UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

Características, boas práticas e soluções no desenvolvimento de aplicações com arquitetura de microserviços

Trabalho de Conclusão de Curso

João Paulo Feitosa Secundo

São Cristóvão – Sergipe

2022

UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

João Paulo Feitosa Secundo

Características, boas práticas e soluções no desenvolvimento de aplicações com arquitetura de microserviços

Trabalho de Conclusão de Curso submetido ao Departamento de Computação da Universidade Federal de Sergipe como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador(a): Rafael Oliveira Vasconcelos

São Cristóvão – Sergipe

2022

Lista de abreviaturas e siglas

API	Application Programing Interface - Interface para programação de aplicação
HTTP	HyperText Transfer Protocol - Protocolo de transferência de hipertexto
AMS	Arquitetura de microsserviços
DoS	Denial of Service - Negação de serviço
RFC	Request For Comments - (Não traduzido)
JSON	JavaScript Object Notation - Notação de objeto javaScript

Sumário

1 Introdução

Todos que têm contato com o ramo do desenvolvimento de software provavelmente já ouviu o termo *SaaS (Software as a Service)*, ou software como um serviço. Mas ao contrário do que alguns pensam, essa expressão é mais do que apenas um modelo de negócio. O crescimento da internet e a onipresença da computação móvel tem mudado o jeito como software é desenvolvido nos últimos tempos. A tendência que tem-se observado é a de oferecer software não mais como um pacote completo e fechado, mas sim como um pacote flexível e em constante melhoria, o que implica na mudança do foco dos desenvolvedores para construir componentes leves e auto-contidos, que permitam que mudanças sejam desenvolvidas e implantadas rápida e independetemente. A partir disso originou-se um novo paradigma de desenvolvimento, chamado de "microsserviços". (Middleware Lab, 2021).

> Se você quiser projetar um aplicativo que seja multilíngue, facilmente escalável, fácil de manter e implantar, altamente disponível e que minimize falhas, use a arquitetura microservices para projetar e implantar um aplicativo em nuvem. (Oracle Corporation, 2021)

Esse estilo de arquitetura de software é amplamente considerado a melhor maneira de estruturar um sistema de software como um serviço. (XU et al., 2016)

Nesse trabalho serão discutidos as características da arquitetura de microsserviços, as boas práticas que devem ser seguidas no desenvolvimento de aplicações com essa arquitetura, e as (soluções ou ferramentas?) mais usadas.

1.1 Objetivos

1.1.1 Objetivo geral

Discutir a arquitetura de microsserviços e suas características. Analisar as boas práticas e soluções mais usadas no desenvolvimento de aplicações que utilizam essa arquitetura. Modelar e implementar um exemplo de aplicação usando a arquitetura de microsserviços.

1.1.2 Objetivos específicos

- Caracterizar a arquitetura de microsserviços;

- Reunir boas práticas usadas na implementação de aplicações com arquitetura de microsserviços;

- Reunir ferramentas usadas na implementação de aplicações com arquitetura de microsserviços;

- Propor ideias e passos para como migrar da arquitetura monolítica para a de microsserviços;

Analisar a eficiência dessas boas práticas;

Analisar a eficiência dessas ferramentas;

- Combinar algumas das ferramentas propostas para implementar uma aplicação exemplar com arquitetura de microsserviços, usando as boas práticas discutidas.

1.2 Metodologia

Para caracterizar a arquitetura de microsserviços, foram pesquisados as seguintes termos nas bases científicas ScienceDirect, SpringerLink e GoogleScholar:

- (microservices or microservice) and pattern

- (microservices or microservice) and provision

A partir dos principais resultados obtidos com essas buscas e os trabalhos em respectivas referências bibliográficas, foram extraídas as características que todo microsserviço deve ter.

(metodologia quanto aos objetivos, quanto a execução. passo a passo que vai seguir durante o trabalho. explicar como analisar/testar essa eficiencia (objetivos especificos))

2

Fundamentação teórica

Este capítulo aprensenta uma introdução sobre as arquiteturas monolítica e de microsserviços, e investiga trabalhos relacionados.

2.1 As aplicações monolíticas

Aplicações monolíticas, também chamadas de monólitos, são aplicações que possuem as camadas de acesso aos dados, de regras de negócios, e de interface de usuário em um único programa em uma única plataforma. Os monólitos são autocontidos e totalmente independentes de outras aplicações. Eles são feitos não para uma tarefa em particular, mas sim para serem responsáveis por todo o processo para completar determinada função. Em outras palavras, as aplicações monolíticas têm problema de modularidade. Elas podem ser organizadas das mais variadas formas e fazer uso de padrões arquiteturais, mas são limitadas em muitos outros aspectos, citados na subseção 2.1.2.

2.1.1 Benefícios

O maior e melhor benefício da arquitetura monolítica é sua simplicidade. Uma aplicação simples é uma aplicação fácilmente entendida pelos seus desenvolvedores, o que melhora sua manutenibilidade. Para aplicações com um domínio simples, como um e-commerce de calçados por exemplo, optar por uma arquitetura complexa como a de microsserviços significaria adicionar uma enorme complexidade - provavelmente desnecessária - em seu desenvolvimento e infraestrutura.

Outra vantagem dos monólitos é sua facilidade de construção, tanto em relação a sua infraestrutura quanto ao seu desenvolvimento. Dentre todos os tipos de arquitetura, os monólitos têm o tipo de infraestrutura mais fácil de se construir, e além disso, neles geralmente não é necessário haver comunicação entre diferentes serviços ou máquinas, então os desenvolvedores não precisarão se preocupar com a complexidade que acompanha essa comunicação.

Até certo tamanho, são fáceis de manter porque são fáceis de serem entendidos. Porém, depois de crescer excessivamente, um monólito pode se tornar um emaranhado complexo de funcionalidades que são difíceis de diferenciar, de separar, e de manter. E então começam a surgir as limitações deles...

