

# Spring Microservices IN ACTION



Juan Carnell



MANTENIMIENTO

## Microservicios de Spring en acción



# Microservicios de Spring en acción

JUAN CARNELL



MANTENIMIENTO  
ISLA REFUGIO

Para obtener información en línea y realizar pedidos de este y otros libros de Manning, visite [www.manning.com](http://www.manning.com). La editorial ofrece descuentos en este libro cuando se pide en cantidad.  
Para obtener más información, póngase en contacto

Departamento de Ventas Especiales  
Publicaciones Manning Co.  
20 Baldwin Road  
Apartado postal 761  
Isla Shelter, Nueva York 11964  
Correo electrónico: [pedidos@manning.com](mailto:pedidos@manning.com)

©2017 por Manning Publications Co. Todos los derechos reservados.

Ninguna parte de esta publicación puede reproducirse, almacenarse en un sistema de recuperación o transmitirse, de ninguna forma o por medios electrónicos, mecánicos, fotocopias o de otro tipo, sin el permiso previo por escrito del editor.

Muchas de las designaciones utilizadas por fabricantes y vendedores para distinguir sus productos se consideran marcas comerciales. Cuando esas designaciones aparecen en el libro y Manning Publications tenía conocimiento de un reclamo de marca registrada, las designaciones se imprimieron en mayúsculas iniciales o todas en mayúsculas.

- ⊗ Reconociendo la importancia de preservar lo que se ha escrito, la política de Manning es imprimir los libros que publicamos en papel libre de ácido, y hacemos nuestros mejores esfuerzos para lograr ese fin.  
Reconociendo también nuestra responsabilidad de conservar los recursos de nuestro planeta, Manning publica están impresos en papel que es al menos un 15 por ciento reciclado y procesado sin el uso de cloro elemental.



Publicaciones Manning Co.  
20 Baldwin Road PO  
Box 761  
Shelter Island, Nueva York 11964

Editor de adquisiciones: Greg Wild  
Editora de desarrollo: Marina Michaels  
Editor de desarrollo técnico: Raphael Villela  
Editora: Katie Petito  
Corrector: Melody Dolab  
Corrector técnico: Joshua White  
Editor de reseñas: Aleksandar Dragosavljevic  
Tipógrafo: Marija Tudor  
Diseño de portada: Marija Tudor

ISBN 9781617293986

Impreso en los Estados Unidos de América.

1 2 3 4 5 6 7 8 9 10 – MBE – 22 21 20 19 18 17

A mi hermano Jason, quien incluso en sus momentos más oscuros me mostró el verdadero significado de fuerza y dignidad.  
Eres un modelo a seguir como hermano, esposo y padre.



## contenidos breves

---

- 1 ■ Bienvenido a la nube, Spring 1 2 ■  
Creación de microservicios con Spring Boot 35 3 ■  
Control de su configuración con Spring Cloud  
    servidor de configuración 64 4 ■  
■ Sobre el descubrimiento de servicios  
96 5 ■ Cuando suceden cosas malas: patrones de resiliencia del cliente con  
    Nube de primavera y Netflix Hystrix 119  
6 ■ Enrutamiento de servicios con Spring Cloud y Zuul  
153 7 ■ Protección de sus microservicios  
192 8 ■ Arquitectura basada en eventos con Spring Cloud Stream  
228 9 ■ Seguimiento distribuido con Spring Cloud Sleuth y Zipkin 259  
10 ■ Implementación de sus microservicios 288



# contenido

---

prefacio xv  
agradecimientos xvii acerca  
de este libro xix  
sobre el autor xxii  
sobre la ilustración de portada xxiii

## 1 Bienvenido a la nube, Primavera 1

1.1 ¿Qué es un microservicio? 2  
1.2 ¿Qué es Spring y por qué es relevante para los microservicios? 5 1.3 Qué  
aprenderá en este libro 6 1.4 ¿Por qué este libro  
es relevante para usted? 7 1.5 Construyendo un  
microservicio con Spring Boot 8 1.6 ¿Por qué cambiar la forma  
en que construimos aplicaciones? 12 1.7 ¿Qué es exactamente la  
nube? 13 1.8 ¿Por qué la nube y los  
microservicios? 15 1.9 Los microservicios son más que  
escribir el código 17

Patrón de desarrollo de microservicios principales 19 ■ Patrones de enrutamiento de  
microservicios 20 ■ Patrones de resiliencia del cliente de microservicios 21  
Patrones de seguridad de microservicios 23 ■ Patrones de registro y seguimiento  
de microservicios 24 ■ Patrones de creación/implementación de microservicios 25

### 1.10 Uso de Spring Cloud para crear sus microservicios 26

Spring Boot 28 ■ Spring Cloud Config 28 ■ Descubrimiento del servicio Spring  
Cloud 28 ■ Spring Cloud/Netflix Hystrix y

Ribbon 29 ■ Spring Cloud/Netflix Zuul 29 ■ Spring Cloud Stream 29 ■ Spring Cloud Sleuth 29 ■ Spring Cloud Security 30 ■ ¿Qué pasa con el aprovisionamiento? 30

- 1.11 Spring Cloud por ejemplo 30
- 1.12 Asegurarnos de que nuestros ejemplos sean relevantes 33
- 1.13 Resumen 33

## 2 Creación de microservicios con Spring Boot 35

- 2.1 La historia del arquitecto: diseño de la arquitectura de microservicios 38
  - Descomponer el problema empresarial 38 ■ Establecer la granularidad del servicio 41 ■ Hablar entre nosotros: interfaces de servicio 43
- 2.2 Cuándo no utilizar microservicios 44
  - Complejidad de construir sistemas distribuidos 44 ■ Expansión de servidores 44 ■ Tipo de aplicación 44 ■ Transformaciones y consistencia de datos 45
- 2.3 La historia del desarrollador: creación de un microservicio con Spring Boot y Java 45
  - Comenzando con el proyecto esqueleto 46 ■ Arrancar su aplicación Spring Boot: escribir la clase Bootstrap 47 ■ Construir la puerta de entrada al microservicio: el controlador Spring Boot 48
- 2.4 La historia de DevOps: construir para los rigores del tiempo de ejecución 53
  - Montaje de servicios: empaquetar e implementar sus microservicios 56
  - Arranque de servicios: gestión de la configuración de sus microservicios 58 ■ Registro y descubrimiento de servicios: cómo se comunican los clientes con sus microservicios 59 ■ Comunicación del estado de un microservicio 60
- 2.5 Uniendo las perspectivas 62
- 2.6 Resumen 63

## 3 Controlar su configuración con el servidor de configuración Spring Cloud 64

- 3.1 Sobre la gestión de la configuración (y la complejidad) 65
  - Su arquitectura de gestión de configuración 67 ■ Opciones de implementación 69
- 3.2 Construyendo nuestro servidor de configuración Spring Cloud 70
  - Configuración de la clase Spring Cloud Config Bootstrap 74
  - Usando el servidor de configuración Spring Cloud con el sistema de archivos 75

### 3.3 Integración de Spring Cloud Config con un cliente Spring Boot 77

Configurar las dependencias del servidor Spring Cloud Config del servicio de licencias 79 ■ Configurar el servicio de licencias para usar Spring Cloud Config 79 ■ Cablear una fuente de datos usando el servidor de configuración Spring Cloud 83 ■ Lectura directa de propiedades usando la anotación @Value 86 ■ Usar el servidor de configuración Spring Cloud con Git 87 ■ Actualizar sus propiedades usando el servidor de configuración Spring Cloud 88

### 3.4 Protección de información de configuración confidencial 89

Descargue e instale los archivos jar de Oracle JCE necesarios para el cifrado 90  
Configurar una clave de cifrado 91 ■ Cifrar y descifrar una propiedad 91 ■  
Configurar microservicios para usar cifrado en el lado del cliente 93

### 3.5 Pensamientos finales 95

### 3.6 Resumen 95

## 4 Sobre el descubrimiento de servicios 96

### 4.1 ¿Dónde está mi servicio? 97

#### 4.2 Sobre el descubrimiento de servicios en la nube 100

La arquitectura del descubrimiento de servicios 100 ■ Descubrimiento de servicios en acción usando Spring y Netflix Eureka 103

#### 4.3 Creación de su servicio Spring Eureka 105

#### 4.4 Registro de servicios con Spring Eureka 107

#### 4.5 Uso del descubrimiento de servicios para buscar un servicio 111

Buscar instancias de servicio con Spring DiscoveryClient 112  
Invocación de servicios con Spring RestTemplate 114 compatible con cintas  
Invocar servicios con el cliente Netflix Feign 116

#### 4.6 Resumen 118

## 5 Cuando suceden cosas malas: patrones de resiliencia del cliente con Spring Cloud y Netflix Hystrix 119

### 5.1 ¿Qué son los patrones de resiliencia del lado del cliente? 120

Equilibrio de carga del lado del cliente 121 ■ Disyuntor 122  
Procesamiento alternativo 122 ■ Mamparos 122

#### 5.2 Por qué es importante la resiliencia del cliente 123

#### 5.3 Ingrese Hystrix 126

#### 5.4 Configurar el servidor de licencias para usar Spring Cloud y Hystrix 127

## 5.5 Implementación de un disyuntor usando Hystrix 128

Temporización de una llamada al microservicio de la organización 131

Personalizar el tiempo de espera en un disyuntor 132

## 5.6 Procesamiento alternativo 133

### 5.7 Implementación del patrón de mamparo 136

### 5.8 Ir más allá de lo básico; ajuste fino Hystrix 138

Configuración de Hystrix revisada 142

## 5.9 Contexto del hilo y Hystrix 144

ThreadLocal y Hystrix 144 ■ La estrategia de concurrencia de Hystrix en acción 147

## 5.10 Resumen 151

# 6 Enrutamiento de servicios con Spring Cloud y Zuul 153

## 6.1 ¿Qué es una pasarela de servicios? 154

## 6.2 Presentación de Spring Cloud y Netflix Zuul 157

Configurando el proyecto Zuul Spring Boot 157 ■ Usando Spring

Anotación en la nube para el servicio Zuul 157 ■ Configuración de Zuul para comunicarse con Eureka 158

## 6.3 Configuración de rutas en Zuul 159

Rutas de mapeo automatizadas a través del descubrimiento de servicios 159

Mapeo de rutas manualmente usando el descubrimiento de servicios 161

Mapeo manual de rutas usando URL estáticas 165

Recargar dinámicamente la configuración de ruta 168 ■ Zuul y tiempos de espera del servicio 169

## 6.4 El verdadero poder de Zuul: filtros 169

## 6.5 Construyendo su primer prefiltro Zuul generando ID de correlación 173

Usando el ID de correlación en sus llamadas de servicio 176

## 6.6 Creación de un filtro posterior que recibe ID de correlación 182

## 6.7 Construyendo un filtro de ruta dinámico 184

Construyendo el esqueleto del filtro de enrutamiento 186 ■ Implementando el método run() 187 ■ Reenviando la ruta 188 ■ Juntándolo todo 190

## 6.8 Resumen 191

# 7 Proteger sus microservicios 192

## 7.1 Introducción a OAuth2 193

## 7.2 Empezar poco a poco: usar Spring y OAuth2 para proteger un único punto final 195

Configurar el servicio de autenticación EagleEye OAuth2 196

Registro de aplicaciones cliente con el servicio OAuth2 197

Configurar usuarios de EagleEye 200 ■ Autenticar al usuario 202

## 7.3 Proteger el servicio de la organización utilizando OAuth2 205

Agregar los archivos jar Spring Security y OAuth2 a los servicios

individuales 205 ■ Configurar el servicio para que apunte a su servicio de autenticación OAuth2 206 ■ Definir quién y qué puede acceder al servicio 207 ■ Propagar el token de acceso OAuth2 210

## 7.4 Tokens web de JavaScript y OAuth2 213

Modificación del servicio de autenticación para emitir JavaScript Web

Tokens 214 ■ Consumir tokens web de JavaScript en sus microservicios 218 ■ Ampliar el token JWT 220

Analizando un campo personalizado a partir de un token de JavaScript 222

## 7.5 Algunas reflexiones finales sobre la seguridad de los microservicios 224

### 7.6 Resumen 227

## 8 Arquitectura basada en eventos con Spring Cloud Stream 228

### 8.1 El caso de la mensajería, EDA y los microservicios 229

Uso de un enfoque de solicitud-respuesta sincrónica para comunicar cambios de estado 230 ■ Uso de mensajería para comunicar cambios de estado entre servicios 233 ■ Desventajas de una arquitectura de mensajería 235

### 8.2 Presentación de Spring Cloud Stream 236

La arquitectura Spring Cloud Stream 237

### 8.3 Escribir un mensaje simple productor y consumidor 238

Escribir el productor de mensajes en el servicio de organización 239

Escribir el mensaje consumidor en el servicio de licencias 244

Ver el servicio de mensajes en acción 247

### 8.4 Un caso de uso de Spring Cloud Stream: almacenamiento en caché distribuido 249

Usar Redis para almacenar en caché las búsquedas 250 ■ Definir canales personalizados 256

Reuniéndolo todo: borrar el caché cuando se recibe un mensaje 257

### 8.5 Resumen 258

## 9 Seguimiento distribuido con Spring Cloud Sleuth y Zipkin 259

### 9.1 Spring Cloud Sleuth y la correlación ID 260

Agregar Spring Cloud Sleuth a las licencias y la organización 261

Anatomía de un rastro de Spring Cloud Sleuth 262

9.2 Agregación de registros y Spring Cloud Sleuth	263
Una implementación de Spring Cloud Sleuth/Papertrail en acción	265
Cree una cuenta de Papertrail y configure un conector syslog	267
Redirigir la salida de Docker a Papertrail	268 ■ Buscando
ID de seguimiento de Spring Cloud Sleuth en Papertrail	270 ■ Agregar el ID de correlación a la respuesta HTTP con Zuul
9.3 Seguimiento distribuido con Open Zipkin	274
Configuración de las dependencias de Spring Cloud Sleuth y Zipkin	275
Configurar los servicios para que apunten a Zipkin	275 ■ Instalar y configurar un servidor Zipkin
Establecer niveles de seguimiento	278
Usar Zipkin para rastrear transacciones	278 ■ Visualizar una transacción más compleja
Capturar rastros de mensajes	282
Agregar tramos personalizados	284
9.4 Resumen	287
<b>10 Implementación de sus microservicios</b>	<b>288</b>
10.1 EagleEye: configurando su infraestructura central en la nube	290
Creación de la base de datos PostgreSQL utilizando Amazon RDS	293
Crear el clúster de Redis en Amazon	296 ■ Crear un clúster de ECS
10.2 Más allá de la infraestructura: implementación de EagleEye	302
Implementar los servicios EagleEye en ECS manualmente	303
10.3 La arquitectura de un proceso de construcción/implementación	305
10.4 Su proceso de construcción e implementación en acción	309
10.5 Comenzando la implementación/canalización de su compilación: GitHub y Travis CI	311
10.6 Habilitar su servicio para construir en Travis CI	312
Configuración del tiempo de ejecución de la compilación principal	315 ■
Instalaciones de herramientas previas a la compilación	318 ■ Ejecutar la compilación
Etiquetar el código de control de fuente	320 ■ Construir los microservicios y crear las imágenes de Docker
Enviar las imágenes a Docker Hub	322 ■
Inicio de los servicios en Amazon ECS	323 ■ Inicio de las pruebas de la plataforma
10.7 Reflexiones finales sobre el proceso de construcción/implementación	325
10.8 Resumen	325
Apéndice A Ejecución de una nube en su escritorio	327
Apéndice B Tipos de concesión OAuth2	336
índice	345

# prefacio

---

Es irónico que al escribir un libro, la última parte del mismo sea a menudo el comienzo del mismo. También suele ser la parte más difícil de plasmar en papel. ¿Por qué?

Porque tienes que explicarle a todo el mundo por qué te apasiona tanto un tema que

Pasaste el último año y medio de tu vida escribiendo un libro al respecto. Es difícil

Articule por qué alguien dedicaría tanto tiempo a un libro técnico.

Rara vez se escriben libros de software por dinero o fama.

Ésta es la razón por la que escribí este libro: me encanta escribir código. es un llamado para mí y también es una actividad creativa, similar a dibujar, pintar o tocar un instrumento.

Quienes están fuera del campo del desarrollo de software tienen dificultades para entender esto.

Me gusta especialmente crear aplicaciones distribuidas. Para mí, es algo increíble de ver.

una aplicación funciona en docenas (incluso cientos) de servidores. Es como mirar un orquesta tocando una pieza musical. Si bien el producto final de una orquesta es hermoso, realizarlo suele requerir mucho trabajo y una cantidad significativa de esfuerzo.

práctica. Lo mismo ocurre con la escritura de una aplicación distribuida masivamente.

Desde que entré al campo del desarrollo de software hace 25 años, he observado el

La industria lucha por encontrar la forma “correcta” de crear aplicaciones distribuidas. He visto estándares

de servicios distribuidos como CORBA subir y bajar. Empresas monstruosamente grandes

han intentado impulsar protocolos grandes y, a menudo, propietarios. ¿Alguien recuerda el modelo de objetos componentes distribuidos (DCOM) de Microsoft o el J2EE Enterprise de Oracle?

¿ Java Beans 2 (EJB)? Observé cómo las empresas de tecnología y sus seguidores se apresuraban a

Cree arquitecturas orientadas a servicios (SOA) utilizando esquemas pesados basados en XML.

En cada caso, estos enfoques para construir sistemas distribuidos a menudo colapsaron

bajo su propio peso. No estoy diciendo que estas tecnologías no se utilizaron para construir

algunas aplicaciones muy potentes. La realidad es que no pudieron seguir el ritmo

demandas de los usuarios. Hace diez años, los teléfonos inteligentes apenas se estaban introduciendo en el mercado. El mercado y la computación en la nube se encontraban en sus primeras etapas. Asimismo, las normas y la tecnología para el desarrollo de aplicaciones distribuidas eran demasiado complicadas para el desarrollador promedio para comprender y utilizar fácilmente en la práctica. Nada dice la verdad en la industria del desarrollo de software como código escrito. Cuando los estándares entran en juego De esta manera, los estándares rápidamente se descartan.

Cuando escuché por primera vez sobre el enfoque de microservicios para crear aplicaciones, estaba más que un poco escéptico. "Genial, otro enfoque milagroso para crear aplicaciones distribuidas", pensé. Sin embargo, cuando comencé a profundizar en los conceptos, me di cuenta

La simplicidad de los microservicios podría cambiar las reglas del juego. Una arquitectura de microservicio se centra en crear pequeños servicios que utilizan protocolos simples (HTTP y JSON) para comunicarse. Eso es todo. Puede escribir un microservicio con casi cualquier lenguaje de programación. Hay belleza en esta simplicidad.

Sin embargo, aunque crear un microservicio individual es fácil, ponerlo en funcionamiento y escalarlo es difícil. Hacer funcionar cientos de pequeños componentes distribuidos Trabajar juntos y luego crear una aplicación resistente a partir de ellos puede ser increíblemente difícil de hacer. En la informática distribuida, el fracaso es una realidad y cómo su aplicación se ocupa de ello es increíblemente difícil hacerlo bien. Parafraseando a mis colegas Chris Miller y Shawn Hagwood: "Si no se rompe de vez en cuando, no se está construyendo".

Son estos fracasos los que me inspiraron a escribir este libro. Odio construir cosas a partir de rascarme cuando no es necesario. La realidad es que Java es la lengua franca para la mayoría de los esfuerzos de desarrollo de aplicaciones, especialmente en la empresa. El marco Spring tiene para muchas organizaciones se convierten en el marco de facto para la mayor parte del desarrollo de aplicaciones. Ya llevaba casi 20 años desarrollando aplicaciones en Java (tenía recuerde el subprograma Dancing Duke) y Spring durante casi 10 años. Cuando comencé mi viaje de microservicios, estuve encantado y emocionado de ver el surgimiento de Spring Nube.

El marco Spring Cloud proporciona soluciones listas para usar para muchos de los Problemas operativos y de desarrollo comunes que encontrará como microservicio desarrollador. Spring Cloud le permite usar solo las piezas que necesita y minimiza el cantidad de trabajo que necesita hacer para crear e implementar microservicios Java listos para producción. Para ello, utiliza otras tecnologías arraigadas de empresas y grupos como Netflix, HashiCorp y la fundación Apache.

Siempre me he considerado un desarrollador promedio que, al final del día, tiene plazos a cumplir. Por eso emprendí el proyecto de escribir este libro. quería un Libro que podría utilizar en mi trabajo diario. Quería algo con directo (y con suerte) ejemplos de código sencillos. Siempre quiero asegurarme de que el material de este libro pueda consumirse como capítulos individuales o en su totalidad. te espero Considero útil este libro y espero que disfrutes leyéndolo tanto como yo disfruté escribiéndolo.

## expresiones de gratitud

---

Mientras me siento a escribir estos agradecimientos, no puedo evitar pensar en 2014. cuando corrí mi primer maratón. Escribir un libro es muy parecido a correr una maratón. Escribiendo La propuesta y el esquema del libro se parecen mucho al proceso de formación. Se pone tus pensamientos en forma, te enfoca para lo que está por delante y, sí, cerca del final del proceso, puede ser más que un poco tedioso y brutal.

Cuando empiezas a escribir el libro, se parece mucho al día de la carrera. Empiezas el maratón emocionado y lleno de energía. Sabes que estás intentando hacer algo más grande que cualquier cosa que hayas hecho antes y es a la vez emocionante y estresante. Esto es aquello para lo que has entrenado, pero al mismo tiempo, siempre está esa vocecita de duda en el fondo de tu mente que dice que no terminarás lo que empezaste.

Lo que he aprendido al correr es que las carreras no se completan milla a milla. En cambio, corren un pie delante del otro. Las millas recorridas son la suma de las pasos individuales. Cuando mis hijos tienen dificultades con algo, me río y les pregunto. ellos: “¿Cómo se escribe un libro? Una palabra, un solo paso a la vez”. Ellos usualmente ponen los ojos en blanco, pero al final no hay otra manera de eludir esta ley indiscutible y férrea.

Sin embargo, cuando corres un maratón, puede que seas tú quien corra la carrera, pero nunca lo ejecutarás solo. Hay todo un equipo de personas allí para brindarte apoyo, tiempo y consejos a lo largo del camino. Ha sido la misma experiencia escribir esto. libro.

Me gustaría comenzar agradeciendo a Manning por el apoyo que me brindó al escribir esto. libro. Todo empezó con Greg Wild, mi editor de adquisiciones, que trabajó pacientemente conmigo. mientras refinaba los conceptos centrales de este libro y me guiaba a través del proceso de propuesta. En el camino, Marina Michaels, mi editora de desarrollo, me mantuvo honesta y

Me desafió a convertirme en un mejor autor. También me gustaría agradecer a Raphael Villela y Joshua White, mis editores técnicos, quienes revisaron constantemente mi trabajo y aseguraron la Calidad general de los ejemplos y el código que produce. Estoy extremadamente agradecido por el tiempo, el talento y el compromiso que cada uno de estos individuos pone en el proyecto general. proyecto. También me gustaría agradecer a los revisores que brindaron comentarios sobre el manuscrito. durante todo el proceso de redacción y desarrollo: Aditya Kumar, Adrian M. Rossi, Ashwin Raj, Christian Bach, Edgar Knapp, Jared Duncan, Jiri Pik, John Guthrie, Mirko Bernardoni, Paul Balogh, Pierluigi Riti, Raju Myadam, Rambabu Posa, Sergey Esvikov, y Vipul Gupta.

Quiero cerrar estos agradecimientos con un profundo sentimiento de agradecimiento por el cariño y tiempo que mi familia me ha brindado para trabajar en este proyecto. Para mi esposa Janet, tienes sido mi mejor amigo y el amor de mi vida. Cuando estoy cansado y quiero rendirme, sólo Tengo que escuchar el sonido de tus pasos a mi lado para saber que siempre estás corriendo a mi lado, nunca diciéndome que no y siempre empujándome hacia adelante.

Para mi hijo Christopher, estás creciendo para convertirte en un joven increíble. No puedo esperar a que llegue el día en que realmente descubras tu pasión, porque no habrá nada en este mundo que pueda impedirte alcanzar tus metas.

A mi hija Agatha le daría todo el dinero que tengo para ver el mundo. tus ojos por sólo 10 minutos. La experiencia me haría un mejor autor y Más importante aún, una mejor persona. Tu intelecto, tu poder de observación y tu creatividad me humillan.

A mi hijo de cuatro años, Jack: Buddy, gracias por tu paciencia conmigo siempre que dijiste: "No puedo jugar ahora porque papá tiene que trabajar en el libro". tu siempre haces Me río y haces que toda esta familia esté completa. Nada me hace más feliz que cuando te veo siendo el bromista y jugando con todos los miembros de la familia.

Mi carrera con este libro ha terminado. Al igual que mi maratón, no dejé nada sobre la mesa en escribiendo este libro. No tengo más que gratitud para el equipo de Manning y el MEAP. Lectores que compraron este libro temprano y me brindaron comentarios muy valiosos. espero en Al final disfrutarás este libro tanto como yo disfruté escribiéndolo. Gracias.

# sobre este libro

---

Spring Microservices in Action fue escrito para el desarrollador practicante de Java/Spring que necesita consejos prácticos y ejemplos de cómo construir y poner en funcionamiento aplicaciones basadas en microservicios. Cuando escribí este libro, quería que se basara en temas centrales. patrones de microservicios que se alinearon con ejemplos de Spring Boot y Spring Cloud que demostró los patrones en acción. Como tal, encontrará un diseño de microservicio específico patrones discutidos en casi todos los capítulos, junto con ejemplos de los patrones implementados usando Spring Boot y Spring Cloud.

## Deberías leer este libro si

Eres un desarrollador de Java y tienes experiencia en la creación de aplicaciones distribuidas.  
(1-3 años).

Tienes experiencia en Spring (más de 1 año). Está interesado en aprender a crear aplicaciones basadas en microservicios. ¿Le interesa saber cómo puede utilizar microservicios para crear servicios basados en la nube? aplicaciones.

Quiere saber si Java y Spring son tecnologías relevantes para construir Aplicaciones basadas en microservicios.

Le interesa ver qué implica implementar una aplicación basada en microservicios. catión a la nube.

## Cómo está organizado este libro

Spring Microservices in Action consta de 10 capítulos y dos apéndices:

El Capítulo 1 le presenta por qué la arquitectura de microservicios es una parte importante. y enfoque relevante para la creación de aplicaciones, especialmente basadas en la nube aplicaciones.

El Capítulo 2 le explica cómo crear su primer microservicio basado en REST, usando arranque de primavera. Este capítulo lo guiará sobre cómo mirar sus microservicios a través de los ojos de un arquitecto, un ingeniero de aplicaciones y un DevOps ingeniero.

El Capítulo 3 le presenta cómo administrar la configuración de sus microservicios usando Spring Cloud Config. Spring Cloud Config le ayuda a garantizar que la información de configuración de su servicio está centralizada en un único repositorio, versionado y repetible en todas las instancias de sus servicios.

El Capítulo 4 le presenta uno de los primeros patrones de enrutamiento de microservicios: el descubrimiento de servicios. En este capítulo, aprenderá cómo utilizar Spring Cloud y el servicio Eureka de Netflix para abstraer la ubicación de sus servicios del clientes que los consumen.

El Capítulo 5 trata sobre proteger a los consumidores de sus microservicios cuando uno o más instancias de microservicio están inactivas o en un estado degradado. Este capítulo demuestre cómo usar Spring Cloud y Netflix Hystrix (y Netflix Ribbon) para implementar el equilibrio de carga de llamadas del lado del cliente, el patrón de disyuntor, el patrón de respaldo y el patrón de mamparo.

El Capítulo 6 cubre el patrón de enrutamiento de microservicios: la puerta de enlace de servicios. Usando Spring Cloud con el servidor Zuul de Netflix, creará un punto de entrada único para todos los microservicios a través de los cuales se llamará. Discutiremos cómo usar la API de filtro de Zuul para crear políticas que puedan aplicarse a todos los servicios que fluyen a través de la puerta de enlace de servicios.

El Capítulo 7 cubre cómo implementar la autenticación y autorización del servicio, utilizando la seguridad de Spring Cloud y OAuth2. Cubriremos los conceptos básicos para configurar un Servicio OAuth2 para proteger sus servicios y también cómo utilizar JavaScript Web Tokens (JWT) en su implementación de OAuth2 .

El Capítulo 8 analiza cómo puede introducir la mensajería asíncrona en sus microservicios utilizando Spring Cloud Stream y Apache Kafka.

El Capítulo 9 muestra cómo implementar patrones de registro comunes, como correlación de registros, agregación de registros y seguimiento, utilizando Spring Cloud Sleuth y Open Zipkin.

El capítulo 10 es el proyecto fundamental del libro. Tomarás los servicios que ha creado en el libro e implementarlos en Amazon Elastic Container Service (ECS). También discutiremos cómo automatizar la construcción y la implementación de sus microservicios utilizando herramientas como Travis CI.

El Apéndice A cubre cómo configurar su entorno de desarrollo de escritorio para que pueda ejecutar todos los ejemplos de código de este libro. Este apéndice cubre cómo el proceso de compilación local funciona y también cómo iniciar Docker localmente si lo desea para ejecutar los ejemplos de código localmente.

El Apéndice B es material complementario sobre OAuth2. OAuth2 es un modelo de autenticación extremadamente flexible y este capítulo proporciona una breve descripción general de las diferentes maneras en que OAuth2 se puede utilizar para proteger una aplicación y sus microservicios correspondientes.

### Sobre el código

Spring Microservices in Action incluye código en cada capítulo. Todos los ejemplos de código están disponibles en mi repositorio de GitHub y cada capítulo tiene su propio repositorio. Puede encontrar una página de descripción general con enlaces al repositorio de código de cada capítulo en [https://github.com/carnelliJ/spmia\\_overview](https://github.com/carnelliJ/spmia_overview). También está disponible un zip que contiene todo el código fuente en el sitio web del editor en [www.manning.com/books/spring-microservices-in-action](http://www.manning.com/books/spring-microservices-in-action).

Todo el código de este libro está diseñado para ejecutarse en Java 8 utilizando Maven como herramienta de compilación principal. Consulte el apéndice A de este libro para obtener detalles completos sobre las herramientas de software que necesitará para compilar y ejecutar los ejemplos de código.

Uno de los conceptos centrales que seguí mientras escribía este libro fue que los ejemplos de código de cada capítulo deberían ejecutarse independientemente de los de los otros capítulos. Como tal, cada servicio que creamos para un capítulo se basa en una imagen de Docker correspondiente.

Cuando se utiliza el código de los capítulos anteriores, se incluye como fuente y como imagen de Docker integrada. Usamos Docker Compose y las imágenes de Docker creadas para garantizar que tenga un entorno de ejecución reproducible para cada capítulo.

Este libro contiene muchos ejemplos de código fuente tanto en listados numerados como en línea con el texto normal. En ambos casos, el código fuente tiene el formato de una fuente de ancho fijo como esta para separarlo del texto normal. A veces, el código también está en negrita para resaltar el código que ha cambiado con respecto a los pasos anteriores del capítulo, como cuando se agrega una nueva característica a una línea de código existente.

En muchos casos, se ha reformateado el código fuente original; Agregamos saltos de línea y modificamos la sangría para adaptarla al espacio de página disponible en el libro. En casos raros, ni siquiera esto fue suficiente y los listados incluyen marcadores de continuación de línea ( ). Además, los comentarios en el código fuente a menudo se eliminan de los listados cuando el código se describe en el texto. Muchas de las listas están acompañadas de anotaciones de código que resaltan conceptos importantes.

### Autor en línea

La compra de Spring Microservices in Action incluye acceso gratuito a un foro web privado dirigido por Manning Publications donde puede hacer comentarios sobre el libro, hacer preguntas técnicas y recibir ayuda del autor y de otros usuarios. Para acceder al foro y suscribirse a él, dirija su navegador web a [www.manning.com/books/spring-microservices-in-action](http://www.manning.com/books/spring-microservices-in-action). Esta página proporciona información sobre cómo ingresar al foro una vez que esté registrado, qué tipo de ayuda está disponible y las reglas de conducta en el foro.

El compromiso de Manning con nuestros lectores es proporcionar un lugar donde pueda tener lugar un diálogo significativo entre lectores individuales y entre lectores y el autor.

No es un compromiso de ningún monto específico de participación por parte del autor, cuyas contribuciones al AO siguen siendo voluntarias (y no remuneradas). Le sugerimos que le haga preguntas desafiantes al autor, ¡para que no se desvíe su interés!

## Sobre el Autor

---



JOHN CARNELL es ingeniero senior de nube en Genesys, donde trabaja en la división PureCloud de Genesys. John dedica la mayor parte de su día a crear microservicios basados en telefonía, utilizando la plataforma AWS . Su trabajo diario se centra en el diseño, y crear microservicios en varias plataformas tecnológicas, incluidas Java, Clojure y Go.

John es un prolífico orador y escritor. Habla regularmente en grupos de usuarios locales y ha sido orador habitual en "The No Simposio de software Fluff Just Stuff ". Durante los últimos 20 años, John ha sido autor, coautor y revisor técnico de varios libros sobre tecnología basada en Java, y publicaciones de la industria.

John tiene una Licenciatura en Artes (BA) de la Universidad de Marquette y una Maestría en Administración de Empresas (MBA) de la Universidad de Wisconsin Oshkosh.

John es un tecnólogo apasionado y explora constantemente nuevas tecnologías y lenguajes de programación. Cuando John no habla, escribe o codifica, vive con su esposa Janet, sus tres hijos, Christopher, Agatha y Jack, y sí, su perro. Vader, en Cary, Carolina del Norte.

Durante su tiempo libre (del cual hay muy poco), John corre, persigue a sus hijos, niños y estudia artes marciales filipinas.

Puede comunicarse con John en [john\\_carnell@yahoo.com](mailto:john_carnell@yahoo.com).

## sobre la ilustración de la portada

---

La figura de la portada de Spring Microservices in Action lleva la leyenda "Un hombre de Croacia". Esta ilustración está tomada de una reimpresión reciente de Imágenes y descripciones de Wenda, ilirios y eslavos del suroeste y este de Balthasar Hacquet, publicada por el Museo Etnográfico de Split, Croacia, en 2008. Hacquet (1739–1815) fue un médico austriaco y científico que pasó muchos años estudiando la botánica, la geología y la etnografía de muchas partes del Imperio austriaco, así como del Véneto, los Alpes Julianos y los Balcanes occidentales, habitados en el pasado por pueblos de las tribus ilirias.

Ilustraciones dibujadas a mano acompañan los numerosos artículos y libros científicos que publicó Hacquet.

La rica diversidad de los dibujos de las publicaciones de Hacquet habla vívidamente de la singularidad y la individualidad de las regiones de los Alpes orientales y de los Balcanes noroccidentales hace apenas 200 años. Era una época en la que los códigos de vestimenta de dos pueblos separados por unos pocos kilómetros identificaban a las personas únicamente como pertenecientes a uno u otro, y cuando los miembros de una clase social u oficio podían distinguirse fácilmente por su vestimenta. Los códigos de vestimenta han cambiado desde entonces y la diversidad por regiones, tan rica en ese momento, se ha desvanecido. Actualmente resulta difícil distinguir a los habitantes de un continente de otros, y hoy en día los habitantes de las pintorescas ciudades y pueblos de los Alpes eslovenos o de las ciudades costeras de los Balcanes no se distinguen fácilmente de los residentes de otras partes de Europa.

En Manning celebramos la inventiva, la iniciativa y la diversión del negocio de las computadoras con portadas de libros basadas en trajes de hace dos siglos, revividos por ilustraciones como ésta.



# Bienvenido a la nube,

## Primavera



### Este capítulo cubre

Comprender los microservicios y por qué las empresas los utilizan

Uso de Spring, Spring Boot y Spring Cloud para crear microservicios

Aprender por qué la nube y los microservicios son relevantes para las aplicaciones basadas en microservicios

Crear microservicios implica más que crear código de servicio

Comprender las partes del desarrollo basado en la nube

Uso de Spring Boot y Spring Cloud en el desarrollo de microservicios

La única constante en el campo del desarrollo de software es que nosotros, como desarrolladores de software, nos encontramos en medio de un mar de caos y cambios. Todos sentimos la rotación como nueva. Las tecnologías y los enfoques aparecen repentinamente en escena, lo que nos lleva a reevaluar cómo construimos y ofrecemos soluciones para nuestros clientes. Un ejemplo de esto es la rotación, la rápida adopción por parte de muchas organizaciones de crear aplicaciones utilizando

microservicios. Los microservicios son servicios de software distribuidos y débilmente acoplados que Realizar un pequeño número de tareas bien definidas.

Este libro le presenta la arquitectura de microservicios y por qué debería hacerlo. considere crear sus aplicaciones con ellos. Vamos a ver cómo construir microservicios que utilizan Java y dos proyectos de Spring Framework: Spring Boot y Spring Nube. Si es desarrollador de Java, Spring Boot y Spring Cloud le proporcionarán una solución sencilla ruta de migración desde la creación de aplicaciones Spring monolíticas y tradicionales hasta aplicaciones de microservicios que se pueden implementar en la nube.

## 1.1 ¿Qué es un microservicio?

Antes de que evolucionara el concepto de microservicios, la mayoría de las aplicaciones basadas en web se creaban utilizando un estilo arquitectónico monolítico. En una arquitectura monolítica, una aplicación es entregado como un único artefacto de software implementable. Toda la UI (interfaz de usuario), empresarial, y la lógica de acceso a la base de datos se empaquetan en un único artefacto de aplicación y implementado en un servidor de aplicaciones.

Si bien una aplicación puede implementarse como una única unidad de trabajo, la mayoría de las veces Habrá varios equipos de desarrollo trabajando en la aplicación. Cada equipo de desarrollo tendrá sus propias partes discretas de la aplicación de la que son responsables. para y, a menudo, clientes específicos a los que atienden con su pieza funcional. Para Por ejemplo, cuando trabajaba en una gran empresa de servicios financieros, teníamos una empresa interna, aplicación personalizada de gestión de relaciones con el cliente (CRM) que implicaba coordinación de múltiples equipos, incluida la interfaz de usuario, el maestro de clientes, el almacén de datos y el equipo de fondos mutuos. La Figura 1.1 ilustra la arquitectura básica de este solicitud.

El problema aquí es que a medida que crecieron el tamaño y la complejidad de la aplicación monolítica de CRM , los costos de comunicación y coordinación de los equipos individuales que trabajaban en la aplicación no aumentaron. Cada vez que un equipo individual necesitaba hacer una cambio, toda la aplicación tuvo que ser reconstruida, probada y reimplementada.

El concepto de microservicio originalmente se coló en la conciencia de la comunidad de desarrollo de software alrededor de 2014 y fue una respuesta directa a muchos de los desafíos de intentar escalar, tanto técnica como organizativamente, grandes y monolíticos. aplicaciones. Recuerde, un microservicio es un servicio distribuido, pequeño y poco acoplado. Los microservicios le permiten tomar una aplicación grande y descomponerla en componentes fáciles de administrar con responsabilidades estrictamente definidas. Los microservicios ayudan a combatir los problemas tradicionales de complejidad en una gran base de código al descomponer los grandes base del código en partes pequeñas y bien definidas. El concepto clave que debes adoptar lo que piensas en los microservicios es descomponer y desagregar la funcionalidad de

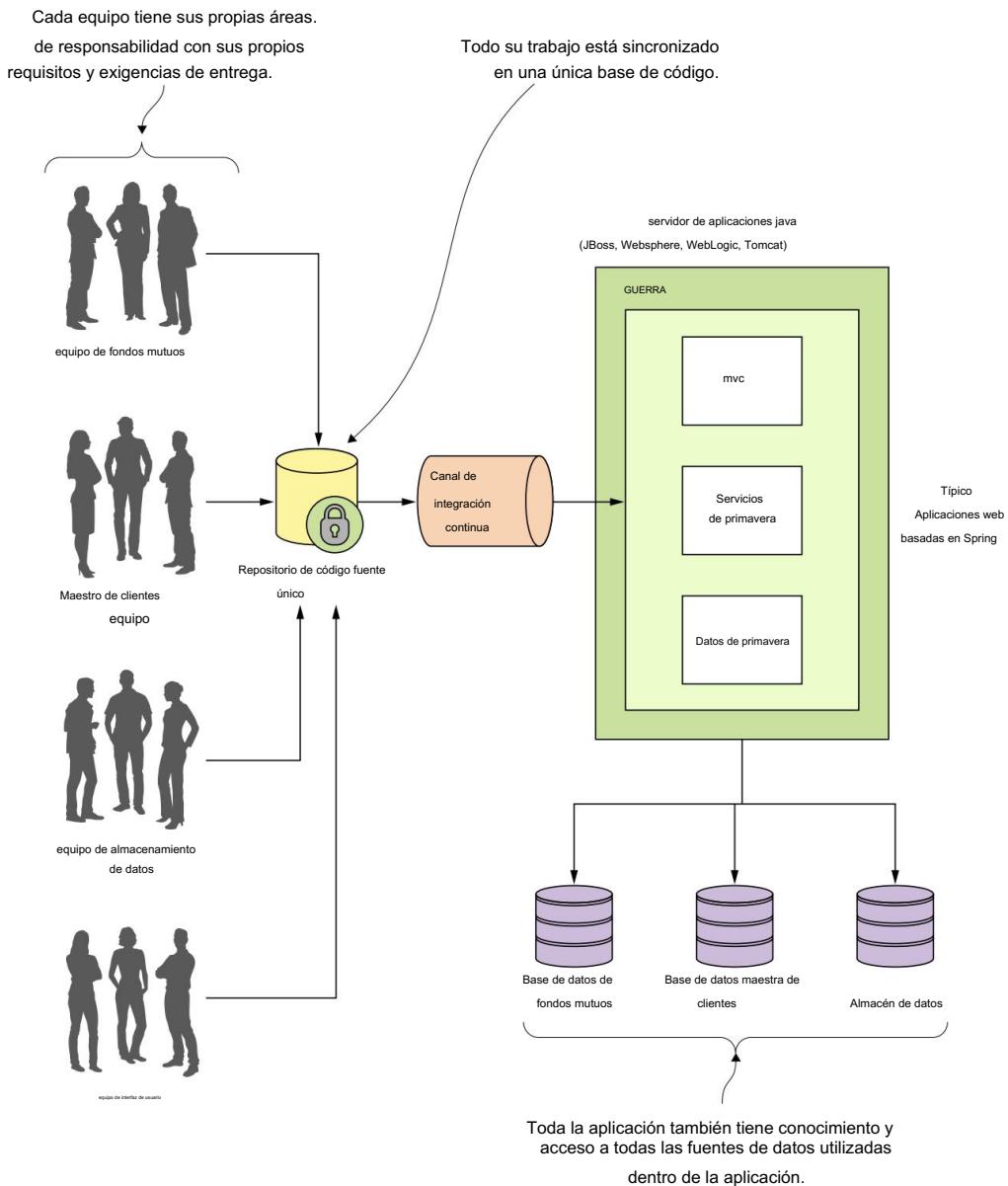


Figura 1.1 Las aplicaciones monolíticas obligan a múltiples equipos de desarrollo a sincronizar artificialmente su entrega porque su código debe construirse, probarse e implementarse como una unidad completa.

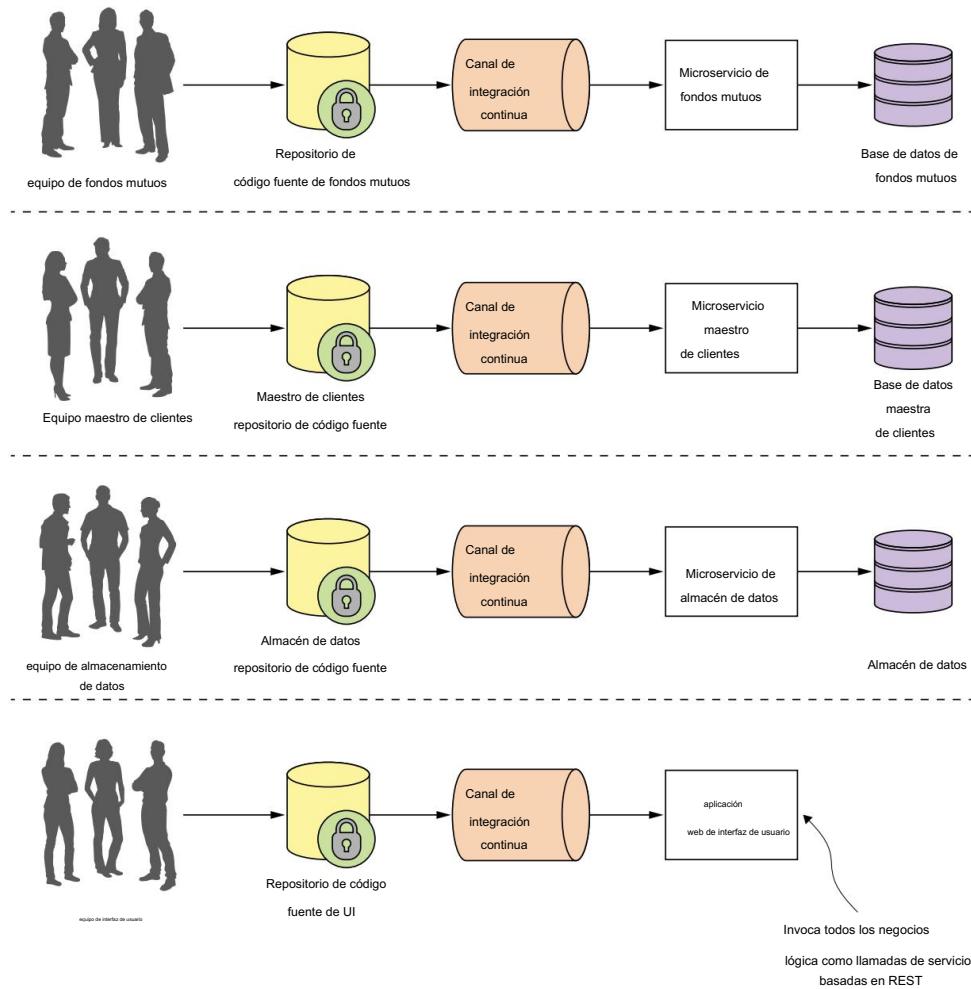


Figura 1.2 Utilizando una arquitectura de microservicios, nuestra aplicación CRM se descompondría en un conjunto de microservicios completamente independientes entre sí, lo que permitiría que cada equipo de desarrollo avanzara a su propio ritmo.

sus aplicaciones para que sean completamente independientes unas de otras. Si tomamos la aplicación CRM que vimos en la figura 1.1 y la descomponemos en microservicios, podría verse como se muestra en la figura 1.2.

Si observa la figura 1.2, puede ver que cada equipo funcional posee completamente su código de servicio y su infraestructura de servicio. Pueden construir, implementar y probar independientemente unos de otros porque su código, repositorio de control de fuente y la infraestructura (servidor de aplicaciones y base de datos) ahora son completamente independientes de las otras partes de la aplicación.

Una arquitectura de microservicio tiene las siguientes características:

La lógica de la aplicación se divide en componentes pequeños con límites de responsabilidad bien definidos que se coordinan para ofrecer una solución. Cada componente tiene un pequeño dominio de responsabilidad y se implementa de forma completamente independiente entre sí. Los microservicios deben tener responsabilidad para una sola parte de un dominio empresarial. Además, un microservicio debería ser reutilizable. en múltiples aplicaciones.

Los microservicios se comunican basándose en algunos principios básicos (tenga en cuenta que dije principios, no estándares) y emplean protocolos de comunicación livianos como HTTP y JSON (notación de objetos JavaScript) para intercambiar datos entre consumidor de servicios y proveedor de servicios.

La implementación técnica subyacente del servicio es irrelevante porque las aplicaciones siempre se comunican mediante un protocolo tecnológicamente neutro (JSON es el más común). Esto significa una aplicación creada utilizando un microservicio.

La aplicación podría construirse con múltiples lenguajes y tecnologías.

Los microservicios, por su naturaleza pequeña, independiente y distribuida, permiten organizaciones tener pequeños equipos de desarrollo con áreas de trabajo bien definidas. responsabilidad. Estos equipos podrían trabajar hacia un solo objetivo, como entregar una aplicación, pero cada equipo es responsable sólo de los servicios en los que están trabajando.

A menudo bromeo con mis colegas diciendo que los microservicios son la puerta de entrada para construir aplicaciones en la nube. Empiezas a crear microservicios porque te dan un alto grado de flexibilidad y autonomía con sus equipos de desarrollo, pero usted y su El equipo descubre rápidamente que la naturaleza pequeña e independiente de los microservicios los hace fácilmente desplegable en la nube. Una vez que los servicios están en la nube, su pequeño tamaño facilita el inicio de un gran número de instancias del mismo servicio y, de repente, sus aplicaciones se vuelven más escalables y, si se planifican con antelación, más resistentes.

## 1.2 ¿Qué es Spring y por qué es relevante para los microservicios?

Spring se ha convertido en el marco de desarrollo de facto para crear aplicaciones basadas en Java. En esencia, Spring se basa en el concepto de inyección de dependencia. En una aplicación Java normal, la aplicación se descompone en clases donde cada clase a menudo tiene vínculos explícitos con otras clases en la aplicación. Los vínculos son los invocación de un constructor de clase directamente en el código. Una vez compilado el código, Estos puntos de conexión no se pueden cambiar.

Esto es problemático en un proyecto grande porque estos vínculos externos son frágiles y realizar un cambio puede generar múltiples impactos posteriores en otro código. Un marco de inyección de dependencia, como Spring, le permite administrar más fácilmente Java de gran tamaño. proyectos externalizando la relación entre objetos dentro de su aplicación a través de convenciones (y anotaciones) en lugar de aquellos objetos que tienen códigos fijos conocimiento unos de otros. Spring actúa como intermediario entre los diferentes Java.

clases de su aplicación y gestiona sus dependencias. Básicamente, Spring te permite ensamblar tu código como un conjunto de ladrillos de Lego que se unen.

La rápida inclusión de características de Spring impulsó su utilidad, y el marco se convirtió rápidamente en una alternativa más liviana para los desarrolladores de aplicaciones empresariales Java que buscaban una forma de crear aplicaciones utilizando la pila J2EE . La pila J2EE , aunque potente, muchos la consideraban bloatware, con muchas características que nunca fueron utilizadas por los equipos de desarrollo de aplicaciones. Además, una aplicación J2EE le obligaba a utilizar un servidor de aplicaciones Java completo (y pesado) para implementar sus aplicaciones.

Lo sorprendente del marco Spring y un testimonio de su comunidad de desarrollo es su capacidad para seguir siendo relevante y reinventarse. El equipo de desarrollo de Spring vio rápidamente que muchos equipos de desarrollo se estaban alejando de las aplicaciones monolíticas donde la lógica de presentación, negocios y acceso a datos de la aplicación se empaquetaban e implementaban como un solo artefacto. En cambio, los equipos estaban pasando a modelos altamente distribuidos en los que los servicios se creaban como servicios pequeños y distribuidos que podían implementarse fácilmente en la nube. En respuesta a este cambio, el equipo de desarrollo de Spring lanzó dos proyectos: Spring Boot y Spring Cloud.

Spring Boot es una revisión del marco Spring. Si bien adopta las características principales de Spring, Spring Boot elimina muchas de las características "empresariales" que se encuentran en Spring y en su lugar ofrece un marco orientado a microservicios basados en Java y orientados a REST (Transferencia de estado representacional)<sup>1</sup> . Con unas pocas anotaciones simples, un desarrollador de Java puede crear rápidamente un microservicio REST que se puede empaquetar e implementar sin la necesidad de un contenedor de aplicaciones externo.

**NOTA** Si bien cubrimos REST con más detalle en el capítulo 2, el concepto central detrás de REST es que sus servicios deben adoptar el uso de los verbos HTTP (GET, POST, PUT y DELETE) para representar las acciones principales del servicio y utilizar un Protocolo ligero de serialización de datos orientado a web, como JSON, para solicitar y recibir datos del servicio.

Debido a que los microservicios se han convertido en uno de los patrones arquitectónicos más comunes para crear aplicaciones basadas en la nube, la comunidad de desarrollo de Spring nos ha proporcionado Spring Cloud. El marco Spring Cloud simplifica la puesta en funcionamiento y la implementación de microservicios en una nube pública o privada. Spring Cloud engloba varios marcos de microservicios de administración de la nube populares bajo un marco común y hace que el uso y la implementación de estas tecnologías sean tan fáciles de usar como anotar su código. Cubro los diferentes componentes dentro de Spring Cloud más adelante en este capítulo.

## 1.3 Qué aprenderás en este libro

Este libro trata sobre la creación de aplicaciones basadas en microservicios utilizando Spring Boot y Spring Cloud que se puede implementar en una nube privada administrada por su empresa o por una empresa pública

<sup>1</sup> Si bien cubrimos REST más adelante en el capítulo 2, vale la pena leer la tesis doctoral de Roy Fielding sobre la creación de aplicaciones basadas en REST (<http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>). Sigue siendo una de las mejores explicaciones de REST disponibles.

nube como Amazon, Google o Pivotal. Con este libro, cubrimos con prácticas ejemplos

Qué es un microservicio y las consideraciones de diseño que intervienen en la creación de un microservicio. aplicación basada en microservicios

Cuando no deberías crear una aplicación basada en microservicios

Cómo crear microservicios utilizando el marco Spring Boot

Los patrones operativos básicos que deben implementarse para respaldar el microservicio. aplicaciones, particularmente una aplicación basada en la nube

Cómo se puede utilizar Spring Cloud para implementar estos patrones operativos

Cómo tomar lo que ha aprendido y construir un canal de implementación que pueda ser

Se utiliza para implementar sus servicios en una nube privada administrada internamente o en una pública. proveedor de nube

Cuando haya terminado de leer este libro, debería tener los conocimientos necesarios para construir e implementar un microservicio basado en Spring Boot. También entenderás la clave.

Las decisiones de diseño deben hacer operativos sus microservicios. entenderás como gestión de configuración de servicios, descubrimiento de servicios, mensajería, registro y seguimiento, y la seguridad se combinan para ofrecer un entorno de microservicios sólido. Finalmente, Verá cómo sus microservicios se pueden implementar dentro de una nube pública o privada.

## 1.4 ¿Por qué este libro es relevante para usted?

Si has llegado hasta aquí leyendo el capítulo 1, sospecho que

Eres un desarrollador de Java.

Tienes experiencia en Spring. Está

interesado en aprender a crear aplicaciones basadas en microservicios. Está interesado en cómo utilizar microservicios para crear aplicaciones basadas en la nube. Quiere saber si Java y Spring son tecnologías relevantes para construir

Aplicaciones basadas en microservicios.

Le interesa ver qué implica implementar una aplicación basada en microservicios. catión a la nube.

Elegí escribir este libro por dos razones. Primero, si bien he visto muchos buenos libros sobre el aspectos conceptuales de los microservicios, no pude encontrar un buen libro basado en Java sobre la implementación de microservicios. Si bien siempre me he considerado un lenguaje de programación políglota (alguien que conoce y habla varios idiomas), Java es mi lenguaje de desarrollo principal y Spring ha sido el marco de desarrollo al que "alcanzo" cada vez que construyo una nueva aplicación. Cuando me encontré por primera vez con Spring Boot y Spring Cloud,

Me quedé asombrado. Spring Boot y Spring Cloud simplificaron enormemente mi vida de desarrollo cuando se trataba de crear aplicaciones basadas en microservicios que se ejecutan en la nube.

En segundo lugar, como he trabajado durante toda mi carrera como arquitecto e ingeniero,

He descubierto que muchas veces los libros de tecnología que compro tienden a ir a uno de dos extremos. Son conceptuales sin ejemplos de código concretos, o

son descripciones mecánicas de un marco o lenguaje de programación en particular. Quería un libro que fuera un buen puente y un punto medio entre las disciplinas de arquitectura e ingeniería. Mientras lee este libro, quiero brindarle una sólida introducción al desarrollo de patrones de microservicios y cómo se usan en el desarrollo de aplicaciones del mundo real, y luego respaldar estos patrones con ejemplos de código prácticos y fáciles de entender usando Spring. Arranque y nube de primavera.

Cambiemos de tema por un momento y analicemos la construcción de un microservicio simple usando Spring Boot.

### 1.5 Creación de un microservicio con Spring Boot

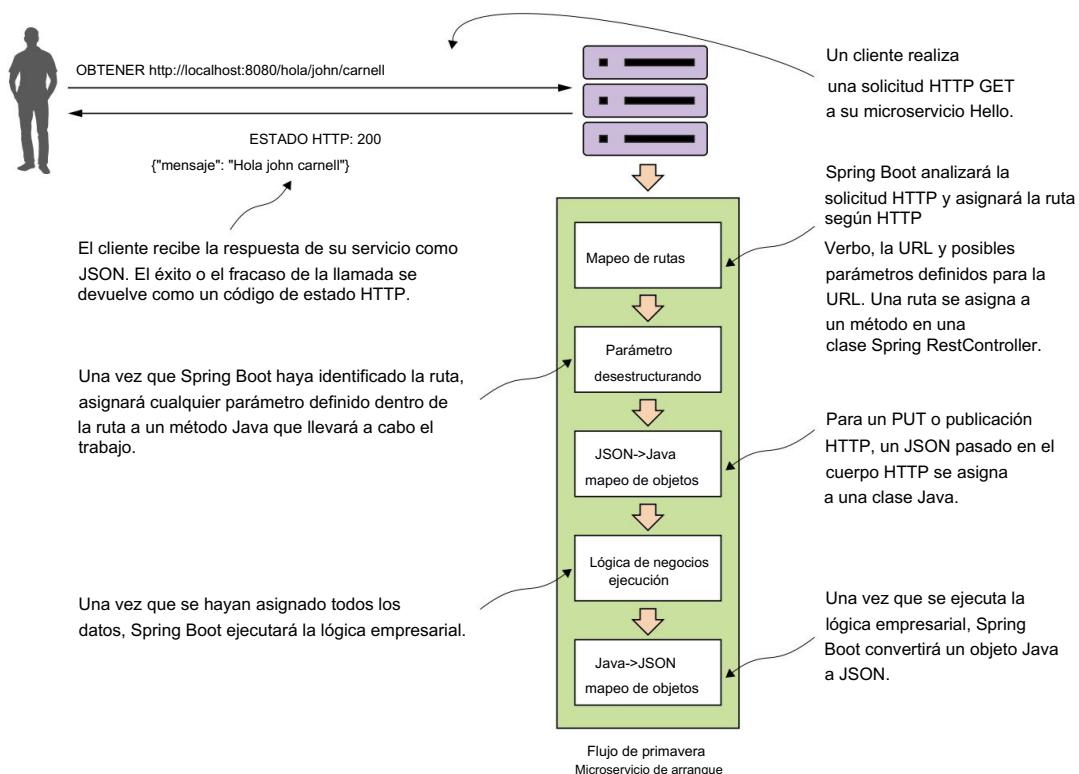
Siempre he tenido la opinión de que un marco de desarrollo de software está bien pensado y es fácil de usar si pasa lo que cariñosamente llamo la "prueba del mono Carnell". Si un mono como yo (el autor) puede descifrar un marco en 10 minutos o menos, es prometedor. Así me sentí la primera vez que escribí un servicio Spring Boot de muestra. Quiero que tengas la misma experiencia y alegría, así que tomemos un minuto para ver cómo escribir un servicio REST simple "Hola mundo" usando Spring Boot.

En esta sección, no haremos un recorrido detallado de gran parte del código presentado. Nuestro objetivo es darle una idea de cómo escribir un servicio Spring Boot. Entraremos en muchos más detalles en el capítulo 2.

La Figura 1.3 muestra lo que hará su servicio y el flujo general de cómo El microservicio Spring Boot procesará la solicitud de un usuario.

Este ejemplo no es de ninguna manera exhaustivo ni siquiera ilustrativo de cómo se debe crear un microservicio a nivel de producción, pero debería hacer que usted haga una pausa debido al poco código que se necesitó para escribirlo. No explicaremos cómo configurar los archivos de compilación del proyecto o los detalles del código hasta el capítulo 2. Si desea ver el archivo Maven pom.xml y el código real, puede encontrarlo en la Sección del capítulo 1 del código descargable. Todo el código fuente del capítulo 1 se puede recuperar del repositorio GitHub del libro en <https://github.com/carnelliJ/spmia-chapter1>.

**NOTA** Asegúrese de leer el apéndice A antes de intentar ejecutar los ejemplos de código de los capítulos de este libro. El Apéndice A cubre el diseño general de todos los proyectos del libro, cómo ejecutar los scripts de compilación y cómo iniciar el entorno Docker. Los ejemplos de código de este capítulo son simples y están diseñados para ejecutarse de forma nativa directamente desde su escritorio sin la información de capítulos adicionales. Sin embargo, en capítulos posteriores comenzará rápidamente a utilizar Docker para ejecutar todos los servicios e infraestructura utilizados en este libro. No avance demasiado en el libro sin leer el apéndice A sobre cómo configurar su entorno de escritorio.



La Figura 1.3 Spring Boot abstrae la tarea común del microservicio REST (enrutamiento a la lógica empresarial, análisis de parámetros HTTP desde la URL, mapeo de JSON hacia/desde objetos Java) y permite al desarrollador centrarse en la lógica empresarial del servicio.

Para este ejemplo, tendrá una única clase Java llamada simpleservice/src/com/thinktmechanix/application/simpleservice/Application.java que se utilizará para exponer un punto final REST llamado /hola.

El siguiente listado muestra el código de Application.java.

#### Listado 1.1 Hola mundo con Spring Boot: un microservicio Spring simple

```

paquete com.thinktmechanix.simplesservice;

importar org.springframework.boot.SpringApplication;
importar org.springframework.boot.autoconfigure.SpringBootApplication;
importar org.springframework.web.bind.annotation.RequestMapping;
importar org.springframework.web.bind.annotation.RequestMethod;
importar org.springframework.web.bind.annotation.RestController;
importar org.springframework.web.bind.annotation.PathVariable;
```

```

Le dice al marco Spring Boot que esta clase
es el punto de entrada para el servicio Spring Boot
@SpringBootApplication
@RestController

@RequestMapping(value="hola") Aplicación de
clase pública {

    public static void main(String[] args)
    { SpringApplication.run(Application.class, args);
    }

    @RequestMapping(valor="/{primerNombre}/{apellido}", método =
        RequestMethod.GET)
    public String hola( @PathVariable("firstName") String firstName, @PathVariable("lastName") String
        lastName) {
        return String.format("{\"mensaje\":\"Hola %s %s\"}",
            nombre Apellido);
    }
}

Devuelve una cadena JSON simple que usted crea
manualmente. En el capítulo 2 no crearás ningún JSON.

```

Le dice a Spring Boot que vas  
para exponer el código en esta clase  
como una clase Spring RestController

Todas las URL expuestas en esta aplicación  
estará precedido por el prefijo /hola.

Spring Boot expondrá un punto final como un  
Punto final REST basado en GET que tomará dos  
parámetros: nombre y apellido.

Asigna el nombre y el apellido  
parámetros pasados en la URL a dos  
variables pasadas a la función hola

En el listado 1.1 básicamente estás exponiendo un único punto final GET HTTP que tomará dos parámetros (nombre y apellido) en la URL y luego devolver un JSON simple cadena que tiene una carga útil que contiene el mensaje "Hola nombre apellido". Si usted llama al punto final /hello/john/carnell en su servicio (que le mostraré en breve) la devolución de la llamada sería

```
{"mensaje":"Hola john carnell"}
```

Iniciemos su servicio. Para hacer esto, vaya al símbolo del sistema y ejecute el siguiente comando:

```
mvn arranque de primavera: ejecutar
```

Este comando, mvn, utilizará un complemento Spring Boot para iniciar la aplicación mediante un Servidor Tomcat integrado.

### Java frente a Groovy y Maven frente a Gradle

El marco Spring Boot tiene un fuerte soporte para los lenguajes de programación Java y Groovy. Puede crear microservicios con Groovy y sin configuración de proyecto. Spring Boot también es compatible con las herramientas de compilación Maven y Gradle. He limitado el ejemplos de este libro para Java y Maven. Como aficionado a Groovy y Gradle desde hace mucho tiempo, tengo un gran respeto por el lenguaje y la herramienta de compilación, pero para mantener el libro manejable y el material enfocado, elegí usar Java y Maven para llegar al mayor público posible.

Nuestro punto final /hello está asignado con dos variables: nombre y apellido.

```

o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'requestContextFilter' to: [*]
s.w.s.m.m.a.RequestMappingHandlerAdapter Looking for @ControllerAdvice: org.springframework.boot.context
;startup date [Thu Mar 23 06:09:30 EDT 2017] * root of context hierarchy
s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/hello/{firstName}/{lastName}],methods=[GET]}" onto
on.hello(java.lang.String,java.lang.String)
s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[error]}" onto public org.springframework.http.Respo
nseHandlerFramework.boot.autoconfigure.web.BasicErrorController.error(javax.servlet.http.HttpServletRequest)
s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[error],produces=[text/html]}" onto public org.springfr
mfigure.web.BasicErrorController.errorHtml(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletRe
sponse)
o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto handler of type [class org.
s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler of type [class org.springfr
o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**/favicon.ico] onto handler of type [class
o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
c.t.simpleService.Application : Started Application in 2.261 seconds (JVM running for 5.113)

```

El servicio escuchará el puerto 8080 para solicitudes HTTP entrantes.

Figura 1.4 Su servicio Spring Boot comunicará los puntos finales expuestos y el puerto del servicio a través de la consola.

Si todo comienza correctamente, deberías ver lo que se muestra en la figura 1.4 desde tu ventana de línea de comandos.

Si examina la pantalla de la figura 1.4, notará dos cosas. En primer lugar, se inició un servidor Tomcat en el puerto 8080. En segundo lugar, se expone un punto final GET de /hello/ {firstName}/ {lastName} en el servidor.

HTTP GET para el punto final /hello/john/carnell

GET http://localhost:8080/hello/john/carnell

Authorization Headers Body Pre-request Script Tests

Type No Auth

Body Cookies Headers (3) Tests Status: 200 OK Time: 279 ms

Pretty Raw Preview Text

1 {"message": "Hello john carnell"}

Carga útil JSON devuelta por el servicio

Figura 1.5 La respuesta del punto final /hello muestra los datos que solicitó representados como una carga útil JSON.

Llamará a su servicio utilizando una herramienta REST basada en navegador llamada POSTMAN (<https://www.getpostman.com/>). Hay muchas herramientas, tanto gráficas como de línea de comandos, disponibles para invocar un servicio basado en REST, pero usaré POSTMAN para todos mis ejemplos en este libro. La Figura 1.5 muestra la llamada POSTMAN al punto final `http://localhost:8080/hello/john/carnell` y los resultados devueltos por el servicio.

Obviamente, este simple ejemplo no demuestra todo el poder de Spring Boot. Pero lo que debería mostrar es que se puede escribir un servicio HTTP JSON REST completo con mapeo de rutas de URL y parámetros en Java con tan solo 25 líneas de código. Como le dirá cualquier desarrollador de Java experimentado, escribir algo significativo en 25 líneas de código en Java es extremadamente difícil. Java, si bien es un lenguaje poderoso, ha adquirido la reputación de ser prolífico en comparación con otros lenguajes.

Hemos terminado con nuestro breve recorrido por Spring Boot. Ahora tenemos que hacer esta pregunta: debido a qué podemos escribir nuestras aplicaciones utilizando un enfoque de microservicio, ¿significa esto que deberíamos hacerlo? En la siguiente sección, analizaremos por qué y cuándo se justifica un enfoque de microservicio para crear sus aplicaciones.

## 1.6 ¿Por qué cambiar la forma en que creamos aplicaciones?

Estamos en un punto de inflexión en la historia. Casi todos los aspectos de la sociedad moderna están ahora conectados a través de Internet. Las empresas que solían prestar servicios a los mercados locales de repente descubren que pueden llegar a una base de clientes global. Sin embargo, con una base de clientes global más grande también viene la competencia global. Estas presiones competitivas significan que las siguientes fuerzas están impactando la forma en que los desarrolladores deben pensar en la creación de aplicaciones:

La complejidad ha aumentado considerablemente : los clientes esperan que todas las partes de una organización sepan quiénes son. Las aplicaciones "aisladas" que se comunican con una única base de datos y no se integran con otras aplicaciones ya no son la norma. Las aplicaciones actuales necesitan comunicarse con múltiples servicios y bases de datos que residen no sólo dentro del centro de datos de una empresa, sino también con proveedores de servicios externos a través de Internet. Los clientes quieren una entrega más rápida: los clientes ya no quieren esperar a la próxima versión anual o versión de un paquete de software. En cambio, esperan que las funciones de un producto de software se desagreguen para que se puedan lanzar nuevas funciones rápidamente en semanas (incluso días) sin tener que esperar a que se lance el producto completo. Rendimiento y escalabilidad: las aplicaciones globales hacen que sea extremadamente difícil predecir cuánto volumen de transacciones manejará una aplicación y cuándo llegará ese volumen de transacciones. Las aplicaciones deben ampliarse rápidamente en varios servidores y luego reducirse cuando las necesidades de volumen hayan pasado.

Los clientes esperan que sus aplicaciones estén disponibles: debido a que los clientes están a un clic de distancia de un competidor, las aplicaciones de una empresa deben ser altamente resistentes. Las fallas o problemas en una parte de la aplicación no deberían hacer que toda la aplicación caiga.

Para cumplir con estas expectativas, nosotros, como desarrolladores de aplicaciones, tenemos que aceptar la paradoja de que para crear aplicaciones altamente escalables y redundantes necesitamos romper nuestras aplicaciones en pequeños servicios que pueden construirse e implementarse independientemente de unos y otros. Si "desagregamos" nuestras aplicaciones en pequeños servicios y las movemos lejos de un único artefacto monolítico, podemos construir sistemas que sean

Flexible: los servicios desacoplados se pueden componer y reorganizar para rápidamente ofrecer nueva funcionalidad. Cuanto más pequeña sea la unidad de código con la que se trabaja, cuanto menos complicado es cambiar el código y menos tiempo lleva probarlo implementar el código.

Resiliente: los servicios desacoplados significan que una aplicación ya no es una única "bola de barro" donde una degradación en una parte de la aplicación hace que toda la aplicación falle. Las fallas se pueden localizar en una pequeña parte de la aplicación y contenerse antes de que toda la aplicación experimente una interrupción. Esto también permite que el

las aplicaciones se degraden con gracia en caso de un error irrecuperable. Escalable: los servicios desacoplados se pueden distribuir fácilmente horizontalmente entre varios servidores, lo que permite escalar las funciones/servicios de manera adecuada. Con una aplicación monolítica donde toda la lógica de la aplicación está entrelazada, toda la aplicación necesita escalar incluso si solo es una pequeña parte de la aplicación es el cuello de botella. La ampliación de servicios pequeños es localizada y mucho más rentable.

Con este fin, al comenzar nuestra discusión sobre microservicios, tenga en cuenta lo siguiente:

Servicios pequeños, simples y desacoplados = aplicaciones escalables, resistentes y flexibles

## 1.7 ¿Qué es exactamente la nube?

El término "nube" se ha usado en exceso. Cada proveedor de software tiene una nube y la plataforma de todos está habilitada para la nube, pero si deja de lado las exageraciones, tres modelos básicos existen en la computación basada en la nube. Estos son

Infraestructura como servicio (IaaS)

Plataforma como servicio (PaaS)

Software como servicio (SaaS)

Para comprender mejor estos conceptos, mapeemos la tarea diaria de preparar una comida para los diferentes modelos de computación en la nube. Cuando quieres comer, tienes cuatro opciones:

- 1 Puedes preparar la comida en casa.
- 2 Puedes ir al supermercado y comprar una comida preparada que calientes y atender.
- 3 Puedes recibir una comida en tu casa.
- 4 Puedes subir al coche y comer en el restaurante.

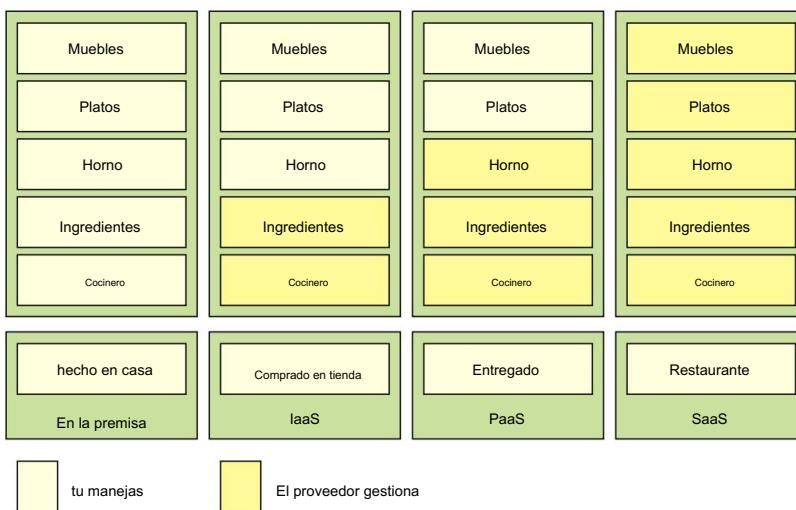


Figura 1.6 Los diferentes modelos de computación en la nube se reducen a quién es responsable de qué: el proveedor de la nube o usted.

La Figura 1.6 muestra cada modelo.

La diferencia entre estas opciones es quién es responsable de cocinar estas comidas y dónde se cocinarán. En el modelo local, comer en casa requiere que usted haga todo el trabajo, utilizando su propio horno y los ingredientes que ya tiene en casa. Una comida comprada en una tienda es como utilizar el modelo informático de infraestructura como servicio (IaaS). Estás utilizando el chef y el horno de la tienda para hornear previamente la comida, pero aún eres responsable de calentarla y comerla en la casa (y limpiar los platos después).

En un modelo de plataforma como servicio (PaaS), usted aún tiene la responsabilidad de la comida, pero además depende de un proveedor para que se encargue de las tareas principales asociadas con la preparación de una comida. Por ejemplo, en un modelo PaaS, tú suministras los platos y los muebles, pero el dueño del restaurante proporciona el horno, los ingredientes y el chef para cocinarlos. En el modelo de software como servicio (SaaS), vas a un restaurante donde te preparan toda la comida. Comes en el restaurante y luego pagas la comida cuando terminas. Tampoco tienes platos que preparar o lavar.

Los elementos clave en juego en cada uno de estos modelos son los de control: ¿quién es responsable del mantenimiento de la infraestructura y cuáles son las opciones tecnológicas disponibles para crear la aplicación? En un modelo IaaS, el proveedor de la nube proporciona la infraestructura básica, pero usted es responsable de seleccionar la tecnología y crear la solución final. En el otro extremo del espectro, con un modelo SaaS, usted es un consumidor pasivo del servicio proporcionado por el proveedor y no tiene participación en la selección de tecnología ni responsabilidad alguna para mantener la infraestructura de la aplicación.

### Plataformas en la nube emergentes

He documentado los tres tipos principales de plataformas en la nube (IaaS, PaaS, SaaS) que se encuentran en utilizar hoy. Sin embargo, están surgiendo nuevos tipos de plataformas en la nube. Estas nuevas plataformas incluye funciones como servicio (FaaS) y contenedor como servicio (CaaS). Basado en FaaS ([https://en.wikipedia.org/wiki/Function\\_as\\_a\\_Service](https://en.wikipedia.org/wiki/Function_as_a_Service)) Las aplicaciones utilizan tecnologías como las tecnologías Lambda de Amazon y las funciones de Google Cloud para crear aplicaciones implementadas como fragmentos de código "sin servidor" que se ejecutan completamente en la nube.

infraestructura informática de la plataforma del proveedor. Con una plataforma FaaS, no es necesario gestionar cualquier infraestructura de servidor y pagar solo por los ciclos informáticos necesarios para ejecutar la función.

Con el modelo de contenedor como servicio (CaaS), los desarrolladores construyen e implementan sus microservicios como contenedores virtuales portátiles (como Docker) a un proveedor de nube.

A diferencia de un modelo IaaS, donde usted, el desarrollador, debe administrar la máquina virtual, donde se implementa el servicio, con CaaS usted está implementando sus servicios de forma liviana contenedor virtual. El proveedor de la nube ejecuta el servidor virtual que ejecuta el contenedor, así como las herramientas integrales del proveedor para construir, implementar, monitorear, y escalar contenedores. El Elastic Container Service (ECS) de Amazon es un ejemplo de Plataforma basada en CaaS. En el capítulo 10 de este libro, veremos cómo implementar el microservicios que ha creado para Amazon ECS.

Es importante tener en cuenta que tanto con los modelos FaaS como CaaS de computación en la nube, aún puede crear una arquitectura basada en microservicios. Recuerde, el concepto de Los microservicios giran en torno a la creación de pequeños servicios, con responsabilidad limitada, utilizando una interfaz basada en HTTP para comunicarse. Las plataformas emergentes de computación en la nube, como FaaS y CaaS, en realidad tratan de mecanismos de infraestructura alternativos para implementar microservicios.

## 1.8 ¿Por qué la nube y los microservicios?

Uno de los conceptos centrales de una arquitectura basada en microservicios es que cada servicio es empaquetado e implementado como su propio artefacto discreto e independiente. Instancias de servicio debe abrirse rápidamente y cada instancia del servicio debe ser indistinguible de otra.

Como desarrollador que escribe un microservicio, tarde o temprano tendrás que decidir si su servicio se implementará en uno de los siguientes:

Servidor físico : si bien puede crear e implementar sus microservicios en una máquina física, pocas organizaciones lo hacen porque los servidores físicos están limitados. No se puede aumentar rápidamente la capacidad de un servidor físico y puede volverse extremadamente costoso escalar su microservicio horizontalmente en múltiples servidores físicos.

Imágenes de máquinas virtuales: uno de los beneficios clave de los microservicios es su capacidad para iniciar y cerrar rápidamente instancias de microservicios en respuesta a eventos de escalabilidad y fallas del servicio. Las máquinas virtuales son el corazón y el alma de la

principales proveedores de nube. Un microservicio se puede empaquetar en una máquina virtual La imagen y múltiples instancias del servicio se pueden implementar rápidamente y iniciado en una nube pública o privada de IaaS.

Contenedor virtual: los contenedores virtuales son una extensión natural de la implementación de su microservicios en una imagen de máquina virtual. En lugar de implementar un servicio en un máquina virtual completa, muchos desarrolladores implementan sus servicios como contenedores Docker (o tecnología de contenedor equivalente) en la nube. Ejecución de contenedores virtuales dentro de una máquina virtual; Al utilizar un contenedor virtual, puede separar una única máquina virtual en una serie de procesos autónomos que comparten la misma máquina virtual.  
Imagen de la máquina.

La ventaja de los microservicios basados en la nube se centra en el concepto de elasticidad.

Los proveedores de servicios en la nube le permiten poner en marcha rápidamente nuevas máquinas virtuales y contenedores en cuestión de minutos. Si sus necesidades de capacidad para sus servicios disminuyen, puede girar desactivar servidores virtuales sin incurrir en ningún coste adicional. Usar un proveedor de nube para implementar sus microservicios le brinda una escalabilidad horizontal significativamente mayor (agregar más servidores e instancias de servicio) para sus aplicaciones. La elasticidad del servidor también significa que sus aplicaciones puedan ser más resistentes. Si uno de sus microservicios tiene problemas y está fallando, crear nuevas instancias de servicio puede mantener viva su aplicación el tiempo suficiente para que su equipo de desarrollo resuelva el problema correctamente.

Para este libro, se incluirán todos los microservicios y la infraestructura de servicios correspondiente. implementarse en un proveedor de nube basado en IaaS utilizando contenedores Docker. Esta es una topología de implementación común utilizada para microservicios:

Gestión de infraestructura simplificada: los proveedores de nube IaaS le ofrecen la posibilidad de tenga el mayor control sobre sus servicios. Se pueden iniciar nuevos servicios y detenido con simples llamadas API . Con una solución en la nube IaaS, solo paga por infraestructura que utiliza.

Escalabilidad horizontal masiva: los proveedores de nube IaaS le permiten iniciar de forma rápida y concisa una o más instancias de un servicio. Esta capacidad significa que puedes escalar rápidamente los servicios y evitar servidores que fallan o se comportan mal.

Alta redundancia a través de la distribución geográfica: por necesidad, los proveedores de IaaS tienen múltiples centros de datos. Implementando sus microservicios utilizando una nube IaaS proveedor, puede obtener un mayor nivel de redundancia más allá del uso de clústeres en un centro de datos.

### ¿Por qué no microservicios basados en PaaS?

Anteriormente en el capítulo analizamos tres tipos de plataformas en la nube (infraestructura como un servicio, una plataforma como servicio y un software como servicio). Para este libro, he elegido centrarme específicamente en la creación de microservicios utilizando un enfoque basado en IaaS.

Si bien ciertos proveedores de nube le permitirán abstraer la infraestructura de implementación para su microservicio, elegí permanecer independiente del proveedor e implementar todas las partes de mi aplicación (incluidos los servidores).

Por ejemplo, Amazon, Cloud Foundry y Heroku le brindan la posibilidad de implementar sus servicios sin tener que conocer el contenedor de la aplicación subyacente. Proporcionan una interfaz web y API para permitirle implementar su aplicación como WAR o JAR.

La configuración y el ajuste del servidor de aplicaciones y el correspondiente contenedor de Java no le corresponden a usted. Si bien esto es conveniente, la opción de cada proveedor de nube La plataforma tiene diferentes idiosincrasias relacionadas con su solución PaaS individual.

Un enfoque IaaS, si bien genera más trabajo, es portátil a través de múltiples proveedores de nube y nos permite llegar a un público más amplio con nuestro material. Personalmente he descubierto que Las soluciones en la nube basadas en PaaS pueden permitirle iniciar rápidamente su desarrollo esfuerzo, pero una vez que su aplicación alcanza suficientes microservicios, comienza a necesitar flexibilidad que proporciona el estilo IaaS de desarrollo en la nube.

Anteriormente en este capítulo, mencioné nuevas plataformas de computación en la nube como Function como servicio (FaaS) y contenedor como servicio (CaaS). Si no tiene cuidado, las plataformas basadas en FaaS pueden bloquear su código en una plataforma de proveedor de nube porque su código se implementa en un motor de ejecución específico del proveedor. Con un modelo basado en FaaS, es posible que Estará escribiendo su servicio usando un lenguaje de programación general (Java, Python, JavaScript, etc.), pero todavía está fuertemente vinculado a las API del proveedor subyacente. y motor de tiempo de ejecución en el que se implementará su función.

Los servicios creados en este libro están empaquetados como contenedores Docker. Una de las razones La razón por la que elegí Docker es que, como tecnología de contenedor, Docker se puede implementar en todos los principales proveedores de nube. Más adelante, en el capítulo 10, demuestro cómo empaquetar microservicios usando Docker y luego implementar estos contenedores en la plataforma en la nube de Amazon.

