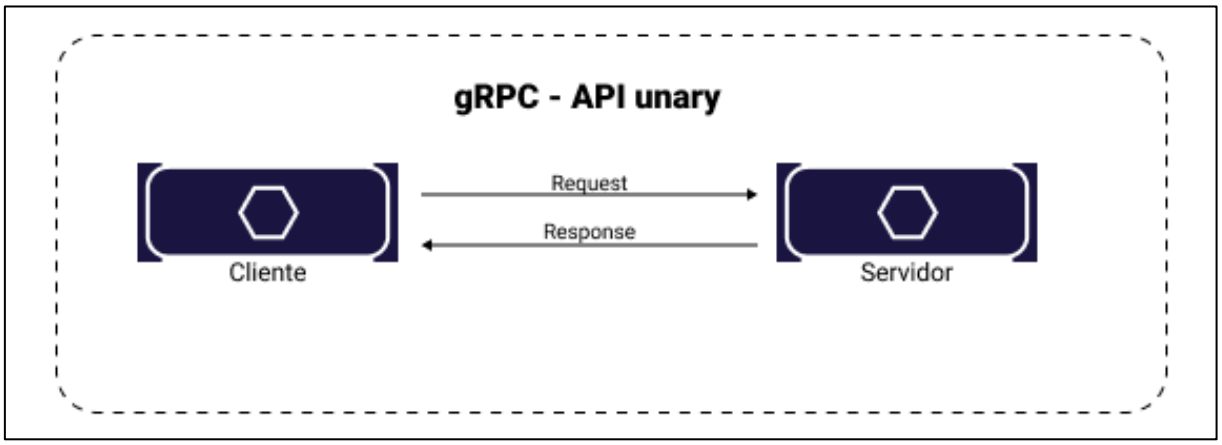
Fonte: Rendle, Steiner (2021).

Como podemos observar no Quadro 4 temos a representação da mesma entidade Pedido representada também na mídia padrão *GraphQL*, aqui notamos como ponto principal a representação de uma lista através da palavra reservada *repeated* no caso dos itens do pedido, também é possível identificar a utilização de uma extensão para representar um tipo que pode receber o valor nulo, como o atributo desconto. Outro ponto importante é a atribuição de valores inteiros aos atributos, estes valores marcam os atributos identificando os atributos para a transformação em binário (RENDLE; STEINER, 2021).

O protocolo *HTTP*/2 também foi criado pelo Goggle, utiliza por padrão mídia binária para tráfego de dados, nele a conexão *TCP* (*Transmission Control Protocol*) estabelecida no primeiro momento entre as aplicações é utilizada durante todo o tempo de vida das aplicações, ou seja, não se necessita a utilização de *requests* e *responses* em conexões diferentes, este padrão é chamado de *multiplex*. O cabeçalho da requisição é comprimido agilizando ainda mais a comunicação (MIRANDA, 2016).

As *APIs gRPC* possuem quatro principais estilos de comunicação, um deles é o unário, que segue o padrão já visto no *REST*, *SOAP* e *GraphQL*. No unário a comunicação é feita através de uma requisição e uma resposta conforme podemos observar na Figura 19 (RENDLE; STEINER, 2021).

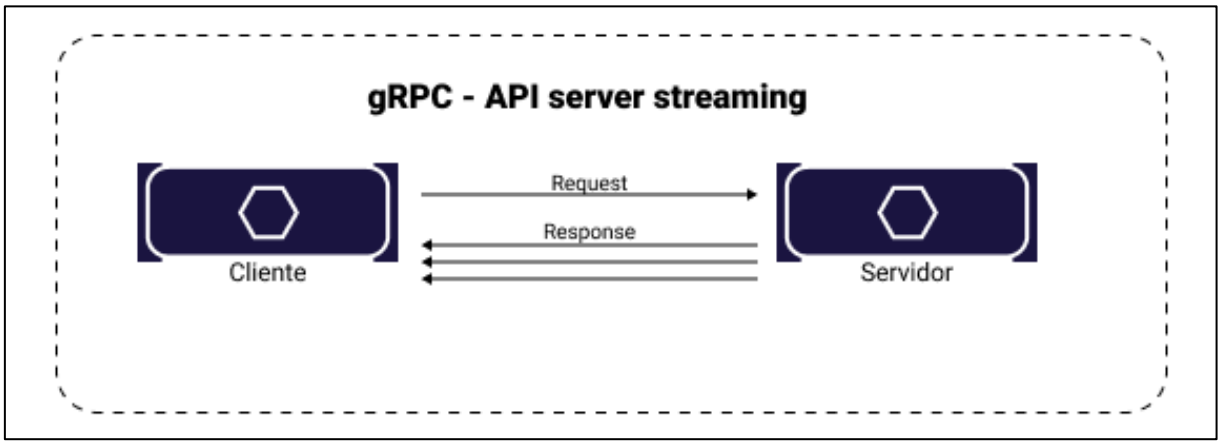
Figura 19 - Comunicação unária



Fonte: Autoria própria.

Outro estilo de comunicação é o *server streaming* este estilo de comunicação se caracteriza pela requisição do cliente receber uma requisição e então o servidor faz transmite diversas respostas ao cliente, um caso de uso para este estilo seria a necessidade de requisitar o processamento, por exemplo, do fechamento de mês de uma empresa ou outro processo que requisitasse demasiado processamento e o tempo de execução fosse demasiado demorado fazendo a necessidade de um acompanhamento assíncrono do processo. Sendo assim o servidor traria as respostas até o fim do processamento da requisição com os dados necessários para consumo do cliente, vemos este esboço na Figura 20 (RENDLE; STEINER, 2021).

