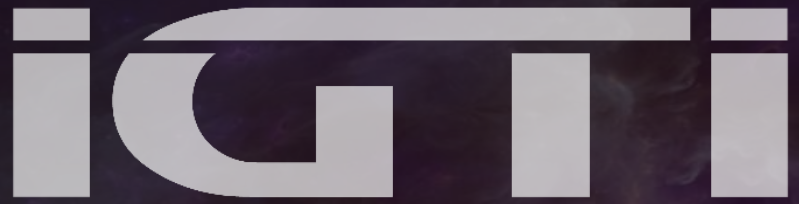


A aula interativa do **Módulo 3 - Bootcamp Arquiteto de Software começa** em breve!

Importante:

1) Não se esqueça de acessar a aula com seu e-mail cadastrado no ambiente de aprendizagem do IGTI e seu nome completo. Entrou com os dados errados? Saia da sala e entre novamente com os dados corretos.

2) Para sua frequência ser computada, responda a enquete no ambiente de aprendizagem, no horário indicado pelo professor. A enquete ficará disponível por 10 minutos.

3) Utilize o Chat para interagir com os colegas durante a aula, a ferramenta Raise Hands para pedir a palavra, e, em caso de dúvidas sobre o conteúdo, utilize o Q&A (perguntas e respostas) para que o professor tutor possa respondê-las.

Design Patterns, Estilos e Padrões Arquiteturais

SEGUNDA AULA INTERATIVA

PROF. VAGNER CLEMENTINO DOS SANTOS

Nesta aula

- Tipos de estilo arquiteturais.
- Falácias da computação distribuída.
- Padrões arquiteturais modernos.
- Exemplo de Circuit Breaker.

Acordos

- Pratique a Paciência.
- Pratique a Empatia.
- Pratique a Curiosidade.
- Pratique o Networking.

Tipos de estilos arquiteturais

- Os estilos de arquitetura podem ser classificados como monolítico ou distribuído.
- No monolítico é possível identificar um único deployment.
- No distribuído, o build resulta em dois ou mais “executáveis” que estão conectados por meio de protocolos de acesso remoto (*HTPP*, *gRPC*).

Exemplos

Tipos de Estilos Arquiteturais	
Monolítico	Distribuído
• **Arquitetura em Camada.** • **Pipeline.** • **Microkernel.**	• Arquitetura Orientada à Eventos. • Arquitetura Orientada à Serviço. • Microsserviços.

Falácias da Computação Distribuída

- A rede é confiável.
- Latência é zero.
- A largura de banda (bandwidth) é infinita.
- A rede é segura.
- A topologia da rede (roteadores, hubs, switch, firewall etc.) nunca muda.
- Existe um único administrador (dono) do sistema para colaborar e se comunicar.
- O custo em dinheiro de transportar algo pela rede, como uma chamada Restful, é zero.

Padrões modernos

- Circuit Breaker.
- Command and Query Responsibility Segregation (CQRS).
- Event Sourcing.
- Sidecar.
- Backend-for-Frontend.

Problema

1

Os sistemas distribuídos devem ser projetados levando em consideração possíveis falhas.

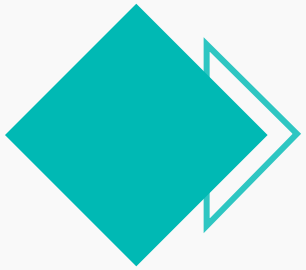
2

Serviços remotos podem não responder por vários motivos.

3

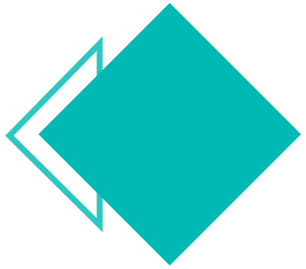
Implementar novas tentativas pode ajudar. Contudo, em casos de falha total, é inútil tentar novamente.

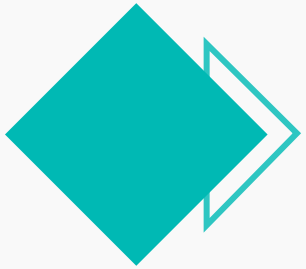
Circuit Breaker



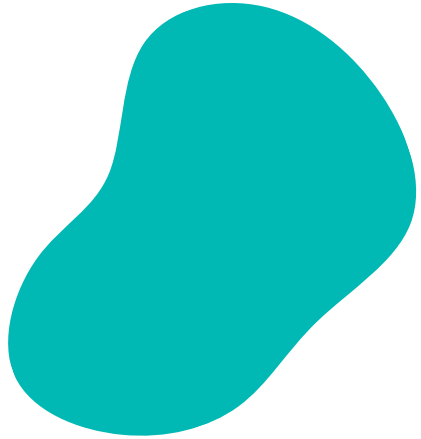
Pode impedir que um aplicativo execute repetidamente uma operação cuja falha é provável.

