

Microserviços e Orquestração de Contêineres



Eduardo Bischoff Grasel Geovane Martins Schmitz, Luiz Felipe Cabral Lima e Sergio Bonini

Agenda

Introdução (Monolíticos vs Microserviços)

O que são Microserviços?

Containers (Docker)

Orquestração (Kubernetes)

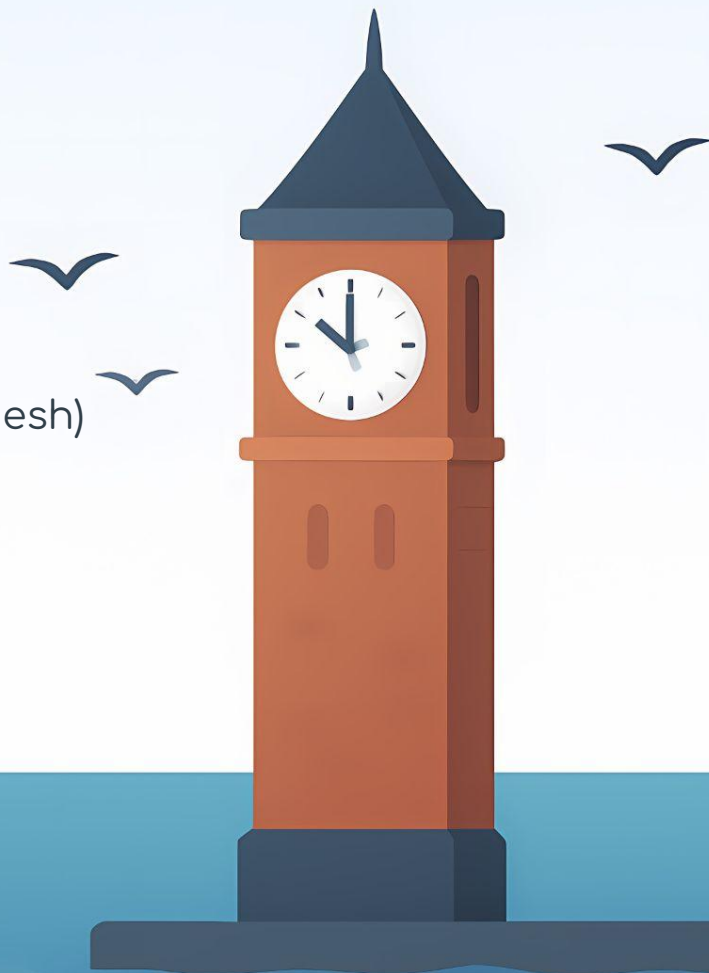
Ferramentas Essenciais (API Gateway, Service Mesh)

Casos de Uso

Desafios

Conclusão

Introdução ao Tutorial.



O problema a ser resolvido

No cenário em constante evolução da arquitetura de software, os microsserviços surgiram como um paradigma transformador, remodelando a forma como concebemos, construímos e implementamos aplicações.



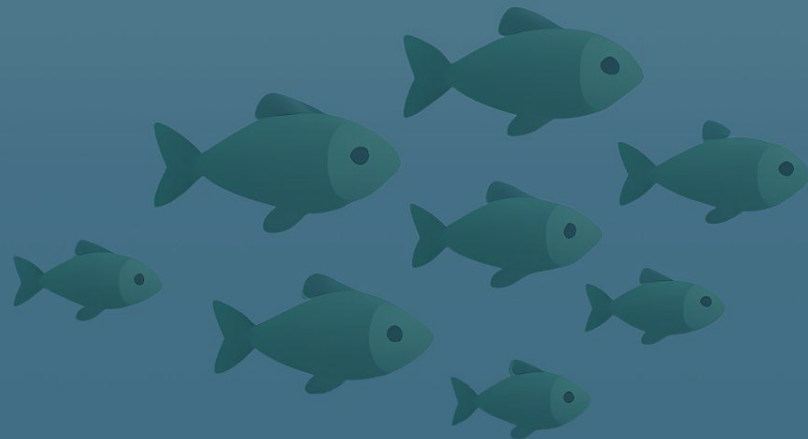
O problema a ser resolvido

A história dos microsserviços é uma jornada marcada pela mudança de estruturas monolíticas para sistemas modulares e distribuídos. Compreender essa evolução é crucial para compreender o impacto que os microsserviços tiveram no desenvolvimento de software moderno.



O problema a ser resolvido

O surgimento dos microsserviços remonta ao início dos anos 2000. Arquiteturas monolíticas tradicionais, embora funcionais, enfrentavam desafios à medida que as aplicações se tornavam maiores e mais complexas. A necessidade de escalabilidade, resiliência e agilidade levou à exploração de arquiteturas alternativas.



Monolíticos vs Microserviços

Criação: Sistemas monolíticos são mais simples de construir porque utilizam um design mais básico. Microserviços são consideravelmente mais complexos e exigem mais planejamento para serem executados.



Monolíticos vs Microserviços

Estrutura: Uma arquitetura monolítica é projetada e construída como uma unidade única. A arquitetura de microserviços defende a ideia de modularidade, utilizando um conjunto de aplicativos menores e implantáveis que permitem a operação de serviços independentes.



Monolíticos vs Microserviços

Complexidade: Quanto mais complexo um sistema, mais adequado ele é para uma arquitetura de microserviços. Microserviços modulares são receptivos a novos recursos e novas tecnologias que tendem a trazer complexidade adicional.



Monolíticos vs Microserviços

Crescimento: Tanto a arquitetura monolítica quanto a arquitetura de microserviços podem ser eficazes em seu uso inicial. Mas o crescimento muda tudo, principalmente quando as organizações percebem que em breve expandirão além do sistema inicial. Nesse ponto, as empresas precisam de um estágio operacional maior, e os microserviços oferecem isso, oferecendo mais maneiras de escalar as operações do que a arquitetura monolítica.



Monolíticos vs Microserviços

Tempo de lançamento no mercado: Esta métrica-chave desempenha um papel fundamental no comércio, pois mede o tempo necessário para fabricar produtos e inseri-los nos canais de distribuição. O tempo de lançamento no mercado é uma área em que a arquitetura monolítica se destaca, indo além dos microserviços. Ao usar apenas uma única base de código, os desenvolvedores podem evitar o tempo e o trabalho extras de incorporar software de diversas fontes.



Monolíticos vs Microsserviços

Escalabilidade: A arquitetura de microsserviços é construída com base em serviços individuais que podem ser compartimentados em formatos modulares e se beneficiam do baixo acoplamento e da intercomunicação obtidos por meio de APIs. Por outro lado, a arquitetura monolítica apresenta menor adaptabilidade geral devido à sua estrutura central densamente composta e software fortemente acoplado.



Monolíticos vs Microserviços

Debugging: A arquitetura monolítica lida melhor com a depuração do que os microserviços por ser mais simples e direta. Depurar uma arquitetura de microserviços é consideravelmente mais lento, mais complexo e trabalhoso.



O que são microsserviços?

Microsserviços, também conhecidos como arquitetura de microsserviços, são um estilo arquitetônico que estrutura uma aplicação como uma coleção de dois ou mais serviços que são:

- Implementáveis de forma independente
- Baixamente acoplados

Os serviços são normalmente organizados em torno das capacidades de negócios. Cada serviço geralmente é de propriedade de uma única equipe pequena.

