Capítulo 5: Dividiendo aplicaciones monolíticas

Víctor Iranzo

26 de mayo de 2018

En este capítulo se explica como abordar la transformación de una aplicación monolítica de forma evolutiva hacia un sistema basado en microservicios. Las aplicaciones monolíticas tienden a crecer con el tiempo, por lo que se hacen frágiles y poco mantenibles al juntar muchas veces código no relacionado. Es por este motivo por el que un equipo pueda preferir no modificar una aplicación así, al menos no de forma descontrolada.

1. Costuras: pasos para dividir lo monolítico

En el libro "Working Effectively with Legacy Code"se define una costura como una porción de código que se puede tratar de manera asilada sin alterar al resto del sistema. Las costuras son firmes candidatos a convertirse en futuros servicios.

El primer paso de nuestra refactorización será identificar las costuras. Muchos lenguajes de programación permiten la creación de espacios de nombres o paquetes. Si podemos, moveremos todo el código del contexto que hemos encontrado a un nuevo paquete mediante refactorizaciones del IDE.

Siguiendo este procedimiento, terminaremos viendo qué código se ha agrupado correctamente y qué código parece que no encaje en ningún paquete. El código sobrante puede estudiarse para ver si se puede agrupar como uno o varios paquetes o se puede añadir a la solución de otra forma.

Durante todo el proceso, el código debe representar una situación real, por lo que las interacciones y dependencias entre los paquetes será similar a la existente en la realidad. A la hora de elegir por qué paquete comenzar, podemos elegir el que supongamos que nos aportará mayor beneficio al separarlo del resto o el paquete que menos dependencias tenga con el resto.

También cabe mencionar que la transformación a microservicios no ne-

cesita realizarse de golpe, sino que se pueden ir transformando paquetes progresivamente para limitar el impacto de hacer algo mal. Muchas dudas sobre las referencias entre servicios pueden resolverse donde el coste del cambio es el menor posible: el papel. Otra posible técnica es el uso de tarjetas clase-responsabilidad-colaboradores (CRC) pero aplicado a servicios, listando las capacidades y las referencias a terceros que tiene cada uno.

2. Beneficios de refactorizar aplicaciones monolíticas

La clave para refactorizar es darse cuenta de cuando una aplicación necesita ser dividida, porque crezca por encima de lo "sano", antes de que resulte demasiado costoso el cambio. Los beneficios obtenidos al hacerlo son los siguientes:

- Tranquilidad de cambio: un cambio en un servicio se podrá hacer más rápido de lo que se hacia antes, además de poderse implementar y desplegar de forma autónoma.
- Organización de los equipos: diferentes equipos pueden encargarse de distintos microservicios sin que interfieran los cambios de unos con los de otros.
- Seguridad: la seguridad ahora puede aplicarse a un servicio concreto en lugar de a todo el sistema.
- Tecnológica: los servicios son más sencillos de refactorizar y cambiar a una nueva tecnología se puede hacer de forma progresiva y aislada en un único contexto.

3. Refactorizaciones en bases de datos

La base de datos es la infraestructura donde más enredo de dependencias hay. El cambio de una aplicación monolítica a una basada en servicios se ha de hacer evolutivamente.

El proceso a seguir para separar la persistencia de la aplicación en diferentes bases de datos consiste también en buscar costuras. Vamos a explicar algunos problemas que se pueden encontrar en este proceso.

3.1. Claves ajenas

Un problema típico es que una tabla en BD que pertenece a uno de los microservicios extraídos referencie a través de una clave ajena a un elemento de una tabla de otro contexto. Estas dos tablas van a estar en servicios diferentes, así que para obtener los datos que una tupla referencia de la otra tabla, en lugar de recorrer la clave ajena, haremos una llamada a la API del otro servicio.

Como consecuencia, cuando queramos obtener un elemento de la primera tabla haremos primero una llamada a la base de datos de nuestro servicio y después una llamada a la API del otro microservicio para obtener el elemento referenciado. Este microservicio hará una llamada a su base de datos y devolverá el elemento referenciado al servicio que lo ha invocado.

Al haber eliminado la clave ajena explícitamente en base de datos, las restricciones asociadas a esta (como por ejemplo la existencia del elemento referenciado) han de mantenerse a nivel de código.

3.2. Datos estáticos compartidos

Algunas tablas son referenciadas por muchas otras y cambian inusualmente debido al carácter de los datos que almacenan, por ejemplo, tablas con códigos de países. Si diferentes contextos acceden a esta tabla, una posible solución es duplicar la tabla en cada nueva base de datos de los servicios e implementar algún mecanismo que mantenga la consistencia en todas ellas.

Otra manera aún más sencilla es mover estos datos al código como un archivo o recurso en cada servicio. El problema de la consistencia sigue estando, pero es más fácil de resolver en el código que en las bases de datos.

Una tercera solución es crear un microservicio que tenga estos datos estáticos. Los servicios que quieran acceder a ellos lo harán llamando a su interfaz. Esta alternativa puede ser demasiado compleja si los datos estáticos tienen poco volumen o complejidad.

3.3. Datos mutables y tablas compartidas

Surge un problema cuando dos de las costuras que hemos encontrado acceden a la misma tabla en modos de lectura y escritura. Esto puede ocurrir porque estamos almacenando en una tabla conceptos que no son de un mismo dominio o porque existe una entidad del dominio en la base de datos que no

existe como tal en el código.

En el primer caso, la respuesta pasa por dividir la tabla en dos porque se están mezclando conceptos de diferentes dominios. Cada uno de los microservicios acogerá una de las dos tablas.

