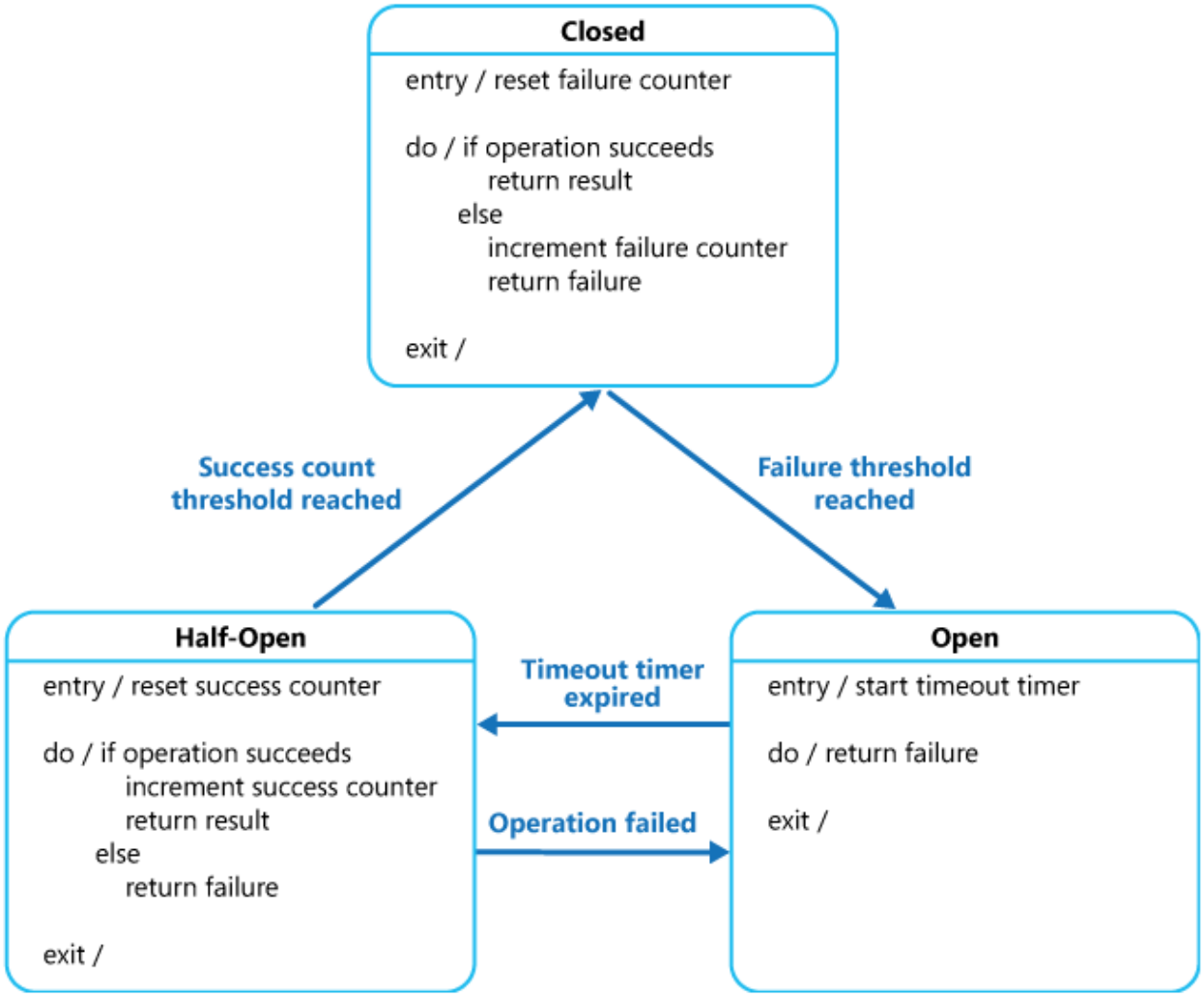
Capítulo 2. Circuit Breaker

Este patrón de diseño, se le llama el patrón de disyuntor, controla los errores cuya recuperación puede tardar una cantidad variable de tiempo durante la conexión a un recurso o servicio remoto

