

Patrón Circuit Breaker

Una guía completa de implementación y mejores prácticas



Definición Conceptual del Patrón Circuit Breaker

¿Qué es?

Un patrón de diseño que **monitorea las fallas** en llamadas a servicios y **evita operaciones destinadas a fallar**.

Propósito Principal

- Prevenir fallos en cascada
- Proteger recursos del sistema
- Fallar rápidamente cuando un servicio está inoperativo
- Recuperación automática cuando el servicio vuelve a estar disponible
- Proporcionar respuestas alternativas durante fallas



Interruptor Eléctrico

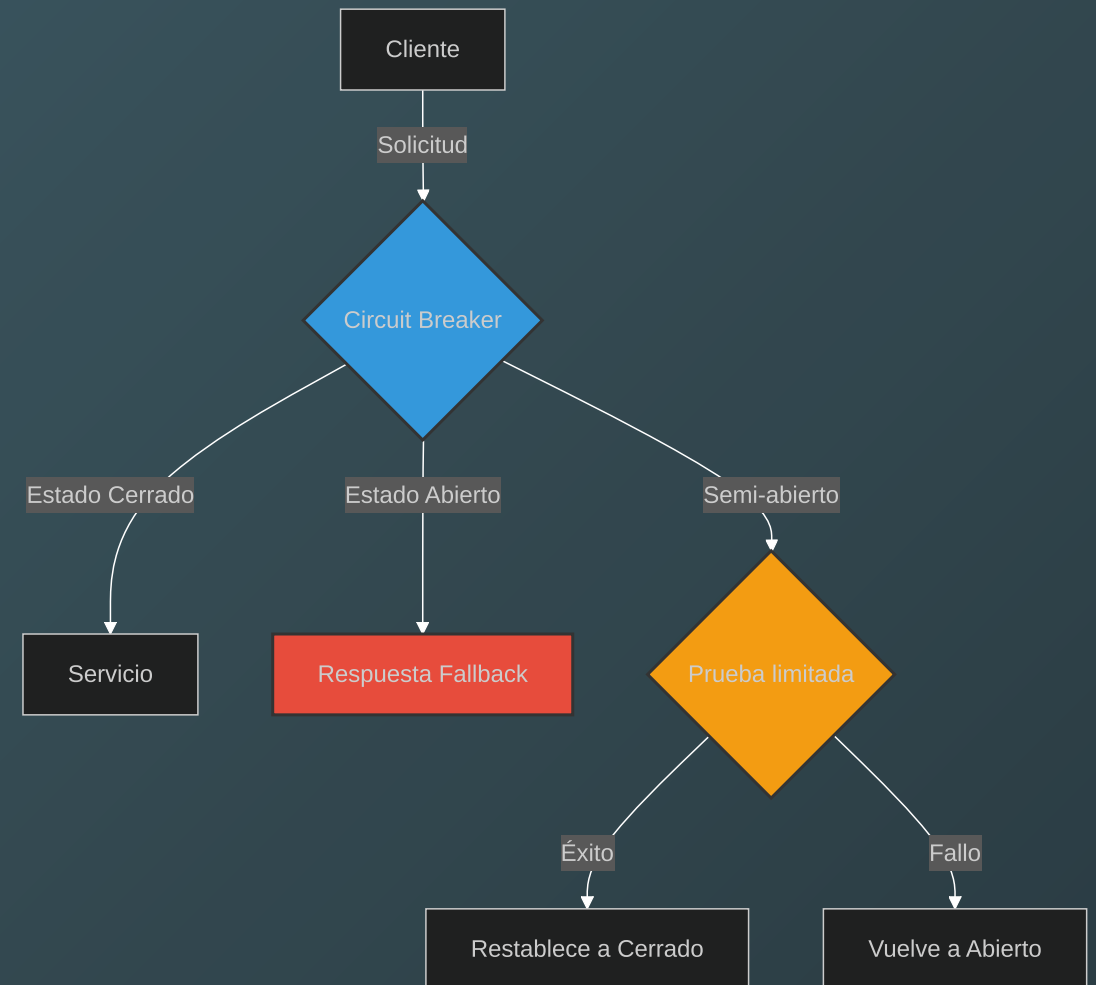
Corta el flujo de electricidad para prevenir daños por sobrecarga



