The **SOLID principles** are five foundational guidelines in object-oriented programming that help developers write cleaner, more maintainable, and scalable code. Each letter in "SOLID" stands for a specific principle

**Single Responsibility Principle (SRP) :**

A class should have only one reason to change, meaning it should have only one responsibility. This promotes high cohesion within a class.

* A class should have **only one reason to change**.
* It should do **one job** and do it well.
* Example: A class that handles user authentication shouldn’t also manage database connections

**Open-Closed Principle (OCP) :**

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. This means you should be able to add new functionality without altering existing, working code.

* Software entities (classes, modules, functions) should be **open for extension but closed for modification**.
* You should be able to add new functionality without changing existing code.
* Example: Use inheritance or interfaces to extend behaviour.

**Liskov Substitution Principle (LSP) :**

Subtypes must be substitutable for their base types without altering the correctness of the program. If class S is a subtype of class T, then objects of type T may be replaced with objects of type S without breaking the application.

* Subtypes must be **substitutable** for their base types.
* If class B is a subclass of class A, then objects of A should be replaceable with objects of B **without breaking the program**.
* Example: If a Bird class has a fly() method, a Penguin subclass shouldn’t override it to throw an error.

**Interface Segregation Principle (ISP) :**

Clients should not be forced to depend on interfaces they do not use. This suggests creating smaller, more specific interfaces rather than large, monolithic ones.

* Clients should not be forced to depend on interfaces they **do not use**.
* Split large interfaces into smaller, more specific ones.
* Example: Instead of one big interface with 10 methods, create 3 smaller interfaces with related methods.

**D — Dependency Inversion Principle (DIP) :**

High-level modules should not depend on low-level modules; both should depend on abstractions. Additionally, abstractions should not depend on details; details should depend on abstractions. This promotes loose coupling and easier testing.

* High-level modules should not depend on low-level modules. Both should depend on **abstractions**.
* Abstractions should not depend on details. Details should depend on abstractions.
* Example: Use interfaces or abstract classes so that high-level logic isn’t tightly coupled to low-level implementations.

**What are Microservices?**

Microservices are an architectural approach to developing software applications as a collection of small, independent services that communicate with each other over a network. Instead of building a monolithic application where all the functionality is tightly integrated into a single codebase, microservices break down the application into smaller, loosely coupled services.

**Microservices Principles :**

**Autonomy**: Each microservice should be independently deployable and operate without relying on others. This allows for faster development and easier scaling.

**Single Responsibility**: A microservice should do one thing and do it well—this keeps services focused and manageable.

**Decentralized Data Management**: Each service should manage its own database to avoid tight coupling and ensure data integrity.

**API-First Communication**: Services interact via well-defined APIs, often RESTful or gRPC, ensuring language-agnostic communication.

**Failure Isolation**: If one service fails, it shouldn’t bring down the entire system. Circuit breakers and retries help maintain **resilience**.

**Scalability**: Services should be designed to scale independently based on demand. This is especially powerful in cloud environments.

**Continuous Delivery & DevOps Integration**: Microservices should support rapid deployment cycles and integrate smoothly with CI/CD pipelines.

**Monitoring & Logging**: Each service must be observable with robust logging and metrics to detect issues early.

**Versioning**: APIs should be versioned to allow backward compatibility and smooth upgrades.

**Realtime Load Balancing:**

Sometimes it may happen when a client sends the request to the server then the database has to retrieve the data from multiple microservices at the same time. At that point, the Load balancer comes into the picture and it defines how much amount of the **Central Processing Unit or GPU** is to be used for a particular service to fetch the data and finally how the client request should be passed.

What Is Load Balancing?

Load balancing is the process of distributing incoming network traffic across multiple servers. It helps ensure:

**High availability**: If one server fails, traffic is rerouted to healthy ones.

**Scalability**: As demand grows, more servers can be added without disrupting service.

**Performance**: Requests are handled faster because no single server is overloaded.

How it works:

**1. Receives requests:**

A load balancer sits in front of multiple instances of a specific microservice (e.g., multiple instances of a user service).

**2. Distributes traffic:**

It takes incoming requests from clients and distributes them to different available instances of that service.

**3. Uses algorithms:**

This distribution is done using various algorithms, such as:

* + **Round Robin:** Distributes requests to each server in turn.
  + **Least Connections:** Sends requests to the server with the fewest active connections.

