

La cascada

Los fallos en cascada pueden ser especialmente peligrosos. Imaginemos una situación en la que se interrumpe la conexión de red entre el sitio web de nuestra tienda de música y el servicio de catálogo. Los servicios parecen funcionar correctamente, pero no pueden comunicarse entre sí. Si solo miráramos el estado de cada servicio individual, no sabríamos que hay un problema. Si utilizáramos un sistema de monitorización sintética (por ejemplo, para imitar a un cliente que busca una canción), detectaríamos el problema. Pero también necesitaríamos informar sobre el hecho de que un servicio no puede ver a otro para determinar la causa del problema.

Por lo tanto, es fundamental supervisar los puntos de integración entre sistemas. Cada instancia de servicio debe realizar un seguimiento y exponer el estado de sus dependencias posteriores, desde la base de datos hasta otros servicios colaboradores. También debe permitir que esta información se agregue para obtener una imagen general. Querrá ver el tiempo de respuesta de las llamadas posteriores y también detectar si hay errores.

Como veremos más adelante en [el Capítulo 11](#), puede usar bibliotecas para implementar un disyuntor en torno a las llamadas de red que lo ayude a manejar fallas en cascada de una manera más elegante, lo que le permitirá degradar su sistema de manera más elegante. Algunas de estas bibliotecas, como Hystrix para la JVM, también hacen un buen trabajo al brindarle estas capacidades de monitoreo.

Normalización

Como ya hemos comentado anteriormente, uno de los equilibrios que deberás mantener es decidir en qué casos es necesario que se tomen decisiones de forma específica para un único servicio y en qué casos es necesario estandarizar todo el sistema. En mi opinión, la supervisión es un área en la que la estandarización es increíblemente importante. Dado que los servicios colaboran de muchas formas diferentes para proporcionar capacidades a los usuarios que utilizan varias interfaces, es necesario ver el sistema de forma holística.

Debes intentar escribir tus registros en un formato estándar. Definitivamente, querrás tener todas tus métricas en un solo lugar y es posible que también quieras tener una lista de nombres estándar para tus métricas; sería muy molesto que un servicio tenga una métrica llamada ResponseTime y otro tenga una llamada RspTimeSecs, cuando significan lo mismo.

Como siempre ocurre con la estandarización, las herramientas pueden ayudar. Como he dicho antes, la clave es facilitar la tarea de hacer lo correcto. ¿Por qué no ofrecer imágenes de máquinas virtuales preconfiguradas con logstash y collectd listas para usar, junto con bibliotecas de aplicaciones que le permitan comunicarse con Graphite con mucha facilidad?

Considera a la audiencia

Todos estos datos que estamos recopilando tienen un propósito. Más específicamente, estamos recopilando todos estos datos para ayudar a diferentes personas a realizar su trabajo; estos datos se convierten en un llamado a la acción. Algunos de estos datos deben activar una llamada de atención inmediata para nuestro equipo de soporte; por ejemplo, en el caso de que una de nuestras pruebas de monitoreo sintético falle. Otros datos, como el hecho de que nuestra carga de CPU aumentó un 2 % durante la última semana, posiblemente solo sean de interés cuando estamos haciendo una planificación de la capacidad. Del mismo modo, su jefe probablemente querrá saber de inmediato que los ingresos cayeron un 25 % después del último lanzamiento, pero probablemente no necesite que lo despierten porque las búsquedas de "Justin Bieber" aumentaron un 5 % en la última hora.

Lo que nuestra gente quiere ver y a lo que quiere reaccionar en este momento es diferente de lo que necesita cuando analiza los datos en profundidad. Por lo tanto, para el tipo de persona que analizará estos datos, considere lo siguiente:

- Lo que necesitan saber ahora mismo
- Lo que podrían querer más adelante
- Cómo les gusta consumir datos

Alerta a los empleados sobre lo que necesitan saber ahora mismo. Cree grandes pantallas visibles con esta información que se ubiquen en un rincón de la sala. Déles un acceso fácil a los datos que necesitan conocer más tarde y dedique tiempo a ellos para saber cómo quieren consumir los datos. Una discusión sobre todos los matices involucrados en la presentación gráfica de información cuantitativa ciertamente queda fuera del alcance de este libro, pero un excelente lugar para comenzar es el excelente libro de Stephen Few, *Information Dashboard Design: Displaying Data for At-a-Glance Monitoring* (Analytics Press).

El futuro

He visto muchas organizaciones donde las métricas están divididas en diferentes sistemas.

Las métricas a nivel de aplicación, como la cantidad de pedidos realizados, terminan en un sistema de análisis propietario como Omniture, que a menudo está disponible solo para partes seleccionadas del negocio, o bien terminan en el temido almacén de datos, también conocido como el lugar donde los datos van a morir. Los informes de estos sistemas no suelen estar disponibles en tiempo real, aunque eso está empezando a cambiar. Mientras tanto, las métricas del sistema , como los tiempos de respuesta, las tasas de error y la carga de la CPU, se almacenan en sistemas a los que los equipos de operaciones pueden acceder. Estos sistemas suelen permitir la elaboración de informes en tiempo real, ya que normalmente su objetivo es provocar una llamada a la acción inmediata.

Históricamente, la idea de que podemos conocer las métricas empresariales clave un día o dos después era buena, ya que normalmente no podíamos reaccionar lo suficientemente rápido a estos datos como para hacer algo al respecto. Sin embargo, ahora operamos en un mundo en el que muchos de nosotros podemos lanzar y lanzamos múltiples versiones por día. Los equipos ahora se miden a sí mismos no en términos de cuántos puntos completan, sino que optimizan el tiempo que tarda el código en llegar desde la computadora portátil hasta la vida real. En un entorno así, necesitamos tener todas nuestras métricas a nuestro alcance para tomar la acción correcta. Irónicamente, los mismos sistemas que almacenan las métricas empresariales a menudo no están ajustados para el acceso inmediato a los datos, pero nuestros sistemas operativos sí lo están.

Entonces, ¿por qué manejar las métricas operativas y comerciales de la misma manera? En última instancia, ambos tipos de cosas se reducen a eventos que indican que algo sucedió en X. Por lo tanto, si podemos unificar los sistemas que utilizamos para recopilar, agregar y almacenar estos eventos, y ponerlos a disposición para la elaboración de informes, obtenemos una arquitectura mucho más simple.

Riemann es un servidor de eventos que permite la agregación y el enrutamiento de eventos bastante avanzados y puede formar parte de una solución de este tipo. **Suro** es el canal de datos de Netflix y opera en un espacio similar. Suro se utiliza explícitamente para gestionar tanto las métricas asociadas con el comportamiento del usuario como los datos más operativos, como los registros de aplicaciones. Estos datos se pueden enviar a una variedad de sistemas, como Storm para el análisis en tiempo real, Hadoop para el procesamiento por lotes sin conexión o Kibana para el análisis de registros.

Muchas organizaciones están avanzando en una dirección fundamentalmente diferente: están dejando de tener cadenas de herramientas especializadas para distintos tipos de métricas y están adoptando sistemas de enrutamiento de eventos más genéricos capaces de alcanzar una escala significativa. Estos sistemas logran brindar mucha más flexibilidad y, al mismo tiempo, simplifican nuestra arquitectura.

Resumen

¡Hemos cubierto muchos temas aquí! Intentaré resumir este capítulo en algunos consejos fáciles de seguir.

Para cada servicio:

- Realice un seguimiento del tiempo de respuesta entrante al mínimo. Una vez que haya hecho eso, continúe con las tasas de error y luego comience a trabajar en las métricas a nivel de aplicación.
- Realice un seguimiento del estado de todas las respuestas posteriores, incluido como mínimo el tiempo de respuesta de las llamadas posteriores y, en el mejor de los casos, el seguimiento de las tasas de error. Las bibliotecas como Hystrix pueden ayudar en este sentido.
- Estandarizar cómo y dónde se recopilan las métricas.
- Inicie sesión en una ubicación estándar y, si es posible, en un formato estándar. ¡La agregación es un problema si cada servicio utiliza un diseño diferente!
- Supervise el sistema operativo subyacente para poder rastrear procesos no autorizados y planificar la capacidad.

Para el sistema:

- Agregue métricas a nivel de host, como CPU, junto con métricas a nivel de aplicación.
- Asegúrese de que su herramienta de almacenamiento de métricas permita la agregación a nivel de sistema o servicio y profundice en los hosts individuales.
- Asegúrese de que su herramienta de almacenamiento de métricas le permita mantener los datos durante el tiempo suficiente para comprender las tendencias en su sistema.
- Disponga de una única herramienta de consulta para agregar y almacenar registros.
- Se debería considerar seriamente estandarizar el uso de identificadores de correlación.
- Comprenda qué requiere un llamado a la acción y estructure las alertas y los paneles de control en consecuencia.
- Investigue la posibilidad de unificar la forma en que agrega todas sus diversas métricas viendo si una herramienta como Suro o Riemann tiene sentido para usted.

También he intentado esbozar la dirección en la que se está moviendo el monitoreo: alejándose de los sistemas especializados para hacer una sola cosa y hacia sistemas de procesamiento de eventos genéricos que le permiten ver su sistema de una manera más holística. Este es un espacio emocionante y emergente, y aunque una investigación completa está fuera del alcance de este libro, espero haberle brindado suficiente información para comenzar. Si desea saber más, analizo algunas de estas ideas.

y más en mi publicación anterior [Sistemas livianos para monitoreo en tiempo real](#) (O'Reilly).

En el próximo capítulo, adoptaremos una visión holística diferente de nuestros sistemas para considerar algunas de las ventajas y desafíos únicos que las arquitecturas de grano fino pueden proporcionar en el área de seguridad.

Capítulo 9. Seguridad

Nos hemos familiarizado con historias sobre violaciones de seguridad de sistemas a gran escala que dan como resultado que nuestros datos queden expuestos a todo tipo de personajes sospechosos. Pero más recientemente, eventos como las revelaciones de Edward Snowden nos han hecho aún más conscientes del valor de los datos que las empresas tienen sobre nosotros y del valor de los datos que nosotros tenemos para nuestros clientes en los sistemas que construimos. Este capítulo brindará una breve descripción general de algunos aspectos de seguridad que debe considerar al diseñar sus sistemas. Si bien no pretende ser exhaustivo, expondrá algunas de las principales opciones disponibles y le brindará un punto de partida para su propia investigación adicional.

Debemos pensar en qué protección necesitan nuestros datos mientras están en tránsito de un punto a otro y qué protección necesitan cuando están en reposo. Debemos pensar en la seguridad de nuestros sistemas operativos subyacentes y también de nuestras redes. ¡Hay tanto en qué pensar y tanto que podríamos hacer! Entonces, ¿cuánta seguridad necesitamos? ¿Cómo podemos determinar qué es suficiente seguridad?

Pero también tenemos que pensar en el elemento humano. ¿Cómo sabemos quién es una persona y qué puede hacer? ¿Y cómo se relaciona esto con la forma en que nuestros servidores se comunican entre sí? Empecemos por ahí.

Autenticación y autorización

La autenticación y la autorización son conceptos fundamentales cuando se trata de personas y cosas que interactúan con nuestro sistema. En el contexto de la seguridad, la autenticación es el proceso mediante el cual confirmamos que una parte es quien dice ser. En el caso de un ser humano, normalmente se autentica a un usuario pidiéndole que escriba su nombre de usuario y contraseña. Suponemos que solo ella tiene acceso a esta información y, por lo tanto, que la persona que ingresa esta información debe ser ella. Por supuesto, también existen otros sistemas más complejos. Mi teléfono ahora me permite usar mi huella digital para confirmar que soy quien digo ser. Por lo general, cuando hablamos de manera abstracta sobre quién o qué se está autenticando, nos referimos a esa parte como el principal.

La autorización es el mecanismo por el cual asignamos una acción a un principal y la que le permitimos realizar. A menudo, cuando se autentica a un principal, se nos dará información sobre él que nos ayudará a decidir qué debemos permitirle hacer. Por ejemplo, se nos podría decir en qué departamento u oficina trabaja, información que nuestros sistemas pueden usar para decidir qué puede y qué no puede hacer.

En el caso de aplicaciones monolíticas individuales, es habitual que la propia aplicación se encargue de la autenticación y la autorización. Django, el marco web de Python, por ejemplo, viene con la gestión de usuarios de fábrica. Sin embargo, cuando se trata de sistemas distribuidos, debemos pensar en esquemas más avanzados. No queremos que todos tengan que iniciar sesión por separado en diferentes sistemas, utilizando un nombre de usuario y una contraseña diferentes para cada uno.

El objetivo es tener una única identidad que podamos autenticar una sola vez.

Implementaciones comunes de inicio de sesión único

Un enfoque común para la autenticación y la autorización es utilizar algún tipo de solución de inicio de sesión único (SSO). SAML, que es la implementación reinante en el ámbito empresarial, y OpenID Connect ofrecen capacidades en esta área. Utilizan más o menos los mismos conceptos básicos, aunque la terminología difiere ligeramente. Los términos utilizados aquí son de SAML.

Cuando un principal intenta acceder a un recurso (como una interfaz basada en la web), se le indica que se autentique con un proveedor de identidad. Este puede solicitarle que proporcione un nombre de usuario y una contraseña, o puede utilizar algo más avanzado como la autenticación de dos factores. Una vez que el proveedor de identidad está seguro de que el principal ha sido autenticado, proporciona información al proveedor de servicios, lo que le permite decidir si le concede acceso al recurso.

Este proveedor de identidad puede ser un sistema alojado externamente o algo dentro de su propia organización. Google, por ejemplo, ofrece un proveedor de identidad OpenID Connect. Sin embargo, para las empresas, es común tener su propio proveedor de identidad, que puede estar vinculado al servicio de directorio de su empresa. Un servicio de directorio puede ser algo como el Protocolo ligero de acceso a directorios (LDAP) o Active Directory. Estos sistemas le permiten almacenar información sobre los principales, como los roles que desempeñan en la organización.

A menudo, el servicio de directorio y el proveedor de identidad son uno y el mismo, mientras que a veces son independientes pero están vinculados. Okta, por ejemplo, es un proveedor de identidad SAML alojado que se encarga de tareas como la autenticación de dos factores, pero que puede vincularse a los servicios de directorio de su empresa como fuente de información.

SAML es un estándar basado en SOAP y es conocido por ser bastante complejo de usar a pesar de las bibliotecas y herramientas disponibles para respaldarlo. OpenID Connect es un estándar que surgió como una implementación específica de OAuth 2.0, basada en la forma en que Google y otros manejan el SSO. Utiliza llamadas REST más simples y, en mi opinión, es probable que se abra paso en las empresas debido a su mayor facilidad de uso. Su mayor obstáculo en este momento es la falta de proveedores de identidad que lo respalden. Para un sitio web público, puede que no le moleste usar a Google como proveedor, pero para sistemas internos o sistemas en los que desea tener más control y visibilidad sobre cómo y dónde se instalan sus datos, querrá su propio proveedor de identidad interno. En el momento de escribir este artículo, OpenAM y Gluu son dos de las pocas opciones disponibles en este espacio, en comparación con una gran cantidad de opciones para SAML (incluido Active Directory, que parece estar en todas partes). Hasta que los proveedores de identidad existentes comiencen a admitir OpenID Connect, su crecimiento puede limitarse a aquellas situaciones en las que las personas estén contentas con usar un proveedor de identidad público.

Si bien creo que OpenID Connect es el futuro, es muy posible que lleve un tiempo hasta que se adopte de forma generalizada.

Puerta de enlace de inicio de sesión único

En una configuración de microservicios, cada servicio podría decidir gestionar la redirección y el enlace con el proveedor de identidad. Obviamente, esto podría significar mucho trabajo duplicado. Una biblioteca compartida podría ayudar, pero tendríamos que tener cuidado para evitar el acoplamiento que puede surgir del código compartido. Esto tampoco sería de ayuda si tuvieras varias pilas de tecnología diferentes.

En lugar de que cada servicio gestione el protocolo de enlace con su proveedor de identidad, puede utilizar una puerta de enlace que actúe como proxy, ubicándose entre sus servicios y el mundo exterior (como se muestra en [la Figura 9-1](#)). La idea es que podamos centralizar el comportamiento para redirigir al usuario y realizar el protocolo de enlace en un solo lugar.

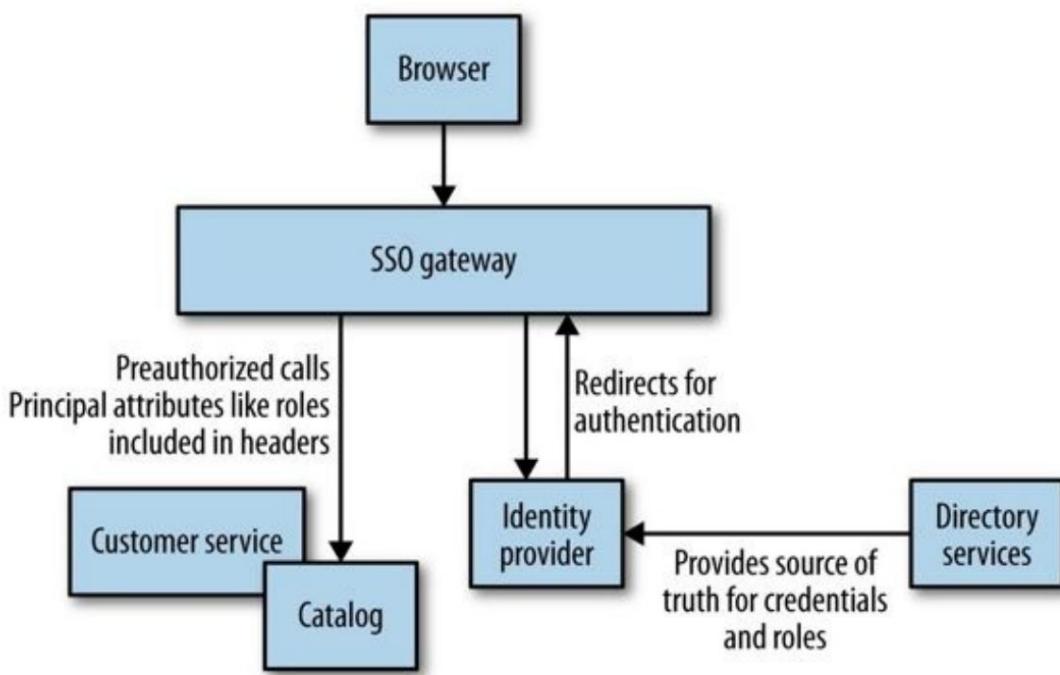


Figura 9-1. Uso de una puerta de enlace para gestionar el inicio de sesión único

Sin embargo, todavía tenemos que resolver el problema de cómo el servicio de bajada recibe información sobre los principales, como su nombre de usuario o qué funciones desempeñan. Si está utilizando HTTP, podría completar los encabezados con esta información. Shibboleth es una herramienta que puede hacer esto por usted y he visto que se utiliza con Apache con gran eficacia para gestionar la integración con proveedores de identidad basados en SAML.

Otro problema es que si hemos decidido delegar la responsabilidad de la autenticación en una puerta de enlace, puede resultar más difícil razonar sobre cómo se comporta un microservicio cuando se lo analiza de forma aislada. ¿Recuerdas en [el Capítulo 7](#), donde exploramos algunos de los desafíos que implica reproducir entornos similares a los de producción? Si optas por la ruta de la puerta de enlace, asegúrate de que tus desarrolladores puedan lanzar sus servicios detrás de una sin demasiado trabajo.

Un problema final con este enfoque es que puedes hacerte sentir una falsa sensación de seguridad. Me gusta la idea de una defensa en profundidad, desde el perímetro de la red, hasta la subred, el firewall y la seguridad.

La seguridad de la máquina, el sistema operativo y el hardware subyacente son aspectos que se pueden implementar en todos estos puntos, algunos de los cuales abordaremos en breve. He visto a algunas personas apostar todos los huevos a una sola carta y confiar en que la puerta de enlace se encargue de cada paso por ellas. Y todos sabemos lo que ocurre cuando tenemos un único punto de fallo...

Obviamente, podrías usar esta puerta de enlace para hacer otras cosas. Si usas una capa de instancias Apache que ejecutan Shibboleth, por ejemplo, también podrías decidir terminar HTTPS en este nivel, ejecutar la detección de intrusiones, etc. Pero ten cuidado. Las capas de puerta de enlace tienden a asumir cada vez más funcionalidad, lo que en sí mismo puede terminar siendo un punto de acoplamiento gigante. Y cuanto más funcionalidad tenga algo, mayor será la superficie de ataque.

Autorización de Granularidad Fina

Una puerta de enlace puede proporcionar una autenticación de grano grueso bastante eficaz. Por ejemplo, podría impedir el acceso a la aplicación de soporte técnico a cualquier usuario que no haya iniciado sesión. Suponiendo que nuestra puerta de enlace puede extraer atributos sobre el principal como resultado de la autenticación, es posible que pueda tomar decisiones más matizadas. Por ejemplo, es común colocar a las personas en grupos o asignarles roles. Podemos usar esta información para comprender lo que pueden hacer. Por lo tanto, para la aplicación de soporte técnico, podríamos permitir el acceso solo a los principales con un rol específico (por ejemplo, PERSONAL). Sin embargo, más allá de permitir (o no permitir) el acceso a recursos o puntos finales específicos, debemos dejar el resto al propio microservicio; deberá tomar decisiones adicionales sobre qué operaciones permitir.

Volviendo a nuestra aplicación de soporte técnico: ¿permitimos que cualquier miembro del personal vea todos los detalles? Lo más probable es que tengamos diferentes roles en el trabajo. Por ejemplo, un director del grupo CALL_CENTER podría tener permiso para ver cualquier dato sobre un cliente, excepto sus detalles de pago. El director también podría emitir reembolsos, pero ese monto podría estar limitado. Sin embargo, alguien que tenga el rol CALL_CENTER_TEAM_LEADER podría emitir reembolsos más grandes.

Estas decisiones deben ser locales para el microservicio en cuestión. He visto a personas usar los diversos atributos proporcionados por los proveedores de identidad de formas horribles, utilizando funciones muy específicas como CALL_CENTER_50_DOLLAR_REFUND, donde terminan colocando información específica de una parte del comportamiento de uno de nuestros sistemas en sus servicios de directorio. Esto es una pesadilla para mantener y da muy poco margen para que nuestros servicios tengan su propio ciclo de vida independiente, ya que de repente una parte de la información sobre cómo se comporta un servicio se encuentra en otra parte, tal vez en un sistema administrado por una parte diferente de la organización.

En lugar de eso, favorezca roles más específicos, diseñados en torno al funcionamiento de su organización. Volviendo a los primeros capítulos, recuerde que estamos creando software para que se adapte al funcionamiento de nuestra organización. Por lo tanto, utilice sus roles de esta manera también.

Autenticación y autorización de servicio a servicio

Hasta este punto, hemos utilizado el término principal para describir cualquier cosa que pueda autenticarse y estar autorizada a hacer cosas, pero nuestros ejemplos en realidad han sido sobre humanos que utilizan computadoras. Pero ¿qué sucede con los programas u otros servicios que se autentican entre sí?

Permitir todo dentro del perímetro

Nuestra primera opción podría ser simplemente asumir que cualquier llamada a un servicio realizada desde dentro de nuestro perímetro es implícitamente confiable.

Dependiendo de la sensibilidad de los datos, esto podría ser aceptable. Algunas organizaciones intentan garantizar la seguridad en el perímetro de sus redes y, por lo tanto, suponen que no necesitan hacer nada más cuando dos servicios se comunican entre sí. Sin embargo, si un atacante penetra en su red, tendrá poca protección contra un ataque típico de intermediario . Si el atacante decide interceptar y leer los datos que se envían, cambiarlos sin que usted lo sepa o incluso, en algunas circunstancias, fingir ser el objeto con el que está hablando, es posible que no sepa mucho al respecto.

Esta es, con diferencia, la forma más común de confianza dentro del perímetro que veo en las organizaciones. Pueden decidir ejecutar este tráfico a través de HTTPS, pero no hacen mucho más. ¡No digo que sea algo bueno! En el caso de la mayoría de las organizaciones que veo que utilizan este modelo, me preocupa que el modelo de confianza implícita no sea una decisión consciente, sino más bien que las personas no sean conscientes de los riesgos en primer lugar.

Autenticación básica HTTP(S)

La autenticación básica HTTP permite que un cliente envíe un nombre de usuario y una contraseña en un encabezado HTTP estándar. El servidor puede comprobar estos detalles y confirmar que el cliente tiene permiso para acceder al servicio. La ventaja es que se trata de un protocolo muy conocido y con un gran respaldo. El problema es que hacerlo a través de HTTP es muy problemático, ya que el nombre de usuario y la contraseña no se envían de forma segura. Cualquier intermediario puede consultar la información del encabezado y ver los datos. Por lo tanto, la autenticación básica HTTP debería utilizarse normalmente a través de HTTPS.

Al utilizar HTTPS, el cliente obtiene garantías sólidas de que el servidor con el que está hablando es quien cree que es. También nos brinda protección adicional contra personas que espían el tráfico entre el cliente y el servidor o que alteran la carga útil.

El servidor debe gestionar sus propios certificados SSL, lo que puede resultar problemático cuando se gestionan varias máquinas. Algunas organizaciones se encargan de su propio proceso de emisión de certificados, lo que supone una carga administrativa y operativa adicional. Las herramientas para gestionar esto de forma automatizada no son tan maduras como podrían serlo, y no es solo el proceso de emisión lo que hay que gestionar. Los certificados autofirmados no son fáciles de revocar y, por lo tanto, requieren mucha más reflexión sobre los escenarios de desastre. Vea si puede evitar todo este trabajo evitando por completo la autofirma.

Otra desventaja es que el tráfico enviado a través de SSL no puede almacenarse en caché mediante servidores proxy inversos como Varnish o Squid. Esto significa que, si necesita almacenar en caché el tráfico, deberá hacerlo dentro del servidor o dentro del cliente. Puede solucionar esto haciendo que un balanceador de carga finalice el tráfico SSL y que la caché se coloque detrás del balanceador de carga.

También tenemos que pensar en lo que sucede si estamos usando una solución SSO existente, como SAML, que ya tiene acceso a nombres de usuario y contraseñas. ¿Queremos que nuestra autenticación de servicio básica use el mismo conjunto de credenciales, lo que nos permite un proceso para emitirlas y revocarlas? Podríamos hacer esto haciendo que el servicio se comunique con el mismo servicio de directorio que respalda nuestra solución SSO. Alternativamente, podríamos almacenar los nombres de usuario y las contraseñas nosotros mismos dentro del servicio, pero entonces corremos el riesgo de duplicar el comportamiento.

Nota: con este método, todo lo que sabe el servidor es que el cliente tiene el nombre de usuario y la contraseña. No tenemos idea de si esta información proviene de una máquina que esperamos; podría provenir de cualquier persona en nuestra red.

Utilice SAML o OpenID Connect

Si ya utiliza SAML u OpenID Connect como esquema de autenticación y autorización, también podría utilizarlos para interacciones entre servicios. Si utiliza una puerta de enlace, también deberá enrutar todo el tráfico dentro de la red a través de la puerta de enlace, pero si cada servicio se encarga de la integración por sí mismo, este enfoque debería funcionar de inmediato. La ventaja aquí es que está haciendo uso de la infraestructura existente y puede centralizar todos los controles de acceso a los servicios en un servidor de directorio central. Aún tendríamos que enrutar esto a través de HTTPS si quisieramos evitar ataques de intermediarios.

Los clientes tienen un conjunto de credenciales que utilizan para autenticarse con el proveedor de identidad, y el servicio obtiene la información que necesita para decidir sobre cualquier autenticación detallada.

Esto significa que necesitará una cuenta para sus clientes, a veces denominada cuenta de servicio. Muchas organizaciones utilizan este enfoque con bastante frecuencia. Sin embargo, una advertencia: si va a crear cuentas de servicio, intente limitar su uso. Por lo tanto, considere que cada microservicio tenga su propio conjunto de credenciales. Esto hace que revocar o cambiar el acceso sea más fácil si las credenciales se ven comprometidas, ya que solo necesita revocar el conjunto de credenciales que se han visto afectadas.

Sin embargo, existen otras desventajas. En primer lugar, al igual que con la autenticación básica, necesitamos almacenar de forma segura nuestras credenciales: ¿dónde se encuentran el nombre de usuario y la contraseña? El cliente deberá encontrar una forma segura de almacenar estos datos. El otro problema es que parte de la tecnología en este espacio para realizar la autenticación es bastante tediosa de codificar. SAML, en particular, hace que la implementación de un cliente sea una tarea complicada. OpenID Connect tiene un flujo de trabajo más simple, pero como comentamos antes, aún no tiene un buen soporte.

Certificados de cliente

Otro enfoque para confirmar la identidad de un cliente es utilizar las capacidades de Seguridad de la capa de transporte (TLS), el sucesor de SSL, en forma de certificados de cliente. Aquí, cada cliente tiene instalado un certificado X.509 que se utiliza para establecer un vínculo entre el cliente y el servidor. El servidor puede verificar la autenticidad del certificado del cliente, lo que proporciona garantías sólidas de que el cliente es válido.

Los desafíos operativos en la gestión de certificados son incluso más onerosos que con el uso de certificados del lado del servidor. No se trata solo de algunos de los problemas básicos de la creación y gestión de una mayor cantidad de certificados; más bien, es que con todas las complejidades en torno a los certificados en sí, es de esperar que pase mucho tiempo tratando de diagnosticar por qué un servicio no acepta lo que usted cree que es un certificado de cliente completamente válido. Y luego tenemos que considerar la dificultad de revocar y volver a emitir certificados si ocurriera lo peor. El uso de certificados comodín puede ayudar, pero no resolverá todos los problemas. Esta carga adicional significa que buscará usar esta técnica cuando esté especialmente preocupado por la confidencialidad de los datos que se envían, o si está enviando datos a través de redes que no controla por completo. Por lo tanto, puede decidir proteger la comunicación de datos muy importantes entre partes que se envían a través de Internet, por ejemplo.

HMAC sobre HTTP

Como ya comentamos antes, el uso de la autenticación básica en lugar del protocolo HTTP simple no es demasiado sensato si nos preocupa que el nombre de usuario y la contraseña se vean comprometidos. La alternativa tradicional es enrutar el tráfico a través de HTTPS, pero tiene algunas desventajas. Además de la gestión de los certificados, la sobrecarga del tráfico HTTPS puede suponer una carga adicional para los servidores (aunque, para ser sinceros, esto tiene un impacto menor que hace varios años) y el tráfico no se puede almacenar en caché fácilmente.

Un enfoque alternativo, utilizado ampliamente en las API S3 de Amazon para AWS y en partes de la especificación OAuth, es utilizar un código de mensajería basado en hash (HMAC) para firmar la solicitud.

Con HMAC, el cuerpo de la solicitud junto con una clave privada se codifican mediante hash y el hash resultante se envía junto con la solicitud. Luego, el servidor utiliza su propia copia de la clave privada y el cuerpo de la solicitud para recrear el hash. Si coincide, permite la solicitud. Lo bueno de esto es que si alguien en el medio se mete con la solicitud, el hash no coincidirá y el servidor sabrá que la solicitud ha sido alterada. Además, la clave privada nunca se envía en la solicitud, por lo que no se puede comprometer en tránsito. El beneficio adicional es que este tráfico se puede almacenar en caché más fácilmente y la sobrecarga de generar los hashes puede ser menor que la de manejar el tráfico HTTPS (aunque su experiencia puede variar).

Este enfoque tiene tres desventajas. En primer lugar, tanto el cliente como el servidor necesitan un secreto compartido que debe comunicarse de alguna manera. ¿Cómo lo comparten? Podría estar codificado en ambos extremos, pero entonces tendría el problema de revocar el acceso si el secreto se ve comprometido. Si comunica esta clave a través de algún protocolo alternativo, entonces debe asegurarse de que ese protocolo también sea muy seguro.

En segundo lugar, se trata de un patrón, no de un estándar, y por lo tanto existen diferentes formas de implementarlo. Como resultado, hay una escasez de implementaciones buenas, abiertas y utilizables de este enfoque. En general, si este enfoque le interesa, lea un poco más para comprender las diferentes formas en que se lleva a cabo. Me atrevería a decir que observe cómo lo hace Amazon para S3 y copie su enfoque, especialmente utilizando una función hash sensata con una clave lo suficientemente larga como SHA-256. [Tokens web JSON \(JWT\)](#) También vale la pena echarle un vistazo, ya que implementan un enfoque muy similar y parecen estar ganando terreno. Pero tenga en cuenta la dificultad de hacer bien las cosas. Mi colega estaba trabajando con un equipo que estaba implementando su propia implementación de JWT, omitió una única comprobación booleana e invalidó todo su código de autenticación. Con suerte, con el tiempo veremos más implementaciones de bibliotecas reutilizables.

Por último, tenga en cuenta que este enfoque solo garantiza que ningún tercero haya manipulado la solicitud y que la clave privada en sí misma siga siendo privada. El resto de los datos de la solicitud seguirán siendo visibles para quienes espíen la red.

Claves API

Todas las API públicas de servicios como Twitter, Google, Flickr y AWS utilizan claves API. Las claves API permiten que un servicio identifique quién realiza una llamada y establezca límites a lo que puede hacer. A menudo, los límites van más allá de simplemente dar acceso a un recurso y pueden extenderse a acciones como limitar la velocidad de llamadas de personas específicas para proteger la calidad del servicio para otras personas.

Cuando se trata de usar claves API para gestionar su propio enfoque de microservicio a microservicio, la mecánica exacta de cómo funciona dependerá de la tecnología que utilice.

Algunos sistemas utilizan una única clave API compartida y utilizan un enfoque similar al de HMAC, como se acaba de describir. Un enfoque más común es utilizar un par de claves pública y privada.

Por lo general, se administran las claves de forma centralizada, de la misma manera que se administran las identidades de las personas de forma centralizada. El modelo de puerta de enlace es muy popular en este ámbito.

Parte de su popularidad se debe al hecho de que las claves API se centran en la facilidad de uso para los programas. En comparación con el manejo de un protocolo de enlace SAML, la autenticación basada en claves API es mucho más sencilla y directa.

Las capacidades exactas de los sistemas varían y existen múltiples opciones tanto en el ámbito comercial como en el de código abierto. Algunos de los productos solo se encargan del intercambio de claves API y de la gestión básica de claves. Otras herramientas ofrecen todo lo necesario, como limitación de velocidad, monetización, catálogos API y sistemas de descubrimiento.

Algunos sistemas API le permiten conectar claves API a servicios de directorio existentes. Esto le permitiría emitir claves API a los principales (que representan a personas o sistemas) de su organización y controlar el ciclo de vida de esas claves de la misma manera que administraría sus credenciales normales. Esto abre la posibilidad de permitir el acceso a sus servicios de diferentes maneras, pero manteniendo la misma fuente de verdad; por ejemplo, utilizando SAML para autenticar a los humanos para el inicio de sesión único y utilizando claves API para la comunicación de servicio a servicio, como se muestra en [la Figura 9-2](#).

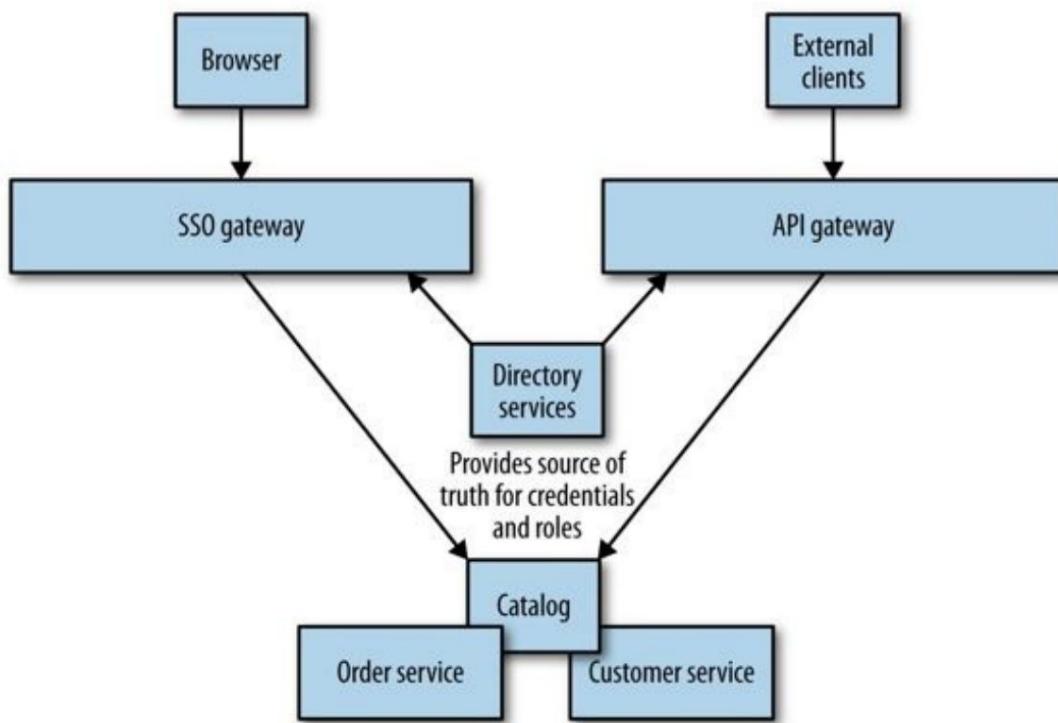


Figura 9-2. Uso de servicios de directorio para sincronizar información principal entre un SSO y una puerta de enlace API

El problema del diputado

Es bastante sencillo que un principal se autentique con un microservicio determinado. Pero, ¿qué sucede si ese servicio necesita realizar llamadas adicionales para completar una operación? Observe la Figura 9-3, que ilustra el sitio de compras en línea de MusicCorp. Nuestra tienda en línea es una interfaz de usuario de JavaScript basada en navegador. Realiza llamadas a una aplicación de tienda del lado del servidor, utilizando el patrón backends-for-frontends que describimos en el Capítulo 4. Las llamadas realizadas entre el navegador y el servidor se pueden autenticar utilizando SAML u OpenID Connect o algo similar.