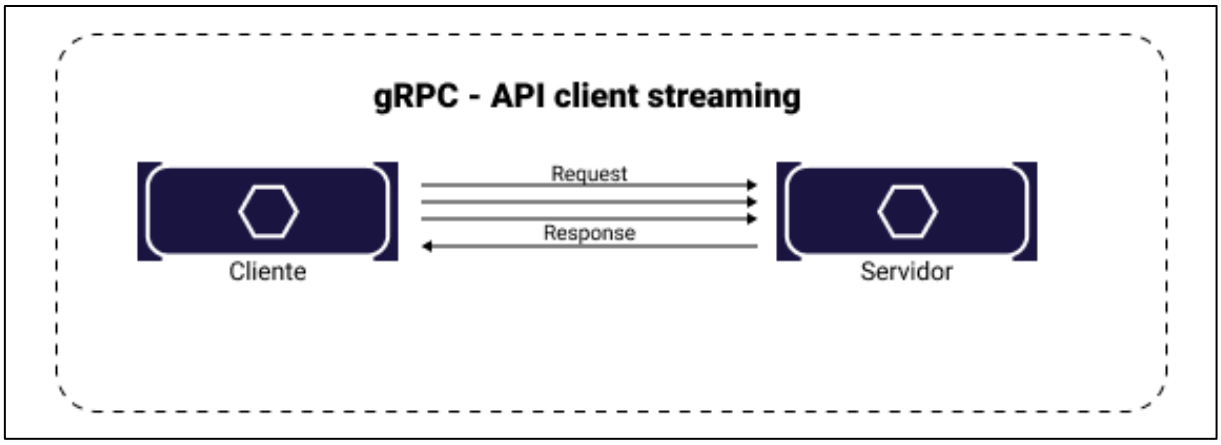
Figura 20 - Demonstração do *server streaming*



Fonte: Autoria própria.

O estilo de comunicação *client streaming* utiliza o mesmo princípio do *server streaming*, porém com a perspectiva do cliente e neste caso o cliente envia diversas requisições ao servidor que sabe aguardar o fim do envio das mensagens do cliente e enfim envia uma resposta para o cliente conforme ilustrado na Figura 21 (RENDLE; STEINER, 2021).

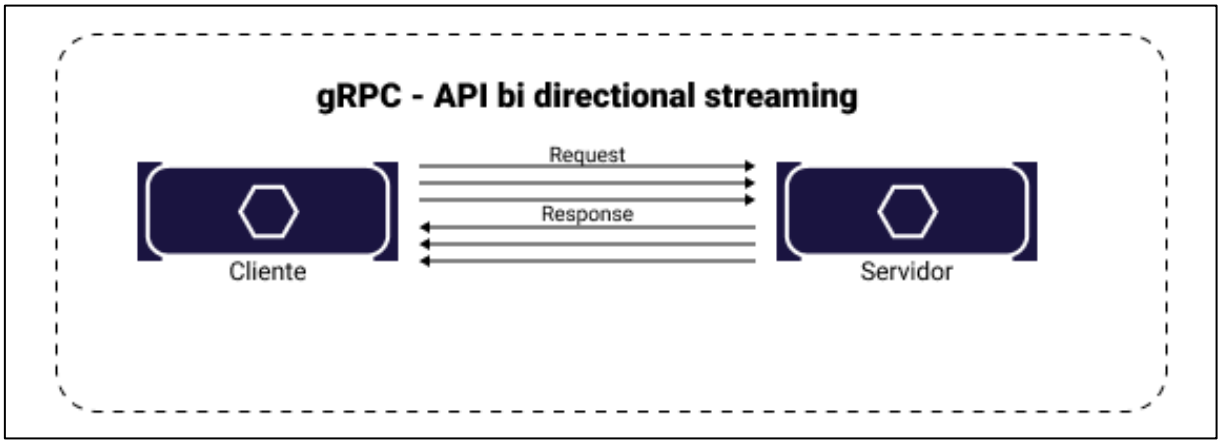
Figura 21 - Demonstração do *client streaming*



Fonte: Autoria própria.

Por último temos um híbrido do *server streaming* com o *cliente streaming* onde existe a emissão de diversas requisições do cliente e após a confirmação do recebimento de todas as requisições o serviço retorna diversas respostas para o cliente até que o processo seja encerrado, vemos a ilustração na Figura 22 (RENDLE; STEINER, 2021).

Figura 22 - Demonstrativo *bi directional streaming*



Fonte: Autoria própria.

Diferente dos demais padrões vistos o *gRPC* delimita um contrato entre cliente e serviço isto evita erros em tempo de execução.

4 ANTIPADRÕES EM MICROSSERVIÇOS

Após debater sobre os padrões arquiteturais do microsserviço e as possíveis escolhas de modelagem de *APIs* é importante entender alguns problemas encontrados na implementação da arquitetura de microsserviços. Um antipadrão pode ser definido como um aprendizado acerca de práticas que são ordinárias e provocam falhas ou induzem a falhas, estas práticas são estudadas e então tornam-se um antipadrão que aponta abordagem de um padrão como solução, como vemos definir Oliveira e França (2020, p 3) “Um antipadrão é a utilização recorrente de uma solução de código ou projeto (design) que leva a um resultado de baixa qualidade”.

4.1 *MEGASERVICE*

O *megaservice* é caracterizado pela alta carga operacional sob um determinado serviço. Este serviço sobrecarregado de funcionalidades torna-se muito grande e proporciona dificuldade de versionamento e manutenção. Isso pode ocorrer devido a uma interpretação errada dos contextos delimitados da aplicação, neste caso a correção seria haver uma revisão do escopo e validar a possibilidade de criação de um novo serviço a partir da divisão deste serviço (OLIVEIRA; FRANÇA, 2020).

Uma possível interpretação em uma aplicação de loja virtual seria, demonstrativamente interpretar dentro do mesmo serviço os contextos de pedido e pagamento, este tipo de interpretação levaria a um serviço que desempenharia diversas operações e sua lógica se tornaria complexa e difícil de manter.

A ação comum para estes casos é a decomposição em microsserviços de acordo com a análise do escopo da aplicação (OLIVEIRA; FRANÇA, 2020).

4.2 *NANOSERVICE*

O *nanoservice* diferentemente do *macroservice* tem característica por ser um com poucas operações, suas atribuições reduzidas fazem com que ele tenha que se comunicar com outros serviços para entregar o resultado esperado. Este tipo de comportamento põe em risco o desempenho da aplicação, uma vez que para obter o resultado o serviço acaba realizando múltiplas requisições a outros serviços e por fim essas comunicações consomem tempo e processamento seja de um *event bus* ou outro tipo de comunicação estabelecida (OLIVEIRA; FRANÇA, 2020).

