# Introduction

[**SUGGEST EDITS**](https://resilience4j.readme.io/docs)

Resilience4j is a lightweight, easy-to-use fault tolerance library inspired by  
[Netflix Hystrix](https://github.com/Netflix/Hystrix), but designed for Java 8 and functional programming. Lightweight, because the library only uses [Vavr](http://www.vavr.io/), which does not have any other external library dependencies. Netflix Hystrix, in contrast, has a compile dependency to Archaius which has many more external library dependencies such as Guava and Apache Commons Configuration.

Resilience4j provides higher-order functions (decorators) to enhance any functional interface, lambda expression or method reference with a Circuit Breaker, Rate Limiter, Retry or Bulkhead. You can stack more than one decorator on any functional interface, lambda expression or method reference. The advantage is that you have the choice to select the decorators you need and nothing else.

With Resilience4j you don’t have to go all-in, you can pick what you need.

## Sneak preview

The following example shows how to decorate a lambda expression with a CircuitBreaker and Retry in order to retry the call at most 3 times when an exception occurs.  
You can configure the wait interval between retries and also configure a custom backoff algorithm.  
The example uses Vavr’s Try monad to recover from an exception and invoke another lambda expression as a fallback, when all retries have failed.

// Create a CircuitBreaker with default configuration

CircuitBreaker circuitBreaker = CircuitBreaker

.ofDefaults("backendService");

// Create a Retry with default configuration

// 3 retry attempts and a fixed time interval between retries of 500ms

Retry retry = Retry

.ofDefaults("backendService");

// Create a Bulkhead with default configuration

Bulkhead bulkhead = Bulkhead

.ofDefaults("backendService");

Supplier<String> supplier = () -> backendService

.doSomething(param1, param2)

// Decorate your call to backendService.doSomething()

// with a Bulkhead, CircuitBreaker and Retry

// \*\*note: you will need the resilience4j-all dependency for this

Supplier<String> decoratedSupplier = Decorators.ofSupplier(supplier)

.withRetry(retry)

.withCircuitBreaker(circuitBreaker)

.withBulkhead(bulkhead)

.decorate();

// Execute the decorated supplier and recover from any exception

String result = Try.ofSupplier(decoratedSupplier)

.recover(throwable -> "Hello from Recovery").get();

// When you don't want to decorate your lambda expression,

// but just execute it and protect the call by a CircuitBreaker.

String result = circuitBreaker

.executeSupplier(backendService::doSomething);

## Modularization

With Resilience4j you don't have to go all-in, you can pick what you need.

Resilience provides several core modules and add-on modules:

# Comparison to Netflix Hystrix

[**SUGGEST EDITS**](https://resilience4j.readme.io/docs/comparison-to-netflix-hystrix)

To highlight a few differences to Netflix Hystrix:

* In Hystrix calls to external systems have to be wrapped in a HystrixCommand. This library, in contrast, provides higher-order functions (decorators) to enhance any functional interface, lambda expression or method reference with a Circuit Breaker, Rate Limiter or Bulkhead. Furthermore, the library provides decorators to retry failed calls or cache call results. You can stack more than one decorator on any functional interface, lambda expression or method reference. That means, you can combine a Bulkhead, RateLimiter and Retry decorator with a CircuitBreaker decorator. The advantage is that you have the choice to select the decorator you need and nothing else. Any decorated function can be executed synchronously or asynchronously by using a CompletableFuture or RxJava.
* Hystrix, by default, stores execution results in 10 1-second window buckets. If a 1-second window bucket is passed, a new bucket is created and the oldest is dropped. This library stores execution results in Ring Bit Buffer without a statistical rolling time window. A successful call is stored as a 0 bit and a failed call is stored as a 1 bit. The Ring Bit Buffer has a configurable fixed-size and stores the bits in a long[] array which is saving memory compared to a boolean array. That means the Ring Bit Buffer only needs an array of 16 long (64-bit) values to store the status of 1024 calls. The advantage is that this CircuitBreaker works out-of-the-box for low and high frequency backend systems, because execution results are not dropped when a time window is passed.
* Hystrix only performs a single execution when in half-open state to determine whether to close a CircuitBreaker. This library allows to perform a configurable number of executions and compares the result against a configurable threshold to determine whether to close a CircuitBreaker.
* This library provides custom Reactor or RxJava operators to decorate any reactive type with a Circuit Breaker, Bulkhead or Ratelimiter.
* Hystrix and this library emit a stream of events which are useful to system operators to monitor metrics about execution outcomes and latency.

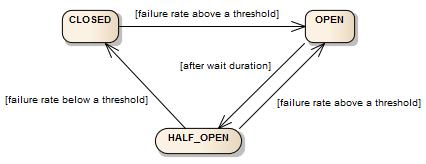
# CircuitBreaker

Getting started with resilience4j-circuitbreaker

[**SUGGEST EDITS**](https://resilience4j.readme.io/docs/circuitbreaker)

## Introduction

The CircuitBreaker is implemented via a finite state machine with three normal states: CLOSED, OPEN and HALF\_OPEN and two special states DISABLED and FORCED\_OPEN.

[](https://files.readme.io/39cdd54-state_machine.jpg)

The CircuitBreaker uses a sliding window to store and aggregate the outcome of calls. You can choose between a count-based sliding window and a time-based sliding window. The count-based sliding window aggregrates the outcome of the last N calls. The time-based sliding window aggregrates the outcome of the calls of the last N seconds.

## Count-based sliding window

The count-based sliding window is implemented with a circular array of N measurements.  
If the time window size is 10, the circular array has always 10 measurements.  
The sliding window incrementally updates a total aggregation. The total aggregation is updated when a new call outcome is recorded. When the oldest measurement is evicted, the measurement is subtracted from the total aggregation and the bucket is reset. (Subtract-on-Evict)

The time to retrieve a Snapshot is constant 0(1), since the Snapshot is pre-aggregated and is independent of the window size.  
The space requirement (memory consumption) of this implementation should be O(n).

## Time-based sliding window

The time-based sliding window is implemented with a circular array of N partial aggregations (buckets).  
If the time window size is 10 seconds, the circular array has always 10 partial aggregations (buckets). Every bucket aggregates the outcome of all calls which happen in a certain epoch second. (Partial aggregation). The head bucket of the circular array stores the call outcomes of the current epoch second. The other partial aggregations store the call outcomes of the previous seconds.  
The sliding window does not store call outcomes (tuples) individually, but incrementally updates partial aggregations (bucket) and a total aggregation.  
The total aggregation is updated incrementally when a new call outcome is recorded. When the oldest bucket is evicted, the partial total aggregation of that bucket is subtracted from the total aggregation and the bucket is reset. (Subtract-on-Evict)

The time to retrieve a Snapshot is constant 0(1), since the Snapshot is pre-aggregated and is independent of the time window size.  
The space requirement (memory consumption) of this implementation should be nearly constant O(n), since the call outcomes (tuples) are not stored individually. Only N partial aggregations and 1 total total aggregation are created.

A partial aggregation consists of 3 integers in order to count the number of failed calls, the number of slow calls and total number of calls. And one long which stores total duration of all calls.