2.1.2 Limitações

Crescimento, velocidade de desenvolvimento, e manutenção

Depois de chegar num certo tamanho, torna-se muito difícil desenvolver funcionalidades novas, ou mesmo prover manutenção às já existentes. Padrões de organização podem amenizar a situação, mas não eliminam o problema.

Confiabilidade

Escalabilidade

Reutilização

Implantação

Necessidade de compilar toda a aplicação, mesmo as partes em que não houve mudanças, a cada implantação.

Resiliência

Falhas relativamente pequenas podem prejudicar toda a aplicação, mesmo as partes que não tiveram relação com a falha.

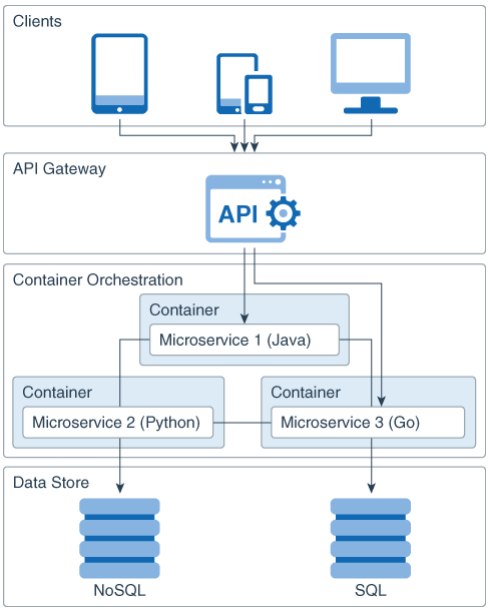
Flexibilidade

As escolhas de tecnologias são mais limitadas. Um projeto tende a usar apenas 1 solução devido a problemas de complexidade ou compatibilidade que podem surgir ao usar mais.

2.2 Os microsserviços

Microsserviços é uma abordagem de arquitetura de software. Aplicações com uma arquitetura de microsserviços são separadas em partes, chamadas de microsserviços, que são classificadas e se comunicam por meio de uma rede. Microsserviços oferecem capacidades de negócio (funcionalidades relacionadas às regras de negócio da aplicação) ou capacidades de plataforma (funcionalidades relacionadas ao ambiente de execução da aplicação), tratando um aspecto em particular da aplicação. Eles se comunicam por meio de APIs bem definidas, contratos de dados, e configurações. O "micro"em microsserviços faz referência não ao tamanho do serviço, mas sim ao seu escopo de funcionalidade. Eles oferecem apenas uma determinada funcionalidade,

Figura 1 – Aplicação com arquitetura de microsserviços



Fonte: Oracle Corporation (2021)

tornando-se especialistas nela. Assim sendo, microsserviços não necessariamente devem ser pequenos em tamanho, mas fazem apenas uma tarefa e a fazem eficientemente. (FAMILIAR, 2015)

Sendo especialistas em apenas uma tarefa, microsserviços têm características e comportamentos que os diferenciam de outras arquiteturas orientadas a serviços, os quais serão discutidos no Capítulo 3.

A Figura 1 exemplifica uma aplicação com arquitetura de microserviços. Inicialmente os usuários da aplicação (camada *Clients*) fazem requisições à API para obter as informações desejadas. O *API Gateway* - que é reponsável por integrar os serviços e será melhor discutido no Capítulo 4 - fará as devidas requisições para os devidos microsserviços (localizados na camada *Container Orchestration*). Esses microsserviços então buscarão a informação necessária no devido banco de dados (camada *Data Store*).

2.3 Trabalhos relacionados

2.3.1 Microservices, IoT and Azure, por Bob Familiar - capítulo 2: What is a microservice

O capítulo 2 do livro de Bob Familiar descreve o que é um microsserviço, suas características e implicações, benefícios, e desafios.

"Microservices do one thing and they do it well". Como é explicado por Familiar (2015) , microsserviços representam business capabilities definidos usando o design orientado a domínio, são testados a cada passo do *pipeline* de implantação, e lançados por meio de automação, como serviços independentes, isolados, altamente escaláveis e resilientes em uma infraestrutura em núvem distribuída. Pertecem a um time único de desenvolvedores, que trata o desenvolvimento do microsserviço como um produto, entregando software de alta qualidade em um processo rápido e iterativo com envolvimento do cliente e satisfação como métrica de sucesso.

2.3.2 A Systematic Mapping Study on Microservices Architecture in DevOps, por Waseem, M., Liang, P. e Shahin, M.

Esse trabalho tem o objetivo de sistematicamente identificar, analisar, e classificar a literatura sobre microsserviços em DevOps.

Inicialmente o leitor é contextualizado no mundo dos microsserviços e a cultura DevOps. Os autores usam a metodologia de pesquisa de um estudo de mapeamento sistemático da literatura publicada entre Janeiro de 2009 e Julho de 2018. Após selecionados 47 estudos, é feita a classificação deles de acordo com os critérios definidos pelos autores, e então é feita a discussão sobre os resultados obtidos - são expostos a quantidade de estudos sobre determinados tópicos em microsserviços, problemas e soluções, desafios, métodos de descrição, design patterns, benefícios, suporte a ferramentas, domínios, e implicações para pesquisadores e praticantes.

Os principais resultados são: (1) São identificados Três temas de pesquisa em AMS com DevOps “desenvolvimento e operações de microsserviços em DevOps”, “abordagens e suporte a ferramentas para sistemas baseados em AMS em DevOps”, e “Experiência de migração de AMS em DevOps”. (2) São identificados 24 problemas e apontadas suas respectivas soluções com respeito a implementação de microsserviços com DevOps. (3) A AMS é descrita principalmente usando caixas e linhas. (4) A maioria das qualidades da AMS são afetadas positivamente quando aplicadas com DevOps. (5) 50 ferramentas que suportam a construção de sistemas baseados em AMS são apontados. (6) A combinação da AMS e DevOps tem sido aplicada em uma ampla variedade de domínios de aplicações.

