Resiliency

Friday, September 6, 2019 2:08 PM

Client Resiliency

if time out occurs,
application will fail fast and prevent resource exhaustion issues
application will fail gracefully, by getting data from alternate source
circuit breaker can periodically check to see if the resource being requested is back on line and re-enable access to it without human intervention.

Service Consumer - Calling Microservice.

1. Client-side load balancing
2. Circuit breakers
3. Fallbacks
4. Bulkheads

These patterns are implemented in the client calling the remote resource. The implementation of these patterns logically sit between the client consuming the remote resources and the resource itself.

With a software circuit breaker, when a remote service is called, the circuit breaker will monitor the call. If the calls take too long, the circuit breaker will intercede and kill the call. In addition, the circuit breaker will monitor all calls to a remote resource and if enough calls fail, the circuit break implementation will pop, failing fast and preventing future calls to the failing remote resource.

5.1.3 *Fallback processing*

With the fallback pattern, when a remote service call fails, rather than generating an exception, the service consumer will execute an alternative code path and try to carry out an action through another means. This usually involves looking for data from

5.1.4 Bulkheads

The bulkhead pattern is based on a concept from building ships. With a bulkhead design, a ship is divided into completely segregated and watertight compartments called bulkheads. Even if the ship's hull is punctured, because the ship is divided into

---

watertight compartments (bulkheads), the bulkhead will keep the water confined to the area of the ship where the puncture occurred and prevent the entire ship from filling with water and sinking.

Finally, the circuit breaker will occasionally let calls through to a degraded service, and if those calls succeed enough times in a row, the circuit breaker will reset itself.

The key thing a circuit break patterns offers is the ability for remote calls to

1. *Fail fast*—When a remote service is experiencing a degradation, the application will fail fast and prevent resource exhaustion issues that normally shut down the entire application. In most outage situations, it's better to be partially down rather than completely down.
2. *Fail gracefully*—By timing out and failing fast, the circuit breaker pattern gives the application developer the ability to fail gracefully or seek alternative mechanisms to carry out the user's intent. For instance, if a user is trying to retrieve data from one data source, and that data source is experiencing a service degradation, then the application developer could try to retrieve that data from another location.
3. *Recover seamlessly*—With the circuit-breaker pattern acting as an intermediary, the circuit breaker can periodically check to see if the resource being requested is back on line and re-enable access to it without human intervention.

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
<dependency>
  <groupId>com.netflix.hystrix</groupId>
  <artifactId>hystrix-javanica</artifactId>
  <version>1.5.9</version>
</dependency>
```

Listing 5.1 The `@EnableCircuitBreaker` annotation used to activate Hystrix in a service

```
package com.thoughtmechanix.licenses

import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;
//Rest of imports removed for conciseness

@SpringBootApplication
@EnableEurekaClient
@EnableCircuitBreaker                    <-- Tells Spring Cloud you're going to use Hystrix for your service
public class Application {
    @LoadBalanced
    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Listing 5.2 Wrappering a remote resource call with a circuit breaker

```
//Imports removed for conciseness
@HystrixCommand                          <-- @HystrixCommand annotation is used to wrapper the getLicenseByOrg() method with a Hystrix circuit breaker.
public List<License> getLicensesByOrg(String organizationId){
    return licenseRepository.findByOrganizationId(organizationId);
  }
```

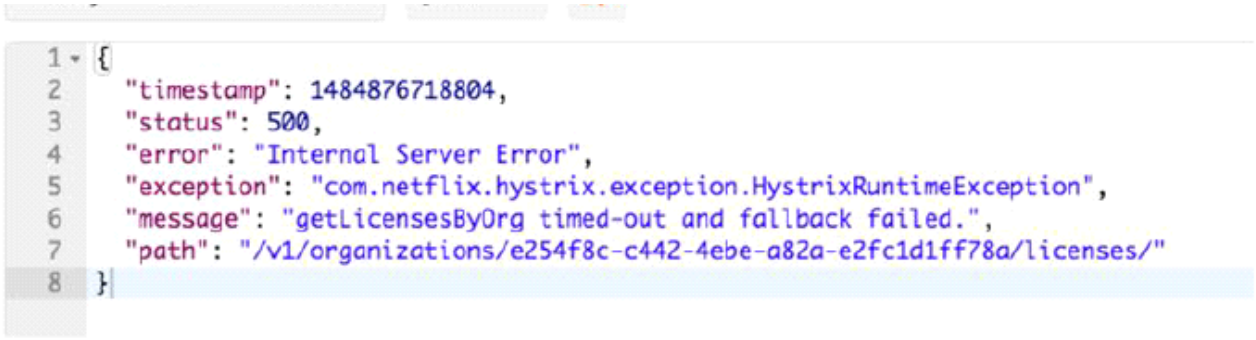

Figure 5.5 A HystrixRuntimeException is thrown when a remote call takes too long.

