Máster en Cloud Apps
Desarrollo y despliegue de aplicaciones en la nube

Curso académico 2020/2021

Trabajo de Fin de Máster

Monolith to Microservices - Examples

Autores: Juan Carlos Blázquez Muñoz
Miguel Ángel Huerta Rodríguez
Tutor: Micael Gallego Carrillo

Índice

1. Resumen

A lo largo de este documento, vamos a realizar un repaso del proyecto final de máster *Monolith to Microservices – Examples* donde investigamos diferentes patrones y sugerencias de migración de monolitos a microservicios que pueden ayudarnos a adoptar una arquitectura de microservicios.

Pondremos en contexto las necesidades y motivaciones que nos han llevado a la realización de este trabajo, comentando las diferentes tecnologías y objetivos planteados en el mismo.

Centrándonos en el estudio del capítulo 3, *Splitting The Monolith*, del libro *Monolith to Microservices* de *Sam Newman* [1], trataremos la partición del monolito a través del código, dejando a un lado, dentro de lo posible, la descomposición de la Base de Datos, tratada en el capítulo 4.

Estudiaremos por tanto, una amplia gama de métodos para la descomposición incremental del código existente brindando una gran variedad de enfoques y alternativas que permitan determinar qué técnicas pueden funcionar mejor en cada situación.

Analizamos patrones en múltiples escenarios sobre sistemas monolíticos basados en el ecosistema Spring apoyándonos en diferentes tecnologías y herramientas actuales.

Valoraremos los pros y contras de cada uno de los patrones dependiendo del contexto, la tecnología y las herramientas disponibles.

Finalmente, argumentaremos las conclusiones a las que hemos llegado a través de la realización del trabajo, apoyándonos en los conocimientos, habilidades y aptitudes adquiridas a lo largo de la realización de este proyecto. Además, plantearemos la posibilidad de ampliar el estudio desarrollado en este proyecto mediante el uso de otras tecnologías y herramientas.

Hemos facilitado la documentación de los proyectos en el repositorio *GitHub*, https://github.com/MasterCloudApps-Projects/Monolith-to-Microservices-Examples con una versión en español y otra en inglés, acompañados de unos vídeos cortos de apoyo agrupados en una lista de reproducción en *YouTube* que permita a cualquiera ejecutar y comprender la resolución de los patrones en las diferentes casuísticas planteadas.

[1] Monolith to Microservices - https://www.oreilly.com/library/view/monolith-to-microservices/9781492047834

2. Introducción y objetivos

La arquitectura basada en microservicios está formando parte cada vez más de los diferentes proyectos de las grandes empresas tecnológicas.

Hoy en día, en el entorno empresarial, lo más habitual es tener grandes aplicaciones monolíticas que han ido creciendo a lo largo del tiempo incorporando nuevas funcionalidades y lógicas de negocio que, sumado al incremento de usuarios, provoca que cada vez sea más complicado su mantenimiento y lograr que ofrezcan un servicio de calidad.

Con la llegada de los microservicios, se plantea la posibilidad de iniciar una migración progresiva y gradual de las funcionalidades de este antiguo sistema a nuevos servicios que se van creando paulatinamente.

Para lograr que el sistema siga funcionando y coexista con las nuevas implementaciones de su funcionalidad en microservicios, es necesario el uso de diferentes técnicas para abordar el problema planteado que difieren en función del enunciado.

Cuando abarcamos el estudio de una nueva tecnología, arquitectura o herramienta el punto principal que nos permite avanzar en la curva de aprendizaje es la realización de ejemplos prácticos, a ser posible, a través de casuísticas reales y su ejecución en nuestra máquina, comprendiendo los pasos que nos llevan a la solución final.

La teoría puede ser fácil de entender, pero otra cosa bien distinta es llevar ese conocimiento a un caso real.

Nuestro principal objetivo del proyecto es la comprensión y aplicación en código, con diferentes tecnologías, de ejemplos de los múltiples patrones planteados por *Sam Newman* en el capítulo 3, *Splitting The Monolith*, del libro *Monolith to Microservices,* valorando los pros y contras de cada uno de ellos dependiendo del contexto, cuidando al detalle la arquitectura de la información, empleando los mismos conceptos, nombres y modelos facilitados.

Además, a través del desarrollo de este trabajo, hemos podido:

- Aplicar diferentes formas de comunicación entre los microservicios, desde peticiones HTTP hasta comunicación mediante colas de mensajería como *Kafka* [2].
- Utilizar proxys, tanto con *NGINX* [3], incluso utilizándolo como balanceador de carga, como con *Spring Cloud Gateway* [4].
- Familiarizarnos con el proceso de contenerización y despliegue de aplicaciones en caliente utilizando y ampliando nuestros conocimientos en *Docker*.
- Utilización de librerías y herramientas específicas como:
    - *FF4J* [5]
    - *Diffy* [6]
    - *Scientist4J* [7]
    - *Debezium* [8]

[2] Kafka - https://kafka.apache.org/
[3] NGINX - https://www.nginx.com
[4] Spring Cloud Gateway - https://spring.io/projects/spring-cloud-gateway
[5] FF4J - https://ff4j.org
[6] Diffy - https://github.com/opendiffy/diffy
[7] Scientist4J - https://github.com/rawls238/Scientist4J
[8] Debezium - https://debezium.io

- Gestión de operaciones concurrentes con variables atómicas y *JPA Optimistic locking* [9].
- Realización de microservicios *Batch* con *Spring Batch* [10].

Hemos conseguido disponer de ejemplos útiles de los diferentes patrones, utilizando enunciados sencillos extrapolables a las casuísticas con las que nos encontramos en nuestro trabajo diario.

[9] JPA Optimistic locking - https://www.baeldung.com/jpa-optimistic-locking
[10] Spring Batch - https://spring.io/projects/spring-batch

3. Patrones de descomposición de monolitos

Los patrones han sido divididos en ejemplos con diferentes casuísticas y enunciados. Hemos orientado la documentación de *GitHub* a tener una ligera explicación del patrón, centrándonos sobre todo en las partes más importantes de la implementación, mostrando detalles de las configuraciones y secciones del código con mayor peso.

Además, cada ejemplo tiene una ejecución dividida en diferentes pasos que permiten tener una similitud con casos reales y facilitan la compresión del patrón.

3.1. Strangler Fig

Esta técnica fue mencionada por primera vez por *Martin Fowler*, el cual se inspiró para nombrar este patrón en un tipo de planta que se apoya en el árbol en el que nace hasta llegar a sus raíces o ramas. El árbol existente se convierte inicialmente en una estructura de soporte para la nueva planta. Según avanza el crecimiento de la planta, poco a poco el árbol original que sirvió de sustento muere y se pudre, dejando solo la nueva.

La idea es que lo viejo y lo nuevo puedan coexistir, dando tiempo al nuevo sistema para crecer y potencialmente reemplazar por completo al antiguo.

Uno de los puntos principales de este patrón, es la capacidad de asegurarnos de que cada paso es fácilmente reversible, reduciendo el riesgo de cometer errores y en el caso de cometerlos que su impacto sea mínimo.

