Spring cloud

Spring Cloud is a collection of projects like load balancing, service discovery, circuit breakers, routing, micro-proxy, etc will be given by Spring Cloud. So spring Cloud basically provides some of the common tools and techniques and projects to quickly develop some common patterns of the microservices.

**Spring Cloud Features**

Some of the major features that are available in Spring Cloud are listed below

* Distributed/versioned configuration
* Service registration and discovery
* Routing
* Service-to-service calls
* Load balancing
* Circuit Breakers
* Distributed messaging

**Key Features and Components**

1. **Service Discovery and Registration**  
   Spring Cloud offers automatic service discovery and registration, enabling services to locate and communicate with each other dynamically. It integrates seamlessly with service registries like Netflix Eureka.
2. **Load Balancing**  
   Load balancing ensures the even distribution of requests across multiple instances of a service. Spring Cloud’s built-in load balancing capabilities intelligently route requests, optimizing resource utilization and improving performance.
3. **Circuit Breakers**  
   Spring Cloud’s circuit breaker pattern provides fault tolerance mechanisms that enable graceful degradation, reducing the impact of failing services on the overall system.
4. **Distributed Configuration Management**  
   Managing configurations across multiple microservices can be challenging. Spring Cloud’s distributed configuration management allows you to externalize configurations, centralize management, and dynamically update settings without service restarts.
5. **Distributed Tracing**  
   Troubleshooting and monitoring distributed systems can be complex. Spring Cloud’s distributed tracing capabilities, with tools like Zipkin or Jaeger, provide end-to-end visibility into request flows, enabling performance monitoring, debugging, and optimization.
6. **API Gateway**  
   Spring Cloud’s API gateway acts as a central entry point for microservices, handling request routing, filtering, and security. It simplifies the client-facing interface, provides security controls, and offloads common cross-cutting concerns.

## Problem: Why Do We Need Circuit Breakers?

In a **microservices architecture**, services often communicate with each other over the network (usually via REST APIs). But what happens if:

* A downstream service is **slow or unresponsive**?
* Calls to that service start to **time out**?
* These timeouts **consume system resources**, like threads or connections?

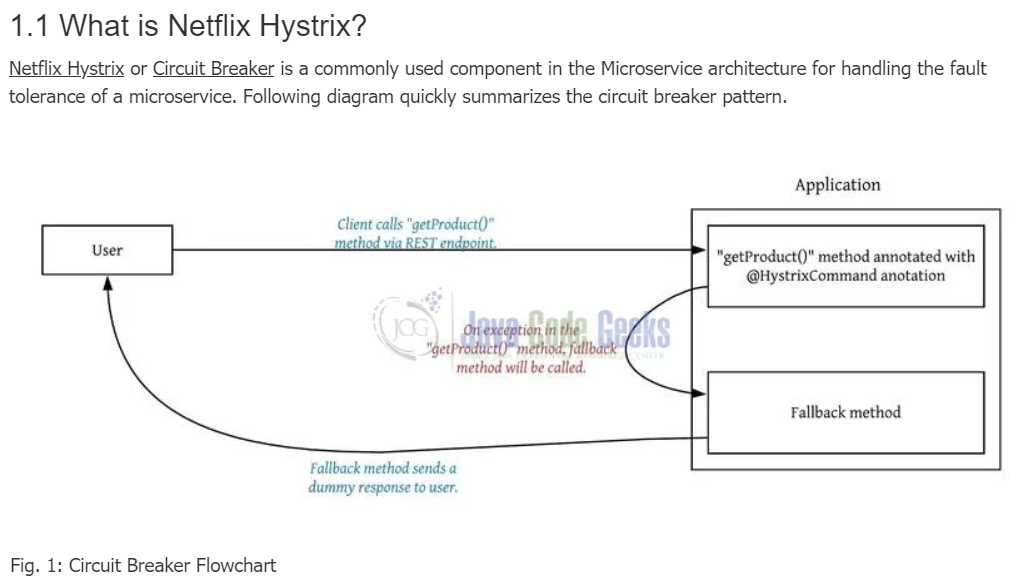
This can cause **cascading failures** and bring the whole system down.

