AARHUS UNIVERSITET

Microservices and DevOps

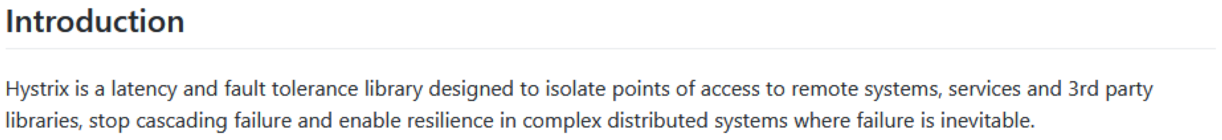
Scalable Microservices

Resilience4J: A Stability Pattern library

Henrik Bærbak Christensen

History

- Hystrix,
  - Developed by NetFlix

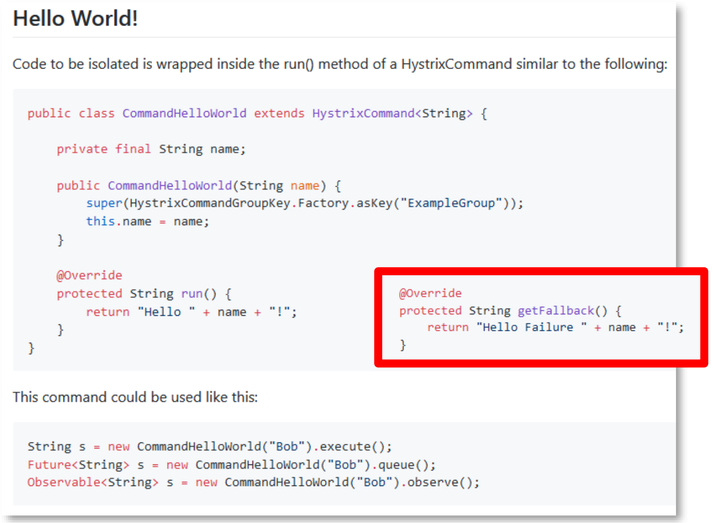


Introduction

Hystrix is a latency and fault tolerance library designed to isolate points of access to remote systems, services and 3rd party libraries, stop cascading failure and enable resilience in complex distributed systems where failure is inevitable.

- Central Hystrix abstraction:
  - The **Command pattern**
    - *Make a method into an object*

- *Disclaimer*
  - *I have never used it…*



Hello World!

Code to be isolated is wrapped inside the run() method of a HystrixCommand similar to the following:

```
public class CommandHelloWorld extends HystrixCommand<String> {

    private final String name;

    public CommandHelloWorld(String name) {
        super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        this.name = name;
    }

    @Override
    protected String run() {
        return "Hello " + name + "!";
    }
}
```

```
@Override
protected String getFallback() {
    return "Hello Failure " + name + "!";
}
```

This command could be used like this:

```
String s = new CommandHelloWorld("Bob").execute();
Future<String> s = new CommandHelloWorld("Bob").queue();
Observable<String> s = new CommandHelloWorld("Bob").observe();
```

Resilience4J

- … is
  - *A lightweight, easy-to-use fault tolerance library inspired by Hystrix, but designed for Java 8 and functional programming.*

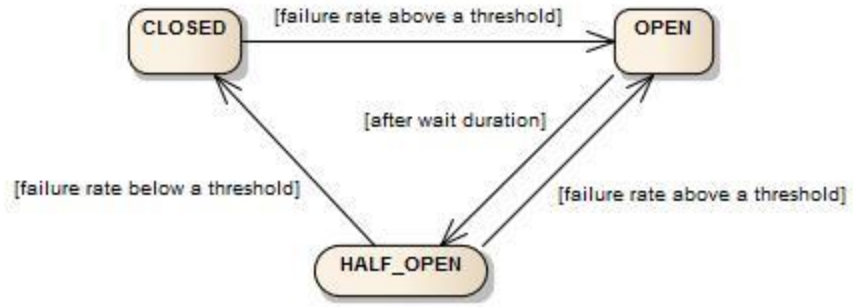
- You can pick and choose just the piece you want