Atua como um proxy que monitora o número de falhas recentes ocorridas e decide se a operação deve continuar ou se deve retornar uma exceção.

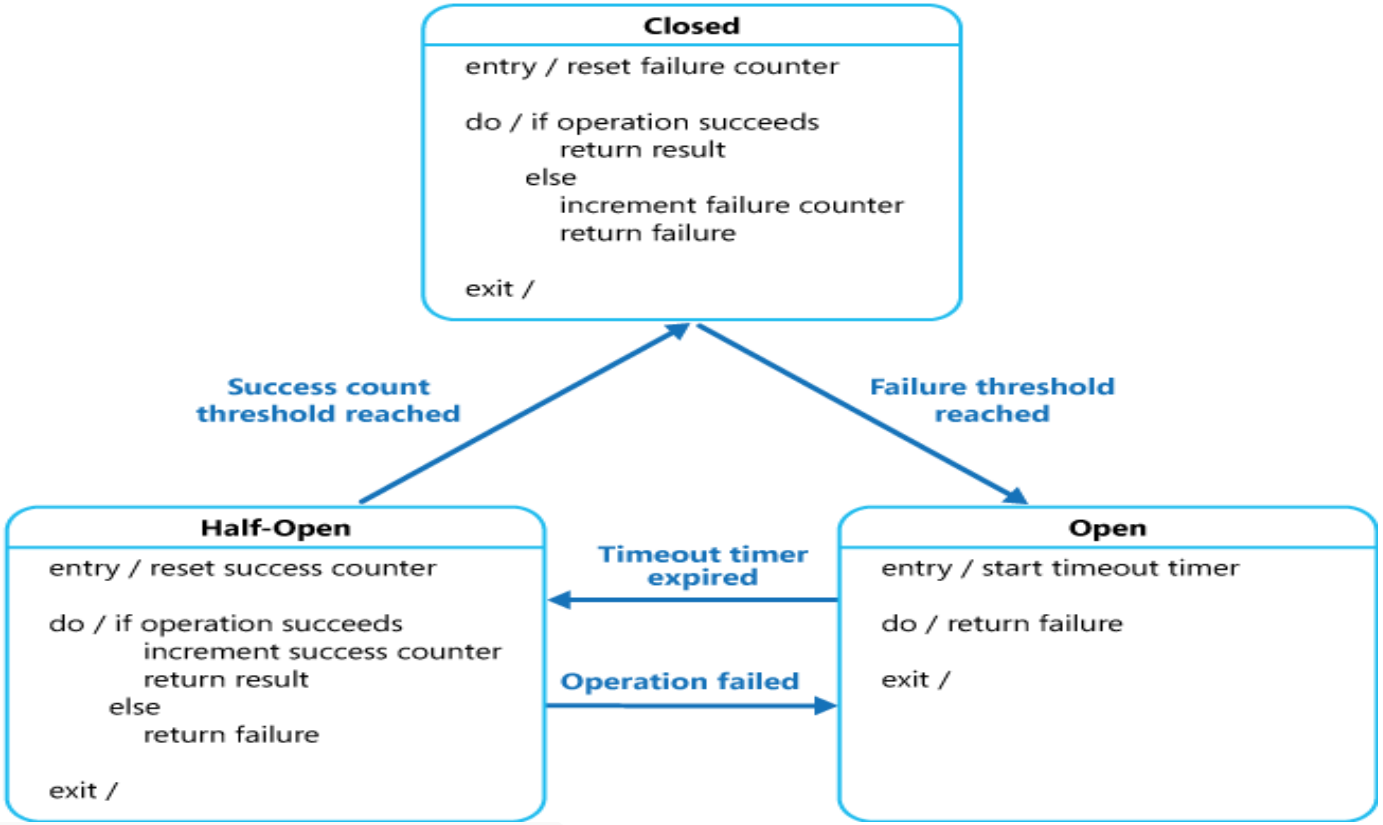




O proxy pode ser implementado como uma máquina de estados: **Fechado**, **Aberto** e **Entreaberto**.



