Capítulo 1: Microservicios

Víctor Iranzo

16 de mayo de 2018

1. ¿Qué son los microservicios?

Los microservicios son servicios pequeños y autónomos que trabajan conjuntamente. Cuanto mayor énfasis se haga en estas dos características de los microservicios a la hora de desarrollarlos, mayores serán los beneficios de usar esta aproximación.

1.1. Pequeños

¿Cómo de pequeño ha de ser un microservicio? En piezas de código muy largas resulta difícil encontrar dónde se debe realizar un cambio o dónde se encuentra un defecto. En todo momento han de cumplirse los principios de cohesión, coherencia y responsabilidad única: el código relacionado debe agruparse conjuntamente porque será modificado por el mismo motivo. Otro factor a tener en cuenta a la hora de establecer los límites de un microservicio es el del tamaño del equipo que lo mantiene: si el tamaño del servicio es demasiado grande para ser gestionado por el equipo, debería ser dividido en partes más pequeñas.

Una regla que puede ser aplicada según el autor Jon Eaves es establecer un tamaño para un microservicio tal que pueda ser completamente reescrito en 2 semanas.

1.2. Autónomos

Un microservicio es una entidad separada: cada uno cambia independientemente del resto y al hacerlo los consumidores no necesiten ser modificados. Para lograrlo, lo más habitual es que cada servicio exponga una interfaz de

programación de aplicaciones (API) y todas las comunicaciones se realicen mediante llamadas a la red.

2. Beneficios de los microservicios

2.1. Heterogeneidad tecnológica

El desarrollo basado en microservicios nos permiten poder escoger una tecnología diferente para cada uno de ellos. De esta manera, escogeremos aquella que mejor se adecue a las necesidades concretas y no una tecnología estándar válida para todos ellos.

Sin embargo, se puede poner límite a esta heterogeneidad. Empresas como Netflix y Twitter emplean mayoritariamente Java Virtual Machine (JVM) como base de sus microservicios combinado con diferentes *stacks tecnológicos* que utilizan según sus necesidades.

2.2. Resilencia

La resilencia es la capacidad de un sistema para seguir funcionando cuando un error ocurre. Si una funcionalidad del sistema falla y este no se propaga en cascada, el error puedo aislarse y el resto del sistema puede continuar funcionando.

En los sistemas distribuidos, hace falta comprender y lidiar nuevas fuentes de errores. Una buena práctica es asumir, por ejemplo, que las redes pueden y van a fallar.

2.3. Escalabilidad

En las arquitecturas monolíticas, para escalar una funcionalidad concreta hace falta escalar el sistema completo. Empleando microservicios, basta con estudiar y escalar solo aquellos que realmente necesitan ser escalados. Esto produce un mejor aprovechamiento de los recursos y un ahorro en los costes.

2.4. Facilidad de despliegue

Un cambio en una línea de código puede suponer el despliegue de la aplicación completa para publicar dicho cambio. Esto se traduce en un aumento

del riesgo a la hora del despliegue. Podría parecer que una solución a este problema fuera hacer un menor número de despliegues. Sin embargo, como la integración continua sugiere, incrementar el número de cambios entre dos publicaciones también supone un gran impacto.

Los microservicios permiten que un cambio en un único servicio se despliegue de manera independiente, haciendo que los cambios lleguen al cliente final de forma más rápida.

2.5. Alineación con la organización del trabajo

Problemas asociados a bases de código inmensas donde colabora un gran equipo pueden ser evitados empleando microservicios. Si el código está repartido entre diferentes componentes, equipos pequeños pueden encargarse de mantener cada una de ellas. De esta manera, la organización del trabajo adquiere un enfoque más ágil al estar formado por equipos independientes y auto-organizados.

2.6. Componibilidad

Si una aplicación se divide en componentes, es más sencillo reutilizar funcionalidad, que puede ser consumida de diferentes formas.

2.7. Reemplazamiento

En la mayoría de empresas abundan los sistemas legados que nadie desea mantener. La refactorización de estos sistemas es imposible por su tamaño y riesgo.

Si en lugar de haber empleado una arquitectura monolítica para su diseño se emplearan microservicios observaríamos como los barreras para la refactorización no existen al poderse reescribir el microservicio al completo en pocos días.

3. Arquitectura orientada a servicios

Las arquitecturas orientadas a servicios (en inglés service-oriented archicture, SOA) son una aproximación de diseño donde múltiples servicios

colaboran para ofrecer una serie de funcionalidades finales. Los microservicios pueden entenderse como una aproximación específica de las arquitecturas SOA. Son una solución para un gran número de problemas de las aplicaciones monolíticas:

- Promueven la reusabilidad del código.
- Son más fáciles de mantener.
- Fomentan buenas prácticas como el principio de responsabilidad única.
- Permiten solventar un mismo problema de diferentes formas.

No obstante, existe una falta de consenso sobre cómo debe ponerse en práctica este tipo de arquitecturas en aspectos como los protocolos de comunicación a emplear o la granularidad de los servicios.

4. Otras técnicas de descomposición

4.1. Librerías

Las librerías son una forma de compartir funcionalidades entre equipos y servicios. Pueden haber sido desarrolladas dentro de la propia organización o por una tercera entidad. Entre las desventajas que suponen su uso frente al de los microservicios se encuentran:

- Pérdida de la heterogeneidad tecnológica: una aplicación que pretenda usar una librería normalmente debe implementarse en la misma plataforma para la que la librería se desarrolló.
- Decremento de la escalabilidad: al depender unas partes del sistema de otras, no se puede escalar una de ellas sin escalar la otra.
- Incapacidad para desplegar de manera independiente: a menos que se empleen librerías dinámicas, no se puede desplegar una librería sin desplegar el sistema completo donde se emplea.
- Aumento del acoplamiento: el código compartido puede suponer que una parte del sistema dependa en gran medida de una librería cuando la funcionalidad que ofrece debería pertenecer íntegramente a esa parte y no a la librería.

4.2. Módulos

Los módulos permite gestionar su propio ciclo de vida sin necesidad de detener el resto de procesos del sistema. Un lenguaje de programación que permite la descomposición en estos elementos es Erlang. Los problemas asociados a este tipo de arquitecturas son similares a los de las librería de código.

Capítulo 2: La arquitectura evolutiva

Víctor Iranzo

18 de mayo de 2018

1. Visión y adaptación de la arquitectura

Con la facilidad de cambio que ofrecen las arquitecturas basadas en microservicios, el rol del arquitecto de software se ve afectado. Su papel principal será el de asegurar la calidad del software y tomar decisiones que ayuden a responder mejor a los cambios, porque el software ha de ser diseñado para ser flexible, adaptarse y evolucionar en función de los requisitos de los usuarios.

Los requisitos en la ingeniería del software cambian más rápidamente que los de otras profesiones. En lugar de centrarse en diseñar un producto final perfecto, el arquitecto debe crear un entorno donde el sistema correcto pueda emerger creciendo progresivamente a medida que se descubren nuevos requisitos.

Una de las responsabilidades del arquitecto de software es la de diseñar el sistema en la que tanto los usuarios como los desarrolladores se sientan cómodos. Para estos últimos, en la solución se debe promover la mantenibilidad, que en la ISO/IEC 25000, conocida como SQuaRE (System and Software Quality Requirements and Evaluation), se divide en las siguientes subcaracterísticas:

- Modularidad: capacidad de un sistema o programa de ordenador (compuesto de componentes discretos) que permite que un cambio en un componente tenga un impacto mínimo en los demás.
- Reusabilidad: capacidad de un activo que permite que sea utilizado en más de un sistema software o en la construcción de otros activos.
- Analizabilidad: facilidad con la que se puede evaluar el impacto de un determinado cambio sobre el resto del software, diagnosticar las

deficiencias o causas de fallos en el software, o identificar las partes a modificar.

- Capacidad para ser modificado: capacidad del producto que permite que sea modificado de forma efectiva y eficiente sin introducir defectos o degradar el desempeño.
- Capacidad para ser probado: facilidad con la que se pueden establecer criterios de prueba para un sistema o componente y con la que se pueden llevar a cabo las pruebas para determinar si se cumplen dichos criterios.

En definitiva, el arquitecto debe garantizar que la visión del sistema, que podríamos definir cómo las características y restricciones a alto nivel del sistema ha desarrollar, sea comprendida tanto por el equipo de desarrollo como por los clientes y evolucione en función de los requisitos de ambos grupos.

2. Colaboración entre microservicios

El arquitecto de software debe preocuparse más por como interaccionan los servicios entre ellos y no tanto en lo que ocurre dentro de los límites de cada uno de ellos. En organizaciones grandes, cada microservicio puede estar desarrollado por un equipo distinto y es el arquitecto quien debe hacer de puente entre ellos.

Como recordamos del capítulo anterior, una de las ventajas de las arquitecturas basadas en microservicios es la heterogeneidad tecnológica. Sin embargo, dejar plena libertad a cada equipo para elegir la tecnología del servicio que va a desarrollar puede traer problemas a la hora de integrarlo con el resto del sistema. En este sentido, una buena práctica sería:

- Establecer normas en aspectos clave como el protocolo de comunicación entre los servicios para facilitar el consumo entre ellos. El uso de interfaces también facilita que el servicio pueda ser consumido por otros.
- Dejar mayor libertad al equipo de desarrollo en otros aspectos como la manera en que se implementa cada servicio de manera particular. De esta forma, cada equipo gana mayor relevancia a la hora de diseñar su propio microservicio y el arquitecto juega un papel menos técnico y más de supervisor y asesor para evitar que se pierda la imagen del sistema completo.

Por último, se debe controlar el manejo de errores entre microservicios para garantizar la autonomía entre ellos. Un servicio que no esté funcionando correctamente no puede influir en el buen funcionamiento de otros servicios.

3. Toma de decisiones e impacto de los cambios

El diseño de software se basa contínuamente en hacer balances entre diferentes alternativas. A la hora de tomar una decisión, cuanta más información se pueda recabar mejor. Conocer el impacto del cambio reducirá el número de posibles errores que puedan surgir como consecuencia de este. El arquitecto es quien mejor debe conocer los principios para tomar una decisión, que se pueden dividir en:

- Objetivos estratégicos: constituyen la dirección que desea tomar una organización a muy alto nivel, sin incluir aspectos a nivel técnico.
- Principios y restricciones: los principos son reglas establecidas para alinear lo que se hace en cada momento con el un objetivo mayor. Cuantos más principios se establezcan en el desarrollo, más fácil será contradecir unos con otros. Las restricciones representan principios inamovibles que se han de seguir obligatoriamente.
- Prácticas: determinan de manera detallada como se llevan a cabo los principios en la práctica. Son de naturaleza más técnica y cambian más frecuentemente que los principios.