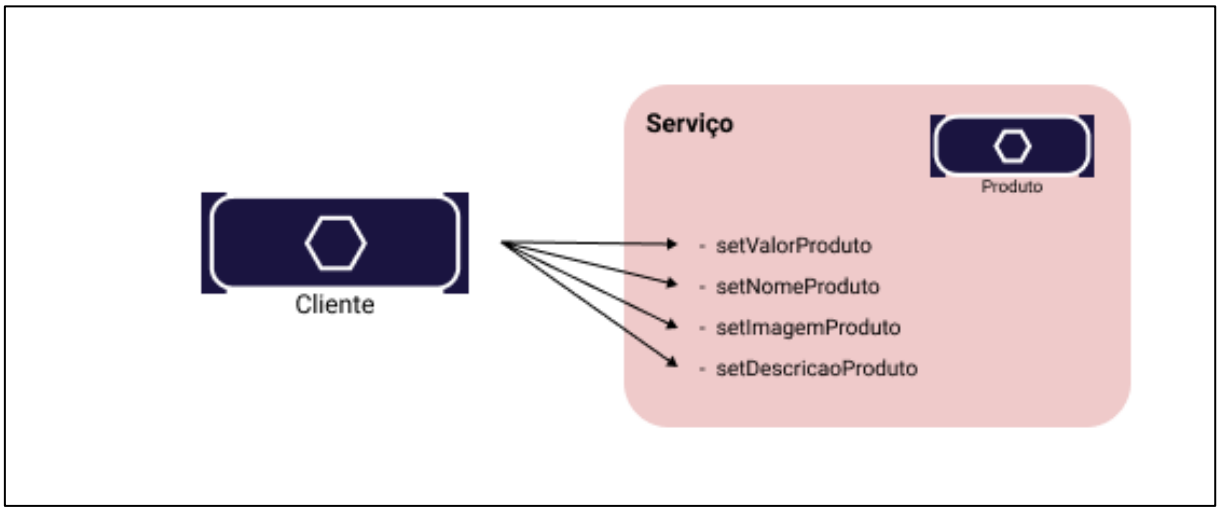
Este problema geralmente é causado pela dificuldade em estabelecer as delimitações de contexto da aplicação onde é criado de forma desnecessária um serviço. Para estes casos também é válido a revisão da delimitação dos contextos da aplicação, fazendo a composição deste serviço com outro serviço mais próximo dos objetivos deste contexto (OLIVEIRA; FRANÇA, 2020).

Uma possível interpretação que poderia levar a este antipadrão seria delimitar um serviço de produto item e outro apenas para produto, esta subdivisão não faria sentido e apenas traria mais trocas de mensagens entre os serviços.

4.3 *CHATTY SERVICE*

Este antipadrão é caracterizado pela separação das operações de um serviço em diversas funções. O serviço é coeso e tem uma atividade fim bem definida e apartada das demais aplicações, porém a complexidade inserida na maneira de consumir os recursos desse serviço provoca impactos de processamento e de consumo de rede, pois é necessário fazer diversas requisições ao serviço para se obter o resultado esperado (OLIVEIRA; FRANCA, 2020).

Figura 23 - Demonstração de *chatty service*



Fonte: Autoria própria.

Na Figura 23 vemos que diversas requisições são feitas para o mesmo serviço a fim fazer registro do recurso ou até mesmo obter informações específicas deste recurso.

5 MICROSSERVIÇO E *.NET*

Desenvolver uma arquitetura de microsserviços requer a utilização de padrões que estão aptos a serem aplicados em muitas tecnologias distintas, dentro das ferramentas de desenvolvimento da *Microsoft*, o *.Net* é uma dessas ferramentas possui código aberto também permite a utilização em outras plataformas além do *Windows*. Essa ferramenta possui algumas bibliotecas que nos ajudam a desenvolver dentro da arquitetura de microsserviços (TORRE; WAGNER; ROUSOS, 2021).

Nas subseções deste capítulo será relatado a utilização da ferramenta .*Net* na versão 6, cuja qual é a versão *LTS* (L*ong-term support*) da ferramenta no momento da, para implementação de *APIs* de requisição síncrona descritos na seção 3.

Nesta seção será demonstrado a implementação dos principais meios de construção de serviços visto na seção 3, também será disponibilizada a aplicação através deste endereço https://github.com/JulioSCr/TCC.Application/tree/master.

5.1 *IMPLEMENTAÇÃO REST*

A aplicação no padrão *REST* requer a implementação de algumas bibliotecas, uma muito importante é o *Swashbuckle*, biblioteca utilizada para apresentar a *IDL* com *swagger*. As demais bibliotecas utilizadas para implementação do *REST* são nativas do .*Net*.

Para utilizar o *Swagger* devemos fazer a configuração da biblioteca do *Swashbuckle e* adicionando seu serviço (*Microsoft*, 2021).

Quadro 5 - Adicionando biblioteca do *swagger*

```
services.AddSwaggerGen(swagger =>
{
    swagger.SwaggerDoc("v1", new OpenApiInfo
    {
        Title = "TCC Service Pais API",
        Description = "API demonstrando utilização de REST para
construção de serviços.",
        Contact = new OpenApiContact { Name = "Julio da Silva Cruz"},
        License = new OpenApiLicense { Name = "MIT", Url = new
Uri("https://opensource.org") }
    });
});
```

Fonte: Microsoft (2021).

No Quadro 5 temos uma das configurações possíveis para adição do *Swagger*, nela temos a versão da *API*, título e descrição. Também temos a adição da licença *MIT*, cuja qual é uma das licenças mais permissivas e por fim podemos adicionar um contato conforme feito na codificação acima (*Microsoft*, 2021).

Quadro 6 - Inserindo no *pipeline* o *swagger*

```
app.UseSwagger();
app.UseSwaggerUI(swagger =>
swagger.SwaggerEndpoint("/swagger/v1/swagger.json", "v1"));
```

Fonte: Microsoft (2021)

O Quadro 6 apresenta a utilização do *Swagger* dentro da aplicação. Após estas configurações as páginas do *Swagger* estarão disponíveis de acordo com a construção necessária dos *endpoints* nas *APIs* (*Microsoft*, 2021).

Para construção dos *endpoints* precisamos criar uma *controller* na aplicação, a *controller* é uma classe que faz a manipulação das requisições e retorna uma resposta da requisição.

