17 - Arquitetura de Microsserviços

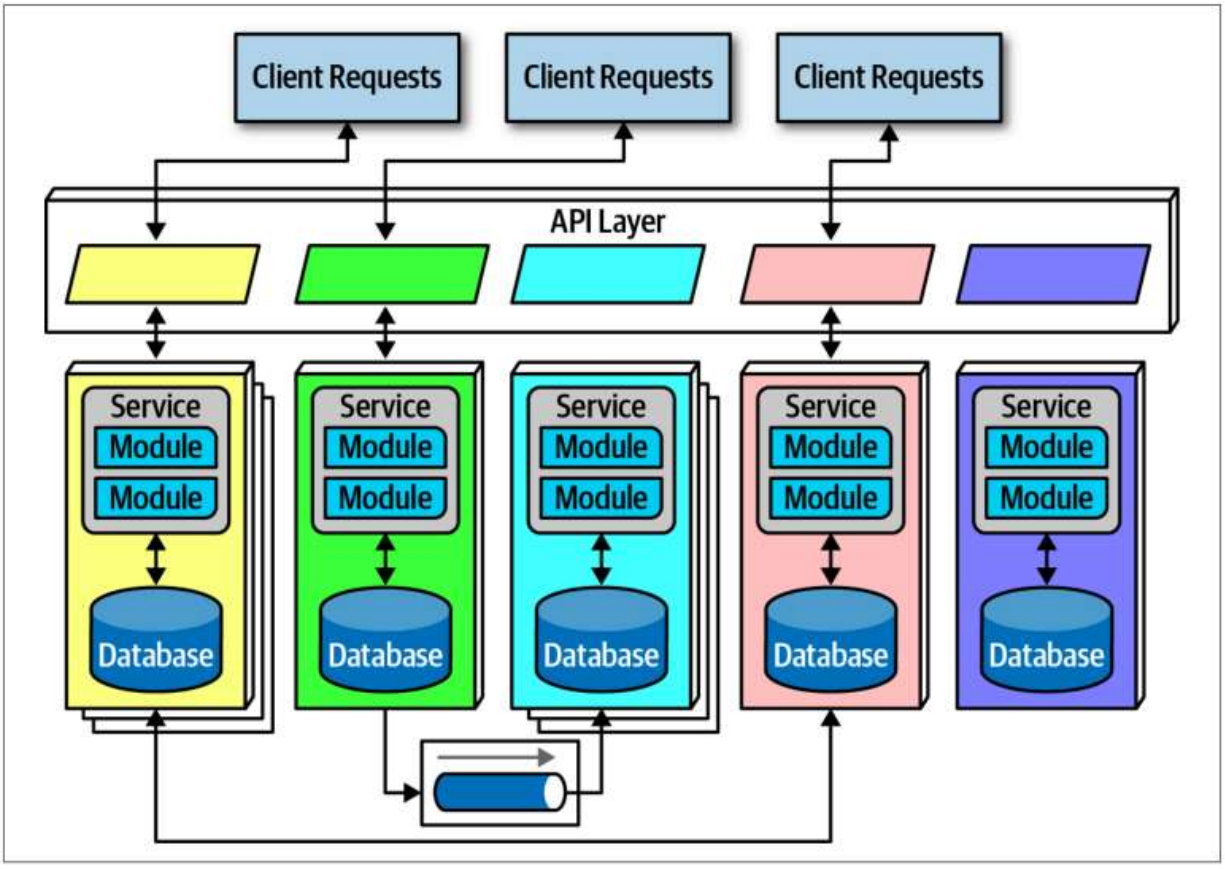
2025-12-22 14:20 pm

- Bastante popular ultimamente

História

- Geralmente uma nova arquitetura é uma soma descentralizada de muitos arquitetos diferentes tomando decisões similares por acaso que acaba se consolidando numa coisa só aos poucos organicamente.
- Mas microsserviços foi definido em Março de 2014 por Martin Fowler e James Lewis num blog.
- Inspirado por *Domain-Driven design*
    - *bounded context*
        - Forma de alto **desacoplamento**.
    - O contexto é acoplado dentro de si mas nunca fora dele.
        - Comportamento autocontido e independente.
- Trade-off entre *Reuso VS Acoplamento*
    - Para maximizar desacoplamento, é comum haver mais duplicação

Topologia



- *Figure 17-1. The topology of the microservices architecture style*
- Cada serviço tem tudo o que precisa para operar independentemente.