Now, with `@HystrixCommand` annotation in place, the licensing service will interrupt a call out to its database if the query takes too long. If the database calls take longer than 1,000 milliseconds to execute the Hystrix code wrapping, your service call will throw a `com.nextflix.hystrix.exception.HystrixRuntimeException` exception.

the amount of time Hystrix waits before timing out a call.

Listing 5.4 Customizing the time out on a circuit breaker call

```
@HystrixCommand(
  commandProperties=
     {@HystrixProperty(
   name="execution.isolation.thread.timeoutInMilliseconds",
   value="12000")})
public List<License> getLicensesByOrg(String organizationId){
  randomlyRunLong();

  return licenseRepository.findByOrganizationId(organizationId);
}
```

The commandProperties attribute lets you provide additional properties to customize Hystrix.

The execution.isolation.thread.timeoutInMilliseconds is used to set the length of the timeout (in milliseconds) of the circuit breaker.

@HystrixCommand(
commandProperties=
{@HystrixProperty(
name="execution.isolation.thread.timeoutInMilliseconds",
value="12000")})

Listing 5.5 Implementing a fallback in Hystrix

The fallbackMethod attribute defines a single function in your class that will be called if the call from Hystrix fails.

```
@HystrixCommand(fallbackMethod = "buildFallbackLicenseList")
  public List<License> getLicensesByOrg(String organizationId){
    randomlyRunLong();

    return licenseRepository.findByOrganizationId(organizationId);
  }

private List<License> buildFallbackLicenseList(String organizationId){
  List<License> fallbackList = new ArrayList<>();
  License license = new License()
    .withId("0000000-00-00000")
    .withOrganizationId( organizationId )
    .withProductName(
    "Sorry no licensing information currently available");
```

In the fallback method you return a hard-coded value.

tected by the `@HystrixCommand`. The fallback method must have the exact same method signature as the originating function as all of the parameters passed into the original method protected by the `@HystrixCommand` will be passed to the fallback.

asynchronous messages to communicate between applications isn't new. What's new is the concept of using messages to communicate events representing changes in state. This concept is called Event Driven Architecture (EDA). It's also known as Message Driven Architecture (MDA). What an EDA-based approach allows you to do is to build highly decoupled systems that can react to changes without being tightly coupled to

1 Fallbacks are a mechanism to provide a course of action when a resource has timed out or failed. If you find yourself using fallbacks to catch a timeout

exception and then doing nothing more than logging the error, then you should probably use a standard `try.. catch` block around your service invocation, catch the `HystrixRuntimeException`, and put the logging logic in the `try..catch` block.

