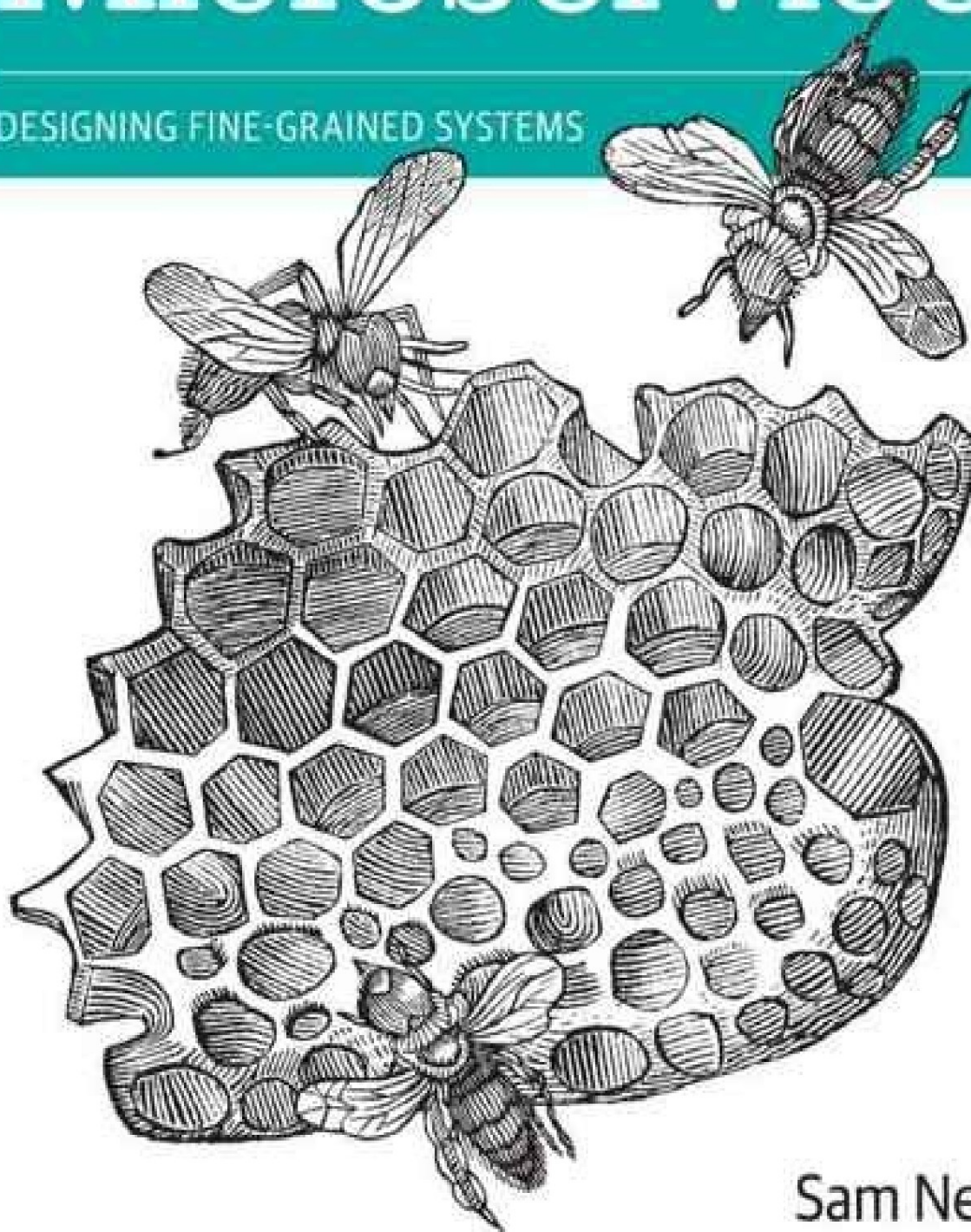


Building Microservices

DESIGNING FINE-GRAINED SYSTEMS



Sam Newman

Creación de microservicios

sam newman

Creación de microservicios

por Sam Newman

Derechos de autor © 2015 Sam Newman. Reservados todos los derechos.

Impreso en los Estados Unidos de América.

Publicado por O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Los libros de O'Reilly se pueden comprar con fines educativos, comerciales o de promoción de ventas.

Las ediciones en línea también están disponibles para la mayoría de los títulos (<http://safaribooksonline.com>). Para obtener más información, comuníquese con nuestro departamento de ventas corporativo/institucional: 800-998-9938 o *corporate@oreilly.com*.

- Editores: Mike Loukides y Brian MacDonald

- Editora de producción: Kristen Brown

- Correctora: Rachel Monaghan

- Correctora: Jasmine Kwityn

- Indexador: Judith McConville

- Diseñador de interiores: David Futato

- Diseño de portada: Ellie Volckhausen

- Ilustrador: Rebecca Demarest

- Febrero 2015: Primera Edición

Historial de revisiones de la primera edición

- 2014-01-30: Primer lanzamiento

Consulte <http://oreilly.com/catalog/errata.csp?isbn=9781491950357> para conocer los detalles de la versión.

El logotipo de O'Reilly es una marca comercial registrada de O'Reilly Media, Inc. *Building Microservices*, la imagen de portada de las abejas melíferas y la imagen comercial relacionada son marcas comerciales de O'Reilly Media, Inc.

Si bien el editor y el autor se han esforzado de buena fe para garantizar que la información y las instrucciones contenidas en este trabajo sean precisas, el editor y el autor renuncian a toda responsabilidad por errores u omisiones, incluida, entre otras, la responsabilidad por daños resultantes del uso o confianza en este trabajo. El uso de la información e instrucciones contenidas en este trabajo es bajo su propio riesgo. Si alguna muestra de código u otra tecnología que este trabajo contiene o describe está sujeta a licencias de código abierto o a los derechos de propiedad intelectual de otros, es su responsabilidad asegurarse de que su uso cumpla con dichas licencias y/o derechos.

978-1-491-95035-7

[LSI]

Prefacio

Los microservicios son un enfoque de los sistemas distribuidos que promueven el uso de servicios detallados con sus propios ciclos de vida, que colaboran entre sí. Debido a que los microservicios se modelan principalmente en torno a dominios comerciales, evitan los problemas de las arquitecturas tradicionales en niveles. Los microservicios también integran nuevas tecnologías y técnicas que surgieron durante la última década, lo que les ayuda a evitar los escollos de muchas implementaciones de arquitectura orientada a servicios.

Este libro está repleto de ejemplos concretos del uso de microservicios en todo el mundo, incluso en organizaciones como Netflix, Amazon, Gilt y el grupo REA, quienes han descubierto que la mayor autonomía que esta arquitectura brinda a sus equipos es una gran ventaja.

Quién debería leer este libro

El alcance de este libro es amplio, ya que las implicaciones de las arquitecturas de microservicios detalladas también son amplias. Como tal, debería atraer a personas interesadas en aspectos de diseño, desarrollo, implementación, prueba y mantenimiento de sistemas. Aquellos de ustedes que ya se han embarcado en el viaje hacia arquitecturas de grano más fino, ya sea para una aplicación totalmente nueva o como parte de la descomposición de un sistema más monolítico existente, encontrarán muchos consejos prácticos para ayudarlos. También ayudará a aquellos de ustedes que quieran saber de qué se trata todo este alboroto, para que puedan determinar si los microservicios son adecuados para usted.

Por qué escribí este libro

Empecé a pensar en el tema de las arquitecturas de aplicaciones hace muchos años, cuando trabajaba para ayudar a las personas a entregar su software más rápido. Me di cuenta de que si bien las técnicas de automatización de infraestructura, pruebas y entrega continua podrían ayudar, si el diseño fundamental del sistema no facilita la realización de cambios, entonces hay límites en lo que se puede lograr.

Al mismo tiempo, muchas organizaciones estaban experimentando con arquitecturas más detalladas para lograr objetivos similares, pero también para lograr cosas como escalabilidad mejorada, aumentar la autonomía de los equipos o adoptar nuevas tecnologías más fácilmente. Mis propias experiencias, así como las de mis colegas en ThoughtWorks y en otros lugares, reforzaron el hecho de que el uso de una mayor cantidad de servicios con sus propios ciclos de vida independientes generaba más dolores de cabeza que debían resolverse. En muchos sentidos, este libro se imaginó como una ventanilla única que ayudaría a abarcar la amplia variedad de temas que son necesarios para comprender los microservicios, ¡algo que me habría ayudado mucho en el pasado!

Una palabra sobre los microservicios hoy

Los microservicios son un tema de rápido movimiento. Aunque la idea no es nueva (incluso si el término en sí lo es), las experiencias de personas de todo el mundo, junto con la aparición de nuevas tecnologías, están teniendo un efecto profundo en la forma en que se utilizan. Debido al rápido ritmo de cambio, he tratado de enfocar este libro en ideas más que en tecnologías específicas, sabiendo que los detalles de implementación siempre cambian más rápido que los pensamientos detrás de ellos.

No obstante, espero que dentro de unos años hayamos aprendido aún más sobre dónde encajan los microservicios y cómo usarlos bien.

Así que, aunque hice lo mejor que pude para destilar la esencia del tema en este libro, si este tema le interesa, ¡prepárese para muchos años de aprendizaje continuo para mantenerse al tanto del estado del arte!

Navegando por este libro

Este libro está organizado principalmente en un formato basado en temas. Como tal, es posible que desee saltar a los temas específicos que más le interesen. Si bien hice todo lo posible para hacer referencia a términos e ideas en los capítulos anteriores, me gustaría pensar que incluso las personas que se consideran bastante experimentadas encontrarán algo de interés en todos los capítulos aquí. Sin duda, le sugiero que eche un vistazo al **Capítulo 2**, que toca la amplitud del tema y proporciona un marco sobre cómo hago las cosas en caso de que quiera profundizar en algunos de los temas posteriores.

Para las personas nuevas en el tema, he estructurado los capítulos de una manera que espero tenga sentido leer de principio a fin.

Aquí hay una descripción general de lo que cubrimos:

Capítulo 1, Microservicios

Comenzaremos con una introducción a los microservicios, incluidos los beneficios clave y algunas de las desventajas.

Capítulo 2, El Arquitecto Evolutivo

Este capítulo analiza las dificultades a las que nos enfrentamos en términos de hacer concesiones como arquitectos y cubre específicamente cuántas cosas debemos considerar con los microservicios.

Capítulo 3, Cómo modelar servicios

Aquí comenzaremos a definir el límite de los microservicios, usando técnicas de diseño basado en dominios para ayudar a enfocar nuestro pensamiento.

Capítulo 4, Integración

Aquí es donde comenzamos a profundizar un poco más en las implicaciones tecnológicas específicas, mientras discutimos qué tipo de técnicas de colaboración de servicios nos ayudarán más. También profundizaremos en el tema de las interfaces de usuario y la integración con productos heredados y comerciales listos para usar (COTS).

Capítulo 5, Dividir el monolito

Muchas personas se interesan en los microservicios como antídoto para los sistemas monolíticos grandes y difíciles de cambiar, y esto es exactamente lo que trataremos en detalle en este capítulo.

Capítulo 6, Implementación

Aunque este libro es principalmente teórico, pocos temas del libro se han visto tan afectados por los cambios recientes en la tecnología como la implementación, que exploraremos aquí.

Capítulo 7, Pruebas

Este capítulo profundiza en el tema de las pruebas, un área de particular interés cuando se maneja la implementación de múltiples servicios discretos. De particular importancia será el papel que pueden desempeñar los contratos impulsados por el consumidor para ayudarnos a garantizar la calidad de nuestro software.

Capítulo 8, Supervisión

Probar nuestro software antes de la producción no ayuda si ocurren problemas una vez que lo ponemos en marcha, y este capítulo explora cómo podemos monitorear nuestros sistemas detallados y lidiar con parte de la complejidad emergente de los sistemas distribuidos.

Capítulo 9, Seguridad

Aquí examinaremos los aspectos de seguridad de los microservicios y consideraremos cómo manejar la autenticación y autorización de usuario a servicio y de servicio a servicio. La seguridad es un tema muy importante en la informática, uno que se ignora con demasiada facilidad. Aunque de ninguna manera soy un experto en seguridad, espero que este capítulo al menos lo ayude a considerar algunos de los aspectos que debe tener en cuenta al crear sistemas y, en particular, sistemas de microservicios.

Capítulo 10, Ley de Conway y diseño de sistemas

Este capítulo se centra en la interacción de la estructura y la arquitectura organizativas. Muchas organizaciones se han dado cuenta de que surgirán problemas si no se mantienen los dos en armonía. Intentaremos llegar al fondo de este dilema y considerar algunas formas diferentes de alinear el diseño del sistema con la estructura de sus equipos.

Capítulo 11, Microservicios a escala Aquí

es donde comenzamos a buscar hacer todo esto a escala, para que podamos manejar la mayor probabilidad de falla que puede ocurrir con una gran cantidad de servicios, así como con grandes volúmenes de tráfico.

Capítulo 12, Uniéndolo todo

El capítulo final intenta destilar la esencia central de lo que hace que los microservicios sean diferentes. Incluye una lista de siete principios de microservicios, así como un resumen de los puntos clave del libro.

Las convenciones usadas en este libro

En este libro se utilizan las siguientes convenciones tipográficas:

Itálico

Indica nuevos términos, URL, direcciones de correo electrónico, nombres de archivo y extensiones de archivo.

Ancho constante

Se utiliza para listas de programas, así como dentro de párrafos para hacer referencia a elementos de programas como nombres de variables o funciones, bases de datos, tipos de datos, variables de entorno, declaraciones y palabras clave.

Negrita de ancho constante

Muestra comandos u otro texto que el usuario debe escribir literalmente.

Cursiva de ancho constante

Muestra texto que debe reemplazarse con valores proporcionados por el usuario o por valores determinados por el contexto.

Safari® Libros en línea

NOTA

Safari Books Online es una biblioteca digital a pedido que ofrece **contenido** experto en forma de libro y video de los principales autores del mundo en tecnología y negocios.

Los profesionales de la tecnología, los desarrolladores de software, los diseñadores web y los profesionales creativos y de negocios utilizan Safari Books Online como su recurso principal para la investigación, la resolución de problemas, el aprendizaje y la capacitación para la certificación.

Safari Books Online ofrece una variedad de **planes y precios** para **empresas, gobiernos, educación** e individuos.

Los miembros tienen acceso a miles de libros, videos de capacitación y manuscritos previos a la publicación en una base de datos de búsqueda completa de editoriales como O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology y cientos **más**. Para obtener más información sobre Safari Books Online, visítenos en **línea**.

Cómo contactarnos

Dirija sus comentarios y preguntas sobre este libro a la editorial:

- O'Reilly Media, Inc.
- 1005 Carretera Gravenstein Norte
- Sebastopol, CA 95472
- 800-998-9938 (en los Estados Unidos o Canadá)
- 707-829-0515 (internacional o local)
- 707-829-0104 (fax)

Tenemos una página web para este libro, donde enumeramos las erratas, ejemplos y cualquier información adicional. Puede acceder a esta página en <http://bit.ly/building-microservices>.

Para comentar o hacer preguntas técnicas sobre este libro, envíe un correo electrónico a bookquestions@oreilly.com.

Para obtener más información sobre nuestros libros, cursos, conferencias y noticias, visite nuestro sitio web en <http://www.oreilly.com>.

Encuétranos en Facebook: <http://facebook.com/oreilly>

Síguenos en Twitter: <http://twitter.com/oreillymedia>

Míranos en YouTube: <http://www.youtube.com/oreillymedia>

Expresiones de gratitud

Este libro está dedicado a Lindy Stephens, sin la cual no existiría. Ella me animó a comenzar este viaje, me apoyó durante el proceso de escritura, a menudo estresante, y es la mejor compañera que podría haber pedido. También me gustaría dedicar esto a mi padre, Howard Newman, quien siempre ha estado ahí para mí. Esto es para los dos.

Me gustaría destacar a Ben Christensen, Vivek Subramaniam y Martin Fowler por proporcionar comentarios detallados durante todo el proceso de escritura, ayudando a dar forma a lo que se convirtió en este libro. También me gustaría agradecer a James Lewis, con quien he bebido muchas cervezas discutiendo las ideas presentadas en este libro. Este libro sería una sombra de sí mismo sin su ayuda y guía.

Además, muchos otros brindaron ayuda y comentarios sobre las primeras versiones del libro. Específicamente, me gustaría agradecer (sin ningún orden en particular) a Kane Venables, Anand Krishnaswamy, Kent McNeil, Charles Haynes, Chris Ford, Aidy Lewis, Will Thames, Jon Eaves, Rolf Russell, Badrinath Janakiraman, Daniel Bryant, Ian Robinson, Jim Webber, Stewart Gleadow, Evan Bottcher, Eric Sword, Olivia Leonard y todos mis otros colegas en ThoughtWorks y en toda la industria que me han ayudado a llegar tan lejos.

Finalmente, me gustaría agradecer a toda la gente de O'Reilly, incluido Mike Loukides por ayudarme, mi editor Brian MacDonald, Rachel Monaghan, Kristen Brown, Betsy Waliszewski y todas las demás personas que me han ayudado nunca se sabe.

Capítulo 1. Microservicios

Durante muchos años, hemos estado encontrando mejores formas de construir sistemas. Hemos estado aprendiendo de lo que ha venido antes, adoptando nuevas tecnologías y observando cómo una nueva ola de empresas de tecnología opera de diferentes maneras para crear sistemas de TI que ayuden a hacer más felices tanto a sus clientes como a sus propios desarrolladores.

El libro de Eric Evans *Domain-Driven Design* (Addison-Wesley) nos ayudó a comprender la importancia de representar el mundo real en nuestro código y nos mostró mejores formas de modelar nuestros sistemas. El concepto de entrega continua mostró cómo podemos poner nuestro software en producción de manera más efectiva y eficiente, inculcando en nosotros la idea de que debemos tratar cada registro como un candidato de lanzamiento. Nuestra comprensión de cómo funciona la Web nos ha llevado a desarrollar mejores formas de hacer que las máquinas hablen con otras máquinas. El concepto de **arquitectura hexagonal** de Alistair Cockburn nos alejó de las arquitecturas en capas donde la lógica empresarial podría esconderse. Las plataformas de virtualización nos permitieron aprovisionar y cambiar el tamaño de nuestras máquinas a voluntad, con la automatización de la infraestructura brindándonos una forma de manejar estas máquinas a escala. Algunas organizaciones grandes y exitosas como Amazon y Google adoptaron la visión de que los equipos pequeños son dueños del ciclo de vida completo de sus servicios. Y, más recientemente, Netflix ha compartido con nosotros formas de construir sistemas antifrágiles a una escala que habría sido difícil de comprender hace solo 10 años.

Diseño dirigido por dominio. Entrega continua. Virtualización bajo demanda. Automatización de infraestructuras. Pequeños equipos autónomos. Sistemas a escala. Los microservicios han surgido de este mundo. No fueron inventados o descritos antes del hecho; surgieron como una tendencia, o un patrón, del uso en el mundo real. Pero existen sólo debido a todo lo que ha pasado antes.

A lo largo de este libro, sacaré hilos de este trabajo previo para ayudar a pintar una imagen de cómo construir, administrar y desarrollar microservicios.

Muchas organizaciones han descubierto que al adoptar arquitecturas de microservicios detalladas, pueden entregar software más rápido y adoptar tecnologías más nuevas.

Los microservicios nos brindan mucha más libertad para reaccionar y tomar diferentes decisiones, lo que nos permite responder más rápido al cambio inevitable que nos afecta a todos.

¿Qué son los microservicios?

Los microservicios son servicios pequeños y autónomos que funcionan juntos. Analicemos un poco esa definición y consideremos las características que hacen que los microservicios sean diferentes.

Pequeño y enfocado en hacer una cosa bien

Las bases de código crecen a medida que escribimos código para agregar nuevas funciones. Con el tiempo, puede ser difícil saber dónde se debe realizar un cambio debido a que la base de código es muy grande. A pesar de un impulso por bases de código monolíticas, modulares y claras, con demasiada frecuencia estos límites arbitrarios en el proceso se rompen. El código relacionado con funciones similares comienza a extenderse por todas partes, lo que dificulta la corrección de errores o las implementaciones.

Dentro de un sistema monolítico, luchamos contra estas fuerzas tratando de garantizar que nuestro código sea más cohesivo, a menudo mediante la creación de abstracciones o módulos. La cohesión, el impulso para agrupar el código relacionado, es un concepto importante cuando pensamos en los microservicios. Esto se ve reforzado por la definición de Robert C. Martin del *principio de responsabilidad única*, que establece: "Reúna las cosas que cambian por la misma razón y separe las cosas que cambian por diferentes razones".

Los microservicios adoptan este mismo enfoque para los servicios independientes. Enfocamos los límites de nuestro servicio en los límites comerciales, lo que hace que sea obvio dónde vive el código para una funcionalidad determinada. Y al mantener este servicio enfocado en un límite explícito, evitamos la tentación de que crezca demasiado, con todas las dificultades asociadas que esto puede presentar.

La pregunta que me hacen a menudo es *¿qué tan pequeño es pequeño?* Dar un número para las líneas de código es problemático, ya que algunos lenguajes son más expresivos que otros y, por lo tanto, pueden hacer más en menos líneas de código. También debemos considerar el hecho de que podríamos estar incorporando múltiples dependencias, que a su vez contienen muchas líneas de código. Además, alguna parte de su dominio puede ser legítimamente compleja y requerir más código. Jon Eaves de RealEstate.com.au en Australia caracteriza un microservicio como algo que podría reescribirse en dos semanas, una regla general que tiene sentido para su contexto particular.

Otra respuesta algo trillada que puedo dar es *lo suficientemente pequeña y no más pequeña*. Cuando hablo en conferencias, casi siempre hago la pregunta: *¿quién tiene un sistema que es demasiado grande y que le gustaría desarmar?* Casi todos levantan la mano. Parece que tenemos un muy buen sentido de lo que es demasiado grande, por lo que se podría argumentar que una vez que una pieza de código ya no se *siente* demasiado grande, probablemente sea lo suficientemente pequeña.

Un factor fuerte para ayudarnos a responder *¿qué tan pequeño?* es qué tan bien se alinea el servicio con las estructuras del equipo. Si el código base es demasiado grande para ser administrado por un equipo pequeño, buscar desglosarlo es muy sensato. Hablaremos más sobre la alineación organizacional más adelante.

Cuando se trata de qué tan pequeño es lo suficientemente pequeño, me gusta pensar en estos términos: cuanto más pequeño es el servicio, más se maximizan los beneficios y las desventajas de la arquitectura de microservicio. A medida que se hace más pequeño, aumentan los beneficios en torno a la interdependencia. Pero también lo hace parte de la complejidad que surge de tener más y más partes móviles, algo que exploraremos a lo largo de este libro. A medida que mejora en el manejo de esta complejidad, puede esforzarse por servicios cada vez más pequeños.

Autónomo

Nuestro microservicio es una entidad separada. Puede implementarse como un servicio aislado en una plataforma como servicio (PAAS), o puede ser su propio proceso de sistema operativo. Tratamos de evitar empaquetar múltiples servicios en la misma máquina, ¡aunque la definición de *máquina* en el mundo actual es bastante confusa! Como veremos más adelante, aunque este aislamiento puede agregar algo de sobrecarga, la simplicidad resultante hace que nuestro sistema distribuido sea mucho más fácil de razonar y las tecnologías más nuevas pueden mitigar muchos de los desafíos asociados con esta forma de implementación.

Toda la comunicación entre los propios servicios se realiza a través de llamadas de red, para imponer la separación entre los servicios y evitar los peligros del acoplamiento estrecho.

Estos servicios deben poder cambiar independientemente unos de otros y desplegarse por sí mismos sin que los consumidores tengan que cambiar. Necesitamos pensar en lo que nuestros servicios deben exponer y lo que deben permitir que se oculte. Si se comparte demasiado, nuestros servicios de consumo se acoplan a nuestras representaciones internas. Esto disminuye nuestra autonomía, ya que requiere una coordinación adicional con los consumidores a la hora de realizar cambios.

Nuestro servicio expone una interfaz de programación de aplicaciones (API) y los servicios colaboradores se comunican con nosotros a través de esas API. También debemos pensar en qué tecnología es apropiada para garantizar que esto en sí mismo no empareje a los consumidores. Esto puede significar elegir API independientes de la tecnología para garantizar que no limitemos las opciones tecnológicas. Volveremos una y otra vez a la importancia de buenas API desacopladas a lo largo de este libro.

Sin desacoplamiento, todo se desmorona para nosotros. La regla de oro: ¿puede realizar un cambio en un servicio e implementarlo solo sin cambiar nada más? Si la respuesta es no, entonces muchas de las ventajas que discutimos a lo largo de este libro serán difíciles de lograr.

Para hacer bien el desacoplamiento, deberá modelar sus servicios correctamente y obtener las API correctas. Hablaré mucho de eso.

Beneficios clave

Los beneficios de los microservicios son muchos y variados. Muchos de estos beneficios se pueden poner en la puerta de cualquier sistema distribuido. Los microservicios, sin embargo, tienden a lograr estos beneficios en mayor medida principalmente debido a lo lejos que llevan los conceptos detrás de los sistemas distribuidos y la arquitectura orientada a servicios.

Heterogeneidad tecnológica

Con un sistema compuesto por múltiples servicios que colaboran, podemos decidir usar diferentes tecnologías dentro de cada uno. Esto nos permite elegir la herramienta adecuada para cada trabajo, en lugar de tener que seleccionar un enfoque más estandarizado y único que a menudo termina siendo el mínimo común denominador.

Si una parte de nuestro sistema necesita mejorar su rendimiento, podríamos decidir utilizar una pila de tecnología diferente que sea más capaz de lograr los niveles de rendimiento requeridos.

También podemos decidir que la forma en que almacenamos nuestros datos debe cambiar para diferentes partes de nuestro sistema. Por ejemplo, para una red social, podríamos almacenar las interacciones de nuestros usuarios en una base de datos orientada a gráficos para reflejar la naturaleza altamente interconectada de un gráfico social, pero quizás las publicaciones que hacen los usuarios podrían almacenarse en un almacén de datos orientado a documentos, dando lugar a una arquitectura heterogénea como la que se muestra en la **Figura 1-1**.

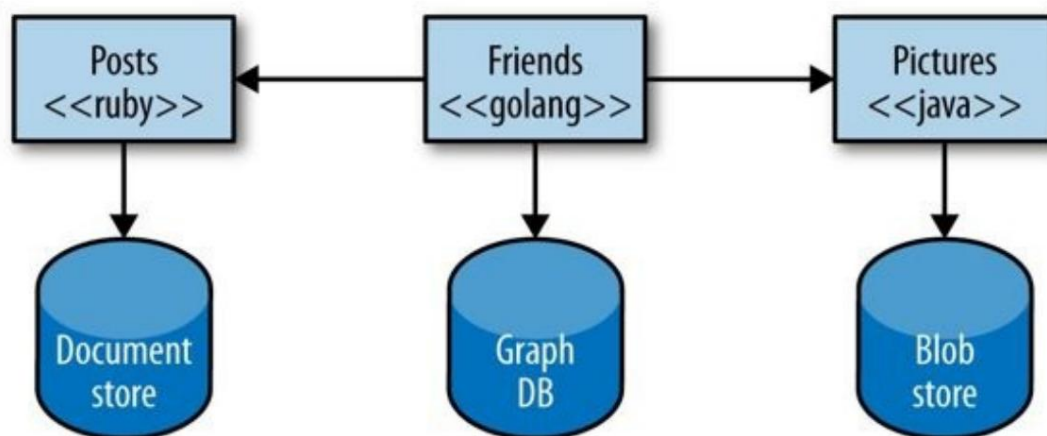


Figura 1-1. Los microservicios pueden permitirle adoptar más fácilmente diferentes tecnologías

Con los microservicios, también podemos adoptar la tecnología más rápidamente y comprender cómo nos pueden ayudar los nuevos avances. Una de las mayores barreras para probar y adoptar nuevas tecnologías son los riesgos asociados con ellas. Con una aplicación monolítica, si quiero probar un nuevo lenguaje de programación, base de datos o marco, cualquier cambio afectará una gran parte de mi sistema. Con un sistema que consta de múltiples servicios, tengo múltiples lugares nuevos en los que probar una nueva tecnología. Puedo elegir un servicio que quizás tenga el riesgo más bajo y usar la tecnología allí, sabiendo que puedo limitar cualquier impacto negativo potencial. Muchas organizaciones consideran que esta capacidad de absorber más rápidamente las nuevas tecnologías es una verdadera ventaja para ellas.

Por supuesto, la adopción de múltiples tecnologías implica gastos generales. Algunas organizaciones optan por imponer algunas restricciones en las opciones de idioma. Netflix y Twitter, por ejemplo, utilizan principalmente Java Virtual Machine (JVM) como plataforma, ya que conocen muy bien la confiabilidad y el rendimiento de ese sistema. También desarrollan bibliotecas y herramientas para la JVM que facilitan mucho el funcionamiento a escala, pero lo hacen más difícil para los servicios o clientes que no están basados en Java. Pero ni Twitter ni Netflix usan solo una pila de tecnología para todos los trabajos. Otro contrapunto a las preocupaciones

acerca de mezclar diferentes tecnologías es el tamaño. Si realmente puedo reescribir mi microservicio en dos semanas, es posible que mitigue los riesgos de adoptar una nueva tecnología.

Como encontrará a lo largo de este libro, al igual que muchas cosas relacionadas con los microservicios, se trata de encontrar el equilibrio adecuado. Discutiremos cómo tomar decisiones tecnológicas en el **Capítulo 2**, que se enfoca en la arquitectura evolutiva; y en el **Capítulo 4**, que trata sobre la integración, aprenderá cómo asegurarse de que sus servicios puedan desarrollar su tecnología independientemente unos de otros sin un acoplamiento indebido.

Resiliencia

Un concepto clave en la ingeniería de resiliencia es el mamparo. Si un componente de un sistema falla, pero esa falla no se produce en cascada, puede aislar el problema y el resto del sistema puede seguir funcionando. Los límites del servicio se convierten en sus mamparos obvios. En un servicio monolítico, si el servicio falla, todo deja de funcionar. Con un sistema monolítico, podemos ejecutar en varias máquinas para reducir la posibilidad de fallas, pero con los microservicios, podemos crear sistemas que manejen la falla total de los servicios y degradar la funcionalidad en consecuencia.

Sin embargo, debemos tener cuidado. Para garantizar que nuestros sistemas de microservicios puedan adoptar adecuadamente esta resiliencia mejorada, debemos comprender las nuevas fuentes de fallas a las que se enfrentan los sistemas distribuidos. Las redes pueden fallar y fallarán, al igual que las máquinas. Necesitamos saber cómo manejar esto y qué impacto (si lo hay) debería tener en el usuario final de nuestro software.

Hablaremos más sobre cómo manejar mejor la resiliencia y cómo manejar los modos de falla en el [Capítulo 11](#).

Escalada

Con un gran servicio monolítico, tenemos que escalar todo junto. Una pequeña parte de nuestro sistema general tiene un rendimiento limitado, pero si ese comportamiento está bloqueado en una aplicación monolítica gigante, tenemos que manejar el escalado de todo como una pieza. Con servicios más pequeños, podemos simplemente escalar aquellos servicios que necesitan escalar, permitiéndonos ejecutar otras partes del sistema en hardware más pequeño y menos poderoso, como en la [Figura 1-2](#).

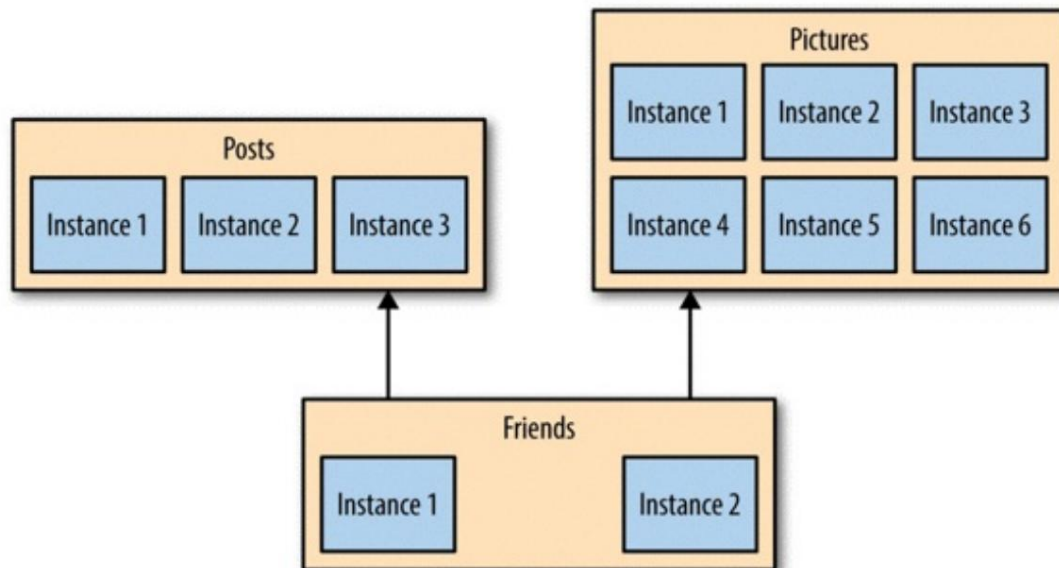


Figura 1-2. Puede orientar el escalado solo a aquellos microservicios que lo necesitan

Gilt, un minorista de moda en línea, adoptó los microservicios precisamente por esta razón. A partir de 2007 con una aplicación Rails monolítica, en 2009 el sistema de Gilt no pudo hacer frente a la carga que se le imponía. Al dividir las partes centrales de su sistema, Gilt pudo lidiar mejor con sus picos de tráfico y hoy tiene más de 450 microservicios, cada uno ejecutándose en múltiples máquinas separadas.

Al adoptar sistemas de aprovisionamiento bajo demanda como los proporcionados por Amazon Web Services, incluso podemos aplicar este escalado bajo demanda para aquellas piezas que lo necesiten. Esto nos permite controlar nuestros costos de manera más efectiva. No es frecuente que un enfoque arquitectónico pueda estar tan estrechamente relacionado con un ahorro de costos casi inmediato.

Facilidad de implementación

Un cambio de una línea a una aplicación monolítica de un millón de líneas requiere que se implemente toda la aplicación para liberar el cambio. Eso podría ser un despliegue de gran impacto y alto riesgo. En la práctica, los despliegues de gran impacto y alto riesgo terminan ocurriendo con poca frecuencia debido a un miedo comprensible. Desafortunadamente, esto significa que nuestros cambios se acumulan y acumulan entre lanzamientos, hasta que la nueva versión de nuestra aplicación que llega a producción tiene una gran cantidad de cambios. ¡Y cuanto mayor sea el delta entre los lanzamientos, mayor será el riesgo de que nos equivoquemos en algo!

Con los microservicios, podemos realizar un cambio en un solo servicio y desplegarlo independientemente del resto del sistema. Esto nos permite implementar nuestro código más rápido. Si ocurre un problema, se puede aislar rápidamente a un servicio individual, lo que facilita la recuperación rápida. También significa que podemos llevar nuestra nueva funcionalidad a los clientes más rápido. Esta es una de las principales razones por las que organizaciones como Amazon y Netflix utilizan estas arquitecturas: para asegurarse de eliminar tantos impedimentos como sea posible para sacar el software a la venta.

La tecnología en este espacio ha cambiado mucho en los últimos años, y profundizaremos en el tema de la implementación en un mundo de microservicios en el [Capítulo 6](#).

Alineación Organizacional

Muchos de nosotros hemos experimentado los problemas asociados con grandes equipos y grandes bases de código. Estos problemas pueden exacerbarse cuando el equipo está distribuido. También sabemos que los equipos más pequeños que trabajan en bases de código más pequeñas tienden a ser más productivos.

Los microservicios nos permiten alinear mejor nuestra arquitectura con nuestra organización, lo que nos ayuda a minimizar la cantidad de personas que trabajan en cualquier base de código para alcanzar el punto óptimo del tamaño y la productividad del equipo. También podemos cambiar la propiedad de los servicios entre los equipos para tratar de mantener a las personas trabajando en un servicio en el mismo lugar. Entraremos en más detalles sobre este tema cuando discutamos la ley de Conway en el [Capítulo 10](#).

componibilidad

Una de las promesas clave de los sistemas distribuidos y las arquitecturas orientadas a servicios es que abrimos oportunidades para la reutilización de la funcionalidad. Con los microservicios, permitimos que nuestra funcionalidad se consuma de diferentes maneras para diferentes propósitos. Esto puede ser especialmente importante cuando pensamos en cómo nuestros consumidores usan nuestro software. Atrás quedó el tiempo en el que podíamos pensar estrechamente en nuestro sitio web de escritorio o en nuestra aplicación móvil. Ahora tenemos que pensar en las innumerables formas en que podríamos querer unir capacidades para la web, la aplicación nativa, la web móvil, la aplicación para tabletas o el dispositivo portátil. A medida que las organizaciones pasan de pensar en términos de canales estrechos a conceptos más holísticos de participación del cliente, necesitamos arquitecturas que puedan mantenerse al día.

Con los microservicios, piense en nosotros abriendo costuras en nuestro sistema que son abordables por terceros. A medida que cambian las circunstancias, podemos construir cosas de diferentes maneras. Con una aplicación monolítica, a menudo tengo una costura de grano grueso que se puede usar desde el exterior. Si quiero romper eso para obtener algo más útil, ¡necesitaré un martillo! En el [Capítulo 5](#), analizaré las formas en que puede dividir los sistemas monolíticos existentes y, con suerte, convertirlos en algunos microservicios reutilizables y reconstituibles.

Optimización para la reemplazabilidad

Si trabaja en una organización de tamaño mediano o grande, es probable que esté al tanto de algún sistema heredado grande y desagradable que se encuentra en la esquina. El que nadie quiere tocar. El que es vital para el funcionamiento de su empresa, pero que está escrito en alguna variante extraña de Fortran y se ejecuta solo en hardware que llegó al final de su vida útil hace 25 años. ¿Por qué no ha sido reemplazado? Ya sabes por qué: es un trabajo demasiado grande y arriesgado.

Dado que nuestros servicios individuales son de tamaño pequeño, el costo de reemplazarlos con una mejor implementación, o incluso eliminarlos por completo, es mucho más fácil de administrar. ¿Con qué frecuencia ha eliminado más de cien líneas de código en un solo día y no se ha preocupado demasiado por ello? Dado que los microservicios suelen tener un tamaño similar, las barreras para reescribir o eliminar servicios por completo son muy bajas.

Los equipos que utilizan enfoques de microservicios se sienten cómodos con la reescritura completa de los servicios cuando es necesario y simplemente eliminando un servicio cuando ya no es necesario. Cuando un código base tiene solo unos pocos cientos de líneas, es difícil que las personas se apeguen emocionalmente a él, y el costo de reemplazarlo es bastante pequeño.

¿Qué pasa con la arquitectura orientada a servicios?

La arquitectura orientada a servicios (SOA) es un enfoque de diseño en el que múltiples servicios colaboran para proporcionar un conjunto final de capacidades. Un servicio aquí generalmente significa un proceso de sistema operativo completamente separado. La comunicación entre estos servicios se produce a través de llamadas a través de una red en lugar de llamadas de método dentro de un límite de proceso.

SOA surgió como un enfoque para combatir los desafíos de las grandes aplicaciones monolíticas. Es un enfoque que tiene como objetivo promover la reutilización del software; dos o más aplicaciones de usuario final, por ejemplo, podrían usar los mismos servicios. Su objetivo es facilitar el mantenimiento o la reescritura del software, ya que, en teoría, podemos reemplazar un servicio por otro sin que nadie lo sepa, siempre que la semántica del servicio no cambie demasiado.

SOA en su esencia es una idea muy sensata. Sin embargo, a pesar de muchos esfuerzos, falta un buen consenso sobre cómo hacer SOA *bien*. En mi opinión, gran parte de la industria no ha logrado analizar el problema de manera lo suficientemente holística y presentar una alternativa convincente a la narrativa establecida por varios proveedores en este espacio.

Muchos de los problemas planteados en la puerta de SOA son en realidad problemas con cosas como los protocolos de comunicación (por ejemplo, SOAP), el software intermedio del proveedor, la falta de orientación sobre la granularidad del servicio o la orientación incorrecta sobre la elección de lugares para dividir su sistema. Abordaremos cada uno de estos a su vez a lo largo del resto del libro. Un cínico podría sugerir que los proveedores cooptaron (y en algunos casos impulsaron) el movimiento SOA como una forma de vender más productos, y esos mismos productos al final socavaron el objetivo de SOA.

Gran parte de la sabiduría convencional sobre SOA no ayuda a entender cómo dividir algo grande en algo pequeño. No habla de cuán grande es demasiado grande. No habla lo suficiente sobre formas prácticas del mundo real para garantizar que los servicios no se acoplen demasiado. La cantidad de cosas que no se dicen es donde se originan muchas de las trampas asociadas con SOA.

El enfoque de microservicio ha surgido del uso en el mundo real, tomando nuestra mejor comprensión de los sistemas y la arquitectura para hacer SOA bien. Por lo tanto, debería pensar en los microservicios como un enfoque específico para SOA de la misma manera que XP o Scrum son enfoques específicos para el desarrollo de software Agile.

Otras técnicas de descomposición

Cuando se pone a ello, muchas de las ventajas de una arquitectura basada en microservicios provienen de su naturaleza granular y del hecho de que le brinda muchas más opciones sobre cómo resolver problemas. Pero, ¿podrían técnicas de descomposición similares lograr los mismos beneficios?

Bibliotecas compartidas

Una técnica de descomposición muy estándar que está integrada en prácticamente cualquier lenguaje es dividir una base de código en varias bibliotecas. Estas bibliotecas pueden ser proporcionadas por terceros o creadas en su propia organización.

Las bibliotecas le brindan una forma de compartir funcionalidades entre equipos y servicios. Podría crear un conjunto de utilidades de recopilación útiles, por ejemplo, o tal vez una biblioteca de estadísticas que se pueda reutilizar.

Los equipos pueden organizarse en torno a estas bibliotecas y las propias bibliotecas pueden reutilizarse. Pero hay algunos inconvenientes.

Primero, se pierde la verdadera heterogeneidad tecnológica. Por lo general, la biblioteca debe estar en el mismo idioma o, al menos, ejecutarse en la misma plataforma. En segundo lugar, se reduce la facilidad con la que puede escalar partes de su sistema independientemente unas de otras. A continuación, a menos que esté utilizando bibliotecas vinculadas dinámicamente, no puede implementar una nueva biblioteca sin volver a implementar todo el proceso, por lo que se reduce su capacidad para implementar cambios de forma aislada.

Y quizás el truco sea que carece de las costuras obvias alrededor de las cuales erigir medidas arquitectónicas de seguridad para garantizar la capacidad de recuperación del sistema.

Las bibliotecas compartidas tienen su lugar. Se encontrará creando código para tareas comunes que no son específicas de su dominio comercial y que desea reutilizar en toda la organización, que es un candidato obvio para convertirse en una biblioteca reutilizable. Sin embargo, debes tener cuidado. El código compartido utilizado para comunicarse entre servicios puede convertirse en un punto de acoplamiento, algo que discutiremos en el **Capítulo 4**.

Los servicios pueden y deben hacer un uso intensivo de bibliotecas de terceros para reutilizar código común. Pero no nos llevan hasta allí.

Módulos

Algunos lenguajes proporcionan sus propias técnicas de descomposición modular que van más allá de las simples bibliotecas. Permiten cierta gestión del ciclo de vida de los módulos, de modo que se pueden implementar en un proceso en ejecución, lo que le permite realizar cambios sin desmantelar todo el proceso.

Vale la pena mencionar la Open Source Gateway Initiative (OSGI) como un enfoque específico de tecnología para la descomposición modular. Java en sí mismo no tiene un verdadero concepto de módulos, y tendremos que esperar al menos hasta Java 9 para ver esto agregado al lenguaje.

OSGI, que surgió como un marco para permitir la instalación de complementos en el IDE de Java de Eclipse, ahora se usa como una forma de adaptar un concepto de módulo en Java a través de una biblioteca.

El problema con OSGI es que está tratando de hacer cumplir cosas como la gestión del ciclo de vida del módulo sin suficiente soporte en el lenguaje en sí. Esto da como resultado que los autores de los módulos deban realizar más trabajo para lograr un aislamiento adecuado del módulo. Dentro de los límites de un proceso, también es mucho más fácil caer en la trampa de hacer que los módulos estén excesivamente acoplados entre sí, causando todo tipo de problemas. Mi propia experiencia con OSGI, que coincide con la de colegas de la industria, es que, incluso con buenos equipos, es fácil que OSGI se convierta en una fuente de complejidad mucho mayor de lo que justifican sus beneficios.

Erlang sigue un enfoque diferente, en el que los módulos se integran en el tiempo de ejecución del lenguaje. Por lo tanto, Erlang es un enfoque muy maduro para la descomposición modular. Los módulos de Erlang se pueden detener, reiniciar y actualizar sin problemas. Erlang incluso admite la ejecución de más de una versión del módulo en un momento dado, lo que permite una actualización del módulo más elegante.

Las capacidades de los módulos de Erlang son realmente impresionantes, pero incluso si tenemos la suerte de usar una plataforma con estas capacidades, todavía tenemos las mismas deficiencias que tenemos con las bibliotecas compartidas normales. Estamos estrictamente limitados en nuestra capacidad para usar nuevas tecnologías, limitados en la forma en que podemos escalar de forma independiente, podemos derivar hacia técnicas de integración que se acoplan demasiado y carecen de costuras para las medidas de seguridad arquitectónica.

Hay una observación final que vale la pena compartir. Técnicamente, debería ser posible crear módulos independientes bien factorizados dentro de un único proceso monolítico. Y, sin embargo, rara vez vemos que esto suceda. Los propios módulos pronto se acoplan estrechamente con el resto del código, entregando uno de sus beneficios clave. Tener una separación de límites de proceso impone una higiene limpia a este respecto (¡o al menos hace que sea más difícil hacer algo incorrecto!). No sugeriría que este debería ser el principal impulsor de la separación de procesos, por supuesto, pero es interesante que las promesas de separación modular dentro de los límites de los procesos rara vez se cumplen en el mundo real.

Entonces, si bien la descomposición modular dentro de los límites de un proceso puede ser algo que desee hacer además de descomponer su sistema en servicios, por sí solo no ayudará a resolverlo todo. Si usted es una tienda de Erlang puro, la calidad de la implementación del módulo de Erlang

puede llegar muy lejos, pero sospecho que muchos de ustedes no están en esa situación. Para el resto de nosotros, deberíamos considerar que los módulos ofrecen el mismo tipo de beneficios que las bibliotecas compartidas.

Sin bala de plata

Antes de terminar, debo señalar que los microservicios no son un almuerzo gratis ni una bala de plata, y son una mala elección como un martillo de oro. Tienen todas las complejidades asociadas de los sistemas distribuidos, y aunque hemos aprendido mucho sobre cómo administrar bien los sistemas distribuidos (lo cual discutiremos a lo largo del libro), todavía es difícil. Si viene desde el punto de vista de un sistema monolítico, tendrá que mejorar mucho en el manejo de la implementación, las pruebas y la supervisión para desbloquear los beneficios que hemos cubierto hasta ahora. También deberá pensar de manera diferente acerca de cómo escalar sus sistemas y asegurarse de que sean resistentes. ¡Tampoco se sorprenda si cosas como transacciones distribuidas o el teorema CAP comienzan a causarle dolores de cabeza!

Cada empresa, organización y sistema es diferente. Una serie de factores influirán en si los microservicios son adecuados para usted o no, y qué tan agresivo puede ser al adoptarlos. A lo largo de cada capítulo de este libro, intentaré orientarlo destacando los peligros potenciales, lo que debería ayudarlo a trazar un camino firme.

Resumen

Con suerte, ya sabrá qué es un microservicio, qué lo hace diferente de otras técnicas de composición y cuáles son algunas de las ventajas clave. En cada uno de los siguientes capítulos entraremos en más detalles sobre cómo lograr estos beneficios y cómo evitar algunos de los errores comunes.

Hay una serie de temas que cubrir, pero tenemos que empezar por alguna parte. Uno de los principales desafíos que presentan los microservicios es un cambio en el rol de quienes a menudo guían la evolución de nuestros sistemas: los arquitectos. A continuación, veremos algunos enfoques diferentes para este rol que pueden garantizar que aprovechemos al máximo esta nueva arquitectura.

Capítulo 2. El arquitecto evolutivo

Como hemos visto hasta ahora, los microservicios nos brindan muchas opciones y, en consecuencia, muchas decisiones que tomar. Por ejemplo, ¿cuántas tecnologías diferentes deberíamos usar, deberíamos permitir que diferentes equipos usen diferentes lenguajes de programación y deberíamos dividir o fusionar un servicio? ¿Cómo hacemos para tomar estas decisiones? Con el ritmo de cambio más rápido y el entorno más fluido que permiten estas arquitecturas, el papel del arquitecto también tiene que cambiar. En este capítulo, adoptaré una visión bastante obstinada de cuál es el papel de un arquitecto y, con suerte, lanzaré un asalto final a la torre de marfil.

Comparaciones inexactas

Usted sigue usando esa palabra. No creo que signifique lo que crees que significa.

Íñigo Montoya, de *La princesa prometida*

Los arquitectos tienen un trabajo importante. Ellos están a cargo de asegurarse de que tengamos una visión técnica conjunta, una que debería ayudarnos a entregar el sistema que nuestros clientes necesitan. En algunos lugares, es posible que solo tengan que trabajar con un equipo, en cuyo caso el papel del arquitecto y del líder técnico suele ser el mismo. En otros, pueden estar definiendo la visión de un programa de trabajo completo, coordinando con múltiples equipos en todo el mundo, o tal vez incluso una organización completa.

Independientemente del nivel en el que operen, el rol es difícil de precisar y, a pesar de que a menudo es la progresión profesional obvia para los desarrolladores en las organizaciones empresariales, también es un rol que recibe más críticas que prácticamente cualquier otro.

Más que cualquier otro rol, los arquitectos pueden tener un impacto directo en la calidad de los sistemas construidos, en las condiciones de trabajo de sus colegas y en la capacidad de su organización para responder al cambio y, sin embargo, con tanta frecuencia parece que nos equivocamos en este rol. ¿Porqué es eso?

Nuestra industria es joven. Esto es algo que parece que olvidamos y, sin embargo, solo hemos estado creando programas que se ejecutan en lo que reconocemos como computadoras durante unos 70 años.

Por lo tanto, estamos constantemente mirando a otras profesiones en un intento de explicar lo que hacemos. No somos médicos ni ingenieros, pero tampoco plomeros ni electricistas.

En cambio, caemos en un punto medio, lo que dificulta que la sociedad nos entienda, o que entendamos dónde encajamos.

Así que tomamos prestado de otras profesiones. Nos llamamos "ingenieros" o "arquitectos" de software.

Pero no lo somos, ¿verdad? Los arquitectos e ingenieros tienen un rigor y una disciplina con los que solo podíamos soñar, y su importancia en la sociedad es bien conocida. Recuerdo haber hablado con un amigo mío, el día antes de que se convirtiera en arquitecto calificado. "Mañana", dijo, "si te doy un consejo en el pub sobre cómo construir algo y está mal, tendré que rendir cuentas. Podrían demandarme, ya que a los ojos de la ley ahora soy un arquitecto calificado y debería ser responsable si me equivoco". La importancia de estos puestos de trabajo para la sociedad significa que hay requisitos que las personas deben cumplir. En el Reino Unido, por ejemplo, se requiere un mínimo de siete años de estudio antes de que pueda ser llamado arquitecto. Pero estos trabajos también se basan en un conjunto de conocimientos que se remonta a miles de años. ¿Y nosotros? No exactamente. Por eso también considero que la mayoría de las formas de certificación de TI son inútiles, ya que sabemos muy poco acerca de lo que es *bueno*.

Una parte de nosotros quiere reconocimiento, por lo que tomamos prestados nombres de otras profesiones que ya tienen el reconocimiento que anhelamos como industria. Pero esto puede ser doblemente dañino. Primero, implica que sabemos lo que estamos haciendo, cuando claramente no lo sabemos. No diría que los edificios y los puentes nunca se caen, pero se caen mucho menos que la cantidad de veces que fallan nuestros programas, lo que hace que las comparaciones con los ingenieros sean bastante injustas. En segundo lugar, las analogías se desmoronan muy rápidamente cuando se les da incluso una mirada superficial. para cambiar las cosas

alrededor, si la construcción de puentes fuera como la programación, a mitad de camino nos daríamos cuenta de que la orilla opuesta ahora estaba 50 metros más lejos, que en realidad era barro en lugar de granito, y que en lugar de construir un puente peatonal, estábamos construyendo un puente de carretera . Nuestro software no está limitado por las mismas reglas físicas con las que tienen que lidiar los arquitectos o ingenieros reales, y lo que creamos está diseñado para adaptarse y evolucionar con los requisitos del usuario.

Quizás el término *arquitecto* ha hecho más daño. La idea de alguien que elabora planes detallados para que otros los interpreten y espera que esto se lleve a cabo. El equilibrio de parte artista, parte ingeniero, supervisando la creación de lo que normalmente es una visión singular, con todos los demás puntos de vista subordinados, excepto por la objeción ocasional del ingeniero estructural con respecto a las leyes de la física. En nuestra industria, esta visión del arquitecto lleva a algunas prácticas terribles. Diagrama tras diagrama, página tras página de documentación, creada con el fin de informar la construcción del sistema perfecto, sin tener en cuenta el futuro fundamentalmente incognoscible. Completamente desprovisto de cualquier comprensión de lo difícil que será implementarlo, o si realmente funcionará o no, y mucho menos tener la capacidad de cambiar a medida que aprendemos más.

Cuando nos comparamos con ingenieros o arquitectos, corremos el peligro de perjudicar a todos.

Desafortunadamente, estamos atascados con la palabra *arquitecto* por ahora. Así que lo mejor que podemos hacer es redefinir lo que significa en nuestro contexto.

Una visión evolutiva para el arquitecto

Nuestros requisitos cambian más rápidamente que para las personas que diseñan y construyen edificios, al igual que las herramientas y técnicas a nuestra disposición. Las cosas que creamos no son puntos fijos en el tiempo. Una vez lanzado a producción, nuestro software continuará evolucionando a medida que cambie la forma en que se usa. Para la mayoría de las cosas que creamos, debemos aceptar que una vez que el software llegue a manos de nuestros clientes, tendremos que reaccionar y adaptarnos, en lugar de que sea un artefacto que nunca cambia. Por lo tanto, nuestros arquitectos deben dejar de pensar en crear el producto final perfecto y, en cambio, centrarse en ayudar a crear un marco en el que puedan surgir los sistemas correctos y seguir creciendo a medida que aprendemos más.

Aunque he pasado gran parte del capítulo hasta ahora advirtiéndole que no nos comparemos demasiado con otras profesiones, hay una analogía que me gusta cuando se trata del rol del arquitecto de TI y que creo que resume mejor lo que queremos que este rol sea. Erik Doernenburg primero compartió conmigo la idea de que deberíamos pensar en nuestro papel más como urbanistas que como arquitectos del entorno construido. El papel del urbanista debería ser familiar para cualquiera de ustedes que haya jugado antes a SimCity. El papel de un urbanista es mirar una multitud de fuentes de información y luego intentar optimizar el diseño de una ciudad para que se adapte mejor a las necesidades de los ciudadanos de hoy, teniendo en cuenta el uso futuro. Sin embargo, la forma en que influye en la evolución de la ciudad es interesante. Él no dice, "construye este edificio específico allí"; en cambio, divide *en zonas una ciudad*. Entonces, como en SimCity, puede designar parte de su ciudad como zona industrial y otra parte como zona residencial. Luego, depende de otras personas decidir qué edificios exactos se crean, pero existen restricciones: si desea construir una fábrica, deberá estar en una zona industrial. En lugar de preocuparse demasiado por lo que sucede en una zona, el urbanista dedicará mucho más tiempo a determinar cómo se mueven las personas y los servicios públicos de una zona a otra.

Más de uno ha comparado una ciudad con un ser vivo. La ciudad cambia con el tiempo. Cambia y evoluciona a medida que sus ocupantes lo usan de diferentes maneras, o según lo moldean fuerzas externas.

El urbanista hace todo lo posible por anticiparse a estos cambios, pero acepta que es inútil tratar de ejercer un control directo sobre todos los aspectos de lo que sucede.

La comparación con el software debería ser obvia. A medida que nuestros usuarios usan nuestro software, debemos reaccionar y cambiar. No podemos prever todo lo que sucederá y, por lo tanto, en lugar de planificar cualquier eventualidad, debemos planificar para permitir el cambio evitando la necesidad de especificar en exceso hasta el último detalle. Nuestra ciudad, el sistema, debe ser un lugar bueno y feliz para todos los que lo usan. Una cosa que la gente suele olvidar es que nuestro sistema no solo se adapta a los usuarios; también acomoda a los desarrolladores y al personal de operaciones que también tienen que trabajar allí y que tienen el trabajo de asegurarse de que pueda cambiar según sea necesario. Tomando prestado un término de Frank Buschmann, los arquitectos tienen el deber de garantizar que el sistema sea *habitable* también para los desarrolladores.

Un urbanista, al igual que un arquitecto, también necesita saber cuándo su plan no está siendo

seguido. Como es menos prescriptivo, la cantidad de veces que debe involucrarse para corregir la dirección debe ser mínima, pero si alguien decide construir una planta de tratamiento de aguas residuales en un área residencial, debe poder cerrarla.

Por lo tanto, nuestros arquitectos, como urbanistas, deben establecer la dirección a grandes rasgos y solo involucrarse en ser muy específicos sobre los detalles de implementación en casos limitados. Deben asegurarse de que el sistema sea adecuado para su propósito ahora, pero también una plataforma para el futuro. Y deben asegurarse de que sea un sistema que haga felices a los usuarios y desarrolladores por igual. Esto suena como una orden bastante alta. ¿Donde empezamos?

Zonificación

Entonces, para continuar por un momento con la metáfora del arquitecto como urbanista, ¿cuáles son nuestras zonas? Estos son nuestros límites de servicio, o quizás grupos de servicios de grano grueso. Como arquitectos, debemos preocuparnos mucho menos por lo que sucede *dentro* de la zona que por lo que sucede *entre* las zonas. Eso significa que debemos dedicar tiempo a pensar en cómo nuestros servicios se comunican entre sí o asegurarnos de que podemos monitorear adecuadamente la salud general de nuestro sistema. La forma en que nos involucremos dentro de la zona variará un poco. Muchas organizaciones han adoptado microservicios para maximizar la autonomía de los equipos, algo que ampliaremos en el **Capítulo 10**. Si está en una organización de este tipo, confiará más en el equipo para tomar la decisión local correcta.

Pero entre las zonas, o las cajas en nuestro diagrama de arquitectura tradicional, debemos tener cuidado; hacer las cosas mal aquí conduce a todo tipo de problemas y puede ser muy difícil de solucionar. correcto.

Dentro de cada servicio, puede estar de acuerdo con que el equipo propietario de esa zona elija una pila de tecnología o almacén de datos diferente. Otras preocupaciones pueden surgir aquí, por supuesto. Su inclinación a dejar que los equipos elijan la herramienta adecuada para el trabajo puede verse atenuada por el hecho de que se vuelve más difícil contratar personas o moverlas entre equipos si tiene 10 pilas de tecnología diferentes para respaldar. Del mismo modo, si cada equipo elige un almacén de datos completamente diferente, es posible que le falte la experiencia suficiente para ejecutar cualquiera de ellos a escala.

Netflix, por ejemplo, se ha estandarizado principalmente en Cassandra como tecnología de almacenamiento de datos. Si bien es posible que no sea la mejor opción para todos sus casos, Netflix considera que el valor que se obtiene al crear herramientas y experiencia en torno a Cassandra es más importante que tener que respaldar y operar a escala muchas otras plataformas que pueden ser más adecuadas para ciertas tareas. .

Netflix es un ejemplo extremo, donde la escala es probablemente el factor primordial más fuerte, pero entiendes la idea.

Sin embargo, entre los servicios es donde las cosas pueden complicarse. Si un servicio decide exponer REST a través de HTTP, otro utiliza búferes de protocolo y un tercero utiliza RMI de Java, la integración puede convertirse en una pesadilla, ya que los servicios de consumo deben comprender y admitir múltiples estilos de intercambio. Es por eso que trato de apegarme a la directriz de que debemos “preocuparnos por lo que sucede entre las cajas y ser liberales con lo que sucede dentro”.

EL ARQUITECTO CODIFICADOR

Si queremos asegurarnos de que los sistemas que creamos sean habitables para nuestros desarrolladores, nuestros arquitectos deben comprender el impacto de sus decisiones. Como mínimo, esto significa pasar tiempo con el equipo, e idealmente debería significar que estos desarrolladores también pasan tiempo programando con el equipo. Para aquellos de ustedes que practican la programación en pareja, se convierte en un asunto simple para un arquitecto unirse a un equipo por un período corto como miembro de la pareja. Idealmente, deberías trabajar en historias normales, para entender realmente cómo es el trabajo *normal*. ¡No puedo enfatizar lo importante que es para el arquitecto sentarse con el equipo! Esto es significativamente más efectivo que tener una llamada o simplemente mirar su código.

En cuanto a la frecuencia con la que debe hacer esto, depende en gran medida del tamaño del equipo con el que esté trabajando. Pero la clave es que debe ser una actividad rutinaria. Si está trabajando con cuatro equipos, por ejemplo, pasar medio día con cada equipo cada cuatro semanas le asegura crear conciencia y mejorar las comunicaciones con los equipos con los que está trabajando.

Un enfoque basado en principios

Las reglas son para la obediencia de los necios y la guía de los sabios.

Generalmente atribuido a Douglas Bader

¡Tomar decisiones en el diseño del sistema tiene que ver con las compensaciones, y las arquitecturas de microservicios nos brindan muchas compensaciones para hacer! Al elegir un almacén de datos, ¿elegimos una plataforma con la que tenemos menos experiencia, pero que nos brinda una mejor escalabilidad? ¿Está bien que tengamos dos pilas de tecnología diferentes en nuestro sistema? ¿Qué hay de tres? Algunas decisiones se pueden tomar completamente en el acto con la información disponible, y estas son las más fáciles de tomar.

Pero, ¿qué pasa con aquellas decisiones que podrían tener que tomarse con información incompleta?

Enmarcar aquí puede ayudar, y una excelente manera de ayudar a enmarcar nuestra toma de decisiones es definir un conjunto de principios y prácticas que la guíen, en función de los objetivos que estamos tratando de lograr.

Echemos un vistazo a cada uno a su vez.

Metas estratégicas

El papel del arquitecto ya es lo suficientemente desalentador, ¡así que afortunadamente, por lo general, no tenemos que definir también objetivos estratégicos! Los objetivos estratégicos deben hablar de hacia dónde se dirige su empresa y cómo se ve a sí misma como la mejor manera de hacer felices a sus clientes. Estos serán objetivos de alto nivel y es posible que no incluyan tecnología en absoluto. Podrían definirse a nivel de empresa o de división. Pueden ser cosas como "Expandirse en el sudeste asiático para desbloquear nuevos mercados" o "Permitir que el cliente logre todo lo posible mediante el autoservicio". La clave es que ahí es donde se dirige su organización, por lo que debe asegurarse de que la tecnología esté alineada con ella.

Si usted es la persona que define la visión técnica de la empresa, esto puede significar que necesitará pasar más tiempo con las partes no técnicas de su organización (o *del negocio*, como se les suele llamar). ¿Cuál es la visión impulsora del negocio? ¿Y cómo cambia?

Principios

Los principios son reglas que ha creado para alinear lo que está haciendo con un objetivo más amplio y, a veces, cambiarán. Por ejemplo, si uno de sus objetivos estratégicos como organización es reducir el tiempo de comercialización de nuevas funciones, puede definir un principio que diga que los equipos de entrega tienen control total sobre el ciclo de vida de su software para enviarlo cuando esté listo, de forma independiente. de cualquier otro equipo. Si otro objetivo es que su organización se esté moviendo para aumentar agresivamente su oferta en otros países, puede decidir implementar el principio de que todo el sistema debe ser portátil para permitir su implementación local a fin de respetar la soberanía de los datos.

Probablemente no quieras muchos de estos. Menos de 10 es un buen número, lo suficientemente pequeño como para que la gente pueda recordarlo o para que quepa en carteles pequeños. Cuantos más principios tenga, mayor será la posibilidad de que se superpongan o se contradigan entre sí.

Los **12 factores** de Heroku son un conjunto de principios de diseño estructurados en torno al objetivo de ayudarlo a crear aplicaciones que funcionen bien en la plataforma Heroku. También pueden tener sentido en otros contextos. Algunos de los principios son en realidad restricciones basadas en comportamientos que su aplicación debe mostrar para funcionar en Heroku. Una restricción es realmente algo que es muy difícil (o virtualmente imposible) de cambiar, mientras que los principios son cosas que decidimos elegir. Puede decidir mencionar explícitamente aquellas cosas que son principios versus aquellas que son restricciones, para ayudar a indicar aquellas cosas que realmente no puede cambiar. Personalmente, creo que puede ser valioso mantenerlos en la misma lista para fomentar restricciones desafiantes de vez en cuando y ver si realmente son inamovibles.

Prácticas

Nuestras prácticas son la forma en que nos aseguramos de que nuestros principios se lleven a cabo. Son un conjunto de guías detalladas y prácticas para realizar tareas. A menudo serán específicos de la tecnología y deben ser de un nivel lo suficientemente bajo como para que cualquier desarrollador pueda entenderlos. Las prácticas podrían incluir pautas de codificación, el hecho de que todos los datos de registro deben capturarse de forma centralizada o que HTTP/REST es el estilo de integración estándar. Debido a su naturaleza técnica, las prácticas cambiarán con más frecuencia que los principios.

Al igual que con los principios, a veces las prácticas reflejan limitaciones en su organización. Por ejemplo, si solo admite CentOS, esto deberá reflejarse en sus prácticas.

Las prácticas deben sustentar nuestros principios. Un principio que establece que los equipos de entrega controlan el ciclo de vida completo de sus sistemas puede significar que tiene una práctica que establece que todos los servicios se implementan en cuentas de AWS aisladas, lo que brinda administración de autoservicio de los recursos y aislamiento de otros equipos.

Combinación de principios y prácticas

Los principios de una persona son las prácticas de otra. Puede decidir llamar al uso de HTTP/REST un principio en lugar de una práctica, por ejemplo. Y eso estaría bien. El punto clave es que es valioso tener ideas generales que guíen cómo evoluciona el sistema y tener suficientes detalles para que las personas sepan cómo implementar esas ideas.

Para un grupo lo suficientemente pequeño, tal vez un solo equipo, combinar principios y prácticas podría estar bien. Sin embargo, para organizaciones más grandes, donde la tecnología y las prácticas de trabajo pueden diferir, es posible que desee un conjunto diferente de prácticas en diferentes lugares, siempre que ambos se correspondan con un conjunto común de principios. Un equipo de .NET, por ejemplo, puede tener un conjunto de prácticas y un equipo de Java otro, con un conjunto de prácticas comunes a ambos. Los principios, sin embargo, podrían ser los mismos para ambos.

Un ejemplo del mundo real

Mi colega Evan Bottcher desarrolló el diagrama que se muestra en la [Figura 2-1](#) mientras trabajaba con uno de nuestros clientes. La figura muestra la interacción de objetivos, principios y prácticas en un formato muy claro. En el transcurso de un par de años, las prácticas de la extrema derecha cambiarán con bastante regularidad, mientras que los principios permanecerán bastante estáticos. Un diagrama como este puede imprimirse muy bien en una sola hoja de papel y compartirse, y cada idea es lo suficientemente simple para que el desarrollador promedio la recuerde. Por supuesto, hay más detalles detrás de cada punto aquí, pero poder articular esto en forma resumida es muy útil.

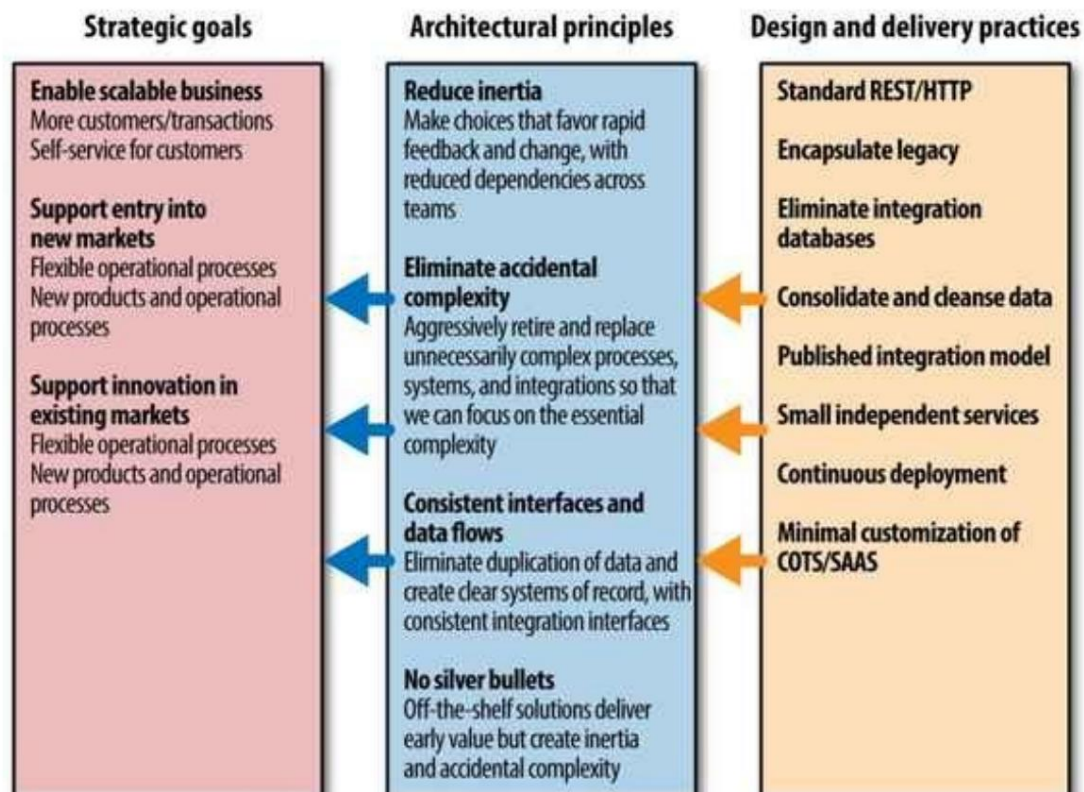


Figura 2-1. Un ejemplo del mundo real de principios y prácticas

Tiene sentido tener documentación que respalde algunos de estos elementos. En general, sin embargo, me gusta la idea de tener un código de ejemplo que pueda mirar, inspeccionar y ejecutar, que encarna estas ideas. Aún mejor, podemos crear herramientas que hacen lo correcto desde el primer momento. Discutiremos eso con más profundidad en un momento.

El estándar requerido

Cuando esté trabajando en sus prácticas y pensando en las compensaciones que necesita hacer, uno de los equilibrios centrales que debe encontrar es cuánta variabilidad debe permitir en su sistema. Una de las formas clave de identificar lo que debe ser constante de un servicio a otro es definir cómo se ve un buen servicio. ¿Qué es un servicio de "buen ciudadano" en su sistema? ¿Qué capacidades debe tener para garantizar que su sistema sea manejable y que un mal servicio no derrumbe todo el sistema? Y, al igual que con las personas, lo que es un buen ciudadano en un contexto no refleja lo que parece en otro lugar. No obstante, hay algunas características comunes de los servicios de buen comportamiento que creo que es bastante importante observar. Estas son las pocas áreas clave en las que permitir demasiada divergencia puede resultar en un momento bastante tórrido. Como dice Ben Christensen de Netflix, cuando pensamos en el panorama general, "debe ser un sistema cohesivo hecho de muchas partes pequeñas con ciclos de vida autónomos pero que se unen". Por lo tanto, debemos encontrar el equilibrio entre optimizar la autonomía del microservicio individual sin perder de vista el panorama general. Definir atributos claros que debe tener cada servicio es una forma de tener claro dónde se encuentra ese equilibrio.

Vigilancia

Es esencial que podamos elaborar visiones coherentes y transversales de la salud de nuestro sistema. Esta tiene que ser una vista de todo el sistema, no una vista específica del servicio. Como discutiremos en el **Capítulo 8**, conocer la salud de un servicio individual es útil, pero a menudo solo cuando intenta diagnosticar un problema más amplio o comprender una tendencia más amplia. Para que esto sea lo más fácil posible, sugeriría asegurarse de que todos los servicios emitan métricas relacionadas con el estado y el monitoreo general de la misma manera.

Puede optar por adoptar un mecanismo de envío, en el que cada servicio necesita enviar estos datos a una ubicación central. Para sus métricas, esto podría ser Graphite, y para su salud, podría ser Nagios. O puede decidir utilizar sistemas de sondeo que extraen datos de los propios nodos. Pero elijas lo que elijas, trata de mantenerlo estandarizado. Haga que la tecnología dentro de la caja sea opaca y no requiera que sus sistemas de monitoreo cambien para admitirla. El registro cae en la misma categoría aquí: lo necesitamos en un solo lugar.

Interfaces

Elegir un pequeño número de tecnologías de interfaz definidas ayuda a integrar nuevos consumidores. Tener un estándar es un buen número. Dos tampoco está tan mal. Tener 20 estilos diferentes de integración es malo. No se trata solo de elegir la tecnología y el protocolo. Si elige HTTP/REST, por ejemplo, ¿usará verbos o sustantivos? ¿Cómo manejará la paginación de los recursos? ¿Cómo manejará el control de versiones de los puntos finales?

Seguridad Arquitectónica

No podemos darnos el lujo de que un servicio mal educado arruine la fiesta para todos. Tenemos que asegurarnos de que nuestros servicios se protejan en consecuencia de las llamadas descendentes no saludables. Cuantos más servicios tengamos que no manejen adecuadamente la posible falla de las llamadas descendentes, más frágiles serán nuestros sistemas. Esto significa que probablemente querrá exigir como mínimo que cada servicio descendente tenga su propio conjunto de conexiones, e incluso puede llegar a decir que cada uno también usa un disyuntor. Esto se cubrirá con más profundidad cuando analicemos los microservicios a escala en el [Capítulo 11](#).

Jugar según las reglas también es importante cuando se trata de códigos de respuesta. Si sus disyuntores se basan en códigos HTTP y un servicio decide devolver los códigos 2XX por errores, o confunde los códigos 4XX con los códigos 5XX, entonces estas medidas de seguridad pueden fracasar. Se aplicarían preocupaciones similares incluso si no está utilizando HTTP; conocer la diferencia entre una solicitud que estuvo bien y se procesó correctamente, una solicitud que fue mala y, por lo tanto, impidió que el servicio hiciera algo con ella, y una solicitud que podría estar bien pero no podemos saberlo porque el servidor no funcionó es clave para asegurando que podamos fallar rápido y rastrear problemas. Si nuestros servicios juegan rápido y suelto con estas reglas, terminaremos con un sistema más vulnerable.

Gobernanza a través del código

Reunirse y ponerse de acuerdo sobre cómo se pueden hacer las cosas es una buena idea. Pero dedicar tiempo a asegurarse de que las personas sigan estas pautas es menos divertido, al igual que imponer a los desarrolladores la carga de implementar todas estas cosas estándar que espera que haga cada servicio. Soy un gran creyente en hacer que sea fácil hacer lo correcto. Dos técnicas que he visto que funcionan bien aquí son el uso de ejemplos y el suministro de plantillas de servicio.

ejemplares

La documentación escrita es buena y útil. Veo claramente el valor; después de todo, he escrito este libro. Pero a los desarrolladores también les gusta el código, y el código que pueden ejecutar y explorar. Si tiene un conjunto de estándares o mejores prácticas que le gustaría fomentar, entonces es útil tener ejemplos que pueda señalar a las personas. La idea es que la gente no se equivoque con solo imitar algunas de las mejores partes de su sistema.

Idealmente, estos deberían ser servicios del mundo real que tenga y que hagan las cosas bien, en lugar de servicios aislados que solo se implementan para ser *ejemplos perfectos*. Al asegurarse de que sus ejemplos realmente se utilicen, se asegura de que todos los principios que tiene realmente tengan sentido.

Plantilla de servicio personalizada

¿No sería genial si pudieras hacer que sea muy fácil para todos los desarrolladores seguir la mayoría de las pautas que tienes con muy poco trabajo? ¿Qué pasaría si, listos para usar, los desarrolladores tuvieran la mayor parte del código listo para implementar los atributos principales que necesita cada servicio?

Dropwizard y **Karyon** son dos microcontenedores de código abierto basados en JVM. Funcionan de manera similar, reuniendo un conjunto de bibliotecas para proporcionar funciones como verificación de estado, servicio HTTP o exposición de métricas. Entonces, listo para usar, tiene un servicio completo con un contenedor de servlet integrado que se puede iniciar desde la línea de comando. Esta es una excelente manera de comenzar, pero ¿por qué detenerse allí? Mientras lo hace, ¿por qué no tomar algo como Dropwizard o Karyon y agregar más funciones para que se ajuste a su contexto?

Por ejemplo, es posible que desee exigir el uso de disyuntores. En ese caso, podría integrar una biblioteca de disyuntores como **Hystrix**. O puede tener la práctica de que todas sus métricas deben enviarse a un servidor Graphite central, por lo que quizás obtenga una biblioteca de código abierto como **Metrics** de Dropwizard y configúrela para que, de forma inmediata, los tiempos de respuesta y las tasas de error se envíen automáticamente. a un lugar conocido.

Al adaptar una plantilla de servicio de este tipo para su propio conjunto de prácticas de desarrollo, se asegura de que los equipos puedan ponerse en marcha más rápido y también de que los desarrolladores tengan que hacer todo lo posible para que sus servicios se comporten mal.

Por supuesto, si adoptó varias pilas de tecnología dispares, necesitaría una plantilla de servicio correspondiente para cada una. Sin embargo, esta puede ser una forma de restringir sutilmente las opciones de idioma en sus equipos. Si la plantilla de servicio interna solo es compatible con Java, es posible que las personas se desanimen a elegir pilas alternativas si tienen que hacer mucho más trabajo por sí mismos. Netflix, por ejemplo, está especialmente preocupado por aspectos como la tolerancia a fallas, para garantizar que la interrupción de una parte de su sistema no pueda acabar con todo. Para manejar esto, se ha realizado una gran cantidad de trabajo para garantizar que haya bibliotecas de clientes en la JVM para proporcionar a los equipos las herramientas que necesitan para mantener el buen comportamiento de sus servicios.

Cualquiera que introduzca una nueva pila de tecnología significaría tener que reproducir todo este esfuerzo. La principal preocupación de Netflix es menos sobre el esfuerzo duplicado y más sobre el hecho de que es tan fácil equivocarse. El riesgo de que un servicio tenga una tolerancia a fallas recién implementada es alto si pudiera afectar más al sistema. Netflix mitiga esto mediante el uso de *servicios sidecar*, que se comunican localmente con una JVM que utiliza las bibliotecas adecuadas.

Debe tener cuidado de que la creación de la plantilla de servicio no se convierta en el trabajo de un equipo central de herramientas o arquitectura que dicte cómo se deben hacer las cosas, aunque sea a través del código.

La definición de las prácticas que utiliza debe ser una actividad colectiva, por lo que, idealmente, su(s) equipo(s) deben asumir la responsabilidad conjunta de actualizar esta plantilla (un enfoque interno de código abierto funciona bien aquí).

También he visto cómo se destruye la moral y la productividad de muchos equipos al imponerles un marco obligatorio. En un intento por mejorar la reutilización del código, cada vez se coloca más trabajo en un marco centralizado hasta que se convierte en una monstruosidad abrumadora. Si decide utilizar una plantilla de servicio personalizada, piense muy detenidamente cuál es su trabajo. Idealmente, su uso debería ser puramente opcional, pero si va a ser más contundente en su adopción, debe comprender que la facilidad de uso para los desarrolladores debe ser una fuerza guía principal.

También tenga en cuenta los peligros del código compartido. En nuestro deseo de crear código reutilizable, podemos introducir fuentes de acoplamiento entre servicios. Al menos una organización con la que hablé está tan preocupada por esto que en realidad copia su código de plantilla de servicio manualmente en cada servicio. Esto significa que una actualización de la plantilla de servicio principal tarda más en aplicarse en todo su sistema, pero esto le preocupa menos que el peligro de acoplamiento. Otros equipos con los que he hablado simplemente han tratado la plantilla de servicio como una dependencia binaria compartida, aunque tienen que ser muy diligentes para no dejar que la tendencia a SECO (no se repita) resulte en un sistema demasiado acoplado. Este es un tema matizado, por lo que lo exploraremos con más detalle en el **Capítulo 4**.

Deuda técnica

A menudo nos encontramos en situaciones en las que no podemos seguir al pie de la letra nuestra visión técnica. A menudo, tenemos que tomar la decisión de cortar algunas esquinas para obtener algunas funciones urgentes. Esta es solo una compensación más que tendremos que hacer. Nuestra visión técnica existe por una razón. Si nos desviamos de esta razón, puede tener un beneficio a corto plazo pero un costo a largo plazo. Un concepto que nos ayuda a entender este trade-off es la deuda técnica. Cuando acumulamos deuda técnica, al igual que la deuda en el mundo real, tiene un costo continuo y es algo que queremos pagar.

A veces, la deuda técnica no es solo algo que causamos al tomar atajos. ¿Qué sucede si nuestra visión del sistema cambia, pero no todo nuestro sistema coincide? También en esta situación hemos creado nuevas fuentes de deuda técnica.

El trabajo del arquitecto es observar el panorama general y comprender este equilibrio. Es importante tener una idea del nivel de deuda y dónde involucrarse. Dependiendo de su organización, es posible que pueda brindar una guía amable, pero haga que los propios equipos decidan cómo rastrear y pagar la deuda. Para otras organizaciones, es posible que deba estar más estructurado, tal vez manteniendo un registro de deuda que se revise periódicamente.

Manejo de excepciones

Por lo tanto, nuestros principios y prácticas guían cómo deben construirse nuestros sistemas. Pero, ¿qué sucede cuando nuestro sistema se desvía de esto? A veces tomamos una decisión que es solo una excepción a la regla. En estos casos, podría valer la pena capturar tal decisión en un registro en algún lugar para referencia futura. Si se encuentran suficientes *excepciones*, eventualmente puede tener sentido cambiar el principio o la práctica para reflejar una nueva comprensión del mundo. Por ejemplo, podríamos tener una práctica que establezca que siempre usaremos MySQL para el almacenamiento de datos. Pero luego vemos razones convincentes para usar Cassandra para almacenamiento altamente escalable, momento en el que cambiamos nuestra práctica para decir: "Use MySQL para la mayoría de los requisitos de almacenamiento, a menos que espere un gran crecimiento en los volúmenes, en cuyo caso use Cassandra".

Sin embargo, probablemente valga la pena reiterar que cada organización es diferente. He trabajado con algunas empresas donde los equipos de desarrollo tienen un alto grado de confianza y autonomía, y allí los principios son livianos (y la necesidad de un manejo de excepciones manifiesto se reduce en gran medida, si no se elimina). En organizaciones más estructuradas en las que los desarrolladores tienen menos libertad, el seguimiento de las excepciones puede ser vital para garantizar que las reglas implementadas reflejen adecuadamente los desafíos que enfrentan las personas. Dicho todo esto, soy fanático de los microservicios como una forma de optimizar la autonomía de los equipos, dándoles la mayor libertad posible para resolver el problema en cuestión. Si trabaja en una organización que impone muchas restricciones sobre cómo los desarrolladores pueden hacer su trabajo, es posible que los microservicios no sean para usted.

Gobernanza y Liderazgo desde el Centro

Parte de lo que los arquitectos deben manejar es la gobernanza. ¿A qué me refiero con *gobernanza*? Resulta que los objetivos de control para la información y la tecnología relacionada (COBIT) tienen una definición bastante buena:

La gobernanza asegura que los objetivos de la empresa se alcancen mediante la evaluación de las necesidades, condiciones y opciones de las partes interesadas; establecer la dirección a través de la priorización y la toma de decisiones; y monitorear el desempeño, el cumplimiento y el progreso en relación con la dirección y los objetivos acordados.

COBIT 5

La gobernanza se puede aplicar a múltiples cosas en el foro de TI. Queremos centrarnos en el aspecto de la gobernanza técnica, algo que siento que es el trabajo del arquitecto. Si uno de los trabajos del arquitecto es garantizar que haya una visión técnica, entonces la gobernanza se trata de garantizar que lo que estamos construyendo coincida con esta visión y evolucionar la visión si es necesario.

Los arquitectos son responsables de muchas cosas. Deben asegurarse de que haya un conjunto de principios que puedan guiar el desarrollo y que estos principios coincidan con la estrategia de la organización. También deben asegurarse de que estos principios no requieran prácticas de trabajo que hagan sentir miserables a los desarrolladores. Necesitan mantenerse al día con la nueva tecnología y saber cuándo hacer las concesiones correctas. Esta es una gran cantidad de responsabilidad. Todo eso, y también necesitan llevar a la gente con ellos, es decir, asegurarse de que los colegas con los que están trabajando entiendan las decisiones que se toman y sean contratados para llevarlas a cabo. Ah, y como ya mencionamos: necesitan pasar algún tiempo con los equipos para comprender el impacto de sus decisiones, y tal vez incluso codificar también.

¿Una orden grande? Absolutamente. Pero soy firmemente de la opinión de que no deberían hacer esto solos. Un grupo de gobierno que funcione correctamente puede trabajar en conjunto para compartir el trabajo y dar forma a la visión.

Normalmente, la gobernanza es una actividad de grupo. Podría ser una conversación informal con un equipo lo suficientemente pequeño o una reunión regular más estructurada con membresía de grupo formal para un alcance más amplio. Aquí es donde creo que los principios que cubrimos anteriormente deben discutirse y cambiarse según sea necesario. Este grupo debe estar dirigido por un tecnólogo y estar formado predominantemente por personas que ejecutan el trabajo que se está gobernando. Este grupo también debe ser responsable del seguimiento y la gestión de los riesgos técnicos.

Un modelo que prefiero en gran medida es que el arquitecto presida el grupo, pero que la mayor parte del grupo provenga de los tecnólogos de cada equipo de entrega: los líderes de cada equipo como mínimo. El arquitecto es responsable de asegurarse de que el grupo funcione, pero el grupo como un todo es responsable de la gobernanza. Esto comparte la carga y asegura que haya un mayor nivel de aceptación. También asegura que la información fluya libremente de los equipos al grupo y, como resultado, la toma de decisiones sea mucho más sensata e informada.

A veces, el grupo puede tomar decisiones con las que el arquitecto no está de acuerdo. En este punto, ¿qué debe hacer el arquitecto? Habiendo estado en esta posición antes, puedo decirles que esta es una de las situaciones más difíciles de enfrentar. A menudo, tomo el enfoque de que debo ir con la decisión del grupo. Considero que hice todo lo posible para convencer a la gente, pero al final no fui lo suficientemente convincente. El grupo suele ser mucho más sabio que el individuo, ¡y se ha demostrado que me equivoco más de una vez! E imagine lo desalentador que puede ser para un grupo que se le haya dado espacio para tomar una decisión y luego, en última instancia, ser ignorado. Pero a veces he desautorizado al grupo. Pero, ¿por qué y cuándo? ¿Cómo eliges las líneas?

Piense en enseñar a los niños a andar en bicicleta. No puedes montarlo por ellos. Los ves tambalearse, pero si intervienes cada vez que parece que se van a caer, nunca aprenderán y, en cualquier caso, se caen mucho menos de lo que crees. Pero si los ve a punto de desviarse hacia el tráfico o hacia un estanque de patos cercano, entonces debe intervenir. Del mismo modo, como arquitecto, debe tener una idea clara de cuándo, en sentido figurado, su equipo se dirige hacia un estanque de patos. También debe ser consciente de que incluso si sabe que tiene razón y anula al equipo, esto puede socavar su posición y también hacer que el equipo sienta que no tiene voz ni voto. A veces, lo correcto es estar de acuerdo con una decisión con la que no estás de acuerdo. Saber cuándo hacer esto y cuándo no es difícil, pero a veces es vital.

Construyendo un equipo

Ser la persona principal responsable de la visión técnica de su sistema y asegurarse de que está ejecutando esta visión no se trata solo de tomar decisiones tecnológicas.

Son las personas con las que trabaja las que harán el trabajo. Gran parte del papel del líder técnico se trata de ayudarlos a crecer, ayudarlos a comprender la visión por sí mismos, y también garantizar que puedan ser participantes activos en la configuración e implementación de la visión.

Ayudar a las personas que lo rodean en su propio crecimiento profesional puede tomar muchas formas, la mayoría de las cuales están fuera del alcance de este libro. Sin embargo, hay un aspecto en el que una arquitectura de microservicio es especialmente relevante. Con sistemas monolíticos más grandes, hay menos oportunidades para que las personas den un paso al frente y sean *dueños* de algo. Con los microservicios, por otro lado, tenemos múltiples bases de código autónomas que tendrán sus propios ciclos de vida independientes. Ayudar a las personas a que se hagan cargo de los servicios individuales antes de aceptar más responsabilidades puede ser una excelente manera de ayudarlos a alcanzar sus propios objetivos profesionales y, al mismo tiempo, aligera la carga de quien esté a cargo.

Soy un firme creyente de que un gran software proviene de grandes personas. Si solo te preocupas por el lado tecnológico de la ecuación, te estás perdiendo más de la mitad de la imagen.

Resumen

Para resumir este capítulo, esto es lo que veo como las responsabilidades centrales del arquitecto evolutivo:

Visión

Asegúrese de que haya una visión técnica claramente comunicada para el sistema que ayudará a que su sistema cumpla con los requisitos de sus clientes y su organización.

Empatía

Entienda el impacto de sus decisiones en sus clientes y colegas

Colaboración

Involúcrese con tantos compañeros y colegas como sea posible para ayudar a definir, refinar y ejecutar la visión.

Adaptabilidad

Asegúrese de que la visión técnica cambie a medida que sus clientes u organización lo requieran

Autonomía

Encuentre el equilibrio adecuado entre estandarizar y permitir la autonomía de sus equipos

Gobernancia

Asegurarse de que el sistema que se está implementando se ajuste a la visión técnica.

El arquitecto evolutivo es aquel que entiende que llevar a cabo esta hazaña es un acto de equilibrio constante. Las fuerzas siempre te están empujando hacia un lado o hacia el otro, y comprender adónde empujar hacia atrás o hacia dónde ir con la corriente es a menudo algo que solo viene con la experiencia. Pero la peor reacción a todas estas fuerzas que nos empujan hacia el cambio es volvernos más rígidos o fijos en nuestro pensamiento.

Si bien gran parte de los consejos de este capítulo se pueden aplicar a cualquier arquitecto de sistemas, los microservicios nos brindan muchas más decisiones que tomar. Por lo tanto, es esencial poder equilibrar mejor todas estas compensaciones.

En el próximo capítulo, tomaremos parte de nuestra nueva conciencia sobre el rol del arquitecto con nosotros a medida que empezamos a pensar en cómo encontrar los límites correctos para nuestros microservicios.

Capítulo 3. Cómo modelar servicios

El razonamiento de mi oponente me recuerda a los paganos, quienes, cuando se les preguntó sobre qué estaba el mundo, respondieron: "Sobre una tortuga". Pero, ¿sobre qué se apoya la tortuga? En otra tortuga.

José Barker (1854)

Entonces sabe qué son los microservicios y, con suerte, tiene una idea de sus beneficios clave. Probablemente estés ansioso por ir y comenzar a hacerlos, ¿verdad? Pero, ¿por dónde empezar? En este capítulo, veremos cómo pensar en los límites de sus microservicios que, con suerte, maximizarán las ventajas y evitarán algunas de las posibles desventajas. Pero primero, necesitamos algo con lo que trabajar.

Presentamos MusicCorp

Los libros sobre ideas funcionan mejor con ejemplos. Siempre que sea posible, compartiré historias de situaciones del mundo real, pero descubrí que también es útil tener un dominio ficticio con el que trabajar. A lo largo del libro, volveremos a este dominio y veremos cómo funciona el concepto de microservicios dentro de este mundo.

Así que dirijamos nuestra atención al minorista en línea de vanguardia MusicCorp. MusicCorp era recientemente un minorista de ladrillo y cemento, pero después de que el negocio de los discos de gramófono cayera al suelo, centró cada vez más sus esfuerzos en línea. La compañía tiene un sitio web, pero siente que ahora es el momento de duplicar el mundo en línea. Después de todo, esos iPods son solo una moda pasajera (los Zunes son mucho mejores, obviamente) y los fanáticos de la música están felices de esperar a que los CD lleguen a sus puertas. Calidad sobre conveniencia, ¿verdad? Y ya que estamos, ¿qué es eso de Spotify de lo que la gente sigue hablando? ¿Algún tipo de tratamiento para la piel de los adolescentes?

A pesar de estar un poco atrasado, MusicCorp tiene grandes ambiciones. Afortunadamente, ha decidido que su mejor oportunidad de dominar el mundo es asegurarse de que puede hacer cambios lo más fácilmente posible. ¡Microservicios para ganar!

¿Qué hace un buen servicio?

Antes de que el equipo de MusicCorp se pierda en la distancia, creando servicio tras servicio en un intento de entregar cintas de ocho pistas a todo el mundo, frenemos y hablemos un poco sobre la idea subyacente más importante que debemos tener en cuenta. . ¿Qué hace un buen servicio? Si ha sobrevivido a una implementación fallida de SOA, es posible que tenga una idea de hacia dónde me dirigiré a continuación. Pero en caso de que no seas tan (desafortunado), quiero que te centres en dos conceptos clave: *acoplamiento débil* y *alta cohesión*. Hablaremos en detalle a lo largo del libro sobre otras ideas y prácticas, pero serán en vano si nos equivocamos en estas dos cosas.

A pesar de que estos dos términos se usan mucho, especialmente en el contexto de los sistemas orientados a objetos, vale la pena discutir qué significan en términos de microservicios.

Bajo acoplamiento

Cuando los servicios están débilmente acoplados, un cambio a un servicio no debería requerir un cambio a otro. El objetivo de un microservicio es poder realizar un cambio en un servicio e implementarlo, sin necesidad de cambiar ninguna otra parte del sistema. Esto es realmente muy importante.

¿Qué tipo de cosas causan un acoplamiento apretado? Un error clásico es elegir un estilo de integración que une estrechamente un servicio con otro, lo que hace que los cambios dentro del servicio requieran un cambio para los consumidores. Discutiremos cómo evitar esto con más profundidad en el [Capítulo 4](#).

Un servicio débilmente acoplado sabe tan poco como necesita sobre los servicios con los que colabora. Esto también significa que probablemente queramos limitar la cantidad de diferentes tipos de llamadas de un servicio a otro, porque más allá del posible problema de rendimiento, la comunicación conversacional puede conducir a un acoplamiento estrecho.

Alta Cohesión

Queremos que el comportamiento relacionado se sienta junto y el comportamiento no relacionado se sienta en otro lugar. ¿Por qué?

Bueno, si queremos cambiar el comportamiento, queremos poder cambiarlo en un lugar y lanzar ese cambio lo antes posible. Si tenemos que cambiar ese comportamiento en muchos lugares diferentes, tendremos que lanzar muchos servicios diferentes (quizás al mismo tiempo) para lograr ese cambio. Hacer cambios en muchos lugares diferentes es más lento, e implementar muchos servicios a la vez es arriesgado, los cuales queremos evitar.

Por lo tanto, queremos encontrar límites dentro de nuestro dominio del problema que ayuden a garantizar que el comportamiento relacionado esté en un lugar y que se comunique con otros límites de la manera más flexible posible.

