UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

Características, boas práticas e ferramentas no desenvolvimento de aplicações com arquitetura de microsserviços

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 ferramentas no desenvolvimento de aplicações com arquitetura de microsserviços

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 - Pedido de comentários
JSON	JavaScript Object Notation - Notação de objeto javaScript
DDD	Domain-driven Design - Projeto orientado a domínio
SSL	Secure Socket Layer - Camada de soquete seguro
RPC	Remote Procedure Call - Chamada de procedimento remoto

Sumário

1 Introdução

Todos que têm contato com o ramo do desenvolvimento de software provavelmente já ouviram 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", inspirado pela arquitetura orientada a serviços, e que tem ganhado grande popularidade nos últimos anos. (Middleware Lab, 2021; WASEEM et al., 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, boas práticas comumente seguidas no desenvolvimento de aplicações com essa arquitetura, e as ferramentas que podem ser 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;

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

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

- Analisar a eficiência das ferramentas apresentadas;

- Combinar algumas das ferramentas apresentadas 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. (MONOLITHIC. . . , 2022)

2.1.1 Benefícios

O maior e melhor benefício da arquitetura monolítica é sua simplicidade. Os monólitos são simples para desenvolver, para implantar, e para escalar. Ademais, uma aplicação simples é uma aplicação fácilmente entendida pelos seus desenvolvedores, fato que por sí só já melhora sua manutenibilidade.

Outra vantagem dos monólitos é sua facilidade de construção em relação a sua infraestrutura. Por não possuirem dependências com outras aplicações e nem precisarem se dedicar a comunicação externa, os monólitos têm uma infraestrutura fácil de se estruturar.

Até certo tamanho, as aplicações monolíticas são fáceis de manter porque são fáceis de serem entendidas. Porém, depois que a aplicação ou o time cresce muito, 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... (RICHARDSON, 2018)

2.1.2 Limitações

As limitações das aplicações monolíticas incluem:

Confiabilidade

Escalabilidade

Reutilização

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.

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.

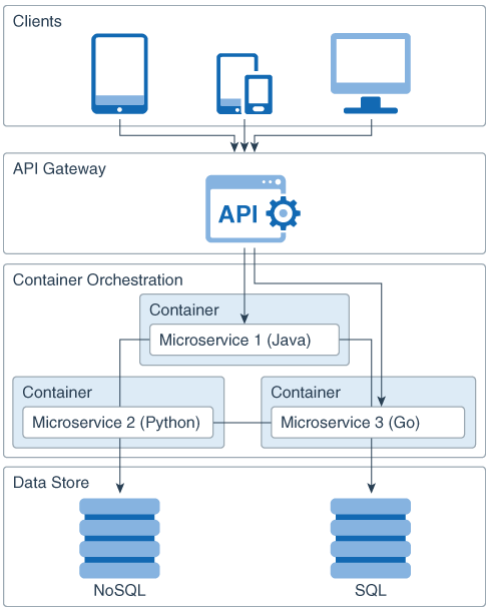
Flexibilidade

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, 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)

Figura 1 – Aplicação com arquitetura de microsserviços



Fonte: Oracle Corporation (2021)

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 gerenciar as chamadas aos microsserviços e será melhor discutido no subseção 4.7.5 - 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*).

2.2.1 Tipos de microsserviços

2.2.1.1 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. Quando um serviço de dados realiza apenas operações relacionadas a um determinado domínio da aplicação, ele também é chamado de serviço de domínio.

2.2.1.2 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, em um site de cursos online, um serviço de negócio poderia prover uma funcionalidade chamada "Matricular Aluno", que envolveria as operações de inserir aluno no serviço de cursos, inserir aluno no serviço de pagamento, e inserir aluno no serviço de gamificação.

2.2.1.3 Serviço de tradução (translation service)

Um serviço de tradução é um intermediário entre a aplicação e um recurso externo, provendo uma forma de acessar esse recurso. No caso desse serviço externo sofrer mudanças, pode-se realizar as alterações consequentemente necessárias em apenas um lugar, nesse serviço de tradução. Por exemplo, a aplicação pode consumir uma API externa por meio do serviço de tradução, pedindo para que ele faça uma requisição para essa API, e então recebendo a resposta.

