Universidad de la República
Facultad de Ingeniería
Instituto de Computación
Uruguay

Implementación de Patrones de Microservicios

Diego Verdier

Gonzalo Rodríguez

Proyecto de Grado
Ingeniería en Computación
Universidad de la República

Montevideo, Uruguay, Setiembre de 2020

Supervisor: Dra. Ing. Laura González
Co-Supervisor: Ing. Sebastián Vergara

Resumen

En los últimos años la arquitectura de microservicios se ha posicionado en un lugar importante en las decisiones de diseño de los nuevos sistemas de software, así como también en lo que respecta a la modernización de sistemas monolíticos existentes y/o legados. Sin embargo, aún no se han estandarizado por completo buenas prácticas que se ajusten a todas las realidades de aplicación, por lo que existen barreras para la adopción de este enfoque.

Un aspecto que facilita la implementación de este estilo arquitectónico es el concepto de patrón de microservicios, que provee de soluciones generales que son aplicadas a problemáticas que usualmente surgen al adoptar esta arquitectura. Sin embargo, aún existen algunas sin resolver. Por ejemplo, contar con un conjunto de patrones bien definidos, la interacción entre múltiples patrones, la dependencia o compatibilidad que se genera con la aplicación de uno u otro y la selección de las tecnologías para la implementación.

Para abordar estas problemáticas, este proyecto propone una solución que asiste en la implementación de la arquitectura de microservicios guiada por patrones.

En primer lugar, se realizó un análisis con el objetivo de identificar las principales funcionalidades que una solución de este estilo debería proveer. Dicho análisis constó de un relevamiento de los patrones utilizados actualmente tanto en la industria como en la academia, junto con el diseño de un modelo conceptual en el cual se instanciaron.

En segundo lugar, se presentó una propuesta de solución que contempla las funcionalidades antes identificadas. La solución busca asistir en la toma de decisiones antes, durante y después del diseño de una aplicación utilizando microservicios. Se plantea una solución que brinda, en base a ciertas características de la realidad concreta, alternativas de diseño de arquitectura mediante la instanciación de patrones de microservicios específicos, conjuntamente con recomendaciones referentes a decisiones tecnológicas y compatibilidad entre los patrones.

Por último, se desarrolló un prototipo que abarca parte del flujo de la solución presentada anteriormente. El prototipo provee de estructuras de código de los patrones, sumado a aplicaciones de ejemplo donde se implementan dichas estructuras.

Palabras clave: arquitectura de microservicios, patrones de microservicios, saga, circuit breaker, implementación de microservicios, plataforma de microservicios.

Tabla de Contenido

1 Introducción

En este documento se presenta el proyecto de grado denominado “Implementación de Patrones de Microservicios”, el cual está enmarcado en las áreas de trabajo del Laboratorio de Integración de Sistemas del Instituto de Computación.

En este capítulo se presenta la introducción del proyecto incluyendo el contexto y motivación, los objetivos, los aportes que este realiza y finalmente se expone una breve reseña sobre la organización del resto del documento.

1.1 Contexto y motivación

En los últimos años la arquitectura de microservicios se ha posicionado en un lugar importante en las decisiones de diseño de los nuevos sistemas de software, así como también en lo que respecta a la modernización de sistemas monolíticos existentes y/o legados [88]. La popularidad que esta arquitectura viene adoptando en el último tiempo ha crecido rápidamente producto de algunos factores como [89]:

- La gran cantidad de ventajas que teóricamente aporta este enfoque.
- Lo alineado que estas ventajas están con gran parte de los desafíos a los que se enfrentan los sistemas informáticos actuales.
- Las necesidades dinámicas y volátiles que requieren estos sistemas.

Por este motivo, muchos actores de gran importancia en la industria tecnológica decidieron adoptar esta arquitectura para implementar sus sistemas. Amazon, Microsoft, IBM, RedHat, Google y Netflix son algunos de los emblemas del mundo de los microservicios y, en parte, es gracias a sus aportes que este paradigma sigue creciendo. En particular, varias empresas han contribuido a la comunidad Open Source con el desarrollo de diferentes tecnologías que facilitan el uso y ayudan a la incursión en el mundo de los microservicios.

Por otra parte, dado que este “nuevo” enfoque propone un cambio importante en la forma en que los sistemas se implementan, ha sido necesario un arduo trabajo por parte tanto de la industria como de la academia para comprender los conceptos propuestos. Algunos de los nuevos desafíos que se introducen son, identificar en qué escenarios es aplicable esta arquitectura en pos de obtener los mayores beneficios posibles y consolidar la forma de aplicación de la arquitectura para tratar de disminuir la complejidad en su adopción. Al mismo tiempo, el contar con soluciones a problemas recurrentes es un factor que viene siendo cada vez más importante, ya que los contextos en los cuales es aplicada esta arquitectura son cada vez más diversos.

Sin embargo, si bien se ha dedicado un esfuerzo considerable aún no se han estandarizado por completo buenas prácticas que se ajusten a todas las realidades de aplicación. Lo cual conjuntamente con otros factores, hacen que exista una barrera importante en lo que respecta a la complejidad que conlleva adoptar este enfoque.

Un aspecto que facilita la implementación de este estilo arquitectónico es el concepto de patrón de microservicios. Concretamente, los patrones de microservicios proveen de soluciones generales que son aplicadas a problemáticas que usualmente surgen al adoptar la arquitectura de microservicios. En particular, desde un tiempo a esta parte los expertos en el área vienen dedicando tiempo y esfuerzo en analizar y proponer patrones específicos de microservicios que sirvan como método de resolución a estos problemas a los que se enfrentan las organizaciones, arquitectos de software y desarrolladores. Específicamente, producto de ese trabajo se han identificado áreas en las cuales se requieren de soluciones estandarizadas (patrones) tales como la consistencia, donde se destaca el patrón Saga, o la tolerancia a fallas, donde se identifica el patrón Circuit Breaker.

Sin embargo, si bien autores reconocidos como Chris Richardson, Sam Newman o Martin Fowler han conseguido proponer una serie de patrones de microservicios, aún existen problemáticas sobre las cuales resta dedicar mucho esfuerzo para que estos patrones sean aplicados de forma práctica y sencilla. Por ejemplo, la interacción entre múltiples patrones, la dependencia o compatibilidad que se genera con la aplicación de uno u otro, la elección de las tecnologías de implementación y cómo estas afectan la compatibilidad de una implementación con otra, la integración entre la soluciones propuestas por los patrones y la lógica específica del negocio.

Por los motivos descritos anteriormente, resulta de sumo interés contar con una plataforma que asista en la implementación de la arquitectura de microservicios guiada por patrones.

1.2 Objetivos

El objetivo general del proyecto es proponer y diseñar una solución que asista en la implementación de aplicaciones basadas en microservicios guiada por patrones.

Para cumplir con lo descrito anteriormente, se plantean los siguientes objetivos específicos:

1. Estudiar conocimiento existente en la arquitectura de microservicios y patrones de mi-

croservicios.

2. Analizar las funcionalidades que una plataforma que asista en la implementación de la arquitectura de microservicios basada en patrones debería proveer, así como trabajos relacionados.
3. Proponer una solución que aborde las funcionalidades identificadas, profundizando en un subconjunto de funcionalidades y patrones.
4. Brindar una implementación de referencia para algunos de estos patrones, con tecnologías específicas.
5. Implementar un prototipo que permita validar la solución propuesta.

1.3 Aportes del proyecto

A continuación se destacan los principales aportes de este proyecto:

1. Modelo conceptual en base al relevamiento de patrones de microservicios.
2. Caracterización de los patrones relevados en base al modelo conceptual.
3. Identificación de las funcionalidades que una plataforma que asista en la implementación de la arquitectura de microservicios debería proveer.
4. Propuesta de solución para dicha plataforma.
5. Implementación de referencia para los patrones Saga y Circuit Breaker, utilizando dos stack tecnológicos distintos para cada uno.
6. Construcción de un prototipo que implementa parte de las funcionalidades identificadas y que provee de información de los patrones, resultado del relevamiento realizado.
7. Aplicaciones de ejemplo para cada implementación de referencia desarrollada.

1.4 Organización del documento

El resto del documento se organiza de la siguiente forma:

En el Capítulo 2 se presenta el marco teórico donde se encuentran los principales conceptos que dan contexto y sirven como base para el entendimiento del documento.

En el Capítulo 3 se identifican las principales funcionalidades que una plataforma que asista en la implementación de la arquitectura de microservicios debería tener. Todo esto producto de un relevamiento de diferentes patrones y la caracterización de los mismos en un modelo conceptual propuesto.

En el Capítulo 4 se presenta la solución propuesta.

En el Capitulo 5 se detallan aspectos de implementación tanto del prototipo de la plataforma como de las implementaciones de referencia de los patrones.

Finalmente, en el Capítulo 6 se presentan las conclusiones generales del proyecto así como lineamientos sobre los cuales continuar el estudio del tema a futuro.

2 Marco Teórico

Este capítulo contiene los conceptos necesarios para el correcto entendimiento del resto del documento.

2.1 Arquitectura de microservicios

Si bien no existe una definición estandarizada, diferentes autores definen "Arquitectura de microservicios" de manera muy similar.

Chris Richardson en [1] [2] la define como "un estilo arquitectónico que estructura una aplicación como una colección de servicios". Martin Fowler y James Lewis [3] la describen como: “una manera de diseñar aplicaciones de software en servicios independientes”.

Si bien ellos mismos destacan que no existe una definición precisa de este tipo de arquitectura, aclaran que hay ciertas características de los microservicios que son comunes a todas las definiciones: ser desarrollados en función de las capacidades de negocio, el despliegue automatizado, el manejo descentralizado de los datos , tecnologías y componentes inteligentes pero comunicaciones simples entre los mismos [3].

Con el objetivo de profundizar en su definición se presentan una serie de conceptos asociados que dan contexto y complementan las definiciones antes presentadas.

2.1.1 Definición de microservicio

Según Chris Richardson [1] un microservicio se define como: "un repositorio de código pequeño, autónomo y desplegado independientemente de los demás". Se desarrollan y diseñan en base a una capacidad de negocio concreta y pueden ser vistos como una aplicación independiente del resto de microservicios, aún cuando coexistan para la implementación del sistema completo.

Como características principales de los microservicios se destacan [1] [2] [3]:

- Se desarrollan en base a una capacidad de negocio concreta.
- Deben ser tratados como una “caja negra”, ya que conocer su implementación no es requerido por ningún servicio externo.
- Encargados de una única funcionalidad concreta.
- Exponen dicha funcionalidad mediante uno o más endpoints pero abstrayendo totalmente a agentes externos de la implementación concreta.
- Límites claros entre los distintos servicios, que permite lograr independencia con respecto a la implementación concreta de cada servicio, lo que deriva en menor acoplamiento y mayor cohesión.
- Independientes, al ocultar su implementación y solo comunicarse a través de una interfaz son capaces de ser mantenidos, desplegados, desarrollados y probados de manera independiente.
- Un único equipo reducido de programadores puede escribir y mantener un servicio.
- No es necesario que los servicios compartan el mismo stack tecnológico, las librerías o los frameworks.

2.1.2 Beneficios de los microservicios

Independencia: Cada microservicio se implementa de manera independiente del resto. Este hecho provoca que se puedan implementar en paralelo, se puedan desplegar según sea necesario y el mantenimiento sea más simple. Esto se materializa en mejoras rápidas y continuas de cada funcionalidad.

Equipos autónomos y ágiles: Esta arquitectura permite que los equipos de desarrollo sean autónomos, desacoplados y pequeños. Como consecuencia, cada microservicio es desarrollado por un único equipo, implicando que la velocidad de desarrollo sea considerablemente mayor.

Versatilidad de tecnologías: Los equipos son libres en lo que respecta a la selección de lenguaje, tecnologías, y frameworks, incluida la base de datos que mejor se adapte al servicio y funcionalidad. Cada servicio se puede implementar en el hardware que mejor se adapte a sus requisitos referentes a recursos. Se elimina cualquier compromiso a largo plazo con un stack tecnológico puntual y se logra adoptar nuevas tecnologías con mayor agilidad, provocando que se pueda experimentar con nuevas tecnologías, con un riesgo significativamente menor.

Aislamiento y resiliencia: Cuantos más microservicios integren el sistema, más complejo será y mayor probabilidad de fallas experimentará, es por este motivo, que se debe asegurar que cada uno de los servicios sea tolerante a fallas y pueda recuperarse luego de la ocurrencia de una.

El aislamiento de las posibles fallas de un sistema debe ser considerado desde su diseño, la arquitectura de microservicios ayuda a lograr esto ya que el alcance de cada servicio está bien delimitado.

Escalabilidad: Debido a la independencia de los servicios, es que cada uno puede escalar de manera individual, sea horizontal o verticalmente.

2.1.3 Desventajas de los microservicios

No existe tecnología que no traiga aparejada desventajas o complicaciones, y la arquitectura de microservicios no es la excepción [1].

Descomponer en servicios: No existe un algoritmo concreto y bien definido para descomponer un sistema en servicios. El trabajo de diseño que requiere la etapa de descomposición es extremadamente desafiante ya que un mal diseño podría provocar desaprovechamiento de los beneficios de esta arquitectura.

Los sistemas distribuidos son complejos: Los desarrolladores deben ser capaces de lidiar con la complejidad que requiere un sistema distribuido, se deben implementar mecanismos de comunicación entre los distintos servicios, además de diseñar los servicios de forma tal que sean capaces de manejar fallas parciales y tratar con un servicio remoto ya sea que no esté disponible o que exhiba alta latencia.

Decidir cuándo adoptar este enfoque: Otro problema con el uso de la arquitectura de microservicios es decidir en qué punto durante el ciclo de vida de la aplicación se debe utilizar. Al desarrollar la primera versión de una aplicación, a menudo no se presentan los problemas que esta resuelve, por lo cual el uso de una arquitectura elaborada y distribuida ralentizará el desarrollo. Sin embargo, en etapas más avanzadas, cuando las aplicaciones comienzan a crecer rápidamente y se vuelven más complejas, comienza a tener sentido descomponer funcionalmente la aplicación en un conjunto de microservicios.

Consistencia: Al involucrar una gran cantidad de servicios independientes interactuando entre si para satisfacer un objetivo en común, sumado a la limitante de no poder aplicar los clásicos protocolos de consistencia como el 2PC hacen de la consistencia de los datos una de los grandes desafíos a superar.

Observación: El hecho de tener múltiples sistemas desplegados al mismo tiempo complejizan considerablemente la tarea de observación de fallas y seguimiento de la evolución del sistema como un todo.

Seguridad: Cuantos más servicios independientes coexisten mayor es el desafío de la gestión de la seguridad contemplando que se deben aplicar políticas de seguridad que protejan a todos estos servicios.

2.1.4 Diferencia con aplicaciones monolíticas:

Para lograr un mejor entendimiento sobre los microservicios, es interesante remarcar la diferencia con el enfoque monolítico. El término “monolito” significa compuesto en una sola

pieza, por lo cual en una aplicación monolítica se combinan diferentes componentes en pos de conformar un único programa.

Un monolito es un único repositorio de código, separado generalmente en tres capas [91]:

- Capa de datos: consiste en una base de datos que contiene todos los datos de la aplicación.
- Capa lógica: contiene toda la lógica de negocio de la aplicación.
- Capa de presentación: contiene la información suficiente y necesaria que será la que verá el consumidor. Lo ideal es que no contenga nada de la lógica de negocio.

La gran desventaja de este tipo de arquitectura es que al tener todo el código en un solo componente, se hace muy complicado de mantener [91].

2.2 Conceptos asociados

2.2.1 Computación en la nube

El NIST (National Institute of Standards and Technology) define "computación en la nube" como: "Un modelo para permitir el acceso ubicuo, conveniente y bajo demanda a un conjunto compartido de recursos informáticos configurables en la red (por ejemplo, redes, servidores, almacenamiento, aplicaciones y servicios) que se pueden aprovisionar y liberar rápidamente con un mínimo esfuerzo de gestión o interacción del proveedor de servicios" [16].

Específicamente, dentro de los modelos de despliegue se destacan los conceptos de nube privada, nube comunitaria, nube pública y nube híbrida. Según el NIST, el concepto de nube pública se refiere a que la infraestructura mediante la cual se brindan los diferentes servicios se encuentra enteramente en las instalaciones del proveedor de nube [16].

2.2.2 Manejo de datos en arquitecturas distribuidas

Por su naturaleza de sistema distribuido las aplicaciones construidas en base a la arquitectura de microservicios adoptan los desafíos a los que este tipo de sistemas usualmente se enfrentan. Mas aún, si se tiene en cuenta que en la gran mayoría de las implementaciones de esta arquitectura se maneja una base de datos por servicio.

Por este motivo, surge la necesidad de estipular políticas para la gestión de los datos con el objetivo de lograr consistencia. Producto de esta necesidad, se plantean una serie de interrogantes dentro de las cuales se destacan: qué servicio tiene la propiedad de cada entidad modelada, cómo modelar la realidad y cómo distribuirla entre los servicios que la componen y finalmente cómo manejar las solicitudes que involucren datos que son propiedad de más de un servicio.

Una alternativa muy aplicada son las transacciones distribuidas, las cuales cumplen con las propiedades ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad) y simplifican el trabajo de los administradores de sistemas al proporcionar la ilusión de que cada transacción

tiene acceso exclusivo a los datos.

Sin embargo, en el contexto de los microservicios, este enfoque únicamente se puede aplicar en aquellas operaciones que tienen alcance acotado a un solo servicio. Esto se debe a que si se aplicara a operaciones que actualizan datos de múltiples servicios se sacrificaría una de las premisas más importantes que esta arquitectura pregona que es la disponibilidad.

Como consecuencia, la implementación del clásico 2PC (two phase commit) [49] que es el protocolo por excelencia basado en el paradigma ACID es inviable en este tipo de arquitectura. Esto se debe a que, además de lo antes mencionado, las bases de datos no relacionales y brokers de mensajería modernos no lo soportan.

Por este motivo, toma relevancia el concepto de Consistencia Eventual [10] como solución aplicable al mundo de los microservicios. En este modelo la consistencia de datos no se alcanza de forma inmediata en el sistema para todos los observadores al mismo tiempo, sino que existe un intervalo de tiempo no específico en el cual potencialmente el sistema puede estar en estado inconsistente. A dicho intervalo se lo denomina ventana de inconsistencia.

Se propone que exista propagación en el tiempo de los cambios del sistema y se definan políticas de resolución a conflictos en caso de que los hubiese (situación que ocurre con mucha menos frecuencia de lo que pudiera suponerse). Lo único que se garantiza es que los datos eventualmente serán consistentes a lo largo del tiempo, y esto es lo importante [10].

Este es un modelo que tiene gran importancia en los sistemas en donde se prioriza la baja latencia, alta disponibilidad, alta escalabilidad y alta tolerancia a fallos por sobre otros aspectos. Usualmente la consistencia eventual es un típico “trade-off” de ingeniería, ya que plantea renunciar a algo que se pensaba que era más importante para el sistema de lo que realmente es (consistencia), a cambio de tener otras cualidades que realmente sí son necesarias (disponibilidad) [10].

2.2.3 Contenedores

En esta sección se presentan los conceptos básicos asociados a la forma de despliegue más habitual en una arquitectura de microservicios.

El despliegue en contenedores es considerado como el mejor enfoque para desplegar aplicaciones con microservicios ya que sus propiedades potencian aún más las ventajas que este estilo arquitectónico brinda [1].

Definición

Docker Inc, la empresa que desarrolló el proyecto Docker (tecnología que permite la creación y el uso de contenedores Linux) define a los contenedores como: “Una unidad de software que permite empaquetar el código y todas las dependencias de manera que la aplicación pueda ejecutar de manera rápida y confiable en cualquier ambiente. A este “paquete” generado se le denomina imagen y tiene la particularidad de ser ligero, independiente e incluye todo lo necesario para poder ejecutar una aplicación: código, herramientas del sistema, librerías, configuración y ambiente de ejecución” [11] [50].

A su vez, se almacenan aplicaciones con el fin de que quedan desacopladas del sistema operativo en el cual ejecuta, esto permite que una aplicación alojada en un contenedor pueda ser transportada y ejecutada independientemente del Sistema Operativo. Para lograrlo, los contenedores suelen contener aplicaciones conjuntamente con todas sus dependencias (como librerías y frameworks), por lo cual en definitiva un contenedor está compuesto por [50]:

- La aplicación.
- Todas sus dependencias, librerías y demás archivos binarios.
- Archivos de configuración que son necesarios para ejecutarlo.

Comparación con las máquinas virtuales

Una máquina virtual es un entorno basado en software diseñado que proporciona la misma funcionalidad que una computadora física. De la misma manera que los equipos físicos, ejecutan aplicaciones y un sistema operativo. La diferencia, es que las máquinas virtuales son archivos de computadora que se ejecutan en una computadora física y se comportan como tal [51].

Como primera diferencia entre máquinas virtuales y contenedores a la hora de correr es el tamaño del paquete que se debe ejecutar. En máquinas virtuales se empaqueta además de la aplicación un sistema operativo. Esto trae como consecuencia, que a medida que se incrementa el número de microservicios en ejecución en un solo servidor se tendrá un mayor desperdicio de recursos que con contenedores: por cada microservicio ejecutándose en máquinas virtuales habrá un sistema operativo corriendo y un hipervisor (agente entre el entorno de las máquinas virtuales y el hardware donde se ejecuta) [51].

Los contenedores son mucho más livianos y consumen menos recursos que las máquinas virtuales, lo cual, dentro de otras ventajas produce que sea más rápido iniciar una aplicación en un contenedor que en un máquina virtual [50].

Ventajas:

Los contenedores además de proporcionar muchas de las ventajas de las máquinas virtuales, cuentan con características que los hacen destacar por sobre estas.

Por ejemplo, son más ligeros en términos de recursos de procesamiento y automatizan tareas repetitivas de configuración del entorno de desarrollo, por lo cual es posible que los desarrolladores se centren únicamente en el código. En promedio, los desarrolladores que utilizan Docker tienen una productividad, en cuanto a velocidad de desarrollo medido en entrega de software, siete veces mayor que los que no lo utilizan [11].

2.2.4 Plataformas de orquestación

Un concepto que se desprende del anterior presentado, es el de plataforma de orquestación.

Una plataforma de orquestación permite automatizar la implementación, el escalado y la

administración de aplicaciones en contenedores. Generalmente agrupan los contenedores que conforman una aplicación en unidades lógicas para una fácil administración y descubrimiento. Proveen un mayor nivel de abstracción eliminando así algunos de los desafíos que, si no fuera por el uso de las mismas, los desarrolladores deberían enfrentar [92].

2.3 Comunicación entre microservicios

En esta sección se presentan los dos modelos de comunicación aplicables entre servicios para la cooperación de los mismos.

Como se ha mencionado previamente, resolver funcionalidades que involucren a más de un servicio es desafiante, más aún considerando que cada uno suele tener su propia base de datos y que potencialmente para implementar una funcionalidad se requiere acceder, modificar o agregar datos a múltiples servicios.

Existen dos estrategias para implementar la comunicación entre servicios [1]:

- Coreografía: Se delega la responsabilidad de la toma de decisiones y la secuenciación. Se comunican principalmente mediante el intercambio de eventos.
- Orquestación: Se centraliza la lógica de coordinación en un orquestador (una especie de maestro de ceremonia). El orquestador envía mensajes a los participantes indicando cuál es la operación que tienen que invocar.

No existe estandarización en cuanto al uso de estos dos enfoques. Por un lado, Sam Newman en su libro “Building Microservices” [6], destaca que si se utiliza orquestación, el sistema tiene un nivel de acoplamiento más bajo y mayor flexibilidad ante cambios. Sin embargo, se requiere trabajo extra para monitorear y controlar todos los procesos. Al mismo tiempo, se destaca que los sistemas se vuelven mas frágiles ya que se introduce un único punto de falla. Es por estos motivos, que dicho autor recomienda el uso de coreografía.

Por otra parte, Chris Richardson en [1], recomienda la utilización de coreografía para interacciones sencillas y orquestación para las comunicaciones que requirieren mayor complejidad.

2.4 Patrones de microservicios

En esta sección se introduce el concepto de patrón de microservicios y su utilidad, para ser profundizado más adelante en el documento.

Los patrones de diseño de software en general describen problemas que se presentan repetidamente en la fase de diseño del software. La aplicación de patrones aportan maneras estandarizadas, probadas y validadas de resolver problemas comunes. Por este motivo, aplicando patrones se pueden estandarizar soluciones a determinados problemas puntuales [90].

En lo que refiere a la arquitectura de microservicios el concepto es el mismo. Esto se debe a que como todo enfoque arquitectónico, posee ciertas desventajas que se presentan comúnmente. Por este motivo, es de sumo interés contar con soluciones del estilo que los patrones

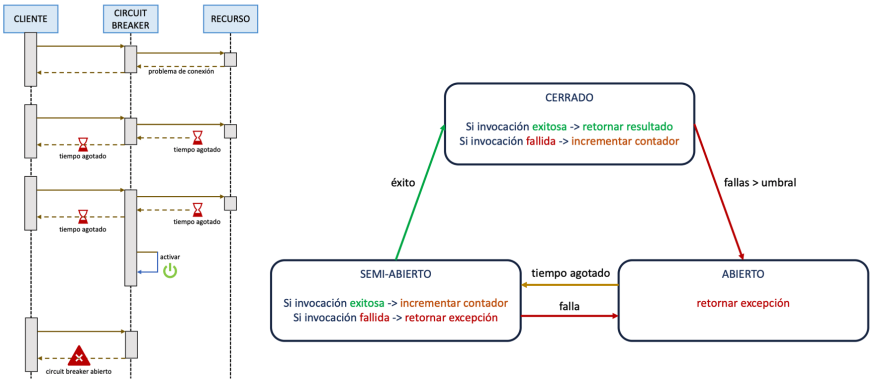

Figure 2.1: Circuit Breaker

aportan.

Al mismo tiempo, referentes de este tema, vienen trabajando en la definición de un lenguaje de patrones de arquitectura de microservicios, el cual encapsule diferentes patrones para abarcar las problemáticas que se presentan al aplicar esta arquitectura [1] [2].

Un patrón de microservicios es compuesto por una serie de conceptos los cuales tienen como fin atacar un área puntual de la aplicación.

A modo de ejemplo se presentan los patrones Circuit Breaker y Saga, que son dos de de los patrones que comúnmente se consideran por autores de la industria y academia (p. ej. Chris Richardson [1] [34], Martin Fowler [3] [33], Sam Newman [6], IBM [61], RedHat [62] [56] y Microsoft [87]).

2.4.1 Circuit breaker:

Este patrón, presentado en la figura 2.1, plantea llevar un registro de las invocaciones exitosas y fallidas entre dos servicios. Si el ratio de invocaciones fallidas de un servicio a otro supera un umbral, el circuito “se abre”. Esto significa que todas las invocaciones futuras fallarán ante la vista del cliente, pero en realidad, la invocación no se hará, evitando sobrecargar el servicio. Después de un período de tiempo previamente definido, el circuito pasa a un estado intermedio, donde ante las primeras nuevas invocaciones si el circuito sigue fallando vuelve al estado abierto y en caso de éxito se cierra y pasa a realizar todas las invocaciones con normalidad.

2.4.2 Saga:

Surge como solución al problema del manejo de las transacciones distribuidas. Es un mecanismo para mantener la consistencia de datos en una arquitectura de microservicio sin la necesidad de utilizar transacciones distribuidas. Más precisamente, una saga es una secuencia

de transacciones locales donde cada una de esas transacciones cumple con las propiedades ACID (atomicidad, consistencia, aislamiento y durabilidad). Por transacciones locales entiéndase aquellas transacciones que solo manipulan datos de un servicio concreto [56] [1].

Es necesario definir una saga para cada transacción que quiera actualizar datos de más de un servicio, y se coordina usando mensajería asíncrona. Si ocurre un error en alguno de los pasos intermedios (transacciones locales intermedias) la saga se recupera mediante transacciones compensatorias que se ejecutan en orden inverso a la secuencia original.

Existen 4 tipos de transacciones [1]:

- Read only spets: No necesitan transacciones compensatorias.
- Compensables transactions: Son seguidas por transacciones que pueden fallar.
- Pivot transaction: Son seguidas por transacciones que no pueden fallar.
- Retriable transaction: siempre tienen éxito

3 Análisis

El objetivo principal de esta sección es el identificar las funcionalidades y características que una plataforma que asista en la implementación de aplicaciones basadas en microservicios guiada por patrones debería tener.

Para lograr dicho objetivo, se plantean una serie de pasos realizados durante el análisis.

En primera instancia se relevaron distintos patrones existentes, y como resultado se identificaron conceptos importantes que rodean a los patrones.

Sumado a esto, y dada la falta de estandarización actual, en la medida que se fue profundizando en el entendimiento de estos conceptos se identificó la necesidad de contar con un modelo conceptual el cual permitiera instanciar a los patrones. Por consecuencia, a través del diseño este modelo se logro tener unificados y uniformizados los conceptos que engloban a cada patrón facilitando el entendimiento de los mismos no solo en lo referente a la solución específica que proveen, si no que también, en la interacción con el ecosistema que lo rodea.

Finalmente, basado en el proceso antes mencionado, se identificaron las principales características que una plataforma de este estilo debería proveer y se contrastó el resultado con trabajos existentes con el fin de identificar aquellos aspectos que están siendo atacados y cuáles no.

3.1 Relevamiento de patrones

El proceso de relevamiento de patrones constó con 3 etapas bien definidas.

- Etapa 1: Identificación de fuentes de información.
- Etapa 2: Estudio de las fuentes para la identificación de patrones y categorías.
- Etapa 3: Refinamiento de la lista de patrones identificados.

