DESIGN FOR FAILURE

PADRÕES DE RESILIÊNCIA

Luiz Schons

QUEM SOU EU?

- Luiz Schons
- Senior Software Architect/Engineer @ Vivaworks
- Canionista / Montanhista
- Atleta de CA
- BJJ
- DevParaná
- UTFPR
- schons.hashnode.dev
- Polyglot.ai & Golazzo
- Uma curiosidade…

DISCLAIMER

- Não sou especialista em nada
- Analise o cenário que você está
- Não existe bala de prata
- Estude e estude…

COMO ASSIM "DESIGN FOR FAILURE"?

Projetar sistemas assumindo que falhas vão acontecer,
não tentando evitá-las.

MAS PERAI, SISTEMAS FALHAM?

Sim, e por diversos motivos - software, hardware, rede ou fatores externos.

Devemos construir sistemas resilientes que continuem funcionando mesmo quando falhas ocorrem.

VAMOS COLOCAR ISSO NA PRÁTICA

A IDEIA REVOLUCIONÁRIA

Você tem uma ideia brilhante para um e-commerce que vai mudar o mundo.

O INÍCIO MODESTO

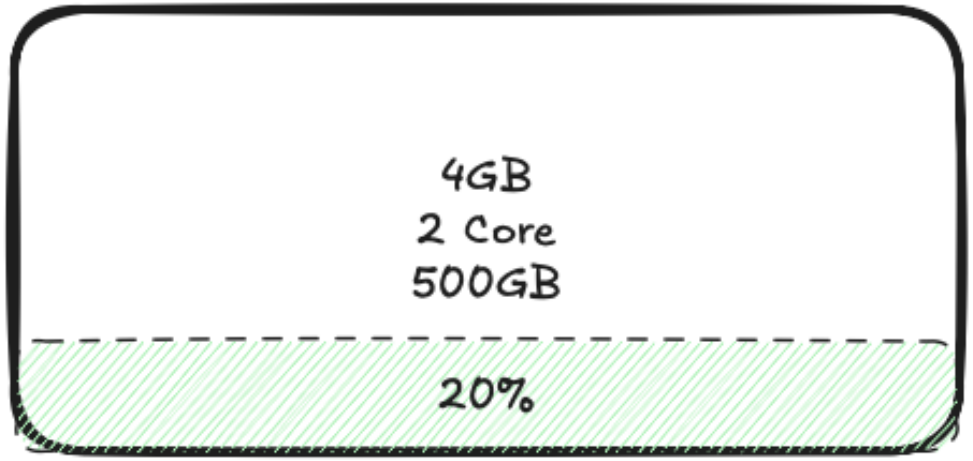
- Um computador antigo como servidor
- Instalação de servidor web e banco de dados
- Início do desenvolvimento do site

Quem nunca começou assim? 😅

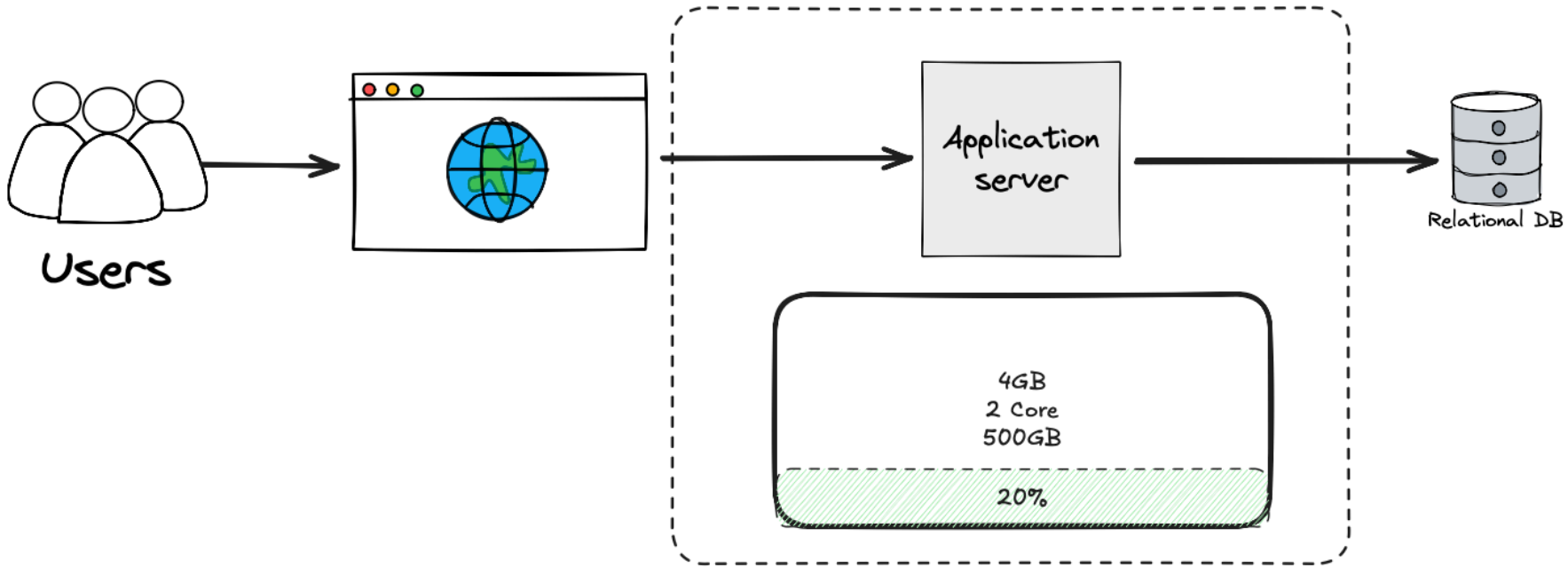
SEU SUPER SERVIDOR

- 4GB de RAM
- Processador dual-core
- HD de 500GB
- Sistema operacional + Servidor web + Banco de dados

Parece até datacenter, não é mesmo? 🤣

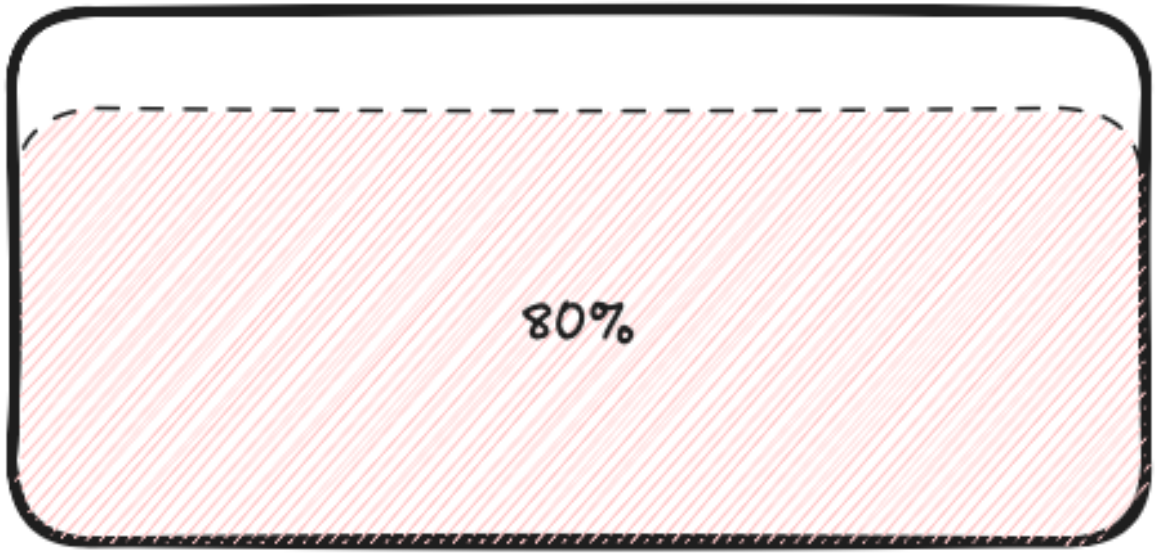


Você ainda tem poucos usuários, então tudo funciona bem.