Estados no Circuit Breaker



Fonte: https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker

Quando usar

- Para impedir que um aplicativo tente acessar um serviço remoto ou acessar um recurso compartilhado se a operação tiver alta probabilidade em falhar.

- Para tratar o acesso a recursos locais em um sistema. Ex.: estruturas de dados em memória.
- Como substituto para o tratamento de exceções na lógica de negócios da aplicação.

Problema

1

Nas arquiteturas tradicionais, o mesmo modelo de dados é usado para consultar e atualizar a base de dados.

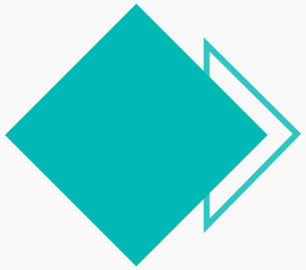
2

Em geral, leitura e escrita são assimétricas, com requisitos de desempenho e de escala muito diferentes.

3

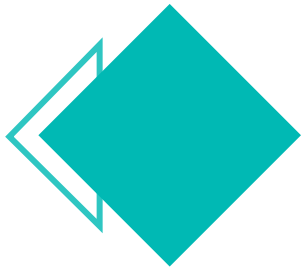
Utilizar uma única fonte de armazenamento para leitura e escrita pode acarretar em problemas de desempenho.

Command and Query Responsibility Segregation (CQRS)

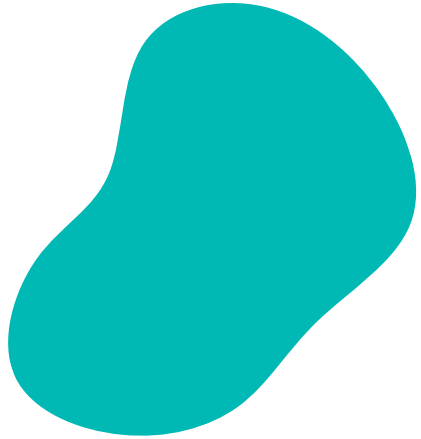


CQRS separa leituras e gravações em modelos diferentes, usando os conceitos de ***commands*** e ***queries***.

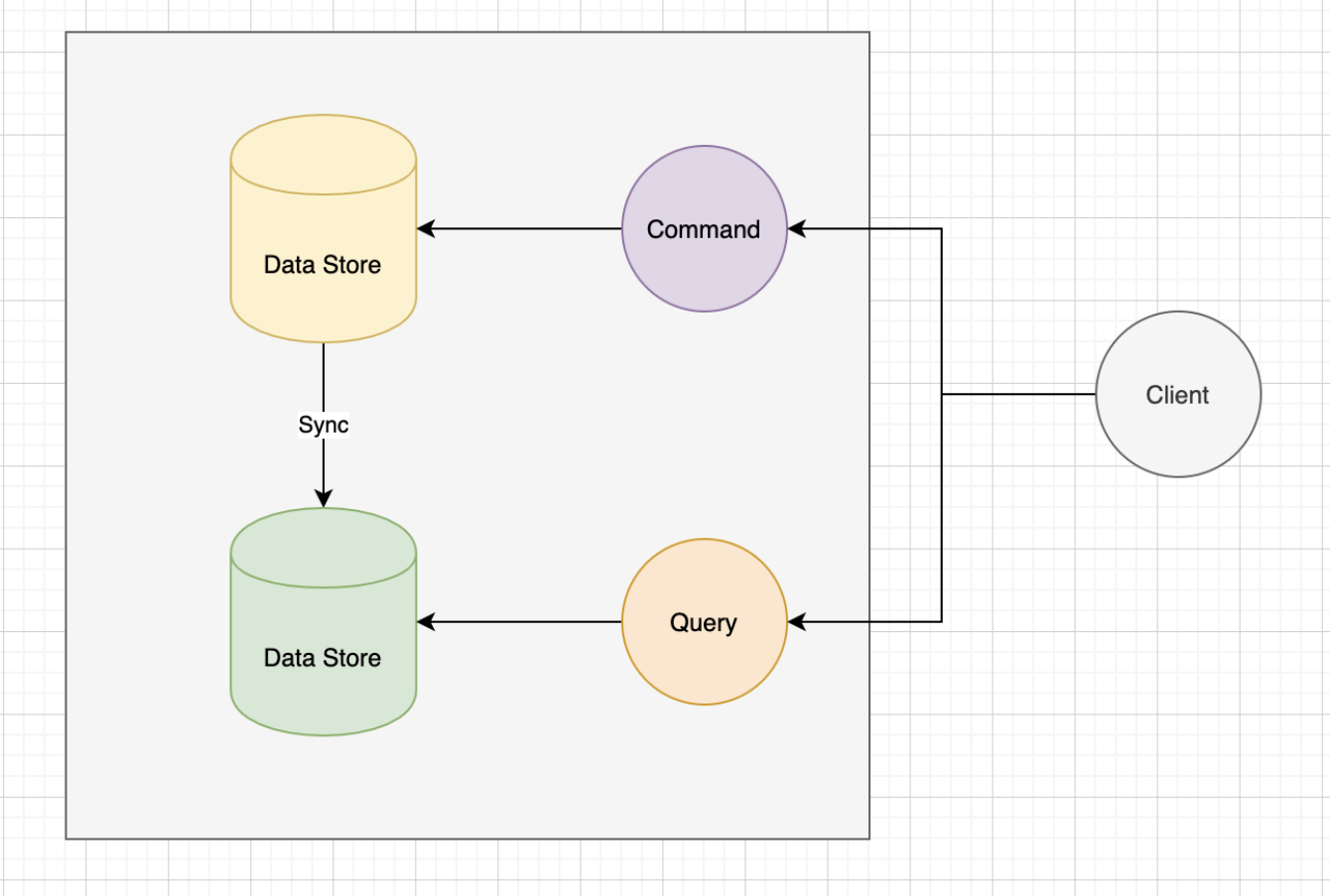
Commands devem ser baseados em tarefas e podem ser processados de maneira assíncrona.



