MAINT.

Protegiendo tu aplicación de fallos externos con el patrón Circuit Breaker

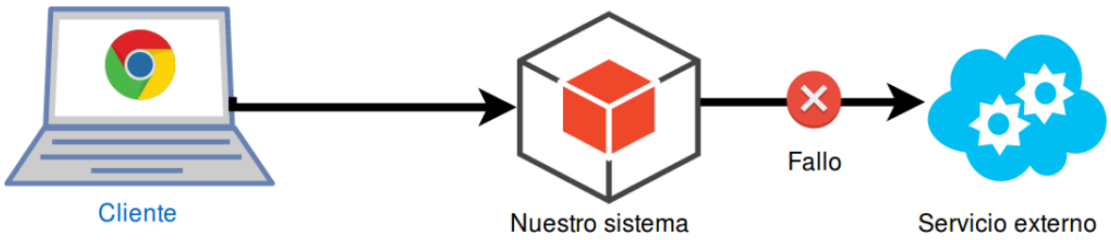
Contenido

1. El problema
2. La solución
3. Un poco de historia…
4. Y como funciona un Circuit Breaker en software?
5. El Escenario
6. La Dependencia Externa
7. La Aplicación Principal
8. Bonus: Dashboard de Hystrix
9. Conclusión

El patrón de estabilidad *Circuit Breaker*, ya muy conocido en el mundo de los microservicios pero útil en cualquier tipo de aplicación que tenga dependencias con sistemas externos (y cual no?). Veremos también su aplicación práctica con Java, utilizando Netflix Hystrix con Spring Boot.

El problema

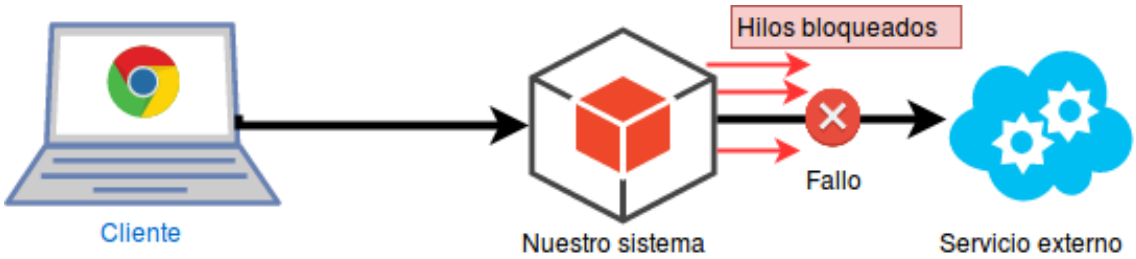
Con muy pocas excepciones, nuestras aplicaciones siempre van a depender de un sistema externo, generalmente una base de datos u otros servicios y esto es especialmente cierto en una arquitectura SOA o una de microservicios en donde ensamblamos nuevos productos y servicios partiendo de la composición de otros. Estos sistemas externos pueden ser hechos por nuestra empresa o simplemente servicios web de otras compañías.



Cuando uno de estos servicios externos se pone lento o falla completamente, nuestra aplicación puede pasar muchos segundos esperando su respuesta hasta que un timeout en la conexión nos lleva a un estado de fallo (ojalá controlado, por lo menos con un try/catch). Esto es una característica del protocolo TCP.

El problema radica en que cada vez que esto sucede, un hilo de nuestra aplicación estará bloqueado esperando la respuesta, ocupando el CPU esperando por nada, sólo para fallar al final de la espera. Durante este tiempo que nuestro hilo está bloqueado, el CPU se desperdicia pues no puede atender otras peticiones y si nuestro sistema está bajo una carga alta el problema se maximiza: cada petición nueva que consulta al sistema externo debe esperar a que las peticiones que llegaron primero se rompan por un timeout. La configuración por defecto usual de los timeout TCP es de 30 segundos.

