Nicolò Pignatelli Seguir
Director de Ingeniería, Diseño Dirigido por Dominio, Gestión Ágil. Suscríbase a mi boletín informativo sobre desarrollo de software aquí: http://eepurl.com/dfQQaz
28 de diciembre de 2017 · Lectura de 5 minutos

La verdad sobre microservicios

Otro caso de redacción infeliz que genera confusión y muchos dolores de cabeza para desarrolladores y mantenedores

Una arquitectura de microservicio

Tenemos un término roto. Son **microservicios** .

Desde que salió en 2014, la definición más extendida y aceptada de un microservicio ha sido, parafraseo, " *una función grasosa que se puede desplegar de forma autónoma con una capa de comunicación delgada, a veces implícita* ".

Se ha desarrollado todo un ecosistema junto con toneladas de literatura para apoyar el concepto subyacente.

El acento estaba puesto en la increíble ventaja que obtiene al desplegar su pequeña pieza de código de manera autónoma y desacoplada entre sí. De repente, cada equipo puede trabajar de forma independiente y entregar con simplicidad, protegido por sus límites de microservicio.

Pero esto es la verdad? Después de todo, esta es la promesa que se ha vendido junto con el paquete.

> *"Un microservicio es una función adiposa que se puede desplegar de forma autónoma con una capa de comunicación delgada, a veces implícita, alrededor de" - El promedio de internet*

Desafortunadamente, resulta que las cosas son más complicadas que eso. Vamos a dar un ejemplo.

Supongamos que tiene un flujo de registro de correo electrónico de usuario que acepta una solicitud a través de una API HTTP y almacena los datos enviados en una tabla de base de datos.

La forma común de microservicio de abordar este escenario es dividir las responsabilidades entre dos, a veces tres microservicios.

1. El microservicio HTTP api, responsable de aceptar, formalizar y reenviar una solicitud de registro.
2. El microservicio de registro de correo electrónico del usuario es responsable de proteger las reglas comerciales, como la exclusividad del correo electrónico.
3. El usuario de correo electrónico persistence microservice, responsable de guardar el correo electrónico de usuario recién creado a la base de datos.

Me detendré aquí por simplicidad, pero ya tenemos tres microservicios para nuestro escenario.

Ahora mi pregunta:

¿Qué sucede cuando tenemos nuevos requisitos como "para que un registro sea exitoso, una solicitud debe contener tanto un correo electrónico válido como un nombre de usuario"?

De repente, **nos encontramos orquestando la actualización y el despliegue de nuestros tres microservicios en conjunto** .

Vamos a entender por qué.

CASO 1 . *Actualizamos primero el microservicio de registro de correo electrónico del usuario (n. ° 2) con el nuevo requisito.*

Nuestro flujo de registro deja de funcionar inmediatamente: dependiendo de la implementación, el # 1 puede o no puede romperse debido a un campo inesperado en la solicitud, pero definitivamente no lo reenviará al # 2, que en consecuencia rechazará todas las solicitudes entrantes.

CASO 2 . *Actualizamos el microservicio API api (n. ° 1) primero con el nuevo requisito.*

1 comienza a enviar el correo electrónico y el nombre del usuario al # 2. Dependiendo de la implementación y el protocolo de comunicación, # 2 puede rechazar o no el formato de solicitud desconocido. En el mejor de los casos, solo el correo electrónico se considera y se reenvía al n. ° 3. El servicio no está entregando lo que promete la API.

CASO 3 . *Actualizamos el servicio de microservicio de almacenamiento (n. ° 3) con el nuevo requisito primero.*

1 puede o no romperse debido a un campo inesperado en la solicitud, pero definitivamente no lo reenviará al # 2. Dependiendo de la implementación, # 3 puede guardar o no los datos del usuario. El peor caso ocurre cuando aplicamos una restricción no vacía en la representación de datos del nombre de usuario al implementar la nueva versión, en cuyo caso el motor de almacenamiento rechazará cualquier solicitud entrante.