**4. Ensures health:**

The load balancer continuously monitors the health of each service instance, only routing traffic to healthy ones.

**Microservice Design Patterns :**

[**https://www.geeksforgeeks.org/system-design/microservices-design-patterns/**](https://www.geeksforgeeks.org/system-design/microservices-design-patterns/)

**1.API Gateway Design Pattern :**

API - It stands for "Application Programming Interface".

API Gateway Pattern is a design approach used in microservices architecture to simplify and centralize communication between client applications and backend services.

**Use of the API Gateway Pattern:**The main responsibility of this pattern is that it routes the request means basically provide a road map for how our request goes, approve or may be cancelled, API  composition, and app authentication.

**How API Gateway Handles the Client requests:**

* API requests mostly come from clients who are external and firstly go to API gateway.
* It is the working of API gateway which routes the request to their place.
* The other additive requests i.e. using multiple services and the aggregating result are handled by API gateway.A diagram of a diagram

  AI-generated content may be incorrect.

**Advantages of API gateway pattern -**

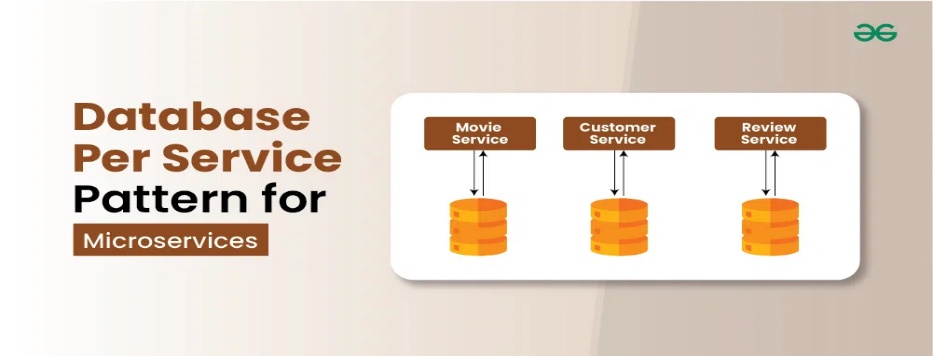
* It encloses the whole internal structure of web applications.
* It never calls a particular service. For example, client interaction with API gateway.
* It helps in the simplification of code of the client-side.

**Disadvantages of API gateway pattern -**

* It is an important component for every web application means the web application services will be shown only if the API is up-to-date means updated.
* It becomes very important for each process for being lightweight because otherwise their time complexity will get increased because their developer has to wait in the process of updating API.

2. [**Database per Service Pattern**](https://www.geeksforgeeks.org/database-per-service-pattern-for-microservices/)

Each microservice has its own database, ensuring loose coupling and independent data management. This pattern avoids a single point of failure and allows services to use different types of databases suited to their needs.



**3. Circuit Breaker :**

This pattern prevents service failure by providing a fallback mechanism when a service is unreachable or fails. It monitors service calls and "breaks" the circuit to prevent further calls when failures exceed a threshold.

When one service fails and gives error messages continuously, we can stop calling it for a period of time and use a fallback.

* Purpose: Prevents continuous failing calls from burdening the system.
* States:
  1. Closed: Normal flow, failures are counted.
  2. Open: When failure rate > threshold (e.g., 50% of 3 calls).
     + All calls are blocked, fallback is used directly.
     + Waits for waitDurationInOpenState (e.g., 5s).
  3. Half-Open: Tests limited number of calls (e.g., 2).
     + If successful → circuit goes back to Closed.
     + If failed → goes back to Open.

=> circuit will remain open for 5 seconds, then tests 2 calls in Half-Open."

**Actual method Fallback method**

A diagram of a process

AI-generated content may be incorrect.

The Circuit Breaker *opens* after 50% of the recent calls in the sliding window fail (not after making 50% of total calls). It's a failure rate threshold, not a count of total calls.

