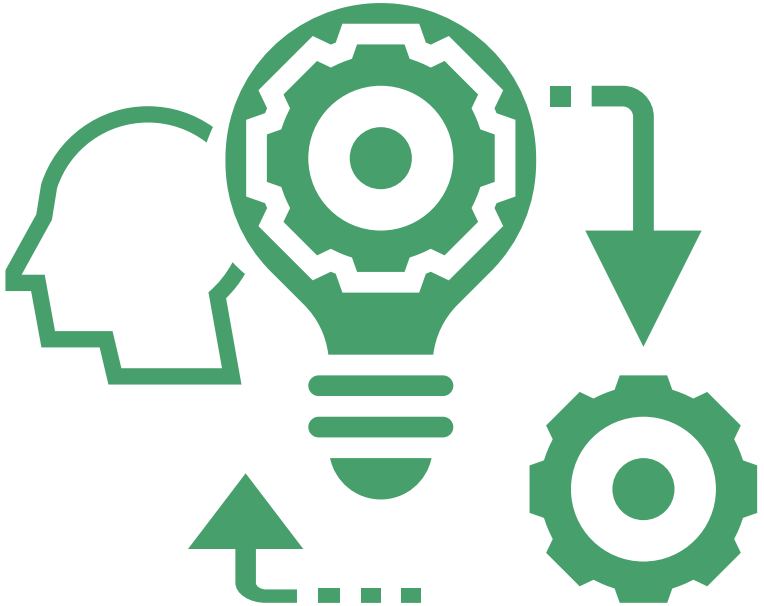


Paradigmas de Programação

Nossa aula – Como vai ser ???

- Entregáveis
- Microservices Patterns
- Resilience Patterns
- Prática – Circuit Breaker, Fallback e Cache

Entregáveis

Prezados alunos,

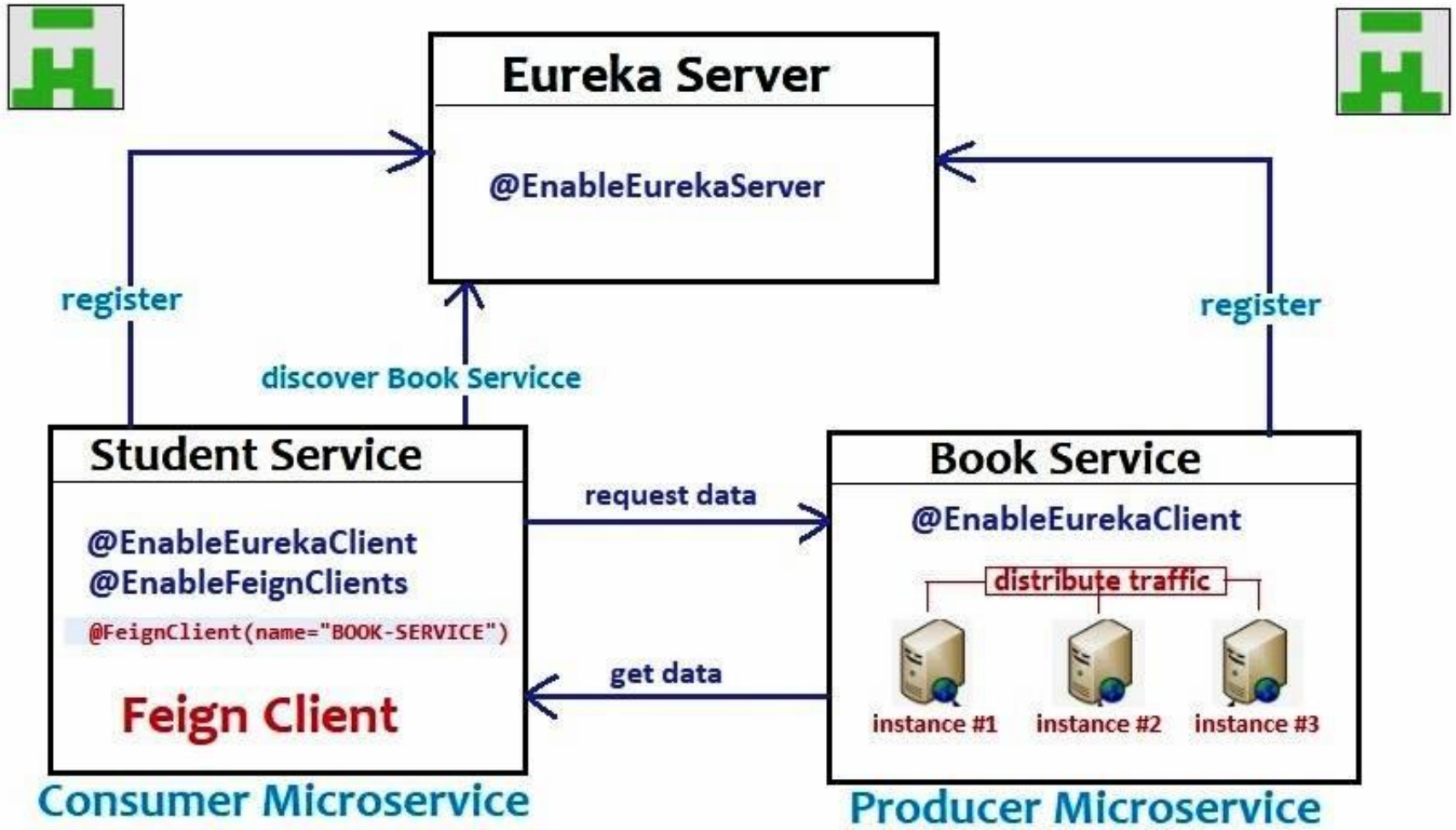
Para aprofundar nossos conhecimentos sobre a comunicação entre microservices, solicito que realizem uma pesquisa sobre dois métodos principais de comunicação: HTTP (APIs REST) e mensageria AMQP. Sua pesquisa deve abordar os seguintes pontos:

1. **Comunicação via HTTP (APIs REST):**
    - **Princípios e Funcionamento:** Explique o que são APIs REST e como elas funcionam para comunicação entre microservices.
    - **Vantagens e Desvantagens:** Discuta as principais vantagens e desvantagens de usar APIs REST para comunicação entre microservices.
    - **Casos de Uso:** Dê exemplos de situações em que a comunicação via HTTP é mais adequada.
2. **Mensageria AMQP:**
    - **Princípios e Funcionamento:** Descreva o que é a mensageria AMQP (Advanced Message Queuing Protocol) e como ela facilita a comunicação entre microservices.
    - **Vantagens e Desvantagens:** Analise as principais vantagens e desvantagens do uso de mensageria AMQP.
    - **Casos de Uso:** Forneça exemplos de cenários onde a mensageria AMQP é a solução mais eficiente.

1. Comunicação via HTTP (APIs REST):

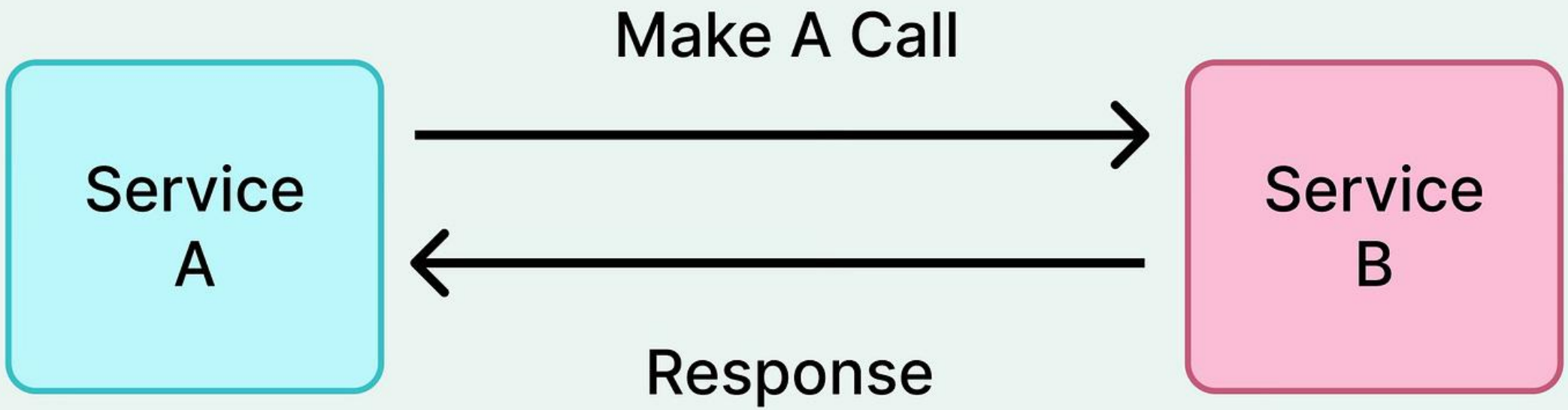
- **Princípios e Funcionamento:** APIs REST (Representational State Transfer) são uma forma de comunicação entre microservices que utiliza o protocolo HTTP para realizar operações em recursos. Cada recurso é acessado através de uma URL, e as operações são realizadas usando os métodos HTTP padrão: GET (para obter dados), POST (para criar novos dados), PUT (para atualizar dados existentes) e DELETE (para remover dados). REST é baseado em princípios como statelessness (cada requisição deve conter todas as informações necessárias), client-server architecture (separação entre cliente e servidor) e cacheability (respostas podem ser armazenadas em cache).
- **Vantagens e Desvantagens:**
  - **Vantagens:** Simplicidade, fácil integração com outras tecnologias, e a utilização de padrões bem estabelecidos. REST é amplamente suportado e pode ser facilmente escalado.
  - **Desvantagens:** Overhead de cada requisição HTTP, falta de suporte nativo para comunicação assíncrona, e a necessidade de gerenciamento explícito de estados e erros. Em sistemas de alta carga, o desempenho pode ser uma limitação.
- **Casos de Uso:** APIs REST são adequadas para sistemas que requerem interação simples e direta entre clientes e servidores, como aplicativos web e serviços que precisam de interoperabilidade com diferentes plataformas. Elas são frequentemente usadas em arquiteturas de microservices para comunicação entre serviços.

