Guía de Microservicios Punto de vista

Entender los microservicios

Roland Barcia – Ingeniero distinguido de IBM, CTO Microservicios/NYC Bluemix Garage
Kyle Brown – Ingeniero distinguido de IBM, CTO Arquitectura de Nube
Richard Osowski – Miembro del personal técnico sénior de IBM, Adopción de microservicios

Contenidos:

Visión general

Las aplicaciones de microservicios se componen de módulos independientes conectados en red, denominados *microservicios*. El estilo arquitectónico de microservicios es una evolución del estilo arquitectónico SOA (Arquitectura Orientada a Servicios). Las aplicaciones construidas usando servicios SOA se enfocan en problemas técnicos de integración y el nivel de los servicios implementados son a menudo interfaces específicas de programación de aplicaciones (API). Por el contrario, el enfoque de microservicios implementa capacidades comerciales claras a través de API comerciales generales.

La diferencia más grande entre los dos enfoques es cómo se despliegan. Durante muchos años, las aplicaciones han sido empaquetadas de una forma monolítica: un equipo de desarrolladores construye una aplicación grande que hace todo lo requerido para una necesidad comercial. Una vez construida, la aplicación se distribuye múltiples veces a lo largo de una colección de servidores de aplicación. En el estilo arquitectónico de microservicios, los desarrolladores construyen y empaquetan de manera independiente varias aplicaciones más pequeñas y a su vez cada una implementa partes de la aplicación.

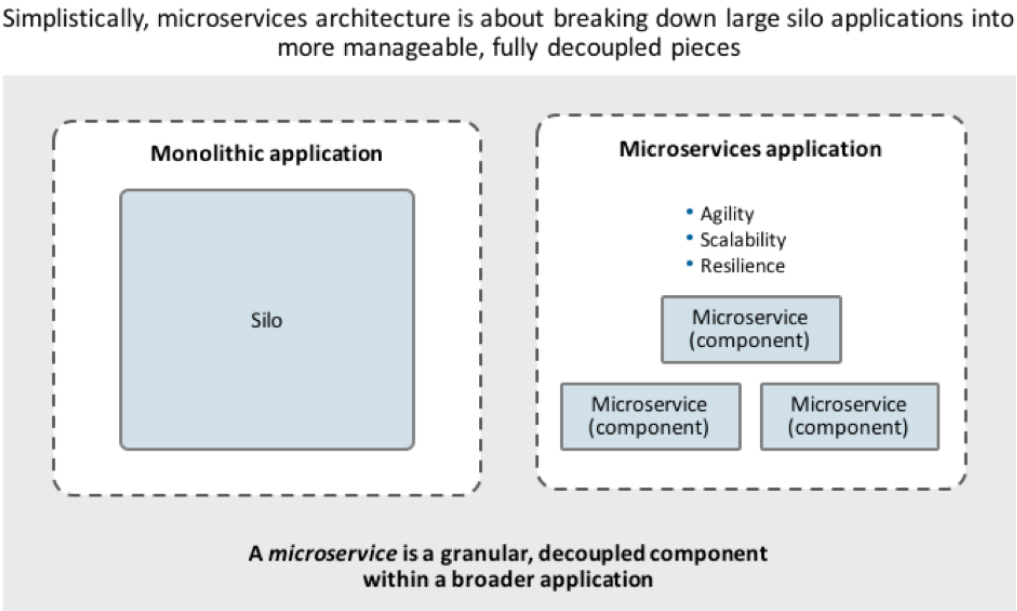

Figura 1: aplicación monolítica versus microservicios

Hay 5 reglas básicas que controlan la implementación de aplicaciones usando la arquitectura de microservicios:

1. **Dividir un sistema monolítico grande en muchos servicios pequeños** – Un solo servicio accesible por red es la unidad más pequeña utilizada para una aplicación de microservicios. Cada servicio ejecuta su propio proceso. Esta regla se denomina un servicio por contenedor. Contenedor se refiere a un contenedor Docker o cualquier otro mecanismo de despliegue ligero tales como un tiempo de ejecución de Cloud Foundry.

2. **Optimizar servicios para una sola función** – En un enfoque SOA monolítico tradicional, una aplicación en tiempo de ejecución actua sobre multiples funciones del negocio. En un enfoque de microservicios, hay solamente una función comercial por servicio. Esto hace a cada servicio más pequeño y más simple de escribir y mantener. Esto se conoce como el Principio de Responsabilidad Única (SRP).

3. **Comunicarse a través de REST API y message brokers** – Una de las desventajas del enfoque SOA es que hay numerosos estándares y opciones para implementar los servicios SOA. El enfoque de microservicios limita estrictamente los tipos de conectividad de red que un servicio puede implementar para lograr máxima simplicidad. Del mismo modo, los microservicios tienden a evitar el fuerte acoplamiento introducido por la comunicación implícita a través de una base de datos. Toda comunicación de servicio a servicio debe ser a través del servicio API o al menos debe usar un patrón de comunicación explícito tal como el patrón Claim Check [Hohpe y Woolf].

4. **Aplicar CI/CD por servicio** – En una aplicación grande compuesta de muchos servicios, diferentes servicios evolucionan a ritmos diferentes. Cada servicio tiene una interconexión continua única de integración/entrega que permite que proceda a un paso natural. Esto no es posible con el enfoque monolítico, donde cada aspecto del sistema es liberado a la fuerza a la velocidad del componente más lento del sistema.

5. **Aplicar alta disponibilidad por servicio (HA)/decisiones sobre clústeres** – Cuando se construyen sistemas grandes, la agrupación en clústeres no es un enfoque del tipo en un tamaño entra todo. El enfoque monolítico de escalar todos los servicios al mismo nivel lleva a la sobrecarga de algunos servidores y al desperdicio de otros; o aún peor, la inanición de algunos servicios por otros que monopolizan todos los recursos compartidos disponibles, tales como agrupaciones de subprocesos. La realidad es que, en un sistema grande, no se necesita escalar todos los servicios; muchos servicios pueden ser desplegados en un número mínimo de servidores para conservar recursos. Otros requieren ampliarse a números muy grandes.

La combinación de estas 5 reglas y sus beneficios son las principales razones por las que la arquitectura de microservicios se ha vuelto tan popular.

Visión general

La arquitectura de microservicios se ha vuelto un estándar de hecho para desarrollar aplicaciones comerciales a gran escala. En Microservicios: una definición de este término de arquitectura, Martin Fowler define microservicios:

"En pocas palabras, el estilo arquitectónico de microservicios es un enfoque para el desarrollo de una sola aplicación como un conjunto de servicios pequeños, cada uno ejecutándose en su propio proceso y comunicándose con mecanismos ligeros, a menudo una API de un recurso HTTP. Estos servicios son construidos alrededor de capacidades comerciales y son desplegables independientemente por maquinaria de despliegue completamente automatizada. Hay un mínimo indispensable de gestión centralizada de estos servicios, que puede estar escrito en distintos lenguajes de programación y usar diferentes tecnologías de almacenamiento de datos."

Una de las diferencias clave entre microservicios y paradigmas como SOA y API es el foco en los componentes desplegados y en ejecución. Los microservicios se enfocan en el nivel de detalle de los componentes desplegados más que en las interfaces, como se muestra en la figura de abajo.

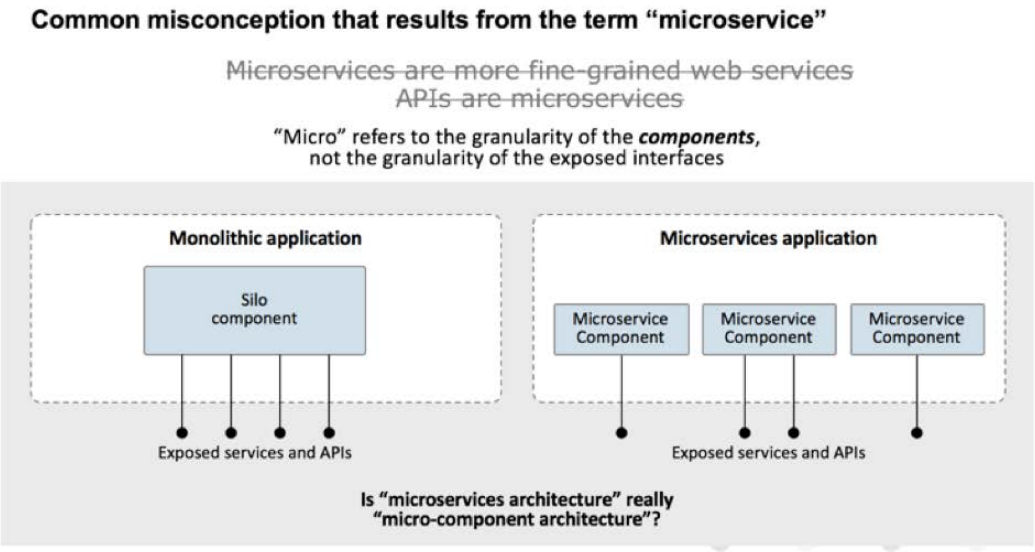

Figura 2: diferencia entre una API y un microservicio

Visión general

Muchas organizaciones, controladas por operaciones, construyen aplicaciones monolíticas donde casos y funciones de uso son desplegados en una sola gran aplicación. Si bien existe una cierta simpleza alrededor de las operaciones, cambiar estas aplicaciones para basarse en microservicios requiere un gran esfuerzo. Tres factores controlan el desarrollo de microservicios:

1. **Cómo se organizan los equipos de desarrollo:** El desarrollo de microservicios se realiza mejor con un enfoque de ingeniería que se enfoca en descomponer una aplicación en módulos de funciones únicas con interfaces bien definidas, que son desplegadas independientemente y operadas por equipos pequeños que poseen todo el ciclo de vida del servicio. Los microservicios aceleran la entrega minimizando la comunicación y coordinación entre personas mientras que reducen el alcance y riesgo de cambio.