Queries não devem modificar o banco e devem retornar um *DTO sem* conhecimento do domínio.



Funcionamento



Fonte: https://medium.com/better-programming/modern-day-architecture-design-patterns-for-software-professionals-9056ee1ed977#4c75

Quando usar

- Aplicação esperando um grande número de leituras e escritas.
- Quando você quer fazer o *tunning* das operações de leitura e escrita.

- O domínio ou as regras de negócio são simples.
- Em um CRUD, onde as operações de acesso a dados são suficientes.

Problema

1

As aplicações trabalham com dados mantendo o estado atual e atualizando-o quando necessário.

2

Atualizações diretas no banco de dados podem diminuir o desempenho e a capacidade de resposta.

3

Atualizações frequentes aumentam a probabilidade de inconsistência nos dados.

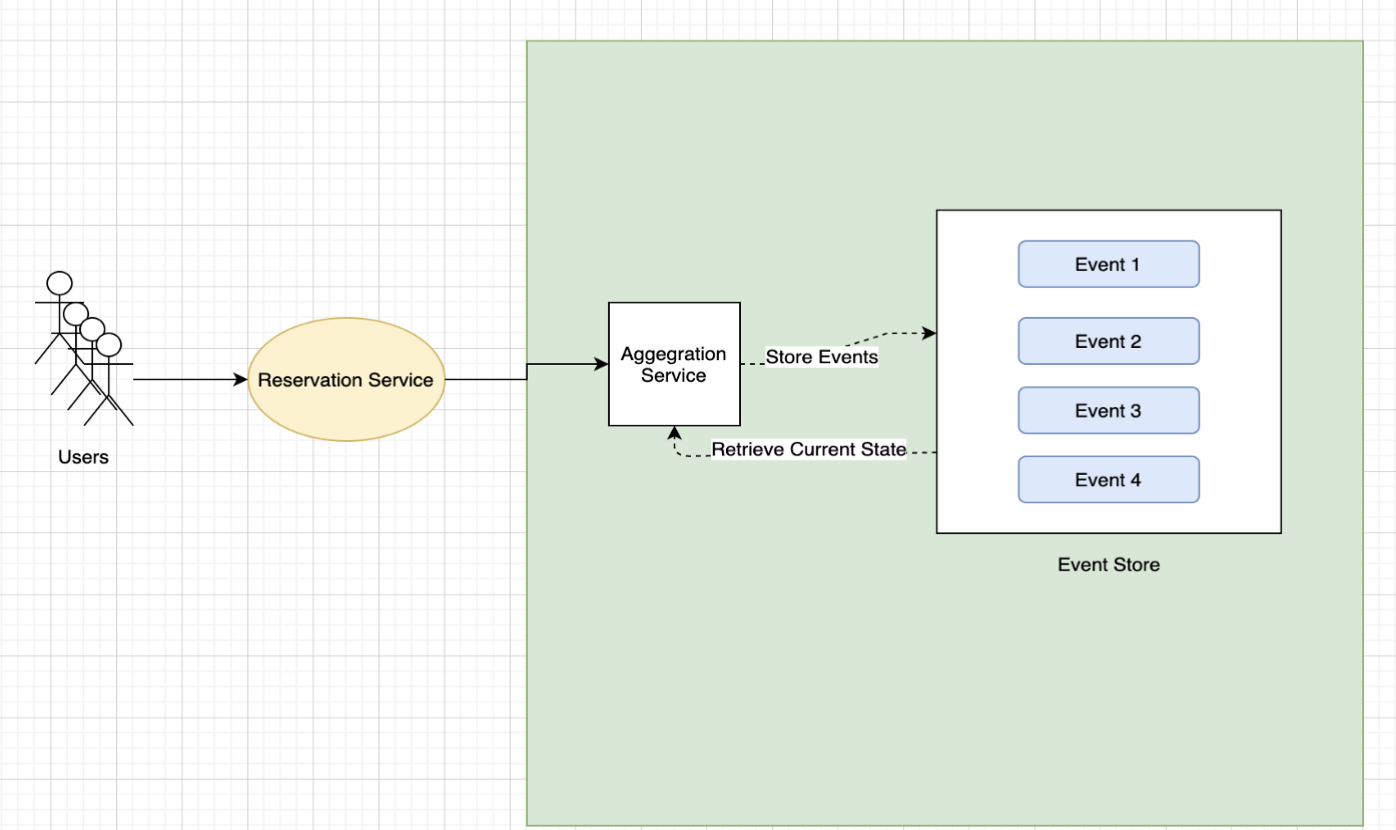
Event Sourcing

Uma sequencia de eventos é armazenada como um journal (*log sequencial*).

Uma visão agregada do *journal* representa o estado atual da aplicação.

Usado para sistemas que não podem arcar com o lock (isolamento) dos dados.

Funcionamento



Fonte: https://medium.com/better-programming/modern-day-architecture-design-patterns-for-software-professionals-9056ee1ed977#4c75

Quando usar

- Quando as operações regulares de CRUD não são suficientemente boas.
- Tipicamente adequado para sistemas de reserva ou de comércio eletrônico.
- Quando há requisito forte de auditoria.

- Domínios pequenos ou simples que funcionam bem com os mecanismos tradicionais de gerenciamento de dados.
- Sistemas em que há uma baixa ocorrência de atualizações nos mesmos dados.

Problema

1

Aplicações dependem de interesses transversais (ex.: monitoramento, log, configuração etc.).

2

Uma interrupção dessa funcionalidades poderia afetar alguns componentes ou toda a aplicação.

3

Essas tarefas periféricas poderiam ser implementadas como componentes ou serviços separados.

