

Ensuring system stability and resilience is crucial for providing a reliable service to users. One of the critical aspects in achieving a stable and resilient system for production is managing the integration points between services over a network.

There exist various patterns for building resilient applications. In the Java ecosystem, Hystrix, a library developed by Netflix, was widely used for implementing such patterns. However, Hystrix entered maintenance mode in 2018 and is no longer being actively developed. To address this, **Resilience4J** has gained significant popularity, stepping in to fill the gap left by Hystrix. Resilience4J provides a comprehensive set of features for building resilient applications and has become a go-to choice for Java developers.



Resilience4j is a lightweight fault tolerance library designed for functional programming. It offers the following patterns for increasing fault tolerance due to network problems or failure of any of the multiple services:



Circuit breaker - Used to stop making requests when a service invoked is failing



Fallback - Alternative paths to failing requests



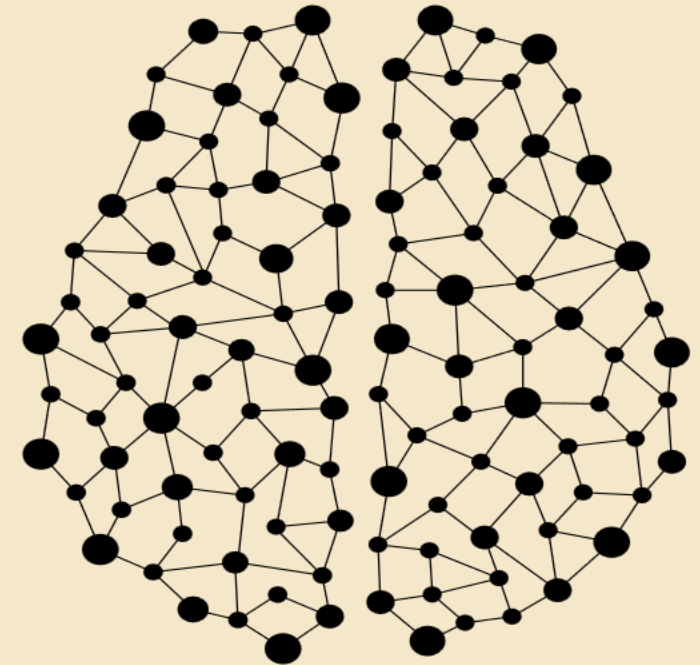
Retry - Used to make retries when a service has temporarily failed



