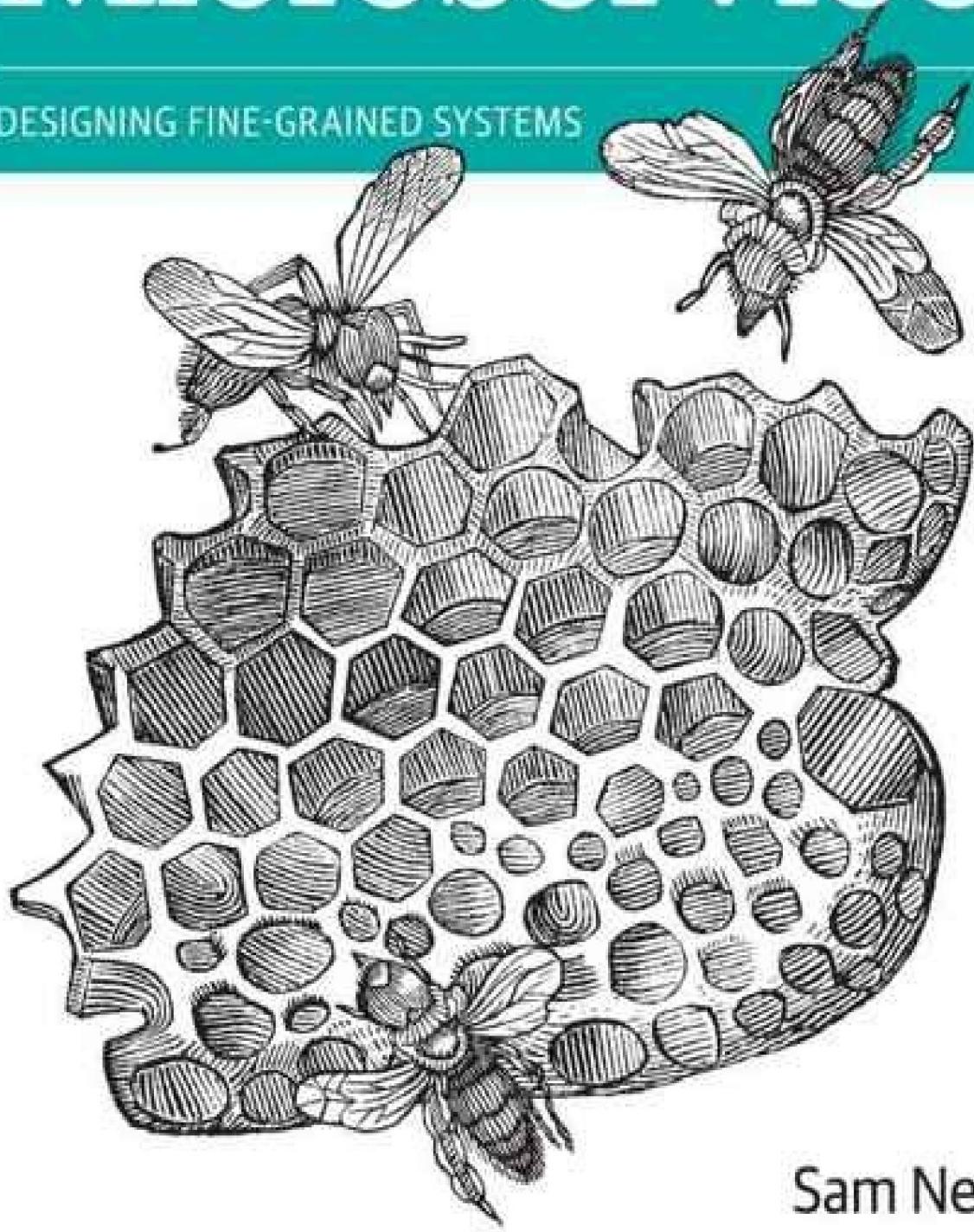


# Building Microservices

DESIGNING FINE-GRAINED SYSTEMS



Sam Newman

# Construyendo microservicios

Sam Newman

Construyendo microservicios

Por Sam Newman

Copyright © 2015 Sam Newman. Todos los derechos reservados.

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 para uso educativo, comercial o promocional de ventas.

También están disponibles ediciones en línea 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](mailto:corporate@oreilly.com).

- Editores: Mike Loukides y Brian MacDonald
- Editor de producción: Kristen Brown
- Correctora de estilo: Rachel Monaghan
- Correctora de pruebas: Jasmine Kwityn
- Indizador: Judith McConville
- Diseñador de interiores: David Futato
- Diseñador de portada: Ellie Volckhausen
- Ilustradora: Rebecca Demarest
- Febrero de 2015: Primera edición

## Historial de revisiones de la primera edición

- 30-01-2014: Primer lanzamiento

Consulte <http://oreilly.com/catalog/errata.csp?isbn=9781491950357> para detalles del lanzamiento.

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

Si bien el editor y el autor han hecho todo lo posible para garantizar que la información y las instrucciones contenidas en este trabajo sean precisas, el editor y el autor no se responsabilizan de errores u omisiones, incluida, entre otras, la responsabilidad por daños que resulten del uso o la confianza depositada en este trabajo. El uso de la información y las instrucciones contenidas en este trabajo se realiza bajo su propio riesgo. Si algún ejemplo de código u otra tecnología que este trabajo contiene o describe está sujeto a licencias de código abierto o a los derechos de propiedad intelectual de terceros, es su responsabilidad asegurarse de que su uso cumpla con dichas licencias o derechos.

978-1-491-95035-7

[LSI]

# Prefacio

---

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

Este libro está lleno 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, como también lo son las implicaciones de las arquitecturas de microservicios de grano fino. Como tal, debería resultar atractivo para las 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 fino, ya sea para una aplicación nueva o como parte de la descomposición de un sistema existente más monolítico, encontrarán muchos consejos prácticos que los ayudarán. 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 tú.

## ¿Por qué escribí este libro?

Comencé 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 la automatización de la infraestructura, las pruebas y las técnicas de entrega continua pueden ayudar, si el diseño fundamental del sistema no facilita la realización de cambios, entonces existen límites a 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 una mejor escalabilidad, una mayor autonomía de los equipos o una adopción más sencilla de nuevas tecnologías. Mis propias experiencias, así como las de mis colegas de ThoughtWorks y de otros lugares, reforzaron el hecho de que el uso de una mayor cantidad de servicios con sus propios ciclos de vida independientes dio lugar a más dolores de cabeza que había que afrontar. En muchos sentidos, este libro fue concebido como una ventanilla única que ayudaría a abarcar la amplia variedad de temas necesarios para comprender los microservicios, algo que me habría ayudado mucho en el pasado.

## Unas palabras sobre los microservicios hoy

Los microservicios son un tema que evoluciona rápidamente. Aunque la idea no es nueva (aunque el término en sí lo sea), las experiencias de personas de todo el mundo, junto con el surgimiento de nuevas tecnologías, están teniendo un profundo efecto en la forma en que se utilizan. Debido al rápido ritmo de cambio, he intentado centrar este libro en ideas más que en tecnologías específicas, sabiendo que los detalles de implementación siempre cambian más rápido que las ideas detrás de ellos. Sin embargo, espero que dentro de unos años aprendamos aún más sobre dónde encajan los microservicios y cómo utilizarlos bien.

Si bien he hecho todo lo posible para resumir la esencia del tema en este libro, si este tema le interesa, ¡prepárese para muchos años de aprendizaje continuo para mantenerse al tanto de la tecnología más avanzada!

## Navegando por este libro

Este libro está organizado principalmente en un formato basado en temas. Por lo tanto, es posible que desees abordar los temas específicos que más te interesen. Si bien he hecho todo lo posible por 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. Sin duda, te sugeriría que eches un vistazo al [Capítulo 2](#), que aborda la amplitud del tema y proporciona un marco para mi forma de abordar las cosas en caso de que quieras profundizar en algunos de los temas posteriores.

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

A continuación se muestra 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 que enfrentamos a la hora de hacer concesiones como arquitectos y cubre específicamente cuántas cosas debemos tener en cuenta con respecto a los microservicios.

### [Capítulo 3, Cómo modelar servicios](#)

Aquí comenzaremos a definir el límite de los microservicios, utilizando técnicas de diseño impulsado por el dominio para ayudarnos a enfocar nuestro pensamiento.

### [Capítulo 4, Integración](#)

Aquí es donde comenzamos a profundizar un poco más en las implicaciones tecnológicas específicas, ya que analizamos qué tipos 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, La división del monolito](#)

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

### [Capítulo 6, Implementación](#)

Aunque este libro es principalmente teórico, pocos temas en él se han visto tan impactados 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 especial interés cuando se maneja la implementación de múltiples servicios discretos. Cabe destacar especialmente el papel que pueden desempeñar los contratos impulsados por el consumidor para ayudarnos a garantizar la calidad de nuestro software.

## Capítulo 8, Monitoreo

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 de grano fino 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 informática, uno que se ignora con demasiada facilidad. Aunque no soy un experto en seguridad, espero que este capítulo al menos te ayude a considerar algunos de los aspectos que debes tener en cuenta al crear sistemas, y sistemas de microservicios en particular.

## Capítulo 10, Ley de Conway y diseño de sistemas

Este capítulo se centra en la interacción entre la estructura y la arquitectura organizacional. Muchas organizaciones se han dado cuenta de que surgirán problemas si no se mantiene la armonía entre ambos. Intentaremos llegar al fondo de este dilema y consideraremos 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 considerar 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, Uniendo todo

El capítulo final intenta resumir la esencia 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.

## Convenciones utilizadas en este libro

En este libro se utilizan las siguientes convenciones tipográficas:

### Itálico

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

### Ancho constante

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

### Ancho constante en negrita

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.

## Libros en línea de Safari®

### NOTA

Libros de Safari en Línea es una biblioteca digital a pedido que ofrece contenido especializado tanto en formato de libro como de vídeo, de los autores más destacados del mundo en tecnología y negocios.

Los profesionales de la tecnología, desarrolladores de software, diseñadores web y profesionales de negocios y creativos 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 [la empresa, gobierno, educación](#), y los individuos.

Los miembros tienen acceso a miles de libros, videos de capacitación y manuscritos previos a su publicación en una base de datos totalmente buscable 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

Por favor, dirija sus comentarios y preguntas sobre este libro al editor:

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

Contamos con una página web para este libro, donde enumeramos 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](mailto:bookquestions@oreilly.com).

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

Encuéntrenos 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 emprender este viaje, me apoyó durante el proceso de escritura, a menudo estresante, y es la mejor compañera que podría haber pedido. También me gustaría dedicárselo a mi padre, Howard Newman, que siempre ha estado ahí para mí. Esto es para los dos.

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

Además, muchas otras personas proporcionaron ayuda y comentarios sobre las primeras versiones del libro. En concreto, 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 de ThoughtWorks y de toda la industria que me han ayudado a llegar hasta aquí.

Por último, me gustaría agradecer a todas las personas de O'Reilly, incluido Mike Loukides por permitirme sumarme, a mi editor Brian MacDonald, a Rachel Monaghan, Kristen Brown, Betsy Waliszewski y a todas las demás personas que me han ayudado de maneras que quizás nunca conozca.

# Capítulo 1. Microservicios

---

Durante muchos años hemos estado buscando mejores formas de construir sistemas. Hemos estado aprendiendo de lo que ha sucedido antes, adoptando nuevas tecnologías y observando cómo una nueva ola de empresas tecnológicas operan 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 Domain-Driven Design (Addison-Wesley) de Eric Evans 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 eficaz y eficiente, inculcándonos la idea de que debemos tratar cada registro como un candidato a lanzamiento. Nuestra comprensión de cómo funciona la Web nos ha llevado a desarrollar mejores formas de hacer que las máquinas se comuniquen con otras máquinas. El concepto de **arquitectura hexagonal** de Alistair Cockburn Nos ayudó a alejarnos de las arquitecturas en capas donde la lógica empresarial podía ocultarse. Las plataformas de virtualización nos permitieron aprovisionar y redimensionar nuestras máquinas a voluntad, y la automatización de la infraestructura nos brindó una forma de manejar estas máquinas a escala. Algunas organizaciones grandes y exitosas como Amazon y Google adoptaron la idea de que los equipos pequeños fueran dueños del ciclo de vida completo de sus servicios. Y, más recientemente, Netflix compartió con nosotros formas de construir sistemas antifrágiles a una escala que habría sido difícil de comprender hace solo 10 años.

Diseño impulsado por dominios. Entrega continua. Virtualización a pedido. Automatización de infraestructura. Pequeños equipos autónomos. Sistemas a escala. Los microservicios han surgido de este mundo. No se inventaron ni se describieron antes de los hechos; surgieron como una tendencia o un patrón a partir del uso en el mundo real. Pero existen solo por todo lo que ha sucedido antes.

A lo largo de este libro, extraeré algunos aspectos de este trabajo anterior para ayudar a pintar un panorama de cómo construir, gestionar y desarrollar microservicios.

Muchas organizaciones han descubierto que al adoptar arquitecturas de microservicios de grano fino, pueden entregar software más rápidamente y adoptar tecnologías más nuevas. Los microservicios nos brindan mucha más libertad para reaccionar y tomar decisiones diferentes, 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 trabajan juntos. Analicemos un poco esa definición y consideremos las características que los diferencian.

# Pequeño y centrado en hacer una cosa bien

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

Dentro de un sistema monolítico, luchamos contra estas fuerzas intentando garantizar que nuestro código sea más coherente, a menudo mediante la creación de abstracciones o módulos. La cohesión (el impulso de 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: "Reúne aquellas cosas que cambian por la misma razón y separa aquellas cosas que cambian por diferentes razones".

Los microservicios adoptan este mismo enfoque para los servicios independientes. Centramos los límites de nuestros servicios en los límites de la empresa, lo que hace evidente dónde se encuentra el código para una determinada pieza de funcionalidad. Y al mantener este servicio centrado en un límite explícito, evitamos la tentación de que crezca demasiado, con todas las dificultades asociadas que esto puede introducir.

La pregunta que me hacen a menudo es: ¿cuánto 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 un tanto trillada que puedo dar es: lo suficientemente pequeño y no más pequeño. Cuando hablo en conferencias, casi siempre pregunto quién tiene un sistema que es demasiado grande y que le gustaría descomponer. Casi todos levantan la mano. Parece que tenemos una idea muy clara de lo que es demasiado grande, por lo que se podría argumentar que una vez que un fragmento de código ya no parece demasiado grande, probablemente sea lo suficientemente pequeño.

Un factor importante que nos ayudará a responder a la pregunta ¿qué tan pequeño es? es qué tan bien se alinea el servicio con las estructuras del equipo. Si la base de código es demasiado grande para que la gestione un equipo pequeño, es muy sensato intentar dividirla. Hablaremos más sobre la alineación organizacional más adelante.

Cuando se trata de cuán pequeño es suficientemente pequeño, me gusta pensar en estos términos: cuanto más pequeño sea el servicio, más se maximizarán los beneficios y las desventajas de la arquitectura de microservicios.

A medida que se hace más pequeño, aumentan los beneficios en torno a la interdependencia, pero también aumenta parte de la complejidad que surge de tener cada vez más partes móviles, algo que exploraremos a lo largo de este libro. A medida que mejore su capacidad para manejar esta complejidad, podrá esforzarse por lograr 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 agrupar 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 sea mucho más fácil razonar sobre nuestro sistema distribuido y las tecnologías más nuevas pueden mitigar muchos de los desafíos asociados con esta forma de implementación.

Toda la comunicación entre los servicios se realiza mediante llamadas de red, para reforzar la separación entre los servicios y evitar los peligros de un acoplamiento estrecho.

Estos servicios deben poder cambiar independientemente unos de otros y desplegarse por sí solos sin necesidad de que los consumidores cambien. Tenemos que pensar en lo que nuestros servicios deben exponer y lo que deben permitir que se oculte. Si hay demasiado intercambio, nuestros servicios de consumo se acoplan a nuestras representaciones internas. Esto disminuye nuestra autonomía, ya que requiere una coordinación adicional con los consumidores al 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 la adecuada para garantizar que esto no acople 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 las API buenas y desacopladas a lo largo de este libro.

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

Para que la disociación sea eficaz, tendrás que modelar correctamente tus servicios y conseguir las API adecuadas. Hablaré mucho de eso.

## Beneficios clave

Los beneficios de los microservicios son muchos y variados. Muchos de estos beneficios se pueden atribuir a cualquier sistema distribuido. Sin embargo, los microservicios tienden a lograr estos beneficios en mayor medida, principalmente debido a que llevan al extremo los conceptos que sustentan los sistemas distribuidos y la arquitectura orientada a servicios.

## Heterogeneidad tecnológica

Con un sistema compuesto por múltiples servicios que colaboran, podemos decidir utilizar diferentes tecnologías dentro de cada uno de ellos. 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 las distintas 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 tal vez 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](#).

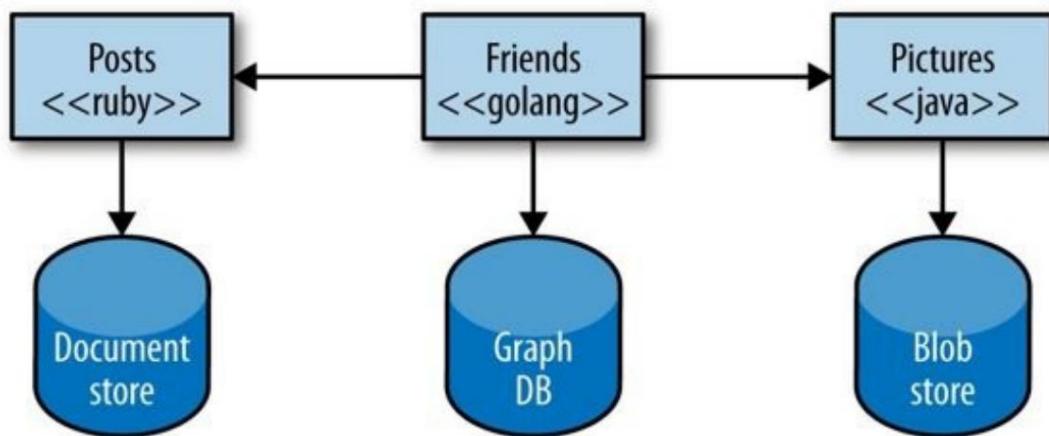


Figura 1-1. Los microservicios pueden permitirle adoptar más fácilmente diferentes tecnologías

Con los microservicios, también podemos adoptar tecnología más rápidamente y comprender cómo los nuevos avances pueden ayudarnos. Una de las mayores barreras para probar y adoptar nuevas tecnologías son los riesgos asociados a 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 pieza de tecnología. Puedo elegir un servicio que sea quizás el de menor riesgo y usar la tecnología allí, sabiendo que puedo limitar cualquier posible impacto negativo. Muchas organizaciones consideran que esta capacidad de absorber más rápidamente nuevas tecnologías es una verdadera ventaja para ellas.

Por supuesto, la adopción de múltiples tecnologías conlleva ciertos costos. Algunas organizaciones optan por imponer algunas restricciones a la elección de lenguajes. Netflix y Twitter, por ejemplo, utilizan principalmente la máquina virtual Java (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 dificultan el trabajo de los servicios o clientes que no están basados en Java. Pero ni Twitter ni Netflix utilizan una única pila de tecnología para todos los trabajos. Otro contrapunto a las preocupaciones

La clave para combinar distintas tecnologías es el tamaño. Si realmente puedo reescribir mi microservicio en dos semanas, es posible que se mitiguen los riesgos de adoptar una nueva tecnología.

Como descubrirá a lo largo de este libro, al igual que ocurre con muchos aspectos relacionados con los microservicios, se trata de encontrar el equilibrio adecuado. Analizaremos cómo tomar decisiones tecnológicas en [el Capítulo 2](#), que se centra en la arquitectura evolutiva; y en [el Capítulo 4](#), que trata sobre la integración, aprenderá a garantizar que sus servicios puedan desarrollar su tecnología de forma independiente sin un acoplamiento excesivo.

## Resiliencia

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

Sin embargo, debemos tener cuidado. Para garantizar que nuestros sistemas de microservicios puedan aprovechar adecuadamente esta mayor resiliencia, debemos comprender las nuevas fuentes de fallas a las que deben enfrentarse los sistemas distribuidos. Las redes pueden fallar y lo harán, 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 los modos de falla en [el Capítulo 11](#).

## Escalada

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

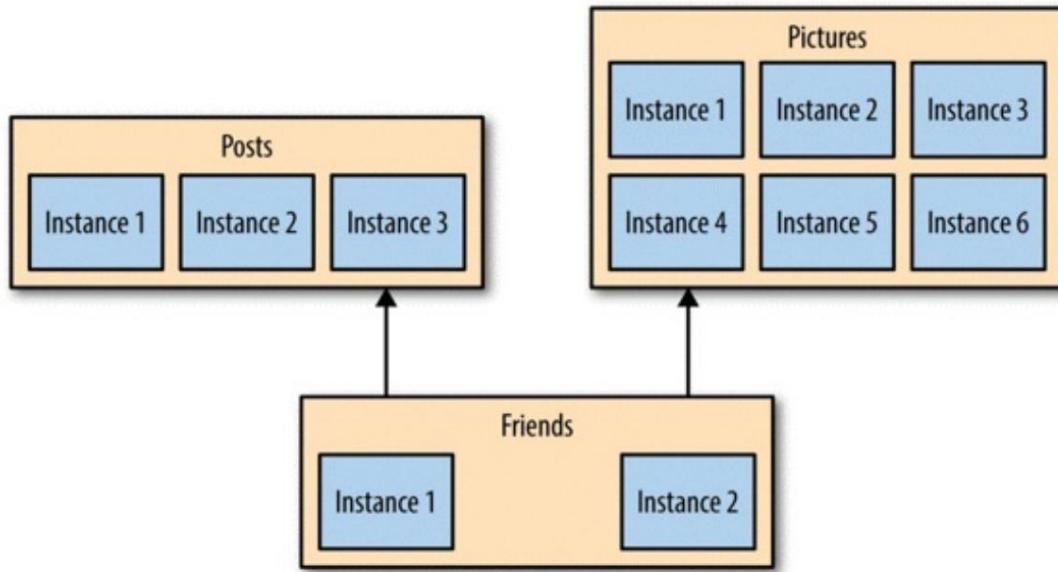


Figura 1-2. Puede enfocar el escalamiento solo en aquellos microservicios que lo necesitan

Gilt, una cadena de tiendas de moda online, adoptó los microservicios precisamente por este motivo. En 2007, Gilt comenzó con una aplicación Rails monolítica, pero en 2009 el sistema de Gilt no pudo hacer frente a la carga que se le imponía. Al dividir las partes principales de su sistema, Gilt pudo lidiar mejor con los picos de tráfico y hoy cuenta con más de 450 microservicios, cada uno de ellos ejecutándose en varias máquinas independientes.

Al adoptar sistemas de aprovisionamiento bajo demanda como los que ofrece Amazon Web Services, podemos incluso aplicar este escalamiento bajo demanda para aquellas partes 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 sola línea en una aplicación monolítica de un millón de líneas requiere que se implemente toda la aplicación para poder lanzar el cambio. Esa podría ser una implementación de gran impacto y alto riesgo. En la práctica, las implementaciones 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 versiones, 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 versiones, mayor será el riesgo de que hagamos algo mal!

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

La tecnología en este espacio ha cambiado mucho en los últimos años y analizaremos más profundamente 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 los equipos grandes y las bases de código grandes. Estos problemas pueden verse exacerbados 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 una base de código para alcanzar el equilibrio perfecto entre tamaño y productividad del equipo. También podemos cambiar la propiedad de los servicios entre equipos para intentar mantener a las personas trabajando en un servicio en el mismo lugar. Analizaremos este tema con más detalle cuando analicemos 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 funcionalidades. 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 la época en la que podíamos pensar de manera estrecha en nuestro sitio web de escritorio o en nuestra aplicación móvil. Ahora tenemos que pensar en las innumerables formas en las que podríamos querer combinar capacidades para la Web, la aplicación nativa, la web móvil, la aplicación para tabletas o los dispositivos portátiles. A medida que las organizaciones dejan de pensar en términos de canales estrechos para adoptar conceptos más holísticos de interacción con el cliente, necesitamos arquitecturas que puedan seguir el ritmo.

Con los microservicios, piense en abrir las costuras de nuestro sistema a las que pueden acceder terceros. A medida que cambian las circunstancias, podemos crear cosas de diferentes maneras. Con una aplicación monolítica, a menudo tengo una costura de grano grueso que se puede utilizar desde el exterior. Si quiero romperla para obtener algo más útil, ¡necesitaré un martillo! En [el Capítulo 5](#), analizaré formas de desmantelar los sistemas monolíticos existentes y, con suerte, transformarlos en microservicios reutilizables y recomponibles.

## Optimización para la reemplazabilidad

Si trabaja en una organización de tamaño mediano o grande, es probable que sepa que hay algún sistema heredado, grande y desagradable, que está en un rincón. Ese que nadie quiere tocar. Ese que es vital para el funcionamiento de su empresa, pero que está escrito en alguna variante extraña de Fortran y solo funciona en hardware que llegó al final de su vida útil hace 25 años. ¿Por qué no se ha reemplazado? Ya sabe por qué: es un trabajo demasiado grande y riesgoso.

Dado que nuestros servicios individuales son de tamaño pequeño, el costo de reemplazarlos por una mejor implementación, o incluso eliminarlos por completo, es mucho más fácil de manejar. ¿Con qué frecuencia ha eliminado más de cien líneas de código en un solo día sin preocuparse 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 reescribiendo por completo los servicios cuando es necesario y eliminando un servicio cuando ya no es necesario. Cuando una base de código tiene solo unos pocos cientos de líneas, es difícil que las personas se apeguen emocionalmente a ella y el costo de reemplazarla es bastante bajo.

## ¿Qué pasa con la arquitectura orientada a servicios?

La arquitectura orientada a servicios (SOA) es un enfoque de diseño en el que varios servicios colaboran para proporcionar un conjunto de capacidades. En este caso, un servicio suele ser un proceso del sistema operativo completamente independiente. La comunicación entre estos servicios se produce a través de llamadas a través de una red, en lugar de llamadas a métodos dentro de un límite de proceso.

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

En esencia, la SOA es una idea muy sensata. Sin embargo, a pesar de los muchos esfuerzos realizados, no hay un consenso sobre cómo implementarla bien. En mi opinión, gran parte de la industria no ha logrado analizar el problema de manera lo suficientemente integral y no ha presentado una alternativa convincente a la narrativa que presentan varios proveedores en este ámbito.

Muchos de los problemas que se le atribuyen a SOA son en realidad problemas con aspectos como los protocolos de comunicación (por ejemplo, SOAP), el middleware de los proveedores, la falta de orientación sobre la granularidad del servicio o la orientación incorrecta sobre la elección de los lugares donde dividir el sistema. Abordaremos cada uno de estos temas a lo largo del resto del libro. Un cínico podría sugerir que los proveedores se apropiaron (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 qué tan grande es demasiado grande. No habla lo suficiente sobre formas prácticas y reales de garantizar que los servicios no se acoplen excesivamente. La cantidad de cosas que no se dicen es donde se originan muchos de los problemas asociados con SOA.

El enfoque de microservicios surgió a partir de su uso en el mundo real, aprovechando nuestra mejor comprensión de los sistemas y la arquitectura para implementar SOA de manera adecuada. 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 ágil.

## Otras técnicas de descomposición

En definitiva, muchas de las ventajas de una arquitectura basada en microservicios provienen de su naturaleza granular y del hecho de que ofrece muchas más opciones para resolver problemas. Pero ¿podrían otras técnicas de descomposición similares lograr los mismos beneficios?

## Bibliotecas compartidas

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

Las bibliotecas ofrecen una manera de compartir funcionalidades entre equipos y servicios. Por ejemplo, podría crear un conjunto de utilidades de recopilación útiles o quizás una biblioteca de estadísticas que se pueda reutilizar.

Los equipos pueden organizarse en torno a estas bibliotecas, y las bibliotecas mismas pueden reutilizarse, pero existen algunas desventajas.

En primer lugar, se pierde la verdadera heterogeneidad tecnológica. La biblioteca normalmente tiene que estar en el mismo lenguaje o, al menos, ejecutarse en la misma plataforma. En segundo lugar, se reduce la facilidad con la que se pueden escalar partes del sistema de forma independiente. Además, a menos que se utilicen bibliotecas vinculadas dinámicamente, no se puede implementar una nueva biblioteca sin volver a implementar todo el proceso, por lo que se reduce la capacidad de implementar cambios de forma aislada.

Y tal vez el problema es que carecen de las costuras obvias alrededor de las cuales se pueden erigir medidas de seguridad arquitectónicas para garantizar la resiliencia del sistema.

Las bibliotecas compartidas tienen su lugar. Te encontrarás creando código para tareas comunes que no son específicas de tu dominio empresarial y que deseas reutilizar en toda la organización, lo que es un candidato obvio para convertirse en una biblioteca reutilizable. Sin embargo, debes tener cuidado. El código compartido que se utiliza para comunicarse entre servicios puede convertirse en un punto de acoplamiento, algo que analizaremos en [el Capítulo 4](#).

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

# Módulos

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

Vale la pena mencionar la Iniciativa de Puerta de Enlace de Código Abierto (OSGI) como un enfoque tecnológico específico para la descomposición modular. Java en sí 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 utiliza como una forma de adaptar un concepto de módulo en Java a través de una biblioteca.

El problema con OSGI es que intenta aplicar cuestiones como la gestión del ciclo de vida de los módulos sin el apoyo suficiente en el propio lenguaje. Esto hace que los autores de los módulos tengan que trabajar más para lograr un aislamiento adecuado de los mismos. Dentro de los límites de un proceso, también es mucho más fácil caer en la trampa de hacer que los módulos estén demasiado acoplados entre sí, lo que causa todo tipo de problemas. Mi propia experiencia con OSGI, que coincide con la de mis 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 entorno 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 más elegante del módulo.

Las capacidades de los módulos de Erlang son impresionantes, sin duda, pero incluso si tenemos la suerte de utilizar una plataforma con estas capacidades, seguimos teniendo las mismas deficiencias que con las bibliotecas compartidas normales. Estamos muy limitados en nuestra capacidad para utilizar nuevas tecnologías, en cómo podemos escalar de forma independiente, podemos derivar hacia técnicas de integración que están demasiado acopladas y carecemos de puntos de apoyo para las medidas de seguridad arquitectónicas.

Hay una última observación que vale la pena compartir. Técnicamente, debería ser posible crear módulos independientes y bien estructurados 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, lo que hace que se pierda uno de sus beneficios clave. Tener una separación de límites de proceso sí refuerza la higiene limpia en este sentido (o al menos hace que sea más difícil hacer lo 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 proceso rara vez se cumplan en el mundo real.

Por lo tanto, si bien la descomposición modular dentro de un límite de proceso puede ser algo que desee hacer además de descomponer su sistema en servicios, por sí sola no ayudará a resolver todo. Si tiene una tienda que utiliza exclusivamente Erlang, la calidad de la implementación del módulo de Erlang

Puede que los módulos los lleven muy lejos, pero sospecho que muchos de ustedes no están en esa situación. Para el resto de nosotros, deberíamos considerar que los módulos ofrecen los mismos tipos de beneficios que las bibliotecas compartidas.

## No hay bala de plata

Antes de terminar, debo señalar que los microservicios no son una panacea ni una solución milagrosa, y que son una mala elección como martillo de oro. Tienen todas las complejidades asociadas a los sistemas distribuidos y, si bien hemos aprendido mucho sobre cómo administrar bien los sistemas distribuidos (algo que analizaremos a lo largo del libro), sigue siendo difícil. Si vienes desde el punto de vista de un sistema monolítico, tendrás que mejorar mucho en el manejo de la implementación, las pruebas y la supervisión para desbloquear los beneficios que hemos cubierto hasta ahora. También tendrás que pensar de manera diferente sobre cómo escalar tus sistemas y asegurarte de que sean resistentes. ¡No te sorprendas si cosas como las transacciones distribuidas o el teorema CAP comienzan a darte 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 en el grado de agresividad con el que puede adoptarlos. A lo largo de cada capítulo de este libro, intentaré ofrecerle orientación destacando los posibles obstáculos, lo que debería ayudarlo a trazar un camino firme.

## Resumen

Con suerte, a esta altura ya sabrá qué es un microservicio, qué lo diferencia de otras técnicas de composición y cuáles son algunas de sus principales ventajas. En cada uno de los siguientes capítulos, analizaremos en más detalle cómo lograr estos beneficios y cómo evitar algunos de los problemas más comunes.

Hay varios temas que abordar, pero es necesario comenzar por algún lado. Uno de los principales desafíos que presentan los microservicios es un cambio en el rol de quienes suelen guiar la evolución de nuestros sistemas: los arquitectos. A continuación, analizaremos algunos enfoques diferentes para este rol que pueden garantizar que aprovechamos al máximo esta nueva arquitectura.

# Capítulo 2. El arquitecto evolutivo

---

Como hemos visto hasta ahora, los microservicios nos ofrecen muchas opciones y, en consecuencia, muchas decisiones que tomar. Por ejemplo, ¿cuántas tecnologías diferentes deberíamos utilizar? ¿Deberíamos permitir que distintos equipos utilicen distintos lenguajes de programación? ¿Deberíamos dividir o fusionar un servicio? ¿Cómo podemos 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 dogmática de cuál es el papel de un arquitecto y, con suerte, lanzaré un último asalto a la torre de marfil.

## Comparaciones inexactas

Sigues usando esa palabra. No creo que signifique lo que tú crees que significa.

Iñigo Montoya, de La princesa prometida

Los arquitectos tienen un trabajo importante. Son los encargados de garantizar que tengamos una visión técnica unificada, que nos ayude a ofrecer el sistema que nuestros clientes necesitan. En algunos lugares, puede que solo tengan que trabajar con un equipo, en cuyo caso el papel del arquitecto y el líder técnico suele ser el mismo. En otros, pueden estar definiendo la visión de un programa de trabajo completo, coordinando con varios equipos en todo el mundo o incluso con una organización entera. Independientemente del nivel en el que operen, el papel es difícil de definir y, a pesar de que suele ser la progresión profesional obvia para los desarrolladores en las organizaciones empresariales, también es un papel que recibe más críticas que prácticamente cualquier otro.

Más que cualquier otra función, 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 frecuencia parecemos equivocarnos en esta función. ¿Por qué?

Nuestra industria es joven. Esto es algo que parecemos olvidar, y sin embargo, sólo llevamos unos 70 años creando programas que funcionan en lo que conocemos como ordenadores.

Por eso, siempre buscamos otras profesiones para intentar explicar lo que hacemos. No somos médicos ni ingenieros, pero tampoco fontaneros ni electricistas.

En cambio, caemos en una especie de punto intermedio, lo que hace que sea difícil para la sociedad entendernos, o para nosotros mismos entender dónde encajamos.

Así que tomamos prestado de otras profesiones. Nos llamamos a nosotros mismos “ingenieros” o “arquitectos” de software. Pero no lo somos, ¿verdad? Los arquitectos e ingenieros tienen un rigor y una disciplina con los que sólo podríamos soñar, y su importancia en la sociedad es bien conocida. Recuerdo haber hablado con un amigo mío el día antes de que se titulara de arquitecto. “Mañana”, me dijo, “si te doy un consejo en el bar 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 cualificado y debería ser responsable si me equivoco”. La importancia de estos trabajos para la sociedad significa que hay cualificaciones obligatorias que las personas deben cumplir. En el Reino Unido, por ejemplo, se exige un mínimo de siete años de estudio antes de poder ser considerado arquitecto. Pero estos trabajos también se basan en un conjunto de conocimientos que se remontan a miles de años. ¿Y nosotros? No exactamente. Por eso también considero que la mayoría de las formas de certificación en TI son inútiles, ya que sabemos muy poco sobre 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 ansiamos como industria. Pero esto puede ser doblemente dañino. En primer lugar, implica que sabemos lo que estamos haciendo, cuando claramente no es así. No diría que los edificios y los puentes nunca se caen, pero se caen mucho menos que la cantidad de veces que se estrellan nuestros programas, lo que hace que las comparaciones con los ingenieros sean bastante injustas. En segundo lugar, las analogías se descomponen muy rápidamente si se les da un vistazo superficial. Para cambiar las cosas,

En otras palabras, si construir un puente fuera como programar, a mitad de camino descubriríamos que la otra orilla estaba ahora 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 ser flexible, adaptarse y evolucionar con los requisitos del usuario.

Tal vez el término arquitecto sea el que más daño ha causado. La idea de alguien que traza planos detallados para que otros los interpreten y espera que se lleven a cabo. El equilibrio entre un artista y un ingeniero que supervisa la creación de lo que normalmente es una visión singular, con todos los demás puntos de vista subordinados, salvo la objeción ocasional del ingeniero estructural con respecto a las leyes de la física. En nuestra industria, esta visión del arquitecto conduce a algunas prácticas terribles. Diagrama tras diagrama, página tras página de documentación, creada con la intención de informar la construcción del sistema perfecto, sin tener en cuenta el futuro fundamentalmente incognoscible. Totalmente carente de cualquier comprensión de lo difícil que será implementarlo, o de si realmente funcionará o no, y mucho menos de tener la capacidad de cambiar a medida que aprendemos más.

Cuando nos comparamos con ingenieros o arquitectos corremos el riesgo de perjudicar a todo el mundo. Lamentablemente, por ahora nos quedamos con la palabra arquitecto , así que lo mejor que podemos hacer es redefinir lo que significa en nuestro contexto.

# Una visión evolutiva para el arquitecto

Nuestras necesidades cambian más rápidamente que las de las personas que diseñan y construyen edificios, al igual que las herramientas y técnicas que tenemos a nuestra disposición. Las cosas que creamos no son puntos fijos en el tiempo. Una vez que se lanzan a producción, nuestro software seguirá evolucionando a medida que cambie la forma en que se utiliza. Para la mayoría de las cosas que creamos, tenemos que 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 cambie. Por lo tanto, nuestros arquitectos deben cambiar su forma de pensar y dejar de pensar en crear el producto final perfecto para centrarse en ayudar a crear un marco en el que puedan surgir los sistemas adecuados y seguir creciendo a medida que aprendemos más.

Aunque he dedicado gran parte del capítulo hasta ahora a advertirles que no nos comparemos demasiado con otras profesiones, hay una analogía que me gusta en relación con el papel del arquitecto de TI y que creo que resume mejor lo que queremos que sea este papel. Erik Doernenburg compartió conmigo por primera vez la idea de que deberíamos pensar en nuestro papel más como planificadores urbanos que como arquitectos del entorno construido. El papel del planificador urbano debería resultar familiar para cualquiera de ustedes que haya jugado a SimCity antes. El papel de un planificador urbano es observar 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 actuales, teniendo en cuenta el uso futuro. Sin embargo, la forma en que influye en la evolución de la ciudad es interesante.

No dice: "construye este edificio específico allí", sino que divide una ciudad en zonas. Por lo tanto, como en SimCity, puedes designar una parte de tu ciudad como zona industrial y otra parte como zona residencial. Luego, otras personas son las que deben decidir qué edificios concretos se van a construir, pero hay restricciones: si se quiere construir una fábrica, ésta 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 trasladan las personas y los servicios públicos de una zona a otra.

Más de una persona ha comparado una ciudad con un ser vivo. La ciudad cambia con el tiempo. Se transforma y evoluciona a medida que sus ocupantes la utilizan de diferentes maneras o a medida que fuerzas externas la moldean. El urbanista hace todo lo posible para anticipar estos cambios, pero acepta que intentar ejercer un control directo sobre todos los aspectos de lo que sucede es inútil.

La comparación con el software debería ser obvia. Cuando nuestros usuarios utilizan nuestro software, debemos reaccionar y cambiar. No podemos prever todo lo que sucederá, por lo que en lugar de planificar para cualquier eventualidad, deberíamos planificar para permitir el cambio evitando la urgencia de especificar en exceso cada 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 se adapta 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. Para tomar prestada una expresión de Frank Buschmann, los arquitectos tienen el deber de garantizar que el sistema también sea habitable para los desarrolladores.

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

Como es menos prescriptivo, el número de veces que necesita intervenir para corregir la dirección debería ser mínimo, pero si alguien decide construir una planta de tratamiento de aguas residuales en una zona residencial, debe poder cerrarla.

Por lo tanto, nuestros arquitectos, como planificadores urbanos, deben establecer una dirección a grandes rasgos y solo participar en la implementación de detalles muy específicos en casos limitados. Deben asegurarse de que el sistema sea adecuado para el propósito actual, pero también una plataforma para el futuro. Y deben asegurarse de que sea un sistema que haga felices a los usuarios y a los desarrolladores por igual. Parece una tarea bastante difícil. ¿Por dónde empezamos?

## Zonificación

Por lo tanto, para continuar con la metáfora del arquitecto como planificador urbano por un momento, ¿cuáles son nuestras zonas? 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 se comunican nuestros servicios entre sí, o asegurarnos de que podemos monitorear adecuadamente el estado general de nuestro sistema. El grado de participación 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 usted está en una organización de este tipo, dependerá más del equipo para tomar la decisión local correcta.

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

Dentro de cada servicio, puede que no le importe que el equipo que posee esa zona elija una pila de tecnología o un almacén de datos diferente. Por supuesto, aquí pueden surgir otras preocupaciones. 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 trasladarlas entre equipos si tiene que dar soporte a 10 pilas de tecnología diferentes. De manera similar, si cada equipo elige un almacén de datos completamente diferente, puede que le falte la experiencia suficiente para ejecutar cualquiera de ellos a gran escala.

Netflix, por ejemplo, ha estandarizado en gran medida Cassandra como tecnología de almacenamiento de datos. Si bien puede no ser la mejor opción para todos sus casos, Netflix considera que el valor obtenido al desarrollar 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 determinadas tareas.

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

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

#### EL ARQUITECTO DE CODIFICACIÓN

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 y, idealmente, debería significar que estos desarrolladores también pasen tiempo codificando con el equipo. Para quienes practican la programación en pareja, resulta sencillo que un arquitecto se una a un equipo durante un breve período como miembro de la pareja. Lo ideal es que trabaje en historias normales para comprender 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, eso depende en gran medida del tamaño del equipo o los equipos con los que trabaja. Pero la clave es que debe ser una actividad rutinaria. Si trabaja con cuatro equipos, por ejemplo, pasar medio día con cada equipo cada cuatro semanas garantiza que se genere conciencia y se mejoren las comunicaciones con los equipos con los que trabaja.

## Un enfoque basado en principios Las

reglas son para la obediencia de los tontos y la guía de los hombres sabios.

Generalmente atribuido a Douglas Bader

La toma de decisiones en el diseño de sistemas se basa en concesiones, y las arquitecturas de microservicios nos plantean muchas concesiones. Al elegir un almacén de datos, ¿elegimos una plataforma con la que tenemos menos experiencia, pero que nos ofrece una mejor escalabilidad? ¿Está bien que tengamos dos pilas de tecnologías diferentes en nuestro sistema? ¿Y tres? Algunas decisiones se pueden tomar completamente en el momento con la información disponible, y estas son las más fáciles de tomar.

¿Pero qué pasa con aquellas decisiones que tal vez deban tomarse con información incompleta?

Enmarcar este punto 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 alcanzar. Veamos cada uno de ellos por separado.

## Metas estratégicas

El papel del arquitecto ya es bastante abrumador, por lo que, afortunadamente, no solemos tener que definir también objetivos estratégicos. Los objetivos estratégicos deben hablar de hacia dónde se dirige tu empresa y de cómo se ve a sí misma como la mejor manera de hacer felices a sus clientes. Serán objetivos de alto nivel y es posible que no incluyan tecnología en absoluto. Se pueden definir a nivel de empresa o de división. Pueden ser cosas como "Expandirse al sudeste asiático para abrir nuevos mercados" o "Permitir que el cliente logre todo lo posible utilizando el autoservicio". La clave es que hacia ahí se dirige tu organización, por lo que debes asegurarte de que la tecnología esté alineada con ello.

Si usted es la persona que define la visión técnica de la empresa, esto puede significar que tendrá que dedicar más tiempo a las partes no técnicas de su organización (o del negocio, como se las suele llamar). ¿Cuál es la visión que impulsa al negocio? ¿Y cómo cambia?

# Principios

Los principios son reglas que usted ha establecido para alinear lo que está haciendo con un objetivo más amplio y que, 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, independientemente de cualquier otro equipo. Si otro objetivo es que su organización se está moviendo para hacer crecer agresivamente su oferta en otros países, puede decidir implementar un principio que diga que todo el sistema debe ser portátil para permitir que se implemente localmente a fin de respetar la soberanía de los datos.

Probablemente no quieras tener muchos de ellos. Menos de 10 es una buena cantidad: lo suficientemente pequeña para que la gente pueda recordarlos o para que quepan en carteles pequeños. Cuantos más principios tengas, mayor será la probabilidad 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 ayudarte 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 tu aplicación necesita exhibir para funcionar en Heroku. Una restricción es realmente algo que es muy difícil (o prácticamente imposible) de cambiar, mientras que los principios son cosas que decidimos elegir. Puedes decidir mencionar explícitamente aquellas cosas que son principios frente a aquellas que son restricciones, para ayudar a indicar aquellas cosas que realmente no puedes cambiar. Personalmente, creo que puede haber algún valor en mantenerlos en la misma lista para alentar el desafío de las restricciones de vez en cuando y ver si realmente son inamovibles.

## Prácticas

Nuestras prácticas son la manera en que garantizamos que nuestros principios se lleven a cabo. Son un conjunto de pautas prácticas y detalladas para realizar tareas. A menudo, serán específicas de la tecnología y deben ser lo suficientemente sencillas como para que cualquier desarrollador pueda entenderlas. Las prácticas pueden 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 suelen cambiar 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 establezca que los equipos de entrega controlan el ciclo de vida completo de sus sistemas puede significar que usted tiene una práctica que establece que todos los servicios se implementan en cuentas de AWS aisladas, lo que proporciona una gestión de autoservicio de los recursos y aislamiento de otros equipos.

## Combinando principios y prácticas

Los principios de una persona son las prácticas de otra. Por ejemplo, se podría decidir llamar al uso de HTTP/REST un principio en lugar de una práctica. Y eso estaría bien. El punto clave es que es valioso tener ideas generales que guíen la evolución del 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, la combinación de principios y prácticas puede ser adecuada. 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 .NET, por ejemplo, puede tener un conjunto de prácticas y un equipo Java otro, con un conjunto de prácticas comunes para ambos. Sin embargo, los principios pueden 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](#) durante su trabajo 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 del extremo derecho cambiarán con bastante regularidad, mientras que los principios permanecerán bastante estáticos. Un diagrama como este se puede imprimir en una sola hoja de papel y compartir, 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.

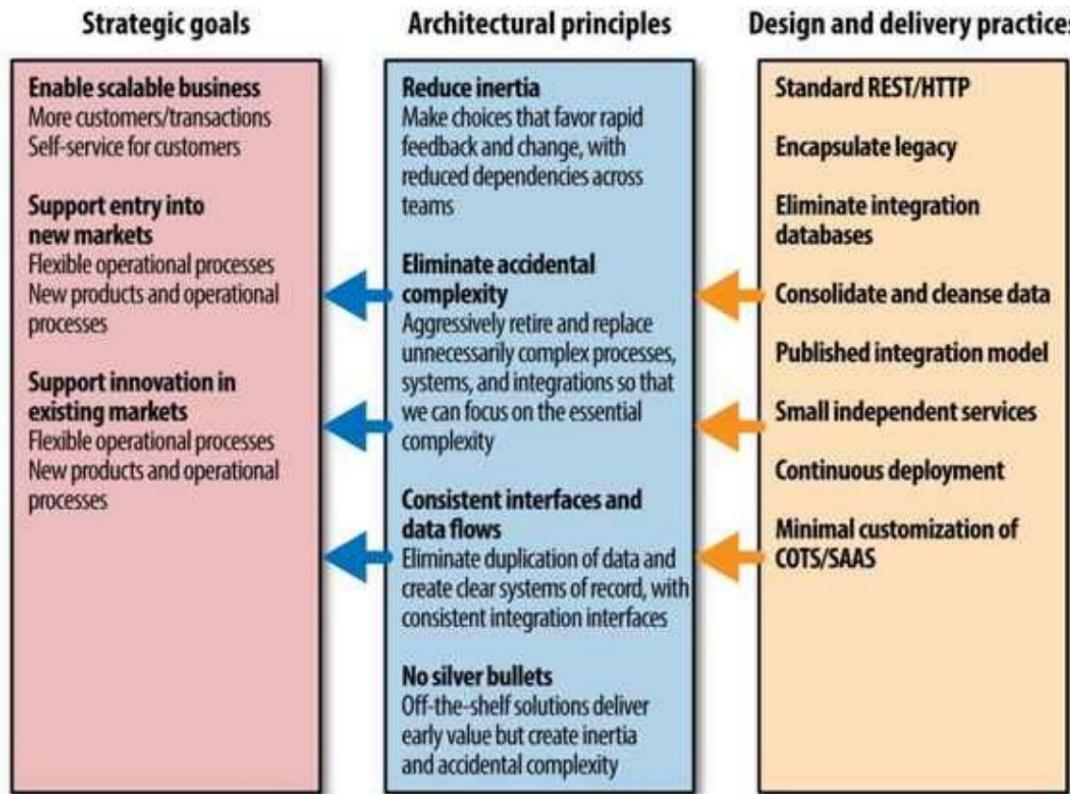


Figura 2-1. Un ejemplo real de principios y prácticas

Tiene sentido tener documentación que respalde algunos de estos elementos. Sin embargo, en general, me gusta la idea de tener un código de ejemplo que se pueda ver, inspeccionar y ejecutar, que incorpore estas ideas. Mejor aún, podemos crear herramientas que hagan lo correcto desde el primer momento. Hablaremos de 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 fundamentales que debe encontrar es cuánta variabilidad permitir en su sistema. Una de las formas clave de identificar lo que debería ser constante de un servicio a otro es definir cómo es un servicio bueno y bien comportado. ¿Qué es un servicio “buen ciudadano” en su sistema? ¿Qué capacidades necesita tener para garantizar que su sistema sea manejable y que un mal servicio no haga caer todo el sistema? Y, como sucede con las personas, lo que es un buen ciudadano en un contexto no refleja cómo es en otro. No obstante, existen algunas características comunes de los servicios que se comportan bien 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 compuesto de muchas partes pequeñas con ciclos de vida autónomos pero que se unan entre sí”. 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.

# Escucha

Es esencial que seamos capaces de elaborar vistas coherentes y transversales de la salud de nuestro sistema. Debe ser una vista de todo el sistema, no de un servicio específico. Como veremos en [el Capítulo 8](#), conocer la salud de un servicio individual es útil, pero a menudo solo cuando se intenta diagnosticar un problema más amplio o comprender una tendencia más amplia. Para que esto sea lo más sencillo posible, sugeriría garantizar que todos los servicios emitan métricas relacionadas con la salud y la supervisión general de la misma manera.

Puede optar por adoptar un mecanismo de envío, en el que cada servicio deba enviar estos datos a una ubicación central. Para sus métricas, esto podría ser Graphite y, para su salud, Nagios. O puede decidir utilizar sistemas de sondeo que extraigan datos de los propios nodos. Pero, sea cual sea su elección, intente mantenerla estandarizada. Haga que la tecnología dentro de la caja sea opaca y no exija que sus sistemas de monitoreo cambien para admitirla. El registro entra en la misma categoría aquí: lo necesitamos en un solo lugar.

## Interfaces

La selección de un pequeño número de tecnologías de interfaz definidas ayuda a integrar nuevos consumidores. Tener un estándar es una buena cifra. Dos tampoco está 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, ¿utilizará verbos o sustantivos? ¿Cómo manejará la paginación de los recursos? ¿Cómo manejará el control de versiones de los puntos finales?

## Seguridad arquitectónica

No podemos permitirnos que un servicio que no se comporte bien arruine la fiesta para todos. Tenemos que asegurarnos de que nuestros servicios se protejan adecuadamente de las llamadas de bajada nocivas. Cuantos más servicios tengamos que no gestionen adecuadamente el posible fallo de las llamadas descendentes, más frágiles serán nuestros sistemas. Esto significa que probablemente querrá exigir, como mínimo, que cada servicio descendente tenga su propio grupo de conexiones, e incluso puede llegar a decir que cada uno también utilice un disyuntor. Esto se tratará con más profundidad cuando analicemos los microservicios a escala en [el Capítulo 11](#).

También es importante respetar las reglas cuando se trata de códigos de respuesta. Si los disyuntores dependen de códigos HTTP y un servicio decide enviar códigos 2XX en caso de error o confunde códigos 4XX con códigos 5XX, estas medidas de seguridad pueden desmoronarse. Se podrían plantear preocupaciones similares incluso si no se utiliza HTTP; conocer la diferencia entre una solicitud que estaba bien y se procesó correctamente, una solicitud que estaba mal 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 estaba inactivo es fundamental para garantizar que podamos fallar rápidamente y rastrear los problemas. Si nuestros servicios no respetan 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 imponerle a los desarrolladores la carga de implementar todas estas cosas estándar que se espera que haga cada servicio. Soy un gran creyente en facilitar que se haga lo correcto. Dos técnicas que he visto que funcionan bien en este caso son el uso de ejemplos y la provisión de plantillas de servicio.

## Ejemplares

La documentación escrita es buena y útil. Veo claramente su valor; después de todo, yo he escrito este libro. Pero a los desarrolladores también les gusta el código, y el código que pueden ejecutar y explorar. Si tienes un conjunto de estándares o mejores prácticas que te gustaría fomentar, entonces es útil tener ejemplos que puedas señalar a la gente. La idea es que la gente no puede equivocarse demasiado simplemente imitando algunas de las mejores partes de tu sistema.