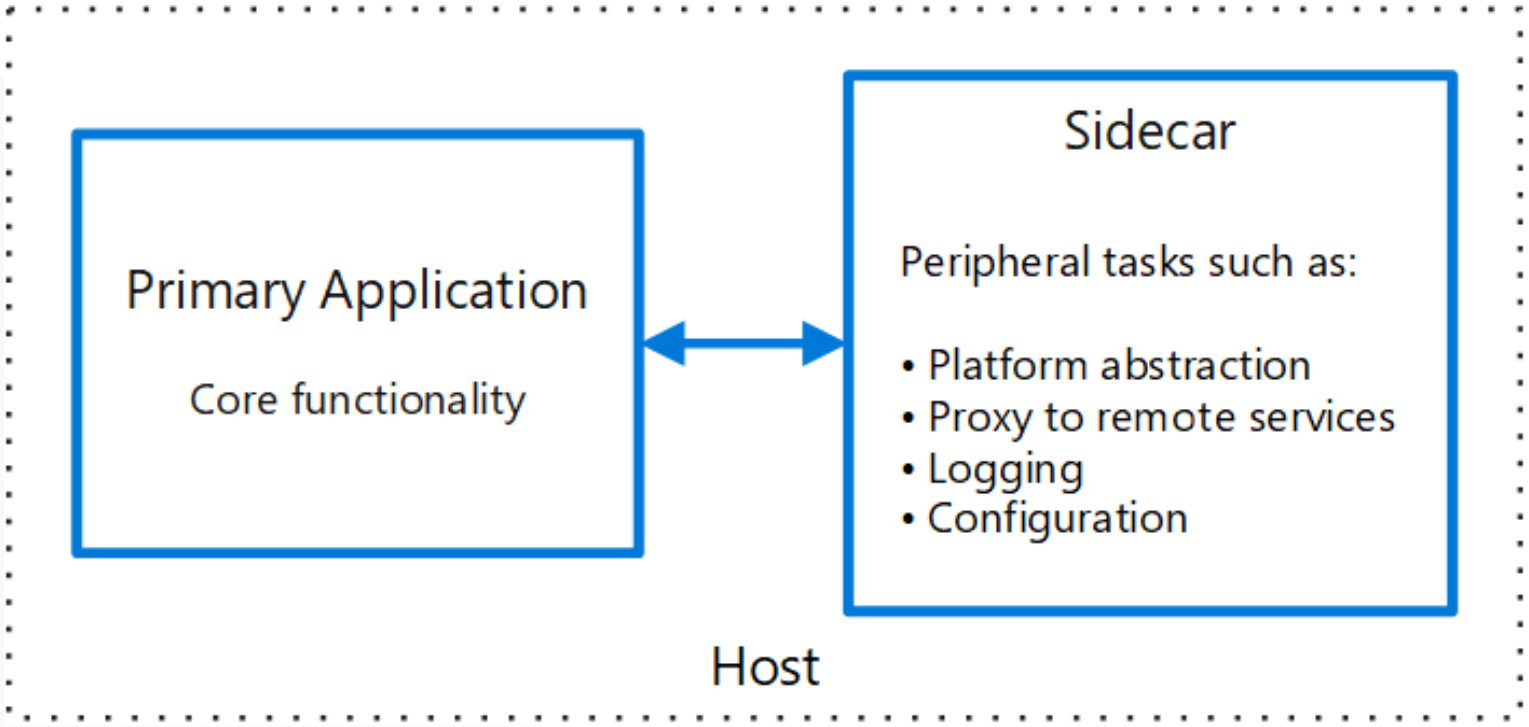
Sidecar

As funcionalidades principais (domínio) ficam na aplicação principal, em seu próprio processo ou container.