Quadro 7 - Definição de *controller REST*

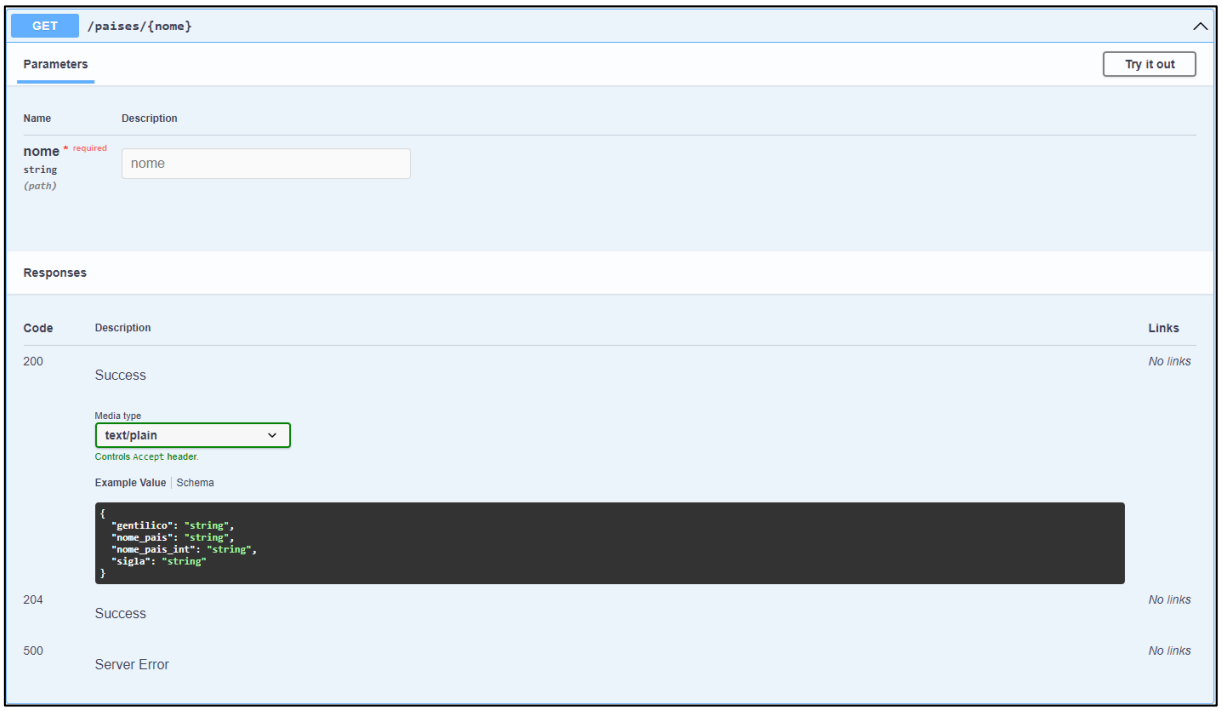
```
[Route("paises")]
public class PaisController : BaseController
{
  [HttpGet()]
  [ProducesResponseType(typeof(List<Pais>), StatusCodes.Status200OK)]
  [ProducesResponseType(StatusCodes.Status500InternalServerError)]
  public async Task<IActionResult> ObterPaisesAsync()...

  [HttpGet("{nome}")]
  [ProducesResponseType(typeof(Pais), StatusCodes.Status200OK)]
  [ProducesResponseType(StatusCodes.Status204NoContent)]
  [ProducesResponseType(StatusCodes.Status500InternalServerError)]
  public async Task<IActionResult> ObterPaisesPorNomeAsync(string
nome)...
}
```

Fonte: Microsoft (2021).

A criação da *controller* define o endpoint que será acessado, como vemos no Quadro 7 definimos o verbo que será utilizado na requisição e a rota. Também é uma boa prática definir os tipos de retorno produzidos pelo *endpoint*. Este contrato para utilização da *API* é exibido no pela *IDL* neste caso o *Swagger* (*Microsoft*, 2021).

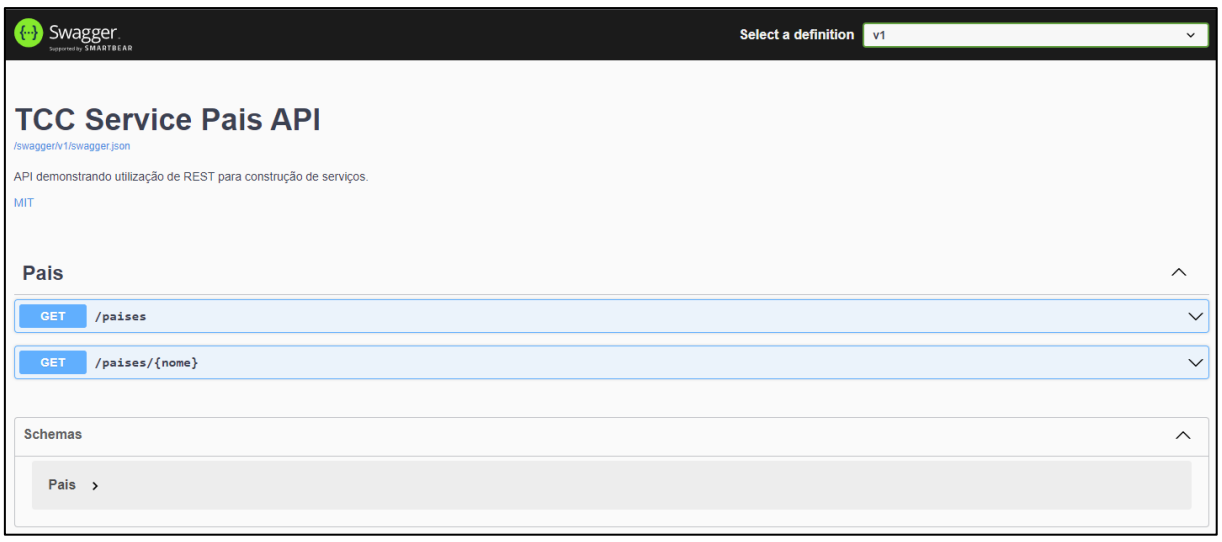
Figura 24 - Especificação da *API* através do *swagger*



Fonte: Swashbuckle (2021).

A Figura 24 demonstra a representação do *endpoint* definido no Quadro 7 para obter país por nome, conseguimos observar os tipos de retorno deste *enpoint* e o seu caminho para as requisições, assim como o que a *API* necessita receber como parâmetros, sendo possível fazer a passagem destes parâmetros em tela para realizar o teste deste *endpoint* (*Microsoft*, 2021).

Figura 25 - *IDL* da *API* construída em *REST*



Fonte: Autoria própria.