## 1.9 Los microservicios son más que escribir código

Si bien los conceptos relacionados con la creación de microservicios individuales son fáciles de entender, ejecutar y soportar una aplicación de microservicio robusta (especialmente cuando se ejecuta

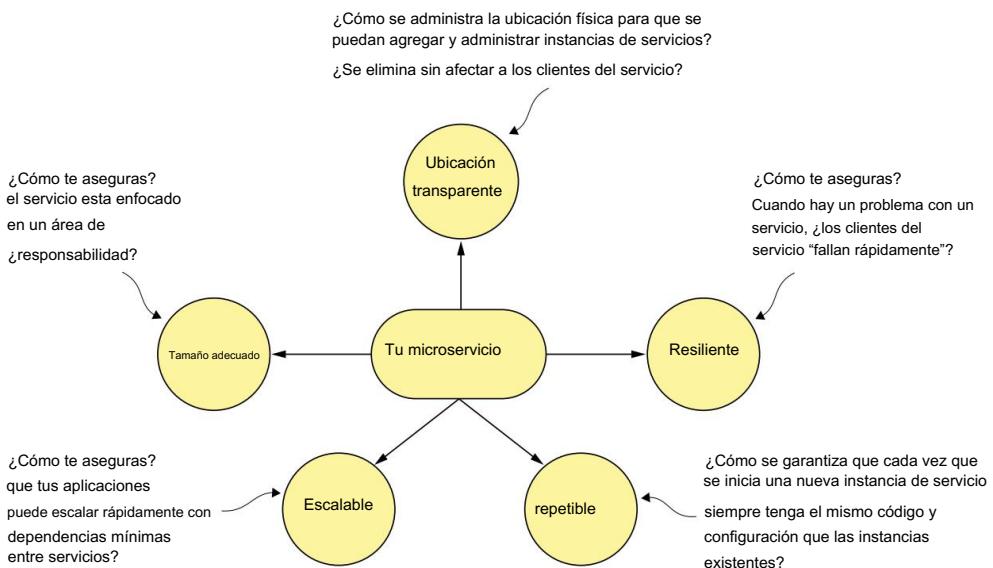


Figura 1.7 Los microservicios son más que la lógica empresarial. Es necesario pensar en el entorno en el que se ejecutarán los servicios y en cómo escalarán y serán resilientes.

en la nube) implica más que escribir el código del servicio. Escribir un servicio sólido incluye considerar varios temas. La Figura 1.7 destaca estos temas.

Repasemos los elementos de la figura 1.7 con más detalle:

**Tamaño adecuado**: ¿cómo puede garantizar que sus microservicios tengan el tamaño adecuado para que ¿Que no tienes un microservicio que asuma demasiada responsabilidad? Recordar, Con el tamaño adecuado, un servicio le permite realizar cambios rápidamente en una aplicación. y reduce el riesgo general de una interrupción en toda la aplicación.

**Ubicación transparente**: cómo gestionamos los detalles físicos de la invocación del servicio cuando en una aplicación de microservicio, múltiples instancias de servicio pueden rápidamente iniciar y apagar?

**Resiliente**: ¿Cómo protege a sus consumidores de microservicios y a la comunidad en general? integridad de su aplicación evitando los servicios defectuosos y garantizando ¿Que adoptas un enfoque de "fallo rápido"?

**Repetible**: ¿cómo puede asegurarse de que cada nueva instancia de su servicio genere Se garantiza que up tendrá la misma configuración y base de código que todos los demás. instancias de servicio en producción?

**Escalable**: ¿cómo se utilizan eventos y procesamiento asincrónico para minimizar el dependencias directas entre sus servicios y garantizar que pueda acceder con gracia escalar sus microservicios?

Este libro adopta un enfoque basado en patrones al responder estas preguntas. Con un enfoque basado en patrones, presentamos diseños comunes que se pueden utilizar en diferentes

implementaciones de tecnología. Si bien hemos elegido usar Spring Boot y Spring Cloud para implementar los patrones que usaremos en este libro, nada se mantendrá que usted tome los conceptos presentados aquí y los utilice con otra tecnología plataformas. Específicamente, cubrimos las siguientes seis categorías de patrones de microservicios:

- Patrones básicos de desarrollo.
- Patrones de enrutamiento
- Patrones de resiliencia del cliente
- Patrones de seguridad
- Registro y seguimiento de patrones
- Patrones de construcción e implementación

Repasemos estos patrones con más detalle.

### 1.9.1 Patrón de desarrollo de microservicios principales

El patrón de desarrollo de microservicios de desarrollo central aborda los conceptos básicos de construir un microservicio. La Figura 1.8 resalta los temas que cubriremos en torno al diseño básico de servicios.

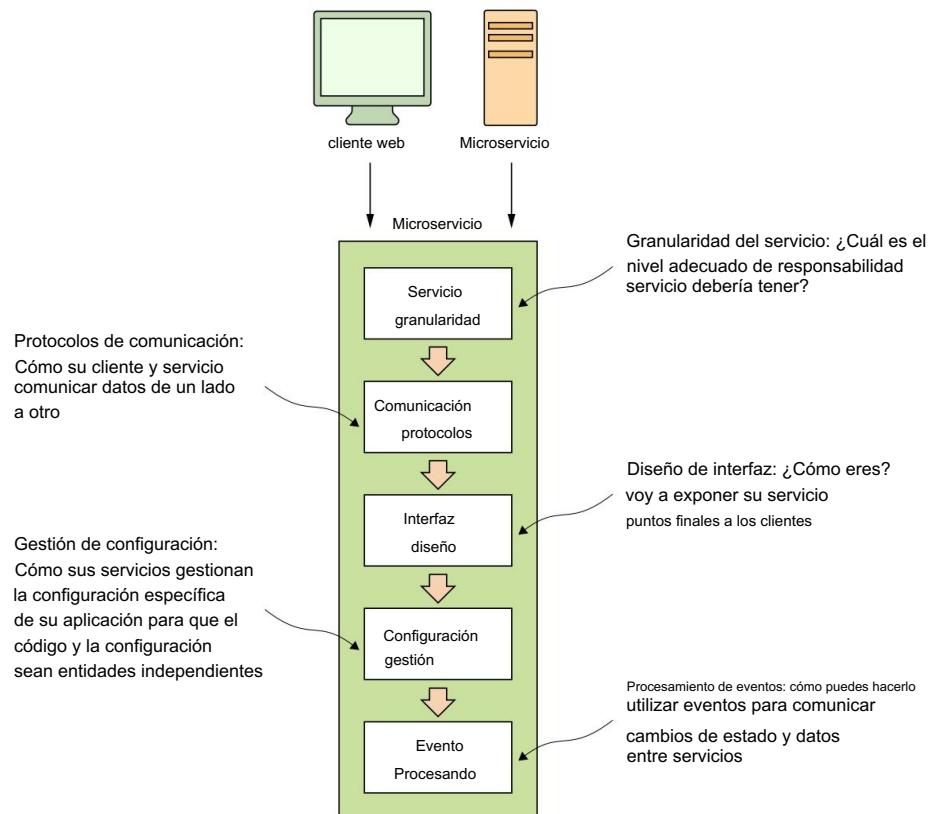


Figura 1.8 Al diseñar su microservicio, debe pensar en cómo se consumirá y se comunicará el servicio.

Granularidad del servicio: ¿cómo se aborda la descomposición de un dominio empresarial en microservicios para que cada microservicio tenga el nivel correcto de responsabilidad? Hacer que un servicio sea demasiado generalizado con responsabilidades que se superponen en diferentes dominios de problemas empresariales hace que el servicio sea difícil de mantener y cambiar con el tiempo. Hacer que el servicio sea demasiado detallado aumenta la complejidad general de la aplicación y convierte el servicio en un servicio de datos "tonto".

capa de abstracción sin lógica excepto la necesaria para acceder al almacén de datos. | Cubra la granularidad del servicio en el capítulo 2.

Protocolos de comunicación: ¿cómo se comunicarán los desarrolladores con su servicio?

¿Utiliza XML (lenguaje de marcado extensible), JSON (notación de objetos JavaScript) o un protocolo binario como Thrift para enviar datos de un lado a otro de sus microservicios? Analizaremos por qué JSON es la opción ideal para microservicios y se ha convertido en la opción más común para enviar y recibir datos a microservicios. Cubro los protocolos de comunicación en el capítulo 2.

Diseño de interfaz : ¿cuál es la mejor manera de diseñar las interfaces de servicio reales que

¿Los desarrolladores van a utilizar para llamar a su servicio? ¿Cómo estructura las URL de sus servicios para comunicar la intención del servicio? ¿Qué pasa con el versionado de sus servicios?

Una interfaz de microservicio bien diseñada hace que el uso de su servicio sea intuitivo. yo cubro diseño de la interfaz en el capítulo 2.

Gestión de la configuración del servicio: ¿cómo se gestiona la configuración del servicio?

su microservicio para que a medida que se mueve entre diferentes entornos en la nube, ¿nunca tendrá que cambiar el código o la configuración de la aplicación principal? | Cubra la gestión de la configuración del servicio en el capítulo 3.

Procesamiento de eventos entre servicios: ¿cómo desacopla su microservicio usando

eventos para minimizar las dependencias codificadas entre sus servicios

y aumentar la resiliencia de su aplicación? Cubro el procesamiento de eventos

entre servicios en el capítulo 8.

### 1.9.2 Patrones de enrutamiento de microservicios

Los patrones de enrutamiento de microservicios abordan cómo una aplicación cliente que desea consumir un microservicio descubre la ubicación del servicio y se enruta a él.

En una aplicación basada en la nube, es posible que tenga cientos de instancias de microservicios en ejecución.

Deberá abstraer la dirección IP física de estos servicios y tener una

único punto de entrada para llamadas de servicio para que pueda hacer cumplir consistentemente la seguridad y políticas de contenido para todas las llamadas de servicio.

El descubrimiento y el enrutamiento de servicios responden a la pregunta: "¿Cómo obtengo la información de mi cliente?". solicitud de un servicio a una instancia específica de un servicio?

Descubrimiento de servicios: ¿cómo puede hacer que su microservicio sea reconocible para que el cliente | ¿Las aplicaciones pueden encontrarlos sin tener la ubicación del servicio codificada en la aplicación?

¿Cómo se garantiza que el microservicio que se comporta mal?

¿Se eliminan las instancias del grupo de instancias de servicio disponibles? Cubro el descubrimiento de servicios en el capítulo 4.

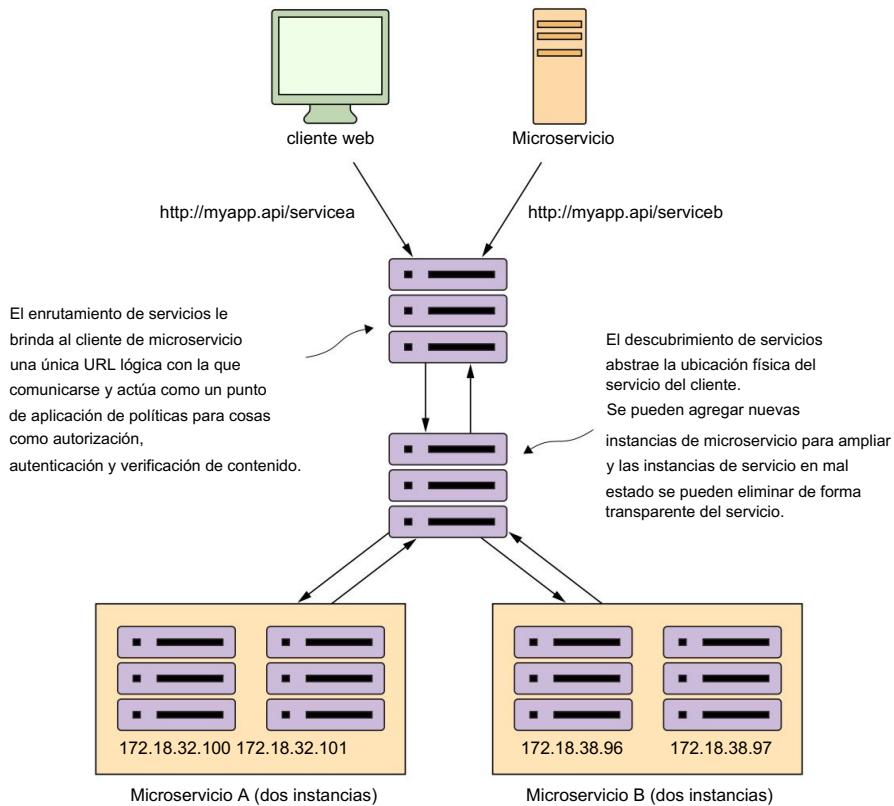


Figura 1.9 El descubrimiento y el enruteamiento de servicios son partes clave de cualquier aplicación de microservicio a gran escala.

Enrutamiento de servicios: ¿cómo puede proporcionar un único punto de entrada para todos sus servicios, de modo que que las políticas de seguridad y las reglas de enruteamiento se apliquen uniformemente a múltiples servicios e instancias de servicio en sus aplicaciones de microservicio? ¿Cómo se asegura de que cada desarrollador de su equipo no tiene que idear sus propias soluciones para proporcionando enruteamiento a sus servicios? Cubro el enruteamiento de servicios en el capítulo 6.

En la figura 1.9, el descubrimiento de servicios y el enruteamiento de servicios parecen tener un código fijo. secuencia de eventos entre ellos (primero viene el enruteamiento del servicio y el descubrimiento del servicio). Sin embargo, los dos patrones no dependen uno del otro. Por ejemplo, nosotros Puede implementar el descubrimiento de servicios sin enruteamiento de servicios. Puedes implementar el servicio. enruteamiento sin descubrimiento de servicios (aunque su implementación es más difícil).

### 1.9.3 Patrones de resiliencia del cliente de microservicio

Debido a que las arquitecturas de microservicios están altamente distribuidas, hay que ser extremadamente sensible en cómo evitar que se produzca un problema en un solo servicio (o instancia de servicio).

en cascada hacia los consumidores del servicio. Con este fin, cubriremos cuatro patrones de resiliencia del cliente:

**Equilibrio de carga del lado del cliente:** ¿cómo se almacena en caché la ubicación de su servicio?

instancias en el cliente de servicio para que las llamadas a múltiples instancias de un microservicio tengan carga equilibrada para todas las instancias de salud de ese microservicio?

**Patrón de disyuntores:** ¿cómo se evita que un cliente siga llamando a un

¿Servicio que está fallando o sufre problemas de rendimiento? Cuando un servicio se ejecuta lentamente, consume recursos del cliente que lo llama. quieres fallar

Las llamadas de microservicio fallan rápidamente para que el cliente que llama pueda responder rápidamente y tomar una acción adecuada.

**Patrón de respaldo:** cuando falla una llamada de servicio, ¿cómo se puede proporcionar un mecanismo de "complemento" que permita al cliente del servicio intentar llevar a cabo su trabajo a través de medios alternativos además del microservicio al que se llama?

**Patrón de mamparo :** las aplicaciones de microservicio utilizan múltiples recursos distribuidos

para realizar su trabajo. ¿Cómo se compartimentan estas llamadas para que el mal comportamiento de una llamada de servicio no afecte negativamente al resto de la aplicación?

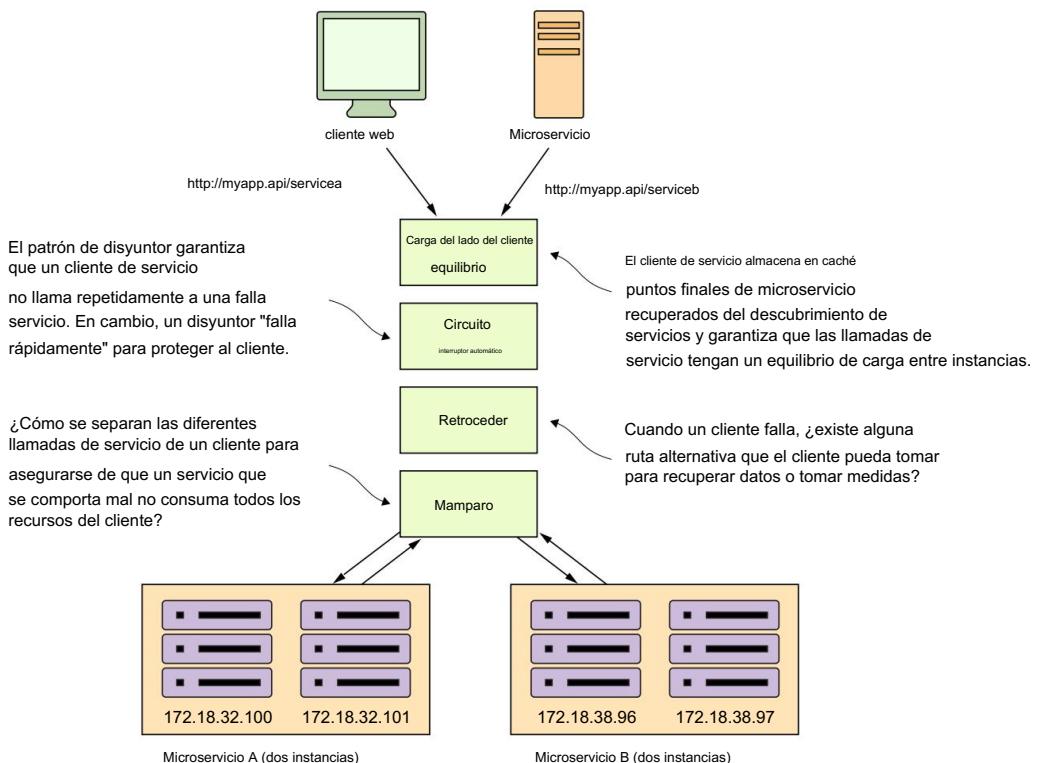


Figura 1.10 Con los microservicios, debe proteger a la persona que llama del servicio de un servicio que se comporta mal. Recuerde, un servicio lento o inactivo puede causar interrupciones más allá del servicio inmediato.

La Figura 1.10 muestra cómo estos patrones protegen al consumidor de un servicio de verse afectado cuando un servicio se comporta mal. Cubro estos cuatro temas en el capítulo 5.

#### 1.9.4 Patrones de seguridad de microservicios

No puedo escribir un libro sobre microservicios sin hablar de la seguridad de los microservicios. En el capítulo 7 cubriremos tres patrones de seguridad básicos. Estos patrones son

Autenticación: ¿cómo se determina que el cliente del servicio que llama al servicio es ¿quiénes dicen que son?

Autorización: ¿cómo se determina si el cliente del servicio que llama a un microservicio tiene permiso para realizar la acción que está intentando realizar? Gestión y propagación de

creenciales: ¿Cómo se puede evitar que un cliente de servicio tenga que presentar constantemente sus credenciales para las llamadas de servicio involucradas en una transacción? Específicamente, veremos cómo se pueden usar los estándares de seguridad basados en tokens, como OAuth2 y JavaScript Web Tokens (JWT), para obtener un token que se puede pasar de una llamada de servicio a otra para autenticar y autorizar al usuario.

La Figura 1.11 muestra cómo puede implementar los tres patrones descritos anteriormente para crear un servicio de autenticación que pueda proteger sus microservicios.

En este punto no voy a profundizar demasiado en los detalles de la figura 1.10. Hay una razón por la cual la seguridad requiere un capítulo completo. (Honestamente, podría ser un libro en sí mismo).

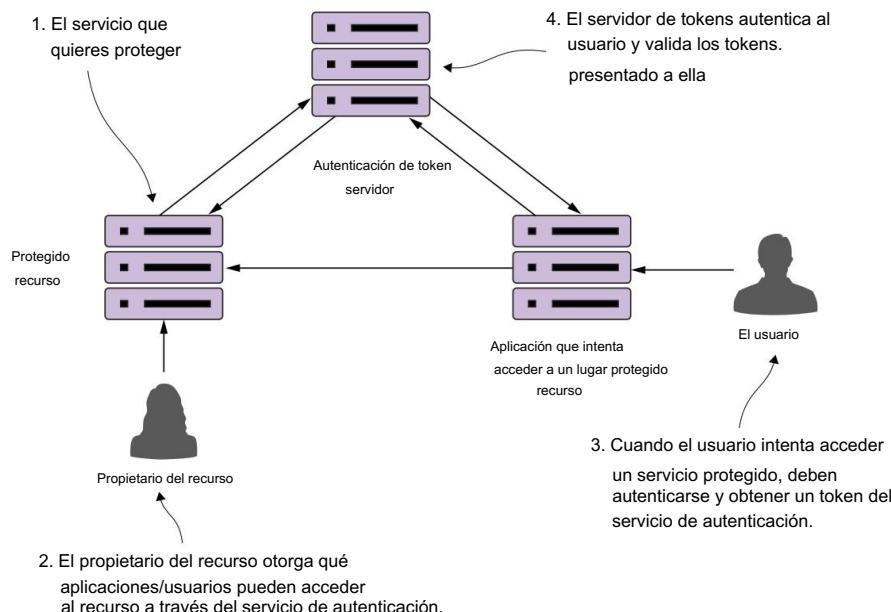


Figura 1.11 Al utilizar un esquema de seguridad basado en tokens, puede implementar la autenticación y autorización del servicio sin pasar las credenciales del cliente.

### 1.9.5 Patrones de registro y seguimiento de microservicios

La belleza de la arquitectura de microservicios es que una aplicación monolítica está rota en pequeñas piezas de funcionalidad que se pueden implementar independientemente de uno otro. La desventaja de una arquitectura de microservicio es que es mucho más difícil para depurar y rastrear qué diablos está sucediendo dentro de su aplicación y servicios.

Por este motivo, veremos tres patrones principales de registro y seguimiento:

Correlación de registros: ¿cómo se unen todos los registros producidos entre servicios? para una transacción de un solo usuario? Con este patrón, veremos cómo implementar un ID de correlación, que es un identificador único que se llevará a través de todos los servicios llamadas en una transacción y se puede utilizar para unir las entradas de registro producidas a partir de cada servicio.

Agregación de registros: con este patrón veremos cómo reunir todos los registros producidos por sus microservicios (y sus instancias individuales) en una única base de datos consultable. También veremos cómo utilizar ID de correlación para ayudar en buscando en sus registros agregados.

Seguimiento de microservicios : finalmente, exploraremos cómo visualizar el flujo de un cliente. transacción en todos los servicios involucrados y comprender el desempeño características de los servicios involucrados en la transacción.

La figura 1.12 muestra cómo encajan estos patrones. Cubriremos el registro y el rastreo. patrones con mayor detalle en el capítulo 9.

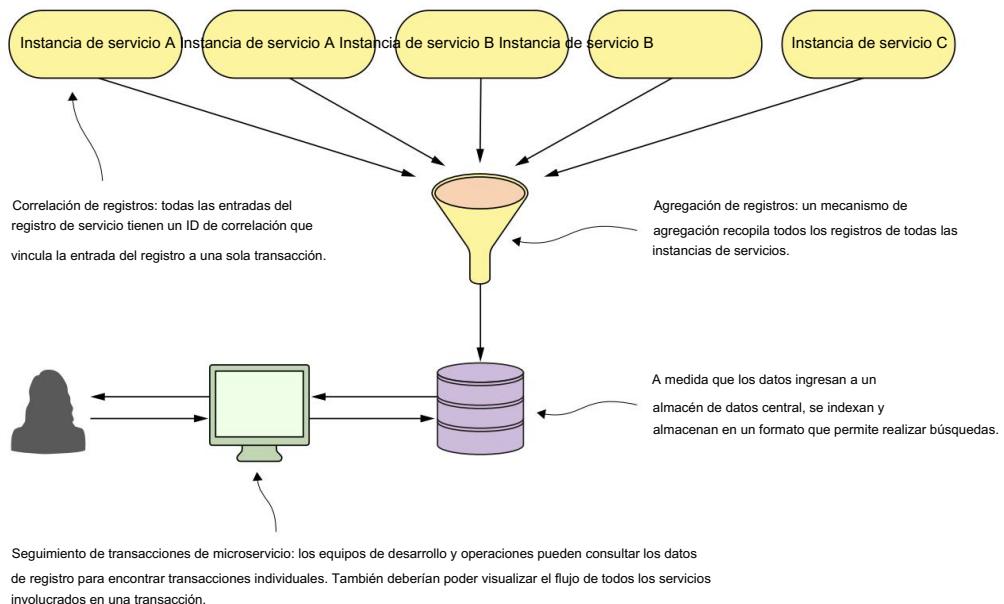


Figura 1.12 Una estrategia de registro y seguimiento bien pensada hace que la depuración de transacciones en múltiples servicios sea manejable.

## 1.9.6 Patrones de construcción/implementación de microservicios

Una de las partes centrales de una arquitectura de microservicio es que cada instancia de un microservicio debe ser idéntico a todas sus demás instancias. No puede permitir que se produzca una "derivación de la configuración" (algo cambia en un servidor después de su implementación), porque esto puede introducir inestabilidad en sus aplicaciones.

Una frase que se dice con demasiada frecuencia

"Sólo hice un pequeño cambio en el servidor de escenario, pero olvidé hacer el cambio en producción." La resolución de muchos sistemas caídos cuando he trabajado en equipos de situaciones críticas a lo largo de los años a menudo ha comenzado con esas palabras de un desarrollador o administrador de sistema. Los ingenieros (y la mayoría de las personas en general) operan con buena intenciones. No se ponen a trabajar para cometer errores o derribar sistemas. En cambio están haciendo lo mejor que pueden, pero están ocupados o distraídos. Modifican algo en un servidor, con la intención de volver atrás y hacerlo en todos los entornos.

Más tarde, se produce un apagón y todo el mundo se queda preguntándose qué es diferente entre los entornos inferiores en producción. He descubierto que el tamaño pequeño y el alcance limitado de un microservicio lo convierten en la oportunidad perfecta para introducir el concepto de "infraestructura inmutable" en una organización: una vez que un servicio una vez desplegado, la infraestructura en la que se ejecuta nunca más será tocada por manos humanas.

Una infraestructura inmutable es una pieza fundamental para utilizar con éxito un microservicio arquitectura, porque hay que garantizar en producción que cada microservicio La instancia que inicia para un microservicio en particular es idéntica a la de sus hermanos.

Con este fin, nuestro objetivo es integrar la configuración de su infraestructura directamente en su proceso de compilación e implementación para que ya no implemente artefactos de software como un Java WAR o EAR a una pieza de infraestructura que ya está en ejecución. En cambio, quieres cree y compile su microservicio y la imagen del servidor virtual en el que se ejecuta como parte del proceso de construcción. Luego, cuando se implemente su microservicio, todo Se implementa la imagen de la máquina con el servidor ejecutándose.

La figura 1.13 ilustra este proceso. Al final del libro veremos cómo Cambie su canal de compilación e implementación para que sus microservicios y servidores sobre los que se ejecutan se implementan como una única unidad de trabajo. En el capítulo 10 cubrimos lo siguiente patrones y temas:

Canal de compilación e implementación: ¿cómo se crea un proceso de compilación e implementación repetible que enfatice la compilación y la implementación con un solo botón en cualquier entorno de su organización?

Infraestructura como código: ¿cómo trata el aprovisionamiento de sus servicios como código? que se puede ejecutar y gestionar bajo control de código fuente? Servidores inmutables: una vez creada una imagen de microservicio, ¿cómo se asegura ¿Que nunca ha cambiado después de haber sido implementado?

Servidores Phoenix: cuanto más tiempo esté funcionando un servidor, mayor será la posibilidad de que se produzcan cambios en la configuración. ¿Cómo se asegura que los servidores que ejecutan microservicios se dañen? ¿Se derriba periódicamente y se recrea a partir de una imagen inmutable?

Todo comienza cuando un desarrollador registra su código en un repositorio de control de fuente. Este es el desencadenante para comenzar el proceso de construcción/implementación.

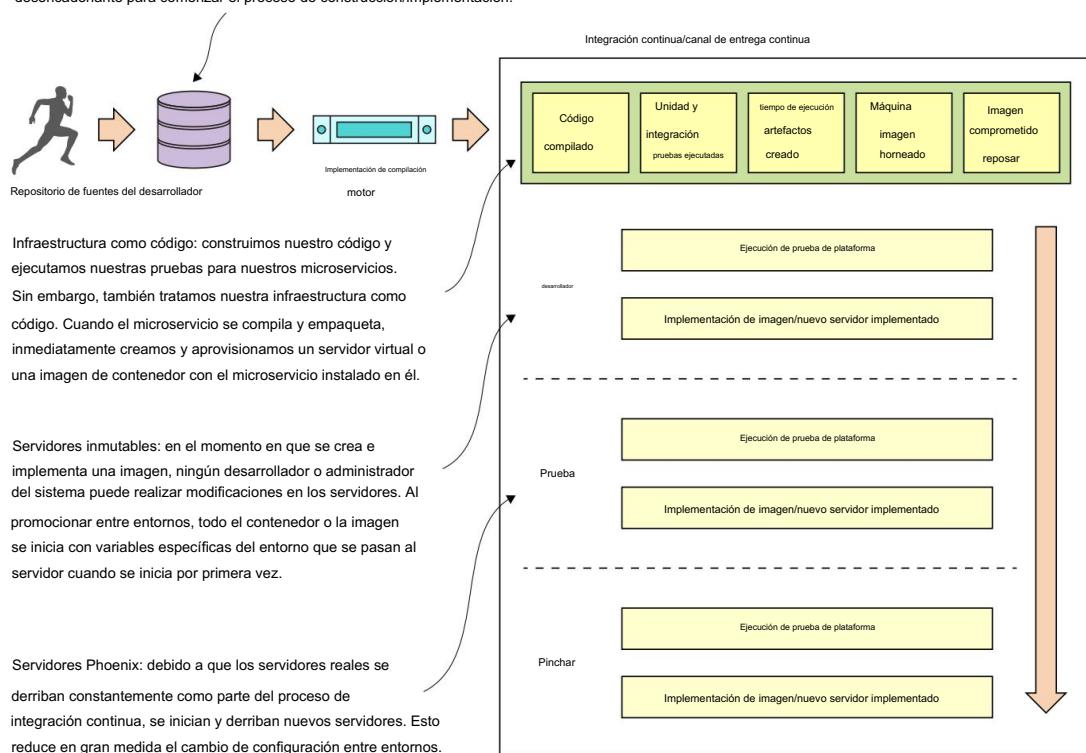


Figura 1.13 Quiere que la implementación del microservicio y el servidor en el que se ejecuta sea un artefacto atómico que se implemente como un todo entre entornos.

Nuestro objetivo con estos patrones y temas es exponer y erradicar despiadadamente la deriva de configuración lo más rápido posible antes de que pueda afectar a sus entornos superiores, como por ejemplo escenario o producción.

**NOTA** Para los ejemplos de código de este libro (excepto el capítulo 10), todo se ejecutará localmente en su máquina de escritorio. Los dos primeros capítulos se pueden ejecutar de forma nativa directamente desde la línea de comando. A partir del capítulo 3, todo el código se compilará y ejecutará como contenedores Docker.

## 1.10 Uso de Spring Cloud para crear sus microservicios

En esta sección, presento brevemente las tecnologías Spring Cloud que usará cuando Desarrolle sus microservicios. Esta es una descripción general de alto nivel; Cuando utilices cada tecnología de este libro, te enseñaré los detalles de cada una según sea necesario.

Implementar todos estos patrones desde cero supondría una enorme cantidad de trabajo.

Afortunadamente para nosotros, el equipo de Spring ha integrado una gran cantidad de proyectos de código abierto probados en batalla en un subproyecto de Spring conocido colectivamente como Spring Cloud. (<http://projects.spring.io/spring-cloud/>).

Spring Cloud engloba el trabajo de empresas de código abierto como Pivotal, HashiCorp y Netflix en la entrega de patrones. Spring Cloud simplifica la instalación y configuración de estos proyectos en su aplicación Spring para que pueda concentrarse en escribir código, sin perderse en los detalles de la configuración de toda la infraestructura que puede acompañar a la creación e implementación de una aplicación de microservicio.

La Figura 1.14 asigna los patrones enumerados en la sección anterior a los proyectos de Spring Cloud que los implementan.

Analicemos estas tecnologías con mayor detalle.

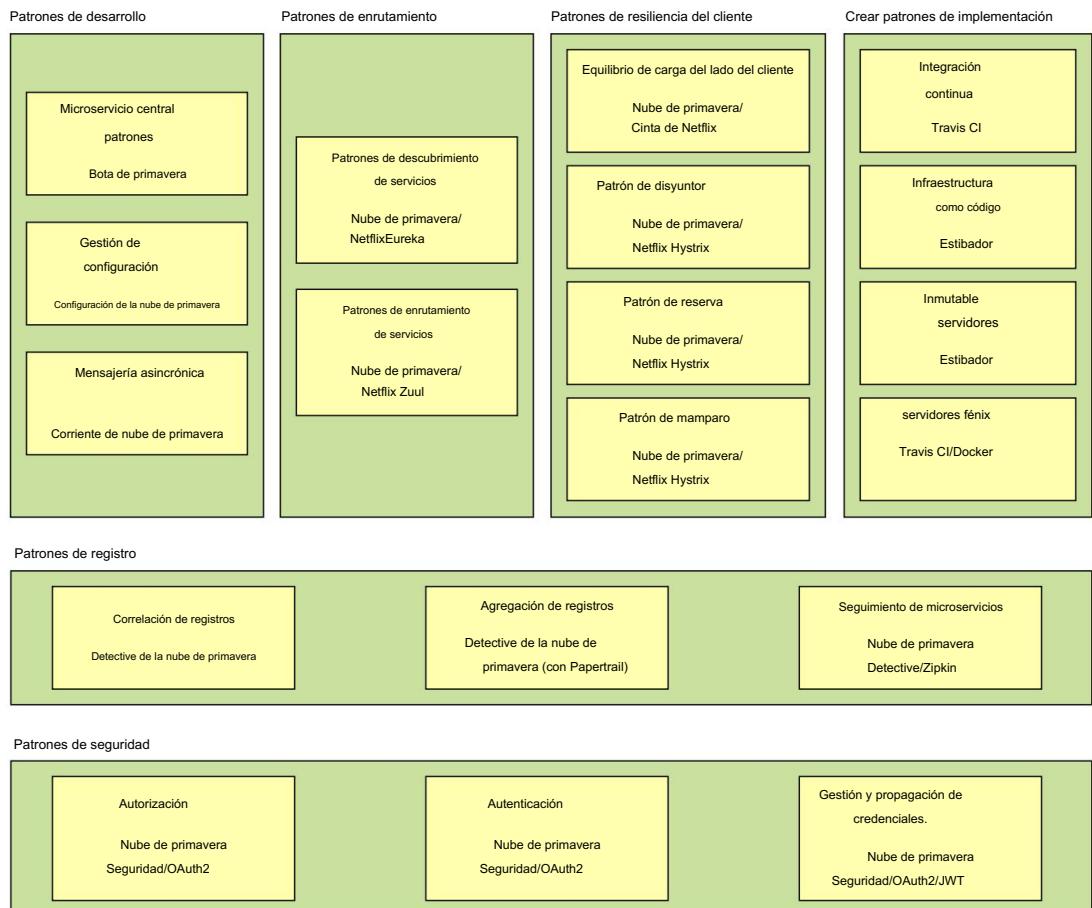


Figura 1.14 Puede asignar las tecnologías que va a utilizar directamente a los patrones de microservicios que hemos explorado hasta ahora en este capítulo.

### 1.10.1 Arranque de resorte

Spring Boot es la tecnología central utilizada en nuestra implementación de microservicios. Spring Boot simplifica enormemente el desarrollo de microservicios al simplificar las tareas principales de creación de microservicios basados en REST. Spring Boot también simplifica enormemente el mapeo de verbos de estilo HTTP (GET, PUT, POST y DELETE) a URL y la serialización del protocolo JSON hacia y desde objetos Java, así como el mapeo de excepciones de Java a códigos de error HTTP estándar .

### 1.10.2 Configuración de la nube de primavera

Spring Cloud Config maneja la administración de los datos de configuración de la aplicación a través de un servicio centralizado, de modo que los datos de configuración de su aplicación (particularmente los datos de configuración específicos de su entorno) estén claramente separados de su microservicio implementado. Esto garantiza que no importa cuántas instancias de microservicio abra, siempre tendrán la misma configuración. Spring Cloud Config tiene su propio repositorio de administración de propiedades, pero también se integra con proyectos de código abierto como los siguientes:

Git—Git (<https://git-scm.com/>) es un sistema de control de versiones de código abierto que le permite administrar y rastrear cambios en cualquier tipo de archivo de texto. Spring Cloud Config puede integrarse con un repositorio respaldado por Git y leer los datos de configuración de la aplicación fuera del repositorio.

Cónsul—Cónsul (<https://www.consul.io/>) es una herramienta de descubrimiento de servicios de código abierto que permite que las instancias de servicio se registren en el servicio. Luego, los clientes del servicio pueden preguntarle a Consul dónde se encuentran las instancias del servicio. Cónsul también incluye una base de datos basada en un almacén de valores clave que Spring Cloud Config puede utilizar para almacenar datos de configuración de la aplicación.

Eureka—Eureka (<https://github.com/Netflix/eureka>) es un proyecto de Netflix de código abierto que, al igual que Consul, ofrece capacidades de descubrimiento de servicios similares. Eureka también tiene una base de datos de valores clave que se puede usar con Spring Cloud Config.

### 1.10.3 Descubrimiento del servicio Spring Cloud

Con el descubrimiento de servicios Spring Cloud, puede abstraer la ubicación física (IP y/o nombre del servidor) de donde se implementan sus servidores de los clientes que consumen el servicio. Los consumidores de servicios invocan la lógica empresarial para los servidores a través de un nombre lógico en lugar de una ubicación física. El descubrimiento de servicios Spring Cloud también maneja el registro y la cancelación del registro de instancias de servicios a medida que se inician y cierran. El descubrimiento del servicio Spring Cloud se puede implementar utilizando Consul (<https://www.consul.io/>) y Eureka (<https://github.com/Netflix/eureka>) como su motor de descubrimiento de servicios.

#### 1.10.4 Spring Cloud/Netflix Hystrix y Ribbon

Spring Cloud se integra en gran medida con los proyectos de código abierto de Netflix. Para patrones de resiliencia de clientes de microservicios, Spring Cloud incluye las bibliotecas de Netflix Hystrix (<https://github.com/Netflix/Hystrix>) y proyecto Ribbon (<https://github.com/Netflix/Ribbon>) y hace que su uso desde dentro de sus propios microservicios sea trivial de implementar.

Con las bibliotecas de Netflix Hystrix, puede implementar rápidamente patrones de resiliencia del cliente de servicio, como los patrones de disyuntor y mamparo.

Si bien el proyecto Netflix Ribbon simplifica la integración con agentes de descubrimiento de servicios como Eureka, también proporciona equilibrio de carga del lado del cliente de las llamadas de servicio de un consumidor de servicios. Esto hace posible que un cliente continúe realizando llamadas de servicio incluso si el agente de descubrimiento de servicios no está disponible temporalmente.

#### 1.10.5 Nube de primavera/Netflix Zuul

Spring Cloud utiliza el proyecto Netflix Zuul (<https://github.com/Netflix/zuul>) para proporcionar capacidades de enrutamiento de servicios para su aplicación de microservicio. Zuul es una puerta de enlace de servicios que representa las solicitudes de servicios y garantiza que todas las llamadas a sus microservicios pasen por una única "puerta de entrada" antes de que se invoque el servicio de destino. Con esta centralización de llamadas de servicio, puede aplicar políticas de servicio estándar, como autenticación de autorización de seguridad, filtrado de contenido y reglas de enrutamiento.

#### 1.10.6 Corriente de nube de primavera

Corriente de la nube de primavera (<https://cloud.spring.io/spring-cloud-stream>) es una tecnología habilitadora que le permite integrar fácilmente el procesamiento de mensajes liviano en su microservicio. Con Spring Cloud Stream, puede crear microservicios inteligentes que pueden utilizar eventos asincrónicos a medida que ocurren en su aplicación. Con Spring Cloud Stream, puede integrar rápidamente sus microservicios con corredores de mensajes como RabbitMQ (<https://www.rabbitmq.com/>) y Kafka (<http://kafka.apache.org/>).

#### 1.10.7 Detective de la nube de primavera

Detective de la nube de primavera (<https://cloud.spring.io/spring-cloud-sleuth>) le permite integrar identificadores de seguimiento únicos en las llamadas HTTP y canales de mensajes (RabbitMQ , Apache Kafka) que se utilizan dentro de su aplicación. Estos números de seguimiento, a veces denominados identificadores de correlación o de seguimiento, le permiten realizar un seguimiento de una transacción a medida que fluye a través de los diferentes servicios de su aplicación. Con Spring Cloud Sleuth, estos ID de seguimiento se agregan automáticamente a cualquier declaración de registro que realice en su microservicio.

La verdadera belleza de Spring Cloud Sleuth se ve cuando se combina con herramientas tecnológicas de agregación de registros como Papertrail (<http://papertrailapp.com>), y herramientas de seguimiento como Zipkin (<http://zipkin.io>). Papertail es una plataforma de registro basada en la nube que se utiliza para agregar registros en tiempo real de diferentes microservicios en uno consultable.

base de datos. Open Zipkin toma datos producidos por Spring Cloud Sleuth y le permite visualice el flujo de sus llamadas de servicio involucradas para una sola transacción.

#### 1.10.8 Seguridad en la nube de Spring

Seguridad en la nube de Spring (<https://cloud.spring.io/spring-cloud-security/>) es un marco de autenticación y autorización que puede controlar quién puede acceder a sus servicios y qué pueden hacer con sus servicios. Spring Cloud Security está basado en tokens y permite servicios para comunicarse entre sí a través de un token emitido por un servidor de autenticación. Cada servicio que recibe una llamada puede verificar el token proporcionado en HTTP llamada para validar la identidad del usuario y sus derechos de acceso al servicio.

Además, Spring Cloud Security admite el token web JavaScript (<https://jwt.io>). El marco JavaScript Web Token (JWT) estandariza el formato de cómo un Se crea el token OAuth2 y proporciona estándares para firmar digitalmente un token creado.

#### 1.10.9 ¿Qué pasa con el aprovisionamiento?

Para las implementaciones de aprovisionamiento, haremos un cambio tecnológico. El Los marcos Spring están orientados al desarrollo de aplicaciones. Los frameworks Spring (incluido Spring Cloud) no tienen herramientas para crear una "construcción e implementación". tubería. Para implementar un proceso de "construcción e implementación", utilizará las siguientes herramientas: Travis CI (<https://travis-ci.org>) para su herramienta de compilación y Docker (<https://www.docker.com>) para construir la imagen final del servidor que contiene su microservicio.

Para implementar sus contenedores Docker creados, finalizamos el libro con un ejemplo de cómo para implementar toda la pila de aplicaciones creada a lo largo de este libro en la nube de Amazon.

#### 1.11 Spring Cloud con el ejemplo En la última

sección, analizamos todas las diferentes tecnologías de Spring Cloud que que utilizará a medida que desarrolle sus microservicios. Debido a que cada una de estas tecnologías son servicios independientes, obviamente tomará más de un capítulo. para explicarlos todos en detalle. Sin embargo, al concluir este capítulo, quiero dejarles con un pequeño ejemplo de código que demuestra nuevamente lo fácil que es integrar estos tecnologías en su propio esfuerzo de desarrollo de microservicios.

A diferencia del primer ejemplo de código del listado 1.1, no puede ejecutar este ejemplo de código. porque es necesario instalar y configurar una serie de servicios de soporte para su uso. Pero no te preocupes; los costos de instalación para estos servicios Spring Cloud (configuración servicio, descubrimiento de servicio) son un costo único en términos de configuración del servicio. Una vez están configurados, sus microservicios individuales pueden usar estas capacidades una y otra vez de nuevo. No pudimos incluir toda esa bondad en un solo ejemplo de código al comienzo de el libro.

El código que se muestra en la siguiente lista demuestra rápidamente cómo el descubrimiento de servicios, el disyuntor, el mamparo y el equilibrio de carga del lado del cliente de los servicios remotos se integraron en nuestro ejemplo "Hola mundo".

### Listado 1.2 Servicio Hello World usando Spring Cloud

```

paquete com.thinkmechanix.simpleservice;

//Se eliminaron otras importaciones para mayor concisión
importar com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
importar com.netflix.hystrix.contrib.javanica.annotation.HystrixProperty;
importar org.springframework.cloud.netflix.eureka.EnableEurekaClient;
importar org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;

@SpringBootAplicación
@RestController
@RequestMapping(valor="hola")
@EnableCircuitBreaker
@EnableEurekaClient
Aplicación de clase pública {

    public static void main(String[] args)
    {
        SpringApplication.run(Application.class, args);
    }

    @HystrixCommand(threadPoolKey = "helloThreadPool") public String
    helloRemoteServiceCall(String firstName,
                          Cadena apellido){}

    ResponseEntity<Cadena> restExchange =
        restoTemplate.exchange(
            "http://id-servicio-lógico/nombre/ [ca]{nombre}/
            {apellido}", HttpMethod.GET,
            nulo, String.class, nombre, apellido);

    devolver restExchange.getBody();

}

@RequestMapping(value="/{firstName}/{lastName}",
    método = RequestMethod.GET)
cadena pública hola( @PathVariable("firstName") String firstName,
                     @PathVariable("apellido") String apellido) {
    devolver holaRemoteServiceCall(nombre, apellido)
}
}

```

Permite que el servicio utilice el  
Bibliotecas Hystrix y Ribbon

Le dice al servicio que debería  
registrarse con un agente de  
descubrimiento de servicios de Eureka y  
que las llamadas de servicio deben  
utilizar el descubrimiento de  
servicios para "buscar" la ubicación de servicios remotos

Cadena apellido()

Llamadas de envoltorios al  
holaRemoteServiceCall  
método con un Hystrix  
cortacircuitos

Utiliza una clase RestTemplate  
decorada para tomar un servicio "lógico"  
ID y Eureka bajo las sábanas para  
buscar la ubicación física  
del servicio

Este código contiene muchas cosas, así que repasémoslo. Tenga en cuenta que esta lista es sólo un ejemplo y no se encuentra en la fuente del repositorio de GitHub del capítulo 1. código. Lo he incluido aquí para darle una idea de lo que vendrá más adelante en el libro.

Lo primero que debe notar es `@EnableCircuitBreaker` y  
Anotaciones `@EnableEurekaClient`. La anotación `@EnableCircuitBreaker`

le dice a su microservicio Spring que va a utilizar las bibliotecas de Netflix Hystrix en su aplicación. La anotación `@EnableEurekaClient` le indica a su microservicio que

registrarse con un agente de Eureka Service Discovery y que va a utilizar descubrimiento de servicios para buscar puntos finales de servicios REST remotos en su código. Tenga en cuenta que La configuración se realiza en un archivo de propiedades que le indicará al servicio simple la ubicación y el número de puerto de un servidor Eureka con el que contactar. Primero ves que se usa Hystrix. cuando declaras tu método hola:

```
@HystrixCommand(threadPoolKey = "holaThreadPool")  
cadena pública helloRemoteServiceCall (cadena nombre, cadena apellido)
```

La anotación @HystrixCommand hace dos cosas. Primero, cada vez que se llama al método helloRemoteServiceCall , no se invocará directamente. En cambio, el El método se delegará a un grupo de subprocessos administrado por Hystrix. Si la llamada toma demasiado larga (el valor predeterminado es un segundo), Hystrix interviene e interrumpe la llamada. Esta es la implementación del patrón de disyuntor. Lo segundo que hace esta anotación es crear un grupo de subprocessos llamado helloThreadPool administrado por Hystrix. todas las llamadas a El método helloRemoteServiceCall solo ocurrirá en este grupo de subprocessos y será aislado de cualquier otra llamada de servicio remoto que se esté realizando.

Lo último que hay que tener en cuenta es lo que ocurre dentro de helloRemoteServiceCall . método. La presencia de @EnableEurekaClient le ha dicho a Spring Boot que vas a utilizar una clase RestTemplate modificada (no es así como se utiliza el Estándar Spring RestTemplate funcionaría de inmediato) siempre que realice un servicio REST llamar. Esta clase RestTemplate le permitirá pasar un ID de servicio lógico para el servicio que está intentando invocar:

```
ResponseEntity<Cadena> restExchange = restTemplate.exchange  
(http://id-servicio-lógico/nombre/{nombre}/{apellido})
```

Deabajo de las sábanas, la clase RestTemplate se pondrá en contacto con el servicio Eureka y buscará hasta la ubicación física de una o más de las instancias de servicio de "nombre". Como consumidor del servicio, su código nunca tiene que saber dónde se encuentra ese servicio.

Además, la clase RestTemplate utiliza la biblioteca Ribbon de Netflix. La cinta se recuperará una lista de todos los puntos finales físicos asociados con un servicio. Cada vez que el servicio es llamado por el cliente, realiza un “round-robin” de la llamada a las diferentes instancias de servicio en el cliente sin tener que pasar por un equilibrador de carga centralizado. Al eliminar un balanceador de carga centralizado y trasladarlo al cliente, se elimina otro punto de falla. (equilibrador de carga cayendo) en la infraestructura de su aplicación.

Espero que en este punto esté impresionado, porque ha agregado una cantidad significativa de capacidades a su microservicio con solo unas pocas anotaciones. eso es lo real belleza detrás de Spring Cloud. Usted, como desarrollador, puede aprovechar las capacidades de microservicios de las principales empresas de nube como Netflix y Con-sul. Estas capacidades, si se usan fuera de Spring Cloud, pueden ser complejas y obtusas para configuración. Spring Cloud simplifica su uso literalmente a nada más que unos simples Anotaciones de Spring Cloud y entradas de configuración.

## 1.12 Asegurarnos de que nuestros ejemplos sean relevantes

Quiero asegurarme de que este libro proporcione ejemplos con los que puedas identificarte a medida que avanzas tu trabajo del día a día. Con este fin, he estructurado los capítulos de este libro y los ejemplos de código correspondientes en torno a las aventuras (desventuras) de una empresa ficticia llamada ThoughtMechanix.

ThoughtMechanix es una empresa de desarrollo de software cuyo producto principal, Eagle-Eye, proporciona una aplicación de gestión de activos de software de nivel empresarial. Proporciona cobertura para todos los elementos críticos: inventario, entrega de software, gestión de licencias, cumplimiento, costos y gestión de recursos. Su objetivo principal es permitir a las organizaciones obtener una imagen precisa de un momento dado de sus activos de software.

La empresa tiene aproximadamente 10 años. Si bien han experimentado un sólido crecimiento de los ingresos, internamente están debatiendo si deberían cambiar la plataforma de sus producto principal desde una aplicación monolítica local o trasladar su aplicación a la nube. El cambio de plataforma involucrado con EagleEye puede ser un "hacer o momento de ruptura" para una empresa.

La empresa está estudiando la posibilidad de reconstruir su producto principal EagleEye con una nueva arquitectura. Si bien gran parte de la lógica empresarial de la aplicación seguirá vigente, la aplicación en sí se dividirá de un diseño monolítico a uno mucho más pequeño. Diseño de microservicios cuyas piezas se pueden desplegar de forma independiente en la nube. El Los ejemplos de este libro no construirán la aplicación ThoughtMechanix completa. En cambio creará microservicios específicos a partir del dominio del problema en cuestión y luego creará la infraestructura que respaldará estos servicios utilizando varios Spring Cloud (y algunas tecnologías que no son Spring-Cloud).

La capacidad de adoptar con éxito una arquitectura de microservicios basada en la nube afectará todas las partes de una organización técnica. Esto incluye los equipos de arquitectura, ingeniería, pruebas y operaciones. Se necesitarán aportaciones de cada grupo y, al final, probablemente necesitarán una reorganización a medida que el equipo reevalúe sus responsabilidades en este nuevo entorno. Comencemos nuestro viaje con ThoughtMechanix mientras usted Comience el trabajo fundamental de identificar y desarrollar varios de los microservicios utilizados en EagleEye y luego construir estos servicios utilizando Spring Boot.

## 1.13 Resumen

Los microservicios son piezas de funcionalidad extremadamente pequeñas que son responsables para un área específica de alcance.

No existen estándares industriales para microservicios. A diferencia de otros servicios web anteriores protocolos, los microservicios adoptan un enfoque basado en principios y se alinean con la conceptos de REST y JSON.

Escribir microservicios es fácil, pero ponerlos en pleno funcionamiento para la producción requiere una previsión adicional. Introdujimos varias categorías de patrones de desarrollo de microservicios, incluido el desarrollo central, los patrones de enrutamiento, la resiliencia del cliente, la seguridad, el registro y los patrones de construcción/implementación.

Si bien los microservicios son independientes del idioma, presentamos dos marcos Spring que ayudan significativamente en la creación de microservicios: Spring Boot y Spring Cloud.

Spring Boot se utiliza para simplificar la creación de microservicios basados en REST/JSON . Su objetivo es permitirle crear microservicios rápidamente con nada más que unas pocas anotaciones.

Spring Cloud es una colección de tecnologías de código abierto de empresas como Netflix y HashiCorp que han sido "envueltas" con anotaciones Spring para simplificar significativamente la instalación y configuración de estos servicios.

# Construyendo microservicios con bota de resorte

## Este capítulo cubre

Aprender las características clave de un microservicio

Comprender cómo encajan los microservicios en una nube arquitectura

Descomponer un dominio empresarial en un conjunto de microservicios

Implementación de un microservicio simple usando Spring Boot

Comprender las perspectivas para crear aplicaciones basadas en microservicios.

Aprender cuándo no utilizar microservicios

La historia del desarrollo de software está plagada de historias de grandes desarrollos. proyectos que tras una inversión de millones de dólares y cientos de miles de horas de desarrollador de software y con muchas de las mejores y más brillantes mentes del mundo. industria trabajando en ellos, de alguna manera nunca logró entregar nada de valor a sus clientes y literalmente colapsaron bajo su propia complejidad y peso.

Estos gigantescos proyectos tendían a seguir grandes metodologías tradicionales de desarrollo en cascada que insistían en que todos los requisitos de la aplicación y El diseño se definiría al inicio del proyecto. Se puso tanto énfasis en

obtener todas las especificaciones del software “correctas” que había poco margen de maniobra para cumplir con nuevos requisitos comerciales, o refactorizar y aprender de los errores cometidos en el primeras etapas de desarrollo.

Sin embargo, la realidad es que el desarrollo de software no es un proceso lineal de definición. y ejecución, sino más bien evolutivo en el que se necesitan varias iteraciones de comunicación, aprendizaje y entrega al cliente antes del desarrollo.

El equipo realmente comprende el problema en cuestión.

Para agravar los desafíos del uso de metodologías tradicionales en cascada, es que muchas veces la granularidad de los artefactos de software que se entregan en estos proyectos es

estrechamente acoplado: la invocación de la lógica empresarial se produce en el nivel del lenguaje de programación en lugar de a través de protocolos neutrales en cuanto a la implementación, como JABÓN y DESCANSO. Esto aumenta en gran medida la posibilidad de que incluso un pequeño cambio en un componente de la aplicación puede dañar otras partes de la aplicación e introducir nuevos errores.

**Leaky:** la mayoría de las aplicaciones de software grandes administran diferentes tipos de datos. Para Por ejemplo, una aplicación de gestión de relaciones con el cliente (CRM) podría gestionar información de clientes, ventas y productos. En un modelo tradicional, estos datos se mantiene en el mismo modelo de datos y dentro del mismo almacén de datos. A pesar de existen límites obvios entre los datos, con demasiada frecuencia es tentador equipo de un dominio para acceder directamente a los datos que pertenecen a otro equipo.

Este fácil acceso a los datos crea dependencias ocultas y permite que los detalles de implementación de las estructuras de datos internas de un componente se filtre a través del sistema. toda la aplicación. Incluso pequeños cambios en una sola tabla de base de datos pueden requerir una número significativo de cambios de código y pruebas de regresión a lo largo del toda la aplicación.

**Monolítico:** porque la mayoría de los componentes de una aplicación tradicional residen en una única base de código que se comparte entre varios equipos, en cualquier momento.

Se realiza un cambio en el código, se debe volver a compilar toda la aplicación. volver a ejecutar un ciclo de prueba completo y volver a implementarlo. Incluso pequeños cambios en la base del código de la aplicación, ya sean requisitos de nuevos clientes o errores correcciones, se vuelven costosas y requieren mucho tiempo, y los grandes cambios se vuelven casi imposible de hacer en el momento oportuno.

Una arquitectura basada en microservicios adopta un enfoque diferente para ofrecer funcionalidad. En concreto, las arquitecturas basadas en microservicios tienen estas características:

**Restringido:** los microservicios tienen un conjunto único de responsabilidades y son limitados. en alcance. Los microservicios adoptan la filosofía UNIX de que una aplicación es nada más que una colección de servicios donde cada servicio hace una cosa y hace eso muy bien. Ligeramente

**acoplado:** una aplicación basada en microservicios es una colección de pequeños servicios que solo interactúan entre sí a través de una interfaz no específica de implementación que utiliza un protocolo de invocación no propietario (por ejemplo, HTTP)

y descansar). Mientras la interfaz del servicio no cambie, los propietarios del microservicio tienen más libertad para realizar modificaciones en el servicio.

que en una arquitectura de aplicación tradicional.

Resumido: los microservicios poseen completamente sus estructuras y fuentes de datos. Los datos propiedad de un microservicio solo pueden ser modificados por ese servicio.

El control de acceso a la base de datos que contiene los datos del microservicio se puede bloquear para permitir que solo el servicio acceda a ella.

Independiente: cada microservicio en una aplicación de microservicio se puede compilar e implementar independientemente de los demás servicios utilizados en la aplicación. Esto significa que los cambios se pueden aislar y probar mucho más fácilmente que con una aplicación monolítica más interdependiente.

¿Por qué estos atributos de la arquitectura de microservicios son importantes para el desarrollo basado en la nube? Las aplicaciones basadas en la nube en general tienen lo siguiente:

Una base de usuarios grande y diversa : diferentes clientes quieren funciones diferentes y no quieren tener que esperar un largo ciclo de lanzamiento de aplicaciones antes de poder comenzar a utilizarlas. Los microservicios permiten que las funciones se entreguen rápidamente, porque cada servicio tiene un alcance pequeño y se accede a él a través de una interfaz bien definida.

Requisitos de tiempo de actividad extremadamente altos: debido a la naturaleza descentralizada de los microservicios, las aplicaciones basadas en microservicios pueden aislar más fácilmente fallas y problemas en partes específicas de una aplicación sin tener que desactivar toda la aplicación. Esto reduce el tiempo de inactividad general de las aplicaciones y las hace más resistentes a los problemas.

Requisitos de volumen desigual: las aplicaciones tradicionales implementadas dentro de las cuatro paredes de un centro de datos corporativo generalmente tienen patrones de uso consistentes que surgen con el tiempo. Esto simplifica la planificación de la capacidad para este tipo de aplicaciones. Pero en una aplicación basada en la nube, un simple tweet en Twitter o una publicación en Slashdot puede disparar la demanda de una aplicación basada en la nube.

Debido a que las aplicaciones de microservicio se dividen en pequeños componentes que se pueden implementar de forma independiente entre sí, es mucho más fácil centrarse en los componentes que están bajo carga y escalar esos componentes horizontalmente en múltiples servidores en una nube.

Este capítulo le proporciona la base que necesita para apuntar e identificar microservicios en su problema empresarial, construir el esqueleto de un microservicio y luego comprender los atributos operativos que deben existir para que un microservicio se implemente y administre exitosamente en producción.

Para diseñar y crear microservicios con éxito, debe abordar los microservicios como si fuera un detective de policía que entrevista a testigos de un crimen. Aunque todos los testigos presenciaron los mismos hechos, su interpretación del crimen está determinada por sus antecedentes, lo que era importante para ellos (por ejemplo, lo que los motiva) y las presiones ambientales que se ejercieron en ese momento.

fueron testigos del suceso. Cada participante tiene sus propias perspectivas (y prejuicios) sobre lo que consideran importante.

Al igual que un detective de policía exitoso que intenta llegar a la verdad, el viaje para construir una arquitectura de microservicio exitosa implica incorporar las perspectivas de varias personas dentro de su organización de desarrollo de software. Aunque se necesita más que personal técnico para entregar una aplicación completa, creo que la base para el desarrollo exitoso de microservicios comienza con las perspectivas de tres roles críticos: El arquitecto: el trabajo del arquitecto es ver el panorama general y comprender cómo una aplicación puede funcionar. Descomponerse en microservicios individuales y cómo los microservicios interactuarán para ofrecer una solución.

El desarrollador de software: el desarrollador de software escribe el código y comprende en detalle cómo se utilizarán el lenguaje y los marcos de desarrollo del lenguaje para brindar un microservicio.

El ingeniero de DevOps: el ingeniero de DevOps aporta inteligencia a cómo se implementan y administran los servicios no solo en los entornos de producción, sino también en todos los entornos que no son de producción. Las consignas para el ingeniero DevOps son coherencia y repetibilidad en todos los entornos.

En este capítulo, demostraré cómo diseñar y construir un conjunto de microservicios desde la perspectiva de cada uno de estos roles usando Spring Boot y Java. Cuando concluya el capítulo, tendrá un servicio que se puede empaquetar e implementar en la nube.

## 2.1 La historia del arquitecto: diseño de la arquitectura de microservicios

El papel de un arquitecto en un proyecto de software es proporcionar un modelo funcional del problema que debe resolverse. El trabajo del arquitecto es proporcionar el andamiaje sobre el cual los desarrolladores construirán su código para que todas las piezas de la aplicación encajen.

Al crear una arquitectura de microservicios, el arquitecto de un proyecto se centra en tres tareas clave:

- 1 Descomponiendo el problema empresarial
- 2 Establecer la granularidad del servicio
- 3 Definición de las interfaces de servicio

### 2.1.1 Descomponiendo el problema empresarial

Ante la complejidad, la mayoría de las personas intentan dividir el problema en el que están trabajando en partes manejables. Lo hacen para no tener que intentar encajar todos los detalles del problema en sus cabezas. En lugar de ello, descomponen el problema de manera abstracta en unas pocas partes clave y luego buscan las relaciones que existen entre estas partes.

En una arquitectura de microservicios, el arquitecto divide el problema empresarial en partes que representan dominios de actividad discretos. Estos fragmentos encapsulan las reglas comerciales y la lógica de datos asociada con una parte particular del dominio comercial.

Aunque desee que los microservicios encapsulen todas las reglas comerciales para llevar realizar una sola transacción, esto no siempre es factible. A menudo tendrás situaciones en las que necesita tener grupos de microservicios trabajando en diferentes partes del dominio empresarial para completar una transacción completa. Un arquitecto desmonta el servicio límites de un conjunto de microservicios observando dónde no llega el dominio de datos parecen encajar.

Por ejemplo, un arquitecto podría analizar un flujo de negocios que debe llevar a cabo código y se dan cuenta de que necesitan información tanto del cliente como del producto. La presencia de dos dominios de datos discretos es una buena indicación de que existen múltiples microservicios. jugando. Cómo interactúan habitualmente las dos partes diferentes de la transacción comercial se convierte en la interfaz de servicio para los microservicios.

Dividir un ámbito empresarial es una forma de arte más que una ciencia en blanco y negro. Utilice las siguientes pautas para identificar y descomponer un problema empresarial en candidatos a microservicio:

**1** Describe el problema empresarial y escucha los sustantivos que estás utilizando para describir el problema.

Usar los mismos sustantivos una y otra vez para describir el problema suele ser una indicación de un dominio empresarial principal y una oportunidad para un microservicio. Ejemplos de sustantivos objetivo para el dominio EagleEye del capítulo 1 podrían verse algo así como contratos, licencias y activos.

**2** Presta atención a los verbos. Los verbos resaltan acciones y a menudo representan lo natural.

contornos de un dominio problemático. Si se encuentra diciendo "la transacción X necesita para obtener datos de la cosa A y la cosa B", eso generalmente indica que hay múltiples servicios en juego. Si aplica a EagleEye el enfoque de observar los verbos, puede buscar declaraciones como: "Cuando Mike, de los servicios de escritorio, Al configurar una nueva PC, busca el número de licencias disponibles para el software X y, si hay licencias disponibles, instala el software. Luego actualiza el número. de licencias utilizadas en su hoja de cálculo de seguimiento". Los verbos clave aquí son miradas y actualizaciones.

**3** Busque la cohesión de los datos. A medida que divides tu problema de negocio en partes discretas piezas, busque piezas de datos que estén altamente relacionadas entre sí. Si de repente, durante el curso de tu conversación, estás leyendo o actualizando datos eso es radicalmente diferente de lo que has estado discutiendo hasta ahora, potencialmente tener otro candidato de servicio. Los microservicios deberían poseer completamente sus datos.

Tomemos estas pautas y aplíquemolas a un problema del mundo real. El Capítulo 1 presentó un producto de software existente llamado EagleEye que se utiliza para administrar software. activos como licencias de software y certificados de capa de conexión segura (SSL) . Estos artículos se implementan en varios servidores de una organización.

EagleEye es una aplicación web monolítica tradicional que se implementa en un J2EE Servidor de aplicaciones que reside dentro del centro de datos de un cliente. Tu objetivo es separarte la aplicación monolítica existente en un conjunto de servicios.

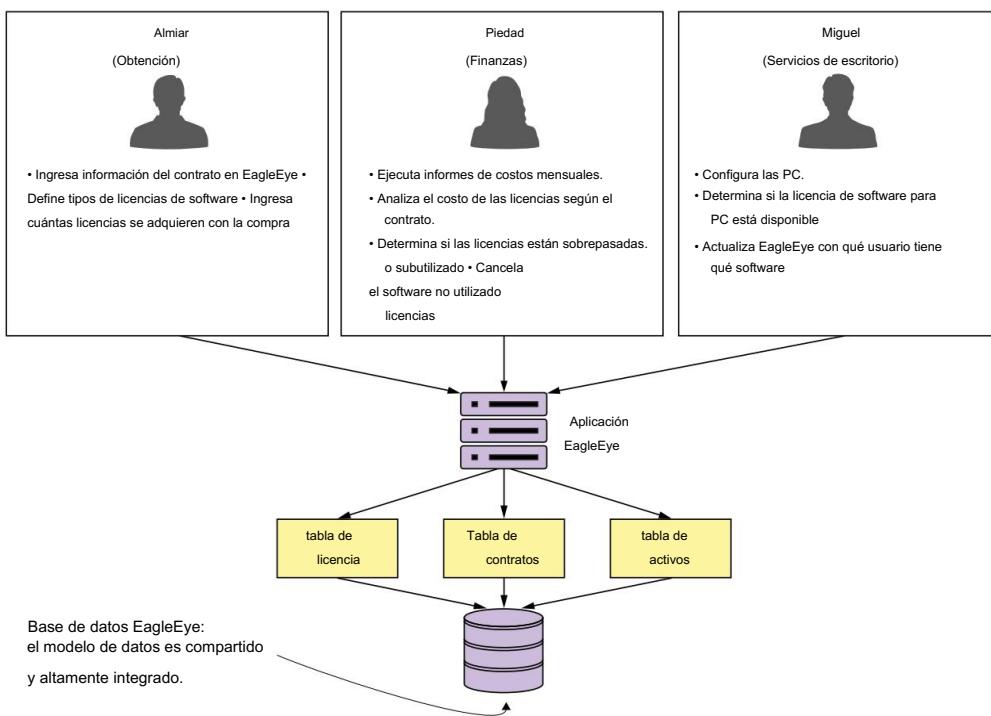


Figura 2.1 Entreviste a los usuarios de EagleEye y comprenda cómo realizan su trabajo diario.

Comenzará entrevistando a todos los usuarios de la aplicación EagleEye y discutiendo con ellos cómo interactúan y usan EagleEye. La Figura 2.1 captura un resumen de las conversaciones que podría tener con los diferentes clientes comerciales. Al observar cómo interactúan los usuarios de EagleEye con la aplicación y cómo se desglosa el modelo de datos de la aplicación, puede descomponer el dominio del problema de EagleEye en los siguientes microservicios candidatos.

En la figura, he resaltado una serie de sustantivos y verbos que surgieron durante las conversaciones con los usuarios comerciales. Como se trata de una aplicación existente, puede mirar la aplicación y asignar los sustantivos principales a tablas en el modelo de datos físicos. Una aplicación existente puede tener cientos de tablas, pero cada tabla normalmente se asignará a un único conjunto de entidades lógicas.

La Figura 2.2 muestra un modelo de datos simplificado basado en conversaciones con

Clientes de EagleEye. Basado en las entrevistas comerciales y los datos.

En este modelo, los candidatos a microservicios son servicios de organización, licencia, contrato y activos.

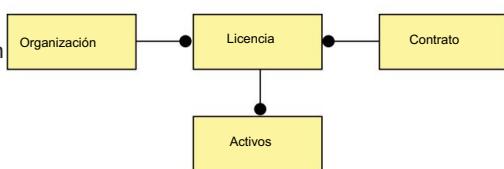


Figura 2.2 Un modelo de datos EagleEye simplificado

## 2.1.2 Establecer granularidad del servicio

Una vez que tenga un modelo de datos simplificado, puede comenzar el proceso de definir qué microservicios que vas a necesitar en la aplicación. Basado en el modelo de datos de la figura.

2.2, puede ver el potencial de cuatro microservicios basados en los siguientes elementos:

- Activos
- Licencia
- Contrato
- Organización

El objetivo es tomar estas piezas principales de funcionalidad y extraerlas en unidades completamente autónomas que puedan construirse e implementarse independientemente de cada una. otro. Pero extraer servicios del modelo de datos implica más que reempaquetar código en proyectos separados. También se trata de desentrañar las tablas de la base de datos reales que los servicios acceden y solo permiten que cada servicio individual acceda a las tablas en su dominio específico. La Figura 2.3 muestra cómo el código de la aplicación y el modelo de datos "fragmentados" en pedazos individuales.

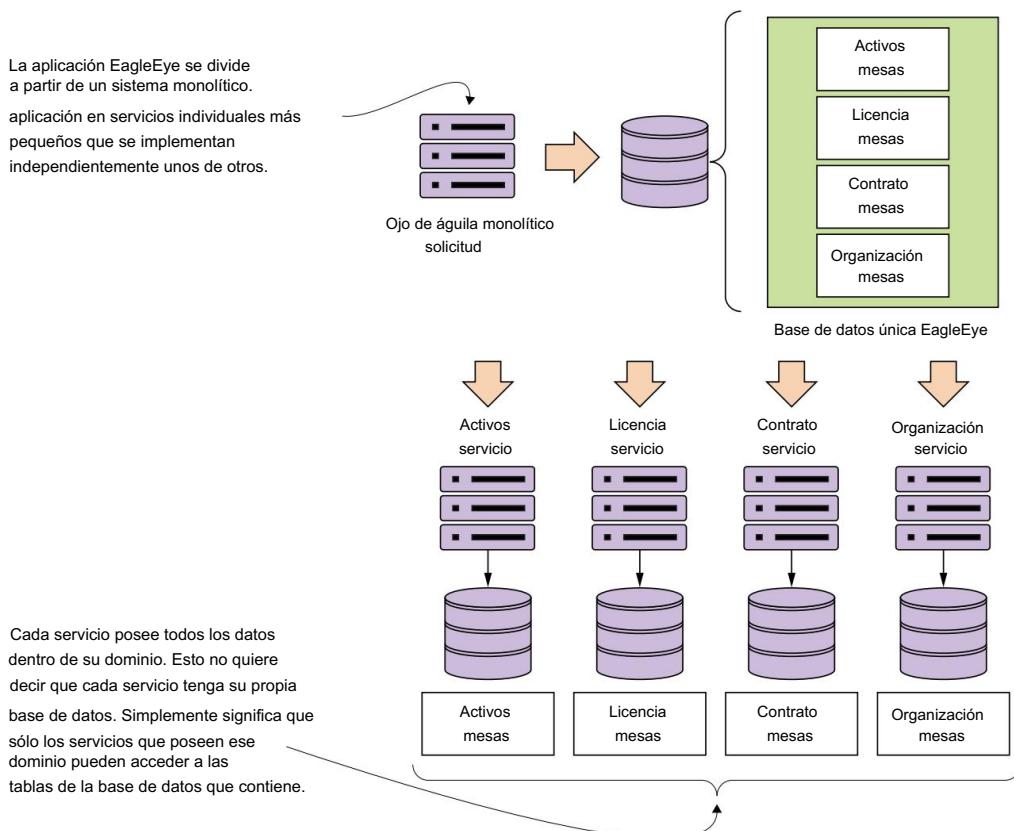


Figura 2.3 Se utiliza el modelo de datos como base para descomponer una aplicación monolítica en microservicios.

Una vez que haya dividido un dominio problemático en partes discretas, a menudo tendrá dificultades para determinar si ha alcanzado el nivel adecuado de granularidad para sus servicios. Un microservicio que sea demasiado grueso o fino tendrá una serie de atributos reveladores que analizaremos en breve.

Cuando crea una arquitectura de microservicio, la cuestión de la granularidad es importante, pero puede utilizar los siguientes conceptos para determinar la solución correcta:

- 1 Es mejor comenzar de manera amplia con su microservicio y refactorizarlo a servicios más pequeños. Es fácil exagerar cuando comienza su recorrido por el microservicio y convierte todo en un microservicio. Pero descomponer el dominio del problema en pequeños servicios a menudo conduce a una complejidad prematura porque los microservicios se convierten en nada más que servicios de datos detallados.
- 2 Concéntrese primero en cómo interactuarán sus servicios entre sí. Esto ayudará a establecer las interfaces generales de su dominio problemático. Es más fácil refactorizar de ser demasiado grueso a demasiado fino.
- 3 Las responsabilidades del servicio cambiarán con el tiempo a medida que crezca su comprensión del dominio del problema. A menudo, un microservicio adquiere responsabilidades a medida que se solicita nueva funcionalidad de la aplicación. Lo que comienza como un único microservicio puede convertirse en múltiples servicios, donde el microservicio original actúa como una capa de orquestación para estos nuevos servicios y encapsula su funcionalidad de otras partes de la aplicación.

### Los olores de un mal microservicio

¿ Cómo saber si sus microservicios tienen el tamaño correcto? Si un microservicio es demasiado generalizado, probablemente verá lo siguiente:

Un servicio con demasiadas responsabilidades: el flujo general de la lógica empresarial en el servicio es complicado y parece imponer una gama demasiado diversa de reglas empresariales.

El servicio gestiona datos en una gran cantidad de tablas: un microservicio es el sistema de registro de los datos que gestiona. Si se encuentra persistiendo datos en varias tablas o accediendo a tablas fuera de la base de datos inmediata, esto es una pista de que el servicio es demasiado grande. Me gusta utilizar la pauta de que un microservicio no debe poseer más de tres a cinco tablas. Si pasa más, es probable que su servicio tenga demasiada responsabilidad.

Demasiados casos de prueba: los servicios pueden crecer en tamaño y responsabilidad con el tiempo. Si tiene un servicio que comenzó con una pequeña cantidad de casos de prueba y termina con cientos de casos de prueba unitarios y de integración, es posible que necesite refactorizar.

¿Qué pasa con un microservicio que es demasiado detallado?

Los microservicios en una parte del dominio del problema se reproducen como conejos: si todo se convierte en un microservicio, componer la lógica de negocios a partir de los servicios se vuelve complejo y difícil porque la cantidad de servicios necesarios para realizar un trabajo crece enormemente. Un olor común es cuando tienes docenas de microservicios en una aplicación y cada servicio interactúa con una sola tabla de base de datos.

Sus microservicios son muy interdependientes entre sí: descubre que los microservicios en una parte del dominio problemático siguen llamándose entre sí para completar una única solicitud de usuario.

Sus microservicios se convierten en una colección de servicios CRUD (Crear, Reemplazar, Actualizar, Eliminar) simples: los microservicios son una expresión de lógica empresarial y no una capa de abstracción sobre sus fuentes de datos. Si sus microservicios no hacen nada más que lógica relacionada con CRUD, probablemente sean demasiado detallados.

Una arquitectura de microservicios debe desarrollarse con un proceso de pensamiento evolutivo en el que sepa que no obtendrá el diseño correcto la primera vez. Es por eso que es mejor comenzar con su primer conjunto de servicios más generales que detallados. También es importante no ser dogmático con su diseño. Es posible que se encuentre con limitaciones físicas en sus servicios donde necesitará crear un servicio de agregación que una los datos porque dos servicios separados serán demasiado comunicativos, o donde no existen límites claros entre las líneas de dominio de un servicio.

Al final, adopte un enfoque pragmático y cumpla, en lugar de perder el tiempo tratando de obtenga el diseño perfecto y luego no tenga nada que mostrar por su esfuerzo.

### 2.1.3 Hablar entre nosotros: interfaces de servicio

La última parte del aporte del arquitecto trata sobre definir cómo los microservicios en su aplicación se comunicarán entre sí. Al crear lógica empresarial con microservicios, las interfaces de los servicios deben ser intuitivas y los desarrolladores deben tener un ritmo de cómo funcionan todos los servicios en la aplicación aprendiendo uno o dos de los servicios de la aplicación.

En general, se pueden utilizar las siguientes pautas para pensar en el diseño de la interfaz de servicio:

- 1 Adopte la filosofía REST : el enfoque REST de los servicios consiste esencialmente en adoptar HTTP como protocolo de invocación para los servicios y el uso de verbos HTTP estándar (GET, PUT, POST y DELETE). Modele sus comportamientos básicos en torno a estos verbos HTTP .
- 2 Utilice URI para comunicar la intención: el URI que utilice como puntos finales para el servicio debe describir los diferentes recursos en su dominio de problema y proporcionar un mecanismo básico para las relaciones de recursos dentro de su dominio de problema.
- 3 Utilice JSON para sus solicitudes y respuestas: la notación de objetos JavaScript (en otras palabras, JSON) es un protocolo de serialización de datos extremadamente liviano y es mucho más fácil de consumir que XML.
- 4 Utilice códigos de estado HTTP para comunicar resultados: el protocolo HTTP tiene un amplio cuerpo de códigos de respuesta estándar para indicar el éxito o el fracaso de un servicio. Conozca estos códigos de estado y, lo más importante, úselos de manera consistente en todos sus servicios.

Todas las pautas básicas conducen a una cosa: hacer que sus interfaces de servicio sean fáciles de usar. comprender y consumir. Quiere que un desarrollador se siente y mire el servicio. interfaces y empezar a utilizarlas. Si un microservicio no es fácil de consumir, los desarrolladores lo harán hacer todo lo posible para evitar y subvertir la intención de la arquitectura.

## 2.2 Cuándo no utilizar microservicios

Hemos dedicado este capítulo a hablar de por qué los microservicios son una herramienta arquitectónica poderosa. patrón para aplicaciones de construcción. Pero no he mencionado cuándo no deberías usar microservicios para construir sus aplicaciones. Repasemos ellos:

- 1 Complejidad en la construcción de sistemas distribuidos.
- 2 Expansión de servidores/contenedores virtuales
- 3 tipo de aplicación
- 4 Transacciones de datos y coherencia

### 2.2.1 Complejidad de la construcción de sistemas distribuidos

Debido a que los microservicios son distribuidos y detallados (pequeños), introducen un nivel de complejidad en su aplicación que no estaría presente en aplicaciones más monolíticas. Las arquitecturas de microservicios requieren un alto grado de madurez operativa.

No considere utilizar microservicios a menos que su organización esté dispuesta a invertir en ellos. automatización y trabajo operativo (monitoreo, escalamiento) que un sistema altamente distribuido la aplicación debe tener éxito.

### 2.2.2 Expansión de servidores

Uno de los modelos de implementación más comunes para microservicios es tener una instancia de microservicio implementada en un servidor. En una aplicación grande basada en microservicios, es posible que termine con entre 50 y 100 servidores o contenedores (generalmente virtuales) que tienen que construirse y mantenerse únicamente en producción. Incluso con el menor coste de funcionamiento Estos servicios en la nube, la complejidad operativa de tener que administrar y monitorear estos servidores puede ser enorme.

**NOTA** La flexibilidad de los microservicios debe sopesarse con el costo de ejecutando todos estos servidores.

### 2.2.3 Tipo de aplicación

Los microservicios están orientados a la reutilización y son extremadamente útiles para construir aplicaciones de gran tamaño que deben ser altamente resilientes y escalables. Ésta es una de las razones por las que tantas empresas basadas en la nube han adoptado microservicios. Si está creando aplicaciones pequeñas a nivel departamental o aplicaciones con una base de usuarios pequeña, el Complejidad asociada con la construcción sobre un modelo distribuido como los microservicios. Podría ser más caro de lo que vale.

## 2.2.4 Transformaciones y coherencia de datos

Al comenzar a analizar los microservicios, debe pensar en los patrones de uso de datos de sus servicios y cómo los utilizarán los consumidores de servicios. Un microservicio envuelve y abstrae un pequeño número de tablas y funciona bien como

Mecanismo para realizar tareas "operativas", como crear, agregar y realizar consultas simples (no complejas) en una tienda.

Si sus aplicaciones necesitan realizar una agregación o transformación de datos complejos entre múltiples fuentes de datos, la naturaleza distribuida de los microservicios hará que esto funcione difícil. Sus microservicios invariablemente asumirán demasiada responsabilidad y pueden También se vuelven vulnerables a problemas de rendimiento.

También tenga en cuenta que no existe ningún estándar para realizar transacciones entre microservicios. Si necesita gestión de transacciones, deberá desarrollar esa lógica tú mismo. Además, como verá en el capítulo 7, los microservicios pueden comunicarse entre ellos mediante el uso de mensajes. La mensajería introduce latencia en las actualizaciones de datos. Sus aplicaciones deben manejar la coherencia eventual donde se aplican las actualizaciones a sus datos puede que no aparezca inmediatamente.

## 2.3 La historia del desarrollador: creación de un microservicio con Spring Boot y Java

Al crear un microservicio, pasar del espacio conceptual al espacio de implementación requiere un cambio de perspectiva. Específicamente, como desarrollador, debes establecer un patrón básico de cómo va cada uno de los microservicios en su aplicación A ser implementado. Si bien cada servicio será único, debes asegurarte de que estás usando un marco que elimina el código repetitivo y que cada pieza de su microservicio se presenta de la misma manera consistente.

En esta sección, exploraremos las prioridades del desarrollador al crear la licencia. microservicio de su modelo de dominio EagleEye. Su servicio de licencia será escrito usando Spring Boot. Spring Boot es una capa de abstracción sobre el estándar Bibliotecas Spring que permiten a los desarrolladores crear rápidamente sitios web basados en Groovy y Java aplicaciones y microservicios con mucha menos ceremonia y configuración que una aplicación Spring en toda regla.

Para su ejemplo de servicio de licencia, utilizará Java como lenguaje de programación principal y Apache Maven como herramienta de compilación.