El patrón *Strangler Fig* consiste en la migración de forma incremental y gradual de las funcionalidades específicas situadas dentro del monolito a microservicios independientes. Se divide en 3 pasos que hemos mantenido a lo largo de los ejemplos:

1. Monolito. Las peticiones y funcionalidades se responden dentro del sistema monolítico.
2. Implementación de la funcionalidad en un nuevo microservicio.
3. Con su nueva implementación lista, migramos las peticiones del monolito al microservicio.

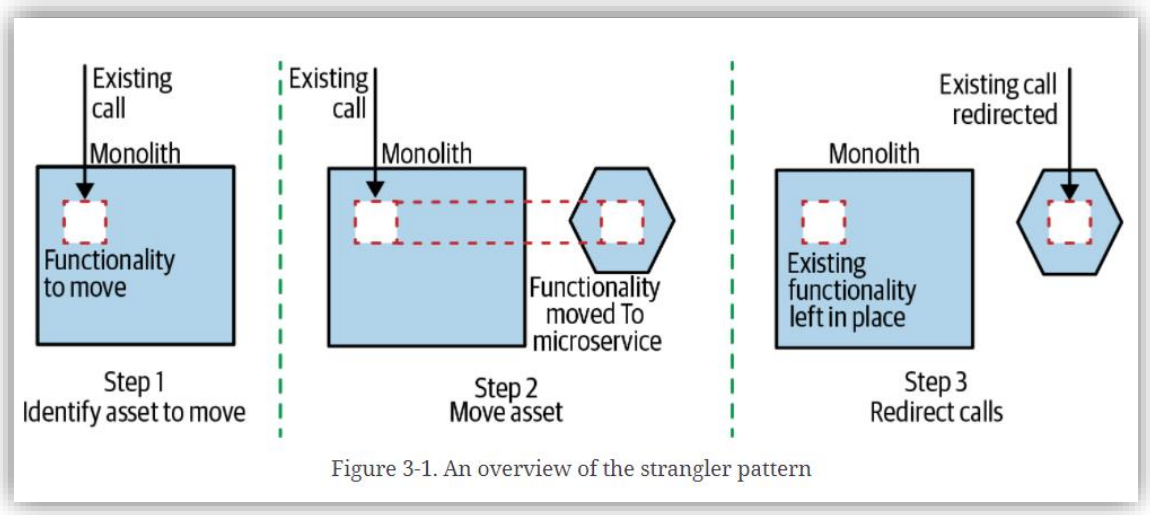

Figure 3-1. An overview of the strangler pattern

Ilustración 1 – Esquema del patrón Strangler Fig

Ejemplo 1. Extracción de funcionalidad independiente

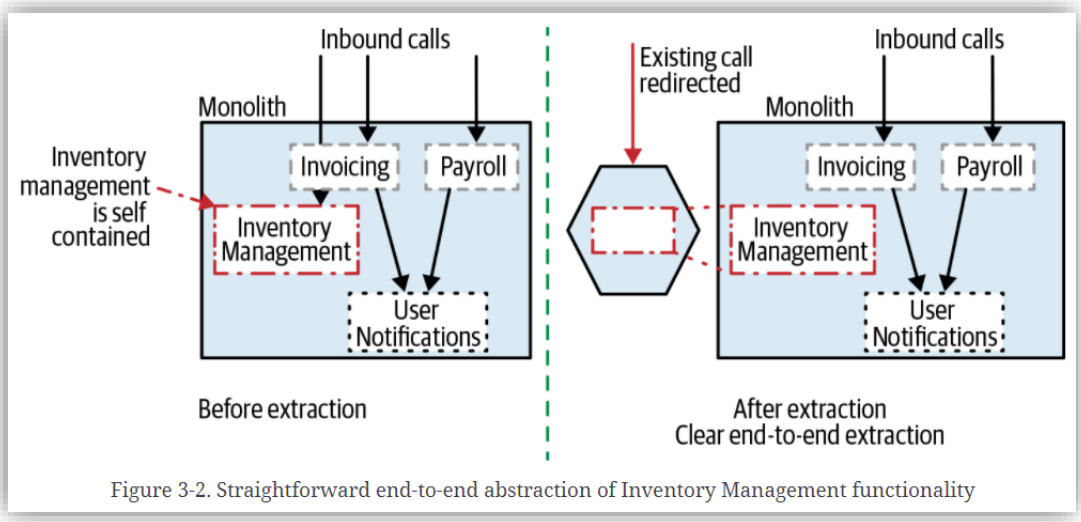

Figure 3-2. Straightforward end-to-end abstraction of Inventory Management functionality

Ilustración 2 – Esquema de la extracción de una funcionalidad independiente

Aplicando los pasos comentados anteriormente:

1. Nuestro monolito, todas las peticiones se responden desde dentro del sistema.
2. Creamos nuestro microservicio, copiamos el código de *Inventory Management*.
3. Redirigimos las peticiones de *Inventory Management* para que vayan a nuestro microservicio.

En caso de error, podemos migrar las peticiones a la configuración inicial.

Ejemplo 2. Extracción de funcionalidad interna

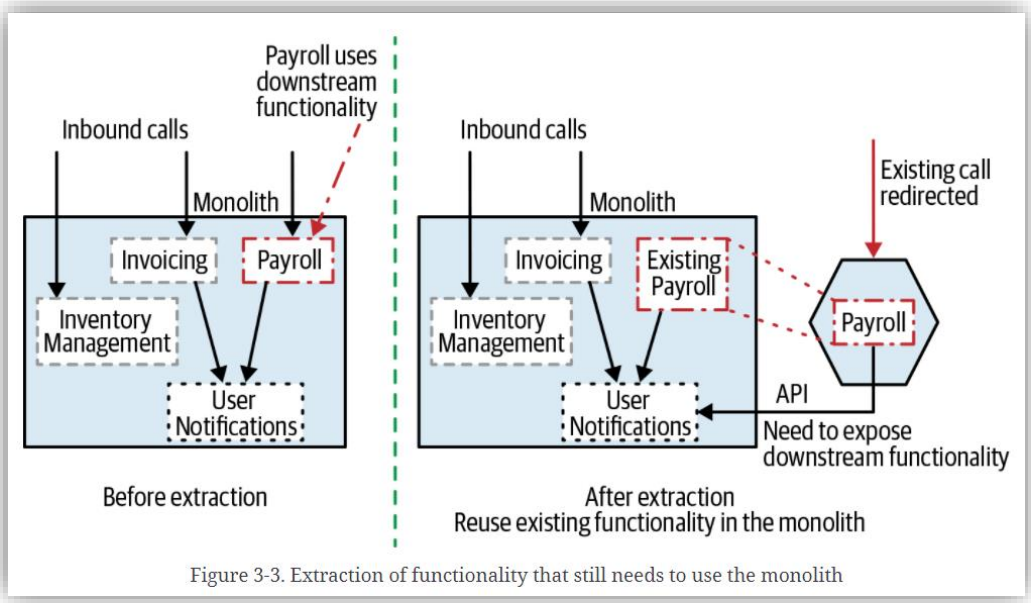

Figure 3-3. Extraction of functionality that still needs to use the monolith

Ilustración 3 – Extracción de una funcionalidad interna

Aplicando los pasos comentados anteriormente:

1. Nuestro monolito, todas las peticiones se responden desde dentro del sistema.
2. Creamos una versión 2 del monolito en el que expongamos una nueva funcionalidad, *User Notifications*. Creamos nuestro microservicio, basándonos en el código de *Payroll* y realizando una petición al monolito cuando debamos enviar una notificación al usuario.
3. Redirigimos las peticiones de *Payroll* al microservicio y el resto de las notificaciones a la versión 2 del monolito.