Na Figura 25 podemos visualizar a *API* e seus *endpoints* através da *IDL* do *Swagger*, esta exibição ajuda no momento de entender as funcionalidades da *API* e a compreender sua utilização (*Microsoft*, 2021).

5.2 IMPLEMENTAÇÃO *GRAPHQL*

Como o *GraphQL* traz em suas requisições uma linguagem própria para *query* e *manipulation* que devem ser processadas em tempo de execução na aplicação sua construção é mais complexa e trabalhosa que a *REST*, mas este dinamismo traz vantagens como visto na seção 3.3.

Apesar da implementação ser complexa podemos utilizar bibliotecas como *HotChocolate* para facilitar a construção dessas APIs. O *HotChocolate* traz uma série de facilidades na construção de uma *API GraphQL* apenas necessitando fazer a configuração da *query GraphQL* ou da *mutation* e a adição no *pipeline*, após esses passos construir seus *schemas* que são basicamente os contextos delimitados de cada *API* torna-se mais fluido.

Quadro 8 - Configuração *HotChocolate*

```
services
  .AddGraphQLServer()
  .AddQueryType<Query>();
```

Fonte: HotChocolate (2021).

No Quadro 8 podemos ver a configuração do servidor *GraphQL* utilizando *HotChocolate* e a adição da *Query* para país, mas para a utilização da *API* precisamos mapear estes serviços, para isso nas configurações fazemos o uso dos *enpoints* com o mapeamento necessário conforme vemos no Quadro 9 (Chillicream, 2021).

Quadro 9 - Utilizando *endpoints GraphQL* para renderização da interface da *API*

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapGraphQL();
});
```

Fonte: HotChocolate (2021).

A partir dessas configurações para fazer a utilização de nossa *query GraphQL* basta fazer a criação de uma classe, ainda para sua criação a biblioteca HotChocolate também traz algumas implementações que facilitam a codificação como vemos no Quadro 10 o *ScopedService* é um atributo fornecido pela fornecido pela biblioteca *HotChocolate* e é responsável por fazer a injeção de dependências do contexto de pais, que tem função de conexão com o banco de dados, esta classe tem um trabalho de codificação reduzido com a utilização desta biblioteca e como vemos basta obter a tabela na conexão com o banco de dados e o restante do tratamento de retorno será feito dentro da biblioteca *HotChocolate* (Chillicream, 2021).

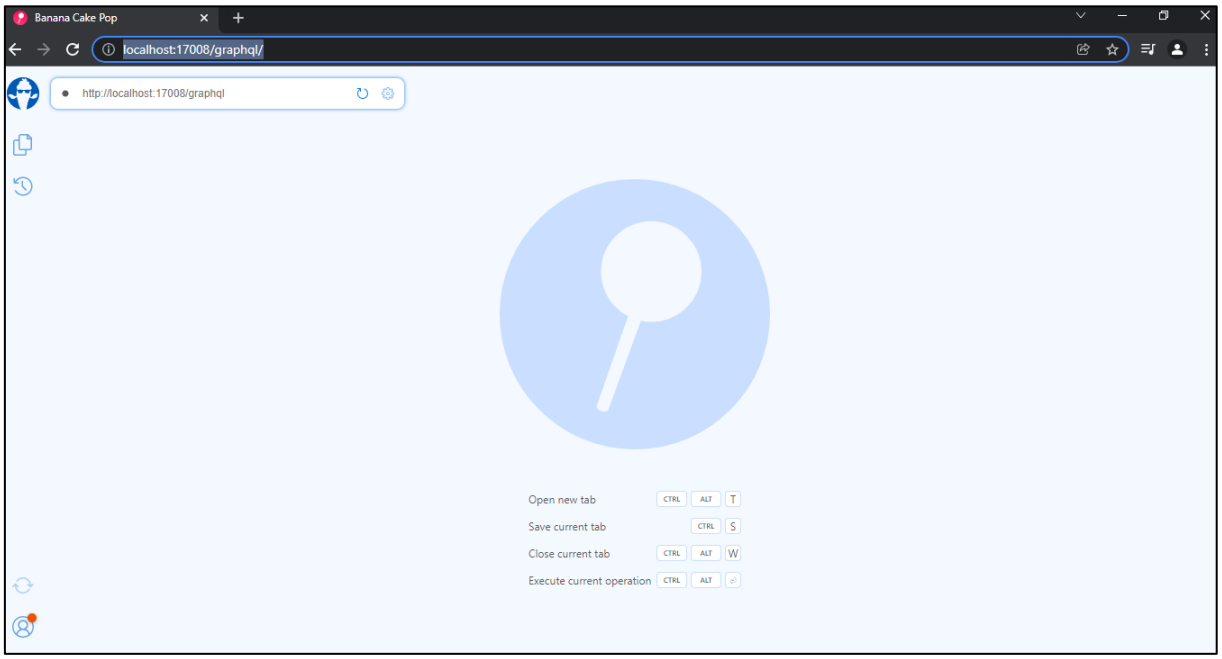
Quadro 10 - Criação de *query* de país para *API GraphQL* com *HotChocolate*