Implementing Spring Cloud OpenFeign : @FeignClient



@javatechonline.com

Synchronous Communication

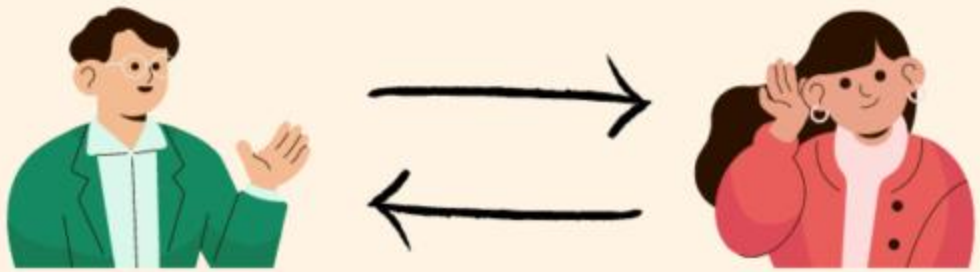


Service A makes a call to Service B and waits for the response

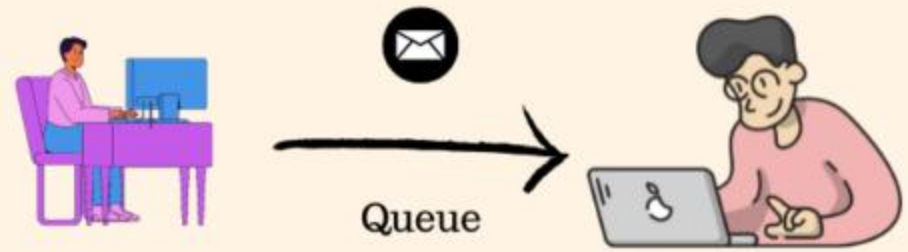
2. Mensageria AMQP:

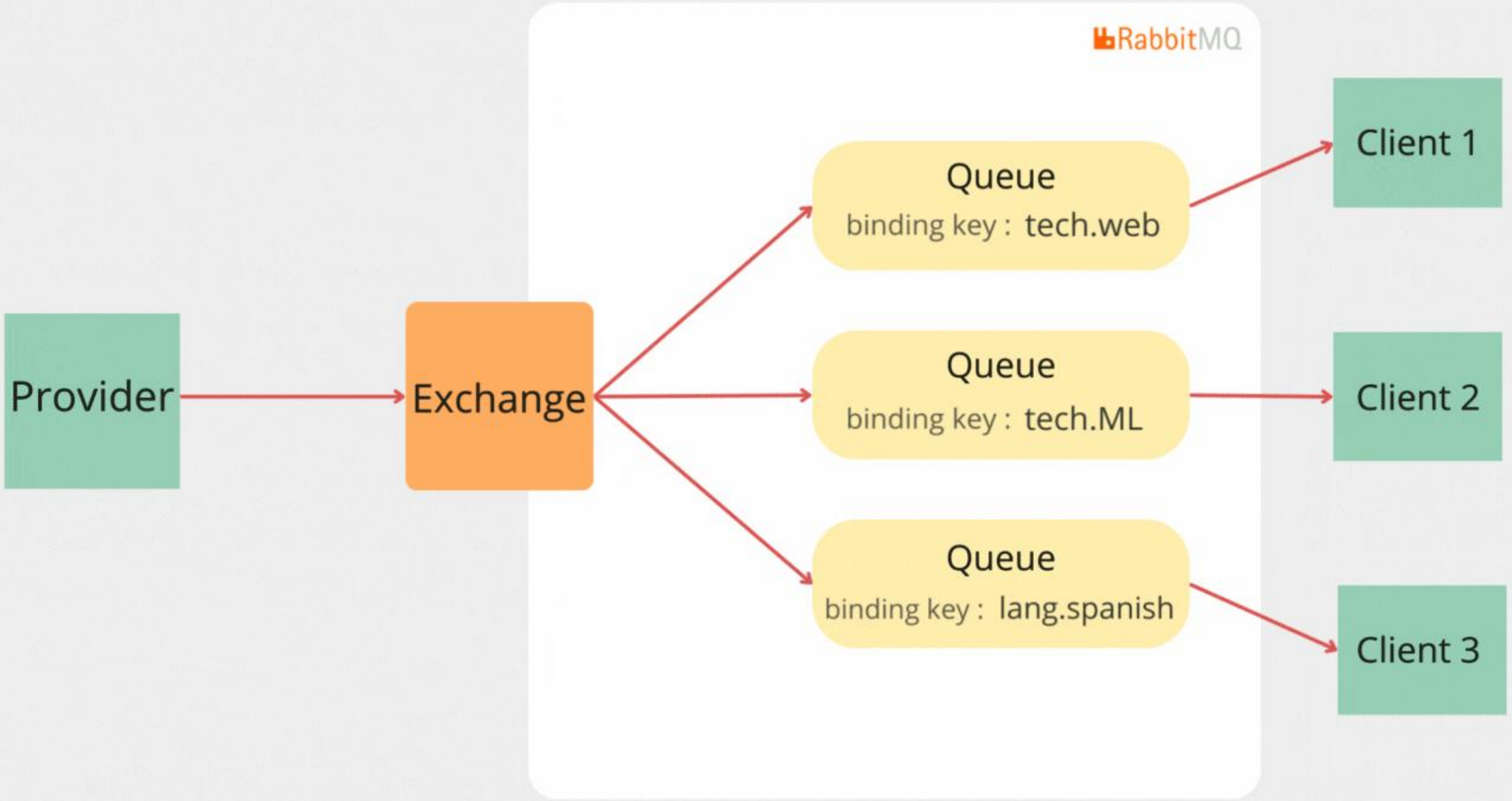
- **Princípios e Funcionamento:** AMQP (Advanced Message Queuing Protocol) é um protocolo de mensageria que facilita a comunicação assíncrona entre microservices. Utiliza um modelo baseado em filas e tópicos para gerenciar mensagens. Em AMQP, as mensagens são enviadas para uma fila e os consumidores as processam quando estiverem prontos. O protocolo garante a entrega e o ordenamento das mensagens, proporcionando uma comunicação mais robusta e resiliente entre serviços desacoplados.
- **Vantagens e Desvantagens:**
  - **Vantagens:** Desacoplamento entre serviços, suporte a comunicação assíncrona, e garantia de entrega de mensagens. AMQP é ideal para cenários que requerem alta escalabilidade e processamento de mensagens em grandes volumes.
  - **Desvantagens:** Maior complexidade na configuração e manutenção do sistema de mensageria, necessidade de gerenciamento de filas e tópicos, e possíveis desafios na garantia de entrega em ambientes distribuídos.
- **Casos de Uso:** Mensageria AMQP é eficaz em cenários que envolvem processamento de tarefas assíncronas e alta carga de trabalho, como sistemas de processamento de pedidos, plataformas de e-commerce e aplicações que precisam de alta disponibilidade e resiliência.

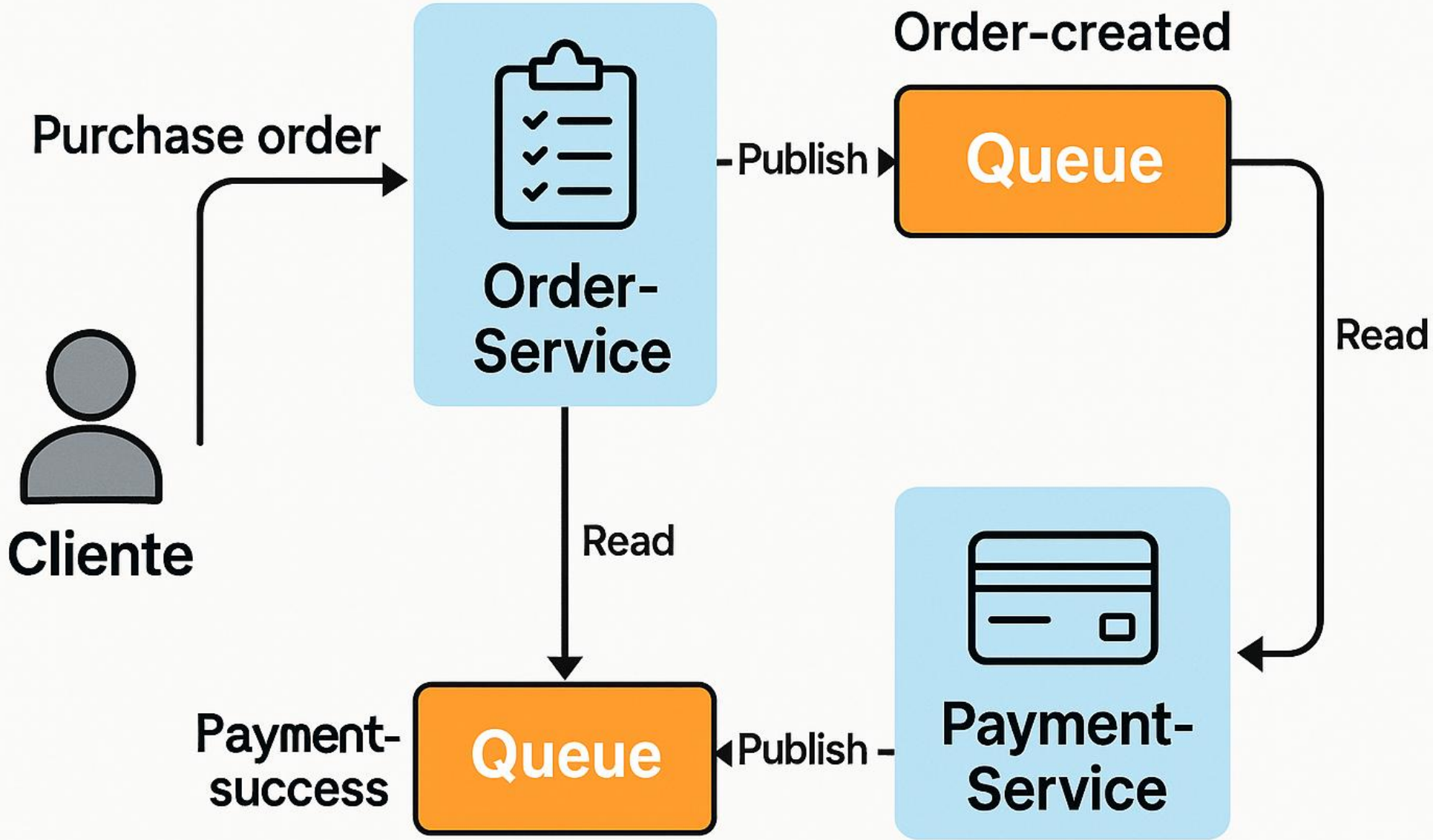
Synchronous



Asynchronous




RabbitMQ
Provider
Exchange
Queue
binding key : tech.web
Queue
binding key : tech.ML
Queue
binding key : lang.spanish
Client 1
Client 2
Client 3