– CircuitBreaker:	Nygard's pattern in it's *frequency* form
– Bulkhead:	Limit number of concurrent executions
– RateLimiter:	Limit rate of requests (or queue them)
– Retry:	Retry call N times with M mS delay between
– TimeLimiter:	Nygard's *Fail Fast* pattern
– Cache:	You guessed it ☺

AARHUS UNIVERSITET

Circuit Breaker

- The standard state machine
  - Not count of failures, but *failure rate*


CLOSED
[failure rate above a threshold]
OPEN
[after wait duration]
[failure rate below a threshold]
[failure rate above a threshold]
HALF_OPEN

  - Two ways of calculating *rate*
    - Count-based sliding window
      - One out of last 5 calls failed = 20% failure rate
    - Time-based sliding window
      - 5 seconds is 5 ‘1 second epoch buckets’
        - » Each bucket aggregate all calls that second and mark ‘pass/fail’
      - 20% failure rate = 1 out of 5 last buckets was a fail

Circuit Breaker

- Closed to Open state transitions happen when
  - Failure rate is exceeded
  - As well as *slow response rate exceeded*
- So both ‘failed’ and ‘slow’ executions are recorded!

- If call to CB in OPEN state, then
  - *CallNotPermittedException* is thrown
    - Catch it to provide the ‘safe failure mode’ response

Circuit Breaker

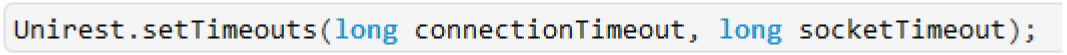
- Half Open state
  - You can configure the number of calls to make in half open
    - Contrast Nygard, who has this number at a single call
  - Example: Half Open calls set to 3
    - Three calls are made, and standard failure rate computation then is used to determine state change to either Open or Closed
    - Forth and consecutive calls are handled as Open calls
      - Throw the *CallNotPermittedException* exception

AARHUS UNIVERSITET

Circuit Breaker

- Note: The CB does NOT itself implement *time out*
  - Setting the slow response time to 10 seconds
    - slowCallDurationThreshold
  - Will still make the CB wait for 8 minutes, if the call really takes 8 minutes to complete!

- Solution
  - Use your REST client library’s time out facilities



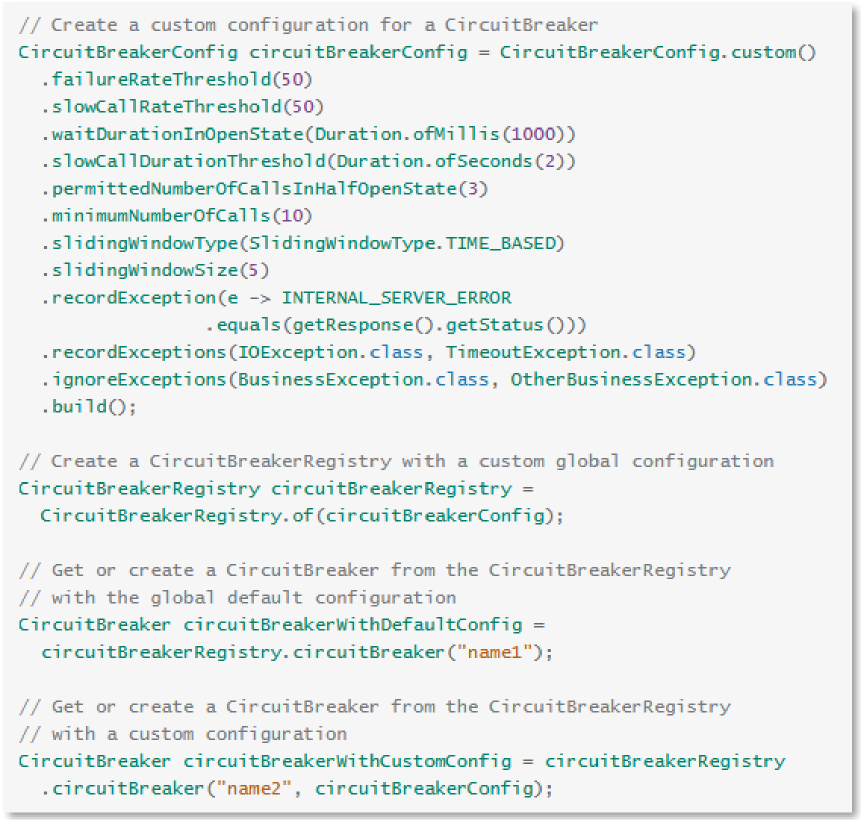
```
Unirest.setTimeouts(long connectionTimeout, long socketTimeout);
```