3.1.1 Identificación de fuentes de información

El objetivo de la primer etapa fue identificar una cantidad considerable de fuentes certeras mediante las cuales poder extraer información fidedigna sobre patrones de microservicios.

Se tuvieron en cuenta diferentes formas de búsqueda. Por un lado, búsquedas informales a través de Google, las cuales arrojaron como resultado varios autores de referencia que a posteriori terminaron siendo fundamentales para el proyecto.

Concretamente, fue el caso de Chris Richardson el cual posteriormente fue el pilar fundamental de información para este trabajo a través de su página web [2] y uno de sus libros [1]. Además, fueron considerados trabajos de Sam Newman como “Building Microservices” [6], "What Are Microservices" [7] y su blog oficial [5], y de Martin Fowler por intermedio de su blog [3].

En lo que respecta a la industria, se consideraron buenas prácticas propuestas por Netflix [92], sumando a lineamientos dados por Microsoft [9] e IBM.

Finalmente, se destaca que los aportes de Chris Richardson fueron fundamentales, ya que su trabajo en patrones de microservicios es el más profundo de los que se estudiaron. Incluso, se llegó a interactuar directamente con el autor en búsqueda de recomendaciones sobre la implementación.

En una segunda instancia, se realizaron búsquedas en portales académicos tales como Timbó [94] y Google Académico [95] para ampliar el espectro de búsqueda en pos de expandir la cantidad de fuentes de información.

Un trabajo destacado, y el cual se considera fundamental para el proyecto, es un artículo presentado por Gastón Márquez y Hernán Astudillo llamado “Actual Use of Architectural Patterns in Microservices-Based Open Source Projects” [12] en el cual se brinda un estudio sobre patrones de microservicios enfocado en la asiduidad con la cual son implementados en proyectos de código abierto.

A través de este artículo se pudo acceder a nuevas fuentes de información y autores que hasta el momento no se habían contemplado. Varias de ellas son blogs dedicados a microservicios o tecnología en general [19] [18] y otros libros o trabajos científicos [8] [15] [22] publicados en importantes conferencias y revistas como [23].

3.1.2 Estudio de las fuentes para la identificación de patrones y categorías

En la segunda etapa del proceso, se analizaron las fuentes en busca de identificar tanto patrones de microservicios, como conceptos asociados que sirvieran para agrupar los patrones en categorías. Como resultado, se identificaron un total de 141 patrones y 24 categorías.

Sin embargo, al profundizar en el análisis se pudo observar que muchos difieren en nomenclatura pero hacen referencia a soluciones iguales o similares que atacan las mismas problemáticas. Por este motivo, se utilizó esta instancia para definir criterios de filtrado para aplicar sobre esta lista, de manera de lograr reducir la cantidad de patrones.

Los criterios de reducción utilizados fueron:

- Nomenclatura
- Posibilidad de acceso a información

3.1.3 Refinamiento de la lista de patrones identificados

En la tercer y última etapa se refinó la lista de patrones y categorías identificadas previamente, con el objetivo de definir un alcance acorde al proyecto.

De los criterios de filtrado mencionados previamente, la nomenclatura fue el principal criterio de reducción. Por nomenclatura se hace referencia a que en muchos de los patrones relevados se notó que si bien la nomenclatura difería, el objetivo y solución propuestos eran idénticos o muy similares. Por este motivo, en primer lugar se analizó la propuesta concreta de cada patrón y se agruparon en base a ese parámetro, para posteriormente dar paso a una etapa en la cual se estableció una convención de nomenclatura a aplicar. Esta idea se ve reafirmada por las conclusiones brindadas en [12], que hace referencia a algunos de estos criterios como base para el filtrado de patrones.

El criterio utilizado para realizar dicho filtro fue: aplicar la nomenclatura que más apariciones tuviese (en la lista original) o por defecto utilizar la nomenclatura brindada por Chris Richardson debido a que fue el autor que más patrones aportó (42) [2].

Al mismo tiempo, se eliminaron de la lista aquellos sobre los cuales la documentación existente es escasa, de difícil acceso o ambigua.

Una vez aplicados estos criterios de reducción el resultado obtenido contempla 49 patrones que forman el alcance de este trabajo.

Por otra parte, en lo que respecta a la categorización de los patrones, en la mayoría de las fuentes se propone, primero encapsular según el área de aplicación y problema concreto para luego agruparlos de forma conveniente. Sin embargo, otras fuentes, no intentan categorizarlos de ninguna manera y solo los exponen de manera independiente.

En este caso puntual, se decidió que las áreas en las cuales se agruparan los patrones tengan relación con las dos variables mencionadas previamente:

- Área de aplicación.
- Problemática concreta.

El concepto "área de aplicación" hace referencia a aspectos generales tales como seguridad, despliegue, observación, implementación. Mientras que "problemática concreta" se refiere sobre qué desventaja (que la arquitectura de microservicios aporta) busca solucionar.

Como resultado de esta etapa se obtuvo una lista de patrones categorizados sobre los cuales trabajar en el proyecto.

Figure 3.1: Esquema de relevamiento

3.1.4 Resumen

A modo de resumen, la figura 3.1 presenta gráficamente el proceso de relevamiento descrito.

3.2 Caracterización de patrones

En esta sección se presenta el trabajo de caracterización realizado en base a los patrones y categorías relevadas.

En primera instancia se presenta un modelo conceptual diseñado con el objetivo de poder uniformizar los conceptos que rodean a los patrones.

Luego, se presenta una definición de cada categoría identificada, profundizando en cada una de ellas en los patrones que la componen. De cada patrón se presenta su instanciación en el modelo presentado anteriormente.

3.2.1 Modelo conceptual

A medida que se fue profundizando en el análisis, se fue haciendo cada vez más acentuada la carencia de estandarización asociada al tema. Por este motivo, fue necesario realizar un trabajo de conceptualización de la realidad para poder diagramar un modelo que permitiera

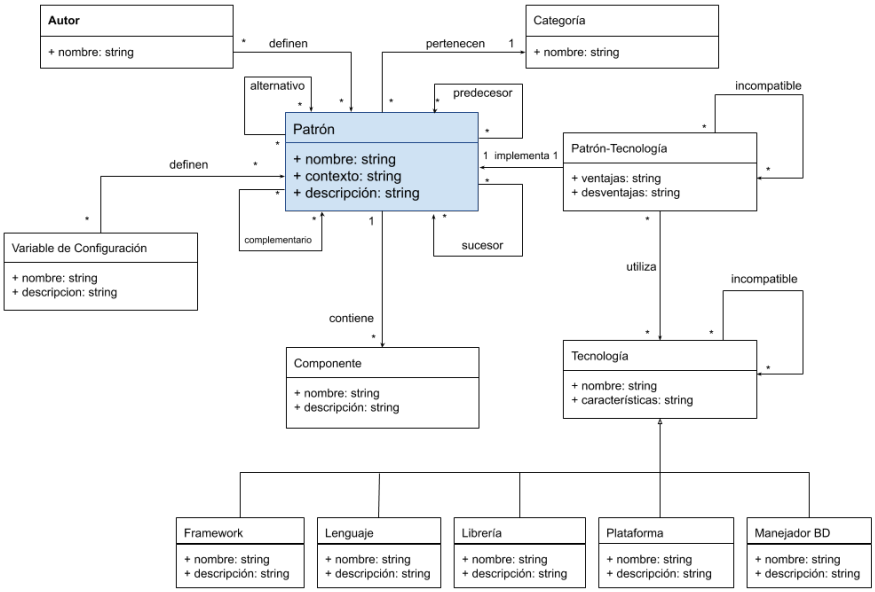

Figure 3.2: Modelo conceptual

describir uniformemente a los patrones.

En la Figura 3.2 se presenta el resultado obtenido luego del proceso de diseño del modelo y seguidamente, se presentan los conceptos allí mencionados.

Patrón:
El patrón es el principal actor en este modelo conceptual. A la hora de aplicar un patrón de microservicios en el diseño de una arquitectura de software se debe pensar en dos aspectos:

- ¿Cuáles son los requisitos de nuestra aplicación?
- ¿Cuál es el problema que queremos resolver?

Además, es importante considerar el hecho de que el uso de un determinado patrón puede incorporar restricciones en el diseño del sistema. Esto se debe a que están relacionados entre si, concretamente, se pueden identificar distintas relaciones como: sucesor, predecesor, alternativo y complementario.

- Sucesor: Un patrón es sucesor de otro cuando la aplicación del primero implica la utilización del segundo.

- Predecesor: Un patrón es predecesor de otro cuando su aplicación desencadena la necesidad de aplicar otro.
- Alternativo: La aplicación de un patrón imposibilita la aplicación de otro. Son alternativos aquellos patrones que brindan soluciones a la misma problemática con enfoques distintos.
- Complementario: Un patrón es complementario de otro cuando pueden aplicarse juntos para un bien común.

Categoría:
Dado que múltiples patrones ofrecen soluciones a problemas concretos surge la necesidad de englobar dichos patrones según su área de aplicación. Por este motivo surge el concepto de categoría bajo la cual se agrupan patrones que proponen distintas soluciones enfocadas en problemáticas puntuales.

Dada la definición del concepto patrón de microservicios presentado previamente, se puede inferir claramente que cada uno sólo puede pertenecer a una categoría ya que brinda solución a una problemática puntual y recurrente.

Es valido destacar que las categorías definidas (las cuales serán presentadas en secciones posteriores) mantienen una relación estrecha y directa con las problemáticas que la arquitectura de microservicios posee.

Componente
El concepto componente tiene como objetivo englobar aquellos conceptos que se desprenden de la solución que el patrón provee. En muchos casos es claro identificar componentes asociados a los patrones, mientras que en otros no es tan sencillo, incluso resultando en que para algunos patrones no se identifican componentes.

Patrón-Tecnología
Este concepto hace referencia a la tecnología mediante la cual se implementan los patrones. Por otro lado, también surge la necesidad de modelar la relación de incompatibilidad que potencialmente existe entre los diferentes patrones y tecnologías.

El uso de una tecnología en la implementación de determinado patrón puede arrojar tres escenarios diferentes:

1. Necesidad de usar una tecnología concreta en otros ámbitos de la aplicación.
2. Imposibilidad de usar determinada tecnología en alguna parte de la aplicación, o incluso en toda la aplicación.
3. Ninguna restricción en cuanto a las tecnologías del resto de la aplicación.

Es por este motivo que además de modelar la entidad tecnología, se modeló una relación entre la entidad patrón y la tecnología para representar la relación entre ambos conceptos: Patrón-Tecnología.

Tecnología
La entidad tecnología puede tomar distintos estados, y en ocasiones la implementación de un patrón podría involucrar más de un tipo de tecnología ya que estos no son disjuntos.

Dentro de los valores que la entidad tecnología puede tomar, se encuentran frameworks, plataformas, librerías, lenguajes y manejadores de bases de datos.

Autor
Este concepto se emplea de manera no uniforme. Dado que en muchos de los patrones relevados se da el fenómeno de que no esta claro de donde surgió (o quien lo definió en primera instancia) este concepto se emplea para representar a la fuente de donde fue relevado en este trabajo. Sin embargo, en otros casos, se utiliza para representar al autor del patrón (en aquellos casos donde su pudo identificar un autor claro).

Variable de Configuración
Este término surge de analizar los requerimientos que se tendría al querer implementar un patrón puntual. Todo contexto donde se desee aplicar la arquitectura de microservicios a través de patrones posee diferentes características y aspectos no funcionales que condicionan la elección y forma de implementación de los patrones. Por ese motivo, bajo el concepto de variable de configuración se busca englobar todos estos aspectos no funcionales y características puntuales de la realidad que impliquen ciertas restricciones sobre la implementación de un patrón de microservicios.

Estas variables son específicas de cada realidad y a través de ellas se pueden tomar decisiones que impliquen la aplicación de ciertos patrones por sobre otros o simplemente podrían definir aspectos puntuales de la implantación de un patrón.

3.2.2 Presentación de los patrones por categoría

En esta sección se presentan los patrones obtenidos producto del relevamiento realizado, agrupados por categoría.

Descomposición:
Proceso mediante el cual se descompone una aplicación en múltiples microservicios de forma tal que, mediante la cooperación y comunicación entre los mismos logren implementar todas las funcionalidades del sistema.

Se debe lograr que una vez que el sistema es descompuesto en servicios, el funcionamiento de los mismos en conjunción logre brindar una imagen del sistema como un todo, siendo transparente para el usuario el hecho de que realmente se están implementando múltiples sistemas autónomos.

En esta categoría se identificaron 2 patrones:

- Descomponer en base a capacidades de negocio
- Descomponer en base a subdominios

Comunicación:
Si bien la arquitectura de microservicios propone implementar los sistemas mediante múltiples microservicios independientes, es de suma importancia la comunicación entre los mismos ya que, por lo general se necesita de su cooperación para satisfacer los requerimientos del sistema.

Englobados en esta categoría se encuentran:

- Invocación a procedimientos remotos
- Mensajería

Gestión de datos:
Forma en que se diagrama el modelo de gestión de datos en una arquitectura de microservicios. Si bien lo más recomendable es aplicar una base de datos por servicio, existen variantes que proponen enfoques distintos.

- Una base de datos compartida
- Una base de datos por servicio

Consistencia:
Mantener la consistencia en sistemas donde se tienen múltiples bases de datos (por ejemplo después de aplicar el patrón antes presentado).

Producto de la investigación en esta área surgió un único patrón:

- Saga

Consulta de datos:
Gestionar las consultas que requieren datos propiedad de múltiples servicios.

Por lo general muchas de las consultas necesitan unir datos que son propiedad de múltiples servicios para satisfacer los requerimientos. A su vez, el realizar consultas distribuidas no es una opción ya que, solo se puede acceder a los datos de un servicio a través de su interfaz (API).

Para realizar esta tarea se destacan dos enfoques:

- Existe una entidad que se encarga de ensamblar los datos necesarios para satisfacer la consulta extrayendo los mismos de los microservicios que sea necesario.
- Mantener réplicas de los datos (propiedad de otros servicios) extra que se requieren para satisfacer la consulta en el servicio consultado.

Para cada una de estas variantes existe un patrón.

- API Composition

- Command Query Responsibility Segregation (CQRS)

Tolerancia a fallas:
En un sistema distribuido, cada vez que un servicio realiza una solicitud sincrónica a otro servicio, existe el riesgo de que ocurra una falla parcial.

Debido a que el cliente y el servicio son procesos separados, es posible que en un momento puntual un servicio no pueda responder de manera oportuna a la solicitud del cliente, por ejemplo porque el servicio podría estar inactivo debido a una falla, encontrarse en estado de mantenimiento o estar sobrecargado y responder extremadamente lento a las solicitudes.

Considerando el hecho de que en este contexto el cliente está bloqueado a la espera de una respuesta, el peligro es que la falla que está afectando al cliente actual podría comenzar a impactar en otros clientes (aquellos microservicios que esperan una respuesta del que está bloqueado).

Se identificó un solo patrón en esta área:

- Circuit Breaker

Despliegue:
Como todas las aplicaciones, las desarrolladas con la arquitectura de microservicios requiere de infraestructura en la cual ser desplegada.

Dado que pueden haber decenas o cientos de servicios desarrollados en distintos lenguajes y frameworks, existen muchas partes móviles que necesitan ser gestionadas, por lo cual es un tema muy importante el cómo se despliega la aplicación.

Se identifican varios patrones que proveen alternativas.

- Service mesh
- Servicios en máquinas virtuales
- Servicios en contenedores
- Plataforma para el despliegue de servicios
- Despliegue sin servidores

Descubrimiento de servicios:
Conocer la ubicación en la red (dirección IP y puerto) de las distintas instancia de servicio desplegadas en la nube.

Dado que en la actualidad las aplicaciones son desplegadas utilizando proveedores de nube pública, que muchas veces ofrecen la gestión de ciertos aspectos como el escalado en función de la demanda, tolerancia a fallas, etc, se ha vuelto todo un desafió el localizar a los servicios a la hora de querer invocarlos. En consecuencia, las aplicaciones deben utilizar un

descubrimiento de servicio.

El descubrimiento de servicios es conceptualmente bastante simple: su componente clave es un registro de servicios (service registry), que es una base de datos de las ubicaciones de red de las instancias de servicio de una aplicación. El mecanismo de descubrimiento de servicios actualiza el registro de servicios cuando las instancias de servicio comienzan y se detienen. Cuando un cliente invoca un servicio, el mecanismo de descubrimiento del servicio consulta el registro del servicio para obtener una lista de instancias de servicio disponibles y enrruta la solicitud a una de ellas.

Existen dos formas de implementar el descubrimiento de servicios:

- Los servicios y sus clientes interactúan directamente con el registro de servicios.
- La infraestructura de despliegue maneja el descubrimiento de servicios.

Englobados en esta categoría se encuentran los siguientes patrones:

- Descubrimiento de servicios del lado del cliente
- Descubrimiento de servicios del lado del servidor
- Registro personal de servicios
- Registro con herramientas de terceros

Observación

Observación de los diferentes servicios en pos de detectar anomalías en el funcionamiento y/o estado de los servicios.

Por otra parte cuando ocurre un falla en el sistema es importante tener un registro de la actividad de los servicios para poder identificar el origen de la falla.

Para este propósito se identifican los patrones presentados a continuación:

- Health Check API
- Registro de logs
- Seguimiento de excepciones
- Seguimiento distribuido
- Métricas de la aplicación
- Audit logging

Seguridad

El tema de la seguridad de las aplicaciones no es un tema puntual de los microservicios si no que concierne a todo tipo de sistemas.

Sin embargo, cuando se implementa seguridad en sistemas con microservicios no se puede mantener el contexto ni las sesiones en memoria ya que los servicios no la comparten.

Es por este motivo que el enfoque utilizado para la autenticación en este tipo de aplicaciones es delegar al API gateway (presentado en la sección 3.2.2) la gestión de la seguridad.

Un solo patrón identificado:

- Token de acceso

Migración:
Migrar desde sistemas monolíticos hacia microservicios.

Patrones:

- Strangler application
- Anti Corruption Layer

Testing
Gestionar las pruebas de la aplicación.

La arquitectura de microservicios hace que los servicios individuales sean más fáciles de testear porque son mucho más pequeños que en una aplicación monolítica. Sin embargo, el gran desafío radica en comprobar que los diferentes servicios funcionan juntos de manera adecuada, y al mismo tiempo evitar el uso de pruebas complejas, lentas y frágiles de extremo a extremo que prueben múltiples servicios juntos.

Con este propósito se identifican 3 patrones:

- Consumer driven contract
- Consumer-side contract
- Service component

Comunicación con el exterior:
Definir y gestionar la comunicación del sistema desarrollado con microservicios con aplicaciones externas. Patrones identificados:

- API Gateway
- Backend for frontends

Mensajería Transaccional:
Gestionar el envió de mensajes de manera transaccional.

Los servicios a menudo necesitan publicar mensajes como parte de una transacción que actualiza la base de datos.

Por ejemplo, servicios que publican eventos de dominio cada vez que crean o actualizan entidades en las bases de datos ya que, si no se hiciera de esta forma, se podría dar que un servicio actualice la base de datos y luego se bloquee, antes de poder enviar el mensaje.

Por lo tanto, si el servicio no realiza estas dos operaciones atómicamente, una falla podría dejar el sistema en un estado inconsistente.

Englobados en esta categoría se identifican:

- Transactional outbox
- Polling publisher

Implementación

Cómo implementar la lógica de negocios del sistema.

Para ello se identifican:

- Transaction Script
- Aggregates
- Eventos de dominio
- Event sourcing

Preocupaciones transversales

Aspectos generales que involucran múltiples áreas del sistema.

- Ubicaciones en la red de diferentes servicios.
- Credenciales.
- Configuración dinámica y estática.
- Logueo de errores.
- Descubrimiento de servicios: Cómo encontrarlos y cómo registrarlos.
- Circuit breaker.
- Health Check APIs.
- Métricas.

Patrones identificados:

- Configuración exterior
- Microservices Chasis

Interfaz de usuario

Interacción con el usuario.

Dentro de las responsabilidades del equipo de desarrollo de cada servicio está el de la experiencia del usuario.

Sin embargo, la complejidad radica en que usualmente existirán pantallas en las cuales sea necesario presentar datos provenientes de múltiples servicios distintos. [86]

Existen dos enfoques para abordar este tema:

- Client-side UI composition
- Server-side page fragment composition

Resumen

En las Tablas 3.1 y 3.2 se presenta la instanciación.

Patrón	**Categoría**
Descomponer en base a capacidades de negocio	Descomposición
Descomponer en base a subdominios	Descomposición
Invocación a procedimientos remotos	Comunicación
Mensajería	Comunicación
Una base de datos compartida	Gestión de datos
Una base de datos por servicio	Gestión de datos
Saga	Consistencia
API Composition	Consulta de datos
Command Query Responsibility Segregation	Consulta de datos
Circuit Breaker	Tolerancia a fallas
Service mesh	Despliegue
Servicios en máquinas virtuales	Despliegue
Servicios en contenedores	Despliegue
Plataforma para el despliegue de servicios	Despliegue
Despliegue sin servidores	Despliegue
Descubrimiento de servicios del lado del cliente	Descubrimiento de servicios
Descubrimiento de servicios del lado del servidor	Descubrimiento de servicios
Registro personal de servicios	Descubrimiento de servicios
Registro con herramientas de terceros	Descubrimiento de servicios

Table 3.1: Patrones relevados

3.2.3 Descubrimiento de servicios

- Descubrimiento de servicios del lado del cliente: Cuando un cliente desea invocar a un servicio, consulta el registro del servicio para obtener una lista de las instancias del

Patrón	Categoría
Health Check API	Observación
Registro de logs	Observación
Seguimiento de excepciones	Observación
Seguimiento distribuido	Observación
Métricas de la aplicación	Observación
Audit logging	Observación
Token de acceso	Seguridad
Strangler application	Migración
Anti Corruption Layer	Migración
Consumer driven contract	Testing
Consumer-side contract	Testing
Service component	Testing
API Gateway	Comunicación con el exterior
Backend for frontends	Comunicación con el exterior
Transactional outbox	Mensajería Transaccional
Polling publisher	Mensajería Transaccional
Transaction Script	Implementación
Aggregates	Implementación
Eventos de dominio	Implementación
Event sourcing	Implementación
Configuración exterior	Preocupaciones transversales
Microservices Chasis	Preocupaciones transversales
Client-side UI composition	Interfaz de usuario
Server-side page fragment composition	Interfaz de usuario

Table 3.2: Patrones relevados

mismo.

Como medida para mejorar el rendimiento, el cliente puede almacenar en caché las instancias del servicio, para luego utilizar un algoritmo de balanceo de carga permitiéndole seleccionar una instancia del servicio específica [68].

En la Tabla 3.3 se presenta la instanciación.

- Descubrimiento de servicios del lado del servidor: En lugar de que un cliente consulte el registro del servicio, realiza una solicitud a un nombre DNS, que se resuelve en un enrutador de solicitud que consulta el registro del servicio.

  En la Tabla 3.4 se presenta la instanciación.

- Registro personal de servicios: El mismo servicio es el encargado de invocar la API

Nombre	Descubrimiento de servicios del lado del cliente
Categoría	Descubrimiento de servicios
Autor	Richardson
Relaciones	Alternativo - Descubrimiento de servicios del lado del servidor Predecesor: Registro Personal de servicios
Componentes	Balanceador de carga, Service registry
Tecnologías	Netflix Ribbon, AWS Elastic Load Balancer (ELB), Kubernetes, Marathon, Spring Framework, Eureka, Consul, Apache Zookeeper, Etcd, Apache Turbine
Variables	Tecnología a utilizar

Table 3.3: Descubrimiento de servicios del lado del cliente

Nombre	Descubrimiento de servicios del lado del servidor
Categoría	Descubrimiento de servicios
Autor	Richardson
Relaciones	Alternativo - Descubrimiento de servicios del lado del cliente Predecesor: Registro Personal de servicios Sucesor Alternativo: Circuit Breaker
Componentes	Balanceador de carga, Service registry
Tecnologías	Netflix Ribbon, AWS Elastic Load Balancer (ELB), Kubernetes, Marathon, Spring Framework, Eureka, Consul, Apache Zookeeper, Etcd, Apache Turbine
Variables	Tecnología a utilizar

Table 3.4: Descubrimiento de servicios del lado del servidor

del registro para registrar su ubicación de red.

En la Tabla 3.5 se presenta la instanciación.

- Registro con herramientas de terceros: En lugar de que un servicio se registre a si mismo en el registro de servicios, un tercero llamado registrador, que generalmente forma parte de la plataforma de implementación, se encarga del registro.

  En la Tabla 3.6 se presenta la instanciación.

3.2.4 Observación

- Health Check API: Plantea exponer un endpoint cuyo propósito sea notificar el estado actual del servicio.

  En la Tabla 3.7 se presenta la instanciación.

- Registro de logs: El objetivo que plantea este patrón es llevar un registro de la activi-

Nombre	Registro personal de servicios
Categoría	Descubrimiento de servicios
Autor	Richardson
Relaciones	Alternativo - Registro con herramientas de terceros Predecesor: Descubrimiento del lado del cliente/servicio
Componentes	Service registry
Tecnologías	Eureka, Consul, Apache Zookeeper, Etcd, Apache Turbine
Variables	Tecnología a utilizar

Table 3.5: Registro Personal de servicios

Nombre	Registro con herramientas de terceros
Categoría	Descubrimiento de servicios
Autor	Richardson
Relaciones	Alternativo - Registro personal de servicios Predecesor: Descubrimiento del lado del cliente/servicio
Componentes	Registrador
Tecnologías	Netflix Prana (Usa Eureka), Zoekeeper, Consul
Variables	Tecnología a utilizar

Table 3.6: Registro con herramientas de terceros

dad de los servicios en un servidor que sea capaz de proveer alertas y capacidad de búsqueda. Para que la idea planteada surta efecto se deben implementar los servicios de forma tal que se almacenen aquellos eventos que verdaderamente sean útiles.

En la Tabla 3.8 se presenta la instanciación.

- Seguimiento de excepciones: La idea detrás de este patrón es muy similar a la presentada en Registro de logs con la diferencia que en este caso se plantea aplicarlo sobre las excepciones.

  En la Tabla 3.9 se presenta la instanciación.

- Seguimiento distribuido: Propone asignar un id único a cada solicitud de forma tal que se pueda realizar un seguimiento de las mismas durante todo su ciclo de vida. El objetivo principal es tener un panorama claro del período en el cual la solicitud estuvo activa para, por ejemplo, cuantificar la latencia que aporta cada servicio al sistema.

  En la Tabla 3.10 se presenta la instanciación.

- Métricas de la aplicación: Los servicios son los responsables de reportar sus propias métricas a un servidor de métricas central, logrando así tener un análisis primario centralizado de cada servicio.

Nombre	Health Check API
Categoría	Observación
Autor	Richardson, Fowler, Newman
Relaciones	Predecesor - Registro de servicios (Es quien lo invoca)
Componentes	Endpoint
Tecnologías	Spring Boot Actuator, HealthChecks
Variables	Tecnología a utilizar

Table 3.7: Health Check API

Nombre	Registro de logs
Categoría	Observación
Autor	Comunidad
Relaciones	Sucesor Opcional: Seguimiento de excepciones, Seguimiento distribuido
Componentes	Logs, Servidor de logs
Tecnologías	DataDog, Loggly, AWS Cloud Watch, ELK (ElasticSearch, Logstash, Kibana), Fluentd, ApacheFlume, Logs, IBM Cloud Log Analysis with LogDNA, Zipkin, Papertrail
Variables	Indices para la busqueda eficiente de logs, Tecnología a utilizar

Table 3.8: Registro de logs

En la Tabla 3.11 se presenta la instanciación.

- Audit logging: Un enfoque similar a algunos presentados anteriormente propone registrar las acciones de los usuarios en un repositorio central.

  En la Tabla 3.12 se presenta la instanciación.

3.2.5 Despliegue

- Service mesh: RedHat define al Service Mesh en [62] como la manera de controlar cómo diferentes partes de una aplicación comparten datos con otras. Es una capa de infraestructura dedicada que está construida por encima de la aplicación. Esta capa asegura que la comunicación entre los servicios es rápida, segura y confiable.

  Un service mesh enruta todo el tráfico de red dentro y fuera de los servicios a través de una capa que implementa por defecto varios de los aspectos tratados como desafíos a lo largo del documento, como por ejemplo circuit breakers, distributed tracing, descubrimiento de servicios y balanceo de cargas [63].

  Tradicionalmente el proceso de despliegue de una aplicación era totalmente responsa-

Nombre	Seguimiento de excepciones
Categoría	Observación
Autor	Richardson, Fowler
Relaciones	Sucesor Opcional: Registro de logs, Seguimiento distribuido
Componentes	excepciones, Servidor de excepciones
Tecnologías	DataDog, Honeybadger, Sentry.io
Variables	Índices para la búsqueda eficiente de excepciones, Tecnología a utilizar

Table 3.9: Seguimiento de excepciones

Nombre	Seguimiento distribuido
Categoría	Observación
Autor	Richardson, Fowler
Relaciones	Predecesor: Registro de logs, Seguimiento de excepciones (se asigna un id a cada log/excepción para su rastreo mas eficiente)
Componentes	Librería de instumentacion, Servidor de rastreo distribuido
Tecnologías	Spring Cloud Sleuth, Zipkin-AWS-X-ray
Variables	Tecnología a utilizar

Table 3.10: Seguimiento distribuido

bilidad del equipo de operaciones y se realizaba de forma manual. Sin embargo, en la actualidad existe una vasta gama de herramientas que permiten la automatización de este tipo de procesos. Por dicho motivo se plantea la utilización de este tipo de tecnologías para implementar una capa que permita abstraer varios de los desafíos a la hora de desplegar aplicaciones que se componen de múltiples servicios [64].

