Retry

Microlearning – 19th Jan 2024
& 22nd Jan 2024

Articles sent earlier this week

- Fail-Fast vs Fail-Safe https://talktotheduck.dev/fail-fast-reliable-software-strategy-debug-failures-effectively
- Story of Retry https://blog.chib.me/story-of-retry/

Contents

- Fail Fast vs Fail Safe Philosophy
- When should we retry?
- Theory
  - Backoff
  - Jitters
  - Adaptive Retry
- Methods
  - Apache HTTP Client
  - Spring-Retry
  - inet.common.util.Retry
- When and why should we NOT retry?
- Resiliency4J / Polly
  - Circuit Breaker
  - Bulkhead Isolation

Fail Fast vs Fail Safe Philosophy

- **Fail fast** = visibly and quickly fail in an unexpected condition
  - Java often uses the fail fast approach e.g. a null value throws a NullPointerException
  - “To make your software robust, make it fragile”
- **Fail safe** = try to recover and proceed even with bad input
  - Javascript uses “undefined” which can propagate through the system leading to polluted data that makes its way to the database or strange/unexpected behaviours later on
- Java <Optional> throws a wrench into this distinction of course

Source: https://www.martinfowler.com/ieeeSoftware/failFast.pdf

Fail Fast vs Fail Safe Philosophy

If as part of the new release, the config.properties file was updated and a developer introduced a typo?

Fail-Safe Approach

```
public int maxConnections() {
   string property =
     getProperty("maxConnections");
   if (property == null) {
     return 10;
   }
   else {
     return property.toInt();
   }
}
```

- The default max connections for every user is 10, the customers start encountering mysterious slowdowns.
- DoDs and Reliability Team pull their hair out and eat them for lunch

Fail-Fast Approach

```
public int maxConnections() {
   string property =
     getProperty("maxConnections");
   if (property == null) {
     throw new NullReferenceException
      ("maxConnections property not
           found in " +
           this.configFilePath);
   }
   else {
     return property.toInt();
   }
}
```

- DoDs and Reliability Team slap their foreheads unanimously and spend the next 30 seconds fixing the typo

Source: https://www.martinfowler.com/ieeeSoftware/failFast.pdf

When should we retry?

- Retry is part of a tool-kit of the fail safe approach (?)
- Transient failure:
  - Network outage
  - Unstable connection
  - Connection refused / Timeout
  - Server crashes
  - Server restarts
  - During failover or switchover (cluster, db, load balancer)
- Others:
  - Database deadlock
  - Flaky tests
- Most of these are not really developers' concerns
  - 🤞👉 Infrastructure and architecture teams 🍿
  - Also, built-in protocols, SDKs
- Except:
  - 3rd party APIs (timeout, 5xx HTTP status code)
  - File System (retryable IOException like timeout or connection exception)
  - Any other thing you can think of!

When should we retry?

- Whatever needs to be retried, make sure the operation is **idempotent**
  - Unique constraint violation in DB
  - POST method is often not idempotent, but GET is

2. What Is Idempotence?

Simply put, we can perform an idempotent operation multiple times without changing the result.

Furthermore, the operation must not cause any side effects after the first successful execution.

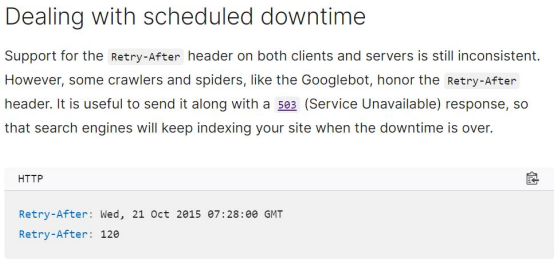
When should we retry?

- 3rd Party API
  - HTTP Status Code
    - Retry on any **5xx** code (server-side error)
    - Do not retry on **4xx code** (client-side error) except
      - **408 (timeout), 429 (too many requests)** but with an intelligent retry policy (backoff, jitter)
      - If expirable token is used, you can retry on **401 (authentication)**
  - If there is a **Retry-after** field in HTTP header
  - If there is no response at all or issue with connection
    - Subclasses of IOException under HTTP call that can be retried: **ConnectionTimeoutException, SocketTimeoutException, NoHttpResponseException, InterruptedIOException, HTTPRetryException**

The HttpClient class throws a ConnectTimeoutException Exception, so you should listen for it:

```
try {
        HttpResponse response = client.execute(post);
                        // do something with response
    } catch (ConnectTimeoutException e) {
        Log.e(TAG, "Timeout", e);
    } catch (SocketTimeoutException e) {
        Log.e(TAG, " Socket timeout", e);
    }
```

- Careful of acceptable wait-time for your user



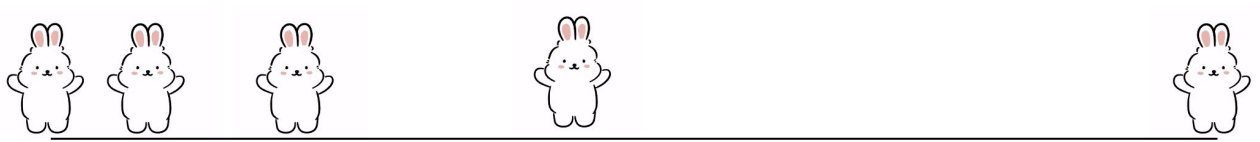
Theory: Backoff

Normal Retry

sec 0 1 2 3

Every 1 s

Exponential Retry



sec 0 1 3 7 15

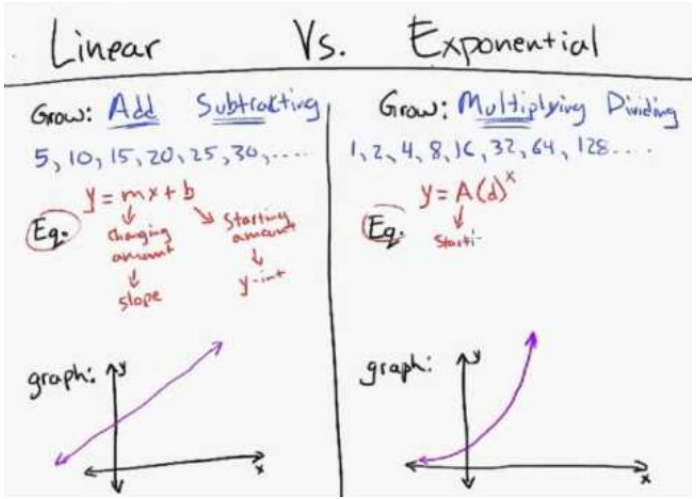
What should be the max bound?

1. Limit #rounds
2. Limit maximum duration e.g. 200ms (**Capped** Backoff Algorithm)

Increase by 1, 2, 4, 8, 16.. s

Increase by 1, 3, 9, 27, 81.. s

Theory: Backoff



Exponential backoff is a common strategy for handling retries of failed network calls. In simple terms, **the clients wait progressively longer intervals between consecutive retries**:

```
wait_interval = base * multiplier^n
```

where,

- *base* is the initial interval, ie, wait for the first retry
- *n* is the number of failures that have occurred
- *multiplier* is an arbitrary multiplier that can be replaced with any suitable value

With this approach, we provide a breathing space to the system to recover from intermittent failures, or even more severe problems.