El contexto acotado

El libro de Eric Evans *Domain-Driven Design* (Addison-Wesley) se centra en cómo crear sistemas que modelen dominios del mundo real. El libro está lleno de grandes ideas como el uso de lenguaje ubicuo, abstracciones de repositorios y similares, pero hay un concepto muy importante que presenta Evans que al principio me pasó por alto por completo: el *contexto limitado*. La idea es que cualquier dominio dado consta de múltiples contextos delimitados, y que residen dentro de cada uno son cosas (Eric usa mucho la palabra *modelo*, que probablemente sea mejor que *cosas*) que no necesitan comunicarse al exterior, así como cosas que se comparten externamente con otros contextos acotados. Cada contexto acotado tiene una interfaz explícita, donde decide qué modelos compartir con otros contextos.

Otra definición de contextos acotados que me gusta es "una responsabilidad específica impuesta por 1 Si solicitudes de funcionalidad de la información de contexto acotado, que se han definido explícitamente en los modelos. En su libro, Evans usa la analogía de las células, donde "las células pueden existir porque sus membranas definen lo que está adentro y afuera y determinan lo que puede pasar".

Volvamos por un momento al negocio de MusicCorp. Nuestro dominio es todo el negocio en el que estamos operando. Cubre todo, desde el almacén hasta el mostrador de recepción, desde finanzas hasta pedidos. Podemos o no modelar todo eso en nuestro software, pero ese es, sin embargo, el dominio en el que estamos operando. Pensemos en partes de ese dominio que se parecen a los contextos acotados a los que se refiere Evans. En MusicCorp, nuestro almacén es un hervidero de actividad: gestionar los pedidos que se envían (y las devoluciones ocasionales), recibir las nuevas existencias, hacer carreras de carretillas elevadoras, etc. En otros lugares, el departamento de finanzas es quizás menos amante de la diversión, pero aún tiene una función muy importante dentro de nuestra organización.

Estos empleados administran la nómina, mantienen las cuentas de la empresa y producen informes importantes. Muchos informes. Probablemente también tengan juguetes de escritorio interesantes.

Modelos compartidos y ocultos

Para MusicCorp, podemos considerar el departamento de finanzas y el almacén como dos contextos delimitados separados. Ambos tienen una interfaz explícita con el mundo exterior (en términos de informes de inventario, nóminas, etc.) y tienen detalles que solo ellos necesitan conocer (carretillas elevadoras, calculadoras).

Ahora el departamento de finanzas no necesita conocer el funcionamiento interno detallado del almacén. Sin embargo, necesita saber algunas cosas, por ejemplo, necesita saber sobre los niveles de existencias para mantener las cuentas actualizadas. La figura 3-1 muestra un diagrama de contexto de ejemplo. Vemos conceptos que son internos al almacén, como Picker (personas que seleccionan pedidos), estantes que representan ubicaciones de existencias, etc. Del mismo modo, el libro mayor de la empresa es parte integral de las finanzas pero no se comparte externamente aquí.

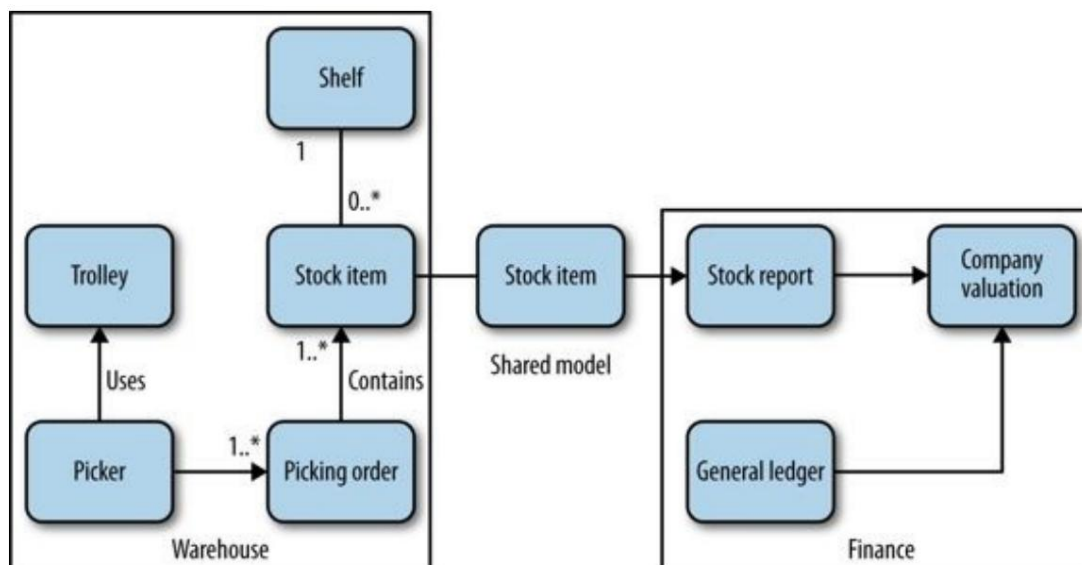


Figura 3-1. Un modelo compartido entre el departamento financiero y el almacén

Sin embargo, para poder calcular la valoración de la empresa, los empleados de finanzas necesitan información sobre las acciones que tenemos. El artículo de stock se convierte entonces en un modelo compartido entre los dos contextos. Sin embargo, tenga en cuenta que no necesitamos exponer a ciegas todo sobre el artículo en stock desde el contexto del almacén. Por ejemplo, aunque internamente mantenemos un registro de un artículo en stock en cuanto a dónde debe vivir dentro del almacén, eso no necesita estar expuesto en el modelo compartido. Así que existe la representación solo interna y la representación externa que exponemos. En muchos sentidos, esto presagia la discusión sobre REST en el Capítulo 4.

A veces podemos encontrar modelos con el mismo nombre que también tienen significados muy diferentes en diferentes contextos. Por ejemplo, podríamos tener el concepto de devolución, que representa que un cliente devuelve algo. En el contexto del cliente, una devolución se trata de imprimir una etiqueta de envío, enviar un paquete y esperar un reembolso. Para el almacén, esto podría representar un paquete que está a punto de llegar y un artículo en stock que necesita ser reabastecido. De ello se deduce que dentro del almacén almacenamos información adicional asociada a la devolución que se relaciona con las tareas a realizar;

por ejemplo, podemos generar una solicitud de reabastecimiento. El modelo compartido del retorno se asocia con diferentes procesos y entidades de apoyo dentro de cada contexto delimitado, pero eso es en gran medida una preocupación interna dentro del contexto mismo.

Módulos y Servicios

Al pensar claramente en qué modelos se deben compartir y no compartir nuestras representaciones internas, evitamos uno de los peligros potenciales que pueden resultar en un acoplamiento estrecho (lo contrario de lo que queremos). También hemos identificado un límite dentro de nuestro dominio donde deben vivir todas las capacidades comerciales afines, dándonos la alta cohesión que queremos.

Estos contextos acotados, entonces, se prestan extremadamente bien para ser límites compositivos.

Como discutimos en el [Capítulo 1](#), tenemos la opción de usar módulos dentro de un límite de proceso para mantener unido el código relacionado e intentar reducir el acoplamiento con otros módulos en el sistema. Cuando está comenzando con un nuevo código base, este es probablemente un buen lugar para comenzar. Entonces, una vez que haya encontrado sus contextos acotados en su dominio, asegúrese de que estén modelados dentro de su base de código como módulos, con modelos compartidos y ocultos.

Estos límites modulares se convierten en excelentes candidatos para microservicios. En general, los microservicios deben alinearse limpiamente con contextos limitados. Una vez que sea muy competente, puede decidir omitir el paso de mantener el contexto delimitado modelado como un módulo dentro de un sistema más monolítico y saltar directamente a un servicio separado. Sin embargo, al comenzar, mantenga un nuevo sistema en el lado más monolítico; equivocarse en los límites del servicio puede ser costoso, por lo que es sensato esperar a que las cosas se estabilicen mientras se familiariza con un nuevo dominio. Discutiremos esto más en el [Capítulo 5](#), junto con técnicas para ayudar a dividir los sistemas existentes en microservicios.

Por lo tanto, si los límites de nuestro servicio se alinean con los contextos acotados de nuestro dominio y nuestros microservicios representan esos contextos acotados, tenemos un excelente comienzo para garantizar que nuestros microservicios estén poco acoplados y fuertemente cohesivos.

Descomposición prematura

En ThoughtWorks, nosotros mismos experimentamos los desafíos de dividir los microservicios demasiado rápido. Aparte de la consultoría, también creamos algunos productos. Uno de ellos es SnapCI, una herramienta alojada de integración continua y entrega continua (hablaremos de esos conceptos más adelante en el **Capítulo 6**). El equipo había trabajado previamente en otra herramienta similar, Go-CD, ahora una herramienta de entrega continua de código abierto que se puede implementar localmente en lugar de estar alojada en la nube.

Aunque hubo algo de reutilización de código desde el principio entre los proyectos SnapCI y Go-CD, al final SnapCI resultó ser una base de código completamente nueva. No obstante, la experiencia previa del equipo en el dominio de las herramientas de CD los animó a avanzar más rápidamente en la identificación de límites y construir su sistema como un conjunto de microservicios.

Sin embargo, después de unos meses, quedó claro que los casos de uso de SnapCI eran lo suficientemente sutilmente diferentes como para que la visión inicial de los límites del servicio no fuera del todo correcta. Esto llevó a que se hicieran muchos cambios en los servicios y a un alto costo de cambio asociado. Finalmente, el equipo volvió a fusionar los servicios en un sistema monolítico, lo que les dio tiempo para comprender mejor dónde deberían existir los límites. Un año más tarde, el equipo pudo dividir el sistema monolítico en microservicios, cuyos límites demostraron ser mucho más estables. Esto está lejos de ser el único ejemplo de esta situación que he visto.

La descomposición prematura de un sistema en microservicios puede resultar costosa, especialmente si es nuevo en el dominio. En muchos sentidos, tener una base de código existente que desea descomponer en microservicios es mucho más fácil que intentar ir a los microservicios desde el principio.

Capacidades comerciales

Cuando empiece a pensar en los contextos limitados que existen en su organización, no debe pensar en términos de datos que se comparten, sino en las capacidades que esos contextos brindan al resto del dominio. El almacén puede proporcionar la capacidad de obtener una lista de existencias actual, por ejemplo, o el contexto financiero puede exponer las cuentas de fin de mes o permitirle configurar la nómina para un nuevo empleado. Estas capacidades pueden requerir el intercambio de información (modelos compartidos), pero he visto con demasiada frecuencia que pensar en *los datos* conduce a servicios anémicos basados en CRUD (crear, leer, actualizar, eliminar). Así que pregunte primero "¿Qué hace este contexto?", y luego "¿Qué datos necesita para hacer eso?"

Cuando se modelan como servicios, estas capacidades se convierten en las operaciones clave que se expondrán por cable a otros colaboradores.

Tortugas hasta el fondo

Al principio, probablemente identificará una serie de contextos limitados de grano grueso. Pero estos contextos acotados pueden a su vez contener otros contextos acotados. Por ejemplo, podría descomponer el almacén en capacidades asociadas con el cumplimiento de pedidos, la gestión de inventario o la recepción de mercancías. Al considerar los límites de sus microservicios, primero piense en términos de contextos más grandes y de grano más grueso, y luego subdivida a lo largo de estos contextos anidados cuando esté buscando los beneficios de dividir estas costuras.

He visto que estos contextos anidados permanecen ocultos para otros microservicios que colaboran con gran efecto. Para el mundo exterior, todavía están haciendo uso de las capacidades comerciales en el almacén, pero no saben que sus solicitudes en realidad se asignan de manera transparente a dos o más servicios separados, como puede ver en [la Figura 3-2](#). A veces, decidirá que tiene más sentido que el contexto delimitado de nivel superior no se modele explícitamente como un límite de servicio, como en [la Figura 3-3](#), por lo que en lugar de un único límite de almacén, podría dividir el inventario, el cumplimiento de pedidos y recepción de mercancías.

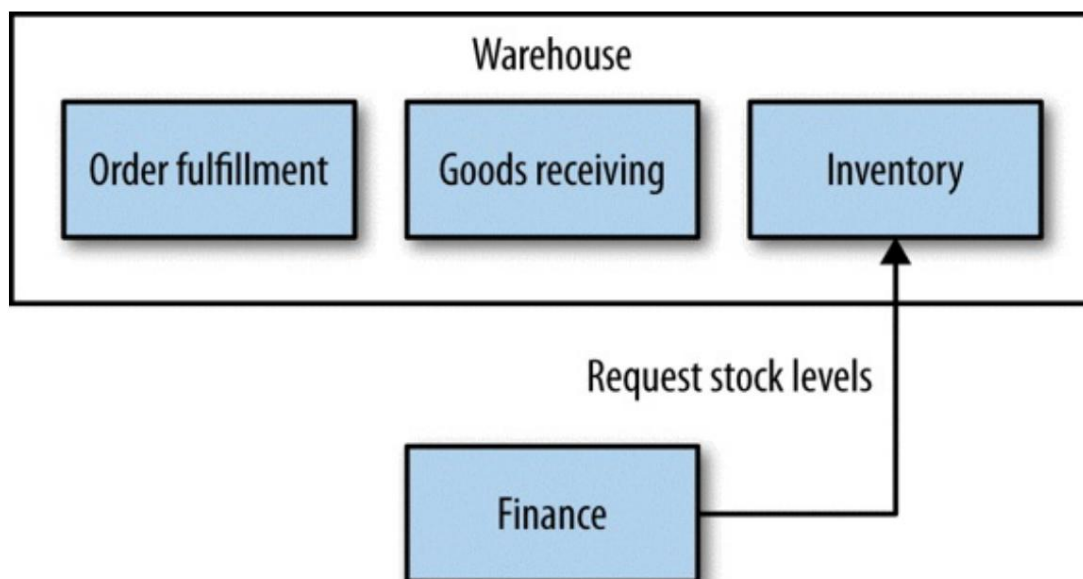


Figura 3-2. Microservicios que representan contextos delimitados anidados ocultos dentro del almacén

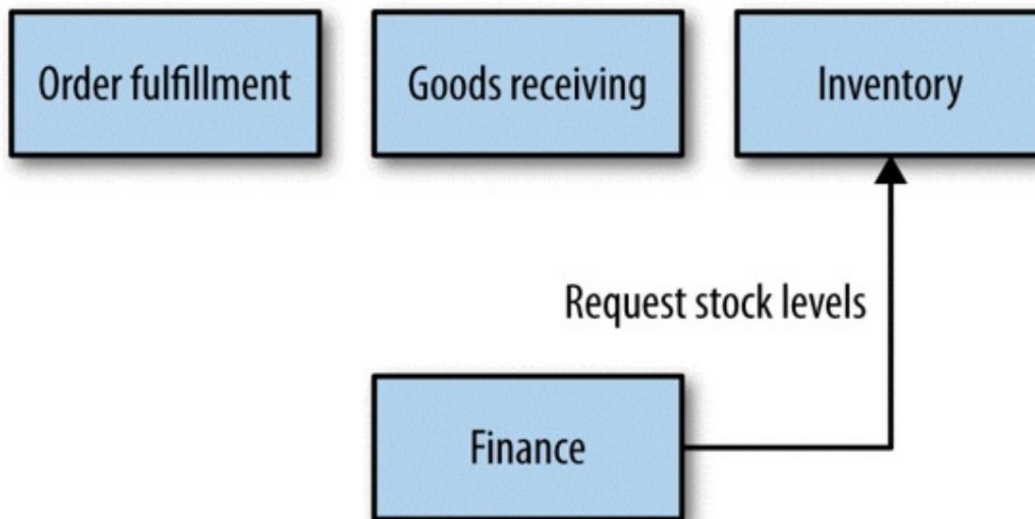


Figura 3-3. Los contextos delimitados dentro del almacén se muestran en sus propios contextos de nivel superior

En general, no existe una regla estricta sobre qué enfoque tiene más sentido.

Sin embargo, si elige el enfoque anidado en lugar del enfoque de separación total, debe basarse en su estructura organizacional. Si el cumplimiento de pedidos, la gestión de inventario y la recepción de mercancías están a cargo de diferentes equipos, probablemente merezcan su estatus de microservicios *de nivel superior*. Si, por otro lado, todos ellos son administrados por un equipo, entonces el modelo anidado tiene más sentido. Esto se debe a la interacción de las estructuras organizacionales y la arquitectura del software, que discutiremos hacia el final del libro en el [Capítulo 10](#).

Otra razón para preferir el enfoque anidado podría ser fragmentar su arquitectura para simplificar las pruebas. Por ejemplo, cuando pruebo servicios que consumen el almacén, no tengo que agregar cada servicio dentro del contexto del almacén, solo la API más detallada.

Esto también puede brindarle una unidad de aislamiento al considerar pruebas de mayor alcance. Puedo, por ejemplo, decidir tener pruebas de extremo a extremo donde lanzo todos los servicios dentro del contexto del almacén, pero para todos los demás colaboradores podría eliminarlos. Exploraremos más sobre las pruebas y el aislamiento en el [Capítulo 7](#).

Comunicación en términos de conceptos comerciales

Los cambios que implementamos en nuestro sistema a menudo se refieren a cambios que la empresa desea realizar en el comportamiento del sistema. Estamos cambiando la funcionalidad, las capacidades, que están expuestas a nuestros clientes. Si nuestros sistemas se descomponen a lo largo de los contextos delimitados que representan nuestro dominio, es más probable que los cambios que queremos realizar estén aislados en un solo límite de microservicio. Esto reduce la cantidad de lugares que necesitamos para realizar un cambio y nos permite implementar ese cambio rápidamente.

También es importante pensar en la comunicación entre estos microservicios en términos de los mismos conceptos comerciales. El modelado de su software según su dominio comercial no debe detenerse en la idea de contextos limitados. Los mismos términos e ideas que se comparten entre partes de su organización deben reflejarse en sus interfaces. Puede ser útil pensar en los formularios que se envían entre estos microservicios, al igual que los formularios se envían en una organización.

El límite técnico

Puede ser útil observar qué puede salir mal cuando los servicios se modelan incorrectamente. Hace un tiempo, algunos colegas y yo trabajábamos con un cliente en California, ayudando a la empresa a adoptar algunas prácticas de código más limpias y avanzar más hacia las pruebas automatizadas. Habíamos comenzado con algunas de las frutas maduras, como la descomposición del servicio, cuando notamos algo mucho más preocupante. No puedo entrar en demasiados detalles sobre lo que hizo la aplicación, pero era una aplicación orientada al público con una gran base de clientes global.

El equipo y el sistema habían crecido. Originalmente la visión de una persona, el sistema había adquirido más y más características y más y más usuarios. Eventualmente, la organización decidió aumentar la capacidad del equipo haciendo que un nuevo grupo de desarrolladores con sede en Brasil asumiera parte del trabajo. El sistema se dividió, con la mitad frontal de la aplicación esencialmente sin estado, implementando el sitio web público, como se muestra en la **Figura 3-4**. La mitad trasera del sistema era simplemente una interfaz de llamada a procedimiento remoto (RPC) sobre un almacén de datos. Esencialmente, imagine que tomó una capa de repositorio en su base de código y la convirtió en un servicio separado.

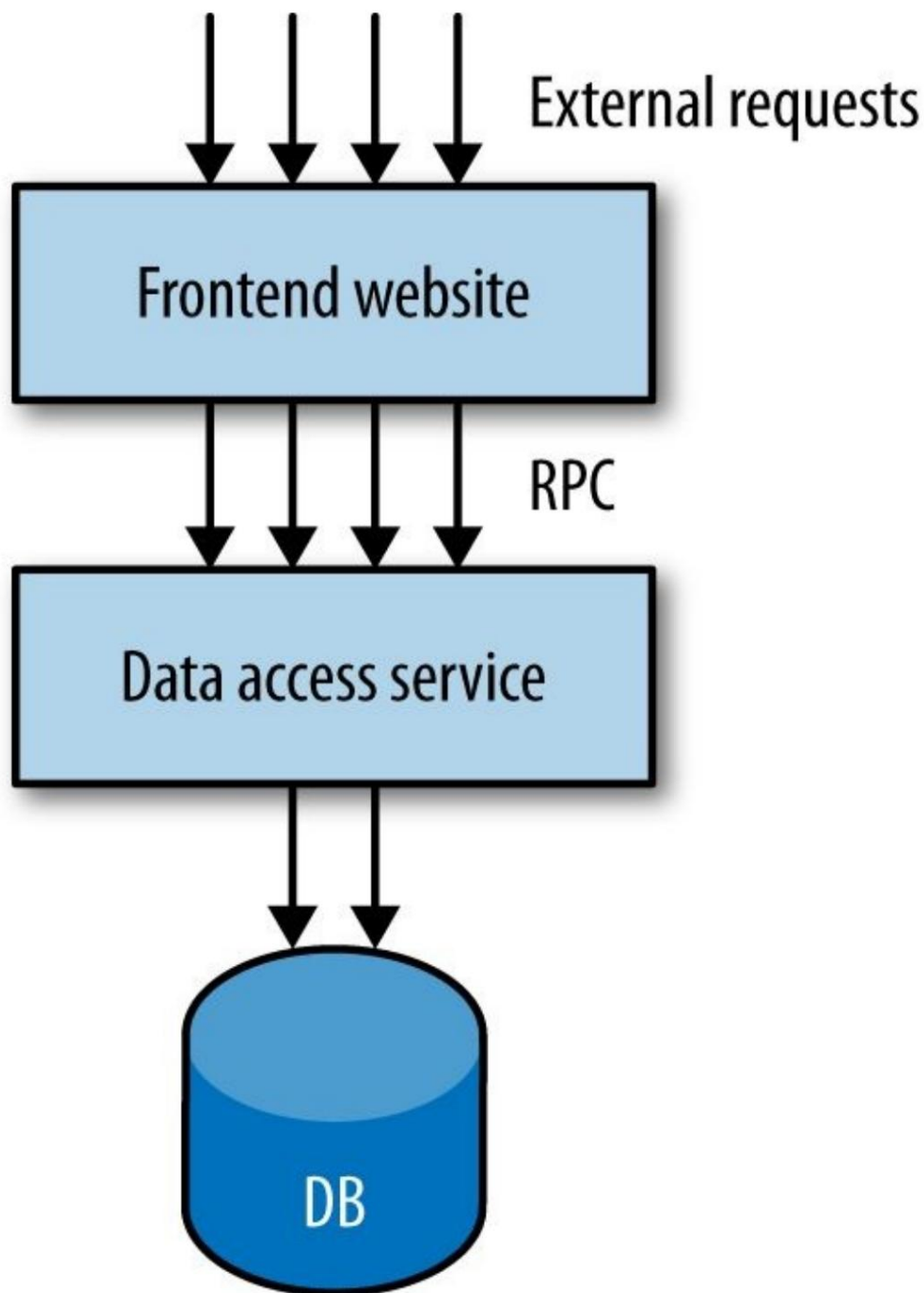


Figura 3-4. Un límite de servicio dividido a través de una costura técnica

Con frecuencia hubo que hacer cambios en ambos servicios. Ambos servicios hablaron en términos de llamadas de método de estilo RPC de bajo nivel, que eran demasiado frágiles (hablaremos de esto más adelante en el [Capítulo 4](#)). La interfaz de servicio también era muy habladora, lo que generaba problemas de rendimiento. Esto dio lugar a la necesidad de elaborar mecanismos de procesamiento por lotes de RPC. Llamé a esto *arquitectura cebolla*, ya que tenía muchas capas y me hizo llorar cuando tuvimos que cortarla.

Ahora, a primera vista, la idea de dividir el sistema previamente monolítico a lo largo de líneas geográficas/organizativas tiene perfecto sentido, como ampliaremos en el [Capítulo 10](#).

Aquí, sin embargo, en lugar de tomar un segmento vertical centrado en el negocio a través de la pila, el equipo eligió lo que anteriormente era una API en proceso e hizo un segmento horizontal.

Tomar decisiones para modelar los límites del servicio a lo largo de las costuras técnicas no siempre es incorrecto. Ciertamente he visto que esto tiene mucho sentido cuando una organización busca alcanzar ciertos objetivos de desempeño, por ejemplo. Sin embargo, debería ser su controlador secundario.

por encontrar estas costuras, no la principal.

Resumen

En este capítulo, aprendí un poco sobre lo que hace un buen servicio y cómo encontrar costuras en nuestro espacio problemático que nos brinden los beneficios duales del acoplamiento flojo y la alta cohesión. Los contextos limitados son una herramienta vital para ayudarnos a encontrar estas costuras y, al alinear nuestros microservicios con estos límites, nos aseguramos de que el sistema resultante tenga todas las posibilidades de mantener intactas esas virtudes. También tenemos una pista sobre cómo podemos subdividir aún más nuestros microservicios, algo que exploraremos con mayor profundidad más adelante. Y también presentamos MusicCorp, el dominio de ejemplo que usaremos a lo largo de este libro.

Las ideas presentadas en *Domain-Driven Design* de Eric Evans son muy útiles para encontrar límites sensatos para nuestros servicios, y aquí solo rasqué la superficie. Recomiendo el libro de Vaughn Vernon *Implementing Domain-Driven Design* (Addison Wesley) para ayudarlo a comprender los aspectos prácticos de este enfoque.

Aunque este capítulo ha sido mayormente de alto nivel, necesitamos ser mucho más técnicos en el siguiente. Hay muchas trampas asociadas con la implementación de interfaces entre servicios que pueden generar todo tipo de problemas, y tendremos que profundizar en este tema si queremos evitar que nuestros sistemas se conviertan en un desastre gigante y enredado.

¹ <http://bit.ly/limited-context-explained>

Capítulo 4. Integración

En mi opinión, lograr la integración correcta es el aspecto más importante de la tecnología asociada con los microservicios. Hágalo bien y sus microservicios conservarán su autonomía, permitiéndole cambiarlos y liberarlos independientemente del todo. Hazlo mal, y el desastre te espera. Con suerte, una vez que haya leído este capítulo, aprenderá a evitar algunos de los mayores obstáculos que han plagado otros intentos de SOA y que aún podrían esperarlo en su viaje hacia los microservicios.

En busca de la tecnología de integración ideal

Existe una variedad desconcertante de opciones sobre cómo un microservicio puede comunicarse con otro. Pero, ¿cuál es el correcto: SOAP? XML-RPC? ¿DESCANSAR? ¿Búferes de protocolo? Nos sumergiremos en eso en un momento, pero antes de hacerlo, pensemos en lo que queremos de cualquier tecnología que elijamos.

Evite romper los cambios

De vez en cuando, podemos hacer un cambio que requiera que nuestros consumidores también cambien. Discutiremos cómo manejar esto más adelante, pero queremos elegir una tecnología que garantice que esto suceda con la menor frecuencia posible. Por ejemplo, si un microservicio agrega nuevos campos a un dato que envía, los consumidores existentes no deberían verse afectados.

Mantenga sus API independientes de la tecnología

Si ha estado en la industria de TI durante más de 15 minutos, no necesita que le diga que trabajamos en un espacio que está cambiando rápidamente. La única certeza es el cambio. Constantemente surgen nuevas herramientas, marcos y lenguajes, que implementan nuevas ideas que pueden ayudarnos a trabajar de manera más rápida y eficaz. En este momento, es posible que sea una tienda de .NET. Pero, ¿qué pasa dentro de un año o dentro de cinco años? ¿Qué sucede si desea experimentar con una pila de tecnología alternativa que podría hacerlo más productivo?

Soy un gran fanático de mantener mis opciones abiertas, por eso soy tan fanático de los microservicios. También es por eso que creo que es muy importante asegurarse de mantener las API utilizadas para la comunicación entre microservicios independientes de la tecnología. Esto significa evitar la tecnología de integración que dicta qué pilas de tecnología podemos usar para implementar nuestros microservicios.

Simplifique su servicio para los consumidores

Queremos facilitar a los consumidores el uso de nuestro servicio. ¡Tener un microservicio bellamente factorizado no cuenta mucho si el costo de usarlo como consumidor es muy alto! Así que pensemos en qué facilita a los consumidores el uso de nuestro nuevo y maravilloso servicio.

Idealmente, nos gustaría permitir a nuestros clientes total libertad en su elección de tecnología, pero por otro lado, proporcionar una biblioteca de clientes puede facilitar la adopción. Sin embargo, a menudo, tales bibliotecas son incompatibles con otras cosas que queremos lograr. Por ejemplo, podríamos usar bibliotecas de clientes para que sea más fácil para los consumidores, pero esto puede tener el costo de un mayor acoplamiento.

Ocultar detalle de implementación interna

No queremos que nuestros consumidores estén atados a nuestra implementación interna. Esto conduce a un mayor acoplamiento. Esto significa que si queremos cambiar algo dentro de nuestro microservicio, podemos quebrantar a nuestros consumidores al exigirles que también cambien. Eso aumenta el costo del cambio, el resultado exacto que estamos tratando de evitar. También significa que es menos probable que deseemos hacer un cambio por temor a tener que actualizar a nuestros consumidores, lo que puede conducir a una mayor deuda técnica dentro del servicio. Por lo tanto, debe evitarse cualquier tecnología que nos empuje a exponer los detalles de la representación interna.

Interfaz con los clientes

Ahora que tenemos algunas pautas que pueden ayudarnos a seleccionar una buena tecnología para usar en la integración entre servicios, veamos algunas de las opciones más comunes e intentemos determinar cuál funciona mejor para nosotros. Para ayudarnos a pensar en esto, elijamos un ejemplo del mundo real de MusicCorp.

La creación de clientes a primera vista podría considerarse un conjunto simple de operaciones CRUD, pero para la mayoría de los sistemas es más complejo que eso. Es posible que la inscripción de un nuevo cliente deba iniciar procesos adicionales, como configurar pagos financieros o enviar correos electrónicos de bienvenida. Y cuando cambiamos o eliminamos un cliente, también se pueden desencadenar otros procesos comerciales.

Entonces, con eso en mente, deberíamos ver algunas formas diferentes en las que podríamos querer trabajar con los clientes en nuestro sistema MusicCorp.

La base de datos compartida

Con mucho, la forma más común de integración que yo o cualquiera de mis colegas vemos en la industria es la integración de base de datos (DB). En este mundo, si otros servicios quieren información de un servicio, acceden a la base de datos. Y si quieren cambiarlo, ¡buscan en la base de datos! Esto es realmente simple cuando lo piensa por primera vez, y es probablemente la forma más rápida de integración para comenzar, lo que probablemente explica su popularidad.

La Figura 4-1 muestra nuestra interfaz de usuario de registro, que crea clientes al realizar operaciones SQL directamente en la base de datos. También muestra nuestra aplicación de centro de llamadas que ve y edita datos de clientes ejecutando SQL en la base de datos. Y el almacén actualiza la información sobre los pedidos de los clientes consultando la base de datos. Este es un patrón bastante común, pero está plagado de dificultades.

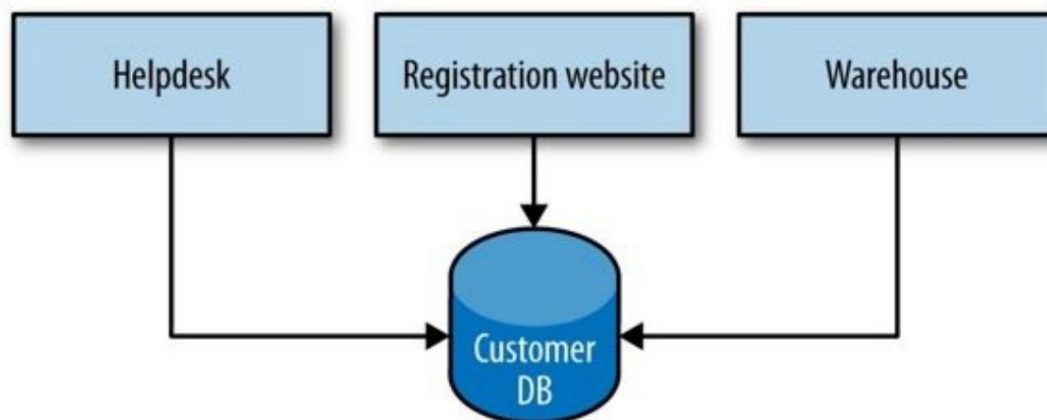


Figura 4-1. Uso de la integración de base de datos para acceder y cambiar la información del cliente

En primer lugar, permitimos que partes externas vean y se vinculen a los detalles de implementación interna.

Las estructuras de datos que almaceno en la base de datos son un juego justo para todos; se comparten en su totalidad con todas las demás partes con acceso a la base de datos. Si decido cambiar mi esquema para representar mejor mis datos, o hacer que mi sistema sea más fácil de mantener, puedo arruinar a mis consumidores. La base de datos es efectivamente una API compartida muy grande que también es bastante frágil. Si quiero cambiar la lógica asociada con, por ejemplo, cómo el servicio de asistencia gestiona a los clientes y esto requiere un cambio en la base de datos, debo tener mucho cuidado de no romper partes del esquema utilizado por otros servicios. Esta situación normalmente resulta en la necesidad de una gran cantidad de pruebas de regresión.

En segundo lugar, mis consumidores están atados a una elección de tecnología específica. Tal vez en este momento tenga sentido almacenar clientes en una base de datos relacional, por lo que mis consumidores usan un controlador apropiado (potencialmente específico de la base de datos) para comunicarse con él. ¿Qué pasa si con el tiempo nos damos cuenta de que sería mejor almacenar datos en una base de datos no relacional? ¿Puede tomar esa decisión? Por lo tanto, los consumidores están íntimamente ligados a la implementación del servicio al cliente. Como discutimos anteriormente, realmente queremos asegurarnos de que los detalles de implementación estén ocultos para los consumidores para permitir que nuestro servicio tenga un nivel de autonomía en términos de cómo cambia sus componentes internos con el tiempo. Adiós, acoplamiento flojo.

Finalmente, pensemos en el comportamiento por un momento. Habrá una lógica asociada con la forma en que se cambia un cliente. ¿Dónde está esa lógica? Si los consumidores manipulan directamente la base de datos, deben poseer la lógica asociada. La lógica para realizar los mismos tipos de manipulación a un cliente ahora puede distribuirse entre múltiples consumidores. Si el almacén, la interfaz de usuario de registro y la interfaz de usuario del centro de llamadas necesitan editar la información del cliente, necesito corregir un error o cambiar el comportamiento en tres lugares diferentes e implementar esos cambios también.

Adiós, cohesión.

¿Recuerdas cuando hablamos sobre los principios básicos detrás de buenos microservicios? Cohesión fuerte y acoplamiento flexible: con la integración de la base de datos, perdemos ambas cosas. La integración de la base de datos facilita que los servicios compartan datos, pero no hace nada sobre *el comportamiento de compartir*. Nuestra representación interna está expuesta por cable a nuestros consumidores, y puede ser muy difícil evitar hacer cambios importantes, lo que inevitablemente genera temor a cualquier cambio. Evitar (casi) a toda costa.

En el resto del capítulo, exploraremos diferentes estilos de integración que implican servicios de colaboración, que a su vez ocultan sus propias representaciones internas.

Sincrónico versus asincrónico

Antes de comenzar a sumergirnos en los detalles de las diferentes opciones tecnológicas, debemos analizar una de las decisiones más importantes que podemos tomar en términos de cómo colaboran los servicios.

¿La comunicación debe ser sincrónica o asincrónica? Esta elección fundamental nos guía inevitablemente hacia ciertos detalles de implementación.

Con la comunicación síncrona, se realiza una llamada a un servidor remoto, que se bloquea hasta que se completa la operación. Con la comunicación asincrónica, la persona que llama no espera a que se complete la operación antes de regresar, y es posible que ni siquiera le importe si la operación se completa o no.

La comunicación sincrónica puede ser más fácil de razonar. Sabemos cuándo las cosas se han completado con éxito o no. La comunicación asíncrona puede ser muy útil para trabajos de ejecución prolongada, en los que mantener una conexión abierta durante un largo período de tiempo entre el cliente y el servidor no es práctico. También funciona muy bien cuando se necesita baja latencia, donde bloquear una llamada mientras se espera el resultado puede ralentizar las cosas. Debido a la naturaleza de las redes y dispositivos móviles, enviar solicitudes y asumir que las cosas han funcionado (a menos que se indique lo contrario) puede garantizar que la interfaz de usuario siga respondiendo incluso si la red está muy retrasada. Por otro lado, la tecnología para manejar la comunicación asíncrona puede ser un poco más complicada, como veremos en breve.

Estos dos modos diferentes de comunicación pueden habilitar dos estilos idiomáticos diferentes de colaboración: *solicitud/respuesta* o *basada en eventos*. Con solicitud/respuesta, un cliente inicia una solicitud y espera la respuesta. Este modelo claramente se alinea bien con la comunicación sincrónica, pero también puede funcionar para la comunicación asincrónica. Podría iniciar una operación y registrar una devolución de llamada, pidiéndole al servidor que me informe cuando se haya completado mi operación.

Con una colaboración basada en eventos, invertimos las cosas. En lugar de que un cliente inicie solicitudes pidiendo que se hagan cosas, dice *que esto sucedió* y espera que otras partes sepan qué hacer. Nunca le decimos a nadie qué hacer. Los sistemas basados en eventos por su naturaleza son asincrónicos. La inteligencia se distribuye de manera más uniforme, es decir, la lógica comercial no se centraliza en los cerebros centrales, sino que se distribuye de manera más uniforme entre los diversos colaboradores. La colaboración basada en eventos también está muy desacoplada. El cliente que emite un evento no tiene forma de saber quién o qué reaccionará ante él, lo que también significa que puede agregar nuevos suscriptores a estos eventos sin que el cliente necesite saberlo.

Entonces, ¿hay otros impulsores que puedan empujarnos a elegir un estilo sobre otro? Un factor importante a considerar es qué tan bien se adaptan estos estilos para resolver un problema a menudo complejo: ¿cómo manejamos los procesos que abarcan los límites del servicio y pueden durar mucho tiempo?

Orquestación Versus Coreografía

A medida que comenzamos a modelar una lógica cada vez más compleja, tenemos que lidiar con el problema de administrar los procesos comerciales que se extienden más allá de los límites de los servicios individuales. Y con los microservicios, alcanzaremos este límite antes de lo habitual. Tomemos un ejemplo de MusicCorp y veamos lo que sucede cuando creamos un cliente:

1. Se crea un nuevo registro en el banco de puntos de fidelidad para el cliente.
2. Nuestro sistema postal envía un paquete de bienvenida.
3. Enviamos un email de bienvenida al cliente.

Esto es muy fácil de modelar conceptualmente como un diagrama de flujo, como lo hacemos en la [Figura 4-2](#).

Cuando se trata de implementar este flujo, hay dos estilos de arquitectura que podríamos seguir. Con la orquestación, confiamos en un cerebro central para guiar e impulsar el proceso, al igual que el director de una orquesta. Con la coreografía, informamos a cada parte del sistema de su trabajo y dejamos que resuelva los detalles, como todos los bailarines encuentran su camino y reaccionan ante los que los rodean en un ballet.

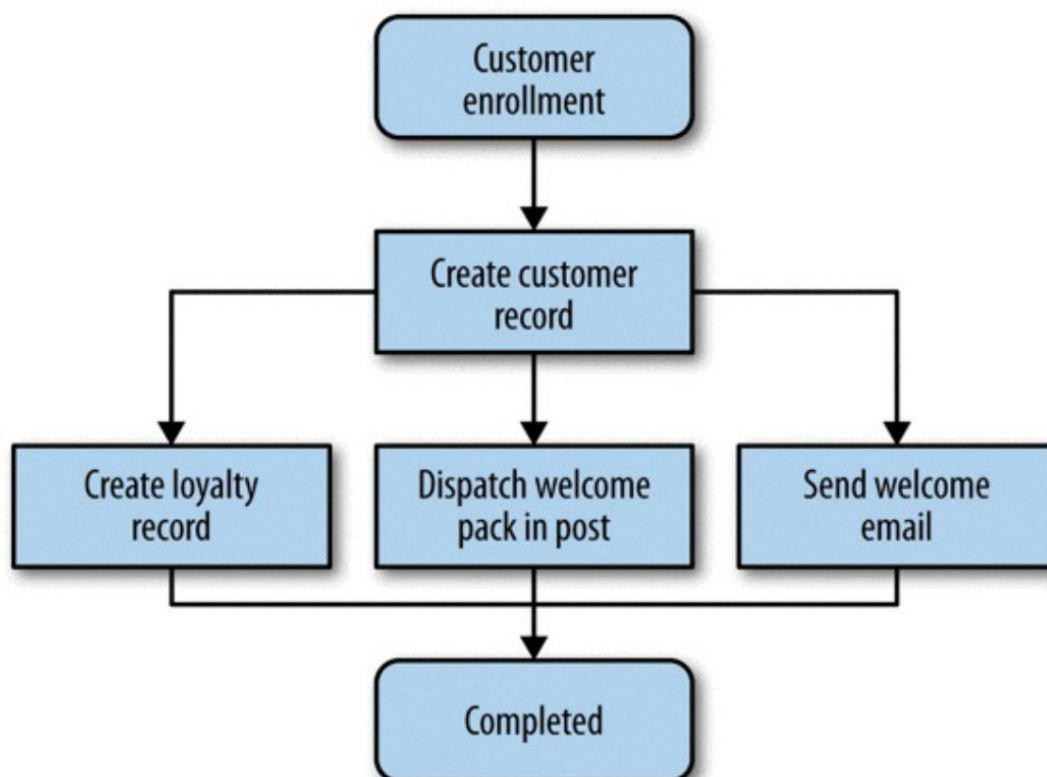


Figura 4-2. El proceso para crear un nuevo cliente

Pensemos en cómo sería una solución de orquestación para este flujo. En este caso, probablemente lo más sencillo sería que nuestro servicio de atención al cliente actuara como el cerebro central. En la creación, se comunica con el banco de puntos de fidelidad, el servicio de correo electrónico y el servicio postal, como vemos en la [Figura 4-3](#), a través de una serie de llamadas de solicitud/respuesta. el servicio de atención al cliente

puede rastrear dónde se encuentra un cliente en este proceso. Puede comprobar si se ha configurado la cuenta del cliente, si se ha enviado el correo electrónico o si se ha entregado la publicación. Tomamos el diagrama de flujo de la [Figura 4-2](#) y lo modelamos directamente en el código. Incluso podríamos usar herramientas que implementen esto por nosotros, tal vez usando un motor de reglas adecuado. Existen herramientas comerciales para este mismo propósito en forma de software de modelado de procesos comerciales. Suponiendo que usamos solicitud/respuesta síncrona, incluso podríamos saber si cada etapa ha funcionado

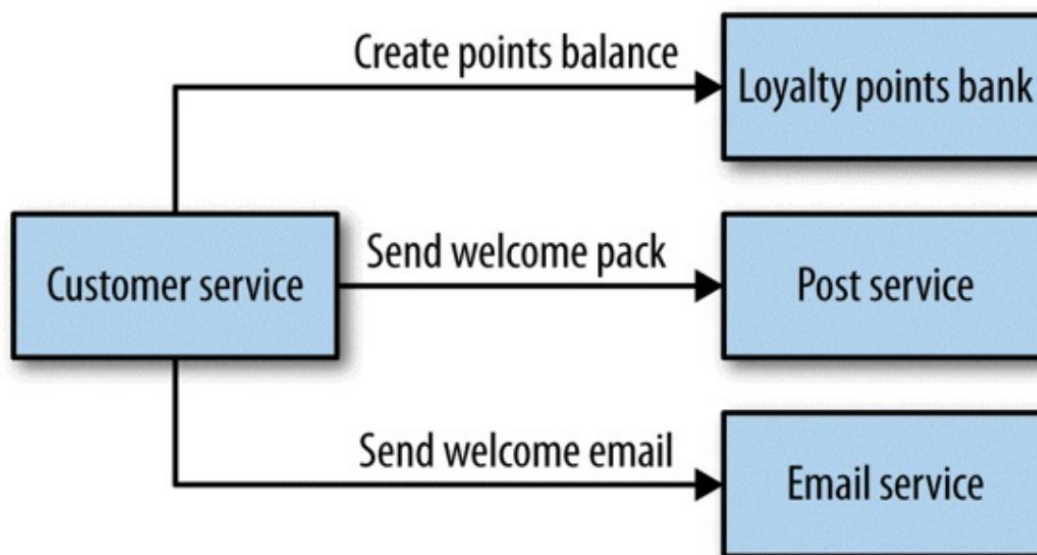


Figura 4-3. Manejar la creación de clientes a través de la orquestación

La desventaja de este enfoque de orquestación es que el servicio al cliente puede convertirse en una autoridad de gobierno central. Puede convertirse en el centro en medio de una red y en un punto central donde la lógica comienza a vivir. He visto que este enfoque da como resultado una pequeña cantidad de servicios inteligentes "dioses" que le dicen a los anémicos servicios basados en CRUD qué hacer.

Con un enfoque coreografiado, podríamos simplemente hacer que el servicio de atención al cliente emita un evento de manera asíncrona, diciendo *Creado por el cliente*. El servicio de correo electrónico, el servicio postal y el banco de puntos de fidelidad simplemente se suscriben a estos eventos y reaccionan en consecuencia, como en la [Figura 4-4](#). Este enfoque está significativamente más desacoplado. Si algún otro servicio necesita llegar a la creación de un cliente, solo necesita suscribirse a los eventos y hacer su trabajo cuando sea necesario. La desventaja es que la vista explícita del proceso comercial que vemos en la [Figura 4-2](#) ahora solo se refleja implícitamente en nuestro sistema.

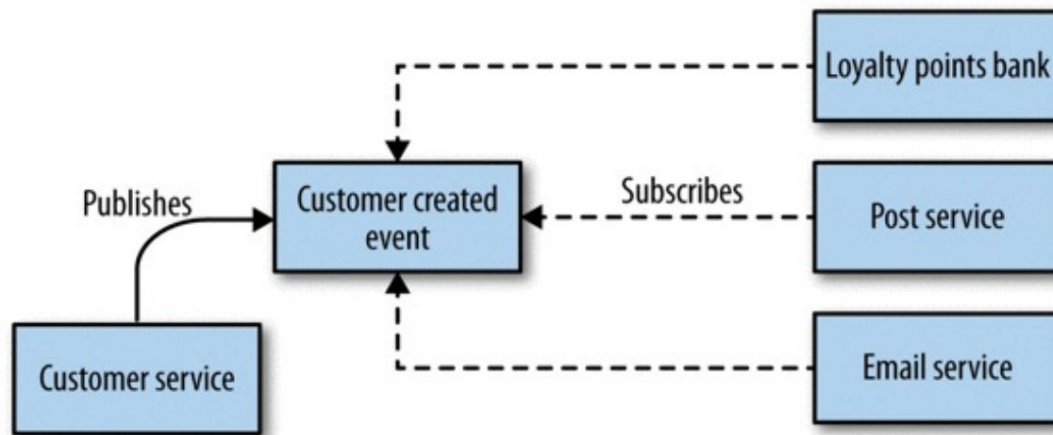


Figura 4-4. Manejar la creación de clientes a través de la coreografía.

Esto significa que se necesita trabajo adicional para garantizar que pueda monitorear y rastrear que sucedieron las cosas correctas. Por ejemplo, ¿sabría si el banco de puntos de fidelidad tuvo un error y, por alguna razón, no configuró la cuenta correcta? Un enfoque que me gusta para lidiar con esto es construir un sistema de monitoreo que coincida explícitamente con la vista del proceso comercial en la [Figura 4-2](#), pero luego rastrea lo que hace cada uno de los servicios como entidades independientes, lo que le permite ver excepciones extrañas asignadas al flujo de proceso más explícito. El diagrama de flujo que vimos antes no es la fuerza impulsora, sino solo una lente a través de la cual podemos ver cómo se comporta el sistema.

En general, he descubierto que los sistemas que tienden más hacia el enfoque coreografiado tienen un acoplamiento más flexible y son más flexibles y susceptibles de cambio. Sin embargo, necesita hacer un trabajo adicional para monitorear y rastrear los procesos a través de los límites del sistema. He descubierto que las implementaciones más fuertemente orquestadas son extremadamente frágiles, con un mayor costo de cambio. Con eso en mente, prefiero apuntar a un sistema coreografiado, donde cada servicio sea lo suficientemente inteligente como para comprender su papel en el baile completo.

Hay bastantes factores para desempacar aquí. Las llamadas sincrónicas son más simples y podemos saber si las cosas funcionaron de inmediato. Si nos gusta la semántica de solicitud/respuesta pero estamos lidiando con procesos de mayor duración, podríamos simplemente iniciar solicitudes asincrónicas y esperar las devoluciones de llamada. Por otro lado, la colaboración de eventos asincrónicos nos ayuda a adoptar un enfoque coreografiado, que puede generar muchos más servicios desacoplados, algo por lo que queremos esforzarnos para garantizar que nuestros servicios se puedan liberar de forma independiente.

Somos, por supuesto, libres de mezclar y combinar. Algunas tecnologías encajarán con más naturalidad en un estilo u otro. Sin embargo, necesitamos apreciar algunos de los diferentes detalles de implementación técnica que nos ayudarán aún más a tomar la decisión correcta.

Para empezar, veamos dos tecnologías que encajan bien cuando estamos considerando una solicitud/respuesta: llamada a procedimiento remoto (RPC) y transferencia de estado representacional (REST).

Llamadas a procedimiento remoto

La llamada a *procedimiento remoto* se refiere a la técnica de realizar una llamada local y ejecutarla en un servicio remoto en alguna parte. Existen varios tipos diferentes de tecnología RPC. Parte de esta tecnología se basa en tener una definición de interfaz (SOAP, Thrift, buffers de protocolo). El uso de una definición de interfaz separada puede facilitar la generación de stubs de cliente y servidor para diferentes pilas de tecnología, por lo que, por ejemplo, podría tener un servidor Java exponiendo una interfaz SOAP y un cliente .NET generado a partir del lenguaje de definición de servicios web. (WSDL) definición de la interfaz. Otra tecnología, como Java RMI, exige un acoplamiento más estrecho entre el cliente y el servidor, lo que requiere que ambos usen la misma tecnología subyacente pero eviten la necesidad de una definición de interfaz compartida. Todas estas tecnologías, sin embargo, tienen la misma característica central en el sentido de que hacen que una llamada local parezca una llamada remota.

Muchas de estas tecnologías son de naturaleza binaria, como Java RMI, Thrift o buffers de protocolo, mientras que SOAP usa XML para sus formatos de mensajes. Algunas implementaciones están vinculadas a un protocolo de red específico (como SOAP, que hace un uso nominal de HTTP), mientras que otras pueden permitirle usar diferentes tipos de protocolos de red, que a su vez pueden proporcionar funciones adicionales. Por ejemplo, TCP ofrece garantías sobre la entrega, mientras que UDP no, pero tiene una sobrecarga mucho menor. Esto puede permitirle utilizar diferentes tecnologías de red para diferentes casos de uso.

Esas implementaciones de RPC que le permiten generar stubs de cliente y servidor lo ayudan a comenzar muy, muy rápido. Puedo enviar contenido a través de un límite de red en muy poco tiempo. Este suele ser uno de los principales puntos de venta de RPC: su facilidad de uso. El hecho de que pueda hacer una llamada de método normal y, en teoría, ignorar el resto es una gran ayuda.

Sin embargo, algunas implementaciones de RPC vienen con algunas desventajas que pueden causar problemas. Estos problemas no siempre son evidentes al principio, pero, no obstante, pueden ser lo suficientemente graves como para superar los beneficios de ser tan fácil de poner en marcha rápidamente.

Acoplamiento tecnológico

Algunos mecanismos de RPC, como Java RMI, están fuertemente vinculados a una plataforma específica, lo que puede limitar qué tecnología se puede usar en el cliente y el servidor. Los búferes de protocolo y ahorro tienen una cantidad impresionante de soporte para lenguajes alternativos, lo que puede reducir un poco este inconveniente, pero tenga en cuenta que a veces la tecnología RPC viene con restricciones en la interoperabilidad.

En cierto modo, este acoplamiento de tecnología puede ser una forma de exponer detalles de implementación técnica interna. Por ejemplo, el uso de RMI vincula no solo al cliente con la JVM, sino también al servidor.

Las llamadas locales no son como las llamadas remotas

La idea central de RPC es ocultar la complejidad de una llamada remota. Sin embargo, muchas implementaciones de RPC ocultan demasiado. El impulso en algunas formas de RPC para hacer que las llamadas a métodos remotos parezcan llamadas a métodos locales oculta el hecho de que estas dos cosas son muy diferentes. Puedo realizar un gran número de llamadas locales en proceso sin preocuparme demasiado por el rendimiento. Sin embargo, con RPC, el costo de ordenar y desarmar las cargas útiles puede ser significativo, sin mencionar el tiempo que lleva enviar las cosas a través de la red. Esto significa que debe pensar de manera diferente sobre el diseño de API para interfaces remotas versus interfaces locales. Simplemente tomar una API local y tratar de convertirla en un límite de servicio sin pensarlo más, es probable que lo meta en problemas. En algunos de los peores ejemplos, los desarrolladores pueden estar usando llamadas remotas sin saberlo si la abstracción es demasiado opaca.

Es necesario pensar en la red en sí. Famosamente, la primera de las falacias de la computación distribuida es "**La red es confiable**". Las redes no son confiables. Pueden fallar y fallarán, incluso si su cliente y el servidor con el que está hablando están bien. Pueden fallar rápido, pueden fallar lentamente e incluso pueden malformar sus paquetes. Debe asumir que sus redes están plagadas de entidades malévolas listas para desatar su ira por capricho.

Por lo tanto, los modos de falla que puede esperar son diferentes. Una falla puede deberse a que el servidor remoto devuelve un error o a que usted realiza una mala llamada. ¿Puedes notar la diferencia, y si es así, puedes hacer algo al respecto? ¿Y qué hace cuando el servidor remoto comienza a responder lentamente? Cubriremos este tema cuando hablemos de resiliencia en el **Capítulo 11**.

fragilidad

Algunas de las implementaciones más populares de RPC pueden conducir a algunas formas desagradables de fragilidad, siendo el RMI de Java un muy buen ejemplo. Consideremos una interfaz Java muy simple que hemos decidido hacer una API remota para nuestro servicio al cliente. El [ejemplo 4-1](#) declara los métodos que vamos a exponer de forma remota. Luego, Java RMI genera los stubs de cliente y servidor para nuestro método.

Ejemplo 4-1. Definición de un punto final de servicio utilizando Java RMI

```
importar java.rmi.Remote;
importar java.rmi.RemoteException;

interfaz pública CustomerRemote extiende Remote { public
    Customer findCustomer(String id) throws RemoteException;

    public Customer createCustomer (String firstname, String apellido, String emailAddress) lanza RemoteException;
}
```

En esta interfaz, findCustomer toma el nombre, el apellido y la dirección de correo electrónico. ¿Qué sucede si decidimos permitir que el objeto Cliente también se cree solo con una dirección de correo electrónico? Podríamos agregar un nuevo método en este punto con bastante facilidad, así:

```
...
public Customer createCustomer(String emailAddress) lanza RemoteException;
...
```

El problema es que ahora también necesitamos regenerar los stubs del cliente. Los clientes que desean consumir el nuevo método necesitan los nuevos stubs y, según la naturaleza de los cambios en la especificación, los consumidores que no necesitan el nuevo método también pueden necesitar que se actualicen sus stubs. Esto es manejable, por supuesto, pero hasta cierto punto. La realidad es que cambios como este son bastante comunes. Los puntos finales de RPC a menudo terminan teniendo una gran cantidad de métodos para diferentes formas de crear o interactuar con objetos. Esto se debe en parte al hecho de que todavía estamos pensando en estas llamadas remotas como llamadas locales.

Sin embargo, hay otro tipo de fragilidad. Echemos un vistazo a cómo se ve nuestro objeto Cliente :

```
clase pública Cliente implementa Serializable {
    cadena privada nombre; privado
    String apellido; dirección de
    correo electrónico de cadena privada ;
    privado Edad de la cuerda;
}
```

Ahora, ¿qué pasa si resulta que, aunque exponemos el campo de edad en nuestros objetos Cliente , ninguno de nuestros consumidores lo usa? Decidimos que queremos eliminar este campo. Pero si la implementación del servidor elimina la edad de su definición de este tipo, y no hacemos lo mismo con todos los consumidores, aunque nunca usaron el campo, el código asociado con la deserialización del objeto Cliente en el lado del consumidor se romperá. . Para implementar este cambio, tendría que implementar un nuevo servidor y clientes al mismo tiempo. Esto es un

Desafío clave con cualquier mecanismo RPC que promueva el uso de la generación de stubs binarios: no puede separar las implementaciones de cliente y servidor. Si usa esta tecnología, las liberaciones de paso de bloqueo pueden estar en su futuro.

Ocurren problemas similares si quiero reestructurar el objeto Cliente incluso si no eliminé los campos; por ejemplo, si quisiera encapsular nombre y apellido en un nuevo tipo de nombre para que sea más fácil de administrar. Por supuesto, podría arreglar esto pasando tipos de diccionario como parámetros de mis llamadas, pero en ese punto, pierdo muchos de los beneficios de los stubs generados porque todavía tendré que hacer coincidir y extraer manualmente los campos que desear.

En la práctica, los objetos que se usan como parte de la serialización binaria a través del cable se pueden considerar como tipos *de solo expansión*. Esta fragilidad hace que los tipos queden expuestos sobre el cable y se conviertan en una masa de campos, algunos de los cuales ya no se usan pero no se pueden eliminar de manera segura.

¿Es terrible el RPC?

A pesar de sus deficiencias, no iría tan lejos como para llamar terrible a RPC. Algunas de las implementaciones comunes que he encontrado pueden dar lugar a los tipos de problemas que he descrito aquí. Debido a los desafíos de usar RMI, ciertamente le daría un amplio margen a esa tecnología. Muchas operaciones encajan bastante bien en el modelo basado en RPC, y los mecanismos más modernos, como los búferes de protocolo o Thrift, mitigan algunos de los pecados del pasado al evitar la necesidad de versiones bloqueadas del código del cliente y del servidor.

Solo tenga en cuenta algunos de los peligros potenciales asociados con RPC si va a elegir este modelo. No abstraiga sus llamadas remotas hasta el punto en que la red quede completamente oculta y asegúrese de que puede evolucionar la interfaz del servidor sin tener que insistir en las actualizaciones de bloqueo para los clientes. Por ejemplo, es importante encontrar el equilibrio adecuado para su código de cliente. Asegúrese de que sus clientes no estén ajenos al hecho de que se va a realizar una llamada de red. Las bibliotecas de clientes se usan a menudo en el contexto de RPC y, si no están bien estructuradas, pueden ser problemáticas. Hablaremos más sobre ellos en breve.

En comparación con la integración de bases de datos, RPC es sin duda una mejora cuando pensamos en las opciones para la colaboración de solicitud/respuesta. Pero hay otra opción a considerar.

DESCANSAR

La transferencia de estado representacional (REST) es un estilo arquitectónico inspirado en la Web. Hay muchos principios y restricciones detrás del estilo REST, pero nos centraremos en aquellos que realmente nos ayudan cuando enfrentamos desafíos de integración en un mundo de microservicios y cuando buscamos un estilo alternativo a RPC para nuestras interfaces de servicio.

El más importante es el concepto de recursos. Puede pensar en un recurso como algo que el propio servicio conoce, como un Cliente. El servidor crea diferentes representaciones de este Cliente a pedido. La forma en que un recurso se muestra externamente está completamente desvinculada de cómo se almacena internamente. Un cliente puede solicitar una representación JSON de un Cliente, por ejemplo, incluso si está almacenada en un formato completamente diferente. Una vez que un cliente tiene una representación de este Cliente, puede realizar solicitudes para cambiarlo, y el servidor puede cumplirlas o no.

Hay muchos estilos diferentes de REST, y solo los menciono brevemente aquí. Le recomiendo encarecidamente que eche un vistazo al [modelo de madurez de Richardson](#), donde se comparan los diferentes estilos de REST.

REST en sí mismo no habla realmente de protocolos subyacentes, aunque se usa más comúnmente sobre HTTP. He visto implementaciones de REST usando protocolos muy diferentes antes, como serial o USB, aunque esto puede requerir mucho trabajo. Algunas de las funciones que HTTP nos brinda como parte de la especificación, como los verbos, facilitan la implementación de REST sobre HTTP, mientras que con otros protocolos tendrá que manejar estas funciones usted mismo.

RESTO y HTTP

El propio HTTP define algunas capacidades útiles que funcionan muy bien con el estilo REST. Por ejemplo, los verbos HTTP (p. ej., GET, POST y PUT) ya tienen significados bien entendidos en la especificación HTTP sobre cómo deberían funcionar con los recursos. El estilo arquitectónico REST en realidad nos dice que los métodos deben comportarse de la misma manera en todos los recursos, y la especificación HTTP define un montón de métodos que podemos usar.

GET recupera un recurso de forma idempotente y POST crea un nuevo recurso. Esto significa que podemos evitar muchos métodos diferentes `createCustomer` o `editCustomer`. En cambio, podemos simplemente POST una representación del cliente para solicitar que el servidor cree un nuevo recurso e iniciar una solicitud GET para recuperar una representación de un recurso.

Conceptualmente, hay un *punto final* en forma de recurso de Cliente en estos casos, y las operaciones que podemos realizar en él se integran en el protocolo HTTP.

HTTP también trae un gran ecosistema de herramientas y tecnología de soporte. Podemos usar proxies de almacenamiento en caché HTTP como Varnish y balanceadores de carga como `mod_proxy`, y muchas herramientas de monitoreo ya tienen mucho soporte para HTTP listo para usar. Estos componentes básicos nos permiten manejar grandes volúmenes de tráfico HTTP y enrutarlos de manera inteligente, de una manera bastante transparente. También podemos usar todos los controles de seguridad disponibles con HTTP para proteger nuestras comunicaciones. Desde la autenticación básica hasta los certificados de cliente, el ecosistema HTTP nos brinda muchas herramientas para facilitar el proceso de seguridad, y exploraremos ese tema más en el [Capítulo](#) Dicho esto, para obtener estos beneficios, debe usar bien HTTP. Úselo mal, y puede ser tan inseguro y difícil de escalar como cualquier otra tecnología disponible. Sin embargo, úsalo bien y obtendrás mucha ayuda.

Tenga en cuenta que HTTP también se puede usar para implementar RPC. SOAP, por ejemplo, se enruta a través de HTTP, pero desafortunadamente usa muy poca especificación. Los verbos se ignoran, al igual que cosas simples como los códigos de error HTTP. Con demasiada frecuencia, parece que los estándares y la tecnología existentes y bien entendidos se ignoran en favor de nuevos estándares que solo se pueden implementar utilizando tecnología completamente nueva, proporcionada convenientemente por las mismas empresas que ayudaron a diseñar los nuevos estándares en primer lugar. !

Hipermedia como motor del estado de la aplicación

Otro principio introducido en REST que puede ayudarnos a evitar el acoplamiento entre el cliente y el servidor es el concepto de *hipermedia como el motor del estado de la aplicación* (a menudo abreviado como HATEOAS, y vaya si necesitaba una abreviatura). Esta es una redacción bastante densa y un concepto bastante interesante, así que analicemos un poco.

Hipermedia es un concepto mediante el cual una pieza de contenido contiene enlaces a varias otras piezas de contenido en una variedad de formatos (por ejemplo, texto, imágenes, sonidos). Esto debería ser bastante familiar para usted, ya que es lo que hace la página web promedio: sigue enlaces, que son una forma de controles hipermedia, para ver contenido relacionado. La idea detrás de HATEOAS es que los clientes deben realizar interacciones (lo que puede conducir a transiciones de estado) con el servidor a través de estos enlaces a otros recursos. No necesita saber exactamente dónde viven los clientes en el servidor sabiendo qué URI seleccionar; en cambio, el cliente busca y navega por los enlaces para encontrar lo que necesita.

Este es un concepto un poco extraño, así que primero retrocedamos y consideremos cómo las personas interactúan con una página web, que ya hemos establecido que es rica en controles hipermedia.

Piense en el sitio de compras de Amazon.com. La ubicación del carrito de compras ha cambiado con el tiempo. El gráfico ha cambiado. El enlace ha cambiado. Pero como humanos, somos lo suficientemente inteligentes como para ver un carrito de compras, saber qué es e interactuar con él. Tenemos una comprensión de lo que significa un carrito de compras, incluso si la forma exacta y el control subyacente utilizado para representarlo han cambiado. Sabemos que si queremos ver el carrito, este es el control con el que queremos interactuar. Esta es la razón por la que las páginas web pueden cambiar gradualmente con el tiempo. Siempre que se cumplan estos contratos implícitos entre el cliente y el sitio web, los cambios no tienen por qué ser cambios importantes.

Con los controles hipermedia, estamos tratando de lograr el mismo nivel de *inteligencia* para nuestros consumidores electrónicos. Veamos un control hipermedia que podríamos tener para MusicCorp. Hemos accedido a un recurso que representa una entrada de catálogo para un álbum dado en el [Ejemplo 4-2](#). Junto con la información sobre el álbum, vemos una serie de controles hipermedia.

Ejemplo 4-2. Controles hipermedia utilizados en una lista de álbumes

```
<album>
  <name>Den sangre</name>
  <link rel="/artist" href="/artist/theBrakes" /> <description> Impresionante, breve, 1
  brutal, divertida y ruidosa. ¡Debes comprar! </description> <enlace rel="/compra
  instantánea" href="/compra instantánea/1234" /> </álbum>
```

²

¹

Este control hipermedia nos muestra dónde encontrar información sobre el artista.

²

Y si queremos comprar el álbum, ahora sabemos a dónde ir.

En este documento, tenemos dos controles hipermedia. El cliente que lee un documento de este tipo necesita saber que un control con una relación de artista es donde debe navegar para obtener información sobre el artista, y que la compra instantánea es parte del protocolo utilizado para comprar el álbum. El cliente debe comprender la semántica de la API de la misma manera que un ser humano debe comprender que en un sitio web de compras, el carrito es donde estarán los artículos que se comprarán.

Como cliente, no necesito saber a qué esquema de URI acceder para *comprar* el álbum, solo necesito acceder al recurso, encontrar el control de compra y navegar hasta allí. El control de compra podría cambiar de ubicación, el URI podría cambiar, o el sitio podría incluso enviarme a otro servicio por completo, y como cliente no me importaría. Esto nos da una gran cantidad de desacoplamiento entre el cliente y el servidor.

Estamos muy abstraídos del detalle subyacente aquí. Podríamos cambiar por completo la implementación de cómo se presenta el control siempre que el cliente aún pueda encontrar un control que coincida con su comprensión del protocolo, de la misma manera que un control de carrito de compras puede pasar de ser un enlace simple a uno más complejo. Control JavaScript. También somos libres de agregar nuevos controles al documento, quizás representando nuevas transiciones de estado que podemos realizar en el recurso en cuestión. Terminaríamos rompiendo a nuestros consumidores solo si cambiáramos fundamentalmente la semántica de uno de los controles para que se comportara de manera muy diferente, o si elimináramos un control por completo.

El uso de estos controles para desacoplar el cliente y el servidor produce beneficios significativos a lo largo del tiempo que compensan en gran medida el pequeño aumento en el tiempo que lleva poner en funcionamiento estos protocolos. Al seguir los enlaces, el cliente descubre progresivamente la API, lo que puede ser una capacidad muy útil cuando estamos implementando nuevos clientes.

Una de las desventajas es que esta navegación de controles puede ser bastante habladora, ya que el cliente necesita seguir los enlaces para encontrar la operación que desea realizar. En última instancia, esto es una compensación. Le sugiero que comience haciendo que sus clientes naveguen por estos controles primero, luego optimice más tarde si es necesario. Recuerde que tenemos una gran cantidad de ayuda lista para usar usando HTTP, que discutimos anteriormente. Los males de la optimización prematura han sido bien documentados antes, por lo que no necesito ampliarlos aquí. También tenga en cuenta que muchos de estos enfoques se desarrollaron para crear sistemas de hipertexto distribuidos, ¡y no todos encajan! A veces te encontrarás queriendo un buen RPC a la antigua.

Personalmente, soy fanático del uso de enlaces para permitir que los consumidores naveguen por los puntos finales de la API. Los beneficios del descubrimiento progresivo de la API y el acoplamiento reducido pueden ser significativos. Sin embargo, está claro que no todos están vendidos, ya que no veo que se use tanto como me gustaría. Creo que una gran parte de esto se debe a que se requiere un trabajo inicial inicial, pero las recompensas a menudo llegan más tarde.

¿JSON, XML o algo más?

El uso de formatos de texto estándar brinda a los clientes mucha flexibilidad en cuanto a cómo consumen recursos, y REST sobre HTTP nos permite usar una variedad de formatos. Los ejemplos que he dado hasta ahora usan XML, pero en esta etapa, JSON es un tipo de contenido mucho más popular para servicios que funcionan sobre HTTP.

El hecho de que JSON sea un formato mucho más simple hace que el consumo también sea más fácil. Algunos defensores también citan su relativa compacidad en comparación con XML como otro factor ganador, aunque esto no suele ser un problema del mundo real.

Sin embargo, JSON tiene algunas desventajas. XML define el control de enlace que usamos anteriormente como un control hipermedia. El estándar JSON no define nada similar, por lo que los estilos internos se utilizan con frecuencia para calzar este concepto. El **lenguaje de aplicación de hipertexto (HAL)** intenta solucionar esto definiendo algunos estándares comunes para hipervínculos para JSON (y XML también). , aunque podría decirse que XML necesita menos ayuda). Si sigue el estándar HAL, puede usar herramientas como el navegador HAL basado en la web para explorar los controles hipermedia, lo que puede facilitar mucho la tarea de crear un cliente.

Por supuesto, no estamos limitados a estos dos formatos. Podemos enviar prácticamente cualquier cosa a través de HTTP si queremos, incluso binario. Cada vez veo más personas que usan HTML como formato en lugar de XML. Para algunas interfaces, el HTML puede cumplir una doble función como interfaz de usuario y API, aunque aquí hay que evitar trampas, ya que las interacciones de un ser humano y una computadora son bastante diferentes. Pero sin duda es una idea atractiva. Después de todo, hay muchos analizadores de HTML por ahí.

Sin embargo, personalmente sigo siendo un fanático de XML. Parte del soporte de herramientas es mejor. Por ejemplo, si quiero extraer solo ciertas partes de la carga útil (una técnica de la que hablaremos más en **"Versionado"**) , puedo usar XPATH, que es un estándar bien entendido con mucha compatibilidad con herramientas, o incluso selectores de CSS, que muchos encuentran aún más fácil. Con JSON, tengo JSONPATH, pero esto no es ampliamente compatible. Me resulta extraño que la gente elija JSON porque es agradable y liviano, luego intente introducir conceptos como controles hipermedia que ya existen en XML. Sin embargo, acepto que probablemente soy una minoría aquí y que JSON es el formato elegido por la mayoría de las personas.

Cuidado con demasiada comodidad

A medida que REST se ha vuelto más popular, también lo han hecho los marcos que nos ayudan a crear servicios web RESTful. Sin embargo, algunas de estas herramientas compensan demasiado en términos de ganancia a corto plazo por dolor a largo plazo; al tratar de hacerlo avanzar rápido, pueden alentar algunos malos comportamientos. Por ejemplo, algunos marcos en realidad hacen que sea muy fácil simplemente tomar representaciones de bases de datos de objetos, deserializarlos en objetos en proceso y luego exponerlos directamente externamente. Recuerdo que en una conferencia vi esto demostrado con Spring Boot y citado como una gran ventaja. El acoplamiento inherente que promueve esta configuración en la mayoría de los casos causará mucho más dolor que el esfuerzo requerido para desacoplar adecuadamente estos conceptos.

Hay un problema más general en juego aquí. Cómo decidimos almacenar nuestros datos y cómo los exponemos a nuestros consumidores, puede dominar fácilmente nuestro pensamiento. Un patrón que vi que uno de nuestros equipos usó de manera efectiva fue retrasar la implementación de la persistencia adecuada para el microservicio, hasta que la interfaz se estabilizó lo suficiente. Durante un período provisional, las entidades solo se conservaron en un archivo en el disco local, lo que obviamente no es una solución adecuada a largo plazo. Esto aseguró que la forma en que los consumidores querían usar el servicio impulsara las decisiones de diseño e implementación. La razón dada, que se confirmó en los resultados, fue que es demasiado fácil que la forma en que almacenamos las entidades de dominio en una tienda de respaldo influya abiertamente en los modelos que enviamos por cable a los colaboradores. Una de las desventajas de este enfoque es que aplazamos el trabajo necesario para conectar nuestro almacén de datos. Sin embargo, creo que para los nuevos límites de servicio, esta es una compensación aceptable.

Desventajas de REST sobre HTTP

En términos de facilidad de consumo, no puede generar fácilmente un código auxiliar de cliente para su REST sobre el protocolo de aplicación HTTP como puede hacerlo con RPC. Claro, el hecho de que se use HTTP significa que puede aprovechar todas las excelentes bibliotecas de clientes HTTP que existen, pero si desea implementar y usar controles de hipermedia como cliente, está prácticamente solo. Personalmente, creo que las bibliotecas de clientes podrían hacer esto mucho mejor de lo que lo hacen, y ciertamente son mejores ahora que en el pasado, pero he visto que esta aparente mayor complejidad da como resultado que las personas recaigan en el contrabando de RPC a través de HTTP o en la creación de bibliotecas de clientes compartidas. El código compartido entre el cliente y el servidor puede ser muy peligroso, como veremos en ["DRY y los peligros de la reutilización de código en un mundo de microservicios"](#).

Un punto menor es que algunos marcos de servidor web en realidad no admiten bien todos los verbos HTTP. Eso significa que puede ser fácil para usted crear un controlador para las solicitudes GET o POST, pero es posible que tenga que saltar a través de los aros para que las solicitudes PUT o DELETE funcionen. Los marcos REST adecuados como Jersey no tienen este problema, y normalmente puede solucionarlo, pero si está bloqueado en ciertas opciones de marco, esto podría limitar el estilo de REST que puede usar.

El rendimiento también puede ser un problema. Las cargas útiles de REST sobre HTTP en realidad pueden ser más compactas que SOAP porque admite formatos alternativos como JSON o incluso binario, pero aún no será un protocolo binario tan delgado como podría ser Thrift. La sobrecarga de HTTP para cada solicitud también puede ser una preocupación para los requisitos de baja latencia.

HTTP, si bien puede adaptarse bien a grandes volúmenes de tráfico, no es ideal para comunicaciones de baja latencia en comparación con protocolos alternativos que se basan en el Protocolo de control de transmisión (TCP) u otra tecnología de red. A pesar del nombre, WebSockets, por ejemplo, tiene muy poco que ver con la Web. Después del protocolo de enlace HTTP inicial, es solo una conexión TCP entre el cliente y el servidor, pero puede ser una forma mucho más eficiente de transmitir datos para un navegador. Si esto es algo que le interesa, tenga en cuenta que en realidad no está usando mucho HTTP, y mucho menos nada que ver con REST.

Para las comunicaciones de servidor a servidor, si es importante una latencia extremadamente baja o un tamaño de mensaje pequeño, las comunicaciones HTTP en general pueden no ser una buena idea. Es posible que deba elegir diferentes protocolos subyacentes, como el Protocolo de datagramas de usuario (UDP), para lograr el rendimiento que desea, y muchos marcos RPC se ejecutarán felizmente sobre protocolos de red distintos de TCP.

El consumo de las propias cargas útiles requiere más trabajo que el proporcionado por algunas implementaciones de RPC que admiten mecanismos avanzados de serialización y deserialización. Estos pueden convertirse en un punto de acoplamiento por derecho propio entre el cliente y el servidor, ya que implementar lectores tolerantes no es una actividad trivial (hablaremos de esto en breve), pero desde el punto de vista de la puesta en marcha, pueden ser muy atractivos.

A pesar de estas desventajas, REST sobre HTTP es una opción predeterminada sensata para las interacciones de servicio a servicio. Si quieres saber más, te recomiendo *REST in Practice* (O'Reilly), que cubre el tema de REST sobre HTTP en profundidad.

Implementación de la colaboración asincrónica basada en eventos

Hemos hablado un poco sobre algunas tecnologías que pueden ayudarnos a implementar patrones de solicitud/respuesta. ¿Qué pasa con la comunicación asincrónica basada en eventos?

Opciones tecnológicas

Hay dos partes principales que debemos considerar aquí: una forma para que nuestros microservicios emitan eventos y una forma para que nuestros consumidores descubran que esos eventos han ocurrido.

Tradicionalmente, los intermediarios de mensajes como RabbitMQ intentan manejar ambos problemas. Los productores usan una API para publicar un evento en el corredor. El corredor maneja las suscripciones, lo que permite que los consumidores estén informados cuando llega un evento. Estos corredores pueden incluso manejar el estado de los consumidores, por ejemplo, ayudando a realizar un seguimiento de los mensajes que han visto antes. Estos sistemas normalmente están diseñados para ser escalables y resistentes, pero eso no es gratis. Puede agregar complejidad al proceso de desarrollo, porque es otro sistema que puede necesitar ejecutar para desarrollar y probar sus servicios. Es posible que también se requieran máquinas y experiencia adicionales para mantener esta infraestructura en funcionamiento. Pero una vez que lo hace, puede ser una forma increíblemente efectiva de implementar arquitecturas impulsadas por eventos y débilmente acopladas. En general, soy un fan.

Sin embargo, tenga cuidado con el mundo del middleware, del cual el intermediario de mensajes es solo una pequeña parte. Las colas en sí mismas son cosas perfectamente sensatas y útiles. Sin embargo, los proveedores tienden a querer empaquetar una gran cantidad de software con ellos, lo que puede llevar a que cada vez se introduzca más inteligencia en el middleware, como lo demuestran cosas como Enterprise Service Bus. Asegúrese de saber lo que está obteniendo: mantenga su middleware tonto y mantenga la inteligencia en los puntos finales.

Otro enfoque es tratar de usar HTTP como una forma de propagar eventos. ATOM es una especificación compatible con REST que define la semántica (entre otras cosas) para publicar fuentes de recursos. Existen muchas bibliotecas de clientes que nos permiten crear y consumir estos feeds.

Por lo tanto, nuestro servicio de atención al cliente podría simplemente publicar un evento en dicha fuente cuando cambie nuestro servicio de atención al cliente. Nuestros consumidores simplemente consultan el feed en busca de cambios. Por un lado, el hecho de que podamos reutilizar la especificación ATOM existente y cualquier biblioteca asociada es útil, y sabemos que los manejos de HTTP escalan muy bien. Sin embargo, HTTP no es bueno con baja latencia (donde sobresalen algunos intermediarios de mensajes), y aún debemos lidiar con el hecho de que los consumidores deben realizar un seguimiento de los mensajes que han visto y administrar su propio programa de encuestas.

He visto a personas pasar una edad implementando más y más de los comportamientos que obtienes de la caja con un intermediario de mensajes apropiado para hacer que ATOM funcione en algunos casos de uso. Por ejemplo, el patrón Consumidor en competencia describe un método mediante el cual se activan varias instancias de trabajadores para competir por los mensajes, lo que funciona bien para aumentar la cantidad de trabajadores para manejar una lista de trabajos independientes. Sin embargo, queremos evitar el caso de que dos o más trabajadores vean el mismo mensaje, ya que terminaremos haciendo la misma tarea más de lo necesario. Con un intermediario de mensajes, una cola estándar manejará esto. Con ATOM, ahora necesitamos administrar nuestro propio estado compartido entre todos los trabajadores para tratar de reducir las posibilidades de reproducir el esfuerzo.

Si ya tiene a su disposición un buen agente de mensajes resistente, considere usarlo para gestionar la publicación y la suscripción a eventos. Pero si aún no tiene uno, eche un vistazo a ATOM, pero tenga en cuenta la falacia del costo irrecuperable. Si se da cuenta de que desea más y más del soporte que le brinda un agente de mensajes, en un momento dado, es posible que desee cambiar su enfoque.

En términos de lo que realmente enviamos a través de estos protocolos asíncronos, se aplican las mismas consideraciones que con la comunicación síncrona. Si actualmente está satisfecho con la codificación de solicitudes y respuestas usando JSON, manténgalo.

Complejidades de las arquitecturas asíncronas

Algunas de estas cosas asincrónicas parecen divertidas, ¿verdad? Las arquitecturas basadas en eventos parecen conducir a sistemas escalables significativamente más desacoplados. Y pueden. Pero estos estilos de programación conducen a un aumento de la complejidad. No se trata solo de la complejidad necesaria para gestionar la publicación y suscripción de mensajes, como acabamos de comentar, sino también de otros problemas a los que nos podemos enfrentar. Por ejemplo, cuando consideramos una solicitud/respuesta asincrónica de larga duración, tenemos que pensar qué hacer cuando se recibe la respuesta. ¿Vuelve al mismo nodo que inició la solicitud? Si es así, ¿qué pasa si ese nodo está inactivo?

Si no es así, ¿debo almacenar información en algún lugar para poder reaccionar en consecuencia? La asincronía de corta duración puede ser más fácil de administrar si tiene las API correctas, pero aun así, es una forma diferente de pensar para los programadores que están acostumbrados a las llamadas de mensajes sincrónicos dentro del proceso.

Es hora de una historia con moraleja. En 2006, estaba trabajando en la creación de un sistema de precios para un banco. Examinaríamos los eventos del mercado y determinaríamos qué elementos de una cartera debían cambiar de precio. Una vez que determinamos la lista de cosas para trabajar, las colocamos todas en una cola de mensajes. Estábamos haciendo uso de una cuadrícula para crear un grupo de trabajadores de fijación de precios, lo que nos permitía escalar hacia arriba y hacia abajo la granja de precios a pedido. Estos trabajadores utilizaron el patrón de consumo competitivo, cada uno engullendo mensajes lo más rápido posible hasta que no quedó nada para procesar.

El sistema estaba en funcionamiento y nos sentíamos bastante engreídos. Sin embargo, un día, justo después de lanzar un lanzamiento, nos encontramos con un problema desagradable. Nuestros trabajadores seguían muriendo. y muriendo y muriendo

Finalmente, localizamos el problema. Se había introducido un error por el cual cierto tipo de solicitud de precios causaba que un trabajador colapsara. Estábamos usando una cola de transacciones: cuando el trabajador moría, su bloqueo en la solicitud caducaba y la solicitud de precios se volvía a colocar en la cola, solo para que otro trabajador la recogiera y muriera. Este fue un ejemplo clásico de lo que Martin Fowler llama una **conmutación por error catastrófica**.

Aparte del error en sí, no pudimos especificar un límite máximo de reintentos para el trabajo en la cola. Solucionamos el error en sí y también configuramos un reintento máximo. Pero también nos dimos cuenta de que necesitábamos una forma de ver y, potencialmente, reproducir estos mensajes incorrectos. Terminamos teniendo que implementar un hospital de mensajes (o cola de mensajes fallidos), donde los mensajes se enviaban si fallaban. También creamos una interfaz de usuario para ver esos mensajes y volver a intentarlos si es necesario. Este tipo de problemas no son inmediatamente obvios si solo está familiarizado con la comunicación sincrónica punto a punto.

La complejidad asociada con las arquitecturas basadas en eventos y la programación asíncrona en general me lleva a creer que debe tener cuidado con el entusiasmo con el que comienza a adoptar estas ideas. Asegúrese de tener un buen monitoreo y considere seriamente el uso de ID de correlación, que le permiten rastrear las solicitudes a lo largo del proceso.

límites, como veremos en profundidad en el **Capítulo 8**.

También recomiendo encarecidamente *Patrones de integración empresarial* (Addison-Wesley), que contiene muchos más detalles sobre los diferentes patrones de programación que puede necesitar considerar en este espacio.

Servicios como máquinas de estado

Ya sea que elija convertirse en un ninja REST o quedarse con un mecanismo basado en RPC como SOAP, el concepto central del servicio como máquina de estado es poderoso. Hemos hablado antes (probablemente hasta la saciedad en este punto) acerca de que nuestros servicios están diseñados en torno a contextos limitados. Nuestro microservicio de cliente *posee* toda la lógica asociada con el comportamiento en este contexto.

Cuando un consumidor quiere cambiar un cliente, envía una solicitud apropiada al servicio de atención al cliente. El servicio de atención al cliente, en base a su lógica, decide si acepta o no esa solicitud. Nuestro servicio de atención al cliente controla todos los eventos del ciclo de vida asociados con el propio cliente. Queremos evitar servicios tontos y anémicos que son poco más que envoltorios CRUD. Si la decisión sobre qué cambios se pueden realizar en un cliente se filtra fuera del propio servicio de atención al cliente, estamos perdiendo cohesión.

Tener el ciclo de vida de los conceptos clave del dominio modelado explícitamente de esta manera es bastante poderoso. No solo tenemos un lugar para tratar las colisiones de estado (por ejemplo, alguien que intenta actualizar un cliente que ya se eliminó), sino que también tenemos un lugar para adjuntar comportamientos basados en esos cambios de estado.

Sigo pensando que REST sobre HTTP es una tecnología de integración mucho más sensata que muchas otras, pero elijas lo que elijas, ten esta idea en mente.

Extensiones reactivas

Las extensiones reactivas, a menudo abreviadas como Rx, son un mecanismo para componer los resultados de varias llamadas y ejecutar operaciones en ellas. Las llamadas en sí mismas pueden ser llamadas de bloqueo o no bloqueo. En esencia, Rx invierte los flujos tradicionales. En lugar de solicitar algunos datos y luego realizar operaciones en ellos, observa el resultado de una operación (o conjunto de operaciones) y reacciona cuando algo cambia. Algunas implementaciones de Rx le permiten realizar funciones en estos observables, como RxJava, que permite usar funciones tradicionales como mapa o filtro .

Las diversas implementaciones de Rx han encontrado un hogar muy feliz en los sistemas distribuidos. Nos permiten abstraer los detalles de cómo se realizan las llamadas y razonar sobre las cosas más fácilmente. Observo el resultado de una llamada a un servicio descendente. No me importa si fue una llamada con bloqueo o sin bloqueo, solo espero la respuesta y reacciono. La belleza es que puedo componer varias llamadas juntas, lo que facilita mucho el manejo de llamadas simultáneas a servicios posteriores.

A medida que se encuentre realizando más llamadas de servicio, especialmente cuando realice varias llamadas para realizar una sola operación, eche un vistazo a las extensiones reactivas para su pila de tecnología elegida. Es posible que se sorprenda de lo mucho más simple que puede volverse su vida.

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

Una de las siglas que escuchamos mucho los desarrolladores es SECO: no te repitas. Aunque su definición a veces se simplifica tratando de evitar la duplicación de código, DRY significa con mayor precisión que queremos evitar duplicar el *comportamiento y el conocimiento de nuestro sistema*.

Este es un consejo muy sensato en general. Tener muchas líneas de código que hacen lo mismo hace que su base de código sea más grande de lo necesario y, por lo tanto, más difícil de razonar. Cuando desea cambiar el comportamiento, y ese comportamiento se duplica en muchas partes de su sistema, es fácil olvidarse de todos los lugares donde debe realizar un cambio, lo que puede generar errores. Entonces, usar DRY como mantra, en general, tiene sentido.

DRY es lo que nos lleva a crear código que se puede reutilizar. Extraemos el código duplicado en abstracciones que luego podemos llamar desde múltiples lugares. ¡Quizás vayamos tan lejos como para crear una biblioteca compartida que podamos usar en todas partes! Este enfoque, sin embargo, puede ser engañosamente peligroso en una arquitectura de microservicio.

Una de las cosas que queremos evitar a toda costa es acoplar demasiado un microservicio y los consumidores de tal manera que cualquier pequeño cambio en el microservicio en sí mismo pueda causar cambios innecesarios en el consumidor. A veces, sin embargo, el uso de código compartido puede crear este mismo acoplamiento. Por ejemplo, en un cliente teníamos una biblioteca de objetos de dominio común que representaban las entidades centrales en uso en nuestro sistema. Esta biblioteca era utilizada por todos los servicios que teníamos. Pero cuando se hacía un cambio en uno de ellos, había que actualizar todos los servicios. Nuestro sistema se comunicaba a través de colas de mensajes, que también tenían que vaciarse de sus contenidos ahora *inválidos*, y ¡ay de ti si lo olvidabas!

Si su uso de código compartido alguna vez se filtra fuera de los límites de su servicio, ha introducido una posible forma de acoplamiento. Usar código común como bibliotecas de registro está bien, ya que son conceptos internos que son invisibles para el mundo exterior. RealEstate.com.au utiliza una plantilla de servicio personalizada para ayudar a impulsar la creación de nuevos servicios. En lugar de compartir este código, la compañía lo copia para cada nuevo servicio para garantizar que el acoplamiento no se filtre.

Mi regla general: no viole DRY dentro de un microservicio, pero no se preocupe por violar DRY en todos los servicios. Los males de un acoplamiento excesivo entre servicios son mucho peores que los problemas causados por la duplicación de código. Sin embargo, hay un caso de uso específico que vale la pena explorar más a fondo.

Bibliotecas de clientes

He hablado con más de un equipo que ha insistido en que crear bibliotecas de clientes para sus servicios es una parte esencial de la creación de servicios en primer lugar. El argumento es que esto facilita el uso de su servicio y evita la duplicación del código necesario para consumir el servicio en sí.

El problema, por supuesto, es que si las mismas personas crean tanto la API del servidor como la API del cliente, existe el peligro de que la lógica que debería existir en el servidor comience a filtrarse en el cliente. Debería saberlo: lo he hecho yo mismo. Cuanta más lógica se cuele en la biblioteca del cliente, más cohesión comienza a romperse y se encuentra con que tiene que cambiar varios clientes para implementar arreglos en su servidor. También limita las opciones de tecnología, especialmente si exige que se use la biblioteca del cliente.

Un modelo de bibliotecas cliente que me gusta es el de Amazon Web Services (AWS). Las llamadas de servicio web SOAP o REST subyacentes se pueden realizar directamente, pero todos terminan usando solo uno de los diversos kits de desarrollo de software (SDK) que existen, que proporcionan abstracciones sobre la API subyacente. Sin embargo, estos SDK están escritos por la comunidad o por personas de AWS distintas de las que trabajan en la propia API. Este grado de separación parece funcionar y evita algunos de los escollos de las bibliotecas cliente. Parte de la razón por la que esto funciona tan bien es que el cliente está a cargo de cuándo ocurre la actualización. Si usted mismo sigue el camino de las bibliotecas de clientes, asegúrese de que este sea el caso.

Netflix, en particular, pone especial énfasis en la biblioteca del cliente, pero me preocupa que la gente lo vea únicamente a través de la lente de evitar la duplicación de código. De hecho, las bibliotecas de clientes utilizadas por Netflix son tanto (si no más) para garantizar la confiabilidad y la escalabilidad de sus sistemas. Las bibliotecas de clientes de Netflix manejan el descubrimiento de servicios, los modos de falla, el registro y otros aspectos que en realidad no tienen que ver con la naturaleza del servicio en sí. Sin estos clientes compartidos, sería difícil garantizar que cada parte de las comunicaciones cliente/servidor se comportara bien en la escala masiva en la que opera Netflix. Su uso en Netflix ciertamente ha facilitado la puesta en marcha y ha aumentado la productividad al tiempo que garantiza que el sistema se comporte bien. Sin embargo, según al menos una persona de Netflix, con el tiempo esto ha llevado a un grado de acoplamiento entre el cliente y el servidor que ha sido problemático.

Si está pensando en el enfoque de la biblioteca del cliente, puede ser importante separar el código del cliente para manejar el protocolo de transporte subyacente, que puede tratar cosas como el descubrimiento y la falla del servicio, de las cosas relacionadas con el servicio de destino en sí. Decida si va a insistir o no en que se use la biblioteca del cliente, o si permitirá que las personas que usan diferentes pilas de tecnología hagan llamadas a la API subyacente.

Y finalmente, asegúrese de que los clientes estén a cargo de cuándo actualizar sus bibliotecas de clientes: ¡debemos asegurarnos de mantener la capacidad de lanzar nuestros servicios de forma independiente!

Acceso por Referencia

Una consideración que quiero tocar es cómo transmitimos información sobre nuestras entidades de dominio. Necesitamos adoptar la idea de que un microservicio abarcará el ciclo de vida de nuestras entidades de dominio central, como el Cliente. Ya hemos hablado de la importancia de que la lógica asociada al cambio de este Cliente se mantenga en el servicio de atención al cliente, y que si queremos cambiarlo tenemos que emitir una solicitud al servicio de atención al cliente.

Pero también se deduce que debemos considerar el servicio al cliente como la fuente de la verdad para los Clientes.

Cuando recuperamos un recurso de Cliente dado del servicio al cliente, podemos ver cómo se veía ese recurso cuando hicimos la solicitud. Es posible que después de que solicitamos ese recurso del Cliente, algo más lo haya cambiado. Lo que tenemos en efecto es un recuerdo de cómo se veía el recurso del Cliente una vez. Cuanto más nos aferremos a este recuerdo, mayor será la probabilidad de que este recuerdo sea falso. Por supuesto, si evitamos solicitar más datos de los necesarios, nuestros sistemas pueden volverse mucho más eficientes.

A veces este recuerdo es lo suficientemente bueno. Otras veces necesitas saber si ha cambiado. Entonces, ya sea que decida compartir un recuerdo de cómo se veía una vez una entidad, asegúrese de incluir también una referencia al recurso original para que se pueda recuperar el nuevo estado.

Consideremos el ejemplo en el que le pedimos al servicio de correo electrónico que envíe un correo electrónico cuando se haya enviado un pedido. Ahora podríamos enviar la solicitud al servicio de correo electrónico con la dirección de correo electrónico, el nombre y los detalles del pedido del cliente. Sin embargo, si el servicio de correo electrónico realmente está poniendo en cola estas solicitudes o extrayéndolas de una cola, las cosas podrían cambiar mientras tanto. Podría tener más sentido simplemente enviar un URI para los recursos Cliente y Pedido, y dejar que el servidor de correo electrónico los busque cuando sea el momento de enviar el correo electrónico.

Un gran contrapunto a esto surge cuando consideramos la colaboración basada en eventos. Con eventos, decimos que *esto sucedió*, pero necesitamos saber *qué* sucedió. Si recibimos actualizaciones debido al cambio de un recurso del Cliente, por ejemplo, podría ser valioso para nosotros saber cómo se veía el Cliente cuando ocurrió el evento. Siempre que también obtengamos una referencia a la entidad en sí misma para que podamos buscar su estado actual, entonces podemos obtener lo mejor de ambos mundos.

Hay otras compensaciones que hacer aquí, por supuesto, cuando estamos accediendo por referencia. Si siempre acudimos al servicio de atención al cliente a mirar la información asociada a un determinado Cliente, la carga en el servicio de atención al cliente puede ser demasiado grande. Si proporcionamos información adicional cuando se recupera el recurso, permitiéndonos saber en qué momento estuvo el recurso en el estado dado y quizás cuánto tiempo podemos considerar que esta información está actualizada, entonces podemos hacer mucho con el almacenamiento en caché para reducir la carga. HTTP nos brinda gran parte de este soporte listo para usar con una amplia variedad de controles de caché, algunos de los cuales discutiremos con más detalle

en el Capítulo 11.

Otro problema es que es posible que algunos de nuestros servicios no necesiten conocer todo el recurso del Cliente , y al insistir en que lo busquen, estamos aumentando potencialmente el acoplamiento. Se podría argumentar, por ejemplo, que nuestro servicio de correo electrónico debería ser más tonto y que simplemente deberíamos enviarle la dirección de correo electrónico y el nombre del cliente. No hay una regla estricta aquí, pero tenga mucho cuidado al pasar datos en solicitudes cuando no sabe si están actualizados.

Versionado

En cada charla que he hecho sobre microservicios, me preguntan *cómo se hace el control de versiones*. Las personas tienen la preocupación legítima de que eventualmente tendrán que hacer un cambio en la interfaz de un servicio y quieren saber cómo administrarlo. Analicemos un poco el problema y veamos los diversos pasos que podemos tomar para manejarlo.

Aplazarlo el mayor tiempo posible

La mejor manera de reducir el impacto de hacer cambios importantes es evitar hacerlos en primer lugar. Puede lograr gran parte de esto eligiendo la tecnología de integración correcta, como hemos discutido a lo largo de este capítulo. La integración de bases de datos es un gran ejemplo de tecnología que puede hacer que sea muy difícil evitar cambios importantes. REST, por otro lado, ayuda porque es menos probable que los cambios en los detalles de la implementación interna resulten en un cambio en la interfaz del servicio.

