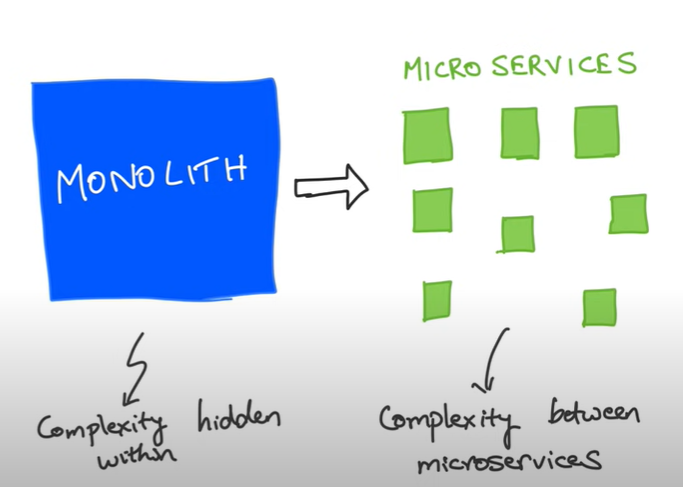
## Agenda

* 3 microservices services
* Make them communicate
* Spring cloud is one way to create microservices



## Urls

<http://localhost:8081/catalog/5>

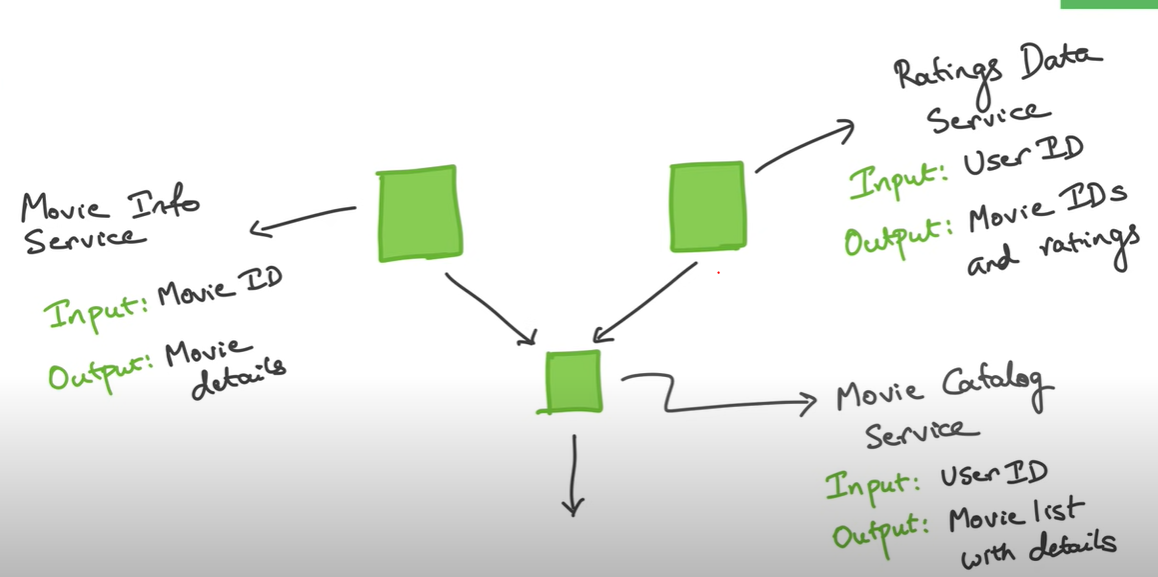
<http://localhost:8082/movies/foo>

http://localhost:8083/ratingsdata/foo

<http://localhost:8083/ratingsdata/users/123>

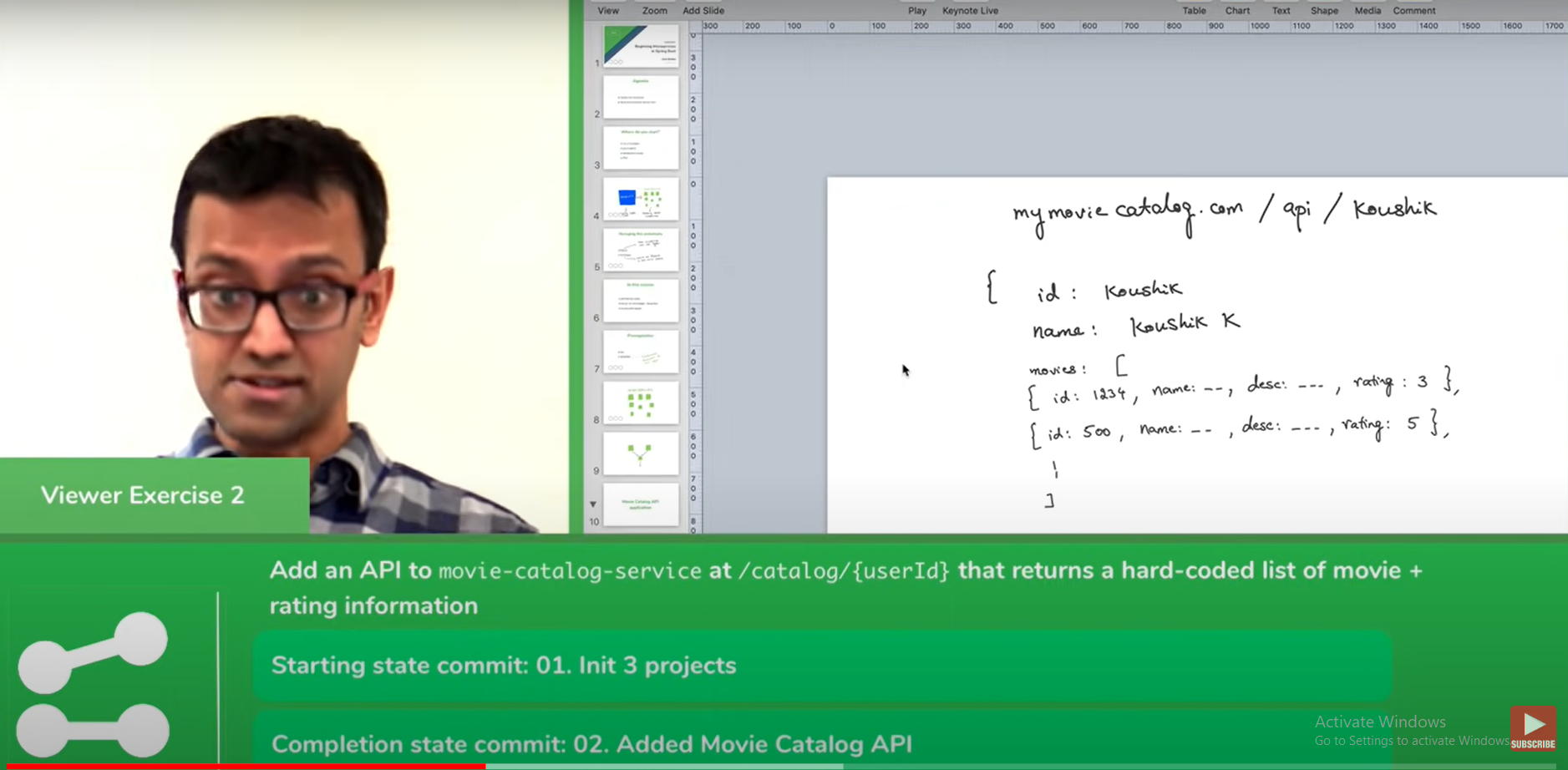
## Microservice vs web services

## Movie catalog application



## Create 3 services with bare minimum dependency. So create 3 services with web dependency

## start with catalog service



## RestTemplate to make API calls

* Can use Feign instead refer udemy 161
* Web client (requires reactive programming) => asynchronous

Note :

**While doing unmarshaling make sure that class have no args constructor**

## how do we create singleton instance of object? Here how do we create single instance of RestTemplate and share it everywhere..

@SpringBootApplication

public class MovieCatalogServiceApplication {

@Bean

public RestTemplate getRestTemplate() {

return new RestTemplate();

