RESILIENCY IN MICROSERVICES

eazy bytes

HOW DO WE AVOID CASCADING FAILURES?

One failed or slow service should not have a ripple effect on the other microservices. Like in the scenarios of multiple microservices are communicating, we need to make sure that the entire chain of microservices does not fail with the failure of a single microservice

HOW DO WE HANDLE FAILURES GRACEFULLY WITH FALLBACKS?

In a chain of multiple microservices, how do we build a fallback mechanism if one of the microservice is not working. Like returning a default value or return values from cache or call another service/DB to fetch the results etc.

HOW TO MAKE OUR SERVICES SELF-HEALING CAPABLE

In the cases of slow performing services, how do we configure timeouts, retries and give time for a failed services to recover itself.

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.

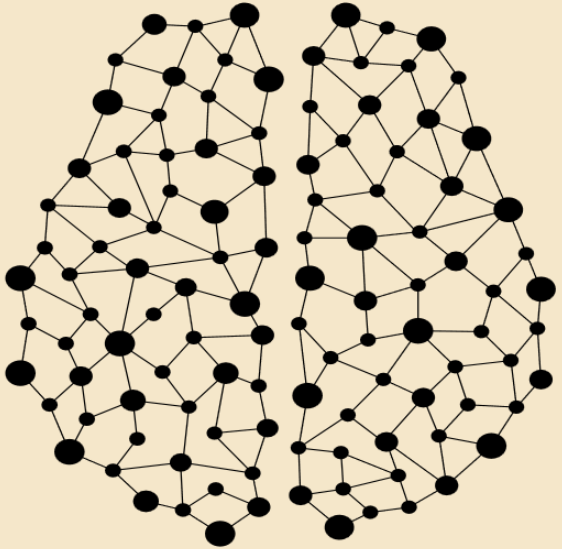
There exist various patterns for building resilient applications. In the Java ecosystem, Hystrix, a library developed by Netflix, was widely used for implementing such patterns. However, Hystrix entered maintenance mode in 2018 and is no longer being actively developed. To address this, Resilience4J 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.

RESILIENCY USING RESILIENCE4J

eazy bytes

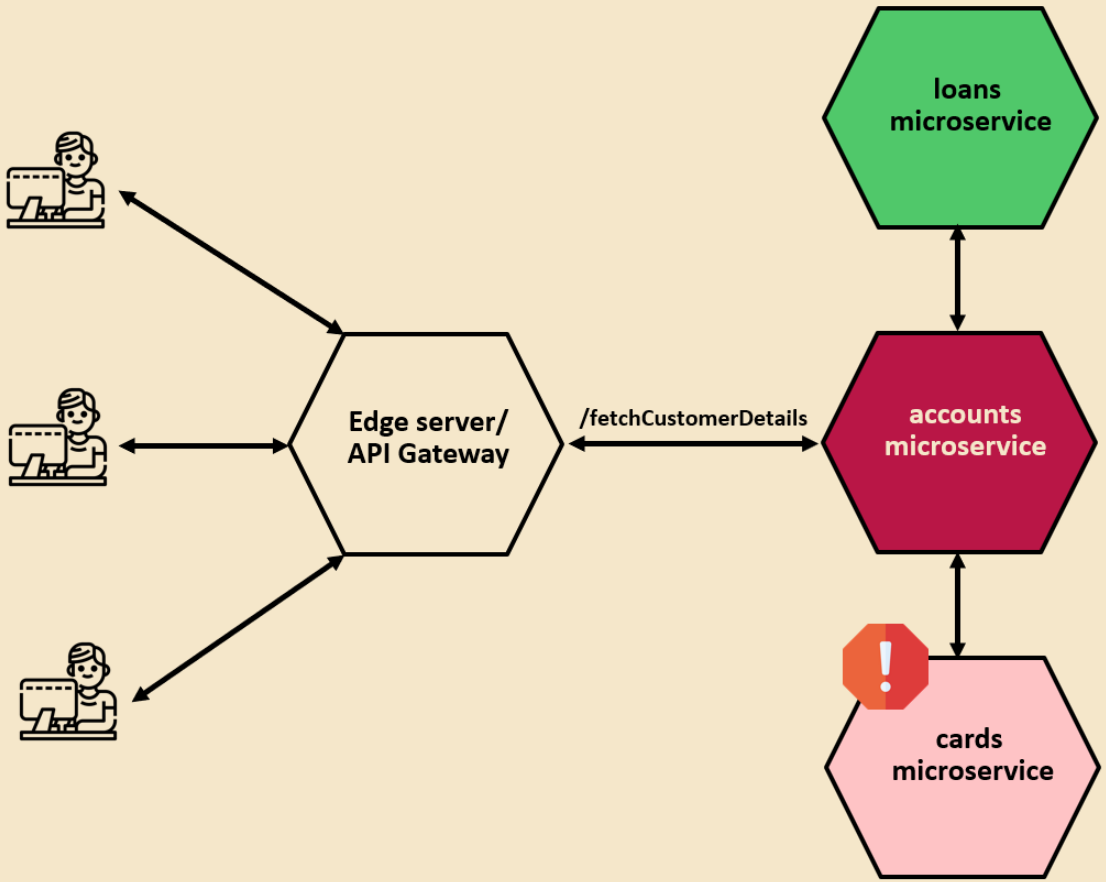
Resilience4j is a lightweight fault tolerance library designed for functional programming. It offers the following patterns for increasing fault tolerance due to network problems or failure of any of the multiple services:

- **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

eazy bytes

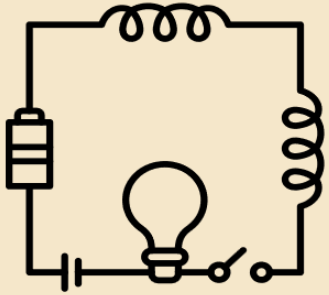


When a microservice responds slowly or fails to function, it can lead to the depletion of resource threads on the Edge server and intermediate services. This, in turn, has a negative impact on the overall performance of the microservice network.

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

CIRCUIT BREAKER PATTERN

eazy bytes



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.

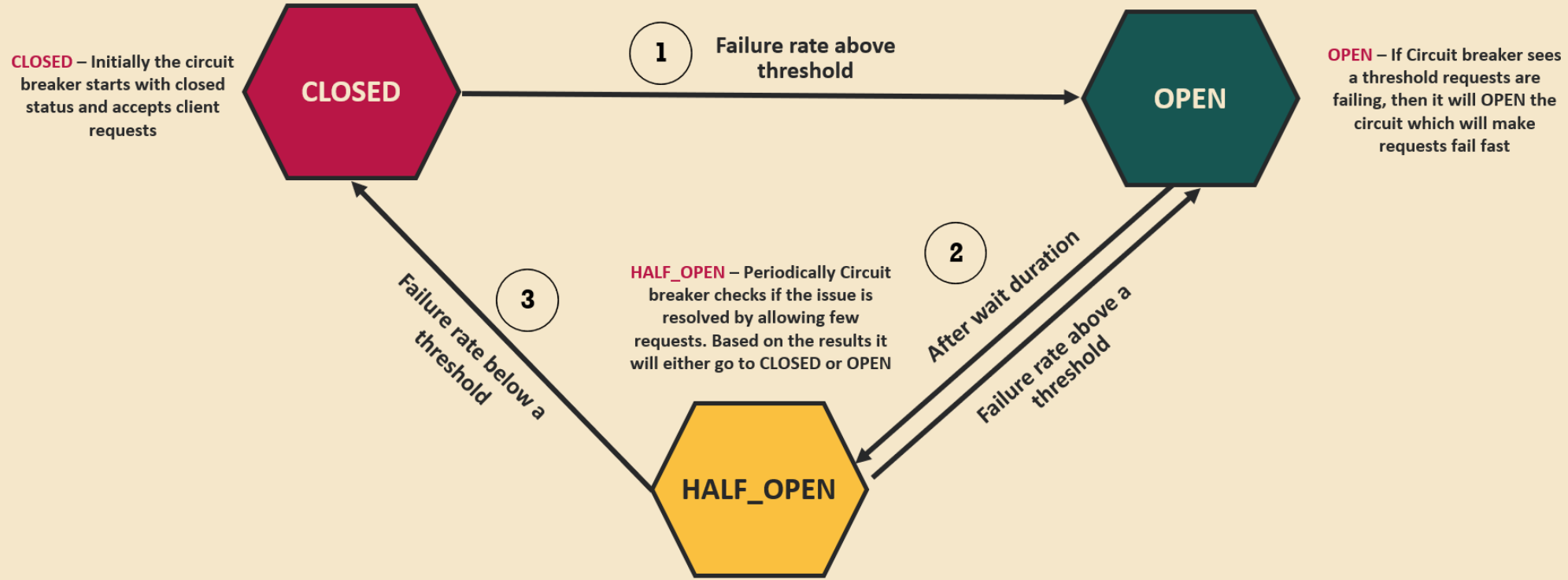
The advantages with circuit breaker pattern are,

