Resilience4j

Taking order-service's inventory-service calls beyond manual try/catch error handling: applying Resilience4j's @CircuitBreaker (a CLOSED/OPEN/HALF_OPEN state machine) and @Retry annotations to ResilientStockClient, what fallback methods do when the circuit opens, why Spring's proxy mechanism doesn't work through self-invocation, and how a rate limiter/bulkhead protects against a different risk than a circuit breaker.

Intermediate 24 min

Last updated: August 23, 2026
Read online: https://www.learnforgex.com/en/topics/resilience4j

---

Resilience4j

The Inter-Service Communication lesson already taught order-service to tell a 404 apart from a connection failure when calling inventory-service, and to translate the latter into an `InventoryServiceUnavailableException` (see its "The Network Is Unreliable: What If inventory-service Is Down?" section). That's honest error handling -- but it has a gap: if inventory-service is down for even a FEW seconds, EVERY single order-service request that needs stock data fails immediately and identically, and order-service keeps hammering a service that's clearly struggling. This lesson introduces the library that fixes both problems: Resilience4j.

What Is Resilience4j?

Resilience4j is a lightweight Java library that wraps a risky call -- a call to another service, most often -- with one or more PROTECTIVE behaviors: a circuit breaker (stop calling a service that's clearly failing), a retry (try again before giving up), a rate limiter (cap how often a call is allowed to happen), and a bulkhead (cap how many calls can be IN FLIGHT at once). Each behavior is applied with an annotation on a method -- the method's own code stays focused on what it actually does, not on how to survive failure.

Why Does It Exist?

Catching `ResourceAccessException` and throwing `InventoryServiceUnavailableException` (as StockClientWithDiscovery already does) is necessary but not SUFFICIENT. Two real problems remain: first, if inventory-service is down, order-service keeps trying every single request, wasting time waiting for connections that will fail anyway, and adding load to a service

that's already struggling to recover. Second, a single flaky network blip shouldn't fail a request outright if trying ONE more time would likely succeed. Handling both well by hand -- tracking failure counts, deciding when to stop calling, retrying with the right pauses -- is exactly the kind of infrastructure code that's easy to get subtly wrong. Resilience4j provides it as configuration instead.

History

Resilience4j was created in 2016, explicitly as a REPLACEMENT for Netflix Hystrix (Netflix's own circuit breaker library, part of the same era as Zuul and Eureka -- see the Service Discovery & Eureka and API Gateway lessons' "History" sections). Hystrix was put into maintenance mode by Netflix in 2018, the same general period Netflix stepped back from several of its open-source infrastructure tools. Resilience4j was designed from the start to be lighter weight (built for Java 8+ functional style, no dependency on RxJava like Hystrix had) and modular -- a project can depend on just the circuit breaker module, just retry, or any combination, instead of one large library.

Adding Resilience4j to order-service

Resilience4j integrates with Spring Boot through the `resilience4j-spring-boot3` starter and annotation-driven configuration in `application.yml` -- no separate server or infrastructure piece is needed, unlike Eureka Server or api-gateway; every protective behavior runs INSIDE order-service itself.

```
# Added on top of order-service's existing application.yml (see the Spring Boot
# Microservice Basics lesson's "Its Own `application.yml`: Port, Application Name,
# and Database" section, and the eureka.client block from the Service Discovery &
# Eureka lesson) -- nothing already there changes, this is purely additive.

resilience4j:
  circuitbreaker:
    instances:
      inventoryService:                          # this name is what @CircuitBreaker
                                                   # in ResilientStockClient refers to
                                                   # -- it does NOT have to match
                                                   # "inventory-service" (the Eureka
                                                   # name), they're independent
        sliding-window-type: COUNT_BASED
        sliding-window-size: 10                   # look at the last 10 calls
        failure-rate-threshold: 50                # if 50% or more of them failed,
                                                    # OPEN the circuit
        wait-duration-in-open-state: 10s          # stay OPEN for 10 seconds before
                                                    # trying a HALF_OPEN test call
        permitted-number-of-calls-in-half-open-state: 3
  retry:
    instances:
      inventoryService:
        max-attempts: 3                           # the ORIGINAL call plus 2 retries
        wait-duration: 500ms                      # pause between attempts
```

> The instance name (`inventoryService` above) is a name Resilience4j uses INTERNALLY to group configuration -- it doesn't have to match the Eureka service name (`inventory-service`) at all, though picking a related name keeps things easy to follow.