(Comparar cada trabalho com o meu trabalho. Coisas que eles não abordam e que eu abordo)

3
Características

Este capítulo aprensenta as propriedades e as vantagens dos microsserviços, assim como os desafios que acompanham suas implementações.

3.1 Propriedades

3.1.1 Autonomia e Isolamento

Autonomia e isolamento significa que microsserviços são unidades auto-contidas de funcionalidade com dependências de outros serviços fracamente acopladas e são projetados, desenvolvidos, testados e lançados independentemente. (FAMILIAR, 2015). O termo autônomo pode ser definido como - existe ou é capaz de existir independetemente das outras partes. O termo isolado, como - separado das outras partes.

3.1.2 Elasticidade, resiliência, e responsividade

Microsserviços são reusados entre muitas soluções diferentes e portanto devem ser escaláveis de acordo com o uso. Devem ser resilientes, isso é, ser tolerantes a falhas e ter um tempo de recuperação razoável quando algo der errado. Além disso, devem ser responsivos, tendo um desempenho razoável de acordo com o uso. (FAMILIAR, 2015) O termo elástico pode ser definido como - capaz de retornar ao tamanho/formato original depois de ser esticado, comprimido ou expandido. O termo resiliente, como - resistente às mudanças negativas. O termo responsivo, como - Rápido em responder e reagir.

3.1.3 Orientação-a-mensagens e programabilidade

Microsserviços dependem de APIs e contratos de dados para definir como interagir com o serviço. A API define um conjunto de endpoints acessíveis por rede, e o contrato de dados define a estrutura da mensagem que é enviada ou retornada. (FAMILIAR, 2015). O termo orientado-a-mensagens pode ser definido como - Software que conecta sistemas separados em

uma rede, carregando e distribuindo mensagens entre eles. O termo programável, como - Obedece a um plano de tarefas que são executadas para alcançar um objetivo específico.

3.1.4 Configurabilidade

Microsserviços devem provêr mais do que apenas uma API e um contrato de dados. Para que seja reusável e para que possa resolver as necessidades do sistema que o use, cada microsserviço tem níveis diferentes de configuração, e esta configuração pode ser feita de diferentes formas. (FAMILIAR, 2015). O termo configurável pode ser definido como - Projetado ou adaptado para formar uma configuração ou para algum propósito.

3.1.5 Automação

O ciclo de vida de um microsserviço deve ser totalmente automatizado, desde o design até a implantação. O termo automatizado pode ser definido como - Funcionar sem precisar ser controlado diretamente.

3.2 Vantagens

3.2.1 Evolução

Quanto maior e mais antigo o software, mais difícil é de dar manutenção, e monólitos envelhecem com maior velocidade do que microsserviços. Entretanto, é possível migrar de um sistema monolítico para a arquitetura de microsserviços aos poucos, um serviço por vez, identificando capacidades de negócio, implementando-as como um microsserviço, e integrando com uso de padrões de baixo acoplamento. Ao longo do tempo, mais e mais funcionalidades podem ser separadas e implementadas como microsserviço, até que o núcleo da aplicação monolítica se transforme em apenas um outro serviço, ou um microsserviço. (FAMILIAR, 2015)

3.2.2 Possibilidade de uso de diferentes ferramentas

Cada microsserviço disponibiliza suas funcionalidades por meio de APIs e contratos de dados em uma rede. A comunicação independe da arquitetura que o microsserviço faz uso, então cada microsserviço pode escolher seu sistema operacional, linguagem e banco de dados. Isso é especialmente valioso para times com dificuldade de comunicação, pois cada time precisa apenas de conhecimento da arquitetura do microsserviço em que trabalha. (FAMILIAR, 2015)

3.2.3 Alta velocidade

Com um time responsável por cuidar do ciclo de desenvolvimento e sua automação, a velocidade com que microsserviços podem ser desenvolvidos é muito maior do que fazer o

equivalente para uma solução monolítica. (FAMILIAR, 2015)

3.2.4 Reusável e combinável

Microsserviços são reusáveis por natureza. Eles são entidades independentes que provêm funcionalidades em um determinado escopo por meio de padrões de internet aberta. Para criar soluções para o usuário final, multiplos microsserviços podem ser combinados. (FAMILIAR, 2015)

3.2.5 Flexibilidade no ambiente de execução

A implantação de microsserviços é altamente dependente de sua automação. Para garantir flexibilidade de ambiente de execução, essa automação pode incluir configuração de cenários diferentes de uso, não apenas para produção, mas também para desenvolvimento e testagem, possibilitando que o microsserviço tenha o melhor desempenho em diversos cenários. Para tanto, é necessário o uso de ferramentas que configurem essa flexibilidade. (FAMILIAR, 2015). Tais ferramentas serão melhor discutidas no Capítulo 5.

3.2.6 Flexibilidade na escolha de tecnologias

Cada microsserviço pode ser desenvolvido usando uma linguagem de programação e estrutura que melhor se adapte ao problema que ele é projetado para resolver, o que oferece mais possibilidades de tecnologias para usar. (Oracle Corporation, 2021)

3.2.7 Versionável e substituível

Com o controle completo dos cenários de implantação, é possível manter versões diferentes de um mesmo serviço rodando ao mesmo tempo, proporcionando retrocompatibilidade e fácil migração. Além disso, serviços podem ser atualizados ou mesmo substituidos sem ocasionar indisponibilidade do serviço. (FAMILIAR, 2015)

3.3 Desafios

However, developing applications in the microservices architecture presents three main challenges: (a) how to program systems that consists of a large number of services running in parallel and distributed over a cluster of computers; (b) how to reduce the communication overhead caused by executing a large number of small services; (c) how to support the flexible deployment of services to a network to achieve system load balance. (XU et al., 2016)

Desvantagens dos microserviços:

- Maior complexidade de desenvolvimento e infraestrutura; - Debug mais complexo; - Comunicação entre os serviços deve ser bem pensada; - Diversas tecnologias pode trazer problemas por inexperiência dos devs; - Monitoramento é crucial e mais complexo; - (Criar um microserviço pode ser complexo, e ter demais pode trazer problemas.).