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

CIRCUIT BREAKER PATTERN

eazy bytes

In Resilience4j, the circuit breaker is implemented via three states



CIRCUIT BREAKER PATTERN

eazy bytes

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

eazy bytes

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("eazybank-correlation-id")
                                    String correlationId, @RequestParam String mobileNumber);

}
```

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

CIRCUIT BREAKER PATTERN

eazy bytes

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
```

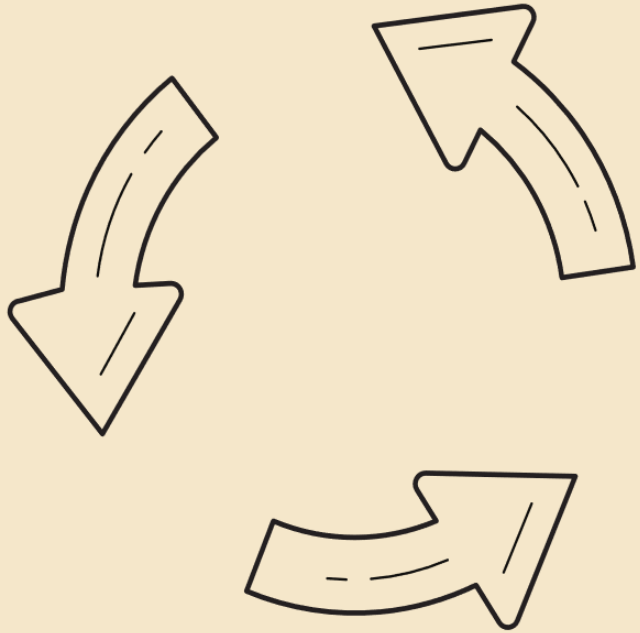
RETRY PATTERN

eazy bytes

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.



RETRY PATTERN

eazy bytes

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();
}
```

RETRY PATTERN

eazy bytes

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) {

}
```

RETRY PATTERN

eazy bytes

2

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

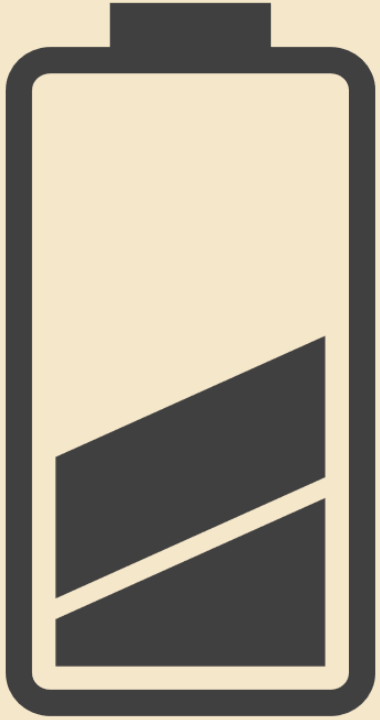
```
resilience4j.retry:
  configs:
    default:
      maxRetryAttempts: 3
      waitDuration: 500
      enableExponentialBackoff: true
      exponentialBackoffMultiplier: 2
      retryExceptions:
       - java.util.concurrent.TimeoutException
      ignoreExceptions:
       - java.lang.NullPointerException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

RATE LIMITTER PATTERN

eazy bytes

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.

RATE LIMITTER PATTERN

eazy bytes

Below are the steps to build a rate limitter 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 limitter filter:** Inside the method where we are creating a bean of RouteLocator, add a filter of rate limitter 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");
}
```