En caso de error, podemos migrar las peticiones a la configuración inicial.

Ejemplo 3. Interceptación de mensajes

La utilización de *Kafka* en este ejemplo marca por completo la implementación y los problemas encontrados a la hora de aplicarlo a un ejemplo, debido a que en *Kafka* no se pueden filtrar los mensajes que quieres o no leer de un *topic*.

Hemos intentado ser fieles a la aplicación de los 3 pasos y a los diagramas orientativos.

Para nutrir de mensajes a la cola de mensajería, hemos creado un microservicio *Producer* que genera datos a partir de una petición *HTTP*. Haría la función del *Upstream system/User interface*.

Los ejemplos se basan en la extracción de la funcionalidad de *Payroll,* para poder comparar con el ejemplo anterior.

Ejemplo 3 a). Podemos cambiar el código del monolito

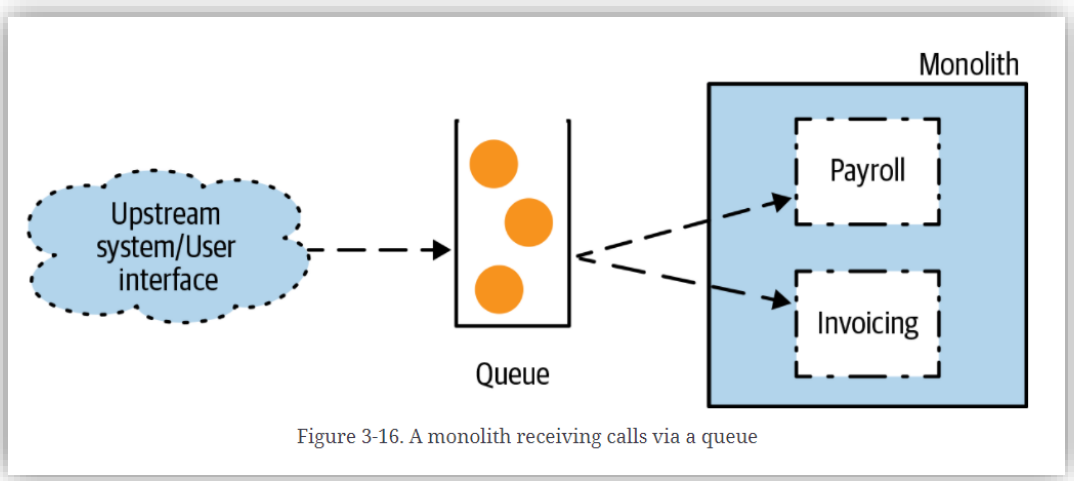

Figure 3-16. A monolith receiving calls via a queue

Ilustración 4 – Monolito recibiendo peticiones de una cola de mensajería

1. Nuestro monolito. Consume datos escritos desde el *Producer* en *invoicing-v1-topic* y *payroll-v1-topic*.
2. Creamos una versión 2 del monolito en el que expongamos una nueva funcionalidad, *User Notifications*. Creamos nuestro microservicio, basándonos en el código de *Payroll* y realizando una petición al monolito cuando debamos enviar una notificación al usuario. El monolito consume datos escritos desde el *Producer* en *invoicing-v2-topic* y el microservicio consume datos de *payroll-v2-topic*.

3. En este punto, la redirección de mensajes la realizamos lanzando una nueva versión de nuestro microservicio generador de datos *Producer*, el cuál ahora escribe mensajes en *invoicing-v2-topic* y *payroll-v2-topic*.

En caso de error, podemos lanzar la versión inicial del *Producer,* escribiendo en *invoicing-v1-topic* y *payroll-v1-topic*.

Ejemplo 3 b). No podemos cambiar el código del monolito

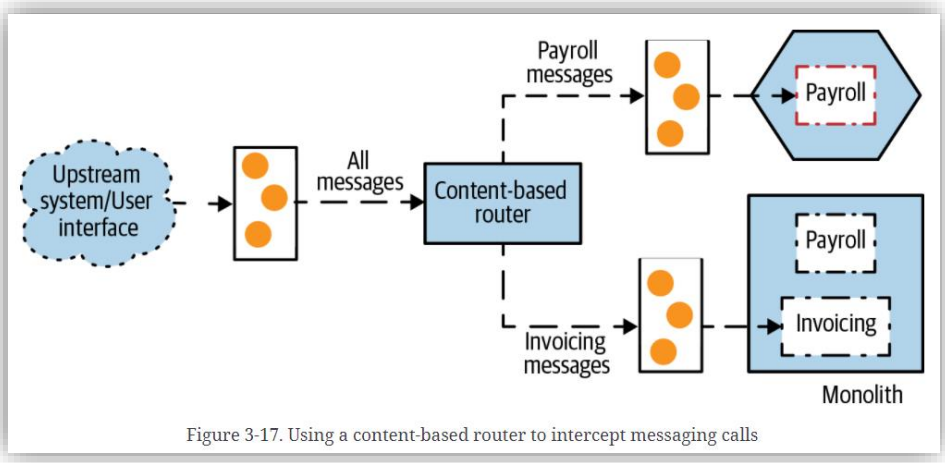

Figure 3-17. Using a content-based router to intercept messaging calls

Ilustración 5 – Utilización de un enrutador de peticiones

En este ejemplo, hemos partido de la premisa de que no podemos tampoco cambiar la configuración del monolito, por lo que no podemos cambiar los *topics* a los que se suscribe.

Para intentar seguir por completo el diagrama anterior, hemos seguido el siguiente flujo:

- Llega una petición *POST* a nuestro *Producer*.
- Genera un mensaje a la cola de *Kafka* a *invoicing-all-msg-topic* o *payroll-all-msg-topic*. Nuestro microservicio *Content-based-router (cbr)* es un microservicio de enrutamiento basado en contenido que consume y redirige los *topics* con destino:
    - *payroll-v1-topic* – Monolito.
    - *payroll-v2-topic* – Microservicio de *Payroll.*
- El *payroll-v1-topic* se quedaría sin uso puesto que vamos a redirigir los mensajes a la versión 2.