En el segundo caso, la solución pasa por crear como tal la entidad en el código y que ya existía en la base de datos. Esta nueva entidad pertenecerá a un nuevo microservicio. Para abordar la refactorización progresivamente, primero lo crearemos como un paquete independiente y después haremos que todos los servicios que accedían a la antigua tabla accedan a la entidad a través de llamadas a la interfaz que expone el servicio.

4. Transacciones

Las trasacciones nos permiten establecer un conjunto de acciones que o bien se ejecutan juntas correctamente o bien ninguna se ejecuta si alguna de ellas falla. Si una de las acciones falla, el resto son deshechas. De esta manera nos aseguramos que siempre transitamos de un estado consistente a otro también consistente.

Al haber separado el esquema de la base de datos de la aplicación monolítica en diferentes esquemas hemos perdido el comportamiento transaccional. Si tenemos que ejecutar en dos esquemas distintos una serie de acciones ligadas, ¿qué podemos hacer si una falla y otra acción no?

- Intentarlo de nuevo más tarde: la acción que ha fallado puede encolarse o escribirse en un registro para volver a ejecutarla más tarde. En lugar de garantizar la consistencia en todos los estados, se está garantizando de forma eventual. La consistencia eventual acepta estados inconsistentes con la premisa de que en algún momento se alcanzará un estado consistente para una transacción.
- Deshacer todas las operaciones: al hacerlo, volveremos a un estado consistente mediante transacciones de compensación. Las transacciones de compensación son transacciones nuevas que deshacen una hecha previamente, como puede ser un DELETE de una tupla. En caso de que la transacción de compensación falle, se pueden establecer procesos periódicos para la búsqueda y limpieza de inconsistencias.
- Transacciones distribuidas: realizan varias transacciones simultáneamente a través de un gestor de transacciones que orquesta las que

tiene que realizar cada servicio. El algoritmo más común de esta solución es el commit en dos fases: en una primera fase, los servicios involucrados en el commit comprueban si pueden realizar la transacción que deben llevar a cabo. En caso de que todos los servicios puedan continuar, se realiza una segunda fase en donde cada uno de ellos hace su transacción. Entre las desventajas de este procedimiento están que hay un proceso centralizado que gobierna al resto y que puede bloquearlos si falla y que puede introducir bloqueos ya que el algoritmo ha de esperar a que todos estén disponibles para realizar el conjunto de acciones.

5. Interacción con grandes volúmenes de datos

Algunos servicios necesitan interaccionar con muchos otros para realizar su función, por ejemplo, para la generación de informes. Por un lado, en una aplicación monolítica los datos necesarios para generar un informe se pueden obtener a través de pocas consultas a la base de datos. Por otro lado, cuando el sistema está formado por microservicios, la obtención de los datos se realiza haciendo múltiples llamadas a las interfaces de los servicios implicados.

Realizar múltiples llamadas a las APIs puede ser un problema si se trabaja con grandes volúmenes de datos, sobretodo si las interfaces no están preparadas para trabajar así. Guardar una copia local de los datos necesarios para generar informes puede ser peligroso si no se implementar los mecanismos necesarios para invalidar datos.

5.1. Obtener los datos a través de llamadas a servicios

Una primera solución pasa por hacer que las APIs puedan trabajar por lotes, es decir, que nos permitan obtener datos de múltiples instancias con una única llamada a través de mecanismos como la paginación.

Si el tiempo para obtener la respuesta con los datos requeridos es muy largo, es mejor no emplear una comunicación basada en el patrón petición-respuesta. Esto se puede realizar a través de archivos accesibles a ambos participantes y códigos HTTP como son el 202 y el 201, que indican respectivamente que la petición ha sido aceptada pero no procesada todavía y que ya se ha terminado de procesar.

Como desventaja podemos decir que añadir a las interfaces de los ser-

vicios funciones para hacer llamadas agregadas supone un esfuerzo para solucionar un problema muy particular como el de generar un informe.

5.2. Bombeo de datos

Otra solución consiste en delegar la responsabilidad de obtener los datos en un proceso independiente. En nuestro ejemplo, no será el servicio de informes el encargado de hacer la llamada a los servicios para obtener los datos, sino que estos datos los encontrará en su propia base de datos porque otro proceso ahí los ha insertado. Dicho proceso se encargará de acceder a la base de datos de otros servicios y copiar en la de informes la información relevante.

Aunque la integración en base de datos es una mala idea, esto puede considerarse una excepción porque facilita la implementación del servicio de informes. Algunas tecnología de BD pueden facilitar el mapping entre el esquema al que pertenecen los datos y el esquema de informes a través de vistas agregadas.

5.3. Bombeo de datos basado en eventos

Una cuestión relevante es cuándo va a ejecutarse el proceso para bombear los datos. Si la consistencia de los datos es muy relevante, no puede ser una tarea que se ejecute periódicamente porque no se actualizaría inmediatamente al hacer una modificación sobre una tupla.

En este caso, se puede hacer el bombeo mediante eventos: cada servicio lanza uno cuando se hace una modificación sobre alguna de las tablas que posee. El proceso captura estos eventos para ejecutarse. La modificación que se ha realizado se puede representar como un delta para que el proceso que se lanza solo tenga que realizar esos cambios sobre la tabla destino.

Entre las ventajas de la aproximación basada en eventos es que el proceso puede evolucionar de forma independiente, reduce el acoplamiento y, por supuesto, mantiene los datos actualizados.