Circuit Breaker

Interrumpe las llamadas a servicios fallidos para evitar sobrecarga del sistema

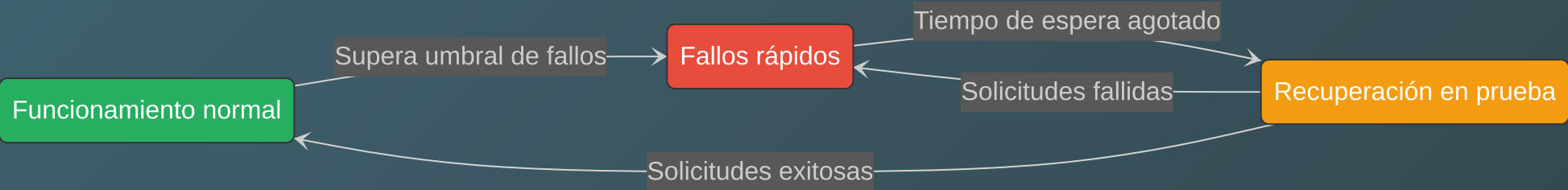
Funcionamiento Básico



Beneficios Clave

- 🛡️ Mejora la resiliencia del sistema
- ⌚ Mantiene tiempos de respuesta
- ⚡ Permite fallo rápido
- 🔄 Recuperación automática

Estados del Circuit Breaker



Estado CERRADO



- **Comportamiento:** Todas las solicitudes pasan al servicio
- **Monitoreo:** Registra éxitos y fallos
- **Transición:** Cambia a ABIERTO cuando los fallos superan el umbral

Métricas clave:

- Tasa de fallos: < umbral configurado
- Solicitudes: procesadas normalmente

Estado ABIERTO



- **Comportamiento:** Fallo rápido, sin intentar llamar al servicio
- **Respuesta:** Retorna fallback o error inmediato
- **Transición:** Cambia a SEMI-ABIERTO tras periodo de espera

Métricas clave:

- Tiempo de reset: configurable (ej: 5s, 30s)
- Solicitudes: rechazadas inmediatamente

Estado SEMI-ABIERTO



- **Comportamiento:** Permite número limitado de solicitudes
- **Propósito:** Probar si el servicio se ha recuperado
- **Transición:** A CERRADO si las pruebas son exitosas, o ABIERTO si fallan

Métricas clave:

- Solicitudes de prueba: número limitado
- Umbral éxito: configurable (ej: 5 exitosas consecutivas)

Reglas de Transición

- **Cerrado** → **Abierto**: Cuando las fallas superan un umbral configurable (ej: > 50% en 10 solicitudes)
- **Abierto** → **Semi-abierto**: Cuando transcurre el tiempo de espera configurado (ej: 30 segundos)
- **Semi-abierto** → **Cerrado**: Cuando las solicitudes de prueba son exitosas (ej: 5 consecutivas)
- **Semi-abierto** → **Abierto**: Si las solicitudes de prueba fallan (ej: 1 fallo)

Ventajas y Consideraciones del Circuit Breaker

Ventajas



Prevención de fallos en cascada

Evita que un fallo en un servicio afecte a todo el sistema, aislando los componentes problemáticos



Fallo rápido (Fail-fast)

Responde inmediatamente sin esperar timeouts cuando un servicio está caído



Reducción de latencia

Evita esperas innecesarias durante periodos de fallo, mejorando tiempos de respuesta



Recuperación automática

Restablecimiento gradual de llamadas cuando el servicio vuelve a estar disponible



Mejora experiencia de usuario

Proporciona respuestas alternativas en lugar de errores o tiempos de espera prolongados

Consideraciones



Complejidad adicional

Requiere implementación y mantenimiento de lógica adicional en el código



Configuración de umbrales

Definir correctamente los umbrales de fallo y tiempos de recuperación puede ser complejo