En las siguientes secciones usted va a

- 1 Cree el esqueleto básico del microservicio y un script Maven para crear el solicitud
- 2 Implemente una clase de arranque Spring que iniciará el contenedor Spring para el microservicio e iniciar el inicio de cualquier trabajo de inicialización para la clase
- 3 Implemente una clase de controlador Spring Boot para mapear un punto final para exponer los puntos finales del servicio

### 2.3.1 Comenzando con el proyecto esqueleto

Para comenzar, creará un proyecto esqueleto para la licencia. Puede bajar el código fuente desde GitHub (<https://github.com/carnellijs/spmia-chapter2>) o cree un directorio de proyecto de servicio de licencias con la siguiente estructura de directorios:

```

servicio de
licencias    src/main/java/com/thinktmechanix/licenses
controladores
    modelo
    servicios
    recursos

```

Una vez que haya desplegado o creado esta estructura de directorios, comience escribiendo su script Maven para el proyecto. Este será el archivo pom.xml ubicado en la raíz del directorio del proyecto. La siguiente lista muestra el archivo Maven POM para su servicio de licencias.

Listado 2.1 Archivo pom de Maven para el servicio de licencias

```

<?xml versión="1.0" codificación="UTF-8"?> <proyecto
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<groupId>com.thinktmechanix</groupId> <artifactId>servicio
de licencias</artifactId> <versión>0.0.1-SNAPSHOT</versión>
<packaging>jar</packaging>

<nombre>Servicio de licencias EagleEye</nombre>
<descripción>Servicio de licencias</descripción>

<padre>
<groupId>org.springframework.boot</groupId> <artifactId>spring-boot-
starter-parent</artifactId> <versión>1.4.4.RELEASE</version> <relativePath/> </
parent> <dependencias> < dependencia>

    <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-
    starter-web</artifactId>
    </dependencia>
    <dependencia>
        <groupId>org.springframework.boot</groupId> <artifactId>actuador-
        arranque-spring-boot</artifactId>
        </dependencia> </
    dependencias>

```

Le dice a Maven que incluya el  
arrancador de arranque de primavera

Dependencias del kit



Le dice a Maven que incluya  
las dependencias  
web de Spring Boot



```
<!--Nota: Algunas propiedades de compilación y complementos de compilación de Docker se han
excluidos del pom.xml en este pom (no en el código fuente del repositorio de github) porque no son relevantes
para nuestra discusión aquí.
-->

<construir>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-
maven-plugin</artifactId> </plugin> </plugins> </build> </project>
```

Le dice a Maven que incluya complementos de
Maven específicos de Spring para construir e implementar
Aplicaciones de arranque de primavera

No repasaremos todo el guión en detalle, pero tomaremos nota de algunas áreas clave al comenzar. Spring Boot se divide en muchos proyectos individuales. La filosofía es que no deberías tener que "derribar el mundo" si no vas a utilizar diferentes piezas de Spring Boot en tu aplicación. Esto también permite que los distintos proyectos Spring Boot lancen nuevas versiones de código de forma independiente entre sí. Para ayudar a simplificar la vida de los desarrolladores, el equipo de Spring Boot ha reunido proyectos dependientes relacionados en varios kits "iniciales". En la parte 1 del POM de Maven , le dice a Maven que necesita desinstalar la versión 1.4.4 del marco Spring Boot. En las partes 2 y 3 del archivo Maven, identifica que está

desplegando el iniciador Spring Web y Spring Actuator. kits. Estos dos proyectos están en el corazón de casi cualquier servicio basado en Spring Boot REST. Descubrirá que a medida que incorpora más funciones en sus servicios, la lista de estos proyectos dependientes se vuelve más larga.

Además, Spring Source ha proporcionado complementos de Maven que simplifican la construcción y la implementación de las aplicaciones Spring Boot. El paso 4 le indica al script de compilación de Maven que instale el último complemento Spring Boot Maven. Este complemento contiene una serie de tareas complementarias (como spring-boot:run) que simplifican la interacción entre Maven y Spring Boot.

Finalmente, verá un comentario que indica que se han eliminado secciones del archivo Maven. Por el bien de los árboles, no incluyó los complementos de Spotify Docker en el listado 2.1.

**NOTA** Cada capítulo de este libro incluye archivos Docker para crear e implementar la aplicación como contenedores Docker. Puede encontrar detalles sobre cómo crear estas imágenes de Docker en el archivo README.md en las secciones de código de cada capítulo.

### 2.3.2 Arrancar su aplicación Spring Boot: escribir la clase Bootstrap

Su objetivo es poner en funcionamiento un microservicio simple en Spring Boot y luego iterarlo para ofrecer funcionalidad. Para ello, debe crear dos clases en el microservicio de su servicio de licencias:

Una clase Spring Bootstrap que Spring Boot utilizará para iniciar e inicializar la aplicación.

Una clase Spring

Controller que expondrá los puntos finales HTTP que pueden ser invocado en el microservicio

Como verá en breve, Spring Boot utiliza anotaciones para simplificar la instalación y configuración del servicio. Esto se hace evidente al observar la clase bootstrap en el siguiente listado. Esta clase de arranque está en `src/main/java/com/thinkmechanix/licenses/Application.java` archivo.

#### Listado 2.2 Presentación de la anotación `@SpringBootApplication`

```
paquete com.thinkmechanix.licenses;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
Aplicación de clase pública {
    público estático vacío principal (String [] argumentos) {
        SpringApplication.run(Application.clase, argumentos);
    }
}
```

Lo primero que hay que tener en cuenta en este código es el uso de `@SpringBootApplication` anotación. Spring Boot usa esta anotación para decirle al contenedor Spring que esto La clase es la fuente de definiciones de beans para usar en Spring. En una aplicación Spring Boot, puedes definir Spring Beans por

- 1 Anotar una clase Java con una anotación `@Component`, `@Service` o `@Repository`  
etiqueta de estación
- 2 Anotar una clase con una etiqueta `@Configuration` y luego definir un constructor  
método para cada Spring Bean que desee compilar con una etiqueta `@Bean`.

Debajo de las sábanas, la anotación `@SpringBootApplication` marca la Aplicación clase en el listado 2.2 como una clase de configuración, luego comienza a escanear automáticamente todas las clases en la ruta de clases de Java para otros Spring Beans.

Lo segundo a tener en cuenta es el método `main()` de la clase Application . En el método `main()` , `SpringApplication.run (Application.class, args)`,

La llamada inicia el contenedor Spring y devuelve un objeto Spring ApplicationContext . (No estás haciendo nada con ApplicationContext , por lo que no se muestra en el código.)

Lo más fácil de recordar sobre la anotación `@SpringBootApplication` y la clase de Application correspondiente es la clase de arranque para todo el microservicio. La lógica de inicialización central del servicio debe ubicarse en esta clase.

#### 2.3.3 Construyendo la puerta de entrada al microservicio:

el controlador de arranque de primavera

Ahora que ha eliminado el script de compilación e implementado un sencillo clase Spring Boot Bootstrap, puede comenzar a escribir su primer código que hará algo. Este código será su clase de Controlador . En una aplicación Spring Boot, un

La clase de controlador expone los puntos finales de los servicios y asigna los datos de una solicitud HTTP entrante a un método Java que procesará la solicitud.

### Dale un descanso

Todos los microservicios de este libro siguen el enfoque REST para crear sus servicios.

Una discusión en profundidad sobre REST está fuera del alcance de este libro, pero para sus propósitos, todos los servicios que cree tendrán las siguientes características:

Utilice HTTP como protocolo de invocación para el servicio: el servicio se expondrá a través de Punto final HTTP y utilizará el protocolo HTTP para transportar datos hacia y desde los servicios.

Asigne el comportamiento del servicio a verbos HTTP estándar: REST enfatiza tener Los servicios asignan su comportamiento a los verbos HTTP de los verbos POST, GET, PUT y DELETE. Estos verbos se asignan a las funciones CRUD que se encuentran en la mayoría de los servicios.

Utilice JSON como formato de serialización para todos los datos que van y vienen del servicio. no es un principio estricto para los microservicios basados en REST, pero JSON se ha convertido lengua franca para serializar datos que serán enviados y devueltos por un microservicio. Se puede utilizar XML, pero muchas aplicaciones basadas en REST hacen un uso intensivo de JavaScript y JSON (Notación de objetos JavaScript). JSON es el formato nativo para serializar y deserializar datos consumidos por interfaces web basadas en JavaScript y servicios.

Utilice códigos de estado HTTP para comunicar el estado de una llamada de servicio: el protocolo HTTP ha desarrollado un rico conjunto de códigos de estado para indicar el éxito o el fracaso de un servicio. Los servicios basados en REST aprovechan estos códigos de estado HTTP y otra infraestructura basada en web, como proxies inversos y cachés, que pueden integrarse con sus microservicios con relativa facilidad.

HTTP es el lenguaje de la web y el uso de HTTP como marco filosófico para Crear su servicio es clave para crear servicios en la nube.

<sup>a</sup> Probablemente la cobertura más completa del diseño de servicios REST sea el libro REST in Practice. por Ian Robinson, et al (O'Reilly, 2010).

Su primera clase de controlador se encuentra en src/main/java/com/thinkmechanix/licencias/controladores/LicenseServiceController.java. Esta clase exponga cuatro puntos finales HTTP que se asignarán a los verbos POST, GET, PUT y DELETE .

Repasemos la clase de controlador y veamos cómo Spring Boot proporciona un conjunto de anotaciones que mantiene el esfuerzo necesario para exponer los puntos finales de su servicio al mínimo y le permite concentrarse en desarrollar la lógica de negocios para el servicio. Empezaremos mirando la definición básica de la clase del controlador sin ningún método de clase todavía.

La siguiente lista muestra la clase de controlador que creó para su servicio de licencias.

#### Listado 2.3 Marcar LicenseServiceController como Spring RestController

```
paquete com.thinkmechanix.licenses.controllers;
```

```
importar... // Eliminado por motivos de concisión
```

```

@RestController
@RequestMapping(value="/v1/organizaciones/{organizationId}/licenses")
clase pública LicenseServiceController {
    //Cuerpo de la clase eliminado por motivos de concisión
}

```

@RestController le dice a Spring Boot que este es un servicio basado en REST y que serializará/deserializará automáticamente la solicitud/respuesta del servicio a JSON.

Expone todos los puntos finales HTTP de esta clase con un prefijo de /v1/organizaciones/(organizationId)/licenses

Comenzaremos nuestra exploración mirando la anotación `@RestController`. El `@RestController` es una anotación Java a nivel de clase y le dice al Spring Container que Esta clase Java se utilizará para un servicio basado en REST. Esta anotación maneja automáticamente la serialización de los datos pasados a los servicios como JSON o XML (por defecto, la clase `@RestController` serializará los datos devueltos en JSON). A diferencia de la anotación tradicional Spring `@Controller`, la anotación `@RestController` no requiere que usted, como desarrollador, devuelva una clase `ResponseBody` de su clase de controlador. Todo esto se maneja mediante la presencia de la anotación `@RestController`, que incluye la anotación `@ResponseBody`.

#### ¿Por qué JSON para microservicios?

Se pueden utilizar múltiples protocolos para enviar datos de ida y vuelta entre servidores basados en HTTP. JSON se ha convertido en el estándar de facto por varias razones.

En primer lugar, en comparación con otros protocolos como el SOAP basado en XML (Simple Object Protocol de acceso), es extremadamente liviano ya que puede expresar sus datos sin tener mucha sobrecarga textual.

En segundo lugar, un ser humano puede leerlo y consumirlo fácilmente. Ésta es una cualidad subestimada a la hora de elegir un protocolo de serialización. Cuando surge un problema, es fundamental que los desarrolladores observen un fragmento de JSON y procesen visualmente y rápidamente lo que contiene. El La simplicidad del protocolo hace que esto sea increíblemente fácil de hacer.

En tercer lugar, JSON es el protocolo de serialización predeterminado utilizado en JavaScript. Desde el dramático ascenso de JavaScript como lenguaje de programación y el igualmente dramático ascenso de Aplicaciones de Internet de una sola página (SPIA) que dependen en gran medida de JavaScript, JSON tiene convertirse en una opción natural para crear aplicaciones basadas en REST porque es lo que los clientes web front-end utilizan para llamar a los servicios.

Otros mecanismos y protocolos son más eficientes que JSON para comunicarse. entre servicios. El ahorro de Apache (<http://thrift.apache.org>) El marco te permite construir servicios multilingües que puedan comunicarse entre sí mediante un protocolo binario. El protocolo Apache Avro (<http://avro.apache.org>) es un protocolo de serialización de datos que convierte datos de un lado a otro a un formato binario entre el cliente y llamadas al servidor.

Si necesita minimizar el tamaño de los datos que envía a través del cable, le recomiendo que consulte estos protocolos. Pero según mi experiencia, el uso de JSON directo en sus microservicios funciona de manera efectiva y no interpone otra capa. de comunicación para depurar entre los consumidores de servicios y los clientes de servicios.

La segunda anotación que se muestra en el listado 2.3 es la anotación `@RequestMapping`. Puede utilizar la anotación `@RequestMapping` como anotación a nivel de clase y de método. La anotación `@RequestMapping` se utiliza para indicarle al contenedor Spring el punto final HTTP que el servicio va a exponer al mundo. Cuando utiliza la anotación `@RequestMapping` a nivel de clase, está estableciendo la raíz de la URL para todos los demás puntos finales expuestos por el controlador.

En el listado 2.3, @RequestMapping(value="/v1/organizations/{organizationId}/licenses") usa el atributo de valor para establecer la raíz de la URL para todos los puntos finales expuestos en la clase de controlador. Todos los puntos finales de servicio expuestos en este controlador comenzarán con /v1/organizations/{organizationId}/licenses como raíz de su punto final. { organizationId } es un marcador de posición que indica cómo espera que se parametrice la URL con un OrganizationId pasado en cada llamada. El uso de OrganizationId en la URL le permite diferenciar entre los diferentes clientes que podrían utilizar su servicio.

Ahora agregará el primer método a su controlador. Este método implementará el verbo GET utilizado en una llamada REST y devolverá una única instancia de clase de Licencia , como se muestra en la siguiente lista. (Para los propósitos de esta discusión, creará una instancia de una clase Java llamada Licencia).

Listado 2.4 Exponer un punto final GET HTTP individual

Crea un punto final GET con el valor v1/organizations/{organizationId}/licenses{licenseId}

```
@RequestMapping(value="/{licenseId}",method = RequestMethod.GET) Licencia pública  
getLicenses(
```

```
    @PathVariable("organizationId") Cadena ID de organización,  
    @PathVariable("licenseId") String LicenseId) { return new
```

License() .withId(licenseId) .withProductName("Teleco") .withLicenseType("Seat") .withOrganizationId("TestOrg");  
}

Lo primero que ha hecho en este listado es anotar el método `getLicenses()` con una anotación `@RequestMapping` de nivel de método , pasando dos parámetros a la anotación: valor y método. Con una anotación `@RequestMapping` a nivel de método , se basa en la anotación de nivel raíz especificada en la parte superior de la clase para hacer coincidir todas las solicitudes HTTP que llegan al controlador con el punto final `/v1/organizations/ {organizationId}/licences/{ Id.licenciado}`. El segundo parámetro de la anotación, método, especifica el verbo HTTP con el que coincidirá el método. En el ejemplo anterior, está haciendo coincidir el método GET representado por la enumeración `RequestMethod.GET` .

La segunda cosa a tener en cuenta sobre el listado 2.4 es que se utiliza la anotación `@PathVariable` en el cuerpo del parámetro del método `getLicenses()`. (2) La anotación `@PathVariable` se utiliza para asignar los valores de los parámetros pasados en la URL entrante.

(como lo indica la sintaxis de {parameterName} ) a los parámetros de su método. En su ejemplo de código del listado 2.4, está asignando dos parámetros de la URL, OrganizationId y Licenseld, a dos variables de nivel de parámetro en el método:

```
@PathVariable("organizationId") Cadena ID de organización,
@PathVariable("licenseld") Cadena licenseld)
```

**Los nombres de los puntos finales importan**

Antes de avanzar demasiado en el camino de la escritura de microservicios, asegúrese de (y potencialmente otros equipos de su organización) establecen estándares para los puntos finales que estarán expuestos a través de sus servicios. Las URL (Localizador uniforme de recursos) para el microservicio debe usarse para comunicar claramente la intención del servicio, los recursos que gestiona el servicio y las relaciones que existen entre los recursos gestionados dentro del servicio. He encontrado útiles las siguientes pautas para puntos finales del servicio de nombres:

- 1 Utilice nombres de URL claros que establezcan qué recurso representa el servicio:  
Tener un formato canónico para definir URL ayudará a que su API se sienta más intuitiva y fácil de usar. Sea coherente en sus convenciones de nomenclatura.
- 2 Utilice la URL para establecer relaciones entre recursos. A menudo, tener una relación padre-hijo entre los recursos dentro de sus microservicios donde el niño no existe fuera del contexto del padre (por lo tanto, Es posible que no tenga un microservicio independiente para el niño). Utilice las URL para expresar estas relaciones. Pero si descubre que sus URL tienden a ser excesivamente largas y anidadas, es posible que su microservicio esté intentando hacer demasiado.
- 3 Establezca un esquema de control de versiones para las URL desde el principio: la URL y su correspondiente Los puntos finales representan un contrato entre el propietario del servicio y el consumidor de el servicio. Un patrón común es anteponer a todos los puntos finales una versión número. Establezca su esquema de versiones con anticipación y cúmplalo. Es extremadamente Es difícil adaptar las versiones a las URL después de que ya hay varios consumidores usándolas.

En este punto, tiene algo a lo que puede llamar como servicio. Desde una ventana de línea de comando, vaya al directorio de su proyecto donde descargó el código de muestra y ejecute el siguiente comando de Maven:

```
mvn arranque de primavera: ejecutar
```

Tan pronto como presione la tecla Retorno, debería ver que Spring Boot inicia un incrustado Servidor Tomcat y comience a escuchar en el puerto 8080.

```
o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup
o.s.c.support.DefaultLifecycleProcessor : Starting beans in phase 0
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http) ←
c.thoughtmechanix.licenses.Application : Started Application in 3.465 seconds (JVM running for
```

La licencia  
servidor iniciando  
en el puerto 8080

Figura 2.4 El servicio de licencias se inicia correctamente

Una vez que se inicia el servicio, puede acceder directamente al punto final expuesto. Porque su El primer método expuesto es una llamada GET , puede utilizar varios métodos para invocar el servicio. Mi método preferido es utilizar una herramienta basada en Chrome como POSTMAN o CURL. por llamar al servicio. La Figura 2.5 muestra un GET realizado en <http://localhost:8080/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a> punto final.

```

1 {
2   "id": "f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a",
3   "organizationId": "TestOrg",
4   "productName": "Teleco",
5   "licenseType": "Seat"
6 }

```

Cuando se llama al punto final GET, se devuelve una carga útil JSON que contiene datos de licencia.

Figura 2.5 Llamada a su servicio de licencias con POSTMAN

En este punto, tiene el esqueleto en ejecución de un servicio. Pero desde una perspectiva de desarrollo, este servicio no está completo. Un buen diseño de microservicio no evita segregar el servicio en capas de acceso a datos y lógica empresarial bien definidas. Como tu Si avanza en capítulos posteriores, continuará iterando sobre este servicio y profundizando más. sobre cómo estructurarlo.

Pasemos a la perspectiva final: explorar cómo un ingeniero de DevOps haría operativo el servicio y lo empaquetaría para su implementación en la nube.

## 2.4 La historia de DevOps: construir para los rigores del tiempo de ejecución

Para el ingeniero de DevOps, el diseño del microservicio tiene que ver con la gestión del servicio después de que entre en producción. Escribir el código suele ser la parte fácil. Manteniéndola correr es la parte difícil.

Si bien DevOps es un campo de TI rico y emergente , comenzará su esfuerzo de desarrollo de microservicios con cuatro principios y se basará en estos principios más adelante en el libro. Estos principios son

- 1 Un microservicio debe ser autónomo y desplegarse de forma independiente con múltiples Casos en los que el servicio se inicia y se deshace con un solo software. artefacto.
- 2 Un microservicio debe ser configurable. Cuando se inicia una instancia de servicio, debe leer los datos que necesita para configurarse desde una ubicación central o tener

su información de configuración se transmite como variables de entorno. Ningún humano  
Se debe requerir intervención para configurar el servicio.

- 3 Una instancia de microservicio debe ser transparente para el cliente. El cliente debe Nunca se sabe la ubicación exacta de un servicio. En cambio, un cliente de microservicio debería hablar con un agente de descubrimiento de servicios que permitirá que la aplicación localice una instancia de un microservicio sin tener que conocer su ubicación física.
- 4 Un microservicio debe comunicar su estado. Esta es una parte crítica de su nube arquitectura. Las instancias de microservicio fallarán y los clientes deberán realizar rutas malos casos de servicio.

Estos cuatro principios exponen la paradoja que puede existir con el desarrollo de microservicios. Los microservicios son más pequeños en tamaño y alcance, pero su uso introduce más partes móviles en una aplicación, especialmente porque los microservicios están distribuidos y funcionando independientemente unos de otros en sus propios contenedores distribuidos. Esto introduce un alto grado de coordinación y más oportunidades para puntos de falla en la solicitud.

Desde una perspectiva de DevOps, debe abordar las necesidades operativas de un microservicio desde el principio y traducir estos cuatro principios en un conjunto estándar de eventos del ciclo de vida. que ocurren cada vez que se construye e implementa un microservicio en un entorno. El Se pueden asignar cuatro principios a los siguientes pasos del ciclo de vida operativo:

- Montaje del servicio: ¿cómo empaqueta e implementa su servicio para garantizar repetibilidad y coherencia para que se utilice el mismo código de servicio y tiempo de ejecución.  
¿Se implementa exactamente de la misma manera?
- Arranque del servicio: ¿cómo se separa la aplicación y el código de configuración específico del entorno del código de tiempo de ejecución para que pueda iniciar e implementar?  
¿Una instancia de microservicio rápidamente en cualquier entorno sin intervención humana para configurar el microservicio?
- Registro/descubrimiento de servicios: cuando se implementa una nueva instancia de microservicio,  
¿Cómo se puede hacer que otra aplicación pueda descubrir la nueva instancia de servicio?  
¿clientela?
- Monitoreo de servicios: en un entorno de microservicios es extremadamente común que Se ejecutarán varias instancias del mismo servicio debido a la alta disponibilidad. necesidades. Desde una perspectiva de DevOps, es necesario monitorear las instancias de microservicios y asegúrese de que cualquier falla en su microservicio se solucione y que las instancias de servicio defectuosas se eliminen.

La figura 2.6 muestra cómo encajan estos cuatro pasos.

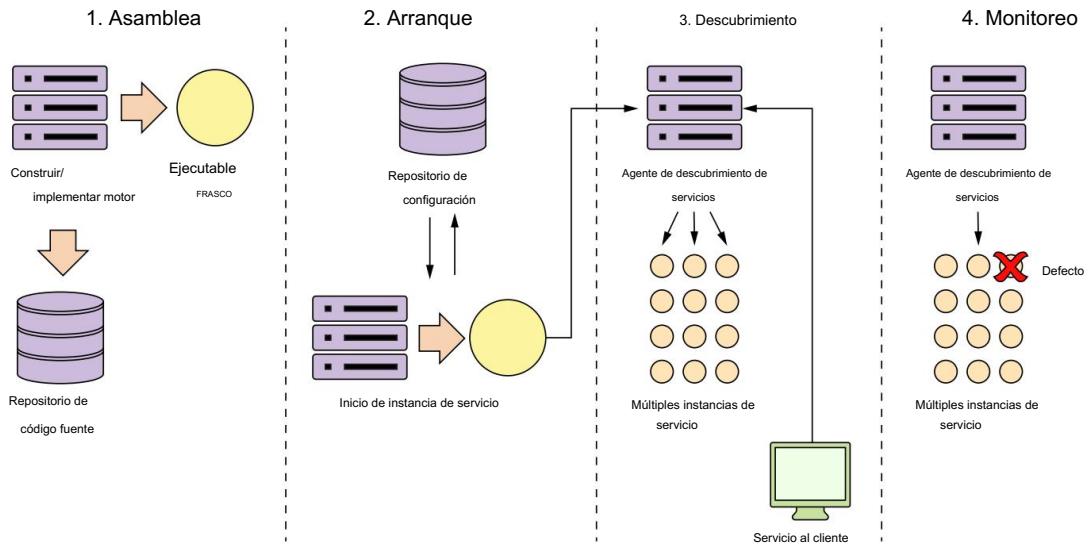


Figura 2.6 Cuando se inicia un microservicio, pasa por varios pasos en su ciclo de vida.

#### Creación de la aplicación de servicio de microservicio de doce factores Una de mis

mayores esperanzas con este libro es que se dé cuenta de que una arquitectura de microservicio exitosa requiere un sólido desarrollo de aplicaciones y prácticas de DevOps. Uno de los resúmenes más sucintos de estas prácticas se puede encontrar en el manifiesto de aplicación de los doce factores de Heroku (<https://12factor.net/>). Este documento proporciona 12 mejores prácticas que siempre debe tener en cuenta al crear microservicios. A medida que lea este libro, verá estas prácticas entrelazadas en los ejemplos.

Los he resumido de la siguiente manera:

**Base de código:** todo el código de la aplicación y la información de aprovisionamiento del servidor deben estar bajo control de versiones. Cada microservicio debe tener su propio repositorio de código independiente dentro de los sistemas de control de código fuente.

**Dependencias:** declare explícitamente las dependencias que utiliza su aplicación a través de herramientas de compilación como Maven (Java). La dependencia de JAR de terceros debe declararse utilizando sus números de versión específicos. Esto permite que su microservicio se cree siempre utilizando la misma versión de bibliotecas.

**Config:** almacene la configuración de su aplicación (especialmente la configuración específica de su entorno) independientemente de su código. La configuración de su aplicación nunca debe estar en el mismo repositorio que su código fuente.

**Servicios de respaldo:** su microservicio a menudo se comunicará a través de una red con una base de datos o un sistema de mensajería. Cuando lo haga, debe asegurarse de que en cualquier momento pueda cambiar su implementación de la base de datos de un servicio administrado internamente a un servicio de terceros. En el capítulo 10, demostramos esto cuando traslada sus servicios de una base de datos de Postgres administrada localmente a una administrada por Amazon.

(continuado)

Compilación, lanzamiento, ejecución: mantenga las partes de compilación, lanzamiento y ejecución de la implementación de su aplicación completamente separadas. Una vez creado el código, el desarrollador nunca debe hacer cambios en el código en tiempo de ejecución. Cualquier cambio debe volver al proceso de compilación. y ser redistribuido. Un servicio creado es inmutable y no se puede cambiar.

Procesos: sus microservicios siempre deben ser sin estado. Se les puede matar y reemplazado en cualquier momento sin temor a que una instancia de pérdida de servicio resulte en pérdida de datos.

Enlace de puerto: un microservicio es completamente autónomo con el motor de ejecución para el servicio empaquetado en el ejecutable del servicio. Debes ejecutar el servicio sin la necesidad de un servidor web o de aplicaciones independiente. El servicio debería iniciarse solo. en la línea de comando y se podrá acceder a él inmediatamente a través de un puerto HTTP expuesto.

Simultaneidad: cuando necesite escalar, no confíe en un modelo de subprocessos dentro de un único servicio. En su lugar, lance más instancias de microservicios y escale horizontalmente. Este no excluye el uso de subprocessos dentro de su microservicio, pero no confíe en él como su único mecanismo para escalar. Ampliar, no aumentar.

Disponibilidad: los microservicios son desechables y se pueden iniciar y detener en demanda. Se debe minimizar el tiempo de inicio y los procesos deben cerrarse correctamente cuando reciben una señal de interrupción del sistema operativo.

Paridad de desarrollo/producción: minimiza las brechas que existen entre todos los entornos en que ejecuta el servicio (incluido el escritorio del desarrollador). Un desarrollador debería utilizar la misma infraestructura localmente para el desarrollo del servicio en la que se ejecutará el servicio real. También significa que la cantidad de tiempo que se implementa un servicio entre Los entornos deben durar horas, no semanas. Tan pronto como se confirma el código, debería ser probado y luego promocionado lo más rápido posible desde Dev hasta Prod.

Registros: los registros son un flujo de eventos. A medida que se escriben los registros, deberían poder transmitirse a herramientas como Splunk (<http://splunk.com>), o Fluentd (<http://fluentd.org>), eso recopilará los registros y los escribirá en una ubicación central. El microservicio debe Nunca te preocunes por la mecánica de cómo sucede esto y el desarrollador Debería mirar visualmente los registros a través de STDOUT a medida que se escriben.

Procesos de administración: los desarrolladores a menudo tendrán que realizar tareas administrativas en contra de sus servicios (migración o conversión de datos). Estas tareas nunca deben ser ad hoc y en su lugar, debe hacerse a través de scripts que se administran y mantienen a través del depósito de código fuente. Estos guiones deben ser repetibles y no cambiantes (el el código del script no se modifica para cada entorno) en cada entorno en el que se ejecutan contra.

#### 2.4.1 Montaje de servicios: empaquetar e implementar sus microservicios

Desde una perspectiva de DevOps, uno de los conceptos clave detrás de una arquitectura de microservicio es que se pueden implementar rápidamente múltiples instancias de un microservicio en respuesta a un entorno de aplicación de cambio (por ejemplo, una afluencia repentina de solicitudes de usuarios, problemas dentro de la infraestructura, etc.).

Para lograr esto, un microservicio debe empaquetarse e instalarse como un único artefacto con todas sus dependencias definidas dentro de él. Este artefacto puede entonces ser implementado en cualquier servidor con un JDK de Java instalado. Estas dependencias también incluir el motor de ejecución (por ejemplo, un servidor HTTP o un contenedor de aplicaciones) que alojará el microservicio.

Este proceso de construcción, empaquetado e implementación consistentes es el servicio montaje (paso 1 en la figura 2.6). La Figura 2.7 muestra detalles adicionales sobre el servicio. paso de montaje.

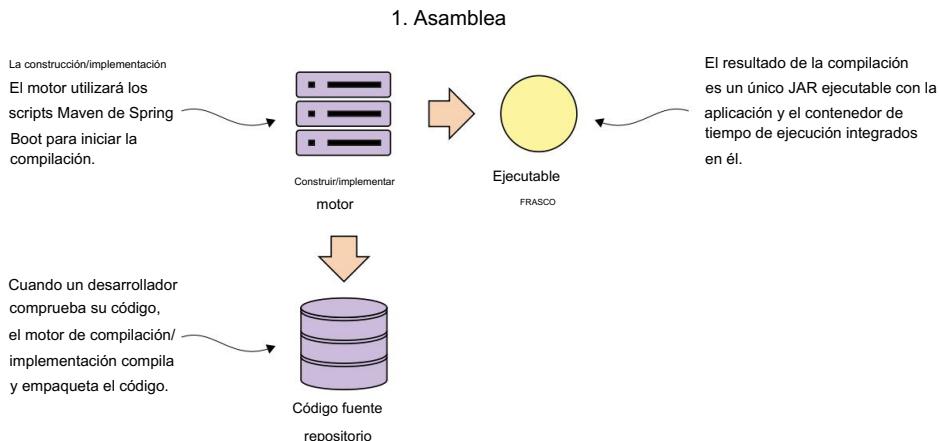


Figura 2.7 En el paso de ensamblaje del servicio, el código fuente se compila y empaqueta con su motor de ejecución.

Afortunadamente, casi todos los marcos de microservicios de Java incluirán un motor de ejecución. que se puede empaquetar e implementar con el código. Por ejemplo, en Spring Boot. Por ejemplo en la figura 2.7, puede usar Maven y Spring Boot para crear un ejecutable Java. jar que tiene un motor Tomcat integrado directamente en el JAR. En el siguiente ejemplo de línea de comandos, está creando el servicio de licencias como un JAR ejecutable y luego iniciando el archivo JAR desde la línea de comandos:

```
Paquete mvn clean && java -jar target/licensing-service-0.0.1-SNAPSHOT.jar
```

Para ciertos equipos de operaciones, el concepto de incorporar un entorno de ejecución en el archivo JAR hay un cambio importante en la forma en que piensan sobre la implementación de aplicaciones. en un organización empresarial J2EE tradicional , una aplicación se implementa en una aplicación servidor. Este modelo implica que el servidor de aplicaciones es una entidad en sí misma y a menudo sería administrado por un equipo de administradores de sistemas que administraban la configuración de los servidores independientemente de las aplicaciones que se implementaban en ellos.

Esta separación de la configuración del servidor de aplicaciones de la aplicación introduce puntos de falla en el proceso de implementación, porque en muchas organizaciones el

La configuración de los servidores de aplicaciones no se mantiene bajo control de origen y se administra a través de una combinación de interfaz de usuario y administración interna. guiones. Es muy fácil que la deriva de la configuración se introduzca en el entorno del servidor de aplicaciones y provoque repentinamente lo que, en la superficie, parecen ser interrupciones aleatorias.

El uso de un único artefacto desplegable con el motor de ejecución integrado en el El artefacto elimina muchas de estas oportunidades de desviación de la configuración. También permite usted puede poner todo el artefacto bajo control de código fuente y permite que el equipo de la aplicación ser capaz de razonar mejor sobre cómo se construye e implementa su aplicación.

#### 2.4.2 Arranque del servicio: gestión de la configuración de su microservicios

El arranque del servicio (paso 2 en la figura 2.6) ocurre cuando el microservicio se inicia por primera vez y necesita cargar la información de configuración de su aplicación. La Figura 2.8 proporciona más contexto para el procesamiento de arranque.

Como sabe cualquier desarrollador de aplicaciones, habrá ocasiones en las que necesitará hacer El comportamiento en tiempo de ejecución de la aplicación es configurable. Generalmente esto implica leer los datos de configuración de su aplicación desde un archivo de propiedades implementado con la aplicación o leer los datos de un almacén de datos, como una base de datos relacional.

Los microservicios suelen encontrarse con el mismo tipo de requisitos de configuración. La diferencia es que en una aplicación de microservicio que se ejecuta en la nube, es posible que tenga

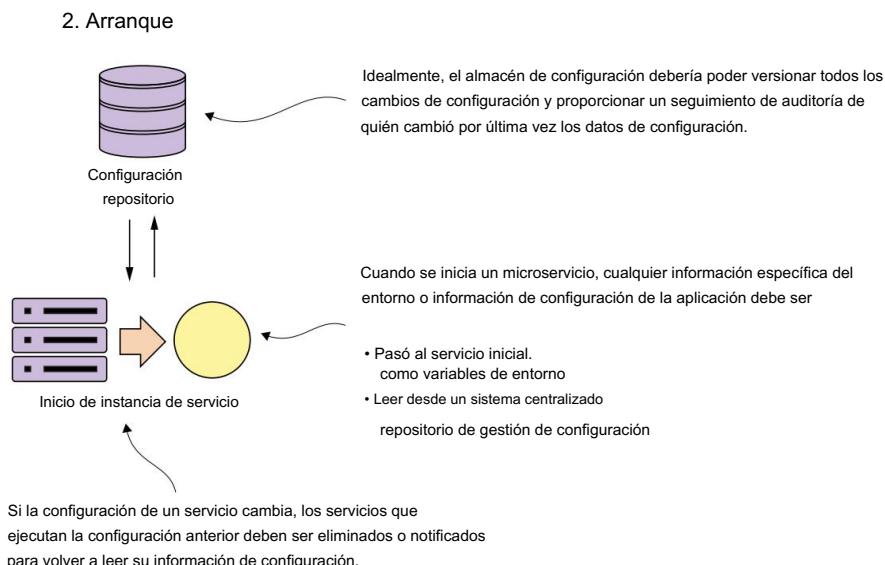


Figura 2.8 Cuando se inicia un servicio (arranque), lee su configuración desde un repositorio central.

cientos o incluso miles de instancias de microservicios en ejecución. Complicando aún más Esto es que los servicios podrían extenderse por todo el mundo. Con una gran cantidad de servicios dispersos geográficamente, resulta inviable volver a implementar sus servicios para elegir crear nuevos datos de configuración.

Almacenar los datos en un almacén de datos externo al servicio resuelve este problema, pero Los microservicios en la nube ofrecen un conjunto de desafíos únicos:

- 1 Los datos de configuración tienden a tener una estructura simple y generalmente se leen con frecuencia y se escriben con poca frecuencia. Las bases de datos relacionales son excesivas en esta situación porque están diseñadas para gestionar modelos de datos mucho más complicados.  
luego, un conjunto simple de pares clave-valor.
- 2 Debido a que se accede a los datos de forma regular pero cambian con poca frecuencia, el  
Los datos deben ser legibles con un bajo nivel de latencia.
- 3 El almacén de datos debe tener alta disponibilidad y estar cerca de los servicios que leen el  
datos. Un almacén de datos de configuración no puede dejar de funcionar por completo porque convertirse en un punto único de falla para su aplicación.

En el capítulo 3, muestro cómo administrar los datos de configuración de su aplicación de microservicio. usando cosas como un simple almacén de datos clave-valor.

#### 2.4.3 Registro y descubrimiento de servicios: cómo se comunican los clientes con tus microservicios

Desde la perspectiva del consumidor de microservicios, un microservicio debe ser transparente en cuanto a ubicación, porque en un entorno basado en la nube, los servidores son efímeros. Efímero significa que los servidores en los que se aloja un servicio generalmente tienen vidas más cortas que un servicio ejecutándose en un centro de datos corporativo. Los servicios basados en la nube se pueden iniciar y desmantelar rápidamente con una dirección IP completamente nueva asignada al servidor en el que se ejecutan los servicios.

Al insistir en que los servicios se traten como objetos desecharables de corta duración, los microservicios Las arquitecturas pueden lograr un alto grado de escalabilidad y disponibilidad al tener varias instancias de un servicio en ejecución. La demanda de servicios y la resiliencia se pueden gestionar como rápidamente según lo amerite la situación. Cada servicio tiene una IP única y no permanente dirección que le fue asignada. La desventaja de los servicios efímeros es que, dado que los servicios suben y bajan constantemente, es necesario gestionar manualmente un gran conjunto de servicios efímeros. o con la mano es una invitación a un apagón.

Una instancia de microservicio debe registrarse con el agente externo. Este proceso de registro se denomina descubrimiento de servicios (consulte el paso 3, descubrimiento de servicios, en la figura 2.6; consulte la figura 2.9 para obtener detalles sobre este proceso). Cuando una instancia de microservicio se registra con un agente de descubrimiento de servicios, le dirá al agente de descubrimiento dos cosas: la IP física dirección o dirección de dominio de la instancia de servicio y un nombre lógico que una aplicación puede utilizar para buscar en un servicio. Ciertos agentes de descubrimiento de servicios también requerirán un

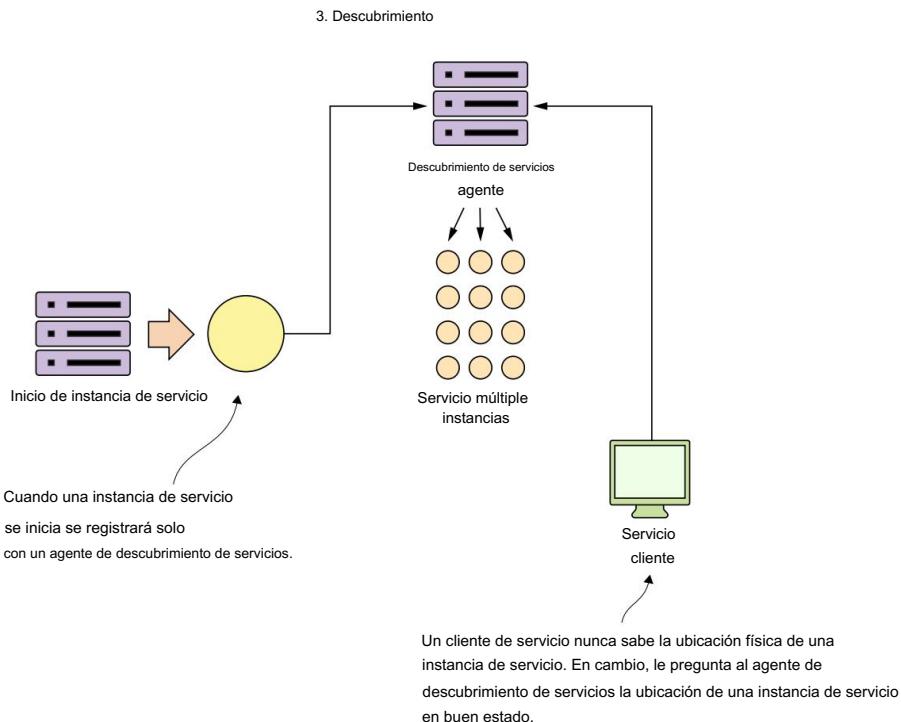


Figura 2.9 Un agente de descubrimiento de servicios abstracta la ubicación física de un servicio.

URL de regreso al servicio de registro que puede utilizar el agente de descubrimiento de servicios para realizar controles de salud.

Luego, el cliente del servicio se comunica con el agente de descubrimiento para buscar la ubicación del servicio.

#### 2.4.4 Comunicar el estado de un microservicio

Un agente de descubrimiento de servicios no actúa sólo como un policía de tráfico que guía al cliente hacia el ubicación del servicio. En una aplicación de microservicio basada en la nube, a menudo tendrás múltiples instancias de un servicio en ejecución. Tarde o temprano, una de esas instancias de servicio fallará. El agente de descubrimiento de servicios monitorea el estado de cada instancia de servicio registrada en él y elimina cualquier instancia de servicio de sus tablas de enrutamiento para garantizar que a los clientes no se les envía una instancia de servicio que haya fallado.

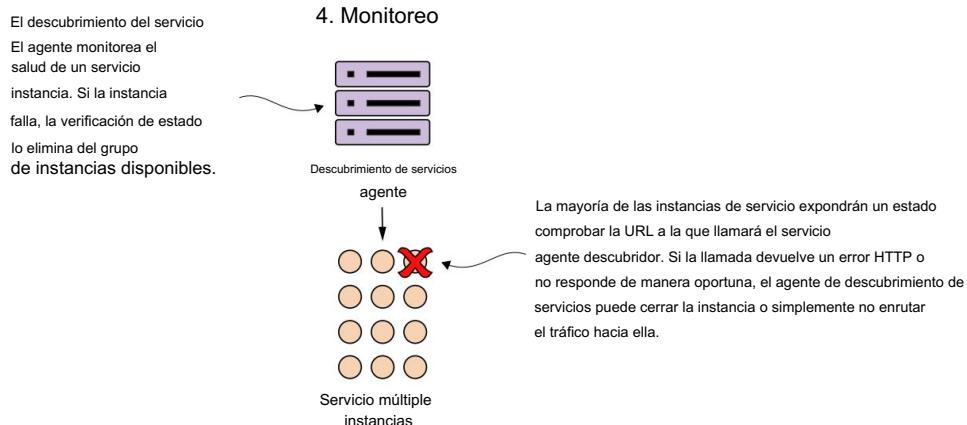


Figura 2.10 El agente de descubrimiento de servicios utiliza la URL de estado de exposición para verificar el estado del microservicio.

Después de que haya surgido un microservicio, el agente de descubrimiento de servicios continuará monitoreando y haga ping a la interfaz de verificación de estado para garantizar que ese servicio esté disponible. este es el paso 4 en la figura 2.6. La Figura 2.10 proporciona el contexto para este paso.

Al crear una interfaz de verificación de estado consistente, puede utilizar herramientas de monitoreo basadas en la nube para detectar problemas y responder a ellos de manera adecuada.

Si el agente de descubrimiento de servicios descubre un problema con una instancia de servicio, puede tomar medidas correctivas, como cerrar la instancia defectuosa o traer recursos adicionales

instancias de servicio arriba.

En un entorno de microservicios que utiliza REST, la forma más sencilla de crear un entorno de salud La interfaz de verificación es exponer un punto final HTTP que puede devolver una carga útil JSON y Código de estado HTTP . En un microservicio no basado en Spring-Boot, a menudo es responsabilidad del desarrollador responsabilidad de escribir un punto final que devolverá el estado del servicio.

En Spring Boot, exponer un punto final es trivial y no implica nada más que modificando su archivo de compilación Maven para incluir el módulo Spring Actuator. Spring Actuator proporciona puntos finales operativos listos para usar que lo ayudarán a comprender y gestionar la salud de su servicio. Para utilizar el actuador de resorte, debe asegurarse de incluya las siguientes dependencias en su archivo de compilación Maven:

```
<dependencia>
    <groupId>org.springframework.boot</groupId>
    <artifactId>actuator-de-arranque-de-arranque-resorte</artifactId>
</dependencia>
```

```

1 {
2   "status": "UP",
3   "diskSpace": {
4     "status": "UP",
5     "total": 63371726848,
6     "free": 22607110144,
7     "threshold": 10485760
8   }
9 }
  
```

La verificación de estado de Spring Boot "lista para usar" devolverá si el servicio está activo y cierta información básica, como cuánto espacio en disco queda en el servidor.

Figura 2.11 Una verificación de estado de cada instancia de servicio permite que las herramientas de monitoreo determinen si la instancia de servicio se está ejecutando.

Si accede al punto final `http://localhost:8080/health` en el servicio de licencias, debería ver los datos de salud devueltos. La Figura 2.11 proporciona un ejemplo de los datos devueltos.

Como puede ver en la figura 2.11, el control de estado puede ser más que un indicador de los altibajos. También puede brindar información sobre el estado del servidor en el que se ejecuta la instancia del microservicio. Esto permite una experiencia de monitoreo mucho más rica.<sup>1</sup>

## 2.5 Uniendo las perspectivas Los microservicios en

la nube parecen engañosamente simples. Pero para tener éxito con ellos, es necesario tener una vista integrada que reúna la perspectiva del arquitecto, el desarrollador y el ingeniero de DevOps en una visión cohesiva. Las conclusiones clave para cada una de estas perspectivas son:

**Arquitecto:** céntrese en los contornos naturales de su problema empresarial. Describe el dominio de tu problema empresarial y escucha la historia que estás contando. **Objetivo**

<sup>1</sup> Spring Boot ofrece una cantidad significativa de opciones para personalizar su control de salud. Para obtener más detalles sobre esto, consulte el excelente libro Spring Boot in Action (Manning Publications, 2015). El autor Craig Walls ofrece una descripción exhaustiva de los diferentes mecanismos para configurar los actuadores Spring Boot.

Surgirán candidatos a microservicios. Recuerde también que es mejor comenzar con un microservicio “de grano grueso” y refactorizarlo a servicios más pequeños que comenzar con un grupo grande de servicios pequeños. Las arquitecturas de microservicios, como la mayoría de las buenas arquitecturas, son emergentes y no están planificadas de antemano al minuto.

- 2 Ingeniero de software: el hecho de que el servicio sea pequeño no significa que los buenos principios de diseño se desperdicien. Concéntrese en crear un servicio en capas donde cada capa del servicio tenga responsabilidades discretas. Evite la tentación de crear marcos en su código e intente que cada microservicio sea completamente independiente. El diseño y la adopción prematuros del marco pueden tener enormes costos de mantenimiento más adelante en el ciclo de vida de la aplicación.
- 3 Ingeniero de DevOps: los servicios no existen en el vacío. Establezca el ciclo de vida de sus servicios con anticipación. La perspectiva de DevOps debe centrarse no sólo en cómo automatizar la creación y la implementación de un servicio, sino también en cómo monitorear el estado del servicio y reaccionar cuando algo sale mal. Operar un servicio a menudo requiere más trabajo y previsión que escribir una lógica empresarial.

## 2.6 Resumen

Para tener éxito con los microservicios, es necesario integrarlos en la arquitectura, perspectivas del desarrollador de software y de DevOps.

Los microservicios, si bien son un poderoso paradigma arquitectónico, tienen sus ventajas y desventajas. No todas las aplicaciones deberían ser aplicaciones de microservicios. Desde la perspectiva de un arquitecto, los microservicios son pequeños, autónomos y distribuidos. Los microservicios deberían tener límites estrechos y gestionar un pequeño conjunto de datos.

Desde la perspectiva de un desarrollador, los microservicios generalmente se crean utilizando un estilo de diseño REST , con JSON como carga útil para enviar y recibir datos del servicio.

Spring Boot es el marco ideal para crear microservicios porque le permite crear un servicio JSON basado en REST con algunas anotaciones simples. Desde la perspectiva de DevOp, la forma en que se empaqueta, implementa y monitorea un microservicio es de vital importancia. Spring Boot, listo para usar, le permite entregar un servicio como un único archivo JAR ejecutable. Un servidor Tomcat integrado en el archivo JAR del productor aloja el servicio. Spring Actuator, que se incluye con el marco Spring Boot, expone información sobre el estado operativo del servicio junto con información sobre el tiempo de ejecución de los servicios.

# Controlando su configuración con Spring Servidor de configuración en la nube

## Este capítulo cubre

Separar la configuración del servicio del servicio código

Configuración de un servidor de configuración de Spring

Cloud Integración de un microservicio Spring

Boot Cifrado de propiedades confidenciales

En un momento u otro, un desarrollador se verá obligado a separar la información de configuración de su código. Después de todo, desde la escuela se les ha inculcado en la cabeza que no deberían codificar valores en el código de la aplicación. Muchos desarrolladores utilizarán un archivo de clase de constantes en su aplicación para ayudar a centralizar toda su configuración en un solo lugar. Los datos de configuración de la aplicación escritos directamente en el código suelen ser problemáticos porque cada vez que se debe realizar un cambio en la configuración, es necesario volver a compilar y/o implementar la aplicación. Para evitar esto, los desarrolladores separarán completamente la información de configuración del código de la aplicación.

Esto facilita la realización de cambios en la configuración sin pasar por un proceso de recompilación, pero también introduce complejidad porque ahora tiene otro artefacto que debe administrarse e implementarse con la aplicación.

Muchos desarrolladores recurrirán al humilde archivo de propiedades (o YAML, JSON o XML) para almacenar su información de configuración. Este archivo de propiedades permanecerá en un servidor con frecuencia, que contiene información de conexión de base de datos y middleware y metadatos sobre el aplicación que impulsará el comportamiento de la aplicación. Segregando su aplicación en un archivo de propiedad es fácil y la mayoría de los desarrolladores nunca hacen más operacionalización de la configuración de su aplicación y luego colocar su archivo de configuración en la fuente control (si es así) e implementarlo como parte de su aplicación.

Este enfoque puede funcionar con una pequeña cantidad de aplicaciones, pero rápidamente deja de funcionar, aparte cuando se trata de aplicaciones basadas en la nube que pueden contener cientos de microservicios, donde cada microservicio, a su vez, puede tener múltiples instancias de servicio correr.

De repente, la gestión de la configuración se convierte en un gran problema, ya que el equipo de aplicaciones y operaciones en un entorno basado en la nube tiene que luchar con un nido de ratas en el que se encuentran.

Los archivos de configuración van a donde. Se enfatiza el desarrollo de microservicios basados en la nube

- 1 Separar completamente la configuración de una aplicación del código real siendo desplegado
- 2 Construyendo el servidor y la aplicación y una imagen inmutable que nunca cambia a medida que se promueve a través de sus entornos
- 3 Inyectar cualquier información de configuración de la aplicación en el momento de inicio del servidor a través de variables de entorno o a través de un repositorio centralizado los microservicios de la aplicación se leen al inicio

Este capítulo le presentará los principios y patrones básicos necesarios para gestionar datos de configuración de la aplicación en una aplicación de microservicio basada en la nube.

### 3.1 Sobre la gestión de la configuración (y la complejidad)

La gestión de la configuración de las aplicaciones es fundamental para los microservicios que se ejecutan en la nube, porque las instancias de microservicios deben lanzarse rápidamente con un mínimo de intervención humana. intervención. Cada vez que un ser humano necesita configurar o tocar manualmente un servicio para implementarlo es una oportunidad para que se produzca una desviación de la configuración, una interrupción inesperada, y un retraso en la respuesta a los desafíos de escalabilidad de la aplicación.

Comencemos nuestra discusión sobre la gestión de la configuración de aplicaciones estableciendo presentando cuatro principios que queremos seguir:

- 1 Segregar: queremos separar completamente la información de configuración de los servicios de la implementación física real de un servicio. La configuración de la aplicación no debe implementarse con la instancia de servicio. En cambio, la configuración La información debe pasarse al servicio inicial como variables de entorno o leerse desde un repositorio centralizado cuando se inicia el servicio.
- 2 Resumen: resumen el acceso a los datos de configuración detrás de una interfaz de servicio. En lugar de escribir código que acceda directamente al repositorio de servicios (que

es decir, leer los datos de un archivo o una base de datos utilizando JDBC), hacer que la aplicación utilice un servicio JSON basado en REST para recuperar los datos de configuración.

- 3 Centralizar: debido a que una aplicación basada en la nube puede tener literalmente cientos de servicios, es fundamental minimizar el número de repositorios diferentes utilizados para contener información de configuración. Centralice la configuración de su aplicación en el menor número posible de repositorios.
- 4 Harden: porque la información de configuración de su aplicación será completamente segregado de su servicio implementado y centralizado, es fundamental que cualquier solución que utilice se puede implementar para que tenga alta disponibilidad y redundante.

Una de las cosas clave que debe recordar es que cuando separa su configuración información fuera de su código real, está creando una dependencia externa que deberá gestionarse y controlarse la versión. No puedo enfatizar lo suficiente que el Es necesario realizar un seguimiento de los datos de configuración de la aplicación y controlar su versión porque La configuración de aplicaciones mal administrada es un caldo de cultivo para errores difíciles de detectar e interrupciones no planificadas.

#### Sobre la complejidad accidental

He experimentado de primera mano los peligros de no tener una estrategia para gestionar tu datos de configuración de la aplicación. Mientras trabajaba en una empresa de servicios financieros de Fortune 500, me pidieron que ayudara a volver a encarrilar un gran proyecto de actualización de WebSphere. La empresa en cuestión tenía más de 120 aplicaciones en WebSphere y necesitaba actualizar su infraestructura de WebSphere 6 a WebSphere 7 antes de que finalice todo. El entorno de la aplicación llegó al final de su vida útil en términos de mantenimiento por parte del proveedor.

El proyecto ya llevaba un año en marcha y sólo se había implementado una de 120 aplicaciones. El proyecto había costado un millón de dólares de esfuerzo en personal y costos de hardware, y con su trayectoria actual iba camino de tardar otros dos años en completarse. finalizar la actualización.

Cuando comencé a trabajar con el equipo de aplicaciones, uno (y solo uno) de los principales problemas que descubrí fue que el equipo de la aplicación gestionó toda su configuración. para sus bases de datos y los puntos finales de sus servicios dentro de archivos de propiedad. Estos Los archivos de propiedades se administraban manualmente y no estaban bajo control de código fuente. con 120 aplicaciones distribuidas en cuatro entornos y múltiples nodos WebSphere para cada aplicación, este nido de archivos de configuración llevó al equipo a intentar migrar 12.000 archivos de configuración distribuidos en cientos de servidores y aplicaciones que se ejecutan en el servidor. (Estás leyendo bien ese número: 12.000). Estos archivos eran solo para la configuración de la aplicación, ni siquiera para la configuración del servidor de aplicaciones.

Convencí al patrocinador del proyecto para que se tomara dos meses para consolidar todas las solicitudes. información hasta un repositorio de configuración centralizado y controlado por versiones con 20 Archivos de configuración. Cuando le pregunté al equipo del marco cómo llegaron las cosas al punto

donde tenían 12.000 archivos de configuración, el ingeniero principal del equipo dijo que originalmente diseñaron su estrategia de configuración en torno a un pequeño grupo de aplicaciones. Sin embargo, la cantidad de aplicaciones web creadas e implementadas se disparó a lo largo de cinco años y, aunque pidieron dinero y tiempo para reelaborar su enfoque de gestión de configuración, sus socios comerciales y líderes de TI nunca lo consideraron una prioridad.

No dedicar tiempo a determinar cómo se va a realizar la gestión de la configuración puede tener impactos posteriores reales (y costosos).

### 3.1.1 Su arquitectura de gestión de configuración

Como recordará del capítulo 2, la carga de la gestión de configuración para un microservicio ocurre durante la fase de arranque del microservicio. Como recordatorio, la figura 3.1 muestra el ciclo de vida del microservicio.

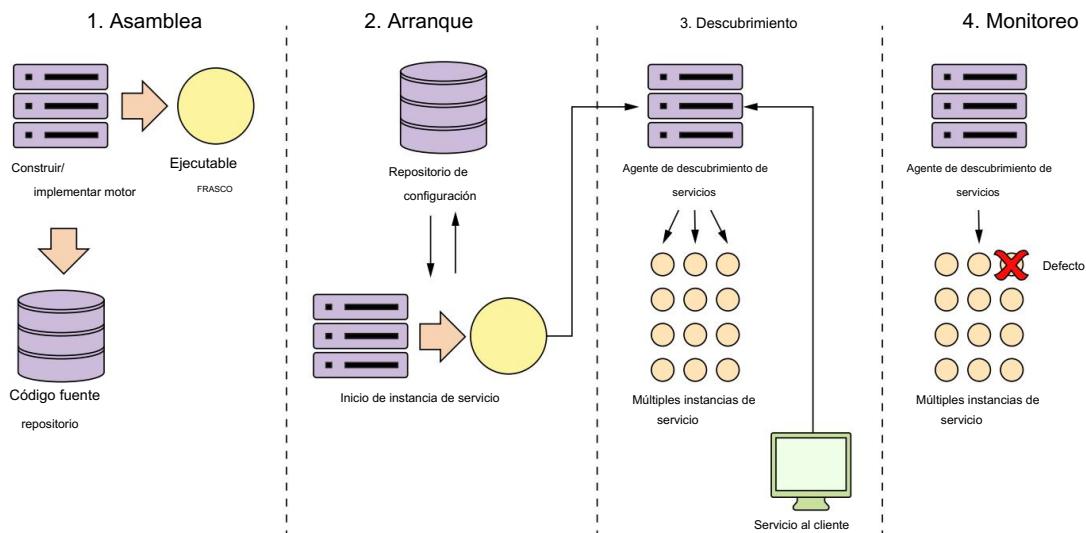


Figura 3.1 Los datos de configuración de la aplicación se leen durante la fase de arranque del servicio.

Tomemos los cuatro principios que establecimos anteriormente en la sección 3.1 (segregar, abstraer, centralizar y reforzar) y veamos cómo se aplican estos cuatro principios cuando el servicio se está iniciando. La Figura 3.2 explora el proceso de arranque con más detalle y muestra cómo un servicio de configuración desempeña un papel fundamental en este paso.

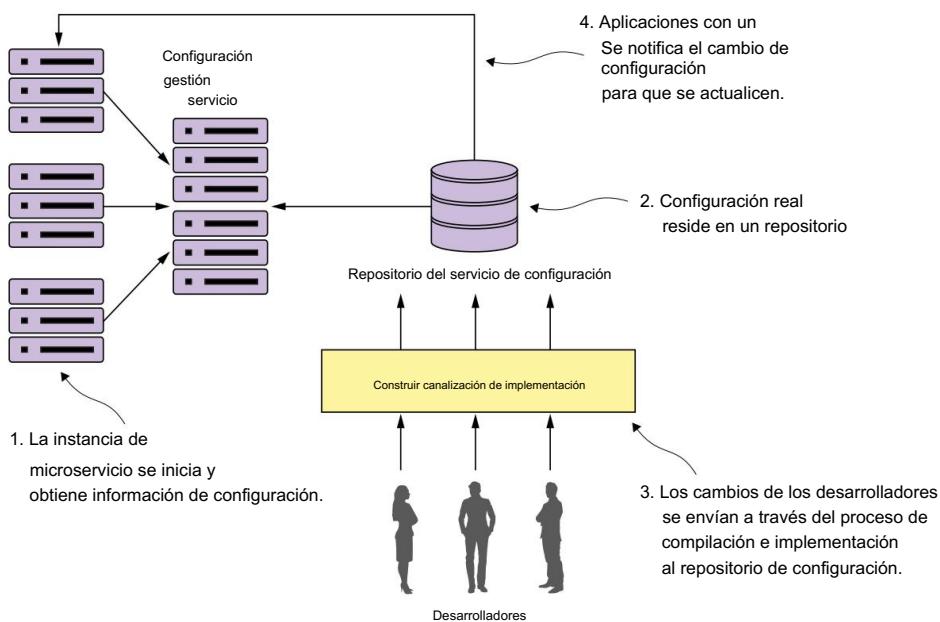


Figura 3.2 Arquitectura conceptual de gestión de configuración

En la figura 3.2, se ven varias actividades que se llevan a cabo:

- 1 Cuando aparece una instancia de microservicio, llamará a un punto final de servicio para leer su información de configuración que es específica del entorno en el que está operando. La información de conexión para la administración de la configuración (credenciales de conexión, punto final de servicio, etc.) se pasará al microservicio cuando se inicia.
- 2 La configuración real residirá en un repositorio. Basado en la implementación de su repositorio de configuración, puede optar por utilizar diferentes implementaciones para guardar sus datos de configuración. Las opciones de implementación pueden incluir archivos bajo control de fuente, una base de datos relacional o un almacenamiento de datos de valores clave.
- 3 La gestión real de los datos de configuración de la aplicación se produce independientemente de cómo se implemente la aplicación. Los cambios en la gestión de la configuración normalmente se manejan a través del proceso de construcción e implementación donde Los cambios de configuración se pueden etiquetar con información de versión y desplegados a través de los diferentes entornos.
- 4 Cuando se realiza un cambio en la gestión de la configuración, los servicios que utilizan ese Los datos de configuración de la aplicación deben ser notificados del cambio y actualizar su copia de los datos de la solicitud.

En este punto hemos trabajado a través de la arquitectura conceptual que ilustra las diferentes piezas de un patrón de gestión de configuración y cómo encajan estas piezas.

juntos. Ahora pasaremos a analizar las diferentes soluciones para el patrón y luego veremos una implementación concreta.

### 3.1.2 Opciones de implementación

Afortunadamente, puede elegir entre una gran cantidad de proyectos de código abierto probados en batalla para implementar una solución de gestión de configuración. Veamos varios de los diferentes opciones disponibles y compararlas. La tabla 3.1 presenta estas opciones.

Tabla 3.1 Proyectos de código abierto para implementar un sistema de gestión de configuración

Nombre del proyecto	Descripción	Características
Etc.	Proyecto de código abierto escrito en Go. Se utiliza para el descubrimiento de servicios y la gestión de valores clave. Utiliza la balsa ( <a href="https://balsa.github.io/">https://balsa.github.io/</a> ) protocolo para su modelo de computación distribuida.	Muy rápido y escalable Distribuible Impulsado por línea de comando Fácil de usar y configurar
eureka	Escrito por Netflix. Extremadamente probado en batalla. Se utiliza tanto para el descubrimiento de servicios como para la gestión de valores clave.	Distribuir almacén de valores clave. Flexible; requiere esfuerzo para configurar Ofrece actualización dinámica del cliente lista para usar
Cónsul	Escrito por Hashicorp. Similar a Etc y Eureka en características, pero utiliza un algoritmo diferente para su modelo de computación distribuida (protocolo SWIM; <a href="https://www.cs.cornell.edu/~asdas/research/dsn02-swim.pdf">https://www.cs.cornell.edu/~asdas/research/dsn02-swim.pdf</a> ).	Rápido Ofrece descubrimiento de servicios nativos con la opción de integrarse directamente con DNS. No ofrece actualización dinámica al cliente desde el primer momento
guardián del zoológico	Un proyecto de Apache que ofrece capacidades de bloqueo distribuido. A menudo se utiliza como solución de gestión de configuración para acceder a datos clave-valor.	La solución más antigua y más probada en batalla El más complejo de utilizar. Puede usarse para la gestión de la configuración, pero debe considerarse solo si ya está usando ZooKeeper en otras partes de su arquitectura.
Configuración de la nube de primavera servidor	Un proyecto de código abierto que ofrece una solución de gestión de configuración general con diferentes backends. Puede integrar con Git, Eureka y Consul como back-end.	Almacén de clave/valor no distribuido Ofrece una estrecha integración para servicios Spring y no Spring. Puede utilizar múltiples backends para registrar datos de configuración, incluido un sistema de archivos compartido, Eureka, Consul y Git.

Todas las soluciones de la tabla 3.1 se pueden utilizar fácilmente para crear un sistema de gestión de configuración. solución. Para los ejemplos de este capítulo y del resto del libro, necesitará Utilice el servidor de configuración Spring Cloud. Elegí esta solución por varias razones, incluyendo lo siguiente:

1. El servidor de configuración Spring Cloud es fácil de configurar y usar.
- 2 La configuración de Spring Cloud se integra estrechamente con Spring Boot. Puede leer literalmente todos los datos de configuración de su aplicación con unos pocos sencillos de usar. anotaciones.

- 3 El servidor de configuración Spring Cloud ofrece múltiples back-ends para almacenar datos de configuración. Si ya está utilizando herramientas como Eureka y Consul, puede conectarlas directamente al servidor de configuración de Spring Cloud.
- 4 De todas las soluciones en la tabla 3.1, el servidor de configuración Spring Cloud puede integrarse directamente con la plataforma de control de fuente Git. La integración de la configuración de Spring Cloud con Git elimina una dependencia adicional en sus soluciones y facilita el control de versiones de los datos de configuración de su aplicación.

Las otras herramientas (Etcd, Consul, Eureka) no ofrecen ningún tipo de versión nativa y, si quisieras eso, tendrías que crearlo tú mismo. Si tu tienda utiliza Git, el uso del servidor de configuración Spring Cloud es una opción atractiva.

Durante el resto de este capítulo, vas a

- 1 Configure un servidor de configuración de Spring Cloud y demuestre dos mecanismos diferentes para servir datos de configuración de aplicaciones: uno usando el sistema de archivos y otro usando un repositorio Git.
- 2 Continuar desarrollando el servicio de licencias para recuperar datos de una base de datos.
- 3 Conecte el servicio de configuración de Spring Cloud a su servicio de licencia para proporcionar datos de configuración de la aplicación.

### 3.2 Construyendo nuestro servidor de configuración Spring Cloud

El servidor de configuración de Spring Cloud es una aplicación basada en REST construida sobre Spring Boot. No viene como un servidor independiente. En su lugar, puede optar por incrustarlo en una aplicación Spring Boot existente o iniciar un nuevo proyecto Spring Boot con el servidor integrado.

Lo primero que debe hacer es configurar un nuevo directorio de proyecto llamado confsvr. Dentro del directorio confsvr, creará un nuevo archivo Maven que se usará para extraer los archivos JAR necesarios para iniciar su servidor de configuración de Spring Cloud. En lugar de recorrer todo el archivo Maven, enumeraré las partes clave en la siguiente lista.

#### Listado 3.1 Configuración de pom.xml para el servidor de configuración Spring Cloud

```
<?xml versión="1.0" codificación="UTF-8"?> <proyecto
xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd"> <modelVersion>4.0.0</
modelVersion>

<groupId>com.thinktmechanix</groupId>
<artifactId>servidor de configuración</artifactId> <versión>0.0.1-
SNAPSHOT</versión> <packaging>jar</
packaging>

<nombre>Config Server</nombre>
<descripción>Proyecto de demostración de Config Server</descripción>
```

```

<parent>
    <groupId>org.springframework.boot</groupId> <artifactId>spring-
    boot-starter-parent</artifactId> <version>1.4.4.RELEASE</version> </parent>
    <dependencyManagement> <dependencias >
        <dependencia>
            <groupId>org.springframework.cloud</groupId> <artifactId>spring-
            cloud-dependencies</artifactId> <versión>Camden.SR5</versión>
            <típo>pom</típo> <alcance>importar</
            alcance> </dependencia>
        </dependencias> </dependencyManagement>
        <propiedades>
            <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding> <start-
            class>com.thinktmechanix.confsvr. ConfigServerApplication
            </start-class> <java.version>1.8</java.version>
            <docker.image.name>johnncarnell/tmx-confsvr</
            docker.image.name> <docker.image.tag>capítulo3</ docker.image.tag> </properties>
        </dependencias>
        <dependencia>
            <groupId>org.springframework.cloud</groupId> <artifactId>spring-
            cloud-starter-config</artifactId>
        </dependencia>
        <dependencia>
            <groupId>org.springframework.cloud</groupId> <artifactId>spring-
            cloud-config-server</artifactId>
        </dependencia> </
        dependencias>
    <!--Configuración de compilación de Docker no mostrada --> </
    project>

```

The diagram illustrates the Maven configuration code with several annotations:

- An annotation "La versión de Spring Boot que usarás" points to the parent dependency's version attribute: <version>1.4.4.RELEASE</version>.
- An annotation "La versión de Spring Cloud que se utilizará." points to the artifactId of the spring-cloud-dependencies dependency: <artifactId>spring-cloud-dependencies</artifactId>.
- An annotation "La clase de arranque que se utilizará para el servidor de configuración" points to the start-class property: <start-class>com.thinktmechanix.confsvr. ConfigServerApplication</start-class>.
- An annotation "Los proyectos de Spring Cloud que utilizará en este servicio específico" points to the two dependencyManagement entries: <dependencyManagement> <dependencias > <dependencia> and <dependencia>.

En el archivo Maven de este listado anterior, comienza declarando la versión de Spring Boot que usará para su microservicio (versión 1.4.4). La siguiente parte importante de la definición de Maven es la lista de materiales ( BOM ) principal de Spring Cloud Configuration que va a utilizar. Spring Cloud es una colección masiva de proyectos independientes, todos ellos con sus propios lanzamientos. Esta lista de materiales principal contiene todas las bibliotecas y dependencias de terceros que se utilizan en el proyecto de nube y los números de versión de los proyectos individuales que componen esa versión. En este ejemplo, estás utilizando la versión Camden.SR5 de Spring Cloud. Al utilizar la definición de BOM , puede garantizar que está utilizando versiones compatibles de los subproyectos en Spring Cloud. También significa que no es necesario declarar núm-

subdependencias. El resto del ejemplo del listado 3.1 trata de declarar las dependencias específicas de Spring Cloud que utilizará en el servicio. La primera dependencia es la dependencia `spring-cloud-starter-config` que utilizan todos los proyectos de Spring Cloud. La segunda dependencia es el proyecto inicial `spring-cloud-config-server`. Contiene las bibliotecas principales para `spring-cloud-config-server`.

#### Vamos, súbete al tren, el tren de lanzamiento Spring

Cloud utiliza un mecanismo no tradicional para etiquetar proyectos Maven. Spring Cloud es una colección de subproyectos independientes. El equipo de Spring Cloud realiza sus lanzamientos a través de lo que ellos llaman el "tren de lanzamiento". Todos los subproyectos que componen Spring Cloud se empaquetan en una lista de materiales (BOM) de Maven y se publican en su conjunto. El equipo de Spring Cloud ha estado utilizando el nombre de las paradas del metro de Londres como nombre de sus lanzamientos, y cada lanzamiento importante incremental proporciona una parada del metro de Londres que tiene la siguiente letra más alta. Ha habido tres lanzamientos: Angel, Brixton y Camden. Camden es, con diferencia, la versión más reciente, pero todavía tiene varias ramas candidatas a versión para los subproyectos que contiene.

Una cosa a tener en cuenta es que Spring Boot se lanza independientemente del tren de lanzamiento de Spring Cloud. Por lo tanto, diferentes versiones de Spring Boot son incompatibles con diferentes versiones de Spring Cloud. Puede ver las dependencias de versiones entre Spring Boot y Spring Cloud, junto con las diferentes versiones de subproyectos contenidas en el tren de lanzamiento, consultando el sitio web de Spring Cloud (<http://projects.spring.io/spring-cloud/> ).

Aún necesita configurar un archivo más para que el servidor de configuración principal esté en funcionamiento. Este archivo es su archivo `application.yml` y se encuentra en el directorio `confsvr/src/main/resources`. El archivo `application.yml` le indicará a su servicio de configuración de Spring Cloud qué puerto escuchar y dónde ubicar el back-end que entregará los datos de configuración.

Está casi listo para abrir su servicio de configuración de Spring Cloud. Debe apuntar el servidor a un repositorio back-end que contendrá sus datos de configuración. Para este capítulo, utilizará el servicio de licencias que comenzó a crear en el capítulo 2 como ejemplo de cómo usar Spring Cloud Config. Para simplificar las cosas, configurará los datos de configuración de la aplicación para tres entornos: un entorno predeterminado para cuando ejecuta el servicio localmente, un entorno de desarrollo y un entorno de producción.

En la configuración de Spring Cloud, todo funciona según una jerarquía. La configuración de su aplicación está representada por el nombre de la aplicación y luego un archivo de propiedades para cada entorno para el que desea tener información de configuración. En cada uno de estos entornos, configurará dos propiedades de configuración:

Una propiedad de ejemplo que usará directamente su servicio de licencias.

La configuración de la base de datos de Postgres que usará para almacenar las licencias.  
datos de servicio

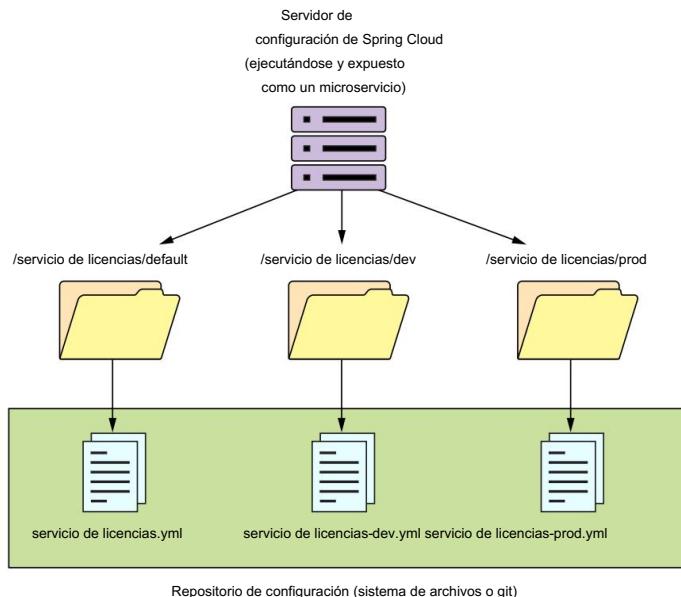


Figura 3.3 La configuración de Spring Cloud expone propiedades específicas del entorno como puntos finales basados en HTTP.

La Figura 3.3 ilustra cómo configurará y utilizará el servicio de configuración de Spring Cloud.

Una cosa a tener en cuenta es que a medida que desarrolle su servicio de configuración, será otro microservicio que se ejecuta en su entorno. Una vez configurado, se puede acceder al contenido del servicio a través de un punto final REST basado en http .

La convención de nomenclatura para los archivos de configuración de la aplicación es appname- env.yml. Como puede ver en el diagrama de la figura 3.3, los nombres de los entornos se traducen directamente a las URL a las que se accederá para explorar la información de configuración.

Más adelante, cuando inicie el ejemplo del microservicio de licencias, el entorno que desea ejecutar el servicio está especificado por el perfil Spring Boot que pasa en el Inicio del servicio de línea de comandos. Si no se pasa un perfil en la línea de comando, Spring El arranque siempre utilizará de forma predeterminada los datos de configuración contenidos en el archivo application.yml. empaquetado con la aplicación.

A continuación se muestra un ejemplo de algunos de los datos de configuración de la aplicación que proporcionará para el servicio de licencia. Estos son los datos que estarán contenidos en el archivo confsvr/src/ archivo principal/resources/config/licensingservice/licensingservice.yml al que se hizo referencia en la figura 3.3. Aquí tenéis parte del contenido de este archivo:

```

tracer.property: "YO SOY EL PREDETERMINADO"
spring.jpa.database: "POSTGRESQL"
spring.datasource.platform: "postgres"
spring.jpa.show-sql: "verdadero"
spring.database.driverClassName: "org.postgresql.Driver"
spring.datasource.url: "jdbc:postgresql://base de datos:5432/eagle_eye_local"
spring.datasource.nombre de usuario: "postgres"

```

```
spring.datasource.contraseña: "p0stgr@s"
spring.datasource.testWhileidle: "verdadero"
spring.datasource.validationQuery: "SELECCIONAR 1"
spring.jpa.properties.hibernate.dialect:
    "org.hibernate.dialect.PostgreSQLDialect"
```

### Piensa antes de implementar

No recomiendo el uso de una solución basada en un sistema de archivos para aplicaciones en la nube de tamaño mediano a grande. Usar el enfoque del sistema de archivos significa que necesita implementar un archivo compartido punto de montaje para todos los servidores de configuración de la nube que quieran acceder a la aplicación datos de configuración. Es posible configurar servidores de sistemas de archivos compartidos en la nube, pero le impone a usted la responsabilidad de mantener este entorno.

Estoy mostrando el enfoque del sistema de archivos como el ejemplo más fácil de usar al obtener sus pies mojados con el servidor de configuración Spring Cloud. En una sección posterior, mostraré cómo configure el servidor de configuración de Spring Cloud para utilizar un proveedor de Git basado en la nube como Bitbucket o GitHub para almacenar la configuración de su aplicación.

### 3.2.1 Configurar la clase Spring Cloud Config Bootstrap

Cada servicio de Spring Cloud cubierto en este libro siempre necesita una clase de arranque que utilizarse para iniciar el servicio. Esta clase de arranque contendrá dos cosas: un Java método main() que actúa como punto de entrada para que comience el Servicio, y un conjunto de Anotaciones de Spring Cloud que le dicen al servicio inicial qué tipo de Spring Cloud comportamientos que va a lanzar para el servicio.

La siguiente lista muestra el archivo confsvr/src/main/java/com/thinkde  
clase mechanix/confsvr/Application.java que se utiliza como clase de arranque para su servicio de configuración.

#### Listado 3.2 La clase de arranque para su servidor Spring Cloud Config

```
paquete com.thinkmechanix.confsvr;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication
@EnableConfigServer clase
pública ConfigServerApplication {
    público estático vacío principal (String [] argumentos) {
        SpringApplication.run(ConfigServerApplication.clase, argumentos);
    }
}
```

El método principal inicia el servicio e inicia el contenedor Spring.

Su servicio Spring Cloud Config es un Aplicación Spring Boot, para que la marques con @SpringBootApplication.

La anotación @EnableConfigServer permite el servicio como un servicio Spring Cloud Config.

A continuación, configurará su servidor de configuración de Spring Cloud con nuestro ejemplo más simple: el sistema de archivos.

### 3.2.2 Uso del servidor de configuración Spring Cloud con el sistema de archivos

El servidor de configuración de Spring Cloud utiliza una entrada en confsvr/src/main/archivo resources/application.yml para apuntar al repositorio que contendrá la aplicación datos de configuración. Configurar un repositorio basado en un sistema de archivos es la forma más sencilla de lograr esto.

Para hacer esto, agregue la siguiente información al archivo application-tion.yml del servidor de configuración. La siguiente lista muestra el contenido del archivo application.yml de su servidor de configuración Spring Cloud.

Listado 3.3 Archivo application.yml de configuración de Spring Cloud

```
servidor:
  puerto: 8888
  primavera:
    perfiles:
      activo: nativo
    nube:
      configuración:
        servidor:
          nativo:
            ubicaciones de búsqueda: file:///Users/johncarnell1/book/
              aplicaciones_nube_nativa/ch4-config-managment/confsvr/src/main/
                recursos/config/servicio de licencias
```

En el archivo de configuración de este listado, comenzó diciéndole al servidor de configuración qué número de puerto debe escuchar para todas las solicitudes de configuración:

```
servidor:
  puerto: 8888
```

Debido a que estás utilizando el sistema de archivos para almacenar información de configuración de la aplicación, debe indicarle al servidor de configuración de Spring Cloud que se ejecute con el perfil "nativo":

```
perfiles:
  activo: nativo
```

La última parte del archivo application.yml proporciona la configuración de Spring Cloud con el directorio donde residen los datos de la aplicación:

```
servidor:
  nativo:
    Ubicaciones de búsqueda: file:///Users/johncarnell1/book/spmia_code/chapter3-
      código/confsvr/src/main/resources/config
```

El parámetro importante en la entrada de configuración es searchLocations atributo. Este atributo proporciona una lista separada por comas de los directorios para cada

aplicación que tendrá propiedades administradas por el servidor de configuración. En el ejemplo anterior, solo tienes configurado el servicio de licencias.

**NOTA** Tenga en cuenta que si utiliza la versión del sistema de archivos local de Spring Cloud Configuración, deberá modificar `spring.cloud.config.server`

Atributo `.native.searchLocations` para reflejar la ruta de su archivo local cuando ejecutando su código localmente.

Ahora ya tiene suficiente trabajo hecho para iniciar el servidor de configuración. Adelante y empieza el servidor de configuración usando el comando `mvn spring-boot:run`. El servidor

Ahora debería aparecer la pantalla de presentación de Spring Boot en la línea de comando. Si usted Apunte su navegador a `http://localhost:8888/licensingService/default`, verá

La carga útil JSON se devuelve con todas las propiedades contenidas en el archivo `licensingService.yml`. La Figura 3.4 muestra los resultados de llamar a este punto final.

```
{
  "name": "licensingService",
  "profiles": [
    "default"
  ],
  "label": "master",
  "version": "8b20dd9432ef9ef08216a5775859afb24a5e7d43",
  "propertySources": [
    {
      "name": "https://github.com/carnellj/config-repo/licensingService/licensingService.yml",
      "source": {
        "example.property": "I AM IN THE DEFAULT",
        "spring.jpa.database": "POSTGRESQL",
        "spring.datasource.platform": "postgres",
        "spring.jpa.show-sql": "true",
        "spring.database.driverClassName": "org.postgresql.Driver",
        "spring.datasource.url": "jdbc:postgresql://database:5432/eagle_eye_local",
        "spring.datasource.username": "postgres",
        "spring.datasource.password": "{cipher}4788dfe1ccbe6485934aec2ffeddb06163ea3d616df5fd75be96aadd4df1da91",
        "spring.datasource.testWhileIdle": "true",
        "spring.datasource.validationQuery": "SELECT 1",
        "spring.jpa.properties.hibernate.dialect": "org.hibernate.dialect.PostgreSQLDialect",
        "redis.server": "redis",
        "redis.port": "6379",
        "signing.key": "345345fsdfsfs5345"
      }
    }
  ]
}
```



El archivo fuente que contiene las propiedades en el repositorio de configuración.

Figura 3.4 Recuperación de información de configuración predeterminada para el servicio de licencias

Si desea ver la información de configuración del servicio de licencias basado en desarrolladores entorno, presione GET `http://localhost:8888/licensingService/dev` punto final. La Figura 3.5 muestra el resultado de llamar a este punto final.