2. **Cómo se construyen las aplicaciones:** Las aplicaciones basadas en microservicios hacen algunas suposiciones sobre la manera en que están construidas y el entorno en el cual se ejecutan. El entorno a menudo se menciona como *aplicaciones nativas de nube* o *aplicaciones de 12 factores*. Una arquitectura basada en microservicios aprovecha las fortalezas y acomoda los desafíos de un entorno de nube estandarizado, incluyendo conceptos tales como escalamiento elástico, despliegue inmutable, instancias descartables e infraestructura menos predecible.

3. **Cómo se entregan y ejecutan las aplicaciones:**
El uso de contenedores como una forma estándar de ejecutar aplicaciones controla el empaquetado y la ejecución de aplicaciones basadas en microservicios. Los contenedores no son una tecnología nueva. Los contenedores Linux son una capacidad de nivel de sistema operativo que posibilita ejecutar múltiples sistemas Linux aislados (o contenedores) en un host de control Linux. Los contenedores Linux sirven como una alternativa ligera a máquinas virtuales completas. Aun cuando los contenedores no son un concepto nuevo, las infraestructuras como Docker han popularizado su uso al diseñar una manera de crear una imagen para contenedores en ejecución. Esto le da a usted una manera estándar de empaquetar una aplicación y todas sus dependencias de modo que pueda ser movida entre entornos y ejecutada sin cambios. Otras infraestructuras como Cloud Foundry usan contenedores para ejecutar aplicaciones, pero abstraen la virtualización.

La figura debajo muestra como la arquitectura de aplicación monolítica evoluciona hacia una basada en microservicios.

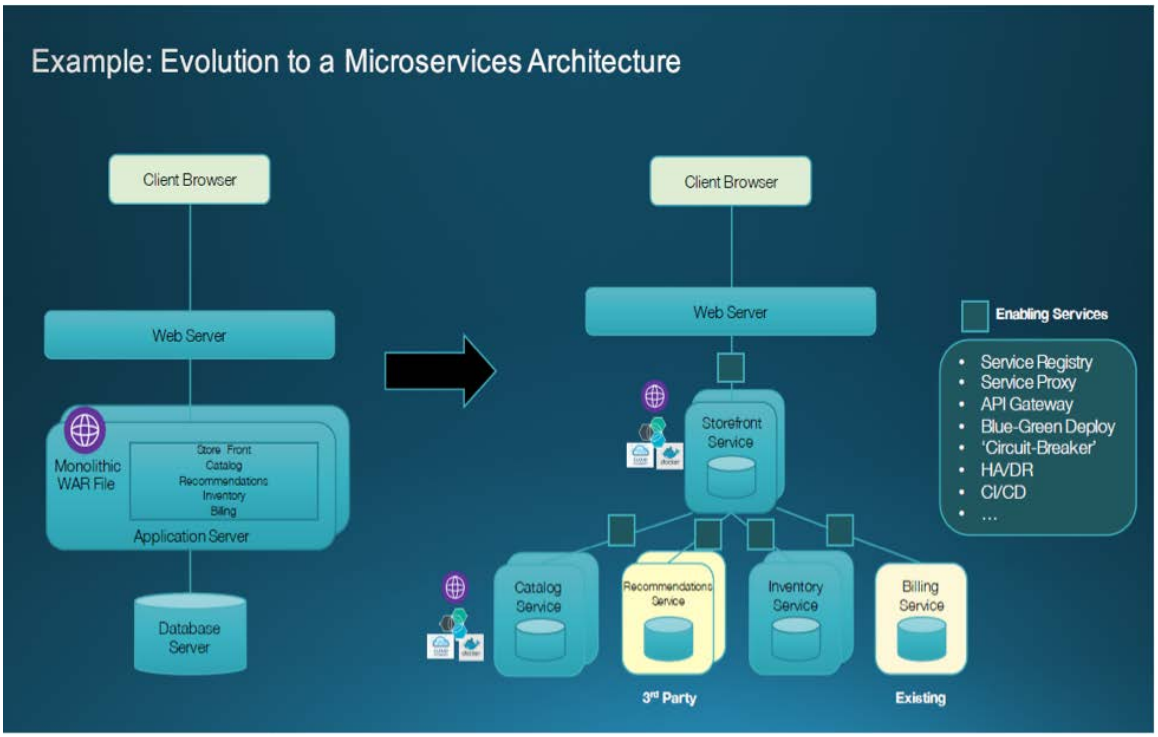

Figura 3: diferencia entre aplicaciones monolíticas y microservicios

La figura 3 muestra un único archivo empresarial que aloja a todos los componentes de un sitio web de venta minorista. El archivo de aplicación contiene todas las funciones comerciales, tales como catálogo e inventario, así como lógica de sitio web, lógica comercial y lógica de persistencia para cada componente. Además, la aplicación comparte una sola base de datos, la cual a menudo contiene modelos de datos estrechamente acoplados y es compartida con otras aplicaciones.

Observe en el costado derecho de la imagen que las capacidades comerciales ahora están desplegadas en aplicaciones separadas con sus propios datos. Este nuevo estilo de arquitectura genera preocupaciones relacionadas a la comunicación de microservicios, propiedad de datos, sincronización de datos y resiliencia.

El capítulo Características de una arquitectura de microservicios por James Lewis y Martin Fowler discute aspectos claves de las aplicaciones basadas en microservicios.

Usando el ejemplo de referencia de arriba, un resumen de los aspectos incluye:

Agrupación de componentes a través de servicios: Este aspecto está cubierto anteriormente en este trabajo.

Organizado alrededor de capacidades comerciales: Anteriormente discutimos la noción de desarrollo de equipo y organizar alrededor de capacidades comerciales requiere cambios con respecto a cómo pensamos sobre los equipos de desarrollo.

Considere cómo a menudo los equipos se organizan por roles como se muestra abajo:

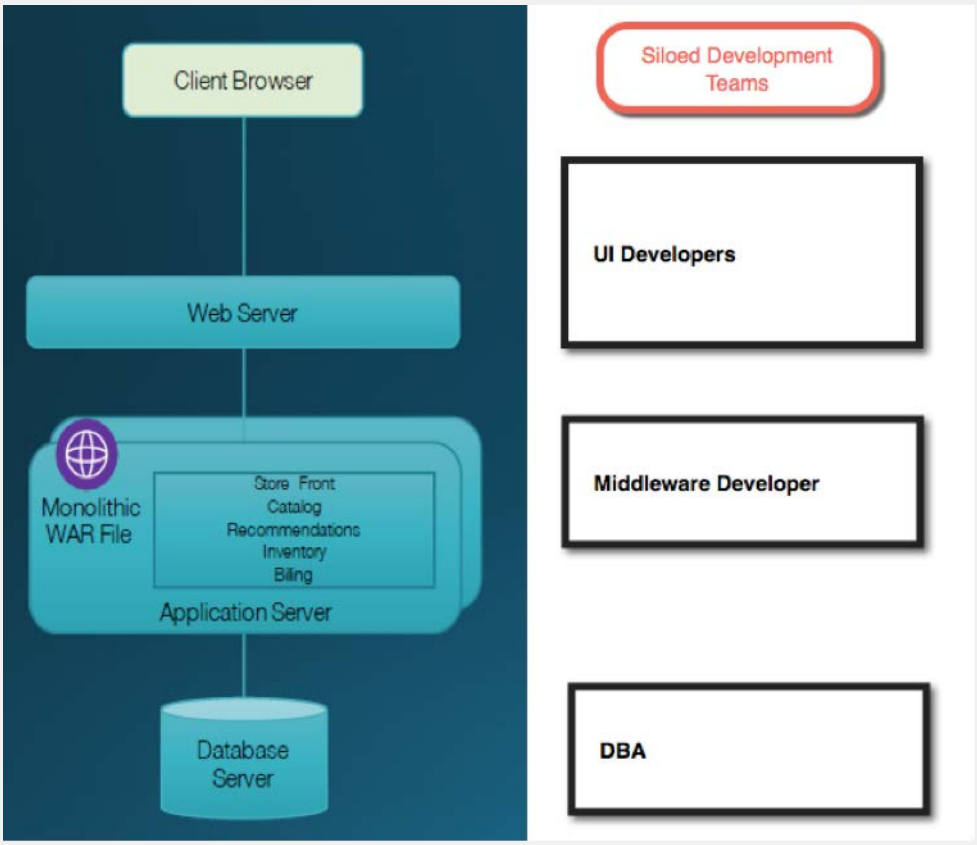

Figura 4: equipos de desarrollo en silo

Compare eso con un enfoque de equipo basado en microservicios, que se organiza alrededor de las capacidades comerciales, como muestra la figura 5.

