APLICANDO ARQUITETURA DE MICROSSERVIÇOS NO DESENVOLVIMENTO DE SOFTWARE

MICROSSERVICE ARCHITECTURE IN SOFTWARE DEVELOPMENT

Roni da Cruz SILVA

ronidacruz@gmail.com

Ciências da Computação, Centro Universitário Padre Anchieta

Prof. Esp. Rodrigo SAITO

rodrigok@anchieta.br

Ciências da Computação, Centro Universitário Padre Anchieta

Resumo

O desenvolvimento de sistemas evolui a cada dia. Novas tecnologias surgem constantemente e é preciso estar atualizado sempre para não ser superado. A arquitetura de software busca organizar os elementos, suas formas e a lógica envolvida independente da tecnologia escolhida. Neste trabalho, analisamos os princípios da Arquitetura de Microsserviços e como ela pode ajudar a resolver a necessidade de construir sistemas que possam ser escalados, que tenham tolerâncias a falhas, que possam implementar mais de uma linguagem de programação e que se adaptem a evolução constante que vivemos atualmente. Este trabalho apresenta os principais padrões de desenvolvimento que podem ser usados em microsserviços e demonstra de forma prática como aplicar estes conceitos.

Palavras-Chave

Arquitetura de Software; Padrões de Desenvolvimento; Microsserviços.

Abstract

Systems development evolves every day. New technologies are constantly appearing and you must always be up to date in order not to be surpassed. Software architecture aims to organize the elements, their formats and the logic involved regardless of the chosen technology. In this article, we analyze the principles of Microservice Architecture and how it can help solve the need to build systems that can scale, with fault tolerance, that can implement more than one programming language and adapt to the constant evolution we live in today. This work presents the main development patterns that can be used in microservices and demonstrates in a practical way how to apply these concepts.

Keywords

Software Architecture; Design Patterns; Microservices.

INTRODUÇÃO

No processo de desenvolvimento de software, é necessário identificar quais são os problemas que devem ser resolvidos pelo sistema e determinar qual será a arquitetura utilizada para resolver estes problemas.

Conforme Perry e Wolf (1992) a arquitetura de software consiste de três componentes: elementos, formato e lógica. Os elementos podem ser processamentos, dados e elementos de comunicação. O formato é definido pelas propriedades e as relações dos elementos. A lógica fornece a base para a arquitetura de acordo com as restrições do sistema que na maioria das vezes derivam dos requisitos do sistema.

Com a evolução da tecnologia, alguns modelos de arquitetura de software surgiram como cliente-servidor, aplicação monolítica, desenvolvimento em camadas, arquitetura orientada a serviço, etc. De acordo com Chris Richardson (2019), as arquiteturas tradicionalmente conhecidas como monolíticas podem oferecer simplicidade para desenvolver, implantar e escalar a aplicação. No entanto, quando a aplicação se torna um pouco maior, essa abordagem pode apresentar várias desvantagens como: o código fonte pode ser difícil de entender, a aplicação pode não ter os limites dos módulos bem definidos, as alterações necessárias podem trazer perda de qualidade no código, o ambiente de desenvolvimento pode ser lento devido ao tamanho do código fonte, o tempo de inicialização pode ser alto, as implantações de atualizações podem causar inconvenientes ao deixar o sistema indisponível em algum momento, o sistema escala apenas de forma vertical, é complicado dividir responsabilidades entre as equipes e causa dependência de determinada tecnologia utilizada.

Segundo o site da Red Hat, a arquitetura orientada a serviço serve pra resolver essas questões, pois estrutura as aplicações em serviços distintos e reutilizáveis que se comunicam entre si.

Neste contexto, surge o termo microsserviço, que para James Lewis e Martin Fowler (25/03/2014), esta arquitetura é uma abordagem para desenvolver um sistema como um conjunto de pequenos serviços, cada um executando em seu próprio processo e se comunicando através de mecanismos leves, geralmente API's baseadas no protocolo HTTP. Estes serviços são construídos de acordo com uma necessidade do negócio e implantados de forma independente e automática. Há um mínimo de centralização e estes serviços podem ser escritos em diferentes linguagens de programação e usar diferentes tecnologias de armazenamento de dados. Eles também afirmam que não há uma definição formal do estilo arquitetural de microsserviços, mas é possível descrever quais características são comuns. Dessa forma, nem todas as arquiteturas de microsserviços possuem todas as características implementadas, mas se espera que a maioria das arquiteturas de microsserviços possuam a maioria destas características.

COMPARAÇÃO DE ARQUITETURAS

Para entender melhor a arquitetura de microsserviços, vamos comparar com a arquitetura monolítica, analisando suas vantagens e desvantagens.

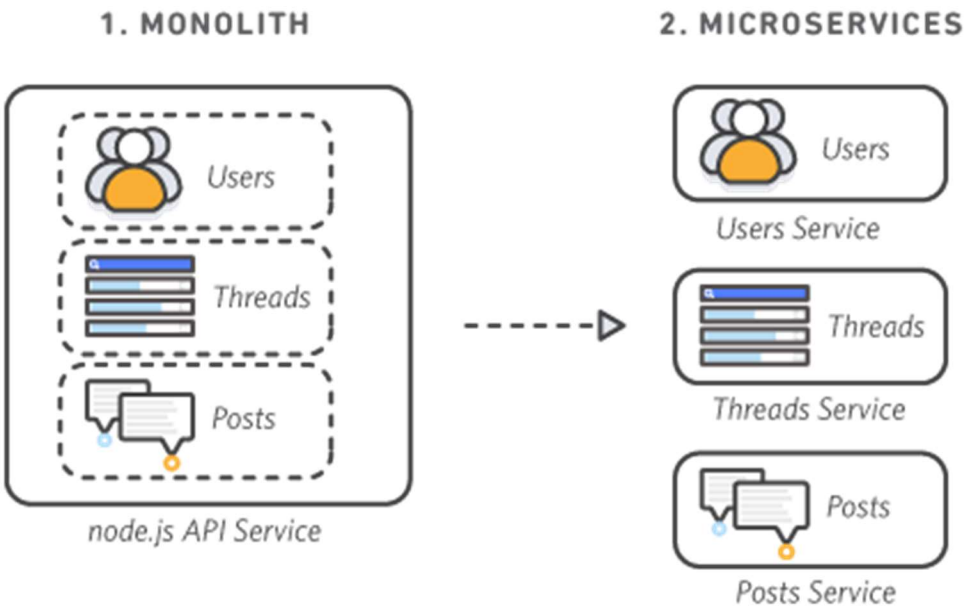

Figura 24 *- Arquitetura Monolítica versus Microsserviços*

https://d1.awsstatic.com/Developer%20Marketing/containers/monolith_1-monolith-microservices.70b547e30e30b013051d58a93a6e35e77408a2a8.png

Arquitetura Monolítica

Conforme Penna (13/05/2020), "arquitetura monolítica é um sistema único, não dividido, que roda em um único processo, uma aplicação de software em que diferentes componentes estão ligados a um único programa dentro de uma única plataforma".

"Com as arquiteturas monolíticas, todos os processos são altamente acoplados e executam como um único serviço. Isso significa que se um processo do aplicativo apresentar um pico de demanda, toda a arquitetura deverá ser escalada." (AMAZON AWS, 2021)

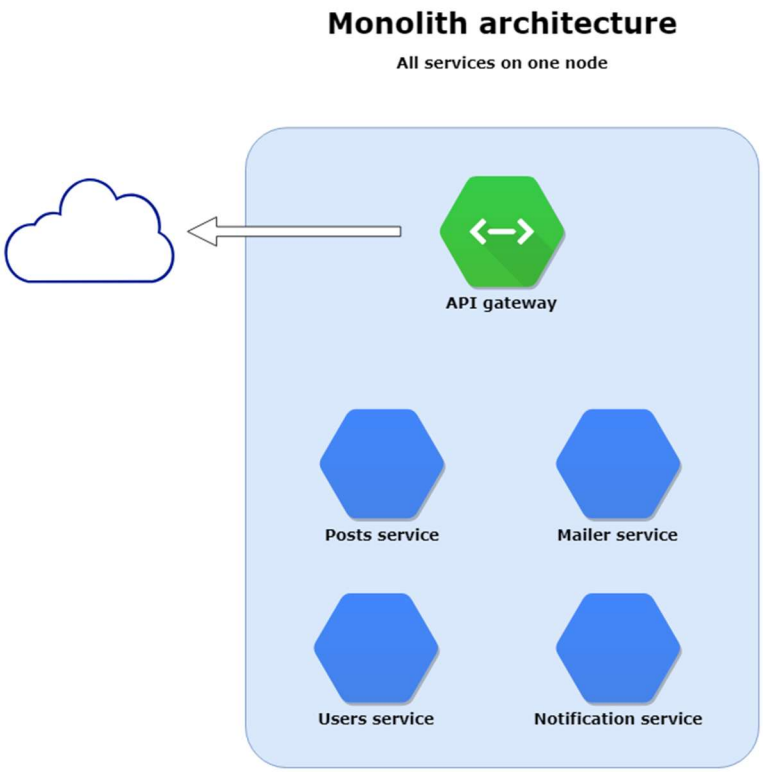

Figura 25 *- Arquitetura Monolítica*

https://moleculer.services/docs/0.14/clustering.html

Vantagens da Arquitetura Monolítica

De acordo com Chris Richardson, inicialmente, a arquitetura monolítica pode apresentar as seguintes vantagens:

- Simplicidade no desenvolvimento - *IDE's* e outras ferramentas de desenvolvimento estão empenhadas em desenvolver uma única aplicação.
- Facilidade para fazer mudanças radicais - É possível alterar o código, os esquemas de banco de dados, empacotar e implantar o sistema. Praticamente tudo em um único lugar.
- Facilidade para criar rotinas de testes - Os desenvolvedores podem criar testes integrados que inicia a aplicação e executa todo um processo do começo ao fim.
- Facilidade na implantação - Tudo que um desenvolvedor precisa fazer é transportar um único pacote para o ambiente produtivo.
- Facilidade para escalar - Nesta arquitetura, é possível escalar de forma vertical, criando novas instâncias de todo o sistema. (RICHARDSON, 2019, p. 4)

Desvantagens da Arquitetura Monolítica

Com o passar do tempo, a aplicação tende a ficar maior e mais complexa e podem surgir as seguintes desvantagens:

- Complexidade - Uma aplicação simples, com o passar do tempo, pode receber constantemente novas funcionalidades, o que torna o código-fonte cada vez mais amplo. O tamanho da equipe também tende a aumentar e é necessário um esforço maior para organização e gerenciamento.

Isso implica que a aplicação é muito grande e difícil de entender completamente. Cada novo ajuste necessário torna-se difícil e consume tempo. Para piorar a situação, toda essa complexidade pode tornar-se um ciclo vicioso, pois se o código é difícil de entender o desenvolvedor pode não aplicar os ajustes da melhor forma possível, aumenta a complexidade. O sistema gradualmente pode tornar-se uma monstruosa, incompreensível, grande bola de lama.