Users
Application server
Relational DB
4GB
2 Core
500GB
20%

O SUCESSO TRAZ NOVOS DESAFIOS

Com o aumento de usuários, seu servidor local começa a mostrar sinais de sobrecarga.

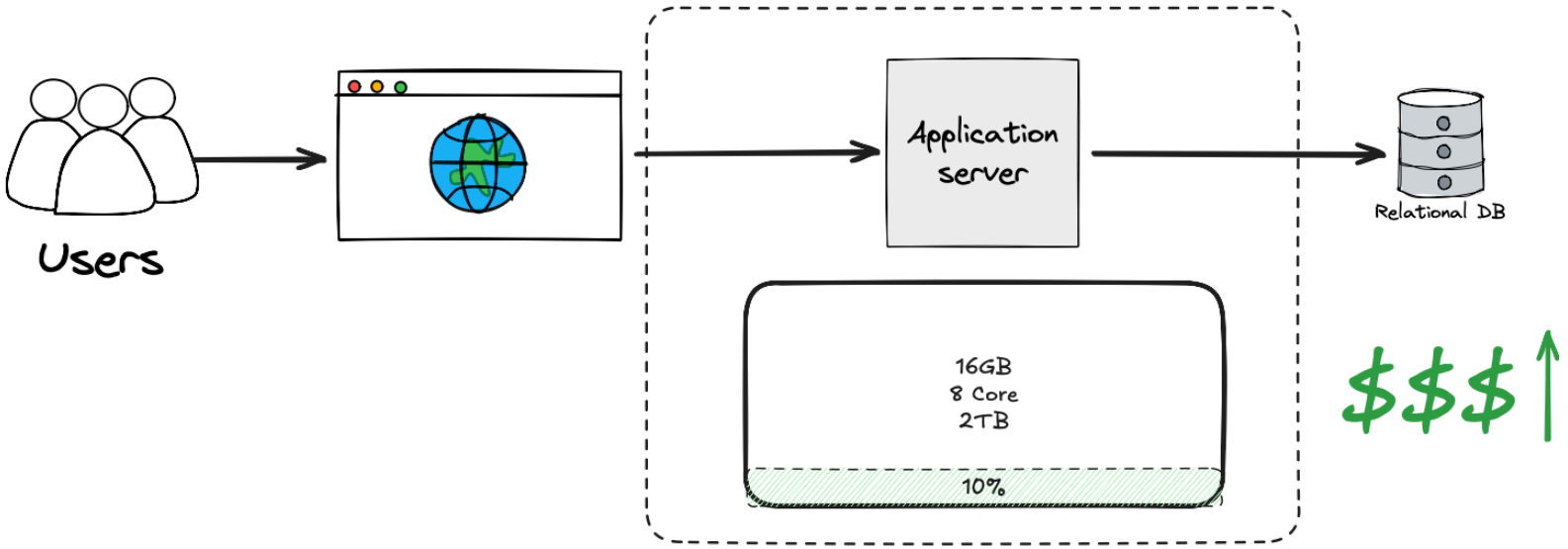


MEU DEUS, O QUE FAZER?

É hora de repensar a infraestrutura: escalar para suportar mais usuários e garantir a continuidade do serviço quando falhas ocorrerem.

A SOLUÇÃO: HARDWARE MAIS POTENTE

Investimento em um servidor dedicado com mais recursos computacionais.



PRONTO!

AGORA VOCÊ TEM UM SERVIDOR POTENTE E UMA CONTA PARA PAGAR.

A SOLUÇÃO FUNCIONA...

POR ENQUANTO

Mas você esqueceu de um "pequeno" detalhe:
servidores precisam de manutenção constante.

E você não tem uma equipe de infraestrutura...

SUPER-HERÓI DA INFRA

- Instalação de atualizações de segurança
- Monitoramento de recursos
- Resolução de problemas técnicos
- Manutenção do banco de dados

Ah, e no tempo livre... desenvolver o seu e-commerce! 😰

OS PESADELOS TE ASSOMBRAM

- E se o servidor parar de funcionar?
- E se ocorrer uma queda de energia?
- E se o disco rígido falhar?

Meu negócio pode suportar esse risco?

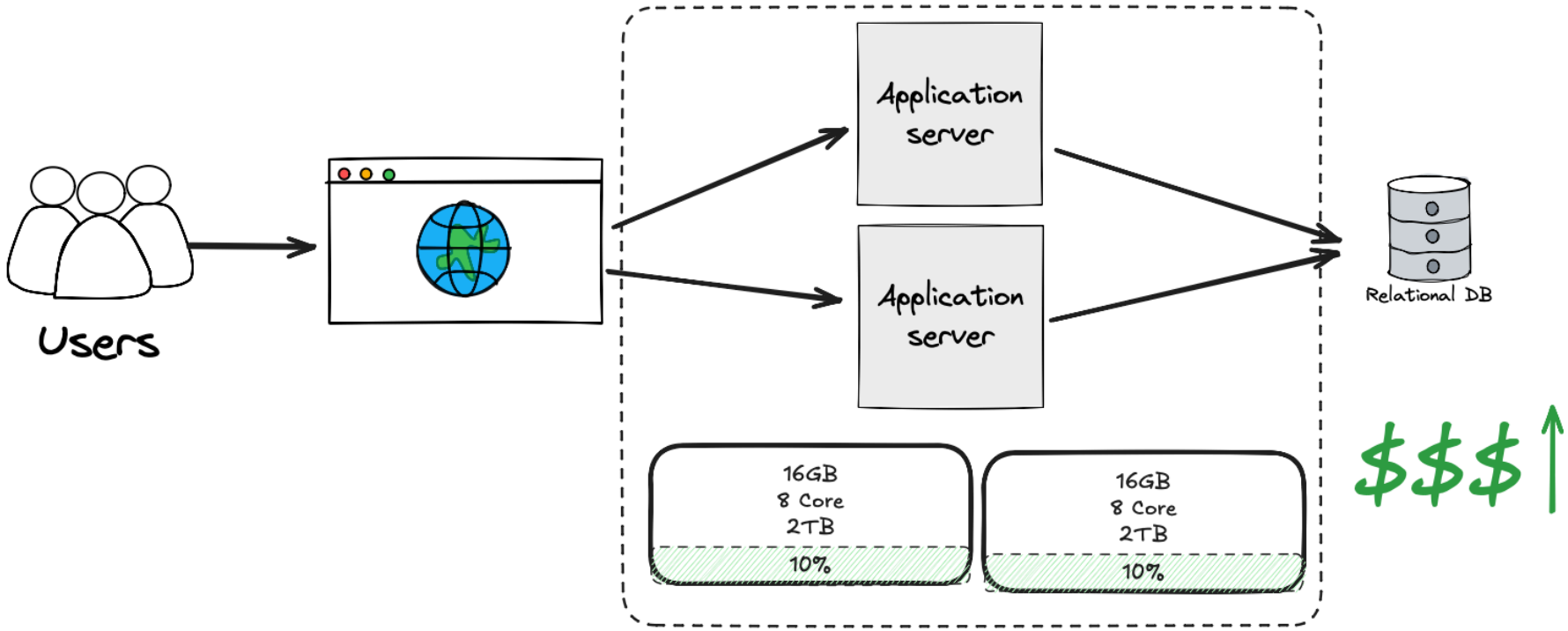
E AGORA?

Um único servidor não é suficiente...

Precisamos de uma estratégia melhor!