3.3.1 Comunicação

- Comunicação entre os serviços deve ser bem pensada

cross-platform compatibility issues and inconsistent call standards issues in the process of development and call microservices. (ZUO et al., 2020)

3.3.2 [re]Organizaçao

Organizar o sistema e o time para sustentar uma arquitetura de microsserviços é um grande desafio. Como explica Familiar (2015):

> If you are part of a command-and-control organization using a waterfall software project management approach, you will struggle because you are not oriented to high-velocity product development. If you lack a DevOps culture and there is no collaboration between development and operations to automate the deployment pipeline, you will struggle. (FAMILIAR, 2015)

3.3.3 Plataforma

Criar o ambiente de execução para microsserviços requer um grande investimento em infraestrutura dinâmica em *data centers* dispersos para garantir maior disponibilidade. Se sua atual plataforma *on-premises* não suporta automação, infraestrutura dinâmica, escalamento elástico e alta disponibilidade, deve-se considerar uma plataforma na núvem. (FAMILIAR, 2015). Mais sobre soluções na núvem será discutido no Capítulo 5.

3.3.4 Identificação

Domain-driven design (projeto orientado a domínio) é uma técnica bem consolidada e muito usada no desenvolvimento de software. Entretanto, para aplica-la em microsserviços, é necessário analisar onde cada peça desse padrão de projeto deve ficar. Em vez de projetar os modelos e os contextos limitados separando-os em camadas, deve-se juntar os contextos com seus respectivos modelos, e procurar por possíveis pontos de separação da aplicação - um lugar onde a linguagem muda, por exemplo. Isso resultaria em um ponto de partida para separar as partes e formar uma arquitetura de microsserviços. (FAMILIAR, 2015)

3.3.5 Testes

Assim como em qualquer aplicação, o teste é uma parte crucial do seu desenvolvimento. Escrever e testar código não muda muito entre as arquiteturas monolítica e de microsserviços, contudo, nos microserviços existem mais testes a serem executados. Não deve-se testar o microsserviço apenas antes de seu lançamento, mas sim em cada passo do *pipeline* de implantação, sempre automatizando o máximo de etapas possível, para assim garantir uma entrega rápida de software de qualidade. (FAMILIAR, 2015)

3.3.6 Descoberta

Encontrar microsserviços em um ambiente distribuido pode ser feito de algumas maneiras diferentes. A informação pode ser armazenada diretamente no código, pode ser guardada e acessada em um arquivo, ou pode ser construido um microsserviço para encontrar outros microsserviços e disponibilizar suas localizações. Contudo, para prover detectabilidade como um serviço será necessário adquirir um produto de terceiros, integrar um projeto aberto, ou desenvolver sua própria solução. (FAMILIAR, 2015)

3.4 Tipos de microsserviços

3.4.1 Serviço de dados (data service)

Tipo de serviço mais baixo-nível. Responsável por receber e tratar dados, assim fornecendo acesso a determinado domínio e suas regras.

3.4.2 Serviço de negócio (business service)

Em determinados momentos as operações precisam de mais de um modelo do domínio para serem representadas em um serviço. Assim, os serviços de negócio agregam dados e oferecem operações mais complexas. Eles englobam vários serviços de domínio e proveem uma funcionalidade do negócio de nível mais alto, podendo também encapsular domínios relacionados. Por exemplo, uma funcionalidade "Matricular Aluno", que envolveria operações em vários serviços de domínio.

3.4.3 Serviço de tradução (translation service)

Acessar um recurso externo. Ex: App –> T.S. –> Api externa

3.4.4 Serviço de ponta (edge service)

Entregue ao cliente, dependendo de suas necessidades. Ex: E.S. para cliente mobile e outro para clientes web

4

Boas práticas

Este capítulo aprensenta as boas práticas comumente seguidas na construção de aplicações com arquitetura de microsserviços.

4.1 Antes de tudo, comece pelo monólito

> But as with any architectural decision there are trade-offs. In particular with microservices there are serious consequences for operations, who now have to handle an ecosystem of small services rather than a single, well-defined monolith. Consequently if you don't have certain baseline competencies, you shouldn't consider using the microservice style. (FOWLER, 2014)

Fowler (2014) afirma que existem 3 pré-requisitos para se adotar uma arquitetura de microserviços, e que é mais fácil lidar com as operações de um monólito bem definido do que de um ecossistema de pequenos serviços. Assim sendo, é uma boa prática começar pela arquitetura monolítica até que o sistema já esteja bem definido e estes pré-requisitos sejam atendidos - provisionamento rápido, monitoramento básico, e implantação rápida de aplicação.

4.1.1 Provisionamento rápido

No contexto da computação, provisionamento significa disponibilizar um recurso, como uma máquina virtual por exemplo. Para produzir software, é necessário provisionar muitos recursos, tanto para os desenvolvedores quanto para o cliente. Naturalmente, o provisionamento é mais fácil na núvem. Na AWS por exemplo, para conseguir uma nova máquina, basta lançar uma nova instância e acessá-la - um processo muito rápido quando comparado ao *on-premises*, onde precisaria-se comprar uma nova máquina, esperar chegar, configurá-la, e só então ela estará pronta. Para alcançar um provisionamento rápido, será necessário bastante automação. (FOWLER, 2014)

4.1.2 Monitoramento básico

Muitas coisas podem dar errado em qualquer tipo de arquitetura, mas em especial nos microserviços pois cada serviço é fracamente acoplado, estando sujeitos não só a falhas no código, mas também na comunicação, na conexão, ou até falhas físicas. Portanto o monitoramento é crucial nesse tipo de arquitetura para que problemas, especialmente os mais graves possam ser detectados no menor tempo possível. Além disso, o monitoramento também pode ser usado para detectar problemas de negócio, como uma redução nos pedidos de um site de vendas, por exemplo. (FOWLER, 2014)