Falsos positivos

Posibilidad de abrir el circuito prematuramente debido a fallos intermitentes o transitorios



Estrategia de fallback

Necesidad de desarrollar y mantener lógica alternativa para respuestas durante fallos



Monitorización necesaria

Requiere observabilidad para analizar comportamiento y ajustar parámetros

Balance y Recomendaciones

- ✓ Imprescindible en comunicaciones entre microservicios dependientes
- ✓ Ajustar parámetros basado en características del servicio
- ✓ Combinable con otros patrones de resiliencia para mayor robustez

Implementación de Circuit Breaker en Java

⚙ Configuración en application.yml

application.ymlSpring Boot

```
resilience4j:
  circuitbreaker:
    instances:
      servicio-inventario:
        registerHealthIndicator: true
        slidingWindowSize: 10
        minimumNumberOfCalls: 5
        permittedNumberOfCallsInHalfOpenState: 3
        automaticTransitionFromOpenToHalfOpenEnabled: true
        waitDurationInOpenState: 5s
        failureRateThreshold: 50
        eventConsumerBufferSize: 10
        recordExceptions:
```

💡 Los valores se personalizan según las necesidades específicas de cada servicio

</> Implementación con anotaciones

InventarioService.javaAnotaciones

```
@Service
public class InventarioService {

    private final RestTemplate restTemplate;

    public InventarioService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    @CircuitBreaker(name = "servicio-inventario",
                    fallbackMethod = "getInventarioFallback")
    public List<ProductoDTO> getInventario() {
        // Llamada al servicio remoto que podría fallar
    }
}
```

📄 Anotación @CircuitBreaker para integración simple con Spring

📄 API funcional para escenarios avanzados y WebFlux reactivo

📄 Método fallback para respuesta alternativa durante fallos

✂ Configuración programática

CircuitBreakerConfig.javaJava Config

```
@Configuration
public class CircuitBreakerConfig {

    @Bean
    public CircuitBreakerRegistry circuitBreakerRegistry() {
        // Definir la configuración del circuit breaker
        CircuitBreakerConfig circuitBreakerConfig = CircuitBreakerConfig
            .failureRateThreshold(50)
            .waitDurationInOpenState(Duration.ofMillis(1000))
            .slidingWindowSize(10)
            .permittedNumberOfCallsInHalfOpenState(5)
            .recordExceptions(IOException.class, TimeoutException.class)
            .build();
    }
}
```

Uso con decoradores funcionales


InventarioClient.javaFuncional

```
@Component
public class InventarioClient {

    private final WebClient webClient;
    private final CircuitBreaker circuitBreaker;

    public InventarioClient(WebClient.Builder webClientBuilder,
                            CircuitBreakerRegistry registry) {
        this.webClient = webClientBuilder
            .baseUrl("http://servicio-inventario")
            .build();
        this.circuitBreaker = registry.circuitBreaker("servicio-inventario");
    }
}
```

Patrones de Resiliencia en Microservicios



Circuit Breaker


Previene llamadas a servicios fallidos, proporcionando respuestas alternativas

CASOS DE USO

Servicios inestables

Prevenir cascadas

Recuperación gradual



Retry


Reintenta operaciones fallidas siguiendo una estrategia configurable

CASOS DE USO

Fallos transitorios

Conectividad intermitente

Recursos momentáneamente no disponibles



Timeout


Limita el tiempo de espera para operaciones y libera recursos

CASOS DE USO

Servicios lentos

Evitar bloqueos

Liberar recursos



Bulkhead


Aísla componentes del sistema para contener fallos

CASOS DE USO

Aislamiento de recursos

Prevenir agotamiento

Priorizar servicios críticos



Rate Limiter

Controla la cantidad de solicitudes por unidad de tiempo

CASOS DE USO



Protección contra sobrecargas

Limitar consumo de API



Prevenir DoS





Combinación de Patrones

 +  **Circuit Breaker + Retry**



Reintentar fallos transitorios mientras se previenen cascadas en fallos persistentes

 +  **Circuit Breaker + Timeout**

Evitar esperas excesivas y abrir el circuito antes fallos por timeout

 +  **Circuit Breaker + Bulkhead**

Aislar recursos y prevenir fallos entre componentes del sistema

 +  **Circuit Breaker + Rate Limiter**

Limitar peticiones a servicios y evitar sobrecarga en recuperación

Estrategia óptima

Combinar varios patrones de resiliencia proporciona capas de protección complementarias. Configurar cada patrón según las necesidades específicas del servicio.