Puede mejorar la estabilidad y la resistencia de una aplicación



2.1. Contexto

En entornos de sistemas distribuídos, las llamadas a los servicios y los recursos remotos pueden producir un error debido a errores transitorios, como son las conexiones de red lentas, el agotamiento de los tiempos de espera o los recursos que se sobrecargan o no están disponibles temporalmente.

Estos errores suelen corregirse por sí mismos tras un breve período de tiempo, una aplicación sólida que funcione en la nube, debe estar preparada para controlarlos.

Puede haber situaciones en las que los errores se deban a eventos no anticipados y cuya corrección puede tardar mucho más tiempo.

La gravedad de estos errores puede abarcar desde una pérdida parcial de la conectividad hasta la

total detención de un servicio.

En estas situaciones, es posible que no tenga sentido para una aplicación volver a intentar contínuamente una operación que no es probable que pueda funcionar de modo correcto y, en su lugar, deba admitir este hecho con rapidez y tratar este error en consecuencia.

Por otra parte, si un servicio está muy ocupado, el error en una parte del sistema podría provocar errores en cascada

Por ejemplo, una operación que invoca un servicio, puede configurarse para implementar un tiempo de espera y responder con un mensaje de errror si el servicio no responde dentro de este período

Sin embargo, esta estrategia puede desencadenar muchas solicitudes simultáneas a la misma operación para que se bloquee hasta que expire el período de tiempo de espera

Estas solicitudes que en principio estarán bloqueadas (hasta que el acceso al servicio o recurso pueda llevarse a cabo), pueden contener recursos críticos del sistema, como memoria, subprocesos o conexiones de datos entre otros

Por lo tanto, estos recursos podrían agotarse y provocar errores de otras partes posíblemente no relacionadas del sistema que tenga que usar los mismos recursos.

Cuando se dan estas situaciones, podría ser preferible para la operación, dejar de funcionar de inmediato y sólo intentar invocar el servicio si es probable que pueda ejecutarse correctamente.

Establecer un tiempo de espera menor podría ayudar a resolver quizás este problema, pero no debería ser tan corto como para que la operación dé error en la mayoría de los casos, incluso aunque la solicitud al servicio finalmente pudiera realizarse correctamente

2.2. Solución

El patrón lo popularizó Michael Nygard en su libro "Release It!", puede impedir que una aplicación intente repetidamente ejecutar una operación que tenga probabilidad de dar error.

Ello le permite continuar sin esperar a corregir el error ni desperdiciar ciclos de CPU mientras se determina si el error continuará durante mucho tiempo.

El patrón Circuit Breaker también permite a una aplicación detectar si el error se ha resuelto.

Si el problema parece haberse corregido, la aplicación puede intentar invocar la operación.

El propósito del patrón Circuit Breaker difiere un tanto de la finalidad del patrón Retry

El patrón Retry permite a una aplicación volver a intentar una operación esperando que se podrá ejecutar correctamente

El patrón Circuit Breaker impide que una aplicación realice una operación que probablemente produzca errores

Una aplicación puede combinar estos dos patrones usando Retry para infocar una operación a través de un disyunto

Sin embargo, la lógica de reintento debe tener en cuenta las excepciones devueltas por el disyuntor y dejar de reintentar la operación si este indica que un error no es transitorio

Un disyuntor actúa como un proxy para las operaciones que podrían producir errores, el proxy debe supervisar el número de errores recientes que se han producido y utilizar esta información para decidir si permitir que continúe la operación, o simplemente devolver de inmediato una excepción

El proxy se puede implementar como una máquina de estados con los siguientes estado que imitan la funcionalidad de un disyuntor eléctrico:

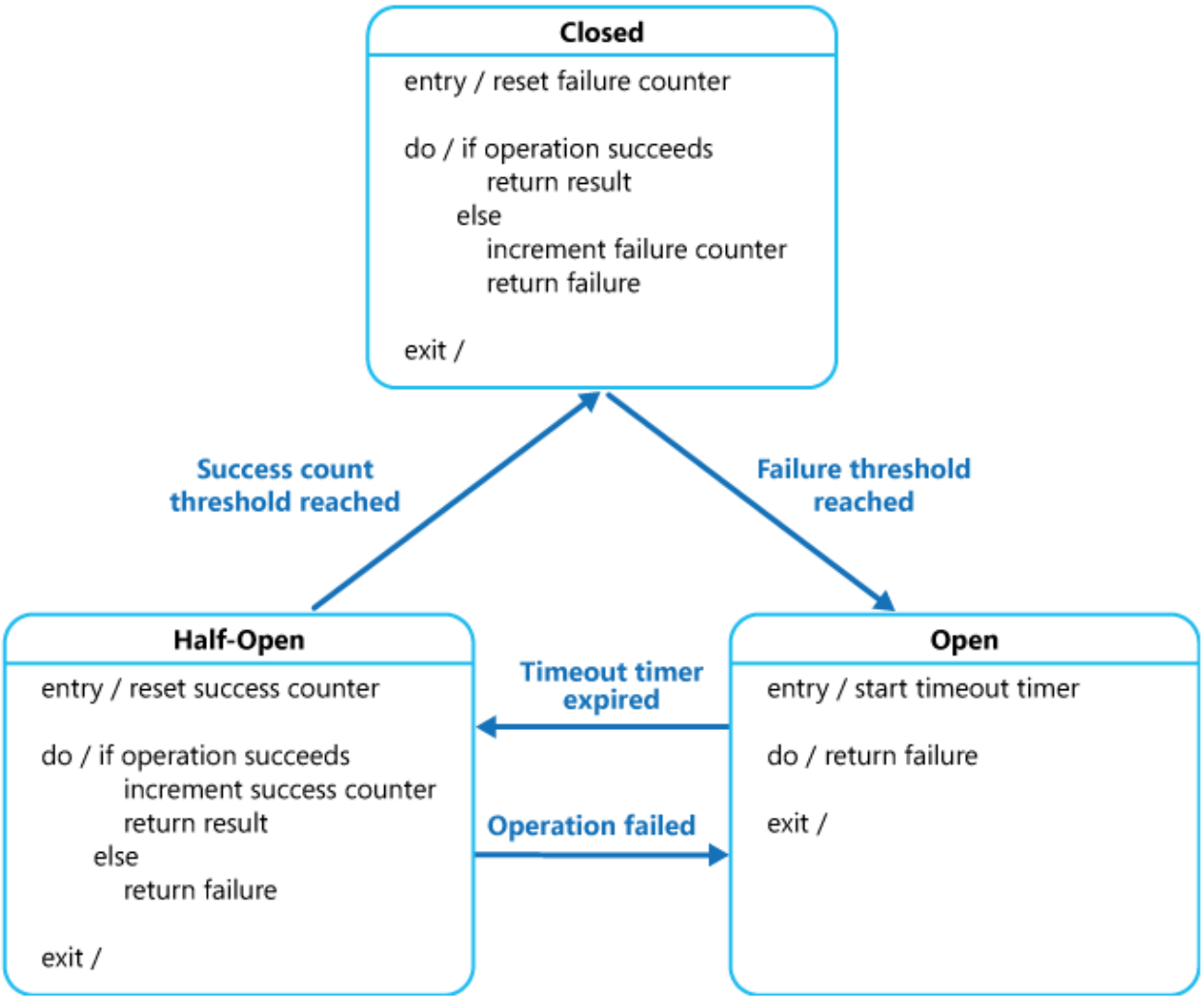
- **Closed (Cerrado):** La solicitud de la aplicación se enruta a la operación, el proxy mantiene un recuento del número de errores recientes y, si la llamada a la operación se realiza correctamente, el proxy incrementa este recuento.
  - Si el número de errores recientes supera un umbral especificado en un período de tiempo determinado, el proxy se coloca en el estado **Open (abierto)**
  - En este momento, el proxy inicia un temporizador de tiempo de espera y, cuando este temporizador expira, se coloca en el estado **Half-open (semiabierto)**