4.1.3 Implantação rápida

Na arquitetura de microserviços a implantação geralmente é feita separadamente para cada microserviço. Com muitos serviços para gerenciar, ela pode se tornar uma tarefa árdua, portanto será novamente necessário uma automação dessa etapa, que geralmente envolve um *pipeline* de implantação, que deve ser automatizado o máximo possível. (FOWLER, 2014)

4.2 Configuração

4.3 Implantação

4.4 Comunicação entre microserviços

- API Gateway (Ponto único de entrada) . Padronizar e controlar o acesso aos serviços e APIs. Dessa forma podemos ter controles de acesso unificados, autenticação em ponto único, etc. Mas essa também é a principal desvantagem: o ponto único de falha . Oferece um proxy ou uma fachada (ponto de entrada) . Desvantagem: Esse gateway pode ser um ponto de falha massiva . Pode simplesmente autorizar e redirecionar requests, ou . Pode também usar Decorators para adicionar informações necessárias aos requests . Pode limitar o acesso ou conteúdo trafegado (mas isso geralmente é feito por outra entidade (?))

- Process agregator pattern (Agregando serviços) . Agrega serviços de negócio (É ainda mais alto nível) . Fazem as chamadas para os serviços necessários e montam uma resposta adequada . Deve ter uma lógica de processamento, e não ser apenas um proxy. No mínimo deve unir a resposta de diversos serviços . Para construir um agregador, define-se um novo modelo para o sistema, que representará os dados agregados como um subnegócio . A partir deste modelo, pensar na API que fornecerá as operações . A ideia é relativamente simples, mas a implementação pode ser complexa

- Edge pattern (Pontos de entrada para cada tipo de cliente) . Gateway específico para determinados clientes . Foco nas necessidades reais de determinados clientes . Esses clientes

podem ser clientes da API, como clientes HTTP, ou clientes de negócio mesmo . Por exemplo, em vez de modificar a lógica de negócios, cria-se um novo 'edge', que receberá a resposta e modificará de acordo com a necessidade do cliente . Uma possibilidade seria trabalhar apenas com edge services, e nenhum API gateway, isso é, não existiria um ponto único de entrada universal, mas sim um para cada tipo de cliente . Para construir uma edge (ponta) ! Identificar o cliente e suas necessidades ! Construir contratos específicos para o cliente, isso é, ter recursos diferentes para cada cliente. Por exemplo, a URL pode ser diferente de cliente web para clientes mobile.

RPC? Ou apenas APIs?

4.5 Monitoramento

- Lidando com logs Logs são importantes sempre, mas especialmente em microserviços. Pode haver um serviço dedicado para logs, ou um sidecar de logs. Logs informam sobre o estado e a saúde do sistema.

. Agregando logs O formato dos logs (pesquisar formatos?) deve ser compartilhado entre todos os serviços, deve ser igual. Também deve existir uma taxonomia comum. (Logs de erro, de warning, etc) Logs de monolitos são agregados por padrão. [12-Factor Application - É uma metodologia de desenvolvimento que diz que os logs devem ser um stream de dados. Esses logs podem ser impressos na saída padrão, e um serviço específico de logs coleta esses logs, faz o parse, categorização, relatório e todo processamento necessário - https://12factor.net/] Todos os logs de todos os microserviços devem ser agregados e organizados em um ponto para que possam ser consultados com facilidade. Um motivo importante para organizar os logs é rastrear as chamadas de uma execução. Devemos poder reconstruir uma operação a partir de um identificador. Isso é o equivalente à stack-trace de um sistema monolítico. Usar padrões bem consolidados de trace ID para gerar logs. Usar ferramentas de gerenciamento (APMs - Application performance management) para visualizar essas stack-traces.

. Agregando métricas Logs precisam ser desenvolvidos, mas métricas só precisam de instrumentação - o que náo é uma tarefa simples - pois geralmente as ferramentas já possuem as próprias métricas ou já existem métodos consolidados para monitoramento, como um servidor web por exemplo que já grava quando é recebida uma requisição, ou para medir uso de cpu do servidor por exemplo, já existem diversas ferramentas para isso. Métricas nos permitem saber o que está acontecendo em qualquer momento, e decidir que ações devem ser tomadas a partir disso. Escalar um serviço que recebe muitas requisições por exemplo, ou diminuir um que não. Métricas podem servir até para questões de negócios e de business inteligence. Usar dashboards de alto nível para facilitar a visualização e monitoramento do status da aplicação Posteriormente, ter dashboards específicos para cada serviço, com mais detalhes.

4.6 Lidando com dados

"A estabilidade do serviço está diretamente relacionada ao banco de dados que ele acessa."

- Single service database Problema: Escalabilidade do serviço e do banco são fortemente relacionados. Solução: Cada serviço (quando necessário) terá seu próprio banco de dados.

- Shared service database Problema: Às vezes precisamos centralizar os dados (talvez por motivos contratuais, por exemplo, dois dados acessados por microserviços diferentes precisam estar disponível no mesmo banco de dados). Nesse caso o banco escala conforme a necessidade do maior desses microserviços. Solução: Tratar esse banco em cada serviço como se ele fosse separado.

Geralmente há preferência pelo **single service database**, não compartilhando bancos de dados entre serviços. Mas quando necessário, usa-se o **shared service database**, mas sempre tratando o banco de dados em cada serviço como se ele fosse separado. Com cada serviço tendo seu próprio banco, a escalabilidade do serviço e banco pode ser feita em conjunto. Assim, serviços que recebem poucos acessos podem ter bancos menos potentes e mais baratos, e vice-versa.

- Um padrão de codificação: CQRS - Command Query Resposibility Segregation (Segregação da responsabilidade entre o comando e uma busca) "At its heart, [CQRS] is the notion that you can use a different model to update information than the model you use to read information. For some situations, this separation can be valuable, but beware that **for most systems, CQRS adds risky complexity**."