```
public class Query
{
  [UseDbContext(typeof(PaisDbContext))]
  public IQueryable<Pais> GetPais([ScopedService] PaisDbContext
context)
  {
      return context.Paises;
  }
}
```

Fonte: HotChocolate (2021).

Fazendo essa codificação chegamos ao resultado de uma *API GraphQL* para requisições de *query*, a partir deste ponto também é possível visualizar a documentação da *API* assim como no *Swagger* visto no *REST* na seção 5.1.

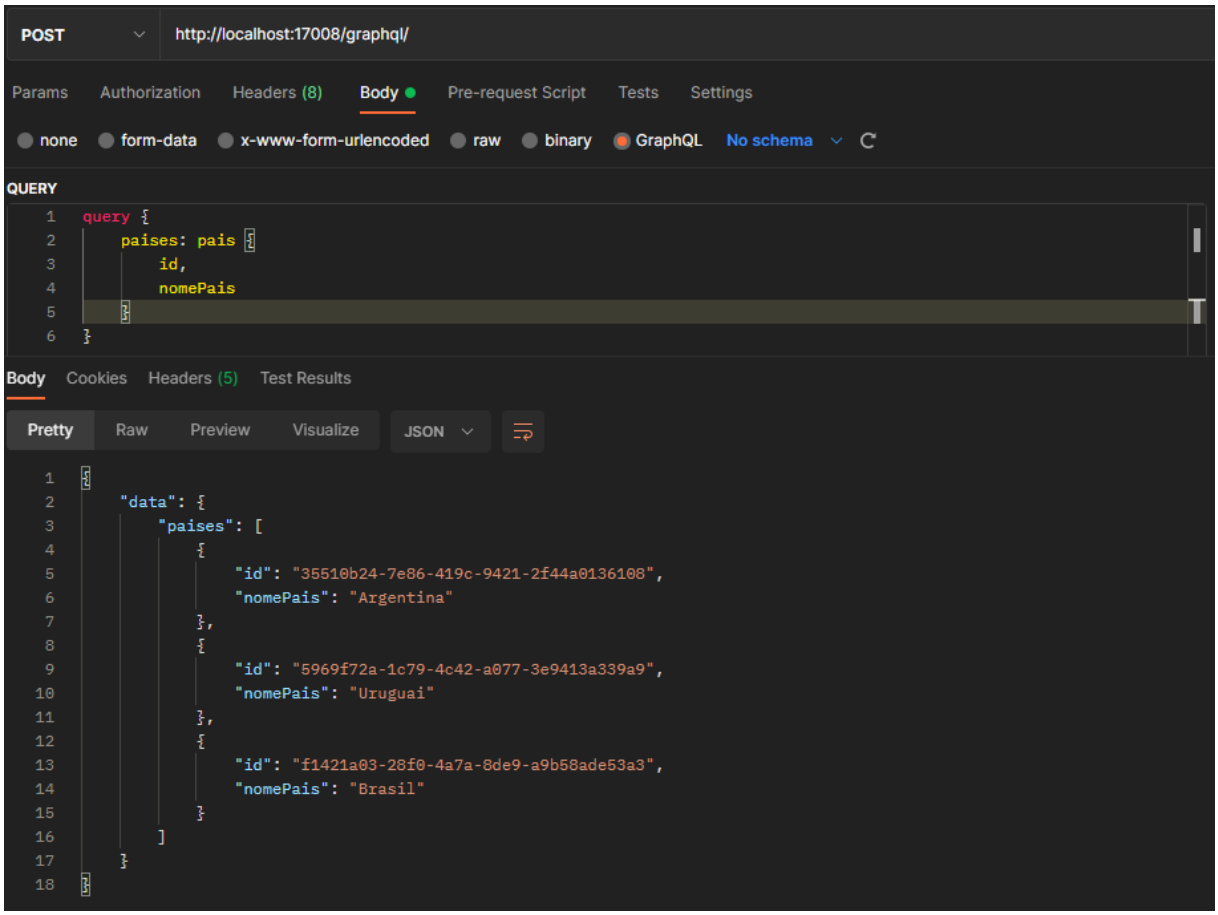
Figura 26 - Documentação *API GraphQL* com *Banana Cake Pop*



Fonte: Autoria própria.

Como vemos na Figura 26 a documentação através da página *web* é obtida através da *url graphql*, neste endereço visualizamos a interface do *Banana Cake Pop* uma interface disponibilizada pelo próprio *HotChocolate* (Chillicream, 2021).

Figura 27 - Consulta *GraphQL*



Fonte: Autoria própria.

Como vemos na Figura 27 conseguimos fazer uma requisição dinâmica com as *queries GraphQL* na *API* criada, esta construção permite os seus clientes fazerem requisições para obter apenas os campos necessários para seu consumo conforme vimos na seção 3.3.

5.3 IMPLEMENTAÇÃO *GRPC*

A implementação de uma *API gRPC* no *.Net* vem desde a versão 3.0 do *SDK* (*Software development kit*) do *.Net Core*. Com o pacote *Grpc.AspNetCore* temos todas as ferramentas necessárias para construção da *API gRPC*, desde a criação dos *protocol buffers* até a configuração de *web kestrel*, cujo qual é um servidor que está incluso e habilitado por padrão nas aplicações *.Net*, para utilização de *HTTP*/2.

O desenvolvimento pode iniciar pela construção do *protocol buffer*, arquivo que será compilado e transmitido em binário pela aplicação (RICHARDSON, 2018). No arquivo *protocol buffer* definimos as funções e modelos que serão utilizados nas requisições (RENDLE; STEINER, 2021).

Quadro 11 - *Protocol buffer* criado para requisição de obtenção de país por nome

```
syntax = "proto3";

option csharp_namespace = "TCC.Application.GRPC.Protos";

service PaisProtoService {
    rpc GetPais(GetPaisNomeRequest) returns (PaisModel);
}

message GetPaisNomeRequest {
    string nome = 1;
}

message PaisModel {
    string Gentilico = 1;
    string NomePais = 2;
    string NomePaisInternacional = 3;
    string Sigla = 4;
}
```

Fonte: Rendle, Steiner (2021).

No Quadro 11 vemos a definição de um serviço *PaisProtoService* com a funcionalidade *GetPais*. Esta funcionalidade tem que receber por parâmetro a mensagem *GetPaisNomeRequest* e retornar outra mensagem *PaisModel* que estão definidas abaixo como já visto na seção 3.4. Quando compilamos a aplicação automaticamente através das ferramentas do *Grpc.AspNetCore* são geradas as classes representativas do arquivo *protocol buffer*, neste exemplo foi gerada a classe *PaisProtoService* e *PaisProtoServiceBase*, a partir desses arquivos podemos criar outra classe que fará o tratamento para esse serviço *gRPC (*MICROSOFT, 2021).

Quadro 12 - Implementando serviço criado a partir de arquivo *protocol buffer*

```
public class PaisService : PaisProtoService.PaisProtoServiceBase
{
  private readonly IMapper _mapper;
  private readonly IPaisRepository _paisRepository;

  public PaisService(IMapper mapper, IPaisRepository paisRepository)
  {
    _mapper = mapper;
    _paisRepository = paisRepository;
  }

  public override async Task<PaisModel> GetPais(GetPaisNomeRequest
request, ServerCallContext context)
  {
    var pais = await _paisRepository.GetPaisAsync(request.Nome);

    if (pais == null) throw new RpcException(new
Status(StatusCode.NotFound, "Pais não encontrado."));

    return _mapper.Map<PaisModel>(pais);
  }
}
```

Fonte: Microsoft (2021).

Como vemos no Quadro 12 herdamos as classes criadas a partir do arquivo *protocol buffer* e integramos com os repositórios necessários para a obtenção das informações que a funcionalidade necessita responder para o cliente *(*MICROSOFT, 2021).

O último passo é a criação da configuração no *pipeline* como fizemos nas implementações com *REST* e *GraphQL*, para isto é necessário mapear o serviço *gRPC* conforme vemos no Quadro 13 através do método *MapGrpcService* passando como parâmetro a classe de serviço criada *(*MICROSOFT, 2021).

Quadro 13 - Adicionando serviço *gRPC* no *pipeline* da aplicação