Imagina que llega una petición por la que debes consultar al sistema externo. Esta petición queda esperando la respuesta del sistema externo por 30 segundos durante los cuales llegan nuevas peticiones a una tasa, por ejemplo, de 5 peticiones por segundo. Para cuando la primera petición se rompa por timeout, habrá 150 peticiones en cola esperando también a romper por timeout!. Este efecto es exponencial y terminará agotando los hilos de tu aplicación hasta que ésta termine fallando también y causando una caída en cascada hacia tus clientes, bien sea en forma de respuesta muy lenta, un error 500 o ninguna respuesta.



La solución

En estos casos lo mejor que puedes hacer es hacer que tu servicio se "*degrade elegantemente*" o gracefully degradation, como se conoce en inglés a la forma en que tu aplicación sigue respondiendo ante un fallo, quizá con menos prestaciones o sin ninguna prestación, **pero aún respondiendo** y no con incomprensibles trazas de error sino con mensajes significativos para el usuario. En otros casos puede que sea posible atender la petición con información guardada localmente en lugar de consultar al servicio externo o simplemente enviar la información de la petición a una cola para su procesamiento posterior. Lo importante es no volcar una traza de error (o una pantalla azul de la muerte…) frente al usuario.

Uno de los patrones de estabilidad bien conocidos y utilizados en estos casos es *Circuit Breaker*, que ha revitalizado su popularidad gracias al auge de los microservicios.

Un poco de historia…

Citando el ejemplo ilustrado en *Release it*!, en las primeras instalaciones eléctricas en los hogares, la gente acostumbraba a conectar muchos aparatos eléctricos al circuito de su casa. Cada aparato extraía cierta cantidad de corriente. La corriente al presentar resistencia (la del aparato) produce calor. Algunas veces el calor era suficiente para producir fuego entre las paredes e incendiar las casas.

Para prevenirlo, la gente empezó a utilizar fusibles: dispositivos similares a bombillas, conectados al circuito que cuando experimentaban corrientes altas, se quemaban dejando abierto el circuito e interrumpiendo la corriente. Su problema es que eran costosos y desechables por lo que muchas personas empezaron a crear sus propias versiones de fusibles y… los incendios regresaron.

Finalmente y desde entonces utilizamos los *circuit breakers* que detectan una corriente alta y cambian automáticamente a una posición en la que abren el circuito, interrumpiendo el

paso de la corriente (*open*). Luego cuando la situación vuelve a la normalidad, manualmente los cambiamos de posición para volver a cerrar el circuito y permitir nuevamente el paso de la corriente (*close*).

Panel de Circuit Breakers

Y como funciona un Circuit Breaker en software?

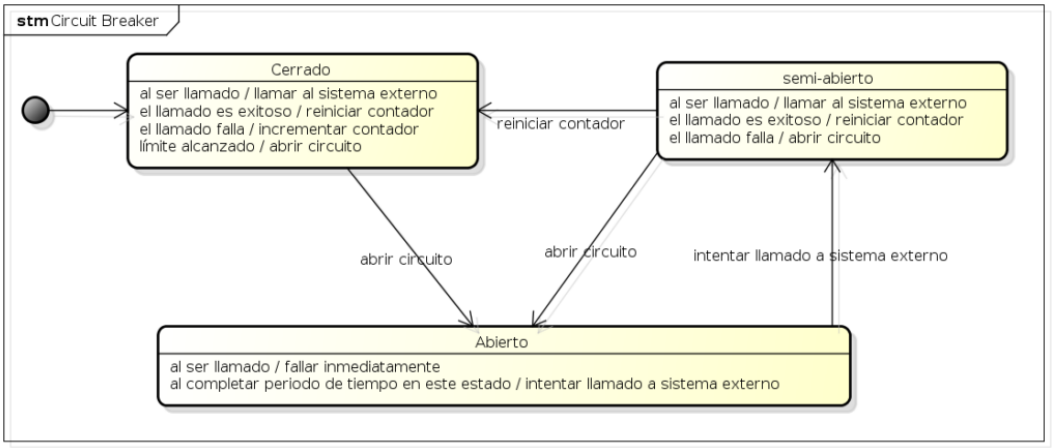
La idea es la misma: aislar nuestro dispositivo (aplicación) cuando se detecte un fallo en el circuito (conexión con una dependencia) y restablecer la conexión cuando el fallo sea solucionado.