Aplicándolo en los 3 pasos del patrón:

1. Nuestro monolito. Consume datos escritos desde el *Producer* en *invoicing-v1-topic* y *payroll-v1-topic*.
2. No podemos sacar una versión 2 del monolito, por lo que no podemos exponer una nueva funcionalidad *User Notifications*. Al crear nuestro microservicio *Payroll* implementamos la funcionalidad de generación de notificaciones al usuario. Creamos el microservicio enrutador *cbr*.
3. Realizamos la redirección de mensajes lanzando una nueva versión de nuestro microservicio generador de datos *Producer*, el cuál ahora escribe mensajes en *invoicing-all-msg-topic* y *payroll-all-msg-topic*.

En caso de error, podemos lanzar la versión inicial del *Producer,* escribiendo en *invoicing-v1-topic* y *payroll-v1-topic*.

Ejemplo 3 c). No podemos cambiar la fuente de datos

Este ejemplo se nos ocurrió tras un debate en el que nos plantemos qué sentido tenía realizar este tipo de migraciones si estábamos cambiando la fuente de datos, cuando en nuestro día a día, lo más común iba a ser que esa variable fuera fija y no pudiéramos cambiarla.

Para ello, partimos de una versión ampliada del monolito, que dispone de un *flag* de *FF4J* como los utilizados en el patrón explicado a continuación *Branch by Abstraction*.

Necesitamos lanzar una versión 1.1 del monolito que incorpore el uso de este *flag* y con el cuál podamos seleccionar si deseamos consumir datos desde el *topic* o no.

Aplicándolo en los 3 pasos del patrón:

1. Nuestro monolito 1.1. Consume datos escritos desde el *Producer* en *invoicing-v1-topic* y *payroll-v1-topic*. Tiene un *flag* que le permite habilitar o deshabilitar la consumición de datos de *Payroll*.
2. Despliegue de nuestro microservicio de *Payroll* y desactivación del consumo de datos de *payroll-v1-topic.*
3. Una vez probado todo y viendo que funciona correctamente, eliminación del uso del *flag* y actualización del monolito eliminando el código no utilizado de *Payroll.*

En caso de error, podemos cambiar el *flag* y mantener el consumo de mensajes de *Payroll* en el monolito.

3.2. Branch by Abstraction

Hemos tratado en el patrón *Strangler Fig* la extracción de la funcionalidad de *Inventory* y de *Payroll* la cual sirve de guía para poder extraer *Invoicing*, pero ¿qué ocurre si deseamos extraer la funcionalidad de *User Notification*?

Nos situamos en el caso de que necesitamos migrar un código interior del monolito el cuál recibe peticiones internas de otros servicios.

Para realizar esta extracción, debemos realizar modificaciones en el sistema ya existente, pudiendo tener cambios significativos y perjudiciales para otros desarrolladores que utilicen esta funcionalidad. Queremos realizar cambios incrementales sin interrumpir a otras personas que estén desarrollando sobre el código base.

Es muy común que al afrontar este tipo de cambios, utilicemos una rama de código fuente separada, de esta forma no afectamos a otros desarrolladores, pero generamos un problema a la hora de integrar los cambios.

Branch by Abstraction se basa en permitir que dos implementaciones del mismo código coexistan en la misma versión, sin romper la funcionalidad. Es importante recalcar que el patrón asume que podemos cambiar el código del sistema existente.

Se aplica en múltiples pasos:

1. Crear una abstracción para reemplazar la funcionalidad.
2. Cambiar los clientes de la funcionalidad existente para utilizar la nueva abstracción.
3. Crear una nueva implementación de la abstracción que realice la petición a nuestro nuevo microservicio.
4. Cambiar la abstracción para usar nuestra nueva implementación.
5. Limpiar la abstracción y eliminar la implementación anterior.
6. (Opcional): Borrar la interfaz.

Esta técnica es muy utilizada en el modelo de desarrollo *Trunk-based development*, donde los desarrolladores colaboran en una única rama.

Ejemplo 1. Extracción de funcionalidad dependiente

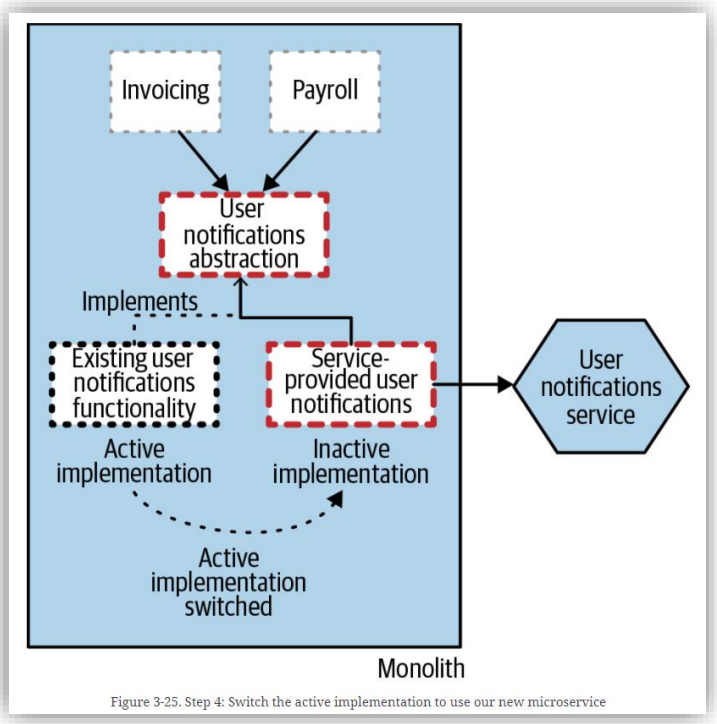

Ilustración 6 – Esquema del patrón Branch by abstraction

Estos pasos son fácilmente aplicables en un entorno de integración y entrega continuas CI/CD donde podamos desplegar nuestros cambios de forma rápida y sencilla.

Aplicando los pasos comentados anteriormente:

1. Creamos la interfaz *UserNotificationService*.
2. Adaptamos la implementación de *UserNotificationService* (que pasa a llamarse *UserNotificationServiceImpl*) existente para utilizar la interfaz.
3. Creamos una nueva implementación de la interfaz, *UserNotificationServiceMSImpl* que realizará una petición a nuestro nuevo microservicio. Es importante entender que en este punto aunque tengamos dos implementaciones de la abstracción en el código al mismo tiempo, solo una implementación está actualmente activa. Cuando terminemos de desarrollar nuestra nueva implementación y esté lista, podremos pasar al siguiente paso.
4. Modificamos a través de *FF4J* el uso de una u otra implementación en tiempo de ejecución.