  - Or – use the Resilience4J’s TimeLimiter

AARHUS UNIVERSITET

Circuit Breaker

- CB can be configured in a zillion ways ☹ ☺



```java
// 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);
```

AARHUS UNIVERSITET

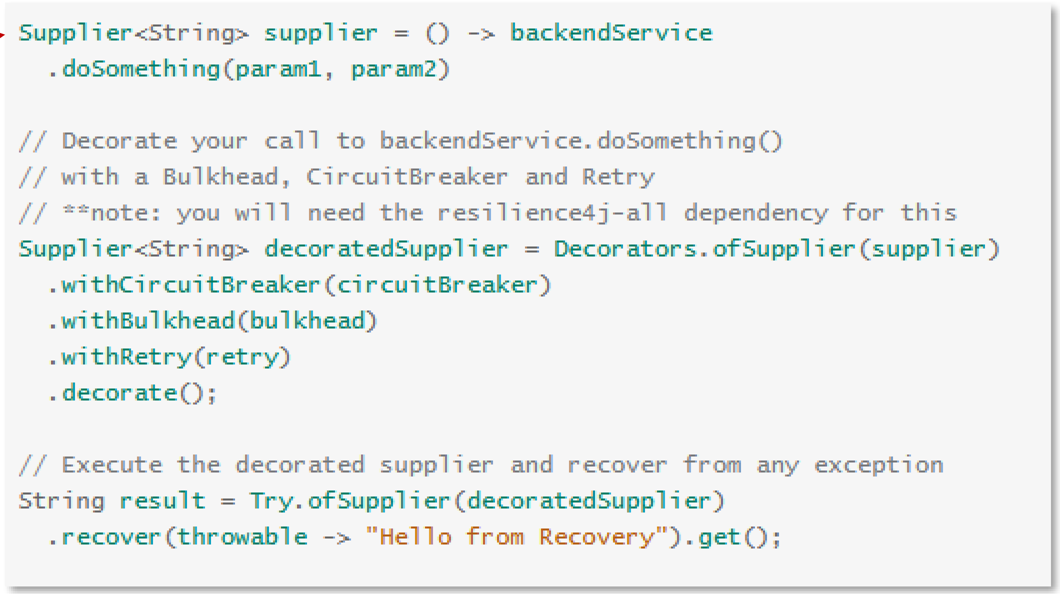
“Decorators”

- Resilience4J relies on the functional version of **Decorator(*)** pattern, using Java8 functional abstractions
  - Meaning they can be ‘onion-like’ wrapped

Integration point

Decorators introduce CB, bulkhead, and retry

Actual call site



```
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();
```

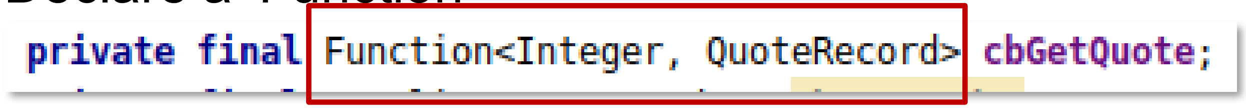
AARHUS UNIVERSITET

“Decorators”

- They call it ‘decorators’ but it find it is actually the ‘proxy’ pattern
- Decorator: *Add additional responsibilities to object*
- Proxy: *Provide a placeholder that controls access to object*
- Anyway – they are behaviorally equivalent…