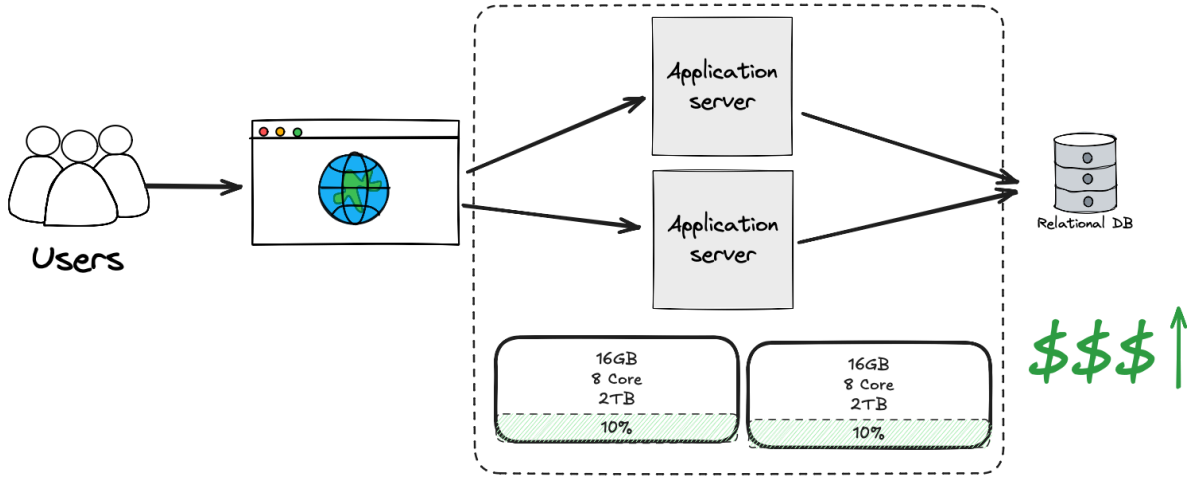
HORA DE ESCALAR COM ALTA DISPONIBILIDADE

O próximo passo: múltiplos servidores para eliminar pontos únicos de falha e garantir que seu e-commerce esteja sempre disponível.


Users
Application server
Application server
Relational DB
16GB
8 Core
2TB
10%
16GB
8 Core
2TB
10%
$$$

ARQUITETURA DE ALTA DISPONIBILIDADE

Infraestrutura robusta com servidor principal, backup e banco de dados separado — tudo configurado para continuar funcionando mesmo quando um componente falha.



SEU DATACENTER CASEIRO TEM UM PROBLEMA

- Falta de monitoramento 24/7
- Vulnerabilidade a desastres físicos (incêndio, inundação)
- Riscos de segurança (roubo, invasão)

Sua sala não foi projetada para ser um datacenter profissional...

O CUSTO OCULTO DA "ECONOMIA"

- Investimento crescente em hardware
- Horas intermináveis de manutenção
- Tempo precioso desviado do seu negócio principal
- Estresse constante com a infraestrutura

O barato sai caro... muito caro!

PRECISAMOS DE ALGO MELHOR

Uma solução que:

- Escale automaticamente conforme a demanda
- Ofereça alta disponibilidade sem dores de cabeça
- Permita foco no desenvolvimento do seu negócio

E QUAL SERIA A SOLUÇÃO?

Alguém tem alguma ideia?

Quais opções temos hoje para:

- Escalar facilmente
- Reduzir manutenção
- Aumentar disponibilidade
- Focar no negócio

A RESPOSTA:

CLOUD COMPUTING

Migração da infraestrutura para provedores especializados em nuvem que oferecem:

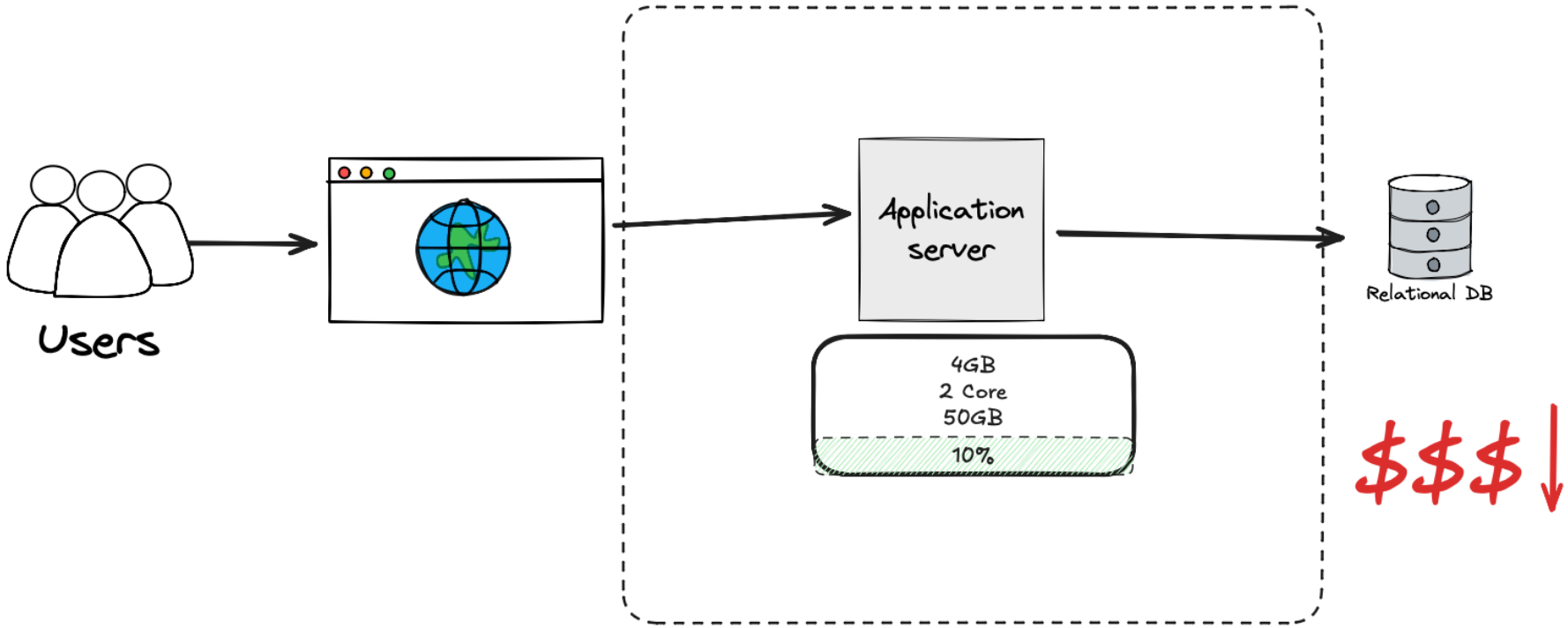
- Infraestrutura como serviço
- Escalabilidade sob demanda
- Alta disponibilidade com garantias por contrato
- Redução de responsabilidades operacionais

DOCKER

- Encapsula aplicações
- Portabilidade total
- Deploy simplificado
- Infraestrutura como código