## Solution: Circuit Breaker Pattern

The **circuit breaker pattern** is a design pattern used to detect failures and encapsulate the logic of preventing a failure from constantly recurring during maintenance, temporary external system failure, or   
  
If we design our systems on microservice based architecture, we will generally develop many Microservices and those will interact with each other heavily in achieving certain business goals. Now, all of us can assume that this will give expected result if all the services are up and running and response time of each service is satisfactory.

Now what will happen if any service, of the current Eco system, has some issue and stopped servicing the requests. It will result in timeouts/exception and the whole Eco system will get unstable due to this single point of failure.

Here circuit breaker pattern comes handy and it redirects traffic to a fall back path once it sees any such scenario. Also it monitors the defective service closely and restore the traffic once the service came back to normalcy.  
So circuit breaker is a kind of a wrapper of the method which is doing the service call and it monitors the service health and once it gets some issue, the circuit breaker trips and all further calls goto the circuit breaker fall back and finally restores automatically once the service came back !! That’s cool right?



## What is Netflix Hystrix?

[Netflix Hystrix](https://github.com/Netflix/Hystrix) is a library that implements the circuit breaker pattern.

It provides:

* **Circuit breaking**
* **Fallbacks**
* **Timeouts**
* **Thread isolation**
* **Request caching and collapsing**
* **Monitoring via Hystrix Dashboard**

In Spring Boot, Spring Cloud Netflix can be used to build the microservices architecture with the help of various components. In Spring Cloud Netflix, Hystrix is one of the components, and it can be used to stop cascading failures and enable resilience in complex distributed systems. It is a latency and fault tolerance library designed to isolate points of access to remote systems.

**Key Terminologies:**

* **Circuit Breaker:** In Spring Cloud Netflix, Hystrix can use the circuit breaker pattern to stop cascading failures in the microservices environment. When failures occur, the circuit breaker calling the failing services is automatically redirected to the fallback mechanism.
* **Fallback:** The fallback Mechanism can be used to execute a call when a service call fails, and it provides the graceful degradation of the service.
* **Command:** Hystrix command can be used to represent a potential failure point in the system or microservice.
* **Matrix:** Matrix can be used to provide the monitoring and metrics to track the health and performance of the microservices.

Why Spring cloud is required?

When developing distributed microservices with Spring Boot we face the following issues-

* **Complexity associated with distributed systems-**  
  This overhead includes network issues, Latency overhead, Bandwidth issues, security issues.
* **Service Discovery-**  
  Service discovery tools manage how processes and services in a cluster can find and talk to one another. It involves a directory of services, registering services in that directory, and then being able to lookup and connect to services in that directory.
* **Redundancy-**  
  Redundancy issues in distributed systems.
* **Loadbalancing-**  
  Load balancing improves the distribution of workloads across multiple computing resources, such as computers, a computer cluster, network links, central processing units, or disk drives.
* **Performance issues-**  
  Performance issues due to various operational overheads.
* **Deployment complexities-** Requirement of Devops skills.

# Spring Cloud - Load Balancer

In a distributed environment, services need to communicate with each other. The communication can either happen synchronously or asynchronously. Now, when a service communicates synchronously, it is better for those services to load balance the request among workers so that a single worker does not get overwhelmed. There are two ways to load balance the request

1. **Client-Side Load Balancing:**
   * The client application itself is responsible for selecting the appropriate service instance to send a request to.
   * This typically involves integrating with a service discovery mechanism (e.g., Eureka) to obtain a list of available service instances.
   * Spring Cloud LoadBalancer: is a key component in the Spring Cloud ecosystem that facilitates client-side load balancing for Spring Boot applications. It offers various load-balancing algorithms (e.g., Round Robin, Random) and integrates seamlessly with Feign clients.

**2) Server-Side Load Balancing:**

* + An external infrastructure layer (e.g., a dedicated load balancer like Nginx, or an API Gateway like Spring Cloud Gateway or Netflix Zuul) handles the distribution of requests to service instances.
  + This layer receives requests from clients and routes them based on configured rules, health checks, and load metrics.
  + Spring Cloud Gateway: can act as an API Gateway and perform load balancing by routing requests to backend services, often leveraging Spring Cloud LoadBalancer internally.

**Spring Cloud load balancer (SLB) and Netflix Ribbon** are two well-known client-side load balancer which are used to handle such situation.