Ou seja, usar um modelo para leitura (busca) e outro modelo diferente para escrita (inserção/edição). É possível ter um banco de dados de escrita e outro de leitura, e fazer uma sincronização entre esses. Essa ideia é muito facilitada usando-se o padrão CQRS.

. Com leitura e escrita separados, cada parte pode realizar operações mais complexas . O modelo de leitura pode ter informações agregadas de outros domínios . O modelo de escrita pode ter dados sendo automaticamente gerados . Aumenta **muito** a complexidade de um sistema

- Eventos assíncronos [Um tipo de arquitetura: Event sourcing - ter toda a base dos dados através de eventos. Para reconstruir os dados, há uma lista de eventos (Pesquisar mais)] . Determinados problemas não podem ser resolvidos na hora. (Um pagamento, por exemplo) . Um serviço emite um evento que será tratado em seu devido tempo . Usar tecnologias de mensageria e serviços de stream de dados - filas de mensageria: RabbitMQ. Serviço de streaming de dados: Kafka)

4.7 APIs

Considerando que APIs são uma parte crucial no desenvolvimento de microserviços, sendo responsável por grande parte da comunicação que se faz necessária para conectar tantos serviços separados e manter um funcionamento eficiente e livre de falhas, esse trabalho terá um foco grande em boas práticas no desenvolvimento de APIs.

4.7.1 Use códigos de status de respostas HTTP

Esses códigos são números entre 100 e 599, cada um tendo um significado diferente, e cada centena sendo classificada em tipos diferentes de resposta. 100-199 representam respostas de informação. 200-299 representam respostas de sucesso. 300-399 representam tipos de redirecionamentos. 400-499 representam erros por parte do cliente. 500-599 representam erros por parte do servidor. Isso é um padrão definido na seção 10 da RFC 2616 (NIELSEN et al., 1999), e facilita com que o cliente entenda o que aconteceu com a requisição à API. Esses códigos devem ser enviados juntos com a resposta à requisição.

4.7.2 Use JSON para trocar dados

JSON é o JSON is commonly used to exchange data between client and server. Although it is derived from JavaScript, it is supported by other major languages, either natively or through libraries. (RapidAPI, 2022)

4.7.3 Implement Endpoint Nesting

If you are developing a REST API with categories, it is good to implement endpoint nesting as it shows the relationship between different endpoints. (RapidAPI, 2022)

4.7.4 Use an SSL Certificate

Secure your REST API by adding a Secure Socket Layer (SSL) certificate. This will make your API use HTTPS protocol instead of HTTP and make it less vulnerable to malicious attacks. (RapidAPI, 2022)

4.7.5 Contratos de dados e versionamento

Uma API depende de contratos de dados, isso é, uma definição dos dados que serão recebidos e retornados. Esse contrato implica num compromisso de manter o serviço correspondente funcionando e inalterado. Entretanto, é possível desenvolver melhorias ou adicionar funcionalidades sem quebrar o contrato. Para tanto, deve ser feito um versionamento da API e deve ser mantida a transparência com os clientes que a usam. Por exemplo, sempre que algo for alterado é preciso atualizar a documentação.

Para fazer o versionamento, pode-se usar o processo de versionamento de software para representar os estados da API. Nesta técnica, usa-se três números para representar a versão, por exemplo "2.3.7". O primeiro representa a versão maior, o segundo, a versão menor, e o terceiro, o patch (pequena atualização para consertar ou melhorar algo). (SOFTWARE. . . , 2022)

Em uma API, para cumprir o contrato de dados, apenas modificações aditivas podem ser feitas, tais como novos endpoints ou novos campos opcionais em algum recurso. Quando isso é feito, a versão da API muda de 2.3.7 para 2.4.0 ou para 2.3.8 dependendo do tamanho da mudança.

Quando é necessário realizar alterações que descumprem o contrato, deve-se alterar a versão maior da API, por exemplo passando de 2.3.7 para 3.0.0. Nesses casos, é importante manter a versão anterior funcionando e inalterada, criando uma nova rota para acessar a versão nova, para que clientes usando a versão anterior não apresentem falhas.

4.7.6 Segurança em APIs

Autenticação

Incluir autenticação em uma API consiste em exigir uma prova de autorização do uso da API. A autenticação nas APIs é indispensável para aumentar a segurança, e existem formas diferentes de implementá-la, as quais serão melhor discutidas no Capítulo 5.

Validação de entradas

Validar entradas significa verificar as requisições que chegam com o intuito de garantir que elas não contém dados impróprios, tais como injeções de SQL ou *scripting* entre sites (scripting significa executar uma determinada sequência de comandos). Essa validação deve ser implementada tanto em nível sintático como em semântico, isso é, tanto impondo correção da sintaxe quanto impondo correção de valores. (RapidAPI, 2022)

API Gateway

Um API Gateway funciona como uma porta única de entrada para as APIs de cada serviço. As requisições dos clientes são direcionadas para um único endereço, facilitando a organização das chamadas. Esse gateway fica situado entre o cliente e os serviços do *backend*, e é responsável por redirecionar as requisições recebidas para os serviços apropriados, assim o gerenciamento das chamadas pode ser feita em apenas um lugar em vez de em cada API de cada serviço. Além disso, pode-se implementar uma camada de segurança e de monitoramento. (RapidAPI, 2022)

Limitação de taxa de requisições

Limitar a taxa de requisições é um jeito de proteger a infraestrutura do servidor nos casos de acontecerem grandes fluxos de requisições, tal como em um ataque de *DoS* (negação de

serviço). Clientes terão seu acesso bloqueado caso enviem uma quantidade de requisições acima do limite determinado. (RapidAPI, 2022)

