Circuit Breaker Design Pattern

Definition:

The Circuit Breaker Design Pattern is used in Java microservices to handle potential failures gracefully and improve the system's resilience. It acts as a safety switch to prevent cascading failures in distributed systems.

Key Concepts:

Closed State: Requests are allowed through. If failures reach a threshold, the circuit trips to Open.
Open State: Requests are blocked, and failures are immediately returned. This prevents overloading the failing service.
Half-Open State: After a timeout, a few requests are allowed to test if the service has recovered. If successful, the circuit closes again; otherwise, it reopens.

Benefits:

Fault Isolation: Prevents a single service failure from affecting others.
Improved Resilience: Allows systems to degrade gracefully instead of failing entirely.
Faster Recovery: Reduces load on failing services, helping them recover faster.

Frameworks like Resilience4j or Hystrix are commonly used to implement the Circuit Breaker pattern.

Single Database Per Service

Definition:

The Single Database per Service design pattern is a key principle in microservices architecture that ensures each microservice has its own dedicated database. This pattern supports the autonomy, scalability, and resilience of microservices.

Features

Service Ownership: Each microservice is responsible for managing its own database schema, data, and access logic.

Decoupling: Services are not tightly coupled through shared databases, ensuring changes in one service do not directly affect others.

Data Isolation: Prevents direct access to a service’s data by other services. Communication happens through APIs or events.

Benefits:

Encapsulation: Data access logic remains confined within the service, promoting loose coupling.
Independent Scaling: Each database can be optimized and scaled based on the specific service’s needs.
Technology Flexibility: Different microservices can use different types of databases (e.g., SQL, NoSQL) based on their requirements.
Fault Isolation: Database issues in one service don't propagate to others.

Challenges:

Data Consistency: Maintaining consistency across multiple services becomes complex, especially during distributed transactions.
Data Duplication: Related data may need to be replicated across databases, increasing storage requirements.
Communication Overhead: Cross-service data access requires inter-service communication (e.g., via APIs), which can add latency.

API Gateway Design Pattern

Features

API Gateway Design Pattern (cont)

Routing: Routes incoming client requests to the appropriate microservices.

Aggregation: Combines responses from multiple services to provide a single, consolidated response.

Cross-Cutting Concerns: Handles common tasks like authentication, authorization, rate limiting, caching, logging, and monitoring.

Protocol Translation: Converts protocols (e.g., HTTP to gRPC) if needed

Advantages

Simplifies client interactions by abstracting service details.

Reduces client-to-service communication complexity.

Centralizes concerns like security and logging, ensuring consistency.

Challenges

Can become a single point of failure if not designed with redundancy.

May introduce latency due to added processing.

Requires proper scaling to handle high traffic.

Tools Used for Implementation:

Netflix Zuul: A flexible API Gateway.

Kong: An open-source API Gateway with plugin support.

Spring Cloud Gateway: A Java-based API Gateway for Spring applications.

AWS API Gateway: A managed solution by AWS.

BulkHead Design Pattern

Key Concept

The system is divided into independent partitions (bulkheads), like compartments in a ship, to contain failures within one partition.

Each bulkhead has dedicated resources (e.g., thread pools, memory, connections) to prevent resource exhaustion from affecting other parts of the system.

Pros:

Fault Isolation: Limits the impact of failures to specific services or components.

Improved Resilience: Ensures critical parts of the system remain operational even when some fail.

Resource Protection: Prevents one service from consuming all available resources.

Challenges

Overhead due to managing multiple resource pools.

Requires careful tuning of resource limits for optimal performance

Implementation Tools

Use **thread pools, connection pools, or process isolation** to allocate resources to specific services.

Frameworks like **Resilience4j** support bulkhead isolation.

Strangler Fig Design Pattern

Key Concept:

Inspired by the way a strangler fig tree grows around a host tree, eventually replacing it.

New features are built as independent microservices, while the monolithic application continues to handle legacy functionality.

Over time, the monolith is "strangled" as its functionality is replaced by microservices.

Process

Strangler Fig Design Pattern (cont)

Identify Modules: Analyze the monolith and identify modules or functionalities to migrate.
Build New Services: Create microservices for new features or existing modules.
Redirect Traffic: Use an API Gateway or routing layer to direct requests to the appropriate service (new microservice or monolith).
Decommission Monolith: Gradually phase out the monolith as its responsibilities are fully transitioned to microservices.