La implementación del patrón es así:

1. En el estado normal o *cerrado* (*close*), el circuit breaker ejecuta operaciones como de costumbre. Estas operaciones pueden ser llamados a sistemas externos o pueden ser operaciones internas sujetas a un timeout u otra condición de error. Si la ejecución es exitosa, nada ocurre.
2. Si el llamado al sistema externo falla, el Circuit Breaker incrementa su conteo interno de fallos
3. Cuando el número de fallos supera un límite, el circuit breaker pasa al estado *open* y abre el circuito, interrumpiendo el flujo de la aplicación. esto significa que cualquier invocación al sistema externo fallara inmediatamente sin siquiera intentar el llamado. Aquí es recomendable fallar con una excepción diferente que indique que el breaker

está abierto y así mismo reaccionar de otro modo indicando al usuario y operador lo que sucede para que puedan tomar las acciones adecuadas.

4. Después de cierto periodo de tiempo, el breaker decide que la operación podría tener éxito y pasa al estado *semi-abierto* o *half-open*. En este estado el breaker no fallará inmediatamente sino que intentará el llamado al sistema externo la próxima vez que lo invoquen. Si el llamado es exitoso, pasará al estado *close* permitiendo los llamados posteriores. Si el llamado falla nuevamente, se quedará en el estado *open* hasta que vuelva a transcurrir otra vez el periodo de tiempo y pase a *half-open* nuevamente.



La características principal de un Circuit Breaker es que sirve para impedir la operación externa en lugar de reintentarla. Esto puede traer un impacto sobre la operación de negocio que se está protegiendo, por lo cual es muy importante involucrar a los responsables de negocio para tomar las decisiones sobre como actuará nuestra aplicación en estos casos.

El Escenario

Vamos a implementar una aplicación web muy simple que actuará como nuestro sistema, en el proyecto `my-app`. Esta aplicación mostrará una lista de posts de un blog, consultando a una dependencia o servicio externo que también construiremos en el proyecto `my-external-dependency`, de esta manera podremos simular un fallo en el servicio externo deteniendo el programa correspondiente.

Para efectos de la demostración, ambos proyectos se mantendrán muy simples, con la menor cantidad de código posible, por eso **NO** vamos a respetar el principio de separación de responsabilidades y veremos mezcladas algunas **capas que en una aplicación real deben estar separadas**, por ejemplo, controladores y servicios con lógica de negocio.

La Dependencia Externa

Para este proyecto crearemos un nuevo proyecto Maven y agregamos las dependencias de Spring Boot Web y la referencia al POM padre. Para crear el proyecto, la forma más fácil y rápida es utilizar la versión de Eclipse para Spring llamada “Spring Tool Suite” y su opción de menú File/New/Spring Starter Project. Otra opción igualmente fácil es utilizar el generador Spring Initializr. En cualquier caso, el POM debe quedar similar al siguiente:

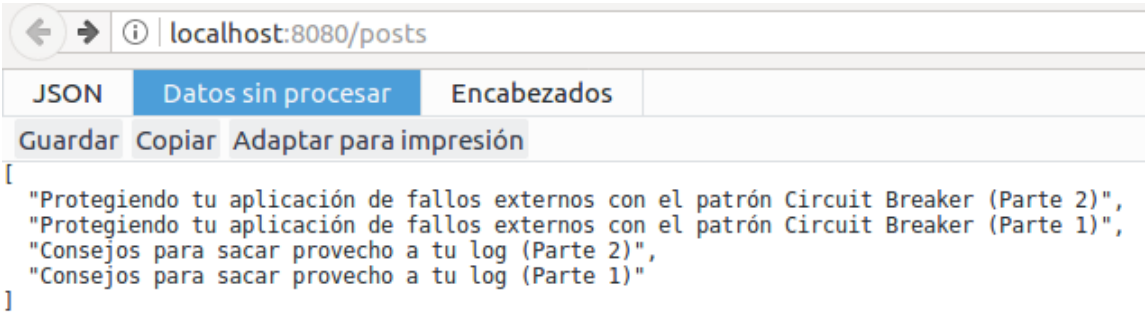
```
1  <parent>
2    <groupId>org.springframework.boot</groupId>
3    <artifactId>spring-boot-starter-parent</artifactId>
4    <version>1.5.4.RELEASE</version>
5    <relativePath/>
6  </parent>
7
8  <dependencies>
9    <dependency>
10    <groupId>org.springframework.boot</groupId>
11    <artifactId>spring-boot-starter-web</artifactId>
12  </dependency>
13<dependencies>
```

Creamos una clase con nuestro controlador de un servicio REST y el código necesario para inicializar la aplicación como una aplicación de Spring Boot:

```
1  @SpringBootApplication
2  @Controller
3  public class MyExternalDependencyApplication {
4
5     public static void main(String[] args) {
6        SpringApplication.run(MyExternalDependencyApplication.class, args);
7     }
8
```

```
9     @RequestMapping("/posts")
10    @ResponseBody
11    public String[] getPosts(){
12        return new String[]{
13            "Protegiendo tu aplicación de fallos externos con el patrón Circuit Breaker (Parte
142)",
15            "Protegiendo tu aplicación de fallos externos con el patrón Circuit Breaker (Parte
161)",
17            "Consejos para sacar provecho a tu log (Parte 2)",
18            "Consejos para sacar provecho a tu log (Parte 1)",
19        };
      }
   }
```

Eso es todo para la dependencia externa. Al ejecutarlo podemos acceder en un navegador a la URL http://localhost:8080/posts para recibir la lista de posts que genera el servicio:



La Aplicación Principal

Esta será la aplicación a la que nuestro cliente accede y que invoca un llamado en el servicio externo que ya construimos. Para construir esta aplicación, debemos crear un proyecto Maven y le agregamos las mismas dependencias de nuestro servicio externo, agregando además Hystrix y Thymeleaf que es un procesador de plantillas HTML, así:

```
1  ...
2  <dependency>
3     <groupId>org.springframework.cloud</groupId>
4     <artifactId>spring-cloud-starter-hystrix</artifactId>
5  </dependency>
6  <dependency>
7     <groupId>org.springframework.boot</groupId>
8     <artifactId>spring-boot-starter-web</artifactId>
```

```
9   </dependency>
10<dependency>
11    <groupId>org.springframework.boot</groupId>
12    <artifactId>spring-boot-starter-thymeleaf</artifactId>
13</dependency>
14...
```

En esta versión de Spring Boot adicionalmente debemos incluir el POM padre de Spring Cloud y la versión de Spring Cloud:

```
1  <properties>
2     <spring-cloud.version>Dalston.SR1</spring-cloud.version>
3  </properties>
4  ...
5  <dependencyManagement>
6     <dependencies>
7        <dependency>
8           <groupId>org.springframework.cloud</groupId>
9           <artifactId>spring-cloud-dependencies</artifactId>
10          <version>${spring-cloud.version}</version>
11          <type>pom</type>
12          <scope>import</scope>
13       </dependency>
14    </dependencies>
15</dependencyManagement>
```

Para evitar conflictos con el puerto del servicio externo, debemos cambiar el puerto de nuestra aplicación. Para hacerlo, agrega la propiedad `server.port` en el archivo `application.properties` con un número de puerto diferente al 8080, así:

```
server.port=9090
```