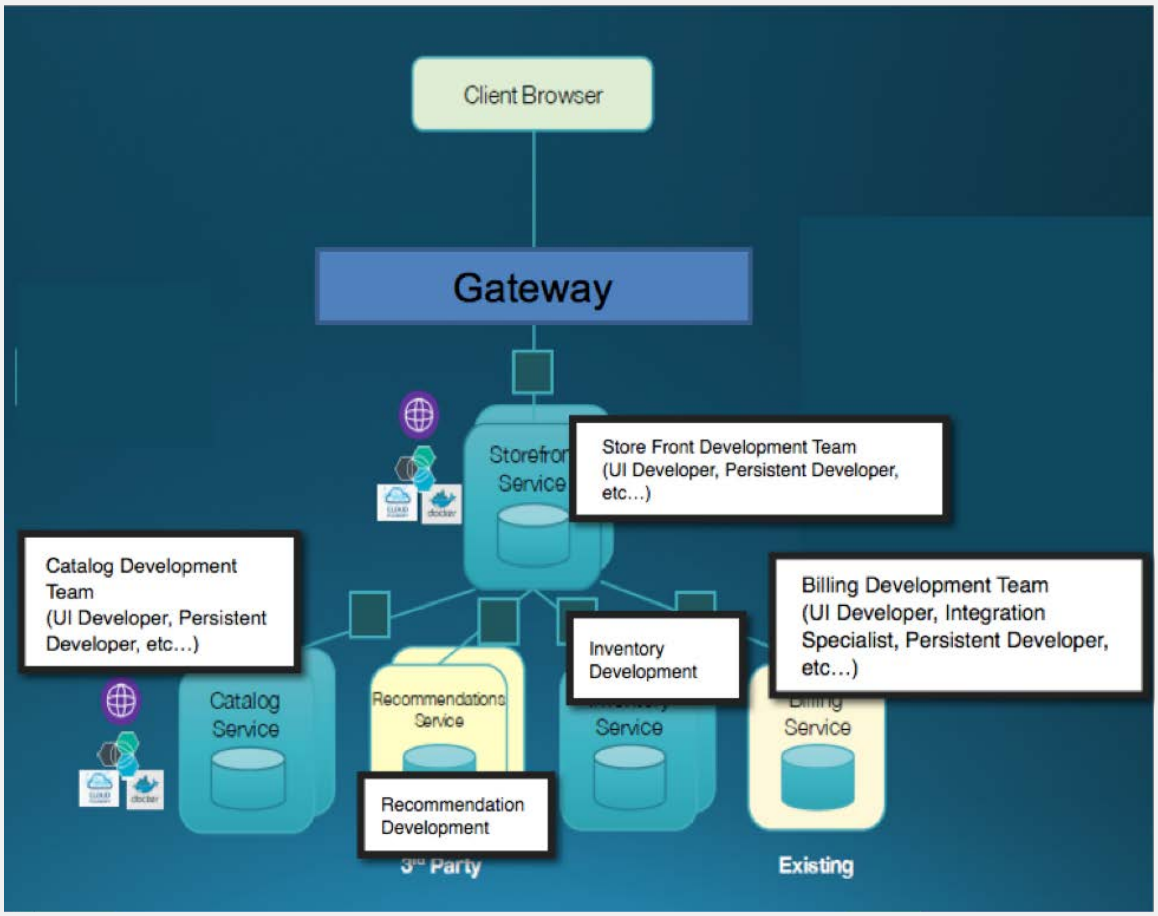

Figura 5: equipos de desarrollo basado en capacidades comerciales

Solución de problemas comerciales mediante microservicios

La mayoría de las compañías construyen aplicaciones nativas de nube nuevas usando un enfoque basado en microservicios. Sin embargo, muchas compañías también están viendo la necesidad de refactorizar las aplicaciones monolíticas existentes para moverse más rápido.

Migrar aplicaciones monolíticas

La figura 6 muestra un ejemplo de migración desde una arquitectura monolítica a una arquitectura basada en microservicios. En este caso, un minorista quiere avanzar hacia microservicios para moverse más rápido, aprender más sobre el cliente e introducir funciones modernas como pagos.

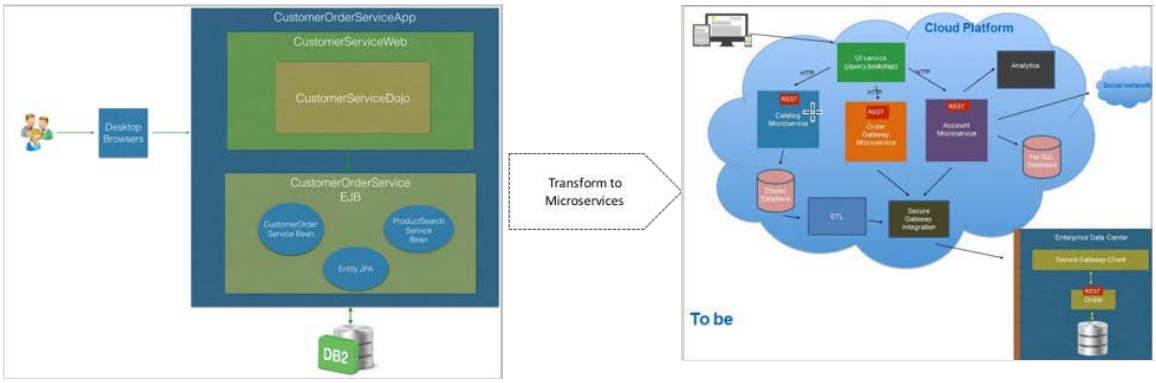

Figura 6: Migrar hacia una arquitectura basada en microservicios

El minorista en la situación de arriba siguió estas etapas para mudarse de una arquitectura monolítica a una arquitectura de microservicios:

1. **Determinar cómo manejar el catálogo.** El primer problema comercial que tuvieron que resolver fue cómo gestionar el catálogo de ítems. Los clientes no podían encontrar los datos de productos y el minorista no podía exponer cosas para otros sitios. El equipo decidió construir un solo microservicio para el catálogo comercial usando los siguientes pasos:

    - Importó datos a una búsqueda elástica para proveer nuevas maneras de buscar datos e identificar nuevos patrones de datos.
    - Enlazó su sitio web existente con la nueva búsqueda.
    - Con este piloto, introdujeron un nuevo modelo CI/CD como base.

El equipo convenció a la gerencia de migrar a un modelo de microservicios y expandir el negocio.

2. **Aprender más sobre el cliente** - Para aprender más sobre el cliente, el equipo creó un microservicio de cuenta:

    - Primero, descubrieron cómo cambiar el negocio de una organización enfocada en el consumidor a una enfocada en el inventario. Diseñaron un nuevo modelo de cliente y usaron una base de datos NOSQL como Mongo DB o Cloudant, que proveyó un modelo de datos no estructurados. Sabían que, con el tiempo, los datos de clientes serían enriquecidos con base en datos de análisis, marketing y cognitivos.
    - Los datos existentes de clientes se usaron para rastrear órdenes.

3. **Crear una nueva experiencia de usuario** - El equipo creó un nuevo front-end de usuario para móviles y plataformas web y una nueva aplicación nativa para móviles. Con una interfaz de usuario y catálogo modernos, crearon una nueva experiencia para el usuario final. El nuevo catálogo fue integrado con la lógica de orden existente, el cual comprendía el negocio núcleo y todavía era demasiado complejo para dividir.

4. **Microservicio de órdenes** - El equipo se enfocó en crear nuevas API de órdenes para móvil, además de integrar con las transacciones existentes. El negocio decidió crear un microservicio adaptador que llamaba al sistema de registro existente (SOR). Aprovecharon la oportunidad de integrar con los nuevos pagos modernos en esta capa del adaptador.

5. **Expandir el negocio** - El minorista expandió el modelo de negocio a añadir una nueva opción de subasta, ampliando el nuevo modelo de microservicio.

Arquitectura de microservicios

Aprenda los detalles de una arquitectura basada en microservicios.

Capacidad de los microservicios

Los componentes de aplicación se ejecutan como microservicios en la nube, comunicándose unos con otros a través de la estructura de microservicios. Diferentes tipos de aplicaciones usan diferentes patrones. Mostramos un ejemplo web/móvil más adelante. La figura 7 muestra las capacidades necesarias para una arquitectura basada en microservicios.

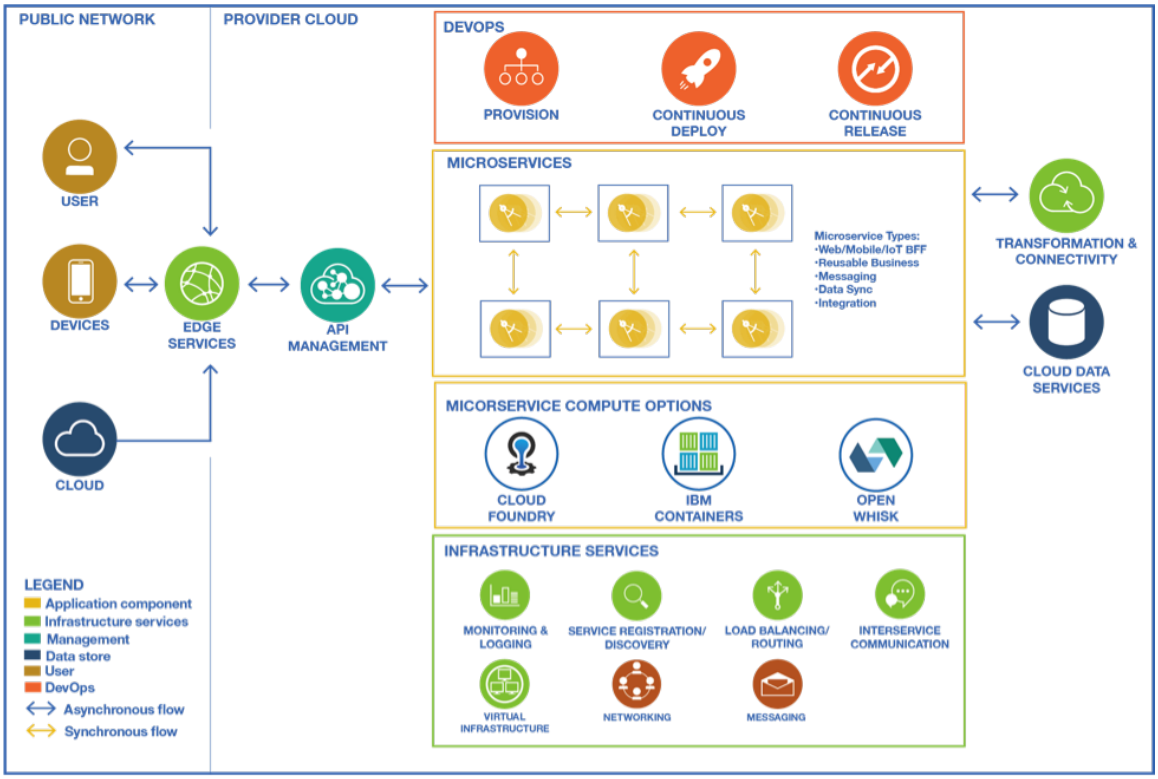

Figura 7. Capacidades de una arquitectura basada en microservicios

Con una lectura de izquierda a derecha en el diagrama, el próximo paso es revisar las capacidades de los microservicios. Muchos microservicios están expuestos a través de las API y algunos de estos microservicios necesitan ser consumidos a través de una *Puerta de enlace de API*.