Pros

Incremental Migration: Avoids the risks of a complete rewrite.
Reduced Downtime: Allows the system to remain operational during migration.
Modernization: Enables the adoption of new technologies and practices.

Challenges:

Complexity in Integration: Requires careful coordination between the monolith and microservices.
Extended Migration Time: The process can take a long time to complete.

The Strangler Fig pattern allows for a smooth and low-risk transition to microservices while continuously delivering value.

Fallback Design Pattern

Definition:

The **Fallback Design Pattern** in microservices is a resilience pattern used to provide a backup response or alternative action when a service fails or is unavailable. It ensures that the system remains partially functional, offering a better user experience during failures.

Features:

Backup Response: Provides a default or cached response when the primary service fails.
Graceful Degradation: Allows the application to continue operating with limited functionality instead of crashing.
User Impact Mitigation: Minimizes disruptions for users by avoiding complete failures.

Pros:

Improves system resilience and user experience.
Reduces the impact of temporary service failures.
Supports fault-tolerant design in distributed systems.

Challenges

Designing meaningful fallback responses can be complex.
Fallback logic must be carefully tested to avoid unexpected behavior.

Frameworks like Resilience4j or Hystrix make implementing fallbacks.

Command Query Responsibility Segregation

Definition

The **CQRS** (Command Query Responsibility Segregation) design pattern in microservices separates the operations that modify data (commands) from the operations that read data (queries).
This pattern optimizes performance, scalability, and flexibility in handling complex business requirements.

Key Concepts

Commands: Handle write operations (e.g., creating, updating, or deleting data).
Queries: Handle read operations to retrieve data.
Separate Models: Use distinct data models for write (transactional consistency) and read (optimized for queries).

Command Query Responsibility Segregation (cont)

Pros

Scalability: Scale read and write operations independently.
Performance: Optimize query responses with denormalized or precomputed data.
Flexibility: Tailor read and write models to different requirements.

Challenges

Increased complexity due to managing separate models and synchronization.
Potential for eventual consistency between write and read models.

Use frameworks like Axon or Spring Boot to simplify CQRS implementation.

Saga Design Pattern

Definition

The **Saga Design Pattern** is a microservices design pattern for managing distributed transactions
in a consistent and reliable way without using a centralized transaction coordinator.
It breaks down a transaction into a sequence of smaller,
independent steps (local transactions) across services, ensuring eventual consistency.

Concepts

Choreography: Each service independently performs its local transaction and publishes events for others to react to.
Orchestration: A central orchestrator coordinates the sequence of steps by invoking the necessary services.
Compensation: If a failure occurs, previously completed steps are undone using compensating actions.

Pros

Enables distributed transactions without locking resources.
Improves system resilience and scalability.
Works well in asynchronous and event-driven architectures.

Implementation Tools

Use tools like Kafka or RabbitMQ for event-driven sagas.
Frameworks like Axon or Camunda help with orchestration.

The Saga pattern ensures that distributed transactions are managed reliably while maintaining the independence of microservices.

Messenging Design Pattern

Definition

The Messaging Design Pattern in microservices facilitates asynchronous communication between services using message brokers or event buses. It decouples services, enabling them to operate independently while improving scalability and resilience.

Important Concepts

Asynchronous Communication: Services exchange messages without waiting for immediate responses.
Message Broker: A central component (e.g., Kafka, RabbitMQ, AWS SQS) routes messages between producers (senders) and consumers (receivers).
Decoupling: Producers and consumers are independent, knowing only about the message format, not each other.

Pros

Scalability: Services can process messages at their own pace.
Fault Tolerance: Messages can be retried or queued if a service is unavailable.
Decoupling: Changes in one service don't require changes in others.

Challenges

Messenging Design Pattern (cont)

Message Ordering: Ensuring the correct sequence of message processing can be complex.

Message Duplication: Requires handling duplicate messages for idempotency.

Operational Overhead: Managing message brokers adds complexity.

Implementation Tools

Use messaging protocols like AMQP, MQTT, or Kafka Streams.

Ensure messages are durable, idempotent, and well-structured (e.g., JSON, Avro).

The Messaging pattern is fundamental in microservices for building resilient, scalable, and loosely coupled systems.

By **Kiran P** (kirandp)
cheatography.com/kirandp/

Published 13th January, 2025.
Last updated 13th January, 2025.