- Lentidão no desenvolvimento - Além de ter que lidar com a complexidade do sistema, os desenvolvedores enfrentam lentidão em tarefas do dia-a-dia. A aplicação gigante sobrecarrega a interface de desenvolvimento e o processo de *building* é bem mais demorado. Além disso, por ser tão grande, a aplicação também demora para iniciar, afetando toda a produtividade da equipe.
- O processo de implantação é longo e árduo - Com o crescimento desordenado da aplicação, cada alteração no sistema precisa passar por um processo longo, lento e doloroso para chegar no ambiente produtivo. A equipe precisa fazer *deploys* em intervalos maiores, tipicamente aos finais de semana para reduzir o impacto.
  Como muitos desenvolvedores estão trabalhando no mesmo código, frequentemente o sistema está com alterações em andamento, então o gerenciamento de mudanças torna-se mais complexo.
- As rotinas de testes também são mais longas, e a cada mudança o sistema demora para ter estabilidade.
- Dificuldade para escalar - Os diversos módulos de um sistema podem ter requisitos bem diferentes de memória, processamento e armazenamento. Como todos estão implantados no mesmo servidor, a configuração do servidor precisa atender a todos. Isso pode causar desperdícios de recursos para tarefas que não precisam e falta de recursos para atividades que requerem mais poder do servidor.
- Desafios de uma entrega confiável - Outra grande desvantagem é a dificuldade para entregar confiança. A rotina de testes pode não cobrir todo o código e bugs podem acabar sendo descobertos em produção. A aplicação também carece de mecanismos de tolerância a falhas porque todos os módulos estão sendo executados no mesmo processo. Frequentemente, uma falha em um módulo específico pode comprometer todos os outros módulos.
- Finalmente, o sistema está fortemente acoplado à tecnologia escolhida inicialmente. A arquitetura monolítica torna mais difícil adotar novas linguagens de programação ou diferentes frameworks. É muito arriscado inovar, por isso, é comum trabalhar em ambientes extremamente obsoletos. (RICHARDSON, 2019, p. 4-7)

Arquitetura de Microsserviços

Conforme a IBM, microsserviços é um estilo de arquitetura, no qual grandes e complexos aplicativos de software são compostos por um ou mais serviços. Os microsserviços podem ser implantados independentemente uns dos outros e estão fracamente acoplados. Cada um desses microsserviços se concentra em completar uma tarefa apenas e faz essa tarefa muito bem. Em todos os casos, essa tarefa representa uma pequena capacidade do negócio. (IBM REDBOOKS, 2015)

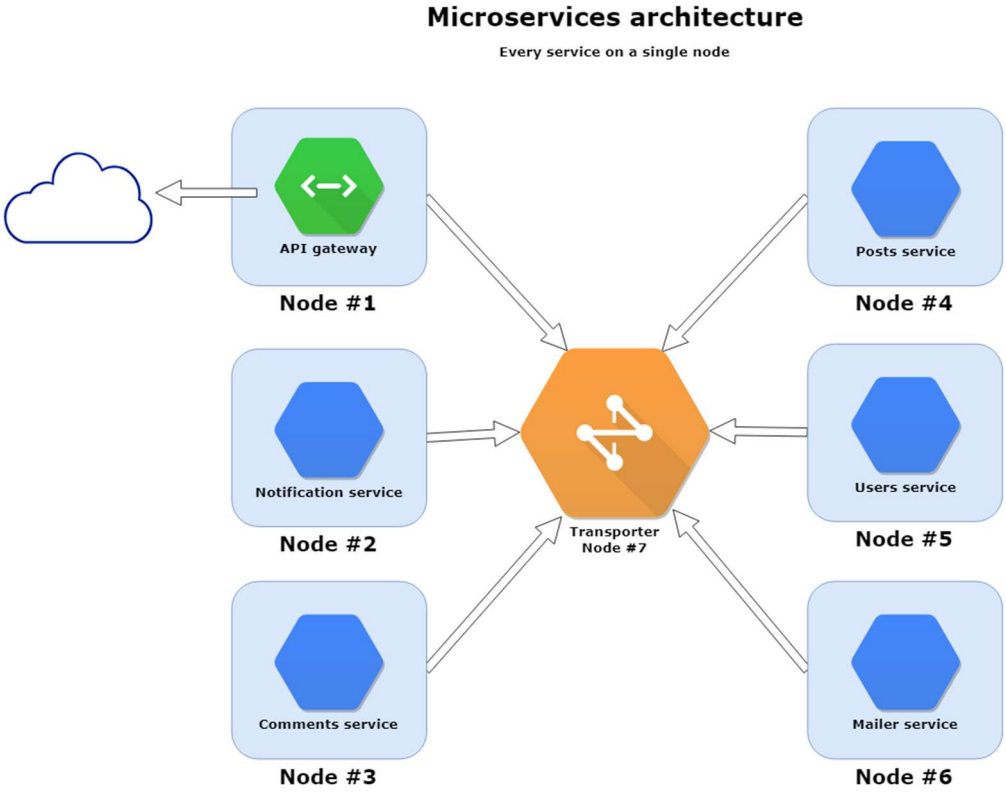

Figura 26 *- Arquitetura de Microsserviços*

https://moleculer.services/docs/0.14/clustering.html

Vantagens da Arquitetura de Microsserviços

Chris Richardson apresenta os seguintes benefícios da arquitetura de microsserviços:

- Permite a entrega e implantação contínuas de sistemas grandes e complexos - Entrega e implantação contínua faz parte do *DevOps*, um conjunto de práticas para a entrega rápida, frequente e confiável de software. Isso é facilitado pela arquitetura de microsserviços de três maneiras:
    - Automação de Testes - Como cada serviço é relativamente pequeno, testes automáticos são fáceis de implementar e executar.
    - *Deploy* - Cada serviço pode ser implantado de forma independente dos demais. Se uma alteração local for necessária em um serviço, a equipe responsável não precisa coordenar a implantação com as demais equipes. Dessa forma é muito mais fácil implantar mudanças em produção e de forma mais frequente.
    - Autonomia das equipes - As equipes podem ser organizadas em times menores, onde cada time pode desenvolver, implantar e escalar seus serviços de forma independente.
- Serviços são pequenos e de fácil manutenção - O código menor é mais fácil de ser entendido pelo desenvolvedor. A interface de desenvolvimento fica mais leve, e os serviços iniciam mais rapidamente, aumentando a produtividade.
- Serviços são escaláveis de forma independente - Cada serviço pode ser escalado de acordo com suas necessidades. Além disso, cada serviço pode ser implantado no hardware mais adequado. Isso resulta em redução de custos.

- Isolamento de falhas - A arquitetura de microsserviços possui um melhor isolamento de falhas, já que se um serviço estiver esgotando os recursos, isso deve afetar somente aquele serviço. E se algum serviço estiver fora do ar, os demais ainda podem funcionar.
- Adoção de novas tecnologias - Finalmente, a arquitetura de microsserviços permite experimentar novas tecnologias, já que a equipe pode escolher livremente a linguagem de programação e frameworks que melhor se ajustarem a necessidade. (RICHARDSON, 2019, p. 14-17)

Desvantagens da Arquitetura de Microsserviços

Implementar uma arquitetura de microsserviços possui vários desafios. Para a IBM, é importante analisar as seguintes situações:

- Não iniciar com microsserviços - Se a aplicação é pequena e a equipe não tem experiência com microsserviços, é melhor manter as coisas simples.
- Não planejar microsserviços sem DevOps - Microsserviços geram muitas partículas que precisam ter um fluxo sério de implantação, passando por testes automatizados e implantação e entrega contínua.
- Evitar administrar a própria infraestrutura - Microsserviços introduzem muitos conceitos com o que se preocupar, como banco de dados, serviços de mensagens, servidores que demandam cuidados. É melhor deixar essa parte com empresas que já fazem isso com excelência do que manter uma equipe própria preocupada com isso.
- Não iniciar com muito serviços - Cada serviço exige recursos. Em alguns casos é melhor iniciar um serviço maior e de acordo com o crescimento, dividi-lo, se for necessário.
- Problemas com latência - Criar serviços que dependam de outros pode introduzir problemas de latência. Ferramentas de monitoramento são essenciais para identificar problemas de comunicação entre os serviços. (IBM REDBOOKS, 2015, p. 10-12)

Arquitetura Orientada a Serviços (SOA)

Arquitetura orientada a serviços (SOA) é um tipo de design de software que torna os componentes reutilizáveis usando interfaces de serviços com uma linguagem de comunicação comum em uma rede. Em outras palavras, a arquitetura SOA integra os componentes de software que foram implantados e são mantidos separadamente, permitindo que eles se comuniquem e trabalhem juntos para formar aplicações de software que funcionam em sistemas diferentes. (RED HAT, 27/07/2020)

De uma forma geral, SOA e microsserviços são padrões de desenvolvimento que estruturam um sistema como um conjunto de serviços. Em um nível alto, existe similaridades. Mas olhando de perto, podemos encontrar diferenças importantes:

- Comunicação - SOA tipicamente usa tecnologias robustas e centralizadas para comunicação como SOAP e frequentemente fazem uso de um ESB (*enterprise service bus*) para integração dos serviços. Microsserviços costumam usar protocolos mais leves como REST ou gRPC.
- Dados - SOA geralmente tem um modelo global de dados com banco de dados compartilhados. Na arquitetura de microsserviços, cada serviço possui sua base de dados própria.
- Tamanho - SOA, geralmente é usado para integrar grandes e complexas aplicações. Mesmo que na abordagem de microsserviços nem sempre os serviços sejam extremamente pequenos, eles são quase sempre bem menores. Dessa forma, uma aplicação SOA geralmente consiste em alguns grandes serviços, enquanto uma aplicação baseada em microsserviços normalmente consiste em dezenas ou centenas de serviços menores. (RICHARDSON, 2019, p. 13-14)

Para concluir a comparação, observamos os objetivos. SOA tenta expor seus serviços a qualquer um que queira usá-los. Os microsserviços, alternativamente, são criados com um objetivo muito mais focado e limitado em mente, que é atuar como parte de um único sistema distribuído. (IBM REDBOOKS, 2015, p. 13)

ARQUITETURA DE MICROSSERVIÇOS - CONCEITOS ESSENCIAIS

Conforme explicado por Prates (2021), as empresas tech quando nascem começam a se desenvolver e criar softwares, que são suas soluções para o mercado, muitas vezes desenvolvidas em monólitos. Ao criá-las, ninguém sabe o tamanho exato que a empresa vai alcançar. Aos poucos, com novas soluções implementadas, a companhia começa a crescer, mudar e se adaptar. Surge então a necessidade de migrar o sistema legado para uma arquitetura mais dinâmica, orientada a microsserviços. Mas como fazer? Quebrar uma parte da aplicação e refazer uma parte específica ou recriar todo o sistema? (PRATES, 08/03/2021)

Strangler Pattern

Martin Fowler apresentou, em 2004, uma técnica de conversão que consiste em envolver a aplicação, substituindo seus módulos por microsserviços.