CURSO GRATUITO DE DOCKER

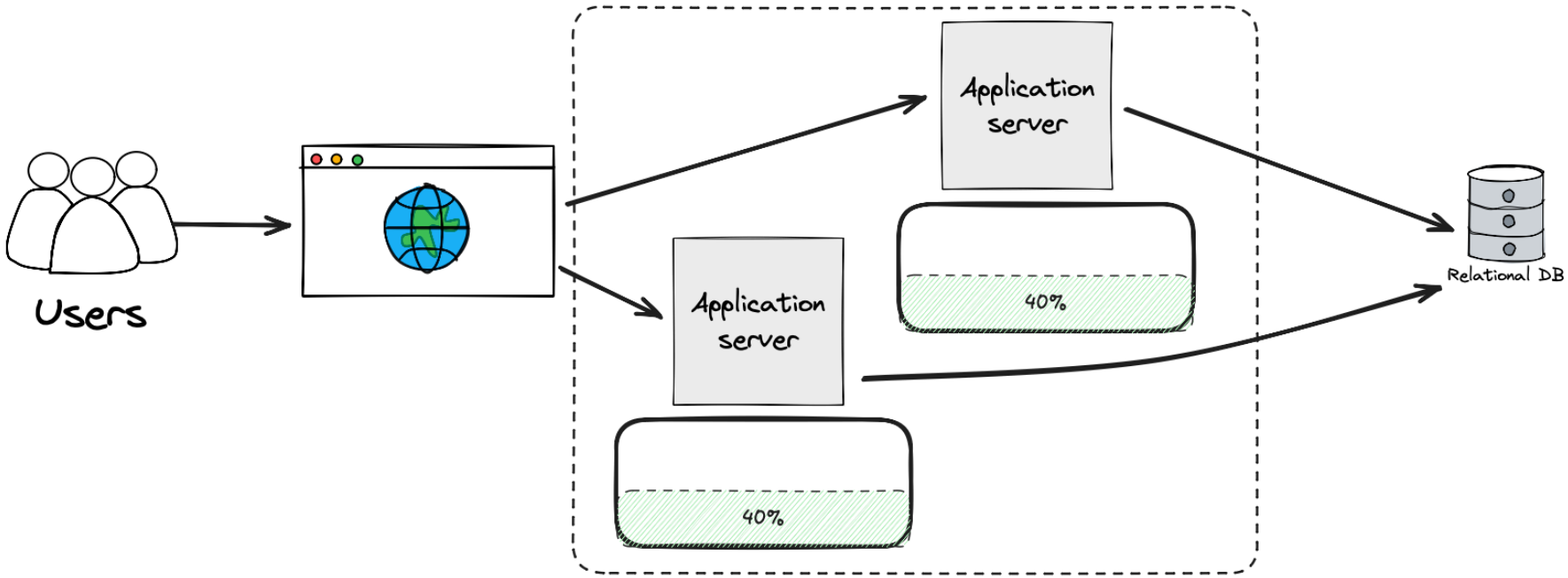
Fernanda Kipper no YouTube


Users
Application server
4GB
2 Core
50GB
10%
Relational DB
$$$↓

CLOUD: BENEFÍCIOS PRÁTICOS

- Menos manutenção
- Escala automática
- Serviços gerenciados
- Backups automáticos

Foco no negócio, não na infraestrutura


Users
Application server
Application server
40%
40%
Relational DB

KUBERNETES

- Orquestra containers
- Gerencia réplicas automaticamente
- Garante disponibilidade dos serviços

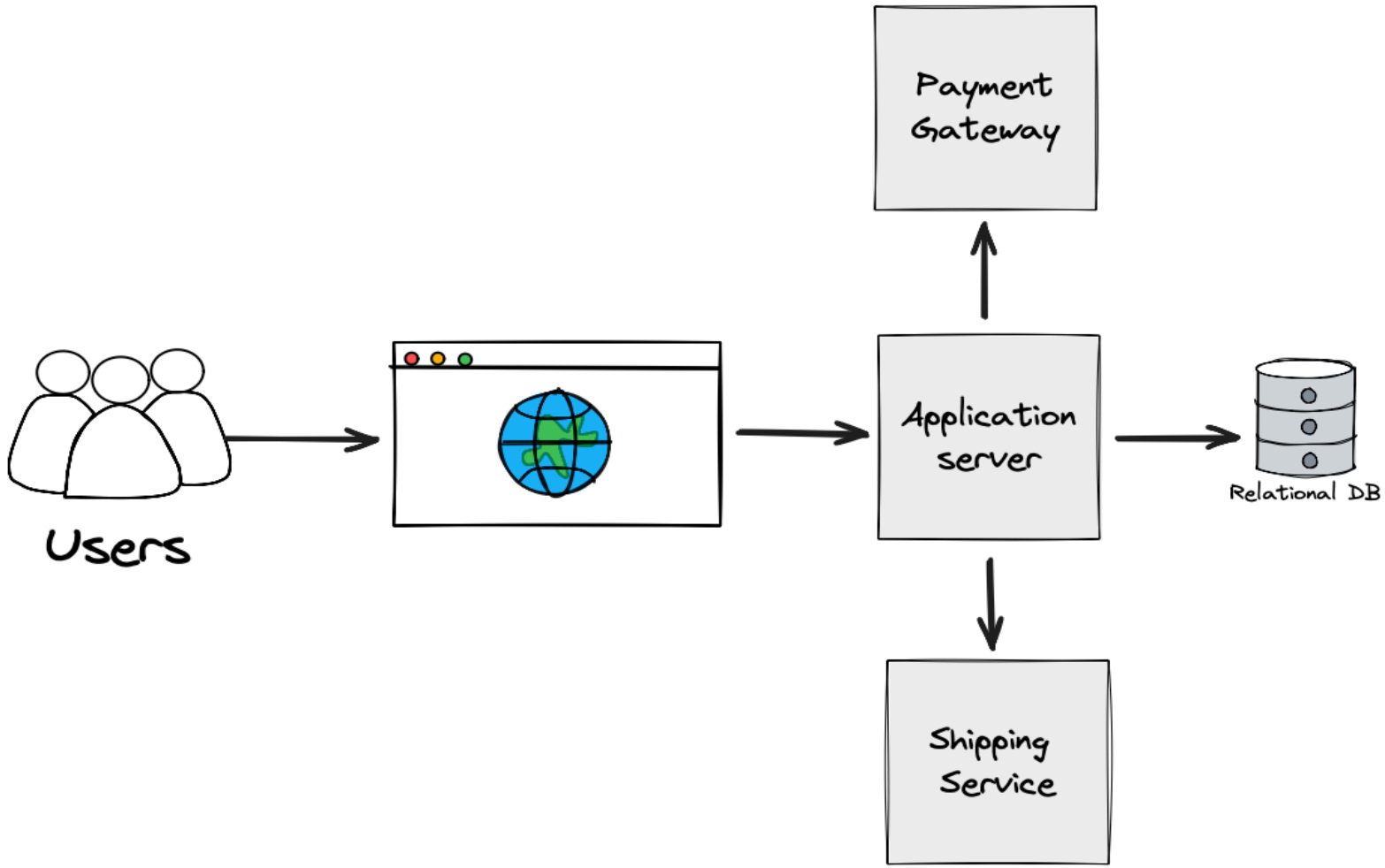
O caminho para a resiliência está apenas começando...

NÃO ESTAMOS AQUI PARA FALAR DE INFRAESTRUTURA

Mas sim de como projetar sistemas que sobrevivam quando as coisas dão errado

Porque, acredite, elas sempre dão! 😅

VOLTANDO AO NOSSO E-COMMERCE


Payment Gateway
Users
Application server
Relational DB
Shipping Service

QUAIS PROBLEMAS PODEM OCORRER NESTE FLUXO?

Pense em todos os pontos de falha possíveis...