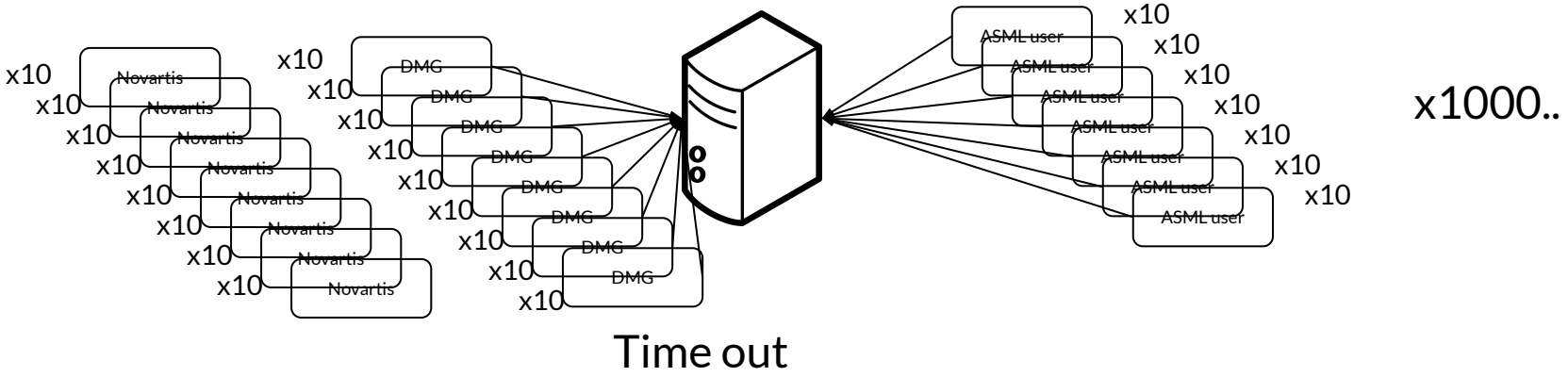
Interval	=	5 * 3^0 = 5	Interval	=	2 * 5^0 = 2
		5 * 3^1 = 15			2 * 5^1 = 10
		5 * 3^2 = 45			2 * 5^2 = 50
		5 * 3^3 = 135			2 * 5^3 = 250

Source: https://www.baeldung.com/resilience4j-backoff-jitter

Theory: Backoff

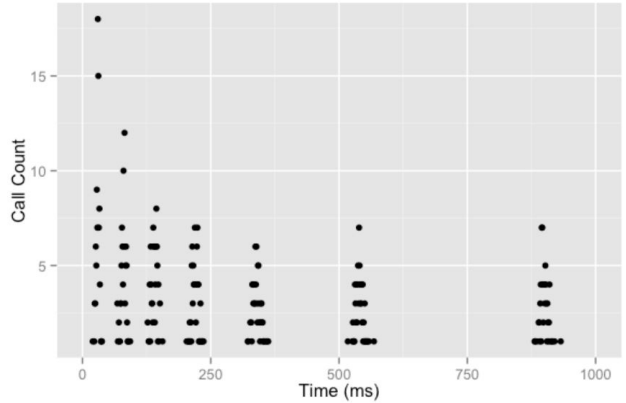
- Does the backoff algorithm help much?
  - As a service caller,
    is probably good enough
  - What about as service provider?

Increase by 1, 2, 4, 8, 16..

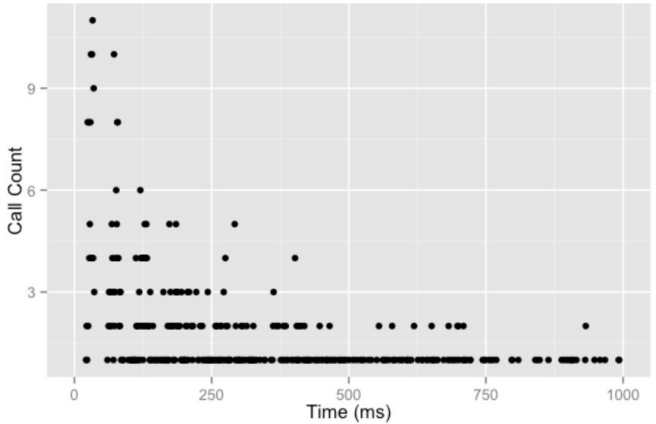


Theory: Jitter

Exponential Backoff without Jitter



Exponential Backoff with Jitter



Retry interval

= min(**base * (2 ^ n),** 200ms)

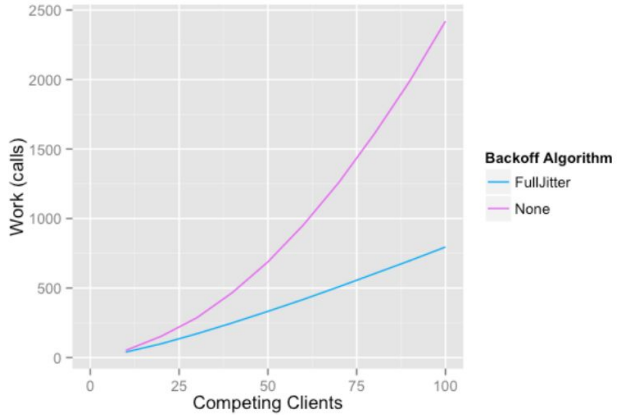
= **random(0**, min(**base * (2 ^ n),** 200ms)**)**

Capped at 200ms

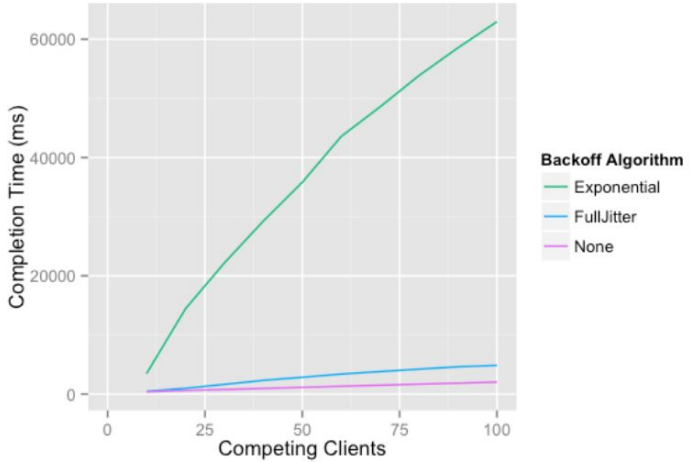
Source: https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/

Theory: Jitter

- Reduces *both* client calls and their completion time



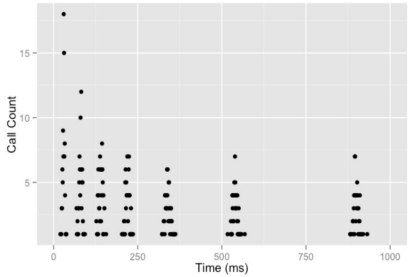
In the case with 100 contending clients, we've reduced our call count by more than half. We've also significantly improved the time to completion, when compared to un-jittered exponential backoff.



Source: https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/

Theory: Jitter

No Jitter

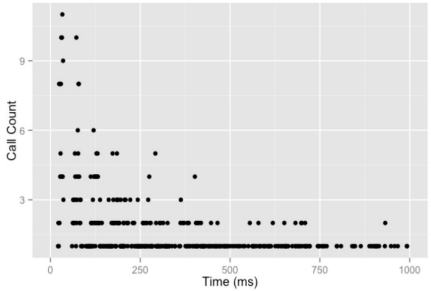


Retry interval

= min(**base * (2 ^ n),** 200ms)

Capped at 200ms

Full Jitter



= **random(0**, min(**base * (2 ^ n),** 200ms)**)**

Actually, this is worse in performance than "Full Jitter" approach

Equal Jitter

= min(**base * (2 ^ n),** 200ms) / 2 **+random(0,** min(**base * (2 ^ n),** 200ms) / 2 **)**

= x / 2 + random (0, x/2)

* value will cluster around 0,2,4,8,16 with some variance

Source: https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/

Theory: Adaptive Retry

Available retry modes

Legacy retry mode

Legacy mode is the default mode used by any Boto3 client you create. As its name implies, `legacy mode` uses an older (v1) retry handler that has limited functionality.