Si miras de cerca, verás que cuando llegues al punto final de desarrollo, regresarás recuperar las propiedades de configuración predeterminadas para el servicio de licencias y el desarrollador configuración del servicio de licencias. La razón por la cual la configuración de Spring Cloud devuelve ambos conjuntos de información de configuración es que el marco Spring implementa un Mecanismo jerárquico para resolver propiedades. Cuando Spring Framework lo hace

```

"propertySources": [
  {
    "name": "https://github.com/carnellj/config-repo/licensingservice/licensingservice-dev.yml",
    "source": {
      "spring.jpa.database": "POSTGRESQL",
      "spring.datasource.platform": "postgres",
      "spring.jpa.show-sql": "true",
      "spring.database.driverClassName": "org.postgresql.Driver",
      "spring.datasource.url": "jdbc:postgresql://database:5432/eagle_eye_dev",
      "spring.datasource.username": "postgres_dev",
      "spring.datasource.password": "{cipher}d495ce8603af958b2526967648aa9620b7e834c4eaff66014aa805450736e119",
      "spring.datasource.testWhileIdle": "true",
      "spring.datasource.validationQuery": "SELECT 1",
      "spring.jpa.properties.hibernate.dialect": "org.hibernate.dialect.PostgreSQLDialect",
      "redis.server": "redis",
      "redis.port": "6379",
      "signing.key": "345345fsdfsfs5345"
    }
  },
  {
    "name": "https://github.com/carnellj/config-repo/licensingservice/licensingservice.yml",
    "source": {
      "example.property": "I AM IN THE DEFAULT",
      "spring.jpa.database": "POSTGRESQL",
      "spring.datasource.platform": "postgres",
      "spring.jpa.show-sql": "true"
    }
  }
]
  
```

Cuando solicita un perfil específico del entorno, se devuelven tanto el perfil solicitado como el perfil predeterminado.

Figura 3.5 Recuperación de información de configuración para el servicio de licencias utilizando el perfil de desarrollo

resolución de propiedad, siempre buscará primero la propiedad en las propiedades predeterminadas y luego anule el valor predeterminado con un valor específico del entorno si hay uno presente.

En términos concretos, si define una propiedad en el archivo `licensingservice.yml` y no lo defina en ninguno de los otros archivos de configuración del entorno (por ejemplo, el `licensingservice-dev.yml`), el marco Spring utilizará el valor predeterminado.

**NOTA** Este no es el comportamiento que verá al llamar directamente a Spring Cloud configuración del punto final REST . El punto final REST devolverá todos los valores de configuración tanto para el valor predeterminado como para el valor específico del entorno que fue llamado.

Veamos cómo puede conectar el servidor de configuración de Spring Cloud a su microservicio de licencias.

### 3.3 Integración de Spring Cloud Config con un cliente Spring Boot

En el capítulo anterior, construyó un esqueleto simple de su servicio de licencias que no nada más que devolver un objeto Java codificado que representa una licencia única registro de su base de datos. En el siguiente ejemplo, desarrollará el servicio de licencias. y hable con una base de datos de Postgres que contenga sus datos de licencia.

Te comunicarás con la base de datos usando Spring Data y mapearás tu datos de la tabla de licencias a un POJO que contiene los datos. Tu conexión a la base de datos

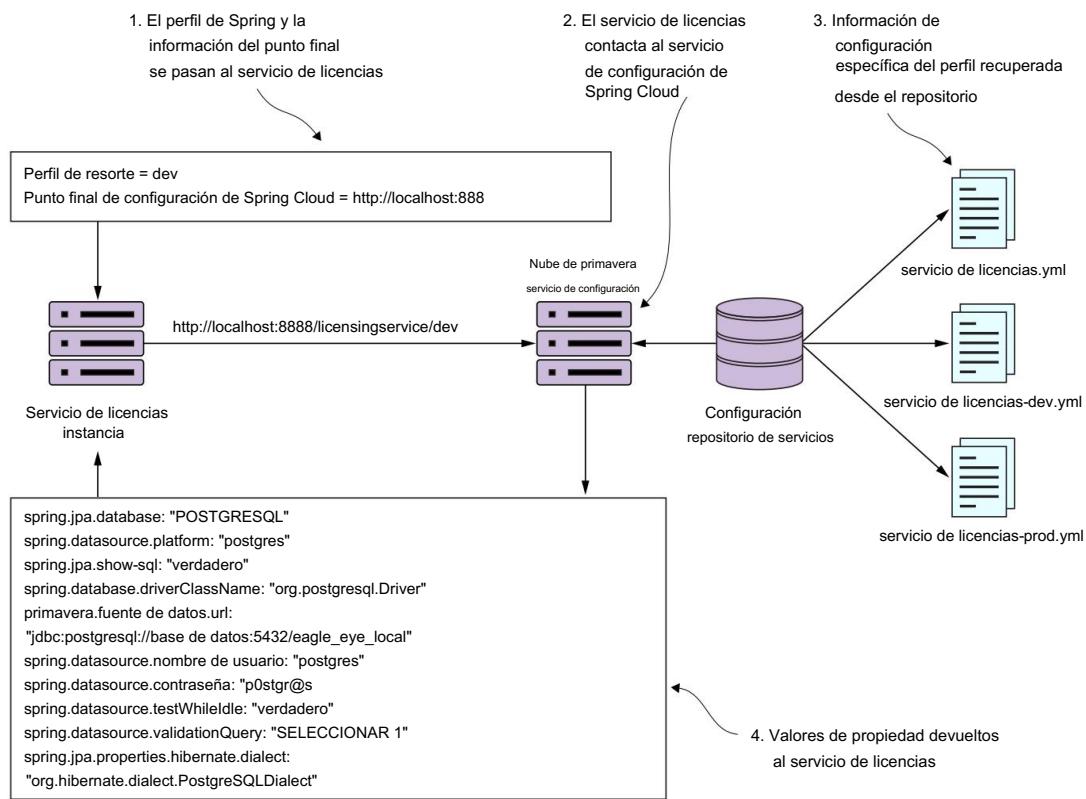


Figura 3.6 Recuperar información de configuración utilizando el perfil de desarrollo

y se leerá una propiedad simple desde el servidor de configuración de Spring Cloud.

La Figura 3.6 muestra lo que sucederá entre el servicio de licencias y Spring.

#### Servicio de configuración en la nube.

Cuando el servicio de licencias se inicia por primera vez, lo pasará a través de la línea de comando dos piezas de información: el perfil de Spring y el punto final el servicio de licencias debe utilizar para comunicarse con el servicio de configuración de Spring Cloud. La primavera El valor del perfil se asigna al entorno de las propiedades que se recuperan para Spring. servicio. Cuando el servicio de licencias se inicia por primera vez, se comunicará con el servicio Spring Cloud Config a través de un punto final creado a partir del perfil de Spring que se le pasó. La primavera El servicio Cloud Config luego utilizará el repositorio de configuración back-end configurado (filesys-tem, Git, Consul, Eureka) para recuperar la información de configuración específica del Valor del perfil de Spring pasado en el URI. Los valores de propiedad apropiados son entonces devuelto al servicio de licencias. El marco Spring Boot luego inyectará estos valores en las partes apropiadas de la aplicación.

### 3.3.1 Configuración del servicio de licencias del servidor Spring Cloud Config dependencias

Cambiemos nuestro enfoque del servidor de configuración al servicio de licencias. La primera Lo que debe hacer es agregar un par de entradas más al archivo Maven en su servicio de licencias. Las entradas que deben agregarse se muestran en la siguiente lista.

Listado 3.4 Dependencias adicionales de Maven que necesita el servicio de licencias

```
<dependencia>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependencia>

<dependencia>
    <groupId>postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>9.1-901.jdbc4</version>
</dependencia>

<dependencia>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>cliente-config-nube-spring</artifactId>
</dependencia>
```

La primera y segunda dependencias, `spring-boot-starter-data-jpa` y `PostgreSQL`, importan Spring Data Java Persistence API (JPA) y los controladores Postgres JDBC. La última dependencia, `spring-cloud-config-client`, contiene todos los clases necesarias para interactuar con el servidor de configuración de Spring Cloud.

### 3.3.2 Configurar el servicio de licencias para usar Spring Cloud Config

Una vez definidas las dependencias de Maven, debe indicarle al servicio de licencias dónde comunicarse con el servidor de configuración de Spring Cloud. En un servicio Spring Boot que utiliza Spring Cloud Config, la información de configuración se puede establecer en uno de dos Archivos de configuración: `bootstrap.yml` y `application.yml`.

El archivo `bootstrap.yml` lee las propiedades de la aplicación antes de utilizar cualquier otra información de configuración. En general, el archivo `bootstrap.yml` contiene el nombre de la aplicación para el servicio, el perfil de la aplicación y el URI para conectarse a un Spring.

Servidor de configuración en la nube. Cualquier otra información de configuración que desee mantener local al servicio (y no almacenado en Spring Cloud Config) se puede configurar localmente en los servicios en el archivo `application.yml`. Por lo general, la información que almacena en el archivo `application.yml` son datos de configuración que quizás desee tener disponibles para un servicio incluso si el servicio Spring Cloud Config no está disponible. Tanto el archivo `boot-strap.yml` como el `application.yml` se almacenan en un proyecto `src/main/` directorio de recursos .

Para que el servicio de licencias se comunique con su servicio Spring Cloud Config, debe agregar un archivo licensing-service/src/main/resources/bootstrap.yml y configurar tres propiedades: spring.application.name, spring.profiles.active y spring.cloud.config.uri.

El archivo bootstrap.yml de servicios de licencia se muestra en la siguiente lista.

#### Listado 3.5 Configuración de los servicios de licencia bootstrap.yml

primavera:		Especifique el nombre del servicio de licencia para que el cliente Spring Cloud Config sepa qué servicio se está buscando.
solicitud:	nombre: perfiles de servicio de	
licencias:	activo: por defecto	Especifique el perfil predeterminado que debe ejecutar el servicio. Mapas de perfil al entorno.
nube:	configuración: URL: http://localhost:8888	Especifique la ubicación del servidor Spring Cloud Config.

**NOTA** Las aplicaciones Spring Boot admiten dos mecanismos para definir una propiedad: YAML (otro lenguaje de marcado más) y un "." propiedad separada nombre. Elegimos YAML (otro lenguaje de marcado) como medio para configurando nuestra aplicación. El formato jerárquico de la propiedad YAML. los valores se asignan directamente a spring.application.name, spring.profiles nombres .active y spring.cloud.config.uri .

spring.application.name es el nombre de su aplicación (por ejemplo, licensingservice) y debe asignarse directamente al nombre del directorio dentro de su Servidor de configuración de Spring Cloud. Para el servicio de licencias, desea un directorio en el servidor de configuración de Spring Cloud llamado licensingservice.

La segunda propiedad, spring.profiles.active, se utiliza para indicarle a Spring Boot con qué perfil debe ejecutarse la aplicación. Un perfil es un mecanismo para diferenciar el datos de configuración consumidos por la aplicación Spring Boot. Para el perfil del servicio de licencia, respaldará el entorno al que el servicio se asignará directamente en su entorno de configuración en la nube. Por ejemplo, al pasar dev como nuestro perfil, el servidor de configuración de Spring Cloud utilizará las propiedades de desarrollo. Si configura un perfil, el El servicio de licencias utilizará el perfil predeterminado.

La tercera y última propiedad, spring.cloud.config.uri, es la ubicación donde el servicio de licencias debe buscar el servidor de configuración de Spring Cloud punto final. De forma predeterminada, el servicio de licencias buscará el servidor de configuración en http://localhost:8888. Más adelante en el capítulo verás cómo anular las diferentes propiedades definidas en los archivos bootstrap.yml y application.yml en la aplicación puesta en marcha. Esto le permitirá indicarle al microservicio de licencias qué entorno debería estar corriendo.

Ahora, si abre el servicio de configuración de Spring Cloud, con la base de datos Postgres correspondiente ejecutándose en su máquina local, puede iniciar el proceso de licencia.

servicio utilizando su perfil predeterminado. Esto se hace cambiando a los servicios de licencia. directorio y emitiendo los siguientes comandos:

```
mvn spring-boot: ejecutar
```

Al ejecutar este comando sin ninguna propiedad establecida, el servidor de licencias intentará conectarse automáticamente al servidor de configuración de Spring Cloud utilizando el punto final (<http://localhost:8888>) y el perfil activo (predeterminado) definido en el archivo `bootstrap.yml` del servicio de licencias.

Si desea anular estos valores predeterminados y apuntar a otro entorno, debe Puede hacer esto compilando el proyecto de licensingservice en un JAR y luego ejecutando el JAR con una anulación de propiedad del sistema -D . La siguiente llamada de línea de comando demuestra cómo iniciar el servicio de licencias con un perfil no predeterminado:

```
java -Dspring.cloud.config.uri=http://localhost:8888  
-Dspring.profiles.active=desarrollador  
-jar target/servicio-de-licencias-0.0.1-SNAPSHOT.jar
```

Con la línea de comando anterior, estás anulando los dos parámetros:

`spring.cloud.config.uri` y `spring.profiles.active`. Con el `-Dspring.cloud.config.uri=http://localhost:8888` propiedad del sistema, estás apuntando a un servidor de configuración que se ejecuta fuera de su caja local.

**NOTA** Si intenta ejecutar el servicio de licencias descargado de GitHub repositorio (<https://github.com/carnellj/spmia-chapter3>) desde tu escritorio usando el comando Java anterior, fallará porque no tienes un escritorio El servidor Postgres se está ejecutando y el código fuente en el repositorio de GitHub se está utilizando cifrado en el servidor de configuración. Cubriremos el uso de cifrado más adelante en el capítulo. El ejemplo anterior demuestra cómo anular las propiedades de Spring mediante la línea de comando.

Con la propiedad del sistema `-Dspring.profiles.active=dev`, le estás diciendo al Servicio de licencias para utilizar el perfil de desarrollo (leído del servidor de configuración) para conectarse a una instancia de desarrollo de una base de datos.

### Utilice variables de entorno para pasar información de inicio

En los ejemplos, está codificando los valores para pasarlo a los valores del parámetro -D. En la nube, la mayoría de los datos de configuración de la aplicación que necesita estarán en su configuración servidor. Sin embargo, para la información que necesita para iniciar su servicio (como los datos para el servidor de configuración), iniciaría la instancia de VM o el contenedor Docker y pasar una variable de entorno.

Todos los ejemplos de código de cada capítulo se pueden ejecutar completamente desde contenedores Docker. Con Docker, usted simula diferentes entornos a través de archivos Docker-compose específicos del entorno que organizan el inicio de todos sus servicios. Los valores específicos del entorno que necesitan los contenedores se pasan como entorno

(continuado)

variables al contenedor. Por ejemplo, para iniciar su servicio de licencias en un entorno de desarrollo, el archivo docker/dev/docker-compose.yml contiene la siguiente entrada para el servicio de licencias:

servicio de licencias:

imagen: ch3-thinkmechanix/servicio-de-licencias

puertos:

- "8080:8080"

ambiente:

PERFIL: "desarrollador"

Especifica el inicio de las variables de entorno para el contenedor del servicio de licencias.

CONFIGSERVER\_URI: http://configserver:8888

CONFIGSERVER\_PORT: "8888"

SERVIDOR\_BASE DE DATOS: "5432"

La variable de entorno PERFIL se pasa al servicio Spring Boot línea de comando y le dice a Spring Arranque qué perfil se debe ejecutar.

El punto final del servicio de configuración.

La entrada de entorno en el archivo contiene los valores de dos variables PERFIL, que es el perfil de Spring Boot bajo el cual se ejecutará el servicio de licencias. El CONFIGSERVER\_URI se pasa a su servicio de licencias y define Spring Cloud Instancia del servidor de configuración desde la que el servicio leerá sus datos de configuración.

En los scripts de inicio que ejecuta el contenedor, luego pasa estos entornos variables como parámetros -D para nuestro JVMS iniciando la aplicación. En cada proyecto, usted hornea un contenedor Docker, y ese contenedor Docker usa un script de inicio que inicia el software en el contenedor. Para el servicio de licencias, el script de inicio que obtiene incorporado en el contenedor se puede encontrar en licensing-service/src/main/docker/run.sh. En el script run.sh, la siguiente entrada inicia la JVM del servicio de licencias:

```
echo ****
echo "Iniciando el servidor de licencias con el servicio de configuración:
$CONFIGSERVER_URI";
echo ****
java -Dspring.cloud.config.uri=$CONFIGSERVER_URI
-Dspring.profiles.active=$PERFIL -jar /usr/local/licensingservice/
    servicio-de-licencias-0.0.1-SNAPSHOT.jar
```

Porque mejora todos sus servicios con capacidades de introspección a través de Spring Boot Actuador, puede confirmar el entorno contra el que se está ejecutando presionando `http://localhost:8080/env`. El punto final /env proporcionará una lista completa de la información de configuración del servicio, incluidas las propiedades y los puntos finales del servicio. ha arrancado, como se muestra en la figura 3.7.

Lo clave a tener en cuenta en la figura 3.7 es que el perfil activo para el servicio de licencia es dev. Al inspeccionar el JSON devuelto, también puede ver que la base de datos de Postgres que se devuelve es un URI de desarrollo de `jdbc:postgresql://database:5432/eagle_eye_dev`.

```
{  
    "profiles": [  
        "default"  
    ],  
    "server.ports": {  
        "local.server.port": 8080  
    },  
    "decrypted": {  
        "spring.datasource.password": "*****"  
    },  
    "configService:configClient": {  
        "config.client.version": "8907411ed638d7a66e2ae4142f83671425f4113f"  
    },  
    "configService:https://github.com/carnellj/config-repo/licensingservice/licensingservice.yml": {  
        "example.property": "I AM IN THE DEFAULT",  
        "spring.jpa.database": "POSTGRESQL",  
        "spring.datasource.platform": "postgres",  
        "spring.jpa.show-sql": "true",  
        "spring.database.driverClassName": "org.postgresql.Driver",  
        "spring.datasource.url": "jdbc:postgresql://database:5432/eagle_eye_local",  
        "spring.datasource.username": "postgres",  
        "spring.datasource.password": "*****",  
        "spring.datasource.testWhileIdle": "true",  
        "spring.datasource.validationQuery": "SELECT 1",  
        "spring.jpa.properties.hibernate.dialect": "org.hibernate.dialect.PostgreSQLDialect",  
        "redis.server": "redis",  
        "redis.port": "6379",  
        "signing.key": "*****"  
    }.
```

Figura 3.7 La configuración que carga el servicio de licencias se puede verificar llamando al punto final /env .

#### Sobre exponer demasiada información

Cada organización tendrá reglas diferentes sobre cómo implementar la seguridad en torno a sus servicios. Muchas organizaciones creen que los servicios no deberían transmitir ningún información sobre ellos mismos y no permitirá que cosas como un punto final /env sean activos en un servicio porque creen (con razón) que esto proporcionará demasiada información a un posible hacker. Spring Boot proporciona una gran cantidad de capacidades sobre cómo para configurar qué información devuelven los puntos finales de Spring Actuators que son fuera del alcance de este libro. El excelente libro de Craig Walls, Spring Boot in Action, Cubre este tema en detalle y le recomiendo encarecidamente que revise su informe corporativo, políticas de seguridad y el libro de Walls para proporcionar el nivel correcto de detalle que desea exponer a través del actuador de resorte.

### 3.3.3 Cableado en una fuente de datos usando el servidor de configuración Spring Cloud

En este punto, la información de configuración de la base de datos se está inyectando directamente en su microservicio. Con la configuración de la base de datos establecida, configurar su microservicio de licencia se convierte en un ejercicio de uso de componentes estándar de Spring para construir y recuperar los datos de la base de datos de Postgres. El servicio de licencias ha sido

refactorizado en diferentes clases y cada clase tiene responsabilidades separadas. Estos Las clases se muestran en la tabla 3.2.

Tabla 3.2 Clases y ubicaciones de servicios de licencia

Nombre de la clase	Ubicación
Licencia	servicio-de-licencias/src/main/java/com/thinkmechanix/licenses/model
Repositorio de licencias	servicio-de-licencias/src/main/java/com/thinkmechanix/licenses/repository
Servicio de licencia	servicio-de-licencias/src/main/java/com/thinkmechanix/licenses/services

La clase de licencia es la clase de modelo que contendrá los datos recuperados de su base de datos de licencias. El siguiente listado muestra el código de la clase Licencia .

Listado 3.6 El código modelo JPA para un registro de licencia único

```
paquete com.thinkmechanix.licenses.model;

importar javax.persistence.Column;
importar javax.persistence.Entity;
importar javax.persistence.Id;
importar javax.persistence.Table;

@Entity
@Table(nombre = "licencias")
de clase pública{
    @Identificación
    @Column(nombre = "license_id", nullable = false) cadena privada
    licensId;

    @Columna(nombre = "organization_id", anulable = falso)
    ID de organización de cadena privada;

    @Columna(nombre = "nombre_producto", anulable = falso)
    cadena privada nombre del producto;

    /*El resto del código se ha eliminado por motivos de concisión*/
}
```

Diagrama que explica las anotaciones JPA en la clase Licencia:

- @Entity**: le dice a Spring que esta es una clase JPA.
- @Table**: se asigna a la tabla de la base de datos.
- @Id**: marca este campo como clave principal.
- @Column**: asigna el campo a una tabla de base de datos específica.

La clase utiliza varias anotaciones de persistencia de Java (JPA) que ayudan a Spring Data framework asigna los datos de la tabla de licencias en la base de datos de Postgres a Java objeto. La anotación `@Entity` le permite a Spring saber que este POJO de Java será mapear objetos que contendrán datos. La anotación `@Table` le dice a Spring/JPA qué La tabla de la base de datos debe estar asignada. La anotación `@Id` identifica la clave principal para la base de datos. Finalmente, cada una de las columnas de la base de datos que se va a asignado a propiedades individuales está marcado con un atributo `@Column` .

El marco Spring Data y JPA proporciona sus métodos CRUD básicos para acceder a una base de datos. Si desea crear métodos más allá de eso, puede usar Spring Data Interfaz del repositorio y convenciones de nomenclatura básicas para crear esos métodos. Primavera

Al inicio, analizará el nombre de los métodos de la interfaz del Repositorio, los convertirá en una declaración SQL basada en los nombres y luego generará una clase de proxy dinámica bajo las sábanas para hacer el trabajo. El repositorio del servicio de licencias se muestra en la siguiente lista.

**Listado 3.7 La interfaz LicenseRepository define los métodos de consulta**

```
paquete com.thinkmechanix.licenses.repository;

importar com.thinktmechanix.licenses.model.License; importar
org.springframework.data.repository.CrudRepository; importar
org.springframework.stereotype.Repository;

importar java.util.List;

@Repository
interfaz pública LicenseRepository
extiende CrudRepository<Licencia,Cadena>
{
    Lista pública<Licencia> findByOrganizationId  (String
organizaciónId); Licencia pública
    findByOrganizationIdAndLicenseld  (String OrganizationId,String
Licenseld);
}
```

La interfaz del repositorio, LicenseRepository, está marcada con la anotación `@Repository` que le indica a Spring que debe tratar esta interfaz como un repositorio y generar un proxy dinámico para ella. Spring ofrece diferentes tipos de repositorios para el acceso a datos. Ha elegido utilizar la clase base Spring `CrudRepository` para ampliar su clase `LicenseRepository`. La clase base `CrudRepository` contiene métodos CRUD básicos. Además del método CRUD extendido desde `CrudRepository`, agregó dos métodos de consulta personalizados para recuperar datos de la tabla de licencias. El marco Spring Data separará el nombre de los métodos para crear una consulta para acceder a los datos subyacentes.

**NOTA** El marco Spring Data proporciona una capa de abstracción sobre varias plataformas de bases de datos y no se limita a bases de datos relacionales. También se admiten bases de datos NoSQL como MongoDB y Cassandra.

A diferencia de la encarnación anterior del servicio de licencias en el capítulo 2, ahora ha separado la lógica comercial y de acceso a datos para el servicio de licencias fuera del `LicenseController` y en una clase de Servicio independiente llamada `LicenseService`.

**Listado 3.8 Clase LicenseService utilizada para ejecutar comandos de base de datos**

```
paquete com.thinkmechanix.licenses.services;

importar com.thinkmechanix.licenses.config.ServiceConfig; importar
com.thinktmechanix.licenses.model.License; importar
com.thinktmechanix.licenses.repository.LicenseRepository; importar
org.springframework.beans.factory.annotation.Autowired; importar
org.springframework.stereotype.Service;
```

```

importar java.util.List; importar
java.util.UUID;

@Service
Servicio de licencia de clase pública {

    @autocableado
    Repositorio de licencias privado Repositorio de licencias;

    @autocableado
    Configuración de ServiceConfig;

    Licencia pública getLicense(String organizaciónId, String licenciald) {
        Licencia licencia = LicenseRepository.findByOrganizationIdAndLicensesId(
            ID de organización, ID de licencia);
        devolver licencia.withComment(config.getExampleProperty());
    }

    Lista pública<Licencia> getLicensesByOrg(String organizaciónId){
        devolver LicenseRepository.findByOrganizationId( OrganizationId );
    }

    public void saveLicense(Licencia de licencia){
        licencia.withId( UUID.randomUUID().toString());
        licenciaRepository.save(licencia);
    }
    /*El resto del código se eliminó por razones de concisión*/
}

```

Las clases de controlador, servicio y repositorio están conectadas entre sí mediante la anotación estándar Spring `@Autowired`.

### 3.3.4 Lectura directa de propiedades utilizando la anotación `@Value`

En la clase LicenseService de la sección anterior, es posible que hayas notado que estás configurando el valor de `licencia.withComment()` en el código `getLicense()` con un valor de la clase `config.getExampleProperty()`. El código al que se hace referencia se muestra aquí:

```

Licencia pública getLicense(String organizaciónId, String licenciald) {
    Licencia licencia = LicenseRepository.findByOrganizationIdAndLicensesId( OrganizationId, LicensesId);
    devolver
    licencia.withComment(config.getExampleProperty());
}

```

Si observa la clase `licensing-service/src/main/java/com/thinktmechanix/licenses/config/ServiceConfig.java`, verá una propiedad anotada con la anotación `@Value`. La siguiente lista muestra la anotación `@Value` que se utiliza.

#### Listado 3.9 ServiceConfig utilizado para centralizar las propiedades de la aplicación

```

paquete com.thinkmechanix.licenses.config;

importar org.springframework.beans.factory.annotation.Value; importar
org.springframework.stereotype.Component;

```

```

@Componente
clase pública ServiceConfig{

    @Value("${example.property}") cadena privada
    propiedad de ejemplo;

    cadena pública getExampleProperty(){
        devolver propiedad de ejemplo;
    }
}

```

Mientras que Spring Data inyecta "automáticamente" los datos de configuración de la base de datos en un objeto de conexión de base de datos, todas las demás propiedades deben inyectarse utilizando el `@Anotación de valor`. Con el ejemplo anterior, la anotación `@Value` extrae el `example.property` del servidor de configuración de Spring Cloud y lo inyecta en el atributo `example.property` en la clase `ServiceConfig`.

**SUGERENCIA** Si bien es posible inyectar valores de configuración directamente en las propiedades En clases individuales, me ha resultado útil centralizar toda la configuración. información en una única clase de configuración y luego inyectar la clase de configuración donde sea necesaria.

### 3.3.5 Uso del servidor de configuración Spring Cloud con Git

Como se mencionó anteriormente, usar un sistema de archivos como repositorio backend para Spring Cloud El servidor de configuración puede resultar poco práctico para una aplicación basada en la nube porque El equipo de desarrollo tiene que configurar y administrar un sistema de archivos compartido que está montado en todos. instancias del servidor de configuración de la nube.

El servidor de configuración Spring Cloud se integra con diferentes repositorios backend que se puede utilizar para alojar las propiedades de configuración de la aplicación. Uno que he usado con éxito es usar el servidor de configuración Spring Cloud con un repositorio de control de fuente Git.

Al utilizar Git, puede obtener todos los beneficios de poner sus propiedades de gestión de configuración bajo control de código fuente y proporcionar un mecanismo fácil de integrar. la implementación de los archivos de configuración de su propiedad en su compilación e implementación tubería.

Para usar Git, cambiarías la configuración del sistema de archivos en la configuración archivo `bootstrap.yml` del servicio con la siguiente configuración del listado.

Listado 3.10 Configuración de Spring Cloud `bootstrap.yml`

```

servidor:
  puerto: 8888
primavera:
nube:
  configuración:
    servidor:
      git:
        uri: https://github.com/carnellj/config-repo/

```

searchPaths: servicio de licencias, nombre de usuario del servicio de organización: contraseña de aplicaciones nativas en la nube: Offended

Le dice a Spring Cloud Config cuál es la ruta en Git para buscar archivos de configuración

Las tres piezas clave de configuración en el ejemplo anterior son las propiedades `spring.cloud.config.server`, `spring.cloud.config.server.git.uri` y `spring.cloud.config.server.git.searchPaths`. La propiedad `spring.cloud.config.server` le dice al servidor de configuración de Spring Cloud que use un repositorio backend no basado en un sistema de archivos. En el ejemplo anterior, se conectará al repositorio Git basado en la nube, GitHub.

Las propiedades `spring.cloud.config.server.git.uri` proporcionan la URL del repositorio al que se está conectando. Finalmente, la propiedad `spring.cloud.config.server.git.searchPaths` le dice al servidor Spring Cloud Config las rutas relativas en el repositorio Git que se deben buscar cuando aparece el servidor de configuración de la nube. Al igual que la versión del sistema de archivos de la configuración, el valor en el atributo `spring.cloud.config.server.git.seachPaths` será una lista separada por comas para cada servicio alojado por el servicio de configuración.

### 3.3.6 Actualizar sus propiedades usando el servidor de configuración Spring Cloud

Una de las primeras preguntas que surge de los equipos de desarrollo cuando quieren utilizar el servidor de configuración de Spring Cloud es cómo pueden actualizar dinámicamente sus aplicaciones cuando cambia una propiedad. El servidor de configuración de Spring Cloud siempre ofrecerá la última versión de una propiedad. Los cambios realizados en una propiedad a través de su repositorio subyacente estarán actualizados.

Sin embargo, las aplicaciones Spring Boot solo leerán sus propiedades en el momento del inicio, por lo que los cambios de propiedad realizados en el servidor de configuración de Spring Cloud no serán recogidos automáticamente por la aplicación Spring Boot. Spring Boot Actuator ofrece una anotación `@RefreshScope` que permitirá a un equipo de desarrollo acceder a un punto final `/refresh` que obligará a la aplicación Spring Boot a volver a leer su configuración de aplicación. La siguiente lista muestra la anotación `@RefreshScope` en acción.

#### Listado 3.11 La anotación `@RefreshScope`

```
paquete com.thinkmechanix.licenses;

importar org.springframework.boot.SpringApplication; importar
org.springframework.boot.autoconfigure.SpringBootApplication; importar
org.springframework.cloud.context.config.annotation.RefreshScope;

@SpringBootApplication
@RefreshScope
aplicación de clase pública { public
    static void main(String[] args)
        { SpringApplication.run(Application.class, args);
    }
}
```

Tenga en cuenta un par de cosas sobre la anotación @RefreshScope . Primero, la anotación solo recargará las propiedades Spring personalizadas que tenga en la configuración de su aplicación. Los elementos como la configuración de su base de datos que utiliza Spring Data no serán recargados por la anotación @RefreshScope . Para realizar la actualización, puede presionar el punto final <http://<yourserver>:8080/refresh> .

#### Sobre la actualización de microservicios

Al utilizar el servicio de configuración de Spring Cloud con microservicios, una cosa que debe considerar antes de cambiar las propiedades dinámicamente es que es posible que tenga varias instancias del mismo servicio ejecutándose y deberá actualizar todos esos servicios con sus nuevas configuraciones de la aplicación. Hay varias formas de abordar este problema:

El servicio de configuración de Spring Cloud ofrece un mecanismo basado en "push" llamado Spring Cloud Bus que permitirá que el servidor de configuración de Spring Cloud publique para todos los clientes que utilizan el servicio que se ha producido un cambio. La configuración de Spring Cloud requiere una pieza adicional de middleware en ejecución (RabbitMQ). Este es un medio extremadamente útil para detectar cambios, pero no todos los backends de configuración de Spring Cloud admiten el mecanismo "push" (es decir, el servidor Consul).

En el próximo capítulo utilizará Spring Service Discovery y Eureka para registrar todas las instancias de un servicio. Una técnica que he usado para manejar los eventos de actualización de la configuración de la aplicación es actualizar las propiedades de la aplicación en la configuración de Spring Cloud y luego escribir un script simple para consultar el motor de descubrimiento de servicios para encontrar todas las instancias de un servicio y llamar al punto final /refresh directamente.

Finalmente, puede reiniciar todos los servidores o contenedores para seleccionar la nueva propiedad. Este es un ejercicio trivial, especialmente si ejecuta sus servicios en un servicio de contenedor como Docker. Reiniciar los contenedores Docker literalmente toma unos segundos y obligará a volver a leer la configuración de la aplicación.

Recuerde, los servidores basados en la nube son efímeros. No tenga miedo de iniciar nuevas instancias de un servicio con su nueva configuración, dirigir el tráfico a los nuevos servicios y luego eliminar los antiguos.

## 3.4 Protección de información de configuración confidencial

De forma predeterminada, el servidor de configuración de Spring Cloud almacena todas las propiedades en texto sin formato dentro de los archivos de configuración de la aplicación. Esto incluye información confidencial, como credenciales de bases de datos.

Es una práctica extremadamente mala mantener las credenciales confidenciales almacenadas como texto sin formato en su repositorio de código fuente. Desafortunadamente, esto sucede con mucha más frecuencia de lo que cree. Spring Cloud Config le brinda la posibilidad de cifrar sus propiedades confidenciales fácilmente. Spring Cloud Config admite el uso de cifrado simétrico (secreto compartido) y asimétrico (clave pública/privada).

Veremos cómo configurar su servidor de configuración de Spring Cloud para usar cifrado con una clave simétrica. Para hacer esto necesitarás

- 1 Descargue e instale los archivos jar de Oracle JCE necesarios para el cifrado
- 2 Configure una clave de cifrado.
- 3 Cifrar y descifrar una propiedad.
- 4 Configurar microservicios para usar cifrado en el lado del cliente

#### 3.4.1 Descargue e instale los archivos jar de Oracle JCE necesarios para el cifrado

Para comenzar, debe descargar e instalar la extensión de criptografía Java (JCE) Unlimited Strength de Oracle. Esto no está disponible a través de Maven y debe descargarse de Oracle Corporation.<sup>1</sup> Una vez que haya descargado los archivos zip que contienen el JCE frascos, debes hacer lo siguiente:

- 1 Localice su directorio \$JAVA\_HOME/jre/lib/security .
- 2 Haga una copia de seguridad de los archivos local\_policy.jar y US\_export\_policy.jar en el directorio \$JAVA\_HOME/jre/lib/security a una ubicación diferente.
- 3 Descomprima el archivo zip JCE que descargó de Oracle.
- 4 Copie local\_policy.jar y US\_export\_policy.jar a su Directorio \$JAVA\_HOME/jre/lib/security .
- 5 Configure Spring Cloud Config para usar cifrado.

#### Automatizar el proceso de instalación de archivos JCE de Oracle

Seguí los pasos manuales que necesitas para instalar JCE en tu computadora portátil. Porque Usamos Docker para construir todos nuestros servicios como contenedores Docker. He programado la descarga e instalación de estos archivos JAR en el contenedor Docker Spring Cloud Config. El siguiente fragmento de script de shell de OS X muestra cómo automatizé esto usando el comando curl (<https://curl.haxx.se/>) herramienta de línea de comandos:

```
disco compacto /tmp/
curl -k-LO "http://download.oracle.com/otn-pub/java/jce/8/jce_policy-
8.zip"
-H 'Cookie: oraclelicense=aceptar-cookie-securebackup' && descomprimir jce_policy-8.zip

rm jce_policy-8.zip sí |cp -v /
tmp/UnlimitedJCEPolicyJDK8/*.jar /usr/lib/jvm/java-1.8-
openjdk/jre/lib/seguidad/
```

No voy a explicar todos los detalles, pero básicamente uso CURL para descargar los archivos zip JCE (tenga en cuenta el parámetro del encabezado Cookie pasado mediante el atributo -H en el comando curl) y luego descomprima los archivos y cópielos en el directorio /usr/lib/jvm/java-1.8-openjdk/jre/lib/security en mi contenedor Docker.

Si observa el archivo src/main/docker/Dockerfile en el código fuente de este capítulo, Puede ver un ejemplo de este script en acción.

<sup>1</sup> <http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html>. Esta URL podría estar sujeto a cambios. Una búsqueda rápida en Google de extensiones de criptografía Java siempre debería devolverte los valores correctos.

### 3.4.2 Configurar una clave de cifrado

Una vez que los archivos JAR estén en su lugar, deberá configurar una clave de cifrado simétrica. La clave de cifrado simétrico no es más que un secreto compartido que utiliza el cifrador para cifrar un valor y el descifrador para descifrar un valor. Con la nube de primavera servidor de configuración, la clave de cifrado simétrica es una cadena de caracteres que usted selecciona que se pasa al servicio a través de una variable de entorno del sistema operativo llamada ENCRYPT\_KEY. Para los propósitos de este libro, siempre configurará la variable de entorno ENCRYPT\_KEY como

```
exportar ENCRYPT_KEY=IMSYMMETRIC
```

Tenga en cuenta dos cosas con respecto a las claves simétricas:

- 1 Su clave simétrica debe tener 12 o más caracteres e idealmente debe ser una clave aleatoria. conjunto de caracteres dom.
- 2 No pierdas tu clave simétrica. Una vez que haya cifrado algo con su clave cifrada, no puede descifrarla.

#### Administrar claves de cifrado

Para los propósitos de este libro, hice dos cosas que normalmente no recomendaría en una implementación de producción:

Configuré la clave de cifrado para que sea una frase. Quería mantener la clave simple para que Podría recordarlo y encajaría muy bien al leer el texto. En un mundo real implementación, usaría una clave de cifrado separada para cada entorno en el que estaba implementando y usaría caracteres aleatorios como mi clave.

He codificado la variable de entorno ENCRYPT\_KEY directamente en Docker archivos utilizados dentro del libro. Hice esto para que usted, como lector, pudiera descargar los archivos e iniciarlos sin tener que acordarse de configurar un entorno variable. En un entorno de ejecución real, haría referencia a ENCRYPT\_KEY como una variable de entorno del sistema operativo dentro de mi Dockerfile. ser consciente de esto y no codifique su clave de cifrado dentro de sus Dockerfiles. Recuerde, se supone que sus Dockerfiles deben mantenerse bajo control de código fuente.

### 3.4.3 Cifrar y descifrar una propiedad

Ahora está listo para comenzar a cifrar propiedades para usarlas en Spring Cloud Config. Cifrárá la contraseña de la base de datos de Postgres de los servicios de licencia que ha estado utilizando para acceder a los datos de EagleEye. Esta propiedad, llamada spring.datasource.password, actualmente está configurada como texto sin formato con el valor p0stgr@s.

Cuando inicia su instancia de Spring Cloud Config, Spring Cloud Config detecta que la variable de entorno ENCRYPT\_KEY está configurada y agrega automáticamente dos nuevos puntos finales (/encrypt y /decrypt) al servicio Spring Cloud Config. Usarás el punto final /encrypt para cifrar el valor p0stgr@s .

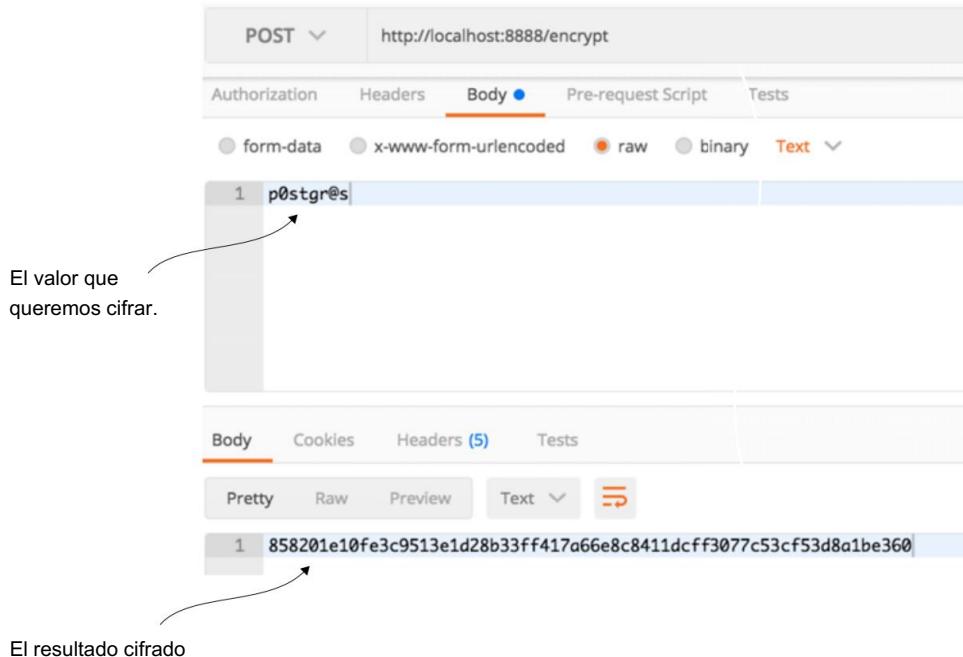


Figura 3.8 Usando el punto final /encrypt puede cifrar valores.

La Figura 3.8 muestra cómo cifrar el valor p0stgr@s utilizando el punto final /encrypt y CARTERO. Tenga en cuenta que cada vez que llame a los puntos finales /encrypt o /decrypt , debe asegurarse de realizar una POST en estos puntos finales.

Si quisiera descifrar el valor, usaría el punto final /decrypt pasando la cadena cifrada en la llamada.

Ahora puede agregar la propiedad cifrada a su GitHub o al archivo de configuración basado en el sistema de archivos para el servicio de licencias usando la siguiente sintaxis:

```
spring.datasource.contraseña:"{cifrado}
858201e10fe3c9513e1d28b33ff417a66e8c8411dcff3077c53cf53d8a1be360"
```

El servidor de configuración de Spring Cloud requiere que se antepongan todas las propiedades cifradas con un valor de {cipher}. El valor {cipher} le dice al servidor de configuración de Spring Cloud se trata de un valor cifrado. Encienda su servidor de configuración Spring Cloud y presione el punto final OBTENER <http://localhost:8888/licensingservice/default> .

La Figura 3.9 muestra los resultados de esta convocatoria.  
Has hecho que spring.datasource.password sea más segura cifrando el  
propiedad, pero todavía tienes un problema. La contraseña de la base de datos se expone como texto sin formato.  
cuando presionas <http://localhost:8888/licensingservice/default>  
punto final.

```

1 "name": "licensingservice",
2 "profiles": [
3   "default"
4 ],
5 ],
6 "label": "master",
7 "version": "8b20dd9432ef9ef08216a5775859afb24a5e7d43",
8 "propertySources": [
9   {
10     "name": "https://github.com/carnellj/config-repo/licensingservice/licensingservice.yml",
11     "source": {
12       "example.property": "I AM IN THE DEFAULT",
13       "spring.jpa.database": "POSTGRESQL",
14       "spring.datasource.platform": "postgres",
15       "spring.jpa.show-sql": "true",
16       "spring.database.driverClassName": "org.postgresql.Driver",
17       "spring.datasource.url": "jdbc:postgresql://database:5432/eagle_eye_local",
18       "spring.datasource.username": "postgres",
19       "spring.datasource.testWhileIdle": "true",
20       "spring.datasource.validationQuery": "SELECT 1",
21       "spring.jpa.properties.hibernate.dialect": "org.hibernate.dialect.PostgreSQLDialect",
22       "redis.server": "redis",
23       "redis.port": "6379",
24       "signing.key": "345345fsdfsfs345",
25       "spring.datasource.password": "p0stgr@$"
26     }
27   }
28 ]
29 
```

propiedad spring.datasource.password  
almacenada como un valor cifrado

Figura 3.9 Si bien spring.datasource.password está cifrado en el archivo de propiedades, se descifra cuando se recupera la configuración del servicio de licencias. Esto sigue siendo problemático.

De forma predeterminada, Spring Cloud Config realizará todo el descifrado de propiedades en el servidor y pasará los resultados a las aplicaciones que consumen las propiedades como texto sin formato y sin cifrar. Sin embargo, puede indicarle a Spring Cloud Config que no descifre en el servidor y que sea responsabilidad de la aplicación recuperar los datos de configuración descifrar las propiedades cifradas.

#### 3.4.4 Configurar microservicios para usar cifrado en el lado del cliente

Para habilitar el descifrado de propiedades del lado del cliente, debe hacer tres cosas:

- 1 Configure Spring Cloud Config para no descifrar propiedades en el lado del servidor.
- 2 Configure la clave simétrica en el servidor de licencias.
- 3 Agregue los archivos JAR spring-security-rsa al archivo pom.xml de servicios de licencia.

Lo primero que debe hacer es deshabilitar el descifrado de propiedades del lado del servidor en Spring Cloud Config. Esto se hace configurando el archivo src/main/resources/application.yml de Spring Cloud Config para establecer la propiedad spring.cloud.config.server.encrypt.enabled: false. Eso es todo lo que tienes que hacer en Spring Cloud Config.

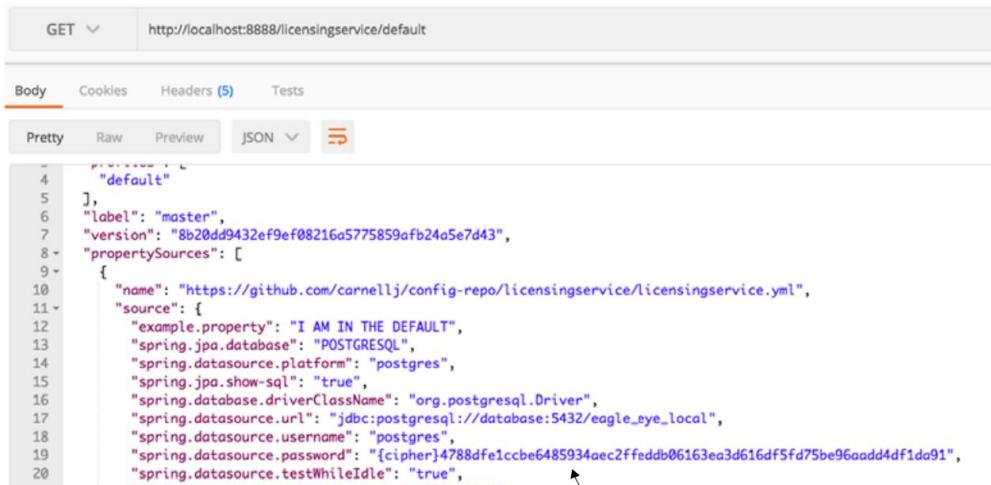
servidor.

Debido a que el servicio de licencias ahora es responsable de descifrar el cifrado propiedades, primero debe configurar la clave simétrica en el servicio de licencias haciendo asegúrese de que la variable de entorno ENCRYPT\_KEY esté configurada con la misma clave simétrica (por ejemplo, IMSYMMETRIC) que utilizó con su servidor Spring Cloud Config.

A continuación, debe incluir las dependencias JAR spring-security-rsa con servicio de licencia:

```
<dependencia>
    <groupId>org.springframework.security</groupId>
    <artifactId>seguridad-de-primavera-rsa</artifactId>
</dependencia>
```

Estos archivos JAR contienen el código Spring necesario para descifrar las propiedades cifradas. recuperándose de Spring Cloud Config. Con estos cambios implementados, puedes comenzar Spring Cloud Config y los servicios de licencia. Si accede al punto final <http://localhost:8888/licensingservice/default>, verá que spring.datasource.password se devuelve en formato cifrado. La figura 3.10 muestra la salida de la llamada.



```

4   "default": [
5     {
6       "label": "master",
7       "version": "8b20dd9432ef9ef08216a5775859afb24a5e7d43",
8       "propertySources": [
9         {
10           "name": "https://github.com/carnellj/config-repo/licensingservice/licensingservice.yml",
11           "source": {
12             "example.property": "I AM IN THE DEFAULT",
13             "spring.jpa.database": "POSTGRESQL",
14             "spring.datasource.platform": "postgres",
15             "spring.jpa.show-sql": "true",
16             "spring.database.driverClassName": "org.postgresql.Driver",
17             "spring.datasource.url": "jdbc:postgresql://database:5432/eagle_eye_local",
18             "spring.datasource.username": "postgres",
19             "spring.datasource.password": "[cipher]4788df1ccbe6485934aec2ffddb06163ea3d616df5fd75be96aadd4df1da91",
20             "spring.datasource.testWhileIdle": "true",
21             "spring.datasource.validationQuery": "SELECT 1",

```

La propiedad `spring.datasource.password` está cifrada.

Figura 3.10 Con el descifrado del lado del cliente activado, las propiedades confidenciales ya no se devolverán en texto sin formato desde la llamada REST de Spring Cloud Config. En cambio, el servicio que llama descifrará la propiedad cuando cargue sus propiedades desde Spring Cloud Config.

## 3.5 Pensamientos finales

La gestión de la configuración de aplicaciones puede parecer un tema mundano, pero es de importancia crítica en un entorno basado en la nube. Como veremos con más detalle en capítulos posteriores, es fundamental que sus aplicaciones y los servidores en los que se ejecutan sean inmutables y que todo el servidor que se promociona nunca se configure manualmente.

entre ambientes. Esto va en contra de los modelos de implementación tradicionales donde implementa un artefacto de aplicación (por ejemplo, un archivo JAR o WAR ) junto con sus archivos de propiedades en un entorno "fijo".

Con un modelo basado en la nube, los datos de configuración de la aplicación deben estar completamente separados de la aplicación, con las necesidades de datos de configuración adecuadas. inyectado en tiempo de ejecución para que el mismo artefacto de servidor/aplicación se promueva consistentemente en todos los entornos.

## 3.6 Resumen

El servidor de configuración de Spring Cloud le permite configurar las propiedades de la aplicación con valores específicos del entorno.

Spring usa perfiles Spring para lanzar un servicio y determinar qué entorno

Las propiedades deben recuperarse del servicio Spring Cloud Config.

El servicio de configuración de Spring Cloud puede utilizar una aplicación basada en archivos o basada en Git.

Repositorio de configuración para almacenar las propiedades de la aplicación.

El servicio de configuración Spring Cloud le permite cifrar archivos de propiedades confidenciales utilizando cifrado simétrico y asimétrico.

## Sobre el descubrimiento de servicios

### Este capítulo cubre

Explicar por qué el descubrimiento de servicios es importante para cualquier proveedor de servicios en la nube.  
entorno de aplicación basado

Comprender los pros y los contras del descubrimiento de servicios frente al  
enfoque más tradicional de equilibrio de carga

Configuración de un servidor Spring Netflix Eureka

Registrar un microservicio basado en Spring-Boot con Eureka

Uso de Spring Cloud y la biblioteca Ribbon de Netflix para usar el  
equilibrio de carga del lado del cliente

En cualquier arquitectura distribuida, necesitamos encontrar la dirección física donde se encuentra  
se encuentra la máquina. Este concepto ha existido desde el comienzo de la computación distribuida  
y se conoce formalmente como descubrimiento de servicios. El descubrimiento de servicios puede  
ser algo tan sencillo como mantener un fichero de propiedades con las direcciones de todos los  
servicios remotos utilizados por una aplicación, o algo tan formalizado (y complicado) como un  
repositorio UDDI (Descripción, Descubrimiento e Integración Universal).<sup>1</sup>

<sup>1</sup> [https://en.wikipedia.org/wiki/Web\\_Services\\_Discovery#Universal\\_Description\\_Discovery\\_and\\_Integration](https://en.wikipedia.org/wiki/Web_Services_Discovery#Universal_Description_Discovery_and_Integration)

El descubrimiento de servicios es fundamental para las aplicaciones de microservicios basadas en la nube para dos claves. razones. En primer lugar, ofrece al equipo de aplicaciones la capacidad de escalar horizontalmente rápidamente y reducir el número de instancias de servicio que se ejecutan en un entorno. El servicio los consumidores son abstraídos de la ubicación física del servicio a través del descubrimiento del servicio. Debido a que los consumidores del servicio no conocen la ubicación física del sitio real instancias de servicio, se pueden agregar o eliminar nuevas instancias de servicio del grupo de servicios disponibles.

Esta capacidad de escalar rápidamente los servicios sin interrumpir a los consumidores del servicio es una concepto extremadamente poderoso, porque mueve a un equipo de desarrollo acostumbrado a construir aplicaciones monolíticas de un solo inquilino (por ejemplo, un cliente) lejos de pensar en escalar solo en términos de agregar hardware mejor y más grande (escalado vertical) a el enfoque más poderoso para escalar agregando más servidores (escalado horizontal).

Un enfoque monolítico suele llevar a los equipos de desarrollo a comprar en exceso sus necesidades de capacidad. Los aumentos de capacidad se producen en grupos y picos y son rara vez un camino suave y estable. Los microservicios nos permiten ampliar o reducir nuevos servicios instancias. El descubrimiento de servicios ayuda a abstraer que estas implementaciones se están produciendo lejos del consumidor del servicio.

El segundo beneficio del descubrimiento de servicios es que ayuda a aumentar la resiliencia de las aplicaciones. Cuando una instancia de microservicio deja de estar en buen estado o no está disponible, la mayoría de los servicios Los motores de descubrimiento eliminarán esa instancia de su lista interna de servicios disponibles. El daño causado por un servicio caído se minimizará porque el descubrimiento del servicio El motor enrutaría los servicios alrededor del servicio no disponible.

Hemos analizado los beneficios del descubrimiento de servicios, pero ¿cuál es el problema? ¿é? Después de todo, ¿no podemos utilizar métodos probados y verdaderos como DNS (Servicio de nombres de dominio)? ¿O un equilibrador de carga para ayudar a facilitar el descubrimiento de servicios? Analicemos por qué eso no funcionará con una aplicación basada en microservicios, particularmente una que se ejecuta en la nube.

## 4.1 ¿Dónde está mi servicio?

Siempre que tenga una aplicación que llame a recursos distribuidos en varios servidores, necesita localizar la ubicación física de esos recursos. En el mundo sin nubes, esto La resolución de la ubicación del servicio a menudo se resolvía mediante una combinación de DNS y un equilibrador de carga de red. La figura 4.1 ilustra este modelo.

Una aplicación necesita invocar un servicio ubicado en otra parte de la organización. Intenta invocar el servicio utilizando un nombre DNS genérico junto con una ruta. que representa de forma única el servicio que la aplicación intentaba invocar. El DNS El nombre se resolvería en un equilibrador de carga comercial, como el popular F5. equilibrador (<http://f5.com>) o un balanceador de carga de código abierto como HAProxy (<http://haproxy.org>).

## Aplicaciones que consumen servicios

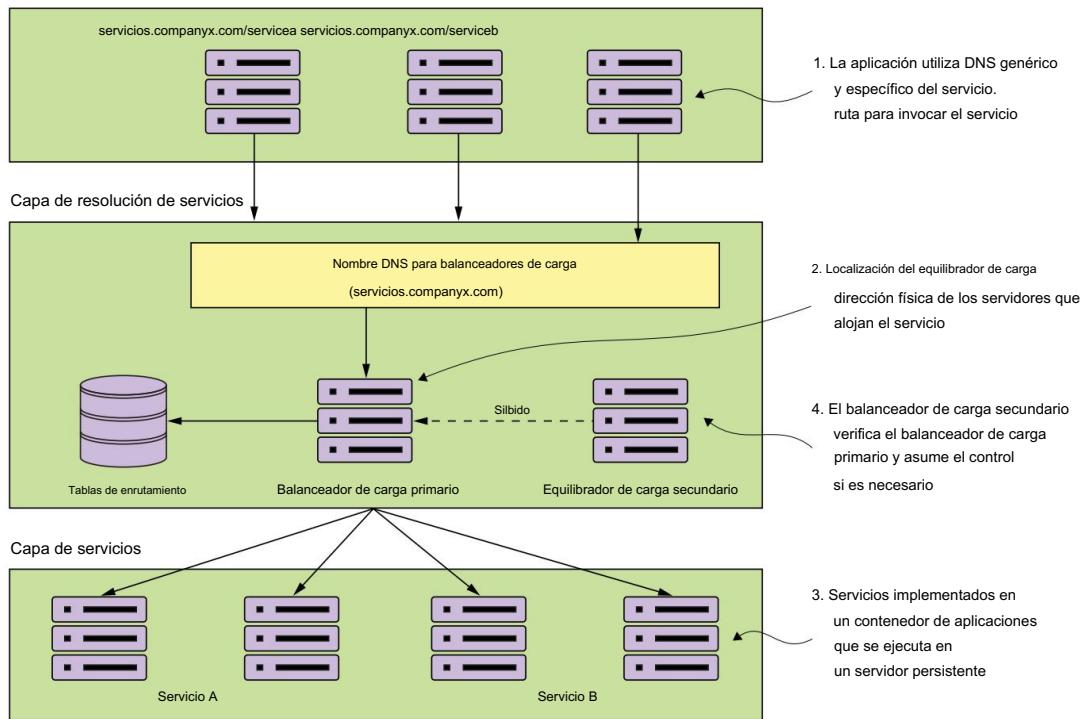


Figura 4.1 Un modelo tradicional de resolución de ubicación de servicios que utiliza DNS y un equilibrador de carga

El balanceador de carga, al recibir la solicitud del consumidor del servicio, localiza el entrada de dirección física en una tabla de enruteamiento basada en la ruta que el usuario estaba intentando acceso. Esta entrada de la tabla de enruteamiento contiene una lista de uno o más servidores que alojan el servicio. Luego, el equilibrador de carga elige uno de los servidores de la lista y reenvía la solicitud en ese servidor.

Cada instancia de un servicio se implementa en uno o más servidores de aplicaciones. El número de estos servidores de aplicaciones era a menudo estático (por ejemplo, el número de los servidores de aplicaciones que alojaban un servicio no subían ni bajaban) y persistentes (por ejemplo, si un servidor que ejecuta un servidor de aplicaciones fallaba, se restauraría al mismo indicar que estaba en el momento del accidente, y tendría la misma IP y configuración que lo había hecho anteriormente.)

Para lograr una forma de alta disponibilidad, un equilibrador de carga secundario está inactivo y haciendo ping al equilibrador de carga principal para ver si está activo. Si no está vivo, la carga secundaria. El balanceador de carga se activa, tomando el control de la dirección IP del balanceador de carga principal y comenzando a atender solicitudes.

Si bien este tipo de modelo funciona bien con aplicaciones que se ejecutan dentro de los cuatro paredes de un centro de datos corporativo y con un número relativamente pequeño de servicios en ejecución

en un grupo de servidores estáticos, no funciona bien para aplicaciones de microservicios basadas en la nube. Las razones para esto incluyen

Punto único de falla: si bien el balanceador de carga puede tener alta disponibilidad, es un único punto de falla para toda su infraestructura. Si el balanceador de carga va cae, todas las aplicaciones que dependen de él también caen. Mientras puedes hacer una carga balanceador altamente disponible, los balanceadores de carga tienden a ser puntos de estrangulamiento centralizados dentro de la infraestructura de su aplicación.

Escalabilidad horizontal limitada: al centralizar sus servicios en un único grupo de balanceadores de carga, tiene una capacidad limitada para escalar horizontalmente su equilibrio de carga infraestructura en múltiples servidores. Muchos balanceadores de carga comerciales están limitados por dos cosas: su modelo de redundancia y los costos de licencia. La mayoría de los balanceadores de carga comerciales utilizan un modelo de intercambio en caliente para lograr redundancia, por lo que solo tiene un único servidor para manejar la carga, mientras que el equilibrador de carga secundario está ahí solo para conmutación por error en caso de una interrupción del equilibrador de carga principal. Estás dentro esencia, limitada por su hardware. En segundo lugar, los balanceadores de carga comerciales también tienen modelos de licencia restrictivos orientados a una capacidad fija en lugar de una modelo más variable.

Administrado estáticamente: la mayoría de los balanceadores de carga tradicionales no están diseñados para una rápida altas y bajas de servicios. Utilizan una base de datos centralizada para almacenar las rutas para las reglas y la única forma de agregar nuevas rutas suele ser a través la API (interfaz de programación de aplicaciones) propiedad del proveedor. Complejo: debido a que un balanceador de carga actúa como un proxy para los servicios, las solicitudes de los consumidores de servicios deben asignarse a los servicios físicos. Esta capa de traducción a menudo agrega una capa de complejidad a su infraestructura de servicio porque las reglas de mapeo para el servicio deben definirse y desplegado a mano. En un escenario de balanceador de carga tradicional, este registro de las nuevas instancias de servicio se realizaron manualmente y no en el momento de inicio de una nueva instancia de servicio.

Estas cuatro razones no son una crítica general a los balanceadores de carga. Funcionan bien en un entorno corporativo donde se puede manejar el tamaño y la escala de la mayoría de las aplicaciones, a través de una infraestructura de red centralizada. Además, los balanceadores de carga todavía tienen una papel que desempeñar en términos de centralización de la terminación SSL y gestión de la seguridad del puerto de servicio. Un equilibrador de carga puede bloquear el puerto de entrada (entrada) y de salida (salida) acceso a todos los servidores que se encuentran detrás de él. Este concepto de mínimo acceso a la red es a menudo una componente crítica al intentar cumplir con los requisitos de certificación estándar de la industria, como el cumplimiento de PCI (Industria de tarjetas de pago).

Sin embargo, en la nube, donde hay que lidiar con cantidades masivas de transacciones y redundancia, una pieza centralizada de infraestructura de red en última instancia no funciona tan bien porque no escala de manera efectiva y no es rentable. Vamos

Ahora veamos cómo se puede implementar un mecanismo sólido de descubrimiento de servicios para aplicaciones basadas en la nube.

## 4.2 Sobre el descubrimiento de servicios en la nube

La solución para un entorno de microservicios basado en la nube es utilizar un servicio de descubrimiento mecanismo que es

Altamente disponible: el descubrimiento de servicios debe ser capaz de admitir un entorno de agrupación en clústeres "caliente" donde las búsquedas de servicios se puedan compartir entre múltiples nodos en un clúster de descubrimiento de servicios. Si un nodo deja de estar disponible, otros nodos en el El cluster debería poder tomar el control.

Punto a punto: cada nodo en el clúster de descubrimiento de servicios comparte el estado de un servidor. vice instancia.

Carga equilibrada: el descubrimiento de servicios necesita equilibrar dinámicamente la carga de las solicitudes. en todas las instancias de servicio para garantizar que las invocaciones de servicio se distribuyan en todas las instancias de servicio gestionadas por él. En muchos sentidos, el descubrimiento de servicios reemplaza los balanceadores de carga más estáticos y administrados manualmente utilizados en muchos de los primeros Implementaciones de aplicaciones web.

Resiliente: el cliente del descubrimiento de servicios debe "almacenar en caché" la información del servicio. en la zona. El almacenamiento en caché local permite una degradación gradual del descubrimiento del servicio. función de modo que si el servicio de descubrimiento de servicios deja de estar disponible, las aplicaciones aún pueden funcionar y localizar los servicios en función de la información mantenida. guardado en su caché local.

Tolerante a fallos: el descubrimiento de servicios necesita detectar cuándo una instancia de servicio no está saludable y elimine la instancia de la lista de servicios disponibles que pueden tomar solicitudes de los clientes. Debería detectar estas fallas en los servicios y tomar medidas sin intervención humana.

En las siguientes secciones vamos a

Recorrer la arquitectura conceptual de cómo funcionará un agente de descubrimiento de servicios basado en la nube.

Mostrar cómo el almacenamiento en caché y el equilibrio de carga del lado del cliente permiten que un servicio continúe funcionar incluso cuando el agente de descubrimiento de servicios no está disponible

Vea cómo implementar el descubrimiento de servicios utilizando Spring Cloud y Netflix.

Agente de descubrimiento de servicios Eureka

### 4.2.1 La arquitectura del descubrimiento de servicios

Para comenzar nuestra discusión sobre la arquitectura de descubrimiento de servicios, debemos comprender cuatro conceptos. Estos conceptos generales se comparten en todas las implementaciones de descubrimiento de servicios:

Registro de servicio: ¿cómo se registra un servicio con el agente de descubrimiento de servicios?

Búsqueda de dirección de servicio por parte del cliente: ¿cuál es el medio por el cual un cliente de servicio busca subir información de servicio?

Intercambio de información: ¿Cómo se comparte la información del servicio entre nodos?

Monitoreo de la salud: ¿cómo comunican los servicios su salud al servicio?

¿Vice agente de descubrimiento?

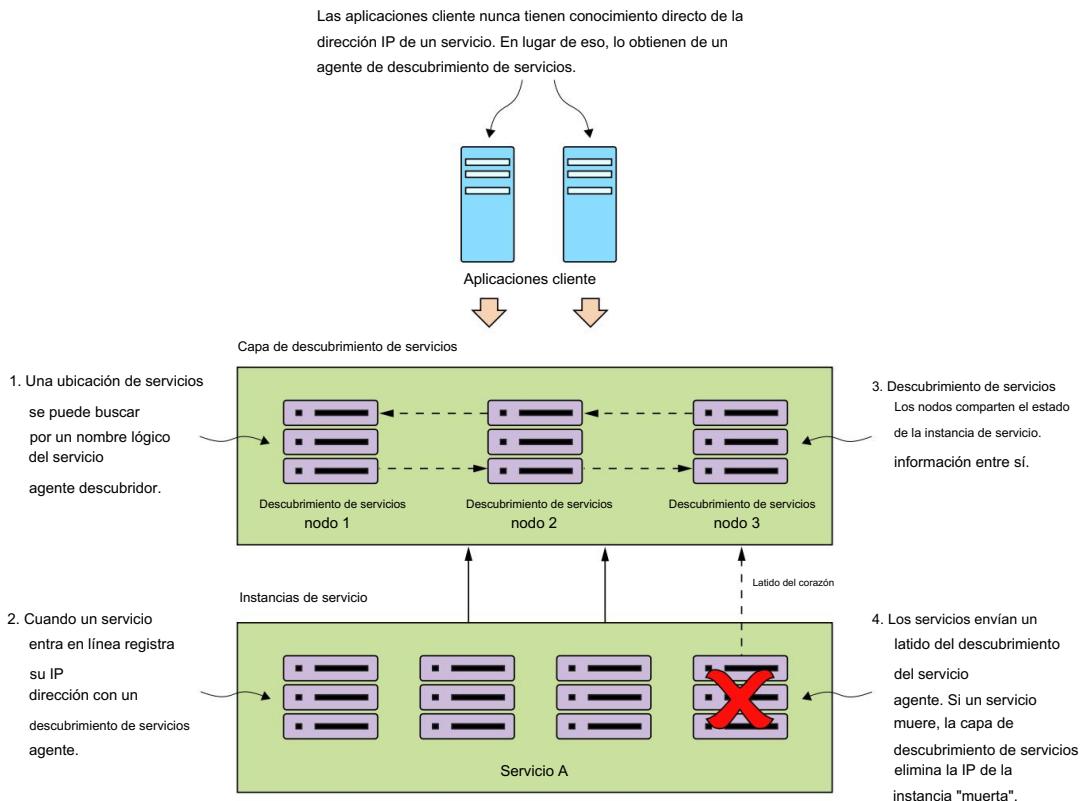


Figura 4.2 A medida que se agregan o eliminan instancias de servicio, actualizarán el agente de descubrimiento de servicios y estarán disponibles para procesar las solicitudes de los usuarios.

La Figura 4.2 muestra el flujo de estos cuatro puntos y lo que normalmente ocurre en un servicio.

#### Implementación del patrón de descubrimiento.

En la figura 4.2, se han iniciado uno o más nodos de descubrimiento de servicios. Estos servicios las instancias de descubrimiento suelen ser únicas y no tienen un equilibrador de carga delante de ellos.

A medida que se inician las instancias de servicio, registrarán su ubicación física, ruta y puerto, a los que se puede acceder con una o más instancias de descubrimiento de servicios. mientras cada uno una instancia de un servicio tendrá una dirección IP y un puerto únicos, cada instancia de servicio que aparece se registrará con el mismo ID de servicio. Un ID de servicio no es más que una clave que identifica de forma única un grupo de las mismas instancias de servicio.

Por lo general, un servicio solo se registrará en una instancia de servicio de descubrimiento de servicios. Mayoría Las implementaciones de descubrimiento de servicios utilizan un modelo de propagación de datos de igual a igual, donde los datos alrededor de cada instancia de servicio se comunican a todos los demás nodos en el cúmulo.

Dependiendo de la implementación del descubrimiento de servicios, el mecanismo de propagación podría utilizar una lista codificada de servicios para propagarse o utilizar un protocolo de multidifusión como el protocolo "gossip"<sup>2</sup> o "estilo de infección"<sup>3</sup> para permitir que otros nodos "descubran" "Cambios en el cluster".

Finalmente, cada instancia de servicio pasará a su estado o será retirado de su estado por el servicio de descubrimiento de servicios. Cualquier servicio que no devuelva una buena verificación de estado se eliminará del grupo de instancias de servicio disponibles.

Una vez que un servicio se ha registrado con un servicio de descubrimiento de servicios, está listo para ser utilizado por una aplicación o servicio que necesita utilizar sus capacidades. Existen diferentes modelos para que un cliente "descubra" un servicio. Un cliente puede confiar únicamente en el motor de descubrimiento de servicios para resolver las ubicaciones de los servicios cada vez que se llama a un servicio. Con este enfoque, el motor de descubrimiento de servicios se invocará cada vez que se realice una llamada a una instancia de microservicio registrada. Desafortunadamente, este enfoque es frágil porque el cliente del servicio depende completamente del motor de descubrimiento de servicios que se ejecuta para encontrar e invocar un servicio.

Un enfoque más sólido es utilizar lo que se llama equilibrio de carga del lado del cliente.<sup>4</sup> Figura 4.3 ilustra este enfoque.

En este modelo, cuando un actor consumidor necesita invocar un servicio **1**, se comunicará con el servicio de descubrimiento de servicios para todas las instancias de servicio que solicita un consumidor de servicios y luego almacenará en caché los datos localmente en la máquina del consumidor de servicios.

**2** Cada vez que un cliente quiera llamar al servicio, el consumidor del servicio buscará la información de ubicación del servicio en la memoria caché. Por lo general, el almacenamiento en caché del lado del cliente utilizará un algoritmo de equilibrio de carga simple, como el algoritmo de equilibrio de carga "round-robin", para garantizar que las llamadas de servicio se distribuyan entre múltiples instancias de servicio.

**3** Luego, el cliente se comunicará periódicamente con el servicio de descubrimiento de servicios y actualizará su caché de instancias de servicio. La caché del cliente eventualmente es consistente, pero siempre existe el riesgo de que entre el momento en que el cliente contacta la instancia de descubrimiento de servicio para una actualización y se realizan las llamadas, las llamadas se puedan dirigir a una instancia de servicio que no esté en buen estado.

Si, durante el curso de la llamada a un servicio, la llamada al servicio falla, la caché de descubrimiento de servicios local se invalida y el cliente de descubrimiento de servicios intentará actualizar sus entradas desde el agente de descubrimiento de servicios.

Ahora tomemos el patrón de descubrimiento de servicios genérico y aplíquemlos a su dominio problemático de EagleEye.

<sup>2</sup> [https://en.wikipedia.org/wiki/Gossip\\_protocol](https://en.wikipedia.org/wiki/Gossip_protocol)

<sup>3</sup> <https://www.cs.cornell.edu/~asdas/research/dsn02-swim.pdf>

<sup>4</sup> [https://en.wikipedia.org/wiki/Load\\_balancing\\_\(computación \)#Client-Side\\_Random\\_Load\\_Balancing](https://en.wikipedia.org/wiki/Load_balancing_(computación )#Client-Side_Random_Load_Balancing)

1. Cuando un cliente de servicio necesita llamar a un servicio, lo comprobará un caché local para las IP de la instancia de servicio. El equilibrio de carga entre instancias de servicio se producirá en el servicio.

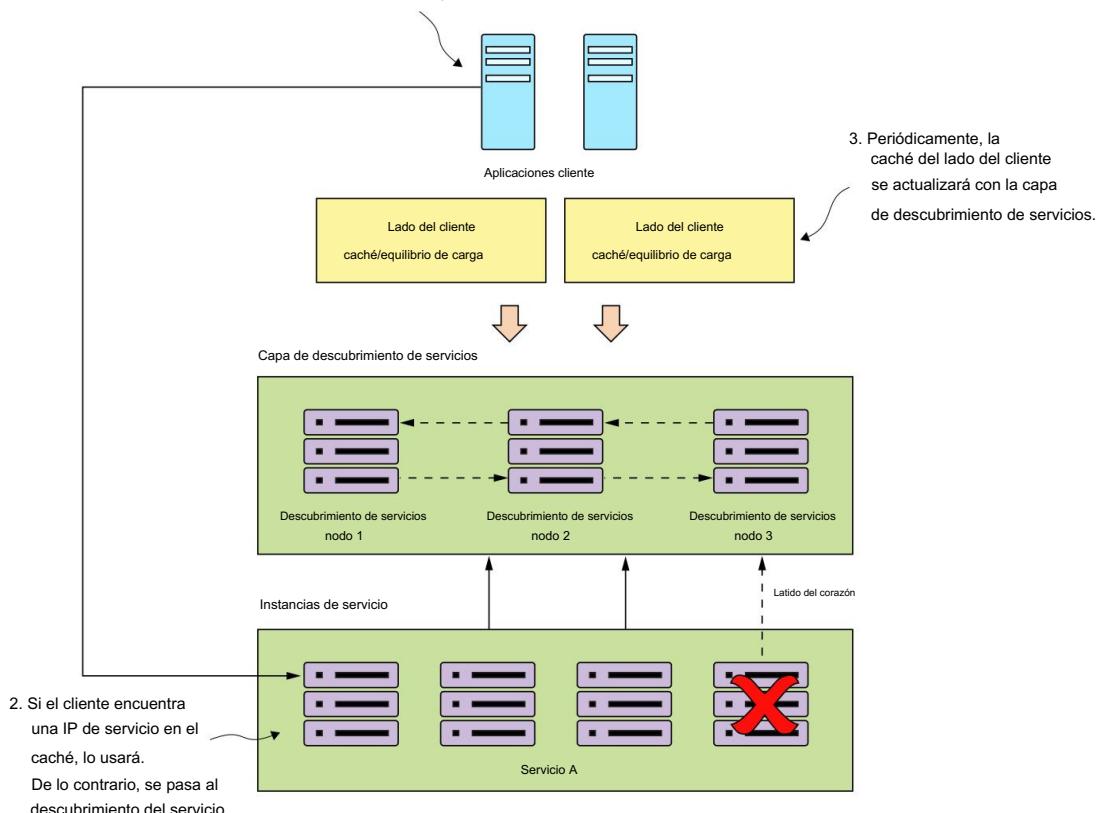


Figura 4.3 El equilibrio de carga del lado del cliente almacena en caché la ubicación de los servicios para que el cliente del servicio no tenga que ponerse en contacto con el descubrimiento del servicio en cada llamada.

#### 4.2.2 Descubrimiento de servicios en acción usando Spring y Netflix Eureka

Ahora implementará el descubrimiento de servicios configurando un descubrimiento de servicios agente y luego registrar dos servicios con el agente. Entonces tendrás un servicio llamar a otro servicio utilizando la información recuperada por el descubrimiento de servicios. Primavera La nube ofrece múltiples métodos para buscar información a partir del descubrimiento de un servicio. agente. También analizaremos las fortalezas y debilidades de cada enfoque.

Una vez más, el proyecto Spring Cloud hace que este tipo de configuración sea trivial. Utilizará Spring Cloud y el motor de descubrimiento de servicios Eureka de Netflix para implementar su patrón de descubrimiento de servicios. Para el equilibrio de carga del lado del cliente usarás Spring Bibliotecas Ribbon de Cloud y Netflix.

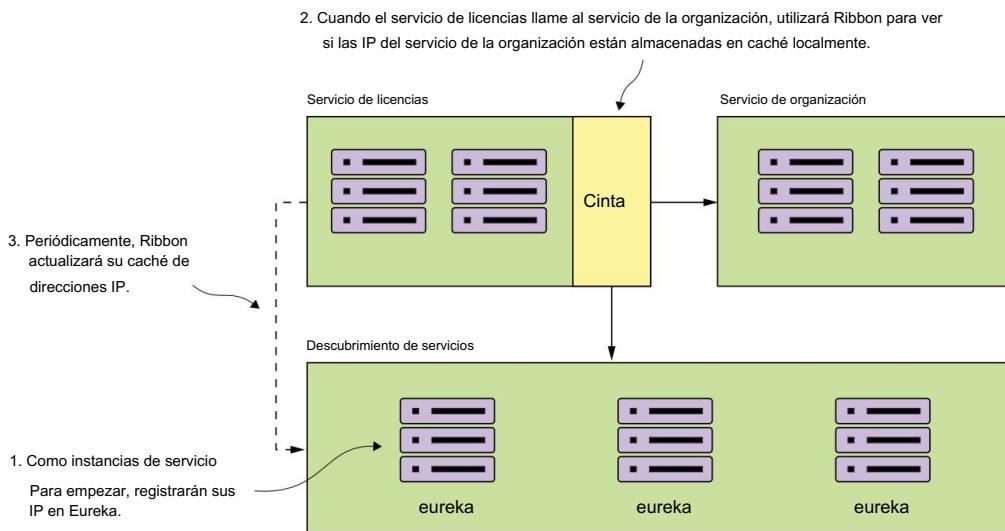


Figura 4.4 Al implementar el almacenamiento en caché del lado del cliente y Eureka con los servicios de organización y licencia, puede reducir la carga en los servidores de Eureka y mejorar la estabilidad del cliente si Eureka deja de estar disponible.

En los dos capítulos anteriores, mantuvo su servicio de licencias simple e incluyó el nombre de la organización para las licencias con los datos de la licencia. En este capítulo, dividirá la información de la organización en su propio servicio.

Cuando se invoca el servicio de licencias, llamará al servicio de la organización para recuperar la información de la organización asociada con el ID de la organización designada. La resolución real de la ubicación del servicio de la organización se llevará a cabo en un registro de descubrimiento de servicios. Para este ejemplo, registrará dos instancias del servicio de la organización con un registro de descubrimiento de servicios y luego utilizará el equilibrio de carga del lado del cliente para buscar y almacenar en caché el registro en cada instancia de servicio. La figura 4.4 muestra esta disposición:

- 1 Como los servicios se están iniciando, los servicios de organización y licencia también se registrarán en el Servicio Eureka. Este proceso de registro le indicará a Eureka la ubicación física y el número de puerto de cada instancia de servicio junto con una identificación del servicio que se está iniciando.
- 2 Cuando el servicio de licencias llama al servicio de la organización, utilizará la biblioteca Netflix Ribbon para proporcionar equilibrio de carga de diapositivas del cliente. Ribbon se comunicará con el servicio Eureka para recuperar la información de ubicación del servicio y luego la almacenará en caché localmente.
- 3 Períódicamente, la biblioteca Netflix Ribbon hará ping al servicio Eureka y actualizará su caché local de ubicaciones de servicio.

Cualquier nueva instancia de servicios de la organización ahora será visible para el servicio de licencias localmente, mientras que cualquier instancia que no esté en buen estado se eliminará de la caché local.

A continuación, implementará este diseño configurando su servicio Spring Cloud Eureka.

## 4.3 Creación de su servicio Spring Eureka

En esta sección, configurarás nuestro servicio Eureka utilizando Spring Boot. Al igual que el servicio de configuración de Spring Cloud, la configuración de un servicio Spring Cloud Eureka comienza con la creación de un nuevo proyecto Spring Boot y la aplicación de anotaciones y configuraciones. Comencemos con su maven pom.xml.<sup>5</sup> La siguiente lista muestra las dependencias del servicio Eureka que necesitará para el proyecto Spring Boot que está configurando.

Listado 4.1 Agregar dependencias a su pom.xml

```
<?xml versión="1.0" codificación="UTF-8"?> <proyecto
xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.
xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.thinktmechanix</groupId> <artifactId>eurekasvr</
artifactId> <versión>0.0.1-SNAPSHOT</versión>
<packaging>jar</packaging>

<nombre>Eureka Server</nombre>
<descripción>Proyecto de demostración de Eureka Server</descripción>

<!--No se muestran las definiciones de Maven para usar Spring Cloud Parent-->
<dependencias>
    <dependencia>
        <groupId>org.springframework.cloud</groupId> <artifactId>spring-
        cloud-starter-eureka-server</artifactId>
    </dependencia> </
dependencias>

Resto de pom.xml eliminado por motivos de concisión
....>
</proyecto>
```



Luego deberá configurar el archivo src/main/resources/application.yml con la configuración necesaria para configurar el servicio Eureka ejecutándose en modo independiente (por ejemplo, sin otros nodos en el clúster), como se muestra en el siguiente listado.

Listado 4.2 Configurando su configuración de Eureka en el archivo application.yml

```
servidor:
  puerto: 8761
  eureka:
    cliente:
      registrarseConEureka: falso
```

<sup>5</sup> Todo el código fuente de este capítulo se puede descargar desde GitHub (<https://github.com/carnellj/spmia-capítulo4>). El servicio Eureka se encuentra en el ejemplo del capítulo 4/eurekasvr. Todos los servicios de este capítulo se crearon utilizando Docker y Docker Compose para que puedan abrirse en una sola instancia.

buscarRegistro: falso  
servidor:

esperarTimeInMsWhenSyncEmpty: 5

No almacene en caché la información del registro localmente.

Tiempo inicial de espera antes de que el servidor acepte solicitudes

Las propiedades clave que se están configurando son el atributo server.port que establece el puerto predeterminado utilizado para el servicio Eureka. El atributo eureka.client.registerWithEureka le dice al servicio que no se registre con un servicio Eureka cuando Spring Boot Eureka La aplicación se inicia porque este es el servicio Eureka. El cliente eureka. El atributo .fetchRegistry se establece en falso para que cuando se inicie el servicio Eureka, no intente almacenar en caché su información de registro localmente. Al ejecutar un cliente Eureka, querrás cambiar este valor para los servicios Spring Boot que se van a registrar con Eureka.

Notarás que el último atributo, eureka.server.waitTimeInMsWhenSync Vacío, está comentado. Cuando pruebas tu servicio localmente, debes descomentar esta línea porque Eureka no anunciará inmediatamente ningún servicio que se registre. con eso. Esperará cinco minutos de forma predeterminada para darles a todos los servicios la oportunidad de registrarse. con él antes de anunciarlos. Descomentar esta línea para pruebas locales ayudará acelerar el tiempo que tardará en iniciarse el servicio Eureka y mostrar los servicios registrados en él.

El registro de servicios individuales tardará hasta 30 segundos en aparecer en Eureka servicio porque Eureka requiere tres pings de latidos consecutivos del servicio con un intervalo de 10 segundos antes de que diga que el servicio está listo para su uso. Mantén esto en mente mientras implementas y pruebas tus propios servicios.