## Failure rate and slow call rate thresholds

The state of the CircuitBreaker changes from CLOSED to OPEN when the failure rate is equal or greater than a configurable threshold. For example when more than 50% of the recorded calls have failed.  
By default all exceptions count as a failure. You can define a list of exceptions which should count as a failure. All other exceptions are then counted as a success, unless they are ignored. Exceptions can also be ignored so that they neither count as a failure nor success.

The CircuitBreaker also changes from CLOSED to OPEN when the percentage of slow calls is equal or greater than a configurable threshold. For example when more than 50% of the recorded calls took longer than 5 seconds. This helps to reduce the load on an external system before it is actually unresponsive.

The failure rate and slow call rate can only be calculated, if a minimum number of calls were recorded. For example, if the minimum number of required calls is 10, then at least 10 calls must be recorded, before the failure rate can be calculated. If only 9 calls have been evaluated the CircuitBreaker will not trip open even if all 9 calls have failed.

The CircuitBreaker rejects calls with a CallNotPermittedException when it is OPEN. After a wait time duration has elapsed, the CircuitBreaker state changes from OPEN to HALF\_OPEN and permits a configurable number of calls to see if the backend is still unavailable or has become available again. Further calls are rejected with a CallNotPermittedException, until all permitted calls have completed.  
If the failure rate or slow call rate is then equal or greater than the configured threshold, the state changes back to OPEN. If the failure rate and slow call rate is below the threshold, the state changes back to CLOSED.

The Circuit Breaker supports two more special states, DISABLED (always allow access) and FORCED\_OPEN (always deny access). In these two states no Circuit Breaker events (apart from the state transition) are generated, and no metrics are recorded. The only way to exit from those states are to trigger a state transition or to reset the Circuit Breaker.

The CircuitBreaker is thread-safe as follows :

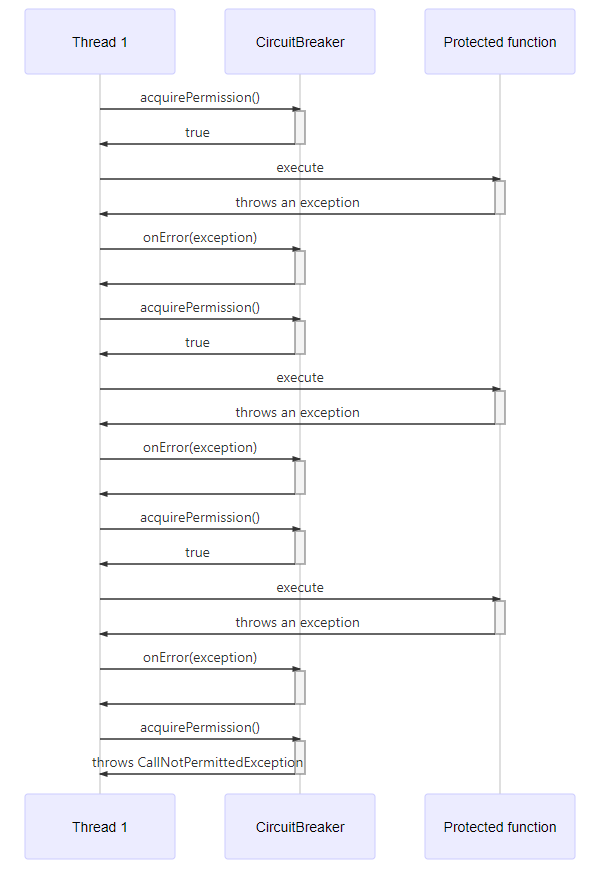
* The state of a CircuitBreaker is stored in a AtomicReference
* The CircuitBreaker uses atomic operations to update the state with side-effect-free functions.
* Recording calls and reading snapshots from the Sliding Window is synchronized

That means atomicity should be guaranteed and only one thread is able to update the state or the Sliding Window at a point in time.

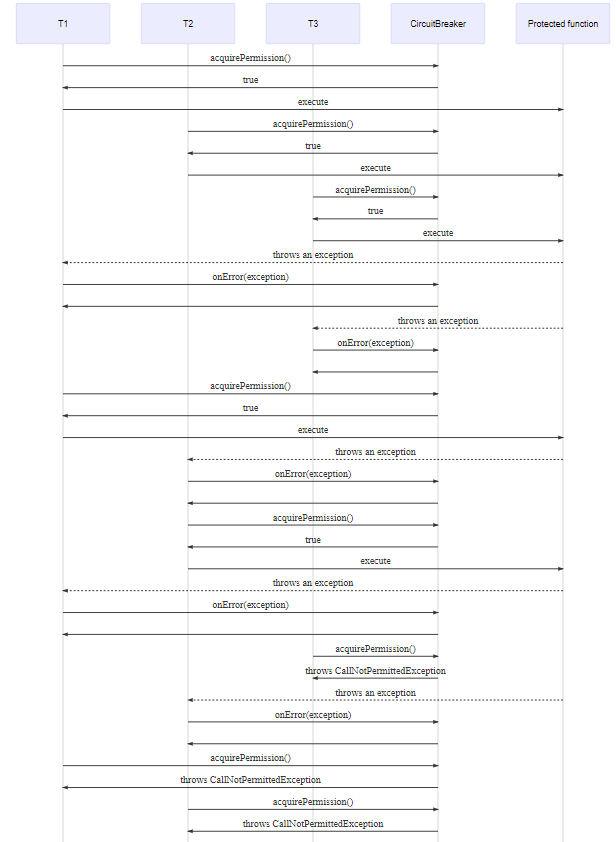
But the CircuitBreaker does not synchronize the function call. That means the function call itself is not part of the critical section. Otherwise a CircuitBreaker would introduce a huge performance penalty and bottleneck. A slow function call would have a huge negative impact to the overall performance/throughput.

If 20 concurrent threads ask for the permission to execute a function and the state of the CircuitBreaker is closed, all threads are allowed to invoke the function. Even if the Ring Bit Buffer size is 15. The size of the Ring Bit Buffer does not mean that only 15 calls are allowed to run concurrently. If you want to restrict the number of concurrent threads, please use a Bulkhead. You can combine a Bulkhead and a CircuitBreaker.

Example with 1 Thread:

[](https://files.readme.io/45dc011-Thread1.PNG)

Example with 3 Threads:

ofDe[](https://files.readme.io/8d10418-Multiplethreads.PNG)

## Create a CircuitBreakerRegistry

Resilience4j comes with an in-memory CircuitBreakerRegistry based on a ConcurrentHashMap which provides thread safety and atomicity guarantees. You can use the CircuitBreakerRegistry to manage (create and retrieve) CircuitBreaker instances. You can create a CircuitBreakerRegistry with a global default CircuitBreakerConfig for all of your CircuitBreaker instances as follows.