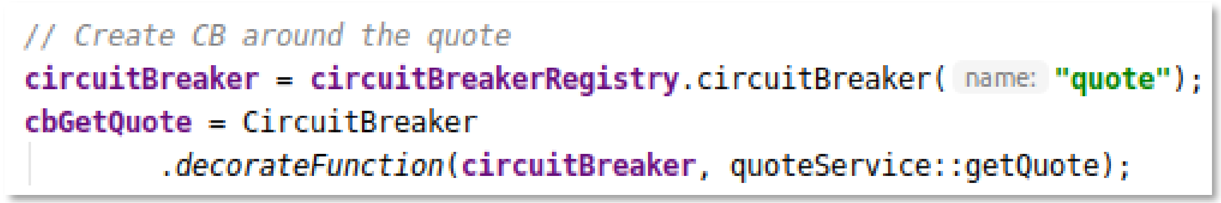
Coding It

- Each integration point (= call to remote service) must be defined as a Java8 functional abstraction
  - Callable, Supplier, Function, …
- Example: QuoteService’s getQuote method
  - Declare a ‘Function’



```
private final Function<Integer, QuoteRecord> cbGetQuote;
```

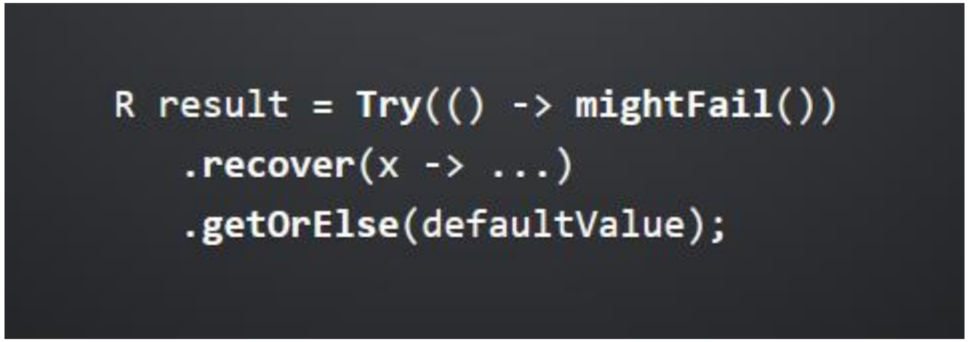
  - And create it as a decorator on a real quote service call



```
// Create CB around the quote
circuitBreaker = circuitBreakerRegistry.circuitBreaker( name: "quote");
cbGetQuote = CircuitBreaker
        .decorateFunction(circuitBreaker, quoteService::getQuote);
```

Coding It

- The functional approach and reliance on Varv library is a bit of a ‘challenge’ for old-school procedural coders like me…
- But there is something about it

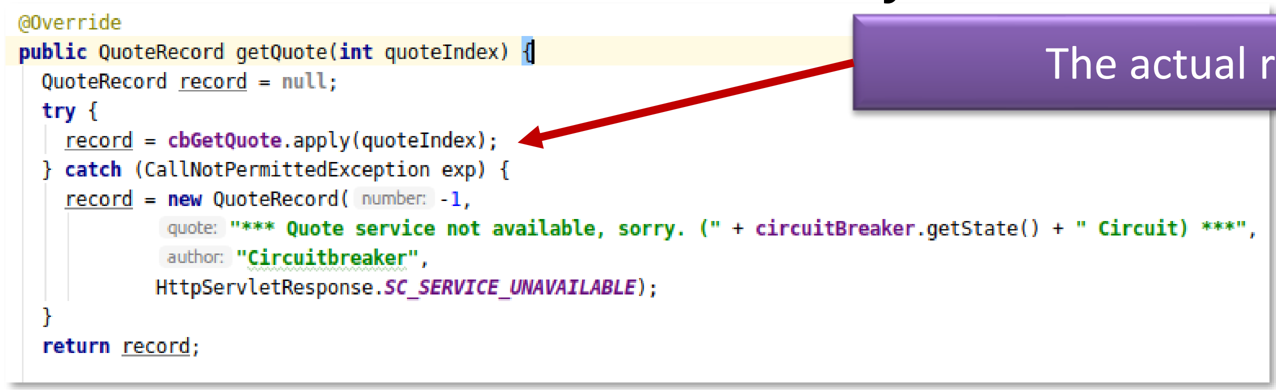


```
R result = Try(() -> mightFail())
    .recover(x -> ...)
    .getOrElse(defaultValue);
```