Circuit Breaker: States and Configuration

A circuit breaker has three states. CLOSED is the normal state -- calls go through, and failures are counted. If the failure rate crosses a configured threshold, the circuit trips to OPEN -- every call fails IMMEDIATELY, without even attempting the real call, for a configured wait duration. After that wait, the circuit moves to HALF_OPEN, where a small number of TEST calls are allowed through -- if they succeed, the circuit closes again; if they fail, it reopens.

Wrapping StockClient with a Circuit Breaker

The `@CircuitBreaker` annotation applies this state machine to a single method -- no change to the method's own logic is needed, only its signature and a fallback method.

```
import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;
import io.github.resilience4j.retry.annotation.Retry;
import org.springframework.stereotype.Component;
import org.springframework.web.client.HttpClientErrorException;
import org.springframework.web.client.ResourceAccessException;
import org.springframework.web.client.RestClient;

// Builds directly on the Service Discovery & Eureka lesson's StockClientWithDiscovery
// (see its own file, and the "Calling a Service by Name with a Load-Balanced RestClient"
// section) -- the constructor and the discovery-aware base URL are UNCHANGED. What's NEW
// is the two annotations on checkStock: @CircuitBreaker and @Retry, both referring to
// the "inventoryService" instance configured in Resilience4jConfig.yml. Neither
// annotation touches the METHOD BODY at all -- Resilience4j wraps the call from the
// OUTSIDE, at the proxy level, exactly like @Transactional does (see the Transaction
// Management lesson).
@Component
class ResilientStockClient {

    private final RestClient restClient;

    ResilientStockClient(RestClient.Builder loadBalancedRestClientBuilder) {
        this.restClient = loadBalancedRestClientBuilder.baseUrl("http://inventory-service").
build();
    }

    // Retry runs FIRST (innermost) -- Resilience4j retries the call up to
    // resilience4j.retry.instances.inventoryService.max-attempts times BEFORE the
    // circuit breaker ever records a single failure from this call. Only once retries
    // are exhausted does the circuit breaker see "this call failed" and count it toward
    // opening the circuit. If the circuit is ALREADY open, neither the method body nor
    // any retry attempt runs at all -- checkStockFallback is called immediately.
    @CircuitBreaker(name = "inventoryService", fallbackMethod = "checkStockFallback")
    @Retry(name = "inventoryService")
```

```
    StockCheckResponse checkStock(String productName) {
        try {
            return restClient.get()
                    .uri("/inventory/{productName}", productName)
                    .retrieve()
                    .body(StockCheckResponse.class);
        } catch (HttpClientErrorException.NotFound e) {
            // Same meaning as in every earlier lesson: inventory-service answered, it
            // just doesn't know this product -- this is NOT a failure Resilience4j
            // should retry or count against the circuit breaker.
            return new StockCheckResponse(productName, 0);
        } catch (ResourceAccessException e) {
            throw new InventoryServiceUnavailableException("inventory-service is not
reachable", e);
        }
    }

    // The fallback method's signature MUST match checkStock's (same parameters), plus
    // one extra Throwable parameter at the end -- Resilience4j calls THIS method
    // instead, with the exception that finally triggered it, whenever the circuit is
    // OPEN or every retry attempt has failed. Returning a degraded-but-VALID
    // StockCheckResponse here (instead of letting the exception propagate) is a
    // deliberate choice: order-service can still let the order proceed, treating stock
    // as "unknown, assume none reserved" rather than failing the whole request just
    // because inventory-service is having trouble.
    StockCheckResponse checkStockFallback(String productName, Throwable t) {
        return new StockCheckResponse(productName, 0);
    }
}

// Unchanged from earlier lessons -- what CAN fail about the call didn't change, only
// what order-service now DOES in response (see checkStockFallback above, instead of
// letting this propagate all the way up).
class InventoryServiceUnavailableException extends RuntimeException {
    InventoryServiceUnavailableException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