CircuitBreakerRegistry circuitBreakerRegistry =

CircuitBreakerRegistry.ofDefaults();

## Create and configure a CircuitBreaker

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

| Config property | Default Value | Description |
| --- | --- | --- |
| failureRateThreshold | 50 | Configures the failure rate threshold in percentage.  When the failure rate is equal or greater than the threshold the CircuitBreaker transitions to open and starts short-circuiting calls. |
| slowCallRateThreshold | 100 | Configures a threshold in percentage. The CircuitBreaker considers a call as slow when the call duration is greater than slowCallDurationThreshold  When the percentage of slow calls is equal or greater the threshold, the CircuitBreaker transitions to open and starts short-circuiting calls. |
| slowCallDurationThreshold | 60 [s] | Configures the duration threshold above which calls are considered as slow and increase the rate of slow calls. |
| permittedNumberOfCalls InHalfOpenState | 10 | Configures the number of permitted calls when the CircuitBreaker is half open. |
| slidingWindowType | COUNT\_BASED | Configures the type of the sliding window which is used to record the outcome of calls when the CircuitBreaker is closed. Sliding window can either be count-based or time-based.  If the sliding window is COUNT\_BASED, the last slidingWindowSize calls are recorded and aggregated. If the sliding window is TIME\_BASED, the calls of the last slidingWindowSize seconds recorded and aggregated. |
| slidingWindowSize | 100 | Configures the size of the sliding window which is used to record the outcome of calls when the CircuitBreaker is closed. |
| minimumNumberOfCalls | 10 | Configures the minimum number of calls which are required (per sliding window period) before the CircuitBreaker can calculate the error rate. For example, if minimumNumberOfCalls is 10, then at least 10 calls must be recorded, before the failure rate can be calculated. If only 9 calls have been recorded the CircuitBreaker will not transition to open even if all 9 calls have failed. |
| waitDurationInOpenState | 60 [s] | The time that the CircuitBreaker should wait before transitioning from open to half-open. |
| automaticTransition FromOpenToHalfOpenEnabled | False | If set to true it means that the CircuitBreaker will automatically transition from open to half-open state and not call is need to trigger the transition. |
| recordExceptions | empty | A list of exceptions that are recorded as a failure and thus increase the failure rate. Any exception matching or inheriting from one of the list counts as a failure, unless explicitly ignored via ignoreExceptions. If you specify a list of exceptions, all other exceptions count as a success, unless they are explicitly ignored by ignoreExceptions. |
| ignoreExceptions | Empty | A list of exceptions that are ignored and neither count as a failure nor success. Any exception matching or inheriting from one of the list will not count as a failure nor success, even if the exceptions is part of recordExceptions. |
| recordException | throwable -> true | By default all exceptions are recored as failures.  A custom Predicate which evaluates if an exception should be recorded as a failure. The Predicate must return true if the exception should count as a failure. The Predicate must return false, if the exception should count as a success, unless the exception is explicitly ignored by ignoreExceptions. |
| ignoreException | throwable -> false | By default no exception is ignored.  A custom Predicate which evaluates if an exception should be ignored and neither count as a failure nor success. The Predicate must return true if the exception should be ignored. The Predicate must return false, if the exception should count as a failure.  // Create a custom configuration for a CircuitBreaker |

CircuitBreakerConfig circuitBreakerConfig = CircuitBreakerConfig.custom()

.failureRateThreshold(50)

.slowCallRateThreshold(50)

.waitDurationInOpenState(Duration.ofMillis(1000))

.slowCallDurationThreshold(Duration.ofSeconds(2))

.permittedNumberOfCallsInHalfOpenState(3)

.minimumNumberOfCalls(10)

.slidingWindowType(SlidingWindowType.TIME\_BASED)

.slidingWindowSize(5)

.recordException(e -> INTERNAL\_SERVER\_ERROR

.equals(getResponse().getStatus()))

.recordExceptions(IOException.class, TimeoutException.class)

.ignoreExceptions(BusinessException.class, OtherBusinessException.class)

.build();

// Create a CircuitBreakerRegistry with a custom global configuration

CircuitBreakerRegistry circuitBreakerRegistry

CircuitBreakerRegistry.of(circuitBreakerConfig);

// Get or create a CircuitBreaker from the CircuitBreakerRegistry

// with the global default configuration

CircuitBreaker circuitBreakerWithDefaultConfig =

circuitBreakerRegistry.circuitBreaker("name1");

// Get or create a CircuitBreaker from the CircuitBreakerRegistry

// with a custom configuration

CircuitBreaker circuitBreakerWithCustomConfig = circuitBreakerRegistry

.circuitBreaker("name2", circuitBreakerConfig);

You can add configurations which can be shared by multiple CircuitBreaker instances.

CircuitBreakerConfig circuitBreakerConfig = CircuitBreakerConfig.custom()

.failureRateThreshold(70)

.build();

circuitBreakerRegistry.addConfiguration("someSharedConfig", config);

CircuitBreaker circuitBreaker = circuitBreakerRegistry

.circuitBreaker("name", "someSharedConfig");

You can overwrite configurations.

CircuitBreakerConfig defaultConfig = circuitBreakerRegistry

.getDefaultConfig();

CircuitBreakerConfig overwrittenConfig = CircuitBreakerConfig

.from(defaultConfig)

.waitDurationInOpenState(Duration.ofSeconds(20))

.build();

If you don’t want to use the CircuitBreakerRegistry to manage CircuitBreaker instances, you can also create instances directly.

// Create a custom configuration for a CircuitBreaker

CircuitBreakerConfig circuitBreakerConfig = CircuitBreakerConfig.custom()

.recordExceptions(IOException.class, TimeoutException.class)

.ignoreExceptions(BusinessException.class, OtherBusinessException.class)

.build();

CircuitBreaker customCircuitBreaker = CircuitBreaker

.of("testName", circuitBreakerConfig);

## Decorate and execute a functional interface

You can decorate any Callable, Supplier, Runnable, Consumer, CheckedRunnable, CheckedSupplier, CheckedConsumer or CompletionStage with a CircuitBreaker.  
You can invoke the decorated function with Try.of(…​) or Try.run(…​) from Vavr. This allows to chain further functions with map, flatMap, filter, recover or andThen. The chained functions are only invoked, if the CircuitBreaker is CLOSED or HALF\_OPEN.  
In the following example, Try.of(…​) returns a Success<String> Monad, if the invocation of the function is successful. If the function throws an exception, a Failure<Throwable> Monad is returned and map is not invoked.

// Given

CircuitBreaker circuitBreaker = CircuitBreaker.ofDefaults("testName");

// When I decorate my function

CheckedFunction0<String> decoratedSupplier = CircuitBreaker

.decorateCheckedSupplier(circuitBreaker, () -> "This can be any method which returns: 'Hello");

// and chain an other function with map

Try<String> result = Try.of(decoratedSupplier)

.map(value -> value + " world'");

// Then the Try Monad returns a Success<String>, if all functions ran successfully.

assertThat(result.isSuccess()).isTrue();

assertThat(result.get()).isEqualTo("This can be any method which returns: 'Hello world'");

## Consume emitted RegistryEvents

You can register event consumer on a CircuitBreakerRegistry and take actions whenever a CircuitBreaker is created, replaced or deleted.