BulkHeadPattern

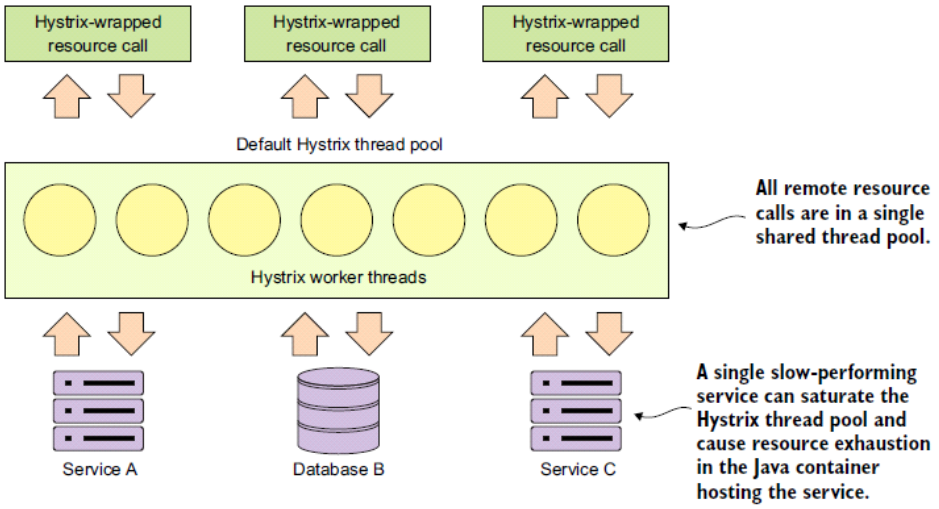

Figure 5.7 Default Hystrix thread pool shared across multiple resource types

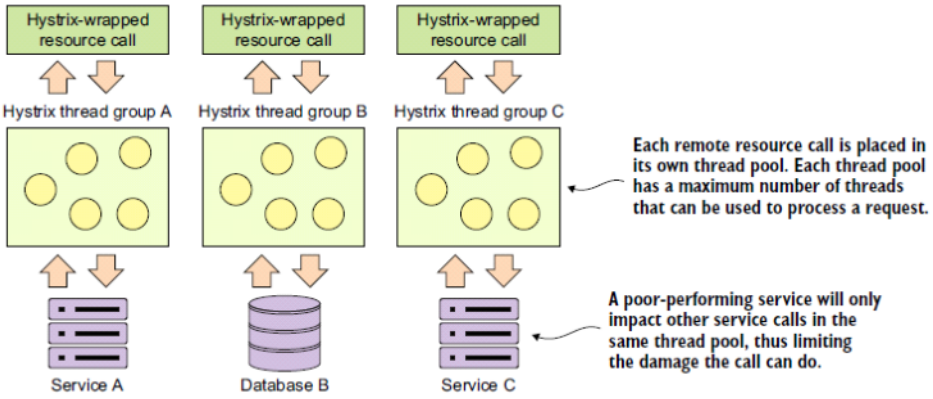

Figure 5.8 Hystrix command tied to segregated thread pools

Listing 5.6 Creating a bulkhead around the `getLicensesByOrg()` method

The threadPoolProperties attribute lets you define and customize the behavior of the threadPool.

The threadPoolKey attribute defines the unique name of thread pool.

```
@HystrixCommand(fallbackMethod = "buildFallbackLicenseList",
      threadPoolKey = "licenseByOrgThreadPool",
      threadPoolProperties =
          {@HystrixProperty(name = "coreSize",value="30"),
          @HystrixProperty(name="maxQueueSize", value="10")}
  )

public List<License> getLicensesByOrg(String organizationId){
  return licenseRepository.findByOrganizationId(organizationId);
)
```

The coreSize attribute lets you define the maximum number of threads in the thread pool.

The maxQueueSize lets you define a queue that sits in front of your thread pool and that can queue incoming requests.

@HystrixCommand(fallbackMethod = "buildFallbackLicenseList",
threadPoolKey = "licenseByOrgThreadPool",
threadPoolProperties =
{@HystrixProperty(name = "coreSize",value="30"),

@HystrixProperty(name="maxQueueSize", value="10")}

5.8 Getting beyond the basics; fine-tuning Hystrix

At this point we've looked at the basic concepts of setting up a circuit breaker and bulkhead pattern using Hystrix. We're now going to go through and see how to really customize the behavior of the Hystrix's circuit breaker. Remember, Hystrix does more than time out long-running calls. Hystrix will also monitor the number of times a call fails and if enough calls fail, Hystrix will automatically prevent future calls from reaching the service by failing the call before the requests ever hit the remote resource.

There are two reasons for this. First, if a remote resource is having performance problems, failing fast will prevent the calling application from having to wait for a call to time out. This significantly reduces the risk that the calling application or service will experience its own resource exhaustion problems and crashes. Second, failing fast and preventing calls from service clients will help a struggling service keep up with its load and not crash completely under the load. Failing fast gives the system experiencing performance degradation time to recover