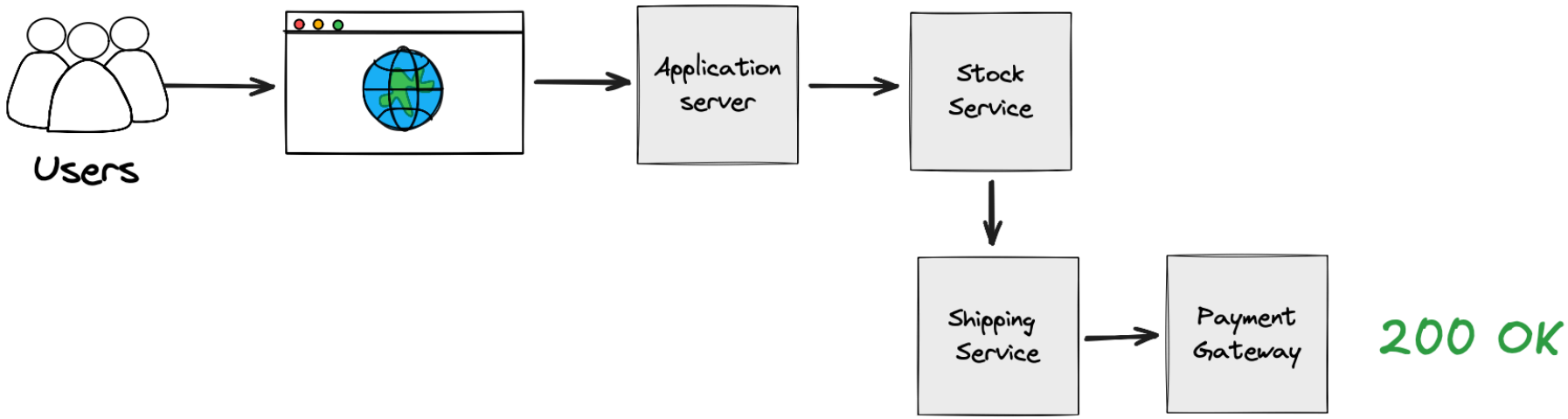
PROBLEMAS QUE PODEM OCORRER

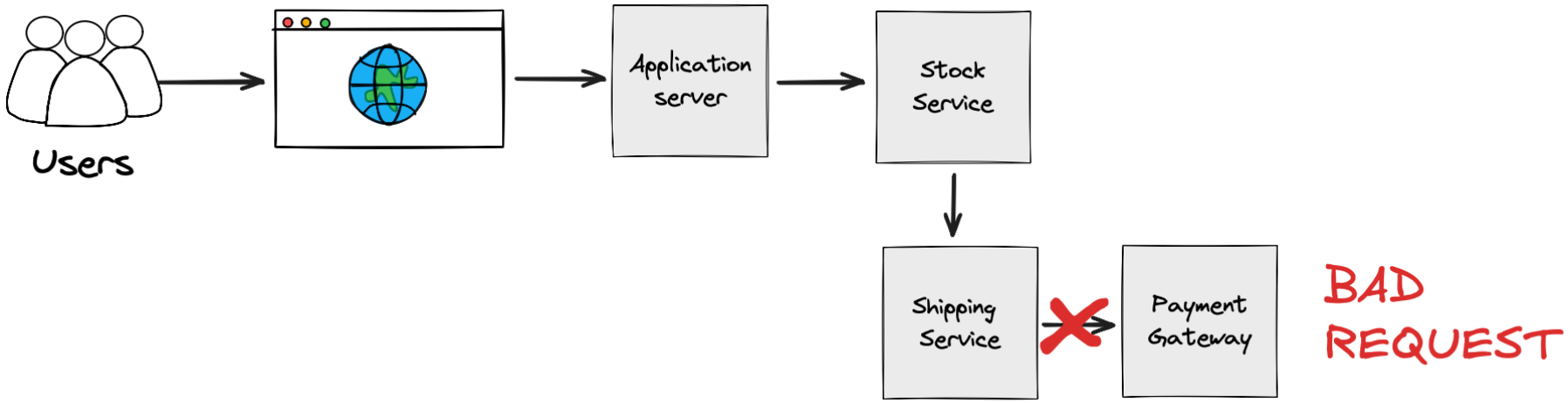
- Serviço de pagamento fora do ar
- Serviço de calcular frete fora do ar
- Ataque em algum endpoint do sistema
- Complexidade ao adicionar mais um serviço

VAMOS ENTENDER ESSES PROBLEMAS MELHOR

E como podemos projetar nossas aplicações para lidar com eles

SERVIÇO DE PAGAMENTO FORA DO AR

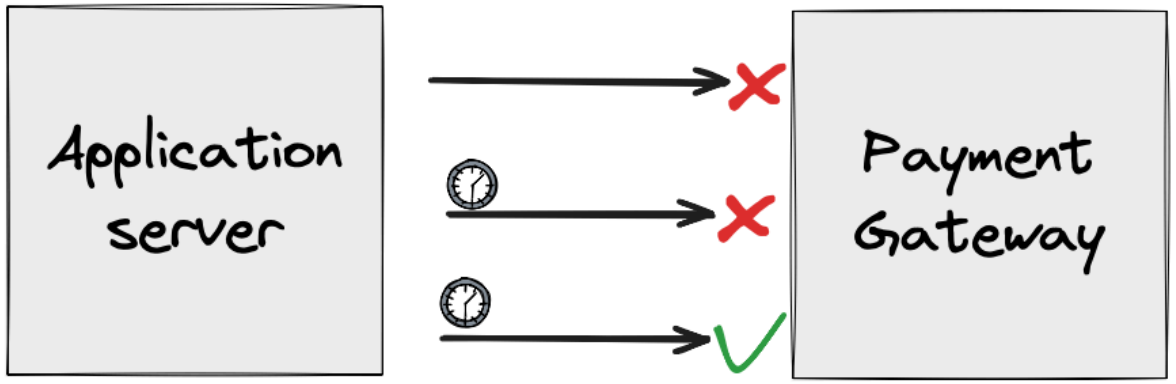

Users
Application server
Stock Service
Shipping Service
Payment Gateway
200 OK


Users
Application server
Stock Service
Shipping Service
Payment Gateway
BAD REQUEST

COMO PODEMOS LIDAR COM ISSO?

RETRY

Tente novamente após uma falha, aumentando gradualmente o tempo de espera entre tentativas:



MAS COMO QUE FAZEMOS ISSO COM CÓDIGO?

CONCEITOS UNIVERSAIS, DIFERENTES LINGUAGENS

Os padrões de resiliência funcionam em qualquer linguagem de programação

Vamos ver exemplos em duas plataformas diferentes:

PHP COM HYPERF

TYPESCRIPT COM NESTJS

Mesmo problema, mesmos conceitos, diferentes implementações

EXEMPLO COM PHP E HYPERF

```php
namespace App\Service;

use Hyperf\Retry\Annotation\Retry;
use Hyperf\Retry\RetryBudget;

class PaymentService
{
    #[Retry(delay: 1000, maxAttempts: 3)]
    public function processPayment()
    {
        // make a remote call
```

EXEMPLO COM TYPESCRIPT E NESTJS

```
1  import { Injectable } from '@nestjs/common';
2  import { Retry } from '@nestjs/axios';
3  import { catchError, delay, retryWhen } from 'rx
4  import { of } from 'rxjs';
5
6  @Injectable()
7  export class PaymentService {
8      @Retry({ delay: 1000, maxAttempts: 3 })
9      async processPayment() {
10         // make a remote call
11     }
```

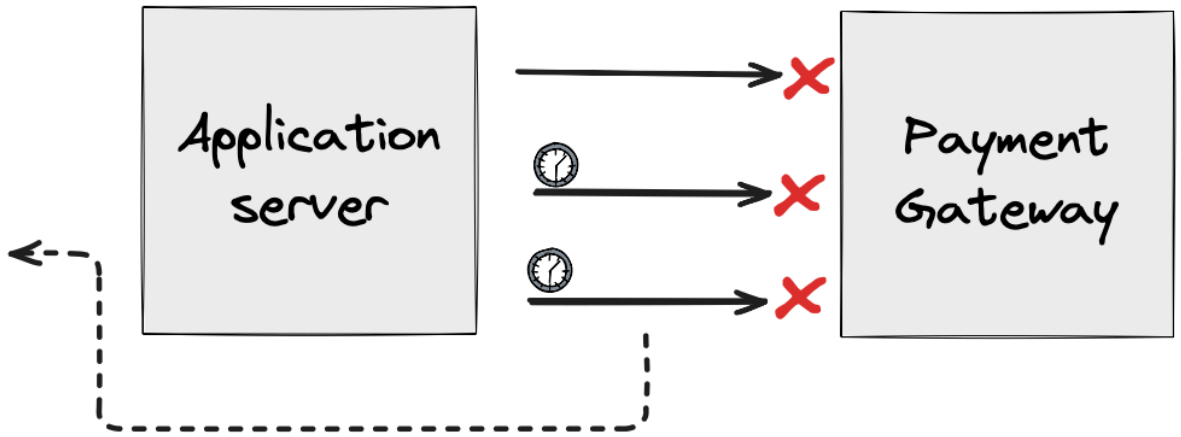
MAS O RETRY TEM UM PROBLEMA...

E se o serviço estiver completamente fora do ar?

Vamos ficar tentando sem parar?

CIRCUIT BREAKER

Interrompe chamadas a serviços com falhas, prevenindo sobrecarga do sistema



COMO FUNCIONA?

Inspirado nos disjuntores elétricos:

- **Fechado (normal)**: Permite as chamadas ao serviço
- **Aberto (falha)**: Bloqueia chamadas após muitos erros
- **Semi-aberto**: Permite algumas chamadas para testar recuperação

RETRY VS CIRCUIT BREAKER

RETRY

- Tenta novamente após falha
- Bom para falhas temporárias
- Pode sobrecarregar sistemas

CIRCUIT BREAKER

- Evita chamadas a serviços falhos
- Bom para falhas persistentes
- Permite recuperação gradual

Funcionam melhor quando usados em conjunto!