Creamos un controlador y una vista HTML (plantilla de Thymeleaf) para mostrar la lista de posts. En un futuro post trataremos Thymeleaf con más detalle pues está fuera del alcance de este post:

```
1  @Controller
2  public class ListingController {
3
4     @Autowired
5     private PostsService postsService;
6
7     @RequestMapping("/show-posts")
8     public String showPosts(Model model){
```

```
9        model.addAttribute("posts", postsService.getPosts());
10       return "posts-view";
11   }
12}
```

```
1  <!DOCTYPE html>
2  <html xmlns:th="http://www.thymeleaf.org">
3     <head>
4        <title>Ejemplo de uso de Hystrix</title>
5     </head>
6     <body>
7
8  <h2>Lista de Posts</h2>
9
10
11<ul>
12
13<li th:each="p : ${posts}" th:text="${p}"></li>
14
15       </ul>
16
17   </body>
18</html>
```

Finalmente creamos nuestra clase de servicio que será invocada por el controlador y esta a su vez, invocará al servicio externo utilizando Hystrix. Esta es la clase que debes observar con detenimiento y es el punto central de este post:

```
1  @Service
2  public class PostsService {
3     @HystrixCommand(fallbackMethod="defaultPosts")
4     public String[] getPosts() {
5        return new RestTemplate().getForObject("http://localhost:8080/posts",
6  String[].class);
7     }
8
9     public String[] defaultPosts(){
10       return new String[]{"Post #1", "Post #2"};
11    }
  }
```

Vamos a analizar las líneas más importantes:

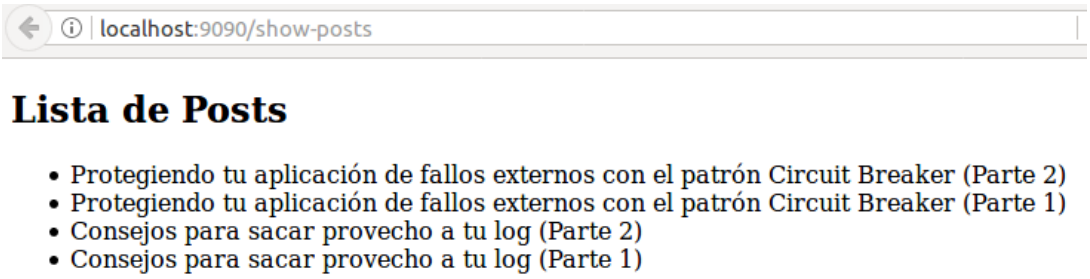
- **Línea 1**: por ahora el circuit breaker de Hystrix sólo funciona con clases @Service o @Component

- **Línea 3**: La anotación `@HystrixCommand` declara un circuit breaker para el método anotado. Opcionalmente, utilizando el parámetro `fallbackMethod`podemos declarar un método que será invocado como contingencia cuando falle el método anotado. En este caso nuestro método de contigencia es defaultPosts.
- **Línea 5**: Es la invocación al servicio externo, aquí es donde se puede producir el fallo
- **Línea 8**: Es el método que se invocará si falla el llamado al servicio externo

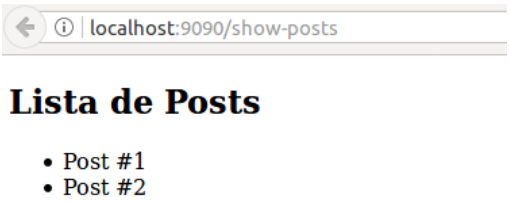
Finalmente debemos activar Hystrix en nuestra aplicación agrando la anotación `@EnableCircuitBreaker` en una clase de configuración:

```
1@SpringBootApplication
2@EnableCircuitBreaker
3public class MyAppApplication {
4
5   public static void main(String[] args) {
6      SpringApplication.run(MyAppApplication.class, args);
7   }
8}
```

Ahora iniciamos tanto el servicio externo como la aplicación y nos dirigimos a la página principal en http://localhost:9090/show-posts. La aplicación invoca al servicio externo y deberíamos ver lo siguiente:



Ahora detén el servicio externo pero deja en ejecución la aplicación y refresca la página que estás viendo, deberías recibir algo así:



Si te fijas bien, estás viendo la respuesta del método `defaultPosts` en lugar de la respuesta del servicio. Esto indica que falló el llamado al servicio externo y se ejecutó el método de contingencia como era de esperarse.

