Patrones utilizados:

Circuit Breaker:

El patrón *Circuit Breaker*, es un patrón de resiliencia utilizado para prevenir que una aplicación realice llamadas a servicios remotos que probablemente fallen. Su objetivo es evitar la sobrecarga de solicitudes hacia servicios inestables o fuera de servicio, mejorando la tolerancia a fallos del sistema.

En nuestro proyecto de arquitectura de microservicios ecommerce-microservice-backend-app, el patrón se ha implementado entre los servicios:

- payment-service (cliente)
- order-service (proveedor)

El **payment-service** realiza llamadas HTTP por medio de *restTemplate* al **order-service** para obtener información de órdenes asociadas a un pago. (Esta lógica es fácilmente identificable en el servicio **PaymentServiceImpl** en los métodos *findAll* y *findById*) Para evitar que fallos repetidos del **order-service** afecten al rendimiento del sistema, se implementó un *Circuit Breaker* con la biblioteca **Resilience4j**.

A continuación, se muestra la configuración definida en el application.yml del payment-service:

```yaml
resilience4j:
  circuitbreaker:
    instances:
      paymentService:
        register-health-indicator: true
        event-consumer-buffer-size: 10
        automatic-transition-from-open-to-half-open-enabled: true
        failure-rate-threshold: 50
        minimum-number-of-calls: 5
        permitted-number-of-calls-in-half-open-state: 3
        sliding-window-size: 10
        wait-duration-in-open-state: 5s
        sliding-window-type: COUNT_BASED
```

Parámetros clave:

- **failure-rate-threshold:** 50: si más del 50% de las llamadas fallan en una ventana de tiempo, se abre el circuito.
- **sliding-window-size:** 10: se toman las últimas 10 llamadas para calcular la tasa de fallos.
- **wait-duration-in-open-state:** 5s: el circuito permanecerá abierto durante 5 segundos antes de probar si el servicio se ha recuperado.
- **permitted-number-of-calls-in-half-open-state:** 3: se permiten hasta 3 llamadas para comprobar si el servicio está disponible.

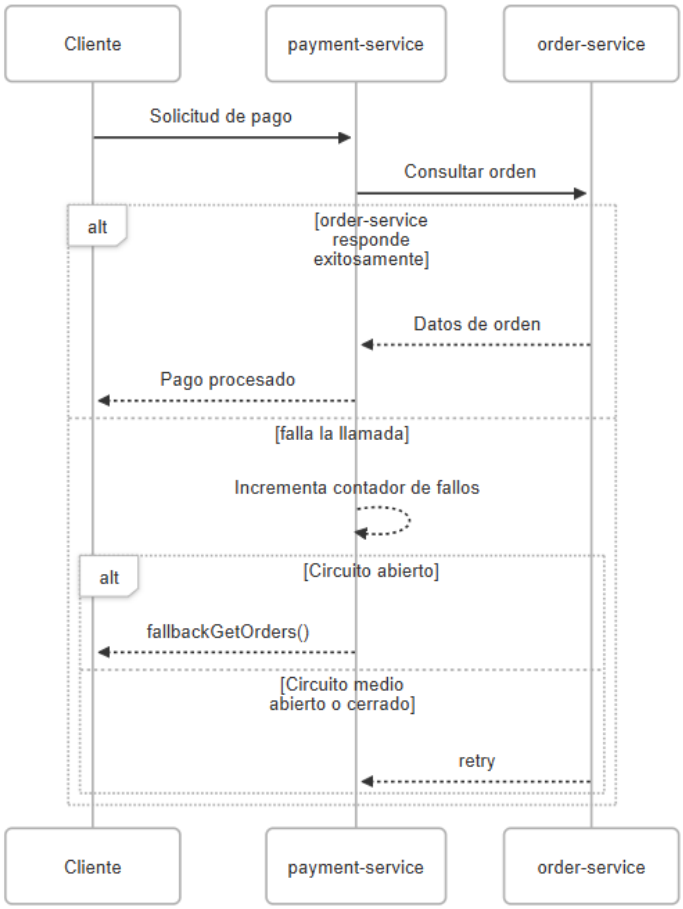
Dentro del **PaymentResource**, la implementación cuando el **order-service** no está disponible y el *Circuit Breaker* está abierto, se activa un método de fallback con el nombre *fallbackGetOrders* para evitar la propagación del fallo:

```
public String fallbackGetOrders(Exception ex) {
    return "Fallback: Unable to retrieve Orders.";
}
```

