INFORME DE PRÁCTICAS EN LA EMPRESA VALID SUCURSAL COLOMBIA

Presentado por:

WILL MANUEL LEYTON LÓPEZ

Código:

2015114201

Presentado a:

Ing. YESID GRANADOS

Tutor de Prácticas Profesionales

Mg. CESAR AUGUSTO SÁNCHEZ BAQUERO

Jefe de Desarrollo VALID Sucursal Colombia

UNIVERSIDAD DEL MAGDALENA

FACULTAD DE INGENIERÍA

INGENIERÍA DE SISTEMAS

SANTA MARTA

2018

Tabla de contenido

1. Presentación 3
2. Objetivos y Funciones 4
   2.1. Objetivo General 4
   2.2. Objetivo Específicos 4
   2.3. Funciones del Practicante en la Organización 4
3. Justificación 5
4. Generalidades de la Empresa 7
   4.1. Compromiso 7
   4.2. Historia 7
   4.3. Sostenibilidad y Responsabilidad Social 8
   4.4. Política de Gestión de Responsabilidad Total 8
5. Bases Teóricas Relacionadas 10
6. Desarrollo de Actividades 15
   6.1. Factores que Intervienen en la Arquitectura de Microservicios de VALID. 15
   6.2. Solución para Comunicación Asíncrona 15
   6.3. Solución para Fallbacks para Errores 27
   6.4. Solución con Motor de Búsqueda ElasticSearch 30
7. Cronograma 36
8. Conclusiones 37
9. Bibliografía 38

1. Presentación

En este informe se desarrolla un conjunto de soluciones que permita solucionar diferentes problemas que se presentan al diseñar y desarrollar aplicaciones nativas en la nube con Spring Framework y Spring Cloud, tales como la comunicación entre sistemas de forma asíncrona, que garantice la entrega de mensajes entre sistemas y permita desacoplar la arquitectura de comunicación, otro tema es la resistencia a fallos y no propagación de errores que no podemos controlar, como es el caso de la disponibilidad de un servicio web, y el aspecto final a tratar es la implementación de un sistema que cumpla con las características de motor de búsqueda, permitiendo realizar consultas que mejoren los rendimientos de búsqueda.

2. Objetivos y Funciones

2.1. Objetivo General

Diseñar, y desarrollar módulos que permitan la comunicación asíncrona entre microservicios, fallbacks para errores y el uso de un motor de búsqueda con Spring Framework y Spring Cloud de Java.

2.2. Objetivo Específicos

- Identificar los factores que intervienen en la arquitectura de microservicios de VALID.
- Desarrollar módulos que permitan la comunicación asíncrona entre microservicios.
- Desarrollar módulos que permitan el fallbacks para errores usando el patrón Circuit Breaker.
- Desarrollar módulos que permitan implementar el motor de búsqueda Elasticsearch.

2.3. Funciones del Practicante en la Organización

- Diseño y desarrollo Backend de servicios REST implementando la arquitectura de microservicios con Spring Framework y Spring Cloud de Java.
- Diagramas UML.
- Documentación de código.
- Pruebas de carga con JMeter.
- Despliegue en entornos AWS (Amazon Web Services)
- Uso de patrones de diseño y buenas prácticas de desarrollo.

3. Justificación

El desarrollo backend es una parte fundamental del desarrollo de software, logrando encapsular la lógica de negocio requerida para el cumplimiento de los requerimientos, en los últimos años se ha utilizado el backend con la arquitectura de microservicios, este término “fue discutido en un taller para arquitectos de software cerca de la ciudad de Venecia en mayo del 2011, se utilizó para describir un estilo arquitectural que muchos habían empezado a explorar recientemente. Más tarde, en mayo de 2012, este mismo grupo decide optar por el nombre de microservicios como el término más apropiado para describir sus experiencias” (Barrios D. A., 2018).

Algo que se debe tener en cuenta es que “aunque el término puede ser relativamente nuevo especialmente a partir del 2014 cuando se convirtió en uno de los términos más populares, atrayendo mucha atención, como una nueva manera de pensar en la estructuración de las aplicaciones—, su idea base ha existido por mucho tiempo; adopta, aparentemente, los mismo ideales de SOA (Service Oriented Architecture), tanto que para muchos no es fácil ver una diferencia real. Sin embargo, es importante recalcar que durante los últimos años el término ha sido utilizado para describir un modo particular para diseñar y estructurar software como un conjunto de servicios desplegables independientemente. Y aunque a pesar de su auge no existe una definición precisa de este estilo arquitectural, sí cuenta con una serie de características que giran alrededor de la organización, su lógica del negocio, despliegues automáticos, control descentralizado del lenguaje de programación y datos” (Fowler, 2014).

En la actualidad los sistemas requieren de comunicación entre ellos, “podemos identificar diferentes estilos de comunicación entre microservicios, es posible clasificarlos en dos dimensiones, la primera de ellas es una división en protocolos de comunicación síncrona y asíncrona. El punto clave de la comunicación asíncrona es que el cliente no debería bloquear un hilo mientras espera una respuesta, el protocolo más popular para ese tipo de comunicación es AMQP. Sin embargo, la principal forma de comunicación entre servicios sigue siendo el protocolo HTTP síncrono” (Mińkowski, Different styles of communication, 2018).

Cuando nuestros sistemas requieren de integraciones y procesamientos basados en herramientas externas, debemos asegurar que sean resistentes a fallos, garantizando que las excepciones no se propaguen en cascada, logrando brindar comportamientos aceptables para que los sistemas no sirvan respuestas complejas de manejar o procesar para los clientes.

En muchos casos las consultas de datos pueden causar muchos inconvenientes, asociados con los tiempos de espera o fallos en los criterios de búsqueda, por lo que se necesita brindar una solución que permitan cumplir con las necesidades de obtener datos para su análisis con la más alta precisión de los criterios establecidos y que además brinde escalabilidad.

En VALID como toda empresa que desarrolla software a medida se requiere brindar soluciones que permitan integrar sistemas para comunicación entre ellos, garantizando

un débil acoplamiento, lo cual se refleja en poca complejidad de desarrollo y mantenimiento, los fallos son un aspecto que preocupan en las diferentes etapas del desarrollo de software, por lo cual, diseñar y desarrollar soluciones que permitan disminuir la incidencia de fallos o control de ellos en tiempo de ejecución son una necesidad imperante, ya que los sistemas serían más confiables y estables, otro aspecto importante es el relacionado con la búsqueda o consulta de información, ya sea para el cumplimiento de los requerimientos de negocio o el análisis de la información para la toma de decisiones, por lo que la implementación de motores de búsqueda que faciliten la consulta y disminuyan sus tiempos son soluciones factibles al momento de realizar búsquedas.

4. Generalidades de la Empresa

VALID desarrolla soluciones digitales que permiten a las empresas, agencias gubernamentales, consumidores y ciudadanos en general identificarse a sí mismos, comunicar y realizar transacciones financieras o de datos totalmente seguras. En la actualidad, se encuentra trabajando en 16 países de todo el mundo y lleva más de 60 años ofreciendo soluciones únicas a clientes, que son los pilares sobre los que se asientan las tecnologías y los procesos seguros (VALID, VALID, s.f.).

4.1. Compromiso

Desarrollar y personalizar la plataforma de soluciones seguras, haciéndolas a medida para cada cliente y buscando satisfacer las necesidades únicas de cada uno. Además, la búsqueda de la innovación inspira a los clientes a seguir avanzando, permitiéndoles convertirse en los protagonistas de un mundo en constante cambio (VALID, VALID, s.f.).