El propósito del temporizador de tiempo de espera es conceder al sistema tiempo para corregir el problema que provocó el error, antes de permitir que la aplicación intente realizar la operación de nuevo

- **Abierta:** La solicitud de la aplicación produce un error inmediatamente y se devuelve una excepción a la aplicación
- **Half-open (Semiabierto):** Se permite pasar un número limitado de solicitudes de la aplicación en invocar la operación
  - Si estas solitidues se realizan correctamente, se supone que la causa del error se ha corregido y el disyuntor cambia al estado **Closed (cerrado)** y el número de errores se restablece.
  - Si alguna solicitud da error, el disnyuntor supone que el anterior error no se solucionó y vuelve al estado **Open (abierto)**, y reinicia el temporizador de tiempo de espera para dar al sistema más tiempo para recuperarse

El estado **Half-open** (semiabierto) es útil para impedir que un servicio de recuperación se inunde de repente con solicitudes

A medida que un servicio se recupera, podría ser capaz de admitir un volumen limitado de solicitudes hasta que la recuperación se completa, pero, mientras está en curso, una saturación de trabajo puede hacer que el servicio agote el tiempo de espera o dé error de nuevo



En el diagrama, el contador de errores que usa el estado **Closed** (cerrado) depende del tiempo

Se restablece automáticamente a intervalos periódicos, esto ayuda a impedir que el disyuntor entre en el estado **Open** (abierto) si experimenta errores ocasionales

El umbral de error que se encuentra con el disnyuntor en el estado **Open** (abierto) sólo se alcanza cuando se produce un número especificado de errores durante un intervalo especificado

El contador utilizado por el estado **Half-Open** (semiabierto) registra el número de intentos correctos para invocar la operación

El disyuntor, vuelve al estado **Closed** (cerrado) después de un número especificado de llamadas consecutivas a la operación que haya tenido éxito

Si se produce un error en alguna invocación, el disyuntor entra en el estado **Open** (abierto) inmediatamente y el contador de éxitos se restablecerá la próxima vez que entre en el estado **Half-Open** (semiabierto)

Externamente, es posible que las recuperaciones del sistema se traten restaurando o reiniciando un componente erróneo o reparando una conexión de red por ejemplo

El patrón Circuit Breaker proporciona estabilidad mientras el sistema se recupera de un error y minimiza el impacto en el rendimiento

Puede ayudar a mentener el tiempo de respueta del sistema al rechazar rápidamente una solicitud para una operación que es probable que dé error, en lugar de esperar a que agote el tiempo de espera o no termine nunca.

Si el disyuntor genera un evento cada vez que cambia el estado, esta información puede utilizarse para supervisar el estado de la parte del sistema protegida por el disyuntor o para alertar a un administrador cuando un disyuntor se active en el estado **Open** (abierto).

El patrón es personalizable y puede adaptarse según el tipo de errores posibles, por ejemplo, se puede aplicar a un disyuntor un temporizador de tiempo de espera que vaya aumentado.