Hasta ahora, todo bien.

Cuando estoy conectado, puedo hacer clic en un enlace para ver los detalles de un pedido. Para mostrar la información, necesitamos recuperar el pedido original del servicio de pedidos, pero también queremos buscar información de envío para el pedido. Entonces, hacer clic en el enlace a /orderStatus/12345 hace que la tienda en línea inicie una llamada desde el servicio de la tienda en línea tanto al servicio de pedidos como al servicio de envío solicitando esos detalles. Pero, ¿deberían estos servicios posteriores aceptar las llamadas de la tienda en línea? Podríamos adoptar una postura de confianza implícita: que debido a que la llamada proviene de nuestro perímetro, está bien. Incluso podríamos usar certificados o claves API para confirmar que sí, realmente es la tienda en línea la que solicita esta información. Pero, ¿es esto suficiente?

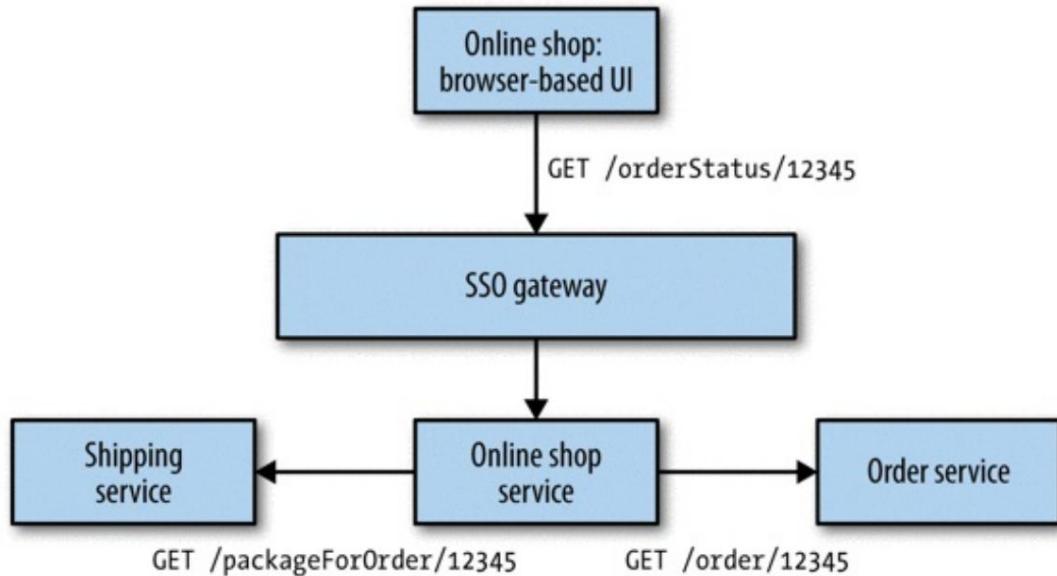


Figura 9-3. Un ejemplo en el que un diputado confundido podría entrar en juego

Existe un tipo de vulnerabilidad denominada problema de subordinado confundido, que en el contexto de la comunicación entre servicios se refiere a una situación en la que un tercero malintencionado puede engañar a un servicio subordinado para que realice llamadas a un servicio posterior en su nombre que no debería poder realizar. Por ejemplo, como cliente, cuando inicio sesión en el sistema de compras en línea, puedo ver los detalles de mi cuenta. ¿Qué sucedería si pudiera engañar a la interfaz de usuario de compras en línea para que solicite los detalles de otra persona, tal vez haciendo una llamada con mis credenciales de inicio de sesión?

En este ejemplo, ¿qué me impide pedir pedidos que no son míos? Una vez que haya iniciado sesión, podría comenzar a enviar solicitudes de otros pedidos que no sean míos para ver si puedo obtenerlos.

Información útil. Podríamos intentar protegernos de esto dentro de la propia tienda online, comprobando a quién va dirigido el pedido y rechazándolo si alguien pide cosas que no debería.

Sin embargo, si tenemos muchas aplicaciones diferentes que muestran esta información, potencialmente podríamos estar duplicando esta lógica en muchos lugares.

Podríamos enrutar solicitudes directamente desde la interfaz de usuario al servicio de pedidos y permitirle validar la solicitud, pero luego nos encontramos con las diversas desventajas que discutimos en [el Capítulo 4](#).

Como alternativa, cuando la tienda online envía la solicitud al servicio de pedidos, podría indicar no solo qué pedido desea, sino también en nombre de quién lo solicita. Algunos esquemas de autenticación nos permiten pasar las credenciales del principal original en sentido descendente, aunque con SAML esto es una pesadilla, ya que implica afirmaciones SAML anidadas que son técnicamente alcanzables, pero tan difíciles que nadie lo hace nunca. Esto puede volverse aún más complejo, por supuesto. Imaginemos si los servicios con los que habla la tienda online a su vez realizan más llamadas en sentido descendente. ¿Hasta dónde tenemos que llegar para validar la confianza de todos esos delegados?

Lamentablemente, este problema no tiene una solución sencilla, porque no es un problema sencillo. Sin embargo, tenga en cuenta que existe. Según la sensibilidad de la operación en cuestión, es posible que tenga que elegir entre la confianza implícita, verificar la identidad de quien llama o pedirle que proporcione las credenciales del principal original.

Protección de datos en reposo

Los datos que se encuentran en el aire son un riesgo, especialmente si son confidenciales. Esperamos haber hecho todo lo posible para garantizar que los atacantes no puedan vulnerar nuestra red, ni nuestras aplicaciones o sistemas operativos para acceder a los datos subyacentes. Sin embargo, debemos estar preparados en caso de que lo hagan: la defensa en profundidad es fundamental.

Muchas de las violaciones de seguridad de alto perfil implican que un atacante obtiene datos en reposo y los puede leer, ya sea porque los datos se almacenaron sin cifrar o porque el mecanismo utilizado para protegerlos tenía una falla fundamental.

Los mecanismos mediante los cuales se puede proteger la información segura son muchos y variados, pero cualquiera sea el enfoque elegido hay algunas cosas generales a tener en cuenta.

Vaya con lo conocido

La forma más fácil de estropear el cifrado de datos es intentar implementar algoritmos de cifrado propios o incluso intentar implementar los de otra persona. Sea cual sea el lenguaje de programación que utilice, tendrá acceso a implementaciones revisadas y parcheadas periódicamente de algoritmos de cifrado de buena reputación. ¡Úselos! Y suscríbase a las listas de correo o listas de asesoramiento de la tecnología que elija para asegurarse de estar al tanto de las vulnerabilidades a medida que se encuentren, de modo que pueda mantenerlas parcheadas y actualizadas.

Para el cifrado en reposo, a menos que tenga una muy buena razón para elegir otra cosa, [3](#) elija una conocida de AES-128 o AES-256 para su plataforma. Los entornos de ejecución de Java y .NET incluyen implementaciones de AES que es muy probable que estén bien probadas (y bien parcheadas), pero también existen bibliotecas independientes para la mayoría de las plataformas, por ejemplo, las [bibliotecas Bouncy Castle](#) para Java y C#.

Para las contraseñas, debería considerar utilizar una técnica llamada [hash de contraseñas salado](#).

Un cifrado mal implementado podría ser peor que no tener ninguno, ya que la falsa sensación de seguridad (perdón por el juego de palabras) puede hacer que usted pierda de vista el objetivo.

Todo es cuestión de las llaves

Como se ha explicado hasta ahora, el cifrado se basa en un algoritmo que toma los datos que se van a cifrar y una clave y luego genera los datos cifrados. Entonces, ¿dónde se almacena la clave? Ahora bien, si estoy cifrando mis datos porque me preocupa que alguien robe toda mi base de datos y guardo la clave que uso en la misma base de datos, entonces no he logrado mucho. Por lo tanto, necesitamos almacenar las claves en otro lugar. ¿Pero dónde?

Una solución es utilizar un dispositivo de seguridad independiente para cifrar y descifrar los datos. Otra es utilizar un almacén de claves independiente al que su servicio pueda acceder cuando necesite una clave. La gestión del ciclo de vida de las claves (y el acceso para cambiarlas) puede ser una operación vital, y estos sistemas pueden encargarse de ello por usted.

Algunas bases de datos incluso incluyen compatibilidad integrada con el cifrado, como el cifrado de datos transparente de SQL Server, que tiene como objetivo gestionar este problema de forma transparente. Incluso si la base de datos que ha elegido lo hace, investigue cómo se gestionan las claves y comprenda si la amenaza contra la que se está protegiendo realmente se está mitigando.

Nuevamente, se trata de un tema complejo. ¡Evita implementarlo tú mismo y realiza una buena investigación!

Elige tus objetivos

Suponer que todo debe estar cifrado puede simplificar un poco las cosas. No hay que adivinar qué se debe proteger o no. Sin embargo, aún hay que pensar en qué datos se pueden incluir en los archivos de registro para ayudar a identificar problemas, y la sobrecarga computacional de cifrar todo puede volverse bastante onerosa, lo que requiere un hardware más potente. Esto es aún más complicado cuando se aplican migraciones de bases de datos como parte de esquemas de refactorización. Según los cambios que se realicen, es posible que sea necesario descifrar, migrar y volver a cifrar los datos.

Si subdivide el sistema en servicios más específicos, es posible que identifique un almacén de datos completo que se pueda cifrar en su totalidad, pero incluso en ese caso es poco probable. Limitar este cifrado a un conjunto conocido de tablas es un enfoque sensato.

Descifrado a pedido

Cifre los datos cuando los vea por primera vez. Descifre los datos solo cuando lo necesite y asegúrese de que nunca se almacenen en ningún lugar.

Cifrar copias de seguridad

Las copias de seguridad son buenas. Queremos hacer copias de seguridad de nuestros datos importantes y, casi por definición, los datos que nos preocupan lo suficiente como para querer cifrarlos son lo suficientemente importantes como para hacer una copia de seguridad. Puede parecer obvio, pero debemos asegurarnos de que nuestras copias de seguridad también estén cifradas. Esto también significa que debemos saber qué claves son necesarias para gestionar qué versión de los datos, especialmente si las claves cambian. Tener una gestión clara de las claves se vuelve bastante importante.

Defensa en profundidad

Como ya he mencionado antes, no me gusta poner todos los huevos en la misma canasta. Lo importante es la defensa en profundidad. Ya hemos hablado de proteger los datos en tránsito y los datos en reposo, pero ¿existen otras protecciones que podríamos implementar para ayudar?

Cortafuegos

Tener uno o más cortafuegos es una precaución muy sensata. Algunos son muy simples y solo pueden restringir el acceso a ciertos tipos de tráfico en ciertos puertos. Otros son más sofisticados. ModSecurity, por ejemplo, es un tipo de cortafuegos de aplicaciones que puede ayudar a limitar las conexiones desde ciertos rangos de IP y detectar otros tipos de ataques maliciosos.

Es útil tener más de un firewall. Por ejemplo, puede decidir usar IPTables localmente en un host para protegerlo y configurar el ingreso y egreso permitidos.

Estas reglas podrían adaptarse a los servicios que se ejecutan localmente, con un firewall en el perímetro para controlar el acceso general.

Explotación forestal

Un buen registro, y específicamente la capacidad de agregar registros de múltiples sistemas, no tiene como objetivo la prevención, sino que puede ayudar a detectar y recuperarse de situaciones malas.

Por ejemplo, después de aplicar parches de seguridad, a menudo se puede ver en los registros si alguien ha estado explotando ciertas vulnerabilidades. La aplicación de parches garantiza que no vuelva a suceder, pero si ya sucedió , es posible que deba pasar al modo de recuperación. Tener registros disponibles le permite ver si sucedió algo malo después del hecho.

Sin embargo, tenga en cuenta que debemos tener cuidado con la información que almacenamos en nuestros registros. Es necesario seleccionar información confidencial para garantizar que no estemos filtrando datos importantes en nuestros registros, que podrían terminar siendo un gran objetivo para los atacantes.

Sistema de detección (y prevención) de intrusiones

Los sistemas de detección de intrusiones (IDS) pueden monitorear redes o hosts en busca de comportamientos sospechosos e informar problemas cuando los detectan. Los sistemas de prevención de intrusiones (IPS), además de monitorear actividades sospechosas, pueden intervenir para evitar que sucedan. A diferencia de un firewall, que principalmente mira hacia afuera para evitar que entren cosas malas, los IDS y los IPS buscan activamente dentro del perímetro en busca de comportamientos sospechosos. Cuando comienza desde cero, los IDS pueden ser lo más lógico. Estos sistemas se basan en heurística (como muchos firewalls de aplicaciones) y es posible que el conjunto de reglas iniciales genéricas sea demasiado indulgente o no lo suficientemente indulgente para el comportamiento de su servicio. Usar un IDS más pasivo para alertarlo sobre problemas es una buena forma de ajustar sus reglas antes de usarlo de manera más activa.

Segregación de la red

Con un sistema monolítico, tenemos límites en cuanto a cómo podemos estructurar nuestras redes para brindar protecciones adicionales. Sin embargo, con los microservicios, puede colocarlos en diferentes segmentos de red para controlar aún más cómo se comunican los servicios entre sí. AWS, por ejemplo, brinda la capacidad de aprovisionar automáticamente una nube privada virtual (VPC), que permite que los hosts vivan en subredes separadas. Luego, puede especificar qué VPC pueden verse entre sí definiendo reglas de emparejamiento e incluso enrutar el tráfico a través de puertas de enlace para el acceso proxy, lo que le brinda múltiples perímetros en los que se pueden implementar medidas de seguridad adicionales.

Esto podría permitirle segmentar redes según la propiedad del equipo o quizás por nivel de riesgo.

Sistema operativo

Nuestros sistemas dependen de una gran cantidad de software que no hemos escrito nosotros y que puede tener vulnerabilidades de seguridad que podrían exponer nuestra aplicación, es decir, nuestros sistemas operativos y las demás herramientas de soporte que ejecutamos en ellos. En este caso, unos consejos básicos pueden ser de gran ayuda. Comience ejecutando únicamente servicios como usuarios del sistema operativo que tengan la menor cantidad de permisos posible, para garantizar que si dicha cuenta se ve comprometida, causará un daño mínimo.

A continuación, aplique parches a su software. Con regularidad. Esto debe automatizarse y usted necesita saber si sus máquinas no están sincronizadas con los últimos niveles de parches. Herramientas como SCCM de Microsoft o Spacewalk de RedHat pueden ser útiles en este caso, ya que pueden ayudarlo a ver si las máquinas están actualizadas con los últimos parches e iniciar actualizaciones si es necesario. Si utiliza herramientas como Ansible, Puppet o Chef, es probable que ya esté bastante satisfecho con la implementación automática de cambios; estas herramientas también pueden ayudarlo mucho, pero no lo harán todo por usted.

En realidad, se trata de algo básico, pero es sorprendente la frecuencia con la que veo software crítico ejecutándose en sistemas operativos antiguos sin parches. Puede tener la seguridad a nivel de aplicación mejor definida y protegida del mundo, pero si tiene una versión antigua de un servidor web ejecutándose en su máquina como root que tiene una vulnerabilidad de desbordamiento de búfer sin parches, entonces su sistema podría seguir siendo extremadamente vulnerable.

Otra cosa que hay que tener en cuenta si se utiliza Linux es la aparición de módulos de seguridad para el propio sistema operativo. AppArmour, por ejemplo, permite definir cómo se espera que se comporte la aplicación, mientras el núcleo la vigila. Si empieza a hacer algo que no debería, el núcleo interviene. AppArmour existe desde hace tiempo, al igual que SELinux. Aunque técnicamente cualquiera de ellos debería funcionar en cualquier sistema Linux moderno, en la práctica algunas distribuciones admiten uno mejor que el otro. AppArmour se utiliza de forma predeterminada en Ubuntu y SuSE, por ejemplo, mientras que SELinux ha contado tradicionalmente con un buen soporte de RedHat. Una opción más nueva es GrSecurity, que pretende ser más sencilla de utilizar que AppArmour o GrSecurity, al tiempo que intenta ampliar sus capacidades, pero requiere un núcleo personalizado para funcionar. Sugeriría echar un vistazo a los tres para ver cuál se adapta mejor a sus casos de uso, pero me gusta la idea de tener otra capa de protección y prevención en funcionamiento.

Un ejemplo práctico

Tener una arquitectura de sistema más detallada nos da mucha más libertad en la forma en que implementamos nuestra seguridad. Para aquellas partes que manejan la información más sensible o exponen las capacidades más valiosas, podemos adoptar las disposiciones de seguridad más estrictas. Pero para otras partes del sistema, podemos permitirnos ser mucho más laxos en lo que nos preocupa.

Consideremos nuevamente MusicCorp y juntemos algunos de los conceptos anteriores para ver dónde y cómo podemos usar algunas de estas técnicas de seguridad. Estamos considerando principalmente las preocupaciones de seguridad de los datos en tránsito y en reposo. La Figura 9-4 muestra un subconjunto del sistema general que analizaremos, que actualmente muestra una aplastante falta de consideración por las preocupaciones de seguridad. Todo se envía a través del viejo y simple HTTP.

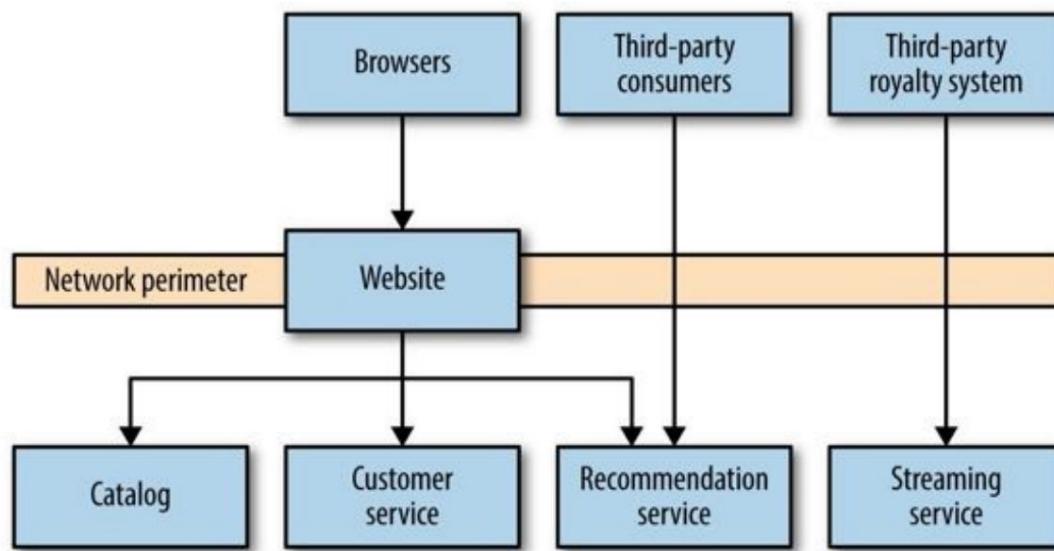


Figura 9-4. Un subconjunto de la arquitectura lamentablemente insegura de MusicCorp

Aquí tenemos navegadores web estándar que utilizan nuestros clientes para comprar en el sitio.

También presentamos el concepto de una pasarela de regalías de terceros: hemos comenzado a trabajar con una empresa externa que se encargará de los pagos de regalías por nuestro nuevo servicio de streaming. De vez en cuando se pone en contacto con nosotros para obtener registros de qué música se ha reproducido y cuándo, información que protegemos celosamente porque nos preocupa la competencia de empresas rivales. Por último, exponemos los datos de nuestro catálogo a otros terceros, por ejemplo, permitiendo que los metadatos sobre el artista o la canción se incorporen en sitios de reseñas musicales. Dentro del perímetro de nuestra red, tenemos algunos servicios colaboradores, que solo se utilizan internamente.

Para el navegador, utilizaremos una combinación de tráfico HTTP estándar para contenido no seguro, a fin de permitir su almacenamiento en caché. Para páginas seguras en las que se haya iniciado sesión, todo el contenido seguro se enviará a través de HTTPS, lo que les dará a nuestros clientes protección adicional si realizan tareas como ejecutarlas en redes WiFi públicas.

Cuando se trata del sistema de pago de regalías de terceros, nos preocupamos no solo por la naturaleza de los datos que estamos exponiendo, sino también por asegurarnos de que las solicitudes que estamos

Los datos que obtenemos son legítimos. En este caso, insistimos en que nuestro tercero utilice certificados de cliente. Todos los datos se envían a través de un canal criptográfico seguro, lo que aumenta nuestra capacidad para garantizar que la persona adecuada nos los solicita. Por supuesto, tenemos que pensar en lo que sucede cuando los datos salen de nuestro control. ¿Nuestro socio se preocupará por los datos tanto como nosotros?

En el caso de los feeds de datos del catálogo, queremos que esta información se comparta lo más ampliamente posible para que la gente pueda comprarnos música fácilmente. Sin embargo, no queremos que se abuse de esto y nos gustaría tener una idea de quién está usando nuestros datos. En este caso, las claves API tienen todo el sentido.

Dentro del perímetro de la red, las cosas son un poco más matizadas. ¿Qué tan preocupados estamos de que la gente ponga en peligro nuestras redes internas? Lo ideal sería utilizar HTTPS como mínimo, pero gestionarlo es algo complicado. En cambio, decidimos dedicar el trabajo (al menos inicialmente) a reforzar nuestro perímetro de red, lo que incluye tener un cortafuegos configurado correctamente y seleccionar un dispositivo de seguridad de hardware o software adecuado para comprobar si hay tráfico malicioso (por ejemplo, escaneo de puertos o ataques de denegación de servicio).

Dicho esto, nos preocupan algunos de nuestros datos y dónde se almacenan. No nos preocupa el servicio de catálogo; después de todo, queremos que esos datos se compartan y hemos proporcionado una API para ello. Pero nos preocupan mucho los datos de nuestros clientes. En este caso, decidimos cifrar los datos que tiene el servicio de atención al cliente y descifrarlos al leerlos. Si los atacantes penetran en nuestra red, aún podrían ejecutar solicitudes contra la API del servicio de atención al cliente, pero la implementación actual no permite la recuperación masiva de datos de clientes. Si lo hiciera, probablemente consideraríamos el uso de certificados de cliente para proteger esta información. Incluso si los atacantes comprometen la máquina en la que se ejecuta la base de datos y logran descargar todo el contenido, necesitarían acceso a la clave utilizada para cifrar y descifrar los datos para poder utilizarla.

La figura 9-5 muestra la imagen final. Como puede ver, las decisiones que tomamos sobre qué tecnología utilizar se basaron en una comprensión de la naturaleza de la información que se desea proteger. Es probable que las preocupaciones de seguridad de su propia arquitectura sean muy diferentes, por lo que puede terminar con una solución de aspecto diferente.

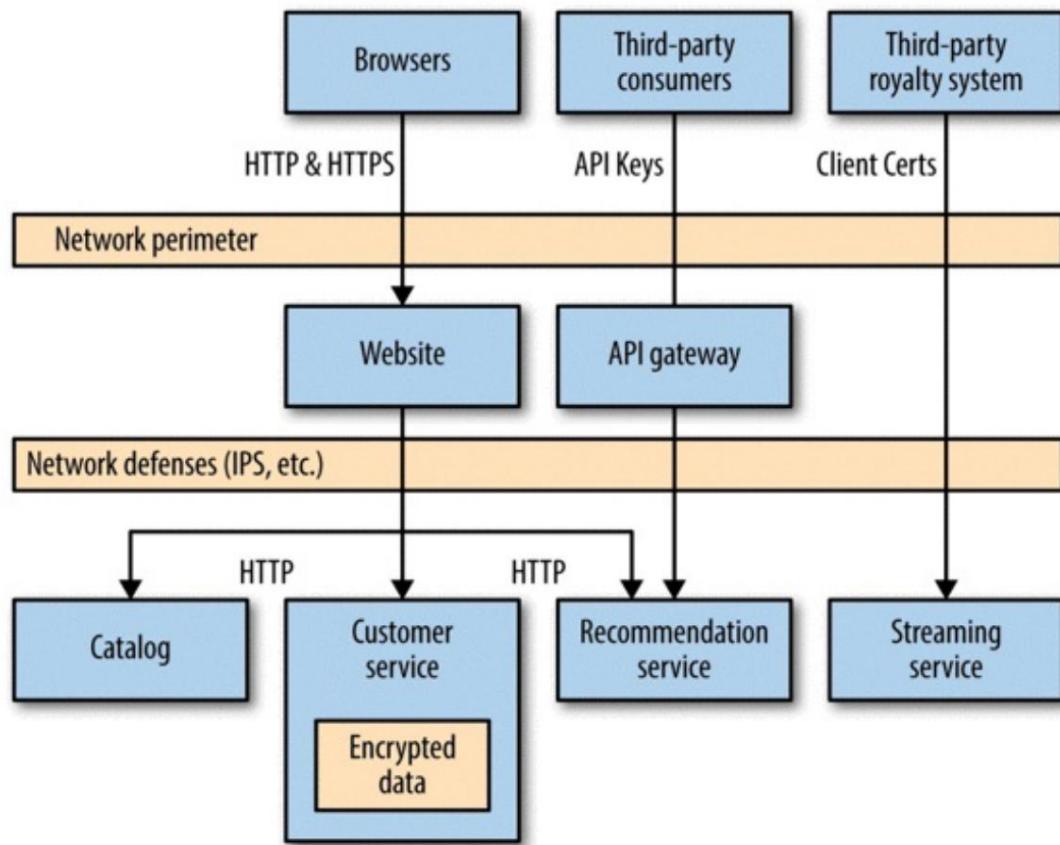


Figura 9-5. El sistema más seguro de MusicCorp

Sea frugal

A medida que el espacio en disco se vuelve más barato y las capacidades de las bases de datos mejoran, la facilidad con la que se pueden capturar y almacenar grandes cantidades de información está mejorando rápidamente. Estos datos son valiosos, no solo para las propias empresas, que cada vez más los consideran un activo valioso, sino también para los usuarios, que valoran su propia privacidad. Los datos que pertenecen a una persona, o que podrían utilizarse para obtener información sobre una persona, son los datos que debemos tratar con más cuidado.

Sin embargo, ¿qué pasaría si nos facilitáramos un poco la vida? ¿Por qué no eliminar la mayor cantidad posible de información que pueda ser personalmente identificable y hacerlo lo antes posible? Al registrar una solicitud de un usuario, ¿necesitamos almacenar la dirección IP completa para siempre o podríamos reemplazar los últimos dígitos por x? ¿Necesitamos almacenar el nombre, la edad, el sexo y la fecha de nacimiento de alguien para ofrecerle ofertas de productos o es suficiente con su rango de edad y código postal?

Las ventajas son múltiples. En primer lugar, si no lo almacenas, nadie puede robarlo. En segundo lugar, si no lo almacenas, nadie (por ejemplo, una agencia gubernamental) puede pedirlo.

La frase alemana *Datensparsamkeit* representa este concepto. Proviene de la legislación alemana sobre privacidad y engloba el concepto de almacenar únicamente la información que sea absolutamente necesaria para llevar a cabo las operaciones comerciales o cumplir con las leyes locales.

Obviamente, esto está en tensión directa con el movimiento hacia almacenar cada vez más información, ipero es un comienzo darse cuenta de que esta tensión existe!

El elemento humano

Gran parte de lo que hemos cubierto aquí son los conceptos básicos sobre cómo implementar salvaguardas tecnológicas para proteger sus sistemas y datos de atacantes externos maliciosos. Sin embargo, es posible que también necesite procesos y políticas implementados para lidiar con el elemento humano en su organización. ¿Cómo revoca el acceso a las credenciales cuando alguien abandona la organización? ¿Cómo puede protegerse contra la ingeniería social? Como buen ejercicio mental, considere el daño que un ex empleado descontento podría causar a sus sistemas si quisiera. Ponerse en el lugar de una parte maliciosa suele ser una buena manera de razonar sobre las protecciones que puede necesitar, ¡y pocas partes maliciosas tienen tanta información privilegiada como un empleado reciente!

La regla de oro

Si no hay nada más que pueda sacar de este capítulo, que sea esto: no escriba su propia criptografía. No invente sus propios protocolos de seguridad. A menos que sea un experto en criptografía con años de experiencia, si intenta inventar su propia codificación o elaborar protecciones criptográficas, se equivocará. E incluso si es un experto en criptografía, puede que se equivoque.

Muchas de las herramientas descritas anteriormente, como AES, son tecnologías robustas en la industria cuyos algoritmos subyacentes han sido revisados por pares y cuya implementación de software ha sido rigurosamente probada y parcheada durante muchos años. ¡Son lo suficientemente buenas! Reinventar la rueda en muchos casos suele ser solo una pérdida de tiempo, pero cuando se trata de seguridad puede ser absolutamente peligroso.

Seguridad en la cocción

Al igual que con las pruebas funcionales automatizadas, no queremos que la seguridad quede en manos de un grupo diferente de personas, ni queremos dejar todo para el último momento. Ayudar a educar a los desarrolladores sobre las preocupaciones de seguridad es clave, ya que aumentar la conciencia general de todos sobre los problemas de seguridad puede ayudar a reducirlos en primer lugar. Hacer que la gente se familiarice con la lista de los diez principales problemas de seguridad de OWASP y el marco de pruebas de seguridad de OWASP puede ser un buen punto de partida. Sin embargo, los especialistas tienen su lugar, y si tiene acceso a ellos, úselos para que lo ayuden.

Existen herramientas automatizadas que pueden investigar nuestros sistemas en busca de vulnerabilidades, como por ejemplo, buscando ataques de secuencias de comandos entre sitios. El Zed Attack Proxy (también conocido como ZAP) es un buen ejemplo. Basándose en el trabajo de OWASP, ZAP intenta recrear ataques maliciosos en su sitio web. Existen otras herramientas que utilizan el análisis estático para buscar errores de codificación comunes que pueden abrir agujeros de seguridad, como [Brakeman](#). para Ruby. Si estas herramientas se pueden integrar fácilmente en compilaciones de CI normales, intégralas en sus registros estándar. Otros tipos de pruebas automatizadas son más complejas. Por ejemplo, usar algo como Nessus para buscar vulnerabilidades es un poco más complejo y puede requerir que un humano interprete los resultados. Dicho esto, estas pruebas aún se pueden automatizar y puede tener sentido ejecutarlas con la misma cadencia que las pruebas de carga.

[Ciclo de vida del desarrollo de seguridad](#) de Microsoft También tiene algunos buenos modelos sobre cómo los equipos de entrega pueden incorporar la seguridad. Algunos aspectos parecen demasiado en cascada, pero échale un vistazo y ve qué aspectos pueden encajar en tu flujo de trabajo actual.

Verificación externa

En materia de seguridad, creo que es muy valioso contar con una evaluación externa. Ejercicios como las pruebas de penetración, cuando las realiza un tercero, realmente imitan los intentos del mundo real.

También evitan el problema de que los equipos no siempre pueden ver los errores que han cometido ellos mismos, ya que están demasiado cerca del problema. Si eres una empresa lo suficientemente grande, es posible que tengas un equipo de seguridad de la información dedicado que pueda ayudarte. Si no es así, busca un tercero que pueda hacerlo. Ponte en contacto con ellos lo antes posible, entiende cómo les gusta trabajar y averigua con cuánta antelación deben realizar una prueba.

También deberás considerar cuánta verificación necesitarás antes de cada lanzamiento.

En general, no es necesario realizar una prueba de penetración completa, por ejemplo, para pequeñas versiones incrementales, pero sí para cambios más grandes. Lo que necesite dependerá de su propio perfil de riesgo.

Resumen

Así que volvemos a un tema central del libro: tener un sistema descompuesto en servicios más detallados nos brinda muchas más opciones para resolver un problema. Tener microservicios no solo puede reducir potencialmente el impacto de cualquier violación de seguridad, sino que también nos brinda más capacidad para compensar la sobrecarga de enfoques más complejos y seguros cuando los datos son sensibles y un enfoque más liviano cuando los riesgos son menores.

Una vez que comprenda los niveles de amenaza de las diferentes partes de su sistema, debería comenzar a tener una idea de cuándo considerar la seguridad durante el tránsito, en reposo o no considerarla en absoluto.

Por último, comprenda la importancia de la defensa en profundidad, asegúrese de parchar sus sistemas operativos e, incluso si se considera una estrella de rock, ¡no intente implementar su propia criptografía!

Si desea una descripción general de la seguridad para aplicaciones basadas en navegador, un excelente lugar para comenzar es el excelente [proyecto sin fines de lucro Open Web Application Security Project \(OWASP\)](#). El documento Top 10 Security Risk Risk , actualizado periódicamente, debería considerarse una lectura esencial para cualquier desarrollador. Por último, si desea una discusión más general sobre criptografía, consulte el libro *Cryptography Engineering* de Niels Ferguson, Bruce Schneier y Tadayoshi Kohno (Wiley).

Para comprender la seguridad, es necesario comprender a las personas y cómo trabajan con nuestros sistemas. Un aspecto relacionado con los humanos que aún no hemos analizado en relación con los microservicios es la interacción entre las estructuras organizativas y las arquitecturas en sí. Pero, al igual que con la seguridad, veremos que ignorar el elemento humano puede ser un grave error.

³ En general, la longitud de la clave aumenta la cantidad de trabajo necesario para descifrar una clave mediante fuerza bruta. Por lo tanto, se puede suponer que cuanto más larga sea la clave, más seguros estarán los datos. Sin embargo, el respetado experto en seguridad Bruce Schneier ha planteado algunas pequeñas inquietudes sobre la implementación de AES-256 para ciertos tipos de claves . ¡Ésta es una de esas áreas en las que necesitas investigar más sobre cuál es el consejo actual al momento de leer!

Capítulo 10. Ley de Conway y diseño de sistemas

Hasta ahora, gran parte del libro se ha centrado en los desafíos técnicos que supone avanzar hacia una arquitectura de grano fino, pero también hay otras cuestiones organizativas que hay que tener en cuenta. Como aprenderemos en este capítulo, ignorar el organigrama de la empresa es hacerlo a su propio riesgo!

Nuestra industria es joven y parece estar reinventándose constantemente. Y, sin embargo, algunas leyes clave han resistido la prueba del tiempo. La ley de Moore, por ejemplo, que establece que la densidad de transistores en circuitos integrados se duplica cada dos años, ha demostrado ser asombrosamente precisa (aunque algunas personas predicen que esta tendencia ya se está desacelerando). Una ley que he encontrado casi universalmente cierta, y mucho más útil en mi trabajo diario, es la ley de Conway.

El artículo de Melvin Conway How Do Committees Invent, publicado en la revista Datamation en abril de 1968, observó que:

Cualquier organización que diseñe un sistema (definido aquí de manera más amplia que simplemente sistemas de información) inevitablemente producirá un diseño cuya estructura sea una copia de la estructura de comunicación de la organización.

Esta afirmación se cita a menudo, en diversas formas, como la ley de Conway. Eric S. Raymond resumió este fenómeno en The New Hacker's Dictionary (MIT Press) diciendo: "Si tienes cuatro grupos trabajando en un compilador, obtendrás un compilador de 4 pasadas".

Evidencia

Se dice que cuando Melvin Conway presentó su artículo sobre este tema a la Harvard Business Review, lo rechazaron, alegando que no había demostrado su tesis. He visto esta teoría confirmada en tantas situaciones diferentes que la he aceptado como cierta. Pero no tienen por qué creerme a mí: desde la presentación original de Conway, se ha trabajado mucho en este ámbito. Se han llevado a cabo varios estudios para explorar la interrelación entre la estructura organizacional y los sistemas que crean.

Organizaciones flexibles y fuertemente acopladas

En su artículo Exploring the Duality Between Product and Organizational Architectures (Harvard Business School), los autores Alan MacCormack, John Rusnak y Carliss Baldwin analizan una serie de sistemas de software diferentes, categorizados de forma general como creados por organizaciones débilmente acopladas o fuertemente acopladas. En el caso de las organizaciones fuertemente acopladas, pensemos en empresas de productos comerciales que suelen estar ubicadas en el mismo lugar con visiones y objetivos fuertemente alineados, mientras que las organizaciones débilmente acopladas están bien representadas por comunidades de código abierto distribuidas.

En su estudio, en el que combinaron pares de productos similares de cada tipo de organización, los autores descubrieron que las organizaciones menos acopladas en realidad creaban sistemas más modulares y menos acoplados, mientras que el software de las organizaciones más estrechamente enfocadas estaba menos modularizado.

Windows Vista

Microsoft realizó **un estudio empírico** En este estudio, se analizó el impacto de la propia estructura organizativa en la calidad del software de un producto específico, Windows Vista. En concreto, los investigadores analizaron múltiples factores para determinar la propensión a errores de un incluidas las métricas componente del sistema. **4** Después de analizar múltiples métricas, de calidad del software de uso común, como la complejidad del código, descubrieron que las métricas asociadas con las estructuras organizativas resultaron ser las medidas estadísticamente más relevantes.

Así que aquí tenemos otro ejemplo de cómo la estructura organizacional impacta la naturaleza del sistema que esa organización crea.

Netflix y Amazon

Probablemente los dos ejemplos más representativos de la idea de que las organizaciones y la arquitectura deben estar alineadas son Amazon y Netflix. Desde el principio, Amazon empezó a comprender los beneficios de que los equipos fueran dueños de todo el ciclo de vida de los sistemas que gestionaban. Quería que los equipos fueran dueños y operaran los sistemas que ellos cuidaban, gestionando todo el ciclo de vida. Pero Amazon también sabía que los equipos pequeños pueden trabajar más rápido que los equipos grandes. Esto condujo a sus famosos equipos de dos pizzas, donde ningún equipo debería ser tan grande que no pudiera ser alimentado con dos pizzas. Este impulso para que los equipos pequeños sean dueños de todo el ciclo de vida de sus servicios es una de las principales razones por las que Amazon desarrolló Amazon Web Services. Necesitaba crear las herramientas que permitieran a sus equipos ser autosuficientes.