CircuitBreakerRegistry circuitBreakerRegistry = CircuitBreakerRegistry.ofDefaults();

circuitBreakerRegistry.getEventPublisher()

.onEntryAdded(entryAddedEvent -> {

CircuitBreaker addedCircuitBreaker = entryAddedEvent.getAddedEntry();

LOG.info("CircuitBreaker {} added", addedCircuitBreaker.getName());

})

.onEntryRemoved(entryRemovedEvent -> {

CircuitBreaker removedCircuitBreaker = entryRemovedEvent.getRemovedEntry();

LOG.info("CircuitBreaker {} removed", removedCircuitBreaker.getName());

});

## Consume emitted CircuitBreakerEvents

A CircuitBreakerEvent can be a state transition, a circuit breaker reset, a successful call, a recorded error or an ignored error. All events contains additional information like event creation time and processing duration of the call. If you want to consume events, you have to register an event consumer.

circuitBreaker.getEventPublisher()

.onSuccess(event -> logger.info(...))

.onError(event -> logger.info(...))

.onIgnoredError(event -> logger.info(...))

.onReset(event -> logger.info(...))

.onStateTransition(event -> logger.info(...));

// Or if you want to register a consumer listening

// to all events, you can do:

circuitBreaker.getEventPublisher()

.onEvent(event -> logger.info(...));

You could use the CircularEventConsumer to store events in a circular buffer with a fixed capacity.

CircularEventConsumer<CircuitBreakerEvent> ringBuffer =

new CircularEventConsumer<>(10);

circuitBreaker.getEventPublisher().onEvent(ringBuffer);

List<CircuitBreakerEvent> bufferedEvents = ringBuffer.getBufferedEvents()

You can use RxJava or RxJava2 Adapters to convert the EventPublisher into a Reactive Stream.

## Examples

Examples of resilience4j-circuitbreaker

[**SUGGEST EDITS**](https://resilience4j.readme.io/docs/examples)

### Create a CircuitBreakerRegistry

Create a CircuitBreakerRegistry with a custom CircuitBreakerConfig.

// Create a custom configuration for a CircuitBreaker

CircuitBreakerConfig circuitBreakerConfig = CircuitBreakerConfig.custom()

.failureRateThreshold(50)

.waitDurationInOpenState(Duration.ofMillis(1000))

.permittedNumberOfCallsInHalfOpenState(2)

.slidingWindowSize(2)

.recordExceptions(IOException.class, TimeoutException.class)

.ignoreExceptions(BusinessException.class, OtherBusinessException.class)

.build();

// Create a CircuitBreakerRegistry with a custom global configuration

CircuitBreakerRegistry circuitBreakerRegistry =

CircuitBreakerRegistry.of(circuitBreakerConfig);

### Create a CircuitBreaker

Get a CircuitBreaker from the CircuitBreakerRegistry with the global default configuration

CircuitBreaker circuitBreaker = circuitBreakerRegistry

.circuitBreaker("name");

### Decorate a functional interface

Decorate your call to BackendService.doSomething() with a CircuitBreaker and execute the decorated supplier and recover from any exception.

Supplier<String> decoratedSupplier = CircuitBreaker

.decorateSupplier(circuitBreaker, backendService::doSomething);

String result = Try.ofSupplier(decoratedSupplier)

.recover(throwable -> "Hello from Recovery").get();

### Execute a decorated functional interface

When you don't want to decorate your lambda expression, but just execute it and protect the call by a CircuitBreaker.

String result = circuitBreaker

.executeSupplier(backendService::doSomething);

### Recover from an exception

If you want to recover from an exception after the CircuitBreaker recorded it as a failure, you can chain the method Try.recover(). The recovery method is only invoked, if Try.of() returns a Failure<Throwable> Monad.

// Given

CircuitBreaker circuitBreaker = CircuitBreaker.ofDefaults("testName");

// When I decorate my function and invoke the decorated function

CheckedFunction0<String> checkedSupplier =

CircuitBreaker.decorateCheckedSupplier(circuitBreaker, () -> {

throw new RuntimeException("BAM!");

});

Try<String> result = Try.of(checkedSupplier)

.recover(throwable -> "Hello Recovery");

// Then the function should be a success,

// because the exception could be recovered

assertThat(result.isSuccess()).isTrue();

// and the result must match the result of the recovery function.

assertThat(result.get()).isEqualTo("Hello Recovery");

If you want to recover from an exception before the CircuitBreaker records it as a failure, you can do the following:

Supplier<String> supplier = () -> {

throw new RuntimeException("BAM!");

};

Supplier<String> supplierWithRecovery = SupplierUtils

.recover(supplier, (exception) -> "Hello Recovery");

String result = circuitBreaker.executeSupplier(supplierWithRecovery);

assertThat(result).isEqualTo("Hello Recovery");

SupplierUtils and CallableUtils contain other methods like andThen which can take can be used to chain functions. For example to check the status code of a HTTP response, so that exceptions can be thrown.

Supplier<String> supplierWithResultAndExceptionHandler = SupplierUtils

.andThen(supplier, (result, exception) -> "Hello Recovery");

Supplier<HttpResponse> supplier = () -> httpClient.doRemoteCall();

Supplier<HttpResponse> supplierWithResultHandling = SupplierUtils.andThen(supplier, result -> {

if (result.getStatusCode() == 400) {

throw new ClientException();

} else if (result.getStatusCode() == 500) {

throw new ServerException();

}

return result;

});

HttpResponse httpResponse = circuitBreaker

.executeSupplier(supplierWithResultHandling);

### Reset CircuitBreaker

The Circuit Breaker supports resetting to its original state, losing all the metrics and effectively resetting its Ring Bit Buffer.

CircuitBreaker circuitBreaker = CircuitBreaker.ofDefaults("testName");

circuitBreaker.reset();

### Transition to states manually

CircuitBreaker circuitBreaker = CircuitBreaker.ofDefaults("testName");

circuitBreaker.transitionToDisabledState();

// circuitBreaker.onFailure(...) won't trigger a state change

circuitBreaker.transitionToClosedState(); // will transition to CLOSED state and re-enable normal behaviour, keeping metrics

circuitBreaker.transitionToForcedOpenState();

// circuitBreaker.onSuccess(...) won't trigger a state change

circuitBreaker.reset(); // will transition to CLOSED state and re-enable norma

# Bulkhead

Getting started with resilience4j-bulkhead

[**SUGGEST EDITS**](https://resilience4j.readme.io/docs/bulkhead)

## Introduction

Resilience4j provides two implementations of a bulkhead pattern that can be used to limit the number of concurrent execution:

* a SemaphoreBulkhead which uses Semaphores
* a FixedThreadPoolBulkhead which uses a bounded queue and a fixed thread pool.

The SemaphoreBulkheadshould work well across a variety of threading and I/O models. It is based on a semaphore, and unlike Hystrix, does not provide "shadow" thread pool option. It is up to the client to ensure correct thread pool sizing that will be consistent with bulkhead configuration.

## Create a BulkheadRegistry

Just like the CircuitBreaker module, this module provides an in-memory BulkheadRegistry and a ThreadPoolBulkheadRegistry which you can use to manage (create and retrieve) Bulkhead instances.