Se puede colocar el disyuntor en el estado **Open** (abierto) inicialmente durante unos segundos y, a continuación, incrementar el tiempo de espera unos minutos, si el error no se ha resuelto, y así sucesivamente

En algunos casos, en lugar de que el estado **Open** (abierto) devuelva un error y genera una excepción, puede ser útil devolver un valor predeterminado que sea significativo para la aplicación

2.2.1. Consideraciones a tener en cuenta

En el momento de dedicir cómo implementar este patrón, debemos de considerar los siguientes puntos

- **Control de excepciones:** Una aplicación que invoca una operación a través de un disyuntor debe estar preparada para controlar las excepciones que se produzcan si la operación no está disponible
  - La forma en que se controlan las excepciones será especificada de la aplicación
  - Por ejemplo, una aplicación podría degradar temporalmente su funcionalidad, invocar una operación alternativa para intentar realizar la misma tarea u obtener los mismos datos, o notificar la excepción al usuario y pedirle que la vuelva a intentar más tarde
- **Tipos de excepciones:** Una solicitud podría producir un error por diversos motivos, algunos de los cuales podrían indicar un tipo de error más grave que otras
  - Por ejemplo, una solicitud podría dar error por que un servicio remoto haya dejado de funcionar y tardará varios minutos en recuperarse, o por un tiempo de espera agotado debido a que el servicio está sobrecargado temporalmente
  - Un disyuntor puede examinar los tipos de excepciones que se producen y ajustar su

estrategia en función de la naturaleza de estas excepciones

  - Por ejemplo, podría ser necesario un mayor número de excepciones de tiempo de espera para activar el disyuntor en el estado **Open** (abierto) en comparación con el número de errores debidos a que el servicio no esté completamente disponible

- **Registro:** Un disyuntor debe registrar todas las solicitudes con error (y, posiblemente, las correctas) para permitir que un administrador supervise el estado de la operación
- **Capacidad de recuperación:** Se debe de configurar el disyuntor para que coincida con el modelo de recuperación más probable de la operación que se protege
  - Por ejemplo, si el disyuntor permanece en el estado **Open** (abierto) durante un largo período, podría producir excepciones aunque el motivo del error se hubiese resuelto
  - De igual forma, podría fluctuar y reducir los tiempos de respuesta de las aplicaciones si pasa del estado **Open** a **Half-Open** demasiado rápidamente.
- **Prueba de las operaciones con error:** En el estado **Open**, en lugar de usar un temporizador para determinar cuándo cambiar al estado **Half-Open**, un disyuntor puede en cambio hacer ping periódicamente al servicio remoto o al recurso para determinar si tiene que estar disponible de nuevo
  - Este ping podría adoptar la forma de intento de invocar una operación que hubiera generado el error previamente o podría usar una operación especial proporcionada por el servicio remoto de forma específica para probar el estado del servicio, como se describe en el patrón **Health Endpoint Monitoring** (supervisión del punto de conexión de estado)
- **Invalidación manual:** En un sistema en el que el tiempo de recuperación de una operación que da error es muy variable, es conveniente proporcionar una opción de restablecimiento manual que permita a un administrador cerrar un disyuntor (y restablecer el contador de errores).
  - De forma similar, un administrador podría forzar que un disyuntor pase al estado **Open** y reiniciar así el temporizador de tiempo de espera, si la operación protegida por el disyuntor no está disponible temporalmente.
- **Simultaneidad**: El mismo disyuntor podría tener acceso a un gran número de instancias simultáneas de una aplicación. La implementación no debe bloquear las solicitudes simultáneas ni agregar una sobrecarga excesiva a cada llamada a una operación.
- **Diferenciación de los recursos:** Hay que tener cuidado al usar un único disyuntor para un tipo de recurso si puede haber varios proveedores independientes subyacentes.
  - Por ejemplo, en un almacén de datos que contenga varias particiones, una podría ser totalmente accesible mientras otra experimenta un problema temporal.
  - Si se combinan las respuestas de error en estos casos, una aplicación podría intentar acceder a algunas particiones incluso cuando fuese muy probable que se produjera un error, mientras que el acceso a las otras podría bloquearse, aunque hubiera mucha probabilidad de que no sufrieran problemas
- **Disyuntor acelerado:** A veces, una respuesta de error puede contener suficiente información para que el disyuntor se active inmediatamente y permanezca así una cantidad mínima de tiempo.
  - Por ejemplo, la respuesta de error de un recurso compartido que está sobrecargado podría indicar que no se recomienda un reintento inmediato y que la aplicación, en cambio, debe

intentarse de nuevo al cabo de unos minutos

Un servicio puede devolver HTTP 429 (demasiadas solicitudes), si está limitando al cliente, o HTTP 503 (servicio no disponible), si el servicio no está disponible actualmente. La respuesta puede incluir información adicional, como la duración prevista del retraso.