Características

The background of the slide features a light blue sky with soft, white clouds. Three stylized grey birds are depicted in flight: one in the upper right, one in the middle right, and one in the middle left. The bottom of the slide is a solid dark blue band with white wavy lines representing water.

Independência

Controle Descentralizado: Os microsserviços podem ser implantados de forma independente, permitindo que as equipes os desenvolvam, implantem e escalem de forma autônoma.

Agnóstico em Tecnologia: Cada microsserviço pode ser desenvolvido utilizando diferentes tecnologias, escolhendo a mais adequada para sua funcionalidade específica.

Características



Autonomia

Funcionalidade Autocontida: Microserviços encapsulam capacidades de negócios específicas, garantindo um escopo focado e bem definido.

Gerenciamento de Dados: Cada serviço gerencia seus próprios dados, reduzindo a dependência de outros serviços.

Características

Acoplamento Fraco

Comunicação por meio de APIs: os microsserviços interagem por meio de APIs bem definidas, garantindo o acoplamento fraco entre os serviços. Isso aumenta a flexibilidade e facilita a substituição ou atualização de componentes.

Características

Escalabilidade

Escalonamento Individual: Microserviços podem ser escalonados de forma independente com base na demanda por funcionalidades específicas. Essa escalabilidade direcionada otimiza o uso de recursos.

Características

Resiliência

Isolamento: Falhas em um microserviço não afetam necessariamente os outros. Esse isolamento contribui para a resiliência geral do sistema.

Degradação Graciosa: Microserviços degradam graciosamente durante falhas parciais, garantindo que o sistema permaneça funcional.

Características

Entrega Contínua

Implementação Independente: Microserviços facilitam a integração e a entrega contínuas, permitindo que as equipes implementem alterações sem impactar todo o aplicativo.

Características

Sistema Distribuído

Comunicação em Rede: Microserviços se comunicam por uma rede, permitindo que sejam distribuídos por diferentes servidores ou até mesmo localizações geográficas.

Consistência por Meio de Transações: Garantir a consistência dos dados em um ambiente distribuído geralmente requer o uso de padrões de transações distribuídas.

Características

Monitoramento e Depuração

Monitoramento Isolado: Cada microsserviço pode ser monitorado individualmente, permitindo depuração direcionada e otimização de desempenho.

Características

Organizado em torno das Capacidades de Negócio

Organização Funcional: Os microsserviços são organizados com base nas capacidades de negócio, e não em considerações técnicas, alinhando-se estreitamente com o domínio do negócio.

Características

Design Evolucionário

Adaptabilidade: Os microserviços suportam uma abordagem de design evolucionário, permitindo que as equipes adaptem e desenvolvam seus serviços de forma independente.

Containers



Containers

O que é

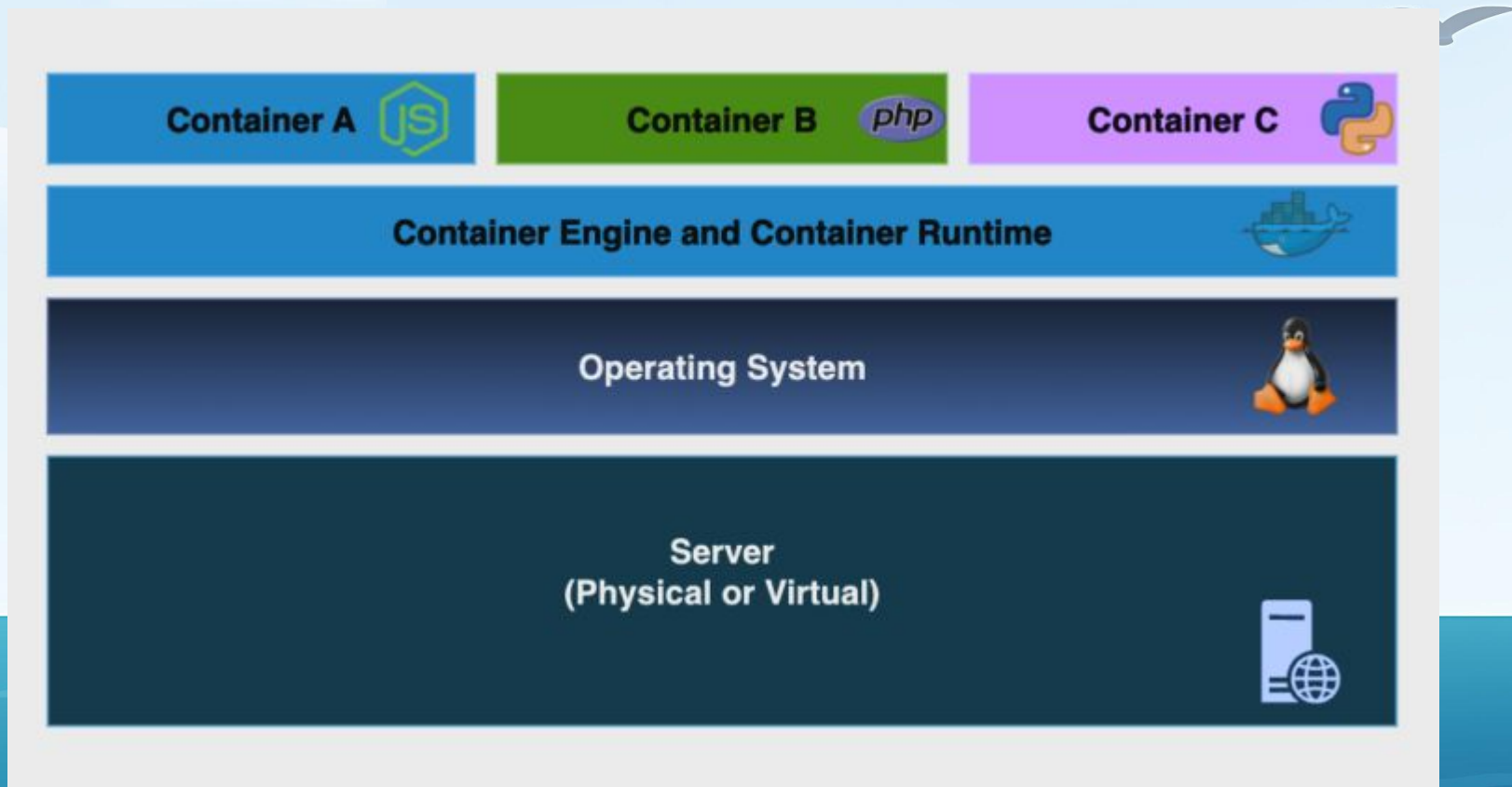
Uma forma leve e portátil de empacotar e executar aplicações junto com todas suas dependências (bibliotecas, arquivos de configuração, binários, etc), garantindo que tudo funcione de forma consistente em diferentes ambientes

Containers

Características:

- Isolamento: Cada container roda de forma isolada, mas compartilha o mesmo kernel do sistema operacional host;
- Leveza: Diferente de VMs, containers não precisam de um sistema operacional completo;
- Portabilidade: Um container pode ser executado em qualquer lugar que tenha uma engine compatível (como docker), o que facilita desenvolvimento, testes e deploys consistentes;
- Escalabilidade: Em sistemas distribuídos, containers facilitam o escalonamento de aplicações, especialmente quando orquestrados com ferramentas como Kubernetes.