**Resilience4j config example:**

failureRateThreshold : 50 # % of failures to trigger open state

slidingWindowSize : 10 # Check the last 10 calls

waitDurationInOpenState: 5s # Remains open for 5s before trying again

permittedNumberOfCallsInHalfOpenState: 2

**Retry**:

* Retry will **immediately retry the failed call** X number of times with Y interval between retries.
* If all retries fail → fallback is triggered (or exception is thrown).

maxAttempts: 3 # initial + 2 retries

waitDuration: 1s # wait 1s between retries

**Rate Limiter:**

* Controls the number of calls allowed in a time window, e.g., 2 requests per 10 seconds.
* If the limit is reached, it throws RequestNotPermitted → fallback (or error).

resilience4j:

ratelimiter:

instances:

externalApiRateLimiter:

limitForPeriod: 2

limitRefreshPeriod: 10s

timeoutDuration: 0s

**4.**[Service Discovery Pattern](https://www.geeksforgeeks.org/server-side-service-discovery-in-microservices/)

Service Discovery allows microservices to find and communicate with each other dynamically. It involves a service registry where services register themselves and look up other services.

There are two types of Service Discovery

* Client-Side Service Discovery
* **Server-Side Service Discovery**

5. [**Event Sourcing Pattern**](https://www.geeksforgeeks.org/event-sourcing-pattern/)

This pattern captures changes to an application state as a sequence of events. Instead of storing just the current state, it stores the state changes (events), allowing the system to reconstruct past states and audit trails.

**What is Event Sourcing?**

The Event Sourcing Pattern is like keeping a detailed diary for your software. Instead of just updating the current state of your data, you record every change as a separate event. These events form a complete history of what happened to your data over time. Therefore, you may rebuild your data by replaying these events to find out how it got to where it is now.

**6.**[CQRS (Command Query Responsibility Segregation) Pattern](https://www.geeksforgeeks.org/cqrs-command-query-responsibility-segregation/)

CQRS separates the read and write operations of a data store. Commands (write operations) update the state, while queries (read operations) fetch data from a different model optimized for reads.

**What is the CQRS Design Pattern?**

CQRS, is a design pattern that divides the task of managing commands and inquiries among several components. Separating the methods for reading and publishing data is the primary goal of the CQRS architectural pattern. It separates the read and update operations on a datastore into two separate models: Queries and Commands, respectively.

**7.**[Saga Pattern](https://www.geeksforgeeks.org/saga-design-pattern/)

Saga manages distributed transactions across multiple microservices by coordinating a sequence of local transactions. Each service performs its transaction and publishes an event triggering the next service’s transaction. If a transaction fails, compensating transactions undo the changes.

**What is a Distributed Transaction?**

A distributed transaction refers to a type of transaction that involves multiple, separate systems or databases, often spread across different locations or networks, which need to work together to complete a task. It’s like a coordinated team effort, where each system handles a small part of the work, but they all must complete their respective tasks successfully for the overall transaction to be considered successful.

This problem started as soon as we moved grom Monolithic application to Microservices Architecture

We will take example of Swiggy, Zomato

* Coose your dishes.
* Add them to Cart and checkout.
* Make Payment.
* Order gets Delivered.
* Our Order is marked as completed after delivery is successful.

**Monolithic:**

In Monolithic it’s not a problem as we have 1 database, multiple tables like orders Payments, Delivery etc…Now in 1 Single atomic transaction we can do all these steps and if payments fails everything gets rolled back.

**Microservices:**

Now we moved to microservices architecture and Segregated the whole Zomato or Swiggy application to

* Order Service
* Payment Service
* Delivery Service

Now your order service accepts your order. Payment service validates the payments done and delivery service is responsible for delivery of your order to your home. When delivery successfully the order is marked completed in the application. This is happy case.

Ever thought about the worst-case Delivery is failed as no delivery partner was available. Your payment was done. Money got deducted and now No food. At least we need to get money back and order must be marked as cancelled.

For this to happen we need a Transaction rollback. Transaction did get rolled back but only the scope of transaction was in Delivery service. The boundary for this transaction ended in Delivery service.

Now what about the order service and payment service ??

Neither your money is returned with this rollback nor your status changed from waiting to failed / Cancelled. Such a bad user experience right ?

This is the classic example where your application completely failed to manage distributed transaction(A transaction what span across multiple Microservices). Now this is a problem and to handled such distributed transaction issues SAGA Design pattern came into picture.

**What is SAGA :**

A saga is that sequence of local transactions.

Each saga has 2 jobs to Do

* Update the current Microservice and make required changes.
* Publish events to trigger the next transaction for the next microservices

**How SAGA DP handles failure of any individual SAGA ?**

The saga patterns provide transaction management with using a sequence of local transactions of microservices. Every microservices ha its own database and it can to manage local transaction in atomic way with strict consistency.

So saga pattern grouping this local transaction and sequentially invoking one by one. Each local transaction updates the database and publishes an event to trigger the next level transaction.