El último trabajo de configuración que realizará para configurar su servicio Eureka es agregando una anotación a la clase de arranque de la aplicación que estás usando para iniciar tu Servicio Eureka. Para el servicio Eureka, se puede encontrar la clase de arranque de la aplicación en la clase src/main/java/com/thinkmechanix/eurekasvr/EurekaServer-Application.java . La siguiente lista muestra dónde agregar sus anotaciones.

#### Listado 4.3 Anotación de la clase bootstrap para habilitar el servidor Eureka

```
paquete com.thinkmechanix.eurekasvr;

importar org.springframework.boot.SpringApplication;
importar org.springframework.boot.autoconfigure.SpringBootApplication;
importar org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer clase
pública Aplicación EurekaServer {
    público estático vacío principal (String [] argumentos) {
        SpringApplication.run(EurekaServerApplication.clase, argumentos);
    }
}
```

Habilitar el servidor Eureka en el servicio de primavera

Utiliza solo una anotación nueva para indicar que su servicio es un servicio Eureka; ese es `@EnableEurekaServer`. En este punto, puede iniciar el servicio Eureka ejecutando `mvn spring-boot:run` o ejecutando `docker-compose` (consulte el apéndice A) para iniciar el servicio. Una vez que se ejecuta este comando, debería tener un servicio Eureka en ejecución sin ningún servicio registrado en él. A continuación, creará el servicio de organización y lo registrará en su servicio Eureka.

#### 4.4 Registrar servicios con Spring Eureka

En este punto, tiene un servidor Eureka basado en Spring en funcionamiento. En esta sección, configurará su organización y los servicios de licencia para registrarse en su servidor Eureka. Este trabajo se realiza como preparación para que un cliente de servicio busque un servicio en su registro de Eureka. Cuando haya terminado con esta sección, debería tener un conocimiento firme de cómo registrar un microservicio Spring Boot con Eureka.

Registrar un microservicio basado en Spring Boot con Eureka es un ejercicio extremadamente simple. Para los propósitos de este capítulo, no vamos a analizar todo el código Java involucrado en la escritura del servicio (mantuvimos esa cantidad de código pequeña a propósito), sino que nos centraremos en registrar el servicio en el registro de servicios Eureka que usted creó. en el apartado anterior.

Lo primero que debe hacer es agregar la dependencia Spring Eureka al archivo `pom.xml` del servicio de su organización:

```
<dependencia>
    <groupId>org.springframework.cloud</groupId> <artifactId>spring-
    cloud-starter-eureka</artifactId>
</dependencia>
```

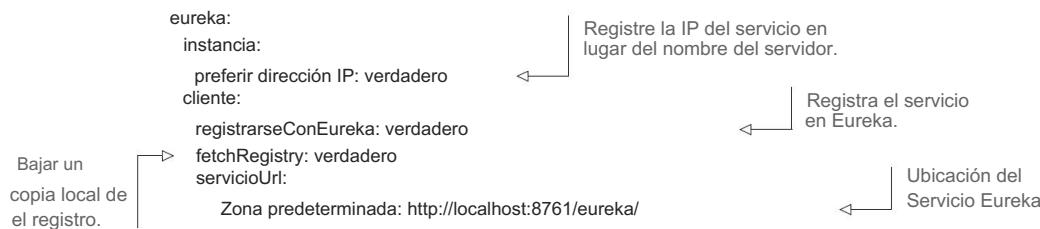
Incluye las bibliotecas de Eureka  
para que el servicio pueda  
registrarse en Eureka

La única biblioteca nueva que se está utilizando es la biblioteca `spring-cloud-starter-eureka` . El artefacto `spring-cloud-starter-eureka` contiene los archivos jar que Spring Cloud utilizará para interactuar con su servicio Eureka.

Después de haber configurado su archivo `pom.xml`, debe indicarle a Spring Boot que registre el servicio de la organización con Eureka. Este registro se realiza mediante una configuración adicional en el archivo `src/main/java/resources/application.yml` del servicio de la organización, como se muestra en el siguiente listado.

##### Listado 4.4 Modificar el archivo `application.yml` del servicio de su organización para hablar con Eureka

```
primavera:
  aplicación:
    nombre: organizaciónservicio perfiles: ← Nombre lógico del servicio que se
    activo: registrará en Eureka.
          por defecto
  nube:
    configuración:
      habilitado: verdadero
```



Cada servicio registrado en Eureka tendrá dos componentes asociados: el ID de la aplicación y el ID de la instancia. El ID de la aplicación se utiliza para representar un grupo. instancia de servicio. En un microservicio basado en Spring-Boot, el ID de la aplicación siempre será el valor establecido por la propiedad `spring.application.name`. Para el servicio de su organización, su `spring.application.name` se denomina creativamente servicio de organización. El ID de instancia será un número aleatorio destinado a representar una única instancia de servicio.

**NOTA** Recuerde que normalmente la propiedad `spring.application.name` va en el archivo `bootstrap.yml`. Lo he incluido en `application.yml` con fines ilustrativos. El código funcionará con `spring.application.name` pero el lugar adecuado a largo plazo para este atributo es el archivo `bootstrap.yml`.

La segunda parte de su configuración proporciona cómo y dónde debe registrarse el servicio con el servicio Eureka. La propiedad `eureka.instance.preferIpAddress` le dice a Eureka que desea registrar la dirección IP del servicio en Eureka en lugar de su nombre de host.

#### ¿Por qué preferir la dirección IP?

De forma predeterminada, Eureka intentará registrar los servicios que lo contactan por nombre de host. Este funciona bien en un entorno basado en servidor donde a un servicio se le asigna un servidor DNS nombre de host. Sin embargo, en una implementación basada en contenedores (por ejemplo, Docker), los contenedores se iniciarán con nombres de host generados aleatoriamente y sin entradas DNS para los contenedores.

Si no configura `eureka.instance.preferIpAddress` en verdadero, su cliente Las aplicaciones no resolverán adecuadamente la ubicación de los nombres de host porque habrá No habrá ninguna entrada DNS para ese contenedor. Configurar el atributo `preferIpAddress` informar al servicio Eureka que el cliente desea ser anunciado por dirección IP.

Personalmente, siempre establecemos este atributo en verdadero. Se supone que los microservicios basados en la nube son efímeros y sin estado. Se pueden poner en marcha y apagar a voluntad. Las direcciones IP son más apropiadas para este tipo de servicios.

El atributo `eureka.client.registerWithEureka` es el disparador que le indica al servicio de la organización que se registre en Eureka. El registro `eureka.client.fetch` El atributo se utiliza para indicarle al cliente Spring Eureka que obtenga una copia local del registro. Establecer este atributo en verdadero almacenará en caché el registro localmente en lugar de llamar a Eureka servicio con cada búsqueda. Cada 30 segundos, el software del cliente volverá a contactar con el Servicio Eureka para cualquier cambio en el registro.

El último atributo, el atributo eureka.serviceUrl.defaultZone , contiene una lista separada por comas de los servicios Eureka que el cliente utilizará para resolver las ubicaciones de los servicios. Para nuestros propósitos, solo tendrá un servicio Eureka.

### Eureka alta disponibilidad

Configurar varios servicios de URL no es suficiente para lograr una alta disponibilidad. El atributo eureka.serviceUrl.defaultZone solo proporciona una lista de servicios de Eureka con los que el cliente puede comunicarse. También necesita configurar los servicios de Eureka para replicar el contenido de su registro entre sí.

Un grupo de registros de Eureka se comunican entre sí utilizando un modelo de comunicación de igual a igual en el que cada servicio de Eureka debe configurarse para conocer los otros nodos del cluster. La configuración de un clúster Eureka está fuera del alcance de este libro. Si está interesado en configurar un clúster de Eureka, visite Spring Cloud Sitio web del proyecto para más información.<sup>a</sup>

<sup>a</sup> <http://projects.spring.io/spring-cloud/spring-cloud.html>

En este punto tendrás un único servicio registrado con tu servicio Eureka.

Puede utilizar la API REST de Eureka para ver el contenido del registro. para ver todos los instancias de un servicio, acceda al siguiente punto final GET :

`http://<servicio eureka>:8761/eureka/apps/<APPID>`

Por ejemplo, para ver el servicio de la organización en el registro puede llamar a `http://localhost:8761/eureka/apps/organizationservice`.



Figura 4.5 Llamar a la API REST de Eureka para ver la organización mostrará la dirección IP de las instancias de servicio registradas en Eureka, junto con el estado del servicio.

El formato predeterminado devuelto por el servicio Eureka es XML. Eureka también puede devolver el datos en la figura 4.5 como una carga útil JSON , pero debe configurar el encabezado Aceptar HTTP en ser aplicación/json. En la figura 4.6 se muestra un ejemplo de la carga útil JSON .

El encabezado HTTP Aceptar establecido en application/json devolverá la información del servicio en JSON.

```

1 - [
2 -   "application": {
3 -     "name": "ORGANIZATIONSERVICE",
4 -     "instance": [
5 -       {
6 -         "instanceId": "255a89c6eb56:organizationservice:8085",
7 -         "hostName": "172.19.0.7",
8 -         "app": "ORGANIZATIONSERVICE",
9 -         "ipAddr": "172.19.0.7",
10 -        "status": "UP",
11 -        "overriddenstatus": "UNKNOWN",
12 -        "port": {
13 -          "g": 8085,
14 -          "@enabled": "true"
15 -        },

```

Figura 4.6 Llamada a la API REST de Eureka y los resultados son JSON

#### Sobre Eureka y las startups de servicios: no seas impaciente

Cuando un servicio se registra en Eureka, Eureka esperará tres controles de salud sucesivos. verificaciones en el transcurso de 30 segundos antes de que el servicio esté disponible a través de un Eureka. Este período de preparación desconcierta a los desarrolladores porque piensan que Eureka no ha registrado sus servicios si intenta llamar a su servicio inmediatamente después del El servicio ha sido lanzado. Esto es evidente en nuestros ejemplos de código que se ejecutan en Docker. entorno, porque el servicio Eureka y los servicios de aplicaciones (licencias y servicios de organización) todos se inician al mismo tiempo. Tenga en cuenta que después de iniciar el aplicación, es posible que reciba errores 404 sobre servicios que no se encuentran, aunque el servicio en sí ha comenzado. Espere 30 segundos antes de intentar llamar a sus servicios.

En un entorno de producción, sus servicios Eureka ya estarán ejecutándose y si está Al implementar un servicio existente, los servicios antiguos seguirán estando disponibles para aceptar solicitudes.

## 4.5 Uso del descubrimiento de servicios para buscar un servicio

Ya tienes el servicio de organización registrado en Eureka. También puedes tener el servicio de licencias llame al servicio de la organización sin tener conocimiento directo de la Ubicación de cualquiera de los servicios de la organización. El servicio de licencias buscará el Ubicación física de la organización mediante el uso de Eureka.

Para nuestros propósitos, veremos tres clientes Spring/Netflix diferentes. Bibliotecas en las que un consumidor de servicios puede interactuar con Ribbon. Estas bibliotecas pasar del nivel más bajo de abstracción para interactuar con Ribbon al más alto. Las bibliotecas que exploraremos incluyen

- Cliente Spring Discovery
- Cliente Spring Discovery habilitado RestTemplate
- Cliente Netflix Feign

Repasemos cada uno de estos clientes y veamos su uso en el contexto del servicio de licencia. Antes de comenzar con los detalles del cliente, escribí algunas recomendaciones clases y métodos en el código para que puedas jugar con los diferentes tipos de clientes usando el mismo punto final de servicio.

Primero, modifiqué el archivo src/main/java/com/thinktmechanix/licenses/controladores/LicenseServiceController.java para incluir una nueva ruta para el servicios de licencia. Esta nueva ruta te permitirá especificar el tipo de cliente que deseas invocar el servicio con. Esta es una ruta de ayuda para que, a medida que exploramos cada uno de los diferentes métodos para invocar el servicio de organización a través de Ribbon, usted pueda probar cada mecanismo a través de una única ruta. El siguiente listado muestra el código de la nueva ruta. en la clase LicenseServiceController .

### Listado 4.5 Llamar al servicio de licencias con diferentes Clientes REST

```
@RequestMapping(value="/{licenseId}/{clientType}",
    método = RequestMethod.GET) Licencia
pública getLicensesWithClient(
    @PathVariable("organizationId") Cadena ID de organización,
    @PathVariable("licenseId")           ID de licencia de cadena,
    @PathVariable("tipo de cliente")     Cadena tipo de cliente) {

    devolver LicenseService.getLicense(organizationId,
    ID de licencia, tipo de cliente);
}
```

clientType  
 determina el tipo de cliente Spring REST que se utilizará.

En este código, el parámetro clientType pasado en la ruta determinará el tipo de cliente que usaremos en los ejemplos de código. Los tipos específicos que puede transmitir en este ruta incluye

Discovery: utiliza el cliente de descubrimiento y una clase estándar Spring RestTemplate para invocar el servicio de organización

Rest: utiliza un Spring RestTemplate mejorado para invocar el servicio basado en Ribbon

Feign: utiliza la biblioteca cliente Feign de Netflix para invocar un servicio a través de Ribbon

**NOTA** Debido a que estoy usando el mismo código para los tres tipos de clientes, es posible que vea situaciones en las que verá anotaciones para ciertos clientes incluso cuando no parezcan ser necesarias. Por ejemplo, verá las anotaciones `@EnableDiscoveryClient` y `@EnableFeignClients` en el código, incluso cuando el texto solo explica uno de los tipos de cliente. Esto es para poder usar una base de código para mis ejemplos. Señalaré estas redundancias y codificaré cada vez que las encuentre.

En la clase `src/main/java/com/thinktmechanix/licenses/services/LicenseService.java`, agregué un método simple llamado `retrieveOrgInfo()` que resolverá, según el tipo de cliente pasado en la ruta, el tipo de cliente que utilizarse para buscar una instancia de servicio de la organización. El método `getLicense()` de la clase `LicenseService` utilizará `retrieveOrgInfo()` para recuperar los datos de la organización de la base de datos de Postgres.

#### Listado 4.6 La función `getLicense()` utilizará múltiples métodos para realizar una llamada REST

```
licencia pública getLicense(String organizaciónId, String licenciad, String clientType) {  
    Licencia licencia = LicenseRepository.findByOrganizationIdAndLicenseld( OrganizationId, Licenseld);  
  
    Organización de la organización = retrieveOrgInfo(organizationId, clientType);  
  
    licencia de devolución  
  
    .withOrganizationName( org.getName() ) .withContactName( org.getContactName() ) .withContactEmail( org.getContactEmail()  
}
```

Puede encontrar cada uno de los clientes que creamos utilizando las bibliotecas Spring `DiscoveryClient`, `Spring RestTemplate` o `Feign` en el paquete `src/main/java/com/thinktmechanix/licenses/clients` del código fuente del servicio de licencias.

#### 4.5.1 Buscar instancias de servicio con Spring `DiscoveryClient`

Spring `DiscoveryClient` ofrece el nivel más bajo de acceso a Ribbon y a los servicios registrados en él. Con `DiscoveryClient`, puede consultar todos los servicios registrados en el cliente de cinta y sus URL correspondientes.

A continuación, creará un ejemplo sencillo del uso de `DiscoveryClient` para recuperar una de las URL del servicio de la organización desde Ribbon y luego llamará al servicio mediante una clase `RestTemplate` estándar. Para comenzar a usar `DiscoveryClient`, primero debe anotar la clase `src/main/java/com/thinktmechanix/licenses/Application.java` con la anotación `@EnableDiscoveryClient`, como se muestra en el siguiente listado.

## Listado 4.7 Configurando la clase bootstrap para usar Eureka Discovery Client

```

@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
Aplicación de clase pública {
    público estático vacío principal (String [] argumentos) {
        SpringApplication.run(Aplicación.clase, argumentos);
    }
}

```

Activa la primavera DiscoveryClient para su uso  
Ignore esto por ahora ya que lo cubriremos más adelante en el capítulo.

La anotación `@EnableDiscoveryClient` es el disparador para que Spring Cloud habilite la aplicación para utilizar las bibliotecas `DiscoveryClient` y `Ribbon`. La anotación `@EnableFeignClients` se puede ignorar por ahora, ya que la cubriremos en breve.

Ahora, veamos su implementación del código que llama al servicio de la organización a través de `Spring DiscoveryClient`, como se muestra en el siguiente listado. Puedes encontrar esto en `src/main/java/com/thinktmechanix/licenses/OrganizationDiscoveryClient.java`.

## Listado 4.8 Uso de DiscoveryClient para buscar información

```

/*Paquetes e importaciones eliminados por motivos de concisión*/

@Component
clase pública OrganizationDiscoveryClient {

    @autocableado
    DiscoveryClient privado DiscoveryClient;

    Organización pública getOrganization (String organizaciónId) {
        RestTemplate restTemplate = nuevo RestTemplate();
        Lista de instancias <ServiceInstance> =
            DiscoveryClient.getInstances("servicio de organización");

        si (instances.size()==0) devuelve nulo;
        Cadena serviceUri = String.format("%s/v1/organizaciones/%s",
            instancias.get(0).getUri().toString(),
            ID de organización);

        ResponseEntity<Organización>restExchange =
            restoTemplate.exchange(
                servicioUri,
                Método Http.GET,
                nulo, Organización.clase, ID de organización);

        devolver restExchange.getBody();
    }
}

```

DiscoveryClient se inyecta automáticamente en la clase.  
Obtiene una lista de todas las instancias de servicios de la organización.  
Utiliza un resorte estándar Clase de plantilla REST para llamar al servicio.

recupera el servicio punto final nosotros van llamar

El primer elemento de interés en el código es DiscoveryClient. Esta es la clase que Úsela para interactuar con Ribbon. Para recuperar todas las instancias de los servicios de la organización registrados en Eureka, puede utilizar el método getInstances() , pasando la clave de servicio que está buscando, para recuperar una lista de objetos ServiceInstance .

La clase ServiceInstance se utiliza para contener información sobre una instancia específica. de un servicio, incluido su nombre de host, puerto y URI.

En el listado 4.8, toma la primera clase ServiceInstance de su lista para crear un objetivo. URL que luego se puede utilizar para llamar a su servicio. Una vez que tenga una URL de destino, puede usar un Spring RestTemplate estándar para llamar al servicio de su organización y recuperar datos.

### El DiscoveryClient y la vida real

Estoy recorriendo DiscoveryClient para completarlo en nuestra discusión sobre la construcción. consumidores de servicios con Ribbon. La realidad es que sólo debe utilizar Discovery-Client directamente cuando su servicio necesite consultar Ribbon para comprender qué servicios y las instancias de servicio están registradas en él. Hay varios problemas con esto código que incluye lo siguiente:

No está aprovechando el equilibrio de carga del lado del cliente de Ribbon. Al llamar directamente a DiscoveryClient, obtiene una lista de servicios, pero se convierte en su responsabilidad. para elegir qué instancias de servicio devueltas vas a invocar.

Estás trabajando demasiado. Ahora mismo tienes que crear la URL que será utilizado para llamar a su servicio. Es algo pequeño, pero cada fragmento de código que puedas evitar escribir es un fragmento de código menos que debes depurar.

Los desarrolladores observadores de Spring podrían haber notado que estás creando una instancia directa del Clase RestTemplate en el código. Esto es antítetico a las invocaciones normales de Spring REST, ya que normalmente Spring Framework inyectaría RestTemplate en la clase. usándolo a través de la anotación @Autowired.

Creaste una instancia de la clase RestTemplate en el listado 4.8 porque una vez que hayas habilitado Spring DiscoveryClient en la clase de aplicación a través de la anotación @EnableDiscovery-Client, todos los RestTemplates administrados por el marco Spring tendrán se les inyecta un interceptor habilitado para Ribbon que cambiará la forma en que se crean las URL con la clase RestTemplate. Creación de instancias directas de la clase RestTemplate le permite evitar este comportamiento.

En resumen, existen mejores mecanismos para llamar a un servicio respaldado por Ribbon.

#### 4.5.2 Invocar servicios con Spring RestTemplate compatible con cintas

A continuación, veremos un ejemplo de cómo utilizar un RestTemplate compatible con Ribbon. Este es uno de los mecanismos más comunes para interactuar con Ribbon a través de Primavera. Para utilizar una clase RestTemplate compatible con Ribbon , debe definir un método de construcción de bean Rest-Template con una anotación Spring Cloud llamada @Load-Balanced. Para el servicio de licencias, el método que se utilizará para crear el El frijol RestTemplate se puede encontrar en src/main/java/com/thinktmechanix/licencias/Applicación.java.

El siguiente listado muestra el método `getRestTemplate()` que creará el Frijol Spring RestTemplate con respaldo de cinta .

#### Listado 4.9 Anotar y definir un método de construcción RestTemplate

```
paquete com.thinkmechanix.licenses;

//...La mayoría de las declaraciones de importación se han eliminado por motivos de coherencia.
importar org.springframework.cloud.client.loadbalancer.LoadBalanced;
importar org.springframework.context.annotation.Bean;
importar org.springframework.web.client.RestTemplate;

@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
Aplicación de clase pública {

    @carga equilibrada
    @Frijol
    plantilla de descanso pública getRestTemplate(){
        devolver nuevo RestTemplate();
    }

    público estático vacío principal (String [] argumentos) {
        SpringApplication.run(Aplicación.clase, argumentos);
    }
}
```

← Debido a que utilizamos múltiples tipos de clientes en los ejemplos, estoy incluyéndolos en el código. Sin embargo, las aplicaciones `@EnableDiscoveryClient` y `@EnableFeignClients` no son necesarias cuando se utiliza `RestTemplate` respaldado por Ribbon y se pueden eliminar.

← La anotación `@LoadBalanced` le dice a Spring Cloud que cree una clase `RestTemplate` respaldada por Ribbon.

**NOTA** En las primeras versiones de Spring Cloud, Ribbon respaldaba automáticamente la clase `RestTemplate`. Era el comportamiento predeterminado. Sin embargo, desde la primavera Cloud Release Angel, `RestTemplate` en Spring Cloud ya no tiene respaldo por Cinta. Si desea utilizar Ribbon con `RestTemplate`, debe anotarlo explícitamente utilizando la anotación `@LoadBalanced`.

Ahora que está definida la definición de bean para la clase `RestTemplate` respaldada por Ribbon , cada vez que desee utilizar el bean `RestTemplate` para llamar a un servicio, solo necesita conéctelo automáticamente a la clase que lo usa.

El uso de la clase `RestTemplate` respaldada por Ribbon se comporta prácticamente como una clase Spring `RestTemplate` estándar , excepto por una pequeña diferencia en cómo se muestra la URL para El servicio de destino está definido. En lugar de utilizar la ubicación física del servicio en el Llamada `RestTemplate` , creará la URL de destino utilizando el ID del servicio Eureka del servicio al que desea llamar.

Veamos esta diferencia mirando el siguiente listado. El código de este listado. se puede encontrar en `src/main/java/com/thinktmechanix/licenses/ clientes/OrganizationRestTemplate.java` clase.

#### Listado 4.10 Uso de un `RestTemplate` respaldado por una cinta para llamar a un servicio

```
/*Las definiciones de paquete e importación se omitieron por motivos de concisión*/
@Component
```

```

clase pública OrganizationRestTemplateClient {
    @autocableado
    RestTemplate restTemplate;

    Organización pública getOrganization (String organizaciónId) {
        ResponseEntity<Organización> restExchange =
            restoTemplate.exchange(
                "http://organizationservice/v1/organizations/{organizationId}", HttpMethod.GET,
                nulo, Organización.clase, ID de organización);

        devolver restExchange.getBody();
    }
}

```

Cuando se utiliza una cinta trasera  
RestTemplate, construye el objetivo  
URL con el ID del servicio Eureka.

Este código debería ser algo similar al ejemplo anterior, excepto por dos claves.  
diferencias. Primero, Spring Cloud DiscoveryClient no está a la vista. Segundo,  
la URL que se utiliza en la llamada `restTemplate.exchange()` debería parecerle extraña:

```

restoTemplate.exchange(
    "http://organizationservice/v1/organizations/{organizationId}",
    Método Http.GET,
    nulo, Organización.clase, ID de organización);

```

El nombre del servidor en la URL coincide con el ID de la aplicación de la clave de servicio de la organización.  
con el que registró el servicio de organización en Eureka:

`http://{id de aplicación}/v1/organizaciones/{ID de organización}`

RestTemplate habilitado para Ribbon analizará la URL que se le pasa y utilizará lo que se pase como nombre  
del servidor como clave para consultar a Ribbon para obtener una instancia de un servicio. La ubicación real  
del servicio y el puerto están completamente abstractos del  
desarrollador.

Además, al utilizar la clase RestTemplate , Ribbon equilibrará la carga por turnos todas las solicitudes  
entre todas las instancias de servicio.

#### 4.5.3 Invocar servicios con el cliente Netflix Feign

Una alternativa a la clase RestTemplate habilitada para Spring Ribbon es Feign de Netflix  
biblioteca cliente. La biblioteca Feign adopta un enfoque diferente para llamar a un servicio REST mediante  
hacer que el desarrollador primero defina una interfaz Java y luego anotar esa interfaz  
con anotaciones de Spring Cloud para mapear qué servicio Ribbon basado en Eureka invocará.  
El marco Spring Cloud generará dinámicamente una clase de proxy que se utilizará  
para invocar el servicio REST de destino . No se está escribiendo ningún código para llamar al servicio que  
no sea una definición de interfaz.

Para habilitar el uso del cliente Feign en su servicio de licencias, debe agregar un nuevo  
anotación, `@EnableFeignClients`, al `src/main/java/` del servicio de licencias  
`com/thinkmechanix/licenses/Application.java` clase. El siguiente listado  
muestra este código.

## Listado 4.11 Habilitando el cliente Spring Cloud/Netflix Feign en el servicio de licencias

```

@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class Application {
    public static void main(String[] arguments) {
        SpringApplication.run(Application.class, arguments);
    }
}

```

Debido a que solo usamos FeignClient, en su propio código puede eliminar la anotación @EnableDiscoveryClient.

La anotación @EnableFeignClients es necesaria para utilizar FeignClient en su código.

Ahora que ha habilitado el uso del cliente Feign en su servicio de licencias, veamos una definición de interfaz de cliente Feign que se puede utilizar para llamar a un punto final en el servicio de la organización. El siguiente listado muestra un ejemplo. El código en este listado puede ser encontrado en src/main/java/com/thinkmechanix/licenses/clients/Clase OrganizationFeignClient.java .

## Listado 4.12 Definiendo una interfaz Feign para llamar al servicio de la organización

```

/*El paquete y la importación se omitieron por motivos de concisión*/
@FeignClient("organizationservice") interface pública
OrganizationFeignClient {
    @RequestMapping(
        método = RequestMethod.GET,
        value = "/v1/organizaciones/{organizationId}",
        consume = "aplicación/json")
    Organización getOrganización(
        @PathVariable("organizationId") String organizationId);
}

```

Identifique su servicio en Feign utilizando la anotación FeignClient.

El camino y la acción hacia tu El punto final se define utilizando la anotación @RequestMapping.

Los parámetros pasados al punto final son definido utilizando el punto final @PathVariable.

Comienza el ejemplo de Feign usando la anotación @FeignClient y pasándola el nombre de la identificación de la aplicación del servicio que desea que represente la interfaz. Próximo definirá un método, getOrganization(), en su interfaz que puede ser llamado por al cliente para invocar el servicio de organización.

La forma en que define el método getOrganization() se ve exactamente igual a como lo expondrá un punto final en una clase Spring Controller. Primero, vas a definir un Anotación @RequestMapping para el método getOrganization() que mapear el verbo HTTP y el punto final que se expondrán en el servicio de la organización invocación. En segundo lugar, asignará el ID de la organización pasado en la URL a un parámetro OrganizationId en la llamada al método, utilizando la anotación @PathVariable . El valor de retorno de la llamada al servicio de organización será automáticamente asignado a la clase de organización que se define como el valor de retorno para el método getOrganization() .

Para utilizar la clase OrganizationFeignClient , todo lo que necesita hacer es autowire y úsalo. El código de Feign Client se encargará de todo el trabajo de codificación por usted.

#### Sobre el manejo de errores

Cuando utiliza la clase estándar Spring RestTemplate, los códigos de estado HTTP de todas las llamadas de servicio se devolverán a través de getStatusCode() de la clase ResponseEntity.

método. Con Feign Client, cualquier código de estado HTTP 4xx – 5xx devuelto por el servicio al que se llama se asignará a una FeignException. La FeignException contiene un cuerpo JSON que se puede analizar para el mensaje de error específico.

Feign le brinda la posibilidad de escribir una clase decodificadora de errores que mapeará el error de nuevo a una clase de excepción personalizada. Escribir este decodificador está fuera del alcance. de este libro, pero puedes encontrar ejemplos de esto en el repositorio Feign GitHub en (<https://github.com/Netflix/feign/wiki/Custom-error-handling>).

## 4.6 Resumen

El patrón de descubrimiento de servicios se utiliza para abstraer la ubicación física de servicios.

Un motor de descubrimiento de servicios como Eureka puede agregar y eliminar sin problemas instancias de servicios de un entorno sin que los clientes del servicio se vean afectados.

El equilibrio de carga del lado del cliente puede proporcionar un nivel adicional de rendimiento y resiliencia al almacenar en caché la ubicación física de un servicio en el cliente que realiza la llamada de servicio.

Eureka es un proyecto de Netflix que, cuando se usa con Spring Cloud, es fácil de configurar. y configurar.

Utilizó tres mecanismos diferentes en Spring Cloud, Netflix Eureka y Netflix Ribbon para invocar un servicio. Estos mecanismos incluían

- Usando un servicio de Spring Cloud DiscoveryClient
- Uso de Spring Cloud y RestTemplate respaldado por cinta
- Usando Spring Cloud y el cliente Feign de Netflix



# Cuando suceden cosas malas: cliente patrones de resiliencia con Spring Nube y Netflix Hystrix

## Este capítulo cubre

Implementar disyuntores, alternativas y  
mamparos

Uso del patrón de disyuntor para conservar los recursos del  
cliente de microservicio

Usar Hystrix cuando un servicio remoto falla

Implementación del patrón de mamparo de Hystrix para  
segregar llamadas de recursos remotos

Ajuste de las implementaciones de disyuntores y mamparas  
de Hystrix

Personalización de la estrategia de concurrencia de Hystrix

Todos los sistemas, especialmente los distribuidos, experimentarán fallas. ¿Cómo construimos nuestro aplicaciones para responder a esa falla es una parte crítica del trabajo de cada desarrollador de software. trabajo. Sin embargo, cuando se trata de construir sistemas resilientes, la mayoría de los ingenieros de software Sólo se tiene en cuenta el fallo total de una pieza de infraestructura o de un servicio clave. Se centran en crear redundancia en cada capa de su aplicación utilizando técnicas como la agrupación de servidores clave, el equilibrio de carga entre servicios y la segregación de la infraestructura en múltiples ubicaciones.

Si bien estos enfoques tienen en cuenta la completa (y a menudo espectacular) pérdida de un componente del sistema, abordan sólo una pequeña parte de la construcción de sistemas resilientes. Cuando un servicio falla, es fácil detectar que ya no está allí y la aplicación puede evitarlo. Sin embargo, cuando un servicio funciona lento, detectar que un rendimiento deficiente y evitarlo es extremadamente difícil porque

- 1 La degradación de un servicio puede comenzar de forma intermitente y ganar impulso. La degradación puede ocurrir sólo en pequeñas ráfagas. Los primeros signos de fracaso podrían ser un Un pequeño grupo de usuarios se queja de un problema, hasta que de repente el contenedor de aplicaciones agota su grupo de subprocessos y colapsa por completo.
- 2 Las llamadas a servicios remotos suelen ser sincrónicas y no interrumpen una llamada de larga duración. llamada: la persona que llama a un servicio no tiene el concepto de tiempo de espera para mantener la llamada de servicio. de pasar el rato para siempre. El desarrollador de la aplicación llama al servicio para realizar una acción y espera a que regrese el servicio.
- 3 Las aplicaciones suelen estar diseñadas para hacer frente a fallos completos de recursos remotos, no parciales. degradaciones. A menudo, mientras el servicio no haya fallado por completo, una aplicación Continuará llamando al servicio y no fallará rápidamente. La aplicación continuará llamar al servicio que se porta mal. La aplicación o servicio de llamada puede degradarse con gracia o, más probablemente, colapsar debido al agotamiento de los recursos. El agotamiento de recursos se produce cuando un recurso limitado, como un grupo de subprocessos o una conexión de base de datos, llega al máximo y el cliente que llama debe esperar a que ese recurso esté disponible.

Lo insidioso de los problemas causados por servicios remotos de bajo rendimiento es que no sólo son difíciles de detectar, sino que también pueden desencadenar un efecto en cascada que puede afectar en todo un ecosistema de aplicaciones. Sin salvaguardias establecidas, un solo servicio con un rendimiento deficiente puede desactivar rápidamente varias aplicaciones. basado en la nube, Las aplicaciones basadas en microservicios son particularmente vulnerables a este tipo de interrupciones. porque estas aplicaciones se componen de una gran cantidad de servicios distribuidos y detallados con diferentes piezas de infraestructura involucradas en completar la tarea de un usuario. transacción.

## 5.1 ¿Qué son los patrones de resiliencia del lado del cliente?

Los patrones de software de resiliencia del cliente se centran en proteger la capacidad de un recurso remoto. (otra llamada de microservicio o búsqueda de base de datos) el cliente se bloquee cuando el control remoto El recurso está fallando porque ese servicio remoto arroja errores o realiza mal. El objetivo de estos patrones es permitir que el cliente "falle rápidamente", no consuma recursos valiosos como conexiones de bases de datos y grupos de subprocessos, y evitar que el problema del servicio remoto se propague "ascendente" a los consumidores del cliente. .

Hay cuatro patrones de resiliencia del cliente:

- 1 Equilibrio de carga del lado del cliente
- 2 disyuntores
- 3 alternativas
- 4 mamparas

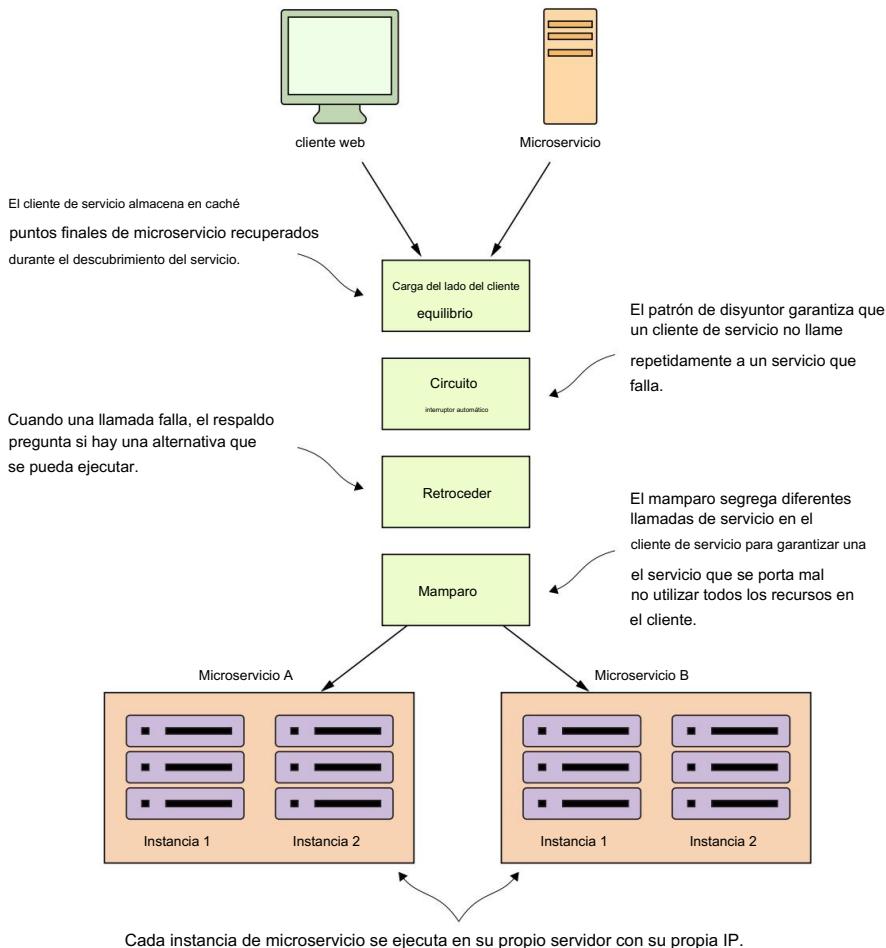


Figura 5.1 Los cuatro patrones de resiliencia del cliente actúan como un amortiguador protector entre el consumidor del servicio y el servicio.

La Figura 5.1 demuestra cómo estos patrones se ubican entre los servicios de microservicio Sumer y el microservicio.

Estos patrones se implementan en el cliente que llama al recurso remoto. La implementación de estos patrones lógicamente se ubica entre el cliente que consume el recurso remoto y el recurso mismo.

### 5.1.1 Equilibrio de carga del lado del cliente

Introducimos el patrón de equilibrio de carga del lado del cliente en el último capítulo (capítulo 4), cuando se habla de descubrimiento de servicios. El equilibrio de carga del lado del cliente implica tener la lista de las instancias individuales de un servicio desde un agente de descubrimiento de servicios (como Netflix Eureka) y luego almacenar en caché la ubicación física de dichas instancias de servicio.

Siempre que un consumidor de servicios necesite llamar a esa instancia de servicio, el balanceador de carga del lado del cliente devolverá una ubicación del conjunto de ubicaciones de servicios que mantiene.

Debido a que el balanceador de carga del lado del cliente se encuentra entre el cliente del servicio y el servicio consumidor, el equilibrador de carga puede detectar si una instancia de servicio arroja errores o se comporta mal. Si el equilibrador de carga del lado del cliente detecta un problema, puede eliminar esa instancia de servicio del grupo de ubicaciones de servicio disponibles y evitar que futuras llamadas de servicio lleguen a esa instancia de servicio.

Este es exactamente el comportamiento que las bibliotecas Ribbon de Netflix brindan de forma inmediata y sin configuración adicional. Debido a que cubrimos el equilibrio de carga del lado del cliente con Netflix Ribbon en el capítulo 4, no entraremos en más detalles al respecto en este capítulo.

### 5.1.2 Disyuntor

El patrón de disyuntor es un patrón de resiliencia del cliente que se modela a partir de un disyuntor eléctrico. En un sistema eléctrico, un disyuntor detectará si fluye demasiada corriente a través del cable. Si el disyuntor detecta un problema, interrumpirá la conexión con el resto del sistema eléctrico y evitará que los componentes posteriores se quemen.

Con un disyuntor de software, cuando se llama a un servicio remoto, el disyuntor monitoreará la llamada. Si las llamadas tardan demasiado, el disyuntor intervendrá y cancelará la llamada. Además, el disyuntor monitoreará todas las llamadas a un recurso remoto y, si fallan suficientes llamadas, la implementación del disyuntor saltará, fallando rápidamente y evitando futuras llamadas al recurso remoto que falla.

### 5.1.3 Procesamiento alternativo

Con el patrón alternativo, cuando falla una llamada de servicio remoto, en lugar de generar una excepción, el consumidor del servicio ejecutará una ruta de código alternativa e intentará llevar a cabo una acción por otros medios. Por lo general, esto implica buscar datos de otra fuente de datos o poner en cola la solicitud del usuario para su procesamiento futuro. La llamada del usuario no mostrará una excepción que indique un problema, pero se le podrá notificar que su solicitud deberá cumplirse en una fecha posterior.

Por ejemplo, supongamos que tiene un sitio de comercio electrónico que monitorea el comportamiento de sus usuarios e intenta darles recomendaciones sobre otros artículos que podrían comprar. Normalmente, puede llamar a un microservicio para ejecutar un análisis del comportamiento anterior del usuario y devolver una lista de recomendaciones adaptadas a ese usuario específico. Sin embargo, si el servicio de preferencias falla, su alternativa podría ser recuperar una lista más general de preferencias que se base en todas las compras de los usuarios y sea mucho más generalizada. Estos datos pueden provenir de un servicio y una fuente de datos completamente diferentes.

### 5.1.4 Mamparos

El patrón de mamparo se basa en un concepto de la construcción de barcos. Con un diseño de mamparo, un barco se divide en compartimentos completamente segregados y estancos llamados mamparos. Incluso si el casco del barco está perforado, porque el barco está dividido en

compartimentos estancos (mamparas), la mampara mantendrá el agua confinada a la zona del barco donde se produjo el pinchazo y evitar que todo el barco llenándose de agua y hundiéndose.

El mismo concepto se puede aplicar a un servicio que debe interactuar con múltiples recursos remotos. Al utilizar el patrón de mamparo, puede dividir las llamadas a control remoto recursos en sus propios grupos de subprocesos y reducir el riesgo de que surja un problema con uno de ellos. una llamada lenta a un recurso remoto desactivará toda la aplicación. Los grupos de hilos actúan como mamparas para su servicio. Cada recurso remoto está segregado y asignado a el grupo de subprocesos. Si un servicio responde lentamente, el grupo de subprocesos para ese tipo de llamadas de servicio se saturarán y dejarán de procesar solicitudes. Llamadas de servicio a otros servicios no se saturarán porque están asignados a otros grupos de subprocesos.

## 5.2 Por qué es importante la resiliencia del cliente

Hemos hablado de estos diferentes patrones en abstracto; sin embargo, profundicemos en un ejemplo más específico de dónde se pueden aplicar estos patrones. Caminemos por un Escenario común con el que me he topado y veo por qué los patrones de resiliencia del cliente, como el patrón de disyuntor, son críticos para implementar una arquitectura basada en servicios, particularmente una arquitectura de microservicios que se ejecuta en la nube.

En la figura 5.2, muestro un escenario típico que involucra el uso de un recurso remoto como un Base de datos y servicio remoto.

En el escenario de la figura 5.2, tres aplicaciones se comunican de una manera u otro con tres servicios diferentes. Las aplicaciones A y B se comunican directamente con el Servicio A. El Servicio A recupera datos de una base de datos y llama al Servicio B para realizar el trabajo. para ello. El servicio B recupera datos de una plataforma de base de datos completamente diferente y llama a otro servicio, el Servicio C, de un proveedor de nube externo cuyo servicio depende en gran medida de un dispositivo interno de almacenamiento de área de red (NAS) para escribir datos en un sistema de archivos compartido. Además, la Aplicación C llama directamente al Servicio C.

Durante el fin de semana, un administrador de red hizo lo que pensó que era un pequeño modifique la configuración en el NAS, como se muestra en negrita en la figura 5.2. Este cambio Parece funcionar bien, pero el lunes por la mañana, cualquier lectura de un subsistema de disco en particular comienza a funcionar de manera extremadamente lenta.

El desarrollador que escribió el Servicio B nunca anticipó que se producirían ralentizaciones con llamadas al Servicio C. Escribieron su código para que las escrituras en su base de datos y el las lecturas del servicio ocurren dentro de la misma transacción. Cuando el Servicio C comienza a ejecutarse funcionando lentamente, no solo el grupo de subprocesos para solicitudes al Servicio C comienza a realizar copias de seguridad, el número de conexiones de bases de datos en los grupos de conexiones del contenedor de servicios se agotan porque estas conexiones se mantienen abiertas porque las llamadas El servicio C nunca se completa.

Finalmente, el Servicio A comienza a quedarse sin recursos porque está llamando al Servicio B. que funciona lento debido al Servicio C. Finalmente, las tres aplicaciones se detienen respondiendo porque se quedan sin recursos mientras esperan que se completen las solicitudes.

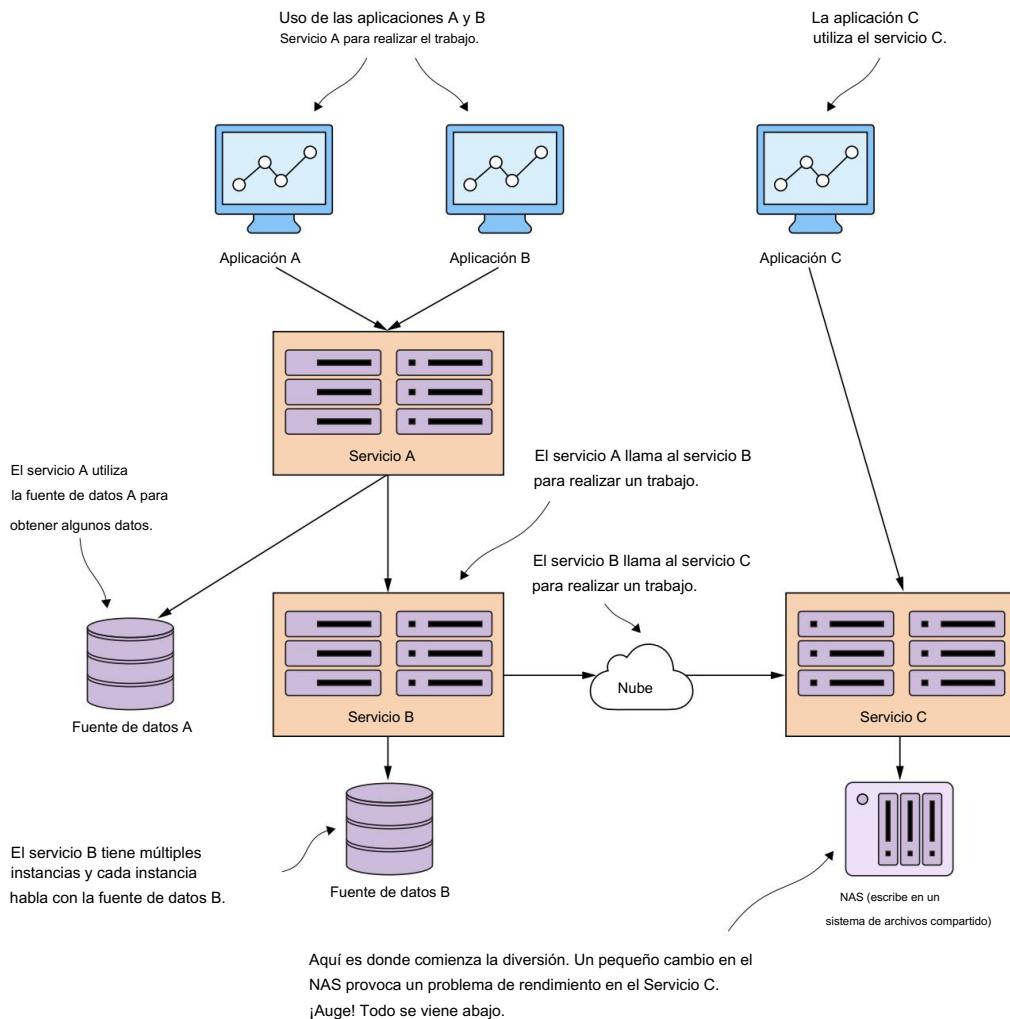


Figura 5.2 Una aplicación es un gráfico de dependencias interconectadas. Si no gestiona las llamadas remotas entre estos, un recurso remoto que se comporte mal puede provocar la caída de todos los servicios del gráfico.

Todo este escenario podría evitarse si se hubiera implementado un patrón de disyuntor en cada punto donde se hubiera llamado a un recurso distribuido (ya sea una llamada a la base de datos o una llamada al servicio). En la figura 5.2, si la llamada al Servicio C hubiera sido implementado con un disyuntor, luego, cuando el servicio C comenzó a funcionar mal, el disyuntor para esa llamada específica al Servicio C se habría disparado y Falló rápidamente sin comerse un hilo. Si el Servicio B tuviera múltiples puntos finales, solo el Los puntos finales que interactuaron con esa llamada específica al Servicio C se verían afectados. El El resto de la funcionalidad del Servicio B aún estaría intacta y podría satisfacer las solicitudes de los usuarios.

Un disyuntor actúa como intermediario entre la aplicación y el servicio remoto. En el escenario anterior, la implementación de un disyuntor podría haber protegido las aplicaciones A, B y C para que no fallaran por completo.

En la figura 5.3, el Servicio B (el cliente) nunca invocará directamente el Servicio C. En cambio, cuando se realiza la llamada, el Servicio B delegará la invocación real del servicio al disyuntor, que tomará la llamada y la envolverá en un subproceso (generalmente administrado por un grupo de subprocessos) que es independiente del persona que llama de origen. Al envolver la llamada en un hilo, el cliente ya no está esperando directamente a que se complete la llamada. En cambio, el disyuntor monitorea el subproceso y puede cancelar la llamada si el subprocesso se prolonga demasiado.

En la figura 5.3 se muestran tres escenarios. En el primer escenario, el camino feliz, el disyuntor mantendrá un temporizador y si la llamada al servicio remoto se completa antes de que se acabe el temporizador, todo está bien y el Servicio B puede continuar su trabajo. En el escenario de degradación parcial, el Servicio B llamará al Servicio C a través del disyuntor. Esta vez, sin embargo, el Servicio C está funcionando lento y el disyuntor cortará la conexión al servicio remoto si no se completa antes de que se agote el tiempo del temporizador en el subprocesso mantenido por el disyuntor.

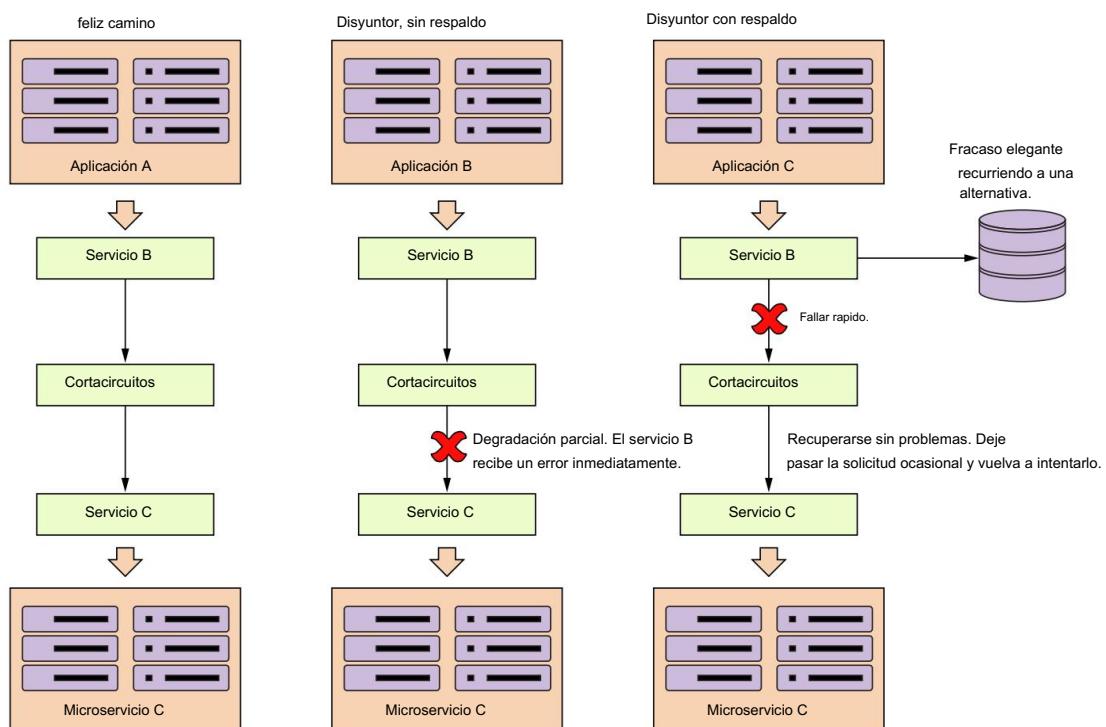


Figura 5.3 El disyuntor se dispara y permite que una llamada de servicio que se comporta mal falle de forma rápida y elegante.

El servicio B recibirá un error al realizar la llamada, pero el servicio B no tendrá recursos (es decir, sus propios subprocesos o grupos de conexiones) inmovilizados esperando que el Servicio C completo. Si el disyuntor agota el tiempo de espera de la llamada al Servicio C, el circuito El disyuntor comenzará a rastrear la cantidad de fallas que han ocurrido.

Si se han producido suficientes errores en el servicio dentro de un período de tiempo determinado, el disyuntor ahora "disparará" el circuito y todas las llamadas al Servicio C fallarán sin llamar. Servicio C.

Este disparo del circuito permite que ocurran tres cosas:

- 1 El servicio B ahora sabe inmediatamente que hay un problema sin tener que esperar a que tiempo de espera del disyuntor.
- 2 El servicio B ahora puede elegir entre fallar completamente o tomar medidas usando una conjunto de código nativo (un recurso alternativo).
- 3 El servicio C tendrá la oportunidad de recuperarse porque el servicio B no llama mientras el disyuntor se ha disparado. Esto permite que el Servicio C tenga espacio para respirar y ayuda a prevenir la muerte en cascada que ocurre cuando ocurre una degradación del servicio.

Finalmente, el disyuntor ocasionalmente permitirá que las llamadas lleguen a un servicio degradado, y si esas llamadas tienen éxito suficientes veces seguidas, el disyuntor se restablecerá solo.

Lo clave que ofrece un patrón de interrupción de circuito es la capacidad de realizar llamadas remotas

- 1 Falla rápida: cuando un servicio remoto experimenta una degradación, la aplicación fallará rápidamente y evitará problemas de agotamiento de recursos que normalmente cierran el toda la aplicación. En la mayoría de situaciones de apagón, es mejor estar parcialmente caído en lugar de completamente abajo.
- 2 Falla con gracia: al agotarse el tiempo y fallar rápidamente, el patrón del disyuntor proporciona el desarrollador de la aplicación la capacidad de fallar con gracia o buscar mecanismos alternativos para llevar a cabo la intención del usuario. Por ejemplo, si un usuario intenta recuperar datos de una fuente de datos, y esa fuente de datos está experimentando una degradación del servicio, entonces el desarrollador de la aplicación podría intentar recuperar esos datos de otra localización.
- 3 Recuperarse sin problemas: con el patrón del interruptor actuando como intermediario, El disyuntor puede comprobar periódicamente si el recurso solicitado vuelve a estar en línea y vuelve a habilitar el acceso a él sin intervención humana.

En una gran aplicación basada en la nube con cientos de servicios, esta elegante recuperación es crítico porque puede reducir significativamente la cantidad de tiempo necesario para restaurar servicio y reducir significativamente el riesgo de que un operador o ingeniero de aplicaciones se canse causando mayores problemas al hacer que intervengan directamente (reiniciando un servicio fallido) en el restablecimiento del servicio.

## 5.3 Ingrese a Hystrix

Implementaciones de construcción de los patrones de disyuntor, respaldo y mamparo.  
Requiere un conocimiento profundo de los subprocesos y la gestión de subprocesos. Seamos realistas, escribiendo

El código de subprocessamiento robusto es un arte (que nunca he dominado) y hacerlo correctamente es difícil. Para implementar un conjunto de implementaciones de alta calidad para el interruptor automático, el respaldo y los patrones de mamparos requerirían una enorme cantidad de trabajo. Afortunadamente, puede utilizar Spring Cloud y la biblioteca Hystrix de Netflix para proporcionarle una biblioteca probada en batalla que se utiliza a diario en la arquitectura de microservicios de Netflix.

En las siguientes secciones de este capítulo cubriremos cómo

Configure el archivo de compilación maven del servicio de licencias (pom.xml) para incluir el Envoltorios Spring Cloud/Hystrix.

Utilice las anotaciones de Spring Cloud/Hystrix para envolver llamadas remotas con un patrón de interruptor de corte.

Personalice los disyuntores individuales en un recurso remoto para usar

tiempos de espera para cada llamada realizada. También demostraré cómo configurar el circuito. disyuntores para que pueda controlar cuántas fallas ocurren antes de que un disyuntor "excursiones."

Implementar una estrategia de respaldo en caso de que un disyuntor tenga que interrumpir una llamada o la llamada falla.

Utilice grupos de subprocessos individuales en su servicio para aislar las llamadas de servicio y crear mamparos entre diferentes recursos remotos que se llaman.

## 5.4 Configurar el servidor de licencias para usar Nube de primavera y Hystrix

Para comenzar nuestra exploración de Hystrix, debe configurar su proyecto pom.xml para importar las dependencias de Spring Hystrix. Tomarás tu servicio de licencias que hemos estado compilando y modificando su pom.xml agregando las dependencias de maven para Hystrix:

```
<dependencia>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependencia>
<dependencia>
    <groupId>com.netflix.hystrix</groupId> <artifactId>hystrix-
    javanica</artifactId>
    <versión>1.5.9</versión>
</dependencia>
```

La primera etiqueta `<dependencia>` (`spring-cloud-starter-hystrix`) le dice a Maven que baje las dependencias de Spring Cloud Hystrix. Esta segunda etiqueta `<dependency>` (`hystrix-javanica`) desplegará las bibliotecas principales de Netflix Hystrix. Con las dependencias de Maven configuradas, puede continuar y comenzar su implementación de Hystrix utilizando los servicios de organización y licencias que creó en los capítulos anteriores.

**NOTA** No es necesario incluir las dependencias de `hystrix-javanica`. directamente en el pom.xml. De forma predeterminada, `spring-cloud-starter-hystrix` incluye una versión de las dependencias `hystrix-javanica`. El Camden.SR5 La publicación del libro utilizó `hystrix-javanica-1.5.6`. la versión de

A hystrix-javanica se le introdujo una inconsistencia que provocó que el código Hystrix sin respaldo arrojara un `java.lang.reflect.UndeclaredThrowableException` en lugar de `com.netflix.hystrix.exception.HystrixRuntimeException`. Este fue un cambio radical para muchos desarrolladores que usaban versiones anteriores de Hystrix. Las bibliotecas hystrix-javanica solucionaron este problema en versiones posteriores, por lo que utilicé deliberadamente una versión posterior de hystrix-javanica en lugar de usar la versión predeterminada obtenida por Spring Cloud.

Lo último que debe hacer antes de poder comenzar a usar los disyuntores Hystrix dentro del código de su aplicación es anotar la clase de arranque de su servicio con la anotación `@EnableCircuitBreaker`. Por ejemplo, para el servicio de licencias, agregaría la anotación `@EnableCircuitBreaker` a la clase `licensing-service/src/main/java/com/thinkmechanix/licenses/Application.java`. La siguiente lista muestra este código.

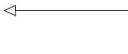
**Listado 5.1 La anotación `@EnableCircuitBreaker` utilizada para activar Hystrix en un servicio**

```
paquete com.thinkmechanix.licenses

importar org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker; //El resto de las importaciones se
eliminaron por motivos de concisión.

@SpringBootAplicación
@EnableEurekaClient
@EnableCircuitBreaker
Aplicación de clase pública {
    @carga equilibrada
    @Friol
    public RestTemplate restTemplate() { return new
        RestTemplate();
    }

    public static void main(String[] args)
    { SpringApplication.run(Application.class, args);
    }
}
```



Le dice a Spring Cloud que usará Hystrix para su servicio

**NOTA** Si olvida agregar la anotación `@EnableCircuitBreaker` a su clase de arranque, ninguno de sus disyuntores Hystrix estará activo. No recibirá ningún mensaje de advertencia o error cuando se inicie el servicio.

## 5.5 Implementación de un disyuntor usando Hystrix

Analizaremos la implementación de Hystrix en dos categorías amplias. En la primera categoría, envolverá todas las llamadas a su base de datos en el servicio de organización y licencia con un disyuntor Hystrix. Luego, empaquetará las llamadas entre servicios entre el servicio de licencias y el servicio de la organización utilizando Hystrix. Mientras estos

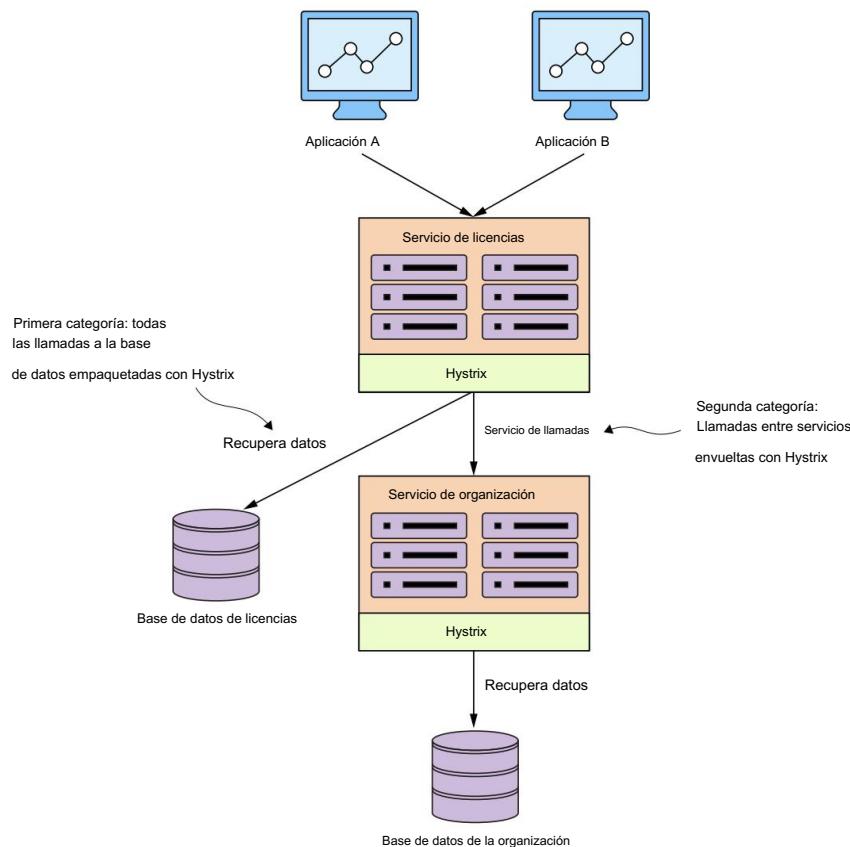


Figura 5.4 Hystrix se ubica entre cada llamada de recurso remoto y protege al cliente. No importa si la llamada al recurso remoto es una llamada a la base de datos o una llamada al servicio basado en REST.

Hay dos categorías de llamadas diferentes, verás que el uso de Hystrix será exactamente el mismo. La Figura 5.4 muestra qué recursos remotos va a proteger con un disyuntor Hystrix.

Comencemos nuestra discusión sobre Hystrix mostrando cómo finalizar la recuperación de licencias. Los datos de servicio de la base de datos de licencias mediante un disyuntor Hystrix síncrono. Con una llamada sincrónica, el servicio de licencias recuperará sus datos pero esperará a que Declaración SQL para completar o para un tiempo de espera del disyuntor antes de continuar Procesando.

Hystrix y Spring Cloud usan la anotación `@HystrixCommand` para marcar Java Los métodos de clase son administrados por un disyuntor Hystrix. Cuando el marco Spring ve `@HystrixCommand`, generará dinámicamente un proxy que envolverá el método y administrará todas las llamadas a ese método a través de un grupo de subprocessos. reservado específicamente para manejar llamadas remotas.

Vas a incluir el método `getLicensesByOrg()` en tu licencia.  
 servicio/src/main/java/com/thinkmechanix/licenses/services/  
 Clase LicenseService.java , como se muestra en el siguiente listado.

Listado 5.2 Encapsulando una llamada de recurso remoto con un disyuntor

```
//Importaciones eliminadas por razones de concisión
@HystrixCommand lista
pública<Licencia> getLicensesByOrg(String organizationId){
    devolver LicenseRepository.findByOrganizationId(organizationId);
}
```

La anotación `@HystrixCommand` se utiliza para envuelve el método `getLicenseByOrg()` con un disyuntor Hystrix.

**NOTA** Si observa el código en el listado 5.2 en el repositorio de código fuente, Verá varios parámetros más en la anotación `@HystrixCommand` que lo que se muestra en el listado anterior. Entraremos en esos parámetros. más adelante en el capítulo. El código del listado 5.2 utiliza `@HystrixCommand` anotación con todos sus valores predeterminados.

Esto no parece mucho código, y no lo es, pero hay mucha funcionalidad. dentro de esta anotación. Con el uso de la anotación `@HystrixCommand` , cualquier Cada vez que se llama al método `getLicensesByOrg()` , la llamada se envolverá con un disyuntor Hystrix. El disyuntor interrumpirá cualquier llamada al método `getLicenses-ByOrg()` cada vez que la llamada demore más de 1000 milisegundos.

Este ejemplo de código sería aburrido si la base de datos funcionara correctamente. Simulemos el método `getLicensesByOrg()` ejecutándose en una consulta de base de datos lenta teniendo la llamada dura un poco más de un segundo en aproximadamente cada una de cada tres llamadas. La siguiente lista lo demuestra.

Listado 5.3 Temporización aleatoria de una llamada a la base de datos del servicio de licencias

```
vacío privado aleatoriamenteRunLong(){
    Rand aleatorio = nuevo Aleatorio();
    int número aleatorio = rand.nextInt((3 - 1) + 1) + 1;
    si (numeroaleatorio==3) dormir();
}

sueño privado vacío(){
    intentar {
        Hilo.sleep(11000); } captura
        (Excepción interrumpida e) {
            e.printStackTrace();
    }
}

@HystrixComando
```

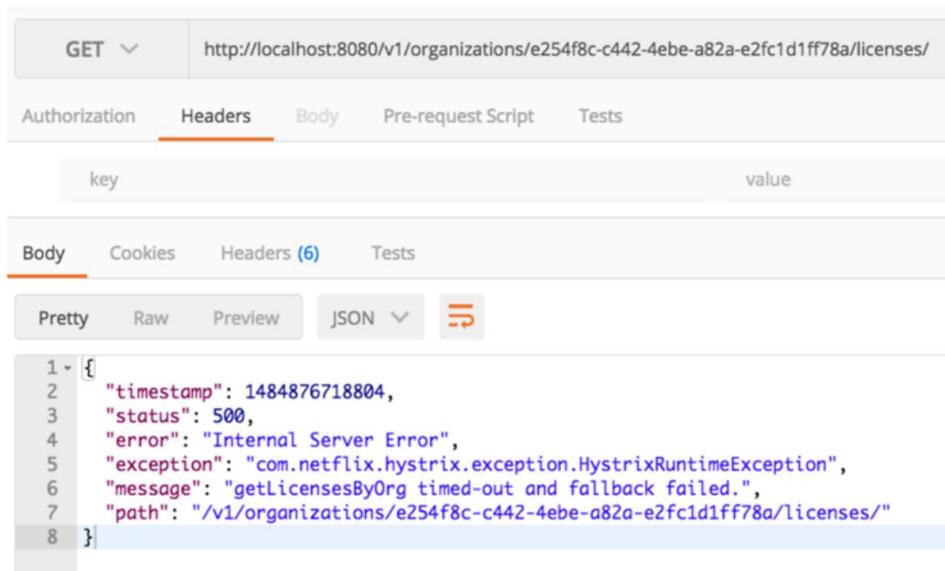
El método `randomlyRunLong()` le ofrece una probabilidad entre tres de que una llamada a la base de datos dure mucho tiempo.

Duermes durante 11.000 milisegundos (11 segundos). Comportamiento predeterminado de Hystrix es cronometrar una llamada después de 1 segundo.

```
Lista pública<Licencia> getLicensesByOrg(String organizaciónId){  
    aleatoriamenteRunLong();  
  
    devolver LicenseRepository.findByOrganizationId(organizationId);  
}
```

Si presiona el punto final `http://localhost/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/` suficientes veces, debería ver un tiempo de espera.

Mensaje de error devuelto por el servicio de licencias. La figura 5.5 muestra este error.



The screenshot shows a POST request to `http://localhost:8080/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/`. The Headers tab is selected, showing a key-value pair for 'Content-Type' with a value of 'application/json'. The Body tab is selected, showing a JSON response with a timestamp, status, error, exception, message, and path. The JSON is formatted as follows:

```
1 - {  
2   "timestamp": 1484876718804,  
3   "status": 500,  
4   "error": "Internal Server Error",  
5   "exception": "com.netflix.hystrix.exception.HystrixRuntimeException",  
6   "message": "getLicensesByOrg timed-out and fallback failed.",  
7   "path": "/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/"  
8 }
```

Figura 5.5 Se genera una excepción `HystrixRuntimeException` cuando una llamada remota tarda demasiado.

Ahora, con la anotación `@HystrixCommand` implementada, el servicio de licencias interrumpirá una llamada a su base de datos si la consulta tarda demasiado. Si las llamadas a la base de datos tardan más de 1000 milisegundos en ejecutar el ajuste del código Hystrix, su llamada de servicio lanzará una excepción `com.netflix.hystrix.exception.HystrixRuntimeException` excepción.

#### 5.5.1 Temporización de una llamada al microservicio de la organización

La belleza de usar anotaciones a nivel de método para etiquetar llamadas con disyuntor. El comportamiento es que es la misma anotación ya sea que acceda a una base de datos o llame a un microservicio.

Por ejemplo, en su servicio de licencias debe buscar el nombre de la organización asociada con la licencia. Si desea finalizar su llamada a la organización

servicio con un disyuntor, es tan simple como dividir la llamada RestTemplate en su propio método y anotarlo con la anotación @HystrixCommand :

```
@HystrixCommand
organización privada getOrganization(String organizaciónId) {
    devolver organizaciónRestClient.getOrganization(organizationId); }
```

**NOTA** Si bien el uso de @HystrixCommand es fácil de implementar, debe tener cuidado al usar la anotación @HystrixCommand predeterminada sin configuración en la anotación. De forma predeterminada, cuando especifica una anotación @HystrixCommand sin propiedades, la anotación colocará todas las llamadas de servicio remoto en el mismo grupo de subprocessos. Esto puede introducir problemas en su aplicación. Más adelante en el capítulo, cuando hablemos sobre la implementación del patrón de tabique, le mostraremos cómo separar estas llamadas de servicio remoto en sus propios grupos de subprocessos y configurar el comportamiento de los grupos de subprocessos para que sean independientes entre sí.

### 5.5.2 Personalización del tiempo de espera en un disyuntor

Una de las primeras preguntas con las que me encuentro a menudo cuando trabajo con nuevos desarrolladores y Hystrix es cómo pueden personalizar la cantidad de tiempo antes de que Hystrix interrumpe una llamada. Esto se logra fácilmente pasando parámetros adicionales a la anotación @HystrixCommand . La siguiente lista muestra cómo personalizar la cantidad de tiempo que Hystrix espera antes de finalizar una llamada.

Listado 5.4 Personalización del tiempo de espera en una llamada de disyuntor

```
@HystrixCommand( propiedades del comando=
    {@PropiedadHystrix(
        name="execution.isolation.thread.timeoutInMillisegundos", valor="12000"}) lista
    pública<Licencia>
    getLicensesByOrg(String organizaciónId){ randomlyRunLong();

    devolver LicenseRepository.findByOrganizationId(organizationId);
}
```

El atributo commandProperties le permite proporcionar propiedades adicionales para personalizar Hystrix.

Execution.isolation.thread.timeoutInMillisegundos se utiliza para establecer la duración del tiempo de espera (en milisegundos) del disyuntor.

Hystrix le permite personalizar el comportamiento del disyuntor a través del atributo commandProperties . El atributo commandProperties acepta una serie de objetos HystrixProperty que pueden pasar propiedades personalizadas para configurar el disyuntor Hystrix. En el listado 5.4, utiliza la propiedad ejecución.isolation.thread .timeoutInMillisegundos para establecer el tiempo de espera máximo que esperará una llamada Hystrix antes de fallar en 12 segundos.

Ahora, si reconstruye y vuelve a ejecutar el ejemplo de código, nunca obtendrá un error de tiempo de espera porque su tiempo de espera artificial en la llamada es de 11 segundos, mientras que su anotación @HystrixCommand ahora está configurada para que solo expire después de 12 segundos.

#### Sobre los tiempos de espera

[del servicio](#) Debería ser obvio que estoy usando un tiempo de espera del disyuntor de 12 segundos como ejemplo didáctico. En un entorno distribuido, a menudo me pongo nervioso si empiezo a escuchar comentarios de los equipos de desarrollo de que un tiempo de espera de 1 segundo en llamadas de servicio remoto es demasiado bajo porque su servicio X demora en promedio entre 5 y 6 segundos.

Por lo general, esto me indica que existen problemas de rendimiento no resueltos con el servicio al que se llama. Evite la tentación de aumentar el tiempo de espera predeterminado en las llamadas de Hystrix a menos que no pueda resolver en absoluto una llamada de servicio que funciona con lentitud.

Si tiene una situación en la que parte de sus llamadas de servicio van a tardar más que otras llamadas de servicio, definitivamente considere segregar estas llamadas de servicio en grupos de subprocesos separados.

## 5.6 Procesamiento alternativo Parte de la

belleza del patrón de disyuntor es que debido a que un "intermediario" se encuentra entre el consumidor de un recurso remoto y el recurso mismo, el desarrollador tiene la oportunidad de interceptar una falla del servicio y elegir un curso alternativo. de acción a tomar.

En Hystrix, esto se conoce como estrategia alternativa y se implementa fácilmente. Veamos cómo crear una estrategia alternativa simple para su base de datos de licencias que simplemente devuelva un objeto de licencia que indique que no hay información de licencia disponible actualmente. La siguiente lista lo demuestra.

#### Listado 5.5 Implementando un respaldo en Hystrix

El atributo fallbackMethod define una única función en su clase que se llamará si falla la llamada desde Hystrix.

```
@HystrixCommand(fallbackMethod = "buildFallbackLicenseList")
Lista pública<Licencia> getLicensesByOrg(String organizaciónId){ randomlyRunLong();}
```

```
    devolver LicenseRepository.findByOrganizationId(organizationId);
}
```

```
Lista privada<Licencia> buildFallbackLicenseList(String organizaciónId){
    Lista<Licencia> fallbackList = new ArrayList<>(); Licencia licencia =
        nueva Licencia() .withId("00000000-00-00000")
        .withOrganizationId( OrganizationId ) .withProductName( "Lo
        sentimos, no hay
        información de licencia disponible actualmente" );
```

En el método alternativo, devuelve un valor codificado.

```
fallbackList.add(licencia);
devolver lista alternativa;
}
```

**NOTA** En el código fuente del repositorio de GitHub, comento el línea fallbackMethod para que pueda ver que la llamada de servicio falla aleatoriamente. A vea el código alternativo en el listado 5.5 en acción, deberá descomentarlo el atributo fallbackMethod . De lo contrario, nunca verás el respaldo. siendo realmente invocado.

Para implementar una estrategia alternativa con Hystrix hay que hacer dos cosas. Primero tú Es necesario agregar un atributo llamado fallbackMethod a la anotación @HystrixCommand . Este atributo contendrá el nombre de un método que se llamará cuando Hystrix tiene que interrumpir una llamada porque está tardando demasiado.

Lo segundo que debe hacer es definir un método alternativo para ejecutar. Este El método alternativo debe residir en la misma clase que el método original protegido por @HystrixCommand. El método alternativo debe tener exactamente el mismo firma del método como función de origen, ya que todos los parámetros pasados al El método original protegido por @HystrixCommand se pasará al respaldo.

En el ejemplo del listado 5.5, el método alternativo buildFallbackLicense-List() consiste simplemente en construir un único objeto de licencia que contiene información ficticia. Podría hacer que su método alternativo lea estos datos desde datos alternativos fuente, pero para fines de demostración, vas a construir una lista que han sido devueltos por su llamada de función original.

#### Sobre las alternativas

La estrategia alternativa funciona muy bien en situaciones en las que su microservicio es recuperando datos y la llamada falla. En una organización en la que trabajé, teníamos clientes información almacenada en un almacén de datos operativos (ODS) y también resumida en un data store depósito.

Nuestro feliz camino fue recuperar siempre los datos más recientes y calcular el resumen. información sobre la marcha. Sin embargo, después de una interrupción particularmente desagradable en la que un lento La conexión de la base de datos eliminó múltiples servicios, decidimos proteger el servicio. llamada que recuperó y resumió la información del cliente con un respaldo de Hystrix implementación. Si la llamada al ODS falló debido a un problema de rendimiento o un error, Usamos un recurso alternativo para recuperar los datos resumidos de nuestras tablas de almacén de datos.

Nuestro equipo comercial decidió que era preferible proporcionar los datos más antiguos del cliente a hacer que el cliente vea un error o que toda la aplicación falle. La clave cuando elegir si utilizar una estrategia alternativa es el nivel de tolerancia de sus clientes tienen que ver con la antigüedad de sus datos y lo importante que es no dejarles ver que la aplicación tiene problemas.

Aquí hay algunas cosas que debe tener en cuenta al determinar si desea implementar una estrategia alternativa:

**1** Los respaldos son un mecanismo para proporcionar un curso de acción cuando un recurso se agotó el tiempo de espera o falló. Si se encuentra utilizando alternativas para aprovechar un tiempo de espera

excepción y luego no hacer nada más que registrar el error, entonces probablemente debería usar un bloque try.. catch estándar alrededor de su invocación de servicio, capturar HystrixRuntimeException y colocar la lógica de registro en el bloque try..catch.

- 2 Sea consciente de las acciones que está realizando con sus funciones de respaldo. Si llama a otro servicio distribuido en su servicio alternativo, es posible que deba ajustar el servicio alternativo con una anotación @HystrixCommand. Recuerde, el mismo error que está experimentando con su curso de acción principal también podría afectar su opción secundaria. Codifique a la defensiva. Me han mordido mucho cuando no tuve esto en cuenta al usar alternativas.

Ahora que tiene su alternativa implementada, continúe y llame a su punto final nuevamente. Esta vez, cuando lo presione y encuentre un error de tiempo de espera (recuerde que tiene una probabilidad entre 3), no debería recibir una excepción de la llamada de servicio, sino que se le devolverán los valores de licencia ficticios.

The screenshot shows a Postman request configuration and its resulting JSON response. The request is a GET to `http://localhost:8080/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/`. The Headers tab is selected, showing a key 'key' and value 'value'. The Body tab is selected, showing a Pretty tab and a Raw tab. The JSON response is displayed in the Raw tab:

```
1 - [ ]  
2 - {  
3 -   "licenseId": "000000-00-0000",  
4 -   "organizationId": "e254f8c-c442-4ebe-a82a-e2fc1d1ff78a",  
5 -   "organizationName": "",  
6 -   "contactName": "",  
7 -   "contactPhone": "",  
8 -   "contactEmail": "",  
9 -   "productName": "Sorry no licensing information currently available",  
10 -  "licenseType": null,  
11 -  "licenseMax": null,  
12 -  "licenseAllocated": null,  
13 -  "comment": null  
14 - }  
15 - ]
```

A curly brace icon is positioned next to the closing brace of the JSON object. A callout bubble points from this icon to the text "Resultados del código alternativo".

Figura 5.6 Su invocación de servicio usando un respaldo de Hystrix

## 5.7 Implementación del patrón de mamparo

En una aplicación basada en microservicios, a menudo necesitarás llamar a varios microservicios para completar una tarea particular. Sin utilizar un patrón de mamparo, el comportamiento predeterminado para estas llamadas es que las llamadas se ejecutan usando los mismos hilos que están reservados para manejo de solicitudes para todo el contenedor Java. En grandes volúmenes, los problemas de rendimiento con un servicio entre muchos pueden provocar que todos los subprocessos del contenedor Java estén al máximo y esperando para procesar el trabajo, mientras se acumulan nuevas solicitudes de trabajo. El contenedor de Java eventualmente fallará. El patrón de mamparo segregá los controles remotos llamadas de recursos en sus propios grupos de subprocessos para que un único servicio que se comporta mal pueda ser contenido y no estrellar el contenedor.

Hystrix utiliza un grupo de subprocessos para delegar todas las solicitudes de servicios remotos. Por defecto, todos los comandos de Hystrix compartirán el mismo grupo de subprocessos para procesar solicitudes. Este hilo tendrá 10 subprocessos para procesar llamadas de servicio remoto y esos servicios remotos. Las llamadas pueden ser cualquier cosa, incluidas invocaciones de servicios REST, llamadas a bases de datos, etc. La figura 5.7 ilustra esto.

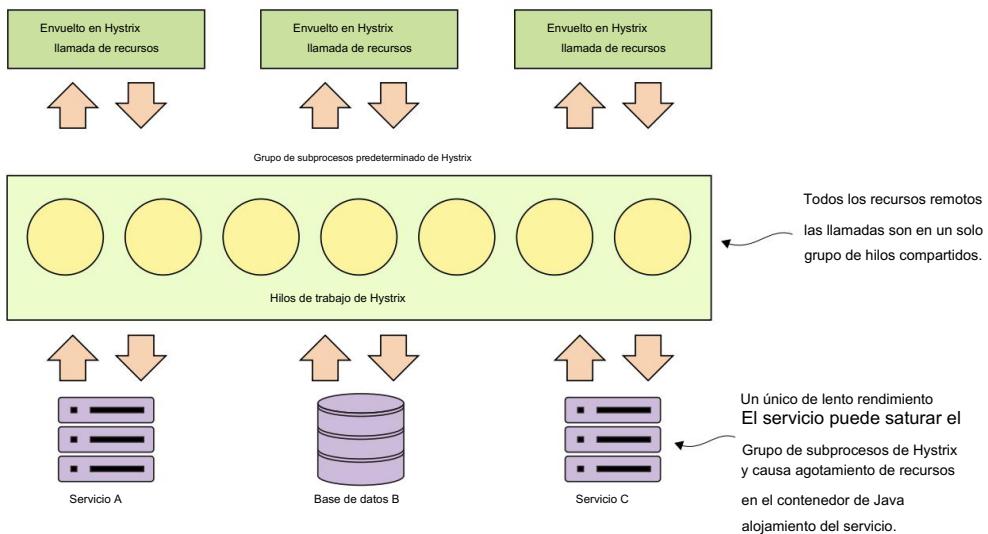


Figura 5.7 Grupo de subprocessos predeterminado de Hystrix compartido entre múltiples tipos de recursos

Este modelo funciona bien cuando tienes una pequeña cantidad de recursos remotos disponibles. Se accede a ellos dentro de una aplicación y los volúmenes de llamadas para los servicios individuales se distribuyen de manera relativamente uniforme. El problema es que si tienes servicios que tienen volúmenes mucho mayores o tiempos de finalización más largos que otros servicios, puedes terminar introduciendo agotamiento de subprocessos en sus grupos de subprocessos de Hystrix porque un servicio termina dominando todos los subprocessos en el grupo de subprocessos predeterminado.

