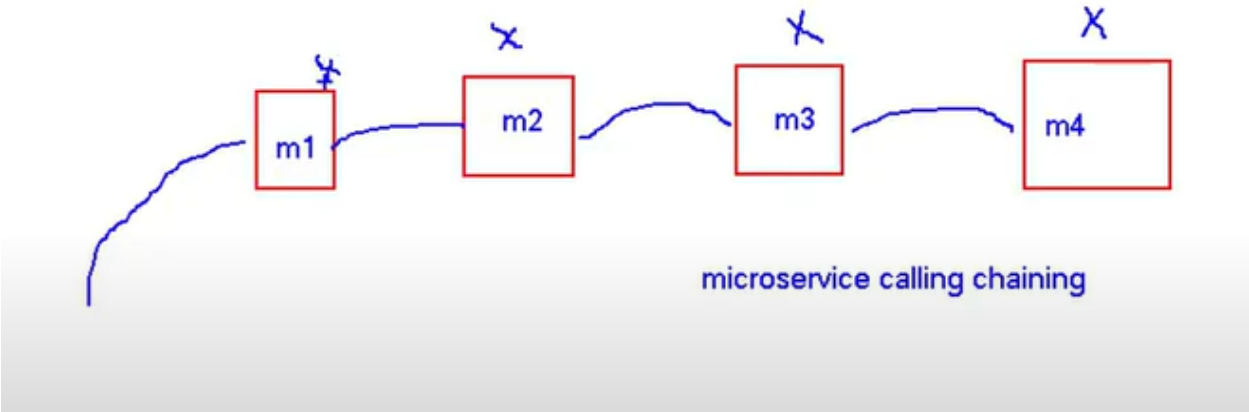
05-03-2025 CIRCUIT BREAKER

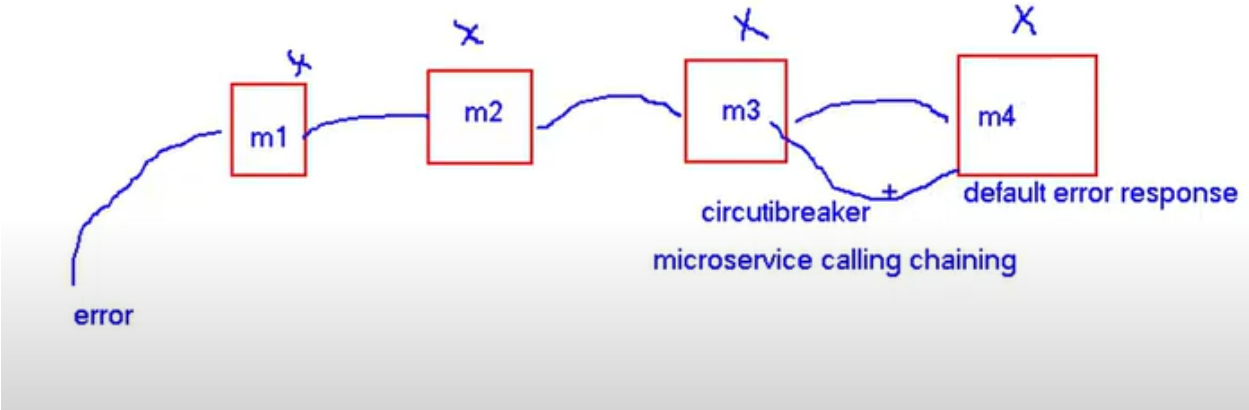
If any one of the microservice is down due to some technical reasons then all the dependent microservices will goes down automatically

For suppose if the 4th mc is down then the dependent 1 ,2,3 mc are also not working properly bcoz **micro service chaining**



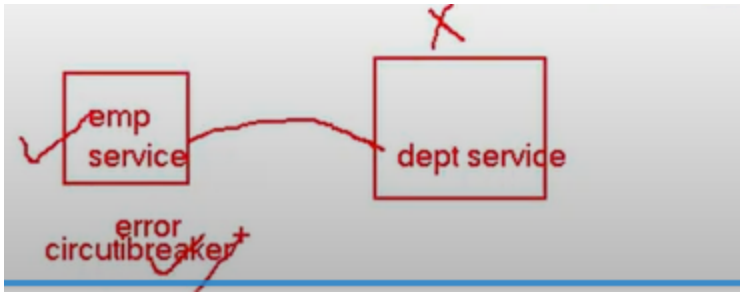
It leads to end user gets an error message

Then we should develop the circuit breaker in the 3rd micro service



Then we will get the default error msg from 4 to 3 instead of all microservices getting down

For suppose if the department micro service is gets down due to some technical reasons we don't want our emp service gets down….for that we need to provide the circuit breaker in the employee service



Then our circuit breaker is in **closed** state….so we can send the request and we can get the response

If the department service is down then the closed state will move to the open state

Then we cannot make and get the request and response from the department service bcoz it is in open state

If we mention 5 ms then mc will goes to half open state to open state

🔹 Circuit Breaker in Microservices

A **Circuit Breaker** is a **fault-tolera** **attern** used in microservices to **prevent cascading failures** when one service is down or responding slowly. It acts as a **safety switch** that stops repeated failed requests to a failing service.

