Capítulo 4: Integración de microservicios

Víctor Iranzo

23 de mayo de 2018

Parte I

Introducción

La integración de servicios es la parte más relevante en los sistemas basados en este concepto. Hacerlo correctamente nos asegurará su autonomía y su despliegue de manera independiente. Existen muchas tecnologías para la integración: SOAP (Simple Object Access Protocol), RPC (Remote Procedure Call) o REST (Representational State Transfer). De cualquiera de estas tecnologías esperamos las siguientes características:

- Evitar cambios en los consumidores: la tecnología escogida debe hacer que el número de cambios en un servicio que impliquen cambios en sus consumidores sean los menos posibles.
- No imponer una tecnología específica: la tecnología empleada para la comunicación entre servicios no debe restringir la tecnología empleada en estos. Se debe mantener la heterogeneidad tecnológica de los servicios y el protocolo empleado para integrarlos debe poderse emplear en cuantas más tecnologías mejor.
- Hacer simple el consumo de un servicio: los consumidores deberían de tener total libertad en la tecnología que emplean y consumir un servicio para ellos no debe ser complejo de implementar.
- Ocultar detalles de la implementación: el consumidor de un servicio no debe conocer los detalles de como este está implementado internamente. Así, los interlocutores están desacoplados y se evitan cambios en el consumidor asociados al servicio.

- Soportar operaciones más allá de las CRUD: las operaciones CRUD para crear, leer, actualizar y eliminar elementos están soportadas en la mayoría de tecnologías de integración. Sin embargo, un sistema requiere dar soporte a más procesos que se deben poder exponer en una interfaz de un servicio.

1. Llamadas síncronas y asíncronas

En la comunicación síncrona, cuando se realiza una llamada a un servidor esta se bloquea hasta que la operación solicitada se complete. En la comunicación asíncrona, el invocador no espera a que la llamada se complete para continuar trabajando.

Es más sencillo razonar sobre las primeras, pero son las llamadas asíncronas las más efectivas cuando se solicitan operaciones largas o cuando se desea hacer una aplicación responsive.

El primer tipo de comunicación está ligado al patrón petición-respuesta (en inglés, request-response), aunque también puede ser empleado de forma asíncrona a través de callbacks que recojan la respuesta del servidor. Otro tipo de patrón es el basado en eventos: el cliente, en lugar de hacer una petición para realizar una operación, comunica al servidor un evento que ha observado y es el servidor quien debe saber qué hacer ante lo ocurrido. Este patrón es de naturaleza asíncrona y disminuye el acoplamiento entre cliente y servidor.

2. Patrones orquestador y coreógrafo

A la hora de implementar un proceso de negocio múltiples servicios deben participar. Para su coordinación, existen pos estilos arquitectónicos que se pueden seguir: la orquestación, basado en un componente que guía y dirige al resto de servicios como un director de orquesta, y la coreografía, donde cada parte sabe el trabajo que debe realizar y se coordina con el resto como bailarines realizando una coreografía.

Por una parte, el patrón orquestador es más sencillo de implementar, pudiéndose hacer con herramientas para el modelado de procesos de negocio. Sin embargo, al usarlo se otorga demasiada autoridad a uno de los servicios del sistema y el resto pueden llegar a transformarse en simples servicios CRUD.

Por otra parte, el patrón coreógrafo se basa en la comunicación asíncrona. El proceso se inicia mediante un evento al que los servicios que deban reaccionar se suscriben. Así se puede formar una cadena donde cuando uno de los servicios acaba puede comenzar la actividad siguiente mediante el envío de un evento. Su principal ventaja es que reduce el acoplamiento entre los servicios, pero puede llegar a dificultar la visión del proceso de negocio y requiere más trabajo para ser monitorizado.

En general, las soluciones que siguen el patrón orquestador son más difíciles de cambiar y se suelen programar de forma síncrona, por lo que puede ser perjudicial para procesos largos. Es por esto que se recomienda más el segundo patrón, mucho más flexible, aunque se puede optar por una solución híbrida.