4. Monitorización de los microservicios

Dentro de la monitorización de microservicios podemos encontrar las siguientes tareas a llevar a cabo:

- Supervisar la comprensión de la visión del sistema: el conjunto de stakeholders ha de mantener en todo momento la imagen del sistema en todo su extensión y no solo la de un servicio concreto.
- Supervisar la aplicación de normas y prácticas: cuando se establecen unas prácticas que deben ser seguidas por el equipo, se debe garantizar de alguna manera que estas son aplicadas. Dos posibles técnicas para fomentar su seguimiento son:

    - Ejemplos de código: muchos desarrolladores prefieren leer código en vez de documentación. Si se promueve que se imiten ejemplos reales de código que siguen las normas de programación y que han sido verificados, aumentará la coherencia en el estilo del código.
    - Uso de plantillas: la velocidad de desarrollo aumenta si se genera o autogenera parte del código empleando plantillas. Las plantillas contienen llamadas a código que se repite en todos los servicios y que puede haber sido extraído a una librería.

- Hacer balance de la deuda técnica: la deuda técnica es trabajo pendiente que surge fruto de no haber hecho una tarea correctamente cuando se llevó a cabo. Uno de los motivos de la aparición de deuda técnica es el proceso de desarrollo: si la organización se compromete a solucionar un fallo lo más rápido posible o a publicar una nueva funcionalidad en la siguiente versión del producto puede surgir trabajo pendiente fruto de hacer las cosas empleando atajos. El arquitecto es quien debe hacer balance sobre cuando se permite y se debe solucionar la deuda técnica.

Capítulo 3: Modelar microservicios

Víctor Iranzo

19 de mayo de 2018

1. Alta cohesión y bajo acoplamiento

Estos dos términos son claves para implementar servicios de buena calidad:

- Alta cohesión: funcionalidades relacionadas deben agruparse juntas y separadas de otras no relacionadas. De esta manera, cuando se realice un cambio sólo se deberá modificar y desplegar un microservicio, haciendo el despliegue más seguro y coherente. Además, agrupar funcionalidades similares reducirá el número de llamadas potenciales a otros servicios.
- Bajo acoplamiento: en los microservicios poco acoplados un cambio en uno de ellos no requiere cambiar ningún otro. Esta propiedad es clave ya que de otra manera no se puede desplegar de manera independiente un servicio sin cambiar otros que lo consuman. Un servicio debe conocer lo mínimo posible de otros con los que colabore. Un síntoma para detectar que dos servicios están acoplados es que estén en continua comunicación (el término en inglés es chatty communication).

2. Contexto de un microservicio

En el libro 'Domain-Driven Design', Eric Evans explica que el dominio de una solución está compuesta de múltiples contextos bien limitados. Cada contexto está formado por modelos que no necesitan ser compartidos con otros a menos que se defina explícitamente una interfaz que los empleen. La interfaz es el punto de entrada para que otros contextos puedan comunicar con el nuestro, empleando los términos y entidades que en nuestros modelos se definan.

Esta perspectiva puede trasladarse fácilmente al modelado de microservicios. Los contextos limitados de Evans que analicemos en nuestro sistema son firmes candidatos a transformarse en servicios. Así, los límites de un servicio quedan bien limitados porque todas las entidades que pueda requerir se encuentran dentro de sus fronteras, garantizándose su alta cohesión y bajo acoplamiento.

No todos los modelos de un servicio son públicos. Por ejemplo, podemos definir una entidad en el modelo con muchos atributos pero solo hacer públicos al resto de servicios parte de ellos. También puede ocurrir que una misma entidad aparezca en modelos de distintos servicios. En este caso, los atributos de la entidad en cada uno de ellos podrían variar porque a cada uno les interese unas propiedades u otras.

3. Descomposición en microservicios según las capacidades del negocio

Cuando se razona sobre los límites de un servicio no se debe pensar en los datos que este almacena sino en las funcionalidades que ofrece. Pensar en los datos nos conduce a desarrollar únicamente servicios CRUD (en inglés, aquellos que nos permiten las operaciones de crear, leer, actualizar y eliminar datos) que ofrecen unas operaciones muy limitadas. Un servicio ofrece ciertas funcionalidades o capacidades que aportan valor al negocio.

Una descomposición temprana de un sistema en microservicios puede conllevar ciertos riesgos. Si el equipo a cargo del desarrollo tiene pocos conocimientos del dominio del problema a resolver, puede ser buena idea comenzar la implementación como si de un sistema monolítico se tratara. Una vez sean conocidos los límites de cada servicio, se puede refactorizar el código implementado. En este momento, se puede comenzar dividiendo la solución en grandes servicios que poco a poco se vayan dividiendo en más pequeños conforme se estudien las ventajas de hacer cada nueva extracción.

Cuando sea necesario realizar un cambio por nuevos requisitos del negocio, estos se localizarán en un contexto bien delimitado porque existirá una correspondencia entre la estructura de la organización y los microservicios del sistema. Como consecuencia, el tiempo medio para realizar un cambio se verá reducido porque solo hará falta volver a desplegar una porción del sistema.

Además, la comunicación entre microservicios se asemejará a la existente entre las entidades del negocio. Los objetos empleados para dicha comuni-

cación serán los detallados en los modelos compartidos, que son usados en la interfaz de uno de los interlocutores.

4. División vertical y horizontal de un servicio

Cuando un servicio crece demasiado por incluir más funcionalidades, es el momento de dividirlo en otros más pequeños. Las funcionalidades que ofrecía el servicio se reparten entre los nuevos atendiendo a las capacidades del negocio. La división que se debe realizar es en vertical ya que todas las capas del servicio quedan partidas. Cada uno de los nuevos servicios tendrá todas las capas del original pero solo con el código necesario para ofrecer sus funcionalidades.

El otro tipo de división de un servicio es la horizontal. Cuando un servicio es muy consumido puede sufrir problemas de rendimiento. Una posible solución es dividir el sistema de forma horizontal. La división horizontal consiste en añadir una capa intermedia entre dos para reducir el número de accesos que la primera capa hace sobre la segunda. La capa añadida implementa una caché para atendar las peticiones de la capa sobre la que está sin recurrir en todo momento a la capa sobre la que está. Otra posible solución es replicar el servicio de forma geográfica u organizacional atendiendo sus consumidores.

En resumen, a la hora de diseñar microservicios es la primera división la que se debe seguir y únicamente se debe aplicar una división en horizontal por motivos de rendimiento.

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.

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.

Capítulo 6: Despliegue

Víctor Iranzo

28 de mayo de 2018

1. Integración continua

1.1. ¿Qué es la integración continua?

La integración continua (cuyas siglas en inglés son CI) tienen el objetivo de mantener a todos los miembros de un equipo sincronizados con los cambios del resto mediante integraciones de las modificaciones hechas con el código fuente. El proceso de integración del cambio puede pasar fases donde se verifica su correcto funcionamiento, como la ejecución de pruebas o la compilación.

Durante este proceso se crean artefactos sobre los que se ejecutarán las validaciones. Los artefactos construidos deben ser lo más parecidos a los que más tarde se despliegan en la versión de producción. Si se reutilizan estos artefactos, nos aseguraremos de que el artefacto sobre al que se han hecho las pruebas coincide con el que se publica.

Entre los beneficios de la integración continua encontramos: una más rápida retroalimentación sobre la calidad de un cambio, la capacidad de automatización la generación de artefactos o la trazabilidad que existe entre el código y un artefacto (como el código está bajo control de versiones, en cualquier momento se puede volver a construir un artefacto de una versión concreta).

1.2. ¿Estás aplicando realmente CI?

Hay muchas organizaciones que presumen de hacer integración continua cuando realmente no la están haciendo. Jez Humble establece tres preguntas para saberlo:

- ¿Realizas e integras tus cambios diariamente? Comprueba que tus cambios se integran con los del resto o harás que más adelante sea más difícil. Si empleas ramas para implementar una funcionalidad, haz que su tiempo de vida sea breve e integra cuanto antes en la rama principal.
- ¿Tienes una batería de pruebas para validar tus cambios? Si los cambios no se validan, estamos simplemente integrando código que compila pero que tal vez no cumple con su comportamiento esperado.
- Si la compilación falla, ¿arreglarla es la primera prioridad del equipo? Si la compilación no se arregla cuanto antes, el número de cambios a integrar aumentará progresivamente y luego será más costosa su integración.

1.3. Relación entre microservicios y CI

El objetivo principal es poder desplegar un servicio de forma independiente al resto. Teniendo esto en cuenta, ¿cómo podemos relacionar las compilaciones de la integración contínua y el código fuente con cada servicio?

La opción más simple es establecer una sola compilación y un único repositorio para todo el código. Un simple cambio lanzaría la compilación de toda la solución, pero esto puede resultar útil si estoy modificando más de un servicio. Esta solución solo es aconsejable en casos muy específicos como proyectos que acaban de comenzar y con un único equipo. Por lo general, no es aconsejable porque despliega artefactos que no son necesarios, aumentando el coste y tiempo de cada compilación.

Otra aproximación es tener un único repositorio para todos los servicios y diferentes compilaciones para cada servicio. El problema de esta aproximación es que puede conducirnos a la mala práctica de hacer cambios que modifiquen código de diferentes servicios, que se puede traducir en aumentar el acoplamiento entre ellos.

La solución más idónea pasa por tener una compilación y un repositorio diferente para cada servicio. De esta forma se evita juntar en uno cambios que afecten a diferentes servicios y se genera un único artefacto por compilación (con la ejecución de sus respectivas pruebas). También hay una mayor correspondencia entre los servicios y los equipos encargados de cada uno porque cada equipo realiza cambios en un único repositorio.

1.4. Entrega continua

Se pueden obtener mayores beneficios de la integración continua a través de las compilaciones por fases o build pipelines. En este tipo de compilaciones se separa por pasos el proceso: la ejecución de las pruebas unitarias, la de las pruebas de rendimiento... El beneficio de hacer esto es que se obtiene una retroalimentación más rápida si se ejecutan de forma separada acciones rápidas de otras más pesadas.

Desde el punto de vista de la pipeline se puede observar como el artefacto generado para la compilación pasa de un paso al siguiente, lo que se traduce en que el código está más cerca de poderse llevar a producción. Los pasos de la pipeline pueden ser manuales, como las pruebas de aceptación de un usuario, o estar automatizados, como la ejecución de tests unitarios. Si nos situamos en el contexto de los microservicios, querremos tener una pipeline para cada servicio con el objetivo de poder desplegarlos de forma independiente.