2.2.1.4 Serviço de ponta (edge service)

É um serviço que serve diretamente ao cliente, sendo customizado para atender necessidades específicas desse cliente. Por exemplo, pode existir um serviço de ponta para clientes móveis e outro serviço de ponta para clientes web.

2.3 Trabalhos relacionados

2.3.1 Microservices, IoT and Azure - capítulo 2: What is a microservice, por Familiar (2015)

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.

Em contraste com o trabalho atual, Familiar (2015) não aborda boas práticas, padrões e ferramentas usadas no desenvolvimento de microsserviços.

2.3.2 A Systematic Mapping Study on Microservices Architecture in DevOps, por Waseem, Liang e Shahin (2020)

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.

Em contraste com o trabalho atual, Waseem, Liang e Shahin (2020) não aborda as características dos microsserviços, mas também mapeia desafios enfrentados e soluções empregadas.

2.3.3 Design, monitoring, and testing of microservices systems: The practitioners' perspective, por Waseem et al. (2021)

Esse trabalho tem o objetivo de entender como sistemas de microsserviços são projetados, monitorados, e testados na indústria. Foi conduzida uma pesquisa relativamente grande que obteve 106 respostas e 6 entrevistas com praticantes de microsserviços. Os resultados obtidos identificam os desafios que os praticantes enfrentam e as soluções empregadas no projeto, monitoramento e teste de microsserviços. Também é feita uma discussão profunda sobre os resultados, da perspectiva dos praticantes, e sobre as implicações para pesquisadores e praticantes.

Em contraste com o atual trabalho, Waseem et al. (2021) não abordam as características dos microsserviços.

3

Características

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

3.1 Propriedades dos microsserviços

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. 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. (FAMILIAR, 2015)

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. 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. (FAMILIAR, 2015)

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. 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. (FAMILIAR, 2015)

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. O termo configurável pode ser definido como - Projetado ou adaptado para formar uma configuração ou para algum propósito. (FAMILIAR, 2015)

3.1.5 Automação

O ciclo de vida de um microsserviço deve ser totalmente automatizado, desde o planejamento (projeto) até a implantação. O termo automatizado pode ser definido como - Funcionar sem precisar ser controlado diretamente. (FAMILIAR, 2015)

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

De acordo com Xu et al. (2016), os três grandes desafios do desenvolvimento de aplicações com arquitetura de microsserviços são

1 - Como programar sistemas que consistem de um grande número de serviços executando em paralelo e distribuidos em um conjunto de máquinas;

2 - Como reduzir a sobrecarga de comunicação causada pela execução de grandes números de pequenos serviços;

3 - Como sustentar a implantação flexível de serviços em uma rede para conseguir realizar o balanceamento de carga.

Esses e outros desafios podem ser divididos em tópicos mais específicos, abordados a seguir.

3.3.1 Complexidade

O uso da arquitetura de microsserviços implica num grande aumento de complexidade não apenas na infraestrutura, mas também em algumas etapas do ciclo de desenvolvimento do software, como no *debug* ou nos testes por exemplo. Consequentemente, muitos outros desafios surgem a partir dessa complexidade. Além disso, o uso de diversas tecnologias pode trazer problemas por inexperiência dos desenvolvedores. De acordo com Waseem et al. (2021), mais pesquisas são necessárias para lidar com a complexidade dos microsserviços no nível de *design* (projeto), de monitoramento, e de testes, desafios para qual não há soluções dedicadas.

3.3.2 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)

3.3.3 [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)

3.3.4 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.

3.3.5 Identificação com DDD

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. Veja a **??** para um possível caminho a ser seguido.

3.3.6 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)

3.3.7 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)

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.

Já (LUMETTA,) concorda com a convenção de começar pelo monólito, mas afirma que podem existir exceções. Ele descreve 3 condições que podem tornar a adoção de uma arquitetura de microserviços em uma nova aplicação a escolha correta. Elas são: Há necessidade de entrega de serviços rapida e independentemente, parte da plataforma precisa ser extremamente eficiente, e planeja-se aumentar o time.