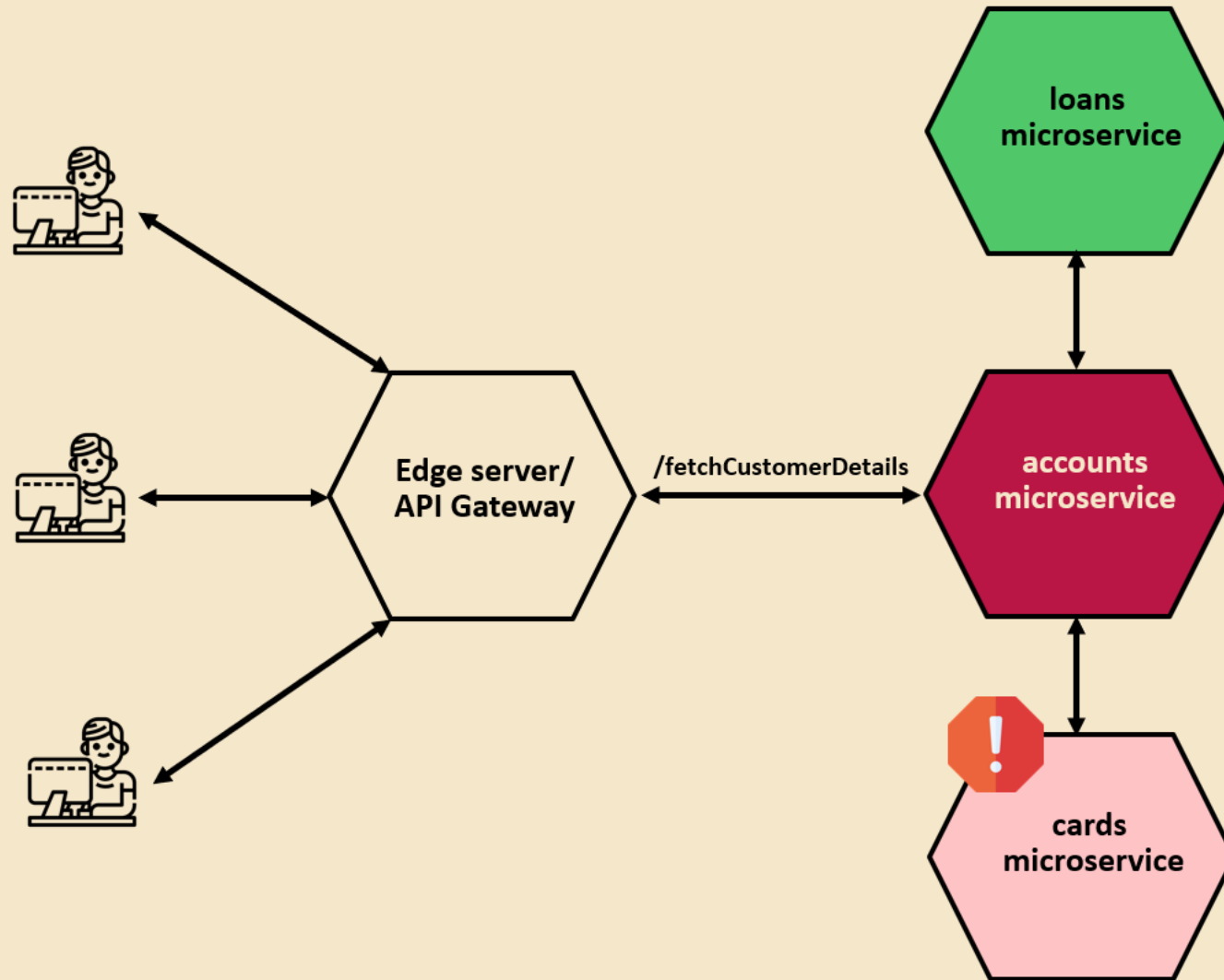
Rate limit - Limits the number of calls that a service receives in a time



Bulkhead - Limits the number of outgoing concurrent requests to a service to avoid overloading

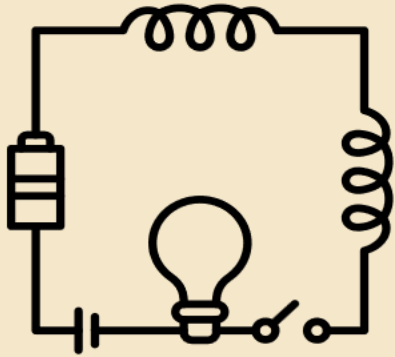


TYPICAL SCENARIO IN MICROSERVICES



When a microservice responds slowly or fails to function, it can lead to the depletion of resource threads on the Edge server and intermediate services. This, in turn, has a negative impact on the overall performance of the microservice network.

To handle this kind of scenarios, we can use Circuit Breaker pattern



In an electrical system, a circuit breaker is a safety device designed to protect the electrical circuit from excessive current, preventing damage to the circuit or potential fire hazards. It automatically interrupts the flow of electricity when it detects a fault, such as a short circuit or overload, to ensure the safety and stability of the system.

The Circuit Breaker pattern in software development takes its inspiration from the concept of an electrical circuit breaker found in electrical systems.

In a distributed environment, calls to remote resources and services can fail due to transient faults, such as slow network connections, timeouts, or the resources being overcommitted or temporarily unavailable. These faults typically correct themselves after a short period of time, and a robust cloud application should be prepared to handle them.

The Circuit Breaker pattern which inspired from electrical circuit breaker will monitor the remote calls. If the calls take too long, the circuit breaker will intercede and kill the call. Also, the circuit breaker will monitor all calls to a remote resource, and if enough calls fail, the circuit break implementation will pop, failing fast and preventing future calls to the failing remote resource.