La entrega continua se basa en dos ideas: la primera, poder obtener feedback de todos y cada uno de los cambios realizados a través de su paso por el pipeline, y la segunda, que es poder tratar a cada uno de los cambios como candidatos a salir a producción. Entre las ventajas de su uso destaca que se hace más tangible el proceso de calidad y publicación del software y que se reduce el tiempo entre versiones de producción porque se puede publicar una por cada cambio hecho.

2. Artefactos

2.1. Artefactos específicos de la plataforma

Algunas tecnología tienen artefactos propios como son los archivos JAR en Java. Desde el punto de vista de los microservicios, estos artefactos no son suficientes para levantar uno de ellos. Necesitamos instalar y configurar más software para el despliegue del servicio y algunos frameworks como Puppet o Chef pueden ayudar.

Como su nombre indica, estos artefactos son específicos de una pila tecnológica y esto puede hacer que el despliegue de un sistema basado en servicios sea más costoso si se emplean en cada uno tecnologías distintas. Para facilitarlo, podemos apoyarnos en las herramientas que hemos citado antes porque soportan diferentes tecnologías.

2.2. Artefactos del sistema operativo

Una forma de evitar los problemas asociados a artefactos específicos de la tecnología empleada es emplear los del sistema operativo. Por ejemplo, en Ubuntu se puede emplear un paquete DEB o en Windows un MSI. Usar estos paquetes nos puede ahorrar trabajo, que de otra forma se haría con frameworks como Puppet o Chef, porque se delega en el gesto de paquetes del SO.

El principal problema es si necesitamos desplegar nuestro sistema en diferentes sistemas operativos. Si esto pasa, haría falta un mayor esfuerzo para producir los paquetes en cada SO. Si no es el caso, esta aproximación puede simplificar mucho el despliegue y evitar complejos scripts para esta tarea.

2.3. Imágenes propias

Uno de los principales problemas en herramientas como Puppet, Chef o Ansible es el tiempo que tardan en ejecutarse scripts de configuración. Estas herramientas son capaces de detectar si el software que necesitan instalar está ya presente en la máquina donde corren.

Puede parecer razonable mantener una máquina durante mucho tiempo para reducir el tiempo de despliegue, ya que al hacerlo el software necesario no haría falta volverlo a instalar. Sin embargo, no es una buena práctica alargar la vida de estas máquinas porque sino los cambios en la configuración del despliegue no se aplicarían. En definitiva, estas herramientas pueden suponer un problema si necesitamos dar retroalimentación rápidamente sobre un artefacto.

Para solucionarlo, una aproximación válida es, en lugar de hacer un script para instalar el software del que depende el servicio en cada despliegue, restaurar una imagen con todo el software necesario ya instalado. Todas las plataformas de virtualización permiten construir imágenes propias. El despliegue se simplifica a simplemente restaurar la imagen con el software necesario e instalar la versión del servicio a desplegar.

Aunque nos ahorremos tiempo durante el despliegue, construir una imagen puede llevarnos largo tiempo. Además, si depende de mucho software para funcionar, la imagen puede ser de un tamaño considerable. Por estos motivos, la tecnología que más se emplea es la basada en contenedores.

2.3.1. Packer

Una herramienta que puede resultar de utilidad a la hora de crear imágenes es Packer. Con Packer, empleando scripts de configuración de diferentes tecnologías (Chef, Ansible, Puppet...) se pueden crear imagénes para diferentes plataformas (VMWare, AWS, Vagrant, Linux...).

2.3.2. Imágenes como artefactos

Organizaciones como Netflix van más allá y construyen la imagen durante la compilación con el servicio ya instalado en ella. De esta forma, el despliegue es más rápido porque en la imagen ya está instalado el servicio.

3. Entornos y configuración de servicios

A medida que nuestro artefacto pasa de una etapa del pipeline a otra, va a desplegarse en diferentes entornos, por ejemplo, uno para pruebas unitarias, otro para pruebas de aceptación... Cada entorno se emplea con una finalidad concreta, pero la diferencia entre los entornos puede traer problemas si por ejemplo se producen errores con una configuración y no con otra.

A medida que el cambio avanza en la pipeline, queremos que el entorno donde se hacen las validaciones sea más parecido a uno de producción. Así podremos capturar lo antes posibles problemas asociados a diferencias entre entornos.

Algunos entornos pueden requerir de ficheros de configuración adicionales. Idealmente, estos no deberían existir para reducir las diferencias entre los entornos. Para añadir esta configuración adicional al artefacto podemos seguir dos aproximaciones: construir un artefacto diferente para cada entorno o construir un único artefacto y gestionar la configuración de forma separada.

En cuanto a la primera, rompe con uno de los conceptos clave de la entrega continua ya que no todos los artefactos construidos podrían ser candidatos para ser desplegados. Además, no podríamos saber si las validaciones hechas sobre uno de los artefactos se cumplen también en el que luego saliera a producción.

La segunda aproximación reduce las diferencias entre los entornos donde se instala el artefacto, por lo que es una mejor opción. En caso de que la

configuración sea muy extensa, en lugar de almacenar la configuración a través de ficheros se puede emplear un servicio especial para proveerla.

3.1. Definición del entorno

La definición de los entornos consiste en establecer los recursos de una máquina que se asociarán a un servicio concreto. Entre los recursos que se pueden asociar están los asociados al hardware, como recursos de computo, memoria y red, credenciales, configuraciones regionales, etc. Entre las formas para definir un entorno podemos mencionar los archivos YAML o los Puppet.

Construir un sistema puede requerir la configuración de muchos entornos si el sistema a desplegar es muy complejo. Algunas herramientas como Terraform pueden facilitarnos este tabajo, aunque todavía están en una fase muy madura. Entre sus capacidades, Terraform nos permite desplegar en diferentes plataformas de forma automatizada.

3.2. Interfaz para el despliegue

Es vital tener una interfaz uniforme para lanzar el despliegue de un servicio bajo demando. La manera más sencilla de implementar esto es a través de una llamada por línea de comandos parametrizada con variables como el entorno donde se desea desplegar o la versión del servicio.

Para desarrollar el script que desplegará el servicio atendiendo a esta variable se pueden usar diferentes herramientas: Fabric (empleando Python), Pair (en AWS) o PowerShell (en Windows).

4. Alojamiento de servicios

Una pregunta habitual es cuántos servicios se deben desplegar por máquina. Vamos a introducir el concepto de host como una unidad genérica de aislamiento, un sistema operativo donde se pueden instalar y ejecutar servicios. Si desplegamos directamente sobre máquinas, entonces un host será el equivalente a una de ellas. En cambio, si empleamos virtualización, cada máquina puede contener más de un host que a su vez pueden contener uno o más servicios.

4.1. Múltiples servicios por host

Desde el punto de vista de gestionar servicios, es una aproximación simple y poco costosa al emplear un menor número de máquinas. Además, puede facilitar el trabajo de los desarrolladores. Sin embargo, cuenta con algunas desventajas:

- La monitorización resulta más difícil: responder preguntas como la CPU que emplea cada servicio puede ser difícil porque todos se encuentran en la misma máquina.
- Posibles efectos secundarios: un servicio que atiende muchas peticiones puede acaparar recursos que son necesarios para el resto de microservicios allí alojados.
- Un único punto de fallo: el fallo del host donde se alojan los servicios puede afectar muy negativamente al sistema porque todos sus servicios dejan de estar disponibles.
- Despliegue más complejo: el despliegue de un nuevo servicio no debe afectar a los ya alojadas. Cada uno de ellos puede depender de software distinto y no podemos controlar completamente los efectos de su instalación. Además, es importante mantener la idea de que cada servicio ha de desplegarse de forma independiente.
- Reduce la autonomía de los equipos: si cada servicio lo gestiona un equipo, la gestión del host donde estos se alojan requiere la colaboración de todos los involucrados.
- No se pueden emplear algunos artefactos: no se pueden emplear artefactos como imágenes para el despliegue de cada servicio porque distintos servicios se ejecutan en el mismo sistema operativo.
- Necesidades distintas se tratan de la misma manera: en un mismo los host, necesidades como la escalabilidad han de ser tratadas igual para todos los servicios porque todos ellos comparten una única configuración del host.

4.2. Contenedores de aplicaciones

Es un caso particular del modelo de múltiples servicios por host pero con ciertos beneficios para la gestión de los servicios alojados como herramientas para la monitorización. Algunos ejemplos de este modelo son IIS en aplicaciones .NET o contenedores servlet en Java.

Entre las desventajas de emplearlos encontramos las siguientes:

- Limitan la tecnología que se puede emplear: en los ejemplos que hemos puesto, a .NET o Java.
- Ralentizan el tiempo de ejecución: esto puede afectar a la retroalimentación que el desarrollador tiene del servicio que implementa.
- El uso de recursos es más complejo: en un mismo proceso se ejecutan las diversas aplicaciones como hilos.

En definitiva, es mucho mejor modelo emplear contenedores independientes para cada microservicio y tecnologías como las que hemos citado ya están recorriendo ese camino con frameworks como Nancy o Jetty.

4.3. Un servicio por host

Con este modelo se remedian los problemas de monitorización, efectos secundarios entre servicios y único punto de fallo que caracterizaban al modelo de más de un servicio por host. Además, los servicios pueden escalar independientemente de otros y la seguridad se aplica más fácilmente a nivel de host según las necesidades de cada servicio.

Como consecuencia de esta aproximación tendremos que mantener más hosts, aumentando el coste de su gestión. Aún así, a la hora de razonar resulta más sencillo tener un único servicio por host.

4.4. Plataforma como servicio (PaaS)

La plataforma como servicio (PaaS) es un nivel de abstracción más alto que el de un host. Se basa en delegar el aprovisionamiento y ejecución de un artefacto a otra organización, que también se encarga de escalar el sistema según la carga de cada nodo. Heroku es una de estas plataformas.

Cuando estas soluciones no funcionan correctamente en un servicio desplegado, es difícil tomar el control para tratar de solucionarlo. Por eso, el artefacto a desplegar es muy sensible a cómo está implementado, y cuanto más se salga del estándar que la plataforma espera, más probabilidad hay de que no funcione correctamente.

5. Automatización del despliegue

Cuando el número de máquinas a gestionar es pequeño, es sencillo gestionarlas de forma manual. Pero cuando el número es alto, es recomendable automatizar el despliegue de los servicios, sobretodo si se sigue un modelo de un servicio por host. De esta manera, el aumentar el número de servicios del sistema no hará que aumente proporcionalmente el coste de su mantenimiento.

La automatización repercutirá positivamente en la productividad de los desarrolladores y el tiempo ganado se podrá invertir en desarrollar nuevos servicios.

6. Tecnologías para el despliegue

6.1. Virtualización tradicional

Tener muchos hosts es costoso si se cada host consiste en una máquina. Este coste se reduce mediante la virtualización, que nos permite dividir una máquina en hosts separados donde pueden ejecutarse servicios distintos. Sin embargo, no podemos dividir una máquina en tantas piezas como queramos porque la separación de los hosts supone un coste.