En la Tabla 3.13 se presenta la instanciación.

- Servicios en máquinas virtuales: Desplegar cada servicio como una imagen en una máquina virtual (MV).

  Beneficios:

  - Se encapsula el stack tecnológico utilizado, ya que la imagen contiene las dependencias del servicio por lo cual disminuye la probabilidad de introducir errores en la configuración del entorno de ejecución.
  - Aporta abstracción ya que una vez que se crea y configura la imagen es una caja negra que puede ser desplegada donde sea.

Nombre	Métricas de la aplicación
Categoría	Observación
Autor	Richardson
Relaciones	
Componentes	Servidor de Métricas, Métrica
Tecnologías	Micrometers, AWS CloudWatch Metrics, Prometheus, Grafana, Java Metrics Library
Variables	Métricas importantes, Tecnología a utilizar

Table 3.11: Métricas de la aplicación

Nombre	Audit logging
Categoría	Observación
Autor	Richardson
Relaciones	Predecesor: Event sourcing (Se implementa a través de este)
Componentes	Eventos importantes, Repositorio de eventos
Tecnologías	Google analytics, Segment, Matomo
Variables	Eventos a persistir, Tecnología a utilizar

Table 3.12: Audit logging

  - Instancias de los servicios aisladas unas de otras, propiedad adquirida de las MV.

  Desventajas:

  - Desaprovechamiento de recursos, cada MV tiene la sobrecarga que aporta el sistema operativo. La capacidad mínima de las MV provistas en la nube puede ser significativamente mayor a lo requerido.
  - Lentitud en el despliegue: Las MV en general demoran unos minutos en iniciar.
  - Sobrecarga relacionada con la administración del sistema: Se utiliza el sistema propio de cada MV lo que requiere administración. Esta desventaja se acrecienta sobre todo en contraposición con el enfoque sin servidor presentado a continuación.

  En la Tabla 3.14 se presenta la instanciación.

- Servicios en contenedores: Propone desplegar los servicios como contenedores, donde cada instancia de un servicio se mapea con un contenedor, obteniendo así todas las ventajas que tienen los contenedores por sobre las MV [65].

  En la Tabla 3.15 se presenta la instanciación.

Nombre	Service Mesh
Categoría	Despliegue
Autor	RedHat, Kasun Indrasiri, Nginx, Richardson
Relaciones	Alternativo - Servicios en máquinas virtuales, Servicios en contenedores, Plataforma para el despliegue de servicios, Despliegue sin servidores
Componentes	
Tecnologías	Istio, Linkred, Conduit, Jaeger, Kiali, Red Hat Service Mesh
Variables	Tecnología a utilizar

Table 3.13: Service Mesh

Nombre	Servicios en máquinas virtuales
Categoría	Despliegue
Autor	Richardson
Relaciones	Alternativo - Service Mesh, Servicios en contenedores, Plataforma para el despliegue de servicios, Despliegue sin servidores
Componentes	VM Builder, VM Image
Tecnologías	Animator de Netflix, Packer, Elastic Beacnstalk
Variables	Tecnología a utilizar

Table 3.14: Servicios en máquinas virtuales

- Plataforma para el despliegue de servicios: En este enfoque se propone desplegar los servicios sobre plataformas de orquestación de contenedores. Esta solución está intrínsecamente relacionada con la anteriormente presentada, sin embargo, la diferencia radica en el nivel de abstracción que se provee.

  En este enfoque las plataformas implementan parte de las responsabilidades que tendrían los desarrolladores si no las utilizaran.

  Como principales ventajas se destacan:

  - Agilidad en el despliegue
  - Facilidad para escalar
  - Mayor tolerancia a los fallos

  En la Tabla 3.16 se presenta la instanciación.

- Despliegue sin servidores: Utilizar infraestructura de implementación que oculte cualquier concepto de servidores. Este enfoque está basado en el concepto de funciones como

Nombre	Servicios en contenedores
Categoría	Despliegue
Autor	Comunidad
Relaciones	Alternativo - Service Mesh, Servicios en máquinas virtuales, Plataforma para el despliegue de servicios, Despliegue sin servidores
Componentes	Contenedor
Tecnologías	Docker, Solaris Zones, Docker Cloud Registry, AWS EC2 Container Registry, Docker Compose, Kubernetes, Google Container Engine, AWS ECS, OpenShift, Marathon
Variables	Tecnología a utilizar

Table 3.15: Servicios en contenedores

Nombre	Plataformas para el despliegue de servicios
Categoría	Despliegue
Autor	Richardson, RedHat, IBM
Relaciones	Alternativo - Service Mesh, Servicios en máquinas virtuales, Servicios en contenedores, Despliegue sin servidores Predecesor: Provee Registro de Servicios y Descubrimiento de servicios
Componentes	Plataforma
Tecnologías	Kubernetes, OpenShift
Variables	Tecnología a utilizar

Table 3.16: Plataforma para el despliegue de servicios

servicios [66]. Es un modelo de ejecución en el que el proveedor de nube es responsable de ejecutar un fragmento de código mediante la asignación dinámica de los recursos y, cobrando solo por la cantidad de recursos utilizados para ejecutar el código.

Por otra parte, la infraestructura de implementación es una utilidad operada por el proveedor externo que por lo general utiliza contenedores o máquinas virtuales para aislar los servicios, pero esta configuración es transparente para la organización.

Se aumenta agilidad e innovación y se elimina el tiempo en el aprovisionamiento de servidores y mantenimiento de sistemas [67].

En la tabla 3.17 se presenta la instanciación.

3.3 Detalle de patrones Circuit Breaker y Saga

En esta sección se brindan detalles de los dos patrones en los que se enfocó el proyecto.

Nombre	Despliegue sin servidores
Categoría	Despliegue
Autor	Azure, AWS, IBM, GCP, Serverless Stack
Relaciones	Alternativo - Service Mesh, Servicios en máquinas virtuales, Servicios en contenedores, Plataforma para el despliegue de servicios
Componentes	Funciones ejecutables
Tecnologías	IBM Cloud Functions, AWS Lambda, Google Cloud Functions, Azure Functions
Variables	Funciones a ejecutar, Tecnología a utilizar

Table 3.17: Despliegue sin servidores

3.3.1 Circuit Breaker:

Concretamente, en esta sección se detalla el patrón Circuit Breaker tomando como referencia su instanciación en el modelo conceptual.

Para la realización de esta sección se utilizaron IBM [61], Microsoft [87], Martin Fowler [33] y Chris Richardson [34], [1] como fuentes de información.

Categoría:
Tolerancia a fallas

Contexto:
En la arquitectura de microservicios donde los sistemas se implementan mediante múltiples microservicios independientes interactuando entre sí, es muy probable que alguno falle.

Incluso, cuantos más microservicios coexistan la probabilidad de experimentar fallas es mayor.

Al mismo tiempo, el impacto de una falla puntual de un microservicio sobre el sistema tiene una relación directa con como se implemente la interacción entre los microservicios. Esto se debe a que una implementación inadecuada de la tolerancia a fallas del sistema podría decantarse en que la falla de un único servicio impacte al sistema entero provocando caídas o comportamientos anómalos. Por el contrario, si se tiene una buena implementación en este aspecto, se podría mitigar la falla (que ningún sistema puede evitar al 100%) al punto tal de que solo se vea comprometido el microservicio que la experimenta, sin afectar al resto del sistema.

A la falla de múltiples servicios consecutivamente se lo denomina "falla en cascada" y es un aspecto que se trata de evitar a toda costa en las aplicaciones desarrolladas con la arquitectura de microservicios.

Problema que resuelve:
El patrón Circuit Breaker tiene como objetivo el evitar que un solo componente que experimente una falla provoque un fallo en cascada más allá de sus límites, y de ese modo afectar a

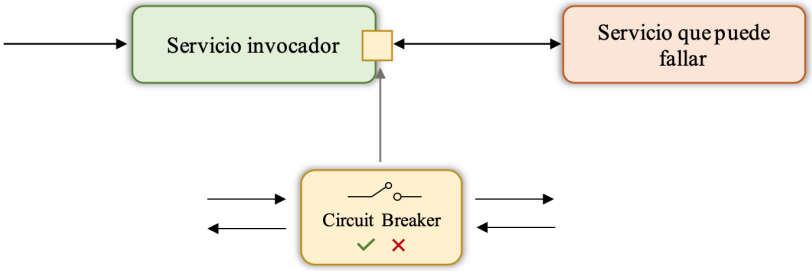

Figure 3.3: Circuit Breaker

todo el sistema junto con él.

El lema bajo el cual se define este patrón es "fallar rápidamente". Es decir, cuando un servicio deja de responder, sus invocadores deben dejar de esperar recibir una respuesta, asumir lo peor y comenzar a lidiar con el hecho de que el servicio que falla puede no estar disponible.

Los Circuit Breakers contribuyen a la estabilidad y la resistencia de los clientes (aplicaciones externas) y los microservicios: los clientes limitan su desperdicio de recursos al tratar de acceder a servicios que no responden, y los microservicios sobrecargados tienen la oportunidad de recuperarse al terminar algunas de las tareas que están procesando actualmente.

La idea es que cuando el servicio objetivo se vuelve demasiado lento o experimenta fallas con demasiada frecuencia, el Circuit Breaker se disparará e invocaciones futuras del cliente devolverán inmediatamente un error.

Solución:
Este patrón consta de un circuito que encapsula las llamadas a los microservicios no permitiendo que se los invoque de manera directa. Por el contrario, todas las invocaciones a los microservicios se realizan a través del Circuit Breaker como se puede apreciar en la Figura 3.3 extraída de el libro "Microservices Patterns" [1] de Chris Richardson.

Concretamente, el patrón puede ser implementado como una máquina de estado finito con tres estados. Los mismos se pueden apreciar gráficamente en la Figura 3.4:

- Cerrado: Las solicitudes llegan al microservicio destino de manera directa. Fallas causadas por la operación solicitada, como excepciones o los tiempos de espera agotados aumentan los fallos contabilizados en el Circuit Breaker así como también los contadores de tiempos de espera.

  Entonces, si estos contadores exceden un umbral especificado (configurado en la implementación), o si se cumplen otros criterios predefinidos (por ejemplo, si se produjo un fallo en particular), el interruptor se dispara y pasa al estado abierto.

- Abierto: Las solicitudes no llegan directamente al microservicio destino. En cam-

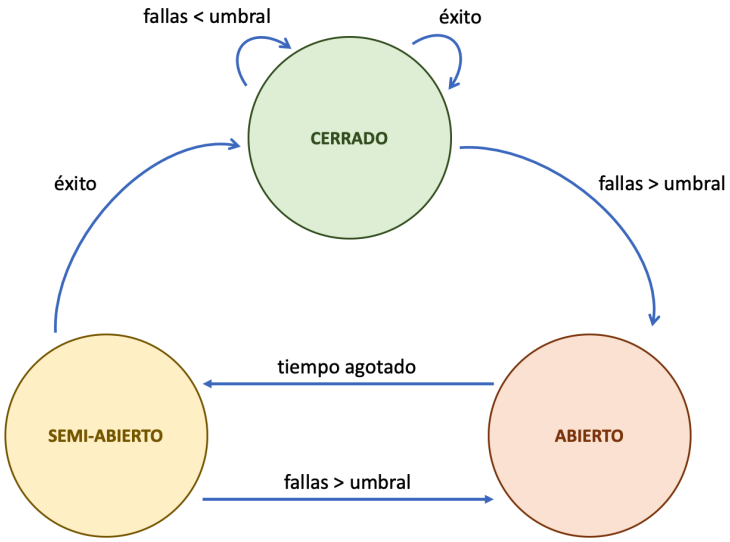

Figure 3.4: Circuit Breaker

bio, un mensaje de falla se entrega inmediatamente al cliente como respuesta, con la salvedad de que potencialmente se pueden retornar mecanismos de manejo de fallas personalizados definidos previamente.

El Circuit Breaker puede transicionar desde el estado abierto al estado semi-abierto, ya sea haciendo invocaciones periódicas al microservicio caído para verificar cuándo vuelve a responder o después de una cantidad específica de tiempo (configurable desde la implementación).

- Semi-Abierto: Mientras se está en este estado, se permite un número limitado de solicitudes a través del microservicio. Si a lo largo del tiempo se ve que el servicio destino envía respuestas exitosas continuamente, el Circuit Breaker se restablece y transiciona al estado cerrado. Esta transición significa que los contadores de falla y tiempo de espera se restablezcan.

  Sin embargo, si cualquiera de las solicitudes falla mientras se está en medio abierto, el circuito volverá a estado abierto.

Componentes:

- Circuito

Escenario resultante:

La aplicación de este patrón tiene los siguientes beneficios:

- Los servicios manejan la falla de los servicios que invocan.

A su vez, tiene la siguiente desventaja:

- Es difícil elegir valores de tiempo de espera sin crear falsos positivos o introducir latencia excesiva.

Variables de Configuración:

- Cantidad de servicios involucrados
- Umbral de tolerancia a fallos (cuántos intentos fallidos se soportan antes de cerrar el circuito).
- Umbral de recuperación (tiempo de espera antes de la verificación de un servicio el cual estaba caído).
- Políticas para transición de los diferentes estados
- Tecnología a utilizar

Tecnologías:

- Resilience4j: Resilience4j es una biblioteca para la implementación de mecanismos de tolerancia a fallas ligera y fácil de usar. Está inspirada en Netflix Hystrix [31], pero diseñado para Java 8 y programación funcional.

  Permite la configuración de los umbrales necesarios de manera sencilla al mismo tiempo que brinda mecanismos para la encapsulación de llamadas a servicios.

- Opossum: "Es un Circuit Breaker Node.js que ejecuta funciones asíncronas y monitorea su estado de ejecución. Cuando se detecta una falla, la librería produce una falla inmediatamente. Además, permite agregar una función de respaldo para que se ejecute en estado de falla" [40].

Variantes a la solución presentada por el patrón:
Este patrón es considerado básico, y por unanimidad se decide siempre incluirlo en las implementaciones ya que el manejo de esta temática es indispensable [1] [3].

Patrones relacionados:

- Microservice Chassis generalmente implementa este patrón.
- API Gateway usa este patrón para invocar a los servicios.
- Service-side discovery podría usar este patrón para invocar servicios

3.3.2 Saga

En esta sección se detalla el patrón Saga tomando como referencia su instanciación en el modelo conceptual presentado anteriormente.

Para la realización de esta sección se utilizaron RedHat [56], Martin Fowler [3] y Chris Richardson [1], [2] como fuentes de información.

Categoría:
Gestión de datos/Consulta de datos

Contexto:
Considerando la naturaleza independiente, abierta y heterogénea de los microservicios, estos suelen usar el patrón de una base de datos por servicio.

Sin embargo, este enfoque aporta complejidad en lo referente a la consistencia de datos en el sistema. Especialmente, un problema que surge de tener diferentes base de datos en el sistema es el manejo de transacciones distribuidas que involucran datos de diferentes servicios.

Una transacción es un conjunto de operaciones que se deben de ejecutar de forma atómica, esto es, o se ejecutan todas o no se ejecuta ninguna. De esta forma se asegura la integridad de datos si algo sale mal en cualquier paso del proceso. Por este motivo, cada paso de la transacción tiene que estar sumamente coordinado y en caso de que alguno de estos falle el sistema debería quedar en un estado consistente sin degradar el rendimiento.

Es por estos motivos anteriormente descritos que el tema del manejo de la consistencia no es un tema trivial si no que requiere de tecnologías y patrones de microservicios específicos para mitigarlo.

Si bien las transacciones distribuidas son un problema inherente a todos los sistemas distribuidos, las soluciones tradicionales suelen padecer algunos problemas en el contexto de microservicios [1].

Por dichos motivos, es que como solución se adopta el modelo de consistencia eventual.

Problema que resuelve:
Mantener la consistencia de los datos en sistemas implementados con la arquitectura de microservicios.

Solución:
Este patrón propone una forma alternativa de mantener la consistencia a través del concepto de sagas.

Se puede definir una saga como una secuencia de transacciones locales coordinadas. Además, para cada una de estas transacciones se debe definir una acción compensatoria que deshaga el cambio que había introducido la transacción dejando el sistema en el mismo estado que tenia antes de realizar la misma.

Explicado de otra manera, la implementación del patrón saga es la ejecución secuencial de una serie de pasos (transacciones), de modo que si uno de estos pasos falla se deban ejecutar las acciones compensatorias definidas para cada paso en el orden inverso.

Existen dos formas de implementar este patrón (se pueden apreciar en la Figura 3.5 y la Figura 3.6):

- Coreografía: Este enfoque se basa en el intercambio de eventos asíncronos. Cada

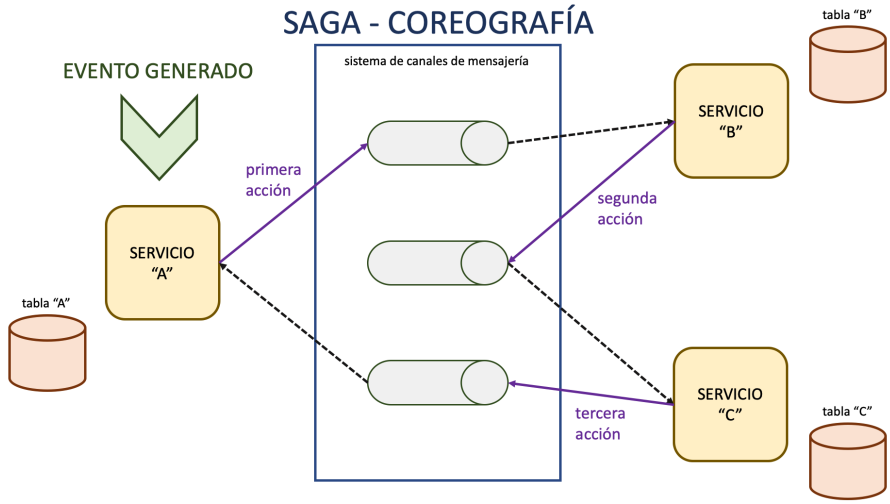

Figure 3.5: Coreografía

microservicio produce un evento cada vez que completa un paso de la saga o un error ocurre, otros microservicios están suscritos a dichos eventos y continúan con el proceso acorde.

- Orquestación: Existe un coordinador central de transacciones, quien orquesta cada paso de la transacción y coordina las operaciones de compensación en caso de fallos. Cada vez que un paso se completa, el microservicio le avisa al coordinador que ha terminado. Esta orquestación permite que los microservicios se mantengan desacoplados.

Implementación usando coreografía:

La implementación de la lógica del patrón estará distribuida a lo largo de los servicios que intervienen (participantes). Es decir no existe un coordinador central el cual indique a los participantes qué acciones deben realizar. Cada servicio debe conocer cómo responder a uno o más estados (eventos) de la saga.

Beneficios:

- Simplicidad: Los servicios publican eventos atómicamente cuando crean, actualizan o eliminan objetos.
- Bajo acoplamiento: Los servicios participantes únicamente se suscriben a los eventos de quien interactúa con él en las transacciones, pero no tienen conocimiento sobre ningún otro participante.

Desventajas:

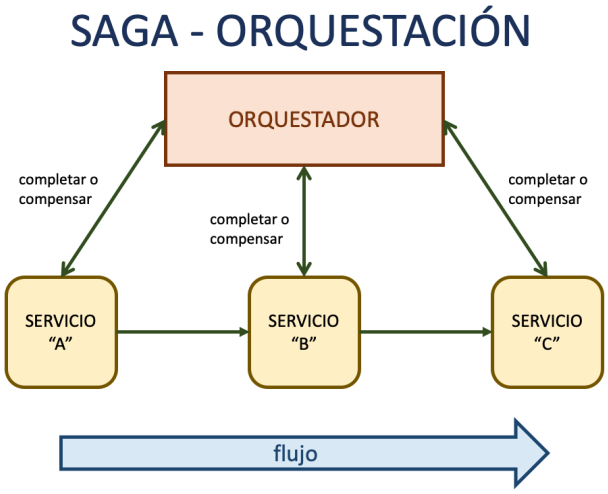

Figure 3.6: Orquestación

- Dificultad en entender el código: La lógica de la implementación del patrón saga está distribuida en varios servicios. Esto conlleva a que sea más difícil de entender el código.
- Dependencias cíclicas entre servicios: Generalmente se dan dependencia cíclicas entre diferentes servicios.
- Riesgo de caer en alto acoplamiento: Puede darse que por la lógica de un servicio quede suscrito a todas las actividades de otro generando así que cada vez que se produce una acción en uno, el otro se vea obligado a actualizarse.
- Cada evento publicado debería tener un identificador que permita a los demás participantes interpretar correctamente el evento en el contexto adecuado, generando una complejidad extra.
- Cada servicio debería realizar la transacción y publicar el evento de manera atómica para que funcione correctamente el sistema.

Implementación usando orquestación:

Existe un proceso o servicio que se encarga de la coordinación de los pasos de la saga, denominado orquestador. Este componente extra contiene toda la lógica de la saga, simplificando el desarrollo y evitando dependencias cíclicas entre servicios.

Una buena manera de modelar un orquestador es como una máquina de estados. Cada transición tiene una acción, que es la invocación a un participante de la saga. Al mismo tiempo, las transiciones entre estados se desencadenan al completarse una transacción local realizada por

otro participante, el estado actual y el resultado específico de la transacción local determinan la transición de estado y que acción, si corresponde, se debe realizar.

Beneficios

- Dependencias más simples: No introduce dependencias cíclicas ya que siempre es el orquestador quién invoca a los distintos participantes, pero los participantes no invocan al orquestador. Como resultado, el orquestador depende de los participantes pero no al revés, por lo cual no se producen dependencias cíclicas.
- Menos Acoplamiento: Cada servicio es responsable de implementar una API que es invocada por el orquestador, por lo cual no es necesario conocer los eventos publicados por los participantes de la saga.
- Lógica simplificada: La lógica de coordinación se localiza solamente en el orquestador. Los objetos de dominio son más simples ya que no conocen las sagas en las que participan.

Desventajas

- Único punto de falla: Al centralizar la lógica de negocio encargada de la orquestación se introduce un único punto de falla lo que deja al sistema propenso a fallar cuando este falla, para evitar este problema se busca evitar que los orquestadores contengan cualquier otra lógica que no tenga que ver con la secuenciación de las acciones de los saga que coordina.

Componentes:
Este patrón tiene como componente al orquestador si es que se decide implementarlo mediante orquestación.

En caso de usar coreografía se podría llegar a usar algún canal de mensajería como moderador de eventos.

Escenario resultante:
Este patrón tiene como beneficio que la aplicación mantenga la consistencia de los datos que maneja en múltiples servicios sin utilizar transacciones distribuidas clásicas.

Como inconveniente, los desarrolladores deben diseñar transacciones compensatorias que deshagan los cambios realizados en pasos anteriores de que se produzca una falla en la saga, incrementando la complejidad.

Además, hay que contemplar que para poder ser confiable, un servicio debe actualizar su base de datos y publicar un evento de manera atómica.

Variables de Configuración:
- Para la implementación de este patrón, se deberá definir según el contexto y la realidad en la que se quiera aplicar el método de coordinación adecuado (orquestación o coreografía).

- Además se tiene que definir las tecnologías que se utilizarán para la implementación (si utilizar o no frameworks de terceros).
- A su vez, optar por usar un enfoque basado en eventos o en mensajería.

Tecnologías asociadas:

- Eventuate Tram: Es un framework desarrollado por Chris Richardson que es uno de los autores de referencia en esta temática y muy distinguido a lo largo del tiempo por sus aportes constantes en esta área de los microservicios.

  Está orientado a microservicios basados en Java/Spring o Micronaut que usan el modelo tradicional de persistencia. Como principal característica se puede destacar que permite el envío de eventos y mensajes de forma atómica con respecto a la actualización de la base de datos. Esto es una gran ventaja a la hora de la implementación de la concurrencia en los sagas.

  Por otra parte también se encarga de la eliminación de mensajes o eventos duplicados, por lo cual el desarrollador no debe preocuparse en implementar políticas para la gestión de este problema.

  Tecnologías compatibles:

  - Frameworks: Spring Boot o Micronaut
  - Base de datos: MySQL, Postgres.
  - Canales de mensajería: Apache Kafka, ActiveMQ, RabbitMQ y Redis.

- Axon: Es un framework liviano que ayuda a los desarrolladores a crear aplicaciones escalables y extensibles abordándolo desde la arquitectura.

  Está basado en paradigmas arquitectónicos tales como Diseño Controlado por Dominio (DDD) y CQRS, este framework fue diseñado para proporcionar componentes básicos que requiere el patrón CQRS y ayudar a crear aplicaciones escalables y extensibles mientras al mismo tiempo se mantiene la consistencia de la aplicación en sistemas distribuidos.

  Tecnologías compatibles:

  - Frameworks: Spring.
  - Base de datos: Conexión JDBC.
  - Canales de mensajería: No hay restricción.

Variantes a la solución presentada por el patrón:

Two-Phase Commit: Es un protocolo muy conocido que aprovecha los beneficios de las transacciones ACID pero en sistemas distribuidos.

Sin embargo, lidiar con esta tecnología no es tan simple, 2PC consiste en dos etapas bien definidas, la etapa de preparación "PREPARE" y la de "COMMIT".

Sumado a esto, resuelve el problema solo hasta cierto punto ya que por ejemplo, no resuelve los problemas planteados a continuación.

- No existe ningún mecanismo para deshacer la otra transacción si un microservicio no está disponible en la fase de confirmación.
- Todos los participantes de la transacción tienen que esperar hasta que el más lento termine su confirmación.

Además, el uso de transacciones distribuidas con 2PC no es una opción para sistemas con microservicios ya que, por ejemplo, algunas bases noSQL no soportan este tipo de transacciones ni tampoco son soportadas por algunos brokers como RabbitMQ y Apache Kafka.

Patrones relacionados:

- La utilización del patrón una base de datos por servicio genera la necesidad de usar este patrón.
- Además debe utilizarse alguno de los patrones pensados para poder actualizar la base de datos y publicar los eventos o mensajes de manera atómica, estos patrones son Event Sourcing o Transaccional Outbox.

3.4 Identificación de funcionalidades

El objetivo de esta sección es presentar las funcionalidades identificadas producto del relevamiento e caracterización de los diferentes patrones contemplados.

Dada la cantidad de fuentes de información y la poca estandarización que existe en la actualidad se cree de fundamental importancia el hecho de contar con una plataforma que asista al usuario desde las etapas más tempranas del diseño del sistema hasta la generación de parte del código del mismo.

3.4.1 Funcionalidades

Presentación de un catálogo de patrones

Brindar un catálogo de patrones, presentando información sobre los mismos. De cada patrón se podrá consultar información asociada al contexto de aplicación, tecnologías disponibles, solución propuestas y conceptos relacionados.

Por otra parte, se brindara una serie de filtros mediantes los cuales los usuarios sean capaces de consultar los patrones de manera ágil e intuitiva.

Extensión de la plataforma

Brindar la capacidad de extender patrones ya existentes o agregar nuevos patrones de una forma intuitiva y directa.

Modelado de la realidad a través de aspectos no funcionales

El objetivo es extraer información desde el usuario a través de preguntas concretas, para identificar aspectos no funcionales de la realidad.

De esta manera se permite que el público objetivo de la plataforma no se acote necesariamente a usuarios netamente técnicos, si no que, a través de esta funcionalidad sería posible que usuarios con solamente conocimiento del dominio de negocio, sean capaces de brindar información suficiente como para que la plataforma pueda diagramar un primer boceto de la arquitectura del sistema a desarrollar.

Como resultado del procesamiento de los datos proveídos por los usuarios el sistema será capaz de disponibilizar una visualización de la arquitectura resultante explicitando los posibles patrones a utilizar en pos de abarcar los aspectos relevados.

Manipulación gráfica de la arquitectura

Luego de interpretar la información extraída desde el usuario se tendrá la capacidad de sugerir combinaciones de patrones y tecnologías disponibles para modelar la realidad presentada.

A través de la información surgida de las preguntas formuladas al usuario, la plataforma tendrá la capacidad de sugerir distintas combinaciones de patrones disponibles para satisfacer los requerimientos del sistema.

Conjuntamente con los patrones, se brindará un conjunto de tecnologías disponibles para la implementación de dichos patrones, de forma tal, que el usuario sea capaz de visualizar gráficamente la sugerencia de arquitectura basada en patrones.

Al mismo tiempo, el usuario será capaz de modificar dicha arquitectura, agregando o removiendo patrones pudiendo visualizar el impacto (en términos de compatibilidad entre patrones y entre patrones y tecnologías) de sus acciones.

De esta manera, se logra dar una visión gráfica de la arquitectura del sistema y las distintas variantes tecnológicas disponibles.

Obtención de implementaciones de referencia

Finalmente, utilizando el diagrama de arquitectura obtenido como resultado del paso anterior, la plataforma será capaz de generar dicho modelo en un formato estándar (XML, JSON, YAML, etc) a partir del cual se pueda generar código de manera automatizada.

Concretamente, se plantea el hecho de que a partir de: un modelo de arquitectura de microservicios basado en patrones, un conjunto de tecnologías y una serie de restricciones en términos de compatibilidad, la plataforma sea capaz de generar estructuras de código de cada patrón.

Dichas estructuras (denominadas implementaciones de referencia) servirán de base para la implementación de nuevas realidades con microservicios, permitiendo que los usuarios cuenten con una guía practica sobre la cual profundizar para satisfacer sus objetivos.