Lo ideal es que se trate de servicios del mundo real que funcionan bien, en lugar de servicios aislados que se implementan simplemente para ser ejemplos perfectos. Al garantizar que sus ejemplos se utilicen realmente, se asegura de que todos los principios que tiene tengan sentido.

## Plantilla de servicio personalizado

¿No sería fantástico si pudieras hacer que fuera realmente fácil para todos los desarrolladores seguir la mayoría de las pautas que tienes con muy poco trabajo? ¿Qué pasaría si, de manera inmediata, los desarrolladores tuvieran la mayor parte del código listo para implementar los atributos básicos que necesita cada servicio?

[Asistente para soltar](#) y [Karyon](#) son dos microcontenedores de código abierto basados en JVM. Funcionan de manera similar, combinando un conjunto de bibliotecas para proporcionar funciones como verificación de estado, servicio HTTP o exposición de métricas. De manera inmediata, tiene un servicio completo con un contenedor de servlets integrado que se puede iniciar desde la línea de comandos. Esta es una excelente manera de comenzar, pero ¿por qué detenerse allí? Mientras está en ello, ¿por qué no toma algo como Dropwizard o Karyon y le agrega más funciones para que sea compatible con su contexto?

Por ejemplo, es posible que desees exigir el uso de disyuntores. En ese caso, puedes integrar una biblioteca de disyuntores como [Hystrix](#). O puede que tenga una práctica en la que todas sus métricas deben enviarse a un servidor Graphite central, por lo que tal vez pueda utilizar una biblioteca de código abierto como [Metrics](#) de Dropwizard. y configurarlo para que, desde el primer momento, los tiempos de respuesta y las tasas de error se envíen automáticamente a una ubicación conocida.

Al adaptar dicha plantilla de servicio a su propio conjunto de prácticas de desarrollo, se asegura de que los equipos puedan comenzar a trabajar más rápido y también de que los desarrolladores tengan que esforzarse para que sus servicios se comporten mal.

Por supuesto, si adoptas múltiples pilas de tecnología dispares, necesitarás una plantilla de servicio correspondiente para cada una. Sin embargo, esta puede ser una forma de restringir sutilmente las opciones de lenguaje en tus equipos. Si la plantilla de servicio interna solo admite Java, es posible que las personas se sientan desanimadas a elegir pilas alternativas si tienen que hacer mucho más trabajo por sí mismas. Netflix, por ejemplo, está especialmente preocupada por aspectos como la tolerancia a fallas, para garantizar que la interrupción de una parte de su sistema no pueda hacer que todo se caiga. Para manejar esto, se ha realizado una gran cantidad de trabajo para garantizar que haya bibliotecas de cliente 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 tendría que reproducir todo este esfuerzo. La principal preocupación de Netflix no es tanto la duplicación de esfuerzos, sino el hecho de que es muy fácil equivocarse. El riesgo de que un servicio no implemente correctamente la tolerancia a fallos es alto si puede afectar a más partes del sistema. Netflix mitiga esto utilizando servicios secundarios, que se comunican localmente con una JVM que utiliza las bibliotecas adecuadas.

Hay que 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 deben hacerse las cosas, aunque sea mediante código. Definir las prácticas que utiliza debe ser una actividad colectiva, por lo que lo ideal sería que su(s) equipo(s) asuman la responsabilidad conjunta de actualizar esta plantilla (un enfoque interno de código abierto funciona bien en este caso).

También he visto cómo se desplomaba la moral y la productividad de muchos equipos al tener que aplicarles un marco obligatorio. En un esfuerzo por mejorar la reutilización del código, se asigna cada vez más trabajo a un marco centralizado hasta que se convierte en una monstruosidad abrumadora. Si decide utilizar una plantilla de servicio personalizada, piense con mucho cuidado cuál es su función. Lo ideal sería que su uso fuera 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 orientadora primordial.

También hay que tener 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, de hecho, 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 el sistema, pero esto le preocupa menos que el peligro del 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 DRY (no se repita) resulte en un sistema excesivamente acoplado. Este es un tema con matices, 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 cumplir al pie de la letra nuestra visión técnica. A menudo, tenemos que tomar la decisión de recortar algunos aspectos para sacar algunas características urgentes. Se trata de una disyuntiva más que tendremos que afrontar. Nuestra visión técnica existe por una razón. Si nos desviamos de esta razón, puede que tengamos un beneficio a corto plazo, pero un coste a largo plazo. Un concepto que nos ayuda a entender esta disyuntiva es la deuda técnica. Cuando acumulamos deuda técnica, al igual que la deuda en el mundo real, tiene un coste continuo y es algo que queremos pagar.

A veces, la deuda técnica no es algo que simplemente provocamos al tomar atajos. ¿Qué sucede si nuestra visión del sistema cambia, pero no todo el sistema coincide con ella? En esta situación, también hemos creado nuevas fuentes de deuda técnica.

El trabajo del arquitecto es observar el panorama general y comprender este equilibrio. Es importante tener una visión del nivel de deuda y de dónde intervenir. Según la organización, es posible que pueda brindar una orientación amable, pero que los equipos decidan por sí mismos cómo realizar un seguimiento y pagar la deuda. En el caso de otras organizaciones, es posible que deba ser más estructurado y, tal vez, mantener un registro de la deuda que se revise periódicamente.

## Manejo de excepciones

Por lo tanto, nuestros principios y prácticas guían la manera en que se deben construir 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 registrar dicha decisión en un registro en algún lugar para futuras referencias. Si se encuentran suficientes excepciones , puede que tenga 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 un almacenamiento altamente escalable, en cuyo caso 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 en las que los equipos de desarrollo tienen un alto grado de confianza y autonomía, y allí los principios son livianos (y la necesidad de un manejo manifiesto de excepciones 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 establecidas reflejen adecuadamente los desafíos que enfrentan las personas. Dicho todo esto, soy partidario 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, entonces los microservicios pueden no ser 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 Tecnologías Relacionadas (COBIT) tienen una definición bastante buena:

La gobernanza garantiza que se logren los objetivos de la empresa evaluando las necesidades, condiciones y opciones de las partes interesadas; estableciendo una dirección a través de la priorización y la toma de decisiones; y monitoreando el desempeño, el cumplimiento y el progreso en relación con la dirección y los objetivos acordados.

### COBIT 5

La gobernanza puede aplicarse a múltiples aspectos en el ámbito de las TI. Queremos centrarnos en el aspecto de la gobernanza técnica, algo que considero que es tarea del arquitecto. Si una de las tareas del arquitecto es garantizar que exista una visión técnica, entonces la gobernanza consiste en garantizar que lo que estamos construyendo coincida con esa visión y hacerla evolucionar si es necesario.

Los arquitectos son responsables de muchas cosas. Deben asegurarse de que exista un conjunto de principios que puedan guiar el desarrollo y de que estos principios coincidan con la estrategia de la organización. También deben asegurarse de que estos principios no requieran prácticas laborales que hagan que los desarrolladores se sientan miserables. Deben mantenerse al día con las nuevas tecnologías y saber cuándo hacer las concesiones adecuadas. Esto supone una enorme responsabilidad. Y además, deben llevar a gente con ellos, es decir, asegurarse de que los colegas con los que trabajan comprendan las decisiones que se toman y estén involucrados para llevarlas a cabo. Ah, y como ya hemos mencionado: deben pasar algún tiempo con los equipos para comprender el impacto de sus decisiones, y quizás incluso del código también.

¿Es una tarea difícil? Sin duda, pero estoy firmemente convencido de que no deberían hacerlo solos.

Un grupo de gobernanza que funcione correctamente puede trabajar en conjunto para compartir el trabajo y dar forma a la visión.

Normalmente, la gobernanza es una actividad grupal. Puede ser una charla informal con un equipo lo suficientemente pequeño o una reunión regular más estructurada con miembros formales del grupo para un alcance mayor. Aquí es donde creo que se deben discutir los principios que abordamos anteriormente y cambiarlos según sea necesario. Este grupo debe estar dirigido por un tecnólogo y debe estar compuesto principalmente por personas que estén ejecutando el trabajo que se está gobernando. Este grupo también debe ser responsable del seguimiento y la gestión de los riesgos técnicos.

Un modelo que me gusta mucho es que el arquitecto presida el grupo, pero que la mayor parte del grupo esté formado por los tecnólogos de cada equipo de entrega (los líderes de cada equipo, como mínimo). El arquitecto es responsable de garantizar que el grupo funcione, pero el grupo en su conjunto es responsable de la gobernanza. Esto comparte la carga y garantiza que haya un mayor nivel de aceptación. También garantiza 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 ese punto, ¿qué debe hacer el arquitecto? Habiendo estado en esta posición antes, puedo decirles que esta es una de las situaciones más difíciles a las que se puede enfrentar. A menudo, adopto el enfoque de que debo aceptar la decisión del grupo. Considero que he hecho 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 me han demostrado que estaba equivocado más de una vez! E imaginénlo desmoralizante que puede ser para un grupo haber tenido espacio para llegar a una decisión y luego, al final, ser ignorado. Pero a veces he hecho caso omiso del grupo. Pero ¿por qué y cuándo? ¿Cómo se eligen las líneas?

Piensa en enseñar a los niños a montar en bicicleta. No puedes montarla por ellos. Los ves tambalearse, pero si intervienes cada vez que pareciese que se van a caer, entonces nunca aprenderían, y en cualquier caso, se caen mucho menos de lo que crees que lo harán. Pero si los ves a punto de desviarse hacia el tráfico o hacia un estanque de patos cercano, entonces tienes que intervenir.

Del mismo modo, como arquitecto, debes tener una idea clara de cuándo, en sentido figurado, tu equipo se está metiendo en un lío. También debes ser consciente de que, incluso si sabes que tienes razón y desautorizas al equipo, esto puede socavar tu posición y hacer que el equipo sienta que no tiene voz ni voto. A veces, lo correcto es aceptar una decisión con la que no estás de acuerdo. Saber cuándo hacerlo y cuándo no es difícil, pero a veces es vital.

## Construyendo un equipo

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

Las personas con las que trabajas son las que harán el trabajo. Gran parte del papel del líder técnico consiste en ayudar a que crezcan (para que comprendan la visión por sí mismos) y también en garantizar que puedan participar activamente en la configuración e implementación de la visión.

Ayudar a las personas que te rodean en su propio crecimiento profesional puede adoptar muchas formas, la mayoría de las cuales quedan fuera del alcance de este libro. Sin embargo, hay un aspecto en el que una arquitectura de microservicios es especialmente relevante. Con sistemas más grandes y monolíticos, hay menos oportunidades para que las personas den un paso adelante y se apropien 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 dar un paso adelante haciéndoles asumir la propiedad de servicios individuales antes de aceptar más responsabilidad puede ser una excelente manera de ayudarlas a lograr sus propias metas profesionales y, al mismo tiempo, alivia la carga de quien esté a cargo.

Creo firmemente que el gran software surge de grandes personas. Si solo te preocupas por el aspecto tecnológico de la ecuación, te estás perdiendo mucho más de la mitad del panorama.

# Resumen

Para resumir este capítulo, aquí están las que considero como las principales responsabilidades del arquitecto evolutivo:

## Visión

Asegúrese de que exista 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

Comprenda el impacto de sus decisiones en sus clientes y colegas

## Colaboración

Interactúe 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

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

El arquitecto evolutivo es aquel que comprende que lograr esta hazaña es un acto de equilibrio constante.

Siempre hay fuerzas que te empujan en una u otra dirección, y comprender dónde rechazarlas o dónde dejarte llevar por la corriente es algo que a menudo solo se adquiere con la experiencia. Pero la peor reacción a todas estas fuerzas que nos empujan hacia el cambio es volvemos más rígidos o estancados en nuestro modo de pensar.

Si bien muchos de los consejos de este capítulo pueden aplicarse a cualquier arquitecto de sistemas, los microservicios nos obligan a tomar muchas más decisiones. Por lo tanto, es fundamental poder equilibrar mejor todas estas ventajas y desventajas.

En el próximo capítulo, aplicaremos parte de nuestro nuevo conocimiento del rol del arquitecto mientras comenzamos a pensar en cómo encontrar los límites adecuados para nuestros microservicios.

# Capítulo 3. Cómo modelar servicios

---

El razonamiento de mi oponente me recuerda al pagano que, cuando se le preguntó sobre qué se sustentaba el mundo, respondió: "Sobre una tortuga". Pero ¿sobre qué se sustenta la tortuga? "Sobre otra tortuga".

José Barker (1854)

Ahora ya sabes qué son los microservicios y, con suerte, tienes una idea de sus principales beneficios. Probablemente ahora estés ansioso por empezar a crearlos, ¿verdad? Pero, ¿por dónde empezar? En este capítulo, veremos cómo pensar en los límites de tus microservicios para maximizar las ventajas y evitar algunas de las posibles desventajas. Pero primero, necesitamos algo con lo que trabajar.

## Presentando 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 en este mundo.

Así que dirijamos nuestra atención al vanguardista minorista online MusicCorp. Hace poco, MusicCorp era un minorista tradicional, pero después de que el negocio de los discos de gramófono se desmoronara, centró cada vez más sus esfuerzos en Internet. La empresa tiene un sitio web, pero cree que ahora es el momento de redoblar sus esfuerzos en el mundo online. Después de todo, esos iPods son sólo una moda pasajera (los Zunes son mucho mejores, obviamente) y los aficionados a la música están muy contentos de esperar a que los CD lleguen a sus puertas. La calidad por encima de la comodidad, ¿no? Y ya que estamos, ¿qué es eso de Spotify del que la gente no para de hablar? ¿Una especie de tratamiento para la piel para adolescentes?

A pesar de estar un poco atrasada, MusicCorp tiene grandes ambiciones. Afortunadamente, decidió que su mejor oportunidad de conquistar el mundo es asegurarse de poder hacer cambios con la mayor facilidad posible. ¡Los microservicios son la clave!

## ¿Qué hace que un servicio sea bueno?

Antes de que el equipo de MusicCorp se aleje y cree un servicio tras otro en un intento de entregar cintas de ocho pistas a todo el mundo, pongamos el freno y hablemos un poco sobre la idea subyacente más importante que debemos tener en mente. ¿Qué hace que un servicio sea bueno? Si ha sobrevivido a una implementación de SOA fallida, puede que tenga una idea de hacia dónde voy a dirigirme a continuación. Pero, por si acaso no tiene tanta (des)afortunada, quiero que se centre en dos conceptos clave: acoplamiento débil y alta cohesión. Hablaremos en detalle a lo largo del libro sobre otras ideas y prácticas, pero todas serán en vano si nos equivocamos en estas dos cosas.

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

## Acoplamiento suelto

Cuando los servicios están acoplados de forma flexible, un cambio en un servicio no debería requerir un cambio en 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 provocan un acoplamiento estrecho? Un error clásico es elegir un estilo de integración que vincule estrechamente un servicio con otro, lo que hace que los cambios dentro del servicio requieran un cambio para los consumidores. Analizaremos 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 potencial problema de rendimiento, la comunicación conversacional puede conducir a un acoplamiento estrecho.

# Alta cohesión

Queremos que los comportamientos relacionados se ubiquen juntos y que los comportamientos no relacionados se ubiquen en otro lugar. ¿Por qué?

Bueno, si queremos cambiar un comportamiento, queremos poder cambiarlo en un solo lugar y publicar ese cambio lo antes posible. Si tenemos que cambiar ese comportamiento en muchos lugares diferentes, tendremos que publicar muchos servicios diferentes (quizás al mismo tiempo) para implementar ese cambio. Realizar cambios en muchos lugares diferentes es más lento e implementar muchos servicios a la vez es riesgoso, y queremos evitar ambas cosas.

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

## El contexto delimitado

El libro Domain-Driven Design (Addison-Wesley) de Eric Evans 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 Evans introduce y que al principio no entendí: el contexto acotado. La idea es que cualquier dominio dado consta de múltiples contextos acotados y que dentro de cada uno residen cosas (Eric usa mucho la palabra modelo , que probablemente sea mejor que cosas) que no necesitan ser comunicadas 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 delimitados que me gusta es “una responsabilidad específica impuesta por límite explícito mediante **solicitudes** de funcionalidad dentro de un contexto delimitado, te comunicas con su modelos. En su libro, Evans utiliza la analogía de las células, donde “las células pueden existir porque sus membranas definen lo que entra y lo que sale y determinan lo que puede pasar”.

Volvamos por un momento al negocio de MusicCorp. Nuestro dominio es el negocio completo en el que operamos. Abarca todo, desde el almacén hasta el mostrador de recepción, desde las finanzas hasta los pedidos. Puede que modelemos todo eso en nuestro software o no, pero ese es, no obstante, el dominio en el que operamos. Pensemos en partes de ese dominio que se parecen a los contextos delimitados a los que se refiere Evans. En MusicCorp, nuestro almacén es un hervidero de actividad: gestionamos los pedidos que se envían (y alguna que otra devolución), recibimos la entrega de nuevo stock, organizamos carreras de carretillas elevadoras, etc. En otros lugares, el departamento de finanzas es quizás menos divertido, pero aún tiene una función muy importante dentro de nuestra organización.

Estos empleados gestionan las nóminas, llevan las cuentas de la empresa y elaboran informes importantes. Muchos informes. Probablemente también tengan juguetes de escritorio interesantes.

## Modelos compartidos y ocultos

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

Ahora bien, el departamento de finanzas no necesita conocer el funcionamiento interno detallado del almacén, pero sí necesita saber algunas cosas; por ejemplo, necesita saber los niveles de existencias para mantener las cuentas actualizadas. [La Figura 3-1](#) muestra un ejemplo de diagrama de contexto. Vemos conceptos que son internos al almacén, como el encargado de la preparación de pedidos (las personas que preparan los pedidos), los estantes que representan las ubicaciones de las existencias, etc. Asimismo, el libro mayor de la empresa es parte integral de las finanzas, pero aquí no se comparte externamente.

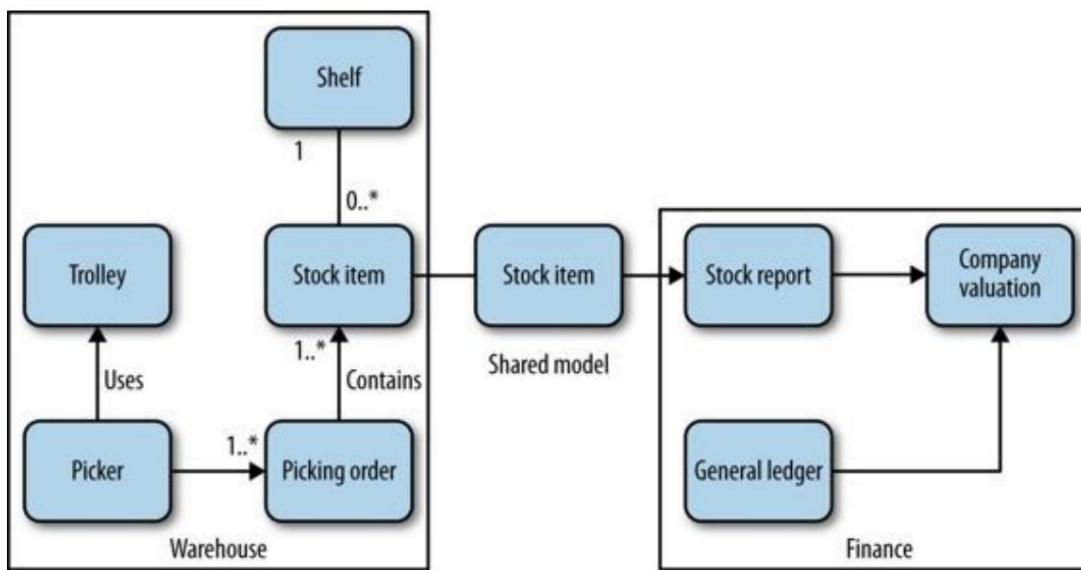


Figura 3-1. Un modelo compartido entre el departamento de finanzas y el almacén

Sin embargo, para poder calcular la valoración de la empresa, los empleados de finanzas necesitan información sobre las existencias que tenemos. El artículo en existencias se convierte entonces en un modelo compartido entre los dos contextos. Sin embargo, tenga en cuenta que no necesitamos exponer ciegamente todo lo relacionado con el artículo en existencias desde el contexto del almacén. Por ejemplo, aunque internamente mantenemos un registro de un artículo en existencias en cuanto a dónde debería estar dentro del almacén, no es necesario exponerlo en el modelo compartido. Por lo tanto, existe la representación interna únicamente y la representación externa que exponemos. En muchos sentidos, esto anticipa el debate sobre REST en [el Capítulo 4](#).

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

Por ejemplo, podemos generar una solicitud de reposición de existencias. El modelo compartido de la devolución 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 con claridad sobre qué modelos se deben compartir y no compartir nuestras representaciones internas, evitamos uno de los posibles obstáculos que pueden derivar en un acoplamiento estrecho (lo opuesto a lo que queremos). También hemos identificado un límite dentro de nuestro dominio donde deberían vivir todas las capacidades empresariales afines, lo que nos da la alta cohesión que queremos.

Estos contextos delimitados se prestan entonces muy bien a ser límites compositivos.

Como comentamos en [el Capítulo 1](#), tenemos la opción de utilizar módulos dentro de un límite de proceso para mantener unido el código relacionado e intentar reducir el acoplamiento con otros módulos del sistema.

Cuando se empieza a trabajar en una nueva base de código, este es probablemente un buen punto de partida. Por lo tanto, una vez que se han encontrado los contextos delimitados en el dominio, hay que asegurarse de que estén modelados dentro de la base de código como módulos, con modelos compartidos y ocultos.

Estos límites modulares se convierten entonces en excelentes candidatos para los microservicios. En general, los microservicios deberían alinearse claramente con los contextos delimitados. Una vez que se vuelva muy competente, puede decidir omitir el paso de mantener el contexto delimitado modelado como un módulo dentro de un sistema más monolítico y pasar 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 stabilicen mientras se familiariza con un nuevo dominio. Hablaremos más de esto en [el Capítulo 5](#), junto con técnicas para ayudar a dividir los sistemas existentes en microservicios.

Entonces, si nuestros límites de servicio se alinean con los contextos delimitados en nuestro dominio, y nuestros microservicios representan esos contextos delimitados, estamos en un excelente comienzo para garantizar que nuestros microservicios estén acoplados de forma flexible y sean fuertemente cohesivos.

## Descomposición prematura

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

Aunque hubo cierta 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 ámbito de las herramientas de CD los animó a avanzar más rápidamente en la identificación de límites y en la construcción de 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 diferentes como para que la idea inicial sobre los límites del servicio no fuera del todo correcta. Esto llevó a realizar muchos cambios en los servicios y a un alto costo asociado por los cambios.

Finalmente, el equipo volvió a fusionar los servicios en un sistema monolítico, lo que les dio tiempo para comprender mejor dónde deberían estar los límites. Un año después, el equipo pudo dividir el sistema monolítico en microservicios, cuyos límites resultaron ser mucho más estables. Este no es ni de lejos el único ejemplo de esta situación que he visto.

Descomponer un sistema en microservicios de forma prematura puede resultar costoso, especialmente si eres nuevo en el ámbito. En muchos sentidos, tener una base de código existente que deseas descomponer en microservicios es mucho más fácil que intentar pasar a los microservicios desde el principio.

## Capacidades empresariales

Cuando empiece a pensar en los contextos limitados que existen en su organización, no debería pensar en términos de datos compartidos, sino en las capacidades que esos contextos proporcionan al resto del dominio. El almacén puede proporcionar la capacidad de obtener una lista de existencias actualizada, 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). Por lo tanto, 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 quedarán expuestas a través de la red a otros colaboradores.

## Tortugas hasta el fondo

Al principio, probablemente identificará una serie de contextos delimitados de grano grueso. Pero estos contextos delimitados pueden, a su vez, contener otros contextos delimitados. 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, piense primero en términos de contextos más amplios y de grano grueso, y luego subdivida a lo largo de estos contextos anidados cuando busque los beneficios de dividir estas uniones.

He visto que estos contextos anidados permanecen ocultos para otros microservicios que colaboran entre sí con gran éxito. Para el mundo exterior, siguen haciendo uso de las capacidades empresariales del almacén, pero no saben que sus solicitudes se están asignando de forma transparente a dos o más servicios independientes, como se 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, puede dividir el inventario, el cumplimiento de pedidos y la recepción de mercancías.

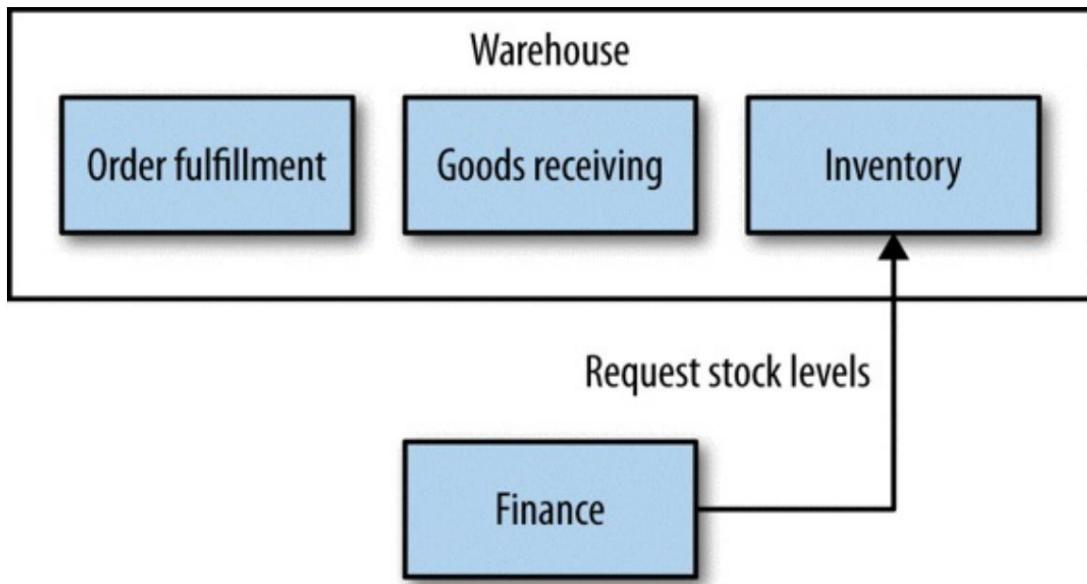


Figura 3-2. Microservicios que representan contextos delimitados y anidados ocultos dentro del almacén

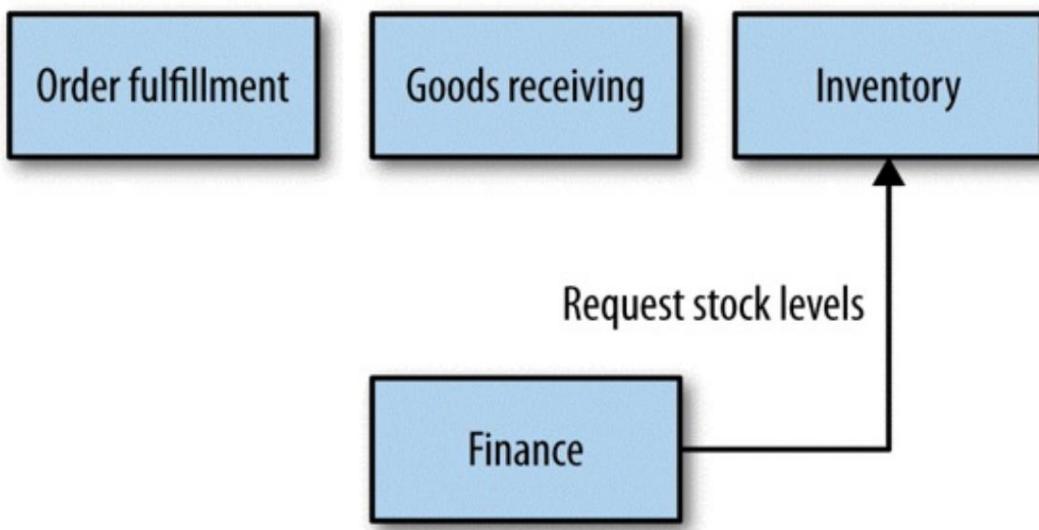


Figura 3-3. Los contextos delimitados dentro del almacén se muestran en sus propios contextos de nivel superior

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

Sin embargo, la elección del enfoque anidado en lugar del enfoque de separación total debe basarse en la estructura de la organización. Si el cumplimiento de pedidos, la gestión de inventario y la recepción de mercancías son gestionados por equipos diferentes, probablemente merezcan su condición de microservicios de nivel superior . Si, por otro lado, todos ellos son gestionados por un solo equipo, entonces el modelo anidado tiene más sentido. Esto se debe a la interacción de las estructuras organizativas y la arquitectura del software, que analizaremos hacia el final del libro en [el Capítulo 10](#).

Otra razón para preferir el enfoque anidado podría ser dividir la arquitectura para simplificar las pruebas. Por ejemplo, al probar servicios que consumen el almacén, no tengo que crear un stub de cada servicio dentro del contexto del almacén, solo la API más general.

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

## La comunicación en términos de conceptos empresariales

Los cambios que implementamos en nuestro sistema suelen estar relacionados con los cambios que la empresa quiere hacer en el comportamiento del sistema. Estamos modificando la funcionalidad (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 hacer se aíslen en un único límite de microservicio. Esto reduce la cantidad de lugares en los que necesitamos hacer 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 empresariales. El modelado de su software según su dominio empresarial no debe detenerse en la idea de contextos delimitados. Los mismos términos e ideas que se comparten entre las partes de su organización deben reflejarse en sus interfaces. Puede ser útil pensar en formularios que se envían entre estos microservicios, de la misma manera que se envían formularios dentro de una organización.

# El límite técnico

Puede resultar útil analizar qué puede salir mal cuando los servicios se modelan de forma incorrecta. Hace un tiempo, algunos colegas y yo estábamos trabajando con un cliente en California, ayudando a la empresa a adoptar prácticas de código más limpias y avanzar más hacia las pruebas automatizadas.

Habíamos empezado con algunas de las tareas más sencillas, como la descomposición del servicio, cuando nos dimos cuenta de algo mucho más preocupante. No puedo entrar en demasiados detalles sobre lo que hacía la aplicación, pero era una aplicación pública con una gran base de clientes global.

El equipo y el sistema habían crecido. Originalmente, el sistema, que era la visión de una sola persona, había incorporado cada vez más funciones y cada vez más usuarios. Finalmente, la organización decidió aumentar la capacidad del equipo y contratar a un nuevo grupo de desarrolladores con base en Brasil para que se hiciera cargo de parte del trabajo. El sistema se dividió y la mitad frontal de la aplicación quedó esencialmente sin estado, implementando el sitio web público, como se muestra en [la Figura 3-4](#). La mitad posterior del sistema era simplemente una interfaz de llamada a procedimiento remoto (RPC) sobre un almacén de datos.

Básicamente, imagine que hubiera tomado una capa de repositorio en su base de código y la hubiera convertido en un servicio independiente.

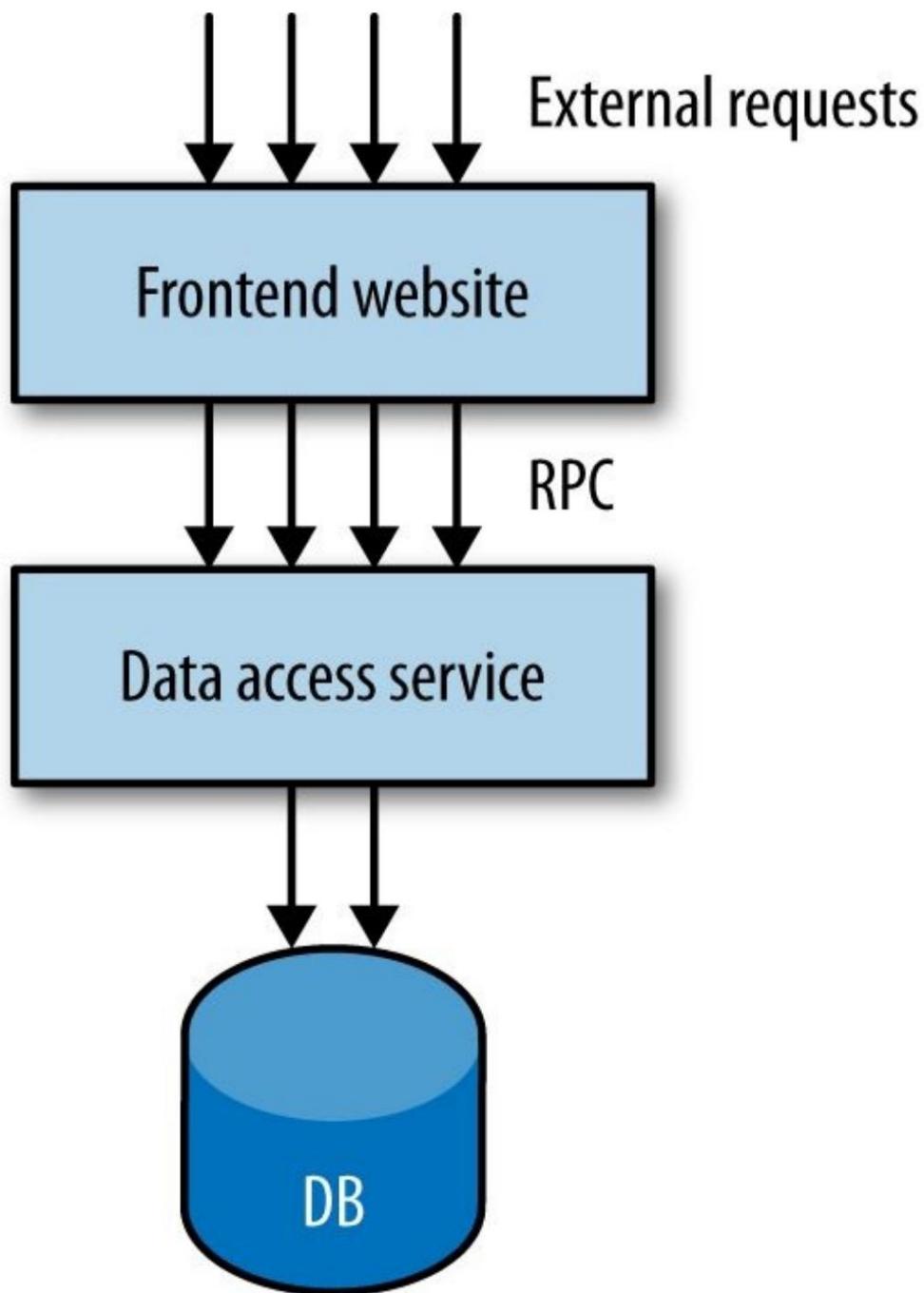


Figura 3-4. Límite de servicio dividido a lo largo de una costura técnica

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

A primera vista, la idea de dividir el sistema previamente monolítico según líneas geográficas y organizativas tiene mucho sentido, como explicaremos en [el Capítulo 10](#).

Sin embargo, aquí, en lugar de tomar una porción vertical y centrada en el negocio, el equipo eligió lo que anteriormente era una API en proceso e hizo una porción 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 lograr ciertos objetivos de desempeño, por ejemplo. Sin embargo, debería ser su impulsor secundario

para encontrar estas costuras, no la principal.

## Resumen

En este capítulo, aprendiste un poco sobre qué hace que un servicio sea bueno y cómo encontrar puntos de unión en nuestro espacio de problemas que nos brinden los beneficios dobles de un acoplamiento flexible y una alta cohesión. Los contextos delimitados son una herramienta vital para ayudarnos a encontrar estos puntos de unión 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 obtuvimos una pista sobre cómo podemos subdividir aún más nuestros microservicios, algo que exploraremos en 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 nos resultan muy útiles para encontrar límites razonables para nuestros servicios, y aquí apenas he comenzado a dar una idea general. 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 principalmente de alto nivel, en el próximo debemos profundizar más en aspectos técnicos. Existen muchos obstáculos asociados 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 caos gigantesco y enredado.

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

# Capítulo 4. Integración

---

En mi opinión, la integración correcta es el aspecto más importante de la tecnología asociada a los microservicios. Si lo haces bien, tus microservicios conservarán su autonomía, lo que te permitirá cambiarlos y lanzarlos independientemente del conjunto. Si lo haces mal, te espera el desastre. Con suerte, una vez que hayas leído este capítulo, aprenderás a evitar algunos de los mayores obstáculos que han plagado otros intentos de SOA y que aún podrían aguardarte en tu viaje hacia los microservicios.

## En busca de la tecnología de integración ideal

Existe una variedad desconcertante de opciones para que un microservicio se comunique con otro. Pero, ¿cuál es la correcta: SOAP? ¿XML-RPC? ¿REST? ¿Búferes de protocolo? Nos adentraremos en ellas en un momento, pero antes de hacerlo, pensemos en lo que queremos de la tecnología que elijamos.

## Evite cambios disruptivos

De vez en cuando, podemos realizar un cambio que requiera que nuestros consumidores también cambien. Hablaremos sobre cómo manejar esto más adelante, pero queremos elegir tecnología que garantice que esto suceda con la menor frecuencia posible. Por ejemplo, si un microservicio agrega nuevos campos a un fragmento de datos 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. Todo el tiempo aparecen nuevas herramientas, marcos y lenguajes que implementan nuevas ideas que pueden ayudarnos a trabajar más rápido y de manera más eficaz. En este momento, es posible que tenga una tienda .NET.

Pero ¿qué sucederá dentro de un año o de cinco años? ¿Qué sucederá si desea experimentar con una pila de tecnología alternativa que podría aumentar su productividad?

Soy un gran partidario de mantener abiertas mis opciones, por eso soy un gran admirador de los microservicios. Por eso también creo que es muy importante asegurarse de que las API que se utilizan para la comunicación entre microservicios sean independientes de la tecnología. Esto significa evitar la tecnología de integración que dicta qué pilas de tecnologías podemos utilizar para implementar nuestros microservicios.

## Simplifique su servicio para los consumidores

Queremos que a los consumidores les resulte fácil utilizar nuestro servicio. Tener un microservicio bien diseñado no sirve de mucho si el coste de su uso como consumidor es altísimo. Por eso, pensemos en qué hace que a los consumidores les resulte fácil utilizar nuestro maravilloso nuevo servicio.

Lo ideal sería que nuestros clientes tuvieran total libertad para elegir la tecnología que desean, pero, por otro lado, ofrecer una biblioteca de clientes puede facilitar la adopción. Sin embargo, a menudo, dichas bibliotecas son incompatibles con otras cosas que queremos lograr. Por ejemplo, podríamos utilizar bibliotecas de clientes para facilitarles las cosas a los consumidores, pero esto puede tener como consecuencia un mayor acoplamiento.

## Ocultar detalles de implementación interna

No queremos que nuestros consumidores estén ligados a nuestra implementación interna. Esto conduce a un mayor acoplamiento. Esto significa que si queremos cambiar algo dentro de nuestro microservicio, podemos romper 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 queramos hacer un cambio por miedo a tener que actualizar nuestros consumidores, lo que puede conducir a una mayor deuda técnica dentro del servicio. Por lo tanto, se debe evitar cualquier tecnología que nos presione para exponer los detalles de la representación interna.

## Interactuar 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 que existen e intentemos determinar cuál funciona mejor para nosotros. Para ayudarnos a pensar en esto, elijamos un ejemplo real de MusicCorp.

A primera vista, la creación de un cliente podría considerarse un conjunto simple de operaciones CRUD, pero para la mayoría de los sistemas es más compleja que eso. Para inscribir a un nuevo cliente, es posible que se deban iniciar procesos adicionales, como configurar pagos financieros o enviar correos electrónicos de bienvenida. Y cuando cambiamos o eliminamos un cliente, también pueden activarse otros procesos comerciales.

Teniendo esto en mente, deberíamos considerar algunas formas diferentes en las que podríamos trabajar con los clientes en nuestro sistema MusicCorp.

## La base de datos compartida

La forma más común de integración que yo o cualquiera de mis colegas vemos en la industria es la integración de bases de datos (BD). En este mundo, si otros servicios quieren información de un servicio, recurren a la base de datos. Y si quieren cambiarla, recurren a la base de datos. Esto es realmente simple cuando lo piensas por primera vez y es probablemente la forma más rápida de integración con la que comenzar, lo que probablemente explica su popularidad.

**La Figura 4-1** muestra nuestra interfaz de usuario de registro, que crea clientes realizando operaciones SQL directamente en la base de datos. También muestra nuestra aplicación de centro de llamadas que visualiza 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.

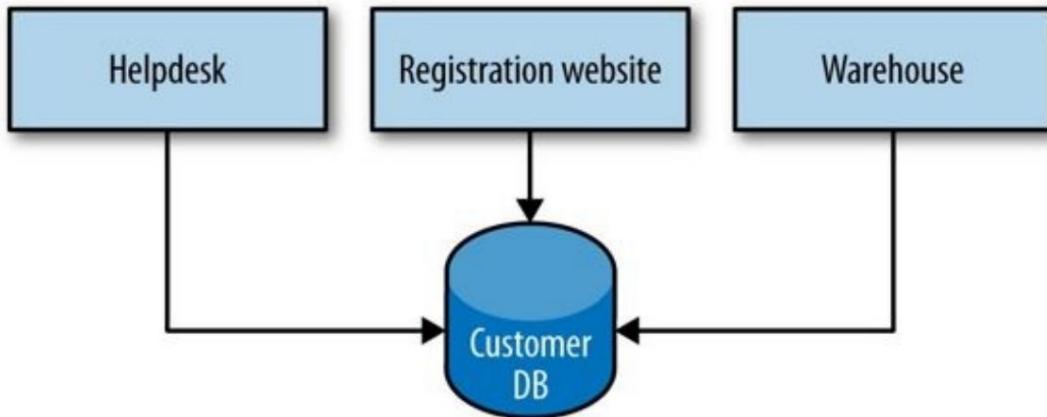


Figura 4-1. Uso de la integración de bases de datos para acceder y modificar la información del cliente

En primer lugar, permitimos que partes externas vean y se vinculen con los detalles de implementación interna. Las estructuras de datos que almaceno en la base de datos son de uso común para todos; se comparten en su totalidad con todas las demás partes que tienen 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 romper 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 técnica administra a los clientes y esto requiere un cambio en la base de datos, tengo que ser extremadamente cuidadoso para no romper partes del esquema utilizado por otros servicios. Esta situación normalmente da como resultado la necesidad de una gran cantidad de pruebas de regresión.

En segundo lugar, mis consumidores están vinculados a una elección tecnológica específica. Tal vez ahora mismo tenga sentido almacenar a los clientes en una base de datos relacional, de modo que mis consumidores utilicen un controlador adecuado (posiblemente específico de la base de datos) para comunicarse con ella. ¿Qué sucede si con el tiempo nos damos cuenta de que sería mejor almacenar los datos en una base de datos no relacional? ¿Puede tomar esa decisión? Por lo tanto, los consumidores están íntimamente vinculados a la implementación del servicio al cliente. Como comentamos antes, realmente queremos asegurarnos de que los detalles de la implementación estén ocultos a 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 débil.

Por último, pensemos en el comportamiento por un momento. Habrá una lógica asociada con la forma en que se modifica un cliente. ¿Dónde está esa lógica? Si los consumidores manipulan directamente la base de datos, entonces deben poseer la lógica asociada. La lógica para realizar el mismo tipo de manipulación a un cliente ahora puede estar distribuida 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 de los principios básicos detrás de los buenos microservicios? Cohesión fuerte y acoplamiento flexible: con la integración de bases de datos, perdemos ambas cosas. La integración de bases de datos facilita que los servicios comparten datos, pero no hace nada con respecto a compartir comportamientos. Nuestra representación interna está expuesta a través de la red a nuestros consumidores, y puede ser muy difícil evitar realizar cambios disruptivos, lo que inevitablemente conduce al temor a cualquier cambio. Evítalo a (casi) toda costa.

Durante el resto del capítulo, exploraremos diferentes estilos de integración que involucran servicios colaboradores, que ocultan sus propias representaciones internas.

# Sincrónico versus asincrónico

Antes de comenzar a profundizar en los detalles de las diferentes opciones tecnológicas, deberíamos 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 lleva inevitablemente a ciertos detalles de implementación.

Con la comunicación sincrónica, se realiza una llamada a un servidor remoto, que se bloquea hasta que se completa la operación. Con la comunicación asincrónica, el autor de la llamada 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 sencilla de entender. Sabemos cuándo las cosas se han completado con éxito o no. La comunicación asincrónica puede ser muy útil para trabajos de larga duración, 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 una latencia baja, en los que 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 suponer 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 tiene un gran retraso. Por otro lado, la tecnología para gestionar la comunicación asincrónica puede ser un poco más compleja, como analizaremos en breve.

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

Con una colaboración basada en eventos, invertimos las cosas. En lugar de que un cliente inicie solicitudes para que se hagan cosas, dice que sucedió algo y espera que las otras partes sepan qué hacer. Nunca le decimos a nadie más qué hacer. Los sistemas basados en eventos son asincrónicos por naturaleza. La inteligencia se distribuye de manera más uniforme, es decir, la lógica empresarial no está centralizada en cerebros centrales, sino que se distribuye de manera más uniforme entre los distintos 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 a él, lo que también significa que puede agregar nuevos suscriptores a estos eventos sin que el cliente tenga que saberlo.

¿Existen otros factores que podrían impulsarnos a elegir un estilo en lugar de otro? Un factor importante a tener en cuenta es la idoneidad de estos estilos para resolver un problema a menudo complejo: ¿cómo gestionamos procesos que abarcan los límites del servicio y pueden ser de larga duración?

## Orquestación versus coreografía

A medida que empezamos a modelar una lógica cada vez más compleja, tenemos que lidiar con el problema de gestionar procesos empresariales que se extienden más allá de los límites de los servicios individuales. Y con los microservicios, llegaremos a este límite antes de lo habitual. Tomemos un ejemplo de MusicCorp y observemos 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 correo electrónico de bienvenida al cliente.

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

A la hora de implementar este flujo, existen dos estilos de arquitectura que podemos seguir. Con la orquestación, nos apoyamos en un cerebro central que guía y dirige el proceso, de forma muy similar al director de una orquesta. Con la coreografía, informamos a cada parte del sistema de su función y dejamos que se ocupe de los detalles, como si fueran bailarines que encuentran su camino y reaccionan ante los demás en un ballet.

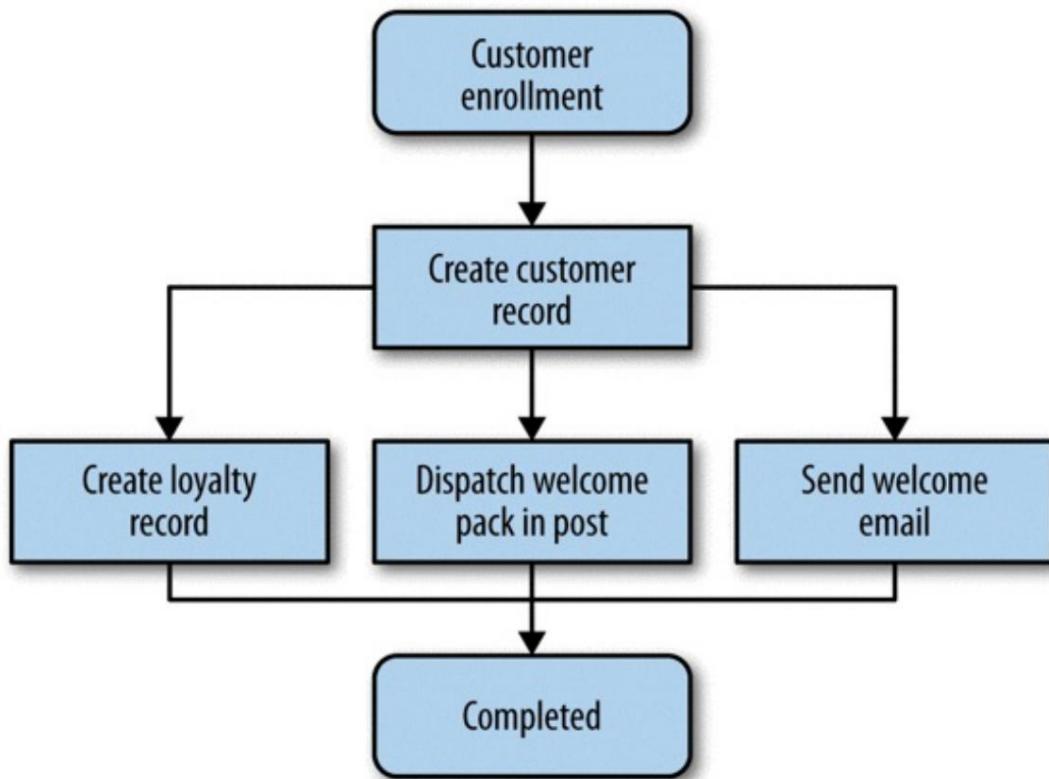


Figura 4-2. Proceso de creación de un nuevo cliente

Pensemos en cómo se vería una solución de orquestación para este flujo. En este caso, probablemente lo más sencillo sería que nuestro servicio de atención al cliente actuara como cerebro central. En el momento de 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

El propio cliente puede entonces rastrear dónde se encuentra un cliente en este proceso. Puede verificar si la cuenta del cliente ha sido configurada, o si se envió el correo electrónico, o si se entregó el correo postal. Podemos tomar el diagrama de flujo de [la Figura 4-2](#) y modelarlo directamente en código. Incluso podríamos usar herramientas que implementen esto por nosotros, tal vez usando un motor de reglas apropiado. Existen herramientas comerciales para este mismo propósito en forma de software de modelado de procesos comerciales. Suponiendo que usemos una solicitud/respuesta sincrónica, incluso podríamos saber si cada etapa ha funcionado.

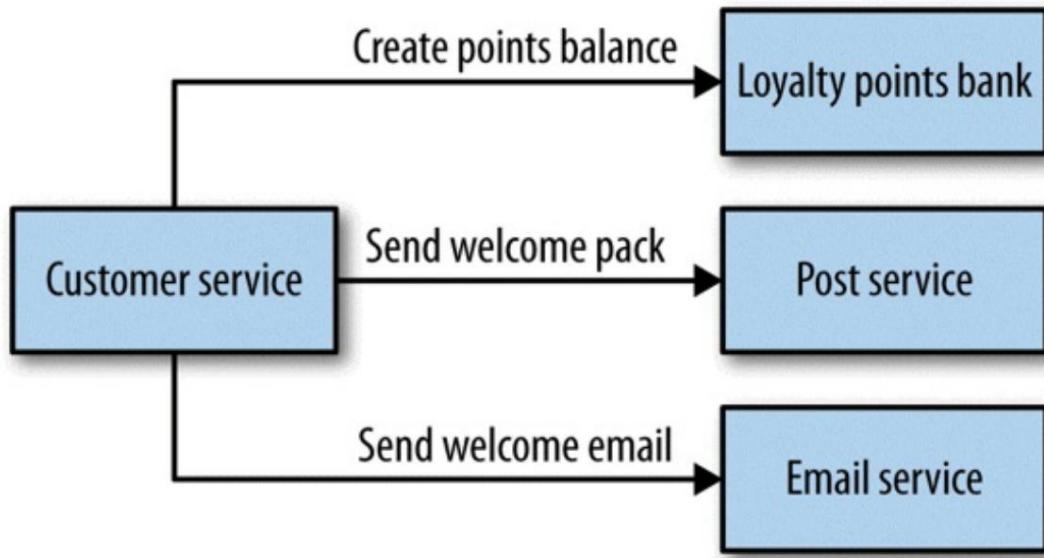


Figura 4-3. Gestión de la creación de clientes mediante orquestación

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

Con un enfoque coreografiado, podríamos simplemente hacer que el servicio de atención al cliente emita un evento de manera asincrónica, diciendo "Cliente creado". El servicio de correo electrónico, el servicio postal y el banco de puntos de fidelidad se suscriben a estos eventos y reaccionan en consecuencia, como en [la Figura 4-4](#). Este enfoque es significativamente más desacoplado. Si algún otro servicio necesitara 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 empresarial que vemos en [la Figura 4-2](#) ahora solo se refleja implícitamente en nuestro sistema.

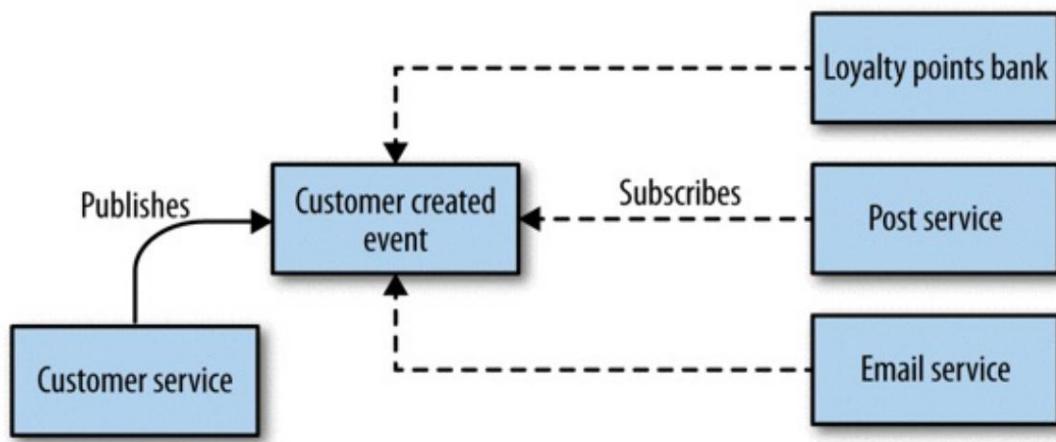


Figura 4-4. Gestión de la creación de clientes mediante coreografía

Esto significa que se necesita trabajo adicional para garantizar que se pueda supervisar y hacer un seguimiento de que se han producido 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 crear un sistema de supervisión que coincida explícitamente con la vista del proceso empresarial en [la Figura 4-2](#), pero que luego haga un seguimiento de lo que hace cada uno de los servicios como entidades independientes, lo que le permite ver excepciones extrañas mapeadas en el 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 está comportando el sistema.