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 A metodologia de 12 fatores

A metodologia de 12 fatores para o desenvolvimento de aplicativos é um conjunto de regras e diretrizes para o desenvolvimento de aplicativos nativos da nuvem e software como um serviço. De acordo com ela, os microsserviços devem respeitar as seguintes orientações:

I. Base de Código - Cada microsserviço deve ter uma base de código única e particular, com rastreamento utilizando controle de revisão, e devem ser criados várias implantações.

II. Dependências - Cada microsserviço deve declarar e isolar suas dependências.

III. Configurações - Configurações de ambiente devem ser armazenadas fora do microsserviço, para que ele possa decidir a configuração apropriada a ser usada.

IV. Serviços de Apoio - Os microsserviços não devem fazer distinção entre serviços de terceiros e serviços locais.

V. Construir, lançar, e executar - Deve-se separar e distinguir cada etapa do processo de desenvolvimento. Na etapa de construção, o código é transformado em um executável. Na etapa de lançamento, o executável se combina com a configuração atual da implantação, seja teste, desenvolvimento ou produção. Na etapa de execução, o aplicativo é executado no ambiente adequado ao lançamento selecionado.

VI. Processos - Deve-se executar a aplicação como um ou mais processos que não armazenam estado, assim diminuindo o acoplamento e facilitando o escalamento.

VII. Vínculo de porta - Um microsserviço deve ser executado em um container e exposto por meio de portas.

VIII. Concorrência - Cada processo deve ser independente e executado separadamente, para se ter um melhor dimensionamento e ser capaz de executar mais ao mesmo tempo.

IX. Descartabilidade - Deve ser possível iniciar ou interromper a aplicação imediatamente sempre que necessário. Caso a aplicação pare de funcionar, deve ser capaz de iniciar novamente sem perdas.

X. Paridade de desenvolvimento e produção - Deve-se manter os ambientes de desenvolvimento, teste e produção o mais semelhantes possível.

XI. Logs - Logs devem ser tratatos como um fluxo de eventos.

XII. Processos administrativos - Tarefas de administração ou de gerenciamento devem ser executadas como processos únicos. (WIGGINS, 2017; Oracle Corporation, 2021)

4.3 Persistência de 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)

The recommended pattern to implement persistence for a microservice is to use a single database. For each microservice, keep the persistent data private, and create the database as a part of the microservice implementation.

In this pattern, the private persistent data is accessible through only the microservice API.

The following illustration shows the persistence design for microservices.

The following variants of this microservice implementation apply to relational databases:

Private tables: Each service owns a set of tables. Schema: Each service owns a private database schema. Database: Each service owns a database server, as shown in the illustration.

A persistence anti-pattern for your microservices is to share one database schema across multiple microservices. You can implement atomic, consistent, isolated, and durable transactions for data consistency. An advantage with this anti-pattern is that it uses a simple database. The disadvantages are that the microservices might interfere with each other while accessing the database, and the development cycles may slow down because developers of different microservices need to coordinate the schema changes, which also increases inter-service dependencies.

Your microservices can connect to an Oracle Database instance that is running on Oracle Cloud Infrastructure. The Oracle multi-tenant database supports multiple pluggable databases (PDBs) within a container. This is the best choice for the persistence layer for microservices, for bounded context isolation of data, security, and for high availability. In many cases, fewer PDBs can be used with schema-level isolation.

4.4 Implantação

Containers. Automação.

After you build your microservice, you must containerize it. A microservice running in its own container doesn't affect the microservices deployed in the other containers.

A container is a standardized unit of software, used to develop, ship and deploy applications.

Containers are managed using a container engine, such as Docker. The container engine provides the tools that are necessary to bundle all the application dependencies as a container.