Define-se uma interface homogênea de comunicação do serviço com o *Sidecar.*

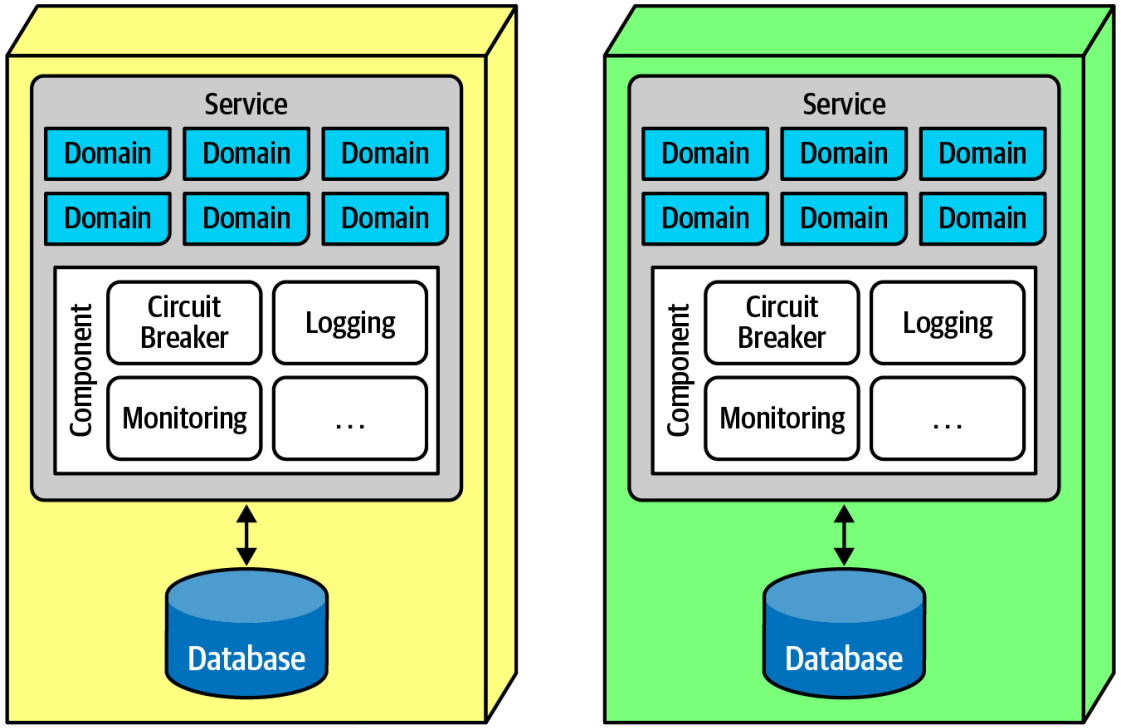
Um serviço com o papel de *Sidecar* não faz parte da aplicação, mas está conectado a ela.