En general, he descubierto que los sistemas que tienden más hacia el enfoque coreografiado están acoplados de forma más flexible y son más susceptibles de cambio. Sin embargo, es necesario realizar un trabajo adicional para monitorear y rastrear los procesos a través de los límites del sistema. He descubierto que la mayoría de las implementaciones muy orquestadas son extremadamente frágiles y tienen un mayor costo de cambio. Teniendo esto en cuenta, prefiero firmemente apuntar a un sistema coreografiado, donde cada servicio sea lo suficientemente inteligente como para comprender su papel en todo el proceso.

Hay varios factores que analizar aquí. Las llamadas sincrónicas son más simples y nos permiten saber si las cosas funcionaron de inmediato. Si nos gusta la semántica de solicitud/respuesta pero estamos lidiando con procesos de larga duración, podríamos simplemente iniciar solicitudes asincrónicas y esperar las devoluciones de llamadas. Por otro lado, la colaboración de eventos asincrónicos nos ayuda a adoptar un enfoque coreografiado, que puede generar servicios significativamente más desacoplados, algo que queremos lograr para asegurar que nuestros servicios se puedan lanzar de forma independiente.

Por supuesto, tenemos la libertad de combinar y mezclar. Algunas tecnologías encajarán de forma más natural en un estilo u otro. Sin embargo, debemos tener en cuenta algunos de los diferentes detalles de implementación técnica que nos ayudarán a tomar la decisión correcta.

Para comenzar, veamos dos tecnologías que se adaptan bien cuando consideramos la solicitud/respuesta: llamada a procedimiento remoto (RPC) y transferencia de estado representacional (REST).

## Llamadas a procedimientos remotos

La llamada a procedimiento remoto se refiere a la técnica de realizar una llamada local y hacer que se ejecute en un servicio remoto en algún lugar. Existen varios tipos diferentes de tecnología RPC. Algunas de estas tecnologías se basan 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 que exponga una interfaz SOAP y un cliente .NET generado a partir de la definición de la interfaz en el lenguaje de definición de servicio web (WSDL). Otras tecnologías, como Java RMI, requieren un acoplamiento más estrecho entre el cliente y el servidor, lo que requiere que ambos utilicen la misma tecnología subyacente pero eviten la necesidad de una definición de interfaz compartida. Sin embargo, todas estas tecnologías tienen la misma característica principal: hacen que una llamada local parezca una llamada remota.

Muchas de estas tecnologías son de naturaleza binaria, como Java RMI, Thrift o los buffers de protocolo, mientras que SOAP utiliza 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 utilizar diferentes tipos de protocolos de red, que a su vez pueden proporcionar características adicionales. Por ejemplo, TCP ofrece garantías sobre la entrega, mientras que UDP no lo hace, pero tiene una sobrecarga mucho menor. Esto puede permitirle utilizar diferentes tecnologías de red para diferentes casos de uso.

Las implementaciones de RPC que permiten generar stubs de cliente y servidor ayudan a empezar muy, muy rápido. Puedo enviar contenido a través de un límite de red en un abrir y cerrar de ojos. Este suele ser uno de los principales puntos fuertes de RPC: su facilidad de uso. El hecho de que pueda simplemente realizar una llamada de método normal y, en teoría, ignorar el resto es una gran ventaja.

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

## Acoplamiento de tecnología

Algunos mecanismos de RPC, como Java RMI, están fuertemente ligados a una plataforma específica, lo que puede limitar qué tecnología se puede utilizar en el cliente y el servidor. Thrift y los buffers de protocolo tienen una cantidad impresionante de soporte para lenguajes alternativos, lo que puede reducir esta desventaja en cierta medida, pero tenga en cuenta que a veces la tecnología RPC viene con restricciones en cuanto a interoperabilidad.

En cierto modo, esta combinación de tecnologías puede ser una forma de exponer detalles técnicos internos de implementación. 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 de 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 hacer una gran cantidad de llamadas locales en proceso sin preocuparme demasiado por el rendimiento. Sin embargo, con RPC, el costo de ordenar y desordenar cargas útiles puede ser significativo, sin mencionar el tiempo que lleva enviar cosas a través de la red. Esto significa que debe pensar de manera diferente sobre el diseño de API para interfaces remotas en comparación con las interfaces locales. Simplemente tomar una API local e intentar convertirla en un límite de servicio sin pensarlo más probablemente lo meterá en problemas. En algunos de los peores ejemplos, los desarrolladores pueden estar usando llamadas remotas sin saberlo si la abstracción es

Es necesario pensar en la red en sí. La primera de las falacias de la computación distribuida es que “[la red es confiable](#)”. Las redes no son confiables. Pueden fallar y lo harán, incluso si su cliente y el servidor con el que está hablando están bien. Pueden fallar rápido, pueden fallar lentamente e incluso pueden deformar sus paquetes. Debe asumir que sus redes están plagadas de entidades malévolas listas para desatar su ira a su antojo.

Por lo tanto, los modos de falla que puede esperar son diferentes. Una falla puede ser causada por un error del servidor remoto o por una mala decisión suya. ¿Puede notar la diferencia y, de ser así, puede hacer algo al respecto? ¿Y qué hace cuando el servidor remoto comienza a responder lentamente? Trataremos este tema cuando hablemos de resiliencia en [el Capítulo 11](#).

## Fragilidad

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

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

---

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

interfaz pública CustomerRemote extiende Remote { público
    Customer findCustomer(String id) lanza RemoteException;

    público Cliente createCustomer(String nombre, String apellido, String emailAddress) lanza RemoteException;
}

}
```

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

```
...
público Cliente createCustomer(String emailAddress) lanza RemoteException;
...
```

El problema es que ahora también necesitamos regenerar los stubs del cliente. Los clientes que quieren 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, existe otro tipo de fragilidad. Veamos cómo se ve nuestro objeto `Cliente`:

```
clase pública Cliente implementa Serializable {
    cadena privada nombre; cadena
    privada apellido; cadena
    privada dirección de correo
    electrónico; cadena privada edad;
}
```

Ahora bien, ¿qué sucede si resulta que, aunque exponemos el campo de edad en nuestros objetos de `Cliente`, ninguno de nuestros consumidores lo utiliza nunca? 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, entonces, aunque nunca hayan utilizado el campo, el código asociado con la deserialización del objeto `Cliente` en el lado del consumidor dejará de funcionar. Para implementar este cambio, tendría que implementar un nuevo servidor y clientes al mismo tiempo. Esta es una

El principal desafío de cualquier mecanismo de RPC que promueva el uso de generación de stubs binarios es que no se pueden separar las implementaciones de cliente y servidor. Si utiliza esta tecnología, es posible que en el futuro tenga que implementar versiones en secuencia.

Se producen problemas similares si quiero reestructurar el objeto Cliente incluso si no eliminé campos; por ejemplo, si quiero encapsular firstName y Surnames en un nuevo tipo de nombre para que sea más fácil de administrar. Por supuesto, podría solucionar 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 elegí.  
desear.

En la práctica, los objetos utilizados como parte de la serialización binaria a través de la red pueden considerarse como tipos de solo expansión . Esta fragilidad hace que los tipos queden expuestos a través de la red y se conviertan en una masa de campos, algunos de los cuales ya no se utilizan pero no se pueden eliminar de forma segura.

## ¿Es terrible el RPC?

A pesar de sus defectos, no me atrevería a decir que RPC es terrible. Algunas de las implementaciones comunes que he encontrado pueden dar lugar a los tipos de problemas que he descrito aquí. Debido a los desafíos que supone utilizar RMI, sin duda evitaría esa tecnología. Muchas operaciones encajan perfectamente en el modelo basado en RPC, y mecanismos más modernos como los buffers de protocolo o Thrift mitigan algunos de los pecados del pasado al evitar la necesidad de versiones en bloque del código del cliente y del servidor.

Tenga en cuenta algunos de los posibles inconvenientes asociados con RPC si va a elegir este modelo. No abstraiga sus llamadas remotas hasta el punto en que la red esté completamente oculta y asegúrese de poder desarrollar la interfaz del servidor sin tener que insistir en actualizaciones continuas para los clientes. Por ejemplo, es importante encontrar el equilibrio adecuado para su código de cliente. Asegúrese de que sus clientes no sean ajenos al hecho de que se va a realizar una llamada de red. Las bibliotecas de cliente se utilizan a menudo en el contexto de RPC y, si no están estructuradas correctamente, pueden ser problemáticas. Hablaremos más sobre ellas en breve.

En comparación con la integración de bases de datos, RPC es sin duda una mejora si tenemos en cuenta las opciones de colaboración entre solicitudes y respuestas. Pero hay otra opción que se debe tener en cuenta.

## DESCANSAR

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.

Lo más importante es el concepto de recursos. Se puede pensar en un recurso como algo que el propio servicio conoce, como un cliente. El servidor crea diferentes representaciones de este cliente a petición. La forma en que se muestra un recurso externamente está completamente desvinculada de la forma en que 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 cambiarla, y el servidor puede o no cumplirlas.

Existen muchos estilos diferentes de REST y aquí sólo los mencionaré brevemente. Recomiendo encarecidamente que eche un vistazo al [modelo de madurez de Richardson](#). Donde se comparan los diferentes estilos de REST.

REST en sí no habla realmente de protocolos subyacentes, aunque se utiliza más comúnmente sobre HTTP. He visto implementaciones de REST utilizando protocolos muy diferentes antes, como serial o USB, aunque esto puede requerir mucho trabajo. Algunas de las características que HTTP nos da como parte de la especificación, como los verbos, hacen que la implementación de REST sobre HTTP sea más fácil, mientras que con otros protocolos tendrás que manejar estas características tú mismo.

## REST 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 en cuanto a cómo deberían funcionar con los recursos. El estilo arquitectónico REST en realidad nos dice que los métodos deberían comportarse de la misma manera en todos los recursos, y resulta que la especificación HTTP define una serie de métodos que podemos utilizar.

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

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

HTTP también trae consigo un gran ecosistema de herramientas y tecnología de apoyo. Podemos utilizar servidores proxy de almacenamiento en caché HTTP como Varnish y平衡adores de carga como mod\_proxy, y muchas herramientas de monitorización ya tienen un amplio soporte para HTTP de fábrica. Estos componentes básicos nos permiten gestionar grandes volúmenes de tráfico HTTP y enrutarlos de forma inteligente, de una forma bastante transparente. También podemos utilizar 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 proporciona muchas herramientas para facilitar el proceso de seguridad, y exploraremos ese tema más en profundidad en el Capítulo 10. Dicho esto, para obtener estos beneficios, hay que utilizar HTTP correctamente. Si se utiliza mal, puede resultar tan inseguro y difícil de escalar como cualquier otra tecnología. Sin embargo, si se utiliza correctamente, se obtendrán muchas ventajas.

Tenga en cuenta que también se puede utilizar HTTP para implementar RPC. SOAP, por ejemplo, se enruta a través de HTTP, pero lamentablemente utiliza muy poco de la especificación. Se ignoran los verbos, al igual que cosas simples como los códigos de error HTTP. Con demasiada frecuencia, parece que se ignoran los estándares y la tecnología existentes y bien entendidos 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 cliente y servidor es el concepto de hipermedia como motor del estado de la aplicación (que suele abreviarse como HATEOAS, y vaya si necesitaba una abreviatura). Se trata de un concepto bastante interesante y con un vocabulario bastante denso, así que vamos a desglosarlo un poco.

El hipermedio es un concepto en el que un fragmento de contenido contiene enlaces a otros fragmentos de contenido en una variedad de formatos (por ejemplo, texto, imágenes, sonidos). Esto debería resultarte bastante familiar, ya que es lo que hace la página web promedio: sigues enlaces, que son una forma de controles hipermedia, para ver contenido relacionado. La idea detrás de HATEOAS es que los clientes deben realizar interacciones (que potencialmente conducen 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 al saber qué URI utilizar; en cambio, el cliente busca y navega por los enlaces para encontrar lo que necesita.

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

Pensemos en el sitio de compras 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 seguir viendo un carrito de compras, saber qué es e interactuar con él. Entendemos 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. Es por eso que las páginas web pueden cambiar de forma incremental con el tiempo. Mientras se sigan cumpliendo estos contratos implícitos entre el cliente y el sitio web, los cambios no tienen por qué ser cambios radicales.

Con los controles hipermedia, intentamos 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 determinado en el Ejemplo 4-2. Junto con información sobre el álbum, vemos una serie de controles hipermedia.

---

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

```
<álbum>
  <nombre>Donar sangre</nombre>
  <link rel="/artist" href="/artist/theBrakes" /> <description> Impresionante, breve, ①
  brutal, divertido y
  ruidoso. ¡Debes comprarlo! </description> <link rel="/instantpurchase" href="/
  instantPurchase/1234" /> </álbum>
```

①

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

②

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

En este documento, tenemos dos controles hipermedia. El cliente que lee un documento de este tipo debe saber que un control con una relación de artista es el lugar al que 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 se encontrarán los artículos que se comprarán.

Como cliente, no necesito saber a qué esquema de URL acceder para comprar el álbum, solo necesito acceder al recurso, encontrar el control de compra y navegar hasta él. 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 brinda una gran cantidad de desacoplamiento entre el cliente y el servidor.

Aquí nos abstraemos en gran medida de los detalles subyacentes. 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 simple enlace a un control de JavaScript más complejo. También tenemos la libertad de agregar nuevos controles al documento, tal vez representando nuevas transiciones de estado que podemos realizar en el recurso en cuestión. Terminaríamos rompiendo nuestros consumidores solo si cambiamos fundamentalmente la semántica de uno de los controles para que se comporte de manera muy diferente, o si eliminamos un control por completo.

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

Una de las desventajas es que esta navegación de controles puede ser bastante confusa, ya que el cliente necesita seguir enlaces para encontrar la operación que desea realizar. En última instancia, se trata de una cuestión de compromiso. Le sugeriría que comience haciendo que sus clientes naveguen por estos controles primero y luego optimicen más tarde si es necesario. Recuerde que tenemos una gran cantidad de ayuda lista para usar mediante el uso de HTTP, que analizamos anteriormente. Los males de la optimización prematura ya se han documentado bien antes, por lo que no es necesario extenderme en ellos aquí. Tenga en cuenta también que muchos de estos enfoques se desarrollaron para crear sistemas de hipertexto distribuidos, ¡y no todos encajan! A veces, se encontrará simplemente queriendo el buen y antiguo RPC.

Personalmente, soy partidario de utilizar enlaces para permitir que los consumidores naveguen por los puntos finales de la API. Los beneficios del descubrimiento progresivo de la API y la reducción del acoplamiento pueden ser significativos. Sin embargo, está claro que no todo el mundo está convencido, ya que no creo que se utilice tanto como me gustaría. Creo que gran parte de esto se debe a que se requiere un trabajo inicial, pero las recompensas suelen llegar 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 los recursos, y REST sobre HTTP nos permite usar una variedad de formatos. Los ejemplos que he dado hasta ahora usaban XML, pero en esta etapa, JSON es un tipo de contenido mucho más popular para los servicios que funcionan sobre HTTP.

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

Sin embargo, JSON tiene algunas desventajas. XML define el control de enlace que usamos antes como un control de hipermedia. El estándar JSON no define nada similar, por lo que se usan estilos internos con frecuencia para incluir este concepto. El lenguaje [de aplicación de hipertexto \(HAL\)](#) intenta solucionar esto definiendo algunos estándares comunes para la creación de hipervínculos para JSON (y XML también, aunque se podría decir 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 de hipermedia, lo que puede hacer que la tarea de crear un cliente sea mucho más sencilla.

Por supuesto, no estamos limitados a estos dos formatos. Podemos enviar prácticamente cualquier cosa a través de HTTP si queremos, incluso binarios. Veo cada vez más gente que utiliza HTML como formato en lugar de XML. Para algunas interfaces, el HTML puede cumplir una doble función como interfaz de usuario y API, aunque hay que evitar algunos inconvenientes, ya que las interacciones entre un ser humano y una computadora son bastante diferentes. Pero sin duda es una idea atractiva. Después de todo, hay muchos analizadores HTML.

Personalmente, sin embargo, sigo siendo un fanático de XML. Algunas de las herramientas que lo admiten son mejores. Por ejemplo, si quiero extraer solo ciertas partes de la carga útil (una técnica que analizaremos más en “[Control de versiones](#)”), puedo usar XPATH, que es un estándar bien conocido con muchas herramientas que lo admiten, o incluso selectores CSS, que muchos encuentran aún más fáciles. Con JSON, tengo JSONPATH, pero no es ampliamente compatible. Me parece extraño que la gente elija JSON porque es agradable y liviano, y luego intente introducir conceptos en él como controles de hipermedia que ya existen en XML. Sin embargo, acepto que probablemente soy una minoría en este aspecto y que JSON es el formato elegido por la mayoría de las personas.

## Cuidado con la comodidad excesiva

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 sacrifican demasiado en términos de beneficios a corto plazo por problemas a largo plazo; al intentar que comiences a trabajar rápido, pueden fomentar algunos comportamientos negativos. Por ejemplo, algunos marcos realmente hacen que sea muy fácil simplemente tomar representaciones de objetos de bases de datos, deserializarlas en objetos en proceso y luego exponerlas directamente de manera externa. Recuerdo que en una conferencia vi una demostración de esto usando Spring Boot y se citó 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.

Aquí hay un problema más general en juego. La forma en que 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 utilizó de manera efectiva fue retrasar la implementación de la persistencia adecuada para el microservicio hasta que la interfaz se hubiera estabilizado lo suficiente. Durante un período provisional, las entidades simplemente se conservaban 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 determinara las decisiones de diseño e implementación. La justificación presentada, que se confirmó en los resultados, fue que es demasiado fácil que la forma en que almacenamos las entidades de dominio en un almacén de respaldo influya abiertamente en los modelos que enviamos por cable a los colaboradores. Una de las desventajas de este enfoque es que estamos aplazando el trabajo necesario para conectar nuestro almacén de datos. Sin embargo, creo que para los nuevos límites del servicio, esta es una compensación aceptable.

## Desventajas de REST sobre HTTP

En términos de facilidad de uso, no es fácil generar un código auxiliar de cliente para el protocolo de aplicación REST sobre HTTP como se puede hacer con RPC. Por supuesto, el hecho de que se utilice HTTP significa que se pueden aprovechar todas las excelentes bibliotecas de cliente HTTP que existen, pero si se desea implementar y utilizar controles de hipermedia como cliente, se está prácticamente solo. Personalmente, creo que las bibliotecas de cliente podrían hacerlo mucho mejor de lo que lo hacen, y sin duda ahora son mejores que en el pasado, pero he visto que esta aparente mayor complejidad hace que la gente vuelva a introducir RPC sobre HTTP de contrabando o a crear bibliotecas de cliente compartidas. El código compartido entre el cliente y el servidor puede ser muy peligroso, como analizaremos en “[DRY y los peligros de la reutilización de código en un mundo de microservicios](#)”.

Un aspecto menor es que algunos frameworks de servidores web en realidad no admiten todos los verbos HTTP. Eso significa que puede resultarte fácil crear un controlador para solicitudes GET o POST, pero es posible que tengas que hacer algunos trámites para que funcionen las solicitudes PUT o DELETE. Los frameworks REST adecuados como Jersey no tienen este problema y normalmente puedes solucionarlo, pero si estás limitado a determinadas opciones de framework, esto puede limitar el estilo de REST que puedes usar.

El rendimiento también puede ser un problema. Las cargas útiles de REST sobre HTTP pueden ser más compactas que las de SOAP porque admiten formatos alternativos como JSON o incluso binarios, pero no serán ni de lejos un protocolo binario tan sencillo como Thrift. La sobrecarga de HTTP para cada solicitud también puede ser un problema para los requisitos de baja latencia.

El protocolo HTTP, si bien puede ser adecuado para 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 utilizando mucho HTTP, y mucho menos algo que tenga que ver con REST.

Para las comunicaciones entre servidores, 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 de trabajo de RPC se ejecutarán sin problemas sobre protocolos de red distintos de TCP.

El consumo de las propias cargas útiles requiere más trabajo que el que ofrecen 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 (lo discutiremos en breve), pero desde el punto de vista de su puesta en funcionamiento, pueden ser muy atractivos.

A pesar de estas desventajas, REST sobre HTTP es una opción predeterminada sensata para las interacciones entre servicios. Si quieras saber más, te recomiendo [REST en la práctica](#) (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 agentes de mensajes como RabbitMQ intentan solucionar ambos problemas. Los productores utilizan una API para publicar un evento en el agente. El agente gestiona las suscripciones, lo que permite informar a los consumidores cuando llega un evento. Estos agentes pueden incluso gestionar el estado de los consumidores, por ejemplo, ayudándolos 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. También pueden requerirse 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 acopladas de manera flexible. En general, soy un fanático.

Sin embargo, tenga cuidado con el mundo del middleware, del cual el agente 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 grandes cantidades de software con ellas, lo que puede llevar a que se introduzcan cada vez más elementos inteligentes en el middleware, como lo demuestran elementos como Enterprise Service Bus. Asegúrese de saber lo que está obteniendo: mantenga su middleware en silencio y mantenga la inteligencia en los puntos finales.

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

De esta manera, nuestro servicio de atención al cliente podría simplemente publicar un evento en un feed de este tipo cuando se produzcan cambios. 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 HTTP gestiona muy bien la escalabilidad. Sin embargo, HTTP no es bueno en baja latencia (donde algunos agentes de mensajes sobresalen), y aún tenemos que lidiar con el hecho de que los consumidores necesitan realizar un seguimiento de los mensajes que han visto y administrar su propio cronograma de consulta.

He visto a gente dedicar mucho tiempo a implementar cada vez más de los comportamientos que se obtienen de fábrica con un agente de mensajes adecuado para que ATOM funcione en algunos casos de uso. Por ejemplo, el patrón Competing Consumer describe un método mediante el cual se generan múltiples 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 en el que dos o más trabajadores vean el mismo mensaje, ya que terminaremos haciendo la misma tarea más de lo necesario. Con un agente de mensajes, una cola estándar se encargará de 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 un buen y resistente agente de mensajes disponible, considere usarlo para gestionar la publicación y suscripción a eventos. Pero si aún no tiene uno, échale un vistazo a ATOM, pero tenga en cuenta la falacia del costo irrecuperable. Si descubre que desea cada vez más el soporte que le brinda un agente de mensajes, en cierto punto es posible que desee cambiar su enfoque.

En cuanto a lo que realmente enviamos a través de estos protocolos asincrónicos, se aplican las mismas consideraciones que con la comunicación sincrónica. Si actualmente está satisfecho con la codificación de solicitudes y respuestas mediante JSON, continúe con esta práctica.

## Complejidades de las arquitecturas asincrónicas

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

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

Es hora de contar una historia que sirva de advertencia. En 2006, estaba trabajando en la creación de un sistema de fijación de precios para un banco. Analizábamos los acontecimientos del mercado y determinábamos qué artículos de una cartera debían tener un nuevo precio. Una vez que determinábamos la lista de cosas que debíamos analizar, las poníamos todas en una cola de mensajes. Utilizábamos una cuadrícula para crear un grupo de trabajadores de fijación de precios, lo que nos permitía ampliar o reducir la granja de fijación de precios según las necesidades. Estos trabajadores utilizaban el patrón de consumidor competitivo, cada uno devorando los mensajes lo más rápido posible hasta que no quedaba nada por procesar.

El sistema estaba funcionando y nos sentíamos bastante satisfechos. Sin embargo, un día, justo después de lanzar un nuevo producto, nos topamos con un problema desagradable. Nuestros trabajadores seguían muriendo. Y muriendo. Y muriendo.

Finalmente, localizamos el problema. Se había introducido un error que provocaba que un determinado tipo de solicitud de precios hiciera que un trabajador dejara de funcionar. Estábamos utilizando una cola de transacciones: cuando el trabajador murió, su bloqueo en la solicitud se agotó y la solicitud de precios se volvió a colocar en la cola, solo para que otro trabajador la recogiera y dejara de funcionar. Este fue un ejemplo clásico de lo que Martin Fowler llama una [comutación por error catastrófica](#).

Aparte del error en sí, no habíamos podido especificar un límite máximo de reintentos para el trabajo en la cola. Solucionamos el error en sí y también configuramos un límite máximo de reintentos. 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 se enviaban los mensajes si fallaban. También creamos una interfaz de usuario para ver esos mensajes y reintentarlos si era necesario. Este tipo de problemas no son inmediatamente obvios si solo estás familiarizado con la comunicación sincrónica punto a punto.

La complejidad asociada con las arquitecturas basadas en eventos y la programación asincrónica en general me lleva a creer que debes ser cauteloso con el entusiasmo con el que comienzas a adoptar estas ideas. Asegúrate de tener un buen control y considera seriamente el uso de identificadores de correlación, que te permiten rastrear solicitudes en todos los procesos.

límites, como veremos en profundidad en [el Capítulo 8](#).

También recomiendo encarecidamente Enterprise Integration Patterns (Addison-Wesley), que contiene muchos más detalles sobre los diferentes patrones de programación que puedes necesitar considerar en este espacio.

## Los servicios como máquinas de estados

Ya sea que elijas convertirte en un ninja de REST o te quedes con un mecanismo basado en RPC como SOAP, el concepto central del servicio como máquina de estados es poderoso. Hemos hablado antes (probablemente hasta la saciedad a esta altura) sobre cómo nuestros servicios se diseñan en torno a contextos delimitados. Nuestro microservicio de cliente posee toda la lógica asociada con el comportamiento en este contexto.

Cuando un consumidor quiere cambiar de cliente, envía una solicitud adecuada al servicio de atención al cliente. El servicio de atención al cliente, basándose en su lógica, decide si acepta o no la 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 hacer a 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 modelados explícitamente de esta manera es bastante poderoso. No solo tenemos un lugar para lidiar con las colisiones de estado (por ejemplo, alguien que intenta actualizar un cliente que ya ha sido eliminado), sino que también tenemos un lugar para adjuntar un comportamiento basado en esos cambios de estado.

Sigo pensando que REST sobre HTTP es una tecnología de integración mucho más sensata que muchas otras, pero sea cual sea la opción que elijas, ten en cuenta esta idea.

## Extensiones reactivas

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

Las diversas implementaciones de Rx han encontrado un hogar muy adecuado en los sistemas distribuidos. Nos permiten abstraer los detalles de cómo se realizan las llamadas y razonar sobre las cosas con mayor facilidad. Observo el resultado de una llamada a un servicio descendente. No me importa si fue una llamada de bloqueo o no bloqueo, simplemente espero la respuesta y reacciono. Lo bueno es que puedo componer varias llamadas juntas, lo que hace que manejar llamadas concurrentes a servicios descendentes sea mucho más fácil.

A medida que realice más llamadas de servicio, especialmente cuando realice varias llamadas para realizar una sola operación, eche un vistazo a las extensiones reactivas para la pila de tecnología que haya elegido. Es posible que se sorprenda de lo mucho más sencilla que puede llegar a ser su vida.

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

Una de las siglas que los desarrolladores escuchamos mucho es DRY: no te repitas. Aunque a veces su definición se simplifica y dice que queremos evitar duplicar el código, DRY significa más exactamente que queremos evitar duplicar el comportamiento y el conocimiento de nuestro sistema.

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

DRY es lo que nos lleva a crear código que se puede reutilizar. Extraemos código duplicado y lo convertimos en abstracciones que luego podemos llamar desde varios lugares. ¡Quizás lleguemos al punto de crear una biblioteca compartida que podamos usar en todas partes! Sin embargo, este enfoque puede ser engañosamente peligroso en una arquitectura de microservicios.

Una de las cosas que queremos evitar a toda costa es acoplar excesivamente un microservicio y los consumidores, de modo que cualquier pequeño cambio en el propio microservicio pueda provocar cambios innecesarios en el consumidor. Sin embargo, a veces el uso de código compartido puede crear precisamente este acoplamiento. Por ejemplo, en un cliente teníamos una biblioteca de objetos de dominio común que representaban las entidades principales que se utilizaban en nuestro sistema. Esta biblioteca era utilizada por todos los servicios que teníamos, pero cuando se realizaba un cambio en uno de ellos, todos los servicios debían actualizarse. Nuestro sistema se comunicaba a través de colas de mensajes, que también debían vaciarse de su contenido, que ahora era inválido, y ¡pobre de ti si lo olvidabas!

Si alguna vez su uso de código compartido se filtra fuera de los límites de su servicio, habrá introducido una forma potencial de acoplamiento. El uso de 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 hacer que este código sea compartido, la empresa lo copia para cada nuevo servicio para garantizar que el acoplamiento no se filtre.

Mi regla general es no violar DRY dentro de un microservicio, pero no se exceda en la violación de DRY en todos los servicios. Los males de un exceso de acoplamiento 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 la creación de 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 al cliente. Yo debería saberlo: yo mismo he hecho esto. Cuanta más lógica se cuela en la biblioteca del cliente, más se empieza a romper la cohesión y te encuentras teniendo que cambiar varios clientes para implementar correcciones en tu servidor. También limita las opciones tecnológicas, especialmente si exiges que se use la biblioteca del cliente.

Un modelo de bibliotecas de 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 todo el mundo acaba utilizando 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 que no trabajan en la propia API. Este grado de separación parece funcionar y evita algunos de los problemas de las bibliotecas de cliente. Parte de la razón por la que esto funciona tan bien es que el cliente está a cargo de cuándo se produce la actualización. Si opta por el camino de las bibliotecas de cliente, asegúrese de que este sea el caso.

Netflix, en particular, hace especial hincapié en la biblioteca de clientes, pero me preocupa que la gente lo vea únicamente desde el punto de vista de evitar la duplicación de código. De hecho, las bibliotecas de clientes que utiliza Netflix tienen tanto (si no más) que ver con garantizar la fiabilidad y la escalabilidad de sus sistemas. Las bibliotecas de clientes de Netflix se encargan del descubrimiento de servicios, los modos de fallo, 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 pieza de comunicaciones entre cliente y servidor se comportara bien en la escala masiva en la que opera Netflix. Su uso en Netflix ha facilitado sin duda la puesta en marcha y ha aumentado la productividad, al tiempo que ha garantizado el buen comportamiento del sistema. Sin embargo, según al menos una persona de Netflix, con el tiempo esto ha provocado un grado de acoplamiento entre cliente y servidor que ha sido problemático.

Si está pensando en utilizar una biblioteca de cliente, puede ser importante separar el código de cliente que se encarga del protocolo de transporte subyacente, que puede ocuparse de cuestiones como el descubrimiento y la falla del servicio, de las cuestiones relacionadas con el servicio de destino en sí. Decida si insistirá o no en que se utilice la biblioteca de cliente o si permitirá que personas que utilicen diferentes tecnologías realicen llamadas a la API subyacente.

Y, por último, asegúrese de que los clientes sean responsables de cuándo actualizar sus bibliotecas de cliente: ¡debemos asegurarnos de mantener la capacidad de lanzar nuestros servicios independientemente unos de otros!

## Acceso por referencia

Una consideración que quiero abordar 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 principales, como el Cliente. Ya hemos hablado sobre la importancia de la lógica asociada con el cambio de este Cliente que se mantiene en el servicio de atención al cliente, y que si queremos cambiarlo tenemos que enviar una solicitud al servicio de atención al cliente.

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

Cuando recuperamos un recurso de Cliente determinado del servicio de atención al cliente, podemos ver cómo se veía ese recurso cuando hicimos la solicitud. Es posible que después de haber solicitado ese recurso de Cliente , algo más lo haya cambiado. Lo que tenemos en realidad es un recuerdo de cómo se veía el recurso de Cliente en el pasado. Cuanto más tiempo conservemos este recuerdo, mayor será la probabilidad de que sea falso. Por supuesto, si evitamos solicitar datos más de lo necesario, nuestros sistemas pueden volverse mucho más eficientes.

A veces, este recuerdo es suficiente. Otras veces, es necesario saber si ha cambiado.

Entonces, si decide transmitir un recuerdo de cómo alguna vez se veía 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 solicitamos 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 está realmente poniendo en cola estas solicitudes o sacándolas 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 los eventos, decimos que esto sucedió, pero necesitamos saber qué sucedió. Si recibimos actualizaciones debido a un cambio en el recurso de un 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 para que podamos buscar su estado actual, podemos obtener lo mejor de ambos mundos.

Por supuesto, aquí hay que hacer otras concesiones cuando accedemos por referencia.

Si siempre vamos al servicio de atención al cliente para consultar la información asociada a un cliente determinado, la carga en el servicio de atención al cliente puede ser demasiado grande. Si proporcionamos información adicional cuando se recupera el recurso, que nos permita saber en qué momento se encontraba el recurso en el estado dado y quizás durante cuánto tiempo podemos considerar que esta información es reciente, entonces podemos hacer mucho con el almacenamiento en caché para reducir la carga. HTTP nos brinda gran parte de este soporte de manera predeterminada con una amplia variedad de controles de caché, algunos de los cuales analizaremos en más detalle.

en el Capítulo 11.

Otro problema es que algunos de nuestros servicios podrían no necesitar 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 deberíamos enviarle simplemente la dirección de correo electrónico y el nombre del cliente. No hay una regla estricta en este caso, pero tenga mucho cuidado al pasar datos en solicitudes cuando no sepa si están actualizados.

## Control de versiones

En todas las charlas que he dado sobre microservicios, me preguntan cómo se hace el control de versiones. La gente tiene la preocupación legítima de que, en algún momento, tendrán que hacer un cambio en la interfaz de un servicio y quieren entender cómo gestionarlo. Analicemos un poco el problema y veamos los distintos pasos que podemos seguir para solucionarlo.

## Aplazarlo tanto como sea posible

La mejor manera de reducir el impacto de realizar cambios importantes es evitarlos en primer lugar. Puede lograr esto eligiendo la tecnología de integración adecuada, como hemos comentado a lo largo de este capítulo. La integración de bases de datos es un gran ejemplo de tecnología que puede dificultar enormemente la posibilidad de evitar cambios importantes. REST, por otro lado, ayuda porque es menos probable que los cambios en los detalles de implementación interna resulten en un cambio en la interfaz del servicio.

Otra clave para aplazar un cambio radical es fomentar el buen comportamiento en los clientes y evitar que se apeguen demasiado a sus servicios desde el principio. 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 el ID 1234. Se activa y recupera al cliente con ese ID y obtiene algo parecido a la respuesta que se muestra en [el Ejemplo 4-3](#).

---

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

```
<cliente>
  <nOMBRE>Sam</nOMBRE> <apELLIDO>Newman</
  apELLIDO> <correo electrónico>sam@maggiebrain.com</
  correo electrónico> <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 extraer los campos que nos interesan e ignorar el resto. Algunas tecnologías de enlace, especialmente las que se utilizan en lenguajes fuertemente tipados, pueden intentar enlazar todos los campos, ya sea que el consumidor los quiera o no. ¿Qué ocurre si nos damos cuenta de que nadie está utilizando el número de teléfono y decidimos eliminarlo? Esto podría provocar que los consumidores se rompan innecesariamente.

De la misma manera, ¿qué sucedería si quisieramos reestructurar nuestro objeto Cliente para admitir más detalles, quizás agregando más estructura como en el [Ejemplo 4-4](#)? Los datos que nuestro servicio de correo electrónico desea siguen estando allí, y con el mismo nombre, pero si nuestro código hace suposiciones muy explícitas sobre dónde se almacenarán los campos de nombre y apellido , podría volver a fallar. En este caso, podríamos usar XPath para extraer los campos que nos interesan, lo que nos permitiría ser ambivalentes sobre dónde se encuentran los campos, siempre y cuando podamos encontrarlos. Este patrón (implementar un lector capaz de ignorar los cambios que no nos interesan) es lo que Martin Fowler llama un [Lector Tolerante](#).

---

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

```
<cliente> <naming>
  <nOMBRE>Sam</
  nombre> <apELLIDO>Newman</apELLIDO>
  <apodo>Maggiebrain</apodo> <nOMBRE
  completo>Sam "Maggiebrain" Newman</nOMBRE completo>
  </naming> <email>sam@maggiebrain.com</email>
```

</cliente>

El ejemplo de un cliente que intenta ser lo más flexible posible al consumir un servicio demuestra [la Ley de Postel](#). (también conocido como principio de robustez), que establece: "Sé conservador en lo que haces, sé liberal en lo que aceptas de los demás". El contexto original de esta sabiduría era la interacción de dispositivos en redes, donde deberías esperar que sucedieran 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 está consumiendo cambie sin que nosotros tengamos que cambiar.

## Detecte cambios importantes a tiempo

Es fundamental asegurarse de detectar los cambios que afectarán a los consumidores lo antes posible, porque incluso si elegimos la mejor tecnología posible, las interrupciones pueden ocurrir. Estoy totalmente a favor de utilizar contratos impulsados por el consumidor, que abordaremos en [el Capítulo 7](#), para ayudar a detectar estos problemas en una etapa temprana. Si está brindando soporte a varias bibliotecas de cliente diferentes, ejecutar pruebas utilizando cada biblioteca que brinda soporte contra el servicio más reciente es otra técnica que puede ayudar. Una vez que se da cuenta de que va a causar una interrupción a un consumidor, tiene la opción de intentar evitar la interrupción por completo o aceptarla y comenzar a tener las conversaciones adecuadas con las personas que se encargan de los servicios que consumen.

## Utilice el control de versiones semántico

¿No sería fantástico si, como cliente, pudieras ver simplemente el número de versión de un servicio y saber si puedes integrarlo? **Versiones semánticas** es una especificación que permite justamente eso. Con el control de versiones semántico, cada número de versión tiene el formato PRINCIPAL.MENOR.PARCHE. Cuando el número PRINCIPAL aumenta, significa que se han realizado cambios incompatibles con versiones anteriores. Cuando el número MENOR aumenta, se ha agregado una nueva funcionalidad que debería ser compatible con versiones anteriores. Finalmente, un cambio en PARCHE indica que se han realizado correcciones de errores en la funcionalidad existente.

Para ver lo útil que 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 característica que haga que el servicio de atención al cliente cambie a la versión 1.3.0, nuestra aplicación de soporte técnico no debería ver ningún cambio en su comportamiento y no se debería esperar que realice ningún cambio. Sin embargo, no podemos garantizar que podamos funcionar con la versión 1.1.0 del servicio de atención al cliente, ya que podemos depender de la funcionalidad agregada en la versión 1.2.0. También podríamos esperar tener que realizar cambios en nuestra aplicación si sale una nueva versión 2.0.0 del servicio de atención al cliente.

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

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

## Coexisten diferentes puntos finales

Si hemos hecho todo lo posible para evitar introducir un cambio de interfaz que rompa las reglas, nuestro próximo trabajo es limitar el impacto. Lo que queremos evitar es obligar a los consumidores a actualizar al mismo ritmo que nosotros, ya que siempre queremos mantener la capacidad de lanzar microservicios de forma independiente. Un enfoque que he utilizado con éxito para manejar esto es hacer que coexistan las interfaces antiguas y nuevas en el mismo servicio en ejecución. Por lo tanto, si queremos lanzar un cambio que rompa las reglas, implementamos una nueva versión del servicio que exponga tanto la versión antigua como la nueva del punto final.

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

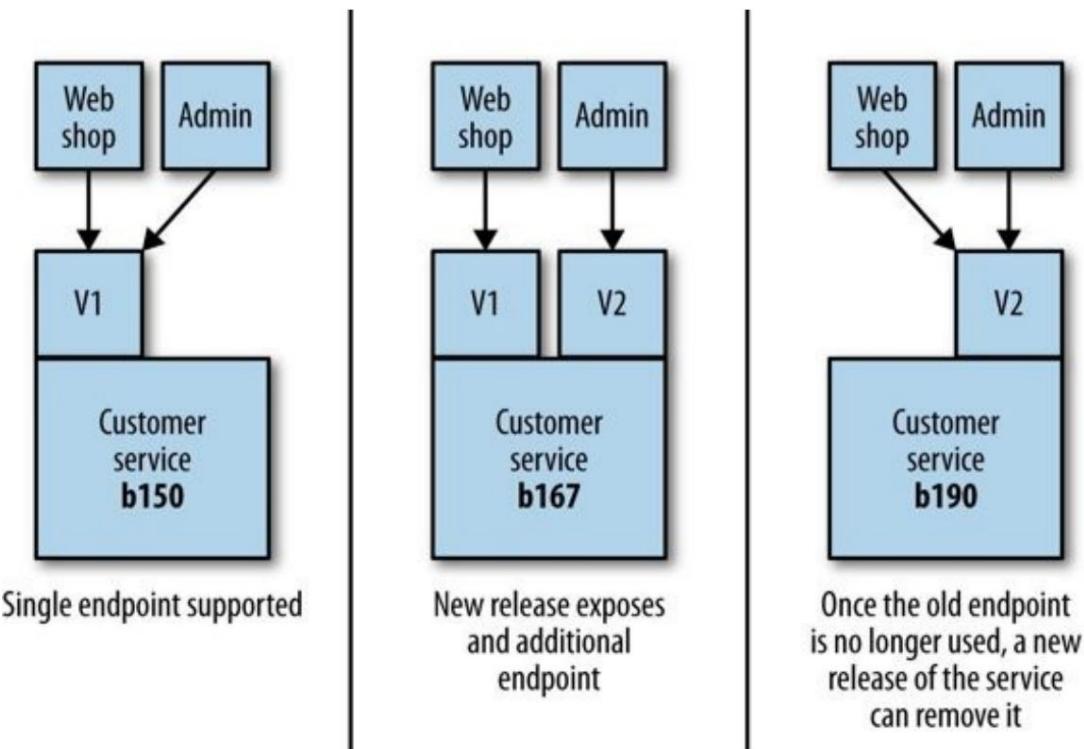


Figura 4-5. La coexistencia de diferentes versiones de puntos finales permite que los consumidores migren gradualmente

La última vez que utilicé 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 tres versiones diferentes del punto de conexión. ¡Esto no es algo que recomendaría! Mantener todo el código y las pruebas asociadas necesarias para garantizar que todo funcionara fue una carga absolutamente adicional. Para que esto fuera más manejable, transformamos internamente todas las solicitudes al punto de conexión V1 en una solicitud V2 y, luego, las solicitudes V2 al punto de conexión V3. Esto significó que pudimos delinear claramente qué código se retiraría cuando el o los puntos de conexión antiguos dejaran de funcionar.

Este es, en efecto, un ejemplo del patrón de expansión y contracción, que nos permite introducir cambios radicales en fases. Ampliamos las capacidades que ofrecemos, respaldando tanto las antiguas como las nuevas.

Maneras de hacer algo. Una vez que los antiguos consumidores hacen las cosas de la nueva manera, contraemos nuestra API y eliminamos la funcionalidad antigua.

Si va a hacer que los puntos finales coexistan, necesita una forma para que los llamantes enruten sus solicitudes en consecuencia. En el caso de los sistemas que utilizan HTTP, he visto que esto se hace con números de versión en los encabezados de solicitud y también en el propio URI, por ejemplo, /v1/customer/ o /v2/customer/. No sé qué enfoque tiene más sentido. Por un lado, me gusta que los URI sean opacos para disuadir a los clientes de codificar plantillas de URI, pero por otro lado, este enfoque hace que las cosas sean muy obvias y puede simplificar el enrutamiento de solicitudes.

En el caso de RPC, las cosas pueden ser un poco más complicadas. He solucionado este problema con los buffers de protocolo colocando mis métodos en diferentes espacios de nombres (por ejemplo, v1.createCustomer y v2.createCustomer ), pero cuando intentas admitir diferentes versiones de los mismos tipos que se envían a través de la red, esto puede volverse realmente complicado.

## Utilice múltiples versiones de servicio concurrentes

Otra solución de control de versiones que se suele citar es la de tener distintas versiones del servicio activas a la vez y que los consumidores más antiguos dirijan su tráfico a la versión anterior, mientras que las versiones más nuevas ven la nueva, como se muestra en [la Figura 4-6](#). Este es el enfoque que Netflix utiliza con moderación en situaciones en las que el coste de cambiar a los consumidores más antiguos es demasiado alto, especialmente en casos excepcionales en los que los dispositivos antiguos siguen vinculados a versiones anteriores de la API.

Personalmente, no soy partidario de esta idea y comprendo por qué Netflix la utiliza rara vez. En primer lugar, si necesito corregir un error interno en mi servicio, ahora tengo que corregir e implementar dos conjuntos diferentes de servicios. Probablemente esto significaría que tengo que ramificar el código base de mi servicio, y esto siempre es problemático. En segundo lugar, significa que necesito inteligencia para manejar la dirección de los consumidores al microservicio correcto. Este comportamiento inevitablemente termina en algún middleware o en un montón de scripts de nginx , lo que hace que sea más difícil razonar sobre el comportamiento del sistema.

Por último, tenga en cuenta cualquier estado persistente que nuestro servicio pueda gestionar. 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.

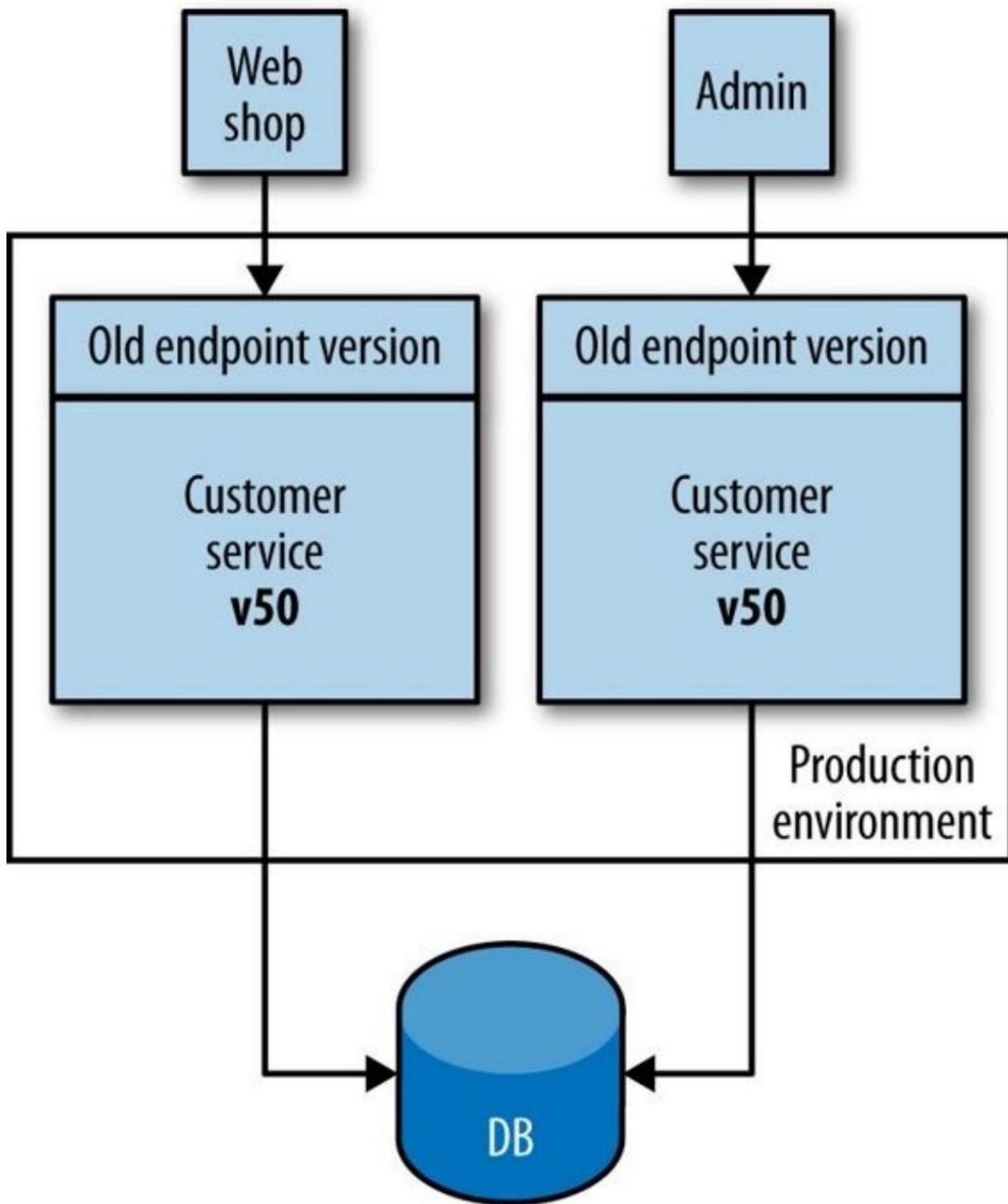


Figura 4-6. Ejecución de varias versiones del mismo servicio para dar soporte a puntos de conexión antiguos

La coexistencia de versiones de servicio simultáneas durante un breve período de tiempo puede tener mucho sentido, especialmente cuando se realizan tareas como implementaciones blue/green o lanzamientos canary (hablaremos más sobre estos patrones en [el Capítulo 7](#)). En estas situaciones, es posible que coexistan versiones solo durante unos minutos o quizás horas, y normalmente tendremos solo dos versiones diferentes del servicio presentes al mismo tiempo. Cuanto más tiempo lleve que los consumidores se actualicen a la versión más nueva y se lancen, 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 abordado realmente el mundo de la interfaz de usuario. Algunos de ustedes pueden estar simplemente proporcionando una API fría, dura y clínica a sus clientes, pero muchos de nosotros queremos crear interfaces de usuario hermosas y funcionales que deleiten a nuestros clientes. Pero realmente necesitamos pensar en ellas en el contexto de la integración. La interfaz de usuario, después de todo, es donde reuniremos todos estos microservicios en algo que tenga sentido para nuestros clientes.

En el pasado, cuando empecé a trabajar con la informática, hablábamos principalmente de grandes clientes que se ejecutaban en nuestros equipos de escritorio. Pasé muchas horas con Motif y luego con Swing intentando que mi software fuera lo más agradable de usar posible. A menudo, estos sistemas solo servían para la creación y manipulación de archivos locales, pero muchos de ellos tenían un componente del lado del servidor. Mi primer trabajo en ThoughtWorks implicó la creación de un sistema de punto de venta electrónico basado en Swing que era solo una parte de una gran cantidad de partes móviles, la mayoría de las cuales estaban en el servidor.

Luego llegó la Web. Empezamos a pensar en nuestras interfaces de usuario como algo más simple , con más lógica del 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. Todas las interacciones se gestionaban del lado del servidor, a través de GET y POST que se activaban cuando el usuario hacía clic en enlaces o completaba formularios.

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

## Hacia lo digital

En los últimos años, las organizaciones han dejado de pensar que la web o los dispositivos móviles deberían tratarse de forma diferente y, en cambio, están pensando en lo digital de forma más integral. ¿Cuál es la mejor manera de que nuestros clientes utilicen los servicios que ofrecemos? ¿Y qué efectos tiene eso en 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 crear diferentes experiencias para nuestros clientes en su aplicación de escritorio, dispositivo móvil, dispositivo portátil o incluso en forma física si visitan nuestra tienda física.

Por lo tanto, piense en las interfaces de usuario como capas de composición: lugares donde entrelazamos los distintos hilos de las capacidades que ofrecemos. Teniendo esto en mente, ¿cómo unimos 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 el navegador que utilizan los visitantes o su resolución. Pero los dispositivos móviles han traído consigo toda una serie de nuevas restricciones. 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 jugar 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 con el botón derecho del ratón en una tableta. En un teléfono móvil, quizá quiera diseñar mi interfaz para que se utilice principalmente con una sola mano, y que la mayoría de las operaciones se controlen con el pulgar. En otros lugares, podría permitir que la gente interactúe con los servicios a través de SMS en lugares donde el ancho de banda es escaso (el uso de SMS como interfaz es muy común en el sur global, por ejemplo).

Por lo tanto, aunque nuestros servicios básicos (nuestra oferta básica) puedan ser los mismos, necesitamos una forma de adaptarlos a las diferentes restricciones que existen para cada tipo de interfaz. Cuando analizamos diferentes estilos de composición de 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 de la API

Suponiendo que nuestros servicios ya se comunican entre sí en XML o JSON a través de HTTP, una opción obvia disponible para nosotros es 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 utilizar solicitudes GET de JavaScript para recuperar datos, o solicitudes POST para cambiarlos. Incluso para las aplicaciones móviles nativas, iniciar comunicaciones HTTP es bastante sencillo. La interfaz de usuario tendría que crear los distintos componentes que forman la interfaz, manejando la sincronización del estado y similares con el servidor. Si estuviéramos utilizando 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.

Este enfoque tiene un par de desventajas. En primer lugar, tenemos poca capacidad para adaptar las respuestas a diferentes tipos de dispositivos. Por ejemplo, cuando recupero el registro de un cliente, ¿tengo que recuperar los mismos datos para una tienda de telefonía móvil que para una aplicación de soporte técnico? Una solución a este enfoque es permitir que los consumidores especifiquen qué campos recuperar 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 no tienen ninguna influencia en la forma en que estos se muestran a los usuarios; por ejemplo, si otro equipo está creando la interfaz de usuario, podríamos estar volviendo a los viejos y malos tiempos de la arquitectura en capas, en los que incluso para realizar pequeños cambios era necesario solicitar cambios a varios equipos.