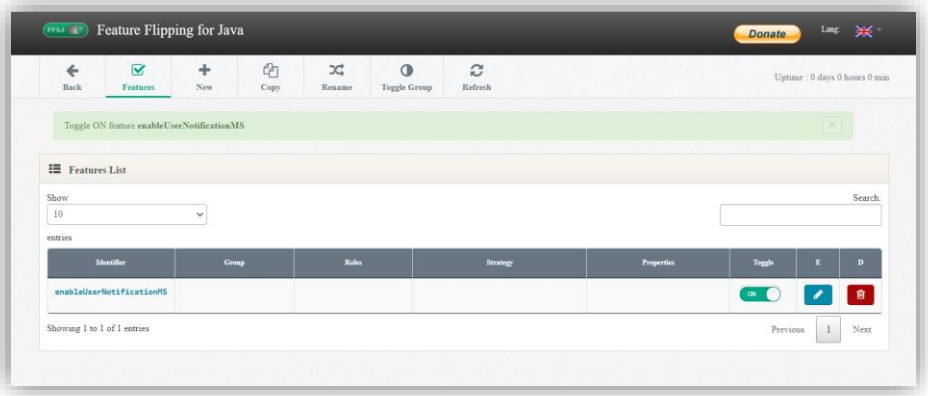

Ilustración 7 – Feature Flags For Java

5. Eliminamos el *flag* y la implementación antigua.
6. (Opcional): Borrar la interfaz.

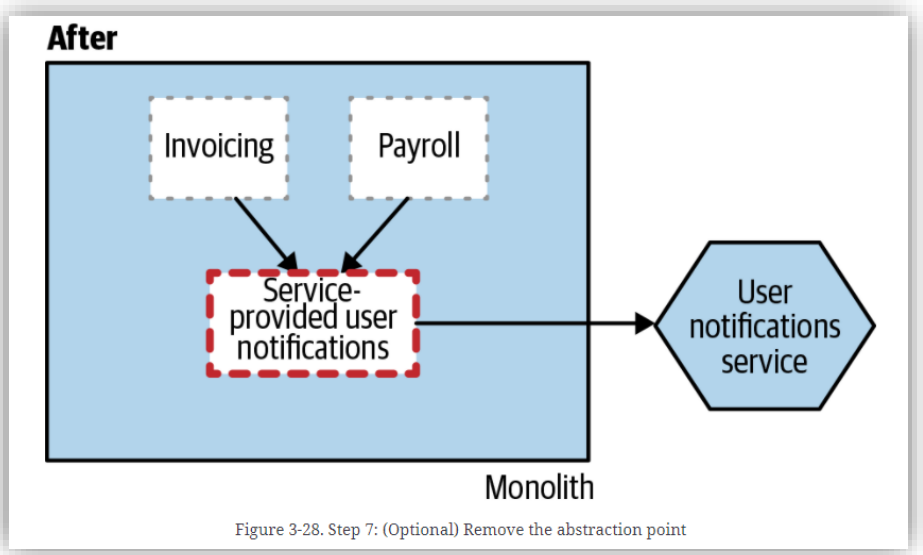

Ilustración 8 – Estado del sistema tras el paso 6

3.3. Parallel Run

En los patrones anteriormente estudiados *Strangler Fig* y *Branch By Abstraction*, teníamos la posibilidad de que coexistiera la versión antigua y nueva de la funcionalidad, pero solamente una de ellas se activaba en un momento concreto.

Este patrón, *Parallel Run* en lugar de llamar a la implementación antigua o nueva, llama a ambas, lo que nos permite comparar los resultados para asegurarnos de que sean equivalentes.

Esta técnica puede usarse para verificar que nuestra nueva implementación está dando las mismas respuestas que la implementación previamente existente y también que está operando dentro de parámetros no funcionales aceptables, tiempos de respuesta, latencias, etc.

Ejemplo 1. Uso de Spies

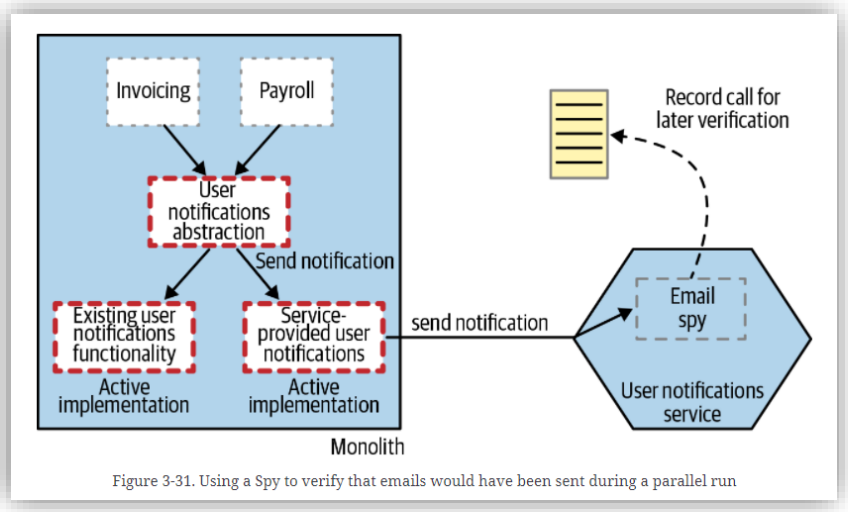

Figure 3-31. Using a Spy to verify that emails would have been sent during a parallel run

Ilustración 9 – Parallel Run Spies

En el caso de nuestro ejemplo de notificaciones anterior, no queremos enviar un correo electrónico a nuestro usuario dos veces. En esa situación, el uso de espías nos es realmente útil.

El concepto es el mismo que el utilizado en pruebas unitarias, un *Spy* representa una funcionalidad y nos permite verificar después que se hicieron ciertas cosas.

Entonces, para nuestra nueva funcionalidad de notificaciones implementada en el microservicio *User Notifications*, hemos reemplazado el código de envío de mails, para poder ejecutar ambas implementaciones de forma paralela, de esta forma sólo se registra que se ha enviado un mail sin haberlo hecho realmente.

En este ejemplo, hemos implementado un proceso *Batch* que captura todos los registros de envío de mails del monolito y del microservicio y una vez al día los compara, permitiéndonos ver si el funcionamiento de la nueva implementación es la correcta, pudiendo plantear el uso del microservicio de forma única a futuro.

Ejemplo 2. Github Scientist

En este ejemplo del patrón *parallel run*, *Sam Newman* sugiere el uso de una librería concreta, *scientist*, en nuestro caso *scientist4j*. Esta dependencia añadida a nuestro proyecto permite realizar lo que denomina como "experimentos", comparaciones síncronas y asíncronas de los resultados de llamadas a funciones *Java*.

Por lo tanto, permite realizar comparaciones del código implementado en el microservicio y del código existente en el monolito.

Es posible que esta técnica requiera ser combinada con la anteriormente mencionada, *Spies* debido a que no queremos duplicar información.

Además, la librería tiene un amplio soporte con otras dependencias de métricas que nos permiten tomar medidas y comprobar si la nueva implementación cumple los requisitos no funcionales del proyecto.

Ejemplo 3. Diffy

Hemos realizado una investigación de la herramienta *Diffy* la cual permite encontrar posibles errores en un servicio utilizando instancias en ejecución en paralelo de nuestro código antiguo y nuevo.

Diffy se comporta como un proxy, realiza comparaciones de las respuestas a las peticiones HTTP e informa de cualquier error que pueda surgir en esas comparaciones.