AARHUS UNIVERSITET

Example

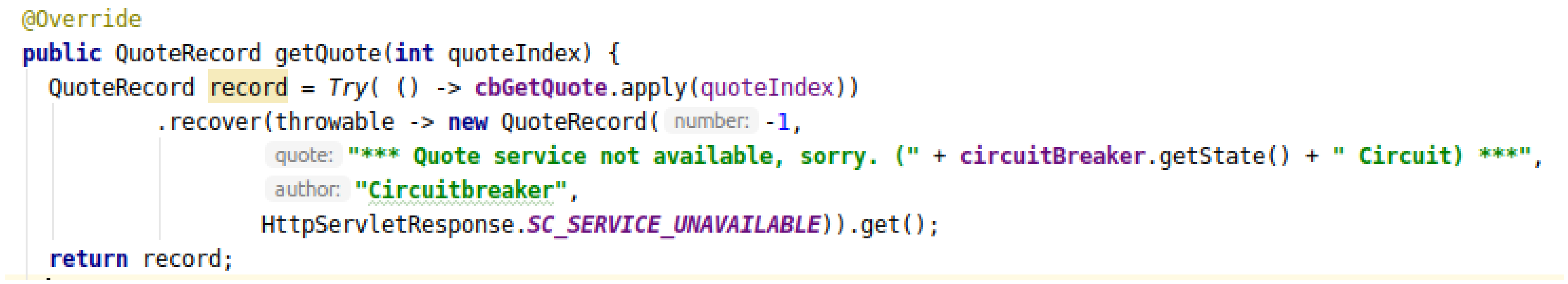
- Old-school = I wrote this initially ☺



```
@Override
public QuoteRecord getQuote(int quoteIndex) {
  QuoteRecord record = null;
  try {
    record = cbGetQuote.apply(quoteIndex);
  } catch (CallNotPermittedException exp) {
    record = new QuoteRecord( number: -1,
            quote: "*** Quote service not available, sorry. (" + circuitBreaker.getState() + " Circuit) ***",
            author: "Circuitbreaker",
            HttpServletResponse.SC_SERVICE_UNAVAILABLE);
  }
  return record;
```

The actual remote call

- The Varv way has quite some value, though ☺



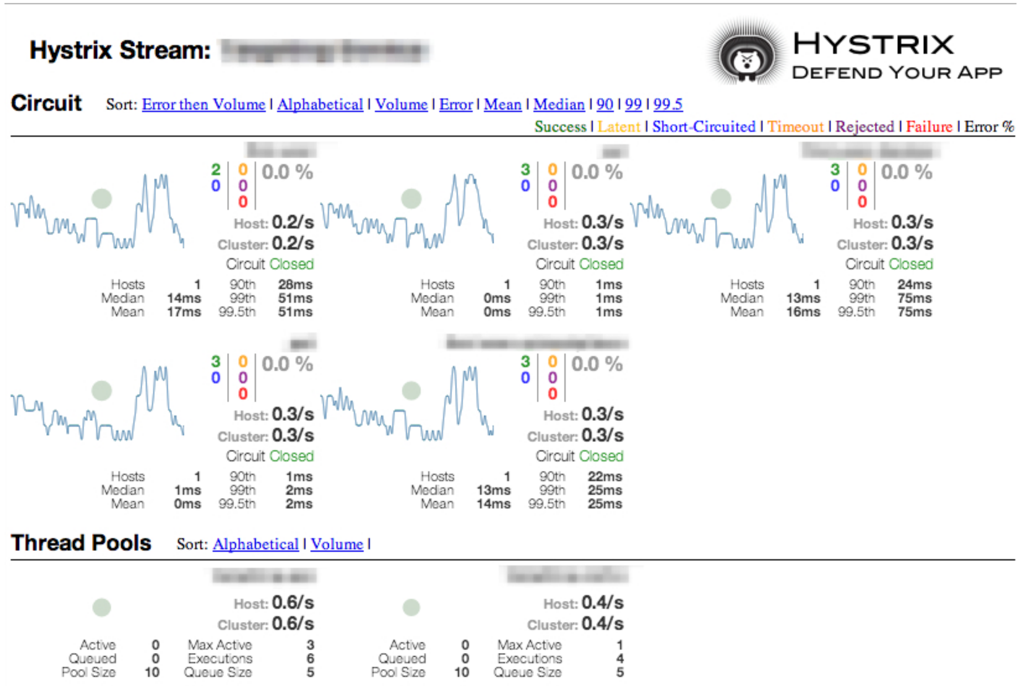
```
@Override
public QuoteRecord getQuote(int quoteIndex) {
  QuoteRecord record = Try( () -> cbGetQuote.apply(quoteIndex))
          .recover(throwable -> new QuoteRecord( number: -1,
                  quote: "*** Quote service not available, sorry. (" + circuitBreaker.getState() + " Circuit) ***",
                  author: "Circuitbreaker",
                  HttpServletResponse.SC_SERVICE_UNAVAILABLE)).get();
  return record;
```