Cada vez que falle el llamado al servicio externo, se ejecutará el método de contingencia hasta que se alcance un límite (configurable) y empezarás a ver un mensaje en el log como este:

`java.lang.RuntimeException: Hystrix circuit short-circuited and is OPEN`. Si inicias nuevamente el servicio externo, pasarán 5 segundos (también configurable) desde la última falla hasta que el circuit breaker pase al estado semi-abierto y permita el llamado al servicio externo, volviendo todo a la normalidad.

Si quieres ver como configurar estos y otros valores, descarga el proyecto de nuestro repositorio y consulta guía de configuración de Hystrix.

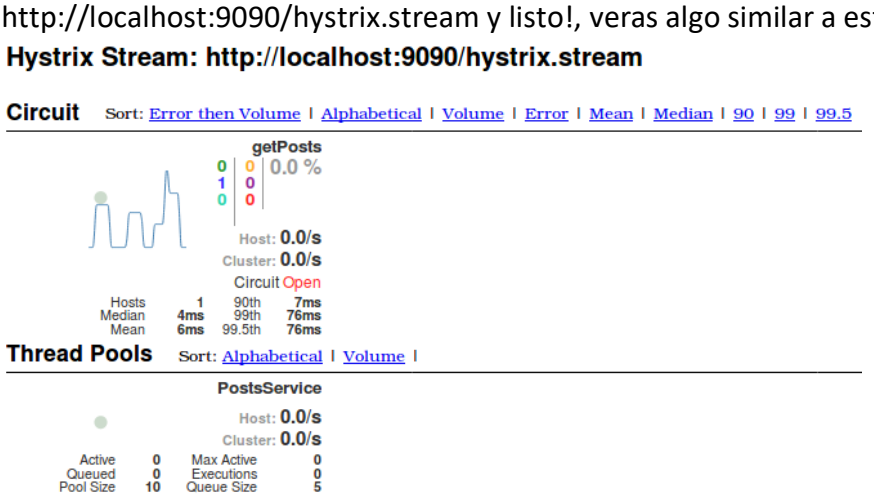
Bonus: Dashboard de Hystrix

Spring Boot nos ofrece también un sencillo dashboard para ver el estado de nuestros breakers. Para configurarlo, agrega las siguientes dependencias al POM de la aplicación principal:

```
1<dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId
4</dependency>
```

Para activarlo, agrega la anotación `@EnableHystrixDashboard` a una clase de configuración, reinicia la aplicación principal y ve a http://localhost:9090/hystrixe ingresa como valor a monitorear, el stream de histrix de nuestra aplicación que está en

http://localhost:9090/hystrix.stream y listo!, veras algo similar a esto:



Conclusión

Los puntos de integración con sistemas externos siempre tendrán la más alta probabilidad de producir un fallo en nuestra aplicación y debemos protegerla para evitar que esos fallos la arrastren consigo.

En caso de fallo de una dependencia externa, debemos procurar una degradación sútil de las funcionalidades para lo cual podemos utilizar el patrón *Circuit Breaker* mediante la librería Hystrix.

Hystrix tiene muchas más opciones interesantes que vale la pena explorar, como el dashboard, el stream de eventos para monitoreo (*Turbine Stream*) y los collapsers, que te ayudan a agrupar peticiones múltiples y similares a sistemas externo, por ejemplo cuando el usuario refresca repetidamente una página web, para así reducir el número de peticiones que se hacen a dicho sistema externo.