Netflix aprendió de este ejemplo y se aseguró de que desde el principio se estructurara en torno a equipos pequeños e independientes, de modo que los servicios que crearan también fueran independientes entre sí. Esto garantizó que la arquitectura del sistema estuviera optimizada para la velocidad de los cambios. En efecto, Netflix diseñó la estructura organizativa para la arquitectura de sistema que deseaba.

¿Qué podemos hacer con esto?

Así pues, la evidencia, tanto anecdótica como empírica, indica que nuestra estructura organizacional tiene una fuerte influencia en la naturaleza (y calidad) de los sistemas que ofrecemos. ¿Cómo nos ayuda entonces esta comprensión? Analicemos algunas situaciones organizacionales diferentes y entendamos qué impacto podría tener cada una de ellas en el diseño de nuestro sistema.

Adaptación a las vías de comunicación

Consideremos primero un equipo simple y único. Está a cargo de todos los aspectos del diseño y la implementación del sistema. Puede tener una comunicación frecuente y detallada. Imaginemos que este equipo está a cargo de un solo servicio, por ejemplo, el servicio de catálogo de nuestra tienda de música. Ahora, pensemos en el interior de un servicio: muchas llamadas a métodos o funciones detalladas. Como hemos comentado antes, nuestro objetivo es asegurarnos de que nuestros servicios se descompongan de manera tal que el ritmo de cambio dentro de un servicio sea mucho mayor que el ritmo de cambio entre servicios. Este equipo único, con su capacidad para una comunicación detallada, se adapta muy bien a las vías de comunicación del código dentro del servicio.

A este equipo único le resulta fácil comunicarse sobre los cambios y refactorizaciones propuestos y, por lo general, tiene un buen sentido de pertenencia.

Ahora imaginemos un escenario diferente. En lugar de que un único equipo geolocalizado sea el propietario de nuestro servicio de catálogo, supongamos que los equipos del Reino Unido y la India participan activamente en el cambio de un servicio, lo que significa que tienen una propiedad conjunta del servicio. Los límites geográficos y de zona horaria dificultan la comunicación detallada entre esos equipos. En cambio, dependen de una comunicación más general a través de videoconferencias y correo electrónico. ¿Qué tan fácil es para un miembro del equipo del Reino Unido hacer una refactorización simple con confianza? El costo de las comunicaciones en un equipo distribuido geográficamente es mayor y, por lo tanto, el costo de coordinar los cambios es mayor.

Cuando el costo de coordinar el cambio aumenta, suceden dos cosas: o bien las personas encuentran formas de reducir los costos de coordinación y comunicación, o dejan de hacer cambios.

Esto último es exactamente cómo terminamos con bases de código grandes y difíciles de mantener.

Recuerdo un proyecto para un cliente en el que trabajé, en el que la propiedad de un único servicio se compartía entre dos ubicaciones geográficas. Con el tiempo, cada sitio empezó a especializar el trabajo que manejaba. Esto le permitió asumir la propiedad de una parte del código base, dentro de la cual podía tener un costo de cambio más fácil. Los equipos entonces tenían una comunicación más general sobre cómo se interrelacionaban las dos partes; efectivamente, las vías de comunicación que se hicieron posibles dentro de la estructura organizacional coincidían con la API general que formaba el límite entre las dos mitades del código base.

¿En qué situación nos deja esto cuando consideramos la posibilidad de desarrollar nuestro propio diseño de servicios? Bueno, yo sugeriría que las fronteras geográficas entre las personas involucradas en el desarrollo de un sistema pueden ser una excelente manera de determinar cuándo se deben descomponer los servicios y que, en general, se debe intentar asignar la propiedad de un servicio a un solo equipo ubicado en el mismo lugar que pueda mantener bajo el costo del cambio.

Tal vez su organización decida que desea aumentar la cantidad de personas que trabajan en su proyecto abriendo una oficina en otro país. En este punto, piense activamente en qué partes de su sistema se pueden trasladar. Tal vez esto sea lo que impulse sus decisiones sobre qué uniones dividir a continuación.

También vale la pena señalar en este punto que, al menos según las observaciones de los autores del informe Exploración de la dualidad entre arquitecturas de producto y arquitecturas organizacionales al que se hizo referencia anteriormente, si la organización que construye el sistema está acoplada de manera más flexible (por ejemplo, si está formada por equipos distribuidos geográficamente), los sistemas que se construyen tienden a ser más modulares y, por lo tanto, es de esperar que estén menos acoplados. La tendencia de un solo equipo que posee muchos servicios a inclinarse hacia una integración más estrecha es muy difícil de mantener en una organización más distribuida.

Propiedad del servicio

¿Qué quiero decir con propiedad del servicio? En general, significa que el equipo que posee un servicio es responsable de realizar cambios en ese servicio. El equipo debe sentirse libre de reestructurar el código como quiera, siempre y cuando ese cambio no afecte a los servicios que lo consumen. Para muchos equipos, la propiedad se extiende a todos los aspectos del servicio, desde la obtención de requisitos hasta la creación, implementación y mantenimiento de la aplicación. Este modelo es especialmente frecuente en el caso de los microservicios, donde es más fácil para un equipo pequeño poseer un servicio pequeño. Este mayor nivel de propiedad conduce a una mayor autonomía y velocidad de entrega. Tener un equipo responsable de implementar y mantener la aplicación significa que tiene un incentivo para crear servicios que sean fáciles de implementar; es decir, las preocupaciones sobre "lanzar algo por encima del muro" se disipan cuando no hay nadie a quien lanzárselo.

Este modelo es sin duda uno de mis favoritos. Deja las decisiones en manos de las personas más capacitadas para tomarlas, lo que otorga al equipo mayor poder y autonomía, pero también lo hace responsable de su trabajo. He visto a demasiados desarrolladores entregar su sistema para las fases de prueba o implementación y pensar que su trabajo está hecho en ese punto.

Impulsores de los servicios compartidos

He visto a muchos equipos adoptar un modelo de propiedad compartida de servicios. Considero que este enfoque no es óptimo, por las razones que ya he comentado. Sin embargo, es importante comprender los factores que llevan a las personas a elegir los servicios compartidos, especialmente porque tal vez podamos encontrar algunos modelos alternativos convincentes que puedan abordar las preocupaciones subyacentes de las personas.

Demasiado difícil de dividir

Obviamente, una de las razones por las que puede encontrarse con un único servicio que pertenece a más de un equipo es que el coste de dividir el servicio es demasiado alto, o tal vez su organización no le vea sentido a ello. Esto es algo habitual en los grandes sistemas monolíticos.

Si este es el principal desafío al que se enfrenta, espero que algunos de los consejos que se dan en el [Capítulo 5](#) le resulten útiles. También podría considerar la posibilidad de fusionar equipos para que se alineen más estrechamente con la propia arquitectura.

Equipos destacados

La idea de los equipos de funciones (también conocidos como equipos basados en funciones) es que un equipo pequeño impulse el desarrollo de un conjunto de funciones e implemente todas las funcionalidades necesarias, incluso si trascienden los límites de los componentes (o incluso de los servicios). Los objetivos de los equipos de funciones son bastante sensatos. Esta estructura permite que el equipo mantenga el foco en el resultado final y garantiza que el trabajo esté coordinado, lo que evita algunos de los desafíos que supone intentar coordinar los cambios entre varios equipos diferentes.

En muchas situaciones, el equipo de funciones es una reacción a las organizaciones de TI tradicionales, donde la estructura del equipo se alinea en torno a límites técnicos. Por ejemplo, puede tener un equipo que sea responsable de la interfaz de usuario, otro que sea responsable de la lógica de la aplicación y un tercero que se encargue de la base de datos. En este entorno, un equipo de funciones es un avance significativo, ya que trabaja en todas estas capas para brindar la funcionalidad.

Con la adopción generalizada de equipos de funciones, todos los servicios pueden considerarse compartidos. Todos pueden cambiar todos los servicios, todos los fragmentos de código. El papel de los custodios de los servicios aquí se vuelve mucho más complejo, si es que existe. Lamentablemente, rara vez veo custodios en funcionamiento en los lugares donde se adopta este patrón, lo que conduce a los tipos de problemas que analizamos anteriormente.

Pero volvamos a pensar en lo que son los microservicios: servicios modelados a partir de un dominio empresarial, no técnico. Y si nuestro equipo, que es dueño de un servicio determinado, está alineado de manera similar con el dominio empresarial, es mucho más probable que el equipo pueda mantener un enfoque en el cliente y supervisar más el desarrollo de funciones, porque tiene una comprensión y propiedad holísticas de toda la tecnología asociada con un servicio.

Por supuesto, pueden ocurrir cambios transversales, pero su probabilidad se reduce significativamente si evitamos los equipos orientados a la tecnología.

Cuellos de botella en la entrega

Una razón clave por la que las personas optan por servicios compartidos es para evitar cuellos de botella en la entrega. ¿Qué sucede si hay una gran cantidad de cambios pendientes que deben realizarse en un solo servicio? Imaginemos que estamos implementando la posibilidad de que un cliente vea el género de una canción en todos nuestros productos, además de agregar un nuevo tipo de stock: tonos de llamada musicales virtuales para el teléfono móvil. El equipo del sitio web debe realizar un cambio para mostrar la información del género, mientras que el equipo de la aplicación móvil debe trabajar para permitir que los usuarios exploren, obtengan una vista previa y compren los tonos de llamada. Ambos cambios deben realizarse en el servicio de catálogo, pero desafortunadamente la mitad del equipo está fuera por la gripe y la otra mitad está atascada diagnosticando una falla de producción.

Tenemos un par de opciones que no implican servicios compartidos para evitar esta situación. La primera es simplemente esperar. Los equipos del sitio web y de la aplicación móvil pasan a otra cosa.

Dependiendo de qué tan importante sea la característica o de cuánto tiempo probablemente se demore, esto puede estar bien o puede ser un problema importante.

En lugar de eso, podría agregar personas al equipo de catálogo para ayudarlos a avanzar más rápido en su trabajo. Cuanto más estandarizados estén el conjunto de tecnologías y los modismos de programación que se utilizan en su sistema, más fácil será para otras personas realizar cambios en sus servicios. La otra cara de la moneda, por supuesto, como comentamos antes, es que la estandarización tiende a reducir la capacidad de un equipo para adoptar la solución adecuada para el trabajo y puede conducir a diferentes tipos de ineficiencias. Sin embargo, si el equipo está al otro lado del planeta, esto podría ser imposible.

Otra opción podría ser dividir el catálogo en un catálogo de música general independiente y un catálogo de tonos de llamada. Si el cambio que se está realizando para admitir tonos de llamada es bastante pequeño y la probabilidad de que sea un área en la que desarrollaremos mucho en el futuro también es bastante baja, es posible que esto sea prematuro. Por otro lado, si hay 10 semanas de funciones relacionadas con los tonos de llamada acumuladas, dividir el servicio podría tener sentido y que el equipo móvil se haga cargo.

Aunque hay otro modelo que podría funcionar bien para nosotros.

Código abierto interno

¿Y qué pasa si hemos hecho todo lo posible pero no encontramos la manera de no tener algunos servicios compartidos? En este punto, adoptar adecuadamente el modelo interno de código abierto puede tener mucho sentido.

En el código abierto normal, un pequeño grupo de personas se considera el responsable principal del proyecto. Son los custodios del código. Si quieres hacer un cambio en un proyecto de código abierto, puedes pedirle a uno de los responsables que lo haga por ti o puedes hacer el cambio tú mismo y enviarles una solicitud de incorporación de cambios. Los responsables principales siguen estando a cargo del código base; son los propietarios.

Dentro de la organización, este patrón también puede funcionar bien. Tal vez las personas que trabajaron originalmente en el servicio ya no formen parte de un equipo; tal vez ahora estén dispersas por toda la organización. Bueno, si aún tienen derechos de confirmación, puedes buscarlas y pedirles ayuda, tal vez asociándote con ellas o, si tienes las herramientas adecuadas, puedes enviarles una solicitud de incorporación de cambios.

Papel de los custodios

Aún queremos que nuestros servicios sean sensatos. Queremos que el código sea de una calidad decente y que el servicio en sí mismo muestre algún tipo de coherencia en su forma de construcción. También queremos asegurarnos de que los cambios que se están realizando ahora no hagan que los cambios planificados para el futuro sean mucho más difíciles de lo que deberían ser. Esto significa que también debemos adoptar los mismos patrones que se utilizan en el código abierto normal a nivel interno, lo que significa separar un grupo de colaboradores de confianza (el equipo central) y colaboradores no confiables (personas externas al equipo que envían cam-

El equipo central debe tener alguna forma de examinar y aprobar los cambios. Debe asegurarse de que los cambios sean coherentes idiomáticamente, es decir, que sigan las pautas generales de codificación del resto del código base. Por lo tanto, las personas que realizan la revisión tendrán que dedicar tiempo a trabajar con los autores para asegurarse de que el cambio sea de la calidad suficiente.

Los buenos guardianes se esfuerzan mucho en esto, se comunican claramente con los remitentes y fomentan el buen comportamiento. Los malos guardianes pueden usar esto como excusa para ejercer poder sobre los demás o tener guerras religiosas sobre decisiones técnicas arbitrarias. Habiendo visto ambos tipos de comportamiento, puedo decirte que una cosa está clara: de cualquier manera, lleva tiempo. Al considerar permitir que los autores no confiables envíen cambios a tu base de código, tienes que decidir si la sobrecarga que supone ser un guardián vale la pena: ¿podría el equipo central estar haciendo mejores cosas con el tiempo que dedica a examinar los parches?

Madurez

Cuanto menos estable o maduro sea un servicio, más difícil será permitir que personas ajena al equipo central envíen parches. Antes de que la columna vertebral de un servicio esté en su lugar, el equipo puede no saber qué significa algo bueno y, por lo tanto, puede tener dificultades para saber cómo es un buen envío. Durante esta etapa, el servicio en sí mismo está experimentando un alto grado de cambio.

La mayoría de los proyectos de código abierto tienden a no aceptar contribuciones de un grupo más amplio de colaboradores no confiables hasta que el núcleo de la primera versión está listo. Seguir un modelo similar para sus propias organizaciones tiene sentido. Si un servicio es bastante maduro y rara vez se modifica (por ejemplo, nuestro servicio de carrito de compras), entonces tal vez sea el momento de abrirlo a otras contribuciones.

Estampación

Para dar el mejor soporte a un modelo interno de código abierto, necesitará contar con algunas herramientas. Es importante utilizar una herramienta de control de versiones distribuida con la capacidad de que las personas envíen solicitudes de incorporación de cambios (o algo similar). Según el tamaño de la organización, es posible que también necesite herramientas que permitan la discusión y evolución de las solicitudes de parches; esto puede significar o no un sistema de revisión de código completo, pero la capacidad de comentar en línea sobre los parches es muy útil. Por último, deberá facilitarle al colaborador la creación e implementación de su software, y ponerlo a disposición de otros. Por lo general, esto implica tener canales de creación e implementación bien definidos y repositorios de artefactos centralizados.

Contextos delimitados y estructuras de equipo

Como se mencionó anteriormente, buscamos trazar los límites de nuestros servicios en torno a contextos delimitados. Por lo tanto, se deduce que nos gustaría que nuestros equipos también estuvieran alineados en contextos delimitados. Esto tiene múltiples beneficios. En primer lugar, a un equipo le resultará más fácil comprender los conceptos del dominio dentro de un contexto delimitado, ya que están interrelacionados. En segundo lugar, es más probable que los servicios dentro de un contexto delimitado sean servicios que se comuniquen entre sí, lo que facilita el diseño del sistema y la coordinación de la versión. Por último, en términos de cómo el equipo de entrega interactúa con las partes interesadas del negocio, se vuelve más fácil para el equipo crear buenas relaciones con uno o dos expertos en esa área.

¿El servicio huérfano?

¿Y qué ocurre con los servicios que ya no reciben mantenimiento activo? A medida que avanzamos hacia arquitecturas más detalladas, los propios servicios se vuelven más pequeños. Uno de los objetivos de los servicios más pequeños, como hemos comentado, es el hecho de que son más sencillos. Los servicios más sencillos con menos funcionalidades pueden no necesitar cambios durante un tiempo. Pensemos en el humilde servicio de carrito de compras, que ofrece algunas funciones bastante modestas: añadir al carrito, quitar del carrito, etc. Es bastante concebible que este servicio no tenga que cambiar durante meses después de su primera redacción, incluso si el desarrollo activo sigue en curso. ¿Qué ocurre aquí? ¿Quién es el propietario de este servicio?

Si las estructuras de su equipo están alineadas con los contextos delimitados de su organización, incluso los servicios que no se modifican con frecuencia siguen teniendo un propietario de facto. Imagine un equipo que esté alineado con el contexto de ventas web para consumidores. Podría encargarse del sitio web, el carrito de compras y los servicios de recomendación. Incluso si el servicio del carrito de compras no se ha modificado en meses, naturalmente le correspondería a este equipo realizar el cambio. Uno de los beneficios de los microservicios, por supuesto, es que si el equipo necesita cambiar el servicio para agregar una nueva característica y no le gusta, reescribirlo no debería llevar demasiado tiempo.

Dicho esto, si ha adoptado un enfoque verdaderamente políglota, haciendo uso de múltiples pilas de tecnología, entonces los desafíos de realizar cambios en un servicio huérfano podrían agravarse si su equipo ya no conoce la pila de tecnología.

Estudio de caso: RealEstate.com.au

El negocio principal de REA es el inmobiliario, pero abarca múltiples facetas diferentes, cada una de las cuales opera como una única línea de negocio (LOB). Por ejemplo, una línea de negocio se ocupa de propiedades residenciales en Australia, otra de propiedades comerciales, mientras que otra puede estar relacionada con uno de los negocios de REA en el extranjero. Estas líneas de negocio tienen equipos (o escuadrones) de entrega de TI asociados a ellas; algunas pueden tener solo un escuadrón, mientras que la más grande tiene cuatro. Por lo tanto, para las propiedades residenciales, hay varios equipos involucrados en la creación del sitio web y los servicios de listado para permitir que las personas exploren las propiedades. Las personas rotan entre estos equipos de vez en cuando, pero tienden a permanecer dentro de esa línea de negocio durante períodos prolongados, lo que garantiza que los miembros del equipo puedan desarrollar un fuerte conocimiento de esa parte del dominio. Esto, a su vez, ayuda a la comunicación entre las diversas partes interesadas del negocio y el equipo que les ofrece funciones.

Se espera que cada equipo dentro de una línea de negocios sea dueño de todo el ciclo de vida de los servicios que crea, incluyendo la creación, prueba y lanzamiento, soporte e incluso el desmantelamiento.

Un equipo de servicios de entrega central brinda asesoramiento y orientación a estos equipos, así como herramientas para ayudarlos a hacer el trabajo. Una fuerte cultura de automatización es clave, y REA hace un uso intensivo de AWS como parte clave para permitir que los equipos sean más autónomos. [La Figura 10-1](#) ilustra cómo funciona todo esto.

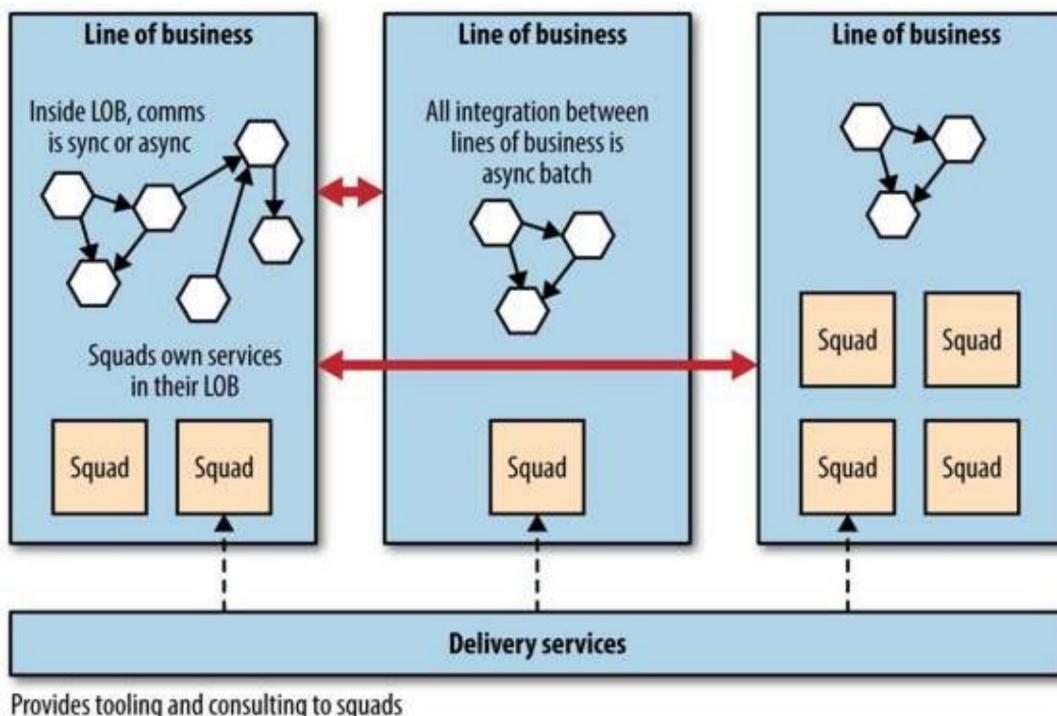


Figura 10-1. Descripción general de la estructura organizativa y de equipo de Realestate.com.au y su alineación con la arquitectura

No es solo la organización de entrega la que está alineada con el funcionamiento del negocio. También se extiende a la arquitectura. Un ejemplo de esto son los métodos de integración. Dentro de una línea de negocio, todos los servicios tienen la libertad de comunicarse entre sí de la forma que consideren adecuada, según lo decidan los equipos que actúan como sus custodios. Pero entre líneas de negocio, toda la comunicación está sujeta a

Sea un lote asincrónico, una de las pocas reglas inmutables de los equipos de arquitectura muy pequeños. Esta comunicación de grano grueso coincide también con la comunicación de grano grueso que existe entre las diferentes partes de la empresa. Al insistir en que sea por lotes, cada línea de negocio tiene mucha libertad en cuanto a cómo actúa y se gestiona a sí misma. Puede darse el lujo de dejar de prestar sus servicios cuando quiera, sabiendo que mientras pueda satisfacer la integración por lotes con otras partes de la empresa y sus propias partes interesadas, a nadie le importará.

Esta estructura ha permitido una autonomía significativa no sólo de los equipos, sino también de las diferentes partes de la empresa. Hace unos años, REA contaba con un puñado de servicios, pero ahora cuenta con cientos, con más servicios que personas, y está creciendo a un ritmo rápido. La capacidad de implementar cambios ha ayudado a la empresa a lograr un éxito significativo en el mercado local, hasta el punto de expandirse al extranjero. Y lo más alentador de todo es que, al hablar con la gente de allí, tengo la impresión de que tanto la arquitectura como la estructura organizativa tal como están ahora son sólo la última versión, más que el destino. Me atrevo a decir que en otros cinco años REA volverá a verse muy diferente.

Las organizaciones que son lo suficientemente adaptables como para cambiar no solo la arquitectura de sus sistemas, sino también su estructura organizacional, pueden obtener enormes beneficios en términos de una mayor autonomía de los equipos y una comercialización más rápida de nuevas características y funcionalidades. REA es solo una de las muchas organizaciones que se están dando cuenta de que la arquitectura de sistemas no existe en el vacío.

La ley de Conway al revés

Hasta ahora, hemos hablado de cómo la organización afecta el diseño del sistema. Pero ¿qué sucede a la inversa? Es decir, ¿puede un diseño de sistema cambiar la organización? Si bien no he podido encontrar la misma calidad de evidencia que respalde la idea de que la ley de Conway funciona a la inversa, lo he visto anecdotamente.

Probablemente el mejor ejemplo fue el de un cliente con el que trabajé hace muchos años. En la época en que la Web era bastante incipiente e Internet se consideraba algo que llegaba en un disquete de AOL a través de la puerta, esta empresa era una gran imprenta que tenía un sitio web pequeño y modesto. Tenía un sitio web porque era lo que había que hacer, pero en el gran esquema de las cosas no era muy importante para el funcionamiento de la empresa. Cuando se creó el sistema original, se tomó una decisión técnica bastante arbitraria sobre cómo funcionaría el sistema.

El contenido de este sistema se obtenía de diversas formas, pero la mayor parte provenía de terceros que colocaban anuncios para que los viera el público en general. Había un sistema de entrada que permitía que los terceros que pagaban crearan contenido, un sistema central que tomaba esos datos y los enriquecía de diversas formas, y un sistema de salida que creaba el sitio web final que el público en general podía explorar.

Si las decisiones de diseño originales fueron las correctas en su momento es una conversación para los historiadores, pero muchos años después la empresa había cambiado bastante y yo y muchos de mis colegas empezábamos a preguntarnos si el diseño del sistema era adecuado para el estado actual de la empresa. Su negocio de impresión física había disminuido significativamente y los ingresos y, por lo tanto, las operaciones comerciales de la organización ahora estaban dominados por su presencia en línea.

Lo que vimos en ese momento fue una organización estrechamente alineada a este sistema de tres partes. Tres canales o divisiones en el sector de TI de la empresa se alineaban con cada una de las partes de entrada, núcleo y salida de la empresa. Dentro de esos canales, había equipos de entrega separados. Lo que no me di cuenta en ese momento fue que estas estructuras organizativas no eran anteriores al diseño del sistema, sino que en realidad crecieron a su alrededor. A medida que el sector de impresión de la empresa disminuía y el sector digital crecía, el diseño del sistema sentó inadvertidamente las bases para el crecimiento de la organización.

Al final nos dimos cuenta de que, cualesquiera que fueran las deficiencias del diseño del sistema, tendríamos que hacer cambios en la estructura organizativa para lograr un cambio. Muchos años después, ese proceso sigue siendo un trabajo en progreso.

Gente

No importa cómo se vea al principio, siempre es un problema de personas.

Gerry Weinberg, La segunda ley de la consultoría

Tenemos que aceptar que, en un entorno de microservicios, es más difícil para un desarrollador pensar simplemente en escribir código en su propio pequeño mundo. Tiene que ser más consciente de las implicaciones de cosas como las llamadas a través de los límites de la red o las implicaciones de un fallo.

También hemos hablado de la capacidad de los microservicios para facilitar la prueba de nuevas tecnologías, desde almacenes de datos hasta lenguajes. Pero si estás pasando de un mundo en el que tienes un sistema monolítico, donde la mayoría de tus desarrolladores han tenido que usar un solo lenguaje y permanecer completamente ajenos a las preocupaciones operativas, entonces lanzarlos al mundo de los microservicios puede ser un duro despertar para ellos.

De la misma manera, transferir poder a los equipos de desarrollo para aumentar su autonomía puede tener sus riesgos. Las personas que en el pasado han dejado en manos de otros el trabajo por encima del muro están acostumbradas a tener a alguien más a quien culpar y es posible que no se sientan cómodas siendo totalmente responsables de su trabajo. ¡Incluso puede que encuentre barreras contractuales para que sus desarrolladores lleven localizadores de soporte para los sistemas a los que dan soporte!

Aunque este libro ha tratado principalmente sobre tecnología, las personas no son sólo un tema secundario que se debe considerar; son las personas que construyeron lo que usted tiene ahora y construirán lo que sucederá en el futuro. Idear una visión de cómo se deben hacer las cosas sin tener en cuenta cómo se sentirá su personal actual al respecto o sin considerar qué capacidades tienen es probable que conduzca a un mal lugar.

Cada organización tiene su propia dinámica en torno a este tema. Comprenda el deseo de cambio de su personal. ¡No los presione demasiado! Tal vez todavía tenga un equipo separado que se ocupe del soporte de primera línea o de la implementación durante un breve período de tiempo, lo que les da tiempo a sus desarrolladores para adaptarse a otras nuevas prácticas. Sin embargo, es posible que deba aceptar que necesita diferentes tipos de personas en su organización para que todo esto funcione. Sea cual sea su enfoque, comprenda que debe ser claro al articular las responsabilidades de su gente en un mundo de microservicios y también tener claro por qué esas responsabilidades son importantes para usted.

Esto puede ayudarle a ver cuáles podrían ser sus carencias en habilidades y pensar en cómo solucionarlas.

Para muchas personas, este será un viaje bastante aterrador. Solo recuerda que, sin personas a bordo, cualquier cambio que quieras hacer podría estar condenado al fracaso desde el principio.

Resumen

La ley de Conway destaca los peligros de intentar aplicar un diseño de sistema que no se adapta a la organización. Esto nos lleva a intentar alinear la propiedad del servicio con equipos ubicados en el mismo lugar, que a su vez están alineados en torno a los mismos contextos delimitados de la organización.

Cuando ambos no están alineados, surgen puntos de tensión, como se describe a lo largo de este capítulo.

Al reconocer el vínculo entre ambos, nos aseguraremos de que el sistema que estamos tratando de construir tenga sentido para la organización para la que lo estamos construyendo.

Algunos de los temas que abordamos aquí se relacionaron con los desafíos de trabajar con organizaciones a gran escala. Sin embargo, hay otras consideraciones técnicas que debemos tener en cuenta cuando nuestros sistemas comienzan a crecer más allá de unos pocos servicios discretos. Las abordaremos a continuación.

⁴ ¡Y todos sabemos que Windows Vista era bastante propenso a errores!

Capítulo 11. Microservicios a escala

Cuando se trata de ejemplos agradables, pequeños y del tamaño de un libro, todo parece simple. Pero el mundo real es un espacio más complejo. ¿Qué sucede cuando nuestras arquitecturas de microservicios crecen desde comienzos más simples y humildes hasta algo más complejo? ¿Qué sucede cuando tenemos que gestionar fallos en varios servicios independientes o cientos de servicios? ¿Cuáles son algunos de los patrones de afrontamiento cuando hay más microservicios que personas? Vamos a averiguarlo.

El fracaso está en todas partes

Entendemos que las cosas pueden salir mal. Los discos duros pueden fallar. Nuestro software puede fallar. Y como cualquiera que haya leído las **falacias de la computación distribuida** Podemos decirles que sabemos que la red no es confiable. Podemos hacer todo lo posible para intentar limitar las causas de falla, pero a cierta escala, la falla se vuelve inevitable. Los discos duros, por ejemplo, ahora son más confiables que nunca, pero se romperán tarde o temprano. Cuantos más discos duros tenga, mayor será la probabilidad de que falle una unidad individual; la falla se convierte en una certeza estadística a gran escala.

Incluso para aquellos de nosotros que no pensamos a escala extrema, si podemos aceptar la posibilidad de un fallo, estaremos en mejores condiciones. Por ejemplo, si podemos gestionar el fallo de un servicio con elegancia, se deduce que también podemos hacer actualizaciones locales de un servicio, ya que una interrupción planificada es mucho más fácil de manejar que una no planificada.

También podemos dedicar un poco menos de nuestro tiempo a intentar detener lo inevitable y un poco más de nuestro tiempo a afrontarlo con elegancia. Me sorprende la cantidad de organizaciones que implementan procesos y controles para intentar evitar que se produzcan fallos, pero que dedican poco o ningún pensamiento a facilitar la recuperación de los mismos.

Asumir que todo puede y va a fallar nos lleva a pensar diferente sobre cómo resolver los problemas.

Hace muchos años, cuando pasé un tiempo en el campus de Google, vi un ejemplo de esta forma de pensar. En la zona de recepción de uno de los edificios de Mountain View había un viejo estante con máquinas, que estaba allí como una especie de exposición. Observé un par de cosas. En primer lugar, estos servidores no estaban en cajas de servidores, sino que eran simplemente placas base desnudas encajadas en el estante. Sin embargo, lo principal que noté fue que los discos duros estaban sujetos con velcro. Le pregunté a uno de los empleados de Google por qué era así. “Ah”, dijo, “los discos duros fallan tanto que no queremos que estén atornillados. Simplemente los sacamos, los tiramos a la basura y colocamos uno nuevo con velcro”.

Permítanme repetirlo: a gran escala, incluso si compra el mejor equipo, el hardware más caro, no puede evitar el hecho de que las cosas pueden fallar y lo harán. Por lo tanto, debe asumir que el fracaso puede suceder. Si incorpora esta forma de pensar en todo lo que hace y planifica para el fracaso, puede hacer diferentes concesiones. Si sabe que su sistema puede manejar el hecho de que un servidor puede fallar y lo hará, ¿por qué molestarte en gastar mucho en él? ¿Por qué no usar una placa base desnuda con componentes más baratos (y algo de velcro) como lo hizo Google, en lugar de preocuparse demasiado por la resiliencia de un solo nodo?

¿Cuánto es demasiado?

En el capítulo 7 abordamos el tema de los requisitos multifuncionales. Para comprender los requisitos multifuncionales es necesario tener en cuenta aspectos como la durabilidad de los datos, la disponibilidad de los servicios, el rendimiento y la latencia aceptable de los servicios. Muchas de las técnicas que se tratan en este capítulo y en otros lugares hablan de enfoques para implementar estos requisitos, pero solo usted sabe exactamente cuáles podrían ser los requisitos en sí.

Tener un sistema de escalado automático capaz de reaccionar ante una mayor carga o una falla de nodos individuales puede ser fantástico, pero podría ser excesivo para un sistema de generación de informes que solo necesita ejecutarse dos veces al mes, donde estar fuera de servicio durante un día o dos no es un gran problema. Del mismo modo, averiguar cómo hacer implementaciones azules/verdes para eliminar el tiempo de inactividad de un servicio puede tener sentido para su sistema de comercio electrónico en línea, pero para su base de conocimiento de intranet corporativa probablemente sea un paso demasiado lejos.

Los usuarios del sistema son los que determinan cuántos fallos puede tolerar o qué tan rápido debe ser su sistema. Eso, a su vez, le ayudará a entender qué técnicas serán las más adecuadas para usted. Dicho esto, sus usuarios no siempre podrán expresar cuáles son los requisitos exactos. Por lo tanto, debe hacer preguntas para extraer la información correcta y ayudarlos a comprender los costos relativos de brindar diferentes niveles de servicio.

Como mencioné anteriormente, estos requisitos multifuncionales pueden variar de un servicio a otro, pero sugeriría definir algunos requisitos multifuncionales generales y luego anularlos para casos de uso particulares. Cuando se trata de considerar si es necesario escalar horizontalmente el sistema para manejar mejor la carga o las fallas, comience por tratar de comprender los siguientes requisitos:

Tiempo de respuesta/latencia

¿Cuánto tiempo deberían tardar las distintas operaciones? Puede ser útil medir esto con diferentes cantidades de usuarios para entender cómo el aumento de la carga afectará el tiempo de respuesta. Dada la naturaleza de las redes, siempre habrá valores atípicos, por lo que puede ser útil establecer objetivos para un percentil determinado de las respuestas monitoreadas. El objetivo también debe incluir la cantidad de conexiones/usuarios simultáneos que espera que su software gestione. Por lo tanto, podría decir: "Esperamos que el sitio web tenga un tiempo de respuesta del percentil 90 de 2 segundos al gestionar 200 conexiones simultáneas por segundo".

Disponibilidad

¿Se puede esperar que un servicio esté inactivo? ¿Se considera que es un servicio 24 horas al día, 7 días a la semana? A algunas personas les gusta observar los períodos de inactividad aceptables al medir la disponibilidad, pero ¿qué tan útil es esto para alguien que llama a su servicio? Debería poder confiar en que su servicio responderá o no. Medir los períodos de inactividad es realmente más útil desde el punto de vista de los informes históricos.

Durabilidad de los datos

¿Cuánta pérdida de datos es aceptable? ¿Durante cuánto tiempo se deben conservar los datos? Es muy probable que esto cambie según el caso. Por ejemplo, puede optar por conservar los registros de las sesiones de los usuarios durante un año o menos para ahorrar espacio, pero es posible que deba conservar los registros de sus transacciones financieras durante muchos años.

Una vez que haya establecido estos requisitos, necesitará una forma de medirlos sistemáticamente de forma continua. Puede decidir utilizar pruebas de rendimiento, por ejemplo, para asegurarse de que su sistema cumpla con los objetivos de rendimiento aceptables, pero deberá asegurarse de supervisar estas estadísticas también en producción.

Funcionalidad degradada

Una parte esencial de la creación de un sistema resistente, especialmente cuando su funcionalidad se distribuye en una serie de microservicios diferentes que pueden estar inactivos o inactivos, es la capacidad de degradar la funcionalidad de forma segura. Imaginemos una página web estándar en nuestro sitio de comercio electrónico. Para reunir las distintas partes de ese sitio web, es posible que necesitemos que participen varios microservicios. Un microservicio podría mostrar los detalles sobre el álbum que se ofrece a la venta.

Otro podría mostrar el precio y el nivel de existencias. Y probablemente también mostraremos el contenido del carrito de compras, que puede ser otro microservicio. Ahora bien, si uno de esos servicios no funciona y eso hace que toda la página web no esté disponible, entonces podríamos decir que hemos creado un sistema que es menos resistente que uno que requiere que solo un servicio esté disponible.

Lo que tenemos que hacer es comprender el impacto de cada interrupción y determinar cómo degradar adecuadamente la funcionalidad. Si el servicio de carrito de compras no está disponible, probablemente tengamos muchos problemas, pero aún así podríamos mostrar la página web con el anuncio. Quizás simplemente ocultemos el carrito de compras o lo reemplazemos con un ícono que diga "¡Volveremos pronto!".

Con una única aplicación monolítica, no tenemos que tomar muchas decisiones. El estado del sistema es binario. Pero con una arquitectura de microservicios, debemos considerar una situación mucho más matizada. Lo correcto que hay que hacer en cualquier situación no suele ser una decisión técnica.

Quizás sepamos qué es técnicamente posible cuando el carrito de compras está caído, pero a menos que entendamos el contexto comercial, no entenderemos qué acción deberíamos tomar.