Una Puerta de enlace de API puede ser tan simple como un proxy de terminales. Algunas Puertas de enlace de API son más sofisticadas, con seguridad al primer contacto, a través del monitoreo y versionado de la API, o sistemas de gestión total de API con un portal desarrollador para consumo de terceras partes.

DevOps

Usted debe desarrollar una estrategia de automatización fuerte mediante el uso de DevOps para crear una arquitectura de microservicios exitosa, la cual incluya suministro, despliegue continuo y liberación continua. Su estrategia deberá incluir lo siguiente:

- **Suministro -** Una arquitectura de microservicios exitosa requiere el suministro automatizado para entornos de aplicación. La Plataforma como una capa de servicio usualmente provee este servicio a través de un servicio gestionado, tal como Contenedor como servicio (Caas) en IBM® Bluemix®. Si ejecuta una infraestructura de contenedor sobre una capa de Infraestructura como servicio (IaaS) mediante máquinas de orquestación Docker como Kubernetes o Docker Datacenter (DDC), usted debe automatizar cómo estos entornos son suministrados y actualizados utilizando suministro automatizado en un entorno de máquina virtual.

- **Despliegue continuo -** Se requiere un proceso automatizado de construcción y despliegue para imágenes Docker o aplicaciones de microservicios en el entorno de desarrollo. Si usted tiene una estrategia multinube, su automatización de construcción y despliegue necesita abstraer las diferencias en cómo usted realiza un autoescalado o políticas de nube.

- **Liberación continua -** Un entorno DevOps soporta una cultura fuerte de desarrollo controlado por pruebas y pruebas automatizadas. Esto incluye la prueba de unidades, prueba de validación de entorno de prueba funcional y de rendimiento automatizada.

Opciones de cálculo de microservicios

Hay varias opciones de cálculo para microservicios en ejecución:

- **Contenedores Docker -** Estos contenedores proveen la mayoría de la portabilidad a través de entornos de nube y locales. Utilizar contenedores Docker requiere DevOps fuertes.

- **Cloud Foundry -** Esta PaaS de código abierto provee abstracción para cómo se ejecutan los microservicios y se comunican unos con otros y para virtualización tal como contenedores. Si bien Cloud Foundry ofrece algún nivel de portabilidad, requiere una pila completa en el entorno de nube para ejecutarse.

- **OpenWhisk -** Esta nube de código abierto basada en eventos ofrece el nivel más alto de abstracción y facilidad de uso. La nube emite eventos y los desarrolladores despliegan administradores simples de eventos para responder a esos eventos. Toda la virtualización es abstraída.

- **Máquinas virtuales o nativas -** Usted puede crear aplicaciones basadas en microservicios y ejecutarlas en máquinas virtuales (VM). Esto requiere mucho más suministro y DevOps para tener éxito. Las VM y nativas ofrecen mayor flexibilidad a expensas de la facilidad.

La figura 8 resume estas opciones:

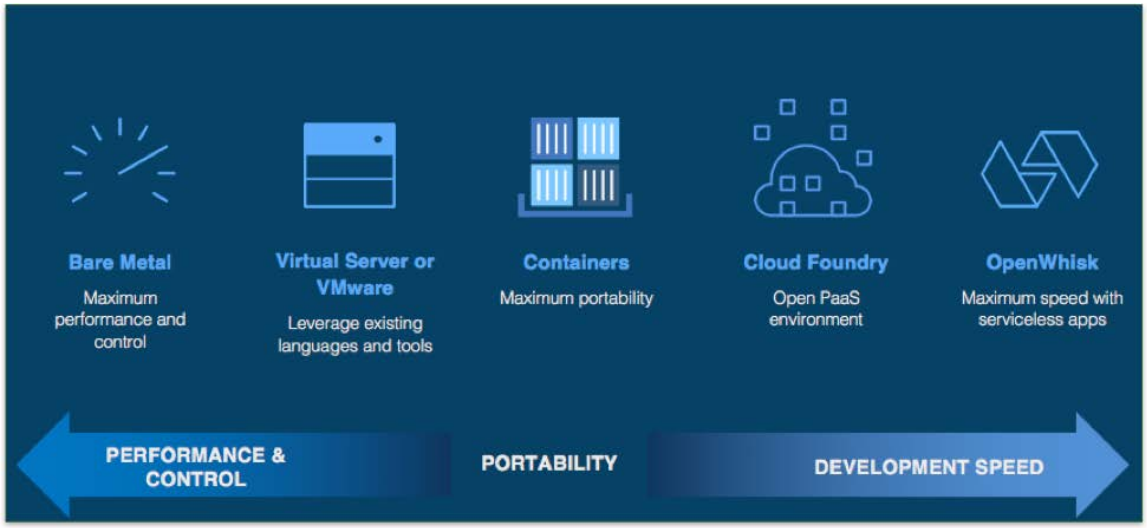

Figura 8. Opciones de cálculo de microservicios

Servicios de infraestructura

El entorno de nube que rodea a los microservicios provee la estructura y servicios necesarios para la conexión en red, mensajería, comunicación de microservicio, registro y monitoreo, virtualización, descubrimiento y proxy de servicios, funciones de resiliencia y más.

Arquitectura de aplicaciones

Debajo hay un ejemplo de una arquitectura de aplicación de microservicios.

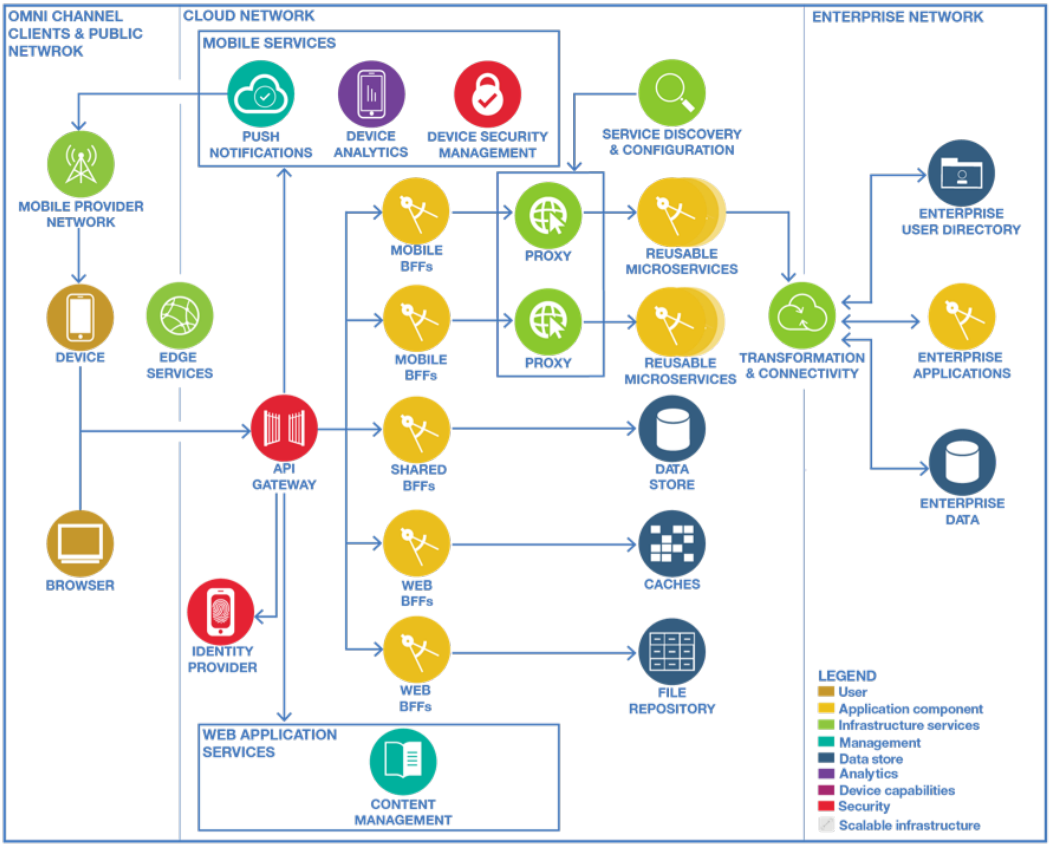

Figura 9. Arquitectura de aplicación de microservicios

Para resumir los componentes de esta arquitectura:

- Las aplicaciones OmniChannel en este ejemplo contienen tanto una aplicación iOS nativa como una aplicación web basada en Angular. El diagrama las representa como un dispositivo y un navegador.

- Las aplicaciones para móviles utilizan el servicio IBM Mobile Analytics for Bluemix para recolectar el análisis de dispositivos para operaciones y negocios.

Arquitectura de microservicios

- Ambas aplicaciones de clientes realizan llamadas API a través de una Puerta de enlace de API. La Puerta de enlace de API, IBM API Connect, provee un Proveedor OAuth para implementar seguridad de API.