Esto permite mantener operativa la aplicación, retornando un mensaje controlado al usuario o sistema consumidor, indicándole que el proceso por el momento no está disponible. Por otro lado, y para completar la implementación del patrón, se utiliza el decorador *@CircuitBreaker(name="" fallbackMethod="")* En los métodos del **PaymentResource** que tiene la posibilidad de fallo (findAll, findById).

```
@CircuitBreaker(name = "paymentService", fallbackMethod = "fallbackGetOrders")
@GetMapping
public ResponseEntity<DtoCollectionResponse<PaymentDto>> findAll() {
    log.info(msg:"*** PaymentDto List, controller; fetch all payments *");
    return ResponseEntity.ok(new DtoCollectionResponse<>(this.paymentService.findAll()));
}

@CircuitBreaker(name = "paymentService", fallbackMethod = "fallbackGetUsers")
@GetMapping("/{paymentId}")
public ResponseEntity<PaymentDto> findById(
        @PathVariable("paymentId")
        @NotBlank(message = "Input must not be blank")
        @Valid final String paymentId) {
    log.info(msg:"*** PaymentDto, resource; fetch payment by id *");
    return ResponseEntity.ok(this.paymentService.findById(Integer.parseInt(paymentId)));
```

En el siguiente diagrama se puede observar el funcionamiento de dicho patrón en los tres posibles casos 1. El microservicio order-service esta respondiendo correctamente. 2. El microservicio order-service no esta funcionando. 3. El microservicio order-service vuelve a funcionar correctamente.



Adicionalmente, se cuenta con integración con *Spring Actuator* para exponer el estado del *circuit breaker*. La configuración para lo anterior, también se encuentra definida el el **application.yml** del microservicio **payment-service**.

```yaml
management:
  health:
    circuitbreakers:
      enabled: true
  endpoint:
    health:
      show-details: always
```

Esto permite monitorear el estado del Circuit Breaker desde endpoints como "/actuator/health".

Zipkin:

El patrón de observabilidad distribuida busca recolectar, correlacionar y visualizar trazas de solicitudes que recorren múltiples microservicios. Esto permite identificar cuellos de

botella, errores de comunicación, tiempos de respuesta, y puntos críticos en un sistema distribuido. Se facilita así el monitoreo, el análisis de fallos y el mantenimiento del sistema. Lo que se busca con este patrón es:

- Monitoreo distribuido sin necesidad de instrumentar manualmente cada método.
- Detección proactiva de fallas o demoras.
- Trazabilidad total entre servicios sin modificar lógica de negocio.
- Interfaz visual (Zipkin) que permite analizar gráficamente las trazas y tiempos de cada salto

En nuestro proyecto de arquitectura de microservicios ecommerce-microservice-backend-app, el patrón se ha implementado entre los servicios:

- Zipkin
- Api-gateway

Para la implementación de este patrón fue necesario adicionar las siguientes dependencias dentro del microservicio de **api-gateway.** Adicional a esto fue necesario incluir en el archivo *compose.yml* un nuevo contenedor de Zipkin.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