}

public static void main(String[] args) {

SpringApplication.run(MovieCatalogServiceApplication.class, args);

}

}

*@Autowired*

private RestTemplate restTemplate;

@Bean create singleton instance it is producer, @Autowired is consumer

## Playing with WebClient instead of RestTemplate

Add dependency

<dependency>

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

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

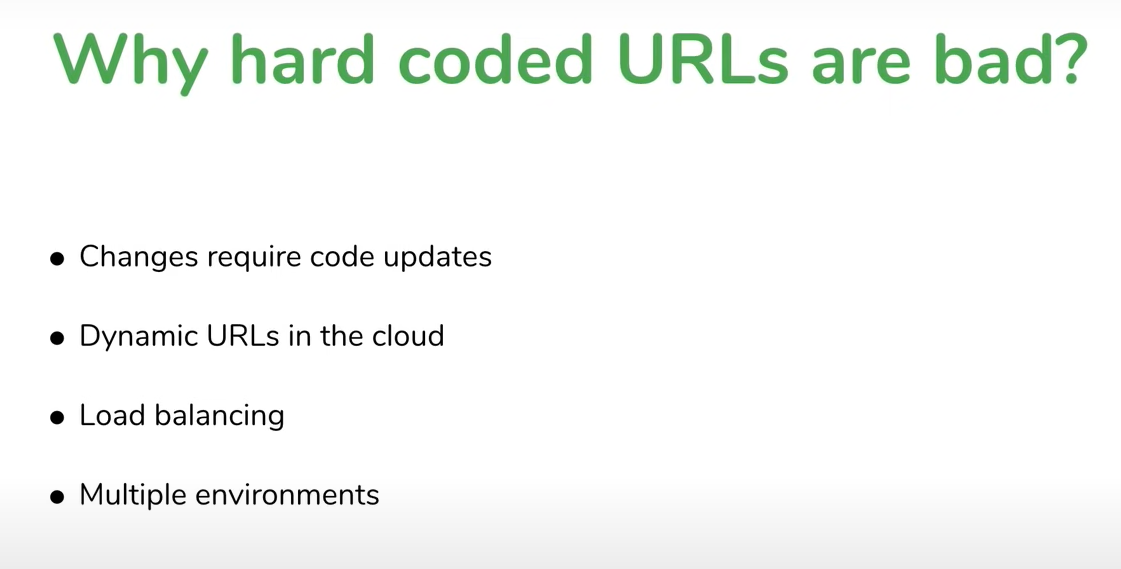
</dependency>

<Refer 13>

## Do not return a list from controller rather wrap it inside object and then return.

<Refer 14,15>

## What wrong are we doing?



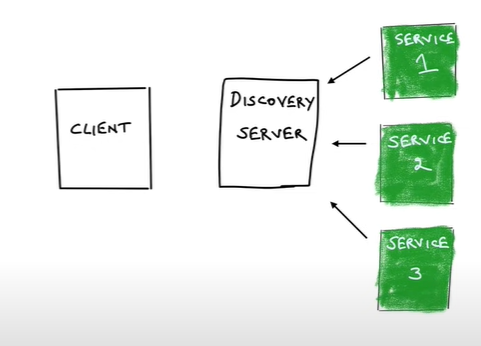
Solution : Discovery server

* https://www.baeldung.com/cs/service-discovery-microservices

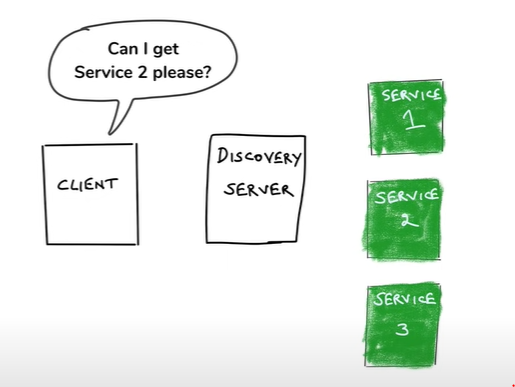
Refer for more information

1. Client side discovery server

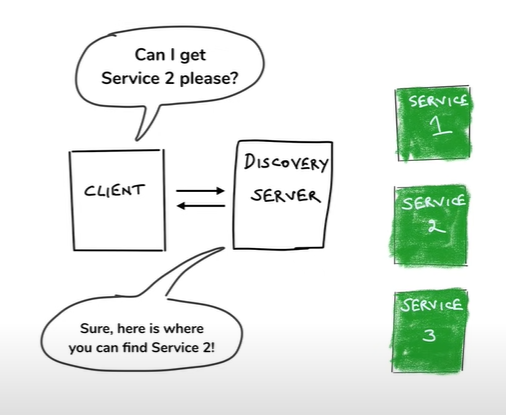
* All the three servers register with discovery server



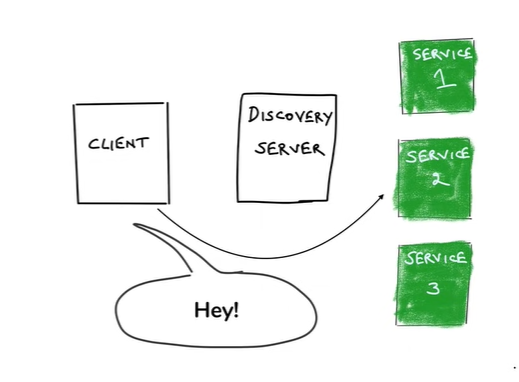
* When client wants service2 it ask to discovery server.