Distribuído

- Microsserviços são uma arquitetura distribuída
    - Pode haver vários numa mesma máquina.
        - Problemas de isolamento.
    - Hoje é fácil provisionar mais máquinas com mais sistema operacionais.
        - Antes era necessário compartilhar mais os recursos.
- Ter latência de rede e verificação de segurança em cada endpoint pode prejudicar o desempenho.
- Importante definir bem as fronteiras do sistema e evitar interações não apropriadas.

Bounded Context

- Cada serviço modela um domínio ou um workflow.
    - Contém dentro de si schema, classes, etc.
- Melhor duplicar código do que compartilhar para evitar acoplamento.
- Domain partitioned.

Granularidade

- Não é fácil achar a melhor granularidade para os serviços em "microsserviços".
    - É um erro fazê-los pequenos demais.
- Alguns critérios para definir as fronteiras:
    - **Propósito:**
        - Fronteira mais óbvia. Coesão funcional, contém um comportamento.
    - **Transações:**
        - Quais entidades que precisam cooperar revelam quais funções devem estar juntas. Tudo o mais constante, quanto menos transações, melhor.
    - **Coreografia:**
        - Mesmo vários serviços bem isolados poderiam ser um só serviço se houver muita comunicação entre eles.
- Granularidade ideal vem após refinar várias iterações.

Isolamento de Dados

- Requisito da arquitetura.
- Para evitar acoplamento, não pode haver um único database.
- *Entity Trap:*
    - Erro de modelar cada serviço para cada entidade num database.
- Não há como ter uma única fonte de verdade.
    - Duas alternativas:
        1. Definir um dado domínio para um dado como fonte de verdade
        2. Usar replicação de dados ou caching para distribuir informação
- Cada serviço com cada time pode tomar as melhores decisões para si sem afetar ou depender de mais ninguém.

Camada de API

- Pode ou não existir, entre os consumidores do sistema.
- Não pode ser um mediador ou orquestrador.

- Contra filosofia de desacoplamento.

Reuso Operacional

- Como lidar com aspectos gerais do sistema, como logging, monitoramento, etc?
  - Cada microsserviço faz do seu modo, duplicando tudo?
- Para resolver isso, pode haver um sidecar em cada serviço com esses componentes, comum a todos os serviços.