Otra clave para aplazar un cambio radical es fomentar el buen comportamiento de sus clientes y evitar que, en primer lugar, se vinculen demasiado a sus servicios. Consideremos nuestro servicio de correo electrónico, cuyo trabajo es enviar correos electrónicos a nuestros clientes de vez en cuando. Se le pide que envíe un correo electrónico de pedido enviado al cliente con la ID 1234. Se apaga y recupera al cliente con esa ID, y obtiene algo como la respuesta que se muestra en el [Ejemplo 4-3](#).

Ejemplo 4-3. Ejemplo de respuesta del servicio de atención al cliente

```
<cliente>
  <primer nombre>Sam</primer
  nombre> <apellido>Newman</
  apellido> <email>sam@magpiebrain.com</
  email> <número de teléfono>555-1234-5678</número de teléfono>
</cliente>
```

Ahora, para enviar el correo electrónico, solo necesitamos los campos de nombre , apellido y correo electrónico . No necesitamos saber el número de teléfono. Simplemente queremos sacar los campos que nos interesan e ignorar el resto. Cierta tecnología de vinculación, especialmente la utilizada por lenguajes fuertemente tipados, puede intentar vincular *todos los* campos, ya sea que el consumidor los desee o no.

¿Qué pasa si nos damos cuenta de que nadie está usando el número de teléfono y decidimos eliminarlo?

Esto podría hacer que los consumidores se rompan innecesariamente.

Del mismo modo, ¿qué pasaría si quisiéramos reestructurar nuestro objeto Cliente para admitir más detalles, tal vez agregando alguna estructura adicional como en el [Ejemplo 4-4](#)? Los datos que desea nuestro servicio de correo electrónico todavía están allí, y aún con el mismo nombre, pero si nuestro código hace suposiciones muy explícitas sobre dónde se almacenarán los campos de nombre y apellido , entonces podría fallar nuevamente. En este caso, podríamos usar XPath para extraer los campos que nos interesan, lo que nos permitiría ser ambivalentes acerca de dónde están los campos, siempre que podamos encontrarlos. Este patrón, de implementar un lector capaz de ignorar los cambios que no nos interesan, es lo que Martin Fowler llama [Lector Tolerante](#).

Ejemplo 4-4. Un recurso de cliente reestructurado: todos los datos siguen ahí, pero ¿pueden encontrarlos nuestros consumidores?

```
<cliente>
  <naming>
    <firstname>Sam</firstname>
    <lastname>Newman</lastname>
    <nickname>Magpiebrain</nickname>
    <fullname>Sam "Magpiebrain" Newman</fullname> </
  naming> <email>sam @magpiebrain.com</correo electrónico>
```

</cliente>

El ejemplo de un cliente que trata de ser lo más flexible posible al consumir un servicio demuestra **la Ley de Postel** (también conocida como el *principio de robustez*), que establece: "Sea conservador en lo que hace, sea liberal en lo que acepta de los demás". El contexto original de esta pieza de sabiduría fue la interacción de dispositivos a través de redes, donde debe esperar que sucedan todo tipo de cosas extrañas. En el contexto de nuestra interacción de solicitud/respuesta, puede llevarnos a hacer todo lo posible para permitir que el servicio que se consume cambie sin requerir que nosotros cambiemos.

Atrapa los cambios importantes antes de tiempo

Es fundamental asegurarse de que detectamos los cambios que afectarán a los consumidores lo antes posible, porque incluso si elegimos la mejor tecnología posible, aún pueden ocurrir rupturas. Estoy firmemente a favor del uso de contratos impulsados por el consumidor, que trataremos en el **Capítulo 7**, para ayudar a detectar estos problemas desde el principio. Si admite varias bibliotecas de clientes diferentes, la ejecución de pruebas con cada biblioteca compatible con el servicio más reciente es otra técnica que puede ayudar. Una vez que se da cuenta de que va a quebrar a un consumidor, tiene la opción de tratar de evitar la ruptura por completo o aceptarla y comenzar a tener las conversaciones correctas con las personas que se ocupan de los servicios de consumo.

Usar versiones semánticas

¿No sería genial si, como cliente, pudiera mirar solo el número de versión de un servicio y saber si puede integrarse con él? El control de *versiones semántico* es una especificación que permite precisamente eso. Con el control de versiones semántico, cada número de versión tiene el formato MAYOR.MENOR.PARCHÉ. Cuando el número PRINCIPAL aumenta, significa que se han realizado cambios incompatibles con versiones anteriores. Cuando MINOR aumenta, se ha agregado una nueva funcionalidad que debería ser compatible con versiones anteriores. Finalmente, un cambio en PATCH indica que se han realizado correcciones de errores en la funcionalidad existente.

Para ver cuán útil puede ser el control de versiones semántico, veamos un caso de uso simple. Nuestra aplicación de soporte técnico está diseñada para funcionar con la versión 1.2.0 del servicio de atención al cliente. Si se agrega una nueva función, lo que hace que el servicio de atención al cliente cambie a 1.3.0, nuestra aplicación de soporte técnico no debería ver ningún cambio en el comportamiento y no debería esperarse que realice ningún cambio. Sin embargo, no podemos garantizar que podamos trabajar con la versión 1.1.0 del servicio de atención al cliente, ya que podemos confiar en la funcionalidad agregada en la versión 1.2.0. También podríamos esperar tener que hacer cambios en nuestra aplicación si sale una nueva versión 2.0.0 del servicio al cliente.

Puede decidir tener una versión semántica para el servicio, o incluso para un extremo individual en un servicio si los coexiste, como se detalla en la siguiente sección.

Este esquema de control de versiones nos permite empaquetar mucha información y expectativas en solo tres campos. La especificación completa describe en términos muy simples las expectativas que los clientes pueden tener sobre los cambios en estos números y puede simplificar el proceso de comunicación sobre si los cambios deberían afectar a los consumidores. Desafortunadamente, no he visto que este enfoque se use lo suficiente en el contexto de los sistemas distribuidos.

Coexisten diferentes puntos finales

Si hemos hecho todo lo posible para evitar la introducción de un cambio radical en la interfaz, nuestro siguiente trabajo es limitar el impacto. Lo que queremos evitar es obligar a los consumidores a actualizar al unísono con nosotros, ya que siempre queremos mantener la capacidad de lanzar microservicios de forma independiente entre sí. Un enfoque que he usado con éxito para manejar esto es coexistir tanto la interfaz antigua como la nueva en el mismo servicio en ejecución. Entonces, si queremos lanzar un cambio importante, implementamos una nueva versión del servicio que expone tanto la versión anterior como la nueva del punto final.

Esto nos permite sacar el nuevo microservicio lo antes posible, junto con la nueva interfaz, pero da tiempo para que los consumidores se cambien. Una vez que todos los consumidores ya no utilicen el punto final anterior, puede eliminarlo junto con cualquier código asociado, como se muestra en la [Figura 4-5](#).

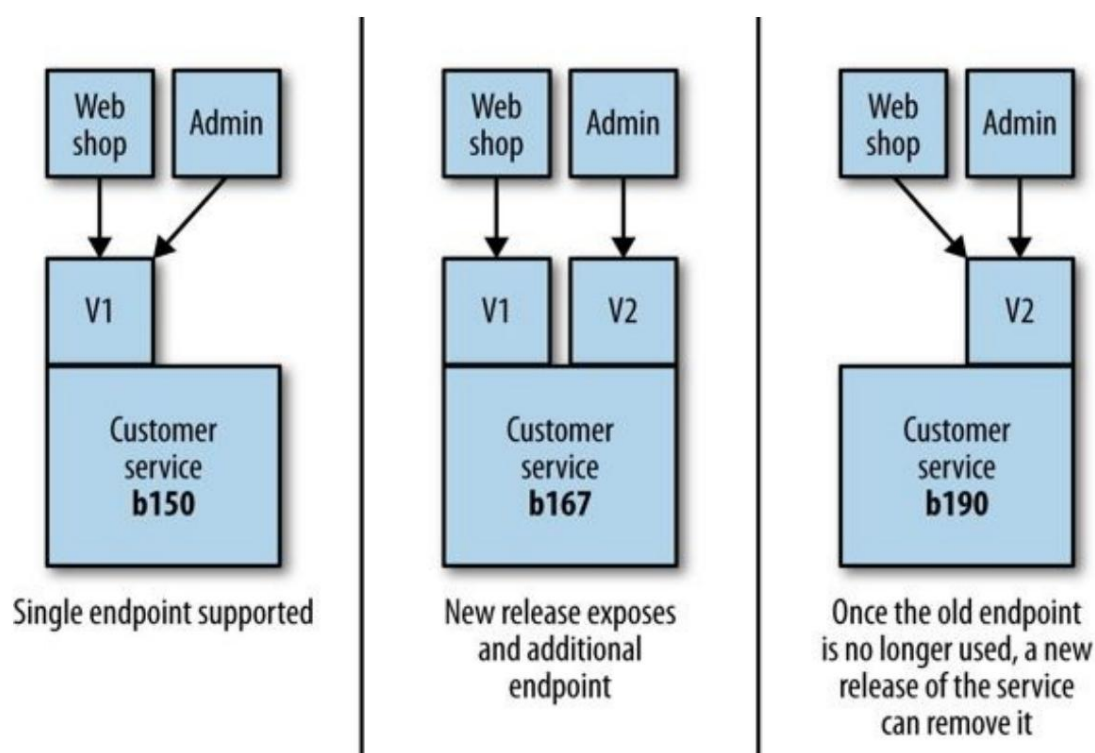


Figura 4-5. La coexistencia de diferentes versiones de terminales permite a los consumidores migrar gradualmente

La última vez que usé este enfoque, nos habíamos metido en un lío con la cantidad de consumidores que teníamos y la cantidad de cambios importantes que habíamos realizado. Esto significaba que en realidad estábamos coexistiendo con tres versiones diferentes del punto final. ¡Esto no es algo que recomendaría! Mantener todo el código y las pruebas asociadas requeridas para garantizar que todo funcionara era absolutamente una carga adicional. Para que esto sea más manejable, transformamos internamente todas las solicitudes al extremo V1 en una solicitud V2, y luego las solicitudes V2 al extremo V3. Esto significaba que podíamos delinear claramente qué código se retiraría cuando murieran los extremos anteriores.

Este es, en efecto, un ejemplo del patrón de expansión y contracción, que nos permite incorporar cambios graduales. *Expandimos* las capacidades que ofrecemos, admitiendo tanto las antiguas como las nuevas .

formas de hacer algo. Una vez que los antiguos consumidores hacen las cosas de la nueva manera, *contratamos* nuestra API, eliminando la funcionalidad anterior.

Si va a coexistir puntos finales, necesita una forma para que las personas que llaman enruten sus solicitudes en consecuencia. Para los sistemas que usan HTTP, he visto que esto se hace con ambos números de versión en los encabezados de solicitud y también en el URI mismo, por ejemplo, */v1/cliente/* o */v2/cliente/*. Estoy dividido en cuanto a qué enfoque tiene más sentido. Por un lado, me gusta que los URI sean opacos para disuadir a los clientes de codificar las plantillas de URI, pero por otro lado, este enfoque hace que las cosas sean muy obvias y puede simplificar el enrutamiento de solicitudes.

Para RPC, las cosas pueden ser un poco más complicadas. He manejado esto con búferes de protocolo colocando mis métodos en diferentes espacios de nombres, por ejemplo, *v1.createCustomer* y *v2.createCustomer*, pero cuando intenta admitir diferentes versiones de los mismos tipos que se envían a través de la red, esto puede volverse realmente doloroso. .

Usar múltiples versiones de servicio simultáneas

Otra solución de control de versiones citada a menudo es tener diferentes versiones del servicio en vivo a la vez, y que los consumidores mayores enruten su tráfico a la versión anterior, y las versiones más nuevas vean la nueva, como se muestra en la [Figura 4-6](#). Este es el enfoque que utiliza Netflix con moderación en situaciones en las que el costo de cambiar a los consumidores mayores es demasiado alto, especialmente en casos excepcionales en los que los dispositivos heredados todavía están vinculados a versiones anteriores de la API. Personalmente, no soy fanático de esta idea y entiendo por qué Netflix la usa raramente. Primero, si necesito corregir un error interno en mi servicio, ahora tengo que corregir e implementar dos conjuntos de servicios diferentes. Esto probablemente significaría que tengo que bifurcar el código base para mi servicio, y esto siempre es problemático. En segundo lugar, significa que necesito inteligencia para dirigir a los consumidores al microservicio correcto. Este comportamiento inevitablemente termina en algún lugar del middleware o en un montón de scripts nginx, lo que dificulta razonar sobre el comportamiento del sistema. Finalmente, considere cualquier estado persistente que nuestro servicio pueda administrar. Los clientes creados por cualquiera de las versiones del servicio deben almacenarse y hacerse visibles para todos los servicios, sin importar qué versión se utilizó para crear los datos en primer lugar. Esto puede ser una fuente adicional de complejidad.

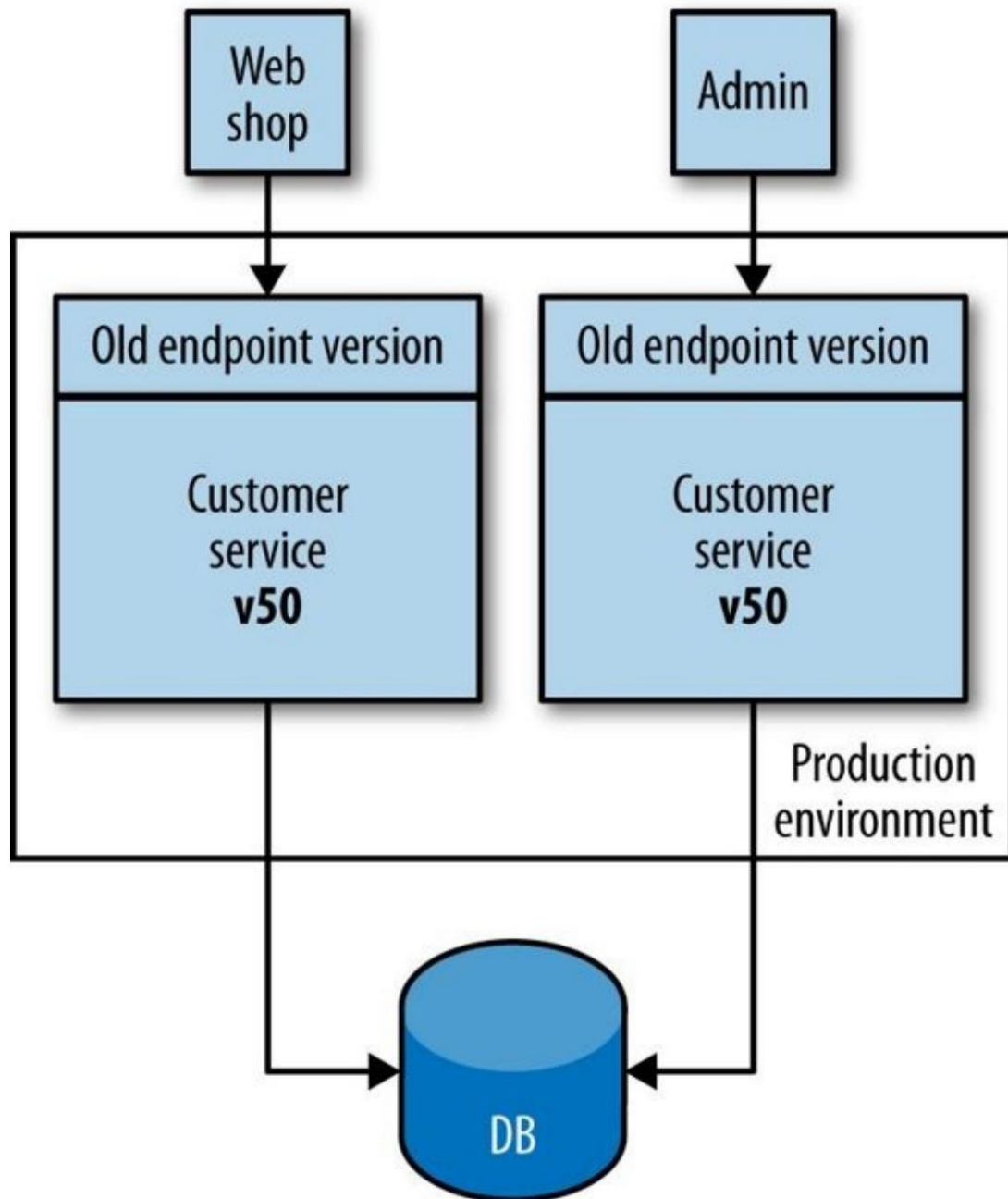


Figura 4-6. Ejecución de varias versiones del mismo servicio para admitir puntos finales antiguos

Las versiones de servicio simultáneas coexistentes durante un período corto de tiempo pueden tener mucho sentido, especialmente cuando está haciendo cosas como implementaciones azul/verde o lanzamientos controlados (hablaremos más sobre estos patrones en el [Capítulo 7](#)). En estas situaciones, podemos estar coexistiendo versiones solo durante unos minutos o quizás horas, y normalmente solo tendremos dos versiones diferentes del servicio presentes al mismo tiempo. Cuanto más tarde en actualizar a los consumidores a la versión más nueva y lanzarla, más debería buscar la coexistencia de diferentes puntos finales en el mismo microservicio en lugar de coexistir versiones completamente diferentes. Sigo sin estar convencido de que este trabajo valga la pena para el proyecto promedio.

Interfaces de usuario

Hasta ahora, no hemos tocado realmente el mundo de la interfaz de usuario. Es posible que algunos de ustedes simplemente proporcionen una API fría, dura y clínica a sus clientes, pero muchos de nosotros deseamos crear interfaces de usuario hermosas y funcionales que deleitarán a nuestros clientes. Pero realmente necesitamos pensar en ellos en el contexto de la integración. Después de todo, la interfaz de usuario es donde uniremos todos estos microservicios en algo que tenga sentido para nuestros clientes.

En el pasado, cuando comencé con la informática, principalmente hablábamos de clientes grandes y voluminosos que se ejecutaban en nuestros escritorios. Pasé muchas horas con Motif y luego con Swing tratando de hacer que mi software fuera lo más agradable posible. A menudo, estos sistemas eran solo para la creación y manipulación de archivos locales, pero muchos de ellos tenían un componente del lado del servidor. Mi primer trabajo en ThoughtWorks involucró la creación de un sistema de punto de venta electrónico basado en Swing que era solo una parte de una gran cantidad de piezas móviles, la mayoría de las cuales estaban en el servidor.

Luego vino la Web. En cambio, comenzamos a pensar en nuestras interfaces de usuario como *delgadas*, con más lógica en el lado del servidor. Al principio, nuestros programas del lado del servidor representaban la página completa y la enviaban al navegador del cliente, que hacía muy poco. Cualquier interacción se manejó en el lado del servidor, a través de GET y POST activados por el usuario al hacer clic en enlaces o completar formularios.

Con el tiempo, JavaScript se convirtió en una opción más popular para agregar un comportamiento dinámico a la interfaz de usuario basada en el navegador, y ahora se podría argumentar que algunas aplicaciones son tan *grandes* como los antiguos clientes de escritorio.

Hacia lo digital

En los últimos años, las organizaciones han comenzado a dejar de pensar que la web o los dispositivos móviles deben tratarse de manera diferente; en cambio, están pensando en lo digital de manera más holística. ¿Cuál es la mejor manera para que nuestros clientes utilicen los servicios que ofrecemos? ¿Y qué le hace eso a la arquitectura de nuestro sistema? La comprensión de que no podemos predecir exactamente cómo un cliente podría terminar interactuando con nuestra empresa ha impulsado la adopción de API más granulares, como las que ofrecen los microservicios. Al combinar las capacidades que nuestros servicios exponen de diferentes maneras, podemos seleccionar diferentes experiencias para nuestros clientes en su aplicación de escritorio, dispositivo móvil, dispositivo portátil o incluso en forma física si visitan nuestra tienda física.

Así que piense en las interfaces de usuario como capas de composición: lugares donde entrelazamos los diversos hilos de las capacidades que ofrecemos. Entonces, con eso en mente, ¿cómo juntamos todos estos hilos?

Restricciones

Las restricciones son las diferentes formas en que nuestros usuarios interactúan con nuestro sistema. En una aplicación web de escritorio, por ejemplo, consideramos restricciones como qué navegador usan los visitantes o su resolución. Pero los dispositivos móviles han traído toda una serie de nuevas limitaciones. La forma en que nuestras aplicaciones móviles se comunican con el servidor puede tener un impacto. No se trata solo de preocupaciones puras de ancho de banda, donde las limitaciones de las redes móviles pueden desempeñar un papel. Diferentes tipos de interacciones pueden agotar la vida útil de la batería, lo que lleva a algunos clientes cruzados.

La naturaleza de las interacciones también cambia. No puedo hacer clic derecho fácilmente en una tableta. En un teléfono móvil, es posible que desee diseñar mi interfaz para que se use principalmente con una mano, y que la mayoría de las operaciones se controlen con el pulgar. En otros lugares, podría permitir que las personas interactúen con los servicios a través de SMS en lugares donde el ancho de banda es escaso; el uso de SMS como interfaz es enorme en el sur global, por ejemplo.

Por lo tanto, aunque nuestros servicios principales, nuestra oferta principal, pueden ser los mismos, necesitamos una forma de adaptarlos a las diferentes limitaciones que existen para cada tipo de interfaz. Cuando observamos diferentes estilos de composición de la interfaz de usuario, debemos asegurarnos de que aborden este desafío. Veamos algunos modelos de interfaces de usuario para ver cómo se puede lograr esto.

Composición API

Suponiendo que nuestros servicios ya hablan XML o JSON entre sí a través de HTTP, una opción obvia disponible para nosotros es hacer que nuestra interfaz de usuario interactúe directamente con estas API, como en la **Figura 4-7**. Una interfaz de usuario basada en la web podría usar solicitudes GET de JavaScript para recuperar datos o solicitudes POST para cambiarlos. Incluso para aplicaciones móviles nativas, iniciar comunicaciones HTTP es bastante sencillo. Luego, la interfaz de usuario necesitaría crear los diversos componentes que componen la interfaz, manejar la sincronización del estado y similares con el servidor. Si usáramos un protocolo binario para la comunicación de servicio a servicio, esto sería más difícil para los clientes basados en la web, pero podría estar bien para los dispositivos móviles nativos.

Hay un par de inconvenientes con este enfoque. Primero, tenemos poca capacidad para adaptar las respuestas a diferentes tipos de dispositivos. Por ejemplo, cuando recupero el registro de un cliente, ¿debo recuperar todos los mismos datos para una tienda móvil que para una aplicación de soporte técnico? Una solución a este enfoque es permitir que los consumidores especifiquen qué campos retirar cuando realizan una solicitud, pero esto supone que cada servicio admite esta forma de interacción.

Otra pregunta clave: ¿quién crea la interfaz de usuario? Las personas que se encargan de los servicios se eliminan de la forma en que sus servicios se muestran a los usuarios; por ejemplo, si otro equipo está creando la interfaz de usuario, podríamos estar retrocediendo a los malos viejos tiempos de la arquitectura en capas, donde incluso hacer pequeños cambios requiere un cambio. Solicitudes a varios equipos.

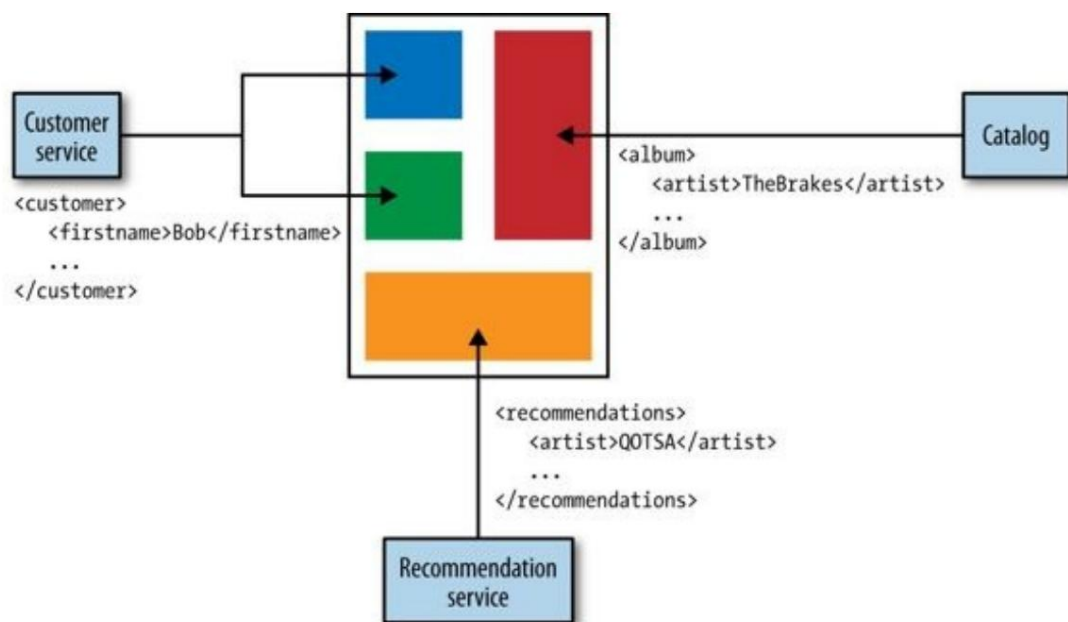


Figura 4-7. Uso de varias API para presentar una interfaz de usuario

Esta comunicación también podría ser bastante habladora. ¡Abrir muchas llamadas directamente a los servicios puede ser bastante intensivo para los dispositivos móviles y podría ser un uso muy ineficiente del plan móvil de un cliente! Tener una puerta de enlace API puede ayudar aquí, ya que podría exponer llamadas que agregan varias llamadas subyacentes, aunque eso en sí mismo puede tener algunas desventajas que exploraremos en breve.

Composición de fragmentos de interfaz de usuario

En lugar de que nuestra interfaz de usuario haga llamadas a la API y mapee todo de nuevo a los controles de la interfaz de usuario, podríamos hacer que nuestros servicios proporcionen partes de la interfaz de usuario directamente y luego extraer estos fragmentos para crear una interfaz de usuario, como en la [figura 4-8](#). Imagine, por ejemplo, que el servicio de recomendaciones proporciona un widget de recomendaciones que se combina con otros controles o fragmentos de la interfaz de usuario para crear una interfaz de usuario general. Puede aparecer como un cuadro en una página web junto con otros contenido.

Una variación de este enfoque que puede funcionar bien es ensamblar una serie de partes de grano más grueso de una interfaz de usuario. Entonces, en lugar de crear pequeños widgets, está ensamblando paneles completos de una aplicación cliente gruesa, o quizás un conjunto de páginas para un sitio web.

Estos fragmentos de grano más grueso se sirven desde aplicaciones del lado del servidor que, a su vez, realizan las llamadas API apropiadas. Este modelo funciona mejor cuando los fragmentos se alinean bien con la propiedad del equipo. Por ejemplo, quizás el equipo que se ocupa de la gestión de pedidos en la tienda de música sirve todas las páginas asociadas con la gestión de pedidos.

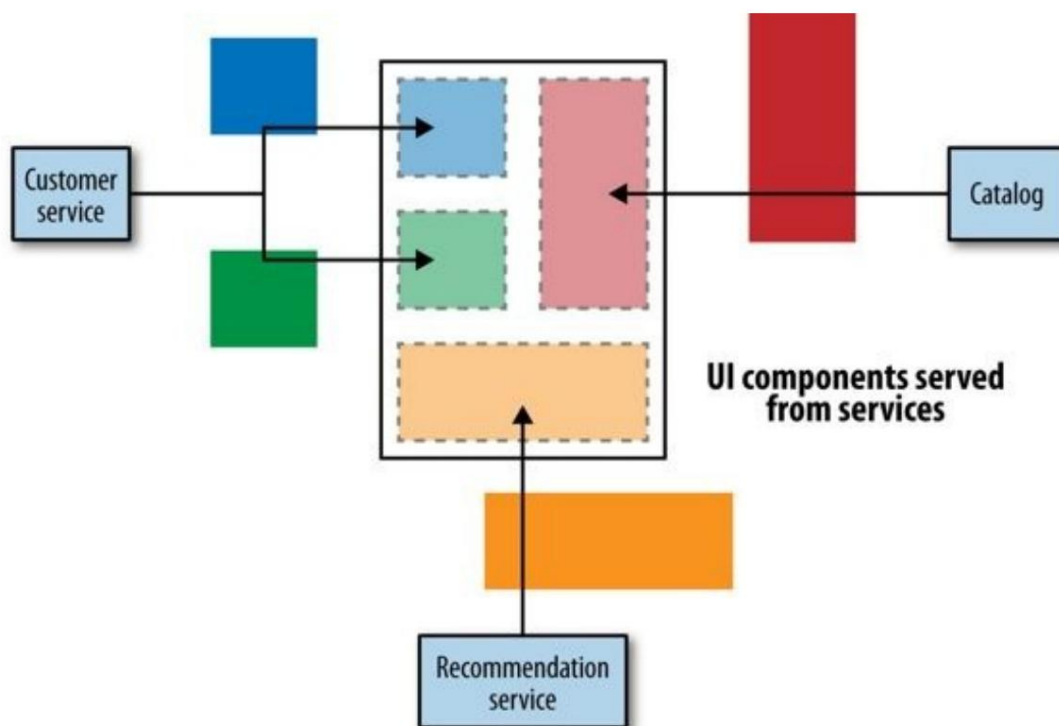


Figura 4-8. Servicios que sirven directamente componentes de interfaz de usuario para el ensamblaje

Todavía necesita algún tipo de capa de ensamblaje para juntar estas partes. Esto podría ser tan simple como algunas plantillas del lado del servidor o, donde cada conjunto de páginas proviene de una aplicación diferente, tal vez necesite un enrutamiento de URI inteligente.

Una de las ventajas clave de este enfoque es que el mismo equipo que realiza cambios en los servicios también puede estar a cargo de realizar cambios en esas partes de la interfaz de usuario. Nos permite sacar los cambios más rápido. Pero hay algunos problemas con este enfoque.

Primero, garantizar la consistencia de la experiencia del usuario es algo que debemos abordar. Usuarios

desea tener una experiencia fluida, no sentir que las diferentes partes de la interfaz funcionan de manera diferente o presentan un lenguaje de diseño diferente. Sin embargo, existen técnicas para evitar este problema, como las guías de estilo de vida, donde se pueden compartir activos como componentes HTML, CSS e imágenes para ayudar a brindar cierto nivel de coherencia.

Otro problema es más difícil de tratar. ¿Qué sucede con las aplicaciones nativas o los clientes pesados? No podemos servir componentes de interfaz de usuario. Podríamos usar un enfoque híbrido y usar aplicaciones nativas para ofrecer componentes HTML, pero se ha demostrado una y otra vez que este enfoque tiene desventajas. Entonces, si necesita una experiencia nativa, tendremos que recurrir a un enfoque en el que la aplicación frontend realiza llamadas a la API y maneja la interfaz de usuario. Pero incluso si consideramos las interfaces de usuario solo web, aún podemos querer tratamientos muy diferentes para diferentes tipos de dispositivos. La creación de componentes receptivos puede ayudar, por supuesto.

Hay un problema clave con este enfoque que no estoy seguro de que pueda resolverse. A veces, las capacidades que ofrece un servicio no encajan perfectamente en un widget o una página. Claro, es posible que desee mostrar recomendaciones en un cuadro en una página de nuestro sitio web, pero ¿qué pasa si quiero entretelar recomendaciones dinámicas en otro lugar? Cuando busco, quiero que el tipo de texto adelante active automáticamente nuevas recomendaciones, por ejemplo. Cuanto más transversal sea una forma de interacción, menos probable es que este modelo se ajuste y más probable es que volvamos a hacer llamadas a la API.

Backends para Frontends

Una solución común al problema de las interfaces comunicativas con los servicios de back-end, o la necesidad de variar el contenido para diferentes tipos de dispositivos, es tener un extremo de agregación del lado del servidor o una puerta de enlace API. Esto puede ordenar múltiples llamadas de back-end, variar y agregar contenido si es necesario para diferentes dispositivos y servirlo, como vemos en la [Figura 4-9](#). He visto que este enfoque conduce al desastre cuando estos puntos finales del lado del servidor se convierten en capas gruesas con demasiado comportamiento. Terminan siendo administrados por equipos separados y son otro lugar donde la lógica tiene que cambiar cada vez que cambia alguna funcionalidad.

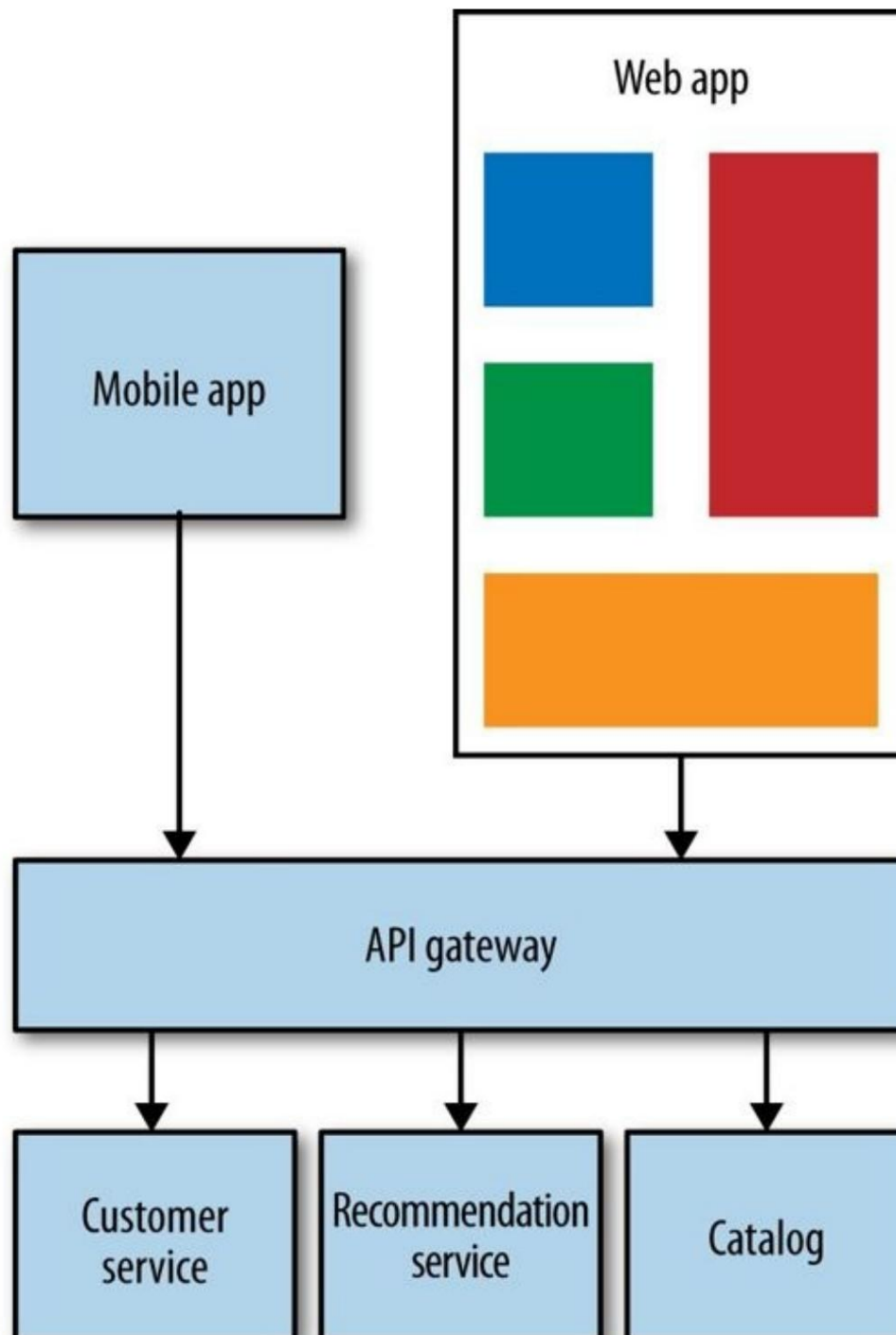


Figura 4-9. Uso de una sola puerta de enlace monolítica para manejar llamadas hacia/desde IU

El problema que puede ocurrir es que normalmente tendremos una capa gigante para todos nuestros servicios. Esto lleva a que todo se junte, y de repente comenzamos a perder el aislamiento de nuestras diversas interfaces de usuario, lo que limita nuestra capacidad para lanzarlas de forma independiente. Un modelo que prefiero y que he visto que funciona bien es restringir el uso de estos backends a una interfaz de usuario o aplicación específica, como vemos en la [Figura 4-10](#).

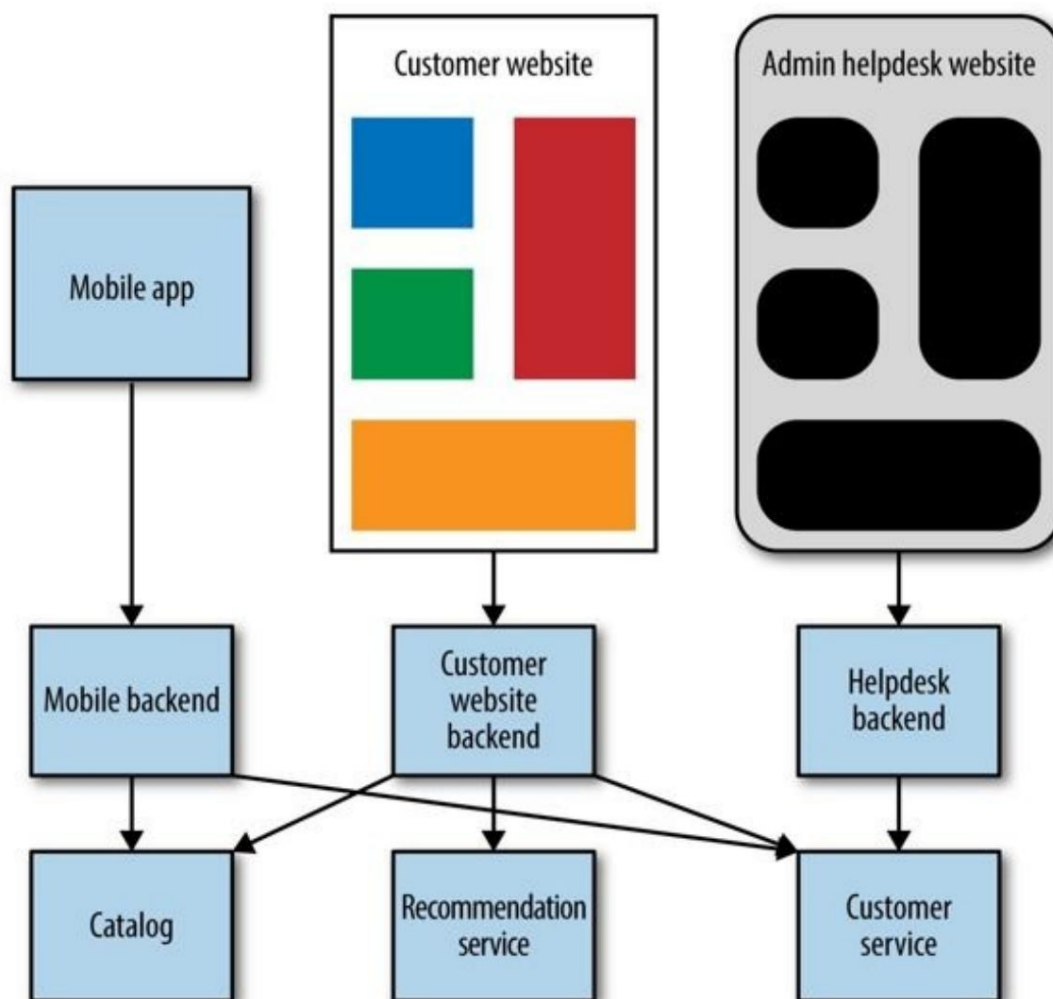


Figura 4-10. Uso de backends dedicados para frontends

Este patrón a veces se denomina *backends para frontends (BFF)*. Permite que el equipo se centre en cualquier interfaz de usuario dada para manejar también sus propios componentes del lado del servidor. Puede ver estos backends como partes de la interfaz de usuario que están integradas en el servidor.

Algunos tipos de interfaz de usuario pueden necesitar un espacio mínimo del lado del servidor, mientras que otros pueden necesitar mucho más. Si necesita una capa de autenticación y autorización de API, esta puede ubicarse entre nuestros BFF y nuestras UI. Exploraremos esto más en el [Capítulo 9](#).

El peligro con este enfoque es el mismo que con cualquier capa de agregación; puede asumir la lógica que no debería. La lógica comercial para las diversas capacidades que utilizan estos backends debe permanecer en los propios servicios. Estos BFF solo deben contener un comportamiento específico para brindar una experiencia de usuario particular.

Un enfoque híbrido

Muchas de las opciones antes mencionadas no necesitan ser de talla única. Pude ver a una organización adoptando el enfoque de ensamblaje basado en fragmentos para crear un sitio web, pero utilizando un enfoque de backends para frontends cuando se trata de su aplicación móvil. El punto clave es que necesitamos conservar la cohesión de las capacidades subyacentes que ofrecemos a nuestros usuarios. Necesitamos asegurarnos de que la lógica asociada con el pedido de música o el cambio de detalles de los clientes viva dentro de los servicios que manejan esas operaciones, y no se difumine en todo nuestro sistema. Evitar la trampa de poner demasiado comportamiento en las capas intermedias es un acto de equilibrio complicado.

Integración con software de terceros

Hemos analizado enfoques para romper los sistemas existentes que están bajo nuestro control, pero ¿qué pasa cuando no podemos cambiar las cosas con las que hablamos? Por muchas razones válidas, las organizaciones para las que trabajamos compran software estándar comercial (COTS) o utilizan ofertas de software como servicio (SaaS) sobre las que tenemos poco control. Entonces, ¿cómo nos integramos con ellos de manera sensata?

Si está leyendo este libro, probablemente trabaje en una organización que escribe código. Puede escribir software para sus propios fines internos o para un cliente externo, o ambos.

No obstante, incluso si es una organización con la capacidad de crear una cantidad significativa de software personalizado, seguirá utilizando productos de software proporcionados por terceros, ya sean comerciales o de código abierto. ¿Por qué es esto?

En primer lugar, es casi seguro que su organización tiene una mayor demanda de software de la que puede satisfacer internamente. Piense en todos los productos que utiliza, desde herramientas de productividad de oficina como Excel hasta sistemas operativos y sistemas de nómina. Crear todos esos para su propio uso sería una empresa gigantesca. En segundo lugar, y lo más importante, ¡no sería rentable! El costo de construir su propio sistema de correo electrónico, por ejemplo, probablemente eclipsará el costo de usar una combinación existente de servidor de correo y cliente, incluso si opta por opciones comerciales.

Mis clientes a menudo luchan con la pregunta "¿Debo construir o debo comprar?" En general, el consejo que mis colegas y yo damos al tener esta conversación con la organización empresarial promedio se reduce a: "Construya si es exclusivo de lo que hace y puede considerarse un activo estratégico; compre si su uso de la herramienta no es tan especial".

Por ejemplo, la organización promedio no consideraría su sistema de nómina como un activo estratégico. A la gente en general se le paga lo mismo en todo el mundo. Del mismo modo, la mayoría de las organizaciones tienden a comprar sistemas de administración de contenido (CMS) listos para usar, ya que el uso de dicha herramienta no se considera algo clave para su negocio. Por otro lado, participé desde el principio en la reconstrucción del sitio web de *The Guardian*, y allí se tomó la decisión de construir un sistema de administración de contenido a medida, ya que era fundamental para el negocio del periódico.

Por lo tanto, la idea de que ocasionalmente nos encontraremos con software comercial de terceros es sensata y bienvenida. Sin embargo, muchos de nosotros terminamos maldiciendo algunos de estos sistemas. ¿Porqué es eso?

Falta de control

Un desafío asociado con la integración y la ampliación de las capacidades de los productos COTS, como las herramientas CMS o SaaS, es que, por lo general, muchas de las decisiones técnicas se han tomado por usted. ¿Cómo se integra con la herramienta? Esa es una decisión del proveedor. ¿Qué lenguaje de programación puede utilizar para ampliar la herramienta? Hasta el vendedor. ¿Puede almacenar la configuración de la herramienta en el control de versiones y reconstruirla desde cero para permitir la integración continua de las personalizaciones? Depende de las elecciones que haga el proveedor.

Si tiene suerte, lo fácil (o difícil) que es trabajar con la herramienta desde el punto de vista del desarrollo se ha considerado como parte del proceso de selección de la herramienta. Pero incluso entonces, está cediendo cierto nivel de control a una parte externa. El truco es volver a llevar el trabajo de integración y personalización a sus términos.

personalización

Muchas herramientas que compran las organizaciones empresariales se venden por su capacidad de personalizarse en gran medida *solo para usted*. ¡Tener cuidado! A menudo, debido a la naturaleza de la cadena de herramientas a la que tiene acceso, el costo de la personalización puede ser más costoso que construir algo a medida desde cero. Si ha decidido comprar un producto pero las capacidades particulares que proporciona no son tan especiales para usted, podría tener más sentido cambiar la forma en que funciona su organización en lugar de embarcarse en una personalización compleja.

Los sistemas de gestión de contenidos son un gran ejemplo de este peligro. He trabajado con varios CMS que, por diseño, no admiten la integración continua, que tienen API terribles y para los cuales incluso una actualización de punto menor en la herramienta subyacente puede romper cualquier personalización que haya realizado.

Salesforce es especialmente problemático en este sentido. Durante muchos años, ha impulsado su plataforma Force.com, que requiere el uso de un lenguaje de programación, Apex, ¡que solo existe dentro del ecosistema Force.com!

Espagueti de integración

Otro desafío es cómo se integra con la herramienta. Como comentamos anteriormente, es importante pensar detenidamente en cómo se integra entre los servicios e, idealmente, desea estandarizar en una pequeña cantidad de tipos de integración. Pero si un producto decide usar un protocolo binario propietario, otro algo de SOAP y otro XML-RPC, ¿qué le queda? Peor aún son las herramientas que le permiten acceder directamente a sus almacenes de datos subyacentes, lo que lleva a los mismos problemas de acoplamiento que discutimos anteriormente.

En sus propios términos

Los productos COTS y SAAS tienen absolutamente su lugar, y no es factible (o sensato) para la mayoría de nosotros construir todo desde cero. Entonces, ¿cómo resolvemos estos desafíos?

La clave es hacer que las cosas vuelvan a sus propios términos.

La idea central aquí es realizar cualquier personalización en una plataforma que controle y limite la cantidad de consumidores diferentes de la herramienta en sí. Para explorar esta idea en detalle, veamos un par de ejemplos.

Ejemplo: CMS como servicio

En mi experiencia, el CMS es uno de los productos más utilizados que necesita ser personalizado o integrado. La razón de esto es que, a menos que desee un sitio estático básico, la organización empresarial promedio desea enriquecer la funcionalidad de su sitio web con contenido dinámico, como registros de clientes o las últimas ofertas de productos. La fuente de este contenido dinámico suele ser otros servicios dentro de la organización, que usted mismo puede haber creado.

La tentación, y a menudo el punto de venta del CMS, es que puede personalizar el CMS para extraer todo este contenido especial y mostrarlo al mundo exterior. Sin embargo, el entorno de desarrollo para el CMS promedio es *terrible*.

Veamos en qué se especializa el CMS promedio y para qué lo compramos probablemente: creación y administración de contenido. La mayoría de los CMS son bastante malos incluso para hacer el diseño de la página, por lo general proporcionan herramientas de arrastrar y soltar que no cortan la mostaza. E incluso entonces, terminas necesitando tener a alguien que entienda HTML y CSS para ajustar las plantillas de CMS. Tienden a ser plataformas terribles en las que construir código personalizado.

¿La respuesta? Frente al CMS con su propio servicio que proporciona el sitio web al mundo exterior, como se muestra en la **Figura 4-11**. Trate al CMS como un servicio cuya función es permitir la creación y recuperación de contenido. En su propio servicio, escribe el código e integra con los servicios como desee. Usted tiene control sobre el escalado del sitio web (muchos CMS comerciales proporcionan sus propios complementos patentados para manejar la carga) y puede elegir el sistema de plantillas que tenga sentido.

La mayoría de los CMS también proporcionan API para permitir la creación de contenido, por lo que también tiene la capacidad de enfrentarlo con su propia fachada de servicio. En algunas situaciones, incluso hemos usado esa fachada para abstraer las API para recuperar contenido.

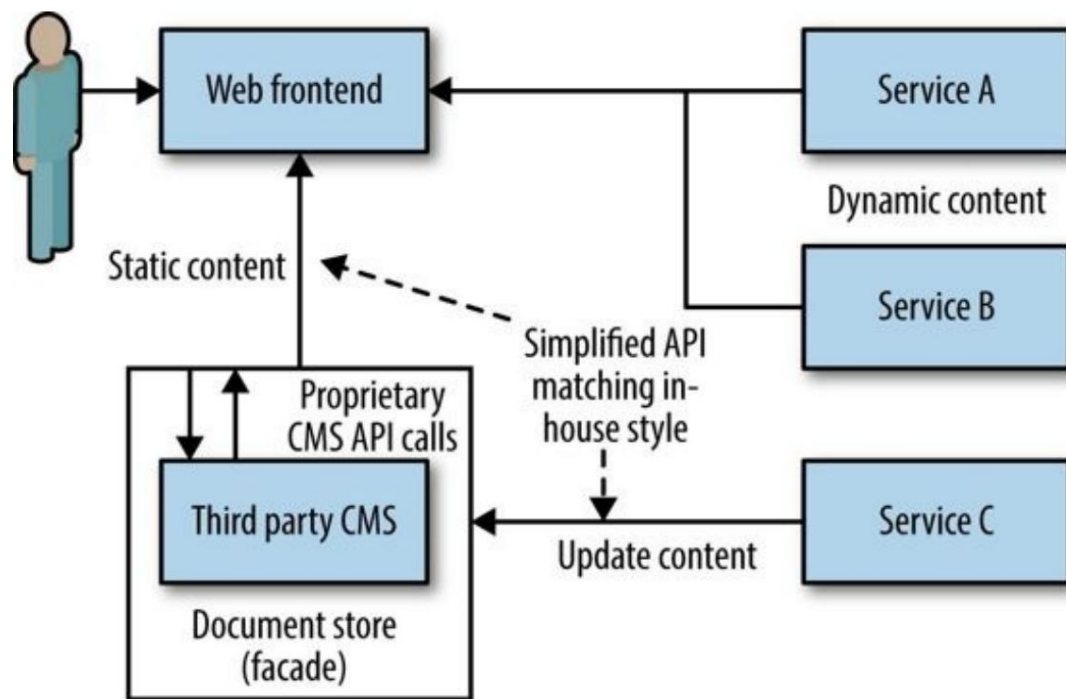


Figura 4-11. Ocultar un CMS utilizando su propio servicio

Hemos usado este patrón varias veces en ThoughtWorks en los últimos años, y yo mismo lo he hecho más de una vez. Un ejemplo notable fue un cliente que buscaba lanzar un nuevo sitio web para sus productos. Inicialmente, quería construir la solución completa en el CMS, pero aún tenía que elegir uno. En cambio, sugerimos este enfoque y comenzamos a desarrollar el sitio web de fachada. Mientras esperábamos que se seleccionara la herramienta CMS, la *falsificamos* al tener un servicio web que solo mostraba contenido estático. Terminamos lanzando el sitio mucho antes de que se seleccionara el CMS mediante el uso de nuestro servicio de contenido falso en producción para mostrar el contenido en el sitio en vivo. Más tarde, pudimos colocar la herramienta finalmente seleccionada sin ningún cambio en la aplicación frontal.

Con este enfoque, mantenemos el alcance de lo que hace el CMS al mínimo y movemos las personalizaciones a nuestra propia pila de tecnología.

Ejemplo: El sistema CRM polivalente

La herramienta CRM, o Customer Relationship Management, es una bestia que se encuentra a menudo y que puede infundir miedo en el corazón incluso del arquitecto más resistente. Este sector, como lo tipifican proveedores como Salesforce o SAP, está plagado de ejemplos de herramientas que intentan hacer todo por usted. Esto puede llevar a que la herramienta misma se convierta en un punto único de falla y en un nudo enredado de dependencias. Muchas implementaciones de herramientas de CRM que he visto se encuentran entre los mejores ejemplos de servicios *adhesivos* (en lugar de cohesivos).

El alcance de una herramienta de este tipo generalmente comienza pequeño, pero con el tiempo se convierte en una parte cada vez más importante de cómo funciona su organización. El problema es que la dirección y las elecciones que se toman en torno a este sistema ahora vital a menudo las hace el propio proveedor de la herramienta, no usted.

Hace poco participé en un ejercicio para tratar de recuperar algo de control. La organización con la que trabajaba se dio cuenta de que, aunque estaba usando la herramienta de CRM para muchas cosas,

no estaba obteniendo el valor de los crecientes costos asociados con la plataforma. Al mismo tiempo, varios sistemas internos usaban las API de CRM menos que ideales para la integración. Queríamos mover la arquitectura del sistema hacia un lugar donde tuviéramos servicios que modelaran el dominio de nuestro negocio y también sentar las bases para una posible migración.

Lo primero que hicimos fue identificar los conceptos centrales de nuestro dominio que actualmente posee el sistema CRM. Uno de ellos era el concepto de *proyecto*, es decir, algo a lo que se podía asignar a un miembro del personal. Muchos otros sistemas necesitaban información del proyecto. En su lugar, lo que hicimos fue crear un servicio de proyecto. Este servicio expuso proyectos como recursos RESTful, y los sistemas externos podrían trasladar sus puntos de integración al servicio nuevo, más fácil de usar. Internamente, el servicio del proyecto era solo una fachada que ocultaba los detalles de la integración subyacente. Puede ver esto en la [Figura 4-12](#).

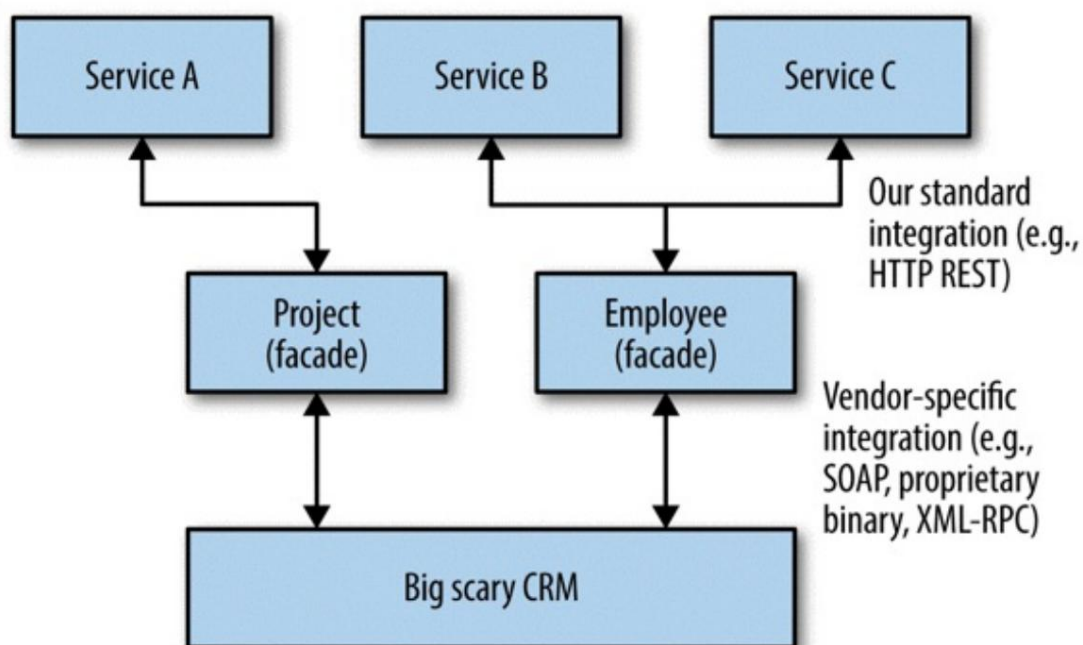


Figura 4-12. Uso de servicios de fachada para enmascarar el CRM subyacente

El trabajo, que en el momento de escribir este artículo aún estaba en marcha, consistía en identificar otros conceptos de dominio que manejaba el CRM y crear más fachadas para ellos. Cuando llegue el momento de migrar fuera del CRM subyacente, podríamos mirar cada fachada a su vez para decidir si una solución de software interna o algo listo para usar podría cumplir con los requisitos.

El patrón del estrangulador

Cuando se trata de plataformas heredadas o incluso COTS que no están totalmente bajo nuestro control, también tenemos que lidiar con lo que sucede cuando queremos eliminarlas o al menos alejarnos de ellas. Un patrón útil aquí es el **patrón de aplicación Strangler**. Al igual que con nuestro ejemplo de enfrentar el sistema CMS con nuestro propio código, con un estrangulador captura e intercepta llamadas al sistema anterior. Esto le permite decidir si enruta estas llamadas a un código heredado existente o si las dirige a un nuevo código que haya escrito. Esto le permite reemplazar la funcionalidad con el tiempo sin necesidad de una reescritura a lo grande.

Cuando se trata de microservicios, en lugar de tener una sola aplicación monolítica que intercepte todas las llamadas al sistema heredado existente, puede usar una serie de microservicios para realizar esta interceptación. Capturar y redirigir las llamadas originales puede volverse más complejo en esta situación y es posible que necesite el uso de un proxy para que lo haga por usted.

Resumen

Hemos analizado varias opciones diferentes en torno a la integración y he compartido mis pensamientos sobre qué opciones tienen más probabilidades de garantizar que nuestros microservicios permanezcan lo más desvinculados posible de sus otros colaboradores:

- Evite la integración de bases de datos a toda costa.
- Comprenda las compensaciones entre REST y RPC, pero considere REST como un buen punto de partida para la integración de solicitud/respuesta.
- Prefiere la coreografía a la orquestación.
- Evite los cambios bruscos y la necesidad de versiones mediante la comprensión de la Ley de Postel y el uso de lectores tolerantes.
- Piense en las interfaces de usuario como capas de composición.

Cubrimos mucho aquí y no pudimos profundizar en todos estos temas.

No obstante, esto debería ser una buena base para ponerlo en marcha y orientarlo en la dirección correcta si desea obtener más información.

También dedicamos un tiempo a ver cómo trabajar con sistemas que no están completamente bajo nuestro control en forma de productos COTS. ¡Resulta que esta descripción puede aplicarse fácilmente al software que escribimos!

Algunos de los enfoques descritos aquí se aplican igualmente bien al software *heredado*, pero ¿qué sucede si queremos abordar la tarea, a menudo monumental, de controlar estos sistemas más antiguos y descomponerlos en partes más utilizables? Hablaremos de eso en detalle en el próximo capítulo.

Capítulo 5. Dividir el monolito

Hemos discutido cómo es un buen servicio y por qué los servidores más pequeños pueden ser mejores para nosotros. También discutimos anteriormente la importancia de poder evolucionar el diseño de nuestros sistemas. Pero, ¿cómo manejamos el hecho de que es posible que ya tengamos una gran cantidad de bases de código que no siguen estos patrones? ¿Cómo hacemos para descomponer estas aplicaciones monolíticas sin tener que embarcarnos en una reescritura a lo grande?

El monolito crece con el tiempo. Adquiere nuevas funcionalidades y líneas de código a un ritmo alarmante. En poco tiempo se convierte en una presencia gigante grande y aterradora en nuestra organización que la gente tiene miedo de tocar o cambiar. ¡Pero no todo está perdido! Con las herramientas adecuadas a nuestra disposición, podemos matar a esta bestia.

Se trata de costuras

Discutimos en el [Capítulo 3](#) que queremos que nuestros servicios sean altamente cohesivos y poco acoplados. El problema con el monolito es que con demasiada frecuencia es lo opuesto a ambos.

En lugar de tender hacia la cohesión y mantener juntas las cosas que tienden a cambiar juntas, adquirimos y unimos todo tipo de código no relacionado. Del mismo modo, el acoplamiento flexible no existe realmente: si quiero hacer un cambio en una línea de código, puedo hacerlo con bastante facilidad, pero no puedo implementar ese cambio sin afectar potencialmente gran parte del resto del monolito, y ciertamente tendré que volver a implementar todo el sistema.

En su libro *Trabajar de forma eficaz con el código heredado* (Prentice-Hall), Michael Feathers define el concepto de *costura*, es decir, una parte del código que se puede tratar de forma aislada y trabajar en ella sin que afecte al resto de la base de código. También queremos identificar costuras. Pero en lugar de encontrarlos con el propósito de limpiar nuestra base de código, queremos identificar costuras que puedan convertirse en límites de servicio.

Entonces, ¿qué hace una buena costura? Bueno, como discutimos anteriormente, los contextos acotados son excelentes costuras porque, por definición, representan límites cohesivos y, sin embargo, poco acoplados en una organización. Entonces, el primer paso es comenzar a identificar estos límites en nuestro código.

La mayoría de los lenguajes de programación proporcionan conceptos de espacio de nombres que nos permiten agrupar código similar. El concepto de paquete de Java es un ejemplo bastante débil, pero nos brinda mucho de lo que necesitamos. Todos los demás lenguajes de programación principales tienen conceptos similares incorporados, y JavaScript es muy posiblemente una excepción.

Breaking Apart MusicCorp

Imagine que tenemos un gran servicio monolítico de back-end que representa una parte sustancial del comportamiento de los sistemas en línea de MusicCorp. Para comenzar, debemos identificar los contextos limitados de alto nivel que creemos que existen en nuestra organización, como discutimos en el [Capítulo 3](#). Luego queremos tratar de entender a qué contextos limitados se asigna el monolito. Imaginemos que inicialmente identificamos cuatro contextos que creemos que cubre nuestro backend monolítico:

Catalogar

Todo lo relacionado con metadatos sobre los artículos que ofrecemos a la venta

Finanzas

Informes de cuentas, pagos, devoluciones, etc.

Depósito

Envío y devolución de pedidos de clientes, gestión de niveles de inventario, etc.

Recomendación

Nuestro revolucionario sistema de recomendación pendiente de patente, que es un código muy complejo escrito por un equipo con más doctorados que el laboratorio de ciencias promedio

Lo primero que debe hacer es crear paquetes que representen estos contextos y luego mover el código existente a ellos. Con los IDE modernos, el movimiento de código se puede realizar automáticamente a través de refactorizaciones y se puede realizar de forma incremental mientras hacemos otras cosas. Sin embargo, aún necesitará pruebas para detectar las roturas realizadas al mover el código, especialmente si está utilizando un lenguaje de escritura dinámica en el que los IDE tienen más dificultades para realizar la refactorización. Con el tiempo, empezamos a ver qué código encaja bien y qué código *sobra* y realmente no encaja en ninguna parte. ¡Este código restante a menudo identificará contextos limitados que podríamos haber pasado por alto!

Durante este proceso, también podemos usar código para analizar las dependencias entre estos paquetes. Nuestro código debe representar a nuestra organización, por lo que nuestros paquetes que representan los contextos delimitados en nuestra organización deben interactuar de la misma manera que interactúan los grupos organizacionales de la vida real en nuestro dominio. Por ejemplo, herramientas como Structure 101 nos permiten ver gráficamente las dependencias entre paquetes. Si detectamos cosas que parecen incorrectas, por ejemplo, el paquete de almacén depende del código del paquete de finanzas cuando no existe tal dependencia en la organización real, entonces podemos investigar este problema e intentar resolverlo.

Este proceso puede tomar una tarde en un código base pequeño, o varias semanas o meses cuando se trata de millones de líneas de código. Es posible que no necesite clasificar todo el código en paquetes orientados al dominio antes de dividir su primer servicio y, de hecho, puede ser más valioso concentrar su esfuerzo en un solo lugar. No hay necesidad de que esto sea un enfoque de big-bang. Es algo que se puede hacer poco a poco, día a día, y tenemos muchas herramientas

a nuestra disposición para seguir nuestro progreso.

Entonces, ahora que tenemos nuestro código base organizado alrededor de estas costuras, ¿qué sigue?

Las razones para dividir el monolito

Decidir que desea que un servicio o una aplicación monolíticos sea más pequeño es un buen comienzo. Pero le recomendaría enfáticamente que elimine estos sistemas. Un enfoque incremental lo ayudará a aprender sobre los microservicios a medida que avanza, y también limitará el impacto de hacer algo mal (¡y se equivocará!). Piense en nuestro monolito como un bloque de mármol. Podríamos volarlo todo, pero eso rara vez termina bien. Tiene mucho más sentido simplemente eliminarlo de forma incremental.

Entonces, si vamos a romper el monolito pieza por pieza, ¿por dónde deberíamos empezar? Ya tenemos nuestras costuras, pero ¿cuál deberíamos sacar primero? Es mejor pensar en dónde obtendrá el mayor beneficio de la separación de una parte de su base de código, en lugar de simplemente dividir las cosas por el simple hecho de hacerlo. Consideremos algunos impulsores que podrían ayudar a guiar nuestro cincel.

Ritmo de cambio

Tal vez sabemos que pronto tendremos muchos cambios en la forma en que administramos el inventario. Si dividimos la costura del almacén como un servicio ahora, podríamos cambiar ese servicio más rápido, ya que es una unidad autónoma separada.

Estructura de equipo

El equipo de entrega de MusicCorp en realidad está dividido en dos regiones geográficas. Un equipo está en Londres, el otro en Hawái (¡algunas personas lo tienen fácil!). Sería genial si pudiéramos dividir el código en el que más trabaja el equipo de Hawái, para que pueda asumir la propiedad total. Exploraremos más esta idea en el [Capítulo 10](#).

Seguridad

MusicCorp ha tenido una auditoría de seguridad y ha decidido reforzar su protección de información confidencial. Actualmente, todo esto lo maneja el código relacionado con las finanzas. Si dividimos este servicio, podemos brindar protecciones adicionales a este servicio individual en términos de monitoreo, protección de datos en tránsito y protección de datos en reposo: ideas que veremos con más detalle en el **Capítulo 9**.

Tecnología

El equipo que se ocupa de nuestro sistema de recomendación ha estado desarrollando algunos algoritmos nuevos utilizando una biblioteca de programación lógica en el lenguaje Clojure. El equipo cree que esto podría beneficiar a nuestros clientes al mejorar lo que les ofrecemos. Si pudiéramos dividir el código de recomendación en un servicio separado, sería fácil considerar la creación de una implementación alternativa con la que podamos probar.

Dependencias enredadas

El otro punto a considerar cuando haya identificado un par de costuras para separar es qué tan enredado está ese código con el resto del sistema. Queremos sacar la costura de la que menos se dependa, si podemos. Si puede ver las diversas costuras que ha encontrado como un gráfico acíclico dirigido de dependencias (algo en lo que las herramientas de modelado de paquetes que mencioné anteriormente son muy buenas), esto puede ayudarlo a detectar las costuras que probablemente serán más difíciles de desenredar.

Esto nos lleva a lo que suele ser la madre de todas las dependencias enredadas: la base de datos.

La base de datos

Ya hemos discutido extensamente los desafíos de usar bases de datos como un método para integrar múltiples servicios. Como lo dejé bastante claro antes, ¡no soy un fanático! Esto significa que también debemos encontrar costuras en nuestras bases de datos para poder dividirlos limpiamente. Las bases de datos, sin embargo, son bestias engañosas.

Abordar el problema

El primer paso es echar un vistazo al código en sí y ver qué partes leen y escriben desde la base de datos. Una práctica común es tener una capa de repositorio, respaldada por algún tipo de marco como Hibernate, para vincular su código a la base de datos, lo que facilita el mapeo de objetos o estructuras de datos hacia y desde la base de datos. Si ha estado siguiendo hasta ahora, habrá agrupado nuestro código en paquetes que representan nuestros contextos acotados; queremos hacer lo mismo con nuestro código de acceso a la base de datos. Esto puede requerir dividir la capa del repositorio en varias partes, como se muestra en **la Figura 5-1**.

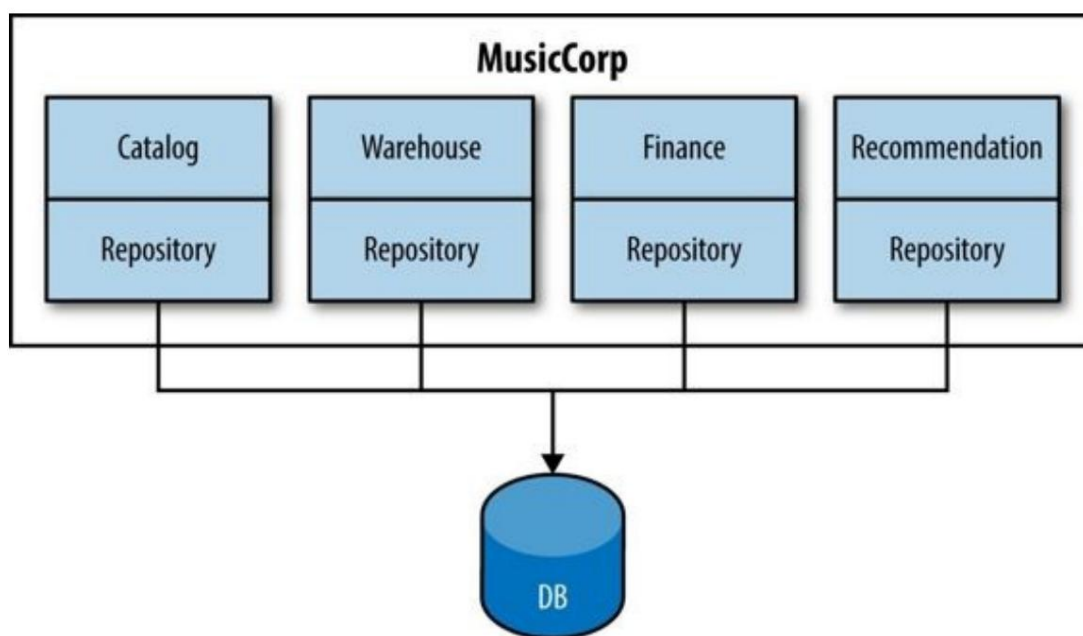


Figura 5-1. Dividiendo nuestras capas de repositorio

Tener el código de mapeo de la base de datos colocado dentro del código para un contexto dado puede ayudarnos a comprender qué partes de la base de datos son utilizadas por qué bits de código. Hibernate, por ejemplo, puede dejar esto muy claro si está utilizando algo como un archivo de mapeo por contexto acotado.

Esto no nos da toda la historia, sin embargo. Por ejemplo, es posible que podamos decir que el código de finanzas usa la tabla de contabilidad y que el código de catálogo usa la tabla de elementos de línea, pero es posible que no quede claro que la base de datos impone una relación de clave externa desde la tabla de contabilidad hasta la tabla de línea. Para ver estas restricciones a nivel de base de datos, que pueden ser un obstáculo, necesitamos usar otra herramienta para visualizar los datos. Un excelente lugar para comenzar es usar una herramienta como **SchemaSpy disponible gratuitamente**, que puede generar representaciones gráficas de las relaciones entre tablas.

Todo esto lo ayuda a comprender el acoplamiento entre las tablas que pueden abarcar lo que eventualmente se convertirá en los límites del servicio. Pero, ¿cómo se cortan esos lazos? ¿Y qué pasa con los casos en los que se usan las mismas tablas desde múltiples contextos acotados diferentes? Manejar problemas como estos no es fácil y hay muchas respuestas, pero es factible.

Volviendo a algunos ejemplos concretos, volvamos a considerar nuestra tienda de música. Hemos identificado cuatro contextos acotados y queremos avanzar para convertirlos en cuatro servicios de colaboración distintos. Vamos a ver algunos ejemplos concretos de problemas que podríamos enfrentar y sus posibles soluciones. Y aunque algunos de estos ejemplos hablan específicamente sobre los desafíos que se encuentran en las bases de datos relacionales estándar, encontrará problemas similares en otras tiendas NOSQL alternativas.

Ejemplo: ruptura de relaciones de clave externa

En este ejemplo, nuestro código de catálogo utiliza una tabla de elementos de línea genérica para almacenar información sobre un álbum. Nuestro código de finanzas utiliza una tabla de contabilidad para realizar un seguimiento de las transacciones financieras. Al final de cada mes, necesitamos generar informes para varias personas de la organización para que puedan ver cómo lo estamos haciendo. Queremos que los informes sean agradables y fáciles de leer, así que en lugar de decir: "Vendimos 400 copias del SKU 12345 y ganamos \$1300", nos gustaría agregar más información sobre lo que se vendió (es decir, "Vendimos 400 copias de los *grandes éxitos* de Bruce Springsteen y recaudó 1.300 dólares"). Para hacer esto, nuestro código de informe en el paquete financiero llegará a la tabla de elementos de línea para extraer el título del SKU. También puede tener una restricción de clave externa del libro mayor a la tabla de elementos de línea, como vemos en [la figura 5-2](#).

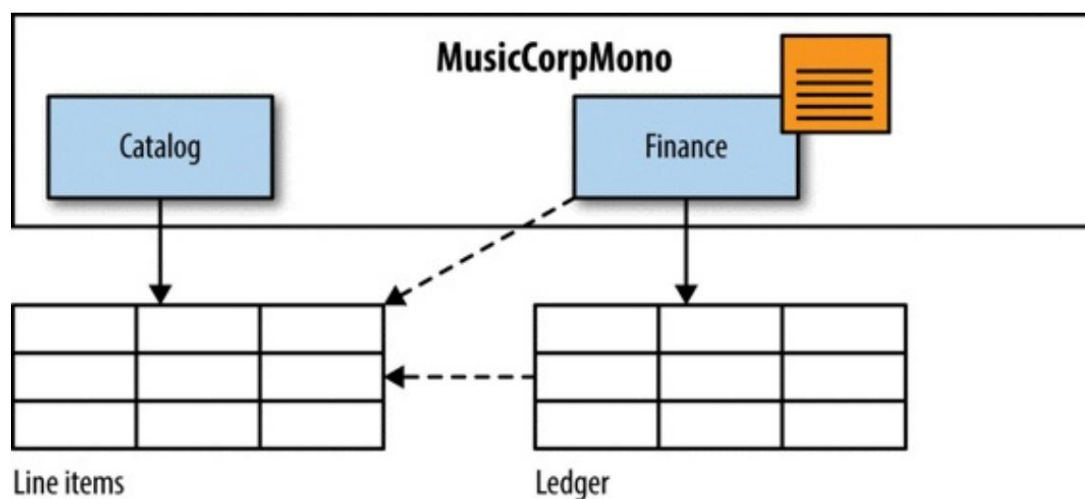


Figura 5-2. Relación de clave externa

Entonces, ¿cómo arreglamos las cosas aquí? Bueno, tenemos que hacer un cambio en dos lugares. Primero, debemos evitar que el código de finanzas llegue a la tabla de elementos de línea, ya que esta tabla realmente pertenece al código del catálogo, y no queremos que la integración de la base de datos ocurra una vez que el catálogo y las finanzas sean servicios por derecho propio. La forma más rápida de abordar esto es que, en lugar de que el código de finanzas llegue a la tabla de elementos de línea, exponremos los datos a través de una llamada a la API en el paquete de catálogo al que puede llamar el código de finanzas. Esta llamada a la API será la precursora de una llamada que haremos por cable, como vemos en [la Figura 5-3](#).

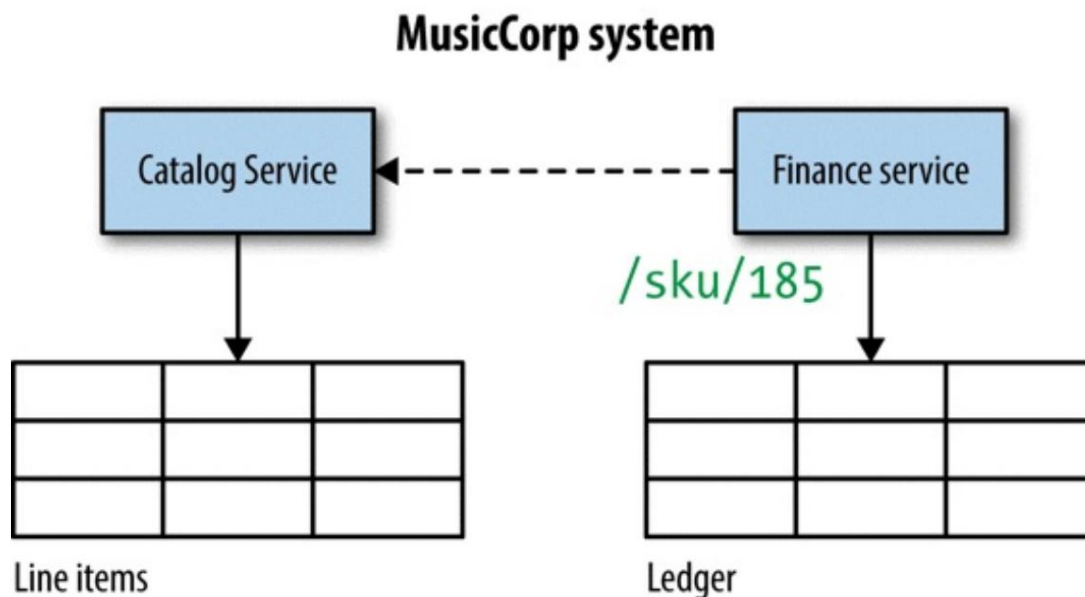


Figura 5-3. Después de la eliminación de la relación de clave externa

En este punto, queda claro que es posible que terminemos teniendo que hacer dos llamadas a la base de datos para generar el informe. Esto es correcto. Y lo mismo ocurrirá si se trata de dos servicios separados. Por lo general, ahora se plantean preocupaciones sobre el rendimiento. Tengo una respuesta bastante fácil para eso: ¿qué tan rápido debe ser su sistema? ¿Y qué tan rápido es ahora? Si puede probar su rendimiento actual y saber cómo se ve un buen rendimiento, entonces debe sentirse seguro para hacer un cambio. A veces, hacer una cosa más lenta a cambio de otras cosas es lo correcto, especialmente si la *lentitud* sigue siendo perfectamente aceptable.

Pero, ¿qué pasa con la relación de clave externa? Bueno, perdemos esto por completo. Esto se convierte en una restricción que debemos administrar ahora en nuestros servicios resultantes en lugar de en el nivel de la base de datos. Esto puede significar que necesitamos implementar nuestra propia verificación de consistencia entre los servicios o desencadenar acciones para limpiar los datos relacionados. Si esto es necesario o no, a menudo no es una elección del tecnólogo. Por ejemplo, si nuestro servicio de pedidos contiene una lista de ID para artículos del catálogo, ¿qué sucede si se elimina un artículo del catálogo y ahora un pedido hace referencia a un ID de catálogo no válido? ¿Deberíamos permitirlo? Si lo hacemos, ¿cómo se representa esto en el orden cuando se muestra? Si no lo hacemos, ¿cómo podemos verificar que esto no se viole? Estas son preguntas que necesitará que le respondan las personas que definen cómo debe comportarse su sistema para sus usuarios.

Ejemplo: datos estáticos compartidos

He visto tal vez tantos códigos de países almacenados en bases de datos (que se muestran en la [Figura 5-4](#)) como clases de `StringUtils` escritas para proyectos internos de Java. Esto parece implicar que planeamos cambiar los países que admite nuestro sistema con mucha más frecuencia de lo que implementaremos un código nuevo, pero cualquiera que sea la razón real, estos ejemplos de datos estáticos compartidos que se almacenan en bases de datos surgen mucho. Entonces, ¿qué hacemos en nuestra tienda de música si todos nuestros servicios potenciales se leen desde la misma tabla como esta?

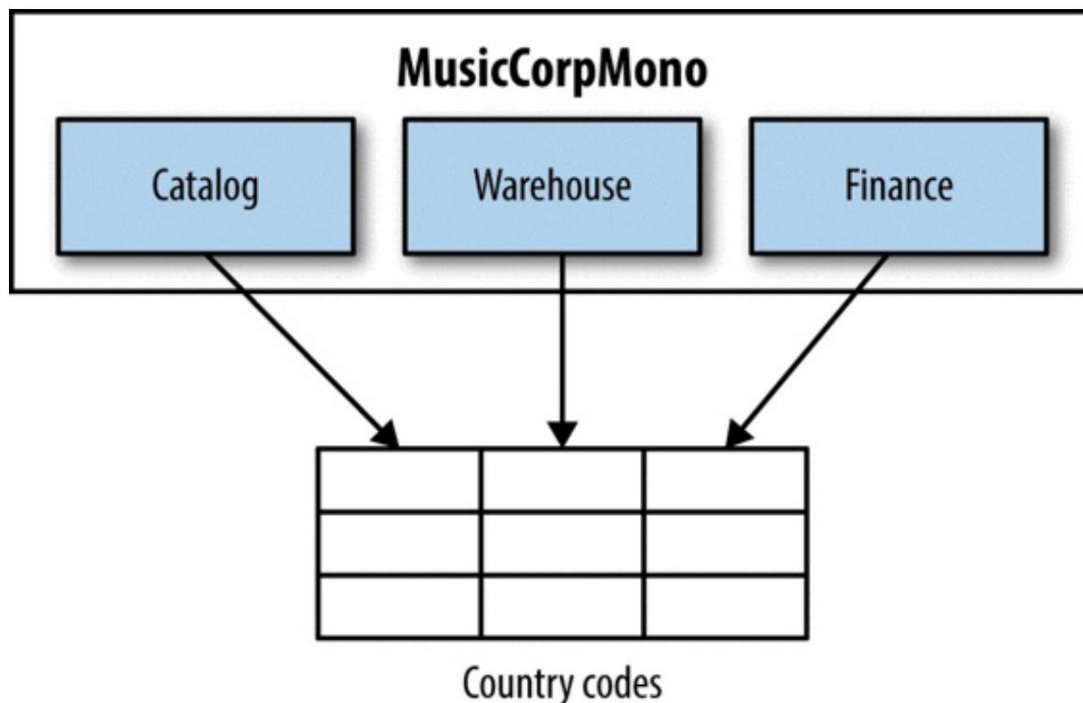


Figura 5-4. Códigos de países en la base de datos

Bueno, tenemos algunas opciones. Una es duplicar esta tabla para cada uno de nuestros paquetes, con la perspectiva a largo plazo de que también se duplicará dentro de cada servicio. Por supuesto, esto conduce a un posible desafío de coherencia: ¿qué sucede si actualizo una tabla para reflejar la creación de Newmantopia frente a la costa este de Australia, pero no otra?

Una segunda opción es tratar estos datos estáticos compartidos como código. Tal vez podría estar en un archivo de propiedades implementado como parte del servicio, o tal vez solo como una enumeración. Los problemas relacionados con la consistencia de los datos persisten, aunque la experiencia ha demostrado que es mucho más fácil enviar cambios a los archivos de configuración que alterar las tablas de la base de datos en vivo. Este es a menudo un enfoque muy sensato.

Una tercera opción, que bien puede ser extrema, es enviar estos datos estáticos a un servicio por derecho propio. En un par de situaciones que he encontrado, el volumen, la complejidad y las reglas asociadas con los datos de referencia estáticos fueron suficientes para justificar este enfoque, ¡pero probablemente sea excesivo si solo estamos hablando de códigos de países!

Personalmente, en la mayoría de las situaciones intentaría mantener estos datos en los archivos de configuración o directamente en el código, ya que es la opción más sencilla en la mayoría de los casos.

Ejemplo: datos compartidos

Ahora profundicemos en un ejemplo más complejo, pero que puede ser un problema común cuando intenta separar sistemas: datos mutables compartidos. Nuestro código de finanzas rastrea los pagos realizados por los clientes por sus pedidos, y también rastrea los reembolsos que se les otorgan cuando devuelven artículos. Mientras tanto, el código de almacén actualiza los registros para mostrar que los pedidos de los clientes se han enviado o recibido. Todos estos datos se muestran en un lugar conveniente en el sitio web para que los clientes puedan ver qué está pasando con su cuenta. Para simplificar las cosas, hemos almacenado toda esta información en una tabla de registro de clientes bastante genérica, como se muestra en la **Figura 5-5**.

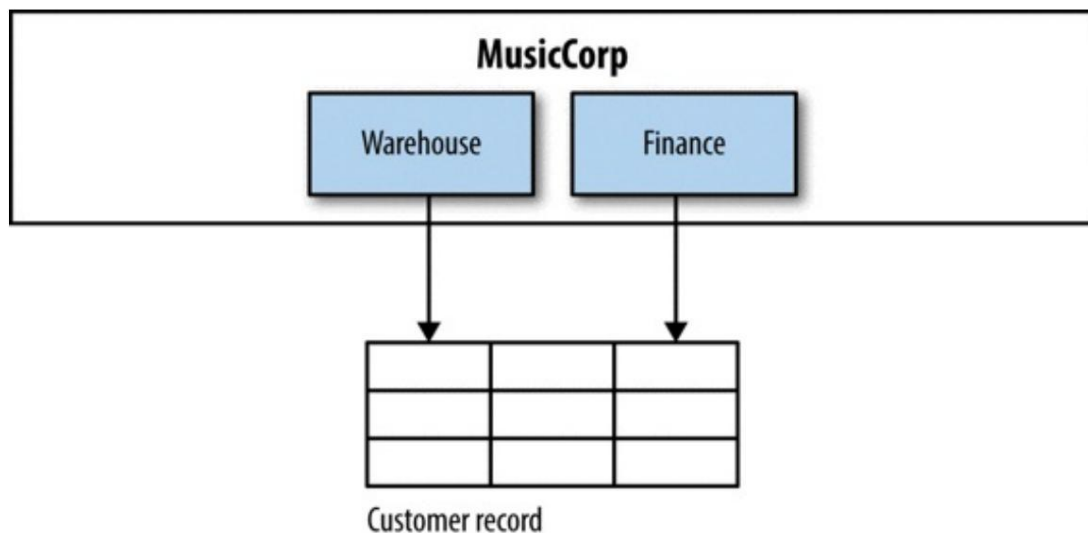


Figura 5-5. Acceder a los datos de los clientes: ¿nos estamos perdiendo algo?

Por lo tanto, tanto el código de finanzas como el de almacén se escriben en la misma tabla y, en ocasiones, se leen. ¿Cómo podemos desmenuzar esto? Lo que realmente tenemos aquí es algo que verá a menudo: un concepto de dominio que no está modelado en el código y, de hecho, está modelado implícitamente en la base de datos. Aquí, el concepto de dominio que falta es el de Cliente.

Necesitamos concretar el actual concepto abstracto del cliente. Como paso transitorio, creamos un nuevo paquete llamado Cliente. Luego podemos usar una API para exponer el código del Cliente a otros paquetes, como finanzas o almacén. Prolongando todo esto, ahora podemos terminar con un servicio de atención al cliente distinto (**Figura 5-6**).

MusicCorp system

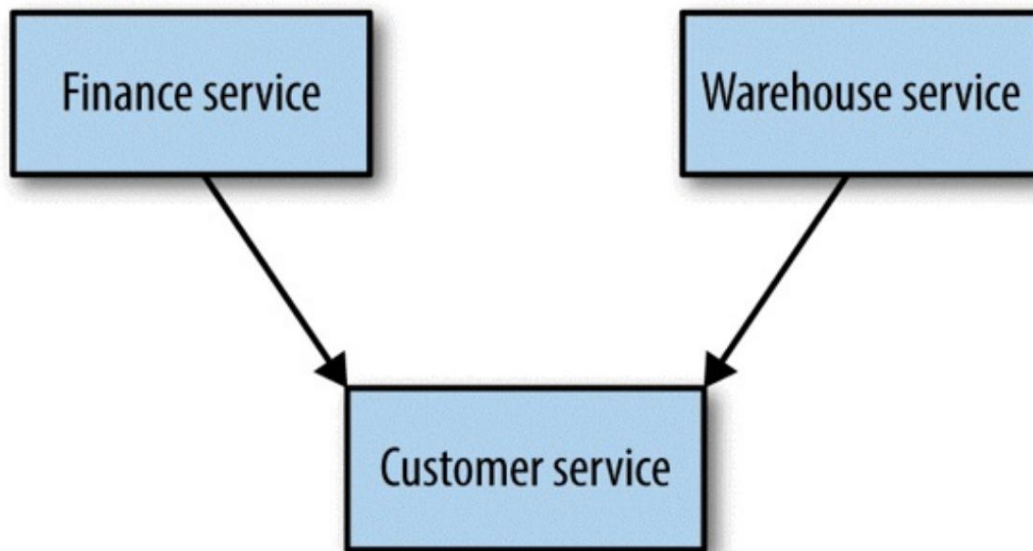


Figura 5-6. Reconocimiento del contexto acotado del cliente

Ejemplo: Mesas Compartidas

La figura 5-7 muestra nuestro último ejemplo. Nuestro catálogo necesita almacenar el nombre y el precio de los discos que vendemos, y el almacén necesita mantener un registro electrónico de inventario.

Decidimos mantener estas dos cosas en el mismo lugar en una tabla de partidas genéricas. Antes, con todo el código fusionado, no estaba claro que en realidad estábamos fusionando preocupaciones, pero ahora podemos ver que, de hecho, tenemos dos conceptos separados que podrían almacenarse de manera diferente.

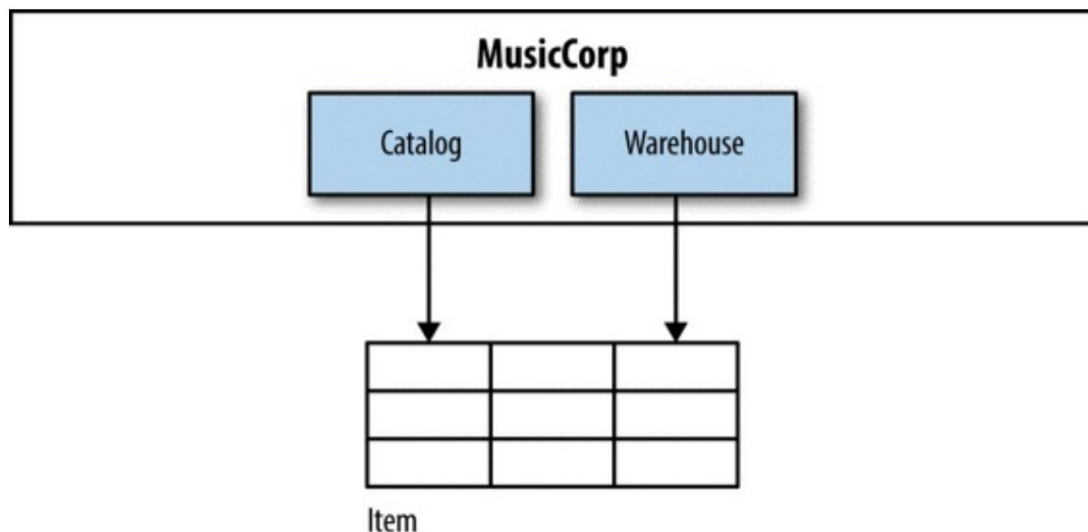


Figura 5-7. Mesas compartidas entre diferentes contextos

La respuesta aquí es dividir la tabla en dos como tenemos en la Figura 5-8, tal vez creando una tabla de lista de existencias para el almacén y una tabla de entrada de catálogo para los detalles del catálogo.

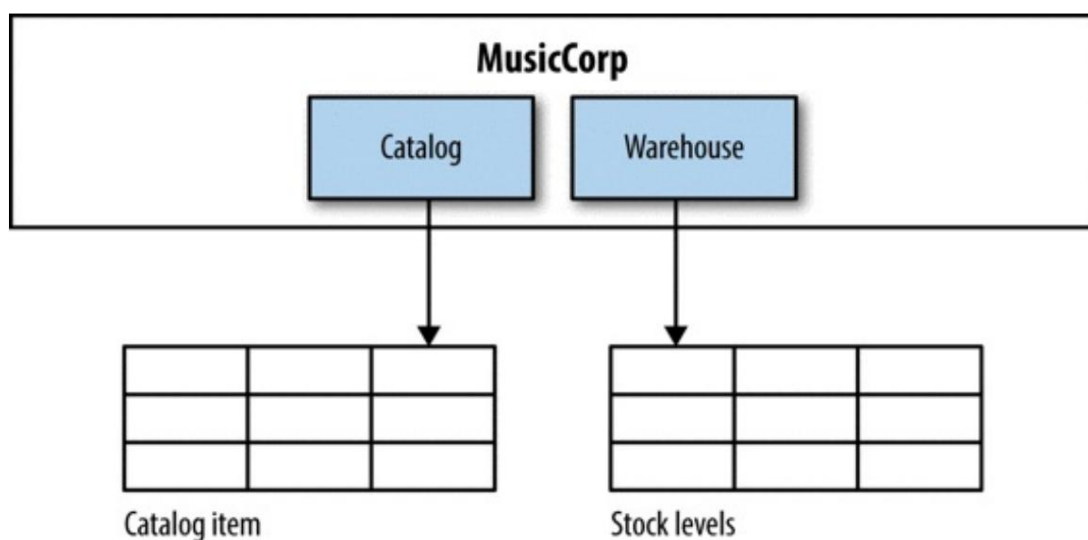


Figura 5-8. Separando la mesa compartida

Bases de datos de refactorización

Lo que hemos cubierto en los ejemplos anteriores son algunas refactorizaciones de bases de datos que pueden ayudarlo a separar sus esquemas. Para una discusión más detallada del tema, puede consultar *Refactoring Databases* de Scott J. Ambler y Pramod J. Sadalage (Addison-Wesley).

Escenificando el descanso

Así que hemos encontrado costuras en nuestro código de aplicación, agrupándolo en contextos acotados. Hemos usado esto para identificar costuras en la base de datos, y hemos hecho todo lo posible para dividirlos. ¿Qué sigue? ¿Realiza un lanzamiento a lo grande, pasando de un servicio monolítico con un solo esquema a dos servicios, cada uno con su propio esquema? De hecho, recomendaría dividir el esquema pero mantener el servicio junto antes de dividir el código de la aplicación en microservicios separados, como se muestra en la **Figura 5-9**.

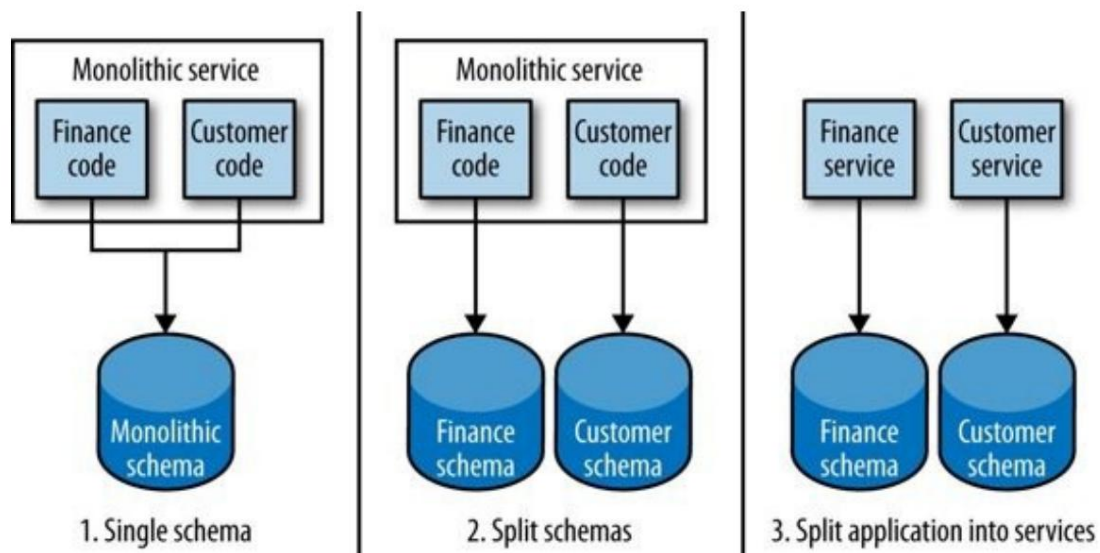


Figura 5-9. Organizar una separación de servicios

Con un esquema separado, aumentaremos potencialmente la cantidad de llamadas a la base de datos para realizar una sola acción. Donde antes podríamos haber podido tener todos los datos que queríamos en una sola instrucción `SELECT`, ahora es posible que necesitemos recuperar los datos de dos ubicaciones y unirlos en la memoria. Además, terminamos rompiendo la integridad transaccional cuando pasamos a dos esquemas, lo que podría tener un impacto significativo en nuestras aplicaciones; discutiremos esto a continuación. Al dividir los esquemas pero manteniendo el código de la aplicación unido, nos damos la capacidad de revertir nuestros cambios o continuar modificando las cosas sin afectar a los consumidores de nuestro servicio. Una vez que estemos satisfechos de que la separación de la base de datos tiene sentido, podemos pensar en dividir el código de la aplicación en dos servicios.