Legacy mode's functionality includes:

- A default value of 5 for maximum retry attempts. This value can be overwritten through the `max_attempts` configuration parameter.
- Retry attempts for a limited number of errors/exceptions:

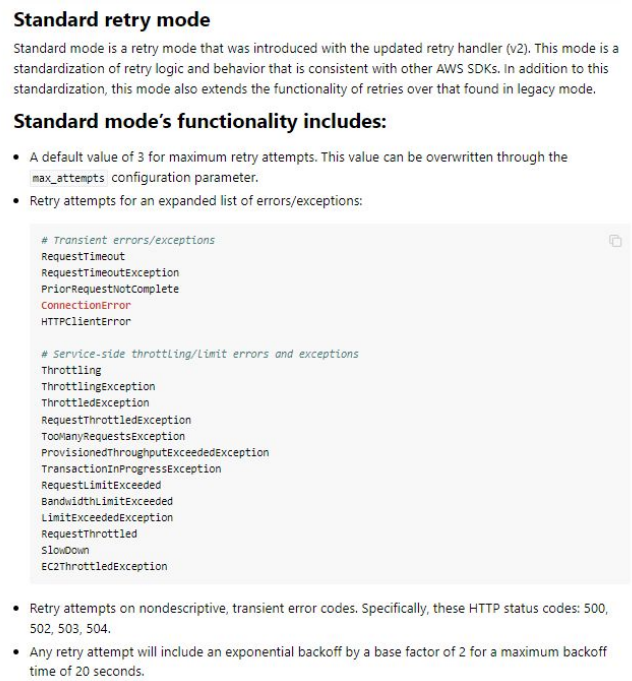
```
# General socket/connection errors
ConnectionError
ConnectionClosedError
ReadTimeoutError
EndpointConnectionError

# Service-side throttling/limit errors and exceptions
Throttling
ThrottlingException
ThrottledException
RequestThrottledException
ProvisionedThroughputExceededException
```

- Retry attempts on several HTTP status codes, including 429, 500, 502, 503, 504, and 509.
- Any retry attempt will include an exponential backoff by a base factor of 2.

- Legacy mode
  - Exponential backoff
  - 5 retries, no time cap
  - Retry on:
    - HTTP Status Codes
      - 429,500,502,503, 504,509
    - Connection error
    - Throttling error

Source: https://boto3.amazonaws.com/v1/documentation/api/latest/guide/retries.html

Theory: Adaptive Retry



Standard retry mode

Standard mode is a retry mode that was introduced with the updated retry handler (v2). This mode is a standardization of retry logic and behavior that is consistent with other AWS SDKs. In addition to this standardization, this mode also extends the functionality of retries over that found in legacy mode.

Standard mode's functionality includes:

- A default value of 3 for maximum retry attempts. This value can be overwritten through the `max_attempts` configuration parameter.
- Retry attempts for an expanded list of errors/exceptions:

```
# Transient errors/exceptions
RequestTimeout
RequestTimeoutException
PriorRequestNotComplete
ConnectionError
HTTPClientError

# Service-side throttling/limit errors and exceptions
Throttling
ThrottlingException
ThrottledException
RequestThrottledException
TooManyRequestsException
ProvisionedThroughputExceededException
TransactionInProgressException
RequestLimitExceeded
BandwidthLimitExceeded
LimitExceededException
RequestThrottled
SlowDown
EC2ThrottledException
```

- Retry attempts on nondescriptive, transient error codes. Specifically, these HTTP status codes: 500, 502, 503, 504.
- Any retry attempt will include an exponential backoff by a base factor of 2 for a maximum backoff time of 20 seconds.

- Standard mode for all SDKs
  - Exponential backoff
  - **3 retries, time cap at 20 s**
  - Retry on:
    - HTTP Status Codes
      - 500,502,503,504
    - Connection error
    - Throttling error
      - More Exceptions included

Source: https://boto3.amazonaws.com/v1/documentation/api/latest/guide/retries.html

Theory: Adaptive Retry

Adaptive retry mode

Adaptive retry mode is an experimental retry mode that includes all the features of standard mode. In addition to the standard mode features, adaptive mode also introduces client-side rate limiting through the use of a token bucket and rate-limit variables that are dynamically updated with each retry attempt. This mode offers flexibility in client-side retries that adapts to the error/exception state response from an AWS service.

With each new retry attempt, adaptive mode modifies the rate-limit variables based on the error, exception, or HTTP status code presented in the response from the AWS service. These rate-limit variables are then used to calculate a new call rate for the client. Each exception/error or non-success HTTP response (provided in the list above) from an AWS service updates the rate-limit variables as retries occur until success is reached, the token bucket is exhausted, or the configured maximum attempts value is reached.

Note

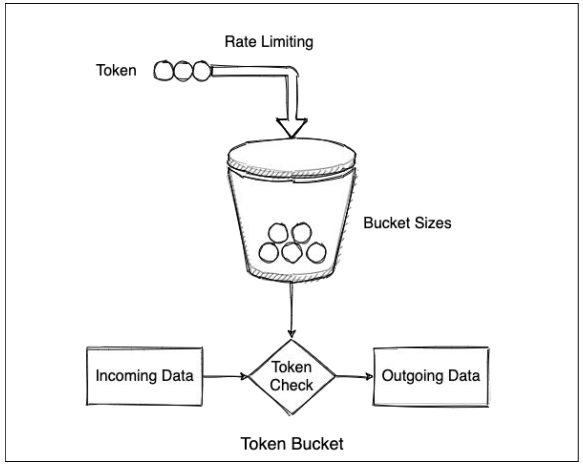
Adaptive mode is an experimental mode and is subject to change, both in features and behavior.

Experimental Feature

- ~~Exponential backoff~~
- **Re-calculates the retry interval after each attempt**
  - Based on HTTP Status Codes, type of connection or throttling error
- Uses **“Token bucket” retry**

Source: https://boto3.amazonaws.com/v1/documentation/api/latest/guide/retries.html

Theory: Adaptive Retry



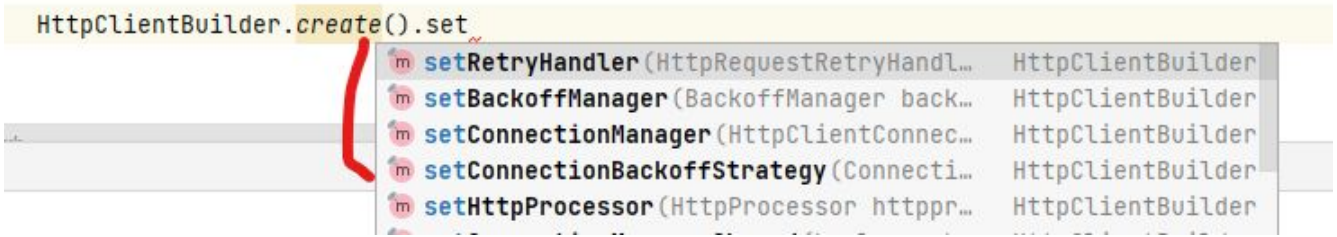
- "Token bucket" Algorithm
  - Method:
    - + 0.1 token if success
    - -1 token if fail/retry
    - Stop retrying if bucket is empty
  - Normal Retry if failure rate is low (for N times), but fewer retries (fewer than N times) if failure rate is high

Is bucket shared among users for that particular call/method/endpoint? Or one giant bucket for an entire service?