Monitorability

Resilience/Stability/Availability
Monitoring

Monitoring your CBs

- CB states tell a lot about the state of your architecture



Hystrix Stream: [redacted]

HYSTRIX
DEFEND YOUR APP

Circuit Sort: Error then Volume | Alphabetical | Volume | Error | Mean | Median | 90 | 99 | 99.5

Success | Latent | Short-Circuited | Timeout | Rejected | Failure | Error %

[redacted]
2 | 0 | 0.0 %
0 | 0
0
Host: 0.2/s
Cluster: 0.2/s
Circuit Closed

Hosts	1	90th	28ms
Median	14ms	99th	51ms
Mean	17ms	99.5th	51ms

[redacted]
3 | 0 | 0.0 %
0 | 0
0
Host: 0.3/s
Cluster: 0.3/s
Circuit Closed

Hosts	1	90th	1ms
Median	0ms	99th	1ms
Mean	0ms	99.5th	1ms

[redacted]
3 | 0 | 0.0 %
0 | 0
0
Host: 0.3/s
Cluster: 0.3/s
Circuit Closed

Hosts	1	90th	24ms
Median	13ms	99th	75ms
Mean	16ms	99.5th	75ms

[redacted]
3 | 0 | 0.0 %
0 | 0
0
Host: 0.3/s
Cluster: 0.3/s
Circuit Closed

Hosts	1	90th	1ms
Median	1ms	99th	2ms
Mean	0ms	99.5th	2ms

[redacted]
3 | 0 | 0.0 %
0 | 0
0
Host: 0.3/s
Cluster: 0.3/s
Circuit Closed

Hosts	1	90th	22ms
Median	13ms	99th	25ms
Mean	14ms	99.5th	25ms

Thread Pools Sort: Alphabetical | Volume |

[redacted]
Host: 0.6/s
Cluster: 0.6/s

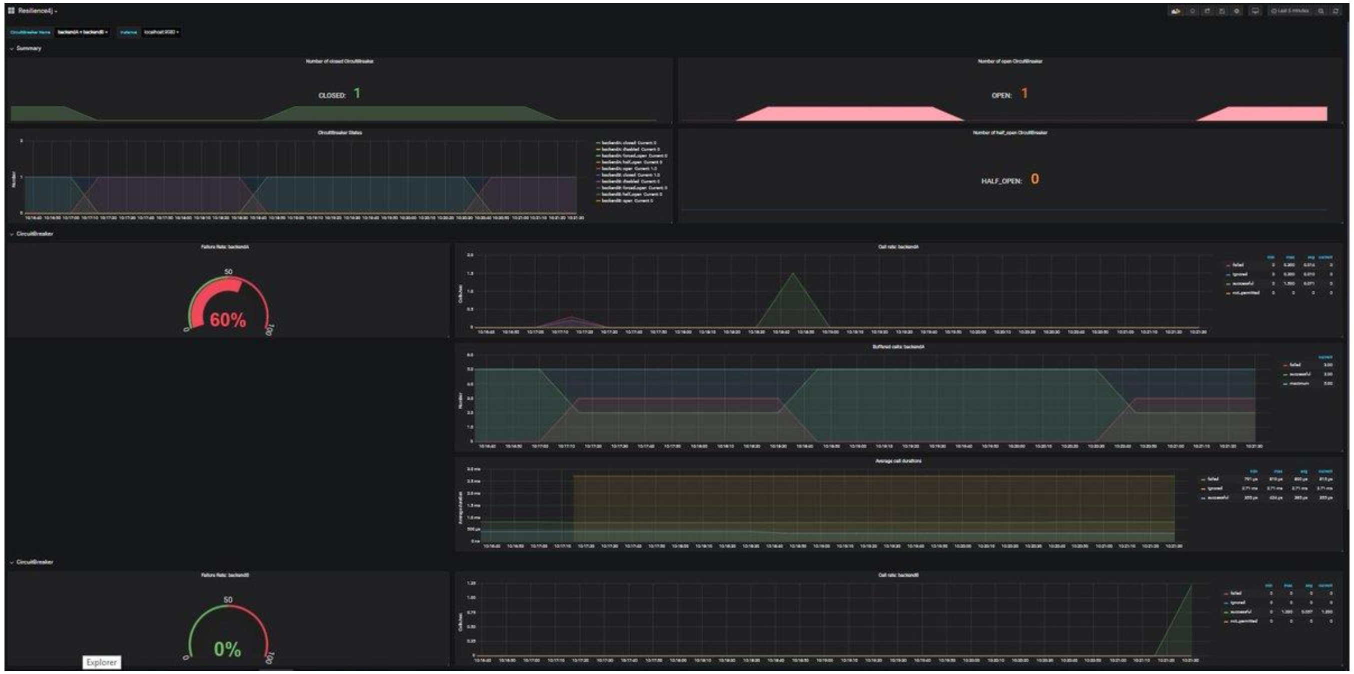
Active	0	Max Active	3
Queued	0	Executions	6
Pool Size	10	Queue Size	5