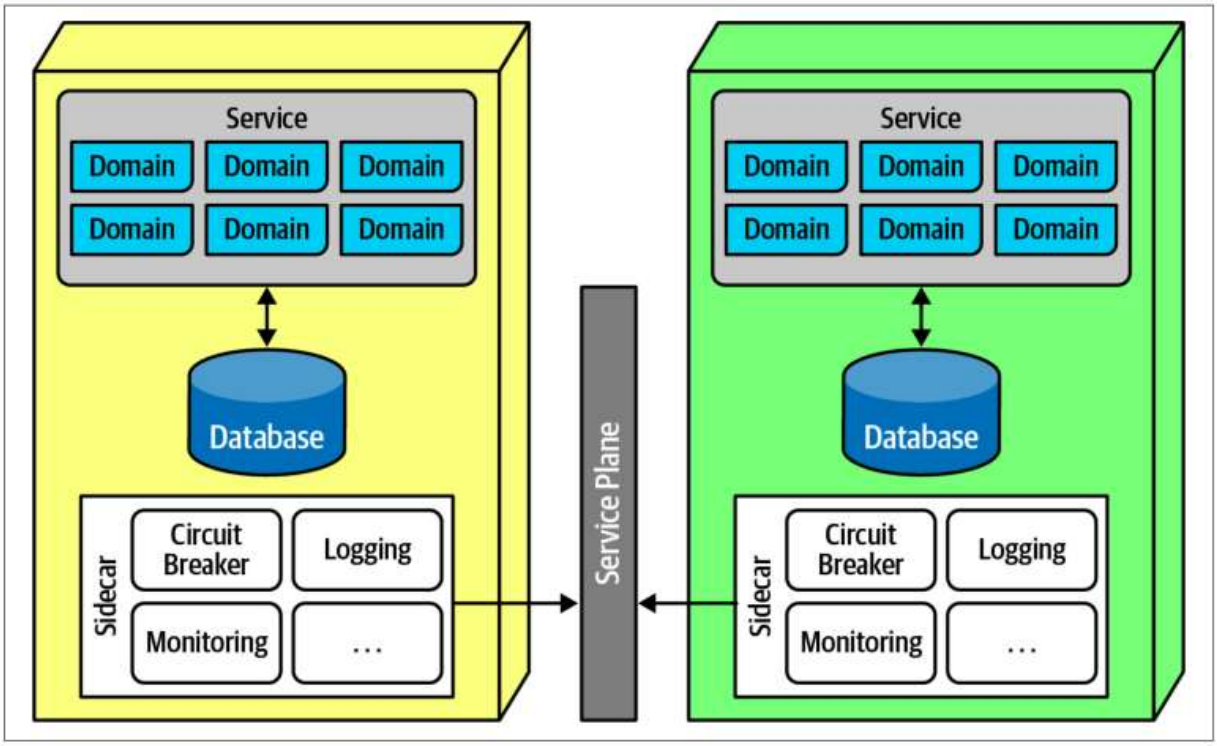


Figure 17-3. The service plane connects the sidecars in a service mesh

- *Service Discovery* tool faz parte de cada microsserviço para saber quais usar dinamicamente.
  - Pode estar na camade de API.

Frontends

- Difiícil ter o desacoplamento completo na interface do usuário.
- Duas alternativas comuns:
  - **Frontend Monolítico**
    - Se conecta à camada de API.
  - **Microfrontends**
    - Dividido em vários componentes.
    - e.g. **React**

Comunicação

- Pode ser **síncrona** ou **assíncrona**.
  - Espera ou não pela resposta.
- Para comunicação **síncrona**, costuma usar **protocol-aware[1] heterogeneous[2] interoperability[3]**:
  1. Microsserviços devem saber/descobrir como chamar uns aos outros
  2. Cada serviço pode ter uma stack de tecnologias diferente.

3. Operam entre si pela rede.
- Usar stacks diferentes é uma forma não-convencional de garantir desacoplamento.
    - Oposto de padronização de tecnologias.
    - Garante que cada problema tenha a solução justa.
- Para comunicação **assíncrona:**
    - Comum usar eventos e mensagens.
    - Arquitetura orientada a eventos.

Coreografia e Orquestração

- *Coreografia* utiliza comunicação análoga a eventos com broker - simbiótico um com outro.
    - Sem coordenação central.
- *Isomorfismo de domínio e arquitetura* é quanto que uma dada arquitetura se encaixa com um certo estilo de arquitetura.
- Pode ser intuitivo usar um orquestrador em uma arquitetura de microsserviço para operações mais complexas.
    - Mas é contra a filosofia de desacoplamento.
    - É uma escolha do arquiteto fazer isso ou não. **Trade-off**

Transações e Sagas

- Desacoplamento extremo gera desafio de coordenar transações.
- Talvez surja vontade de fazer transações que violem as fronteiras de domínio
    - NÃO DEVE SER FEITO. (mas há exceções)
    - Geralmente se resolve aumentando a granularidade dos serviços.
- Padrão Saga de microsserviços:
    - Bem popular