Funcionamento



Fonte: https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar

Sidecar em microsserviços



Fonte: RICHARDS e FORD, 2020.

Quando usar

- Trabalhando com múltiplos e heterogêneos microsserviços no mesmo produto.
- Você precisa de um serviço que compartilhe do ciclo de vida geral de sua aplicação principal, mas que possa ser atualizado independentemente.

- Quando a comunicação entre processos precisa ser otimizada.
- Para aplicações onde o custo de implantação de um Sidecar para cada instância é alto.

Problema

1

Uma aplicação é inicialmente projetada para atender uma interface web.

2

Com o aumento da base de usuários é desenvolvido um aplicativo móvel que interage com o mesmo *backend*.

3

O backend torna-se um serviço para fins gerais, atendendo os requisitos da versão web e da mobile.

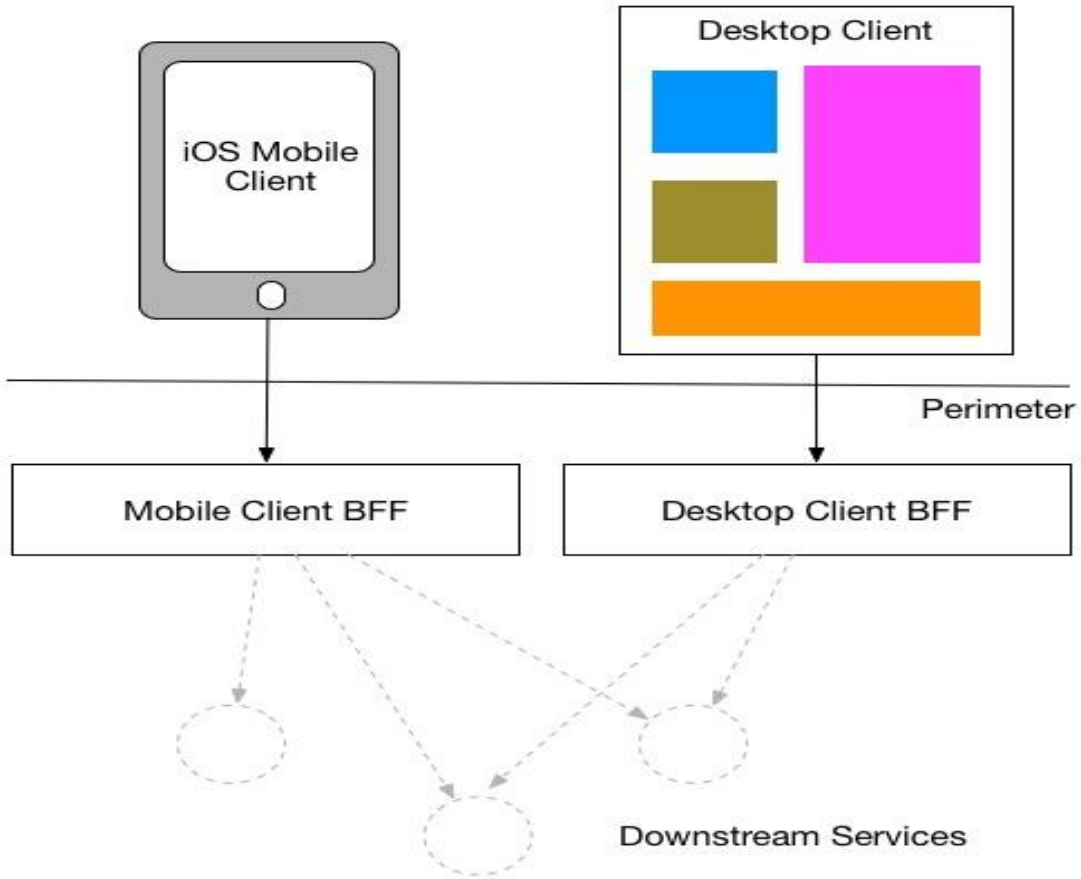
Backend-for-Frontend

Implementa um *backend* por interface de usuário.