```
// Unchanged from the Service Discovery & Eureka lesson (itself unchanged from Inter-
// Service Communication) -- wrapping the call in a circuit breaker and retry doesn't
// change WHAT inventory-service tells order-service, only what happens when it CAN'T
// be reached at all.
record StockCheckResponse(String productName, int quantityInStock) {
}
```

> `@CircuitBreaker` only works through Spring's PROXY mechanism, exactly like `@Transactional` (see the Transaction Management lesson) -- calling `checkStock` from ANOTHER method on the SAME class (`this.checkStock(...)`) bypasses the proxy entirely, and neither the circuit breaker nor the retry would ever run.

Fallback Methods: What Happens When the Circuit Opens?

A fallback method is what runs INSTEAD of the real method, whenever the circuit is open or every retry attempt has failed -- its signature must match the original method's parameters, plus one extra `Throwable` at the end. `checkStockFallback` above returns a degraded-but-valid `StockCheckResponse` rather than letting the exception propagate -- a deliberate choice to let order processing continue with "assume no stock reserved" instead of failing the whole request outright.

Retry: Trying Again Before Giving Up

`@Retry`, also visible on `checkStock` above, retries a failed call a configured number of times, with a pause between attempts, BEFORE the circuit breaker records a failure at all. This matters for genuinely transient problems -- a single dropped packet, a brief network blip -- where trying again immediately would likely succeed, and giving up on the very first failure would be premature.

> Retry and circuit breaker are not competing choices -- they answer different questions. Retry asks "is this ONE failure worth trying again?"; the circuit breaker asks "has this service been failing SO consistently that it's not even worth trying?" Applying both to the same call (as `ResilientStockClient` does) is a common, sensible combination.

Rate Limiter and Bulkhead: Two More Guards

A circuit breaker and retry both react to FAILURES. A rate limiter and a bulkhead guard against a different risk entirely: order-service overwhelming inventory-service (or itself) even when everything is HEALTHY. A rate limiter caps how many calls are allowed in a time window; a bulkhead caps how many calls can be IN FLIGHT at the same time -- both borrow their names from real-world safety mechanisms (an electrical rate limiter, a ship's bulkhead compartments preventing one flooded section from sinking the whole vessel).

```
# A SEPARATE Resilience4j config block from Resilience4jConfig.yml -- rate limiter
# and bulkhead protect against two DIFFERENT problems than a circuit breaker does
# (see "Rate Limiter and Bulkhead: Two More Guards"), so they get their own
# instance names here, but the SAME "inventoryService" name could reuse settings
# across all four guard types if a single call needed every one of them at once.

resilience4j:
  ratelimiter:
    instances:
      inventoryService:
        limit-for-period: 20           # at most 20 calls...
        limit-refresh-period: 1s       # ...per 1-second window
        timeout-duration: 0            # don't wait for a free slot -- reject
                                        # immediately if the limit is already hit
  bulkhead:
    instances:
      inventoryService:
        max-concurrent-calls: 5        # at most 5 calls to inventory-service IN
                                        # FLIGHT at once, from THIS service
        max-wait-duration: 0           # reject immediately if all 5 slots are busy,
                                        # don't queue
```

```
import io.github.resilience4j.circuitbreaker.CircuitBreaker;
import io.github.resilience4j.circuitbreaker.CircuitBreakerRegistry;
import jakarta.annotation.PostConstruct;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

// The circuit breaker's STATE (see "Circuit Breaker: States and Configuration") isn't
// directly visible anywhere in ResilientStockClient -- @CircuitBreaker manages it behind
// the scenes. This class subscribes to the SAME "inventoryService" instance's state
// transitions, purely to make CLOSED -> OPEN -> HALF_OPEN -> CLOSED visible in the logs
// -- useful while learning, and a real precursor to the metrics/dashboards the
// Observability lesson covers later.
@Component
class CircuitBreakerEventListener {

    private static final Logger log = LoggerFactory.getLogger(CircuitBreakerEventListener.
class);

    private final CircuitBreakerRegistry circuitBreakerRegistry;