4.2. Historia

Año	Descripción
1957	Valid empieza a operar en Brasil, bajo el nombre de Thomas de La Rue, como filial de una compañía inglesa que presta servicios de papel de alta seguridad y tecnología de impresión (VALID, VALID, s.f.).
1993	Después de tres décadas, American Banknote compra Thomas de La Rue, lo que supone el comienzo de una etapa de gran expansión. La compañía pronto se convierte en un referente en Brasil para la emisión segura de documentos (VALID, VALID, s.f.).
2006	La compañía lanza una oferta pública inicial con el 57% de su capital social, consiguiendo un total de 480 millones de reales brasileños (VALID, VALID, s.f.).
2007	La cartera de soluciones de Medios de Pago ya está disponibles en Argentina (VALID, VALID, s.f.).
2010	La compañía cambia de nombre convirtiéndose en Valid, una compañía muy importante que empieza a operar en Europa, estableciendo las bases para una nueva era (VALID, VALID, s.f.).
2012	Con presencia en Brasil, Argentina y España, Valid empieza a consolidar su presencia en Estados Unidos, ofreciendo soluciones de Data, Pago y Mobile (VALID, VALID, s.f.).
2013	Valid amplía su cartera de soluciones en Estados Unidos ofreciendo la solución de Identificación Segura (VALID, VALID, s.f.).

2015	Las nuevas adquisiciones refuerzan la presencia de Valid en Estados Unidos, Europa, África, Asia y Oriente Medio, expandiendo su presencia en los cinco continentes (VALID, VALID, s.f.).
2017	Valid se une al universo IoT a través de su inversión en Cubic Telecom (VALID, VALID, s.f.).

Fuente: Autor

4.3. Sostenibilidad y Responsabilidad Social

En VALID creemos que impulsar la innovación en el mercado global mediante la ejecución de iniciativas que promuevan la sostenibilidad ambiental y la responsabilidad social, es nuestra responsabilidad para con nuestros grupos de interés y las regiones económicas donde operamos. Nuestro marco de sostenibilidad abarca objetivos ambientales y sociales dentro de nuestro negocio, que impactan con nuestro mundo. Con más de 6.000 empleados, tenemos una profunda influencia en las comunidades donde operamos, mientras que nuestras iniciativas reflejan un enfoque en el desarrollo social continuo (VALID, VALID, s.f.).

Los procesos de producción que llevamos a cabo en nuestras plantas hacen un uso eficiente de los recursos naturales, desde la adquisición de las materias primas hasta el uso de materiales no tóxicos ni contaminantes para la gestión de los residuos (VALID, VALID, s.f.).

Reconocemos que la sostenibilidad es muy variada y, por eso, buscamos continuamente formas de fomentar una cultura que promueva la responsabilidad social y el progreso medioambiental a través de la innovación operativa (VALID, VALID, s.f.).

4.4. Política de Gestión de Responsabilidad Total

El sistema de gestión de responsabilidad total de VALID incluye certificaciones, normas, el Código de ética y conducta de la empresa, la Política de seguridad patrimonial, la Política de seguridad de información, la Política de gestión de riesgos, así como otros requisitos legales aplicables. Declaramos nuestra política ante todas las partes interesadas (VALID, VALID, s.f.).

Calidad en los productos y servicios	Trabajamos sin descanso para garantizar la total satisfacción y fidelidad de nuestros clientes a través de la innovación y la mejora continua de nuestros procesos, servicios y soluciones (VALID, VALID, s.f.).
Seguridad física y seguridad de la información	Nuestro principal patrimonio es la confianza de nuestros clientes. Valid posee políticas destinadas a garantizar

	la seguridad física de las instalaciones, la confidencialidad y la seguridad de la información (VALID, VALID, s.f.). La Política de seguridad patrimonial y la Política de seguridad de la información establecen los métodos y las conductas que se deben cumplir íntegramente (VALID, VALID, s.f.).
Responsabilidad social	Nos preocupamos por el bienestar, la salud y la seguridad de nuestros empleados, y nuestro principio es el respeto de los derechos humanos y la dignidad. Las normas éticas que guían las relaciones personales y profesionales entre las partes interesadas aparecen declaradas y definidas en el Código de ética y conducta de Valid (VALID, VALID, s.f.).
Responsabilidad ambiental	Expresamos la responsabilidad ambiental mediante la implementación de programas que fomentan el uso racional de los recursos naturales así como el cumplimiento de la legislación aplicable (VALID, VALID, s.f.).
Gestión de riesgos	En Valid adoptamos medidas que buscan gestionar los riesgos operacionales y estratégicos, las cuales se definen en la Política de gestión de riesgos (VALID, VALID, s.f.).

Fuente: Autor

5. Bases Teóricas Relacionadas

En el artículo *Arquitectura de Software basada en Microservicios para Desarrollo de Aplicaciones Web* los autores tienen como objetivo identificar las tecnologías y metodologías disponibles para el desarrollo e implementación de microservicios utilizando un enfoque cualitativo, además de la implementación y discriminación de requisitos y necesidades para desarrollos web, satisfaciéndolos con soluciones orientadas a microservicios (López & Maya, 2107).

En el libro *Mastering Spring Cloud* el autor introduce a los microservicios como "una de las tendencias más importantes que han surgido en el mundo de las TI durante los últimos años. Es relativamente fácil identificar las razones más importantes de su creciente popularidad. Tanto sus ventajas como sus desventajas son bien conocidas, aunque lo que describimos como desventajas se puede resolver fácilmente utilizando las herramientas adecuadas. Las ventajas que ofrecen incluyen escalabilidad, flexibilidad y entrega independiente; Estas son las razones de su creciente popularidad. Hay algunas tendencias de TI anteriores que tuvieron cierta influencia sobre este crecimiento en la popularidad de los microservicios. Me refiero a tendencias como el uso de entornos comunes basados en la nube y la migración de bases de datos relacionales a NoSQL (Mińkowski, Introduction to Microservices, 2018).

El concepto de microservicios "define una aproximación a la arquitectura de sistemas de TI que dividen una aplicación en una colección de servicios acoplados libremente que implementan los requisitos del negocio. De hecho, esta es una variante del concepto de arquitectura orientada a servicios (SOA). Uno de los beneficios más importantes de una migración a una arquitectura basada en microservicios Es una capacidad para realizar la entrega continua de aplicaciones grandes y complejas" (Mińkowski, Introduction to Microservices, 2018).

En el artículo *Microservice architecture* los autores identifican las características más significativas de los microservicios, la cuales son:

- Pequeño y enfocado en hacer una sola cosa bien: donde los microservicios intentan aislar la especialidad de una lógica de negocio específica en un solo lugar, lo que permite disminuir la tentación de hacer crecer mucho el componente (Barrios D. , 2018).
- Organizado en torno a la lógica de negocio: Divide las organizaciones alrededor de la lógica de negocio (Barrios D. , 2018).
- Autonomía: Los microservicios son una entidad separada. Deben ser desplegados como un servicio aislado en infraestructuras (Barrios D. , 2018).
- Diversidad de Tecnologías: Dado que cada microservicio es una unidad de despliegue independiente, se cuenta con una considerable libertad en las elecciones de tecnología que se puedan tomar (Barrios D. , 2018).

Los microservicios "están estrechamente vinculados a las plataformas de computación en la nube, pero el concepto real de microservicios no es nada nuevo. Este enfoque se ha aplicado en el mundo del desarrollo de TI durante muchos años,

pero ahora, a través de la popularidad de las soluciones en la nube, ha evolucionado a un nivel más alto. No es difícil señalar las razones de esta popularidad. El uso de una nube le ofrece escalabilidad, confiabilidad y bajos costos de mantenimiento en comparación con las soluciones locales dentro de la organización. Esto ha llevado al aumento de los enfoques de desarrollo de aplicaciones nativas de la nube que pretenden brindarle los beneficios de todas las ventajas que ofrece la escala elástica similar a la nube, las implementaciones inmutables y las instancias desechables. Todo se reduce a una cosa: disminuir el tiempo y el costo necesarios para cumplir con los nuevos requisitos. Hoy en día, los sistemas de software y las aplicaciones se están mejorando continuamente. Si tiene un enfoque tradicional para el desarrollo, basado en monolitos, una base de código crece y se vuelve demasiado compleja para modificaciones y mantenimiento. La introducción de nuevas características, marcos y tecnologías se vuelve difícil, lo que a su vez impacta las innovaciones e inhibe las nuevas ideas. No podemos discutir con eso. También hay otra cara de esta moneda. Hoy en día, prácticamente todos piensan en la migración a la nube, en parte porque está de moda. ¿Todos necesitan esto? Ciertamente no. Aquellos que no están absolutamente seguros de migrar sus aplicaciones a un proveedor de nube remota, como AWS, Azure o Google, desearían tener al menos una nube privada local o contenedores Docker" (Mińkowski, Cloud-native development, 2018).