Por ejemplo, tal vez cerremos todo el sitio, sigamos permitiendo que la gente navegue por el catálogo de artículos o reemplacemos la parte de la interfaz de usuario que contiene el control del carrito con un número de teléfono para realizar un pedido. Pero para cada interfaz orientada al cliente que utiliza múltiples microservicios, o cada microservicio que depende de múltiples colaboradores posteriores, debe preguntarse: "¿Qué sucede si esto no funciona?" y saber qué hacer.

Si pensamos en la importancia de cada una de nuestras capacidades en términos de nuestros requisitos multifuncionales, estaremos en una posición mucho mejor para saber qué podemos hacer. Ahora, consideraremos algunas cosas que podemos hacer desde un punto de vista técnico para asegurarnos de que, cuando ocurra un fallo, podamos manejarlo con elegancia.

Medidas de seguridad arquitectónica

Existen algunos patrones, a los que en conjunto me refiero como medidas de seguridad arquitectónicas, que podemos utilizar para garantizar que, si algo sale mal, no provoque efectos secundarios desagradables. Estos son puntos que es esencial que comprenda y que debería considerar seriamente estandarizar en su sistema para garantizar que un mal ciudadano no haga que el mundo entero se derrumbe ante sus ojos. En un momento, analizaremos algunas medidas de seguridad clave que debería considerar, pero antes de hacerlo, me gustaría compartir una breve historia para describir el tipo de cosas que pueden salir mal.

Yo era el responsable técnico de un proyecto en el que estábamos creando un sitio web de anuncios clasificados en línea. El sitio web en sí mismo manejaba volúmenes bastante altos y generaba una buena cantidad de ingresos para la empresa. Nuestra aplicación principal manejaba la visualización de algunos anuncios clasificados y también enviaba llamadas a otros servicios que proporcionaban diferentes tipos de productos, como se muestra en [la Figura 11-1](#). En realidad, se trata de un ejemplo de una aplicación estranguladora, en la que un nuevo sistema intercepta las llamadas realizadas a aplicaciones heredadas y las reemplaza por completo de manera gradual. Como parte de este proyecto, estábamos a mitad de camino de retirar las aplicaciones más antiguas. Acabábamos de trasladar el producto de mayor volumen y mayores ganancias, pero gran parte del resto de los anuncios seguían siendo distribuidos por varias aplicaciones más antiguas. En términos tanto de la cantidad de búsquedas como del dinero generado por estas aplicaciones, había una cola muy larga.

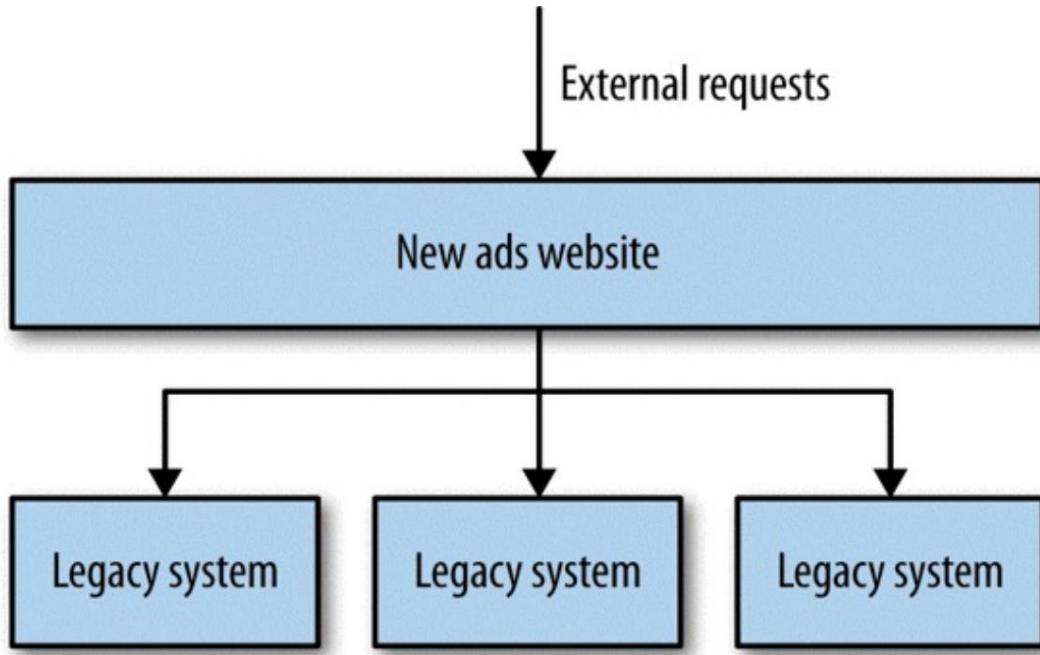


Figura 11-1. Un sitio web de anuncios clasificados que está estrangulando a las aplicaciones antiguas

Nuestro sistema había estado activo durante un tiempo y se comportaba muy bien, manejando una carga nada despreciable. En ese momento, debíamos estar manejando alrededor de 6000 a 7000 solicitudes por segundo durante los picos, y aunque la mayor parte de eso estaba almacenada en caché por servidores proxy inversos ubicados frente a nuestros servidores de aplicaciones, las búsquedas de productos (el aspecto más importante del sitio) en su mayoría no estaban almacenadas en caché y requerían un viaje de ida y vuelta completo del servidor.

Una mañana, justo antes de alcanzar nuestro pico diario de trabajo a la hora del almuerzo, el sistema comenzó a comportarse lentamente y luego, gradualmente, comenzó a fallar. Teníamos cierto nivel de monitoreo en nuestra nueva aplicación principal, suficiente para indicarnos que cada uno de los nodos de nuestra aplicación estaba alcanzando un pico de uso de CPU del 100 %, muy por encima de los niveles normales incluso en el pico. En un corto período de tiempo, todo el sitio dejó de funcionar.

Logramos localizar al culpable y hacer que el sitio volviera a funcionar. Resultó que uno de los sistemas de publicidad de bajada, uno de los más antiguos y menos mantenidos, había comenzado a responder muy lentamente. Responder muy lentamente es uno de los peores modos de falla que puede experimentar. Si un sistema simplemente no está ahí, lo descubres bastante rápido. Cuando simplemente es lento, terminas esperando un tiempo antes de rendirte. Pero sea cual sea la causa de la falla, habíamos creado un sistema que era vulnerable a una falla en cascada. Un servicio de bajada, sobre el cual teníamos poco control, fue capaz de derribar todo nuestro sistema.

Mientras un equipo analizaba los problemas con el sistema de bajada, el resto de nosotros comenzamos a analizar qué había fallado en nuestra aplicación. Encontramos algunos problemas. Estábamos usando un pool de conexiones HTTP para manejar nuestras conexiones de bajada. Los subprocessos en el pool en sí tenían tiempos de espera configurados para el tiempo que esperarían al realizar la llamada HTTP de bajada, lo cual es bueno. El problema era que todos los trabajadores tardaban un tiempo en agotarse debido a la lentitud del sistema de bajada. Mientras esperaban, más solicitudes iban al pool pidiendo subprocessos de trabajadores. Como no había trabajadores disponibles, estas solicitudes se colgaban. Resultó que la biblioteca del pool de conexiones que estábamos usando tenía un tiempo de espera para esperar a los trabajadores, ¡pero estaba deshabilitado de manera predeterminada! Esto llevó a una enorme acumulación de subprocessos bloqueados. Nuestra aplicación normalmente tenía 40 conexiones simultáneas en un momento dado. En el espacio de cinco minutos, esta situación hizo que alcanzáramos un pico de alrededor de 800 conexiones, lo que provocó la caída del sistema.

Lo peor fue que el servicio de bajada del que hablábamos representaba una funcionalidad que utilizaba menos del 5 % de nuestra base de clientes y generaba incluso menos ingresos que eso. En el fondo, descubrimos por las malas que los sistemas que actúan con lentitud son mucho más difíciles de manejar que los sistemas que fallan rápidamente. En un sistema distribuido, la latencia mata.

Incluso si hubiéramos configurado correctamente los tiempos de espera en el grupo, también estábamos compartiendo un único grupo de conexiones HTTP para todas las solicitudes salientes. Esto significaba que un servicio lento podía agotar la cantidad de trabajadores disponibles por sí solo, incluso si todo lo demás funcionaba correctamente. Por último, estaba claro que el servicio de bajada en cuestión no funcionaba correctamente, pero seguimos enviándole tráfico. En nuestra situación, esto significaba que estábamos empeorando una situación que ya era mala, ya que el servicio de bajada no tenía posibilidad de recuperarse. Terminamos implementando tres soluciones para evitar que esto volviera a suceder: establecer los tiempos de espera correctos, implementar mamparas para separar los diferentes grupos de conexiones e implementar un disyuntor para evitar enviar llamadas a un sistema que no funcionaba correctamente en primer lugar.

La organización antifrágil

En su libro Antifragil (Random House), Nassim Taleb habla de las cosas que realmente se benefician del fracaso y el desorden. Ariel Tseitlin utilizó este concepto para acuñar el concepto de [organización antifrágil](#), en lo que respecta a cómo funciona Netflix.

La escala en la que opera Netflix es bien conocida, como también lo es el hecho de que Netflix se basa completamente en la infraestructura de AWS. Estos dos factores implican que debe aceptar bien los fallos. Netflix va más allá de eso al incitar a los fallos para asegurarse de que sus sistemas sean tolerantes a ellos.

Algunas organizaciones estarían contentas con los días de juego, donde se simulan fallas apagando los sistemas y haciendo que los distintos equipos reaccionen. Durante mi tiempo en Google, esto era algo bastante común para varios sistemas, y ciertamente creo que muchas organizaciones podrían beneficiarse de tener este tipo de ejercicios regularmente. Google va más allá de las pruebas simples para imitar fallas de servidores, y como parte de sus ejercicios anuales [DiRT \(Prueba de recuperación ante desastres\)](#) ha simulado desastres a gran escala como terremotos. Netflix también adopta un enfoque más agresivo, al escribir programas que causan fallas y ejecutarlos en producción a diario.

El más famoso de estos programas es Chaos Monkey, que durante ciertas horas del día apaga máquinas al azar. Saber que esto puede suceder y sucederá en producción significa que los desarrolladores que crean los sistemas realmente tienen que estar preparados para ello. Chaos Monkey es solo una parte del ejército simio de robots de Netflix que se encargan de los fallos. Chaos Gorilla se utiliza para eliminar un centro de disponibilidad completo (el equivalente de AWS a un centro de datos), mientras que Latency Monkey simula una conectividad de red lenta entre máquinas. Netflix ha puesto estas herramientas a disposición bajo una [licencia de código abierto](#). Para muchos, la prueba definitiva de si su sistema es realmente robusto podría ser desatar su propio ejército simio en su infraestructura de producción.

Aceptar y fomentar el fracaso a través del software y crear sistemas que puedan manejarlo es solo una parte de lo que hace Netflix. También comprende la importancia de aprender de los errores cuando ocurren y de adoptar una cultura de no culpar a nadie cuando ocurren.

Los desarrolladores tienen aún más poder para ser parte de este proceso de aprendizaje y evolución, ya que cada desarrollador también es responsable de administrar sus propios servicios de producción.

Al provocar fallos y prepararse para ellos, Netflix ha garantizado que los sistemas que tiene se escalen mejor y satisfagan mejor las necesidades de sus clientes.

No todo el mundo tiene por qué llegar a los extremos que Google o Netflix, pero es importante entender el cambio de mentalidad que se requiere con los sistemas distribuidos. Las cosas fallarán. El hecho de que su sistema ahora esté distribuido en múltiples máquinas (que pueden fallar y fallarán) a través de una red (que no será confiable) en realidad puede hacer que su sistema sea más vulnerable, no menos. Por lo tanto, independientemente de si está tratando de brindar un servicio a la escala de Google o Netflix, prepárese para el tipo de fallas que ocurren con

Es muy importante contar con arquitecturas más distribuidas. ¿Qué debemos hacer entonces para manejar las fallas en nuestros sistemas?

Tiempos de espera

Los tiempos de espera son algo que es fácil pasar por alto, pero en un sistema de bajada es importante hacerlos correctamente. ¿Cuánto tiempo puedo esperar antes de poder considerar que un sistema de bajada está realmente inactivo?

Si espera demasiado para decidir que una llamada ha fallado, puede ralentizar todo el sistema.

Si se agota el tiempo de espera demasiado rápido, se considerará que una llamada que podría haber funcionado falló. Si no se establecen tiempos de espera, un sistema descendente que se caiga podría bloquear todo el sistema.

Establezca tiempos de espera en todas las llamadas fuera de proceso y elija un tiempo de espera predeterminado para todo. Registre cuándo se producen tiempos de espera, observe qué sucede y cámbielos según corresponda.

Disyuntores

En su propia casa, existen disyuntores para proteger sus dispositivos eléctricos de picos de tensión. Si se produce un pico, el disyuntor salta, protegiendo sus costosos electrodomésticos. También puede desactivar manualmente un disyuntor para cortar la corriente en una parte de su casa, lo que le permitirá trabajar de forma segura con los sistemas eléctricos. El libro de Michael Nygard, *Release It!* (Pragmatic Programmers), muestra cómo la misma idea puede hacer maravillas como mecanismo de protección para nuestro software.

Considere la historia que compartí hace un momento. La aplicación de anuncios heredada de bajada respondía muy lentamente, antes de finalmente devolver un error. Incluso si hubiéramos establecido los tiempos de espera correctos, tendríamos que esperar mucho tiempo antes de recibir el error. Y luego lo intentaríamos nuevamente la próxima vez que recibieran una solicitud y esperaríamos. Ya es bastante malo que el servicio de bajada no funcione correctamente, pero también nos está haciendo ir lentos.

Con un disyuntor, después de que una cierta cantidad de solicitudes al recurso de bajada hayan fallado, el disyuntor se activa. Todas las solicitudes posteriores fallan rápidamente mientras el disyuntor se encuentra en su estado activado. Después de un cierto período de tiempo, el cliente envía algunas solicitudes para ver si el servicio de bajada se ha recuperado y, si obtiene suficientes respuestas en buen estado, restablece el disyuntor. Puede ver una descripción general de este proceso en [la Figura 11-2](#).

La forma de implementar un disyuntor depende de lo que signifique una solicitud fallida , pero cuando los he implementado para conexiones HTTP, he entendido que un fallo significa un tiempo de espera o un código de retorno HTTP 5XX. De esta manera, cuando un recurso descendente está inactivo, se agota el tiempo de espera o devuelve errores, después de que se alcanza un cierto umbral, dejamos de enviar tráfico automáticamente y comenzamos a fallar rápidamente. Y podemos comenzar de nuevo automáticamente cuando todo está en orden.

Configurar correctamente los parámetros puede ser un poco complicado. No conviene que el disyuntor salte demasiado rápido ni que tarde demasiado en saltar. Asimismo, conviene asegurarse de que el servicio descendente vuelva a funcionar correctamente antes de enviar tráfico. Al igual que con los tiempos de espera, yo elegiría algunos valores predeterminados razonables y los mantendría en todas partes, para luego cambiarlos en casos específicos.

Mientras el disyuntor está activado, tienes algunas opciones. Una es poner en cola las solicitudes y volver a intentarlas más tarde. Para algunos casos de uso, esto puede ser adecuado, especialmente si estás realizando algún trabajo como parte de un trabajo asíncrono. Sin embargo, si esta llamada se realiza como parte de una cadena de llamadas sincrónicas, probablemente sea mejor fallar rápido. Esto podría significar propagar un error a lo largo de la cadena de llamadas o una degradación más sutil de la funcionalidad.

Si tenemos este mecanismo en funcionamiento (como los disyuntores de nuestro hogar), podríamos usarlos manualmente para que sea más seguro hacer nuestro trabajo. Por ejemplo, si quisieramos desconectar un microservicio como parte del mantenimiento de rutina, podríamos hacer estallar manualmente todos los disyuntores de los sistemas dependientes para que fallen rápidamente mientras el microservicio esté fuera de línea. Una vez que vuelva a funcionar, podemos restablecer los disyuntores y todo debería volver a la normalidad.

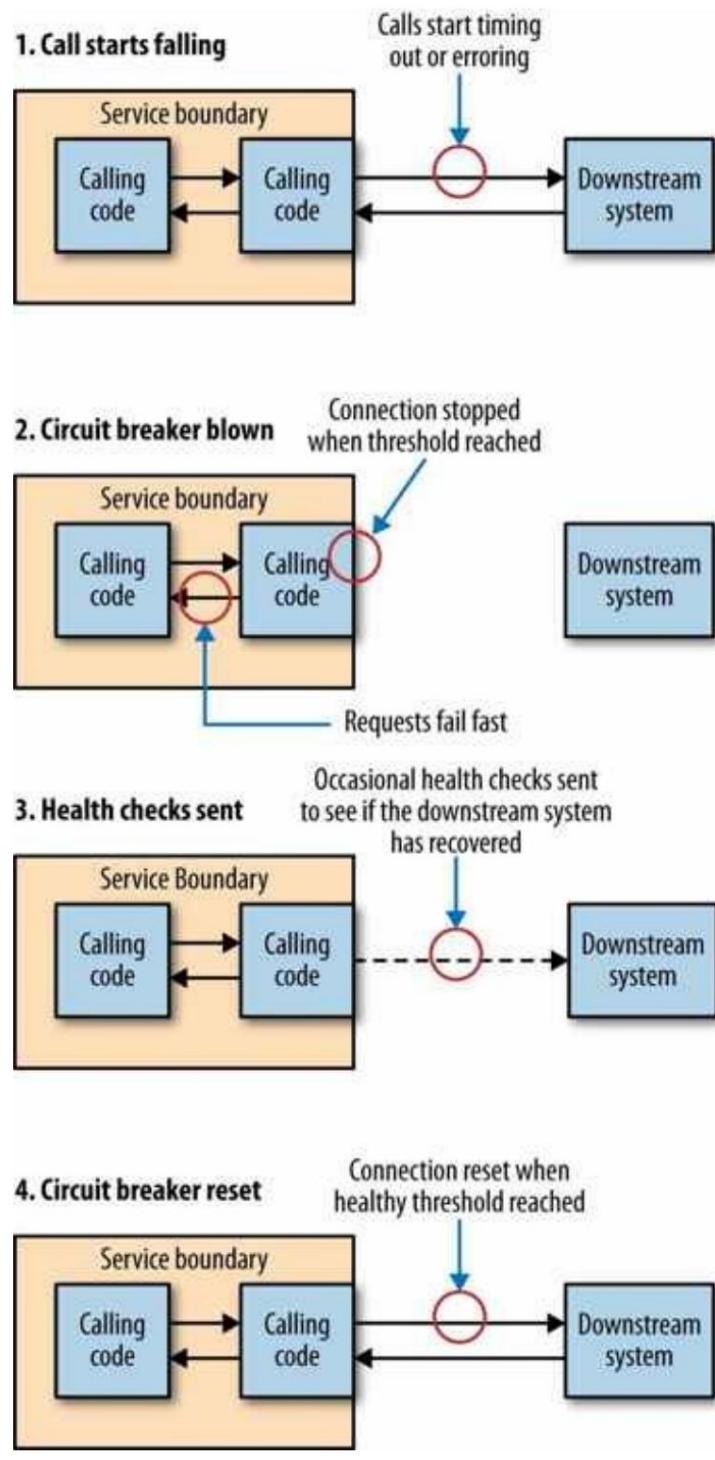


Figura 11-2. Descripción general de los disyuntores

Mamparos

En otro patrón de Release It!, Nygard introduce el concepto de mamparo como una forma de aislarse de las fallas. En el transporte marítimo, un mamparo es una parte del barco que se puede sellar para proteger el resto del barco. De modo que si el barco tiene una fuga, se pueden cerrar las puertas del mamparo. Se pierde parte del barco, pero el resto permanece intacto.

En términos de arquitectura de software, hay muchos mamparos diferentes que podemos considerar. Volviendo a mi propia experiencia, en realidad perdimos la oportunidad de implementar un mamparo. Deberíamos haber utilizado diferentes grupos de conexiones para cada conexión descendente. De esa manera, si un grupo de conexiones se agota, las demás conexiones no se verían afectadas, como vemos en [la Figura 11-3](#). Esto garantizaría que si un servicio descendente comenzara a comportarse lentamente en el futuro, solo ese grupo de conexiones se vería afectado, lo que permitiría que las demás llamadas procedieran con normalidad.

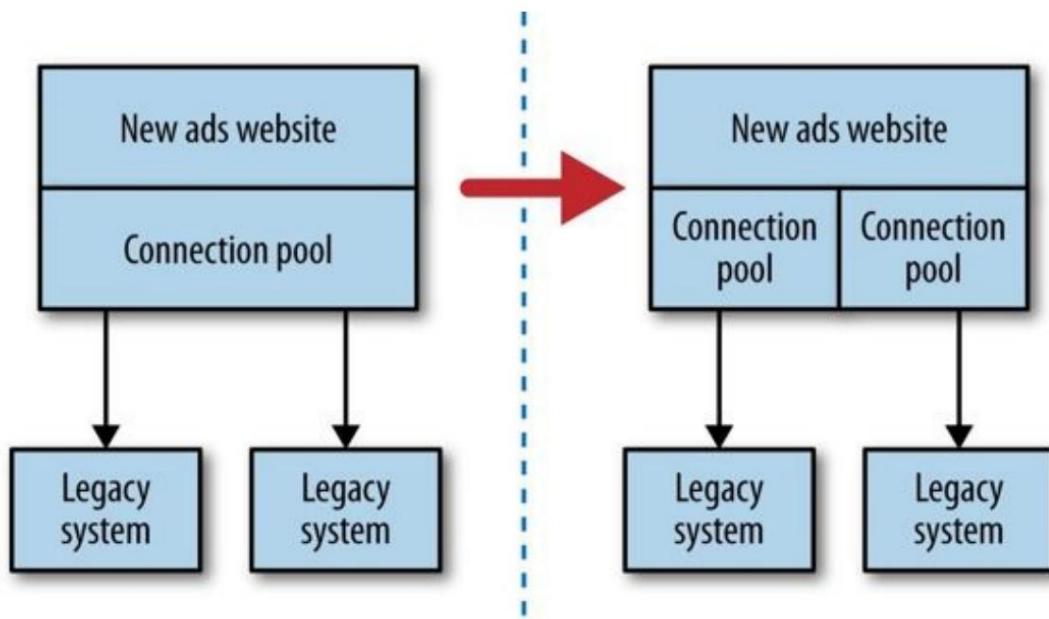


Figura 11-3. Uso de un grupo de conexiones por servicio descendente para proporcionar mamparos

La separación de preocupaciones también puede ser una forma de implementar barreras. Al separar la funcionalidad en microservicios separados, reducimos la posibilidad de que una interrupción en un área afecte a otra.

Analiza todos los aspectos de tu sistema que pueden fallar, tanto dentro de tus microservicios como entre ellos. ¿Tienes mamparos instalados? Te sugeriría comenzar con grupos de conexiones separados para cada conexión descendente, como mínimo. Sin embargo, es posible que quieras ir más allá y considerar también el uso de disyuntores.

Podemos pensar en nuestros disyuntores como un mecanismo automático para sellar un mamparo, no solo para proteger al consumidor del problema aguas abajo, sino también para proteger potencialmente el servicio aguas abajo de más llamadas que puedan tener un impacto adverso. Dados los peligros de una falla en cascada, recomendaría obligar a instalar disyuntores para todas sus llamadas aguas abajo sincrónicas. Tampoco tiene que escribir los suyos propios. [Hystrix](#) de Netflix

La biblioteca es una abstracción de disyuntor JVM que viene con un monitoreo poderoso, pero existen otras implementaciones para diferentes pilas de tecnología, como [Polly para .NET](#), o el [mixin circuit_breaker para Ruby](#).

En muchos sentidos, los mamparos son los más importantes de estos tres patrones. Los tiempos de espera y los disyuntores ayudan a liberar recursos cuando se ven limitados, pero los mamparos pueden garantizar que no se vean limitados en primer lugar. Hystrix le permite, por ejemplo, implementar mamparos que realmente rechacen solicitudes en determinadas condiciones para garantizar que los recursos no se saturen aún más; esto se conoce como desastre de cascada. A veces, rechazar una solicitud es la mejor forma de evitar que un sistema importante se sature y se convierta en un cuello de botella para múltiples servicios ascendentes.

Aislamiento

Cuento más depende un servicio del funcionamiento de otro, más afecta la salud de uno a la capacidad del otro para hacer su trabajo. Si podemos utilizar técnicas de integración que permitan que un servidor de bajada esté fuera de línea, es menos probable que los servicios de subida se vean afectados por cortes, ya sean planificados o no.

Existe otra ventaja de aumentar el aislamiento entre los servicios: cuando los servicios están aislados entre sí, se necesita mucha menos coordinación entre los propietarios de los servicios. Cuanto menos coordinación se necesite entre los equipos, más autonomía tendrán estos, ya que podrán operar y desarrollar sus servicios con mayor libertad.

Idempotencia

En las operaciones idempotentes , el resultado no cambia después de la primera aplicación, incluso si la operación se aplica posteriormente varias veces. Si las operaciones son idempotentes, podemos repetir la llamada varias veces sin que se produzcan efectos adversos. Esto resulta muy útil cuando queremos reproducir mensajes que no estamos seguros de que se hayan procesado, una forma habitual de recuperarse de un error.

Consideremos una llamada simple para sumar algunos puntos como resultado de que uno de nuestros clientes haya realizado un pedido. Podríamos realizar una llamada con el tipo de carga útil que se muestra en [el Ejemplo 11-1](#).

Ejemplo 11-1. Acreditación de puntos a una cuenta

```
<crédito>
  <cantidad>100</cantidad>
  <paraCuenta>1234</cuenta> </ crédito>
```

Si esta llamada se recibe varias veces, sumaremos 100 puntos varias veces. Por lo tanto, tal como está ahora, esta llamada no es idempotente. Sin embargo, con un poco más de información, permitimos que el banco de puntos haga que esta llamada sea idempotente, como se muestra en [el Ejemplo 11-2](#).

Ejemplo 11-2. Adición de más información al crédito de puntos para hacerlo idempotente

```
<crédito>
  <cantidad>100</cantidad>
  <paraCuenta>1234</cuenta>
  <razón>
    <forPurchase>4567</forPurchase> </reason> <
  credit>
```

Ahora sabemos que este crédito se relaciona con un pedido específico, 4567. Suponiendo que pudiéramos recibir solo un crédito por un pedido determinado, podríamos aplicar este crédito nuevamente sin aumentar la cantidad total de puntos.

Este mecanismo funciona igual de bien con la colaboración basada en eventos y puede ser especialmente útil si tienes varias instancias del mismo tipo de servicio suscrito a eventos.

Incluso si almacenamos qué eventos se han procesado, con algunas formas de entrega de mensajes asincrónicos puede haber pequeñas ventanas en las que dos trabajadores pueden ver el mismo mensaje. Al procesar los eventos de manera idempotente, nos aseguramos de que esto no nos cause ningún problema.

Algunas personas se obsesionan demasiado con este concepto y suponen que significa que las llamadas posteriores con los mismos parámetros no pueden tener ningún impacto, lo que nos deja en una posición interesante.

Realmente, nos gustaría registrar el hecho de que se recibió una llamada en nuestros registros, por ejemplo. Queremos registrar el tiempo de respuesta de la llamada y recopilar estos datos para su seguimiento. El punto clave aquí es que es la operación comercial subyacente la que estamos considerando idempotente, no el estado completo del sistema.

Algunos de los verbos HTTP, como GET y PUT, están definidos en la especificación HTTP para ser idempotentes, pero para que ese sea el caso, dependen de que su servicio maneje estas llamadas en

de manera idempotente. Si comienza a hacer que estos verbos sean no idempotentes, pero los que llaman creen que pueden ejecutarlos de manera segura repetidamente, puede meterse en problemas. Recuerde, ¡solo porque esté usando HTTP como protocolo subyacente no significa que obtenga todo gratis!

Escalada

En general, escalamos nuestros sistemas por una de dos razones. En primer lugar, para ayudar a lidiar con las fallas: si nos preocupa que algo falle, entonces tener más de eso ayudará, ¿no es así? En segundo lugar, escalamos para mejorar el rendimiento, ya sea en términos de manejar más carga, reducir la latencia o ambas cosas. Veamos algunas técnicas de escalamiento comunes que podemos usar y pensemos en cómo se aplican a las arquitecturas de microservicios.

Ve más grande

Algunas operaciones pueden beneficiarse simplemente de una mayor potencia. Conseguir una máquina más grande con una CPU más rápida y una mejor E/S a menudo puede mejorar la latencia y el rendimiento, lo que le permite procesar más trabajo en menos tiempo. Sin embargo, esta forma de escalamiento, a menudo denominada escalamiento vertical, puede ser costosa: a veces, un servidor grande puede costar más que dos servidores más pequeños con la misma potencia bruta combinada, especialmente cuando se empiezan a utilizar máquinas realmente grandes. A veces nuestro software por sí solo no puede hacer mucho con los recursos adicionales que tiene disponibles.

Las máquinas más grandes a menudo nos dan más núcleos de CPU, pero no hay suficiente software escrito para aprovecharlos. El otro problema es que esta forma de escalado puede no hacer mucho para mejorar la resistencia de nuestro servidor si solo tenemos uno de ellos. No obstante, esto puede ser una buena victoria rápida, especialmente si estás usando un proveedor de virtualización que te permite cambiar el tamaño de las máquinas fácilmente.

División de cargas de trabajo

Como se describe en [el Capítulo 6](#), tener un solo microservicio por host es ciertamente preferible a un modelo de múltiples servicios por host. Sin embargo, inicialmente, muchas personas deciden que coexistan múltiples microservicios en una sola máquina para mantener bajos los costos o simplificar la administración del host (aunque esa es una razón discutible). Como los microservicios son procesos independientes que se comunican a través de la red, debería ser una tarea fácil moverlos a sus propios hosts para mejorar el rendimiento y la escalabilidad. Esto también puede aumentar la resiliencia del sistema, ya que una interrupción en un solo host afectará a una cantidad reducida de microservicios.

Por supuesto, también podríamos aprovechar la necesidad de una mayor escala para dividir un microservicio existente en partes para manejar mejor la carga. Como ejemplo simplista, imaginemos que nuestro servicio de cuentas brinda la capacidad de crear y administrar las cuentas financieras de clientes individuales, pero también expone una API para ejecutar consultas para generar informes. Esta capacidad de consulta coloca una carga significativa en el sistema. La capacidad de consulta se considera no crítica, ya que no es necesaria para mantener el flujo de pedidos durante el día. Sin embargo, la capacidad de administrar los registros financieros de nuestros clientes es crítica y no podemos permitirnos que no funcione. Al dividir estas dos capacidades en servicios separados, reducimos la carga en el servicio de cuentas crítico e introducimos un nuevo servicio de informes de cuentas que está diseñado no solo con las consultas en mente (quizás utilizando algunas de las técnicas que describimos en el [Capítulo 4](#)), sino también como un sistema no crítico que no necesita implementarse de una manera tan resiliente como el servicio de cuentas principal.

Distribuyendo el riesgo

Una forma de aumentar la resiliencia es asegurarse de no poner todos los huevos en una sola canasta.

Un ejemplo simple de esto es asegurarse de no tener varios servicios en un host, ya que una interrupción afectaría a varios servicios. Pero consideremos qué significa host .

En la mayoría de las situaciones actuales, un host es en realidad un concepto virtual. ¿Qué pasa si tengo todos mis servicios en diferentes hosts, pero todos esos hosts son en realidad hosts virtuales que se ejecutan en el mismo equipo físico? Si ese equipo falla, podría perder varios servicios. Algunas plataformas de virtualización le permiten garantizar que sus hosts se distribuyan en varios equipos físicos diferentes para reducir esta posibilidad.

En el caso de las plataformas de virtualización interna, es una práctica habitual asignar la partición raíz de la máquina virtual a una única SAN (red de área de almacenamiento). Si esa SAN deja de funcionar, puede hacer caer todas las máquinas virtuales conectadas. Las SAN son grandes, caras y están diseñadas para no fallar. Dicho esto, he tenido SAN grandes y costosas que fallaron al menos dos veces en los últimos 10 años, y cada vez los resultados fueron bastante graves.

Otra forma común de separación para reducir las fallas es asegurarse de que no todos los servicios se ejecuten en un solo rack en el centro de datos o que los servicios se distribuyan en más de un centro de datos. Si utiliza un proveedor de servicios subyacente, es importante saber si se ofrece un acuerdo de nivel de servicio (SLA) y planificar en consecuencia. Si necesita asegurarse de que sus servicios no estén inactivos durante más de cuatro horas cada trimestre, pero su proveedor de alojamiento solo puede garantizar un tiempo de inactividad de ocho horas por trimestre, debe cambiar el SLA o buscar una solución alternativa.

AWS, por ejemplo, se divide en regiones, que se pueden considerar como nubes distintas. Cada región se divide a su vez en dos o más zonas de disponibilidad (AZ). Las AZ son el equivalente de AWS a un centro de datos. Es esencial tener servicios distribuidos en varias zonas de disponibilidad, ya que AWS no ofrece garantías sobre la disponibilidad de un solo nodo, o incluso de una zona de disponibilidad completa. Para su servicio de cómputo, ofrece solo un tiempo de actividad del 99,95 % durante un período mensual determinado de la región en su conjunto, por lo que querrá distribuir sus cargas de trabajo en varias zonas de disponibilidad dentro de una sola región. Para algunas personas, esto no es suficiente y, en su lugar, ejecutan sus servicios también en varias regiones.

Por supuesto, cabe señalar que, como los proveedores ofrecen una garantía de SLA, tienden a limitar su responsabilidad. Si no cumplen sus objetivos, esto le cuesta clientes y una gran cantidad de dinero, es posible que tenga que revisar los contratos para ver si puede recuperar algo de ellos. Por lo tanto, le recomiendo encarecidamente que comprenda el impacto que tiene el incumplimiento de las obligaciones de un proveedor con usted y que determine si necesita tener un plan B (o C) en el bolsillo. Más de un cliente con el que he trabajado tenía una plataforma de alojamiento de recuperación ante desastres con un proveedor diferente, por ejemplo, para asegurarse de que no fueran demasiado vulnerables a los errores de una empresa.

Equilibrio de carga

Cuando necesita que su servicio sea resistente, desea evitar puntos únicos de falla. Para un microservicio típico que expone un punto final HTTP sincrónico, la forma más fácil de lograr esto es tener múltiples hosts que ejecuten su instancia de microservicio, ubicados detrás de un balanceador de carga, como se muestra en [la Figura 11-4](#). Para los consumidores del microservicio, no sabe si está hablando con una instancia de microservicio o con cien.

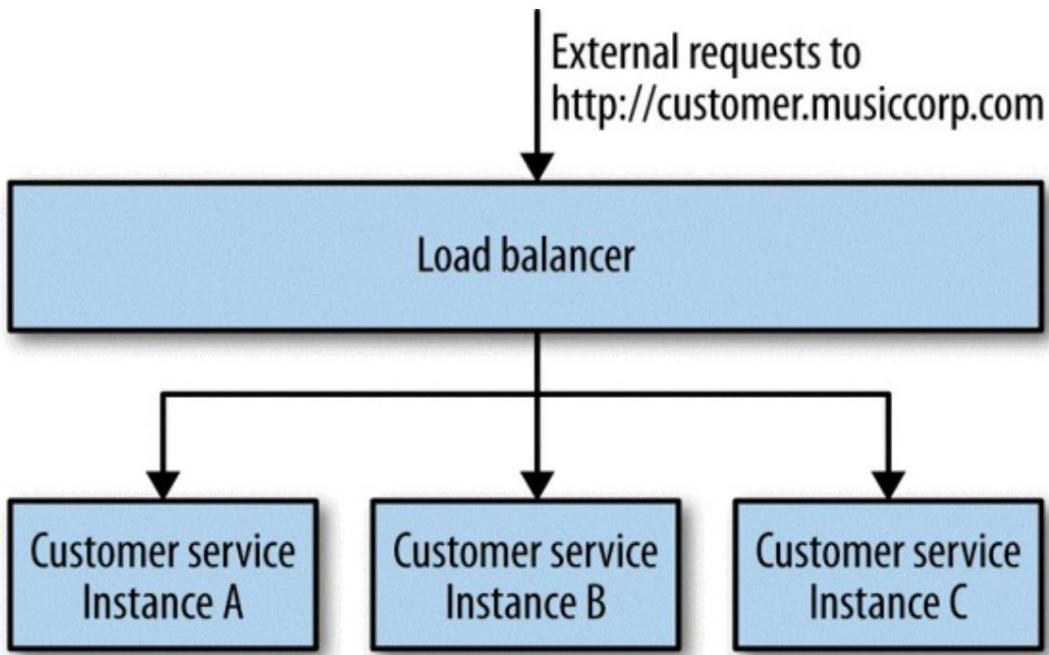


Figura 11-4. Ejemplo de un enfoque de equilibrio de carga para escalar la cantidad de instancias de servicio al cliente

Los平衡adores de carga vienen en todas las formas y tamaños, desde dispositivos de hardware grandes y costosos hasta balanceadores de carga basados en software como mod_proxy. Todos ellos comparten algunas capacidades clave. Distribuyen las llamadas que se les envían a una o más instancias según algún algoritmo, eliminan las instancias cuando ya no están en buen estado y, con suerte, las vuelven a agregar cuando lo están.

Algunos平衡adores de carga ofrecen funciones útiles. Una de las más comunes es la terminación SSL, en la que las conexiones HTTPS entrantes al balanceador de carga se transforman en conexiones HTTP una vez que llegan a la instancia. Históricamente, la sobrecarga de la gestión de SSL era lo suficientemente significativa como para que fuera bastante útil tener un balanceador de carga que se encargara de este proceso por ti. Hoy en día, esto tiene que ver tanto con simplificar la configuración de los hosts individuales que ejecutan la instancia. Sin embargo, el objetivo de usar HTTPS es garantizar que las solicitudes no sean vulnerables a un ataque de intermediario, como analizamos en el [Capítulo 9](#), por lo que si usamos la terminación SSL, potencialmente nos estamos exponiendo un poco. Una mitigación es tener todas las instancias del microservicio dentro de una sola VLAN, como vemos en [la Figura 11-5](#).