RATE LIMITTER PATTERN

eazy bytes

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

1 **Add Rate limitter pattern annotations:** Choose a method and mention rate limitter 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,

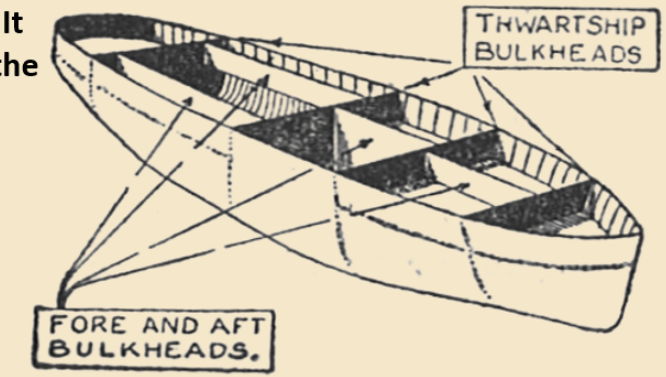
```
resilience4j.ratelimiter:
  configs:
    default:
      timeoutDuration: 5000
      limitRefreshPeriod: 5000
      limitForPeriod: 1
```

BULKHEAD PATTERN

eazy bytes

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.

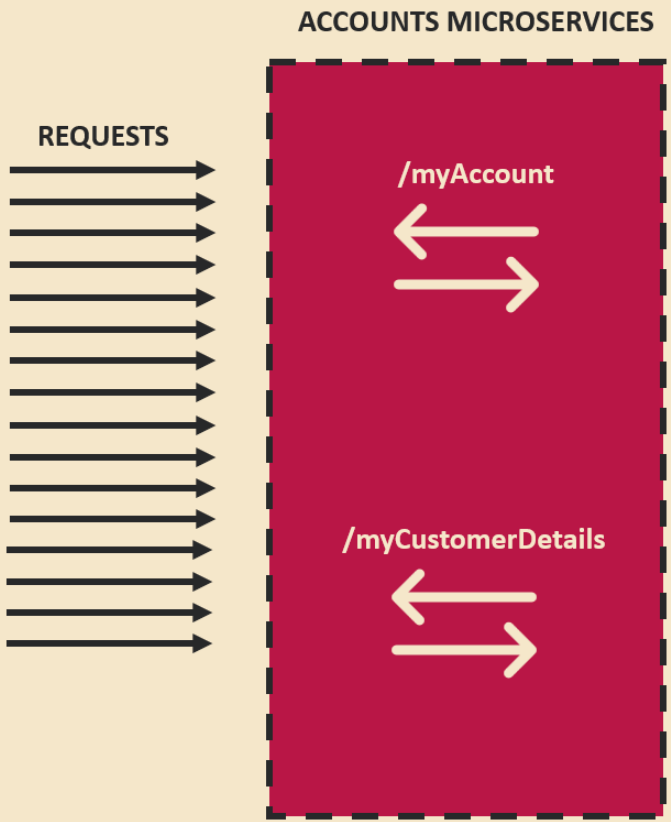


Bulkhead Pattern helps us to allocate limit the resources which can be used for specific services. So that resource exhaustion can be reduced.