La virtualización más usada tradicionalmente para este propósito es la de tipo 2, que es la implementada por plataformas como AWS o VMWare. En ella, sobre el sistema operativo de la máquina se ejecuta el hipervisor. El hipervisor se encarga de repartir los recursos de la máquina como la CPU o la RAM entre los distintos hosts virtualizados y permite al usuario la gestión de las máquinas virtuales. Cada una de las máquinas virtuales puede ejecutar su propio sistema operativo y kernel. Cada una de las máquinas virtuales se puede considerar aislada de la máquina física y de las otras máquinas virtuales.

Uno de los problemas de esta aproximación es que el hipervisor también necesita consumir recursos, y cuantas más son las máquinas que debe gestionar, mayor será su consumo.

6.2. Vagrant

Vagrant es una plataforma de despliegue que se emplea principalmente durante las fases de desarrollo y pruebas de un sistema. Se basa en la virtua-

lización tradicional a través de archivos de texto donde indica las máquinas e imágenes a restaurar y como están conectadas entre ellas. Ofrece un gran número de funcionalidades como asociar una máquina a un directorio local de la máquina sobre la que se ejecuta.

No obstante, la ejecución un un gran número de máquinas virtuales puede sobrecargar los recursos de la máquina física donde se ejecutan. Por este motivo no suele emplearse en fases de producción y en las fases de prueba suelen emplearse stubs para evitar tener que desplegar todas las dependencias de un servicio bajo pruebas.

6.3. Contenedores Linux

Para los usuarios de Linux existe una alternativa a la virtualización. Los contenedores Linux (LXC) permite crear un espacio de procesos separado en el que puede ejecutarse de forma aislada un servicio. Cada contenedor es un subárbol del principal y puede tener asociado recursos físicos gestionados por el kernel.

En los contenedores Linux no encontramos un gestor como el hipervisor: los contenedores pueden ejecutar su propio sistema operativo y comparten el kernel de la máquina física. Aparte del ahorro en recursos que supone no tener este middleware, también se reduce el tiempo de aprovisionamiento porque cada contenedor requiere de menos recursos. Esto influye también en la retroalimentación sobre el funcionamiento de un servicio porque su despliegue será más rápido.

El número de contenedores que una máquina puede alojar es mayor que el número de máquinas virtuales debido a la naturaleza ligera de estos. Por esta razón el coste de esta solución es menor. Para obtener las ventajas de ambas aproximaciones se puede seguir una versión híbrida: se puede obtener una máquina virtual de una plataforma con AWS que nos asegure la escalabilidad bajo demanda y sobre ella ejecutar contenedores LXC.

Sin embargo, el grado de aislamiento de los contenedores no es perfecto ya que existen formas en las que uno puede interferir en el funcionamiento de otro debido a problemas de diseño y bugs conocidos. La configuración de los contenedores tampoco es trivial. Es necesario implementar un mecanismo que en la virtualización tradicional hace el hipervisor para enrutar las peticiones recibidas a cada contenedor.

6.4. Docker

Docker es una plataforma basada en contenedores que nos permite la creación y despliegue de aplicaciones de forma similar a las imágenes en las máquinas virtuales. Docker gestiona el aprovisionamiento de los contenedores, problemas en la red, la versión de las aplicaciones... Además, oculta la implementación de los servicios que envuelve.

En esta tecnología se ha invertido con el objetivo de hacerla lo más óptima posible. En este sentido, existen sistemas operativos desarrollados específicamente para Docker como CoreOS con el objetivo de incluir en una imagen solo las funcionalidades necesarias y reducir el consumo de recursos.

Sin embargo, Docker no resuelve todos los problemas relativos al despliegue. Existen varias tecnologías que amplían sus capacidades, como por ejemplo Deis o Kubernetes, desarrolada por Google. Esta última nos ofrece una capa adicional para solicitar y ejecutar contenedores bajo demanda. Esta aproximación se puede situar entre las soluciones de infraestructura como servicio (IaaS) y plataforma como servicio (PaaS) y se suele catalagorar como contenedores como servicio (CaaS).

Capítulo 7: Pruebas

Víctor Iranzo

26 de julio de 2018

1. Tipos de pruebas

El siguiente esquema es una modificación del cuadrante de Marick empleada para catalogar pruebas en distintos tipos:

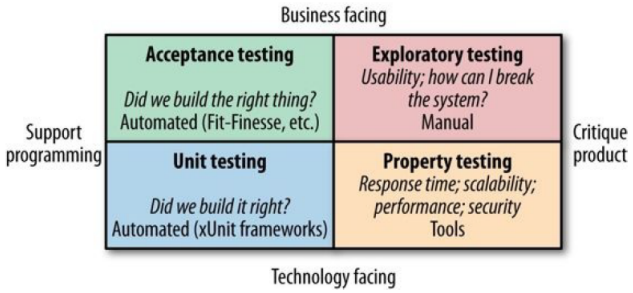

Figura 1: Modificación del cuadrante de Marick hecha por L.Crispin y J.Gregory en su libro Agile Testing.

Las pruebas situadas en la mitad inferior están relacionadas con la tecnología empleada, como son las pruebas unitarias o las de rendimiento, que deben ser implementadas por los desarrolladores y que se pueden automatizar. Las pruebas situadas en la mitad superior están orientadas a que los stakeholders conozcan cómo funciona el sistema y que no guardan relación con aspectos tecnológicos. Entran dentro de esta mitad las pruebas de aceptación o las pruebas manuales.

La proporción necesaria de cada tipo de test depende del sistema a desarrollar. La tendencia actual es en favor de los tests de pequeña escala y automatizados. En un sistema basado en microservicios, son esenciales las pruebas de este tipo ya que validarán que no se despliegue a producción código defectuoso.

Podemos clasificar las pruebas en los microservicios en 3 tipos según su alcance: unitarias, de servicios y de extremo a extremo. Estos términos pueden llevar a discusión: por ejemplo, en una prueba unitarias se puede incluir para algunas personas la validación de diferentes clases o funciones mientras que para otras no.

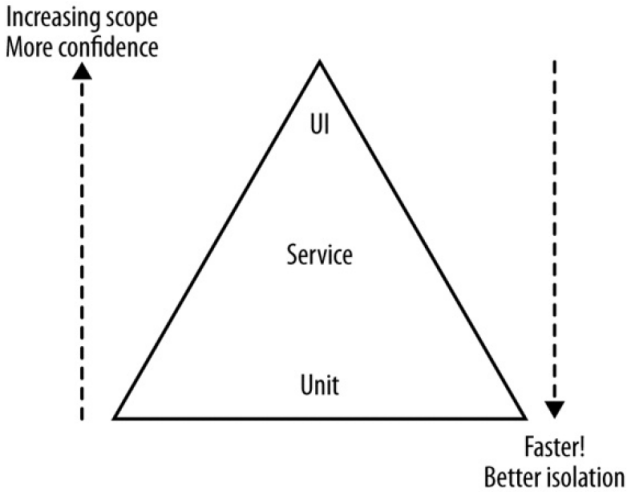

Figura 2: Pirámide de pruebas diseñada por Mike Cohn en su libro Succeding with Agile.

1.1. Pruebas unitarias

Son pruebas que validan una única función. Se pueden generar fruto de procesos de desarrollo como el test-driven development (TDD). En general, se prefiere tener un gran número de este tipo de pruebas por su rapidez, porque pueden ayudar a la refactorización del código y porque es donde mayor cantidad de defectos se suele capturar.

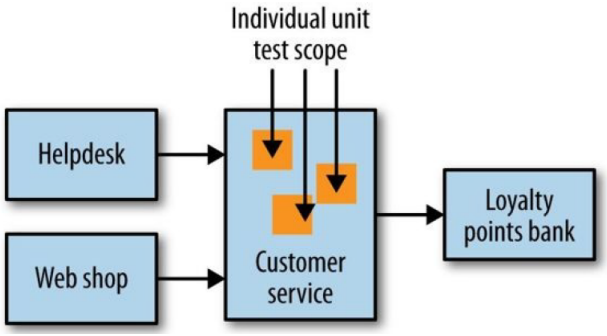

Figura 3: En las pruebas unitarias no se ejecuta ningún servicio, sino que se prueban métodos específicos de ellos.

1.2. Pruebas de servicios

En estas pruebas se verifica la funcionalidad de los servicios sin emplear las interfaces de usuario. Cada prueba verifica una de las funcionalidades que el servicio expone. Se pretende comprobar la separación entre los servicios y para solventar las dependencias que el servicio bajo pruebas tiene sobre otros, se reemplazan los servicios colaboradores por stubs o fakes.

Los stubs y los fakes imitan el comportamiento real del servicio al que representan. Los stubs contienen menos lógica que los fakes y ante una petición responden siempre de la misma manera. Por su parte, los fakes pueden emplearse para comprobar que los efectos secundarios de la interacción entre los servicios ocurre. Por ejemplo, si al invocar a un servicio el comportamiento esperado es que este añada una tupla en base de datos, un stub podría responder simplemente con un 200 OK mientras que un fake tendría más lógica para reproducir esto.

Este tipo de pruebas pueden ser igual de rápidas que las unitarias siempre que no se tengan que emplear un gran número de infraestructuras como bases de datos o redes.

1.3. Pruebas de extremo a extremo

Son pruebas que se ejecutan sobre todo el sistema. Cubren gran parte de código, por lo que su correcta ejecución dan mucho grado de confianza. En su ejecución se levantan varios servicios diferentes.

De los servicios que se prueban, ¿qué versión se debe emplear de cada uno? ¿Quién debe poder ejecutar esta prueba? Cada vez que se realice un

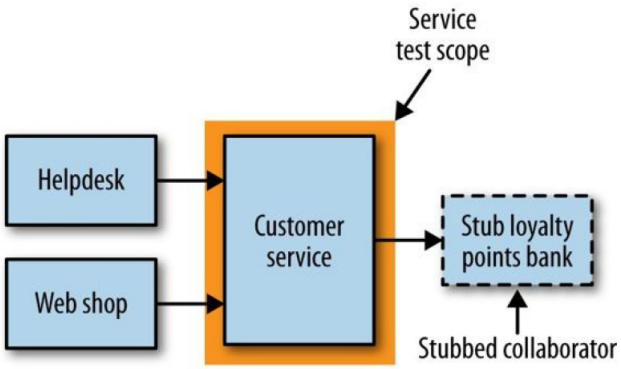

Figura 4: Las pruebas de servicio se ejecutan sobre un único servicio y emplean fakes para reemplazar a los colaboradores de este.

cambio en alguno de los participantes deberían ejecutarse todas las pruebas en las que este se emplea. Una manera elegante de conseguir esto es hacer que en la compilación de la integración continua, la fase de pruebas de extremo a extremo sea compartida entre todos los involucrados para que las pruebas se ejecuten cada vez que cambie uno de ellos y la versión a probar de todos sea la asociada al cambio.