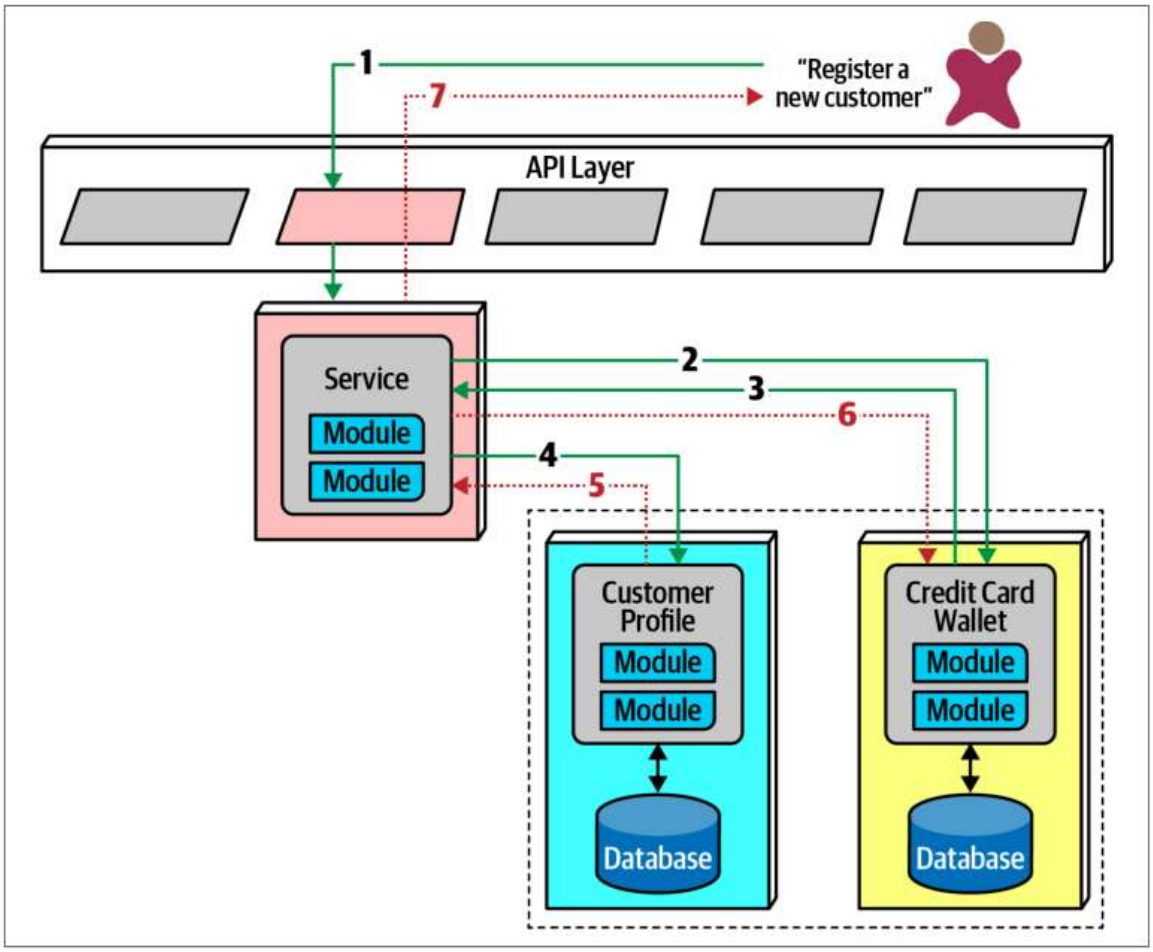


Figure 17-12. Saga pattern compensating transactions for error conditions

- Serviço que media interação entre microsserviços sincronamente.
- Consegue lidar com erros, pede para desfazer requisições se for preciso.
  - Mas desfazer costuma ser operação complexa.
- Pode se tornar complexo se houver necessidade de coordenação e esperas de respostas.

Características

Architecture characteristic	Star rating
Partitioning type	Domain
Number of quanta	1 to many
Deployability	★★★★
Elasticity	★★★★★
Evolutionary	★★★★★
Fault tolerance	★★★★
Modularity	★★★★★
Overall cost	★
Performance	★★
Reliability	★★★★
Scalability	★★★★★
Simplicity	★
Testability	★★★★

-
- Moderno
- Tolerância a falhas e confiabilidade são desafios, contornáveis.
- Muito escalável e evolucionário.
- Latência de rede pode ser um gargalo de desempenho.
- Particionado por domínio.