Order-created
Purchase order
Publish
Queue
Order-
Service
Read
Cliente
Read
Payment-
success
Queue
Publish
Payment-
Service

Microservices Patterns

About Microservices.io

Microservices.io is brought to you by Chris Richardson. Experienced software architect, author of POJOs in Action, the creator of the original CloudFoundry.com, and the author of Microservices patterns.

ASK CHRIS

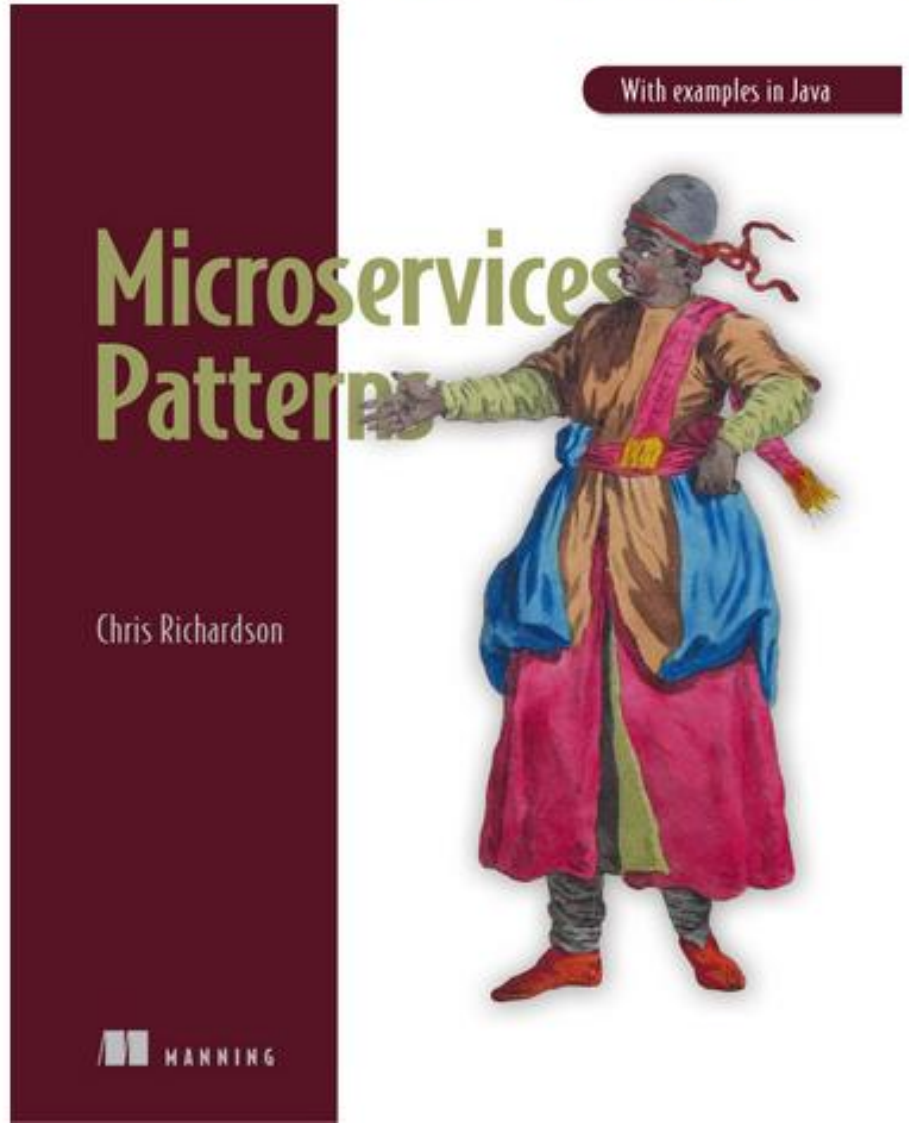
Got a question about microservices?

Fill in this form. If I can, I'll write a blog post that answers your question.

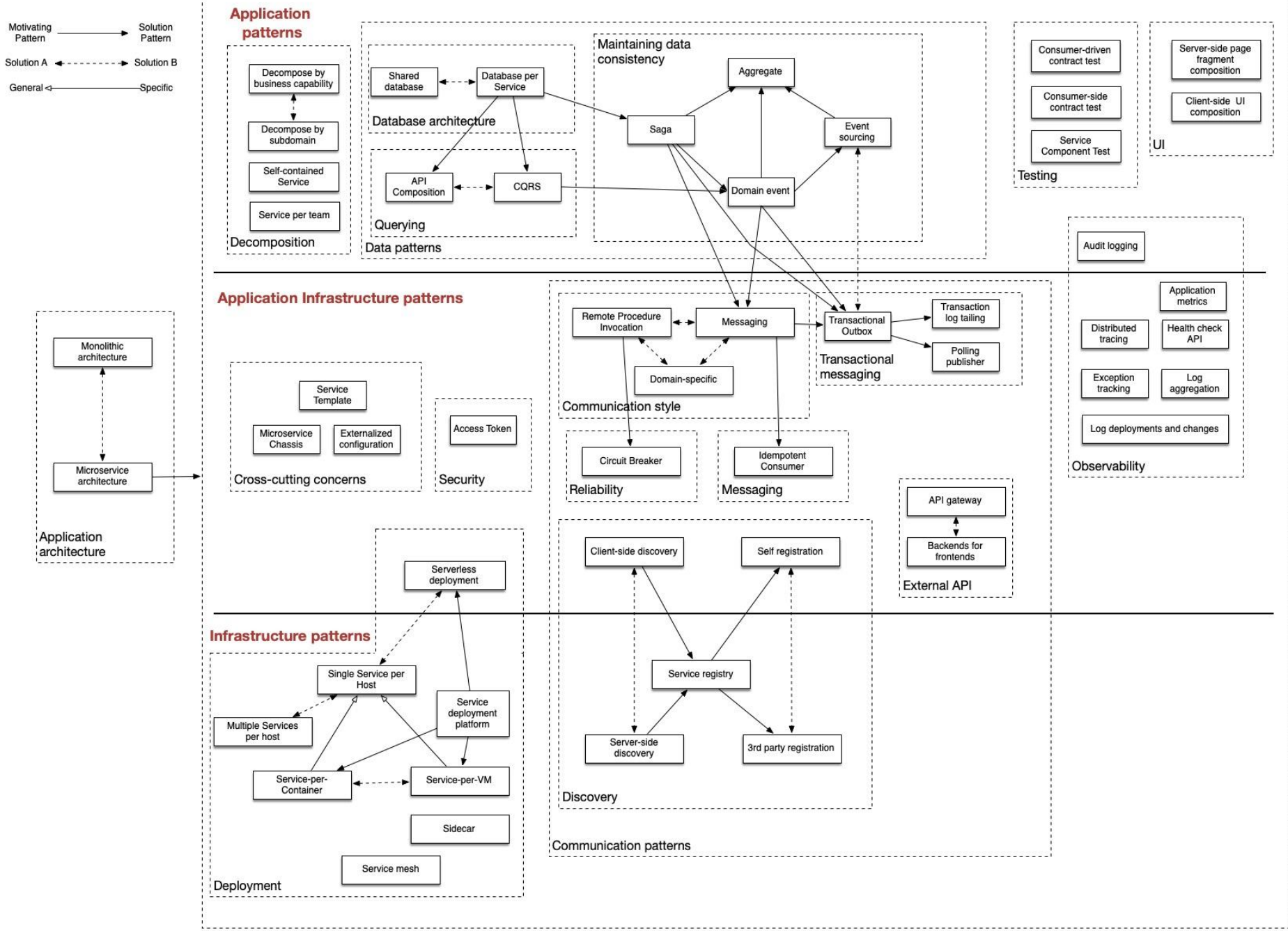
?

NEED HELP?