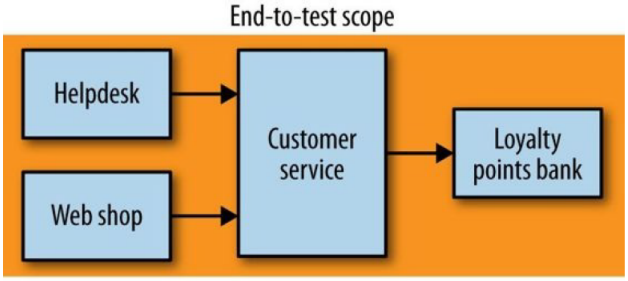

Figura 5: Las pruebas de extremo a extremo cubren una funcionalidad del sistema ejecutando los servicios que se involucran para ofrecerla.

1.3.1. Desventajas de las pruebas de extremo a extremo

- Pruebas frágiles: conforme el alcance del test aumenta mayor son las partes bajo prueba sobre las que no se puede tener control. Estas partes pueden introducir fallos que no demuestran que la funcionalidad tenga un defecto, sino fallan porque otro problema ha ocurrido como puede ser un fallo en la red. Las pruebas así son menos deterministas y el desarrollador puede acostumbrarse a que fallen si consigue hacerlas pasar al ejecutarlas por segunda vez, lo que se conoce como normalizar

la desviación. Este tipo de pruebas han de ser eliminadas cuanto antes de la suite de pruebas y refactorizadas en pruebas de menor alcance.

- Responsable de las pruebas: si los servicios que participan en las pruebas son mantenidos por distintos equipos, es necesario que todos ellos se hagan responsables de la suite de pruebas. Diferentes situaciones pueden ocurrir sino:
  - Explosión de pruebas: si nadie controla la salud de la suite, los equipos pueden añadir pruebas descontroladamente que no aporten valor.
  - Ignorar resultados de las pruebas: cuando una prueba falla, nadie trata de arreglarla porque cree que la responsabilidad la tiene otro.
  - Equipos especializados en pruebas: esto puede alargar el tiempo de desarrollo de una funcionalidad por requerir mayor coordinación entre equipos. Además, con esta metodología los desarrolladores se involucran menos en las pruebas, haciendo su código más difícil de probar e ignorando los resultados de las pruebas.
- Tiempo de ejecución mayor: al levantar un mayor número de servicios y cubrir más código se alarga el tiempo de ejecución de la prueba. Esto puede ser sobrellevado a través de la ejecución en paralelo de pruebas. Esto combinado con la fragilidad de los tests puede desencadenar que encontrar la causa del fallo de una prueba requiera de un mayor esfuerzo.
- Mayor tiempo de retroalimentación: si las pruebas tardan más en ejecutarse, el desarrollador tardará más tiempo en darse cuenta de que en los cambios realizados ha introducido un defecto. Este hecho puede repercutir en el resto del equipo, ya que si la compilación está rota debido a una prueba que no pasa, los cambios que otros suben pueden apilarse, dificultando su futura integración.

2. Balance de pruebas a realizar

Si volvemos a la pirámide de Cohn, uno de los conceptos clave es que a medida que aumenta el alcance las pruebas lo hace el nivel de confianza que las pruebas dan sobre la ausencia de defectos. Por otro lado, cuanto más arriba en la pirámide más tiempo tardará una prueba en ejecutarse y dar retroalimentación al desarrollador. Además, determinar el motivo de fallo de una prueba será más costoso cuanto mayor sean las líneas de código probadas.

El número de pruebas que se aconseja tener de cada tipo aumenta conforme descendemos por la pirámide. Un antipatrón sería seguir como convención la misma pirámide invertida. En este caso, nuestra cobertura de pruebas se basaría en pruebas de largo alcance y larga ejecución, nada deseable según los principios de integración continua.

En cuanto a las pruebas de extremo a extremo, no podemos añadir una prueba de este tipo por cada nueva funcionalidad que el sistema ofrezca. El número de servicios que participan para ofrecer una funcionalidad puede ser muy alto y en una prueba no se deberían de levantar más de 3 o 4 servicios para no potenciar las desventajas que hemos mencionado. Si es necesario, se pueden emplear fakes o stubs para no levantar a todos los servicios en la prueba.

3. Pruebas dirigidas por el consumidor

Durante las pruebas queremos asegurarnos que desplegar un cambio en un microservicio no rompe ninguno de sus consumidores. Esto se puede verificar mediante contratos o consumer-driven contract (CDC). Un contrato representa las expectativas que un consumidor tiene sobre un servicio. La manera de asegurarnos de que no se va a romper ningún consumidor es no desplegando cambios que puedan romper el contrato entre ambos.

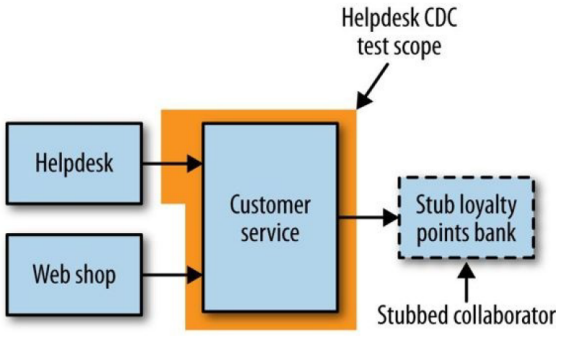

Figura 6: Dentro de la pirámide de Cohn, las pruebas dirigidas por el consumidor se sitúan al mismo nivel que las pruebas de servicio.

Para la elaboración de un contrato se puede contar con el equipo del servicio que va a consumir este. La colaboración entre ambos equipos también es fundamental si es necesario hacer un cambio que rompe el contrato para que el consumidor este preparado. En casos donde el servicio con el que se integre sea de terceros, la comunicación cuando se rompe un contrato no podrá ser tan frecuente como la deseada.

4. Pruebas antes de producción

Muchas organizaciones invierten en pruebas durante los días previos a publicar una versión a producción. Los usuarios usan nuestro sistema en formas que no podemos preveer, por lo que no siempre podemos detectar todos los errores de nuestro código. La reacción habitual cuando se produce un fallo es añadir más pruebas para evitar que vuelva a ocurrir. Sin embargo, es imposible garantizar una aplicación real libre de fallos.

Una forma de capturar más errores es extender los pasos del despliegue donde tradicionalmente las pruebas se ejecutan. Un ejemplo es una suite de pruebas humo (en inglés, smoke tests), una colección de pruebas de rápida ejecución que se ejecuta en cada nuevo componente desplegado para garantizar que el despliegue ha ido correctamente. Otras aproximaciones son los despliegues azul y verde y las canary releases.

4.1. Despliegues azul y verde

Este proceso consiste en contar con dos copias del software desplegadas al mismo tiempo, aunque solo una de ellas recibe realmente peticiones. Cuando se despliega una nueva versión de un servicio, se realizan pruebas de humo sobre la nueva versión y una vez se verifica que las pruebas pasan, se redirige el tráfico de la vieja versión a la nueva. La vieja versión se mantiene desplegada pero sin aceptar peticiones por si se tiene que redirigir de nuevo el tráfico debido a un error en la nueva versión.

Para seguir esta aproximación se debe tener la infraestructura suficiente para dirigir el tráfico hacia diferentes hosts y tener los suficinetes nodos para ejecutar diferentes versiones de un mismo servicio. Como ventajas encontramos que reduce el riesgo del despliegue porque nos permite revertir un cambio si se detecta un problema, proceso que se puede automatizar para mayor eficacia.

4.2. Canary releases

Empleando esta aproximación verificamos la nueva versión de un servicio, dirigiendo hacia él algunas de las peticiones que se realizan a la versión en producción para ver si se comporta de la manera esperada. El comportamiento esperado se puede medir a través de métricas como el ratio de errores. El mayor reto de este modelo es el enrutamiento del tráfico a una u otra versión.

Si la nueva versión no funciona correctamente, el cambio se puede revertir rápidamente. Sino, se puede incrementar progresivamente el tráfico real que atiende. Netflix emplea esta aproximación en muchos servicios.

Existen dos vertientes posibles: dirigir parte del tráfico real a la nueva versión o duplicar el tráfico que el servicio recibe, de tal forma que la nueva versión recibe peticiones pero sus respuestas no son vistas por el invocador.

4.3. Tiempo medio entre fallos y tiempo medio entre reparaciones

Muchas organizaciones invierten mucho esfuerzo en producir un gran número de pruebas funcionales y ningún esfuerzo en mecanismos para monitorizar o revertir el sistema cuando un fallo ocurre.

Se debe hacer un balance entre optimizar la media de tiempo entre fallos (MTBF) y la media de tiempo de reparación (MTTR) para alcanzar un equilibro. Mientras que el primero puede reducirse aunmentando el número de pruebas, el segundo puede mejorar con las técnicas que hemos citado antes y buenos sistemas de monitorización.

5. Pruebas de requisitos no funcionales

Los requisitos no funcionales representan aquellas características que el sistema exhibe pero que no pueden contabilizarse como una funcionalidad. Las pruebas que sobre estas características se hacen encajan en el cuadrante de Marick de pruebas de propiedades.

Su implementación puede hacerse de forma unitaria o de extremo a extremo, pero se aconseja seguir la relación de la pirámide de Cohn sobre el alcance y número de pruebas.

5.1. Pruebas de rendimiento

Las pruebas de rendimiento pertenecen al grupo de pruebas de requisitos no funcionales. Este tipo de pruebas es más importante en los sistemas basados en microservicios que en los monolíticos porque las integraciones entre los servicios (por ejemplo, invocaciones HTTP) pueden introducir un sobrecoste en el rendimiento si hay muchos servicios colaborando.

En estas pruebas se busca simular un entorno lo más cercano posible al

de producción, por lo que puede resultar útil contar con diferentes máquinas con las que ejecutar las pruebas. Debido al tiempo que pueden tardar en ejecutarse, no suele incluirse un estado para este tipo de pruebas en la compilación de cada cambio. En su lugar, suelen ejecutarse periódicamente. Esto puede producir que no se observen los resultados de las pruebas y esto puede resultar un problema si se pierde trazabilidad hacia el cambio que empeoró el rendimiento.

Capítulo 9: Seguridad

Víctor Iranzo

30 de mayo de 2018

1. Autenticación y autorización

La autenticación es el proceso por el cual confirmamos que una entidad es quien dice ser. Generalmente, un usuario se autentica mediante un nombre de usuario y una contraseña.