Un aspecto importante es la comunicación entre microservicios, en el artículo *Un Acercamiento a los Microservicios* los autores abordan la "implementación de dos maneras: mediante peticiones HTTP o a través de cola de mensajes (Azure Service Bus, RabbitMQ, Apache Kafka, AtciveMQ, etc.). HTTP es comunicación directa y debe usarse cuando se desea una respuesta inmediata del otro servicio, mientras que el mecanismo publicación/suscripción de la cola de mensaje, es una comunicación asíncrona, que hace la solicitud del servicio y continúa la operación cuando recibe la respuesta de la contraparte" (Gómez, Anaya, & Cano, 2018).

Este tipo de comunicación asíncrona es basada en los "Patrones de Integración Empresarial (EIP) define una serie de componentes que normalmente se utilizan para la orquestación en sistemas distribuidos. Probablemente haya escuchado sobre patrones como canales de mensajes, enrutadores, agregadores o endpoints" (Mińkowski, Message-Driven Microservices, 2018).

La topología de mensajería centralizada, "es similar a la basada en REST, excepto que en lugar de usar REST para acceso remoto, esta utiliza un intermediario de mensajes centralizado ligero. El agente de mensajes ligeros que se encuentra en esta topología no realiza ninguna orquestación, transformación o enrutamiento complejo, es sólo un trasporte ligero para acceder a los componentes de servicios remotos. Esta topología se encuentra frecuentemente en aplicaciones de negocios más grandes o aplicaciones que requieren un control más sofisticado sobre la capa de transporte entre la interfaz de usuario y los componentes de servicio. Entre los beneficios que aporta frente a otros métodos es que son mecanismos avanzados de colas, con mensajería asíncrona, monitoreo, manejo de errores y mejor balanceo

de carga y escalabilidad. El único punto de fallo y los problemas de cuello de botella arquitectónico generalmente asociados con un intermediario de mensajes se abordan a través de la agrupación de intermediarios y de la federación de intermediarios (dividir una única instancia de intermediario en múltiples instancias para dividir la carga de procesamiento de mensajes en función de las áreas funcionales del sistema)" (López & Maya, 2107).

En los diferentes procesos de software es posible que ocurran fallos, "los sistemas de microservicios consisten en múltiples unidades pequeñas que pueden fallar, pero debido a que estas están ligeramente acopladas, pueden ser restauradas automáticamente, lo que resulta en un sistema más estable y robusto" (López & Maya, 2107). Sin embargo cuando los errores ocurren en procesos que están relacionados con integraciones o solicitudes, es posible que estos fallos sean más difíciles de controlar, y en la mayoría de casos un servicio que falla probablemente tenga otros servicios que dependen de él, "por lo que si no planificamos adecuadamente este evento, corremos el riesgo de que todos los demás servicios que dependen de él también dejen de responder. Esto se llama un fallo en cascada. El patrón Circuit Breaker tiene como objetivo evitar que un solo componente caiga en cascada más allá de sus límites y, por lo tanto, derribe todo el sistema. El lema aquí es fallar rápidamente: cuando un servicio deja de responder, sus invocadores deben dejar de esperar, asumir lo peor y comenzar a lidiar con el hecho de que el servicio que falla no está disponible. Por lo tanto, los interruptores de circuito contribuyen a la estabilidad y la capacidad de recuperación tanto de los clientes como de los servicios: los clientes limitan su desperdicio de recursos al intentar acceder a los servicios que no responden, y los servicios sobrecargados tienen la oportunidad de recuperarse al terminar algunas de las tareas que están procesando actualmente. Concretamente, un interruptor automático funciona envolviendo las llamadas hacia un servicio objetivo y monitoreando sus tasas de falla. La idea es que cuando el servicio de destino se vuelve demasiado lento o responde con frecuencia con fallas, el interruptor se disparará y las futuras invocaciones del cliente devolverán inmediatamente una falla. Más específicamente, el patrón se puede implementar como una máquina de estados finitos" (Montesi & Weber, 2016). En el siguiente diagrama podemos ilustrar el funcionamiento del patrón Circuit Breaker.

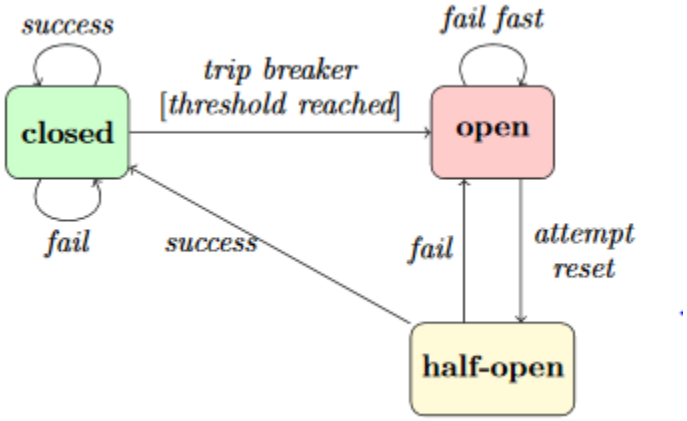


Figura 1: Montesi, Fabrizio; Weber, Janine. (2016). *Circuit Breakers, Discovery, and API Gateways in Microservices.*

- **Closed:** “Las solicitudes se pasan al servicio de destino. Las fallas causadas por la operación solicitada, tales como excepciones o tiempos de espera aumentan los contadores de fallas y tiempos de espera respectivos del interruptor. Si estos contadores superan un umbral especificado, o si se cumplen otros criterios predefinidos (por ejemplo, se generó una falla particular), el interruptor se dispara y las transiciones pasan al estado abierto“ (Montesi & Weber, 2016).
- **Open:** “Las solicitudes no se pasan al servicio de destino. En su lugar, un mensaje de error se entrega inmediatamente al cliente como respuesta. Se pueden llamar a los posibles mecanismos de recuperación para manejar la falla. El interruptor automático puede pasar del estado abierto al estado semi-abierto, ya sea haciendo ping periódicamente al servicio para verificar cuándo vuelve a responder, o después de un período de tiempo específico” (Montesi & Weber, 2016).
- **Half-open:** “Mientras se encuentre en este estado, se permite un número limitado de solicitudes al servicio. Siempre que el servicio de destino envíe respuestas exitosas, el interruptor automático se restablece al estado cerrado y se reinician los contadores de fallos y tiempos de espera. Sin embargo, en caso de que alguna de las solicitudes falle mientras se encuentra en estado semi abierto, el interruptor automático vuelve a pasar al estado abierto.” (Montesi & Weber, 2016).