Chris is a software architect and serial entrepreneur. He is a Java
applications with frameworks such as Spring and Hibernate. Chris



The Microservice Architecture Pattern Language



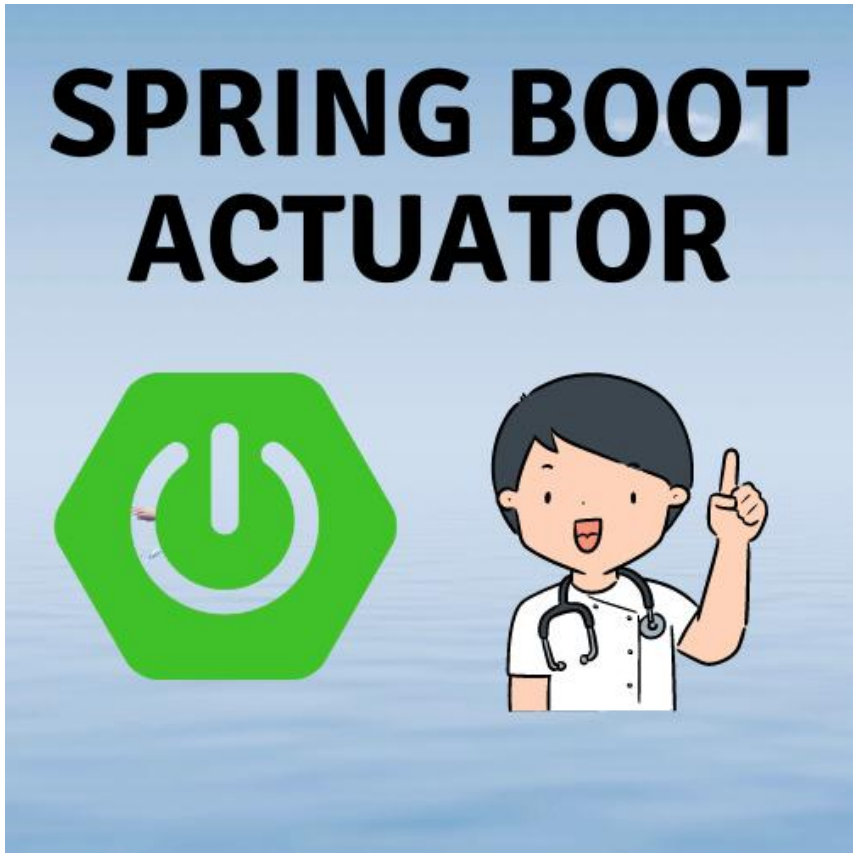
Microservices Patterns

Microservices Patterns

Pattern: Health Check API

Microservices Patterns

Pattern: Health Check API



Microservices Patterns

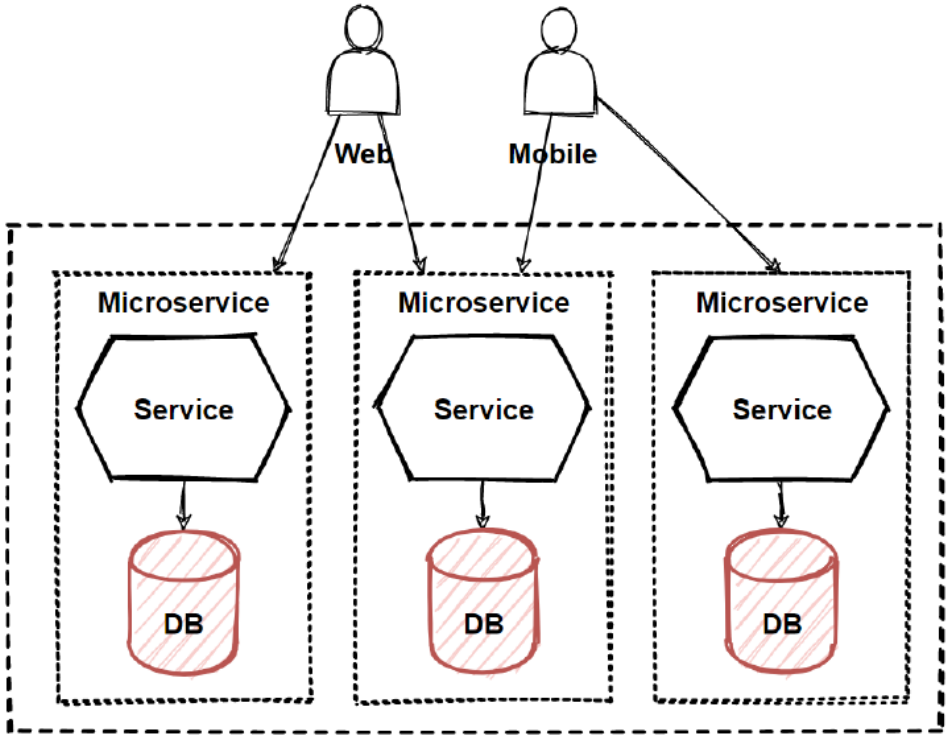
Pattern: Health Check API

Pattern: Database per service

Microservices Patterns

Pattern: Health Check API

Pattern: Database per service



Microservices Patterns

Pattern: Health Check API

Pattern: Database per service

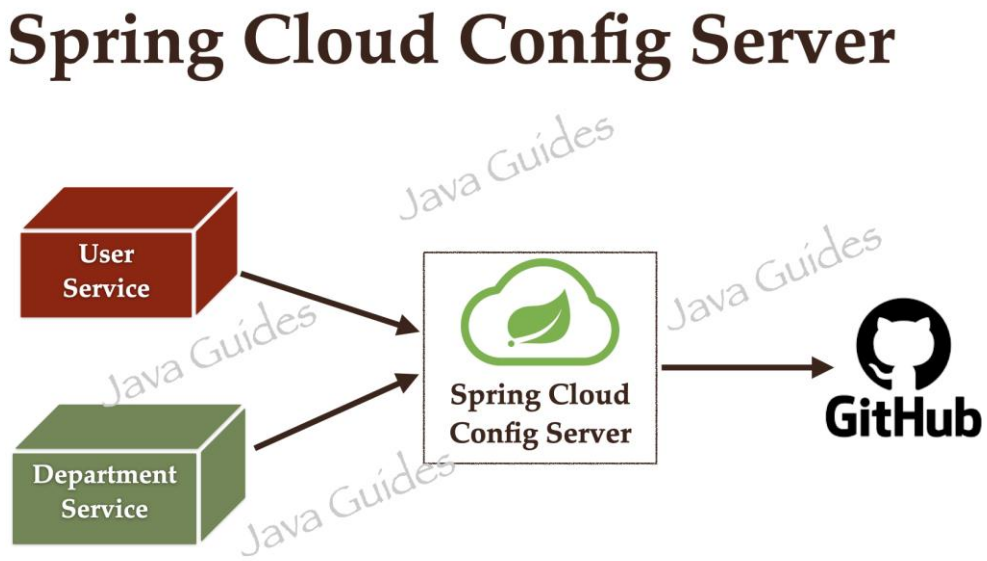
Pattern: Externalized configuration

Microservices Patterns

Pattern: Health Check API

Pattern: Database per service

Pattern: Externalized configuration



Microservices Patterns

Pattern: Health Check API

Pattern: Database per service

Pattern: Externalized configuration

Pattern: Service registry

- Pattern: Self Registration

Microservices Patterns

Pattern: Health Check API

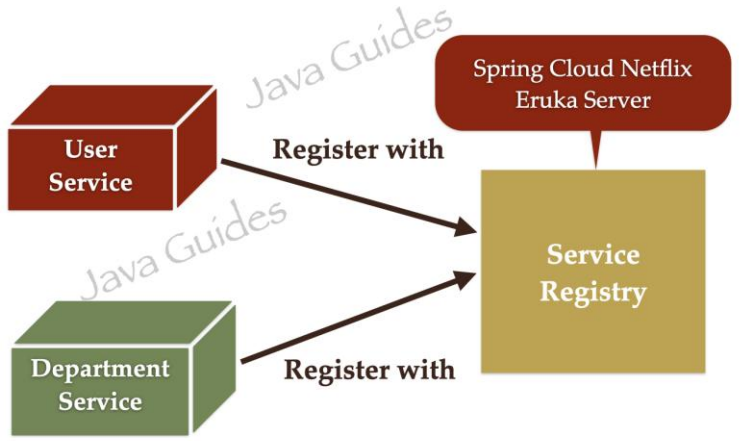
Pattern: Database per service

Pattern: Externalized configuration

Pattern: Service registry

Pattern: Self Registration

Spring Cloud Netflix Eureka Server



Microservices Patterns

Pattern: Health Check API

Pattern: Database per service

Pattern: Externalized configuration

Pattern: Service registry

- Pattern: Self Registration

Pattern: Client-side service discovery

Microservices Patterns

Pattern: Health Check API

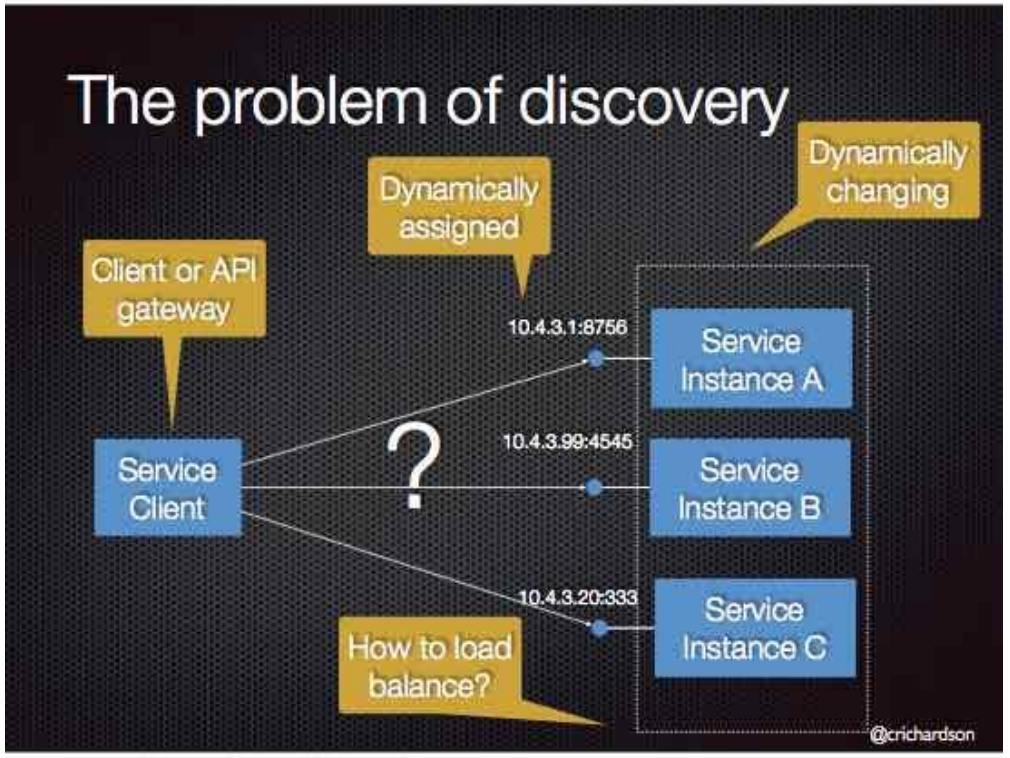
Pattern: Database per service

Pattern: Externalized configuration

Pattern: Service registry

Pattern: Self Registration

Pattern: Client-side service discovery



Microservices Patterns

Pattern: Health Check API

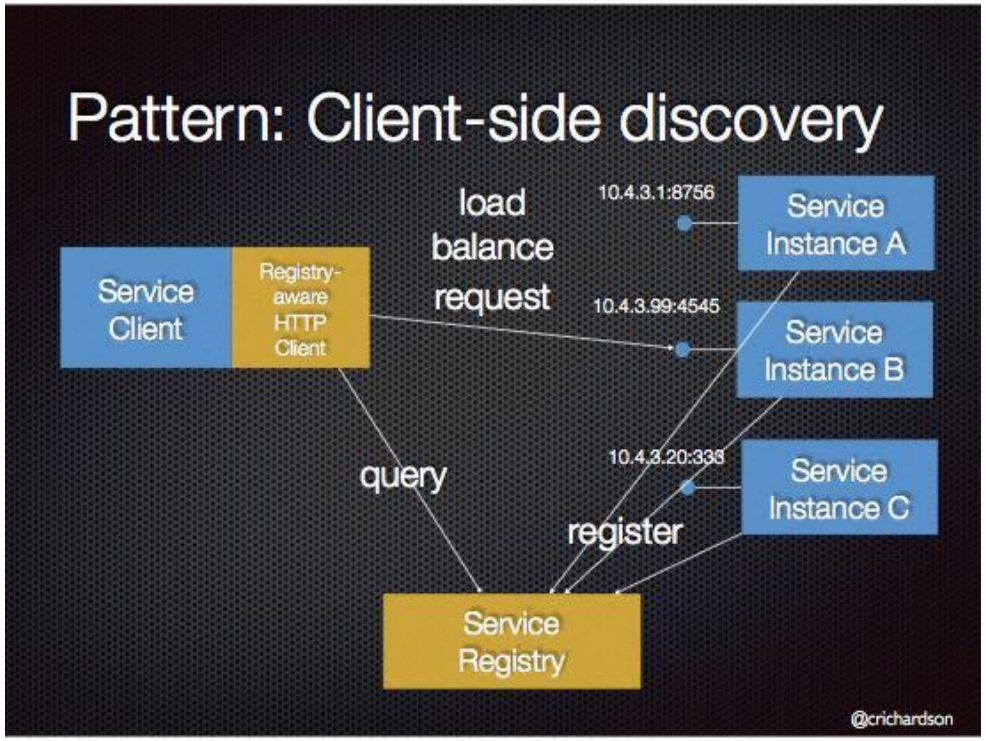
Pattern: Database per service

Pattern: Externalized configuration

Pattern: Service registry

Pattern: Self Registration

Pattern: Client-side service discovery



Resilience Patterns

Constuindo sistemas resilientes

O que significa “Resiliência”???

O que significa “Resiliência”???

Dicionário

Definições de Oxford Languages · Saiba mais

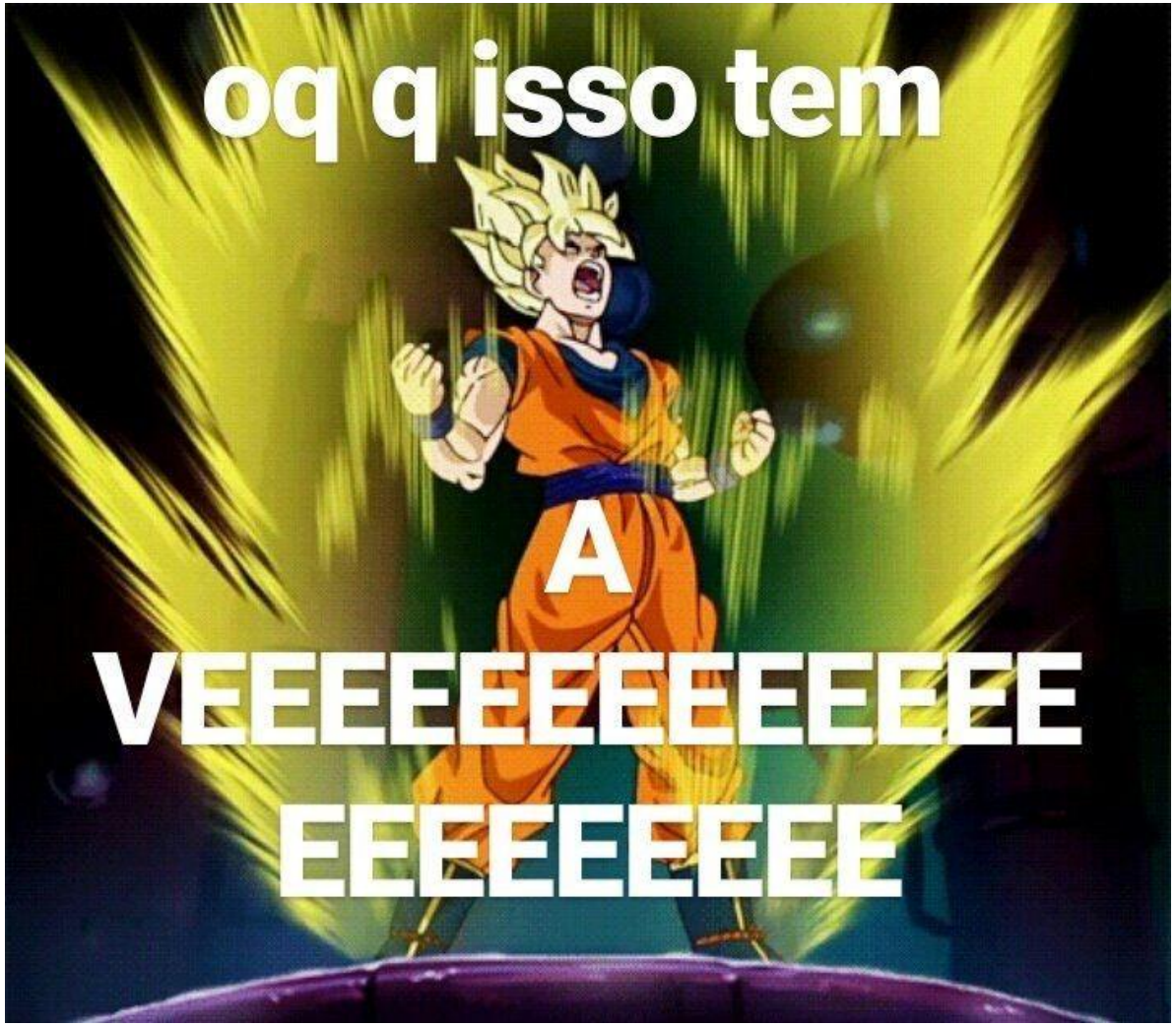
resiliência

substantivo feminino

1. **FÍSICA**