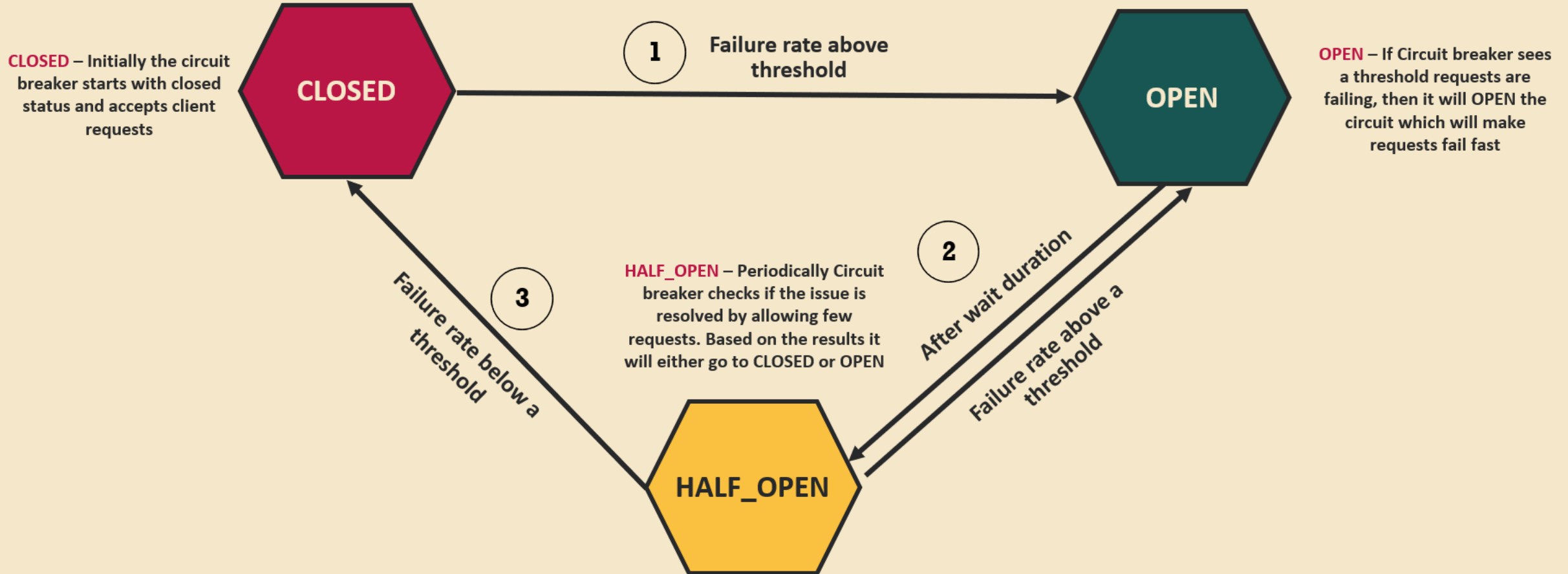
The Circuit Breaker pattern also enables an application to detect whether the fault has been resolved. If the problem appears to have been fixed, the application can try to invoke the operation.

The advantages with circuit breaker pattern are,

- ✓ Fail fast
- ✓ Fail gracefully
- ✓ Recover seamlessly

CIRCUIT BREAKER PATTERN

In Resilience4j, the circuit breaker is implemented via three states



Below are the steps to build a circuit breaker pattern using **Spring Cloud Gateway filter**,

1 Add maven dependency: Add spring-cloud-starter-circuitbreaker-reactor-resilience4j maven dependency inside pom.xml

2 Add circuit breaker filter: Inside the method where we are creating a bean of RouteLocator, add a filter of circuit breaker like highlighted below and create a REST API handling the fallback uri `/contactSupport`

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
    return builder.routes()
        .route(p -> p.path("/eazybank/accounts/**")
            .filters(f -> f.rewritePath("/eazybank/accounts/(?<segment>.*)", "/${segment}")
                .addResponseHeader("X-Response-Time", new Date().toString())
                .circuitBreaker(config -> config.setName("accountsCircuitBreaker")
                    .setFallbackUri("forward:/contactSupport")))
            .uri("lb://ACCOUNTS")).build();
}
```

3 Add properties: Add the below properties inside the application.yml file,

