(/)

Guide to Resilience4j

Last modified: May 6, 2020

by baeldung (https://www.baeldung.com/author/baeldung/)

Cloud (https://www.baeldung.com/category/cloud/)
DevOps (https://www.baeldung.com/category/devops/)

1. Overview

In this tutorial, we'll talk about the Resilience4j (https://resilience4j.github.io/resilience4j/) library.

The library helps with implementing resilient systems by managing fault tolerance for remote communications.

The library is inspired by Hystrix (/introduction-to-hystrix) but offers a much more convenient API and a number of other features like Rate Limiter (block too frequent requests), Bulkhead (avoid too many concurrent requests) etc.

2. Maven Setup

To start, we need to add the target modules to our *pom.xml* (e.g. here we add the Circuit Breaker)*:*

```
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-circuitbreaker</artifactId>
    <version>0.12.1</version>
</dependency>
```

Here, we're using the *circuitbreaker* module. All modules and their latest versions can be found on Maven Central (https://search.maven.org/classic/#search%7Cga%7C1%7Cg%3A%22io.github.resilience4j%22).

In the next sections, we'll go through the most commonly used modules of the library.

3. Circuit Breaker

Note that for this module we need the *resilience4j-circuitbreaker* dependency shown above.

The Circuit Breaker pattern (https://martinfowler.com/bliki/CircuitBreaker.html) helps us in preventing a cascade of failures when a remote service is down.

After a number of failed attempts, we can consider that the service is unavailable/overloaded and eagerly reject all subsequent requests to it. In this way, we can save system resources for calls which are likely to fail.

Let's see how we can achieve that with Resilience4j.

First, we need to define the settings to use. The simplest way is to use default settings:

```
CircuitBreakerRegistry circuitBreakerRegistry
  = CircuitBreakerRegistry.ofDefaults();
```

It's also possible to use custom parameters:

```
CircuitBreakerConfig config = CircuitBreakerConfig.custom()
  .failureRateThreshold(20)
  .ringBufferSizeInClosedState(5)
  .build();
```

Here, we've set the rate threshold to 20% and a minimum number of 5 call attempts.

Then, we create a *CircuitBreaker* object and call the remote service through it:

```
interface RemoteService {
    int process(int i);
}

CircuitBreakerRegistry registry = CircuitBreakerRegistry.of(config);
CircuitBreaker circuitBreaker = registry.circuitBreaker("my");
Function<Integer, Integer> decorated = CircuitBreaker
  .decorateFunction(circuitBreaker, service::process);
```

Finally, let's see how this works through a JUnit test.

Deliver an Engaging Employee Experience with an Intelligence-driven Digital Workspace Platform.
Simply and securely deliver and manage any app on any device with VMware Workspace ONE, an intelligence-driven digital workspace platform.
LEARN MORE
TechData
vmw

We'll attempt to call the service 10 times. We should be able to verify that the call was attempted a minimum of 5 times, then stopped as soon as 20% of calls failed:

```
when(service.process(any(Integer.class))).thenThrow(new
RuntimeException());

for (int i = 0; i < 10; i++) {
    try {
        decorated.apply(i);
    } catch (Exception ignore) {}
}

verify(service, times(5)).process(any(Integer.class));
```

3.1. Circuit Breaker's States and Settings

A *CircuitBreaker* can be in one of the three states:

- *CLOSED* – everything is fine, no short-circuiting involved
- *OPEN* – remote server is down, all requests to it are short-circuited

- *HALF_OPEN* – a configured amount of time since entering OPEN state has elapsed and *CircuitBreaker* allows requests to check if the remote service is back online

We can configure the following settings:

- the failure rate threshold above which the *CircuitBreaker* opens and starts short-circuiting calls
- the wait duration which defines how long the *CircuitBreaker* should stay open before it switches to half open
- the size of the ring buffer when the *CircuitBreaker* is half open or closed
- a custom *CircuitBreakerEventListener* which handles *CircuitBreaker* events
- a custom *Predicate* which evaluates if an exception should count as a failure and thus increase the failure rate

4. Rate Limiter

Similar to the previous section, this features requires the *resilience4j-ratelimiter* (https://search.maven.org/classic/#search%7Cga%7C1%7Ca%3A%22resilience4j-ratelimiter%22) dependency.