Límites transaccionales

Las transacciones son cosas útiles. Nos permiten decir que *estos eventos suceden todos juntos o ninguno de ellos sucede*. Son muy útiles cuando estamos insertando datos en una base de datos; nos permiten actualizar varias tablas a la vez, sabiendo que si algo falla, todo se revierte, lo que garantiza que nuestros datos no entren en un estado inconsistente.

En pocas palabras, una transacción nos permite agrupar varias actividades diferentes que llevan nuestro sistema de un estado constante a otro: todo funciona o nada cambia.

Las transacciones no solo se aplican a las bases de datos, aunque las usamos con mayor frecuencia en ese contexto. Los intermediarios de mensajes, por ejemplo, le han permitido durante mucho tiempo publicar y recibir mensajes dentro de las transacciones también.

Con un esquema monolítico, todas nuestras creaciones o actualizaciones probablemente se realizarán dentro de un único límite transaccional. Cuando separamos nuestras bases de datos, perdemos la seguridad que nos brinda tener una sola transacción. Considere un ejemplo simple en el contexto de MusicCorp.

Al crear un pedido, quiero actualizar la tabla de pedidos indicando que se ha creado un pedido de cliente, y también poner una entrada en una tabla para el equipo de almacén para que sepa que hay un pedido que debe recogerse para su envío. Hemos llegado tan lejos como para agrupar el código de nuestra aplicación en paquetes separados, y también hemos separado las partes del esquema del cliente y del almacén lo suficientemente bien como para estar listos para colocarlas en sus propios esquemas antes de separar el código de la aplicación.

Dentro de una sola transacción en nuestro esquema monolítico existente, la creación del pedido y la inserción del registro para el equipo de almacén se lleva a cabo dentro de una sola transacción, como se muestra en la **Figura 5-10**.

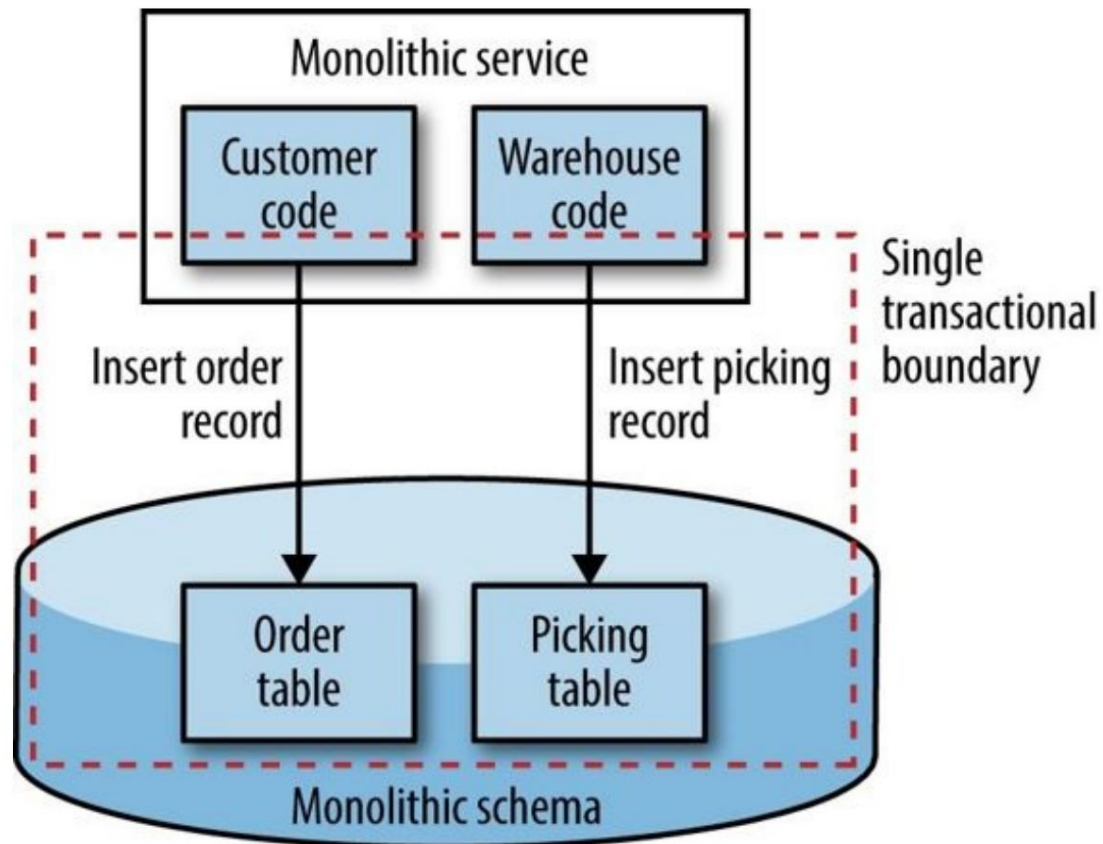


Figura 5-10. Actualización de dos tablas en una sola transacción

Pero si hemos separado el esquema en dos esquemas separados, uno para los datos relacionados con el cliente, incluida nuestra tabla de pedidos, y otro para el almacén, hemos perdido esta seguridad transaccional. El proceso de colocación de pedidos ahora abarca dos límites transaccionales separados, como vemos en la [Figura 5-11](#). Si nuestra inserción en la tabla de pedidos falla, claramente podemos detener todo, dejándonos en un estado consistente. Pero, ¿qué sucede cuando la inserción en la tabla de pedidos funciona, pero falla la inserción en la tabla de selección?

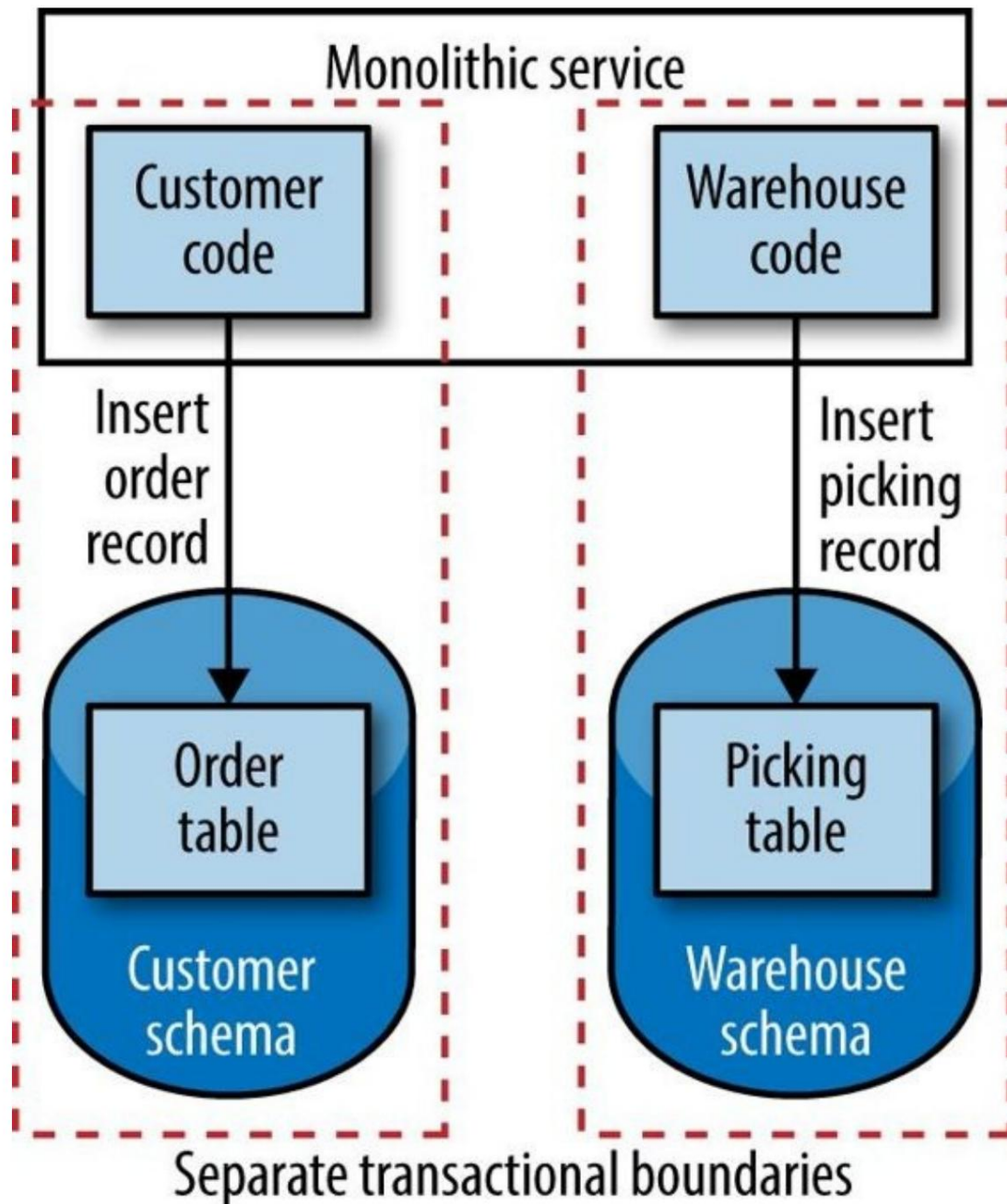


Figura 5-11. Ampliación de los límites transaccionales para una sola operación

Inténtalo de nuevo más tarde

El hecho de que el pedido haya sido capturado y realizado puede ser suficiente para nosotros, y podemos decidir volver a intentar la inserción en la mesa de selección del almacén en una fecha posterior. Podríamos poner en cola esta parte de la operación en una cola o archivo de registro y volver a intentarlo más tarde. Para algunos tipos de operaciones esto tiene sentido, pero debemos suponer que un reintento lo arreglaría.

En muchos sentidos, esta es otra forma de lo que se denomina *consistencia eventual*. En lugar de utilizar un límite transaccional para garantizar que el sistema esté en un estado coherente cuando se complete la transacción, aceptamos que el sistema entrará en un estado coherente en algún momento en el futuro. Este enfoque es especialmente útil con operaciones comerciales que pueden ser de larga duración. Discutiremos esta idea con más profundidad en el **Capítulo 11** cuando cubramos los patrones de escala.

Abortar toda la operación

Otra opción es rechazar toda la operación. En este caso, tenemos que volver a poner el sistema en un estado consistente. La tabla de selección es fácil, ya que falló esa inserción, pero tenemos una transacción confirmada en la tabla de pedidos. Tenemos que deshacer esto. Lo que tenemos que hacer es emitir una *transacción de compensación*, iniciando una nueva transacción para retroceder lo que acaba de suceder. Para nosotros, eso podría ser algo tan simple como emitir una instrucción DELETE para eliminar el pedido de la base de datos. Luego, también tendríamos que informar a través de la interfaz de usuario que la operación falló. Nuestra aplicación podría manejar ambos aspectos dentro de un sistema monolítico, pero tendríamos que considerar qué podríamos hacer cuando dividimos el código de la aplicación. ¿La lógica para manejar la transacción de compensación reside en el servicio al cliente, el servicio de pedidos o en algún otro lugar?

Pero, ¿qué sucede si nuestra transacción de compensación falla? Ciertamente es posible. Entonces tendríamos un pedido en la tabla de pedidos sin instrucciones de selección coincidentes. En esta situación, deberá volver a intentar la transacción de compensación o permitir que algún proceso de back-end limpie la incoherencia más adelante. Esto podría ser algo tan simple como una pantalla de mantenimiento a la que el personal administrativo tuviera acceso o un proceso automatizado.

Ahora piensa en lo que sucede si no tenemos una o dos operaciones que queremos que sean consistentes, sino tres, cuatro o cinco. El manejo de transacciones de compensación para cada modo de falla se vuelve bastante difícil de comprender, y mucho menos de implementar.

Transacciones distribuidas

Una alternativa a la orquestación manual de transacciones de compensación es utilizar una *transacción distribuida*. Las transacciones distribuidas intentan abarcar múltiples transacciones dentro de ellas, utilizando algún proceso de gobierno general llamado *administrador de transacciones* para orquestar las diversas transacciones que realizan los sistemas subyacentes. Al igual que con una transacción normal, una transacción distribuida intenta garantizar que todo permanezca en un estado coherente, solo que en este caso intenta hacerlo en múltiples sistemas diferentes que se ejecutan en diferentes procesos, a menudo comunicándose a través de los límites de la red.

El algoritmo más común para manejar transacciones distribuidas, especialmente transacciones de corta duración, como en el caso de manejar el pedido de nuestro cliente, es usar una *confirmación de dos fases*. Con un compromiso de dos fases, primero viene la fase de votación. Aquí es donde cada participante (también llamado *cohorte* en este contexto) en la transacción distribuida le dice al administrador de la transacción si cree que su transacción local puede continuar. Si el administrador de transacciones obtiene un voto *afirmativo* de todos los participantes, les dice a todos que continúen y realicen sus compromisos. Un solo voto negativo es suficiente para que el administrador de transacciones envíe una reversión a todas las partes .

Este enfoque se basa en que todas las partes se detengan hasta que el proceso de coordinación central les indique que continúen. Esto significa que somos vulnerables a las interrupciones. Si el administrador de transacciones deja de funcionar, las transacciones pendientes nunca se completan. Si una cohorte no responde durante la votación, todo se bloquea. Y también está el caso de lo que pasa si falla un commit después de votar. Hay una suposición implícita en este algoritmo de que esto no puede suceder: si una cohorte dice que *sí* durante el período de votación, entonces debemos asumir que se comprometerá. Las cohortes necesitan una forma de hacer que este compromiso funcione en algún momento. Esto significa que este algoritmo no es infalible; más bien, solo intenta detectar la mayoría de los casos de falla.

Este proceso de coordinación también significa bloqueos; es decir, las transacciones pendientes pueden bloquear los recursos. Los bloqueos en los recursos pueden generar contención, lo que dificulta mucho más el escalado de los sistemas, especialmente en el contexto de los sistemas distribuidos.

Se han implementado transacciones distribuidas para pilas de tecnología específicas, como la API de transacciones de Java, lo que permite que recursos dispares, como una base de datos y una cola de mensajes, participen todos en la misma transacción global. Los diversos algoritmos son difíciles de acertar, por lo que le sugiero que evite intentar crear el suyo propio. En su lugar, investigue mucho sobre este tema si parece ser la ruta que desea tomar, y vea si puede usar una implementación existente.

¿Entonces lo que hay que hacer?

Todas estas soluciones añaden complejidad. Como puede ver, las transacciones distribuidas son difíciles de realizar correctamente y, de hecho, pueden inhibir el escalado. Los sistemas que eventualmente convergen a través de la lógica de reintento de compensación pueden ser más difíciles de razonar y pueden necesitar otro comportamiento de compensación para corregir las inconsistencias en los datos.

Cuando encuentre operaciones comerciales que actualmente ocurren dentro de una sola transacción, pregúntese si realmente es necesario. ¿Pueden ocurrir en diferentes transacciones locales y confiar en el concepto de consistencia eventual? Estos sistemas son mucho más fáciles de construir y escalar (hablaremos de esto más en el [Capítulo 11](#)).

Si encuentra un estado que realmente quiere mantenerse consistente, haga todo lo posible para evitar dividirlo en primer lugar. Esfuérzate *mucho*. Si realmente necesita continuar con la división, piense en pasar de una vista puramente técnica del proceso (por ejemplo, una transacción de base de datos) y realmente cree un concepto concreto para representar la transacción en sí. Esto le brinda un identificador, o un gancho, para ejecutar otras operaciones, como transacciones de compensación, y una forma de monitorear y administrar estos conceptos más complejos en su sistema. Por ejemplo, puede crear la idea de un "pedido en proceso" que le brinde un lugar natural para enfocar toda la lógica en torno al procesamiento del pedido de principio a fin (y el tratamiento de las excepciones).

Informes

Como ya hemos visto, al dividir un servicio en partes más pequeñas, también necesitamos dividir potencialmente cómo y dónde se almacenan los datos. Sin embargo, esto crea un problema cuando se trata de un caso de uso vital y común: la generación de informes.

Un cambio en la arquitectura tan fundamental como pasar a una arquitectura de microservicios causará muchas interrupciones, pero eso no significa que tengamos que abandonar todo lo que hacemos. La audiencia de nuestros sistemas de informes son usuarios como cualquier otro, y debemos considerar sus necesidades. Sería arrogante cambiar fundamentalmente nuestra arquitectura y simplemente pedirles que se adapten. Si bien no estoy sugiriendo que el espacio de los informes no esté maduro para la disrupción, ciertamente lo está, es valioso determinar primero cómo trabajar con los procesos existentes. A veces tenemos que escoger nuestras batallas.

La base de datos de informes

Los informes generalmente necesitan agrupar datos de varias partes de nuestra organización para generar resultados útiles. Por ejemplo, podríamos querer enriquecer los datos de nuestro libro mayor con descripciones de lo que se vendió, que obtenemos de un catálogo. O podríamos querer ver el comportamiento de compra de clientes específicos de alto valor, lo que podría requerir información de su historial de compras y su perfil de cliente.

En una arquitectura de servicio monolítica estándar, todos nuestros datos se almacenan en una gran base de datos. Esto significa que todos los datos están en un solo lugar, por lo que informar sobre toda la información es bastante fácil, ya que simplemente podemos unir los datos a través de consultas SQL o similares. Por lo general, no ejecutaremos estos informes en la base de datos principal por temor a que la carga generada por nuestras consultas afecte el rendimiento del sistema principal, por lo que a menudo estos sistemas de informes dependen de una réplica de lectura, como se muestra en la [Figura 5-12](#).

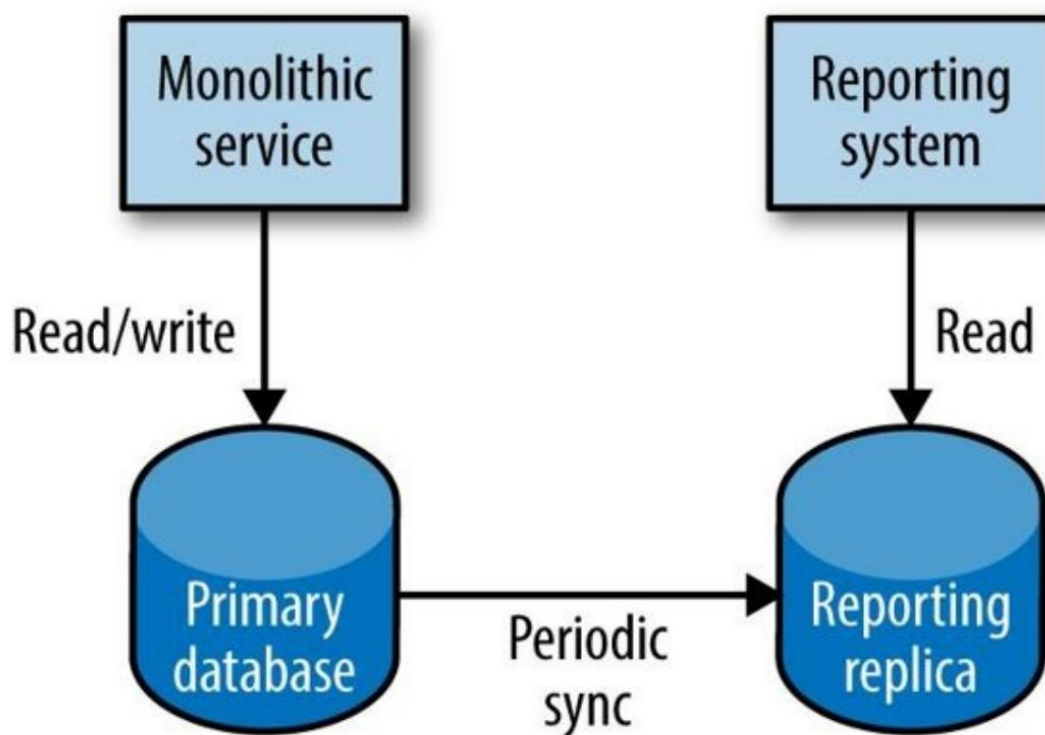


Figura 5-12. Replicación de lectura estándar

Con este enfoque, tenemos una ventaja considerable: todos los datos ya están en un solo lugar, por lo que podemos usar herramientas bastante sencillas para consultarlos. Pero también hay un par de inconvenientes con este enfoque. Primero, el esquema de la base de datos ahora es efectivamente una API compartida entre los servicios monolíticos en ejecución y cualquier sistema de informes. Por lo tanto, un cambio en el esquema debe manejarse con cuidado. En realidad, este es otro impedimento que reduce las posibilidades de que quien quiera asumir la tarea de realizar y coordinar dicho cambio.

En segundo lugar, tenemos opciones limitadas en cuanto a cómo se puede optimizar la base de datos para cualquier caso de uso: respaldar el sistema en vivo o el sistema de informes. Algunas bases de datos nos permiten hacer

optimizaciones en réplicas de lectura para permitir informes más rápidos y eficientes; por ejemplo, MySQL nos permitiría ejecutar un backend diferente que no tiene la sobrecarga de administrar transacciones. Sin embargo, no podemos estructurar los datos de manera diferente para que los informes sean más rápidos si ese cambio en la estructura de datos tiene un impacto negativo en el sistema en ejecución.

Lo que sucede a menudo es que el esquema termina siendo excelente para un caso de uso y pésimo para el otro, o bien se convierte en el mínimo común denominador, excelente para ningún propósito.

Finalmente, las opciones de bases de datos disponibles para nosotros se han disparado recientemente. Si bien las bases de datos relacionales estándar exponen interfaces de consulta SQL que funcionan con muchas herramientas de generación de informes, no siempre son la mejor opción para almacenar datos para nuestros servicios en ejecución. ¿Qué pasa si los datos de nuestra aplicación se modelan mejor como un gráfico, como en Neo4j? ¿O qué tal si preferimos usar un almacén de documentos como MongoDB? Del mismo modo, ¿qué pasaría si quisiéramos explorar el uso de una base de datos orientada a columnas como Cassandra para nuestro sistema de informes, lo que hace que sea mucho más fácil escalar para volúmenes más grandes? La limitación de tener que tener una base de datos para ambos propósitos hace que a menudo no podamos tomar estas decisiones y explorar nuevas opciones.

Así que no es perfecto, pero funciona (en su mayoría). Ahora bien, si nuestra información se almacena en múltiples sistemas diferentes, ¿qué hacemos? ¿Hay alguna manera de que podamos reunir todos los datos para ejecutar nuestros informes? ¿Y también podríamos encontrar una forma de eliminar algunas de las desventajas asociadas con el modelo de base de datos de informes estándar?

Resulta que tenemos una serie de alternativas viables a este enfoque. La solución que tenga más sentido para usted dependerá de una serie de factores, pero exploraremos algunas opciones diferentes que he visto en la práctica.

Recuperación de datos a través de llamadas de servicio

Hay muchas variantes de este modelo, pero todas se basan en extraer los datos necesarios de los sistemas de origen a través de llamadas API. Para un sistema de informes muy simple, como un panel de control que solo quiera mostrar la cantidad de pedidos realizados en los últimos 15 minutos, esto podría estar bien. Para generar informes sobre datos de dos o más sistemas, debe realizar varias llamadas para recopilar estos datos.

Sin embargo, este enfoque falla rápidamente con los casos de uso que requieren mayores volúmenes de datos.

Imagine un caso de uso en el que queremos informar sobre el comportamiento de compra de los clientes de nuestra tienda de música durante los últimos 24 meses, observando varias tendencias en el comportamiento de los clientes y cómo esto ha impactado en los ingresos. Necesitamos extraer grandes volúmenes de datos al menos de los sistemas de clientes y finanzas. Mantener una copia local de estos datos en el sistema de informes es peligroso, ya que es posible que no sepamos si ha cambiado (incluso los datos históricos pueden cambiar después del hecho), por lo que para generar un informe preciso necesitamos todos los registros financieros y de clientes por los últimos dos años. Incluso con un número modesto de clientes, puede ver que esto rápidamente se convertirá en una operación muy lenta.

Los sistemas de generación de informes también suelen depender de herramientas de terceros que esperan recuperar datos de cierta manera, y en este caso, proporcionar una interfaz SQL es la forma más rápida de garantizar que su cadena de herramientas de generación de informes sea lo más fácil de integrar posible. Todavía podríamos usar este enfoque para extraer datos periódicamente en una base de datos SQL, por supuesto, pero esto aún nos presenta algunos desafíos.

Uno de los desafíos clave es que las API expuestas por los diversos microservicios pueden no estar diseñadas para informar casos de uso. Por ejemplo, un servicio de atención al cliente puede permitirnos encontrar un cliente por una identificación, o buscar un cliente por varios campos, pero no necesariamente expondrá una API para recuperar a todos los clientes. Esto podría llevar a que se realicen muchas llamadas para recuperar todos los datos, por ejemplo, tener que recorrer una lista de todos los clientes, haciendo una llamada separada para cada uno. Esto no solo podría ser ineficiente para el sistema de informes, sino que también podría generar carga para el servicio en cuestión.

Si bien podríamos acelerar parte de la recuperación de datos agregando encabezados de caché a los recursos expuestos por nuestro servicio, y tener estos datos almacenados en caché en algo así como un proxy inverso, la naturaleza de los informes a menudo es que accedemos a la *cola larga* de datos. Esto significa que es posible que solicitemos recursos que nadie más ha solicitado antes (o al menos no durante un tiempo suficientemente largo), lo que resulta en una *falta de memoria caché potencialmente costosa*.

Puede resolver esto exponiendo las API por lotes para facilitar la generación de informes. Por ejemplo, nuestro servicio de atención al cliente podría permitirle pasar una lista de ID de clientes para recuperarlos en lotes, o incluso puede exponer una interfaz que le permita buscar a través de todos los clientes. Una versión más extrema de esto es modelar la solicitud por lotes como un recurso por derecho propio.

Por ejemplo, el servicio de atención al cliente podría exponer algo como un extremo de recursos de BatchCustomerExport. El sistema de llamada PUBLICARÍA una Solicitud por lotes, tal vez pasando un

ubicación donde se puede colocar un archivo con todos los datos. El servicio de atención al cliente devolvería un código de respuesta HTTP 202, indicando que la solicitud fue aceptada pero aún no ha sido procesada. Luego, el sistema de llamadas podría sondear el recurso que espera hasta que recupere un estado 201 Creado, lo que indica que la solicitud se cumplió, y luego el sistema de llamadas podría ir a buscar los datos. Esto permitiría exportar archivos de datos potencialmente grandes sin la sobrecarga de enviarlos a través de HTTP; en cambio, el sistema podría simplemente guardar un archivo CSV en una ubicación compartida.

He visto el enfoque anterior utilizado para la inserción de datos por lotes, donde funcionó bien.

Sin embargo, estoy menos a favor de los sistemas de generación de informes, ya que creo que existen otras soluciones potencialmente más simples que pueden escalar de manera más efectiva cuando se trata de necesidades de generación de informes tradicionales.

Bombas de datos

En lugar de que el sistema de informes extraiga los datos, podríamos enviar los datos al sistema de informes. Una de las desventajas de recuperar los datos mediante llamadas HTTP estándar es la sobrecarga de HTTP cuando realizamos una gran cantidad de llamadas, junto con la sobrecarga de tener que crear API que pueden existir solo para fines de informes. Una opción alternativa es tener un programa independiente que acceda directamente a la base de datos del servicio que es la fuente de datos y la bombee a una base de datos de informes, como se muestra en la [Figura 5-13](#).

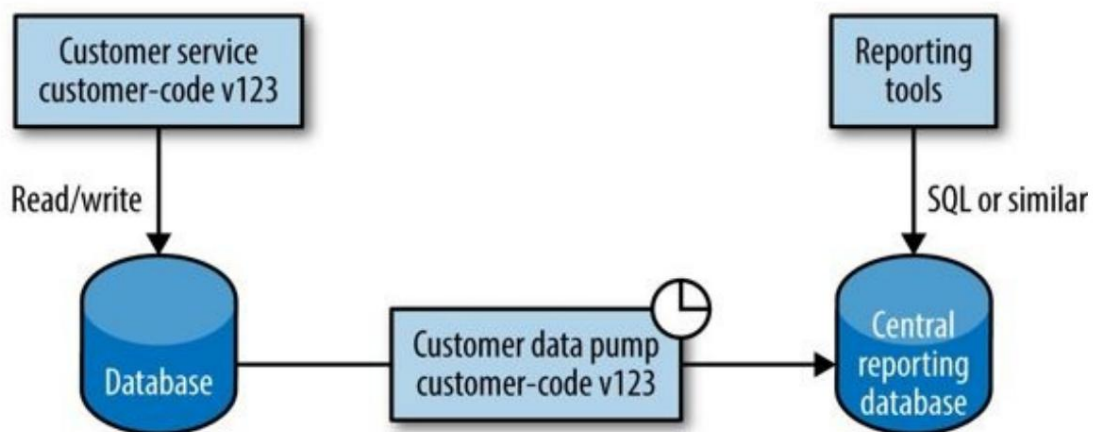


Figura 5-13. Uso de una bomba de datos para enviar datos periódicamente a una base de datos central de informes

En este punto estarás diciendo: “¡Pero Sam, dijiste que tener muchos programas integrados en la misma base de datos es una mala idea!” ¡Al menos *espero* que digas eso, dada la firmeza con la que lo dije antes! Este enfoque, si se implementa correctamente, es una excepción notable, donde las desventajas del acoplamiento se mitigan con creces al facilitar la elaboración de informes.

Para empezar, la bomba de datos debe ser construida y administrada por el mismo equipo que administra el servicio. Esto puede ser algo tan simple como un programa de línea de comandos activado a través de Cron. Este programa debe tener un conocimiento profundo tanto de la base de datos interna del servicio como del esquema de generación de informes. El trabajo de la bomba es mapear uno desde el otro. Intentamos reducir los problemas de acoplamiento al esquema del servicio haciendo que el mismo equipo que gestiona el servicio también gestione la bomba. Sugeriría, de hecho, que controle la versión de estos juntos, y que tenga compilaciones de la bomba de datos creadas como un artefacto adicional como parte de la compilación del servicio en sí, con la suposición de que cada vez que implementa uno de ellos, los implementa ambas cosas. Como declaramos explícitamente que los implementamos juntos y no abrimos el acceso al esquema a nadie fuera del equipo de servicio, muchos de los desafíos tradicionales de integración de bases de datos se mitigan en gran medida.

El acoplamiento en el esquema de informes en sí permanece, pero tenemos que tratarlo como una API publicada que es difícil de cambiar. Algunas bases de datos nos brindan técnicas en las que podríamos mitigar aún más este costo. La [Figura 5-14](#) muestra un ejemplo de esto para bases de datos relacionales, donde podríamos tener un esquema en la base de datos de informes para cada servicio, usando cosas como

vistas materializadas para crear la vista agregada. De esa forma, exponemos solo el esquema de informes para los datos del cliente a la bomba de datos del cliente. Sin embargo, si esto es algo que puede hacer de manera eficiente dependerá de las capacidades de la base de datos que eligió para generar informes.

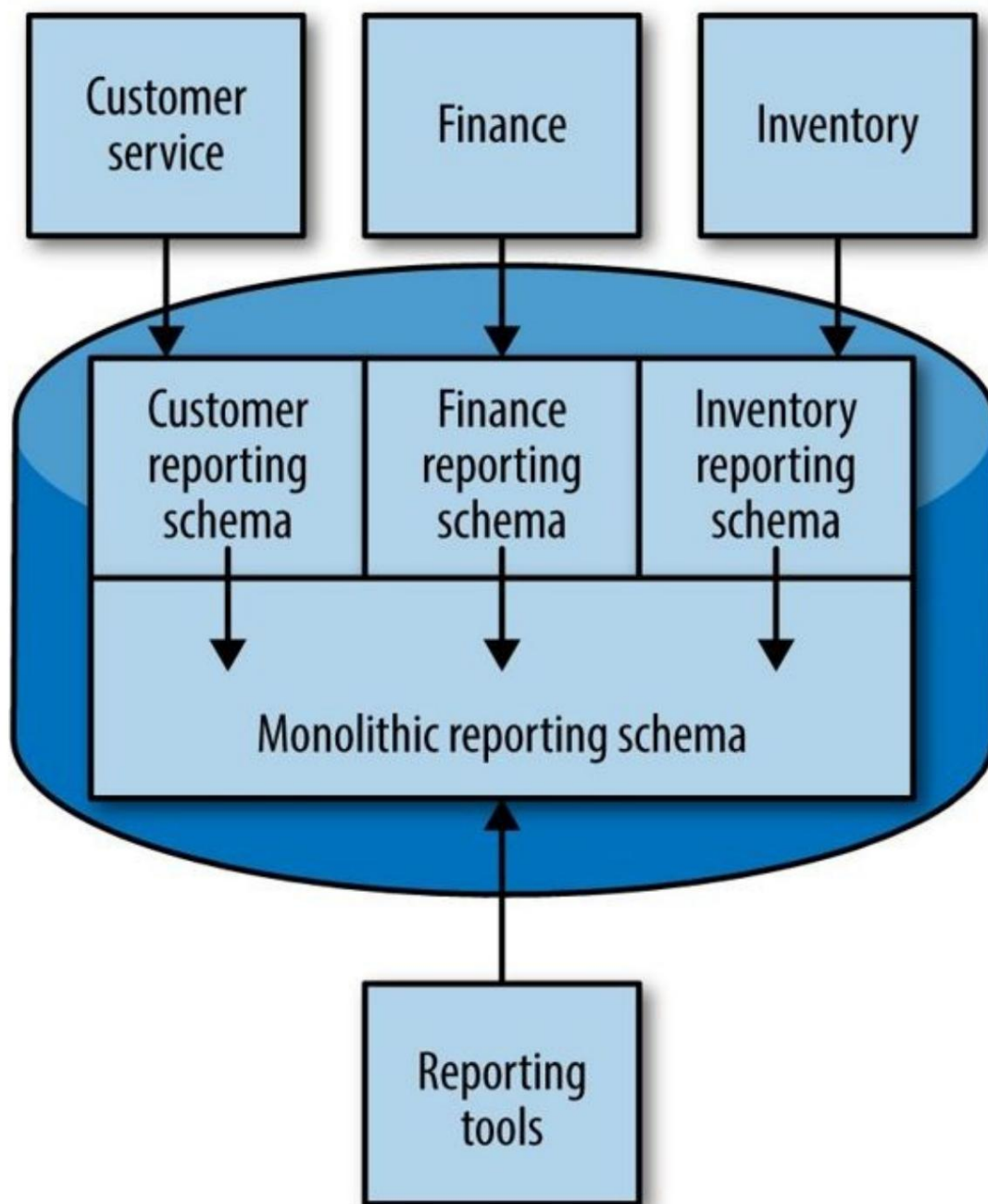


Figura 5-14. Utilización de vistas materializadas para formar un único esquema de informes monolítico

Aquí, por supuesto, la complejidad de la integración se profundiza en el esquema y se basará en las capacidades de la base de datos para hacer que dicha configuración funcione. Si bien creo que las bombas de datos en general son una sugerencia sensata y viable, estoy menos convencido de que la complejidad de un esquema segmentado valga la pena, especialmente dados los desafíos en la gestión de cambios en la base de datos.

Destinos alternativos

En un proyecto en el que participé, usamos una serie de bombas de datos para poblar archivos JSON en AWS S3, ¡efectivamente usando S3 para disfrazarnos de un data mart gigante! Este enfoque funcionó muy bien hasta que necesitábamos escalar nuestra solución y, en el momento de escribir este artículo, buscamos cambiar estas bombas para, en su lugar, llenar un cubo que se pueda integrar con herramientas de informes estándar como Excel y Tableau.

Bomba de datos de eventos

En el [Capítulo 4](#), abordamos la idea de los microservicios que emiten eventos en función del cambio de estado de las entidades que administran. Por ejemplo, nuestro servicio de atención al cliente puede emitir un evento cuando se crea, actualiza o elimina un cliente determinado. Para aquellos microservicios que exponen dichas fuentes de eventos, tenemos la opción de escribir nuestro propio suscriptor de eventos que envía datos a la base de datos de informes, como se muestra en la [Figura 5-15](#).

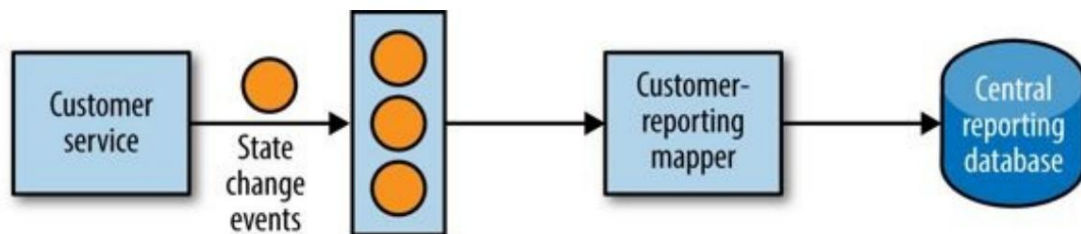


Figura 5-15. Una bomba de datos de eventos que usa eventos de cambio de estado para llenar una base de datos de informes

Ahora se evita el acoplamiento en la base de datos subyacente del microservicio de origen.

En cambio, solo nos vinculamos a los eventos emitidos por el servicio, que están diseñados para ser expuestos a consumidores externos. Dado que los eventos son de naturaleza temporal, también nos facilita ser más inteligentes en cuanto a los datos que enviamos a nuestro almacén central de informes; podemos enviar datos al sistema de informes cuando vemos un evento, lo que permite que los datos fluyan más rápido a nuestro sistema de informes, en lugar de depender de un programa regular como con la bomba de datos.

Además, si almacenamos qué eventos ya se han procesado, podemos procesar los nuevos eventos a medida que llegan, suponiendo que los eventos antiguos ya se hayan mapeado en el sistema de informes. Esto significa que nuestra inserción será más eficiente, ya que solo necesitamos enviar deltas.

Podemos hacer cosas similares con una bomba de datos, pero tenemos que administrar esto nosotros mismos, mientras que la naturaleza fundamentalmente temporal del flujo de eventos (x sucede en la marca de tiempo y) nos ayuda mucho.

Como nuestra bomba de datos de eventos está menos acoplada a las partes internas del servicio, también es más fácil considerar que la administre un grupo separado del equipo que se ocupa del microservicio en sí. Siempre que la naturaleza de nuestro flujo de eventos no asocie demasiado a los suscriptores a los cambios en el servicio, este mapeador de eventos puede evolucionar independientemente del servicio al que se suscriba.

Las principales desventajas de este enfoque son que toda la información requerida debe transmitirse como eventos y es posible que no se escale tan bien como una bomba de datos para volúmenes más grandes de datos que tiene la ventaja de operar directamente en el nivel de la base de datos. No obstante, el acoplamiento más flexible y los datos más recientes disponibles a través de este enfoque hacen que valga la pena considerarlo si ya está exponiendo los eventos apropiados.

Bomba de datos de respaldo

Esta opción se basa en un enfoque utilizado en Netflix, que aprovecha las soluciones de respaldo existentes y también resuelve algunos problemas de escala con los que Netflix tiene que lidiar. De alguna manera, puede considerar esto como un caso especial de una bomba de datos, pero parecía una solución tan interesante que merece ser incluida.

Netflix ha decidido estandarizar a Cassandra como la tienda de respaldo para sus servicios, de los cuales hay muchos. Netflix ha invertido mucho tiempo en la creación de herramientas para facilitar el trabajo con Cassandra, muchas de las cuales la empresa ha compartido con el resto del mundo a través de numerosos proyectos de código abierto. Obviamente, es muy importante que los datos que almacena Netflix estén correctamente respaldados. Para hacer una copia de seguridad de los datos de Cassandra, el enfoque estándar es hacer una copia de los archivos de datos que la respaldan y almacenarlos en un lugar seguro. Netflix almacena estos archivos, conocidos como SSTables, en el almacén de objetos S3 de Amazon, lo que brinda importantes garantías de durabilidad de los datos.

Netflix necesita informar sobre todos estos datos, pero dada la escala involucrada, este es un desafío no trivial. Su enfoque es utilizar Hadoop que utiliza la copia de seguridad de SSTable como origen de sus trabajos. Al final, Netflix terminó implementando una canalización capaz de procesar grandes cantidades de datos utilizando este enfoque, que luego abrió como **proyecto Aegisthus**.

Sin embargo, al igual que las bombas de datos, con este patrón todavía tenemos un acoplamiento con el esquema de informes de destino (o sistema de destino).

Es concebible que el uso de un enfoque similar, es decir, el uso de mapeadores que funcionan con copias de seguridad, también funcione en otros contextos. Y si ya estás usando Cassandra, ¡Netflix ya ha hecho gran parte del trabajo por ti!

Hacia el tiempo real

Muchos de los patrones descritos anteriormente son formas diferentes de obtener una gran cantidad de datos de muchos lugares diferentes en un solo lugar. Pero, ¿la idea de que todos nuestros informes se realizarán desde una ubicación realmente se acumula más? Tenemos tableros, alertas, informes financieros, análisis de usuarios: todos estos casos de uso tienen diferentes tolerancias de precisión y puntualidad, lo que puede resultar en diferentes opciones técnicas. Como detallaré en el **Capítulo 8**, nos estamos moviendo cada vez más hacia sistemas de eventos genéricos capaces de enrutar nuestros datos a múltiples lugares diferentes según la necesidad.

Costo del cambio

Hay muchas razones por las que, a lo largo del libro, promuevo la necesidad de hacer cambios pequeños e incrementales, pero uno de los impulsores clave es comprender el impacto de cada alteración que hacemos y cambiar de rumbo si es necesario. Esto nos permite mitigar mejor el costo de los errores, pero no elimina por completo la posibilidad de errores. Podemos, y lo haremos, cometer errores, y debemos aceptar eso. Sin embargo, lo que también debemos hacer es entender cuál es la mejor manera de mitigar los costos de esos errores.

Como hemos visto, el costo involucrado en mover el código dentro de una base de código es bastante pequeño. Tenemos muchas herramientas que nos respaldan, y si causamos un problema, la solución generalmente es rápida. Sin embargo, dividir una base de datos es mucho más trabajo, y revertir un cambio de base de datos es igual de complejo. Del mismo modo, desenredar una integración excesivamente acoplada entre servicios, o tener que reescribir por completo una API que utilizan múltiples consumidores, puede ser una tarea considerable. El gran costo del cambio hace que estas operaciones sean cada vez más arriesgadas. ¿Cómo podemos gestionar este riesgo? Mi enfoque es tratar de cometer errores donde el impacto será menor.

Tiendo a pensar mucho en el lugar donde el costo del cambio y el costo de los errores es lo más bajo posible: la pizarra. Haz un bosquejo de tu diseño propuesto. Vea lo que sucede cuando ejecuta casos de uso en lo que cree que serán los límites de su servicio.

Para nuestra tienda de música, por ejemplo, imagine lo que sucede cuando un cliente busca un disco, se registra en el sitio web o compra un álbum. ¿Qué llamadas se hacen? ¿Empiezas a ver extrañas referencias circulares? ¿Ves dos servicios que hablan demasiado, lo que podría indicar que deberían ser una sola cosa?

Una gran técnica aquí es adaptar un enfoque más típicamente enseñado para el diseño de sistemas orientados a objetos: tarjetas de colaboración de responsabilidad de clase (CRC). Con las tarjetas CRC, escribe en una tarjeta el nombre de la clase, cuáles son sus responsabilidades y con quién colabora. Al trabajar en un diseño propuesto, para cada servicio enumero sus responsabilidades en términos de las capacidades que brinda, con los colaboradores especificados en el diagrama. A medida que trabaja en más casos de uso, comienza a tener una idea de si todo esto encaja correctamente.

Comprender las causas fundamentales

Hemos discutido cómo dividir los servicios más grandes en otros más pequeños, pero ¿por qué estos servicios crecieron tanto en primer lugar? Lo primero que debe entender es que hacer crecer un servicio hasta el punto en que debe dividirse está completamente bien. Queremos *que* la arquitectura de nuestro sistema cambie con el tiempo de forma incremental. La clave es saber que debe dividirse antes de que la división se vuelva demasiado costosa.

Pero en la práctica, muchos de nosotros habremos visto crecer los servicios mucho más allá del punto de la cordura. A pesar de saber que un conjunto más pequeño de servicios sería más fácil de manejar que la enorme monstruosidad que tenemos actualmente, aún seguimos con el crecimiento de la bestia. ¿Por qué?

Parte del problema es saber por dónde empezar, y espero que este capítulo le haya ayudado. Pero otro desafío es el costo asociado con la división de servicios. Encontrar un lugar para ejecutar el servicio, poner en marcha una nueva pila de servicios, etc., no son tareas triviales. Entonces, ¿cómo abordamos esto? Bueno, si hacer algo es correcto pero difícil, debemos esforzarnos por hacer las cosas más fáciles. La inversión en bibliotecas y marcos de servicio livianos puede reducir el costo asociado con la creación del nuevo servicio. Brindar a las personas acceso a máquinas virtuales de aprovisionamiento de autoservicio o incluso hacer que una plataforma como servicio (PaaS) esté disponible facilitará el aprovisionamiento de sistemas y su prueba. A lo largo del resto del libro, analizaremos varias formas de ayudarlo a mantener este costo bajo.

Resumen

Descomponemos nuestro sistema encontrando costuras a lo largo de las cuales pueden surgir los límites del servicio, y este puede ser un enfoque incremental. Al ser buenos para encontrar estas costuras y trabajar para reducir el costo de dividir los servicios en primer lugar, podemos continuar creciendo y evolucionando nuestros sistemas para cumplir con los requisitos que surjan en el camino. Como puede ver, parte de este trabajo puede ser laborioso. Pero el hecho mismo de que se pueda hacer de forma incremental significa que no hay necesidad de temer este trabajo.

Ahora podemos dividir nuestros servicios, pero también hemos introducido algunos problemas nuevos.

¡Tenemos muchas más piezas móviles para entrar en producción ahora! A continuación, nos sumergiremos en el mundo de la implementación.

Capítulo 6. Despliegue

La implementación de una aplicación monolítica es un proceso bastante sencillo. Los microservicios, con su interdependencia, son una olla de pescado completamente diferente. Si no aborda la implementación correctamente, es una de esas áreas donde la complejidad puede convertir su vida en una miseria. En este capítulo, veremos algunas técnicas y tecnologías que pueden ayudarnos al implementar microservicios en arquitecturas detalladas.

Sin embargo, vamos a comenzar echando un vistazo a la integración continua y la entrega continua. Estos conceptos relacionados pero diferentes ayudarán a dar forma a las otras decisiones que tomaremos al pensar en qué construir, cómo construirlo y cómo implementarlo.

Una breve introducción a la integración continua

La integración continua (CI) ha existido durante varios años en este momento. Sin embargo, vale la pena dedicar un poco de tiempo a repasar los conceptos básicos, especialmente cuando pensamos en el mapeo entre microservicios, compilaciones y repositorios de control de versiones, hay algunas opciones diferentes a considerar.

Con CI, el objetivo principal es mantener a todos sincronizados entre sí, lo que logramos asegurándonos de que el código recién registrado se integre correctamente con el código existente. Para hacer esto, un servidor de CI detecta que el código se ha confirmado, lo verifica y lleva a cabo alguna verificación, como asegurarse de que el código se compile y pasen las pruebas.

Como parte de este proceso, a menudo creamos artefactos que se utilizan para una mayor validación, como implementar un servicio en ejecución para ejecutar pruebas en él. Idealmente, queremos construir estos artefactos una sola vez y usarlos para todas las implementaciones de esa versión del código.

Esto es para evitar hacer lo mismo una y otra vez, y para que podamos confirmar que el artefacto que implementamos es el que probamos. Para permitir que estos artefactos se reutilicen, los colocamos en un depósito de algún tipo, ya sea proporcionado por la propia herramienta de CI o en un sistema separado.

Veremos qué tipo de artefactos podemos usar para los microservicios en breve, y analizaremos en profundidad las pruebas en el [Capítulo 7](#).

CI tiene una serie de beneficios. Recibimos cierto nivel de retroalimentación rápida en cuanto a la calidad de nuestro código. Nos permite automatizar la creación de nuestros artefactos binarios. Todo el código necesario para crear el artefacto está controlado por versiones, por lo que podemos volver a crear el artefacto si es necesario. También obtenemos cierto nivel de trazabilidad desde un artefacto implementado hasta el código y, según las capacidades de la herramienta de CI en sí, podemos ver qué pruebas se ejecutaron en el código y el artefacto también. Es por estas razones que CI ha tenido tanto éxito.

¿Realmente lo estás haciendo?

Sospecho que probablemente esté utilizando la integración continua en su propia organización. Si no, deberías empezar. Es una práctica clave que nos permite realizar cambios de forma rápida y sencilla, y sin la cual el camino hacia los microservicios será doloroso. Dicho esto, he trabajado con muchos equipos que, a pesar de decir que hacen IC, en realidad no lo hacen en absoluto. Confunden el uso de una herramienta de IC con adoptar la práctica de IC. La herramienta es solo algo que permite el enfoque.

Me gustan mucho las tres preguntas de Jez Humble que le hace a la gente para comprobar si realmente entienden de qué se trata la IC:

¿Te registras en la línea principal una vez al día?

Debe asegurarse de que su código se integre. Si no revisa su código junto con los cambios de todos los demás con frecuencia, terminará dificultando la integración futura.

Incluso si está utilizando sucursales de corta duración para administrar los cambios, integre con la mayor frecuencia posible en una sola sucursal principal.

¿Tiene un conjunto de pruebas para validar sus cambios?

Sin pruebas, solo sabemos que sintácticamente nuestra integración ha funcionado, pero no sabemos si hemos roto el comportamiento del sistema. CI sin alguna verificación de que nuestro código se comporta como se esperaba no es CI.

Cuando la construcción se rompe, ¿es la prioridad número 1 del equipo arreglarla?

Una compilación ecológica aprobada significa que nuestros cambios se han integrado de manera segura. Una compilación roja significa que el último cambio posiblemente no se integró. Debe detener todos los registros adicionales que no estén involucrados en la reparación de las compilaciones para que vuelva a pasar. Si deja que se acumulen más cambios, el tiempo necesario para arreglar la compilación aumentará drásticamente. He trabajado con equipos en los que la compilación se ha interrumpido durante días, lo que resultó en esfuerzos sustanciales para finalmente obtener una compilación aprobada.

Asignación de integración continua a microservicios

Al pensar en los microservicios y la integración continua, debemos pensar en cómo nuestra CI crea un mapa para los microservicios individuales. Como he dicho muchas veces, queremos asegurarnos de que podemos realizar un cambio en un solo servicio e implementarlo independientemente del resto. Con esto en mente, ¿cómo debemos asignar microservicios individuales a compilaciones de CI y código fuente?

Si comenzamos con la opción más simple, podríamos agrupar todo junto. Tenemos un único repositorio gigante que almacena todo nuestro código y tenemos una sola compilación, como vemos en la [Figura 6-1](#). Cualquier registro en este repositorio de código fuente hará que nuestra compilación se active, donde ejecutaremos todos los pasos de verificación asociados con todos nuestros microservicios y produciremos múltiples artefactos, todos vinculados a la misma compilación.

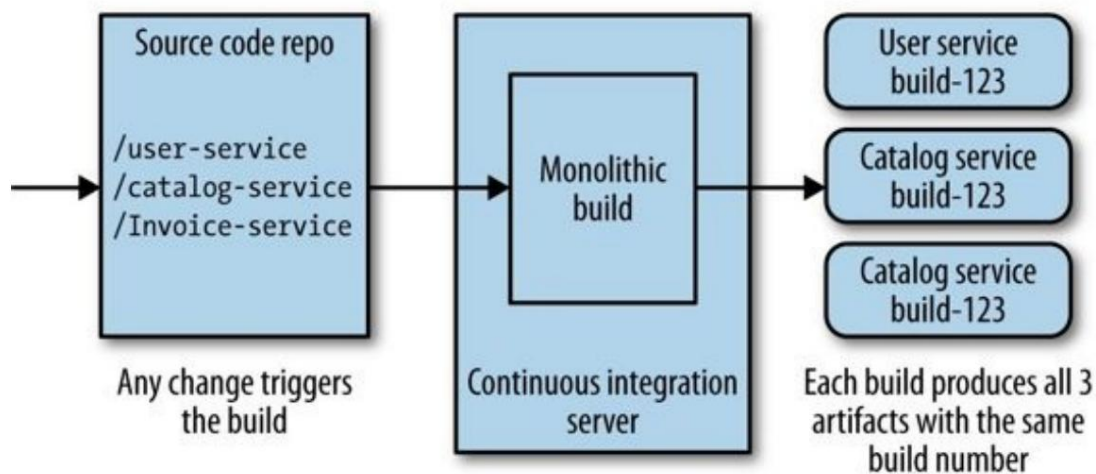


Figura 6-1. Uso de un único repositorio de código fuente y compilación de CI para todos los microservicios

Esto parece mucho más simple en la superficie que otros enfoques: menos repositorios de los que preocuparse y una construcción conceptualmente más simple. Desde el punto de vista del desarrollador, las cosas también son bastante sencillas. Solo reviso el código. Si tengo que trabajar en varios servicios a la vez, solo tengo que preocuparme por una confirmación.

Este modelo puede funcionar perfectamente bien si acepta la idea de lanzamientos de pasos fijos, en los que no le importa implementar varios servicios a la vez. En general, este es un patrón que se debe evitar, pero muy al principio de un proyecto, especialmente si solo un equipo está trabajando en todo, esto podría tener sentido por períodos cortos de tiempo.

Sin embargo, hay algunas desventajas significativas. Si realizo un cambio de una línea en un solo servicio, por ejemplo, cambiando el comportamiento en el servicio de usuario en la [Figura 6-1](#), todos los demás servicios se verifican y construyen. Esto podría tomar más tiempo del necesario. Estoy esperando cosas que probablemente no necesiten ser probadas. Esto afecta nuestro tiempo de ciclo, la velocidad a la que podemos mover un solo cambio del desarrollo a la vida. Sin embargo, lo más preocupante es saber qué artefactos deben o no implementarse. ¿Necesito ahora implementar todos los servicios de compilación para llevar mi pequeño cambio a producción? Puede ser difícil saberlo; tratar de adivinar qué servicios *realmente* cambiaron simplemente leyendo los mensajes de confirmación es

difícil. Las organizaciones que utilizan este enfoque a menudo recurren a simplemente implementar todo junto, lo que realmente queremos evitar.

Además, si mi cambio de una línea en el servicio de usuario interrumpe la compilación, no se pueden realizar otros cambios en los otros servicios hasta que se solucione ese quiebre. Y piense en un escenario en el que tenga varios equipos que compartan esta construcción gigante. ¿Quién está a cargo?

Una variación de este enfoque es tener un solo árbol fuente con todo el código en él, con múltiples compilaciones de CI mapeadas a partes de este árbol fuente, como vemos en la **Figura 6-2**. Con una estructura bien definida, puede asignar fácilmente las compilaciones a ciertas partes del árbol de fuentes. En general, no soy partidario de este enfoque, ya que este modelo puede ser una bendición a medias. Por un lado, mi proceso de check-in/check-out puede ser más simple ya que solo tengo un repositorio del que preocuparme. Por otro lado, se vuelve muy fácil adquirir el hábito de verificar el código fuente para múltiples servicios a la vez, lo que puede hacer que sea igualmente fácil realizar cambios que combinen servicios. Sin embargo, preferiría mucho este enfoque a tener una sola compilación para múltiples servicios.

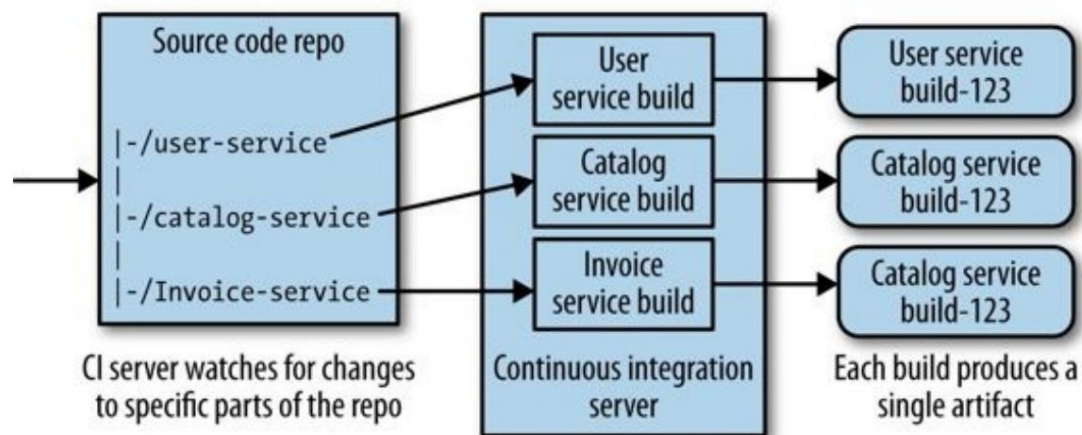


Figura 6-2. Un repositorio de fuente única con subdirectorios asignados a compilaciones independientes

Entonces, ¿hay otra alternativa? El enfoque que prefiero es tener una sola compilación de CI por microservicio, que nos permita realizar y validar rápidamente un cambio antes de la implementación en producción, como se muestra en la **Figura 6-3**. Aquí, cada microservicio tiene su propio repositorio de código fuente, asignado a su propia compilación de CI. Al realizar un cambio, solo ejecuto la compilación y las pruebas que necesito. Obtengo un solo artefacto para implementar. La alineación con la propiedad del equipo también es más clara. Si es propietario del servicio, es propietario del repositorio y de la compilación. Hacer cambios en los repositorios puede ser más difícil en este mundo, pero mantendría que esto es más fácil de resolver (por ejemplo, mediante el uso de scripts de línea de comandos) que la desventaja del control de código fuente monolítico y el proceso de construcción.

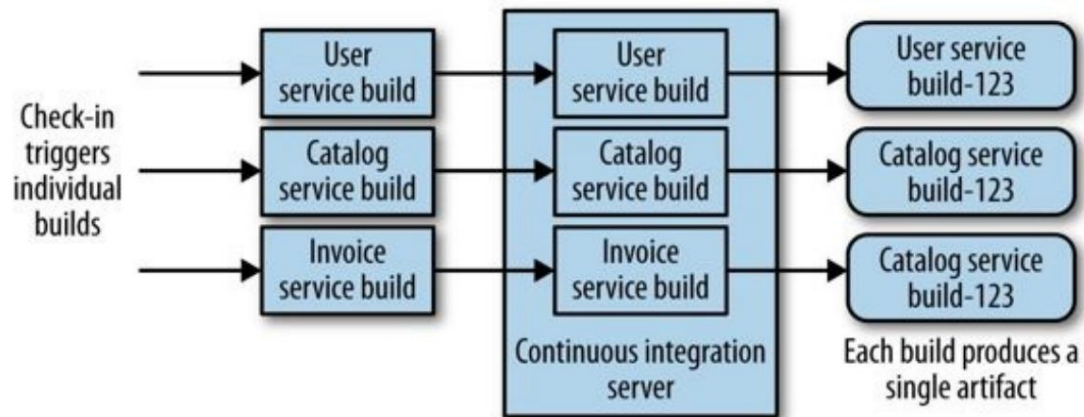


Figura 6-3. Usar un repositorio de código fuente y compilación de CI por microservicio

Las pruebas para un microservicio determinado también deben vivir en el control de código fuente con el código fuente del microservicio, para garantizar que siempre sepamos qué pruebas se deben ejecutar en un servicio determinado.

Por lo tanto, cada microservicio vivirá en su propio repositorio de código fuente y su propio proceso de creación de CI. Usaremos el proceso de compilación de CI para crear nuestros artefactos desplegables también de forma totalmente automatizada. Ahora miremos más allá de CI para ver cómo encaja la entrega continua.

Construir canalizaciones y entrega continua

Muy pronto en el uso de la integración continua, nos dimos cuenta del valor de tener a veces varias etapas dentro de una compilación. Las pruebas son un caso muy común donde esto entra en juego. Es posible que tenga muchas pruebas rápidas de alcance pequeño y una pequeña cantidad de pruebas lentas de alcance amplio.

Si ejecutamos todas las pruebas juntas, es posible que no podamos obtener comentarios rápidos cuando nuestras pruebas rápidas fallan si estamos esperando que nuestras pruebas lentas de largo alcance finalmente terminen. Y si las pruebas rápidas fallan, ¡probablemente no tenga mucho sentido ejecutar las pruebas más lentas de todos modos! Una solución a este problema es tener diferentes etapas en nuestra compilación, creando lo que se conoce como *canalización de compilación*. Una etapa para las pruebas más rápidas, otra para las pruebas más lentas.

Este concepto de canalización de compilación nos brinda una buena manera de rastrear el progreso de nuestro software a medida que supera cada etapa, lo que nos ayuda a comprender la calidad de nuestro software. Construimos nuestro artefacto, y ese artefacto se usa en toda la canalización. A medida que nuestro artefacto avanza por estas etapas, nos sentimos cada vez más seguros de que el software funcionará en producción.

La *entrega continua (CD)* se basa en este concepto y algo más. Como se describe en el libro del mismo nombre de Jez Humble y Dave Farley, la entrega continua es el enfoque mediante el cual recibimos comentarios constantes sobre la preparación de producción de todos y cada uno de los registros y, además, tratamos todos y cada uno de los registros como un candidato de lanzamiento.

Para adoptar completamente este concepto, necesitamos modelar todos los procesos involucrados en llevar nuestro software desde el registro hasta la producción, y saber dónde se encuentra cualquier versión del software en términos de aprobación para su lanzamiento. En CD, hacemos esto extendiendo la idea de la canalización de compilación de varias etapas para modelar todas y cada una de las etapas por las que debe pasar nuestro software, tanto manuales como automatizadas. En la **Figura 6-4**, vemos una canalización de muestra que puede ser familiar.



Figura 6-4. Un proceso de lanzamiento estándar modelado como una canalización de compilación

Aquí realmente queremos una herramienta que adopte el CD como un concepto de primera clase. He visto a muchas personas intentar piratear y ampliar las herramientas de CI para que funcionen en CD, lo que a menudo da como resultado sistemas complejos que no son tan fáciles de usar como las herramientas que se construyen en CD desde el principio. Las herramientas que son totalmente compatibles con CD le permiten definir y visualizar estas canalizaciones, modelando todo el camino hacia la producción de su software. A medida que una versión de nuestro código se mueve a través de la canalización, si pasa uno de estos pasos de verificación automatizados, pasa a la siguiente etapa. Otras etapas pueden ser manuales. Por ejemplo, si tenemos un proceso de prueba de aceptación del usuario (UAT) manual, debería poder usar una herramienta de CD para modelarlo. Puedo ver la próxima compilación disponible lista para implementarse en nuestro entorno UAT, implementarla y, si pasa nuestras comprobaciones manuales, marcar esa etapa como exitosa para que pueda pasar a la siguiente.

Al modelar todo el camino hacia la producción de nuestro software, mejoramos en gran medida la visibilidad de la calidad de nuestro software y también podemos reducir en gran medida el tiempo que transcurre entre las versiones.

ya que tenemos un lugar para observar nuestro proceso de compilación y lanzamiento, y un punto focal obvio para introducir mejoras.

En un mundo de microservicios, donde queremos asegurarnos de que podemos lanzar nuestros servicios de forma independiente, se deduce que, al igual que con CI, queremos una canalización por servicio. En nuestras canalizaciones, es un artefacto que queremos crear y mover a través de nuestro camino hacia la producción. Como siempre, resulta que nuestros artefactos pueden venir en muchos tamaños y formas. Veremos algunas de las opciones más comunes disponibles para nosotros en un momento.

Y las excepciones inevitables

Como con todas las buenas reglas, también hay excepciones que debemos considerar. El enfoque de "un microservicio por compilación" es absolutamente algo a lo que debe apuntar, pero ¿hay momentos en que algo más tiene sentido? Cuando un equipo comienza con un nuevo proyecto, especialmente uno completamente nuevo en el que trabajan con una hoja de papel en blanco, es muy probable que haya una gran cantidad de cambios en términos de determinar dónde se encuentran los límites del servicio. Esta es una buena razón, de hecho, para mantener sus servicios iniciales en el lado más grande hasta que se establezca su comprensión del dominio.

Durante este tiempo de abandono, es más probable que se produzcan cambios en los límites del servicio, y es probable que lo que está o no en un servicio determinado cambie con frecuencia. Durante este período, puede tener sentido tener todos los servicios en una sola compilación para reducir el costo de los cambios entre servicios.

Sin embargo, se deduce que, en este caso, debe comprar el lanzamiento de todos los servicios como un paquete. También tiene que ser absolutamente un paso de transición. A medida que las API de servicio se estabilicen, comience a trasladarlas a sus propias compilaciones. Si después de unas pocas semanas (o una cantidad muy pequeña de meses) no puede obtener estabilidad en los límites del servicio para poder separarlos adecuadamente, vuélvalos a fusionar en un servicio más monolítico (aunque conservando la separación modular dentro del límite) y dése hora de familiarizarse con el dominio.

Esto refleja las experiencias de nuestro propio equipo SnapCI, como discutimos en el [Capítulo 3](#).

Artefactos específicos de la plataforma

La mayoría de las pilas de tecnología tienen algún tipo de artefacto de primera clase, junto con herramientas para ayudar a crearlas e instalarlas. Ruby tiene gemas, Java tiene archivos JAR y archivos WAR, y Python tiene huevos. Los desarrolladores con experiencia en una de estas pilas sabrán trabajar con (y con suerte crear) estos artefactos.

Sin embargo, desde el punto de vista de un microservicio, dependiendo de su pila de tecnología, este artefacto puede no ser suficiente por sí solo. Mientras que un archivo JAR de Java se puede hacer para que sea ejecutable y ejecute un proceso HTTP incorporado, para aplicaciones como Ruby y Python, esperará usar un administrador de procesos que se ejecute dentro de Apache o Nginx. Por lo tanto, es posible que necesitemos alguna forma de instalar y configurar otro software que necesitemos para implementar y lanzar nuestros artefactos. Aquí es donde las herramientas de administración de configuración automatizada como Puppet y Chef pueden ayudar.

Otra desventaja aquí es que estos artefactos son específicos de una determinada pila de tecnología, lo que puede dificultar la implementación cuando tenemos una combinación de tecnologías en juego. Piénselo desde el punto de vista de alguien que intenta implementar varios servicios juntos.

Podría ser un desarrollador o evaluador que desee probar alguna funcionalidad, o podría ser alguien que administre una implementación de producción. Ahora imagine que esos servicios usan tres mecanismos de implementación completamente diferentes. Tal vez tengamos un Ruby Gem, un archivo JAR y un paquete NPM de nodeJS. ¿Te lo agradecerían?

La automatización puede contribuir en gran medida a ocultar las diferencias en los mecanismos de implementación de los artefactos subyacentes. Chef, Puppet y Ansible también son compatibles con varios artefactos de compilación comunes específicos de la tecnología. Pero hay diferentes tipos de artefactos con los que podría ser aún más fácil trabajar.

Artefactos del sistema operativo

Una forma de evitar los problemas asociados con los artefactos específicos de la tecnología es crear artefactos que sean nativos del sistema operativo subyacente. Por ejemplo, para un sistema basado en RedHat o CentOS, podría crear RPM; para Ubuntu, podría construir un paquete deb; o para Windows, un MSI.

La ventaja de usar artefactos específicos del sistema operativo es que, desde el punto de vista de la implementación, no nos importa cuál es la tecnología subyacente. Solo usamos las herramientas nativas del sistema operativo para instalar el paquete. Las herramientas del sistema operativo también pueden ayudarnos a desinstalar y obtener información sobre los paquetes, e incluso pueden proporcionar repositorios de paquetes a los que pueden enviar nuestras herramientas de CI. Gran parte del trabajo realizado por el administrador de paquetes del sistema operativo también puede compensar el trabajo que de otro modo podría realizar en una herramienta como Puppet o Chef. En todas las plataformas Linux que he usado, por ejemplo, puede definir dependencias de sus paquetes a otros paquetes en los que confía, y las herramientas del sistema operativo también las instalarán automáticamente.

La desventaja puede ser la dificultad para crear los paquetes en primer lugar. Para Linux, la **herramienta del administrador de paquetes FPM** brinda una mejor abstracción para crear paquetes de SO Linux, y la conversión de una implementación basada en tarball a una implementación basada en SO puede ser bastante sencilla. El espacio de Windows es algo más complicado. El sistema de empaquetado nativo en forma de instaladores MSI y similares deja mucho que desear en comparación con las capacidades en el espacio de Linux. El sistema de paquetes NuGet ha comenzado a ayudar a abordar esto, al menos en términos de ayudar a administrar las bibliotecas de desarrollo. Más recientemente, Chocolatey NuGet ha ampliado estas ideas, proporcionando un administrador de paquetes para Windows diseñado para implementar herramientas y servicios, que se parece mucho más a los administradores de paquetes en el espacio de Linux. Sin duda, este es un paso en la dirección correcta, aunque el hecho de que el estilo idiomático en Windows todavía *implemente algo en IIS* significa que este enfoque puede ser poco atractivo para algunos equipos de Windows.

Otro inconveniente, por supuesto, podría ser si está implementando en múltiples sistemas operativos diferentes.

La sobrecarga de administrar artefactos para diferentes sistemas operativos podría ser bastante elevada. Si está creando software para que otras personas lo instalen, es posible que no tenga otra opción. Sin embargo, si está instalando software en las máquinas que controla, le sugiero que busque unificar o al menos reducir la cantidad de sistemas operativos diferentes que usa. Puede reducir en gran medida las variaciones en el comportamiento de una máquina a la siguiente y simplificar las tareas de implementación y mantenimiento.

En general, los equipos que he visto que se han pasado a la administración de paquetes basada en el sistema operativo han simplificado su enfoque de implementación y tienden a evitar la trampa de los scripts de implementación grandes y complejos. Especialmente si está en Linux, esta puede ser una buena manera de simplificar la implementación de microservicios utilizando pilas de tecnología dispares.

Imágenes personalizadas

Uno de los desafíos con los sistemas de gestión de configuración automatizados como Puppet, Chef y Ansible puede ser el tiempo necesario para ejecutar los scripts en una máquina. Tomemos un ejemplo simple de un servidor que se aprovisiona y configura para permitir la implementación de una aplicación Java. Supongamos que estoy usando AWS para aprovisionar el servidor, usando la imagen estándar de Ubuntu. Lo primero que debo hacer es instalar Oracle JVM para ejecutar mi aplicación Java. He visto que este proceso simple toma alrededor de cinco minutos, con un par de minutos ocupados por la máquina que se aprovisiona y algunos más para instalar la JVM.

Entonces podemos pensar en poner nuestro software en él.

Este es en realidad un ejemplo bastante trivial. A menudo queremos instalar otros bits comunes de software. Por ejemplo, es posible que deseemos usar collectd para recopilar estadísticas del sistema operativo, usar logstash para agregar registros y tal vez instalar los bits apropiados de nagios para monitorear (hablaremos más sobre este software en el [Capítulo 8](#)). Con el tiempo, es posible que se agreguen más cosas, lo que lleva a que se necesiten cantidades de tiempo cada vez más largas para el aprovisionamiento de estas dependencias.

Puppet, Chef, Ansible y los de su clase pueden ser inteligentes y evitarán instalar software que ya está presente. Esto no significa que ejecutar los scripts en las máquinas existentes siempre será rápido, desafortunadamente, ya que ejecutar todas las comprobaciones lleva tiempo. También queremos evitar mantener nuestras máquinas por mucho tiempo, ya que no queremos permitir demasiada desviación de la configuración (que exploraremos con mayor profundidad en breve). Y si usamos una plataforma de cómputo bajo demanda, es posible que estemos constantemente apagando y activando nuevas instancias diariamente (si no con más frecuencia), por lo que la naturaleza declarativa de estas herramientas de administración de configuración puede tener un uso limitado.

Con el tiempo, ver cómo se instalan las mismas herramientas una y otra vez puede convertirse en un verdadero lastre. Si intenta hacer esto varias veces al día, tal vez como parte del desarrollo o CI, esto se convierte en un problema real en términos de proporcionar comentarios rápidos. También puede generar un mayor tiempo de inactividad al implementar en producción si sus sistemas no permiten una implementación sin tiempo de inactividad, ya que está esperando instalar todos los requisitos previos en sus máquinas incluso antes de instalar su software. Los modelos como la implementación azul/verde (que analizaremos en el [Capítulo 7](#)) pueden ayudar a mitigar esto, ya que nos permiten implementar una nueva versión de nuestro servicio sin desconectar la anterior.

Un enfoque para reducir este tiempo de activación es crear una imagen de máquina virtual que se integre en algunas de las dependencias comunes que usamos, como se muestra en la [Figura 6-5](#). Todas las plataformas de virtualización que he usado te permiten crear tus propias imágenes, y las herramientas para hacerlo son mucho más avanzadas que hace unos años. Esto cambia un poco las cosas. Ahora podríamos hornear las herramientas comunes a nuestra propia imagen. Cuando queremos implementar nuestro software, activamos una instancia de esta imagen personalizada y todo lo que tenemos que hacer es instalar la última versión de nuestro servicio.

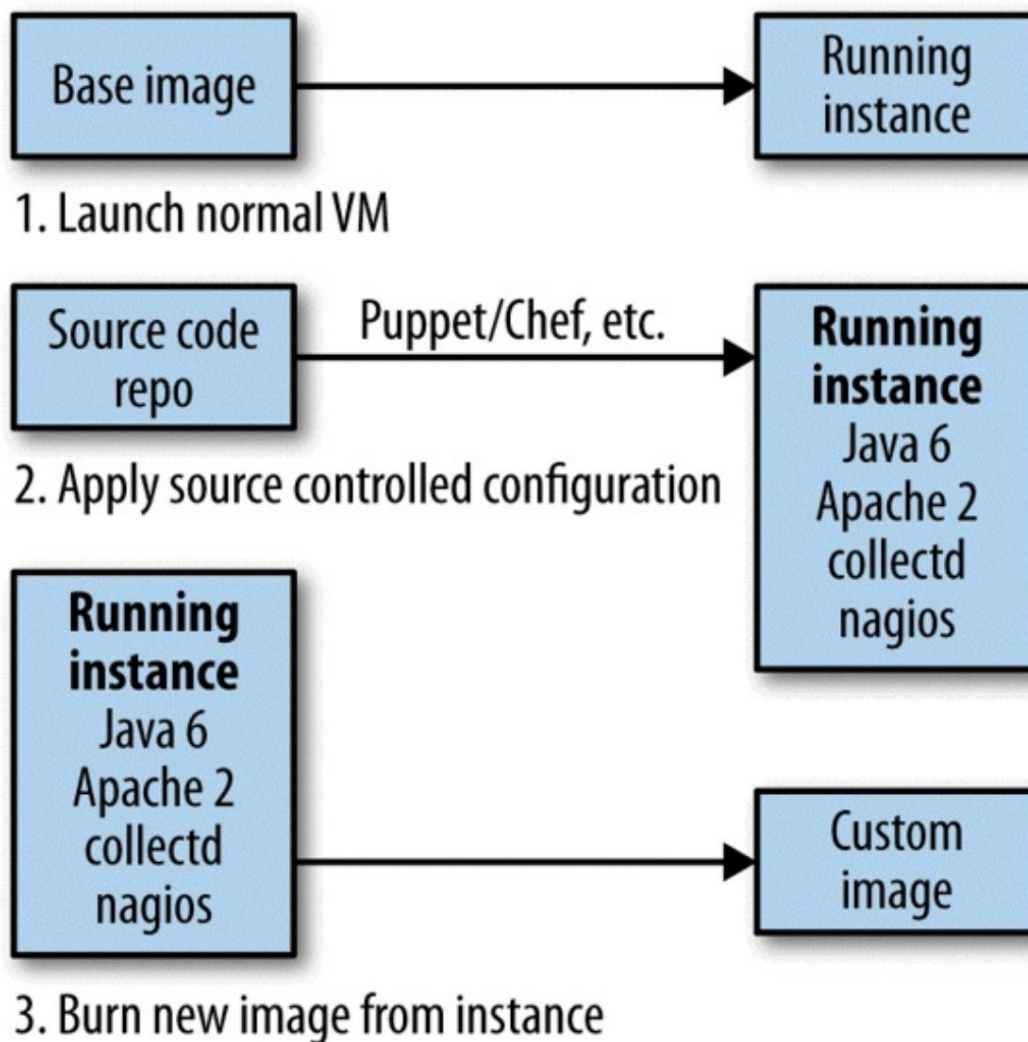


Figura 6-5. Creación de una imagen de máquina virtual personalizada

Por supuesto, debido a que crea la imagen solo una vez, cuando lanza copias de esta imagen posteriormente, no necesita perder tiempo instalando sus dependencias, ya que ya están allí. Esto puede resultar en un ahorro de tiempo significativo. Si sus dependencias principales no cambian, las nuevas versiones de su servicio pueden continuar usando la misma imagen base.

Sin embargo, hay algunos inconvenientes con este enfoque. Crear imágenes puede llevar mucho tiempo. Esto significa que, para los desarrolladores, es posible que desee admitir otras formas de implementar servicios para asegurarse de que no tengan que esperar media hora solo para crear una implementación binaria. En segundo lugar, algunas de las imágenes resultantes pueden ser grandes. Esto podría ser un problema real si está creando sus propias imágenes de VMWare, por ejemplo, ya que mover una imagen de 20 GB en una red no siempre es una actividad sencilla. En breve analizaremos la tecnología de contenedores, y específicamente Docker, que puede evitar algunos de estos inconvenientes.

Históricamente, uno de los desafíos es que la cadena de herramientas requerida para construir tal imagen variaba de una plataforma a otra. Crear una imagen de VMWare es diferente de crear una AMI de AWS, una imagen de Vagrant o una imagen de Rackspace. Es posible que esto no haya sido un problema si tuviera la misma plataforma en todas partes, pero no todas las organizaciones tuvieron esta suerte. E incluso si lo fueran, las herramientas en este espacio a menudo eran difíciles de usar y no funcionaban bien con otras herramientas que podría estar usando para la configuración de la máquina.

Packer es una herramienta diseñada para hacer mucho más fácil la creación de imágenes. Usando los scripts de configuración de su elección (se admiten Chef, Ansible, Puppet y más), nos permite crear imágenes para diferentes plataformas desde la misma configuración. En el momento de escribir este artículo, es compatible con VMWare, AWS, Rackspace Cloud, Digital Ocean y Vagrant, y he visto equipos que lo utilizan con éxito para crear imágenes de Linux y Windows. Esto significa que podría crear una imagen para su implementación en su entorno de producción de AWS y una imagen de Vagrant correspondiente para el desarrollo y las pruebas locales, todo desde la misma configuración.

Imágenes como artefactos

Entonces, podemos crear imágenes de máquinas virtuales que integren dependencias para acelerar la retroalimentación, pero ¿por qué detenerse ahí? Podríamos ir más allá, integrar nuestro servicio en la imagen misma y adoptar el modelo de nuestro artefacto de servicio como una imagen. Ahora, cuando lanzamos nuestra imagen, nuestro servicio está listo para funcionar. Este tiempo de aceleración realmente rápido es la razón por la que Netflix ha adoptado el modelo de hornear sus propios servicios como AMI de AWS.

Al igual que con los paquetes específicos del sistema operativo, estas imágenes de VM se convierten en una buena forma de abstraer las diferencias en las pilas de tecnología utilizadas para crear los servicios. ¿Nos importa si el servicio que se ejecuta en la imagen está escrito en Ruby o Java y usa un archivo gem o JAR? Lo único que nos importa es que funcione. Podemos centrar nuestros esfuerzos, entonces, en automatizar la creación y el despliegue de estas imágenes. Esto también se convierte en una forma realmente ingeniosa de implementar otro concepto de implementación, el *servidor inmutable*.

Servidores inmutables

Al almacenar toda nuestra configuración en el control de código fuente, intentamos asegurarnos de que podemos reproducir automáticamente servicios y, con suerte, entornos completos a voluntad. Pero una vez que ejecutamos nuestro proceso de implementación, ¿qué sucede si aparece alguien, inicia sesión en el cuadro y cambia las cosas independientemente de lo que está en el control de código fuente? Este problema a menudo se denomina *desviación de configuración* : el código en el control de fuente ya no refleja la configuración del host en ejecución.

Para evitar esto, podemos asegurarnos de que nunca se realicen cambios en un servidor en ejecución. En cambio, cualquier cambio, por pequeño que sea, tiene que pasar por una canalización de compilación para crear una nueva máquina. Puede implementar este patrón sin usar implementaciones basadas en imágenes, pero también es una extensión lógica del uso de imágenes como artefactos. Durante la creación de nuestra imagen, por ejemplo, podríamos deshabilitar SSH, ¡asegurándonos de que nadie pudiera iniciar sesión en el cuadro para hacer un cambio!

Las mismas advertencias que discutimos anteriormente sobre el tiempo del ciclo aún se aplican, por supuesto. Y también debemos asegurarnos de que cualquier dato que nos interese que esté almacenado en la caja se almacene en otro lugar. Dejando a un lado estas complejidades, he visto que la adopción de este patrón conduce a implementaciones mucho más sencillas y entornos más fáciles de razonar. Y como ya he dicho, ¡cualquier cosa que podamos hacer para simplificar las cosas debe llevarse a cabo!

Entornos

A medida que nuestro software avanza a través de nuestras etapas de canalización de CD, también se implementará en diferentes tipos de entornos. Si pensamos en el canal de compilación de ejemplo de la [figura 6-4](#), probablemente tengamos que considerar al menos cuatro entornos distintos: un entorno en el que ejecutamos nuestras pruebas lentas, otro para UAT, otro para rendimiento y uno final para producción. Nuestro microservicio debe ser el mismo en todas partes, pero el entorno será diferente. Como mínimo, serán colecciones separadas y distintas de configuración y hosts. Pero a menudo pueden variar mucho más que eso. Por ejemplo, nuestro entorno de producción para nuestro servicio puede consistir en varios hosts con equilibrio de carga distribuidos en dos centros de datos, mientras que nuestro entorno de prueba puede tener todo ejecutándose en un solo host. Estas diferencias en los entornos pueden presentar algunos problemas.

Me mordió esto personalmente hace muchos años. Estábamos implementando un servicio web Java en un contenedor de aplicaciones WebLogic en clúster en producción. Este clúster de WebLogic reprodujo el estado de la sesión entre varios nodos, lo que nos brinda cierto nivel de resiliencia si falla un solo nodo. Sin embargo, las licencias de WebLogic eran costosas, al igual que las máquinas en las que se implementó nuestro software. Esto significó que en nuestro entorno de prueba, nuestro software se implementó en una sola máquina, en una configuración no agrupada.

Esto nos lastimó mucho durante un lanzamiento. Para que WebLogic pueda copiar el estado de la sesión entre nodos, los datos de la sesión deben poder serializarse correctamente. Desafortunadamente, uno de nuestros compromisos rompió esto, por lo que cuando implementamos en producción, la replicación de nuestra sesión falló. Terminamos resolviendo esto presionando mucho para replicar una configuración en clúster en nuestro entorno de prueba.

El servicio que queremos implementar es el mismo en todos estos entornos diferentes, pero cada uno de los entornos tiene un propósito diferente. En mi computadora portátil de desarrollador, quiero implementar rápidamente el servicio, potencialmente contra los colaboradores tachados, para ejecutar pruebas o llevar a cabo alguna validación manual del comportamiento, mientras que cuando implemento en un entorno de producción, es posible que desee implementar varias copias de mi servicio en una carga. -modo equilibrado, tal vez dividido en uno o más centros de datos por razones de durabilidad.

A medida que pasa de su computadora portátil para construir un servidor a un entorno UAT hasta la producción, querrá asegurarse de que sus entornos sean cada vez más productivos para detectar cualquier problema asociado con estas diferencias ambientales antes. Este será un equilibrio constante. A veces, el tiempo y el costo para reproducir entornos de producción pueden ser prohibitivos, por lo que debe hacer concesiones. Además, a veces, el uso de un entorno similar al de producción puede ralentizar los bucles de retroalimentación; esperar a que 25 máquinas instalen su software en AWS puede ser mucho más lento que simplemente implementar su servicio en una instancia local de Vagrant, por ejemplo.

Este equilibrio, entre entornos de producción y comentarios rápidos, no será estático.

Esté atento a los errores que encuentre más adelante y a los tiempos de retroalimentación, y ajuste

este saldo según sea necesario.

Administrar entornos para sistemas monolíticos de un solo artefacto puede ser un desafío, especialmente si no tiene acceso a sistemas que sean fácilmente automatizables. Cuando piensa en múltiples entornos por microservicio, esto puede ser aún más desalentador. Veremos en breve algunas plataformas de implementación diferentes que pueden hacer esto mucho más fácil para nosotros.

Configuración del servicio

Nuestros servicios necesitan alguna configuración. Idealmente, esto debería ser una cantidad pequeña y limitada a aquellas características que cambian de un entorno a otro, como *qué nombre de usuario y contraseña debo usar para conectarme a mi base de datos*. La configuración que cambia de un entorno a otro debe mantenerse al mínimo absoluto. Cuanto más cambie su configuración el comportamiento fundamental del servicio, y cuanto más varíe esa configuración de un entorno a otro, más problemas encontrará solo en ciertos entornos, lo que es extremadamente doloroso.

Entonces, si tenemos alguna configuración para nuestro servicio que cambia de un entorno a otro, ¿cómo debemos manejar esto como parte de nuestro proceso de implementación? Una opción es crear un artefacto por entorno, con la configuración dentro del propio artefacto. Inicialmente, esto parece sensato. La configuración está integrada; simplemente despléguelo y todo debería funcionar bien, ¿verdad? Esto es problemático. Recuerde el concepto de entrega continua. Queremos crear un artefacto que represente nuestra versión candidata y moverlo a través de nuestra tubería, confirmando que es lo suficientemente bueno para entrar en producción. Imaginemos que construyo artefactos Customer-Service-Test y Customer-Service-Prod. Si mi artefacto Customer-Service-Test pasa las pruebas, pero es el artefacto Customer-Service-Prod el que realmente implemento, ¿puedo estar seguro de que he verificado el software que realmente termina en producción?

Hay otros desafíos también. En primer lugar, está el tiempo adicional necesario para construir estos artefactos. Luego, el hecho de que necesita saber en el momento de la compilación qué entornos existen. ¿Y cómo maneja los datos de configuración confidenciales? No quiero información sobre las contraseñas de producción registradas con mi código fuente, pero si es necesario en el momento de la compilación para crear todos esos artefactos, a menudo es difícil de evitar.

Un mejor enfoque es crear un solo artefacto y administrar la configuración por separado. Esto podría ser un archivo de propiedades que existe para cada entorno o diferentes parámetros pasados a un proceso de instalación. Otra opción popular, especialmente cuando se trata de una gran cantidad de microservicios, es usar un sistema dedicado para proporcionar la configuración, que exploraremos más en el [Capítulo 11](#).

Maapeo de servicio a host

Una de las preguntas que surge bastante temprano en la discusión sobre los microservicios es "¿Cuántos servicios por máquina?" Antes de continuar, deberíamos elegir un término mejor que *máquina*, o incluso el *cuadro* más genérico que usé antes. En esta era de virtualización, la asignación entre un solo host que ejecuta un sistema operativo y la infraestructura física subyacente puede variar en gran medida. Por lo tanto, tiendo a hablar de *hosts*, utilizándolos como una unidad genérica de aislamiento, es decir, un sistema operativo en el que puedo instalar y ejecutar mis servicios. Si está implementando directamente en máquinas físicas, entonces un servidor físico se asigna a un *host* (lo que quizás no sea una terminología completamente correcta en este contexto, pero en ausencia de mejores términos puede ser suficiente). Si está utilizando la virtualización, una sola máquina física puede asignarse a múltiples hosts independientes, cada uno de los cuales podría contener uno o más servicios.

Entonces, cuando pensemos en diferentes modelos de implementación, hablaremos de hosts. Entonces, ¿cuántos servicios por host deberíamos tener?

Tengo una opinión definitiva sobre qué modelo es preferible, pero hay una serie de factores a considerar al determinar qué modelo será el adecuado para usted. También es importante comprender que algunas elecciones que hacemos en este sentido limitarán algunas de las opciones de implementación disponibles para nosotros.

Múltiples servicios por host

Tener múltiples servicios por host, como se muestra en la [Figura 6-6](#), es atractivo por varias razones.

Primero, puramente desde el punto de vista de la administración del host, es más simple. En un mundo donde un equipo administra la infraestructura y otro equipo administra el software, la carga de trabajo del equipo de infraestructura a menudo depende de la cantidad de hosts que debe administrar.

Si se empaquetan más servicios en un solo host, la carga de trabajo de administración del host no aumenta a medida que aumenta la cantidad de servicios. El segundo es el costo. Incluso si tiene acceso a una plataforma de virtualización que le permite aprovisionar y cambiar el tamaño de los hosts virtuales, la virtualización puede agregar una sobrecarga que reduce los recursos subyacentes disponibles para sus servicios. En mi opinión, ambos problemas se pueden abordar con nuevas prácticas de trabajo y tecnología, y lo exploraremos en breve.

Este modelo también es familiar para aquellos que implementan en algún tipo de contenedor de aplicaciones. De alguna manera, el uso de un contenedor de aplicaciones es un caso especial del modelo de servicios múltiples por host, por lo que lo analizaremos por separado. Este modelo también puede simplificar la vida del desarrollador. La implementación de varios servicios en un único host en producción es sinónimo de la implementación de varios servicios en una estación de trabajo o portátil de desarrollo local. Si queremos ver un modelo alternativo, queremos encontrar una manera de mantener esto conceptualmente simple para los desarrolladores.

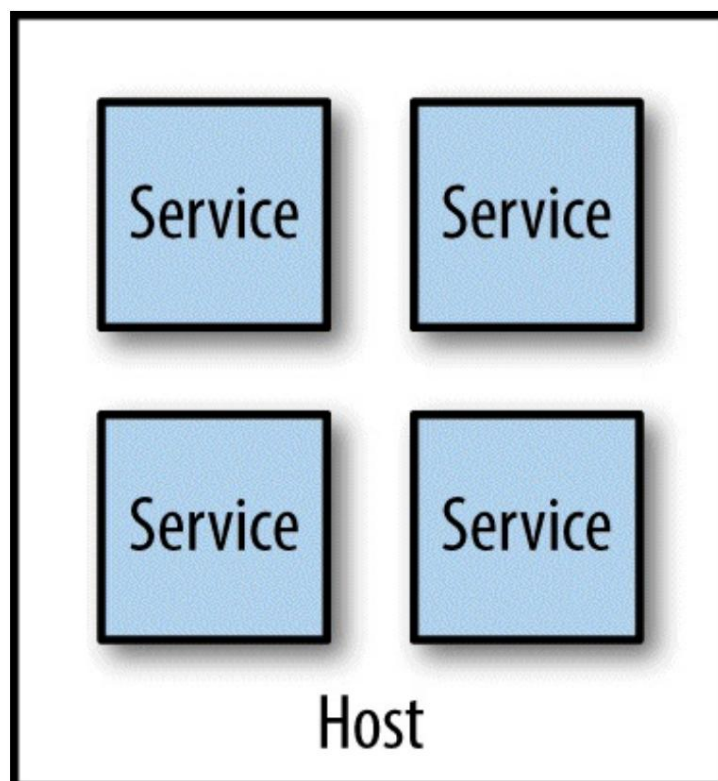


Figura 6-6. Múltiples microservicios por host

Sin embargo, hay algunos desafíos con este modelo. En primer lugar, puede dificultar el seguimiento. Por ejemplo, al rastrear la CPU, ¿debo rastrear la CPU de un servicio independientemente de los demás? ¿O me importa la CPU de la caja en su conjunto? Efectos secundarios

también puede ser difícil de evitar. Si un servicio tiene una carga significativa, puede terminar reduciendo los recursos disponibles para otras partes del sistema. Gilt, al escalar la cantidad de servicios que ejecutaba, se encontró con este problema. Inicialmente, coexistían muchos servicios en una sola caja, pero la carga desigual en uno de los servicios tendría un impacto adverso en todo lo demás que se ejecutaba en ese host. Esto también hace que el análisis de impacto de las fallas del host sea más complejo: dejar fuera de servicio a un solo host puede tener un gran efecto dominó.

La implementación de servicios también puede ser algo más compleja, ya que garantizar que una implementación no afecte a otra genera dolores de cabeza adicionales. Por ejemplo, si uso Puppet para preparar un host, pero cada servicio tiene dependencias diferentes (y potencialmente contradictorias), ¿cómo puedo hacer que funcione? En el peor de los casos, he visto personas vincular múltiples implementaciones de servicios, implementando múltiples servicios diferentes en un solo host en un solo paso, para intentar simplificar la implementación de múltiples servicios en un host. En mi opinión, la pequeña ventaja de mejorar la simplicidad se ve compensada con creces por el hecho de que hemos renunciado a uno de los beneficios clave de los microservicios: luchar por el lanzamiento independiente de nuestro software. Si adopta el modelo de múltiples servicios por host, asegúrese de mantener la idea de que cada servicio debe implementarse de forma independiente.

Este modelo también puede inhibir la autonomía de los equipos. Si se instalan servicios para diferentes equipos en el mismo host, ¿quién puede configurar el host para sus servicios? Con toda probabilidad, esto termina siendo manejado por un equipo centralizado, lo que significa que se necesita más coordinación para implementar los servicios.

Otro problema es que esta opción puede limitar nuestras opciones de artefactos de implementación. Las implementaciones basadas en imágenes están fuera, al igual que los servidores inmutables, a menos que vincule varios servicios diferentes en un solo artefacto, lo que realmente queremos evitar.

El hecho de que tengamos múltiples servicios en un solo host significa que los esfuerzos para orientar la escala al servicio que más lo necesita pueden ser complicados. Del mismo modo, si un microservicio maneja datos y operaciones que son especialmente confidenciales, es posible que deseemos configurar el host subyacente de manera diferente, o tal vez incluso colocar el host en un segmento de red separado.

Tener todo en un host significa que podríamos terminar tratando todos los servicios de la misma manera, incluso si sus necesidades son diferentes.

Como dice mi colega Neal Ford, muchas de nuestras prácticas de trabajo relacionadas con la implementación y la administración de host son un intento de optimizar la escasez de recursos. En el pasado, la única opción si queríamos otro host era comprar o alquilar otra máquina física. Esto a menudo tenía un gran tiempo de espera y resultó en un compromiso financiero a largo plazo. No era raro que los clientes con los que trabajé aprovisionaran nuevos servidores solo cada dos o tres años, y tratar de obtener máquinas adicionales fuera de estos plazos fue difícil. Pero las plataformas informáticas bajo demanda han reducido drásticamente los costos de los recursos informáticos, y las mejoras en la tecnología de virtualización significan que incluso para la infraestructura alojada internamente hay más flexibilidad.

Contenedores de aplicaciones

Si está familiarizado con la implementación de aplicaciones .NET detrás de aplicaciones IIS o Java en un contenedor de servlet, estará familiarizado con el modelo en el que varios servicios o aplicaciones distintos se ubican dentro de un solo contenedor de aplicaciones, que a su vez se ubica en un solo host, como vemos en la [Figura 6-7](#). La idea es que el contenedor de la aplicación en el que viven sus servicios le brinde beneficios en términos de capacidad de administración mejorada, como soporte de agrupación para manejar la agrupación de varias instancias, herramientas de monitoreo y similares.