Containers



Containers

Orquestração:

Esse é um processo para automatizar o gerenciamento, implementação, escalabilidade, agendamento e comunicação entre múltiplos ambientes distribuídos. Ela é essencial quando se trabalha com centenas ou milhares de containers, especialmente em arquiteturas como microsserviços

Implementáveis de forma independente



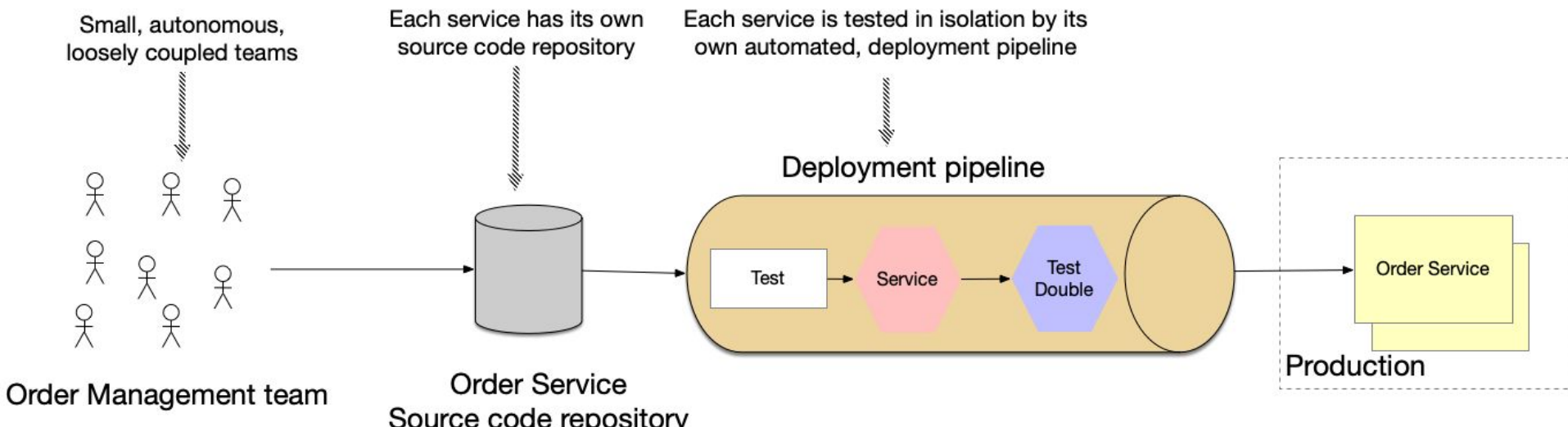
Serviços empacotados cada um como uma unidade implementável ou executável e que está pronta para a produção após ser testada isoladamente.

Se você precisar testar seu serviço com outros serviços para verificar se ele está pronto para produção, ele não será implementável de forma independente.

Pode-se considerar colocar esses serviços em um único repositório, garantindo que a saída do pipeline de implementação único esteja realmente pronta para produção e eliminando a complexidade de desenvolvimento em vários repositórios.

Implementáveis de forma independente

Aceleração do pipeline de implementação, eliminando a necessidade de testes de ponta a ponta lentos, frágeis e complexos de vários serviços, além de excluir a necessidade de as equipes coordenarem suas atividades e, potencialmente, obstruírem umas às outras.



Implementáveis de forma independente



Um obstáculo são os testes de aceitação do usuário em nível de sistema.

Os testes de aceitação geralmente são escritos da perspectiva de um usuário e frequentemente abrangem vários serviços, fazendo com que uma implementação simples desses testes exija testar vários serviços em conjunto.

Para que os serviços sejam implementados de forma independente, é necessário substituir os testes de aceitação do usuário em nível de sistema por testes de aceitação do usuário em nível de serviço.

Implementáveis de forma independente

Um serviço implementável de forma independente requer uma especificação bem definida.

De forma mais geral, para que um serviço seja testável isoladamente, ele precisa possuir:

- uma especificação bem definida
- um conjunto de testes que verifique se seu comportamento está em conformidade com essa especificação.

Se não for possível especificar precisamente o comportamento de um serviço, provavelmente deve-se reconsiderar se ele faz sentido como serviço.

Baixamente acoplados

Existem dois tipos de acoplamento:

- Acoplamento em tempo de execução - influencia a disponibilidade
- Acoplamento em tempo de design - influencia a velocidade de desenvolvimento

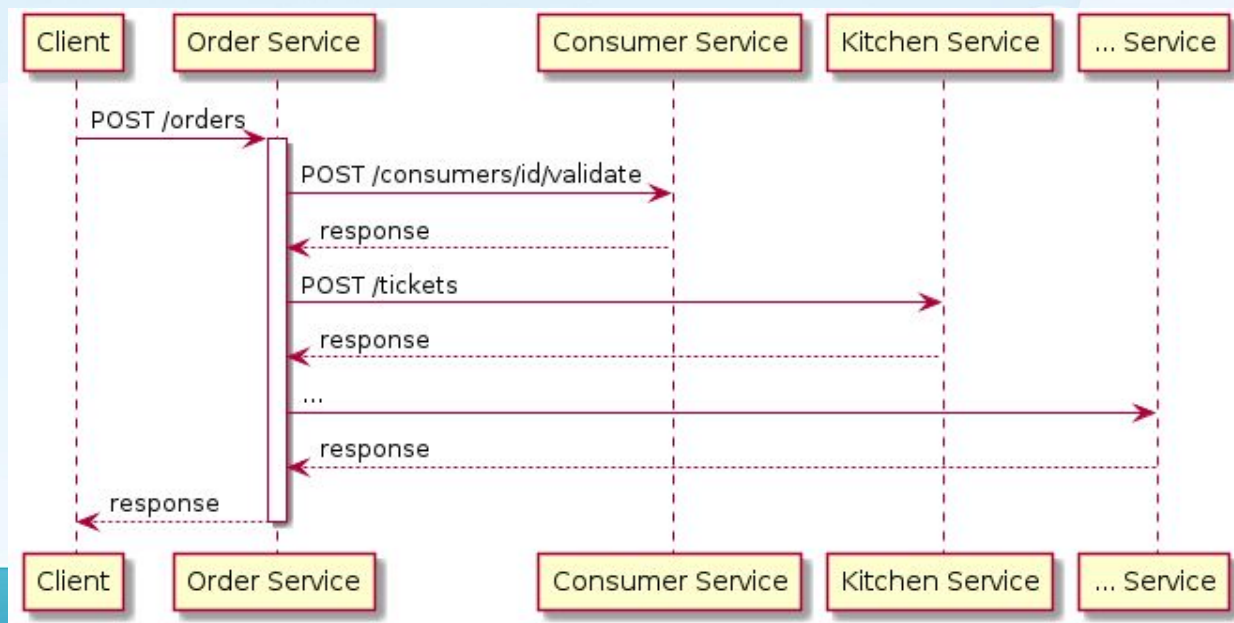
Acoplamento em tempo de execução e disponibilidade

O acoplamento em tempo de execução entre serviços é o grau em que a disponibilidade de um serviço é afetada pela disponibilidade de outro serviço. Ou, para ser mais preciso, é o grau em que a disponibilidade de uma operação implementada por um serviço é afetada pela disponibilidade de outro serviço.