Una VLAN es una red de área local virtual, que está aislada de tal manera que las solicitudes desde fuera de ella solo pueden llegar a través de un enrutador, y en este caso nuestro enrutador también es nuestro balanceador de carga con terminación SSL. La única comunicación con el microservicio desde fuera de la VLAN se realiza a través de HTTPS, pero internamente todo es HTTP.

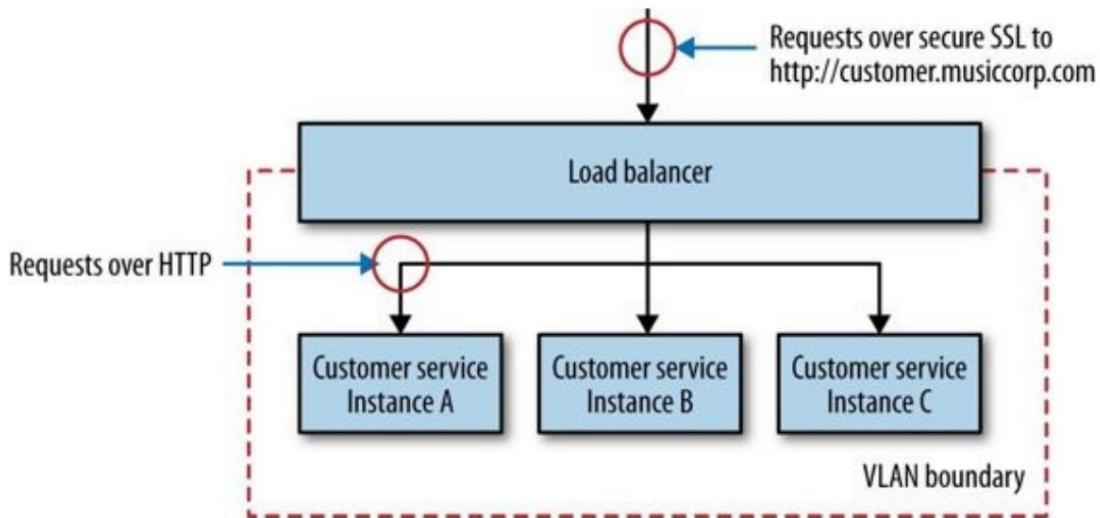


Figura 11-5. Uso de la terminación HTTPS en el balanceador de carga con una VLAN para mejorar la seguridad

AWS ofrece balanceadores de carga que terminan en HTTPS en forma de ELB (balanceadores de carga elásticos) y puede utilizar sus grupos de seguridad o nubes privadas virtuales (VPC) para implementar la VLAN. De lo contrario, un software como mod_proxy puede desempeñar una función similar a la de un balanceador de carga de software. Muchas organizaciones tienen balanceadores de carga de hardware, que pueden ser difíciles de automatizar. En tales circunstancias, me he encontrado abogando por balanceadores de carga de software ubicados detrás de los balanceadores de carga de hardware para permitir que los equipos tengan la libertad de reconfigurarlos según sea necesario. Debe tener en cuenta el hecho de que, con demasiada frecuencia, los balanceadores de carga de hardware son puntos únicos de falla. Cualquiera sea el enfoque que adopte, al considerar la configuración de un balanceador de carga, trátelo como trata la configuración de su servicio: asegúrese de que esté almacenado en el control de versiones y se pueda aplicar automáticamente.

Los balanceadores de carga nos permiten agregar más instancias de nuestro microservicio de una manera transparente para cualquier consumidor de servicios. Esto nos brinda una mayor capacidad para manejar la carga y también reduce el impacto de una falla en un solo host. Sin embargo, muchos, si no la mayoría, de sus microservicios tendrán algún tipo de almacenamiento de datos persistente, probablemente una base de datos ubicada en una máquina diferente. Si tenemos múltiples instancias de microservicio en diferentes máquinas, pero solo un único host que ejecuta la instancia de base de datos, nuestra base de datos sigue siendo una única fuente de falla. Hablaremos sobre patrones para manejar esto en breve.

Sistemas basados en trabajadores

El equilibrio de carga no es la única forma de que varias instancias de su servicio compartan la carga y reduzcan la fragilidad. Según la naturaleza de las operaciones, un sistema basado en trabajadores podría ser igual de eficaz. En este caso, una colección de instancias trabaja en una lista de tareas pendientes compartida. Podría tratarse de varios procesos Hadoop o quizás de varios oyentes de una cola de trabajo compartida. Este tipo de operaciones son adecuadas para trabajos por lotes o asíncronos. Piense en tareas como el procesamiento de miniaturas de imágenes, el envío de correos electrónicos o la generación de informes.

El modelo también funciona bien en picos de carga, donde se pueden poner en marcha instancias adicionales a pedido para que coincidan con la carga entrante. Mientras la cola de trabajo en sí sea resiliente, este modelo se puede utilizar para escalar tanto para mejorar el rendimiento del trabajo como para mejorar la resiliencia: el impacto de que un trabajador falle (o no esté presente) es fácil de manejar. El trabajo llevará más tiempo, pero no se pierde nada.

He visto que esto funciona bien en organizaciones donde hay mucha capacidad de cómputo sin usar en determinados momentos del día. Por ejemplo, durante la noche es posible que no necesite tantas máquinas para ejecutar su sistema de comercio electrónico, por lo que puede usarlas temporalmente para ejecutar trabajadores para un trabajo de informes.

En el caso de los sistemas basados en trabajadores, aunque los trabajadores en sí no necesitan ser tan confiables, el sistema que contiene el trabajo a realizar sí lo es. Esto se puede solucionar ejecutando un agente de mensajes persistentes, por ejemplo, o quizás un sistema como Zookeeper. La ventaja aquí es que si utilizamos software existente para este propósito, alguien ha hecho gran parte del trabajo duro por nosotros. Sin embargo, todavía necesitamos saber cómo configurar y mantener estos sistemas de manera resiliente.

Empezando de nuevo

La arquitectura que le permita empezar puede no ser la arquitectura que le permita seguir adelante cuando su sistema tenga que gestionar volúmenes de carga muy diferentes. Como dijo Jeff Dean en su presentación “Desafíos en la construcción de sistemas de recuperación de información a gran escala” (Conferencia WSDM 2009), se debe “diseñar para un crecimiento de ~10x, pero planificar la reescritura antes de ~100x”. En ciertos puntos, es necesario hacer algo bastante radical para respaldar el siguiente nivel de crecimiento.

Recordemos la historia de Gilt, que abordamos en [el capítulo 6](#). Una aplicación Rails monolítica y sencilla funcionó bien para Gilt durante dos años. Su negocio fue cada vez más exitoso, lo que significó más clientes y más carga. En un momento dado, la empresa tuvo que rediseñar la aplicación para que pudiera manejar la carga que estaba viendo.

Un rediseño puede implicar la división de un monolito existente, como sucedió con Gilt. O puede implicar la elección de nuevos almacenes de datos que puedan manejar mejor la carga, algo que analizaremos en breve. También puede implicar la adopción de nuevas técnicas, como pasar de sistemas de solicitud/respuesta sincrónica a sistemas basados en eventos, adoptar nuevas plataformas de implementación, cambiar pilas de tecnología completas o cualquier otra cosa intermedia.

Existe el peligro de que la gente vea la necesidad de rediseñar cuando se alcanzan ciertos umbrales de escala como una razón para construir para una escala masiva desde el principio. Esto puede ser desastroso. Al comienzo de un nuevo proyecto, a menudo no sabemos exactamente qué queremos construir, ni sabemos si tendrá éxito. Necesitamos poder experimentar rápidamente y entender qué capacidades necesitamos construir. Si intentáramos construir para una escala masiva desde el principio, terminaríamos cargando por adelantado una enorme cantidad de trabajo para prepararnos para una carga que tal vez nunca llegue, mientras desviamos esfuerzos de actividades más importantes, como entender si alguien realmente querrá usar nuestro producto. Eric Ries cuenta la historia de pasar seis meses construyendo un producto que nadie nunca descargó. Reflexionó que podría haber puesto un enlace en una página web que generara un error 404 cuando la gente hiciera clic en él para ver si había alguna demanda, haber pasado seis meses en la playa en lugar de eso, ¡y haber aprendido lo mismo!

La necesidad de cambiar nuestros sistemas para hacer frente a la escala no es un signo de fracaso. Es un signo de éxito.

Escalado de bases de datos

Escalar microservicios sin estado es bastante sencillo. Pero, ¿qué sucede si almacenamos datos en una base de datos? También necesitaremos saber cómo escalar eso. Los distintos tipos de bases de datos ofrecen distintas formas de escalado, y comprender qué forma se adapta mejor a su caso de uso le garantizará la selección de la tecnología de base de datos adecuada desde el principio.

Disponibilidad del servicio versus durabilidad de los datos

Para empezar, es importante separar el concepto de disponibilidad del servicio del de durabilidad de los datos en sí.

Es necesario entender que son dos cosas diferentes y, por lo tanto, tendrán soluciones diferentes.

Por ejemplo, podría almacenar una copia de todos los datos escritos en mi base de datos en un sistema de archivos resistente. Si la base de datos deja de funcionar, mis datos no se pierden, ya que tengo una copia, pero la base de datos en sí no está disponible, lo que puede hacer que mi microservicio tampoco esté disponible. Un modelo más común sería usar una base de datos de respaldo. Todos los datos escritos en la base de datos principal se copian en la base de datos de réplica de respaldo. Si la base de datos principal deja de funcionar, mis datos están seguros, pero sin un mecanismo para volver a ponerlos en funcionamiento o promover la réplica a la base de datos principal, no tenemos una base de datos disponible, aunque nuestros datos estén seguros.

Escalado para lecturas

Muchos servicios se basan principalmente en la lectura. Pensemos en un servicio de catálogo que almacena información de los artículos que tenemos a la venta. Agregamos registros de artículos nuevos de manera bastante irregular y no sería de extrañar que obtengamos más de 100 lecturas de los datos de nuestro catálogo por cada escritura. Afortunadamente, escalar para las lecturas es mucho más fácil que escalar para las escrituras. El almacenamiento en caché de datos puede desempeñar un papel importante en este caso, y lo analizaremos con más profundidad en breve. Otro modelo es utilizar réplicas de lectura.

En un sistema de gestión de bases de datos relacionales (RDBMS) como MySQL o Postgres, los datos se pueden copiar desde un nodo principal a una o más réplicas. Esto se hace a menudo para garantizar que se mantenga segura una copia de nuestros datos, pero también podemos usarlo para distribuir nuestras lecturas. Un servicio podría dirigir todas las escrituras al único nodo principal, pero distribuir las lecturas a una o más réplicas de lectura, como vemos en [la Figura 11-6](#). La replicación de la base de datos principal a las réplicas ocurre en algún momento después de la escritura. Esto significa que con esta técnica, las lecturas a veces pueden ver datos obsoletos hasta que se complete la replicación. Finalmente, las lecturas verán los datos consistentes. Este tipo de configuración se denomina eventualmente consistente y, si puede manejar la inconsistencia temporal, es una forma bastante fácil y común de ayudar a escalar los sistemas. Analizaremos esto con más profundidad en breve cuando veamos el teorema CAP.

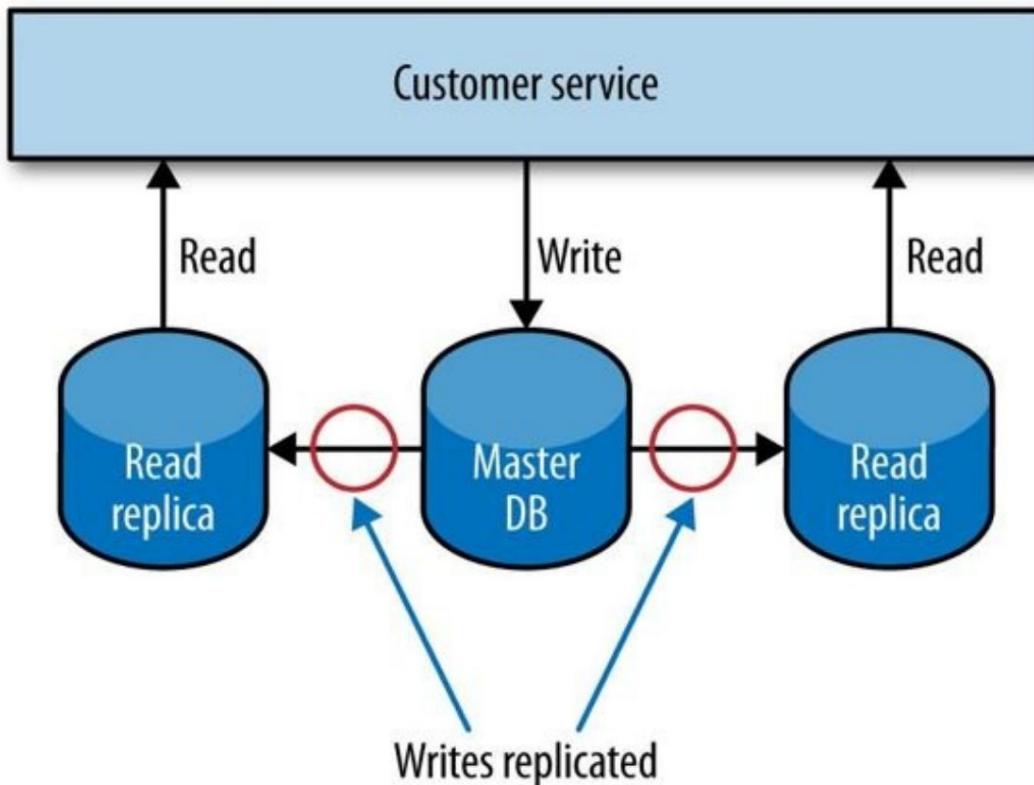


Figura 11-6. Uso de réplicas de lectura para escalar las lecturas

Hace años, usar réplicas de lectura para escalar estaba de moda, aunque hoy en día te sugeriría que primero consideres el almacenamiento en caché, ya que puede ofrecer mejoras mucho más significativas en el rendimiento, a menudo con menos trabajo.

Escalado para escrituras

Las lecturas son comparativamente fáciles de escalar. ¿Qué sucede con las escrituras? Un enfoque es utilizar la fragmentación. Con la fragmentación, tienes múltiples nodos de base de datos. Toma un fragmento de datos que se va a escribir, aplica una función hash a la clave de los datos y, en función del resultado de la función, aprendes a dónde enviar los datos. Para elegir un ejemplo muy simple (y en realidad malo), imagina que los registros de clientes A–M van a una instancia de base de datos y los N–Z a otra. Puedes gestionar esto tú mismo en tu aplicación, pero algunas bases de datos, como Mongo, se encargan de gran parte de esto por ti.

La complejidad de la fragmentación para escrituras proviene del manejo de consultas. Buscar un registro individual es fácil, ya que puedo simplemente aplicar la función hash para encontrar en qué instancia deberían estar los datos y luego recuperarlos del fragmento correcto. Pero ¿qué sucede con las consultas que abarcan los datos en varios nodos, por ejemplo, encontrar todos los clientes que tienen más de 18 años? Si desea consultar todos los fragmentos, debe consultar cada fragmento individual y unirlos en la memoria, o tener un almacén de lectura alternativo donde ambos conjuntos de datos estén disponibles. A menudo, las consultas entre fragmentos se manejan mediante un mecanismo asincrónico, utilizando resultados almacenados en caché. Mongo utiliza trabajos de mapa/reducción, por ejemplo, para realizar estas consultas.

Una de las preguntas que surgen con los sistemas fragmentados es qué sucede si quiero agregar un nodo de base de datos adicional. En el pasado, esto solía requerir un tiempo de inactividad significativo, especialmente para clústeres grandes, ya que era posible que tuviera que desconectar toda la base de datos y reequilibrar los datos. Más recientemente, más sistemas admiten la adición de fragmentos adicionales a un sistema en vivo, donde el reequilibrio de los datos se realiza en segundo plano; Cassandra, por ejemplo, maneja esto muy bien. Sin embargo, agregar fragmentos a un clúster existente no es para los débiles de corazón, así que asegúrese de probar esto a fondo.

La fragmentación para escrituras puede escalar para el volumen de escritura, pero no mejorar la resiliencia. Si los registros de cliente A–M siempre van a la instancia X y la instancia X no está disponible, se puede perder el acceso a los registros A–M. Cassandra ofrece capacidades adicionales en este caso, donde podemos asegurar que los datos se repliquen en múltiples nodos en un anillo (término de Cassandra para una colección de nodos de Cassandra).

Como puede haber deducido de esta breve descripción general, la escalabilidad de las bases de datos para las escrituras es donde las cosas se complican y donde las capacidades de las distintas bases de datos realmente comienzan a diferenciarse. A menudo veo personas que cambian la tecnología de las bases de datos cuando comienzan a alcanzar límites en la facilidad con la que pueden escalar su volumen de escritura existente. Si esto le sucede, comprar una máquina más grande suele ser la forma más rápida de resolver el problema, pero en segundo plano es posible que desee analizar sistemas como Cassandra, Mongo o Riak para ver si sus modelos de escalabilidad alternativos pueden ofrecerle una mejor solución a largo plazo.

Infraestructura de base de datos compartida

Algunos tipos de bases de datos, como los RDBMS tradicionales, separan el concepto de la base de datos en sí y el esquema. Esto significa que una base de datos en ejecución podría albergar varios esquemas independientes, uno para cada microservicio. Esto puede ser muy útil en términos de reducir la cantidad de máquinas que necesitamos para ejecutar nuestro sistema, pero estamos introduciendo un único punto de falla significativo. Si esta infraestructura de base de datos falla, puede afectar a varios microservicios a la vez, lo que podría provocar una interrupción catastrófica. Si está ejecutando este tipo de configuración, asegúrese de considerar los riesgos. Y asegúrese de que la base de datos en sí sea lo más resistente posible.

CQRS

El patrón de segregación de responsabilidades de consulta y comando (CQRS) se refiere a un modelo alternativo para almacenar y consultar información. Con bases de datos normales, utilizamos un sistema para realizar modificaciones a los datos y consultarlos. Con CQRS, una parte del sistema se ocupa de los comandos, que capturan las solicitudes para modificar el estado, mientras que otra parte del sistema se ocupa de las consultas.

Los comandos se envían para solicitar cambios de estado. Estos comandos se validan y, si funcionan, se aplicarán al modelo. Los comandos deben contener información sobre su propósito. Se pueden procesar de forma sincrónica o asincrónica, lo que permite que diferentes modelos gestionen el escalamiento; por ejemplo, podríamos simplemente poner en cola las solicitudes entrantes y procesarlas más tarde.

La conclusión clave aquí es que los modelos internos utilizados para manejar comandos y consultas son completamente independientes. Por ejemplo, podría optar por manejar y procesar comandos como eventos, tal vez simplemente almacenando la lista de comandos en un almacén de datos (un proceso conocido como abastecimiento de eventos). Mi modelo de consulta podría consultar un almacén de eventos y crear proyecciones a partir de eventos almacenados para ensamblar el estado de los objetos de dominio, o podría simplemente tomar una fuente de la parte de comandos del sistema para actualizar un tipo diferente de almacén. En muchos sentidos, obtenemos los mismos beneficios de las réplicas de lectura que analizamos anteriormente, sin el requisito de que el almacén de respaldo para las réplicas sea el mismo que el almacén de datos utilizado para manejar las modificaciones de datos.

Esta forma de separación permite distintos tipos de escalabilidad. Las partes de comandos y consultas de nuestro sistema podrían residir en distintos servicios o en distintos hardware, y podrían utilizar tipos de almacenamiento de datos radicalmente diferentes. Esto puede desbloquear una gran cantidad de formas de gestionar la escalabilidad. Incluso se podrían admitir distintos tipos de formato de lectura al tener múltiples implementaciones de la parte de consulta, tal vez admitiendo una representación basada en gráficos de los datos o una forma basada en clave/valor de los datos.

Sin embargo, tenga cuidado: este tipo de patrón es un gran cambio con respecto a un modelo en el que un único almacén de datos gestiona todas nuestras operaciones CRUD. ¡He visto a más de un equipo de desarrollo experimentado luchar por conseguir que este patrón funcione!

Aimacenamiento en caché

El almacenamiento en caché es una optimización del rendimiento de uso común mediante la cual se almacena el resultado anterior de alguna operación, de modo que las solicitudes posteriores puedan utilizar este valor almacenado en lugar de gastar tiempo y recursos en recalcularlo. En la mayoría de los casos, el almacenamiento en caché consiste en eliminar viajes de ida y vuelta innecesarios a bases de datos u otros servicios para entregar resultados más rápido. Si se utiliza bien, puede producir enormes beneficios en el rendimiento. La razón por la que HTTP escala tan bien al manejar grandes cantidades de solicitudes es que el concepto de almacenamiento en caché está incorporado.

Incluso con una aplicación web monolítica sencilla, hay muchas opciones sobre dónde y cómo almacenar en caché. Con una arquitectura de microservicios, donde cada servicio es su propia fuente de datos y comportamiento, tenemos muchas más opciones sobre dónde y cómo almacenar en caché.

En un sistema distribuido, normalmente pensamos en el almacenamiento en caché ya sea del lado del cliente o del lado del servidor. Pero ¿cuál es mejor?

Almacenamiento en caché del lado del cliente, proxy y servidor

En el almacenamiento en caché del lado del cliente, el cliente almacena el resultado almacenado en caché. El cliente decide cuándo (y si) recupera una copia nueva. Lo ideal es que el servicio descendente proporcione sugerencias para ayudar al cliente a entender qué hacer con la respuesta, de modo que sepa cuándo y si debe realizar una nueva solicitud. Con el almacenamiento en caché de proxy, se coloca un proxy entre el cliente y el servidor. Un gran ejemplo de esto es el uso de un proxy inverso o una red de entrega de contenido (CDN). Con el almacenamiento en caché del lado del servidor, el servidor maneja la responsabilidad del almacenamiento en caché, tal vez haciendo uso de un sistema como Redis o Memcache, o incluso un simple caché en memoria.

La opción que tenga más sentido depende de lo que intentes optimizar. El almacenamiento en caché del lado del cliente puede ayudar a reducir drásticamente las llamadas de red y puede ser una de las formas más rápidas de reducir la carga en un servicio descendente. En este caso, el cliente está a cargo del comportamiento del almacenamiento en caché y, si quieras realizar cambios en la forma en que se realiza el almacenamiento en caché, implementar los cambios en una cantidad de consumidores puede resultar difícil. La invalidación de datos obsoletos también puede ser más complicada, aunque analizaremos algunos mecanismos de adaptación para esto en un momento.

Con el almacenamiento en caché de proxy, todo es opaco tanto para el cliente como para el servidor. Esta suele ser una forma muy sencilla de añadir almacenamiento en caché a un sistema existente. Si el proxy está diseñado para almacenar en caché tráfico genérico, también puede almacenar en caché más de un servicio; un ejemplo común es un proxy inverso como Squid o Varnish, que puede almacenar en caché cualquier tráfico HTTP. Tener un proxy entre el cliente y el servidor introduce saltos de red adicionales, aunque en mi experiencia es muy raro que esto cause problemas, ya que las optimizaciones de rendimiento resultantes del almacenamiento en caché en sí superan cualquier costo de red adicional.

Con el almacenamiento en caché del lado del servidor, todo es opaco para los clientes; no necesitan preocuparse por nada. Con un caché cerca o dentro de un límite de servicio, puede ser más fácil razonar sobre cuestiones como la invalidación de datos o rastrear y optimizar los accesos al caché. En una situación en la que tienes varios tipos de clientes, un caché del lado del servidor podría ser la forma más rápida de mejorar el rendimiento.

En todos los sitios web públicos en los que he trabajado, hemos acabado utilizando una combinación de los tres enfoques. Sin embargo, en más de un sistema distribuido, he podido salir adelante sin ningún tipo de almacenamiento en caché. Pero todo se reduce a saber qué carga necesitas manejar, qué tan actualizados deben estar tus datos y qué puedes hacer tu sistema en este momento. Saber que tienes varias herramientas diferentes a tu disposición es solo el comienzo.

Almacenamiento en caché en HTTP

HTTP proporciona algunos controles realmente útiles para ayudarnos a almacenar en caché, ya sea en el lado del cliente o en el lado del servidor, que vale la pena comprender incluso si no estás usando HTTP.

En primer lugar, con HTTP, podemos usar directivas de control de caché en nuestras respuestas a los clientes. Estas directivas les indican a los clientes si deben almacenar en caché el recurso y, de ser así, durante cuánto tiempo deben almacenarlo en caché en segundos. También tenemos la opción de configurar un encabezado `Expires`, donde en lugar de decir durante cuánto tiempo se puede almacenar en caché un fragmento de contenido, especificamos una hora y fecha en la que un recurso debe considerarse obsoleto y debe recuperarse nuevamente. La naturaleza de los recursos que está compartiendo determina cuál es el más adecuado. El contenido de sitios web estáticos estándar, como CSS o imágenes, a menudo se adapta bien a un tiempo de vida (TTL) de control de caché simple. Por otro lado, si sabe de antemano cuándo se actualizará una nueva versión de un recurso, configurar un encabezado `Expires` tendrá más sentido. Todo esto es muy útil para evitar que un cliente necesite realizar una solicitud al servidor en primer lugar.

Además de `cache-control` y `Expires`, tenemos otra opción en nuestro arsenal de ventajas HTTP: las etiquetas de entidad o ETags. Una ETag se utiliza para determinar si el valor de un recurso ha cambiado. Si actualizo un registro de cliente, la URI del recurso es la misma, pero el valor es diferente, por lo que esperaría que la ETag cambie. Esto se vuelve poderoso cuando usamos lo que se llama un GET condicional. Al realizar una solicitud GET, podemos especificar encabezados adicionales, indicándole al servicio que nos envíe el recurso solo si se cumplen algunos criterios.

Por ejemplo, imaginemos que recuperamos un registro de cliente y su ETag aparece como `o5t6fk2sa`. Más adelante, quizás porque una directiva de control de caché nos ha dicho que el recurso debe considerarse obsoleto, queremos asegurarnos de obtener la última versión. Al emitir la siguiente solicitud GET, podemos pasar un `If-None-Match: o5t6fk2sa`. Esto le dice al servidor que queremos el recurso en la URI especificada, a menos que ya coincida con este valor de ETag. Si ya tenemos la versión actualizada, el servicio nos envía una respuesta `304 Not Modified`, que nos dice que tenemos la última versión. Si hay una versión más nueva disponible, obtenemos un `200 OK` con el recurso modificado y una nueva ETag para el recurso.

El hecho de que estos controles estén integrados en una especificación tan ampliamente utilizada significa que podemos aprovechar una gran cantidad de software preexistente que maneja el almacenamiento en caché por nosotros. Los servidores proxy inversos como Squid o Varnish pueden ubicarse de manera transparente en la red entre el cliente y el servidor, almacenando y haciendo caducar el contenido en caché según sea necesario. Estos sistemas están diseñados para atender una gran cantidad de solicitudes simultáneas muy rápido y son una forma estándar de escalar sitios web públicos. Las CDN como CloudFront de AWS o Akamai pueden garantizar que las solicitudes se enruten a cachés cerca del cliente que realiza la llamada, lo que garantiza que el tráfico no se vaya al otro lado del mundo cuando sea necesario. Y, de manera más prosaica, las bibliotecas de cliente HTTP y los cachés de cliente pueden manejar gran parte de este trabajo por nosotros.

Las etiquetas ETag, las fechas de caducidad y el control de caché pueden superponerse un poco y, si no tienes cuidado, puedes terminar brindando información contradictoria si decides usarlas todas. Para obtener más información, consulta

Para una discusión más detallada de los diversos méritos, eche un vistazo al libro **REST In Practice** (O'Reilly) o lea la sección 13 de la **especificación HTTP 1.1**, que describe cómo se supone que tanto los clientes como los servidores deben implementar estos diversos controles.

Si decide utilizar HTTP como protocolo entre servicios, vale la pena almacenar en caché en el cliente y reducir la necesidad de viajes de ida y vuelta al cliente. Si decide elegir un protocolo diferente, comprenda cuándo y cómo puede proporcionar sugerencias al cliente para ayudarlo a comprender durante cuánto tiempo puede almacenar en caché.

Almacenamiento en caché para escrituras

Aunque te encontrarás usando el almacenamiento en caché para lecturas con más frecuencia, hay algunos casos de uso en los que el almacenamiento en caché para escrituras tiene sentido. Por ejemplo, si haces uso de un caché de escritura en segundo plano, puedes escribir en un caché local y, en algún momento posterior, los datos se volcarán a una fuente descendente, probablemente la fuente de datos canónica. Esto puede ser útil cuando tienes ráfagas de escrituras o cuando hay una buena probabilidad de que los mismos datos se escriban varias veces. Cuando se utilizan para almacenar en búfer y potencialmente para escrituras por lotes, los cachés de escritura en segundo plano pueden ser una optimización adicional del rendimiento útil.

Con un caché de escritura posterior, si las escrituras almacenadas en el búfer son lo suficientemente persistentes, incluso si el servicio descendente no está disponible, podríamos poner en cola las escrituras y enviarlas cuando esté disponible nuevamente.

Almacenamiento en caché para la resiliencia

El almacenamiento en caché se puede utilizar para implementar resiliencia en caso de falla. Con el almacenamiento en caché del lado del cliente, si el servicio descendente no está disponible, el cliente podría decidir simplemente usar datos almacenados en caché pero potencialmente obsoletos. También podríamos usar algo como un proxy inverso para entregar datos obsoletos. Para algunos sistemas, estar disponible incluso con datos obsoletos es mejor que no devolver ningún resultado, pero esa es una decisión que deberá tomar. Obviamente, si no tenemos los datos solicitados en la caché, entonces no podemos hacer mucho para ayudar, pero hay formas de mitigar esto.

Una técnica que vi en The Guardian, y posteriormente en otros lugares, consistía en rastrear periódicamente el sitio web existente para generar una versión estática del sitio web que pudiera mostrarse en caso de interrupción del servicio. Aunque esta versión rastreada no era tan actualizada como el contenido almacenado en caché que se mostraba desde el sistema en vivo, en caso de urgencia podía garantizar que se mostrara una versión del sitio.

Ocultando el origen

Con una memoria caché normal, si una solicitud da como resultado un error de caché, la solicitud continúa hasta el origen para obtener los datos nuevos y el autor de la llamada se bloquea y espera el resultado. En circunstancias normales, esto es de esperar. Pero si sufrimos un error de caché masivo, tal vez porque falla una máquina entera (o un grupo de máquinas) que proporciona nuestra memoria caché, una gran cantidad de solicitudes llegarán al origen.

En el caso de los servicios que ofrecen datos que se pueden almacenar en caché con facilidad, es habitual que el origen en sí mismo se escala para manejar solo una fracción del tráfico total, ya que la mayoría de las solicitudes se atienden desde la memoria caché que se encuentra frente al origen. Si de repente nos encontramos con una multitud atronadora debido a que desaparece una región de caché completa, nuestro origen podría desaparecer.

Una forma de proteger el origen en una situación como esta es no permitir nunca que las solicitudes lleguen al origen en primer lugar. En cambio, el propio origen llena la caché de forma asíncrona cuando es necesario, como se muestra en la Figura 11-7. Si se produce un error en la caché, se activa un evento que el origen puede detectar y que le avisa de que necesita volver a llenar la caché. Por lo tanto, si un fragmento entero ha desaparecido, podemos reconstruir la caché en segundo plano. Podríamos decidir bloquear la solicitud original a la espera de que se vuelva a llenar la región, pero esto podría provocar una contención en la propia caché, lo que provocaría más problemas. Es más probable que, si priorizamos mantener estable el sistema, la solicitud original falle, pero fallará rápidamente.

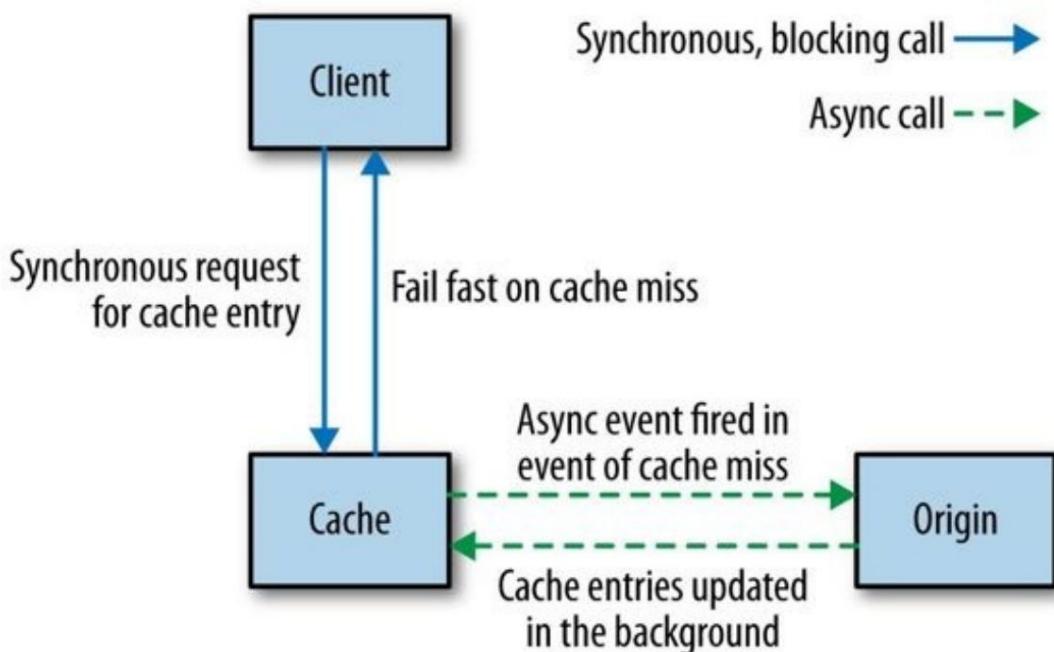


Figura 11-7. Ocultar el origen al cliente y rellenar la caché de forma asíncrona

Este tipo de enfoque puede no tener sentido en algunas situaciones, pero puede ser una forma de garantizar que el sistema siga funcionando cuando algunas partes de él fallan. Al rechazar solicitudes rápidamente y asegurarnos de no consumir recursos ni aumentar la latencia, evitamos que una falla en nuestra caché se propague en cascada y nos damos la oportunidad de recuperarnos.

Mantenlo simple

¡Tenga cuidado con el almacenamiento en caché en demasiados lugares! Cuantos más cachés haya entre usted y la fuente de datos nuevos, más obsoletos pueden estar los datos y más difícil puede ser determinar la actualidad de los datos que finalmente ve un cliente. Esto puede ser especialmente problemático con una arquitectura de microservicios en la que hay varios servicios involucrados en una cadena de llamadas. Nuevamente, cuanto más almacenamiento en caché tenga, más difícil será evaluar la actualidad de cualquier dato. Por lo tanto, si cree que un caché es una buena idea, manténgalo simple, límítense a uno y piense detenidamente antes de agregar más.

Envenenamiento de caché: una historia que sirve de advertencia

Con el almacenamiento en caché, solemos pensar que, si nos equivocamos, lo peor que puede pasar es que entreguemos datos obsoletos durante un tiempo. Pero, ¿qué ocurre si terminamos entregando datos obsoletos para siempre?

Anteriormente mencioné el proyecto en el que trabajé, en el que estábamos usando una aplicación estranguladora para ayudar a interceptar llamadas a múltiples sistemas heredados con el objetivo de retirarlos gradualmente.

Nuestra aplicación funcionó de manera eficaz como proxy. El tráfico hacia nuestra aplicación se enrutaba a través de la aplicación heredada. En el camino de regreso, hicimos algunas cosas de mantenimiento; por ejemplo, nos aseguramos de que los resultados de la aplicación heredada tuvieran los encabezados de caché HTTP adecuados aplicados.

Un día, poco después de un lanzamiento de rutina normal, algo extraño comenzó a suceder. Se había introducido un error por el cual un pequeño subconjunto de páginas no cumplía con una condición lógica en nuestro código de inserción de encabezado de caché, lo que hacía que no cambiáramos el encabezado en absoluto.

Lamentablemente, esta aplicación de flujo descendente también se había modificado en algún momento anterior para incluir un encabezado HTTP Expires: Never . Esto no había tenido ningún efecto antes, ya que estábamos anulando este encabezado. Ahora no lo estábamos haciendo.

Nuestra aplicación hacía un uso intensivo de Squid para almacenar en caché el tráfico HTTP y nos dimos cuenta del problema bastante rápido, ya que veíamos más solicitudes que pasaban por alto Squid para llegar a nuestros servidores de aplicaciones. Arreglamos el código del encabezado de caché y lanzamos una versión, y también borramos manualmente la región relevante de la caché de Squid. Sin embargo, eso no fue suficiente.

Como mencioné anteriormente, puedes almacenar en caché en varios lugares. Cuando se trata de ofrecer contenido a los usuarios de una aplicación web pública, puedes tener múltiples cachés entre tú y tu cliente. No solo podrías estar protegiendo tu sitio web con algo como una CDN, sino que algunos ISP hacen uso del almacenamiento en caché. ¿Puedes controlar esos cachés?

E incluso si pudieras, hay un caché sobre el cual tienes poco control: el caché del navegador del usuario.

Esas páginas con fecha de caducidad: nunca se quedan atascadas en los cachés de muchos de nuestros usuarios y nunca se invalidan hasta que el caché se llena o el usuario las limpia manualmente.

Claramente no podíamos lograr que ninguna de las dos cosas sucediera; nuestra única opción era cambiar las URL de estas páginas para que se volvieran a cargar.

El almacenamiento en caché puede ser realmente muy poderoso, pero es necesario comprender la ruta completa de los datos almacenados en caché desde el origen hasta el destino para apreciar realmente sus complejidades y lo que puede salir mal.