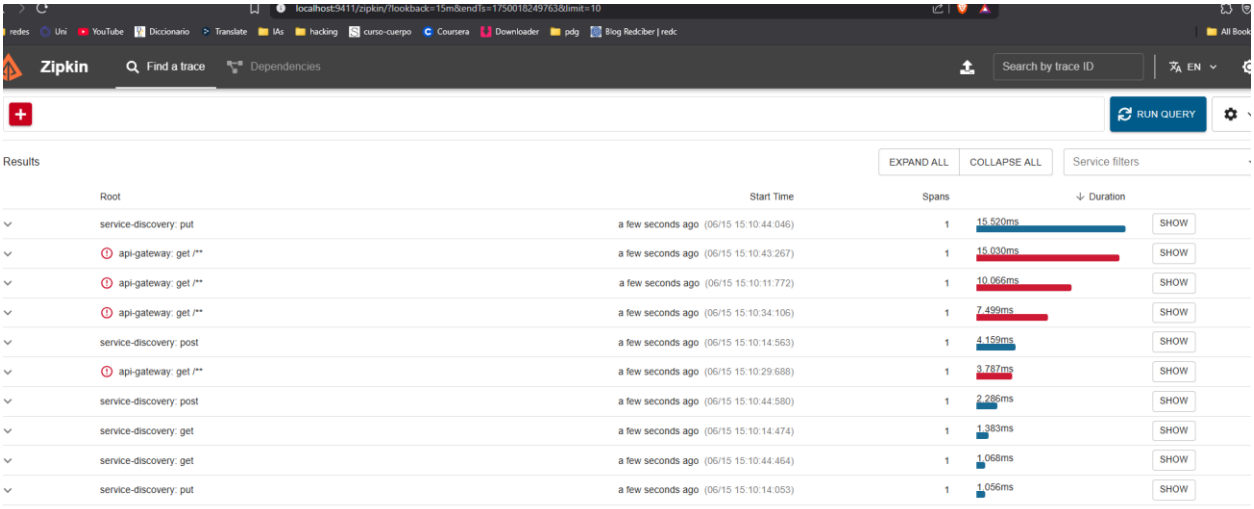
Adicionalmente se agregaron cambios en el archivo application.yml del microservicio api-gateway. Con la intención de mejorar la trazabilidad.

```
spring:
  zipkin:
    base-url: ${SPRING_ZIPKIN_BASE_URL:http://localhost:9411}
  sleuth:
    sampler:
      probability: 1.0
```

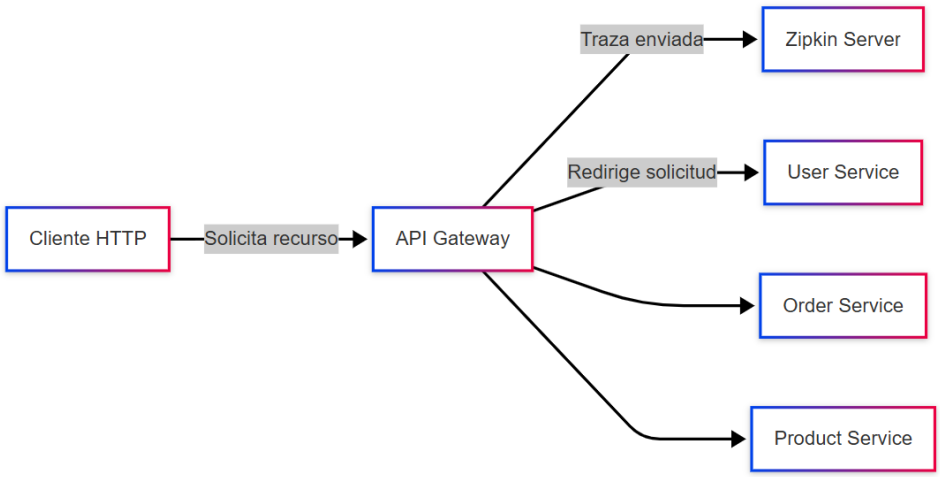
Esto permite:

- Que el microservicio envíe todas las trazas a Zipkin (probability: 1.0).
- Que se use la URL local de Zipkin solo si no se provee una variable de entorno (localhost:9411).

A continuación se puede observar el la interfaz del Zipkin donde ya se encuentra reportando api-gateway al microservici



A contuniación se puede ver el diagrama de arquitectura de la implementación del patrón Zipkin.



Feature toggles:

El patrón *Feature Toggle* permite habilitar o deshabilitar funcionalidades específicas de una aplicación sin necesidad de desplegar nuevas versiones del código. Esta técnica es muy útil para:

- Activar funcionalidades incompletas de forma segura.
- Realizar pruebas A/B o despliegues progresivos (canary releases).
- Activar rápidamente features para ciertos entornos o usuarios.

