1. Introduction

Resilience4j is a lightweight fault tolerance library inspired by Netflix Hystrix, but designed for Java 8 and functional programming. Lightweight, because the library only uses Vavr, 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.

⚠	Netflix Hystrix is no longer in active development, and is currently in maintenance mode.

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.

```
// 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)
  .withCircuitBreaker(circuitBreaker)
  .withBulkhead(bulkhead)
  .withRetry(retry)
  .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);

// You can also run the supplier asynchronously in a ThreadPoolBulkhead
 ThreadPoolBulkhead threadPoolBulkhead = ThreadPoolBulkhead
  .ofDefaults("backendService");

// The Scheduler is needed to schedule a timeout on a non-blocking CompletableFuture
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(3);
TimeLimiter timeLimiter = TimeLimiter.of(Duration.ofSeconds(1));

CompletableFuture<String> future = Decorators.ofSupplier(supplier)
    .withThreadPoolBulkhead(threadPoolBulkhead)
    .withTimeLimiter(timeLimiter, scheduler)
    .withCircuitBreaker(circuitBreaker)
    .withFallback(asList(TimeoutException.class, CallNotPermittedException.class, Bul
      throwable -> "Hello from Recovery")
    .get().toCompletableFuture();
```

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

2. Documentation

Setup and usage is described in our **User Guide**.

- 有志による日本語訳(非公式) Japanese translation by volunteers(Unofficial)
- 这是Resilience4j的非官方中文文档 Chinese translation by volunteers(Unofficial)

3. Overview

Resilience4j provides several core modules:

- resilience4j-circuitbreaker: Circuit breaking
- resilience4j-ratelimiter: Rate limiting
- resilience4j-bulkhead: Bulkheading
- resilience4j-retry: Automatic retrying (sync and async)
- resilience4j-timelimiter: Timeout handling
- resilience4j-cache: Result caching

There are also add-on modules for metrics, Retrofit, Feign, Kotlin, Spring, Ratpack, Vertx, RxJava2 and more.

	Find out full list of modules in our **User Guide**.

	For core modules package or `Decorators` builder see **resilience4j-all**.

4. Resilience patterns

name	how does it work?	description	links
Retry	repeats failed executions	Many faults are transient and may self-correct after a short delay.	overview, documentation, Spring
Circuit Breaker	temporary blocks possible failures	When a system is seriously struggling, failing fast is better than making clients wait.	overview, documentation, Feign, Retrofit, Spring
Rate Limiter	limits executions/period	Limit the rate of incoming requests.	overview, documentation, Feign, Retrofit, Spring
Time Limiter	limits duration of execution	Beyond a certain wait interval, a successful result is unlikely.	documentation, Retrofit, Spring
Bulkhead	limits concurrent executions	Resources are isolated into pools so that if one fails, the others will continue working.	overview, documentation, Spring
Cache	memorizes a successful result	Some proportion of requests may be similar.	documentation
Fallback	provides an alternative result for failures	Things will still fail - plan what you will do when that happens.	Try::recover, Spring, Feign

Above table is based on Polly: resilience policies.

To find more information about resilience patterns check **Talks** section. Find out more about components in our **User Guide**.

5. Spring Boot

Setup and usage in Spring Boot 2 is demonstrated here.

6. Usage examples

6.1. 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
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))
```

Resilience4j also provides RxJava operators for `RateLimiter`, `Bulkhead`, `TimeLimiter` and `Retry`. Find out more in our User Guide.

6.1.2. 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)
    .transformDeferred(CircuitBreakerOperator.of(circuitBreaker))
```

Resilience4j also provides Reactor operators for `RateLimiter`, `Bulkhead`, `TimeLimiter` and `Retry`. Find out more in our User Guide.

6.2. 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);
```

6.3. Bulkhead

There are two isolation strategies and bulkhead implementations.

6.3.1. 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)
    .maxWaitDuration(100)
    .build();

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

Supplier<String> supplier = Bulkhead
    .decorateSupplier(bulkhead, backendService::doSomething);
```

6.3.2. 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);
```

7. 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.

7.1. 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(...))
```

ⓘ	You can also consume events from other components. Find out more in our **User Guide**.

8. Talks

0:34	Battle of the Circuit Breakers: Resilience4J vs Istio	Nicolas Frankel	GOTO Berlin
0:33	Battle of the Circuit Breakers: Istio vs. Hystrix/Resilience4J	Nicolas Frankel	JFuture

0:42	Resilience patterns in the post-Hystrix world	Tomasz Skowroński	Cloud Native Warsaw
0:52	Building Robust and Resilient Apps Using Spring Boot and Resilience4j	David Caron	SpringOne
0:22	Hystrix is dead, now what?	Tomasz Skowroński	DevoxxPL

9. Companies that use Resilience4j

- **Deutsche Telekom** (In an application with over 400 million requests per day)
- **AOL** (In an application with low latency requirements)
- **Netpulse** (In a system with 40+ integrations)
- **wescale.de** (In a B2B integration platform)
- **Topia** (In an HR application built with microservices architecture)
- **Auto Trader Group plc** (The largest Britain digital automotive marketplace)
- **PlayStation Network** (A platform backend)

10. License

README.adoc

Releases 29

Release v1.7.1 Latest