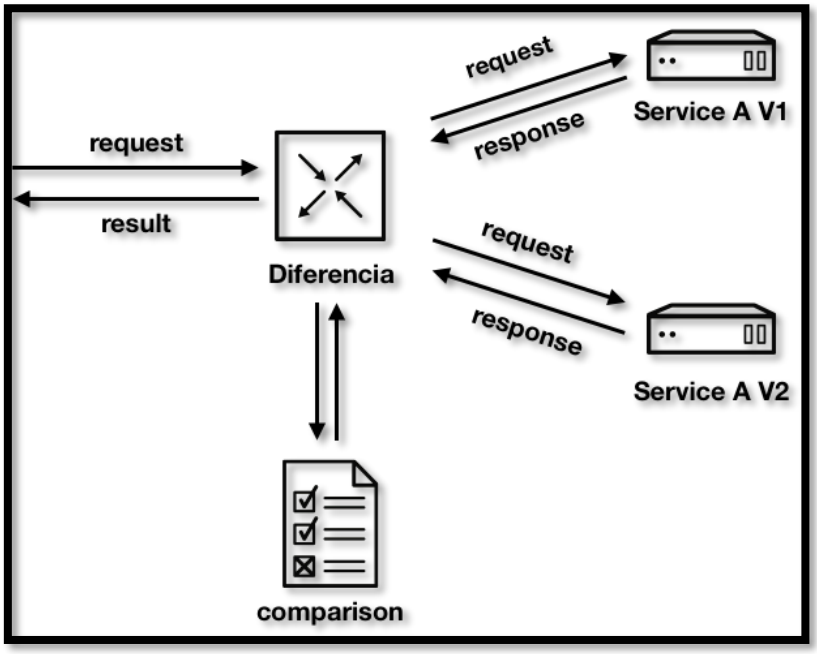

Ilustración 10 – Esquema de funcionamiento de Diffy [2]

La herramienta cuenta con múltiples métricas y configuraciones que nos permiten comprobar de forma fiable si la nueva implementación cumple los requisitos no funcionales del proyecto.

Es posible que esta técnica requiera ser combinada con la anteriormente mencionada, *Spies* debido a que no queremos duplicar información.

Ejemplo 4. Canary Releasing

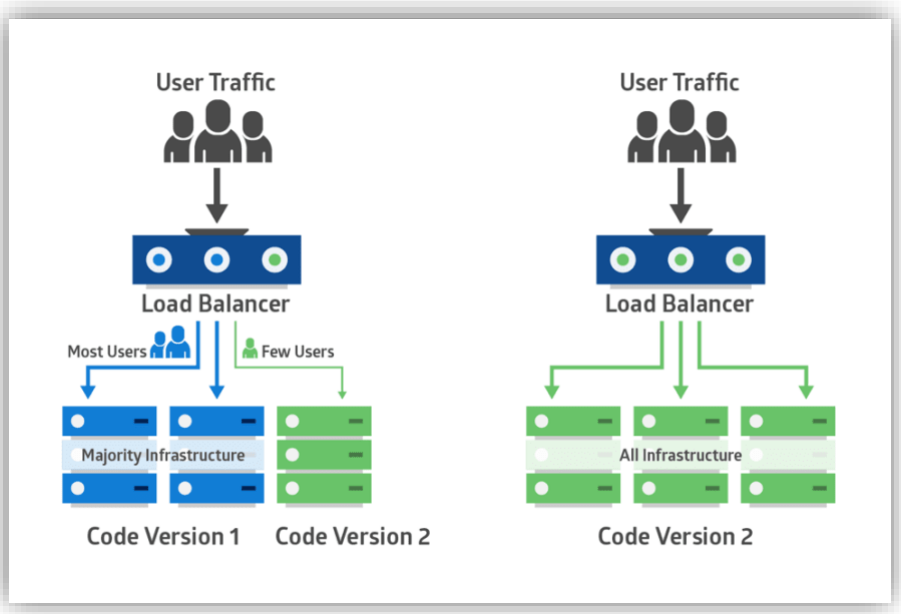

Ilustración 11 – Canary Releasing [11]

Una versión *Canary* implica dirigir un subconjunto de sus usuarios a la nueva funcionalidad y el resto, la gran mayoría, seguirían viendo la implementación anterior. La idea es que si el nuevo sistema tiene un problema, solo un pequeño grupo de usuarios se verán afectados.

Este ejemplo, lo hemos realizado con los mismos pasos utilizados en el *Strangler Fig* configurando un *NGINX* proxy como *Load Balancer* que nos permite balancear la carga utilizando diferentes pesos.

1. Todas las peticiones van dirigidas a nuestro monolito.
2. Creamos una versión 2 del monolito y desplegamos nuestro microservicio que recibirá peticiones para realizar las notificaciones al usuario.
3. Redirigimos parte de las peticiones de la versión inicial del monolito a la versión final de forma incremental, balanceando la carga a través de pesos.

En caso de que los nuevos usuarios notifiquen errores, podemos redirigir todas las peticiones al monolito inicial.

[11] INTRO TO DEPLOYMENT STRATEGIES: BLUE-GREEN, CANARY, AND MORE - https://dev.to/mostlyjason/intro-to-deployment-strategies-blue-green-canary-and-more-3a3

3.4. Decorating Collaborator

El patrón *Decorating Collaborator* permite añadir una nueva funcionalidad a un sistema sin que la aplicación subyacente sepa nada al respecto. Permite hacer que parezca que nuestro monolito está haciendo llamadas a nuestros nuevos microservicios directamente, aunque en realidad no hemos cambiado el monolito subyacente, por lo que es un patrón ideal para añadir funcionalidades cuando no podemos cambiar el código del sistema principal.

En lugar de interceptar estas llamadas antes de que lleguen al monolito, permitimos que la llamada siga adelante con normalidad. Luego, cuando la petición es respondida, según su resultado, podemos llamar a nuestros microservicios externos a través de un proxy.

Este nuevo microservicio podrá hacer uso o no de información que debe exponer el monolito. Debemos tener cuidado si necesitamos información del monolito y realizar peticiones al mismo porque puede que estemos añadiendo una dependencia circular.

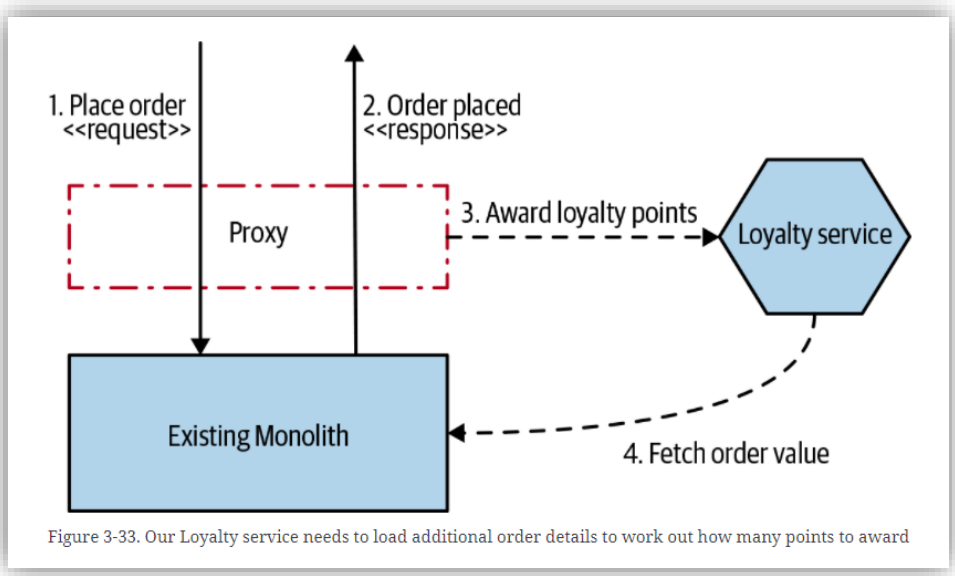

Ilustración 12 – Caso en el que necesitamos más información del monolito

Este patrón funciona muy bien cuando la información requerida se puede extraer de la propia solicitud o la respuesta generada del monolito.