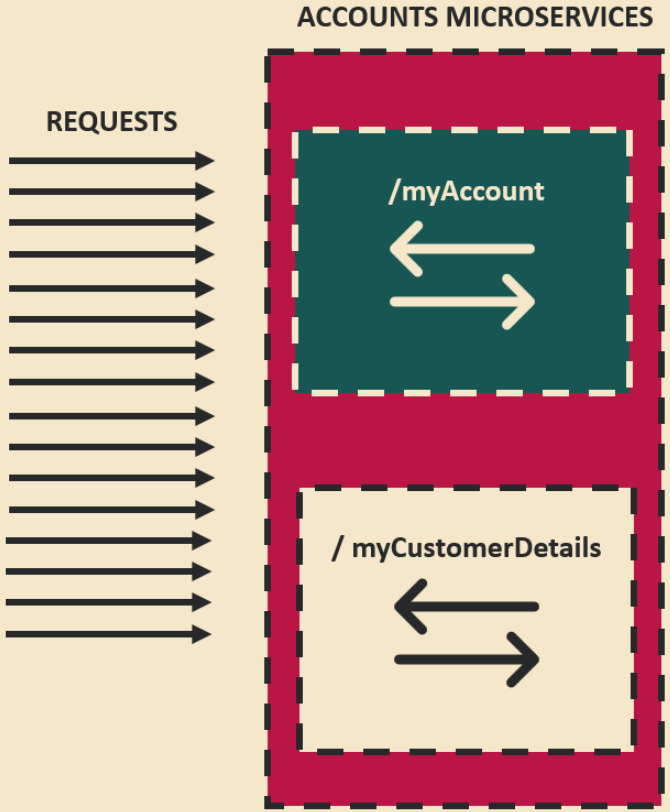
The Bulkhead pattern is particularly useful in systems that require high availability, fault tolerance, and isolation between components. By compartmentalizing components and enforcing resource boundaries, the Bulkhead pattern enhances the resilience and stability of the system, ensuring that failures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

BULKHEAD PATTERN

eazy bytes



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

Link for official doc: https://resilience4j.readme.io/docs/getting-started

- *docker run -p 3306:3306 --name accountsdb -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=accountsdb -d mysql*

- We tested "CIRCUIT BREAKER" fall back method, with gateway-server delegating request to account service (basically checking fault tolerance our account service can be, when tested from gateway-server). we did below steps in 1st phase, one can follow pdf for more info until Page 115

    - we implemented necessary dependency in pom.xml file of gateway-server
    - later we added necessary configurations info in application.yml file of gateway-server
    - Then created a controller class inside gateway-server application.
    - At the end we included a filter inside customConfigRoute of Gateway-server application.

- Now we might want to test different services like "cards" and "loans" via open-feign, basically we don't want our customer to be blocked from our application when they are fetching their details (In our application we are having an API - fetchCustomerDetails, which fetches all the details of a customer like accounts, cards and loans). That's where we need to follow below steps so in a scenario where one of the service is down such as cards or loans then the customer won't get completely blocked. Official doc for circuit breaker with open-feign: https://docs.spring.io/spring-cloud-openfeign/docs/current/reference/html/#spring-cloud-feign-circuitbreaker

    - First we need to necessary dependency in pom.xml file (for example in our case we have it in "accounts" and "gateway-server")
    - Define Fallback mechanism inside client's interface (Such as CardsFeignClient and LoansFeignClient).
    - Later we need to create a Bean that has the implementation of Fallback mechanism.
    - Also we do have include necessary configuration inside application.yml which give us more insight on fallback scenario.

- Later we implemented some properties by following below documentation to add a "http-timeout" to cards, gateway-server and loans. Http timeout configuration: https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-gateway/http-timeouts-configuration.html

Some important points to remember:

- Please note that, as of now there is no resilience pattern to handle timeouts as this is kind of covered with retry. We can set a desired timeout for all HTTP calls and based on this, Retry can attempt retries and trigger fallback as well.
- Also when defining the "http timeouts", we need to ensure that the timeouts of gateway-server is more than the per route, because if it's other way around than we might see "timeout" exception even before the services communicate with gateway-server, as timeout in gateway-server is applied globally within the application.
- **Why we used spring-cloud-starter-circuitbreaker-resilience4j in account service and use spring-cloud-starter-circuitbreaker-reactor-resilience4j in gateway server** - Spring framework supports development using 2 styles. One is using traditional servlets and other one using reactive based approach. Since accounts micro-service is developed using traditional approach I have used spring-cloud-starter-circuitbreaker-resilience4j where as the Spring Cloud Gateway is developed using reactive approach, so we need to use spring-cloud-starter-circuitbreaker-reactor-resilience4j dependency