```
resilience4j.circuitbreaker:
  configs:
    default:
      slidingWindowSize: 10
      permittedNumberOfCallsInHalfOpenState: 2
      failureRateThreshold: 50
      waitDurationInOpenState: 10000
```

CIRCUIT BREAKER PATTERN

Below are the steps to build a circuit breaker pattern using **normal Spring Boot service**,

1 Add maven dependency: Add spring-cloud-starter-circuitbreaker-resilience4j maven dependency inside pom.xml

2 Add circuit breaker related changes in Feign Client interfaces like shown below:

```
@FeignClient(name = "cards", fallback = CardsFallback.class)
public interface CardsFeignClient {

    @GetMapping(value = "/api/fetch", consumes = "application/json")
    public ResponseEntity<CardsDto> fetchCardDetails(@RequestHeader("easybank-correlation-id")
                                                    String correlationId, @RequestParam String mobileNumber);
}
```

```
@Component
public class CardsFallback implements CardsFeignClient{
    @Override
    public ResponseEntity<CardsDto> fetchCardDetails(String correlationId, String mobileNumber){
        return null;
    }
}
```





3

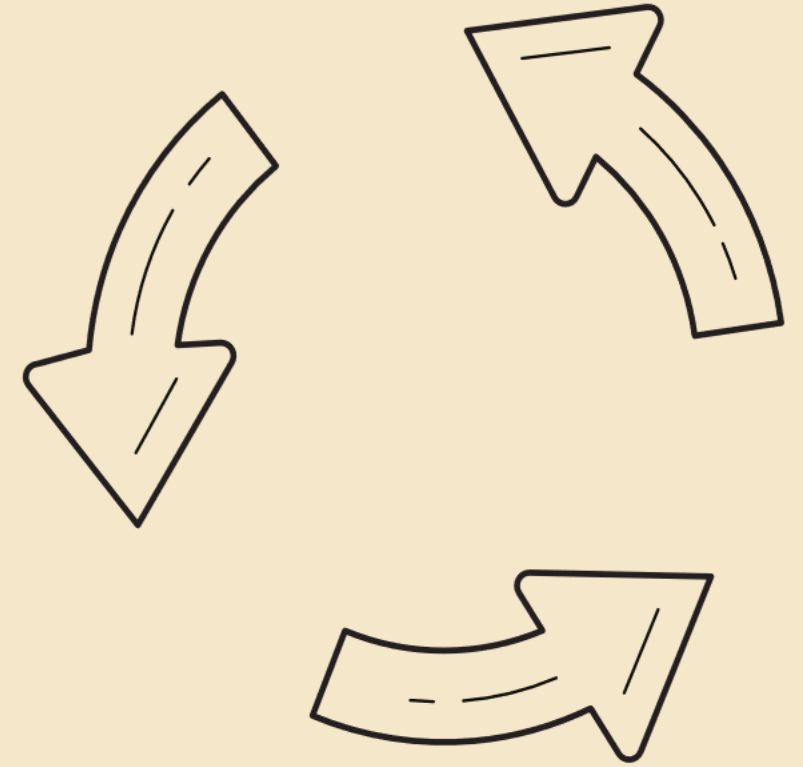
Add properties: Add the below properties inside the application.yml file,

```
spring:
  cloud:
    openfeign:
      circuitbreaker:
        enabled: true
resilience4j.circuitbreaker:
  configs:
    default:
      slidingWindowSize: 5
      failureRateThreshold: 50
      waitDurationInOpenState: 10000
      permittedNumberOfCallsInHalfOpenState: 2
```


The retry pattern will make configured multiple retry attempts when a service has temporarily failed. This pattern is very helpful in the scenarios like network disruption where the client request may successful after a retry attempt.

Here are some key components and considerations of implementing the Retry pattern in microservices:

-  **Retry Logic:** Determine when and how many times to retry an operation. This can be based on factors such as error codes, exceptions, or response status.
-  **Backoff Strategy:** Define a strategy for delaying retries to avoid overwhelming the system or exacerbating the underlying issue. This strategy can involve gradually increasing the delay between each retry, known as exponential backoff.
-  **Circuit Breaker Integration:** Consider combining the Retry pattern with the Circuit Breaker pattern. If a certain number of retries fail consecutively, the circuit can be opened to prevent further attempts and preserve system resources.
-  **Idempotent Operations:** Ensure that the retried operation is idempotent, meaning it produces the same result regardless of how many times it is invoked. This prevents unintended side effects or duplicate operations.



Below are the steps to build a retry pattern using **Spring Cloud Gateway filter**,

1

Add Retry filter: Inside the method where we are creating a bean of RouteLocator, add a filter of retry like highlighted below,

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
    return builder.routes()
        .route(p -> p.path("/eazybank/loans/**")
            .filters(f -> f.rewritePath("/eazybank/loans/(?<segment>.*)", "/${segment}")
                .addResponseHeader("X-Response-Time", new Date().toString())
                .retry(retryConfig -> retryConfig.setRetries(3).setMethods(HttpMethod.GET)
                    .setBackoff(Duration.ofMillis(100), Duration.ofMillis(1000), 2, true)))
            .uri("lb://LOANS")) .build();
}
```