   propriedade que alguns corpos apresentam de retornar à forma original após terem sido submetidos a uma deformação elástica.
2. **FIGURADO**
   capacidade de se recobrar facilmente ou se adaptar à má sorte ou às mudanças.

O que significa “Resiliência”???

Certo, mas onde isso entra em Sistemas ???



A Importância da Resiliência em Microsserviços

- **Falhas são inevitáveis:** Falhas de rede, tempo de resposta lento, ou indisponibilidade de serviços são comuns.
- **Alta Disponibilidade:** Os serviços precisam funcionar mesmo quando ocorrem falhas em uma parte do sistema = Online 24/7
- **Isolamento de falhas:** Independência dos serviços. Se um serviço falha, ele não deve afetar outros serviços.
- **Recuperação automática:** Um sistema resiliente é capaz de se recuperar de falhas automaticamente.
- **Experiência do usuário:** Impacto das falhas nos usuários finais minimizado. Isso pode ser feito por meio de fallbacks (soluções alternativas), degradação graciosa (onde funcionalidades críticas continuam a operar), ou retries controlados (tentativas de reconectar ao serviço falho).

Resilience Patterns (Padrões de Resiliência)

- São práticas e estratégias usadas para tornar sistemas distribuídos mais robustos e capazes de lidar com falhas.

- Em arquiteturas de microservices, a resiliência é crítica porque esses sistemas são compostos por muitos serviços independentes que podem falhar de forma imprevisível.

- Implementar padrões de resiliência ajuda a minimizar o impacto dessas falhas e manter o sistema funcionando de maneira adequada.

Principais Padrões de Resiliência

- Rate Limiting
- Retry
- Timeout
- Fallback
- Caching
- Bulkhead
- Circuit Breaker

Rate Limiting

- Controla o número de solicitações que um serviço pode aceitar em um determinado intervalo de tempo, protegendo-o contra sobrecarga. Isso é especialmente importante em situações onde há picos de tráfego inesperado.
- **Exemplo**: Um microservice de autenticação pode ser configurado para aceitar no máximo 1000 requisições por segundo, negando as requisições adicionais até que haja capacidade disponível.
- **Benefícios**: Evita a sobrecarga de serviços críticos e assegura que o sistema possa processar as requisições de forma estável, mesmo sob alta demanda.