La autorización es el mecanismo por el cual relacionamos una entidad autenticada con las acciones que puede llevar a cabo.

En las aplicaciones basadas en microservicios hace falta definir un mecanismo para que los usuarios no tengan que autenticarse en cada uno de los microservicios de forma separada.

1.1. Proveedor de identidades

Una aproximación para ofrecer estas características son las soluciones single sing-on (SSO). Entre ellas podemos mencionar SAML y OpenID Connect. SAML es un estándar basado en SOAP con el que es bastante complejo trabajar. OpenID Connect es un estándar que surge como una implementación específica de OAuth 2.0 que imita la manera en que organizaciones como Google gestionan sus soluciones SSO. Por su facilidad de uso, se espera que OpenID Connect sea en un futuro acogido por cada vez más organizaciones.

Cuando una entidad trata de acceder a un recurso o realizar una operación, es redirigido a un proveedor de identidades para que se autentique. Una vez autenticados, el proveedor de identidades responderá al servicio que lo ha invocado indicando los permisos que tiene la entidad sobre la operación o recurso solicitados.

El proveedor de identidades puede ser un servicio externo, como el de Google. Sin embargo, la mayoría de empresas emplear su propio servicio

de directorio (similar a Active Directory) donde se almacena información sobre las entidades conocidas, sus roles y permisos. Los permisos efectivos para un microservicio no se pueden almacenar de forma centralizada ya que esta información pertenece al servicio en si y puede suponer un punto de acoplamiento. Además, los roles que se modelan deben ser lo más similares a los existentes en el mundo real donde se ejecuta el sistema.

Para centralizar el contacto entre los servicios del sistema y el proveedor de identidades se puede emplear un SSO gateway. Este mecanismo consiste en un proxy situado entre los servicios y el mundo exterior, lo que puede suponer un único punto de fallo. Además, al ser una capa intermedia puede tender a ir aumentando progresivamente en las funcionalidades que ofrece hasta convertirse en un punto de acoplamiento.

2. Seguridad en tránsito

2.1. HTTP y HTTPS

No sólo se deben autenticar los usuarios para realizar una acción, otros servicios también se pueden modelar para saber si se les da acceso a un recurso o funcionalidad.

Una posible opción es hacer que cualquier invocación a un servicio hecha desde el propio servicio se considera autorizada. Esta solución puede ser peligrosa ante atacantes que penetren en la red, pero puede ser adecuada si los datos almacenados no son de alta sensibilidad.

La autorización HTTP permite a un cliente enviar un nombre de usuario y una contraseña en la cabecera del mensaje. Aunque es un protocolo muy extendido, puede ser peligroso porque estos datos son enviados sin ninguna seguridad. Por este motivo se suele emplear el protocolo de HTTPS.

Entre los problemas de usar HTTPS están que el tráfico en este protocolo no puede ser capturado por proxies inversos, aunque en caso de necesidad se puede capturar en alguno de los extremos o en un balanceador de carga. Además, se deben gestionar los certificados SSL, que puede resultar un problema cuando el sistema se distribuye entre múltiples máquinas, y se puede aumentar el tiempo de entrega de los mensajes.

2.2. Certificados de clientes

Con esta aproximación cada cliente emplea un certificado en su interacción con el servidor. Esta solución es aconsejable si la sensibilidad de los datos que se envían es muy alta. Sin embargo, puede traer dificultades a la hora de revocar y emitir certificados.

2.3. Códigos de autentificación de mensajes en clave-hash (HMAC)

El tráfico a través de HTTPS puede afectar al rendimiento del sistema. Una solución a este problema pasa por, en lugar de usar este protocolo, usar mensajes cifrados con claves hash y enviarlos a través de HTTP.

Con HMAC, el cuerpo de una petición es convertido en un hash empleado una llave privada y es enviado junto con la petición. El servidor, cuando recibe una petición, usa la misma llave privada que comparten para generar el hash con el cuerpo del mensaje recibido y compararlo con el hash del mensaje.

Este mecanismo cuenta con 3 desventajas. Primero, entre servidor y cliente tiene que existir un secreto compartido, que debe ser enviada usando un protocolo seguro. Segundo, este mecanismo no es un estándar y existen muchas implementaciones distintas disponibles que deben ser evaluadas. Tercero, un atacante pueda interceptar y leer una petición, pero no puede modificar esta porque si lo hace el hash que genera el servidor y el de la petición no coincidirían.

2.4. Claves API

Muchas APIs públicas emplean este mecanismo para ofrecer sus servicios a terceros. Se pueden emplear para identificar quién hace una llamada y limitar las que puede hacer para que no sobrecargue el servicio. Entre sus ventajas podemos nombrar su facilidad de uso.

2.5. Problema del reemplazo

Una entidad autenticada puede invocar a un servicio que a su vez tenga que invocar a un tercero. En esta segunda petición, si aceptamos la aproximación de que cualquier petición hecha desde dentro del sistema es válida, podemos incurrir en un problema de reemplazo de identidades.

Por ejemplo, un atacante que haya conseguido acceso a la red puede obligar a un servicio a hacer peticiones a otro y obtener los datos del servicio atacado. Este problema es grave de acuerdo a la sensibilidad de los datos. Una posible solución es solicitar que en cada petición sean provistas las credenciales de la entidad que comienza la interacción.

3. Seguridad en los datos

La encriptación de los datos almacenados es necesaria en función de su sensibilidad. Existen muchos algoritmos para este propósito en los principales lenguajes de programación y que han sido ampliamente probados. No obstante, hay que estar siempre actualizado ante parches que solucionen vulnerabilidades detectadas. No hacerlo, al igual que tratar de implementar un algoritmo de encriptación propio, puede darnos una falsa sensación de seguridad que en cierto punto puede llegar a ser incluso más peligroso que no tener seguridad alguna.

Algunas bases de datos como SQL Server ofrecen la encriptación de datos de forma transparente. En otras tecnologías hará falta una llave explícita que se debe almacenar en un lugar seguro o de otra forma el atacante solo necesitaría encontrar la llave para acceder a los datos.

En un sistema basado en microservicios, puede ser una buena idea sólo encriptar las bases de datos o tablas de los servicios más sensibles. La encriptación debe realizarse la primera vez que se ven los datos y sólo debe almacenarse el resultado en un único sitio. La desencriptación debe realizarse bajo demanda para entidades reconocidas. La encriptación en las copias de seguridad también es fundamental.

4. Seguridad en profundidad: otras medidas de protección

La seguridad en profundidad es clave para interponer más barreras entre nuestro sistema y un atacante. Se pueden tomar medidas en diferentes niveles:

- Cortafuegos: puede consistir simplemente en limitar el acceso a cierto tipo de tráfico o de puertos. Tener varios cortafuegos puede ser interesante para diferenciar perímetros de seguridad.

- Logging: no es una medida de prevención, pero puede resultar útil para detectar y recuperarse de un ataque. No se debe almacenar información sensible en los logs.
- Sistemas de detección y prevención de intrusos (IDS e IPS): los primeros, se emplean para monitorizar las redes y hosts en busca de comportamientos sospechosos. Los sistemas IPS van un paso más allá y permiten detener actividades sospechosas.
- Segregación de redes: los servicios se pueden situar en diferentes segmentos de la red para controlar con cuáles puede hablar cada uno. Cada perímetro de la red puede pertenecer a un equipo y puede contar con diferentes medidas de seguridad. AWS ofrece la posibilidad de crear nubes virtuales privadas (VPC) para separar los hosts en subredes y controlar cuáles son accesibles por otras.
- Sistemas operativos: los servicios deben ejecutarse con un usuario con el menor número de permisos necesarios en el SO. Es importante aplicar los parches de seguridad que se publican ya que la mayoría de ataques se hacen a través de vulnerabilidades para las que existía un parche. Portales como Microsoft SCCM pueden mantenernos informados sobre esto. En Linux, se pueden aplicar medidas de seguridad que se controlan a nivel del kernel con aplicaciones como AppArmor.

5. Caso de estudio

En un sistema basado en microservicis, necesidades diferentes de seguridad pueden suponer medidas de seguridad diferentes en cada uno.

Para las aplicaciones clientes, como un navegador web, se puede emplear tráfico HTTPS en todas las páginas accesibles una vez el usuario se ha autenticado.

Para integrar con otros servicios, como un sistema de pagos, se pueden emplear certificados de cliente para garantizar la legitimidad de los mensajes intercambiados.

Para las APIs que nuestro sistema ofrece al público general, como hay interés en que sea fácil de usar y se emplee lo máximo posible, se pueden emplear claves API para identificar quien realiza cada petición e incluso limitar el número de peticiones que puede hacer.

Si contamos con servicios internos que son solo accesibles por nuestros servicios, se pueden emplear subredes para aislarlos del exterior y firewalls

para impedir tráfico no permitido.

En caso de que un servicio almacene datos sensibles, se pueden encriptar sus tablas, pero no las de servicios triviales.

6. Otros factores de la seguridad

- Datensparsamkeit: este término alemán significa que solo se debe almacenar la información que es absolutamente necesaria para llevar a cabo los objetivos del negocio o satisfacer las leyes gubernamentales. Los datos tienen valor porque pueden ser explotados a través de técnicas de Big Data. Esto entra en conflicto con la privacidad de los datos de los usuarios. Si aquellos datos que no son relevantes no se almacenan, entonces no pueden ser robados ni solicitados por otras entidades (como las judiciales).
- El factor humano: la ingeniería social o la forma en que se revocan permisos a antiguos empleados deben ser tenidos en cuenta. Hay que ponerse en la peor situación, ¿cuánto daño puede hacer un antiguo empleado descontento?
- No escribas tus propios algoritmos de encriptación: reinventar la rueda es una pérdida de tiempo y es potencialmente peligroso.
- Seguridad en la fase de Desarrollo: al igual que con las pruebas, la seguridad no debe ser implementada por otro equipo o en el último momento. Hace falta educar a los desarrolladores en conceptos clave de la seguridad como el Top 10 de OWASP de medidas.
- Herramientas para el CI: herramientas como Zed Attack Proxy (ZAP) para detectar vulnerabilidades se pueden añadir a las compilaciones para su ejecución automática como una medida de integración continua. La ejecución de analizadores estáticos también pueden avisarnos de agujeros de seguridad en nuestro código.
- Verificación externa: las pruebas de penetración se pueden llevar a cabo para imitar un ataque real. Estas pruebas deberían ser realizadas por una tercera organización ya que los equipos no siempre son capaces de detectar errores de seguridad que ellos mismos han cometido. Estas pruebas pueden aplicarse solo en la publicación a producción de grandes cambios y no necesariamente en cada versión.