Below are the steps to build a retry pattern using **normal Spring Boot service**,

1

Add Retry pattern annotations: Choose a method and mention retry pattern related annotation along with the below configs. Post that create a fallback method matching the same method signature like we discussed inside the course,

```
@Retry(name = "getBuildInfo", fallbackMethod = "getBuildInfoFallBack")
@GetMapping("/build-info")
public ResponseEntity<String> getBuildInfo() {

}

private ResponseEntity<String> getBuildInfoFallBack(Throwable t) {

}
```

2

Add properties: Add the below properties inside the application.yml file,

```
resilience4j.retry:
  configs:
    default:
      maxRetryAttempts: 3
      waitDuration: 500
      enableExponentialBackoff: true
      exponentialBackoffMultiplier: 2
      retryExceptions:
        - java.util.concurrent.TimeoutException
      ignoreExceptions:
        - java.lang.NullPointerException
```

The Rate Limiter pattern in microservices is a design pattern that helps control and limit the rate of incoming requests to a service or API. It is used to prevent abuse, protect system resources, and ensure fair usage of the service.

In a microservices architecture, multiple services may depend on each other and make requests to communicate. However, unrestricted and uncontrolled requests can lead to performance degradation, resource exhaustion, and potential denial-of-service (DoS) attacks. The Rate Limiter pattern provides a mechanism to enforce limits on the rate of incoming requests.

Implementing the Rate Limiter pattern helps protect microservices from being overwhelmed by excessive or malicious requests. It ensures the stability, performance, and availability of the system while providing controlled access to resources. By enforcing rate limits, the Rate Limiter pattern helps maintain a fair and reliable environment for both the service provider and its consumers.

When a user surpasses the permitted number of requests within a designated time frame, any additional requests are declined with an HTTP 429 - Too Many Requests status. The specific limit is enforced based on a chosen strategy, such as limiting requests per session, IP address, user, or tenant. The primary objective is to maintain system availability for all users, especially during challenging circumstances. This exemplifies the essence of resilience. Additionally, the Rate Limiter pattern proves beneficial for providing services to users based on their subscription tiers. For instance, distinct rate limits can be defined for basic, premium, and enterprise users.



Below are the steps to build a rate limiter pattern using **Spring Cloud Gateway filter**,

1

Add maven dependency: Add spring-boot-starter-data-redis-reactive maven dependency inside pom.xml and make sure a redis container started. Mention redis connection details inside the application.yml file

2

Add rate limiter filter: Inside the method where we are creating a bean of RouteLocator, add a filter of rate limiter like highlighted below and creating supporting beans of RedisRateLimiter and KeyResolver

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
    return builder.routes()
        .route(p -> p.path("/eazybank/cards/**")
            .filters(f -> f.rewritePath("/eazybank/cards/(?<segment>.*)", "/${segment}")
                .addResponseHeader("X-Response-Time", new Date().toString())
                .requestRateLimiter(config ->
                    config.setRateLimiter(redisRateLimiter()).setKeyResolver(userKeyResolver()))
            .uri("lb://CARDS")).build();
}

@Bean
public RedisRateLimiter redisRateLimiter() {
    return new RedisRateLimiter(1, 1, 1);
}