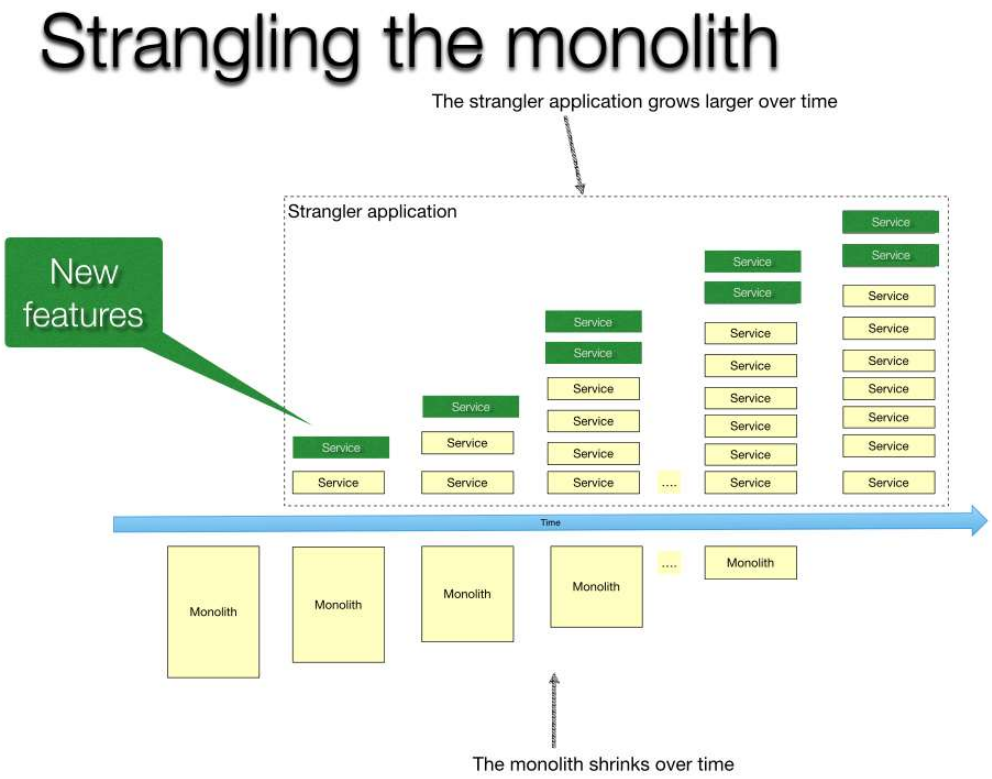

Figura 27 *- Strangler Pattern*

https://microservices.io/i/decompose-your-monolith-devnexus-feb-2020.001.jpeg

O padrão de desenvolvimento *Strangler application* permite modernizar uma aplicação de forma incremental, desenvolvendo uma nova aplicação em volta da aplicação legado, estrangulando aos poucos as funcionalidades. O aplicativo estrangulador consiste em dois tipos de serviços. Primeiro, existem serviços que implementam funcionalidades que residiam anteriormente no monólito. Em segundo lugar, existem serviços que implementam novos recursos. Gradualmente o tamanho do monólito diminui e a aplicação orientada a microsserviços aumenta (RICHARDSON, 2017).

Divisão de Responsabilidades

Com base em Gonçalves (2020), a implementação dos microsserviços consiste em um conjunto de padrões de design conceituais, divididos em categorias como aplicação, redes, infraestrutura, banco de dados, etc. Sendo cada unidade executando dentro de seu próprio processo, descrevendo características de serviços auto-contidos, autônomos e independentes.

A arquitetura de microsserviços propõe descentralizar a responsabilidade dos dados. Cada escopo de transação é tratado pelo microsserviço responsável dentro de seu contexto e limites transacionais adequados, devendo trabalhar com etapas de compensação das operações em mente do início ao fim do processo. A independência contribui para agilidade, trazendo maior liberdade para reagir rapidamente a mudanças e tomar decisões.

Ao desenvolvermos microsserviços, precisamos praticar o design de composição dos componentes com o domínio em mente. Modelos de domínio fazem mais ou menos sentido dependendo do contexto ao qual eles são observados, sendo assim, práticas de *domain-driven design* nos ajudam a compreender e construir modelos mais fiéis ao domínio de negócio.

Isolamento e Independência

Uma instância, que também pode ser chamada de nó, é um simples processo sendo executado numa rede local ou externa. Uma única instância de um nó pode hospedar um ou mais serviços. Dois (ou mais) serviços rodando em um único nó são considerados serviços locais. Eles compartilham recursos de hardware e usam o barramento local para se comunicar entre si, sem latência da rede (módulo de transporte não é usado). Serviços distribuídos através de vários "nós" são considerados remotos. Neste caso, a comunicação é feita através de um módulo de transporte (MOLECULERJS, 2021).

A computação distribuída trata de possibilitar maior disponibilidade e alto poder de escalabilidade quando necessário. Uma das necessidades que teríamos seria a de isolamento entre os componentes, pois ao executarmos cada microsserviço em seu próprio processo dentro de um servidor, adquirimos maior agilidade. Uma das razões para querermos o isolamento (independência) seria para lidar com falhas separadamente. (GONÇALVES, 01/06/2020)

Comunicação entre serviços

A arquitetura de microsserviços estrutura uma aplicação como um conjunto de serviços. Em muitos casos, esses serviços precisam colaborar entre si para responder uma requisição. Dessa forma, é necessário reservar tempo para analisar esse processo de comunicação entre os serviços.

Existem várias tecnologias que podem ser adotadas, como o padrão REST baseado no protocolo HTTP ou gRPC. Também é possível usar sistemas de mensagens assíncronas, baseadas em filas, com diversos formatos de mensagens. Os serviços podem usar formatos baseados em textos como JSON ou XML, ou também protocolos em formatos binários. (RICHARDSON, 2019, p. 65-67)

Estilos de comunicação

Há uma variedade de estilos de comunicação. Uma requisição pode ser processada exatamente por um único serviço ou por vários. E a comunicação também pode ser síncrona, assíncrona ou em sentido único.

- Síncrona: Um cliente de serviço faz uma solicitação a um serviço e espera por uma resposta. O cliente espera que a resposta chegue em tempo hábil. Pode haver um bloqueio de eventos enquanto espera. Este é um estilo de interação que geralmente resulta em serviços fortemente acoplados.
- Assíncrona: Um cliente de serviço envia uma solicitação a um serviço, que responde de forma assíncrona. O cliente não bloqueia enquanto espera, porque o serviço pode levar muito tempo para enviar a resposta.
- Sentido único: Um cliente de serviço envia uma solicitação a um serviço, mas nenhuma resposta é esperada ou enviada.

Ao estabelecer uma comunicação múltipla, o sistema de eventos é aplicado. Um cliente publica uma mensagem que é consumida por zero ou muitos serviços interessados naquele evento. (RICHARDSON, 2019, p. 67-68)

Usando REST

Conforme Silva (2017), REST é um conjunto de regras e padrões, enquanto *RESTful* é a implementação dessas regras em uma API. De acordo com Roy Fielding, um dos idealizadores do modelo arquitetural REST, para que uma API seja considerada *RESTful* esta deve obrigatoriamente seguir um conjunto de regras pré-definidas. Richardson (2019) propôs um modelo que define o nível de maturidade de uma API.

- Nível 0: Os clientes de um serviço de nível 0 invocam o serviço fazendo solicitações HTTP para seu *endpoint* único. No entanto, não existe um padrão.
- Nível 1: Um serviço de nível 1 introduz a ideia de recursos. Para realizar uma ação em um recurso, um cliente faz uma solicitação POST que especifica a ação a ser executada.
- Nível 2: Um serviço de nível 2 usa verbos HTTP para realizar ações: GET para recuperar, POST para criar e PUT para atualizar. Os parâmetros e o corpo da consulta da solicitação, se existir, especifica os parâmetros das ações. As respostas também contêm um código de status HTTP padrão.
- Nível 3: Conhecido como HATEOAS, a ideia básica é que a representação de um recurso retornado por uma solicitação GET contém links para executar ações naquele recurso. (RICHARDSON, 2019, p. 74)

Service registry

Para Peyrott (2015), o *service registry* é um catálogo de serviços preenchido com informações sobre como encaminhar solicitações para cada instância dos microsserviços.

A maioria das arquiteturas baseadas em microsserviços está em constante evolução. Os serviços aumentam e diminuem conforme as equipes de desenvolvimento se dividem, aprimoram, descontinuam e fazem seu trabalho. Sempre que um *endpoint* de serviço muda, o registro precisa saber sobre a mudança. É disso que se trata o registro: é o responsável por publicar ou atualizar as informações de como chegar a cada serviço.

Service discovery

O *service discovery* é a contrapartida do s*ervice registry* do ponto de vista dos clientes. Quando um cliente deseja acessar um serviço, ele deve descobrir onde o serviço está localizado e outras informações relevantes para realizar a solicitação.

Quando este serviço está no lado do cliente ele pode forçar o mesmo a consultar o *service discovery* antes de executar a requisição final.

Quando este serviço está do lado do servidor, o serviço de API Gateway é o responsável por descobrir o *endpoint* exato para cada requisição. (PEYROTT, 02/10/2015)

PADRÕES DE DESENVOLVIMENTO

Uma plataforma de aplicações engloba todas as ferramentas que são independentes de um microsserviço. Essas ferramentas devem ser construídas e dispostas de forma que o time de desenvolvimento não tenha que se preocupar com nada além da lógica da própria aplicação. (OPUS SOFTWARE, 31/05/2021)

Service Mesh

O padrão *service mesh* é uma maneira de controlar como diferentes componentes de uma aplicação compartilham dados entre si. (RED HAT)

Um *service mesh* gerencia toda a comunicação "serviço a serviço" em um sistema de software distribuído. Um *service mesh* fornece a descoberta dinâmica de serviços e gerenciamento de tráfego. Um *service mesh* também suporta a implementação e imposição de requisitos transversais, como segurança e confiabilidade (limitação de taxa, quebra de circuito). Como o *service mesh* está no caminho crítico para todas as solicitações tratadas no sistema, também pode fornecer "observabilidade" adicional, como rastreamento distribuído de uma solicitação, frequência de códigos de erro HTTP, latência global e serviço a serviço. (BRYANT, 30/07/2020)

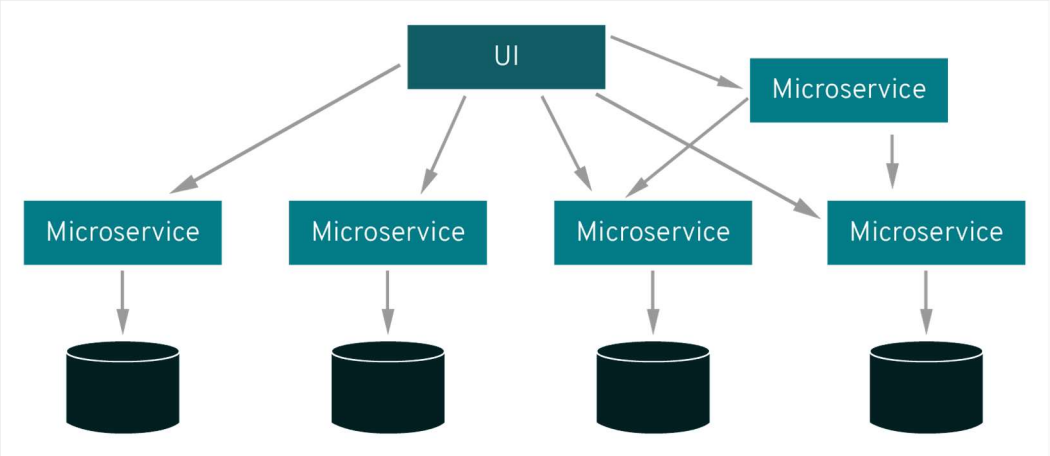

Figura 28 *- Comunicação entre microsserviços*

https://www.redhat.com/cms/managed-files/microservices-1680.png

Tolerância a falhas

Uma consequência do uso de serviços como componentes é que os aplicativos precisam ser projetados para que possam tolerar a falha desses serviços. Como os serviços podem falhar a qualquer momento, é importante ser capaz de detectar as falhas rapidamente e, se possível, restaurar o serviço automaticamente. As equipes de microsserviços esperam ver configurações sofisticadas de monitoramento e registro para cada serviço individual, como painéis que mostram o status ativo ou inativo e uma variedade de métricas operacionais e de negócios relevantes. (LEWIS e FOWLER, 25/03/2014)

Retry

Conforme Prashant (2021), o padrão *retry* possui uma ideia bastante simples. Se quem chama uma ação recebe uma resposta inesperada para uma requisição, o serviço que está chamando envia novamente a requisição para o serviço que está sendo requisitado. Se a requisição falhar devido a problemas ocasionais de rede, problemas de conexão, esse padrão pode ser bastante útil.