- Las API se implementan como patrones de microservicios Node.js que se mencionan como Backend for Frontends (BFF). Otros nombres posibles incluyen Experience API (xAPI) o Iteration API. En esta capa, los desarrolladores front-end usualmente escriben una lógica back-end para sus front-end. El BFF de Inventario se implementa mediante el uso de la infraestructura Express. El BFF de Revisión Social se implementa mediante el uso de la infraestructura Connect LoopBack de API. Estos microservicios se ejecutan en Bluemix como aplicaciones de Cloud Foundry.

Leer un caso de estudio

Obtener detalles técnicos

- Los BFF de Node.JS invocan otra capa de microservicios Java reutilizables. En un proyecto del mundo real, un equipo diferente escribirá esto generalmente. Estos microservicios reutilizables están escritos en Java mediante el uso de Spring Boot. Se ejecutan dentro de contenedores de IBM mediante el uso de Docker.

- Los BFF de nodo y microservicios de Java se comunican unos con otros usando una estructura de microservicios. Ejemplos incluyen Amalgam8 y Netflix OSS.

- Los microservicios Java pueden interactuar con bases de datos en la nube y capas de integración.

Pila de microservicios

Es importante definir su pila de microservicios. Debajo hay un ejemplo de una pila con algunas elecciones ya hechas. Usted necesita definir las diferentes capas, incluyendo la aplicación, estructura y DevOps. Refiérase a la Guía de decisiones de microservicios de IBM para ver cómo IBM determinó estas tecnologías de habilitación de microservicios.

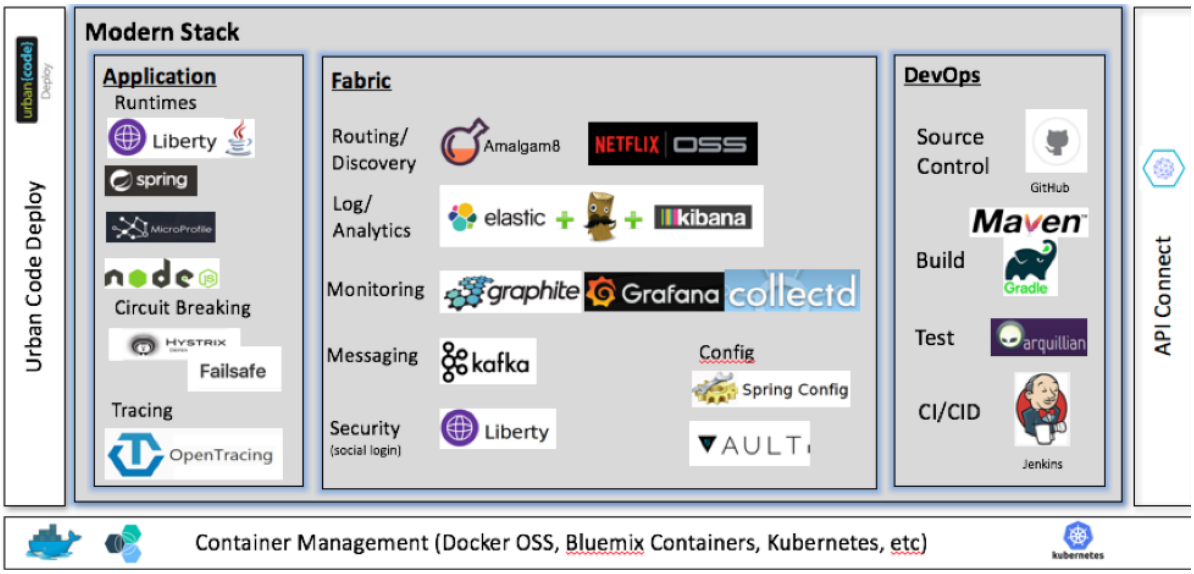

Figura 10. Pila de microservicios

Aplicación

Un lenguaje y una pila de tiempo de ejecución escriben su aplicación. Ejemplos incluyen pilas modernas ligeras de Java Platform, Enterprise Edition (Java EE) como MicroProfile, WebSphere Liberty, pilas alternativas de Java como Spring, o pilas Node.jd como StrongLoop.

Se espera que bibliotecas de lenguaje específicas manejen elementos de aplicaciones de microservicios tales como interrupción y trazado de circuito.

Estructura de microservicios

Una estructura de microservicios es una pila de capacidades que provee un conjunto de capacidades necesarias:

- El enrutado y descubrimiento es una capacidad central para una comunicación interna de microservicios basada en la nube. Una instancia de microservicios podría ser autoescalada y tener clústeres dinámicos, por ejemplo, usted debe descubrir el microservicio por nombre y luego enrutarlo a una instancia en ejecución. Infraestructuras como Zuul/Eureka proveen esto como parte de la pila Netflix OSS. Amalgam8 es otra opción que está basada en poliglotía. La figura de abajo muestra un ejemplo de la arquitectura Amalgam8.

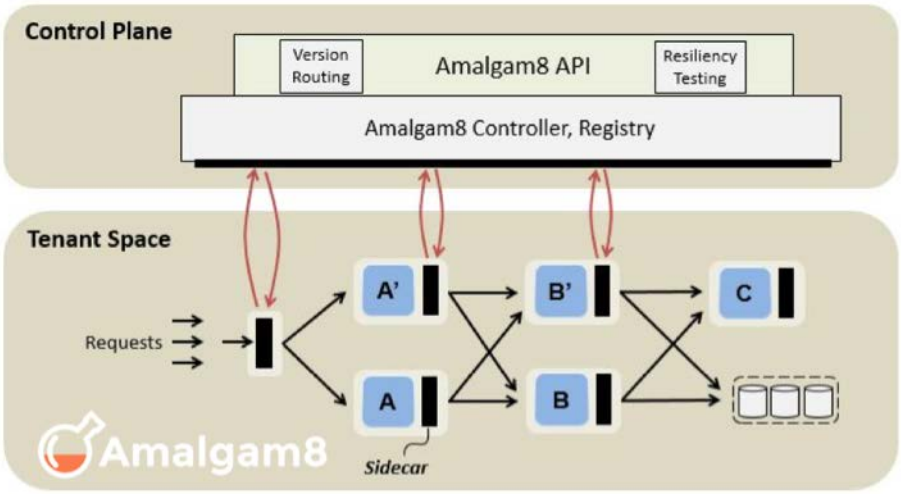

Figura 11. Arquitectura de Amalgam8

- Para registro y análisis, se requiere una pila que captura registros de transmisiones y los reenvía a varios lugares. Hay varias infraestructuras disponibles para eso.

- Se requieren paneles de instrumentos de monitoreo para datos variados, incluyendo tiempo de ejecución, información de aplicaciones y de interruptores de circuitos. Mediante el uso de la pila de registro, estas herramientas proveen una visión unificada.

- Las capas de mensajería y seguridad también son importantes.

DevOps

Usted debe tener un conjunto fuerte de herramientas de DevOps para completar requisitos para el suministro, orquestación, construcción y despliegue y operaciones.

Gestión de contenedores

Se necesita una organización basada en contenedores y un entorno en tiempo de ejecución como IBM Bluemix Container Service, Kubernetes o Docker Data Center para soportar una pila de microservicios.

Con tales componentes específicos, usted debe ser consciente de todos los puntos de resiliencia en una arquitectura de microservicios. Esto incluye alta disponibilidad, recuperación tras fallo, DR, interrupción de circuito y aislamiento.

La figura 12 ilustra la resiliencia en arquitecturas basadas en microservicios y muestra el uso de un equilibrador de carga global y redundancia multisitio.

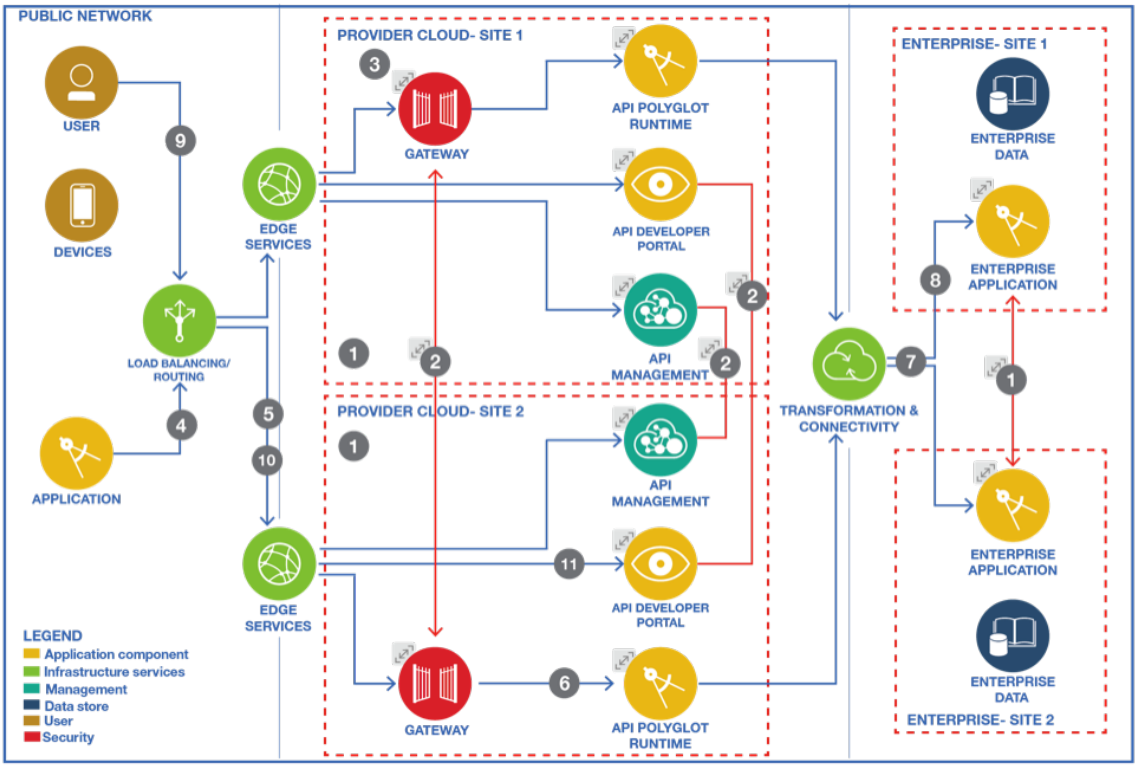

Figura 12. Resiliencia en una arquitectura de microservicios

Alta disponibilidad y patrones de recuperación de desastres para microservicios

Consideraciones de resiliencia:

- Back-ends de datos redundantes
- Back-ends de datos replicados
- Desplegar múltiples veces por región
- Desplegar a múltiples regiones (tres centros de datos)
- Equilibrio de carga global

Al tratar con resiliencia, es importante hacer algunas distinciones entre alta disponibilidad (HA) y recuperación de desastres (DR).