@Bean
KeyResolver userKeyResolver() {
    return exchange -> Mono.justOrEmpty(exchange.getRequest().getHeaders().getFirst("user"))
        .defaultIfEmpty("anonymous");
}
```

Below are the steps to build a rate limiter pattern using **normal Spring Boot service**,

1

Add Rate limiter pattern annotations: Choose a method and mention rate limiter pattern related annotation along with the below configs. Post that create a fallback method matching the same method signature like we discussed inside the course,

```
@RateLimiter(name = "getJavaVersion", fallbackMethod = "getJavaVersionFallback")
@GetMapping("/java-version")
public ResponseEntity<String> getJavaVersion() {

}

private ResponseEntity<String> getJavaVersionFallback(Throwable t) {

}
```

2

Add properties: Add the below properties inside the application.yml file,

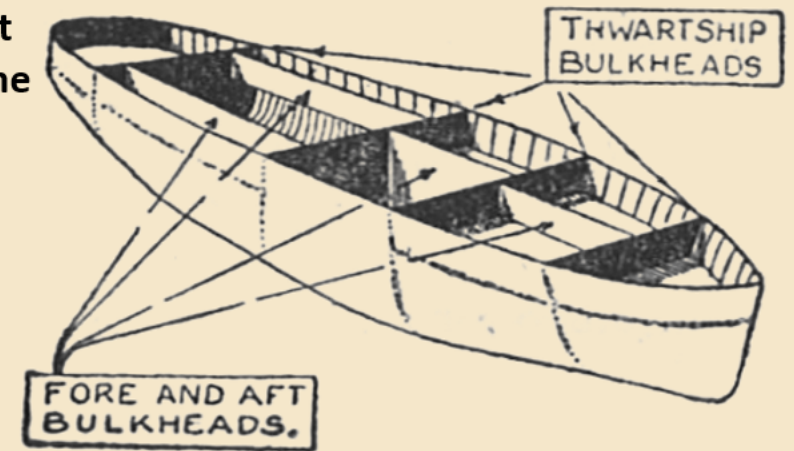
```
resilience4j.ratelimiter:
  configs:
    default:
      timeoutDuration: 5000
      limitRefreshPeriod: 5000
      limitForPeriod: 1
```

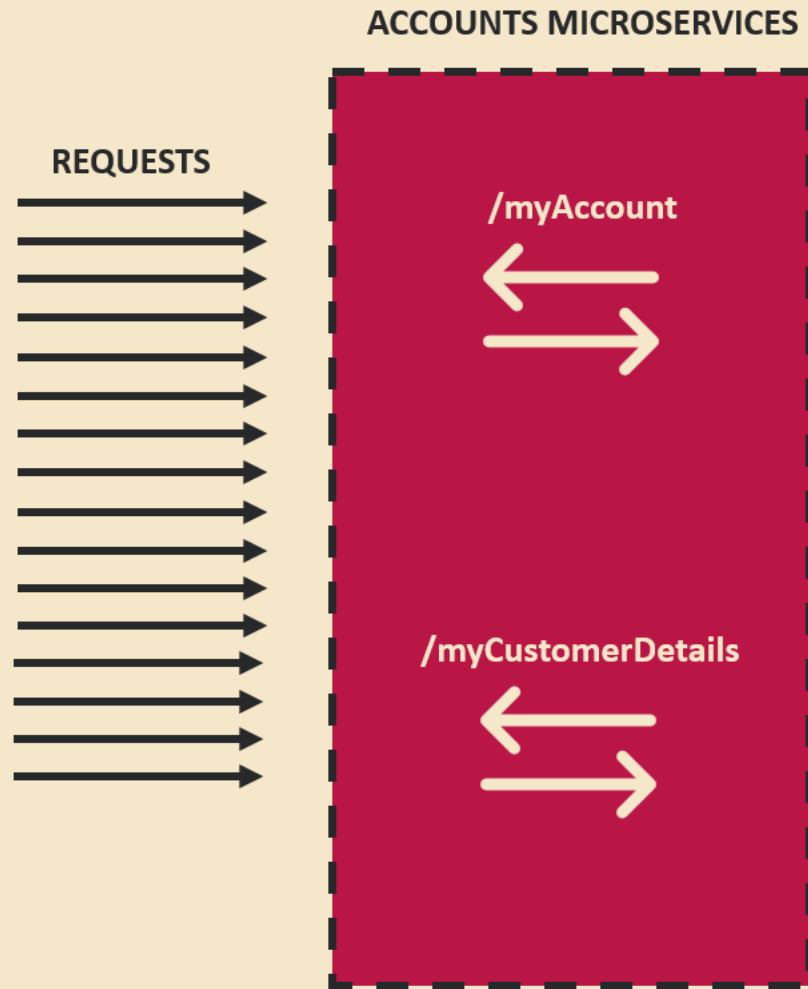
The Bulkhead pattern in software architecture is a design pattern that aims to improve the resilience and isolation of components or services within a system. It draws inspiration from the concept of bulkheads in ships, which are physical partitions that prevent the flooding of one compartment from affecting others, enhancing the overall stability and safety of the vessel.