Parte II

Integración por base de datos

La integración a través de bases de datos es una de las más empleadas en la industria. En nuestro contexto, esto se traduce a que todos los servicios de un sistema comparten la misma base de datos y cuando uno de ellos quiere hacer una operación u obtener información de otro, lo hace a través de la base de datos sobre las entidades del servicio consumido.

Este tipo de integración tiene un gran número de desventajas. En primer lugar, se está permitiendo a terceros ver y modificar detalles de la implementación interna de un servicio. Como consecuencia, un cambio en el esquema de la base de datos puede suponer romper los consumidores, que se puede comprobar invirtiendo en pruebas de regresión.

En segundo lugar, los consumidores están obligados a emplear una tecnología específica. De esta forma los servicios pierden autonomía a la hora de elegir su propia tecnología y quedan acoplados unos con otros. Por ejemplo, un cambio en el tipo de base de datos empleada (de un esquema relacional a uno clave-valor) implica cambios en todo el sistema.

Por último, con esta aproximación la lógica del servicio queda repartida entre sus consumidores. Acciones como modificar un elemento del servicio pueden ser invocadas por múltiples consumidores. Pongamos por caso que esto lo realizan 3 consumidores. Cada uno es responsable de hacer una consulta a la base de datos para este propósito, lo que se traduce en que la misma lógica quede replicada en tres sitios. Esto supone la pérdida de cohe-

sión en el sistema, porque servicios independientes cambiarían juntos por consumir a otro.

En definitiva, la integración por base de datos es una buena forma de compartir datos pero no lógica y emplearla en una solución basada en microservicios implica la pérdida de la cohesión y el acoplamiento entre servicios.

Parte III

Integraciones basadas en el patrón petición-respuesta

3. Integración RPC

La llamada a procedimiento remoto (en inglés, Remote Procedure Call o RPC)es una técnica que permite ejecutar una llamada a un servicio remoto como si de una llamada local se tratara. Existen diferentes tecnologías que la implementan a través del uso de interfaces, que facilitan la comunicación entre cliente y servidor. No es necesario que cliente y servidor empleen la misma pila tecnológica, aunque algunas tecnologías como Java RMI sí lo requieren y limitan al resto de servicios.

El formato de los mensajes o los protocolos de red empleados también varían de una tecnología a otra. En cuanto a los formatos, SOAP (proviene del inglés Simple Object Access Protocol) emplea XML (también del inglés eXtensible Markup Language) mientras que por ejemplo Java RMI transmite mensajes binarios. Los protocoles de red empleados van desde HTTP sobre TCP, que emplea SOAP, hasta UDP, un protocolo más rápido y ligero.

El coste asociado a hacer una llamada remota es mucho mayor al de una local. Los objetos han de serializarse y mandarse a través de la red, teniendo en cuenta que esta no es segura. El tratamiento de errores no puede ser el mismo y ha de estar prevenido de la latencia de la red.

Algunas tecnologías RPC pueden ser frágiles, como por ejemplo Java RMI. Las interfaces que ofrecen los servidores están muy ligadas a la representación del objeto sobre el que se desea hacer una operación. Además, un cambio en la interfaz puede suponer romper alguno de los consumidores, con lo que puede ocurrir que al desplegar el servidor se tengan que desplegar todos los clientes que lo invocan.

4. Integración REST

La transferencia de estado representacional (en inglés, REpresentational State Transfer) es un estilo arquitectónico inspirado en la Web. Se basa en el concepto de recurso, un objeto que el servicio conoce y del que puede crear diferentes representaciones bajo demando. La representación del recurso está completamente desacoplada de como se almacena.

La arquitectura REST es ampliamente usada junto con HTTP (término que proviene del inglés Hypertext Transfer Protocol). Los verbos que se definen en HTTP actúan sobre recursos: por ejemplo, con el verbo GET se puede obtener una representación del recurso y con el POST crear uno nuevo.