Compartilhar o mínimo possível

Compartilhar o mínimo possível é uma medida de segurança genérica que pode ser adotada em qualquer microsserviço. Especificamente nas APIs, deve-se retornar estritamente apenas os dados necessários para o cliente. Muitas ferramentas usadas para implementar APIs incluem por padrão informações como se fossem marcas d'água, mas que podem ser removidas, tal como headers "X-Powered-By", que vazam informações do servidor que podem auxiliar usuários mal-intencionados. (RapidAPI, 2022)

4.7.7 Testar a API

Testar uma API isoladamente serve para determinar se ela atende a parâmetros pré-definidos ou não. Tais parâmetros podem ser o cumprimento da funcionalidade, a confiabilidade, a latência, o desempenho, e a segurança. Quando um teste de API falha, deverá ser possível saber precisamente onde o problema se encontra, assim aumentando a velocidade de desenvolvimento e a qualidade do produto. As ferramentas que podem ser utilizadas para testes em APIs são discutidas na sessão subseção 5.4.3.

4.7.8 Salvar a resposta no cache

Caching avoids excessive database queries. For endpoints that frequently return the same response, caching can be implemented to reduce the number of calls to the API and improve performance. (RapidAPI, 2022)

4.7.9 Comprimir os dados

The transfer of large payloads will slow down an API. Data compression combats this issue by decreasing the data size and improving speed. There are various compression methods available, a common one being GZIP. (RapidAPI, 2022)

4.7.10 Evitar trazer ou buscar resultados a mais ou a menos

Over-fetching results in unnecessary and unusable data, and under-fetching results in an incomplete response. Good architecture, planning, and appropriate API management tools are essential to avoid these. (RapidAPI, 2022)

4.7.11 Paginar e filtrar

Both pagination and filtering are great methods to reduce response complexity and improve user experience. Pagination enables the separation and categorization of data, and filtering limits the results of parameters. (RapidAPI, 2022)

4.7.12 Usar PATCH, não PUT

The PATCH and PUT methods are similar, but PATCH has performance advantages. When modifying a resource, PUT updates the entire resource, which is often unnecessary, whereas PATCH only updates a specific part. Therefore PATCH has a smaller payload. (RapidAPI, 2022)

4.8 Testes

Além dos métodos mais conhecidos de testes, como test-driven development, teste de unidade e teste funcional, é necessário testar os microsserviços conforme passam pelo *pipeline* de implantação. Isso inclui:

- Testes internos: Testar as funções internas do serviço, inclusive uso de acesso de dados, e caching.

- Teste de serviço: Testar a a implementação de serviço da API. Essa é uma implementação privada da API e seus modelos associados.

- Teste de protocolo: Testar o serviço no nível de protocolo, chamando a API sobre o determinado protocolo (geralmente HTTP).

- Teste de composição: Testar o serviço em cooperação com outros serviços no contexto de uma solução.

- Teste de escalabilidade/taxa de transferência: Testar a escalabilidade e elasticidade do microsserviço implantado.

- Teste de tolerância a falha: Testar a capacidade do microsserviço de recupera-se após uma falha.

- Teste de penetração: Trabalhar com uma empresa terceirizada de segurança de software para realizar testes de penetração no sistema.

4.9 A metodologia de 12 fatores

Asdf. (Oracle Corporation, 2021).

Qwer. (WIGGINS, 2017)

4.10 Do monólito aos microserviços

Separando serviços monolíticos

- Separando serviços de domínio (Data Service): . Usar Domain-Driven Design (DDD) . Começar modelando o domínio, não pensando na persistância. Definir previamente as regras e o domínio (Programação orientada a interfaces), para então... . Avaliar **quais ações serão disponibilizadas neste serviço**. (Ex: Inserir, editar, recuperar, exibir, etc...) . Construir o serviço, pensando primeiro no contrato . REST e RPC podem andar juntos

- Separando serviços de negócio (Business Service): . Identificar o processo que deve ser exposto . Identificar os domínios que serão necessários nesse serviço . Defina a API que será utilizada, focando no domínio e não nos dados. (Ex: Passar uma matrícula, e não um ID) . Consumir serviços de domínio para executar os processos

- Padrões . Strangler pattern Quebrar um monolito, tirando as funcionalidades dele aos poucos Pode-se começar isolando os dados Ou pode-se começar isolando o domínio

. Sidecar pattern Compartilhar código sem que seja necessário criar um novo microserviço. Usa-se pacotes a parte, que podem ser facilmente instalados. Esses pacotes só precisam ser alterados em um lugar para ter efeito em todos os microserviços. Ex: pacotes do npm, do composer, do maven, etc

4.10.1 Identificação

If you are currently working with a complex layered architecture and have a reasonable domain model defined, the domain model will provide a roadmap to an evolutionary approach to migrating to a microservice architecture. If a domain model does not exist, you can apply domain-driven design in reverse to identify the bounded contexts, the capabilities within the system. (FAMILIAR, 2015)

4.10.2 Organização

Em uma mudança do monólito para microsserviços, é recomendado que não sejam feitas mudanças grandes e abruptas na sua organização. Em vez disso, deve-se procurar uma oportunidade com uma iniciativa de negócio para testar a fórmula proposta por Familiar (2015) :

- Formar um pequeno time inderdisciplinar (cross-functional?).

- Oferecer treinamento e orientação na adoção de práticas ágeis, como o scrum.

- Oferecer uma localização física separada para esse time trabalhar a fim de não afetá-lo negativamente por politicas internas ou hábitos antigos.

- Adotar uma abordagem de minimo produto viável para entregar pequenos mas incrementais *releases* de software, usando essa abordagem durante todo o ciclo de vida.

- Integrar esse serviço com sistemas existentes, usando um acoplamento solto.

- Percorrer esse ciclo de vida do microsserviço diversas vezes, fazendo as adaptações necessárias até chegar a equipe ficar confortável com o processo.

- Colocar o time principal em posições de liderança enquanto são formados novos times interdisciplinares para disseminar o conhecimento e a prática.