In the context of software systems, the Bulkhead pattern is used to isolate and limit the impact of failures or high loads in one component from spreading to other components. It helps ensure that a failure or heavy load in one part of the system does not bring down the entire system, enabling other components to continue functioning independently.

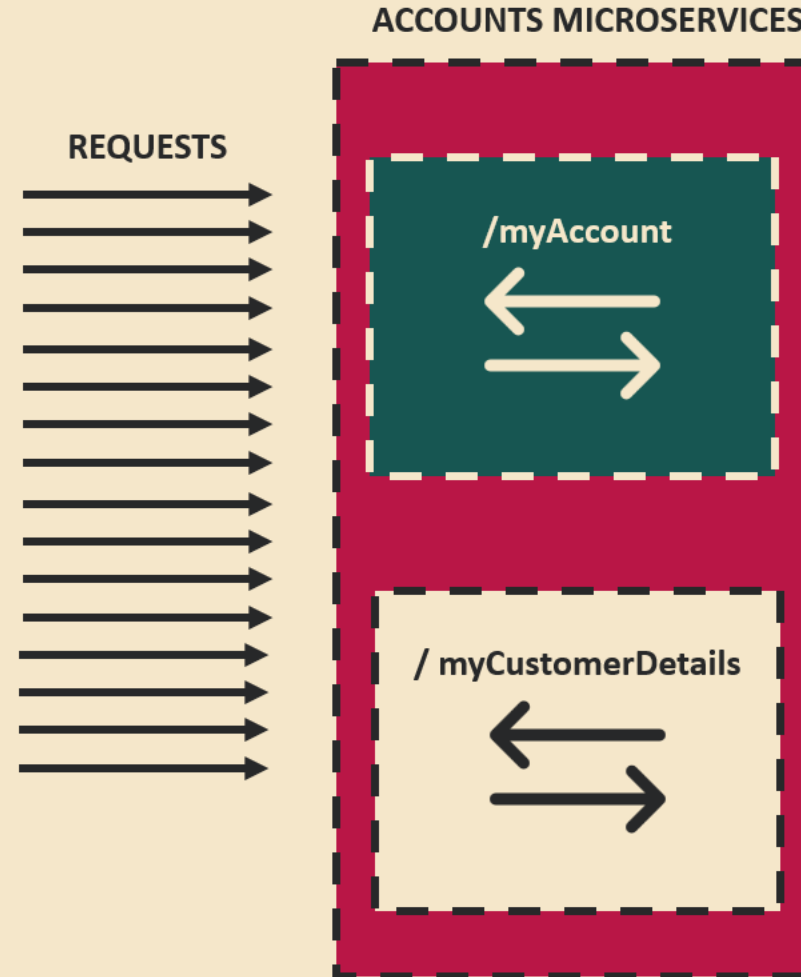
Bulkhead Pattern helps us to allocate limit the resources which can be used for specific services. So that resource exhaustion can be reduced.

The Bulkhead pattern is particularly useful in systems that require high availability, fault tolerance, and isolation between components. By compartmentalizing components and enforcing resource boundaries, the Bulkhead pattern enhances the resilience and stability of the system, ensuring that failures or heavy loads in one area do not bring down the entire system.





Without Bulkhead, /myCustomerDetails will start eating all the threads, resources available which will effect the performance of /myAccount



With Bulkhead, /myCustomerDetails and /myAccount will have their own resources, threads pool defined