Otro aspecto importante es el relacionado con las búsquedas de datos, en el artículo *ElasticSearch An advanced and quick search technique to handle voluminous data* los autores estudian a Elasticsearch, definiéndola como “una nueva base de datos creada para manejar grandes cantidades de volumen de datos con muy alta disponibilidad y para distribuirse en muchas máquinas para que sea tolerante a fallos y escalable, al mismo tiempo que mantiene una API simple pero potente que permite que aplicaciones de cualquier idioma o marco accedan a la base de datos. Una base de datos distribuida horizontalmente escalable basada en LUCENE de APACHE que brinda una experiencia de búsqueda completa en terabytes de datos con una API simple pero potente. Muchas empresas utilizan elasticsearch para ayudarles a implementar potentes capacidades de búsqueda en sus aplicaciones que son fáciles de configurar, escalables y diseñadas para la nube” (Sai & Kumar, 2013). Las características principales de ElasticSearch son las siguientes:

- **API REST:** “ElasticSearch almacena / recupera objetos a través de una API REST. Se proporcionan los métodos PUT, POST, GET y DELETE que implementan las verificaciones de versión (opcionalmente en PUT), generan identificadores (opcionalmente en POST) y le permiten leer sus propias escrituras (en GET). Esto es lo que lo convierte en un almacén de valores clave” (Sai & Kumar, 2013).
- **Almacén de valor clave:** “En ElasticSearch, cada dato tiene un índice y tipo definidos. Puede pensar en un índice como una colección de documentos o

una tabla en una base de datos. Sin embargo, aquí los documentos agregados a un índice no tienen tipos definidos de estructura y campo. Los objetos tienen un tipo y van en un índice. Entonces, desde un punto de vista REST, el uri relativo a cualquier objeto es /{index}/{type}/{id}. Crea índices y tipos en tiempo de ejecución a través de una API REST" (Sai & Kumar, 2013).

- **Multi-tenancy:** "Puede crear, actualizar, recuperar y eliminar índices. Puede configurar la fragmentación y la replicación por índice. Eso significa que ElasticSearch es multi arrendatario y bastante flexible" (Sai & Kumar, 2013).
- **Mapeo:** "ElasticSearch indexa los documentos que almacena usando un mapeo dinámico o un mapeo que usted proporciona (recomendado). Esto significa que también puede volver a encontrar sus documentos a través de la API de búsqueda" (Sai & Kumar, 2013).
- **Fragmentación y Replicación:** "Se logra una mejor disponibilidad y rendimiento a través de las réplicas (copias de las partes del índice)" (Sai & Kumar, 2013).

En el API de búsqueda de ElasticSearch es donde entra en juego "Lucene. A diferencia de GET, la búsqueda no le permite leer sus propias escrituras de inmediato, ya que los índices se actualizan y los tiempos de replicación son más eficientes. La API de búsqueda se expone como un recurso _search que está disponible en el nivel de servidor (/ _search), nivel de índice (/{índice}/_ búsqueda, o nivel de tipo (/{índice}/{tipo}/_búsqueda). puede buscar en múltiples índices, porque ElasticSearch se está replicando y fragmentando, también en múltiples máquinas" (Sai & Kumar, 2013).

6. Desarrollo de Actividades

6.1. Factores que Intervienen en la Arquitectura de Microservicios de VALID.

VALID implementa la arquitectura de microservicios usando el framework para aplicaciones nativas en la nube Spring Cloud, utilizando los componentes que provee este framework, tales como servidor de configuración distribuida, servidor de registro de servicios, Gateway y los diferentes microservicios que cumplen con los requerimientos de negocio, en el siguiente diagrama se ilustran los componentes básicos de la arquitectura de VALID.

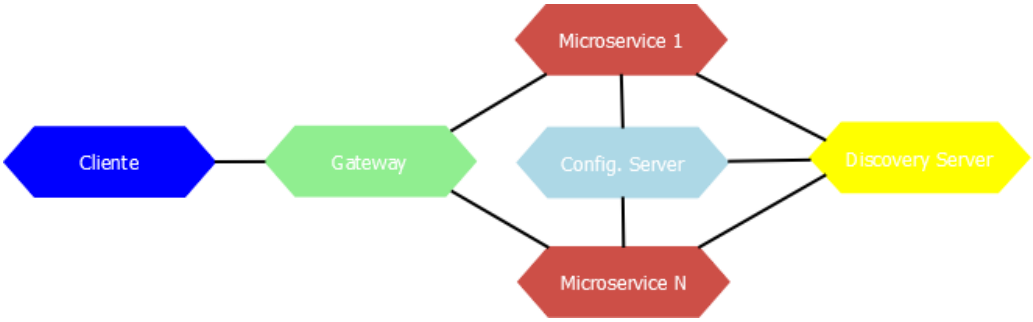


Figura 2: *Autor*

6.2. Solución para Comunicación Asíncrona

En todo ambiente que implemente la arquitectura de microservicios es necesario que exista comunicación entre algunos de los microservicios, la comunicación puede ser de dos tipos: síncrona o asíncrona, algunas veces es necesario que se realice una comunicación y se sigan efectuando las operaciones sin esperar el resultado del envió del mensaje, para este caso afrontaremos dos enfoques:

a) **Comunicación asíncrona mediante un bróker de mensajerías, implementado un sistema de colas:** Existen diversos sistemas que funcionan como intermediarios de mensajes, entre las tecnologías open source más populares tenemos *RabbitMQ*, *Apache Kafka* y *ActiveMQ*, en este caso realizaremos una implementación con *RabbitMQ* debido a su madures y soporte con conectores para diferentes lenguajes como *Java*, *Erlang*, *Python*, *Ruby*, *JavaScript*, *PHP*, entre otros, evidenciando la ventaja de poder funcionar si se desea agregar microservicios escritos con otros lenguajes. Se utilizara un modelo basado en *topics*, ya que permite implementar un modelo en el que muchos microservicos puedan recibir el mensaje del microservicio emisor, para la implementación es necesario tener instalado el bróker RabbitMQ en en servidor, al cual tendrán acceso para conexión los diferentes microservicios que queremos que participen en la

comunicación, a continuación se desarrollara el código a agregar a los microservicios encargados del envio y recepción de mensajes:

Sender Microservice:

(a) Agregar dependencia para conexión al bróker en el archivo ***build.gradle***:

compile('org.springframework.boot:spring-boot-starter-amqp')

(b) Agregar propiedades de conexión en el archivo **.yml** del microservicio alojado en el servidor de configuraciones o en ***bootstap.yml***, estas configuraciones nos permiten conectarnos a RabbitMQ y realizar el enrrutamiento:

```
spring:
  rabbitmq:
    host: {hostname}
    port: {port}
    username: {username}
    password: {password}
rabbitmq:
  exchange:
    name: {exchange-name}
    durable: true
    routing-key: {your-routing-key}.#
```

(c) Agregar clase de configuración ***RabbitMqConfiguration.java***, la cual crea los bean necesarios para la operaciones de envió de mensajes:

```
package com.valid.sender.configuration;

import org.springframework.amqp.core.AmqpTemplate;
import org.springframework.amqp.core.Exchange;
import org.springframework.amqp.core.ExchangeBuilder;
import org.springframework.amqp.rabbit.connection.CachingConnectionFactory;
import org.springframework.amqp.rabbit.connection.ConnectionFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import
    org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;
import org.springframework.amqp.support.converter.MessageConverter;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class RabbitMqConfiguration {

```

```
19.  @Value(
20.          value = "${spring.rabbitmq.host}")
21.  private String rabbitHost;
22.
23.  @Value(
24.          value = "${spring.rabbitmq.port}")
25.  private int rabbitPort;
26.
27.  @Value(
28.          value = "${spring.rabbitmq.username}")
29.  private String rabbitUsername;
30.
31.  @Value(
32.          value = "${spring.rabbitmq.password}")
33.  private String rabbitPassword;
34.
35.  @Value(
36.          value = "${rabbitmq.exchange.name}")
37.  private String exchangeName;
38.
39.  @Value(
40.          value = "${rabbitmq.exchange.durable:true}")
41.  private boolean exchangeDurable;
42.
43.  @Bean
44.  public ConnectionFactory connectionFactory() {
45.     CachingConnectionFactory connectionFactory =
46.          new CachingConnectionFactory(rabbitHost);
47.     connectionFactory.setPort(rabbitPort);
48.     connectionFactory.setUsername(rabbitUsername);
49.     connectionFactory.setPassword(rabbitPassword);
50.     return connectionFactory;
51.  }
52.
53.  @Bean
54.  public Exchange exchange() {
55.     return ExchangeBuilder.topicExchange(exchangeName)
56.                .durable(exchangeDurable).build();
57.  }
58.
59.  @Bean
60.  public MessageConverter messageConverter() {
61.     return new Jackson2JsonMessageConverter();
62.  }
63.
64.  @Bean
65.  public AmqpTemplate amqpTemplate(ConnectionFactory connectionFactory,
```

```
66.             MessageConverter messageConverter) {
67.         RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
68.         rabbitTemplate.setMessageConverter(messageConverter);
69.         return rabbitTemplate;
70.     }
71. }
```