Escalado automático

Si tiene la suerte de contar con un aprovisionamiento totalmente automatizable de hosts virtuales y puede automatizar por completo la implementación de sus instancias de microservicios, entonces tiene los componentes básicos que le permitirán escalar automáticamente sus microservicios.

Por ejemplo, también podrías hacer que el escalado se active en función de tendencias conocidas. Es posible que sepas que la carga máxima de tu sistema es entre las 9:00 a. m. y las 5:00 p. m., por lo que activas instancias adicionales a las 8:45 a. m. y las apagas a las 5:15 p. m. Si estás usando algo como AWS (que tiene un muy buen soporte para el escalado automático integrado), apagar las instancias que ya no necesitas te ayudará a ahorrar dinero. Necesitarás datos para comprender cómo cambia tu carga con el tiempo, de un día para otro, de una semana para otra. Algunas empresas también tienen ciclos estacionales obvios, por lo que es posible que necesites datos que se remonten bastante tiempo atrás para tomar decisiones adecuadas.

Por otro lado, puedes ser reactivo, activando instancias adicionales cuando veas un aumento en la carga o una falla de instancia, y eliminando instancias cuando ya no las necesites. Saber qué tan rápido puedes escalar una vez que detectes una tendencia al alza es clave. Si sabes que solo recibirás un aviso de un par de minutos sobre un aumento en la carga, pero escalar te llevará al menos 10 minutos, sabes que necesitarás mantener capacidad adicional para cubrir esta brecha. Tener un buen conjunto de pruebas de carga es casi esencial aquí. Puedes usarlas para probar tus reglas de escalado automático. Si no tienes pruebas que puedan reproducir diferentes cargas que activarán el escalado, entonces solo descubrirás en producción si te equivocaste con las reglas. ¡Y las consecuencias del fracaso no son grandes!

Un sitio de noticias es un gran ejemplo de un tipo de negocio en el que es posible que desee una combinación de escalamiento predictivo y reactivo. En el último sitio de noticias en el que trabajé, vimos tendencias diarias muy claras: las visitas subían desde la mañana hasta la hora del almuerzo y luego empezaban a disminuir. Este patrón se repitió día tras día, con tráfico menos pronunciado durante el fin de semana. Eso nos dio una tendencia bastante clara que podría impulsar una ampliación (o reducción) proactiva de los recursos. Por otro lado, una noticia importante podría causar un aumento inesperado, lo que requeriría más capacidad a menudo con poca antelación.

En realidad, veo que el escalado automático se utiliza mucho más para gestionar fallos de instancias que para reaccionar a las condiciones de carga. AWS te permite especificar reglas como "Debe haber al menos 5 instancias en este grupo", de modo que si una deja de funcionar, se inicia automáticamente una nueva. He visto que este enfoque conduce a un divertido juego de golpear al topo cuando alguien se olvida de desactivar la regla y luego intenta desactivar las instancias para realizar tareas de mantenimiento, ¡solo para ver que siguen funcionando!

Tanto el escalamiento reactivo como el predictivo son muy útiles y pueden ayudarte a ser mucho más rentable si utiliza una plataforma que le permite pagar solo por los recursos informáticos que utiliza. Pero también requieren una observación cuidadosa de los datos disponibles.

Sugeriría utilizar primero el escalado automático para las condiciones de falla mientras recopila los datos. Una vez

Si desea comenzar a escalar para aumentar la carga, asegúrese de tener mucho cuidado de no reducir la escala demasiado rápido. En la mayoría de las situaciones, tener más potencia de procesamiento de la que necesita es mucho mejor que no tener suficiente.

Teorema CAP

Nos gustaría tenerlo todo, pero lamentablemente sabemos que no podemos. Y cuando se trata de sistemas distribuidos como los que construimos utilizando arquitecturas de microservicios, incluso tenemos una prueba matemática que nos dice que no podemos. Es posible que haya oído hablar del teorema CAP, especialmente en debates sobre los méritos de los distintos tipos de almacenamiento de datos. En esencia, nos dice que en un sistema distribuido tenemos tres cosas que podemos sacrificar entre sí: consistencia, disponibilidad y tolerancia a particiones. En concreto, el teorema nos dice que podemos mantener dos en modo de fallo.

La consistencia es la característica del sistema por la cual obtendré la misma respuesta si voy a varios nodos. La disponibilidad significa que cada solicitud recibe una respuesta. La tolerancia a la partición es la capacidad del sistema para manejar el hecho de que la comunicación entre sus partes a veces es imposible.

Desde que Eric Brewer publicó su conjectura original, la idea ha obtenido una demostración matemática. No voy a profundizar en las matemáticas de la demostración en sí, ya que no solo no es ese tipo de libro, sino que además puedo garantizar que me equivocaría. En lugar de eso, utilicemos algunos ejemplos prácticos que nos ayudarán a entender que, en el fondo, el teorema CAP es una destilación de un conjunto muy lógico de razonamientos.

Ya hemos hablado de algunas técnicas sencillas de escalado de bases de datos. Utilicemos una de ellas para investigar las ideas que sustentan el teorema CAP. Imaginemos que nuestro servicio de inventario se implementa en dos centros de datos separados, como se muestra en [la Figura 11-8](#). Como respaldo de nuestra instancia de servicio en cada centro de datos hay una base de datos, y estas dos bases de datos se comunican entre sí para intentar sincronizar los datos entre ellas. Las lecturas y escrituras se realizan a través del nodo de base de datos local, y se utiliza la replicación para sincronizar los datos entre los nodos.

Ahora pensemos en lo que sucede cuando algo falla. Imaginemos que algo tan simple como el enlace de red entre los dos centros de datos deja de funcionar. La sincronización en este punto falla.

Las escrituras realizadas en la base de datos principal en DC1 no se propagarán a DC2 y viceversa. La mayoría de las bases de datos que admiten estas configuraciones también admiten algún tipo de técnica de cola para garantizar que podamos recuperarnos de esto después, pero ¿qué sucede mientras tanto?

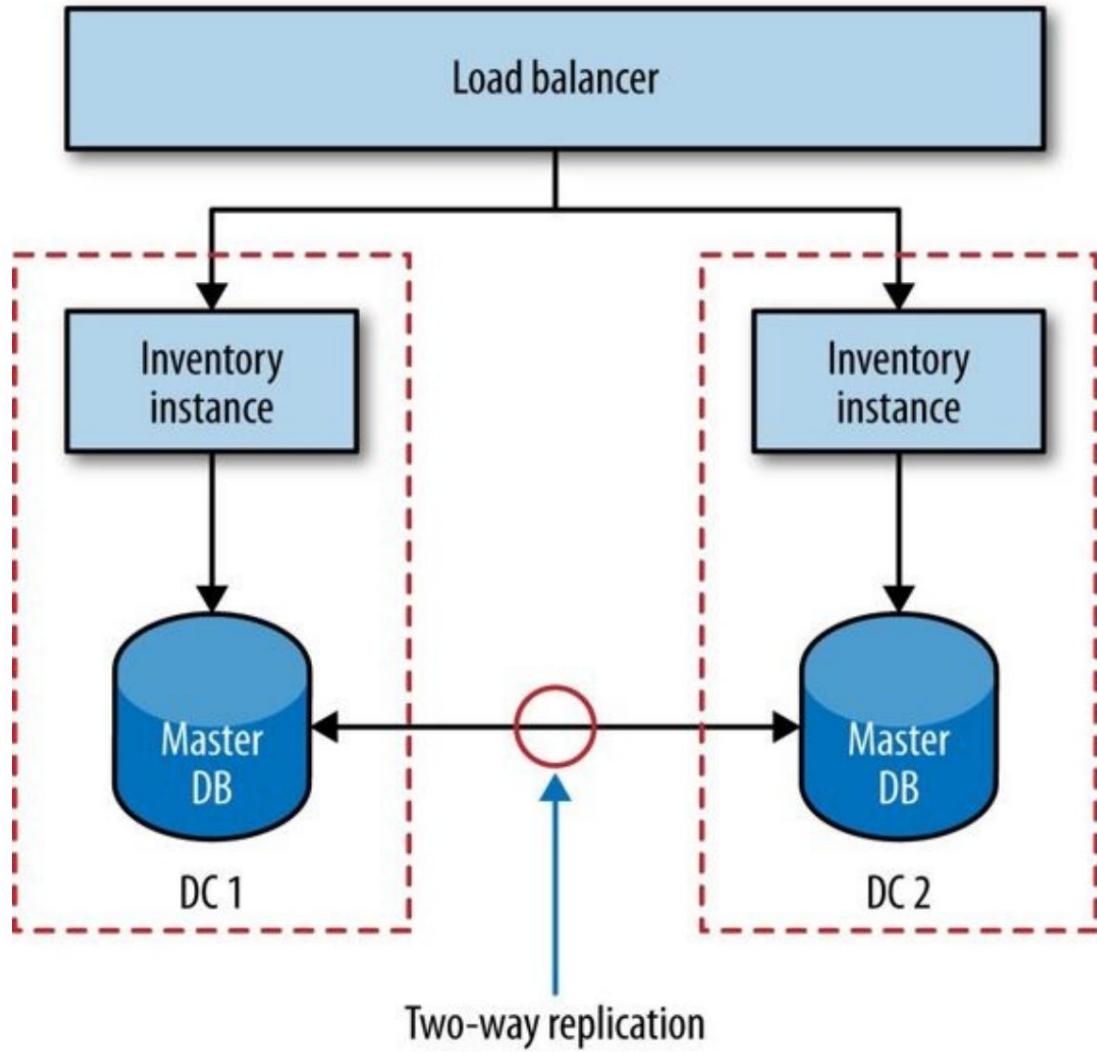


Figura 11-8. Uso de la replicación multiprimaria para compartir datos entre dos nodos de base de datos

Sacrificar la consistencia

Supongamos que no cerramos el servicio de inventario por completo. Si hago un cambio ahora en los datos en DC1, la base de datos en DC2 no lo ve. Esto significa que cualquier solicitud realizada a nuestro nodo de inventario en DC2 ve datos potencialmente obsoletos. En otras palabras, nuestro sistema aún está disponible en el sentido de que ambos nodos pueden atender solicitudes y hemos mantenido el sistema en funcionamiento a pesar de la partición, pero hemos perdido la coherencia. Esto a menudo se denomina sistema AP . No podemos mantener los tres.

Durante esta partición, si seguimos aceptando escrituras, aceptamos el hecho de que en algún momento en el futuro deberán volver a sincronizarse. Cuanto más dure la partición, más difícil puede resultar esta resincronización.

La realidad es que, incluso si no tenemos una falla de red entre los nodos de nuestra base de datos, la replicación de datos no es instantánea. Como se mencionó anteriormente, se dice que los sistemas que están dispuestos a ceder la consistencia para mantener la tolerancia y disponibilidad de las particiones son eventualmente consistentes; es decir, esperamos que en algún momento en el futuro todos los nodos vean los datos actualizados, pero no sucederá de inmediato, por lo que debemos vivir con la posibilidad de que los usuarios vean datos antiguos.

Sacrificar la disponibilidad

¿Qué pasa si necesitamos mantener la coherencia y queremos dejar algo más en su lugar?

Bueno, para mantener la coherencia, cada nodo de la base de datos necesita saber que la copia de los datos que tiene es la misma que la del otro nodo de la base de datos. Ahora bien, en la partición, si los nodos de la base de datos no pueden comunicarse entre sí, no pueden coordinarse para garantizar la coherencia. No podemos garantizar la coherencia, por lo que nuestra única opción es negarnos a responder a la solicitud. En otras palabras, hemos sacrificado la disponibilidad. Nuestro sistema es coherente y tolerante a particiones, o CP. En este modo, nuestro servicio tendría que encontrar la manera de degradar la funcionalidad hasta que se repare la partición y se puedan volver a sincronizar los nodos de la base de datos.

La coherencia entre varios nodos es realmente difícil. Hay pocas cosas (quizás ninguna) más difíciles en los sistemas distribuidos. Piénselo por un momento. Imagine que quiero leer un registro del nodo de la base de datos local. ¿Cómo sé que está actualizado? Tengo que ir y preguntarle al otro nodo. Pero también tengo que pedirle a ese nodo de la base de datos que no permita que se actualice mientras se completa la lectura; en otras palabras, necesito iniciar una lectura transaccional en varios nodos de la base de datos para garantizar la coherencia. Pero, en general, la gente no realiza lecturas transaccionales, ¿verdad? Porque las lecturas transaccionales son lentas. Requieren bloqueos. Una lectura puede bloquear un sistema entero. Todos los sistemas coherentes requieren cierto nivel de bloqueo para hacer su trabajo.

Como ya hemos comentado, los sistemas distribuidos tienen que esperar que se produzcan fallos.

Consideremos nuestra lectura transaccional en un conjunto de nodos consistentes. Le pido a un nodo remoto que bloquee un registro determinado mientras se inicia la lectura. Completo la lectura y le pido al nodo remoto que libere su bloqueo, pero ahora no puedo comunicarme con él. ¿Qué sucede ahora? Es muy difícil hacer bien los bloqueos incluso en un sistema de un solo proceso, y son significativamente más difíciles de implementar bien en un sistema distribuido.

¿Recuerdas cuando hablamos de transacciones distribuidas en [el Capítulo 5](#)? El motivo principal por el que son un desafío es el problema de garantizar la coherencia entre varios nodos.

Obtener la consistencia multinodo correctamente es tan difícil que yo sugeriría enfáticamente que si la necesitas, no intentes inventarla tú mismo. En su lugar, elige un almacén de datos o servicio de bloqueo que ofrezca estas características. Consulta, por ejemplo, que analizaremos en breve, implementa un almacén de clave/valor fuertemente consistente diseñado para compartir la configuración entre múltiples nodos. Junto con "Los amigos no dejan que sus amigos escriban su propia criptografía" debería ir "Los amigos no dejan que sus amigos escriban su propio almacén de datos distribuido consistente". Si crees que necesitas escribir tu propio almacén de datos CP, lee primero todos los artículos sobre el tema, luego hazte un doctorado y luego espera pasar unos años equivocándote. Mientras tanto, usaré algo que esté disponible comercialmente que lo haga por mí, o más probablemente, me esforzaré mucho para construir sistemas AP eventualmente consistentes en su lugar.

¿Sacrificar la tolerancia de partición?

Podemos elegir dos, ¿no? Tenemos nuestro sistema AP consistente en última instancia. Tenemos nuestro sistema CP consistente, pero difícil de construir y escalar. ¿Por qué no un sistema CA? Bueno, ¿cómo podemos sacrificar la tolerancia a particiones? Si nuestro sistema no tiene tolerancia a particiones, no puede ejecutarse en una red. En otras palabras, debe ser un solo proceso que funcione localmente. Los sistemas CA no existen en sistemas distribuidos.

¿AP o CP?

¿Cuál es el correcto, AP o CP? Bueno, la realidad es que depende. Como personas que construyen el sistema, sabemos que existe una disyuntiva. Sabemos que los sistemas AP escalan más fácilmente y son más simples de construir, y sabemos que un sistema CP requerirá más trabajo debido a los desafíos de respaldar la consistencia distribuida. Pero es posible que no comprendamos el impacto comercial de esta disyuntiva. Para nuestro sistema de inventario, si un registro está desactualizado por cinco minutos, ¿eso está bien? Si la respuesta es sí, un sistema AP podría ser la respuesta. Pero ¿qué sucede con el saldo que se mantiene para un cliente en un banco?

¿Puede estar desactualizado? Sin conocer el contexto en el que se utiliza la operación, no podemos saber qué es lo correcto que debemos hacer. Conocer el teorema CAP solo lo ayuda a comprender que existe esta disyuntiva y qué preguntas hacer.

No es todo o nada

Nuestro sistema en su conjunto no necesita ser AP o CP. Nuestro catálogo podría ser AP, ya que no nos preocupa demasiado tener un registro obsoleto. Pero podríamos decidir que nuestro servicio de inventario debe ser CP, ya que no queremos venderle a un cliente algo que no tenemos y luego tener que disculparnos.

Pero los servicios individuales ni siquiera necesitan ser CP o AP.

Pensemos en nuestro servicio de saldo de puntos, donde almacenamos registros de cuántos puntos de fidelidad han acumulado nuestros clientes. Podríamos decidir que no nos importa si el saldo que mostramos para un cliente está desactualizado, pero que cuando se trata de actualizar un saldo necesitamos que sea consistente para asegurarnos de que los clientes no usen más puntos de los que tienen disponibles. ¿Se trata de un microservicio CP, AP o ambos? En realidad, lo que hemos hecho es trasladar las compensaciones en torno al teorema CAP a las capacidades de servicio individuales.

Otra complejidad es que ni la consistencia ni la disponibilidad son todo o nada. Muchos sistemas nos permiten un equilibrio mucho más matizado. Por ejemplo, con Cassandra puedo hacer diferentes equilibrios para llamadas individuales. Entonces, si necesito una consistencia estricta, puedo realizar una lectura que se bloquee hasta que todas las réplicas hayan respondido confirmando que el valor es consistente, o hasta que haya respondido un quórum específico de réplicas, o incluso hasta que solo un nodo. Obviamente, si bloleo esperando que todas las réplicas respondan y una de ellas no está disponible, estaré bloqueado durante mucho tiempo. Pero si estoy satisfecho con solo un quórum simple de nodos que respondan, puedo aceptar cierta falta de consistencia para ser menos vulnerable a que una sola réplica no esté disponible.

A menudo verás publicaciones sobre personas que superan el teorema CAP. No lo han hecho. Lo que han hecho es crear un sistema en el que algunas capacidades son CP y otras AP. La prueba matemática detrás del teorema CAP se mantiene. A pesar de muchos intentos en la escuela, he aprendido que no se puede superar a las matemáticas.

Y el mundo real

Gran parte de lo que hemos hablado se refiere al mundo electrónico: bits y bytes almacenados en la memoria. Hablamos de coherencia de una manera casi infantil: imaginamos que, dentro del alcance del sistema que hemos creado, podemos detener el mundo y hacer que todo tenga sentido.

Y, sin embargo, gran parte de lo que construimos es solo un reflejo del mundo real, y no podemos controlarlo, ¿verdad?

Repasemos nuestro sistema de inventario. Esto se aplica a los artículos físicos del mundo real. Llevamos un registro en nuestro sistema de cuántos álbumes tenemos. Al principio del día teníamos 100 copias de Give Blood de The Brakes. Vendimos una. Ahora tenemos 99 copias. Fácil, ¿verdad?

¿Qué ocurre si, cuando se está enviando el pedido, alguien tira una copia del álbum al suelo y la pisan y la rompen? ¿Qué ocurre ahora? Nuestros sistemas indican 99, pero hay 98 copias en la estantería.

¿Qué pasaría si hicieramos que nuestro sistema de inventario fuera AP y, ocasionalmente, tuviéramos que comunicarnos con un usuario más tarde y decirle que uno de sus artículos está realmente agotado? ¿Sería eso lo peor del mundo? Sin duda, sería mucho más fácil construirlo, escalarlo y garantizar su funcionamiento correcto.

Debemos reconocer que, por muy coherentes que sean nuestros sistemas, no pueden saber todo lo que sucede, especialmente cuando llevamos registros del mundo real. Esta es una de las principales razones por las que los sistemas de AP terminan siendo la opción correcta en muchas situaciones. Además de la complejidad de crear sistemas de CP, de todos modos no pueden solucionar todos nuestros problemas.

Descubrimiento de servicios

Una vez que tienes más de unos pocos microservicios por ahí, tu atención se centra inevitablemente en saber dónde está todo. Tal vez quieras saber qué se está ejecutando en un entorno determinado para saber qué deberías estar monitoreando. Tal vez sea tan simple como saber dónde está tu servicio de cuentas para que los microservicios que lo usan sepan dónde encontrarlo. O tal vez solo quieras facilitarles a los desarrolladores de tu organización saber qué API están disponibles para que no reinventen la rueda. En términos generales, todos estos casos de uso caen bajo el estandarte del descubrimiento de servicios. Y como siempre con los microservicios, tenemos bastantes opciones diferentes a nuestra disposición para lidiar con ello.

Todas las soluciones que veremos abordan las cosas en dos partes. En primer lugar, proporcionan algún mecanismo para que una instancia se registre y diga "¡Estoy aquí!". En segundo lugar, proporcionan una forma de encontrar el servicio una vez que está registrado. Sin embargo, el descubrimiento de servicios se vuelve más complicado cuando consideramos un entorno en el que estamos destruyendo e implementando constantemente nuevas instancias de servicios. Lo ideal sería que cualquier solución que elijamos pudiera hacer frente a esto.

Veamos algunas de las soluciones más comunes para la prestación de servicios y consideremos nuestras opciones.

Es bueno empezar de forma sencilla. El DNS nos permite asociar un nombre con la dirección IP de una o más máquinas. Podríamos decidir, por ejemplo, que nuestro servicio de cuentas siempre se encuentre en accounts.musiccorp.com. Entonces, tendríamos ese punto de entrada a la dirección IP del host que ejecuta ese servicio, o tal vez haríamos que se resuelva a un balanceador de carga que distribuye la carga entre varias instancias. Esto significa que tendríamos que manejar la actualización de estas entradas como parte de la implementación de nuestro servicio.

Al trabajar con instancias de un servicio en diferentes entornos, he visto que una plantilla de dominio basada en convenciones funciona bien. Por ejemplo, podríamos tener una plantilla definida como <servicename>-<environment>.musiccorp.com, lo que nos daría entradas como accounts-uat.musiccorp.com o accounts-dev.musiccorp.com.

Una forma más avanzada de manejar diferentes entornos es tener diferentes servidores de nombres de dominio para diferentes entornos. Por lo tanto, podría suponer que accounts.musiccorp.com es donde siempre encuentro el servicio de cuentas, pero podría resolverse en diferentes hosts según dónde haga la búsqueda. Si ya tienes tus entornos en diferentes segmentos de red y te sientes cómodo administrando tus propios servidores y entradas DNS, esta podría ser una solución bastante buena, pero es mucho trabajo si no obtienes otros beneficios de esta configuración.

El DNS tiene una serie de ventajas, la principal de las cuales es que es un estándar tan bien entendido y utilizado que prácticamente cualquier conjunto de tecnologías lo soportará. Desafortunadamente, si bien existen varios servicios para administrar el DNS dentro de una organización, pocos de ellos parecen diseñados para un entorno en el que estamos tratando con hosts altamente descartables, lo que hace que la actualización de las entradas DNS sea algo complicada. El servicio Route53 de Amazon hace un buen trabajo en esto, pero aún no he visto una opción alojada en el servidor que sea tan buena, aunque (como analizaremos en breve) Consul puede ayudarnos en este aspecto. Además de los problemas para actualizar las entradas DNS, la especificación DNS en sí misma puede causarnos algunos problemas.

Las entradas DNS para nombres de dominio tienen un tiempo de vida (TTL). Este es el tiempo que un cliente puede considerar que la entrada es nueva. Cuando queremos cambiar el host al que hace referencia el nombre de dominio, actualizamos esa entrada, pero tenemos que asumir que los clientes mantendrán la IP antigua al menos durante el tiempo que indique el TTL. Las entradas DNS se pueden almacenar en caché en varios lugares (incluso la JVM almacenará en caché las entradas DNS a menos que le indiques que no lo haga) y cuantos más lugares las almacenen en caché, más obsoletas pueden quedar.

Una forma de solucionar este problema es hacer que la entrada del nombre de dominio de su servicio apunte a un balanceador de carga, que a su vez apunta a las instancias de su servicio, como se muestra en [la Figura 11-9](#). Cuando implementa una nueva instancia, puede quitar la antigua de la entrada del balanceador de carga y agregar la nueva. Algunas personas utilizan la función de rotación de DNS, donde las entradas de DNS se refieren a un grupo de máquinas. Esta técnica es extremadamente problemática, ya que el cliente está oculto del host subyacente y, por lo tanto, no puede acceder fácilmente a las instancias de su servicio.

dejar de enrutar el tráfico a uno de los hosts si éste deja de funcionar.

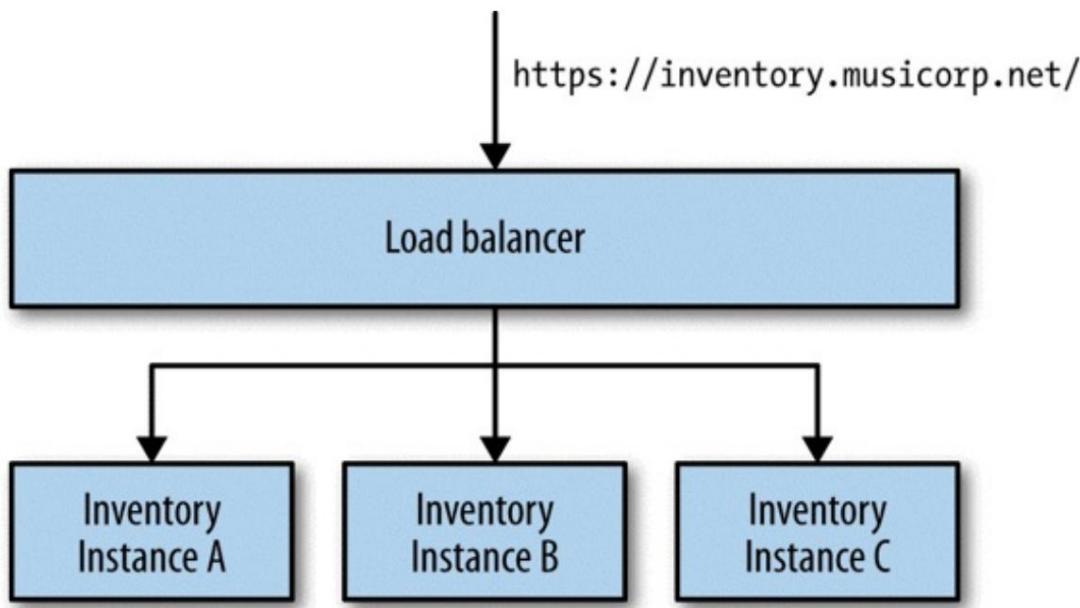


Figura 11-9. Uso de DNS para resolver un balanceador de carga y evitar entradas DNS obsoletas

Como se mencionó, el DNS es un sistema bien conocido y ampliamente compatible, pero tiene una o dos desventajas. Sugeriría investigar si es una buena opción para usted antes de elegir algo más complejo. Para una situación en la que solo tiene nodos individuales, probablemente sea adecuado que el DNS haga referencia directamente a los hosts. Pero para aquellas situaciones en las que necesita más de una instancia de un host, haga que las entradas del DNS se resuelvan en平衡adores de carga que puedan manejar la activación y desactivación de hosts individuales según corresponda.

Registros de servicios dinámicos

Las desventajas del DNS como forma de encontrar nodos en un entorno altamente dinámico han dado lugar a una serie de sistemas alternativos, la mayoría de los cuales implican que el servicio se registre en algún registro central, que a su vez ofrece la posibilidad de buscar estos servicios más adelante. A menudo, estos sistemas hacen más que simplemente proporcionar registro y descubrimiento de servicios, lo que puede ser bueno o no. Este es un campo concurrido, por lo que solo veremos algunas opciones para darle una idea de lo que está disponible.

Guardián del zoológico

Guardián del zoológico Se desarrolló originalmente como parte del proyecto Hadoop. Se utiliza para una variedad de casos de uso casi desconcertante, incluida la gestión de configuración, la sincronización de datos entre servicios, la elección de líderes, las colas de mensajes y (de manera útil para nosotros) como servicio de nombres.

Al igual que muchos tipos de sistemas similares, Zookeeper depende de la ejecución de una serie de nodos en un clúster para brindar diversas garantías. Esto significa que debe esperar ejecutar al menos tres nodos de Zookeeper. La mayor parte de la inteligencia de Zookeeper se centra en garantizar que los datos se repliquen de forma segura entre estos nodos y que las cosas se mantengan constantes cuando los nodos fallan.

En esencia, Zookeeper proporciona un espacio de nombres jerárquico para almacenar información. Los clientes pueden insertar nuevos nodos en esta jerarquía, cambiarlos o consultarlos. Además, pueden agregar relojes a los nodos para que se les informe cuando cambien. Esto significa que podríamos almacenar la información sobre dónde se encuentran nuestros servicios en esta estructura y, como cliente, recibir información cuando cambien. Zookeeper se usa a menudo como un almacén de configuración general, por lo que también podría almacenar la configuración específica del servicio en él, lo que le permite realizar tareas como cambiar dinámicamente los niveles de registro o desactivar las características de un sistema en ejecución. Personalmente, tiendo a evitar el uso de sistemas como Zookeeper como fuente de configuración, ya que creo que puede dificultar el razonamiento sobre el comportamiento de un servicio determinado.

Zookeeper es bastante genérico en lo que ofrece, por lo que se utiliza para tantos casos de uso. Puedes pensar en él como un árbol replicado de información del que puedes recibir alertas cuando cambia. Esto significa que normalmente crearás cosas sobre él para que se adapten a tu caso de uso particular. Afortunadamente, existen bibliotecas de cliente para la mayoría de los lenguajes que existen.

En el gran esquema de las cosas, Zookeeper podría considerarse obsoleto a estas alturas y no nos proporciona tanta funcionalidad lista para usar que ayude con el descubrimiento de servicios en comparación con algunas de las alternativas más nuevas. Dicho esto, sin duda está probado y se usa ampliamente. Los algoritmos subyacentes que implementa Zookeeper son bastante difíciles de hacer bien. Conozco a un proveedor de bases de datos, por ejemplo, que usaba Zookeeper solo para la elección de líder con el fin de garantizar que un nodo principal se promoviera correctamente durante las condiciones de falla. El cliente sintió que Zookeeper era demasiado pesado y pasó mucho tiempo solucionando errores en su propia implementación del algoritmo PAXOS para reemplazar lo que hacía Zookeeper. La gente suele decir que no deberías escribir tus propias bibliotecas de criptografía. Yo ampliaría eso diciendo que tampoco deberías escribir tus propios sistemas de coordinación distribuidos. Hay mucho que decir a favor de usar cosas existentes que simplemente funcionan.

Cónsul

Al igual que el guardián del zoológico, [el cónsul](#) Admite tanto la gestión de configuración como el descubrimiento de servicios. Pero va más allá que Zookeeper al proporcionar más soporte para estos casos de uso clave. Por ejemplo, expone una interfaz HTTP para el descubrimiento de servicios, y una de las características más destacadas de Consul es que realmente proporciona un servidor DNS listo para usar; específicamente, puede servir registros SRV, que le brindan una IP y un puerto para un nombre determinado. Esto significa que si parte de su sistema ya usa DNS y puede admitir registros SRV, puede simplemente instalar Consul y comenzar a usarlo sin realizar ningún cambio en su sistema existente.

Consul también incorpora otras funciones que pueden resultarle útiles, como la capacidad de realizar comprobaciones de estado de los nodos. Esto significa que Consul podría superponerse a las funciones que ofrecen otras herramientas de monitorización dedicadas, aunque lo más probable es que utilice Consul como fuente de esta información y luego la incorpore a un panel de control o un sistema de alerta más completo. Sin embargo, el diseño altamente tolerante a fallos de Consul y su enfoque en el manejo de sistemas que hacen un uso intensivo de nodos efímeros me hacen preguntarme si puede acabar sustituyendo a sistemas como Nagios y Sensu en algunos casos de uso.

Consul utiliza una interfaz HTTP RESTful para todo, desde registrar un servicio, consultar el almacén de claves y valores o insertar comprobaciones de estado. Esto hace que la integración con diferentes pilas de tecnología sea muy sencilla. Otra de las cosas que realmente me gustan de Consul es que el equipo que está detrás ha dividido la parte de gestión de clústeres subyacente. Serf, sobre la que se asienta Consul, se encarga de la detección de nodos en un clúster, la gestión de fallos y las alertas. A continuación, Consul añade el descubrimiento de servicios y la gestión de la configuración. Esta separación de preocupaciones me resulta atractiva, lo que no debería sorprenderle dados los temas que se tratan en este libro.

Consul es muy nuevo y, dada la complejidad de los algoritmos que utiliza, normalmente no me atrevería a recomendarlo para un trabajo tan importante. Dicho esto, Hashicorp, el equipo que está detrás de él, sin duda tiene una gran trayectoria en la creación de tecnología de código abierto muy útil (en forma de Packer y Vagrant), el proyecto se está desarrollando activamente y he hablado con algunas personas que lo están utilizando con gusto en producción. Teniendo en cuenta todo esto, creo que vale la pena echarle un vistazo.

Eureka

El sistema Eureka de código abierto de Netflix Va en contra de la tendencia de sistemas como Consul y Zookeeper en el sentido de que no intenta ser un almacén de configuración de propósito general. En realidad, está muy orientado a su uso.

Eureka también ofrece capacidades básicas de equilibrio de carga, ya que puede admitir la búsqueda básica de instancias de servicio por turnos. Proporciona un punto final basado en REST para que puedas escribir tus propios clientes o puedes usar su propio cliente Java. El cliente Java ofrece capacidades adicionales, como la comprobación del estado de las instancias. Obviamente, si omites el propio cliente de Eureka y vas directamente al punto final REST, tendrás que arreglártelas solo.

Al permitir que los clientes se ocupen directamente del descubrimiento de servicios, evitamos la necesidad de un proceso independiente. Sin embargo, es necesario que cada cliente implemente el descubrimiento de servicios. Netflix, que estandariza en la JVM, logra esto haciendo que todos los clientes utilicen Eureka. Si se encuentra en un entorno más políglota, esto puede ser un mayor desafío.

Arma tu propio rollo

Un enfoque que he utilizado personalmente y que he visto en otros lugares es el de crear tu propio sistema. En un proyecto estábamos haciendo un uso intensivo de AWS, que ofrece la posibilidad de agregar etiquetas a las instancias. Al iniciar instancias de servicio, aplicaba etiquetas para ayudar a definir qué era la instancia y para qué se utilizaba. Esto permitía asociar algunos metadatos Enriquecidos con un host determinado, por ejemplo:

- servicio = cuentas
- medio ambiente = producción
- versión = 154

Luego, pude usar las API de AWS para consultar todas las instancias asociadas con una cuenta de AWS determinada para encontrar las máquinas que me interesaban. Aquí, AWS se encarga del almacenamiento de los metadatos asociados con cada instancia y nos brinda la capacidad de consultarlos. Luego, creé herramientas de línea de comandos para interactuar con estas instancias y crear paneles para monitorear el estado se vuelve bastante fácil, especialmente si adoptas la idea de que cada instancia de servicio exponga los detalles de la verificación de estado.

La última vez que hice esto, no llegamos tan lejos como para que los servicios utilizaran las API de AWS para encontrar sus dependencias de servicio, pero no hay ninguna razón por la que no pudieras hacerlo. Obviamente, si quieras que los servicios ascendentes reciban alertas cuando cambia la ubicación de un servicio descendente, estás solo.

¡No olvidemos a los humanos!

Los sistemas que hemos analizado hasta ahora facilitan que una instancia de servicio se registre y busque otros servicios con los que necesita comunicarse. Pero, como humanos, a veces también queremos esta información. Sea cual sea el sistema que elija, asegúrese de tener herramientas disponibles que le permitan crear informes y paneles de control sobre estos registros para crear visualizaciones para humanos, no solo para computadoras.

Servicios de documentación

Al descomponer nuestros sistemas en microservicios más detallados, esperamos exponer muchas costuras en forma de API que las personas puedan usar para hacer muchas cosas, ojalá maravillosas. Si realizamos el descubrimiento correctamente, sabremos dónde están las cosas. Pero, ¿cómo sabemos qué hacen esas cosas o cómo usarlas? Una opción es, obviamente, tener documentación sobre las API. Por supuesto, la documentación a menudo puede estar desactualizada. Lo ideal sería asegurarnos de que nuestra documentación esté siempre actualizada con la API del microservicio y facilitar la visualización de esta documentación cuando sepamos dónde se encuentra un punto final del servicio. Dos piezas de tecnología diferentes, Swagger y HAL, intentan hacer que esto sea una realidad, y ambas va

Pavonearse

Swagger te permite describir tu API para generar una interfaz de usuario web muy agradable que te permite ver la documentación e interactuar con la API a través de un navegador web. La capacidad de ejecutar solicitudes es muy útil: puedes definir plantillas POST, por ejemplo, para dejar en claro qué tipo de contenido espera el servidor.

Para hacer todo esto, Swagger necesita que el servicio exponga un archivo sidecar que coincida con el formato de Swagger. Swagger tiene varias bibliotecas para diferentes lenguajes que hacen esto por usted. Por ejemplo, para Java puedes anotar métodos que coincidan con tus llamadas API y el archivo se generará automáticamente.

Me gusta la experiencia de usuario final que ofrece Swagger, pero no contribuye mucho al concepto de exploración incremental que es el núcleo del hipermedio. Aun así, es una forma bastante agradable de exponer documentación sobre sus servicios.

HAL y el navegador HAL

Por sí solo, el [lenguaje de aplicación de hipertexto \(HAL\)](#) es un estándar que describe los estándares para los controles de hipermedia que exponemos. Como cubrimos en [el Capítulo 4](#), los controles de hipermedia son los medios por los cuales permitimos a los clientes explorar progresivamente nuestras API para usar las capacidades de nuestro servicio de una manera menos acoplada que otras técnicas de integración. Si decide adoptar el estándar de hipermedia de HAL, entonces no solo puede hacer uso de una amplia cantidad de bibliotecas de cliente para consumir la API (al momento de escribir esto, la wiki de HAL enumeraba 50 bibliotecas de soporte para varios idiomas diferentes), sino que también puede hacer uso del navegador HAL, que le brinda una forma de explorar la API a través de un navegador web.

Al igual que Swagger, esta interfaz de usuario se puede utilizar no solo para actuar como documentación viva, sino también para ejecutar llamadas contra el propio servicio. Sin embargo, la ejecución de llamadas no es tan sencilla. Mientras que con Swagger puedes definir plantillas para hacer cosas como emitir una solicitud POST, con HAL estás más solo. La otra cara de esto es que el poder inherente de los controles hipermedia te permite explorar de manera mucho más efectiva la API expuesta por el servicio, ya que puedes seguir enlaces con mucha facilidad. ¡Resulta que los navegadores web son bastante buenos en ese tipo de cosas!