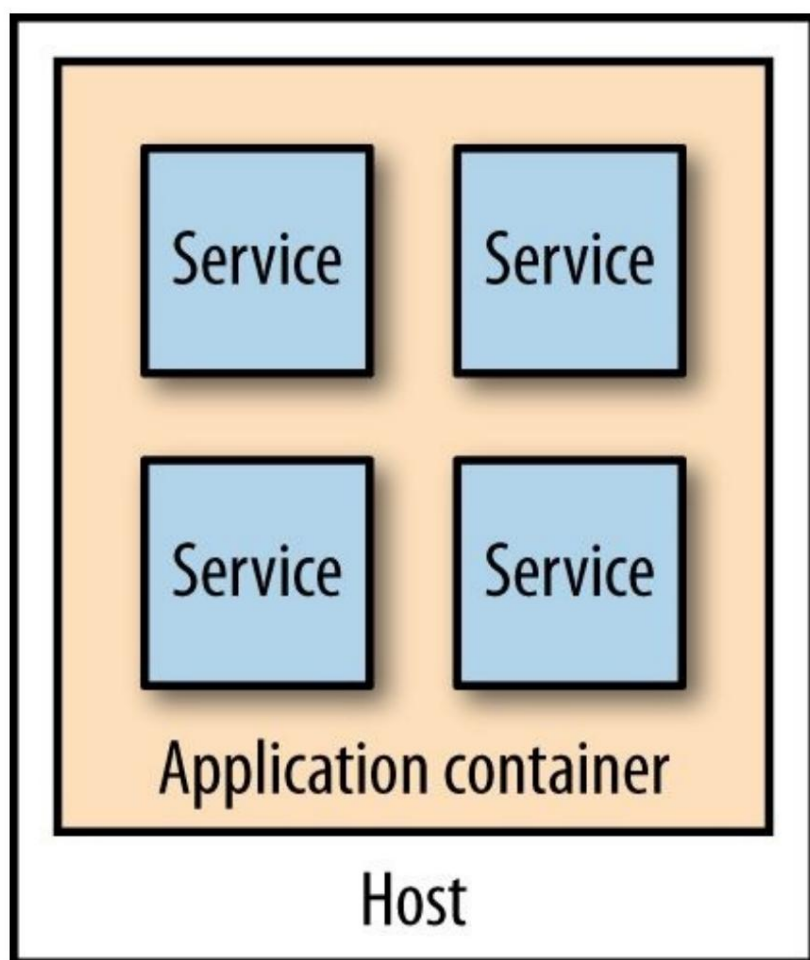


Figura 6-7. Múltiples microservicios por host

Esta configuración también puede generar beneficios en términos de reducir la sobrecarga de los tiempos de ejecución de lenguaje. Considere la posibilidad de ejecutar cinco servicios de Java en un solo contenedor de servlets de Java. Solo tengo la sobrecarga de una sola JVM. Compare esto con la ejecución de cinco JVM independientes en el mismo host cuando se utilizan contenedores integrados. Dicho esto, sigo sintiendo que estos contenedores de aplicaciones tienen suficientes desventajas como para que debas desafiarte a ti mismo para ver si realmente son necesarios.

La primera de las desventajas es que inevitablemente restringen la elección de tecnología. Tienes que comprar en una pila de tecnología. Esto puede limitar no solo las opciones tecnológicas para la implementación del servicio en sí, sino también las opciones que tiene en términos de automatización y administración de sus sistemas. Como discutiremos en breve, una de las formas en que podemos abordar la sobrecarga de administrar múltiples hosts es en torno a la automatización, y así

restringir nuestras opciones para resolver esto bien puede ser doblemente dañino.

También cuestionaría parte del valor de las características del contenedor. Muchos de ellos promocionan la capacidad de administrar clústeres para admitir el estado de sesión compartido en memoria, algo que queremos evitar absolutamente en cualquier caso debido a los desafíos que esto crea al escalar nuestros servicios. Y las capacidades de monitoreo que brindan no serán suficientes cuando consideramos los tipos de monitoreo conjunto que queremos hacer en un mundo de microservicios, como veremos en el **Capítulo 8**. Muchos de ellos también tienen tiempos de activación bastante lentos, impactando en los ciclos de comentarios de los desarrolladores.

También hay otros conjuntos de problemas. Intentar realizar una gestión adecuada del ciclo de vida de las aplicaciones sobre plataformas como la JVM puede ser problemático y más complejo que simplemente reiniciar una JVM. Analizar el uso de recursos y los subprocesos también es mucho más complejo, ya que tiene varias aplicaciones que comparten el mismo proceso. Y recuerde, incluso si obtiene valor de un contenedor de tecnología específica, no son gratuitos. Además del hecho de que muchos de ellos son comerciales y, por lo tanto, tienen una implicación de costos, agregan una sobrecarga de recursos en sí mismos.

En última instancia, este enfoque es nuevamente un intento de optimizar la escasez de recursos que simplemente ya no pueden resistir. Ya sea que decida tener varios servicios por host como modelo de implementación, le sugiero encarecidamente que considere los microservicios implementables autónomos como artefactos. Para .NET, esto es posible con elementos como Nancy, y Java ha sido compatible con este modelo durante años. Por ejemplo, el venerable contenedor integrado Jetty lo convierte en un servidor HTTP autónomo muy liviano, que es el núcleo de la pila Dropwizard. Se sabe que Google usa con bastante gusto contenedores Embedded Jetty para entregar contenido estático directamente, por lo que sabemos que estas cosas pueden operar a escala.

Servicio único por host

Con un modelo de un solo servicio por host que se muestra en la [Figura 6-8](#), evitamos los efectos secundarios de múltiples hosts que viven en un solo host, lo que hace que el monitoreo y la corrección sean mucho más simples. Hemos reducido potencialmente nuestros puntos únicos de falla. Una interrupción en un host debería afectar solo a un único servicio, aunque eso no siempre está claro cuando se utiliza una plataforma virtualizada. Cubriremos el diseño para escalar y fallar con más detalle en el [Capítulo 11](#). También podemos escalar más fácilmente un servicio independientemente de los demás, y lidiar con las preocupaciones de seguridad más fácilmente enfocando nuestra atención solo en el servicio y el host que lo requiere.

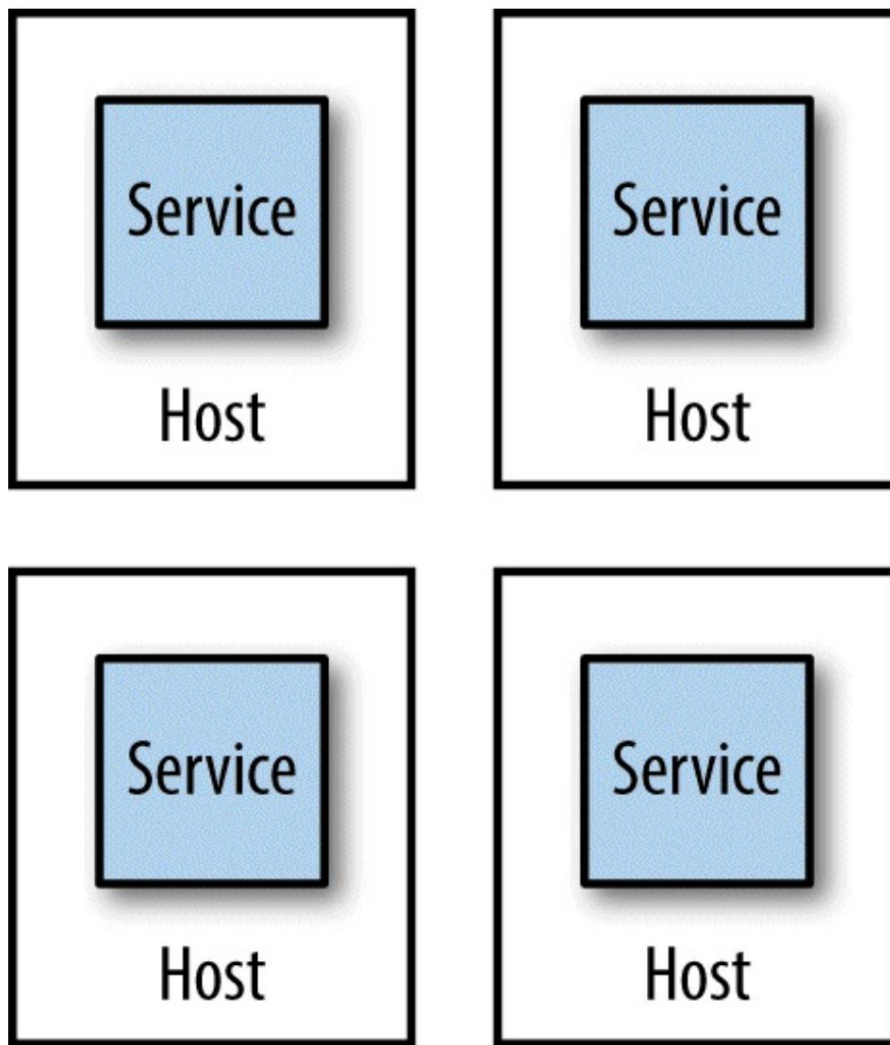


Figura 6-8. Un solo microservicio por host

Igual de importante es que hemos abierto el potencial para usar técnicas de implementación alternativas, como las implementaciones basadas en imágenes o el patrón de servidor inmutable, que discutimos anteriormente.

Hemos agregado mucha complejidad al adoptar una arquitectura de microservicio. Lo último que queremos hacer es buscar más fuentes de complejidad. En mi opinión, si no tiene un PaaS viable disponible, entonces este modelo hace un muy buen trabajo al reducir la complejidad general de un sistema. Tener un modelo de servicio único por host es significativamente más fácil de

razonar y puede ayudar a reducir la complejidad. Si aún no puede adoptar este modelo, no diré que los microservicios no son para usted. Pero le sugiero que busque moverse hacia este modelo con el tiempo como una forma de reducir la complejidad que puede traer una arquitectura de microservicio.

Sin embargo, tener una mayor cantidad de hosts tiene desventajas potenciales. Tenemos más servidores para administrar, y también podría haber una implicación de costos de ejecutar más hosts distintos. A pesar de estos problemas, este sigue siendo el modelo que prefiero para las arquitecturas de microservicios. Y hablaremos sobre algunas cosas que podemos hacer para reducir la sobrecarga de manejar una gran cantidad de hosts en breve.

Plataforma como servicio

Cuando utiliza una plataforma como servicio (PaaS), está trabajando en una abstracción de mayor nivel que en un solo host. La mayoría de estas plataformas se basan en tomar un artefacto específico de la tecnología, como un archivo WAR de Java o una gema de Ruby, y aprovisionarlo y ejecutarlo automáticamente. Algunas de estas plataformas intentarán de manera transparente manejar la ampliación y reducción del sistema por usted, aunque una forma más común (y en mi experiencia menos propensa a errores) le permitirá cierto control sobre cuántos nodos se puede ejecutar su servicio, pero se encarga del resto.

Al momento de escribir este artículo, la mayoría de las mejores y más pulidas soluciones PaaS están alojadas. Heroku viene a la mente como probablemente la clase dorada de PaaS. No solo maneja la ejecución de su servicio, también admite servicios como bases de datos de una manera muy simple. Las soluciones autohospedadas existen en este espacio, aunque son menos maduras que las soluciones alojadas.

Cuando las soluciones PaaS funcionan bien, funcionan muy bien. Sin embargo, cuando no funcionan para usted, a menudo no tiene mucho control en términos de meterse debajo del capó para arreglar las cosas. Esto es parte de la compensación que haces. Diría que, según mi experiencia, cuanto más inteligentes intentan ser las soluciones PaaS, más fallan. He usado más de un PaaS que intenta escalar automáticamente según el uso de la aplicación, pero lo hace mal. Invariablemente, las heurísticas que impulsan esta inteligencia tienden a adaptarse a la aplicación promedio en lugar de a su caso de uso específico. Cuanto menos estándar sea su aplicación, más probable es que no funcione bien con una PaaS.

Como las buenas soluciones PaaS manejan tanto por usted, pueden ser una excelente manera de manejar el aumento de los gastos generales que obtenemos al tener muchas más partes móviles. Dicho esto, todavía no estoy seguro de que tengamos todos los modelos correctos en este espacio todavía, y las opciones limitadas de alojamiento propio significan que este enfoque podría no funcionar para usted. En la próxima década, sin embargo, espero que nos centremos en PaaS para la implementación más que tener que autogestionar hosts e implementaciones de servicios individuales.

Automatización

La respuesta a tantos problemas que hemos planteado hasta ahora se reduce a la automatización. Con un pequeño número de máquinas, es posible gestionar todo manualmente. Solía hacer esto.

Recuerdo ejecutar un pequeño conjunto de máquinas de producción y recopilaba registros, implementaba software y verificaba procesos iniciando sesión manualmente en la caja. Mi productividad parecía estar limitada por la cantidad de ventanas de terminales que podía tener abiertas a la vez: un segundo monitor era un gran paso adelante. Sin embargo, esto se descompone muy rápido.

Uno de los retrocesos contra la configuración de un solo servicio por host es la percepción de que aumentará la cantidad de gastos generales para administrar estos hosts. Esto es ciertamente cierto si está haciendo todo manualmente. ¡Duplica los servidores, duplica el trabajo! Pero si automatizamos el control de nuestros hosts y la implementación de los servicios, entonces no hay razón por la que agregar más hosts aumente nuestra carga de trabajo de manera lineal.

Pero incluso si mantenemos una cantidad pequeña de hosts, seguiremos teniendo muchos servicios.

Eso significa múltiples implementaciones para manejar, servicios para monitorear, registros para recopilar.

La automatización es esencial.

La automatización también es la forma en que podemos asegurarnos de que nuestros desarrolladores sigan siendo productivos.

Darles la capacidad de autoservicio de prestación de servicios individuales o grupos de servicios es clave para facilitar la vida de los desarrolladores. Idealmente, los desarrolladores deberían tener acceso exactamente a la misma cadena de herramientas que se utiliza para la implementación de nuestros servicios de producción para garantizar que podamos detectar problemas desde el principio.

Veremos mucha tecnología en este capítulo que adopta este punto de vista.

La tecnología de selección que permite la automatización es muy importante. Esto comienza con las herramientas utilizadas para administrar hosts. ¿Puedes escribir una línea de código para iniciar una máquina virtual o apagarla? ¿Puede implementar el software que ha escrito automáticamente? ¿Puede implementar cambios en la base de datos sin intervención manual? Adoptar una cultura de automatización es clave si desea mantener bajo control las complejidades de las arquitecturas de microservicios.

Dos estudios de caso sobre el poder de la automatización

Probablemente sea útil darle un par de ejemplos concretos que expliquen el poder de una buena automatización. Uno de nuestros clientes en Australia es RealEstate.com.au (REA). Entre otras cosas, la empresa proporciona listados de bienes raíces para clientes minoristas y comerciales en Australia y en otras partes de la región de Asia-Pacífico. Durante varios años, ha estado moviendo su plataforma hacia un diseño de microservicios distribuidos. Cuando comenzó este viaje, tuvo que dedicar mucho tiempo a obtener las herramientas adecuadas para los servicios, lo que facilitó a los desarrolladores el aprovisionamiento de máquinas, la implementación de su código o su supervisión. Esto provocó una carga inicial de trabajo para comenzar.

En los primeros tres meses de este ejercicio, REA pudo poner en producción solo dos nuevos microservicios, con el equipo de desarrollo asumiendo la responsabilidad total de la construcción, implementación y soporte de los servicios. En los siguientes tres meses, entre 10 y 15 servicios se pusieron en marcha de manera similar. Al final del período de 18 meses, REA tenía entre 60 y 70 servicios.

Este tipo de patrón también se ve confirmado por las experiencias de **Gilt**, un minorista de moda en línea que comenzó en 2007. La aplicación monolítica Rails de Gilt comenzaba a ser difícil de escalar y la empresa decidió en 2009 comenzar a descomponer el sistema en microservicios. Una vez más, la automatización, especialmente las herramientas para ayudar a los desarrolladores, se mencionó como una razón clave para impulsar la explosión de Gilt en el uso de microservicios. Un año después, Gilt tenía alrededor de 10 microservicios activos; para 2012, más de 100; y en 2014, más de 450 microservicios según el recuento de Gilt; en otras palabras, alrededor de tres servicios por cada desarrollador en Gilt.

De lo físico a lo virtual

Una de las herramientas clave que tenemos a nuestra disposición para administrar una gran cantidad de hosts es encontrar formas de fragmentar las máquinas físicas existentes en partes más pequeñas. La virtualización tradicional como VMWare o la utilizada por AWS ha producido enormes beneficios en la reducción de la sobrecarga de administración de host. Sin embargo, ha habido algunos avances nuevos en este espacio que vale la pena explorar, ya que pueden abrir posibilidades aún más interesantes para manejar nuestra arquitectura de microservicios.

Virtualización tradicional

¿Por qué es caro tener muchos hosts? Bueno, si necesita un servidor físico por host, la respuesta es bastante obvia. Si este es el mundo en el que está operando, entonces el modelo de servicio múltiple por host probablemente sea adecuado para usted, aunque no se sorprenda si esto se convierte en una restricción cada vez más desafiante. Sin embargo, sospecho que la mayoría de ustedes está utilizando algún tipo de virtualización. La virtualización nos permite dividir un servidor físico en hosts separados, cada uno de los cuales puede ejecutar cosas diferentes. Entonces, si queremos un servicio por host, ¿no podemos dividir nuestra infraestructura física en partes cada vez más pequeñas?

Bueno, para algunas personas, puedes. Sin embargo, dividir la máquina en máquinas virtuales cada vez mayores no es gratis. Piense en nuestra máquina física como un cajón de calcetines. Si ponemos muchos separadores de madera en nuestro cajón, ¿podemos guardar más calcetines o menos? La respuesta es menos: ¡los propios separadores también ocupan espacio! Nuestro cajón puede ser más fácil de manejar y organizar, y tal vez podríamos decidir colocar camisetas en uno de los espacios ahora en lugar de solo calcetines, pero más divisores significan menos espacio general.

En el mundo de la virtualización, tenemos una sobrecarga similar a la de nuestros separadores de cajones. Para comprender de dónde proviene esta sobrecarga, veamos cómo se realiza la mayor parte de la virtualización.

La figura 6-9 muestra una comparación de dos tipos de virtualización. A la izquierda, vemos las diversas capas involucradas en lo que se denomina *virtualización de tipo 2*, que es el tipo implementado por AWS, VMWare, VSphere, Xen y KVM. (La virtualización de tipo 1 se refiere a la tecnología en la que las máquinas virtuales se ejecutan directamente en el hardware, no sobre otro sistema operativo). En nuestra infraestructura física tenemos un sistema operativo host. En este sistema operativo ejecutamos algo llamado *hipervisor*, que tiene dos funciones clave. Primero, asigna recursos como la CPU y la memoria del host virtual al host físico. En segundo lugar, actúa como una capa de control, permitiéndonos manipular las propias máquinas virtuales.

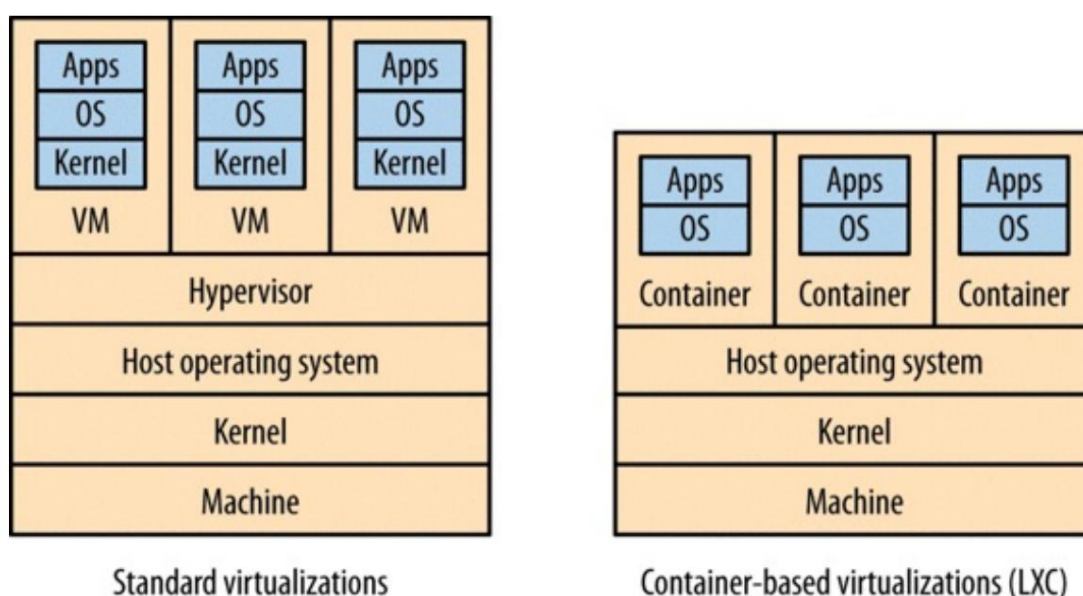


Figura 6-9. Una comparación de la virtualización estándar de tipo 2 y los contenedores ligeros

Dentro de las máquinas virtuales, obtenemos lo que parecen hosts completamente diferentes. Pueden ejecutar su propia

sistemas operativos, con sus propios kernels. Pueden considerarse máquinas selladas casi herméticamente, mantenidas aisladas del host físico subyacente y de las otras máquinas virtuales por el hipervisor.

El problema es que el hipervisor aquí necesita reservar recursos para hacer su trabajo. Esto quita CPU, E/S y memoria que podrían usarse en otros lugares. Cuantos más hosts gestione el hipervisor, más recursos necesitará. En cierto punto, esta sobrecarga se convierte en una restricción para dividir aún más su infraestructura física. En la práctica, esto significa que a menudo hay rendimientos decrecientes al dividir una caja física en partes cada vez más pequeñas, ya que proporcionalmente más y más recursos se destinan a la sobrecarga del hipervisor.

Vagabundo

Vagrant es una plataforma de implementación muy útil, que normalmente se usa para desarrollo y prueba en lugar de producción. Vagrant le proporciona una nube virtual en su computadora portátil.

Debajo, utiliza un sistema de virtualización estándar (normalmente VirtualBox, aunque puede utilizar otras plataformas). Le permite definir un conjunto de máquinas virtuales en un archivo de texto, junto con cómo se conectan en red las máquinas virtuales y en qué imágenes se deben basar las máquinas virtuales. Este archivo de texto se puede registrar y compartir entre los miembros del equipo.

Esto le facilita la creación de entornos de producción en su máquina local.

Puede activar varias VM a la vez, cerrar las individuales para probar los modos de falla y hacer que las VM se asignen a directorios locales para que pueda realizar cambios y verlos reflejados de inmediato. Incluso para los equipos que usan plataformas en la nube bajo demanda como AWS, el cambio más rápido de usar Vagrant puede ser una gran ayuda para los equipos de desarrollo.

Sin embargo, una de las desventajas es que ejecutar muchas máquinas virtuales puede afectar a la máquina de desarrollo promedio. Si tenemos un servicio para una máquina virtual, es posible que no pueda activar todo el sistema en su máquina local. Esto puede resultar en la necesidad de eliminar algunas dependencias para que las cosas sean manejables, que es una cosa más que tendrá que manejar para asegurarse de que la experiencia de desarrollo y prueba sea buena.

Contenedores Linux

Para los usuarios de Linux, existe una alternativa a la virtualización. En lugar de tener un hipervisor para segmentar y controlar hosts virtuales separados, los contenedores de Linux crean un espacio de proceso separado en el que viven otros procesos.

En Linux, los procesos son ejecutados por un usuario determinado y tienen ciertas capacidades según cómo se establezcan los permisos. Los procesos pueden generar otros procesos. Por ejemplo, si lanzo un proceso en una terminal, ese proceso secundario generalmente se considera un proceso secundario del proceso terminal. El trabajo del kernel de Linux es mantener este árbol de procesos.

Los contenedores de Linux amplían esta idea. Cada contenedor es efectivamente un subárbol del árbol de procesos del sistema general. Estos contenedores pueden tener recursos físicos asignados, algo que el kernel maneja por nosotros. Este enfoque general ha existido de muchas formas, como Solaris Zones y OpenVZ, pero es LXC el que se ha vuelto más popular.

LXC ahora está disponible de forma inmediata en cualquier kernel de Linux moderno.

Si observamos un diagrama de pila para un host que ejecuta LXC en la [Figura 6-9](#), vemos algunas diferencias. Primero, no necesitamos un hipervisor. En segundo lugar, aunque cada contenedor puede ejecutar su propia distribución de sistema operativo, tiene que compartir el mismo kernel (porque el kernel es donde vive el árbol de procesos). Esto significa que nuestro sistema operativo host podría ejecutar Ubuntu y nuestros contenedores CentOS, siempre que ambos puedan compartir el mismo kernel.

No solo nos beneficiamos de los recursos ahorrados al no necesitar un hipervisor. También ganamos en términos de retroalimentación. Los contenedores de Linux son *mucho* más rápidos de aprovisionar que las máquinas virtuales completas. No es raro que una máquina virtual tarde varios minutos en iniciarse, pero con los contenedores de Linux, el inicio puede demorar unos segundos. También tiene un control más detallado sobre los propios contenedores en términos de asignación de recursos, lo que hace que sea mucho más fácil ajustar la configuración para aprovechar al máximo el hardware subyacente.

Debido a la naturaleza más liviana de los contenedores, podemos tener muchos más ejecutándose en el mismo hardware de lo que sería posible con las máquinas virtuales. Al implementar un servicio por contenedor, como en la [Figura 6-10](#), obtenemos un grado de aislamiento de otros contenedores (aunque esto no es perfecto) y podemos hacerlo mucho más rentable de lo que sería posible si quisiéramos ejecutar cada uno. servicio en su propia máquina virtual.

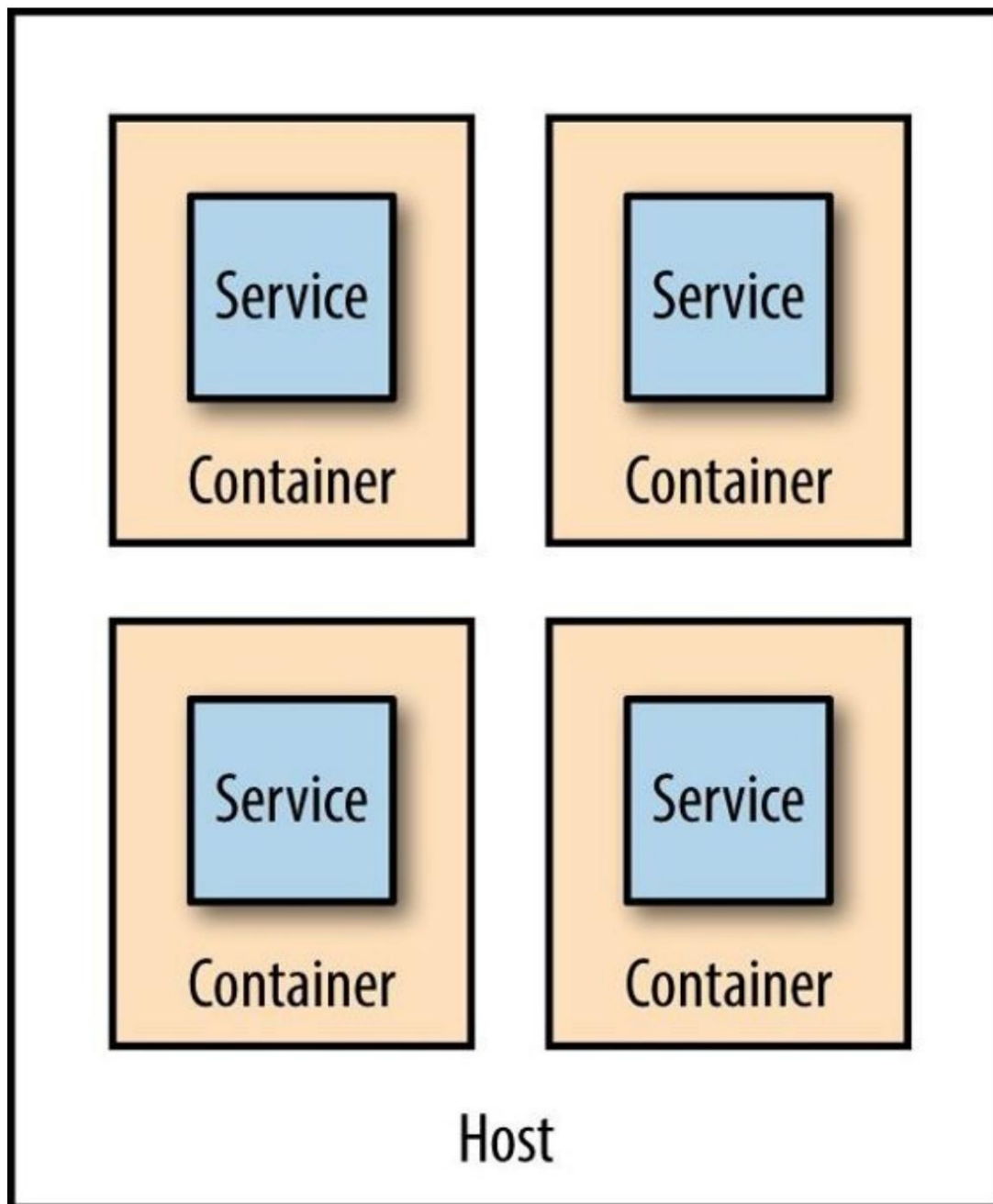


Figura 6-10. Ejecutar servicios en contenedores separados

Los contenedores también se pueden usar bien con la virtualización completa. He visto más de un proyecto aprovisionar una gran instancia de AWS EC2 y ejecutar contenedores LXC en ella para obtener lo mejor de ambos mundos: una plataforma informática efímera bajo demanda en forma de EC2, junto con contenedores altamente flexibles y rápidos que se ejecutan en arriba de eso

Sin embargo, los contenedores de Linux no están exentos de problemas. Imagine que tengo muchos microservicios ejecutándose en sus propios contenedores en un host. ¿Cómo los ve el mundo exterior? Necesita alguna forma de enrutar el mundo exterior a través de los contenedores subyacentes, algo que muchos de los hipervisores hacen por usted con la virtualización normal. He visto a muchas personas dedicar cantidades excesivas de tiempo a configurar el reenvío de puertos usando IPTables para exponer contenedores directamente. Otro punto a tener en cuenta es que estos contenedores no pueden considerarse completamente sellados entre sí. Hay muchas formas documentadas y conocidas en las que un proceso de un contenedor puede explotar e interactuar con otros contenedores o el host subyacente. Algunos de estos problemas son por diseño y algunos son errores que son

siendo abordado, pero de cualquier manera, si no confía en el código que está ejecutando, no espere que pueda ejecutarlo en un contenedor y estar seguro. Si necesita ese tipo de aislamiento, deberá considerar usar máquinas virtuales en su lugar.

Estibador

Docker es una plataforma construida sobre contenedores ligeros. Maneja gran parte del trabajo relacionado con el manejo de contenedores por usted. En Docker, crea e implementa *aplicaciones*, que son sinónimo de imágenes en el mundo de las máquinas virtuales, aunque para una plataforma basada en contenedores.

Docker administra el aprovisionamiento de contenedores, maneja algunos de los problemas de red por usted e incluso proporciona su propio concepto de registro que le permite almacenar y crear versiones de las aplicaciones de Docker.

La abstracción de la aplicación Docker es útil para nosotros, porque al igual que con las imágenes de VM, la tecnología subyacente utilizada para implementar el servicio está oculta para nosotros. Tenemos nuestras compilaciones para nuestros servicios, creamos aplicaciones de Docker, las almacenamos en el registro de Docker y listo.

Docker también puede aliviar algunas de las desventajas de ejecutar muchos servicios localmente para fines de desarrollo y prueba. En lugar de usar Vagrant para alojar varias máquinas virtuales independientes, cada una con su propio servicio, podemos alojar una única máquina virtual en Vagrant que ejecuta una instancia de Docker. Luego usamos Vagrant para configurar y dismantelar la propia plataforma Docker, y usamos Docker para el aprovisionamiento rápido de servicios individuales.

Se están desarrollando varias tecnologías diferentes para aprovechar Docker.

CoreOS es un sistema operativo muy interesante diseñado con Docker en mente. Es un sistema operativo Linux simplificado que proporciona solo los servicios esenciales para permitir que Docker se ejecute. Esto significa que consume menos recursos que otros sistemas operativos, lo que permite dedicar aún más recursos de la máquina subyacente a nuestros contenedores. En lugar de usar un administrador de paquetes como *debs* o *RPM*, todo el software se instala como aplicaciones Docker independientes, cada una ejecutándose en su propio contenedor.

Docker en sí mismo no resuelve todos los problemas por nosotros. Piense en ello como un PaaS simple que funciona en una sola máquina. Si desea herramientas que lo ayuden a administrar servicios en varias instancias de Docker en varias máquinas, deberá buscar otro software que agregue estas capacidades. Existe una necesidad clave de una capa de programación que le permita solicitar un contenedor y luego encuentre un contenedor Docker que pueda ejecutarlo por usted. En este espacio, la tecnología de clúster de CoreOS y Kubernetes de código abierto recientemente de Google pueden ayudar, y parece que cada mes hay un nuevo participante en este espacio. *Deis* es otra herramienta interesante basada en Docker, que intenta proporcionar un PaaS similar a Heroku sobre Docker.

Hablé anteriormente sobre las soluciones PaaS. Mi problema con ellos siempre ha sido que a menudo se equivocan en el nivel de abstracción y que las soluciones autohospedadas van muy por detrás de las soluciones alojadas como Heroku. Docker obtiene mucho más de este derecho, y la explosión de interés en este espacio significa que sospecho que se convertirá en una plataforma mucho más viable para todo tipo de implementaciones en los próximos años para todo tipo de casos de uso diferentes. En muchos sentidos, Docker con una capa de programación adecuada se encuentra entre las soluciones IaaS y PaaS: el término *contenedores como servicio (CaaS)* ya se usa para

describirlo.

Docker está siendo utilizado en producción por varias empresas. Brinda muchos de los beneficios de los contenedores livianos en términos de eficiencia y velocidad de aprovisionamiento, junto con las herramientas para evitar muchas de las desventajas. Si está interesado en buscar plataformas de implementación alternativas, le sugiero encarecidamente que eche un vistazo a Docker.

Una interfaz de implementación

Independientemente de la plataforma o los artefactos subyacentes que utilice, es fundamental contar con una interfaz uniforme para implementar un servicio determinado. Querremos desencadenar la implementación de un microservicio a pedido en una variedad de situaciones diferentes, desde implementaciones locales para desarrollo y prueba hasta implementaciones de producción. También querremos mantener nuestros mecanismos de implementación lo más similares posible desde el desarrollo hasta la producción, ya que lo último que queremos es encontrarnos con problemas en producción porque la implementación utiliza un proceso completamente diferente.

Después de muchos años de trabajar en este espacio, estoy convencido de que la forma más sensata de activar cualquier implementación es a través de una única llamada de línea de comandos parametrizable. Esto puede ser activado por secuencias de comandos, iniciado por su herramienta de CI o escrito a mano. He creado scripts de contenedor en una variedad de pilas de tecnología para que esto funcione, desde lotes de Windows, bash, scripts de Python Fabric y más, pero todas las líneas de comando comparten el mismo formato básico.

Necesitamos saber qué estamos implementando, por lo que debemos proporcionar el nombre de una entidad conocida o, en nuestro caso, un microservicio. También necesitamos saber qué versión de la entidad queremos. La respuesta a *qué versión* tiende a ser una de tres posibilidades. Cuando trabaje localmente, será la versión que esté en su máquina local. Al realizar la prueba, querrá la última versión *ecológica*, que podría ser el artefacto bendecido más reciente en nuestro repositorio de artefactos. O al probar/diagnosticar problemas, es posible que deseemos implementar una compilación exacta.

Lo tercero y último que necesitaremos saber es en qué entorno queremos implementar el microservicio. Como discutimos anteriormente, la topología de nuestro microservicio puede diferir de un entorno a otro, pero eso debería estar oculto para nosotros aquí.

Entonces, imagine que creamos un script de implementación simple que toma estos tres parámetros. Digamos que estamos desarrollando localmente y queremos implementar nuestro servicio de catálogo en nuestro entorno local. Podría escribir:

```
$ implementar artefacto=entorno de catálogo=versión local=local
```

Una vez que me registré, nuestro servicio de compilación de CI recoge el cambio y crea un nuevo artefacto de compilación, dándole el número de compilación b456. Como es estándar en la mayoría de las herramientas de CI, este valor se pasa a lo largo de la canalización. Cuando se active nuestra etapa de prueba, la etapa de CI se ejecutará:

```
$ implementar artefacto=entorno de catálogo=versión ci=b456
```

Mientras tanto, nuestro control de calidad quiere llevar la última versión del servicio de catálogo a un entorno de prueba integrado para realizar algunas pruebas exploratorias y ayudar con una demostración. Ese equipo corre:

```
$ implementar artefacto=entorno de catálogo=versión integrada_qa=más reciente
```

La herramienta que más he usado para esto es Fabric, una biblioteca de Python diseñada para asignar llamadas de línea de comando a funciones, junto con un buen soporte para manejar tareas como SSH en máquinas remotas. Combínelo con una biblioteca de clientes de AWS como Boto y tendrá todo lo que necesita para automatizar por completo entornos de AWS muy grandes. Para Ruby, Capistrano es similar en algunos aspectos a Fabric, y en Windows podría recorrer un largo camino con PowerShell.

Definición de entorno

Claramente, para que esto funcione, debemos tener alguna forma de definir cómo se ven nuestros entornos y cómo se ve nuestro servicio en un entorno determinado. Puede pensar en una definición de entorno como una asignación de un microservicio a recursos informáticos, de red y de almacenamiento. He hecho esto con archivos YAML antes y usé mis scripts para obtener estos datos. El [ejemplo 6-1](#) es una versión simplificada de un trabajo que hice hace un par de años para un proyecto que usaba AWS.

Ejemplo 6-1. Una definición de entorno de ejemplo

```
desarrollo:
  nodos: -
    ami_id: ami-e1e1234
    tamaño: t1.micro ❶
    credentials_name: eu-west-ssh ❷
    servicios: [catalog-service] región: eu-west-1

producción:
  nodos: -
    ami_id: ami-e1e1234
    tamaño: m3.xlarge ❶
    credentials_name: prod-credentials servicios: ❷
    [servicio de catálogo] número: 5 ❸
```

❶

Variamos el tamaño de las instancias que usamos para ser más rentables. ¡No necesita una caja de 16 núcleos con 64 GB de RAM para las pruebas exploratorias!

❷

Poder especificar diferentes credenciales para diferentes entornos es clave.

Las credenciales para entornos confidenciales se almacenaron en diferentes repositorios de código fuente a los que solo tendrían acceso personas seleccionadas.

❸

Decidimos que, de forma predeterminada, si un servicio tenía más de un nodo configurado, crearíamos automáticamente un balanceador de carga para él.

He eliminado algunos detalles en aras de la brevedad.

La información del servicio de catálogo se almacenó en otro lugar. No difirió de un entorno a otro, como puede ver en el [Ejemplo 6-2](#).

Ejemplo 6-2. Una definición de entorno de ejemplo

```
servicio de catálogo:
  puppet_manifest: catalog.pp ❶
  conectividad: - protocolo: tcp puertos:
    [8080, 8081] permitido: [MUNDO]
```

❶

Este era el nombre del archivo de Puppet que se ejecutaría: usamos Puppet solo en esta situación, pero teóricamente podría haber admitido sistemas de configuración alternativos.

Obviamente, mucho del comportamiento aquí estaba basado en convenciones. Por ejemplo, decidimos normalizar qué puertos usaban los servicios dondequiera que se ejecutaran y configuramos automáticamente los balanceadores de carga si un servicio tenía más de una instancia (algo que los ELB de AWS facilitan bastante).

Construir un sistema como este requirió una cantidad significativa de trabajo. El esfuerzo a menudo se carga al principio, pero puede ser esencial para administrar la complejidad de la implementación que tiene. Espero que en el futuro no tengas que hacer esto tú mismo. Terraform es una herramienta muy nueva de Hashicorp, que funciona en este espacio. En general, evitaría mencionar una herramienta tan nueva en un libro que trata más sobre ideas que sobre tecnología, pero está intentando crear una herramienta de código abierto en este sentido. Todavía es pronto, pero sus capacidades ya parecen realmente interesantes. Con la capacidad de orientar implementaciones en varias plataformas diferentes, en el futuro podría ser la herramienta ideal para el trabajo.

Resumen

Hemos cubierto mucho terreno aquí, por lo que se requiere una recapitulación. En primer lugar, concéntrese en mantener la capacidad de lanzar un servicio independientemente de otro y asegúrese de que la tecnología que seleccione lo admita. Prefiero en gran medida tener un solo repositorio por microservicio, pero estoy más firme aún en que necesita una compilación de CI por microservicio si desea implementarlos por separado.

A continuación, si es posible, pase a un servicio único por host/contenedor. Busque tecnologías alternativas como LXC o Docker para que la gestión de las piezas móviles sea más económica y sencilla, pero comprenda que independientemente de la tecnología que adopte, una cultura de automatización es clave para gestionarlo todo. Automatiza todo, y si la tecnología que tienes no te lo permite, ¡obtén nueva tecnología! Poder utilizar una plataforma como AWS le brindará enormes beneficios en lo que respecta a la automatización.

Asegúrese de comprender el impacto que sus opciones de implementación tienen en los desarrolladores y asegúrese de que ellos también sientan el amor. La creación de herramientas que le permitan implementar por sí mismo cualquier servicio dado en una cantidad de entornos diferentes es realmente importante y ayudará a los desarrolladores, evaluadores y personas de operaciones por igual.

Finalmente, si desea profundizar en este tema, le recomiendo que lea *Continuous Delivery* (Addison-Wesley) de Jez Humble y David Farley, que profundiza mucho más en temas como el diseño de canalización y la gestión de artefactos.

En el próximo capítulo, profundizaremos en un tema que tocamos brevemente aquí. Es decir, ¿cómo probamos nuestros microservicios para asegurarnos de que realmente funcionan?

Capítulo 7. Pruebas

El mundo de las pruebas automatizadas ha avanzado significativamente desde que comencé a escribir código, y cada mes parece haber alguna herramienta o técnica nueva para hacerlo aún mejor. Pero quedan desafíos en cuanto a cómo probar de manera efectiva y eficiente nuestra funcionalidad cuando se extiende por un sistema distribuido. Este capítulo desglosa los problemas asociados con las pruebas de sistemas más detallados y presenta algunas soluciones para ayudarlo a asegurarse de que puede lanzar su nueva funcionalidad con confianza.

Las pruebas cubren mucho terreno. Incluso cuando *solo* estamos hablando de pruebas automatizadas, hay un gran número a considerar. Con los microservicios, hemos agregado otro nivel de complejidad. Comprender los diferentes tipos de pruebas que podemos ejecutar es importante para ayudarnos a equilibrar las fuerzas a veces opuestas de hacer que nuestro software entre en producción lo más rápido posible versus asegurarse de que nuestro software tenga la calidad suficiente.

Tipos de Pruebas

Como consultor, me gusta sacar el cuadrante extraño como una forma de categorizar el mundo, y estaba empezando a preocuparme de que este libro no tuviera uno. Afortunadamente, a Brian Marick se le ocurrió un fantástico sistema de categorización para las pruebas que encaja perfectamente. La Figura 7-1 muestra una variación del cuadrante de Marick del libro *Agile Testing* (Addison Wesley) de Lisa Crispin y Janet Gregory que ayuda a clasificar los diferentes tipos de pruebas.

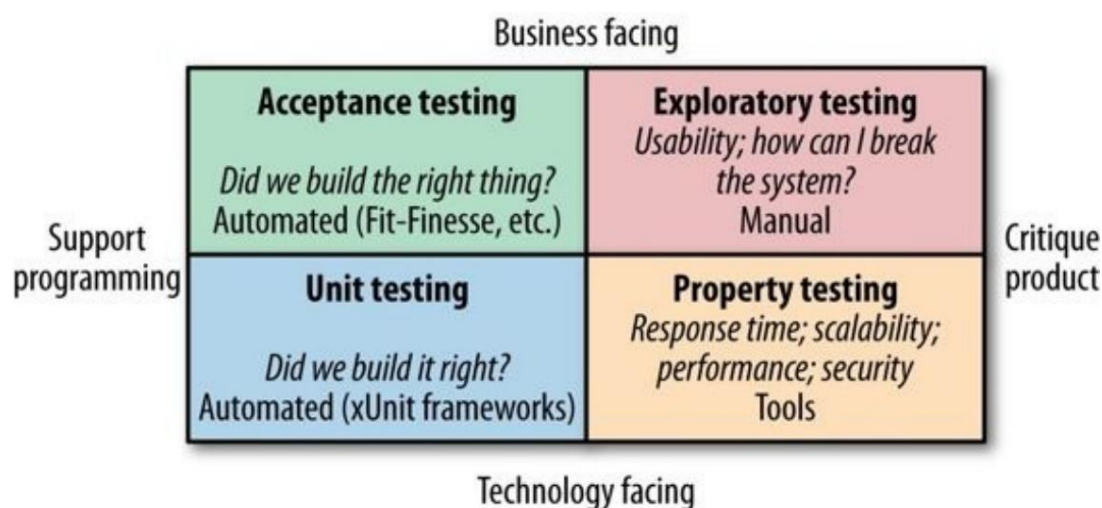


Figura 7-1. Cuadrante de prueba de Brian Marick. Crispin, Lisa; Gregory, Janet, *Agile Testing: A Practical Guide for Testers and Agile Teams*, 1.^a edición, © 2009. Adaptado con autorización de Pearson Education, Inc., Upper Saddle River, NJ.

En la parte inferior, tenemos pruebas *orientadas a la tecnología*, es decir, pruebas que ayudan a los desarrolladores a crear el sistema en primer lugar. Las pruebas de rendimiento y las pruebas *unitarias* de alcance pequeño entran en esta categoría, todas típicamente automatizadas. Esto se compara con la mitad superior del cuadrante, donde las pruebas ayudan a las partes interesadas no técnicas a comprender cómo funciona su sistema. Estas pueden ser pruebas de extremo a extremo de gran alcance, como se muestra en el cuadro de prueba de aceptación de la parte superior izquierda, o pruebas manuales como lo tipifican las pruebas de usuario realizadas en un sistema UAT, como se muestra en el cuadro de prueba exploratoria.

Cada tipo de prueba que se muestra en este cuadrante tiene un lugar. Exactamente cuánto de cada prueba desea hacer dependerá de la naturaleza de su sistema, pero el punto clave que debe comprender es que tiene múltiples opciones en términos de cómo probar su sistema. Recientemente, la tendencia se ha alejado de cualquier prueba manual a gran escala, a favor de la automatización tanto como sea posible, y ciertamente estoy de acuerdo con este enfoque. Si actualmente realiza una gran cantidad de pruebas manuales, le sugiero que las aborde antes de avanzar demasiado en el camino de los microservicios, ya que no obtendrá muchos de sus beneficios si no puede validar su software de manera rápida y eficiente.

A los efectos de este capítulo, ignoraremos las pruebas manuales. Si bien este tipo de prueba puede ser muy útil y sin duda tiene un papel que desempeñar, las diferencias con la prueba de una arquitectura de microservicio se manifiestan principalmente en el contexto de varios tipos de pruebas automatizadas, por lo que concentraremos nuestro tiempo en eso.

Pero cuando se trata de pruebas automatizadas, ¿cuántas de cada prueba queremos? Otro modelo será muy útil para ayudarnos a responder esta pregunta y comprender cuáles podrían ser las diferentes compensaciones.

Alcance de prueba

En su libro *Succeeding with Agile* (Addison-Wesley), Mike Cohn describe un modelo llamado Test Pyramid para ayudar a explicar qué tipos de pruebas automatizadas necesita. La pirámide nos ayuda a pensar en los alcances que deben cubrir las pruebas, pero también en las proporciones de los diferentes tipos de pruebas a las que debemos aspirar. El modelo original de Cohn dividió las pruebas automatizadas en Unidad, Servicio y UI, que puede ver en la [Figura 7-2](#).

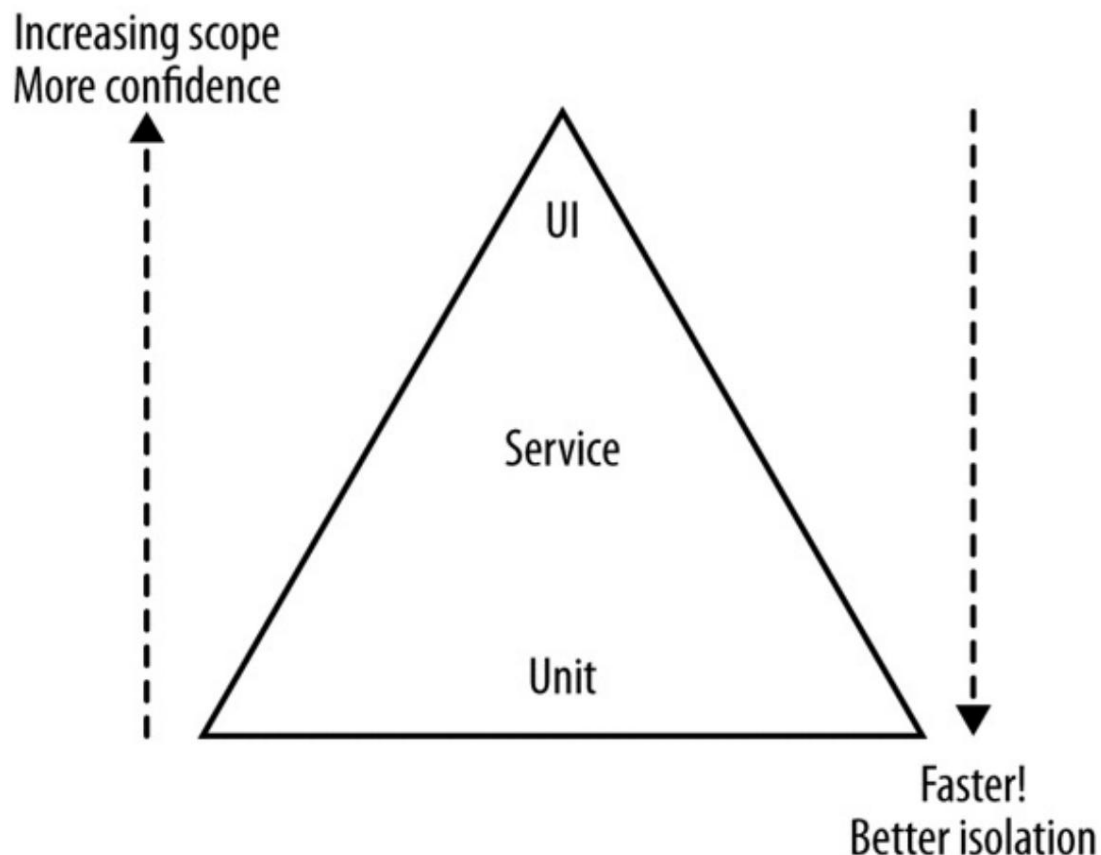


Figura 7-2. Pirámide de prueba de Mike Cohn. Cohn, Mike, Succeeding with Agile: Software Development Using Scrum, 1.ª edición, © 2010. Adaptado con autorización de Pearson Education, Inc., Upper Saddle River, NJ.

El problema con este modelo es que todos estos términos significan cosas diferentes para diferentes personas.

El "servicio" está especialmente sobrecargado, y existen muchas definiciones de una prueba unitaria. ¿Es una prueba una prueba unitaria si solo pruebo una línea de código? Yo diría que sí. ¿Sigue siendo una prueba unitaria si pruebo varias funciones o clases? Yo diría que no, ¡pero muchos no estarían de acuerdo! Tiendo a quedarme con los nombres de Unidad y Servicio a pesar de su ambigüedad, pero prefiero llamar a las pruebas de *interfaz* de usuario pruebas *de extremo a extremo*, lo que haremos a partir de ahora.

Dada la confusión, vale la pena ver qué significan estas diferentes capas.

Veamos un ejemplo trabajado. En la [Figura 7-3](#), tenemos nuestra aplicación de soporte técnico y nuestro sitio web principal, los cuales interactúan con nuestro servicio al cliente para recuperar, revisar y editar los detalles del cliente. Nuestro servicio de atención al cliente, a su vez, está hablando con nuestro banco de puntos de fidelidad, donde nuestros clientes acumulan puntos comprando CD de Justin Bieber. Probablemente. Esto es obviamente una porción de nuestro sistema general de tiendas de música, pero es una porción lo suficientemente buena como para que podamos

sumérase en algunos escenarios diferentes que tal vez queramos probar.

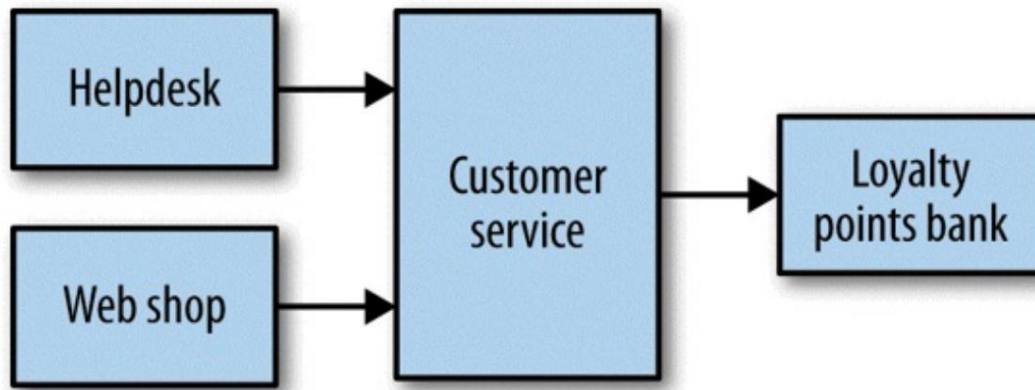


Figura 7-3. Parte de nuestra tienda de música bajo prueba

Pruebas unitarias

Estas son pruebas que normalmente prueban una sola función o llamada de método. Las pruebas generadas como un efecto secundario del *diseño basado en pruebas* (TDD) entrarán en esta categoría, al igual que los tipos de pruebas generadas por técnicas como las pruebas basadas en propiedades. No estamos lanzando servicios aquí y estamos limitando el uso de archivos externos o conexiones de red. En general, desea una gran cantidad de este tipo de pruebas. Si se hacen bien, son muy, muy rápidos, y en hardware moderno podrías esperar ejecutar muchos miles de estos en menos de un minuto.

Estas son pruebas que nos ayudan a los desarrolladores y, por lo tanto, estarían *orientados a la tecnología*, no *al negocio*, en la terminología de Marick. También es donde esperamos atrapar la mayoría de nuestros errores.

Entonces, en nuestro ejemplo, cuando pensamos en el servicio al cliente, las pruebas unitarias cubrirían pequeñas partes del código de forma aislada, como se muestra en la [figura 7-4](#).

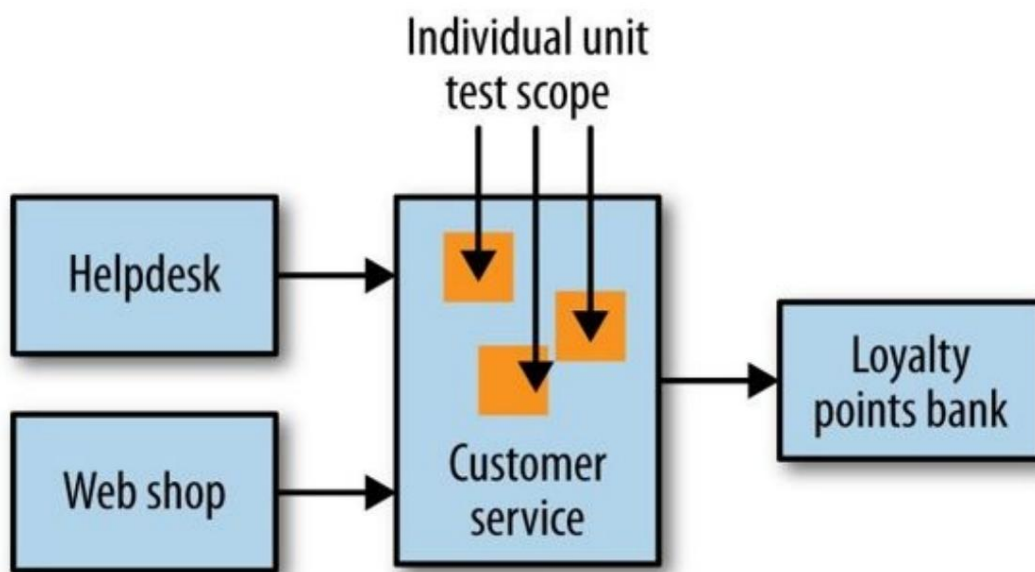


Figura 7-4. Alcance de las pruebas unitarias en nuestro sistema de ejemplo

El objetivo principal de estas pruebas es darnos una respuesta muy rápida sobre si nuestra funcionalidad es buena. Las pruebas pueden ser importantes para respaldar la refactorización del código, permitiéndonos reestructurar nuestro código a medida que avanzamos, sabiendo que nuestras pruebas de alcance pequeño nos atraparán si cometemos un error.

Pruebas de servicio

Las pruebas de servicio están diseñadas para pasar por alto la interfaz de usuario y probar los servicios directamente. En una aplicación monolítica, es posible que solo estemos probando una colección de clases que brindan un *servicio* a la interfaz de usuario. Para un sistema que comprende varios servicios, una prueba de servicio probaría las capacidades de un servicio individual.

La razón por la que queremos probar un solo servicio por sí mismo es mejorar el aislamiento de la prueba para encontrar y solucionar problemas más rápido. Para lograr este aislamiento, debemos desconectar a todos los colaboradores externos para que solo el servicio en sí esté dentro del alcance, como muestra la [Figura 7-5](#).

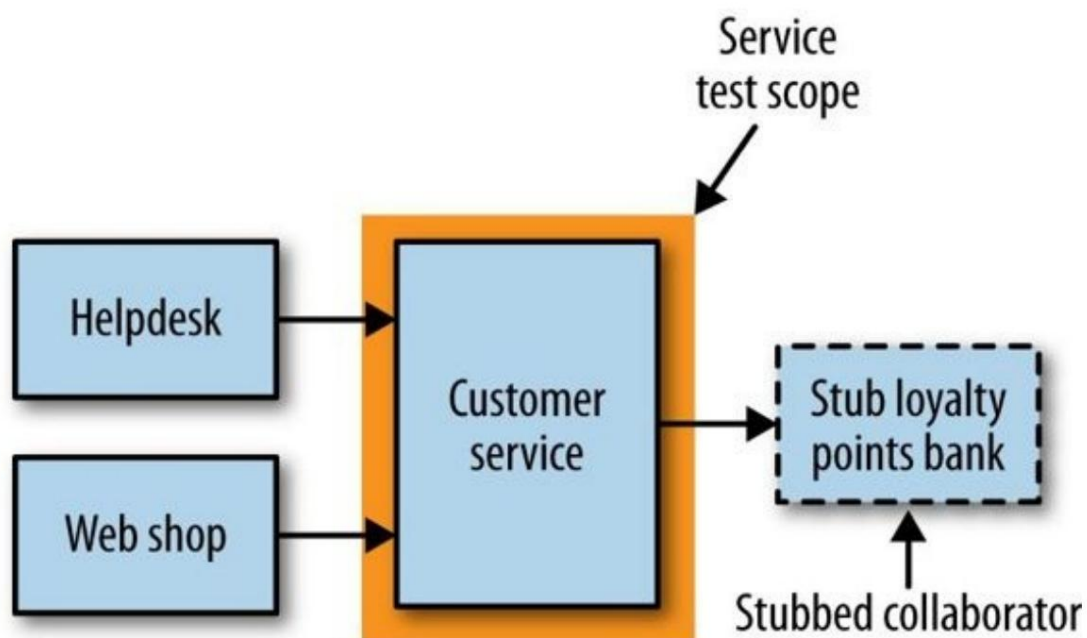


Figura 7-5. Pruebas de alcance de servicio en nuestro sistema de ejemplo

Algunas de estas pruebas pueden ser tan rápidas como las pruebas pequeñas, pero si decide realizar la prueba en una base de datos real, o pasar por las redes a los colaboradores aguas abajo, los tiempos de prueba pueden aumentar. También cubren más alcance que una simple prueba unitaria, por lo que cuando fallan puede ser más difícil detectar lo que está roto que con una prueba unitaria. Sin embargo, tienen muchas menos piezas móviles y, por lo tanto, son menos frágiles que las pruebas de mayor alcance.

Pruebas de extremo a extremo

Las pruebas de extremo a extremo son pruebas que se ejecutan en todo el sistema. A menudo, conducirán una GUI a través de un navegador, pero podrían imitar fácilmente otros tipos de interacción del usuario, como cargar un archivo.

Estas pruebas cubren una gran cantidad de código de producción, como vemos . Así que cuando pasan, en la **Figura 7-6** , se siente bien: tiene un alto grado de confianza en que el código que se está probando funcionará en producción. Pero este mayor alcance tiene sus inconvenientes y, como veremos en breve, puede ser muy complicado hacerlo bien en un contexto de microservicios.

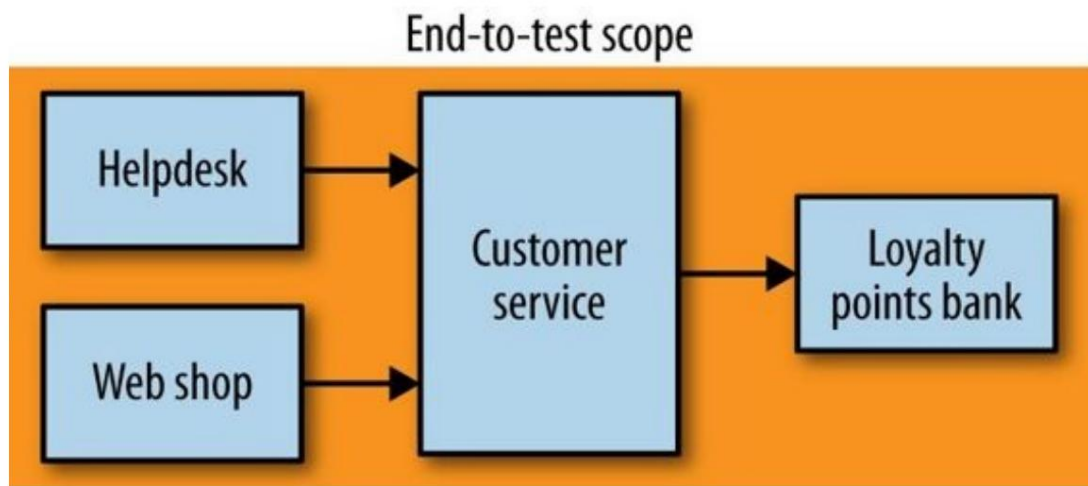


Figura 7-6. Alcance de las pruebas de extremo a extremo en nuestro sistema de ejemplo

Compensaciones

Cuando está leyendo la pirámide, lo más importante es que a medida que sube en la pirámide, el alcance de la prueba aumenta, al igual que nuestra confianza en que la funcionalidad que se está probando funciona. Por otro lado, el tiempo del ciclo de retroalimentación aumenta a medida que las pruebas tardan más en ejecutarse, y cuando una prueba falla, puede ser más difícil determinar qué funcionalidad se ha estropeado.

A medida que desciende en la pirámide, en general, las pruebas se vuelven mucho más rápidas, por lo que obtenemos ciclos de retroalimentación mucho más rápidos. Encontramos funcionalidad rota más rápido, nuestras compilaciones de integración continua son más rápidas y es menos probable que pasemos a una nueva tarea antes de descubrir que hemos roto algo.

Cuando esas pruebas de menor alcance fallan, también tendemos a saber qué se rompió, a menudo exactamente qué línea de código. Por otro lado, no tenemos mucha confianza en que nuestro sistema en su conjunto funcione si solo hemos probado una línea de código.

Cuando las pruebas de alcance más amplio como nuestro servicio o las pruebas de un extremo a otro fallan, intentaremos escribir una prueba unitaria rápida para detectar ese problema en el futuro. De esa manera, estamos constantemente tratando de mejorar nuestros ciclos de retroalimentación.

Prácticamente todos los equipos en los que he trabajado han usado nombres diferentes a los que usa Cohn en la pirámide.

Como sea que los llame, la conclusión clave es que querrá pruebas de diferente alcance para diferentes propósitos.

¿Cuanto?

Entonces, si todas estas pruebas tienen compensaciones, ¿cuántas de cada tipo quiere? Una buena regla general es que probablemente desee un orden de magnitud más de pruebas a medida que desciende de la pirámide, pero lo importante es saber que tiene diferentes tipos de pruebas automatizadas y comprender si su saldo actual le da un problema.

Trabajé en un sistema monolítico, por ejemplo, donde teníamos 4000 pruebas unitarias, 1000 pruebas de servicio y 60 pruebas de extremo a extremo. Decidimos que, desde el punto de vista de la retroalimentación, teníamos demasiadas pruebas de servicio y de extremo a extremo (las últimas fueron las peores en el impacto de los ciclos de retroalimentación), por lo que trabajamos arduamente para reemplazar la cobertura de la prueba con pruebas de menor alcance. pruebas

Un antipatrón común es lo que a menudo se denomina *cono de nieve de prueba* o pirámide invertida. Aquí, hay pocas o ninguna prueba de alcance pequeño, con toda la cobertura en pruebas de alcance grande. Estos proyectos a menudo tienen ejecuciones de prueba extremadamente lentas y ciclos de retroalimentación muy largos. Si estas pruebas se ejecutan como parte de la integración continua, no obtendrá muchas compilaciones, y la naturaleza de los tiempos de compilación significa que la compilación puede permanecer rota durante un período prolongado cuando algo falla.

Implementación de pruebas de servicio

La implementación de pruebas unitarias es un asunto bastante simple en el gran esquema de las cosas, y hay mucha documentación que explica cómo escribirlas. Las pruebas de servicio y de extremo a extremo son las que resultan más interesantes.

Nuestras pruebas de servicio quieren probar una parte de la funcionalidad en todo el servicio, pero para aislarnos de otros servicios necesitamos encontrar alguna forma de aislar a todos nuestros colaboradores. Entonces, si quisiéramos escribir una prueba como esta para el servicio al cliente de la **Figura 7-3**, implementaríamos una instancia del servicio al cliente y, como se discutió anteriormente, querríamos desconectar cualquier servicio posterior.

Una de las primeras cosas que hará nuestra compilación de integración continua es crear un artefacto binario para nuestro servicio, por lo que implementarlo es bastante sencillo. Pero, ¿cómo manejamos falsificar a los colaboradores posteriores?

Nuestro conjunto de pruebas de servicios debe iniciar servicios de código auxiliar para cualquier colaborador posterior (o asegurarse de que se estén ejecutando) y configurar el servicio bajo prueba para conectarse a los servicios de código auxiliar. Luego, debemos configurar los stubs para enviar respuestas para imitar los servicios del mundo real. Por ejemplo, podríamos configurar el resguardo del banco de puntos de lealtad para devolver saldos de puntos conocidos para ciertos clientes.

Burlarse o golpearse

Cuando hablo de aplicar stubs a colaboradores posteriores, me refiero a que creamos un servicio stub que responde con respuestas enlatadas a solicitudes conocidas del servicio que se está probando. Por ejemplo, podría decirle a mi banco de puntos de talón que cuando se le solicite el saldo del cliente 123, debe devolver 15,000. A la prueba no le importa si el código auxiliar se llama 0, 1 o 100 veces. Una variación de esto es usar un simulacro en lugar de un resguardo.

Cuando uso un simulacro, en realidad voy más allá y me aseguro de que se haya realizado la llamada. Si no se realiza la llamada esperada, la prueba falla. La implementación de este enfoque requiere más inteligencia en los colaboradores falsos que creamos, y si se usa en exceso, las pruebas pueden volverse frágiles. Sin embargo, como se señaló, a un stub no le importa si se llama 0, 1 o muchas veces.

A veces, sin embargo, los simulacros pueden ser muy útiles para garantizar que ocurran los efectos secundarios esperados. Por ejemplo, podría querer verificar que cuando creo un cliente, se configura un nuevo saldo de puntos para ese cliente. El equilibrio entre el stubing y las llamadas simuladas es delicado, y es tan complicado en las pruebas de servicio como en las pruebas unitarias. En general, sin embargo, utilizo mucho más stubs que simulacros para las pruebas de servicio. Para una discusión más profunda de esta compensación, eche un vistazo a *Software orientado a objetos en crecimiento, guiado por pruebas*, por Steve Freeman y Nat Pryce (Addison-Wesley).

En general, rara vez uso simulacros para este tipo de pruebas. Pero tener una herramienta que pueda hacer ambas cosas es útil.

Si bien siento que los stubs y los simulacros en realidad están bastante bien diferenciados, sé que la distinción puede ser confusa para algunos, especialmente cuando algunas personas agregan otros términos como *falsificaciones*, *espías* y *tontos*. Martin Fowler llama a todas estas cosas, incluidos los stubs y los simulacros, **dobles de prueba**.

Un servicio de talón más inteligente

Normalmente, para los servicios de talón, los he enrollado yo mismo. He usado de todo, desde Apache o Nginx hasta contenedores Embedded Jetty o incluso servidores web Python lanzados desde la línea de comandos que se usan para lanzar servidores auxiliares para tales casos de prueba. Probablemente he reproducido el mismo trabajo una y otra vez al crear estos fragmentos. Mi colega de ThoughtWorks, Brandon Bryars, potencialmente nos ha ahorrado a muchos de nosotros una gran cantidad de trabajo con su servidor auxiliar/simulacro llamado **Mountebank**.

Puede pensar en Mountebank como un pequeño dispositivo de software programable a través de HTTP. El hecho de que esté escrito en NodeJS es completamente opaco para cualquier servicio de llamadas. Cuando se inicia, le envía comandos que le indican en qué puerto conectarse, qué protocolo manejar (actualmente se admiten TCP, HTTP y HTTPS, con más planificados) y qué respuestas debe enviar cuando se envían las solicitudes. También admite establecer expectativas si desea usarlo como simulacro. Puede agregar o eliminar estos extremos de código auxiliar a voluntad, lo que hace posible que una sola instancia de Mountebank cree más de una dependencia descendente.

Entonces, si queremos ejecutar nuestras pruebas de servicio solo para nuestro servicio al cliente, podemos lanzar el servicio al cliente y una instancia de Mountebank que actúa como nuestro banco de puntos de fidelidad. Y si esas pruebas pasan, ¡puedo implementar el servicio al cliente de inmediato! ¿O puedo? ¿Qué pasa con los servicios que llaman al servicio de atención al cliente: el servicio de asistencia técnica y la tienda web? ¿Sabemos si hemos hecho algún cambio que pueda romperlos? Por supuesto, nos hemos olvidado de las pruebas importantes en la parte superior de la pirámide: las pruebas de extremo a extremo.

Esas pruebas engañosas de extremo a extremo

En un sistema de microservicios, las capacidades que exponemos a través de nuestras interfaces de usuario son proporcionadas por una serie de servicios. El punto de las pruebas de extremo a extremo como se describe en la pirámide de Mike Cohn es impulsar la funcionalidad a través de estas interfaces de usuario contra todo lo que hay debajo para darnos una visión general de una gran parte de nuestro sistema.

Entonces, para implementar una prueba de extremo a extremo, necesitamos implementar varios servicios juntos y luego ejecutar una prueba contra todos ellos. ¡Obviamente, esta prueba tiene mucho más alcance, lo que resulta en una mayor confianza en que nuestro sistema funciona! Por otro lado, estas pruebas pueden ser más lentas y dificultar el diagnóstico de fallas. Profundicemos un poco más usando nuestro ejemplo anterior para ver cómo pueden encajar estas pruebas.

Imagina que queremos lanzar una nueva versión del servicio al cliente. Queremos implementar nuestros cambios en producción lo antes posible, pero nos preocupa que podamos haber introducido un cambio que podría dañar el servicio de asistencia o la tienda web. No hay problema: implementemos todos nuestros servicios juntos y ejecutemos algunas pruebas con el servicio de asistencia técnica y la tienda web para ver si hemos introducido un error. Ahora, un enfoque ingenuo sería simplemente agregar estas pruebas al final de nuestra tubería de servicio al cliente, como en la [Figura 7-7](#).



Figura 7-7. Agregar nuestra etapa de pruebas de extremo a extremo: ¿el enfoque correcto?

Hasta aquí todo bien. Pero la primera pregunta que debemos hacernos es ¿qué versión de los otros servicios debemos usar? ¿Deberíamos ejecutar nuestras pruebas con las versiones de helpdesk y web shop que están en producción? Es una suposición sensata, pero ¿qué pasa si hay una nueva versión del servicio de asistencia técnica o de la tienda web en cola para publicarse? ¿Qué debemos hacer entonces?

Otro problema: si tenemos un conjunto de pruebas de servicio al cliente que implementan muchos servicios y ejecutan pruebas contra ellos, ¿qué pasa con las pruebas de extremo a extremo que ejecutan los otros servicios? Si están probando lo mismo, es posible que nos encontremos cubriendo gran parte del mismo terreno y duplicando gran parte del esfuerzo para implementar todos esos servicios en primer lugar.

Podemos lidiar con estos dos problemas de manera elegante al tener varias canalizaciones *en abanico* en una sola etapa de prueba de extremo a extremo. Aquí, cada vez que se activa una nueva compilación de uno de nuestros servicios, ejecutamos nuestras pruebas de extremo a extremo, un ejemplo de lo que podemos ver en la [Figura 7-8](#).

Algunas herramientas de CI con una mejor compatibilidad con la canalización de compilación permitirán modelos de abanico como este listos para usar.

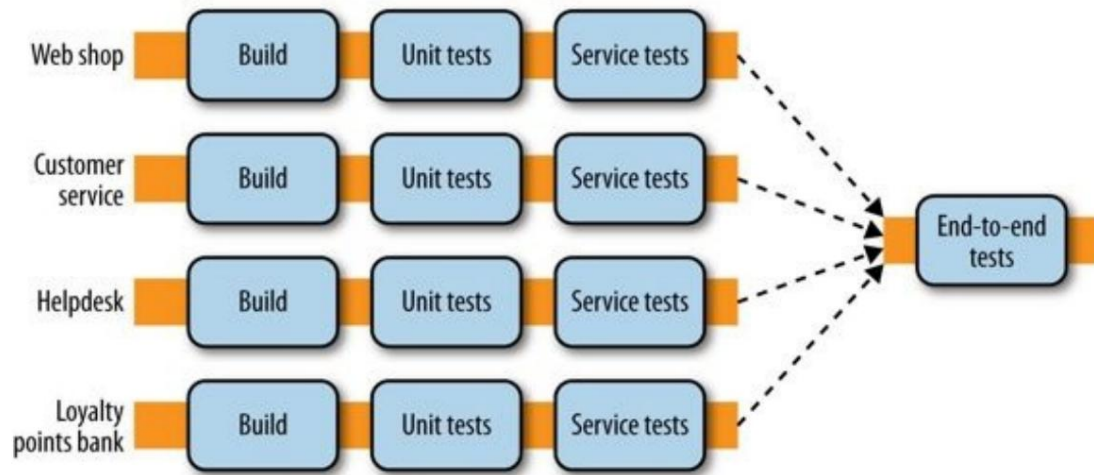


Figura 7-8. Una forma estándar de manejar pruebas de extremo a extremo en todos los servicios

Entonces, cada vez que cambia alguno de nuestros servicios, ejecutamos las pruebas locales para ese servicio. Si esas pruebas pasan, activamos nuestras pruebas de integración. Genial, ¿eh? Bueno, hay algunos problemas.

Desventajas de las pruebas de extremo a extremo

Desafortunadamente, existen muchas desventajas en las pruebas de extremo a extremo.

Pruebas escamosas y frágiles

A medida que aumenta el alcance de la prueba, también aumenta el número de piezas móviles. Estas partes móviles pueden introducir fallas en las pruebas que no muestran que la funcionalidad bajo prueba esté rota, sino que ha ocurrido algún otro problema. Como ejemplo, si tenemos una prueba para verificar que podemos hacer un pedido de un solo CD, pero estamos ejecutando esa prueba en cuatro o cinco servicios, si alguno de ellos está caído, podríamos obtener una falla que no tiene nada que hacer con la naturaleza de la prueba en sí. Del mismo modo, una falla temporal en la red podría causar que una prueba falle sin decir nada sobre la funcionalidad que se está probando.

Cuanto más partes móviles, más frágiles pueden ser nuestras pruebas y menos deterministas. Si tiene pruebas que a veces fallan, pero todos las vuelven a ejecutar porque pueden pasar nuevamente más tarde, entonces tiene pruebas escamosas. No son solo las pruebas que cubren muchos procesos diferentes las culpables aquí. Las pruebas que cubren la funcionalidad que se ejerce en múltiples subprocesos a menudo son problemáticas, donde una falla podría significar una condición de carrera, un tiempo de espera o que la funcionalidad está realmente rota. Las pruebas escamosas son el enemigo. Cuando fallan, no nos dicen mucho. Volvemos a ejecutar nuestras compilaciones de CI con la esperanza de que vuelvan a pasar más tarde, solo para ver que los registros se acumulan y, de repente, nos encontramos con una gran cantidad de funciones rotas.

Cuando detectamos pruebas escamosas, es fundamental que hagamos todo lo posible para eliminarlas. De lo contrario, comenzamos a perder la fe en un conjunto de pruebas que "siempre falla así". Un conjunto de pruebas con pruebas irregulares puede convertirse en víctima de lo que Diane Vaughan llama la *normalización de la desviación*: la idea de que, con el tiempo, podemos acostumbrarnos tanto a que las cosas salgan mal que comencemos a aceptarlas como normales y no como un problema. [?] Esta tendencia es buena y significa que tan pronto como podamos antes de comenzar a asumir que las pruebas fallidas están bien.

En "**Erradicación del no determinismo en las pruebas**", Martin Fowler defiende el enfoque de que si tiene pruebas inestables, debe rastrearlas y, si no puede corregirlas de inmediato, eliminarlas de la suite para poder tratarlas. Vea si puede reescribirlos para evitar probar el código que se ejecuta en varios subprocesos. Vea si puede hacer que el entorno subyacente sea más estable. Mejor aún, vea si puede reemplazar la prueba escamosa con una prueba de menor alcance que es menos probable que muestre problemas. En algunos casos, cambiar el software bajo prueba para que sea más fácil de probar también puede ser el camino correcto.

¿Quién escribe estas pruebas?

Con las pruebas que se ejecutan como parte de la tubería para un servicio específico, el punto de partida sensato es que el equipo propietario de ese servicio debe escribir esas pruebas (hablaremos más sobre la propiedad del servicio en el [Capítulo 10](#)). Pero si consideramos que podríamos tener varios equipos involucrados, y el paso de las pruebas de extremo a extremo ahora se comparte efectivamente entre los equipos, ¿quién escribe y se ocupa de estas pruebas?

He visto una serie de anti-patrones causados aquí. Estas pruebas se vuelven gratuitas, y todos los equipos tienen acceso para agregar pruebas sin comprender el estado de todo el conjunto. Esto a menudo puede resultar en una explosión de casos de prueba, lo que a veces resulta en el cono de nieve de prueba del que hablamos anteriormente. He visto situaciones en las que, debido a que no había una propiedad obvia real de estas pruebas, sus resultados se ignoran. Cuando se rompen, todos asumen que es problema de otra persona, por lo que no les importa si las pruebas están pasando.

A veces, las organizaciones reaccionan haciendo que un equipo dedicado escriba estas pruebas. Esto puede ser desastroso. El equipo que desarrolla el software se aleja cada vez más de las pruebas de su código. Los tiempos de ciclo aumentan, ya que los propietarios de los servicios terminan esperando que el equipo de prueba escriba pruebas de extremo a extremo para la funcionalidad que acaban de escribir. Debido a que otro equipo escribe estas pruebas, el equipo que escribió el servicio está menos involucrado y, por lo tanto, es menos probable que sepa cómo ejecutar y corregir estas pruebas. Aunque lamentablemente sigue siendo un patrón organizativo común, veo que se produce un daño significativo cada vez que un equipo se distancia de escribir pruebas para el código que escribió en primer lugar.

Acertar en este aspecto es realmente difícil. No queremos duplicar el esfuerzo, ni queremos centralizar completamente esto en la medida en que los servicios de creación de equipos estén demasiado alejados de las cosas. El mejor equilibrio que he encontrado es tratar el conjunto de pruebas de extremo a extremo como una base de código compartida, pero con propiedad conjunta. Los equipos son libres de registrarse en esta suite, pero la propiedad de la salud de la suite debe ser compartida entre los equipos que desarrollan los servicios. Si desea hacer un uso extensivo de las pruebas de extremo a extremo con varios equipos, creo que este enfoque es esencial y, sin embargo, lo he visto muy raramente y nunca sin problemas.

¿Cuánto tiempo?

Estas pruebas de extremo a extremo pueden llevar un tiempo. Los he visto tardar hasta un día en ejecutarse, si no más, y en un proyecto en el que trabajé, ¡una suite de regresión completa tardó seis semanas! Rara vez veo que los equipos realmente seleccionen sus suites de prueba de un extremo a otro para reducir la superposición en la cobertura de la prueba, o que dediquen suficiente tiempo a hacerlas más rápidas.

Esta lentitud, combinada con el hecho de que estas pruebas a menudo pueden ser escamosas, puede ser un problema importante. Un conjunto de pruebas que lleva todo el día y que a menudo tiene fallas que no tienen nada que ver con la funcionalidad rota es un desastre. Incluso si su funcionalidad *está* rota, podría llevarle muchas horas descubrirlo, momento en el que muchos de nosotros ya habríamos pasado a otras actividades, y el cambio de contexto al cambiar nuestros cerebros para solucionar el problema es doloroso.

Podemos mejorar algo de esto ejecutando pruebas en paralelo, por ejemplo, haciendo uso de herramientas como Selenium Grid. Sin embargo, este enfoque no reemplaza la comprensión real de lo que se debe probar y la *eliminación* activa de las pruebas que ya no se necesitan.

La eliminación de las pruebas es a veces un ejercicio tenso, y sospecho que tiene mucho en común con las personas que quieren eliminar ciertas medidas de seguridad en los aeropuertos. No importa cuán ineficaces puedan ser las medidas de seguridad, cualquier conversación sobre eliminarlas a menudo se contrarresta con reacciones instintivas sobre no preocuparse por la seguridad de las personas o querer que los terroristas ganen. Es difícil tener una conversación equilibrada sobre el valor que agrega algo frente a la carga que conlleva. También puede ser una compensación difícil de riesgo/recompensa. ¿Te dan las gracias si eliminas una prueba? Quizás. Pero ciertamente lo culparán si una prueba que eliminó deja pasar un error. Sin embargo, cuando se trata de conjuntos de pruebas de mayor alcance, esto es exactamente lo que debemos poder hacer. Si la misma característica se cubre en 20 pruebas diferentes, tal vez podamos deshacernos de la mitad de ellas, ¡ya que esas 20 pruebas tardan 10 minutos en ejecutarse!

Lo que esto requiere es una mejor comprensión del riesgo, algo en lo que los humanos somos famosos. Como resultado, esta selección y gestión inteligente de pruebas de mayor alcance y alta carga ocurre con una frecuencia increíblemente baja. Desear que la gente hiciera esto más no es lo mismo que hacer que suceda.

El gran amontonamiento

Los largos ciclos de retroalimentación asociados con las pruebas de extremo a extremo no son solo un problema cuando se trata de la productividad del desarrollador. Con un conjunto de pruebas largo, se tarda un tiempo en corregir cualquier interrupción, lo que reduce la cantidad de tiempo que se puede esperar que pasen las pruebas de un extremo a otro.

Si implementamos solo el software que ha superado con éxito todas nuestras pruebas (¡lo cual deberíamos hacer!), esto significa que menos de nuestros servicios llegarán al punto de poder implementarse en producción.

Esto puede conducir a una acumulación. Mientras se repara una etapa de prueba de integración rota, se pueden acumular más cambios de los equipos ascendentes. Aparte del hecho de que esto puede dificultar la reparación de la compilación, significa que aumenta el alcance de los cambios que se implementarán. Una forma de resolver esto es no permitir que las personas se registren si las pruebas de un extremo a otro están fallando, pero dado el largo tiempo del conjunto de pruebas, esto a menudo no es práctico. Intente decir: "Ustedes, los 30 desarrolladores: ¡no se registren hasta que arreglemos esta compilación de siete horas de duración!"

Cuanto mayor sea el alcance de una implementación y mayor sea el riesgo de una versión, es más probable que rompamos algo. Un factor clave para garantizar que podamos lanzar nuestro software con frecuencia se basa en la idea de que lanzamos pequeños cambios tan pronto como estén listos.

La metaversión

Con el paso de prueba de un extremo a otro, es fácil comenzar a pensar : *Sé que todos estos servicios en estas versiones funcionan juntos, entonces, ¿por qué no implementarlos todos juntos?* Esto se convierte rápidamente en una conversación del tipo: Entonces, *¿por qué no usar un número de versión para todo el sistema?* Para citar a **Brandon Bryars**, "Ahora tienes problemas con 2.1.0".

Al versionar los cambios realizados en varios servicios, adoptamos de manera efectiva la idea de que cambiar e implementar varios servicios a la vez es aceptable. Se convierte en la norma, se convierte en OK. Al hacerlo, cedemos una de las principales ventajas de los microservicios: la capacidad de implementar un servicio por sí mismo, independientemente de otros servicios.

Con demasiada frecuencia, el enfoque de aceptar que se implementen varios servicios juntos deriva en una situación en la que los servicios se acoplan. En poco tiempo, los servicios bien separados se enredan cada vez más con otros, y nunca se da cuenta de que nunca intenta implementarlos por sí mismos. Terminas con un lío enredado en el que tienes que orquestar la implementación de múltiples servicios a la vez y, como discutimos anteriormente, este tipo de acoplamiento puede dejarnos en un lugar peor de lo que estaríamos con una sola aplicación monolítica.

Esto es malo.

Viajes de prueba, no historias

A pesar de las desventajas que se acaban de describir, para muchos usuarios, las pruebas de extremo a extremo aún pueden ser manejables con uno o dos servicios, y en estas situaciones aún tienen mucho sentido. Pero, ¿qué pasa con 3, 4, 10 o 20 servicios? Muy rápidamente, estos conjuntos de pruebas se hinchan enormemente y, en el peor de los casos, pueden provocar una explosión similar a la cartesiana en los escenarios bajo prueba.

Esta situación empeora si caemos en la trampa de agregar una nueva prueba de extremo a extremo para cada pieza de funcionalidad que agregamos. Muéstrenme una base de código en la que cada nueva historia resulte en una nueva prueba integral y les mostraré un conjunto de pruebas inflado que tiene ciclos de retroalimentación deficientes y grandes superposiciones en la cobertura de las pruebas.

La mejor manera de contrarrestar esto es centrarse en una *pequeña* cantidad de recorridos principales para probar todo el sistema. Cualquier funcionalidad que no se cubra en estos viajes principales debe cubrirse en pruebas que analicen los servicios de forma aislada unos de otros. Estos viajes deben ser acordados mutuamente y de propiedad conjunta. Para nuestra tienda de música, podríamos centrarnos en acciones como pedir un CD, devolver un producto o tal vez crear un nuevo cliente: interacciones de alto valor y muy pocas.

Al centrarnos en un número pequeño (y me refiero a *pequeño*: dos dígitos muy bajos incluso para sistemas complejos) de pruebas, podemos reducir las desventajas de las pruebas de integración, pero no podemos evitarlas todas. ¿Hay una mejor manera?

Pruebas impulsadas por el consumidor al rescate

¿Cuál es uno de los problemas clave que estamos tratando de abordar cuando usamos las pruebas de integración descritas anteriormente? Estamos tratando de asegurarnos de que cuando implementamos un nuevo servicio en producción, nuestros cambios no afecten a los consumidores. Una forma en que podemos hacer esto sin requerir pruebas contra el consumidor real es mediante un *contrato impulsado por el consumidor (CDC)*.

Con los CDC, estamos definiendo las expectativas de un consumidor sobre un servicio (o productor).

Las expectativas de los consumidores se capturan en forma de código como pruebas, que luego se ejecutan contra el productor. Si se hace correctamente, estos CDC deben ejecutarse como parte de la compilación de CI del productor, asegurándose de que nunca se implemente si rompe uno de estos contratos. Muy importante desde el punto de vista de la retroalimentación de la prueba, estas pruebas deben ejecutarse solo contra un único productor de forma aislada, por lo que pueden ser más rápidas y confiables que las pruebas de extremo a extremo que podrían reemplazar.

Como ejemplo, revisemos nuestro escenario de servicio al cliente. El servicio de atención al cliente tiene dos consumidores separados: el servicio de asistencia técnica y la tienda web. Ambos servicios de consumo tienen expectativas sobre cómo se comportará el servicio al cliente. En este ejemplo, crea dos conjuntos de pruebas: uno para cada consumidor que representa el uso del servicio de atención al cliente por parte del servicio de asistencia técnica y de la tienda web. Una buena práctica aquí es hacer que alguien de los equipos de productores y consumidores colabore en la creación de las pruebas, por lo que quizás las personas de la tienda web y los equipos de asistencia técnica se emparejen con personas del equipo de servicio al cliente.

Debido a que estos CDC son expectativas sobre cómo debe comportarse el servicio de atención al cliente, se pueden ejecutar contra el servicio de atención al cliente por sí mismo con cualquiera de sus dependencias posteriores eliminadas, como muestra la **Figura 7-9**. Desde el punto de vista del alcance, se ubican en el mismo nivel en la pirámide de prueba que las pruebas de servicio, aunque con un enfoque muy diferente, como se muestra en la **Figura 7-10**. Estas pruebas se enfocan en cómo un consumidor usará el servicio y el gatillo si se rompen es muy diferente en comparación con las pruebas de servicio. Si uno de estos CDC se rompe durante la creación del servicio al cliente, resulta obvio qué consumidor se vería afectado. En este punto, puede solucionar el problema o iniciar la discusión sobre la introducción de un cambio importante de la manera que discutimos en el **Capítulo 4**. Entonces, con los CDC, podemos identificar un cambio importante antes de que nuestro software entre en producción sin tener que usar una prueba de extremo a extremo potencialmente costosa.

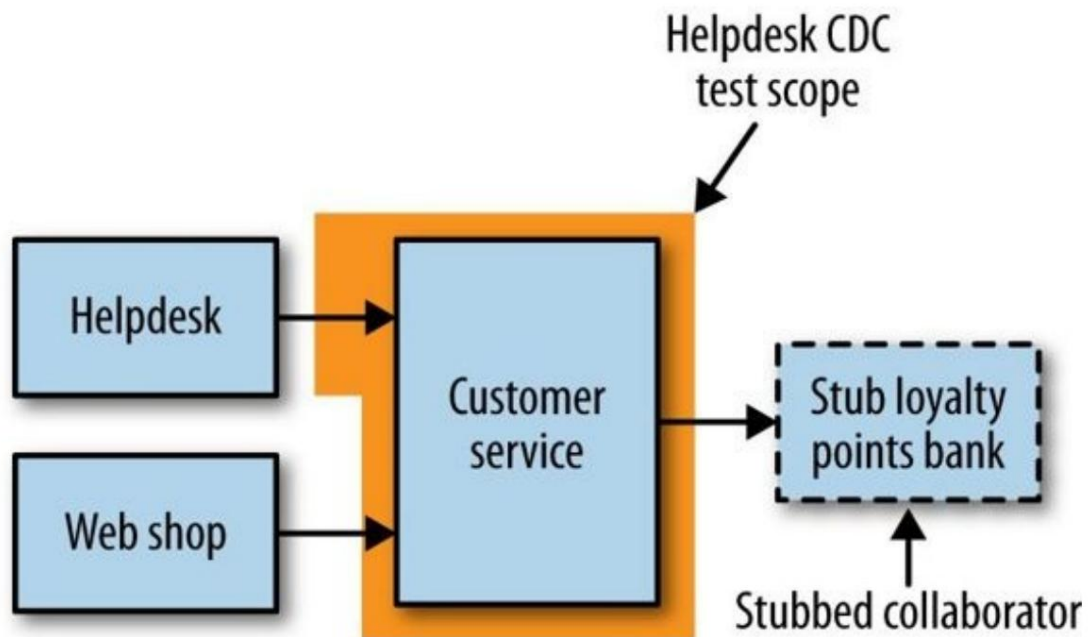


Figura 7-9. Pruebas impulsadas por el consumidor en el contexto de nuestro ejemplo de servicio al cliente

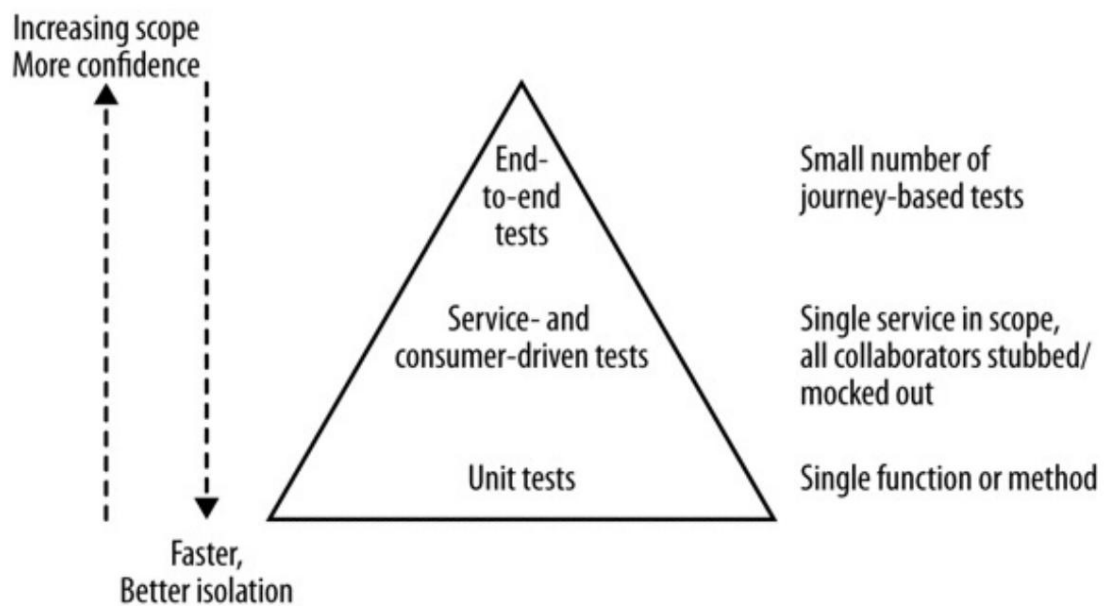


Figura 7-10. Integración de pruebas impulsadas por el consumidor en la pirámide de prueba

Pacto

Pact es una herramienta de prueba impulsada por el consumidor que se desarrolló originalmente internamente en RealEstate.com.au, pero ahora es de código abierto, con Beth Skurrie dirigiendo la mayor parte del desarrollo. Originalmente solo para Ruby, Pact ahora incluye puertos JVM y .NET.

Pact funciona de una manera muy interesante, como se resume en la **Figura 7-11**. El consumidor comienza definiendo las expectativas del productor que utiliza un Ruby DSL. Luego, inicia un servidor simulado local y ejecuta esta expectativa contra él para crear el archivo de especificación de Pact. El archivo Pact es solo una especificación JSON formal; obviamente podría codificarlos a mano, pero usar la API de lenguaje es mucho más fácil. Esto también le brinda un servidor simulado en ejecución que se puede usar para más pruebas aisladas del consumidor.