No entanto, solicitar novamente de forma indefinida, até receber uma resposta adequada, pode não ser uma boa ideia. Por isso é interessante estabelecer um número limite de re-tentativas. Uma outra técnica é estabelecer um tempo de atraso entre uma retentativa e outra, para dar a chance do sistema se restabelecer. Também é interessante determinar que somente ações que possuem uma chance de responder corretamente numa re-tentativa possam usar este mecanismo.

É importante desenhar processos que possam detectar duplicidade de chamadas e não causar inconsistência de dados.

Circuit Breaker

Para Fowler (2014), a ideia básica por trás do *circuit breaker* é muito simples. Você envolve uma chamada de função protegida em um objeto disjuntor, que monitora as falhas. Uma vez que as falhas atingem um certo limite, o disjuntor desarma e todas as chamadas adicionais para o disjuntor retornam com um erro, sem que a chamada protegida seja feita.

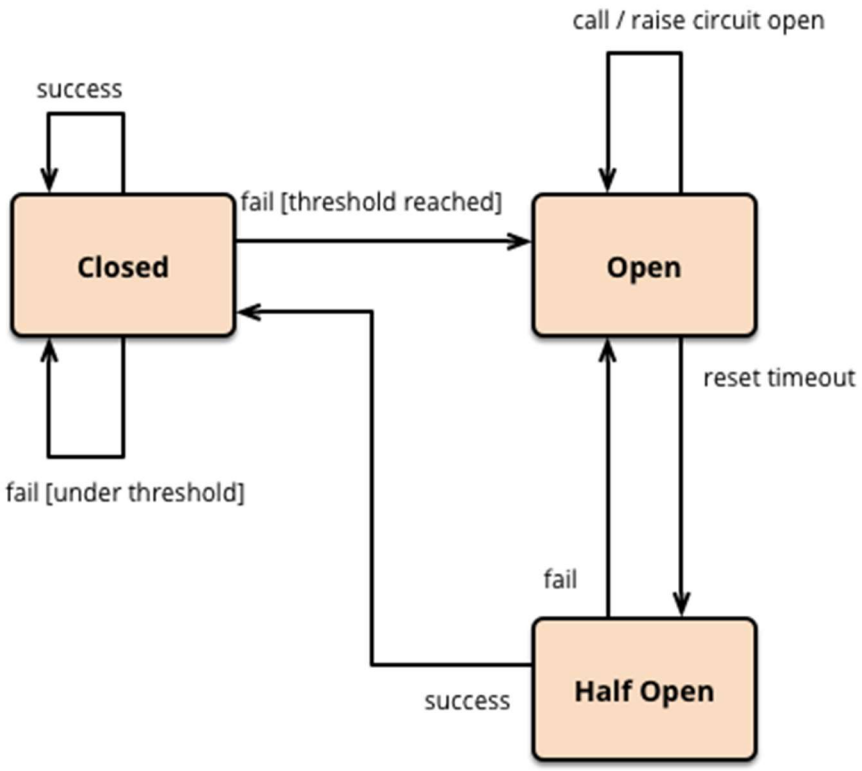

Figura 29 - Circuit Breaker

https://martinfowler.com/bliki/images/circuitBreaker/state.png

Este simples disjuntor evita fazer a chamada protegida quando o circuito está aberto, mas precisaria de uma intervenção externa para reiniciá-lo quando tudo estiver bem novamente. É possível implementar um comportamento de reinicialização automática tentando a chamada protegida novamente, após um intervalo adequado e reiniciando o disjuntor, caso seja bem-sucedido. Criar este tipo de disjuntor significa adicionar um limite para tentar o reset e configurar uma variável para conter a hora do último erro. Também existe um terceiro estado presente - meio aberto - o que significa que o circuito está pronto para fazer uma chamada real, com o intuito de verificar se o problema foi resolvido. No estado semi-aberto, uma chamada de teste é feita, e se for bem sucedida, fechará o disjuntor, se não, resetará o tempo de espera (FOWLER, 06/03/2014).

Bulkhead

Conforme escreveu Selvaraj (2020), um navio é dividido em pequenos compartimentos múltiplos usando anteparos. Anteparos são usados para selar partes do navio para evitar que todo o navio afunde em caso de inundação. Da mesma forma, falhas devem ser esperadas quando projetamos software. O aplicativo deve ser dividido em vários componentes e os recursos devem ser isolados de forma que a falha de um componente não afete o outro.

Usando o padrão *bulkhead*, alocamos um limite para os recursos de um componente específico, evitando consumir todos os recursos do aplicativo desnecessariamente. Nosso aplicativo permanece funcional mesmo sob carga inesperada.

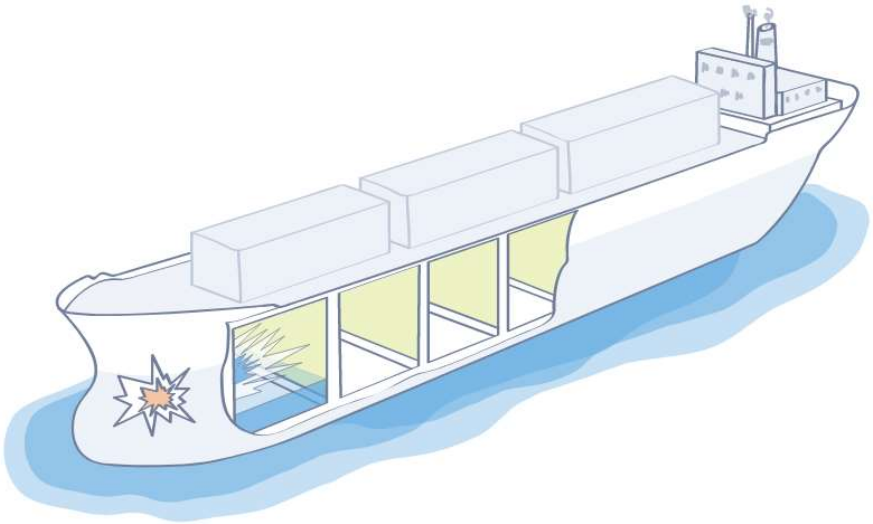

Figura 30 *- Bulkhead*

https://openliberty.io/guides/iguide-bulkhead/html/images/with-bulkhead.svg

Fallback

Para Seeley (2018), o padrão *Fallback* consiste em detectar um problema e, em seguida, executar um caminho de código alternativo. Esse padrão é usado quando o caminho do código original falha e fornece um mecanismo que permitirá ao cliente do serviço responder por meios alternativos. Outros caminhos podem incluir respostas estáticas, respostas em cache ou até mesmo serviços alternativos que fornecem informações

semelhantes. Depois que uma falha for detectada, talvez por meio de um dos outros padrões de resiliência, o sistema pode fazer fallback.

Timeout

De acordo com Selvaraj (2020), o padrão Timeout é uma das técnicas mais simples. Nós experimentamos lentidão intermitente de aplicativos de vez em quando, sem motivos óbvios. Isso pode acontecer com qualquer aplicativo, inclusive em grandes plataformas. Como não é um problema incomum, é melhor levar esse problema de indisponibilidade de serviço em consideração ao projetar os microsserviços.

Dessa forma, temos os seguintes benefícios:

- Fazer os serviços principais funcionarem conforme o esperado, mesmo quando os serviços dependentes não estiverem disponíveis.
- Não esperar uma resposta por tempo indeterminado
- Não bloquear o processamento
- Lidar com problemas de rede oferecendo uma resposta em cache.

Registro de Logs

Segundo Richardson (2019), logs são parte importante quando queremos saber o que há de errado com nossa aplicação. Mas é importante padronizar os registros para que possam ser consultados posteriormente. Existe o padrão *Log Agregation* que mantém todos os registros de todos os serviços em um banco de dados centralizado, que suporta consulta e alertas automáticos. Existem estruturas prontas em várias linguagens que lidam com armazenamento de logs, como por exemplo o combo ELK:

- Elasticsearch - Um banco de dados NoSQL orientado a pesquisa de textos, que é usado para armazenamento
- Logstash - Uma pilha de execução que organiza os logs e os envia para o Elasticsearch
- Kibana - Uma ferramenta de visualização para o Elasticsearch. (RICHARDSON, 2019, p. 369-370)

Tracing

Tracing consiste em atribuir a cada solicitação externa um ID exclusivo, bem como, um registro de como ele flui pelo sistema de um serviço para o outro, em um servidor centralizado, que fornece visualização e análise. Um registro típico contém o nome da aplicação, o ID do rastreamento, tempo de início e fim. (RICHARDSON, 2019, p. 371)

Conforme Ribenzaft (2019), *tracing* é uma forma de identificar e monitorar eventos em aplicativos. Com as informações certas, um rastreamento pode revelar o desempenho de operações críticas. O rastreamento distribuído é uma nova forma de rastreamento que se adaptou melhor a aplicativos baseados em microsserviços. Ele permite que os engenheiros vejam os rastros de ponta a ponta, localizem falhas e melhorem o desempenho geral. Em vez de rastrear o caminho em um único domínio de aplicativo, o rastreio distribuído segue uma solicitação do início ao fim. Existem ferramentas de código aberto que facilitam essa tarefa, como o projeto OpenTelemetry.

Métricas

De acordo com a organização Prometheus, métricas são medidas numéricas, e as séries temporais significam as mudanças que são registradas ao longo do tempo. O que os usuários desejam medir difere de

aplicativo para aplicativo. Para um servidor web, pode ser o número de solicitações, para um banco de dados pode ser o número de conexões ativas ou o número de consultas ativas, etc.