Ejemplo 1. Nueva funcionalidad de fidelización

Uno de los ejemplos más extendidos de este patrón es el de un servicio de fidelización, donde queremos recompensar a nuestros clientes por la compra de nuestros productos.

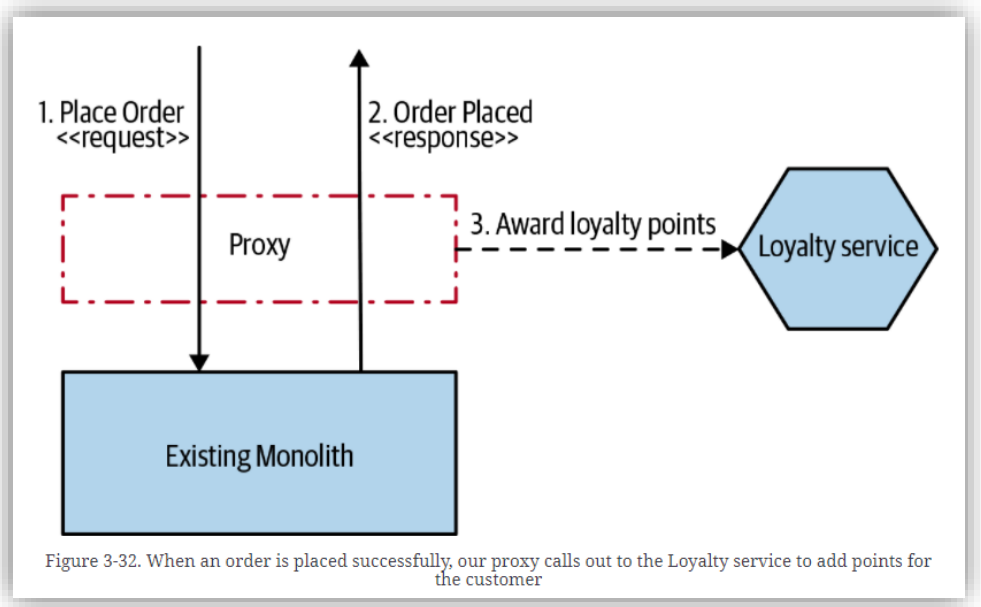

Figure 3-32. When an order is placed successfully, our proxy calls out to the Loyalty service to add points for the customer

Ilustración 13 – Estructura del patrón Decorating Collaborator

Cuando se realiza un pedido con éxito, nuestro proxy llama al servicio de Fidelización para agregar puntos para el cliente.

Uno de los grandes retos de implementar este patrón es utilizar *Spring Cloud Gateway,* puesto que está basado en *Spring WebFlux* y utiliza programación reactiva funcional.

Durante la ejecución de este patrón, hicimos una similitud con el patrón *Strangler Fig*, utilizando sus pasos para implementar la nueva funcionalidad.

Los 3 pasos serían:

1. Nuestro monolito. Disponemos de un proxy de *NGINX* que dirige todas las peticiones a nuestra aplicación monolítica.
2. Despliegue de nuestro microservicio de *Loyalty* y nuestro *Spring Cloud Gateway.*
3. Una vez probado todo y viendo que funciona correctamente, redireccionamos las peticiones del proxy de *NGINX* del paso 1, para que ahora vayan a nuestro *Spring Cloud Gateway* el cuál se encargará de direccionarlas al monolito o al microservicio. Además, en el caso de realizar una creación de una *Order*, si se realiza satisfactoriamente y la respuesta es correcta, añadiremos puntos al usuario en nuestro servicio de fidelización *Loyalty*.

En caso de error, podemos migrar las peticiones con la configuración inicial del proxy.

3.5. Change Data Capture

Este patrón, en lugar de intentar interceptar y actuar en las llamadas realizadas en el monolito, reacciona a los cambios realizados en la base de datos.

En esta ocasión hemos planteado un nuevo enunciado partiendo del problema localizado en el patrón anterior, en el que necesitábamos más información del monolito que la que enviábamos y recibíamos en las peticiones de creación y respuesta.

Nuestro monolito, al realizar una inscripción de un usuario, sólo responde que se realizó correctamente. Esto provoca que realizar el patrón anterior *Decorating Collaborator* sea difícil de aplicar, tendríamos que hacer consultas adicionales al monolito que puede que no estén expuestas a través de una API.

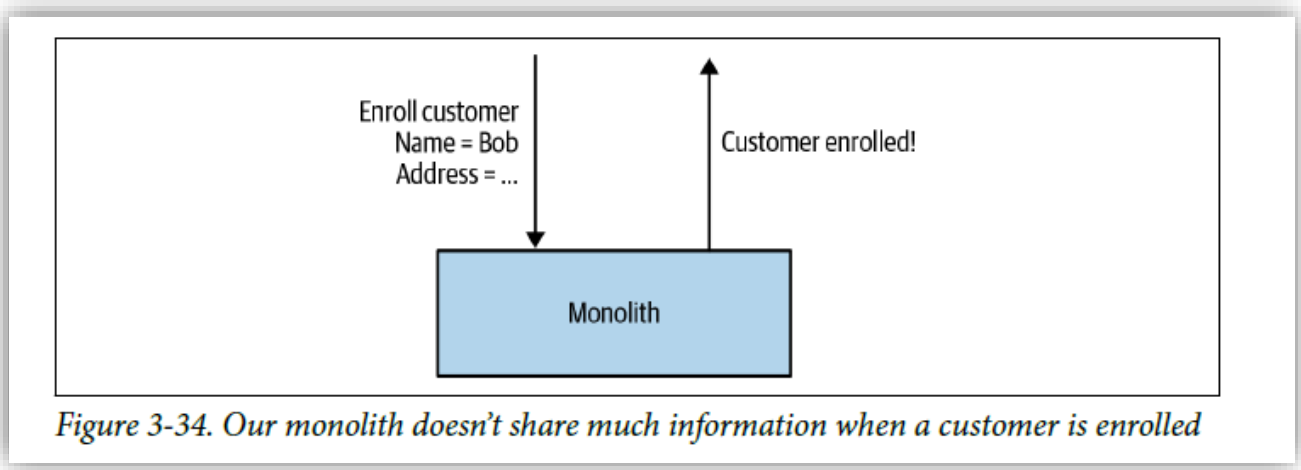

Figure 3-34. Our monolith doesn't share much information when a customer is enrolled

Ilustración 14 – El monolito devuelve poca información al responder a nuestra petición

Por lo que en este caso, utilizamos el patrón *Change Data Capture*.