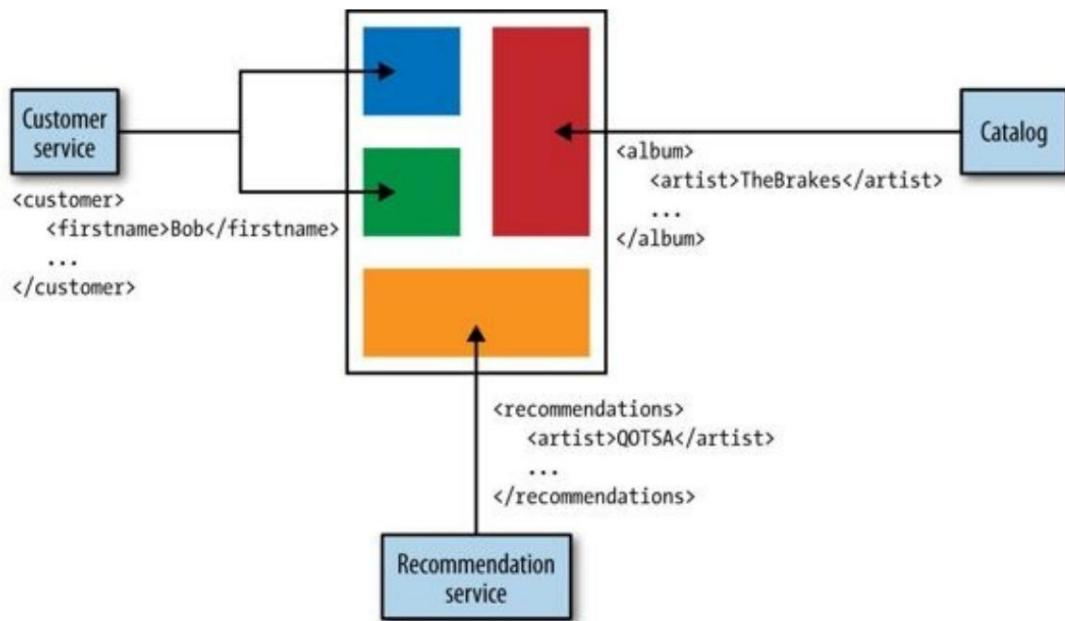


Figura 4-7. Uso de múltiples API para presentar una interfaz de usuario

Esta comunicación también podría ser bastante conversacional. 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 en este caso, ya que podría exponer llamadas que agreguen múltiples 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 realice llamadas a la API y asigne todo a los controles de la interfaz de usuario, podríamos hacer que nuestros servicios proporcionen partes de la interfaz de usuario directamente y luego simplemente 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 interfaz de usuario para crear una interfaz de usuario general. Podría representarse 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 más generales de una interfaz de usuario. De esta manera, en lugar de crear pequeños widgets, se ensamblan paneles completos de una aplicación cliente gruesa o, tal vez, un conjunto de páginas para un sitio web.

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

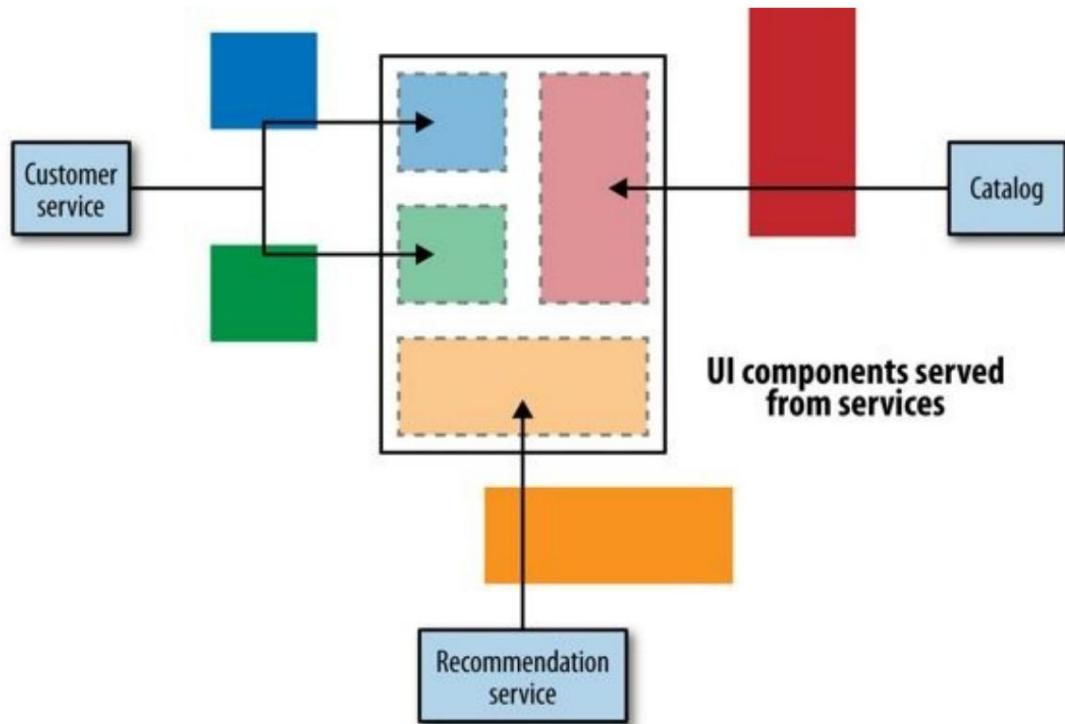


Figura 4-8. Servicios que proporcionan directamente componentes de la interfaz de usuario para el ensamblaje

Aún necesitas algún tipo de capa de ensamblaje para unir estas partes. Esto podría ser tan simple como una plantilla del lado del servidor o, si cada conjunto de páginas proviene de una aplicación diferente, tal vez necesites algún enrutamiento de URL inteligente.

Una de las principales ventajas de este enfoque es que el mismo equipo que realiza cambios en los servicios también puede encargarse de realizar cambios en esas partes de la interfaz de usuario. Esto nos permite implementar los cambios más rápidamente. Sin embargo, este enfoque presenta algunos problemas.

En primer lugar, garantizar la coherencia de la experiencia del usuario es algo que debemos abordar. Usuarios

Desea tener una experiencia fluida, no sentir que las distintas 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 dinámicas, donde se pueden compartir recursos como componentes HTML, CSS e imágenes para ayudar a brindar cierto nivel de coherencia.

Otro problema es más difícil de abordar. ¿Qué sucede con las aplicaciones nativas o los clientes pesados? No podemos ofrecer componentes de interfaz de usuario. Podríamos utilizar un enfoque híbrido y utilizar aplicaciones nativas para ofrecer componentes HTML, pero se ha demostrado una y otra vez que este enfoque tiene desventajas. Por lo tanto, si necesita una experiencia nativa, tendremos que recurrir a un enfoque en el que la aplicación frontend realice llamadas a la API y gestione la interfaz de usuario por sí misma. Pero incluso si consideramos interfaces de usuario solo web, es posible que aún queramos tratamientos muy diferentes para diferentes tipos de dispositivos. La creación de componentes responsivos 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 incorporar recomendaciones dinámicas en otro lugar? Cuando hago una búsqueda, quiero que el tipo que aparece a continuación 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 recurramos a realizar simplemente llamadas a la API.

## Backends para frontends

Una solución común al problema de las interfaces conversacionales con los servicios de backend, o la necesidad de variar el contenido para diferentes tipos de dispositivos, es tener un punto final de agregación del lado del servidor, o puerta de enlace de API. Esto puede ordenar múltiples llamadas de backend, variar y agregar contenido si es necesario para diferentes dispositivos y ofrecerlo, 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.

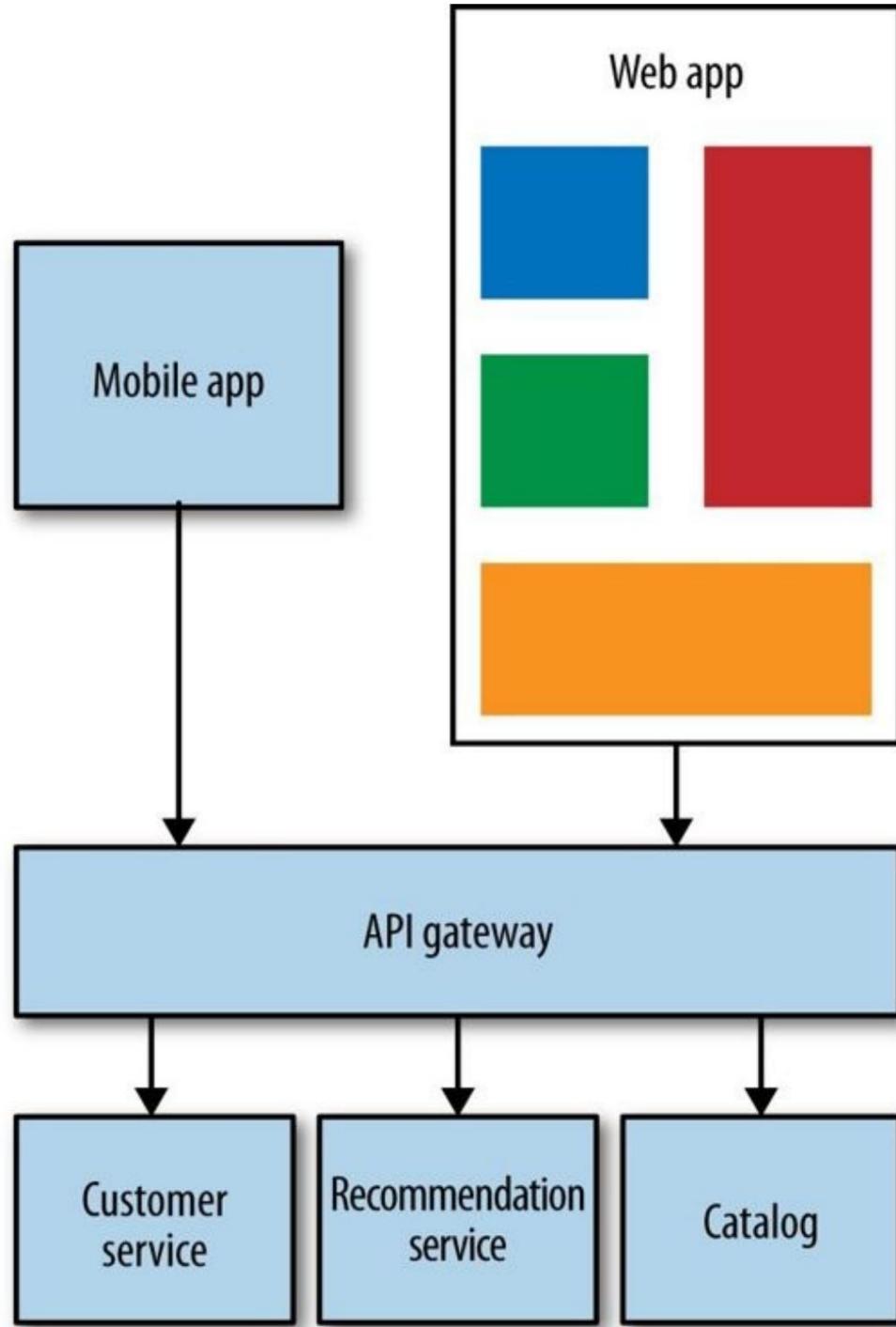


Figura 4-9. Uso de una única puerta de enlace monolítica para gestionar llamadas hacia y desde las interfaces de usuario

El problema que puede ocurrir es que normalmente tendremos una capa gigante para todos nuestros servicios.

Esto hace que todo se mezcle y, de repente, empezamos a perder el aislamiento de nuestras distintas interfaces de usuario, lo que limita nuestra capacidad de publicarlas 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](#).

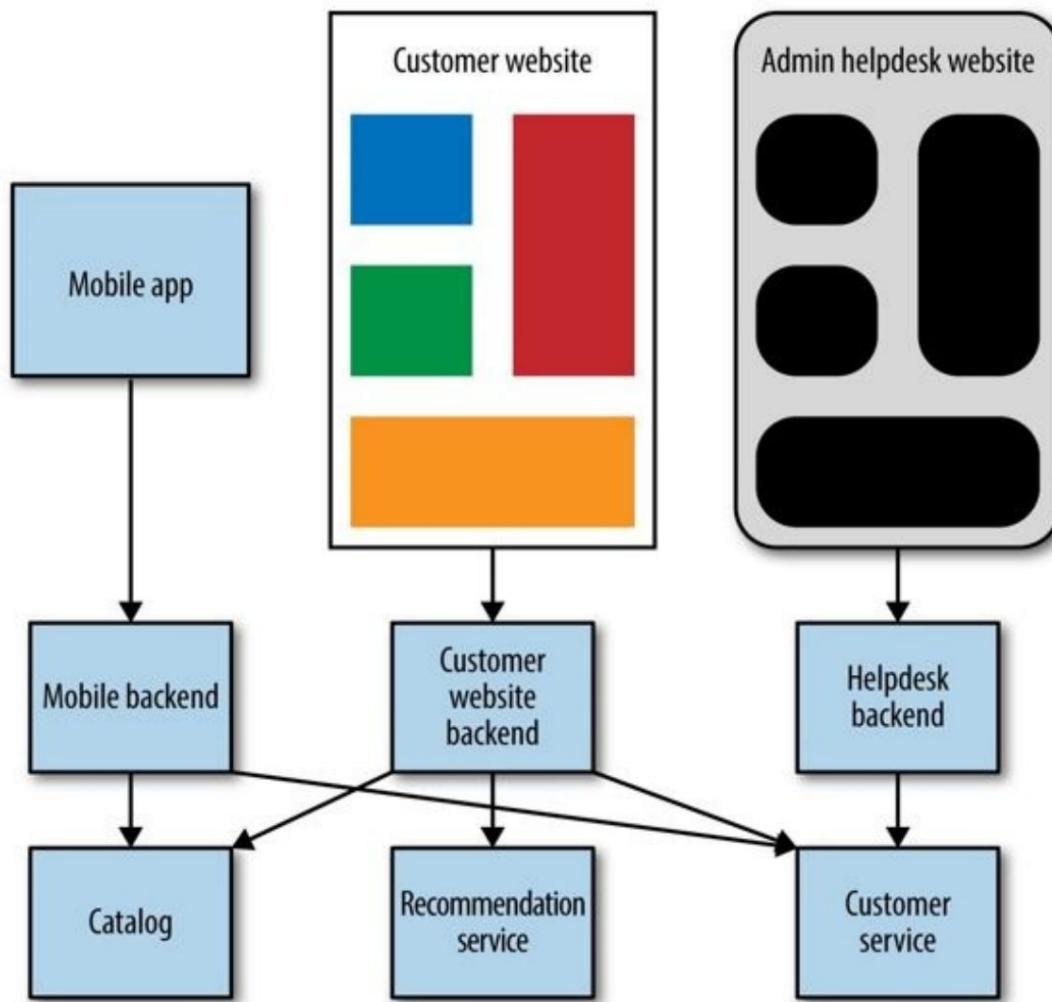


Figura 4-10. Uso de backends dedicados para frontends

Este patrón se conoce a veces como backends para frontends (BFFs). Permite que el equipo que se centra en una determinada interfaz de usuario también gestione sus propios componentes del lado del servidor. Puedes 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 en el servidor, mientras que otros pueden necesitar mucho más. Si necesita una capa de autenticación y autorización de API, esta puede ubicarse entre nuestras mejores amigas y nuestras interfaces de usuario. Analizaremos esto más a fondo en [el Capítulo 9](#).

El peligro de este enfoque es el mismo que el de cualquier capa de agregación: puede adoptar una lógica que no debería.

La lógica empresarial de las distintas capacidades que utilizan estos backends debería permanecer en los propios servicios. Estos BFF solo deberían contener comportamientos específicos para ofrecer una experiencia de usuario particular.

## Un enfoque híbrido

Muchas de las opciones mencionadas anteriormente no tienen por qué ser válidas para todos. Podría ver una organización adoptando el enfoque de ensamblaje basado en fragmentos para crear un sitio web, pero utilizando un enfoque de backends por frontends cuando se trata de su aplicación móvil. El punto clave es que necesitamos mantener 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 los datos de los clientes se encuentre dentro de esos servicios que manejan esas operaciones y no se disperse por 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 estrategias para desmantelar los sistemas existentes que están bajo nuestro control, pero ¿qué sucede cuando no podemos cambiar las cosas con las que hablamos? Por muchas razones válidas, las organizaciones para las que trabajamos compran software comercial listo para usar (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 estás leyendo este libro, probablemente trabajes en una organización que escribe código. Es posible que escribas software para tus propios fines internos o para un cliente externo, o para ambos.

Sin embargo, incluso si eres una organización con la capacidad de crear una cantidad significativa de software personalizado, seguirás utilizando productos de software proporcionados por terceros, ya sean comerciales o de código abierto. ¿A qué se debe 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 nóminas. Crear todos ellos para su propio uso sería una tarea titánica. En segundo lugar, y lo más importante, ¡no sería rentable! El coste de crear su propio sistema de correo electrónico, por ejemplo, probablemente eclipsará el coste de utilizar una combinación existente de servidor y cliente de correo, incluso si opta por opciones comerciales.

Mis clientes a menudo se enfrentan a la pregunta “¿Debería desarrollar o debería comprar?”. En general, el consejo que mis colegas y yo damos cuando mantenemos esta conversación con una organización empresarial promedio se reduce a “Desarrolle si es exclusivo de lo que usted hace y puede considerarse un activo estratégico; compre si su uso de la herramienta no es tan especial”.

Por ejemplo, una organización promedio no consideraría que su sistema de nóminas sea un activo estratégico. En general, la gente recibe el mismo salario en todo el mundo. Asimismo, la mayoría de las organizaciones tienden a comprar sistemas de gestión de contenido (CMS) listos para usar, ya que el uso de una herramienta de este tipo 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 gestión 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. ¿Por qué?

## Falta de control

Uno de los desafíos asociados 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 ya 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? Depende del proveedor. ¿Puede almacenar la configuración de la herramienta en el control de versiones y reconstruirla desde cero, de modo de permitir la integración continua de personalizaciones? Depende de las decisiones que tome el proveedor.

Si tiene suerte, el proceso de selección de la herramienta ha tenido en cuenta lo fácil (o difícil) que resulta trabajar con ella desde el punto de vista del desarrollo. Pero incluso en ese caso, en realidad está cediendo cierto nivel de control a un tercero. El truco consiste en volver a adaptar el trabajo de integración y personalización a sus condiciones.

## Personalización

Muchas de las herramientas que compran las organizaciones empresariales se venden por su capacidad de personalizarse en gran medida solo para usted. ¡Tenga 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 caro que construir algo a medida desde cero. Si ha decidido comprar un producto pero las capacidades particulares que ofrece no son tan especiales para usted, puede que tenga más sentido cambiar el modo en que funciona su organización en lugar de embarcarse en una personalización compleja.

Los sistemas de gestión de contenido 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 mínima en la herramienta subyacente puede arruinar cualquier personalización que hayas realizado.

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

## Espaguetis de integración

Otro desafío es cómo se integra con la herramienta. Como comentamos antes, es importante pensar detenidamente cómo se integran los servicios y, lo ideal, es estandarizar una pequeña cantidad de tipos de integración. Pero si un producto decide utilizar un protocolo binario propietario, otro alguna variante de SOAP y otro XML-RPC, ¿qué queda? Peor aún son las herramientas que permiten acceder directamente a sus almacenes de datos subyacentes, lo que genera los mismos problemas de acoplamiento que comentamos antes.

## En tus propios términos

Los productos COTS y SAAS tienen su lugar, y para la mayoría de nosotros no es viable (ni sensato) crear todo desde cero. ¿Cómo podemos resolver estos desafíos?

La clave es volver a hacer las cosas según tus propios términos.

La idea central aquí es hacer cualquier personalización en una plataforma que tú controlas y limitar 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 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 puede haber creado usted mismo.

La tentación (y a menudo el atractivo de los CMS) es que se puede personalizar el CMS para que recopile todo este contenido especial y lo muestre al mundo exterior. Sin embargo, el entorno de desarrollo de los CMS promedio es terrible.

Veamos en qué se especializa el CMS promedio y para qué probablemente lo compramos: creación y gestión de contenido. La mayoría de los CMS son bastante malos incluso en el diseño de páginas, y suelen ofrecer herramientas de arrastrar y soltar que no dan la talla. E incluso así, terminas necesitando a alguien que entienda HTML y CSS para ajustar las plantillas del CMS. Suelen ser plataformas terribles para crear código personalizado.

¿La respuesta? Proporcione al CMS su propio servicio que proporcione 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, usted escribe el código y lo integra con los servicios como desee. Tiene control sobre la escalabilidad del sitio web (muchos CMS comerciales proporcionan sus propios complementos propietarios para manejar la carga) y puede elegir el sistema de plantillas que tenga sentido.

La mayoría de los CMS también ofrecen API para permitir la creación de contenido, por lo que también tiene la posibilidad de presentar su propia fachada de servicio. En algunas situaciones, incluso hemos utilizado una fachada de este tipo para abstraer las API para recuperar contenido.

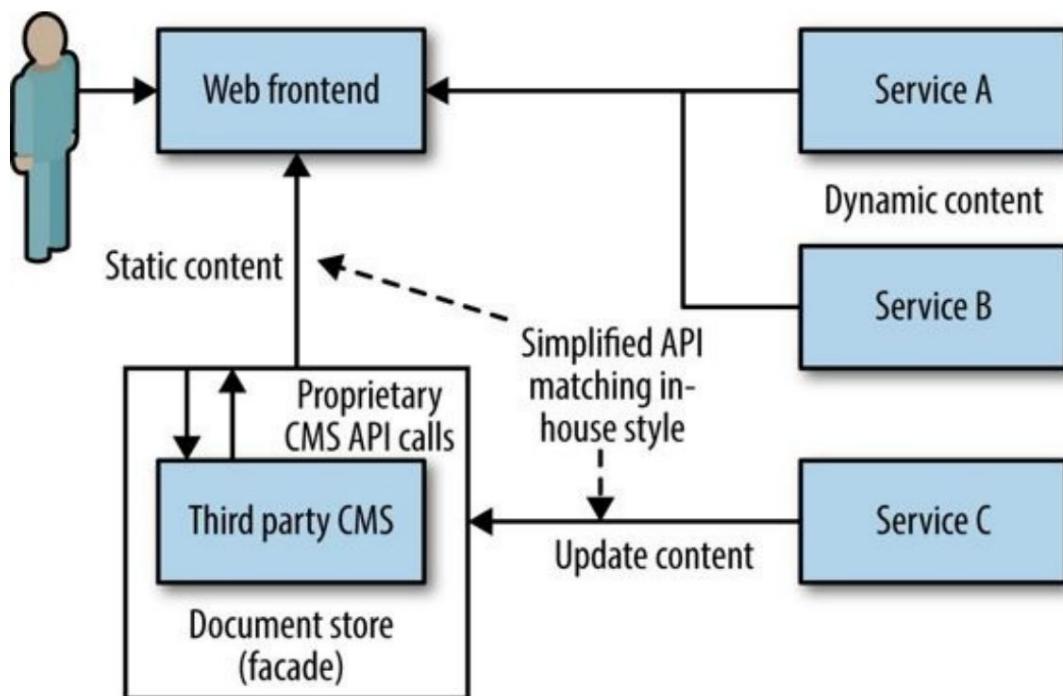


Figura 4-11. Ocultar un CMS utilizando su propio servicio

Hemos utilizado este patrón varias veces en ThoughtWorks en los últimos años, y yo mismo lo he hecho más de una vez. Un ejemplo notable fue el de un cliente que buscaba lanzar un nuevo sitio web para sus productos. Inicialmente, quería crear toda la solución en el CMS, pero aún no había elegido uno. En su lugar, sugerimos este enfoque y comenzamos el desarrollo del sitio web principal. Mientras esperábamos que se seleccionara la herramienta CMS, lo simulamos con un servicio web que solo mostraba contenido estático. Terminamos lanzando el sitio mucho antes de que se seleccionara el CMS utilizando nuestro servicio de contenido falso en producción para mostrar el contenido en el sitio en vivo. Más tarde, pudimos simplemente colocar la herramienta finalmente seleccionada sin ningún cambio en la aplicación principal.

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

#### Ejemplo: El sistema CRM multifunción

La herramienta CRM (Customer Relationship Management) es una bestia con la que nos topamos a menudo y que puede infundir miedo incluso en el corazón del arquitecto más aguerrido. Este sector, como lo ejemplifican proveedores como Salesforce o SAP, está plagado de herramientas que intentan hacer todo por ti. Esto puede hacer que la propia herramienta se convierta en un único punto de fallo y en un enredo de dependencias. Muchas de las implementaciones de herramientas 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 suele ser pequeño al principio, pero con el tiempo se convierte en una parte cada vez más importante del funcionamiento de su organización. El problema es que la dirección y las decisiones que se toman en torno a este sistema, que ahora es vital, suelen estar a cargo del propio proveedor de la herramienta, no de usted.

Recientemente participé en un ejercicio para intentar recuperar algo de control. La organización con la que trabajaba se dio cuenta de que, si bien utilizaba la herramienta CRM para muchas cosas,

No estaba obteniendo el valor de los crecientes costos asociados con la plataforma. Al mismo tiempo, varios sistemas internos estaban utilizando 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 nuestros negocios y también sentaran las bases para una posible migración.

Lo primero que hicimos fue identificar los conceptos básicos de nuestro dominio que el sistema CRM poseía actualmente. Uno de ellos era el concepto de proyecto , es decir, algo a lo que se podía asignar un miembro del personal. Muchos otros sistemas necesitaban información sobre proyectos.

Lo que hicimos fue crear un servicio de proyectos. Este servicio exponía los proyectos como recursos RESTful y los sistemas externos podían trasladar sus puntos de integración al nuevo servicio, más fácil de usar. Internamente, el servicio de proyectos era solo una fachada que ocultaba los detalles de la integración subyacente. Puede verlo en [la Figura 4-12](#).

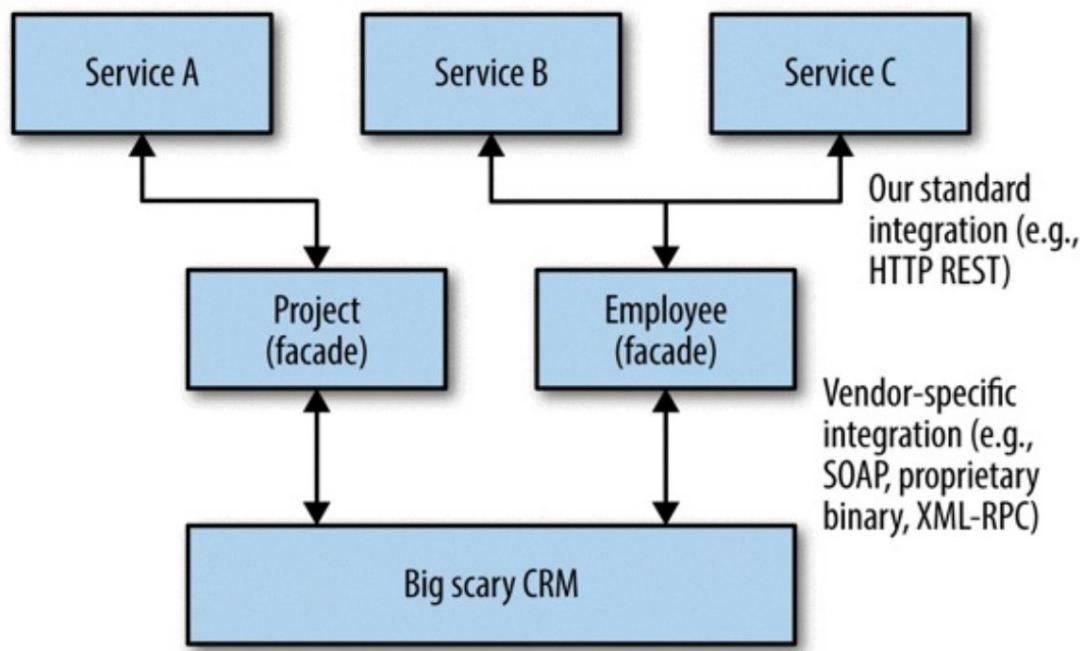


Figura 4-12. Uso de servicios de fachada para enmascarar el CRM subyacente

El trabajo, que en el momento de escribir este artículo todavía estaba en marcha, consistía en identificar otros conceptos de dominio que el CRM estaba manejando y crear más fachadas para ellos. Cuando llegue el momento de migrar del CRM subyacente, podremos analizar cada fachada por separado para decidir si una solución de software interna o algo disponible en el mercado podría ser la solución adecuada.

## 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 en este caso es el [patrón de aplicación Strangler](#). Al igual que en nuestro ejemplo de poner al frente del sistema CMS nuestro propio código, con un estrangulador se capturan e interceptan las llamadas al sistema antiguo. Esto le permite decidir si dirige estas llamadas al código existente, heredado, o las dirige al nuevo código que haya escrito. Esto le permite reemplazar la funcionalidad con el tiempo sin necesidad de una reescritura radical.

En el caso de los microservicios, en lugar de tener una única aplicación monolítica que intercepte todas las llamadas al sistema heredado existente, puede utilizar 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 utilizar un proxy para que lo haga por usted.

## Resumen

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

- Evite la integración de bases de datos a toda costa.
- Comprenda las ventajas y desventajas entre REST y RPC, pero considere seriamente REST como un buen punto de partida para la integración de solicitud/respondida.
- Prefiero la coreografía a la orquestación.
- Evite cambios radicales y la necesidad de crear versiones comprendiendo la Ley de Postel y utilizando lectores tolerantes.
- Piense en las interfaces de usuario como capas de composición.

Hemos cubierto bastante aquí y no pudimos profundizar en todos estos temas.

Sin embargo, esto debería ser una buena base para comenzar y orientarte en la dirección correcta si deseas aprender más.

También dedicamos algún tiempo a analizar cómo trabajar con sistemas que no están completamente bajo nuestro control en forma de productos COTS. ¡Resulta que esta descripción se puede aplicar con la misma facilidad al software que escribimos!

Algunos de los enfoques que se describen aquí se aplican igualmente bien al software heredado , pero ¿qué sucede si queremos abordar la tarea, a menudo monumental, de poner en orden estos sistemas antiguos y descomponerlos en partes más utilizables? Analizaremos este tema en detalle en el próximo capítulo.

# Capítulo 5. División del monolito

---

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

El monolito crece con el tiempo. Adquiere nuevas funciones 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 acabar con esta bestia.

## Todo es cuestión de costuras

En el capítulo 3 , comentamos que queremos que nuestros servicios sean muy cohesivos y estén 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. De la misma manera, el acoplamiento flexible en realidad no existe: si quiero hacer un cambio en una línea de código, puedo hacerlo con bastante facilidad, pero no puedo implementar ese cambio sin afectar potencialmente a gran parte del resto del monolito, y seguramente tendrá que volver a implementar todo el sistema.

En su libro Working Effectively with Legacy Code (Prentice-Hall), Michael Feathers define el concepto de costura , es decir, una parte del código que se puede tratar de forma aislada y en la que se puede trabajar sin afectar al resto de la base de código. También queremos identificar las costuras, pero en lugar de buscarlas con el fin de limpiar nuestra base de código, queremos identificar las costuras que pueden convertirse en límites del servicio.

Entonces, ¿qué hace que una unión sea buena? Bueno, como hemos comentado anteriormente, los contextos delimitados son excelentes uniones, porque por definición representan límites cohesivos y, sin embargo, débilmente acoplados en una organización. Por lo tanto, el primer paso es comenzar a identificar estos límites en nuestro código.

La mayoría de los lenguajes de programación ofrecen 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 gran parte de lo que necesitamos. Todos los demás lenguajes de programación convencionales tienen conceptos similares incorporados, y JavaScript es posiblemente una excepción.

# Desmembrando MusicCorp

Imaginemos que tenemos un gran servicio monolítico de backend que representa una cantidad sustancial del comportamiento de los sistemas en línea de MusicCorp. Para comenzar, deberíamos identificar los contextos delimitados de alto nivel que creemos que existen en nuestra organización, como analizamos en el [Capítulo 3](#). A continuación, queremos intentar comprender a qué contextos acotados se asigna el monolito. Imaginemos que inicialmente identificamos cuatro contextos que creemos que cubre nuestro backend monolítico:

## Catalogar

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

## Finanzas

Informes de cuentas, pagos, reembolsos, etc.

## Depósito

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

## Recomendación

Nuestro revolucionario sistema de recomendación, pendiente de patente, es un código altamente complejo escrito por un equipo con más doctores que el laboratorio científico promedio.

Lo primero que hay que hacer es crear paquetes que representen estos contextos y, a continuación, 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 hacer de forma incremental mientras hacemos otras cosas. Sin embargo, seguirá siendo necesario realizar pruebas para detectar cualquier fallo que se produzca al mover el código, especialmente si se utiliza un lenguaje de tipado dinámico en el que los IDE tienen más dificultades para realizar refactorizaciones. Con el tiempo, empezamos a ver qué código encaja bien y qué código sobra y no encaja realmente en ningún sitio. ¡Este código restante a menudo identificará contextos delimitados que podríamos haber pasado por alto!

Durante este proceso, también podemos utilizar 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 las dependencias entre paquetes de forma gráfica. 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), podemos investigar este problema e intentar resolverlo.

Este proceso puede llevar una tarde si se trata de una base de código pequeña, o varias semanas o meses si se trabaja con millones de líneas de código. Es posible que no sea necesario clasificar todo el código en paquetes orientados al dominio antes de dividir el primer servicio y, de hecho, puede resultar más valioso concentrar el esfuerzo en un solo lugar. No es necesario que se trate de un enfoque radical. 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 nuestra base de código organizada alrededor de estas costuras, ¿qué sigue?

## Las razones para dividir el monolito

Decidir que desea que un servicio o aplicación monolítico sea más pequeño es un buen comienzo. Pero le recomiendo encarecidamente que vaya trabajando en estos sistemas poco a poco. Un enfoque gradual le 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 hacer estallar todo, pero eso rara vez termina bien. Tiene mucho más sentido ir trabajando en ello poco a poco.

Entonces, si vamos a desmontar el monolito pieza por pieza, ¿por dónde deberíamos empezar? Ya tenemos nuestras costuras, pero ¿cuál deberíamos quitar primero? Es mejor pensar en dónde obtendrá el mayor beneficio de separar alguna parte de su base de código, en lugar de dividir las cosas solo por hacerlo. Consideraremos algunos factores que podrían ayudar a guiar nuestro cincel.

## Ritmo del cambio

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

## Estructura del equipo

El equipo de distribución de MusicCorp está dividido en dos regiones geográficas. Un equipo está en Londres y el otro en Hawái (¡para algunas personas es fácil!). Sería genial si pudiéramos dividir el código en el que más trabaja el equipo de Hawái, para que pueda asumir la responsabilidad total.

Exploraremos esta idea más a fondo en [el Capítulo 10](#).

## Seguridad

MusicCorp se ha sometido a una auditoría de seguridad y ha decidido reforzar la protección de la información confidencial. Actualmente, todo esto lo gestiona el código relacionado con las finanzas. Si dividimos este servicio, podemos proporcionar protecciones adicionales a este servicio individual en términos de supervisión, protección de datos en tránsito y protección de datos en reposo; ideas que analizaremos con más detalle en [el Capítulo 9](#).

## Tecnología

El equipo que se encarga de nuestro sistema de recomendaciones ha estado desarrollando nuevos algoritmos 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 independiente, sería fácil considerar la creación de una implementación alternativa con la que pudiéramos realizar pruebas.

## Dependencias enredadas

El otro punto a tener en cuenta cuando hayas identificado un par de uniones para separar es cuán enredado está ese código con el resto del sistema. Queremos extraer la unión de la que menos dependemos, si es posible. Si puedes ver las distintas uniones que has encontrado como un gráfico acíclico dirigido de dependencias (algo en lo que las herramientas de modelado de paquetes que mencioné antes son muy buenas), esto puede ayudarte a detectar las uniones que probablemente serán más difíciles de desenredar.

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

## La base de datos

Ya hemos hablado extensamente de los desafíos que supone utilizar bases de datos como método para integrar múltiples servicios. Como dejé bastante claro antes, ¡no soy partidario de ello! Esto significa que también tenemos que encontrar fisuras en nuestras bases de datos para poder separarlas de forma ordenada. Sin embargo, las bases de datos son un enigma complicado.

## Enfrentando el problema

El primer paso es echar un vistazo al código en sí y ver qué partes del mismo leen y escriben en la base de datos. Una práctica común es tener una capa de repositorio, respaldada por algún tipo de marco como Hibernate, para vincular el código a la base de datos, lo que facilita la asignación 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 delimitados; queremos hacer lo mismo con nuestro código de acceso a la base de datos. Esto puede requerir dividir la capa de repositorio en varias partes, como se muestra en [la Figura 5-1](#).

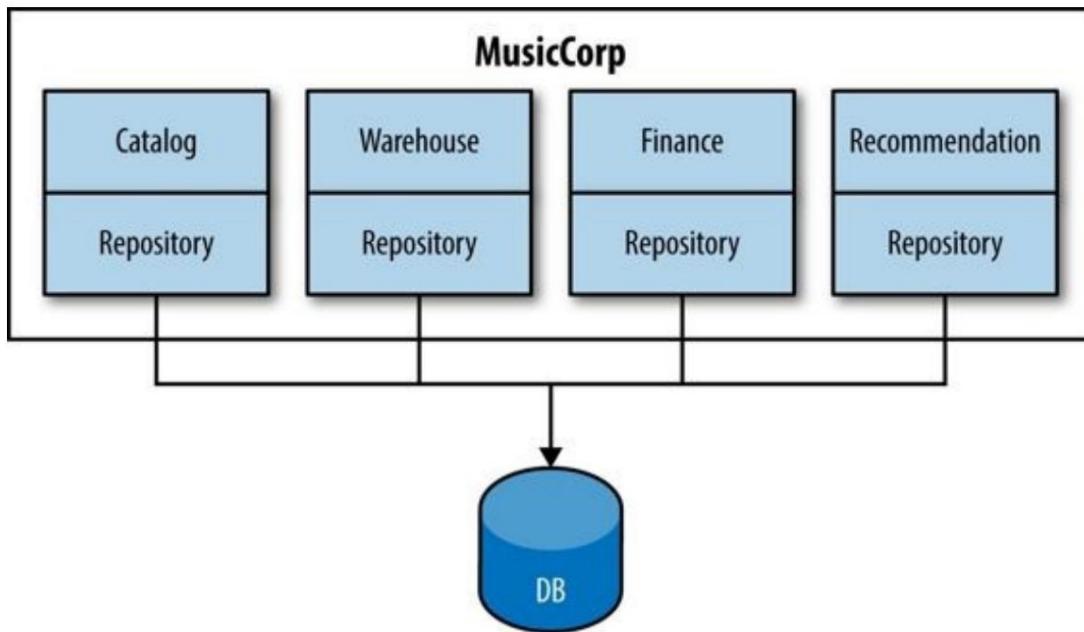


Figura 5-1. División de las capas de nuestro repositorio

Tener el código de mapeo de la base de datos ubicado dentro del código para un contexto determinado puede ayudarnos a entender qué partes de la base de datos se utilizan en qué fragmentos de código. Hibernate, por ejemplo, puede dejar esto muy claro si se utiliza algo como un archivo de mapeo por contexto delimitado.

Sin embargo, esto no nos da toda la historia. Por ejemplo, podemos saber que el código de finanzas utiliza la tabla de contabilidad y que el código de catálogo utiliza la tabla de artículos de línea, pero puede que no esté claro que la base de datos aplica una relación de clave externa desde la tabla de contabilidad hasta la tabla de artículos de línea. Para ver estas restricciones a nivel de base de datos, que pueden ser un obstáculo, necesitamos utilizar otra herramienta para visualizar los datos. Un buen lugar para comenzar es utilizar una herramienta como [SchemaSpy](#), disponible de forma gratuita. que puede generar representaciones gráficas de las relaciones entre tablas.

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

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

## Ejemplo: Romper relaciones de claves externas

En este ejemplo, nuestro código de catálogo utiliza una tabla de artículos de línea genérica para almacenar información sobre un álbum. Nuestro código de finanzas utiliza una tabla de libro mayor 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 nos está yendo. Queremos que los informes sean agradables y fáciles de leer, por lo 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 Grandes éxitos de Bruce Springsteen y ganamos \$1300"). Para hacer esto, nuestro código de informes en el paquete de finanzas llegará a la tabla de artículos de línea para extraer el título del SKU. También puede tener una restricción de clave externa del libro mayor a la tabla de artículos de línea, como vemos en [la Figura 5-2](#).

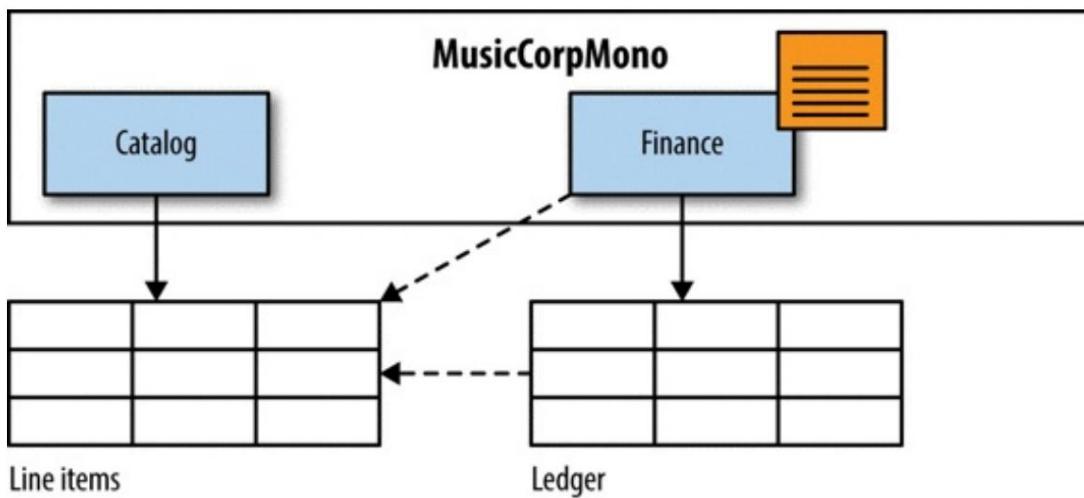


Figura 5-2. Relación de clave externa

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

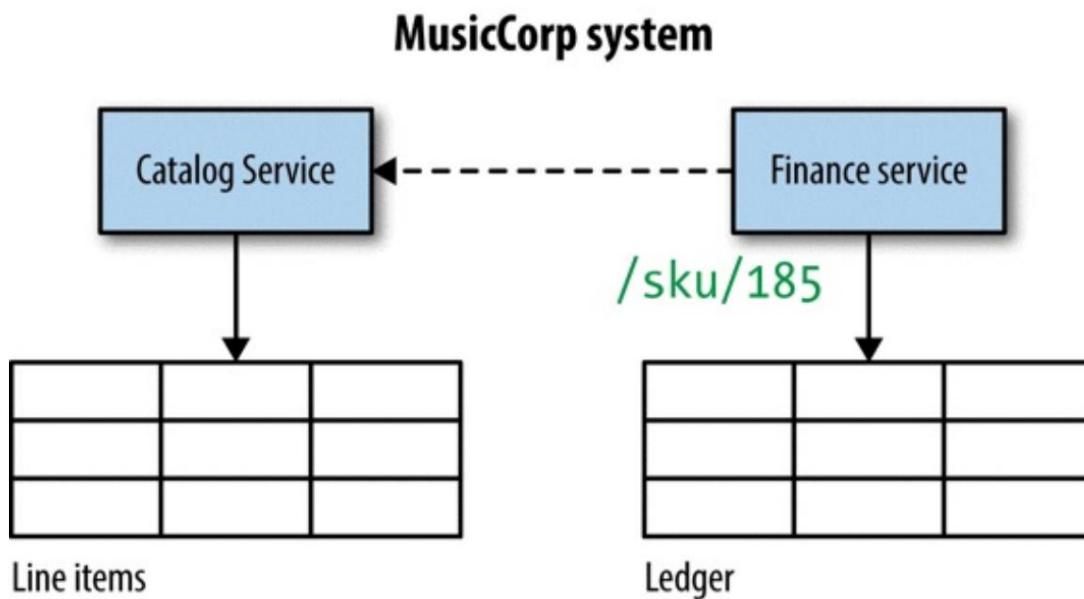


Figura 5-3. Relación de clave externa después de la eliminación

En este punto, queda claro que es posible que tengamos 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. Normalmente, ahora surgen preocupaciones sobre el rendimiento. Tengo una respuesta bastante fácil para ellas: ¿qué tan rápido necesita ser su sistema? ¿Y qué tan rápido es ahora? Si puede probar su rendimiento actual y sabe cómo se ve un buen rendimiento, entonces debería sentirse seguro de hacer un cambio. A veces, hacer que una cosa sea 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, la perdemos por completo. Esto se convierte en una restricción que ahora debemos gestionar en nuestros servicios resultantes en lugar de en el nivel de base de datos. Esto puede significar que necesitamos implementar nuestra propia comprobación de coherencia en todos los servicios o, de lo contrario, activar acciones para limpiar los datos relacionados. Si esto es necesario o no, a menudo no es una decisión que deba tomar un tecnólogo. Por ejemplo, si nuestro servicio de pedidos contiene una lista de identificadores para elementos del catálogo, ¿qué sucede si se elimina un elemento del catálogo y un pedido ahora hace referencia a un identificador de catálogo no válido? ¿Deberíamos permitirlo? Si lo hacemos, ¿cómo se representa esto en el pedido cuando se muestra? Si no lo hacemos, ¿cómo podemos comprobar que no se infringe? Estas son preguntas que necesitará que respondan las personas que definen cómo debe comportarse su sistema para sus usuarios.

## Ejemplo: Datos estáticos compartidos

He visto quizás tantos códigos de países almacenados en bases de datos (mostrados en [la Figura 5-4](#)) como clases `StringUtils` he escrito para proyectos Java internos. Esto parece implicar que planeamos cambiar los países que nuestro sistema admite con mucha más frecuencia de la que implementaremos código nuevo, pero sea cual sea la verdadera razón, estos ejemplos de datos estáticos compartidos almacenados en bases de datos surgen con mucha frecuencia. Entonces, ¿qué hacemos en nuestra tienda de música si todos nuestros servicios potenciales leen de la misma tabla de esta manera?

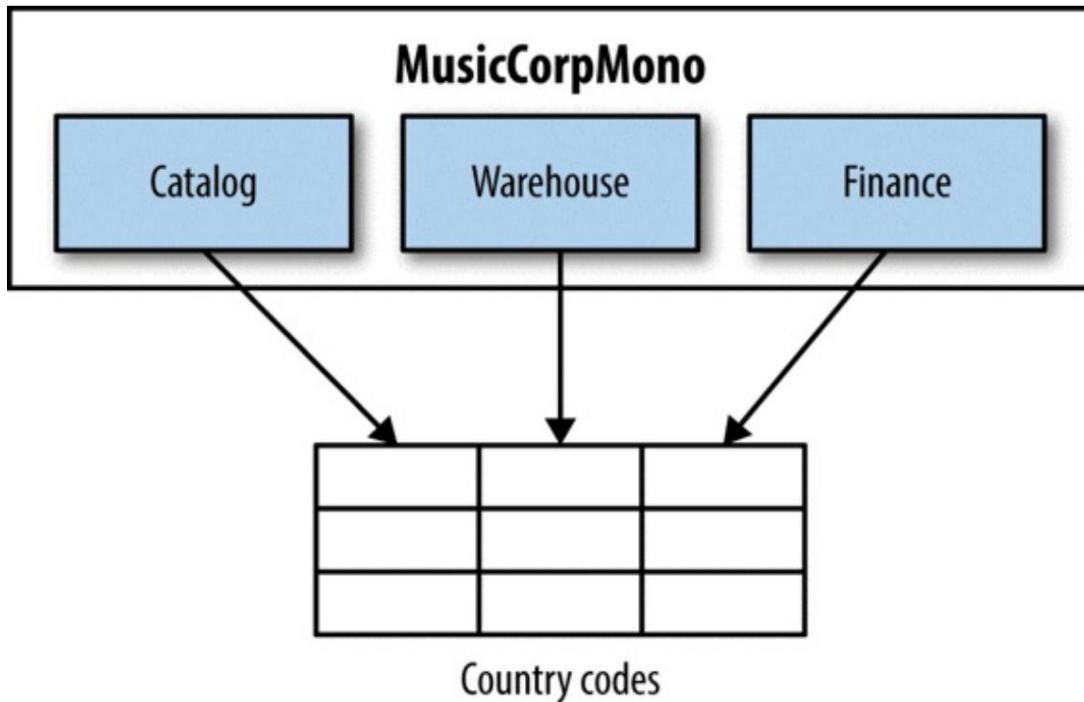


Figura 5-4. Códigos de países en la base de datos

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 duplique dentro de cada servicio. Esto lleva a un posible problema de coherencia, por supuesto: ¿qué sucede si actualizo una tabla para reflejar la creación de Newmantopia en la costa este de Australia, pero no otra?

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

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

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



## Ejemplo: Datos compartidos

Ahora, analicemos un ejemplo más complejo, pero que puede ser un problema común cuando se intenta analizar los sistemas: los datos mutables compartidos. Nuestro código financiero hace un seguimiento de los pagos que realizan los clientes por sus pedidos y también hace un seguimiento de 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 sucediendo con su cuenta. Para simplificar las cosas, hemos almacenado toda esta información en una tabla de registros de clientes bastante genérica, como se muestra en [la Figura 5-5](#).

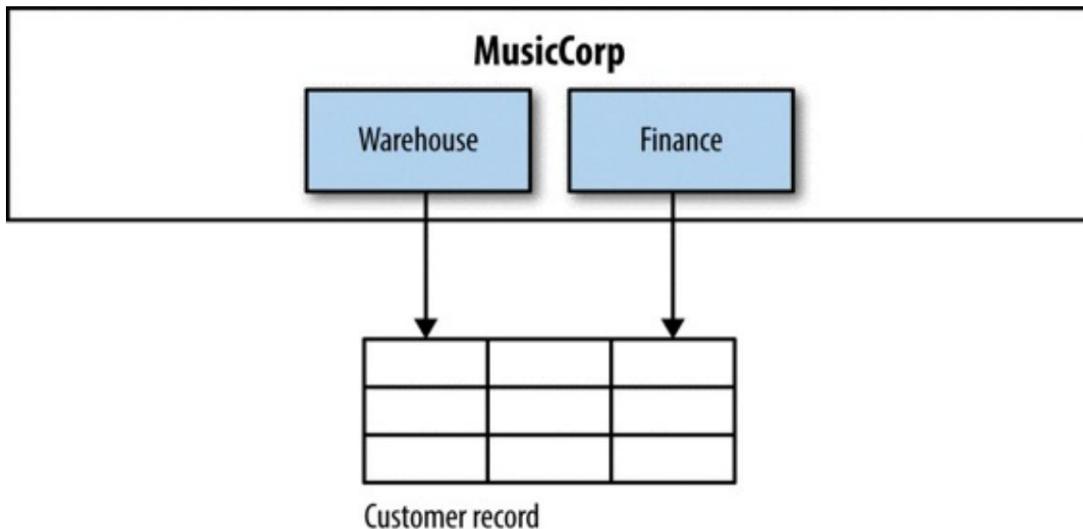


Figura 5-5. Acceso a los datos de los clientes: ¿nos estamos perdiendo algo?

Por lo tanto, tanto el código de finanzas como el de almacén escriben en la misma tabla y, probablemente, ocasionalmente la leen. ¿Cómo podemos distinguir esto? Lo que 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 hacer concreto el concepto abstracto actual 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. Si continuamos con esto hasta el final, es posible que ahora tengamos un servicio de atención al cliente distinto ([Figura 5-6](#)).

## MusicCorp system

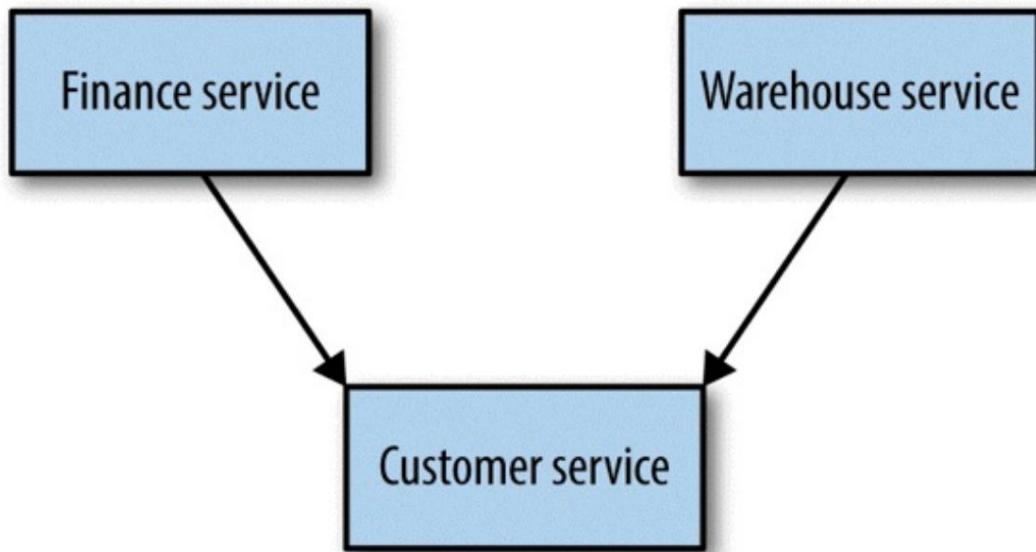


Figura 5-6. Reconocimiento del contexto delimitado del cliente

## Ejemplo: Tablas compartidas

La figura 5-7 muestra nuestro último ejemplo. Nuestro catálogo debe almacenar el nombre y el precio de los registros que vendemos, y el almacén debe mantener un registro electrónico del inventario. Decidimos mantener estas dos cosas en el mismo lugar en una tabla de artículos de línea genérica. Antes, con todo el código fusionado, no estaba claro que en realidad estuvíramos fusionando cuestiones, pero ahora podemos ver que, de hecho, tenemos dos conceptos separados que podrían almacenarse de forma diferente.