* Discovery server gives address to client back.

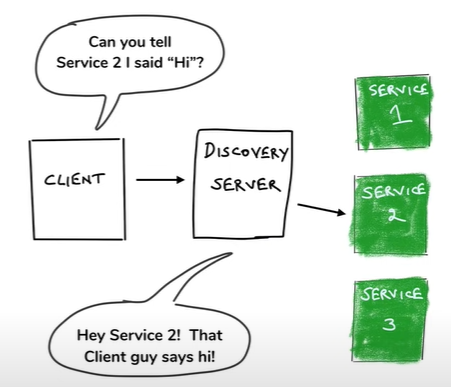


* Then client based on that address accesses service2



Disadvantages: its little more chatty

1. Server side discovery server



Client passes message directly to a medium which it suppose to pass service2. Then that medium itself pass that message to service2. So here unlike client side discovery server there is no extra hop.

**\*\* Spring Cloud uses CLIENT SIDE DISCOVERY SERVICE \*\***

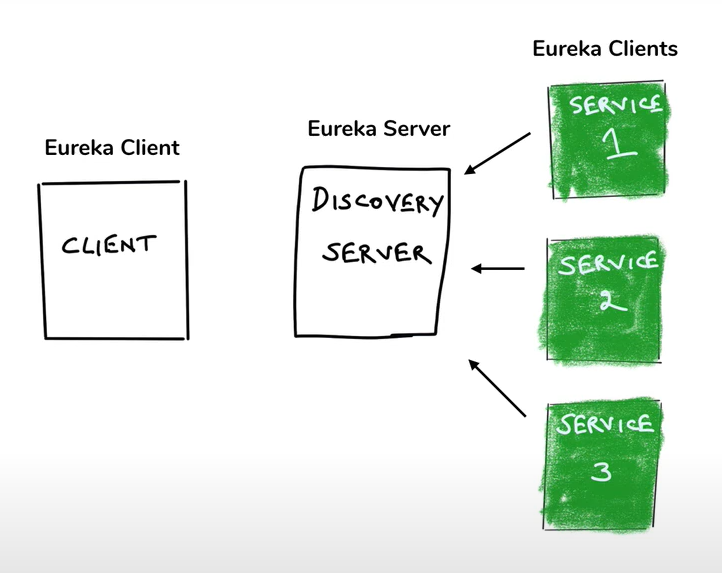
## Technology to implement discovery service

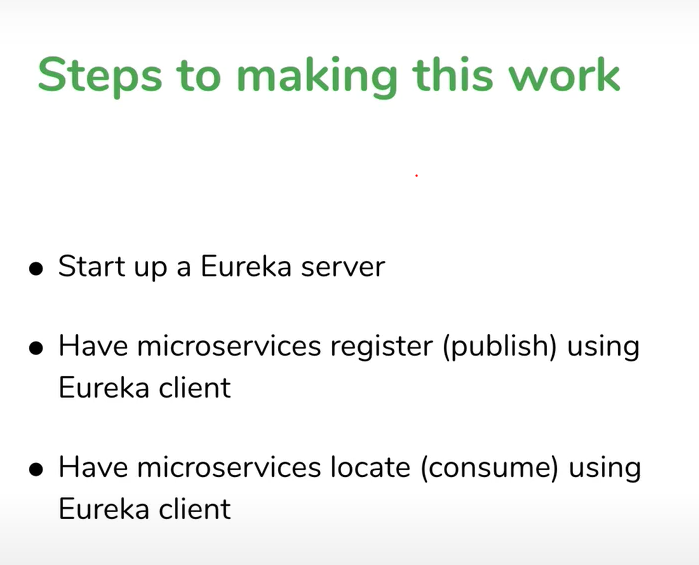
Eureka

Netflix OSS

Eureka is one of projects that made them open source

Like Eureka, Ribbon, Hysterix, Zuul, …….





## Creating discovery server

Create new service with Eureka server as dependency

Add @EnableEurekaServer in main

Registering microservices with discovery server (eureka)

Add dependency

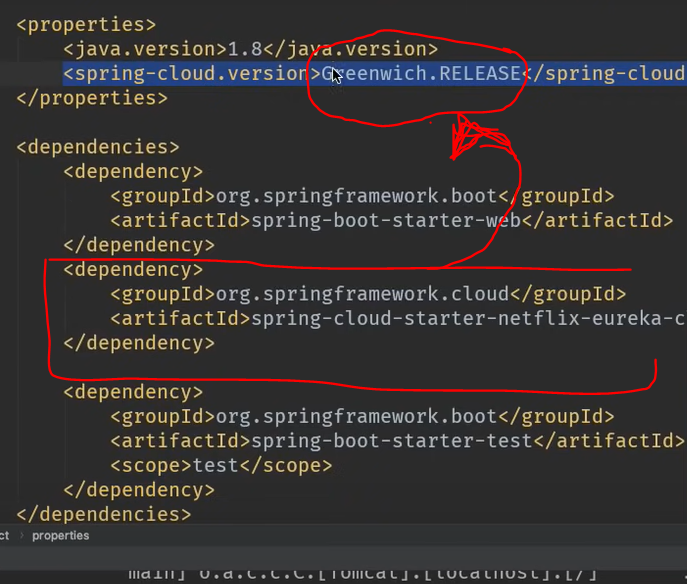
<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

<version>4.1.1</version>

</dependency>



## To avoid hardcoded url’s

*@SpringBootApplication*

public class MovieCatalogServiceApplication {

*@Bean*

*@LoadBalanced*

public RestTemplate getRestTemplate() {

return new RestTemplate();