(d) Crear interface ***SenderService.java***, la cual tiene el contrato del método send:

```
1.  package com.valid.sender.service.inter;
2.
3.  public interface SenderService {
4.
5.      public void send(Object message);
6.  }
```

(e) Crear implementación ***SenderServiceImp.java***, que aloja la lógica para envió de mensajes:

```
1.  package com.valid.sender.service.inter;
2.
3.  import org.slf4j.Logger;
4.  import org.slf4j.LoggerFactory;
5.  import org.springframework.amqp.AmqpException;
6.  import org.springframework.amqp.core.AmqpTemplate;
7.  import org.springframework.amqp.core.Exchange;
8.  import org.springframework.beans.factory.annotation.Autowired;
9.  import org.springframework.beans.factory.annotation.Value;
10. import org.springframework.stereotype.Service;
11.
12. import com.fasterxml.jackson.core.JsonProcessingException;
13. import com.fasterxml.jackson.databind.ObjectMapper;
14. import com.valid.sender.service.inter.SenderService;
15.
16. @Service
17. public class SenderServiceImp implements SenderService {
18.
19.     private static final Logger LOGGER =
20.             LoggerFactory.getLogger(SenderServiceImp.class);
21.
22.     @Value(
23.             value = "${rabbitmq.routing-key}")
24.     private String routingKey;
25.
26.     @Autowired
27.     private AmqpTemplate rabbitTemplate;
```

```
28.
29.     @Autowired
30.     private Exchange exchange;
31.
32.     @Override
33.     public void send(Object message) {
34.         try {
35.           rabbitTemplate
36.                .convertAndSend(exchange.getName(), routingKey, message);
37.         } catch (AmqpException e) {
38.           LOGGER.error("error {}", e);
39.         }
40.     }
41. }
```

(f) Implementar el SenderService.java utilizando la inyección de dependencias en los microservicios que se requiera.

Receiver Microservice:

(a) Agregar dependencia para conexión al bróker en el archivo ***build.gradle***:

```
compile('org.springframework.boot:spring-boot-starter-amqp')
```

(b) Agregar propiedades de conexión en el archivo **.yml** del microservicio alojado en el servidor de configuraciones o en ***bootstap.yml***, estas configuraciones nos permiten conectarnos a RabbitMQ y realizar el enrutamiento mediante el exchange y la queue:

```
1.  spring:
2.    rabbitmq:
3.      host: {hostname}
4.      port: {port}
5.      username: {username}
6.      password: {password}
7.
8.  rabbitmq:
9.    queue:
10.     name: {queue-name}
11.     durable: true
12.   exchange:
13.     name: {exchange-name}
14.     durable: true
15.     routing-key: {your-routing-key}.#
```

(c) Agregar clase de configuración ***RabbitMqConfiguration.java***, la cual crea los bean necesarios para la operaciones de conexión a RabbitMQ y recepción de mensajes:

```
2.  package com.valid.receiver.configuration;
3.
4.  import org.springframework.amqp.core.Binding;
5.  import org.springframework.amqp.core.BindingBuilder;
6.  import org.springframework.amqp.core.Exchange;
7.  import org.springframework.amqp.core.ExchangeBuilder;
8.  import org.springframework.amqp.core.Queue;
9.  import org.springframework.amqp.core.QueueBuilder;
10. import org.springframework.amqp.rabbit.annotation.EnableRabbit;
11. import
    org.springframework.amqp.rabbit.config.SimpleRabbitListenerContainerFactory;
12. import org.springframework.amqp.rabbit.connection.CachingConnectionFactory;
13. import org.springframework.amqp.rabbit.connection.ConnectionFactory;
14. import
    org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;
15. import org.springframework.amqp.support.converter.MessageConverter;
16. import org.springframework.beans.factory.annotation.Value;
17. import org.springframework.context.annotation.Bean;
18. import org.springframework.context.annotation.Configuration;
19.
20. @EnableRabbit
21. @Configuration
22. public class RabbitMqConfiguration {
23.
24.     @Value(
25.             value = "${spring.rabbitmq.host}")
26.     private String rabbitHost;
27.
28.     @Value(
29.             value = "${spring.rabbitmq.port}")
30.     private int rabbitPort;
31.
32.     @Value(
33.             value = "${spring.rabbitmq.username}")
34.     private String rabbitUsername;
35.
36.     @Value(
37.             value = "${spring.rabbitmq.password}")
38.     private String rabbitPassword;
39.
40.     @Value(
41.             value = "${rabbitmq.queue.name}")
42.     private String queueName;
43.
44.     @Value(
45.             value = "${rabbitmq.queue.durable:true}")
46.     private boolean queueDurable;
```

```
47.
48.     @Value(
49.             value = "${rabbitmq.exchange.name}")
50.     private String exchangeName;
51.
52.     @Value(
53.             value = "${rabbitmq.exchange.durable:true}")
54.     private boolean exchangeDurable;
55.
56.     @Value(
57.             value = "${rabbitmq.routing-key}")
58.     private String routingKey;
59.
60.     @Bean
61.     public ConnectionFactory connectionFactory() {
62.         CachingConnectionFactory connectionFactory =
63.                 new CachingConnectionFactory(rabbitHost, rabbitPort);
64.         connectionFactory.setUsername(rabbitUsername);
65.         connectionFactory.setPassword(rabbitPassword);
66.         return connectionFactory;
67.     }
68.
69.     @Bean
70.     public Exchange exchange() {
71.         return ExchangeBuilder.topicExchange(exchangeName)
72.                 .durable(true).build();
73.     }
74.
75.     @Bean
76.     public Binding binding(Exchange exchange, Queue queue) {
77.         return BindingBuilder.bind(queue)
78.                 .to(exchange).with(routingKey).noargs();
79.     }
80.
81.     @Bean
82.     public Queue queue() {
83.         return QueueBuilder.durable(queueName).build();
84.     }
85.
86.     @Bean
87.     public MessageConverter messageConverter() {
88.         return new Jackson2JsonMessageConverter();
89.     }
90.
91.     @Bean(
92.             name = {
93.                 "rabbitListenerContainerFactory"
```

```
94.         })
95.     public SimpleRabbitListenerContainerFactory rabbitListenerContainerFactory(
96.         ConnectionFactory connectionFactory,
97.         Queue queue, MessageConverter messageConverter) {
98.       SimpleRabbitListenerContainerFactory factory =
99.         new SimpleRabbitListenerContainerFactory();
100.            factory.setConnectionFactory(connectionFactory);
101.            factory.setMessageConverter(messageConverter);
102.            factory.setConcurrentConsumers(3);
103.            factory.setMaxConcurrentConsumers(10);
104.            return factory;
105.          }
106.      }
```

(d) Crear interface ***ReceiverService.java***, la cual tiene el contrato del método receiver:

```
1.  package com.valid.receiver.service.inter;
2.
3.  public interface ReceiverService {
4.
5.      public void receive(Object message);
6.  }
```