```
app.UseEndpoints(endpoints =>
{
  endpoints.MapGrpcService<PaisService>();

  endpoints.MapGet("/", async context =>
  {
    await context.Response.WriteAsync("Communication with gRPC
endpoints must be made through a gRPC client.");
  });
});
```

Fonte: Microsoft (2021).

O cliente que consumirá este serviço em *gRPC* deve também implementar a biblioteca *Grpc.AspNetCore*, caso seja uma aplicação em *.Net*. Como demonstrativo foi construído a partir de uma *API* em *REST* uma requisição para este serviço em *gRPC* utilizando o padrão unário visto na seção 3.4. Para isso também foi necessário criar o mesmo arquivo *protocol buffer* na aplicação *REST*, também é necessário construir o serviço, porém desta vez não é feita a criação deste serviço para inserir comportamento na funcionalidade, mas sim para efetuar a requisição com a passagem dos parâmetros necessários para consumo como vemos no Quadro 14 desta vez utilizando a classe *PaisProtoServiceClient* obtida a partir da compilação do arquivo *protocol buffer (*MICROSOFT, 2021).

Quadro 14 - Criação de serviço cliente do serviço *gRPC*

```
public class PaisGrpcService
{
  private readonly PaisProtoService.PaisProtoServiceClient _client;

  public PaisGrpcService(PaisProtoService.PaisProtoServiceClient
client)
  {
    _client = client;
  }

  public async Task<PaisModel> GetPaisNomeAsync(string nome)
  {
    var getPaisRequest = new GetPaisNomeRequest { Nome = nome };
    return await _client.GetPaisAsync(getPaisRequest);
  }
}
```

Fonte: Microsoft (2021).

Por fim, como em todas as outras aplicações feitas nesta seção, se faz necessário fazer a configuração no *pipeline*, como vemos no Quadro 15. Através do método *AddGrpcClient* onde nas opções das configurações definimos o endereço *web* do serviço *gRPC* que será acessado *(*MICROSOFT, 2021).

Quadro 15 - Adicionando serviço cliente de *API gRPC*

```
services.AddGrpcClient<PaisProtoService.PaisProtoServiceClient>(optio
ns =>
  options.Address = new
System.Uri(Configuration["http://localhost:5003"]));
```

Fonte: Microsoft (2021).

5.4 IMPLEMENTAÇÃO *GATEWAY*

As *APIs Gateway* fazem parte da implementação de um sistema de microsserviços, principalmente quando utilizamos o padrão *REST* na construção de nossos serviços (RICHARDSON, 2018).

Para implementação de uma *API Gateway* a *Microsoft* disponibiliza uma biblioteca para facilitar a construção o *Ocelot* fornecendo, além do roteamento característico de um *Gateway*, a possibilidade de efetuar autenticações entre as requisições. Esta ferramenta não requer codificação em C# por parte da *API Gateway* que não sejam as configurações de serviço e *pipeline*, apenas em casos de autenticação será necessário fazer a criação do código específico para a validação (MICROSOFT, 2021).

Para iniciar a criação de uma *API Gateway* com *Ocelot* precisamos criar um arquivo *Json* onde iremos inserir as configurações de roteamento de acordo com as rotas e *endpoints* das *APIs* que serão consumidas (MICROSOFT, 2021).

Quadro 16 - Configuração *Ocelot API Gateway*

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/paises",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 39959
        }
      ],
      "UpstreamPathTemplate": "/paises",
      "UpstreamHttpMethod": [ "Get" ]
    }
  ],
  "GlobalConfiguration": {
    "BaseUrl": "http://localhost:39406"
  }
}
```

Fonte: Microsoft (2021).

No Quadro 16 vemos uma configuração básica de roteamento de uma *API Gateway* com *Ocelot*, a *GlobalConfiguration* corresponde as configurações globais da *API* através dela definimos a *Url* da *API Gateway* que estamos construindo através da propriedade *BaseUrl*. A lista de rotas estabelecida com o nome *Rotes* é responsável por mapear os caminhos que serão acessados na *API Gateway* e o redirecionamento que será feito para o serviço mapeado. Para realizar o mapeamento seguimos a lógica proposta pela documentação da biblioteca, onde basicamente as propriedades iniciadas com *Downstream* são responsáveis pela identificação do serviço que será consumido, já as propriedades iniciadas com *Upstream* delimitam as configurações das rotas acessadas na *API Gateway* (MICROSOFT, 2021).

O arquivo *Json* utilizado para configurar as rotas da *API Gateway* necessita ser adicionado nas configurações da aplicação, isto pode ser feito através da classe *Program*, onde conforme o Quadro 17 vemos a codificação para adicionar nas configurações os dados informados no arquivo *Json*.

Quadro 17 - Adicionando arquivo *Json* com configurações do *Ocelot*

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
  Host.CreateDefaultBuilder(args)
    .ConfigureAppConfiguration((host, config) =>
    {
      config
        .SetBasePath(host.HostingEnvironment.ContentRootPath)
        .AddJsonFile("ocelot.json", false, true)
        .AddEnvironmentVariables();
    })
    .ConfigureWebHostDefaults(webBuilder =>
    {
      webBuilder.UseStartup<Startup>();
    });
```

Fonte: Microsoft (2021).

As únicas configurações necessárias são a adição do serviço e o uso no *pipeline* na classe *Startup* conforme o Quadro 18.

Quadro 18 - Configurando o *Ocelot* para *API Gateway*

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddOcelot(Configuration);
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment
env)
{
  app.UseOcelot().Wait();
}
```

Fonte: Microsoft (2021).