🔍 Why Do We Need a Circuit Breaker?

In a microservices architecture, services communicate over the network. If a dependent **service is down or slow**, it can:

- Cause **timeouts** and **increase response times**.
- **Overload** the failing service with repeated requests.
- **Crash the entire system** due to excessive failures.

A **Circuit Breaker** prevents this by **detecting failures** and **stopping further requests** until the service recovers.

🔄 How Circuit Breaker Works?

It operates in **three states:**

1️⃣ Closed (Normal State)

- All requests are forwarded to the target service.
- If failures **exceed a threshold**, the circuit breaker **opens.**

Means it is running successfully without any errors…

If failures exceed a threshold, the circuit breaker opens. explain in layman term as i am calling department service from employeeservice microservice

Scenario Without Circuit Breaker

🔹 Imagine you have an **EmployeeService** that calls **DepartmentService** to fetch department details for an employee.

🔹 But suddenly, **DepartmentService crashes** or becomes **too slow.**

🔹 Now, EmployeeSer eeps **sending requests**, waiting for a response.

🔹 This leads to **delays, timeouts, and possibly crashing EmployeeService too.**

This is like **knocking on a locked door continuously**, hoping someone will open, but nobody is inside! 🚪😳

How Circuit Breaker Helps

🔹 Now, suppose you **install a Circuit Breaker** in EmployeeService.

🔹 If EmployeeService **notices too many failures (e.g., 5 failures in a row)**, it **"opens the circuit"** 🚫.

🔹 Once op ployeeService **stops calling DepartmentService** for some time.

🔹 Instead, it **immediately returns a default/fallback respon**se like:

"Department details are unavailable right now. Please try again later."

🔹 After some time, it **tests DepartmentService again** (Half-Open state).

🔹 If DepartmentService is back, the **circuit closes,** a d normal requests continue.

This is like **seeing a locked door, and instead of knocking forever, you wait for a while before trying**

This is like **seeing a locked door, and instead of knocking forever, you wait for a while before trying** **again!** 🚪 ✅

🚀 Benefits of Circuit Breaker

- ✅ Prevents EmployeeService from crashing if DepartmentService fails
- ✅ Avoids overloading a failing service with repeated requests
- ✅ Provides a smooth user experience with fallback responses
- ✅ Auto-recovers when DepartmentService is back

Now, EmployeeService is **smart**—it knows **when to stop asking a broken DepartmentService** and **when to retry!** 🚀

2️⃣ Open (Failure Detected)

- Requests are **blocked immediately** to avoid unnecessary failures.
- The system **returns a fallback response** or an error message.
- A timer starts to periodically check if the service is back.

To work with the **circuit breaker** we need to add the few dependencies

Resilience4J SPRING CLOUD CIRCUIT BREAKER

Spring Cloud Circuit breaker with Resilience4j as the underlying implementation.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
  </dependency>
```

As employee service is calling the department service…so we need to add this dependency in the emp service only

Add it in the pom.xml file

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

Add this one also

Then we should use the circuit breaker annotation to the method where it is calling the external service

```
@Override
@CircuitBreaker(name = "${spring.application.name}" , fallbackMethod = "getDefaultDepartmer
public EmployeeDepartmentDto getEmployeeById(Long id) {
    Employee employee = repo.findById(id)
```

When it is failed then it is calling the get dept methd

```
public EmployeeDepartmentDto getDefaultDepartment(Long id) {
DepartmentDto defaultDepartment = new DepartmentDto();
    defaultDepartment.setDepartment("DEFAULT");
    defaultDepartment.setDeptcode("Default Department");
    defaultDepartment.setDescription("This is a fallback department");
```

localhost:8081/actuator/health


```
{
    "status": "DOWN",
    "components": {
        "binders": {
            "status": "DOWN",
            "components": {
                "rabbit": {
                    "status": "DOWN",
                    "details": {
                        "error": "org.springframework.amqp.AmqpConnectException: java.net.ConnectException: Connection refused: no further information"
                    }
                }
            }
        },
        "circuitBreakers": {
            "status": "UP",
            "details": {
                "employee-service": {
                    "status": "UP",
                    "details": {
                        "failureRate": "-1.0%",
                        "failureRateThreshold": "50.0%",
                        "slowCallRate": "-1.0%",
                        "slowCallRateThreshold": "100.0%",
                        "bufferedCalls": 0,
                        "slowCalls": 0,
                        "slowFailedCalls": 0,
                        "failedCalls": 0,
                        "notPermittedCalls": 0,
                        "state": "CLOSED"
                    }
                }
            }
        },
        "clientConfigServer": {
            "status": "UP",
            "details": {
                "propertySources": [
                    "configserver:https://github.com/PavanKalyan96Dev/ConfigServer/employee-service.properties",
                    "configClient"
```

localhost:8081/actuator/circuitbreakers