- **Respuesta de las solicitudes con error:** En el estado **Open**, en lugar de generar un error rápidamente, un disyuntor también puede registrar los detalles de cada solicitud en un diario y hacer que estas solicitudes se reproduzcan cuando el servicio o el recurso remoto estén disponibles.
- **Tiempos de espera inadecuados en servicios externos:** Un disyuntor podría no ser capaz de proteger completamente las aplicaciones de las operaciones que generan errores en los servicios externos configurados con un período prolongado de tiempo de espera.
  - Si el tiempo de espera es demasiado largo, un subproceso que ejecuta un disyuntor podría bloquearse durante un largo período antes de que este indique que la operación ha fracasado.
  - En este momento, muchas otras instancias de aplicaciones también podrían intentar invocar el servicio a través del disyuntor y ocupar un número significativo de subprocesos antes de que todos den error.

2.3. Ventajas

- Permite implementar mecanismos de auto-recuperación del sistema basados en número de reintentos

2.4. ¿Cuándo resulta adecuado?

- Para evitar que una aplicación intente invocar un servicio remoto o acceda a un recurso compartido, si es muy probable que esta operación produzca un error

2.5. ¿Cuándo no resulta adecuado?

- Para llevar el control del acceso a los recursos locales privados de una aplicación, como la estructura de datos en memoria, en este tipo de entorno, el uso de un disyuntor agregaría sobrecarga al sistema
- Como mecanismo de sustitución para controlar las excepciones en la lógica de negocio de las aplicaciones

2.6. Ejemplo

Para realizar una prueba práctica del patrón de diseño, vamos a crear un módulo maven base, y a continuación crearemos dos módulos maven enganchados a el, quedando la estructura como se indica:

- circuitbreaker-parent (Módulo Java Maven base)
  - circuitbreaker-bookstore (Módulo que abre un endpoint rest api para invocar un web service)
  - circuitbreaker-reading (Módulo que implementa el patrón circuitbreaker de acceso al web service del módulo circuitbreaker-bookstore)

Vamos a utilizar en este caso, el framework de Java SpringBoot

- Estructura del módulo **circuitbreaker-parent**

Archivo pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.patterndesign.tutorial</groupId>
    <artifactId>circuitbreaker-parent</artifactId>
    <version>1.0.0</version>
    <packaging>pom</packaging>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.6.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <modules>
        <module>circuitbreaker-bookstore</module>
        <module>circuitbreaker-reading</module>
    </modules>


    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
</project>
```

- Estructura del módulo **circuitbreaker-bookstore**

Archivo pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>com.patterndesign.tutorial</groupId>
        <artifactId>circuitbreaker-parent</artifactId>
        <version>1.0.0</version>
    </parent>
    <artifactId>circuitbreaker-bookstore</artifactId>
    <packaging>jar</packaging>
    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>

      <dependencies>
        <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-test</artifactId>
          <scope>test</scope>
        </dependency>
      </dependencies>

    <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

Clase Java BookStoreApplication.java

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@SpringBootApplication
public class BookStoreApplication {

    @RequestMapping(value = "/recommended")
    public String readingList() {
        return "Spring Book Store App!!!";
    }

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

Configuramos la aplicación spring boot para que se ejecute en un puerto de escucha concreto

Archivo de configuración de Spring src/main/resources/application.properties