- Used as Rate-Limiting/Throttling algorithm

Methods: Apache Commons

- Apache Commons HTTP Client has some useful methods



Methods: Apache Commons

- RetryHandler

```
} catch (IOException ex) {
    this.log.debug("Closing the connection.");
    managedConn.close();
    if (retryHandler.retryRequest(ex, execCount, context)) {
        if (this.log.isInfoEnabled()) {
            this.log.info("I/O exception ("+ ex.getClass().getName() +
                    ") caught when processing request: "
                    + ex.getMessage());
        }
        if (this.log.isDebugEnabled()) {
            this.log.debug(ex.getMessage(), ex);
        }
        this.log.info("Retrying request");
    } else {
        throw ex;
    }
```

robovm / rt / external / apache-http / src / org / apache / http / impl / client / DefaultHttpRequestRetryHandler.java ↑ Top

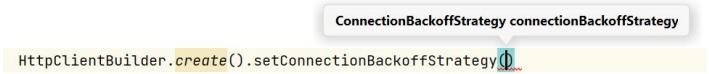
Code | Blame | 134 lines (123 loc) · 4.66 KB | Raw

```
51  public class DefaultHttpRequestRetryHandler implements HttpRequestRetryHandler {
78      public boolean retryRequest(
79              final IOException exception,
80              int executionCount,
81              final HttpContext context) {
82          if (exception == null) {
83              throw new IllegalArgumentException("Exception parameter may not be null");
84          }
85          if (context == null) {
86              throw new IllegalArgumentException("HTTP context may not be null");
87          }
88          if (executionCount > this.retryCount) {
89              // Do not retry if over max retry count
90              return false;
91          }
92          if (exception instanceof NoHttpResponseException) {
93              // Retry if the server dropped connection on us
94              return true;
95          }
96          if (exception instanceof InterruptedIOException) {
97              // Timeout
98              return false;
99          }
100         if (exception instanceof UnknownHostException) {
101             // Unknown host
102             return false;
103         }
104         if (exception instanceof SSLHandshakeException) {
105             // SSL handshake exception
106             return false;
107         }
```

Code from:
https://github.com/robovm/robovm/blob/~~master/rt/external/apache-http/src/org/apache/http/impl/client/DefaultHttpRequestRetryHandler.java~~

Methods: Apache Commons

- ConnectionBackoffStrategy




```java
public class DefaultBackoffStrategy implements ConnectionBackoffStrategy {

    @Override
    public boolean shouldBackoff(final Throwable t) {
        return (t instanceof SocketTimeoutException
                || t instanceof ConnectException);
    }

    @Override
    public boolean shouldBackoff(final HttpResponse resp) {
        return (resp.getStatusLine().getStatusCode() == HttpStatus.SC_SERVICE_UNAVAILABLE);
    }

}
```

Code from:
https://github.com/ibinti/bugvm/blob/master/Core/rt/src/main/java/org/apache/http/impl/client/DefaultBackoffStrategy.java

Methods: Spring-Retry

- Spring-Retry now supports a form of jitter/variance
- Official Documentation:
  - Example: initialInterval = 50 multiplier = 2.0 maxInterval = 3000 numRetries = 5
    - ***ExponentialBackOffPolicy*** yields: [50, 100, 200, 400, 800]
    - ***ExponentialRandomBackOffPolicy*** may yield [76, 151, 304, 580, 901] or [53, 190, 267, 451, 815]
  - ~ Equal Jitter

Source: https://docs.spring.io/spring-retry/docs/api/current/org/springframework/retry/backoff/ExponentialRandomBackOffPolicy.html

Methods: Spring-Retry

```java
/**
 * Retrieve access token, cached or renewed
 * @param url endpoint to retrieve or renew token
 * @param clientId clientId to be included in the payload of HTTP request
 * @param clientSecret clientSecret to be included in the payload of HTTP request
 * @return access token as String
 * @throws JWTAuthorizationException any exception that occurs during token request, including Marken API error e.g.
 *      * failed authentication, other non-ok status code, or other possible exceptions e.g. IOException
 */
2 usages  Lohavittayavikant Salisa *  2 related problems
@Retryable(
    value = {JWTAuthorizationException.class},
    maxAttempts = 5,
    backoff = @Backoff(random = true, delay = 200, maxDelay = 5000, multiplier = 2)
)
public static String getAccessToken(String url, String clientId, String clientSecret)
            throws JWTAuthorizationException {
    if (_magrpService == null) {
        init();
    }

    // re-initialise the map here
    Map<String, String> payload = new HashMap<>();
```

Retry interval = base * multiplier ^n
= 200 * 2^n

Source: https://docs.spring.io/spring-retry/docs/api/current/org/springframework/retry/backoff/ExponentialRandomBackOffPolicy.html

Methods: Spring-Retry

- @Retryable
- @Recover
  - Optional, not necessary
  - Called when your retry is maxed out
- RetryPolicy, RetryTemplate as its own separate class
- See more: https://www.baeldung.com/spring-retry

Methods: inet.common.util.Retry

- We have a Retry class in /common module

```
19   * @param <T> type of exception that shall be caught for retrying
20   * @author Martin.Fey
21   */
     12 usages  Martin.Fey +2
22  public final class Retry<T> {
23
         2 usages
24       private static final Logger LOGGER = LoggerFactory.getLogger(Retry.class);
         1 usage
25       private static final double INCREASE_FACTOR = 1.1d;
         4 usages
26       private final int times;
         3 usages
27       private final Class<T> exceptionClass;
         2 usages
28       private final Duration pause;
29
```

```
for (SendungPO sendung: load.getSendungen()) { // process all TOs assigned to the load
    final boolean success = Retry
                                .on(GWDBException.class)
                                .pause(Duration.ofSeconds(0))
                                .times(3)
                                .run(() -> handle(loadAufId, sendung, ladListId, action), doLogging: true);
    if (!success) {
        handleRetryError(loadAufId, sendung.getSendungNrDfue(), action, messages);
        return false;
    }
```

When and why should we NOT retry?

- Or rather, what are the caveats and pitfalls?
- 1. “Retry Storm”
  - 3-deep stack
    - 1st layer: 3 retries
    - 2nd layer: 10 retries
    - 3rd layer: 3 retries
    - 4th layer - 3rd Party API: 5 retries

    3 x 10 x 3 = 90 retries!!
  - Solution: make sure the retry is done at one or few layers as possible
- 2. Whether we DDoS ourselves or 3rd party services, we put the system under more load + add latency
  - So..
    - Worsens the recovery time
    - Impacts users
  - Solution:
    - Implement intelligent backoff algorithm or adaptive retry
    - Throttle retry as you would server traffic
- 3. Others:
  - Make sure the operation is idempotent - or no side effect when retried
  - Only retry when necessary e.g. connection error not all IOExceptions
  - Should we fail-safe or or fail-fast?

Resilience4J: Circuit Breaker

“For example, when Shopify’s Redis, that stores sessions, is down, the user doesn’t have to see an error. Instead, the problem is logged and the page renders with sessions soft disabled. This results in a much better customer experience. This behaviour is achievable in many cases, however, it’s not as simple as catching exceptions that are raised by a failing service.

Imagine Redis is down and every connection attempt is timing out. Each timeout is 2 seconds long. Response times will be incredibly slow, since requests are waiting for the service to timeout. Additionally, during that time the request is doing nothing useful and will keep the thread busy.

A worker which had a request processing rate of 5 requests per second now can only process half a request per second. That’s a tenfold decrease in throughput! With utilization this high, the service can be considered completely down. This is unacceptable for production level standards.”

Source: https://shopify.engineering/circuit-breaker-misconfigured

Depending on the details of the system, the circuit breaker may track different types of failures separately. For example, you may choose to have a lower threshold for "timeout calling remote system" failures than "connection refused" errors.

When the circuit breaker is open, something has to be done with the calls that come in. The easiest answer would be for the calls to immediately fail, perhaps by throwing an exception (preferably a different exception than an ordinary timeout so that the caller can provide useful feedback). A circuit breaker may also have a "fallback" strategy. Perhaps it returns the last good response or a cached value. It may return a generic answer rather than a personalized one. Or it may even call a secondary service when the primary is not available.

Circuit breakers are effective at guarding against integration points, cascading failures, unbalanced capacities, and slow responses. They work so closely with timeouts that they often track timeout failures separately from execution failures.