Acceso a aplicaciones de ejemplo

A través de la plataforma los usuarios serán capaces de acceder a aplicaciones de ejemplo. En dichas aplicaciones se brindara el patrón concreto implementado en una realidad especifica con tecnologías concretas. De esta manera los usuarios serán capaces de contar con aplicaciones reales en las cuales se implementan los patrones concretos.

Como consecuencia, al contar con implementaciones de referencia y aplicaciones de ejemplo los usuarios contaran con guías claras y practicas sobre las cuales construir sus aplicaciones. Este hecho provoca que la curva de aprendizaje que implican el desarrollo con microservicios (para nada despreciable) se reduzca significativamente atacando uno de los puntos mas desafiantes que esta arquitectura posee en la actualidad.

3.4.2 Público objetivo

La solución esta enfocada en dos tipos de usuarios.

Por un lado usuarios experimentados los cuales se encargarían de la extensión de la plataforma en pos de abarcar cada vez mas patrones y extender las funcionalidades que ya se brindan. Concretamente, serán los encargados de extender las implementaciones de referencia existentes así como también las aplicaciones de ejemplo de cada patrón. A su vez, al agregar nuevos patrones deberán proveer de estas variables antes mencionadas para mantener la calidad de la plataforma.

Por otro, usuarios con menos experiencia o principiantes los cuales tengan la posibilidad de utilizar la plataforma para nutrirse tanto de información concreta, como de código de ejemplo de cada patrón. Reduciendo la complejidad de implementar la arquitectura en nuevos sistemas.

3.4.3 Extensibilidad

Dada la gran cantidad de patrones y tecnologías existentes, un aspecto que se considera de suma importancia desde la etapa de concepción de la plataforma es la extensibilidad de la misma.

Por dicho motivo, se proponen una serie de funcionalidades mediante las cuales, la plataforma pueda ser extendida a lo largo del tiempo, ya sea, en términos de cantidad de patrones soportados, variedad de tecnologías, estructuras de código brindadas, formatos en los que son generados los modelos de arquitectura, etc.

3.4.4 Flujo de la plataforma

En la figura 3.7 se puede apreciar el flujo propuesto.

En primera instancia, se releva información de los usuarios a través de la formulación de preguntas y en base a dicha información se identifica una arquitectura base incluyendo los patrones necesarios para implementarla.

El resultado de dicho proceso es presentado de manera gráfica permitiendo su modificación

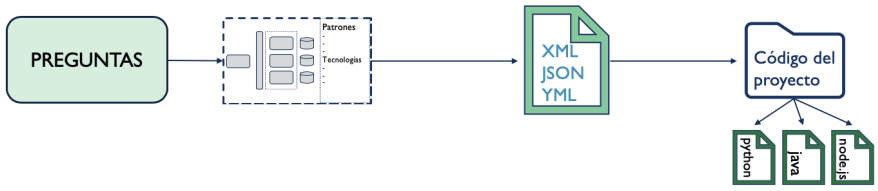

Figure 3.7: Flujo de ejecución

agregando patrones y/o cambiando tecnologías. Mas precisamente, permite visualizar el impacto (en términos de la compatibilidad) de agregar o cambiar un patrón en la implementación del sistema como un todo.

Luego, se genera una representación del modelo de arquitectura en un formato estándar, para posteriormente, generar las implementaciones de referencia en base a dicha representación.

3.5 Trabajo relacionado

En esta sección se presentan trabajos relacionados que puedan ser contemplados como, complemento y/o alternativas a la solución presentada en la siguiente sección.

Micro Freshener se define como [13]: "Un prototipo basado en la web que permite identificar aspectos en una arquitectura que posiblemente violen los principios de los microservicios y seleccionar los componentes necesarios para refactorizar". Es un prototipo que provee parte del flujo presentado anteriormente, ya que, brinda el modelado gráfico de la arquitectura.

Además, profundizando en la implementación de este prototipo, se pudo constatar que el modelado gráfico de la arquitectura se representa mediante el formato YAML. Por lo cual, las similitudes con parte del flujo presentado anteriormente son evidentes.

Sin embargo, no se ajusta en su totalidad ya que no provee ninguna funcionalidad mediante la cual permita a los usuarios visualizar compatibilidad entre patrones ni entre patrones y tecnologías.

De todas maneras, se considera de gran interés el hecho de que exista un prototipo de este estilo que implemente parte del flujo propuesto.

Por otra parte, otro trabajo similar se presenta en [98] y se denomina Microservices Migration Patterns.

Dicho estudio está enfocado en patrones de migración hacia microservicios y puntualmente se aboca a diseñar un "template" único para englobar los patrones de migración existentes. En el proceso, define un meta modelo en el cual estandarizar la información existente sobre los distintos patrones.

Lo interesante de "Microservices Migration Patterns" es que si bien está enfocado en una parte especifica de los patrones de microservicios, dedica esfuerzo en la búsqueda de la estandarización de la información.

La diferencia radica en que en este proyecto de grado se propone un modelo conceptual pensado para los patrones de microservicios en general, mientras que, por otra parte, en Microservices Migration Patterns se diseña uno similar pero enfocado únicamente en patrones de migración.

No obstante, se pueden apreciar similitudes significativas entre los dos modelos definidos por lo cual este trabajo relacionado fue muy importante para reafirmar algunas de las decisiones tomadas.

3.6 Resumen

Por último, en esta sección se presenta un resumen de la etapa de análisis.

Este capítulo tuvo como objetivo el identificar las principales funcionalidades que una plataforma que asista en la implementación de aplicaciones basadas en microservicios guiadas por patrones debería proveer.

Para lograrlo, fue necesario realizar un relevamiento en pos de identificar los patrones existentes y categorizarlos en función de su propósito.

Una vez realizado el relevamiento, fue clara la falta de estandarización en la información por lo cual se diseñó un modelo conceptual donde se instanciaron los patrones identificados.

Una vez que se tuvo la información unificada, se procedió a identificar las funcionalidades que una plataforma de este estilo debería proveer.

Luego, se relevaron trabajos relacionados con el objetivo de visualizar claramente cuáles de las funcionalidades identificadas se proveen desde otras implementaciones.

Sin embargo, si bien se encontraron algunos trabajos que cubren parte del flujo propuesto, no se constató la existencia de ningún sistema que provea a los usuarios de las capacidades que debería proveer una plataforma de este estilo.

Por este motivo, en el siguiente capítulo se presenta una solución que busca atacar parte de las funcionalidades identificadas previamente.

4 Solución propuesta

En esta sección se presenta la solución propuesta.

Después de analizar la realidad e identificar las funcionalidades que una plataforma que asista en la implementación de aplicaciones basadas en microservicios y guiadas por patrones deberían tener, se presenta una solución para atacar parte del flujo identificado.

En primera instancia se brinda una descripción general de la solución, para luego ir profundizando en cada uno de los aspectos principales. Al mismo tiempo, se presenta la arquitectura de la solución y finalmente las decisiones importantes referentes al diseño de la misma.

4.1 Descripción general

Por un tema de alcance del proyecto se tomó la decisión de que la solución propuesta hiciera foco en la generación de código y en la interacción con el usuario.

La plataforma está enfocada en dos roles de usuarios diferentes. Por un lado, usuarios administradores o gestores quienes serán los encargados de la mantención y extensión de la plataforma en términos de cantidad de patrones contemplados y tecnologías para implementarlos. Por otro lado, los usuarios desarrolladores los cuales podrán interactuar con la plataforma obteniendo documentación, implementaciones de referencia y pudiendo visualizar aplicaciones de ejemplo.

En la Figura 4.1 se puede apreciar una vista general de la plataforma, incluyendo la interacción con los distintos tipos de usuarios.

La propuesta de solución consta de una plataforma web, a través de la cual los usuarios puedan realizar una serie de acciones que se detallarán a continuación.

- Navegación: La interfaz web propuesta brinda a los usuarios la capacidad de navegar por los diferentes patrones pudiendo visualizar su documentación, además de contar

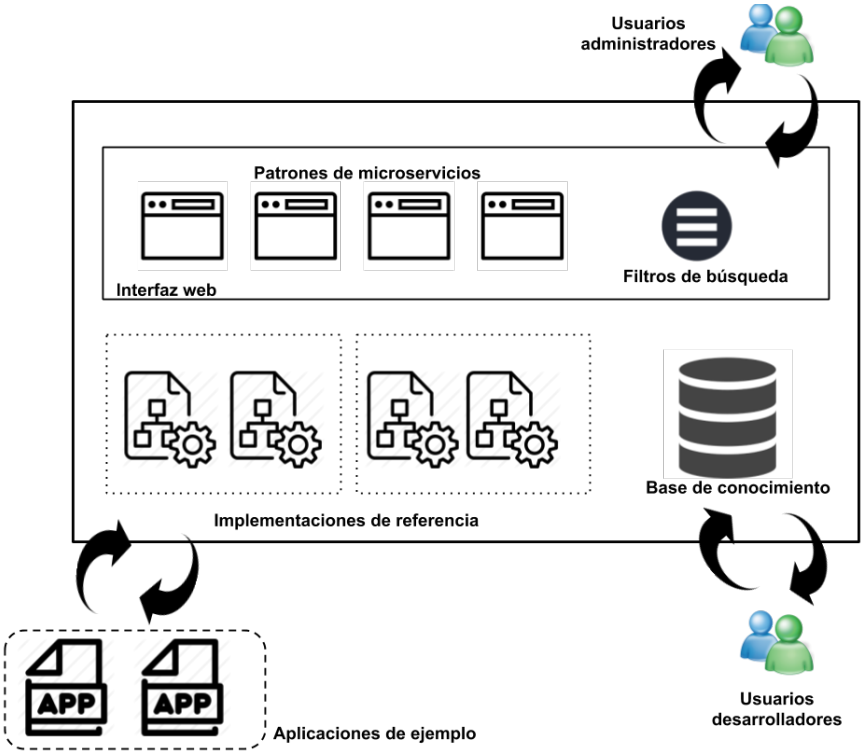

Figure 4.1: Vista general de la plataforma

con filtros de búsqueda específicos en pos de facilitar la consulta de datos.

- Consulta de información: A través de la selección de un patrón especifico los usuarios pueden consultar información concreta del patrón la cual se estructura en base al modelo conceptual presentado anteriormente.
- Obtención de implementaciones de referencia: Los usuarios pueden acceder a implementaciones de referencia en distintas tecnologías las cuales, concretamente son estructuras de código sobre los patrones. El objetivo de esta funcionalidad es ayudar a los usuarios a la hora de aplicar los diferentes patrones contemplados.
- Obtención de aplicaciones de ejemplo: Se provee de aplicaciones de ejemplo implementadas con diferentes stacks tecnológicos. Las aplicaciones de ejemplo son implementaciones concretas de una realidad puntual en la cual se tienen las implementaciones de referencia funcionando en un ambiente real.

4.2 Catálogo de patrones

A través de la interfaz web se provee la posibilidad de navegar por los diferentes patrones presentados en forma de catálogo como se muestra en la Figura 4.2.

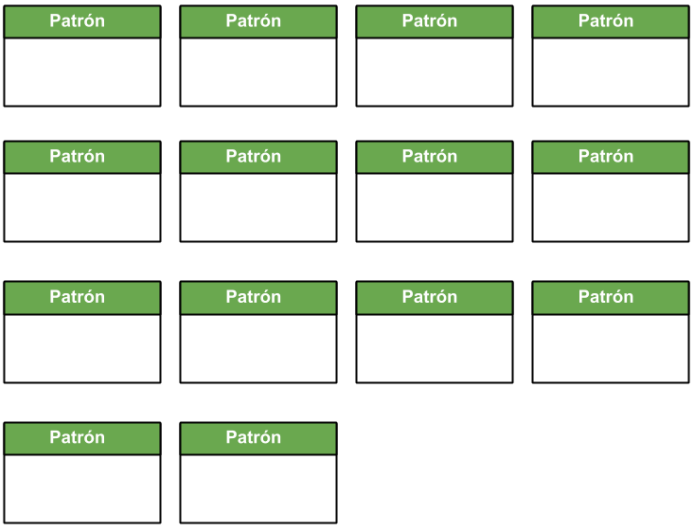

Figure 4.2: Catálogo de patrones

Con el objetivo de mejorar la usabilidad del sistema, se busca facilitar la navegación entre los distintos patrones a través de la implementación de filtros. Particularmente, se filtran los patrones según su nombre o categoría logrando que sean accesibles de manera sencilla.

En la Figura 4.3 se presenta en forma de esquema una representación de la búsqueda de un usuario filtrando los patrones por categoría.

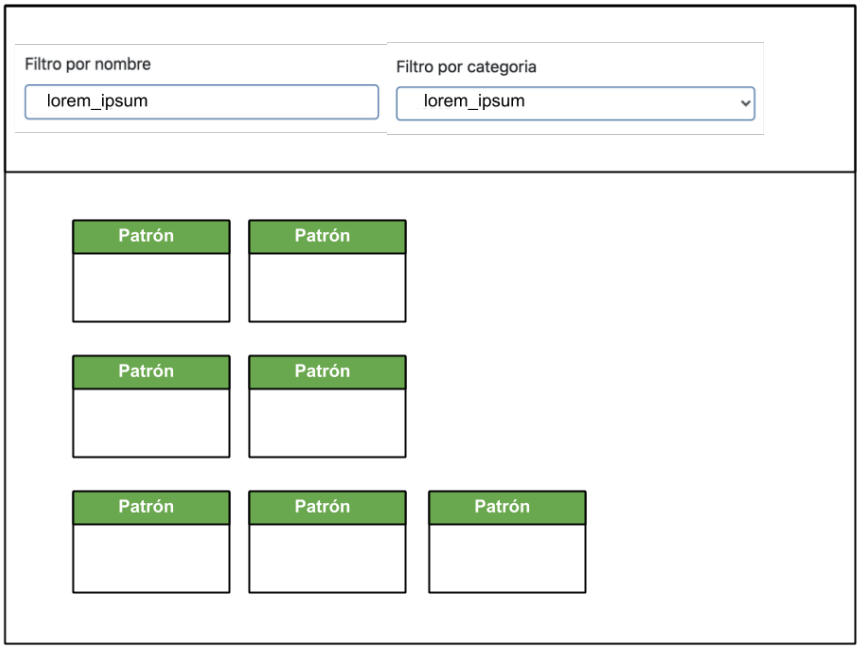

Figure 4.3: Filtro de búsqueda

De cada patrón se brinda documentación sobre diferentes aspectos. Específicamente, se provee de la instanciación del patrón en el modelo conceptual, sumado a la posibilidad de acceder a las implementaciones de referencia asociadas.

Sin embargo, esta no es la única forma de visualización de la información. Conjuntamente con la estructura antes detallada, se provee para algunos patrones, una estructura extendida que complementa la información.

En la Figura 4.4 se provee un esquema en el cual se puede observar aspectos puntuales de la documentación ofrecida. Cabe destacar que la información brindada surge de la instanciación de los patrones presentada en la sección 3.2.

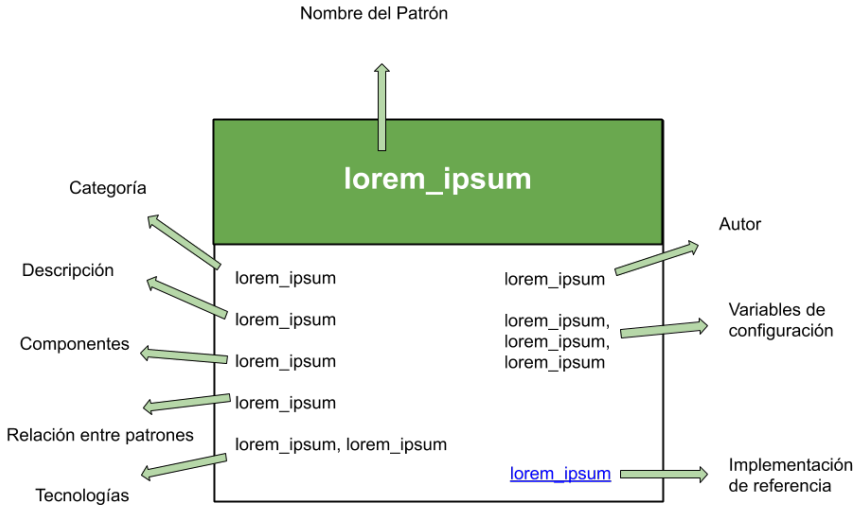

Figure 4.4: Visualización del patrón

4.3 Implementaciones de referencia

Conjuntamente con la documentación brindada sobre cada patrón, se proveen implementaciones de referencia.

Concretamente, estas implementaciones son estructuras de código genéricas las cuales facilitan la implementación de los patrones. Los usuarios pueden simplemente copiando estas implementaciones de referencia tener un punto de partida para la implementación del patrón puntual en su realidad especifica. Al mismo tiempo, se provee la capacidad de configurar ciertas variables sobre las implementaciones de referencia ya que son parametrizables, permitiendo a los usuarios desarrolladores instanciarlas de acuerdo a sus necesidades.

En la Figura 4.5 se presenta un esquema simbólico de la funcionalidad antes detallada.

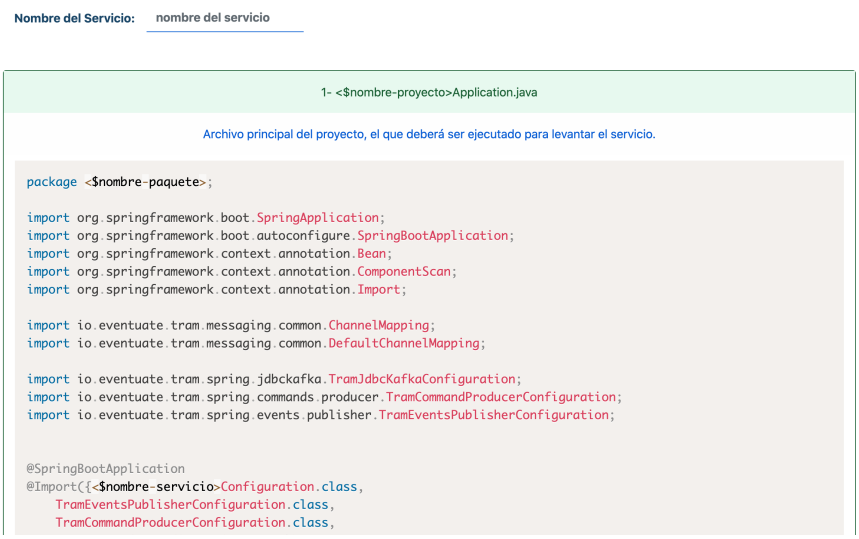

Figure 4.5: Personalizar las implementaciones de referencia

Como se puede apreciar, al introducir el nombre del servicio automáticamente se remplazarán las ocurrencias de "$nombre-servicio" personalizando así la estructura obtenida.

4.4 Aplicaciones de ejemplo

Una de las características más importantes de la plataforma es la de proveer una aplicación de ejemplo asociada a cada implementación de referencia.

Cada aplicación de ejemplo está basada en una realidad específica e implementa el patrón concreto instanciando la implementación de referencia asociada al mismo.

Se cree firmemente que esta característica es un diferenciador claro respecto de la documentación oficial de las tecnologías y proyectos ya existentes, ya que ofrece la posibilidad de tener el patrón implementado en tecnologías concretas como parte de un contexto realista.

Este hecho es de gran valor para los usuarios ya que es muy difícil encontrar proyectos de código abierto implementados en tecnologías puntuales que sirvan como ejemplo de patrones de microservicios.

4.5 Extensibilidad

Una característica muy importante de esta solución es la posibilidad de extenderla.

Esta idea de extensibilidad se refleja en dos aspectos, por un lado el hecho de extender un patrón existente o agregar nuevos patrones a la plataforma.

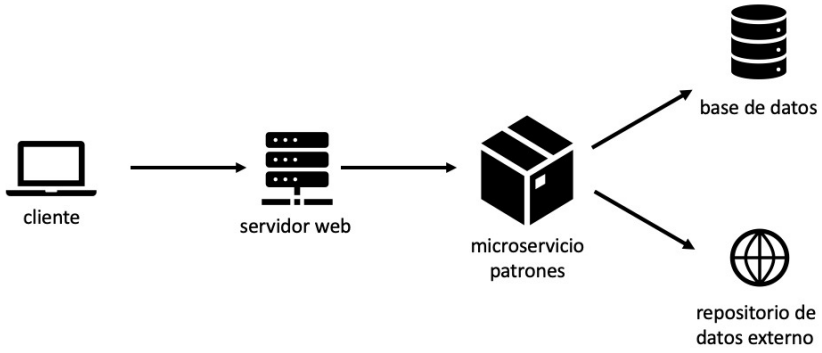

Figure 4.6: Arquitectura de la Solución

En lo que refiere a extender patrones existentes, la plataforma permite extender la documentación general del patrón a través de la interfaz web, así como también, agregar nuevas implementaciones de referencia simplemente introduciendo el link al repositorio donde reside el código a dichas estructuras.

De esta manera se logra que los usuarios sean capaces de extender los patrones ya contemplados en la plataforma.

Por otra parte, la otra forma en la que se puede extender la plataforma es agregando nuevos patrones no contemplados actualmente, a través de la interfaz web. Al seleccionar la opción "Agregar nuevo patrón" se despliega un formulario mediante el cual los usuarios pueden agregar la documentación del patrón conjuntamente con las implementaciones de referencias (en tecnologías concretas) asociadas.

El único requerimiento que se impone a los nuevos patrones ingresados es que se respete la estructura de información definida.

4.6 Arquitectura de la solución

En esta sección se presenta, a nivel general, la arquitectura propuesta.

En la Figura 4.6 se puede apreciar un diagrama de alto nivel de la arquitectura la cual sigue los lineamientos típicos de las aplicaciones web tradicionales.

Consta de 3 componentes internos en los cuales se distribuye la lógica de la aplicación y un componente externo donde residen las aplicaciones de ejemplo de cada patrón.

Más precisamente, los componentes son:

4.6.1 Frontend

En el Frontend se encuentra encapsulada la capa de presentación de la aplicación que se encarga de toda la interacción con el usuario. Concretamente, se encarga de la visualización de los diferentes patrones, la navegación entre los mismos, los filtros específicos mediante los cuales se pueden consultar los patrones y permite la extensión de la plataforma a través de las formas antes presentadas.

Por otro lado, este componente interactúa con el backend para obtener la información necesaria, al mismo que tiempo que, como se mencionó previamente, sirve como punto de entrada de nueva información introducida a la plataforma.

4.6.2 Backend

Este componente es el encargado de implementar la lógica de la aplicación. A través del backend se exponen una serie de métodos que son ejecutados por el frontend mediante la invocación a los mismos.

Concretamente, estos métodos que exponen las funcionalidades de la aplicación son implementados siguiendo los lineamientos del protocolo REST mediante los cuales se permite la manipulación de los patrones.

Dentro de las funcionalidades disponibles se puede encontrar:

- Agregar un nuevo patrón: Genera un nuevo patrón en la base de datos y asocia la información pertinente.
- Obtener información de un patrón específico: Disponibiliza al usuario la información asociada al patrón que se está consultando extrayendo la misma desde la base de datos. Por otra parte expone, mediante URLs, las diferentes implementaciones que residen en repositorios externos.
- Actualizar la información referente a un patrón: Modificaciones tanto en la información básica de los patrones, como en la incorporación de nuevas implementaciones.
- Eliminar patrones.

En otras palabras, el backend es una API REST que se encarga de proveer los métodos necesarios para la manipulación de los patrones por parte del frontend (quien consume dichas funcionalidades y las presenta al usuario).

Por último, también se encarga de la comunicación con la base de datos donde se almacena la información.

4.6.3 Base de datos:

Como su nombre lo indica, este componente es la base de datos de la aplicación donde reside la documentación asociada de cada patrón, conjuntamente con las implementaciones de ejemplo.

Sumado a esto, cabe destacar que el código de las aplicaciones de ejemplo no está directamente alojado en la base de datos, si no que se encuentra una referencia al mismo.

4.6.4 Repositorios Externos:

Se utilizan repositorios externos para almacenar las aplicaciones de ejemplo asociadas a las implementaciones de referencia.

4.7 Problemas encontrados

- Durante la definición de la solución surgió el dilema de qué valor real aportaría el entregar implementaciones de referencia (estructuras de código generadas) en base al conocimiento de los patrones, pero sin haber sido validadas en instancias reales (instancias generadas que simulen un contexto de aplicación).

  Fue de este modo que surge la idea de implementar para cada implementación de referencia una aplicación de ejemplo que muestre que las estructuras brindadas efectivamente son aplicables.

  A su vez, de esta manera se logra entregar aplicaciones funcionales en las cuales se incluyen las estructuras como parte de la implementación, permitiendo al usuario tener un ejemplo concreto y funcionando de la estructura que desea incorporar a su proyecto.

- Buscando que la solución sea lo más flexible posible, se decidió que el código de las aplicaciones de ejemplo se almacene en repositorios externos. De esta manera se logra integrar a la plataforma, repositorios externos los cuales son muy utilizados en la actualidad por todas las ventajas que ofrecen, como por ejemplo, manejo de las versiones eficiente, facilidad a la hora de descargar y aplicar el código, estandarización en la forma de colaboración, etc.

- Para algunos de los patrones se utilizó una estructura de información extendida. Fundamentalmente este hecho se debe a que por un tema de alcance se profundizó más en algunos patrones que en otros. Por dicho motivo, pareció importante contar con una estructura extendida mediante la cual presentar información mas detallada.

- En una primera instancia se contempló la inclusión del concepto de "template" en el modelo conceptual. Con dicho término se buscaba representar el hecho de la variación que la arquitectura de microservicios podría tener en función del tipo (e-commerce, empresarial, etc) de aplicación que se desarrollara.

  Sin embargo, luego de analizar con mayor profundidad la variación de la arquitectura de microservicios en función del tipo de aplicación, se notó que el impacto que los objetivos puntuales de las aplicaciones tienen en el diseño de la arquitectura es muy bajo o nulo.

  Por dicho motivo se tomó la determinación de eliminar el término template del modelo conceptual.

- Desde etapas iniciales en el diseño de la solución se tuvo muy presente el tema de la extensibilidad. Esto se debe a lo amplio de la temática y el alcance reducido que se logra contemplar en este proyecto.

  Por estos motivos, se cree firmemente que sería de mucha importancia que la solución se pueda extender con el fin de que incrementalmente se pueda abarcar más contenido.

5 Desarrollo del prototipo

En este capítulo se presenta el proceso de desarrollo y características del prototipo implementado.

5.1 Descripción general

A partir de la propuesta de solución antes presentada, se decidió desarrollar un prototipo, cuyo objetivo principal fue validar dicha solución.

Teniendo en cuenta la cantidad de patrones y categorías obtenidos como resultado de la etapa de análisis presentada en el Capítulo 3, y contraponiendo los lineamientos expuestos como parte de la propuesta de solución con la capacidad (en términos de alcance) de este proyecto, en el siguiente párrafo se presenta el alcance concreto del prototipo implementado.

En primer término, se optó por incluir la lista completa de 49 patrones al catálogo presentado mediante la interfaz web, sobre los cuales se presenta, documentación general y la caracterización en el modelo conceptual diseñado.

Por otra parte, se tomó la decisión de que la generación de implementaciones de referencia y aplicaciones de ejemplo se centrara en dos patrones puntuales (el patrón Saga y el Circuit Breaker). Como consecuencia, se obtuvo un alcance acorde al proyecto, que permitió profundizar en mayor medida buscando aportar el mayor valor posible a los usuarios de la plataforma.

Concretamente, se brindan dos implementaciones de referencia diferentes con sus respectivas aplicaciones de ejemplo para cada uno de los patrones contemplados.

5.2 Tecnologías utilizadas

En está sección se presentan las tecnologías utilizadas tanto para el desarrollo del prototipo en sí, como para las implementaciones de referencia y aplicaciones de ejemplo.

5.2.1 Prototipo

Para el desarrollo de la interfaz web de la plataforma se decidió utilizar ReactJs, ya que brinda una serie de ventajas que facilitan el desarrollo de interfaces amigables y estéticas. Dentro de dichas ventajas se destaca el hecho que brinda la posibilidad de escribir código HTML y CSS dentro de objetos de Javascript de manera sencilla.

Específicamente, ReactJs se define como una librería de código abierto de Javascript desarrollada por Facebook que permite la construcción de interfaces de usuario de manera sencilla e intuitiva [105]. Está basado en un paradigma denominado "programación orientada a componentes" donde se propone que cada componente se implementa como una pieza reutilizable con la que el usuario interactúe [99].

Para el desarrollo del backend se utilizó NodeJs que se define como: un entorno de ejecución de código abierto para Javascript orientado a eventos asíncronos [27]. El principal fundamento por el cual se decidió utilizar está tecnología es la facilidad que provee para la implementación de APIs, sumado al hecho de que es una de las tecnologías más utilizadas actualmente para el desarrollo de este tipo de aplicaciones.

En lo que respecta al almacenamiento, se optó por MongoDB que se define como "una base de datos documental, lo que significa que almacena datos en forma de documentos tipo JSON" [100].

Aprovechando las características de la tecnología, la información de cada patrón se almacena como un objeto JSON el cual se compone de los siguientes atributos: nombre, categoría, autor, descripción, tipo e implementaciones.

Sumado a esto, se utiliza "Mongoose" [101] que es un paquete desarrollado para el modelado de bases de datos Mongo en ambientes desarrollados en Node.JS.

Además, como repositorio de información donde almacenar tanto las implementaciones de referencia como las diferentes aplicaciones de ejemplo se utilizó GitLab [102] que es un repositorio de archivos basado en GIT [103].

A continuación se pueden observar los repositorios utilizados para cada una de las implementaciones desarrolladas.