Retry

- Tenta realizar a operação novamente após uma falha temporária, com a esperança de que a falha seja transitória. Normalmente, ele é combinado com uma estratégia de espera exponencial, onde o intervalo entre as tentativas aumenta exponencialmente após cada falha.
- **Exemplo**: Um serviço de pagamento falha temporariamente. O sistema pode esperar alguns segundos e tentar a operação novamente, aumentando o tempo de espera a cada nova tentativa.
- **Benefícios**: Ajuda a recuperar falhas temporárias, como falhas de rede ou indisponibilidades breves de serviços.

Timeout

- Define um limite de tempo para as chamadas a serviços externos. Se um serviço não responder dentro do tempo configurado, a chamada é abortada, e uma ação de recuperação pode ser tomada.
- **Exemplo**: Um microservice A faz uma chamada a um microservice B, mas se B demorar mais de 3 segundos para responder, A cancela a requisição e, por exemplo, pode retornar uma resposta padrão ou informar ao usuário que o serviço está indisponível.
- **Benefícios**: Evita que o sistema fique "preso" aguardando respostas que podem nunca vir, liberando recursos e melhorando a eficiência do sistema.

Fallback

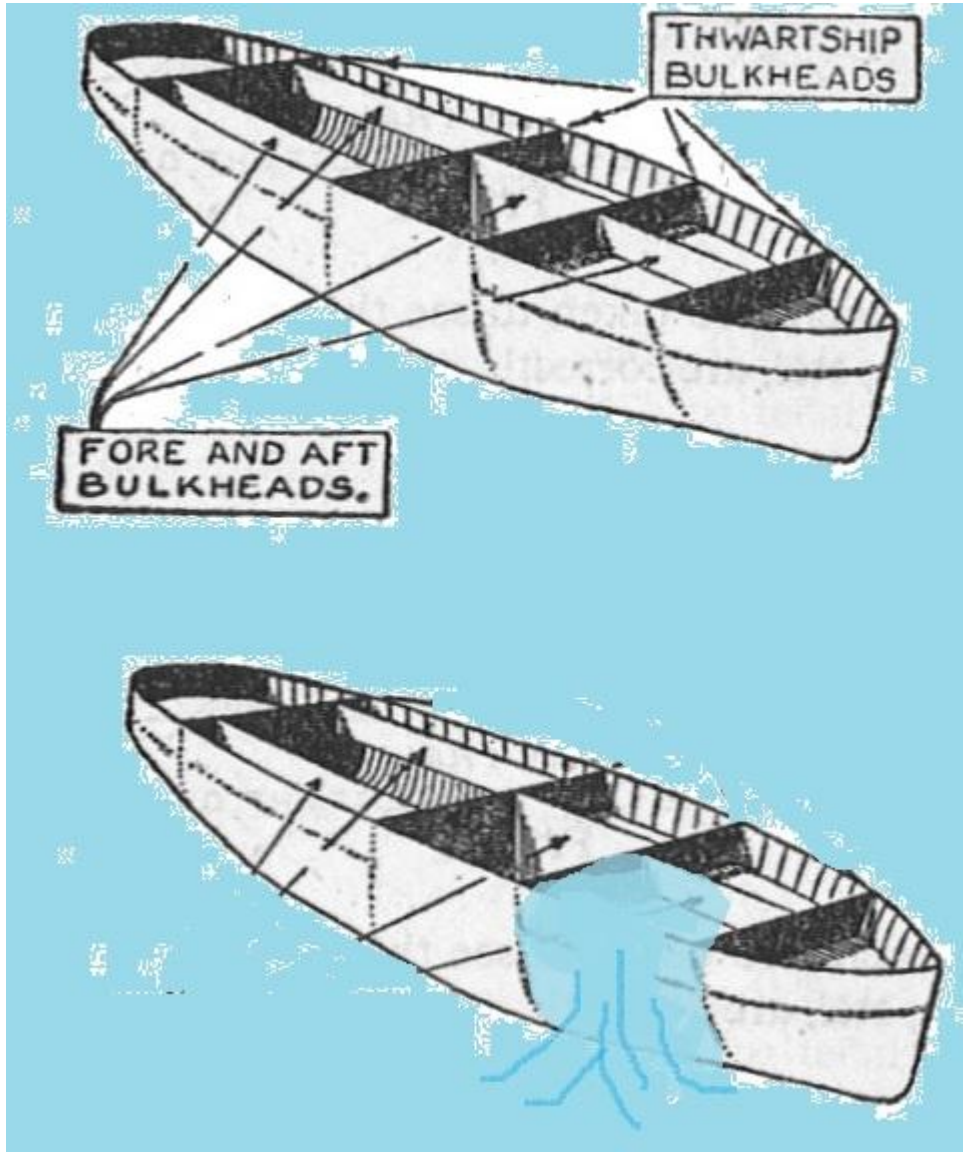
- Especifica uma ação alternativa a ser executada quando uma operação falha. Isso pode ser retornar valores padrão ou realizar uma ação diferente para garantir que o sistema continue a funcionar, mesmo que parcialmente.
- **Exemplo**: Se o serviço de recomendação de produtos falhar, o sistema pode exibir uma lista de produtos populares como fallback.
- **Benefícios**: Mantém a funcionalidade básica do sistema para o usuário, oferecendo uma experiência degradada, mas aceitável, em vez de uma interrupção completa.

Caching (Cache)

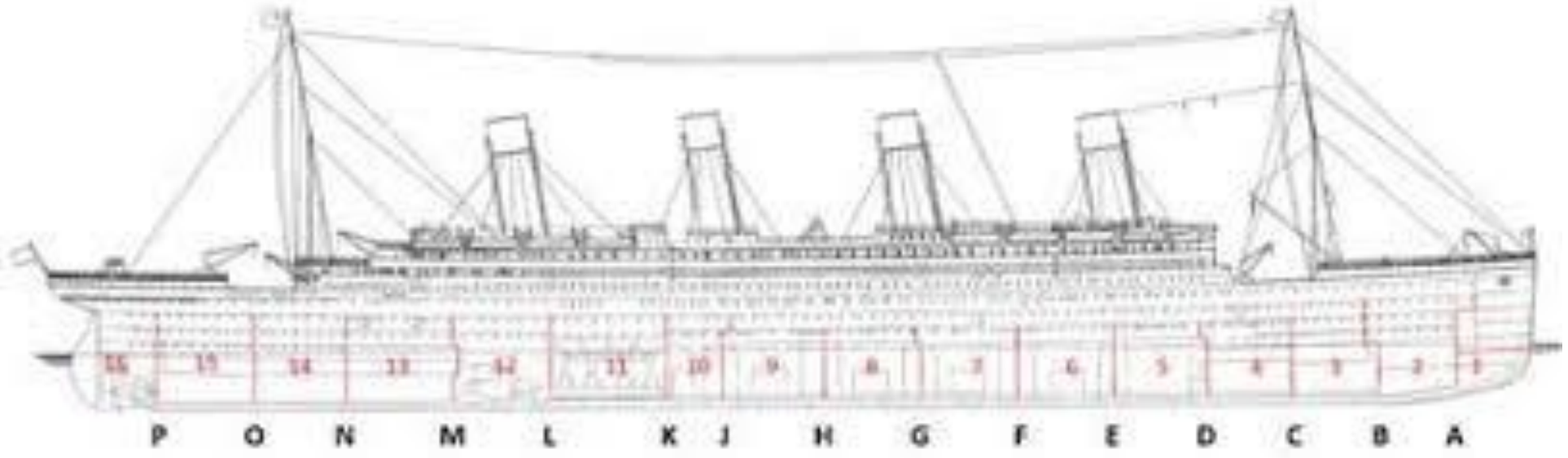
- Armazena temporariamente respostas ou dados, para evitar chamar repetidamente um serviço externo. Isso pode reduzir a carga sobre o sistema e melhorar a performance, especialmente em serviços frequentemente acessados.
- **Exemplo**: Um sistema de preços pode armazenar em cache os preços de produtos por alguns minutos, evitando consultas frequentes ao serviço de precificação.
- **Benefícios**: Melhora a performance e a resiliência do sistema, minimizando a necessidade de acessar recursos externos.