Retry Pattern

we did "retry" example within gateway-server for loans service

```
.route(p -> p
        .path( ...patterns: "/mdfinance/loans/**") BooleanSpec
        .filters( f -> f.rewritePath( regex: "/mdfinance/loans/(?<segment>.*)", replacement: "/${segment}")
                .addResponseHeader( headerName: "X-Response-Time", LocalDateTime.now().toString())
                .retry(retryConfig -> retryConfig.setRetries(3).setMethods(HttpMethod.GET)
                        .setBackoff(Duration.ofMillis(100), Duration.ofMillis(1000), factor: 2,
                                basedOnPreviousValue: true))) UriSpec

        .uri("lb://LOANS"))
```

- setRetries(3): This sets the maximum number of retry attempts. If a request fails, the gateway will try to send it again up to 3 times.
- setMethods(HttpMethod.GET): This specifies that only GET requests should be retried. If a request uses a different HTTP method (like POST or PUT), it won't be retried.
- setBackoff(Duration.ofMillis(100), Duration.ofMillis(1000), 2, true): This configures a backoff policy. If a request fails and needs to be retried, the gateway will wait before sending the request again. The wait time starts at 100 milliseconds and increases exponentially (by a factor of 2) for each subsequent retry, up to a maximum of 1000 milliseconds. The true argument means that the wait time will be randomized to some extent, which can help prevent a flood of retries from overwhelming the server.
- Note that there isn't any fallback method that we can used when defining retry pattern over gateway-server

we can also define the "retry" configuration within application.yml file, below is the example from Account service.

```yaml
resilience4j.retry: # This wll be applicable to only account service
    configs:
        default:
            maxRetryAttempts: 3
            waitDuration: 500
            enableExponentialBackoff: true
            exponentialBackoffMultiplier: 2
            # Below exception won't be retried, to see the retry in action, comment the below line
            # Add logger.info statement in the controller method (getContactInfo, getContactInfoFallback),
            # to see the retry in action
            ignoreExceptions:
                - java.lang.NullPointerException
            retryExceptions:
                - java.util.concurrent.TimeoutException
```

With the above configuration, we can test how the an API from account service can handle failure with retrying operation. It's very important that we overall timeout for circuit breaker is higher than the timeout of each retry operation, because if that's not the case than fault back uri of account service will executed before any retry operation.

One of the way to overcome this issue is to define custom timeout to circuit breaker (which should be higher than timeout of retrying operation). And we can achieve this by defining a custom bean within gateway-server. look at the snippet in next page.

```java
@Retry(name = "retryServiceForContactInfo", fallbackMethod = "getContactInfoFallback")
@GetMapping("/contact-info")
public ResponseEntity<AccountsContactInfoDto> getContactInfo() {
    logger.info( msg: "Trying to fetch contact info");
    throw new NullPointerException();
}

/**
 * Fallback method for getContactInfo
 *
 * Params: e – Exception
 *
 * Returns: AccountsContactInfoDto
 */
public ResponseEntity<AccountsContactInfoDto> getContactInfoFallback(Exception e) {  no usages
    logger.info( msg: "Executing ContactInfoFallback method");
    Map<String, String> contactDetails = Map.of( k1: "name",  v1: "Maulik Davra - Developer",
            k2: "email",  v2: "maulik.davra@mdfinance.com");
    List<String> onCallSupport = List.of("(412) 419-3491)", "(915) 382-1932"); // replace with actual support contacts

    AccountsContactInfoDto fallbackContactInfo = new AccountsContactInfoDto(
            message: "Fallback message: Unable to fetch contact info at the moment",
            contactDetails,
            onCallSupport
    );

    return ResponseEntity
            .status(HttpStatus.INTERNAL_SERVER_ERROR)
            .body(fallbackContactInfo);
}
```

```java
/**
 * defaultCustomizer() is used to configure the default circuit breaker configuration for the gateway
 * server. The reason behind implementing below bean is that whenever a service is taking longer in
 * retrying the request, the circuit breaker will be triggered and the request will be forwarded to the
 * fallback URI. As the circuit breaker has less timeout duration, the request will be forwarded to the
 * fallback URI.
 *
 * The default configuration is set to timeout after 4 seconds.
 * The circuit breaker configuration is set to default.
 *
 * Returns: Customizer object for the ReactiveResilience4JCircuitBreakerFactory
 */
@Bean  no usages
public Customizer<ReactiveResilience4JCircuitBreakerFactory> defaultCustomizer() {
    return factory -> factory.configureDefault(id -> new Resilience4JConfigBuilder(id)
            .circuitBreakerConfig(CircuitBreakerConfig.ofDefaults())
            .timeLimiterConfig(TimeLimiterConfig.custom().timeoutDuration(Duration.ofSeconds(4)).build()).build());
}
```