Aplicaciones de ejemplo:

- Circuit Breaker en Java → https://gitlab.fing.edu.uy/proyectogrado1/ejemplo-circuitbreaker-resilience4j
- Circuit Breaker en JavaScript → https://gitlab.fing.edu.uy/proyectogrado1/ejemplo-circuitbreaker-opossum
- Saga en Java → https://gitlab.fing.edu.uy/proyectogrado1/ejemplo-saga-eventuate
- Saga en JavaScript → https://gitlab.fing.edu.uy/proyectogrado1/ejemplo-saga-debezium

Implementaciones de referencia:

- Circuit Breaker en Java → https://gitlab.fing.edu.uy/proyectogrado1/circuitbreaker-resilience4j
- Circuit Breaker en JavaScript → https://gitlab.fing.edu.uy/proyectogrado1/circuitbreaker-opossum
- Saga en Java → https://gitlab.fing.edu.uy/proyectogrado1/saga-eventuate
- Saga en JavaScript → https://gitlab.fing.edu.uy/proyectogrado1/saga-debezium

Finalmente, para el despliegue de la aplicación se usó un servidor de MiNube-Antel provisto por la facultad. MiNube-Antel se define como: "Una plataforma que permite de forma fácil y confiable, comprar y administrar soluciones digitales en la nube para desarrollar tu negocio" [104].

La plataforma se encuentra disponible mediante la dirección:: http://179.27.98.103:8080/

5.2.2 Implementaciones de referencia y aplicaciones de ejemplo

Como se mencionó previamente, se desarrollaron dos implementaciones de referencia en tecnologías distintas para cada patrón y sobre cada una de ellas una aplicación de ejemplo.

Las implementaciones de referencia fueron desarrolladas con las mismas tecnologías que la aplicación de ejemplo asociada.

Patrón Saga

Para este patrón se utilizó, por un lado Java, SpringBoot [28], Eventuate Tram [30] y MySQL [52] (referenciada posteriormente como "implementación 1"), y por otro lado, NodeJs [27], Debezium [44], Apache Kafka [45], Apache Zookepeer [46] y MySQL [52](referenciada posteriormente como "implementación 2").

Implementación 1

Como lenguaje de desarrollo se utilizó Java dado que según los trabajos que se pudieron relevar es el lenguaje más utilizado para microservicios. Por ejemplo, Chris Richardson en [1].

Sumado a esto, el hecho de que se contaba con experiencia profesional en el lenguaje fue otra razón para la elección de este lenguaje.

Además, como framework de desarrollo de Java se utilizó Spring Boot junto con "Spring Tool Suite" como ambiente de desarrollo, lo cual facilitó el desarrollo y el testing de la aplicación. La elección de dichos frameworks se basó directamente en una recomendación de Chris Richardson en una comunicación directa que se tuvo con dicho autor. Spring Boot se define como: "Una infraestructura ligera que elimina la mayor parte del trabajo de configurar las aplicaciones basadas en Spring" [28].

También se utilizó el framework Eventuate para la implementación concreta del patrón Saga

en el lenguaje Java. Dicho framework es desarrollado por Chris Richardson (razón principal por la cual se decidió utilizar).

Eventuate se define como una plataforma que soluciona los problemas de la gestión de datos en sistemas distribuidos dado que, por un lado, permite mantener la consistencia de datos utilizando los sagas para ejecutar las transacciones y por otro colabora en la comunicación entre servicios utilizando eventos y mensajes.

Concretamente, esta plataforma encapsula y provee todo lo que es necesario para implementar un saga, desde una base de datos para el almacenamiento de transacciones (y su estado) hasta canales de mensajería.

Finalmente, como base de datos se utilizó MySQL. Al usar el framework Eventuate se introduce una limitación que reduce las posibles bases de datos a MySQL o PostgreSQL y por un tema de manejo de la tecnología se utilizó MySQL.

Cabe destacar que esta decisión fue extendida a todo el desarrollo para evitar diversificar las tecnologías en sobre manera.

Implementación 2

Como lenguaje de desarrollo se utilizó JavaScript dada la gran popularidad de este lenguaje en el desarrollo de software en general. Los motivos por los cuales se incluyó fueron dos, por un lado el interés de proveer una solución en una tecnología muy popular pero que aún no se ha aplicado masivamente a los microservicios y por otro la experiencia profesional en el lenguaje.

Como entorno de ejecución de JavaScript se utilizó NodeJs conjuntamente con el servidor ExpressJs (para implementar la API Rest). Está elección estuvo relacionada, nuevamente, a un tema de experiencia en el manejo de las herramientas.

Como librería específica para la implementación del patrón Saga se utilizó Debezium. Debezium es una plataforma de código abierto desarrollada por Red Hat que se encarga de capturar e identificar cambios en bases de datos de manera automática (configurable).

Además, como canal de mensajería se utilizó Apache Kafka que se define como una plataforma distribuida de transmisión de datos que permite publicar, almacenar y procesar flujos de registros, y suscribirse a ellos, en tiempo real.

Está elección se debe principalmente a que también es utilizado por el framework Eventuate de manera interna.

Por último, se utilizó Apache Zookepeer dado que la utilización de Apache Kakfa lo requiere. Está herramienta es un software desarrollado por Apache que lleva el control, registro de la información y configuración de todo lo relacionado a Apache Kafka: tópicos, particiones, etc.

Patrón Circuit Breaker

Para este patrón se utilizó, por un lado Java,Resilience 4j: [35] y MySQL [52] (referenciada posteriormente como "implementación 1"), y por otro lado, NodeJs [27], Opossum [40], y MySQL [52] (referenciada posteriormente como "implementación 2").

Implementación 1

Como lenguaje de desarrollo se utilizó Java con el framework Spring por las razones detalladas anteriormente.

Sumado a esto, como librería específica para la implementación del patrón Circuit Breaker se usó Resilience4j, que es una librería de código abierto para la implementación de Circuit Breaker diseñada para Java 8. Además, se define como una librería ligera debido a que solo depende de una biblioteca externa.

La elección de esta tecnología se debe a que un referente en la industria como lo es Netflix lo recomienda. Esto se puede apreciar mismo a través de la página oficial de Hystrix, la librería para Circuit Breaker que provee el mismo Netflix (la cual se dejó de desarrollar hace un tiempo).

Implementación 2

Como lenguaje de desarrollo se utilizó JavaScript con NodeJs por las mismas razones presentadas previamente.

Sumado a esto, se empleó la librería Opossum para el desarrollo específico de Circuit Breakers en JavaScript. Opossum es una librería de código abierto desarrollada por NodeShift (proyectos para Node.JS propiedad de Red Hat) diseñada para Node.js.

5.3 Proceso de implementación

En esta sección se presenta el proceso de implementación llevado a cabo en el proyecto. Dicho proceso abarca desde el desarrollo de las implementaciones de referencia hasta la implementación de las aplicaciones de ejemplo.

5.3.1 Estructura del proceso

El enfoque aplicado durante el desarrollo se puede categorizar como un enfoque ascendente (bottom-up), el cual constó de tres etapas bien definidas.

- Diseñar una realidad sobre la cual implementar las aplicaciones de ejemplo.
- Implementar las aplicaciones de ejemplo aplicando los patrones puntuales en las tecnologías dadas.
- Generalizar las aplicaciones de ejemplo hasta obtener las implementaciones de referencia.

En primera instancia, se diseñó una realidad la cual implementar como aplicación de ejemplo. Una vez logrado esto, se procedió a implementar dicha realidad aplicando los patrones puntuales. De esta manera se logró tener una aplicación real funcionando en la cual se tienen implementados los patrones contemplados.

Finalmente, se aplicó una generalización para obtener a partir de dichas aplicaciones las distintas implementaciones de referencia las cuales se brindaría.

5.3.2 Implementación de las aplicaciones de ejemplo

Dado que la complejidad de implementar los patrones de manera aislada sin tener una realidad concreta se tornó sumamente complicado. Surgió la idea de tomar un enfoque en el cual, primero se desarrollara una aplicación concreta utilizando las tecnologías y patrones dados para luego generalizar dicha implementación, logrando así obtener las estructuras deseadas.

Por este motivo, se diseñó una realidad la cual implementar. Si se observa desde una perspectiva funcional, se puede decir que la realidad utilizada para cada una de las diferentes

combinaciones patrón-tecnología implementadas es idéntica.

Concretamente, se definió un contexto en el cual se modela el típico sistema e-commerce donde se tiene un sistema de órdenes las cuales son despachadas a los consumidores finales mediante un sistema de envíos. Sin embargo, a los efectos de este proyecto se tomó una simplificación de esta realidad donde se contemplan únicamente dos microservicios para su representación: órdenes y envíos .

Por un lado, se tiene la gestión de las órdenes encapsulado en un microservicio y por otro la gestión de las fechas de envío en el restante.

- Gestión de Órdenes: Servicio que se encarga de ingresar nuevas órdenes, solicitar una fecha de envío y devolverle al cliente la fecha. Los atributos que tiene una orden son: identificador de usuario, nombre del ítem, monto, comentarios y estado. Este último atributo es usado principalmente en la implementación del patrón Saga.
- Gestión de Envíos: Servicio que se encarga del envío de los productos. Para poder obtener una fecha de envío se precisan tres atributos: identificador de usuario, ítem y monto. La idea general es en base a disponibilidad del ítem y si el usuario tiene crédito suficiente se devolverá una fecha de envío o no. En caso de que no se devuelve una fecha de envío se retornará el motivo.

Si bien, como se mencionó previamente la realidad implementada es conceptualmente idéntica, se desarrollaron 4 variantes de la misma. Dos aplicando el patrón Saga y dos el Circuit Breaker con los stacks de tecnología mencionados anteriormente.

5.4 Patrón Saga

El primer desafío a la hora de abordar la implementación del Saga fue tomar la decisión de cuáles de los dos enfoques existentes (Coreografía u Orquestación) se utilizaría. Después de analizar las dos opciones, pareció oportuno el contemplarlas ambas con el objetivo de que el valor brindado por el proyecto abarque la mayor cantidad de situaciones posibles.

Por este motivo, se provee de una implementación de cada una de estas dos alternativas.

Concretamente, la versión de Saga siguiendo el enfoque de orquestación se implementa con las tecnologías Java/Spring y el framework Eventuate mientras que en la versión con coreografía se utiliza Node.js con Debezium y Apache Kafka.

- Orquestación con Java sobre Spring Boot y el framework Eventuate: Para implementar el patrón según este enfoque es necesario delegar, a un componente externo y centralizado, la ejecución de las transacciones.

  En este caso concreto, el orquestador es el encargado de identificar las nuevas órdenes que se generan y solicitar una fecha de envió para estas. Luego, en base a la respuesta que obtiene del servicio encargado de la gestión de envíos, se actualizará el estado de la orden generada. Dicho estado puede tomar el valor de aprobado o rechazado.

- Coreografía con Node.js junto a Debezium y Apache Kafka: Este enfoque requiere la coordinación cuidadosa de los diferentes servicios involucrados, para lograr que el sistema nunca quede en estado ocioso. Al mismo tiempo, cada microservicio debe saber de antemano cuál será su rol en las diferentes transacciones (ya que en este enfoque son los mismos servicios quienes se coordinan).

  En el contexto específico de este proyecto, el microservicio encargado de la gestión de órdenes es el encargado de ingresar cada nueva orden a una base de datos específica. Por otro lado, existe un componente externo que se encarga de obtener los datos ingresados en esa base de datos y automáticamente depositarlos en un canal de mensajería específico.

  El servicio encargado de las fechas de envío deberá estar suscrito al canal donde se ingresan las órdenes de forma tal de saber cuándo una nueva orden se ingresa al sistema.

  Por último, el resultado producto del procesamiento del microservicio de fechas es depositado en otro canal de mensajería (diferente al de ingreso de órdenes) al cual estará suscrito el microservicio encargado de la gestión de órdenes.

5.4.1 Java/Spring + Eventuate

La implementación del patrón Saga con Java/Spring y Eventuate se realizó siguiendo el enfoque de orquestación, donde el componente orquestador es implementado utilizando el framework Eventuate. En la Figura 5.1 se puede apreciar el funcionamiento del framework.

Esta sección se organiza de la siguiente manera. En primera instancia se comienza introduciendo como se resolvió la comunicación entre los diferentes servicios. Luego, se detallan aspectos generales del framework, incluido que cosas este provee, cuales no y algunas restricciones que se introducen. Seguidamente, se detalla el rol del orquestador y como se implementó. Finalmente, se presentan aspectos generales sobre la implementación concreta de la Saga así como también se detalla la configuración de las variables involucradas en el proceso de implementación.

Para este caso se establece que el servicio de gestión de órdenes sea el encargado de iniciar la comunicación con el servicio de fechas en pos de obtener una fecha de envío para una orden específica. Además, se mapea la respuesta proveniente desde el servicio de fechas a través de un campo en la orden denominado “estado” el cual toma los valores pendiente, rechazado, aprobado dependiendo de la misma.

En este contexto, dado que se aplicó el enfoque de orquestación, se da la particularidad que los dos servicios nunca se ven en la necesidad de comunicarse directamente el uno con el otro si no que las comunicaciones se producen por intermedio del componente orquestador.

La utilización del framework Eventuate permitió contar con una serie de funcionalidades concretas, al mismo tiempo que introdujo algunas restricciones sobre la implementación.

Por un lado, brinda total abstracción al desarrollador en lo que respecta a la sincronización

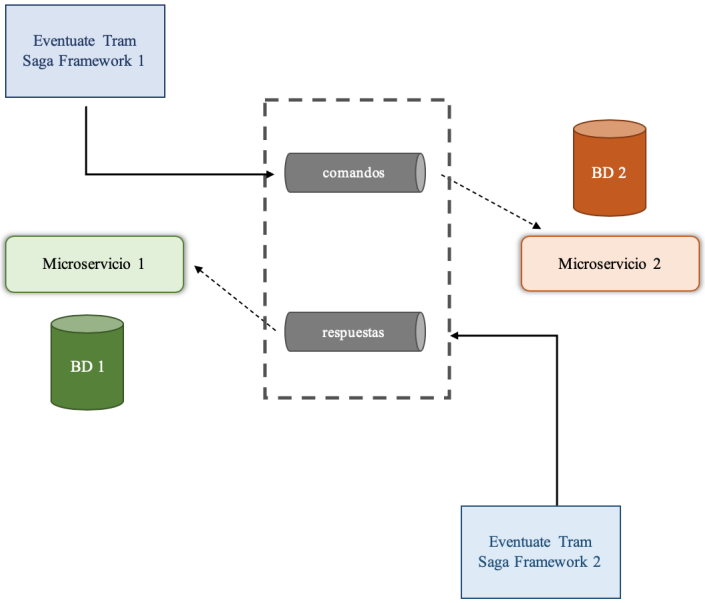

Figure 5.1: Eventuate

de los procesos (emisores y receptores) entre los diferentes servicios y a la interacción con el broker de mensajería. De esta forma, los desarrolladores se enfocan únicamente en la lógica de negocio. Por esta razón, cada servicio tiene total independencia y abstracción del otro ya que ni siquiera tienen interacción directa mediante los canales de mensajería.

A su vez, implementa la comunicación entre los dos servicios se a través de canales de mensajería asíncrona implementados mediante el servidor de mensajería Apache Kafka, por lo cual toda la complejidad referente a estos aspectos queda por fuera de la responsabilidad de los desarrolladores.

Sumado a esto, como parte de las herramientas para la facilitación de la comunicación entre los diferentes servicios Eventuate provee de dos canales de comunicación uno para el envío de comandos (requieren de la ejecución de una función) y otro para la devolución de dichas ejecuciones. Los comandos en el contexto de esta tecnología son la forma en que un servicio explicita qué operación quiere invocar desde el otro. Por lo cual el servicio de fechas, está esperando que ingresen solicitudes en forma de comando y responderá a este pedido con una respuesta (afirmativa o negativa) también en forma de comando.

Concretamente en el siguiente código se puede apreciar la definición de una función que interactúa con un servicio externo.

```
private CommandWithDestination operacion(CrearSagaData data) {
    return send(new ReserveCreditCommand())
        .to("segundoServicio")
        .build();
}
```

En el caso particular que se viene tratando sería de esta manera que se define la función que intenta obtener la fecha de envío para asociarla a la orden. Se aprecia claramente que las funcionalidades provistas por el framework simplifican de manera considerable el desarrollo resultando en un código simple y legible.

Sin embargo, conjuntamente con los beneficios que aporta Eventuate genera dependencia desde el punto de vista tecnológico ya que para poder implementar el patrón Saga en esta tecnología es necesario que todos los participantes de la saga la utilicen.

Además, es claro apreciar que la lógica de cambiar los estados de la orden dependiendo de la devolución del servicio fecha es parte de la lógica de negocio del caso implementado por lo cual no forma parte de las herramientas que Eventuate provee.

Por otra parte, una interrogante que surge es qué rol cumple el orquestador y cómo Eventuate resuelve este desafío.

El orquestador es el encargado de efectuar la comunicación necesaria para obtener una respuesta desde el servicio de fechas e impactar dicho resultado en el estado de la orden propiedad del otro servicio.

Más precisamente, Eventuate resuelve esto proveyendo funcionalidades que permiten configurar la saga de manera sencilla logrando abstraer al programador del manejo de eventos y cambios de estados logrando que estos aspectos sean “invisibles” para quien lo implementa.

A continuación se puede observar como se definiría la saga en el servicio de ordenes:

```
public class CrearSaga implements SimpleSaga<CrearSagaData> {
private SagaDefinition<CrearSagaData> definicionSaga =
    step()
        .invokeLocal(this::primeraOperacion)
        .withCompensation(this::operacionCompensatoria)
    .step()
        .invokeParticipant(this::segundaOperacion)
    ...
    .step()
        .invokeLocal(this::terceraOperacion)
    .build();
}
```

NOTA: El código presentado surge producto de la generalización aplicada a la implementación

del caso tratado.

En este ejemplo se puede apreciar la simplicidad con la que Eventuate permite declarar una saga. En este caso se define una secuencia ordenada de transacciones cuya ejecución depende del resultado de las demás y se definen transacciones compensatorias las cuales son configuradas para dispararse en caso de que la transacción original (a la cual compensa) falle.

En el contexto puntual en el cual se viene trabajando, se definiría la operación de crear una Orden en estado "pendiente" con la sentencia

```
.invokeLocal(this::primeraOperacion)
```

la cual tiene como compensación una transacción definida mediante

```
.withCompensation(this::operacionCompensatoria)
```

que modifica el estado de dicha orden a "rechazado" en caso de que falle la obtención de la fecha de envío desde el servicio de fechas, lo cual se produce por intermedio de la transacción definida en

```
.invokeParticipant(this::segundaOperacion)
```

Si la obtención de la fecha es exitosa se continúa con la ejecución de la secuencia definida a través del siguiente *step()* que en este caso particular sería la acción de poner la orden en estado "aprobado".

En lo que respecta a la implementación concreta de la Saga, cabe destacar el hecho de que en el proyecto se implementa una sola, es decir, la operación que involucra a más de un servicio que se implementa mediante sagas es solamente la de solicitar una fecha de envío. Por lo general el concepto de saga no se implementa como ente único, sino que se define una saga para cada posible tipo de operación.

El servicio de ordenes únicamente se encargará de crear nuevos pedidos y registrar dicho evento en una base de datos. Sin tener que preocuparse por la coordinación con el otro servicio para la obtención de una fecha de envío.

En particular, las órdenes son creadas y almacenadas en la base de datos en un estado pendiente, estado que tomará distintos valores dependiendo del resultado de la interacción con el servicio de fechas. Si se obtiene una respuesta afirmativa (lo cual implica que hay una fecha disponible para su despacho) dicho estado cambiará desde pendiente a aprobado y en caso contrario tomará el valor de rechazado.

Por último, a los efectos de poder comprobar el funcionamiento del saga en caso de éxito o fracaso de la petición al servicio de fechas, se implementa la lógica del servicio proveedor

de fechas de forma tal que se tenga control sobre las fallas del mismo. Concretamente, se definió que para ciertos usuarios la respuesta fuera negativa y de esta forma poder probar las transacciones de compensación. Sumado a esto, la definición de tiempos de espera límites en el servicio que espera por la respuesta (órdenes) permite que ante una eventual no disponibilidad del servicio de fechas las órdenes puedan fallar y dispararse la compensación al igual que en el caso de fracaso.

5.4.2 Node.js + Debezium + Kafka

Esta implementación sigue el enfoque de coreografía explicado previamente en el cual no existe un componente orquestador que centralice y gestione la comunicación entre servicios. En cambio, cada servicio se encarga de gestionar la comunicación con los demás.

Al igual que la implementación con Java Spring y Eventuate, la realidad que da contexto es la misma que se explicó anteriormente donde los dos actores principales son el servicio de órdenes y el de fechas. En la Figura 5.2 se puede apreciar la estructura de la solución.

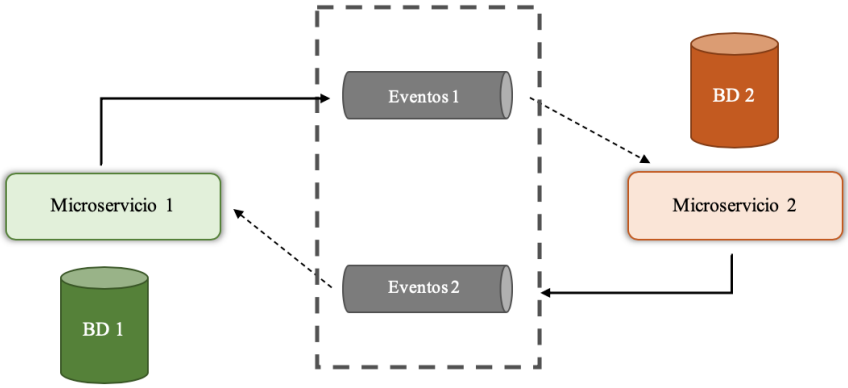

Figure 5.2: Estructura de la solución

Esta sección se organiza de la siguiente manera. En primera instancia se detalla la imposibilidad de utilizar un framework especifico como en el caso anterior. Luego, se destacan los pilares sobre los cuales se diseño la solución. Posteriormente, se presentan algunos aspectos generales como: implementación del patrón OUTBOX, canales de mensajería y asincronía en las comunicaciones. Finalmente, se destacan las decisiones tomadas sobre algunos aspectos generales.

A diferencia del caso anterior no se utilizó un framework ya que, no fue clara la existencia de uno concreto para las tecnologías utilizadas. Por este motivo, se decidió implementar el patrón únicamente utilizando el lenguaje Javascript. Este hecho provoca que no se genere

dependencia entre los distintos servicios en lo que respecta a la tecnología utilizada ya que no existe ese componente tecnológico que fuerce a la utilización de una tecnología específica. Incluso el desarrollo del patrón fue ideado para que el acoplamiento entre las distintas tecnologías sea bajo permitiendo que la selección de las tecnologías esté lo menos condicionada posible.

La implementación está basada en dos grandes aspectos, por un lado, en la definición teórica del patrón desde la cual se relevaron las funcionalidades que se deberían implementar y por otro, el funcionamiento del framework Eventuate utilizado anteriormente.

En primer lugar, fue necesario profundizar en el funcionamiento interno de Eventuate ya que en el fondo se implementaron las principales funcionalidades del framework pero en Javascript. Producto de este hecho antes mencionado, surgió el estudio de cómo Eventuate implementa la lógica encargada de la coordinación de los servicios. Desde que se inserta un registro en la base de datos (se crea una orden en estado pendiente) y este evento dispara (de forma transparente para el desarrollador) que se ejecute un función del lado del servicio de fechas y cómo la respuesta desencadena una reacción en el servicio de órdenes (modificar la orden a estado aprobado o rechazado dependiendo de dicha respuesta).

Como consecuencia de dicho estudio se comprobó que internamente Eventuate implementa otro patrón, el cual se incluye entre los relevados y se denomina Transactional OUTBOX. Este patrón se define básicamente, como una base de datos independiente donde se gestiona el manejo de los eventos del servicio. Por lo cual, cuando un microservicio genera un nuevo registro que disparara la interacción con un segundo, se inserta la información necesaria en esta tabla (OUTBOX). Desde esta tabla el segundo servicio extraerá los datos y ejecutará la operación pertinente.

Para la implementación de la lógica antes mencionada se utilizó la librería Debezium [44], la cual es utilizada para procesar los eventos insertados en la tabla OUTBOX y depositarlos en el broker de mensajería. Cada vez que una orden es generada e insertada en la tabla OUTBOX en estado pendiente Debezium se encarga de procesar dicho evento enviándolo por el canal adecuado. Al mismo tiempo cabe destacar que desde el servicio de fechas se podría aplicar cualquier mecanismo para la recepción de mensajes, incluso se podría implementar con Debezium, pero no es un requerimiento excluyente. Este hecho es de importancia ya que logra independencia a la hora de seleccionar los stack tecnológicos.

Por otro lado, para el manejo de los canales de comunicación se utiliza el broker Apache Kafka [45] tal cual utiliza Eventuate y al igual que en su implementación se implementan dos canales, uno para el envío de comandos y otro para la recepción.

Sumado a esto, para lograr la asincronía en la coordinación de los diferentes procesos se emplea un mecanismo de asignación de identificadores de correlación. Este hecho se debe a que cuando la respuesta a una petición de una fecha de envío para una orden en específico es recibida, el servicio de órdenes sea capaz de identificar la orden en cuestión y actuar en consecuencia.

Este hecho logra que el servicio de órdenes no dependa de la ejecución del servicio de fechas ya que es capaz de generar nuevas órdenes en estado pendiente y seguir ejecutando sus operación sin necesidad de esperar por una respuesta, y cuando una es recibida puede asociarla a la orden que la genero sin inconvenientes.

En la Figura 5.3 se puede apreciar un esquema del funcionamiento de la solución antes presentada.

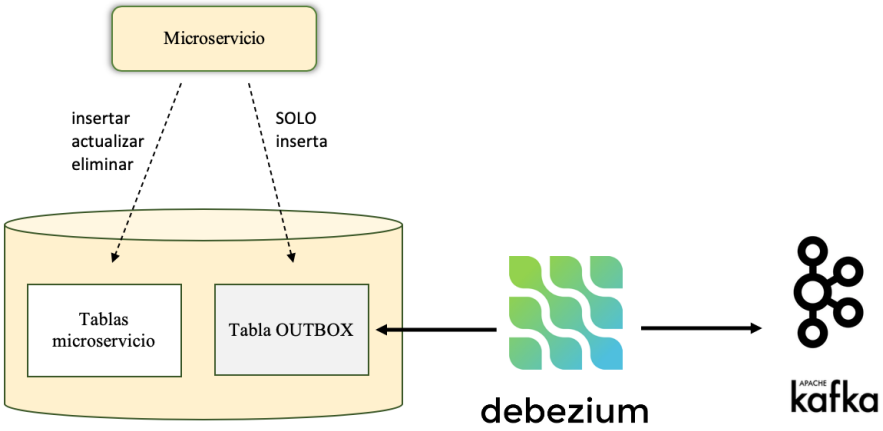

Figure 5.3: Esquema del funcionamiento con la tabla OUTBOX

Finalmente, de forma similar a la implementación con Java/Spring se utilizan escenarios de prueba donde se fuerza al servicio de fechas a fallar de manera eventual (usuarios específicos) para, de esta manera, probar todos los posibles casos en el servicio de órdenes. También, se implementan tiempos de espera en las órdenes de modo tal que ante una eventual no disponibilidad del servicio de fechas las órdenes puedan ser rechazadas sin bloquearse.

5.5 Circuit Breaker:

En la Figura 5.4 se puede apreciar una representación del funcionamiento del patrón.

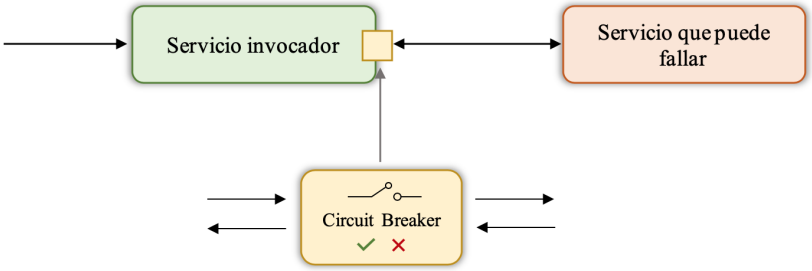

Figure 5.4: Circuit Breaker

En este caso, la implementación del patrón requiere únicamente del microservicio el cual inicia la comunicación (gestión de órdenes) ya que es desde dicho microservicio desde donde se pretende controlar las fallas del sistema.

Incluso, el otro microservicio (gestión de fechas de envío) ni siquiera tiene conocimiento de la existencia del Circuit Breaker que está controlando las invocaciones entrantes a el mismo.

Como consecuencia de lo antes mencionado, es claro notar que los servicios son independientes desde todo punto de vista ya que la implementación es totalmente transparente hacia el entorno. A diferencia del patrón Saga, no se necesita de componentes externos tales como canales de mensajería o base de datos adicionales.

Las invocaciones sobre las cuales se quiere controlar la ocurrencia de fallas (y por lo tanto se ejecutan mediante el Circuit Breaker) son las que realiza el microservicio de órdenes hacia el microservicio de gestión de fechas de envío. Al encapsular las llamadas se logra tener un control de las invocaciones sobre el microservicio de fechas permitiendo no invocarlo innecesariamente en situaciones de no disponibilidad.

5.5.1 Spring + Resilience4j

Como se mencionó previamente, una de las tecnologías seleccionadas para la implementación de este patrón fue la librería Resilience4j con el lenguaje Java en el framework Spring.

Está tecnología brinda al usuario la capacidad de no invocar a los servicios los cuales se quiere controlar. Como consecuencia, todas las operaciones que dicho servicio brinda se invocan mediante el Circuit Breaker a través de primitivas especificas que la librería provee.