BulkheadRegistry bulkheadRegistry = BulkheadRegistry.ofDefaults();

ThreadPoolBulkheadRegistry threadPoolBulkheadRegistry =

ThreadPoolBulkheadRegistry.ofDefaults();

## 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. |

// Create a custom configuration for a Bulkhead

BulkheadConfig config = BulkheadConfig.custom()

.maxConcurrentCalls(150)

.maxWaitDuration(Duration.ofMillis(500))

.build();

// Create a BulkheadRegistry with a custom global configuration

BulkheadRegistry registry = BulkheadRegistry.of(config);

// Get or create a Bulkhead from the registry -

// bulkhead will be backed by the default config

Bulkhead bulkheadWithDefaultConfig = registry.bulkhead("name1");

// Get or create a Bulkhead from the registry,

// use a custom configuration when creating the bulkhead

Bulkhead bulkheadWithCustomConfig = registry.bulkhead("name2", custom);

## Create and configure a ThreadPoolBulkhead

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

|  |  |  |
| --- | --- | --- |
| 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. |

ThreadPoolBulkheadConfig config = ThreadPoolBulkheadConfig.custom()

.maxThreadPoolSize(10)

.coreThreadPoolSize(2)

.queueCapacity(20)

.build();

// Create a BulkheadRegistry with a custom global configuration

ThreadPoolBulkheadRegistry registry = ThreadPoolBulkheadRegistry.of(config);

// Get or create a ThreadPoolBulkhead from the registry -

// bulkhead will be backed by the default config

ThreadPoolBulkhead bulkheadWithDefaultConfig = registry.bulkhead("name1");

// Get or create a Bulkhead from the registry,

// use a custom configuration when creating the bulkhead

ThreadPoolBulkheadConfig custom = BulkheadConfig.custom()

.maxThreadPoolSize(5)

.build();

ThreadPoolBulkhead bulkheadWithCustomConfig = registry.bulkhead("name2", custom);

## Decorate and execute a functional interface

As you can guess Bulkhead has all sort of higher order decorator functions just like CircuitBreaker. You can decorate any Callable, Supplier, Runnable, Consumer, CheckedRunnable, CheckedSupplier, CheckedConsumer or CompletionStage with a Bulkhead.

// Given

Bulkhead bulkhead = Bulkhead.of("name", config);

// When I decorate my function

CheckedFunction0<String> decoratedSupplier = Bulkhead

.decorateCheckedSupplier(bulkhead, () -> "This can be any method which returns: 'Hello");

// and chain an other function with map

Try<String> result = Try.of(decoratedSupplier)

.map(value -> value + " world'");

// Then the Try Monad returns a Success<String>, if all functions ran successfully.

assertThat(result.isSuccess()).isTrue();

assertThat(result.get()).isEqualTo("This can be any method which returns: 'Hello world'");

assertThat(bulkhead.getMetrics().getAvailableConcurrentCalls()).isEqualTo(1);

ThreadPoolBulkheadConfig config = ThreadPoolBulkheadConfig.custom()

.maxThreadPoolSize(10)

.coreThreadPoolSize(2)

.queueCapacity(20)

.build();

ThreadPoolBulkhead bulkhead = ThreadPoolBulkhead.of("name", config);

CompletionStage<String> supplier = ThreadPoolBulkhead

.executeSupplier(bulkhead, backendService::doSomething);

## Consume emitted RegistryEvents

You can register event consumer on a BulkheadRegistry and take actions whenever a Bulkhead is created, replaced or deleted.

BulkheadRegistry registry = BulkheadRegistry.ofDefaults();

registry.getEventPublisher()

.onEntryAdded(entryAddedEvent -> {

Bulkhead addedBulkhead = entryAddedEvent.getAddedEntry();

LOG.info("Bulkhead {} added", addedBulkhead.getName());

})

.onEntryRemoved(entryRemovedEvent -> {

Bulkhead removedBulkhead = entryRemovedEvent.getRemovedEntry();

LOG.info("Bulkhead {} removed", removedBulkhead.getName());

});

## Consume emitted BulkheadEvents

The BulkHead emits a stream of BulkHeadEvents. There are two types of events emitted: permitted execution, rejected execution & finished execution. If you want to consume these events, you have to register an event consumer.

bulkhead.getEventPublisher()

.onCallPermitted(event -> logger.info(...))

.onCallRejected(event -> logger.info(...))

.onCallFinished(event -> logger.info(...));

## Examples

Examples of resilience4j-bulkhead

[**SUGGEST EDITS**](https://resilience4j.readme.io/docs/examples-3)

### Create a BulkheadRegistry

Create a BulkheadRegistry with a custom BulkheadConfig.

// Create a custom configuration for a RateLimiter

RateLimiterConfig config = RateLimiterConfig.custom()

.limitRefreshPeriod(Duration.ofMillis(1))

.limitForPeriod(10)

.timeoutDuration(Duration.ofMillis(25))

.build();

// Create a RateLimiterRegistry with a custom global configuration

RateLimiterRegistry rateLimiterRegistry = RateLimiterRegistry.of(config);

### Create a Bulkhead

Get a Bulkhead from the BulkheadRegistry with the global default configuration

Bulkhead bulkhead = bulkheadRegistry

.circuitBreaker("name");

### Decorate a functional interface

Decorate your call to BackendService.doSomething() with a Bulkhead and execute the decorated supplier and recover from any exception.

Supplier<String> decoratedSupplier = Bulkhead

.decorateSupplier(retry, backendService::doSomething);

String result = Try.ofSupplier(decoratedSupplier)

.recover(throwable -> "Hello from Recovery").get();

# RateLimiter

Getting started with resilience4j-ratelimiter

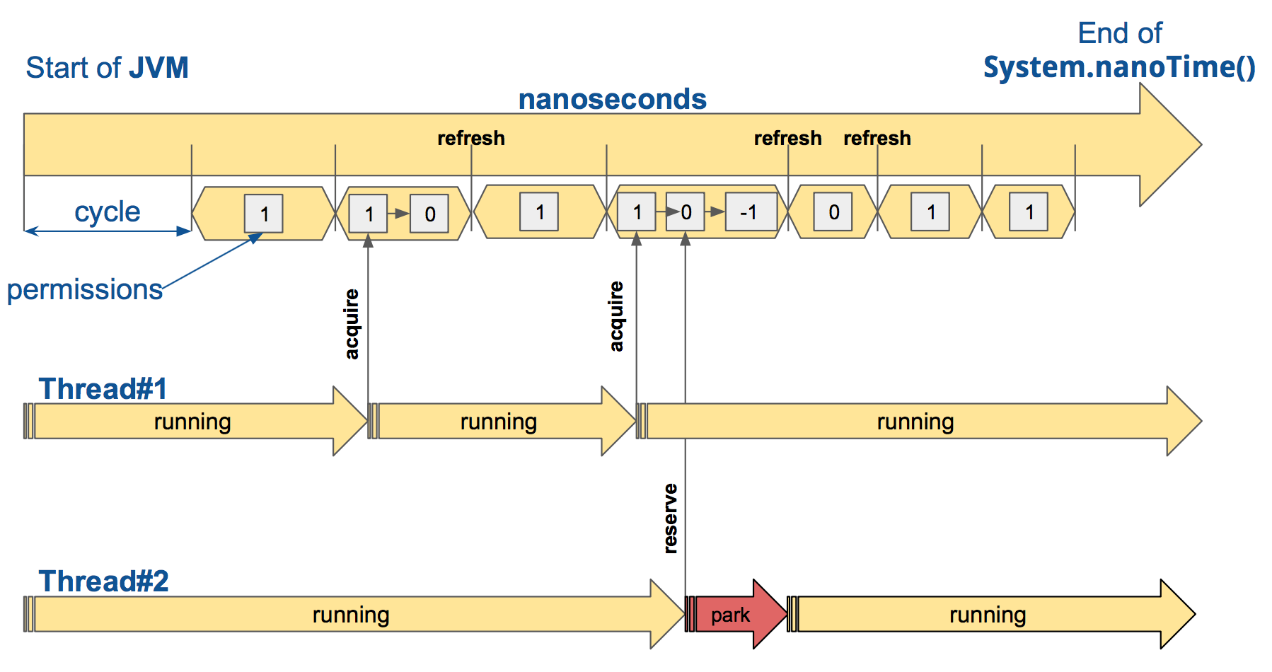
[**SUGGEST EDITS**](https://resilience4j.readme.io/docs/ratelimiter)