[redacted]
Host: 0.4/s
Cluster: 0.4/s

Active	0	Max Active	1
Queued	0	Executions	4
Pool Size	10	Queue Size	5

Can Do

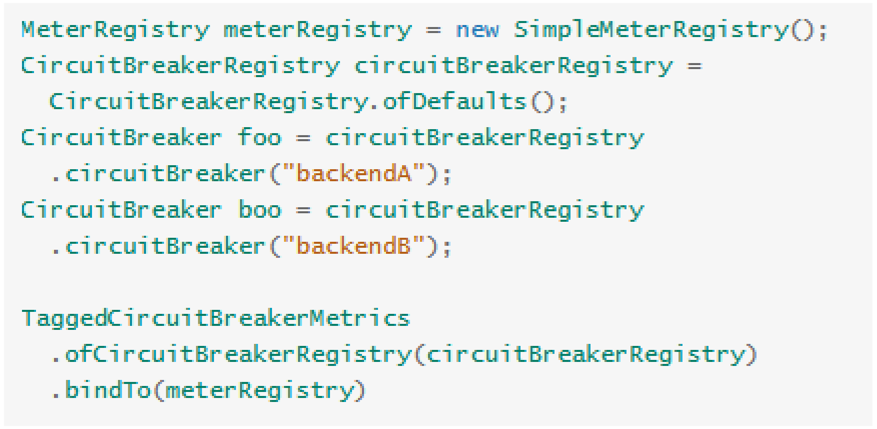
- Resilience4J integrate with Micrometer and Grafana



Micrometer

- Micrometer is Dropwizard's replacement
  – They say themselves ☺
- Micrometer integrate with Humio ☺
  – Probably have to have the paid version…

- The Newman
  Data Pumps pattern
  (p 96 ff, 'modernization' slides)



```
MeterRegistry meterRegistry = new SimpleMeterRegistry();
CircuitBreakerRegistry circuitBreakerRegistry =
  CircuitBreakerRegistry.ofDefaults();
CircuitBreaker foo = circuitBreakerRegistry
  .circuitBreaker("backendA");
CircuitBreaker boo = circuitBreakerRegistry
  .circuitBreaker("backendB");

TaggedCircuitBreakerMetrics
  .ofCircuitBreakerRegistry(circuitBreakerRegistry)
  .bindTo(meterRegistry)
```

Summary

- Cool stuff…
  - Avoid a lot of hand-coding myself

- But
  - Documentation assumes you know your patterns beforehand
    - Delve into tech detail, misses the intro and big picture ☹
  - A bit of a learning curve
  - Zillion handles to crank
  - And forced to learn bits of Java8 functional programming style…