En el caso puntual de este proyecto, se busca controlar las fallas ocurridas en el servicio de fechas específicamente sobre la operación de obtener una fecha de envío. Por este motivo la invocación desde el servicio de órdenes a esta operación se realiza mediante las funcionalidades provistas por Resilience4j. En el código a continuación se presentan de forma explícita los aspectos antes señalados.

```
Supplier<String> supplier =
    CircuitBreaker.decorateSupplier(circuitBreaker,

Try<String> result = Try.ofSupplier(supplier);
    if (result.isSuccess()) {
        return new ResponseEntity<Object>( result.get(),
                                           new HttpHeaders(),
                                           HttpStatus.OK);
    } else {
        return new ResponseEntity<Object>(
            "Circuit Breaker: " + circuitBreaker.getState(),
            new HttpHeaders(),
            HttpStatus.INTERNAL_SERVER_ERROR
        );
    }
```

Mediante la definición del "Supplier" (concepto introducido por la librería) se genera la llamada a la operación Solicitar fecha de envío del servicio de fechas decorada a través de la operación "decorateSupplier" provista por Resilience4j. En esta llamada, a su vez, se explicita qué configuración de Circuit Breaker se utilizará, aportando la misma al decorador antes mencionado. Acto seguido, mediante sintaxis específica de la librería ("Try.ofSupplier") se procesa la respuesta del otro servicio tomando diferentes acciones en función de la misma.

Por otra parte, dado que para la implementación de este patrón es necesario definir ciertos umbrales los cuales guiarán el comportamiento del circuito, Resilience4j provee, de manera sencilla, soporte a la configuración de estos.

Específicamente, entre otros, permite la configuración de:

- **Ratio de fallas:** Parámetro mediante el cual se estipula el límite de fallas toleradas por el circuito antes de transicionar a estado abierto. Una vez alcanzado este estado se comienza a rechazar las innovaciones automáticamente sin consultar al servicio destino ya que se asume incapacidad de responder.
- **Tiempo de espera:** Límite de tiempo tolerado como espera a una invocación puntual. Si este umbral es superado se considera a la invocación como una falla.
- **Ratio de tiempo de espera en las llamadas:** Parámetro mediante el cual se estipula el tiempo de espera límite ante una invocación al otro servicio. Una vez superado este tiempo el circuito pasa a estado abierto adoptando el mismo comportamiento descrito anteriormente.
- **Tiempo en estado abierto:** Este parámetro establece el límite de tiempo durante el cual el circuito permanece en estado abierto sin invocar al servicio destino esperando por la recuperación del mismo.

- **Transición de abierto a semi abierto automático:** Variable booleana que indica si el circuito posee la capacidad de transicionar de manera automática desde el estado abierto a el estado semi abierto.
- **Llamadas permitidas en estado semi abierto:** Establece el número de invocaciones al servicio destino que se permitirán mientras que el circuito se encuentra en estado semi-abierto.
- **Mínimo número de llamadas:** Número mínimo de llamadas para empezar a calcular las estadísticas. Si no se llega a este volumen de llamadas el Circuit Breaker no se abrirá, independientemente del éxito o fracaso que hayan tenido.

En el proyecto se instancian estos parámetros antes mencionados para generar un contexto de prueba y de esta forma implementar el patrón. Un aspecto el cual es importante destacar es que la instanciación de un Circuit Breaker (configuración específica de estos parámetros) no está limitada a una por servicio si no que se podrían definir distintas configuraciones del Circuit Breaker para diferentes operaciones y servicios de destino.

Como se mencionó en líneas anteriores, en el caso del proyecto se instancia un Circuit Breaker para la operación de obtener fechas de envío que se implementa en el servicio de fechas.

En el código presentado a continuación se puede apreciar la configuración específica implementada:

```
private CircuitBreakerConfig circuitBreakerConfig =
    CircuitBreakerConfig.custom()
        .failureRateThreshold(50)
        .waitDurationInOpenState(Duration.ofMillis(10000))
        .slowCallDurationThreshold(Duration.ofSeconds(2))
        .permittedNumberOfCallsInHalfOpenState(3)
        .minimumNumberOfCalls(10)
        .build();
```

Conjuntamente con la definición de estos parámetros se estableció, a través de la implementación del servicio de fechas, escenarios en los cuales se producen fallas. Todo esto con el objetivo de controlar la devolución de la operación obtener fecha de envío y así lograr comprobar el funcionamiento del patrón ante todas las posibles situaciones.

Nuevamente, al igual que en casos anteriores, se estableció que el servicio de fechas fallará ante la petición de cierto usuario específico y de esa forma poner en práctica el funcionamiento de los umbrales correspondientes.

5.5.2 Node.js + Opossum

En el caso de la implementación del patrón con Javascript, se decidió utilizar la librería Opossum como bien se comentó previamente.

Está implementación es similar a la anterior en muchos aspectos tales como: la posibilidad de definir múltiples Circuit Breaker dependiendo de la operación que se desee controlar, el invocar la operación por intermedio del circuito delegando la responsabilidad de devolver una respuesta o la definición de parámetros mediante los cuales se instancia el Circuit Breaker.

Sin embargo, Opossum si bien define umbrales similares a Resilience4j, la nomenclatura varía:

- **Tiempo de espera:** Límite de tiempo tolerado como espera a una invocación puntual. Si este umbral es superado se considera a la invocación como una falla.
- **Tiempo de espera para el restablecimiento:** Variable booleana que indica si el circuito posee la capacidad de transicionar de manera automática desde el estado abierto a el estado semi abierto.
- **"RollingCountTimeout":** Ventana de tiempo en la cual se lleva la estadística del Circuit Breaker.
- **Capacidad:** El número de llamadas concurrentes permitidas. Si se supera este umbral el resto serán rechazadas.
- **Porcentaje de error:** El porcentaje de llamadas fallidas toleradas, si se supera este ratio, el Circuit Breaker transiciona a estado abierto.
- **Volumen:** Número mínimo de llamadas para empezar a calcular las estadísticas. Si no se llega a este volumen de llamadas el Circuit Breaker no se abrirá, independientemente del éxito o fracaso que hayan tenido.

Por otra parte, a los efectos de este proyecto, y al igual que en el caso anterior, se definió una configuración específica para el control de las fallas de la operación obtener fecha de envío desde el servicio de fechas.

A continuación se puede apreciar la configuración resultante:

```
const options = {
    timeout: 3000,
    errorThresholdPercentage: 50,
    resetTimeout: 10000,
    volumeThreshold: 10
}
```

En el caso de esta librería, la forma de invocar a un nuevo circuit breaker se da mediante la

siguiente sintaxis:

```
const breaker = new CircuitBreaker(funcionQuePuedeFallar,
                                   options)
```

Como se puede apreciar al momento de la instanciación del Circuit Breaker se estipula la operación la cual se quiere controlar y los parámetros de configuración para dicha instancia, por lo cual el definir múltiples instancias de este patrón dependiendo de la operación objetivo es un hecho de fácil implementación.

La forma de ejecutar el Circuit Breaker definido se detalla en el siguiente código:

```
 breaker
      .fire()
      .then((body) => {    })
      .catch((err) => {
});
```

Por último, para la verificación de la implementación se utilizó el mismo enfoque aplicado anteriormente en el proyecto el cual consta de definir umbrales específicos para instanciar un Circuit Breaker que controle las ocurrencias de fallas sobre la operación obtener fecha envío. Como consecuencia el circuito se define en el servicio de órdenes y mediante el mismo se invoca al servicio de fechas.

Al igual que en casos anteriores, se controlan las fallas de la operación en cuestión a través de la definición de falla ante un usuario específico de modo tal de poder manifestar el comportamiento del Circuit Breaker implementado en todos los escenarios posibles.

5.6 Proceso de generalización:

Luego de aplicar cada patrón en las aplicaciones de ejemplo, se obtuvieron las implementaciones de referencia.

En base al código generado luego de aplicar cada patrón en su respectiva aplicación de ejemplo, se fue estandarizando el código.

En primer lugar se eliminaron todas los atributos propios de la idea de negocio de la aplicación. Es decir, se eliminó el concepto de Orden y Fecha Envío, conjuntamente con todos los parámetros asociados.

En el caso del servicio de órdenes, se eliminaron los atributos de esta clase (ítem, monto, userId, comentarios) y se dejó la libertad para que el usuario que desee utilizar esta estructura de código agregue lo que su negocio requiera.

En el caso de servicio de Fecha de envío, se eliminaron los atributos requeridos para poder obtener una nueva fecha.

Por otro lado, se introdujeron comentarios que servirán de guía para que cualquier usuario que decida utilizar alguna implementación de referencia tenga claro qué partes del código debe personalizar.

Por último, se modificaron los nombres de los archivos y de los conceptos asociados, de tal manera que el usuario defina el nombre del servicio y automáticamente el código se actualice apropiadamente.

5.7 Problemas encontrados

Un problema recurrente durante todo el desarrollo del proyecto fue el de la escasa documentación de las tecnologías de referencia. Si bien se hizo un esfuerzo considerable en seleccionar stacks que sean ampliamente utilizados, se notó la falta de madurez de este tema en lo que respecta a la escasez de proyectos con arquitectura de microservicios implementados. Aún más si se considera aquellos de código abierto a los cuales se puede tener acceso para tomar de referencia.

Este hecho provocó que la dificultad en la implementación del proyecto sea de una complejidad elevada. Tal fue el caso que en la implementación del Patrón Saga con Java Spring y Eventuate después de experimentar dificultad en el manejo del framework se tomó contacto con su creador, Chris Richardson, a través de un canal de Slack de la comunidad que rodea al framework.

Afortunadamente, se contó con el apoyo y consejo del mismo Richardson el cual sirvió de guía y aportó gran valor en la implementación del saga con Eventuate. Tal fue así que se pudo obtener un conocimiento profundo en la herramienta lo que derivó en que, en una etapa posterior del desarrollo del proyecto, se tuvo la capacidad de implementar muchas de sus funcionalidades en otra tecnología (Javascript).

Por otra parte, un hecho similar se dio con la utilización de la plataforma Debezium sobre la cual, en una primera instancia, no se encontró gran referencia y documentación. No obstante, luego de investigar se dio con una conferencia brindada por un arquitecto de RedHat (organización que desarrolla la librería) en el año 2019 en la cual se exponen las principales características y ventajas que esta posee.

Fue de esta manera que se contactó a dicho orador llamado Mauro Vocale, el cual con buena disposición colaboró en el entendimiento del poder que está librería provee.

5.7.1 Decisiones tomadas en la implementación

- La decisión de que los requerimientos de la aplicación de ejemplo sea la misma para cada par patron-tecnología tiene su fundamento en que se considera de sumo interés el hecho de aplicar las diferentes combinaciones implementadas sobre una realidad única, lo cual permite exponer con mayor claridad las distintas características que cada una

de las tecnologías y patrones poseen.

- El hecho de desarrollar la aplicación de ejemplo con solo dos servicios se da producto de que para los requerimientos y objetivos perseguidos por este proyecto se considera suficiente contemplar la interacción entre solo dos microservicios. Aumentar la cantidad de entes interactuando implicaría un nivel de complejidad significativamente superior al valor aportado.

- En cuanto a implementación del patrón Saga se tuvo que decidir qué enfoques (coreografía u orquestación) utilizar en las implementaciones de referencia. Después de analizar las dos opciones, pareció oportuno el contemplarlas ambas con el objetivo de que el valor brindado por el proyecto abarque la mayor cantidad de situaciones posibles.

5.7.2 Decisiones tomadas en la selección de los stacks tecnológicos

- Eventuate: Chris Richardson, autor de referencia en Patrones de Microservicios y en este proyecto, es el creador del framework que provee funcionalidades para la implementación de diferentes patrones, dentro de los cuales se incluye Saga. Por este motivo, resultó oportuno la utilización de dicho framework como parte de la solución.

- Saga con Javascript: Para implementar el patrón Saga con Javascript se buscó incluir librerías que faciliten la implementación. En primera instancia surgió una librería llamada redux-saga que tiene un promedio de 670 mil descargas semanales [42] y se define como una librería que apunta a facilitar el manejo de los efectos secundarios de una aplicación web desde el lado del cliente [43]. Estos efectos secundarios comprenden la obtención de datos y el acceso al caché del navegador.

  Sin embargo, si bien el nombre de está librería hizo pensar que se podría utilizar para el propósito del proyecto, luego de indagar con mayor profundidad quedó claro que el foco de Redux-Saga no estaba alineado con el objetivo perseguido: poder solucionar el problema de la consistencia de datos entre microservicios provocado por la interacción de múltiples servicios cada uno con su base de datos independiente.

  Curiosamente, no se pudo encontrar ningún framework, plataforma o librería que simplifique la implementación de Sagas (enfocados de la manera antes mencionada), por lo cual se decidió implementar este patrón con javascript estándar sin el apoyo de este tipo de herramientas. Al no contar con una herramienta que guíe la implementación fue necesario idear una solución desde cero. La resolución de este desafío se basó mayormente en las definiciones teóricas relevadas sobre el patrón y sumado a esto, se tomó como referencia el funcionamiento del framework Eventuate (estudiado previamente) para de esta manera tener una base en la cual apoyarse para la resolución del problema.

- Netflix es uno de los referentes en lo que respecta al mundo de los microservicios.

  Esto se debe a que, junto a otros factores, dentro de la organización han desarrollado un espacio denominado “Centro de Software Libre” (Netflix OSS) [4] donde se provee

a la comunidad de varias herramientas de diferentes índoles: plataformas, frameworks y librerías. Estas herramientas abarcan gran cantidad de temas dentro de los cuales se destacan varios de los desafíos que aporta la utilización de la arquitectura de microservicios. En particular, se encuentra una librería para la implementación del patrón Circuit Breaker enfocada en sistemas desarrollados utilizando Java con Spring denominada Hystrix.

Esta se define oficialmente en [47] como "Una librería diseñada para el aislamiento de puntos de acceso a sistemas remotos, servicios y librerías de terceros, detener las fallas en cascada y permitir la resistencia en sistemas distribuidos complejos donde la falla es inevitable".

Sumado al hecho de que el desarrollador de esta librería sea uno de los grandes referentes de la industria en microservicios, se destaca que muchos otros actores importantes hacen referencia a esta como la más utilizada para la implementación del patrón Circuit Breaker.

Tal es así que, por ejemplo, Spring ofrece una implementación del Circuit Breaker utilizando Hystrix [32]. Al mismo tiempo, otro de los autores de referencia del proyecto, Martin Fowler, en su blog [33], la define como: "Una sofisticada herramienta para lidiar con la latencia y la tolerancia a fallas en sistemas distribuidas". También es recomendada por Chris Richardson en [34] y por Sam Newman en su libro [6].

Sin embargo, en Noviembre de 2018 Netflix anunció que dejaría de ser desarrollada para pasar a ser únicamente mantenida [36]. No obstante, desde Netflix oficial se estipula que para sus nuevos proyectos donde se hubiese requerido la utilización de Hystrix se comenzaría a utilizar otra librería llamada Resilience4j [35] y se recomienda a la comunidad seguir la misma línea.

- En el caso de la implementación del patrón circuit Breaker utilizando Javascript se decidió utilizar Node.js como entorno de ejecución. Por este motivo, se investigó sobre la existencia de alguna librería para la implementación del patrón en el gestor de paquetes de NodeJs (NPM [37]).

  NPM contiene un buscador de paquetes, donde cada uno de estos tiene asociadas palabras claves que facilitan su búsqueda. A su vez cada paquete tiene una valoración dependiendo de tres variables: popularidad, mantenimiento y calidad. Por este motivo, se realizaron dos búsquedas, una bajo el término "Circuit Breaker" y otra con el término "Hystrix".

  La primera arrojó 92 paquetes como resultado y la segunda 42. La librería "Opossum" fue la más popular dentro de la lista contemplada. No obstante, también se tuvo en consideración una librería denominada Cockatiel [41], y el motivo principal fue que a diferencia de Opossum, fue lanzada hace solo seis meses y ya cuenta con Microsoft como uno de sus usuarios de mayor renombre. Sin embargo, debido a la madurez y la cantidad de documentación disponible se optó por utilizar Opossum como librería de

Javascript para la implementación del patrón Circuit Breaker.

6 Conclusiones y trabajo futuro

En este capítulo se presentan las conclusiones del trabajo realizado y posibles mejoras que podrían ser realizadas como trabajo a futuro.

6.1 Resumen

El objetivo general del proyecto fue la propuesta y diseño de una solución que asista en la implementación de la arquitectura de microservicios guiada por patrones.

En la primer etapa se identificaron las funcionalidades que una solución de este tipo debería proveer. Para lograrlo se realizó una investigación sobre los distintos patrones de microservicios existentes con el objetivo de relevar información sobre estos y poder definir un alcance adecuado para el trabajo. Está actividad generó como resultado una gran cantidad de patrones y categorías, los cuales fueron refinados con el objetivo de definir un alcance acorde con un trabajo de estas características.

Como resultado general de la etapa de análisis se obtuvo un modelo conceptual en el cual se instanciaron los patrones resultantes con el objetivo de uniformizar los conceptos más importantes que los rodean.

Basado en dicho análisis se presentó una propuesta de solución la cual satisficiera las funcionalidades identificadas.

Luego del proceso de implementación de este proyecto se logró brindar la implementación de un prototipo que valide la propuesta de solución presentada. Concretamente, el resultado de este proyecto provee distintas funcionalidades que asisten a los usuarios desde la etapa más temprana del diseño de la arquitectura hasta la generación de implementaciones de referencia (estructuras de código) asociadas a cada patrón. Esto se contempla conjuntamente con aplicaciones de ejemplo en donde se tienen dichas estructuras implementadas en el contexto de una aplicación real y funcional.

La solución se desarrolló utilizando como base tecnologías modernas. En particular, Node.Js para la implementación del servidor o backend y ReactJS (biblioteca de Javascript) para el desarrollo de la interfaz de usuario o frontend. Además, como repositorio de datos se utilizó la base de datos no relacional MongoDB y se integró la plataforma con el repositorio git-Lab donde residen las estructuras de código y aplicaciones de ejemplo que se brindan a los usuarios.

6.2 Conclusiones

La arquitectura de microservicios, a pesar de no ser un concepto introducido tan recientemente, aún carece de una estandarización y buenas prácticas adecuadas. Esta carencia se acrecienta cuando a patrones de microservicios específicos se refiere.

Esto se aprecia, por ejemplo, en la cantidad de patrones existentes con nomenclaturas tan heterogéneas y que muchos terminan proponiendo soluciones idénticas. Además, la existencia de una gran cantidad de fuentes de información contrastado con la falta de estandarización hace que se dificulte relevar información.

Por otra parte, la poca documentación existente en referencia a algunos de los patrones relevados denota que aún no se han establecido convenciones en relación a los patrones de microservicios que permitan tener un conjunto claro de patrones a través de los cuales implementar la arquitectura. En contraposición, con los patrones asociados a la programación orientada a objetos en donde la estandarización existente produce que la incertidumbre a la hora de seleccionar los patrones a utilizar sea realmente baja.

Incluso, esta reflexión se ve reforzada por el hecho de que en trabajos relacionados se concluyen ideas similares a la presentadas anteriormente.

A nivel general, se cree que el resultado de la etapa de análisis fue muy productivo. Se pudieron relevar una gran cantidad de patrones y refinarlos en base a criterios definidos, se diseñó un modelo conceptual en el cual instanciar a dichos patrones permitió tener una mayor claridad en la información y en base a dicho trabajo se pudo identificar una serie de funcionalidades que una plataforma de este estilo debería proveer.

Una vez culminada la etapa de identificación de funcionalidades tuvo lugar el diseño de la solución presentada para la obtención de los objetivos planteados. Si bien se realizó el ejercicio de plantear una solución completa en base a las funcionalidades relevadas anteriormente, considerando el contexto de un proyecto de grado y un equipo integrado por dos personas, se decidió limitar la propuesta de solución planteada como ideal y enfocar el trabajo en una parte.

Además, el hecho de tener que limitar dicho alcance obligó a encarar la propuesta de solución de una forma más abstracta y centrarse en los aspectos más importantes. Como resultado, se llegó a la conclusión de que el aspecto más importante es el de entregar a los usuarios código concreto sobre los patrones que genere valor de forma rápida y que conlleve el mínimo esfuerzo posible para adaptar ese código a cada realidad en donde se utilice.

Aunque se optó por presentar una propuesta de solución que se centre principalmente en este aspecto y aporte valor a través de las estructuras de código y aplicaciones de ejemplo brindadas, se consideró importante la interacción con el usuario y el disponer de una base de información procesada la cual puede ser tomada como referencia por usuarios interesados en implementar esta arquitectura.

Para la implementación se evaluaron diferentes tecnologías y se pudo corroborar que no es del todo verdadera la afirmación de que la arquitectura de microservicios aporta total independencia tecnológica entre los diferentes servicios que componen a una aplicación. Esta conclusión se basa en que se pudo constatar que el hecho de emplear cierta tecnología implica la dependencia tecnológica de otros servicios tal fue el caso de la implementación del patrón saga con Java Spring y Eventuate.

Además, se considera de gran valor la profundización lograda en las tecnologías utilizadas al punto tal que se tuvo el suficiente conocimiento sobre, por ejemplo, Eventuate para implementar muchas de sus principales características en una tecnología diferente como lo es Javascript.

A su vez, en el afán de implementar funcionalidades similares a las de Eventuate se constató que internamente implementa otro patrón de los relevados en este trabajo como lo es el Transactional OUTBOX por lo cual fue interesante el que este haya sido incluido en el alcance del proyecto. Esto se debe a que se terminó implementando como parte del patrón Saga.

Al mismo tiempo, se destaca el hecho de que se lograron implementar los dos enfoques existentes (orquestación y coreografía) para el patrón Saga lo cual aporta el valor de tener ejemplos de los dos enfoques para los usuarios.

A nivel general, se entiende que se cumplieron los objetivos planteados y se pudo aportar valor a los usuarios mediante una herramienta intuitiva y útil la cual se espera sea de ayuda real en la implementación de la arquitectura de microservicios a través de patrones.

Se cree que el aportar un modelo conceptual en el cual se puedan instanciar patrones de microservicios no es un detalle menor ya que aclara y ayuda a lograr un mayor entendimiento de los conceptos relacionados.

Un aspecto que se tuvo en cuenta desde las etapas más tempranas del proyecto fue el de la extensibilidad de la plataforma. Una vez culminado el proyecto se cree que se logró cumplir con este objetivo, ya que, si bien la arquitectura de solución propuesta es sencilla, cuenta con los requerimientos adecuados para poder extenderse fácilmente, sea agregando información a los patrones existentes o incorporando nuevos patrones.

Sumado a esto, el ofrecer aplicaciones de ejemplo para cada implementación brindada permite tener funcionando en un contexto concreto las estructuras brindadas por la plataforma.

Finalmente, se permite navegar y consultar los diferentes patrones obteniendo información procesada lo cual aporta una fuente concreta de información a los usuarios.

Es válido destacar que el hecho de estar en contacto con un gran referente en el área como lo es Chris Richardson (fundador de Eventuate) y poder aprender directamente desde sus consejos fue muy valioso para el proyecto. Lo mismo sucedió con el contacto mantenido con Mauro Vocale. Se considera de suma importante el hecho de establecer comunicación directa con actores importantes de la industria que permitieron, sin lugar a dudas, enriquecer el proyecto.

6.3 Trabajo a futuro

A continuación se mencionan algunos puntos interesantes que fueron identificados como posibles mejoras.

Múltiples patrones interactuando entre sí: Un aspecto de sumo interés que se podría implementar para extender las funcionalidades de la herramienta es el soporte a que más de un patrón pueda ser implementado en un stack tecnológico específico y colaborar entre sí en la implementación de una estructura única. De esta manera el abanico de soluciones que la plataforma aportaría sería más amplio. Por ejemplo, permitiendo la implementación de *templates* específicos que involucren distintas combinaciones de patrones.

Validar la calidad de las implementaciones agregadas: Si bien la plataforma está preparada para que se pueda extender tanto la cantidad de patrones como las diferentes implementaciones en stack tecnológicos distintos de cada patrón, sería una buena mejora incorporar validaciones de calidad a las nuevas implementaciones de patrones.

Concretamente sería importante contar con algún mecanismo que permita validar el funcionamiento de las estructuras de códigos que se agregue o en su defecto permita cerciorarse de que las nuevas implementaciones (independientemente de la forma en que se manifiesten ya que cada patrón puede diferir en su forma de implementación) cumplen con ciertos estándares de calidad que habría que definir. Además, la validación de las aplicaciones de ejemplo sobre cada implementación también es un detalle no menor.

Agregar implementaciones a patrones existentes: Ya que la gran mayoría de los patrones contemplados solo proveen de información sería importante contar con implementaciones de los mismos.

Estas implementaciones pueden variar dependiendo del patrón, ya que algunos (como en el caso de Saga y Circuit Breaker) se implementan a través de código concreto, mientras que otros (como descomponer en capacidades de negocio) se implementan de una forma más abstracta, por lo cual lo correcto sería proveer lineamientos concretos que ayuden en la implementación del mismo.

Incorporar más patrones: Una mejora obvia es la extensión de la plataforma para que se

contemplen una mayor cantidad de patrones.

Mejorar la usabilidad: A medida que se vayan agregando más patrones e información sería importante contar con mejoras a nivel de interfaz de usuario, ya sea con la estética de la aplicación así como también con el agregado de nuevos y mejores filtros que permitan a los usuarios que la búsqueda y obtención de información valiosa desde la plataforma se produzca de una forma más amigable e intuitiva.

Implementar más de las características identificadas en la sección de análisis: Por un tema de alcance este proyecto se centró en brindar código al usuario. Como se expone en los párrafos anteriores aún resta un gran esfuerzo en dicha área. Sin embargo, las partes del flujo ideal identificado que el proyecto no abarca serían un buen candidato para extender este trabajo. Concretamente, se hace referencia a la implementación de un modelador gráfico donde los usuarios sean capaces de ensamblar la arquitectura de sus aplicaciones mediante un lenguaje específico para patrones de microservicios. Relacionado con esto, definir un DSL mediante el cual se puedan representar los diferentes patrones y modelar las restricciones entre sí.

Anexos

A Presentación de los patrones por categoría

En este apéndice se presenta el trabajo de relevamiento realizado.

La organización del mismo, consta de la presentación de la instanciación de todos los patrones agrupados por categorías.

A.1 Descomposición:

Proceso mediante el cual se descompone una aplicación en múltiples microservicios de forma tal que, mediante la cooperación y comunicación entre los mismos logren implementar todas las funcionalidades del sistema.

Se debe lograr que una vez que el sistema es descompuesto en servicios, el funcionamiento de los mismos en conjunción logre brindar una imagen del sistema como un todo, siendo transparente para el usuario el hecho de que realmente se están implementando múltiples sistemas autónomos.

En esta categoría se identificaron 2 patrones:

- Descomponer en base a capacidades de negocio: Se plantea que la descomposición en microservicios debe estar alineada a las capacidades de negocio.

  Las capacidades de negocio deben capturar los puntos importantes del negocio y en general son estables en comparación con la forma en que una organización lleva a cabo sus negocios, que es dinámica a lo largo del tiempo.

  Eso es especialmente cierto hoy en día, con el uso cada vez mayor de la tecnología para automatizar muchos procesos comerciales. Fowler en [53] define capacidad de negocio como lo que hace el negocio en un dominio particular para poder cumplir con sus objetivos y responsabilidades.

Finalmente, en la Tabla A.1 se puede apreciar la instanciación del patrón en el modelo previamente definido.

Nombre	Descomponer en base a capacidades de negocio
Categoría	Descomposición
Autor	Richardson, Fowler
Relaciones	Alternativo - Descomponer en subdominios
Componentes	Microservicios, Funcionalidades del sistema
Tecnologías	
Variables	Capacidades del negocio

Table A.1: Descomposición por capacidad de negocio

- Descomponer en base a subdominios: La idea es identificar los subdominios en los que se compone la organización y atacar los problemas dentro de un dominio particular. Cada subdominio particular hace referencia a una característica del negocio.

  Richardson en [1] clasifica a los subdominios en dos:

  – Core: Diferenciador claro del negocio y la parte más valiosa de la aplicación.

  – Supporting: Relacionado con qué hace el negocio pero no es un diferenciador.

  La idea es definir varios modelos que muestren las distintas visiones de cada concepto del negocio, ya que si se utiliza el enfoque tradicional en donde se define un solo modelo para mapear la realidad se puede estar sobregeneralizando algunos aspectos.

  Finalmente, en la Tabla A.2 se puede apreciar la instanciación del patrón en el modelo previamente definido.

Nombre	Descomponer en base a subdominios
Categoría	Descomposición
Autor	Richardson, Fowler, Comunidad
Relaciones	Alternativo - Descomponer en capacidades de negocio
Componentes	Microservicios, Subdominios del sistema
Tecnologías	
Variables	Subdominios, Aspectos del ecosistema

Table A.2: Descomposición por subdominio

A.2 Comunicación:

Si bien la arquitectura de microservicios propone implementar los sistemas mediante múltiples microservicios independientes, es de suma importancia la comunicación entre los mis-

mos ya que, por lo general se necesita de su cooperación para satisfacer los requerimientos del sistema.

Englobados en esta categoría se encuentran:

- Invocación a procedimientos remotos: La comunicación entre los microservicios se logra mediante invocación a procedimientos remotos. Es decir, el microservicio cliente envía una solicitud a un servicio servidor donde se procesa la solicitud y se retorna una respuesta.

  Existen dos variantes:

  - Bloqueante: El microservicio que invoca al procedimiento remoto se bloquea en espera de la respuesta.
  - No Bloqueante: El microservicio invoca al procedimiento remoto sin la necesidad de bloquearse en espera de la respuesta.