## Introduction

Rate limiting is an imperative technique to prepare your API for scale and establish high availability and reliability of your service. But also, this technique comes with a whole bunch of different options of how to handle a detected limits surplus, or what type of requests you want to limit. You can simply decline this over limit request, or build a queue to execute them later or combine these two approaches in some way.

### Internals

Resilience4j provides a RateLimiter which splits all nanoseconds from the start of epoch into cycles. Each cycle has a duration configured by RateLimiterConfig.limitRefreshPeriod. At the start of each cycle, the RateLimiter sets the number of active permissions to RateLimiterConfig.limitForPeriod.  
For the RateLimiter callers it really looks like this, but for the AtomicRateLimiter implementation has some optimizations under the hood that will skip this refresh, if RateLimiter is not used actively.

[](https://files.readme.io/44ca055-rate_limiter.png)

The default implementation of RateLimiter is AtomicRateLimiter which manages it's state via AtomicReference. The AtomicRateLimiter.State is completely immutable and has the folowing fields:

* activeCycle - cycle number that was used by the last call
* activePermissions - count of available permissions after the last call.  
  Can be negative if some permissions where reserved
* nanosToWait - count of nanoseconds to wait for permission for the last call

There is also a SemaphoreBasedRateLimiter which uses Semaphores and a scheduler that will refresh permissions after each RateLimiterConfig#limitRefreshPeriod.

## Create a RateLimiterRegistry

Just like the CircuitBreaker module, this module provides an in-memory 'RateLimiterRegistry' which you can use to manage (create and retrieve) RateLimiter instances.

RateLimiterRegistry rateLimiterRegistry = RateLimiterRegistry.ofDefaults();

## Create and configure a RateLimiter

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

|  |  |  |
| --- | --- | --- |
| Config property | Default value | Description |
| timeoutDuration | 5 [s] | The default wait time a thread waits for a permission |
| limitRefreshPeriod | 500 [ns] | The period of a limit refresh. After each period the rate limiter sets its permissions count back to the limitForPeriod value |
| limitForPeriod | 50 | The number of permissions available during one limit refresh period |

For example you want to restrict the calling rate of some method to be not higher than 10 req/ms.

RateLimiterConfig config = RateLimiterConfig.custom()

.limitRefreshPeriod(Duration.ofMillis(1))

.limitForPeriod(10)

.timeoutDuration(Duration.ofMillis(25))

.build();

// Create registry

RateLimiterRegistry rateLimiterRegistry = RateLimiterRegistry.of(config);

// Use registry

RateLimiter rateLimiterWithDefaultConfig = rateLimiterRegistry

.rateLimiter("name1");

RateLimiter rateLimiterWithCustomConfig = rateLimiterRegistry

.rateLimiter("name2", config);

## Decorate and execute a functional interface

As you can guess RateLimiter has all sort of higher order decorator functions just like CircuitBreaker. You can decorate any Callable, Supplier, Runnable, Consumer, CheckedRunnable, CheckedSupplier, CheckedConsumer or CompletionStage with a RateLimiter.

// Decorate your call to BackendService.doSomething()

CheckedRunnable restrictedCall = RateLimiter

.decorateCheckedRunnable(rateLimiter, backendService::doSomething);

Try.run(restrictedCall)

.andThenTry(restrictedCall)

.onFailure((RequestNotPermitted throwable) -> LOG.info("Wait before call it again :)"));

You can use changeTimeoutDuration and changeLimitForPeriod methods to change rate limiter params in runtime.  
New timeout duration won’t affect threads that are currently waiting for permission.  
New limit won’t affect current period permissions and will apply only from next one.

// Decorate your call to BackendService.doSomething()

CheckedRunnable restrictedCall = RateLimiter

.decorateCheckedRunnable(rateLimiter, backendService::doSomething);

// durring second refresh cycle limiter will get 100 permissions

rateLimiter.changeLimitForPeriod(100);

## Consume emitted RegistryEvents

You can register event consumer on a RateLimiterRegistry and take actions whenever a RateLimiter is created, replaced or deleted.

RateLimiterRegistry registry = RateLimiterRegistry.ofDefaults();

registry.getEventPublisher()

.onEntryAdded(entryAddedEvent -> {

RateLimiter addedRateLimiter = entryAddedEvent.getAddedEntry();

LOG.info("RateLimiter {} added", addedRateLimiter.getName());

})

.onEntryRemoved(entryRemovedEvent -> {

RateLimiter removedRateLimiter = entryRemovedEvent.getRemovedEntry();

LOG.info("RateLimiter {} removed", removedRateLimiter.getName());

});

## Consume emitted RateLimiterEvents

The RateLimiter emits a stream of RateLimiterEvents. An event can be a successful permission acquire or acquire failure.  
All events contains additional information like event creation time and rate limiter name.  
If you want to consume events, you have to register an event consumer.

rateLimiter.getEventPublisher()

.onSuccess(event -> logger.info(...))

.onFailure(event -> logger.info(...));

You can use RxJava or RxJava2 Adapters to convert the EventPublisher into a Reactive Stream.

ReactorAdapter.toFlux(rateLimiter.getEventPublisher())

.filter(event -> event.getEventType() == FAILED\_ACQUIRE)

.subscribe(event -> logger.info(...))

## Examples

Examples of resilience4j-ratelimiter

[**SUGGEST EDITS**](https://resilience4j.readme.io/docs/examples-4)

### Create a RateLimiterRegistry

Create a RateLimiterRegistry with a custom RateLimiterConfig.

// Create a custom configuration for a RateLimiter

RateLimiterConfig config = RateLimiterConfig.custom()

.timeoutDuration(TIMEOUT)

.limitRefreshPeriod(REFRESH\_PERIOD)

.limitForPeriod(LIMIT)

.build();

// Create a RateLimiterRegistry with a custom global configuration

RateLimiterRegistry registry = RateLimiterRegistry.of(config);

# Retry

Getting started with resilience4j-retry

## Create a RetryRegistry

Just like the CircuitBreaker module, this module provides an in-memory 'RetryRegistry' which you can use to manage (create and retrieve) Retry instances.