Bulkhead

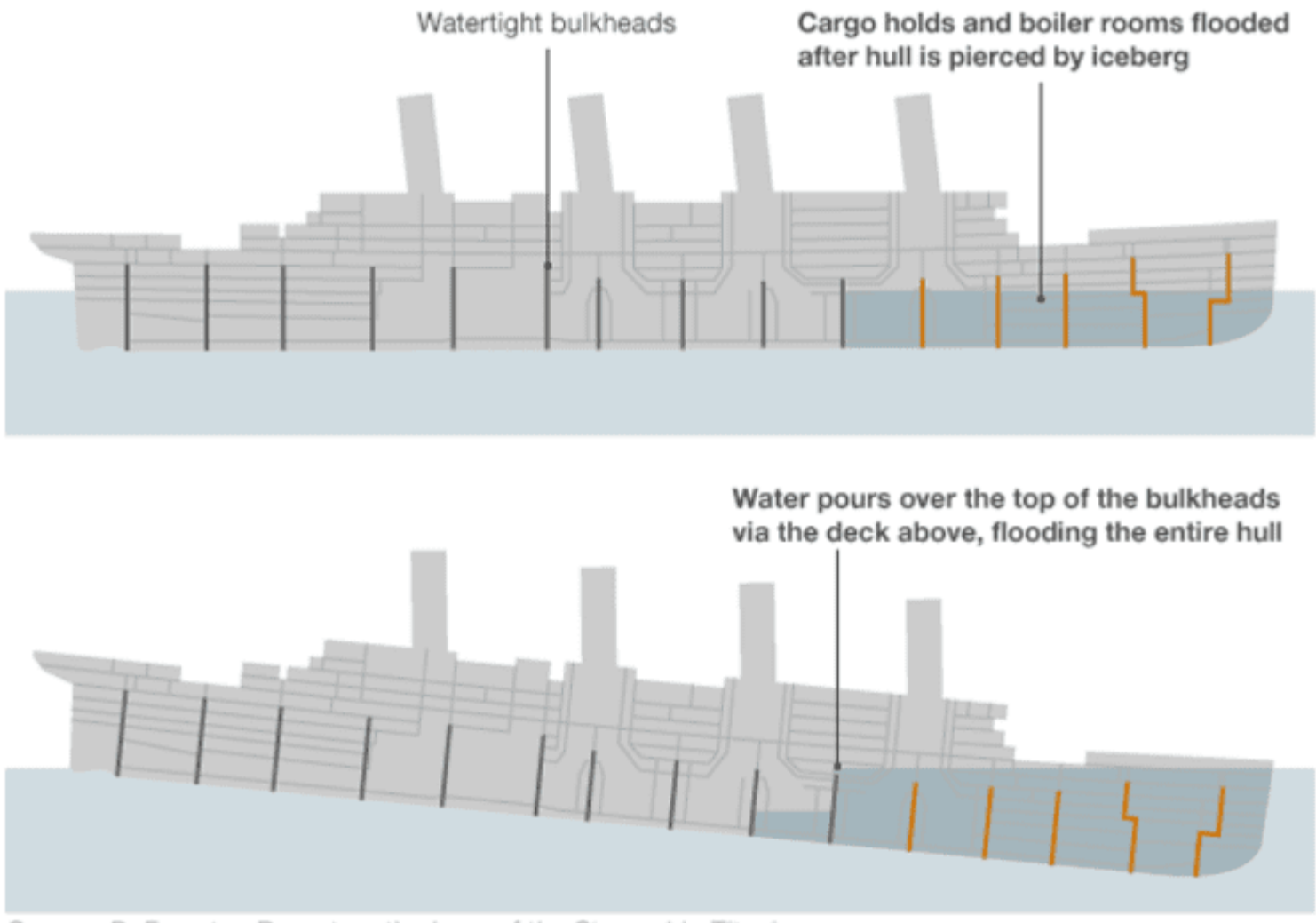
- O padrão **Bulkhead** (compartimento estanque) é uma técnica de **resiliência** usada para isolar falhas em sistemas distribuídos, como os microservices.
- Imagine um navio dividido em compartimentos estanques.
- Se um compartimento enche de água, os outros permanecem seguros, evitando que o navio afunde.



E o Titanic



RMS Titanic - key design fault



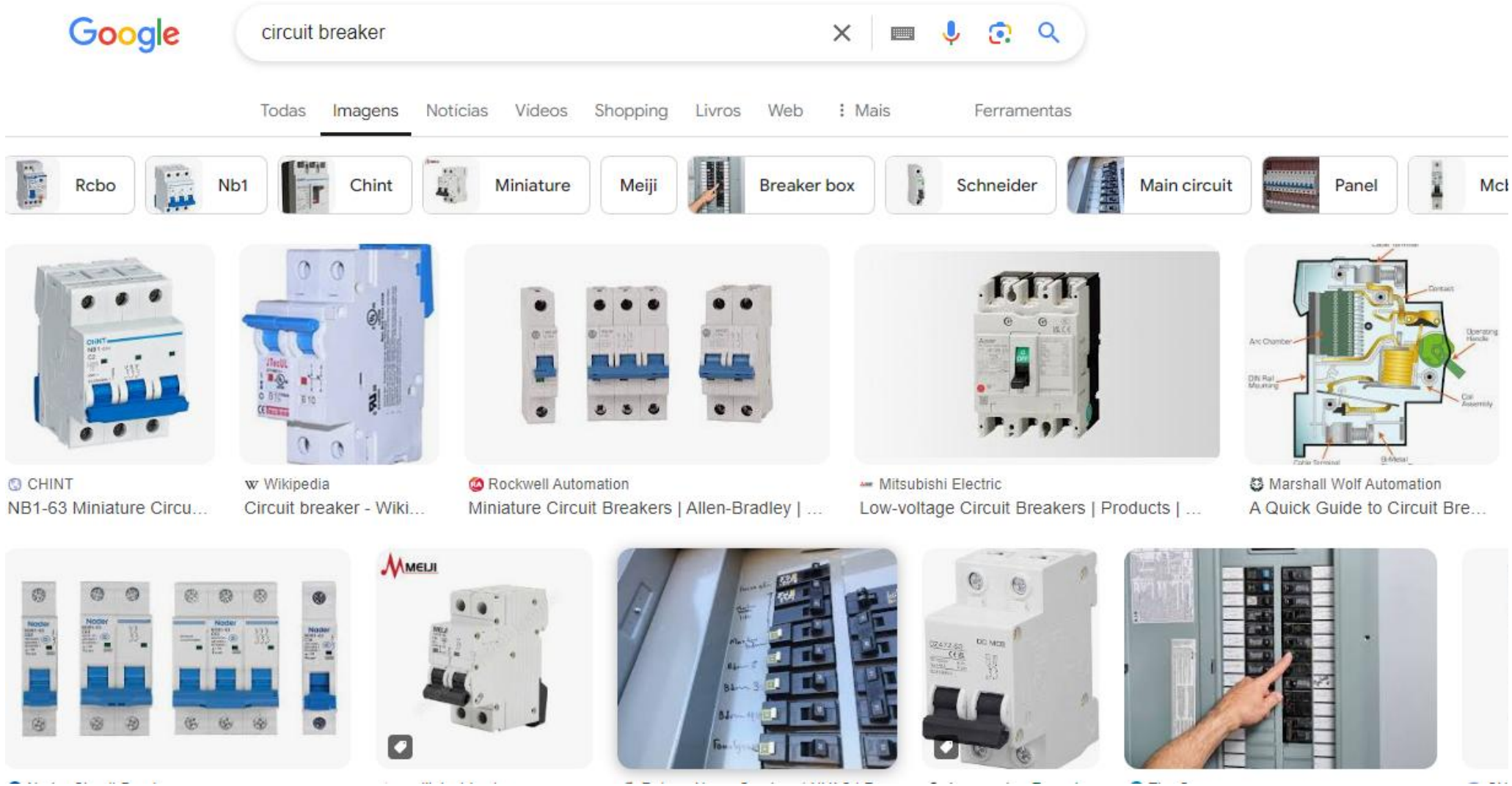
Bulkhead em Softwares

- **Isolamento de recursos:** Cada componente ou serviço crítico recebe uma quantidade limitada de recursos (como threads, conexões de banco de dados ou memória).
- **Contenção de falhas:** Se um componente falha e consome todos os seus recursos alocados, a falha é contida dentro desse compartimento, evitando que se propague para outros.
- **Proteção dos demais componentes:** Os demais componentes continuam funcionando normalmente, pois seus recursos não são afetados pela falha.

Implementação do Bulkhead

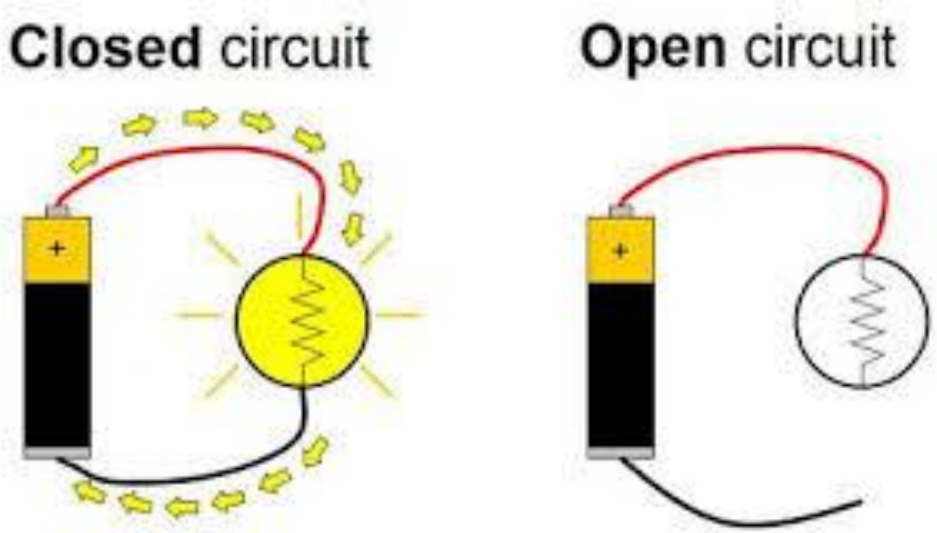
- **Threads:** Criar threads separadas para cada componente crítico, limitando o número de threads por componente.
- **Processos:** Isolar componentes em processos separados, utilizando mecanismos de comunicação entre processos (IPC).
- **Containers:** Utilizar contêineres (como Docker) para isolar componentes em ambientes separados.