Remember This

Don't do it if it hurts.

Circuit Breaker is the fundamental pattern for protecting your system from all manner of Integration Points problems. When there's a difficulty with Integration Points, stop calling it!

Use together with Timeouts.

Circuit Breaker is good at avoiding calls when Integration Points has a problem. The Timeouts pattern indicates that there's a problem in Integration Points.

Expose, track, and report state changes.

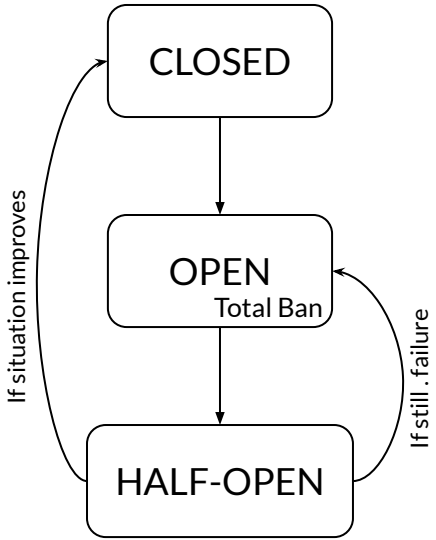
Popping a Circuit Breaker *always* indicates something abnormal. It should be visible to Operations. It should be reported, recorded, trended, and correlated.

Resilience4J: Circuit Breaker

- If an external service is slow or down
  - Should we let all our users time out?
  - Or fail right away, until situation improves?

Source: "Release It!" book by Michael Nygard

Resilience4J: Circuit Breaker



- States – Circuit Breaker is
  - CLOSED : No failure or acceptable rate of failure (< threshold)
  - OPEN : Unacceptable rate of failure (> threshold)
    - Total Ban is imposed for X milliseconds/seconds/minutes
  - HALF-OPEN: Calls are throttled
    - Allows fewer calls to be executed to feel out the situation + update the statistics
- What is considered failure?
  - Exception
  - Timeout/Slowness

COUNT-BASED = if 3 out of the **last 10 requests** fail (30% failure)
TIME-BASED = if 30% of the requests fail in the **last 10 seconds**

```
// Create a custom configuration for a CircuitBreaker                    ⚠ 2
var circuitBreakerConfig : CircuitBreakerConfig = CircuitBreakerConfig.custom()

        .slidingWindowType(SlidingWindowType.COUNT_BASED) // or TIME_BASED
        .slidingWindowSize(10) // last 10 calls - if TIME_BASED, window size = last 10 seconds

        // Configure Failure
        .failureRateThreshold(30) // trip open if > 30% fails
        .recordExceptions(TemporaryFailureException.class) // included exceptions
        .ignoreExceptions(ParseException.class, // excluded exceptions
                          IllegalStateException.class,
                          IOException.class) // if not explicitly included here, will consider a failure


        // Configure Slow call
        .slowCallRateThreshold(50) // trip open if > 50% is slow
        .slowCallDurationThreshold(Duration.ofSeconds(1)) // slow if > 1s

        // Configure Total Ban Duration
        .waitDurationInOpenState(Duration.ofSeconds(2)) // let's impose 2s of total ban

        // Configure Transition
        .permittedNumberOfCallsInHalfOpenState(10) // allow 10 calls in risky situation
        .build();
```

1. Wrap with Circuit Breaker

```java
var response : CloseableHttpResponse  = circuitBreaker.executeCheckedSupplier(CircuitBreaker::getCountries);
```

2. Use @CircuitBreaker annotation

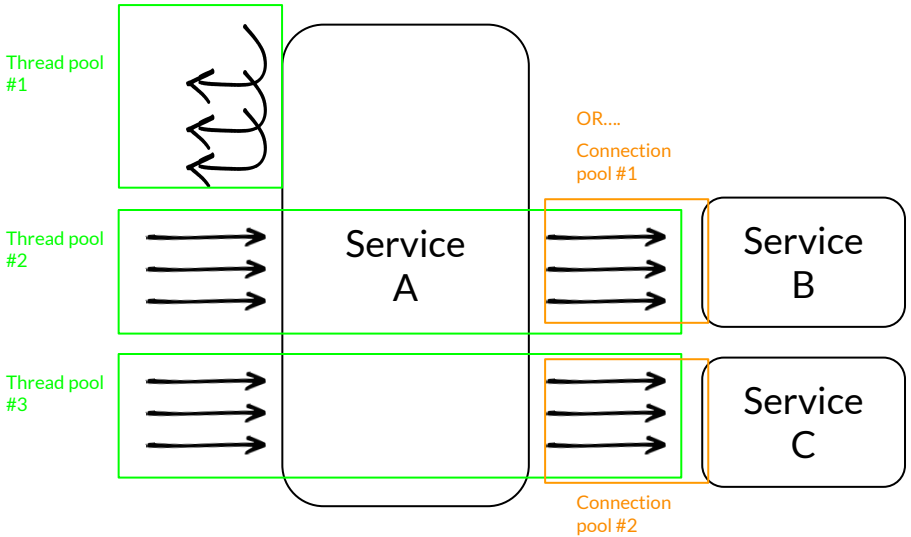
```java
1 usage
@CircuitBreaker(name="CountryService", fallbackMethod="getFromCache")
private CloseableHttpResponse httpGetHelper(String URL, RequestConfig reqConfig)
                                            throws TemporaryFailureException, IOException {
    final HttpGet request = new HttpGet(URL);
    try (CloseableHttpClient client = HttpClientBuilder.create().setDefaultRequestConfig(reqConfig).build())
        var response : CloseableHttpResponse  = client.execute(request);
```

resilience4j [:CircuitBreaker.main()]

```
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
HttpResponseProxy{HTTP/1.1 200 OK [Date: Mon, 22 Jan 2024 04:12:18 GMT, Server: Apache/2.4.38 (Debian), Cache-Control: public, immutable, max-a
throws TemporaryFailureException
HttpResponseProxy{HTTP/1.1 200 OK [Date: Mon, 22 Jan 2024 04:12:19 GMT, Server: Apache/2.4.38 (Debian), Cache-Control: public, immutable, max-a
HttpResponseProxy{HTTP/1.1 200 OK [Date: Mon, 22 Jan 2024 04:12:20 GMT, Server: Apache/2.4.38 (Debian), Cache-Control: public, immutable, max-a
throws TemporaryFailureException
io.github.resilience4j.circuitbreaker.CallNotPermittedException: CircuitBreaker 'CountryService' is OPEN and does not permit further calls
io.github.resilience4j.circuitbreaker.CallNotPermittedException: CircuitBreaker 'CountryService' is OPEN and does not permit further calls
io.github.resilience4j.circuitbreaker.CallNotPermittedException: CircuitBreaker 'CountryService' is OPEN and does not permit further calls
io.github.resilience4j.circuitbreaker.CallNotPermittedException: CircuitBreaker 'CountryService' is OPEN and does not permit further calls
io.github.resilience4j.circuitbreaker.CallNotPermittedException: CircuitBreaker 'CountryService' is OPEN and does not permit further calls
io.github.resilience4j.circuitbreaker.CallNotPermittedException: CircuitBreaker 'CountryService' is OPEN and does not permit further calls
io.github.resilience4j.circuitbreaker.CallNotPermittedException: CircuitBreaker 'CountryService' is OPEN and does not permit further calls
io.github.resilience4j.circuitbreaker.CallNotPermittedException: CircuitBreaker 'CountryService' is OPEN and does not permit further calls
io.github.resilience4j.circuitbreaker.CallNotPermittedException: CircuitBreaker 'CountryService' is OPEN and does not permit further calls
io.github.resilience4j.circuitbreaker.CallNotPermittedException: CircuitBreaker 'CountryService' is OPEN and does not permit further calls
```