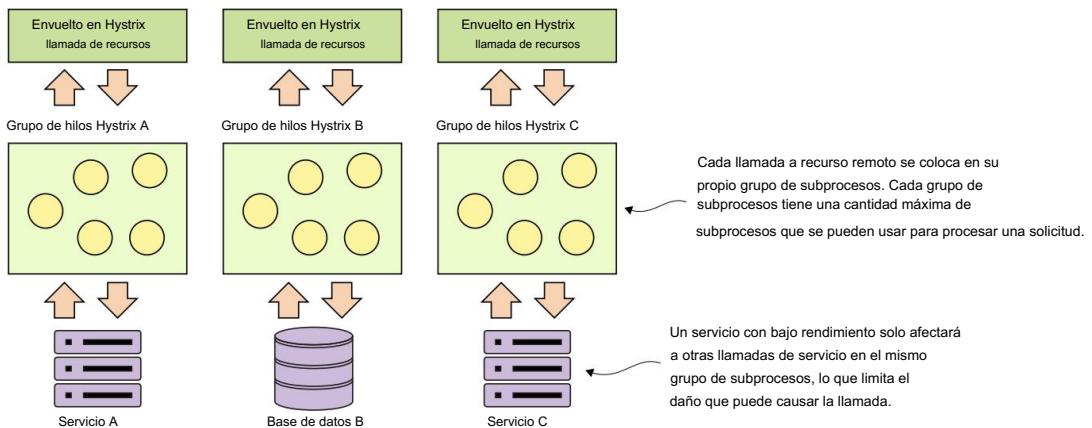


Figura 5.8 Comando Hystrix vinculado a grupos de subprocesos segregados

Afortunadamente, Hystrix proporciona un mecanismo fácil de usar para crear mamparos entre diferentes llamadas de recursos remotos. La Figura 5.8 muestra lo que Hystrix logró. Los recursos se ven cuando están segregados en sus propios "mamparos".

Para implementar grupos de subprocesos segregados, debe utilizar atributos adicionales expuesto a través de la anotación `@HystrixCommand`. Veamos un código que

- 1 Configure un grupo de subprocesos independiente para la llamada `getLicensesByOrg()`
- 2 Establezca el número de subprocesos en el grupo de subprocesos
- 3 Establezca el tamaño de la cola para el número de solicitudes que pueden ponerse en cola si el individuo los hilos están ocupados

La siguiente lista muestra cómo configurar un panel divisorio alrededor de todas las llamadas relacionadas con la búsqueda de datos de licencia desde nuestro servicio de licencias.

#### Listado 5.6 Creando un mamparo alrededor del método `getLicensesByOrg()`

El atributo `threadPoolProperties` le permite definir y personalizar el comportamiento de `threadPool`.

```
@HystrixCommand(fallbackMethod = "buildFallbackLicenseList",
    threadPoolKey = "licenseByOrgThreadPool",
    threadPoolProperties =
        {
            @HystrixProperty(nombre = "coreSize", valor="30"),
            @HystrixProperty(nombre="maxQueueSize", valor="10")
        }
)
```

```
Lista pública<Licencia> getLicensesByOrg(String organizaciónId){
    devolver LicenseRepository.findByOrganizationId(organizationId);
}
```

El atributo `coreSize` le permite definir el número máximo de subprocesos en el grupo de subprocesos.

El atributo `threadPoolKey` define el nombre único del grupo de subprocesos.

`maxQueueSize` te permite definir una cola que se encuentra frente a su grupo de subprocesos y que puede poner en cola las solicitudes entrantes.

Lo primero que debe notar es que hemos introducido un nuevo atributo, thread-Poolkey, en su anotación @HystrixCommand . Esto le indica a Hystrix que usted desea configurar un nuevo grupo de subprocessos. Si no establece más valores en el grupo de subprocessos, Hystrix configura un grupo de subprocessos con el nombre en el atributo threadPoolKey , pero utilizará todos los valores predeterminados para configurar el grupo de subprocessos.

Para personalizar su grupo de subprocessos, utilice el atributo threadPoolProperties en el @HystrixCommand . Este atributo toma una serie de objetos HystrixProperty .

Estos objetos HystrixProperty se pueden utilizar para controlar el comportamiento del hilo. piscina. Puede establecer el tamaño del grupo de subprocessos utilizando el atributo coreSize .

También puede configurar una cola frente al grupo de subprocessos que controlará cuántos Las solicitudes podrán realizar copias de seguridad cuando los subprocessos en el grupo de subprocessos estén ocupados. Este El tamaño de la cola lo establece el atributo maxQueueSize . Una vez que el número de solicitudes excede el tamaño de la cola, cualquier solicitud adicional al grupo de subprocessos fallará hasta que haya Hay espacio en la cola.

Tenga en cuenta dos cosas sobre el atributo maxQueueSize . Primero, si establece el valor en -1, Se utilizará una Java SynchronousQueue para contener todas las solicitudes entrantes. Un sincrónico La cola esencialmente hará cumplir que nunca podrá tener más solicitudes en proceso que entonces. el número de subprocessos disponibles en el grupo de subprocessos. Establecer maxQueueSize en un Un valor mayor que uno hará que Hystrix utilice Java LinkedBlockingQueue. El El uso de LinkedBlockingQueue permite al desarrollador poner en cola solicitudes incluso si Todos los hilos están ocupados procesando solicitudes.

La segunda cosa a tener en cuenta es que el atributo maxQueueSize solo se puede establecer cuando el grupo de subprocessos se inicializa primero (por ejemplo, al iniciar la aplicación). Hystrix le permite cambiar dinámicamente el tamaño de la cola utilizando el atributo queue-SizeRejectionThreshold , pero este atributo solo se puede establecer cuando el El atributo maxQueueSize es un valor mayor que 0.

¿Cuál es el tamaño adecuado para un grupo de subprocessos personalizado? Netflix recomienda la siguiente fórmula:

$$\text{(solicitudes por segundo en el pico cuando el servicio está en buen estado)} * \text{Latencia percentil 99 en} \\ + \text{pequeña cantidad de subprocessos adicionales para gastos generales}$$

A menudo no se conocen las características de rendimiento de un servicio hasta que se ha bajo carga. Un indicador clave de que es necesario ajustar las propiedades del grupo de subprocessos es cuando una llamada de servicio está agotando el tiempo de espera incluso si el recurso remoto de destino está en buen estado.

## 5.8 Ir más allá de lo básico; ajuste fino de Hystrix

En este punto hemos analizado los conceptos básicos de configuración de un disyuntor y patrón de mamparo usando Hystrix. Ahora vamos a repasar y ver cómo realmente Personalice el comportamiento del disyuntor de Hystrix. Recuerde, Hystrix hace más en lugar de cronometrar las llamadas de larga duración. Hystrix también monitoreará la cantidad de veces que se realiza una llamada. falla y si fallan suficientes llamadas, Hystrix evitará automáticamente que futuras llamadas lleguen al servicio fallando la llamada antes de que las solicitudes lleguen al recurso remoto.

Hay dos razones para esto. Primero, si un recurso remoto tiene rendimiento problemas, fallar rápidamente evitará que la aplicación que llama tenga que esperar una llamada para terminar el tiempo. Esto reduce significativamente el riesgo de que la aplicación o servicio que llama experimentará sus propios problemas de agotamiento de recursos y fallas. Segundo, fallar rápido y evitar llamadas de clientes de servicio ayudará a que un servicio en dificultades se mantenga al día con su cargar y no chocar completamente bajo la carga. Una falla rápida le da al sistema que experimenta una degradación del rendimiento tiempo para recuperarse.

Para comprender cómo configurar el disyuntor en Hystrix, primero debe comprender el flujo de cómo Hystrix determina cuándo disparar el disyuntor. La figura 5.9 muestra el proceso de decisión utilizado por Hystrix cuando falla una llamada a un recurso remoto.

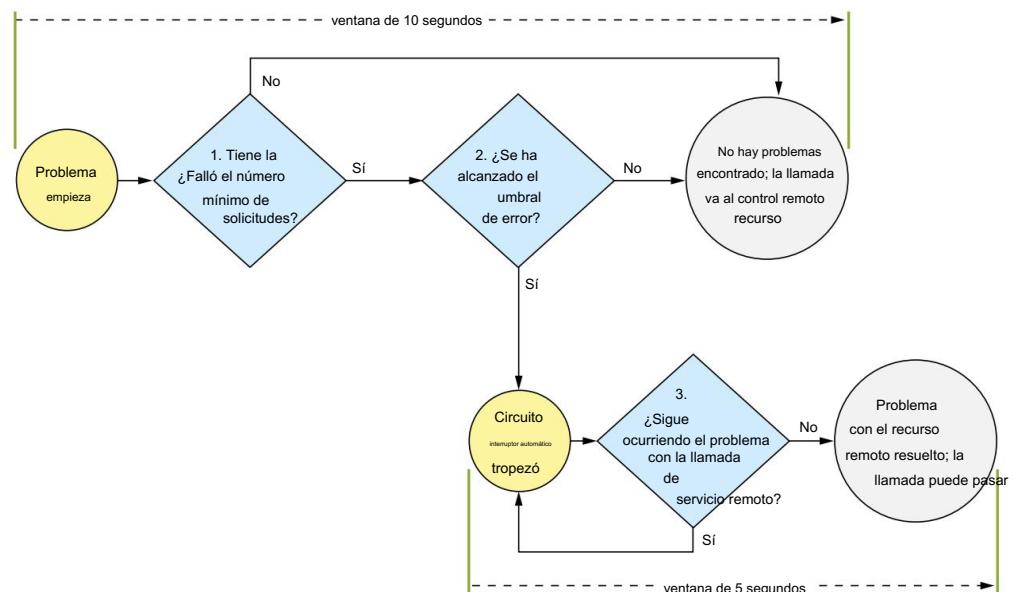


Figura 5.9 Hystrix pasa por una serie de comprobaciones para determinar si debe disparar o no el disyuntor.

Cada vez que un comando de Hystrix encuentra un error con un servicio, comenzará un ciclo de 10-segundo temporizador que se utilizará para examinar con qué frecuencia falla la llamada de servicio. Este 10-segundo ventana es configurable. Lo primero que hace Hystrix es mirar el número de llamadas que han ocurrido dentro de la ventana de 10 segundos. Si el número de llamadas es menor que un número mínimo de llamadas que deben ocurrir dentro de la ventana, entonces Hystrix no tomará medidas incluso si varias de las llamadas fallan. Por ejemplo, el número predeterminado de llamadas que deben ocurrir antes de que Hystrix considere siquiera tomar medidas dentro del 10-segundo ventana es 20. Si 15 de esas llamadas fallan dentro de un período de 10 segundos, no es suficiente.

de las llamadas se han producido para que "disparen" el disyuntor incluso si las 15 llamadas fallido. Hystrix seguirá permitiendo que las llamadas pasen al servicio remoto.

Cuando se haya producido el número mínimo de llamadas a recursos remotos dentro de los 10

En la segunda ventana, Hystrix comenzará a observar el porcentaje de fallas generales que ha ocurrido. Si el porcentaje general de fallas supera el umbral, Hystrix activar el disyuntor y fallar en casi todas las llamadas futuras. Como veremos en breve, Hystrix dejará pasar parte de las llamadas para "probar" y ver si el servicio está respaldado. El valor predeterminado para el umbral de error es 50%.

Si se ha excedido ese porcentaje, Hystrix "disparará" el disyuntor y evitar que más llamadas lleguen al recurso remoto. Si ese porcentaje de remotos llamadas no se han activado y se ha pasado la ventana de 10 segundos, Hystrix restablecer las estadísticas del disyuntor.

Cuando Hystrix haya "disparado" el disyuntor en una llamada remota, intentará iniciar una nueva ventana de actividad. Cada cinco segundos (este valor es configurable), Hystrix permitirá que llame al servicio que tiene problemas. Si la llamada tiene éxito, Hystrix restablecerá el circuito. interruptor y comenzar a dejar pasar llamadas nuevamente. Si la llamada falla, Hystrix mantendrá el disyuntor cerrado y volverá a intentarlo en otros cinco segundos.

En base a esto, puedes ver que hay cinco atributos que puedes usar para personalizar el comportamiento del disyuntor. La anotación @HystrixCommand expone estos cinco atributos a través del atributo commandPoolProperties . Si bien el atributo threadPoolProperties le permite establecer el comportamiento del grupo de subprocesos subyacente utilizado en el comando Hystrix, el atributo commandPoolProperties le permite personalizar el comportamiento del disyuntor asociado con el comando Hystrix. La siguiente lista muestra los nombres de los atributos junto con cómo establecer valores en cada uno de ellos.

#### Listado 5.7 Configuración del comportamiento de un disyuntor

```
@HystrixCommand(  
    fallbackMethod = "buildFallbackLicenseList",  
    threadPoolKey = "licenseByOrgThreadPool",  
    threadPoolProperties ={  
        @HystrixProperty(nombre = "coreSize",valor="30"),  
        @HystrixProperty(nombre="maxQueueSize"valor="10"),  
    },  
    comandoPoolProperties ={  
        @PropiedadHystrix(  
            nombre="circuitBreaker.requestVolumeThreshold",      valor="10"),  
        @HystrixProperty(  
            nombre="circuitBreaker.errorThresholdPercentage",      valor="75"),  
  
        @PropiedadHystrix(  
            nombre="circuitBreaker.sleepWindowInMilliseconds",  
            valor="7000"),  
        @HystrixProperty(
```

```

nombre="metrics.rollingStats.timeInMilliseconds",    valor="15000"

@PropiedadHystrix(
    nombre="metrics.rollingStats.numBuckets",    valor="5"))

)

Lista pública<Licencia> getLicensesByOrg(String organizaciónId){
    logger.debug("getLicensesByOrg ID de correlación: {}",    UserContextHolder

        .getContext()
        .getCorrelationId());
    aleatoriamenteRunLong();

    devolver LicenseRepository.findByOrganizationId(organizationId);
}

```

La primera propiedad, circuitoBreaker.requestVolumeThreshold, controla el cantidad de llamadas consecutivas que deben ocurrir dentro de una ventana de 10 segundos antes de que Hystrix considerará disparar el disyuntor para la llamada. La segunda propiedad, circuito-Breaker.errorThresholdPercentage, es el porcentaje de llamadas que deben fallar.

(debido a tiempos de espera, una excepción lanzada o un HTTP 500 devuelto) después de la El valor circuitoBreaker.requestVolumeThreshold se pasó antes de que se disparara el disyuntor. La última propiedad en el ejemplo de código anterior, circuito-Breaker.sleepWindowInMilliseconds, es la cantidad de tiempo que Hystrix dormirá.

una vez que se dispara el disyuntor antes de que Hystrix permita que pase otra llamada ver si el servicio vuelve a estar sano.

Las dos últimas propiedades de Hystrix (metrics.rollingStats.timeInMillisec-onds y metrics.rollingStats.numBuckets) tienen nombres un poco diferentes a las propiedades anteriores, pero aún controlan el comportamiento del disyuntor. El La primera propiedad, metrics.rollingStats.timeInMilliseconds, se utiliza para controlar el tamaño de la ventana que utilizará Hystrix para monitorear problemas con una llamada de servicio. El valor predeterminado para esto es 10.000 milisegundos (es decir, 10 segundos).

La segunda propiedad, metrics.rollingStats.numBuckets, controla el número de veces que se recopilan estadísticas en la ventana que ha definido. Hystrix recopila métricas en depósitos durante esta ventana y verifica las estadísticas en esos depósitos para determinar si la llamada al recurso remoto falla. El número de depósitos definidos debe ser uniforme divida entre el número total de milisegundos establecidos para las estadísticas de RollingStatus.inMilli-segundos . Por ejemplo, en su configuración personalizada en el listado anterior, Hystrix utilizará una ventana de 15 segundos y recopilará datos estadísticos en cinco grupos de tres segundos. segundos de longitud.

**NOTA** Cuanto más pequeña sea la ventana de estadísticas que registre y mayor será el La cantidad de depósitos que mantenga dentro de la ventana aumentará la utilización de la CPU y la memoria en un servicio de gran volumen. Sea consciente de esto y luche contra la tentación de configurar las ventanas y los depósitos de recopilación de métricas para que sean detallados hasta que necesitas ese nivel de visibilidad.

### 5.8.1 Configuración de Hystrix revisada

La biblioteca Hystrix es extremadamente configurable y le permite controlar estrictamente el comportamiento del disyuntor y los patrones de mamparo que defina con él. Al modificar la configuración de un disyuntor Hystrix, puede controlar la cantidad de tiempo que Hystrix permanecerá esperando antes de finalizar el tiempo de una llamada remota. También puede controlar el comportamiento de cuándo se disparará un disyuntor Hystrix y cuándo Hystrix intenta restablecer el disyuntor.

Con Hystrix también puede ajustar sus implementaciones de mamparo definiendo grupos de subprocessos individuales para cada llamada de servicio remoto y luego configurar el número de hilos asociados con cada grupo de hilos. Esto le permite ajustar su control remoto llamadas de servicio porque ciertas llamadas tendrán mayores volúmenes que otras, mientras que otras llamadas a recursos remotos tendrán mayores volúmenes.

Lo clave que debe recordar al configurar su entorno Hystrix es que tienes tres niveles de configuración con Hystrix:

- 1 Predeterminado para toda la aplicación
- 2 Predeterminado para la clase
- 3 Nivel de grupo de subprocessos definido dentro de la clase

Cada propiedad de Hystrix tiene valores establecidos de forma predeterminada que serán utilizados por cada anotación `@Hystrix-Command` en la aplicación a menos que estén configurados en el nivel de clase Java o se anulen para grupos de subprocessos individuales de Hystrix dentro de una clase.

Hystrix le permite establecer parámetros predeterminados a nivel de clase para que todos los comandos de Hystrix dentro de una clase específica comparten las mismas configuraciones. Las propiedades a nivel de clase se configuran mediante una anotación a nivel de clase llamada `@DefaultProperties`. Por ejemplo, si usted quería que todos los recursos dentro de una clase específica tuvieran un tiempo de espera de 10 segundos, podría configurar `@DefaultProperties` de la siguiente manera:

```
@PropiedadesDefectuosas(  
    propiedades de comando = {  
        @PropiedadHystrix(  
            nombre = "execution.isolation.thread.timeoutInMilisegundos", valor = "10000")}  
  
clase MiServicio {...}
```

A menos que se anule explícitamente a nivel de grupo de subprocessos, todos los grupos de subprocessos heredarán las propiedades predeterminadas en el nivel de aplicación o las propiedades predeterminadas definidas en el clase. Hystrix threadPoolProperties y commandProperties también están vinculados a la tecla de comando definida.

**NOTA** Para los ejemplos de codificación, he codificado todos los valores de Hystrix en el código de aplicación. En un sistema de producción, los datos de Hystrix que tienen más probabilidades de que necesitan ser modificados (parámetros de tiempo de espera, recuentos de grupos de subprocessos) se externalizarían a Spring Cloud Config. De esta manera, si necesita cambiar los valores de los parámetros, puede cambiar los valores y luego reiniciar las instancias de servicio. sin tener que volver a compilar y volver a implementar la aplicación.

Para grupos individuales de Hystrix, mantendré la configuración lo más cerca posible del código y colocaré la configuración del grupo de subprocessos directamente en la anotación `@HystrixCommand`. La Tabla 5.1 resume todos los valores de configuración utilizados para instalar y configurar nuestras anotaciones `@HystrixCommand`.

Tabla 5.1 Valores de configuración para anotaciones `@HystrixCommand`

Nombre de la propiedad	Por defecto Valor	Descripción
método alternativo	Ninguno	Identifica el método dentro de la clase que será llamado si la llamada remota se agota. El método de devolución de llamada debe estar en la misma clase que la anotación <code>@HystrixCommand</code> y debe tener la misma firma de método que la clase que llama. Si no hay ningún valor, Hystrix lanzará una excepción.
hiloPoolKey	Ninguno	Le da a <code>@HystrixCommand</code> un nombre único y crea un grupo de subprocessos que es independiente del grupo de subprocessos predeterminado. Si no se define ningún valor, se utilizará el grupo de subprocessos predeterminado de Hystrix.
threadPoolProperties	Ninguno	Atributo de anotación Core Hystrix que se utiliza para configurar el comportamiento de un grupo de subprocessos.
tamaño del núcleo	10	Establece el tamaño del grupo de subprocessos.
maxQueueSize	-1	Tamaño máximo de cola que se establecerá frente al grupo de subprocessos. Si se establece en -1, no se utiliza ninguna cola y, en su lugar, Hystrix se bloqueará hasta que un hilo esté disponible para su procesamiento.
circuitoBreaker.requestVolumeThreshold	20	Establece el número mínimo de solicitudes que deben procesarse dentro de la ventana móvil antes de que Hystrix comience a examinar si se activará el disyuntor.  Nota: Este valor solo se puede establecer con el atributo <code>commandPoolProperties</code> .
circuitoBreaker.error- UmbralPorcentaje	50	El porcentaje de fallas que deben ocurrir dentro de la ventana móvil antes de que se dispare el disyuntor.  Nota: Este valor solo se puede establecer con el atributo <code>commandPoolProperties</code> .
circuitoBreaker.sleep- WindowInMilisegundos	5000	La cantidad de milisegundos que Hystrix esperará antes de intentar una llamada de servicio después de que se haya disparado el disyuntor.  Nota: Este valor solo se puede establecer con el atributo <code>commandPoolProperties</code> .
métricasRollingStats. tiempo en milisegundos	10,000	La cantidad de milisegundos que Hystrix recopilará y monitoreará estadísticas sobre llamadas de servicio dentro de una ventana.
métricasRollingStats. .numCubos	10	La cantidad de depósitos de métricas que Hystrix mantendrá dentro de la ventana de monitoreo. Cuantos más depósitos haya dentro de la ventana de monitoreo, menor será el nivel de tiempo que Hystrix monitoreará para detectar fallas dentro de la ventana.

## 5.9 Contexto del hilo y Hystrix

Cuando se ejecuta un `@HystrixCommand`, se puede ejecutar con dos estrategias de aislamiento diferentes: THREAD y SEMAPHORE. De forma predeterminada, Hystrix se ejecuta con un aislamiento THREAD. Cada comando de Hystrix utilizado para proteger una llamada se ejecuta en un grupo de subprocessos aislado que no comparte su contexto con el subprocesso principal que realiza la llamada. Esto significa que Hystrix puede interrumpir la ejecución de un subprocesso bajo su control sin preocuparse por interrumpir cualquier otra actividad asociada con el subprocesso principal que realiza la invocación original.

Con el aislamiento basado en SEMAPHORE, Hystrix administra la llamada distribuida protegida por la anotación `@HystrixCommand` sin iniciar un nuevo hilo e interrumpirá el hilo principal si la llamada se agota. En un entorno de servidor de contenedor síncrono (Tomcat), la interrupción del subprocesso principal provocará que se genere una excepción que el desarrollador no puede detectar. Esto puede tener consecuencias inesperadas para el desarrollador que escribe el código porque no puede detectar la excepción generada ni realizar ninguna limpieza de recursos o manejo de errores.

Para controlar la configuración de aislamiento para un grupo de comandos, puede establecer un atributo de propiedades de comando en su anotación `@HystrixCommand`. Por ejemplo, si quisiera establecer el nivel de aislamiento en un comando Hystrix para usar un aislamiento SEMAPHORE, usaría

```
@HystrixCommand( propiedades del comando = {  
    @PropiedadHystrix(  
        nombre="ejecución.isolación.estategia", valor="SEMÁFORO"))}
```

**NOTA** De forma predeterminada, el equipo de Hystrix recomienda utilizar la estrategia de aislamiento predeterminada de THREAD para la mayoría de los comandos. Esto mantiene un mayor nivel de aislamiento entre usted y el hilo principal. El aislamiento THREAD es más pesado que usar el aislamiento SEMAPHORE. El modelo de aislamiento SEMAPHORE es más liviano y debe usarse cuando tiene un gran volumen de servicios y se ejecuta en un modelo de programación de E/S asíncrono (está utilizando un contenedor de E/S asíncrono como Netty).

### 5.9.1 ThreadLocal y Hystrix

Hystrix, de forma predeterminada, no propagará el contexto del subprocesso principal a los subprocessos administrados por un comando de Hystrix. Por ejemplo, cualquier valor establecido como valor `ThreadLocal` en el hilo principal no estará disponible de forma predeterminada para un método llamado por el hilo principal y protegido por el objeto `@HystrixCommand`. (Nuevamente, esto supone que está utilizando un nivel de aislamiento THREAD).

Esto puede resultar un poco obtuso, así que veamos un ejemplo concreto. A menudo, en un entorno basado en REST, querrá pasar información contextual a una llamada de servicio que le ayudará a gestionar operativamente el servicio. Por ejemplo, puede pasar un ID de correlación o un token de autenticación en el encabezado HTTP de la llamada REST que puede

luego se propagará a cualquier llamada de servicio descendente. El ID de correlación le permite

tener un identificador único que se puede rastrear a través de múltiples llamadas de servicio en una sola transacción.

Para que este valor esté disponible en cualquier lugar de su llamada de servicio, puede usar una clase Spring Filter para interceptar cada llamada a su servicio REST y recuperar esta información de la solicitud HTTP entrante y almacenar esta información contextual en un objeto de contexto de usuario personalizado. Luego, en cualquier momento que su código necesite acceder a este valor en su servicio REST llame, su código puede recuperar el UserContext de la variable de almacenamiento ThreadLocal y leer el valor. La siguiente lista muestra un ejemplo de filtro de resorte que puede utilizar en su servicio de licencias. Puede encontrar el código en licensingservice/src/main/java/com/thinkmechanix/licenses/utils/UserContextFilter.java.

#### Listado 5.8 UserContextFilter analiza el encabezado HTTP y recupera datos

```
paquete com.thinkmechanix.licenses.utils;

//Algo de código eliminado por razones de concisión
@Component
la clase pública UserContextFilter implementa el filtro {
    final estático privado Logger
    logger =
    LoggerFactory.getLogger(UserContextFilter.class);
    @Anular
    filtro público vacío (
    ServletRequest servletRequest,   ServletResponse
    servletResponse,   FilterChain filterChain)

    lanza IOException, ServletException {
        HttpServletRequest httpServletRequest =
        (HttpServletRequest) servletRequest;
        Titular del contexto de usuario
        .getContext()
        .setCorrelationId(
            httpServletRequest.getHeader(UserContext.CORRELATION_ID));
        Titular del contexto de usuario
        .getContext()
        .setUserId( httpServletRequest.getHeader(UserContext.USER_ID));
        Titular del contexto de usuario
        .getContext()
        .setAuthToken(
            httpServletRequest.getHeader(UserContext.AUTH_TOKEN));
        Titular del contexto de usuario
        .getContext()
        .setOrgId(
            httpServletRequest.getHeader(UserContext.ORG_ID));

        filterChain.doFilter(httpServletRequest, servletResponse);
    }
}
```

El diagrama es un diagrama de flujo que muestra la recuperación de valores establecidos en el Encabezado HTTP de la llamada en un UserContext, que se almacena en UserContextHolder. Se muestra una secuencia de flechas que parten de los métodos de recuperación de los campos (getUserId(), setCorrelationId(), setAuthToken(), setOrgId()) y apuntan hacia un cuadro que contiene el texto: "Recuperar valores establecidos en el Encabezado HTTP de la llamada en un UserContext, que se almacena en UserContextHolder".

La clase UserContextHolder se utiliza para almacenar UserContext en un ThreadLocal clase. Una vez almacenado en el almacenamiento ThreadLocal , cualquier código que se ejecute durante un La solicitud utilizará el objeto UserContext almacenado en UserContextHolder. El La clase UserContextHolder se muestra en el siguiente listado. Esta clase se encuentra en servicio-de-licencias/src/main/java/com/thinktmechanix/licenses/utils /UserContextHolder.java.

#### Listado 5.9 Todos los datos de UserContext son administrados por UserContextHolder

```

clase pública UserContextHolder {
    ThreadLocal final estático privado<UserContext> userContext
        = nuevo ThreadLocal<UserContext>();

    público estático final UserContext getContext(){ Contexto de usuario
        contexto = userContext.get();

        si (contexto == nulo) {
            contexto = crearContextoEmpty();
            usuarioContext.set(contexto);
        }
        devolver userContext.get();
    }

    público estático final vacío setContext (contexto UserContext) {
        Afirmar.notNull(contexto,
            "Sólo se permiten instancias de UserContext no nulas");
        usuarioContext.set(contexto);
    }

    UserContext final estático público createEmptyContext(){
        devolver nuevo UserContext();
    }
}

```

El contexto de usuario es almacenado en una estática Variable ThreadLocal.

El método getContext() recuperará el objeto UserContext para su consumo.

En este punto, puede agregar un par de declaraciones de registro a su servicio de licencia. usted agregue el registro a las siguientes clases y métodos de servicios de licencias:

com/thinktmechanix/licenses/utils/UserContextFilter.java

Método doFilter() com/

thinktmechanix/licenses/controllers/LicenseService-Controller.java método getLicenses()

com/thinktmechanix/licenses/services/LicenseService.java

método getLicensesByOrg() . Este método está anotado con @Hystrix-Dominio.

A continuación, llamará a su servicio pasando un ID de correlación mediante un encabezado HTTP llamado tmx-correlation-id y un valor de TEST-CORRELATION-ID. La figura 5.10 muestra una Llamada HTTP GET a http://localhost:8080/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/ en Postman.

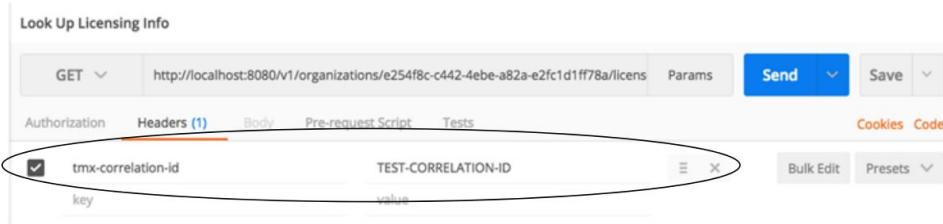


Figura 5.10 Agregar un ID de correlación al encabezado HTTP de la llamada al servicio de licencias

Una vez enviada esta llamada, debería ver tres mensajes de registro que escriben el ID de correlación pasado a medida que fluye a través de las clases UserContext, LicenseServiceController y LicenseServer :

ID de correlación de UserContext: TEST-CORRELATION-ID  
 LicenseServiceController ID de correlación: TEST-CORRELATION-ID  
 Correlación LicenseService.getLicenseByOrg:

Como era de esperar, una vez que la llamada llegue al método protegido de Hystrix en LicenseService.getLicensesByOrder(), no obtendrá ningún valor escrito para el ID de correlación. Afortunadamente, Hystrix y Spring Cloud ofrecen un mecanismo para propagar el contexto del subproceso principal a los subprocesos administrados por el grupo de subprocesos Hystrix. Este mecanismo se llama HystrixConcurrencyStrategy.

### 5.9.2 La estrategia de concurrencia de Hystrix en acción

Hystrix le permite definir una estrategia de concurrencia personalizada que envolverá su Hystrix llama y le permite inyectar cualquier contexto de hilo principal adicional en los hilos administrado por el comando Hystrix. Para implementar una estrategia de concurrencia de Hystrix personalizada , debe realizar tres acciones: 1. Definir

su clase de estrategia de concurrencia de Hystrix personalizada.

2 Defina una clase Java invocable para inyectar UserContext en Hystrix Dominio

3 Configure Spring Cloud para usar su estrategia de concurrencia Hystrix personalizada

Todos los ejemplos de HystrixConcurrencyStrategy se pueden encontrar en el paquete licens-ing-service/src/main/java/com/thinktmechanix/licenses/hystrix.

#### DEFINIR SU CLASE DE ESTRATEGIA DE CONCURRENCIA PERSONALIZADA DE HYSTRIX

Lo primero que debe hacer es definir su HystrixConcurrencyStrategy. Por De forma predeterminada, Hystrix solo le permite definir una HystrixConcurrencyStrategy para Una aplicación. Spring Cloud ya define una estrategia de concurrencia utilizada para manejar propagar información de seguridad de Spring. Afortunadamente, Spring Cloud te permite Encadene estrategias de concurrencia de Hystrix para que pueda definir y utilizar las suyas propias. estrategia de concurrencia “conectándola” a la estrategia de concurrencia de Hystrix.

Nuestra implementación de una estrategia de concurrencia de Hystrix se puede encontrar en el paquete hystrix de servicios de licencia y se llama ThreadLocalAwareStrategy.java. El siguiente listado muestra el código de esta clase.

```
paquete com.thinkmechanix.licenses.hystrix;  
extender la base  
Clase HystrixConcurrencyStrategy.  
//importaciones eliminadas por razones de concisión  
la clase pública ThreadLocalAwareStrategy extiende HystrixConcurrencyStrategy{  
    HystrixConcurrencyStrategy privado existenteConcurrencyStrategy;  
  
    ThreadLocalAwareStrategy público (←  
        HystrixConcurrencyStrategy existenteConcurrencyStrategy) {←  
            this.existingConcurrencyStrategy = existenteConcurrencyStrategy;  
    }  
    Spring Cloud ya tiene una clase de concurrencia definida.  
    Pasar la estrategia de concurrencia existente a la clase.  
    constructor de su HystrixConcurrencyStrategy.  
    @Anular  
    público BlockingQueue<Ejecutable> getBlockingQueue(int maxQueueSize){  
        devolver existenteConcurrencyStrategy! = nulo  
            ? existenteConcurrencyStrategy.getBlockingQueue(maxQueueSize)  
            : super.getBlockingQueue(maxQueueSize);  
    }  
  
    @Override  
    public <T> HystrixRequestVariable<T> getRequestVariable(  
        HystrixRequestVariableLifecycle<T> rv)  
    {  
        //Código eliminado por razones de concisión  
  
        //Código eliminado por razones de concisión  
        @Anular  
        público ThreadPoolExecutor getThreadPool (←  
            HystrixThreadPoolKey threadPoolKey,  
            HystrixProperty<Integer> corePoolSize,  
            HystrixProperty<Integer> máximoPoolSize,  
            HystrixProperty<Integer> keepAliveTime,  
            unidad de tiempo,  
            BlockingQueue<Runnable> workQueue) {  
                //código  
                eliminado por razones de concisión}  
  
        @Anular  
        public <T> Invocable<T> wrapCallable(Invocable<T> invocable) {  
            devolver existenteConcurrencyStrategy! = nulo  
                ? existenteConcurrencyStrategy.wrapCallable(  
                    nuevo DelegatingUserContextCallable<T>( invocable,  
                        UserContextHolder.getContext()));  
                : super.wrapCallable(  
                    nuevo DelegatingUserContextCallable<T>(←  
                        invocable,  
                        UserContextHolder.getContext())));  
        }  
    }  
    Es necesario utilizar varios métodos  
    anulado. O llame al  
    método existenteConcurrencyStrategy  
    implementación o llamar a la base  
    Estrategia de concurrencia de Hystrix.  
    Inyecte su implementación  
    invocable que establecerá el  
    UserContext.
```

Tenga en cuenta un par de cosas en la implementación de la clase en el listado 5.10. Primero, porque Spring Cloud ya define una `HystrixConcurrencyStrategy`, cada método que podría ser anulado necesita verificar si una estrategia de concurrencia existente está presente y luego llamar al método de la estrategia de concurrencia existente o al método de estrategia de concurrencia base Hystrix. Tienes que hacer esto como una convención para asegurar que invoca correctamente la estrategia `HystrixConcurrencyStrategy` de Spring Cloud ya existente que se ocupa de la seguridad. De lo contrario, puede tener un comportamiento desagradable al intentar utilizar el contexto de seguridad de Spring en su código protegido de Hystrix.

La segunda cosa a tener en cuenta es el método `wrapCallable()` en el listado 5.11. En este método, pasa la implementación invocable, `DelegatingUserContext Callable`, que se usará para configurar el `UserContext` desde el hilo principal que ejecuta la llamada de servicio REST del usuario al hilo de comando de Hystrix que protege el método.

eso es hacer el trabajo interior.

#### DEFINIR UNA CLASE JAVA LLAMABLE PARA INYECTAR EL CONTEXTO DE USUARIO EN EL COMANDO HYSTRIX

El siguiente paso para propagar el contexto del hilo principal a su Hystrix

El comando es implementar la clase `Callable` que hará la propagación. Para esto

Por ejemplo, esta llamada está en el paquete `hystrix` y se llama `DelegatingUserContextCallable.java`.

El siguiente listado muestra el código de esta clase.

Listado 5.11 Propagar el `UserContext` con `DelegatingUserContextCallable.java`

```
paquete com.thinkmechanix.licenses.hystrix;

//importar eliminar concisión
clase final pública DelegatingUserContextCallable<V>
    implementa invocable<V> {
    delegado final privado invocable<V>;
    contexto de usuario privado contexto de usuario original;

    público DelegatingUserContextCallable(
        Delegado <V> invocable,
        Contexto de usuarioContexto de usuario) {
        this.delegate = delegado;
        this.originalUserContext = usuarioContext;
    }

    llamada pública V () lanza una excepción
        {UserContextHolder.setContext (originalUserContext);

        intentar {
            devolver delegado.call();
        }
        finalmente {
            this.originalUserContext = nulo;
        }
    }

    public static <V> Callable<V> create(Callable<V> delegado,
        Contexto de usuarioContexto de usuario) {
        devolver nuevo DelegatingUserContextCallable<V>(delegado, userContext);
    }
}
```

La función llamada() se invoca antes del método protegido por la anotación @HystrixCommand. →

La clase invocable personalizada será pasó el invocable original clase que invocará tu Código protegido Hystrix y `UserContext` procedente de el hilo principal ←

El contexto de usuario está configurado. El Variable `ThreadLocal` que almacena el `UserContext` está asociado con el hilo corriendo Método protegido de Hystrix. ←

Una vez configurado `UserContext`, invoque el método `call()` en el método protegido de Hystrix; por ejemplo, tu Método `LicenseServer.getLicenseByOrg()`. ←

Cuando se realiza una llamada a un método protegido de Hystrix, Hystrix y Spring Cloud crear una instancia de la clase `DelegatingUserContextCallable`, pasando la clase invocable que normalmente sería invocada por un hilo administrado por Hystrix grupo de comandos. En el listado anterior, esta clase invocable se almacena en una propiedad de Java. llamado delegado. Conceptualmente, puedes pensar en la propiedad delegada como la maneja el método protegido por una anotación `@HystrixCommand`.

Además de la clase `Callable` delegada, Spring Cloud también está transmitiendo el objeto `UserContext` fuera del hilo principal que inició la llamada. con estos dos valores establecidos en el momento en que se crea la instancia `DelegatingUserContextCallable`, la acción real ocurrirá en el método `call()` de tu clase.

Lo primero que debe hacer en el método `call()` es configurar `UserContext` mediante el método `UserContextHolder.setContext()`. Recuerde, el método `setContext()` almacena un objeto `UserContext` en una variable `ThreadLocal` específica del hilo que se está correr. Una vez que se establece `UserContext`, se invoca el método `call()` de la clase `Callable` delegada. Esta llamada a delegado.`call()` invoca el método protegido por la anotación `@HystrixCommand`.

#### CONFIGURA SPRING CLOUD PARA UTILIZAR TU ESTRATEGIA DE CONCURRENCIA PERSONALIZADA DE HYSTRIX

Ahora que tiene su `HystrixConcurrencyStrategy` a través de la clase `ThreadLocal-AwareStrategy` y su clase `Callable` definida a través de la clase `DelegatingUser-ContextCallable`, debe conectarlas en Spring Cloud y Hystrix. Hacer esto, vas a definir una nueva clase de configuración. Esta configuración, llamada `ThreadLocalConfiguration`, se muestra en el siguiente listado.

#### Listado 5.12 Conexión de la clase `HystrixConcurrencyStrategy` personalizada en Spring Cloud

```
paquete com.thinkmechanix.licenses.hystrix;
//Importaciones eliminadas por razones de concisión
@Configuration
clase pública ThreadLocalConfiguration {
    @Autowired(obligatorio = falso)
    HystrixConcurrencyStrategy privado existenteConcurrencyStrategy;
    @PostConstruct
    inicio vacío público() {
        // Mantiene referencias de complementos existentes de Hystrix.
        HystrixEventNotifier eventNotifier =  HystrixPlugins
            .obtener Instancia()
            .getEventNotificador();
        Métricas de HystrixMetricsPublisherPublisher =  HystrixPlugins
            .obtener Instancia()
            .getMetricsPublisher();
        HystrixPropertiesStrategy propertiesStrategy =  HystrixPlugins
            .obtener Instancia()
            .getPropertiesStrategy();
    }
}
```

El diagrama es un diagrama de flujo que muestra la relación entre la clase `ThreadLocalConfiguration` y las estrategias de concurrencia de Hystrix. Se divide en tres secciones principales:

- Configuración de la clase `HystrixConcurrencyStrategy`:** Se muestra la declaración de la clase `ThreadLocalConfiguration` y su anotación `@Configuration`. La descripción indica: "Cuando la configuración El objeto está construido. autowire en el existente Estrategia de concurrencia de Hystrix." Un cuadro de diálogo aparece sobre la parte derecha de la clase.
- Instanciación de las estrategias:** Se muestra el constructor vacío de la clase `ThreadLocalConfiguration` que crea instancias de `HystrixEventNotifier`, `HystrixMetricsPublisher` y `HystrixPropertiesStrategy` usando el método `.obtener Instancia()` de `HystrixPlugins`. Un cuadro de diálogo aparece debajo de este código: "Debido a que está registrando una nueva estrategia de concurrencia, tomará todos los demás componentes de Hystrix y luego restablecerá el complemento de Hystrix." Un cuadro de diálogo aparece a la derecha de la línea de código.

```

HystrixCommandExecutionHook comandoExecutionHook = HystrixPlugins

    .obtener Instancia()
    .getCommandExecutionHook();
HystrixPlugins.reset();

HystrixPlugins.getInstance()
    .registerConcurrencyStrategy(
        nueva ThreadLocalAwareStrategy (existingConcurrencyStrategy));
HystrixPlugins.getInstance()
    .registerEventNotifier(eventNotificador);
HystrixPlugins.getInstance()
    .registerMetricsPublisher(métricasPublisher);
HystrixPlugins.getInstance()
    .registerPropertiesStrategy(propertiesStrategy); HystrixPlugins.getInstance()

    .registerCommandExecutionHook(commandExecutionHook);
}

}

```

Esta clase de configuración de Spring básicamente reconstruye el complemento Hystrix que administra todos los diferentes componentes que se ejecutan dentro de su servicio. En el método init() , estás obteniendo referencias a todos los componentes de Hystrix utilizados por el complemento. Luego registra su HystrixConcurrencyStrategy personalizada (ThreadLocalAwareStrategy).

```
HystrixPlugins.getInstance().registerConcurrencyStrategy(
    nueva ThreadLocalAwareStrategy (existingConcurrencyStrategy));
```

Recuerde, Hystrix permite solo una HystrixConcurrencyStrategy. la primavera lo hará intente realizar la conexión automática en cualquier HystrixConcurrencyStrategy existente (si existe). Finalmente, cuando haya terminado, vuelva a registrar los componentes originales de Hystrix que tomado al comienzo del método init() con el complemento Hystrix.

Con estas piezas en su lugar, ahora puede reconstruir y reiniciar su servicio de licencias. y llámelo a través de GET (<http://localhost:8080/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/>) mostrado anteriormente en la figura 5.10. Ahora, cuando se complete esta llamada, debería ver el siguiente resultado en la ventana de su consola:

```
ID de correlación de UserContext: TEST-CORRELATION-ID
LicenseServiceController ID de correlación: TEST-CORRELATION-ID
LicenseService.getLicenseByOrg Correlación: TEST-CORRELATION-ID
```

Es mucho trabajo producir un pequeño resultado, pero lamentablemente es necesario cuando use Hystrix con aislamiento de nivel THREAD.

## 5.10 Resumen

Al diseñar aplicaciones altamente distribuidas, como una basada en microservicios. aplicación, se debe tener en cuenta la resiliencia del cliente.

Los fallos absolutos de un servicio (por ejemplo, el servidor falla) son fáciles de detectar y tratar con.

Un único servicio con bajo rendimiento puede desencadenar un efecto en cascada de agotamiento de recursos, ya que los subprocesos en el cliente que llama se bloquean esperando a que se complete un servicio.

Tres patrones centrales de resiliencia del cliente son el patrón de interruptor, el patrón de respaldo y el patrón de mamparo. El patrón de

disyuntor busca eliminar las llamadas al sistema degradadas y de ejecución lenta para que las llamadas fallen rápidamente y eviten el agotamiento de los recursos.

El patrón alternativo le permite a usted, como desarrollador, definir rutas de código alternativas en caso de que falle una llamada de servicio remoto o que falle el disyuntor de la llamada.

El patrón de encabezado masivo separa las llamadas de recursos remotos entre sí, aislando las llamadas a un servicio remoto en su propio grupo de subprocesos. Si un conjunto de llamadas de servicio falla, no se debe permitir que sus fallas consuman todos los recursos del contenedor de la aplicación.

Spring Cloud y las bibliotecas Netflix Hystrix proporcionan implementaciones para patrones de disyuntor, respaldo y mamparo.

Las bibliotecas de Hystrix son altamente configurables y se pueden configurar a nivel global, de clase y de grupo

de subprocesos. Hystrix admite dos modelos de aislamiento: THREAD y SEMAPHORE. El modelo de aislamiento predeterminado de Hystrix, THREAD, aísla completamente una llamada protegida de Hystrix, pero no propaga el contexto del subproceso principal al

subproceso administrado de Hystrix. El otro modelo de aislamiento de Hystrix, SEMAPHORE, no utiliza un hilo separado para realizar una llamada a Hystrix. Si bien esto es más eficiente, también expone el servicio a un comportamiento impredecible

si Hystrix interrumpe la llamada. Hystrix le permite injectar el contexto del subproceso principal en un subproceso administrado por Hystrix a través de una implementación personalizada de HystrixConcurrency

# Enrutamiento de servicios con Nube de primavera y Zuul

## Este capítulo cubre

Usar una puerta de enlace de servicios con sus  
microservicios

Implementación de una puerta de enlace de servicios utilizando Spring  
Nube y Netflix Zuul

Mapeo de rutas de microservicios en Zuul

Creación de filtros para utilizar ID de correlación y seguimiento

Enrutamiento dinámico con Zuul

En una arquitectura distribuida como la de microservicios, llegará un punto donde deberá garantizar que se produzcan comportamientos clave, como seguridad, registro y seguimiento de usuarios en múltiples llamadas de servicio. Para implementar esta funcionalidad, querrá que estos atributos se apliquen de manera consistente en todos sus servicios sin la necesidad de que cada equipo de desarrollo individual construya sus propias soluciones. Si bien es posible utilizar una biblioteca o marco común para ayudar a crear estas capacidades directamente en un servicio individual, hacerlo tiene tres trascendencia.

Primeramente, es difícil implementar consistentemente estas capacidades en cada servicio que se ofrece. construido. Los desarrolladores se centran en ofrecer funcionalidad y, en el torbellino de la actividad diaria, pueden olvidarse fácilmente de implementar el registro o seguimiento del servicio. (Yo personalmente soy culpable de esto.) Desafortunadamente, para aquellos de nosotros que trabajamos en un entorno fuertemente regulado industria, como servicios financieros o atención médica, que muestren información consistente y documentada El comportamiento de sus sistemas es a menudo un requisito clave para cumplir con las normas gubernamentales. regulaciones.

En segundo lugar, implementar adecuadamente estas capacidades es un desafío. Cosas como La seguridad de los microservicios puede ser difícil de configurar y configurar con cada servicio que se esté ejecutando. implementado. Impulsar las responsabilidades para implementar una preocupación transversal como La seguridad hasta los equipos de desarrollo individuales aumenta en gran medida las probabilidades de que alguien no lo implementará correctamente o se olvidará de hacerlo.

En tercer lugar, ahora ha creado una fuerte dependencia en todos sus servicios. Cuanto más capacidades que usted construye en un marco común compartido en todos sus servicios, el Más difícil es cambiar o agregar comportamiento en su código común sin tener que hacerlo. recompile y vuelva a implementar todos sus servicios. Puede que esto no parezca gran cosa cuando Tienes seis microservicios en tu aplicación, pero es un gran problema cuando tienes un mayor número de servicios, quizás 30 o más. De repente, una actualización de las capacidades principales integradas en una biblioteca compartida se convierte en un proceso de migración que dura meses.

Para resolver este problema, es necesario abstraer estas preocupaciones transversales en un servicio que pueda funcionar de forma independiente y actuar como filtro y enrutador para todos los microservicios. Llamadas en su aplicación. Esta preocupación transversal se denomina puerta de enlace de servicios. Los clientes de su servicio ya no llaman directamente a un servicio. En cambio, todas las llamadas se enrutan a través del puerta de enlace de servicio, que actúa como un único punto de aplicación de políticas (PEP), y luego encaminado a un destino final.

En este capítulo, veremos cómo usar Spring Cloud y Zuul de Netflix para implementar una puerta de enlace de servicios. Zuul es la implementación de la puerta de enlace de servicios de código abierto de Netflix. Específicamente, veremos cómo usar Spring Cloud y Zuul para

Coloque todas las llamadas de servicio detrás de una única URL y asigne esas llamadas mediante el descubrimiento de servicios a sus instancias de servicio reales.

Inyectar ID de correlación en cada llamada de servicio que fluye a través de la puerta de enlace de servicio Inyectar el ID de correlación a partir de la respuesta HTTP enviada desde el cliente Cree un mecanismo de enrutamiento dinámico que enrute organizaciones individuales específicas a un punto final de instancia de servicio que sea diferente al de los demás. esta usando

Profundicemos en más detalles sobre cómo encaja una puerta de enlace de servicios en los microservicios generales. que se está construyendo en este libro.

## 6.1 ¿Qué es una pasarela de servicios?

Hasta ahora, con los microservicios que ha creado en capítulos anteriores, ha llamó directamente a los servicios individuales a través de un cliente web o los llamó mediante programación a través de un motor de descubrimiento de servicios como Eureka.

¿Qué es una puerta de enlace de servicios?

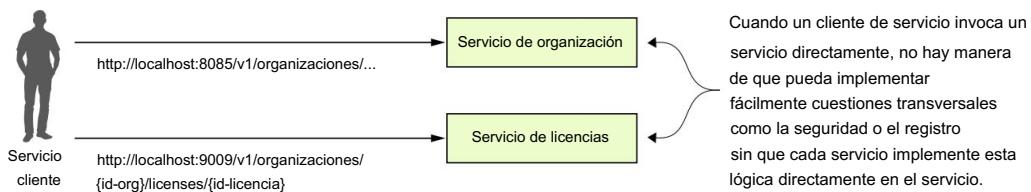


Figura 6.1 Sin una puerta de enlace de servicios, el cliente del servicio llamará a distintos puntos finales para cada servicio.

Una puerta de enlace de servicio actúa como intermediario entre el cliente del servicio y un servicio. siendo invocado. El cliente del servicio habla solo con una única URL administrada por el servicio. puerta. La puerta de enlace del servicio separa la ruta que llega desde el cliente del servicio. llamar y determina qué servicio está intentando invocar el cliente de servicio. La Figura 6.2 ilustra cómo, al igual que un policía de "tráfico" que dirige el tráfico, la puerta de enlace del servicio dirige al usuario a una microservicio de destino y la instancia correspondiente. La puerta de enlace de servicio actúa como guardián de todo el tráfico entrante a las llamadas de microservicio dentro de su aplicación. Con una puerta de enlace de servicio implementada, los clientes de su servicio nunca llaman directamente a la URL de un individuo. servicio, sino que realiza todas las llamadas a la puerta de enlace del servicio.

El cliente invoca el servicio llamando a la puerta de enlace de servicios.

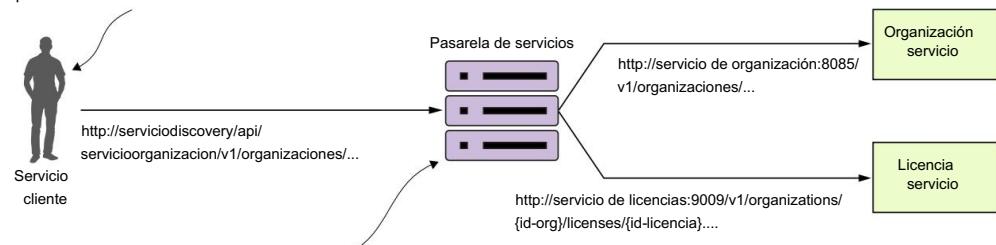


Figura 6.2 La puerta de enlace del servicio se encuentra entre el cliente del servicio y las instancias del servicio correspondientes. Todas las llamadas de servicio (tanto internas como externas) deben fluir a través del portal de servicios.

Debido a que una puerta de enlace de servicio se ubica entre todas las llamadas del cliente a los servicios individuales, también actúa como un punto central de aplicación de políticas (PEP) para las llamadas de servicio. El uso de un PEP centralizado significa que las preocupaciones de servicios transversales se pueden implementar en un solo lugar sin que los equipos de desarrollo individuales tengan que implementar estos preocupaciones. Ejemplos de preocupaciones transversales que se pueden implementar en un servicio puerta de enlace incluye

Enrutamiento estático: una puerta de enlace de servicio coloca todas las llamadas de servicio detrás de una única URL y Ruta API . Esto simplifica el desarrollo ya que los desarrolladores sólo tienen que conocer un punto final de servicio para todos sus servicios.

Enrutamiento dinámico: una puerta de enlace de servicio puede inspeccionar las solicitudes de servicio entrantes y, basándose en los datos de la solicitud entrante, realice un enrutamiento inteligente basado en quién es la persona que llama al servicio. Por ejemplo, los clientes que participan en un programa beta podrían tener todas las llamadas a un servicio enrutadas a un grupo específico de servicios que están ejecutando una versión de código diferente a la que usan todos los demás.

Autenticación y autorización: debido a que todas las llamadas de servicio se enrutan a través de un servicio puerta de enlace, la puerta de enlace de servicio es un lugar natural para comprobar si la persona que llama a un servicio se ha autenticado y está autorizado a realizar la llamada de servicio.

Recopilación y registro de métricas: se puede utilizar una puerta de enlace de servicio para recopilar métricas y registrar información a medida que una llamada de servicio pasa a través del portal de servicio. Puede También utilice el portal de servicios para garantizar que la información clave esté en colocar en la solicitud del usuario para garantizar que el registro sea uniforme. Esto no significa que ¿No debería seguir recopilando métricas de sus servicios individuales, pero más bien, una puerta de enlace de servicios le permite centralizar la recopilación de muchos de sus métricas básicas, como la cantidad de veces que se invoca el servicio y el número de veces que se invoca el servicio. tiempo de respuesta.

### Espere, ¿no es una puerta de enlace de servicio un punto único de falla y un posible cuello de botella?

Anteriormente en el capítulo 4, cuando presenté Eureka, hablé de cómo los balanceadores de carga centralizados pueden ser un punto único de falla y un cuello de botella para sus servicios. Una puerta de enlace de servicio, si no se implementa correctamente, puede conllevar el mismo riesgo. Tenga en cuenta lo siguiente a medida que construye la implementación de su puerta de enlace de servicios.

Los balanceadores de carga siguen siendo útiles cuando se trata de grupos individuales de servicios. En En este caso, un equilibrador de carga ubicado frente a múltiples instancias de puerta de enlace de servicio es un diseño apropiado y garantiza que la implementación de su puerta de enlace de servicios pueda escalar. Tener un balanceador de carga frente a todas sus instancias de servicio no es una buena idea porque se convierte en un cuello de botella.

Mantenga cualquier código que escriba para su puerta de enlace de servicio sin estado. No almacene ninguna información en la memoria para la puerta de enlace del servicio. Si no tiene cuidado, puede limitar la capacidad de escalación de la puerta de enlace y debe asegurarse de que los datos se repliquen en todos las instancias de puerta de enlace de servicio.

Mantenga iluminado el código que escribe para la puerta de enlace de servicio. La puerta de enlace del servicio es el "punto de estrangulamiento" para su invocación de servicio. Código complejo con múltiples llamadas a bases de datos. puede ser la fuente de problemas de rendimiento difíciles de rastrear en el servicio puerta.

Veamos ahora cómo implementar una puerta de enlace de servicios usando Spring Cloud y Netflix. Zuul.

## 6.2 Presentación de Spring Cloud y Netflix Zuul

Spring Cloud se integra con el proyecto de código abierto de Netflix Zuul. Zuul es una puerta de enlace de servicios que es extremadamente fácil de configurar y usar a través de anotaciones de Spring Cloud. Zuul ofrece una serie de capacidades, que incluyen

Mapear las rutas para todos los servicios en su aplicación a una sola URL: Zuul no se limita a una sola URL. En Zuul, puede definir múltiples entradas de ruta, lo que hace que el mapeo de rutas sea extremadamente detallado (cada punto final de servicio obtiene su propio mapeo de rutas). Sin embargo, el primer y más común caso de uso de Zuul es crear un único punto de entrada a través del cual fluirán todas las llamadas de los clientes del servicio.

Crear filtros que puedan inspeccionar y actuar sobre las solicitudes que llegan a través de la puerta de enlace: estos filtros le permiten inyectar puntos de cumplimiento de políticas en su código y realizar una amplia cantidad de acciones en todas sus llamadas de servicio de manera consistente.

Para comenzar con Zuul, deberás hacer tres cosas:

- 1 Configure un proyecto Zuul Spring Boot y configure la dependencia de Maven adecuada.
- 2 Modifique su proyecto Spring Boot con anotaciones de Spring Cloud para indicarle que Será un servicio Zuul.
- 3 Configure Zuul para comunicarse con Eureka (opcional).

### 6.2.1 Configuración del proyecto Zuul Spring Boot

Si ha seguido los capítulos secuencialmente de este libro, el trabajo que está a punto de realizar le resultará familiar. Para construir un servidor Zuul, debe configurar un nuevo servicio Spring Boot y definir las dependencias de Maven correspondientes. Puede encontrar el código fuente del proyecto para este capítulo en el repositorio de GitHub de este libro (<https://github.com/carnellj/spmia-chapter6> ). Afortunadamente, se necesita poco para configurar Zuul en Maven. Solo necesita definir una dependencia en su archivo zuulsvr/pom.xml:

```
<dependencia>
    <groupId>org.springframework.cloud</groupId> <artifactId>spring-
    cloud-starter-zuul</artifactId>
</dependencia>
```

Esta dependencia le dice al marco de Spring Cloud que este servicio ejecutará Zuul e inicializará Zuul de manera adecuada.

### 6.2.2 Uso de la anotación Spring Cloud para el servicio Zuul

Una vez que haya definido las dependencias de Maven, debe anotar la clase de arranque para los servicios de Zuul. La clase de arranque para la implementación del servicio Zuul se puede encontrar en la clase zuulsvr/src/main/java/com/thinkmechanix/zuulsvr/Application.java .

#### Listado 6.1 Configurando la clase de arranque del servidor Zuul

```

paquete com.thinkmechanix.zuulsvr;

importar org.springframework.boot.SpringApplication; importar
org.springframework.boot.autoconfigure.SpringBootApplication; importar
org.springframework.cloud.netflix.zuul.EnableZuulProxy; importar
org.springframework.context.annotation.Bean;

@SpringBootApplication
@EnableZuulProxy clase
pública ZuulServerApplication {
    principal vacío estático público (String[] args) {
        SpringApplication.run (
            ZuulServerApplication.clase, argumentos);

    }
}

```

Permite que el servicio sea un servidor Zuul.

Eso es todo. Solo es necesario implementar una anotación: `@EnableZuulProxy`.

**NOTA** Si consulta la documentación o tiene activada la función de autocompletar, es posible que observe una anotación llamada `@EnableZuulServer`. El uso de esta anotación creará un servidor Zuul que no carga ninguno de los filtros de proxy inverso de Zuul ni utiliza Netflix Eureka para el descubrimiento de servicios. (Entraremos en el tema de la integración de Zuul y Eureka en breve). `@EnableZuulServer` se utiliza cuando desea crear su propio servicio de enrutamiento y no utilizar ninguna capacidad prediseñada de Zuul. Un ejemplo de esto sería si quisiera utilizar Zuul para integrarse con un motor de descubrimiento de servicios distinto de Eureka (por ejemplo, Consul). En este libro solo usaremos la anotación `@EnableZuulProxy`.

### 6.2.3 Configurar Zuul para comunicarse con Eureka

El servidor proxy Zuul está diseñado de forma predeterminada para funcionar en los productos Spring. Como tal, Zuul usará automáticamente Eureka para buscar servicios por sus ID de servicio y luego usará Netflix Ribbon para equilibrar la carga de las solicitudes del lado del cliente desde Zuul.

**NOTA** A menudo leo capítulos desordenados en un libro, saltando a los temas específicos que más me interesan. Si haces lo mismo y no sabes qué son Netflix Eureka y Ribbon, te sugiero que leas el capítulo 4 antes de continuar. más. Zuul utiliza mucho esas tecnologías para realizar el trabajo, por lo que comprender las capacidades de descubrimiento de servicios que Eureka y Ribbon aportan hará que comprender Zuul sea mucho más fácil.

El último paso en el proceso de configuración es modificar el archivo `zuulsvr/src/main/resources/application.yml` de su servidor Zuul para que apunte a su servidor Eureka. La siguiente lista muestra la configuración de Zuul necesaria para que Zuul se comunique con Eureka. El

La configuración en el listado debería resultar familiar porque es la misma configuración. repasamos en el capítulo 4.

#### Listado 6.2 Configurando el servidor Zuul para hablar con Eureka

```
eureka:  
  instancia:  
    preferir dirección IP: verdadero  
  cliente:  
    registrarseConEureka: verdadero  
    buscarRegistro: verdadero  
    URL de servicio:  
      Zona predeterminada: http://localhost:8761/eureka/
```

### 6.3 Configurar rutas en Zuul

Zuul en el fondo es un proxy inverso. Un proxy inverso es un servidor intermedio que se encuentra entre el cliente que intenta llegar a un recurso y el recurso mismo. El cliente no tiene La idea es que incluso se esté comunicando con un servidor que no sea un proxy. El proxy inverso toma se encarga de capturar la solicitud del cliente y luego llama al recurso remoto en nombre del cliente.

En el caso de una arquitectura de microservicios, Zuul (su proxy inverso) toma una llamada de microservicio de un cliente y la reenvía al servicio descendente. El ser-  
El vicepresidente cree que solo se está comunicando con Zuul. Para que Zuul se comunique con  
Para los clientes descendentes, Zuul debe saber cómo asignar la llamada entrante a una ruta descendente. Zuul tiene varios mecanismos para hacer esto, incluyendo

- Mapeo automatizado de rutas mediante descubrimiento de servicios
- Mapeo manual de rutas mediante descubrimiento de servicios
- Mapeo manual de rutas usando URL estáticas

#### 6.3.1 Rutas de mapeo automatizadas mediante descubrimiento de servicios

Todos los mapeos de rutas para Zuul se realizan definiendo las rutas en zuulsrc/src/main/ archivo recursos/aplicación.yml. Sin embargo, Zuul puede enrutar solicitudes automáticamente según en sus ID de servicio con configuración cero. Si no especifica ninguna ruta, Zuul utilizar automáticamente el ID del servicio Eureka del servicio al que se llama y asignarlo a un instancia de servicio descendente. Por ejemplo, si quisiera llamar al servicio de su organización y utilizará el enrutamiento automatizado a través de Zuul, le pediría a su cliente que llame al Instancia de servicio Zuul, utilizando la siguiente URL como punto final:

<http://localhost:5555/organizationservice/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a>

Se accede a su servidor Zuul a través de <http://localhost:5555>. El servicio que está intentando invocar (servicio de organización) está representado por la primera parte de la ruta del punto final del servicio.

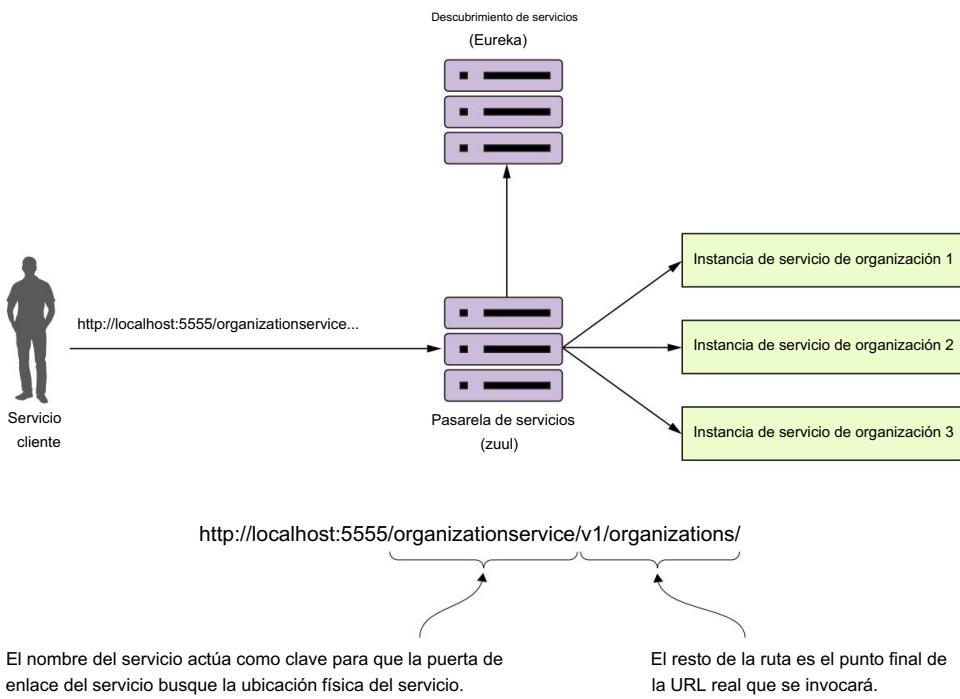


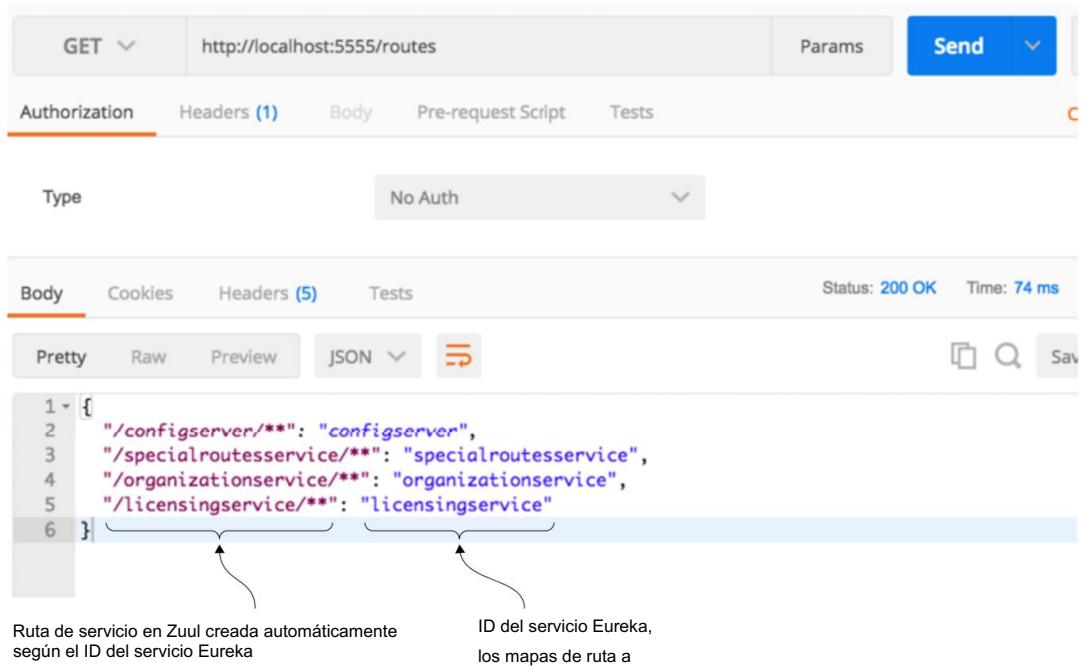
Figura 6.3 Zuul utilizará el **nombre de la aplicación** del servicio de la organización para asignar solicitudes a instancias del servicio de la organización.

La figura 6.3 ilustra este mapeo en acción.

Lo bueno de usar Zuul con Eureka es que ahora no sólo tienes un único punto final a través del cual puedes realizar llamadas, pero con Eureka, también puedes agregar y eliminar instancias de un servicio sin tener que modificar Zuul. Por ejemplo, puedes agregar un nuevo servicio a Eureka, y Zuul lo enrutaría automáticamente porque se comunicaría con Eureka sobre dónde están los puntos finales reales de los servicios físicos situados.

Si deseas ver las rutas que gestiona el servidor Zuul, puedes acceder a las rutas a través del punto final `/routes` en el servidor Zuul. Esto devolverá una lista de todos los asignaciones en su servicio. La Figura 6.4 muestra el resultado al presionar `http://localhost:5555/routes`.

En la figura 6.4 se muestran las asignaciones de los servicios registrados con Zuul en la lado izquierdo del cuerpo JSON devuelto por las llamadas `/route`. El verdadero Eureka Los ID de servicio a los que se asignan las rutas se muestran a la derecha.



The screenshot shows a Postman request to `http://localhost:5555/routes`. The response body is a JSON object:

```

1 * {
2   "/configserver/**": "configserver",
3   "/specialroutesservice/**": "specialroutesservice",
4   "/organizationservice/**": "organizationservice",
5   "/licensingservice/**": "licensingservice"
6 }

```

Annotations explain the mapping:

- Ruta de servicio en Zuul creada automáticamente según el ID del servicio Eureka
- ID del servicio Eureka, los mapas de ruta a

Figura 6.4 Cada servicio mapeado en Eureka ahora se mapeará como una ruta Zuul.

### 6.3.2 Mapeo de rutas manualmente usando el descubrimiento de servicios

Zuul le permite ser más detallado al permitirle definir explícitamente la ruta mapeos en lugar de depender únicamente de las rutas automatizadas creadas con el servicio Identificación del servicio Eureka . Suponga que desea simplificar la ruta acortando el nombre de la organización en lugar de acceder al servicio de su organización en Zuul a través del ruta predeterminada de `/organizationservice/v1/organizations/{organization-id}`. Puede hacer esto definiendo manualmente el mapeo de ruta en `zuulsrc/src/main/resources/aplicación.yml`:

```

zuul:
  rutas:
    servicio de organización: /organización/**

```

Al agregar esta configuración, ahora puede acceder al servicio de organización presionando el botón `/organización/v1/organizaciones/{id-organización}` ruta. Si marca el nuevamente el punto final del servidor Zuul, debería ver los resultados que se muestran en la figura 6.5.

The screenshot shows the Zuul Routes API endpoint in Postman. The request method is GET, the URL is `http://localhost:5555/routes`, and the response status is 200 OK with a time of 29 ms. The response body is a JSON object:

```

1 {  
2   "/organization/**": "organizationservice",  
3   "/licensing/**": "licensingservice",  
4   "/configserver/**": "configserver",  
5   "/specialroutesservice/**": "specialroutesservice",  
6   "/organizationservice/**": "organizationservice",  
7   "/licensingservice/**": "licensingservice"  
8 }

```

Annotations on the right side of the JSON output:

- A callout points to the entry `"/organization/**": "organizationservice"` with the text: "Todavía tenemos aquí la ruta basada en ID del servicio Eureka."
- A callout points to the entry `"/organizationservice/**": "organizationservice"` with the text: "Observe la ruta personalizada para el servicio de organización."

Figura 6.5 Los resultados de la llamada Zuul /routes con un mapeo manual del servicio de organización

Si observa detenidamente la figura 6.5, notará que hay dos entradas para el servicio de organización. La primera entrada de servicio es la asignación que definió en el archivo application.yml: `"organización/**": "organizationservice"`. El segundo servicio La entrada es el mapeo automático creado por Zuul basado en el servicio de la organización. ID de Eureka : `"/servicioorganización/**": "servicioorganización"`.

**NOTA** Cuando utiliza el mapeo de rutas automatizado donde Zuul expone el servicio basándose únicamente en el ID del servicio Eureka, si no hay instancias del servicio en ejecución, Zuul no expondrá la ruta para el servicio. Sin embargo, si asigna manualmente una ruta a un ID de descubrimiento de servicio y no hay instancias registradas en Eureka, Zuul seguirá mostrando la ruta. Si intenta llamar a la ruta para Si el servicio no existe, Zuul devolverá un error 500.

Si desea excluir el mapeo automatizado de la ruta de ID del servicio Eureka y Sólo tienes disponible la ruta de servicio de la organización que has definido, puedes agregar una Parámetro Zuul adicional a su archivo application.yml, llamado servicios ignorados.

El siguiente fragmento de código muestra cómo se puede utilizar el atributo de servicios ignorados para excluir el servicio de organización ID del servicio Eureka de las asignaciones automatizadas realizadas por Zuul:

```
zuul:  
    servicios-ignorados: rutas 'servicio de organización':  
        servicio de organización: /organización/**
```

El atributo de servicios ignorados le permite definir una lista separada por comas de ID de servicios de Eureka que desea excluir del registro. Ahora, cuando llame al punto final /routes en Zuul, solo debería ver el mapeo de servicios de la organización que ha definido. La Figura 6.6 muestra el resultado de este mapeo.

The screenshot shows a Postman request to `http://localhost:5555/routes`. The response body is a JSON object with one entry:

```
1 {  
2     "/configserver/**": "configserver",  
3     "/specialroutesservice/**": "specialroutesservice",  
4     "/organizationservice/**": "organizationservice",  
5     "/licensingservice/**": "licensingservice"  
6 }
```

A callout arrow points from the text "Ahora sólo hay una entrada de servicio de organización." to the single entry in the JSON.

Figura 6.6 Ahora solo hay un servicio de organización definido en Zuul.

Si desea excluir todas las rutas basadas en Eureka, puede establecer el atributo de servicios ignorados en `"**"`.

Un patrón común con una puerta de enlace de servicios es diferenciar las rutas API de las rutas de contenido anteponiendo todas las llamadas de servicio con un tipo de etiqueta como `/api`. Apoyos zuul

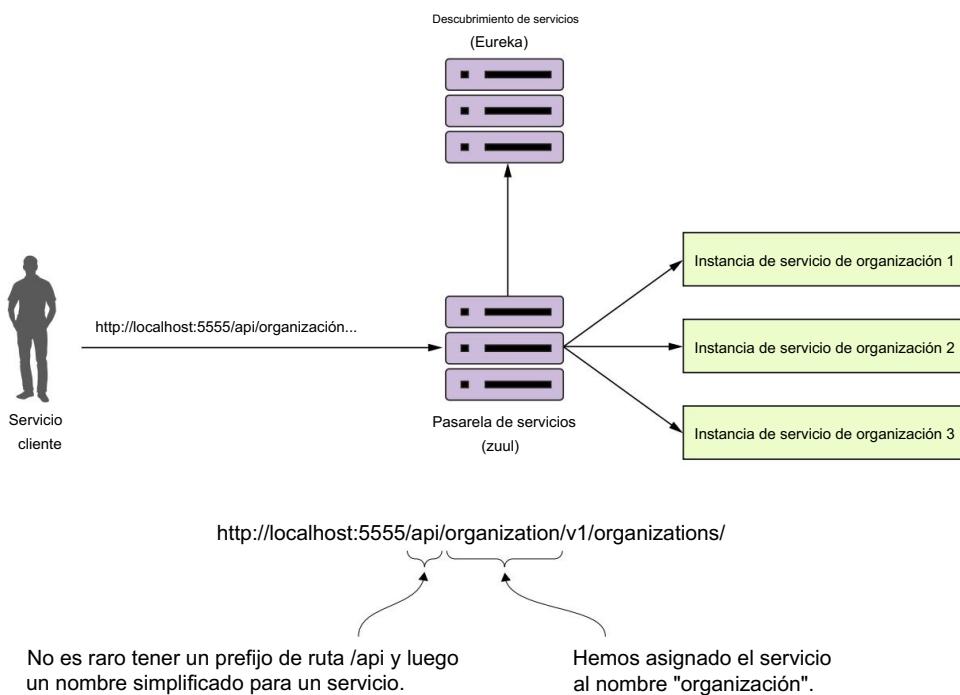


Figura 6.7 Usando un prefijo, Zuul asignará un prefijo /api a cada servicio que administre.

esto usando el atributo `prefix` en la configuración de Zuul. La figura 6.7 muestra conceptualmente cómo se verá este prefijo de mapeo.

En el siguiente listado, veremos cómo configurar rutas específicas a su persona organización y servicios de licencia, excluir todos los servicios generados por eureka, y prefije sus servicios con un prefijo /api .

#### Listado 6.3 Configurar rutas personalizadas con un prefijo

```

zuul:
  servicios ignorados: prefijo **: /api
  Todo definido →
    los servicios serán
    tener prefijo
    con /api.

rutas:
  servicio de organización: /organización/** servicio de licencias: /
  licencias/** ←
    El servicio de su organización y el servicio de licencias se
    asignan a los puntos finales de organización y licencias,
    respectivamente.
  
```

El atributo de servicios ignorados está configurado \* para excluir el registro de todas las rutas basadas en ID de servicio eureka.

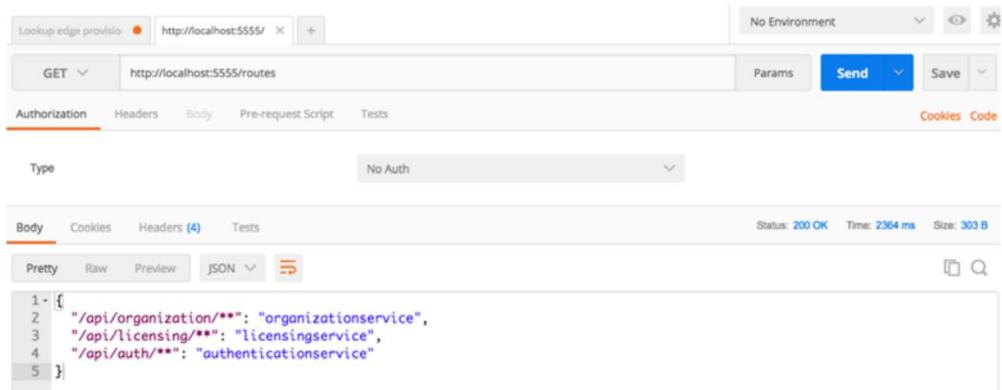


Figura 6.8 Sus rutas en Zuul ahora tienen un prefijo /api.

Una vez realizada esta configuración y recargado el servicio Zuul, deberás consultar las dos entradas siguientes al acceder al punto final /route : /api/organization y /api/licensing. La figura 6.8 muestra estas entradas.

Veamos ahora cómo puedes usar Zuul para asignar URL estáticas. Las URL estáticas son URL que apuntan a servicios que no están registrados con un motor de descubrimiento de servicios Eureka.

### 6.3.3 Mapeo manual de rutas usando URL estáticas

Zuul se puede utilizar para enrutar servicios que no gestiona Eureka. En estos casos, Zuul Se puede configurar para enrutar directamente a una URL definida estéticamente. Por ejemplo, imaginemos que su servicio de licencia está escrito en Python y aún desea utilizarlo como proxy Zuul. Usaría la configuración de Zuul en la siguiente lista para lograr esto.

#### Listado 6.4 Mapeo del servicio de licencias a una ruta estática

```

zuul:
  rutas:
    licenciaestatica: ruta: /
      licensesstatic/** URL: http://
      licenseservice-static:8081
  
```

La estática ruta para tu servicio de licencia

Nombre clave que Zuul utilizará para identificar el servicio internamente.

Ha configurado una instancia estática de su servicio de licencia. eso será llamado directamente, no a través de Eureka por Zuul.

The screenshot shows a Postman request to `http://localhost:5555/routes`. The response body is a JSON object:

```

1 - {
2   "/api/licensesstatic/**": "http://licenseservice-static:8081",
3   "/api/organization/**": "organizationservice",
4   "/api/licensing/**": "licensingservice",
5   "/api/auth/**": "authenticationservice"
6 }

```

A callout arrow points from the text "Nuestra entrada de ruta estática" to the line containing the URL `"/api/licensesstatic/**": "http://licenseservice-static:8081"`.

Figura 6.9 Ahora ha asignado una ruta estática a su servicio de licencias.

Una vez que se haya realizado este cambio de configuración, puede acceder al punto final `/routes` y vea la ruta estática agregada a Zuul. La Figura 6.10 muestra los resultados de la `/ lista de rutas`.

En este punto, el punto final de licencia estática no utilizará Eureka y en su lugar enrute directamente la solicitud al punto final `http://licenseservice-static:8081`. El problema es que al pasar por alto Eureka, sólo tienes una única ruta para apuntar solicitudes en. Afortunadamente, puede configurar Zuul manualmente para deshabilitar la integración de Ribbon con Eureka y luego enumerar las instancias de servicio individuales que Ribbon cargará en balde. ance en contra. El siguiente listado muestra esto.

#### Listado 6.5 Servicio de asignación de licencias estáticamente a múltiples rutas

```

zuul:
  rutas:
    licenciaestática:
      ruta: /licensesstatic/** serviceld:
        licenciaestática
      Define un ID de servicio que se utilizará para
      buscar el servicio en Ribbon
    cinta:
      eureka:
        habilitado: falso
      Deshabilita el
      soporte de Eureka en Ribbon
    licenciaestática:
      cinta:
        lista de servidores: http://licenseservice-static1:8081,
        http://licenciaservicio-static2:8082
      Lista de servidores utilizados
      para enrutar la solicitud a

```

Una vez que esta configuración esté implementada, una llamada al punto final /routes ahora muestra que la ruta /api/licensesstatic se ha asignado a un ID de servicio llamado licenses-tatic. La figura 6.10 muestra esto.

```

1 - {
2   "/api/licensesstatic/**": "licensesstatic",
3   "/api/organization/**": "organizationservice",
4   "/api/licensing/**": "licensingservice",
5   "/api/auth/**": "authenticationservice"
6 }

```

Nuestra entrada de ruta estática es ahora detrás de una identificación de servicio.

Figura 6.10 Ahora verá que /api/licensesstatic ahora se asigna a un ID de servicio llamado Licensestatic

### Tratar con servicios que no son JVM

El problema con el mapeo estático de rutas y la desactivación del soporte de Eureka en Ribbon es que has desactivado el soporte Ribbon para todos tus servicios que se ejecutan a través de tu Zuul puerta de enlace del servicio. Esto significa que se colocará más carga en sus servidores Eureka, porque Zuul no puede usar Ribbon para almacenar en caché la búsqueda de servicios. Recuerda, cinta no llama a Eureka cada vez que hace una llamada. En cambio, almacena en caché la ubicación del instancias de servicio localmente y luego verifica periódicamente con Eureka los cambios. Con Fuera de escena, Zuul llamará a Eureka cada vez que necesite resolver la ubicación de un servicio.

Anteriormente en este capítulo, hablé sobre cómo podría terminar con múltiples puertas de enlace de servicio donde se aplicarían diferentes reglas y políticas de enrutamiento según el tipo. de servicios que se llaman. Para aplicaciones que no sean JVM, puede configurar un Zuul separado servidor para manejar estas rutas. Sin embargo, descubrí que con lenguajes no basados en JVM, es mejor configurar una instancia "Sidecar" de Spring Cloud. La primavera El sidecar en la nube le permite registrar servicios que no sean JVM con una instancia de Eureka y luego

(continuado)

apoderarlos a través de Zuul. No cubro Spring Sidecar en este libro porque no estás escribiendo ningún servicio que no sea JVM, pero es extremadamente fácil configurar una instancia de sidecar. Puede encontrar instrucciones sobre cómo hacerlo en el sitio web de Spring Cloud (<http://cloud.spring.io/spring-cloud-netflix/spring-cloud-netflix.html#spring-cloud-ribbon-without-eureka>).