As métricas desempenham um papel importante na compreensão de por que seu aplicativo está funcionando de determinada maneira. Vamos supor que você esteja executando um aplicativo da web e descubra que ele está lento. Precisaremos de algumas informações para descobrir o que está acontecendo com seu aplicativo. Por exemplo, o aplicativo pode ficar lento quando o número de solicitações é alto. Se você tiver a métrica de contagem de requisições, poderá identificar o motivo e aumentar o número de servidores para lidar com a carga.

API GATEWAY

Uma abordagem para o *design* de uma API é os clientes invocarem os serviços diretamente. A princípio, isso parece adequado, mas essa abordagem raramente é usada em uma arquitetura de microsserviços devido às seguintes desvantagens:

- Os serviços são especialistas e podem exigir que os clientes façam várias solicitações para recuperar os dados de que precisam, o que é ineficiente e pode resultar em uma experiência de usuário insatisfatória.
- A falta de encapsulamento causada pelo conhecimento dos clientes sobre cada serviço e seus endpoints torna difícil mudar a arquitetura e a estrutura interna de cada serviço.
- Os serviços podem usar mecanismos de comunicação que não são convenientes ou práticos para os clientes usarem, especialmente aqueles clientes externos.
- Clientes diferentes podem ter largura de banda reduzida, como é o caso dos celulares e, portanto, deveriam ter uma resposta mais personalizada, para otimizar o desempenho. (RICHARDSON, 2019, p. 254-255)

O padrão de desenvolvimento *API Gateway* trata de implementar um serviço que é uma porta de entrada para os clientes externos. O *Api Gateway* lida com requisições simplesmente roteando para o serviço apropriado ou então distribui a chamada para vários serviços. Ao invés de fornecer um estilo de API único, o *API Gateway* pode expor uma API diferente para cada cliente.

Implementar um *API Gateway* oferece os seguintes benefícios:

- Isola os clientes de como o aplicativo é particionado em microsserviços;
- Isola os clientes do problema de determinar a localização das instâncias de serviço;
- Fornece a API ideal para cada cliente;
- Reduz o número de requisições de ida e volta. Por exemplo, o API Gateway permite que os clientes recuperem dados de vários serviços com uma única requisição externa. Menos requisições também significa menos sobrecarga e melhora a experiência do usuário. Um API Gateway é essencial para aplicativos móveis;
- Simplifica o cliente transferindo a responsabilidade de chamar vários serviços para o API Gateway;
- Traduz em um protocolo de API público amigável para a web para quaisquer protocolos usados internamente.

O padrão *API Gateway* tem algumas desvantagens:

- Maior complexidade - o API Gateway é mais uma parte que deve ser desenvolvida, implantada e gerenciada.
- Aumento do tempo de resposta devido a passagem adicional através do API Gateway - no entanto, para a maioria dos aplicativos, o custo dessa passagem é insignificante. (RICHARDSON, 2017)

Para a Red Hat, um *API gateway* faz parte do sistema de gerenciamento da API. Ele intercepta todas as solicitações de entrada e as envia por meio desse sistema, que processa diversas funções necessárias. Sendo esta porta de entrada, a função exata do *gateway* varia de uma implementação para outra. Algumas funções comuns incluem, roteamento, limitação de taxa, faturamento, monitoramento, análise, políticas, alertas e segurança.

Funções essenciais de um gateway

O gateway é responsável pelo roteamento da requisição, composição de API e tradução de protocolo. (RICHARDSON, 2019, p. 260)

Roteamento

Para Richardson (2019), uma das principais funções de um A*PI gateway* é o roteamento de requisições. O *gateway* implementa algumas operações de API roteando requisições para o serviço correspondente. Ao receber uma solicitação, o *gateway* consulta um mapa de roteamento que especifica para qual serviço rotear a solicitação.

Composição de API

Conforme Siahaan (2019), um *API gateway* também fornece o recurso de composição de API, que permite que determinado cliente recupere dados de forma eficiente usando uma única solicitação de API.

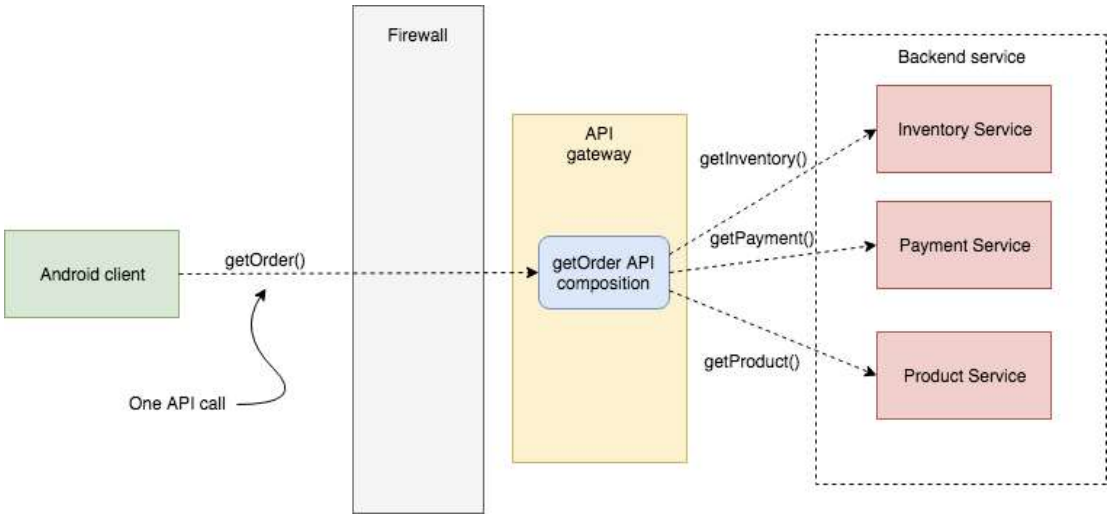

Figura 31 *- Composição de API*

https://miro.medium.com/max/766/1*xco_AbzG10GLrSSqDSqSFg.png

Tradução de protocolo

Um *gateway* também pode realizar a tradução de protocolo. Pode fornecer uma API RESTful para clientes externos, embora os serviços de aplicativo usem uma mistura de protocolos internamente, incluindo REST e gRPC. Quando necessário, a implementação de algumas operações da API traduz entre a API RESTful externa e a API interna baseada em gRPC. (RICHARDSON, 2019, p. 262)

Funções extras de um gateway

Um gateway também pode implementar funções relacionadas com a interceptação das requisições.

- Autenticação: Verifica a identidade do cliente em cada requisição.
- Autorização: Verifica se o cliente está autorizado a executar determinada operação.
- Limite de requisição: Limita o número de requisições por período de um cliente específico ou de todos os clientes.
- Cache: Mantém respostas em cache para reduzir o número de requisições feitas aos serviços.
- Captura de métricas: Colete métricas sobre o uso da API para fins de análise e de faturamento.
- Log de requisições: Registra o log das requisições. (RICHARDSON, 2019, p. 262)

Balanceamento de carga

Os balanceadores de carga são os responsáveis por rotear as requisições que vêm dos clientes para as instâncias do microsserviço requisitado, garantindo que nenhum servidor seja sobrecarregado, maximizando a velocidade e a capacidade de execução.

Se a instância de um determinado microsserviço cair, o balanceador de carga para de rotear requisições de clientes para esta instância. Da mesma forma, quando uma nova instância estiver disponível, o balanceador começa a repassar requisições para ela automaticamente. (OPUS SOFTWARE, 31/05/2021).

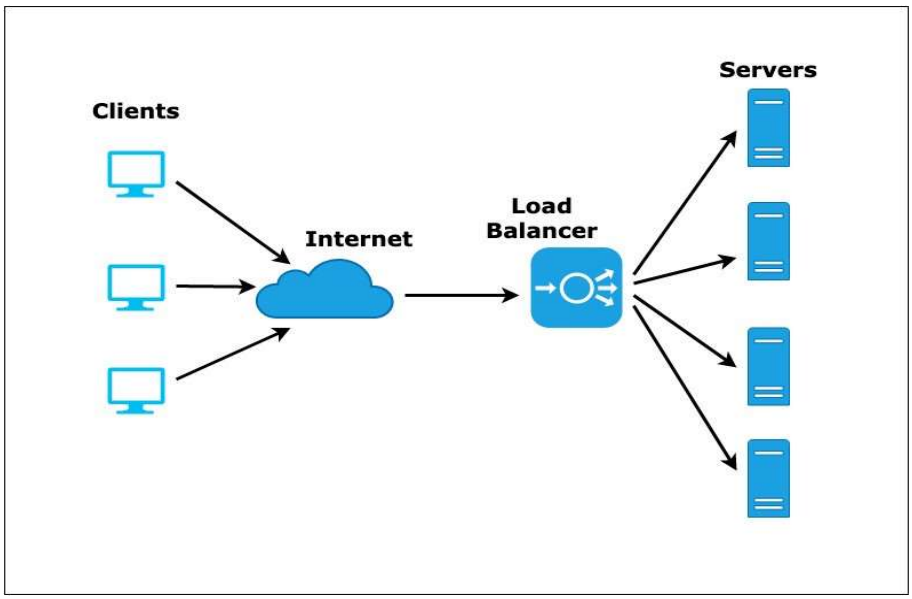

Figura 32 *- Load Balancer*

https://miro.medium.com/max/1400/1*tEaZGz-p1-E2ytNjl5RPJg.jpeg

APLICAÇÃO

Para demonstrar vários conceitos apresentados neste trabalho, um sistema foi desenvolvido usando o ambiente NodeJs e disponibilizado via API REST.

Visão geral do sistema

Este é um sistema fictício responsável pelo controle de fluxo de veículos em determinados locais cadastrados na aplicação. Os seguintes serviços foram implementados:

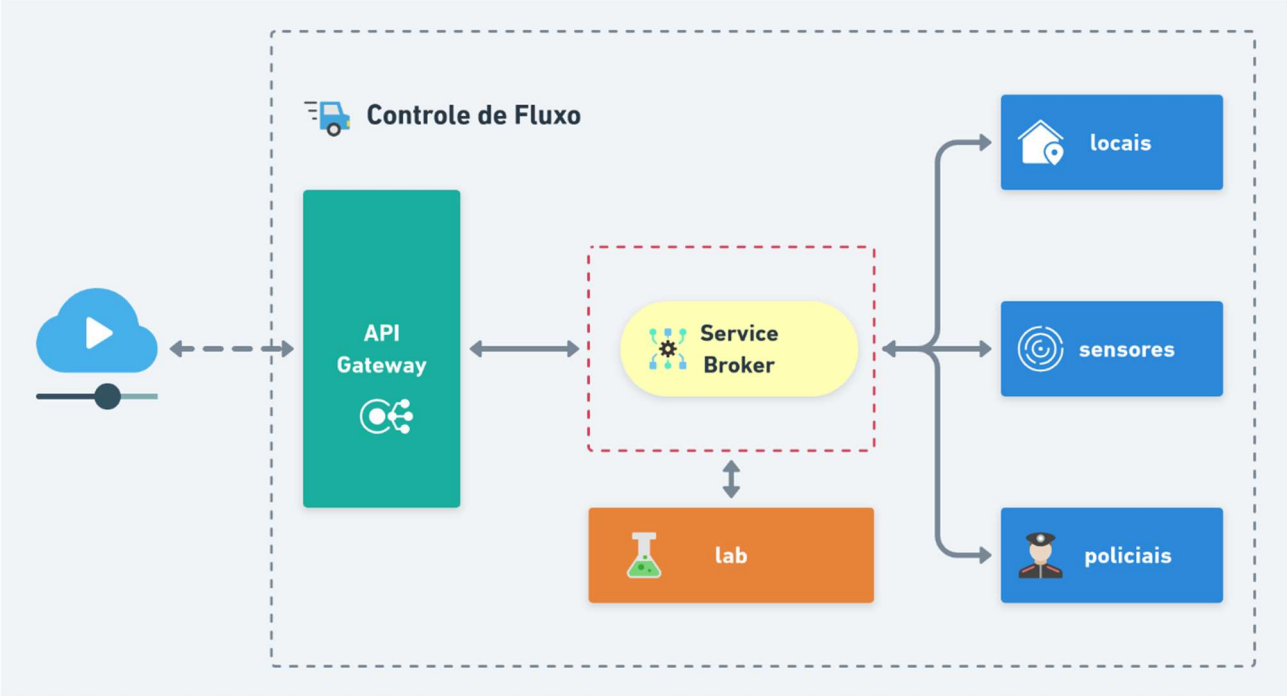

Figura 33 - *Controle de Fluxo*

Gerada pelo autor

- API: API *Gateway*, responsável por receber todas as requisições e direcionar para os serviços internos.
- Lab: Dashboard dos serviços, contendo métricas, logs e *tracings*.
- Locais: Responsável por cadastrar locais onde serão instalados os sensores de fluxo.
- Sensores: Responsável por cadastrar a passagem de um veículo, registrando a placa, o valor da tarifa e a velocidade. Se a velocidade for maior que 40 Km/h, um evento assíncrono "velocidade.alta" será emitido, informando os dados capturados pelo sensor.
- Policiais: Responsável por escutar se algum evento "velocidade.alta" for emitido e tomar as devidas providências.

Implementação local

Para reproduzir o cenário localmente, é necessário ter o ambiente NodeJs instalado. Em seguida o repositório git deve ser clonado, usando no terminal o seguinte comando:

git clone https://github.com/brasiqui/fluxo.git

Dentro da pasta criada, executar o comando npm install para instalar as dependências. Após a instalação, o comando npm run dev pode ser usado para iniciar a aplicação, e usando a url base http://localhost:3000/api, acessar os endpoints desenvolvidos.

Implementação em plataforma Cloud

Usando a tecnologia de containers, um cluster foi criado com *Docker Swarm*, que possibilita, através de arquivos de configuração, facilidade no *deploy*, escalabilidade e balanceamento de carga. O serviço

de sensores foi escalado com três instâncias e os demais com uma instância para cada um. Para a comunicação direta entre os serviços foi criada uma instância do *redis*, atuando como *service broker*. Portanto, a estrutura em *cloud* ficou dessa forma:

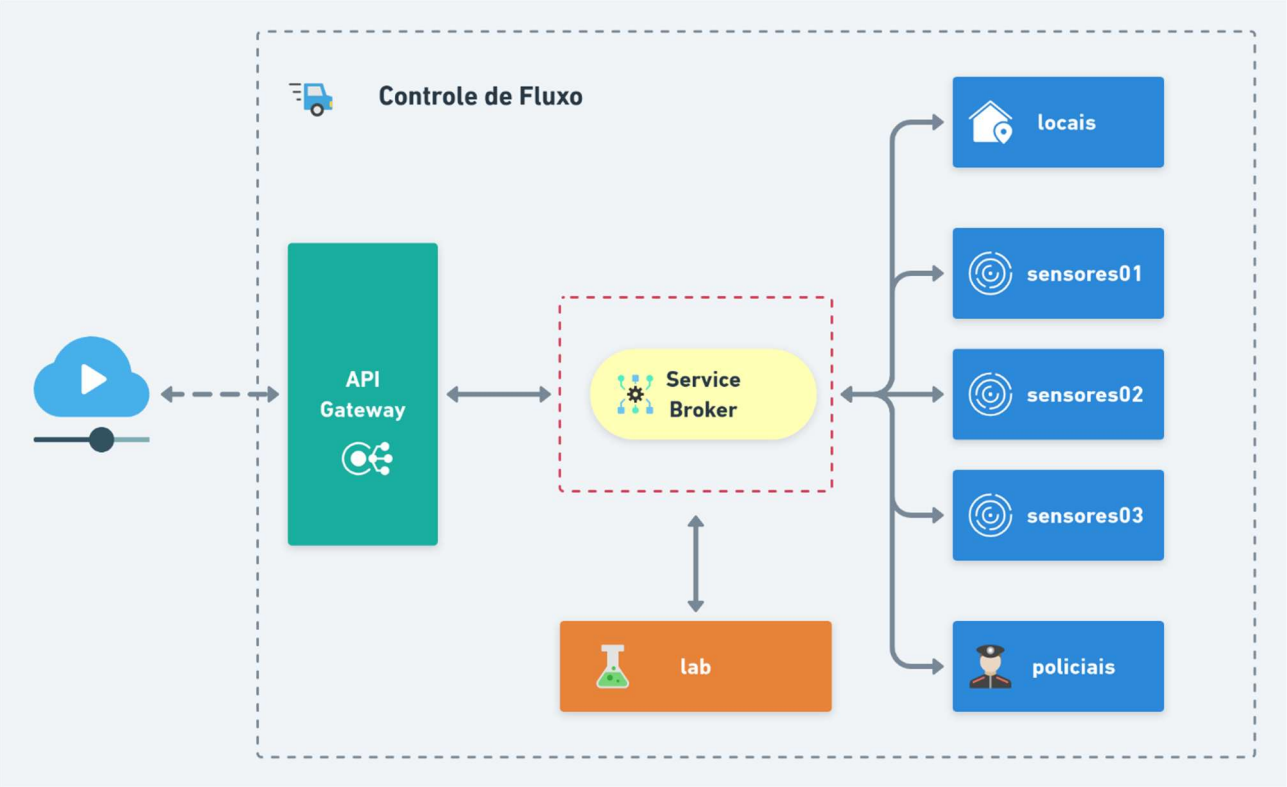

Figura 34 - Controle de Fluxo - Cloud

Gerada pelo autor

Demonstração

Cada serviço pode ter ações que são chamadas internamente ou expostas através de *endpoints*. Nos próximos tópicos, será exibido um exemplo das principais ações disponíveis no sistema.

Locais

Este serviço é responsável por cadastrar os locais e possui os seguintes *endpoints*:

- *Create* - POST /api/locais

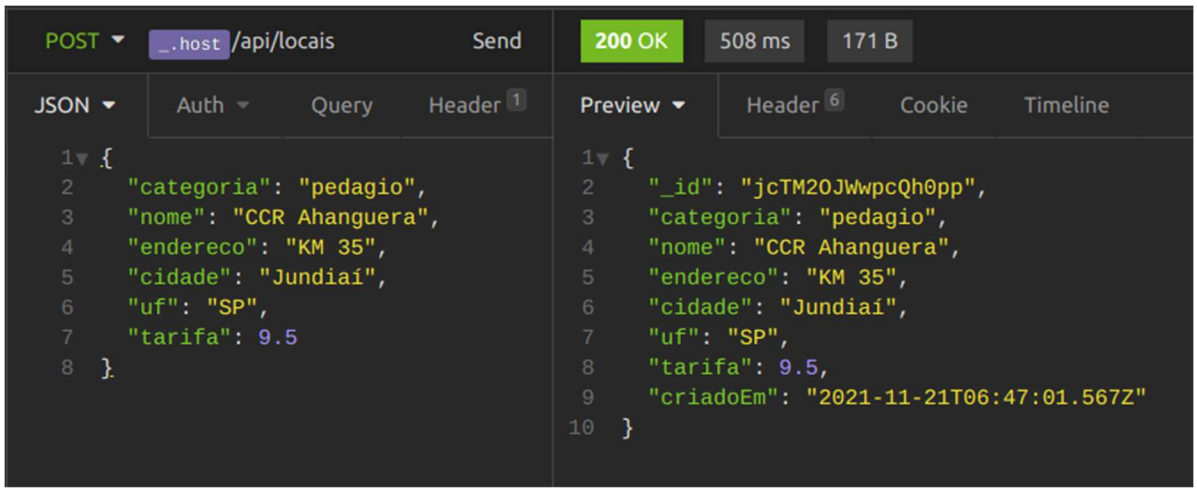

Figura 35 *- POST /api/locais*

Gerada pelo autor

- *Update* - PUT /api/locais/:id

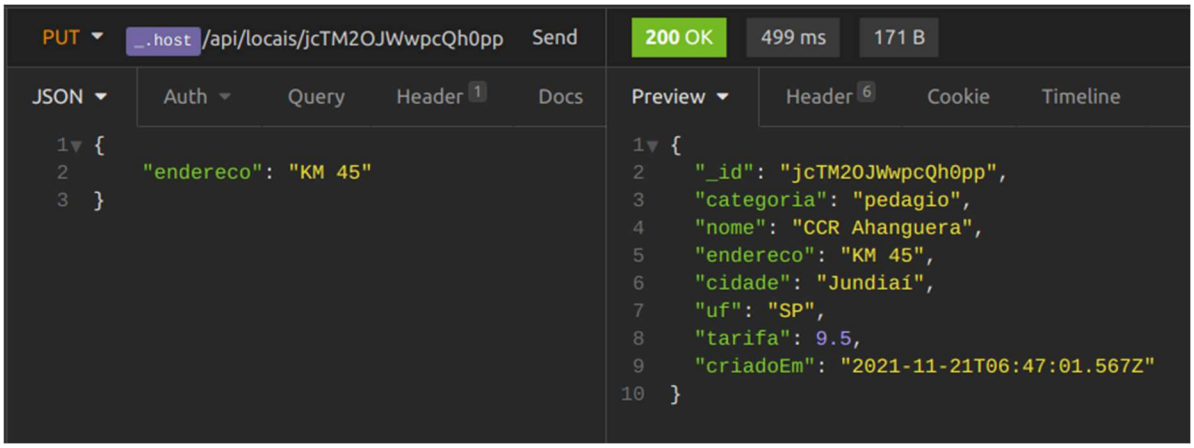

Figura 36 *- PUT /api/locais/:id*

Gerada pelo autor

- *Get* - GET /api/locais/:id

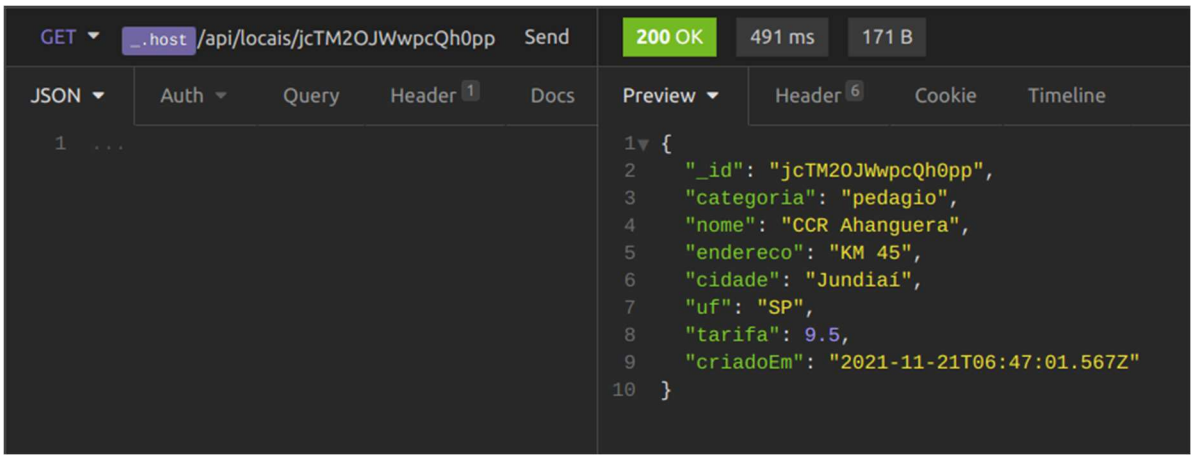

Figura 37 - GET /api/locais/:id

Gerada pelo autor

- *List* - GET /api/locais

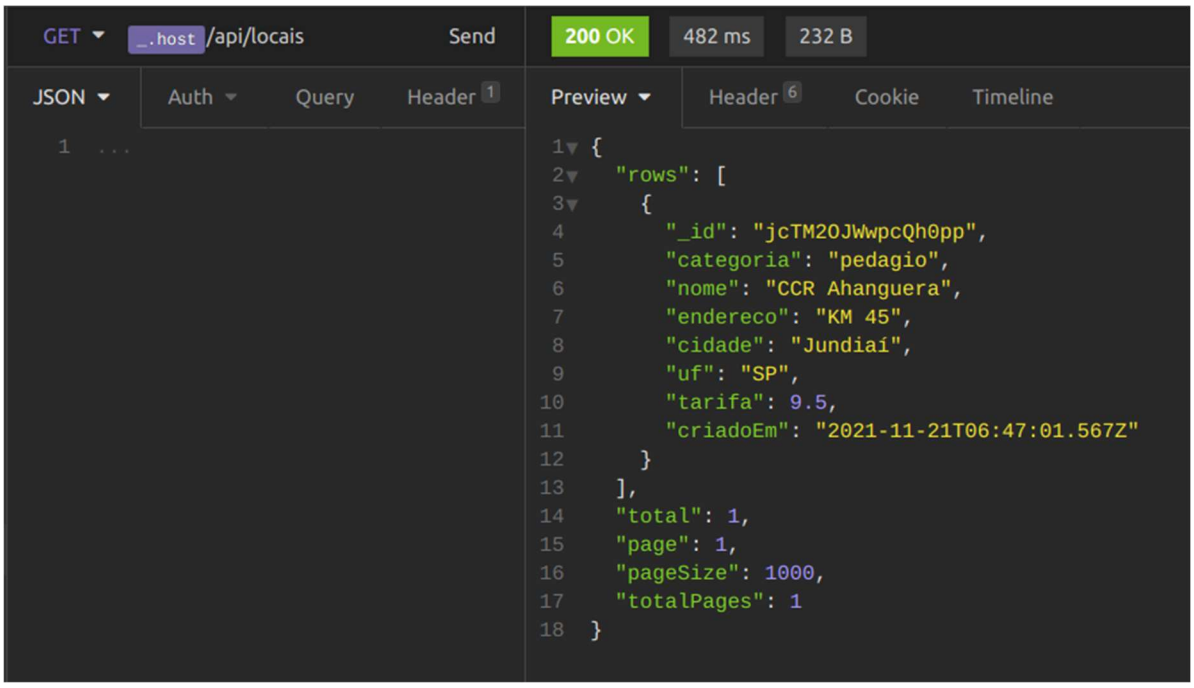

Figura 38 - GET /api/locais

Gerada pelo autor

- *Remove* - DELETE /api/locais/:id

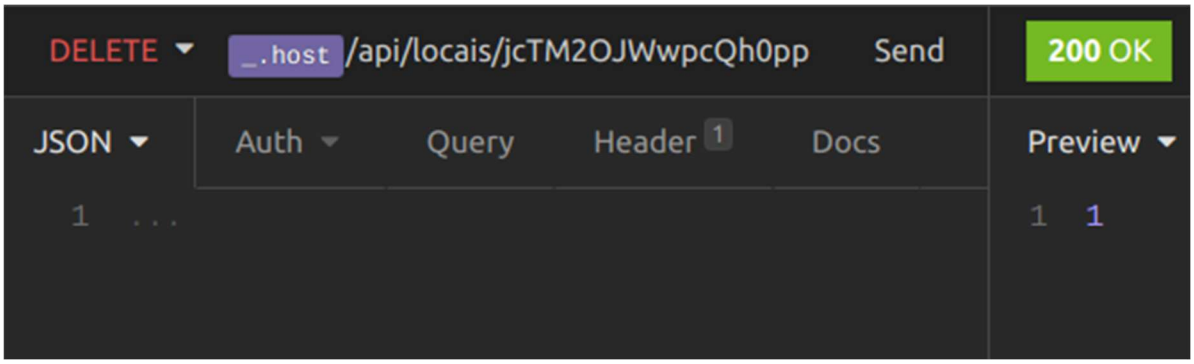

***Figura 39** - DELETE /api/locais/:id*

Gerada pelo autor

Sensores

Este serviço é responsável por cadastrar a passagem de um veículo e possui os seguintes endpoints, com formato similar ao anterior:

- *Create* - POST /api/sensores
- *Update* - PUT /api/sensores/:id
- *Get* - GET /api/sensores/:id
- *List* - GET /api/sensores
- *Remove* - DELETE /api/sensores/:id

Para testar o balanceamento de carga entre as três instâncias de sensores, uma ação fará o cadastro automático de uma passagem de veículo, utilizando dados aleatórios gerados pelo próprio sistema. Na resposta, virá o id do nó que respondeu a essa requisição. É possível notar que a cada execução, um nó diferente é acionado.

- *Seed* - POST /api/sensores/seed

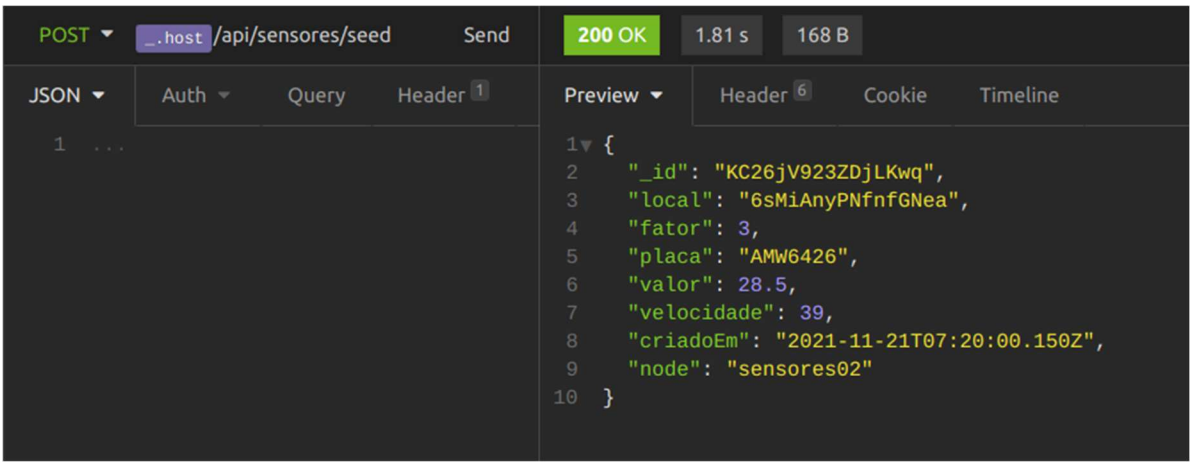

Figura 40 *– POST /api/sensores/seed*

Gerada pelo autor

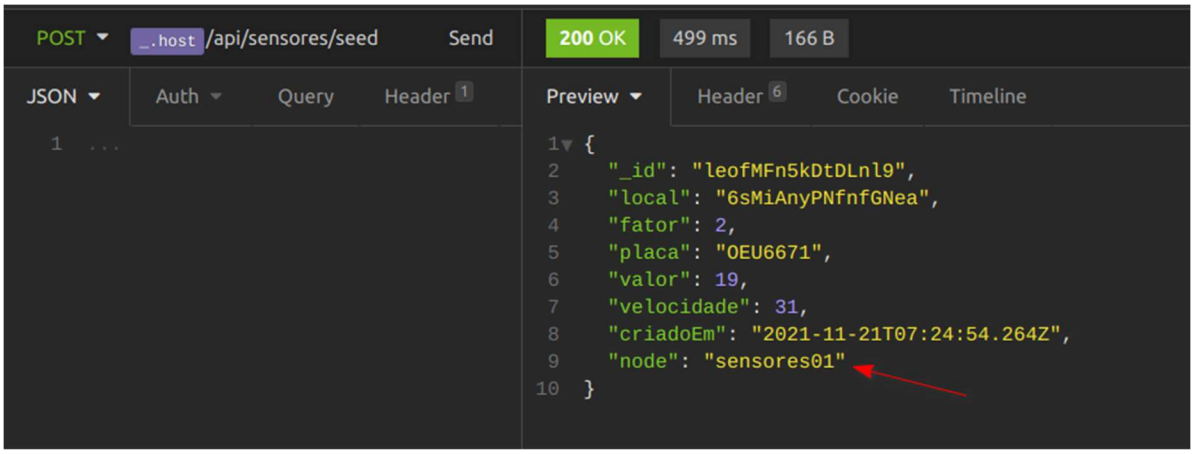

Figura 41 *- Balanceamento de sensores*

Gerada pelo autor

Policiais

Este serviço é responsável por escutar se algum evento "*velocidade.alta*" for emitido e tomar as devidas providências. No nosso caso, logo após cadastrar uma passagem de veículo acima de 40 Km/h, o serviço de sensores emite um evento que é recebido de forma assíncrona pelo serviço de policiais. Esta interação pode ser observada através dos logs do container.

```
[2021-11-21T07:26:25.389Z] WARN  policiais-19/POLICIAIS: Velocidade alta registrada.
[2021-11-21T07:26:25.389Z] WARN  policiais-19/POLICIAIS: Placa: FML9193
[2021-11-21T07:26:25.389Z] WARN  policiais-19/POLICIAIS: Velocidade: 49
[2021-11-21T07:26:25.395Z] INFO  policiais-19/TRACER: ┌──────────────────────────────────────
[2021-11-21T07:26:25.396Z] INFO  policiais-19/TRACER: │ ID: 714e9c82-dc04-475b-a964-f2cdf9f409eb
[2021-11-21T07:26:25.396Z] INFO  policiais-19/TRACER: ├──────────────────────────────────────
[2021-11-21T07:26:25.398Z] INFO  policiais-19/TRACER: │ Polícia Rodoviária
[2021-11-21T07:26:25.399Z] INFO  policiais-19/TRACER: └──────────────────────────────────────
```

Figura 42 *- Evento assíncrono*

Gerada pelo autor

Lab (Dashboard)

Este serviço é responsável por exportar informações do sistema de modo que possam ser lidas por outras aplicações. Assim, é possível visualizar métricas e informações do sistema em tempo real. Cada requisição executada grava logs e um *tracing* completo é registrado.