}

public static void main(String[] args) {

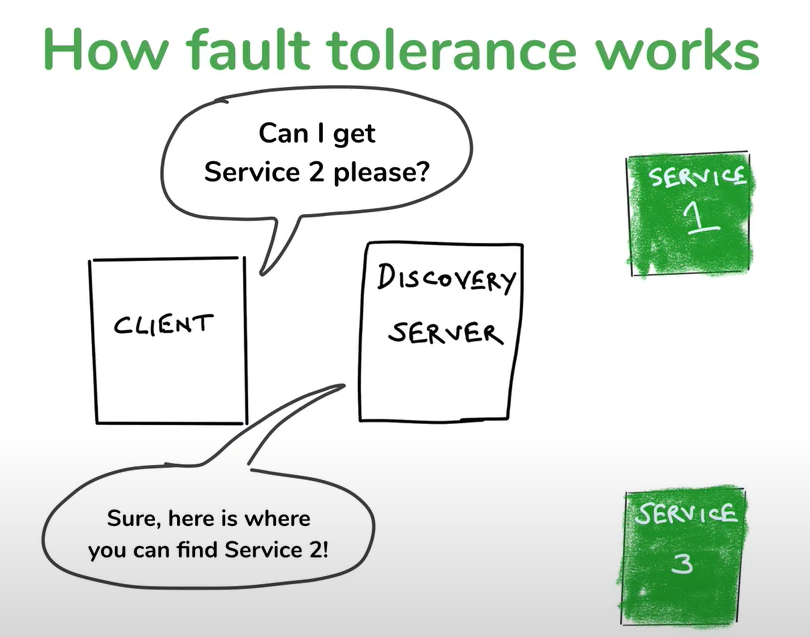
SpringApplication.*run*(MovieCatalogServiceApplication.class, args);

}

}

## How fault tolerance works?

Service 2 is down

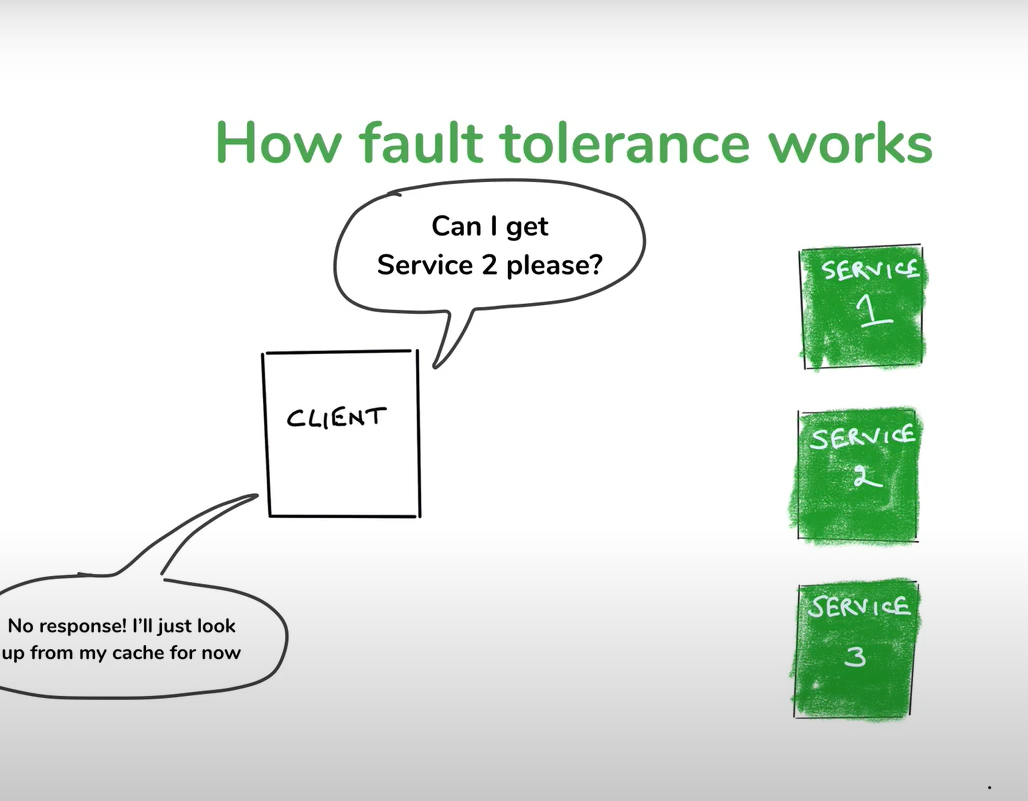


Solution is “Heart Beat” :- Services will ping discovery client after every interval of time that I am alive. So that discovery server knows service 2 is down.

## What if discovery server goes down?

* If client don’t get response from server it goes to cache get response

**\*\* All these things happening in background with just one annotation @LoadBalancer \*\***



## What if microservice goes down?

* Solution – Create multiple instances. So that even if one instance goes down then it goes to another service and system will not down.

## What if microservice is slow?

Much much bigger problem.

Why?

A diagram of a service

Description automatically generated

Here in this case the Movie DB is slow, so because of that movie-info-service is also slow and ultimately the catalog-service. **But after some time we find that rating-data-service is slow though it is not slow in actual**. Why?

A diagram of a service

Description automatically generated

Why?

* Answer : Threads

How thread in Web server?

* When request comes to server web server creates thread for that request.

A diagram of a web server

Description automatically generated

It does its work and it removes the thread from thread pool

A white board with blue writing

Description automatically generated

But what happens if thread is taking longer and request keeps coming. So multiple threads keeps on creating. And the time will come when number of threads exceed the limit. And suppose request for another service comes which is faster, then this request will not execute as limit of thread already exceeded. That’s why because one slow service it might feels that faster service is also getting slow.

* Solution 1

We might give timeout to each service. If suppose time exceeds certain limit for a service then that thread will be terminated.

But this solution is not full proof. Suppose if we timeout for thread as 3 milliseconds. And suppose new thread is getting created after every 1 millisecond. So in this case thread blocking situation may occur.

* Solution 2

Circuit breaker pattern

## What if microservice goes down?

* Solution – Create multiple instances. So that even if one instance goes down then it goes to another service and system will not down.

## What if microservice is slow?

Much much bigger problem.

Why?

A diagram of a service

Description automatically generated

Here in this case the Movie DB is slow, so because of that movie-info-service is also slow and ultimately the catalog-service. **But after some time we find that rating-data-service is slow though it is not slow in actual**. Why?

A diagram of a service

Description automatically generated

Why?

* Answer : Threads

How thread in Web server?

* When request comes to server web server creates thread for that request.

A diagram of a web server

Description automatically generated

It does its work and it removes the thread from thread pool

A white board with blue writing

Description automatically generated

But what happens if thread is taking longer and request keeps coming. So multiple threads keeps on creating. And the time will come when number of threads exceed the limit. And suppose request for another service comes which is faster, then this request will not execute as limit of thread already exceeded. That’s why because one slow service it might feels that faster service is also getting slow.