  En la Tabla A.3 se presenta la instanciación.

Nombre	Invocación a procedimientos remotos
Categoría	Comunicación
Autor	Richardson, IBM, Microsoft
Relaciones	Alternativo - Mensajería Sucesor: Patrones de Descubrimiento Sucesor opcionales: Circuit Breaker (Mejorar la tolerancia a fallas), Configuración exteriorizada
Componentes	RPI Server, RPI Proxy, Proceso remoto,
Tecnologías	REST(OpenAPI, Swagger, GraphQL, NetflixFalcor), GRPC(Apache Thrift, Nubes que lo brindan(ej: Azure))
Variables	Tipo(bloqueante/no bloqueante), Tecnología a utilizar

Table A.3: Invocación a procedimientos remotos

- Mensajería: La comunicación se logra a través del intercambio de mensajes de manera asíncrona. Una aplicación basada en mensajería, generalmente, utiliza los denominados canales de mensajería (brokers) que son componentes externos con la función de actuar como intermediario entre los servicios que desean comunicarse.

  Estos canales pueden ser:

  - Punto-a-punto: Se entrega el mensaje solo a uno de los receptores que lee desde ese canal.
  - Publish/subscribe: Se entrega el mensaje a todos los consumidores que están suscritos al canal.

Como variante se destaca que podría darse el caso de que la mensajería sea asíncrona y sin intermediarios (denominada arquitectura brokerless) [54].

Sin importar si se use intermediarios, Richardson [1] define tres tipos de mensajes:

- Comando: un mensaje donde se especifica la operación a invocar y sus parámetros.
- Documento: un mensaje genérico que contiene solo datos. El receptor decide cómo interpretarlo. La respuesta a un comando es un ejemplo de mensaje de documento.
- Evento: un mensaje que indica que ha ocurrido algo notable en el remitente. Un evento es a menudo un evento de dominio, que representa un cambio de estado de un objeto de dominio.

En la Tabla A.4 se encuentra la instanciación.

Nombre	Mensajería
Categoría	Comunicación
Autor	Richardson, Fowler, Newman, Netflix, IBM, Microsoft
Relaciones	Alternativo - Invocación a procedimientos remotos Sucesor: Saga y CQRS (Usan mensajería), Si es brokerless: Patrones de descubrimiento Sucesor opcionales: Transactional Outbox (Permite enviar mensajes de manera transaccional), Configuración exteriorizada
Componentes	Canal, Mensaje, Brokers
Tecnologías	Broker-based(ActiveMQ, RabitMQ, Apache Kafka, AWS Kinesis, AWS SQS, IronMQ, Webshare MQ, Redis), Brokerless(ZeroMQ)
Variables	Cantidad de participantes, tipos de canales, Uso de brockers (si/no), Tecnología a utilizar

Table A.4: Mensajería

A.3 Gestión de datos:

Forma en que se diagrama el modelo de gestión de datos en una arquitectura de microservicios. Si bien lo más recomendable es aplicar una base de datos por servicio, existen variantes que proponen enfoques distintos.

- Una base de datos compartida: En este enfoque se plantea la utilización de una única base de datos compartida por todos los servicios que componen a la aplicación. Lo propuesto por este patrón atenta contra muchas de las ventajas que aporta el uso de la

arquitectura de microservicios, algunas de ellas son:

- simplificar las pruebas
- incrementar la velocidad de los desarrolladores

Esto se debe al cuello de botella que significa que todos los servicio persistan datos en una única base. [55]

En la Tabla A.5 se encuentra la instanciación.

Nombre	Una base de datos compartida
Categoría	Gestión de datos
Autor	Richardson
Relaciones	Alternativo - Una base de datos por servicio
Componentes	Base de datos
Tecnologías	Manejadores existentes SQL y noSQL
Variables	SQL/no SQL, Tecnología a utilizar

Table A.5: Una base de datos compartida

- Una base de datos por servicio: Se propone la utilización de una base de datos por servicio. Cada base de datos es independiente del resto incluso en lo referente al modelo de datos específico.

  Es el enfoque que más se adecua a las características inherentes a este tipo de arquitectura, motivo por el cual es el patrón más aplicado en este aspecto. Sin embargo, aporta complejidad en el manejo de la consistencia.

  En la Tabla A.6 se encuentra la instanciación.

Nombre	Una base de datos por servicio
Categoría	Gestión de datos
Autor	Richardson, IBM, Fowler, Newman
Relaciones	Alternativo - Una base de datos compartida Sucesor: Saga (Manejo de la consistencia), Sucesor Opcional: API Composition, CQRS (Implementación de las consultas)
Componentes	Bases de datos
Tecnologías	Manejadores existentes SQL y noSQL
Variables	SQL/no SQL (Cada base de datos se configura independientemente), Tecnología a utilizar

Table A.6: Una base de datos por servicio

A.3.1 Consistencia:

Mantener la consistencia en sistemas donde se tienen múltiples bases de datos (por ejemplo después de aplicar el patrón antes presentado).

El único patrón que surgió producto de la investigación en esta área es el patrón Saga el cual se profundiza en la sección 3.3.

A.3.2 Consulta de datos:

Gestionar las consultas que requieren datos propiedad de múltiples servicios.

Por lo general muchas de las consultas necesitan unir datos que son propiedad de múltiples servicios para satisfacer los requerimientos. A su vez, el realizar consultas distribuidas no es una opción ya que, solo se puede acceder a los datos de un servicio a través de su interfaz (API).

Para realizar esta tarea se destacan dos enfoques:

- Existe una entidad que se encarga de ensamblar los datos necesarios para satisfacer la consulta extrayendo los mismos de los microservicios que sea necesario.
- Mantener réplicas de los datos (propiedad de otros servicios) extra que se requieren para satisfacer la consulta en el servicio consultado.

Para cada una de estas variantes existe un patrón.

- API Composition: Es el enfoque más simple y debe utilizarse siempre que sea posible. Su funcionamiento se basa en dos conceptos principales, el "API Composer" y el "provider service". Se implementa la lógica de las consultas sobre la base de datos que posee el dato buscado y luego se combinan los resultados para satisfacer la consulta que requiere datos de varios microservicios.

  El "API Composer" es el encargado de generar una respuesta a la solicitud original agregando las diferentes respuestas de los servicios involucrados, las cuales obtiene de cada uno de ellos. La interacción entre el "composer" y los "providers" se logra a través de la exposición de API's por lo cual es simple de implementar y a su vez las responsabilidades están bien definidas ya que únicamente el "composer" es el encargado de presentar una respuesta a la solicitud del cliente.

  En la Tabla A.7 se presenta la instanciación.

- Command Query Responsibility Segregation (CQRS): Este patrón es definido por diversas fuentes [57], [58], [59] y [60].

  Según Martin Fowler, se tiene un modelo de datos para actualizar la información y otro para consultarla. Se propone que los servicios mantengan una o más réplicas de datos que son propiedad de otros servicios pero que son de utilidad para consultas frecuentes.

Nombre	API Composition
Categoría	Gestión de datos/Consulta de datos
Autor	Richardson
Relaciones	Alternativo - CQRS Predecesor: Una base de datos por servicio
Componentes	API Composer, Service Provider
Tecnologías	OpenAPI, API Transaformer, OpenAPItoUML (Visualizar las API de los diferentes servicios), Apache Olingo
Variables	API de los distintos servicios, Tecnologías a utilizar

Table A.7: API Composition

No obstante, se debe realizar un diseño cuidadoso de las replicar a implementar de forma tal de lograr la mayor eficiencia posible en las consultas replicando la menor cantidad de datos.

Se pueden destacar varios aspectos los cuales impulsan la existencia de este enfoque:

1. El uso del patrón de API Composition es muy costoso en aquellos casos donde se requiere recuperar datos dispersos en múltiples servicios.

2. El servicio que posee los datos almacena los datos en un formulario o en una base de datos que no admite de manera eficiente la consulta requerida.

3. El servicio que posee los datos no es el servicio que debe implementar la operación de consulta.

En la Tabla A.8 se presenta la instanciación.

Nombre	CQRS
Categoría	Gestión de datos/Consulta de datos
Autor	Fowler, Romain Pierlot, Microsoft, IBM
Relaciones	Alternativo - API Composition Sucesor Opcional: Eventos de dominio Predecesor: Una base de datos por servicio
Componentes	Replica, Manejador de eventos
Tecnologías	MongoDB, DynamoDB, Redis, Elastic Search, Neo4j
Variables	Réplicas a realizar, Servicios replicados, Servicios en donde replicar, Tecnologías a utilizar

Table A.8: CQRS

A.4 Tolerancia a fallas:

En un sistema distribuido, cada vez que un servicio realiza una solicitud sincrónica a otro servicio, existe el riesgo de que ocurra una falla parcial.

Debido a que el cliente y el servicio son procesos separados, es posible que en un momento puntual un servicio no pueda responder de manera oportuna a la solicitud del cliente, por ejemplo porque el servicio podría estar inactivo debido a una falla, encontrarse en estado de mantenimiento o estar sobrecargado y responder extremadamente lento a las solicitudes.

Considerando el hecho de que en este contexto el cliente está bloqueado a la espera de una respuesta, el peligro es que la falla que está afectando al cliente actual podría comenzar a impactar en otros clientes (aquellos microservicios que esperan una respuesta del que está bloqueado).

Circuit Breaker es el único patrón identificado en esta área y sera profundizado en la sección 3.3.

A.5 Despliegue:

Como todas las aplicaciones, las desarrolladas con la arquitectura de microservicios requiere de infraestructura en la cual ser desplegada.

Dado que pueden haber decenas o cientos de servicios desarrollados en distintos lenguajes y frameworks, existen muchas partes móviles que necesitan ser gestionadas, por lo cual es un tema muy importante el cómo se despliega la aplicación.

Se identifican varios patrones que proveen alternativas.

- Service mesh: RedHat define al Service Mesh en [62] como la manera de controlar cómo diferentes partes de una aplicación comparten datos con otras. Es una capa de infraestructura dedicada que está construida por encima de la aplicación. Esta capa asegura que la comunicación entre los servicios es rápida, segura y confiable.

  Un service mesh enruta todo el tráfico de red dentro y fuera de los servicios a través de una capa que implementa por defecto varios de los aspectos tratados como desafíos a lo largo del documento, como por ejemplo circuit breakers, distributed tracing, descubrimiento de servicios y balanceo de cargas [63].

  Tradicionalmente el proceso de despliegue de una aplicación era totalmente responsabilidad del equipo de operaciones y se realizaba de forma manual. Sin embargo, en la actualidad existe una vasta gama de herramientas que permiten la automatización de este tipo de procesos. Por dicho motivo se plantea la utilización de este tipo de tecnologías para implementar una capa que permita abstraer varios de los desafíos a la hora de desplegar aplicaciones que se componen de múltiples servicios [64].

  En la Tabla A.9 se presenta la instanciación.

Nombre	Service Mesh
Categoría	Despliegue
Autor	RedHat, Kasun Indrasiri, Nginx, Richardson
Relaciones	Alternativo - Servicios en máquinas virtuales, Servicios en contenedores, Plataforma para el despliegue de servicios, Despliegue sin servidores
Componentes	
Tecnologías	Istio, Linkred, Conduit, Jaeger, Kiali, Red Hat Service Mesh
Variables	Tecnología a utilizar

Table A.9: Service Mesh

- Servicios en máquinas virtuales: Desplegar cada servicio como una imagen en una máquina virtual (MV).

  Beneficios:

  - Se encapsula el stack tecnológico utilizado, ya que la imagen contiene las dependencias del servicio por lo cual disminuye la probabilidad de introducir errores en la configuración del entorno de ejecución.
  - Aporta abstracción ya que una vez que se crea y configura la imagen es una caja negra que puede ser desplegada donde sea.
  - Instancias de los servicios aisladas unas de otras, propiedad adquirida de las MV.

  Desventajas:

  - Desaprovechamiento de recursos, cada MV tiene la sobrecarga que aporta el sistema operativo. La capacidad mínima de las MV provistas en la nube puede ser significativamente mayor a lo requerido.
  - Lentitud en el despliegue: Las MV en general demoran unos minutos en iniciar.
  - Sobrecarga relacionada con la administración del sistema: Se utiliza el sistema propio de cada MV lo que requiere administración. Esta desventaja se acrecienta sobre todo en contraposición con el enfoque sin servidor presentado a continuación.

  En la Tabla A.10 se presenta la instanciación.

- Servicios en contenedores: Propone desplegar los servicios como contenedores, donde cada instancia de un servicio se mapea con un contenedor, obteniendo así todas las ventajas que tienen los contenedores por sobre las MV [65].

Nombre	Servicios en máquinas virtuales
Categoría	Despliegue
Autor	Richardson
Relaciones	Alternativo - Service Mesh, Servicios en contenedores, Plataforma para el despliegue de servicios, Despliegue sin servidores
Componentes	VM Builder, VM Image
Tecnologías	Animator de Netflix, Packer, Elastic Beacnstalk
Variables	Tecnología a utilizar

Table A.10: Servicios en máquinas virtuales

En la Tabla A.11 se presenta la instanciación.

Nombre	Servicios en contenedores
Categoría	Despliegue
Autor	Comunidad
Relaciones	Alternativo - Service Mesh, Servicios en máquinas virtuales, Plataforma para el despliegue de servicios, Despliegue sin servidores
Componentes	Contenedor
Tecnologías	Docker, Solaris Zones, Docker Cloud Registry, AWS EC2 Container Registry, Docker Compose, Kubernetes, Google Container Engine, AWS ECS, OpenShift, Marathon
Variables	Tecnología a utilizar

Table A.11: Servicios en contenedores

- Plataforma para el despliegue de servicios: En este enfoque se propone desplegar los servicios sobre plataformas de orquestación de contenedores. Esta solución está intrínsecamente relacionada con la anteriormente presentada, sin embargo, la diferencia radica en el nivel de abstracción que se provee.

  En este enfoque las plataformas implementan parte de las responsabilidades que tendrían los desarrolladores si no las utilizaran.

  Como principales ventajas se destacan:

  - Agilidad en el despliegue
  - Facilidad para escalar
  - Mayor tolerancia a los fallos

En la Tabla A.12 se presenta la instanciación.

Nombre	Plataformas para el despliegue de servicios
Categoría	Despliegue
Autor	Richardson, RedHat, IBM
Relaciones	Alternativo - Service Mesh, Servicios en máquinas virtuales, Servicios en contenedores, Despliegue sin servidores Predecesor: Provee Registro de Servicios y Descubrimiento de servicios
Componentes	Plataforma
Tecnologías	Kubernetes, OpenShift
Variables	Tecnología a utilizar

Table A.12: Plataforma para el despliegue de servicios

- Despliegue sin servidores: Utilizar infraestructura de implementación que oculte cualquier concepto de servidores. Este enfoque está basado en el concepto de funciones como servicios [66]. Es un modelo de ejecución en el que el proveedor de nube es responsable de ejecutar un fragmento de código mediante la asignación dinámica de los recursos y, cobrando solo por la cantidad de recursos utilizados para ejecutar el código.

  Por otra parte, la infraestructura de implementación es una utilidad operada por el proveedor externo que por lo general utiliza contenedores o máquinas virtuales para aislar los servicios, pero esta configuración es transparente para la organización.

  Se aumenta agilidad e innovación y se elimina el tiempo en el aprovisionamiento de servidores y mantenimiento de sistemas [67].

  En la tabla A.13 se presenta la instanciación.

Nombre	Despliegue sin servidores
Categoría	Despliegue
Autor	Azure, AWS, IBM, GCP, Serverless Stack
Relaciones	Alternativo - Service Mesh, Servicios en máquinas virtuales, Servicios en contenedores, Plataforma para el despliegue de servicios
Componentes	Funciones ejecutables
Tecnologías	IBM Cloud Functions, AWS Lambda, Google Cloud Functions, Azure Functions
Variables	Funciones a ejecutar, Tecnología a utilizar

Table A.13: Despliegue sin servidores

A.6 Descubrimiento de servicios:

Conocer la ubicación en la red (dirección IP y puerto) de las distintas instancia de servicio desplegadas en la nube.

Dado que en la actualidad las aplicaciones son desplegadas utilizando proveedores de nube pública, que muchas veces ofrecen la gestión de ciertos aspectos como el escalado en función de la demanda, tolerancia a fallas, etc, se ha vuelto todo un desafió el localizar a los servicios a la hora de querer invocarlos. En consecuencia, las aplicaciones deben utilizar un descubrimiento de servicio.

El descubrimiento de servicios es conceptualmente bastante simple: su componente clave es un registro de servicios (service registry), que es una base de datos de las ubicaciones de red de las instancias de servicio de una aplicación. El mecanismo de descubrimiento de servicios actualiza el registro de servicios cuando las instancias de servicio comienzan y se detienen. Cuando un cliente invoca un servicio, el mecanismo de descubrimiento del servicio consulta el registro del servicio para obtener una lista de instancias de servicio disponibles y enrruta la solicitud a una de ellas.

Existen dos formas de implementar el descubrimiento de servicios:

- Los servicios y sus clientes interactúan directamente con el registro de servicios.
- La infraestructura de despliegue maneja el descubrimiento de servicios.

Englobados en esta categoría se encuentran los siguientes patrones:

- Descubrimiento de servicios del lado del cliente: Cuando un cliente desea invocar a un servicio, consulta el registro del servicio para obtener una lista de las instancias del mismo.

  Como medida para mejorar el rendimiento, el cliente puede almacenar en caché las instancias del servicio, para luego utilizar un algoritmo de balanceo de carga permitiéndole seleccionar una instancia del servicio específica [68].

  En la Tabla A.14 se presenta la instanciación.

- Descubrimiento de servicios del lado del servidor: En lugar de que un cliente consulte el registro del servicio, realiza una solicitud a un nombre DNS, que se resuelve en un enrutador de solicitud que consulta el registro del servicio.

  En la Tabla A.15 se presenta la instanciación.

- Registro personal de servicios: El mismo servicio es el encargado de invocar la API del registro para registrar su ubicación de red.

  En la Tabla A.16 se presenta la instanciación.

- Registro con herramientas de terceros: En lugar de que un servicio se registre a si

Nombre	Descubrimiento de servicios del lado del cliente
Categoría	Descubrimiento de servicios
Autor	Richardson
Relaciones	Alternativo - Descubrimiento de servicios del lado del servidor Predecesor: Registro Personal de servicios
Componentes	Balanceador de carga, Service registry
Tecnologías	Netflix Ribbon, AWS Elastic Load Balancer (ELB), Kubernetes, Marathon, Spring Framework, Eureka, Consul, Apache Zookeeper, Etcd, Apache Turbine
Variables	Tecnología a utilizar

Table A.14: Descubrimiento de servicios del lado del cliente

Nombre	Descubrimiento de servicios del lado del servidor
Categoría	Descubrimiento de servicios
Autor	Richardson
Relaciones	Alternativo - Descubrimiento de servicios del lado del cliente Predecesor: Registro Personal de servicios Sucesor Alternativo: Circuit Breaker
Componentes	Balanceador de carga, Service registry
Tecnologías	Netflix Ribbon, AWS Elastic Load Balancer (ELB), Kubernetes, Marathon, Spring Framework, Eureka, Consul, Apache Zookeeper, Etcd, Apache Turbine
Variables	Tecnología a utilizar

Table A.15: Descubrimiento de servicios del lado del servidor

mismo en el registro de servicios, un tercero llamado registrador, que generalmente forma parte de la plataforma de implementación, se encarga del registro.

En la Tabla A.17 se presenta la instanciación.

A.7 Observación

Observación de los diferentes servicios en pos de detectar anomalías en el funcionamiento y/o estado de los servicios.

Por otra parte cuando ocurre un falla en el sistema es importante tener un registro de la actividad de los servicios para poder identificar el origen de la falla.

Para este propósito se identifican los patrones presentados a continuación:

- Health Check API: Plantea exponer un endpoint cuyo propósito sea notificar el estado actual del servicio.

Nombre	Registro personal de servicios
Categoría	Descubrimiento de servicios
Autor	Richardson
Relaciones	Alternativo - Registro con herramientas de terceros Predecesor: Descubrimiento del lado del cliente/servicio
Componentes	Service registry
Tecnologías	Eureka, Consul, Apache Zookeeper, Etcd, Apache Turbine
Variables	Tecnología a utilizar

Table A.16: Registro Personal de servicios

Nombre	Registro con herramientas de terceros
Categoría	Descubrimiento de servicios
Autor	Richardson
Relaciones	Alternativo - Registro personal de servicios Predecesor: Descubrimiento del lado del cliente/servicio
Componentes	Registrador
Tecnologías	Netflix Prana (Usa Eureka), Zoekeeper, Consul
Variables	Tecnología a utilizar

Table A.17: Registro con herramientas de terceros

En la Tabla A.18 se presenta la instanciación.

Nombre	Health Check API
Categoría	Observación
Autor	Richardson, Fowler, Newman
Relaciones	Predecesor - Registro de servicios (Es quien lo invoca)
Componentes	Endpoint
Tecnologías	Spring Boot Actuator, HealthChecks
Variables	Tecnología a utilizar

Table A.18: Health Check API

- Registro de logs: El objetivo que plantea este patrón es llevar un registro de la actividad de los servicios en un servidor que sea capaz de proveer alertas y capacidad de búsqueda. Para que la idea planteada surta efecto se deben implementar los servicios de forma tal que se almacenen aquellos eventos que verdaderamente sean útiles.

  En la Tabla A.19 se presenta la instanciación.

- Seguimiento de excepciones: La idea detrás de este patrón es muy similar a la presen-

Nombre	Registro de logs
Categoría	Observación
Autor	Comunidad
Relaciones	Sucesor Opcional: Seguimiento de excepciones, Seguimiento distribuido
Componentes	Logs, Servidor de logs
Tecnologías	DataDog, Loggly, AWS Cloud Watch, ELK (ElasticSearch, Logstash, Kibana), Fluentd, ApacheFlume, Logs, IBM Cloud Log Analysis with LogDNA, Zipkin, Papertrail
Variables	Indices para la busqueda eficiente de logs, Tecnología a utilizar

Table A.19: Registro de logs

tada en Registro de logs con la diferencia que en este caso se plantea aplicarlo sobre las excepciones.

En la Tabla A.20 se presenta la instanciación.

Nombre	Seguimiento de excepciones
Categoría	Observación
Autor	Richardson, Fowler
Relaciones	Sucesor Opcional: Registro de logs, Seguimiento distribuido
Componentes	excepciones, Servidor de excepciones
Tecnologías	DataDog, Honeybadger, Sentry.io
Variables	Índices para la búsqueda eficiente de excepciones, Tecnología a utilizar

Table A.20: Seguimiento de excepciones

- Seguimiento distribuido: Propone asignar un id único a cada solicitud de forma tal que se pueda realizar un seguimiento de las mismas durante todo su ciclo de vida. El objetivo principal es tener un panorama claro del período en el cual la solicitud estuvo activa para, por ejemplo, cuantificar la latencia que aporta cada servicio al sistema.

  En la Tabla A.21 se presenta la instanciación.

- Métricas de la aplicación: Los servicios son los responsables de reportar sus propias métricas a un servidor de métricas central, logrando así tener un análisis primario centralizado de cada servicio.

  En la Tabla A.22 se presenta la instanciación.

Nombre	Seguimiento distribuido
Categoría	Observación
Autor	Richardson, Fowler
Relaciones	Predecesor: Registro de logs, Seguimiento de excepciones (se asigna un id a cada log/excepción para su rastreo mas eficiente)
Componentes	Librería de instumentacion, Servidor de rastreo distribuido
Tecnologías	Spring Cloud Sleuth, Zipkin-AWS-X-ray
Variables	Tecnología a utilizar

Table A.21: Seguimiento distribuido

Nombre	Métricas de la aplicación
Categoría	Observación
Autor	Richardson
Relaciones	
Componentes	Servidor de Métricas, Métrica
Tecnologías	Micrometers, AWS CloudWatch Metrics, Prometheus, Grafana, Java Metrics Library
Variables	Métricas importantes, Tecnología a utilizar

Table A.22: Métricas de la aplicación

- Audit logging: Un enfoque similar a algunos presentados anteriormente propone registrar las acciones de los usuarios en un repositorio central.

  En la Tabla A.23 se presenta la instanciación.

A.8 Seguridad

El tema de la seguridad de las aplicaciones no es un tema puntual de los microservicios si no que concierne a todo tipo de sistemas.

Sin embargo, cuando se implementa seguridad en sistemas con microservicios no se puede mantener el contexto ni las sesiones en memoria ya que los servicios no la comparten.

Es por este motivo que el enfoque utilizado para la autenticación en este tipo de aplicaciones es delegar al API gateway (presentado en la sección A.11) la gestión de la seguridad.

El patrón identificado se presenta a continuación:

- Token de acceso: Propone la utilización de un token mediante el cual se realizan las invocaciones. Cuando en el punto de autenticación se recibe una invocación con un

Nombre	Audit logging
Categoría	Observación
Autor	Richardson
Relaciones	Predecesor: Event sourcing (Se implementa a través de este)
Componentes	Eventos importantes, Repositorio de eventos
Tecnologías	Google analytics, Segment, Matomo
Variables	Eventos a persistir, Tecnología a utilizar

Table A.23: Audit logging

token autorizado se reenvía dicho token a los servicios destino, logrando así un correcto manejo de la autenticación de los usuarios.

Sin embargo, en lo referente a la autorización no es recomendable que se maneje en un único punto ya que para ello, este componente debería conocer al detalle los objetos de cada uno de los servicios lo que implicaría un alto acoplamiento entre los mismos.

Por este motivo es que se propone que la autorización se implemente en cada servicio.

Existen dos tipos de tokens [69]:

- Token opaco: Contiene información no accesible ya que para validarlos es necesario consultar al servicio que lo emitió. Tienen la desventaja de reducir la performance y disponibilidad e incrementar la latencia, ya que utiliza un mecanismo sincrónico (RPC) para la validación del token.
- Token transparente: Contiene información en formato JSON agregando una clave secreta que solo conocerá el emisor y el receptor. Como desventaja se destaca que no se pueden revocar, para utilizarlos se verifica la firma y la fecha de expiración, por lo cual si cae en manos de un tercero malicioso la única forma de invalidarlo es que expire su tiempo de vida.

  Como contra medida se busca configurar tiempos de expiración cortos, lo cual genera que para mantener una sesión de larga duración se deberá renovar el token varias veces.

En la Tabla A.24 se presenta la instanciación.

A.9 Migración:

Migrar desde sistemas monolíticos hacia microservicios.

Patrones:

- Strangler application: La definición de este patrón se puede encontrar en diversas

Nombre	Token de acceso
Categoría	Seguridad
Autor	Comunidad
Relaciones	Predecesor: API Gateway (Es usado por este patrón)
Componentes	Token
Tecnologías	Spring Security, Apache Shiro, Passport, JSON Web Token
Variables	Políticas para revocarlo/otorgarlo, Tiempo de expiración, Tecnología a utilizar

Table A.24: Token de acceso

fuentes tales como Chris Richardson [1], Microsoft [70], Martin Fowler [71], IBM [72] y Samir Behara [73].

Se define como un proceso que consta de ir creando un nuevo sistema gradualmente alrededor del viejo (monolítico) que vaya creciendo lentamente hasta que el nuevo sistema “estrangule” al viejo.

Por ejemplo, Richardson plantea un enfoque incremental donde en el proceso se genera una aplicación denominada "strangler" (estranguladora) que consiste en microservicios que implementan nuevas funcionalidades que extienden a la aplicación original monolítica y la extracción de servicios desde el monolito se va implementando a medida que los requerimientos van mutando.

En la Tabla A.25 se presenta la instanciación.

Nombre	Strangler aplication
Categoría	Migración
Autor	Richardson, Microsoft, Fowler, IBM, Samir Behara
Relaciones	
Componentes	Funcionalidades monolíticas, Funcionalidades en microservicios
Tecnologías	
Variables	Nuevos requerimientos (A implementar con microservicios)

Table A.25: Strangler application

- Anti Corruption Layer: Se presenta un enfoque en el cual se implementa una capa de software que oficia de intermediario entre el nuevo sistema (microservicios) y el sistema legado (monolítico).

  Es necesario ya que en el proceso de migración (cuando coexisten ambos sistemas) es

posible que se generen inconsistencias relacionadas a la representación de los conceptos. Por lo cual es necesaria una capa de intermediación [74].

Un desafío que surge al aplicar esta idea es que implementar la generación y recepción de eventos en el nuevo servicio es sencillo (ya que se construye desde cero), sin embargo, si esta capacidad no está implementada en el legado, se dificulta hacerlo [75].

En la Tabla A.26 se presenta la instanciación.

Nombre	Anti Corruption Layer
Categoría	Migración
Autor	Richardson, Microsoft, Ahmed Shirin
Relaciones	
Componentes	Capa intermediaria
Tecnologías	
Variables	Modelo de datos del nuevo sistema (sistema en microservicios)

Table A.26: Anti Corruption Layer

A.10 Testing

Gestionar las pruebas de la aplicación.

La arquitectura de microservicios hace que los servicios individuales sean más fáciles de testear porque son mucho más pequeños que en una aplicación monolítica. Sin embargo, el gran desafío radica en comprobar que los diferentes servicios funcionan juntos de manera adecuada, y al mismo tiempo evitar el uso de pruebas complejas, lentas y frágiles de extremo a extremo que prueben múltiples servicios juntos.

Con este propósito se identifican 3 patrones:

- Consumer driven contract: Se propone que los microservicios interactúen en pos de realizar verificaciones cruzadas. Lo particular de este enfoque es que quien desarrolla el test es el equipo que gestiona el servicio consumidor, dichos tests comprueban que la definición de las API del proveedor sean efectivamente las que se esperan.