You can use the Docker engine to create, deploy, and run your microservices applications in containers. Microservices running in Docker containers have the following characteristics:

Standard: The microservices are portable. They can run anywhere. Lightweight: Docker shares the operating system (OS) kernel, doesn't require an OS for each instance, and runs as a lightweight process. Secure: Each container runs as an isolated process. So the microservices are secure.

The process of containerizing a microservice involves creating a Dockerfile, creating and building a container image that includes its dependencies and the environmental configuration, deploying the image to a Docker engine, and uploading the image to a container registry for storage and retrieval.

4.5 Comunicação entre microserviços

Requisições HTTP são o método mais simples para realizar comunicação síncrona entre microsserviços. Uma requisição HTTP é feita por um cliente, para um dado *host* em um servidor, com o propósito de acessar um recurso nesse servidor. Por usar o *Transmition Control Protocol* (TCP), é um método de comunicação confiável, mas não tão eficiente quanto poderia ser.

Para comunicação assíncrona, filas de mensagens são amplamente usadas. Quando um serviço precisa enviar informações a outro de modo assíncrono, ele envia uma mensagem para a fila de mensagens, e ela será armazenada até ser processada ou excluída. Cada mensagem é processada uma única vez, por um único consumidor. As filas de mensagens podem ser usadas para dividir um processamento pesado, para armazenar trabalho em *buffers* ou lotes, ou para processar uniformemente picos de cargas de trabalho.

Embora menos comum, chamada de procedimento remoto (RPC) também é utilizado para realizar comunicação síncrona ou assíncrona entre microsserviços. Uma chamada de procedimento remoto se dá quando um programa faz com que um procedimento ou uma sub-rotina execute em um espaço de endereço diferente, comumente em outra máquina numa rede compartilhada. Essa chamada é feita como se fosse um procedimento local, isso é, o programador não precisa

explicitar que se trata de um procedimento remoto. gRPC é uma ferramenta moderna com alto desempenho que tem ganhado grande popularidade em meio aos praticantes de microsserviços e é considerada um "projeto de incubação"pela Cloud Native Computing Foundation. Essa ferramenta será melhor discutida no Capítulo 5 (Microsoft Corporation, 2022b)

De acordo com Waseem et al. (2021), *API Gateway* e *Backend for frontend* são os padrões de projeto mais utilizados para lidar com a comunicação de microsserviços. São padrões similares - a diferença é que no *Backend for Frontend* há um *gateway* para cada tipo de cliente. Esses padrões serão discutidos na subseção 4.7.5.

4.6 Monitoramento

Em qualquer aplicação o monitoramento é importante para garantir um bom funcionamento. Entretanto, na arquitetura de microsserviços o monitoramento se torna indispensável, além de mais complexo.

resource usage and load balancing as monitoring metrics, log management and exception tracking as monitoring practices are widely used (WASEEM et al., 2021)

4.6.1 Históricos

Um histórico, também conhecido como *log*, descreve o que aconteceu em determinado sistema e provê informações sobre o estado e a saúde dele. Manter um histórico do microsserviço é uma das formas mais simples de se implementar monitoramento, e é fortemente indicado.

Para facilitar a escrita, leitura e operação dos históricos, deve-se padronizar o formato deles em todos os microsserviços e diferenciar entradas de erros, de avisos e de informação. Além disso, eles devem ser agregados e organizados em um único lugar para que possam ser facilmente consultados.

Pode haver um serviço dedicado para logs, ou um sidecar de logs (pacote instalável).

4.6.2 Métricas

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 inclusive servir para questões de negócios e de *business inteligence*.

Enquanto históricos precisam ser desenvolvidos, métricas apenas precisam de instrumentação pois muitas ferramentas já possuem as próprias métricas ou já existem métodos consolidados para as obter. Nos servidores web mais populares, por exemplo, as informações básicas sobre uma requisição já são gravadas por padrão.

É recomendado usar dashboards de alto nível para melhorar a visualização e monitoramento do status da aplicação a partir de suas métricas.