4.1. HATEOAS: Hypermedia As the Engine of Application State

Un principio introducido en HTTP y que reduce el acoplamiento es el de hipermedia como motor del estado de la aplicación (HATEOAS). Este consiste en que un cliente, a través de enlaces que contiene un recurso hacia otros, interactúa con el servidor navegando con los enlaces hacia los recursos que le interesan.

En una conversación normal entre cliente y servidor, el cliente comenzará solicitando un recurso que estará enlace con otros a través de referencias. El cliente debe conocer la semántica del resultado devuelto por el servidor para saber que significan cada uno de los enlaces del recurso. Aunque cambie la lógica del servidor, el significado del enlace seguirá siendo el mismo y se podrá consumir el recurso de igual manera. Esto reduce el acoplamiento entre servidor y cliente, a menos que se cambie la semántica del servidor y cada enlace pase a tener otro significado. Como consecuencia, esto puede suponer que se rompan todos los consumidores.

Entre las desventajas de este principio se encuentra que la comunicación entre cliente y servidor se ve alargada porque el primero tiene que hacer numerosas navegaciones para encontrar el recurso que busca.

4.2. Frameworks para crear servicios RESTFul

Existen algunos frameworks como Sprint Boot que promueven malas prácticas para fabricar servicios más rápidamente. Estos frameworks toman la representación en base de datos de los objetos, la deserializan y exponen

al exterior del servicio para su consumo. Se debe hacer una diferenciación entre cómo se almacenan los datos y cómo se exponen a otros servicios para reducir el acoplamiento entre ambas representaciones.

Una técnica que se puede emplear es retrasar la implementación de cómo se almacenan los datos hasta que se estabilice la interfaz del servicio. De esta manera nos aseguraremos que la representación que obtienen los consumidores es la que realmente quieren y no depende de cómo se almacena.

4.3. Desventajas de usar REST junto con HTTP

- Generar un servidor REST empleando HTTP no es tan sencillo como hacerlo empleando RPC.
- Algunos frameworks no soportan todos los verbos HTTP.
- No es recomendable en redes con latencias bajas: HTTP es más recomendable que SOAP para enviar grandes volumenes de datos por ser más compacto y permitir formatos como el JSON o el binario. Sin embargo, ante latencias bajas es mejor emplear sobre el protocolo TCP otro protocolo de aplicación, como WebSockets, o directamente usar UDP, que además trabaja mejor con mensajes cortos.

Parte IV

Integración basada en eventos

5. Colas empleando RabbitMQ y HTTP con ATOM

En la comunicación asíncrona existen dos problemas a resolver: cómo los microservicios emiten eventos y cómo otros conocen que se ha producido dicho eventos.

Un bróker de mensajería es un patrón arquitectónico para la validación, la transformación y el ruteo de mensajes. Algunos como RabbitMQ buscan solucionar ambos problemas. En RabbitMQ el productor de un evento publica este a través de una API, donde será tramitado por el bróker para hacerlo llegar a todos los consumidores suscritos.

Por una parte, el uso de este framework añade complejidad porque se debe mantener una nueva infraestructura basada en colas para el envío del

eventos.

Por otra, fomenta la escalabilidad y resilencia de los servicios. Las colas son una excelente manera de implementar arquitecturas basadas en eventos y una manera de reducir el acoplamiento. Además, previene de añadir lógica en el middleware que debe ser añadida en los propios microservicios.

Otra aproximación posible es la basada en HTTP para la propagación de eventos. ATOM es una especificación web para la publicación de fuentes web (en inglés, web feeds). Con esta tecnología el emisor publicaría el evento como un feed que después el cliente recupera. Aunque puedan emplearse librerías existentes para este propósito, los receptores deben implementar un mecanismo para gestionar y recuperar periódicamente los mensajes. También recodar que el uso de HTTP con latencias bajas es poco recomendable.

6. Complejidad de arquitecturas asíncronas

Las arquitecturas basadas en eventos están más desacopladas, pero requieren de otro tipo de razonamiento a la hora de ser implementadas. Algunas buenas prácticas para afrontar su complejidad van desde el uso efectivo de sistemas de monitorización hasta el uso de identificadores de correlación para trazar las llamadas entre servicios.