As the name implies, **this functionality allows limiting access to some service**. Its API is very similar to *CircuitBreaker's* – there are *Registry*, *Config* and *Limiter* classes.

Here's an example of how it looks:

```
RateLimiterConfig config =
RateLimiterConfig.custom().limitForPeriod(2).build();
RateLimiterRegistry registry = RateLimiterRegistry.of(config);
RateLimiter rateLimiter = registry.rateLimiter("my");
Function<Integer, Integer> decorated
  = RateLimiter.decorateFunction(rateLimiter, service::process);
```

Now all calls on the decorated service block if necessary to conform to the rate limiter configuration.

We can configure parameters like:

- the period of the limit refresh
- the permissions limit for the refresh period
- the default wait for permission duration

5. Bulkhead

Here, we'll first need the *resilience4j-bulkhead* (https://search.maven.org/classic/#search%7Cga%7C1%7Cresilience4j-bulkhead) dependency.

It's possible **to limit the number of concurrent calls to a particular service.**

Let's see an example of using the Bulkhead API to configure a max number of one concurrent calls:

```
BulkheadConfig config =
BulkheadConfig.custom().maxConcurrentCalls(1).build();
BulkheadRegistry registry = BulkheadRegistry.of(config);
Bulkhead bulkhead = registry.bulkhead("my");
Function<Integer, Integer> decorated
  = Bulkhead.decorateFunction(bulkhead, service::process);
```

To test this configuration, we'll call a mock service's method.

Then, we ensure that *Bulkhead* doesn't allow any other calls:

```
CountDownLatch latch = new CountDownLatch(1);
when(service.process(anyInt())).thenAnswer(invocation -> {
    latch.countDown();
    Thread.currentThread().join();
    return null;
});

ForkJoinTask<?> task = ForkJoinPool.commonPool().submit(() -> {
    try {
        decorated.apply(1);
    } finally {
        bulkhead.onComplete();
    }
});
latch.await();
assertThat(bulkhead.isCallPermitted()).isFalse();
```

We can configure the following settings:

- the max amount of parallel executions allowed by the bulkhead
- the max amount of time a thread will wait for when attempting to enter a saturated bulkhead

6. Retry

For this feature, we'll need to add the *resilience4j-retry* (https://search.maven.org/classic/#search%7Cga%7C1%7Cresilience4j-retry) library to the project.

We can **automatically retry a failed call** using the Retry API:

```
RetryConfig config = RetryConfig.custom().maxAttempts(2).build();
RetryRegistry registry = RetryRegistry.of(config);
Retry retry = registry.retry("my");
Function<Integer, Void> decorated
  = Retry.decorateFunction(retry, (Integer s) -> {
        service.process(s);
        return null;
    });
```

Now let's emulate a situation where an exception is thrown during a remote service call and ensure that the library automatically retries the failed call:

```
when(service.process(anyInt())).thenThrow(new RuntimeException());
try {
    decorated.apply(1);
    fail("Expected an exception to be thrown if all retries failed");
} catch (Exception e) {
    verify(service, times(2)).process(any(Integer.class));
}
```

We can also configure the following:

- the max attempts number
- the wait duration before retries
- a custom function to modify the waiting interval after a failure
- a custom *Predicate* which evaluates if an exception should result in retrying the call

7. Cache

The Cache module requires the *resilience4j-cache* (https://search.maven.org/classic/#search%7Cga%7C1%7Cresilience4j-cache) dependency.