Capítulo 11: Escalabilidad

Víctor Iranzo

3 de junio de 2018

1. Introducción

Podemos hacer todo lo posible para evitar que un sistema falle, pero llego un punto en la escala, la probabilidad de fallo es inevitable. Muchas organizaciones invierten mucho esfuerzo en evitar que un fallo se produzca, pero muy poco en mecanismos para recuperar el sistema una vez se ha producido.

Los mecanismos de escalabilidad de nuestro sistema han de ser conformes a sus necesidades: tener un sistema capaz de reaccionar automáticamente al incremento de carga o a un fallo es fantástico pero exagerado para ciertos servicios. También es necesario cuantos fallos pueden tolerar o cuánto tiempo puede estar el sistema caído para nuestros usuarios. Estos requisitos se pueden definir así:

- Latencia o tiempo de respuesta: cuánto tiempo puede una operación tardar. La carga que sufre un sistema puede influir en su tiempo de respuesta.
- Disponibilidad: cuánto tiempo está disponible un servicio respecto del tiempo que se deseaba que estuviera en funcionamiento. ¿Puede estar el servicio caído o tiene que estar disponible 24/7?
- Durabilidad de los datos: cuánto tiempo se deben almacenar ciertos datos y cuánta pérdida de datos se considera aceptable.
- Resilencia: es la capacidad de un sistema para tolerar fallos y continuar trabajando. Un sistema que por culpa de un servicio caído deja de funcionar es menos resilente que un sistema que puede continuar ofreciendo el resto de sus funcionalidades.

Existen varios modos de fallo que se pueden dar cuando un sistema está caído. Responder de forma lenta es uno de los peores porque aumenta la latencia de todas las peticiones para responder que el servicio está caído. Esto puede producir fallos en cascada dentro de la cadena de invocaciones.

2. Gestión de fallos en el sistema

En el libro "Antifragile"se explica como una organización como Netflix basada en la infraestructura de AWS prueba la tolerancia a fallos de sus servicios incitando al fallo de estos. Para ello, emplea un conjunto de programas que componen el "ejército de simios".

Chaos Monkey se encarga durante ciertas horas al día de apagar máquinas de forma aleatoria. Chaos Gorilla hace una función similar pero con los centros de datos. Latency Monkey simula redes de bajo rendimiento. Todos estos fallos pueden darse en producción y esta librería es una buena manera para comprobar si se está preparado para tolerarlos. Sin embargo, ¿cómo podemos gestionarlos una vez se producen?

2.1. Timeouts

Una manera de detectar los fallos es con una cuenta atrás. Esperar demasiado tiempo puede ralentizar al sistema entero. Esperar poco tiempo puede suponer considerar como fallidas tareas exitosas que han tardado más de lo normal. Una buena táctica para establecerlos es poner el mismo valor por defecto a todas las llamadas del sistema y ajustar los timeouts en función de los registros que obtengamos.

2.2. Cortocircuitos

Esta esta técnica se aplica cuando un servicio está caído o pasa un límite fijado por devolver timeouts o errores. Cuando esto ocurre, se dejan de mandar peticiones al servicio para que este devuelva un fallo rápido y no ralentice al resto del sistema. De esta forma, después de un período de tiempo podemos comprobar automáticamente si el servicio se ha recuperado y volverle a redirigir tráfico.

Como en el caso anterior, podemos establecer un cortocircuito con una condición por defecto y ajustarla según los casos específicos. Mientras la conexión está cerrada, las peticiones recibidas se pueden encolar para aten-

darlas más tarde. Existen diferentes librerías que implementan esto, como son la libería de Polly para .NET y la librería de Netflix Hystrix para Java.

Los cortocircuitos también se pueden emplear para tareas de mantenimiento programadas: cuando se tiene que modificar un servicio, se puede cortar la conexión de este y restablecerla una vez realizado el mantenimiento.

2.3. Bulkhead

Son una aproximación para aislar al sistema de un fallo en uno de sus componentes. Si tenemos una conexión distinta para cada servicio con el que interaccionemos, el que una de las conexiones vaya despacio no afectará al resto.

La separación en microservicios facilita el uso de bulkheads. A diferencia de los timeouts y los cortocircuitos que se emplean para actuar cuando un fallo se detecta, los bulkheads se pueden emplear para evitar que el fallo ocurra. Hystrix, por ejemplo, permite rechazar peticiones bajo ciertas condiciones para evitar que un servicio se sature, lo que se llama load schedding. Esto puede ser una buena medida para no convertir un servicio en un cuello de botella y evitar su fallo.

2.4. Aislamiento

Cuanto más dependa un servicio de otro para funcionar, más impacta la "‘salud"de uno sobre el otro. Esto se percibe sobretodo en las llamadas síncronas: un servicio puede tardar en responder no porque él no esté funcionando correctamente sino porque otro a quién ha invocado no contesta, produciendo un fallo en cascada.

3. Idempotencia

Si una operación se idempotente, podemos repetirla más de una vez sin un impacto adverso. Este tipo de operaciones es útil cuando no estamos seguros de si una operación se ha realizado y reenviamos la misma petición. En situaciones en las que tenemos varias instancias de un mismo servicio suscritas a los mismos eventos, aunque dos instancias procesen la misma petición no se producirá ningún error.

El sistema entero no tiene porque ser idempotente, solo las funcionalidades de negocio que se deseen. Algunos verbos HTTP como el GET o el PUT están definidos en su especificación como idempotentes. Si estos verbos no son idempotentes y los clientes piensan que sí, pueden surgir problemas a la hora de consumir el servicio.

4. Técnicas de escalabilidad

Generalmente un sistema se escala para lidiar con posibles fallos o mejorar su rendimiento. Vamos a ver diferentes técnicas para este propósito.

Rediseñar un sistema para que sea escalable no debe considerarse un fallo sino un éxito. En las fases iniciales de un proyecto no tiene sentido invertir demasiado tiempo en hacer el sistema escalable porque puedes estar invirtiendo esfuerzo para soportar una carga de trabajo que posiblemente igual no se dé nunca.

La mejor manera de hacer un sistema resilente es no situar todos los servicios en una misma infraestructura, ya sea una máquina física, red, etc. Algunos proveedores de infraestructura ofrecen facilidades para este propósito. Con esto se consigue que no haya un único punto de fallo.

4.1. Escalabilidad vertical

Consiste en aumentar las capacidades de la máquina donde se ejecuta parte del sistema, como mejorar su RAM o CPU. Esta aproximación puede resultar cara: un servidor grande puede costar más que dos pequeños que juntos sumen las mismas capacidades. Además, muchas veces el software no está preparado para beneficiarse de estas mejoras.

Aunque puede ser una manera sencilla para mejorar el rendimiento, problemas como la resilencia no se solucionan ya que si solo hay un servidor hay un único punto de fallo del sistema.

4.2. Dividir la carga de trabajo de un servicio

Si un servicio ofrece funcionalidades con diferente nivel de uso o criticidad, el servicio se puede dividir en otros más pequeños. De esta forma, se puede crear un servicio con la funcionalidad más crítica que se escale para aumentar su resilencia y de forma diferente al otro menos crítico.

4.3. Balanceadores de carga

Una forma de aumentar la resilencia de un servicio es evitar que este sea el único punto de fallo. Esto se puede conseguir teniendo varias instancias del mismo a las que se accede a través de un balanceador de carga. Un balanceador de carga se basa en un algoritmo para distribuir las llamadas que recibe entre las diferentes instancias de un servicio y además elimina instancias cuando no están accesibles y las vuelve a añadir cuando se recuperan. El balanceador de carga es transparente a los consumidores de un servicio.

Los balanceadores de carga pueden implementarse a nivel del hardware o el software. Los primeros son más difíciles de automatizar mientras que los segundos son más flexibles (mod_proxy).

4.4. Sistema basado en trabajadores

Emplear un balanceador de carga no es la única manera de tener diferentes instancias de un servicio. Un aproximación basada en trabajadores consiste en tener diferentes instancias que trabajan sobre un mismo backlog de peticiones. El número de trabajadores se puede adaptar a la demanda de cada momento (peaky load).

Los trabajadores en sí no necesitan ser resilentes: aunque un trabajador falle mientras atienda una petición, esta se terminará atendiendo por otra instancia más tarde. Una herramienta para implementar este tipo de sistemas es Zookeper.

5. Escalabilidad en bases de datos

Aunque se haya aumentado la resilencia del servicio ejecutando varias instancias de este en diferentes máquinas, todas ellas siguen accediendo a una misma infraestructura: la base de datos. La base de datos sigue siendo un único punto de fallo y puede convertirse en un cuello de botella. Además, si una misma base de datos contiene esquemas de diferentes servicios, su fallo puede ser catastrófico.

En esta sección veremos diferentes soluciones para escalar la base de datos según su tipo. Sin embargo, la solución más empleada suele ser aumentar las capacidades de la máquina donde se hospeda la base de datos, aunque esto puede emplearse hasta cierto límite.

5.1. Escalabilidad en la lectura

En bases de datos relacionales, los datos pueden ser copiados del nodo principal a varias replicas para distribuir las lecturas. Así, la escritura debe hacerse en el nodo principal pero la lectura se puede hacer en cualquier nodo.

Las modificaciones que se hagan en el nodo principal se propagan a las replicas en algún momento posterior a cada escritura. Esto significa que algunos datos que se leen en una replica pueden ser inconsistentes porque el modelo de consistencia es eventual.

Esta aproximación se emplea comúnmente y es fácil de emplear, pero pueden obtenerse beneficios similares a través de memorias cachés.

5.2. Escalabilidad en la escritura

La escalabilidad en la escritura es mucho más compleja. Una aproximación para llevarla a cabo es el sharding, donde se tiene la base de datos distribuida en diferentes nodos. Cuando se realiza una operación, a los datos (para su lectura o modificación) se les aplica una función hash y según el resultado obtenido se manda la consulta a un nodo u otro.

La complejidad del sharding proviene de consultas que requieren datos de múltiples nodos porque requiere consultar cada uno de los shards y juntar los resultados en memoria o tener un nodo con todos los datos de los shards para la lectura. Algunas herramientas suelen emplear cachés para las consultas que involucran a más de un shard.

Entre las herramientas existentes se pueden mencionar Mongo y Cassandra. Entre las funcionalidades de Cassandra encontramos por ejemplo que permite añadir nuevos shards en tiempo de ejecución y recolocar los contenidos del resto una vez hecho esto y que ayuda a mejorar la resilencia mediante la replicación de un mismo dato en más de un nodo.

5.3. CQRS

La segregación de responsabilidades de consultas y comandos (CQRS) consiste en separar el sistema de base de datos en dos partes: uno que atienda los comandos para modificarla y otro para atender las consultas. Así, cada una puede escalar de forma diferente.