If one of the step is failed than patterns trigger to rollback transaction that are a set of compensating transaction that rollback the changes on previous microservices and restore and consistency.

There are two types of saga implantation ways.

Choreography

Orchestration

**Choreography :**

Choreography is a way to coordinate sagas where participate exchange events without a centralized point of control.

With choreography each microservices run its own local transaction and publishes events to message broker system and that trigger local transactions in other microservices.

**Advantages :**

* Good for simple workflows that require few participants and don’t need coordination logic.
* Doesn’t require additional service implementation and maintenance.
* Doesn’t introduce a single point of failure, since the responsibilities are distributed across the saga participants.

**Orchestration :**

Orchestration is way to **coordinate sagas where a centralized controller** tells the saga participants what local transactions to execute.

The **saga Orchestrator handles all the transactions** and **tells the participants which operation to perform based on events**.

The Orchestrator

Executes saga requests,

Stores and interprets the states of each task.

Handles failure recovery with compensating transactions.

**Disadvantages :**

Workflow can become confusing when adding new steps, as it’s difficult to track which saga participants listen to which commands.

There’s a risk of cycle dependency between saga participants because they have to consume each other’s commands.

Integration testing difficult because all services must be running to simulate a transaction.

**Advantages :**

Good for complex workflows involving many participants or new participants added over time.

Suitable when there is control over participants in the process and control over the flow of activities.

Doesn’t introduce cyclic dependencies, because the orchestrator unilaterally depends on saga participants.

Saga participants don’t need to know about commands for the other participants. Clear separation of concerns simplifies business logic

**Feign Client:**

**Feign** is a **declarative REST client** provided by **Spring Cloud**. It simplifies calling REST services in **microservice** architecture by eliminating the boilerplate code you would write using Rest Template or Web Client.

Instead of manually building HTTP requests, you **define an interface**, annotate it, and Feign does the rest.

| **Traditional Way (RestTemplate)** | **Vs Feign Client** |
| --- | --- |
| Manually build URL, headers, etc. | Just define method and annotate |
| More boilerplate code | Cleaner, declarative approach |
| Error handling/manual retry | Easily integrates with Resilience4j (Circuit Breaker, Retry) |
| No service discovery | Integrates with Eureka for dynamic lookup |

**Zipkin – Distributed Tracing**

**Purpose:** Zipkin is a **distributed tracing system**. It helps **track requests across multiple services** to understand the flow and identify latency bottlenecks.

**Key Use Cases:**

* Trace the path of a single request across microservices
* Visualize service-to-service communication
* Debug performance and availability issues
* Latency bottlenecks- shows the **slowest component** in your system causing **delays** in overall performance.
* Failures in downstream services

Spring Cloud micrometer adds tracing capabilities (e.g., trace IDs, span IDs) and can send the trace data to Zipkin.

Client → API Gateway → Order Service → Payment Service

To track this request end-to-end, distributed tracing uses **Trace ID** and **Span ID**.

**1. Trace ID**

* A unique ID for the **entire request journey**.

**2. Span ID**

* A unique ID for a **single unit of work** within a service.
* Every service or method creates its own **span**.

Example:

Client → API Gateway [Span ID = 1a]

→ Order Service [Span ID = 2b]

→ Payment Service [Span ID = 3c]

Each spanID also stores:

* Duration (how long it took)
* Parent span (who called it)
* Tags (extra info like HTTP method, status, etc.)

**Actuators:**

Spring Boot Actuator provides **production-ready endpoints** that help you monitor and manage your application without having to write custom code.

Enable Spring Boot Actuator using below dependency.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

In application. properties file use below key-values:

management.endpoints.web.exposure.include=\*

| **Feature** | **Description** |
| --- | --- |
| /actuator/health 🡪 | Shows application health (DB up/down, etc.) |
| /actuator/metrics 🡪 | Shows JVM, memory, CPU, HTTP request metrics, etc. |
| /actuator/env 🡪 | Exposes environment properties (with security) |
| /actuator/beans 🡪 | Lists all Spring beans |
| /actuator/mappings🡪 | Lists all HTTP mappings (controller endpoints) |
| /actuator/loggers 🡪 | View or change logging levels dynamically |

**Admin Server:**

Spring Boot Admin is a powerful tool used to monitor and manage Spring Boot applications in real-time through a web-based UI.

Used to monitor and visualize the status, health and metrics of the Springboot applications that exposes Actuators endpoints.