O acoplamento em tempo de execução reduz a disponibilidade

Por exemplo, imaginemos que a operação do sistema *createOrder()* seja implementada por um endpoint *HTTP POST /orders* no Serviço de Pedidos. O Serviço de Pedidos processa o *HTTP POST* invocando outros serviços, aguardando a resposta deles e, em seguida, enviando uma resposta ao seu cliente.

O acoplamento em tempo de execução reduz a disponibilidade



O acoplamento em tempo de execução reduz a disponibilidade

Neste design, o Serviço de Pedidos não pode responder à solicitação *POST* até que os outros serviços respondam a ela. O Serviço de Pedidos (ou a operação *createOrder()*) é considerado acoplado em tempo de execução a esses outros serviços. Como resultado, a disponibilidade da operação *createOrder()* é reduzida, pois todos os serviços devem estar disponíveis.

Redução do acoplamento em tempo de execução



Minimizar o acoplamento em tempo de execução é uma das forças atrativas da matéria escura que resiste à decomposição.

Uma maneira de reduzir o acoplamento em tempo de execução de uma operação é reduzir o número de serviços que a implementam. De fato, podemos eliminar completamente o acoplamento em tempo de execução tornando uma operação local para um único serviço.

No entanto, nem sempre é possível criar uma arquitetura de microsserviços em que todas as operações sejam locais. Isso provavelmente violará as forças da energia escura, que incentivam a decomposição.

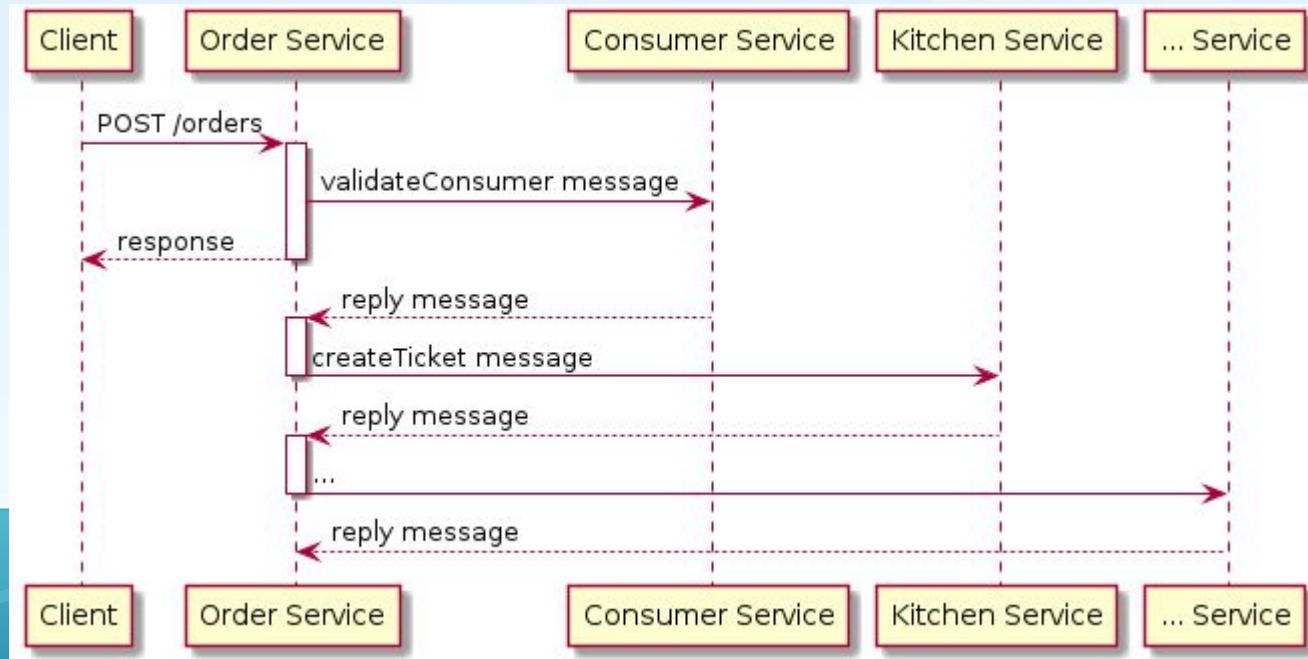
Redução do acoplamento em tempo de execução

A outra maneira de reduzir o acoplamento em tempo de execução e, ao mesmo tempo, satisfazer as forças da energia escura é projetar serviços autocontidos. Um serviço autocontido responde a uma solicitação síncrona com um resultado parcial e, em seguida, conclui a operação de forma assíncrona.

Redução do acoplamento em tempo de execução

Por exemplo, o Serviço de Pedidos poderia responder à solicitação *HTTP POST /orders* com uma resposta *202 Accepted* e, em seguida, iniciar uma Saga de Criação de Pedidos para concluir a operação. Essa abordagem melhora a disponibilidade do Serviço de Pedidos. A desvantagem é que torna o cliente mais complexo, pois ele precisa ser capaz de lidar com resultados parciais e, de alguma forma, determinar o resultado final da operação.

Redução do acoplamento em tempo de execução



Acoplamento em tempo de design e velocidade de desenvolvimento

O grau de acoplamento em tempo de design entre um par de elementos de software — classes...serviços — é a probabilidade de que eles precisem mudar juntos pelo mesmo motivo.

O acoplamento em tempo de design entre serviços em uma arquitetura de microsserviços é especialmente problemático.

O acoplamento em tempo de design reduz a velocidade de desenvolvimento

Por exemplo, imaginemos que o Serviço de Pedidos e o Serviço de Atendimento ao Cliente estejam fortemente acoplados. Sempre que uma alteração significativa precisa ser feita no Serviço de Atendimento ao Cliente, a sequência de etapas é a seguinte:

O acoplamento em tempo de design reduz a velocidade de desenvolvimento

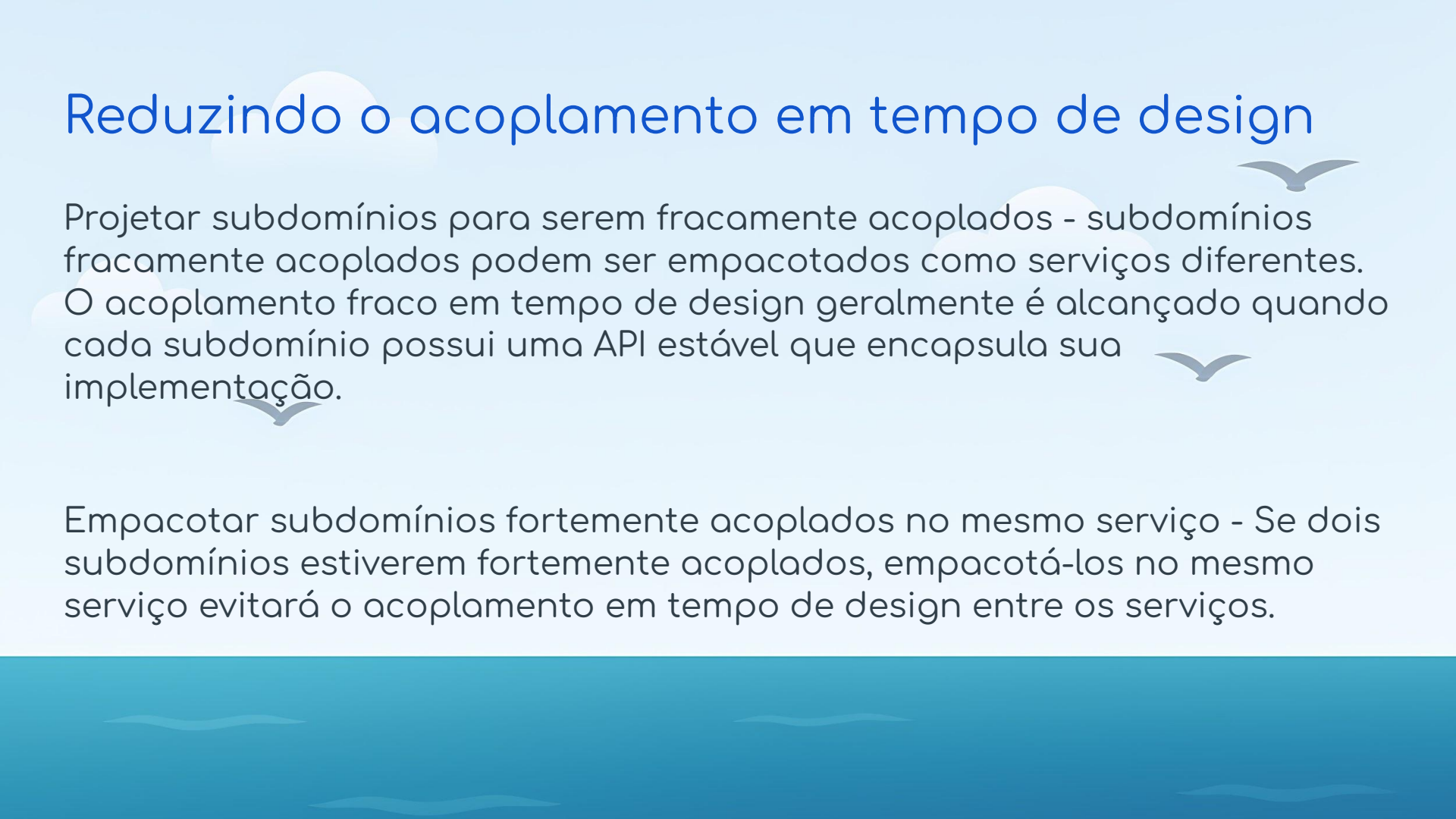
- Alterar o Serviço de Atendimento ao Cliente para adicionar uma nova versão principal de sua API. O serviço deve implementar a versão antiga e a nova de suas APIs até que todos os clientes tenham sido migrados.
- Migrar o Serviço de Pedidos para a nova versão da API.
- Remover a versão antiga da API do Serviço de Atendimento ao Cliente.

Além disso, muitas vezes, os serviços pertencem a equipes diferentes, o que exige que essas equipes coordenem as alterações.

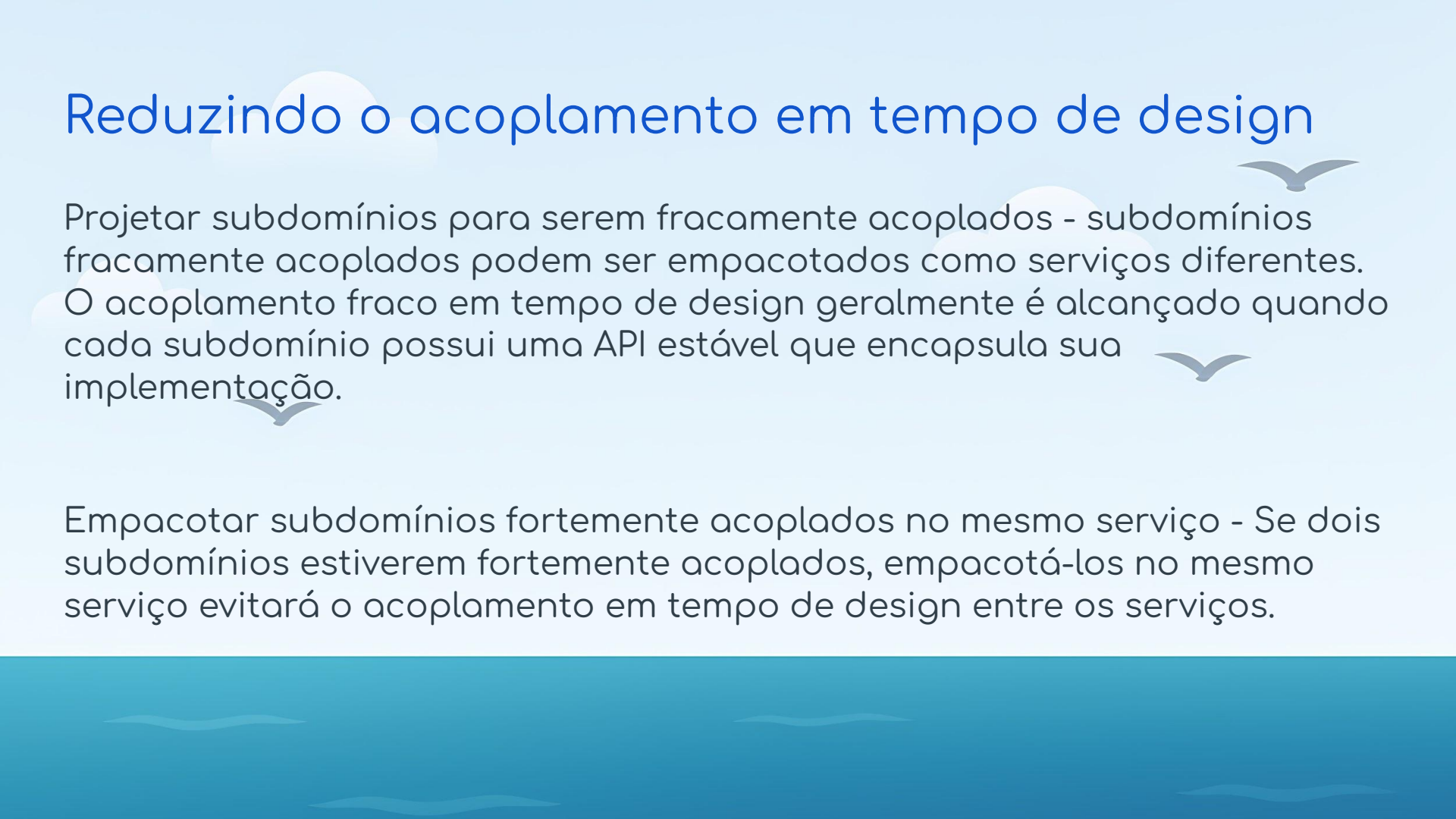
Reduzindo o acoplamento em tempo de design



Projetar subdomínios para serem fracamente acoplados - subdomínios fracamente acoplados podem ser empacotados como serviços diferentes. O acoplamento fraco em tempo de design geralmente é alcançado quando cada subdomínio possui uma API estável que encapsula sua implementação.