RetryRegistry retryRegistry = RetryRegistry.ofDefaults();

## Create and configure Retry

You can provide a custom global RetryConfig. In order to create a custom global RetryConfig, you can use the RetryConfig builder. You can use the builder to configure:

* the maximum number of retry attempts
* the wait duration between successive attempts
* a custom Predicate which evalutas if a certain reponse should trigger a retry attempt
* a custom Predicate which evaluates if an exception should trigger a retry attempt
* a list of exceptions which should trigger a retry attempt
* a list of exceptions which should be ignored and not trigger a retry attempt

|  |  |  |
| --- | --- | --- |
| Config property | Default value | Description |
| maxAttempts | 3 | The maximum number of retry attempts |
| waitDuration | 500 [ms] | A fixed wait duration between retry attempts |
| intervalFunction | numOfAttempts -> waitDuration | A function to modify the waiting interval after a failure. By default the wait duration remains constant. |
| retryOnResultPredicate | result -> false | Configures a Predicate which evaluates if a result should be retried. The Predicate must return true, if the result should be retried, otherwise it must return false. |
| retryOnExceptionPredicate | throwable -> true | Configures a Predicate which evaluates if an exception should be retried. The Predicate must return true, if the exception should be retried, otherwise it must return false. |
| retryExceptions | empty | Configures a list of error classes that are recorded as a failure and thus are retried. |
| ignoreExceptions | Empty | Configures a list of error classes that are ignored and thus are not retried. |

RetryConfig config = RetryConfig.custom()

.maxAttempts(2)

.waitDuration(Duration.ofMillis(1000))

.retryOnResult(response -> response.getStatus() == 500)

.retryOnException(e -> e instanceof WebServiceException)

.retryExceptions(IOException.class, TimeoutException.class)

.ignoreExceptions(BunsinessException.class, OtherBunsinessException.class)

.build();

// Create a RetryRegistry with a custom global configuration

RetryRegistry registry = RetryRegistry.of(config);

// Get or create a Retry from the registry -

// Retry will be backed by the default config

Retry retryWithDefaultConfig = registry.retry("name1");

// Get or create a Bulkhead from the registry,

// use a custom configuration when creating the bulkhead

RetryConfig custom = RetryConfig.custom()

.waitDuration(Duration.ofMillis(100))

.build();

Retry retryWithCustomConfig = registry.retry("name2", custom);

## Decorate and execute a functional interface

As you can guess Retry has all sort of higher order decorator functions just like CircuitBreaker. You can decorate any Callable, Supplier, Runnable, Consumer, CheckedRunnable, CheckedSupplier, CheckedConsumer or CompletionStage with a Retry.

// Given I have a HelloWorldService which throws an exception

HelloWorldService helloWorldService = mock(HelloWorldService.class);

given(helloWorldService.sayHelloWorld())

.willThrow(new WebServiceException("BAM!"));

// Create a Retry with default configuration

Retry retry = Retry.ofDefaults("id");

// Decorate the invocation of the HelloWorldService

CheckedFunction0<String> retryableSupplier = Retry

.decorateCheckedSupplier(retry, helloWorldService::sayHelloWorld);

// When I invoke the function

Try<String> result = Try.of(retryableSupplier)

.recover((throwable) -> "Hello world from recovery function");

// Then the helloWorldService should be invoked 3 times

BDDMockito.then(helloWorldService).should(times(3)).sayHelloWorld();

// and the exception should be handled by the recovery function

assertThat(result.get()).isEqualTo("Hello world from recovery function");

## Consume emitted RegistryEvents

You can register event consumer on a RetryRegistry and take actions whenever a Retry is created, replaced or deleted.

RetryRegistry registry = RetryRegistry.ofDefaults();

registry.getEventPublisher()

.onEntryAdded(entryAddedEvent -> {

Retry addedRetry = entryAddedEvent.getAddedEntry();

LOG.info("Retry {} added", addedRetry.getName());

})

.onEntryRemoved(entryRemovedEvent -> {

Retry removedRetry = entryRemovedEvent.getRemovedEntry();

LOG.info("Retry {} removed", removedRetry.getName());

});

## Use a custom IntervalFunction

If you don't want to use a fixed wait duration between retry attempts, you can configure an IntervalFunction which is used instead to calculate the wait duration for every attempt. Resilience4j provides several factory methods to simplify the creation of an IntervalFunction.

IntervalFunction defaultWaitInterval = IntervalFunction

.ofDefaults();

// This interval function is used internally

// when you only configure waitDuration

IntervalFunction fixedWaitInterval = IntervalFunction

.of(Duration.ofSeconds(5));

IntervalFunction intervalWithExponentialBackoff = IntervalFunction

.ofExponentialBackoff();

IntervalFunction intervalWithCustomExponentialBackoff = IntervalFunction

.ofExponentialBackoff(IntervalFunction.DEFAULT\_INITIAL\_INTERVAL, 2d);

IntervalFunction randomWaitInterval = IntervalFunction

.ofRandomized();

// Overwrite the default intervalFunction with your custom one

RetryConfig retryConfig = RetryConfig.custom()

.intervalFunction(intervalWithExponentialBackoff)

.build();

## Examples

Examples of resilience4j-retry

[**SUGGEST EDITS**](https://resilience4j.readme.io/docs/examples-5)

### Create a RetryRegistry

Create a RetryRegistry with a custom RetryConfig.

RetryConfig config = RetryConfig.custom()

.maxAttempts(2)

.waitDuration(Duration.ofMillis(100))

.retryOnResult(response -> response.getStatus() == 500)

.retryOnException(e -> e instanceof WebServiceException)

.retryExceptions(IOException.class, TimeoutException.class)

.ignoreExceptions(BunsinessException.class, OtherBunsinessException.class)

.build();

// Create a RetryRegistry with a custom global configuration

RetryRegistry registry = RetryRegistry.of(config);

# Resilience patterns

| **name** | **how does it work?** | **description** | **slogans** | **links** |
| --- | --- | --- | --- | --- |
| **Retry** | repeats failed executions | Many faults are transient and may self-correct after a short delay. | "Insert coin to try again", "Maybe it’s just a blip" | [overview](https://github.com/resilience4j/resilience4j#circuitbreaker-retry-fallback), [documentation](https://resilience4j.readme.io/docs/retry) |
| **Circuit Breaker** | temporary blocks possible failures | When a system is seriously struggling, failing fast is better than making clients wait. | "Stop doing it if it hurts", "Give that system a break", "Baby, don’t hurt me, no more" | [overview](https://github.com/resilience4j/resilience4j#circuitbreaker-retry-fallback), [documentation](https://resilience4j.readme.io/docs/circuitbreaker), [Feign](https://github.com/resilience4j/resilience4j/blob/master/resilience4j-feign/README.adoc), [Retrofit](https://github.com/resilience4j/resilience4j/blob/master/resilience4j-retrofit/README.adoc) |
| **Rate Limiter** | limits executions/period | Prepare for a scale and establish reliability and HA of your service. | "That’s enough for this minute!", "Well, it’ll work next time" | [overview](https://github.com/resilience4j/resilience4j#bulkhead), [documentation](https://resilience4j.readme.io/docs/ratelimiter), [Feign](https://github.com/resilience4j/resilience4j/blob/master/resilience4j-feign/README.adoc), [Retrofit](https://github.com/resilience4j/resilience4j/blob/master/resilience4j-retrofit/README.adoc) |
| **Time Limiter** | limits duration of execution | Beyond a certain wait, a success result is unlikely. | "Don’t wait forever" |  |
| **Bulkhead** | limits concurrent executions | Resources are isolated into pools so that if one fails, the others will continue. | "One fault shouldn’t sink the whole ship", "Please, please, not all at once." | [overview](https://github.com/resilience4j/resilience4j#bulkhead), [documentation](https://resilience4j.readme.io/docs/bulkhead) |
| **Cache** | memorizes a successful result | Some proportion of requests may be similar. | "You’ve asked that one before" |  |
| **Fallback** | provides an alternative result for failures | Things will still fail - plan what you will do when that happens. | "Degrade gracefully", "A bird in the hand is worth two in the bush" | [Try::recover](https://github.com/resilience4j/resilience4j#circuitbreaker-retry-fallback), [Spring](https://resilience4j.readme.io/docs/getting-started-3#section-annotations), [Feign](https://github.com/resilience4j/resilience4j/blob/master/resilience4j-feign/README.adoc) |