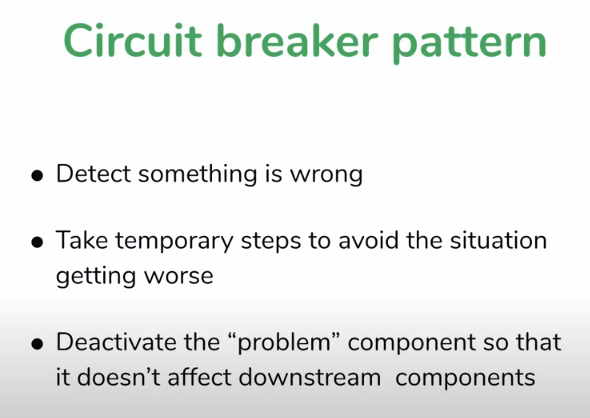
* Solution 1

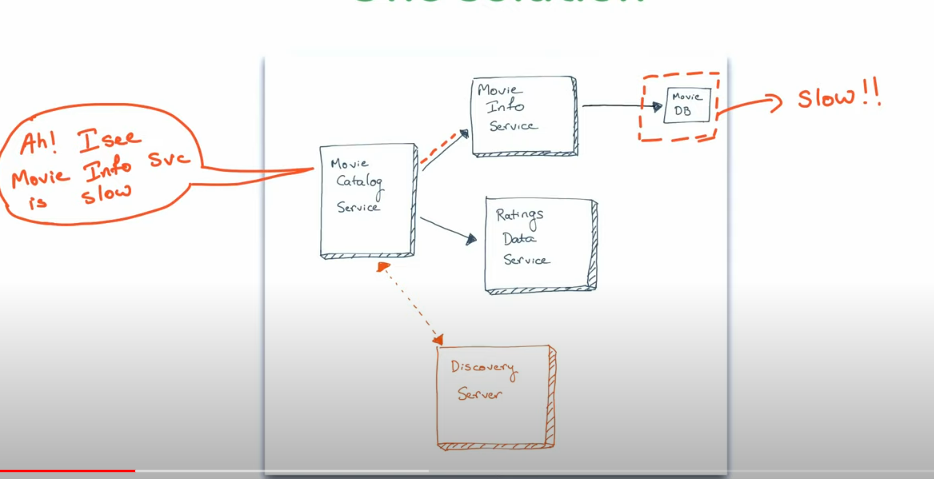
We might give timeout to each service. If suppose time exceeds certain limit for a service then that thread will be terminated.

But this solution is not full proof. Suppose if we timeout for thread as 3 milliseconds. And suppose new thread is getting created after every 1 millisecond. So in this case thread blocking situation may occur.