We can include some basic timeout configuration like below within the application.yml file too.

```yaml
server:
  port: 9000
spring:
  application:
    name: "cards"
  cloud:
    gateway:
      httpclient: # This configuration is local and will be applied to the card service.
        connect-timeout: 1000
        response-timeout: 3s
```

Rate Limiter pattern

First we have implemented rate limiter within gateway-server application (configured into cards service). please look at the two beans that created within gateway-server application, they are responsible to handle the number request sent to the server for a user. Detail explanation is documented within gateway-server application. Here's the official doc for rate-limiter via gateway filter: https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-gateway/gatewayfilter-factories/requestratelimiter-factory.html#page-title (don't forget to import required dependency inside gateway-server)

Below is the snippet to look for!

```
.route(p -> p
        .path( ...patterns: "/mdfinance/cards/**") BooleanSpec
        .filters( f -> f.rewritePath( regex: "/mdfinance/cards/(?<segment>.*)", replacement: "/${segment}")
                .addResponseHeader( headerName: "X-Response-Time", LocalDateTime.now().toString())
                .requestRateLimiter(config -> config.setRateLimiter(redisRateLimiter())
                        .setKeyResolver(userKeyResolver()))) UriSpec
        .uri("lb://CARDS"))
.build();
```

```
Below, bean is used to configure the RedisRateLimiter for the gateway server. With the below
configuration user is allowed to make one request per second. The RedisRateLimiter is used to limit
the number of requests to the gateway server.

defaultReplenishRate defines how many requests per second to allow (without any dropped
requests). This is the rate at which the token bucket is filled.

defaultBurstCapacity is the maximum number of requests a user is allowed in a single second
(without any dropped requests). This is the number of tokens the token bucket can hold. Setting
this value to zero blocks all requests.

defaultRequestedTokens defines the number of tokens to consume for each request. This is the
number of tokens taken from the bucket for each request and defaults to 1

Returns: RedisRateLimiter object
@Bean  1 usage
public RedisRateLimiter redisRateLimiter() {
    return new RedisRateLimiter( defaultReplenishRate: 1,  defaultBurstCapacity: 1,  defaultRequestedTokens: 1);
}
```

```
The KeyResolver is a functional interface used in Spring Cloud Gateway for rate limiting purposes.
It's used to determine the key that will be used to identify a particular request. This key is then used
by the rate limiter to keep track of how many requests have been made with that key.

In the context of the GatewayserverApplication.java file, the userKeyResolver() method is
implemented to resolve the user key based on the "user" header in the request. If the "user" header
is not present in the request, then the user key is set to "anonymous". This means that the rate
limiting will be applied per user, as identified by the "user" header.

The use case for this is when you want to limit the rate of requests on a per-user basis. For
example, you might want to prevent a single user from making too many requests in a short period
of time, which could overload your server.

If you don't implement a KeyResolver, the default KeyResolver provided by Spring Cloud Gateway
will be used. The default KeyResolver uses the Principal Name from the ServerWebExchange object,
which might not be suitable for your use case. If the Principal Name is not available, it will throw an
exception. Therefore, it's often a good idea to provide a custom KeyResolver that suits your specific
needs.

Returns: KeyResolver object
@Bean  1 usage
KeyResolver userKeyResolver() {
    return exchange -> Mono.justOrEmpty(exchange.getRequest().getHeaders().getFirst( headerName: "user"))
            .defaultIfEmpty( defaultV: "anonymous");
}
```