5

Soluções: Ferramentas

Este capítulo aprensenta ferramentas comumente usadas na construção de aplicações com arquitetura de microsserviços

5.1 Flexibilidade

(tais como as ferramentas de Auto Scaling da AWS)

5.2 Plataforma

Microsoft Azure is a microservice platform, and it provides a fully automated dynamic infrastructure, SDKs, and runtime containers along with a large portfolio of existing microservices that you can leverage, such as DocumentDb, Redis In-Memory Cache, and Service Bus, to build your own microservices catalog. (FAMILIAR, 2015)

5.3 Testes de microsserviços

Teste de penetração: NOTE: This will requires cooperation with Microsoft if you are pen testing microservices deployed to Azure.

5.4 APIs

5.4.1 GraphQL

A query language for your API

GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your

API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools. (The GraphQL Foundation, 2018)

5.4.2 API Gateway

Amazon API Gateway <https://aws.amazon.com/pt/api-gateway/?nc1=h_ls>

API Gateways are an all-in-one way to implement security, monitoring, and overall API management. They are a single entry point for API calls. They sit between the clients and a number of backend services to handle calls appropriately.

5.4.3 Ferramentas para testes em APIs

RapidAPI offers RapidAPI Client for VS Code to test APIs locally inside Visual Studio Code. You can also schedule API tests using RapidAPI Studio.

programar os testes em uma linguagem de programação

cURL

postman

solução integrada ao VSCode - thunderclient

5.4.4 Ferramentas para segurança em APIs

Autenticação - Always use secure authentication methods such as OAuth, JWTs, or API Keys. It's not recommended to use basic HTTP authentication as it sends user credentials with each request. It is considered the least secure method.

Validação de entradas - Métodos de validação de entrada: JSON and XML Schema validation; Regular expressions; Data type validators available in framework; Minimum and maximum value range check for numerical inputs; Minimum and maximum length check for strings.

5.4.4.1 Métodos de autenticação

API Keys are unique identifiers assigned to clients, which grant them access to an API. They are passed to the server with every request and authenticate the client. They also provide authorization and can be used to identify a user's individual access permissions. API Keys are long alphanumerical strings designed to be almost impossible to guess. They are passed to servers as a query parameter or in an HTTP request header.

OAuth is a powerful framework that uses tokens to give apps limited access to a user's data without needing the user's password. The tokens used are restricted and only allow access to data that the user specified for the particular app. It works by the user(client) first requesting

authorization from the resource owner. The user is then given a unique access token from an authorization server used in each request to the resource server.

Basic HTTP authentication involves the client passing the user's username and password with every request. This is done using an HTTP Header. Basic HTTP authentication is generally considered the least secure. However, if you decide to use it, ensure you are using an HTTPS connection. If not, data is a risk of being leaked.

Ferramenta para rate limiting. (RapidAPI, 2022)

5.4.5 Ferramentas completas

Uma solução para a sobrecarga na execução de tantos microserviços é a um ambiente de desenvolvimento integrado na linguagem CAOPLE (XU et al., 2016). Essa plataforma oferece grande controle sobre a implantação e testagem de microserviços.

6
Conclusão

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Plano de Continuidade

Analisar a eficiência desses padrões e práticas.

Analisar a eficiência dessas soluções e ferramentas.

Propor uma combinação desses padrões e dessas ferramentas para a construção de uma aplicação com arquitetura de microsserviços.

Referências

FAMILIAR, B. What is a microservice? In: ____. *Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions*. Berkeley, CA: Apress, 2015. p. 9–19. ISBN 978-1-4842-1275-2. Disponível em: <https://doi.org/10.1007/978-1-4842-1275-2_2>. Citado 9 vezes nas páginas 10, 11, 12, 13, 14, 15, 16, 25 e 27.

FOWLER, M. *bliki: MicroservicePrerequisites*. 2014. Blog do Martin Fowler. Disponível em: <https://martinfowler.com/bliki/MicroservicePrerequisites.html>. Acesso em: 06 Oct 2022. Citado 2 vezes nas páginas 17 e 18.

Middleware Lab. *What are Microservices? How Microservices architecture works*. Middleware Lab, 2021. Disponível em: <https://middleware.io/blog/microservices-architecture/>. Citado na página 6.

NIELSEN, H. et al. *Hypertext Transfer Protocol – HTTP/1.1*. [S.l.], 1999. Disponível em: <https://datatracker.ietf.org/doc/rfc2616/>. Citado na página 21.

Oracle Corporation. topic, *Learn about architecting microservices-based applications on Oracle Cloud*. Oracle Corporation, 2021. Disponível em: <https://docs.oracle.com/pt-br/solutions/learn-architect-microservice>. Citado 4 vezes nas páginas 6, 10, 14 e 24.

RapidAPI. social media. 2022. Disponível em: <https://twitter.com/Rapid_API>. Acesso em: 25 Oct 2022. Citado 5 vezes nas páginas 21, 22, 23, 24 e 29.

SOFTWARE versioning. 2022. Page Version ID: 1116804459. Disponível em: <https://en.wikipedia.org/w/index.php?title=Software_versioning&oldid=1116804459>. Citado na página 22.

The GraphQL Foundation. Documentation, *GraphQL | A query language for your API*. 2018. Disponível em: <https://graphql.org/>. Acesso em: 26 Oct 2022. Citado na página 28.

WIGGINS, A. topic, *The Twelve-Factor App*. 2017. Disponível em: <https://12factor.net/>. Acesso em: 21 Oct 2022. Citado na página 24.

XU, C. et al. CAOPLE: A Programming Language for Microservices SaaS. In: *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. [S.l.: s.n.], 2016. p. 34–43. Citado 3 vezes nas páginas 6, 14 e 29.

ZUO, X. et al. An api gateway design strategy optimized for persistence and coupling. *Advances in Engineering Software*, v. 148, p. 102878, 2020. ISSN 0965-9978. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0965997820304452>. Citado na página 15.