Parte V

Control de versiones de microservicios

7. La ley de Postel y los lectores tolerantes

La mejor manera de evitar cambios que puedan romper los clientes es elegir una tecnología de integración adecuada. La tecnología escogida debe separar los detalles de la implementación del servicio de la interfaz de este.

Los consumidores pueden romperse fácilmente aunque se presuponga que un cambio no es significativo, como por ejemplo eliminar un atributo que nadie emplea. Los lectores tolerantes son aquellos que obtienen de un mensaje solo los atributos que le interesan, ignorando el resto. Estos lectores

cumplen con la ley de Postel sobre la robustez, ya que se previenen de elementos ajenos que pueda contener un mensaje:

> Sé conservador en lo que haces y liberal en lo que aceptas del resto.

7.1. Control de versiones semántico

Con el control de versiones semántico, cada versión se numera de la forma MAYOR.MENOR.PARCHE.

- Cuando se incrementa el número MAYOR significa que se ha realizado un cambio que incompatibiliza con versiones antiguas.
- Cuando se incrementa el número MENOR significa que se han añadido funcionalidades nuevas pero que las existentes previamente se pueden seguir empleando.
- Por último, el incremento en el PARCHE quiere decir que se ha solucionado un bug detectado en una funcionalidad existente.

Este tipo de versionado hace al cliente comprensible las expectativas que debe tener sobre una versión del servicio y el impacto de una nueva versión sobre el cliente.

7.2. Coexistencia de diferentes interfaces

Esta aproximación consiste en mantener la vieja y la nueva interfaz de un servicio disponibles. La interfaz vieja se puede dejar hasta asegurarse de que ningún cliente esta empleándola, para evitar su ruptura. Esto último requiere monitorizar el servicio para saber cuántos usuarios usan cada punto de entrada.

Internamente, las llamadas a la vieja interfaz pueden ser transformadas en llamadas a la nueva. De esta forma, el código a eliminar cuando se deje de mantener la interfaz antigua está localizado.

Esta técnica está basada en el patrón de expansión y contrato porque cuando ninguno de los clientes emplea la vieja funcionalidad, esta se elimina y la nueva API representa el contrato entre ambas partes.

7.3. Versiones concurrentes de un servicio

Otra solución para el control de versiones es tener diferentes versiones de un servicio funcionando y redireccionar las peticiones viejas a la versión vieja del servicio y al contrario con las nuevas peticiones. Esta solución es ocasionalmente empleada por Netflix cuando el coste de cambiar a los viejos consumidores es muy alto.

Esta aproximación tiene algunas desventajas. En primer lugar, si se debe hacer una modificación como solucionar un defecto, posiblemente se deba hacer y desplegar en ambas versiones. En segundo lugar, se necesita un middleware con la lógica para redireccionar las peticiones a una u otra versión. En tercer lugar, si se debe persistir el estado de alguna entidad, las modificaciones hechas sobre este estado deben ser visibles en una y otra versión del servicio. En conclusión, esta solución puede resultar apta para periodos cortos de tiempo, pero cuanto más tiempo permanezcan los clientes empleando la vieja versión del servicio más riesgo tendremos de que ocurran estos problemas.

Parte VI

Integración en las interfaces de usuario

Las interfaces de usuario (UI) es donde juntamos los datos y funcionalidades de diferentes servicios para ofrecer al usuario algo que le aporte valor. La tendencia es que los terminales donde se renderizan estas interfaces sean cada vez más simples y con menos lógica (en inglés, el término es thin client).

La parte servidora debe estar preparada para tratar de igual manera diferentes plataformas como los dispositivos móviles o los navegadores web. Las restricciones son las diferentes maneras en que un usuario puede interactuar con nuestro sistema. Aunque los servicios de back-end sean los mismos, es necesario adaptar la UI a las diferentes restricciones. Se pueden emplear diferentes modelos para su composición o una solución híbrida de todos ellos.