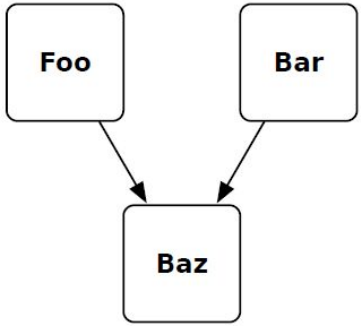
Resilience4J: Bulkhead Isolation

Resilience4J: Bulkhead Isolation



- Allocate/partition resources (thread pool, connection pool, etc.) for
  - A
  - A -> B
  - A -> C
- So that fault in one part, does not end up blocking the functioning of the rest of A
- If it makes sense to isolation these resources, otherwise unnecessary complexity

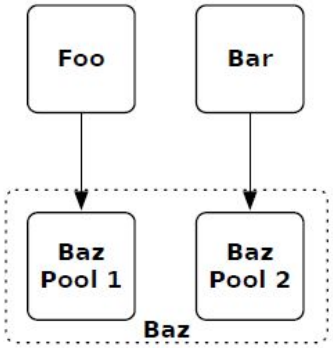
In the figure that follows, Foo and Bar both use the enterprise service Baz. Because both depend on a common service, each system has some vulnerability to the other. If Foo suddenly gets crushed under user load, goes rogue because of some defect, or triggers a bug in Baz, Bar—and its users—also suffer. This kind of unseen coupling makes diagnosing problems (particularly performance problems) in Bar very difficult. Scheduling maintenance windows for Baz also requires coordination with both Foo and Bar, and it may be difficult to find a window that works for both clients.



Assuming both Foo and Bar are critical systems with strict SLAs, it'd be safer to partition Baz, as shown in this revised figure on page 100. Dedicating some capacity to each critical client removes most of the hidden linkage. They probably still share a database and are, therefore, subject to deadlocks across instances, but that's another antipattern.

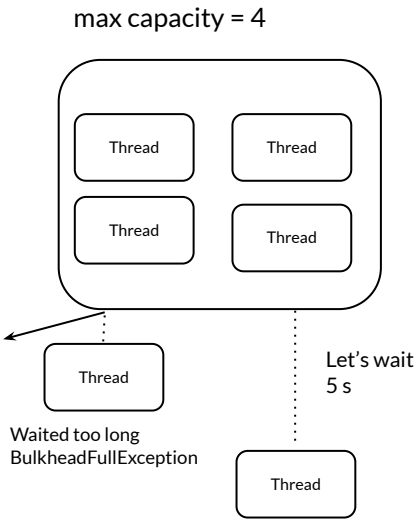
Source: "Release It!" book by Michael Nygard

Of course, it would be better to preserve all capabilities. Assuming that failures will occur, however, you must consider how to minimize the damage caused by a failure. It is not an easy effort, and one rule cannot apply in every case. Instead, you must examine the impact to the business of each loss of capability and



cross-reference those impacts against the architecture of the systems. The goal is to identify the natural boundaries that let you partition the system in a way that is both technically feasible and financially beneficial. The boundaries of this partitioning may be aligned with the callers, with functionality, or with the topology of the system.

Resilience4J: Bulkhead Isolation



- Two types of Bulkhead Implementations
  - Semaphore Bulkhead
    - A limited pool of concurrent users/threads that for making a call to the external service
    - If the pool is exhausted, wait for X milliseconds/seconds/minutes until a semaphore is released by another thread (first-come, first-serve basis)
    - If > max wait time, a BulkheadFullException is thrown

Create and configure a Bulkhead

You can provide a custom global BulkheadConfig. In order to create a custom global BulkheadConfig, you can use the BulkheadConfig builder. You can use the builder to configure the following properties.

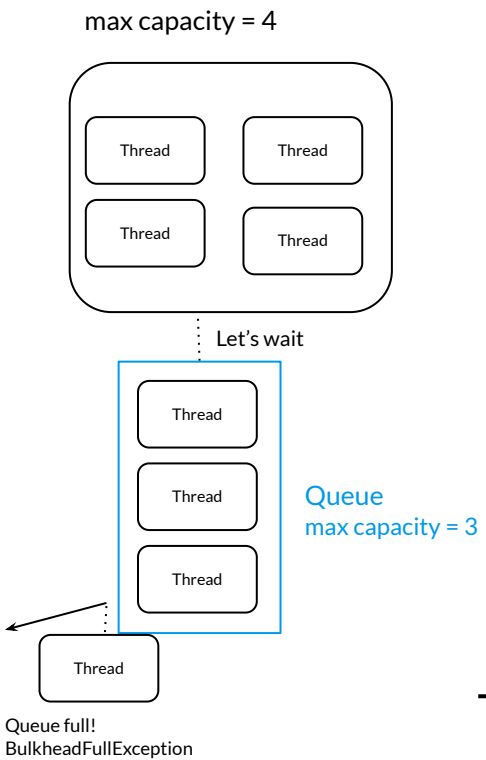
Config property	Default value	Description
maxConcurrentCalls	25	Max amount of parallel executions allowed by the bulkhead
maxWaitDuration	0	Max amount of time a thread should be blocked for when attempting to enter a saturated bulkhead.

Java

```
// Create a custom configuration for a Bulkhead
BulkheadConfig config = BulkheadConfig.custom()
        .maxConcurrentCalls(150)
        .maxWaitDuration(Duration.ofMillis(500))
        .build();
```

doc: https://resilience4j.readme.io/docs/bulkhead

Resilience4J: Bulkhead Isolation



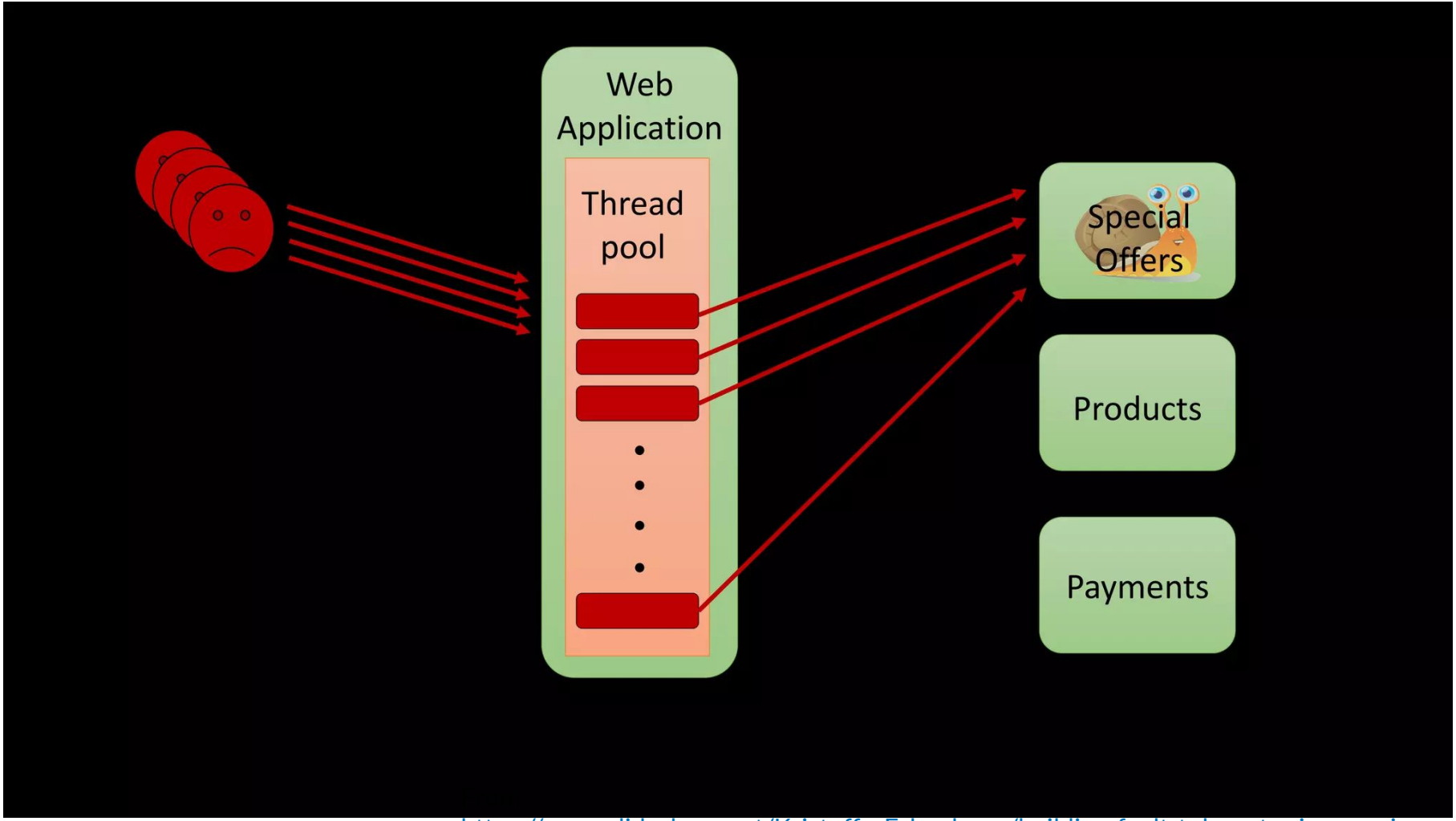
- Two types of Bulkhead Implementations
  - ThreadPool Bulkhead
    - ThreadPool + Queue
    - If both full, BulkheadFullException is thrown