    CircuitBreakerEventListener(CircuitBreakerRegistry circuitBreakerRegistry) {
        this.circuitBreakerRegistry = circuitBreakerRegistry;
    }

    @PostConstruct
    void subscribeToStateTransitions() {
        CircuitBreaker inventoryServiceBreaker = circuitBreakerRegistry.circuitBreaker
("inventoryService");
        inventoryServiceBreaker.getEventPublisher()
                .onStateTransition(event ->
                        log.warn("inventoryService circuit breaker: {} -> {}",
                                event.getStateTransition().getFromState(),
                                event.getStateTransition().getToState()));
    }
}
```

Best Practices

- **Apply `@Retry` and `@CircuitBreaker` TOGETHER on calls that can genuinely fail transiently** -- retry handles the brief blip, the circuit breaker handles sustained failure, and each answers a question the other doesn't.
- **Always provide a fallback that makes sense for the CALLER**, not just a generic error -- `checkStockFallback`'s "assume no stock reserved" lets order-service's larger flow continue, rather than failing outright.
- **Give circuit breaker/retry instances names that map cleanly to what they protect** -- `inventoryService` here, not a generic name shared across unrelated calls.
- **Log state transitions (or expose them as metrics)** during development, exactly like `CircuitBreakerEventListener` -- an OPEN circuit that fails silently is hard to diagnose.

Common Mistakes

- **Calling an `@CircuitBreaker` / `@Retry`-annotated method from another method in the SAME class.** This bypasses Spring's proxy entirely -- neither annotation has any effect (see the warning above).
- **Writing a fallback method with a mismatched signature.** It must accept the same parameters as the original method plus a trailing `Throwable`, or Resilience4j can't wire it up.
- **Setting `wait-duration-in-open-state` far too short.** The circuit reopens the test call to a service that likely hasn't recovered yet, defeating the point of giving it breathing room.
- **Using a bulkhead or rate limiter as a substitute for a circuit breaker.** They protect against DIFFERENT risks (overload vs. sustained failure) -- a service that's genuinely down still needs a circuit breaker, no amount of concurrency limiting fixes that.

Summary, Cheat Sheet, and Glossary

Resilience4j wraps a risky call with protective behaviors applied through annotations: `@CircuitBreaker` stops calling a service that's failing consistently (CLOSED -> OPEN -> HALF_OPEN -> CLOSED), `@Retry` tries a transiently-failed call again before giving up, and rate limiters/bulkheads guard against overload even when a service is healthy. A fallback method runs instead of the real one whenever the circuit is open or retries are exhausted -- its signature must match plus a trailing `Throwable`. All of this only works through Spring's proxy mechanism, so self-invocation within the same class bypasses it entirely.

Quick reference:

```
@CircuitBreaker(name = "inventoryService", fallbackMethod = "checkStockFallback")
@Retry(name = "inventoryService")
StockCheckResponse checkStock(String productName) { ... }

StockCheckResponse checkStockFallback(String productName, Throwable t) {
    return new StockCheckResponse(productName, 0);   // degraded but valid response
}

// application.yml
// resilience4j.circuitbreaker.instances.inventoryService.failure-rate-threshold: 50
// resilience4j.retry.instances.inventoryService.max-attempts: 3
```

Glossary

Circuit Breaker — A guard that stops calling a consistently-failing service, cycling through CLOSED, OPEN, and HALF_OPEN states.

Retry — A guard that automatically tries a failed call again a configured number of times before giving up.

Rate Limiter — A guard that caps how many calls are allowed within a time window, regardless of success or failure.

Bulkhead — A guard that caps how many calls to a dependency can be in flight at the same time.

Fallback Method — The method Resilience4j calls instead of the real one, whenever a circuit is open or retries are exhausted.