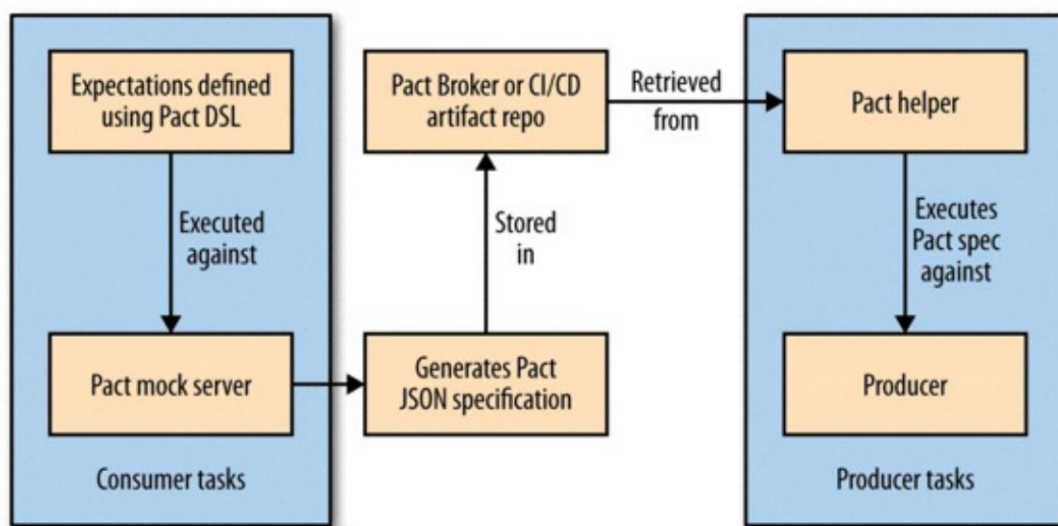


Figura 7-11. Una descripción general de cómo Pact realiza pruebas impulsadas por el consumidor

En el lado del productor, luego verifica que esta especificación del consumidor se cumpla mediante el uso de la especificación JSON Pact para generar llamadas contra su API y verificar las respuestas. Para que esto funcione, el código base del productor necesita acceso al archivo Pact. Como discutimos anteriormente en el **Capítulo 6**, esperamos que tanto el consumidor como el productor estén en diferentes construcciones. El uso de una especificación JSON independiente del idioma es un toque especialmente agradable. Significa que puede generar la especificación del consumidor usando un cliente Ruby, pero utilícelo para verificar un productor Java usando el puerto JVM de Pact.

A medida que el consumidor crea la especificación JSON Pact, debe convertirse en un artefacto al que tenga acceso la compilación del productor. Puede almacenar esto en el repositorio de artefactos de su herramienta de CI/CD, o bien utilizar Pact Broker, que le permite almacenar múltiples versiones de sus especificaciones de Pact. Esto podría permitirle ejecutar sus pruebas de contrato impulsadas por el consumidor en varias versiones diferentes de los consumidores, si desea probar, por ejemplo, la versión del consumidor en producción y la versión del consumidor que se creó más recientemente.

De manera confusa, hay un proyecto de código abierto de ThoughtWorks llamado **Pacto**, que también es una herramienta de Ruby utilizada para pruebas dirigidas por el consumidor. Tiene la capacidad de registrar interacciones.

entre cliente y servidor para generar las expectativas. Esto hace que la redacción de contratos orientados al consumidor para los servicios existentes sea bastante fácil. Con Pacto, una vez generadas, estas expectativas son más o menos estáticas, mientras que con Pacto regeneras las expectativas en el consumidor con cada construcción. El hecho de que pueda definir las expectativas de las capacidades que el productor puede que aún no tenga también encaja mejor en un flujo de trabajo en el que el servicio de producción aún se está desarrollando (o aún no se ha desarrollado).

Se trata de conversaciones

En ágil, las historias a menudo se denominan marcadores de posición para una conversación. Los CDC son así. Se convierten en la codificación de un conjunto de debates sobre cómo debería ser una API de servicio y, cuando fallan, se convierten en un punto desencadenante para tener conversaciones sobre cómo debería evolucionar esa API.

Es importante comprender que los CDC requieren una buena comunicación y confianza entre el consumidor y el servicio que produce. Si ambas partes están en el mismo equipo (¡o en la misma persona!), entonces esto no debería ser difícil. Sin embargo, si está consumiendo un servicio proporcionado por un tercero, es posible que no tenga la frecuencia de comunicación o la confianza para que los CDC funcionen. En estas situaciones, es posible que tenga que conformarse con pruebas de integración limitadas de mayor alcance alrededor del componente que no es de *confianza*. Alternativamente, si está creando una API para miles de consumidores potenciales, como una API de servicio web disponible públicamente, es posible que usted mismo deba desempeñar el papel del consumidor (o tal vez trabajar con un subconjunto de sus consumidores) al definir estas pruebas. . Romper un gran número de consumidores externos es una idea bastante mala, por lo que, en todo caso, ¡aumenta la importancia de los CDC!

Entonces, ¿debería usar pruebas de extremo a extremo?

Como se describió en detalle anteriormente en el capítulo, las pruebas de un extremo a otro tienen una gran cantidad de desventajas que crecen significativamente a medida que agrega más partes móviles bajo prueba. Hablando con personas que han estado implementando microservicios a escala durante un tiempo, he aprendido que la mayoría de ellos, con el tiempo, eliminan por completo la necesidad de pruebas de extremo a extremo a favor de herramientas como CDC y monitoreo mejorado. Pero no necesariamente descartan esas pruebas. Terminan usando muchas de esas pruebas de viaje de extremo a extremo para monitorear el sistema de producción usando una técnica llamada *monitoreo semántico*, que discutiremos más en el [Capítulo 8](#).

Puede ver la ejecución de pruebas de extremo a extremo antes de la implementación de producción como ruedas de entrenamiento. Mientras aprende cómo funcionan los CDC y mejora sus técnicas de implementación y monitoreo de producción, estas pruebas integrales pueden formar una red de seguridad útil, en la que está intercambiando el tiempo del ciclo por un menor riesgo. Pero a medida que mejora esas otras áreas, puede comenzar a reducir su dependencia de las pruebas de extremo a extremo hasta el punto en que ya no sean necesarias.

Del mismo modo, puede trabajar en un entorno donde el apetito por *aprender en producción* es bajo, y la gente preferiría trabajar tan duro como sea posible para eliminar cualquier defecto antes de la producción, incluso si eso significa que el software tarda más en enviarse. Siempre que comprenda que no puede estar seguro de haber eliminado todas las fuentes de defectos, y que aún necesitará contar con un control y una corrección efectivos en la producción, esta puede ser una decisión sensata.

Obviamente, tendrá una mejor comprensión del perfil de riesgo de su propia organización que yo, pero lo desafiaría a pensar detenidamente sobre la cantidad de pruebas de extremo a extremo que realmente necesita hacer.

Pruebas después de la producción

La mayoría de las pruebas se realizan antes de que el sistema esté en producción. Con nuestras pruebas estamos definiendo una serie de modelos con los que esperamos comprobar si nuestro sistema funciona y se comporta como nos gustaría, tanto funcional como no funcionalmente. Pero si nuestros modelos no son perfectos, encontraremos problemas cuando nuestros sistemas se utilicen con ira. Los errores entran en producción, se descubren nuevos modos de falla y nuestros usuarios usan el sistema de formas que nunca podríamos esperar.

Una reacción a esto es a menudo definir más y más pruebas y refinar nuestros modelos, para detectar más problemas temprano y reducir la cantidad de problemas que encontramos con nuestro sistema de producción en ejecución. Sin embargo, en cierto punto tenemos que aceptar que obtenemos rendimientos decrecientes con este enfoque. Con las pruebas antes de la implementación, no podemos reducir la posibilidad de falla a cero.

Separación de la implementación de la versión

Una forma en la que podemos detectar más problemas antes de que ocurran es ampliar el lugar donde ejecutamos nuestras pruebas más allá de los pasos tradicionales previos a la implementación. En cambio, si podemos implementar nuestro software y probarlo in situ antes de dirigir las cargas de producción contra él, podemos detectar problemas específicos de un entorno determinado. Un ejemplo común de esto es el *conjunto de pruebas de humo*, una colección de pruebas diseñadas para ejecutarse en software recién implementado para confirmar que la implementación funcionó. Estas pruebas lo ayudan a detectar cualquier problema ambiental local. Si está utilizando un solo comando de línea de comandos para implementar un microservicio determinado (y debería hacerlo), este comando debería ejecutar las pruebas de humo automáticamente.

Otro ejemplo de esto es lo que se denomina *despliegue azul/verde*. Con azul/verde, tenemos dos copias de nuestro software implementadas a la vez, pero solo una versión recibe solicitudes reales.

Consideremos un ejemplo simple, visto en la [Figura 7-12](#). En producción tenemos la v123 del servicio al cliente en vivo. Queremos implementar una nueva versión, v456. Implementamos esto junto con v123, pero no le dirigimos ningún tráfico. En su lugar, realizamos algunas pruebas in situ con la versión recién implementada. Una vez que las pruebas han funcionado, dirigimos la carga de producción a la nueva versión v456 del servicio de atención al cliente. Es común mantener la versión anterior durante un período corto de tiempo, lo que permite una recuperación rápida si detecta algún error.

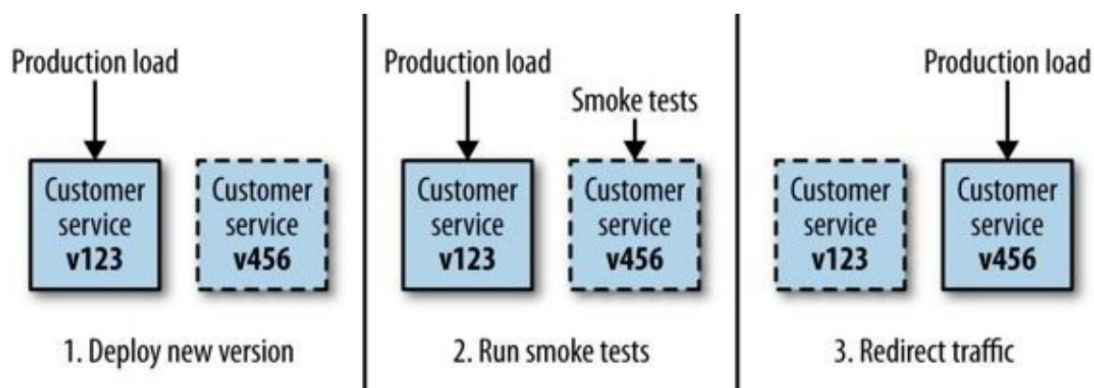


Figura 7-12. Uso de implementaciones azul/verde para separar la implementación del lanzamiento

La implementación de la implementación azul/verde requiere algunas cosas. Primero, debe poder dirigir el tráfico de producción a diferentes hosts (o colecciones de hosts). Puede hacer esto cambiando las entradas de DNS o actualizando la configuración de equilibrio de carga. También debe poder aprovisionar suficientes hosts para que ambas versiones del microservicio se ejecuten a la vez. Si está utilizando un proveedor de nube elástica, esto podría ser sencillo. El uso de implementaciones azul/verde le permite reducir el riesgo de implementación, así como también le brinda la oportunidad de revertir si encuentra un problema. Si se le da bien esto, todo el proceso se puede automatizar por completo, con la implementación completa o la reversión sin intervención humana.

Aparte del beneficio de permitirnos probar nuestros servicios in situ antes de enviarles tráfico de producción, al mantener la versión anterior en ejecución mientras realizamos nuestro lanzamiento.

reducimos en gran medida el tiempo de inactividad asociado con el lanzamiento de nuestro software. Según el mecanismo que se use para implementar la redirección del tráfico, el cambio entre versiones puede ser completamente invisible para el cliente, lo que nos brinda implementaciones sin tiempo de inactividad.

También hay otra técnica que vale la pena analizar brevemente aquí, que a veces se confunde con las implementaciones azul/verde, ya que puede usar algunas de las mismas implementaciones técnicas. Se le conoce como *liberación de canarios*.

Liberación de Canarias

Con la versión canary, estamos verificando nuestro software recién implementado dirigiendo cantidades de tráfico de producción contra el sistema para ver si funciona como se esperaba. "Rendimiento como se esperaba" puede cubrir una serie de cosas, tanto funcionales como no funcionales. Por ejemplo, podríamos verificar que un servicio recién implementado responda a las solicitudes dentro de los 500 ms, o que veamos las mismas tasas de error proporcionales del servicio nuevo y antiguo. Pero podrías ir más profundo que eso. Imagina que hemos lanzado una nueva versión del servicio de recomendación. Podríamos ejecutar ambos en paralelo, pero ver si las recomendaciones generadas por la nueva versión del servicio dan como resultado tantas ventas esperadas, asegurándonos de que no hemos lanzado un algoritmo subóptimo.

Si la nueva versión es mala, puedes revertirla rápidamente. Si es bueno, puede impulsar cantidades crecientes de tráfico a través de la nueva versión. El lanzamiento canario se diferencia del azul/verde en que puede esperar que las versiones coexistan durante más tiempo y, a menudo, variará la cantidad de tráfico.

Netflix utiliza este enfoque ampliamente. Antes del lanzamiento, las nuevas versiones del servicio se implementan junto con un clúster de referencia que representa la misma versión que la producción. Luego, Netflix ejecuta un subconjunto de la carga de producción durante varias horas contra la nueva versión y la línea de base, y califica ambas. Si el canario pasa, la empresa procede a un despliegue completo en producción.

Al considerar la liberación de Canary, debe decidir si va a desviar una parte de las solicitudes de producción al Canary o simplemente copiar la carga de producción. Algunos equipos pueden seguir el tráfico de producción y dirigirlo a su canario. De esta forma, las versiones Canary y de producción existentes pueden ver exactamente las mismas solicitudes, pero solo se ven externamente los resultados de las solicitudes de producción. Esto le permite hacer una comparación lado a lado mientras elimina la posibilidad de que una solicitud del cliente pueda ver una falla en el canario.

Sin embargo, el trabajo de emular el tráfico de producción puede ser complejo, especialmente si los eventos/solicitudes que se reproducen no son idempotentes.

El lanzamiento de Canary es una técnica poderosa y puede ayudarlo a verificar nuevas versiones de su software con tráfico real, al tiempo que le brinda herramientas para administrar el riesgo de lanzar un lanzamiento incorrecto. Sin embargo, requiere una configuración más compleja que la implementación azul/verde y un poco más de reflexión. Podría esperar coexistir diferentes versiones de sus servicios durante más tiempo que con azul/verde, por lo que es posible que esté inmovilizando más hardware durante más tiempo que antes. También necesitará un enrutamiento de tráfico más sofisticado, ya que es posible que desee aumentar o disminuir los porcentajes del tráfico para tener más confianza en que su versión funciona. Si ya maneja implementaciones azules/verdes, es posible que ya tenga algunos de los componentes básicos.

¿Tiempo medio de reparación sobre el tiempo medio entre fallas?

Entonces, al observar técnicas como la implementación azul/verde o la liberación canary, encontramos una manera de probar más cerca (o incluso en) la producción, y también creamos herramientas para ayudarnos a administrar una falla si ocurre. El uso de estos enfoques es un reconocimiento tácito de que no podemos detectar todos los problemas antes de lanzar nuestro software.

A veces, dedicar el mismo esfuerzo a mejorar la corrección de una versión puede ser significativamente más beneficioso que agregar más pruebas funcionales automatizadas. En el mundo de las operaciones web, esto a menudo se conoce como la compensación entre la optimización del *tiempo medio entre fallas (MTBF)* y el *tiempo medio de reparación (MTTR)*.

Las técnicas para reducir el tiempo de recuperación pueden ser tan simples como reversiones muy rápidas junto con un buen monitoreo (que discutiremos en el [Capítulo 8](#)), como implementaciones azul/verde. Si podemos detectar un problema en la producción temprano y retroceder temprano, reducimos el impacto para nuestros clientes. También podemos usar técnicas como la implementación azul/verde, donde implementamos una nueva versión de nuestro software y la probamos in situ antes de dirigir a nuestros usuarios a la nueva versión.

Para diferentes organizaciones, esta compensación entre MTBF y MTTR variará, y gran parte de esto depende de la comprensión del verdadero impacto de las fallas en un entorno de producción.

Sin embargo, la mayoría de las organizaciones que veo dedican tiempo a crear conjuntos de pruebas funcionales a menudo dedican poco o ningún esfuerzo a mejorar la supervisión o la recuperación de fallas. Por lo tanto, si bien pueden reducir la cantidad de defectos que ocurren en primer lugar, no pueden eliminarlos todos y no están preparados para lidiar con ellos si aparecen en la producción.

Existen compensaciones distintas de MTBF y MTTR. Por ejemplo, si está tratando de determinar si alguien realmente usará su software, puede tener mucho más sentido publicar algo ahora, para probar la idea o el modelo comercial antes de crear un software sólido. En un entorno en el que este sea el caso, las pruebas pueden ser excesivas, ya que el impacto de no saber si su idea funciona es mucho mayor que tener un defecto en la producción. En estas situaciones, puede ser bastante sensato evitar por completo las pruebas antes de la producción.

Pruebas de funciones cruzadas

La mayor parte de este capítulo se ha centrado en probar piezas específicas de funcionalidad y cómo esto difiere cuando está probando un sistema basado en microservicios. Sin embargo, hay otra categoría de pruebas que es importante discutir. *Los requisitos no funcionales* son un término genérico que se utiliza para describir las características que presenta su sistema y que no pueden implementarse simplemente como una característica normal. Incluyen aspectos como la latencia aceptable de una página web, la cantidad de usuarios que debe admitir un sistema, qué tan accesible debe ser su interfaz de usuario para las personas con discapacidades o qué tan seguros deben ser los datos de sus clientes.

El término *no funcional* nunca me cayó bien. ¡Algunas de las cosas que cubre este término parecen de naturaleza muy funcional! Una de mis colegas, Sarah Taraporewalla, acuñó la frase *requisitos multifuncionales (CFR, por sus siglas en inglés)*, que yo prefiero mucho. Habla más del hecho de que estos comportamientos del sistema realmente solo surgen como resultado de mucho trabajo transversal.

Muchos, si no la mayoría, de los CFR solo pueden cumplirse en la producción. Dicho esto, podemos definir estrategias de prueba para ayudarnos a ver si al menos estamos avanzando hacia el cumplimiento de estos objetivos. Este tipo de pruebas se incluyen en el cuadrante de *Pruebas de propiedades*. Un gran ejemplo de esto es la prueba de rendimiento, de la que hablaremos más a fondo en breve.

Para algunos CFR, es posible que desee realizar un seguimiento de ellos en un nivel de servicio individual. Por ejemplo, puede decidir que la durabilidad del servicio que requiere de su servicio de pago es significativamente mayor, pero está contento con más tiempo de inactividad para su servicio de recomendación de música, sabiendo que su negocio principal puede sobrevivir si no puede recomendar artistas similares a Metallica durante 10 minutos más o menos. Estas compensaciones terminarán teniendo un gran impacto en la forma en que diseña y desarrolla su sistema y, una vez más, la naturaleza detallada de un sistema basado en microservicios le brinda muchas más oportunidades de hacer estas compensaciones.

Las pruebas en torno a los CFR también deben seguir la pirámide. Algunas pruebas tendrán que ser de extremo a extremo, como las pruebas de carga, pero otras no. Por ejemplo, una vez que haya encontrado un cuello de botella de rendimiento en una prueba de carga de un extremo a otro, escriba una prueba de alcance más pequeño para ayudarlo a detectar el problema en el futuro. Otros CFR se ajustan a pruebas más rápidas con bastante facilidad. Recuerdo haber trabajado en un proyecto en el que habíamos insistido en asegurarnos de que nuestro marcado HTML usara las funciones de accesibilidad adecuadas para ayudar a las personas con discapacidades a usar nuestro sitio web. Verificar el marcado generado para asegurarse de que los controles apropiados se pudieran realizar muy rápidamente sin la necesidad de ningún viaje de ida y vuelta en la red.

Con demasiada frecuencia, las consideraciones sobre los CFR llegan demasiado tarde. Le sugiero encarecidamente que revise sus CFR lo antes posible y que los revise con regularidad.

Pruebas de rendimiento

Vale la pena mencionar explícitamente las pruebas de rendimiento como una forma de garantizar que se puedan cumplir algunos de nuestros requisitos multifuncionales. Al descomponer los sistemas en microservicios más pequeños, aumentamos la cantidad de llamadas que se realizarán a través de los límites de la red. Donde anteriormente una operación podría haber involucrado una llamada a la base de datos, ahora puede involucrar tres o cuatro llamadas a través de los límites de la red a otros servicios, con un número coincidente de llamadas a la base de datos. Todo esto puede disminuir la velocidad a la que operan nuestros sistemas. Rastrear las fuentes de latencia es especialmente importante. Cuando tiene una cadena de llamadas de múltiples llamadas sincrónicas, si alguna parte de la cadena comienza a actuar lentamente, todo se ve afectado, lo que puede generar un impacto significativo. Esto hace que tener alguna forma de probar el rendimiento de sus aplicaciones sea aún más importante de lo que podría ser con un sistema más monolítico. A menudo, la razón por la que este tipo de prueba se retrasa es porque inicialmente no hay suficiente sistema para probar. Entiendo este problema, pero con demasiada frecuencia conduce a patear la lata por el camino, y las pruebas de rendimiento a menudo solo se realizan justo antes de que se active por primera vez, ¡si es que se realizan! No caigas en esta trampa.

Al igual que con las pruebas funcionales, es posible que desee una combinación. Puede decidir que desea pruebas de rendimiento que aíslen los servicios individuales, pero comience con pruebas que verifiquen los recorridos principales en su sistema. Es posible que pueda realizar pruebas de viaje de extremo a extremo y simplemente ejecutarlas en volumen.

Para generar resultados que valgan la pena, a menudo necesitará ejecutar escenarios determinados con un número cada vez mayor de clientes simulados. Esto le permite ver cómo varía la latencia de las llamadas con el aumento de la carga. Esto significa que las pruebas de rendimiento pueden tardar un tiempo en ejecutarse. Además, querrá que el sistema coincida lo más posible con la producción, para asegurarse de que los resultados que vea sean indicativos del rendimiento que puede esperar en los sistemas de producción. Esto puede significar que tendrá que adquirir un volumen de datos más similar al de producción, y es posible que necesite más máquinas para adaptarse a la infraestructura, tareas que pueden ser desafiantes.

Incluso si tiene dificultades para hacer que el entorno de rendimiento sea realmente similar al de producción, las pruebas aún pueden tener valor para rastrear los cuellos de botella. Solo tenga en cuenta que puede obtener falsos negativos o, lo que es peor, falsos positivos.

Debido al tiempo que lleva ejecutar las pruebas de rendimiento, no siempre es factible ejecutarlas en cada registro.

Es una práctica común ejecutar un subconjunto todos los días y un conjunto más grande cada semana.

Independientemente del enfoque que elija, asegúrese de ejecutarlos con la mayor regularidad posible. Cuanto más tiempo pase sin ejecutar pruebas de rendimiento, más difícil será localizar al culpable. Los problemas de rendimiento son especialmente difíciles de resolver, por lo que si puede reducir la cantidad de confirmaciones que necesita para ver un problema recién introducido, su vida será mucho más fácil.

¡Y asegúrate de mirar también los resultados! Me ha sorprendido mucho la cantidad de equipos con los que me he encontrado que han dedicado mucho trabajo a implementar pruebas y ejecutarlas, y nunca verifican los números. A menudo, esto se debe a que las personas no saben cómo es un *buen* resultado. Realmente necesitas tener objetivos. De esta manera, puedes hacer que la compilación se vuelva *roja*.

o *verde* según los resultados, con una compilación roja (fallida) que es un claro llamado a la acción.

Las pruebas de rendimiento deben realizarse junto con la supervisión del rendimiento real del sistema (que trataremos más detalladamente en el **Capítulo 8**), e idealmente deberían usar las mismas herramientas en su entorno de prueba de rendimiento para visualizar el comportamiento del sistema que las que usa en producción. Esto puede hacer que sea mucho más fácil comparar cosas similares.

Resumen

Reuniendo todo esto, lo que he esbozado aquí es un enfoque holístico para las pruebas que, con suerte, le brinda una guía general sobre cómo proceder al probar sus propios sistemas. Para reiterar lo básico:

- Optimice para obtener comentarios rápidos y separe los tipos de pruebas en consecuencia.
- Evite la necesidad de pruebas de extremo a extremo siempre que sea posible mediante el uso de contratos impulsados por el consumidor.
- Utilice contratos impulsados por el consumidor para proporcionar puntos de enfoque para las conversaciones entre equipos
- Trate de comprender el equilibrio entre esforzarse más en las pruebas y detectar problemas más rápido en la producción (optimización para MTBF versus MTTR).

Si está interesado en leer más sobre las pruebas, le recomiendo *Pruebas ágiles* de Lisa Crispin y Janet Gregory (Addison-Wesley), que, entre otras cosas, cubre el uso del cuadrante de pruebas con más detalle.

Este capítulo se centró principalmente en asegurarnos de que nuestro código funcione antes de que llegue a la producción, pero también necesitamos saber cómo asegurarnos de que nuestro código funcione una vez que se implemente. En el próximo capítulo, veremos cómo monitorear nuestros sistemas basados en microservicios.

² Diane Vaughan, *La decisión de lanzamiento del Challenger: Tecnología arriesgada, cultura y Desviación en la NASA* (Chicago: University of Chicago Press, 1996).

Capítulo 8. Monitoreo

Como espero haber demostrado hasta ahora, dividir nuestro sistema en microservicios más pequeños y detallados genera múltiples beneficios. Sin embargo, también agrega complejidad cuando se trata de monitorear el sistema en producción. En este capítulo, veremos los desafíos asociados con el monitoreo y la identificación de problemas en nuestros sistemas de grano fino, y describiré algunas de las cosas que puede hacer para tener su pastel y comérselo también.

Imagina la escena. Es una tranquila tarde de viernes y el equipo está ansioso por irse temprano al pub como una forma de comenzar un fin de semana fuera del trabajo. Entonces, de repente, llegan los correos electrónicos. ¡El sitio web se está portando mal! Twitter está en llamas con las fallas de su empresa, su jefe le está mordiendo la oreja y las perspectivas de un fin de semana tranquilo se desvanecen.

¿Qué es lo primero que necesitas saber? ¿Qué diablos ha salido mal?

En el mundo de la aplicación monolítica, al menos tenemos un lugar muy obvio para comenzar nuestras investigaciones. ¿Sitio web lento? Es el monolito. ¿Sitio web dando errores extraños? Es el monolito. CPU al 100%? Monolito. ¿Olor a quemado? Bueno, ya captas la idea. ¡Tener un único punto de falla también simplifica un poco la investigación de fallas!

Ahora pensemos en nuestro propio sistema basado en microservicios. Las capacidades que ofrecemos a nuestros usuarios provienen de múltiples servicios pequeños, algunos de los cuales se comunican con más servicios para realizar sus tareas. Este enfoque tiene muchas ventajas (lo cual es bueno, ya que de lo contrario este libro sería una pérdida de tiempo), pero en el mundo del monitoreo, tenemos un problema más complejo entre manos.

Ahora tenemos múltiples servidores para monitorear, múltiples archivos de registro para filtrar y múltiples lugares donde la latencia de la red podría causar problemas. Entonces, ¿cómo abordamos esto? Necesitamos dar sentido a lo que, de otro modo, podría ser un lío caótico y enredado, lo último con lo que cualquiera de nosotros quiere lidiar un viernes por la tarde (o en cualquier momento, ¡vamos a eso!).

La respuesta aquí es bastante sencilla: supervise las cosas pequeñas y use la agregación para ver el panorama general. Para ver cómo, comenzaremos con el sistema más simple que podamos: un solo nodo.

Servicio único, servidor único

La Figura 8-1 presenta una configuración muy simple: un host, ejecutando un servicio. Ahora necesitamos monitorearlo para saber cuándo algo sale mal, para que podamos arreglarlo. Entonces, ¿qué debemos buscar?

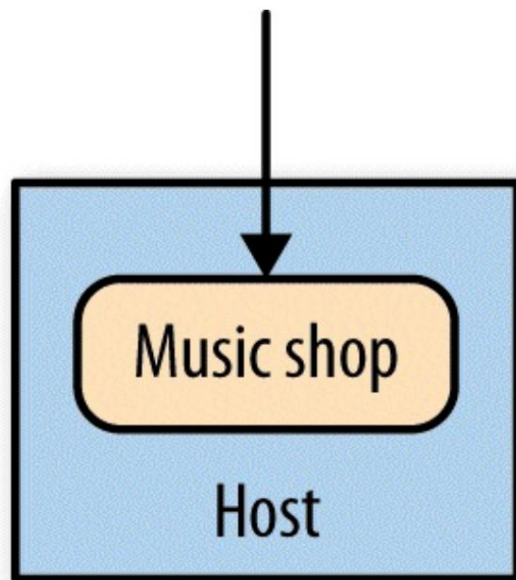


Figura 8-1. Un solo servicio en un solo host

Primero, queremos monitorear el propio host. CPU, memoria: todas estas cosas son útiles.

Querremos saber cuáles deberían ser cuando las cosas están saludables, para que podamos alertar cuando se salgan de los límites. Si queremos ejecutar nuestro propio software de monitoreo, podríamos usar algo como Nagios para hacerlo, o bien usar un servicio alojado como New Relic.

A continuación, queremos tener acceso a los registros desde el propio servidor. Si un usuario informa un error, estos registros deberían detectarlo y, con suerte, decirnos cuándo y dónde está el error. En este punto, con nuestro host único, probablemente podamos arreglárnoslas simplemente iniciando sesión en el host y usando herramientas de línea de comandos para escanear el registro. Incluso podemos avanzar y usar logrotate para quitar los registros antiguos y evitar que ocupen todo nuestro espacio en disco.

Finalmente, es posible que queramos monitorear la aplicación en sí. Como mínimo, monitorear el tiempo de respuesta del servicio es una buena idea. Probablemente podrá hacer esto mirando los registros que provienen de un servidor web frente a su servicio, o quizás del propio servicio. Si avanzamos mucho, es posible que deseemos realizar un seguimiento de la cantidad de errores que estamos informando.

El tiempo pasa, las cargas aumentan y nos encontramos con la necesidad de escalar...

Servicio único, servidores múltiples

Ahora tenemos múltiples copias del servicio ejecutándose en hosts separados, como se muestra en la **Figura 8-2**, con solicitudes a las diferentes instancias de servicio distribuidas a través de un balanceador de carga. Las cosas empiezan a ponerse un poco más complicadas ahora. Todavía queremos monitorear las mismas cosas que antes, pero debemos hacerlo de tal manera que podamos aislar el problema. Cuando la CPU es alta, ¿es un problema que estamos viendo en todos los hosts, lo que indicaría un problema con el servicio en sí, o está aislado en un solo host, lo que implica que el host mismo tiene el problema, tal vez un sistema operativo no autorizado? ¿proceso?

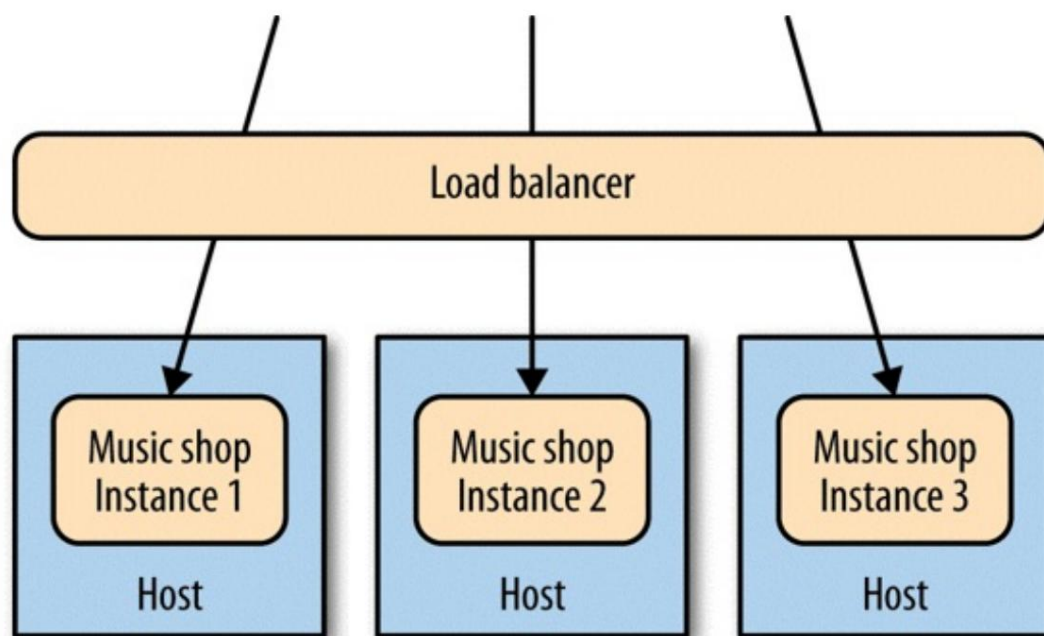


Figura 8-2. Un único servicio distribuido en múltiples hosts

Entonces, en este punto, todavía queremos rastrear las métricas a nivel de host y alertar sobre ellas. Pero ahora queremos ver cuáles son en todos los hosts, así como en hosts individuales. En otras palabras, queremos agregarlos y aún así poder profundizar. Nagios nos permite agrupar nuestros anfitriones de esta manera: hasta ahora, todo bien. Un enfoque similar probablemente sea suficiente para nuestra aplicación.

Entonces tenemos nuestros registros. Con nuestro servicio ejecutándose en más de un servidor, probablemente nos cansaremos de iniciar sesión en cada cuadro para verlo. Sin embargo, con solo unos pocos hosts, podemos usar herramientas como `ssh-multiplexers`, que nos permiten ejecutar los mismos comandos en múltiples hosts. Un gran monitor y un `grep "Error" app.log` más tarde, y podemos encontrar al culpable.

Para tareas como el seguimiento del tiempo de respuesta, podemos obtener parte de la agregación de forma gratuita mediante el seguimiento en el balanceador de carga. Pero también necesitamos rastrear el balanceador de carga, por supuesto; si eso se porta mal, tenemos un problema. En este punto, probablemente también nos importe mucho más cómo se ve un servicio en buen estado, ya que configuraremos nuestro balanceador de carga para eliminar los nodos en mal estado de nuestra aplicación. Esperemos que para cuando lleguemos aquí tengamos al menos una idea de eso...

Múltiples Servicios, Múltiples Servidores

En la **Figura 8-3**, las cosas se ponen mucho más interesantes. Múltiples servicios colaboran para brindar capacidades a nuestros usuarios, y esos servicios se ejecutan en múltiples hosts, ya sean físicos o virtuales. ¿Cómo encuentra el error que está buscando en miles de líneas de registros en múltiples hosts? ¿Cómo determina si un servidor se está portando mal o si se trata de un problema sistemático? ¿Y cómo rastrea un error que se encuentra en el fondo de una cadena de llamadas entre múltiples hosts y averigua qué lo causó?

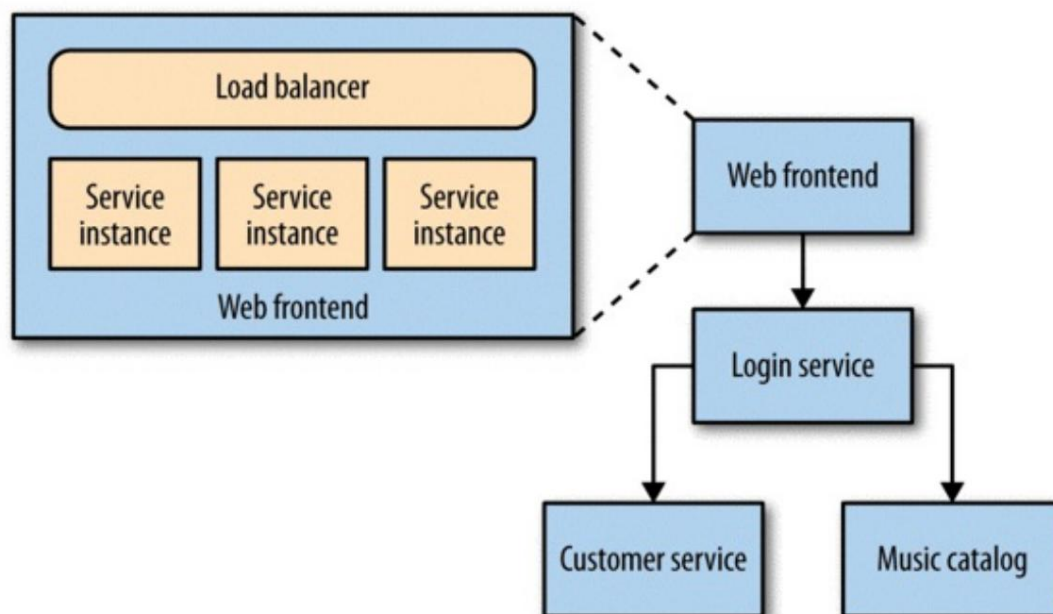


Figura 8-3. Múltiples servicios de colaboración distribuidos en múltiples hosts

La respuesta es la recopilación y la agregación central de todo lo que podamos tener en nuestras manos, desde registros hasta métricas de aplicaciones.

Registros, registros y aún más registros...

Ahora, la cantidad de hosts en los que nos estamos ejecutando se está convirtiendo en un desafío. La multiplexación SSH para recuperar registros probablemente no sea suficiente ahora, y no hay una pantalla lo suficientemente grande como para tener terminales abiertas en cada host. En su lugar, buscamos utilizar subsistemas especializados para obtener nuestros registros y ponerlos a disposición de forma centralizada. Un ejemplo de esto es **logstash**, que puede analizar múltiples formatos de archivo de registro y puede enviarlos a sistemas posteriores para una mayor investigación.

Kibana es un sistema respaldado por Elasticsearch para ver registros, ilustrado en la **Figura 8-4**. Puede usar una sintaxis de consulta para buscar en los registros, lo que le permite hacer cosas como restringir intervalos de fecha y hora o usar expresiones regulares para encontrar cadenas coincidentes. Kibana puede incluso generar gráficos a partir de los registros que le envías, lo que te permite ver de un vistazo cuántos errores se han generado a lo largo del tiempo, por ejemplo.

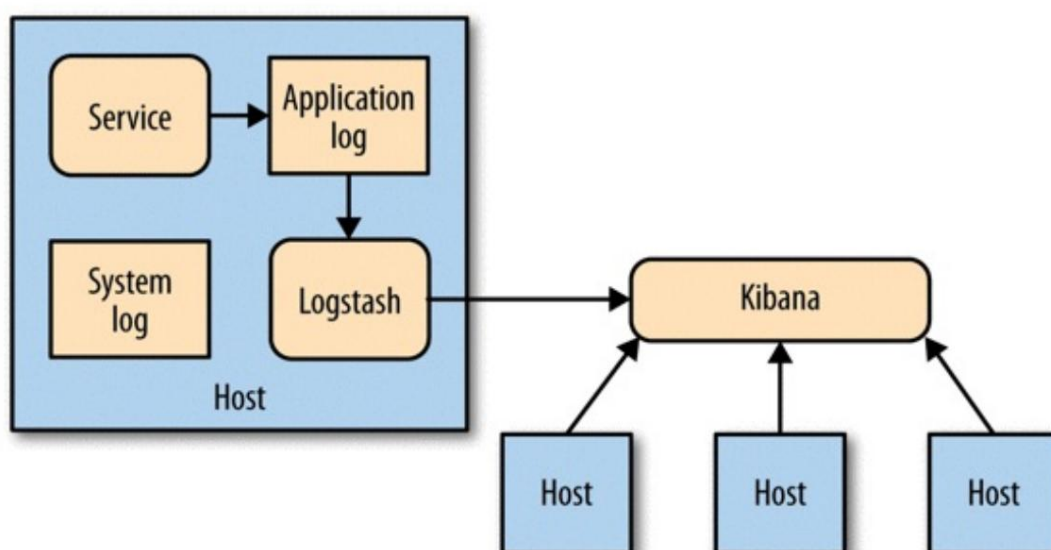


Figura 8-4. Uso de Kibana para ver registros agregados

Seguimiento de métricas en múltiples servicios

Al igual que con el desafío de buscar registros para diferentes hosts, debemos buscar mejores formas de recopilar y ver nuestras métricas. Puede ser difícil saber cómo se ve *bien* cuando estamos buscando métricas para un sistema más complejo. Nuestro sitio web está viendo casi 50 códigos de error HTTP 4XX por segundo. ¿Es tan malo? La carga de la CPU en el servicio de catálogo ha aumentado un 20 % desde el almuerzo; ha salido algo mal? El secreto para saber cuándo entrar en pánico y cuándo relajarse es recopilar métricas sobre cómo se comporta su sistema durante un período de tiempo lo suficientemente largo como para que surjan patrones claros.

En un entorno más complejo, proporcionaremos nuevas instancias de nuestros servicios con bastante frecuencia, por lo que queremos que el sistema que elijamos facilite la recopilación de métricas de nuevos hosts. Querremos poder ver una métrica agregada para todo el sistema, por ejemplo, la carga promedio de la CPU, pero también queremos agregar esa métrica para todas las instancias de un servicio dado, o incluso para una sola instancia de ese servicio. Eso significa que necesitaremos poder asociar metadatos con la métrica para permitirnos inferir esto.

estructura.

El grafito es uno de esos sistemas que lo hace muy fácil. Expone una API muy simple y le permite enviar métricas en tiempo real. Luego le permite consultar esas métricas para producir gráficos y otras pantallas para ver lo que está sucediendo. La forma en que maneja el volumen también es interesante. Efectivamente, lo configura para que reduzca la resolución de las métricas más antiguas para asegurarse de que los volúmenes no sean demasiado grandes. Entonces, por ejemplo, podría registrar la CPU para mis hosts una vez cada 10 segundos durante los últimos 10 minutos, luego una muestra agregada cada minuto durante el último día, hasta quizás una muestra cada 30 minutos durante los últimos años. De esta forma, puede almacenar información sobre cómo se ha comportado su sistema durante un largo período de tiempo sin necesidad de grandes cantidades de almacenamiento.

Graphite también le permite agregar muestras o profundizar en una sola serie, para que pueda ver el tiempo de respuesta de todo su sistema, un grupo de servicios o una sola instancia. Si Graphite no funciona para usted por cualquier motivo, asegúrese de obtener capacidades similares en cualquier otra herramienta que seleccione. Y, por supuesto, asegúrese de que puede obtener acceso a los datos sin procesar para proporcionar sus propios informes o paneles si es necesario.

Otro beneficio clave de comprender sus tendencias es cuando se trata de la planificación de la capacidad. ¿Estamos llegando a nuestro límite? ¿Cuánto tiempo hasta que necesitemos más anfitriones? En el pasado, cuando trajimos anfitriones físicos, este era a menudo un trabajo anual. En la nueva era de la informática bajo demanda proporcionada por los proveedores de infraestructura como servicio (IaaS), ahora podemos escalar hacia arriba o hacia abajo en minutos, si no segundos. Esto significa que si entendemos nuestros patrones de uso, podemos asegurarnos de que tenemos la infraestructura suficiente para satisfacer nuestras necesidades. Cuanto más inteligentes seamos en el seguimiento de nuestras tendencias y sepamos qué hacer con ellas, más rentables y receptivos pueden ser nuestros sistemas.

Métricas de servicio

Los sistemas operativos en los que nos ejecutamos generan una gran cantidad de métricas para nosotros, ya que encontrarás el momento en que instalas Collectd en una caja de Linux y apuntas a Graphite. Del mismo modo, los subsistemas compatibles como Nginx o Varnish exponen información útil, como los tiempos de respuesta o las tasas de aciertos de caché. Pero ¿qué pasa con su propio servicio?

Recomiendo encarecidamente que sus servicios expongan las métricas básicas por sí mismos. Como mínimo, para un servicio web, probablemente debería exponer métricas como los tiempos de respuesta y las tasas de error, vitales si su servidor no está al frente de un servidor web que está haciendo esto por usted.

Pero realmente debería ir más allá. Por ejemplo, nuestro servicio de cuentas puede querer exponer la cantidad de veces que los clientes ven sus pedidos anteriores, o su tienda web puede querer capturar cuánto dinero se ha ganado durante el último día.

¿Por qué nos importa esto? Bueno, por varias razones. Primero, hay un viejo adagio que dice que el 80% de las funciones del software nunca se utilizan. Ahora no puedo comentar qué tan precisa es esa cifra, pero como alguien que ha estado desarrollando software durante casi 20 años, sé que he dedicado mucho tiempo a funciones que en realidad nunca se usan. ¿No sería bueno saber cuáles son?

En segundo lugar, estamos mejorando más que nunca en la forma en que nuestros usuarios utilizan nuestro sistema para descubrir cómo mejorarlo. Las métricas que nos informan sobre cómo se comportan nuestros sistemas solo pueden ayudarnos aquí. Lanzamos una nueva versión del sitio web y descubrimos que la cantidad de búsquedas por género ha aumentado significativamente en el servicio de catálogo. ¿Es eso un problema, o esperado?

Finalmente, ¡nunca podemos saber qué datos serán útiles! Más veces de las que puedo contar, he querido capturar datos que me ayuden a comprender algo solo después de que la oportunidad de hacerlo haya pasado hace mucho tiempo. Tiendo a equivocarme al exponer todo y confiar en mi sistema de métricas para manejar esto más tarde.

Existen bibliotecas para varias plataformas diferentes que permiten que nuestros servicios envíen métricas a sistemas estándar. La **biblioteca Metrics** de Codahale es una de esas bibliotecas de ejemplo para la JVM.

Le permite almacenar métricas como contadores, temporizadores o indicadores; admite métricas de time-boxing (para que pueda especificar métricas como "número de pedidos en los últimos cinco minutos"); y también viene listo para usar con soporte para enviar datos a Graphite y otros sistemas de agregación y generación de informes.

Monitoreo Sintético

Podemos intentar averiguar si un servicio está en *buen* estado, por ejemplo, decidiendo cuál es un buen nivel de CPU o qué hace que el tiempo de respuesta sea aceptable. Si nuestro sistema de monitoreo detecta que los valores reales están fuera de este nivel seguro, podemos activar una alerta, algo de lo que una herramienta como Nagios es más que capaz.

Sin embargo, en muchos sentidos, estos valores están a un paso de lo que realmente queremos rastrear, es decir, *¿está funcionando el sistema?* Cuanto más complejas son las interacciones entre los servicios, más alejados estamos de responder esa pregunta. Entonces, ¿qué pasaría si nuestros sistemas de monitoreo estuvieran programados para actuar un poco como nuestros usuarios y pudieran informar si algo sale mal?

Hice esto por primera vez en 2005. Formaba parte de un pequeño equipo de ThoughtWorks que estaba construyendo un sistema para un banco de inversión. A lo largo del día de negociación, se produjeron muchos eventos que representaban cambios en el mercado. Nuestro trabajo era reaccionar a estos cambios y observar el impacto en la cartera del banco. Estábamos trabajando con unos plazos bastante ajustados, ya que el objetivo era haber hecho todos nuestros cálculos en menos de 10 segundos después de que llegara el evento. El sistema en sí constaba de alrededor de cinco servicios discretos, al menos uno de los cuales se ejecutaba en una red informática que, entre otras cosas, recuperaba los ciclos de CPU no utilizados en alrededor de 250 hosts de escritorio en el centro de recuperación de desastres del banco.

La cantidad de partes móviles en el sistema significaba que muchas de las métricas de nivel inferior que estábamos recopilando generaban mucho ruido. No tuvimos la ventaja de escalar gradualmente o hacer que el sistema funcionara durante unos meses para comprender cómo se veían *las* métricas como nuestra velocidad de CPU o incluso las latencias de algunos de los componentes individuales.

Nuestro enfoque fue generar eventos falsos para cotizar parte de la cartera que no estaba registrada en los sistemas posteriores. Aproximadamente cada minuto, Nagios ejecutaba un trabajo de línea de comandos que insertaba un evento falso en una de nuestras colas. Nuestro sistema lo recogió y ejecutó todos los cálculos como cualquier otro trabajo, excepto que los resultados aparecían en el libro *basura*, que se usaba solo para pruebas. Si no se veía un cambio de precio dentro de un tiempo determinado, Nagios lo informaba como un problema.

Este evento falso que creamos es un ejemplo de *transacción sintética*. Usamos esta transacción sintética para asegurarnos de que el sistema se comportaba semánticamente, razón por la cual esta técnica a menudo se denomina *monitoreo semántico*.

En la práctica, descubrí que el uso de transacciones sintéticas para realizar un monitoreo semántico como este es un indicador mucho mejor de los problemas en los sistemas que alertar sobre las métricas de nivel inferior. Sin embargo, no reemplazan la necesidad de las métricas de nivel inferior; aún queremos ese detalle cuando necesitemos averiguar por *qué* nuestro monitoreo semántico informa un problema.

Implementando el Monitoreo Semántico

Ahora, en el pasado, implementar el monitoreo semántico era una tarea bastante desalentadora. ¡Pero el mundo ha avanzado y los medios para hacerlo están al alcance de la mano! Estás ejecutando pruebas para tus sistemas, ¿verdad? Si no, vaya a leer el [Capítulo 7](#) y regrese. ¿Todo listo? ¡Bueno!

Si observamos las pruebas que tenemos que prueban un servicio determinado de principio a fin, o incluso todo nuestro sistema de principio a fin, tenemos mucho de lo que necesitamos para implementar el monitoreo semántico. Nuestro sistema ya expone los ganchos necesarios para lanzar la prueba y comprobar el resultado. Entonces, ¿por qué no ejecutar un subconjunto de estas pruebas, de manera continua, como una forma de monitorear nuestro sistema?

Hay algunas cosas que tenemos que hacer, por supuesto. Primero, debemos tener cuidado con los requisitos de datos de nuestras pruebas. Es posible que necesitemos encontrar una manera de que nuestras pruebas se adapten a diferentes datos en vivo si esto cambia con el tiempo, o bien establecer una fuente de datos diferente. Por ejemplo, podríamos tener un conjunto de usuarios falsos que usamos en producción con un conjunto de datos conocido.

Del mismo modo, debemos asegurarnos de no desencadenar accidentalmente efectos secundarios imprevistos. Un amigo me contó una historia sobre una empresa de comercio electrónico que accidentalmente realizó sus pruebas contra sus sistemas de pedidos de producción. No se dio cuenta de su error hasta que llegó una gran cantidad de lavadoras a la oficina central.

ID de correlación

Con una gran cantidad de servicios que interactúan para proporcionar cualquier capacidad de usuario final dada, una sola llamada de inicio puede terminar generando varias llamadas de servicio descendente más. Por ejemplo, considere el ejemplo de un cliente registrado. El cliente completa todos sus datos en un formulario y hace clic en enviar. Detrás de escena, verificamos la validez de los detalles de la tarjeta de crédito con nuestro servicio de pago, hablamos con nuestro servicio postal para enviar un paquete de bienvenida por correo y enviamos un correo electrónico de bienvenida utilizando nuestro servicio de correo electrónico. Ahora bien, ¿qué pasa si la llamada al servicio de pago acaba generando algún que otro error? Hablaremos extensamente sobre cómo manejar la falla en el **Capítulo 11**, pero considere la dificultad de diagnosticar lo que sucedió.

Si miramos los registros, el único servicio que registra un error es nuestro servicio de pago. Si tenemos suerte, podemos averiguar qué solicitud causó el problema e incluso podemos ver los parámetros de la llamada. Ahora considere que este es un ejemplo simple, y que una solicitud de inicio podría generar una cadena de llamadas descendentes y tal vez desencadenar eventos que se manejen de manera asincrónica. ¿Cómo podemos reconstruir el flujo de llamadas para reproducir y solucionar el problema? A menudo, lo que necesitamos es ver ese error en el contexto más amplio de la llamada de inicio; en otras palabras, nos gustaría rastrear la cadena de llamadas aguas arriba, tal como lo hacemos con un rastreo de pila.

Un enfoque que puede ser útil aquí es usar ID de correlación. Cuando se realiza la primera llamada, genera un GUID para la llamada. Esto luego se pasa a todas las llamadas subsiguientes, como se ve en la **Figura 8-5**, y se puede colocar en sus registros de una manera estructurada, como ya lo hará con componentes como el nivel de registro o la fecha. Con las herramientas adecuadas de agregación de registros, podrá rastrear ese evento en todo su sistema:

```
15-02-2014 16:01:01 Web-Frontend INFO [abc-123] Registrarse 15-02-2014
16:01:02 RegistrarServicio INFO [abc-123] Registrar Cliente...
15-02-2014 16:01:03 INFO del sistema postal [abc-123] Enviar paquete de bienvenida...
15-02-2014 16:01:03 EmailSystem INFO [abc-123] SendWelcomeEmail...
15-02-2014 16:01:03 ERROR de PaymentGateway [abc-123] ValidatePayment...
```

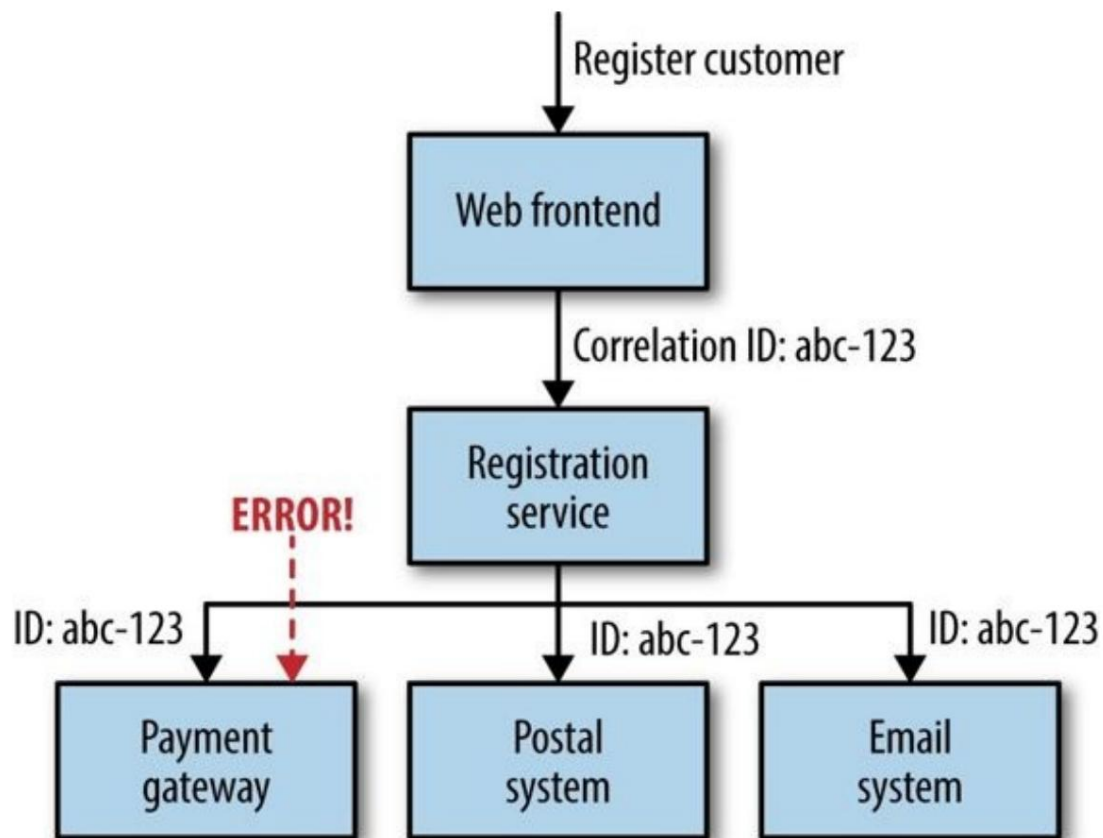


Figura 8-5. Uso de ID de correlación para rastrear cadenas de llamadas en múltiples servicios

Por supuesto, deberá asegurarse de que cada servicio sepa transmitir el ID de correlación. Aquí es donde necesita estandarizar y ser más fuerte para hacer cumplir esto en todo su sistema. Pero una vez que haya hecho esto, puede crear herramientas para rastrear todo tipo de interacciones. Dichas herramientas pueden ser útiles para rastrear tormentas de eventos, casos de esquinas extrañas o incluso identificar transacciones especialmente costosas, ya que puede imaginarse toda la cascada de llamadas.

Software como [Zipkin](#) también puede rastrear llamadas a través de múltiples límites del sistema. Basado en las ideas del propio sistema de seguimiento de Google, Dapper, Zipkin puede proporcionar un seguimiento muy detallado de las llamadas entre servicios, junto con una interfaz de usuario para ayudar a presentar los datos. Personalmente, he encontrado que los requisitos de Zipkin son algo pesados, requiriendo clientes personalizados y sistemas de cobro compatibles. Dado que ya deseará la agregación de registros para otros fines, parece mucho más sencillo utilizar los datos que ya está recopilando que tener que buscar fuentes de datos adicionales. Dicho esto, si encuentra que necesita una herramienta más avanzada para rastrear llamadas entre servicios como esta, es posible que desee echarles un vistazo.

Uno de los problemas reales con los ID de correlación es que a menudo no sabe que necesita uno hasta que ya tiene un problema que podría diagnosticarse solo si tuviera el ID desde el principio. Esto es especialmente problemático, ya que la actualización de ID de correlación es muy difícil; debe manejarlos de manera estandarizada para poder reconstituir fácilmente las cadenas de llamadas. Si bien puede parecer un trabajo adicional desde el principio, le sugiero encarecidamente que considere instalarlos tan pronto como pueda, especialmente si su sistema utilizará patrones de arquitectura de control de eventos, lo que puede conducir a un comportamiento emergente extraño.

La necesidad de manejar tareas como pasar consistentemente a través de ID de correlación puede ser un argumento sólido para el uso de bibliotecas contenedoras de clientes compartidos ligeros. A cierta escala, se vuelve difícil garantizar que todos llamen a los servicios posteriores de la manera correcta y recopilen el tipo correcto de datos. Solo se necesita un servicio en la mitad de la cadena para olvidarse de hacer esto y perder información crítica. Si decide crear una biblioteca de clientes interna para que cosas como esta funcionen de manera inmediata, asegúrese de mantenerla muy delgada y no atada a ningún servicio de producción en particular. Por ejemplo, si usa HTTP como protocolo subyacente para la comunicación, simplemente envuelva una biblioteca de cliente HTTP estándar y agregue código para asegurarse de que propaga los ID de correlación en los encabezados.

la cascada

Las fallas en cascada pueden ser especialmente peligrosas. Imagine una situación en la que se cae la conexión de red entre el sitio web de nuestra tienda de música y el servicio de catálogo. Los servicios en sí parecen estar en buen estado, pero no pueden comunicarse entre sí. Si solo observáramos la salud del servicio individual, no sabríamos que hay un problema. El uso de monitoreo sintético, por ejemplo, para imitar a un cliente que busca una canción, resolvería 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, monitorear los puntos de integración entre sistemas es clave. Cada instancia de servicio debe realizar un seguimiento y exponer el estado de sus dependencias posteriores, desde la base de datos hasta otros servicios de colaboración. También debe permitir que esta información se agregue para obtener una imagen acumulada. Querrá ver el tiempo de respuesta de las llamadas descendentes y también detectar si hay errores.

Como discutiremos más en el **Capítulo 11**, puede usar bibliotecas para implementar un interruptor de circuito alrededor de las llamadas de red para ayudarlo a manejar las fallas en cascada de una manera más elegante, lo que le permite degradar su sistema con más gracia. Algunas de estas bibliotecas, como Hystrix para JVM, también hacen un buen trabajo al proporcionarle estas capacidades de monitoreo.

Estandarización

Como hemos cubierto anteriormente, uno de los actos de equilibrio en curso que deberá lograr es dónde permitir que las decisiones se tomen de manera limitada para un solo servicio frente a dónde necesita estandarizar todo su sistema. En mi opinión, el monitoreo es un área donde la estandarización es increíblemente importante. Con los servicios que colaboran de muchas maneras diferentes para brindar capacidades a los usuarios que utilizan múltiples interfaces, debe ver el sistema de una manera holística.

Debe intentar escribir sus registros en un formato estándar. Definitivamente desea tener todas sus métricas en un solo lugar, y es posible que también desee tener una lista de nombres estándar para sus métricas; sería muy molesto para un servicio tener una métrica llamada `ResponseTime` y otro para tener una llamada `RspTimeSecs`, cuando significan lo mismo.

Como siempre con la estandarización, las herramientas pueden ayudar. Como dije antes, la clave es hacer que sea fácil hacer lo correcto, entonces, ¿por qué no proporcionar imágenes de máquinas virtuales preconfiguradas con logstash y recopilar listas para usar, junto con bibliotecas de aplicaciones que le permitan hablar con Graphite con mucha facilidad?

Considere la audiencia

Todos estos datos que recopilamos tienen un propósito. Más específicamente, estamos recopilando todos estos datos para diferentes personas para ayudarlos a hacer su trabajo; estos datos se convierten en una llamada a la acción.

Algunos de estos datos deben activar un llamado a la acción inmediato para nuestro equipo de soporte, por ejemplo, en el caso de que falle una de nuestras pruebas de monitoreo sintético. Otros datos, como el hecho de que nuestra carga de CPU haya aumentado un 2 % durante la última semana, solo pueden ser de interés cuando estamos planificando la capacidad. Del mismo modo, es probable que su jefe quiera 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" han aumentado un 5 % en la última hora.

Lo que nuestra gente quiere ver y reaccionar en *este momento* es diferente de lo que necesitan cuando profundizan. Entonces, para el tipo de persona que verá estos datos, considere lo siguiente:

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

Alerta sobre las cosas que necesitan saber en este momento. Cree grandes pantallas visibles con esta información que se sientan en la esquina de la habitación. Deles fácil acceso a los datos que necesitan saber más adelante. Y pasa tiempo con ellos para saber cómo quieren consumir datos. Una discusión sobre todos los matices involucrados en la visualización gráfica de información cuantitativa ciertamente está fuera del alcance de este libro, pero un buen 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 aisladas en diferentes sistemas.

Las métricas a nivel de aplicación, como la cantidad de pedidos realizados, terminan en un sistema de análisis patentado 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 mueren. .

Los informes de dichos sistemas a menudo no están disponibles en tiempo real, aunque eso está comenzando 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 pueden acceder los equipos de operaciones. Estos sistemas generalmente permiten informes en tiempo real, ya que normalmente el objetivo de ellos es provocar un llamado a la acción inmediato.

Históricamente, la idea de que podemos conocer las métricas comerciales clave uno o dos días después estuvo bien, ya que, de todos modos, generalmente no podíamos reaccionar lo suficientemente rápido a estos datos para hacer algo al respecto. Ahora, sin embargo, operamos en un mundo en el que muchos de nosotros podemos lanzar múltiples lanzamientos por día. Los equipos ahora no se miden a sí mismos en términos de cuántos *puntos* completan, sino que optimizan el tiempo que tarda el código en pasar de la computadora portátil a la versión activa. En tal entorno, necesitamos todas nuestras métricas al alcance de la mano para tomar la acción correcta. Irónicamente, los mismos sistemas que almacenan las métricas comerciales 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 descomponen en eventos que dicen que *algo sucedió en X*. Por lo tanto, si podemos unificar los sistemas que usamos para recopilar, agregar y almacenar estos eventos, y ponerlos a disposición para generar informes, terminaremos con mucho más. arquitectura más simple.

Riemann es un servidor de eventos que permite la agregación y el enrutamiento de eventos bastante avanzados y puede formar parte de dicha solución. **Suro** es el *conducto* de datos de Netflix y opera en un espacio similar. Suro se usa explícitamente para manejar métricas asociadas con el comportamiento del usuario y más datos operativos como registros de aplicaciones. Luego, estos datos se pueden enviar a una variedad de sistemas, como Storm para análisis en tiempo real, Hadoop para procesamiento por lotes fuera de línea o Kibana para análisis de registros.

Muchas organizaciones se están moviendo en una dirección fundamentalmente diferente: lejos de tener cadenas de herramientas especializadas para diferentes tipos de métricas y hacia sistemas de enrutamiento de eventos más genéricos capaces de una escala significativa. Estos sistemas logran proporcionar mucha más flexibilidad y, al mismo tiempo, simplifican nuestra arquitectura.

Resumen

Entonces, ¡hemos cubierto mucho aquí! Intentaré resumir este capítulo en algunos consejos fáciles de seguir.

Para cada servicio:

- Realice un seguimiento del tiempo de respuesta entrante como mínimo. Una vez que haya hecho eso, siga 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, como mínimo, incluido el tiempo de respuesta de las llamadas posteriores y, en el mejor de los casos, las tasas de error de seguimiento. Las bibliotecas como Hystrix pueden ayudar aquí.
- Estandarice cómo y dónde se recopilan las métricas.
- Inicie sesión en una ubicación estándar, en un formato estándar si es posible. ¡La agregación es una molestia si cada servicio usa un diseño diferente!
- Supervise el sistema operativo subyacente para que pueda rastrear los 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 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.
- Tenga una única herramienta consultable para agregar y almacenar registros.
- Considere seriamente estandarizar el uso de ID de correlación.
- Comprenda qué requiere una llamada a la acción y estructure las alertas y los paneles en consecuencia.
- Investigue la posibilidad de unificar cómo agrega todas sus diversas métricas al ver si una herramienta como Suro o Riemann tiene sentido para usted.

También intenté delinear la dirección en la que se está moviendo el monitoreo: lejos de los sistemas especializados para hacer una sola cosa, y hacia sistemas genéricos de procesamiento de eventos 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 haberte dado suficiente para empezar. Si quieres saber más, entro en algunas de estas ideas

y más en mi publicación anterior *Sistemas ligeros para monitoreo en tiempo real* (O'Reilly).

En el próximo capítulo, tomaremos una visión holística diferente de nuestros sistemas para considerar algunas de las ventajas y desafíos únicos que las arquitecturas detalladas pueden proporcionar en el área de la 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 estén expuestos a todo tipo de personajes dudosos. 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 el valor de los datos que tenemos para nuestros clientes en los sistemas que construimos. Este capítulo brindará una breve descripción general de algunos aspectos de la seguridad que debe tener en cuenta al diseñar sus sistemas. Si bien no pretende ser exhaustivo, presentará algunas de las principales opciones disponibles para usted y le brindará un punto de partida para su propia investigación.

Necesitamos pensar qué protección necesitan nuestros datos mientras están en tránsito de un punto a otro, y qué protección necesitan en reposo. Necesitamos pensar en la seguridad de nuestros sistemas operativos subyacentes y también en nuestras redes. ¡Hay tanto en lo que 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.

Autenticacion y autorizacion

La autenticación y la autorización son conceptos centrales 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. Para un ser humano, normalmente 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 quien 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. Generalmente, 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 mapeamos de un director a la acción que le estamos permitiendo hacer. A menudo, cuando se autentica a un director, se nos dará información sobre él que nos ayudará a decidir qué debemos dejar que haga. Por ejemplo, es posible que nos digan en qué departamento u oficina trabaja, información que nuestros sistemas pueden usar para decidir qué puede y qué no puede hacer.

Para aplicaciones únicas y monolíticas, es común que la propia aplicación maneje la autenticación y la autorización por usted. Django, el marco web de Python, viene listo para usar con administración de usuarios, por ejemplo. Sin embargo, cuando se trata de sistemas distribuidos, debemos pensar en esquemas más avanzados. No queremos que todos tengan que iniciar sesión por separado para 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 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 espacio empresarial, y OpenID Connect brindan capacidades en esta área. Más o menos utilizan 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 web), se le indica que se autentique con un *proveedor de identidad*. Esto puede pedirle que proporcione un nombre de usuario y una contraseña, o puede usar algo más avanzado como la autenticación de dos factores. Una vez que el proveedor de identidad está satisfecho de que el principal ha sido autenticado, proporciona información al *proveedor de servicios*, lo que le permite decidir si le otorga acceso al recurso.

Este proveedor de identidad podría ser un sistema alojado externamente o algo dentro de su propia organización. Google, por ejemplo, proporciona un proveedor de identidad de 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 podría ser algo así como el Protocolo ligero de acceso a directorios (LDAP) o Active Directory. Estos sistemas le permiten almacenar información sobre los directores, como las funciones que desempeñan en la organización.

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

SAML es un estándar basado en SOAP y es conocido por ser bastante complejo para trabajar a pesar de las bibliotecas y las herramientas disponibles para admitirlo. 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 SSO. Utiliza llamadas REST más simples y, en mi opinión, es probable que haga incursiones en las empresas debido a su facilidad de uso mejorada. Su mayor obstáculo en este momento es la falta de proveedores de identidad que lo respalden. Para un sitio web público, es posible que esté de acuerdo con Google como su proveedor, pero para sistemas internos o sistemas en los que desea más control y visibilidad sobre cómo y dónde se instalan sus datos, querrá su propia identidad interna. proveedor. Al 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 ya menos que los proveedores de identidad existentes comiencen a admitir OpenID Connect, su crecimiento puede estar limitado a aquellas situaciones en las que las personas están felices de usar un proveedor de identidad pública.

Entonces, aunque creo que OpenID Connect es el futuro, es muy posible que lleve un tiempo lograr una adopción generalizada.

Puerta de enlace de inicio de sesión único

Dentro de una configuración de microservicio, cada servicio podría decidir manejar la redirección y el protocolo de 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 provenir del código compartido. Esto tampoco ayudaría si tuviera varias pilas de tecnología diferentes.

En lugar de que cada servicio administre el protocolo de enlace con su proveedor de identidad, puede usar una puerta de enlace para que actúe como un proxy, sentado 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 apretón de manos en un solo lugar.

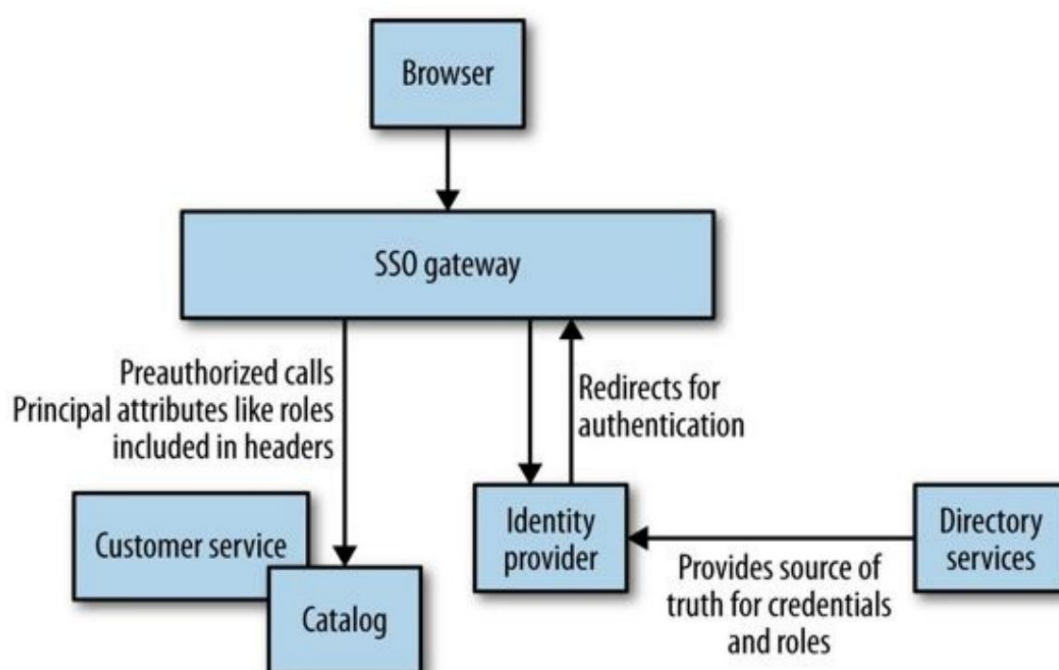


Figura 9-1. Uso de una puerta de enlace para manejar SSO

Sin embargo, todavía tenemos que resolver el problema de cómo el servicio descendente 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 usa con Apache con gran efecto para manejar 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 ser más difícil razonar sobre cómo se comporta un microservicio cuando se mira de forma aislada. ¿Recuerda en el [Capítulo 7](#) donde exploramos algunos de los desafíos en la reproducción de entornos similares a los de producción? Si elige la ruta de la puerta de enlace, asegúrese de que sus desarrolladores puedan lanzar sus servicios detrás de uno sin demasiado trabajo.

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

máquina, al sistema operativo, al hardware subyacente. Tiene la capacidad de implementar medidas de seguridad en todos estos puntos, algunos de los cuales abordaremos en breve. He visto a algunas personas poner todos sus huevos en una canasta, confiando en que la puerta de enlace manejará cada paso por ellos. Y todos sabemos lo que sucede cuando tenemos un único punto de falla...

Obviamente, podría usar esta puerta de enlace para hacer otras cosas. Si usa una capa de instancias de Apache que ejecutan Shibboleth, por ejemplo, también podría decidir finalizar HTTPS en este nivel, ejecutar la detección de intrusos, etc. Eso sí, ten cuidado. Las capas de puerta de enlace tienden a adquirir más y más funciones, lo que en sí mismo puede terminar siendo un punto de acoplamiento gigante. Y cuanto más funcionalidad tiene algo, mayor es la superficie de ataque.

Autorización detallada

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 pueda extraer atributos sobre el principal como resultado de la autenticación, es posible que pueda tomar decisiones más matizadas. Por ejemplo, es común ubicar a las personas en grupos o asignarles roles. Podemos usar esta información para comprender lo que pueden hacer. Entonces, para la aplicación de la mesa de ayuda, podríamos permitir el acceso solo a los principales con un rol específico (por ejemplo, PERSONAL). Sin embargo, más allá de permitir (o prohibir) el acceso a recursos o puntos finales específicos, debemos dejar el resto en manos del propio microservicio; tendrá que tomar más decisiones sobre qué operaciones permitir.

Volvamos a nuestra aplicación de soporte técnico: ¿permitimos que algún miembro del personal vea todos los detalles? Lo más probable es que tengamos diferentes roles en el trabajo. Por ejemplo, a un principal en el grupo CALL_CENTER se le puede permitir ver cualquier información sobre un cliente excepto sus detalles de pago. El director también podría emitir reembolsos, pero esa cantidad podría tener un tope. 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 maneras horribles, usando roles muy detallados como CALL_CENTER_50_DOLLAR_REFUND, donde terminan poniendo información específica de una parte del comportamiento de nuestro sistema 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 vive en otro lugar, tal vez en un sistema administrado por una parte diferente de la organización.

En su lugar, favorezca los roles de granularidad gruesa, modelados en torno a cómo funciona su organización. Volviendo a los primeros capítulos, recuerde que estamos creando software para que coincida con el funcionamiento de nuestra organización. Así que usa tus roles de esta manera también.

Autenticación y autorización de servicio a servicio

Hasta este punto, hemos estado usando el término *principal* para describir cualquier cosa que pueda autenticar y estar autorizado para hacer cosas, pero nuestros ejemplos en realidad han sido sobre humanos que usan 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 estar bien. Algunas organizaciones intentan garantizar la seguridad en el perímetro de sus redes y, por lo tanto, asumen 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 típico ataque *de intermediario*. Si el atacante decide interceptar y leer los datos que se envían, cambiar los datos sin que usted lo sepa o incluso, en algunas circunstancias, fingir ser la cosa con la que está hablando, es posible que no sepa mucho al respecto.

Esta es, con mucho, 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! Para la mayoría de las organizaciones que veo que usan este modelo, me preocupa que el modelo de confianza implícita no sea una decisión consciente, sino más bien que las personas desconozcan 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 verificar estos detalles y confirmar que el cliente puede acceder al servicio. La ventaja aquí es que este es un protocolo extremadamente bien entendido y bien respaldado. El problema es que hacer esto a través de HTTP es muy problemático, ya que el nombre de usuario y la contraseña no se envían de forma segura. Cualquier parte intermediaria puede mirar la información en el encabezado y ver los datos. Por lo tanto, la autenticación básica HTTP normalmente se debe usar sobre HTTPS.

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

El servidor necesita administrar sus propios certificados SSL, lo que puede resultar problemático cuando administra varias máquinas. Algunas organizaciones asumen su propio proceso de emisión de certificados, lo que supone una carga administrativa y operativa adicional. Las herramientas para administrar esto de manera automatizada no son tan maduras como podrían ser, y no es solo el proceso de emisión lo que debe manejar. Los certificados autofirmados no son fácilmente revocables y, por lo tanto, requieren mucha más reflexión sobre los escenarios de desastre. Vea si puede esquivar todo este trabajo evitando la firma automática por completo.

Otro inconveniente es que el tráfico enviado a través de SSL no puede almacenarse en caché mediante proxies inversos como Varnish o Squid. Esto significa que si necesita almacenar en caché el tráfico, tendrá que hacerlo dentro del servidor o dentro del cliente. Puede solucionar esto haciendo que un balanceador de carga finalice el tráfico SSL y que el caché se asiente 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, permitiéndonos un proceso para emitirlas y revocarlas? Podríamos hacer esto haciendo que el servicio se comuniqué con el mismo servicio de directorio que respalda nuestra solución SSO. Alternativamente, podríamos almacenar los nombres de usuario y las contraseñas dentro del servicio, pero luego corremos el riesgo de duplicar el comportamiento.

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

Usar SAML u OpenID Connect

Si ya usa SAML u OpenID Connect como su esquema de autenticación y autorización, también puede usarlo para interacciones de servicio a servicio. Si está utilizando una puerta de enlace, también deberá enrutar todo el tráfico de la red a través de la puerta de enlace, pero si cada servicio está manejando 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 sus servicios en un servidor de directorio central. Todavía necesitaríamos enrutar esto a través de HTTPS si quisiéramos evitar ataques de intermediarios.

Los clientes tienen un conjunto de credenciales que usan 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, trate de limitar su uso. Así que considere que cada microservicio tenga su propio conjunto de credenciales. Esto facilita la revocación/cambio de acceso si las credenciales se ven comprometidas, ya que solo necesita revocar el conjunto de credenciales que se han visto afectadas.

Sin embargo, hay un par de otras desventajas. En primer lugar, al igual que con la Autenticación básica, debemos almacenar de forma segura nuestras credenciales: ¿dónde se encuentran el nombre de usuario y la contraseña? El cliente deberá encontrar alguna 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 un asunto doloroso. OpenID Connect tiene un flujo de trabajo más simple, pero como comentamos anteriormente, todavía no es tan compatible.

Certificados de clientes

Otro enfoque para confirmar la identidad de un cliente es hacer uso de las capacidades de Transport Layer Security (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, brindando fuertes garantías de que el cliente es válido.

Los desafíos operativos aquí 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 crear y administrar una mayor cantidad de certificados; más bien, es que con todas las complejidades en torno a los certificados en sí, puede esperar pasar mucho tiempo tratando de diagnosticar por qué un servicio no acepta lo que cree que es un certificado de cliente completamente válido. Y luego tenemos que considerar la dificultad de revocar y volver a emitir certificados en caso de que suceda lo peor. El uso de certificados comodín puede ayudar, pero no resolverá todos los problemas. Esta carga adicional significa que buscará utilizar 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 las partes que se envían a través de Internet, por ejemplo.

HMAC sobre HTTP

Como discutimos anteriormente, el uso de la Autenticación básica sobre HTTP simple no es muy sensato si nos preocupa que el nombre de usuario y la contraseña se vean comprometidos. La alternativa tradicional es enrutar el tráfico HTTPS, pero tiene algunas desventajas. Además de administrar los certificados, la sobrecarga del tráfico HTTPS puede ejercer una presión adicional sobre los servidores (aunque, para ser honesto, esto tiene un impacto menor que hace varios años), y el tráfico no se puede almacenar en caché fácilmente.

Un enfoque alternativo, como el que utilizan ampliamente 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, la solicitud del cuerpo junto con una clave privada se codifica y el hash resultante se envía junto con la solicitud. Luego, el servidor usa su propia copia de la clave privada y el cuerpo de la solicitud para volver a crear el hash. Si coincide, permite la solicitud. Lo bueno aquí es que si un hombre en el medio se mete con la solicitud, entonces el hash no coincidirá y el servidor sabrá que la solicitud ha sido manipulada. ¡Y la clave privada nunca se envía en la solicitud, por lo que no puede verse comprometida 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 manejar el tráfico HTTPS (aunque su kilometraje puede variar).

Hay tres desventajas en este enfoque. Primero, 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 luego tiene el problema de revocar el acceso si el secreto se ve comprometido. Si comunica esta clave a través de algún protocolo alternativo, ¡debe asegurarse de que ese protocolo también sea muy seguro!

En segundo lugar, se trata de un patrón, no de un estándar, por lo que existen formas divergentes de implementarlo. Como resultado, hay 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 hace. Iría tan lejos como para decir que solo mire cómo Amazon hace esto para S3 y copie su enfoque, especialmente usando una función hash sensible con una clave adecuadamente larga como SHA-256. También vale la pena considerar **los tokens web JSON (JWT)**, ya que implementan un enfoque muy similar y parecen estar ganando terreno. Pero tenga en cuenta la dificultad de hacerlo bien. Mi colega estaba trabajando con un equipo que estaba implementando su propia implementación de JWT, omitió una sola verificación booleana e invalidó todo su código de autenticación. Con suerte, con el tiempo veremos más implementaciones de bibliotecas reutilizables.

Finalmente, comprenda que este enfoque solo garantiza que ningún tercero haya manipulado la solicitud y que la clave privada en sí misma permanezca privada. El resto de los datos de la solicitud seguirán siendo visibles para las partes que husmean en la red.

Claves API

Todas las API públicas de servicios como Twitter, Google, Flickr y AWS utilizan claves de API. Las claves API permiten que un servicio identifique quién está realizando una llamada y establezca límites sobre lo que pueden 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 específicas para proteger la calidad del servicio para otras personas.

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

Algunos sistemas usan una sola clave API que se comparte y usan un enfoque similar al HMAC como se acaba de describir. Un enfoque más común es utilizar un par de claves pública y privada.

Por lo general, administrará las claves de forma centralizada, tal como administraríamos las identidades de las personas de forma centralizada. El modelo pasarela es muy popular en este espacio.

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

Las capacidades exactas de los sistemas varían y tiene múltiples opciones tanto en el espacio comercial como en el de código abierto. Algunos de los productos solo manejan el intercambio de claves de la API y alguna gestión básica de claves. Otras herramientas ofrecen todo, incluido el límite de velocidad, la monetización, los catálogos de API y los sistemas de descubrimiento.

Algunos sistemas de API le permiten vincular las claves de API con los servicios de directorio existentes. Esto le permitiría emitir claves API a directores (que representan personas o sistemas) en 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, usando SAML para autenticar humanos para SSO y usando claves API para la comunicación de servicio a servicio, como se muestra en la **Figura 9. -2.**

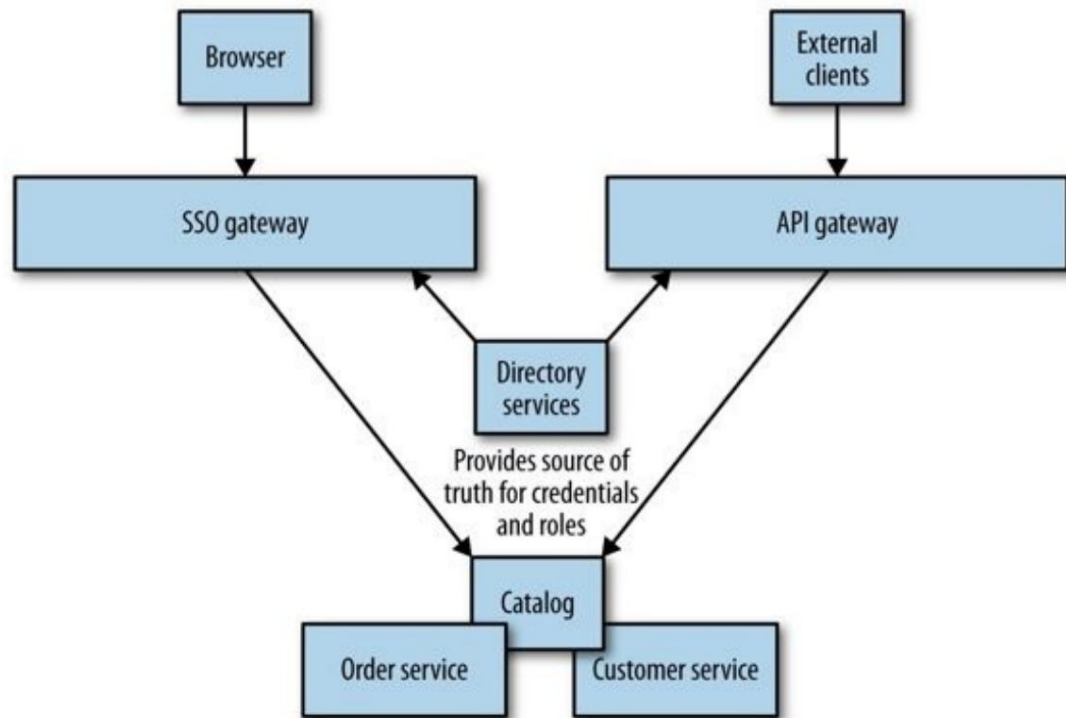


Figura 9-2. Uso de servicios de directorio para sincronizar información principal entre un SSO y una puerta de enlace API

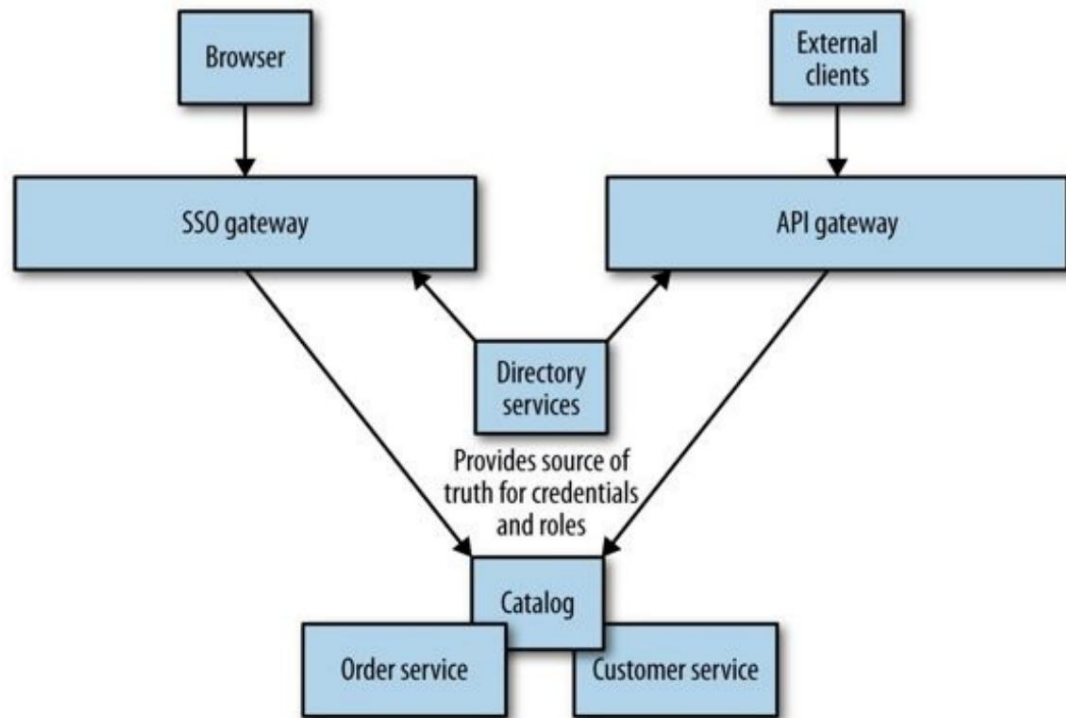


Figura 9-2. Uso de servicios de directorio para sincronizar información principal entre un SSO y una puerta de enlace API

El problema del diputado

Tener un principal autenticado con un microservicio dado es bastante simple. Pero, ¿qué sucede si ese servicio luego necesita realizar llamadas adicionales para completar una operación? Mire 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. Hace llamadas a una aplicación de tienda del lado del servidor, usando el patrón backends-for-frontends que describimos en el [Capítulo 4](#). Las llamadas realizadas entre el navegador y las llamadas del servidor se pueden autenticar usando SAML u OpenID Connect o similar.

Hasta aquí todo bien.

Cuando estoy conectado, puedo hacer clic en un enlace para ver los detalles de un pedido. Para mostrar la información, debemos retirar el pedido original del servicio de pedidos, pero también queremos buscar la información de envío del pedido. Por lo tanto, 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 al servicio de pedidos y 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 vino desde dentro de nuestro perímetro, está bien. Incluso podríamos usar certificados o claves API para confirmar que sí, que realmente es la tienda online la que pide esta información. ¿Pero es esto suficiente?

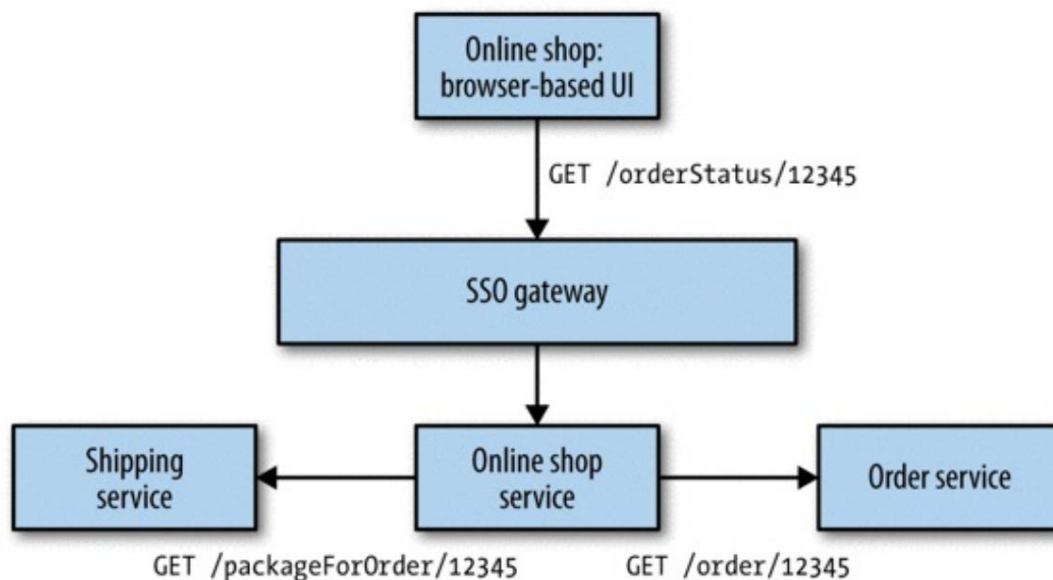


Figura 9-3. Un ejemplo en el que un diputado confundido podría entrar en juego

Existe un tipo de vulnerabilidad denominada *problema del adjunto confuso*, que en el contexto de la comunicación de servicio a servicio se refiere a una situación en la que una parte malintencionada puede engañar a un servicio adjunto para que haga llamadas a un servicio descendente en su nombre que no debería. no poder. Por ejemplo, como cliente, cuando inicio sesión en el sistema de compras en línea, puedo ver los detalles de mi cuenta. ¿Qué pasaría 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 órdenes que no son mías? Una vez que haya iniciado sesión, podría comenzar a enviar solicitudes de otros pedidos que no son míos para ver si puedo obtener

información útil. Podríamos intentar protegernos de esto dentro de la propia tienda online, comprobando para quién es el pedido y rechazándolo si alguien pide cosas que no debe.

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

Podríamos enrutar las solicitudes directamente desde la interfaz de usuario al servicio de pedidos y permitir que valide la solicitud, pero luego nos encontramos con las diversas desventajas que discutimos en el **Capítulo 4**.

Alternativamente, cuando la tienda en línea 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 director original en sentido descendente, aunque con SAML esto es un poco una pesadilla, ya que involucra aserciones de SAML anidadas que son técnicamente alcanzables, pero tan difíciles que nadie hace esto. Esto puede volverse aún más complejo, por supuesto. Imagínese si los servicios con los que habla la tienda en línea a su vez hacen más llamadas descendentes. ¿Hasta dónde tenemos que llegar para validar la confianza de todos esos diputados?

Este problema, desafortunadamente, no tiene una respuesta simple, porque no es un problema simple. Sin embargo, tenga en cuenta que existe. Dependiendo de la confidencialidad de la operación en cuestión, es posible que deba elegir entre la confianza implícita, verificar la identidad de la persona que llama o pedirle a la persona que llama que proporcione las credenciales del principal original.

Protección de datos en reposo

La protección de datos es una responsabilidad, especialmente si es confidencial. Esperamos que hayamos hecho todo lo posible para asegurarnos de que los atacantes no puedan violar nuestra red, y también que no puedan violar nuestras aplicaciones o sistemas operativos para obtener acceso al primer plano subyacente. Sin embargo, debemos estar preparados en caso de que lo hagan: la defensa en profundidad es clave.

Muchas de las infracciones de seguridad de alto perfil implican que un atacante adquiere datos en reposo y que el atacante puede leer esos datos. Esto se debe a que los datos se almacenaron sin cifrar o a que el mecanismo utilizado para proteger los datos tenía un defecto fundamental.

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

Ir con el bien conocido

La forma más fácil de estropear el cifrado de datos es intentar implementar sus propios algoritmos de cifrado, o incluso intentar implementar los de otra persona. Independientemente del lenguaje de programación que utilice, tendrá acceso a implementaciones revisadas y parcheadas regularmente de algoritmos de cifrado bien considerados. ¡Usa esos! Y suscríbase a las listas de correo/listas de avisos para la tecnología que elija para asegurarse de estar al tanto de las vulnerabilidades a medida que se encuentran, 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** Ambos eligen una implementación conocida de AES-128 o AES-256 para su plataforma.

Los tiempos 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 separadas para la mayoría de las plataformas, por ejemplo, las **bibliotecas Bouncy Castle** para Java y C#.

Para las contraseñas, debe considerar el uso de una técnica llamada *hash de contraseña con sal*.

El cifrado mal implementado podría ser peor que no tener ninguno, ya que la falsa sensación de seguridad (perdón por el juego de palabras) puede llevarlo a perder la vista.

Todo se trata de las llaves

Como se ha cubierto hasta ahora, el cifrado se basa en un algoritmo que toma los datos que se van a cifrar y una clave y luego produce los datos cifrados. Entonces, ¿dónde está almacenada su clave? Ahora bien, si estoy cifrando mis datos porque me preocupa que alguien robe toda mi base de datos, y almaceno la clave que uso en la misma base de datos, ¡entonces realmente no he logrado mucho! Por lo tanto, necesitamos guardar las llaves en otro lugar. ¿Pero donde?

Una solución es utilizar un dispositivo de seguridad independiente para cifrar y descifrar datos. Otra es usar un almacén de claves separado 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 manejar esto por usted.

Algunas bases de datos incluso incluyen soporte incorporado para el cifrado, como el Cifrado de datos transparente de SQL Server, cuyo objetivo es manejar esto de manera transparente. Incluso si su base de datos de elección lo hace, investigue cómo se manejan las claves y comprenda si la amenaza contra la que está protegiéndose realmente se está mitigando.

Una vez más, esto es complejo. ¡Evite implementar el suyo propio y haga una buena investigación!

Elige tus objetivos

Asumir que todo debe estar encriptado puede simplificar un poco las cosas. No hay conjeturas sobre lo que debe o no debe protegerse. Sin embargo, aún deberá pensar en qué datos se pueden colocar en los archivos de registro para ayudar a identificar problemas, y la sobrecarga computacional de cifrar todo puede volverse bastante onerosa y, como resultado, necesita un hardware más potente. Esto es aún más desafiante cuando aplica 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.

Al subdividir su sistema en servicios más detallados, puede identificar un almacén de datos completo que se puede cifrar al por mayor, pero incluso así es poco probable. Limitar este cifrado a un conjunto conocido de tablas es un enfoque sensato.

Descifrar bajo demanda

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

Cifrar copias de seguridad

Las copias de seguridad son buenas. Queremos hacer una copia de seguridad de nuestros datos importantes, y casi por definición, los datos que nos preocupan lo suficiente como para cifrarlos son lo suficientemente importantes como para hacer una copia de seguridad. Por lo tanto, puede parecer un punto obvio, pero debemos asegurarnos de que *nuestras copias de seguridad también estén encriptadas*. Esto también significa que necesitamos saber qué claves se necesitan para manejar qué versión de datos, especialmente si las claves cambian. Tener una gestión de claves clara se vuelve bastante importante.