6 CONCLUSÃO

A partir das premissas levantadas através de hipótese, confirmamos o comportamento da arquitetura de microsserviços onde os contextos delimitados da aplicação são divididos em serviços independentes. Os serviços têm a liberdade de implementar as tecnologias necessárias para sua funcionalidade sem impactar os outros serviços graças a escalabilidade fornecida pela arquitetura, desde que a comunicação seja transparente com os serviços consumidores utilizando. Estes serviços podem ser implantados separadamente, relativo à manutenção, podemos subir uma aplicação sem impactar o restante do sistema trazendo maior segurança e estabilidade, pois os serviços apenas comunicam-se um com o outro e seus códigos fonte estão em projetos distintos.

As aplicações com arquitetura em microsserviços trazem algumas complexidades, essas complexidades são geradas pelo desafio de encontrar o grupo certo de serviços para a aplicação, a má interpretação do negócio pode gerar abstrações incondizentes com a realidade e a arquitetura pode sofrer com antipadrões. Porém existem benefícios para as aplicações de grande porte, como a facilidade de divisão em times que, por fim, agiliza o processo de produção. Outro benefício é a confiabilidade gerada com base no fator de que o serviço que apresentar falha não impactará a aplicação como um todo, pois todos os serviços são apartados.

Para a comunicação dos serviços os meios assíncronos são utilizados com *event bus* fazendo comunicação através de mensageria, já as comunicações síncronas são realizadas através de padrões de modelagem de *APIs* como *REST*, *gRPC*, *SOAP* e *GraphQL*.

O *SOAP* é um padrão mais antigo e pode ser encontrado em aplicações antigas como a dos bancos, por exemplo. É possível que em um sistema de microsserviços alguns serviços façam requisições para este tipo de *API*, porém seu padrão já não é mais frequentemente utilizado para construção de novas aplicações, pela arquitetura defasada da construção desse modelo de *APIs*.

O *REST* é a arquitetura mais comum para a construção de serviços, ele traz facilidade de interpretação através dos verbos *HTTP*, porém seus retornos são engessados de acordo com o contrato pré-estabelecido na criação de seus *endpoints*, ou seja, a partir do momento de criação de um *endpoint* em *REST* os parâmetros necessários para consumi-la sempre serão os mesmos, também seu retorno sempre será o mesmo previsto na implementação da *API*. Além disso este padrão de arquitetura sofre no quesito de compartilhamento de dados de outras entidades, muitas vezes sendo necessário a criação de uma *API Gateway* para fazer a obtenção vários recursos em uma requisição é algo complexo na arquitetura destas *APIs*.

As *APIs GraphQL* são excelentes para consumo e velocidade de processamento, pois na requisição passamos os dados que queremos obter e até podemos passar os filtros que desejamos fazer para a obtenção destes dados, porém a implementação deste tipo de serviço é complexa e mais demorada por este mesmo motivo que os outros padrões como *REST*.

Já as *APIs gRPC* apresentam protocolo mais recente em mercado até o momento, o *HTTP*/2. Porém sua implementação é mais complexa que o padrão *REST*, mesmo apresentando melhor performance e diversas estratégias de integração com o cliente, o custo de implementação deste modelo é superior aos padrões mais simples como o *REST* que conseguem suprir atender as necessidades de comunicação dos serviços em uma velocidade menor.

O melhor padrão para utilizar na construção dos serviços da arquitetura é o padrão que atenderá aos requisitos de sistema e do prazo de construção do sistema. O *GraphQL* traz vantagem para consumo, mas sua construção no é complexa. Já o *REST* é fácil de interpretá-lo através dos verbos *HTTP* e entender a utilização dele, porém seus retornos são engessados através de um contrato pré-estabelecido em sua documentação, igual ao *SOAP* e diferentemente do *GraphQL* que possibilita a consulta dinâmica. O *gRPC* apresenta uma velocidade maior para trafegar informações graças ao seu protocolo e seu tipo de mídia padrão, mas sua implementação também é complexa.

Saber utilizar arquiteturas diferentes para construção dos serviços de nossa aplicação, não é o suficiente para obter uma aplicação coesa e funcional, para tal é importante atentar-se também aos antipadrões existentes, pois assim evitamos construir serviços que são muito grandes e que devem ser divididos em novos serviços, ou em serviços minúsculos cujas operações podem ser aglutinadas em outro serviço.

REFERÊNCIAS

BUNA, S. **GraphQL in Action**, Shelter Island: Manning, 2021.

CHILLICREAM. **HotChocolate documentation**. 23 nov. 2021. Disponível em: <https://chillicream.com/docs/hotchocolate/>. Acesso em 23 nov. 2021.

ENGLANDER, R. **Java and SOAP**, Sebastopol: O'Reilly, 2002.

EVANS, E. **Domain-Driven Design Taking Complexity in the heart of software**. Boston: Addilson-Wesley Professional, 2003.

FIELDING, R. T. **Architectural styles and the design of network-based software architectures**, Irvine, 2000.
Disponível em:
https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. Acesso em: 19 nov. 2021.

GOOGLE. **Protocol buffer**. 25 nov. 2021. Disponível em: <https://developers.google.com/protocol-buffers>. Acesso em 25 nov. 2021.

JR, C. C.; SCHMELMER, T. **Microservices from day one**. New York: Apress, 2016.

MICROSOFT. **Dotnet documentation**. 24 nov. 2021. Disponível em: <https://docs.microsoft.com/en-us/dotnet/>. Acesso em 24 nov. 2021.

MIRANDA, D. Q. **Um servidor HTTP/2 reativo em scala**, São Paulo, 2016.
Disponível em:
https://bcc.ime.usp.br/tccs/2015/rec/reviewupdatenotdone/danielqm/monografia.pdf.
Acesso em: 24 nov. 2021.

OLIVEIRA, D. H. P.; FRANÇA, B. B. N. **Estudo sobre Antipadrões em microsserviços e os impactos na evolução do processo de desenvolvimento de aplicações**. Estudo – Instituto de Computação, Universidade estadual de campinas, Campinas, 2020.

RENDLE, M.; STEINER, M. **gRPC for WCF Developers**, Redmond: Microsoft, 2021.

RICHARDSON, C. **Microservices patterns**. Shelter Island: Manning Publications Company, 2018.

SUBRAMANIAN, H.; RAJ, P. **Hands-On RESTful API design pattern and best practices**, Birmingham: Packt, 2019.

TORRE, C.; WAGNER, B.; ROUSOS, B. **.NET Microservices: Architecture for Containerized .NET Applications**. Redmond: Microsoft Corporation, 2021.