Ajusta o comportamento do backend para melhor atender às necessidades do frontend, sem se preocupar em afetar as demais experiências do usuário.

Como cada backend é específico por interface gráfica, dessa forma pode ser otimizado.

Funcionamento



Fonte: https://samnewman.io/patterns/architectural/bff

Quando usar

- Um serviço backend de propósito geral está ficando complexo de manter.
- O usuário deseja otimizar o backend para as exigências específicas de cada interface do usuário.

- Não há muita diferença entre as requisições realizadas pelas diferentes interfaces gráficas ao *backend*.
- Quando apenas uma interface é usada para interagir com o backend.

Vamos praticar...

Exemplo

1

2

3

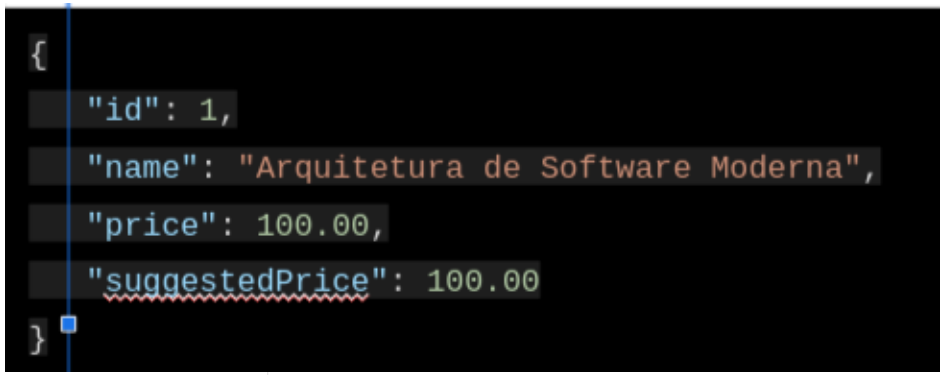
Em uma arquitetura de microsserviços temos uma API responsável por fornecer informações sobre produtos digitais.

O criador do produto define o preço do produto, contudo, o preço definido pode não ser tão interessante para uma venda (*Preço Psicológico*).

Como a definição do preço é um domínio de negócio específico (*DDD*), foi criado a *Price API*.

Get Product By ID

GET /api/v1/products/{product_id}



Get Price By Product

GET /prices/products/{product_id}

Responses

```
{
  "product_id": 1,
  "price": 122.99
}
```

```
{
  "message": "Internal Server Error"
}
```

Problemas...

Recentemente a Price API está ficando indisponível por um curto período.

Foi definido timeout ao fazer chamadas para serviço, contudo os clientes da Product API ainda continuam recebendo os erros por causa da PRICE API.

Para o negócio, devolver o preço base é melhor do que retornar um erro para o usuário.

Solução...

Estória do Usuário

AO INVÉS DO cliente receber um erro quando a Price API estiver indisponível,

GOSTARIA DE retornar o preço base e registrar que a Price API estava fora do ar.

Critérios de Aceitação

- **CA01 - Price API retorna sucesso:**

  Dado uma chamada para a Price API com um produto válido.

  Quando a Price API retorna sucesso (status code 200).

  Então eu retorno o produto com o preço fornecido pela Price API.

- **CA02 - Price API retorna erro:**

  Dado uma chamada para a Price API com um produto válido.

  Quando a Price API retorna um erro (status code 5xx).

  Então eu retorno o produto com o seu preço base.

  E registro o problema da Price API.

Mao na Massa

https://github.com/vagnerclementino/circuit-breaker

Espaço para dúvidas

Referências

- https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker
- https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english
- https://spring.io/guides/gs/circuit-breaker/
- https://github.com/Netflix/Hystrix/wiki/How-it-Works
- https://github.com/mtakaki/dropwizard-circuitbreaker