A diferencia de Swagger, toda la información necesaria para manejar esta documentación y el entorno de pruebas está integrada en los controles de hipermedia. Esto es un arma de doble filo. Si ya está utilizando controles de hipermedia, no requiere mucho esfuerzo exponer un navegador HAL y hacer que los clientes exploren su API. Sin embargo, si no está utilizando hipermedia, no puede utilizar HAL o tiene que adaptar su API para que utilice hipermedia, lo que probablemente sea un ejercicio que rompa con los consumidores existentes.

El hecho de que HAL también describa un estándar de hipermedia con algunas bibliotecas de cliente de apoyo es una ventaja adicional y sospecho que es una de las principales razones por las que he visto una mayor adopción de HAL como forma de documentar API que Swagger por parte de aquellas personas que ya utilizan controles de hipermedia. Si estás utilizando hipermedia, mi recomendación es que optes por HAL en lugar de Swagger. Pero si estás utilizando hipermedia y no puedes justificar el cambio, definitivamente te sugeriría que pruebes Swagger.

El sistema de autodescripción

Durante la evolución inicial de SOA, surgieron estándares como Universal Description, Discovery, and Integration (UDDI) para ayudar a las personas a comprender qué servicios se estaban ejecutando.

Estos enfoques eran bastante contundentes, lo que dio lugar a técnicas alternativas para intentar dar sentido a nuestros sistemas. Martin Fowler analizó el concepto de [registro humanitario](#), donde un enfoque mucho más ligero es simplemente tener un lugar donde los humanos puedan registrar información sobre los servicios de la organización en algo tan básico como un wiki.

Obtener una imagen de nuestro sistema y de cómo se comporta es importante, especialmente cuando trabajamos a gran escala. Hemos cubierto una serie de técnicas diferentes que nos ayudarán a comprender directamente nuestro sistema. Al rastrear el estado de nuestros servicios posteriores junto con los identificadores de correlación para ayudarnos a ver las cadenas de llamadas, podemos obtener datos reales en términos de cómo se interrelacionan nuestros servicios. Al usar sistemas de descubrimiento de servicios como Consul, podemos ver dónde se están ejecutando nuestros microservicios. HAL nos permite ver qué capacidades se están alojando en un punto final determinado, mientras que nuestras páginas de verificación de estado y los sistemas de monitoreo nos permiten conocer el estado tanto del sistema general como de los servicios individuales.

Toda esta información está disponible de forma programática. Todos estos datos nos permiten hacer que nuestro registro humanitario sea más potente que una simple página wiki que sin duda quedará obsoleta.

En lugar de ello, deberíamos utilizarla para aprovechar y mostrar toda la información que emitirá nuestro sistema. Al crear paneles personalizados, podemos reunir la amplia gama de información disponible para ayudarnos a comprender nuestro ecosistema.

Por supuesto, comience con algo tan simple como una página web estática o una wiki que tal vez recopile un poco de información del sistema en vivo. Pero trate de incorporar cada vez más información con el tiempo. Hacer que esta información esté disponible de inmediato es una herramienta clave para gestionar la complejidad emergente que surgirá de la ejecución de estos sistemas a gran escala.

Resumen

Como enfoque de diseño, los microservicios aún son bastante jóvenes, por lo que, aunque tenemos algunas experiencias notables de las que podemos sacar partido, estoy seguro de que en los próximos años surgirán patrones más útiles para gestionarlos a gran escala. No obstante, espero que este capítulo haya esbozado algunos pasos que puede seguir en su viaje hacia los microservicios a gran escala que le resulten de utilidad.

Además de lo que he tratado aquí, recomiendo el excelente libro de Michael Nygard, *Release It!*. En él, comparte una colección de historias sobre fallas del sistema y algunos patrones para ayudar a lidiar con ellas. Vale la pena leer el libro (de hecho, me atrevería a decir que debería considerarse una lectura esencial para cualquiera que construya sistemas a gran escala).

Hemos cubierto bastante terreno y nos estamos acercando al final. En nuestro próximo y último capítulo, intentaremos resumir todo lo que hemos aprendido en el libro en general.

Capítulo 12. Uniendo todo

En los capítulos anteriores hemos abordado una gran cantidad de temas, desde qué son los microservicios hasta cómo definir sus límites, y desde la tecnología de integración hasta las preocupaciones sobre seguridad y monitoreo. Incluso encontramos tiempo para analizar cómo encaja el rol del arquitecto. Hay mucho que aprender, ya que, si bien los microservicios en sí mismos pueden ser pequeños, la amplitud y el impacto de su arquitectura no lo son. Por eso, aquí intentaré resumir algunos de los puntos clave que se tratan a lo largo del libro.

Principios de los microservicios

En [el capítulo 2](#) analizamos el papel que pueden desempeñar los principios. Son declaraciones sobre cómo se deben hacer las cosas y por qué creemos que se deben hacer de esa manera. Nos ayudan a enmarcar las distintas decisiones que debemos tomar al crear nuestros sistemas. Sin duda, usted debe definir sus propios principios, pero pensé que valía la pena explicar en detalle los que considero que son los principios clave para las arquitecturas de microservicios, que puede ver resumidos en [la Figura 12-1](#). Estos son los principios que nos ayudarán a crear pequeños servicios autónomos que funcionen bien juntos. Ya hemos cubierto todo aquí al menos una vez hasta ahora, por lo que nada debería ser nuevo, pero es valioso resumirlo hasta su esencia central.

Puede optar por adoptar estos principios en su totalidad o quizás modificarlos para que tengan sentido en su propia organización. Pero tenga en cuenta el valor que se obtiene al utilizarlos en combinación: el conjunto debe ser mayor que la suma de las partes. Por lo tanto, si decide dejar de lado uno de ellos, asegúrese de comprender lo que se perderá.

Para cada uno de estos principios, he intentado extraer algunas de las prácticas de apoyo que hemos tratado en el libro. Como dice el dicho, hay más de una forma de despellejar a un gato: es posible que encuentres tus propias prácticas que te ayuden a cumplir con estos principios, pero esto debería servirte para empezar.

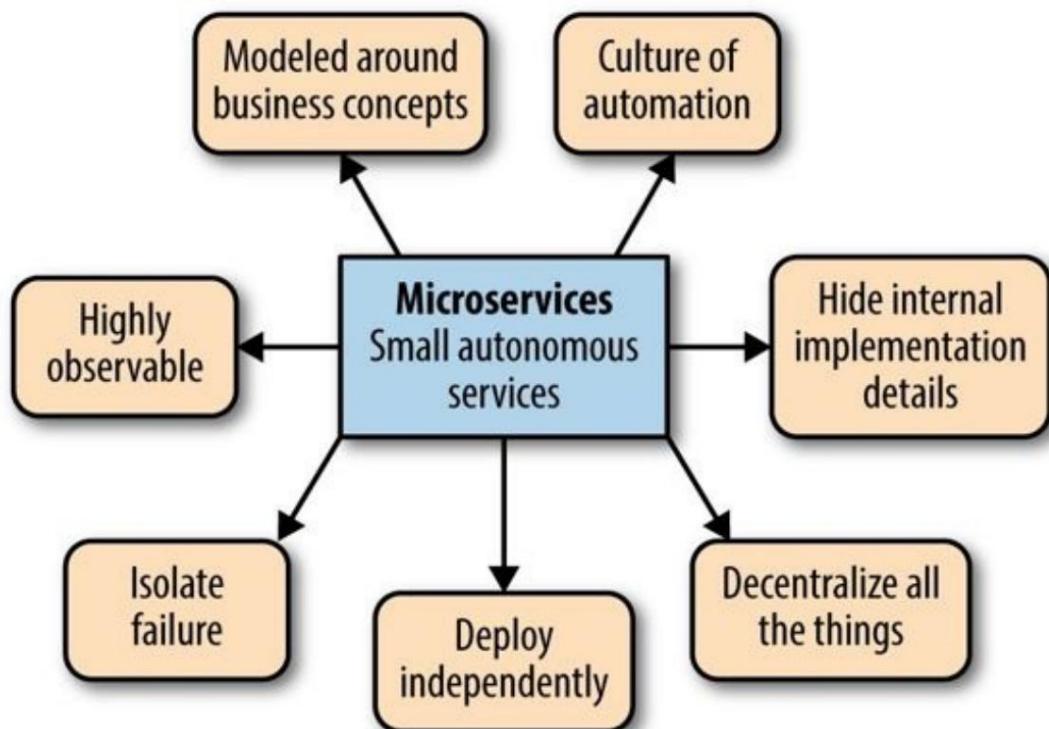


Figura 12-1. Principios de los microservicios

Modelo en torno a conceptos empresariales

La experiencia nos ha demostrado que las interfaces estructuradas en torno a contextos limitados por el negocio son más estables que las estructuradas en torno a conceptos técnicos. Al modelar el dominio en el que opera nuestro sistema, no solo intentamos formar interfaces más estables, sino que también nos aseguramos de poder reflejar mejor los cambios en los procesos de negocio con facilidad. Utilice contextos limitados para definir posibles límites de dominio.

Adoptar una cultura de automatización

Los microservicios agregan mucha complejidad, una parte clave de la cual proviene de la gran cantidad de partes móviles con las que tenemos que lidiar. Adoptar una cultura de automatización es una forma clave de abordar esto, y concentrar esfuerzos en crear las herramientas para respaldar los microservicios puede tener mucho sentido. Las pruebas automatizadas son esenciales, ya que garantizar que nuestros servicios sigan funcionando es un proceso más complejo que con sistemas monolíticos. Tener una llamada de línea de comandos uniforme para implementar de la misma manera en todas partes puede ayudar, y esto puede ser una parte clave de la adopción de la entrega continua para brindarnos una retroalimentación rápida sobre la calidad de producción de cada registro.

Considere la posibilidad de utilizar definiciones de entorno para ayudarle a especificar las diferencias entre un entorno y otro, sin sacrificar la capacidad de utilizar un método de implementación uniforme. Piense en crear imágenes personalizadas para acelerar la implementación y en adoptar la creación de servidores inmutables totalmente automatizados para facilitar el razonamiento sobre sus sistemas.

Ocultar detalles de implementación interna

Para maximizar la capacidad de un servicio de evolucionar independientemente de los demás, es fundamental que ocultemos los detalles de implementación. El modelado de contextos delimitados puede resultar de ayuda, ya que nos ayuda a centrarnos en los modelos que se deben compartir y los que se deben ocultar. Los servicios también deben ocultar sus bases de datos para evitar caer en uno de los tipos de acoplamiento más comunes que pueden aparecer en las arquitecturas orientadas a servicios tradicionales, y utilizar bombas de datos o bombas de datos de eventos para consolidar los datos de varios servicios con fines de generación de informes.

Siempre que sea posible, elija API independientes de la tecnología para tener la libertad de usar diferentes pilas de tecnología. Considere usar REST, que formaliza la separación de los detalles de implementación internos y externos, aunque incluso si usa llamadas a procedimientos remotos (RPC), puede adoptar estas ideas.

Descentralizar todas las cosas

Para maximizar la autonomía que permiten los microservicios, debemos buscar constantemente la posibilidad de delegar la toma de decisiones y el control a los equipos que son los dueños de los servicios. Este proceso comienza con la adopción del autoservicio siempre que sea posible, permitiendo que las personas implementen software a pedido, haciendo que el desarrollo y las pruebas sean lo más fáciles posible y evitando la necesidad de equipos separados para realizar estas actividades.

Asegurarse de que los equipos sean dueños de sus servicios es un paso importante en este camino, haciendo que los equipos sean responsables de los cambios que se realizan, idealmente incluso dejándolos decidir cuándo publicar esos cambios. El uso de código abierto interno garantiza que las personas puedan realizar cambios en los servicios que son propiedad de otros equipos, aunque recuerde que esto requiere trabajo para implementarlo. Alinee a los equipos con la organización para asegurarse de que la ley de Conway funcione para usted y ayude a su equipo a convertirse en expertos en el dominio de los servicios centrados en el negocio que están creando. Cuando se necesite una guía general, intente adoptar un modelo de gobernanza compartida donde las personas de cada equipo comparten colectivamente la responsabilidad de desarrollar la visión técnica del sistema.

Este principio también se puede aplicar a la arquitectura. Evite enfoques como el bus de servicios empresariales o los sistemas de orquestación, que pueden llevar a la centralización de la lógica empresarial y a servicios tontos. En su lugar, prefiera la coreografía a la orquestación y al middleware tonto, con puntos finales inteligentes para garantizar que la lógica y los datos asociados se mantengan dentro de los límites del servicio, lo que ayuda a mantener la cohesión.

Desplegable de forma independiente

Siempre debemos esforzarnos por garantizar que nuestros microservicios puedan implementarse y se implementen por sí solos. Incluso cuando se requieren cambios importantes, debemos buscar la coexistencia de puntos finales con versiones para permitir que nuestros consumidores cambien con el tiempo. Esto nos permite optimizar la velocidad de lanzamiento de nuevas funciones, así como aumentar la autonomía de los equipos que poseen estos microservicios al garantizar que no tengan que organizar constantemente sus implementaciones. Al utilizar la integración basada en RPC, evite la generación de stubs de cliente/servidor estrechamente vinculados, como el que promueve Java RMI.

Al adoptar un modelo de un servicio por host , se reducen los efectos secundarios que podrían provocar que la implementación de un servicio afecte a otro servicio no relacionado. Considere la posibilidad de utilizar técnicas de lanzamiento azul/verde o canary para separar la implementación del lanzamiento, lo que reduce el riesgo de que un lanzamiento salga mal. Utilice contratos impulsados por el consumidor para detectar cambios importantes antes de que se produzcan.

Recuerde que debería ser la norma, no la excepción, que pueda realizar un cambio en un solo servicio y lanzarlo a producción, sin tener que implementar ningún otro servicio al mismo tiempo. Sus consumidores deben decidir cuándo actualizarse por sí mismos, y usted debe adaptarse a esto.

Aislamiento de fallas

Una arquitectura de microservicios puede ser más resistente que un sistema monolítico, pero solo si comprendemos y planificamos las fallas en una parte de nuestro sistema. Si no tenemos en cuenta el hecho de que una llamada descendente puede fallar y fallará, nuestros sistemas podrían sufrir una falla catastrófica en cascada y podríamos encontrarnos con un sistema mucho más frágil que antes.

Al utilizar llamadas de red, no trate las llamadas remotas como llamadas locales, ya que esto ocultará diferentes tipos de modos de falla. Por lo tanto, asegúrese de que, si está utilizando bibliotecas de cliente, la abstracción de la llamada remota no sea demasiado extensa.

Si tenemos en mente los principios de la antifragilidad y esperamos que el fracaso ocurra en cualquier lugar y en todas partes, vamos por buen camino. Asegúrese de que los tiempos de espera estén configurados de manera adecuada. Comprenda cuándo y cómo utilizar mamparos y disyuntores para limitar las consecuencias de un componente defectuoso. Comprenda cuál será el impacto para el cliente si solo una parte del sistema funciona mal. Conozca cuáles podrían ser las implicaciones de una partición de red y si sacrificar la disponibilidad o la consistencia en una situación determinada es la decisión correcta.

Altamente observable

No podemos depender de la observación del comportamiento de una única instancia de servicio o del estado de una única máquina para ver si el sistema está funcionando correctamente. En cambio, necesitamos una visión conjunta de lo que está sucediendo. Utilice la supervisión semántica para ver si su sistema se está comportando correctamente, inyectando transacciones sintéticas en su sistema para simular el comportamiento de un usuario real. Agregue sus registros y sus estadísticas, de modo que cuando vea un problema pueda profundizar hasta la fuente. Y cuando se trate de reproducir problemas desagradables o simplemente ver cómo interactúa su sistema en producción, utilice identificadores de correlación para poder rastrear las llamadas a través del sistema.

¿Cuándo no deberías utilizar microservicios?

Me hacen esta pregunta con mucha frecuencia. Mi primer consejo sería que cuanto menos conozcas un dominio, más difícil te resultará encontrar contextos delimitados adecuados para tus servicios. Como hemos comentado anteriormente, si no estableces los límites de los servicios correctamente, puedes tener que hacer muchos cambios en la colaboración entre servicios, lo que resulta una operación costosa. Por lo tanto, si te encuentras ante un sistema monolítico para el que no entiendes el dominio, dedica algo de tiempo a aprender qué hace el sistema primero y luego intenta identificar límites de módulos claros antes de dividir los servicios.

El desarrollo de un nuevo dominio también es bastante complicado. No se trata solo de que el dominio probablemente sea nuevo, sino que es mucho más fácil fragmentar algo que ya tienes que algo que no tienes. Por lo tanto, nuevamente, considera comenzar de manera monolítica primero y dividir las cosas cuando estés es-

Muchos de los desafíos que enfrentará con los microservicios empeoran con la escala. Si hace las cosas en forma manual, puede estar satisfecho con 1 o 2 servicios, pero ¿5 o 10?

Seguir con las antiguas prácticas de supervisión en las que solo se observan estadísticas como la CPU y la memoria también puede funcionar bien para algunos servicios, pero cuanto más colaboración entre servicios se haga, más complicado será. Te encontrarás con estos puntos críticos a medida que agregues más servicios, y espero que los consejos de este libro te ayuden a ver algunos de estos problemas que se avecinan y te den algunos consejos concretos sobre cómo lidiar con ellos. Hablé antes sobre el tiempo que tardaron REA y Gilt en desarrollar las herramientas y prácticas para administrar bien los microservicios, antes de poder usarlos en grandes cantidades. Estas historias solo refuerzan para mí la importancia de comenzar gradualmente para que comprendas el apetito y la capacidad de tu organización para cambiar, lo que te ayudará a adoptar los microservicios de manera adecuada.

Palabras de despedida

Las arquitecturas de microservicios le brindan más opciones y más decisiones para tomar. Tomar decisiones en este mundo es una actividad mucho más común que en sistemas monolíticos más simples. No se pueden tomar todas las decisiones correctas, se lo garantizo. Por lo tanto, sabiendo que vamos a equivocarnos en algunas cosas, ¿cuáles son nuestras opciones? Bueno, yo sugeriría encontrar formas de que cada decisión tenga un alcance pequeño; de esa manera, si se equivoca, solo afectará a una pequeña parte de su sistema. Aprenda a adoptar el concepto de arquitectura evolutiva, donde su sistema se dobla, se flexiona y cambia con el tiempo a medida que aprende cosas nuevas. No piense en reescrituras radicales, sino en una serie de cambios realizados en su sistema a lo largo del tiempo para mantenerlo flexible.

Espero que a esta altura haya compartido contigo suficiente información y experiencias para ayudarte a decidir si los microservicios son para ti. Si es así, espero que lo consideres un viaje, no un destino. Ve de forma gradual. Desarma tu sistema pieza por pieza, aprendiendo sobre la marcha.

Y acostúmbrate a ello: en muchos sentidos, la disciplina para cambiar y hacer evolucionar continuamente nuestros sistemas es una lección mucho más importante que cualquier otra que haya compartido con usted a través de este libro. El cambio es inevitable. Acéptelo.

Índice

A

Pruebas de aceptación, [tipos de pruebas](#)

acceso por referencia, [Acceso por Referencia](#)

rendición de cuentas, [personas](#)

Adaptabilidad, [Resumen](#)

Proyecto Aegisthus, [bomba de datos de respaldo](#)

registros agregados, [registros, registros y más registros...](#)

sistemas antirrágiles, [microservicios, organización antirrágil-aislamiento](#)

mamparos, [mamparos](#)

disyuntores, [disyuntores](#)

Ejemplos de, [La Organización Antirrágil](#)

mayor uso de [microservicios](#)

aislamiento, [aislamiento](#)

deslastre de carga, [mamparos](#)

tiempos de espera, [tiempos de espera](#)

Sistema AP

definición del término, [sacrificar la consistencia](#)

vs. sistema CP, [¿AP o CP?](#)

Autenticación basada en claves API, [claves API, todo es cuestión de claves](#)

contenedores de aplicaciones, [contenedores de aplicaciones](#)

arquitectos (ver arquitectos de sistemas)

principios arquitectónicos

desarrollo de, [Principios](#)

Los 12 factores y [principios](#) de Heroku

Principios clave de los microservicios: [cómo unirlo todo](#)

Ejemplo del mundo real, [Un ejemplo del mundo real](#)

seguridad arquitectónica, [seguridad arquitectónica](#), medidas de seguridad arquitectónica

artefactos

imágenes, [imágenes como artefactos](#)

sistema operativo, [Artefactos del sistema operativo](#)

específico de la plataforma, [Artefactos específicos de la plataforma](#)

colaboración asincrónica

complejidades de, [complejidades de las arquitecturas asincrónicas](#)

Implementación, [Implementación de colaboración asincrónica basada en eventos](#)

vs. sincrónico, [sincrónico versus asincrónico](#)

Especificaciones ATOM, [opciones tecnológicas](#)

autenticación/autorización, [Autenticación y Autorización-El Diputado](#)

Problema

Definición de términos, [Autenticación y Autorización](#)

Autorización de grano fino, [Autorización de grano fino](#)

servicio a servicio, [autenticación y autorización de servicio a servicio](#)

Inicio de sesión único (SSO), [Implementaciones comunes de inicio de sesión único](#)

puerta de enlace de inicio de sesión único, [puerta de enlace de inicio de sesión único](#)

Terminología, [Implementaciones comunes de inicio de sesión único](#)

automatización

Beneficios para la implementación, [automatización](#)

Estudios de caso sobre, [Dos estudios de caso sobre el poder de la automatización](#)

autonomía

microservicios y, [Autónomos](#)

El papel del arquitecto de sistemas en, [Resumen](#)

escalamiento automático, [Escalamiento automático](#)

disponibilidad

en el teorema CAP, [Teorema CAP](#)

Principio clave de microservicios: [¿Cuánto es demasiado?](#)

sacrificando, [sacrificando disponibilidad](#)

B

backends para frontends (BFFs), [Backends para Frontends](#)

bombas de datos de respaldo, [bomba de datos de respaldo](#)

copias de seguridad, cifrado de, [cifrar copias de seguridad](#)

Implementación azul/verde: [separación de la implementación del lanzamiento](#)

cuellos de botella, [cuellos de botella en la entrega](#)

contextos delimitados

concepto de, [El contexto delimitado](#)

módulos y servicios, [Módulos y Servicios](#)

Anidado, [Tortugas hasta el fondo](#)

descomposición prematura, [descomposición prematura](#)

modelos compartidos vs. modelos ocultos, [modelos compartidos y ocultos](#)

Diseño de sistemas y [contextos delimitados y estructuras de equipo](#)

Guardafrenos, **Seguridad en la Horneada**

cambios importantes

Evitar, **Evitar cambios disruptivos**

aplazarlo, **aplazarlo tanto como sea posible**

Detección temprana de **cambios disruptivos**

Pruebas frágiles, **pruebas escamosas y frágiles**

fragilidad, **fragilidad**

Construir tuberías, **Construir tuberías y entrega continua**

mamparos, **mamparos**

Lanzamiento de servicios agrupados y las inevitables excepciones

Capacidades empresariales, **Capacidades empresariales**

Conceptos de negocios, **Comunicación en términos de conceptos de negocios**

Pruebas orientadas al negocio, **tipos de pruebas**

do

almacenamiento en caché

Beneficios del **almacenamiento en caché**

Fallos de caché, **ocultando el origen**

Envenenamiento de caché : una historia con moraleja

Almacenamiento en caché del lado del cliente, **del lado del cliente, del proxy y del lado del servidor**

para escrituras, **almacenamiento en caché para escrituras**

en HTTP, **almacenamiento en caché en HTTP**

Proxy, **almacenamiento en caché del lado del cliente, proxy y del lado del servidor**

Almacenamiento en caché del lado del servidor, **del lado del cliente, del proxy y del lado del servidor**

suelta de canarios, suelta de canarios

Teorema CAP, Teorema CAP y el mundo real

Sistemas AP/CP: no es todo o nada

aplicación de, Y el Mundo Real

Fundamentos del teorema CAP

sacrificar la disponibilidad, sacrificar la disponibilidad

sacrificando la consistencia, sacrificando la consistencia

sacrificar la tolerancia de partición, ¿Sacrificar la tolerancia de partición?

Fallos en cascada, La Cascada

Gestión de certificados, Certificados de cliente

Gorila del Caos, la organización antirrágil

Chaos Monkey, la organización antirrágil

Arquitectura coreografiada, Orquestación versus Coreografía

Disyuntores, Plantilla de servicio a medida, Disyuntores

Mixin de circuit_breaker para Ruby, Bulkheads

responsabilidad-de-clase-colaboración (CRC), costo del cambio

certificados de cliente, certificados de cliente

bibliotecas de cliente, bibliotecas de cliente

almacenamiento en caché del lado del cliente, almacenamiento en caché del lado del cliente, proxy y del lado del servidor

Reutilización de código, DRY y los peligros de la reutilización de código en un mundo de microservicios

arquitecto codificador, zonificación

Cohesión, Pequeña y Enfocada en Hacer una Cosa Bien, Alta Cohesión

Pirámide de pruebas de Cohn, [alcance de las pruebas](#)

colaboración, [Resumen](#)

Basado en eventos, [sincrónico versus asincrónico](#)

Solicitud/respuesta, [sincrónica versus asincrónica](#)

Segregación de responsabilidades de consulta y comando (CQRS), [CQRS](#)

confirmaciones, dos fases, [transacciones distribuidas](#)

comunicación

Adaptación a las vías de comunicación, [Adaptación a las vías de comunicación](#)

Protocolos para (SOAP), [¿Qué pasa con la arquitectura orientada a servicios?](#)

Sincrónico vs. asincrónico, [Sincrónico versus asincrónico](#)

transacciones de compensación, [abortar toda la operación](#)

Componibilidad, [Componibilidad](#)

Desviación de configuración, [servidores inmutables](#)

configuración, servicio, [configuración del servicio](#)

Problema del diputado confundido, [El problema del diputado](#)

consistencia

en el teorema CAP, [Teorema CAP](#)

sacrificando, [sacrificando la consistencia](#)

restricciones, [Restricciones](#)

Cónsul, [Cónsul](#)

Contratos impulsados por el consumidor (CDC), [pruebas impulsadas por el consumidor al rescate.](#)

Acerca de Conversaciones

Red de distribución de contenido (CDN), [almacenamiento en caché del lado del cliente, proxy y del lado del servidor](#)

Sistemas de gestión de contenidos (CMS), **Ejemplo: CMS como servicio**

Entrega continua (CD), **microservicios, canalizaciones de compilación y entrega continua**

Entrega

integración continua (IC)

Conceptos básicos: **una breve introducción a la integración continua**

Lista de verificación para: **¿Realmente lo estás haciendo?**

Mapeo a microservicios, **Mapeo de la integración continua a Microservicios**

Ley de Conway

evidencia de, **evidencia**

Al revés, **la Ley de Conway al revés**

Declaración de **la Ley de Conway y el Diseño de Sistemas**

resumen de, **Resumen**

Proceso de coordinación, **Transacciones Distribuidas**

Equipo central, **Rol de los Custodios**

CoreOS, **Docker**

identificadores de correlación, **identificadores de correlación**

¿Sistema CP, AP o CP?

requisitos multifuncionales (CFR), **pruebas multifuncionales, ¿cuánto es demasiado?**

custodios, papel de los custodios

imágenes personalizadas, **Imágenes personalizadas**

Gestión de las relaciones con los clientes (CRM), **ejemplo: el sistema CRM multifunción**

clientes, interactuando con

Inscripción de nuevos clientes, [Interacción con clientes](#), [Orquestación](#)

[Versus Coreografía](#)

bases de datos compartidas, [La base de datos compartida](#)

D

datos

Inserción por lotes de, [Recuperación de datos mediante llamadas de servicio](#)

Durabilidad de, [¿Cuánto es demasiado?](#)

cifrado de copias de seguridad, [cifrar copias de seguridad](#)

Recuperación de datos mediante llamadas de servicio, [Recuperación de datos mediante llamadas de servicio](#)

Asegurar en reposo, [Asegurar datos en reposo](#)

(ver también seguridad)

compartido, [Ejemplo: Datos compartidos](#)

estática compartida, [Ejemplo: Datos estáticos compartidos](#)

cifrado de datos, [vaya con lo conocido](#)

bombas de datos

copia de seguridad, [bomba de datos de copia de seguridad](#)

Recuperación de datos mediante [bombas de datos](#)

evento, [bomba de datos de eventos](#)

Uso en serie de [Destinos Alternativos](#)

Descomposición de [la base de datos](#), [Base de datos: comprensión de las causas fundamentales](#)

Romper relaciones de clave externa, [Ejemplo: Romper relaciones de clave externa](#)

[Relaciones](#)

Enfoque incremental del costo del cambio

Visión general de, [Resumen](#)

Refactorización de bases de datos, [preparación de la ruptura](#)

Selección de puntos de separación: [Cómo afrontar el problema](#)

Selección del momento de separación: [comprensión de las causas fundamentales](#)

Datos compartidos, [Ejemplo: Datos compartidos](#)

Datos estáticos compartidos, [Ejemplo: Datos estáticos compartidos](#)

Tablas compartidas, [Ejemplo: Tablas compartidas](#)

Límites transaccionales, [Límites transaccionales](#)

Integración de bases de datos, [La base de datos compartida](#)

escalamiento de base de datos

Segregación de responsabilidades de consulta y comando (CQRS), [CQRS](#)

para lecturas, [Escalado para lecturas](#)

para escrituras, [Escalado para escrituras](#)

Disponibilidad del servicio versus durabilidad de los datos, [Disponibilidad del servicio versus durabilidad de los datos](#)

Infraestructura compartida, [Infraestructura de base de datos compartida](#)

Directrices para la toma de decisiones: [un enfoque basado en principios: un ejemplo del mundo real](#)

Enfoque personalizado [que combina principios y prácticas](#)

prácticas para, [Prácticas](#)

principios para, [Principios](#)

Ejemplo del mundo real, [Un ejemplo del mundo real](#)

objetivos estratégicos, [objetivos estratégicos](#)

técnicas de descomposición

bases de datos (ver descomposición de bases de datos)

Identificación/empaquetado de contextos, [Breaking Apart MusicCorp](#)

módulos, [Módulos](#)

Concepto de costura, [todo es cuestión de costuras](#)

Selección de puntos de separación, [Las razones para dividir el monolito](#)

Selección del momento de separación: [comprensión de las causas fundamentales](#)

bibliotecas compartidas, [Bibliotecas compartidas](#)

disociación, [autonomía, orquestación versus coreografía](#)

funcionalidad degradante, [funcionalidad degradante](#)

cuellos de botella en la entrega, [cuellos de botella en la entrega](#)

despliegue

artefactos, imágenes como, [imágenes como artefactos](#)

Artefactos, sistema operativo, [Artefactos del sistema operativo](#)

Artefactos específicos de la plataforma [Artefactos específicos de la plataforma](#)

automatización, [automatización](#)

Implementación azul/verde: [separación de la implementación del lanzamiento](#)

Construir pipeline, [Construir pipelines y entrega continua](#)

Lanzamiento de servicios agrupados [y las inevitables excepciones](#)

Fundamentos de la integración continua, [una breve introducción a la integración continua](#)

Integración

Lista de verificación de integración continua: [¿realmente lo estás haciendo?](#)

Integración continua en microservicios, [Mapeo Continuo](#)

Integración con microservicios

imágenes personalizadas, [Imágenes personalizadas](#)

definición de entorno, Definición de entorno

Entornos a tener en cuenta, Entornos

servidores inmutables, servidores inmutables

Interfaces, una interfaz de implementación

microservicios vs. sistemas monolíticos, facilidad de implementación, implementación

Visión general de, Resumen

Separación del lanzamiento, Separación de la implementación del lanzamiento

configuración del servicio, configuración del servicio

Enfoque de virtualización, de lo físico a lo virtual

Virtualización, hipervisores, virtualización tradicional

virtualización, tradicional, Virtualización tradicional

Virtualización, tipo 2, Virtualización tradicional

Problema del diputado, El problema del diputado

Principios de diseño, Principios, Uniéndolo todo: Altamente observable

(ver también principios arquitectónicos)

Prácticas de diseño y entrega

desarrollo de, Prácticas

Ejemplo del mundo real, Un ejemplo del mundo real

servicio de directorio, Implementaciones comunes de inicio de sesión único

DiRT (Prueba de recuperación ante desastres), la organización antifrágil

sistemas distribuidos

falacias de, Las llamadas locales no son como las llamadas remotas, El fracaso es

En todos lados

Promesas clave de la Componibilidad

transacciones distribuidas, [Transacciones Distribuidas](#)

Servicio DNS, [DNS](#)

Docker, [Docker](#)

documentación

HAL (lenguaje de aplicación de hipertexto), [HAL y el navegador HAL](#)

Importancia de [los servicios de documentación](#)

Sistemas autodescriptivos, [HAL y el navegador HAL](#)

Fanfarronería, [fanfarronería](#)

Diseño basado en dominios, [microservicios](#)

Dropwizard, [plantilla de servicio personalizada](#)

DRY (Don't Repeat Yourself), [DRY y los peligros de la reutilización de código en un Mundo de microservicios](#)

maniquíes, [burlándose o tropezando](#)

Durabilidad, [¿cuánto es demasiado?](#)

registros de servicios dinámicos

Beneficios de [los registros de servicios dinámicos](#)

Cónsul, [Cónsul](#)

Eureka, [Eureka](#)

Lanzamiento, [Rolling Your Own](#)

Guardián del zoológico, [guardián del zoológico](#)

mi

empatía, [Resumen](#)

cifrado, [vaya con lo conocido](#)

pruebas de extremo a extremo

Usos apropiados para, **Viajes de prueba, no historias, ¿debería entonces utilizar pruebas de extremo a extremo?**

Pirámide de pruebas de Cohn, **alcance de las pruebas**

creación de, **¿Quién escribe estas pruebas?**

Desventajas de **las pruebas de fragilidad y descamación**

Ciclo de retroalimentación, **La gran acumulación**

Implementación de **esas complicadas pruebas de extremo a extremo**

metaversión, **La Metaversión**

Alcance de **las pruebas de extremo a extremo**

tiempo de, **¿Cuánto tiempo?**

puntos finales

coexistencia de diferentes, **coexistencia de diferentes puntos finales**

Agregación del lado del servidor, **backends para frontends**

entornos

Definición durante la implementación, **Definición del entorno**

Consideraciones de implementación, **Entornos**

gestión, **entornos**

Módulos Erlang, **Módulos**

Eureka, **Eureka**

Bombas de datos de eventos, **Bomba de datos de eventos**

abastecimiento de eventos, **CQRS**

Colaboración basada en eventos, **sincrónica versus asincrónica**

Consistencia eventual, [Inténtalo de nuevo más tarde, Sacrificando la consistencia](#)

arquitectos evolutivos (ver arquitectos de sistemas)

Manejo de excepciones, [Manejo de excepciones](#)

Ejemplares, [Ejemplares](#)

Pruebas exploratorias, [tipos de pruebas](#)

F

Los robots de fracaso, [La Organización Antifrágil](#)

fallas

en cascada, [La Cascada](#)

(ver también seguimiento)

Lidiando con el fracaso está en todas partes

falsificaciones, [burlas o tonterías](#)

equipos basados en funciones, [equipos de funciones](#)

cortafuegos, [cortafuegos](#)

Pruebas de escamas, [pruebas de escamas y frágiles](#)

Relaciones de clave externa, ruptura, [Ejemplo: ruptura de clave externa](#)

Relaciones

Herramienta de gestión de paquetes FPM, [Artefactos del sistema operativo](#)

funcionalidad, degradando, [Degradación de la funcionalidad](#)

GRAMO

Días de juego, [La Organización Antifrágil](#)

Guardianes, [papel de los custodios](#)

gobernancia

Concepto de Gobernanza y Liderazgo desde el Centro

El papel del arquitecto de sistemas en, Resumen

Granularidad, ¿Qué pasa con la arquitectura orientada a servicios?

Graphite, seguimiento de métricas en múltiples servicios

yo

Sistemas habitables, una visión evolutiva para el arquitecto

HAL (lenguaje de aplicación de hipertexto), HAL y el navegador HAL

Código de mensajería basado en hash (HMAC), HMAC sobre HTTP

Principio HATEOS: hipermedia como motor del estado de la aplicación

heterogeneidad

Beneficios de la heterogeneidad tecnológica

bibliotecas compartidas y, Bibliotecas compartidas

Arquitectura hexagonal, microservicios

Modelos ocultos, modelos compartidos y ocultos

alta cohesión, Alta Cohesión

HMAC (código de mensajería basado en hash), HMAC sobre HTTP

HTTP (Protocolo de transferencia de hipertexto)

almacenamiento en caché, almacenamiento en caché en HTTP

Principio HATEOS: hipermedia como motor del estado de la aplicación

Beneficios de HTTP sobre REST, REST y HTTP

Desventajas de HTTP sobre REST, Desventajas de REST sobre HTTP

Terminación de HTTP, equilibrio de carga

Autenticación básica HTTP(S), Autenticación básica HTTP(S)

Registro humanitario, HAL y el navegador HAL

hipermedia, Hipermedia como motor del estado de la aplicación

Hipervisores, Virtualización tradicional

Biblioteca Hystrix, plantilla de servicio a medida, mamparos

I

operaciones idempotentes, idempotencia

Proveedor de identidad, Implementaciones comunes de inicio de sesión único

imágenes

como artefactos, imágenes como artefactos

personalizado, Imágenes personalizadas

servidores inmutables, servidores inmutables

automatización de infraestructura, microservicios

integración

acceso por referencia, Acceso por Referencia

Colaboración asincrónica basada en eventos, Implementación de colaboración asincrónica basada en eventos

Colaboración basada en eventos

Interfaz con el cliente, Interacción con los clientes

DRY (Don't Repeat Yourself), DRY y los peligros de la reutilización de código en un Mundo de microservicios

Objetivos para, Buscando la Tecnología de Integración Ideal

Directrices para, Resumen

Importancia de la Integración

Orquestación versus coreografía, Orquestación versus coreografía

extensiones reactivas, extensiones reactivas

Llamadas a procedimientos remotos, Llamadas a procedimientos remotos

REST (Transferencia de estado representacional), REST

servicios como máquinas de estados, Servicios como máquinas de estados

bases de datos compartidas, La base de datos compartida

Comunicación sincrónica vs. asincrónica, Síncronica vs. Asincrónica

software de terceros, integración con software de terceros

Interfaces de usuario, Interfaces de usuario

control de versiones, Control de versiones

interfaces

coexistiendo lo nuevo y lo viejo, coexisten diferentes puntos finales

Implementación, una interfaz de implementación

Establecimiento de normas para interfaces

(ver también interfaces de usuario)

Detalle de implementación interna, Ocultar detalle de implementación interna

modelo interno de código abierto, código abierto interno

sistemas de detección de intrusiones (IDS), sistema de detección (y prevención) de intrusiones

Sistemas de prevención de intrusiones (IPS), Sistema de detección (y prevención) de intrusiones

aislamiento, aislamiento

Arquitectos de TI (ver arquitectos de sistemas)

Yo

JSON, JSON, XML o ¿algo más?