Now we also need to implement the Redis inside the application.yml file of gateway-server.

```yaml
spring:
  application:
    name: "gatewayserver"
  config:
    import: "optional:configserver:http://localhost:8071/"
  cloud:
    gateway:
      discovery:
        locator:
          enabled: false
          lowerCaseServiceId: true
      httpclient: # This configuration is global and will be applied to all the services passing through the gateway.
        connect-timeout: 1000
        response-timeout: 3s
  data:
    redis:
      connect-timeout: 2s
      host: localhost
      port: 6379
      timeout: 1s
```

later we can run the redis-container locally with below command
Redis: docker run -p 6379:6379 --name mdfinanceredis -d redis

To test the functionality, we need to run all the application and then below command in terminal to see the report.
Testing via apache tool: ab -n 10 -c 2 -v 3 http://localhost:8072/mdfinance/cards/api/contact-info

- With the above command we are testing total of 10 request, 2 request per second

Below is the screenshot of the summary where only one request was allowed per second, other 9 got failed.

```
Server Hostname:        localhost
Server Port:            8072

Document Path:          /mdfinance/cards/api/contact-info
Document Length:        204 bytes

Concurrency Level:      2
Time taken for tests:   0.080 seconds
Complete requests:      10
Failed requests:        9
   (Connect: 0, Receive: 0, Length: 9, Exceptions: 0)
Non-2xx responses:      9
Total transferred:      2135 bytes
HTML transferred:       204 bytes
Requests per second:    124.40 [#/sec] (mean)
Time per request:       16.078 [ms] (mean)
Time per request:       8.039 [ms] (mean, across all concurrent requests)
Transfer rate:          25.94 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.2      1       1
Processing:     3   10  15.6      5      54
Waiting:        3    9  15.4      5      53
Total:          4   10  15.7      5      55
ERROR: The median and mean for the initial connection time are more than twice the standard
       deviation apart. These results are NOT reliable.

Percentage of the requests served within a certain time (ms)
  50%      5
  66%      5
  75%      6
  80%      7
  90%     55
  95%     55
  98%     55
  99%     55
 100%     55 (longest request)
MD@Mauliks-MacBook section-10 %
```

we can also implement the Rate limiter within the service itself, here is an example with account service

```java
@Operation( no usages
        summary = "Get Java version",
        description = "Get Java versions details that is installed into accounts microservice"
)
@ApiResponses({
        @ApiResponse(
                responseCode = "200",
                description = "HTTP Status OK"
        ),
        @ApiResponse(
                responseCode = "500",
                description = "HTTP Status Internal Server Error",
                content = @Content(
                        schema = @Schema(implementation = ErrorResponseDto.class)
                )
        )
})
)
@RateLimiter(name = "rateLimiterServiceForJavaVersion", fallbackMethod = "getJavaVersionFallback")
@GetMapping("/java-version")
public ResponseEntity<String> getJavaVersion() {
    return ResponseEntity
            .status(HttpStatus.OK)
            .body(environment.getProperty("JAVA_HOME"));
}

Fallback method for getJavaVersion

Params: e – Exception

Returns: String

public ResponseEntity<String> getJavaVersionFallback(Exception e) { no usages
    logger.info( msg: "Executing JavaVersionFallback method");
    return ResponseEntity
            .status(HttpStatus.INTERNAL_SERVER_ERROR)
            .body("Fallback message: Unable to fetch Java version at the moment");
}
```

Here's the configuration that needs to be included in application.yml file of account service.

```yaml
resilience4j.rateLimiter: # This will be applicable to only account service
    configs:
        default:
            timeoutDuration: 1000 # This is the time to wait for the response from the service before throwing an exception
            limitRefreshPeriod: 5000 # This is the time to reset the limit for the next period
            limitForPeriod: 1 # This is the number of requests allowed in the limitRefreshPeriod
```

Now, we can test the below API via postman (try to hit the "send" button multiple times to see the different response)
http://localhost:8072/mdfinance/accounts/api/java-version