#### 6.3.4 Recargar dinámicamente la configuración de ruta

Lo siguiente que veremos en términos de configuración de rutas en Zuul es cómo recargar rutas dinámicamente. La capacidad de recargar rutas dinámicamente es útil porque le permite cambiar el mapeo de rutas sin tener que reciclar los servidores Zuul. Las rutas existentes se pueden modificar rápidamente y las nuevas rutas agregadas deben pasar por el acto de reciclar cada servidor Zuul en su entorno. En el capítulo 3, cubrimos cómo utilizar el servicio Spring Cloud Configuration para externalizar datos de configuración de microservicios. Puede utilizar la configuración de Spring Cloud para externalizar las rutas de Zuul.

En los ejemplos de EagleEye, puede configurar una nueva carpeta de aplicación en su repositorio de configuración (<http://github.com/carnellj/config-repo>) llamado zuulservice. Al igual que sus servicios de organización y licencias, creará tres archivos (zuulservice.yml, zuulservice-dev.yml y zuulservice-prod.yml) que contendrán su configuración de ruta.

Para ser coherente con los ejemplos de la configuración del capítulo 3, cambié los formatos de ruta para pasar de un formato jerárquico al "." formato. La inicial

La configuración de ruta tendrá una única entrada:

```
zuul.prefix=/api
```

Si accede al punto final /routes , debería ver todos sus servicios basados en Eureka mostrados actualmente en Zuul con el prefijo /api. Ahora, si desea agregar nuevas asignaciones de rutas sobre la marcha, todo lo que tiene que hacer es realizar los cambios en el archivo de configuración y luego enviarlos nuevamente al repositorio de Git de donde Spring Cloud Config extrae sus datos de configuración. Por ejemplo, si desea deshabilitar todos los registros de servicios basados en Eureka y exponer solo dos rutas (una para la organización y otra para el servicio de licencia), puede modificar los archivos zuulservice-\*.yml para que se vean así:

```
zuul.ignored-services: '*' zuul.prefix: /
api
zuul.routes.organizationservice: /organization/**
zuul.routes.organizationservice: /licensing/**
```

Luego puedes enviar los cambios a GitHub. Zuul expone una ruta /actualización de punto final basada en POST que hará que vuelva a cargar su configuración de ruta. Una vez que se presiona /refresh , si luego presiona el punto final /routes , verá que las dos nuevas rutas están expuestas y todas las rutas basadas en Eureka desaparecieron.

### 6.3.5 Zuul y tiempos de espera del servicio

Zuul utiliza las bibliotecas Hystrix y Ribbon de Netflix para ayudar a evitar que las llamadas de servicio de larga duración afecten el rendimiento de la puerta de enlace de servicios. De forma predeterminada, Zuul finalizará y devolverá un error HTTP 500 para cualquier llamada que tarde más de un segundo en procesar una solicitud. (Este es el valor predeterminado de Hystrix). Afortunadamente, puede configurar este comportamiento configurando las propiedades de tiempo de espera de Hystrix en la configuración de su servidor Zuul.

Para configurar el tiempo de espera de Hystrix para todos los servicios que se ejecutan a través de Zuul, puede usar la propiedad `hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds`.

Por ejemplo, si desea establecer el tiempo de espera predeterminado de Hystrix en 2,5 segundos, puede usar la siguiente configuración en el archivo de configuración Spring Cloud de Zuul:

```
zuul.prefix: /api  
zuul.routes.organizationservice: /organization/** zuul.routes.licensingservice: /  
licensing/** zuul.debug.request: true
```

```
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 2500
```

Si necesita configurar el tiempo de espera de Hystrix para un servicio específico, puede reemplazar la parte predeterminada de la propiedad con el nombre de ID del servicio Eureka cuyo tiempo de espera desea anular. Por ejemplo, si quisiera cambiar solo el tiempo de espera del servicio de licencias a tres segundos y dejar que el resto de los servicios usen el tiempo de espera predeterminado de Hystrix, podría usar algo como esto en su configuración:

```
hystrix.command.licensingservice.execution.isolation.thread.timeoutInMillisec  
segundos: 3000
```

Por último, es necesario tener en cuenta otra propiedad del tiempo de espera. Si bien anuló el tiempo de espera de Hystrix, Netflix Ribbon también anula el tiempo de espera de cualquier llamada que demore más de cinco segundos. Si bien le recomiendo que revise el diseño de cualquier llamada que demore más de cinco segundos, puede anular el tiempo de espera de la cinta configurando la siguiente propiedad: `servicename.ribbon.ReadTimeout`. Por ejemplo, si quisiera anular el servicio de licencias para que tenga un tiempo de espera de siete segundos, usaría la siguiente configuración:

```
hystrix.command.licensingservice.ejecución.  
aislamiento.thread.timeoutInMillisegundos: 7000  
licensingservice.ribbon.ReadTimeout: 7000
```

**NOTA** Para configuraciones de más de cinco segundos, debe configurar los tiempos de espera de Hystrix y Ribbon.

## 6.4 El verdadero poder de Zuul: filtros

Si bien poder enviar todas las solicitudes a través de la puerta de enlace de Zuul le permite simplificar sus invocaciones de servicio, el verdadero poder de Zuul entra en juego cuando desea escribir una lógica personalizada que se aplicará a todas las llamadas de servicio que fluyen a través de ella.

la salida. La mayoría de las veces, esta lógica personalizada se utiliza para aplicar un conjunto coherente de políticas de aplicaciones, como seguridad, registro y seguimiento, en todos los servicios.

Estas políticas de aplicación se consideran preocupaciones transversales porque desea para que se apliquen a todos los servicios de su aplicación sin tener que modificarlos cada servicio para implementarlos. De esta manera, los filtros Zuul se pueden utilizar de forma similar. De la misma manera como un filtro de servlet J2EE o un Spring Aspect que puede interceptar un amplio conjunto de comportamientos y decorar o cambiar el comportamiento de la llamada sin que el codificador original sea consciente del cambio. Mientras que un filtro de servlet o Spring Aspect está localizado en un área específica servicio, el uso de filtros Zuul y Zuul le permite implementar inquietudes transversales en todos los servicios que pasan por Zuul.

Zuul le permite crear una lógica personalizada utilizando un filtro dentro de la puerta de enlace de Zuul. Un filtro le permite implementar una cadena de lógica de negocios que pasa por cada solicitud de servicio a medida que se implementa.

Zuul admite tres tipos de filtros:

Prefiltros: se invoca un prefiltro antes de que se produzca la solicitud real al destino de destino con Zuul.

Un prefiltro suele realizar la tarea de garantizar que el servicio tenga un formato de mensaje consistente (los encabezados HTTP clave están en su lugar, por ejemplo) o actúa como un guardián para garantizar que el usuario que llama al servicio está autenticado (son quienes dicen ser) y autorizado (son quienes dicen ser). Pueden hacer lo que piden).

Filtros de publicación:

se invoca un filtro de publicación después de que se ha invitado el servicio de destino y se envía una respuesta al cliente. Por lo general, se implementará un filtro posterior para registrar la respuesta del servicio de destino, manejar errores o auditar.

la respuesta para información sensible.

Filtros de ruta: el filtro de ruta se utiliza para interceptar la llamada antes de que se invoque el servicio de destino.

Por lo general, se utiliza un filtro de ruta para determinar si algún nivel de

Es necesario realizar un enrutamiento dinámico. Por ejemplo, más adelante en el capítulo use un filtro a nivel de ruta que enrutará entre dos versiones diferentes de la misma

servicio de modo que un pequeño porcentaje de llamadas a un servicio se enrute a una nueva versión de un servicio en lugar del servicio existente. Esto le permitirá exponer una un pequeño número de usuarios a la nueva funcionalidad sin que todos utilicen la nuevo servicio.

La Figura 6.11 muestra cómo los filtros previo, posterior y de ruta encajan en términos de procesar la solicitud de un cliente de servicio.

Si sigue el flujo establecido en la figura 6.11, verá que todo comienza con un cliente de servicio que realiza una llamada a un servicio expuesto a través del portal de servicio. Desde allí se llevan a cabo las siguientes actividades:

1 Zuul invocará todos los prefiltros definidos en la puerta de enlace de Zuul como una solicitud.

Entrar por la puerta de entrada Zuul. Los prefiltros pueden inspeccionar y modificar una solicitud HTTP antes de llegar al servicio real. Un prefiltro no puede redirigir al usuario a un punto final o servicio diferente.

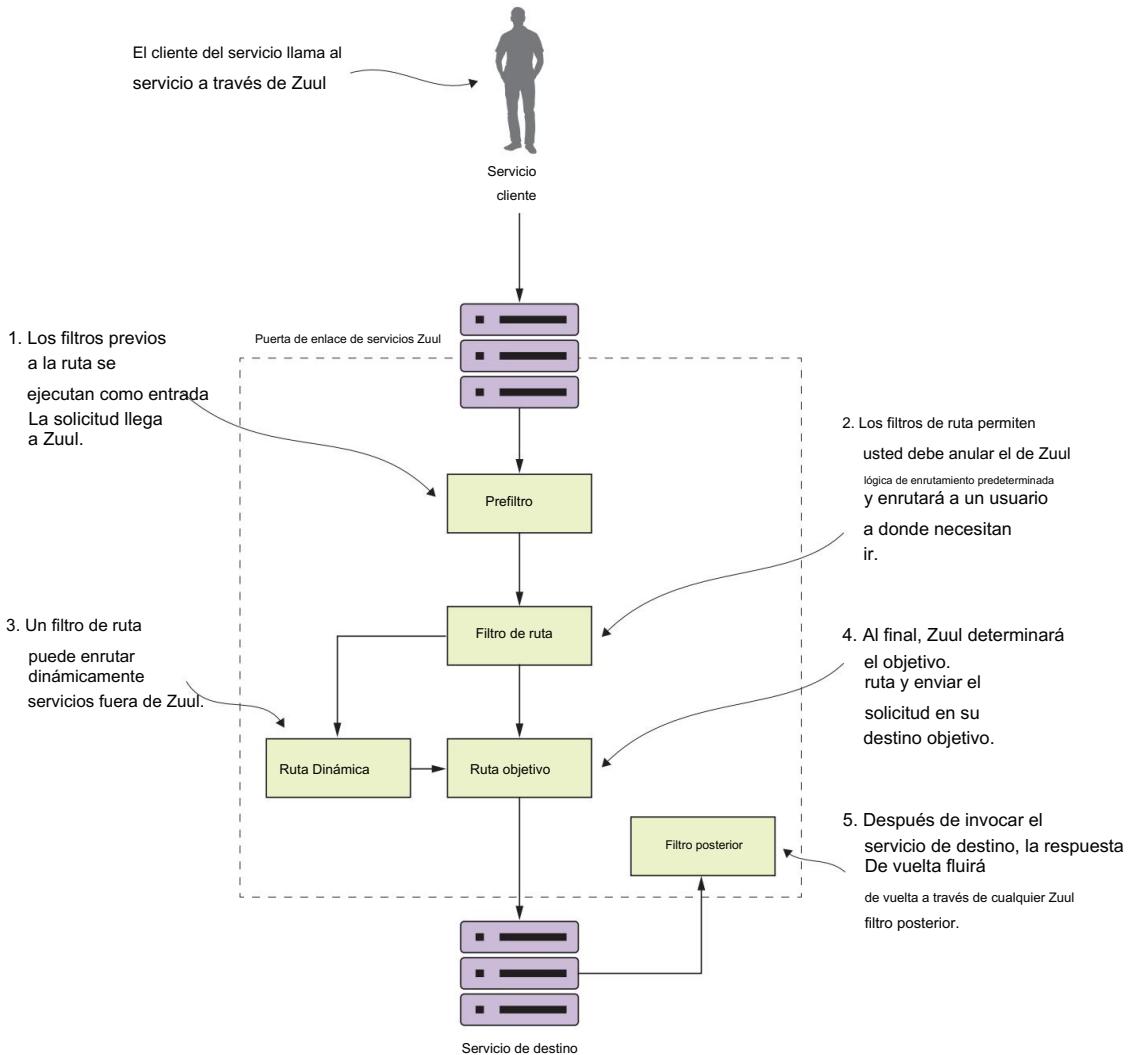


Figura 6.11 Los filtros previo, de ruta y posterior forman una tubería por la que fluye la solicitud de un cliente. Cuando una solicitud llega a Zuul, estos filtros pueden manipular la solicitud entrante.

**2** Despues de que Zuul ejecute los prefiltros contra la solicitud entrante, Zuul ejecutará cualquier filtro de ruta definido. Los filtros de ruta pueden cambiar el destino de hacia dónde se dirige el servicio.

**3** Si un filtro de ruta quiere redirigir la llamada de servicio a un lugar distinto al que El servidor Zuul está configurado para enviar la ruta, puede hacerlo. Sin embargo, una ruta zuul El filtro no realiza una redirección HTTP , sino que finalizará la redirección entrante.

Solicitud HTTP y luego llame a la ruta en nombre de la persona que llama original. Este

significa que el filtro de ruta tiene que poseer completamente la llamada de la ruta dinámica y no puedo hacer una redirección HTTP .

- 4 Si el filtro de ruta no redirige dinámicamente a la persona que llama a una nueva ruta, el servidor Zuul enviará la ruta al servicio de destino original.
- 5 Después de que se haya invocado el servicio de destino, se invocarán los filtros de Zuul Post. A El filtro posterior puede inspeccionar y modificar la respuesta del servicio invocado.

La mejor manera de entender cómo implementar los filtros Zuul es verlos en uso. A Con este fin, en las siguientes secciones creará un filtro previo, de ruta y posterior y luego ejecutar solicitudes de clientes de servicio a través de ellos.

La Figura 6.12 muestra cómo estos filtros encararán al procesar solicitudes a su Servicios EagleEye.

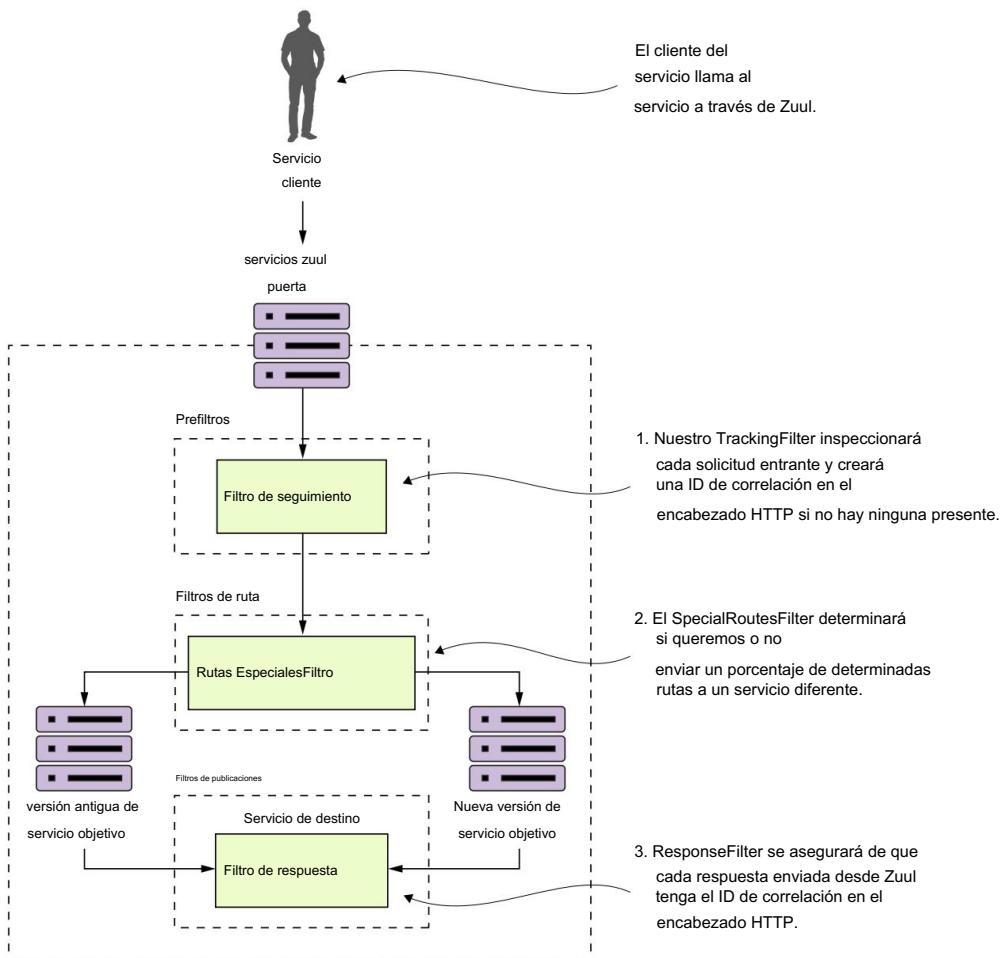


Figura 6.12 Los filtros Zuul proporcionan seguimiento centralizado de llamadas de servicio, registros y enrutamiento dinámico. Los filtros Zuul le permiten aplicar reglas y políticas personalizadas contra llamadas de microservicios.

Siguiendo el flujo de la figura 6.12, verá que se utilizan los siguientes filtros:

- 1 **TrackingFilter:** TrackingFilter será un filtro previo que garantizará que cada solicitud que fluye desde Zuul tiene un ID de correlación asociado. A El ID de correlación es un ID único que se transmite a través de todos los microservicios que se ejecutan al realizar una solicitud de cliente. Un ID de correlación le permite rastrear la cadena de eventos que ocurren cuando una llamada pasa por una serie de llamadas de microservicio.
- 2 **SpecialRoutesFilter:** SpecialRoutesFilter es un filtro de rutas de Zuul que verificará la ruta entrante y determinará si desea realizar pruebas A/B en la ruta. La prueba A/B es una técnica en la que un usuario (en este caso un servicio) es Se presentan aleatoriamente dos versiones diferentes de servicios que utilizan el mismo servicio. La idea detrás de las pruebas A/B es que las nuevas funciones se pueden probar antes se implementan para toda la base de usuarios. En nuestro ejemplo, tendrás dos versiones diferentes del mismo servicio de organización. Un pequeño número de usuarios será redirigido a la versión más reciente del servicio, mientras que la mayoría de los usuarios ser enrutado a la versión anterior del servicio.
- 3 **ResponseFilter:** ResponseFilter es un filtro posterior que inyectará el ID de correlación asociado con la llamada de servicio en el encabezado de respuesta HTTP siendo devuelto al cliente. De esta manera, el cliente tendrá acceso al ID de correlación asociado a la solicitud que realizó.

## 6.5 Construyendo su primera generación de prefiltro Zuul ID de correlación

Construir filtros en Zuul es una actividad extremadamente sencilla. Para empezar, construirás un Zuul prefiltro, llamado TrackingFilter, que inspeccionará todas las solicitudes entrantes al gateway y determine si hay un encabezado HTTP llamado tmx-correlation-id presente en la solicitud. El encabezado tmx-correlation-id contendrá un valor único GUID (Identificación global universal) que se puede utilizar para rastrear la solicitud de un usuario en múltiples microservicios.

**NOTA** Discutimos el concepto de ID de correlación en el capítulo 5. Aquí estamos Vamos a explicar con más detalle cómo usar Zuul para generar una identificación de correlación . Si te saltaste el libro, te recomiendo encarecidamente que leas capítulo 5 y lea la sección sobre el contexto de Hystrix y Thread. Su implementación de ID de correlación se implementará utilizando variables ThreadLocal y hay trabajo adicional por hacer para que las variables ThreadLocal funcionen con Hystrix.

Si tmx-correlation-id no está presente en el encabezado HTTP , su Zuul Track-ingFilter generará y establecerá el ID de correlación. Si ya existe un ID de correlación presente, Zuul no hará nada con el ID de correlación. La presencia de una correlación ID significa que esta llamada de servicio en particular es parte de una cadena de llamadas de servicio que realizan la petición del usuario. En este caso, su clase TrackingFilter no hará nada.

Sigamos adelante y observemos la implementación de TrackingFilter en la siguiente lista. Este código también se puede encontrar en los ejemplos de libros en zuulsvr/src/main/java/com/thinkmechanix/zuulsvr/filters/TrackingFilter.java.

**Listado 6.6 Prefiltro Zuul para generar ID de correlación**

```
paquete com.thinkmechanix.zuulsvr.filters;

import com.netflix.zuul.ZuulFilter;
import org.springframework.beans.factory.annotation.Autowired;

//Se eliminaron otras importaciones para mayor concisión
```

```
@Componente
la clase pública TrackingFilter extiende ZuulFilter {privado estático final int
    FILTER_ORDER = 1; booleano final estático privado
    SHOULD_FILTER=true;
    registrador final estático privado Logger =
    LoggerFactory.getLogger(TrackingFilter.class);
```

Todos los filtros Zuul deben extender la clase ZuulFilter y anular cuatro métodos:  
filterType(), filterOrder(),  
shouldFilter() y run().

```
@autocableado
FilterUtils filterUtils;
```

Las funciones comúnmente utilizadas que se utilizan en todos sus filtros se han encapsulado en la clase FilterUtils.

```
@Anular
public String filterType() { return
    FilterUtils.PRE_FILTER_TYPE;
}
```

El método filterType() se utiliza para indicarle a Zuul si el filtro es un filtro previo, de ruta o posterior.

```
@Anular
public int filterOrder() { return
    FILTER_ORDER;
}
```

El método filterOrder() devuelve un número entero Valor que indica en qué orden Zuul debe enviar las solicitudes a través de los diferentes tipos de filtros.

```
public boolean deberíaFilter() { return DEBE_FILTER;
}
```

El método deberíaFilter() devuelve un valor booleano que indica si No, el filtro debería estar activo.

```
booleano privado isCorrelationIdPresent(){ if
    (filterUtils.getCorrelationId()!=null){
        devolver verdadero;
    }
    falso retorno;
}
```

Los métodos auxiliares que realmente comprueba si el tmx-correlation-id es presente y también puede generar un ID de correlación valor GUID

```
cadena privada generarCorrelationId(){
    devolver java.util.UUID.randomUUID().toString();
}
```

El método run() es el código que se ejecuta cada vez que el servicio pasa por el filtrar. En su función ejecutar(), verifica si el tmx-correlation-id está presente y si no es así, genera un valor de correlación y establezca el tmx-id-de-correlación HTTP

```
ejecución pública de objetos() {si
    (isCorrelationIdPresent()) {
        logger.debug("tmx-correlation-id encontrado en el filtro de seguimiento: {}.
    ",
```

```
filterUtils.getCorrelationId());  
  
} demás{  
    filterUtils  
    .setCorrelationId(generarCorrelationId());  
  
    logger.debug("tmx-correlation-id generado en el filtro de  
seguimiento: {}.",  
    filterUtils.getCorrelationId());  
}  
  
SolicitudContexto ctx =  
    RequestContext.getCurrentContext();  
logger.debug("Procesando solicitud entrante para {}.",  
    ctx.getRequest().getRequestURI());  
devolver nulo;  
  
}}}
```

Para implementar un filtro en Zuul, debe extender la clase ZuulFilter y luego anular cuatro métodos: filterType(), filterOrder(), shouldFilter() y run(). Los primeros tres métodos de esta lista describen a Zuul qué tipo de filtro está creando, en qué orden debe ejecutarse en comparación con los otros filtros de su tipo y si debe estar activo. El último método, run(), contiene la lógica empresarial que implementará el filtro.

Ha implementado una clase llamada FilterUtils. Esta clase se utiliza para encapsular la funcionalidad común utilizada por todos sus filtros. La clase FilterUtils se encuentra en zuulsvr/src/main/java/com/thinkmechanix/zuulsvr/FilterUtils.java.

No vamos a recorrer toda la clase FilterUtils , pero los métodos clave que discutiremos aquí son las funciones getCorrelationId() y setCorrelationId() . La siguiente lista muestra el código del método getCorrelationId() de FilterUtils .

#### Listado 6.7 Recuperando el tmx-correlation-id de los encabezados HTTP

```
cadena pública getCorrelationId(){ RequestContext  
    ctx = RequestContext.getCurrentContext();  
  
    if  
        (ctx.getRequest() .getHeader(CORRELATION_ID) !=null)  
    { return  
        ctx.getRequest() .getHeader(CORRELATION_ID);  
  
    } demás{  
    devolver  
        ctx.getZuulRequestHeaders() .get(CORRELATION_ID);  
    }  
}
```

Lo clave a tener en cuenta en el listado 6.7 es que primero verifica si el tmx-correlation-ID ya está configurado en los encabezados HTTP para la solicitud entrante. Esto se hace mediante la llamada ctx.getRequest().getHeader(CORRELATION\_ID) .

**NOTA** En un servicio Spring MVC o Spring Boot normal, RequestContext sería de tipo org.springframework.web.servlet.support.RequestContext. Sin embargo, Zuul ofrece un RequestContext especializado que tiene varios métodos adicionales para acceder a valores específicos de Zuul. Este contexto de solicitud es parte del paquete com.netflix.zuul.context .

Si no está allí, consulte ZuulRequestHeaders . Zuul no te permite agregue o modifique directamente los encabezados de solicitud HTTP en una solicitud entrante. si sumamos el tmx-correlation-id y luego intente acceder a él nuevamente más tarde en el filtro, no será disponible como parte de la llamada ctx.getRequestHeader() . Para solucionar este problema, utiliza el método FilterUtils getCorrelationId() . Quizás recuerdes eso antes en el método run() de su clase TrackingFilter , hizo exactamente esto con el siguiente fragmento de código:

```
demás{
    filterUtils.setCorrelationId(generateCorrelationId());
    logger.debug("tmx-correlation-id generado en el filtro de seguimiento: {}", filterUtils.getCorrelationId());
}
```

La configuración de tmx-correlation-id se produce con la configuración FilterUtils-Método CorrelationId () :

```
setCorrelationId público vacío (Id. de correlación de cadena) {
    RequestContext ctx =
        RequestContext.getCurrentContext();
    ctx.addZuulRequestHeader (CORRELATION_ID, correlaciónId);
}
```

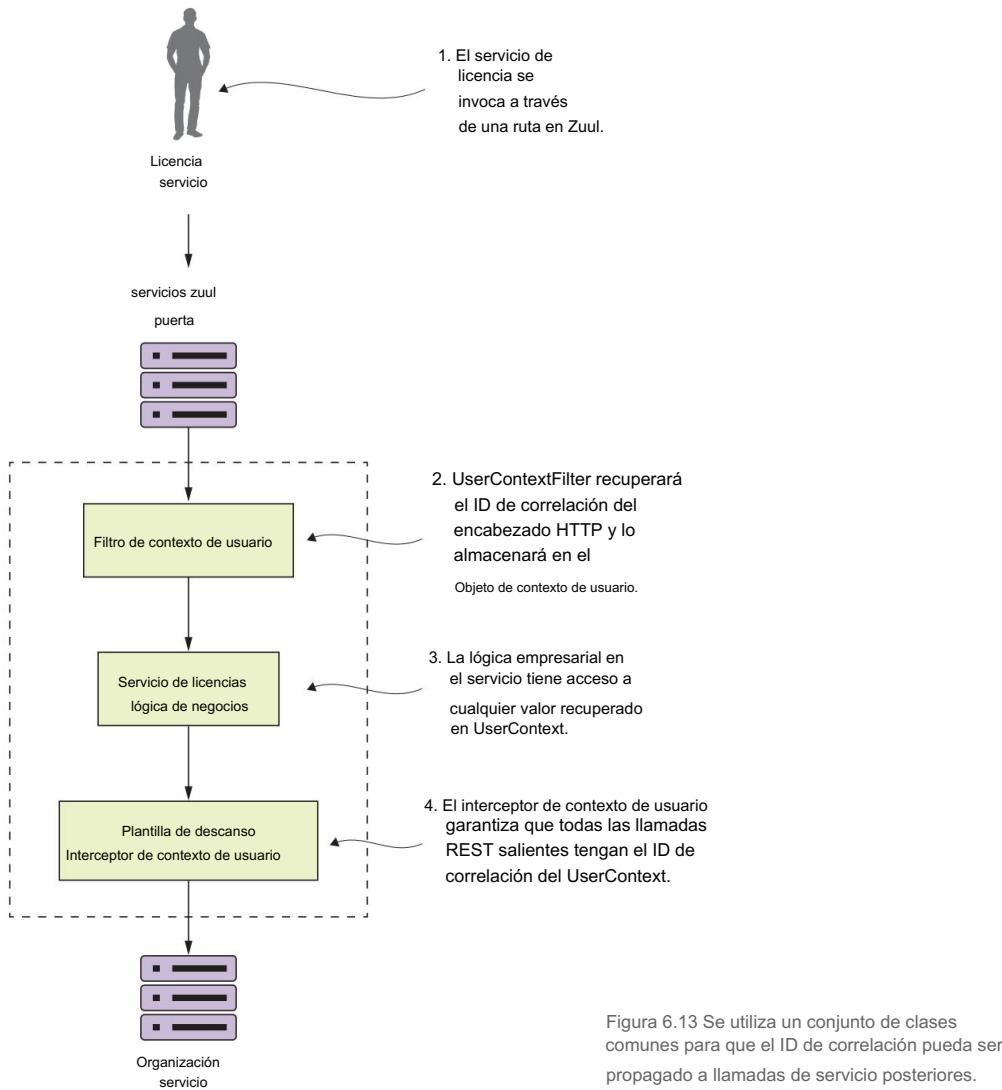
En el método FilterUtils setCorrelationId() , cuando desea agregar un valor al Encabezados de solicitud HTTP , utiliza addZuulRequestHeader() de RequestContext método. Este método mantendrá un mapa separado de los encabezados HTTP que se agregaron mientras una solicitud fluía a través de los filtros con su servidor Zuul. Los datos contenidos en el mapa ZuulRequestHeader se fusionarán cuando el servicio de destino sea invocado por su servidor Zuul.

### 6.5.1 Uso del ID de correlación en sus llamadas de servicio

Ahora que ha garantizado que se ha agregado una ID de correlación a cada llamada de microservicio que fluye a través de Zuul, ¿cómo puede asegurarse de que

El ID de correlación es fácilmente accesible para el microservicio que se está invocando. Cualquier llamada a servicio descendente que el microservicio pueda realizar también propaga el ID de correlación en la llamada descendente

Para implementar esto, creará un conjunto de tres clases en cada una de sus microservicios. Estas clases trabajarán juntas para leer el ID de correlación (junto con otra información que agregará más adelante) de la solicitud HTTP entrante , asignela a una clase que sea fácilmente accesible y utilizable por la lógica de negocios en la aplicación, y luego asegúrese de que el ID de correlación se propague a cualquier llamada de servicio descendente.



La Figura 6.13 muestra cómo se construirán estas diferentes piezas utilizando su servicio de licencias.

Repasemos lo que sucede en la figura 6.13:

- 1 Cuando se realiza una llamada al servicio de licencias a través del portal Zuul, el TrackingFilter inyectará un ID de correlación en el encabezado HTTP entrante para cualquier llamada que llegue a Zuul.
- 2 La clase UserContextFilter es un ServletFilter HTTP personalizado . Asigna un ID de correlación a la clase UserContext . La clase UserContext almacena valores en almacenamiento local de subprocesos para su uso más adelante en la llamada.

- 3 La lógica empresarial del servicio de licencias debe ejecutar una llamada a la organización. servicio.
- 4 Se utiliza un RestTemplate para invocar el servicio de la organización. El restoPlantilla utilizará una clase Spring Interceptor personalizada (UserContextInterceptor) para Inyecte el ID de correlación en la llamada saliente como un encabezado HTTP .

#### Código repetido versus bibliotecas compartidas

La cuestión de si debería utilizar bibliotecas comunes en todos sus microservicios Es un área gris en el diseño de microservicios. Los puristas de los microservicios le dirán que No debería utilizar un marco personalizado en todos sus servicios porque introduce dependencias artificiales en sus servicios. Se pueden introducir cambios en la lógica empresarial o un error. refactorización a gran escala de todos sus servicios. Por otro lado, otros profesionales de microservicios dirán que un enfoque purista no es práctico porque existen ciertas situaciones. (como el ejemplo anterior de UserContextFilter) donde tiene sentido construir un biblioteca común y compartirlo entre servicios.

Creo que aquí hay un término medio. Las bibliotecas comunes están bien cuando se trata de Tareas de estilo infraestructura. Si comienzas a compartir clases orientadas a los negocios, te estás metiendo en problemas porque estás rompiendo los límites entre los servicios.

Parece que estoy rompiendo mi propio consejo con los ejemplos de código de este capítulo, porque Si observa todos los servicios del capítulo, todos tienen sus propias copias del UserContextFilter, UserContext y UserContextInterceptor clases. La razón por la que adopté aquí un enfoque de no compartir nada es que no quiero complicar los ejemplos de código de este libro al tener que crear una biblioteca compartida que deben publicarse en un repositorio Maven de terceros. Por lo tanto, todas las clases en el El paquete de utilidades del servicio se comparte entre todos los servicios.

#### USERCONTEXTFILTER: INTERCEPTANDO LA SOLICITUD HTTP ENTRANTE

La primera clase que vas a construir es la clase UserContextFilter . Esta clase es una Filtro de servlet HTTP que interceptará todas las solicitudes HTTP entrantes que ingresan al servicio y asignará el ID de correlación (y algunos otros valores) de la solicitud HTTP a la clase UserContext . La siguiente lista muestra el código para UserContext clase. La fuente de esta clase se puede encontrar en licensing-service/src/main/java/com/thinkmechanix/licenses/utils/UserContextFilter.java.

#### Listado 6.8 Mapeo del ID de correlación a la clase UserContext

```
paquete com.thinkmechanixlicenses.utils;

//Eliminar las importaciones por razones de concisión
@Component
clase pública UserContextFilter implementa Filtro { registrador final estático
    privado registrador =
        LoggerFactory.getLogger(
            UserContextFilter.clase);
    @Anular
```

El filtro está registrado y seleccionado.  
en primavera mediante el uso de  
Anotación Spring @Component y por  
implementando una interfaz javax.servler.Filter.

```

doFilter público vacío (ServletRequest servletRequest,
    ServletResponse servletResponse,
    Cadena de filtros cadena de filtros)
    lanza IOException, ServletException {
    HttpServletRequest httpServletRequest = (HttpServletRequest) servletRequest;

    Titular del contexto de usuario
        .getContext()
        .setCorrelationId(
            httpServletRequest
                .getEncabezado(
                    UserContext.CORRELATION_ID));
    UserContextHolder.getContext().setUserId( httpServletRequest

        .getHeader(UserContext.USER_ID));
    Titular del contexto de usuario
        .getContext()
        .setAuthToken(
            httpServletRequest
                .getHeader(UserContext.AUTH_TOKEN));
    Titular del contexto de usuario
        .getContext()

        .setOrgId( httpServletRequest
            .getHeader(UserContext.ORG_ID));

    filtroCadena
        .doFilter(httpServletRequest, servletResponse);
}

// No se muestran los métodos init y destroy vacíos}

```

Tu filtro recupera la correlación.  
ID del encabezado y establece el valor en la clase UserContext.

Los otros valores que se extraen de los encabezados HTTP entrarán en juego si utiliza el ejemplo del servicio de autenticación definido en el archivo README del código.

En última instancia, UserContextFilter se utiliza para asignar los valores del encabezado HTTP que está interesado en una clase Java, UserContext.

**CONTEXTO DE USUARIO: HACER QUE LOS ENCABEZADOS HTTP SON FÁCILMENTE ACCESIBLES AL SERVICIO**

La clase UserContext se utiliza para contener los valores del encabezado HTTP para un servidor individual.

La solicitud del cliente vice está siendo procesada por su microservicio. Consta de un getter/setter. método que recupera y almacena valores de java.lang.ThreadLocal. La siguiente lista muestra el código de la clase UserContext . La fuente de esta clase. se puede encontrar en licensing-service/src/main/java/com/thinktmechanix/licencias/utils/UserContext.java.

#### Listado 6.9 Almacenamiento de los valores del encabezado HTTP dentro de la clase UserContext

```

@Componente
clase públicaContexto de usuario {
    Cadena final estática pública CORRELATION_ID = "tmx-correlation-id";
    Cadena final estática pública AUTH_TOKEN Cadena final = "tmx-token-autenticación";
    estática pública USER_ID Cadena final estática pública = "tmx-id-usuario";
    ORG_ID = "tmx-org-id";

```

```

ID de correlación de cadena privada = nueva cadena (); token de
autenticación de cadena privada = nueva cadena (); ID de
usuario de cadena privada = nueva cadena(); cadena privada
orgId = nueva cadena();

cadena pública getCorrelationId() { return correlaciónId; } public void setCorrelationId(String
correlaciónId) { this.correlationId = correlaciónId; }

public String getAuthToken() { return authToken; } public void setAuthToken(String
authToken) { this.authToken = authToken; }

public String getUserId() { return userId; } public void setUserId(String
userId) { this.userId = userId; }

public String getOrgId() { return orgId; } public void setOrgId(String
orgId) { this.orgId = orgId; }

}

```

Ahora la clase UserContext no es más que un POJO que contiene los valores extraídos de la solicitud HTTP entrante . Se utiliza una clase llamada zuulsvr/src/main/java/com/thinktmechanix/zuulsvr/filters/UserContextHolder.java para almacenar UserContext en una variable ThreadLocal a la que se puede acceder en cualquier método invocado por el subproceso que procesa la solicitud del usuario. El código para UserContextHolder se muestra en el siguiente listado.

#### Listado 6.10 UserContextHolder almacena el UserContext en un ThreadLocal

```

clase pública UserContextHolder { hilo final estático
privado<UserContext> userContext = new ThreadLocal<UserContext>();

público estático final UserContext getContext(){ Contexto de usuario
contexto = userContext.get();

if (contexto == nulo) { contexto =
createEmptyContext(); usuarioContext.set(contexto);

}

devolver userContext.get();
}

público estático final vacío setContext (contexto UserContext) {
Assert.notNull(context, "Sólo se
permiten instancias de UserContext no nulas");
usuarioContext.set(contexto);
}

UserContext final estático público createEmptyContext(){
devolver nuevo UserContext();
}
}

```

**RESTTEMPLATE PERSONALIZADO Y USERCONTEXTINTERCEPTOR: ASEGURANDO QUE EL ID DE CORRELACIÓN SE PROPAGA HACIA ADELANTE**

El último fragmento de código que veremos es UserContextInterceptor.

clase. Esta clase se utiliza para injectar el ID de correlación en cualquier solicitud de servicio saliente basada en HTTP que se ejecute desde una instancia de RestTemplate . Esto se hace para asegurar que pueda establecer un vínculo entre las llamadas de servicio.

Para hacer esto, usarás un Spring Interceptor que se inyecta en el

**Clase RestTemplate** . Veamos el UserContextInterceptor a continuación listado.

**Listado 6.11** A todas las llamadas de microservicio salientes se les inyecta el ID de correlación

```
paquete com.thinkmechanix.licenses.utils;
//Importaciones eliminadas para mayor concisión
clase pública UserContextInterceptor
    implementa ClientHttpRequestInterceptor {

        @Anular
        intercepción pública ClientHttpResponse (
            Solicitud HttpRequest, byte[] cuerpo,
            Ejecución de ClientHttpRequestExecution)
            lanza IOException {

            encabezados HttpHeaders = request.getHeaders();
            encabezados.add(
                UserContext.CORRELATION_ID,
                UserContextHolder
                    .getContext()
                    .getCorrelationId());
            encabezados.add(UserContext.AUTH_TOKEN,
                Titular del contexto de usuario
                    .getContext()
                    .getAuthToken());

            devolver ejecución.execute (solicitud, cuerpo);
        }
    }
```

Para utilizar UserContextInterceptor es necesario definir un bean RestTemplate y luego agregue el UserContextInterceptor . Para hacer esto, vas a agregar su propia definición de bean RestTemplate en licensing-service/src/main/ clase java/com/thinktmechanix/licenses/Application.java . La siguiente El listado muestra el método que se agrega a esta clase.

**Listado 6.12** Agregando el UserContextInterceptor a la clase RestTemplate

```
@carga equilibrada
@Frijol
plantilla de descanso pública getRestTemplate()
```

```

Plantilla RestTemplate = nueva RestTemplate(); Lista de
interceptores = template.getInterceptors();
    si (interceptores == nulo) {
        plantilla.setInterceptores(
            Colecciones.singletonList (nuevo
                UserContextInterceptor()));
    }
    else{ interceptores.add(new UserContextInterceptor());
        template.setInterceptores(interceptores);
    }
}

plantilla de devolución; }

Aregar UserContextInterceptor a la
instancia RestTemplate que se
ha creado

```

Con esta definición de bean implementada, cada vez que use la anotación @Autowired e inyecte un RestTemplate en una clase, usará el RestTemplate creado en el listado 6.11 con el UserContextInterceptor adjunto.

#### Agregación y autenticación de registros y más Ahora que

tiene ID de correlación que se pasan a cada servicio, es posible rastrear una transacción a medida que fluye a través de todos los servicios involucrados en la llamada. Para hacer esto, debe asegurarse de que cada servicio se registre en un punto central de agregación de registros que capture las entradas de registro de todos sus servicios en un solo punto. Cada entrada de registro capturada en el servicio de agregación de registros tendrá un ID de correlación asociado a cada entrada. La implementación de una solución de agregación de registros está fuera del alcance de este capítulo, pero en el capítulo 9 veremos cómo usar Spring Cloud Sleuth. Spring Cloud Sleuth no utilizará el TrackingFilter que creó aquí, pero utilizará los mismos conceptos de seguimiento del ID de correlación y garantizar que se inserte en cada llamada.

## 6.6 Creación de un filtro posterior que recibe ID de correlación

Recuerde, Zuul ejecuta la llamada HTTP real en nombre del cliente del servicio. Zuul tiene la oportunidad de inspeccionar la respuesta de la llamada de servicio de destino y luego modificar la respuesta o decorarla con información adicional. Cuando se combina con la captura de datos con el prefiltro, un filtro posterior de Zuul es una ubicación ideal para recopilar métricas y completar cualquier registro asociado con la transacción del usuario. Querrá aprovechar esto inyectando al usuario el ID de correlación que ha estado pasando a sus microservicios.

Hará esto utilizando un filtro de publicación de Zuul para inyectar el ID de correlación nuevamente en los encabezados de respuesta HTTP que se devuelven a la persona que llama del servicio. De esta manera, puede devolver el ID de correlación a la persona que llama sin tener que tocar el cuerpo del mensaje. La siguiente lista muestra el código para crear un filtro de publicaciones. Este código se puede encontrar en zuulsrc/src/main/java/com/thinktmechanix/zuulsrc/filters/ResponseFilter.java.

## Listado 6.13 Inyectar el ID de correlación en la respuesta HTTP

```

paquete com.thinkmechanix.zuulsvr.filters;

//Eliminar importaciones para mayor concisión

@Component
La clase pública ResponseFilter extiende ZuulFilter{
    privado estático final int FILTER_ORDER=1;
    booleano final estático privado SHOULD_FILTER=true;
    registrador final estático privado registrador =
    LoggerFactory
        .getLogger(ResponseFilter.clase);

    @autocableado
    FilterUtils filterUtils;

    @Anular
    cadena pública filterType() { return
        FilterUtils.POST_FILTER_TYPE;
    }

    @Anular
    orden de filtro pública int() {
        devolver FILTER_ORDER;
    }

    @Anular
    público booleano deberíaFilter() {
        devolver DEBE_FILTER;
    }

    @Anular
    Ejecución pública de objetos() {
        SolicitudContexto ctx =
            RequestContext.getCurrentContext();

        logger.debug("Agregar la identificación de correlación a los
encabezados salientes. {}", filterUtils.getCorrelationId());

        ctx.getResponse().addHeader(
            FilterUtils.CORRELATION_ID,
            filterUtils.getCorrelationId());
    }

    logger.debug("Completando la solicitud saliente para {}.",
    ctx.getRequest().getRequestURI());

    devolver nulo;
}
}

```

Para crear un filtro de publicaciones, debe configurar el tipo de filtro en POST\_FILTER\_TYPE.

Tome el ID de correlación que se pasó en la solicitud HTTP original e inyéctelo en la respuesta.

Registre el URI de solicitud saliente para tener "sujetalibros" que mostrarán las entradas y salidas entrada saliente de la solicitud del usuario en Zuul.

El ID de correlación devuelto en la respuesta HTTP.

Figura 6.14 El tmx-correlation-id se ha agregado a los encabezados de respuesta enviados al cliente del servicio.

Una vez que se haya implementado ResponseFilter , puede iniciar su servicio Zuul y llame al servicio de licencias de EagleEye a través de él. Una vez finalizado el servicio, Verá un tmx-correlation-id en el encabezado de respuesta HTTP de la llamada. La figura 6.14 muestra el tmx-correlation-id que se envía de vuelta desde la llamada.

Hasta este punto, todos nuestros ejemplos de filtros han tratado de manipular el servicio. llamadas del cliente antes y después de haber sido enrutadas a su destino objetivo. Para nuestro último ejemplo de filtro, veamos cómo puede cambiar dinámicamente la ruta de destino que desea enviar al usuario a.

## 6.7 Construyendo un filtro de ruta dinámico

El último filtro Zuul que veremos es el filtro de ruta Zuul. Sin un filtro de ruta personalizado en su lugar, Zuul hará todo su enrutamiento basándose en las definiciones de mapeo que vio anteriormente en el capítulo. Sin embargo, al crear un filtro de ruta Zuul, puede agregar inteligencia a cómo se enrutaría la invocación de un cliente de servicio.

En esta sección, aprenderá sobre el filtro de ruta de Zuul creando un filtro de ruta que Le permite realizar pruebas A/B de una nueva versión de un servicio. Las pruebas A/B son lo que necesitas crear una nueva característica y luego hacer que un porcentaje de la población total de usuarios utilice esa característica. El resto de la población de usuarios todavía utiliza el antiguo servicio. En este ejemplo, estás vamos a simular la implementación de una nueva versión del servicio de organización donde desee El 50% de los usuarios van al servicio antiguo y el 50% de los usuarios van al servicio nuevo.

Para hacer esto, creará un filtro de ruta Zuul, llamado SpecialRoutes-Filter, que tomará el ID del servicio Eureka del servicio al que llama Zuul y

llamar a otro microservicio llamado SpecialRoutes. El servicio SpecialRoutes comprobará una base de datos interna para ver si el nombre del servicio existe. Si el nombre del servicio de destino existe, devolverá un peso y un destino de destino de una alternativa ubicación para el servicio. El SpecialRoutesFilter tomará entonces el peso devuelto y, según el peso, generar aleatoriamente un número que se utilizará para determinar si la llamada del usuario se enrutaría al servicio de organización alternativo o al servicio de la organización definido en los mapeos de rutas de Zuul. La figura 6.15 muestra el flujo de lo que sucede cuando se utiliza SpecialRoutesFilter .

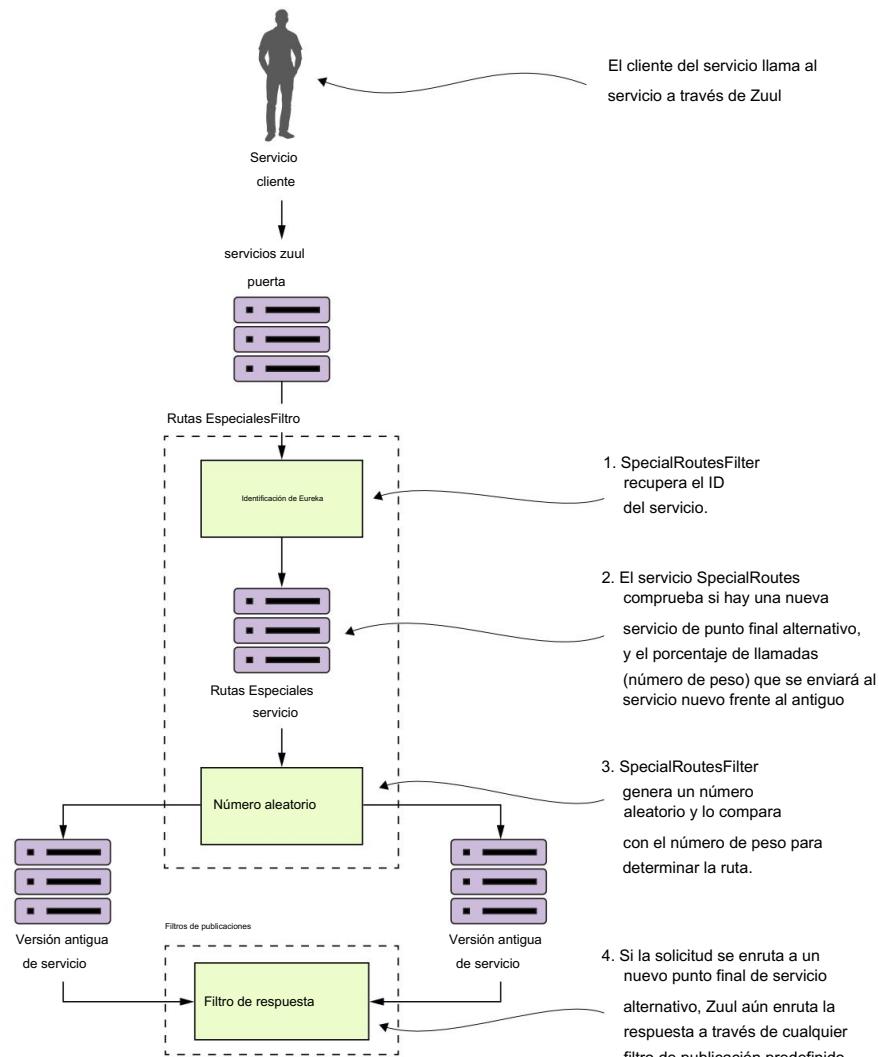


Figura 6.15 El flujo de una llamada al servicio de la organización a través del SpecialRoutesFilter

En la figura 6.15, después de que el cliente del servicio ha llamado a un servicio "enfrentado" por Zuul, SpecialRoutesFilter realiza las siguientes acciones:

- 1 SpecialRoutesFilter recupera el ID del servicio que se está llamado.
- 2 SpecialRoutesFilter llama al servicio SpecialRoutes . El servicio de rutas especiales comprueba si hay un punto final alternativo definido para el punto final de destino. Si se encuentra un registro, contendrá un peso que le indicará a Zuul el porcentaje de llamadas de servicio que deben enviarse al servicio antiguo y al nuevo.
- 3 Luego , SpecialRoutesFilter genera un número aleatorio y lo compara con el peso devuelto por el servicio SpecialRoutes . Si el número generado aleatoriamente está por debajo del valor del peso del punto final alternativo, SpecialRoutesFilter envía la solicitud a la nueva versión del servicio.
- 4 Si SpecialRoutesFilter envía la solicitud a una nueva versión del servicio, Zuul mantiene las canalizaciones predefinidas originales y envía la respuesta desde el punto final del servicio alternativo a través de cualquier filtro de publicación definido.

#### 6.7.1 Construyendo el esqueleto del filtro de enrutamiento

Comenzaremos a analizar el código que utilizó para crear el filtro de rutas especiales. De todos los filtros que hemos visto hasta ahora, implementar un filtro de ruta Zuul requiere el mayor esfuerzo de codificación, porque con un filtro de ruta estás asumiendo una pieza central de la funcionalidad de Zuul, enrutando y reemplazándola con la tuya propia. funcionalidad.

No vamos a repasar toda la clase en detalle aquí, sino que analizaremos los detalles pertinentes.

El SpecialRoutesFilter sigue el mismo patrón básico que los otros filtros Zuul. Extiende la clase ZuulFilter y establece el método filterType() para devolver el valor de "ruta". No voy a entrar en más explicaciones sobre los métodos filter-Order() y deberíasFilter() ya que no son diferentes de los filtros anteriores discutidos anteriormente en este capítulo. La siguiente lista muestra el esqueleto del filtro de ruta.

##### Listado 6.14 El esqueleto de su filtro de ruta

```
paquete com.thinkmechanix.zuulsvr.filters;

@Component
clase pública SpecialRoutesFilter extiende ZuulFilter {
    @Anular
    public String filterType() { return
        filterUtils.ROUTE_FILTER_TYPE;
    }

    @Anular
    orden de filtro pública int() {
```

```

@Anular
booleano público deberíaFilter() {}

@Anular
ejecución de objeto público()
{} }

```

### 6.7.2 Implementación del método run()

El verdadero trabajo para SpecialRoutesFilter comienza en el método run() del código. La siguiente lista muestra el código de este método.

Listado 6.15 El método run() para SpecialRoutesFilter es donde comienza el trabajo

```

Ejecución pública de objetos() {
    RequestContext ctx = RequestContext.getCurrentContext();

    AbTestingRoute abTestRoute =
        getAbRoutingInfo( filterUtils.getServiceId() );

    if (abTestRoute!=null &&
        useSpecialRoute(abTestRoute)) { Ruta de
        cadena =
            buildRouteString(
                ctx.getRequest().getRequestURI(),
                abTestRoute.getEndpoint(),
                ctx.get("servic eid").toString());
        forwardToSpecialRoute(ruta);
    }

    devolver nulo;
}

```

Ejecución pública de objetos() {  
RequestContext ctx = RequestContext.getCurrentContext();  
  
AbTestingRoute abTestRoute =  
getAbRoutingInfo( filterUtils.getServiceId() );  
  
if (abTestRoute!=null &&  
useSpecialRoute(abTestRoute)) { Ruta de  
cadena =  
buildRouteString(  
ctx.getRequest().getRequestURI(),  
abTestRoute.getEndpoint(),  
ctx.get("servic eid").toString());  
forwardToSpecialRoute(ruta); }  
  
devolver nulo;  
}  
}

El método useSpecialRoute()  
tomará el peso de la ruta,  
generar un número aleatorio,  
y determinar si vas  
para remitir la solicitud a  
el servicio alternativo.

Si hay un registro de enrutamiento, cree la URL  
completa (con la ruta) a la ubicación del servicio.  
especificado por el servicio de rutas especiales.

El flujo general de código en el listado 6.15 es que cuando una solicitud de ruta llega al comando run() método en SpecialRoutesFilter, ejecutará una llamada REST al servicio Special-Routes . Este servicio ejecutará una búsqueda y determinará si un registro de ruta existe para el ID del servicio Eureka del servicio de destino al que se llama. El llamado a El servicio SpecialRoutes se realiza en el método getAbRoutingInfo() . El método get-AbRoutingInfo() se muestra en el siguiente listado.

Listado 6.16 Invocar el servicio SpecialRoute para ver si existe un registro de enrutamiento

```

Private AbTestingRoute getAbRoutingInfo (String serviceName) {
    ResponseEntity<AbTestingRoute> restExchange = null;
    intentar {
        restExchange = restTemplate.exchange( "http://servicio
        de rutas especiales/v1
        /ruta/abtesting/{nombresservicio}",
        HttpMethod.GET,null, AbTestingRoute.class, serviceName);
    }
}

```

Private AbTestingRoute getAbRoutingInfo (String serviceName) {  
 ResponseEntity<AbTestingRoute> restExchange = null;  
 intentar {  
 restExchange = restTemplate.exchange( "http://servicio  
 de rutas especiales/v1  
 /ruta/abtesting/{nombresservicio}",  
 HttpMethod.GET,null, AbTestingRoute.class, serviceName);  
 }  
}

Llama al punto final  
SpecialRoutesService

```

catch(HttpClientErrorException ex){ si (ex.getStatusCode() ==
    HttpStatus.NOT_FOUND){
        devolver nulo;
        tirar ex;
    }
    devolver restExchange.getBody(); }

```

Si los servicios de rutas no encuentran un registro (devolverá un código de estado HTTP 404), el método devolverá nulo.

Una vez que haya determinado que hay un registro de enrutamiento presente para el servicio de destino, debe determinar si debe enrutar la solicitud de servicio de destino al ubicación de servicio alternativa o a la ubicación de servicio predeterminada administrada estáticamente por el Mapas de ruta de Zuul. Para tomar esta determinación, llame a `useSpecialRoute()` método. La siguiente lista muestra este método.

#### Listado 6.17 Determinar si se utiliza la ruta de servicio alternativa

```

uso booleano públicoSpecialRoute(AbTestingRoute testRoute){
    Aleatorio aleatorio = nuevo Aleatorio();

    si (testRoute.getActive().equals("N"))
        falso retorno;

    valor entero =
        aleatorio.siguienteInt((10 - 1) + 1) + 1;

    si (testRoute.getWeight()<valor) devuelve verdadero;

    falso retorno;
}

```

Comprueba si la ruta está activa

Determina si debe utilizar la ruta de servicio alternativa

Este método hace dos cosas. Primero, el método verifica el campo activo en la Registro `AbTestingRoute` devuelto por el servicio `SpecialRoutes`. Si el registro es establecido en "N", el método `useSpecialRoute()` no debería hacer nada porque no Quiero hacer alguna ruta en este momento. En segundo lugar, el método genera un número aleatorio entre 1 y 10. Luego, el método comprobará si el peso de la devolución La ruta es menor que el número generado aleatoriamente. Si la condición es verdadera, el método `useSpecialRoute` devuelve verdadero , lo que indica que desea utilizar la ruta.

Una vez que haya determinado que desea enrutar la solicitud de servicio que llega en `SpecialRoutesFilter`, reenviará la solicitud al destino servicio.

### 6.7.3 Reenviar la ruta

El reenvío real de la ruta al servicio descendente es donde la mayoría de el trabajo se produce en `SpecialRoutesFilter`. Si bien Zuul proporciona funciones de ayuda para facilitar esta tarea, la mayor parte del trabajo aún recae en el desarrollador. El método `forwardToSpecialRoute()` hace el trabajo de reenvío por usted. El Código de este método se basa en gran medida en el código fuente de Spring Cloud.

**Clase SimpleHostRoutingFilter**. Si bien no vamos a repasar todos los funciones auxiliares llamadas en el método `forwardToSpecialRoute()`, caminaremos a través del código en este método, como se muestra en el siguiente listado.

Listado 6.18 `forwardToSpecialRoute` invoca el servicio alternativo

```

ayudante privado de ProxyRequestHelper = nuevo
    ProxyRequestHelper ();

vacío privado forwardToSpecialRoute (ruta de cadena) {
    Contexto RequestContext =
        RequestContext.getCurrentContext ();
    Solicitud HttpServletRequest = contexto.getRequest ();

    MultiValueMap<String, String> headers =
        helper.buildZuulRequestHeaders(solicitud);

    MultiValueMap<String, String> params =
        helper.buildZuulRequestqueryParams(solicitud);

    Verbo de cadena = getVerb(solicitud); InputStream
    requestEntity = getRequestBody(solicitud); si (request.getContentLength () < 0)
    contexto.setChunkedRequestBody();

    this.helper.addIgnoredHeaders();
    CloseableHttpClient httpClient = nulo;
    Respuesta HttpResponse = nula;

    intentar {
        httpClient = HttpClient.createDefault();
        respuesta = reenviar( httpClient,
            verbo,
            ruta,
            solicitud,
            encabezados,
            parámetros,
            requestEntity);
        setResponse(respuesta);
    }
    catch (Exception ex) { // Eliminado por razones de concisión}
}

```

La variable auxiliar es una variable de instancia de tipo clase `ProxyRequestHelper`. Esta es una clase de Spring Cloud con funciones auxiliares para solicitudes de servicio de proxy.

Crea una copia de todos los encabezados de solicitud HTTP que se enviarán al servicio.

Crea una copia de todos los Parámetros de solicitud HTTP

Hace una copia del Cuerpo HTTP que se reenviará al servicio alternativo

Invoca el servicio alternativo utilizando el asistente de reenvío. método (no mostrado)

El resultado de la llamada de servicio se guarda en el servidor Zuul. a través del método auxiliar `setResponse()`.

La conclusión clave del código del listado 6.18 es que estás copiando todos los valores de la solicitud HTTP entrante (los parámetros del encabezado, el verbo HTTP y el cuerpo) en una nueva solicitud que se invocará en el servicio de destino. El método `forwardToSpecialRoute()` luego recupera la respuesta del servicio de destino y la establece en el Contexto de solicitud HTTP utilizado por Zuul. Esto se hace a través del asistente `setResponse()` método (no mostrado). Zuul usa el contexto de solicitud HTTP para devolver la respuesta del cliente del servicio llamadas.

## 6.7.4 Juntándolo todo

Ahora que ha implementado SpecialRoutesFilter, puede verlo como acción llamando al servicio de licencias. Como recordará de capítulos anteriores, el servicio de licencias llama al servicio de la organización para recuperar los datos de contacto de la organización.

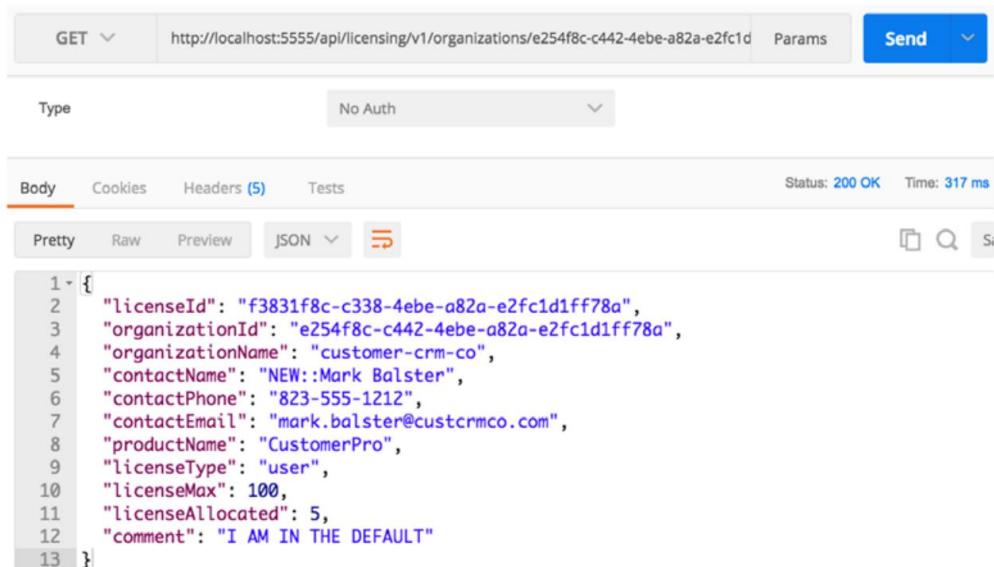
En el ejemplo de código, el servicio de rutas especiales tiene un registro de base de datos para el servicio de la organización que enrutará las solicitudes de llamadas al servicio de la organización el 50 % del tiempo al servicio de la organización existente (el asignado en Zuul) y el 50 % del tiempo al servicio de la organización. un servicio de organización alternativo. La ruta alternativa del servicio de organización devuelta por el servicio SpecialRoutes será `http://orgservice-new` y no será accesible directamente desde Zuul. Para diferenciar entre los dos servicios, modifiqué los servicios de la organización para anteponer el texto “ANTIGUO::” y “NUEVO::” a los nombres de contacto devueltos por el servicio de la organización.

Si ahora accede al punto final del servicio de licencias a través de Zuul

```
http://localhost:5555/api/licensing/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a
```

Debería ver el nombre de contacto devuelto por la llamada del servicio de licencias alternando entre los valores ANTIGUO:: y NUEVO:: . La figura 6.16 muestra esto.

Un filtro de rutas de Zuul requiere más esfuerzo para implementar que un filtro previo o posterior, pero también es una de las partes más poderosas de Zuul porque puede agregar fácilmente inteligencia a la forma en que se enrutan sus servicios.



The screenshot shows a Postman request configuration and a JSON response body. The request method is GET, the URL is `http://localhost:5555/api/licensing/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a`, and the status is 200 OK. The response body is a JSON object with the following content:

```
1 {  
2   "licenseId": "f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a",  
3   "organizationId": "e254f8c-c442-4ebe-a82a-e2fc1d1ff78a",  
4   "organizationName": "customer-crm-co",  
5   "contactName": "NEW::Mark Balster",  
6   "contactPhone": "823-555-1212",  
7   "contactEmail": "mark.balster@custcrmco.com",  
8   "productName": "CustomerPro",  
9   "licenseType": "user",  
10  "licenseMax": 100,  
11  "licenseAllocated": 5,  
12  "comment": "I AM IN THE DEFAULT"  
13 }
```

Figura 6.16 Cuando accede al servicio de organización alternativo, verá NUEVO antepuesto al nombre del contacto.

## 6.8 Resumen

Spring Cloud hace que sea trivial construir una puerta de enlace de servicios. La puerta de enlace de servicios Zuul se integra con el servidor Eureka de Netflix y puede automapear automáticamente los servicios registrados con Eureka a una ruta Zuul. Zuul puede prefijar todas las rutas que se administran, por lo que puede prefijar fácilmente sus rutas con algo como /api. Con Zuul, puede definir manualmente asignaciones de rutas. Estas asignaciones de rutas se definen manualmente en los archivos de configuración de las aplicaciones. Al utilizar el servidor Spring Cloud Config, puede recargar dinámicamente las asignaciones de rutas sin tener que reiniciar el servidor Zuul. Puede personalizar los tiempos de espera de Hystrix y Ribbon de Zuul a nivel global e individual niveles de servicio. Zuul le permite implementar lógica empresarial personalizada a través de filtros Zuul. Zuul tiene tres tipos de filtros: filtros Zuul previos, posteriores y de enruteamiento. Los prefiltros de Zuul se pueden utilizar para generar una ID de correlación que se puede inyectar en cada servicio que fluye a través de Zuul.

Un filtro de publicaciones de Zuul puede inyectar una ID de correlación en cada respuesta de servicio HTTP a un cliente de servicio.

Un filtro de ruta Zuul personalizado puede realizar enruteamiento dinámico basado en un ID de servicio Eureka para realizar pruebas A/B entre diferentes versiones del mismo servicio.



# Proteger sus microservicios

## Este capítulo cubre

Aprender por qué la seguridad es importante en un entorno de microservicios   Comprender el estándar OAuth2   Instalar y configurar un servicio OAuth2 basado en Spring   Realizar autenticación y autorización de usuarios con OAuth2  
Proteger su microservicio Spring usando OAuth2  
Propagar su token de acceso OAuth2 entre servicios

Seguridad. La mención de la palabra a menudo provocará un gemido involuntario por parte del desarrollador que la escuche. Los oirás murmurar y maldecir en voz baja: "Es obtuso, difícil de entender y aún más difícil de depurar". Sin embargo, no encontrará ningún desarrollador (excepto quizás desarrolladores sin experiencia) que diga que no se preocupa por la seguridad.

Una aplicación segura implica múltiples capas de protección, incluyendo   Garantizar que se implementen los controles de usuario adecuados para que pueda validar que un usuario es quien dice ser y que tiene permiso para hacer lo que está intentando hacer.

Mantener la infraestructura en la que se ejecuta el servicio parcheada y actualizada para minimizar el riesgo de vulnerabilidades.

Implementar controles de acceso a la red para que un servicio sólo sea accesible a través  
Puertos bien definidos y accesibles a un pequeño número de servidores autorizados.

Este capítulo solo abordará el primer punto de esta lista: cómo autenticar que el usuario que llama a su  
microservicio es quien dice ser y determinar  
si están autorizados a realizar la acción que solicitan a su  
microservicio. Los otros dos temas son temas de seguridad extremadamente amplios que están fuera  
el alcance de este libro.

Para implementar controles de autorización y autenticación, utilizará  
Seguridad de Spring Cloud y el estándar OAuth2 (Autenticación Abierta) para proteger  
sus servicios basados en Spring. OAuth2 es un marco de seguridad basado en tokens que permite  
usuario autenticarse con un servicio de autenticación de terceros. Si el usuario  
se autentica exitosamente, se les presentará un token que debe enviarse con cada  
pedido. Luego, el token se puede volver a validar en el servicio de autenticación. El principal  
El objetivo detrás de OAuth2 es que cuando se llaman múltiples servicios para cumplir con la solicitud de un usuario,  
el usuario puede ser autenticado por cada servicio sin tener que presentar sus credenciales a cada  
servicio que procese su solicitud. Spring Boot y Spring Cloud proporcionan cada uno una implementación  
lista para usar de un servicio OAuth2 y lo hacen extremadamente  
Fácil de integrar la seguridad OAuth2 en su servicio.

**NOTA** En este capítulo, le mostraremos cómo proteger sus microservicios utilizando  
OAuth2; sin embargo, una implementación completa de OAuth2 también requiere una  
aplicación web frontal para ingresar sus credenciales de usuario. No pasaremos por  
cómo configurar la aplicación front-end porque eso está fuera del alcance de un libro  
sobre microservicios. En su lugar, usaremos un cliente REST , como POSTMAN, para simular  
la presentación de credenciales. Para un buen tutorial sobre cómo configurar su  
aplicación front-end, te recomiendo que consultes el siguiente tutorial de Spring:  
<https://spring.io/blog/2015/02/03/sso-with-oauth2-angular-js-and-spring-security-part-v> .

El verdadero poder detrás de OAuth2 es que permite a los desarrolladores de aplicaciones integrarse  
fácilmente con proveedores de nube externos y realizar autenticación y autorización de usuarios.  
con esos servicios sin tener que pasar constantemente las credenciales del usuario al servicio de  
terceros. Los proveedores de nube como Facebook, GitHub y Salesforce son compatibles  
OAuth2 como estándar.

Antes de entrar en los detalles técnicos de la protección de nuestros servicios con OAuth2, veamos  
recorra la arquitectura OAuth2 .

## 7.1 Introducción a OAuth2

OAuth2 es un marco de autorización y autenticación de seguridad basado en tokens que  
divide la seguridad en cuatro componentes. Estos cuatro componentes son

- 1 Un recurso protegido: este es el recurso (en nuestro caso, un microservicio) que desea  
para proteger y garantizar que sólo puedan acceder los usuarios autenticados que tengan la  
autorización adecuada.

- 2 Un propietario de recurso: un propietario de recurso define qué aplicaciones pueden llamar a su servicio, qué usuarios pueden acceder al servicio y qué pueden hacer con él. Cada aplicación registrada por el propietario del recurso recibirá un nombre de la aplicación que identifica la aplicación junto con una aplicación clave secreta. La combinación del nombre de la aplicación y la clave secreta son parte de las credenciales que se pasan al autenticar un token OAuth2.
- 3 Una aplicación: esta es la aplicación que llamará al servicio en un nombre de un usuario. Después de todo, los usuarios rara vez invocan un servicio directamente. En cambio, confían en una solicitud para hacer el trabajo por ellos.
- 4 Servidor de autenticación OAuth2: el servidor de autenticación OAuth2 es el intermediario entre la aplicación y los servicios que se consumen. El OAuth2 permite al usuario autenticarse sin tener que pasar sus credenciales de usuario para cada servicio que la aplicación llamará en su nombre.

Los cuatro componentes interactúan entre sí para autenticar al usuario. El usuario sólo tiene que presentar sus credenciales. Si se autentican exitosamente, se les emite un token de autenticación que puede pasar de un servicio a otro. Esto se muestra en la figura 7.1.

OAuth2 es un marco de seguridad basado en tokens. Un usuario se autentica contra OAuth2 servidor proporcionando sus credenciales junto con la aplicación que están utilizando para acceder al recurso. Si las credenciales del usuario son válidas, el servidor OAuth2 proporciona una

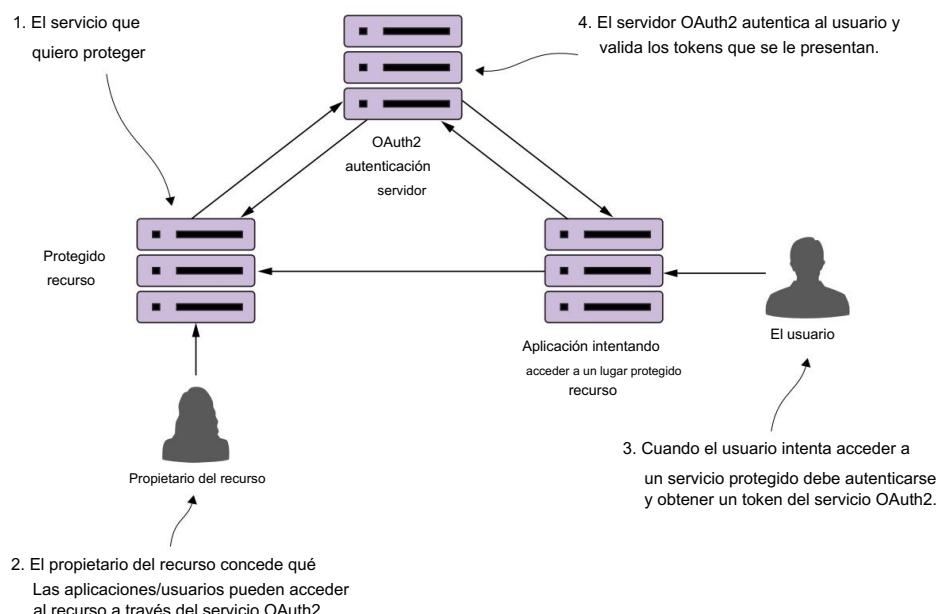


Figura 7.1 OAuth2 permite a un usuario autenticarse sin tener que presentar credenciales constantemente.

token que se puede presentar cada vez que un servicio utilizado por la aplicación del usuario intenta acceder a un recurso protegido (el microservicio).

Luego, el recurso protegido puede comunicarse con el servidor OAuth2 para determinar la validez del token y recuperar qué roles le ha asignado un usuario. Los roles se utilizan para agrupar usuarios relacionados y definir a qué recursos puede acceder ese grupo de usuarios . Para los propósitos de este capítulo, utilizará OAuth2 y roles para definir qué puntos finales de servicio y qué verbos HTTP puede llamar un usuario en un punto final.

La seguridad de los servicios web es un tema extremadamente complicado. Debe comprender quién llamará a sus servicios (usuarios internos de su red corporativa, usuarios externos), cómo llamarán a su servicio (cliente interno basado en web, dispositivo móvil, aplicación web fuera de su red corporativa) y qué acciones van a tomar con su código. OAuth2 le permite proteger sus servicios basados en REST en estos diferentes escenarios mediante diferentes esquemas de autenticación llamados subvenciones. La especificación OAuth2 tiene cuatro tipos de concesiones:

- Contraseña
- Credencial de cliente
- Código de autorización
- Implícito

No analizaremos cada uno de estos tipos de concesión ni proporcionaremos ejemplos de código para cada tipo de concesión. Eso es simplemente demasiado material para cubrirlo en un solo capítulo. En lugar de eso, haré lo siguiente:

Analice cómo su servicio de microservicios puede utilizar OAuth2 a través de uno de los tipos de concesión de OAuth2 más simples (el tipo de concesión de contraseña). Utilice tokens web de JavaScript para proporcionar una solución OAuth2 más sólida y establecer un estándar para codificar información en un token OAuth2 . Analice otras consideraciones de seguridad que deben tenerse en cuenta al crear microservicios.

Proporciono material general sobre los otros tipos de concesiones de OAuth2 en el apéndice B, " Tipos de concesiones de OAuth2". Si está interesado en profundizar en más detalles sobre la especificación OAuth2 y cómo implementar todos los tipos de concesiones, le recomiendo el libro de Justin Richer y Antonio Sanso, OAuth2 in Action (Manning, 2017), que es una explicación completa de OAuth2.

## 7.2 Comenzando poco a poco: usar Spring y OAuth2 para proteger un único punto final

Para comprender cómo

configurar las piezas de autenticación y autorización de OAuth2, implementará el tipo de concesión de contraseña de OAuth2 . Para implementar esta subvención, hará lo siguiente:

Configure un servicio de autenticación OAuth2 basado en Spring-Cloud .  
Registre una aplicación EagleEye UI falsa como una aplicación autorizada que puede autenticar y autorizar identidades de usuario con su servicio OAuth2 .

Utilice la concesión de contraseña OAuth2 para proteger sus servicios EagleEye. No creará una interfaz de usuario para EagleEye, por lo que simulará un usuario que inicia sesión para usar POSTMAN para autenticarse en su servicio EagleEye OAuth2 .

Proteger el servicio de licencias y organización para que solo puedan ser llamados por un usuario autenticado.

### 7.2.1 Configuración del servicio de autenticación EagleEye OAuth2

Como todos los ejemplos de los capítulos de este libro, su servicio de autenticación OAuth2 será otro servicio Spring Boot. El servicio de autenticación autenticará las credenciales del usuario y emitirá un token. Cada vez que el usuario intenta acceder a un servicio protegido por el servicio de autenticación, el servicio de autenticación validará que el token OAuth2 fue emitido por él y que no ha caducado. El servicio de autenticación será el equivalente al servicio de autenticación de la figura 7.1.

Para comenzar, configurará dos cosas:

- 1 Las dependencias de compilación de Maven apropiadas necesarias para su clase de arranque
- 2 Una clase de arranque que actuará como punto de entrada al servicio

Puede encontrar todos los ejemplos de código para el servicio de autenticación en el directorio del servicio de autenticación. Para configurar un servidor de autenticación OAuth2 , necesita las siguientes dependencias de Spring Cloud en el archivo authentication-service/pom.xml:

```
<dependencia>
    <groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-security</artifactId> </dependencia>

<dependencia>
    <groupId>org.springframework.security.oauth</groupId> <artifactId>spring-security-oauth2</artifactId>
</dependencia>
```

La primera dependencia, spring-cloud-security, incluye las bibliotecas de seguridad generales Spring y Spring Cloud. La segunda dependencia, spring-security-oauth2, extrae las bibliotecas Spring OAuth2 .

Ahora que las dependencias de Maven están definidas, puede trabajar en la clase de arranque. Esta clase se puede encontrar en la clase authentication-service/src/main/java/com/thinkmechanix/authentication/Application.java . La siguiente lista muestra el código de la clase Application.java .

#### Listado 7.1 La clase de arranque del servicio de autenticación

```
//Importaciones eliminadas por razones de concisión

@SpringBootApplication
@RestController
@EnableResourceServer
@EnableAuthorizationServer
Aplicación de clase pública {
```

Se utiliza para decirle a Spring Cloud que este servicio actuará como un servicio OAuth2.

```

@RequestMapping(valor = { "/usuario" }, produce = "aplicación/json")
Mapa público <Cadena, Objeto> usuario (usuario OAuth2Authentication) {
    Mapa<Cadena, Objeto> userInfo = new HashMap<>();
    información de usuario.put(
        "usuario",
        usuario.getUserAuthentication()
        .getPrincipal());
    información de usuario.put(
        "autoridades",
        AuthorityUtils.authorityListToSet(
            usuario.getUserAuthentication()
            .getAuthorities())));
    devolver información de usuario;
}

público estático vacío principal (String [] argumentos) {
    SpringApplication.run(Aplicación.clase, argumentos);
}
}

```

Usado más adelante en el capítulo.  
para recuperar información  
sobre el usuario

Lo primero que hay que tener en cuenta en este listado es la anotación `@EnableAuthorizationServer`. Esta anotación le dice a Spring Cloud que este servicio se usará como un servicio OAuth2 y que agregará varios puntos finales basados en REST que se usarán en OAuth2. procesos de autenticación y autorización.

La segunda cosa que verá en el listado 7.1 es la adición de un punto final llamado `/usuario` (que se asigna a `/auth/usuario`). Utilizará este punto final más adelante en el capítulo. cuando intentas acceder a un servicio protegido por OAuth2. Este punto final es llamado por el servicio protegido para validar el token de acceso OAuth2 y recuperar el asignado roles del usuario que accede al servicio protegido. Discutiré este punto final en mayor detalle. detalle más adelante en el capítulo.

### 7.2.2 Registro de aplicaciones cliente con el servicio OAuth2

En este punto tienes un servicio de autenticación, pero no has definido ninguna aplicación, usuarios o roles dentro del servidor de autenticación. Puede comenzar registrando la aplicación Eagle-Eye en su servicio de autenticación. Para hacer esto, vas a configurar un clase adicional en su servicio de autenticación llamada servicio de autenticación/`src/main/java/com/thinkmechanix/authentication/security/OAuth2Config.java`.

Esta clase definirá qué aplicaciones están registradas con su servicio de autenticación OAuth2. Es importante tener en cuenta que sólo porque una aplicación esté registrada con su servicio OAuth2, no significa que el servicio tendrá acceso a cualquier contenido protegido recursos.

#### Sobre autenticación versus autorización

A menudo he descubierto que los desarrolladores “mezclan y combinan” el significado de los términos autenticación y autorización. La autenticación es el acto por el que un usuario demuestra quién es proporcionando credenciales. La autorización determina si un usuario puede hacer lo que

(continuado)

están intentando hacer. Por ejemplo, el usuario Jim podría probar su identidad proporcionando un ID de usuario y contraseña, pero es posible que no esté autorizado a ver datos confidenciales como como datos de nómina. Para los propósitos de nuestra discusión, un usuario debe estar autenticado antes de que se produzca la autorización.

La clase OAuth2Config define qué aplicaciones y las credenciales de usuario El servicio OAuth2 lo conoce. En el siguiente listado puedes ver el Código OAuth2Config.java .

Listado 7.2 El servicio OAuth2Config define qué aplicaciones pueden usar su servicio

```
//Importaciones eliminadas por razones de concisión
@Configuration clase
pública OAuth2Config extiende AuthorizationServerConfigurerAdapter {

    @autocableado
    administrador de autenticación privado administrador de autenticación;
    @autocableado
    servicio de detalles de usuario privado servicio de detalles de usuario;

    @Override
    public void configure (clientes ClientDetailsServiceConfigurer) lanza
        Excepción {
        clientes.inMemory() .withClient("ojo"
            de águila")
            .secret("esto es secreto")
            .autorizadosGrantTypes(
                "refresh_token",
                "contraseña",
                "credenciales_cliente"
            .scopes("cliente web","cliente móvil");
    }

    @Anular
    configuración de anulación pública (
        AuthorizationServerEndpointsConfigurer puntos finales)
        lanza una excepción {
        puntos finales
            .authenticationManager(authenticationManager) .userDetailsService(usuarioDetailsService);
}

    Extiende la clase AuthenticationServerConfigurer
    marca la clase con la anotación @Configuration
    ←

    ←

    Anula el método configure().
    Esto define qué clientes van
    para registrar su servicio.
    ←

    Este método define la
    diferentes componentes utilizados
    de AuthorizationServerEndpointsConfigurer, este cod
    le está diciendo a Spring que use
    administrador de autenticación predeterminada
    y servicio de datos de usuario que
    surge la primavera.
    ←
```

Lo primero que debes notar en el código es que estás extendiendo la versión de Spring. clase `AuthenticationServerConfigurer` y luego marcar la clase con un

**@Anotación de configuración . La clase AuthenticationServerConfigurer es una pieza central de Spring Security. Proporciona los mecanismos básicos para llevar a cabo tareas clave. Funciones de autenticación y autorización. Para la clase OAuth2Config estás Vamos a anular dos métodos. El primer método, configure(), se utiliza para definir qué aplicaciones cliente están registradas con su servicio de autenticación. El método config-ure() toma un único parámetro llamado clientes de tipo ClientDetails-ServiceConfigurer. Comencemos a recorrer el código en configure()**

método con un poco más de detalle. Lo primero que debes hacer en este método es registrar cuál las aplicaciones cliente pueden acceder a servicios protegidos por el servicio OAuth2 . Soy usando "acceso" aquí en los términos más amplios, porque usted controla lo que los usuarios del Las aplicaciones cliente pueden hacerlo más tarde comprobando si el usuario que está recibiendo el servicio invocado está autorizado a realizar las acciones que está intentando realizar:

```
clientes.inMemory().withClient("ojode águila")
    .secret("esto es secreto")
    .authorizedGrantTypes("contraseña",
        "creenciales_cliente")
    .scopes("cliente web", "cliente móvil");
```

La clase ClientDetailsServiceConfigurer admite dos tipos diferentes de almacenes para información de aplicaciones: un almacén en memoria y un almacén JDBC . Para los fines de este ejemplo, utilizará el almacén client.inMemory().