IMPLEMENTAÇÃO EM PHP COM HYPERF

```php
namespace App\Services;

use App\Gateway\GatewayServiceClient;
use Hyperf\CircuitBreaker\Annotation\CircuitBrea
use Hyperf\Di\Annotation\Inject;

class GatewayService
{
    #[Inject]
    private GatewayServiceClient $client;

```

IMPLEMENTAÇÃO EM TYPESCRIPT COM NESTJS

```typescript
import { Injectable } from '@nestjs/common';
import { CircuitBreaker } from '@nestjs/axios';

@Injectable()
export class PaymentService {
    @CircuitBreaker({
        timeout: 5000,
        fallback: () => 'Service unavailable'
    })
    async processPayment() {
        // make a remote call
```

VOCÊS PERCEBERAM ALGUM PADRÃO?

Algo comum nas duas implementações...

FALLBACKS

O QUE SÃO FALLBACKS?

- Planos B para quando algo falha
- Podem ser simples mensagens ou lógicas alternativas complexas

É como levar um guarda-chuva mesmo quando o céu está limpo 

FALLBACK EM PHP

```php
namespace App\Services;

use App\Gateway\GatewayServiceClient;
use Hyperf\CircuitBreaker\Annotation\CircuitBrea
use Hyperf\Di\Annotation\Inject;

class GatewayService
{
    #[Inject]
    private GatewayServiceClient $client;

```

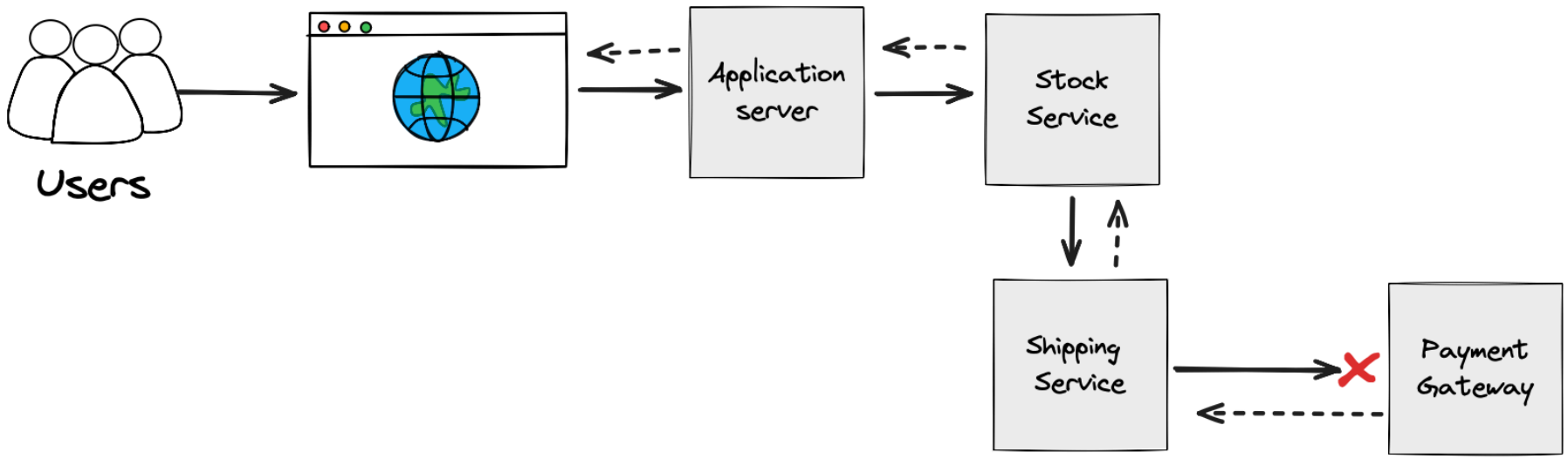
FALLBACK EM TYPESCRIPT

```
3
4  @Injectable()
5  export class PaymentService {
6      @CircuitBreaker({
7          timeout: 5000,
8          fallback: () => 'Service unavailable'
9      })
10     async processPayment() {
11         // make a remote call
12     }
13 }
```

E QUANDO TEMOS MÚLTIPLOS SERVIÇOS?

SAGA PATTERN

Gerencia transações distribuídas em sistemas complexos


Users
Application server
Stock Service
Shipping Service
Payment Gateway

SAGA PATTERN + FALLBACKS

Transações robustas com recuperação em cada etapa

FALLBACKS

- Plano B
- Resposta imediata

SAGA

- Transações distribuídas
- Compensação de erros

IMPLEMENTAÇÃO DE COMPENSAÇÕES

Quando algo falha, precisamos desfazer as operações anteriores

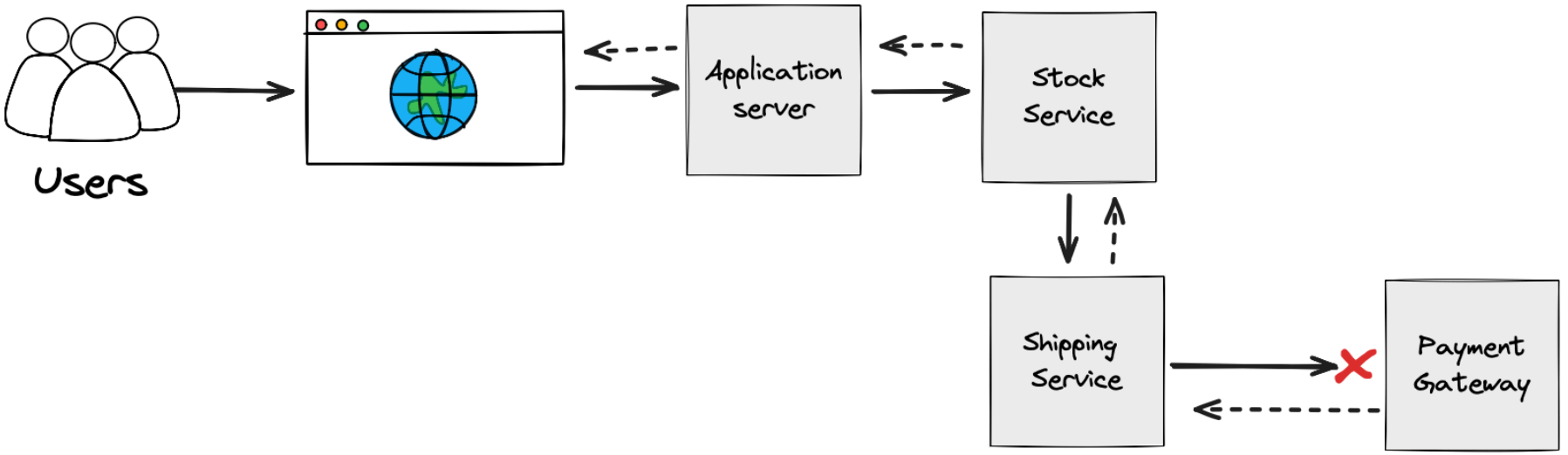
COMPENSAÇÃO EM PHP COM HYPERF

```php
<!--?php
namespace App\Listener;

use Hyperf\Event\Contract\ListenerInterface;
use App\Event\OrderCancelled;

class OrderCancelledListener implements Listene
{
    public function listen(): array
    {
        return [OrderCancelled::class];
```

COMPENSAÇÃO EM NESTJS COM RABBITMQ

```
import { RabbitSubscribe } from '@golevelup/nest
import { Injectable } from '@nestjs/common';
import { PaymentService } from '../services/payment
import { InventoryService } from '../services/inv
import { NotificationService } from '../services
import { LoggingService } from '../services/logg

@Injectable()
export class OrderCancelledListener {
  constructor(
    private paymentService: PaymentService,
```