The initialization looks slightly different than the other modules:

```
javax.cache.Cache cache = ...; // Use appropriate cache here
Cache<Integer, Integer> cacheContext = Cache.of(cache);
Function<Integer, Integer> decorated
  = Cache.decorateSupplier(cacheContext, () -> service.process(1));
```

Here the caching is done by the JSR-107 Cache (/jcache) implementation used and Resilience4j provides a way to apply it.

Note that there is no API for decorating functions (like *Cache.decorateFunction(Function)*), the API only supports *Supplier* and *Callable* types.

8. TimeLimiter

For this module, we have to add the *resilience4j-timelimiter* (https://search.maven.org/classic/#search%7Cga%7C1%7Cresilience4j-timelimiter) dependency.

It's possible to **limit the amount of time spent calling a remote service** using the TimeLimiter.

To demonstrate, let's set up a *TimeLimiter* with a configured timeout of 1 millisecond:

```
long ttl = 1;
TimeLimiterConfig config
  =
TimeLimiterConfig.custom().timeoutDuration(Duration.ofMillis(ttl)).build
();
TimeLimiter timeLimiter = TimeLimiter.of(config);
```

Next, let's verify that Resilience4j calls *Future.get()* with the expected timeout:

```
Future futureMock = mock(Future.class);
Callable restrictedCall
  = TimeLimiter.decorateFutureSupplier(timeLimiter, () -> futureMock);
restrictedCall.call();

verify(futureMock).get(ttl, TimeUnit.MILLISECONDS);
```

We can also combine it with *CircuitBreaker*:

```
Callable chainedCallable
  = CircuitBreaker.decorateCallable(circuitBreaker, restrictedCall);
```

9. Add-on Modules

Resilience4j also offers a number of add-on modules which ease its integration with popular frameworks and libraries.

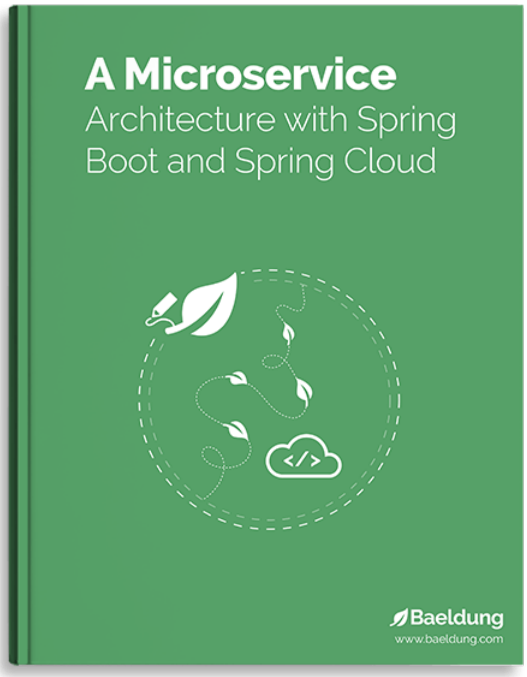
Some of the more well-known integrations are:

- Spring Boot – *resilience4j-spring-boot* module
- Ratpack – *resilience4j-ratpack* module
- Retrofit – *resilience4j-retrofit* module
- Vertx – *resilience4j-vertx* module
- Dropwizard – *resilience4j-metrics* module
- Prometheus – *resilience4j-prometheus* module

10. Conclusion

In this article, we went through different aspects of the Resilience4j library and learned how to use it for addressing various fault-tolerance concerns in inter-server communications.

As always, the source code for the samples above can be found over on GitHub (https://github.com/eugenp/tutorials/tree/master/libraries-6).



Comments are closed on this article!

COURSES

ALL COURSES (/ALL-COURSES)
ALL BULK COURSES (/ALL-BULK-COURSES)
THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)
SPRING REACTIVE TUTORIALS (/SPRING-REACTIVE-GUIDE)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)
WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORS (/EDITORS)
JOBS (/TAG/ACTIVE-JOB/)
OUR PARTNERS (/PARTNERS)
ADVERTISE ON BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)