Empacotar subdomínios fortemente acoplados no mesmo serviço - Se dois subdomínios estiverem fortemente acoplados, empacotá-los no mesmo serviço evitará o acoplamento em tempo de design entre os serviços.



Sistemas monolíticos: casos de uso

Startups: Empresas em início de atividade precisam de duas coisas: flexibilidade e financiamento inicial (e bastante de ambos). Uma arquitetura monolítica é a melhor maneira de começar negócios incipientes. Além disso, ela pode ser construída por equipes de desenvolvimento enxutas de forma econômica, sem impor uma curva de aprendizado muito acentuada a essas equipes pequenas.

Sistemas monolíticos: casos de uso

Projetos básicos: Ter uma única base de código traz benefícios em termos de conveniência, especialmente em projetos de escopo rudimentar. Quando o software consegue passar pelo processo de desenvolvimento sem precisar incorporar dados de múltiplas fontes, é uma vitória para a organização.

Contêineres: empacotando microsserviços

Um contêiner é uma unidade padrão de software que agrupa o código e todas as suas dependências para que o aplicativo seja executado de forma rápida e confiável de um ambiente de computação para outro.

Uma imagem de contêiner Docker é um pacote de software leve, autônomo e executável que inclui tudo o que é necessário para executar um aplicativo: código, tempo de execução, ferramentas do sistema, bibliotecas do sistema e configurações.

Contêineres: empacotando microsserviços



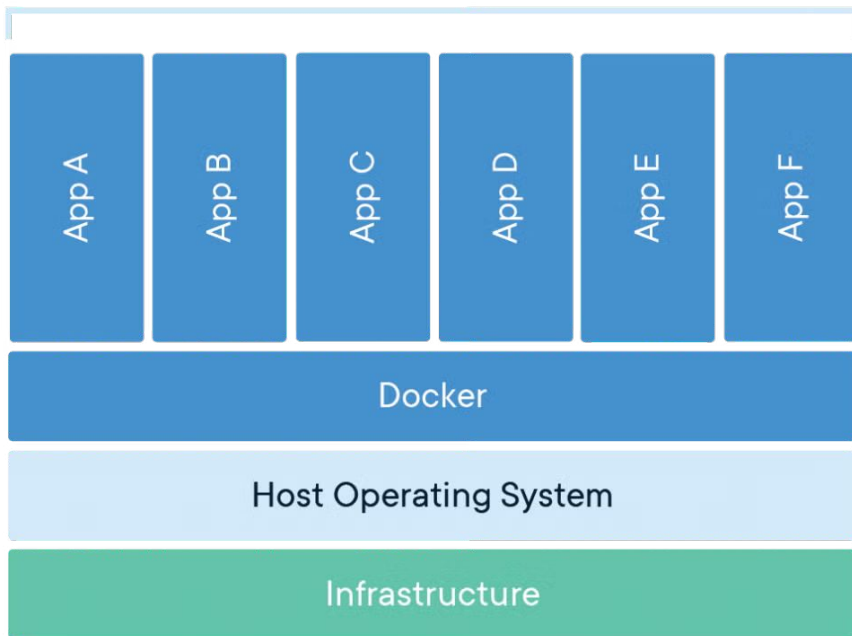
Imagens de contêiner tornam-se contêineres em tempo de execução e, no caso de contêineres Docker, imagens tornam-se contêineres quando executadas no Docker Engine.

Disponível para aplicativos Linux e Windows, o software em contêiner sempre será executado da mesma forma, independentemente da infraestrutura.

Os contêineres isolam o software de seu ambiente e garantem que ele funcione uniformemente, apesar das diferenças, por exemplo, entre desenvolvimento e apresentação.

Contêineres: empacotando microsserviços

Containerized Applications



Desafios na implementação de contêineres

Durante a implementação de contêineres, deve-se manter atenção em alguns aspectos para garantir a segurança dos mesmos, tais como:

- Vulnerabilidade: É necessário que contêineres estejam configurados adequadamente para evitar vulnerabilidades de segurança, caso contrário um contêiner estará suscetível a sofrer um ciberataque.
- Isolamento: Contêineres devem estar devidamente isolados, para caso ocorra uma falha, este não afete o sistema hospedeiro ou outros contêineres.
- Monitoramento: devido a curta vida dos contêineres, o monitoramento dos mesmo se torna difícil, já que estes são criados e destruídos a todo momento.
- Uso de recursos: Deve ser bem distribuído dentre os contêineres do sistema, caso contrário pode ocasionar na lentidão e falhas no sistema.
- Escalabilidade:

Pods



Em plataformas como a Kubernetes, a plataforma não executa containers diretamente, a plataforma engloba um ou mais containers em uma estrutura de nível mais alta chamada “Pods”.




Utilizados como uma unidade de replicação, Pods são uma estrutura que permitem execução, comunicação e compartilhamento de recursos entre containers de um mesmo pod, resultando em um sistema onde containers parecem utilizar do mesmo hardware, enquanto mantêm um certo grau de isolamento.

Graças a utilização de pods, plataformas podem utilizar da replicação para escalar horizontalmente o sistema assim que necessário, em outras palavras, caso um pod esteja sobrecarregado, é possível replicar do mesmo para que a carga de trabalho seja dividida entre os pods, resultando em uma distribuição equilibrada da carga, gerando também um certo grau de resistência a falhas.

Serviços



Com a utilização de contêineres se tornando cada vez mais comum, muitas empresas estão começando a adotá-los no local de VMs em suas plataformas de computação em nuvem de uso geral, porém tendo seus principais usos em Microsserviços, DevOps, MultiNuvem Híbrida e Serverless, onde, já por contêineres serem pequenos, leves e portáteis, facilita e amplia a implementação destes tipos de modelos.



Deployments

Ao realizar um “Deployment” de um container, significa que um container está sendo inserido a um ambiente, sendo o mesmo irrelevante devido às propriedades de um contêiner, permitindo que o mesmo possa ser executado em qualquer ambiente independente do OS instalado.

Contêineres possuem em seus designs a característica de ser colocado e retirado de um ambiente rapidamente dependendo de sua aplicação, pois este geralmente são utilizados para criar e lançar microsserviços, gerando flexibilidade e agilidade a um projeto.