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 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 Troca de dados com JSON

Atualmente JSON é um dos formatos mais populares para troca de dados na web, pelo fato de ser facilmente lido tanto por humanos quanto por máquinas. Em APIs, JSON é usado para enviar e receber requisições por meio do protocolo HTTP, sendo uma solução robusta para a comunicação entre cliente e servidor. Embora seja derivado do JavaScript, JSON também é suportado por muitas outras linguagens, seja nativamente ou por meio de bibliotecas. (BOURHIS; REUTTER; VRGOč, 2020)

4.7.3 Buffers de Protocolo

Alternativa ao JSON. É mais eficiente.

4.7.4 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.

4.7.5 API Gateway

Numa arquitetura de microsserviços, os clientes geralmente consomem funcionalidades de mais de um microsserviço. Se esse consumo é feito do cliente diretamente para o microsserviço, o cliente precisa lidar com várias requisições aos *endpoints*. Quando os microsserviços mudam ou mais microsserviços surgem frequentemente, fica inviável tratar tantos endpoints por parte do cliente. Uma solução para isso é usar um *API Gateway*.

Um *API Gateway* é um padrão de projeto e funciona como uma porta única de entrada para as APIs de cada microsserviço, padronizando e controlando o acesso aos microsserviços e APIs. Esse gateway fica situado entre o cliente e os microsserviços, e é responsável por redirecionar as requisições recebidas para os microsserviços apropriados, assim o gerenciamento das chamadas pode ser feita em apenas um lugar em vez de em cada API de cada microsserviço. Além disso, nele também podem ser implementadas camadas de segurança e de monitoramento.

Outra vantagem é que esse *API Gateway* pode agregar requisições, permitindo que o cliente envie apenas uma requisição para o *API Gateway* para recuperar informações de diferentes microsserviços, o que normalmente exigiria múltiplas requisições. Nesse caso, quando recebida a requisição do cliente, o *API Gateway* fica responsável por disparar as requisições correspondentes, agregar as respostas e as devolver ao cliente.

Entretando, também há desvantagens no uso desse padrão: (1) há um acoplamento forte entre os microsserviços e o *API Gateway*, (2) surge um possível ponto massivo de falha nesse *API Gateway*, (3) Se não escalado adequadamente, esse *API Gateway* pode se tornar um gargalo. (Microsoft Corporation, 2022a)

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)

Certificado Secute Socket Layer (SSL)

Usar um certificado SSL permite que o protocolo HTTPS seja usado em vez do HTTP, criptografando as informações que estão trafegando, o que adiciona uma camada de segurança. (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*

Às vezes referido como *cachear*, salvar informações no *cache* melhora o tempo de busca da informação. Em uma API podem haver múltiplas requisições para a mesma informação em um curto intervalo de tempo, e para cada requisição será necessário buscar a informação. Entretanto, se a informação estiver salva no *cache*, não será necessário buscar essa informação, o que melhora o tempo de resposta da API, especialmente em *endpoints* que frequentemente retornam a mesma resposta. (RapidAPI, 2022)

4.7.9 Comprimir os dados

A transferência de cargas grandes pode diminuir a velocidade da API. Comprimir os dados auxilia nesse problema, diminuindo o tamanho da carga e aumentando a velocidade de transferência. (RapidAPI, 2022)

4.7.10 Paginar e filtrar

A Paginação separa e categoriza resultados, enquanto a filtragem retorna apenas os resultados relevantes de acordo com os parâmetros da requisição. A paginação e filtragem de resultados reduz a complexidade da resposta e facilitam o uso da API. (RapidAPI, 2022)

4.7.11 PATCH ou PUT

Quando é necessário modificar um recurso em uma API, usa-se os métodos HTTP PUT ou PATCH. Enquanto PUT atualiza o recurso inteiro, PATCH atualiza apenas uma parte específica do recurso, assim usando uma carga de dados menor. Portanto, quando possível deve-se usar PATCH em vez de PUT para modificar um recurso. (RapidAPI, 2022)

4.8 Testes

O processo de testes para microsserviços engloba várias estratégias diferentes de testes. Essas estratégias podem ser incluir testes funcionais, como um teste de unidade, ou testes não-funcionais, como um teste de desempenho. Usar múltiplas estratégias de testes garante que a aplicação opera com sucesso em ambientes e plataformas diferentes. De acordo com Waseem et al. (2021), testes de unidade e testes fim-a-fim são as estratégias de testes mais usadas.