VEJA COMO O PADRÃO SAGA FUNCIONA



Cada ação tem sua compensação correspondente

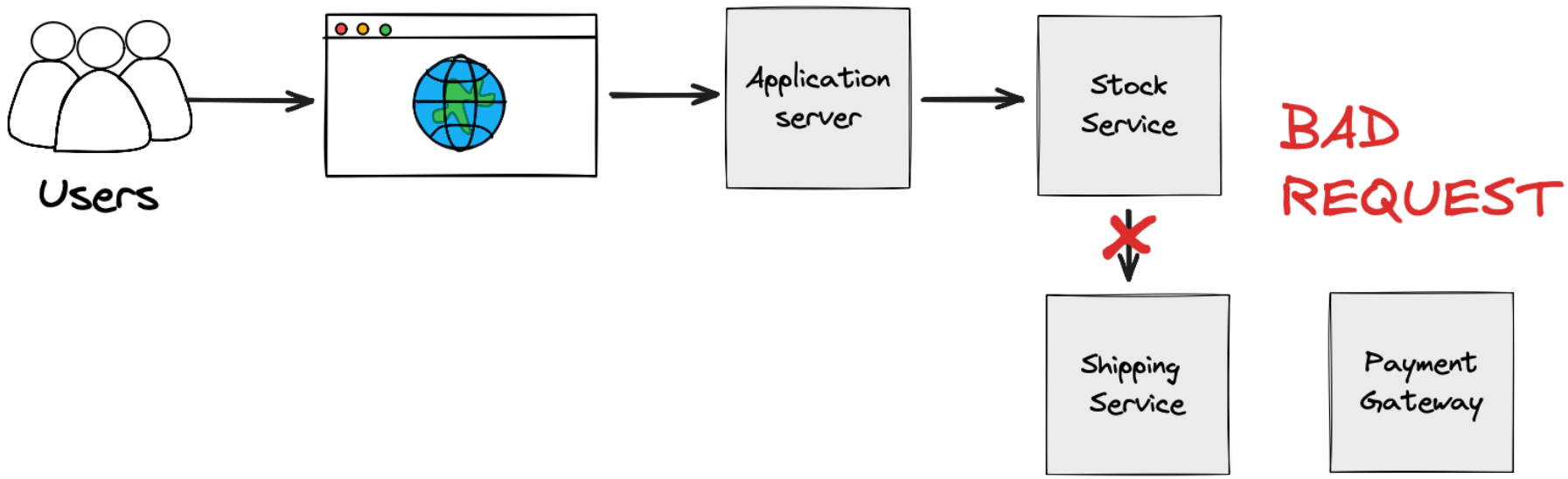
OUTRO PROBLEMA COMUM:

API DE CÁLCULO DE FRETE INDISPONÍVEL

CENÁRIO:

- Cliente está finalizando uma compra
- API de frete dos Correios está fora do ar
- Se não calcularmos o frete, perdemos a venda

Como lidar com esse tipo de problema?


Users
Application server
Stock Service
BAD REQUEST
Shipping Service
Payment Gateway

COMO PODEMOS RESOLVER ISSO?

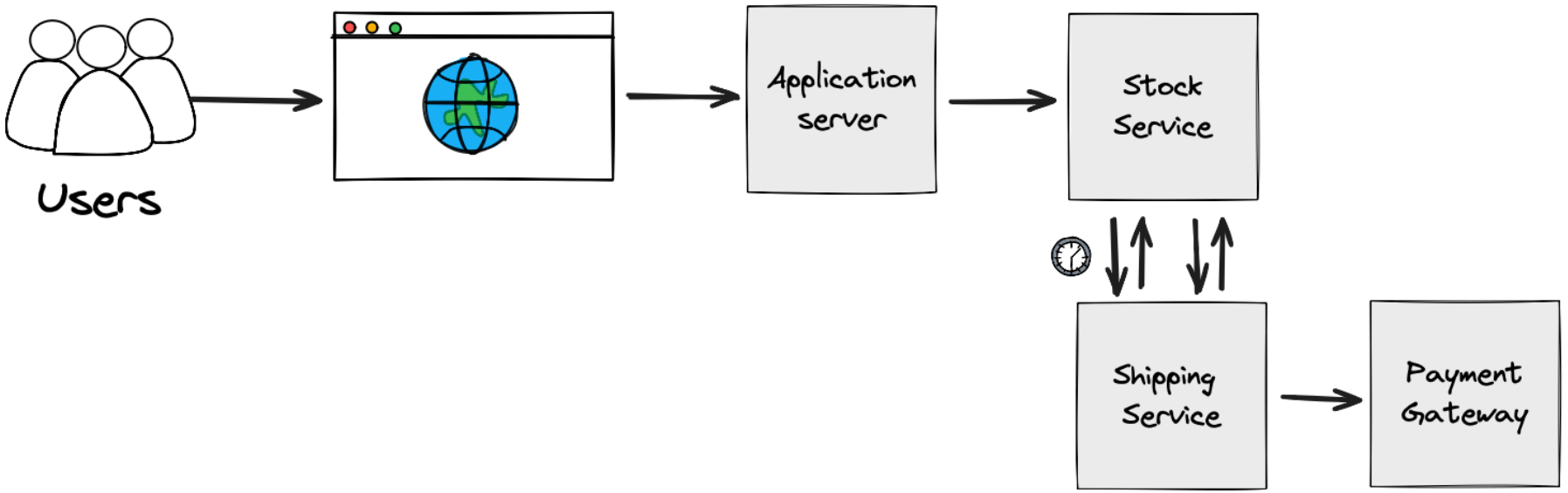
CACHE + VALORES DEFAULT

Uma estratégia para lidar com serviços externos indisponíveis

IMPLEMENTAÇÃO DE CACHE

```php
namespace App\Services;

use App\Shipping\ShippingServiceClient;
use Hyperf\CircuitBreaker\Annotation\CircuitBrea
use Hyperf\Di\Annotation\Inject;

class GatewayService
{
    #[Inject]
    private ShippingServiceClient $client;

```

RESUMINDO O QUE APRENDEMOS



Um sistema resiliente combina múltiplas estratégias

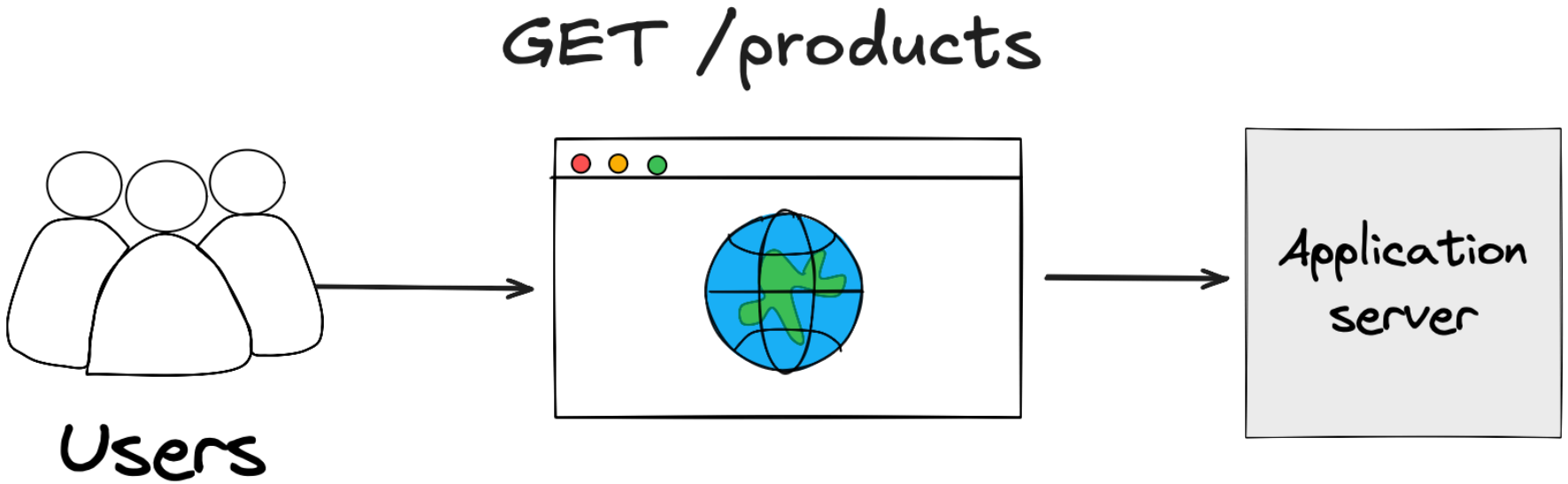
OUTRO PROBLEMA COMUM:

PROBLEMA: ATAQUES OU TRÁFEGO EXCESSIVO

ENDPOINT DE LISTAGEM DE PRODUTOS SOBRECARREGADO

CENÁRIO NORMAL: POUCOS USUÁRIOS

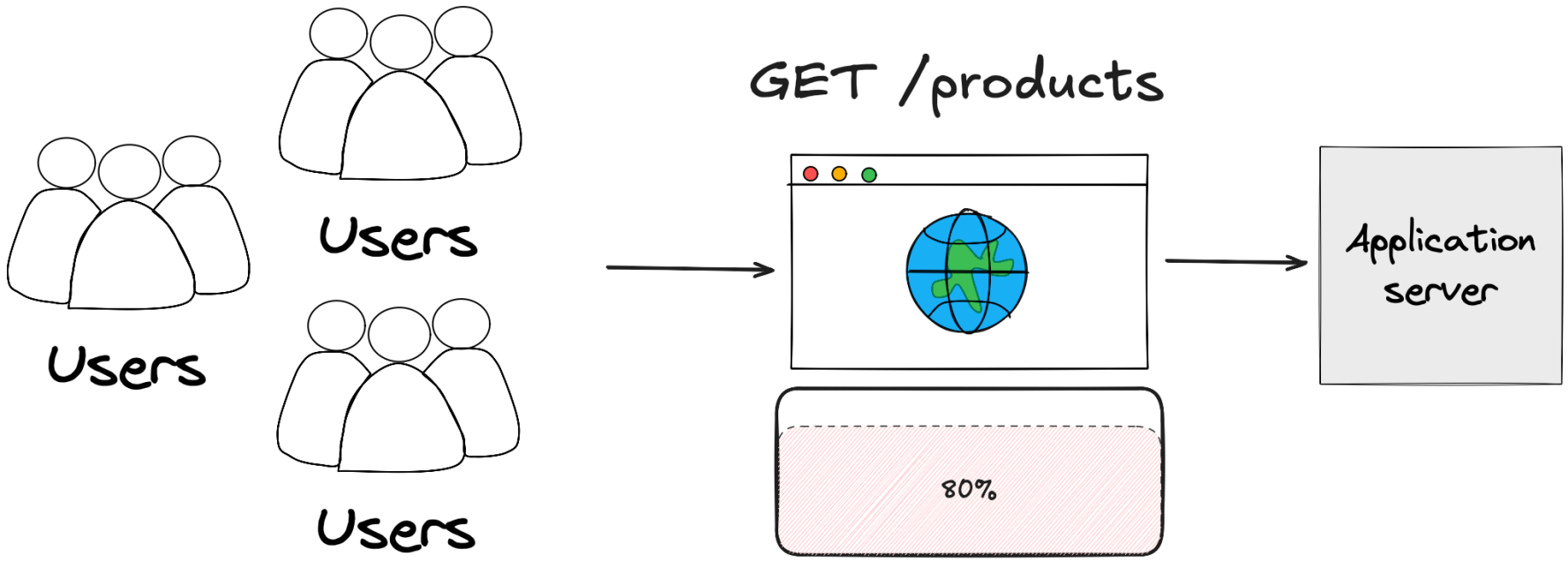
Endpoint GET /products responde rapidamente para cada usuário



Recursos do servidor são suficientes para atender a todos

CENÁRIO PROBLEMÁTICO: MUITOS USUÁRIOS OU ATAQUE

- Bots fazendo scraping da sua loja
- Ataque de negação de serviço (DDoS)
- Pico de tráfego em promoções



O servidor fica sobrecarregado e ninguém consegue usar o sistema

SOLUÇÃO: RATE LIMITING

Limita o número de requisições que um cliente pode fazer em um período de tempo

COMO FUNCIONA:

- Define limites por IP, usuário ou token
- Conta requisições em janelas de tempo
- Rejeita requisições quando o limite é atingido
- Retorna erro 429 (Too Many Requests)
- Retorna conteúdo cacheado para requisições repetidas

IMPLEMENTAÇÃO EM NESTJS

```
1  import { Injectable, NestMiddleware } from '@nes
2  import { Request, Response, NextFunction } from
3  import { RateLimiterRedis } from 'rate-limiter-
4
5  @Injectable()
6  export class RateLimiterMiddleware implements Ne
7    private rateLimiter: RateLimiterRedis;
8
9    constructor() {
10     // Configuração do rate limiter
11     this.rateLimiter = new RateLimiterRedis({
```

BENEFÍCIOS DO RATE LIMITING

- Protege contra ataques de força bruta
- Evita que um único cliente monopolize recursos
- Mantém o sistema disponível para todos os usuários
- Reduz custos de infraestrutura

Equilibra as cargas para um serviço mais justo

MAS E PARA ADICIONAR NOVOS SERVIÇOS?

Como expandir nosso sistema sem quebrar o que já funciona?

OPEN/CLOSED PRINCIPLE

Princípio Aberto/Fechado do SOLID

> *"Entidades de software devem estar abertas para extensão, mas fechadas para modificação."*

Em outras palavras: adicione comportamentos sem mexer no código existente

EXEMPLO: MÚLTIPLOS PROVEDORES DE FRETE

E se quisermos adicionar novos provedores além dos Correios?

INTERFACE E IMPLEMENTAÇÕES

```php
// Interface comum para todos os provedores de
interface ShippingProviderInterface
{
    /**
     * Calcula o frete para um pedido
     */
    public function calculateShipping(array $or
    public function trackShipment(string $track
    public function getName(): string;
}

```

UTILIZANDO A FACTORY

```php
// Configuração da aplicação
$factory = new ShippingProviderFactory();

// Registramos todos os provedores disponíveis
$factory->registerProvider(new CorreiosShippingP
$factory->registerProvider(new FastShippingProvi
// Podemos adicionar novos provedores aqui sem m
$factory->registerProvider(new InternationalShip

// Cliente escolhe qual provedor usar
$providerName = $_POST['shipping_provider'] ??
```

BENEFÍCIOS DO OPEN/CLOSED PRINCIPLE

- Código mais estável e menos propenso a bugs
- Podemos adicionar novos provedores sem risco
- Testes existentes continuam válidos
- Facilita a manutenção e a extensão do sistema

Aplicando SOLID, ganhamos resiliência e flexibilidade.

RECAPITULANDO

Quais padrões de resiliência vimos hoje?

RETRY

Tentar novamente após uma falha, aumentando gradualmente o tempo de espera entre tentativas.

CIRCUIT BREAKER

Interromper chamadas a serviços com falhas persistentes, prevenindo sobrecarga e permitindo recuperação gradual.

FALLBACKS

Planos B para quando algo falha, oferecendo alternativas em vez de deixar a operação falhar completamente.

SAGA PATTERN

Gerenciar transações distribuídas em sistemas complexos, com compensações para desfazer operações em caso de falha.

CACHE + VALORES DEFAULT

Armazenar resultados de operações para uso quando serviços externos estiverem indisponíveis, melhorando disponibilidade.

RATE LIMITING

Limitar o número de requisições que um cliente pode fazer em um período, protegendo contra abusos e distribuindo recursos de forma justa.

OPEN/CLOSED PRINCIPLE

Adicionar novos comportamentos sem modificar o código existente, aumentando a flexibilidade e reduzindo riscos.

PRÓXIMOS PASSOS

- Estude mais sobre padrões de resiliência
- Implemente esses conceitos em seus projetos
- Participe de comunidades e compartilhe experiências

CONCLUSÃO

- Falhas são inevitáveis
- Sistemas resilientes prevêm e se adaptam a falhas
- Combine diferentes padrões para maior robustez
- A resiliência deve ser parte do design inicial

Projete para falhar!

OBRIGADO!

Luiz Schons

Senior Software Architect