HA asegura que los servicios están disponibles para los usuarios finales cuando las actividades de mantenimiento, como el despliegue de actualizaciones, reinicio de las máquinas virtuales de alojamiento y la aplicación de parches de seguridad al SO de alojamiento se realizan en el sistema.

HA generalmente no se refiere a problemas mayores no planificados como la pérdida completa del sitio debido a cortes mayores de energía, terremotos, fallas severas de hardware o pérdida de conectividad de todo el sitio. En tales casos, si los servicios tienen objetivos estrictos de nivel de servicio (SLO), usted deberá hacer redundante a toda la pila de aplicaciones (infraestructura, servicios y componentes de aplicaciones) dependiendo de al menos dos regiones Bluemix diferentes. Esto se define típicamente como una arquitectura de Recuperación de Desastres (DR).

Hay muchas opciones para implementar soluciones de DR. Por una cuestión de simplicidad, las diferentes opciones pueden agruparse en tres categorías mayores:

Activa/Pasiva, Activa/En espera, y **Activa/Activa.**

Activa/Pasiva	Activa/En espera	Activa/Activa
Esta opción mantiene toda la pila de aplicaciones activa en un lugar, mientras que otra pila de aplicaciones es desplegada en una ubicación distinta pero mantenida inactiva o apagada. En caso de indisponibilidad prolongada del sitio primario, la pila de aplicaciones es activada en el sitio de respaldo. Generalmente eso requiere restaurar copias de seguridad tomadas en el sitio primario. Este enfoque no se recomienda si la pérdida de datos es un problema o la disponibilidad del servicio es crítica a causa de que el objetivo de tiempo de recuperación (RTO) es menor a unas pocas horas.	Con esta opción, toda la pila de aplicaciones esta activa tanto en las ubicaciones primarias como de respaldo; sin embargo, solo el sitio primario sirve a las transacciones de usuarios. El sitio de respaldo almacena una réplica del estado de la ubicación principal a través de la replicación de datos como la replicación de base de datos o replicación de disco. En caso de indisponibilidad prolongada del sitio primario, todas las transacciones de clientes son enrutadas al sitio de respaldo. Este enfoque provee un buen objetivo del punto de recuperación (RPO) y RTO (minutos), pero es significativamente más caro que Activa/Pasiva debido al doble despliegue. Hay un desperdicio de recursos a causa de que los activos en espera no pueden usarse para mejorar la escalabilidad y rendimiento.	En este caso, ambas ubicaciones están activas y las transacciones de clientes se distribuyen a ambas regiones según políticas predefinidas como round-robin, equilibrio de carga y ubicación. En caso de que un sitio falle, el otro sitio sirve a todos los clientes. Es posible lograr RPO y RTO cercanos a cero con esta configuración. La desventaja: ambas regiones deben ser dimensionadas para manejar la carga completa, aun cuando sean usadas a la mitad de sus capacidades cuando ambas ubicaciones estén disponibles. En ese caso, el Autoescalado para servicio Bluemix asigna recursos de acuerdo con las necesidades, similar a la aplicación de muestra BlueCompute.

Consideraciones de escalabilidad y rendimiento

Agregar resiliencia generalmente implica tener despliegues redundantes, que también pueden usarse para mejorar el rendimiento y la escalabilidad. Esto es así para el caso Activa/Activa, descripto en la sección de arriba. En caso de aplicaciones globales, es posible redirigir las transacciones de los usuarios a la ubicación más cercana para mejorar el tiempo de respuesta y latencia usando soluciones de enrutado global de Akamai o Dyn.

Implementación de proyectos basados en microservicios

El próximo paso lógico es conocer cómo implementar proyectos basados en microservicios en su organización.

Usted puede construir un sistema de microservicios desde el principio o desde un sistema monolítico existente. La mayoría de los clientes de IBM comienzan su viaje de microservicios buscando actualizar sus monolitos existentes, por lo que esta sección se enfoca en cómo evaluar e implementar microservicios cuando se trabaja a partir de una arquitectura monolítica existente.

Si bien construir proyectos de microservicios sin una aplicación previa es relevante o importante, utilice un enfoque de un monolito previo al construir microservicios. En pocas palabras, esto significa que usted construye su aplicación de cualquier modo que pueda validar primero su idea.

Luego, usted aplica los principios descritos en este white paper para escalar y evolucionar su monolito inicial en un proyecto de microservicios. No tiene utilidad implementar una arquitectura de microservicios si no ofrece un valor al negocio.

Una excelente explicación de primera mano de la implementación de una aplicación de microservicios utilizando el enfoque de un monolito previo es documentada por el equipo Gamo On! en The Chronicles.

Evolución de las aplicaciones existentes

A esta altura usted sabe que hay muchos conceptos que necesita entender al pasar por una transformación basada en microservicios. En esta sección discutimos tres áreas que usted necesita entender para implementar un proyecto de microservicios exitoso: su negocio, su cultura y conjunto de habilidades y su tecnología.

Entender y definir sus necesidades comerciales

¿Por qué está pensando en mudarse a microservicios? Para muchos negocios, se requiere un desarrollo de software y prácticas operativas más eficientes para entregar valor al negocio más rápidamente, y entregar una mejor experiencia de usuario.

Antes de que usted pueda entender el impacto de un proyecto de microservicios en sus aplicaciones e infraestructura existentes, debe entender qué partes de su negocio se están moviendo demasiado lento para producir resultados satisfactorios. En muchos casos, los sistemas de compromiso (SOE) de la organización están causando la lentitud. Estos sistemas están disponibles a través de muchos canales, incluyendo la web, móviles y las API. La falta de velocidad es la razón primaria para mudarse a arquitecturas basadas en microservicios.

Antes de que usted pueda adoptar un enfoque orientado a microservicios, debe saber qué es lo que no está llegando al mercado lo suficientemente rápido. Primero necesita identificar qué partes de la aplicación necesitan mejoras y modificaciones para hacerlas más rápidas. A partir de allí usted puede identificar qué partes del monolito existente deberá migrarse a microservicio.

Los artefactos de diseño y arquitectura, tales como flujos de experiencia de usuario o diagramas de arquitectura, son de uso valioso en este paso. El equipo puede identificar rápidamente y priorizar secciones del monolito, como se muestra en la figura 13.

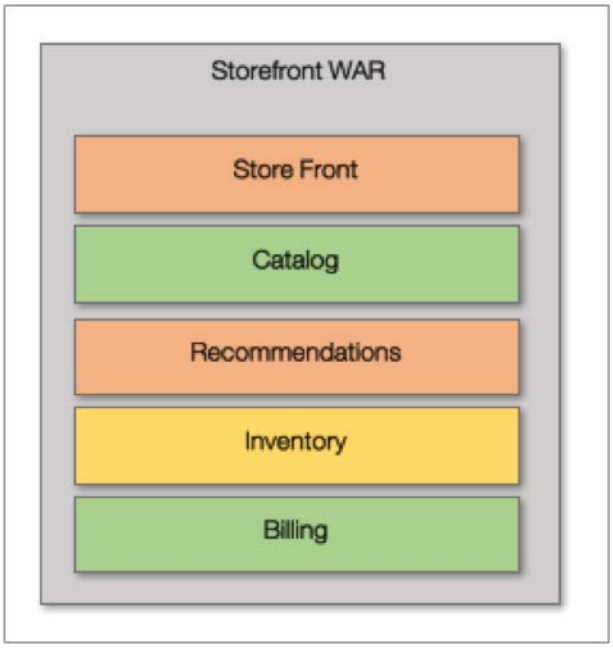

Figura 13. Puntos problemáticos existentes en el monolito, mediante el uso de la escala Rojo-Amarillo-Verde para la prioridad

Este proceso de identificación no necesita ser exhaustivo y deberá ser reiterativo en su naturaleza. Los objetivos clave son:

- Identificar las funciones comerciales separadas que su monolito está suministrando.
- Entender la velocidad y complejidad relativas requeridas para cambiar esas funciones comerciales.
- Entender el deseo del negocio de ver ciclos de retroalimentación más rápidos para funciones comerciales específicas.

Entender su cultura y conjunto de habilidades

Si bien no es específico de arquitecturas basadas en microservicios, una comprensión general de los equipos de una organización, la cultura y el conjunto de habilidades son fundamentales durante la transformación digital.

Típicamente, en un enfoque monolítico, la mayoría de las organizaciones están construidas en silos, con una participación según se necesite, a lo largo del ciclo de vida del desarrollo del software. Esto a menudo genera límites bien definidos, con roles y responsabilidades restringidas debido a estos límites. La figura 14 muestra una estructura organizativa monolítica típica.