Se utilizó **Togglz** como framework para gestionar *feature toggles* en el microservicio. Esta biblioteca facilita la definición, persistencia y visualización de los toggles a través de una consola web integrada. El microservicio que fue intervenido fue:

- Order-service

Adicional a esto fue necesario agregar diferentes dependencias para la implementación de dicho patrón en el microservicio.

```
    <artifactId>togglz-core</artifactId>
    <version>3.1.2</version>
</dependency>
<dependency>
    <groupId>org.togglz</groupId>
    <artifactId>togglz-servlet</artifactId>
    <version>3.1.2</version>
</dependency>
<dependency>
    <groupId>org.togglz</groupId>
    <artifactId>togglz-cdi</artifactId>
    <version>3.1.2</version>
</dependency>
<dependency>
    <groupId>org.togglz</groupId>
    <artifactId>togglz-spring-web</artifactId>
    <version>3.1.2</version>
</dependency>
<dependency>
    <groupId>org.togglz</groupId>
    <artifactId>togglz-console</artifactId>
    <version>3.1.2</version>
</dependency>
<dependency>
    <groupId>org.togglz</groupId>
    <artifactId>togglz-spring-boot-starter</artifactId>
    <version>3.1.2</version>
</dependency>
```

Por otro lado, fue necesario agregar configuraciones en el application.yml del microservicio **order-service**. Esto habilita la consola web para administrar los feature toggles en la ruta http://localhost:8300/order-service/togglz-console. Además, de agregar el valor de la constante que nos indica el disparador (DESCOUNT_ON_ORDER_CREATION).

```yaml
togglz:
  console:
    enabled: true
    path: /togglz-console
  features:
    DISCOUNT_ON_ORDER_CREATION:
      enabled: true
```

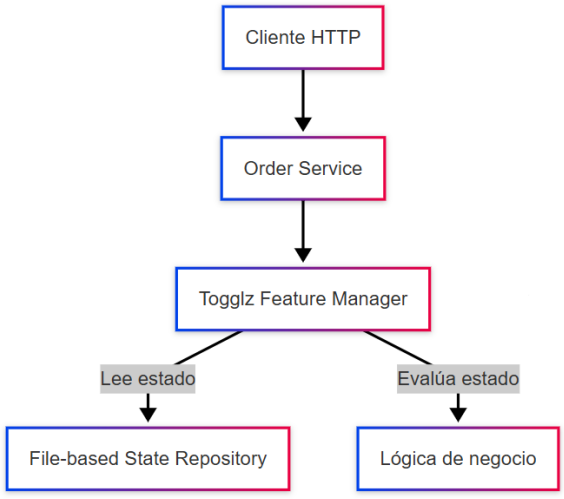
Dentro del módulo *config* del microservicio **order-service**, se creó el archivo *OrderFeatures.java*, el cual es necesario para definir y utilizar el método **isActive()**. Lo anterior nos permite crear la implementación para activar el *toggle* en el momento que sea necesario, es decir, cuando el valor de la constante perteneciente al **enum OrderFeatures** sea true.

```java
package com.selimhorri.app.config.feature;

import org.togglz.core.Feature;
import org.togglz.core.annotation.Label;
import org.togglz.core.context.FeatureContext;

public enum OrderFeatures implements Feature {

    @Label("Descount Feature for Orders")
    DISCOUNT_ON_ORDER_CREATION;

    public boolean isActive() {
        return FeatureContext.getFeatureManager().isActive(this);
    }
}
```

Para el caso de poder modificar el valor del toggle en caliente si necesidad de reconstruir y levantar el servicio. Se realiza la configuración del togglez-console.

```java
package com.selimhorri.app.config.feature;

import java.io.File;

import org.springframework.context.annotation.Configuration;
import org.togglz.core.Feature;
import org.togglz.core.manager.TogglzConfig;
import org.togglz.core.repository.StateRepository;
import org.togglz.core.repository.file.FileBasedStateRepository;
import org.togglz.core.spi.FeatureProvider;
import org.togglz.servlet.user.ServletUserProvider;
import org.togglz.core.user.UserProvider;

@Configuration
public class DemoTogglzConfig implements TogglzConfig {

    @Override
    public Class<? extends Feature> getFeatureClass() {
        return OrderFeatures.class;
    }

    @Override
    public StateRepository getStateRepository() {
        return new FileBasedStateRepository(new File("/tmp/features.properties"));
    }

    @Override
    public UserProvider getUserProvider() {
        return new ServletUserProvider("admin");
    }
}
```

En la siguiente imagen se encuentra un diagrama donde de explica el funcionamiento de la comunicación entre los componentes del microservicio **order-service** a la hora de utilizar el *toggle* implementado.



No fue posible la implementación correcta del patrón a falta de conocimientos.