Config property	Default value	Description
maxThreadPoolSize	Runtime.getRuntime().availableProcessors()	Configures the max thread pool size.
coreThreadPoolSize	Runtime.getRuntime().availableProcessors() - 1	Configures the core thread pool size
queueCapacity	100	Configures the capacity of the queue.
keepAliveDuration	20 [ms]	When the number of threads is greater than the core, this is the maximum time that excess idle threads will wait for new tasks before terminating.
writableStackTraceEnabled	true	Output the stack trace error when a bulkhead exception is thrown. If false, output a single line with bulkhead exception.

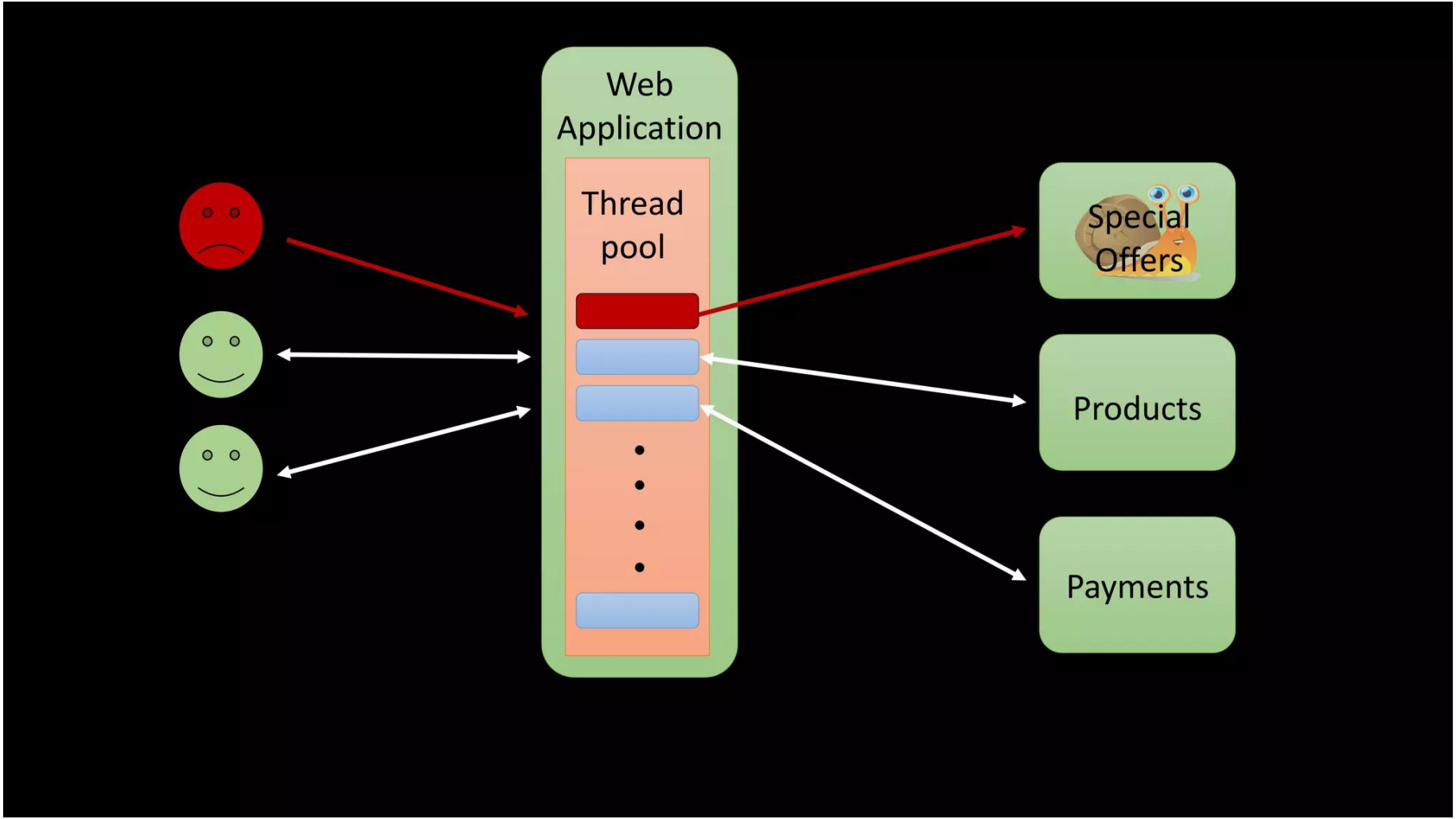
Java

```
ThreadPoolBulkheadConfig config = ThreadPoolBulkheadConfig.custom()
  .maxThreadPoolSize(10)
  .coreThreadPoolSize(2)
  .queueCapacity(20)
  .build();
```