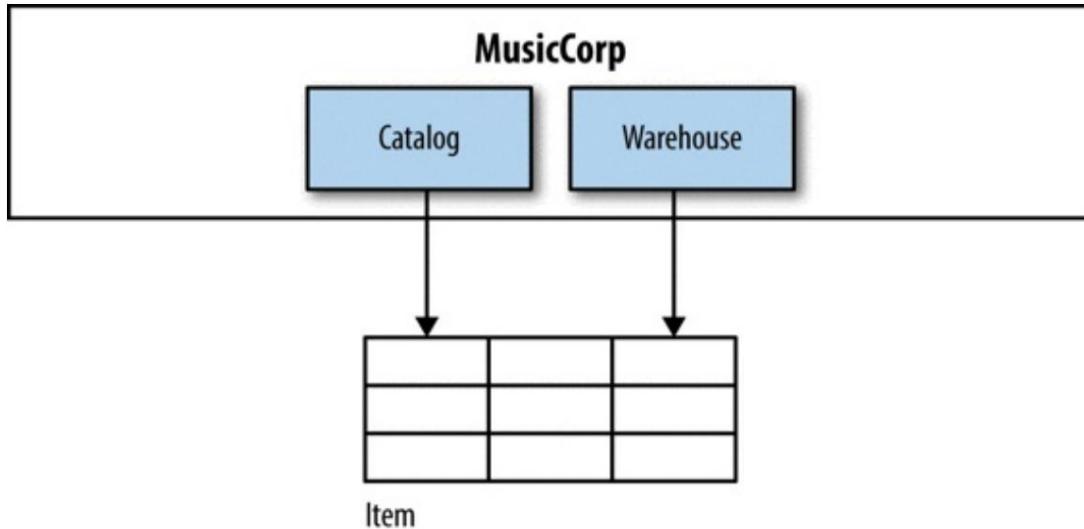


Figura 5-7. Tablas compartidas entre diferentes contextos

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

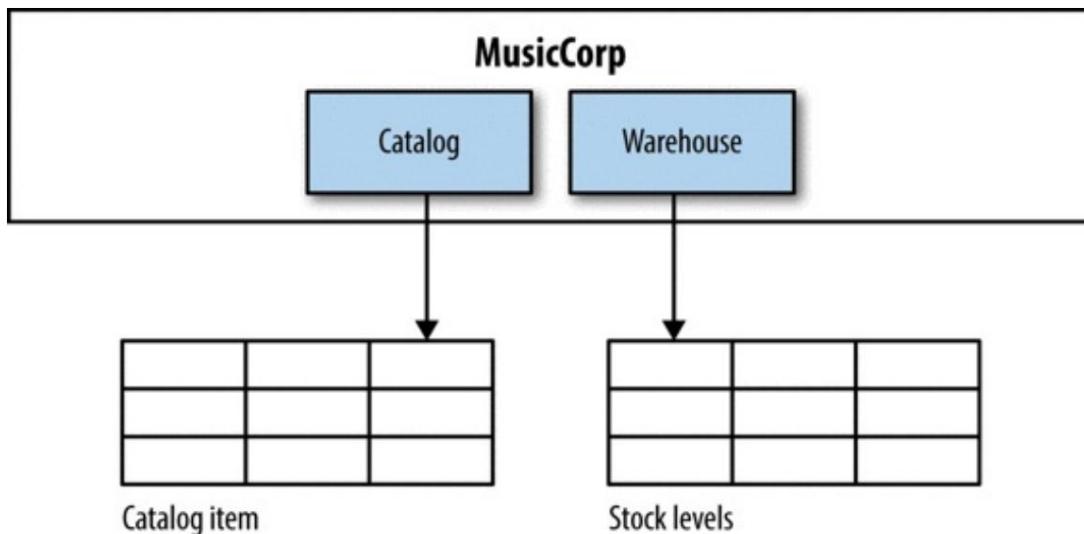


Figura 5-8. Desmontaje de la tabla compartida

## Refactorización de bases de datos

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

## Preparando el descanso

Así que hemos encontrado costuras en nuestro código de aplicación, agrupándolo alrededor de contextos delimitados.

Hemos utilizado esto para identificar las uniones en la base de datos y hemos hecho todo lo posible para separarlas. ¿Qué sigue? ¿Haremos un lanzamiento espectacular, pasando de un servicio monolítico con un solo esquema a dos servicios, cada uno con su propio esquema? En realidad, recomendaría que separe el esquema pero mantenga el servicio junto antes de dividir el código de la aplicación en microservicios separados, como se muestra en la Figura 5-9.

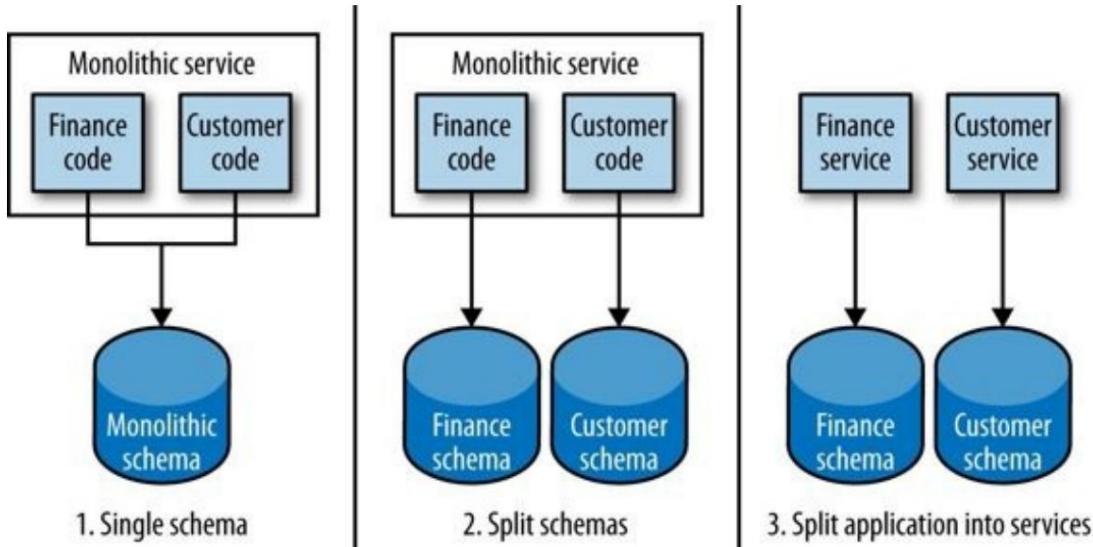


Figura 5-9. Preparación de una separación de servicios

Con un esquema independiente, potencialmente aumentaremos 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 declaración 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 mantener el código de la aplicación junto, nos damos la capacidad de revertir nuestros cambios o continuar ajustando 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 útiles. Nos permiten decir que estos eventos ocurren todos juntos o que ninguno de ellos ocurre. Son muy útiles cuando insertamos 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 múltiples actividades diferentes que llevan nuestro sistema de un estado consistente a otro: todo funciona o nada cambia.

Las transacciones no se aplican únicamente a las bases de datos, aunque las utilizamos con más frecuencia en ese contexto. Los intermediarios de mensajes, por ejemplo, permiten desde hace mucho tiempo publicar y recibir mensajes también dentro de las transacciones.

Con un esquema monolítico, todas nuestras creaciones o actualizaciones probablemente se realizarán dentro de un único límite transaccional. Cuando dividimos nuestras bases de datos, perdemos la seguridad que nos brinda tener una única transacción. Consideraremos un ejemplo sencillo 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 del almacén para que sepa que hay un pedido que se debe preparar para su envío. Hemos llegado al punto de agrupar nuestro código de aplicación en paquetes separados y también hemos separado las partes del cliente y del almacén del esquema lo suficientemente bien como para que estemos listos para ponerlas en sus propios esquemas antes de separar el código de aplicación.

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

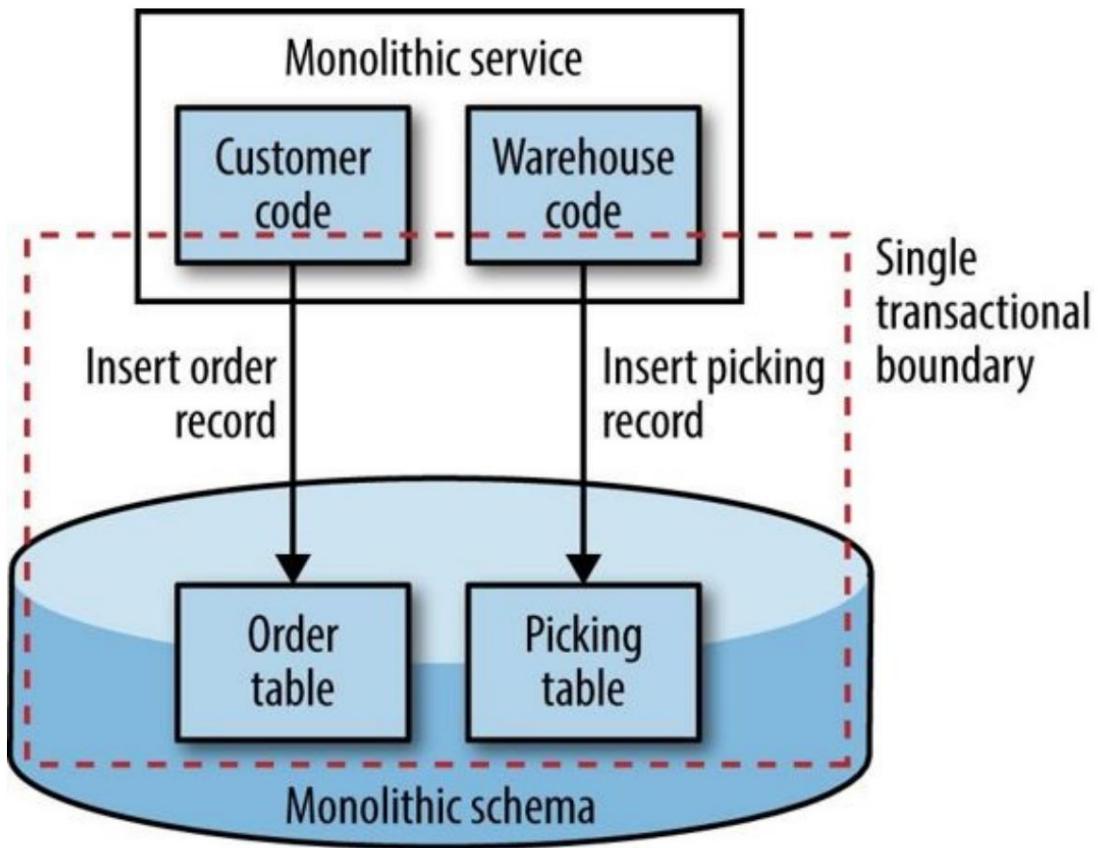


Figura 5-10. Actualización de dos tablas en una sola transacción

Pero si hemos dividido 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, podemos detener todo claramente, dejándonos en un estado consistente. Pero ¿qué sucede cuando la inserción en la tabla de pedidos funciona, pero la inserción en la tabla de selección falla?

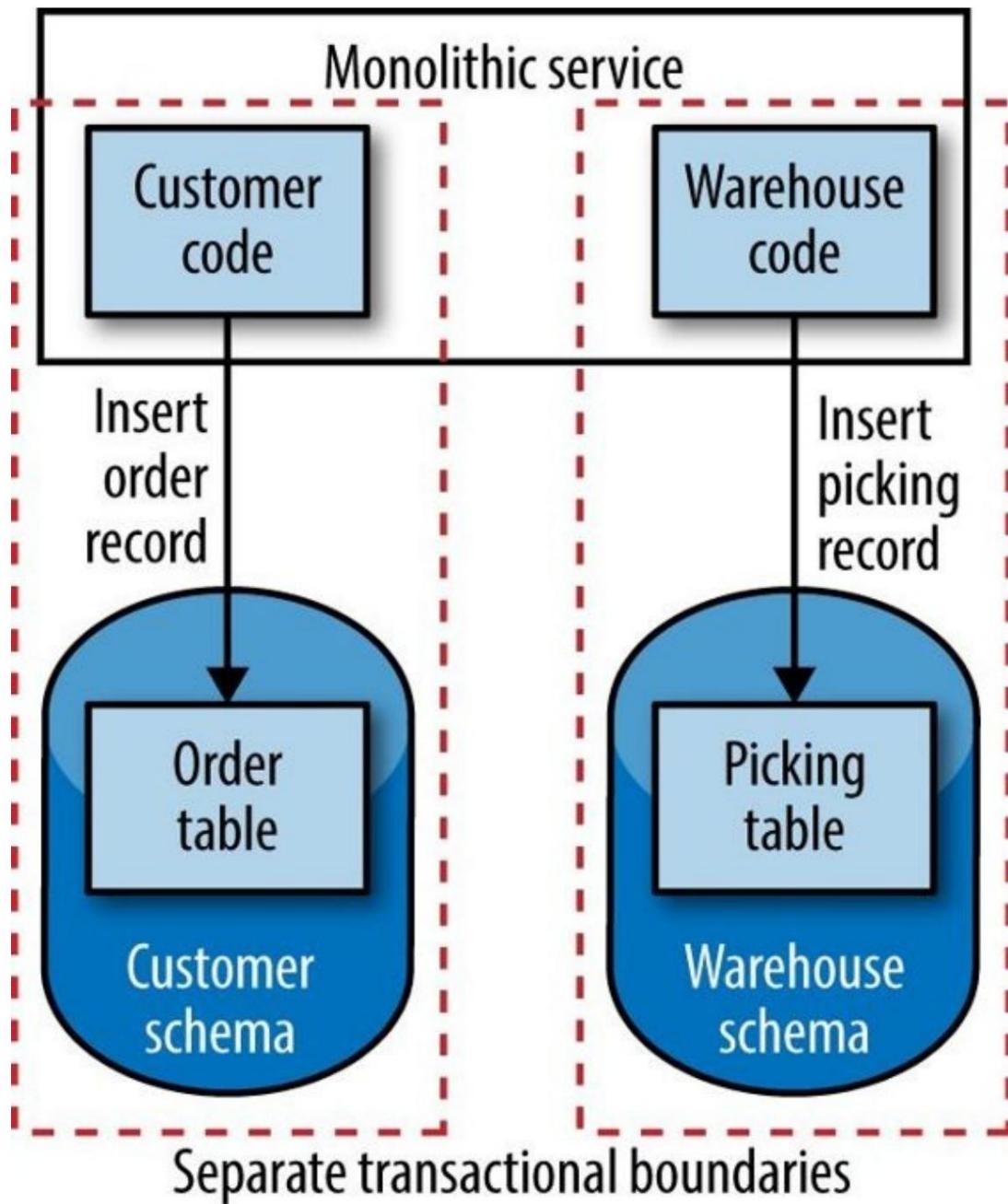


Figura 5-11. Superación de límites transaccionales para una sola operación

## Inténtalo de nuevo más tarde

El hecho de que el pedido se haya capturado y realizado puede ser suficiente para nosotros y podemos decidir volver a intentar la inserción en la tabla 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 solucionaría.

En muchos sentidos, se trata de otra forma de lo que se denomina consistencia eventual. En lugar de utilizar un límite transaccional para garantizar que el sistema se encuentre en un estado consistente cuando se complete la transacción, aceptamos que el sistema alcanzará por sí mismo un estado consistente en algún momento en el futuro. Este enfoque es especialmente útil con operaciones comerciales que pueden durar mucho tiempo. Analizaremos esta idea con más profundidad en el [Capítulo 11](#), cuando tratemos los patrones de escalado.

## Cancelar toda la operación

Otra opción es rechazar toda la operación. En este caso, tenemos que poner el sistema de nuevo en un estado consistente. La tabla de selección es fácil, ya que esa inserción falló, pero tenemos una transacción confirmada en la tabla de pedidos. Necesitamos deshacer esto. Lo que tenemos que hacer es emitir una transacción de compensación, iniciando una nueva transacción para deshacer lo que acaba de suceder. Para nosotros, eso podría ser algo tan simple como emitir una declaración `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 lo que podríamos hacer cuando dividimos el código de la aplicación. ¿La lógica para manejar la transacción de compensación reside en el servicio de atención 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 ninguna instrucción de selección correspondiente. En esta situación, tendría que volver a intentar la transacción de compensación o permitir que algún proceso de back-end corrija la inconsistencia 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, pensemos en lo que sucede si no tenemos una o dos operaciones que queremos que sean consistentes, sino tres, cuatro o cinco. La gestión de transacciones compensatorias para cada modo de falla se vuelve bastante difícil de comprender, y más aún de implementar.

## Transacciones distribuidas

Una alternativa a la orquestación manual de transacciones compensatorias es utilizar una transacción distribuida. Las transacciones distribuidas intentan abarcar múltiples transacciones dentro de ellas, utilizando un 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 consistente, 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 gestionar transacciones distribuidas (especialmente transacciones de corta duración, como en el caso de gestionar el pedido de nuestro cliente) es utilizar una confirmación en dos fases. En una confirmación en 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 transacciones si cree que su transacción local puede seguir adelante. Si el administrador de transacciones obtiene un voto afirmativo de todos los participantes, entonces les dice a todos que sigan adelante y realicen sus confirmaciones. 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 pueden continuar. Esto significa que somos vulnerables a 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 sucede si una confirmación falla después de la votación. 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 tenemos que asumir que se confirmará. Las cohortes necesitan una forma de hacer que esta confirmación 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 implica bloqueos, es decir, las transacciones pendientes pueden tener bloqueos sobre los recursos. Los bloqueos sobre los recursos pueden generar contención, lo que dificulta mucho la ampliación de los sistemas, especialmente en el contexto de los sistemas distribuidos.

Las transacciones distribuidas se han implementado 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 en la misma transacción general. Es difícil hacer que los distintos algoritmos funcionen correctamente, por lo que le sugiero que evite intentar crear uno propio. En cambio, investigue mucho sobre este tema si parece que esta es la ruta que desea tomar y vea si puede usar una implementación existente.

## Entonces, ¿qué hacer?

Todas estas soluciones añaden complejidad. Como puede ver, las transacciones distribuidas son difíciles de ejecutar correctamente y pueden inhibir la escalabilidad. Los sistemas que finalmente convergen mediante una lógica de reintento compensatorio pueden ser más difíciles de analizar y pueden necesitar otro comportamiento compensatorio para corregir las inconsistencias en los datos.

Cuando se encuentre con operaciones comerciales que actualmente se llevan a cabo dentro de una sola transacción, pregúntese si realmente es necesario hacerlo. ¿Pueden realizarse en transacciones locales diferentes y depender del concepto de coherencia final? Estos sistemas son mucho más fáciles de construir y escalar (hablaremos más sobre esto en [el Capítulo 11](#)).

Si encuentra un estado que realmente quiere mantenerse consistente, haga todo lo posible para evitar dividirlo en primer lugar. Inténtelo con todas sus fuerzas. Si realmente necesita seguir adelante con la división, piense en pasar de una visión puramente técnica del proceso (por ejemplo, una transacción de base de datos) y realmente crear un concepto concreto para representar la transacción en sí. Esto le brinda un punto de referencia, o un gancho, en el que 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 concentrar toda la lógica en torno al procesamiento del pedido de principio a fin (y el manejo de las excepciones).

# Informes

Como ya hemos visto, al dividir un servicio en partes más pequeñas, también es posible que debamos dividir cómo y dónde se almacenan los datos. Sin embargo, esto genera un problema cuando se trata de un caso de uso vital y común: los informes.

Un cambio de arquitectura tan fundamental como pasar a una arquitectura de microservicios provocará muchos trastornos, pero no significa que tengamos que abandonar todo lo que hacemos. El público al que van dirigidos nuestros sistemas de informes son usuarios como cualquier otro, y debemos tener en cuenta sus necesidades. Sería arrogante cambiar radicalmente nuestra arquitectura y simplemente pedirles que se adapten. Si bien no estoy sugiriendo que el espacio de los informes no esté maduro para sufrir trastornos (ciertamente lo está), es valioso determinar primero cómo trabajar con los procesos existentes. A veces tenemos que elegir nuestras batallas.

## La base de datos de informes

Los informes suelen tener que agrupar datos de varias partes de nuestra organización para generar resultados útiles. Por ejemplo, es posible que queramos enriquecer los datos de nuestro libro mayor con descripciones de lo que se vendió, que obtenemos de un catálogo. O es posible que queramos observar el comportamiento de compra de clientes específicos y 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 generar informes sobre toda la información es bastante fácil, ya que podemos unir los datos mediante consultas SQL o similares. Normalmente, no ejecutamos 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 generación de informes se bloquean en una réplica de lectura, como se muestra en [la Figura 5-12](#).

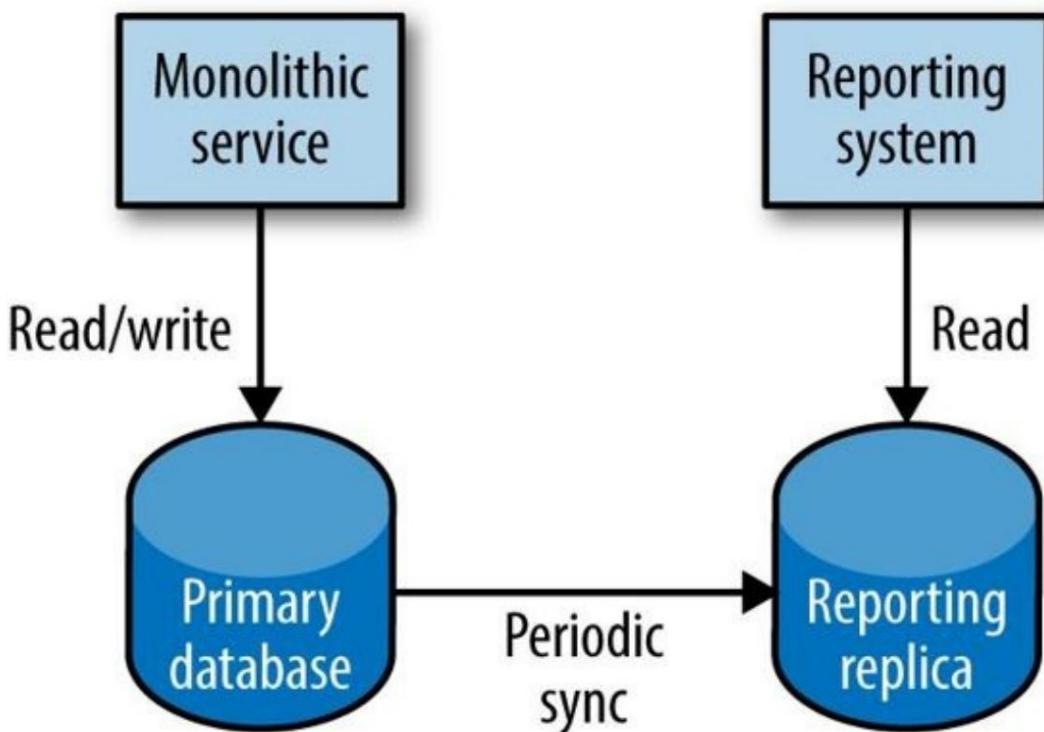


Figura 5-12. Replicación de lectura estándar

Con este enfoque tenemos una ventaja considerable: todos los datos ya están en un solo lugar, por lo que podemos utilizar herramientas bastante sencillas para consultarlos. Pero también hay un par de desventajas con este enfoque. En primer lugar, 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 gestionarse con cuidado. En realidad, este es otro impedimento que reduce las posibilidades de que alguien 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 cada caso de uso (respaldo del sistema en vivo o del sistema de informes). Algunas bases de datos nos permiten hacer

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

Lo que suele ocurrir 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 y no es excelente para ninguno de los propósitos.

Por último, las opciones de bases de datos disponibles han aumentado 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 sucede si los datos de nuestra aplicación se modelan mejor como un gráfico, como en Neo4j? ¿O qué sucede si preferimos utilizar un almacén de documentos como MongoDB? Del mismo modo, ¿qué sucede si queremos explorar el uso de una base de datos orientada a columnas como Cassandra para nuestro sistema de generación de informes, lo que facilita mucho la escalabilidad para volúmenes mayores? 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

No es perfecto, pero funciona (en su mayor parte). Ahora bien, si nuestra información está almacenada en varios sistemas diferentes, ¿qué hacemos? ¿Hay alguna forma de 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 varias alternativas viables a este enfoque. La solución que tenga más sentido para usted dependerá de varios factores, pero exploraremos algunas opciones diferentes que he visto en la práctica.

## Recuperación de datos mediante llamadas de servicio

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

Sin embargo, este enfoque fracasa rápidamente en los casos de uso que requieren mayores volúmenes de datos. Imaginemos 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, analizando diversas tendencias en el comportamiento de los clientes y cómo esto ha afectado a los ingresos. Necesitamos extraer grandes volúmenes de datos de al menos 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 han 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 de 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 una determinada manera, y en este caso, proporcionar una interfaz SQL es la forma más rápida de garantizar que la cadena de herramientas de generación de informes sea lo más fácil de integrar posible. Por supuesto, todavía podríamos utilizar este enfoque para extraer datos periódicamente a una base de datos SQL, pero esto aún nos presenta algunos desafíos.

Uno de los desafíos clave es que las API expuestas por los distintos microservicios pueden no estar diseñadas para casos de uso de informes. Por ejemplo, un servicio de atención al cliente puede permitirnos encontrar un cliente por un ID o buscar un cliente por varios campos, pero no necesariamente expondría una API para recuperar todos los clientes. Esto podría generar que se realicen muchas llamadas para recuperar todos los datos; por ejemplo, tener que iterar a través de 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 implica que accedemos a la cola larga de datos. Esto significa que es posible que solicitemos recursos que nadie más haya solicitado antes (o al menos no durante un tiempo suficientemente largo), lo que da como resultado una falla de caché potencialmente costosa.

Puede resolver esto exponiendo API por lotes para facilitar la generación de informes. Por ejemplo, nuestro servicio de atención al cliente podría permitirle pasarle una lista de identificaciones de clientes para recuperarlas en lotes, o incluso podría exponer una interfaz que le permita buscar entre 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 punto final de recurso BatchCustomerExport . El sistema que realiza la llamada publicaría un BatchRequest, 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 se procesó. El sistema que realiza la llamada podría entonces sondear el recurso esperando hasta que recupere un estado 201 Creado, indicando que la solicitud se ha cumplido, y luego el sistema que realiza la llamada podría ir a buscar los datos. Esto permitiría exportar archivos de datos potencialmente grandes sin la sobrecarga de enviarlos a través de HTTP; en su lugar, 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, no soy tan partidario de ello para los sistemas de informes, ya que creo que hay otras soluciones potencialmente más simples que pueden escalar de manera más efectiva cuando se trata de necesidades de informes tradicionales.

## Bombas de datos

En lugar de que el sistema de informes extraiga los datos, podríamos enviarlos 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 los datos y los envíe a una base de datos de informes, como se muestra en [la Figura 5-13](#).

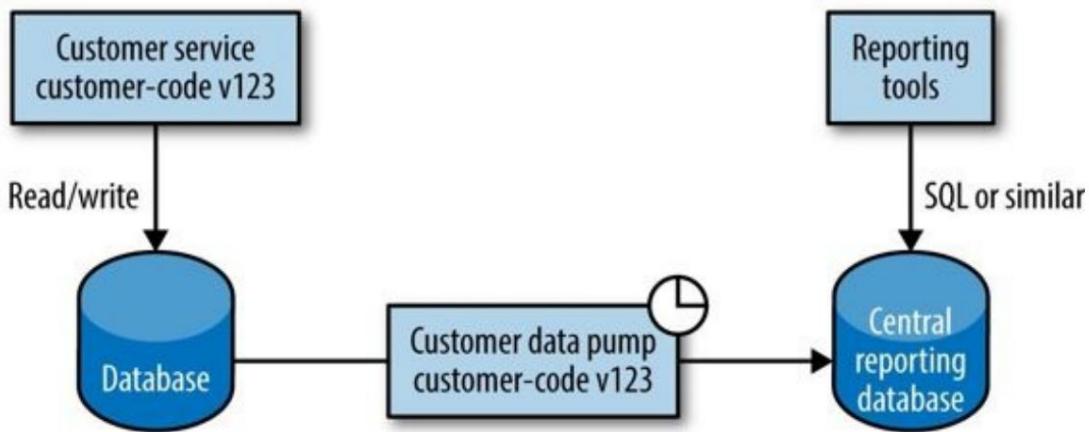


Figura 5-13. Uso de una bomba de datos para enviar periódicamente datos a una base de datos de informes central

En este punto, usted estará diciendo: "Pero Sam, ¡dijiste que tener muchos programas integrados en la misma base de datos es una mala idea!". Al menos espero que esté diciendo eso, dada la firmeza con la que planteé el punto anteriormente. Este enfoque, si se implementa correctamente, es una excepción notable, donde las desventajas del acoplamiento se mitigan con creces al facilitar la generación 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 informes. El trabajo de la bomba es mapear una a la otra. Intentamos reducir los problemas con el acoplamiento al esquema del servicio haciendo que el mismo equipo que administra el servicio también administre la bomba. De hecho, sugeriría que controle las versiones de estos elementos juntos y que cree compilaciones de la bomba de datos como un artefacto adicional como parte de la compilación del servicio en sí, con el supuesto de que siempre que implemente uno de ellos, implemente ambos. 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 se mantiene, pero tenemos que tratarlo como una API publicada que es difícil de cambiar. Algunas bases de datos nos brindan técnicas con las que podemos 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, utilizando cosas como

Vistas materializadas para crear la vista agregada. De esa manera, 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 haya elegido para los informes.

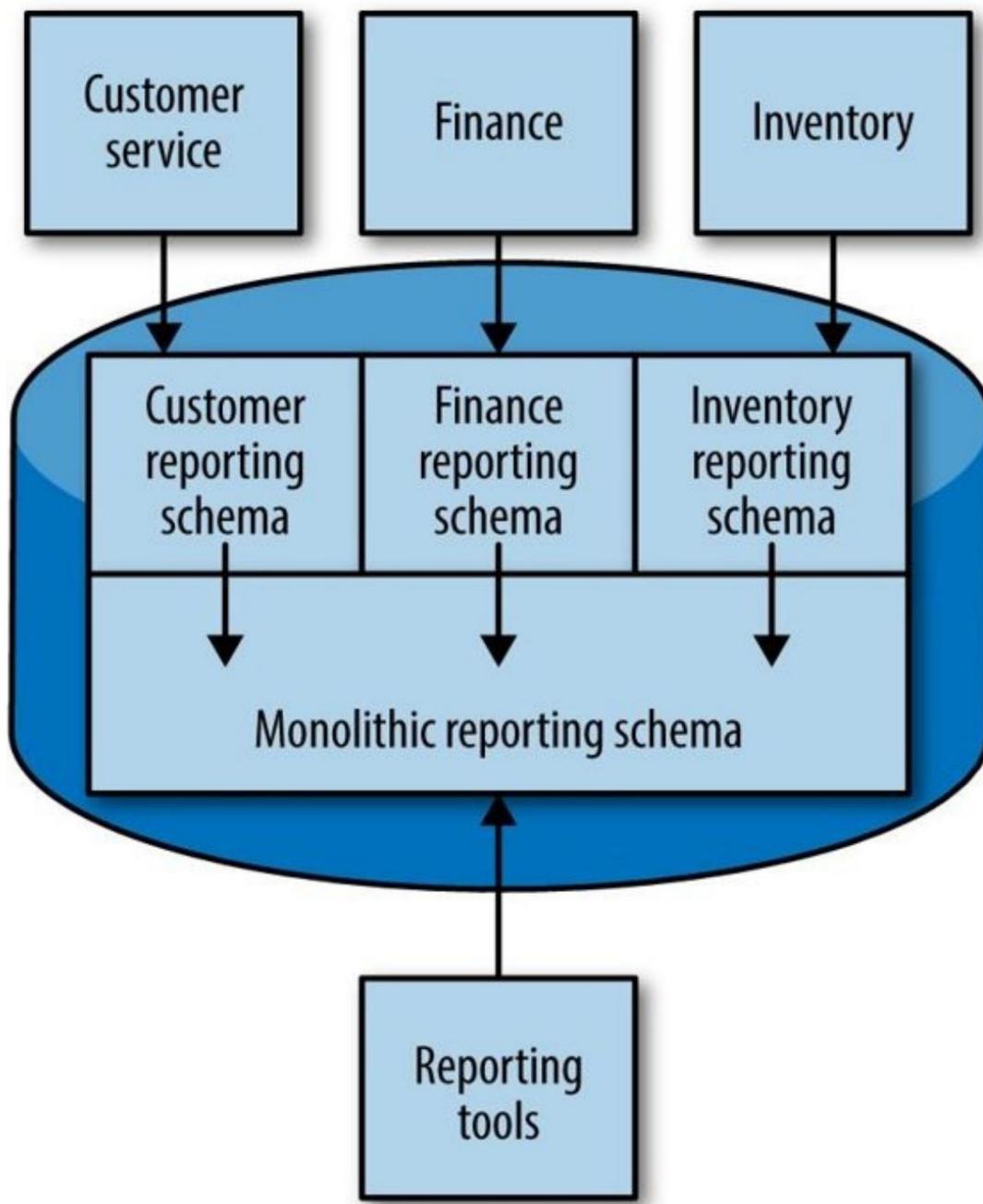


Figura 5-14. Utilización de vistas materializadas para formar un único esquema de informes monolítico

En este caso, por supuesto, la complejidad de la integración se profundiza más en el esquema y dependerá de las capacidades de la base de datos para que dicha configuración sea eficaz. Si bien creo que las bombas de datos en general son una sugerencia sensata y viable, no estoy tan convencido de que valga la pena la complejidad de un esquema segmentado, especialmente considerando los desafíos que implica gestionar los cambios en la base de datos.

## Destinos alternativos

En un proyecto en el que participé, utilizamos una serie de bombas de datos para completar archivos JSON en AWS S3, lo que efectivamente hizo que S3 se hiciera pasar por un gigantesco almacén de datos.

Este enfoque funcionó muy bien hasta que tuvimos que escalar nuestra solución y, al momento de escribir esto, estamos buscando cambiar estas bombas para completar 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 que los microservicios 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.

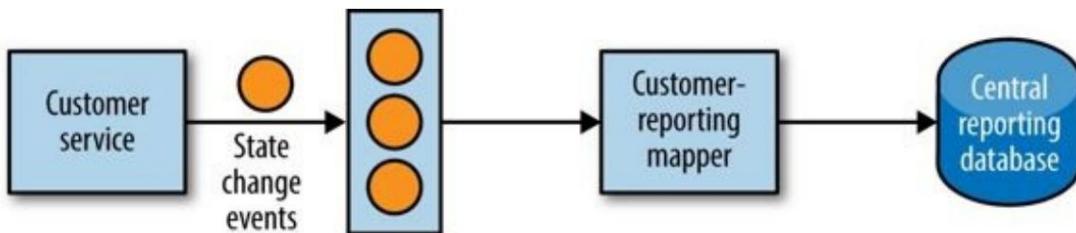


Figura 5-15. Una bomba de datos de eventos que utiliza eventos de cambio de estado para llenar una base de datos de informes

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

En cambio, simplemente 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 resulta más fácil ser más inteligentes en cuanto a los datos que enviamos a nuestro almacén de informes central; 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 cronograma regular como con el bombeo de datos.

Además, si almacenamos los eventos que ya se han procesado, podemos procesar los nuevos eventos a medida que llegan, suponiendo que los eventos anteriores ya se han asignado al 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 gestionarla nosotros mismos, mientras que la naturaleza fundamentalmente temporal del flujo de eventos ( $x$  sucede en la marca de tiempo  $y$ ) nos ayuda enormemente.

Como nuestro flujo de datos de eventos está menos acoplado a los aspectos internos del servicio, también es más fácil considerar que esto lo administre un grupo separado del equipo que se encarga del microservicio en sí. Siempre que la naturaleza de nuestro flujo de eventos no vincule demasiado a los suscriptores con los cambios en el servicio, este mapeador de eventos puede evolucionar independientemente del servicio al que se suscribe.

Las principales desventajas de este enfoque son que toda la información requerida debe transmitirse como eventos y es posible que no sea tan escalable como un bombeo de datos para volúmenes mayores de datos que tiene el beneficio de operar directamente a nivel de base de datos. No obstante, el acoplamiento más flexible y los datos más actualizados 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 copia de seguridad existentes y también resuelve algunos problemas de escala con los que Netflix tiene que lidiar. En cierto modo, se puede considerar que se trata de un caso especial de bombeo de datos, pero parecía una solución tan interesante que merece ser incluida.

Netflix ha decidido utilizar Cassandra como almacén de respaldo para sus servicios, de los cuales hay muchos. Netflix ha invertido un tiempo significativo en crear herramientas para que sea fácil trabajar con Cassandra, gran parte de las cuales la compañía 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 respaldados adecuadamente. Para respaldar los datos de Cassandra, el enfoque estándar es hacer una copia de los archivos de datos que los respaldan y almacenarlos en un lugar seguro. Netflix almacena estos archivos, conocidos como SSTables, en el almacén de objetos S3 de Amazon, que proporciona importantes garantías de durabilidad de los datos.

Netflix necesita generar informes sobre todos estos datos, pero dada la escala involucrada, este es un desafío nada trivial. Su enfoque es utilizar Hadoop, que utiliza la copia de seguridad SSTable como fuente de sus trabajos. Al final, Netflix terminó implementando un pipeline capaz de procesar grandes cantidades de datos utilizando este enfoque, que luego publicó 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 posible que el uso de un enfoque similar (es decir, el uso de mapeadores que funcionan a partir de copias de seguridad) funcione también en otros contextos. Y si ya usa Cassandra, ¡Netflix ya ha hecho gran parte del trabajo por usted!

## Hacia el tiempo real

Muchos de los patrones que se han descrito anteriormente son formas diferentes de obtener una gran cantidad de datos de muchos lugares diferentes en un solo lugar. Pero, ¿acaso la idea de que todos nuestros informes se realizarán desde un solo lugar realmente se sostiene hoy en día? Tenemos paneles de control, alertas, informes financieros, análisis de usuarios; todos estos casos de uso tienen diferentes tolerancias en cuanto a precisión y puntualidad, lo que puede dar lugar a que se apliquen diferentes opciones técnicas. Como detallaré en [el Capítulo 8](#), nos estamos moviendo cada vez más hacia sistemas de eventos genéricos capaces de enviar 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 factores clave es comprender el impacto de cada modificación 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 cometerlos. Podemos cometer errores (y lo haremos), y debemos aceptarlo. Sin embargo, lo que también deberíamos hacer es comprender cuál es la mejor manera de mitigar los costos de esos errores.

Como hemos visto, el costo que implica mover código dentro de una base de código es bastante pequeño. Contamos con muchas herramientas que nos ayudan y, si causamos un problema, la solución suele ser rápida. Sin embargo, dividir una base de datos es mucho más trabajo y deshacer un cambio en la base de datos es igual de complejo. Asimismo, desenredar una integración demasiado acoplada entre servicios o tener que reescribir por completo una API que utilizan varios consumidores puede ser una tarea considerable. El alto costo del cambio significa que estas operaciones son cada vez más riesgosas. ¿Cómo podemos gestionar este riesgo? Mi enfoque es tratar de cometer errores donde el impacto sea menor.

Suelo pensar mucho en el lugar donde el costo del cambio y el costo de los errores es lo más bajo posible: la pizarra. Esbozo el diseño que propongo. Ve qué sucede cuando ejecuta casos de uso que se ajusten a lo que cree que serán los límites de su servicio.

Por ejemplo, en el caso de nuestra tienda de música, imaginemos qué sucede cuando un cliente busca un disco, se registra en el sitio web o compra un álbum. ¿Qué llamadas se realizan? ¿Comienza a ver referencias circulares extrañas? ¿Ve dos servicios que hablan demasiado, lo que podría indicar que deberían ser una sola cosa?

Una técnica muy útil en este caso es adaptar un enfoque que se suele enseñar para el diseño de sistemas orientados a objetos: las tarjetas de colaboración entre clases y responsabilidades (CRC). En las tarjetas CRC, se escribe en una tarjeta el nombre de la clase, cuáles son sus responsabilidades y con quién colabora. Cuando trabajo en un diseño propuesto, para cada servicio enumero sus responsabilidades en términos de las capacidades que proporciona, con los colaboradores especificados en el diagrama. A medida que se trabaja con más casos de uso, se empieza a tener una idea de si todo esto encaja correctamente.

## Comprender las causas fundamentales

Hemos analizado cómo dividir servicios más grandes en servicios más pequeños, pero ¿por qué estos servicios crecieron tanto en primer lugar? Lo primero que hay que entender es que hacer crecer un servicio hasta el punto de que sea necesario dividirlo está perfectamente bien. Queremos que la arquitectura de nuestro sistema cambie con el tiempo de forma incremental. La clave es saber que es necesario dividirlo antes de que la división resulte demasiado costosa.

Pero en la práctica, muchos de nosotros habremos visto cómo los servicios han crecido mucho más allá del punto de lo sensato. A pesar de saber que sería más fácil gestionar un conjunto más pequeño de servicios que la monstruosidad que tenemos actualmente, seguimos adelante con el crecimiento de la bestia. ¿Por qué?

Parte del problema es saber por dónde empezar, y espero que este capítulo haya sido de ayuda. Pero otro desafío es el costo asociado con la división de servicios. Encontrar un lugar para ejecutar el servicio, crear una nueva pila de servicios, etc., son tareas no 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 servicios livianos puede reducir el costo asociado con la creación del nuevo servicio. Dar a las personas acceso a máquinas virtuales de provisión de autoservicio o incluso hacer que una plataforma como servicio (PaaS) esté disponible facilitará la provisión 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 buscando puntos de unión a lo largo de los cuales puedan surgir límites entre los servicios, y este puede ser un enfoque incremental. Si nos volvemos buenos en encontrar estos puntos de unión y trabajamos para reducir el costo de dividir los servicios en primer lugar, podemos seguir creciendo y evolucionando nuestros sistemas para cumplir con cualquier requisito que surja en el futuro. Como puede ver, parte de este trabajo puede ser minucioso, pero el hecho mismo de que se pueda hacer de manera incremental significa que no hay necesidad de temerle a este trabajo.

Ahora podemos dividir nuestros servicios, pero también hemos introducido algunos problemas nuevos. ¡Tenemos muchas más piezas móviles que poner en producción ahora! A continuación, nos sumergiremos en el mundo de la implementación.

# Capítulo 6. Despliegue

---

Implementar una aplicación monolítica es un proceso bastante sencillo. Los microservicios, con su interdependencia, son un asunto completamente distinto. Si no abordas la implementación de la forma correcta, es una de esas áreas en las que la complejidad puede hacerte la vida imposible. En este capítulo, veremos algunas técnicas y tecnologías que pueden ayudarnos a la hora de implementar microservicios en arquitecturas de grano fino.

Sin embargo, comenzaremos analizando la integración continua y la entrega continua. Estos conceptos, relacionados pero diferentes, nos ayudarán a dar forma a las demás decisiones que tomaremos cuando pensemos en qué construir, cómo construirlo y cómo implementarlo.

## Una breve introducción a la integración continua

La integración continua (CI) ya existe desde hace varios años. Sin embargo, vale la pena dedicar un poco de tiempo a repasar los conceptos básicos, ya que, especialmente cuando pensamos en la correlación entre microservicios, compilaciones y repositorios de control de versiones, hay algunas opciones diferentes que se deben considerar.

Con CI, el objetivo principal es mantener a todos sincronizados entre sí, lo que logramos al asegurarnos de que el código recién incorporado se integre correctamente con el código existente. Para ello, un servidor de CI detecta que el código se ha confirmado, lo verifica y realiza algunas verificaciones, como asegurarse de que el código se compila y de que las pruebas pasan.

Como parte de este proceso, a menudo creamos artefactos que se utilizan para una validación posterior, como la implementación de un servicio en ejecución para ejecutar pruebas en él. Lo ideal es crear estos artefactos una sola vez y utilizarlos para todas las implementaciones de esa versión del código.

Esto se hace 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 puedan reutilizar, los colocamos en un repositorio de algún tipo, ya sea proporcionado por la propia herramienta de CI o en un sistema independiente.

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

La CI tiene una serie de ventajas. Obtenemos un cierto nivel de retroalimentación rápida sobre 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 un cierto nivel de trazabilidad desde un artefacto implementado hasta el código y, según las capacidades de la propia herramienta de CI, podemos ver qué pruebas se ejecutaron en el código y el artefacto también. Es por estas razones que la CI ha tenido tanto éxito.

## ¿Realmente lo estás haciendo?

Sospecho que probablemente estés usando la integración continua en tu propia organización. Si no es así, deberías empezar. Es una práctica clave que nos permite hacer cambios de forma rápida y sencilla, y sin la cual el viaje hacia los microservicios será complicado. Dicho esto, he trabajado con muchos equipos que, a pesar de decir que hacen CI, en realidad no lo hacen en absoluto. Confunden el uso de una herramienta de CI con la adopción de la práctica de CI. La herramienta es simplemente algo que posibilita el enfoque.

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

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

Debes asegurarte de que tu código se integre. Si no revisas tu código junto con los cambios de los demás con frecuencia, terminarás dificultando la integración futura.

Incluso si utiliza ramas de corta duración para administrar los cambios, intégralas con la mayor frecuencia posible en una única rama principal.

¿Tiene un conjunto de pruebas para validar sus cambios?

Sin pruebas, solo sabemos que nuestra integración sintácticamente ha funcionado, pero no sabemos si hemos alterado el comportamiento del sistema. La integración continua sin algún tipo de verificación de que nuestro código se comporta como se espera no es integración continua.

Cuando la compilación está rota, ¿es la prioridad número uno del equipo arreglarla?

Una compilación verde que se aprueba significa que nuestros cambios se han integrado de forma segura. Una compilación roja significa que es posible que el último cambio no se haya integrado. Debe detener todos los registros posteriores que no estén relacionados con la reparación de las compilaciones para que se apruebe nuevamente. Si permite que se acumulen más cambios, el tiempo que lleva reparar la compilación aumentará drásticamente. He trabajado con equipos en los que la compilación estuvo rota durante días, lo que resultó en esfuerzos sustanciales para finalmente obtener una compilación que se apruebe.

## Mapeo de la integración continua a los microservicios

Al pensar en microservicios e integración continua, debemos pensar en cómo nuestras compilaciones de CI se asignan a 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 deberíamos asignar microservicios individuales a compilaciones de CI y código fuente?

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

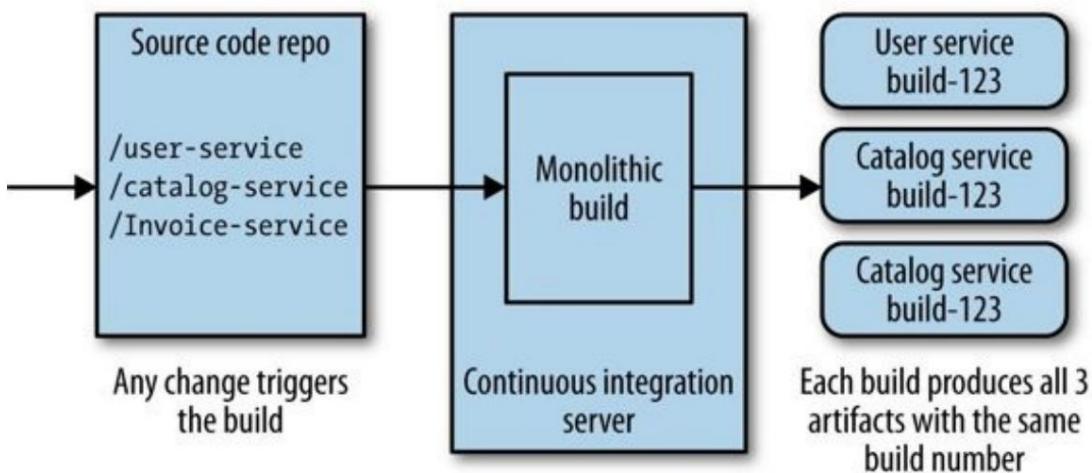


Figura 6-1. Uso de un único repositorio de código fuente y compilación de integración continua para todos los microservicios

A primera vista, esto parece mucho más simple que otros enfoques: hay menos repositorios de los que preocuparse y una compilación conceptualmente más simple. Desde el punto de vista de un desarrollador, las cosas también son bastante sencillas. Simplemente registro el código. Si tengo que trabajar en varios servicios a la vez, solo tengo que preocuparme por una confirmación.

Este modelo puede funcionar perfectamente si se adopta la idea de las versiones en bloque, en las que no le importa implementar varios servicios a la vez. En general, se trata de un patrón que se debe evitar, pero en las primeras fases de un proyecto, especialmente si solo un equipo trabaja en todo, puede tener sentido durante períodos cortos.

Sin embargo, existen algunas desventajas significativas. Si realizo un cambio de una sola línea en un solo servicio (por ejemplo, cambiar el comportamiento en el servicio de usuario en [la Figura 6-1](#)), todos los demás servicios se verifican y se crean. Esto podría llevar 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 producción. Sin embargo, lo más problemático es saber qué artefactos se deben o no se deben implementar. ¿Ahora necesito implementar todos los servicios de compilación para llevar mi pequeño cambio a producción? Puede ser difícil saberlo; tratar de adivinar qué servicios cambiaron realmente con solo leer los mensajes de confirmación es

Difícil. Las organizaciones que utilizan este enfoque a menudo recurren a implementar todo junto, algo que realmente queremos evitar.

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

Una variación de este enfoque es tener un solo árbol de código fuente con todo el código en él, con múltiples compilaciones de CI que se asignan a partes de este árbol de código fuente, como vemos en [la Figura 6-2](#). Con una estructura bien definida, puede asignar fácilmente las compilaciones a ciertas partes del árbol de código fuente. En general, no soy partidario de este enfoque, ya que este modelo puede tener ventajas y desventajas. Por un lado, mi proceso de registro de entrada y salida puede ser más simple ya que solo tengo un repositorio del que preocuparme. Por otro lado, se vuelve muy fácil adquirir el hábito de registrar el código fuente para múltiples servicios a la vez, lo que puede hacer que sea igualmente fácil cometer errores al realizar cambios que asocien los servicios. Sin embargo, preferiría mucho más este enfoque que tener una única compilación para múltiples servicios.

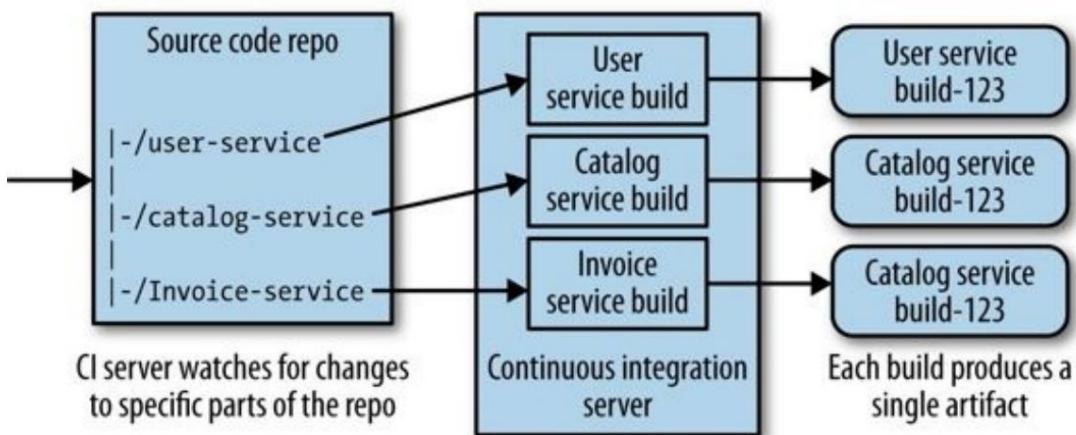


Figura 6-2. Un único repositorio de código fuente con subdirectorios asignados a compilaciones independientes

¿Existe entonces otra alternativa? El enfoque que prefiero es tener una única compilación de CI por microservicio, para permitirnos 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 único artefacto para implementar. La alineación con la propiedad del equipo también es más clara. Si eres el propietario del servicio, eres el propietario del repositorio y de la compilación. Realizar 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 fuente y el proceso de compilación monolíticos.

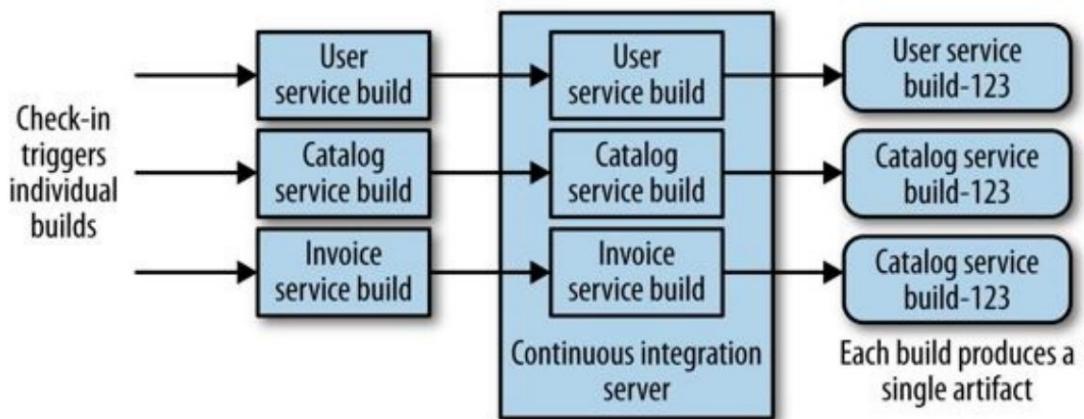


Figura 6-3. Uso de un repositorio de código fuente y una compilación de integración continua por microservicio

Las pruebas para un microservicio determinado también deben residir en el control de origen junto con el código fuente del microservicio, para garantizar que siempre sepamos qué pruebas deben ejecutarse en un servicio determinado.

