Os **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.

Aqui estão os principais padrões de resiliência:

1. Circuit Breaker

O padrão **Circuit Breaker** funciona como um disjuntor elétrico, interrompendo o fluxo de chamadas para um serviço que está falhando repetidamente. Quando um serviço downstream (serviço chamado) não responde ou está falhando, o **circuit breaker** "abre", bloqueando temporariamente novas chamadas até que o serviço esteja estável novamente.

- **Estados**:
  - **Closed**: O disjuntor permite todas as solicitações.
  - **Open**: O disjuntor interrompe as solicitações, indicando que o serviço está falhando.
  - **Half-Open**: Após um período de tempo, o disjuntor permite algumas solicitações para verificar se o serviço se recuperou.
- **Benefícios**: Evita sobrecarregar um serviço em falha e proporciona uma recuperação mais suave do sistema.

2. Timeout

O padrão **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.

3. Retry

O padrão **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 **backoff** 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.

4. Fallback

O padrão **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.

5. Bulkhead

O padrão **Bulkhead** compartimenta os recursos do sistema, como memória, conexões ou threads, em diferentes "compartimentos". Isso impede que uma falha em um serviço consuma todos os recursos disponíveis, isolando o impacto de falhas em um único serviço ou componente.

- **Exemplo**: Um sistema pode dividir as conexões de banco de dados em pools separados para cada microservice. Se um microservice tiver um pico de uso e esgotar suas conexões, outros microservices ainda terão suas próprias conexões intactas.
- **Benefícios**: Garante que um único componente com falhas não derrube o sistema inteiro, promovendo isolamento de falhas.

6. Rate Limiting

O padrão **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.

7. Graceful Degradation (Degradação graciosa)

Esse padrão foca em manter o sistema funcionando, mesmo que com funcionalidade limitada, quando um serviço falha ou está sobrecarregado. Em vez de uma falha total, algumas funcionalidades podem ser desativadas ou reduzidas temporariamente.

- **Exemplo**: Em um site de comércio eletrônico, se o serviço de recomendação falhar, o sistema continua permitindo compras, apenas sem sugestões de produtos.
- **Benefícios**: Proporciona uma experiência contínua ao usuário, mesmo em momentos de falhas parciais, garantindo que funções críticas permaneçam acessíveis.

8. Caching (Cache)

O padrão **Caching** 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.

9. Steady State (Estado Estável)

Este padrão garante que o sistema volte a um estado consistente e estável após uma falha. Os sistemas monitoram continuamente o seu estado e tentam retornar ao estado desejado após eventos anômalos.

- **Benefícios**: Mantém o sistema funcionando conforme esperado, mesmo depois de situações inesperadas.

Resumo dos Benefícios de Padrões de Resiliência:

- **Melhoria na disponibilidade**: Mantêm o sistema disponível mesmo em situações de falhas.
- **Redução do impacto das falhas**: Limita o escopo e as consequências das falhas, prevenindo que elas se espalhem para todo o sistema.
- **Experiência do usuário aprimorada**: Minimiza interrupções e garante que os usuários finais percebam menos problemas durante falhas.
- **Otimização de recursos**: Garante que o sistema utilize seus recursos de maneira mais eficiente, mesmo durante sobrecargas ou picos de tráfego.

Implementar padrões de resiliência de forma eficaz é essencial para garantir que sistemas distribuídos de microservices sejam robustos, escaláveis e capazes de lidar com os desafios inevitáveis de falhas em um ambiente complexo.