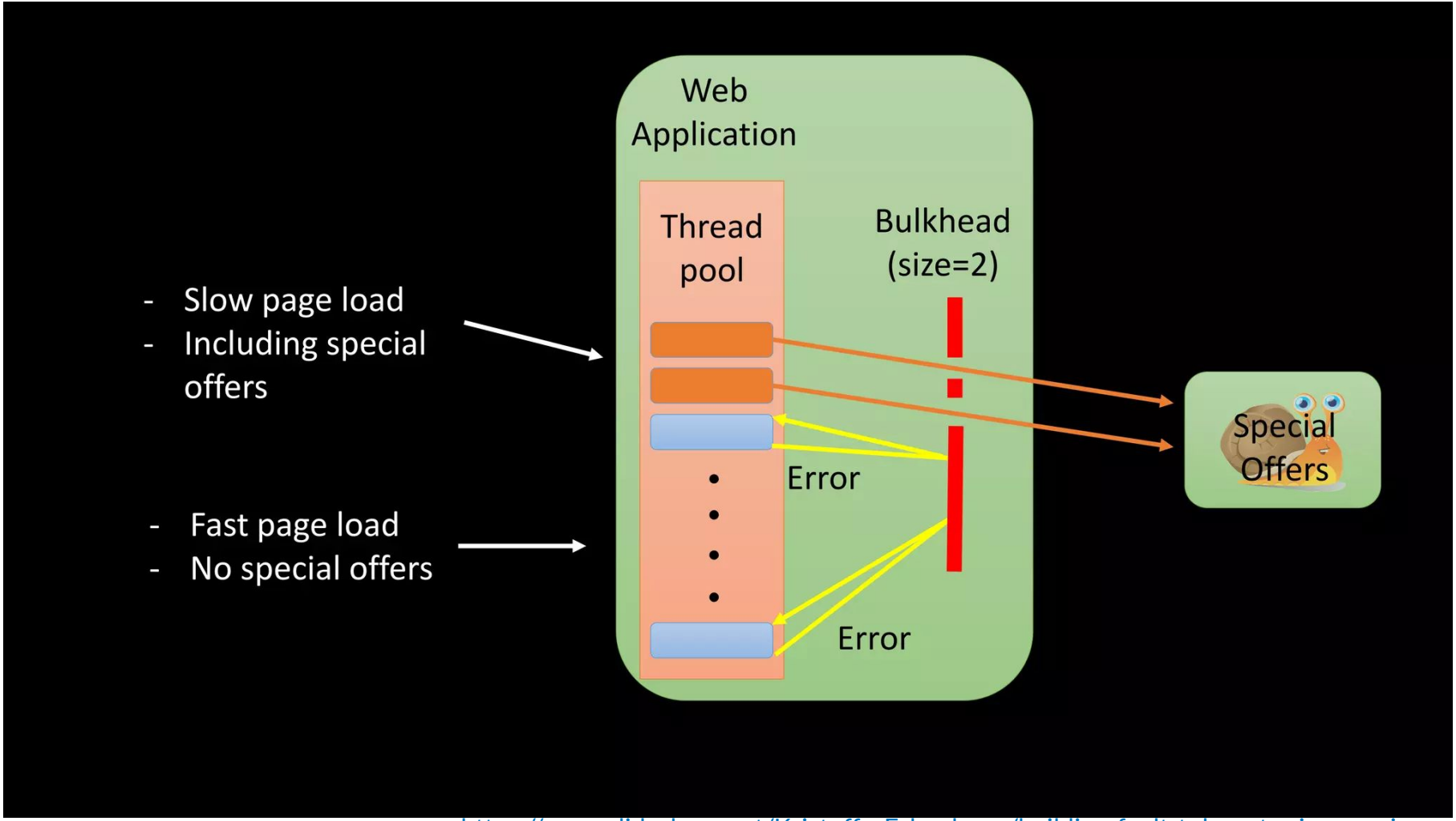
doc: https://resilience4j.readme.io/docs/bulkhead



https://www.slideshare.net/KristofferErlandsson/building-fault-tolerant-microservices

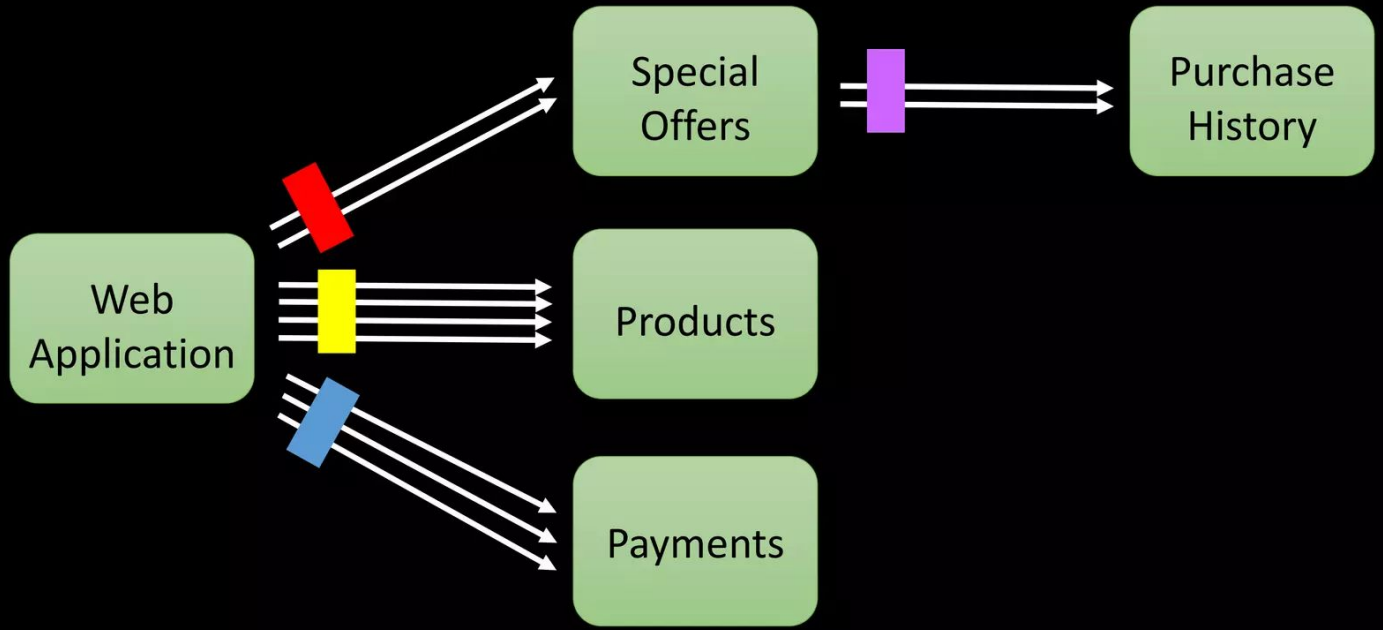


https://www.slideshare.net/KristofferErlandsson/building-fault-tolerant-microservices

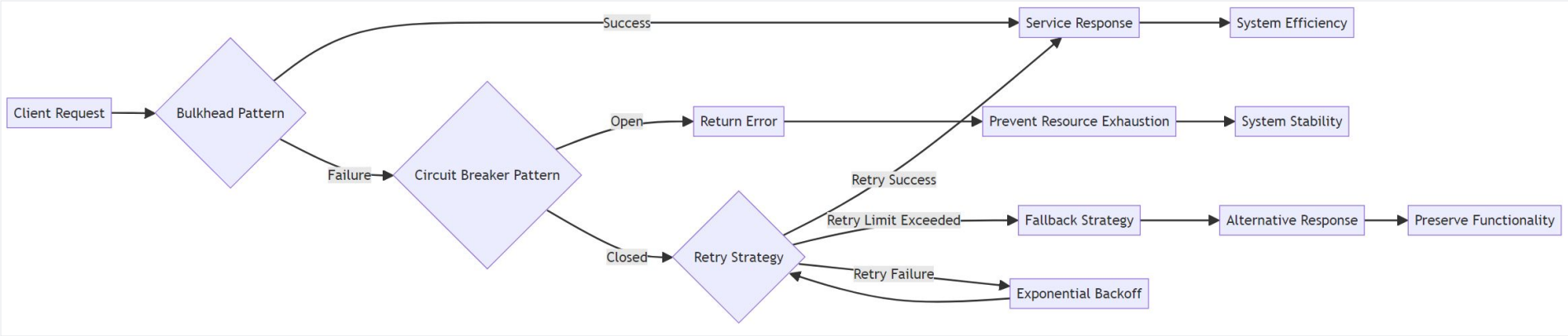


https://www.slideshare.net/KristofferErlandsson/building-fault-tolerant-microservices

One bulkhead per service



https://www.slideshare.net/KristofferErlandsson/building-fault-tolerant-microservices



Integrating Retry strategies with the Bulkhead pattern boosts fault tolerance by providing superior fault handling and improving application resilience and availability in the face of transient failures. These strategies involve implementing retry logic within bulkheads to handle transient failures and preserve functionality. They also include the exponential backoff strategy, which progressively increases the delay between attempts, mitigating system overload and facilitating recovery from temporary outages.

Source: https://im5tu.io/learn/cloud-patterns/bulkhead/