Por lo tanto, cada microservicio vivirá en su propio repositorio de código fuente y en su propio proceso de compilación de CI. También utilizaremos el proceso de compilación de CI para crear nuestros artefactos implementables de manera totalmente automatizada. Ahora, veamos más allá de la CI para ver cómo encaja la entrega continua.

# Construir pipelines y entrega continua

Al principio, cuando empezamos a usar la integración continua, nos dimos cuenta del valor que a veces tiene tener varias etapas dentro de una compilación. Las pruebas son un caso muy común en el que esto entra en juego. Puedo tener muchas pruebas rápidas y de alcance reducido, y una pequeña cantidad de pruebas lentas y de alcance amplio.

Si ejecutamos todas las pruebas juntas, es posible que no podamos obtener una respuesta rápida cuando nuestras pruebas rápidas fallen 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 una secuencia de compilación.

Una etapa para las pruebas más rápidas, otra para las pruebas más lentas.

Este concepto de canal de desarrollo nos brinda una buena manera de hacer un seguimiento del progreso de nuestro software a medida que supera cada etapa, lo que nos ayuda a obtener información sobre la calidad de nuestro software. Creamos nuestro artefacto y ese artefacto se utiliza en todo el canal de desarrollo. 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 más. Como se describe en el libro de Jez Humble y Dave Farley del mismo nombre, la entrega continua es el enfoque mediante el cual recibimos comentarios constantes sobre la preparación para la producción de cada uno de los registros y, además, tratamos cada uno de ellos como un candidato para su lanzamiento.

Para adoptar plenamente este concepto, necesitamos modelar todos los procesos que intervienen en el proceso de llevar nuestro software desde el registro hasta la producción, y saber en qué punto se encuentra una determinada versión del software en términos de autorización para su lanzamiento. En CD, lo hacemos ampliando la idea del proceso de compilación de varias etapas para modelar todas y cada una de las etapas por las que tiene que pasar nuestro software, tanto manual como automatizada. En [la Figura 6-4](#), vemos un proceso de ejemplo que puede resultar familiar.



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

En este caso, lo que realmente necesitamos es una herramienta que adopte el CD como un concepto de primera clase. He visto a muchas personas intentar modificar y ampliar las herramientas de integración continua para que puedan ejecutar el CD, lo que a menudo da como resultado sistemas complejos que no son tan fáciles de usar como las herramientas que incorporan el CD desde el principio. Las herramientas que son totalmente compatibles con CD le permiten definir y visualizar estos canales, modelando todo el camino hacia la producción de su software. A medida que una versión de nuestro código avanza por el canal, si pasa uno de estos pasos de verificación automática, 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 de UAT, implementarla y, si pasa nuestras verificaciones manuales, marcar esa etapa como exitosa para que pueda pasar a la siguiente.

Al modelar todo el camino hacia la producción de nuestro software, mejoraremos enormemente la visibilidad de la calidad de nuestro software y también podemos reducir en gran medida el tiempo transcurrido entre lanzamientos.

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 independientemente unos de otros, se deduce que, al igual que con CI, querremos 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 en un momento.

## Y las inevitables excepciones

Como ocurre con todas las buenas reglas, también hay excepciones que debemos tener en cuenta. El enfoque de “un microservicio por compilación” es algo a lo que definitivamente debería aspirar, pero ¿hay momentos en los que algo más tiene sentido? Cuando un equipo comienza con un nuevo proyecto, especialmente uno desde cero en el que trabaja con una hoja de papel en blanco, es muy probable que haya una gran cantidad de rotación en términos de determinar dónde están los límites del servicio. De hecho, esta es una buena razón para mantener sus servicios iniciales en el lado más grande hasta que su comprensión del dominio se estabilice.

Durante este período de rotación, es más probable que se produzcan cambios entre los límites de los servicios 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 única compilación para reducir el costo de los cambios entre servicios.

De esto se desprende, sin embargo, que en este caso es necesario aceptar la liberación de todos los servicios como un paquete. También es absolutamente necesario que sea 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 lograr estabilidad en los límites de los servicios para separarlos adecuadamente, vuelva a fusionarlos en un servicio más monolítico (aunque mantenga la separación modular dentro del límite) y tómese el tiempo necesario para 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 respaldar su creación e instalación. Ruby tiene gemas, Java tiene archivos JAR y archivos WAR, y Python tiene huevos. Los desarrolladores con experiencia en una de estas pilas estarán bien versados en el trabajo con estos artefactos (y, con suerte, en su creación).

Sin embargo, desde el punto de vista de un microservicio, dependiendo de la pila de tecnología que se utilice, este artefacto puede no ser suficiente por sí solo. Si bien se puede hacer que un archivo JAR de Java sea ejecutable y ejecute un proceso HTTP integrado, para aplicaciones como Ruby y Python, se espera utilizar 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 ejecutar nuestros artefactos. Aquí es donde las herramientas de administración de configuración automatizada como Puppet y Chef pueden ayudar.

Otra desventaja 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 múltiples servicios juntos. Podría ser un desarrollador o evaluador que deseé probar alguna funcionalidad, o podría ser alguien que gestione una implementación de producción. Ahora imagine que esos servicios utilizan tres mecanismos de implementación completamente diferentes. Tal vez tengamos una gema Ruby, un archivo JAR y un paquete NPM de nodeJS. ¿Le agradecerían?

La automatización puede ser de gran ayuda para ocultar las diferencias en los mecanismos de implementación de los artefactos subyacentes. Chef, Puppet y Ansible también admiten varios artefactos de compilación comunes específicos de cada tecnología. Pero existen diferentes tipos de artefactos con los que puede resultar incluso 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 crear un paquete deb; o para Windows, un MSI.

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

La desventaja puede ser la dificultad de crear los paquetes en primer lugar. Para Linux, la [herramienta de administración de paquetes FPM](#) ofrece una abstracción más agradable 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 desechar en comparación con las capacidades del espacio Linux. El sistema de paquetes NuGet ha comenzado a ayudar a abordar esto, al menos en términos de ayudar a administrar bibliotecas de desarrollo. Más recientemente, Chocolatey NuGet ha extendido estas ideas, proporcionando un administrador de paquetes para Windows diseñado para implementar herramientas y servicios, que es mucho más parecido a los administradores de paquetes en el espacio Linux. Este es ciertamente un paso en la dirección correcta, aunque el hecho de que el estilo idiomático en Windows todavía sea implementar algo en IIS significa que este enfoque puede ser poco atractivo para algunos equipos de Windows.

Otra desventaja, por supuesto, podría ser si está implementando en varios sistemas operativos diferentes. La sobrecarga de administrar artefactos para diferentes sistemas operativos puede ser bastante alta. Si está creando software para que lo instalen otras personas, es posible que no tenga otra opción. Sin embargo, si está instalando software en máquinas que usted controla, le sugeriría que considere unificar o al menos reducir la cantidad de sistemas operativos diferentes que utiliza. Puede reducir en gran medida las variaciones en el comportamiento de una máquina a otra y simplificar las tareas de implementación y mantenimiento.

En general, los equipos que he visto que han adoptado la gestión de paquetes basada en SO han simplificado su enfoque de implementación y tienden a evitar la trampa de los scripts de implementación grandes y complejos. Especialmente si utiliza Linux, esta puede ser una buena manera de simplificar la implementación de microservicios utilizando diferentes pilas de tecnología.

## Imágenes personalizadas

Uno de los desafíos de los sistemas de gestión de configuración automatizada como Puppet, Chef y Ansible puede ser el tiempo que lleva ejecutar los scripts en una máquina. Tomemos un ejemplo simple de un servidor que se está aprovisionando y configurando para permitir la implementación de una aplicación Java. Supongamos que estoy usando AWS para aprovisionar el servidor, utilizando la imagen estándar de Ubuntu. Lo primero que tengo que hacer es instalar Oracle JVM para ejecutar mi aplicación Java. He visto que este sencillo proceso lleva alrededor de cinco minutos, con un par de minutos ocupados por la máquina que se está aprovisionando y unos pocos más para instalar la JVM. Luego podremos pensar en implementar nuestro software en él.

En realidad, este es un ejemplo bastante trivial. A menudo, querremos instalar otros programas comunes. Por ejemplo, podríamos querer usar collectd para recopilar estadísticas del sistema operativo, usar logstash para la agregación de registros y quizás instalar los programas adecuados de nagios para la monitorización (hablaremos más sobre este programa en [el Capítulo 8](#)). Con el tiempo, es posible que se agreguen más cosas, lo que hará que se necesiten más y más tiempos para el aprovisionamiento de estas dependencias.

Puppet, Chef, Ansible y similares pueden ser inteligentes y evitarán instalar software que ya esté 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 en funcionamiento durante demasiado tiempo, ya que no queremos permitir demasiadas desviaciones de configuración (que exploraremos en mayor profundidad en breve). Y si estamos usando una plataforma de computación a pedido, es posible que estemos apagando y activando nuevas instancias constantemente a diario (o con mayor frecuencia), por lo que la naturaleza declarativa de estas herramientas de administración de configuración puede ser de uso limitado.

Con el tiempo, ver que se instalan las mismas herramientas una y otra vez puede convertirse en un verdadero fastidio. Si intentas hacer esto varias veces al día (quizás como parte del desarrollo o la integración continua), esto se convierte en un verdadero problema en términos de proporcionar una respuesta rápida. También puede generar un mayor tiempo de inactividad al implementar en producción si tus sistemas no permiten una implementación sin tiempo de inactividad, ya que estás esperando instalar todos los requisitos previos en tus máquinas incluso antes de instalar tu 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 método para reducir este tiempo de puesta en marcha es crear una imagen de máquina virtual que incorpore algunas de las dependencias comunes que utilizamos, como se muestra en [la Figura 6-5](#). Todas las plataformas de virtualización que he utilizado permiten crear imágenes propias, y las herramientas para hacerlo son mucho más avanzadas que hace unos años. Esto cambia un poco las cosas. Ahora podemos incorporar las herramientas comunes a nuestra propia imagen. Cuando queremos implementar nuestro software, creamos una instancia de esta imagen personalizada y todo lo que tenemos que hacer es instalar la última versión de nuestro servicio.

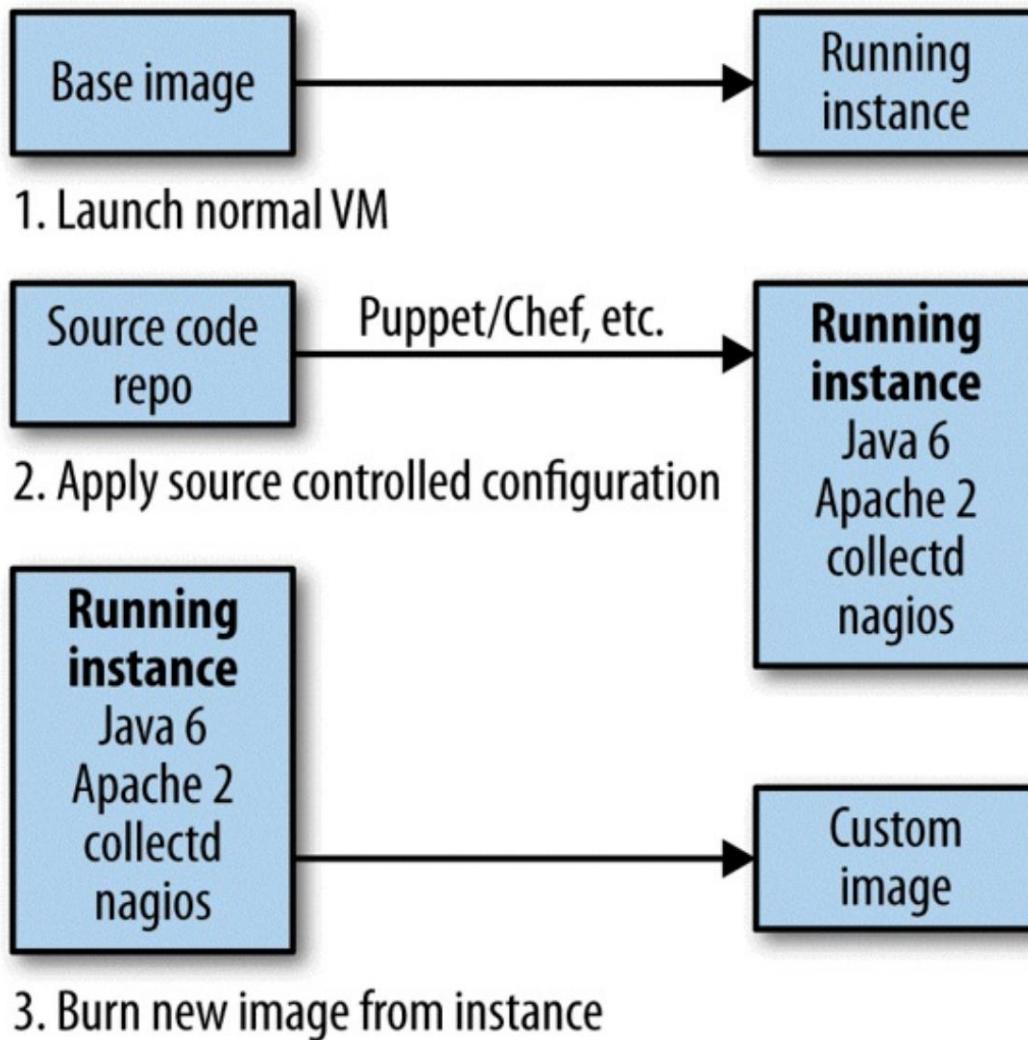


Figura 6-5. Creación de una imagen de máquina virtual personalizada

Por supuesto, como creas la imagen solo una vez, cuando ejecutes copias de esta imagen posteriormente no necesitarás dedicar tiempo a instalar las dependencias, ya que ya están ahí. Esto puede resultar en un ahorro de tiempo significativo. Si tus dependencias principales no cambian, las nuevas versiones de tu servicio pueden seguir usando la misma imagen base.

Sin embargo, este enfoque tiene algunas desventajas. La creación de imágenes puede llevar mucho tiempo. Esto significa que es posible que los desarrolladores quieran 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 verdadero problema si estás creando tus propias imágenes de VMWare, por ejemplo, ya que mover una imagen de 20 GB por una red no siempre es una actividad sencilla. En breve analizaremos la tecnología de contenedores, y en concreto Docker, que puede evitar algunos de estos inconvenientes.

Históricamente, uno de los desafíos es que la cadena de herramientas necesaria para crear una imagen de este tipo varía de una plataforma a otra. Crear una imagen de VMWare es diferente a crear una AMI de AWS, una imagen de Vagrant o una imagen de Rackspace. Esto podría no haber sido un problema si tuviera la misma plataforma en todas partes, pero no todas las organizaciones tuvieron tanta suerte. E incluso si la tuvieron, las herramientas en este espacio a menudo eran difíciles de usar y no funcionaban bien con otras herramientas que pudiera estar usando para la configuración de la máquina.

[Envasador](#) es una herramienta diseñada para facilitar enormemente la creación de imágenes. Mediante el uso de scripts de configuración de su elección (se admiten Chef, Ansible, Puppet y más), nos permite crear imágenes para diferentes plataformas a partir de la misma configuración. Al momento de escribir esto, tiene soporte para 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 puede crear una imagen para implementar en su entorno de producción de AWS y una imagen Vagrant correspondiente para el desarrollo y la prueba locales, todo desde la misma configuración.

## Las imágenes como artefactos

Por lo tanto, podemos crear imágenes de máquinas virtuales que incorporen dependencias para acelerar la retroalimentación, pero ¿por qué detenernos allí? Podríamos ir más allá, integrar nuestro servicio en la propia imagen 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 puesta en marcha realmente rápido es la razón por la que Netflix ha adoptado el modelo de integrar sus propios servicios como AMI de AWS.

Al igual que con los paquetes específicos del sistema operativo, estas imágenes de máquinas virtuales se convierten en una buena manera 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 utiliza un archivo JAR o gema? Lo único que nos importa es que funcione. Podemos centrar nuestros esfuerzos, entonces, en automatizar la creación y la implementación de estas imágenes. Esto también se convierte en una forma muy elegante 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 poder reproducir automáticamente los servicios y, con suerte, entornos completos a voluntad. Pero una vez que ejecutamos nuestro proceso de implementación, ¿qué sucede si alguien llega, inicia sesión en el equipo y cambia 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 código 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, sin importar cuán pequeño sea, debe pasar por un proceso de compilación para crear una nueva máquina. Puedes 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, lo que garantizaría que nadie pudiera iniciar sesión en la máquina para realizar un cambio.

Por supuesto, las mismas advertencias que comentamos antes sobre el tiempo de ciclo siguen siendo válidas. Y también debemos asegurarnos de que todos los datos que nos interesan y que están almacenados en la caja se almacenen en otro lugar. Dejando de lado estas complejidades, he visto que la adopción de este patrón conduce a implementaciones mucho más sencillas y entornos sobre los que es más fácil razonar. Y, como ya he dicho, ¡hay que hacer todo lo que podamos para simplificar las cosas!

## Entornos

A medida que nuestro software avanza por las etapas de la canalización de CD, también se implementará en diferentes tipos de entornos. Si pensamos en la canalización de compilación de ejemplo de la Figura 6-4, probablemente tengamos que considerar al menos cuatro entornos distintos: un entorno donde ejecutamos nuestras pruebas lentas, otro para UAT, otro para rendimiento y uno final para producción. Nuestro microservicio debería ser el mismo en todo momento, pero el entorno será diferente. Como mínimo, serán conjuntos separados y distintos de configuración y hosts. Pero a menudo pueden variar mucho más que eso. Por ejemplo, nuestro entorno de producción para nuestro servicio podría constar de varios hosts con equilibrio de carga distribuidos en dos centros de datos, mientras que nuestro entorno de prueba podría tener todo ejecutándose en un solo host. Estas diferencias en los entornos pueden presentar algunos problemas.

Hace muchos años, me pasó algo parecido. Estábamos implementando un servicio web Java en un contenedor de aplicaciones WebLogic en clúster en producción. Este clúster WebLogic replicaba el estado de la sesión entre varios nodos, lo que nos daba cierto nivel de resiliencia si fallaba un solo nodo. Sin embargo, las licencias de WebLogic eran caras, al igual que las máquinas en las que se implementaba nuestro software. Esto significaba que, en nuestro entorno de prueba, nuestro software se implementaba en una sola máquina, en una configuración no agrupada.

Esto nos afectó mucho durante una versión. Para que WebLogic pueda copiar el estado de la sesión entre nodos, los datos de la sesión deben ser serializables correctamente. Lamentablemente, una de nuestras confirmaciones rompió esto, por lo que cuando implementamos en producción nuestra replicación de sesión falló. Terminamos resolviendo esto haciendo un gran esfuerzo 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 ellos cumple una función diferente. En mi computadora portátil de desarrollador, quiero implementar rápidamente el servicio, posiblemente contra colaboradores que no tienen acceso a la base de datos, para ejecutar pruebas o llevar a cabo alguna validación manual del comportamiento, mientras que cuando lo implemento en un entorno de producción, es posible que desee implementar varias copias de mi servicio de manera equilibrada, tal vez divididas en uno o más centros de datos por razones de durabilidad.

A medida que pasa de su computadora portátil al servidor de compilación, al entorno de UAT y finalmente a la producción, querrá asegurarse de que sus entornos sean cada vez más similares a los de producción para detectar antes cualquier problema asociado con estas diferencias ambientales. Este será un equilibrio constante. A veces, el tiempo y el costo para reproducir entornos similares a los de producción pueden ser prohibitivos, por lo que debe hacer concesiones. Además, a veces el uso de un entorno similar a la 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 similares a los de producción y retroalimentación rápida, no será estático. Esté atento a los errores que encuentre más adelante y a los tiempos de respuesta, y ajústelos.

Este equilibrio según sea necesario.

Administrar entornos para sistemas monolíticos con un solo artefacto puede ser un desafío, especialmente si no tiene acceso a sistemas que se puedan automatizar fácilmente. Cuando piensa en múltiples entornos por microservicio, esto puede resultar aún más abrumador. En breve, analizaremos algunas plataformas de implementación diferentes que pueden facilitarnos mucho esta tarea.

## Configuración del servicio

Nuestros servicios necesitan cierta configuración. Lo ideal es que sea una cantidad pequeña y limitada a aquellas características que cambian de un entorno a otro, como por ejemplo, ¿ 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 reducirse al mínimo. Cuanto más cambie su configuración el comportamiento fundamental del servicio y cuanto más varíe dicha configuración de un entorno a otro, más problemas encontrará solo en ciertos entornos, lo que resulta extremadamente complicado.

Entonces, si tenemos alguna configuración para nuestro servicio que cambia de un entorno a otro, ¿cómo deberíamos manejar esto como parte de nuestro proceso de implementación? Una opción es crear un artefacto por entorno, con la configuración dentro del artefacto mismo. Inicialmente, esto parece sensato. La configuración está incorporada; solo hay que implementarla y todo debería funcionar bien, ¿verdad? Esto es problemático. Recuerde el concepto de entrega continua. Queremos crear un artefacto que represente nuestro candidato de lanzamiento y moverlo a través de nuestro proceso de producción, confirmando que es lo suficientemente bueno para entrar en producción. Imaginemos que construyo un artefacto de prueba de servicio al cliente y un artefacto de producción de servicio al cliente. Si mi artefacto de prueba de servicio al cliente pasa las pruebas, pero es el artefacto de producción de servicio al cliente el que realmente implemento, ¿puedo estar seguro de que he verificado el software que realmente termina en producción?

También existen otros desafíos. En primer lugar, está el tiempo adicional que se necesita para crear estos artefactos. Luego, el hecho de que necesitas saber en el momento de la compilación qué entornos existen. ¿Y cómo manejas los datos de configuración confidenciales? No quiero que la información sobre las contraseñas de producción se registre con mi código fuente, pero si se necesita en el momento de la compilación para crear todos esos artefactos, esto suele ser difícil de evitar.

Un mejor enfoque es crear un único artefacto y administrar la configuración por separado.

Este podría ser un archivo de propiedades que exista para cada entorno o diferentes parámetros que se pasen a un proceso de instalación. Otra opción popular, especialmente cuando se trabaja con una gran cantidad de microservicios, es utilizar un sistema dedicado para proporcionar la configuración, que exploraremos más en [el Capítulo 11](#).

## Mapeo de servicio a host

Una de las preguntas que surge bastante temprano en el debate sobre los microservicios es "¿Cuántos servicios hay por máquina?". Antes de continuar, deberíamos elegir un término mejor que "máquina", o incluso el término más genérico que utilicé antes. En esta era de la virtualización, la correspondencia entre un único 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 (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 única máquina física puede asignarse a varios host independientes, cada uno de los cuales podría albergar uno o más servicios.

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

Tengo una opinión clara sobre qué modelo es preferible, pero hay varios factores que se deben tener en cuenta para determinar cuál será el modelo adecuado para usted. También es importante comprender que algunas de las decisiones que tomamos en este sentido limitarán algunas de las opciones de implementación disponibles.

## Múltiples servicios por host

Tener varios servicios por host, como se muestra en [la Figura 6-6](#), es atractivo por varias razones. En primer lugar, desde el punto de vista puramente 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 es una función de la cantidad de hosts que debe administrar. Si se agrupan más servicios en un único host, la carga de trabajo de administración del host no aumenta a medida que aumenta la cantidad de servicios. En segundo lugar, está el costo. Incluso si tiene acceso a una plataforma de virtualización que le permite aprovisionar y redimensionar 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 los analizaremos en breve.

Este modelo también resulta familiar para quienes implementan en algún tipo de contenedor de aplicaciones. En cierto modo, el uso de un contenedor de aplicaciones es un caso especial del modelo de múltiples servicios por host, por lo que lo analizaremos por separado. Este modelo también puede simplificar la vida del desarrollador. Implementar múltiples servicios en un solo host en producción es sinónimo de implementar múltiples servicios en una estación de trabajo o computadora portátil de desarrollo local. Si queremos analizar un modelo alternativo, queremos encontrar una forma de mantener esto conceptualmente simple para los desarrolladores.

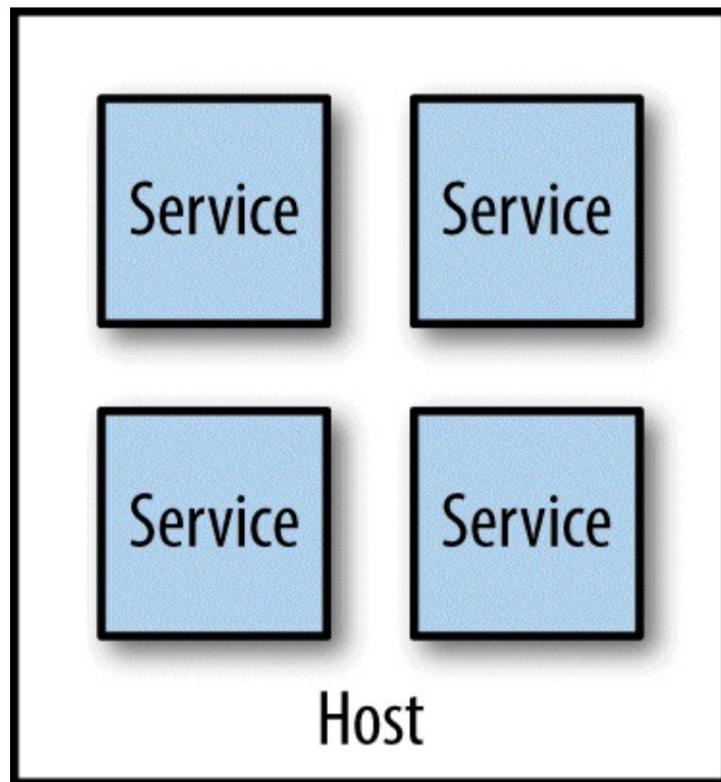


Figura 6-6. Múltiples microservicios por host

Sin embargo, este modelo presenta algunos desafíos. En primer lugar, puede dificultar el monitoreo. Por ejemplo, al rastrear la CPU, ¿necesito rastrear la CPU de un servicio independientemente de los demás? ¿O me preocupa la CPU del equipo en su conjunto? Efectos secundarios

También puede ser difícil evitarlo. Si un servicio está bajo 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 máquina, 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 del impacto de las fallas del host sea más complejo: dejar fuera de servicio 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 a personas unir varias implementaciones de servicios, implementando varios servicios diferentes en un solo host en un solo paso, para intentar simplificar la implementación de varios servicios en un solo host. En mi opinión, la pequeña ventaja de mejorar la simplicidad se ve más que compensada por el hecho de que hemos renunciado a uno de los beneficios clave de los microservicios: esforzarnos por lograr una publicación 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 los servicios para diferentes equipos se instalan en el mismo host, ¿quién configura el host para sus servicios? Lo más probable es que esto termine 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 descartadas, al igual que los servidores inmutables, a menos que vincules varios servicios diferentes en un solo artefacto, lo que realmente queremos evitar.

El hecho de que tengamos varios servicios en un único host significa que los esfuerzos por enfocar la escalabilidad al servicio que más lo necesita pueden ser complicados. Asimismo, si un microservicio maneja datos y operaciones que son especialmente sensibles, es posible que queramos configurar el host subyacente de manera diferente o incluso colocar el host en un segmento de red separado.

Tener todo en un solo host significa que podríamos terminar teniendo que tratar 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 en torno a la implementación y la gestión de hosts 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 solía implicar un largo plazo de entrega y un compromiso financiero a largo plazo. No era raro que los clientes con los que he trabajado aprovisionaran nuevos servidores solo cada dos o tres años, y tratar de conseguir máquinas adicionales fuera de estos plazos era difícil. Pero las plataformas informáticas a pedido 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 IIS o aplicaciones Java en un contenedor de servlets, estará familiarizado con el modelo en el que múltiples servicios o aplicaciones distintos se encuentran dentro de un único contenedor de aplicaciones, que a su vez se encuentra en un único host, como vemos en [la Figura 6-7](#). La idea es que el contenedor de aplicaciones en el que residen sus servicios le brinde beneficios en términos de una mejor capacidad de administración, como soporte de agrupamiento para manejar la agrupación de múltiples instancias, herramientas de monitoreo y similares.

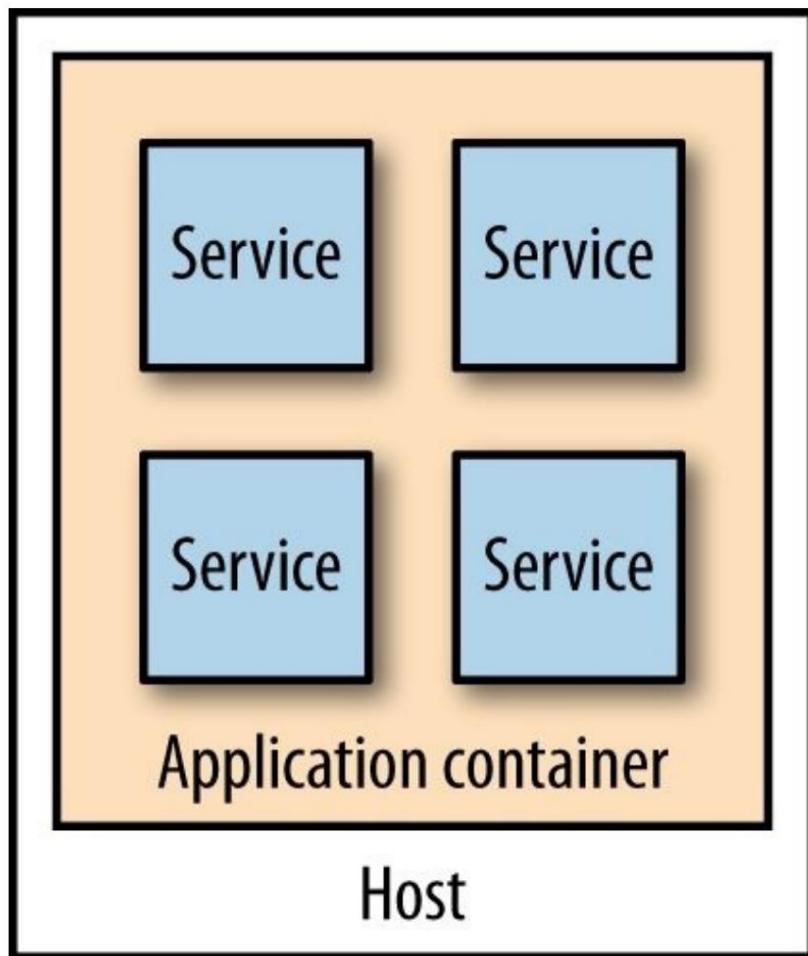


Figura 6-7. Múltiples microservicios por host

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

La primera de las desventajas es que inevitablemente limitan la elección de tecnología. Tienes que comprar una pila de tecnología. Esto puede limitar no solo las opciones de tecnología para la implementación del servicio en sí, sino también las opciones que tienes en términos de automatización y administración de tus sistemas. Como analizaremos en breve, una de las formas en que podemos abordar la sobrecarga de administrar múltiples hosts es mediante la automatización, por lo que

Limitar nuestras opciones para resolver esto puede ser doblemente perjudicial.

También cuestionaría el valor de algunas de las características de los contenedores. Muchos de ellos promocionan la capacidad de administrar clústeres para admitir estados de sesión en memoria compartida, 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 consideremos 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 puesta en marcha bastante lentos, lo que afecta los ciclos de retroalimentación de los desarrolladores.

También existen otros tipos 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 subprocessos también es mucho más complejo, ya que hay varias aplicaciones que comparten el mismo proceso. Y recuerde, incluso si obtiene valor de un contenedor específico de la tecnología, no son gratuitos. Aparte del hecho de que muchos de ellos son comerciales y, por lo tanto, implican un costo, agregan una sobrecarga de recursos en sí mismos.

En definitiva, este enfoque es un intento de optimizar la escasez de recursos que, sencillamente, ya no pueden ser suficientes. Si decide tener varios servicios por host como modelo de implementación, le recomendaría encarecidamente que considere microservicios implementables autónomos como artefactos. Para .NET, esto es posible con cosas como Nancy, y Java ha respaldado este modelo durante años. Por ejemplo, el venerable contenedor integrado Jetty crea un servidor HTTP autónomo muy liviano, que es el núcleo de la pila Dropwizard. Se sabe que Google usa con bastante agrado los contenedores integrados Jetty para servir contenido estático directamente, por lo que sabemos que estas cosas pueden funcionar a escala.

## Servicio único por host

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

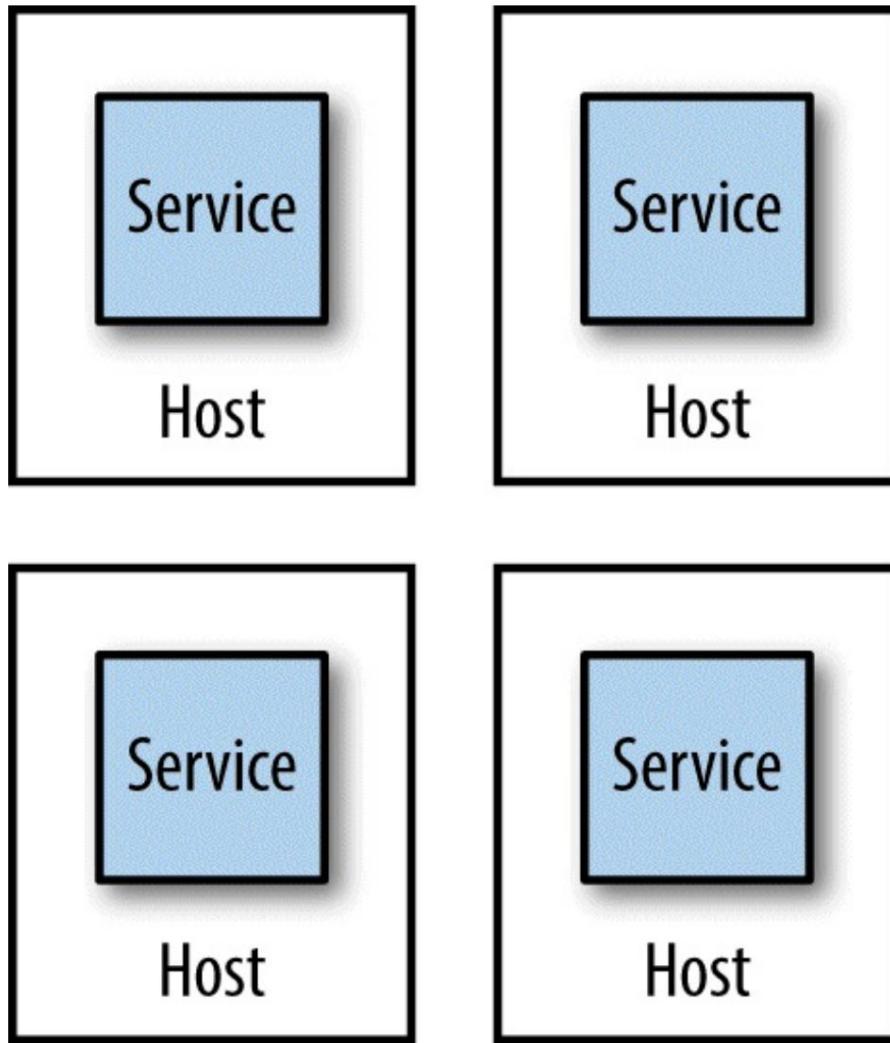


Figura 6-8. Un único microservicio por host

Igualmente importante es que hemos abierto la posibilidad de utilizar técnicas de implementación alternativas, como las implementaciones basadas en imágenes o el patrón de servidor inmutable, que analizamos anteriormente.

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

Razonar sobre esto puede ayudar a reducir la complejidad. Si aún no puede adoptar este modelo, no diré que los microservicios no sean para usted. Pero le sugeriría que busque avanzar hacia este modelo con el tiempo como una forma de reducir la complejidad que puede traer una arquitectura de microservicios.

Sin embargo, tener una mayor cantidad de hosts tiene posibles desventajas. Tenemos que administrar más servidores y también puede haber una implicación de costos por 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 que supone manejar una gran cantidad de hosts en breve.

## Plataforma como servicio

Cuando se utiliza una plataforma como servicio (PaaS), se trabaja en un nivel de abstracción más alto que en un único 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 para usted. Algunas de estas plataformas intentarán de forma transparente gestionar el escalado del sistema hacia arriba y hacia abajo por usted, aunque una forma más común (y en mi experiencia menos propensa a errores) le permitirá cierto control sobre la cantidad de nodos en los que puede ejecutarse su servicio, pero se encarga del resto.

En el momento de escribir este artículo, la mayoría de las mejores y más pulidas soluciones PaaS están alojadas. Heroku me viene a la mente como probablemente la clase dorada de PaaS. No solo se encarga de ejecutar su servicio, sino que también admite servicios como bases de datos de una manera muy simple. Existen soluciones alojadas por uno mismo en este espacio, aunque son más inmaduras que las soluciones alojadas.

Cuando las soluciones PaaS funcionan bien, lo hacen muy bien. Sin embargo, cuando no funcionan del todo bien para ti, a menudo no tienes mucho control en términos de intervenir para solucionar los problemas. Esto es parte del equilibrio que debes asumir. Yo diría que, en mi experiencia, cuanto más inteligentes intentan ser las soluciones PaaS, más fallan. He utilizado más de una PaaS que intenta escalar automáticamente en función del uso de la aplicación, pero lo hace mal. Invariablemente, las heurísticas que impulsan estas inteligencias tienden a estar diseñadas para la aplicación promedio en lugar de para tu caso de uso específico. Cuanto más no estándar sea tu aplicación, más probable es que no funcione bien con una PaaS.

Como las buenas soluciones PaaS se encargan de muchas cosas por ti, pueden ser una excelente manera de manejar el aumento de los gastos generales que se generan al tener muchas más partes móviles. Dicho esto, todavía no estoy seguro de que tengamos todos los modelos adecuados en este espacio, y las limitadas opciones de alojamiento propio significan que este enfoque podría no funcionar para ti. Sin embargo, en la próxima década espero que apuntemos a PaaS para la implementación más que a 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 número reducido de máquinas, es posible gestionar todo manualmente. Yo solía hacerlo así.

Recuerdo que manejaba un pequeño conjunto de máquinas de producción y recopilaba registros, implementaba software y verificaba procesos iniciando sesión manualmente en la máquina. Mi productividad parecía estar limitada por la cantidad de ventanas de terminal que podía tener abiertas a la vez (un segundo monitor era un gran avance). Sin embargo, esto se estropea muy rápido.

Una de las desventajas de la configuración de un solo servicio por host es la percepción de que aumentará la cantidad de sobrecarga para administrar estos hosts. Esto es ciertamente cierto si se hace todo manualmente. ¡Duplicar los servidores duplica el trabajo! Pero si automatizamos el control de nuestros hosts y la implementación de los servicios, entonces no hay razón para que agregar más hosts aumente nuestra carga de trabajo de manera lineal.

Pero incluso si mantenemos pequeño el número de hosts, todavía tendremos muchos servicios.

Esto significa múltiples implementaciones que manejar, servicios que monitorear y registros que recopilar.

La automatización es esencial.

La automatización también es la forma en que podemos garantizar que nuestros desarrolladores sigan siendo productivos. Darles la capacidad de autoaprovisionarse de servicios individuales o grupos de servicios es fundamental para facilitarles la vida a los desarrolladores. Lo ideal sería que los desarrolladores tuvieran acceso exactamente a la misma cadena de herramientas que se utiliza para la implementación de nuestros servicios de producción, de modo de garantizar que podamos detectar los problemas en una etapa temprana. En este capítulo, analizaremos muchas tecnologías que adoptan este enfoque.

Elegir una tecnología que permita la automatización es muy importante. Esto comienza con las herramientas que se utilizan para administrar los hosts. ¿Puede 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 ofrece listados de bienes raíces para clientes minoristas y comerciales en Australia y en otras partes de la región de Asia y el Pacífico. Durante varios años, ha estado moviendo su plataforma hacia un diseño distribuido de microservicios. Cuando comenzó este viaje, tuvo que dedicar mucho tiempo a lograr que las herramientas en torno a los servicios fueran las adecuadas, lo que facilitó a los desarrolladores el aprovisionamiento de máquinas, la implementación de su código o el monitoreo de las mismas. Esto provocó una carga de trabajo inicial para poner las cosas en marcha.

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

Este tipo de patrón también se ve confirmado por las experiencias de [Gilt](#), Gilt es una empresa de venta de moda online que comenzó en 2007. La aplicación Rails monolítica de Gilt empezaba a resultar difícil de escalar y, en 2009, la empresa decidió empezar a descomponer el sistema en microservicios. Una vez más, la automatización, especialmente las herramientas para ayudar a los desarrolladores, se presentó como una razón clave para impulsar la explosión de Gilt en el uso de microservicios. Un año después, Gilt tenía alrededor de 10 microservicios en funcionamiento; en 2012, más de 100; y en 2014, más de 450 microservicios según el propio recuento de Gilt; en otras palabras, alrededor de tres servicios por cada desarrollador de Gilt.

## De lo físico a lo virtual

Una de las herramientas clave que tenemos a nuestra disposición para gestionar una gran cantidad de hosts es encontrar formas de dividir las máquinas físicas existentes en partes más pequeñas. La virtualización tradicional, como la que utiliza VMWare o la que utiliza AWS, ha generado enormes beneficios a la hora de reducir los gastos generales de la gestión de hosts. Sin embargo, ha habido algunos avances nuevos en este ámbito que vale la pena explorar, ya que pueden abrir posibilidades aún más interesantes para gestionar nuestra arquitectura de microservicios.

# Virtualización tradicional

¿Por qué es caro tener muchos hosts? Bueno, si necesitas un servidor físico por host, la respuesta es bastante obvia. Si este es el mundo en el que te mueves, entonces el modelo de múltiples servicios por host es probablemente adecuado para ti, aunque no te sorprendas si esto se convierte en una restricción cada vez más difícil. Sin embargo, sospecho que la mayoría de ustedes están usando 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 simplemente dividir nuestra infraestructura física en partes cada vez más pequeñas?

Bueno, para algunas personas, sí es posible. Sin embargo, dividir la máquina en cada vez más máquinas virtuales 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 podría ser más fácil de manejar y organizar, y tal vez podríamos decidir poner camisetas en uno de los espacios ahora en lugar de solo calcetines, pero más separadores significan menos espacio en general.

En el mundo de la virtualización, tenemos un gasto similar al de los separadores de cajones de calcetines. Para entender de dónde proviene este gasto, veamos cómo se lleva a cabo la mayor parte de la virtualización.

[La Figura 6-9](#) muestra una comparación de dos tipos de virtualización. A la izquierda, vemos las distintas 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 SO ejecutamos algo llamado hipervisor, que tiene dos funciones clave. En primer lugar, asigna recursos como CPU y memoria desde el host virtual al host físico. En segundo lugar, actúa como una capa de control, lo que nos permite manipular las propias máquinas virtuales.

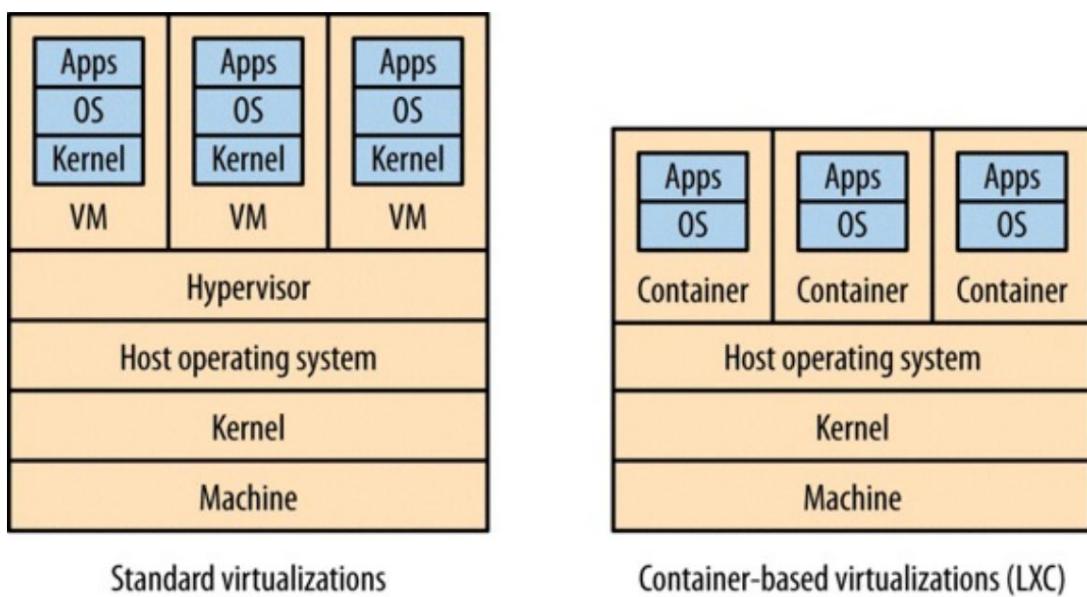


Figura 6-9. Comparación entre la virtualización estándar de tipo 2 y los contenedores livianos

Dentro de las máquinas virtuales, obtenemos lo que parecen ser hosts completamente diferentes. Pueden ejecutar sus propios

Sistemas operativos con sus propios núcleos. Pueden considerarse máquinas casi herméticamente selladas, que se mantienen aisladas del host físico subyacente y de las otras máquinas virtuales por el hipervisor.

El problema es que el hipervisor necesita reservar recursos para hacer su trabajo, lo que quita CPU, E/S y memoria que podrían usarse en otras partes. Cuantos más hosts administre el hipervisor, más recursos necesitará. En un momento determinado, esta sobrecarga se convierte en una limitación para seguir fragmentando la infraestructura física. En la práctica, esto significa que, a menudo, hay rendimientos decrecientes al fragmentar una caja física en partes cada vez más pequeñas, ya que proporcionalmente se destinan más y más recursos a la sobrecarga del hipervisor.

## Vagabundo

Vagrant es una plataforma de implementación muy útil que normalmente se utiliza para desarrollo y pruebas en lugar de producción. Vagrant te ofrece una nube virtual en tu computadora portátil. En el fondo, utiliza un sistema de virtualización estándar (normalmente VirtualBox, aunque puede utilizar otras plataformas). Permite definir un conjunto de máquinas virtuales en un archivo de texto, junto con la forma en que se conectan en red y en qué imágenes deben basarse. Este archivo de texto se puede registrar y compartir entre los miembros del equipo.

Esto hace que sea más fácil crear entornos similares a los de producción en su máquina local. Puede poner en marcha varias máquinas virtuales a la vez, apagar algunas de ellas para probar los modos de falla y asignar las máquinas virtuales a directorios locales para poder realizar cambios y verlos reflejados de inmediato. Incluso para los equipos que utilizan plataformas de nube a pedido como AWS, la respuesta más rápida que brinda el uso de Vagrant puede ser una gran ventaja para los equipos de desarrollo.

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

## Contenedores de 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 residen 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 ejecuto un proceso en una terminal, ese proceso secundario generalmente se considera un proceso secundario del proceso de la terminal. El trabajo del núcleo de Linux es mantener este árbol de procesos.

Los contenedores de Linux amplían esta idea. Cada contenedor es, en efecto, un subárbol del árbol de procesos del sistema general. A estos contenedores se les pueden asignar recursos físicos, algo que el núcleo gestiona por nosotros. Este enfoque general ha existido en muchas formas, como Solaris Zones y OpenVZ, pero es LXC el que se ha vuelto más popular.

LXC ahora está disponible de fábrica en cualquier kernel Linux moderno.

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

No solo nos beneficiamos de los recursos que ahorramos al no necesitar un hipervisor, sino que también ganamos en términos de retroalimentación. Los contenedores Linux se aprovisionan mucho más rápido que las máquinas virtuales completas. No es raro que una máquina virtual tarde muchos minutos en iniciarse, pero con los contenedores Linux, el inicio puede tardar unos segundos. También tienes un control más preciso sobre los propios contenedores en términos de asignarles recursos, lo que hace que sea mucho más fácil ajustar la configuración para aprovechar al máximo el hardware subyacente.

Debido a la naturaleza liviana de los contenedores, podemos tener muchos más de ellos 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 de manera mucho más rentable de lo que sería posible si quisieramos ejecutar cada servicio en su propia máquina virtual.

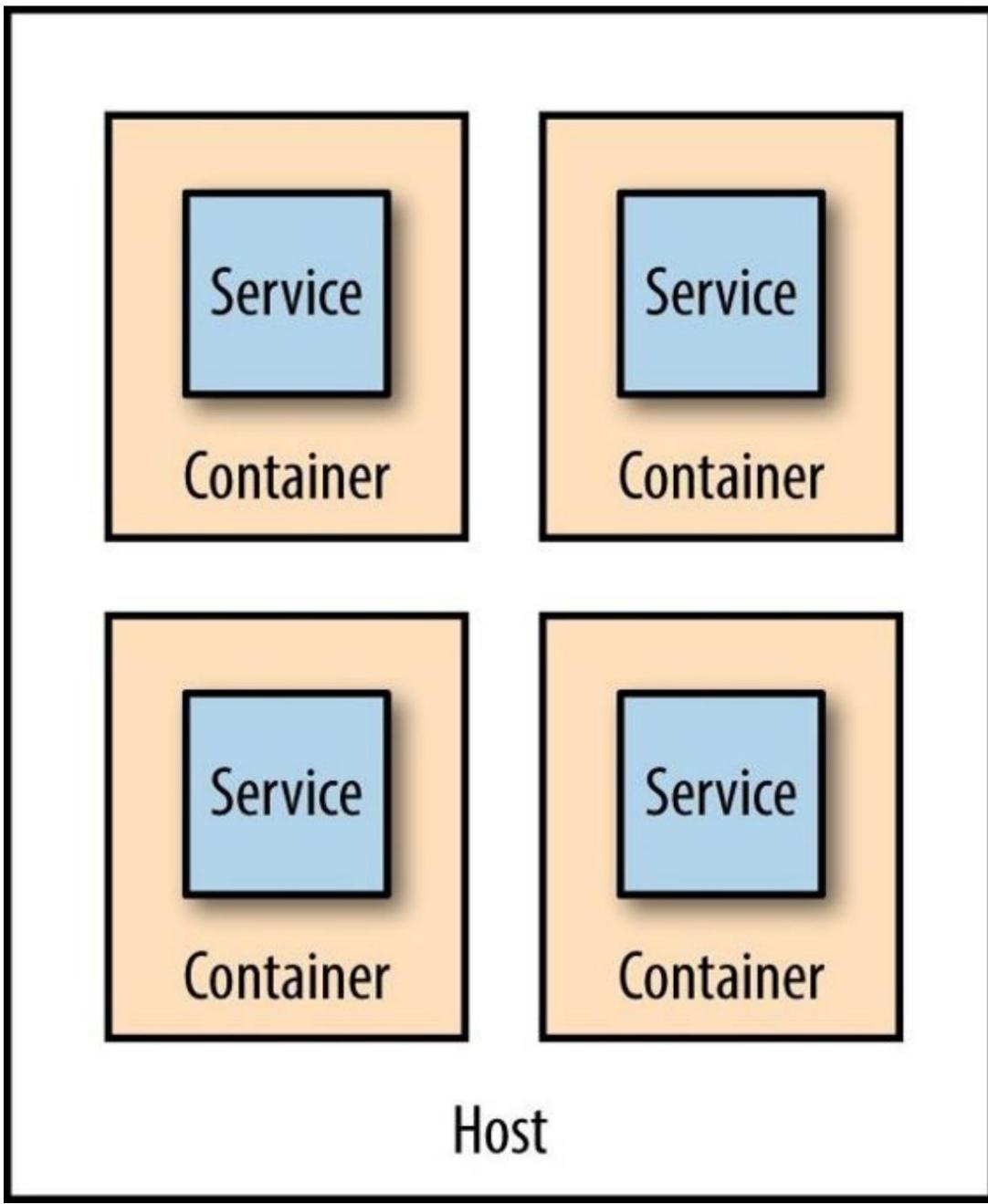


Figura 6-10. Ejecución de servicios en contenedores separados

Los contenedores también se pueden utilizar bien con la virtualización completa. He visto más de un proyecto que aprovisiona una gran instancia de AWS EC2 y ejecuta contenedores LXC en ella para obtener lo mejor de ambos mundos: una plataforma de computación efímera a pedido en forma de EC2, junto con contenedores altamente flexibles y rápidos que se ejecutan sobre ella.

Sin embargo, los contenedores de Linux no están exentos de problemas. Imaginemos que tengo muchos microservicios ejecutándose en sus propios contenedores en un host. ¿Cómo los ve el mundo exterior? Se 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 mediante IPTables para exponer contenedores directamente. Otro punto a tener en cuenta es que estos contenedores no se pueden considerar completamente aislados entre sí. Hay muchas formas documentadas y conocidas en las que un proceso de un contenedor puede salir e interactuar con otros contenedores o el host subyacente. Algunos de estos problemas son por diseño y otros son errores que son

Se está abordando este problema, pero, de cualquier manera, si no confía en el código que está ejecutando, no espere poder ejecutarlo en un contenedor y estar seguro. Si necesita ese tipo de aislamiento, deberá considerar el uso de máquinas virtuales.