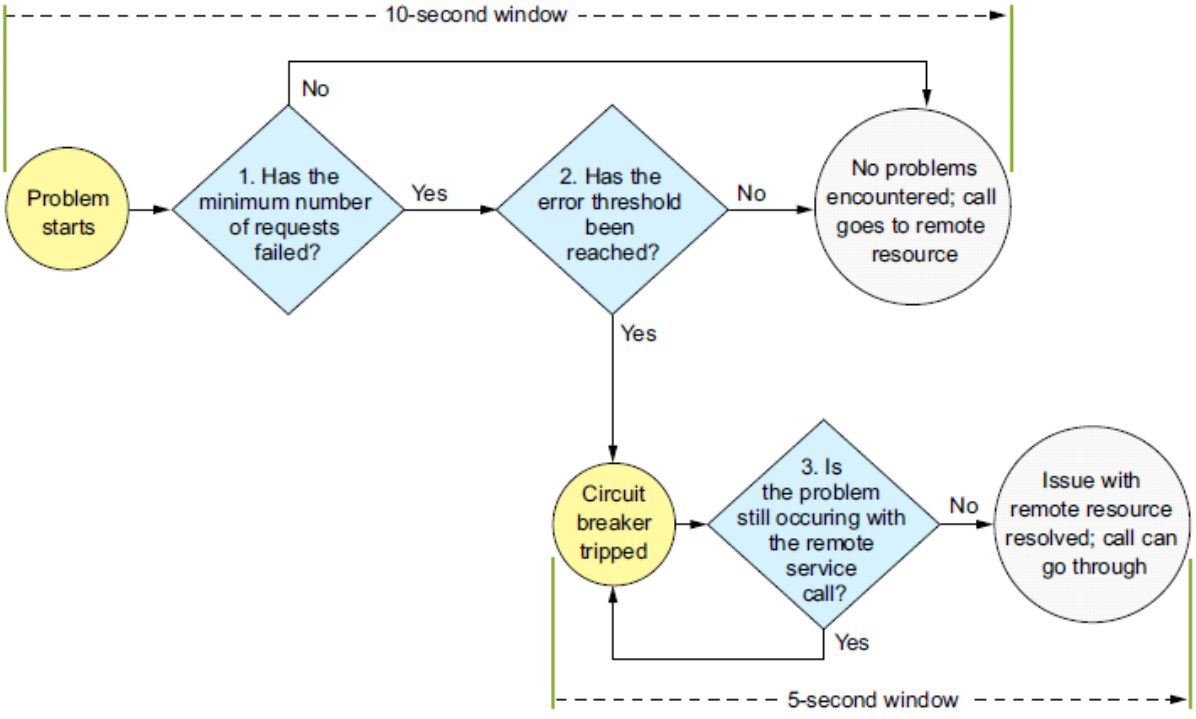

Figure 5.9 Hystrix goes through a series of checks to determine whether or not to trip the circuit breaker.

When Hystrix has "tripped" the circuit breaker on a remote call, it will try to start a new window of activity. Every five seconds (this value is configurable), Hystrix will let a call through to the struggling service. If the call succeeds, Hystrix will reset the circuit breaker and start letting calls through again. If the call fails, Hystrix will keep the circuit breaker closed and try again in another five seconds.

Listing 5.7 Configuring the behavior of a circuit breaker

```
@HystrixCommand(
   fallbackMethod = "buildFallbackLicenseList",
   threadPoolKey = "licenseByOrgThreadPool",
   threadPoolProperties ={
            @HystrixProperty(name = "coreSize",value="30"),
        @HystrixProperty(name="maxQueueSize"value="10"),
   },
   commandPoolProperties ={
     @HystrixProperty(
      name="circuitBreaker.requestVolumeThreshold",
      ➥ value="10"),
      ➥ @HystrixProperty(
      name="circuitBreaker.errorThresholdPercentage",
      ➥ value="75"),

         @HystrixProperty(
           name="circuitBreaker.sleepWindowInMilliseconds",
           value="7000"),
           ➥ @HystrixProperty(
```



```
           name="metrics.rollingStats.timeInMilliseconds",
           ➥ value="15000")
         @HystrixProperty(
           name="metrics.rollingStats.numBuckets",
           ➥ value="5")}
    )
  public List<License> getLicensesByOrg(String organizationId){
  ➥ logger.debug("getLicensesByOrg Correlation id: {}",
  ➥ UserContextHolder
              .getContext()
              .getCorrelationId());
    randomlyRunLong();

    return licenseRepository.findByOrganizationId(organizationId);
  }
```

The first property, `circuitBreaker.requestVolumeTheshold`, controls the amount of consecutive calls that must occur within a 10-second window before Hystrix will consider tripping the circuit breaker for the call. The second property, `circuitBreaker.errorThresholdPercentage`, is the percentage of calls that must fail (due to timeouts, an exception being thrown, or a HTTP 500 being returned) after the `circuitBreaker.requestVolumeThreshold` value has been passed before the circuit breaker it tripped. The last property in the previous code example, `circuitBreaker.sleepWindowInMilliseconds`, is the amount of time Hystrix will sleep once the circuit breaker is tripped before Hystrix will allow another call through to see if the service is healthy again.

are set via a class-level annotation called `@DefaultProperties`. For example, if you wanted all the resources within a specific class to have a timeout of 10 seconds, you could set the `@DefaultProperties` in the following manner:

```
@DefaultProperties(
  commandProperties = {
   @HystrixProperty(
   ➥ name = "execution.isolation.thread.timeoutInMilliseconds",
   value = "10000")}
class MyService { ... }
```