¿Cómo resolvemos el problema? Como se dijo antes, tenemos que organizar múltiples implementaciones. Esto se puede hacer de varias maneras, una de ellas es:

1. actualice # 1 para que acepte un nombre de usuario opcional junto con el correo electrónico, pero no lo reenvíe al # 2
2. actualice # 2 para que acepte un nombre de usuario opcional junto con el correo electrónico, pero no lo reenvíe al # 3

3. actualización # 3 para que acepte y almacene un nombre de usuario opcional junto con el correo electrónico
4. actualice # 1 para que solo acepte una solicitud con el correo electrónico y el nombre del usuario, luego reenvíelo al # 2
5. actualice # 2 para que solo acepte una solicitud con el correo electrónico y el nombre del usuario, y luego reenvíelo a # 3
6. actualización # 3 para que solo acepte y almacene el correo electrónico y el nombre del usuario

Guau, 6 actualizaciones, 2 en cada microservicio. ¿No es esto derrotar la promesa de despliegue autónomo? Lo parece.

El ejemplo fue trivial, y sin embargo descubrimos lo complicado que es desplegar un requisito simple como ese.

¿Hueles igual que yo? Al no poder actualizar su software fácilmente, todo está enredado en todo, el software se rompe incluso si formalmente no ha hecho nada incorrecto ... ¡sí! Es un monolito Y ahora es uno distribuido.

Este tweet de Kelsey Hightower lo dice todo:

Kelsey Hightower
@kelseyhightower

Predicción 2020: las aplicaciones monolíticas volverán a estar de moda una vez que las personas descubran los inconvenientes de las aplicaciones monolíticas

Una arquitectura de microservicio no es una excusa para reducir tanto como puedas los límites de tus unidades desplegables. Hacerlo solo creará una gran sobrecarga en la orquestación del desarrollo y la implementación de su proyecto. Hay aún más inconvenientes en poder probar su aplicación de extremo a extremo cuando tiene una arquitectura de microservicio, pero esto llevaría demasiado tiempo para el propósito de esta publicación.

¿Qué debería uno hacer, entonces?

La solución a este problema es simple, pero no es nada fácil: **uno debe encontrar los límites correctos para sus servicios** micro.

¿Por qué no es una tarea fácil? Porque los límites son un objetivo en movimiento y no son visibles cuando comienzas a buscarlos. Una gran parte de la especulación acerca del Diseño Dirigido por el Dominio trata exactamente sobre cómo encontrar los límites de sus sistemas.

Pero no temas, algunas heurísticas y trucos se aplican aquí y son definitivamente útiles en la búsqueda del "tamaño" perfecto del servicio.

1. si dos componentes necesitan hablar directamente entre sí (en una respuesta de solicitud), manténlos juntos en el mismo servicio;
2. si identifica una relación aguas arriba-aguas abajo entre dos componentes, puede separarlos en dos servicios diferentes;
3. los servicios upstream pueden anunciar lo que sucedió al publicar eventos en un sistema pub-sub;
4. los servicios descendentes escucharán los eventos publicados a los que están interesados y en consecuencia reaccionarán;
5. los eventos publicados deben contener solo detalles relevantes a lo sucedido, pero suficientes datos para ser útiles para los receptores. Tú decides el equilibrio aquí;
6. un equipo no debe depender de otro para entregar una característica de extremo a extremo; modifique sus límites de servicio en consecuencia si se encuentra en la situación opuesta (vea la Ley de Conway y sus postulados);
7. las responsabilidades del software son diferentes de las responsabilidades del equipo; un equipo puede ser responsable tanto del backend como de la parte frontend de un servicio, pero estos dos componentes se pueden implementar de forma independiente; el equipo es responsable de decidir la arquitectura interna del servicio;
8. si está iniciando greenfield, opte por una arquitectura de monolito / repositorio único y tenga cuidado de desacoplar los módulos internos uno de otro. Es mucho más fácil descomponer un monolito más tarde que agregar múltiples microservicios (o al menos los desarrolladores tienen más experiencia en hacer lo anterior);
9. por último, pero no menos importante, una sugerencia general: ser ágil, hacer experimentos. Si algo funciona para usted y no lo

impide, hágalo. De lo contrario, sea lo suficientemente valiente como para reconocer la falla, aprenda de ella y cambie.

Nicolò Pignatelli
@nicolopigna

Regla general de la arquitectura distribuida: mantener juntos lo que requiere solicitud-respuesta. Separe lo que puede seguir un modelo pub-sub

. . .

Si disfrutaste esta publicación, haz clic en el botón and y comparte para ayudar a otros a encontrarla. Siéntase libre de dejar un comentario más abajo.

También puede suscribirse a mi boletín de Desarrollo de software aquí: http://eepurl.com/dfQQaz