Defensa en profundidad

Como mencioné anteriormente, no me gusta poner todos los huevos en una canasta. Se trata de una defensa en profundidad. Ya hemos hablado sobre la seguridad de los datos en tránsito y la seguridad de los datos en reposo. Pero, ¿hay otras protecciones que podamos implementar para ayudar?

cortafuegos

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

Es valioso tener más de un firewall. Por ejemplo, puede decidir usar IPTables localmente en un host para asegurar ese host, configurando la entrada y la salida permitidas.

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

Inicio sesión

Un buen registro, y específicamente la capacidad de agregar registros de múltiples sistemas, no se trata de prevención, sino que puede ayudar a detectar y recuperarse de cosas malas que suceden.

Por ejemplo, después de aplicar parches de seguridad, a menudo puede ver en los registros si las personas han estado explotando ciertas vulnerabilidades. La aplicación de parches asegura que no vuelva a suceder, pero si ya *sucedio*, es posible que deba ingresar 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.

La información confidencial debe seleccionarse para garantizar que no estemos filtrando datos importantes en nuestros registros, lo que podría terminar siendo un gran objetivo para los atacantes.

Sistema de detección (y prevención) de intrusos

Los sistemas de detección de intrusos (IDS) pueden monitorear redes o hosts en busca de comportamientos sospechosos, informando problemas cuando los detecta. *Los sistemas de prevención de intrusiones (IPS)*, así como el monitoreo de actividades sospechosas, pueden intervenir para evitar que suceda. A diferencia de un firewall, que mira principalmente hacia afuera para evitar que entren cosas malas, IDS e IPS buscan activamente dentro del perímetro en busca de comportamientos sospechosos. Cuando comienza desde cero, IDS puede tener más sentido. Estos sistemas se basan en la heurística (al igual que muchos firewalls de aplicaciones), y es posible que el conjunto de reglas de inicio genérico sea demasiado indulgente o no lo suficientemente indulgente para el comportamiento de su servicio. El uso de un IDS más pasivo para alertarlo sobre problemas es una buena manera de ajustar sus reglas antes de usarlo en una capacidad más activa.

Segregación de red

Con un sistema monolítico, tenemos límites sobre 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 los servicios se comunican 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 interconexión e incluso enrutar el tráfico a través de puertas de enlace para acceder al proxy, lo que le brinda múltiples perímetros en los que se pueden implementar medidas de seguridad adicionales.

Esto podría permitirle segmentar redes en función de la propiedad del equipo, o tal vez por nivel de riesgo.

Sistema operativo

Nuestros sistemas dependen de una gran cantidad de software que no escribimos y pueden tener vulnerabilidades de seguridad que podrían exponer nuestra aplicación, es decir, nuestros sistemas operativos y otras herramientas de soporte que ejecutamos en ellos. Aquí, los consejos básicos pueden ayudarte mucho. Comience con solo ejecutar servicios como usuarios del sistema operativo que tienen la menor cantidad de permisos posible, para asegurarse de que si dicha cuenta se ve comprometida, causará un daño mínimo.

A continuación, parchee su software. Regularmente. Esto debe automatizarse y debe saber si sus máquinas no están sincronizadas con los últimos niveles de parches. Herramientas como SCCM de Microsoft o Spacewalk de RedHat pueden ser beneficiosas aquí, ya que pueden ayudarlo a ver si las máquinas están actualizadas con los últimos parches e iniciar actualizaciones si es necesario. Si está utilizando 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 harán todo por usted.

Esto realmente es 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 parchear, entonces su sistema aún podría ser extremadamente vulnerable.

Otra cosa a tener en cuenta si está utilizando Linux es la aparición de módulos de seguridad para el propio sistema operativo. AppArmor, por ejemplo, le permite definir cómo se espera que se comporte su aplicación, con el kernel vigilándola. Si comienza a hacer algo que no debería, el kernel interviene. AppArmor ha existido por un tiempo, al igual que SELinux. Aunque técnicamente cualquiera de ellos debería funcionar en cualquier sistema Linux moderno, en la práctica algunas distribuciones soportan uno mejor que el otro. AppArmor se usa de forma predeterminada en Ubuntu y SuSE, por ejemplo, mientras que SELinux tradicionalmente ha sido bien soportado por RedHat. Una opción más nueva es GrSecurity, que pretende ser más fácil de usar que AppArmor o GrSecurity y al mismo tiempo intenta expandir sus capacidades, pero requiere un kernel 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 el trabajo.

Un ejemplo resuelto

Tener una arquitectura de sistema de grano más fino 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 darnos el lujo de ser mucho más laxos en lo que nos preocupa.

Consideremos MusicCorp una vez más y juntemos algunos de los conceptos anteriores para ver dónde y cómo podemos usar algunas de estas técnicas de seguridad. Estamos analizando 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 falta aplastante de consideración por las preocupaciones de seguridad. Todo se envía a través de HTTP simple y antiguo.

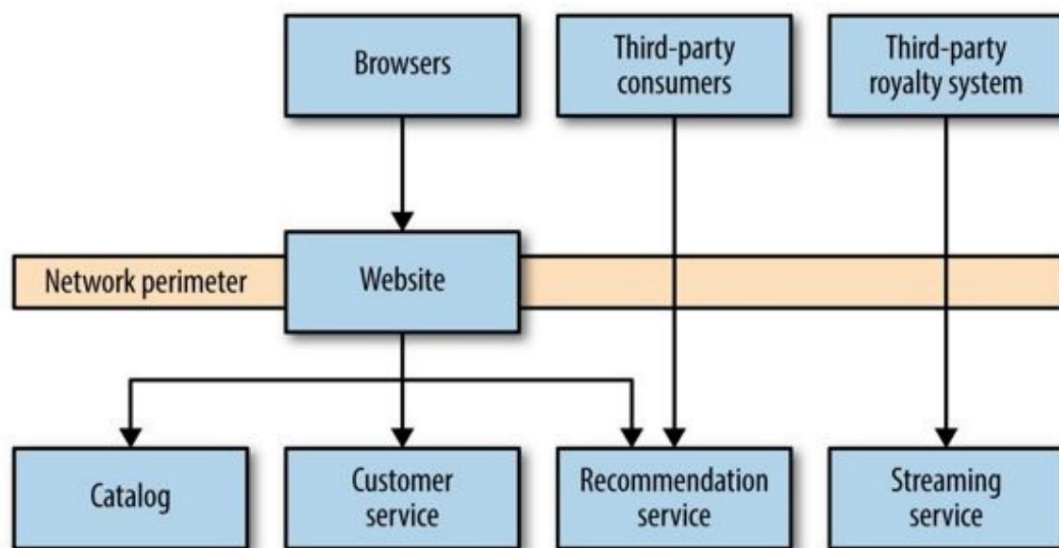


Figura 9-4. Un subconjunto de la arquitectura lamentablemente insegura de MusicCorp

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

También presentamos el concepto de una puerta de enlace de regalías de terceros: comenzamos a trabajar con una empresa de terceros que manejará los pagos de regalías para nuestro nuevo servicio de transmisión. Nos contacta ocasionalmente para obtener registros de qué música se ha transmitido y cuándo: información que protegemos celosamente porque nos preocupa la competencia de compañías rivales. Finalmente, exponemos los datos de nuestro catálogo a otros terceros; por ejemplo, permitimos que los metadatos sobre el artista o la canción se incrusten en sitios de revisión de música. Dentro del perímetro de nuestra red, tenemos algunos servicios de colaboración, que solo se usan internamente.

Para el navegador, usaremos una combinación de tráfico HTTP estándar para contenido no seguro, para permitir que se almacene en caché. Para las páginas seguras con inicio de sesión, todo el contenido seguro se enviará a través de HTTPS, lo que brindará a nuestros clientes protección adicional si están haciendo cosas como funcionar en redes WiFi públicas.

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

obtener son legítimos. Aquí, 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 solicite estos datos. 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?

Para las fuentes de datos del catálogo, ¡queremos que esta información se comparta lo más ampliamente posible para permitir que las personas nos compren música fácilmente! Sin embargo, no queremos que se abuse de esto y nos gustaría tener una idea de quién está usando nuestros datos. Aquí, las claves API tienen mucho sentido.

Dentro del perímetro de la red, las cosas son un poco más matizadas. ¿Qué tan preocupados estamos de que las personas comprometan nuestras redes internas? Idealmente, nos gustaría usar HTTPS como mínimo, pero administrarlo es algo complicado. En cambio, decidimos poner el trabajo (inicialmente, al menos) en fortalecer el perímetro de nuestra red, lo que incluye tener un firewall configurado correctamente y seleccionar un dispositivo de seguridad de hardware o software apropiado para verificar el tráfico malicioso (por ejemplo, escaneo de puertos o denegación de servicio). ataques).

Dicho esto, nos preocupan *algunos* de nuestros datos y dónde residen. No nos preocupa el servicio de catálogo; después de todo, ¡queremos que esos datos se compartan y hemos proporcionado una API para ello! Pero estamos muy preocupados por los datos de nuestros clientes. Aquí, decidimos cifrar los datos en poder del servicio de atención al cliente y descifrar los datos 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 utilizarlos.

La figura 9-5 muestra la imagen final. Como puede ver, las elecciones que hicimos sobre qué tecnología usar se basaron en una comprensión de la naturaleza de la información que se protege. 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.

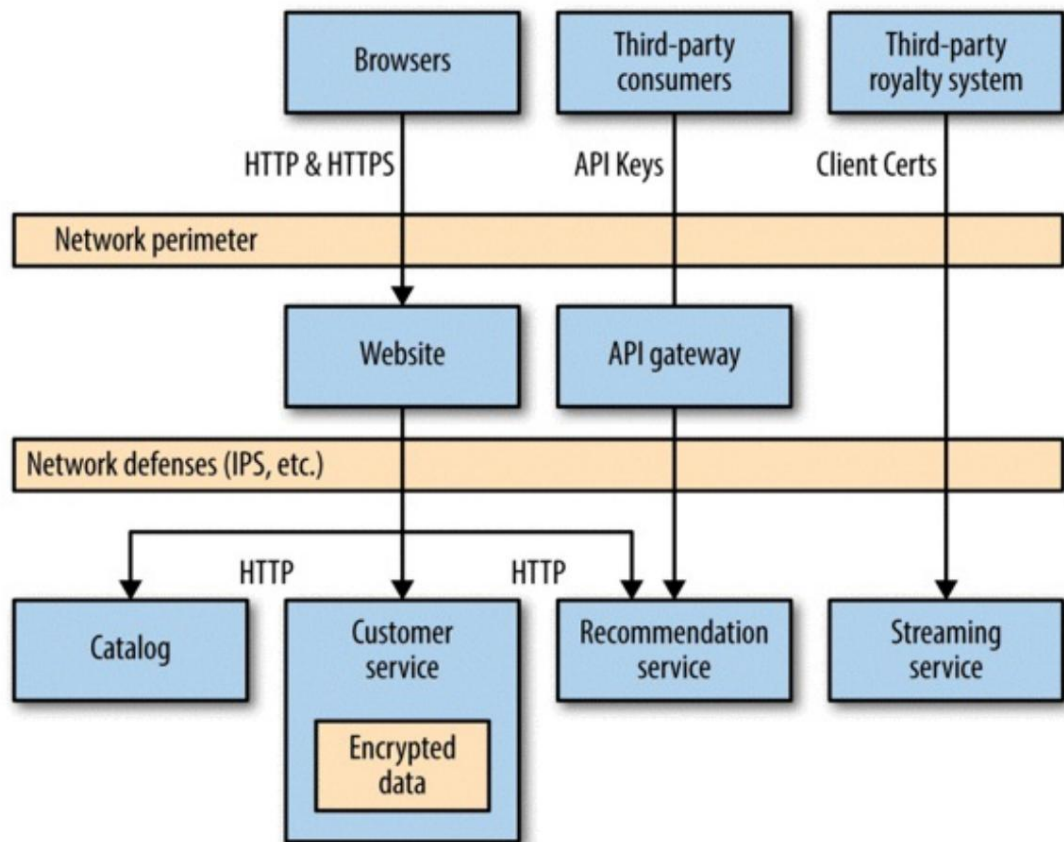


Figura 9-5. El sistema más seguro de MusicCorp

ser frugal

A medida que el espacio en disco se abarata 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 ven cada vez más los datos como un activo valioso, sino también para los usuarios que valoran su propia privacidad. Los datos que pertenecen a un individuo, o que podrían usarse para obtener información sobre un individuo, deben ser los datos con los que tenemos más cuidado.

Sin embargo, ¿y si nos facilitamos un poco la vida? ¿Por qué no borrar la mayor cantidad de información posible 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 con x? ¿Necesitamos almacenar el nombre, la edad, el sexo y la fecha de nacimiento de alguien para ofrecerle ofertas de productos, o su rango de edad y código postal es suficiente información?

Las ventajas aquí son múltiples. Primero, si no lo guardas, nadie puede robarlo. En segundo lugar, si no lo almacena, ¡nadie (p. ej., una agencia gubernamental) puede solicitarlo tampoco!

La frase alemana *Datensparsamkeit* representa este concepto. Con origen en la legislación de privacidad alemana, encapsula el concepto de almacenar solo la información que sea *absolutamente necesaria* para cumplir con las operaciones comerciales o cumplir con las leyes locales.

Obviamente, esto está en tensión directa con el movimiento hacia el almacenamiento de más y más información, ¡pero es un comienzo para darse cuenta de que esta tensión incluso existe!

El elemento humano

Gran parte de lo que hemos cubierto aquí son los conceptos básicos de cómo implementar medidas de seguridad tecnológicas para proteger sus sistemas y datos de atacantes externos malintencionados. Sin embargo, es posible que también necesite procesos y políticas para tratar con el elemento humano en su organización. ¿Cómo se revoca el acceso a las credenciales cuando alguien deja la organización? ¿Cómo puedes protegerte contra la ingeniería social? Como un buen ejercicio mental, considere el daño que un ex empleado descontento podría causarle a sus sistemas si quisiera. Ponerse en la mentalidad de una parte malintencionada suele ser una buena manera de razonar sobre las protecciones que puede necesitar, ¡y pocas partes malintencionadas tienen tanta información interna como un empleado reciente!

La regla de oro

Si no hay nada más que se lleve de este capítulo, que sea esto: no escriba su propia criptografía. No inventes tus 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, es posible que se equivoque.

Muchas de las herramientas descritas anteriormente, como AES, son tecnologías reforzadas por la industria cuyos algoritmos subyacentes han sido revisados por pares y cuya implementación de software se ha probado y reparado rigurosamente durante muchos años. ¡Son lo suficientemente buenos! En muchos casos, reinventar la rueda es solo una pérdida de tiempo, pero cuando se trata de seguridad, puede ser completamente peligroso.

Seguridad para hornear en

Al igual que con las pruebas funcionales automatizadas, no queremos que la seguridad se deje en manos de un grupo diferente de personas, ni queremos dejar todo para el último minuto. 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 las personas se familiaricen con la lista OWASP Top Ten y el marco de prueba de seguridad de OWASP puede ser un excelente lugar para comenzar. Sin embargo, los especialistas tienen absolutamente su lugar y, si tiene acceso a ellos, utilícelos para que lo ayuden.

Existen herramientas automatizadas que pueden sondear nuestros sistemas en busca de vulnerabilidades, por ejemplo, buscando ataques de secuencias de comandos entre sitios. El Zed Attack Proxy (también conocido como ZAP) es un buen ejemplo. Informado por el trabajo de OWASP, ZAP intenta recrear ataques maliciosos en su sitio web. Existen otras herramientas que usan análisis estático para buscar errores de codificación comunes que pueden abrir brechas de seguridad, como **Brakeman** para Ruby. Cuando estas herramientas se puedan integrar fácilmente en compilaciones de CI normales, intégreles en sus comprobaciones estándar. Otros tipos de pruebas automatizadas son más complicadas. Por ejemplo, usar algo como Nessus para buscar vulnerabilidades es un poco más complicado y puede requerir que un ser humano interprete los resultados. Dicho esto, estas pruebas aún son automatizables y puede tener sentido ejecutarlas con el mismo tipo de cadencia que las pruebas de carga.

El **ciclo de vida de desarrollo de seguridad** de Microsoft también tiene algunos buenos modelos sobre cómo los equipos de entrega pueden integrar la seguridad. Algunos aspectos se sienten demasiado en cascada, pero eche un vistazo y vea qué aspectos pueden encajar en su flujo de trabajo actual.

Verificación externa

Con seguridad, creo que es de gran valor tener una evaluación externa. Los ejercicios como las pruebas de penetración, cuando los realiza una parte externa, realmente imitan los intentos del mundo real. También eluden el problema de que los equipos no siempre pueden ver los errores que han cometido, ya que están demasiado cerca del problema. Si es una empresa lo suficientemente grande, es posible que tenga un equipo de seguridad de la información dedicado que pueda ayudarlo. Si no, busque una parte externa que pueda hacerlo. Comuníquese con ellos lo antes posible, comprenda cómo les gusta trabajar y averigüe con cuánta antelación necesitan hacer una prueba.

También deberá considerar cuánta verificación necesita antes de cada lanzamiento.

Por lo general, no es necesario realizar una prueba de penetración completa, por ejemplo, para versiones incrementales pequeñas, pero puede serlo para cambios más grandes. Lo que necesita depende de su propio perfil de riesgo.

Resumen

Entonces, nuevamente volvemos a un tema central del libro: tener un sistema descompuesto en servicios más detallados nos brinda muchas más opciones sobre cómo resolver un problema. Tener microservicios no solo puede reducir potencialmente el impacto de cualquier infracción, sino que también nos brinda una mayor capacidad para compensar la sobrecarga de enfoques más complejos y seguros donde los datos son confidenciales, 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, debe comenzar a tener una idea de cuándo considerar la seguridad durante el tránsito, en reposo o en absoluto.

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

Si desea obtener una descripción general de la seguridad para las aplicaciones basadas en navegador, un excelente lugar para comenzar es la excelente [organización sin fines de lucro Open Web Application Security Project \(OWASP\)](#), cuyo documento *Top 10 Security Risk*, actualizado regularmente, debe considerarse una lectura esencial para cualquier desarrollador. Finalmente, si desea una discusión más general sobre criptografía, consulte el libro *Ingeniería criptográfica* de Niels Ferguson, Bruce Schneier y Tadayoshi Kohno (Wiley).

Familiarizarse con la seguridad a menudo se trata de comprender a las personas y cómo funcionan con nuestros sistemas. Un aspecto relacionado con el ser humano que aún no hemos discutido en términos de microservicios es la interacción entre las estructuras organizativas y las arquitecturas mismas. 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 romper una clave por fuerza bruta. Por lo tanto, puede suponer que cuanto más larga sea la clave, más seguros serán sus datos. Sin embargo, el respetado experto en seguridad [Bruce Schneier](#) ha planteado algunas preocupaciones menores sobre la implementación de AES-256 para ciertos tipos de claves. ¡Esta es una de esas áreas en las que necesita investigar más sobre cuáles son los consejos actuales en el momento de la lectura!

Capítulo 10. Ley de Conway y diseño de sistemas

Gran parte del libro hasta ahora se ha centrado en los desafíos técnicos para avanzar hacia una arquitectura de grano fino. Pero también hay otras cuestiones organizativas a tener en cuenta. Como aprenderemos en este capítulo, ¡usted ignora el organigrama de su empresa bajo su propio riesgo!

Nuestra industria es joven y parece reinventarse 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 los transistores en los 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 verdadera 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 más ampliamente aquí que solo sistemas de información) inevitablemente producirá un diseño cuya estructura es una copia de la estructura de comunicación de la organización.

Esta declaración se cita a menudo, en varias formas, como la ley de Conway. Eric S. Raymond resumió este fenómeno en *The New Hacker's Dictionary* (MIT Press) al afirmar: "Si tiene cuatro grupos trabajando en un compilador, obtendrá un compilador de 4 pasos".

Evidencia

La historia cuenta que cuando Melvin Conway envió su artículo sobre este tema a Harvard Business Review, lo rechazaron, alegando que no había probado su tesis. He visto esta teoría confirmada en tantas situaciones diferentes que la he aceptado como cierta. Pero no tiene que creer en mi palabra: desde la presentación original de Conway, se ha trabajado mucho en esta área. Se han llevado a cabo varios estudios para explorar la interrelación de la estructura organizativa y los sistemas que crean.

Organizaciones sueltas y fuertemente acopladas

En *Exploring the Duality Between Product and Organizational Architectures* (Harvard Business School), los autores Alan MacCormack, John Rusnak y Carliss Baldwin analizan varios sistemas de software diferentes, categorizados libremente como creados por *organizaciones débilmente acopladas* u organizaciones *estrechamente acopladas*. Para organizaciones fuertemente acopladas, piense en firmas de productos comerciales que normalmente están ubicadas con visiones y objetivos fuertemente alineados, mientras que las organizaciones débilmente acopladas están bien representadas por comunidades distribuidas de código abierto.

En su estudio, en el que emparejaron pares de productos similares de cada tipo de organización, los autores encontraron que las organizaciones menos acopladas en realidad crearon sistemas más modulares y menos acoplados, mientras que el software de la organización más enfocada estaba menos modularizado.

Windows Vista

Microsoft llevó a cabo **un estudio empírico** en el que analizó cómo su propia estructura organizativa afectaba la calidad del software de un producto específico, Windows Vista. Específicamente, los investigadores observaron múltiples factores para determinar qué tan propenso a errores sería un común del sistema. métricas de calidad de código. Después de observar múltiples ejemplos de código, se eliminaron estas métricas asociadas con las estructuras organizacionales demostraron ser las medidas estadísticamente más relevantes.

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

netflix y amazon

Probablemente los dos exponentes de la idea de que las organizaciones y la arquitectura deben estar alineadas sean Amazon y Netflix. Desde el principio, Amazon comenzó a comprender los beneficios de que los equipos sean dueños de todo el ciclo de vida de los sistemas que administran. Quería que los equipos fueran propietarios y operaran los sistemas que cuidaban, administrando todo el ciclo de vida. Pero Amazon también sabía que los equipos pequeños pueden trabajar más rápido que los grandes. Esto condujo a sus famosos *equipos de dos pizzas*, donde ningún equipo debería ser tan grande que no pudiera alimentarse con dos pizzas. Este impulsor de 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 para permitir que sus equipos fueran autosuficientes.

Netflix aprendió de este ejemplo, y aseguró que desde un principio se estructuró en torno a equipos pequeños e independientes, para que los servicios que crearan también fueran independientes entre sí. Esto aseguró que la arquitectura del sistema estuviera optimizada para la velocidad del cambio. Efectivamente, Netflix diseñó la estructura organizacional para la arquitectura del sistema que quería.

¿Qué podemos hacer con esto?

Entonces, la evidencia, anecdótica y empírica, apunta a que nuestra estructura organizacional tiene una fuerte influencia en la naturaleza (y calidad) de los sistemas que brindamos. Entonces, ¿cómo nos ayuda este entendimiento? Veamos algunas situaciones organizacionales diferentes y comprendamos qué impacto podría tener cada una en el diseño de nuestro sistema.

Adaptarse a las vías de comunicación

Consideremos primero un solo equipo simple. Está a cargo de todos los aspectos del diseño e implementación del sistema. Puede tener una comunicación frecuente y detallada. Imagine que este equipo está a cargo de un solo servicio, por ejemplo, el servicio de catálogo de nuestra tienda de música. Ahora considere el interior de un servicio: muchas llamadas a métodos o funciones detalladas. Como hemos discutido antes, nuestro objetivo es garantizar que nuestros servicios se descompongan de tal manera que el ritmo de cambio dentro de un servicio sea mucho más alto que el ritmo de cambio entre servicios. Este único equipo, con su capacidad de 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 propiedad.

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 y, de hecho, tienen la propiedad conjunta del servicio. Los límites geográficos y de zona horaria aquí dificultan la comunicación detallada entre esos equipos. En cambio, confían en una comunicación más detallada a través de videoconferencias y correo electrónico. ¿Qué tan fácil es para un miembro del equipo en el 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 aumenta el costo de coordinar el cambio, sucede una de dos cosas. O las personas encuentran formas de reducir los costos de coordinación/comunicación, o dejan de hacer cambios.

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

Recuerdo un proyecto de cliente en el que trabajé donde la propiedad de un solo servicio se compartía entre dos ubicaciones geográficas. Eventualmente, cada sitio comenzó a especializarse en el trabajo que manejaba. Esto le permitió tomar posesión de parte del código base, dentro del cual podría tener un costo de cambio más fácil. Luego, los equipos tuvieron una comunicación más detallada sobre cómo se interrelacionaban las dos partes; efectivamente, las vías de comunicación posibles dentro de la estructura organizativa coincidían con la API de granularidad gruesa que formaba el límite entre las dos mitades del código base.

Entonces, ¿dónde nos deja esto al considerar la evolución de nuestro propio diseño de servicio? Bueno, sugeriría que los límites geográficos 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, debe buscar asignar la propiedad de un servicio a un solo equipo ubicado. que puede mantener bajo el costo del cambio.

Quizás su organización decida que quiere aumentar la cantidad de personas que trabajan en su proyecto abriendo una oficina en otro país. En este punto, piense activamente sobre qué partes de su sistema se pueden mover. Quizás esto es lo que impulsa sus decisiones sobre qué costuras dividir a continuación.

También vale la pena señalar en este punto que, al menos según las observaciones de los autores del informe *Exploring the Duality Between Product and Organizational Architectures* al que se hizo referencia anteriormente, si la organización que construye el sistema está menos acoplada (p. equipos), los sistemas que se están construyendo tienden a ser más modulares y, por lo tanto, 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 propietario de un servicio es responsable de realizar cambios en ese servicio. El equipo debe sentirse libre de reestructurar el código como quiera, siempre que ese cambio no interrumpa el consumo de servicios. Para muchos equipos, *la propiedad* se extiende a todos los aspectos del servicio, desde los requisitos de abastecimiento hasta la creación, implementación y mantenimiento de la aplicación. Este modelo prevalece especialmente con 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 “tirar algo por encima del muro” se disipan cuando no hay a quién tirarlo.

Este modelo es sin duda uno de mis favoritos. Empuja las decisiones a las personas más capacitadas para tomarlas, otorgando al equipo mayor poder y autonomía, pero también haciéndolo 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.

Controladores para servicios compartidos

He visto a muchos equipos adoptar un modelo de propiedad de servicios compartidos. Encuentro este enfoque subóptimo, por razones ya discutidas. Sin embargo, es importante comprender los impulsores que hacen que las personas elijan servicios compartidos, especialmente porque podemos 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 solo servicio propiedad de más de un equipo es que el costo de dividir el servicio es demasiado alto, o quizás su organización no le vea el sentido. Esta es una ocurrencia común con grandes sistemas monolíticos. Si este es el principal desafío al que se enfrenta, entonces espero que algunos de los consejos que se dan en el [Capítulo 5](#) le sean de utilidad. También podría considerar fusionar equipos para alinearse más estrechamente con la arquitectura misma.

Equipos destacados

La idea de los equipos de funciones (también conocidos como equipos basados en funciones) es que un pequeño equipo impulse el desarrollo de un conjunto de funciones, implementando todas las funciones requeridas, incluso si traspasa los límites de los componentes (o incluso del servicio). Los objetivos de los equipos de características son lo suficientemente sensatos. Esta estructura permite que el equipo se concentre en el resultado final y asegura que el trabajo esté unido, lo que evita algunos de los desafíos de tratar de coordinar los cambios entre múltiples equipos diferentes.

En muchas situaciones, el equipo de características es una reacción a las organizaciones de TI tradicionales donde la estructura del equipo se alinea en torno a los 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 maneje la base de datos. En este entorno, un equipo de características es un paso importante, ya que trabaja en todas estas capas para ofrecer la funcionalidad.

Con la adopción masiva de equipos de características, todos los servicios pueden considerarse compartidos. Todos pueden cambiar cada servicio, cada pieza de código. El papel de los custodios del servicio aquí se vuelve mucho más complejo, si es que existe. Desafortunadamente, rara vez veo custodios en funcionamiento donde se adopta este patrón, lo que lleva a los tipos de problemas que discutimos anteriormente.

Pero volvamos a considerar qué son los microservicios: servicios modelados según un dominio comercial, no uno técnico. Y si nuestro equipo que posee un servicio dado está alineado de manera similar a lo largo del dominio comercial, es mucho más probable que el equipo pueda mantener un enfoque en el cliente y ver más del desarrollo de funciones, porque tiene una comprensión holística y propiedad 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 los servicios compartidos es evitar los cuellos de botella en la entrega.

¿Qué sucede si hay una gran acumulación de cambios que deben realizarse en un solo servicio? Imaginemos que estamos implementando la capacidad para que un cliente vea el género de una pista en nuestros productos, además de agregar un nuevo tipo de stock: tonos de llamada musicales virtuales para el teléfono móvil. El equipo del sitio web necesita hacer un cambio para mostrar la información del género, y el equipo de la aplicación móvil está trabajando para permitir que los usuarios exploren, obtengan una vista previa y compren los tonos de llamada. Es necesario realizar ambos cambios en el servicio de catálogo, pero desafortunadamente la mitad del equipo está con 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 la aplicación móvil pasan a otra cosa.

Dependiendo de la importancia de la función o de la duración probable de la demora, esto puede estar bien o puede ser un problema importante.

En su lugar, podría agregar personas al equipo del catálogo para ayudarlos a avanzar en su trabajo más rápido. Cuanto más estandarizada sea la pila de tecnología y los lenguajes de programación en uso en su sistema, más fácil será para otras personas realizar cambios en sus servicios. La otra cara, por supuesto, como discutimos anteriormente, es que la estandarización tiende a reducir la capacidad de un equipo para adoptar la solución correcta 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 separado 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 esta sea un área en la que desarrollaremos mucho en el futuro también es bastante baja, esto bien puede ser prematuro. Por otro lado, si hay acumuladas 10 semanas de funciones relacionadas con los tonos de llamada, podría tener sentido dividir el servicio y que el equipo móvil tome la propiedad.

Sin embargo, hay otro modelo que podría funcionar bien para nosotros.

Código abierto interno

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

Con el código abierto normal, un pequeño grupo de personas se consideran responsables principales. Son los custodios del código. Si desea un cambio en un proyecto de código abierto, puede pedirle a uno de los encargados de confirmar que lo haga por usted, o puede hacerlo usted mismo y enviarle una solicitud de incorporación de cambios. Los encargados principales todavía están a cargo de la base de código; ellos son los dueños.

Dentro de la organización, este patrón también puede funcionar bien. Quizás las personas que trabajaron originalmente en el servicio ya no forman parte de un equipo; tal vez ahora están dispersos por toda la organización. Bueno, si todavía tienen derechos de compromiso, puede encontrarlos y pedirles ayuda, tal vez emparejarse con ellos, o si tiene las herramientas adecuadas, puede enviarles una solicitud de extracción.

Rol de los Custodios

Todavía queremos que nuestros servicios sean sensatos. Queremos que el código sea de una calidad decente y que el servicio en sí muestre algún tipo de consistencia en la forma en que se ensambla. 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 internamente los mismos patrones que se usan en el código abierto normal, lo que significa separar un grupo de confirmadores de confianza (el equipo central) y los que no son de confianza (personas externas al equipo que envían cambios).

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

Los buenos guardianes ponen mucho trabajo en esto, comunicándose claramente con los remitentes y alentando el buen comportamiento. Los malos guardianes pueden usar esto como una excusa para ejercer poder sobre otros o tener guerras religiosas sobre decisiones técnicas arbitrarias. Habiendo visto ambos conjuntos de comportamiento, puedo decirles que una cosa está clara: de cualquier manera lleva tiempo. Al considerar permitir que los autores que no son de confianza envíen cambios a su base de código, debe decidir si los gastos generales de ser un guardián valen 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 ajenas al equipo principal envíen parches. Antes de que la columna vertebral clave de un servicio esté en su lugar, es posible que el equipo no sepa cómo se ve *bien* y, por lo tanto, puede tener dificultades para saber cómo se ve una buena presentación. Durante esta etapa, el propio servicio está experimentando un alto grado de cambio.

La mayoría de los proyectos de código abierto tienden a no recibir presentaciones de un grupo más amplio de personas que no son de confianza hasta que se completa el núcleo de la primera versión. Seguir un modelo similar para sus propias organizaciones tiene sentido. Si un servicio es bastante maduro y rara vez se cambia, por ejemplo, nuestro servicio de carrito, entonces tal vez ese sea el momento de abrirlo para otras contribuciones.

Estampación

Para admitir mejor un modelo interno de código abierto, necesitará algunas herramientas en su lugar. El uso de una herramienta de control de versiones distribuidas con la capacidad de que las personas envíen solicitudes de extracción (o algo similar) es importante. Dependiendo del tamaño de la organización, es posible que también necesite herramientas para permitir una discusión y evolución de las solicitudes de parches; esto puede o no significar un sistema completo de revisión de código, pero la capacidad de comentar en línea sobre los parches es muy útil. Por último, deberá hacer que sea muy fácil para un responsable de compilar e implementar su software, y ponerlo a disposición de otros. Por lo general, esto implica tener canales de desarrollo e implementación bien definidos y repositorios de artefactos centralizados.

Contextos acotados y estructuras de equipo

Como se mencionó anteriormente, buscamos trazar los límites de nuestro servicio en torno a contextos limitados. Por lo tanto, nos gustaría que nuestros equipos también estuvieran alineados en contextos limitados. Esto tiene múltiples beneficios. Primero, a un equipo le resultará más fácil comprender los conceptos de dominio dentro de un contexto limitado, ya que están interrelacionados. En segundo lugar, es más probable que los servicios dentro de un contexto limitado sean servicios que se comunican entre sí, lo que facilita el diseño del sistema y la coordinación de versiones. Finalmente, 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 de huérfanos?

Entonces, ¿qué pasa con los servicios que ya no se mantienen activamente? 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 sean más simples. Es posible que los servicios más simples con menos funcionalidades no necesiten cambiar por un tiempo. Considere el humilde servicio de carrito, que proporciona algunas capacidades bastante modestas: Agregar al carrito, Eliminar del carrito, etc. Es bastante concebible que este servicio no tenga que cambiar durante meses después de haber sido escrito por primera vez, incluso si el desarrollo activo aún continúa. ¿Qué pasa aquí? ¿Quién es el propietario de este servicio?

Si las estructuras de su equipo están alineadas a lo largo de los contextos limitados de su organización, incluso los servicios que no se cambian con frecuencia aún tienen un propietario de facto. Imagine un equipo que esté alineado con el contexto de ventas web del consumidor. Puede manejar el sitio web, el carrito y los servicios de recomendación. Incluso si el servicio de carrito no se ha cambiado 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, volver a escribirlo no debería llevar mucho tiempo.

Dicho esto, si ha adoptado un enfoque verdaderamente políglota, haciendo uso de varias pilas de tecnología, entonces los desafíos de realizar cambios en un servicio huérfano podrían complicarse 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 esto abarca múltiples facetas diferentes, cada una de las cuales opera como una sola línea de negocio (LOB). Por ejemplo, una línea de negocios trata con propiedades residenciales en Australia, otra comercial, mientras que otra podría relacionarse con uno de los negocios de REA en el extranjero. Estas líneas de negocio tienen equipos (o *brigadas*) de entrega de TI asociados con ellas; algunos pueden tener un solo equipo, mientras que el más grande tiene cuatro. Entonces, para la propiedad residencial, hay varios equipos involucrados en la creación del sitio web y los servicios de listado para permitir que las personas exploren la propiedad. Las personas rotan entre estos equipos de vez en cuando, pero tienden a permanecer dentro de esa línea de negocios durante períodos prolongados, lo que garantiza que los miembros del equipo puedan desarrollar una sólida conciencia 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 escuadrón dentro de una línea de negocio posea todo el ciclo de vida de los servicios que crea, incluida la construcción, prueba y lanzamiento, soporte e incluso desmantelamiento. Un equipo central de servicios de entrega brinda asesoramiento y orientación a estos equipos, así como herramientas para ayudarlos a realizar el trabajo. Una sólida cultura de automatización es clave, y REA hace un uso intensivo de AWS como una parte clave para permitir que los equipos sean más autónomos. La figura 10-1 ilustra cómo funciona todo esto.

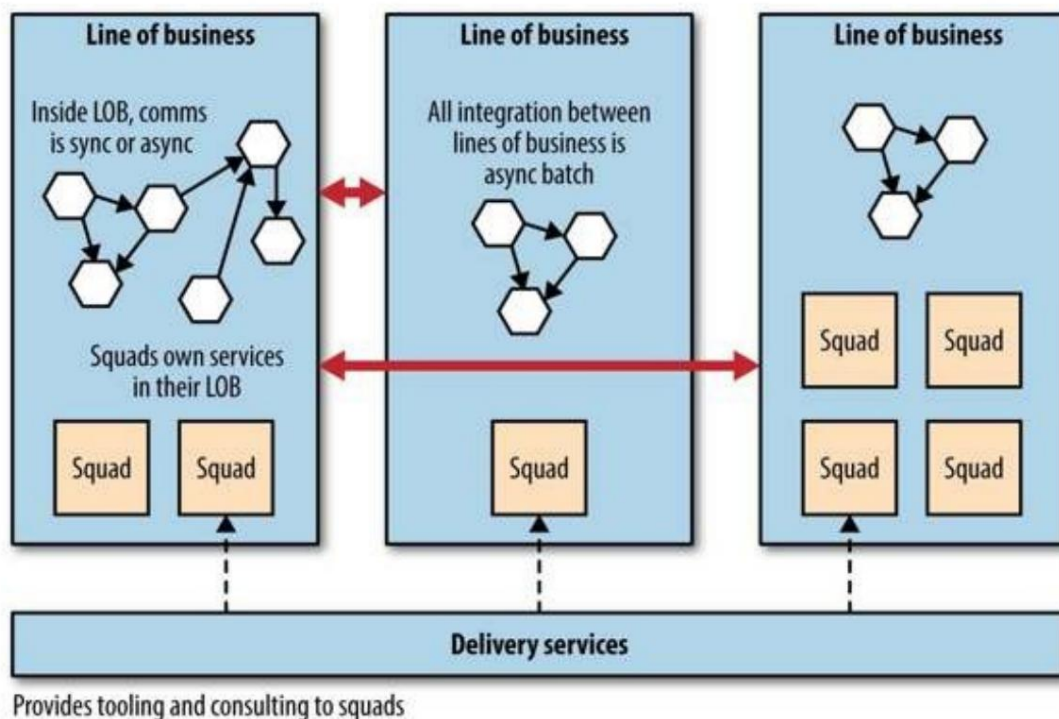


Figura 10-1. Una descripción general de la estructura organizacional y del equipo de RealEstate.com.au, y la alineación con la arquitectura

No es solo la organización de entrega la que está alineada con la forma en que opera el negocio. Se extiende a la arquitectura también. Un ejemplo de esto son los métodos de integración. Dentro de un LOB, todos los servicios son libres de comunicarse entre sí de la forma que consideren adecuada, según lo decidan los escuadrones que actúan como sus custodios. Pero entre LOB, toda comunicación tiene el mandato de

ser lote asincrónico, una de las pocas reglas de hierro fundido del equipo de arquitectura muy pequeño.

Esta comunicación de grano grueso coincide con la comunicación de grano grueso que existe entre las diferentes partes del negocio también. Al insistir en que sea por lotes, cada LOB tiene mucha libertad en cómo actúa y se administra a sí mismo. Podría darse el lujo de cancelar sus servicios cuando quisiera, sabiendo que mientras pueda satisfacer la integración por lotes con otras partes del negocio y sus propios accionistas comerciales, a nadie le importará.

Esta estructura ha permitido una importante autonomía no solo de los equipos sino también de las diferentes partes del negocio. De un puñado de servicios hace unos años, REA ahora tiene cientos, con más servicios que personas, y está creciendo a un ritmo acelerado. La capacidad de generar cambios ha ayudado a la empresa a lograr un éxito significativo en el mercado local hasta el punto de expandirse en el extranjero. Y, lo más alentador de todo, al hablar con la gente allí, tengo la impresión de que tanto la arquitectura como la estructura organizativa tal como están ahora son solo la última iteración en lugar del destino. Me atrevo a decir que dentro de otros cinco años REA volverá a verse muy diferente.

Aquellas organizaciones que son lo suficientemente adaptables para cambiar no solo la arquitectura de su sistema, sino también su estructura organizativa pueden generar enormes beneficios en términos de una mayor autonomía de los equipos y un tiempo de comercialización más rápido para las nuevas características y funcionalidades. REA es solo una de varias organizaciones que se están dando cuenta de que la arquitectura del sistema no existe en el vacío.

Ley de Conway al revés

Hasta ahora, hemos hablado sobre cómo la organización impacta el diseño del sistema. Pero, ¿y al revés? Es decir, ¿puede un diseño de sistema cambiar la organización? Si bien no he podido encontrar la misma calidad de evidencia para respaldar la idea de que la ley de Conway funciona a la inversa, lo he visto de manera anecdótica.

Probablemente el mejor ejemplo fue un cliente con el que trabajé hace muchos años. En los días en que la Web era bastante incipiente e Internet se veía como algo que llegaba en un disquete de AOL a través de la puerta, esta empresa era una gran empresa de impresión 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 era bastante poco importante para la forma en que operaba el negocio. Cuando se creó el sistema original, se tomó una decisión técnica bastante arbitraria sobre cómo funcionaría el sistema.

El contenido de este sistema se obtuvo de varias maneras, 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 varias maneras, y un sistema de salida que creaba el sitio web final que el público en general podía navegar.

Si las decisiones de diseño originales fueron correctas en ese momento es una conversación para los historiadores, pero muchos años después, la empresa había cambiado bastante y yo y muchos de mis colegas comenzamos 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 con este sistema de tres partes. Tres canales o divisiones en el lado de TI del negocio alineados con cada una de las partes de entrada, núcleo y salida del negocio. Dentro de esos canales, había equipos de entrega separados. De 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 disminuía el lado impreso del negocio y crecía el lado digital del negocio, el diseño del sistema inadvertidamente abrió el camino para el crecimiento de la organización.

Al final, nos dimos cuenta de que, independientemente de 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 pequeño mundo. Tiene que ser más consciente de las implicaciones de cosas como llamadas a través de los límites de la red o las implicaciones de fallas.

También hemos hablado sobre la capacidad de los microservicios para facilitar la prueba de nuevas tecnologías, desde almacenes de datos hasta idiomas. Pero si se está mudando de un mundo donde tiene un sistema monolítico, donde la mayoría de sus desarrolladores solo han tenido que usar un idioma y permanecer completamente ajenos a las preocupaciones operativas, entonces lanzarlos al mundo de los microservicios puede ser una mala educación. despertar para ellos.

Del mismo modo, impulsar el poder en los equipos de desarrollo para aumentar la autonomía puede ser complicado. Las personas que en el pasado han tirado trabajo por la borda a otra persona están acostumbradas a tener a alguien más a quien culpar, y es posible que no se sientan cómodas siendo plenamente responsables de su trabajo. ¡Incluso puede encontrar barreras contractuales para que sus desarrolladores lleven buscapersonas de soporte para los sistemas que soportan!

Aunque este libro ha sido principalmente sobre tecnología, las personas no son solo un tema secundario a considerar; son las personas que construyeron lo que tienes ahora y construirán lo que sucederá a continuación. Proponer una visión de cómo se deben hacer las cosas sin considerar cómo se sentirá su personal actual al respecto o sin considerar qué capacidades tienen es probable que lo lleve a un mal lugar.

Cada organización tiene su propio conjunto de dinámicas en torno a este tema. Comprenda el apetito de cambio de su personal. ¡No los presione demasiado rápido! Tal vez todavía tenga un equipo separado que se encargue de la implementación o el soporte de primera línea durante un período corto de tiempo, lo que le da a sus desarrolladores tiempo para adaptarse a otras prácticas nuevas. Sin embargo, es posible que tenga que 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 debe tener claro por qué esas responsabilidades son importantes para usted.

Esto puede ayudarlo a ver cuáles podrían ser sus brechas de habilidades y pensar en cómo cerrarlas.

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

Resumen

La ley de Conway destaca los peligros de tratar de imponer un diseño de sistema que no coincida con la organización. Esto nos lleva a tratar de alinear la propiedad del servicio con los equipos ubicados, que a su vez están alineados en torno a los mismos contextos delimitados de la organización.

Cuando los dos no están alineados, obtenemos puntos de tensión como se describe a lo largo de este capítulo. Al reconocer el vínculo entre los dos, nos aseguraremos de que el sistema que estamos tratando de construir tenga sentido para la organización para la que lo estamos construyendo.

Parte de lo que cubrimos aquí se refirió a los desafíos de trabajar con organizaciones a gran escala. Sin embargo, hay otras consideraciones técnicas de las que debemos preocuparnos cuando nuestros sistemas comienzan a crecer más allá de unos pocos servicios discretos. Abordaremos esos 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 de un comienzo más simple y humilde a algo más complejo?

¿Qué sucede cuando tenemos que manejar la falla de múltiples servicios separados o administrar cientos de servicios? ¿Cuáles son algunos de los patrones de afrontamiento cuando tiene más microservicios que personas? Vamos a averiguar.

El fracaso está en todas partes

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

Incluso para aquellos de nosotros que no pensamos en una escala extrema, si podemos aceptar la posibilidad de fallar, estaremos mejor. Por ejemplo, si podemos manejar la falla de un servicio con gracia, entonces también podemos hacer actualizaciones de un servicio en el lugar, ya que una interrupción planificada es mucho más fácil de manejar que una no planificada.

También podemos pasar un poco menos de nuestro tiempo tratando de detener lo inevitable y un poco más de nuestro tiempo lidiando con eso con gracia. Estoy asombrado de cuántas organizaciones implementan procesos y controles para tratar de evitar que ocurran fallas, pero ponen poco o nada en pensar en hacer que sea más fácil recuperarse de las fallas en primer lugar.

Hornear en la suposición de que todo puede fallar y fallará lo lleva a pensar de manera diferente sobre cómo resuelve los problemas.

Vi un ejemplo de este pensamiento cuando pasé un tiempo en el campus de Google hace muchos años. En el área de recepción de uno de los edificios en Mountain View había un viejo estante de máquinas, allí como una especie de exhibición. Me di cuenta de un par de cosas. En primer lugar, estos servidores no estaban en gabinetes de servidores, eran simplemente placas base insertadas en el bastidor. 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. "Oh", dijo, "las unidades de disco duro fallan tanto que no las queremos enroscar. Simplemente las arrancamos, las tiramos a la basura y colocamos con velcro una nueva".

Así que permítanme repetir: a escala, incluso si compra el mejor kit, el hardware más caro, no puede evitar el hecho de que las cosas pueden fallar y fallarán. Por lo tanto, debe asumir que puede ocurrir una falla. Si incorpora este pensamiento en todo lo que hace y planifica el fracaso, puede hacer diferentes compensaciones. Si sabe que su sistema puede manejar el hecho de que un servidor puede fallar y fallará, ¿por qué molestarse en gastar mucho en él? ¿Por qué no usar una placa base simple con componentes más baratos (y algo de velcro) como hizo Google, en lugar de preocuparse demasiado por la resistencia de un solo nodo?

¿Cuánto es demasiado?

Tocamos el tema de los requisitos multifuncionales en el [Capítulo 7](#). La comprensión de los requisitos multifuncionales consiste en considerar 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 cubiertas en este capítulo y en otros lugares hablan sobre 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 un aumento de la carga o la falla de los nodos individuales puede ser fantástico, pero podría ser excesivo para un sistema de informes que solo necesita ejecutarse dos veces al mes, donde estar inactivo durante uno o dos días no es tan importante. un trato. 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 la base de conocimientos de su intranet corporativa probablemente sea un paso demasiado lejos.

Saber cuántas fallas puede tolerar o qué tan rápido debe ser su sistema depende de los usuarios de su sistema. Eso, a su vez, lo ayudará a comprender qué técnicas tendrán más sentido para usted. Dicho esto, sus usuarios no siempre podrán articular cuáles son los requisitos exactos. Por lo tanto, debe hacer preguntas para ayudar a extraer la información correcta y ayudarlos a comprender los costos relativos de brindar diferentes niveles de servicio.

Como mencioné anteriormente, estos requisitos interfuncionales pueden variar de un servicio a otro, pero sugeriría definir algunos interfuncionales generales y luego anularlos para casos de uso particulares. Cuando se trata de considerar si escalar horizontalmente su sistema y cómo hacerlo para manejar mejor la carga o la falla, comience tratando de comprender los siguientes requisitos:

Tiempo de *respuesta/latencia*

¿Cuánto tiempo deben tomar varias operaciones? Puede ser útil aquí medir esto con diferentes números de usuarios para comprender cómo el aumento de la carga afectará el tiempo de respuesta. Dada la naturaleza de las redes, siempre tendrá valores atípicos, por lo que puede ser útil establecer objetivos para un percentil dado de las respuestas monitoreadas. El objetivo también debe incluir la cantidad de conexiones/usuarios simultáneos que espera que maneje su software. Por lo tanto, podría decir: "Esperamos que el sitio web tenga un tiempo de respuesta del percentil 90 de 2 segundos al manejar 200 conexiones simultáneas por segundo".

Disponibilidad

¿Puede esperar que un servicio esté caído? ¿Se considera un servicio 24/7? A algunas personas les gusta ver los períodos de tiempo de inactividad aceptable cuando miden la disponibilidad, pero ¿qué tan útil es esto para alguien que llama a su servicio? Debería poder confiar en que su servicio responda 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? ¿Cuánto tiempo se deben conservar los datos? Es muy probable que esto cambie caso por caso. Por ejemplo, puede optar por mantener los registros de sesión de usuario durante un año o menos para ahorrar espacio, pero es posible que deba conservar sus registros de transacciones financieras durante muchos años.

Una vez que tenga estos requisitos establecidos, querrá una forma de medirlos sistemáticamente de forma continua. Puede decidir hacer uso de pruebas de rendimiento, por ejemplo, para asegurarse de que su sistema cumpla con los objetivos de rendimiento aceptables, ¡pero querrá asegurarse de que también está monitoreando estas estadísticas en producción!

Funcionalidad degradante

Una parte esencial de la creación de un sistema resiliente, especialmente cuando su funcionalidad se distribuye en varios microservicios diferentes que pueden estar activos 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 diversas partes de ese sitio web, es posible que necesitemos varios microservicios para desempeñar un papel. Un microservicio puede 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, lo que puede ser otro microservicio. Ahora bien, si uno de esos servicios no funciona y eso da como resultado que toda la página web no esté disponible, entonces posiblemente hemos creado un sistema que es menos resistente que uno que requiere que solo un servicio esté disponible.

Lo que debemos 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 podríamos mostrar la página web con el listado. Tal vez solo ocultemos el carrito de compras o lo reemplacemos con un ícono que diga "¡Vuelve pronto!"

Con una sola aplicación monolítica, no tenemos muchas decisiones que tomar. El estado del sistema es binario. Pero con una arquitectura de microservicio, debemos considerar una situación mucho más matizada. Lo correcto en cualquier situación a menudo no es una decisión técnica.

Es posible que sepamos lo que es técnicamente posible cuando el carrito de la compra está inactivo, pero a menos que entendamos el contexto comercial, no entenderemos qué acción debemos tomar.

Por ejemplo, tal vez cerremos todo el sitio, sigamos permitiendo que las personas naveguen 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.

Al pensar en la importancia de cada una de nuestras capacidades en términos de nuestros requisitos multifuncionales, estaremos mucho mejor posicionados para saber qué podemos hacer. Ahora, consideremos algunas cosas que podemos hacer desde un punto de vista técnico para asegurarnos de que cuando ocurra una falla, podamos manejarla correctamente.

Medidas de seguridad arquitectónica

Hay algunos patrones, a los que colectivamente me refiero como *medidas arquitectónicas de seguridad*, que podemos utilizar para asegurarnos de que si algo sale mal, no cause efectos desagradables. Estos son puntos que es esencial que entienda, y debe considerar seriamente la estandarización en su sistema para garantizar que un mal ciudadano no haga que el mundo entero se derrumbe alrededor de sus oídos. En un momento, veremos algunas medidas de seguridad clave que debe considerar, pero antes de hacerlo, me gustaría compartir una breve historia para describir el tipo de cosas que pueden salir mal.

Fui líder técnico en un proyecto en el que estábamos construyendo un sitio web de anuncios clasificados en línea. El sitio web en sí manejó volúmenes bastante altos y generó una gran cantidad de ingresos para el negocio. Nuestra aplicación central manejó parte de la visualización de anuncios clasificados y también transmitió llamadas a otros servicios que brindaban diferentes tipos de productos, como se muestra en la [Figura 11-1](#). En realidad, este es un ejemplo de una *aplicación estranguladora*, donde un nuevo sistema intercepta las llamadas realizadas a las aplicaciones heredadas y las reemplaza gradualmente por completo. Como parte de este proyecto, estábamos a medio camino del retiro de las aplicaciones más antiguas. Acabábamos de pasar al producto de mayor volumen y mayores ganancias, pero gran parte del resto de los anuncios todavía se publicaban en varias aplicaciones más antiguas. Tanto en términos de número de búsquedas como de dinero generado por estas aplicaciones, hubo una cola muy larga.

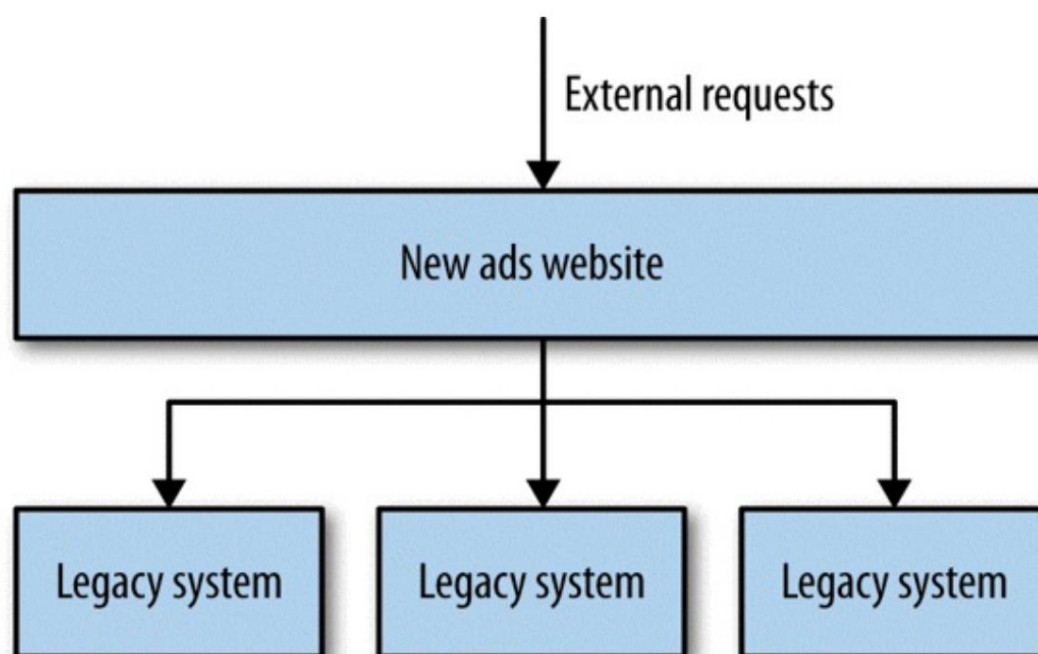


Figura 11-1. Un sitio web de anuncios clasificados que estrangula aplicaciones heredadas más antiguas

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

Una mañana, justo antes de llegar a nuestro pico diario a la hora del almuerzo, el sistema comenzó a comportarse lentamente y luego comenzó a fallar gradualmente. Tuvimos cierto nivel de monitoreo en nuestra nueva aplicación central, suficiente para decirnos que cada uno de nuestros nodos de aplicación estaba alcanzando un pico de CPU del 100 %, muy por encima de los niveles normales, incluso en el punto máximo. En un corto período de tiempo, todo el sitio se cayó.

Logramos rastrear al culpable y recuperar el sitio. Resultó que uno de los sistemas de anuncios posteriores, 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 allí, lo descubre bastante rápido. Cuando es *lento*, terminas esperando un rato antes de darte por vencido. Pero cualquiera que sea la causa de la falla, habíamos creado un sistema que era vulnerable a una falla en cascada. Un servicio posterior, sobre el cual teníamos poco control, pudo dismantelar todo nuestro sistema.

Mientras un equipo analizaba los problemas con el sistema descendente, el resto de nosotros comenzamos a analizar qué había fallado en nuestra aplicación. Encontramos algunos problemas. Estábamos usando un grupo de conexiones HTTP para manejar nuestras conexiones descendentes. Los subprocesos en el grupo en sí tenían tiempos de espera configurados por cuánto tiempo esperarían al realizar la llamada HTTP descendente, lo cual es bueno. El problema era que todos los trabajadores tardaban un poco en desconectarse debido a la lentitud del sistema aguas abajo. Mientras esperaban, más solicitudes fueron al grupo solicitando subprocesos de trabajo. Sin trabajadores disponibles, estas solicitudes colgaron. Resultó que la biblioteca del grupo de conexiones que estábamos usando tenía un tiempo de espera para los trabajadores, ¡pero esto estaba *deshabilitado de forma predeterminada!* Esto condujo a una gran acumulación de hilos bloqueados. Nuestra aplicación normalmente tenía 40 conexiones simultáneas en un momento dado. En el espacio de cinco minutos, esta situación hizo que llegáramos a un máximo de alrededor de 800 conexiones, lo que hizo que el sistema se cayera.

Lo que era peor era que el servicio descendente con el que hablábamos representaba una funcionalidad que usaba menos del 5 % de nuestra base de clientes y generaba incluso menos ingresos que eso. Cuando se llega al fondo, descubrimos de la manera difícil que los sistemas que simplemente actúan lentamente son *mucho* más difíciles de manejar que los sistemas que simplemente fallan rápidamente. En un sistema distribuido, la latencia mata.

Incluso si hubiéramos configurado correctamente los tiempos de espera en el grupo, también compartíamos un único grupo de conexiones HTTP para todas las solicitudes salientes. Esto significaba que un servicio lento podría agotar la cantidad de trabajadores disponibles por sí solo, incluso si todo lo demás estaba en buen estado. Por último, estaba claro que el servicio descendente en cuestión no estaba en buenas condiciones, pero seguimos enviando tráfico hacia él. En nuestra situación, esto significaba que en realidad estábamos empeorando una mala situación, ya que el servicio descendente no tenía posibilidad de recuperarse. Terminamos implementando tres soluciones para evitar que esto volviera a suceder: acertar con nuestros *tiempos* de espera , implementar *mamparos* para separar diferentes grupos de conexiones e implementar un *interruptor de circuito* para evitar enviar llamadas a un sistema en mal estado en primer lugar.

La Organización Antifrágil

En su libro *Antifragile* (Random House), Nassim Taleb habla de cosas que realmente se benefician del fracaso y el desorden. Ariel Tseitlin usó este concepto para acuñar el concepto de **organización antifrágil** con respecto a cómo opera Netflix.

La escala a la que opera Netflix es bien conocida, al igual que el hecho de que Netflix se basa completamente en la infraestructura de AWS. Estos dos factores hacen que tenga que afrontar bien el fracaso. Netflix va más allá al *incitar* al fracaso para garantizar que sus sistemas lo toleren.

Algunas organizaciones estarían contentas con *los días de juego*, donde la falla se simula 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 con regularidad. Google va más allá de las simples pruebas para imitar la falla del servidor y, como parte de sus **ejercicios anuales DiRT (Disaster Recovery Test)**, 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 diariamente.

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

Aceptar e incitar al 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 del fracaso cuando ocurre y de adoptar una cultura intachable cuando se cometen errores.

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

Al hacer que ocurran fallas y construir para ellas, Netflix se ha asegurado de que los sistemas que tiene se escalen mejor y respalden mejor las necesidades de sus clientes.

No todo el mundo necesita ir a los tipos de extremos que hacen Google o Netflix, pero es importante comprender el cambio de mentalidad que se requiere con los sistemas distribuidos. Las cosas fallarán. El hecho de que su sistema ahora esté distribuido en varias 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. Entonces, independientemente de si está tratando de brindar un servicio a la escala de Google o Netflix, prepárese para los tipos de fallas que ocurren con

más arquitecturas distribuidas es bastante importante. Entonces, ¿qué debemos hacer 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 posterior es importante corregirlos.

¿Cuánto tiempo puedo esperar antes de que pueda considerar que un sistema descendente realmente está inactivo?

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

El tiempo de espera se agota demasiado rápido y considerará que una llamada que podría haber funcionado falló. No tenga tiempos de espera en absoluto, y un sistema descendente que esté inactivo podría bloquear todo su sistema.

Ponga tiempos de espera en todas las llamadas fuera de proceso y elija un tiempo de espera predeterminado para todo.

Registre cuándo ocurren los tiempos de espera, observe lo que sucede y cámbielos en consecuencia.

Rompedores de circuito

En su propia casa, existen disyuntores para proteger sus dispositivos eléctricos de los picos de energía. Si ocurre un pico, el disyuntor se quema, protegiendo sus costosos electrodomésticos. También puede deshabilitar manualmente un disyuntor para cortar la energía a una parte de su hogar, lo que le permite trabajar de manera segura con la electricidad. El libro de Michael Nygard ¡Suéltalo ! (Programadores pragmáticos) 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 publicidad heredada descendente estaba respondiendo muy lentamente, antes de que finalmente devolviera un error. Incluso si hubiéramos acertado con los tiempos de espera, estaríamos esperando mucho tiempo antes de recibir el error. Y luego lo intentaríamos de nuevo la próxima vez que llegara una solicitud, y esperaríamos. Ya es bastante malo que el servicio descendente esté funcionando mal, pero también nos está haciendo ir lento.

Con un disyuntor, después de que falle un cierto número de solicitudes al recurso de flujo descendente, el disyuntor se funde. Todas las solicitudes posteriores fallan rápidamente mientras el disyuntor está en su estado quemado. Después de un cierto período de tiempo, el cliente envía algunas solicitudes para ver si el servicio descendente se ha recuperado y, si obtiene suficientes respuestas saludables, restablece el interruptor de circuito. Puede ver una descripción general de este proceso en la [Figura 11-2](#).

La forma en que implementa un disyuntor depende de lo que significa una solicitud *fallida* , pero cuando los implementé para conexiones HTTP, consideré que el error significaba 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 alcanzar un cierto umbral, automáticamente dejamos de enviar tráfico y comenzamos a fallar rápidamente. Y podemos comenzar automáticamente de nuevo cuando las cosas estén saludables.

Obtener la configuración correcta puede ser un poco complicado. No desea quemar el disyuntor con demasiada facilidad, ni tampoco desea tomar demasiado tiempo para quemarlo. Del mismo modo, realmente desea asegurarse de que el servicio descendente vuelva a estar en buen estado antes de enviar tráfico. Al igual que con los tiempos de espera, elegiría algunos valores predeterminados razonables y los mantendría en todas partes, luego los cambiaría para casos específicos.

Mientras el disyuntor está fundido, tiene algunas opciones. Una es poner en cola las solicitudes y volver a intentarlo más tarde. Para algunos casos de uso, esto podría ser apropiado, especialmente si está 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 la propagación de un error en la cadena de llamadas o una degradación más sutil de la funcionalidad.

Si contamos con este mecanismo (como los interruptores automáticos de nuestra casa), podríamos usarlos manualmente para que sea más seguro realizar nuestro trabajo. Por ejemplo, si quisiéramos desactivar un microservicio como parte del mantenimiento de rutina, podríamos activar manualmente todos los disyuntores de los sistemas dependientes para que fallen rápidamente mientras el microservicio está fuera de línea. Una vez que regrese, podemos restablecer los disyuntores y todo debería volver a la normalidad.

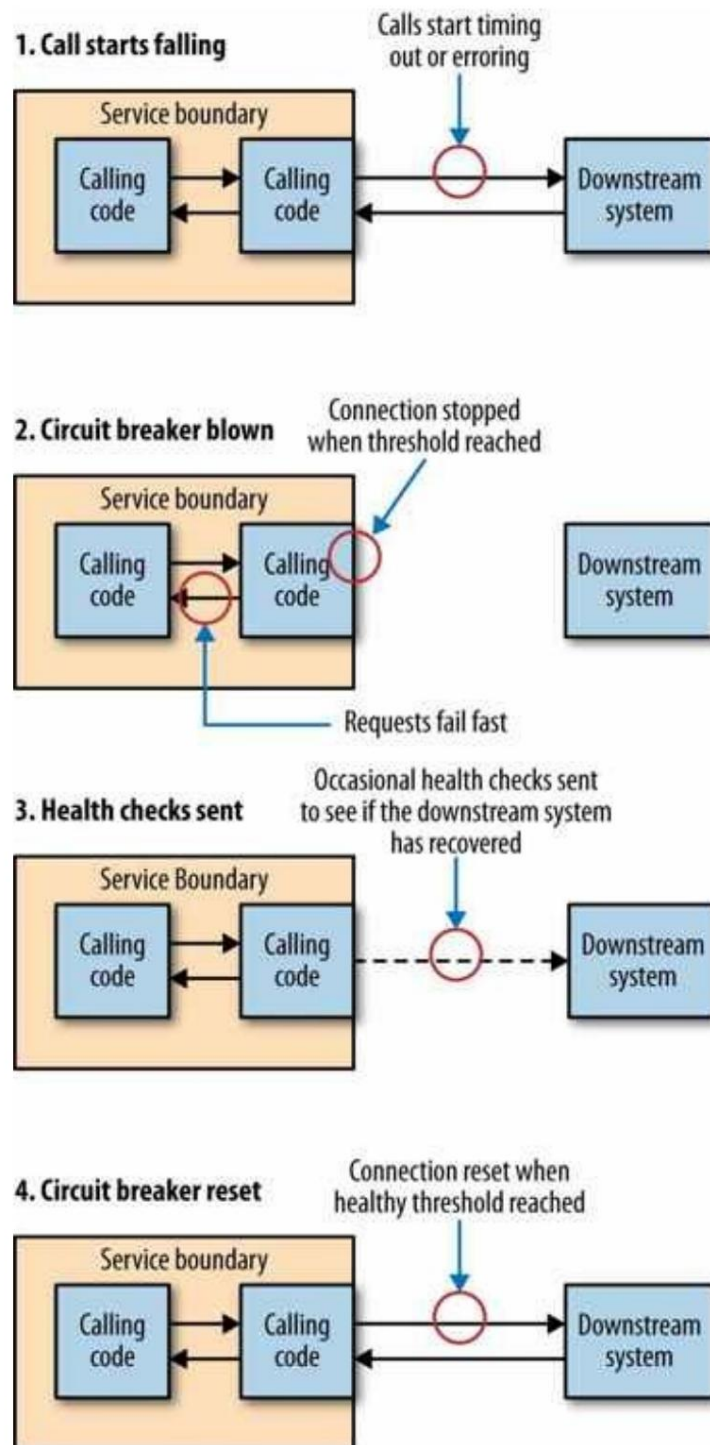


Figura 11-2. Una descripción general de los interruptores automáticos

Mamparos

En otro patrón de *Release It!*, Nygard introduce el concepto de *mamparo* como una forma de aislarse del fracaso. En el transporte marítimo, un mamparo es una parte del barco que se puede sellar para proteger el resto del barco. Entonces, si el barco tiene una fuga, puede cerrar las puertas de los mamparos. Pierdes 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 usado diferentes grupos de conexiones para cada conexión descendente. De esa forma, si un grupo de conexiones se agota, las otras conexiones no se ven afectadas, como vemos en la **Figura 11-3**. Esto garantizaría que si un servicio descendente comenzara a comportarse lentamente en el futuro, solo se vería afectado un grupo de conexiones, lo que permitiría que otras llamadas continuaran con normalidad.

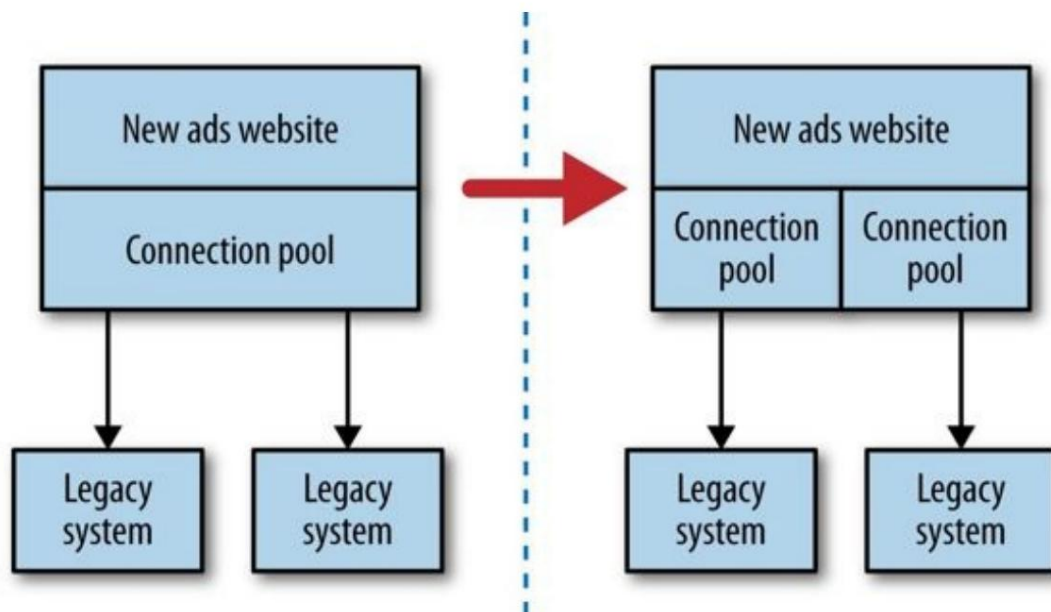


Figura 11-3. Uso de un grupo de conexiones por servicio descendente para proporcionar mamparos

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

Mire todos los aspectos de su sistema que pueden fallar, tanto dentro de sus microservicios como entre ellos. ¿Tiene mamparas en su lugar? Sugeriría comenzar con grupos de conexiones separados para cada conexión descendente como mínimo. Sin embargo, es posible que desee ir más allá y considerar el uso de disyuntores también.

Podemos pensar en nuestros disyuntores como un mecanismo automático para sellar un mamparo, no solo para proteger al consumidor del problema descendente, sino también para proteger potencialmente el servicio descendente de más llamadas que pueden tener un impacto adverso. Dados los peligros de fallas en cascada, recomendaría exigir disyuntores para todas sus llamadas síncronas descendentes. Tampoco tienes que escribir el tuyo propio. **Hystrix** de Netflix

La **biblioteca** es una abstracción de disyuntor de JVM que viene con una supervisión potente, pero existen otras implementaciones para diferentes pilas de tecnología, como **Polly para .NET** o la **combinación de disyuntor para Ruby**.

En muchos sentidos, los mamparos son los más importantes de estos tres patrones. Los tiempos de espera y los disyuntores lo ayudan a liberar recursos cuando se están limitando, pero los mamparos pueden garantizar que no se limiten en primer lugar. Hystrix le permite, por ejemplo, implementar mamparos que realmente rechacen solicitudes en ciertas condiciones para garantizar que los recursos no se saturen aún más; esto se conoce como deslastre de *carga*.

A veces, rechazar una solicitud es la mejor manera de evitar que un sistema importante se sobrecargue y se convierta en un cuello de botella para múltiples servicios ascendentes.

Aislamiento

Cuanto más depende un servicio de que otro esté activo, más afecta la salud de uno a la capacidad del otro para hacer su trabajo. Si podemos usar técnicas de integración que permitan que un servidor descendente esté fuera de línea, es menos probable que los servicios ascendentes se vean afectados por interrupciones, planificadas o no planificadas.

Hay otro beneficio 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. Cuanta menos coordinación se necesite entre los equipos, más autonomía tendrán esos equipos, 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 un impacto adverso. Esto es muy útil cuando queremos reproducir mensajes que no estamos seguros de que hayan sido procesados, una forma común de recuperarse de un error.

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

Ejemplo 11-1. Acreditar puntos en una cuenta

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

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

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

```
<crédito>
  <cantidad>100</cantidad>
  <forAccount>1234</account>
  <motivo>
    <paraCompra>4567</paraCompra> </razón> </
crédito>
```

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 tiene varias instancias del mismo tipo de servicio suscribiéndose a eventos.

Incluso si almacenamos qué eventos se han procesado, con algunas formas de entrega de mensajes asíncronos puede haber pequeñas ventanas donde 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 quedan atrapadas con este concepto y asumen 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 monitorear. 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 que sean idempotentes, pero para que ese sea el caso, dependen de que su servicio maneje estas llamadas en

una manera idempotente. Si comienza a hacer estos verbos no idempotentes, pero las personas que llaman creen que pueden ejecutarlos repetidamente de manera segura, es posible que se meta en un lío. ¡Recuerde, el hecho de que esté usando HTTP como protocolo subyacente no significa que obtenga todo gratis!

Escalada

Escalamos nuestros sistemas en general por una de dos razones. Primero, para ayudar a lidiar con el fracaso: si nos preocupa que algo falle, entonces tener más de eso ayudará, ¿verdad? En segundo lugar, escalamos para el rendimiento, ya sea en términos de manejar más carga, reducir la latencia o ambos. Veamos algunas técnicas de escalado comunes que podemos usar y pensemos cómo se aplican a las arquitecturas de microservicios.

ir más grande

Algunas operaciones pueden beneficiarse de más gruñido. Obtener una caja 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 escalado, a menudo llamada *escalado 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 comienza a utilizar máquinas realmente grandes.

A veces, nuestro software en sí mismo no puede hacer mucho con los recursos adicionales disponibles.

Las máquinas más grandes a menudo solo nos dan más núcleos de CPU, pero nuestro software no está escrito lo suficiente como para aprovecharlos. El otro problema es que esta forma de escalado puede no hacer mucho para mejorar la resiliencia de nuestro servidor si solo tenemos uno de ellos. No obstante, esto puede ser una buena victoria rápida, especialmente si está utilizando un proveedor de virtualización que le 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 multiservicio por host. Inicialmente, sin embargo, muchas personas deciden coexistir múltiples microservicios en una caja para mantener bajos los costos o simplificar la administración del host (aunque esa es una razón discutible). Dado que los microservicios son procesos independientes que se comunican a través de la red, debería ser una tarea fácil trasladarlos a sus propios hosts para mejorar el rendimiento y la escalabilidad. Esto también puede aumentar la resiliencia del sistema, ya que una sola interrupción del host afectará a una cantidad reducida de microservicios.

Por supuesto, también podríamos usar la necesidad de una mayor escala para dividir un microservicio existente en partes para manejar mejor la carga. Como ejemplo simple, imaginemos que nuestro servicio de cuentas brinda la capacidad de crear y administrar cuentas financieras de clientes individuales, pero también expone una API para ejecutar consultas para generar informes. Esta capacidad de consulta supone una carga significativa para 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 fundamental y no podemos permitirnos que se caiga. Al dividir estas dos capacidades en servicios separados, reducimos la carga en el servicio de cuentas críticas e introducimos un nuevo servicio de informes de cuentas que está diseñado no solo teniendo en cuenta las consultas (quizás usando algunas de las técnicas que describimos en el [Capítulo 4](#), sino también ya que un sistema no crítico no necesita implementarse de una manera tan resistente como el servicio de cuentas central.

Difundir su 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 que no tenga varios servicios en un host, donde una interrupción afectaría a varios servicios. Pero consideremos qué significa *host*.

En la mayoría de las situaciones hoy en día, un *host* es en realidad un concepto virtual. Entonces, ¿qué sucede si tengo todos mis servicios en diferentes hosts, pero todos esos hosts son en realidad hosts virtuales que se ejecutan en la misma caja física? Si ese cuadro se cae, podría perder varios servicios. Algunas plataformas de virtualización le permiten asegurarse de que sus hosts estén distribuidos en varias cajas físicas diferentes para reducir esta posibilidad.

Para las plataformas de virtualización interna, es una práctica común tener la partición raíz de la máquina virtual asignada a una sola SAN (red de área de almacenamiento). Si esa SAN falla, puede desactivar todas las máquinas virtuales conectadas. Las SAN son grandes, costosas y están diseñadas para no fallar. Dicho esto, me han fallado SAN grandes y costosas 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 sus servicios se ejecuten en un solo bastidor en el centro de datos, o que sus servicios se distribuyan en más de un centro de datos. Si está utilizando 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 estén inactivos durante no 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 encontrar una solución alternativa.

AWS, por ejemplo, se divide en regiones, que puede considerar nubes distintas. Cada región se divide a su vez en dos o más zonas de disponibilidad (AZ). Las zonas de disponibilidad son el equivalente de AWS a un centro de datos. Es fundamental tener servicios distribuidos en varias zonas de disponibilidad, ya que AWS no ofrece ninguna garantía sobre la disponibilidad de un solo nodo, o incluso de una zona de disponibilidad completa. Para su servicio informático, 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 lo suficientemente bueno y, en cambio, también ejecutan sus servicios en varias regiones.

Cabe señalar, por supuesto, que debido a que los proveedores le brindan una *garantía de SLA*, ¡tenderán a limitar su responsabilidad! Si no cumplir con sus objetivos le cuesta clientes y una gran cantidad de dinero, es posible que tenga que buscar en los contratos para ver si puede recuperar algo de ellos. Por lo tanto, le sugiero enfáticamente que comprenda el impacto de que un proveedor no cumpla con sus obligaciones para con usted y determine si necesita tener un plan B (o C) en su bolsillo. Más de un cliente con el que he trabajado ha tenido 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.

Balanceo de carga

Cuando necesita que su servicio sea resistente, desea evitar los puntos únicos de falla. Para un microservicio típico que expone un extremo HTTP síncrono, la forma más fácil de lograrlo es tener varios hosts ejecutando 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.

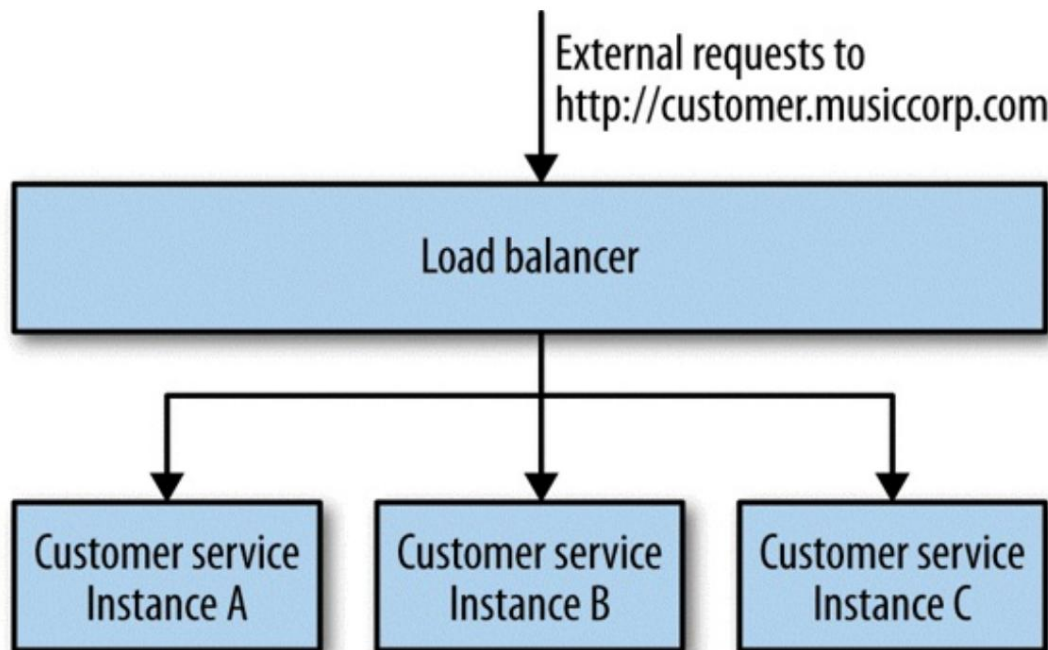


Figura 11-4. Un ejemplo de un enfoque de equilibrio de carga para escalar la cantidad de instancias de servicio al cliente

Los balanceadores 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 en función de algún algoritmo, eliminan las instancias cuando ya no están en buen estado y, con suerte, las vuelven a agregar cuando lo estén.

Algunos balanceadores de carga proporcionan características útiles. Una común es la *terminación SSL*, donde las conexiones HTTPS entrantes al balanceador de carga se transforman en conexiones HTTP una vez que llegan a la instancia misma. Históricamente, la sobrecarga de administrar SSL era lo suficientemente significativa como para que tener un equilibrador de carga que manejara este proceso por usted fuera bastante útil. Hoy en día, se trata tanto de 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 discutimos en el [Capítulo 9](#), por lo que si usamos la terminación SSL, potencialmente nos exponemos 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 el exterior solo pueden llegar a través de un enrutador, y en este caso nuestro enrutador también es nuestro balanceador de carga de terminación SSL. La única comunicación con el microservicio desde fuera de la VLAN se realiza a través de HTTPS, pero internamente todo es HTTP.

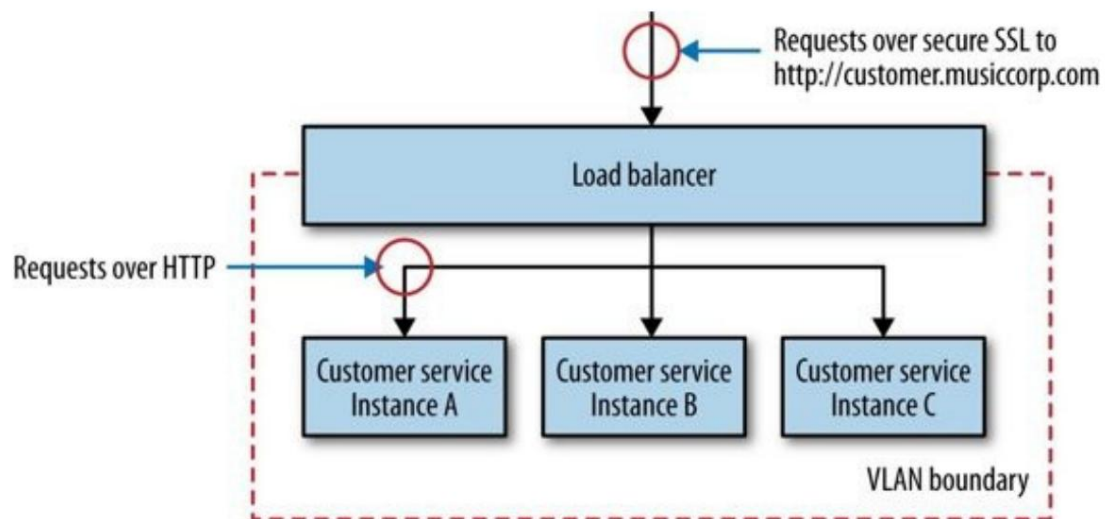


Figura 11-5. Uso de la terminación HTTPS en el balanceador de carga con una VLAN para mejorar la seguridad

AWS proporciona balanceadores de carga que terminan en HTTPS en forma de ELB (balanceadores de carga elásticos) y puede usar sus grupos de seguridad o nubes privadas virtuales (VPC) para implementar la VLAN. De lo contrario, el software como `mod_proxy` puede desempeñar un papel similar al de un equilibrador de carga de software. Muchas organizaciones tienen balanceadores de carga de hardware, que pueden ser difíciles de automatizar. En tales circunstancias, me encontré abogando por balanceadores de carga de software ubicados *detrás* de los balanceadores de carga de hardware para permitir a los equipos la libertad de reconfigurarlos según sea necesario. ¡Desea observar el hecho de que, con demasiada frecuencia, los equilibradores de carga de hardware en sí mismos son puntos únicos de falla! Independientemente del enfoque que adopte, al considerar la configuración de un equilibrador de carga, trátelo como trata la configuración de su servicio: asegúrese de que esté almacenado en el control de versiones y que se pueda aplicar automáticamente.

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

Sistemas basados en trabajadores

El equilibrio de carga no es la única manera de tener varias instancias de su servicio para compartir la carga y reducir la fragilidad. Dependiendo de la naturaleza de las operaciones, un sistema basado en el trabajador podría ser igual de efectivo. Aquí, una colección de instancias todas funcionan en una acumulación de trabajo compartida. Esto podría ser una cantidad de procesos de Hadoop, o tal vez una cantidad de oyentes en una cola de trabajo compartida. Estos tipos de operaciones se adaptan bien al trabajo por lotes o trabajos asincrónicos. 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 para la *carga* máxima, donde puede generar instancias adicionales a pedido para que coincida con la carga entrante. Siempre que la cola de trabajo en sí sea resistente, este modelo se puede usar para escalar tanto para mejorar el rendimiento del trabajo, pero también para mejorar la resiliencia: el impacto de que un trabajador falle (o no esté allí) es fácil de manejar. El trabajo llevará más tiempo, pero nada se pierde.

He visto que esto funciona bien en organizaciones donde hay mucha capacidad de cómputo sin usar en ciertos momentos del día. Por ejemplo, de la noche a la mañana 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 generación de informes.

Con los sistemas basados en trabajadores, aunque los trabajadores mismos no necesitan ser tan confiables, el sistema que contiene el trabajo a realizar sí lo es. Podría manejar esto ejecutando un intermediario de mensajes persistentes, por ejemplo, o quizás un sistema como Zookeeper. El beneficio aquí es que si usamos el 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 lo inicia puede no ser la arquitectura que lo mantiene en funcionamiento cuando su sistema tiene que manejar 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), debe "diseñar para un crecimiento de ~10x, pero planear reescribir antes de ~100x". En ciertos puntos, debe hacer algo bastante radical para respaldar el siguiente nivel de crecimiento.

Recuerde la historia de Gilt, que mencionamos en el [Capítulo 6](#). Una aplicación Rails monolítica simple funcionó bien para Gilt durante dos años. Su negocio se volvió cada vez más exitoso, lo que significó más clientes y más carga. En cierto punto de inflexión, la empresa tuvo que rediseñar la aplicación para manejar la carga que estaba viendo.

Un rediseño puede significar dividir un monolito existente, como sucedió con Gilt. O podría significar elegir nuevos almacenes de datos que puedan manejar mejor la carga, que veremos en un momento. También podría significar adoptar nuevas técnicas, como pasar de una solicitud/respuesta síncrona a sistemas basados en eventos, adoptar nuevas plataformas de implementación, cambiar pilas de tecnología completas o todo lo demás.

Existe el peligro de que las personas vean la necesidad de rediseñar cuando se alcanzan ciertos umbrales de escala como una razón para construir a gran escala desde el principio. Esto puede ser desastroso. Al comienzo de un nuevo proyecto, a menudo no sabemos exactamente lo que queremos construir, ni sabemos si tendrá éxito. Necesitamos poder experimentar rápidamente y comprender qué capacidades necesitamos desarrollar. Si tratáramos de construir para una escala masiva desde el principio, terminaríamos cargando una gran cantidad de trabajo por adelantado para prepararnos para una carga que quizás nunca llegue, mientras desviamos el esfuerzo de actividades más importantes, como saber si alguien realmente querrá usar nuestro producto. Eric Ries cuenta la historia de pasar seis meses construyendo un producto que nadie descargó. Reflexionó que podría haber puesto un enlace en una página web que apareciera como 404 cuando la gente hacía clic en él para ver si había alguna demanda, ¡pasó seis meses en la playa y aprendió lo mismo!

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

Escalando bases de datos

Escalar microservicios sin estado es bastante sencillo. Pero, ¿y si estamos almacenando datos en una base de datos? Necesitaremos saber cómo escalar eso también. Los diferentes tipos de bases de datos brindan diferentes formas de escalado, y comprender qué forma se adapta mejor a su caso de uso garantizará que seleccione la tecnología de base de datos adecuada desde el principio.

Disponibilidad del servicio versus durabilidad de los datos

De inmediato, es importante separar el concepto de disponibilidad del servicio de la durabilidad de los datos en sí. Debe comprender que estas son dos cosas diferentes y, como tales, 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 un modo de espera. Todos los datos escritos en la base de datos principal se copian en la base de datos de réplica en espera. Si el principal falla, mis datos están seguros, pero sin un mecanismo para recuperarlos o promover la réplica al principal, no tenemos una base de datos disponible, aunque nuestros datos están seguros.

Escalado para lecturas

Muchos servicios son de lectura en su mayoría. Piense en un servicio de catálogo que almacene información de los artículos que tenemos a la venta. Agregamos registros para artículos nuevos de manera bastante irregular, y no sería sorprendente si obtenemos más de 100 lecturas de los datos de nuestro catálogo por cada escritura. Felizmente, escalar para lecturas es mucho más fácil que escalar para escrituras. El almacenamiento en caché de datos puede desempeñar un papel importante aquí, y lo discutiremos con más profundidad en breve. Otro modelo es hacer uso de *réplicas de lectura*.

En un sistema de administración 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 a menudo se hace para garantizar que una copia de nuestros datos se mantenga segura, pero también podemos usarla 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. Eventualmente, las lecturas verán los datos consistentes. Tal configuración se llama *eventualmente consistente*, y si puede manejar la inconsistencia temporal, es una forma bastante fácil y común de ayudar a escalar los sistemas. Veremos esto con más profundidad en breve cuando veamos el teorema CAP.

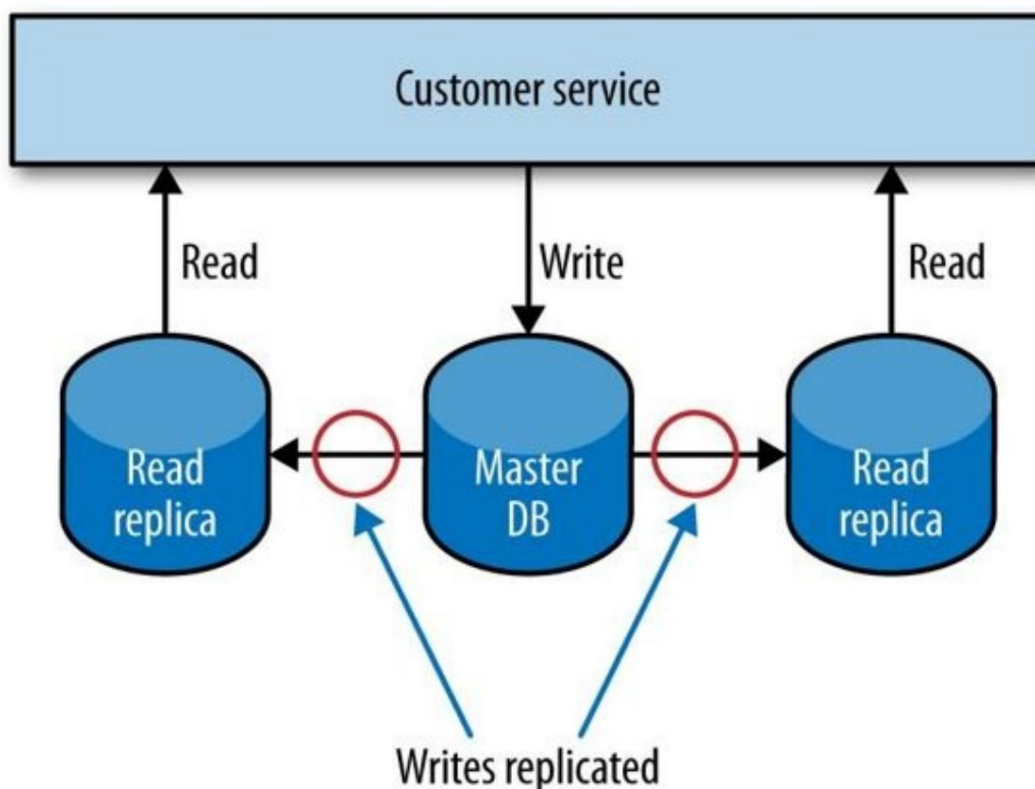


Figura 11-6. Uso de réplicas de lectura para escalar lecturas

Hace años, el uso de réplicas de lectura para escalar estaba de moda, aunque hoy en día le sugiero que busque primero 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 pasa con las escrituras? Un enfoque es usar *fragmentación*. Con la fragmentación, tiene varios nodos de base de datos. Toma un dato para escribirlo, aplica alguna función hash a la clave de los datos y, en función del resultado de la función, aprende a dónde enviar los datos. Para elegir un ejemplo muy simple (y realmente malo), imagine que los registros de clientes A–M van a una instancia de la base de datos y N–Z a otra. Puede administrar esto usted mismo en su aplicación, pero algunas bases de datos, como Mongo, manejan gran parte por usted.

La complejidad de la fragmentación para escrituras proviene del manejo de consultas. Buscar un registro individual es fácil, ya que solo puedo aplicar la función hash para encontrar en qué instancia deben estar los datos y luego recuperarlos del fragmento correcto. Pero, ¿qué pasa con las consultas que abarcan los datos en varios nodos, por ejemplo, encontrar todos los clientes mayores 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 asíncrono que usa resultados almacenados en caché.

Mongo utiliza trabajos de asignación/reducción, por ejemplo, para realizar estas consultas.

Una de las preguntas que surge con los sistemas fragmentados es, ¿qué sucede si quiero agregar un nodo de base de datos adicional? En el pasado, esto a menudo requería un tiempo de inactividad significativo, especialmente para clústeres grandes, ya que es posible que deba desactivar toda la base de datos y reequilibrar los datos. Más recientemente, más sistemas admiten agregar fragmentos adicionales a un sistema en vivo, donde el reequilibrio de datos ocurre 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 probarlo a fondo.

La fragmentación para escrituras puede escalar para el volumen de escritura, pero es posible que no mejore la resiliencia. Si los registros de clientes 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 aquí, donde podemos garantizar que los datos se repliquen en múltiples nodos en un *anillo* (término de Cassandra para una colección de nodos de Cassandra).

Como puede haber inferido de esta breve descripción general, escalar bases de datos para escrituras es donde las cosas se vuelven muy complicadas y donde las capacidades de las diversas bases de datos realmente comienzan a diferenciarse. A menudo veo personas que cambian la tecnología de la base 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 caja más grande suele ser la forma más rápida de resolver el problema, pero en el fondo, es posible que desee ver sistemas como Cassandra, Mongo o Riak para ver si sus modelos de escala alternativos pueden ofrecerle una mejor solución de largo plazo.

Infraestructura de base de datos compartida

Algunos tipos de bases de datos, como el RDBMS tradicional, 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 importante punto único de falla. Si la infraestructura de esta base de datos deja de funcionar, 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 Command-Query Responsibility Segregation (CQRS) hace referencia a un modelo alternativo para almacenar y consultar información. Con las bases de datos normales, usamos un sistema para realizar modificaciones a los datos y consultar los datos. Con CQRS, 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 llegan solicitando cambios de estado. Estos comandos se validan y, si funcionan, se aplicarán al modelo. Los comandos deben contener información sobre su intención. Se pueden procesar de forma síncrona o asíncrona, lo que permite que diferentes modelos manejen el escalado; podríamos, por ejemplo, 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 están completamente separados. Por ejemplo, podría elegir manejar y procesar comandos como eventos, tal vez solo 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 del dominio, o podría simplemente tomar una fuente de la parte de comando del sistema para actualizar un tipo diferente de almacén. En muchos sentidos, obtenemos los mismos beneficios de las réplicas de lectura que discutimos 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 diferentes tipos de escalado. Las partes de comando y consulta de nuestro sistema podrían vivir en diferentes servicios, o en diferentes hardware, y podrían hacer uso de tipos de almacenamiento de datos radicalmente diferentes. Esto puede desbloquear una gran cantidad de formas de manejar la escala. Incluso podría admitir diferentes tipos de formato de lectura al tener múltiples implementaciones de la pieza de consulta, tal vez admitiendo una representación basada en gráficos de sus datos, o una forma de sus datos basada en clave/valor.