Ferramentas Principais

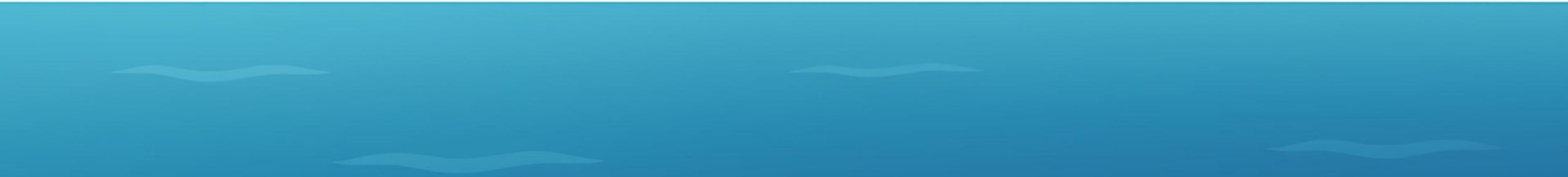
- Docker: Ferramenta mais popular para criação, empacotamento e execução de containers. Possui seu próprio formato de imagem e runtime
- Podman: Alternativa ao Docker, compatível com imagens Docker, sem necessidade de daemon e com foco em segurança
- Containerd: Runtime de containers usado por ferramentas como docker e Kubernetes
- Kubernetes: principal plataforma de orquestração de containers

Frameworks: Spring Boot para o desenvolvimento de microsserviços

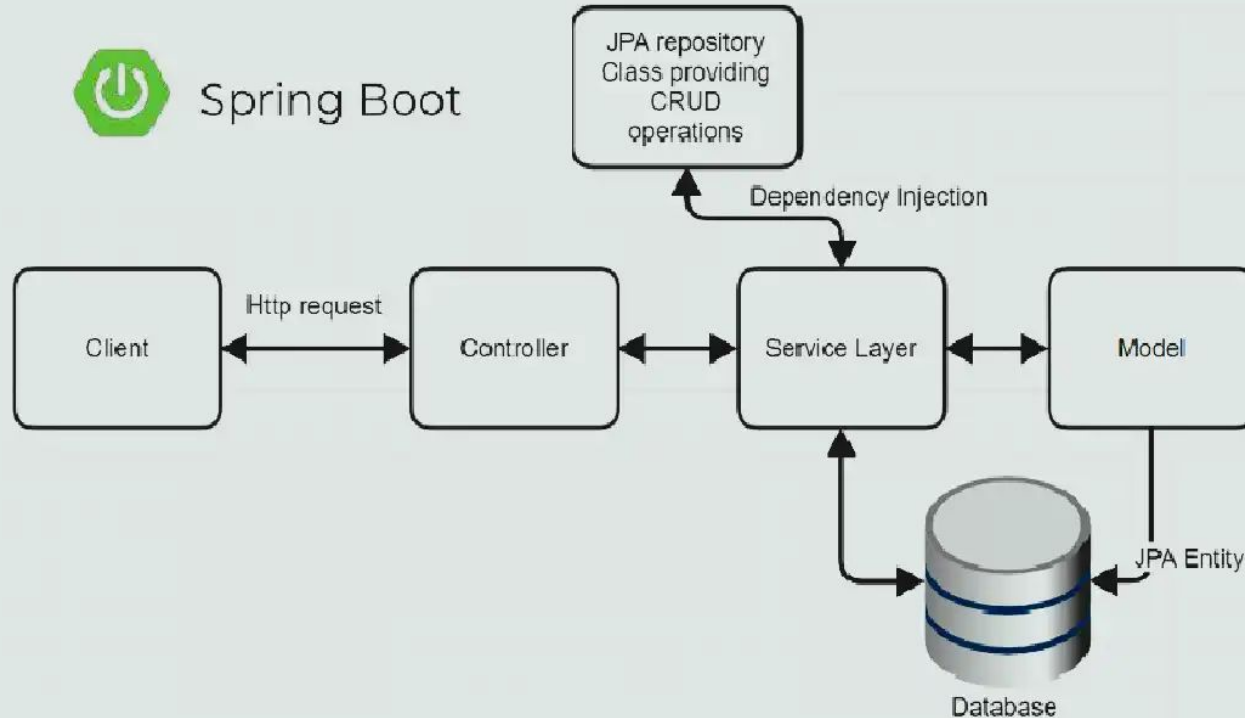


Spring Boot é um framework java que facilita a criação de aplicações standalone e prontas para produção com configuração mínima, No contexto de microsserviços, ele é amplamente utilizado por ser leve, modular e integrar-se facilmente com outras ferramentas.

Compatível com containers docker e fácil de orquestrar com kubernetes



Frameworks: Spring Boot para o desenvolvimento de microsserviços



Contêineres: Docker

Utilizando o kernel de linux e funcionalidades como cGroups e namespaces, essa tecnologia visa executar processos de maneira independente, visando executar diversos processos e apps separadamente em um único sistema.

Esta ferramenta cria containers a partir de uma base em imagem, possuindo todas as dependências necessárias deste container, também automatizando a implantação da aplicação dentro desse ambiente de contêineres. Estas características fazem com que seja fácil a implantação e utilização deste sistema.

Com isso, os containers docker possuem vantagens como sua modularidade, implantação rápida e fácil reversão, fazendo que uma aplicação seja altamente flexível e aberta a mudanças.

Orquestração: Kubernetes

O kubernetes é um software de código aberto utilizada para a orquestração de containers, criado inicialmente pela google com base nos 15 anos de execução de cargas de trabalho containerizadas da empresa.

Esta ferramenta é utilizada para simplificar o gerenciamento de contêineres, agrupando tais contêineres em pods para escalonamento com base na demanda e disponibilidade de recursos, automatizando a alocação de serviços e o balanceamento de carga enquanto monitora a integridade para permitir autocorreção reiniciando e replicando contêineres.

API Gateway: Kong, Zuul

Agindo como um intermediário, um API Gateway é o que permite que o usuário se comunique com os serviços backend, agindo como um proxy reverso para aceitar todas as chamadas da API e então enviando-as para os serviços apropriados, em outras palavras, sua função é separar a interface do cliente da implementação back-end.

Neste sentido, temos Zuul e Kong, ambos sendo gateways, porém com diferentes propostas.

Zuul foi desenvolvido pela Netflix em Java, sendo mais orientado para a integração com a plataforma Spring Cloud, proporcionando roteamento dinâmico, monitoramento, resiliência e segurança, permitindo uma experiência unificada para clientes interagindo com múltiplos microservices.

Em contrapartida, Kong é um gateway mais genérico e versátil, sendo implementado em Lua com uma versão open source, este se destaca por sua robustez e flexibilidade, capaz de lidar com uma maior capacidade de tratamento de tráfego e escalabilidade, tendo em vista que este foi projetado também para ambientes cloud-native.

Service Mesh: Istio, Linkerd

Sendo uma camada de infraestrutura dedicada dentro de uma aplicação de software, este lida com a comunicação entre serviços.

Entre as aplicações existentes, possuímos Istio, um service mesh open source que controla como os microsserviços compartilham dados entre si, utilizado junto a kubernetes para controlar o fluxo do tráfego, aplicar políticas e monitora as comunicações em um ambiente de microsserviços.

Outro serviço existente é o Linkerd, projetado pelo twitter em 2013 quando a plataforma estava mudando de uma arquitetura de plataforma em camadas para uma arquitetura de microsserviços, este então foi doado para a CNCF (cloud native computing foundation), que então lançou sua segunda versão, corrigindo erros de sua antecessora. Linkerd v2, foi criado com o objetivos de zero configuração, simples e leve e pensado para kubernetes, possuindo detecção de protocolo, proxy HTTP 1.1 / 2 e gRPC, injeção automática e zero configuração, mTLS automatico, observabilidade e divisão de trafego.

Exemplos de Uso Real



Netflix

Dentro de sua arquitetura, o netflix utiliza docker para a containerização de suas aplicações, então permitindo que as mesmas possam ser replicadas e reutilizadas, proporcionando modularidade e escalabilidade, enquanto o serviço de computação em nuvem, neste caso a Amazon Web Services (AWS), permite que o netflix escale seus recursos conforme a demanda, garantindo disponibilidade e desempenho.