Using Admin Client, we can register our all Springboot applications with the Server.

Use @EnableAdminServer and below dependencies.

For Admin Server

<dependency>

<groupId>de.codecentric</groupId>

<artifactId>spring-boot-admin-starter-server</artifactId>

</dependency>

Admin Server starts at <http://localhost:8093> [port number can be anything]

For Admin Client,

<dependency>

<groupId>de.codecentric</groupId>

<artifactId>spring-boot-admin-starter-client</artifactId>

</dependency>

In application.properties, add below key

spring.boot.admin.client.url=http://localhost:8093

Messaging Queues- ActiveMQ:

Apache **ActiveMQ** is an open-source **message broker** that supports **Java Message Service (JMS)** and allows **asynchronous communication** between microservices using queues or topics.

**Why ActiveMQ in Microservices?**

In a microservices architecture, services often need to **communicate without being tightly coupled**. ActiveMQ helps by:

* **Decoupling** services (no direct dependency)
* Supporting **asynchronous messaging**
* Improving **resilience and scalability**

| **Concept** | **Description** |
| --- | --- |
| **Producer** | Sends messages to a queue/topic |
| **Consumer** | Receives messages from the queue/topic |
| **Queue (Point-to-Point)** | One consumer receives a message |
| **Topic (Publish-Subscribe)** | Multiple consumers receive a copy of the message |
| **Broker** | The ActiveMQ server that routes messages |

**How it works in Microservices**

1. **Producer Microservice** sends a message (e.g., order created).
2. **ActiveMQ Broker** stores and routes the message.
3. **Consumer Microservice** listens and processes the message (e.g., inventory service updates stock).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Column** | **Meaning** | | --- | --- | | **Name** | Name of the queue. Example: order.queue and ActiveMQ.DLQ. | | **Number Of Pending Messages** | Messages in the queue that **haven’t been consumed** yet. | | **Number Of Consumers** | Number of **active consumers** (listeners) subscribed to the queue. | | **Messages Enqueued** | Total number of messages **sent** to the queue. | | **Messages Dequeued** | Total number of messages **consumed** from the queue. | | **Views** | Options to view active producers and consumers (via Atom/RSS feeds). | | **Operations** | Actions you can take: Send To, Purge, Delete, or Pause the queue. | |

ActiveMQ.DLQ is the default **Dead Letter Queue (DLQ)** where **undeliverable messages** go (e.g., message expires, or no consumer is available).

**Example**: Add below dependency and properties in both Producer and Consumer applications.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-activemq</artifactId>

</dependency>

* Producer application uses JmsTemplate to send the messages .
* Consumer application uses @JmsListener to listen the message from the queue and process the message.
* ActiveMQ should download and running in the Local with port.

<http://localhost:8161/admin>

It helps monitor and manage the state of queues, topics, producers, and consumers.

**RabbitMQ:**

**RabbitMQ** is a **message broker** that enables communication between microservices using **asynchronous messaging**. It's built on the **AMQP (Advanced Message Queuing Protocol)** standard and is widely used for **decoupling** services, ensuring **scalability**, **resilience**, and **load balancing**.

| **Feature** | **Purpose** |
| --- | --- |
| **Decoupling** | Services communicate indirectly, reducing dependencies. |
| **Asynchronous Communication** | Services don’t need to wait for each other’s responses. |
| **Load Balancing** | Messages can be consumed by multiple instances of a service. |
| **Reliability** | Messages can be persisted and retried on failure. |
| **Scalability** | Producers and consumers can scale independently. |

**[Order Service] --(send)--> [RabbitMQ] --(receive)--> [Inventory Service]**

We use below dependency to work with RabbitMQ.

<!-- RabbitMQ Starter -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-amqp</artifactId>

</dependency>

Application.properties file

spring:

rabbitmq:

host: localhost

port: 5672

username: guest

password: guest

* Producer application uses RabbitTemplate to send the messages .
* Consumer application uses @ RabbitListener to listen the message from the queue and process the message.

How to download RabbitMQ:

* RabbitMQ is written in Erlang, so **you must install Erlang first**.
* Download the latest RabbitMQ installer:  
  👉 <https://www.rabbitmq.com/install-windows.html>
* Enable management dashboard using cmd- *rabbitmq-plugins enable rabbitmq\_management*
* Finally, start/stop the RabbitMQ server:

net start RabbitMQ

net stop RabbitMQ