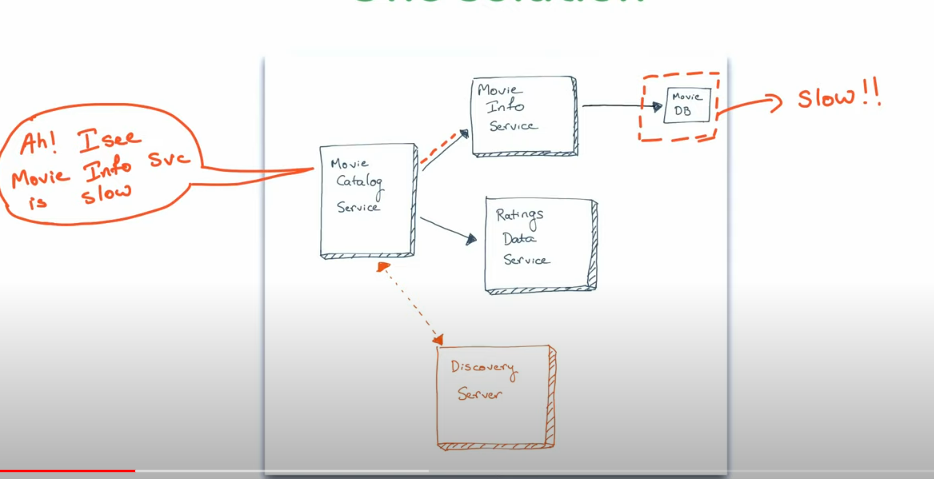
* Solution 2

Circuit breaker pattern



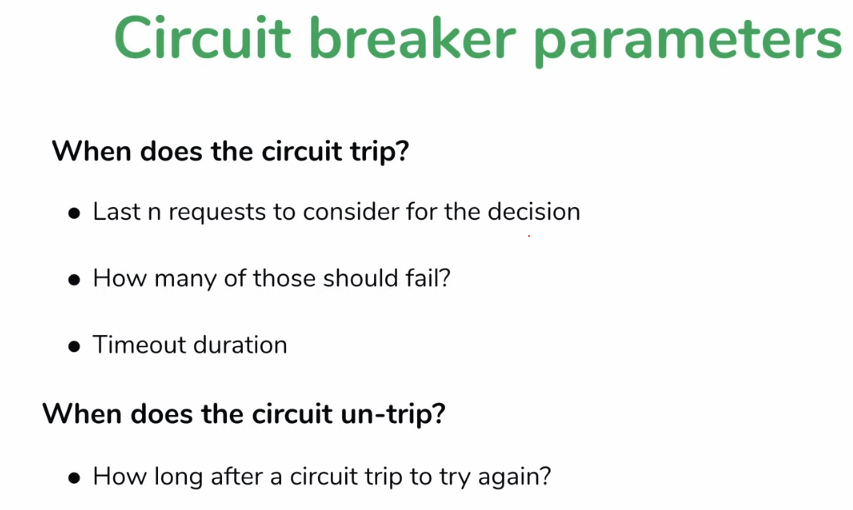


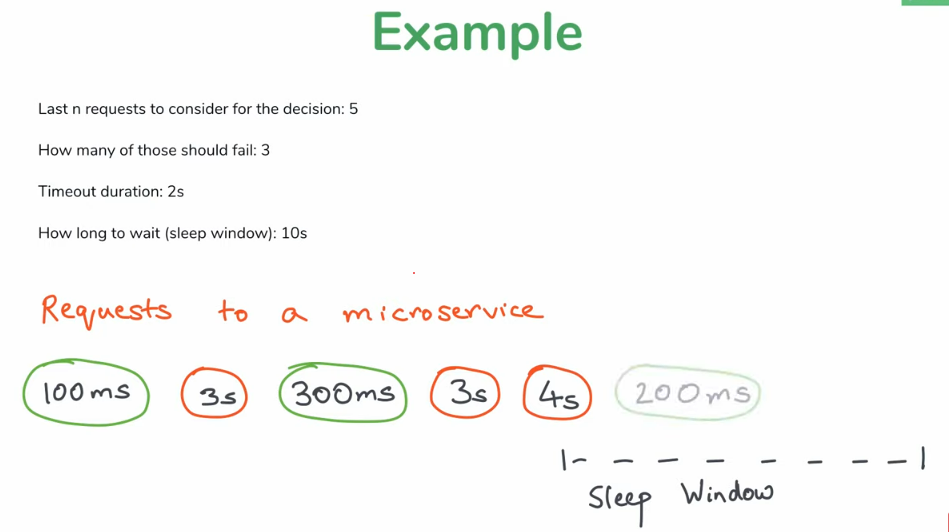
* Here movie-info-service is slow. Which affect the performance.
* As movie-catalog-service calling movie-info-service and rating-service, so this time catalog-service is going to be smarter. It sends request to info-service but if it don’t get response it will sending request to info-service for a while. So that number threads should not exceeds its limit. And whenever catalog-service feels that info-service gives response back and earlier thread get vanished from thread pool it again resumes sending request to info-service.
* So this like breaking circuit if something goes wrong so that whole system wont collapse.



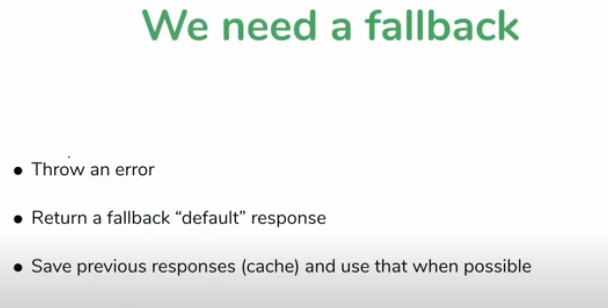
* Where could it be used?
  + If microservice is calling another microservice or multiple microservices, there we can use it.

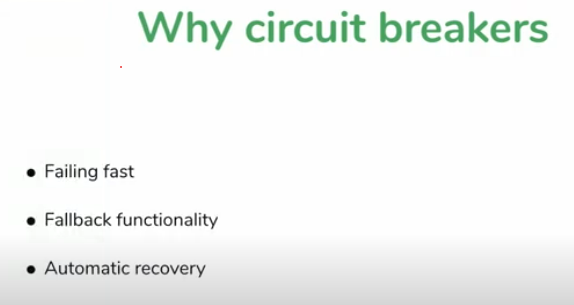
Circuit breaker parameters:





* What if circuit breaks and request keep coming?
  + Solution : Fallback



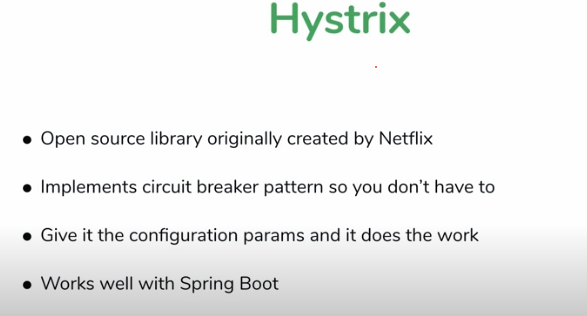


Failing fast : we fail system early if know that it is going to fail anyway

Ex: if already know that request is taking longer to complete then thread pool is going get full. We fail it early.

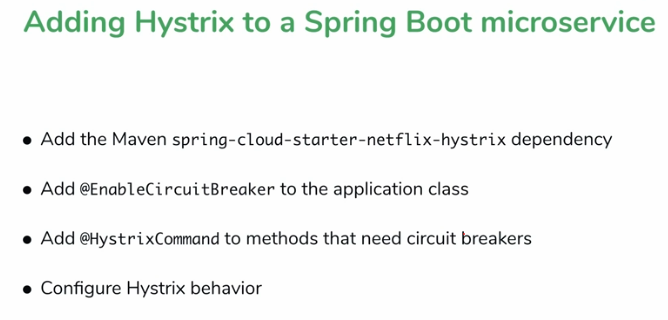
## Hysterix:

Use to implement circuit breaking.



## Note : Hystrix is not in development now, so it will get updated.

Setting up hystrix



* **Resilience4j**
  + <https://resilience4j.readme.io/docs/getting-started>
  + Dependencies
    - Actuator

<dependency>

<groupId>io.github.resilience4j</groupId>

<artifactId>resilience4j-spring-boot2</artifactId>

<version>2.1.0</version>

</dependency>

<dependency>

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

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

</dependency>

* Notes

1. Circuit breaker

The Circuit Breaker pattern is a design pattern used to detect failures and encapsulate the logic of preventing a failure from constantly reoccurring, allowing a system to recover. It monitors the system, tracks failures, and acts by temporarily stopping requests to the failing service (**when a threshold is crossed**) until it recovers.

**Overview of Circuit Breaker States**

1. **Closed State**:
   * All requests pass through as normal.
   * Failures are counted within a "sliding window" of either time or number of requests.
   * If the failure rate exceeds a configured threshold, the Circuit Breaker transitions to the **Open State**.
2. **Open State**:
   * Requests fail immediately without being executed (they're blocked).
   * The system waits for a configured amount of time (called waitDurationInOpenState).
   * After the wait time, it transitions to the **Half-Open State**.
3. **Half-Open State**:
   * Only a limited number of test requests are allowed through.
   * If the test requests succeed, the Circuit Breaker transitions back to **Closed State**.
   * If the test requests fail, the Circuit Breaker returns to the **Open State**.

**Example Scenario**

Let's say you have a microservice that calls another service to retrieve movie details. We will use Resilience4j's Circuit Breaker to protect against failures when calling this movie service.

**Initial Setup**

You configure a Circuit Breaker for the getMovieInfo service call:

* **Failure Rate Threshold**: 50% (if more than 50% of the requests fail, the circuit will open).
* **Sliding Window Size**: 10 requests.
* **Wait Duration in Open State**: 10 seconds (the circuit will remain open for 10 seconds before transitioning to half-open).
* **Permitted Calls in Half-Open State**: 2 (only two test requests are allowed when the circuit is half-open).

*Copy code*

*resilience4j.circuitbreaker:*

*instances:*

*movieInfoService:*

*slidingWindowType: COUNT\_BASED*

*slidingWindowSize: 10*

*failureRateThreshold: 50*

*waitDurationInOpenState: 10s*

*permittedNumberOfCallsInHalfOpenState: 2*

**1. Closed State (Normal Operation)**

Initially, the Circuit Breaker is in the **Closed State**:

* All requests to the movie-info-service go through.
* The Circuit Breaker monitors these requests and tracks failures.