Mejores Prácticas para Circuit Breaker



Configuración de Umbrales

- ✓ Umbral de fallos: 50-60% para equilibrar sensibilidad
- ✓ Tamaño de ventana: 10-20 solicitudes o 60s para servicios de bajo volumen
- ✓ Número mínimo de llamadas: 5-10 antes de evaluar tasa de fallos
- ✓ Ajustar umbrales según la criticidad del servicio y patrón de tráfico



Tiempos de Espera

- ✓ Tiempo en estado abierto: 10-60 segundos según tiempo de recuperación típico
- ✓ Timeout de llamada: inferior al del Load Balancer o API Gateway
- ✓ Permitir transición automática de abierto a semi-abierto
- ✓ Implementar backoff exponencial para reintentos en estado semi-abierto



Monitorización

- ✓ Implementar dashboard de estado para todos los circuit breakers
- ✓ Configurar alertas para transiciones frecuentes entre estados
- ✓ Registrar métricas: tasa de fallos, latencia, tasa de llamadas
- ✓ Exponer endpoint de Health Check para estado del circuit breaker



Estrategias de Fallback

- ✓ Implementar respuestas predeterminadas significativas para el usuario
- ✓ Considerar cache local para datos críticos o de cambio poco frecuente
- ✓ Utilizar servicios redundantes cuando sea posible
- ✓ Degradar funcionalidad en lugar de fallar completamente



Consideraciones Generales

- ✓ Personalizar configuración para cada servicio según SLA
- ✓ Documentar estrategias de fallback y comportamiento esperado
- ✓ Testear comportamiento con simulaciones de fallos
- ✓ Revisar y ajustar configuraciones periódicamente



Consejo de Experto

Implementa Circuit Breakers específicos para diferentes tipos de errores. Un circuito para errores de red podría tener una configuración distinta a uno para errores de base de datos o de autenticación.

Herramientas y Bibliotecas Recomendadas



Resilience4j

Ligero

Java 8+

Activo

Biblioteca inspirada en Hystrix pero diseñada específicamente para Java 8 con programación funcional

+ Pros

✓ Ligero, sin dependencias

✓ Programación funcional

✓ Desarrollo activo

- Contras

✗ Curva de aprendizaje

✗ Menos plugins



Netflix Hystrix

Maduro

Java 7+

Mantenimiento

Biblioteca pionera de Netflix para implementar el patrón Circuit Breaker y tolerancia a fallos

+ Pros

✓ Muy probado

✓ Dashboard

✓ Bastante documentación

- Contras

✗ Modo mantenimiento

✗ Monolítico

✗ Overhead



Spring Cloud Circuit Breaker

Abstracción

Spring

Activo

API de abstracción que permite elegir entre múltiples implementaciones (Resilience4j, Hystrix, etc.)

+ Pros

✓ API unificada

✓ Integración Spring

✓ Intercambiable

- Contras

✗ Funcionalidad reducida

✗ Otra capa de abstracción

Comparativa de Características

Característica	Resilience4j	Hystrix	Spring Cloud CB
Estado de desarrollo	✓ Activo	⏸ Mantenimiento	✓ Activo
Footprint	Ligero	Pesado	Medio
Patrones soportados	Completo	Limitado	Varía
Integración con Spring	Nativa	Legacy	Nativa
Monitorización	Customizable	Dashboard	Depende
Recomendación	Proyectos nuevos	Legacy	Ecosistema Spring

Evaluación Global




Recursos Adicionales

 [Documentación Resilience4j](#)

 [GitHub Resilience4j](#)

 [Wiki Hystrix](#)

 [Spring Cloud Circuit Breaker](#)

 [Martin Fowler - Circuit Breaker](#)