# Usage examples

## CircuitBreaker, Retry and Fallback

The following example shows how to decorate a lambda expression (Supplier) with a CircuitBreaker and how to retry the call at most 3 times when an exception occurs. You can configure the wait interval between retries and also configure a custom backoff algorithm.

The example uses Vavr’s Try Monad to recover from an exception and invoke another lambda expression as a fallback, when even all retries have failed.

// Simulates a Backend Service

public interface BackendService {

String doSomething();

}

// Create a CircuitBreaker (use default configuration)

CircuitBreaker circuitBreaker = CircuitBreaker.ofDefaults("backendName");

// Create a Retry with at most 3 retries and a fixed time interval between retries of 500ms

Retry retry = Retry.ofDefaults("backendName");

// Decorate your call to BackendService.doSomething() with a CircuitBreaker

Supplier<String> decoratedSupplier = CircuitBreaker

.decorateSupplier(circuitBreaker, backendService::doSomething);

// Decorate your call with automatic retry

decoratedSupplier = Retry

.decorateSupplier(retry, decoratedSupplier);

// Execute the decorated supplier and recover from any exception

String result = Try.ofSupplier(decoratedSupplier)

.recover(throwable -> "Hello from Recovery").get();

// When you don't want to decorate your lambda expression,

// but just execute it and protect the call by a CircuitBreaker.

String result = circuitBreaker.executeSupplier(backendService::doSomething);

### 6.1.1. CircuitBreaker and RxJava2

The following example shows how to decorate an Observable by using the custom RxJava operator.

CircuitBreaker circuitBreaker = CircuitBreaker.ofDefaults("testName");

Observable.fromCallable(backendService::doSomething)

.compose(CircuitBreakerOperator.of(circuitBreaker))

### CircuitBreaker and Spring Reactor

The following example shows how to decorate a Mono by using the custom Reactor operator.

CircuitBreaker circuitBreaker = CircuitBreaker.ofDefaults("testName");

Mono.fromCallable(backendService::doSomething)

.compose(CircuitBreakerOperator.of(circuitBreaker))

## RateLimiter

The following example shows how to restrict the calling rate of some method to be not higher than 1 request/second.

// Create a custom RateLimiter configuration

RateLimiterConfig config = RateLimiterConfig.custom()

.timeoutDuration(Duration.ofMillis(100))

.limitRefreshPeriod(Duration.ofSeconds(1))

.limitForPeriod(1)

.build();

// Create a RateLimiter

RateLimiter rateLimiter = RateLimiter.of("backendName", config);

// Decorate your call to BackendService.doSomething()

Supplier<String> restrictedSupplier = RateLimiter

.decorateSupplier(rateLimiter, backendService::doSomething);

// First call is successful

Try<String> firstTry = Try.ofSupplier(restrictedSupplier);

assertThat(firstTry.isSuccess()).isTrue();

// Second call fails, because the call was not permitted

Try<String> secondTry = Try.of(restrictedSupplier);

assertThat(secondTry.isFailure()).isTrue();

assertThat(secondTry.getCause()).isInstanceOf(RequestNotPermitted.class);

## Bulkhead

There are two isolation strategies and bulkhead implementations.

### SemaphoreBulkhead

The following example shows how to decorate a lambda expression with a Bulkhead. A Bulkhead can be used to limit the amount of parallel executions. This bulkhead abstraction should work well across a variety of threading and io models. It is based on a semaphore, and unlike Hystrix, does not provide "shadow" thread pool option.

// Create a custom Bulkhead configuration

BulkheadConfig config = BulkheadConfig.custom()

.maxConcurrentCalls(150)

.maxWaitTime(100)

.build();

Bulkhead bulkhead = Bulkhead.of("backendName", config);

Supplier<String> supplier = Bulkhead

.decorateSupplier(bulkhead, backendService::doSomething);

### ThreadPoolBulkhead

The following example shows how to use a lambda expression with a ThreadPoolBulkhead which uses a bounded queue and a fixed thread pool.

// Create a custom ThreadPoolBulkhead configuration

ThreadPoolBulkheadConfig config = ThreadPoolBulkheadConfig.custom()

.maxThreadPoolSize(10)

.coreThreadPoolSize(2)

.queueCapacity(20)

.build();

ThreadPoolBulkhead bulkhead = ThreadPoolBulkhead.of("backendName", config);

// Decorate or execute immediately a lambda expression with a ThreadPoolBulkhead.

Supplier<CompletionStage<String>> supplier = ThreadPoolBulkhead

.decorateSupplier(bulkhead, backendService::doSomething);

CompletionStage<String> execution = bulkhead

.executeSupplier(backendService::doSomething);

# Consume emitted events

CircuitBreaker, RateLimiter, Cache, Bulkhead, TimeLimiter and Retry components emit a stream of events. It can be consumed for logging, assertions and any other purpose.

## Examples

A CircuitBreakerEvent can be a state transition, a circuit breaker reset, a successful call, a recorded error or an ignored error. All events contains additional information like event creation time and processing duration of the call. If you want to consume events, you have to register an event consumer.

circuitBreaker.getEventPublisher()

.onSuccess(event -> logger.info(...))

.onError(event -> logger.info(...))

.onIgnoredError(event -> logger.info(...))

.onReset(event -> logger.info(...))

.onStateTransition(event -> logger.info(...));

// Or if you want to register a consumer listening to all events, you can do:

circuitBreaker.getEventPublisher()

.onEvent(event -> logger.info(...));

You can use RxJava or Spring Reactor Adapters to convert the EventPublisher into a Reactive Stream. The advantage of a Reactive Stream is that you can use RxJava’s observeOn operator to specify a different Scheduler that the CircuitBreaker will use to send notifications to its observers/consumers.

RxJava2Adapter.toFlowable(circuitBreaker.getEventPublisher())

.filter(event -> event.getEventType() == Type.ERROR)

.cast(CircuitBreakerOnErrorEvent.class)

.subscribe(event -> logger.info(...))