(e) Crear implementación ***ReceiverServiceImp.java***, que aloja la lógica para la recepción de mensajes:

```
107.    package com.valid.receiver.service.imple;
108.
109.    import org.slf4j.Logger;
110.    import org.slf4j.LoggerFactory;
111.    import org.springframework.amqp.rabbit.annotation.RabbitListener;
112.    import org.springframework.beans.factory.annotation.Autowired;
113.    import org.springframework.beans.factory.annotation.Value;
114.    import org.springframework.stereotype.Service;
115.
116.    import com.fasterxml.jackson.core.JsonProcessingException;
117.    import com.fasterxml.jackson.databind.ObjectMapper;
118.    import com.valid.receiver.service.inter.ReceiverService;
119.
120.    @Service
121.    public class ReceiverServiceImp implements ReceiverService {
122.
123.        private static final Logger LOGGER =
124.            LoggerFactory.getLogger(ReceiverServiceImp.class);
125.
```

```
126.         @Autowired
127.         private ObjectMapper objectMapper;
128.
129.         @RabbitListener(
130.                 queues = {
131.                    "${rabbitmq.queue.name}"
132.                 },
133.                 containerFactory = "rabbitListenerContainerFactory")
134.         @Override
135.         public void receive(Object message) {
136.             String info = "Error";
137.             try {
138.                 info = objectMapper.writeValueAsString(message);
139.             } catch (JsonProcessingException e) {
140.                 LOGGER.error("{}", e.getMessage());
141.             }
142.             LOGGER.info("receive message -> {}", info);
143.         }
144.     }
```

(f) Implementar el ReceiverService.java utilizando la inyección de dependencias en los microservicios que se requiera.

b) **Comunicación asíncrona mediante el API de threads de Spring Framework:** Otra opción es comunicarnos mediante el protocola HTTP usando threads, Spring Framework proporciona en su core un API robusta para la implementación de trabajos asíncronos, dicha configuración se debe realizar en el sender microservice y además se requiere exponer servicios para comunicación mediante HTTP en el receive microservice, a continuación se desarrollara el código a agregar a los microservicios encargados del envio y recepción de mensajes:

Sender Microservice:

(a) Agregar propiedades para el manejo de threads y la url de conexión para el consumo de un servicio rest en el archivo **.yml** del microservicio alojado en el servidor de configuraciones o en ***bootstap.yml***:

```
1. spring:
2.   async:
3.     size:
4.       core: ${your-size-core}
5.       max: ${your-size-max}
6.     queue:
7.       capacity: ${your-queue-capacity}
```

```
8.      name-prefix: ${your-name-prefix}
9.
10. rest:
11.   url: ${your-rest-url}
```

(b) Agregar clase de configuración ***RestConfiguration.java***, la cual crea los bean necesarios para la operaciones de comunicación mediante clientes rest:

```
1.  package com.valid.sender.configuration;
2.
3.  import org.springframework.boot.web.client.RestTemplateBuilder;
4.  import org.springframework.context.annotation.Bean;
5.  import org.springframework.context.annotation.Configuration;
6.  import org.springframework.web.client.RestTemplate;
7.
8.  @Configuration
9.  public class RestConfiguration {
10.
11.    @Bean
12.    public RestTemplate restTemplate(RestTemplateBuilder builder) {
13.       return builder.build();
14.    }
15. }
```

(c) Agregar clase handler ***AsyncExceptionHandler.java***, la cual es la encargada de manejar los errores que ocurran en threads asincronos:

```
2)
3)  package com.valid.sender.handler;
4)
5)  import java.lang.reflect.Method;
6)
7)  import org.slf4j.Logger;
8)  import org.slf4j.LoggerFactory;
9)  import org.springframework.aop.interceptor.AsyncUncaughtExceptionHandler;
10)
11) public class AsyncExceptionHandler implements
12)    AsyncUncaughtExceptionHandler {
13)
14)    private static final Logger LOGGER = LoggerFactory
15)       .getLogger(AsyncExceptionHandler.class);
16)
17)    @Override
18)    public void handleUncaughtException(Throwable ex,
```

```
19)         Method method, Object... params) {
20)         LOGGER.error("error -> Method: {}, Cause: {}", method.getName(), ex);
21)     }
22) }
```

(d) Agregar clase de configuración ***AsyncConfiguration.java***, donde agregaremos las caracteristicas para el manejo de threads como son el tamaño de pool de threads (corePoolSize), tamaño de cola de threads (queueCapacity) y el tamaño máximo del pool de threads (maxPoolSize) que se activara cuando la cola de threads alcance la queueCapacity, además es posible agregarle un prefix name para los threads necesarios para la operaciones de comunicación mediante clientes rest:

```
1.  package com.valid.sender.configuration;
2.
3.  import java.util.concurrent.Executor;
4.
5.  import org.springframework.aop.interceptor.AsyncUncaughtExceptionHandler;
6.  import org.springframework.beans.factory.annotation.Value;
7.  import org.springframework.context.annotation.Bean;
8.  import org.springframework.context.annotation.Configuration;
9.  import org.springframework.scheduling.annotation.AsyncConfigurer;
10. import org.springframework.scheduling.annotation.EnableAsync;
11. import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;
12.
13. import com.valid.sender.handler.AsyncExceptionHandler;
14.
15. @EnableAsync
16. @Configuration
17. public class AsyncConfiguration implements AsyncConfigurer {
18.
19.     @Value(
20.             value = "spring.async.size.core")
21.     private int corePoolSize;
22.
23.     @Value(
24.             value = "spring.async.size.max")
25.     private int maxPoolSize;
26.
27.     @Value(
28.             value = "spring.async.size.queue.capacity")
29.     private int queueCapacity;
30.
31.     @Value(
32.             value = "spring.async.name-prefix ")
```

```
33.     private String namePrefix;
34.
35.     @Override
36.     @Bean(
37.             name = {
38.                 "restAsyncExecutor"
39.             })
40.     public Executor getAsyncExecutor() {
41.         ThreadPoolTaskExecutor executor =
42.             new ThreadPoolTaskExecutor();
43.         executor.setThreadNamePrefix(namePrefix);
44.         executor.setCorePoolSize(corePoolSize);
45.         executor.setQueueCapacity(queueCapacity);
46.         executor.setMaxPoolSize(maxPoolSize);
47.         return executor;
48.     }
49.
50.     @Override
51.     public AsyncUncaughtExceptionHandler
52.             getAsyncUncaughtExceptionHandler() {
53.         return new AsyncExceptionHandler();
54.     }
55. }
```

(e) Crear interface ***SenderService.java***, la cual tiene el contrato del método send:

```
1.  package com.valid.sender.service.inter;
2.
3.  import java.util.concurrent.CompletableFuture;
4.
5.  public interface SenderService {
6.
7.      public CompletableFuture<Object> send(Object message);
8.  }
```

(d) Crear implementación ***SenderServiceImp.java***, que aloja la lógica para envió de mensajes:

```
1.  package com.valid.sender.service.imple;
2.
3.  import java.util.concurrent.CompletableFuture;
4.
5.  import org.slf4j.Logger;
6.  import org.slf4j.LoggerFactory;
```

```
7.  import org.springframework.beans.factory.annotation.Autowired;
8.  import org.springframework.beans.factory.annotation.Value;
9.  import org.springframework.scheduling.annotation.Async;
10. import org.springframework.stereotype.Service;
11. import org.springframework.web.client.RestTemplate;
12.
13. import com.valid.sender.service.inter.SenderService;
14.
15. @Service
16. public class SenderServiceImp implements SenderService {
17.
18.     private static final Logger LOGGER = LoggerFactory
19.         .getLogger(SenderServiceImp.class);
20.
21.     @Value(
22.             value = "${rest.url}")
23.     private String restUrl;
24.
25.     @Autowired
26.     private RestTemplate restTemplate;
27.
28.     @Override
29.     @Async(
30.             value = "restAsyncExecutor")
31.     public CompletableFuture<Object> send(Object message) {
32.         Object response = restTemplate.getForObject(restUrl, Object.class);
33.         LOGGER.info("response -> {}", response);
34.         return CompletableFuture.completedFuture(response);
35.     }
36. }
```

(d) Implementar el ***SenderService.java*** utilizando la inyección de dependencias en los microservicios que se requiera:

Se brindaron dos soluciones para soporte de comunicación asíncrona entre microservicios, una utilizando un intermediario de mensajes como RabbitMQ y la otra utilizando Threads de java.