Familiar (2015) afirma que além de usar os métodos mais comuns de testes, deve-se também 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. (FAMILIAR, 2015)

5 Ferramentas

Este capítulo aprensenta ferramentas que podem ser usadas na construção de aplicações com arquitetura de microsserviços

5.1 Design, testes, e monitoramento

De acordo com Waseem et al. (2021), mais pesquisas são necessárias para lidar com a complexidade dos microsserviços no nível de *design* (projeto), de monitoramento, e de testes, desafios para qual não há soluções dedicadas.

Our findings reveal that more research is needed to address the monitoring and testing challenges through dedicated solutions. (WASEEM et al., 2021)

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

5.2 Comunicação

API Gateway

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

RPC

gRPC <https://grpc.io/>

Comunicação Assíncrona

RabbitMQ, Azure Service Bus, Amazon Simple Queue Service

5.3 Flexibilidade

Ferramentas de Auto Scaling da AWS

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)

5.5 Implantação

docker, kubernetes

5.6 Plataformas

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)

AWS

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.

Explicar e detalhar as ferramentas.

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

Diagrama de gantt <https://www.seomartin.com/modelo-diagrama-de-gantt/>

Atividade | tabela

Referências

BOURHIS, P.; REUTTER, J. L.; VRGOč, D. JSON: Data model and query languages. *Information Systems*, v. 89, p. 101478, mar. 2020. ISSN 03064379. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0306437919305307>. Citado na página 24.

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 11 vezes nas páginas 3, 9, 11, 13, 14, 15, 16, 17, 27, 28 e 31.

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 18 e 19.

LUMETTA, J. *Microservices for Startups: Should you always start with a monolith?* Disponível em: <https://buttercms.com/books/microservices-for-startups/should-you-always-start-with-a-monolith/>. Citado na página 18.

Microsoft Corporation. *The API gateway pattern versus the Direct client-to-microservice communication*. 2022. Disponível em: <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern>. Citado na página 25.

Microsoft Corporation. *gRPC*. 2022. Disponível em: <https://learn.microsoft.com/pt-br/dotnet/architecture/cloud-native/grpc>. Citado na página 23.

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

MONOLITHIC application. 2022. Page Version ID: 1118196758. Disponível em: <https://en.wikipedia.org/w/index.php?title=Monolithic_application&oldid=1118196758>. Citado na página 8.

NIELSEN, H. et al. *Hypertext Transfer Protocol – HTTP/1.1*. [S.l.], 1999. Disponível em: <https://datatracker.ietf.org/doc/rfc2616/>. Citado na página 24.

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, 15 e 20.

RapidAPI. Mídia Social. 2022. Disponível em: <https://twitter.com/Rapid_API>. Acesso em: 25 Oct 2022. Citado 3 vezes nas páginas 26, 27 e 31.

RICHARDSON, C. *Microservices Pattern: Monolithic Architecture pattern*. 2018. Disponível em: <http://microservices.io/patterns/monolithic.html>. Citado na página 8.

SOFTWARE versioning. 2022. Page Version ID: 1116804459. Disponível em: <https://en.wikipedia.org/w/index.php?title=Software_versioning&oldid=1116804459>. Citado na página 25.

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 30.

WASEEM, M.; LIANG, P.; SHAHIN, M. A systematic mapping study on microservices architecture in devops. *Journal of Systems and Software*, v. 170, p. 110798, 2020. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121220302053>. Citado 2 vezes nas páginas 3 e 12.

WASEEM, M. et al. Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and Software*, v. 182, p. 111061, 2021. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121221001588>. Citado 7 vezes nas páginas 3, 6, 12, 16, 23, 27 e 29.

WIGGINS, A. topic, *The Twelve-Factor App*. 2017. Disponível em: <https://12factor.net/>. Acesso em: 21 Oct 2022. Citado na página 20.

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, 15 e 31.

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 16.