  Una vez que los tests son implementados se los publica en un repositorio propiedad del servicio consumidor (el cual se está testeando), una vez allí el equipo encargado del proveedor los podrá ejecutar y en su defecto corregir sus API para satisfacer lo que los consumidores esperan.

  En la Tabla A.27 se presenta la instanciación.

- Consumer-side contract: El objetivo de este patrón es verificar que el consumidor de un servicio (el cliente) pueda comunicarse con el proveedor. Para ello se propone utilizar

Nombre	Consumer driven contract
Categoría	Testing
Autor	Richardson
Relaciones	Complementarios: Consumer-side contract, Service component
Componentes	Contrato
Tecnologías	Spring Cloud Contract
Variables	

Table A.27: Consumer driven contract

los mismos contratos que en "Consumer driven contract test" para testear también al consumidor. Concretamente esto se logra generando un servicio stub que testea la API de consumidor.

En la Tabla A.28 se presenta la instanciación.

Nombre	Consumer-side contract
Categoría	Testing
Autor	Richardson
Relaciones	Complementarios: Consumer driven contract, Service component
Componentes	Contrato, Stubs
Tecnologías	Spring Cloud Contract
Variables	

Table A.28: Consumer-side contract

- Service component: Una vez que cada servicio se probó unitariamente es necesario comprobar que el comportamiento de los servicios es el adecuado en el ecosistema en donde coexisten. Por este motivo, surge la necesidad de probar a los servicios considerándolos "cajas negras" y verificar la interacción de los mismos con los demás servicios a través de sus APIs.

  En la Tabla A.29 se presenta la instanciación.

A.11 Comunicación con el exterior:

Definir y gestionar la comunicación del sistema desarrollado con microservicios con aplicaciones externas.

- API Gateway: Dado que este patrón es utilizado con mucha frecuencia se puede encontrar información en importantes fuentes, como: Microsoft [76], Nginx [77], Amazon

Nombre	Service component
Categoría	Testing
Autor	Richardson
Relaciones	Complementarios: Consumer driven contract, Consumer-side contract
Componentes	Contrato
Tecnologías	Spring Cloud Contract
Variables	

Table A.29: Service component

[78] y Chris Richardson [2].

El API Gateway es el punto de entrada al sistema que permite interconexión más eficiente entre aplicaciones externas y el sistema en microservicios.

Sin embargo, al aplicarlo se deben gestionar algunos aspectos, tales como:

- Diferentes clientes requieren diferentes datos.
- El rendimiento de la red es diferente para diferentes tipos de clientes.
- El número de instancias de servicio y sus ubicaciones (host + puerto) cambian dinámicamente.
- La partición en los servicios puede cambiar con el tiempo y debe ocultarse a los clientes.
- Los servicios pueden usar un conjunto diverso de protocolos, algunos de los cuales pueden no ser compatibles con la web.

El API Gateway cumple la función de un proxy inverso, enrutando las solicitudes de los clientes a los servicios. Se sitúa entre los clientes (web/móvil, aplicaciones de terceros) y los microservicios que componen a la aplicación.

En la Tabla A.30 se presenta la instanciación.

- Backend for frontends: Fue propuesto por Phil Calçado [21], y tiene una fuerte relación con API Gateway ya que lo que se propone es una extensión de este. Concretamente, se plantea tener un API Gateway por cada tipo diferente de cliente que se maneje en la aplicación, donde cada equipo encargado del manejo de esos clientes tiene la potestad de gestión e implementación de esa API.

  Además de definir claramente las responsabilidades, el patrón Backend For Frontend tiene otros beneficios:

  - Los API Gateway están aislados entre sí, lo que mejora la confiabilidad.

Nombre	API Gateway
Categoría	Comunicación con el exterior
Autor	Richardson, Amazon, Nginx, Microsoft
Relaciones	Alternartivo: Backends for Frontends Sucesor: Token de acceso (Se implementa en el API Gateway)
Componentes	Api Gateway
Tecnologías	AWS API Gateway, AWS Load Balancer, Kong, Traefik, Spring Cloud Gateway, Guava, Spring Cloud Config, Netflix Archaius
Variables	Funcionalidades a brindar (Para exponer los endpoints adecuados)

Table A.30: API Gateway

- Una API que experimenta un comportamiento erróneo no puede afectar fácilmente a otras API.
- Mejora la observabilidad, ya que las diferentes API son procesos independientes.
- Cada API escala independientemente.
- Reduce el tiempo de inicio que cada API Gateway insume ya que la implementación de los mismos es de un tamaño considerablemente menor a que si se tuviera todo englobado en un solo API Gateway.

En la Tabla A.31 se presenta la instanciación.

Nombre	Backend for frontends
Categoría	Comunicación con el exterior
Autor	Richardson, Phil Calçado
Relaciones	Alternartivo: API Gateway Sucesor: Token de acceso (Se implementa en cada uno de los API Gateway)
Componentes	Multiples API Gateway
Tecnologías	AWS API Gateway, AWS Load Balancer, Kong, Traefik, Spring Cloud Gateway, Guava, Spring Cloud Config, Netflix Archaius
Variables	Funcionalidades a brindar (Para exponer los endpoints adecuados), Tipos de clientes (Define cuantos API Gateway se deben implementar)

Table A.31: Backend for frontends

A.12 Mensajería Transaccional:

Gestionar el envió de mensajes de manera transaccional.

Los servicios a menudo necesitan publicar mensajes como parte de una transacción que actualiza la base de datos.

Por ejemplo, servicios que publican eventos de dominio cada vez que crean o actualizan entidades en las bases de datos ya que, si no se hiciera de esta forma, se podría dar que un servicio actualice la base de datos y luego se bloquee, antes de poder enviar el mensaje.

Por lo tanto, si el servicio no realiza estas dos operaciones atómicamente, una falla podría dejar el sistema en un estado inconsistente.

Englobados en esta categoría se identifican:

- Transactional outbox: Este patrón propone, para las bases de datos relacionales, utilizar una tabla de base de datos (denominada "OUTBOX") como una cola de mensajes temporal.

  De esta forma, cuando se ejecuta una transacción (create, update, delete, etc) además de realizar la operación tradicional, se inserta un nuevo registro en la tabla OUTBOX (Se garantiza la atomicidad ya que es una transacción local).

  Por otra parte, el modelado de este comportamiento se puede lograr en bases de datos NoSQL forzando a que cada entidad almacenada tenga un atributo especial cuyo valor sea una lista de mensajes que deben publicarse.

  En la Tabla A.32 se presenta la instanciación.

Nombre	Transactional outbox
Categoría	Mensajería Transaccional
Autor	Richardson
Relaciones	Complementario: Polling publisher Predecesor: Polling publisher Sucesor: Saga (Crea la necesidad de este patrón)
Componentes	Tabla "OUTBOX", Broker, "Message Relay"
Tecnologías	Debzium, LinkedIn Databus, DynamoDB, Eventuate Tram
Variables	

Table A.32: Transactional outbox

- Polling publisher: Este patrón es complementario al anterior presentado ya que provee una forma muy simple de publicar los mensajes insertados en la tabla OUTBOX. El enfoque consiste en sondear periódicamente la tabla en busca de mensajes no publica-

dos y cuando se detecta alguno, estos se envían al broker de mensajería y se eliminan de la tabla OUTBOX.

En la Tabla A.33 se presenta la instanciación.

Nombre	Polling publisher
Categoría	Mensajería Transaccional
Autor	Richardson
Relaciones	Complementario: Transactional outbox Sucesor: Transactional outbox
Componentes	Tabla "OUTBOX", Broker
Tecnologías	Eventuate Tram Framework (No todas las bases NoSQL soportan este patrón, si las SQL)
Variables	

Table A.33: Polling publisher

A.13 Implementación

Cómo implementar la lógica de negocios del sistema.

Para ello se identifican:

- Transaction Script: Usar transaction scripts es la manera más simple de organizar la lógica si esta es simple. Es presentado por Martin Fowler [3] y consta en organizar la lógica a base de procedimientos que se encargan por completo de las invocaciones de la capa de presentación. Reciben datos de entrada, los procesan, actualizan la base de datos y devuelven datos de respuesta a la capa de presentación.

  Dado que toda la lógica reside en un solo archivo, a medida que la aplicación va creciendo, se va complejizando tal y como pasa en la arquitectura monolítica. Por lo cual no se recomienda la aplicación de este patrón en casos donde la lógica de negocios implementada por la aplicación es compleja [79].

  En la Tabla A.34 se presenta la instanciación.

- Aggregates: Se sugiere estructurar la lógica en un conjunto de aggregates. Richardson en [1] los define como un conjunto de objetos de dominio que pueden ser tratados como una sola unidad. Lo cual deriva en que toda manipulación sobre un objeto concreto deba realizarse sobre el aggregate que compone.

  Los aggregates cumplen ciertas reglas:

  – Comunicación hacia un aggregate: Se requiere que la raíz sea la única entidad de un aggregate visible al exterior del mismo, es decir, la entidad raíz es a la única a la que puedan hacer referencia las clases externas.

Nombre	Transaction script
Categoría	Implementación
Autor	Fowler
Relaciones	Alternativo: Aggregates, Eventos de dominio
Componentes	Scripts
Tecnologías	Librerías de código usuales
Variables	

Table A.34: Transaction Script

  - Referenciación inter-aggregate: Entre dos distintos aggregates se deben referenciar mediante la utilización de clave primarias en lugar de referencias directas (buscando tener bajo acoplamiento entre aggregates).

En la Tabla A.35 se presenta la instanciación.

Nombre	Aggregates
Categoría	Implementación
Autor	Richardson
Relaciones	Alternativo: Transactional Scripts, Eventos de dominio
Componentes	Aggregates
Tecnologías	
Variables	Modelo de dominio de cada microservicio (Para identificar los aggregates)

Table A.35: Aggregates

- Eventos de dominio: Un concepto que toma relevancia en el ámbito de los microservicios es el de Domain Driven Design (DDD). El mismo es definido por Eric Evans en [80] como: "Un enfoque, para construir aplicaciones de software complejas, centrado en el desarrollo de un modelo de dominio orientado a objetos". Concretamente, este enfoque consta de tener un modelo de dominio general que contiene varios sub dominios pertenecientes a cada microservicio de la aplicación.

  Por otra parte, para comunicar los sucesos ocurridos en cada subdominio puntual, se emplea el concepto de evento de dominio, cuya función es notificar al resto, los sucesos ocurridos.

  Concretamente, cada aggregate genera un nuevo evento cada vez que experimenta un cambio que merece ser comunicado y lo difunde al resto del sistema.

  En la Tabla A.36 se presenta la instanciación.

- Event sourcing: Propone implementar toda la lógica basada en el concepto de evento

Nombre	Eventos de dominio
Categoría	Implementación
Autor	Eric Evans, Richardson, Fowler
Relaciones	Alternativo: Aggregates, Transactrional Scripts Sucesor: Event sourcing, Saga y CQRS (Generan la necesidad de contar con este patrón)
Componentes	Eventos, Subdominios (De cada microservicio)
Tecnologías	
Variables	Sucesos que dispararan eventos

Table A.36: Eventos de dominio

anteriormente presentado, incluso los aggregates, que se persisten en la base de datos como una serie de eventos donde cada uno representa un cambio de estado en dicho aggregate.

Dichos aggregates (secuencias de eventos) se persisten en un repositorio denominado “tienda de eventos", desde donde pueden ser recuperados y re-ensamblados a través de la serie de eventos que lo conforman.

Es importante notar que el hecho de que se pueda volver a generar un aggregate a partir de los eventos que lo conforman, se da gracias a dos cuestiones. Por un lado, todos los eventos generados por el sistema se guardan en dicho repositorio y por el otro, el sistema genera eventos para todas las acciones efectuadas sobre los aggregates de modo tal que sea posible volver a conformarlo únicamente a partir de dicha serie de eventos.

El mayor beneficio que este enfoque aporta radica en que se tiene un historial de cambios completo de cada aggregate existente lo cual puede ser muy provechoso a la hora de implementar mecanismos de auditoría y seguridad.

En la Tabla A.37 se presenta la instanciación.

Nombre	Event sourcing
Categoría	Implementación
Autor	Richardson, Fowler, Newman
Relaciones	Sucesor: Saga y CQRS (Generan la necesidad de contar con este patrón)
Componentes	Tienda de eventos, Eventos, Aggregates
Tecnologías	Lagoom, Axon Event Sourcing y CQRS), Eventuate Framework
Variables	Tecnología a utilizar

Table A.37: Event Sourcing

A.14 Preocupaciones transversales

Aspectos generales que involucran múltiples áreas del sistema.

- Ubicaciones en la red de diferentes servicios.
- Credenciales.
- Configuración dinámica y estática.
- Logueo de errores.
- Descubrimiento de servicios: Cómo encontrarlos y cómo registrarlos.
- Circuit breaker.
- Health Check APIs.
- Métricas.

Patrones identificados:

- Configuración exterior Gestión de la configuración del sistema.

  Se identifican dos variantes [81]:

  1. Push model: La infraestructura de despliegue proporciona la configuración a los servicios a través de variables o archivos de configuración.
  2. Pull model: Los servicios obtienen la configuración desde un servidor de configuración. Se centraliza la configuración lo que torna más sencillo la gestión de la misma. Tiene como principal limitante que el componente de configuración, a menos que lo brinde la infraestructura de despliegue, es una pieza más que se debe mantener y configurar.

  En la Tabla A.38 se presenta la instanciación.

Nombre	Configuración exterior
Categoría	Preocupaciones transversales
Autor	Richardson
Relaciones	Complementarios: Patrones de descubrimiento de servicios (Implementan parte de los aspectos referidos por este patrón)
Componentes	Servidor de configuración
Tecnologías	Spring Cloud Config
Variables	Configuraciones del sistema

Table A.38: Configuración exterior

- Microservices Chasis: Este patrón es referenciado desde diversas fuentes tales como los trabajos de Linjith Kunnon [82], Architectural Patterns [83], Akhil Raj [84], pero sin dudas que lo mas destacable es que Chris Richardson aportó sus ideas sobre el patrón en [85].

  El objetivo es abolir el hecho de que al desarrollar los múltiples microservicios que componen el sistema, usualmente se termina configurando y desarrollando varias veces lo mismo, generando pérdida de tiempo en el desarrollo. Para dicho propósito, se propone la utilización de distintos frameworks de manera conjunta teniendo como objetivo eliminar del alcance de los equipos encargados del desarrollo de cada microservicio las preocupaciones transversales.

  En la Tabla A.39 se presenta la instanciación.

Nombre	Microservices Chasis
Categoría	Preocupaciones transversales
Autor	Linjith Kunnon, Architectural Patterns, Akhil Raj, Richardson
Relaciones	Alternativo: Circuit Breaker, Registro personal de servicios, Descubrimiento de servicios del lado del cliente, Seguimiento distribuido (Implementa estos patrones a través de uso de múltiples frameworks)
Componentes	Frameworks
Tecnologías	Spring Boot, Spring Cloud, Dropwizard, Go Kit, Micro, Gizmo
Variables	Parámetros necesarios para los patrones que implementa

Table A.39: Microservices Chasis

A.15 Interfaz de usuario

Interacción con el usuario.

Dentro de las responsabilidades del equipo de desarrollo de cada servicio está el de la experiencia del usuario.

Sin embargo, la complejidad radica en que usualmente existirán pantallas en las cuales sea necesario presentar datos provenientes de múltiples servicios distintos. [86]

Existen dos enfoques para abordar este tema:

- Client-side UI composition: Cada equipo encargado de un servicio desarrolla el componente de interfaz de usuario que presenta datos del servicio en cuestión. Por otra parte, existe un equipo de UI general que es responsable de implementar los esqueletos de página utilizando los componentes desarrollados por los múltiples equipos de

servicios.

En la Tabla A.40 se presenta la instanciación.

Nombre	Client-side UI composition
Categoría	Interfaz de usuario
Autor	Richardson
Relaciones	Alternativo: Server-side page fragment composition
Componentes	Componentes de UI, Esqueletos de pagina
Tecnologías	Tecnologías de desarrollo frontend usuales
Variables	Diseño de la interfaz

Table A.40: Client-side UI composition

- Server-side page fragment composition: Cada equipo desarrolla una aplicación web que genera el fragmento HTML que implementa la región de la página principal que presenta información de su servicio.

  Existe un equipo de UI general responsable de desarrollar las plantillas que generan páginas mediante la agregación del lado del servidor de los fragmentos HTML específicos del servicio.

  En la Tabla A.41 se presenta la instanciación.

Nombre	Server-side page fragment composition
Categoría	Interfaz de usuario
Autor	Richardson
Relaciones	Alternativo: Client-side UI composition
Componentes	Fragmentos HTML, Plantillas de generación de paginas
Tecnologías	Tecnologías de desarrollo frontend usuales
Variables	Diseño de la interfaz

Table A.41: Server-side page fragment composition

Referencias

[1] Richardson, Chris. Microservices Patterns: With Examples in Java. Manning Publications, 2019.

[2] Richardson, Chris. https://microservices.io Blog.

[3] Fowler, Martin. https://martinfowler.com/articles/microservices.html Blog.

[4] Netflix OSS. https://netflix.github.io/ GitHub Oficial.

[5] Newman, Sam. Principles of Microservices. https://samnewman.io/talks/principles-of-microservices/ Publisher(s): O'Reilly Media, Inc. Released: August 2015 Duration: 2 hours 45 minutes Conference

[6] Newman, Sam. Building Microservices. https://learning.oreilly.com/library/view/building-microservices/9781491950340 Publisher(s): O'Reilly Media, Inc. Released: February 2015 ISBN: 9781491950357

[7] Newman, Sam. What Are Microservices. https://learning.oreilly.com/library/view/what-are-microservices/9781492074946/ Publisher(s): O'Reilly Media, Inc. Released: December 2019 ISBN: 9781492074939

[8] Yanaga, Edson. Migrating to Microservice Databases. https://www.oreilly.com/library/view/migrating-to-microservice/9781492048824/ Publisher(s): O'Reilly Media, Inc. Released: April 2017 ISBN: 9781491971864

[9] Microsoft. Design Microservices Patterns. https://azure.microsoft.com/de-de/blog/design-patterns-for-microservices/ Blog.

[10] Vogels, Werner. "Eventually consistent". https://doi.org/10.1145/1435417.1435432 Publisher(s): Association for Computing Machinery (ACM) Article 10.1145/1435417.1435432 Date:

January 2009. ISSN: 0001-0782

[11] Amazon. Docker. https://aws.amazon.com/es/docker/ Definición

[12] Márquez, G y Astudillo, H "Actual Use of Architectural Patterns in Microservices-Based Open Source Projects". 2018 25th Asia-Pacific Software Engineering Conference (APSEC) Nara, Japan, 2018 pp. 31-40, doi: 10.1109/APSEC.2018.00017

[13] Soldani, Jacopo. MicroFreshener https://github.com/di-unipi-socc/microFreshener GitHub Oficial

[14] Balalaie, A, Heydarnoori, A y Jamshidi, P. Microservices migration patterns Technical Report No. 1, TR- SUT-CE-ASE-2015-01 Automated Software Engineering Group Sharif University of Technology, 2015.

[15] Brown, K y Woolf, B. Implementation patterns of microservices architectures Conference on Pattern Languages of Programs (PLoP), pp. 7:1–7:35, 2016.

[16] NIST. Definición de computación en la nube y nube publica https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf Definición

[17] D. Taibi, V. Lenarduzzi, and C. Pahl. Architectural patterns for microservices: a systematic mapping study Proc. 8th Int. Conf. Cloud Computing and Services Science, 2018.

[18] Arcitura. http://microservicepatterns.org/ Web Especializada.

[19] Wasson, Mike. Design patterns for microservices https://azure.microsoft.com/en-us/blog/design-patterns-for-microservices/ Date: 2017

[20] Churchman, M. Top Patterns for Building a Successful Microservices Architecture https://www.sumologic.com/blog/devops/top-patterns-building-successful-microservices-architecture/ Date: 2017

[21] Gupta, A. Microservice Design Patterns https://progressivewebexperience.io/blog/arun_gupta/2015/04/microservice_design_patterns Date: 2015

[22] MuleSoft. The top six microservices patterns - how to choose the right architecture for your organization https://www.scribd.com/document/348903027/The-Top-6-Microservices-Patterns Date: 2017

[23] PloP Conference. https://conf.researchr.org/series/PLOP

[24] Most Popular Programing Lenguaje in GitHub.

https://www.businessinsider.com/most-popular-programming-languages-github-2019-11

[25] GitHub. https://github.com/ Web Oficial.

[26] Uso de GitHub. https://www.xataka.com/makers/como-y-por-que-se-esta-usando-github-mas-alla-del-software

[27] Node Js https://nodejs.org/en/ Web Oficial.

[28] Framework Spring https://spring.io/ Web Oficial.

[29] Sánchez, C., Tuesta, V. y Mejía, I. Análisis comparativo de framework para el desarrollo de aplicaciones web en java Ingeniería: Ciencia, Tecnología e Innovación. Date: Julio 2015 ISSN: 2313-1926

[30] Framework Eventuate https://eventuate.io/ Web Oficial.

[31] Netflix OSS Kystrix https://github.com/Netflix/Hysvatrix GitHub Oficial.

[32] Implementación de Circuit Breaker con Spring https://spring.io/guides/gs/circuit-breaker/

[33] Fowler, Martin. Circuit Breaker https://martinfowler.com/bliki/CircuitBreaker.html Blog.

[34] Richardson, Chris. Circuit Breaker https://microservices.io/patterns/reliability/circuit-breaker.html Blog.

[35] Libreria Resilience4j. https://github.com/resilience4j/resilience4j GitHub Oficial

[36] The Future of Spring Cloud's Hystrix Project. https://www.infoq.com/articles/spring-cloud-hystrix

[37] Gestor NPM. https://www.npmjs.com/about Web Oficial.

[38] Red Hat. Node Shift https://nodeshift.dev/ Web Oficial.

[39] Red Hat https://www.redhat.com/ Web Oficial.

[40] Red Hat Opossum https://github.com/nodeshift/opossum/network/dependents?package_id GitHub Oficial.

[41] Cockatiel https://github.com/connor4312/cockatiel/network/dependents?package_id=UGl GitHub Oficial.

[42] Descargas semanales Redux Saga https://www.npmjs.com/package/redux-saga Documentación NPM.

[43] Redux Saga https://redux-saga.js.org/ Documentación Oficial.

[44] Librería Debezium https://debezium.io/ Documentación Oficial

[45] Apache Kafka https://kafka.apache.org/ Documentación Oficial.

[46] Libreria Zookeeper https://zookeeper.apache.org/ Documentación Oficial.

[47] Informacion Hystrix https://github.com/Netflix/Hystrix/commit/a7df971cbaddd8c5e976b3cc5f14

[48] Netflix OSS en Java https://tecnologia4dev.wordpress.com/2017/04/14/netflix-oss-una-genialidad-escrita-en-java/

[49] Al-Houmaily, Yousef J. and Chrysanthis, Panos K. The One-Two Phase Atomic Commit Protocol https://doi.org/10.1145/967900.968044 ISBN 1581138121 Publisher: Association for Computing Machinery

[50] Docker https://www.docker.com/ Página Web Oficial de Docker

[51] Venners, Bill. The java virtual machine Publisher: McGraw-Hill, New York

[52] MY SQL. https://www.mysql.com/

[53] Fowler, Martin. https://martinfowler.com/articles/break-monolith-into-microservices.html Definición de capacidad de negocio.

[54] Mensajería. https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture

[55] Base de datos compartida. https://medium.com/@matipazw/bases-de-datos-monol%C3%ADticas-en-un-mundo-de-microservicios-5e6dd572d59e

[56] Saga. https://www.redhat.com/files/summit/session-assets/2019/T42224.pdf Consistencia en sistemas distribuidos.

[57] Fowler, Martin. https://martinfowler.com/bliki/CQRS.html CQRS.

[58] IBM. https://ibm-cloud-architecture.github.io/refarch-eda/design-patterns/cqrs/ CQRS.

[59] Microsoft. https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs CQRS.

[60] Pierlot, Romain. https://medium.com/eleven-labs/cqrs-pattern-c1d6f8517314 CQRS.

[61] IBM. https://www.ibm.com/garage/method/practices/manage/practice_circuit_breaker_pattern Circuit Breaker.

[62] RedHat https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh Service Mesh.

[63] Nginx https://www.nginx.com/blog/what-is-a-service-mesh/ Service Mesh.

[64] Indrasiri, Kasun. https://medium.com/microservices-in-practice/service-mesh-for-microservices-2953109a3c9a Service Mesh.

[65] Shetty, Rahul. https://www.commonlounge.com/discussion/722433ee5c3c4664852a2d79c single-instance-of-a-microservice-per-vm Un servicio por contenedor.

[66] AWS. https://aws.amazon.com/es/serverless/ Despliegue sin servidores.

[67] Serverless Stack https://serverless-stack.com/chapters/es/what-is-serverless.html Despliegue sin servidores.

[68] Nginx https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/ Descubrimiento de servicios del lado del cliente.

[69] Auth0 https://auth0.com/docs/tokens/concepts/access-tokens Tipos de Tokens.

[70] Microsoft https://docs.microsoft.com/en-us/azure/architecture/patterns/strangler Strangler.

[71] Fowler, Martin. https://martinfowler.com/bliki/StranglerFigApplication.html Strangler.

[72] IBM https://developer.ibm.com/technologies/microservices/articles/cl-strangler-application-pattern-microservices-apps-trs/ Strangler.

[73] Behara, Samir. https://dzone.com/articles/monolith-to-microservices-using-the-strangler-patt Strangler.

[74] Microsoft https://docs.microsoft.com/en-us/azure/architecture/patterns/anti-corruption-layer Anti Corruption Layer.

[75] Shirin, Ahmed. https://dev.to/asarnaout/the-anti-corruption-layer-pattern-pcd Anti Corruption Layer.

[76] Microsoft https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern API Gateway.

[77] Nginx https://www.nginx.com/blog/building-microservices-using-an-api-gateway/ API Gateway.

[78] AWS https://aws.amazon.com/es/blogs/architecture/using-api-gateway-as-a-single-entry-point-for-web-applications-and-api-microservices/ API Gateway.

[79] Peipan, Gunnar. https://dzone.com/articles/transaction-script-pattern Transaction Script Pattern.

[80] Evan, Eric. Microsoft https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-events-design-implementation Domain Event.

[81] Duarte, Alejandro https://vaadin.com/blog/microservices-externalized-configuration Configuración exteriorizada.

[82] Kunnon, Linjith. https://medium.com/@linjith/micro-service-chassis-9709ad8044ff Microservices Chassis.

[83] Pethuru Raj, Anupama Raman, Harihara Subramanian. https://learning.oreilly.com/library/view/architectural-patterns/9781787287495/c6b174ce-4cdb-400c-96d4-04297c8fde3a.xhtml Microservices Chassis.

[84] Raj, Akhil. https://dzone.com/articles/ms-chassis-pattern Microservices Chassis.

[85] Richardson, Chris. https://www.youtube.com/watch?v=57anw5XBl98 Microservices Chassis.

[86] Fowler, Martin. https://martinfowler.com/articles/micro-frontends.html Micro Frontends.

[87] Microsoft. https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker Circuit Breaker.

[88] N. Dragoni, I. Lanese, S.T. Larsen, M. Mazzara, R. Mustafin, L. Safina "Microservices: How To Make Your Application Scale". In: Petrenko A., Voronkov A. (eds.) Perspectives of System Informatics. PSI 2017. Lecture Notes in Computer Science, 10742., Springer, Cham (2017) https://link.springer.com/chapter/10.1007/978-3-319-74313-4_8

[89] Dragoni N. et al. (2017) Microservices: Yesterday, Today, and Tomorrow. In: Mazzara M., Meyer B. (eds) Present and Ulterior Software Engineering. Springer, Cham https://link.springer.com/chapter/10.1007/978-3-319-67425-4_12#citeas

[90] H. Mu and S. Jiang "Design patterns in software development," 2011 IEEE 2nd International Conference on Software Engineering and Service Science, Beijing, 2011, pp. 322-325, doi: 10.1109/ICSESS.2011.5982228.

[91] Gos, Konrad, and Wojciech Zabierowski. "The Comparison of Microservice and Monolithic Architecture." IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEM-

STECH), Perspective Technologies and Methods in MEMS Design (MEMSTECH), 2020 IEEE XVIth International Conference on The, April, 150–53. doi:10.1109/MEMSTECH49584.2020.9109514.

[92] Plataforma de orquestación de contenedores. https://kubernetes.io/es/docs/home/ Kubernetes.

[93] Netflix Tech Blog https://netflixtechblog.com/tagged/microservices

[94] Portal Timbó Foco https://foco.timbo.org.uy/

[95] Google Académico https://scholar.google.es/schhp?hl=es

[96] Vavr https://www.vavr.io/

[97] Archiaus https://github.com/Netflix/archaius

[98] A. Balalaie, A. Heydarnoori, P. Jamshidi Microservices Migration Patterns, Technical Report No. 1 TRSUT-CE-ASE-2015-01, Automated Software Engineering Group, Sharif University of Technology, October, 2015 http://ase.ce.sharif.edu/pubs/techreports/TR-SUT-CE-ASE-2015-01-Microservices.pdf

[99] Descripción ReactJs https://www.drauta.com/que-es-react-y-para-que-sirve

[100] MongoDB https://www.mongodb.com/es

[101] Mongoose https://www.npmjs.com/package/mongoose

[102] GitLab https://about.gitlab.com

[103] Git https://git-scm.com/

[104] Mi Nube Antel https://minubeantel.uy/

[105] ReactJs https://es.reactjs.org