6.3. Solución para Fallbacks para Errores

El control de errores es un aspecto muy importante para brindar fiabilidad y garantizar que operaciones de las cuales no tenemos control debido a que requiere integración con sistemas externos no propaguen errores en tiempo de ejecución. Para solucionar ese tipo de inconvenientes se plantea una solución

usando la librería Hystrix de Netflix que nos brinda una solución basada en el patrón Circuit Breaker, la implementación será para el caso de comunicación entre microservicios usando el protocolo HTTP, para la que tendremos el siguiente enfoque:

a) Comunicación entre microservicios usando el protocolo HTTP con el cliente RestTemplate:

(a) Agregar dependencia para Hystrix en el archivo ***build.gradle***:

compile('org.springframework.cloud:spring-cloud-starter-netflix-hystrix')

(b) Agregar propiedades para el manejo de threads y la url de conexión para el consumo de un servicio rest en el archivo **.yml** del microservicio alojado en el servidor de configuraciones o en ***bootstap.yml***:

```
rest:
  url: ${your-rest-url}
```

(c) Agregar clase de configuración ***RestConfiguration.java***, la cual crea los bean necesarios para la operaciones de comunicación mediante clientes rest y habilitar Hystrix:

```
package com.valid.sender.configuration;

import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.cloud.netflix.hystrix.EnableHystrix;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
@ EnableHystrix
public class RestConfiguration {

    @Bean
    @ LoadBalanced
    public RestTemplate restTemplate(RestTemplateBuilder builder) {
        return builder.build();
    }
}
```

(d) Crear interface ***SenderService.java***, la cual tiene el contrato del método send:

```java
package com.valid.sender.service.inter;

import java.util.concurrent.CompletableFuture;

public interface SenderService {

    public Object send(Object message);
}
```

(e) Crear implementación ***SenderServiceImp.java***, que aloja la lógica para envió de mensajes:

```java
package com.valid.sender.service.imple;

import java.util.concurrent.CompletableFuture;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.scheduling.annotation.Async;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

import com.valid.sender.service.inter.SenderService;
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import com.netflix.hystrix.contrib.javanica.annotation.HystrixProperty;

@Service
public class SenderServiceImp implements SenderService {

    private static final Logger LOGGER = LoggerFactory
        .getLogger(SenderServiceImp.class);

    @Value(
            value = "${rest.url}")
    private String restUrl;

    @Autowired
    private RestTemplate restTemplate;

    @Override
    @HystrixCommand(
            fallbackMethod = "sendFallback",
            commandKey = "send-fallback",
            commandProperties = {
                @HystrixProperty(
```

```
36.                 name = "execution.isolation.thread.timeoutInMilliseconds",
37.                 value = "500"),
38.             @HystrixProperty(
39.                 name = "circuitBreaker.requestVolumeThreshold",
40.                 value = "10"),
41.             @HystrixProperty(
42.                 name = "circuitBreaker.errorThresholdPercentage",
43.                 value = "30"),
44.             @HystrixProperty(
45.                 name = "circuitBreaker.sleepWindowInMilliseconds",
46.                 value = "10000"),
47.             @HystrixProperty(
48.                 name = "metrics.rollingStats.timeInMilliseconds",
49.                 value = "10000")
50.         })
51.   public Object send(Object request) {
52.     Object response = restTemplate.postForObject(restUrl,
53.         request, Object.class);
54.     LOGGER.info("response -> {}", response);
55.     return response;
56.   }
57.
58.   public Object sendFallback(Object request) {
59.     Object response = new Object();
60.     LOGGER.info("fallback -> {}", response);
61.     return response;
62.   }
63. }
```

(f) Implementar el ReceiverService.java utilizando la inyección de dependencias en los microservicios que se requiera.

6.4. Solución con Motor de Búsqueda ElasticSearch

En algunos casos se necesitar obtener información de manera rápida y siguiendo patrones de búsqueda, por lo que los motores de búsqueda toman una alta relevancia en la medida que manejamos altos volúmenes de datos y requerimos procesar búsquedas, ElasticSearch es una de las tecnologías que cumple con estas condiciones, por lo que se brindara una solución utilizando este motor:

a) Integrar ElasticSearch como motor de búsqueda en un microservicio con Spring Framework:

(a) Agregar dependencia para cliente Jest en el archivo ***build.gradle***:

compile('io.searchbox:jest:5.3.3')

(b) Agregar propiedades para conexión y procesamientos con ElasticSearch en el archivo **.yml** del microservicio alojado en el servidor de configuraciones o en ***bootstap.yml***:

```
elasticsearch:
  cluster-nodes: ${your-es-url}
  index: ${your-index}
  type: ${your-type}
  multi-threaded: true
  default-max-connection: ${your-default-max-connection}
  max-connection: ${your-max-connection}
  time-out:
    read: ${your-timeout-read}
    connection: ${your-timeout-connection}
```

(c) Agregar clase de configuración ***ElasticsearchConfiguration.java***, donde agregaremos las características para nuestro cliente:

```
package com.valid.search.configuration;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import io.searchbox.client.JestClient;
import io.searchbox.client.JestClientFactory;
import io.searchbox.client.config.HttpClientConfig;

@Configuration
public class ElasticsearchConfiguration {

	@Value(
		value = "${elasticsearch.cluster-nodes}")
	private String clusterNode;

	@Value(
		value = "${elasticsearch.multi-threaded:true}")
	private boolean multiThreaded;

	@Value(
		value = "${elasticsearch.default-max-connection:2}")
	private int defaultMaxConnection;

	@Value(
```

```
27.             value = "${elasticsearch.max-connection}")
28.     private int maxConnection;
29.
30.     @Value(
31.             value = "${elasticsearch.time-out.read}")
32.     private int readTimeout;
33.
34.     @Value(
35.             value = "${elasticsearch.time-out.connection}")
36.     private int connectionTimeout;
37.
38.     @Bean
39.     public HttpClientConfig httpClientConfig() {
40.             return new HttpClientConfig.Builder(clusterNode)
41.                             .multiThreaded(multiThreaded)
42.                             .defaultMaxTotalConnectionPerRoute(
43.                                     defaultMaxConnection)
44.                             .maxTotalConnection(maxConnection)
45.                             .connTimeout(connectionTimeout)
46.                             .readTimeout(readTimeout).build();
47.     }
48.
49.     @Bean
50.     public JestClientFactory jestClientFactory(HttpClientConfig config) {
51.             JestClientFactory factory = new JestClientFactory();
52.             factory.setHttpClientConfig(config);
53.             return factory;
54.     }
55.
56.     @Bean
57.     public JestClient jestClient(JestClientFactory factory) {
58.             return factory.getObject();
59.     }
60. }
```

(d) Agregar clase ***Util.java*** con la lógica genérica para el procesamiento y transformación de los datos en objetos java:

```
1.  package com.valid.search.util;
2.
3.  import java.io.IOException;
4.  import java.util.ArrayList;
5.  import java.util.List;
6.
7.  import org.slf4j.Logger;
8.  import org.slf4j.LoggerFactory;
9.
```

```
10. import io.searchbox.client.JestClient;
11. import io.searchbox.core.Search;
12. import io.searchbox.core.SearchResult;
13.
14. public class Util {
15.
16.       private static final Logger LOGGER =
    LoggerFactory.getLogger(Util.class);
17.
18.       private Util() {}
19.
20.       public static String queryReplace(String query, Object... values) {
21.             return String.format(query, values);
22.      }
23.
24.       public static <T> List<T> queryResult(JestClient client, String query,
25.                   String index, String type, Class<T> clazz) {
26.             LOGGER.info("query -> {}", query);
27.             List<T> resultQuery = null;
28.             try {
29.                   Search search = new
30.                         Search.Builder(query).addIndex(index)
31.                               .addType(type).build();
32.                   SearchResult result = client.execute(search);
33.                   resultQuery = resultToList(result, clazz);
34.             } catch (IOException e) {
35.                   resultQuery = new ArrayList<>();
36.             }
37.             return resultQuery;
38.       }
39.
40.       @SuppressWarnings(
41.             value = {
42.                "deprecation"})
43.       public static <T> List<T> resultToList(SearchResult result,
44.             Class<T> clazz) {
45.             return result.getSourceAsObjectList(clazz);
46.       }
47. }
```