Example: You make 10 requests, and 2 of them fail. This is a 20% failure rate, which is below the 50% threshold, so the Circuit Breaker remains **Closed**.

plaintext

Copy code

Request 1: Success

Request 2: Success

Request 3: Fail

Request 4: Success

...

Request 10: Fail

Failure Rate = 20% (Below Threshold)

Circuit Breaker State = Closed

**2. Transition to Open State**

Now, let's say the movie service starts having issues, and 6 out of the next 10 requests fail. The failure rate is now 60%, which is above the 50% threshold. The Circuit Breaker transitions to the **Open State**.

plaintext

Copy code

Request 11: Fail

Request 12: Fail

...

Request 20: Fail

Failure Rate = 60% (Above Threshold)

Circuit Breaker State = Open

**3. Open State (Blocking Requests)**

In the **Open State**, all incoming requests to the movie-info-service are immediately failed without being sent to the movie service. This prevents overloading the failing service and helps the system maintain stability.

* For the next 10 seconds (the configured waitDurationInOpenState), every request to getMovieInfo will instantly return the fallback response.

plaintext

Copy code

Request 21: Blocked (Fallback Triggered)

Request 22: Blocked (Fallback Triggered)

...

Circuit Breaker State = Open (Waiting for 10 seconds)

**4. Transition to Half-Open State (Testing)**

After 10 seconds, the Circuit Breaker transitions to the **Half-Open State**. In this state, only a few requests (2 in our configuration) are allowed through as test requests.

* If these test requests succeed, the Circuit Breaker assumes the service has recovered and transitions back to **Closed State**.
* If these test requests fail, the Circuit Breaker goes back to **Open State** for another waiting period.

**Test Scenario 1: Successful Test Requests**

plaintext

Copy code

Request 31: Allowed (Success)

Request 32: Allowed (Success)

Circuit Breaker State = Closed (Service Recovered)

Since both test requests succeeded, the Circuit Breaker closes again, and normal traffic resumes.

**Test Scenario 2: Failed Test Requests**

plaintext

Copy code

Request 31: Allowed (Fail)

Request 32: Allowed (Fail)

Circuit Breaker State = Open (Service Still Failing)

In this case, both test requests failed, so the Circuit Breaker transitions back to **Open State**, and the cycle repeats.

**Summary of States:**

* **Closed**: Everything works normally. Failures are tracked.
* **Open**: All requests are blocked. Circuit breaker waits before trying again.
* **Half-Open**: Some requests are allowed through to check if the service has recovered. Based on the outcome, the Circuit Breaker will either close (recover) or open again (failure).

**Code Example**

Here’s how this could look in code using Resilience4j's Circuit Breaker:

java

Copy code

@Service

public class MovieInfoService {

@CircuitBreaker(name = "movieInfoService", fallbackMethod = "fallbackGetMovieInfo")

public Movie getMovieInfo(String movieId) {

// Calling the remote movie service

return restTemplate.getForObject("http://movie-service/movies/" + movieId, Movie.class);

}

// Fallback method triggered when Circuit Breaker is open

public Movie fallbackGetMovieInfo(String movieId, Throwable t) {

return new Movie("No Movie", "Fallback Description");

}

}

In this code:

* The @CircuitBreaker annotation protects the getMovieInfo method.
* If the movie service fails repeatedly, the Circuit Breaker will open, and fallbackGetMovieInfo will be called.
* The circuit will transition to half-open after a wait period to see if the service has recovered.

**Real-World Use Cases for Circuit Breakers**

* **Database Connectivity Issues**: If your database becomes unreachable, the Circuit Breaker will open and block queries to prevent overwhelming the database.
* **External API Failures**: When relying on third-party APIs, the Circuit Breaker helps protect your system from cascading failures by preventing repeated calls to a failing API.
* **Network Latency Issues**: If a remote service is slow, the Circuit Breaker can help by limiting the number of requests sent to it, preventing bottlenecks.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Scenario Recap:**

1. **Sliding Window:** 5 requests.
2. **Last 3 requests** are taking a long time to complete when the circuit breaker transitions to **Open** state.
3. **Half-Open State:** After the **Open** state timeout period, the circuit breaker transitions to **Half-Open** and allows a few test requests (let's say 2).
4. These **2 test requests** also take a long time to complete, so the circuit breaker transitions back to the **Open** state.
5. Meanwhile, the previous long-running requests are still not completed, and this cycle repeats.

This situation can indeed lead to a **thread exhaustion** issue or **resource depletion**, potentially causing the entire system to degrade or even crash.

**Key Points to Consider:**

1. **Accumulation of Slow Requests:**
   * The system might accumulate slow-running requests if they are not completing, and as new requests continue to pile up in **Half-Open** or **Open** states, the thread pool can fill up with long-running threads. This would eventually lead to thread starvation.
2. **System Resource Overload:**
   * Each long-running request consumes system resources (threads, memory, CPU). If no requests are completing and the breaker keeps allowing test requests in **Half-Open**, the system could become overloaded with pending requests.

**Possible Mitigations:**

**1. Timeout Mechanism**

* **Ensure Proper Timeout Settings:** Configure a **timeout** for requests using Resilience4j's TimeLimiter or other timeout mechanisms. If a request exceeds a specific threshold, it should be automatically canceled or aborted. This prevents long-running requests from consuming threads indefinitely.
* **TimeLimiter Example:**

yaml

Copy code

resilience4j.timelimiter:

instances:

myService:

timeoutDuration: 2s # Requests that take longer than 2 seconds will be timed out.

**2. Thread Pool and Bulkhead Patterns**

* **Bulkhead Pattern:** Use a bulkhead pattern to isolate resources for specific services, preventing one slow service from exhausting all system resources.
* **Thread Pool Isolation:** Each circuit breaker instance can have its own thread pool. By isolating each service's calls into their own thread pools, you prevent the entire system from being affected by long-running requests.

Example in YAML:

yaml

Copy code

resilience4j.bulkhead:

instances:

myService:

maxConcurrentCalls: 5

maxWaitDuration: 100ms

**3. Failure Rate Consideration**

* **Increase Failure Threshold:** Consider fine-tuning your failure rate or sliding window thresholds so that they better reflect your actual system's tolerance for failures.
* This helps prevent the system from prematurely transitioning to **Open** just because a few requests are slow.