```
{
  "circuitBreakers": {
    "employee-service": {
      "failureRate": "-1.0%",
      "slowCallRate": "-1.0%",
      "failureRateThreshold": "50.0%",
      "slowCallRateThreshold": "100.0%",
      "bufferedCalls": 2,
      "failedCalls": 2,
      "slowCalls": 0,
      "slowFailedCalls": 0,
      "notPermittedCalls": 0,
      "state": "CLOSED"
    }
  }
}
```


GET localhost:8081/actuator/circuitbreakers Send

Params Authorization Headers (7) Body Scripts Settings Cookies

Query Params

Key	Value	Description	Bulk Edit

Body Cookies Headers (5) Test Results 200 OK 46 ms 450 B

{} JSON Preview Visualize


```
{
  "circuitBreakers": {
    "employee-service": {
      "failureRate": "-1.0%",
      "slowCallRate": "-1.0%",
      "failureRateThreshold": "50.0%",
      "slowCallRateThreshold": "100.0%",
      "bufferedCalls": 2,
      "failedCalls": 2,
      "slowCalls": 0,
      "slowFailedCalls": 0,
      "notPermittedCalls": 0,
      "state": "CLOSED"
    }
  }
```


Now stop the department service and run the url in the postman

localhost:8081/employees/352 for 5 times then

http://localhost:8081/actuator/circuitbreakers refresh it

localhost:8081/actuator/circuitbreakers


```
{
  "circuitBreakers": {
    "employee-service": {
      "failureRate": "-1.0%",
      "slowCallRate": "-1.0%",
      "failureRateThreshold": "50.0%",
      "slowCallRateThreshold": "100.0%",
      "bufferedCalls": 0,
      "failedCalls": 0,
      "slowCalls": 0,
      "slowFailedCalls": 0,
      "notPermittedCalls": 0,
      "state": "HALF_OPEN"
    }
  }
}
```


what is the main advantage of retry pattern in microservice

Main Advantages of the Retry Pattern in Microservices

The **Retry Pattern** is crucial in microservices architecture to handle **transient** failures effectively. Here are its **key advantages**:

✅ 1. Improves Resilience

- Microservices communicate over networks, which are **unreliable**.
- A **temporary failure** (e.g., network timeout, slow response) should not immediately result in failure.
- **Retry ensures that transient issues do not cause service failure.**

🔹 **Example:**

If a **database service** is temporarily down, retrying can help avoid a system-wide failure when it comes back online.

When Should You Use Retry?

✅ Yes, Use Retry When:

- The failure is **temporary** (e.g., network issue, database timeout).
- The request is **idempotent** (can be retried without side effects).
- The retry delay does not cause **performance degradation**.

❌ Avoid Retry When:

- The failure is **permanent** (e.g., authentication failure, incorrect input).
- The request is **non-idempotent** (e.g., duplicate transactions in payments).

Seek

Conclusion

✔ The **Retry Pattern** ensures **high availability, better user experience, and service resilience.**

✔ When used with **Exponential Backoff** and **Circuit Breaker**, it **prevents unnecessary load** on failing services.

✔ Essential for **network-heavy microservices architectures.**

Retry

```
#Retry configuration

management.endpoint.retryevents.enabled=true

resilience4j.retry.instances.employee-service.max-attempts=5

resilience4j.retry.instances.employee-service.wait-duration=2s
```

```
@Retry(name="${spring.application.name}",fallbackMethod = "getDefaultDepartment")
    @Override
    public APIResponseDto getEmployeeById(Long employeeId) {
        // TODO Auto-generated method stub
        System.out.println("iam in getEmployeeById");
        Employee employee = employeeRepository.findById(employeeId).get();

        //ResponseEntity<DepartmentDto> responseEntity =
restTemplate.getForEntity("http://localhost:9090/departments/"+employee.getDepartmentCod
e(), DepartmentDto.class);

        //DepartmentDto departmentDto = responseEntity.getBody();

        DepartmentDto departmentDto = webClient.get()
                .uri("http://localhost:9090/departments/"+employee.getDepartmentCode())
                .retrieve()
                .bodyToMono(DepartmentDto.class)
                .block();

       //DepartmentDto departmentDto =
apiClient.getDepartmentByCode(employee.getDepartmentCode());

        EmployeeDto employeeDto = mapper.map(employee, EmployeeDto.class);
```

```
        APIResponseDto apiresponseDto = new APIResponseDto();

        apiresponseDto.setDepartmentDto(departmentDto);

        apiresponseDto.setEmployeeDto(employeeDto);

        return apiresponseDto;


    }
```