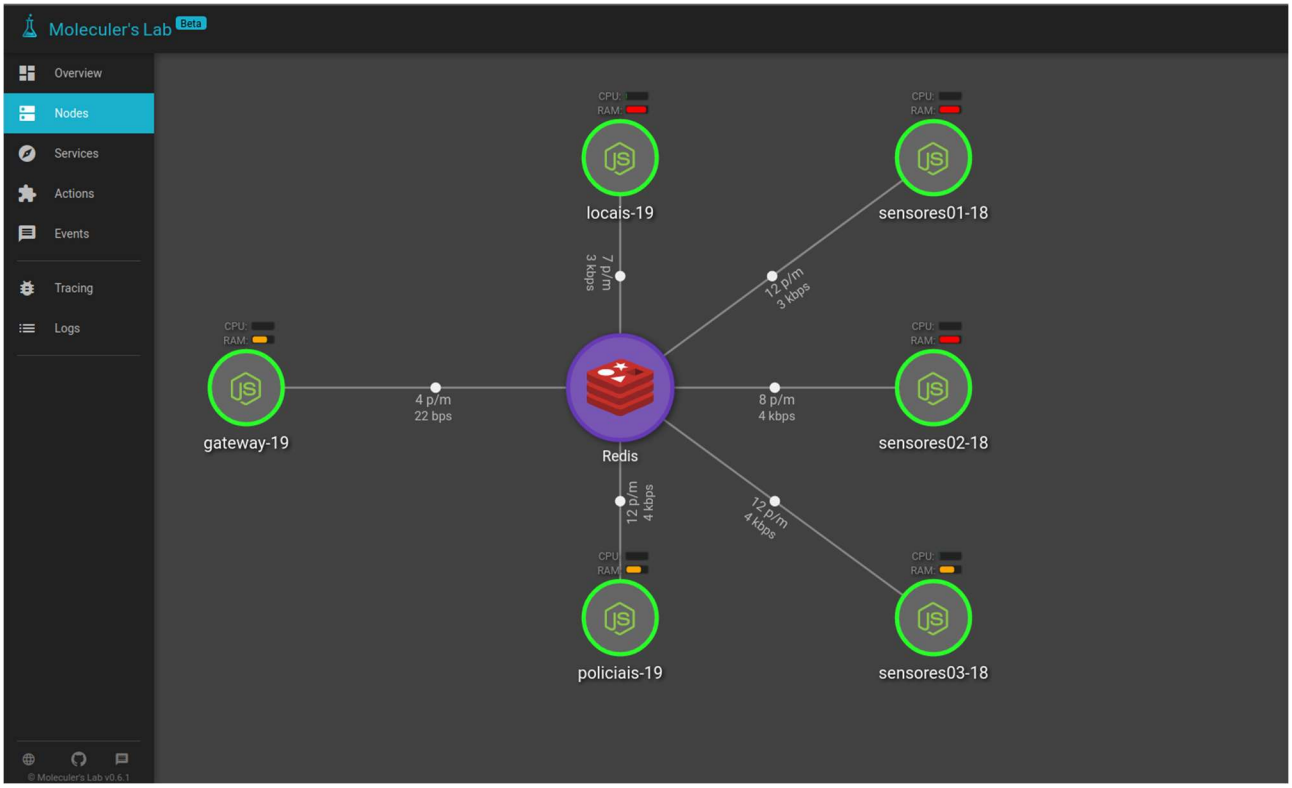

Figura 43 *- Dashboard*

Gerada pelo autor

CONCLUSÕES

Embora não exista uma definição formal para a arquitetura baseada em microsserviços, é possível identificar padrões de desenvolvimento essenciais, como divisão de responsabilidades, isolamento de independência, elementos de comunicação, tolerância a falhas e observabilidade.

Durante a implementação desses conceitos, foi possível notar que para implantar um sistema baseado em microsserviços, é necessário ter um conjunto de recursos que gerenciem essa plataforma de desenvolvimento, dando suporte à entrega e melhorias contínuas.

A adoção da arquitetura de microsserviços não é recomendada para todos os projetos, pois exige uma maturidade no desenvolvimento de aplicações e infraestruturas. Em muitos casos será interessante começar de forma mais simples e quando for necessário, aplicar de forma gradual os conceitos aqui apresentados. Em sistemas grandes e complexos, a arquitetura de microsserviços permite construir sistemas distribuídos altamente escaláveis, simplificando o desenvolvimento, homologação e implantações automatizadas, dando flexibilidade para adoção de novas tecnologias e entregando produtos mais confiáveis.

REFERÊNCIAS

AMAZON AWS. **Microsserviços**, 2021. Disponivel em: <https://aws.amazon.com/pt/microservices/>. Acesso em: 28 set. 2021.

PRASHANT, A.. **Retry pattern in microservices**, 26/01/2021. Disponivel em: <https://engineering.mercari.com/en/blog/entry/20210126-retry-pattern-in-microservices/>. Acesso em: 17 out. 2021.

BRYANT, D. **Service Mesh guia final: Gerenciando as comunicações serviço a serviço na era dos microservices**, 30/07/2020. Disponivel em: <https://www.infoq.com/br/articles/service-mesh-ultimate-guide/>. Acesso em: 17 out. 2021.

FOWLER, M. **CircuitBreaker**, 06/03/2014. Disponivel em: <https://martinfowler.com/bliki/CircuitBreaker.html>. Acesso em: 17 out. 2021.

FOWLER, M. **StranglerFigApplication**, 2004. Disponivel em: <https://martinfowler.com/bliki/StranglerFigApplication.html>. Acesso em: 17 out. 2021.

GONÇALVES, M. M. **Arquitetura de Microsserviços**, 01/06/2020. Disponivel em: <https://medium.com/@marcelomg21/arquitetura-de-microsservi%C3%A7os-bc38d03fbf64>. Acesso em: 17 out. 2021.

IBM REDBOOKS. **Microservices from Theory to Practice**. [S.l.]: IBM Redbooks, 2015. ISBN 0738440817.

LEWIS, J.; FOWLER, M. **Microservices - a definition of this new architectural term**, 25/03/2014. Disponivel em: <https://martinfowler.com/articles/microservices.html>. Acesso em: 19 set. 2021.

MOLECULERJS. **Core Concepts**, 2021. Disponivel em: <https://moleculer.services/docs/0.14/concepts.html>. Acesso em: 17 out. 2021.

OPUS SOFTWARE. **Como criar o ecossistema de ideal para a arquitetura de microsserviços**, 31/05/2021. Disponivel em: <https://www.opus-software.com.br/arquitetura-de-microsservicos/>. Acesso em: 18 out. 2021.

PENNA, W. **Arquitetura Monolítica e Microsserviços**, 13/05/2020. Disponivel em: <https://www.zappts.com/blog/arquitetura-monolitica-e-microsservicos/>. Acesso em: 28 set. 2021.

PERRY, D. E.; WOLF, A. L. Foundations for the Study of Software Architecture. **SOFTWARE ENGINEERING NOTES**, 17, 1992. 40-53.

PEYROTT, S. **An Introduction to Microservices, Part 3: The Service Registry**, 02/10/2015. Disponivel em: <https://auth0.com/blog/an-introduction-to-microservices-part-3-the-service-registry/>. Acesso em: 18 out. 2021.

PRATES, B. G. **Strangler Pattern: como migrar um monólito para microsserviços**, 08/03/2021. Disponivel em: <https://imasters.com.br/apis-microsservicos/strangler-pattern-migrar-monolito-para-microsservicos>. Acesso em: 17 out. 2021.

PROMETHEUS. **Overview - Prometheus**. Disponivel em: <https://prometheus.io/docs/introduction/overview/>. Acesso em: 18 out. 2021.

RED HAT. **O que é arquitetura orientada a serviços (SOA)?**, 27/07/2020. Disponivel em: <https://www.redhat.com/pt-br/topics/cloud-native-apps/what-is-service-oriented-architecture>. Acesso em: 08 set. 2021.

RED HAT. **O que é service mesh?** Disponivel em: <https://www.redhat.com/pt-br/topics/microservices/what-is-a-service-mesh>. Acesso em: 17 out. 2021.

RED HAT. **O que são os microsserviços?** Disponivel em: <https://www.redhat.com/pt-br/topics/microservices/what-are-microservices>. Acesso em: 19 set. 2021.

RED HAT. **Qual é a função de um gateway de API?** Disponivel em: <https://www.redhat.com/pt-br/topics/api/what-does-an-api-gateway-do>. Acesso em: 18 out. 2021.

RIBENZAFT, R. **Microservices: Using Distributed Tracing for Monitoring & Troubleshooting**, 12/07/2019. Disponivel em: <https://cloudacademy.com/blog/microservices-using-distributed-tracing-monitoring-troubleshooting/>. Acesso em: 18 out. 2021.

RICHARDSON, C. **Pattern: API Gateway**, 2017. Disponivel em: <https://microservices.io/patterns/apigateway.html>. Acesso em: 18 out. 2021.

RICHARDSON, C. **Pattern: Strangler application**, 2017. Disponivel em: <https://microservices.io/patterns/refactoring/strangler-application.html>. Acesso em: 17 out. 2021.

RICHARDSON, C. **Microservices Patterns**. 1. ed. Shelter Island, NY: Manning Publications, 2019. ISBN 9781617294549.

RICHARDSON, C. **Pattern: Monolithic Architecture**, 2019. Disponivel em: <https://microservices.io/patterns/monolithic.html>. Acesso em: 19 set. 2021.

SEELEY, L. **4 Resiliency Patterns Leveraged by Capital One's Mobile Edge Engineering Team**, 19/09/2018. Disponivel em: <https://medium.com/capital-one-tech/resiliency-patterns-at-the-edge-capital-one-a5b4d41d477e>. Acesso em: 17 out. 2021.

SELVARAJ, V. **Resilient Microservice Design – Bulkhead Pattern**, 04/11/2020. Disponivel em: <https://dzone.com/articles/resilient-microservices-pattern-bulkhead-pattern>. Acesso em: 17 out. 2021.

SELVARAJ, V. **Timeout Pattern — Resilient Microservice Design With Spring Boot**, 25/10/2020. Disponivel em: <https://vinsguru.medium.com/resilient-microservice-design-with-spring-boot-timeout-pattern-72b5f5174d2a>. Acesso em: 17 out. 2021.

SIAHAAN, J. N. **API Gateway Part 2**, 07/09/2019. Disponivel em: <https://medium.com/easyread/api-gateway-part-2-7264ee5be187>. Acesso em: 18 out. 2021.

SILVA, A. F. **Alcançando a excelência do REST com um Modelo de Maturidade eficiente**, 02/10/2017. Disponivel em: <https://mundoapi.com.br/destaques/alcancando-a-excelencia-do-rest-com-um-modelo-de-maturidade-eficiente/>. Acesso em: 17 out. 2021.