**4. Control Circuit Breaker Test Requests**

* **Limit Half-Open Test Requests:** Set the number of test requests allowed during the **Half-Open** state to a manageable level to prevent the system from overwhelming itself.
* Example in YAML:

yaml

Copy code

resilience4j.circuitbreaker:

instances:

myService:

slidingWindowSize: 5

permittedNumberOfCallsInHalfOpenState: 1

waitDurationInOpenState: 10s

**5. Monitoring and Alerts**

* **Proactive Monitoring:** Implement monitoring on your circuit breakers to detect potential problems early. Track metrics like the number of open circuits, slow requests, and the state of your thread pools. Set alerts to warn you when your system approaches dangerous thresholds.

**6. Rate Limiting**

* Implement rate limiting to control the flow of requests when the system is under heavy load. This prevents overloading the system with requests that it cannot handle in a timely manner.

**Conclusion:**

Yes, the scenario you're describing could lead to thread exhaustion and system failure if not properly mitigated. By implementing **timeouts, bulkheads, proper circuit breaker configurations, and monitoring**, you can prevent the system from becoming overloaded due to long-running requests and the repeated opening and closing of the circuit breaker.

**What is a Rate Limiter?**

A Rate Limiter controls the rate at which requests are allowed to execute. It restricts the number of calls to a service in a given time period. This helps prevent overwhelming the system or third-party services by limiting excessive traffic.

**Core Concepts of Rate Limiter**

1. **Permits per Time Period**: You specify how many requests are allowed within a certain time frame (e.g., 10 requests per second).
2. **Waiting Time**: If the number of requests exceeds the rate limit, requests can either be immediately rejected or made to wait for a certain time until a permit becomes available.
3. **Time Period**: The window in which the rate limiting is applied (e.g., per second, per minute).
4. **Failure Behavior**: If the limit is exceeded, you can decide how to handle the situation. You can either fail fast or queue up the requests for later processing.

**Example Code with Rate Limiter**

Let’s integrate a RateLimiter into your existing service:

java

Copy code

*@RateLimiter(name = "ratingRateLimit", fallbackMethod = "rateLimitFallback")*

*public UserRating getUserRating(String userId) {*

*return restTemplate.getForObject("http://movie-rating-service/ratingsdata/users/" + userId, UserRating.class);*

*}*

*public UserRating rateLimitFallback(String userId, Exception e) {*

*return new UserRating(List.of(new Rating("-1", -1)));*

*}*

**Explanation**

* **@RateLimiter(name = "ratingRateLimit")**: This annotation applies the Rate Limiter with the given configuration (ratingRateLimit is the name of the Rate Limiter).
* **rateLimitFallback**: If the rate limit is exceeded, this method will be called as a fallback.

**Configuration Example**

You can configure the RateLimiter in your application.yml or application.properties file like this:

yaml

Copy code

*resilience4j.ratelimiter:*

*instances:*

*ratingRateLimit:*

*limitForPeriod: 5*

*limitRefreshPeriod: 1s*

*timeoutDuration: 500ms*

In this example:

* limitForPeriod: 5 calls are allowed within a time window.
* limitRefreshPeriod: The window refreshes every 1 second.
* timeoutDuration: If a request cannot acquire a permit within 500ms, it fails and the fallback is triggered.

**When to Use Rate Limiter?**

* To prevent **overloading** downstream services or APIs.
* When working with **rate-limited third-party services** (e.g., APIs that have a strict usage quota).
* To **control access** in high-traffic situations and ensure service stability.

**1. limitForPeriod**

This parameter defines the **number of permitted calls** within each time period (known as the "refresh period"). It sets a hard limit on how many requests can be made.

* **Example**: If limitForPeriod: 5, this means **only 5 calls** can be made within a defined period (e.g., per second or minute).
* **Use case**: You set this limit to control how many calls the system can handle or how many calls are allowed by a third-party service, such as an API that may charge per request or throttle usage.

**2. limitRefreshPeriod**

The limitRefreshPeriod defines the **time window** in which the permitted calls (defined by limitForPeriod) are allowed.

* **Example**: If limitRefreshPeriod: 1s, the rate limiter allows up to 5 calls every second (assuming limitForPeriod is 5).
* **Use case**: You use this to reset the rate limit after a certain time interval. For example, you might want to allow 100 requests per minute or 10 requests per second, depending on your use case.

This period defines how frequently the system is "refilled" with tokens for calls.

**3. timeoutDuration**

This parameter defines the **maximum time** a request will wait to acquire a permit before failing or triggering the fallback method. If no permit becomes available within this time, the request will be rejected, and the fallback method (if defined) will be invoked.

* **Example**: If timeoutDuration: 500ms, when the rate limit is exceeded, a request will wait for a maximum of **500 milliseconds** for a permit before giving up and failing.
* **Use case**: This prevents requests from indefinitely waiting in line when the system is overloaded. It's helpful for quickly failing or redirecting users to an alternative service instead of making them wait for long periods.

**Putting it Together**

Here’s how these parameters work in practice:

yaml

Copy code

resilience4j.ratelimiter:

instances:

ratingRateLimit:

limitForPeriod: 5 # Allow 5 requests

limitRefreshPeriod: 1s # Within every 1 second window

timeoutDuration: 500ms # If no permit is available, wait for 500ms before failing

**Explanation:**

* **limitForPeriod** (5): You can make up to **5 requests**.
* **limitRefreshPeriod** (1s): After every **1 second**, the rate limiter resets, allowing another 5 requests.
* **timeoutDuration** (500ms): If the rate limit has been reached, subsequent requests will wait for **up to 500ms** to acquire a permit. If a permit doesn't become available within this time, the request fails, and the fallback method is triggered (if provided).

**Example Scenario**

Imagine you have a REST API that only allows up to 5 requests per second:

1. Within the first second, the first 5 requests will proceed as usual.
2. If a 6th request comes in during that same second:
   * The system will either immediately fail or wait for up to 500ms for a permit (if any of the previous 5 requests complete within that time).
   * If after 500ms, no permit becomes available, the fallback method is triggered (or the request fails).
3. After 1 second has passed, the limit resets, and another batch of 5 requests can be made.

This ensures that the system maintains a steady rate of requests and prevents overwhelming the service, while allowing for quick fallback if limits are exceeded.

**Practical Use Case**

Consider a service that calls an external API that has strict rate limits (e.g., 100 requests per minute). You can configure a RateLimiter to ensure your service never exceeds this limit, preventing potential errors from the third-party service and helping manage traffic.

## Bulk head