```
server.port=8090
```

- Estructura del módulo **circuitbreaker-reading**

Archivo pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>com.patterndesign.tutorial</groupId>
        <artifactId>circuitbreaker-parent</artifactId>
        <version>1.0.0</version>
    </parent>

    <properties>
            <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
            <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
```

```
            <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Finchley.SR2</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

    <artifactId>circuitbreaker-reading</artifactId>
    <packaging>jar</packaging>
</project>
```

Clase Java BookService.java

```
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

import java.net.URI;

@Service
public class BookService {
  private final RestTemplate restTemplate;

  public BookService(RestTemplate rest) {
    this.restTemplate = rest;
  }

  /*El método lleva una anotación para indicar el método alternativo
  a invocar en caso de detección de fallo*/
  @HystrixCommand(fallbackMethod = "reliable")
  public String readingList() {
    URI uri = URI.create("http://localhost:8090/recommended");

    return this.restTemplate.getForObject(uri, String.class);
  }

  /*Método alternativo*/
  public String reliable() {
    return "Main Endpoint Down... Alternative Message!";
  }
}
```

Clase Java ReadingApplication.java

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;
import org.springframework.web.client.RestTemplate;

@EnableCircuitBreaker
@RestController
@SpringBootApplication
public class ReadingApplication {

  @Autowired
  private BookService bookService;

  @Bean
  public RestTemplate rest(RestTemplateBuilder builder) {
    return builder.build();
  }

  @RequestMapping("/to-read")
  public String toRead() {
    return bookService.readingList();
  }

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

Configuramos la aplicación spring boot para que se ejecute en un puerto de escucha concreto

Archivo de configuración de Spring src/main/resources/application.properties

```
server.port=8080
```

- Compilamos ambos módulos por separado
- Una vez compilados nos situamos en la carpeta **target** y ejecutamos el .jar tanto del endpoint servidor (circuitbreaker-bookstore), como del endpoint que implementa el patrón circuitbreaker (circuitbreaker-reading)
- Ejecutamos en 2 consolas separadas cada ejecutable

Ejecutamos el circuitbreaker-bookstore

```
$ java -jar circuitbreaker-bookstore/target/circuitbreaker-bookstore-1.0.0.jar
```

Ejecutamos el circuitbreaker-reading

```
$ java -jar circuitbreaker-reading/target/circuitbreaker-reading-1.0.0.jar
```

- Abrimos un navegador web, e indicamos la siguiente url para comprobar lo que nos devuelve el resultado de la invocación del endpoint del bookstore
- Deberíamos de observar que aparece en pantalla el texto: "Spring Book Store App!!!"

```
http://localhost:8090/recommended
```

A continuación:

- Indicamos la url del servicio del pattern circuitbreaker
- Deberíamos de observar lo mismo que invocando a la url anterior

```
http://localhost:8080/to-read
```

Ahora, hacemos la prueba de:

- Tumbar el servidor del bookstore
- Invocar de nuevo la URL: http://localhost:8080/to-read
- Comprobamos ahora que, se muestra en lugar de la invocación al web service, el texto de: "Main Endpoint Down... Alternative Message!"
- Volvemos a arrancar el servidor bookstore
- Invocar de nuevo la URL: http://localhost:8080/to-read
- Comprobamos ahora que, se muestra el resultado de la invocación original del servicio: "Spring Book Store App!!!" == Compensating Transaction

Este patrón, consiste en deshacer el trabajo realizado por una serie de pasos, que juntos definen una operación eventualmente consistente si uno o más de los pasos fallan.

Las operaciones que siguen el modelo de consistencia eventual, se encuentran comúnmente en aplicaciones alojadas en la nube, que implementan procesos de negocio y flujos de trabajo complejos.

2.7. Contexto

Las aplicaciones que se ejecutan en la nube con frecuencia modifican los datos.

Estos datos pueden distribuirse en varias fuentes de datos en diferentes ubicaciones geográficas.