8. Composición con interfaces de servicios

Los servicios se comunican entre ellos empleando formatos conocidos como JSON o XML. Esta propuesta consiste en poder invocar desde la UI cualquier interfaz de un servicio.

Esta solución presenta algunos inconvenientes. Primero, la respuesta obtenida de un servicio es igual independientemente de la plataforma de la aplicación. Segundo, las interfaces de los servicios usados no están preparadas para ser usadas por la UI. Esto puede conducirnos a que para componer una interfaz de usuario sea necesario hacer múltiples llamadas al servicio. Como solución a este último problema se puede emplear una fachada de los servicios del back-end que realice de forma agregada estas llamadas.

9. Composición empleando fragmentos

En lugar de dar la responsabilidad a la UI de llenar con datos sus propios controles, esta aproximación se basa en que cada servicio proporcione directamente una parte de la interfaz de usuario asociada a él.

Entre las ventajas podemos citar que el equipo responsable de un servicio es el responsable final de crear su fragmento de la UI asociado, lo que puede aumentar la velocidad de desarrollo.

Sin embargo, que cada equipo se encargue de sus propios fragmentos puede suponer una pérdida de consistencia en la experiencia del usuario si no se siguen unas guías comunes entre todos los equipos. Además, no se puede dar soporte nativo a todas las plataformas soportadas porque ello supondría crear un fragmento por cada plataforma. No obstante, el problema más difícil de solucionar es que a veces las capacidades de un servicio no se pueden representar en un fragmento o bien haría falta que en él participasen múltiples servicios.

10. Composición a través de una fachada

Esta aproximación se basa en exponer todos los servicios de back-end en una única interfaz donde además se puedan agregar llamadas que simplifiquen la lógica de la UI. Este patrón es llamado en inglés backends for frontends (BFFs).

Puede suponer un inconveniente si esta nueva capa añade demasiada

lógica que no es la directamente expuesta por los servicios. En su estadio más grave, esto se traduce en una pérdida de la separación entre servicios y su despliegue independiente. Una solución a este problema es, en lugar de exponer una interfaz para todos los servicios, exponer una para cada plataforma o UI.

Parte VII

Integración con servicios de terceros y sistemas legados

Las empresas emplean software comercial disponible en el mercado (en inglés, commercial off-the-shelf software o COTS) y consumen software como servicios (SaaS) que ofrecen poco control sobre el código que venden. Antes de usarlo es bueno plantearse si es mejor que nuestra organización lo construya o lo compre: la mejor respuesta es construir aquello que otorgue a la organización una ventaja estratégica y comprar las herramientas que no.

La integración con este software depende del proveedor, pero nuestro equipo debe tratar de customizarlo, es decir, adaptarlo a las necesidades de la organización. Para tomar control sobre este código se deben hacer las customizaciones de la herramienta sobre una plataforma que el equipo controle y en lugar de referenciar a la herramienta en sí misma, referenciar a la customización hecha.

Pongamos un ejemplo: los sistemas para la gestión de las relaciones con los clientes (Customer Relationship Management, CRM) son herramientas vitales para una organización. El problema es que la toma de decisiones de estos sistemas recae mayoritariamente en el proveedor del software. Para que nuestra organización tome control sobre esto, una posible solución es exponer una fachada para cada entidad del dominio del CRM y hacer que nuestros servicios interactúen con estas fachadas en lugar de con la propia API.

Cada una de las herramientas de terceros con las que integremos pueden emplear diferentes estrategias para integrar. Se debe limitar el número de tipos de integraciones diferentes que hagamos para simplificar el mantenimiento.

Para los sistemas legados se sigue una aproximación similar: como en las servicios de terceros, son sistemas sobre los que no se tiene total control. Un

patrón útil para progresivamente refactorizar el código legado y moverlo hacia microservicios es el Strangler Pattern. Este patrón nos permite capturar llamadas a sistemas legados y decidir si enrutarlas hacia el código legado o una solución nueva.