Estas dos partes pueden hospedarse en servicios diferentes, aunque es-

to choca con servicios que ofrecen operaciones CRUD sobre una entidad. Además, la base de datos de lectura debe sincronizarse con los cambios que se hacen sobre la base de datos de escritura, lo que significa que puede dar datos obsoletos.

6. Caching

El uso de una caché consiste en almacenar el resultado de una operación para poder emplearlo en una petición posterior en lugar de gastar tiempo recalculando la misma operación.

Otros posibles usos de una caché son los siguientes: en la escritura, para mejorar el rendimiento de tal forma que el cambio se añade a un buffer que más adelante será escrito, o para mejorar la resilencia, ya en algunos sistemas si un servicio está caído podemos servirnos de datos almacenados en la caché (aunque estos puedan estar obsoletos) en lugar de devolver un fallo.

6.1. Tipos de caching

- Caching en la parte cliente: el cliente es quien almacena los datos cacheados. Con la ayuda del servidor decide durante cuánto tiempo son válidos los datos y cuándo debe solicitarlos de nuevo. Con esta aproximación se pueden reducir drásticamente las llamadas a través de la red. Sin embargo, la invalidación de los datos es compleja porque se debe trasladar a todos los clientes.
- Caching en el proxy: con esta aproximación se sitúa un proxy entre el cliente y el servidor que almacena los datos. Esta aproximación es opaca al cliente y al servidor y se puede emplear para cachear los resultados de más de un servicio a la vez. Los datos que se pueden almacenar son los del tráfico HTTP que redirige un proxy inverso a través de herramientas como Squid o Varnish.
- Caching en la parte servidora: el servidor es quien gestiona este sistema a través de cachés en memoria o herramientas como Redis, Memcache. Si la caché está dentro de los límites del servicio puede ser más fácil razonar sobre ella. Para el cliente la caché es transparente.

Según nuestras necesidades de carga o con que frecuencia se modifican los datos elegiremos una aproximación u otra, pudiendo emplearse diferentes en el conjunto del sistema. Hay que tener en cuenta que cuantas más cachés

haya entre la fuente de datos y el cliente, más probabilidad hay de que los datos estén obsoletos y más difícil será determinar si estos son todavía válidos.

6.2. Caching en el protocolo HTTP

El protocolo HTTP es ampliamente usado y existen un gran número de herramientas que gestionan de esta forma la caché de datos como son Squid o Varnish. En las cabeceras HTTP se pueden especificar directivas como cache-control para cachear respuestas durante un período de tiempo especificado en el valor de la clave Expires.

Otra aproximación es emplear etiquetas (ETags) para determinar si un recurso ha cambiado. Mediante el verbo GET se puede solicitar un recurso de tal forma que el servidor solo nos enviará este si se cumple cierta condición. Si hemos obtenido un recurso con un ETag y su tiempo de vida (TTL) ha expirado, podemos enviar al servidor una petición en forma de un GET condicional con la cláusula If-None-Match y la etiqueta del recurso. El servidor nos podrá responder de 2 posibles formas: con un "304 Not modifiedïndicando que el recurso que tenemos cacheado no ha sido modificado o con un "200 OK"que contenga la última versión del recurso y su nuevo ETag.

6.3. Fallos en la caché

Si se producen un gran número de fallos en la caché, las peticiones a la fuente de datos aumentarán drásticamente ralentizando el sistema entero.

Una manera de solucionar esto es haciendo que las llamadas entre la caché y la fuente de datos se hagan de forma asíncrona. De esta forma, cuando se produzca un fallo en la caché se devolverá un fallo al cliente inmediatamente y una invocación al origen de los datos, que eventualmente actualizará la caché. Obviamente, esta aproximación no tiene sentido para muchas situaciones.

7. Autoescalado

El escalado automático de los servicios es sencillo si se tienen implementados mecanismos para la provisión de hosts virtuales. También se debe

gestionar la eliminación de instancias que apenas reciben peticiones. El escalado puede estar conducido de dos formas distintas:

- Escalado reactivo: en este aproximación aumentamos el número de hosts en función de tendencias bien conocidas, como puede ser un período de facturación, la hora del día con más visitas a la web...
- Escalado predictivo: se aumentan el número de hosts cuando se observa un aumento en las peticiones de un servicio o cuando una de sus instancias falla. Esta aproximación es más difícil porque la carga puede aumentar en pocos minutos, pero el aprovisionamiento de una nueva instancia puede tardar al menos 10 minutos. Es aconsejable tener una suite de tests de carga para probar las reglas de autoescalado. Las reglas para gestionar los fallos de una instancia son mucho más usadas que las de carga de trabajo. Como consejo, tener más instancias de las necesarias es mucho mejor que no tener las suficientes, por lo que las reglas han de ser capaces de reaccionar rápidamente.

8. El teorema de CAP

En los sistemas distribuidos existren 3 características sobre las que se debe hacer balance: la consistencia, que establece que vamos a obtener la misma respuesta de nodos diferentes, la disponibilidad, que significa que toda petición recibirá una respuesta, y la tolerancia a particiones, que es la habilidad de gestionar situaciones en las que la comunicación entre las partes de un sistema se interrumpe.

Pongamos un ejemplo en el que una base de datos está replicada y se produce un fallo por el cual la comunicación entre ambas se interrumpe y los cambios en una no se pueden propagar a la otra. En una situación como esta, el teorema de CAP establece que solo podemos mantener 2 de las 3 características que hemos mencionado antes.

El caso más trivial es el de renunciar a la tolerancia de particiones: si no existe, el sistema no puede ejecutarse sobre una red y el sistema deja de ser distribuido. El sacrificio de las otras dos características da lugar a los llamados sistemas AP y CP.

8.1. Sistemas AP

Son los sistemas que surgen fruto de sacrificar la consistencia cuando un fallo se produce. Se permite realizar cambios en ambas bases de datos, con

lo que se mantiene la disponibilidad como característica, pero como estas no se sincronizan podemos estar obteniendo datos obsoletos al hacer una consulta.

En algún momento cuando la conexión se restablezca los cambios se sincronizarán, hecho que será más difícil cuanto mayor sea el período sin conexión. La consistencia es eventual, como ocurre en la propagación de los cambios de una base de datos a otra por no poder ser inmediata.

8.2. Sistemas CP

Para mantener la consistencia entre las bases de datos cuando no existe comunicación entre ellas se tiene que rechazar cualquier petición, con lo que dejan de estar disponibles. La única manera de saber que la información de una base de datos es consistente es preguntado al resto de nodos. La lectura por tanto se debe hacer de forma transaccional mediante bloqueos para evitar que mientras se está leyendo un dato este pueda estar siendo modificado al mismo tiempo en otro nodo.

El uso de bloqueos es difícil de gestionar en sistemas distribuidos: si se interrumpe la comunicación, un nodo puede quedar bloqueado permanentemente a la espera de ser desbloqueado. Por este motivo, no debemos tratar de implementar este tipo de sistemas, sino que podemos aprovecharnos de herramientas como Consul.

8.3. Balance entre sistemas AP y sistemas CP

Los sistemas AP escalan más fácilmente y son más sencillos de construir. Los segundos son los únicos que nos aseguran la consistencia de los datos, pero son más difíciles de construir. A la hora de hacer balance entre una solución u otra, debemos preguntarnos cuánto tiempo puede en nuestro servicio estar un dato obsoleto. El sistema no necesita hacerse completamente de una u otra manera.

9. Descubrimiento de servicios

Es imprescindible en un sistema basado en microservicios conocer donde se están ejecutando cada uno de ellos. Estas capacidades se encuentran dentro del llamado descubrimiento de servicios, que se divide en dos partes: el registro de un servicio y el acceso a los servicios registrados. El descu-

brimiento de servicios se vuelve más complicado en un entorno en el que constantemente se despliegan y destruyen instancias de un servicio.

9.1. DNS

DNS es un estándar muy conocido y extendido con un gran número de tecnologías que le dan soporte. Una manera de establecer donde se encuentra cada servicio es mediante una entrada por servicio que apunte a un balanceador de carga que a su vez apunte a las diferentes instancias de ese servicio desplegadas. En caso de que solo haya una instancia del servicio, la entrada del servicio puede referir directamente al host donde se ejecuta.

Por último, hay que tener en cuenta que las entradas DNS tienen un tiempo de vida y es peligro la cantidad de sitios donde pueden ser cacheadas ya que pueden quedar obsoletas.

9.2. Herramientas para el descubrimiento de servicios

Entre las desventajas del sistema de DNSs es que no se adapta a un entorno que cambia frecuentemente como es el de los microservicios. Como soluciones alternativas están las siguientes, que en su mayoría se basan en contar con un servicio central donde el resto se registran:

- Zookeeper: se emplea para un gran número de casos de uso como la gestión de la configuración, la sincronización de datos entre servicios o como un servicio para registrar a otros. Ofrece un espacio de nombres jerárquico para almacenar información y al que nos podemos suscribir para estar alerta de cuando un nodo se modificaa. Este espacio de nombres se puede emplear como estructura para almacenar la localización de cada servicio.
- Consul: entre sus funcionalidades está que expone una interfaz HTTP para el descubrimiento de servicios y que ofrece un servidor DNS para almacenar el IP y el puerto de un servicio indexado por su nombre.
- Eureka: es una librería de código abierto de Netflix. Al emplearlo son los clientes quienes deben lidiar con el descubrimiento de servicios directamente, sin necesidad de un proceso o servicio aparte. Como consecuencia, la tecnología empleada tiene que soportar JVM.

Para concluir, cualquier herramienta elegida debería permitir construir

un panel para que los humanos puedan monitorizar los diferentes servicios desplegados.

10. Documentación de servicios

Sabemos dónde se encuentra cada servicio, pero también necesitamos saber que funcionalidades ofrece cada uno. Esto se puede conseguir mediante la documentación a través de herramientas como las siguientes:

- Swagger: describe la API de un servicio a través de una interfaz web con la que se puede interactuar. Los métodos que pertenecen a la API pueden ser anotados dentro del código para que se genere la documentación automáticamente.
- Hypertext Application Language (HAL): es un estándar que describe los controles hipermedia que hemos expuesto. El navegador HAL ofrece la posibilidad de explor la API y ejecutar llamada a través de un servicio mediante el navegador web.

11. Descripción de un servicio

El estándar Universal Description, Discovery and Integration (UDDI) surgió en las fases iniciales de las arquitecturas basadas en servicios. Esta aproximación para el registro de un servicio es muy pesada y se pueden realizar aproximaciones más ligeras a través de una simple wiki.

La información que necesitamos conocer de cada servicio es su estado, la localización de sus instancias o su relación con otros servicios. Mantener esta información accesible y de forma visual es clave para luchar contra la complejidad que supone la escalabilidad del sistema.