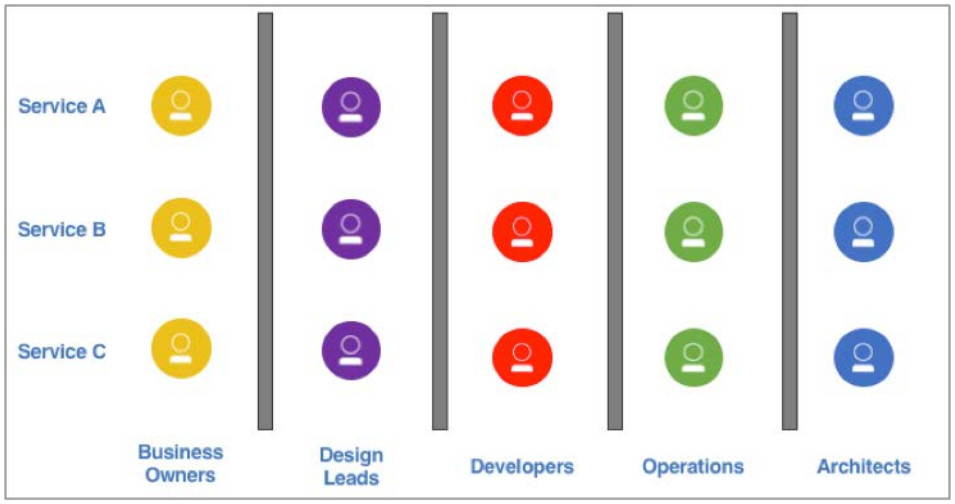

Figura 14. Clásica estructura organizacional de TI

Por comparación, las arquitecturas de microservicios solo pueden tener éxito cuando los equipos tienen el poder de poseer el ciclo de vida completo del desarrollo de software y las operaciones. Para poseer el ciclo de vida completo de DevOps, los equipos necesitan miembros con roles y responsabilidades diferentes.

Estos equipos interfuncionales se construyen para empoderar arquitecturas basadas en microservicios. En vez de miembros de equipos en silos, todos los roles y responsabilidades están contenidos en el mismo equipo. Todos, desde el diseño al desarrollo a operaciones, trabajan muy cerca y a menudo están juntos. Las estructuras del equipo físico están fuera del alcance de este trabajo, pero los límites del equipo virtual son más exitosos en transformar el negocio cuando se forman de una manera interfuncional.

Esto permite que el negocio identifique claramente experiencias de diseño, entienda posibles líneas de tiempo de entrega y minimice gastos operativos, ya que las partes interesadas en el diseño, desarrollo y operaciones están todas representadas en el equipo. La figura 15 muestra una configuración de equipo DevOps óptima.

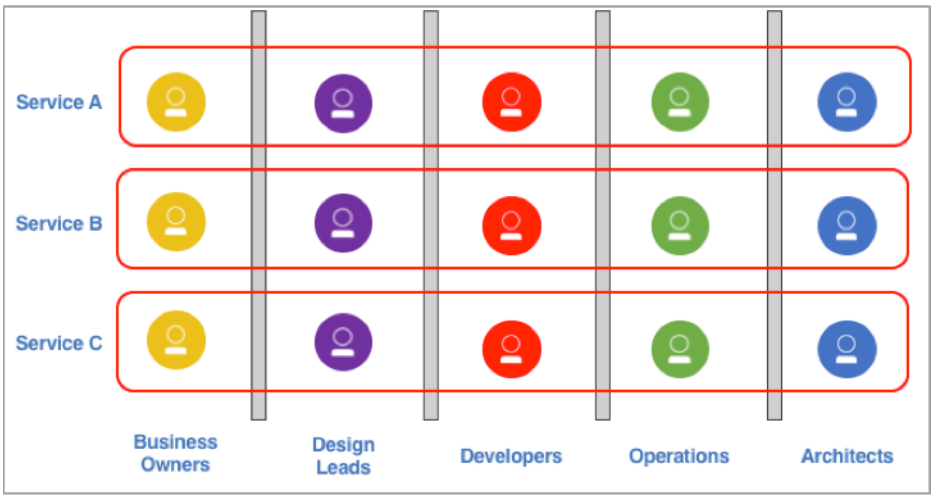

Figura 15. Equipos DevOps optimizados para entregar éxito de microservicios

Un equipo interfuncional también soporta el crecimiento rápido de habilidades individuales en el equipo. Cuando un equipo posee todo aquello de lo que el microservicio es responsable, desde el diseño hasta operaciones y datos de tiempo de ejecución, los miembros individuales del equipo no son relegados a tareas únicas. A menudo, los ingenieros front-end desarrollan habilidades de administración de bases de datos, mientras que los miembros del equipo orientados a operaciones aprenden más sobre las diferencias en infraestructuras de interfaz de usuario. La expansión de habilidades ayuda a toda la organización de TI a tener éxito con los microservicios; es mucho más fácil construir equipos nuevos con aptitudes en los multiples roles versus buscar especialistas para completar roles muy específicos.

A menos que usted solucione el problema comercial y la cultura y conjunto de habilidades de su equipo, no podrá implementar efectivamente la tecnología de microservicios y mantendrá los mismos procesos y estructuras en su lugar.

Entender la tecnología

El análisis apropiado de las pilas de tecnología existentes varía ampliamente de organización a organización, pero el enfoque simplificado que describimos ayuda a asegurar tanto el éxito inicial como el sostenido de sus proyectos de microservicios. Empezar de a poco y definir éxitos reiterados y progresivos es un enfoque mucho más alcanzable y fructífero que un enfoque de transformar todo de una vez.

Figura 16. Enfoque reiterativo para la evolución de microservicios

La primera fase de comprensión de su tecnología es identificar los servicios generales que están en el monolito existente. Usted a menudo puede alinear esto con principios de diseño guiado por el dominio (DDD) que le permiten aplicar principios de diseño bien definidos a aplicaciones existentes que nunca fueron consideradas bajo este esquema. Identificar estos servicios generales le ayuda a entender la complejidad de las estructuras de datos, el nivel de acoplamiento entre los componentes actuales y los equipos que son responsables de los nuevos servicios generales. Una revisión exitosa le da una clara comprensión de los límites de datos, tanto dentro de un servicio dado como a través de los servicios.

Una vez que usted identifica los servicios generales, debe crear un plan para cómo hacerlos evolucionar a microservicios específicos. Estos microservicios, basados en su trabajo previo deberán trabajar todos con datos similares, gestionar sus propios datos y entender que datos necesitan leer y escribir a los otros servicios. Desde aquí, usted puede identificar e implementar resiliencia, escalabilidad y agilidad de los microservicios específicos individuales.

Como se mencionó previamente, las API y microservicios no son una comparación uno a uno, sino que simplemente son dos partes de un todo más grande. Una vez que usted tiene una mejor comprensión de sus microservicios específicos, también tendrá una mejor comprensión de sus interfaces, incluyendo las que están en la ruta crítica, las que son opcionales y las que ya no son necesarias. Si usted no puede correlacionar una interfaz o API existente a uno de sus microservicios generales o específicos, es altamente probable que no sea necesaria.

Dimensionar el esfuerzo de microservicios

Con todo el trabajo de análisis y planificación completado, es hora de definir líneas de tiempo, velocidades de entrega y resultados esperados. Estos variarán ampliamente pero no serán un proceso completamente nuevo a partir de los proyectos existentes hasta proyectos de transformación digital.

El arduo trabajo de entender el negocio, entender la estructura de equipo y entender la tecnología ayudará a asegurar que la organización esté preparada para entender la totalidad de la evolución de los microservicios de cualquier monolito dado; ya sea en un alcance de prueba de concepto, alcance piloto o alcance de evolución a gran escala.

Patrones de microservicios

Patrones de desarrollo para microservicios

En el desarrollo, los patrones son útiles para aplicar soluciones conocidas a tipos comunes de requerimientos y esto también se aplica a los microservicios. Uno de los principios de diseño de microservicios de Martin Fowler es que los microservicios están "Organizados alrededor de capacidades comerciales".[2] Eso surge directamente del descubrimiento de que solo porque usted puede distribuir algo no significa que usted deba hacerlo.

Estos patrones son usados comúnmente para microservicios:

- El **Patrón de fachada** define una API externa específica para un sistema o subsistema. El subtexto de este patrón es que esta API es controlada por los negocios.

- **Patrones de entidad y agregados** son útiles para identificar conceptos comerciales específicos que se correlacionan directamente con los microservicios, para equipos de desarrollo que no están acostumbrados a diseñar en términos de interfaces comerciales.

- **Patrón de servicios** ofrece una forma de correlacionar operaciones que no corresponden a una sola entidad o agregadas con un enfoque unitario que se necesita para microservicios.

- **Patrón de microservicios adaptadores** es útil en el mundo corporativo, donde en muchos casos los equipos de desarrollo no tienen control descentralizado sobre los datos de patrones. En un Microservicio adaptador, usted adapta entre dos API diferentes. Una API es una API orientada a negocios construida mediante el uso de RESTful o técnicas de mensajería liviana, con las mismas técnicas controladas por el dominio que un microservicio tradicional. La segunda API es una API heredada o un servicio SOAP tradicional basado en WS*.

- **Patrón de aplicación Strangler** aborda el hecho de que los negocios y aplicaciones nunca viven realmente en un entorno nuevo. Los programas que más pueden beneficiarse de los microservicios son aplicaciones monolíticas grandes que pueden ser refactorizadas. El patrón provee un enfoque para gestionar la refactorización. El patrón de Aplicación Strangler es cubierto en detalle más adelante en este trabajo.