Tokens web JSON (JWT), HMAC sobre HTTP

K

Karyon, plantilla de servicio a medida

autenticación basada en claves, [claves API](#)

Kibana, [troncos, troncos y más troncos...](#)

yo

Latencia, [¿cuánto es demasiado?](#)

Latency Monkey, [la organización antifrágil](#)

Arquitecturas en capas, [microservicios](#)

bibliotecas

Cliente, [Bibliotecas de clientes](#)

Métricas de servicio, [Métricas de servicio](#)

compartido, [Bibliotecas compartidas](#)

Contenedores Linux, [Contenedores Linux](#)

equilibrio de carga, [Balanceo de carga](#)

deslastre de carga, [mamparos](#)

Llamadas locales, [Las llamadas locales no son como las llamadas remotas](#)

registros

Agregados, [registros, registros y más registros...](#)

(ver también seguimiento)

Problemas de seguridad, [Registro](#)

estandarización de, [Estandarización](#)

logstash, [registros, registros y más registros...](#)

acoplamiento suelto, [acoplamiento suelto, orquestación versus coreografía, acoplamiento suelto y](#)

[Organizaciones estrechamente acopladas](#)

METRO

Ataques de intermediarios: [Permitir todo dentro del perímetro](#)

Cuadrante de Marick, Tipos de pruebas

madurez, madurez

tiempo medio entre fallos (MTBF), ¿tiempo medio de reparación sobre el tiempo medio entre fallos?

tiempo medio de reparación (MTTR), ¿tiempo medio de reparación sobre el tiempo medio entre fallos?

corredores de mensajes, opciones tecnológicas

métrica

bibliotecas para Métricas de Servicio

Métricas de servicio, Métricas de servicio

Seguimiento en múltiples servicios, Seguimiento de métricas en múltiples servicios
Servicios

Biblioteca de métricas, plantilla de servicio personalizada

microservicios

Aplicación adecuada de ¿Cuándo no debería utilizar microservicios?

Autonomía y, Autónomo, Construyendo un Equipo

Beneficios de los microservicios

Componibilidad de, Componibilidad

Definición del término, ¿Qué son los microservicios?

facilidad de implementación, facilidad de implementación

Desventajas de, No existe una solución milagrosa, ¿Cuándo no debería utilizar microservicios?

Principios clave de, Unir todo

Alineación organizacional y, Alineación organizacional

Orígenes de los microservicios

reemplazabilidad y optimización para reemplazabilidad

resiliencia de, resiliencia

escalamiento y, escalamiento

Tamaño y, Pequeño, y Enfocado en Hacer una Cosa Bien

heterogeneidad tecnológica de, heterogeneidad tecnológica

vs. módulos, Módulos

vs. arquitectura orientada a servicios, ¿Qué pasa con la arquitectura orientada a servicios?
¿Arquitectura?

vs. bibliotecas compartidas, Bibliotecas compartidas

microservicios a escala

sistemas antifragiles, La Organización Antifragil

Medidas de seguridad arquitectónica, Medidas de seguridad arquitectónica

escalamiento automático, Escalamiento automático

almacenamiento en caché, almacenamiento en caché

almacenamiento en caché para escrituras, almacenamiento en caché para escrituras

Teorema CAP, Teorema CAP

Requisitos multifuncionales (CFR), ¿Cuánto es demasiado?

Cómo afrontar los fracasos: el fracaso está en todas partes

funcionalidad degradante, funcionalidad degradante

Servicios de documentación, Servicios de documentación

registros de servicios dinámicos, registros de servicios dinámicos

operaciones idempotentes, idempotencia

escalamiento, Escalamiento

escalado de bases de datos, [escalado de bases de datos](#)

Sistemas autodescriptivos, [HAL](#) y el navegador [HAL](#)

descubrimiento de servicios, [descubrimiento de servicios](#)

middleware, [opciones tecnológicas](#)

burlarse vs. stubbing, [burlarse o stubbing](#)

servicios de modelado

Contextos delimitados, [El contexto delimitado](#)

Capacidades empresariales, [Capacidades empresariales](#)

Conceptos de negocios, [Comunicación en términos de conceptos de negocios](#)

Conceptos clave: [¿Qué hace que un servicio sea bueno?](#)

módulos y servicios, [Módulos y Servicios](#)

Contextos delimitados anidados, [Tortugas hasta el final](#)

descomposición prematura, [descomposición prematura](#)

modelos compartidos vs. modelos ocultos, [modelos compartidos y ocultos](#)

límites técnicos, [El límite técnico](#)

descomposición modular, [módulos](#)

módulos, [Módulos y Servicios](#)

escucha

Fallos en cascada, [La Cascada](#)

Registro central, [registros, registros y más registros...](#)

complejidades de, [Monitoreo](#)

identificadores de correlación, [identificadores de correlación](#)

Mostrar/compartir resultados, [considere a la audiencia](#)

Seguimiento de métricas en múltiples servicios, Seguimiento de métricas en

Servicios Múltiples

múltiples servicios/múltiples servidores, Múltiples servicios, Múltiples servidores

Visión general de, Resumen

Informes en tiempo real, El Futuro

semántica, Implementación de monitoreo semántico

Métricas de servicio, Métricas de servicio

servicio único/varios servidores, servicio único, varios servidores

servicio único/servidor único, servicio único, servidor único

estandarización de, Estandarización

Establecimiento de normas y seguimiento

sintético, Monitoreo Sintético

sistemas monolíticos

bases de código pequeñas y enfocadas en hacer una cosa bien

Falta de cohesión/acoplamiento débil en, Todo es cuestión de costuras

Bases de datos de informes en, La base de datos de informes

vs. arquitectura orientada a servicios, ¿Qué pasa con la arquitectura orientada a servicios?

Ley de Moore, Ley de Conway y Diseño de Sistemas

Charlatán, un servicio de talones más inteligente

MTBF (tiempo medio entre fallos), ¿tiempo medio de reparación sobre el tiempo medio entre fallos?

MTTR (tiempo medio de reparación), ¿tiempo medio de reparación sobre el tiempo medio entre fallos?

norte

Contextos delimitados anidados, [Tortugas hasta el final](#)

segregación de red, [segregación de red](#)

Requisitos no funcionales, [pruebas multifuncionales](#)

Normalización de la desviación, [pruebas frágiles y escamosas](#)

Oh

sistemas de aprovisionamiento bajo demanda, [escalamiento](#)

Virtualización bajo demanda, [microservicios](#)

Arquitectura de cebolla, [El límite técnico](#)

Proyecto de seguridad de aplicaciones web abiertas (OWASP), [seguridad en el hogar](#)

OpenID Connect, [Implementaciones comunes de inicio de sesión único](#), Uso de SAML o [Conexión OpenID](#)

Artefactos del sistema operativo, [Artefactos del sistema operativo](#)

seguridad de los sistemas operativos, [Sistema operativo](#)

Arquitectura de orquestación, [Orquestación versus Coreografía](#)

Alineación organizacional, [Alineación organizacional](#)

Estructura organizacional

Ley de Conway [y diseño de sistemas](#)

Efecto en el diseño de sistemas, [Evidencia](#)

Organizaciones flexibles y organizaciones fuertemente acopladas

Servicios huérfanos, ¿ [El Servicio Huérfanos?](#)

OSGI (Iniciativa de acceso a código abierto), [módulos](#)

propiedad

compartido, [Impulsores de los servicios compartidos](#)

Diseño de sistemas y [propiedad del servicio](#)

PAG

Empaquetador, [imágenes personalizadas](#)

Pacto, [pacto](#)

Pacto, [pacto](#)

tolerancia de partición

en el teorema CAP, [Teorema CAP](#)

sacrificando, [¿Sacrificar la tolerancia de partición?](#)

Contraseñas, [vaya con las más conocidas](#)

pruebas de rendimiento, [pruebas de rendimiento](#)

Plataforma como servicio (PaaS), [Plataforma como servicio](#)

Artefactos específicos de la plataforma, [Artefactos específicos de la plataforma](#)

Polly para .NET, [mamparos](#)

Ley de Postel: [aplazarla lo máximo posible](#)

escalamiento predictivo, [escalamiento automático](#)

Parte principal, [Autenticación y Autorización](#)

Problemas de privacidad: [Sea frugal](#)

Pruebas de propiedad, [tipos de pruebas](#)

almacenamiento en caché de proxy, almacenamiento [en caché del lado del cliente, proxy y del lado del servidor](#)

R

RabbitMQ, [opciones tecnológicas](#)

RDBMS (sistemas de gestión de bases de datos relacionales), [escalabilidad para lecturas](#)

extensiones reactivas (Rx), [extensiones reactivas](#)

escalamiento reactivo, **escalamiento automático**

rélicas de lectura, **escalado para lecturas**

Rediseño, **Empezar de nuevo**

Refactorización de bases de datos, **preparación de la ruptura**

l llamadas a procedimientos remotos, **l llamadas a procedimientos remotos: ¿ RPC es terrible?**

Beneficios y desventajas de, **¿RPC es terrible?**

fragilidad, **fragilidad**

Definición del término, **l llamadas a procedimientos remotos**

tecnologías disponibles, **l llamadas a procedimientos remotos**

acoplamiento de tecnología, **acoplamiento de tecnología**

vs. l llamadas locales, **Las l llamadas locales no son como las l llamadas remotas**

Reemplazabilidad, **Optimización para la reemplazabilidad**

bases de datos de informes

bombas de datos de respaldo, **bomba de datos de respaldo**

Pautas para la bomba de datos, **Bombas de datos**

Recuperación de datos mediante l llamadas de servicio, **Recuperación de datos mediante l llamadas de servicio**

Bombas de datos de eventos, **Bomba de datos de eventos**

Sistemas de eventos genéricos, **hacia el tiempo real**

Enfoque monolítico para **la base de datos de informes**

Software de terceros, **recuperación de datos mediante l llamadas de servicio**

Colaboración solicitud/respuesta, **sincrónica versus asincrónica**

Ingeniería de resiliencia, **Resiliencia, Almacenamiento en caché para la resiliencia**

recursos, **DESCANSO**

Tiempo de respuesta, [¿cuánto es demasiado?](#)

REST (Transferencia de estado representacional), [REST: desventajas de REST sobre REST](#)
[HTTP](#)

Especificaciones ATOM, [opciones tecnológicas](#)

concepto de, [DESCANSO](#)

Marcos para, [Cuidado con la demasiada comodidad](#)

Beneficios de HTTP sobre REST, [REST y HTTP](#)

Desventajas de HTTP sobre REST, [Desventajas de REST sobre HTTP](#)

¿ Formatos textuales, [JSON](#), [XML](#) o algo más?

Proxy inverso, lado del cliente, proxy y almacenamiento en caché del lado del servidor

Riemann, [El Futuro](#)

riesgo, propagación, [Distribuir el riesgo](#)

Principio de robustez: [aplazarlo tanto como sea posible](#)

Rx (extensiones reactivas), [extensiones reactivas](#)

S

SAML, [Implementaciones comunes de inicio de sesión único](#), Uso de SAML o OpenID
Conectar

SAN (redes de área de almacenamiento): [distribución del riesgo](#)

escalando, [escalando-empezando de nuevo](#)

escalamiento automático, [Escalamiento automático](#)

Beneficios de, [Escalamiento](#)

bases de datos, [escalado de bases de datos](#)

equilibrio de carga, [Balanceo de carga](#)

Razones para, [Escalamiento](#)

división de cargas de trabajo, **división de cargas de trabajo**

Distribuir el riesgo, **Distribuir el riesgo**

vertical, **ve más grande**

vs. rediseño, **empezar de nuevo**

sistemas basados en trabajadores, **sistemas basados en trabajadores**

Costuras, concepto de, **Todo es cuestión de costuras**

seguridad, **Resumen de seguridad**

Ataques, defensa de, **Defensa en profundidad**

autenticación/autorización, **Autenticación y autorización**

copias de seguridad, **Backup Data Pump, Cifrar copias de seguridad**

Educación/concienciación sobre **la seguridad en la cocción**

cifrado, **vaya con lo conocido**

Ejemplo de configuración, **un ejemplo práctico**

verificación externa de, **Verificación Externa**

cortafuegos, **cortafuegos**

elemento humano de, **El Elemento Humano**

Importancia de **la seguridad**

Detección/prevención de intrusiones, **Detección (y prevención) de intrusiones**

Sistema

Almacenamiento de llaves: **todo es cuestión de llaves**

registros, **Registro**

Ataques de intermediarios: **Permitir todo dentro del perímetro**

sistemas operativos, **Sistema Operativo**

Visión general de, [Resumen](#)

Contraseñas, [vaya con las más conocidas](#)

Problemas de privacidad: [Sea frugal](#)

Asegurar los datos en reposo, [Asegurar los datos en reposo](#)

autenticación/autorización de servicio a servicio, [servicio a servicio](#)

[Autenticación y autorización](#)

Selección de herramientas, [la regla de oro](#)

Nubes privadas virtuales, [segregación de redes](#)

Ciclo de vida del desarrollo de seguridad, [integración de la seguridad](#)

Sistemas autodescriptivos, [HAL y el navegador HAL](#)

Monitoreo semántico, [¿Deberíamos utilizar pruebas de extremo a extremo?](#), Monitoreo sintético

Versionado semántico, [utilizar el versionado semántico](#)

almacenamiento en caché del lado del servidor, [almacenamiento en caché del lado del cliente, proxy y del lado del servidor](#)

Cuentas de servicio, [utilice SAML o OpenID Connect](#)

límites de servicio, [zonificación](#)

(ver también servicios de modelado)

llamadas de servicio, recuperación de datos a través de, [recuperación de datos a través de llamadas de servicio](#)

configuración del servicio, [configuración del servicio](#)

descubrimiento de servicios, [descubrimiento de servicios](#)

propiedad del servicio

Enfoque integral, [propiedad del servicio](#)

compartido, [Impulsores de los servicios compartidos](#)

Proveedor de servicios, [Implementaciones comunes de inicio de sesión único](#)

separación de servicios, puesta en escena, **puesta en escena de la ruptura**

(ver también descomposición de la base de datos)

Plantillas de servicio, **Plantilla de servicio personalizada**

pruebas de servicio

Pirámide de pruebas de Cohn, **alcance de las pruebas**

Implementación de, **Implementación de pruebas de servicio**

burlarse vs. stubbing, **burlarse o stubbing**

Servidor de Mountebank para, **un servicio de stub más inteligente**

Alcance de **las pruebas de servicio**

Arquitecturas orientadas a servicios (SOA)

Concepto de **¿Qué pasa con la arquitectura orientada a servicios?**

Desventajas de, **¿Qué pasa con la arquitectura orientada a servicios?**

reutilización de funcionalidad en, **Componibilidad**

vs. microservicios, **¿Qué pasa con la arquitectura orientada a servicios?**

Mapeo de servicio a host, Mapeo **de servicio a host : plataforma como servicio**

contenedores de aplicaciones, **contenedores de aplicaciones**

múltiples servicios por host, **Múltiples servicios por host**

Plataforma como servicio (PaaS), **Plataforma como servicio**

servicio único por host, **servicio único por host**

Terminología, **Mapeo de servicio a host**

autenticación/autorización de servicio a servicio, **servicio a servicio**

Autenticación y autorización: el problema del diputado

Claves API, **Claves API**

certificados de cliente, **certificados de cliente**

Problema del diputado confundido, [El problema del diputado](#)

HMAC sobre HTTP, [HMAC sobre HTTP](#)

HTTP(S) básico, [autenticación básica HTTP\(S\)](#)

Ataques de intermediarios: [Permitir todo dentro del perímetro](#)

SAML/OpenID Connect, [utilizar SAML o OpenID Connect](#)

Fragmentación, [escalado para escrituras](#)

Código compartido, [DRY y los peligros de la reutilización de código en un mundo de microservicios](#)

Datos compartidos, [Ejemplo: Datos compartidos](#)

bibliotecas compartidas, [Bibliotecas compartidas](#)

modelos compartidos, [modelos compartidos y ocultos](#)

Datos estáticos compartidos, [Ejemplo: Datos estáticos compartidos](#)

Tablas compartidas, [Ejemplo: Tablas compartidas](#)

Comportamiento compartido, [La base de datos compartida](#)

Inicio de sesión único (SSO), [Implementaciones comunes de inicio de sesión único: inicio de sesión único](#)

Puerta

Conjuntos de pruebas de humo: [separación de la implementación del lanzamiento](#)

Espías, [burlándose o apuñalando](#)

Multiplexación SSH, [servicio único, varios servidores](#)

Certificados SSL, [autenticación básica HTTP\(S\)](#)

Terminación de SSL, [Balanceo de carga](#)

cumplimiento de normas

Ejemplares, [Ejemplares](#)

Plantillas de servicios personalizados, [Plantilla de servicio personalizado](#)

Establecimiento de normas, Norma requerida : seguridad arquitectónica

seguridad arquitectónica, seguridad arquitectónica

Importancia de, El Estándar Requerido

interfaces, Interfaces

seguimiento, Monitoreo

Datos estáticos, Ejemplo: Datos estáticos compartidos

Patrón de aplicación Strangler, El patrón Strangler, Seguridad arquitectónica

Medidas

objetivos estratégicos

Ejemplo del mundo real, Un ejemplo del mundo real

comprensión, objetivos estratégicos

stubbing vs. mocking, burlarse o stubbing

Suro, el futuro

Fanfarronería, fanfarronería

Comunicación sincrónica, sincrónica versus asincrónica

Monitoreo sintético, Monitoreo Sintético

Diseño del sistema

rendición de cuentas y, Personas

Adaptación a las vías de comunicación, Adaptación a la comunicación

Caminos

Contextos delimitados, Contextos delimitados y estructuras de equipo

Estudio de caso, Estudio de caso: RealEstate.com.au

Ley de Conway, Ley de Conway y Diseño de Sistemas

cuellos de botella en la entrega, cuellos de botella en la entrega

Efecto sobre la estructura organizacional, Ley de Conway al revés

equipos destacados, equipos destacados

modelo interno de código abierto, código abierto interno

Estructura organizacional y, Evidencia

Servicios huérfanos, ¿ El Servicio Huérfanos?

Visión general de, Resumen

papel de los custodios, papel de los custodios

madurez del servicio, madurez

propiedad del servicio, propiedad del servicio

Propiedad compartida de servicios, impulsores de los servicios compartidos

herramientas, herramientas

arquitectos de sistemas

desafíos que enfrenta, Resumen

Directrices para la toma de decisiones: Un enfoque basado en principios

Manejo de excepciones, Manejo de excepciones

Gobernanza, Gobernanza y Liderazgo desde el Centro

Responsabilidades de, Comparaciones inexactas, Deuda técnica, Resumen

El papel de la arquitectura: una visión evolutiva para el arquitecto

límites de servicio y zonificación

Aplicación de normas mediante la gobernanza a través de un servicio adaptado al código
Plantilla

Establecimiento de normas por, El Estándar Requerido

Formación de equipos por, Formación de un equipo

participación del equipo por, [Zonificación](#)

deuda técnica y, [Deuda Técnica](#)

yo

tablas, compartidas, [Ejemplo: Tablas compartidas](#)

Plantillas de servicios personalizados, [Plantilla de servicio personalizado](#)

formación de equipos, [Construir un equipo](#)

Estructuras de equipo, [contextos delimitados y estructuras de equipo](#)

límites técnicos, [El límite técnico](#)

deuda técnica, [deuda técnica](#)

heterogeneidad tecnológica, [heterogeneidad tecnológica](#)

API independientes de la tecnología: [mantenga sus API independientes de la tecnología](#)

Pruebas orientadas a la tecnología, [Tipos de pruebas](#)

Plantillas, [Plantilla de servicio a medida](#)

prueba de dobles, [Mocking o Stubbing](#)

Pirámide de pruebas, [alcance de las pruebas y pruebas orientadas al consumidor al rescate](#)

Cono de nieve de prueba, [¿cuántos?](#)

Diseño basado en pruebas (TDD), [pruebas unitarias](#)

pruebas

suelta de canarios, [suelta de canarios](#)

Consideraciones para [las compensaciones](#)

Pruebas impulsadas por el consumidor, [pruebas impulsadas por el consumidor al rescate](#)

[Pruebas](#) multifuncionales

Pruebas de extremo a extremo, [esas complicadas pruebas de extremo a extremo](#)

MTTR sobre MTBF, ¿tiempo medio de reparación sobre tiempo medio entre fallos?

Visión general de, Resumen

pruebas de rendimiento, pruebas de rendimiento

Postproducción, Pruebas posteriores a la producción

alcance de, alcance de la prueba

seleccionando numero de, ¿cuantos?

Monitoreo semántico: ¿Deberíamos utilizar pruebas de extremo a extremo?

Separar la implementación del lanzamiento, Separar la implementación del lanzamiento
Liberar

Implementación de pruebas de servicio, Implementación de pruebas de servicio

tipos de pruebas, Tipos de Pruebas

Software de terceros, Integración con software de terceros: The Strangler

Patrón, recuperación de datos mediante llamadas de servicio

Construir vs. comprar, Integración con software de terceros

Sistemas de gestión de contenidos (CMS), Ejemplo: CMS como servicio

Gestión de la relación con el cliente (CRM), ejemplo: el rol múltiple

Sistema CRM

personalización de, Personalización

Problemas de integración, espaguetis de integración

falta de control sobre, falta de control

Bases de datos de informes, recuperación de datos mediante llamadas de servicio

Patrón de aplicación Strangler, El patrón Strangler

Acoplamiento estrecho, Acoplamiento débil, Organizaciones acopladas débil y fuertemente acopladas

tiempo de vida (TTL), DNS

tiempos de espera, tiempos de espera

Gestores de transacciones, **Transacciones distribuidas**

Límites transaccionales, **Límites transaccionales: ¿Qué hacer entonces?**

actas

Compensando, **abortar toda la operación**

Distribuido, **Transacciones Distribuidas**

Seguridad de la capa de transporte (TLS), **certificados de cliente**

confirmaciones de dos fases, **transacciones distribuidas**

Virtualización tipo 2, **Virtualización tradicional**

tú

UDDI (Descripción, descubrimiento e integración universales), **HAL y HAL**

Navegador

pruebas unitarias

Pirámide de pruebas de Cohn, **alcance de las pruebas**

objetivos de, **Pruebas unitarias**

Cuadrante de Marick, **Tipos de pruebas**

alcance de, **pruebas unitarias**

Interfaces de usuario, **Interfaces de usuario: un enfoque híbrido**

Composición de la API, **Composición de la API**

Pasarelas API, **backends para frontends**

Granularidad de API, **hacia lo digital**

Pirámide de pruebas de Cohn, **alcance de las pruebas**

(ver también pruebas de extremo a extremo)

restrictions, [Restricciones](#)

evolución de las interfaces de usuario

Composición de fragmentos, [Composición de fragmentos de UI](#)

Enfoques híbridos, [Un enfoque híbrido](#)

V

Vagabundo, [vagabundo](#)

Control de versiones, [Control de versiones: utilizar múltiples versiones de servicio concurrentes](#)

Detectar cambios importantes a tiempo, [Detectar cambios importantes a tiempo](#)

coexistencia de diferentes puntos finales, [coexistencia de diferentes puntos finales](#)

Aplazar los cambios importantes, [aplazarlo tanto como sea posible](#)

Varias versiones concurrentes, [utilizar varias versiones de servicio concurrentes](#)

semántico, [utilizar versiones semánticas](#)

escalamiento vertical, [ve más grande](#)

Nubes privadas virtuales (VPC), [segregación de red](#)

virtualización

Hipervisores, [Virtualización tradicional](#)

tradicional, [Virtualización tradicional](#)

Tipo 2, [Virtualización tradicional](#)

plataformas de virtualización

Docker, [Docker](#)

Contenedores Linux, [Contenedores Linux](#)

Bajo demanda, [microservicios](#)

Redes de área de almacenamiento en, [Distribuyendo su riesgo](#)

Vagabundo, [vagabundo](#)

visión, [Resumen](#)

Yo

sistemas basados en trabajadores, [sistemas basados en trabajadores](#)

Cachés de escritura posterior: [almacenamiento en caché para la resiliencia](#)

Incógnita

XML, JSON, XML o ¿algo más?

O

Zed Attack Proxy (ZAP), [la seguridad en marcha](#)

Zipkin, [identificadores de correlación](#)

Guardián del zoológico, [guardián del zoológico](#)

Acerca del autor

Sam Newman es tecnólogo en ThoughtWorks, donde actualmente divide su tiempo entre ayudar a los clientes y trabajar como arquitecto de los sistemas internos de ThoughtWorks. Ha trabajado con una variedad de empresas en múltiples dominios en todo el mundo, a menudo con un pie en el mundo de los desarrolladores y otro en el espacio de operaciones de TI.

Si le preguntaran qué hace, diría: "Trabajo con personas para construir mejores sistemas de software". Ha escrito artículos, realizado presentaciones en conferencias y participa esporádicamente en proyectos de código abierto.

Colofón

Los animales que aparecen en la portada de Building Microservices son abejas melíferas (del género Apis). De las 20.000 especies conocidas de abejas, solo siete se consideran abejas melíferas. Se distinguen porque producen y almacenan miel, además de construir colmenas a partir de cera. La apicultura para recolectar miel ha sido una actividad humana durante miles de años.

Las abejas melíferas viven en colmenas de miles de individuos y tienen una estructura social muy organizada. Existen tres castas: reina, zángano y obrera. Cada colmena tiene una reina, que permanece fértil durante 3 a 5 años después de su vuelo nupcial y pone hasta 2000 huevos por día.

Los zánganos son abejas macho que se aparean con la reina (y mueren en el acto debido a sus órganos sexuales con púas). Las abejas obreras son hembras estériles que desempeñan muchas funciones durante su vida, como niñera, trabajadora de la construcción, tendera, guardiana, funeraria y recolectora.

Las abejas obreras recolectoras se comunican entre sí "bailando" en patrones particulares para compartir información sobre los recursos cercanos.

Las tres castas de abejas melíferas son similares en apariencia, con alas, seis patas y un cuerpo dividido en cabeza, tórax y abdomen. Tienen pelos cortos y difusos con un patrón de rayas amarillas y negras. Su dieta se compone exclusivamente de miel, que se crea mediante un proceso de digestión parcial y regurgitación del néctar de las flores, rico en azúcar.

Las abejas son fundamentales para la agricultura, ya que polinizan los cultivos y otras plantas con flores mientras recolectan polen y néctar. En promedio, cada colmena de abejas recolecta 66 libras de polen al año. En los últimos años, la disminución de muchas especies de abejas ha sido motivo de preocupación y se conoce como "síndrome de colapso de colonias". Todavía no está claro qué está causando esta muerte: algunas teorías incluyen parásitos, uso de insecticidas o enfermedades, pero hasta la fecha no se han encontrado medidas preventivas efectivas.

Muchos de los animales que aparecen en las portadas de O'Reilly están en peligro de extinción; todos ellos son importantes para el mundo. Para obtener más información sobre cómo puedes ayudar, visita animals.oreilly.com.

La imagen de la portada es de la Historia natural de Johnson . Las fuentes de la portada son URW Typewriter y Guardian Sans. La fuente del texto es Adobe Minion Pro; la fuente del encabezado es Adobe Myriad Condensed; y la fuente del código es Ubuntu Mono de Dalton Maag.

Prefacio

¿Quién debería leer este libro?

¿Por qué escribí este libro?

Unas palabras sobre los microservicios hoy

Navegando por este libro

Convenciones utilizadas en este libro

Libros en línea de Safari®

Cómo contactarnos

Expresiones de gratitud

1. Microservicios

¿Qué son los microservicios?

Pequeño y centrado en hacer una cosa bien

Autónomo

Beneficios clave

Heterogeneidad tecnológica

Resiliencia

Escalada

Facilidad de implementación

Alineación organizacional

Componibilidad

Optimización para la reemplazabilidad

¿Qué pasa con la arquitectura orientada a servicios?

Otras técnicas de descomposición

Bibliotecas compartidas

Módulos

No hay bala de plata

Resumen

2. El arquitecto evolutivo Comparaciones

inexactas

Una visión evolutiva para el arquitecto

Zonificación

Un enfoque basado en principios

Metas estratégicas

Principios

Prácticas

Combinando principios y prácticas

Un ejemplo del mundo real

El estándar requerido

Escucha

Interfaces

Seguridad arquitectónica

Gobernanza a través del código

Ejemplares

Plantilla de servicio personalizado

Deuda técnica

Manejo de excepciones

Gobernanza y liderazgo desde el centro

Construyendo un equipo

Resumen

3. Cómo modelar servicios

Presentando MusicCorp

¿Qué hace que un servicio sea bueno?

Acoplamiento suelto

Alta cohesión

El contexto delimitado

Modelos compartidos y ocultos

Módulos y Servicios

Descomposición prematura

Capacidades empresariales

Tortugas hasta el fondo

La comunicación en términos de conceptos empresariales

El límite técnico

Resumen

4. Integración

En busca de la tecnología de integración ideal

Evite cambios disruptivos

Mantenga sus API independientes de la tecnología

Simplifique su servicio para los consumidores

Ocultar detalles de implementación interna

Interactuar con los clientes

La base de datos compartida

Sincrónico versus asincrónico

Orquestación versus coreografía

Llamadas a procedimientos remotos

Acoplamiento de tecnología

Las llamadas locales no son como las llamadas remotas

Fragilidad

¿Es terrible el RPC?

REST y HTTP

Hipermedia como motor del estado de la aplicación

¿JSON, XML o algo más?

Cuidado con la comodidad excesiva

Desventajas de REST sobre HTTP

Implementación de la colaboración asincrónica basada en eventos

Opciones tecnológicas

Complejidades de las arquitecturas asincrónicas

Los servicios como máquinas de estados

Extensiones reactivas

DRY y los peligros de la reutilización de código en un mundo de microservicios

Bibliotecas de clientes

Acceso por referencia

Control de versiones

Aplazarlo tanto como sea posible

Detecte cambios importantes a tiempo

Utilice el control de versiones semántico

Coexisten diferentes puntos finales

Utilice múltiples versiones de servicio concurrentes

Interfaces de usuario

Hacia lo digital

Restricciones

Composición de la API

Composición de fragmentos de interfaz de usuario

Backends para frontends

Un enfoque híbrido

Integración con software de terceros

Falta de control

Personalización

Espaguetis de integración

En tus propios términos

El patrón del estrangulador

Resumen

5. División del monolito

Todo es cuestión de costuras

Desmembrando MusicCorp

Las razones para dividir el monolito

Ritmo del cambio

Estructura del equipo

Seguridad

Tecnología

Dependencias enredadas

La base de datos

Enfrentando el problema

Ejemplo: Romper relaciones de claves externas

Ejemplo: Datos estáticos compartidos

Ejemplo: Datos compartidos

Ejemplo: Tablas compartidas

Refactorización de bases de datos

Preparando el descanso

Límites transaccionales

Inténtalo de nuevo más tarde

[Cancelar toda la operación](#)

[Transacciones distribuidas](#)

[Entonces, ¿qué hacer?](#)

[Informes](#)

[La base de datos de informes](#)

[Recuperación de datos mediante llamadas de servicio](#)

[Bombas de datos](#)

[Destinos alternativos](#)

[Bomba de datos de eventos](#)

[Bomba de datos de respaldo](#)

[Hacia el tiempo real](#)

[Costo del cambio](#)

[Comprender las causas fundamentales](#)

[Resumen](#)

[6. Despliegue](#)

[Una breve introducción a la integración continua ¿Realmente lo estás haciendo?](#)

[Mapeo de la integración continua a los microservicios](#)

[Construir pipelines y entrega continua](#)

[Y las inevitables excepciones](#)

[Artefactos específicos de la plataforma](#)

[Artefactos del sistema operativo](#)

[Imágenes personalizadas](#)

[Las imágenes como artefactos](#)

[Servidores inmutables](#)

[Entornos](#)

Configuración del servicio

Mapeo de servicio a host

Múltiples servicios por host

Contenedores de aplicaciones

Servicio único por host

Plataforma como servicio

Automatización

Dos estudios de caso sobre el poder de la automatización

De lo físico a lo virtual

Virtualización tradicional

Vagabundo

Contenedores de Linux

Estibador

Una interfaz de implementación

Definición de medio ambiente

Resumen

7. Tipos de

pruebas

Alcance de la prueba

Pruebas unitarias

Pruebas de servicio

Pruebas de extremo a extremo

Compensaciones

¿Cuántos?

Implementación de pruebas de servicio

Burlarse o tropezar

Un servicio de talón más inteligente

Esas complicadas pruebas de extremo a extremo

Desventajas de las pruebas de extremo a extremo

Pruebas de fragilidad y descamación

¿Quién escribe estas pruebas?

¿Cuánto tiempo?

La gran colisión

La metaversión

Viajes de prueba, no historias

Las pruebas impulsadas por el consumidor vienen al rescate

Pacto

Se trata de conversaciones

Entonces, ¿debería utilizar pruebas de extremo a extremo?

Pruebas después de la producción

Separación entre implementación y lanzamiento

Liberación de canarios

¿Tiempo medio de reparación frente al tiempo medio entre fallos?

Pruebas multifuncionales

Pruebas de rendimiento

Resumen

8. Seguimiento

Un solo servicio, un solo servidor

Un solo servicio, varios servidores

Múltiples servicios, múltiples servidores

Registros, registros y más registros...

Seguimiento de métricas en múltiples servicios

Métricas de servicio

Monitoreo sintético

Implementación de la monitorización semántica

Identificadores de correlación

La cascada

Normalización

Considere a la audiencia

El futuro

Resumen

9. Seguridad

Autenticación y autorización

Implementaciones comunes de inicio de sesión único

Puerta de enlace de inicio de sesión único

Autorización de Granularidad Fina

Autenticación y autorización de servicio a servicio

Permitir todo dentro del perímetro

Autenticación básica HTTP(S)

Utilice SAML o OpenID Connect

Certificados de cliente

HMAC sobre HTTP

Claves API

El problema del diputado

Protección de datos en reposo

Vaya con lo conocido

Todo es cuestión de las llaves

Elige tus objetivos

Descifrado a pedido

Cifrar copias de seguridad

Defensa en profundidad

Cortafuegos

Explotación forestal

Sistema de detección (y prevención) de intrusiones

Segregación de la red

Sistema operativo

Un ejemplo práctico

Sea frugal

El elemento humano

La regla de oro

Seguridad en la cocción

Verificación externa

Resumen

10. Ley de Conway y diseño de sistemas

Evidencia

Organizaciones flexibles y fuertemente acopladas

Windows Vista

Netflix y Amazon

¿Qué podemos hacer con esto?

Adaptación a las vías de comunicación

Propiedad del servicio

Impulsores de los servicios compartidos

Demasiado difícil de dividir

Equipos destacados

Cuellos de botella en la entrega

Código abierto interno

Papel de los custodios

Madurez

Estampación

Contextos delimitados y estructuras de equipo

¿El servicio huérfano?

Estudio de caso: RealEstate.com.au

La ley de Conway al revés

Gente

Resumen

11. Microservicios a escala

El fracaso está en todas partes

¿Cuánto es demasiado?

Funcionalidad degradada

Medidas de seguridad arquitectónica

La organización antifrágil

Tiempos de espera

Disyuntores

Mamparos

Aislamiento

Idempotencia

Escalada

Ve más grande

División de cargas de trabajo

Distribuyendo el riesgo

Equilibrio de carga

Sistemas basados en trabajadores

Empezando de nuevo

Escalado de bases de datos

Disponibilidad del servicio versus durabilidad de los datos

Escalado para lecturas

Escalado para escrituras

Infraestructura de base de datos compartida

CQRS

Almacenamiento en caché

Almacenamiento en caché del lado del cliente, proxy y servidor

Almacenamiento en caché en HTTP

Almacenamiento en caché para escrituras

Almacenamiento en caché para la resiliencia

Ocultando el origen

Mantenlo simple

Envenenamiento de caché: una historia que sirve de advertencia

Escalado automático

Teorema CAP

Sacrificar la consistencia

Sacrificar la disponibilidad

¿Sacrificar la tolerancia de partición?

¿AP o CP?

No es todo o nada

Y el mundo real

Descubrimiento de servicios

Sistema de nombres de dominio

Registros de servicios dinámicos

Guardián del zoológico

Cónsul

Eureka

Arma tu propio rollo

¡No olvidemos a los humanos!

Servicios de documentación

Pavonearse

HAL y el navegador HAL

El sistema de autodescripción

Resumen

12. Unir todo

Principios de los microservicios

Modelo en torno a conceptos empresariales

Adoptar una cultura de automatización

Ocultar detalles de implementación interna

Descentralizar todas las cosas

Desplegable de forma independiente

Aislar falla

Altamente observable

¿Cuándo no deberías utilizar microservicios?

Palabras de despedida

Índice