(e) Crear interface ***SearchRepository.java***, la cual tiene el contrato del método findById:

```
1.  package com.valid.search.repository.inter;
2.
3.  import java.util.List;
4.
```

```
5.  public interface SearchRepository {
6.
7.      public List<Object> findById(String id);
8.  }
```

(f) Crear implementación SearchRepository.java, con la implementación para las consultas en el motor de búsqueda ElasticSearch:

```
1.  package com.valid.search.repository.imple;
2.
3.  import java.io.IOException;
4.  import java.util.ArrayList;
5.  import java.util.List;
6.
7.  import org.springframework.beans.factory.annotation.Autowired;
8.  import org.springframework.beans.factory.annotation.Value;
9.  import org.springframework.stereotype.Repository;
10.
11. import com.valid.search.repository.inter.SearchRepository;
12. import com.valid.search.util.Util;
13.
14. import io.searchbox.client.JestClient;
15.
16. @Repository
17. public class SearchRepositoryImp implements SearchRepository {
18.
19.     private static final String QUERY_FIND_BY_ ID = "{\"_source\": "
20.                             + {\"includes\": [\"id\"]}," \"query\": {\"bool\":
21.                             + {\"filter\": {\"term\": {\" id\": %s}}}}}";
22.
23.     @Value(
24.         value = "${elasticsearch.index}")
25.     private String index;
26.
27.     @Value(
28.         value = "${elasticsearch.type}")
29.     private String type;
30.
31.     @Autowired
32.     private JestClient jestClient;
33.
34.     @Override
35.     public List<Object> findById(String id) {
36.         String query = Util
37.             .queryReplace(QUERY_FIND_BY_ID,
38.                 id);
```

```
39.             return Util.queryResult(jestClient, query,
40.                     index, type, Object.class);
41.     }
42. }
```

(g) Implementar el SearchRepository.java utilizando la inyección de dependencias en los microservicios que se requiera.

7. Cronograma

Nombre de tarea	Duración	Comienzo	Fin
Proyecto	24 días	lun 04/06/18	jue 05/07/18
Análisis	6 días	lun 04/06/18	mar 12/06/18
Análisis de Ambiente	1 día	lun 04/06/18	mar 05/06/18
Identificar Necesidades	2 días	mar 05/06/18	jue 07/06/18
Requisitos Funcionales	2 días	jue 07/06/18	lun 11/06/18
Requisitos no Funcionales	1 día	lun 11/06/18	mar 12/06/18
Formulación	5 días	mar 12/06/18	mar 19/06/18
Propuesta de Tecnología	5 días	mar 12/06/18	mar 19/06/18
Diseño	7 días	mar 19/06/18	jue 28/06/18
Diseño de Solución para Comunicación Asíncrona	5 días	mar 19/06/18	mar 26/06/18
Diseño de Solución con Broker de Mensajes	4 días	mar 19/06/18	lun 25/06/18
Diseño de Solución con el API de Threads de Spring	1 día	lun 25/06/18	mar 26/06/18
Diseño de Solución para Fallos en Cadena	1 día	mar 26/06/18	mié 27/06/18
Diseño de Solución con Motor de Búsqueda	1 día	mié 27/06/18	jue 28/06/18
Desarrollo y Pruebas	5 días	jue 28/06/18	jue 05/07/18
Desarrollo	4 días	jue 28/06/18	mié 04/07/18
Desarrollo de Solución para Comunicación Asíncrona	2 días	jue 28/06/18	lun 02/07/18
Desarrollo de Solución con Broker de Mensajes	1 día	jue 28/06/18	vie 29/06/18
Desarrollo de Solución con el API de Threads de Spring	1 día	vie 29/06/18	lun 02/07/18
Desarrollo de Solución para Fallos en Cadena	1 día	lun 02/07/18	mar 03/07/18
Desarrollo de Solución con Motor de Búsqueda	1 día	mar 03/07/18	mié 04/07/18
Prueba	1 día	mié 04/07/18	jue 05/07/18

Fuente:Autor

8. Conclusiones

Se lograron desarrollar un conjunto de módulos que permiten mejorar operaciones enmarcadas en los aspectos tecnológicos, logrando desacoplar diversos componentes, evidenciándose en la implementación de un intermediario de mensajes, con lo cual cada microservicio que envía el mensaje no se interesa por quien es el receptor, además de la mejora de tiempos, pues estos intermediarios manejan protocolos basados en colas y enrutamientos hacia ellas de forma óptima. En el tema relacionado a fallos se brindó una solución que puede ser aplicada a diversas lógicas de negocio, como es el caso de comunicación HTTP con otro microservicio, garantizando que los fallos por no disponibilidad no se propaguen en cascada hacia todos los componentes que participan en el proceso, brindando fiabilidad y disponibilidad, este patrón puede ser implementado para diversas operaciones como fallos en conexiones a bases de datos, fallos en conexión a intermediarios de mensajes, fallos en lectura de disco, etc. La implementación de un motor de búsqueda permitió solucionar algunos inconvenientes relacionados a las consultas de grandes volúmenes de datos basados en patrones de búsqueda desde los motores SQL y la no escalabilidad horizontal de estos, ya que ElasticSearch permite la implementación de clústers, dividiendo nodos en diferentes máquinas y garantizando un balanceo seguro de solicitudes hacia estos nodos, además de proveer una potente API de búsqueda basada en criterios de consulta.

9. Bibliografía

Barrios, D. (2018). Arquitectura de Microservicios. *Tecnología Investigación y Academia, 6*(1), 36-46.

Barrios, D. A. (2018). Arquitectura de microservicios. *TIA, Tecnología, Investigación y Academia, VI*(1), 36-46.

Fowler, M. (2014). *Microservices.*

Gómez, K., Anaya, R., & Cano, A. (2018). Un acercamiento a los Microservicios. *UNACiencia, 10*(19), 116-126.

López, D., & Maya, E. (2107). Arquitectura de Software basada en Microservicios para Desarrollo de Aplicaciones Web.

Mińkowski, P. (2018). Cloud-native development. En *Mastering Spring Cloud* (pág. 9). Birmingham: Packt Publishing.

Mińkowski, P. (2018). Different styles of communication. En *Mastering Spring Cloud* (pág. 123). Birmingham: PacktPub.

Mińkowski, P. (2018). Introduction to Microservices. En *Mastering Spring Cloud* (pág. 7). Birmingham: Packt Publishing.

Mińkowski, P. (2018). Message-Driven Microservices. En *Mastering Spring Cloud* (pág. 248). Birminham: Packt Publishing.

Montesi, F., & Weber, J. (2016). Circuit Breakers, Discovery, and API Gateways in Microservices. *eprint arXiv, 2*.

Sai, M., & Kumar, S. (2013). ElasticSearch An advanced and quick search technique to handle voluminous data. *COMPUSOFT, 2*(6), 171-175.

VALID. (s.f.). *VALID*. Recuperado el 11 de Agosto de 2018, de https://valid.com/es/who-we-are/about-us/

VALID. (s.f.). *VALID*. Recuperado el 11 de Agosto de 2018, de https://valid.com/es/who-we-are/history/

VALID. (s.f.). *VALID*. Recuperado el 11 de Agosto de 2018, de https://valid.com/es/who-we-are/sustainability-social-responsability/