## Estibador

Docker es una plataforma construida sobre contenedores livianos. Se encarga de gran parte del trabajo relacionado con el manejo de contenedores por usted. En Docker, usted 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 aplicaciones Docker.

La abstracción de la aplicación Docker nos resulta útil porque, al igual que con las imágenes de máquinas virtuales, la tecnología subyacente que se utiliza para implementar el servicio está oculta. Hacemos que las compilaciones de nuestros servicios creen aplicaciones Docker y 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 múltiples máquinas virtuales independientes, cada una con su propio servicio, podemos alojar una sola máquina virtual en Vagrant que ejecute una instancia de Docker. Luego usamos Vagrant para configurar y desmantelar la plataforma Docker en sí, 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 ofrece únicamente 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 utilizar un gestor de paquetes como debs o RPM, todo el software se instala como aplicaciones Docker independientes, cada una de las cuales se ejecuta en su propio contenedor.

Docker por sí solo no resuelve todos los problemas. Piense en él como una 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 encontrar un contenedor Docker que pueda ejecutarlo por usted. En este espacio, Kubernetes, recientemente abierto en código fuente por Google, y la tecnología de clúster de CoreOS 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.

Ya hablé antes de las soluciones PaaS. Mi problema con ellas siempre ha sido que a menudo se equivocan en el nivel de abstracción y que las soluciones alojadas en servidores propios se quedan muy por detrás de las soluciones alojadas como Heroku. Docker hace mucho más bien esto 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 sitúa entre las soluciones IaaS y PaaS: el término contenedores como servicio (CaaS) ya se está utilizando para

## describelo

Varias empresas utilizan Docker en producción. Ofrece 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 recomiendo encarecidamente que le dé una oportunidad a Docker.

## Una interfaz de implementación

Independientemente de la plataforma o los artefactos subyacentes que utilices, es fundamental contar con una interfaz uniforme para implementar un servicio determinado. Queremos activar la implementación de un microservicio a pedido en una variedad de situaciones diferentes, desde implementaciones locales para desarrollo y pruebas hasta implementaciones de producción. También queremos 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 la producción porque la implementación utiliza un proceso completamente diferente.

Después de muchos años de trabajar en este ámbito, estoy convencido de que la forma más sensata de activar cualquier implementación es mediante una única llamada de línea de comandos parametrizable. Esto se puede activar mediante scripts, ejecutar mediante la herramienta de integración continua o escribir a mano. He creado scripts de contenedor en una variedad de pilas de tecnología para que esto funcione, desde Windows Batch hasta Bash, pasando por scripts de Python Fabric y más, pero todas las líneas de comandos comparten el mismo formato básico.

Necesitamos saber qué estamos implementando, por lo que necesitamos 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 trabajas localmente, será la versión que esté en tu máquina local. Cuando hagas pruebas, querrás la última compilación verde, que podría ser simplemente el artefacto bendecido más reciente en nuestro repositorio de artefactos. O cuando probemos o diagnostiquemos problemas, es posible que queramos implementar una compilación exacta.

El tercer y último aspecto que debemos conocer es en qué entorno queremos implementar el microservicio. Como hemos comentado anteriormente, la topología de nuestro microservicio puede variar de un entorno a otro, pero no debemos informarnos de ello aquí.

Imaginemos 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:

```
$ desplegar artefacto=catálogo entorno=local versión=local
```

Una vez que he realizado el check-in, nuestro servicio de compilación de CI detecta el cambio y crea un nuevo artefacto de compilación, al que le asigna el número de compilación b456. Como es estándar en la mayoría de las herramientas de CI, este valor se transmite a lo largo del proceso. Cuando se activa nuestra etapa de prueba, la etapa de CI se ejecutará:

```
$ desplegar artefacto=catálogo entorno=ci versión=b456
```

Mientras tanto, nuestro equipo de control de calidad quiere incorporar la última versión del servicio de catálogo a un entorno de prueba integrado para realizar algunas pruebas exploratorias y ayudar con una presentación. Ese equipo corre:

```
$ desplegar artefacto=catálogo entorno=integrated_qa versión=más reciente
```

La herramienta que más he utilizado para esto es Fabric, una biblioteca de Python diseñada para mapear llamadas de línea de comandos a funciones, junto con un buen soporte para manejar tareas como SSH en máquinas remotas. Combínala con una biblioteca de cliente de AWS como Boto y tendrás todo lo que necesitas para automatizar por completo entornos de AWS muy grandes. Para Ruby, Capistrano es similar en algunos aspectos a Fabric, y en Windows podrías llegar muy lejos usando PowerShell.

## Definición de medio ambiente

Claramente, para que esto funcione, necesitamos tener alguna forma de definir cómo se ven nuestros entornos y cómo se ve nuestro servicio en un entorno determinado. Puedes pensar en una definición de entorno como una asignación de un microservicio a recursos de computación, red y almacenamiento. He hecho esto con archivos YAML antes y usé mis scripts para extraer 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. Ejemplo de definición de entorno

desarrollo: nodos:

```
- ami_id:
  ami-e1e1234 tamaño: t1.micro
  credenciales_nombre: eu- ❶
  west-ssh servicios: [catálogo-servicio] región: eu- ❷
  west-1
```

producción:

```
nodos: -
ami_id: ami-e1e1234 tamaño:
  m3.xlarge credentials_name: ❸ ❶
  prod-credentials servicios: [catalog-service] número: 5 ❷
```

❸

❶

Variamos el tamaño de las instancias que utilizamos para que fuera más rentable. ¡No necesitas una máquina de 16 núcleos con 64 GB de RAM para realizar pruebas exploratorias!

❷

Poder especificar diferentes credenciales para diferentes entornos es clave.

Las credenciales para entornos sensibles se almacenaron en diferentes repositorios de código fuente a los que solo personas seleccionadas tendrían acceso.

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 por razones de brevedad.

La información del servicio de catálogo se almacenaba en otro lugar y no variaba de un entorno a otro, como se puede ver en [el Ejemplo 6-2](#).

### Ejemplo 6-2. Ejemplo de definición de entorno

catalog-service:

```
puppet_manifest : catalog.pp conectividad: -
  protocolo: tcp puertos:
    [ 8080, 8081 ] permitidos:
      [ MUNDO ] ❶
```

❶

Este era el nombre del archivo Puppet a ejecutar (en esta situación usamos Puppet solo, pero en teoría podríamos haber admitido sistemas de configuración alternativos).

Obviamente, gran parte del comportamiento aquí se basó en convenciones. Por ejemplo, decidimos normalizar qué puertos usaban los servicios dondequiera que se ejecutaran y configuramos automáticamente平衡adores de carga si un servicio tenía más de una instancia (algo que los ELB de AWS hacen bastante fácil).

Construir un sistema como este requirió una cantidad significativa de trabajo. El esfuerzo suele ser inicial, pero puede ser esencial para gestionar la complejidad de la implementación que tienes. Espero que en el futuro no tengas que hacer esto tú mismo. Terraform es una herramienta muy nueva de Hashicorp, que trabaja en este espacio. Por lo 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 dirigir implementaciones en varias plataformas diferentes, en el futuro podría ser la herramienta perfecta para el trabajo.

## Resumen

Hemos cubierto mucho terreno aquí, por lo que es necesario hacer un resumen. Primero, concéntrese en mantener la capacidad de lanzar un servicio independientemente de otro y asegúrese de que la tecnología que seleccione lo admita. Prefiero mucho más tener un solo repositorio por microservicio, pero estoy más convencido de que necesita una compilación de CI por microservicio si desea implementarlos por separado.

A continuación, si es posible, pase a un único servicio por host o contenedor. Considere tecnologías alternativas como LXC o Docker para que la gestión de las partes móviles sea más barata y sencilla, pero comprenda que, independientemente de la tecnología que adopte, una cultura de automatización es clave para gestionar todo. Automatice todo y, si la tecnología que tiene no lo permite, ¡consiga alguna tecnología nueva! Poder utilizar una plataforma como AWS le brindará enormes beneficios en lo que respecta a la automatización.

Asegúrese de comprender el impacto que sus decisiones de implementación tienen sobre los desarrolladores y asegúrese de que ellos también lo sientan. Crear herramientas que le permitan implementar por su cuenta cualquier servicio determinado en distintos entornos es muy importante y ayudará a los desarrolladores, evaluadores y personal de operaciones por igual.

Por último, si quieras profundizar en este tema, te recomiendo leer *Continuous Delivery* (Addison-Wesley) de Jez Humble y David Farley, que profundiza mucho más en temas como el diseño de pipelines y la gestión de artefactos.

En el próximo capítulo, profundizaremos en un tema que abordamos brevemente aquí: ¿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 nueva herramienta o técnica que lo mejora aún más. Pero aún quedan desafíos en cuanto a cómo probar nuestra funcionalidad de manera eficaz y eficiente cuando abarca un sistema distribuido. Este capítulo analiza los problemas asociados con las pruebas de sistemas más detallados y presenta algunas soluciones para ayudarlo a asegurarse de poder lanzar su nueva funcionalidad con confianza.

Las pruebas abarcan muchos aspectos. Incluso cuando solo hablamos de pruebas automatizadas, hay una gran cantidad de ellas que se deben tener en cuenta. Con los microservicios, hemos agregado otro nivel de complejidad. Comprender qué tipos de pruebas diferentes podemos ejecutar es importante para ayudarnos a equilibrar las fuerzas, a veces opuestas, de poner nuestro software en producción lo más rápido posible frente a asegurarnos de que nuestro software tenga la calidad suficiente.

## Tipos de pruebas

Como consultor, me gusta utilizar algún cuadrante para categorizar el mundo y estaba empezando a preocuparme de que este libro no tuviera uno. Afortunadamente, Brian Marick ideó un fantástico sistema de categorización para pruebas que encaja perfectamente. [La Figura 7-1](#) muestra una variación del cuadrante de Marick del libro Agile Testing (Pruebas ágiles) de Lisa Crispin y Janet Gregory que ayuda a categorizar los diferentes tipos de pruebas.

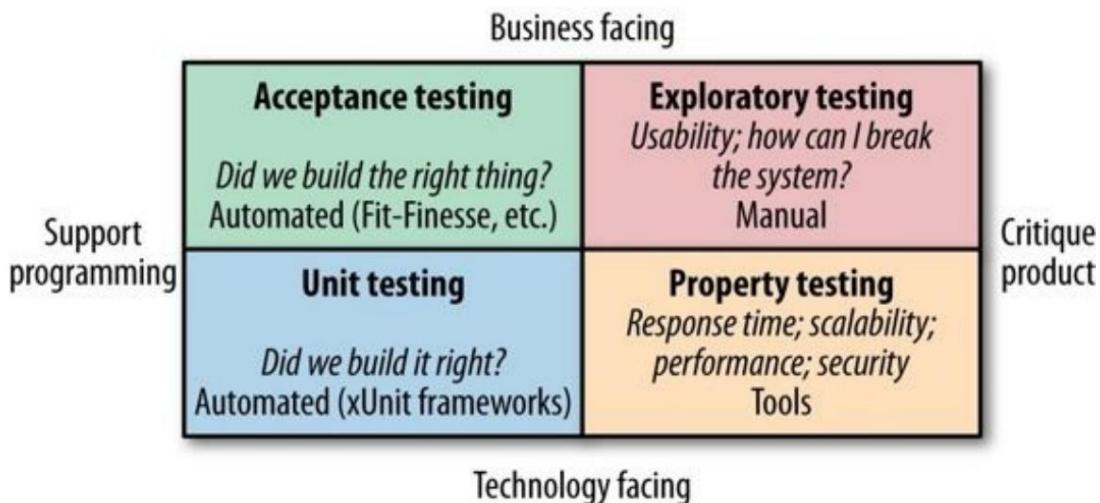


Figura 7-1. Cuadrante de pruebas de Brian Marick. Crispin, Lisa; Gregory, Janet, Agile Testing: A Practical Guide for Testers and Agile Teams, 1.<sup>a</sup> edición, © 2009. Adaptado con autorización de Pearson Education, Inc., Upper Saddle River, NJ.

En la parte inferior, tenemos las pruebas que están orientadas a la tecnología , es decir, las pruebas que ayudan a los desarrolladores a crear el sistema en primer lugar. Las pruebas de rendimiento y las pruebas unitarias de alcance reducido entran en esta categoría, todas ellas normalmente 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 gran alcance y de extremo a extremo, como se muestra en el recuadro de Prueba de aceptación de la parte superior izquierda, o pruebas manuales, como se ejemplifica en las pruebas de usuario realizadas contra un sistema UAT, como se muestra en el recuadro de Prueba exploratoria.

Cada tipo de prueba que se muestra en este cuadrante tiene un lugar. La cantidad exacta de cada prueba que deseé realizar 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. La tendencia reciente ha sido alejarse de cualquier prueba manual a gran escala, en favor de automatizar tanto como sea posible, y ciertamente estoy de acuerdo con este enfoque. Si actualmente realiza una gran cantidad de pruebas manuales, le sugiero que aborde eso 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 pruebas pueden ser muy útiles y sin duda cumplen su función, las diferencias con las pruebas de una arquitectura de microservicios se manifiestan principalmente en el contexto de varios tipos de pruebas automatizadas, por lo que es allí donde centraremos nuestro tiempo.

Pero cuando se trata de pruebas automatizadas, ¿cuántas pruebas de cada tipo queremos? Otro modelo nos resultará muy útil para responder a esta pregunta y entender cuáles podrían ser las diferentes compensaciones.

## Alcance de la prueba

En su libro *Succeeding with Agile* (Addison-Wesley), Mike Cohn describe un modelo llamado la Pirámide de Pruebas para ayudar a explicar qué tipos de pruebas automatizadas necesitas. 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 que debemos buscar. El modelo original de Cohn dividía las pruebas automatizadas en Unidad, Servicio y UI, que puedes ver en [la Figura 7-2](#).

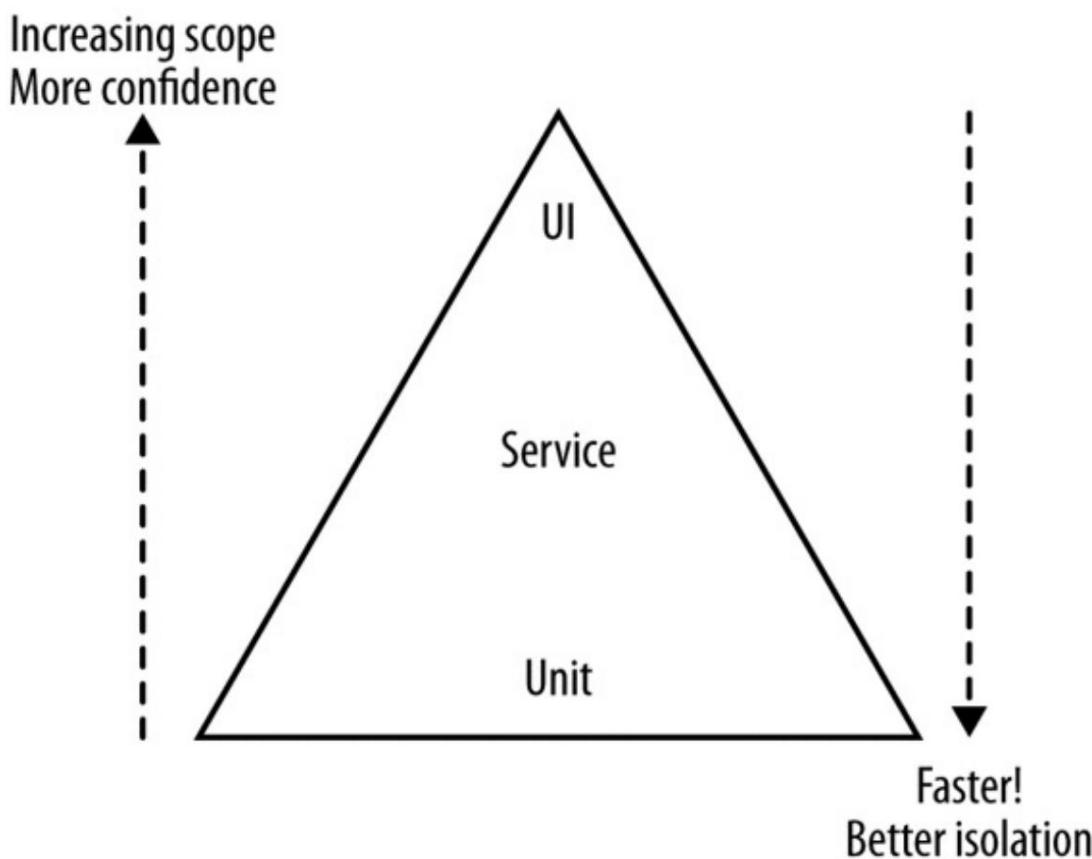


Figura 7-2. Pirámide de pruebas de Mike Cohn. Cohn, Mike, *Succeeding with Agile: Software Development Using Scrum*, 1.<sup>a</sup> edición, © 2010. Adaptado con autorización de Pearson Education, Inc., Upper Saddle River, NJ.

El problema con este modelo es que todos estos términos significan cosas diferentes para distintas personas. El término "servicio" está especialmente sobrecargado y existen muchas definiciones de prueba unitaria. ¿Una prueba es 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 IU pruebas de extremo a extremo , que es lo que haremos a partir de ahora.

Dada la confusión, vale la pena ver qué significan estas diferentes capas.

Veamos un ejemplo práctico. En [la Figura 7-3](#), tenemos nuestra aplicación de soporte técnico y nuestro sitio web principal, que interactúan con nuestro servicio de atención al cliente para recuperar, revisar y editar los datos de los clientes. Nuestro servicio de atención al cliente, a su vez, se comunica con nuestro banco de puntos de fidelidad, donde nuestros clientes acumulan puntos al comprar CD de Justin Bieber. Probablemente. Obviamente, esto es una pequeña parte de nuestro sistema general de tienda de música, pero es una parte lo suficientemente buena para que podamos

Profundizaremos en algunos escenarios diferentes que quizás queramos probar.

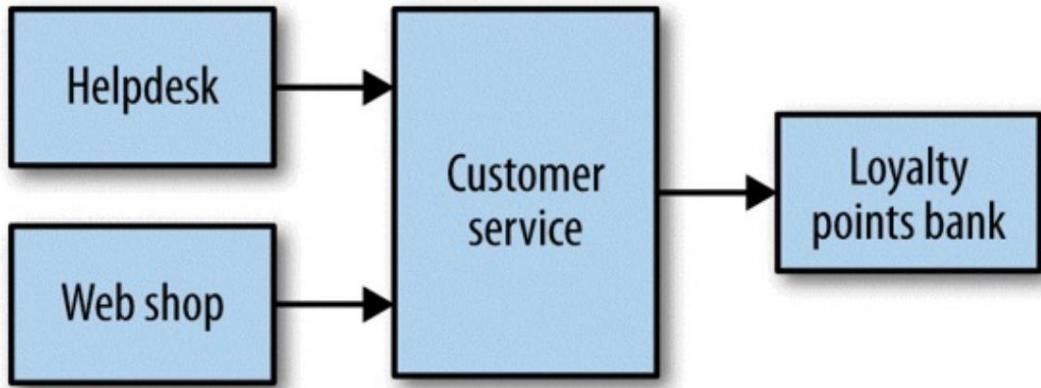


Figura 7-3. Parte de nuestra tienda de música bajo prueba

## Pruebas unitarias

Se trata de pruebas que normalmente prueban una única función o llamada a un método. Las pruebas generadas como efecto secundario del diseño basado en pruebas (TDD) se incluirán 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, se desea una gran cantidad de este tipo de pruebas. Si se hacen bien, son muy, muy rápidas y, en el hardware moderno, se podría esperar ejecutar muchos miles de ellas en menos de un minuto.

Se trata de pruebas que nos ayudan a los desarrolladores y, por lo tanto, estarían orientadas a la tecnología, no al negocio, en la terminología de Marick. También es donde esperamos detectar 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.

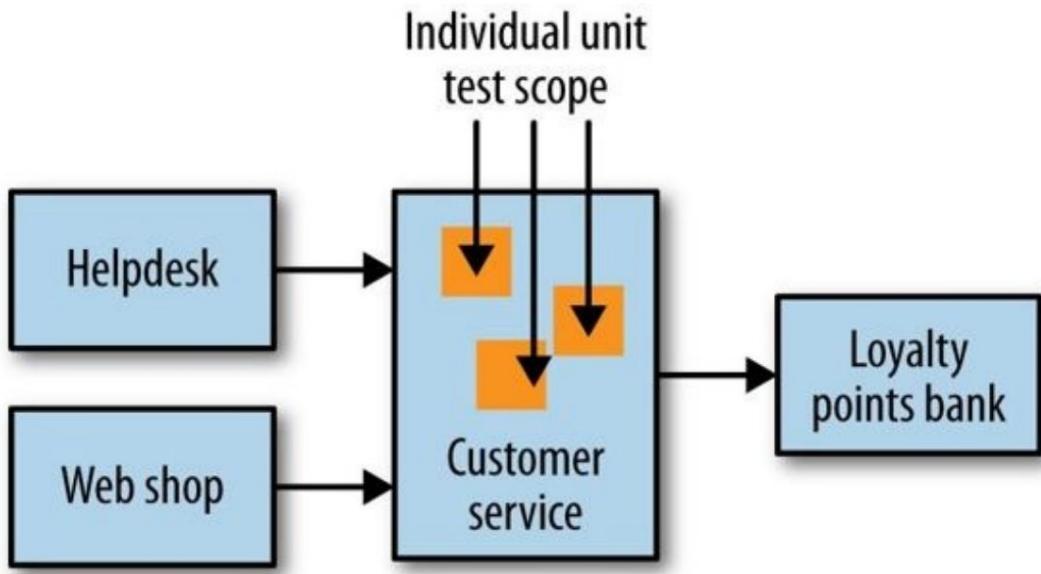


Figura 7-4. Alcance de las pruebas unitarias en nuestro sistema de ejemplo

El objetivo principal de estas pruebas es brindarnos una respuesta rápida sobre si nuestra funcionalidad es buena. Las pruebas pueden ser importantes para respaldar la refactorización del código, lo que nos permite reestructurarlo a medida que avanzamos, sabiendo que nuestras pruebas de alcance reducido nos detectarán 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, podríamos estar probando una colección de clases que proporcionan un servicio a la interfaz de usuario. En el caso de un sistema que comprende varios servicios, una prueba de servicio probaría las capacidades de un servicio individual.

El motivo por el que queremos probar un solo servicio por separado es mejorar el aislamiento de la prueba para que la detección y la solución de problemas sean más rápidas. Para lograr este aislamiento, necesitamos excluir a todos los colaboradores externos de modo que solo el servicio en sí esté dentro del alcance, como se muestra en [la Figura 7-5](#).

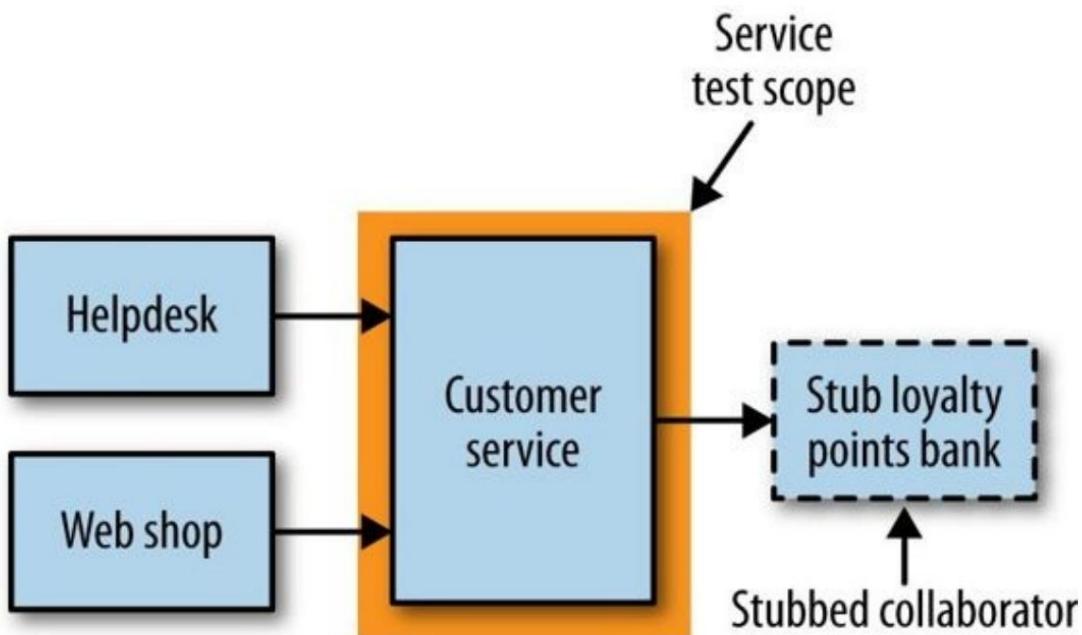


Figura 7-5. Alcance de las pruebas de servicio en nuestro sistema de ejemplo

Algunas de estas pruebas pueden ser tan rápidas como las pruebas pequeñas, pero si decides probar contra una base de datos real o pasar por redes para encontrar colaboradores secundarios, los tiempos de prueba pueden aumentar. También cubren un alcance mayor que una prueba unitaria simple, por lo que cuando fallan puede ser más difícil detectar qué es lo que está mal que con una prueba unitaria. Sin embargo, tienen muchas menos partes móviles y, por lo tanto, son menos frágiles que las pruebas de alcance mayor.

## Pruebas de extremo a extremo

Las pruebas de extremo a extremo son pruebas que se ejecutan en todo el sistema. A menudo, se ejecutan en una interfaz gráfica de usuario a través de un navegador, pero también pueden imitar fácilmente otros tipos de interacción del usuario, como la carga de un archivo.

Estas pruebas cubren una gran cantidad de código de producción, como vemos . Así que cuando pasan, en la Figura 7-6, y te sientes bien: tienes un alto grado de confianza en que el código que se está probando funcionará en producción. Pero este mayor alcance tiene sus desventajas y, como veremos en breve, puede ser muy difícil realizarlas bien en un contexto de microservicios.

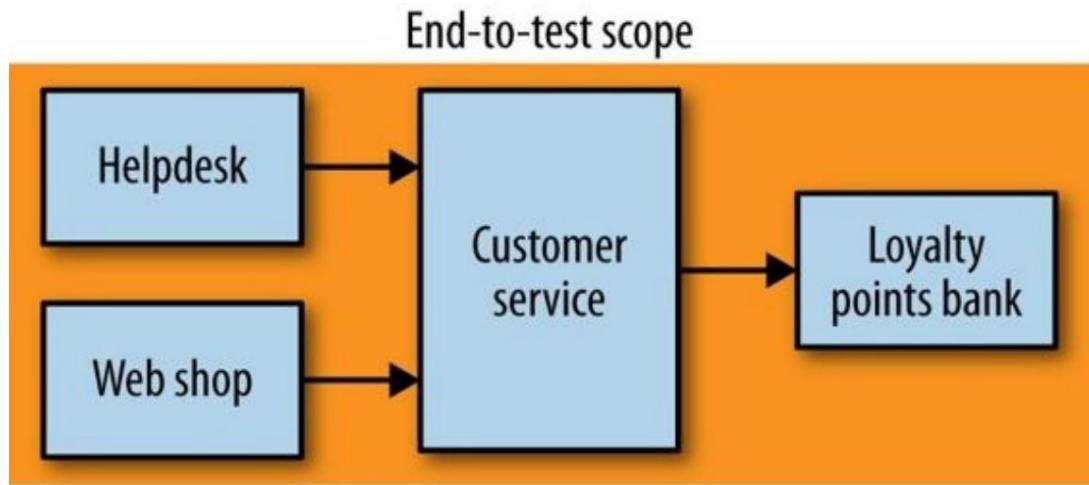


Figura 7-6. Alcance de las pruebas de extremo a extremo en nuestro sistema de ejemplo

## Compensaciones

Al leer la pirámide, lo más importante es que, a medida que se asciende por ella, aumenta el alcance de la prueba y 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 falló.

A medida que se desciende por la pirámide, en general las pruebas se vuelven mucho más rápidas, por lo que obtenemos ciclos de retroalimentación mucho más rápidos. Encontramos funcionalidades defectuosas más rápido, nuestras compilaciones de integración continua son más rápidas y es menos probable que avancemos a una nueva tarea antes de descubrir que hemos roto algo. Cuando fallan esas pruebas de alcance más pequeño, también solemos 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 fallan pruebas de alcance más amplio, como las pruebas de servicio o de extremo a extremo, intentaremos escribir una prueba unitaria rápida para detectar ese problema en el futuro. De esa manera, intentamos mejorar constantemente nuestros ciclos de retroalimentación.

Prácticamente todos los equipos en los que he trabajado han utilizado nombres distintos a los que utiliza Cohn en la pirámide. Independientemente de cómo los llames, la conclusión clave es que necesitarás pruebas de alcance diferente para distintos propósitos.

## ¿Cuántos?

Entonces, si todas estas pruebas tienen ventajas y desventajas, ¿cuántas de cada tipo desea? Una buena regla general es que probablemente desee un orden de magnitud más de pruebas a medida que desciende en la pirámide, pero lo importante es saber que tiene diferentes tipos de pruebas automatizadas y comprender si su saldo actual le da un problema.

Por ejemplo, trabajé en un sistema monolítico en el que 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 (estas últimas eran las que más afectaban a los bucles de retroalimentación), por lo que trabajamos arduamente para reemplazar la cobertura de pruebas con pruebas de alcance más reducido.

Un antipatrón común es lo que a menudo se conoce como cono de nieve o pirámide invertida. Aquí, hay pocas o ninguna prueba de alcance pequeño, mientras que toda la cobertura se encuentra 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 largo período si algo falla.

## Implementación de pruebas de servicio

Implementar pruebas unitarias es una tarea bastante sencilla en términos generales y hay mucha documentación que explica cómo escribirlas. Las pruebas de servicio y de extremo a extremo son las más interesantes.

Nuestras pruebas de servicio quieren probar una parte de la funcionalidad de todo el servicio, pero para aislarnos de otros servicios necesitamos encontrar alguna forma de excluir a todos nuestros colaboradores. Por lo tanto, si quisieramos 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 mencionó anteriormente, queríamos excluir todos los servicios posteriores.

Una de las primeras cosas que hará nuestra compilación de integración continua es crear un artefacto binario para nuestro servicio, por lo que su implementación es bastante sencilla. Pero ¿cómo manejamos la falsificación de los colaboradores posteriores?

Nuestro conjunto de pruebas de servicios debe iniciar servicios auxiliares para cualquier colaborador posterior (o asegurarse de que se estén ejecutando) y configurar el servicio en prueba para que se conecte a los servicios auxiliares. Luego, debemos configurar los servicios auxiliares para que envíen respuestas de vuelta para imitar los servicios del mundo real. Por ejemplo, podríamos configurar el servicio auxiliar para que el banco de puntos de fidelidad devuelva los saldos de puntos conocidos para ciertos clientes.

## Burlarse o tropezar

Cuando hablo de crear un stub para los colaboradores posteriores, me refiero a que creamos un servicio de stub que responde con respuestas predefinidas a las solicitudes conocidas del servicio en prueba. Por ejemplo, podría decirle a mi banco de puntos de stub que cuando se le solicite el saldo del cliente 123, debe devolver 15 000. A la prueba no le importa si el stub se llama 0, 1 o 100 veces. Una variación de esto es usar un mock en lugar de un stub.

Cuando utilizo 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. Implementar este enfoque requiere más inteligencia en los colaboradores falsos que creamos y, si se usa en exceso, puede hacer que las pruebas se vuelvan frágiles. Sin embargo, como se señaló, a un stub no le importa si se lo llama 0, 1 o muchas veces.

A veces, sin embargo, los mocks pueden ser muy útiles para garantizar que se produzcan los efectos secundarios esperados. Por ejemplo, es posible que desee comprobar que, cuando creo un cliente, se establece un nuevo saldo de puntos para ese cliente. El equilibrio entre las llamadas de stub y de mock es delicado y está tan plagado de problemas en las pruebas de servicio como en las pruebas unitarias. En general, sin embargo, utilizo stubs mucho más que mocks para las pruebas de servicio. Para una discusión más profunda de esta disyuntiva, eche un vistazo a *Growing Object-Oriented Software, Guided by Tests*, de Steve Freeman y Nat Pryce (Addison-Wesley).

En general, rara vez utilizo simulacros para este tipo de pruebas, pero resulta útil contar con una herramienta que pueda hacer ambas cosas.

Si bien creo que los stubs y los mocks están bastante bien diferenciados, sé que la distinción puede ser confusa para algunos, especialmente cuando algunas personas incluyen otros términos como fakes, spys y dummies. Martin Fowler llama a todas estas cosas, incluidos los stubs y los mocks, [dobles de prueba](#).

## Un servicio de talón más inteligente

Normalmente, para los servicios de prueba, los he creado yo mismo. He utilizado de todo, desde Apache o Nginx hasta contenedores Jetty integrados o incluso servidores web Python lanzados desde la línea de comandos que se utilizan para lanzar servidores de prueba para dichos casos de prueba. Probablemente he reproducido el mismo trabajo una y otra vez al crear estos stubs. Mi colega de ThoughtWorks, Brandon Bryars, potencialmente nos ha ahorrado a muchos de nosotros una gran cantidad de trabajo con su servidor de prueba/stub 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 que lo llame. Cuando se inicia, le envía comandos que le indican en qué puerto hacer stub, qué protocolo manejar (actualmente se admiten TCP, HTTP y HTTPS, pero se planean más) y qué respuestas debe enviar cuando se envían solicitudes. También admite la configuración de expectativas si desea usarlo como simulación. Puede agregar o eliminar estos puntos finales de stub a voluntad, lo que hace posible que una sola instancia de Mountebank haga stub de más de una dependencia descendente.

Por lo tanto, si queremos ejecutar nuestras pruebas de servicio solo para nuestro servicio de atención al cliente, podemos iniciar el servicio de atención al cliente y una instancia de Mountebank que actúe como nuestro banco de puntos de fidelidad. Y si esas pruebas pasan, ¡puedo implementar el servicio de atención al cliente de inmediato! ¿O no? ¿Qué pasa con los servicios que llaman al servicio de atención al cliente (el servicio de asistencia y la tienda web)? ¿Sabemos si hemos realizado un cambio que pueda afectarlos? Por supuesto, nos hemos olvidado de las pruebas importantes en la parte superior de la pirámide: las pruebas de extremo a extremo.

## Esas complicadas pruebas de extremo a extremo

En un sistema de microservicios, las capacidades que exponemos a través de nuestras interfaces de usuario son entregadas por una serie de servicios. El objetivo 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 en comparación con todo lo que está debajo para brindarnos una descripción general de una gran parte de nuestro sistema.

Por lo tanto, para implementar una prueba de extremo a extremo, necesitamos implementar varios servicios juntos y luego ejecutar una prueba con todos ellos. Obviamente, esta prueba tiene un alcance mucho mayor, lo que genera más confianza en que nuestro sistema funciona. Por otro lado, estas pruebas tienden a ser más lentas y dificultan el diagnóstico de fallas. Profundicemos un poco más en ellas usando nuestro ejemplo anterior para ver cómo pueden encajar estas pruebas.

Imaginemos que queremos lanzar una nueva versión del servicio de atención al cliente. Queremos implementar nuestros cambios en producción lo antes posible, pero nos preocupa haber introducido un cambio que podría afectar al servicio de asistencia o a la tienda web. No hay problema: implementemos todos nuestros servicios juntos y ejecutemos algunas pruebas en el servicio de asistencia y en la tienda web para ver si hemos introducido un error. Ahora bien, un enfoque ingenuo sería simplemente agregar estas pruebas al final de nuestro flujo de trabajo de servicio al cliente, como en [la Figura 7-7](#).

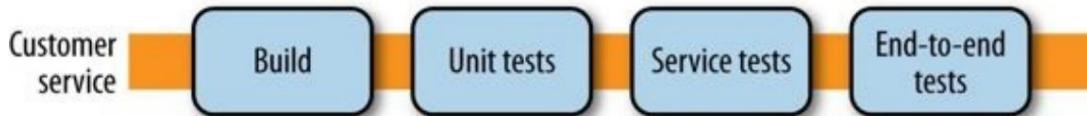


Figura 7-7. ¿Es el enfoque correcto agregar nuestra etapa de pruebas de extremo a extremo?

Hasta aquí, todo bien. Pero la primera pregunta que debemos hacernos es ¿qué versión de los otros servicios deberíamos utilizar? ¿Deberíamos ejecutar nuestras pruebas con las versiones del servicio de asistencia y de la tienda web que están en producción? Es una suposición sensata, pero ¿qué pasa si hay una nueva versión del servicio de asistencia o de la tienda web en cola para su lanzamiento? ¿Qué deberíamos hacer entonces?

Otro problema: si tenemos un conjunto de pruebas de servicio al cliente que implementan muchos servicios y ejecutan pruebas en ellos, ¿qué sucede con las pruebas de extremo a extremo que ejecutan los otros servicios? Si están probando lo mismo, es posible que abarquemos gran parte del mismo terreno y dupliquemos gran parte del esfuerzo para implementar todos esos servicios en primer lugar.

Podemos solucionar ambos problemas de forma elegante si hacemos que varias canalizaciones se distribuyan en una única etapa de prueba de extremo a extremo. En este caso, siempre que se activa una nueva compilación de uno de nuestros servicios, ejecutamos nuestras pruebas de extremo a extremo, como puede verse en [la Figura 7-8](#).

Algunas herramientas de CI con mejor soporte de canalización de compilación habilitarán modelos fan-in como este de manera inmediata.

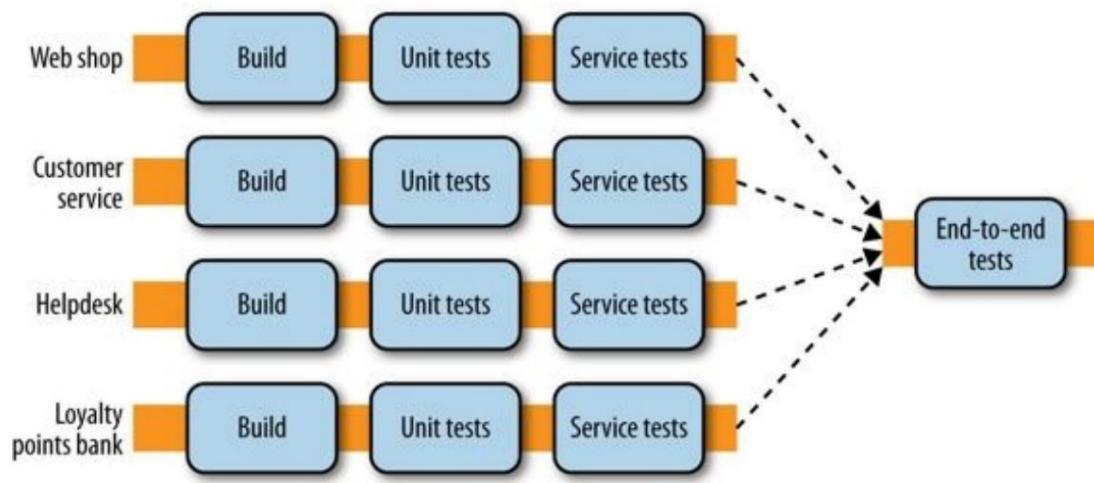


Figura 7-8. Una forma estándar de gestionar pruebas de extremo a extremo en todos los servicios

De modo que, cada vez que cambia alguno de nuestros servicios, ejecutamos las pruebas locales para ese servicio. Si esas pruebas son satisfactorias, activamos nuestras pruebas de integración. Genial, ¿no? Bueno, hay algunos problemas.

## Desventajas de las pruebas de extremo a extremo

Desafortunadamente, las pruebas de extremo a extremo tienen muchas desventajas.

## Pruebas de fragilidad y descamación

A medida que aumenta el alcance de la prueba, también lo hace el número de partes móviles. Estas partes móviles pueden introducir fallas en la prueba que no muestran que la funcionalidad bajo prueba está rota, sino que ha ocurrido algún otro problema. Por ejemplo, si tenemos una prueba para verificar que podemos realizar un pedido de un solo CD, pero estamos ejecutando esa prueba contra cuatro o cinco servicios, si alguno de ellos está inactivo, podríamos obtener una falla que no tenga nada que ver con la naturaleza de la prueba en sí. Del mismo modo, una falla temporal de la red podría hacer que una prueba falle sin decir nada sobre la funcionalidad bajo prueba.

Cuanto más partes móviles haya, más frágiles serán nuestras pruebas y menos deterministas serán. Si tienes pruebas que a veces fallan, pero todo el mundo las vuelve a ejecutar porque pueden volver a pasar más tarde, entonces tienes pruebas inestables. No son solo las pruebas que cubren muchos procesos diferentes las culpables aquí. Las pruebas que cubren la funcionalidad que se ejercita en múltiples subprocesos suelen ser problemáticas, donde un fallo podría significar una condición de carrera, un tiempo de espera o que la funcionalidad está realmente rota. Las pruebas inestables 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 se acumulan los registros y, de repente, nos encontramos con un montón de funcionalidad rota.

Cuando detectamos pruebas defectuosas, es esencial que hagamos todo lo posible por eliminarlas. De lo contrario, comenzamos a perder la fe en un conjunto de pruebas que "siempre falla de esa manera". Un conjunto de pruebas con pruebas defectuosas 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 estén mal que comenzamos a aceptarlas como algo normal <sup>2</sup>. Esta tendencia tan humana significa que y no un problema. Necesitamos encontrar y eliminar estas pruebas lo antes posible antes de comenzar a asumir que las pruebas que fallan están bien.

En “[Erradicar el no determinismo en las pruebas](#)”, Martin Fowler defiende el enfoque de que si tiene pruebas inestables, debe rastrearlas y, si no puede solucionarlas de inmediato, eliminarlas de la suite para poder tratarlas. Vea si puede reescribirlas para evitar que el código de prueba se ejecute en varios subprocesos. Vea si puede hacer que el entorno subyacente sea más estable. Mejor aún, vea si puede reemplazar la prueba inestable con una prueba de alcance más pequeño que tenga menos probabilidades de presentar problemas. En algunos casos, cambiar el software bajo prueba para que sea más fácil de probar también puede ser el camino correcto a seguir.

## ¿Quién escribe estas pruebas?

En el caso de las pruebas que se ejecutan como parte de la cadena de suministro de un servicio específico, el punto de partida sensato es que el equipo propietario de ese servicio debería escribir esas pruebas (hablaremos más sobre la propiedad del servicio en el [Capítulo 10](#)). Pero si consideramos que podríamos tener varios equipos involucrados y que el paso de pruebas de extremo a extremo ahora se comparte de manera efectiva entre los equipos, ¿quién escribe y supervisa estas pruebas?

He visto que aquí se han producido varios antipatrones. Estas pruebas se convierten en una batalla campal, en la que todos los equipos tienen acceso para añadir pruebas sin tener ni idea del estado de todo el conjunto. Esto puede dar lugar a menudo a una explosión de casos de prueba, que a veces acaba en el granizado de pruebas del que hablamos antes. He visto situaciones en las que, como no había una propiedad real y evidente de estas pruebas, se ignoran sus resultados. Cuando fallan, todo el mundo supone que es un problema de otra persona, por lo que no les importa si las pruebas pasan o no.

En ocasiones, las organizaciones reaccionan haciendo que un equipo dedicado escriba estas pruebas. Esto puede ser desastroso. El equipo que desarrolla el software se distancia cada vez más de las pruebas para su código. Los tiempos de ciclo aumentan, ya que los propietarios de servicios terminan esperando que el equipo de pruebas 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 solucionar estas pruebas. Aunque, lamentablemente, sigue siendo un patrón organizacional común, veo que se produce un daño significativo cuando un equipo se distancia de escribir pruebas para el código que escribió en primer lugar.

Conseguir que este aspecto funcione correctamente es realmente difícil. No queremos duplicar esfuerzos ni centralizarlo por completo hasta el punto de que los equipos que crean los servicios estén demasiado alejados de las cosas. El mejor equilibrio que he encontrado es tratar la suite 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 del estado de la suite debe ser compartida entre los equipos que desarrollan los propios servicios. Si quieras 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 aplicado muy pocas veces y nunca sin problemas.

## ¿Cuánto tiempo?

Estas pruebas de extremo a extremo pueden llevar un tiempo. He visto que tardan 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 equipos que realmente organicen sus suites de pruebas de extremo a extremo para reducir la superposición en la cobertura de pruebas o que dediquen suficiente tiempo a hacerlas rápidas.

Esta lentitud, combinada con el hecho de que estas pruebas a menudo pueden ser inestables, puede ser un problema importante. Un conjunto de pruebas que lleva todo el día y que a menudo tiene fallos que no tienen nada que ver con la funcionalidad defectuosa es un desastre. Incluso si su funcionalidad está defectuosa, podría llevarle muchas horas descubrirlo, momento en el que muchos de nosotros ya habríamos pasado a otras actividades, y el cambio de contexto que supone volver a poner nuestro cerebro en marcha para solucionar el problema es doloroso.

Podemos mejorar esto en parte ejecutando pruebas en paralelo (por ejemplo, utilizando herramientas como Selenium Grid). Sin embargo, este enfoque no sustituye a comprender realmente qué es lo que se debe probar y eliminar activamente las pruebas que ya no son necesarias.

Eliminar pruebas es a veces una tarea complicada, y sospecho que tiene mucho en común con la gente que quiere eliminar ciertas medidas de seguridad de los aeropuertos. No importa lo ineficaces que puedan ser las medidas de seguridad, cualquier conversación sobre su eliminación suele ser contrarrestada 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 algo añade frente a la carga que conlleva. También puede ser un difícil equilibrio entre riesgo y recompensa. ¿Te agradecen si eliminas una prueba? Tal vez. Pero sin duda te culparán si una prueba que eliminaste deja pasar un error. Sin embargo, cuando se trata de conjuntos de pruebas de mayor alcance, esto es exactamente lo que necesitamos poder hacer. Si la misma característica está cubierta 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 notoriamente malos. Como resultado, esta selección y gestión inteligente de pruebas de mayor alcance y alta carga se produce con una frecuencia increíblemente escasa. Desear que la gente haga esto con más frecuencia no es lo mismo que lograr que suceda.

## La gran colisión

Los largos ciclos de retroalimentación asociados con las pruebas de extremo a extremo no solo son un problema cuando se trata de la productividad de los desarrolladores. Con un conjunto de pruebas extenso, cualquier falla demora un tiempo en solucionarse, lo que reduce la cantidad de tiempo que se puede esperar que pasen las pruebas de extremo a extremo. Si implementamos únicamente software que ha pasado todas nuestras pruebas con éxito (¡lo cual deberíamos hacer!), esto significa que menos de nuestros servicios llegan al punto de poder implementarse en producción.

Esto puede generar una acumulación de tareas. Mientras se arregla una etapa de prueba de integración que no funciona, se pueden acumular más cambios de los equipos anteriores. Además de que esto puede dificultar la reparación de la compilación, significa que aumenta el alcance de los cambios que se deben implementar. Una forma de resolver esto es no permitir que la gente se registre si las pruebas de extremo a extremo fallan, pero dado el largo tiempo de la suite de pruebas, esto suele ser poco 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 el riesgo de un lanzamiento, más probabilidades hay de que se produzcan errores. 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 están listos.

## La metaversión

Con el paso de prueba de extremo a extremo, es fácil empezar a pensar: "Sé que todos estos servicios en estas versiones funcionan juntos, ¿por qué no implementarlos todos juntos?". Esto rápidamente se convierte en una conversación del tipo: "¿Por qué no usar un número de versión para todo el sistema?". Para citar a Brandon Bryars: "Ahora tienes problemas con la versión 2.1.0".

Al crear versiones conjuntas de 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 vuelve aceptable. Al hacerlo, cedemos una de las principales ventajas de los microservicios: la capacidad de implementar un servicio por sí solo, 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 que estaban bien separados se enredan cada vez más con otros, y nunca te das cuenta porque nunca intentas implementarlos por separado. Terminas con un lío en el que tienes que orquestar la implementación de varios servicios a la vez y, como comentamos anteriormente, este tipo de acoplamiento puede dejarnos en una situación peor que la que tendríamos con una única aplicación monolítica.

Esto es malo.

## Viajes de prueba, no historias

A pesar de las desventajas que acabamos de mencionar, 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 sucede con 3, 4, 10 o 20 servicios? Muy rápidamente, estos conjuntos de pruebas se vuelven enormemente inflados y, en el peor de los casos, pueden resultar en una explosión de tipo cartesiano en los escenarios bajo prueba.

Esta situación empeora si caemos en la trampa de agregar una nueva prueba de principio a fin por cada pieza de funcionalidad que agregamos. Muéstreme una base de código donde cada nueva historia resulte en una nueva prueba de principio a fin y le mostraré un conjunto de pruebas inflado con ciclos de retroalimentación deficientes y enormes superposiciones en la cobertura de pruebas.

La mejor manera de contrarrestar esto es centrarse en un pequeño número de recorridos principales para probar todo el sistema. Cualquier funcionalidad que no esté cubierta en estos recorridos principales debe cubrirse en pruebas que analicen los servicios de forma aislada unos de otros. Estos recorridos deben acordarse mutuamente y ser de propiedad conjunta. En el caso de 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 en número.

Si nos centramos en una pequeña cantidad (y me refiero a una cantidad muy pequeña: de dos dígitos, incluso para sistemas complejos) de pruebas, podemos reducir los inconvenientes de las pruebas de integración, pero no podemos evitarlos todos. ¿Existe una forma mejor?

## Las pruebas impulsadas por el consumidor vienen al rescate

¿Cuál es uno de los problemas clave que intentamos abordar cuando utilizamos las pruebas de integración descritas anteriormente? Intentamos asegurarnos de que, cuando implementamos un nuevo servicio en producción, nuestros cambios no afecten a los consumidores. Una forma de lograrlo sin necesidad de realizar pruebas con 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 hacen correctamente, estas CDC se deben ejecutar como parte de la compilación de CI del productor, lo que garantiza que nunca se implemente si rompe uno de estos contratos. Algo muy importante desde el punto de vista de la retroalimentación de las pruebas es que estas pruebas solo deben ejecutarse contra un único productor de forma aislada, por lo que pueden ser más rápidas y más confiables que las pruebas de extremo a extremo que podrían reemplazar.

Como ejemplo, volvamos a examinar nuestro escenario de servicio al cliente. El servicio al cliente tiene dos consumidores independientes: el servicio de asistencia y la tienda web. Ambos servicios consumidores tienen expectativas sobre cómo se comportará el servicio al cliente. En este ejemplo, se crean dos conjuntos de pruebas: uno para cada consumidor que representa el uso que hacen el servicio de asistencia y la tienda web del servicio al cliente. Una buena práctica en este caso es que alguien de los equipos de productores y consumidores colabore en la creación de las pruebas, por lo que quizás las personas de los equipos de la tienda web y del servicio de asistencia se emparejen con personas del equipo de servicio al cliente.

Dado que estas CDC son expectativas sobre cómo debería comportarse el servicio al cliente, se pueden ejecutar contra el servicio al cliente por sí solo con cualquiera de sus dependencias posteriores eliminadas, como se muestra en [la Figura 7-9](#). Desde el punto de vista del alcance, se encuentran en el mismo nivel de la pirámide de pruebas que las pruebas de servicio, aunque con un enfoque muy diferente, como se muestra en [la Figura 7-10](#). Estas pruebas se centran en cómo un consumidor utilizará el servicio, y el desencadenante si fallan es muy diferente en comparación con las pruebas de servicio. Si una de estas CDC falla durante una compilación del servicio al cliente, se vuelve obvio qué consumidor se verá afectado. En este punto, puede solucionar el problema o iniciar la discusión sobre la introducción de un cambio disruptivo de la manera que discutimos en [el Capítulo 4](#). Entonces, con las CDC, podemos identificar un cambio disruptivo antes de que nuestro software entre en producción sin tener que usar una prueba de extremo a extremo potencialmente costosa.