Ejemplo 1. Transaction log pollers - Debezium

En el libro, *Sam Newman* menciona la existencia de un registro de transacciones en la mayoría de base de datos transaccionales. Se trata de un archivo en el que se escribe un registro de todos los cambios que se han realizado en los datos de la base de datos.

Para la aplicación de este ejemplo hemos utilizado *Debezium*, una herramienta que se basa en el registro anteriormente mencionado. Además, nos hemos apoyado de una interfaz de usuario facilitada por *Kafdrop*[12], que nos permite ver los cambios de forma mucho más intuitiva y amigable.

Uno de los principales problemas de este patrón, es que el uso de estas herramientas añade complejidad y es probable que difieran en función del tipo de base de datos, sin embargo es una buena aproximación puesto que se ejecutan desde fuera de la base de datos y disminuyen el acoplamiento y ayudan a la contención de cambios o errores.

[12] Kafdrop - https://github.com/obsidiandynamics/kafdrop

Ejemplo 2. Batch delta copier

Probablemente este sea el enfoque más sencillo. Consiste en escribir un programa que de forma periódica escanee la base de datos para ver qué datos han cambiado y se copien en el destino.

El principal problema es averiguar qué datos han cambiado realmente. El diseño del esquema puede complicar mucho esta tarea.

En nuestro caso, hemos agregado marcas de tiempo en nuestras entidades, haciendo un trabajo bastante grande en comparación con el ejemplo planteado anteriormente, en el que utilizamos un sistema de captura de datos de cambios.

4. Conclusiones y trabajos futuros

Hemos conseguido nuestro principal objetivo del proyecto, la comprensión y aplicación en código, con diferentes tecnologías, de ejemplos de los múltiples patrones planteados por *Sam Newman* en el capítulo 3, *Splitting The Monolith*, del libro *Monolith to Microservices,* valorando los pros y contras de cada uno de ellos dependiendo del contexto.

La migración de un monolito a microservicios es un proceso complicado que debe realizarse de forma incremental con pasos pequeños cuidadosamente medidos, siempre teniendo en cuenta la posibilidad de cometer errores, su impacto y la dificultad de resolverlos.

La aplicación de los patrones es dependiente de la tecnología. Durante la realización de los ejemplos hemos podido ver limitaciones y características especiales que facilitan o complican la aplicación de los patrones en función de la herramienta o tecnología utilizada.

Cabe destacar la importancia del contexto y del punto de partida. Al aplicar los patrones hemos estudiado ejemplos en los que teníamos diferentes restricciones, la modificación del código del monolito, la posibilidad de ampliar la funcionalidad de este para exponer nuevos *endpoints*, la modificación de la base de datos, etc. Es importante adaptarse a la situación inicial para elegir el patrón adecuado. No existe un patrón ideal para todos los casos.

Uno de los puntos más complicados son las migraciones en caliente, sin parada de servicio. Al estar tratando con un estudio de migración a microservicios consideramos que este apartado es imprescindible y es por eso por lo que hemos hecho tanto hincapié en su aplicación.

Tras haber conseguido los objetivos principales, consideramos que los siguientes aspectos podrían aportar mayor valor al proyecto.

- Vídeos en español. Actualmente sólo implementamos los vídeos en inglés entendiendo que sería la forma más rápida de alcanzar un público mayor.
- Documentación en inglés. Transcripción de este documento al inglés.
- Ejemplos con otras tecnologías. Como hemos podido comprobar a lo largo del trabajo, cada ejemplo es dependiente de la tecnología con la que se realiza. Aplicar otras tecnologías diferentes puede completar y abordar otro tipo de problemas que se escapan actualmente de nuestro enfoque:
    - *RabbitMQ* [13]
    - *Kubernetes* [14]
    - *MongoDB* [15]
- Actualización paulatina de las versiones de las herramientas y tecnologías utilizadas para permitir ejecutar los ejemplos con la última versión de cada una de ellas.
- Aplicar y fomentar el uso de estas técnicas dentro de nuestro ambiente laboral.

[13] RabbitMQ - https://www.rabbitmq.com/
[14] Kubernetes - https://kubernetes.io/
[15] MongoDB - https://www.mongodb.com/

5. Bibliografía

La documentación utilizada en este trabajo se encuentra sobre todo en el documento técnico situado en el repositorio de *GitHub*, https://github.com/MasterCloudApps-Projects/Monolith-to-Microservices-Examples. Cada patrón dispone de una bibliografía utilizada para la realización de los diferentes ejemplos.

Aun así, a continuación facilitamos diferentes fuentes de información para la realización de este documento concreto:

[1] MONOLITH TO MICROSERVICES. Sam Newman. Noviembre, 2018. <https://www.oreilly.com/library/view/monolith-to-microservices/9781492047834/>

[2] Kafka. Documentación oficial. Fecha sin determinar. < https://kafka.apache.org/>

[3] NGINX. Documentación oficial. Fecha sin determinar. <https://www.nginx.com>

[4] SPRING CLOUD GATEWAY. Documentación oficial. Fecha sin determinar. <https://spring.io/projects/spring-cloud-gateway>

[5] FF4J. Documentación oficial. Fecha sin determinar. <https://ff4j.org>

[6] Diffy. Documentación oficial. Fecha sin determinar. <https://github.com/opendiffy/diffy>

[7] Scientist4J. Documentación oficial. Fecha sin determinar. <https://github.com/rawls238/Scientist4J>

[8] Debezium. Documentación oficial. Fecha sin determinar. <https://debezium.io>

[9] JPA Optimistic locking. Baeldung. Marzo, 2020. <https://www.baeldung.com/jpa-optimistic-locking>

[10] Spring Batch. Documentación oficial. Fecha sin determinar. <https://spring.io/projects/spring-batch>

[11] INTRO TO DEPLOYMENT STRATEGIES: BLUE-GREEN, CANARY, AND MORE. Jason Skowronski. 21 noviembre, 2018. <https://dev.to/mostlyjason/intro-to-deployment-strategies-blue-green-canary-and-more-3a3>

[12] KAFDROP. Documentación oficial. Fecha sin determinar. <https://github.com/obsidiandynamics/kafdrop>

[13] RABBITMQ. Documentación oficial. Fecha sin determinar. <https://www.rabbitmq.com>

[14] KUBERNETES. Documentación oficial. Fecha sin determinar. <https://kubernetes.io>

[15] MONGODB. Documentación oficial. Fecha sin determinar. <https://www.mongodb.com>

[16] INTRODUCTION TO DIFERENCIA. Documentación oficial. Fecha sin determinar. <https://lordofthejars.github.io/diferencia-docs-site/diferencia/0.6.0/index.html>