Patrones de operaciones para microservicios

Si bien los microservicios hacen más rápido cambiar y desplegar un solo servicio, también hacen la gestión y mantención de un conjunto de servicios un esfuerzo mayor comparado con una aplicación monolítica correspondiente. Estos patrones de operaciones para microservicios que fueron desarrollados originalmente para la gestión de aplicaciones convencionales se aplican al costado de operaciones de DevOps:

- **Patrón de registro de servicio**

hace posible cambiar la implementación de los microservicios descendientes y le da la opción de que la ubicación del servicio varíe en diferentes etapas de su conducto DevOps. Esto se logra evitando codificar con rigidez terminales específicas de microservicios en su código. Sin Registro de servicios, su aplicación rápidamente tendrá problemas a medida que los cambios al código empiecen a propagarse en forma ascendente a través de una cadena de llamadas de microservicios.

- **Patrones de ID de correlación y Agregador de registros**

logran mejor aislación, mientras que al mismo tiempo hacen posible depurar más fácilmente los microservicios. El patrón de ID de correlación permite rastrear la propagación a través de un número de microservicios escritos en un número de lenguajes diferentes. El patrón de Agregador de registros complementa al ID de correlación permitiendo que los registros de un número de microservicios diferentes sean agregados a un repositorio único de búsqueda. Juntos, estos patrones permiten una depuración eficiente y comprensible de los microservicios más allá del número de servicios o profundidad de cada pila de llamadas.

- **Patrón de interruptor de circuitos**

ayuda a evitar desperdiciar tiempo en manejar fallas descendentes si usted sabe que ya están ocurriendo. Para hacer esto, usted planta una sección de código de *interruptor de circuitos* en las llamadas de los servicios ascendentes cuando un servicio descendiente está funcionando mal y evita tratar de llamarlo. El beneficio de este enfoque es que cada llamada falla rápidamente. Usted puede proveer una experiencia general mejor a sus usuarios y evitar la mala gestión de recursos como hebras y agrupaciones de conexión cuando usted sabe que las llamadas descendentes están destinadas a fallar.

Foco en el Patrón Strangler

El Patrón Strangler de Fowler se basa en una analogía de una parra que estrangula a un árbol alrededor del cual está enroscada. La idea es que usted use la estructura de una aplicación web (construida a partir de Identificadores de Recursos Uniformes (URI) individuales que correlacionan funcionalmente diferentes aspectos de un dominio comercial) para dividir una aplicación en dominios funcionales diferentes y reemplazarlos con una nueva implementación basada en microservicios, un dominio a la vez. Estos dos aspectos forman aplicaciones separadas que viven lado a lado en el mismo espacio URI. Con el tiempo, la aplicación recientemente refactorizada estrangula o reemplaza a la aplicación original hasta que finalmente usted puede apagar la aplicación monolítica.

Los pasos del Patrón Strangler son transformar, coexistir y eliminar:

- **Transformar** - Crear un sitio nuevo paralelo, por ejemplo, en Bluemix o en su entorno existente, pero basado en enfoques más modernos.

- **Coexistir** - Dejar el sitio existente donde está por un tiempo. Redirigir gradualmente desde el sitio existente al nuevo sitio para la funcionalidad implementada recientemente.

- **Eliminar** - Remover la antigua funcionalidad del sitio existente, o simplemente dejar de mantenerla, a medida que el tráfico es redirigido fuera de esa porción del sitio.

Lo bueno de aplicar este patrón es que crea valor gradual en un marco de tiempo mucho más rápido que si usted tratara de hacer todo en una gran migración. También le da un enfoque gradual para adoptar microservicios; si usted encuentra que el enfoque no funciona en su entorno, tiene una manera simple de cambiar de dirección.

¿Cuándo funciona el patrón de aplicación Strangler y cuándo no?

- **Monolito basado en la red o en API -** Comenzar de un monolito basado en una web existente o en API es el primer requisito para la aplicación exitosa del patrón. El propósito del patrón Strangler es darle una manera de moverse fácilmente hacia adelante y hacia atrás entre la funcionalidad nueva y la vieja. Si su aplicación es una aplicación web, entonces su estructura URL le da una infraestructura para elegir cómo y qué partes del sistema se implementan. Sin embargo, cualquier aplicación basada en un conjunto fijo de API, tales como un conjunto de API SOAP que usted está transformando a REST o hasta un conjunto de colas implementadas en un sistema de mensajería, le permitirán aplicar el patrón. Por otro lado, las aplicaciones de cliente pesado o numerosas aplicaciones móviles nativas no están bien provistas para este enfoque ya que no necesariamente tienen una estructura que le permita a usted apartar la aplicación fácilmente.

- **Estructura URL estandarizada (verdadero uso de URL) -** A pesar de que las aplicaciones web funcionan todas de acuerdo con estándares impuestos por la estructura de la web, por ejemplo, HTTP y HTML, usted puede usar una amplia variedad de arquitecturas de aplicación para implementar aplicaciones web. Hay un gran margen dentro de este enfoque que puede complicar su intento de separar la aplicación. Por ejemplo, cuando hay una capa intermedia debajo de las solicitudes de servidor, por ejemplo, un enfoque de portal, usted puede tener un problema al usar las URL para dividir la aplicación. La decisión de conmutar y enrutar no se hace a nivel de navegador, sino más profundo en la aplicación, lo cual es más complicado.

- **Meta IU -** Cuando la IU está basada en procesos comerciales, o construida al vuelo, se hace difícil separar el código para la IU y la lógica comercial en microservicios diferentes. El enfoque todavía funciona, pero el tamaño del segmento (ver abajo) es más grande y debe ser implementado todo de una vez.

Cómo no aplicar el Patrón de aplicación Strangler

El patrón de Aplicación Strangler no es una cura para todo. Usted no debe aplicarlo en todos los casos especialmente estos:

- No lo aplique en una página a la vez. La "astilla" más pequeña (ver la sección de gestión de versión abajo) es un solo microservicio. Ese microservicio necesita ser completo y autosuficiente y, lo más importante, es que necesita poseer completamente los datos que gestiona. Usted debe evitar tener dos métodos diferentes de acceso a datos para sus datos al mismo tiempo para evitar problemas de consistencia.

- No aplique el patrón todo al mismo tiempo. Si lo hace, no está aplicando realmente el Patrón Strangler y se encuentra de regreso en el enfoque Big Bang.

Cómo aplicar mejor el Patrón de aplicación Strangler

Por lo tanto, si una sola página es demasiado pequeña y una aplicación entera es demasiado grande, ¿cuál es el nivel adecuado de granularidad para aplicar la Aplicación Strangler? Para tener éxito, usted tiene que entrelazar dos aspectos diferentes de su refactorización de aplicaciones:

- Refactorizar su back-end al diseño de microservicios (la **parte interna**)

- Refactorizar su front-end para alojar los microservicios y para hacer cualquier cambio funcional que esté controlando la refactorización (la **parte exterior**)

Comencemos con la parte interior:

1. Comience identificando los contextos limitados en su diseño de aplicación. Un contexto limitado es otro patrón del Diseño guiado por el dominio *de Eric Evans.* Según el libro, un contexto limitado, "define el contexto dentro del cual se aplica un modelo."[3] Un contexto limitado es una infraestructura conceptual que restringe el significado de un número de entidades dentro de un conjunto mayor de modelos comerciales. En una aplicación de aerolínea, la reserva de vuelo es un contexto limitado, mientras que el programa de fidelidad de la aerolínea es un contexto limitado diferente. Si bien pueden compartir términos tales como "vuelo", la manera en la cual se usan y definen esos términos es muy diferente.

2. Elija el contexto limitado más pequeño y menos costoso para refactorizar. Ordene sus otros contextos limitados por orden de complejidad desde el menos complejo al más complejo. Comience con los contextos limitados menos complejos para demostrar el valor de su refactorización y resuelva cualquier problema al adoptar el proceso, antes de encarar tareas de refactorización más complejas y potencialmente costosas.

3. Planifique de manera conceptual los microservicios dentro del contexto aplicando los patrones de Entidad, Agregado y Servicio desde el *Diseño guiado por el dominio* como se describe arriba. En este punto, usted solo está tratando de obtener una comprensión con respecto a qué microservicios probablemente existan para que usted pueda usar la aproximación en el próximo conjunto de pasos.

A continuación, continúe con la parte exterior:

1. Analice las relaciones entre las pantallas en su IU existente. En particular, busque flujos a gran escala que enlacen estrechamente varias pantallas. Si está construyendo un sitio web de una aerolínea, un flujo puede ser reservar un pasaje, que está compuesto de varias pantallas relacionadas que proveen información para completar el proceso de reserva. Un flujo diferente podría estar centrado en inscribirse en el programa de fidelidad de la aerolínea. Entender el conjunto de flujos lo ayuda a continuar hacia el próximo paso de refactorización.

2. Mientras examina su IU existente o su IU nueva, busque los aspectos que correspondan a los microservicios identificados en la parte interior. Identifique qué flujos corresponden a qué microservicios. La salida de este paso es una lista de flujos en un lado de una página y una lista de los microservicios que podría implementar cada flujo en el otro lado.

3. Dimensione su segmento basado en la suposición de que los cambios de IU deben ser autoconsistentes. Suponga que uno o más flujos son el tamaño mínimo de cambio que puede ser liberado a un cliente, pero el segmento mismo puede ser más grande que un flujo. Por ejemplo, puede considerar que toda la fidelidad de cliente es un solo segmento, aun cuando pueda estar compuesto de dos o tres flujos separados.

4. Decida si liberará un segmento completo a la vez o un segmento como una serie de tiras.

Referencias

- **IBM Cloud Garage Method, Microservices Architecture Center**
- **End-to-End Reference Implementation (GitHub)**
- **Refactoring to Microservices**
- **Microservices Patterns IO**
- **Exploring Microservices: Game-On (GitBook)**

[1, 2]"Martin Fowler, *Microservices a definition of this new architectural term*, *https://martinfowler.com/articles/microservices.html*

[3]Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software, 2003*

Fin.