Sin embargo, tenga cuidado: este tipo de patrón se aleja bastante de un modelo en el que un solo almacén de datos maneja todas nuestras operaciones CRUD. ¡He visto a más de un equipo de desarrollo experimentado luchar para obtener este patrón correctamente!

almacenamiento 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 usar este valor almacenado en lugar de gastar tiempo y recursos en volver a calcular el valor. La mayoría de las veces, el almacenamiento en caché se trata de eliminar los viajes de ida y vuelta innecesarios a las bases de datos u otros servicios para brindar resultados más rápido. Si se usa bien, puede generar enormes beneficios de rendimiento. La razón por la que HTTP escala tan bien en el manejo de grandes cantidades de solicitudes es que el concepto de almacenamiento en caché está integrado.

Incluso con una aplicación web monolítica simple, hay bastantes opciones en cuanto a 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 que tomar sobre dónde y cómo almacenar en caché.

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

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

En el almacenamiento en caché del lado del cliente, el cliente almacena el resultado almacenado en caché. El cliente puede decidir cuándo (y si) va y recupera una copia nueva. Idealmente, el servicio descendente proporcionará sugerencias para ayudar al cliente a comprender qué hacer con la respuesta, para 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 usar 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.

Cuál tiene más sentido depende de lo que esté tratando de 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 desea realizar cambios en la forma en que se realiza el almacenamiento en caché, podría ser difícil implementar cambios en varios consumidores. La invalidación de datos obsoletos también puede ser más complicada, aunque discutiremos algunos mecanismos para hacer frente a esto en un momento.

Con el almacenamiento en caché de proxy, todo es opaco tanto para el cliente como para el servidor. A menudo, esta es una forma muy sencilla de agregar almacenamiento en caché a un sistema existente. Si el proxy está diseñado para almacenar en caché el 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 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 cosas como la invalidación de datos o rastrear y optimizar los aciertos de caché. En una situación en la que tiene varios tipos de clientes, una caché del lado del servidor podría ser la forma más rápida de mejorar el rendimiento.

Por cada sitio web público en el que he trabajado, hemos terminado haciendo una combinación de los tres enfoques. Pero para más de un sistema distribuido, me he salido con la mía sin almacenamiento en caché en absoluto. Pero todo se reduce a saber qué carga necesita manejar, qué tan actualizados deben estar sus datos y qué puede hacer su sistema en este momento. Saber que tiene varias herramientas diferentes a su disposición es solo el comienzo.

Almacenamiento en caché en HTTP

HTTP proporciona algunos controles realmente útiles para ayudarnos a almacenar en caché en el lado del cliente o en el lado del servidor, que vale la pena entender incluso si no está utilizando HTTP.

Primero, con HTTP, podemos usar directivas de control de caché en nuestras respuestas a los clientes. Estos le dicen a los clientes si deben almacenar en caché el recurso y, de ser así, cuánto tiempo deben almacenarlo en segundos. También tenemos la opción de configurar un encabezado Expires , donde en lugar de decir cuánto tiempo se puede almacenar en caché un contenido, especificamos una hora y fecha en la que un recurso debe considerarse obsoleto y recuperarse nuevamente. La naturaleza de los recursos que está compartiendo determina cuál es más probable que encaje. El contenido de sitio web estático estándar, como CSS o imágenes, a menudo encaja bien con 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, tendrá más sentido configurar un encabezado Expires . Todo esto es muy útil para evitar que un cliente necesite realizar una solicitud al servidor en primer lugar.

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

Por ejemplo, imaginemos que buscamos un registro de cliente y su ETag regresa como o5t6fkd2sa. Más adelante, tal vez 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 solicitud GET posterior, podemos pasar un If-None-Match: o5t6fkd2sa. Esto le dice al servidor que queremos el recurso en el URI especificado, a menos que ya coincida con este valor de ETag. Si ya tenemos la versión actualizada, el servicio nos envía una respuesta 304 No modificado , indicándonos 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 proxies inversos como Squid o Varnish pueden ubicarse de forma transparente en la red entre el cliente y el servidor, almacenando y expirando el contenido en caché según sea necesario. Estos sistemas están orientados a atender grandes cantidades 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 llama, asegurándose de que el tráfico no dé la vuelta al otro lado del mundo cuando sea necesario. Y más prosaicamente, las bibliotecas de clientes HTTP y las cachés de clientes pueden manejar gran parte de este trabajo por nosotros.

ETags, Expires y el control de caché pueden superponerse un poco, y si no tiene cuidado, puede terminar dando información contradictoria si decide usarlos todos. Para una más profunda

discusión 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.

Ya sea que decida usar HTTP como un 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 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 utiliza un caché de escritura posterior, puede escribir en un caché local y, en algún momento posterior, los datos se descargarán en una fuente descendente, probablemente la fuente canónica de datos. Esto puede ser útil cuando tiene ráfagas de escrituras o cuando existe una buena posibilidad de que los mismos datos se escriban varias veces. Cuando se utilizan para almacenar en búfer y potencialmente escrituras por lotes, las cachés de escritura en segundo plano pueden ser una optimización adicional útil del rendimiento.

Con una caché de escritura en segundo plano, 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 la 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 servir datos obsoletos. Para algunos sistemas, estar disponible incluso con datos obsoletos es mejor que no devolver ningún resultado, pero esa es una decisión que tendrá que tomar. Obviamente, si no tenemos los datos solicitados en el caché, no podemos hacer mucho para ayudar, pero hay formas de mitigar esto.

Una técnica que vi utilizada en *The Guardian*, y posteriormente en otros lugares, fue rastrear el sitio *en vivo* existente periódicamente para generar una versión estática del sitio web que podría servir en caso de una interrupción. Aunque esta versión rastreada no era tan reciente como el contenido almacenado en caché servido desde el sistema en vivo, en un apuro podría garantizar que se mostraría una versión del sitio.

Ocultar el origen

Con una memoria caché normal, si una solicitud da como resultado una pérdida de memoria caché, la solicitud pasa al origen para obtener los datos nuevos con el bloqueo de la persona que llama, esperando el resultado. En el curso normal de las cosas, esto es de esperar. Pero si sufrimos una pérdida masiva de caché, tal vez porque falla una máquina completa (o un grupo de máquinas) que proporciona nuestro caché, una gran cantidad de solicitudes llegarán al origen.

Para aquellos servicios que brindan datos altamente almacenables en caché, es común que el propio origen se escale para manejar solo una fracción del tráfico total, ya que la mayoría de las solicitudes se atienden sin memoria por los cachés que se encuentran frente al origen. Si de repente tenemos una manada atronadora debido a la desaparición de una región de caché completa, nuestro origen podría desaparecer.

Una forma de proteger el origen en tal situación es nunca permitir que las solicitudes vayan al origen en primer lugar. En su lugar, el propio origen llena la memoria caché de forma asíncrona cuando es necesario, como se muestra en la **figura 11-7**. Si se produce una falta de caché, esto desencadena un evento que el origen puede detectar y lo alerta de que necesita volver a llenar el caché. Entonces, si un fragmento completo ha desaparecido, podemos reconstruir el caché en segundo plano. Podríamos decidir bloquear la solicitud original a la espera de que se vuelva a llenar la región, pero esto podría causar una contención en el propio caché, lo que generaría más problemas. Es más probable que si damos prioridad a mantener estable el sistema, fallemos en la solicitud original, pero fallará rápidamente.

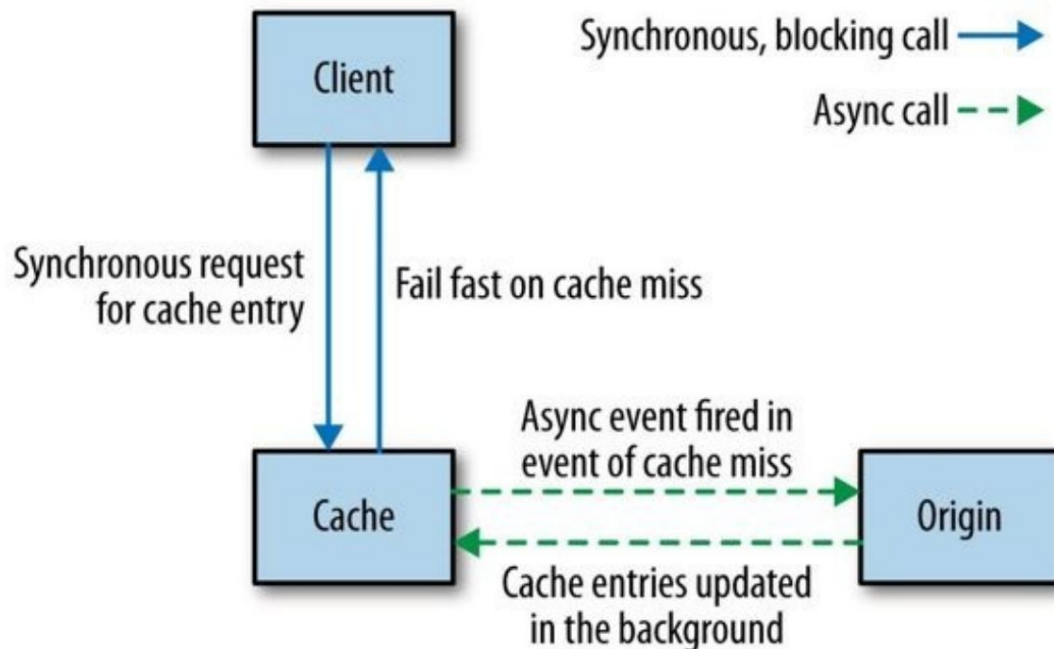


Figura 11-7. Ocultar el origen del cliente y llenar el caché de forma asíncrona

Este tipo de enfoque puede no tener sentido para algunas situaciones, pero puede ser una forma de garantizar que el sistema permanezca activo cuando fallan algunas partes. Al fallar las solicitudes rápidamente y asegurarnos de que no consumamos recursos ni aumentemos la latencia, evitamos que una falla en nuestro caché se transmita en cascada y nos damos la oportunidad de recuperarnos.

Mantenlo simple

¡Tenga cuidado con el almacenamiento en caché en demasiados lugares! Cuantas más cachés haya entre usted y la fuente de datos nuevos, más obsoletos pueden ser los datos y más difícil puede ser determinar la actualización de los datos que finalmente ve un cliente. Esto puede ser especialmente problemático con una arquitectura de microservicios en la que tiene múltiples servicios involucrados en una cadena de llamadas. Una vez más, cuanto más almacenamiento en caché tenga, más difícil será evaluar la frescura de cualquier dato. Entonces, si cree que un caché es una buena idea, manténgalo simple, quédese con uno y piense detenidamente antes de agregar más.

Envenenamiento de caché: una advertencia

Con el almacenamiento en caché, a menudo pensamos que si nos equivocamos, lo peor que puede pasar es que sirvamos datos obsoletos por un tiempo. Pero, ¿qué sucede si termina sirviendo datos obsoletos para siempre? Anteriormente mencioné el proyecto en el que trabajé en el que usábamos una aplicación estranguladora para ayudar a interceptar llamadas a múltiples sistemas heredados con miras a retirarlos gradualmente. Nuestra aplicación funcionó efectivamente como un proxy. El tráfico a nuestra aplicación se enrutaba a través de la aplicación heredada. En el camino de regreso, hicimos algunas cosas de limpieza; 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 introdujo un error por el cual un pequeño subconjunto de páginas caía a través de una condición lógica en nuestro código de inserción de encabezado de caché, lo que provocaba que no cambiáramos el encabezado en absoluto. Desafortunadamente, esta aplicación descendente también se había cambiado en algún momento anterior para incluir un encabezado Expires: Never HTTP. Esto no había tenido ningún efecto antes, ya que estábamos anulando este encabezado. Ahora no lo estábamos.

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

Como mencioné anteriormente, puede almacenar en caché en varios lugares. Cuando se trata de ofrecer contenido a los usuarios de una aplicación web pública, podría tener varios cachés entre usted y su cliente. No solo podría estar al frente de su sitio web con algo como un CDN, sino que algunos ISP utilizan el almacenamiento en caché. ¿Puedes controlar esos cachés?

E incluso si pudiera, hay un caché sobre el que tiene poco control: el caché en el navegador de un usuario.

Esas páginas con Expires: nunca se atascaron en los cachés de muchos de nuestros usuarios, y nunca se invalidarían hasta que el caché se llenara o el usuario las limpiara manualmente.

Claramente, no pudimos hacer que sucediera ninguna de las dos cosas; nuestra única opción era cambiar las URL de estas páginas para que se recuperaran.

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

Autoescalado

Si tiene la suerte de tener un aprovisionamiento completamente automatizable de hosts virtuales y puede automatizar completamente la implementación de sus instancias de microservicios, entonces tiene los elementos básicos que le permitirán escalar automáticamente sus microservicios.

Por ejemplo, también puede hacer que la escala se desencadene por tendencias conocidas. Es posible que sepa que la carga máxima de su sistema es entre las 9 a. m. y las 5 p. m., por lo que abre instancias adicionales a las 8:45 a. m. y las apaga a las 5:15 p. m. Si está utilizando algo como AWS (que tiene un muy buen soporte para el escalado automático incorporado), apagar las instancias que ya no necesita le ayudará a ahorrar dinero. Necesitará datos para comprender cómo cambia su carga a lo largo del tiempo, de un día a otro, de una semana a otra. Algunas empresas también tienen ciclos estacionales obvios, por lo que es posible que necesite datos que se remonten de manera justa para tomar decisiones de juicio adecuadas.

Por otro lado, podría ser reactivo, abriendo instancias adicionales cuando vea un aumento en la carga o una falla de instancia, y elimine instancias cuando ya no las necesite. Saber qué tan rápido puede escalar una vez que detecta una tendencia al alza es clave. Si sabe que solo recibirá un aviso de un par de minutos sobre un aumento en la carga, pero la ampliación le llevará al menos 10 minutos, sabe que necesitará mantener capacidad adicional para cerrar esta brecha. Tener un buen conjunto de pruebas de carga es casi esencial aquí. Puede usarlos para probar sus reglas de escalado automático. Si no tiene pruebas que puedan reproducir diferentes cargas que activarán el escalado, solo lo descubrirá en producción si se equivocó 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 escalabilidad predictiva y reactiva. En el último sitio de noticias en el que trabajé, vimos tendencias diarias muy claras, con vistas que subían desde la mañana hasta la hora del almuerzo y luego comenzaban a declinar.

Este patrón se repitió día tras día, con tráfico menos pronunciado durante el fin de semana.

Eso le dio una tendencia bastante clara que podría impulsar la ampliación (y reducción) proactiva de los recursos.

Por otro lado, una gran noticia provocaría un aumento inesperado, lo que requeriría más capacidad en un plazo a menudo corto.

De hecho, veo que el escalado automático se usa mucho más para manejar fallas de instancias que para reaccionar a las condiciones de carga. AWS le permite especificar reglas como "Debe haber al menos 5 instancias en este grupo", por lo que si una falla, se lanza una nueva automáticamente. He visto que este enfoque conduce a un divertido juego de golpear un topo cuando alguien se olvida de desactivar la regla y luego intenta eliminar las instancias para mantenimiento, ¡solo para ver que siguen girando!

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

Sugeriría usar el ajuste de escala automático para las condiciones de falla primero mientras recopila los datos. Una vez

desea comenzar a escalar para la carga, asegúrese de tener mucho cuidado con la reducción demasiado rápida. En la mayoría de las situaciones, tener más poder de cómputo en sus manos de lo que necesita es mucho mejor que no tener suficiente.

Teorema de la PAC

Nos gustaría tenerlo todo, pero lamentablemente sabemos que no podemos. Y cuando se trata de sistemas distribuidos como los que construimos usando 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 discusiones sobre los méritos de varios tipos diferentes de almacenes de datos. En esencia, nos dice que en un sistema distribuido, tenemos tres cosas que podemos compensar entre sí: *consistencia, disponibilidad y tolerancia a la partición*. Específicamente, el teorema nos dice que podemos mantener dos en modo de falla.

La consistencia es la característica del sistema por la cual obtendré la misma respuesta si voy a varios nodos. Disponibilidad significa que cada solicitud recibe una respuesta. La tolerancia de 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 conjetura original, la idea obtuvo una demostración matemática. No voy a sumergirme en las matemáticas de la prueba en sí, ya que no solo no es ese tipo de libro, sino que también puedo garantizar que me equivocaré. En su lugar, usemos algunos ejemplos resueltos que nos ayudarán a comprender que, bajo todo esto, el teorema CAP es una destilación de un conjunto de razonamiento muy lógico.

Ya hemos hablado de algunas técnicas simples de escalado de bases de datos. Usemos uno de estos para probar las ideas detrás del teorema CAP. Imaginemos que nuestro servicio de inventario se implementa en dos centros de datos separados, como se muestra en la **figura 11-8**. El respaldo de nuestra instancia de servicio en cada centro de datos es una base de datos, y estas dos bases de datos se comunican entre sí para intentar sincronizar datos entre ellas. Las lecturas y escrituras se realizan a través del nodo de la base de datos local y la replicación se utiliza para sincronizar los datos entre los nodos.

Ahora pensemos en lo que sucede cuando algo falla. Imagine 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?

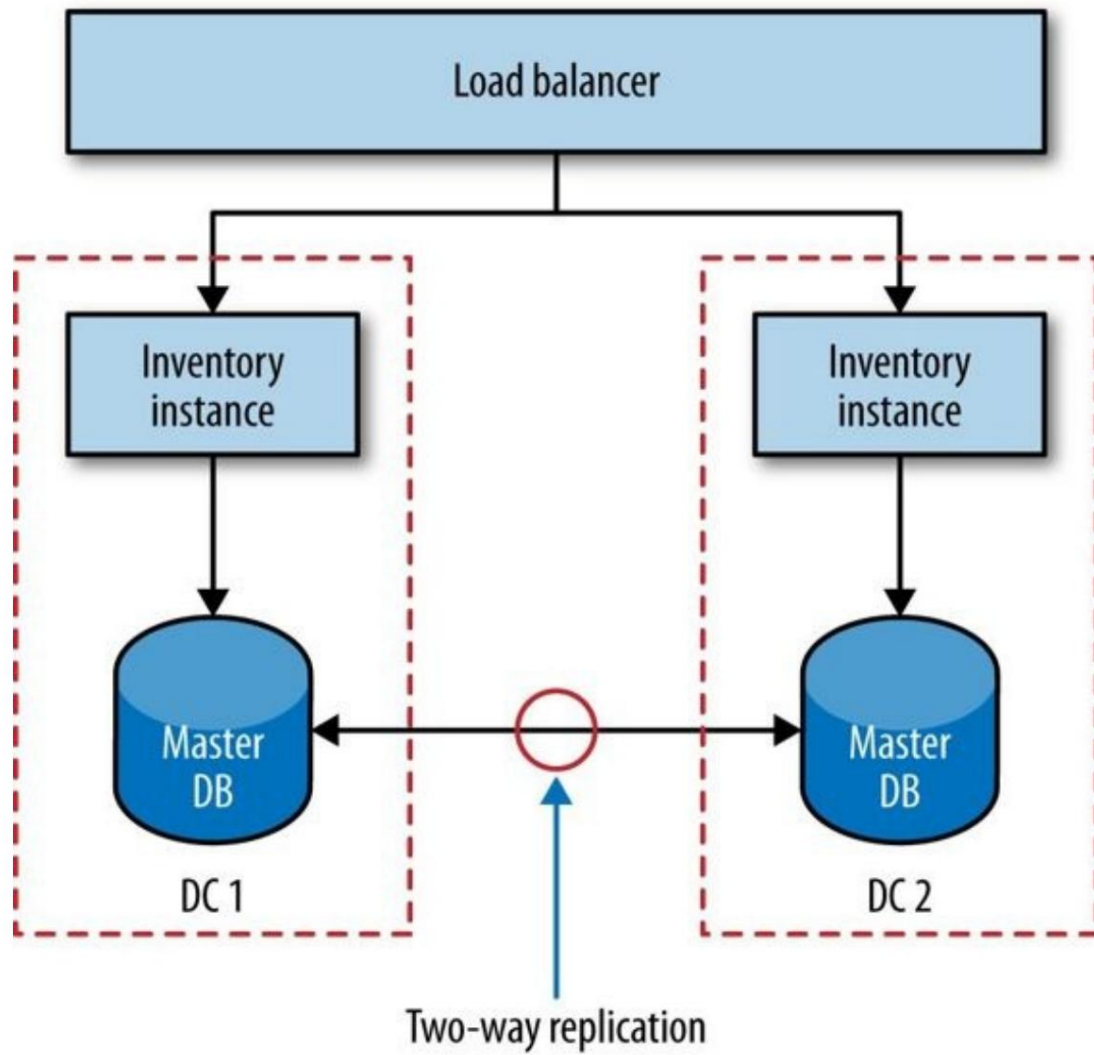


Figura 11-8. Uso de la replicación multiprimaria para compartir datos entre dos nodos de base de datos

Sacrificar la consistencia

Supongamos que no cerramos el servicio de inventario por completo. Si realizo un cambio ahora en los datos de DC1, la base de datos de DC2 no lo verá. Esto significa que cualquier solicitud realizada a nuestro nodo de inventario en DC2 muestra 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 ejecución a pesar de la *partición*, pero hemos perdido *consistencia*. Esto a menudo se llama un sistema *AP*. No podemos quedarnos con los tres.

Durante esta partición, si seguimos aceptando escrituras, entonces aceptamos el hecho de que en algún momento en el futuro deben resincronizarse. Cuanto más dure la partición, más difícil puede volverse 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 se complacen en ceder la consistencia para mantener la tolerancia y la disponibilidad de la partición 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 tenemos que vivir con la posibilidad de que los usuarios vean datos antiguos.

Sacrificar la disponibilidad

¿Qué sucede si necesitamos mantener la consistencia y queremos dejar algo más en su lugar?

Bueno, para mantener la consistencia, 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, 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 consistente y tolerante a particiones, o CP. En este modo, nuestro servicio tendría que averiguar cómo degradar la funcionalidad hasta que la partición se recupere y los nodos de la base de datos puedan volver a sincronizarse.

La consistencia entre múltiples nodos es realmente difícil. Hay pocas cosas (quizás nada) más difíciles en los sistemas distribuidos. Piensa un momento en ello. Imagine que quiero leer un registro del nodo de la base de datos local. ¿Cómo sé que está actualizado? Tengo que ir y preguntar 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 hace lecturas transaccionales, ¿verdad? Porque las lecturas transaccionales son lentas. Requieren candados. Una lectura puede bloquear todo un sistema. Todos los sistemas consistentes requieren cierto nivel de bloqueo para hacer su trabajo.

Como ya hemos discutido, los sistemas distribuidos tienen que esperar fallas. Considere nuestra lectura transaccional a través de un conjunto de nodos consistentes. Le pido a un nodo remoto que bloquee un registro determinado mientras se inicia la lectura. Terminó la lectura y le pido al nodo remoto que libere su bloqueo, pero ahora no puedo hablar con él. ¿Que pasa ahora? Los bloqueos son realmente difíciles de acertar 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](#)? La razón principal por la que son un desafío es debido a este problema para garantizar la coherencia en varios nodos.

Obtener la consistencia multinodo correcta es tan difícil que recomendaría *encarecidamente* que, si lo necesita, no intente inventarlo usted mismo. En su lugar, elija un almacén de datos o un servicio de bloqueo que ofrezca estas características. Consul, por ejemplo, del que hablaremos en breve, implementa un almacén de clave/valor muy coherente diseñado para compartir la configuración entre varios nodos. Junto con "Los amigos no dejen que los amigos escriban su propia criptografía" debería decir "Los amigos no dejen que los amigos escriban su propio almacén de datos consistente distribuido". Si cree que necesita escribir su propio almacén de datos de CP, primero lea todos los documentos sobre el tema, luego obtenga un doctorado y luego espere pasar unos años haciéndolo mal. Mientras tanto, usaré algo comercial que lo haga por mí, o más probablemente *me esforzaré mucho* para construir sistemas AP eventualmente consistentes.

¿Sacrificar la tolerancia a la partición?

Podemos elegir dos, ¿verdad? Así que tenemos nuestro sistema AP eventualmente consistente. Tenemos nuestro sistema CP consistente, pero difícil de construir y escalar. ¿Por qué no un sistema CA? Bueno, ¿cómo podemos sacrificar la tolerancia a la partición? Si nuestro sistema no tiene tolerancia a la partición, no puede ejecutarse en una red. En otras palabras, debe ser un proceso único que opere localmente. Los sistemas de CA no existen en los sistemas distribuidos.

AP o CP?

¿Cuál es correcto, AP o CP? Bueno, la realidad es *que depende*. Como personas que construyen el sistema, sabemos que existe una compensación. Sabemos que los sistemas AP se escalan más fácilmente y son más simples de construir, y sabemos que un sistema CP requerirá más trabajo debido a los desafíos para soportar la consistencia distribuida. Pero es posible que no entendamos el impacto comercial de esta compensación. Para nuestro sistema de inventario, si un registro está desactualizado por cinco minutos, ¿está bien? Si la respuesta es sí, un sistema AP podría ser la respuesta. Pero, ¿qué pasa con el saldo retenido para un cliente en un banco? ¿Eso puede estar desactualizado? Sin conocer el contexto en el que se utiliza la operación, no podemos saber qué hacer correctamente. Saber sobre el teorema CAP solo lo ayuda a comprender que existe esta compensación y qué preguntas hacer.

No es todo o nada

Nuestro sistema como un todo no necesita ser AP o CP. Nuestro catálogo podría ser AP, ya que no nos importa demasiado un disco 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 vencido, pero que cuando se trata de actualizar un saldo, necesitamos que sea consistente para garantizar que los clientes no usen más puntos de los que tienen disponibles. ¿Es este microservicio CP o AP, o ambos? En realidad, lo que hemos hecho es reducir las compensaciones en torno al teorema CAP a las capacidades de servicio individuales.

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

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

y el mundo real

Gran parte de lo que hemos hablado es el mundo electrónico: bits y bytes almacenados en la memoria.

Hablamos de consistencia 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 controlar eso, ¿verdad?

Repasemos nuestro sistema de inventario. Esto se asigna a elementos físicos del mundo real. Mantenemos un conteo en nuestro sistema de cuántos álbumes tenemos. Al comienzo del día teníamos 100 copias de *Give Blood* de The Brakes. Vendimos uno. Ahora tenemos 99 copias. Fácil, ¿verdad?

¿Por qué sucede si cuando se está enviando el pedido, alguien tira una copia del álbum al suelo y se pisa y se rompe? ¿Que pasa ahora? Nuestros sistemas dicen 99, pero hay 98 copias en el estante.

¿Qué pasaría si hiciéramos nuestro sistema de inventario AP en su lugar, y ocasionalmente tuviéramos que contactar a un usuario más adelante y decirle que uno de sus artículos está realmente agotado? ¿Sería eso lo peor del mundo? Sin duda, sería mucho más fácil de construir, escalar y garantizar que sea correcto.

Tenemos que reconocer que no importa cuán consistentes puedan ser nuestros sistemas en sí mismos, no pueden saber todo lo que sucede, especialmente cuando mantenemos registros del mundo real. Esta es una de las principales razones por las que los sistemas de puntos de acceso terminan siendo la decisión correcta en muchas situaciones. Aparte de la complejidad de construir sistemas CP, de todos modos no pueden solucionar todos nuestros problemas.

Descubrimiento de servicios

Una vez que tiene más de unos pocos microservicios, su atención inevitablemente se centra en saber dónde está todo. Tal vez desee saber qué se está ejecutando en un entorno determinado para saber qué debe monitorear. Tal vez sea tan sencillo como saber dónde está tu servicio de cuentas para que aquellos microservicios que lo utilicen sepan dónde encontrarlo. O tal vez solo desee facilitar que los desarrolladores de su organización sepan 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 manejarlo.

Todas las soluciones que veremos manejan las cosas en dos partes. Primero, 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 registrado. Sin embargo, el descubrimiento de servicios se vuelve más complicado cuando consideramos un entorno en el que constantemente estamos destruyendo e implementando nuevas instancias de servicios. Idealmente, querríamos cualquier solución que elijamos para hacer frente a esto.

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

DNS

Es bueno empezar simple. DNS nos permite asociar un nombre con la dirección IP de una o varias máquinas. Podríamos decidir, por ejemplo, que nuestro servicio de cuentas se encuentre siempre en *accounts.musiccorp.com*. Luego tendríamos ese punto de entrada a la dirección IP del host que ejecuta ese servicio, o tal vez lo resolveríamos en un balanceador de carga que está distribuyendo la carga a través de varias instancias. Esto significa que tendríamos que manejar la actualización de estas entradas como parte de la implementación de nuestro servicio.

Cuando se trata de 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 *<nombre del servicio>-<entorno>.musiccorp.com*, que nos proporcione entradas como *cuentas-uat.musiccorp.com* o *cuentas-dev.musiccorp.com*.

Una forma más avanzada de manejar diferentes entornos es tener diferentes servidores de nombres de dominio para diferentes entornos. Así que podría suponer que *accounts.musiccorp.com* es donde siempre encuentro el servicio de cuentas, pero podría resolverse en diferentes hosts dependiendo de dónde realice la búsqueda. Si ya tiene sus entornos ubicados en diferentes segmentos de red y se siente cómodo administrando sus propios servidores y entradas DNS, esta podría ser una solución bastante buena, pero es mucho trabajo si no obtiene otros beneficios de esta configuración.

DNS tiene una gran cantidad de ventajas, la principal es que es un estándar tan bien entendido y bien utilizado que casi cualquier pila de tecnología lo admitirá. 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 tratamos con hosts altamente descartables, lo que hace que la actualización de las entradas de DNS sea algo dolorosa. El servicio Route53 de Amazon hace un buen trabajo con esto, pero no he visto una opción autohospedada que sea tan buena todavía, aunque (como veremos en breve) Consul puede ayudarnos aquí. Además de los problemas para actualizar las entradas de DNS, la especificación de 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 la entrada fresca. Cuando queremos cambiar el host al que se refiere el nombre de dominio, actualizamos esa entrada, pero debemos suponer que los clientes conservarán la IP anterior durante *al menos* el tiempo que indica el TTL. Las entradas de DNS se pueden almacenar en caché en varios lugares (incluso la JVM almacenará en caché las entradas de DNS a menos que usted le indique que no lo haga), y cuantos más lugares se almacenen en caché, más obsoleta puede ser la entrada.

Una forma de solucionar este problema es hacer que la entrada del nombre de dominio para 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 anterior de la entrada del balanceador de carga y agregar la nueva. Algunas personas usan la rotación por turnos 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

deje de enrutar el tráfico a uno de los hosts en caso de que se enferme.

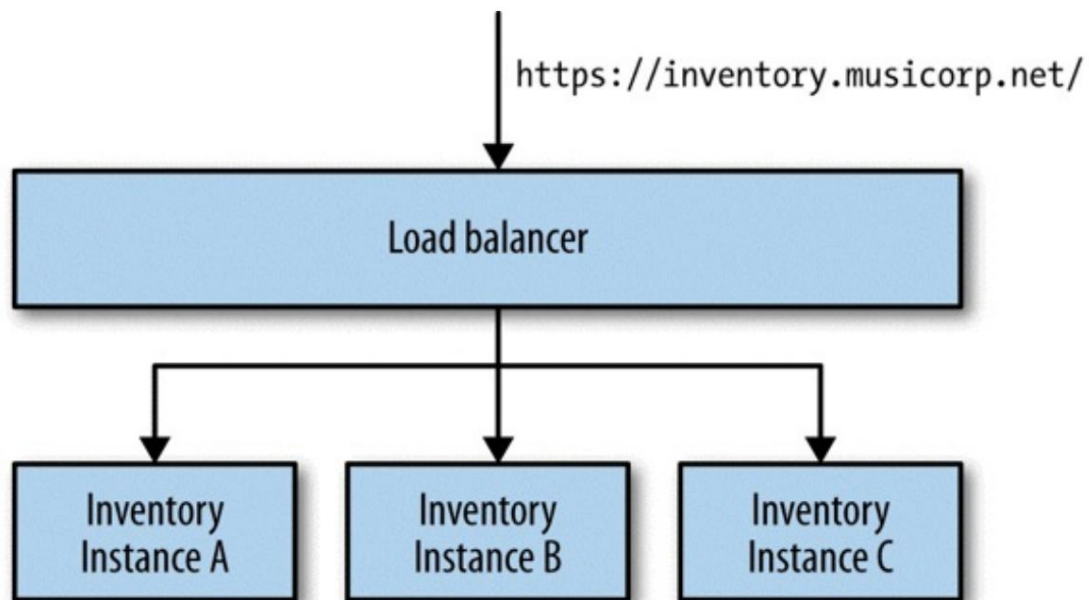


Figura 11-9. Uso de DNS para resolver en un balanceador de carga para evitar entradas de DNS obsoletas

Como se mencionó, DNS se entiende bien y es 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 únicos, es probable que el DNS se refiera directamente a los hosts. Pero para aquellas situaciones en las que necesita más de una instancia de un host, haga que las entradas de DNS resuelven los balanceadores de carga que pueden manejar la puesta en servicio y fuera de servicio de hosts individuales, según corresponda.

Registros de servicios dinámicos

Las desventajas del DNS como una 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, lo que a su vez ofrece la posibilidad de buscar estos servicios más adelante. . A menudo, estos sistemas hacen más que solo proporcionar registro y descubrimiento de servicios, lo que puede o no ser algo bueno. Este es un campo lleno de gente, por lo que solo veremos algunas opciones para darle una idea de lo que está disponible.

cuidador del zoológico

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

Al igual que muchos tipos de sistemas similares, Zookeeper se basa en la ejecución de varios 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 en Zookeeper se trata de garantizar que los datos se repliquen de manera segura entre estos nodos y que las cosas permanezcan consistentes 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 una notificación cuando cambien. Zookeeper se usa a menudo como un almacén de configuración general, por lo que también puede almacenar la configuración específica del servicio en él, lo que le permite realizar tareas como cambiar dinámicamente los niveles de registro o desactivar funciones 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 en sí mismo es bastante genérico en lo que ofrece, razón por la cual se usa para tantos casos de uso. Puede considerarlo simplemente como un árbol de información replicado del que puede recibir alertas cuando cambia. Esto significa que, por lo general, construirá cosas encima para adaptarse a su caso de uso particular. Afortunadamente, existen bibliotecas cliente para la mayoría de los idiomas.

En el gran esquema de las cosas, Zookeeper podría considerarse *antiguo* a estas alturas y no nos brinda tanta funcionalidad lista para usar para ayudar con el descubrimiento de servicios en comparación con algunas de las alternativas más nuevas. Dicho esto, ciertamente está probado y probado, y es ampliamente utilizado. Los algoritmos subyacentes que implementa Zookeeper son bastante difíciles de acertar. Conozco a un proveedor de base de datos, por ejemplo, que usaba Zookeeper solo para la elección de líder a fin de garantizar que un nodo principal se promocionara 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 hizo Zookeeper. La gente suele decir que no debería escribir sus propias bibliotecas de criptografía. Extendería eso diciendo que tampoco deberías escribir tus propios sistemas de coordinación distribuidos. Hay mucho que decir sobre el uso de cosas existentes que simplemente funcionan.

Cónsul

Al igual que Zookeeper, **Consul** admite tanto la gestión de la configuración como el descubrimiento de servicios. Pero va más allá de Zookeeper al brindar más soporte para estos casos de uso clave. Por ejemplo, expone una interfaz HTTP para el descubrimiento de servicios, y una de las características principales de Consul es que en realidad proporciona un servidor DNS listo para usar; específicamente, puede servir registros SRV, que le brindan tanto una IP como un puerto para un nombre dado. Esto significa que si parte de su sistema ya usa DNS y puede admitir registros SRV, puede simplemente ingresar a Consul y comenzar a usarlo sin ningún cambio en su sistema existente.

Consul también incorpora otras capacidades que le pueden resultar útiles, como la capacidad de realizar comprobaciones de estado en los nodos. Esto significa que Consul bien podría superponerse a las capacidades proporcionadas por otras herramientas de monitoreo dedicadas, aunque es más probable que use Consul como fuente de esta información y luego la incorpore a un tablero o sistema de alerta más completo. Sin embargo, el diseño altamente tolerante a fallas de Consul y su enfoque en el manejo de sistemas que hacen un uso intensivo de nodos efímeros me hace preguntarme si puede terminar reemplazando sistemas como Nagios y Sensu para algunos casos de uso.

Consul utiliza una interfaz RESTful HTTP para todo, desde registrar un servicio, consultar el almacén de clave/valor o insertar comprobaciones de estado. Esto hace que la integración con diferentes pilas de tecnología sea muy sencilla. Una de las otras cosas que realmente me gustan de Consul es que el equipo detrás de él ha dividido la pieza de administración de clústeres subyacente. Serf, sobre el que se sienta Consul, se encarga de la detección de nodos en un clúster, la gestión de fallas y las alertas. Luego, Consul agrega el descubrimiento de servicios y la gestión de la configuración. ¡Esta separación de preocupaciones me atrae, lo que no debería sorprenderte dados los temas que se presentan en este libro!

Consul es muy nuevo, y dada la complejidad de los algoritmos que utiliza, esto normalmente me haría dudar en recomendarlo para un trabajo tan importante. Dicho esto, Hashicorp, el equipo detrás de él, sin duda tiene un gran historial en la creación de tecnología de código abierto muy útil (en la forma de Packer y Vagrant), el proyecto se está desarrollando activamente y he hablado con algunas personas que felizmente lo utilizan en la producción. Dado eso, creo que vale la pena echarle un vistazo.

Eureka

El sistema Eureka de código abierto de Netflix contrasta con 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, es muy específico en su caso de uso.

Eureka también proporciona capacidades básicas de equilibrio de carga en el sentido de que puede admitir la búsqueda básica por turnos de instancias de servicio. Proporciona un punto final basado en REST para que pueda escribir sus propios clientes o puede usar su propio cliente Java. El cliente de Java proporciona funciones adicionales, como la comprobación del estado de las instancias. Obviamente, si omite el propio cliente de Eureka y va directamente al punto final REST, estará solo allí.

Al hacer que los clientes se ocupen directamente del descubrimiento de servicios, evitamos la necesidad de un proceso separado. Sin embargo, necesita que cada cliente implemente el descubrimiento de servicios. Netflix, que se estandariza en JVM, logra esto haciendo que todos los clientes usen Eureka. Si estás en un ambiente más políglota, esto puede ser un desafío mayor.

Rodando el tuyo

Un enfoque que he usado yo mismo y que he visto en otros lugares es implementar su propio sistema. En un proyecto estábamos haciendo un uso intensivo de AWS, que ofrece la posibilidad de agregar etiquetas a las instancias. Al lanzar instancias de servicio, aplicaría etiquetas para ayudar a definir qué era la instancia y para qué se usaba. Esto permitió asociar algunos metadatos enriquecidos con un host determinado, por ejemplo:

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

Luego podría usar las API de AWS para consultar todas las instancias asociadas con una cuenta de AWS determinada para encontrar las máquinas que me interesan. Aquí, el propio AWS se encarga del almacenamiento de los metadatos asociados con cada instancia y nos brinda la capacidad de consultarlos. Luego construí herramientas de línea de comandos para interactuar con estas instancias, y crear tableros para monitorear el estado se vuelve bastante fácil, especialmente si adopta 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 usaran las API de AWS para encontrar las dependencias de sus servicios, pero no hay ninguna razón por la que no pueda hacerlo. Obviamente, si desea que los servicios ascendentes reciban una alerta cuando cambia la ubicación de un servicio descendente, está solo.

¡No te olvides de los humanos!

Los sistemas que hemos visto 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. Independientemente del sistema que elija, asegúrese de tener herramientas disponibles que le permitan crear informes y paneles sobre estos registros para crear pantallas para humanos, no solo para computadoras.

Servicios de documentación

Al descomponer nuestros sistemas en microservicios de grano más fino, esperamos exponer muchas costuras en forma de API que las personas pueden usar para hacer muchas cosas, con suerte maravillosas. Si aciertas en tu descubrimiento, sabemos dónde están las cosas. Pero, ¿cómo sabemos qué hacen esas cosas o cómo usarlas? Obviamente, una opción es tener documentación sobre las API. Por supuesto, la documentación a menudo puede estar desactualizada. Idealmente, nos aseguraríamos de que nuestra documentación esté siempre actualizada con la API de microservicios, y facilitaríamos la visualización de esta documentación cuando sepamos dónde está un punto final de servicio. Dos piezas diferentes de tecnología, Swagger y HAL, intentan hacer esto realidad, y vale la pena mirar ambas.

Pavonearse

Swagger le permite describir su API para generar una interfaz de usuario web muy agradable que le permita ver la documentación e interactuar con la API a través de un navegador web. La capacidad de ejecutar solicitudes es muy buena: puede definir plantillas POST, por ejemplo, dejando 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 Swagger. Swagger tiene varias bibliotecas para diferentes idiomas que hacen esto por usted. Por ejemplo, para Java, puede anotar métodos que coincidan con sus llamadas API y el archivo se generará automáticamente.

Me gusta la experiencia del usuario final que le brinda Swagger, pero hace poco por el concepto de exploración incremental en el corazón de los hipermedios. Aún así, es una forma bastante agradable de exponer la documentación sobre sus servicios.

HAL y el navegador HAL

Por sí mismo, el **lenguaje de aplicación de hipertexto (HAL)** es un estándar que describe los estándares para los controles hipermedia que exponemos. Como cubrimos en el **Capítulo 4**, los controles hipermedia son los medios por los cuales permitimos que los clientes exploren 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 hipermedia de HAL, entonces no solo puede hacer uso de una gran cantidad de bibliotecas de clientes para consumir la API (en el momento de escribir este artículo, la wiki de HAL enumeraba 50 bibliotecas compatibles para varios idiomas diferentes), pero también puede utilizar el 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 usar no solo para actuar como documentación viva, sino también para ejecutar llamadas contra el propio servicio. Sin embargo, ejecutar llamadas no es tan hábil. Mientras que con Swagger puede definir plantillas para hacer cosas como emitir una solicitud POST, con HAL está más solo. La otra cara de esto es que el poder inherente de los controles hipermedia le permite explorar de manera mucho más efectiva la API expuesta por el servicio, ya que puede seguir los 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 impulsar esta documentación y sandbox está integrada en los controles hipermedia. Esta es una espada de doble filo. Si ya está utilizando controles de hipermedia, se necesita poco esfuerzo para exponer un navegador HAL y hacer que los clientes exploren su API. Sin embargo, si no usa hipermedia, no puede usar HAL o tiene que actualizar su API para usar hipermedia, lo que probablemente sea un ejercicio que rompa a los consumidores existentes.

El hecho de que HAL también describa un estándar de hipermedia con algunas bibliotecas de clientes compatibles es una ventaja adicional, y sospecho que es una de las principales razones por las que he visto una mayor aceptación de HAL como una forma de documentar las API que Swagger para aquellas personas que ya usan controles de hipermedia. . Si está utilizando hipermedia, mi recomendación es optar por HAL en lugar de Swagger. Pero si está utilizando hipermedia y no puede justificar el cambio, definitivamente le sugiero que pruebe Swagger.

El sistema de autodescripción

Durante la evolución temprana 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 pesados, lo que condujo a técnicas alternativas para tratar de dar sentido a nuestros sistemas. Martin Fowler discutió el concepto del **registro humano**, donde un enfoque mucho más ligero es simplemente tener un lugar donde los humanos puedan registrar información sobre los servicios en la organización en algo tan básico como un wiki.

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

Toda esta información está disponible programáticamente. Todos estos datos nos permiten hacer que nuestro registro humano sea más poderoso que una simple página wiki que sin duda quedará obsoleta.

En cambio, deberíamos usarlo para aprovechar y mostrar toda la información que nuestro sistema emitirá.

Al crear tableros personalizados, podemos reunir la amplia gama de información disponible para ayudarnos a dar sentido a nuestro ecosistema.

Por todos los medios, comience con algo tan simple como una página web estática o wiki que tal vez extraiga un poco de información del sistema en vivo. Pero busque obtener más y más información con el tiempo. Hacer que esta información esté fácilmente disponible es una herramienta clave para administrar la complejidad emergente que surgirá al ejecutar estos sistemas a escala.

Resumen

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

Además de lo que he cubierto aquí, recomiendo el excelente libro *Release It!* de Michael Nygard. En él, comparte una colección de historias sobre fallas del sistema y algunos patrones para ayudarlo a manejarlo bien. Vale la pena leer el libro (de hecho, iría tan lejos como para decir que debería considerarse una lectura esencial para cualquier persona que construya sistemas a escala).

Hemos recorrido bastante terreno y nos estamos acercando al final. En nuestro próximo y último capítulo, buscaremos volver a unir todo y resumir lo que hemos aprendido en el libro en general.

Capítulo 12. Uniéndolo todo

Hemos cubierto bastante en los capítulos anteriores, desde qué son los microservicios hasta cómo definir sus límites, y desde la tecnología de integración hasta las preocupaciones sobre la seguridad y el monitoreo. E incluso encontramos tiempo para averiguar cómo encaja el papel del arquitecto. Hay mucho que asimilar, ya que aunque los microservicios en sí mismos pueden ser pequeños, la amplitud y el impacto de su arquitectura no lo son. Así que aquí trataré de resumir algunos de los puntos clave tratados a lo largo del libro.

Principios de Microservicios

Discutimos el papel que pueden desempeñar los principios en el [Capítulo 2](#). 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 diversas decisiones que tenemos que tomar al construir nuestros sistemas. Debe definir absolutamente sus propios principios, pero pensé que valía la pena explicar en detalle lo que veo como 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 destilarlo hasta su esencia central.

Puede optar por adoptar estos principios al por mayor, o tal vez modificarlos para que tengan sentido en su propia organización. Pero tenga en cuenta el valor que resulta de usarlos en combinación: el todo debe ser mayor que la suma de las partes. Entonces, si decide dejar uno de ellos, asegúrese de comprender lo que se perderá.

Para cada uno de estos principios, he tratado de sacar algunas de las prácticas de apoyo que hemos cubierto en el libro. Como dice el refrán, hay más de una forma de despellejar a un gato: es posible que encuentre sus propias prácticas para ayudar a cumplir con estos principios, pero esto debería ayudarlo a comenzar.

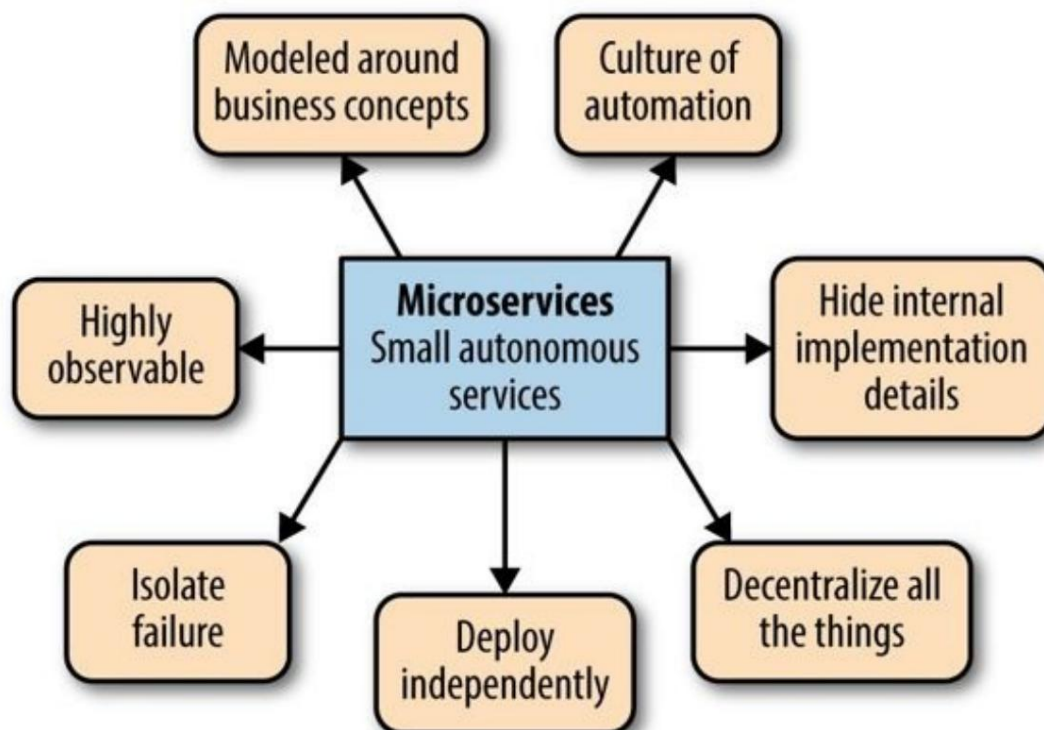


Figura 12-1. Principios de los microservicios

Modelo en torno a conceptos de negocio

La experiencia nos ha demostrado que las interfaces estructuradas en torno a contextos comerciales 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 comerciales fácilmente. Utilice *contextos acotados* para definir posibles límites de dominio.

Adopte 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 el esfuerzo inicial para 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 los sistemas monolíticos. Tener una llamada de línea de comando 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 comentarios rápidos sobre la calidad de producción de cada registro.

Considere el uso de *definiciones de entorno* para ayudarlo a especificar las diferencias de un entorno a otro, sin sacrificar la capacidad de usar un método de implementación uniforme. Piense en crear *imágenes personalizadas* para acelerar la implementación y adopte la creación de *servidores inmutables* completamente automatizados para que sea más fácil razonar sobre sus sistemas.

Ocultar detalles de implementación interna

Para maximizar la capacidad de un servicio para evolucionar independientemente de cualquier otro, es vital que ocultemos los detalles de implementación. El modelado de *contextos limitados* puede ayudar, ya que esto nos ayuda a centrarnos en los modelos que deben compartirse y los que deben ocultarse. Los servicios también deben *ocultar sus bases de datos* para evitar caer en uno de los tipos más comunes de acoplamiento que pueden aparecer en las arquitecturas tradicionales orientadas a servicios, y usar *bombas de datos* o bombas de datos de *eventos* para consolidar datos en múltiples servicios con fines de generación de informes.

Siempre que sea posible, elija *API independientes de la tecnología* para darle 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), aún puede adoptar estas ideas.

Descentralizar todas las cosas

Para maximizar la autonomía que hacen posible los microservicios, debemos buscar constantemente la oportunidad de delegar la toma de decisiones y el control a los equipos que poseen los propios servicios. Este proceso comienza con la adopción *del autoservicio* siempre que sea posible, lo que permite a las personas implementar software a pedido, facilitando al máximo el desarrollo y las pruebas, 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 viaje, haciendo que los equipos sean responsables de los cambios que se realizan, idealmente incluso que decidan cuándo publicar esos cambios. Hacer uso del *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 implementar.

Alinee 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 los servicios centrados en el negocio que están creando. Cuando se necesite alguna orientación general, intente adoptar un modelo de *gobierno compartido* en el que las personas de cada equipo compartan colectivamente la responsabilidad de desarrollar la visión técnica del sistema.

Este principio también puede aplicarse a la arquitectura. Evite enfoques como el bus de servicios empresariales o los sistemas de orquestación, que pueden conducir a la centralización de la lógica empresarial y servicios tontos. En su lugar, *prefiera la coreografía sobre la orquestación* y el *middleware tonto, con terminales inteligentes* para garantizar que mantiene la lógica y los datos asociados dentro de los límites del servicio, lo que ayuda a mantener las cosas cohesionadas.

Desplegable de forma independiente

Siempre debemos esforzarnos por garantizar que nuestros microservicios puedan y se implementen por sí mismos. Incluso cuando se requieren cambios importantes, debemos buscar la *coexistencia de puntos finales versionados* 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 orquestar constantemente sus implementaciones. Cuando utilice la integración basada en RPC, *evite la generación de stubs de cliente/servidor estrechamente vinculados*, como la que promueve Java RMI.

Al adoptar un modelo *de un servicio por host*, reduce los efectos secundarios que podrían provocar que la implementación de un servicio afecte a otro servicio no relacionado. Considere el uso de técnicas de versión *blue/green* o *canary* para separar la implementación de la versión, lo que reduce el riesgo de que una versión salga mal. Utilice *contratos impulsados por el consumidor* para detectar cambios importantes antes de que sucedan.

Recuerde que debe 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 se actualizan* y usted debe adaptarse a esto.

Aislar falla

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

Cuando use llamadas de red, *no trate las llamadas remotas como llamadas locales*, ya que esto ocultará diferentes tipos de modo de falla. Así que asegúrese de que si está utilizando bibliotecas de clientes, la abstracción de la llamada remota no vaya demasiado lejos.

Si tenemos en mente los principios de la *antifragilidad* y esperamos que ocurran fallas en cualquier lugar y en todas partes, estamos en el camino correcto. Asegúrese de que sus *tiempos* de espera estén configurados adecuadamente. Comprenda cuándo y cómo usar *mamparos* e *interruptores automáticos* para limitar las consecuencias de un componente defectuoso. Comprenda cuál será el impacto de cara al cliente si solo una parte del sistema se comporta mal. Sepa 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 confiar en observar el comportamiento de una sola instancia de servicio o el estado de una sola máquina para ver si el sistema funciona correctamente. En cambio, necesitamos una visión conjunta de lo que está sucediendo. Use *el monitoreo semántico* para ver si su sistema se está comportando correctamente, inyectando *transacciones sintéticas* en su sistema para simular el comportamiento del usuario real. *Agregue sus registros y agregue sus estadísticas*, de modo que cuando vea un problema pueda profundizar hasta la fuente. Y cuando se trata de reproducir problemas desagradables o simplemente ver cómo interactúa su sistema en producción, use *ID de correlación* para permitirle rastrear llamadas a través del sistema.

¿Cuándo no debería usar microservicios?

A menudo recibo esta pregunta. Mi primer consejo sería que cuanto menos comprenda un dominio, más difícil le resultará encontrar contextos delimitados adecuados para sus servicios. Como discutimos anteriormente, establecer límites de servicio incorrectos puede resultar en tener que realizar muchos cambios en la colaboración de servicio a servicio, una operación costosa. Entonces, si está llegando a un sistema monolítico para el cual no comprende el dominio, dedique un tiempo a aprender qué hace el sistema primero y luego busque identificar los límites limpios del módulo antes de dividir los servicios.

El desarrollo greenfield también es bastante desafiante. No es solo que el dominio también sea probablemente nuevo; ¡Es que es mucho más fácil fragmentar algo que tienes que algo que no tienes! Entonces, nuevamente, considere comenzar primero monolítico y dividir las cosas cuando esté estable.

Muchos de los desafíos que enfrentará con los microservicios empeoran con la escala. Si principalmente hace las cosas manualmente, puede estar bien con 1 o 2 servicios, pero ¿5 o 10? Cumplir con las prácticas de monitoreo antiguas en las que solo observa estadísticas como la CPU y la memoria también podría funcionar bien para algunos servicios, pero cuanto más colabore de servicio a servicio, más doloroso será. Se encontrará con estos puntos débiles a medida que agregue más servicios, y espero que los consejos de este libro lo ayuden a ver venir algunos de estos problemas y le brinden algunos consejos concretos sobre cómo lidiar con ellos. Antes hablé de que REA y Gilt se tomaron un tiempo para construir las herramientas y las prácticas para administrar bien los microservicios, antes de poder usarlos en grandes cantidades. Estas historias solo me refuerzan la importancia de comenzar gradualmente para que comprenda el apetito y la capacidad de cambio de su organización, lo que lo ayudará a adoptar microservicios de manera adecuada.

Palabras de despedida

Las arquitecturas de microservicio 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 acertarás en todas estas decisiones, te lo puedo garantizar. Entonces, sabiendo que vamos a hacer algunas cosas mal, ¿cuáles son nuestras opciones? Bueno, sugeriría encontrar formas de hacer que cada decisión tenga un alcance pequeño; de esa manera, si te equivocas, solo impactarás en una pequeña parte de tu 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 a lo grande, sino en una serie de cambios realizados en su sistema a lo largo del tiempo para mantenerlo flexible.

Con suerte, ya he compartido con usted suficiente información y experiencias para ayudarlo a decidir si los microservicios son para usted. Si lo son, espero que pienses en esto como un viaje, no como un destino. Ir de forma incremental. Descomponga su sistema pieza por pieza, aprendiendo sobre la marcha. Y acostúmbrate: en muchos sentidos, la disciplina de cambiar y evolucionar continuamente nuestros sistemas es una lección mucho más importante que aprender que cualquier otra que haya compartido contigo a través de este libro. El cambio es inevitable. Abrázalo.

Índice

A

pruebas de aceptación, **tipos de pruebas**

acceso por referencia, **acceso por referencia**

responsabilidad, **gente**

adaptabilidad, **Resumen**

Proyecto Aegisthus, **bomba de datos de respaldo**

registros agregados, **registros, registros y aún más registros...**

sistemas antifrágiles, **Microservicios, La Organización Antifrágil-Aislamiento**

mamparas, **mamparas**

disyuntores, **Disyuntores**

ejemplos de, **La Organización Antifrágil**

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 se trata de las claves**

contenedores de aplicaciones, contenedores de **aplicaciones**

arquitectos (ver arquitectos de sistemas)

principios arquitectónicos

desarrollo de, **Principios**

Los 12 factores de Heroku, **Principios**

Principios clave de los microservicios, **Reuniéndolo 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**

específicos de la **plataforma, artefactos específicos de la plataforma**

colaboración asíncrona

complejidades de, **Complejidades de las arquitecturas asíncronas**

implementando, **Implementando la colaboración asincrónica basada en eventos**

vs. síncrono, **síncrono versus asíncrono**

Especificación ATOM, **opciones tecnológicas**

autenticación/autorización, **Autenticación y Autorización-El Diputado
Problema**

definición de términos, **autenticación y autorización**

Autorización de grano fino, 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 casos sobre, **dos estudios de casos sobre el poder de la automatización**

autonomía

microservicios y, **Autónomos**

rol del arquitecto de sistemas en, **Resumen**

escalado automático, **escalado automático**

disponibilidad

en teorema CAP, teorema **CAP**

principio clave de microservicios de **¿Cuánto es demasiado?**

sacrificar, **sacrificar la disponibilidad**

B

backends para frontends (BFF), **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 de la versión**

cuellos de botella, cuellos de botella de **entrega**

contextos acotados

concepto de, **el contexto acotado**

módulos y servicios, **Módulos y Servicios**

tortugas anidadas hasta el fondo

descomposición prematura, descomposición **prematura**

modelos compartidos frente a modelos ocultos, modelos **compartidos y ocultos**

diseño de sistemas y **contextos acotados y estructuras de equipo**

Guardafrenos, **seguridad para hornear en**

rompiendo cambios

evitar, **Evitar Romper Cambios**

diferir, **aplazarlo el mayor tiempo posible**

detección temprana de **cambios de ruptura de captura temprano**

Pruebas frágiles, Pruebas **escamosas y frágiles** .

fragilidad, **fragilidad**

construir canalizaciones, construir canalizaciones **y entrega continua**

mamparas, **mamparas**

lanzamiento de servicio incluido, **y las excepciones inevitables**

capacidades comerciales, capacidades **comerciales**

conceptos de negocio, **comunicación en términos de conceptos de negocio**

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

C

almacenamiento en caché

beneficios del almacenamiento en **caché**

fallas de caché, **ocultando el origen**

envenenamiento de caché, envenenamiento de **caché: un cuento 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**

Caché **proxy, 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**

canario soltando, canario **soltando**

Teorema **CAP**, Teorema **CAP-Y** el mundo real

Sistemas AP/CP, **no es todo o nada**

aplicación de, **y el mundo real**

conceptos básicos del **teorema CAP**

sacrificar la disponibilidad, **sacrificar la disponibilidad**

sacrificando la consistencia, **sacrificando la consistencia**

sacrificar la tolerancia a la partición, **¿sacrificar la tolerancia a la partición?**

fallas en cascada, **La Cascada**

gestión de certificados, **Certificados de Cliente**

Chaos Gorilla, **La Organización Antifrágil**

Chaos Monkey, **La Organización Antifrágil**

arquitectura coreografiada, **orquestración versus coreografía**

disyuntores, **plantilla de servicio personalizada**, disyuntores

mezcla de disyuntores para Ruby, **Bulkheads**

clase-responsabilidad-colaboración (CRC), **Costo del cambio**

certificados de cliente, **Certificados** de cliente

bibliotecas cliente, bibliotecas **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 prueba de Cohn, **alcance** de prueba

colaboración, **Resumen**

basado en eventos, **síncrono versus asíncrono**

solicitud/respuesta, **síncrono versus asíncrono**

Segregación de responsabilidad de consulta de comando (CQRS), **CQRS**

compromisos, dos fases, **transacciones distribuidas**

comunicación

Adaptación a **vías**, **Adaptación a vías de comunicación**

protocolos para (SOAP), **¿Qué pasa con la arquitectura orientada a servicios?**

síncrono frente a asíncrono, **síncrono frente a asíncrono**

transacciones de compensación, **cancelar toda la operación**

componibilidad, **componibilidad**

desviación de configuración, **servidores inmutables**

configuración, servicio, **Configuración del servicio**

problema adjunto confuso, **el problema adjunto**

consistencia

en teorema CAP, teorema **CAP**

sacrificar, **sacrificar la consistencia**

restricciones, **Restricciones**

cónsul, **cónsul**

contratos impulsados por el consumidor (CDC), **Consumer-Driven Tests to the Rescue-It's
Acerca de las conversaciones**

red de entrega de contenido (CDN), **almacenamiento en caché del lado del cliente, proxy y del lado del servidor**

sistemas de gestión de contenido (CMS), **Ejemplo: CMS como servicio**

entrega continua (CD), **Microservicios, Build Pipelines y Continuous Entrega**

integración continua (CI)

conceptos básicos, **una breve introducción a la integración continua**

lista de verificación para, **¿realmente lo estás haciendo?**

mapeo a microservicios, **Mapeo de Integración Continua a microservicios**

ley de conway

evidencia de, **evidencia**

al revés, **Ley de Conway al revés**

declaración de **la ley de Conway y el diseño del sistema**

resumen de, **resumen**

proceso de coordinación, **Transacciones Distribuidas**

equipo central, **papel de los custodios**

Core OS, ventana **acoplable**

ID de correlación, ID de **correlación**

¿ Sistema **CP, AP o CP?**

requisitos de funciones cruzadas (CFR), **pruebas de funciones cruzadas, ¿cuánto es demasiado?**

custodios, **Rol de los custodios**

imágenes personalizadas, imágenes **personalizadas**

Gestión de la relación con el cliente (CRM), **Ejemplo: El sistema CRM polivalente**

clientes, interactuando con

inscripción de nuevos clientes, **interacción con los clientes, orquestación**

Coreografía Versus

bases de datos compartidas, **la base de datos compartida**

D

datos

inserción por lotes de **recuperación de datos a través de llamadas de servicio**

durabilidad de **¿Cuánto es demasiado?**

cifrado de copia de seguridad, **Cifrar copias de seguridad**

recuperación a través de llamadas de servicio, **Recuperación de datos a través de llamadas de servicio**

protección en reposo, **protección de datos en reposo**

(ver también seguridad)

compartido, **ejemplo: datos compartidos**

estática compartida, **ejemplo: datos estáticos compartidos**

encriptación de datos, **vaya con el bien conocido**

bombas de datos

copia de seguridad, **bomba de datos de copia de seguridad**

recuperación de datos a través **de bombas de datos**

evento, **bomba de datos de eventos**

uso en serie de **Destinos alternativos**

descomposición de **la base de datos, las causas raíz de la comprensión de la base de datos**

ruptura de relaciones de clave externa, **Ejemplo: ruptura de clave externa**

Relaciones

enfoque incremental del **costo del cambio**

descripción general de, **resumen**

refactorización de bases de datos, **Staging the Break**

selección de puntos de separación, **Abordar el problema**

seleccionando el tiempo de separación, **Comprender 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 base de datos, **la base de datos compartida**

escalado de base de datos

Segregación de responsabilidad de consulta de comandos (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**

Pautas para la toma de decisiones, **Un enfoque basado en principios: un ejemplo del mundo real**

enfoque personalizado para la **combinación de 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)

identificar/empaquetar contextos, **Breaking Apart MusicCorp**

módulos, **Módulos**

concepto de costura, **se trata de costuras**

seleccionando puntos de separación, **Las razones para dividir el monolito**

seleccionando el tiempo de separación, **Comprender las causas fundamentales**

bibliotecas compartidas, bibliotecas **compartidas**

desacoplamiento, **Autónomo, 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 de la versión**

construir canalización, construir canalizaciones **y entrega continua**

lanzamiento de servicio incluido, **y las excepciones inevitables**

conceptos básicos de integración continua, **una breve introducción a la integración continua**
Integración

lista de verificación de integración continua, **¿realmente lo estás haciendo?**

integración continua en microservicios, **Mapping Continuous**
Integración a 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 frente a sistemas monolíticos, **Facilidad de implementación, Implementación**

descripción general de, **resumen**

separarse de la versión, separar la **implementación de la versión**

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, Reunir todo: Altamente observable**

(ver también principios arquitectónicos)

prácticas de diseño/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 todas partes

promesas clave de **componibilidad**

transacciones distribuidas, transacciones **distribuidas**

servicio DNS, **DNS**

estibador, **estibador**

documentación

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

importancia de, **servicios de documentación**

sistemas autodescriptivos, **HAL y HAL Browser**

arrogancia, **arrogancia**

diseño dirigido por dominio, **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**

maniqués, **burlándose o golpeando**

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

Y

empatía, **Resumen**

encriptación, **vaya con el bien conocido**

pruebas de extremo a extremo

usos apropiados para los viajes de prueba, no las historias, entonces, ¿debería usar pruebas de extremo a extremo?

Pirámide de prueba de Cohn, alcance de prueba

creación de, ¿Quién escribe estos ensayos?

Inconvenientes de las pruebas de escamas y frágiles .

ciclo de retroalimentación, El Gran Choque

implementación de esas pruebas complicadas de extremo a extremo

metaversión, la metaversión

alcance de las pruebas de extremo a extremo

tiempo de, ¿cuánto tiempo?

puntos finales

coexistiendo diferentes, coexisten diferentes puntos finales

agregación del lado del servidor, backends para frontends

ambientes

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, síncrono versus asíncrono

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

bots fallidos, **La Organización Antifrágil**

fracasos

cascada, **la cascada**

(ver también monitoreo)

lidiando con, **el fracaso está en todas partes**

falsificaciones, **burlas o stubbing**

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 administrador de paquetes FPM, **artefactos del sistema operativo**

funcionalidad, degradación, **Funcionalidad degradante**

GRAMO

días de juego, **La Organización Antifrágil**

guardianes, **papel de los custodios**

gobernancia

concepto de **Gobernanza y Liderazgo desde el Centro**

rol del arquitecto de sistemas en, **Resumen**

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

Grafito, **seguimiento métrico en múltiples servicios**

H

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, **Tecnología Heterogeneidad**

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**

Inconvenientes de HTTP sobre REST, **Desventajas de REST sobre HTTP**

Terminación HTTP, **Equilibrio de carga**

Autenticación básica **HTTP(S)**, autenticación básica **HTTP(S)**

registro humano, **HAL y el navegador HAL**

hypermedia, **Hypermedia como el motor del estado de la aplicación**

hipervisores, **virtualización tradicional**

Biblioteca Hystrix, **plantilla de servicio personalizada, mamparos**

yo

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 asíncrona basada en eventos, **Implementando asíncrono**
Colaboración basada en eventos

interfaz de **cliente**, **interfaz 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, **Integración**

orquestración versus coreografía, **orquestración versus coreografía**

extensiones reactivas, extensiones **reactivas**

Llamadas a procedimiento remoto, Llamadas a procedimiento **remoto**

REST (Transferencia de estado representacional), REST

servicios como máquinas de estado, **Servicios como máquinas de estado**

bases de datos compartidas, **la base de datos compartida**

comunicación síncrona frente a comunicación asíncrona, **síncrona frente a asíncrona**

software de terceros, **Integración con software de terceros**

interfaces de usuario, interfaces de **usuario**

versionado, **versionado**

interfaces

coexistiendo nuevos y viejos, **coexisten diferentes puntos finales**

implementación, **una interfaz** de implementación

establecimiento de estándares para, **Interfaces**

(ver también interfaces de usuario)

detalle de implementación interna, **Ocultar detalle de implementación interna**

modelo de código abierto interno, código abierto **interno**

sistemas de detección de intrusos (IDS), sistema de **detección (y prevención) de intrusos**

sistemas de prevención de intrusiones (IPS), sistema de **detección (y prevención) de intrusiones**

aislamiento, **aislamiento**

Arquitectos de TI (ver arquitectos de sistemas)

j

¿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, **registros, registros y aún más registros...**

L

latencia, **¿cuánto es demasiado?**

Latency Monkey, **la organización antifrágil**

arquitecturas en capas, **Microservicios**

bibliotecas

cliente, **bibliotecas** de cliente

métricas de servicio, métricas de **servicio**

compartido, **bibliotecas compartidas**

Contenedores Linux , **Contenedores Linux**

equilibrio de carga, **Equilibrio** de carga

deslastre de carga, **mamparos**

llamadas locales, **Las llamadas locales no son como las llamadas remotas**

registros

agregados, **registros, registros y aún más registros...**

(ver también monitoreo)

problemas de seguridad, **registro**

estandarización de, **estandarización**

logstash, **Registros, Registros y Más Registros...**

acoplamiento suelto, acoplamiento **suelto**, orquestación versus coreografía, suelto y **Organizaciones estrechamente acopladas**

METRO

ataques man-in-the-middle, **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?**

intermediarios 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**

Biblioteca de métricas, **plantilla de servicio personalizada**

microservicios

aplicación adecuada de **¿Cuándo no debería usar microservicios?**

autonomía y, **Autónomo, Construyendo un Equipo**

beneficios de los **microservicios**

composabilidad de, **componibilidad**

definición del término, **¿Qué son los microservicios?**

facilidad de implementación, **Facilidad de implementación**

inconvenientes de **No Silver Bullet**, **¿cuándo no debería usar microservicios?**

principios clave de **Unir todo**

alineación organizacional y alineación **organizacional**

orígenes de, **microservicios**

reemplazabilidad y, **Optimización para la reemplazabilidad**

resiliencia de, **Resiliencia**

escalado y, **escalado**

tamaño y, **pequeño, y enfocado en hacer una cosa bien**

heterogeneidad tecnológica de, **heterogeneidad tecnológica**

vs módulos, **Módulos**

vs arquitectura orientada a servicios, **What About Service-Oriented ¿Arquitectura?**

frente a bibliotecas compartidas, bibliotecas **compartidas**

microservicios a escala

sistemas antifrágiles, **La Organización Antifrágil**

medidas arquitectónicas de seguridad, **Medidas arquitectónicas de seguridad**

escalado automático, **escalado 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?**

lidiando con 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**

escalar, **escalar**

escalando bases de datos, escalando bases de **datos**

sistemas autodescriptivos, **HAL y HAL Browser**

descubrimiento de servicios, descubrimiento de **servicios**

middleware, **opciones tecnológicas**

burlarse contra stubbing, **Mocking o Stubbing**

servicios de modelado

contextos acotados, **El** contexto acotado

capacidades comerciales, capacidades **comerciales**

conceptos de negocio, **comunicación en términos de conceptos de negocio**

conceptos clave, **¿Qué hace un buen servicio?**

módulos y servicios, **Módulos y Servicios**

contextos delimitados anidados, **Turtles All the Way Down**

descomposición prematura, descomposición **prematura**

modelos compartidos frente a modelos ocultos, modelos **compartidos y ocultos**

límites técnicos, **el límite técnico**

descomposición modular, **Módulos**

módulos, **Módulos y Servicios**

vigilancia

fallas en cascada, **La Cascada**

registro central, **registros, registros y aún más registros...**

complejidades de, **Monitoreo**

ID de correlación, ID de **correlación**

mostrar/compartir resultados, **considerar la audiencia**

seguimiento de métricas en múltiples servicios, **Seguimiento de métricas** en **Servicios Múltiples**

múltiples servicios/múltiples servidores, **Múltiples Servicios, Múltiples Servidores**

descripción general de, **resumen**

informes en tiempo real, **El futuro**

semantic, **Implementando el Monitoreo Semántico**

métricas de servicio, métricas de **servicio**

servicio único/múltiples servidores, **Servicio único, Servidores múltiples**

servicio único/servidor **único, servicio único, servidor único**

estandarización de, **estandarización**

establecimiento de normas y, **Monitoreo**

sintético, **Supervisión sintética**

sistemas monolíticos

bases de código en, **pequeñas y enfocadas en hacer una cosa bien**

falta de cohesión/acoplamiento flojo en, **todo se trata de costuras**

bases de datos de informes en **The Reporting Database**

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

Ley de Moore, Ley de **Conway y Diseño de Sistemas**

Mountebank, **un servicio de talón más inteligente**

MTBF (tiempo medio entre fallas), **¿Tiempo medio de reparación sobre el tiempo medio entre fallas?**

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

norte

contextos delimitados anidados, **Turtles All the Way Down**

segregación de red, segregación de **red**

requisitos no funcionales, **pruebas de funciones cruzadas**

normalización de las **pruebas de desviación**, Flaky y Brittle

O

sistemas de aprovisionamiento bajo demanda, **escalado**

virtualización bajo demanda, **Microservicios**

arquitectura cebolla, **El límite técnico**

Proyecto de seguridad de aplicaciones web abiertas (OWASP), **Baking Security In**

OpenID Connect, **implementaciones comunes de inicio de sesión único**, uso de SAML o

Conexión de identificación abierta

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 organizativa

Ley de Conway y, **Ley de Conway y diseño de sistemas**

efecto en el diseño de sistemas, **Evidencia**

Organizaciones **sueltas vs. fuertemente acopladas**, **Organizaciones** sueltas y estrechamente acopladas

servicios huérfanos, **El Servicio de Huérfanos?**

OSGI (Iniciativa de puerta de enlace de código abierto), **Módulos**

propiedad

compartido, **controladores para servicios compartidos**

diseño del sistema y **propiedad del servicio**

PAGS

Packer, **imágenes personalizadas**

pacto, **pacto**

pacto, **pacto**

tolerancia de partición

en teorema CAP, teorema **CAP**

sacrificando, **¿Sacrificando la tolerancia a la partición?**

contraseñas, **vaya con el bien conocido**

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, **diferirlo el mayor tiempo posible**

escalado predictivo, escalado **automático**

parte principal, **Autenticación y Autorización**

problemas de privacidad, **Sea frugal**

prueba de propiedades, **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), **escalado para lecturas**

extensiones reactivas (Rx), **extensiones reactivas**

escalado reactivo, escalado **automático**

leer réplicas, **escalado para lecturas**

rediseño, **empezar de nuevo**

refactorización de bases de datos, **Staging the Break**

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

ventajas y desventajas de **¿Es terrible el RPC?**

fragilidad, **fragilidad**

definición del término, **llamadas a procedimiento remoto**

tecnologías disponibles, **llamadas a procedimientos remotos**

acoplamiento tecnológico, acoplamiento **tecnológico**

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

reemplazabilidad, **Optimización para la reemplazabilidad**

bases de datos de informes

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

directrices de bombas de **datos**, **Bombas de datos**

recuperación de datos a través de llamadas de servicio, **recuperación de datos a través de llamadas de servicio**

bombas de datos de **eventos**, **bomba de datos** de eventos

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

enfoque monolítico de **la base de datos de informes**

software de terceros, **recuperación de datos a través de llamadas de servicio**

colaboración de solicitud/respuesta, **síncrono frente a asíncrono**

ingeniería de resiliencia, **resiliencia**, **almacenamiento en caché para la resiliencia**

recursos, **DESCANSO**

tiempo de respuesta, **¿Cuánto es demasiado?**

REST (Transferencia de estado representacional), **REST-Downsides to REST Over HTTP**

Especificación ATOM, **opciones tecnológicas**

concepto de, **DESCANSO**

marcos para, **cuidado con demasiada comodidad**

Beneficios de HTTP sobre REST, **REST y HTTP**

Inconvenientes de HTTP sobre REST, **Desventajas de REST sobre HTTP**

formatos de texto, **JSON, XML o algo más?**

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

Riemann, **el futuro**

riesgo, difusión, Propagación de **su riesgo**

Principio de robustez, **aplazarlo el mayor tiempo posible**

Rx (extensiones reactivas), **Extensiones reactivas**

S

SAML, **implementaciones comunes de inicio de sesión único, usar SAML u OpenID Conectar**

SAN (redes de área de almacenamiento), **extendiendo su riesgo**

escalando, escalando **-comenzando de nuevo**

escalado automático, **escalado automático**

beneficios de, **Escalamiento**

bases de datos, **Bases** de datos escalables

equilibrio de carga, **Equilibrio** de carga

Razones para, **Escalamiento**

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

esparciendo el riesgo, esparciendo **su riesgo**

vertical, **ir más grande**

vs. rediseño, **Empezar de nuevo**

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

costuras, concepto de, **Todo se trata de costuras**

seguridad, Resumen de seguridad

ataques, defenderse de, **Defensa en profundidad**

autenticación/autorización, **autenticación y autorización**

copias de **seguridad**, **Bomba de datos de copia de seguridad**, **Cifrar copias de seguridad**

educación/concienciación sobre **seguridad para hornear en**

encriptación, **vaya con el bien conocido**

configuración de ejemplo, **un ejemplo resuelto**

verificación externa de, **Verificación externa**

cortafuegos, **cortafuegos**

elemento humano de, **El Elemento Humano**

importancia de, **Seguridad**

detección/prevención de **intrusiones**, **detección (y prevención)** de intrusiones
Sistema

almacenamiento **de llaves**, **todo se trata de las llaves**

registros, **registro**

ataques man-in-the-middle, **permitir todo dentro del perímetro**

sistemas operativos, **sistema operativo**

descripción general de, **Resumen**

contraseñas, **vaya con el bien conocido**

problemas de privacidad, **Sea frugal**

proteger los datos en reposo, **Proteger los datos en reposo**

autenticación/autorización de servicio a servicio, **servicio a servicio**
Autenticacion y autorizacion

selección de herramientas, **la regla de oro**

nubes privadas virtuales, **segregación de red**

Ciclo de vida del desarrollo de seguridad, **integración de la seguridad**

sistemas autodescriptivos, **HAL y HAL Browser**

monitoreo semántico, entonces, **¿debería usar pruebas de extremo a extremo?, monitoreo sintético**

control de versiones semántico, **utilizar el control de versiones semántico**

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

cuentas de servicio, **usar SAML u OpenID Connect**

límites de servicio, **zonificación**

(ver también servicios de modelaje)

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, **controladores para servicios compartidos**

proveedor de servicios, **implementaciones comunes de inicio de sesión único**

separación de servicios, puesta en escena, **Puesta en escena del descanso**

(ver también descomposición de la base de datos)

plantillas de servicio, plantilla de **servicio personalizada**

pruebas de servicio

Pirámide de prueba de Cohn, **alcance** de prueba

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

burlarse contra stubbing, **Mocking o Stubbing**

Servidor Mountebank para, **un servicio de código auxiliar más inteligente**

alcance de, **Pruebas de servicio**

arquitecturas orientadas a servicios (SOA)

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

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

reutilización de funcionalidad en **Componibilidad**

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

asignación de **servicio a host**, **plataforma de asignación de servicio a host 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 adjunto

claves API, claves **API**

certificados de cliente, **Certificados** de cliente

problema adjunto confuso, **el problema adjunto**

HMAC sobre HTTP, **HMAC sobre HTTP**

HTTP(S) básico, **autenticación básica HTTP(S)**

ataques man-in-the-middle, **permitir todo dentro del perímetro**

SAML/OpenID Connect, **Usar 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, **separando la implementación de la versión**

espías, **burlándose o golpeando**

Multiplexación SSH, **servicio único, varios servidores**

Certificados SSL, **autenticación básica HTTP(S)**

Terminación SSL, **Equilibrio de carga**

cumplimiento de normas

ejemplares, **ejemplares**

plantillas de servicio personalizadas, plantilla de **servicio personalizada**

establecimiento de normas, la **seguridad arquitectónica estándar requerida**

seguridad arquitectónica, seguridad **arquitectónica**

importancia de, **El estándar requerido**

interfaces, **interfaces**

monitoreo, **Monitoreo**

datos estáticos, **ejemplo: datos estáticos compartidos**

Patrón de aplicación de estrangulador, **El patrón de estrangulador, Seguridad arquitectónica Medidas**

metas estratégicas

ejemplo del mundo real, **un ejemplo del mundo real**

comprensión, **Metas Estratégicas**

stubbing vs burlarse, **Mocking o Stubbing**

Suro, **el futuro**

arrogancia, **arrogancia**

comunicación síncrona, **síncrona versus asíncrona**

monitoreo sintético, **Monitoreo Sintético**

diseño de sistemas

responsabilidad y, **Personas**

Adaptación a **las vías de comunicación, Adaptación a la comunicación caminos**

contextos acotados, contextos acotados **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 organizativa, **Ley de Conway al revés**

equipos destacados, equipos **destacados**

modelo de código abierto interno, código abierto **interno**

estructura organizacional y, **Evidencia**

servicios huérfanos, **El Servicio de Huérfanos?**

descripción general de, **resumen**

rol de los custodios, **Rol de los custodios**

madurez del servicio, **Madurez**

propiedad del **servicio**, propiedad **del servicio**

propiedad de servicios compartidos, **controladores para servicios compartidos**

utillaje, **utillaje**

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**

papel de, **Una visión evolutiva para el arquitecto**

límites de servicio y **Zonificación**

aplicación de estándares por parte de **Gobernanza a través de un servicio adaptado al código**
Modelo

establecimiento de estándares por, **el estándar requerido**

trabajo en equipo por, **Construyendo un equipo**

participación del equipo por, **Zonificación**

deuda técnica y, **Deuda Técnica**

T

tablas, compartidas, **Ejemplo: tablas compartidas**

plantillas de servicio personalizadas, plantilla de **servicio personalizada**

trabajo en equipo, formar **un equipo**

estructuras de equipo, **contextos acotados 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**

Dobles de prueba, **Mocking o Stubbing**

Pirámide de **prueba**, **ámbito de prueba**, pruebas impulsadas por el consumidor al rescate

prueba de cono de nieve, **¿cuántos?**

diseño basado en pruebas (TDD), **pruebas unitarias**

pruebas

canario soltando, canario **soltando**

consideraciones para, **compensaciones**

pruebas impulsadas por el **consumidor**, **Pruebas impulsadas por el consumidor al rescate**

Pruebas de funciones cruzadas, de **funciones** cruzadas

pruebas de extremo a extremo, **esas pruebas engañosas de extremo a extremo**

MTTR sobre MTBF, ¿**Tiempo medio de reparación sobre el tiempo medio entre fallas?**

descripción general de, **resumen**

pruebas de rendimiento, pruebas de **rendimiento**

postproducción, **Pruebas después de la producción**

alcance de, alcance de **prueba**

seleccionando el número de, ¿**Cuántos?**

monitoreo semántico, entonces, ¿**debería usar pruebas de extremo a extremo?**

separando la implementación de la versión, **separando la implementación de Liberar**

implementación de prueba 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 a través de llamadas de servicio

construir vs. comprar, **integración con software de terceros**

sistemas de gestión de contenido (CMS), **Ejemplo: CMS como servicio**

Gestión de la relación con el cliente (CRM), **Ejemplo: El polivalente sistema CRM**

personalización de, **personalización**

temas de integración, **espaguetis** de integración

falta de control sobre, **falta de control**

bases de datos de informes, **recuperación de datos a través de llamadas de servicio**

Patrón de aplicación de estrangulador, **el patrón** de estrangulador

Acoplamiento apretado, **Acoplamiento suelto, Organizaciones acopladas sueltas y estrechamente**

tiempo de vida (TTL), **DNS**

tiempos de espera, tiempos de **espera**

gestores de transacciones, **transacciones distribuidas**

límites transaccionales, **Límites transaccionales:** ¿entonces **qué hacer?**

actas

compensando, **abortar toda la operación**

Distribuidas, **Transacciones Distribuidas**

Seguridad de la capa de transporte (TLS), **certificados de cliente**

confirmaciones de dos fases, **transacciones distribuidas**

virtualización tipo 2, virtualización **tradicional**

EN

UDDI (Descripción Universal, Descubrimiento e Integración), **HAL y HAL Navegador**

pruebas unitarias

Pirámide de prueba de Cohn, **alcance** de prueba

objetivos de, **pruebas unitarias**

Cuadrante de Marick, **Tipos de Pruebas**

alcance de, **Pruebas unitarias**

interfaces de usuario, interfaces de **usuario: un enfoque híbrido**

Composición API, **Composición API**

Puertas de **enlace API**, **backends para frontends**

Granularidad de API, **hacia lo digital**

Pirámide de prueba de Cohn, **alcance** de prueba

(ver también pruebas de extremo a extremo)

restricciones, **Restricciones**

evolución de, **interfaces de usuario**

composición de fragmentos, composición de fragmentos de **interfaz** de usuario

enfoques híbridos, **un enfoque híbrido**

EN

vagabundo, **vagabundo**

control de versiones , **control de versiones: uso de varias versiones simultáneas del servicio**

Detectar cambios importantes con anticipación, **Capturar** cambios importantes con anticipación

coexistiendo diferentes puntos finales, coexistir diferentes puntos **finales**

aplazar los cambios importantes, **aplazarlo el mayor tiempo posible**

múltiples versiones concurrentes, **usar múltiples versiones de servicio concurrentes**

semántico, **usar versiones semánticas**

escalado vertical, **Go Bigger**

nubes privadas virtuales (VPC), **segregación de red**

virtualización

hipervisores, **virtualización tradicional**

tradicional, **virtualización tradicional**

tipo 2, **virtualización tradicional**

plataformas de virtualización

estibador, **estibador**

Contenedores Linux , **Contenedores Linux**

bajo demanda, **Microservicios**

redes de área de almacenamiento en, Propagación de **su riesgo**

vagabundo, **vagabundo**

visión, **resumen**

En

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

cachés de escritura posterior, **almacenamiento en caché para la resiliencia**

X

¿ XML, **JSON, XML o algo más?**

DE

Zed Attack Proxy (ZAP), **seguridad de horneado en**

Zipkin, **ID de correlación**

Guardián del zoológico, **Guardián** del zoológico

Sobre el Autor

Sam Newman es tecnólogo en ThoughtWorks, donde actualmente divide su tiempo entre ayudar a los clientes y trabajar como arquitecto de los propios 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 las operaciones de TI.

Si le preguntaras a qué se dedica, diría: “Trabajo con personas para crear mejores sistemas de software”. Ha escrito artículos, presentado en conferencias y esporádicamente se compromete con proyectos de código abierto.

Colofón

Los animales de 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 de cera. La apicultura para recolectar miel ha sido una actividad humana durante miles de años.

Las abejas melíferas viven en colmenas con miles de individuos y tienen una estructura social muy organizada. Hay tres castas: reina, zángano y obrera. Cada colmena tiene una reina, que permanece fértil durante 3 a 5 años después de su vuelo de apareamiento 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 muchos roles durante su vida, como niñera, trabajadora de la construcción, tendera, guardia, funeraria y recolectora. Las abejas obreras que se alimentan se comunican entre sí "bailando" en patrones particulares para compartir información sobre recursos cercanos.

Las tres castas de abejas melíferas son similares en apariencia, con alas, seis patas y un cuerpo segmentado en cabeza, tórax y abdomen. Tienen pelos cortos y borrosos en 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 de néctar de flores rico en azúcar.

Las abejas son cruciales 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 "trastorno de colapso de colonias". Todavía no está claro qué está causando esta mortandad: algunas teorías incluyen parásitos, uso de insecticidas o enfermedades, pero hasta la fecha no se han encontrado medidas preventivas efectivas.

Muchos de los animales de las portadas de O'Reilly están en peligro de extinción; todos ellos son importantes para el mundo. Para obtener más información sobre cómo puede ayudar, visite animals.oreilly.com.

La imagen de la portada es de Johnson's *Natural History*. 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

Una palabra sobre los microservicios hoy

Navegando por este libro

Las convenciones usadas en este libro

Safari® Libros en línea

Cómo contactarnos

Expresiones de gratitud

1. Microservicios

¿Qué son los microservicios?

Pequeño y enfocado 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

Sin bala de plata

Resumen

2. Las comparaciones inexactas del

arquitecto evolutivo

Una visión evolutiva para el arquitecto

Zonificación

Un enfoque basado en principios

Metas estratégicas

Principios

Prácticas

Combinación de principios y prácticas

Un ejemplo del mundo real

El estándar requerido

Vigilancia

Interfaces

Seguridad Arquitectónica

Gobernanza a través del código

ejemplares

Plantilla de servicio personalizada

Deuda técnica

Manejo de excepciones

Gobernanza y Liderazgo desde el Centro

Construyendo un equipo

Resumen

3. Cómo modelar servicios

Presentamos MusicCorp

¿Qué hace un buen servicio?

Bajo acoplamiento

Alta Cohesión

El contexto acotado

Modelos compartidos y ocultos

Módulos y Servicios

Descomposición prematura

Capacidades comerciales

Tortugas hasta el fondo

Comunicación en términos de conceptos comerciales

El límite técnico

Resumen

4. Integración

En busca de la tecnología de integración ideal

Evite romper los cambios

Mantenga sus API independientes de la tecnología

Simplifique su servicio para los consumidores

Ocultar detalle de implementación interna

Interfaz con los clientes

La base de datos compartida

Sincrónico versus asincrónico

Orquestación Versus Coreografía

Llamadas a procedimiento remoto

Acoplamiento tecnológico

Las llamadas locales no son como las llamadas remotas

fragilidad

¿Es terrible el RPC?

RESTO y HTTP

Hipermedia como motor del estado de la aplicación

¿JSON, XML o algo más?

Cuidado con demasiada comodidad

Desventajas de REST sobre HTTP

Implementación de la colaboración asincrónica basada en eventos

Opciones tecnológicas

Complejidades de las arquitecturas asíncronas

Servicios como máquinas de estado

Extensiones reactivas

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

Bibliotecas de clientes

Acceso por Referencia

Versionado

Aplazarlo el mayor tiempo posible

Atrapa los cambios importantes antes de tiempo

Usar versiones semánticas

Coexisten diferentes puntos finales

Usar múltiples versiones de servicio simultáneas

Interfaces de usuario

Hacia lo digital

Restricciones

Composición API

Composición de fragmentos de interfaz de usuario

Backends para Frontends

Un enfoque híbrido

Integración con software de terceros

Falta de control

personalización

Espagueti de integración

En sus propios términos

El patrón del estrangulador

Resumen

5. Dividir el monolito

Se trata de costuras

Breaking Apart MusicCorp

Las razones para dividir el monolito

Ritmo de cambio

Estructura de equipo

Seguridad

Tecnología

Dependencias enredadas

La base de datos

Abordar el problema

Ejemplo: ruptura de relaciones de clave externa

Ejemplo: datos estáticos compartidos

Ejemplo: datos compartidos

Ejemplo: Mesas Compartidas

Bases de datos de refactorización

Escenificando el descanso

Límites transaccionales

Inténtalo de nuevo más tarde

Abortar toda la operación	1
Transacciones distribuidas	2
¿Entonces lo que hay que hacer?	3
Informes	4
La base de datos de informes	5
Recuperación de datos a través de llamadas de servicio	6
Bombas de datos	7
Destinos alternativos	8
Bomba de datos de eventos	9
Bomba de datos de respaldo	10
Hacia el tiempo real	11
Costo del cambio	12
Comprender las causas fundamentales	13
Resumen	14

6. Despliegue

Una breve introducción a la integración continua ¿Realmente lo estás haciendo?	15
Asignación de integración continua a microservicios	16
Construir canalizaciones y entrega continua	17
Y las excepciones inevitables	18
Artefactos específicos de la plataforma	19
Artefactos del sistema operativo	20
Imágenes personalizadas	21
Imágenes como artefactos	22
Servidores inmutables	23
Entornos	24

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 Linux

Estibador

Una interfaz de implementación

Definición de entorno

Resumen

7. Pruebas

Tipos de pruebas

Alcance de prueba

Pruebas unitarias

Pruebas de servicio

Pruebas de extremo a extremo

Compensaciones

¿Cuanto?

Implementación de pruebas de servicio

Burlarse o golpearse

Un servicio de talón más inteligente

Esas pruebas engañosas de extremo a extremo

Desventajas de las pruebas de extremo a extremo

Pruebas escamosas y frágiles

¿Quién escribe estas pruebas?

¿Cuánto tiempo?

El gran amontonamiento

La metaversión

Viajes de prueba, no historias

Pruebas impulsadas por el consumidor al rescate

Pacto

Se trata de conversaciones

Entonces, ¿debería usar pruebas de extremo a extremo?

Pruebas después de la producción

Separación de la implementación de la versión

Liberación de Canarias

¿Tiempo medio de reparación sobre el tiempo medio entre fallas?

Pruebas de funciones cruzadas

Pruebas de rendimiento

Resumen

8. Monitoreo

Servicio único, servidor único

Servicio único, servidores múltiples

Múltiples Servicios, Múltiples Servidores

Registros, registros y aún más registros...

Seguimiento de métricas en múltiples servicios

Métricas de servicio

Monitoreo Sintético

Implementando el Monitoreo Semántico

ID de correlación

la cascada

Estandarización

Considere la audiencia

El futuro

Resumen

9. Seguridad

Autenticacion y autorizacion

Implementaciones comunes de inicio de sesión único

Puerta de enlace de inicio de sesión único

Autorización detallada

Autenticación y autorización de servicio a servicio

Permitir todo dentro del perímetro

Autenticación básica HTTP(S)

Usar SAML u OpenID Connect

Certificados de clientes

HMAC sobre HTTP

Claves API

El problema del diputado

Protección de datos en reposo

Ir con el bien conocido

Todo se trata de las llaves

Elige tus objetivos

Descifrar bajo demanda

	Cifrar copias de seguridad
	Defensa en profundidad
	cortafuegos
	Inicio sesión
	Sistema de detección (y prevención) de intrusos
	Segregación de red
	Sistema operativo
	Un ejemplo resuelto
	ser frugal
	El elemento humano
	La regla de oro
	Seguridad para hornear en
	Verificación externa
	Resumen
10.	Ley de Conway y diseño de sistemas
	Evidencia
	Organizaciones sueltas y fuertemente acopladas
	Windows Vista
	netflix y amazon
	¿Qué podemos hacer con esto?
	Adaptarse a las vías de comunicación
	Propiedad del servicio
	Controladores para servicios compartidos
	Demasiado difícil de dividir
	Equipos destacados
	Cuellos de botella en la entrega

Código abierto interno

Rol de los Custodios

Madurez

Estampación

Contextos acotados y estructuras de equipo

¿El servicio de huérfanos?

Estudio de caso: RealEstate.com.au

Ley de Conway al revés

Gente

Resumen

11. Microservicios a escala

El fracaso está en todas partes

¿Cuánto es demasiado?

Funcionalidad degradante

Medidas de seguridad arquitectónica

La Organización Antifrágil

Tiempos de espera

Rompedores de circuito

Mamparos

Aislamiento

idempotencia

Escalada

ir más grande

División de cargas de trabajo

Difundir su riesgo

Balanceo de carga

Sistemas basados en trabajadores

Empezando de nuevo

Escalando 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 del lado del servidor

Almacenamiento en caché en HTTP

Almacenamiento en caché para escrituras

Almacenamiento en caché para la resiliencia

Ocultar el origen

Mantenlo simple

Envenenamiento de caché: una advertencia

Autoescalado

Teorema de la PAC

Sacrificar la consistencia

Sacrificar la disponibilidad

¿Sacrificar la tolerancia a la partición?

AP o CP?

No es todo o nada

y el mundo real

Descubrimiento de servicios

DNS

Registros de servicios dinámicos

[cuidador del zoológico](#)

[Cónsul](#)

[Eureka](#)

[Rodando el tuyo](#)

[¡No te olvides de los humanos!](#)

Servicios de documentación

[Pavonearse](#)

[HAL y el navegador HAL](#)

El sistema de autodescripción

Resumen

12. Reuniéndolo todo

Principios de Microservicios

[Modelo en torno a conceptos de negocio](#)

[Adopte una cultura de automatización](#)

[Ocultar detalles de implementación interna](#)

[Descentralizar todas las cosas](#)

[Desplegable de forma independiente](#)

[Aislar falla](#)

[Altamente observable](#)

[¿Cuándo no debería usar microservicios?](#)

[Palabras de despedida](#)

Índice