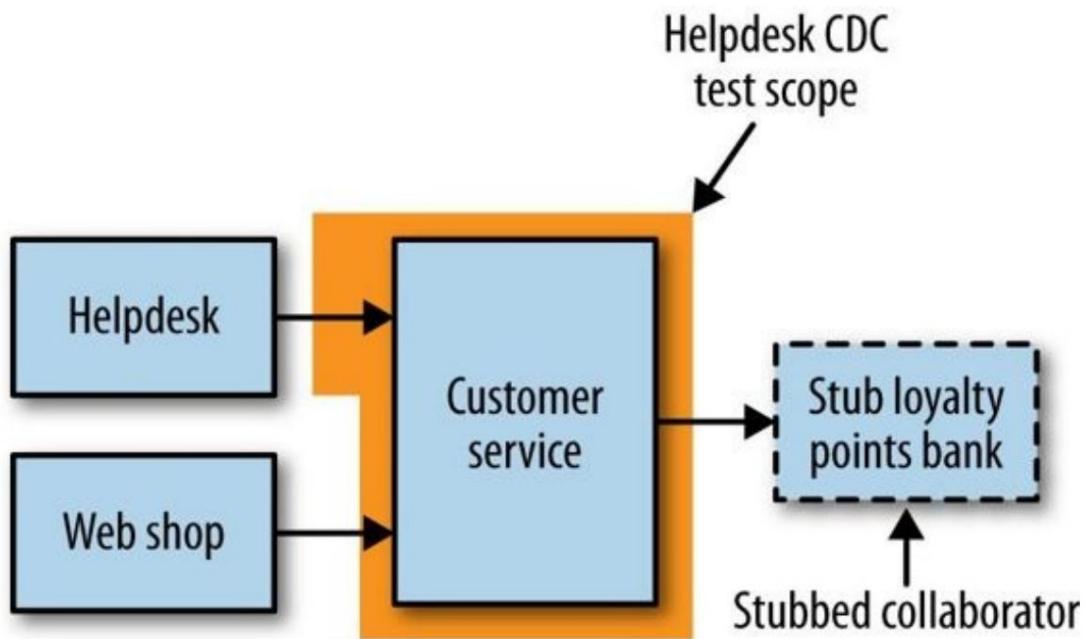


Figura 7-9. Pruebas impulsadas por el consumidor en el contexto de nuestro ejemplo de servicio al cliente

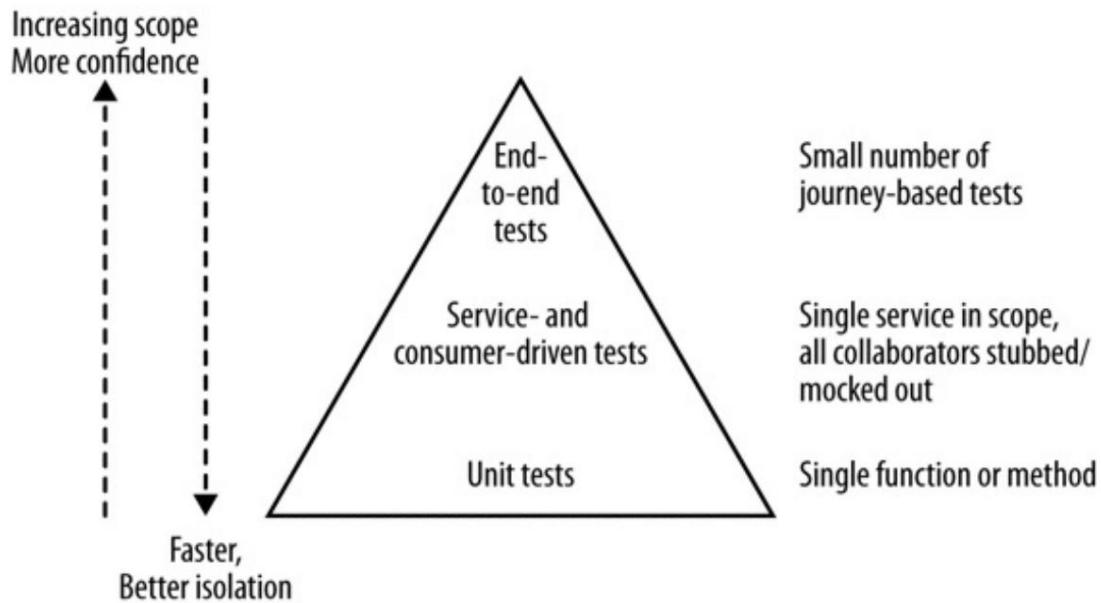


Figura 7-10. Integración de pruebas impulsadas por el consumidor en la pirámide de pruebas

## Pacto

**Pacto** es una herramienta de prueba orientada al consumidor que se desarrolló originalmente en RealEstate.com.au, pero que ahora es de código abierto y Beth Skurrie es la responsable de la mayor parte del desarrollo. Pact, que originalmente era solo para Ruby, 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 utilizando un DSL Ruby. Luego, se inicia un servidor simulado local y se ejecuta esta expectativa en él para crear el archivo de especificación Pact. El archivo Pact es solo una especificación JSON formal; obviamente, se puede codificar manualmente, pero usar la API del lenguaje es mucho más fácil. Esto también le brinda un servidor simulado en ejecución que se puede usar para realizar más pruebas aisladas del consumidor.

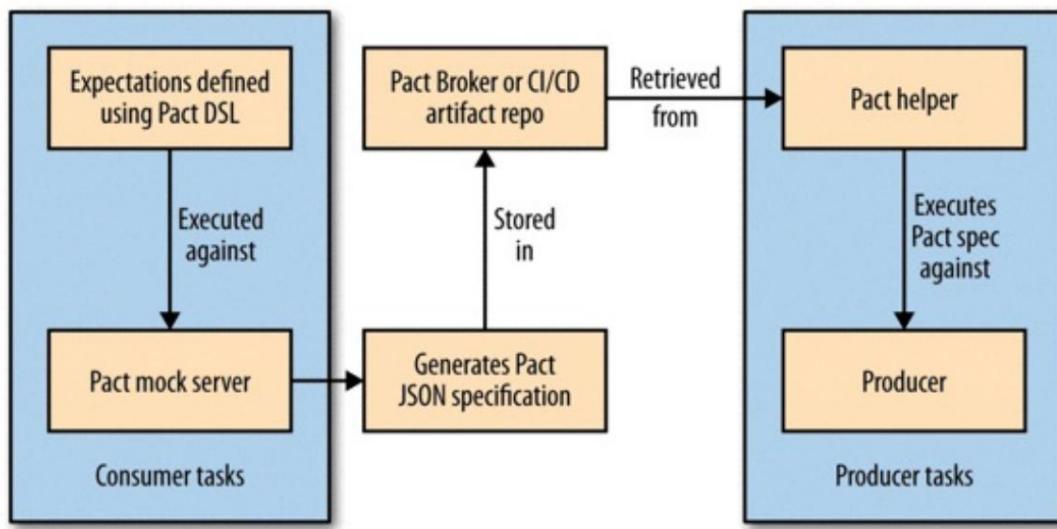


Figura 7-11. Descripción general de cómo Pact realiza pruebas orientadas al consumidor

Del lado del productor, verifica que se cumpla esta especificación del consumidor mediante el uso de la especificación JSON Pact para dirigir las llamadas a tu API y verificar las respuestas. Para que esto funcione, la base de código del productor necesita acceso al archivo Pact. Como comentamos anteriormente en [el Capítulo 6](#), esperamos que tanto el consumidor como el productor estén en compilaciones diferentes. El uso de una especificación JSON independiente del lenguaje es un toque especialmente agradable. Significa que puedes generar la especificación del consumidor mediante un cliente Ruby, pero utilizarla para verificar un productor Java mediante el puerto JVM de Pact.

Como el consumidor crea la especificación JSON Pact, esta debe convertirse en un artefacto al que la compilación del productor tenga acceso. Puede almacenarlo en el repositorio de artefactos de su herramienta CI/CD o bien utilizar Pact Broker, que le permite almacenar múltiples versiones de sus especificaciones Pact. Esto podría permitirle ejecutar sus pruebas de contrato impulsadas por el consumidor contra múltiples versiones diferentes de los consumidores, si desea realizar pruebas contra, por ejemplo, la versión del consumidor en producción y la versión del consumidor que se creó más recientemente.

Resulta confuso que exista un proyecto de código abierto de ThoughtWorks llamado **Pacto**, que también es una herramienta Ruby que se utiliza para pruebas dirigidas al consumidor. Tiene la capacidad de registrar interacciones.

entre el cliente y el servidor para generar las expectativas. Esto hace que escribir contratos impulsados por el consumidor para servicios existentes sea bastante fácil. Con Pacto, una vez generadas, estas expectativas son más o menos estáticas, mientras que con Pact se regeneran las expectativas en el consumidor con cada compilación. El hecho de que se puedan definir expectativas para capacidades que el productor puede que aún no tenga también se adapta mejor a un flujo de trabajo en el que el servicio de producción todavía se está desarrollando (o aún no se ha desarrollado).

## Se trata de conversaciones

En Agile, las historias suelen considerarse un marcador de posición para una conversación. Las CDC son exactamente eso. Se convierten en la codificación de un conjunto de discusiones sobre cómo debería ser una API de servicio y, cuando fallan, se convierten en un punto de activación para tener conversaciones sobre cómo debería evolucionar esa API.

Es importante entender que los CDC requieren una buena comunicación y confianza entre el consumidor y el servicio que los produce. Si ambas partes están en el mismo equipo (¡o la misma persona!), entonces esto no debería ser difícil. Sin embargo, si estás consumiendo un servicio proporcionado por un tercero, es posible que no tengas la frecuencia de comunicación o la confianza necesarias para que los CDC funcionen. En estas situaciones, es posible que tengas que conformarte con pruebas de integración limitadas y de mayor alcance solo en torno al componente no confiable . Alternativamente, si estás creando una API para miles de consumidores potenciales, como con una API de servicio web disponible públicamente, es posible que tengas que desempeñar el papel del consumidor tú mismo (o tal vez trabajar con un subconjunto de tus consumidores) para definir estas pruebas. Romper una gran cantidad de consumidores externos es una muy mala idea, por lo que, en todo caso, ¡la importancia de los CDC aumenta!

## Entonces, ¿debería utilizar pruebas de extremo a extremo?

Como se describió en detalle anteriormente en este capítulo, las pruebas de extremo a extremo tienen una gran cantidad de desventajas que aumentan significativamente a medida que se agregan más partes móviles bajo prueba. Al hablar con personas que han estado implementando microservicios a escala durante un tiempo, aprendí 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 una mejor supervisión. Pero no necesariamente descartan esas pruebas. Terminan usando muchas de esas pruebas de recorrido de extremo a extremo para monitorear el sistema de producción utilizando una técnica llamada supervisión semántica, que analizaremos más en [el Capítulo 8](#).

Puede ver la ejecución de pruebas de extremo a extremo antes de la implementación de producción como si fueran ruedas de entrenamiento. Mientras aprende cómo funcionan los CDC y mejora sus técnicas de implementación y monitoreo de producción, estas pruebas de extremo a extremo pueden formar una red de seguridad útil, donde puede sacrificar tiempo de 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.

De manera similar, es posible que trabaje en un entorno en el que el deseo de aprender en producción sea bajo y la gente prefiera trabajar lo más duro posible para eliminar los defectos 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 solución eficaces en la producción, esta puede ser una decisión sensata.

Obviamente, usted tendrá una mejor comprensión del perfil de riesgo de su propia organización que yo, pero lo desafiaría a pensar detenidamente sobre cuántas pruebas de extremo a extremo realmente necesita realizar.

## Pruebas después de la producción

La mayoría de las pruebas se realizan antes de que el sistema entre en producción. Con nuestras pruebas, definimos una serie de modelos con los que esperamos demostrar si nuestro sistema funciona y se comporta como nos gustaría, tanto funcionalmente como no funcionalmente. Pero si nuestros modelos no son perfectos, nos encontraremos con problemas cuando nuestros sistemas se utilicen de forma agresiva. Los errores se introducen en producción, se descubren nuevos modos de fallo y nuestros usuarios utilizan el sistema de formas que nunca podríamos esperar.

Una reacción a esto suele ser definir más y más pruebas y refinar nuestros modelos para detectar más problemas de forma temprana y reducir la cantidad de problemas que encontramos con nuestro sistema de producción en funcionamiento. Sin embargo, en cierto punto tenemos que aceptar que con este enfoque obtenemos rendimientos decrecientes. Con las pruebas previas a la implementación, no podemos reducir a cero la probabilidad de falla.

## Separación entre implementación y lanzamiento

Una forma de detectar más problemas antes de que ocurran es ampliar el ámbito de ejecución de las pruebas más allá de los pasos previos a la implementación tradicionales. 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 contra 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 cualquier microservicio determinado (y debería hacerlo), este comando debería ejecutar las pruebas de humo automáticamente.

Otro ejemplo de esto es lo que se denomina implementación azul/verde. Con la implementación azul/verde, tenemos dos copias de nuestro software implementadas a la vez, pero solo una versión recibe solicitudes reales.

Consideremos un ejemplo simple, que se ve en [la Figura 7-12](#). En producción, tenemos activa la versión v123 del servicio de atención al cliente. Queremos implementar una nueva versión, v456. La implementamos junto con la v123, pero no dirigimos ningún tráfico hacia ella. En cambio, 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 habitual mantener la versión anterior durante un breve período de tiempo, lo que permite una rápida recuperación si se detecta algún error.

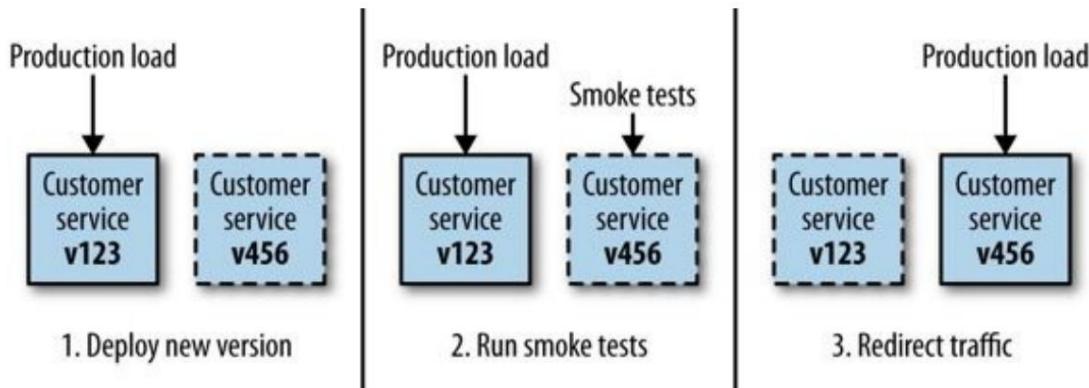


Figura 7-12. Uso de implementaciones azules/verdes para separar la implementación del lanzamiento

Para implementar la implementación azul/verde se requieren algunas cosas. Primero, debe poder dirigir el tráfico de producción a diferentes hosts (o conjuntos de hosts). Puede hacerlo modificando 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 vuelve bueno en esto, todo el proceso se puede automatizar por completo, y la implementación completa o la reversión se realizan 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 funcionamiento mientras realizamos nuestro lanzamiento

Reducimos enormemente el tiempo de inactividad asociado con la liberación de nuestro software. Según el mecanismo que se utilice para implementar la redirección del tráfico, el cambio entre versiones puede ser completamente invisible para el cliente, lo que nos permite implementar implementaciones sin tiempo de inactividad.

Existe otra técnica que vale la pena analizar brevemente aquí, que a veces se confunde con las implementaciones azul/verde, ya que puede utilizar algunas de las mismas implementaciones técnicas. Se la conoce como liberación canaria.

## Liberación de canarios

Con la liberación de Canary, estamos verificando nuestro software recientemente implementado dirigiendo cantidades de tráfico de producción contra el sistema para ver si funciona como se espera. "El rendimiento esperado" puede abarcar una serie de aspectos, tanto funcionales como no funcionales. Por ejemplo, podríamos comprobar que un servicio recién implementado responde a las solicitudes en un plazo de 500 ms, o que vemos las mismas tasas de error proporcionales entre el servicio nuevo y el antiguo. Pero se podría ir más allá. Imaginemos que hemos lanzado una nueva versión del servicio de recomendaciones. Podríamos ejecutar ambos en paralelo, pero ver si las recomendaciones generadas por la nueva versión del servicio dan como resultado tantas ventas como las esperadas, para asegurarnos de que no hemos lanzado un algoritmo que no sea el óptimo.

Si la nueva versión es mala, puedes revertirla rápidamente. Si es buena, puedes impulsar cantidades cada vez mayores de tráfico a través de la nueva versión. La versión Canary se diferencia de la versión azul/verde en que puedes esperar que las versiones coexistan durante más tiempo y, a menudo, variarás las cantidades de tráfico.

Netflix utiliza este enfoque de forma generalizada. Antes del lanzamiento, se implementan nuevas versiones del servicio junto con un clúster de referencia que representa la misma versión que la de producción. Luego, Netflix ejecuta un subconjunto de la carga de producción durante una cantidad de horas tanto con la nueva versión como con la versión de referencia, y evalúa ambas. Si la prueba canary pasa, la empresa procede a implementar la producción por completo.

Al considerar la liberación de Canary, debe decidir si va a desviar una parte de las solicitudes de producción a Canary o simplemente copiar la carga de producción. Algunos equipos pueden copiar el tráfico de producción y dirigirlo a su Canary. De esta manera, las versiones de producción y Canary existentes pueden ver exactamente las mismas solicitudes, pero solo los resultados de las solicitudes de producción se ven externamente. Esto le permite hacer una comparación en paralelo y, al mismo tiempo, eliminar la posibilidad de que una solicitud del cliente pueda ver una falla en Canary.

Sin embargo, el trabajo de monitorear el tráfico de producción puede ser complejo, especialmente si los eventos o solicitudes que se reproducen no son idempotentes.

La implementación Canary es una técnica poderosa que puede ayudarlo a verificar nuevas versiones de su software con tráfico real, al mismo tiempo que le brinda herramientas para administrar el riesgo de lanzar una versión incorrecta. Sin embargo, requiere una configuración más compleja que la implementación azul/verde y un poco más de reflexión. Es posible que deba esperar que coexistan diferentes versiones de sus servicios durante más tiempo que con la implementación azul/verde, por lo que es posible que deba utilizar 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 de tráfico para tener más confianza en que su versión funciona. Si ya maneja implementaciones azul/verde, es posible que ya tenga algunos de los componentes básicos.

## ¿Tiempo medio de reparación frente al tiempo medio entre fallos?

De modo que, al analizar técnicas como la implementación azul/verde o la liberación controlada, encontramos una forma de realizar pruebas más cerca de la producción (o incluso en ella), y también creamos herramientas que nos ayudan a gestionar una falla si ocurre. El uso de estos enfoques es un reconocimiento tácito de que no podemos detectar y detectar todos los problemas antes de lanzar realmente nuestro software.

A veces, dedicar el mismo esfuerzo a mejorar la corrección de una versión puede ser mucho más beneficioso que agregar más pruebas funcionales automatizadas. En el mundo de las operaciones web, esto se conoce a menudo como el equilibrio entre optimizar el 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 combinadas con un buen monitoreo (que analizaremos en [el Capítulo 8](#)), como las implementaciones azul/verde. Si podemos detectar un problema en producción de manera temprana y revertirlo pronto, 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, este equilibrio entre MTBF y MTTR variará, y gran parte de esto radica en comprender el verdadero impacto de una falla en un entorno de producción.

Sin embargo, la mayoría de las organizaciones que veo que dedican tiempo a crear conjuntos de pruebas funcionales a menudo dedican poco o ningún esfuerzo a mejorar la supervisión o la recuperación de errores. 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 otras disyuntivas además del MTBF y el MTTR. Por ejemplo, si está intentando averiguar si alguien realmente usará su software, puede tener mucho más sentido lanzar algo ahora, para probar la idea o el modelo de negocios antes de desarrollar un software sólido. En un entorno en el que este es 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 producción. En estas situaciones, puede ser bastante sensato evitar por completo las pruebas antes de la producción.

# Pruebas multifuncionales

La mayor parte de este capítulo se ha centrado en probar elementos específicos de la funcionalidad y en cómo esto difiere cuando se prueba un sistema basado en microservicios. Sin embargo, hay otra categoría de pruebas que es importante analizar. Los requisitos no funcionales son un término general que se utiliza para describir las características que presenta el sistema y que no se pueden implementar como una función normal. Incluyen aspectos como la latencia aceptable de una página web, la cantidad de usuarios que debe admitir un sistema, cuán accesible debe ser la interfaz de usuario para personas con discapacidades o cuán seguros deben ser los datos de los clientes.

El término no funcional nunca me convenció. Algunas de las cosas que se engloban bajo este término parecen de naturaleza muy funcional. Una de mis colegas, Sarah Taraporewalla, acuñó la frase requisitos multifuncionales (CFR), que prefiero mucho. Habla más del hecho de que estos comportamientos del sistema realmente solo surgen como resultado de mucho trabajo transversal.

Muchos de los CFR, si no la mayoría, solo se pueden cumplir en la producción. Dicho esto, podemos definir estrategias de prueba que nos ayuden 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, que analizaremos con más profundidad en breve.

En el caso de algunos CFR, es posible que desee realizar un seguimiento de ellos a nivel de servicio individual. Por ejemplo, puede decidir que la durabilidad del servicio que necesita de su servicio de pago es significativamente mayor, pero está satisfecho con un mayor tiempo de inactividad de su servicio de recomendación de música, sabiendo que su negocio principal puede sobrevivir si no puede recomendar artistas similares a Metallica durante unos 10 minutos. 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 relacionadas con 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 extremo a extremo, escriba una prueba de alcance más pequeño para ayudarlo a detectar el problema en el futuro. Otros CFR se adaptan a pruebas más rápidas con bastante facilidad. Recuerdo haber trabajado en un proyecto en el que insistimos 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 estuvieran los controles adecuados se podía hacer muy rápidamente sin la necesidad de realizar viajes de ida y vuelta en red.

Con demasiada frecuencia, las consideraciones sobre los CFR llegan demasiado tarde. Recomiendo encarecidamente que consulte sus CFR lo antes posible y los revise periódicamente.

## 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 antes 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 una cantidad correspondiente de llamadas a la base de datos. Todo esto puede reducir 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 potencialmente genera 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 pruebas se retrasa es porque inicialmente no hay suficiente sistema para probar. Entiendo este problema, pero con demasiada frecuencia conduce a postergar el problema, y las pruebas de rendimiento a menudo solo se realizan justo antes de que comience a funcionar por primera vez, ¡si es que se realizan! No caiga 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 servicios individuales, pero comience con pruebas que verifiquen los recorridos principales en su sistema. Es posible que pueda realizar pruebas de recorrido de extremo a extremo y simplemente ejecutarlas en volumen.

Para generar resultados que valgan la pena, a menudo necesitará ejecutar escenarios determinados con cantidades de clientes simulados que aumenten gradualmente. 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 con la producción lo más posible, para garantizar que los resultados que vea sean indicativos del rendimiento que puede esperar en los sistemas de producción. Esto puede significar que necesitará adquirir un volumen de datos más parecido al de producción y es posible que necesite más máquinas para que coincidan con la infraestructura, tareas que pueden ser desafiantes.

Incluso si tiene dificultades para lograr que el entorno de rendimiento sea realmente similar al de producción, las pruebas pueden ser útiles para detectar cuellos de botella. Solo tenga en cuenta que puede obtener falsos negativos o, peor aún, falsos positivos.

Debido al tiempo que lleva ejecutar pruebas de rendimiento, no siempre es posible ejecutarlas en cada registro. Es una práctica común ejecutar un subconjunto todos los días y un conjunto más grande todas las semanas. Sea cual sea el enfoque que elija, asegúrese de ejecutarlas con la mayor regularidad posible. Cuanto más tiempo pase sin ejecutar pruebas de rendimiento, más difícil será rastrear al culpable. Los problemas de rendimiento son especialmente difíciles de resolver, por lo que si puede reducir la cantidad de confirmaciones que necesita revisar para detectar un problema recién introducido, su vida será mucho más sencilla.

¡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 han comprobado los números. A menudo, esto se debe a que la gente no sabe cómo es un buen resultado. Realmente necesitas tener objetivos. De esta manera, puedes hacer que la compilación se ponga roja.

o verde según los resultados, siendo una compilación roja (fallida) un claro llamado a la acción.

Las pruebas de rendimiento deben realizarse en conjunto con el monitoreo del rendimiento real del sistema (que analizaremos más en [el Capítulo 8](#)) y, idealmente, deberían utilizarse las mismas herramientas en el entorno de pruebas de rendimiento para visualizar el comportamiento del sistema que las que se utilizan en producción. Esto puede facilitar mucho la comparación de elementos similares.

## Resumen

En resumen, lo que he esbozado aquí es un enfoque holístico de las pruebas que espero que te sirva de orientación general sobre cómo proceder cuando pruebes tus propios sistemas. Para reiterar los conceptos básicos:

- Optimice para obtener comentarios rápidos y separe los tipos de pruebas según corresponda.
- Evite la necesidad de realizar 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 focales para las conversaciones entre equipos.
- Intente comprender la compensación entre poner más esfuerzo en las pruebas y detectar problemas más rápidamente en la producción (optimizar para MTBF versus MTTR).

Si está interesado en leer más sobre pruebas, le recomiendo Agile Testing 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 producción, pero también debemos 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.

<sup>2</sup> Diane Vaughan, La decisión de lanzar el 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, analizaremos los desafíos asociados con el monitoreo y la identificación de problemas en nuestros sistemas detallados y describiré algunas de las cosas que puede hacer para tener todo bajo control.

Imagínese la escena. Es una tranquila tarde de viernes y el equipo está deseando irse temprano al bar para empezar un fin de semana lejos del trabajo. De repente, llegan los correos electrónicos. ¡El sitio web no funciona bien! Twitter está en llamas con los fallos de su empresa, su jefe le está dando la lata y las perspectivas de un fin de semana tranquilo se desvanecen.

¿Qué es lo primero que necesitas saber? ¿Qué diablos salió mal?

En el mundo de las aplicaciones monolíticas, al menos tenemos un lugar muy obvio para comenzar nuestras investigaciones. ¿Sitio web lento? Es el monolito. ¿Sitio web que da errores extraños? Es el monolito. ¿CPU al 100%? Monolito. ¿Olor a quemado? Bueno, ya entiendes la idea. Tener un único punto de falla también hace que la investigación de fallas sea algo más simple.

Ahora pensemos en nuestro propio sistema basado en microservicios. Las capacidades que ofrecemos a nuestros usuarios se proporcionan desde múltiples servicios pequeños, algunos de los cuales se comunican con otros 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 de la monitorización, tenemos un problema más complejo entre manos.

Ahora tenemos varios servidores que supervisar, varios archivos de registro que examinar y varios lugares donde la latencia de la red podría causar problemas. ¿Cómo abordamos esto? Necesitamos darle sentido a lo que de otro modo podría ser un caos y un enredo, lo último con lo que cualquiera de nosotros quiere lidiar un viernes por la tarde (o en cualquier otro momento, ¡por cierto!).

La respuesta es bastante sencilla: controlar los detalles pequeños y utilizar la agregación para ver el panorama general. Para ver cómo, comenzaremos con el sistema más simple que podamos: un solo nodo.

## Un solo servicio, un solo servidor

La figura 8-1 presenta una configuración muy sencilla: un host que ejecuta un servicio. Ahora debemos supervisarlo para saber cuándo algo sale mal y poder solucionarlo. ¿Qué debemos buscar entonces?

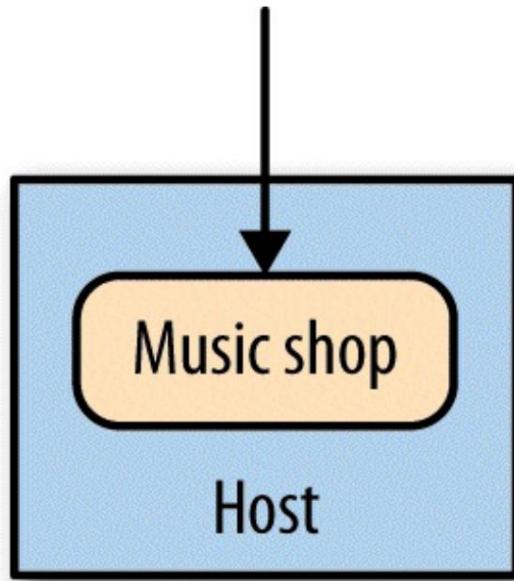


Figura 8-1. Un solo servicio en un solo host

Primero, queremos monitorear el host en sí. CPU, memoria: todas estas cosas son útiles.

Queremos saber cuáles deberían ser cuando todo está en orden, para poder alertar cuando se salgan de los límites. Si queremos ejecutar nuestro propio software de monitoreo, podemos usar algo como Nagios para hacerlo, o bien usar un servicio alojado como New Relic.

A continuación, querremos tener acceso a los registros desde el propio servidor. Si un usuario informa de un error, estos registros deberían detectarlo y, con suerte, indicarnos cuándo y dónde se produjo el error. En este punto, con nuestro único host, probablemente podamos arreglárnoslas con solo iniciar sesión en el host y usar herramientas de línea de comandos para escanear el registro. Incluso podemos avanzar y usar logrotate para quitar del medio los registros antiguos y evitar que ocupen todo nuestro espacio en disco.

Por último, es posible que queramos supervisar la propia aplicación. Como mínimo, es una buena idea supervisar el tiempo de respuesta del servicio. Probablemente podrá hacerlo consultando los registros que provienen de un servidor web que se encuentra frente a su servicio o, tal vez, del propio servicio. Si nos ponemos muy avanzados, es posible que queramos realizar un seguimiento de la cantidad de errores que informamos.

El tiempo pasa, las cargas aumentan y nos vemos en la necesidad de escalar...

## Un solo servicio, varios servidores

Ahora tenemos varias 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.

Ahora las cosas se ponen un poco más complicadas. Queremos seguir controlando las mismas cosas que antes, pero debemos hacerlo de forma que podamos aislar el problema. Cuando el uso de la CPU es alto, ¿se trata de un problema que vemos 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 en sí tiene el problema (quizás un proceso del sistema operativo no autorizado)?

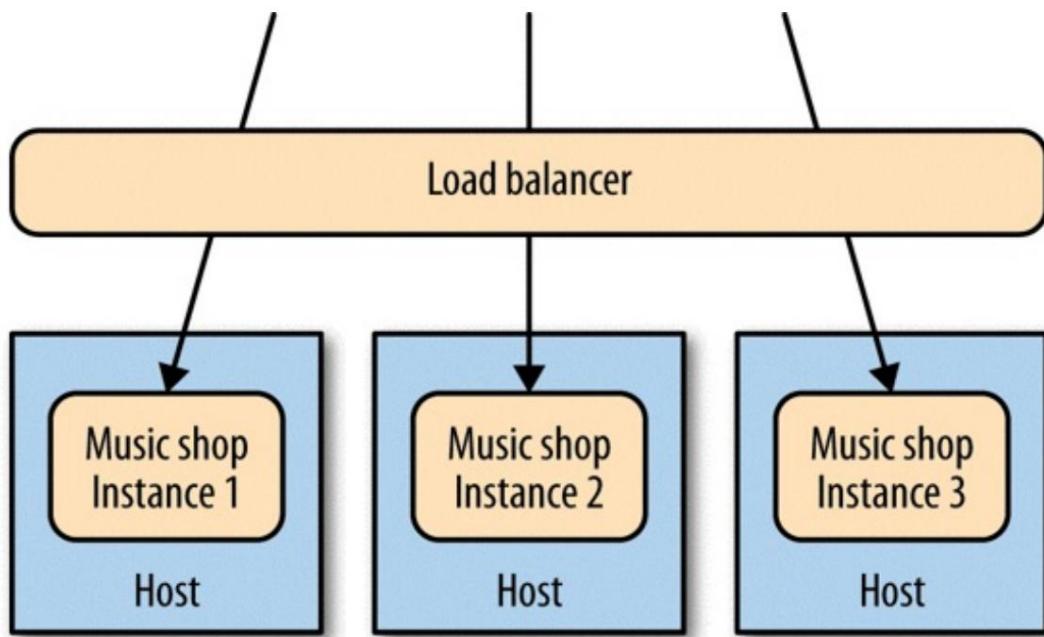


Figura 8-2. Un único servicio distribuido en varios hosts

En este punto, todavía queremos hacer un seguimiento de las métricas a nivel de host y generar alertas al respecto, pero ahora queremos ver cuáles son en todos los hosts, así como en los hosts individuales. En otras palabras, queremos agregarlos y poder seguir desglosándolos. Nagios nos permite agrupar nuestros hosts de esta manera; hasta ahora, todo bien. Un enfoque similar probablemente será suficiente para nuestra aplicación.

Luego tenemos nuestros registros. Si nuestro servicio se ejecuta en más de un servidor, probablemente nos cansaremos de iniciar sesión en cada servidor para verlo. Sin embargo, si solo tenemos unos pocos hosts, podemos usar herramientas como multiplexores ssh, que nos permiten ejecutar los mismos comandos en varios hosts.

Un monitor grande y un grep "Error" app.log más tarde, y podremos encontrar a nuestro culpable.

Para tareas como el seguimiento del tiempo de respuesta, podemos obtener parte de la agregación de forma gratuita mediante el seguimiento en el propio balanceador de carga. Pero también necesitamos realizar un seguimiento del balanceador de carga, por supuesto; si este se comporta 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. Con suerte, cuando lleguemos aquí, tendremos al menos una idea de eso...

## Múltiples servicios, múltiples servidores

En la Figura 8-3, las cosas se ponen mucho más interesantes. Varios servicios colaboran para proporcionar capacidades a nuestros usuarios y esos servicios se ejecutan en varios hosts, ya sean físicos o virtuales. ¿Cómo se encuentra el error que se busca en miles de líneas de registros en varios hosts? ¿Cómo se determina si un servidor se está comportando mal o si se trata de un problema sistemático? ¿Y cómo se rastrea un error encontrado en lo profundo de una cadena de llamadas entre varios hosts y se determina qué lo causó?

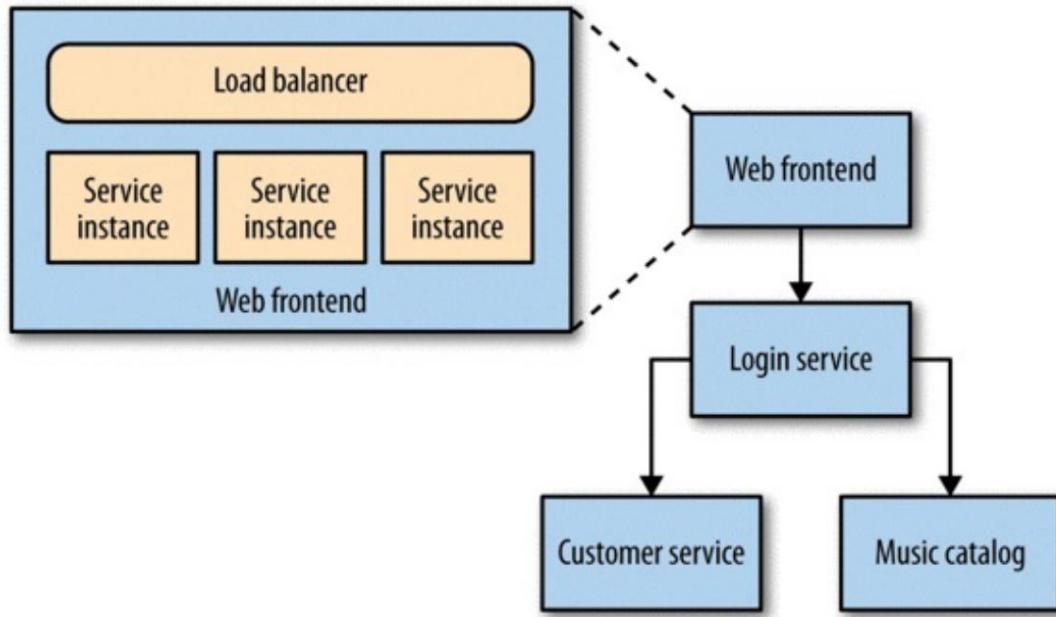


Figura 8-3. Múltiples servicios colaboradores distribuidos en varios hosts

La respuesta es la recopilación y agregación central de toda la información que podamos conseguir, desde registros hasta métricas de aplicaciones.

## Registros, registros y más registros...

Ahora, la cantidad de hosts en los que estamos funcionando 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 cambio, buscamos usar subsistemas especializados para capturar nuestros registros y ponerlos a disposición de manera centralizada. Un ejemplo de esto es [logstash](#), que puede analizar múltiples formatos de archivos de registro y enviarlos a sistemas posteriores para una mayor investigación.

[Kibana](#) es un sistema respaldado por ElasticSearch para ver registros, que se ilustra en [la Figura 8-4](#). Puede utilizar una sintaxis de consulta para buscar en los registros, lo que le permite hacer cosas como restringir rangos de fecha y hora o utilizar expresiones regulares para encontrar cadenas coincidentes. Kibana incluso puede generar gráficos a partir de los registros que le envía, lo que le permite ver de un vistazo cuántos errores se han generado a lo largo del tiempo, por ejemplo.

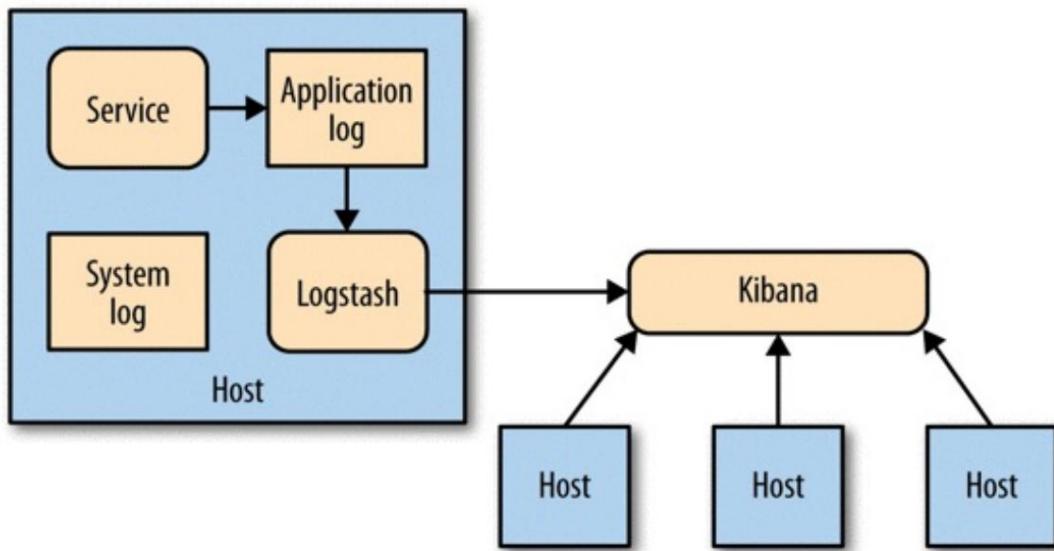


Figura 8-4. Uso de Kibana para ver registros agregados

## Seguimiento de métricas en múltiples servicios

Al igual que con el desafío de mirar los registros de diferentes hosts, necesitamos buscar mejores formas de recopilar y ver nuestras métricas. Puede ser difícil saber qué es lo que está bien cuando miramos las métricas de un sistema más complejo. Nuestro sitio web está viendo casi 50 códigos de error HTTP 4XX por segundo. ¿Eso es malo? La carga de la CPU en el servicio de catálogo ha aumentado un 20% desde el almuerzo; ¿algo salió 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, aprovisionaremos 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. Queremos 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 determinado, o incluso para una sola instancia de ese servicio. Eso significa que necesitaremos poder asociar metadatos con la métrica para permitirnos inferir esto.

estructura.

Graphite es uno de esos sistemas que facilita mucho esta tarea. Expone una API muy simple y permite enviar métricas en tiempo real. Luego permite consultar esas métricas para generar gráficos y otras visualizaciones para ver qué está sucediendo. La forma en que maneja el volumen también es interesante. Efectivamente, se configura de manera que se reduzca la resolución de las métricas más antiguas para garantizar que los volúmenes no sean demasiado grandes. Por ejemplo, podría registrar la CPU de mis hosts una vez cada 10 segundos durante los últimos 10 minutos, luego una muestra agregada cada minuto durante el último día, hasta llegar a una muestra cada 30 minutos durante los últimos años. De esta manera, se puede almacenar información sobre cómo se ha comportado el sistema durante un largo período de tiempo sin necesidad de grandes cantidades de almacenamiento.

Graphite también le permite agregar muestras o desglosar una sola serie, de modo que pueda ver el tiempo de respuesta de todo el sistema, un grupo de servicios o una sola instancia. Si Graphite no le resulta útil por cualquier motivo, asegúrese de obtener capacidades similares en cualquier otra herramienta que seleccione. Y, por supuesto, asegúrese de poder acceder a los datos sin procesar para proporcionar sus propios informes o paneles de control si los necesita.

Otro beneficio clave de comprender sus tendencias es cuando se trata de planificar la capacidad. ¿Estamos llegando a nuestro límite? ¿Cuánto tiempo pasará hasta que necesitemos más hosts? En el pasado, cuando incorporábamos hosts físicos, esto solía ser un trabajo anual. En la nueva era de la informática a pedido que ofrecen los proveedores de infraestructura como servicio (IaaS), ahora podemos aumentar o reducir la escala en minutos, o incluso segundos. Esto significa que, si comprendemos nuestros patrones de uso, podemos asegurarnos de tener la infraestructura suficiente para satisfacer nuestras necesidades. Cuanto más inteligentes seamos a la hora de hacer un seguimiento de nuestras tendencias y saber qué hacer con ellas, más rentables y receptivos serán nuestros sistemas.

## Métricas de servicio

Los sistemas operativos que utilizamos generan una gran cantidad de métricas para nosotros, como lo comprobará en el momento en que instale collectd en una máquina Linux y lo dirija a Graphite. Asimismo, los subsistemas de soporte como Nginx o Varnish exponen información útil como tiempos de respuesta o tasas de aciertos de caché. Pero ¿qué sucede con su propio servicio?

Recomiendo encarecidamente que sus servicios expongan métricas básicas por sí mismos. Como mínimo, en el caso de un servicio web, probablemente debería exponer métricas como los tiempos de respuesta y las tasas de error, algo fundamental si su servidor no está respaldado por un servidor web que haga esto por usted.

Pero debería ir más allá. Por ejemplo, es posible que nuestro servicio de contabilidad quiera mostrar la cantidad de veces que los clientes vieron sus pedidos anteriores, o su tienda web podría querer registrar cuánto dinero ganó durante el último día.

¿Por qué nos importa esto? Bueno, por varias razones. En primer lugar, hay un viejo dicho que dice que el 80% de las características del software nunca se utilizan. No puedo opinar sobre la precisión de 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 utilizan. ¿No sería bueno saber cuáles son?

En segundo lugar, estamos mejorando más que nunca nuestra capacidad de reaccionar ante el uso que hacen nuestros usuarios de nuestro sistema para determinar cómo mejorarlo. Las métricas que nos informan sobre el comportamiento de nuestros sistemas sólo pueden ayudarnos en este aspecto. 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 un problema o es algo que se esperaba?

Por último, 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 mucho tiempo. Tiendo a equivocarme y a exponer todo y confiar en mi sistema de métricas para manejar esto más adelante.

Existen bibliotecas para varias plataformas diferentes que permiten que nuestros servicios envíen métricas a sistemas estándar. [Biblioteca de métricas](#) de Codahale es un ejemplo de biblioteca 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 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 el nivel adecuado de CPU o qué constituye un tiempo de respuesta aceptable. Si nuestro sistema de monitorización detecta que los valores reales están fuera de este nivel seguro, podemos activar una alerta, algo que una herramienta como Nagios es más que capaz de hacer.

Sin embargo, en muchos sentidos, estos valores están un paso más allá de lo que realmente queremos rastrear, es decir, ¿ está funcionando el sistema? Cuanto más complejas sean las interacciones entre los servicios, más lejos estaremos de responder esa pregunta. ¿Qué pasaría si nuestros sistemas de monitoreo estuvieran programados para actuar un poco como nuestros usuarios y pudieran informar si algo sale mal?

La primera vez que hice esto fue en 2005. Formaba parte de un pequeño equipo de ThoughtWorks que estaba creando un sistema para un banco de inversiones. A lo largo de la jornada de negociación, se producían muchos eventos que representaban cambios en el mercado. Nuestro trabajo consistía en reaccionar ante estos cambios y analizar el impacto en la cartera del banco. Trabajábamos con plazos bastante ajustados, ya que el objetivo era haber realizado todos nuestros cálculos en menos de 10 segundos después de que se produjera el evento. El sistema en sí consistía en alrededor de cinco servicios discretos, al menos uno de los cuales se ejecutaba en una red informática que, entre otras cosas, recuperaba ciclos de CPU no utilizados en alrededor de 250 hosts de escritorio en el centro de recuperación ante desastres del banco.

La cantidad de partes móviles del sistema implicaba que se generaba mucho ruido a partir de muchas de las métricas de nivel inferior que estábamos recopilando. No teníamos el beneficio de escalar gradualmente o de tener el sistema funcionando durante algunos meses para entender qué significaba que métricas como la velocidad de la CPU o incluso las latencias de algunos de los componentes individuales funcionaran bien . Nuestro enfoque fue generar eventos falsos para fijar el precio de una parte de la cartera que no estaba registrada en los sistemas posteriores. Cada minuto, aproximadamente, hacíamos que Nagios ejecutara un trabajo de línea de comandos que insertaba un evento falso en una de nuestras colas. Nuestro sistema lo detectaba y ejecutaba todos los cálculos como cualquier otro trabajo, excepto que los resultados aparecían en el libro basura , que se usaba solo para realizar pruebas. Si no se veía un cambio de precio en 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 comportara semánticamente, por lo que esta técnica suele denominarse monitoreo semántico.

En la práctica, he descubierto que el uso de transacciones sintéticas para realizar un seguimiento semántico como este es un indicador mucho mejor de los problemas en los sistemas que las alertas sobre las métricas de nivel inferior. Sin embargo, no reemplazan la necesidad de las métricas de nivel inferior: seguiremos necesitando ese detalle cuando necesitemos averiguar por qué nuestro seguimiento semántico informa de un problema.

## Implementación de la monitorización semántica

En el pasado, implementar el monitoreo semántico era una tarea bastante abrumadora. Pero el mundo ha avanzado y los medios para hacerlo están al alcance de la mano. Estás realizando pruebas para tus sistemas, ¿verdad? Si no es así, lee [el Capítulo 7](#) y vuelve. ¿Ya terminaste? ¡Bien!

Si observamos las pruebas que tenemos para probar un servicio determinado de extremo a extremo, o incluso todo nuestro sistema de extremo a extremo, tenemos mucho de lo que necesitamos para implementar la monitorización semántica. Nuestro sistema ya expone los ganchos necesarios para iniciar la prueba y verificar el resultado. Entonces, ¿por qué no ejecutar un subconjunto de estas pruebas de manera continua como una forma de monitorear nuestro sistema?

Por supuesto, hay algunas cosas que debemos hacer. En primer lugar, debemos tener cuidado con los requisitos de datos de nuestras pruebas. Es posible que tengamos que encontrar una forma de que nuestras pruebas se adapten a diferentes datos en vivo si estos cambian con el tiempo, o bien establecer una fuente de datos diferente. Por ejemplo, podríamos tener un conjunto de usuarios falsos que usemos en producción con un conjunto de datos conocido.

Del mismo modo, debemos asegurarnos de no provocar accidentalmente efectos secundarios imprevistos. Un amigo me contó una historia sobre una empresa de comercio electrónico que, por accidente, ejecutó sus pruebas en comparación con 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.

## Identificadores de correlación

Con una gran cantidad de servicios que interactúan para proporcionar una determinada capacidad al usuario final, una única llamada de inicio puede acabar generando varias llamadas de servicio posteriores. Por ejemplo, considere el caso de un cliente que se está registrando. El cliente completa todos sus datos en un formulario y hace clic en enviar. Tras bastidores, comprobamos la validez de los datos de la tarjeta de crédito con nuestro servicio de pago, hablamos con nuestro servicio postal para enviar un paquete de bienvenida por correo postal y enviamos un correo electrónico de bienvenida mediante nuestro servicio de correo electrónico. Ahora bien, ¿qué sucede si la llamada al servicio de pago acaba generando un error extraño? Hablaremos en profundidad sobre cómo manejar la falla en [el Capítulo 11](#), pero considere la dificultad de diagnosticar lo que sucedió.

Si observamos los registros, el único servicio que registra un error es nuestro servicio de pagos. 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 eventos que se activan y se manejan de manera asíncrona. ¿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 ascendente, tal como lo hacemos con un seguimiento de pila.

Un enfoque que puede resultar útil en este caso es utilizar identificadores de correlación. Cuando se realiza la primera llamada, se genera un GUID para la llamada. Este se transmite a todas las llamadas posteriores, como se ve en [la Figura 8-5](#), y se puede colocar en los registros de forma estructurada, de forma muy similar a como ya se hace con componentes como el nivel de registro o la fecha. Con las herramientas de agregación de registros adecuadas, podrá realizar un seguimiento de ese evento a lo largo de todo el sistema:

```
15-02-2014 16:01:01 Web-Frontend INFO [abc-123] Registrar 15-02-2014  
16:01:02 RegistrarServicio INFO [abc-123] RegistrarCliente...  
15-02-2014 16:01:03 PostalSystem INFO [abc-123] EnviarPaqueteDeBienvenida...  
15-02-2014 16:01:03 EmailSystem INFO [abc-123] EnviarCorreoElectrónicoDeBienvenida...  
15-02-2014 16:01:03 ERROR de PaymentGateway [abc-123] ValidatePayment...
```

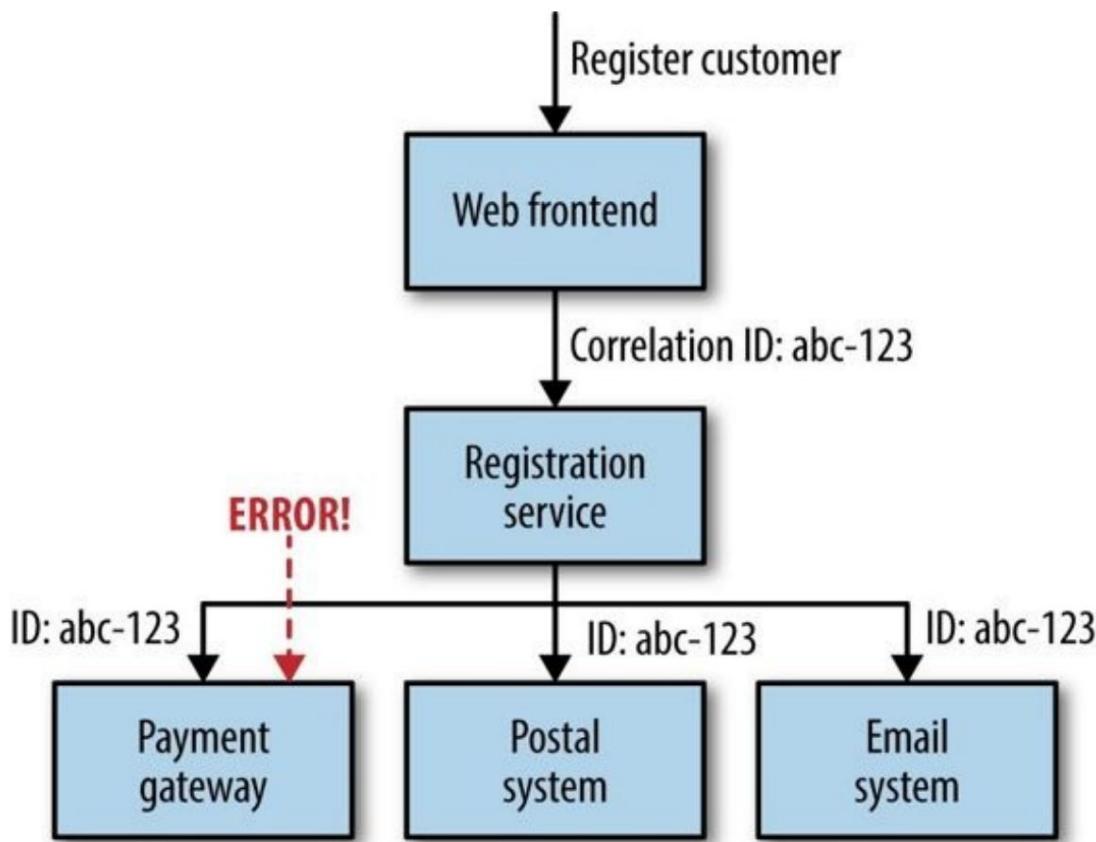


Figura 8-5. Uso de identificadores de correlación para rastrear cadenas de llamadas en múltiples servicios

Por supuesto, deberá asegurarse de que cada servicio sepa transmitir el ID de correlación.

Aquí es donde se necesita estandarizar y reforzar la aplicación de esta norma en todo el sistema. Pero una vez que se haya hecho esto, se pueden crear herramientas para realizar un seguimiento de todo tipo de interacciones. Dichas herramientas pueden ser útiles para rastrear tormentas de eventos, casos excepcionales o incluso para identificar transacciones especialmente costosas, ya que se puede imaginar toda la cascada de llamadas.

Software como [Zipkin](#) También puede rastrear llamadas a través de múltiples límites de sistema. Basándose en las ideas del propio sistema de rastreo de Google, Dapper, Zipkin puede proporcionar un rastreo muy detallado de 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, ya que requieren clientes personalizados y sistemas de recopilación de soporte. Dado que ya querrá la agregación de registros para otros fines, parece mucho más simple hacer uso de los datos que ya está recopilando en lugar de tener que recurrir a fuentes de datos adicionales. Dicho esto, si descubre que necesita una herramienta más avanzada para rastrear llamadas entre servicios como esta, es posible que desee echarle un vistazo.

Uno de los problemas reales con los identificadores de correlación es que, a menudo, no se sabe que se necesitan hasta que ya se tiene un problema que solo se podría diagnosticar si se tuviera el identificador al principio. Esto es especialmente problemático, ya que la adaptación de los identificadores de correlación es muy difícil; es necesario gestionarlos de forma estandarizada para poder reconstituir fácilmente las cadenas de llamadas. Aunque puede parecer un trabajo adicional al principio, le sugiero encarecidamente que considere la posibilidad de incorporarlos lo antes posible, especialmente si su sistema utilizará patrones de arquitectura basados en eventos, lo que puede provocar algún comportamiento emergente extraño.

La necesidad de manejar tareas como pasar constantemente a través de identificadores de correlación puede ser un argumento sólido para el uso de bibliotecas de contenedor de cliente compartidas delgadas. 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 hace falta que un servicio en la mitad de la cadena se olvide de hacer esto para que se pierda información crítica. Si decide crear una biblioteca de cliente interna para que cosas como esta funcionen de inmediato, asegúrese de mantenerla muy delgada y no vinculada a ningún servicio de producción en particular. Por ejemplo, si está utilizando HTTP como protocolo subyacente para la comunicación, simplemente envuelva una biblioteca de cliente HTTP estándar, agregando código para asegurarse de propagar los identificadores de correlación en los encabezados.

## La cascada

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

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

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