Circuit Breaker



Circuit Breaker - Disjuntor

- Um disjuntor é um interruptor elétrico projetado para **proteger** um circuito elétrico de danos causados por falhas na alimentação elétrica, principalmente devido a situações de sobrecorrentes, causadas por exemplo por excesso de carga ou um curto-circuito.
- Quando há uma sobrecarga o Disjuntor **abre** o circuito, impendido assim a continuação da falha

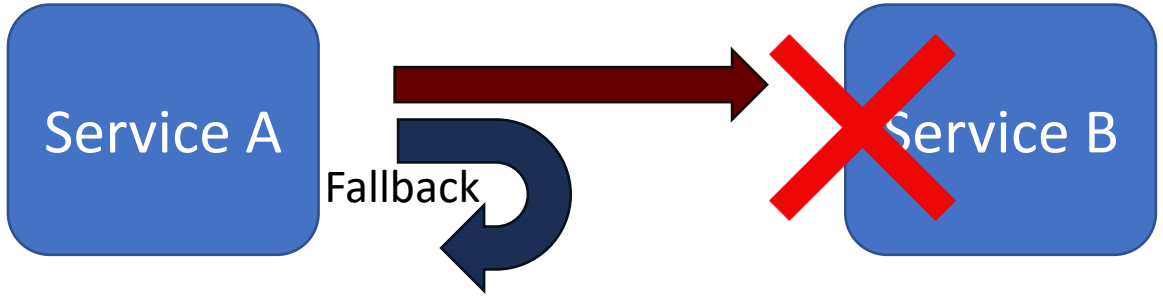


Circuit Breaker em Softwares

Funcionamento do Circuit Breaker

- Estado Fechado

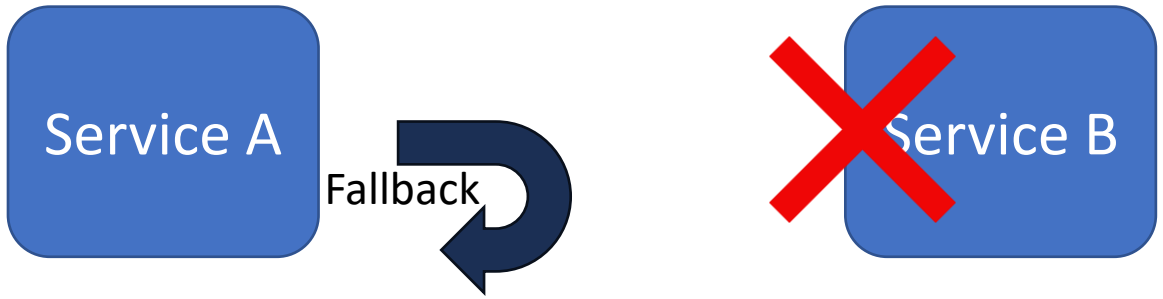
Circuit Breaker em Softwares



Funcionamento do Circuit Breaker

- Estado Fechado
- Falha:
  - Aciona a função fallback
  - Incrementa o contador de falhas

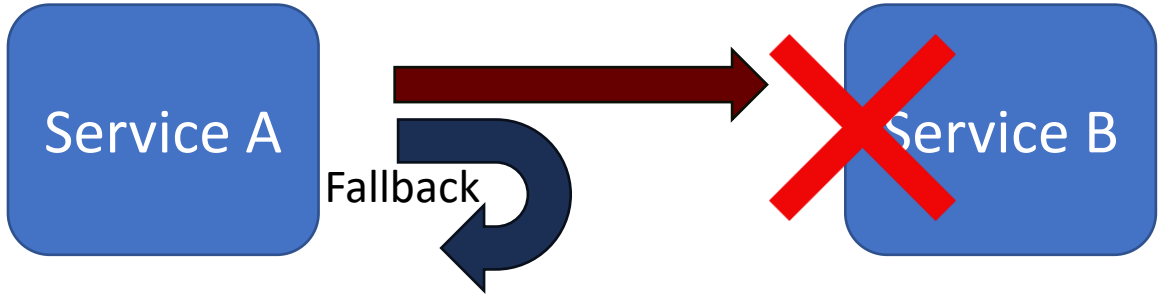
Circuit Breaker em Softwares



Funcionamento do Circuit Breaker

- Estado Fechado
- Falha:
  - Aciona a função fallback
  - Incrementa o contador de falhas
  - ATINGE O LIMITE DE FALHAS
- Estado Aberto (contador de tempo)

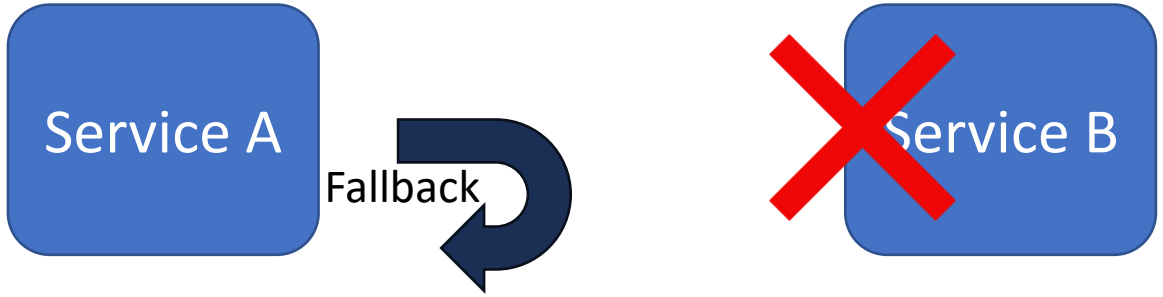
Circuit Breaker em Softwares



Funcionamento do Circuit Breaker

- Estado Fechado
- Falha:
  - Aciona a função fallback
  - Incrementa o contador de falhas
  - ATINGE O LIMITE DE FALHAS
- Estado Aberto (contador de tempo)
- Estado Meio-Aberto

Circuit Breaker em Softwares



Funcionamento do Circuit Breaker

- Estado Fechado
- Falha:
  - Aciona a função fallback
  - Incrementa o contador de falhas
  - ATINGE O LIMITE DE FALHAS
- Estado Aberto (contador de tempo)
- Estado Meio-Aberto
- Estado Aberto (contador de tempo)

Circuit Breaker em Softwares

Funcionamento do Circuit Breaker

- Estado Fechado
- Falha:
  - Aciona a função fallback
  - Incrementa o contador de falhas
  - ATINGE O LIMITE DE FALHAS
- Estado Aberto (contador de tempo)
- Estado Meio-Aberto
- Estado Aberto (contador de tempo)
- Estado Meio-Aberto

Circuit Breaker em Softwares

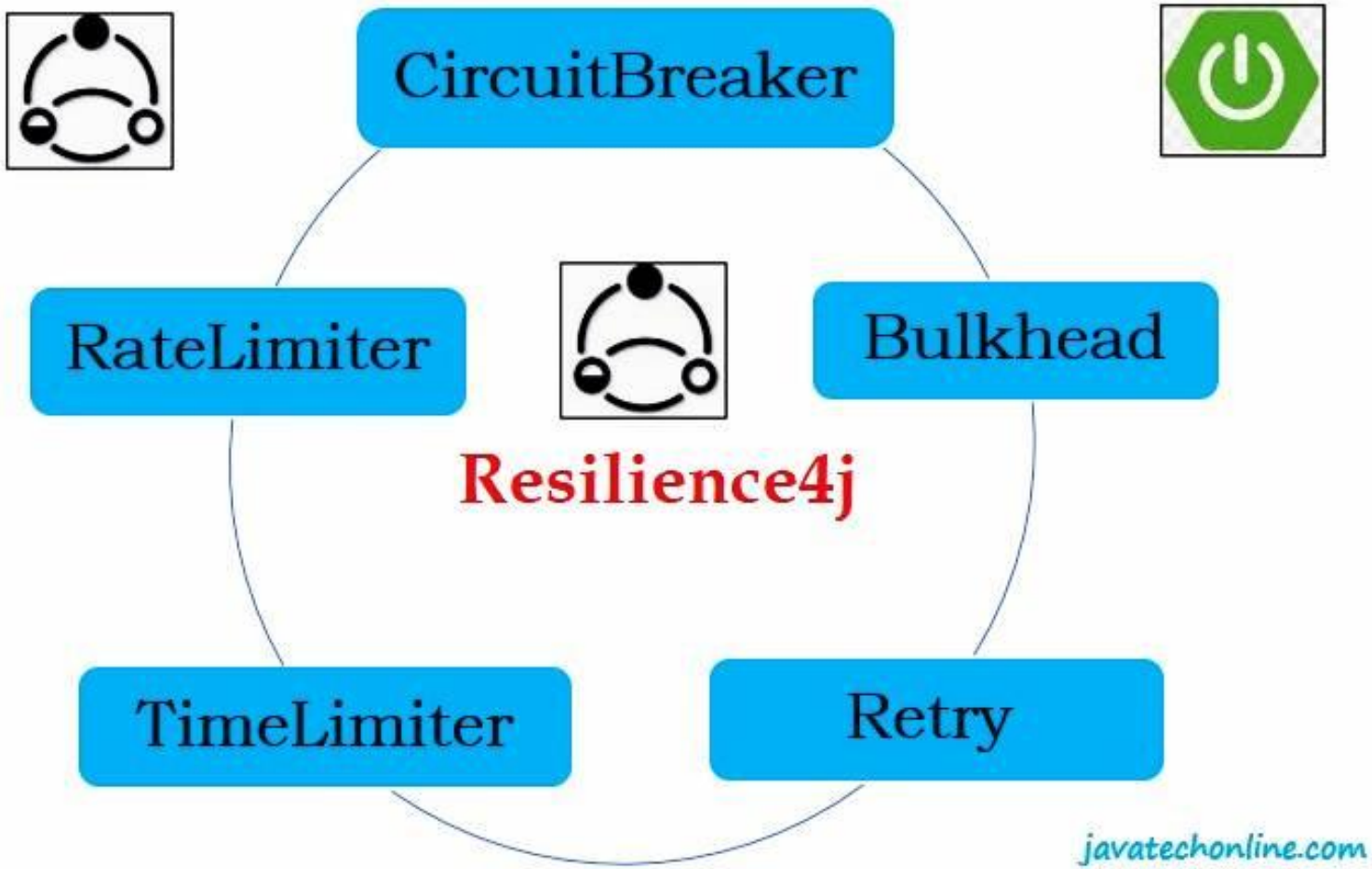
Funcionamento do Circuit Breaker

- Estado Fechado
- Falha:
  - Aciona a função fallback
  - Incrementa o contador de falhas
  - ATINGE O LIMITE DE FALHAS
- Estado Aberto (contador de tempo)
- Estado Meio-Aberto
- Estado Aberto (contador de tempo)
- Estado Meio-Aberto
- Estado Fechado

Resilience Patterns na Prática 💻

☕ ☕ ☕ Em Java é claro! 💪

Resilience4j in Spring Boot Microservices



Agora é com vocês!!!!

Ciência da Computação