Nubank



Escrita em clojure, o nubank funciona a partir da sua arquitetura escrita sobre o paradigma funcional, utilizando uma arquitetura de microsserviços com mais de 1000 microsserviços escritos, utilizando uma boa estrutura de pastas baseado na arquitetura de software, assim, podendo encontrar e entender os microsserviços desejados, facilitando a escalabilidade e fazendo com que diferentes equipes possam aprender e trabalhar sobre o que já foi construído.

Este aplicativo utiliza de docker e kubernetes para construção e orquestração de containers, facilitando o desenvolvimento e implantação de novas funcionalidades sem comprometer a estabilidade do sistema.

Microserviços: casos de uso

Plataformas de entretenimento: Administrar uma plataforma de entretenimento internacional exige a capacidade de acompanhar a onda de mudanças nas cargas de trabalho, seja para cargas leves ou pesadas. No caso da Netflix, a gigante do streaming de vídeo migrou de uma arquitetura monolítica para uma arquitetura de microserviços baseada em nuvem. O novo backend da Netflix conta com amplo suporte a balanceadores de carga, o que auxilia seus esforços para otimizar as cargas de trabalho.

Microserviços: casos de uso



E-commerce: O e-commerce depende da arquitetura de microserviços para dar vida à magia do marketplace eletrônico, proporcionando uma experiência de usuário fluida. Com varejistas ambiciosos como a Amazon (AWS) impulsionando as vendas com maior conveniência e entrega mais rápida, é fácil entender por que o mercado de vendas de e-commerce em 2023 ultrapassou US\$ 5,8 trilhões.

Microserviços: casos de uso

Tarefas mais difíceis: O uso contínuo de microserviços normalmente exige as habilidades de implementação e administração de equipes de DevOps treinadas, que podem criar os serviços específicos necessários para aquela estrutura arquitetônica. Essas habilidades são especialmente úteis ao lidar com aplicações complexas.

Desafios e considerações



Microserviços — Desafios e Considerações



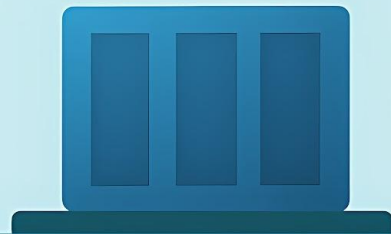
- Comunicação complexa: requer controle de latência, timeout, retry e circuit breaker;
- Gerenciamento de estado: Idealmente stateless; persistência distribuída exige atenção;
- Deploy e versionamento: CI/CD bem estruturado e versionamento de APIs são essenciais;
- testes: Integração entre serviços é complexa, requer testes automatizados robustos;
- Resiliência: falhas parciais são comuns; é necessário aplicar padrões de tolerância a falhas

Orquestração de Containers — Considerações

- Escalabilidade: orquestradores como kubernetes facilitam o autoscaling baseado em carga;
- Configuração e segredos: uso de configMaps e secrets para separar dados sensíveis;
- Infraestrutura: maior complexidade operacional, necessidade de monitoramento constante;
- Deploy e rollbacks: suporte a rolling updates e reversão automática de falhas;

Tutorial: Minikube

Minikube é uma ferramenta que permite executar um cluster Kubernetes de nó único localmente em sua máquina para fins de desenvolvimento e aprendizado.



Minikube

- 1) Acesse <https://minikube.sigs.k8s.io/docs/start/>
- 2) Escolha seu sistema operacional e siga o tutorial (<https://youtu.be/SU-muVhfrGo>)

1 Installation

Click on the buttons that describe your target platform. For other architectures, see [the release page](#) for a complete list of minikube binaries.

Operating system

Linux

macOS

Windows

Architecture

x86-64

ARM64

ARMv7

ppc64

S390x

Release type

Stable

Installer type

Binary download

Debian package

RPM package

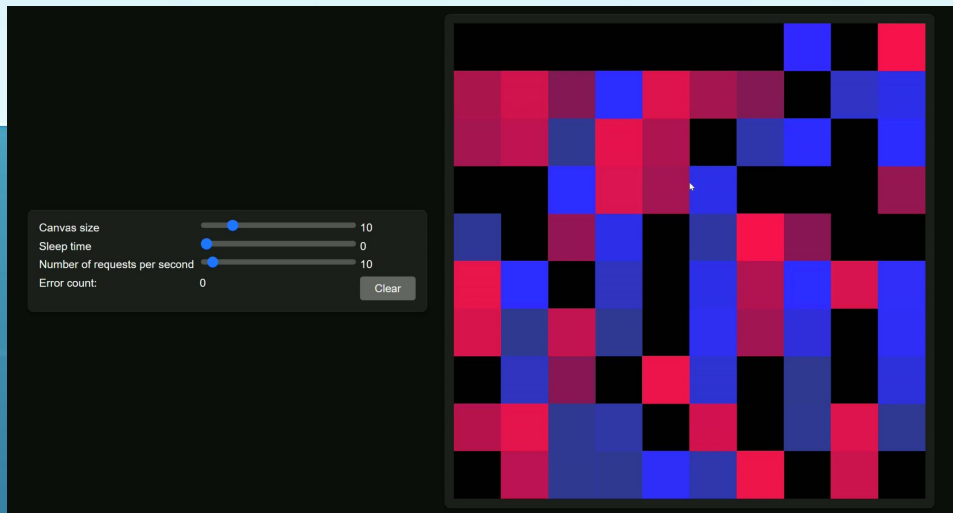
To install the latest minikube **stable** release on **x86-64 Linux** using **binary download**:

```
curl -LO https://github.com/kubernetes/minikube/releases/latest/download/minikube-linux-amd64  
sudo install minikube-linux-amd64 /usr/local/bin/minikube && rm minikube-linux-amd64
```



Minikube

<https://github.com/GeovaneSchmitz/distribute-d-computing-exemple-group-4>



Conclusão

- Microserviços e contêineres transformaram a forma como construímos e escalamos aplicações modernas;
- Essa arquitetura traz agilidade, escalabilidade e resiliência, mas exige aspectos como comunicação, testes, segurança e observabilidade;
- A orquestração com ferramentas como Kubernetes é fundamental para garantir o gerenciamento eficiente e automatizado dos contêineres em ambientes distribuídos ;
- O sucesso na adoção de boas práticas, integração entre equipes e compreensão dos trade-offs.

Perguntas?



Referências

<https://microservices.io/>

<https://microservices.io/post/architecture/2022/05/04/microservice-architecture-essentials-deployability.html>

<https://microservices.io/post/architecture/2023/03/28/microservice-architecture-essentials-loose-coupling.html>

<https://microservices.io/patterns/microservices.html>

<https://medium.com/@wearegap/a-brief-history-of-microservices-part-i-958c41a1555e>

<https://www.ibm.com/think/topics/monolithic-vs-microservices#:~:text=Structure:%20A%20monolithic%20architecture%20is,the%20operation%20of%20independent%20services>

<https://www.docker.com/resources/what-container/>

<https://endjin.com/blog/2022/01/introduction-to-containers-and-docker>

<https://www.veritas.com/blogs/the-10-biggest-challenges-of-deploying-containers>

<https://www.redhat.com/en/topics/containers/what-is-kubernetes-pod>

<https://nerdexpert.com.br/entendendo-a-estrutura-do-netflix/>