Las dos llamadas al método withClient() y secret() proporcionan el nombre del aplicación (eagleeye) que está registrando junto con un secreto (una contraseña, thi-sissecret) que se presentará cuando la aplicación EagleEye llame a su OAuth2 servidor para recibir un token de acceso OAuth2 .

Al siguiente método, autorizedGrantTypes(), se le pasa una lista separada por comas. de los tipos de concesión de autorización que serán admitidos por su servicio OAuth2 . En tus servicio, admitirá la concesión de contraseñas y credenciales de cliente.

El método scopes() se utiliza para definir los límites en los que la aplicación que realiza la llamada puede operar cuando solicita un token de acceso a su servidor OAuth2 . Para Por ejemplo, Thoughtmechanix podría ofrecer dos versiones diferentes de la misma aplicación, una aplicación basada en web y una aplicación basada en teléfono móvil. Cada uno de estos Las aplicaciones pueden usar el mismo nombre de cliente y clave secreta para solicitar acceso a recursos protegidos por el servidor OAuth2 . Sin embargo, cuando las aplicaciones solicitan una clave, deben definir el alcance específico en el que están operando. Al definir el alcance, puede escribir reglas de autorización específicas para el alcance en el que está trabajando la aplicación cliente.

Por ejemplo, es posible que tenga un usuario que pueda acceder a la aplicación EagleEye con tanto el cliente web como el teléfono móvil de la aplicación. Cada versión de la aplicación hace lo siguiente:

- 1 Ofrece la misma funcionalidad
- 2 Es una “aplicación confiable” en la que ThoughtMechanix posee EagleEye aplicaciones front-end y servicios para el usuario final

Por lo tanto, registrará la aplicación EagleEye con el mismo nombre de aplicación y clave secreta, pero la aplicación web solo usará el alcance "webclient", mientras que la versión para teléfono móvil de la aplicación usará el alcance "mobileclient". Al utilizar el alcance, puede definir reglas de autorización en sus servicios protegidos que pueden limitar las acciones que puede realizar una aplicación cliente en función de la aplicación con la que inicia sesión. Esto será independientemente de los permisos que tenga el usuario. Por ejemplo, es posible que desee restringir los datos que un usuario puede ver en función de si utiliza un navegador dentro de la red corporativa o si navega mediante una aplicación en un dispositivo móvil. La práctica de restringir los datos en función del mecanismo de acceso a los datos es común cuando se trata de información confidencial del cliente (como registros médicos o información fiscal).

En este punto, ha registrado una única aplicación, EagleEye, con su servidor OAuth2 . Sin embargo, debido a que está utilizando una concesión de contraseña, debe configurar cuentas de usuario y contraseñas para esos usuarios antes de comenzar.

#### 7.2.3 Configuración de usuarios de EagleEye

Ha definido y almacenado nombres de claves y secretos a nivel de aplicación. Ahora vas a configurar las credenciales de usuario individuales y los roles a los que pertenecen. Los roles de usuario se utilizarán para definir las acciones que un grupo de usuarios puede realizar con un servicio.

Spring puede almacenar y recuperar información del usuario (las credenciales del usuario individual y los roles asignados al usuario) desde un almacén de datos en memoria, una base de datos relacional respaldada por JDBC o un servidor LDAP .

**NOTA** Quiero tener cuidado aquí en términos de definición. La información de la aplicación OAuth2 para Spring puede almacenar sus datos en una base de datos relacional o en memoria. Las credenciales de usuario de Spring y los roles de seguridad se pueden almacenar en una base de datos en memoria, una base de datos relacional o un servidor LDAP (Active Directory). Para simplificar las cosas, porque nuestro objetivo principal es recorrer OAuth2, utilizará un almacén de datos en memoria.

Para los ejemplos de código de este capítulo, definirá roles de usuario utilizando un almacén de datos en memoria. Vas a definir dos cuentas de usuario: john.carnell y william.woodward. La cuenta john.carnell tendrá el rol de USUARIO y la cuenta william.woodward tendrá el rol de ADMIN.

Para configurar su servidor OAuth2 para autenticar ID de usuario, debe configurar un nuevo clase: servicio de autenticación/src/main/com/thinkmechanix/authentication/security/WebSecurityConfigurer.java. El siguiente listado muestra el código de esta clase.

#### Listado 7.3 Definición del ID de usuario, contraseña y roles para su aplicación

```
paquete com.thinkmechanix.authentication.security;  
//Importaciones eliminadas por razones de concisión  
@Configuración
```



Al igual que otras piezas del marco Spring Security, para configurar usuarios (y sus roles), comience extendiendo la clase WebSecurityConfigurerAdapter y márquela con el **@Anotación de configuración**. Spring Security se implementa de una manera similar a cómo ensamblar bloques de Lego para construir un auto o un modelo de juguete. Como tal, es necesario proporcionar al servidor OAuth2 un mecanismo para autenticar usuarios y devolver el usuario información sobre el usuario que se autentica. Esto se hace definiendo dos beans en su Implementación de Spring WebSecurityConfigurerAdapter : authentication-ManagerBean() y userDetailsServiceBean(). Estos dos frijoles están expuestos. utilizando los métodos de autenticación predeterminados authenticationManagerBean() y userDetailsServiceBean() de la clase principal WebSecurityConfigurer-Adapter .

Como recordará del listado 7.2, estos beans se inyectan en el proceso de configuración. (puntos finales AuthorizationServerEndpointsConfigurer) que se muestra en la clase OAuth2Config :

```

configuración de vacío público (
    AuthorizationServerEndpointsConfigurer puntos finales)
        lanza una excepción {
            puntos finales

```

```

    .authenticationManager(authenticationManager) .userDetailsService(usuarioDetailsService);
}

```

Estos dos beans se utilizan para configurar los puntos finales /auth/oauth/token y /auth/user que veremos en acción en breve.

#### 7.2.4 Autenticar al usuario

En este punto, tiene suficiente funcionalidad básica de su servidor OAuth2 para realizar la autenticación de aplicaciones y usuarios para el flujo de concesión de contraseñas. Ahora simulará que un usuario adquiere un token OAuth2 utilizando POSTMAN para PUBLICAR en el punto final http:// localhost:8901/auth/oauth/token y proporcionará la aplicación, la clave secreta, el ID de usuario y la contraseña.

Primero, debe configurar POSTMAN con el nombre de la aplicación y la clave secreta. Pasará estos elementos al punto final de su servidor OAuth2 mediante autenticación básica. La Figura 7.2 muestra cómo está configurado POSTMAN para ejecutar una llamada de autenticación básica.

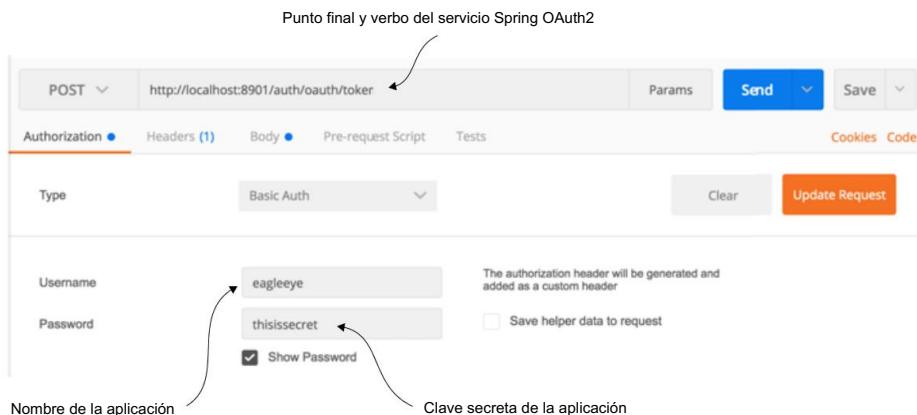


Figura 7.2 Configuración de la autenticación básica utilizando la clave y el secreto de la aplicación

Sin embargo, aún no estás listo para realizar la llamada para obtener el token. Una vez configurados el nombre de la aplicación y la clave secreta, debe pasar la siguiente información en el servicio como parámetros del formulario HTTP :

**Grant\_type:** el tipo de concesión de OAuth2 que está ejecutando. En este ejemplo, utilizará una concesión de contraseña .

**Alcance:** el alcance de las aplicaciones. Debido a que solo definió dos ámbitos legítimos cuando registró la aplicación (cliente web y cliente móvil), el valor pasado debe ser uno de estos dos ámbitos.

Nombre de usuario: nombre del usuario que inicia sesión. Contraseña: contraseña del usuario que inicia sesión.



Parámetros del formulario HTTP

Figura 7.3 Al solicitar un token OAuth2, las credenciales del usuario se pasan como parámetros de formulario HTTP al punto final `/auth/oauth/token`.

A diferencia de otras llamadas REST en este libro, los parámetros en esta lista no se pasarán como un cuerpo de JavaScript. El estándar OAuth2 espera que todos los parámetros se pasen al token. punto final de generación como parámetros de formulario HTTP . La Figura 7.3 muestra cómo se forma HTTP . Los parámetros están configurados para su llamada OAuth2 .

La Figura 7.4 muestra la carga útil de JavaScript devuelta por `/auth/oauth/` llamada simbólica .

La carga útil devuelta contiene cinco atributos:

- access\_token**: el token de OAuth2 que se presentará con cada servicio.
- llamada que realiza el usuario a un recurso protegido.
- token\_type**: el tipo de token. La especificación OAuth2 le permite definir múltiples tipos de tokens. El tipo de token más común utilizado es el portador.
- simbólico. No cubriremos ninguno de los otros tipos de tokens en este capítulo.

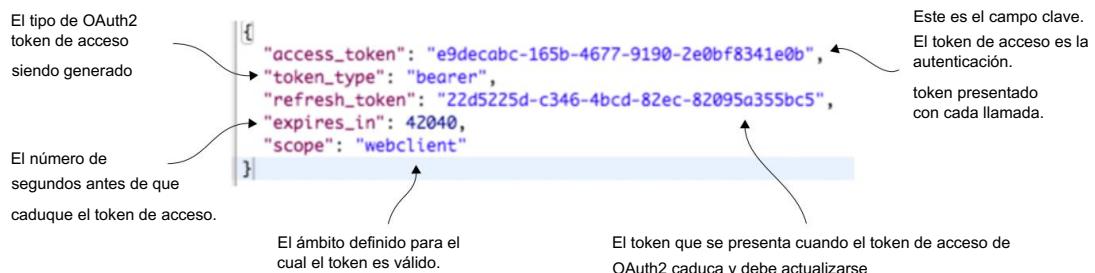


Figura 7.4 Carga útil devuelta después de una validación exitosa de la credencial del cliente

`refresco_token`: contiene un token que se puede presentar de nuevo en OAuth2 .  
 servidor para volver a emitir un token después de que haya  
 caducado. `expires_in`: este es el número de segundos antes de que el token de acceso OAuth2  
 caduca. El valor predeterminado para la caducidad del token de autorización en Spring es  
 12 horas.

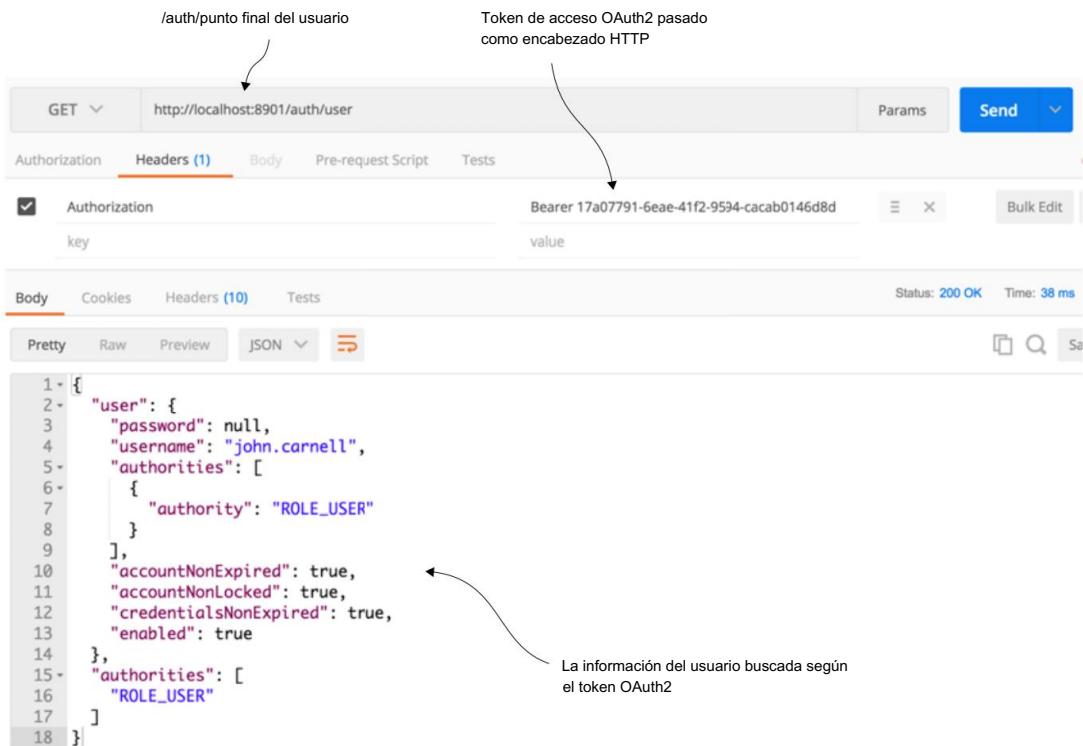
`Alcance`: el alcance para el que es válido este token de OAuth2 .

Ahora que tiene un token de acceso OAuth2 válido , podemos usar el punto final /auth/user que creó en su servicio de autenticación para recuperar información sobre el usuario asociado con el token. Más adelante en el capítulo, cualquier servicio que vaya a ser recurso protegido llamará al punto final /auth/user del servicio de autenticación para validar el token y recuperar la información del usuario.

La Figura 7.5 muestra cuáles serían los resultados si llamara al punto final /auth/user .

Al observar la figura 7.5, observe cómo el token de acceso OAuth2 se pasa como HTTP encabezamiento.

En la figura 7.5, está emitiendo un HTTP GET contra el punto final /auth/user . Sin embargo, cada vez que llame a un punto final protegido OAuth2 (incluido el OAuth2 /auth/ punto final del usuario ) debe pasar el token de acceso OAuth2 . Para hacer esto, siempre



The screenshot shows a Postman request configuration for the endpoint `/auth/punto final del usuario` (GET method, URL `http://localhost:8901/auth/user`). In the `Authorization` tab, there is a `key` field with the value `value` and a `Bearer` prefix. The response body is displayed in JSON format:

```

1. {
2.   "user": {
3.     "password": null,
4.     "username": "john.carnell",
5.     "authorities": [
6.       {
7.         "authority": "ROLE_USER"
8.       }
9.     ],
10.    "accountNonExpired": true,
11.    "accountNonLocked": true,
12.    "credentialsNonExpired": true,
13.    "enabled": true
14.  },
15.  "authorities": [
16.    "ROLE_USER"
17.  ]
18. }

```

An annotation points to the `value` field in the Authorization key with the text: `Token de acceso OAuth2 pasado como encabezado HTTP`. Another annotation points to the JSON response body with the text: `La información del usuario buscada según el token OAuth2`.

Figura 7.5 Búsqueda de información del usuario según el token OAuth2 emitido

cree un encabezado HTTP llamado Autorización y con un valor de Portador XXXXX.

En el caso de su llamada en la figura 7.5, el encabezado HTTP tendrá el valor Al portador e9decabc-165b-4677-9190-2e0bf8341e0b. El token de acceso pasado es el token de acceso devuelto cuando llamó al punto final /auth/oauth/token en figura 7.4.

Si el token de acceso OAuth2 es válido, el punto final /auth/user devolverá información sobre el usuario, incluidos los roles que se le han asignado. Por ejemplo, en la figura 7.10, puede ver que el usuario john.carnell tiene el rol de USUARIO.

**NOTA** Spring asigna el prefijo ROLE\_ a los roles del usuario, por lo que ROLE\_USER significa que john.carnell tiene el rol de USUARIO .

## 7.3 Proteger el servicio de la organización mediante OAuth2

Una vez que haya registrado una aplicación con su servicio de autenticación OAuth2 y configurar cuentas de usuario individuales con roles, puede comenzar a explorar cómo proteger una recurso utilizando OAuth2. Mientras que la creación y gestión de tokens de acceso OAuth2 es responsabilidad del servidor OAuth2 , en Spring, la definición de qué roles de usuario tener permisos para realizar las acciones que ocurren en el nivel de servicio individual.

Para configurar un recurso protegido, debe realizar las siguientes acciones:

Agregue los archivos jar de Spring Security y OAuth2 apropiados al servicio que está proporcionando.  
tectando

Configure el servicio para que apunte a su servicio de autenticación OAuth2  
Definir qué y quién puede acceder al servicio

Comencemos con uno de los ejemplos más simples de configuración de un recurso protegido tomando el servicio de su organización y asegurándose de que solo un usuario autenticado pueda llamarlo.

### 7.3.1 Agregar los archivos jar Spring Security y OAuth2 a los servicios individuales

Como es habitual con los microservicios Spring, debe agregar un par de dependencias al archivo Maven Organization-Service/pom.xml del servicio de organización. Dos dependencias se están agregando: Spring Cloud Security y Spring Security OAuth2. La primavera Los frascos de seguridad en la nube son los frascos de seguridad principales. Contienen código marco, definiciones de anotaciones e interfaces para implementar la seguridad dentro de Spring Cloud. El La dependencia de Spring Security OAuth2 contiene todas las clases necesarias para implementar un Servicio de autenticación OAuth2 . Las entradas de Maven para estas dos dependencias son

```
<dependencia>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>seguridad-nube-spring</artifactId>
</dependencia>
<dependencia>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
</dependencia>
```

### 7.3.2 Configurar el servicio para que apunte a su servicio de autenticación OAuth2

Recuerde que una vez que configure el servicio de la organización como un recurso protegido, cada vez que se realice una llamada al servicio, la persona que llama debe incluir el encabezado HTTP de autenticación que contiene un token de acceso OAuth2 al servicio. Luego, su recurso protegido debe volver a llamar al servicio OAuth2 para ver si el token es válido.

La URL de devolución de llamada se define en el archivo application.yml del servicio de su organización como la propiedad security.oauth2.resource.userInfoUri. Aquí está la configuración de devolución de llamada utilizada en el archivo application.yml del servicio de la organización.

```
seguridad:
  oauth2:
    recurso:
      userInfoUri: http://localhost:8901/auth/user
```

Como puede ver en la propiedad security.oauth2.resource.userInfoUri , la URL de devolución de llamada es al punto final /auth/user . Este punto final se analizó anteriormente en el capítulo de la sección 7.2.4, "Autenticación del usuario".

Por último, también debes informar al servicio de la organización que es un recurso protegido. Nuevamente, esto se hace agregando una anotación de Spring Cloud a la clase de arranque del servicio de la organización. El código de arranque del servicio de la organización se muestra en la siguiente lista y se puede encontrar en la clase Organization-Service/src/main/java/com/thinkmechanix/organization/Application.java .

#### Listado 7.4 Configurar la clase bootstrap para que sea un recurso protegido

```
paquete com.thinkmechanix.organization;

//La mayoría de las importaciones se eliminaron por
razones de concisión import org.springframework.security.oauth2.
    config.annotation.web.configuration.EnableResourceServer;

    @SpringBootApplication
    @EnableEurekaClient
    @EnableCircuitBreaker
    @EnableResourceServer
    Aplicación de clase pública {
        @Frijol
    Filtro público userContextFilter() {
        UserContextFilter userContextFilter = nuevo UserContextFilter(); devolver usuarioContextFilter; }

    público estático vacío principal (String [] argumentos) {
        SpringApplication.run(Aplicación.clase, argumentos); }}}


```

La anotación @EnableResourceServer se utiliza para indicarle a su microservicio que es un recurso protegido.

```
público estático vacío principal (String [] argumentos) {
    SpringApplication.run(Aplicación.clase, argumentos); }}
```

La anotación @EnableResourceServer le dice a Spring Cloud y Spring Security que el servicio es un recurso protegido. @EnableResourceServer aplica un filtro que intercepta todas las llamadas entrantes al servicio y verifica si hay un OAuth2 .

token de acceso presente en el encabezado HTTP de la llamada entrante y luego vuelve a llamar a la URL de devolución de llamada definida en security.oauth2.resource.userInfoUri para ver si el token es válido. Una vez que sabe que el token es válido, la anotación @EnableResourceServer también aplica cualquier regla de control de acceso sobre quién y qué puede acceder a un servicio.

### 7.3.3 Definir quién y qué puede acceder al servicio

Ahora está listo para comenzar a definir las reglas de control de acceso del servicio. A Para definir reglas de control de acceso, debe extender una clase Spring ResourceServerConfigurerAdapter y anular el método configure() de la clase . En el servicio de organización, su clase ResourceServerConfiguration se encuentra en organización -servicio/src/main/java/com/thinkmechanix/organization/security/ ResourceServerConfiguration.java. Las reglas de acceso pueden variar desde extremadamente De grano grueso (cualquier usuario autenticado puede acceder a todo el servicio) a grano fino. (solo se permite la aplicación con este rol, accediendo a esta URL a través de un DELETE ).

Discutimos cada permutación de las reglas de control de acceso de Spring Security, pero podemos Mire varios de los ejemplos más comunes. Estos ejemplos incluyen proteger una recurso para que

Sólo los usuarios autenticados pueden acceder a una URL de servicio

Solo los usuarios con un rol específico pueden acceder a una URL de servicio

#### PROTECCIÓN DE UN SERVICIO POR PARTE DE UN USUARIO AUTENTICADO

Lo primero que vas a hacer es proteger el servicio de la organización para que sólo pueda ser accedido por un usuario autenticado. La siguiente lista muestra cómo puedes construir esta regla en la clase ResourceServerConfiguration.java .

#### Listado 7.5 Restringir el acceso sólo a usuarios autenticados

```
paquete com.thinkmechanix.organization.security;
//Importaciones eliminadas por razones de concisión
@Configuration se
extiende la clase pública ResourceServerConfiguration
    ResourceServerConfigurerAdapter{
        @Override
        public void configure(HttpSecurity http) arroja una excepción {
            http.authorizeRequests().anyRequest().authenticated();
        }
    }
```

Todas las reglas de acceso se definen dentro del método configure() anulado

La clase debe estar marcada con la anotación @Configuration.

La configuración del servicio de recursos la clase necesita extender ResourceServerConfigurerAdapter.

Todas las reglas de acceso están configuradas. fuera del objeto HttpSecurity pasado al método.

Todas las reglas de acceso se definirán dentro del método configure() . Usarás la clase HttpSecurity pasada por Spring para definir sus reglas. En este ejemplo, restringirá todo acceso a cualquier URL en el servicio de la organización únicamente a usuarios autenticados.

Si accediera al servicio de la organización sin un token de acceso OAuth2 presente en el encabezado HTTP , obtendría un código de respuesta HTTP 401 junto con un mensaje que indica que se requiere una autenticación completa en el servicio.

La Figura 7.6 muestra el resultado de una llamada al servicio de la organización sin el Encabezado HTTP OAuth2 .

The screenshot shows a POST request to `http://localhost:8085/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a`. The 'Headers' tab is selected. The response status is 401 Unauthorized, with a detailed error message in JSON:

```

1 - {
2   "error": "unauthorized",
3   "error_description": "Full authentication is required to access this resource"
4 }

```

A callout points to the error message with the annotation: "JSON indica el error e incluye una descripción más detallada." Another callout points to the status code with the annotation: "Se devuelve el código de estado HTTP 401."

Figura 7.6 Intentar llamar al servicio de la organización resultará en una llamada fallida.

A continuación, llamará al servicio de la organización con un token de acceso OAuth2 . Para obtener un token de acceso, consulte la sección 7.2.4, “Autenticación del usuario”, sobre cómo generar el token OAuth2 . Desea cortar y pegar el valor del campo `access_token` de la llamada de JavaScript devuelta al punto final `/auth/oauth/token` y utilizarlo en su llamada al servicio de la organización. Recuerde, cuando llama al servicio de la organización, debe agregar un encabezado HTTP llamado Autorización con el valor Valor de token de acceso del portador.

The screenshot shows a POST request to `http://localhost:5555/api/organization/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a`. The 'Headers' tab is selected, showing an 'Authorization' header with the value `Bearer fc81f8d2-57fb-4at7-b01d-a40c2ea4e5`. The response status is 200 OK, with a JSON payload:

```

1 - {
2   "id": "e254f8c-c442-4ebe-a82a-e2fc1d1ff78a",
3   "name": "customer-crm-co",
4   "contactName": "Mark Balster",
5   "contactEmail": "mark.balster@custcrmco.com",
6   "contactPhone": "823-555-1212"
7 }

```

A callout points to the 'Authorization' header with the annotation: "El token de acceso de OAuth2 se pasa en el encabezado."

Figura 7.7 Pasar el token de acceso OAuth2 en la llamada al servicio de la organización

La Figura 7.7 muestra la llamada al servicio de la organización, pero esta vez con un OAuth2 token de acceso que se le pasó.

Este es probablemente uno de los casos de uso más simples para proteger un punto final usando OAuth2. A continuación, se basará en esto y restringirá el acceso a un punto final específico a una función específica.

#### PROTEGER UN SERVICIO A TRAVÉS DE UN ROL ESPECÍFICO

En el siguiente ejemplo, bloqueará la llamada DELETE en su organización. servicio sólo a aquellos usuarios con acceso ADMIN . Como recordará de la sección 7.2.3, "Configurando algunos usuarios de EagleEye", creó dos cuentas de usuario que podían acceder Servicios EagleEye: john.carnell y william.woodward. La cuenta de john.carnell tenía asignado el rol de USUARIO . La cuenta de william.woodward tenía el rol USUARIO y el rol ADMIN .

La siguiente lista muestra cómo configurar el método configure() para restringir el acceso al punto final DELETE solo para aquellos usuarios autenticados que tienen el rol ADMIN .

#### Listado 7.6 Restringir eliminaciones solo al rol ADMIN

```
paquete com.thinkmechanix.organization.security;

//Importaciones eliminadas por razones de concisión
@Configuration
La clase pública ResourceServerConfiguration se extiende
    ResourceServerConfigurerAdapter{
        @Anular
        La configuración pública vacía (HttpSecurity http) arroja una excepción {
            http
                .autorizarRequests()
                .antMatchers(HttpMethod.DELETE, "/v1/organizations/**") .hasRole("ADMIN") .anyRequest()
                    .autenticado();
        }
    }
```

En el listado 7.6, está restringiendo la llamada DELETE en cualquier punto final que comience con /v1/organizaciones en su servicio al rol ADMIN .:

```
.autorizarRequests()
    .antMatchers(HttpMethod.DELETE, "/v1/organizaciones/**") .hasRole("ADMIN")
```

El método antMatcher() puede tomar una lista de puntos finales separados por comas. Estos puntos finales pueden utilizar una notación de estilo comodín para definir los puntos finales que desea acceso. Por ejemplo, si desea restringir cualquiera de las llamadas DELETE independientemente de la versión en el nombre de la URL , puede usar un \* en lugar del número de versión en su Definiciones de URL :

```
.autorizarRequests()
    .antMatchers(HttpMethod.DELETE, "/*/organizaciones/**")
    .hasRole("ADMINISTRO")
```

La última parte de la definición de la regla de autorización aún define que un usuario autenticado debe acceder a cualquier otro punto final de su servicio:

```
.cualquierRequest() .autenticado();
```

Ahora, si obtuviera un token OAuth2 para el usuario john.carnell (contraseña: contraseña1) e intentara llamar al punto final DELETE para el servicio de la organización (<http://localhost:8085/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a>), obtendría un código de estado HTTP 401 en la llamada y un mensaje de error que indica que el acceso fue denegado. El texto JavaScript devuelto por su llamada sería

```
{ "error": "acceso_denied",  
  "error_description": "Acceso denegado" }
```

Si intenta exactamente la misma llamada usando la cuenta de usuario william.woodward (contraseña: contraseña2) y su token OAuth2 , verá que se devuelve una llamada exitosa (un código de estado HTTP 204 – Sin contenido) y esa organización ser eliminado por el servicio de organización.

En este punto, hemos visto dos ejemplos simples de cómo llamar y proteger un único servicio (el servicio de la organización) con OAuth2. Sin embargo, a menudo en un entorno de microservicios, se utilizarán varias llamadas de servicio para realizar una sola transacción. En este tipo de situaciones, debe asegurarse de que el token de acceso OAuth2 se propague de una llamada de servicio a otra.

#### 7.3.4 Propagar el token de acceso OAuth2

Para demostrar la propagación de un token de OAuth2 entre servicios, ahora veremos cómo proteger su servicio de licencias con OAuth2. Recuerde, el servicio de licencias llama al servicio de la organización para buscar información. La pregunta es: ¿cómo se propaga el token OAuth2 de un servicio a otro?

Configurará un ejemplo simple en el que el servicio de licencias llamará al servicio de la organización. Sobre la base de los ejemplos del capítulo 6, ambos servicios se ejecutan detrás de una puerta de enlace Zuul.

La Figura 7.8 muestra el flujo básico de cómo el token OAuth2 de un usuario autenticado fluirá a través de la puerta de enlace de Zuul, al servicio de licencias y luego al servicio de la organización.

La siguiente actividad ocurre en la figura 7.8:

- 1 El usuario ya se ha autenticado en el servidor OAuth2 y realiza una llamada a la aplicación web EagleEye. El token de acceso OAuth2 del usuario se almacena en la sesión del usuario. La aplicación web EagleEye necesita recuperar algunos datos de licencia y realizará una llamada al punto final REST del servicio de licencia . Como parte de la llamada al punto final REST de licencia , la aplicación web EagleEye agregará el token de acceso OAuth2 a través del encabezado HTTP "Autorización". Solo se puede acceder al servicio de licencias detrás de una puerta de enlace de servicios Zuul.

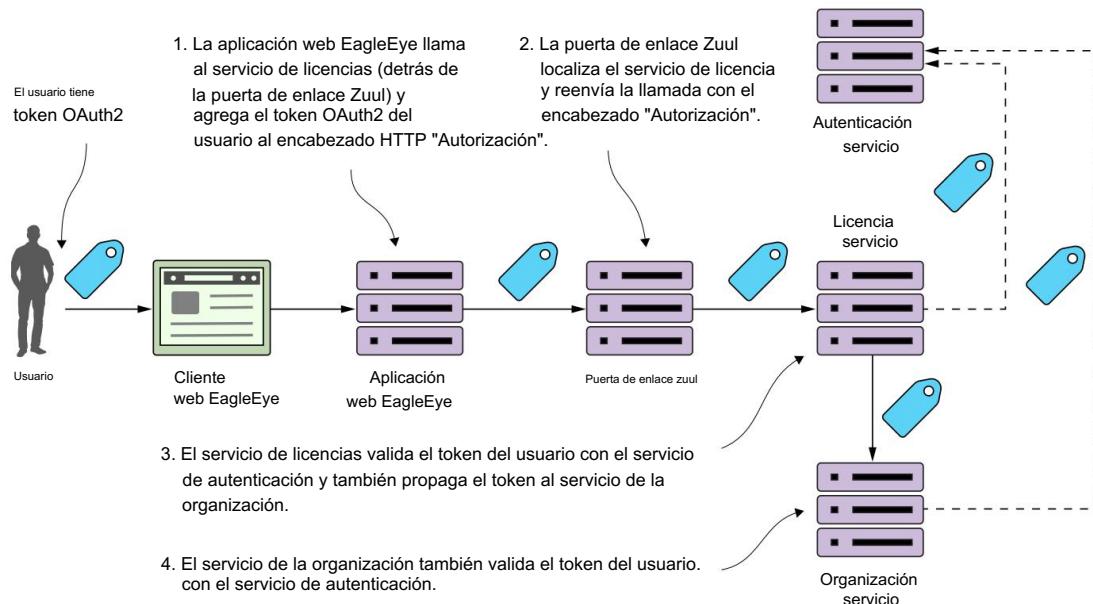


Figura 7.8 El token OAuth2 debe transportarse a lo largo de toda la cadena de llamadas.

**2** Zuul buscará el punto final del servicio de licencias y luego reenviará la llamada a uno de los servidores de servicios de licencia. La puerta de enlace de servicios deberá copiar el encabezado HTTP de "autorización" de la llamada entrante y asegúrese de que el encabezado HTTP "Autorización" se reenvíe al nuevo punto final.

**3** El servicio de licencias recibirá la llamada entrante. Debido a que el servicio de licencias es un recurso protegido, el servicio de licencias validará el token con servicio OAuth2 de EagleEye y luego verifique los roles del usuario para obtener los permisos.

Como parte de su trabajo, el servicio de licencias invoca el servicio de organización. Al realizar esta llamada, el servicio de licencias necesita propagar el OAuth2 del usuario. token de acceso al servicio de la organización.

**4** Cuando el servicio de la organización reciba la llamada, volverá a tomar el token del encabezado HTTP "Autorización" y lo validará con el servidor EagleEye OAuth2 .

Para implementar estos flujos, debe hacer dos cosas. Primero, necesitas modificar tu Puerta de enlace de servicios Zuul para propagar el token OAuth2 al servicio de licencias. Por defecto, Zuul no reenviará encabezados HTTP confidenciales como Cookie, Set-Cookie, y Autorización a servicios posteriores. Para permitir que Zuul propague el encabezado HTTP "Autorización", debe establecer la siguiente configuración en el archivo application.yml o almacenamiento de datos Spring Cloud Config de su puerta de enlace de servicios Zuul:

```
zuul.SENSITIVEHeaders: Cookie, Set-Cookie
```

Esta configuración es una lista negra de encabezados confidenciales que Zuul evitara que se publiquen. propagado a un servicio descendente. La ausencia del valor de Autorización en el La lista anterior significa que Zuul lo permitirá pasar. Si no configura la propiedad zuul.SENSITIVE-HEADERS en absoluto, Zuul bloqueará automáticamente los tres valores para que no se muestren. propagado (Cookie, Set-Cookie y Autorización).

### ¿Qué pasa con las otras capacidades OAuth2 de Zuul?

Zuul puede propagar automáticamente tokens de acceso OAuth2 descendentes y autorizar solicitudes entrantes contra el servicio OAuth2 utilizando la anotación @EnableOAuth2SSo. No he utilizado este enfoque a propósito porque mi objetivo en este capítulo es mostrar los conceptos básicos de cómo funciona OAuth2 sin agregar otro nivel de complejidad (o depuración). Si bien la configuración de la puerta de enlace del servicio Zuul no es demasiado complicada, habría agregado significativamente más contenido a un capítulo ya de por sí extenso. Si Está interesado en que un portal de servicios de Zuul participe en el inicio de sesión único. (SSO), la documentación de Spring Cloud Security tiene un tutorial breve pero completo que cubre la configuración del servidor Spring (<http://cloud.spring.io/spring-cloud-security/spring-cloud-security.html> ).

Lo siguiente que debe hacer es configurar su servicio de licencias para que sea OAuth2. servicio de recursos y configure las reglas de autorización que desee para el servicio. Eran No vamos a discutir en detalle la configuración del servicio de licencia porque ya discutimos las reglas de autorización en la sección 7.3.3, "Definir quién y qué puede acceder a la servicio."

Finalmente, todo lo que necesita hacer es modificar cómo el código en el servicio de licencia llama al servicio de organización. Debe asegurarse de que el encabezado HTTP "Autorización" esté injectado en la aplicación llama al servicio de Organización. Sin Spring Security, tendría que escribir un filtro de servlet para tomar el encabezado HTTP de la llamada de servicio de licencia entrante y luego agregarlo manualmente a cada llamada de servicio saliente en la licencia. servicio. Spring OAuth2 proporciona una nueva clase de plantilla Rest que admite llamadas de OAuth2. La clase se llama OAuth2RestTemplate. Para utilizar la clase OAuth2RestTemplate, Primero debemos exponerlo como un bean que puede conectarse automáticamente a un servicio que llama a otro. Servicios protegidos OAuth2 . Esto se hace en licensing-service/src/main/java/com/thinktmechanix/licenses/Application.java clase:

```
@Frijol
público OAuth2RestTemplate oauth2RestTemplate(
    OAuth2ClientContext oauth2ClientContext,
    Detalles de OAuth2ProtectedResourceDetails) {
    devolver nuevo OAuth2RestTemplate (detalles, oauth2ClientContext);
}
```

Para ver la clase OAuth2RestTemplate en acción, puede buscar en licensing-service/src/main/java/com/thinktmechanix/licenses/clients/

Clase OrganizationRestTemplate.java . La siguiente lista muestra cómo OAuth2 RestTemplate está conectado automáticamente a esta clase.

Listado 7.7 Uso de OAuth2RestTemplate para propagar el token de acceso OAuth2

```

paquete com.thinkmechanix.licenses.clients;

// Eliminado por motivos de concisión

@Component
clase pública OrganizationRestTemplateClient {
@autocableado
OAuth2RestTemplate restTemplate;

registrador final estático privado Logger =
    LoggerFactory.getLogger(
        OrganizationRestTemplateClient.clase);

Organización pública getOrganization (String organizaciónId) {
    logger.debug("En el servicio de licencias
        .getOrganization: {}",
        UserContext.getCorrelationId());

    ResponseEntity<Organización> restExchange =
        restoTemplate.exchange(
            "http://zuulserver:5555/api/organización
                /v1/organizaciones/{organizationId}",
            Método Http.GET,
            nulo, Organización.clase, ID de organización);

    /*Guardar el registro del caché*/
    devolver restExchange.getBody();
}
}

```

OAuth2RestTemplate es un reemplazo directo para el RestTemplate estándar y maneja la propagación del Token de acceso OAuth2.

La invocación del servicio de organización se realiza exactamente de la misma manera que un RestTemplate estándar.

## 7.4 Tokens web de JavaScript y OAuth2

OAuth2 es un marco de autenticación basado en tokens, pero irónicamente no proporciona cualquier estándar sobre cómo se deben definir los tokens en su especificación. para rectificar el falta de estándares en torno a los tokens OAuth2 , está surgiendo un nuevo estándar llamado JavaScript Fichas web (JWT). JWT es un estándar abierto (RFC-7519) propuesto por Internet Engineering Task Force (IETF) que intenta proporcionar una estructura estándar para Fichas OAuth2 . Los tokens JWT son

Pequeño: los tokens JWT están codificados en Base64 y se pueden pasar fácilmente a través de una URL.  
Encabezado HTTP o un parámetro HTTP POST .

Firmado criptográficamente: el servidor de autenticación firma un token JWT que lo emite. Esto significa que puede tener la garantía de que el token no ha sido manipulado.

Autocontenido: debido a que un token JWT está firmado criptográficamente, se puede garantizar al microservicio que recibe el servicio que el contenido del token es válido. No es necesario volver a llamar al servicio de autenticación para validar el contenido del token porque la firma del token se puede validar y

El microservicio receptor puede inspeccionar el contenido (como el tiempo de vencimiento del token y la información del usuario).

Extensible: cuando un servicio de autenticación genera un token, puede colocar información adicional en el token antes de sellar el token. Un servicio receptor puede descifrar la carga útil del token y recuperar ese contexto adicional.

Spring Cloud Security admite JWT desde el primer momento. Sin embargo, para usar y consumir tokens JWT , su servicio de autenticación OAuth2 y los servicios protegidos por el servicio de autenticación deben configurarse de una manera diferente. La configuración no es difícil, así que veamos el cambio.

**NOTA** He elegido mantener la configuración de JWT en una rama separada (llamada JWT\_Example) en el repositorio de GitHub para este capítulo (<https://github.com/carnellijs/spmia-chapter7>) porque la configuración estándar de Spring Cloud Security OAuth2 y la configuración de OAuth2 basada en JWT requieren diferentes clases de configuración.

#### 7.4.1 Modificación del servicio de autenticación para emitir tokens web de JavaScript

Tanto para el servicio de autenticación como para los dos microservicios (servicio de licencias y organización) que estarán protegidos por OAuth2, deberá agregar una nueva dependencia de Spring Security a sus archivos Maven pom.xml para incluir las bibliotecas JWT OAuth2 . . Esta nueva dependencia es

```
<dependencia>
    <groupId>org.springframework.security</groupId> <artifactId>spring-security-jwt</artifactId>

</dependencia>
```

Después de agregar la dependencia de Maven, primero debe decirle a su servicio de autenticación cómo generará y traducirá tokens JWT . Para hacer esto, configurará en el servicio de autenticación una nueva clase de configuración llamada servicio de autenticación/src/java/com/thinktmechanix/authentication/security/JWT TokenStoreConfig.java. El siguiente listado muestra el código de la clase.

##### Listado 7.8 Configurando el almacén de tokens JWT

```
@Configuration clase
pública JWTTokenStoreConfig {

    @autocableado
    serviceConfig privado serviceConfig;

    @Frijol
    tienda de tokens pública tienda de tokens() {
        devolver nuevo JwtTokenStore(jwtAccessTokenConverter());
    }

    @Frijol
    @Primario
```

La anotación @Primary se usa para indicarle a Spring que si hay más de un bean de un tipo específico (en este caso DefaultTokenService), use el tipo de bean marcado como @Primary para la autoinyección.

```

público DefaultTokenServices tokenServices() {
    Servicios de token predeterminados Servicios de token predeterminados
        = nuevos DefaultTokenServices();
    defaultTokenServices.setTokenStore(tokenStore());
    defaultTokenServices.setSupportRefreshToken (verdadero);
    devolver defaultTokenServices;
}

@Frijol
público JwtAccessTokenConverter jwtAccessTokenConverter() {
    Convertidor JwtAccessTokenConverter =
        nuevo JwtAccessTokenConverter();
    convertidor
        .setSigningKey(serviceConfig.getJwtSigningKey());
    convertidor de retorno;
}

@Frijol
TokenEnhancer público jwtTokenEnhancer() {
    devolver nuevo JWTTokenEnhancer();
}
}

```

Annotations and their descriptions:

- `@Frijol` (line 1): Se utiliza para leer datos. hacia y desde un token presentado al servicio.
- `@Frijol` (line 2): Actúa como traductor. entre JWT y el servidor OAuth2.
- `@Frijol` (line 3): Define la firma clave que se utilizará firmar una ficha.

La clase JWTTokenStoreConfig se utiliza para definir cómo Spring gestionará la creación, firma y traducción de un token JWT . El método tokenServices() va utilizar la implementación de servicios de token predeterminada de Spring Security, por lo que el trabajo aquí es de memoria. El método jwtAccessTokenConverter() es en el que queremos centrarnos. Se define cómo se traducirá el token. Lo más importante a tener en cuenta sobre esto. El método es que estás configurando la clave de firma que se usará para firmar tu token.

Para este ejemplo, usará una clave simétrica, lo que significa que tanto la El servicio de autenticación y los servicios protegidos por el servicio de autenticación deben compartir la misma clave entre todos los servicios. La clave no es más que una cadena aleatoria de valores que se almacena en los servicios de autenticación Spring Cloud Config. entrada (<https://github.com/carnelli/config-repo/blob/master/authenticationservice/authenticationservice.yml>). El valor real de la clave de firma es

clave.de.firma: "345345fsdgsf5345"

**NOTA** Spring Cloud Security admite el cifrado de clave simétrica y el cifrado asimétrico mediante claves públicas/privadas. no vamos a caminar mediante la configuración de JWT utilizando claves públicas/privadas. Desafortunadamente, poco oficial. Existe documentación sobre JWT, Spring Security y claves públicas/privadas. Si Si está interesado en cómo hacer esto, le recomiendo que consulte Bael-dung.com (<http://www.baeldung.com/spring-security-oauth-jwt>). ellos hacen un Excelente trabajo al explicar JWT y la configuración de clave pública/privada.

En JWTTokenStoreConfig del listado 7.8, usted definió cómo eran los tokens JWT . va a ser firmado y creado. Ahora necesitas conectar esto a tu OAuth2 general. servicio. En el listado 7.2 utilizó la clase OAuth2Config para definir la configuración de

su servicio OAuth2 . Configuraste el administrador de autenticación que iba a ser utilizado por su servicio junto con el nombre de la aplicación y los secretos. Vas a ir a reemplace la clase OAuth2Config con una nueva clase llamada servicio de autenticación/ src/main/java/com/thinkmechanix/authentication/security /JWTOAuth2Config.java.

La siguiente lista muestra el código para la clase JWTOAuth2Config .

Listado 7.9 Conectando JWT a su servicio de autenticación a través de JWTOAuth2Config clase

```
paquete com.thinkmechanix.authentication.security;

//Importaciones eliminadas por razones de concisión
@Configuration
la clase pública JWTOAuth2Config se extiende
AuthorizationServerConfigurerAdapter {

    @autocableado
    administrador de autenticación privado administrador de autenticación;

    @autocableado
    servicio de detalles de usuario privado servicio de detalles de usuario;

    @autocableado
    tokenStore privado tokenStore;

    @autocableado
    servicios de token privados DefaultTokenServices;

    @autocableado
    privado JwtAccessTokenConverter jwtAccessTokenConverter;

    @Anular
    configuración de vacío público (
        AuthorizationServerEndpointsConfigurer puntos finales)

    lanza una excepción {
        TokenEnhancerChain tokenEnhancerChain =
            nueva TokenEnhancerChain(); tokenEnhancerChain

            .setTokenEnhancers(
                matrices.asList(
                    jwtTokenEnhancer,
                    jwtAccessTokenConverter);

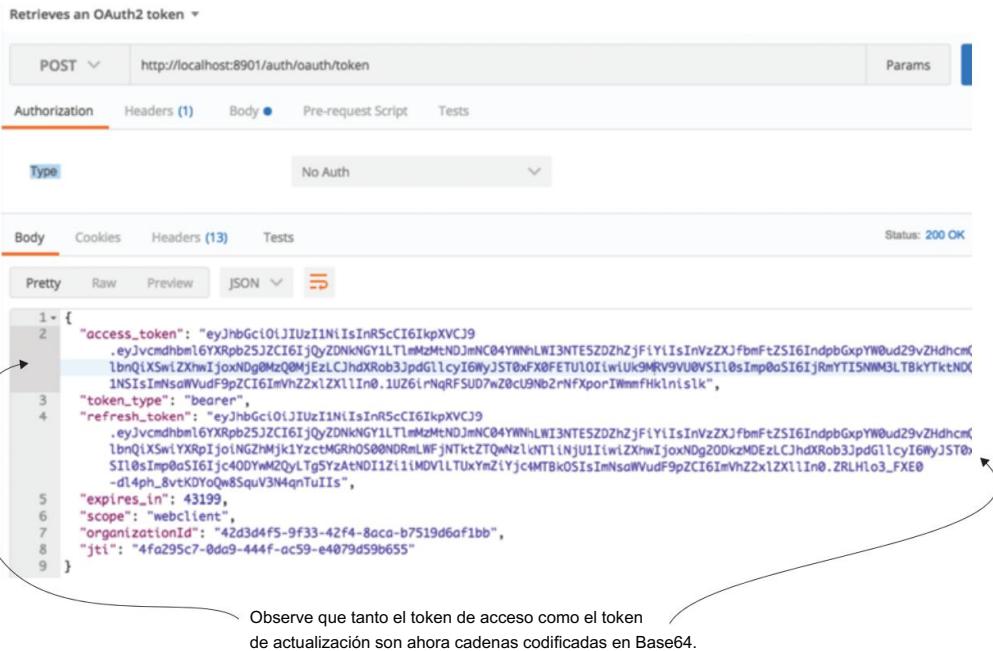
            puntos finales
            .tokenStore(tokenStore)

        .accessTokenConverter(jwtAccessTokenConverter) .authenticationManager(authenticationManager) .userDetailsService(userDetailsService)
    }

    //Se eliminó el resto de la clase por razones de concisión.
}
```

El almacen de tokens que definiste en  
El listado 7.8 se injectará aquí.

Este es el gancho para  
indicarle al código  
Spring Security OAuth2 que use JWT.



```

1 - {
2   "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
3     .eyJvcmdhbml6YXRpb25ZC16IjQyZDNkNGY1LTlmMzMtNDJmNC04YWNhLWI3NTE5ZDZhZjF1YiIsInVzZXJfbmFtZSI6IndpbGxpYW0ud29vZHdhcm
4     lbnQiXSwiZXhwIjoxNDg0MzQ0MjgzLCjhDXRob3JpdGlcIcyI6WyJST0xFETU0Iiwiiuk9MRV9VU0VS10sImp0aSI6IjRmYTISNWMBLTBkYTkthDK
5     1NS1sImNsawVudef9pZCI6ImVmZ2x1ZXl1In0.UUZ6iNqRFSDU7wZ0cU9Nb2rNfxporIWmmfHklnslk",
6   "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
7     .eyJvcmdhbml6YXRpb25ZC16IjQyZDNkNGY1LTlmMzMtNDJmNC04YWNhLWI3NTE5ZDZhZjF1YiIsInVzZXJfbmFtZSI6IndpbGxpYW0ud29vZHdhcm
8     lbnQiXSwiYXRpIjoiNGZhMjk1YzctMGRhOS00NDrlWFjNTktZTQnNzl0NTliNjU1IiwiZXhwIjoxNDg20DkzMDExLCjhDRob3JpdGlcI6WyJST0>
9     SI0sImp0aSI6Ijic4ODYmM2QyLtg5YzAtNDI1Z11MDVlLTUxYmZlYjc4MTBkOSisImNsawVudef9pZCI6ImVmZ2x1ZXl1In0.ZRLHlo3_FXE0
-d14ph_8vEKDYQw8SquV3N4qnTuII",
10  "expires_in": 43199,
11  "scope": "webclient",
12  "organizationId": "42d3df5-9f33-42f4-8aca-b7519d6af1bb",
13  "jti": "4fa295c7-0da9-444f-ac59-e4079d59b655"
}

```

Observe que tanto el token de acceso como el token de actualización son ahora cadenas codificadas en Base64.

Figura 7.9 Los tokens de acceso y actualización de su llamada de autenticación ahora son tokens JWT.

Ahora, si reconstruye su servicio de autenticación y lo reinicia, debería ver un token basado en JWT devuelto. La Figura 7.9 muestra los resultados de su llamada al servicio de autenticación ahora que usa JWT.

El token real en sí no se devuelve directamente como JavaScript. En cambio, el cuerpo de JavaScript está codificado utilizando una codificación Base64. Si está interesado en ver el contenido de un token JWT, puede utilizar herramientas en línea para decodificar el token. Me gusta utilizar una herramienta en línea de una empresa llamada Stormpath. Su herramienta, <http://jsonwebtoken.io>, es un decodificador JWT en línea. La Figura 7.10 muestra el resultado del token decodificado.

**NOTA** Es extremadamente importante comprender que sus tokens JWT están firmados, pero no cifrados. Cualquier herramienta JWT en línea puede decodificar el token JWT y exponer su contenido. Menciono esto porque la especificación JWT le permite extender el token y agregar información adicional al token.

No exponga información confidencial o de identificación personal (PII) en sus tokens JWT.

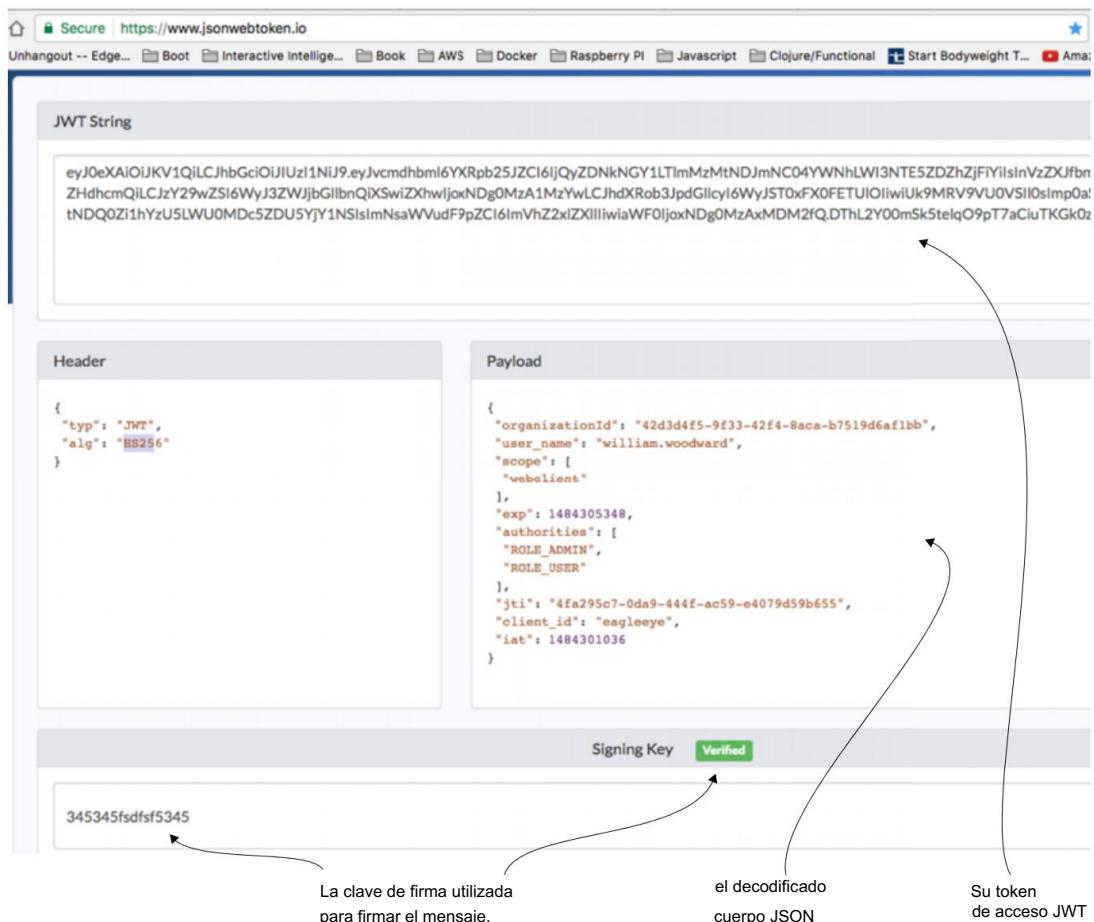


Figura 7.10 Usando <http://jswebtoken.io> le permite decodificar el contenido.

#### 7.4.2 Consumir tokens web de JavaScript en sus microservicios

Ahora tiene su servicio de autenticación OAuth2 creando tokens JWT . El siguiente paso es configurar sus servicios de organización y licencias para utilizar JWT. Este es un ejercicio trivial que requiere que hagas dos cosas:

- 1 Agregue la dependencia spring-security-jwt tanto al servicio de licencias como al archivo pom.xml del servicio de organización. (Consulte el comienzo de la sección 7.4.1, “Modificación del servicio de autenticación para emitir tokens web de JavaScript”, para conocer la dependencia exacta de Maven que debe agregarse).
- 2 Configure una clase JWTTokenStoreConfig tanto en los servicios de licencia como en los de organización. Esta clase es casi exactamente la misma clase que utiliza el servicio de autenticación (ver listado 7.8). No voy a repasar el mismo material otra vez, pero puedes

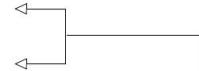
consulte ejemplos de la clase JWTTokenStoreConfig tanto en licensing-service/src/main/com/thinktmechanix/licensing-service/security/JWTTokenStoreConfig.java como en Organization-Service/src/main/com/thinktmechanix/organization-service/security/JWTTokenStoreConfig. Clases .java .

Necesitas hacer un último trabajo. Dado que el servicio de licencias llama al servicio de la organización, debe asegurarse de que se propague el token OAuth2 . Esto normalmente se hace a través de la clase OAuth2RestTemplate ; sin embargo, la clase OAuth2RestTemplate no propaga tokens basados en JWT. Para asegurarse de que su servicio de licencias haga esto, debe agregar un bean RestTemplate personalizado que realizará esta inyección por usted. Este RestTemplate personalizado se puede encontrar en la clase licensing-service/src/main/java/com/thinktmechanix/licenses/Application.java .

La siguiente lista muestra esta definición de bean personalizado.

**Listado 7.10 Creando una clase RestTemplate personalizada para inyectar el token JWT**

```
Aplicación de clase pública {
    //Código eliminado por razones de concisión
    @Primario
    @Frijol
    plantilla de descanso pública getCustomRestTemplate() {
        Plantilla RestTemplate = nueva RestTemplate(); Lista de interceptores
        = template.getInterceptors(); if (interceptores == nulo)
        { plantilla.setInterceptores(
            Colecciones.singletonList(
                nuevo UserContextInterceptor())); } else
        { interceptores.add(new UserContextInterceptor());
            template.setInterceptores(interceptores);
        }
        plantilla de devolución;
    }
}
```



UserContextInterceptor  
inyectará el encabezado de  
Autorización en cada llamada de Rest.

En el código anterior, está definiendo un bean RestTemplate personalizado que utilizará un ClientHttpRequestInterceptor. Recuerde del capítulo 6 que ClientHttp-RequestInterceptor es una clase Spring que le permite conectar la funcionalidad para que se ejecute antes de realizar una llamada basada en REST. Esta clase de interceptor es una variación de la clase UserContextInterceptor que definió en el capítulo 6. Esta clase se encuentra en licensing-service/src/main/java/com/thinktmechanix/licenses/utils /UserContextInterceptor.java. El siguiente listado muestra esta clase.

**Listado 7.11 El UserContextInterceptor inyectará el token JWT en su**

llamadas DESCANSO

la clase pública UserContextInterceptor implementa  
ClientHttpRequestInterceptor {

```

@Anular
intercepción pública ClientHttpResponse {
    Solicitud HttpRequest, byte[] cuerpo, ejecución
    ClientHttpRequestExecution) arroja IOException {

        headers.add(UserContext.CORRELATION_ID,
                    UserContextHolder.getContext().getCorrelationId());
        headers.add(UserContext.AUTH_TOKEN,
                    UserContextHolder.getContext().getAuthToken());

        devolver ejecución.execute (solicitud, cuerpo);
    }
}

```

Agregar el token de autorización  
al encabezado HTTP

UserContextInterceptor utiliza varias de las clases de utilidad del capítulo 6 . Recuerde, cada uno de sus servicios utiliza un filtro de servlet personalizado (llamado User-ContextFilter) para analizar el token de autenticación y el ID de correlación del encabezado HTTP . En el listado 7.11, está utilizando el valor UserContext.AUTH\_TOKEN ya analizado para completar la llamada HTTP saliente.

Eso es todo. Con estas piezas en su lugar, ahora puede llamar al servicio de licencias (o al servicio de la organización) y colocar el JWT codificado en Base64 codificado en su encabezado de Autorización HTTP con el valor Portador <>JWT-Token<>, y su servicio leerá y validará correctamente. el token JWT .

#### 7.4.3 Ampliación del token JWT

Si observa detenidamente el token JWT en la figura 7.10, notará el campo EagleEye organizationId . (La Figura 7.11 muestra una toma más ampliada del token JWT que se muestra

Header	Payload
<pre>{   "typ": "JWT",   "alg": "HS256" }</pre>	<pre>{   "organizationId": "42d3d4f5-9f33-42f4-8aca-b7519d6af1bb",   "user_name": "william.woodward",   "scope": [     "webclient"   ], }</pre>

Este no es un campo JWT estándar.

Figura 7.11 Un ejemplo de extensión del token JWT con un OrganizationId

anteriormente en la figura 7.10.) Este no es un campo de token JWT estándar . Es uno que agregué inyectando un nuevo campo en el token JWT mientras se creaba.

Extender un token JWT se puede hacer fácilmente agregando un potenciador de token Spring OAuth2 class a su servicio de autenticación. La fuente de esta clase se puede encontrar en authentication-service/src/main/java/com/thinkmechanix/authentication /security/JWTTokenEnhancer.java clase. La siguiente lista muestra este código.

**Listado 7.12 Usando una clase de potenciador de token JWT para agregar un campo personalizado**

```
paquete com.thinkmechanix.authentication.security;

//El resto de las importaciones se eliminaron por motivos de concisión.
importar org.springframework.security.oauth2.provider.token.TokenEnhancer;

la clase pública JWTTokenEnhancer implementa TokenEnhancer {
    @autocableado
    OrgUserRepository privado orgUserRepo;

    Cadena privada getOrgId(String nombre de usuario){
        Organización de usuarios orgUser =
            orgUserRepo.findByUserName (nombre de usuario);
        devolver orgUser.getOrganizationId();
    }

    @Anular
    mejora pública de OAuth2AccessToken (
        OAuth2AccessToken accessToken, autenticación
        OAuth2Authentication) {

        Mapa<Cadena, Objeto> información adicional = nuevo HashMap<>();
        String orgId = getOrgId(autenticación.getName());

        adicionallInfo.put("organizationId", orgId);

        ((DefaultOAuth2AccessToken) accessToken)
            .setAdditionalInformation(adicionallInfo); devolver token de acceso;
    }
}
```

Lo último que debe hacer es decirle a su servicio OAuth2 que use su clase JWTToken-Enhancer . Primero necesitas exponer un Spring Bean para tu JWTTokenEnhancer clase. Haga esto agregando una definición de bean a la clase JWTTokenStoreConfig que se definió en el listado 7.8:

```
paquete com.thinkmechanix.authentication.security;

@Configuration
clase pública JWTTokenStoreConfig {
    //El resto de la clase se eliminó por razones de concisión.
    @Frijol
```

```
TokenEnhancer público jwtTokenEnhancer() {
    devolver nuevo JWTTOKENEnhancer();
}
}
```

Una vez que haya expuesto el JWTTOKENEnhancer como un bean, puede conectarlo al Clase JWTOAuth2Config del listado 7.9. Esto se hace en el método configure() de la clase. El siguiente listado muestra la modificación de configure() método de la clase JWTOAuth2Config .

#### Listado 7.13 Conectando su TokenEnhancer

```
paquete com.thinkmechanix.authentication.security;
@Configuration
la clase pública JWTOAuth2Config extiende AuthorizationServerConfigurerAdapter {
    //El resto del código se eliminó por razones de concisión.
    @autocableado
    TokenEnhancer privado jwtTokenEnhancer; ← Cableado automático
    en la clase TokenEnhancer.

    @Anular
    configuración de vacío público (
        AuthorizationServerEndpointsConfigurar puntos finales)
        lanza una excepción {
        TokenEnhancerChain tokenEnhancerChain =
            nueva TokenEnhancerChain(); ← Spring OAuth le permite conectar
            puntos finales
            múltiples tokens
            potenciadores, así que agrega tu
            potenciador de token a
            una clase TokenEnhancerChain.
        tokenEnhancerChain.setTokenEnhancers(
            Arrays.asList(jwtTokenEnhancer, jwtAccessTokenConverter));

        puntos finales.tokenStore(tokenStore)

        .accessTokenConverter(jwtAccessTokenConverter) .tokenEnhancer(tokenEnhancerChain) .authenticationManager(authentication)
        .userDetailsService(usuarioDetailsService);
    }
}
```

Enganche su cadena potenciadora de tokens a  
el parámetro de puntos finales pasado  
en la llamada configure().

En este punto, ha agregado un campo personalizado a su token JWT . La siguiente pregunta que usted debería tener es: "¿Cómo analizo un campo personalizado a partir de un token JWT ?"

#### 7.4.4 Análisis de un campo personalizado a partir de un token de JavaScript

Vamos a recurrir a su puerta de enlace Zuul para ver un ejemplo de cómo analizar una interfaz personalizada. campo en el token JWT . Específicamente, vas a modificar la clase TrackingFilter introdujimos en el capítulo 6 para decodificar el campo OrganizationId fuera del JWT token que fluye a través de la puerta de enlace.

Para hacer esto, obtendrá una biblioteca de analizador JWT y la agregará al servidor Zuul. archivo pom.xml. Hay varios analizadores de tokens disponibles y elegí la biblioteca JJWT

[\(<https://github.com/jwt/jjwt>\)](https://github.com/jwt/jjwt) para hacer el análisis. La dependencia de Maven para el la biblioteca es

```
<dependencia>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.7.0</version>
</dependencia>
```

Una vez que se agrega la biblioteca JJWT , puede agregar un nuevo método a su zuulsrc/src/principal/java/com/thinktmechanix/zuulsrc/filters/TrackingFilter.java clase llamada getOrganizationId(). La siguiente lista muestra este nuevo método.

#### Listado 7.14 Analizando el OrganizationId de su token JWT

```
Cadena privada getOrganizationId(){
    Resultado de cadena="";
    si (filterUtils.getAuthToken() != null){
        Cadena authToken = filterUtils
            .getAuthToken()
            .replace("Portador","");
        intentar {
            Reclamaciones de reclamaciones =
                Jwts.parser()
                    .setSigningKey(
                        configuración de servicio
                        .getJwtSigningKey()
                            .getBytes("UTF-8"))
                    .parseClaimsJws(authToken)
                    .getBody();
            resultado = (Cadena) Claims.get("OrganizationId");
        }
        captura (Excepción e){
            e.printStackTrace();
        }
    }
    resultado de devolución;
}
```

Analice el token del encabezado HTTP de autorización.

Utilice la clase JWTS para analizar el token y pase la clave de firma utilizada para firmar el token.

Extraiga el OrganizationId del token de JavaScript.

Una vez implementada la función getOrganizationId() , agregó un sistema .out.println al método run() en TrackingFilter para imprimir el ID de organización analizado desde su token JWT que fluye a través de la puerta de enlace de Zuul, por lo que llama a cualquier punto final REST habilitado para puerta de enlace . Usé GET <http://localhost:5555/api/licencias/v1/organizaciones/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a>. Recuerda, cuando tú realizar esta llamada, aún necesitas configurar todos los parámetros del formulario HTTP y el HTTP encabezado de autorización para incluir el encabezado de Autorización y el token JWT .

```

zuulserver_1 | 2017-03-31 14:12:07.719 INFO 22 --- [nio-5555-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/]
g FrameworkServlet 'dispatcherServlet'
zuulserver_1 | 2017-03-31 14:12:07.894 INFO 22 --- [nio-5555-exec-2] o.s.c.m.zuul.web.ZuulHandlerMapping
api/organization/**] onto handler of type [class org.springframework.cloud.netflix.zuul.web.ZuulController]
zuulserver_1 | 2017-03-31 14:12:07.895 INFO 22 --- [nio-5555-exec-2] o.s.c.m.zuul.web.ZuulHandlerMapping
api/licensing/**] onto handler of type [class org.springframework.cloud.netflix.zuul.web.ZuulController]
zuulserver_1 | 2017-03-31 14:12:07.895 INFO 22 --- [nio-5555-exec-2] o.s.c.m.zuul.web.ZuulHandlerMapping
api/auth/**] onto handler of type [class org.springframework.cloud.netflix.zuul.web.ZuulController]
zuulserver_1 | 2017-03-31 14:12:08.038 DEBUG 22 --- [nio-5555-exec-2] c.t.zuulsrv.filters.TrackingFilter
generated in tracking filter: 5984dd3e-a155-459b-acab-e6f8be0691cd.
zuulserver_1 | The organization id from the token is : 42d3d4f5-9f33-42f4-8aca-b7519d6af1bb
zuulserver_1 | 2017-03-31 14:12:08.360 DEBUG 22 --- [nio-5555-exec-2] c.t.zuulsrv.filters.TrackingFilter
g request for /api/licensing/v1/organizations/e254f0c-e442-4ebe-a92a-e2fc1d1ff78e/licenses/f13831f8c-c338-4ebe-a92a-e2fc1d1
zuulserver_1 | 2017-03-31 14:12:08.401 INFO 22 --- [nio-5555-exec-2] s.c.a.AnnotationConfigApplicationContext
ingframework.context.annotation.AnnotationConfigApplicationContext@50ed214: startup date [Fri Mar 31 14:12:08 GMT 2017];
mework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@3c09711b
zuulserver_1 | 2017-03-31 14:12:08.460 INFO 22 --- [nio-5555-exec-2] f.a.AutowiredAnnotationBeanPostProcessor

```

Figura 7.12 El servidor Zuul analiza el ID de la organización del token JWT a medida que pasa.

La Figura 7.12 muestra la salida a la consola de línea de comandos que muestra su análisis ID de organización.

## 7.5 Algunas reflexiones finales sobre la seguridad de los microservicios

Si bien este capítulo le ha presentado la especificación OAuth2 y cómo puede utilice la seguridad de Spring Cloud para implementar un servicio de autenticación OAuth2 , OAuth2 es Sólo una pieza del rompecabezas de seguridad de los microservicios. A medida que crea sus microservicios para uso de producción, debe desarrollar la seguridad de sus microservicios en torno a las siguientes prácticas:

- 1 Utilice HTTPS/Capa de sockets seguros (SSL) para todas las comunicaciones de servicio.
- 2 Todas las llamadas de servicio deben pasar por una puerta de enlace API .
- 3 Zona tus servicios en una API pública y una API privada.
- 4 Limite la superficie de ataque de sus microservicios bloqueando las redes innecesarias.

puertos de trabajo.

La figura 7.13 muestra cómo encajan estas diferentes piezas. Cada uno de los elementos con viñetas en la lista se corresponde con los números de la figura 7.13.

Examinemos cada una de las áreas temáticas enumeradas en la lista y los diagramas anteriores. con más detalle.

### UTILICE HTTPS/ CAPA DE ENCHUFES SEGUROS (SSL) PARA TODAS LAS COMUNICACIONES DE SERVICIO

En todos los ejemplos de código de este libro, ha estado usando HTTP porque HTTP es un protocolo simple y no requiere configuración en cada servicio antes de poder comenzar a usar el servicio.

En un entorno de producción, sus microservicios deben comunicarse solo a través de los canales cifrados proporcionados a través de HTTPS y SSL. La configuración y la configuración de HTTPS se puede automatizar a través de sus scripts DevOps.

**NOTA** Si su aplicación necesita cumplir con la industria de tarjetas de pago (PCI) para pagos con tarjeta de crédito, se le pedirá que implemente HTTPS para todos comunicación de servicio. Construir todos sus servicios para usar HTTPS desde el principio es

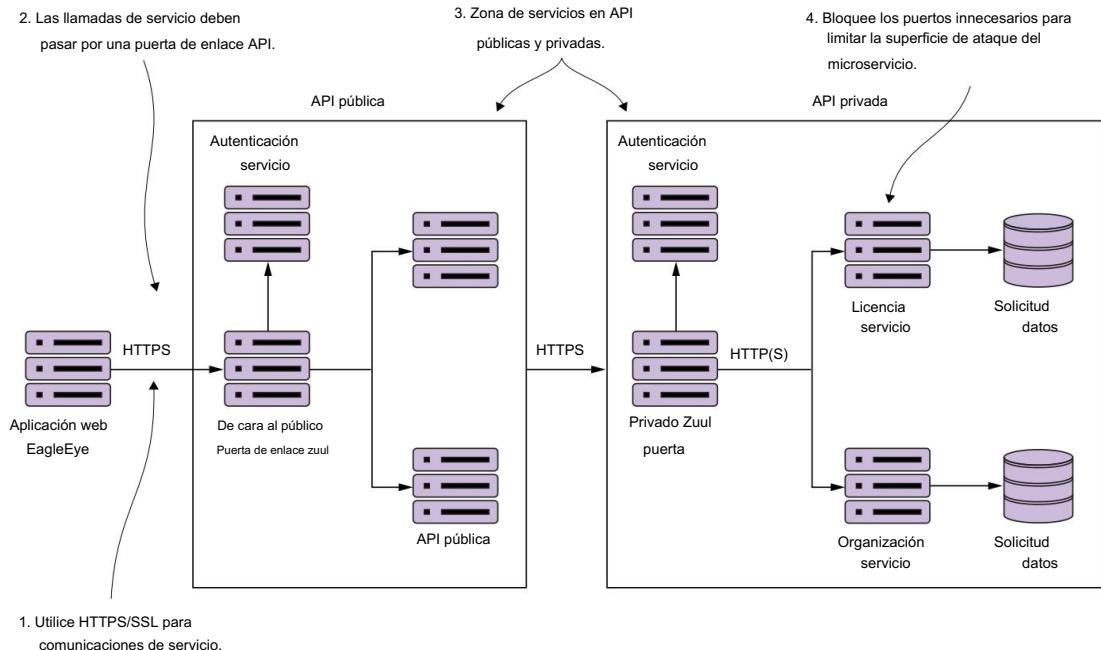


Figura 7.13 Una arquitectura de seguridad de microservicio es más que implementar OAuth2.

mucho más fácil que hacer un proyecto de migración después de su solicitud y Los microservicios están en producción.

#### UTILICE UNA PASARELA DE SERVICIOS PARA ACCEDER A SUS MICROSERVICIOS

Los servidores individuales, los puntos finales de servicio y los puertos en los que se ejecutan sus servicios nunca debe ser directamente accesible al cliente. En su lugar, utilice una puerta de enlace de servicios para actuar como punto de entrada y guardián de sus llamadas de servicio. Configurar la capa de red en el sistema operativo o contenedor en el que se ejecutan sus microservicios para aceptar solo tráfico desde la puerta de enlace de servicios.

Recuerde, la puerta de enlace de servicios puede actuar como un punto de aplicación de políticas (PEP) que puede aplicarse a todos los servicios. Pasar llamadas de servicio a través de una puerta de enlace de servicios como Zuul le permite ser coherente en la forma de proteger y auditar sus servicios. Una puerta de enlace de servicio también le permite bloquear qué puerto y puntos finales está va a exponer al mundo exterior.

#### ZONA SUS SERVICIOS EN UNA API PÚBLICA Y API PRIVADA

La seguridad en general se trata de construir capas de acceso y hacer cumplir el concepto de menor privilegio. El privilegio mínimo es el concepto de que un usuario debe tener el acceso mínimo a la red y los privilegios necesarios para realizar su trabajo diario. Para ello, deberá implementar privilegios mínimos separando sus servicios en dos zonas distintas: pública y privado.

La zona pública contiene las API públicas que consumirán los clientes (aplicación Eagle-Eye). Los microservicios API públicos deben realizar tareas específicas orientadas al flujo de trabajo. Los microservicios API públicos tienden a ser agregadores de servicios y extraen datos. y realizar tareas en múltiples servicios.

Los microservicios públicos deberían estar detrás de su propia puerta de enlace de servicios y tener sus propios Servicio de autenticación propio para realizar la autenticación OAuth2 . Acceso al público Los servicios de las aplicaciones cliente deben pasar por una única ruta protegida por la puerta de enlace de servicios. Además, la zona pública debería contar con su propio servicio de autenticación.

La zona privada actúa como un muro para proteger la funcionalidad principal de su aplicación y datos. La zona privada sólo debería ser accesible a través de un único puerto bien conocido y debe estar bloqueado para aceptar sólo el tráfico de red de la subred de red que el Los servicios privados están funcionando. La zona privada debe tener su propia puerta de enlace de servicios y servicio de autenticación. Los servicios API públicos deben autenticarse en las zonas privadas. servicio de autenticación. Todos los datos de la aplicación deben estar al menos en la subred de red de la zona privada y solo ser accesibles mediante microservicios que residen en la zona privada.

### ¿Qué tan bloqueada debe estar la zona de la red API privada?

Muchas organizaciones adoptan el enfoque de que su modelo de seguridad debería tener una dura centro exterior, con una superficie interior más suave. Lo que esto significa es que una vez que el tráfico esté dentro la zona API privada, la comunicación entre servicios en la zona privada puede ser no cifrado (sin HTTPS) y no requiere un mecanismo de autenticación. La mayoría de tiempo, esto se hace por conveniencia y velocidad del revelador. Cuanta más seguridad tengas implementado, más difícil será depurar problemas, lo que aumentará la complejidad general de administrar su aplicación.

Tiendo a tener una visión paranoica del mundo. (Trabajé en servicios financieros durante ocho años, por lo que la paranoia viene con el territorio). Prefiero compensar la complejidad adicional (que puede mitigarse a través de scripts DevOps) y hacer cumplir que todos los servicios Los archivos que se ejecutan en mi zona API privada utilizan SSL y se autentican con el servicio de autenticación que se ejecuta en la zona privada. La pregunta que tienes que hacerte es: ¿Qué tan dispuesto está a ver a su organización en la portada de su periódico local debido a una violación de la red?

#### LIMITA LA SUPERFICIE DE ATAQUE DE TUS MICROSERVICIOS BLOQUEANDO

#### CERRAR PUERTOS DE RED NO NECESARIOS

Muchos desarrolladores no analizan detenidamente el número mínimo absoluto de puertos. necesitan abrir para que sus servicios funcionen. Configura el sistema operativo de tu El servicio se está ejecutando para permitir solo el acceso entrante y saliente a los puertos necesarios. por su servicio o una pieza de infraestructura necesaria para su servicio (monitoreo, registro agregación).

No se centre únicamente en los puertos de acceso entrantes. Muchos desarrolladores se olvidan de bloquear sus puertos de salida. Bloquear sus puertos de salida puede evitar que los datos filtrarse de su servicio en caso de que el servicio en sí se haya visto comprometido

por un atacante. Además, asegúrese de observar el acceso al puerto de red en sus zonas API públicas y privadas.

## 7.6 Resumen

OAuth2 es un marco de autenticación basado en tokens para autenticar usuarios. OAuth2 garantiza que no sea necesario presentar credenciales de usuario a cada microservicio que lleve a cabo una solicitud de usuario en cada llamada. OAuth2 ofrece diferentes mecanismos para proteger las llamadas a servicios web. Estos Los mecanismos se denominan subvenciones. Para usar OAuth2 en Spring, debe configurar una autenticación basada en OAuth2 servicio. Cada aplicación que quiera llamar a sus servicios debe estar registrada en su servicio de autenticación OAuth2 . Cada aplicación tendrá su propio nombre de aplicación y clave secreta. Las credenciales y roles de usuario están en la memoria o en un almacén de datos y se accede a ellos a través de Seguridad de primavera. Cada servicio debe definir qué acciones puede realizar un rol. Spring Cloud Security admite la especificación JavaScript Web Token (JWT) . JWT define un estándar JavaScript firmado para generar tokens OAuth2 . Con JWT, puede insertar campos personalizados en la especificación. Proteger sus microservicios implica más que simplemente usar OAuth2. Debe Usar HTTPS para cifrar todas las llamadas entre servicios. Utilice una puerta de enlace de servicios para limitar la cantidad de puntos de acceso a través de los cuales se puede acceder a un servicio. Limite la superficie de ataque de un servicio limitando el número de puertos de entrada y salida en el sistema operativo en el que se ejecuta el servicio.

# Arquitectura basada en eventos con Spring Cloud Stream

## Este capítulo cubre

Comprender el procesamiento de la arquitectura basada en eventos y su relevancia para los microservicios.

Uso de Spring Cloud Stream para simplificar el procesamiento de eventos en sus microservicios

Configuración de Spring Cloud Stream

Publicación de mensajes con Spring Cloud Stream y Kafka

Consumir mensajes con Spring Cloud Stream y Kafka

Implementación de almacenamiento en caché distribuido con Spring Cloud Stream, Kafka y Redis

¿Cuándo fue la última vez que te sentaste con otra persona y tuviste una conversación?

Piensa en cómo interactuaste con esa otra persona. ¿Fue totalmente

Intercambio enfocado de información en el que dijiste algo y luego no hiciste nada.

¿Más mientras esperabas que la persona respondiera por completo? ¿Estabas completamente Concéntrate en la conversación y no dejes que nada del mundo exterior te distraiga.

mientras hablabas? Si había más de dos personas en la conversación,

¿Repetiste algo que dijiste perfectamente una y otra vez en cada conversación?

participante y esperar por turno su respuesta? Si respondiste que sí a estas preguntas,

Has alcanzado la iluminación, eres un mejor ser humano que yo y deberías dejar de hacerlo.  
lo que estás haciendo porque ahora puedes responder a la antigua pregunta: "¿Cuál es el  
¿Sonido de un objeto aplaudiendo? Además, sospecho que no tienes hijos.

La realidad es que los seres humanos estamos constantemente en estado de movimiento, interactuando con su entorno que los rodea, mientras envían y reciben información de las cosas que les rodean. En mi casa una conversación típica podría ser algo como este. Estoy ocupado lavando los platos mientras hablo con mi esposa. Le estoy contando sobre mi día. Ella mira su teléfono y escucha, procesa lo que digo y, ocasionalmente, responde. Mientras estoy lavando los platos, escucho una conmoción en el siguiente habitación. Dejo lo que estoy haciendo, corro a la habitación de al lado para descubrir qué pasa y ver que nuestro cachorro bastante grande de nueve meses, Vader, se ha llevado el de mi hijo de tres años zapato, y está trotando por la sala llevando el zapato como si fuera un trofeo. Mi hijo de tres años no está contento con esto. Corro por la casa, persiguiendo al perro hasta que consigo el zapato hacia atrás. Luego vuelvo a los platos y a mi conversación con mi esposa.

Mi punto al decirles esto no es hablarles de un día típico de mi vida, sino más bien señalar que nuestra interacción con el mundo no es sincrónica, lineal y estrechamente definida según un modelo de solicitud-respuesta. Está impulsado por mensajes, donde constantemente enviamos y recibimos mensajes. Cuando recibimos mensajes, reaccionamos a ellos.

mensajes, mientras que a menudo interrumpimos la tarea principal en la que estamos trabajando.

Este capítulo trata sobre cómo diseñar e implementar sus microservicios basados en Spring para comunicarse con otros microservicios mediante mensajes asincrónicos. Usando Los mensajes asincrónicos para comunicarse entre aplicaciones no son nuevos. Lo nuevo es el concepto de utilizar mensajes para comunicar eventos que representan cambios de estado. Este concepto se llama Arquitectura Dirigida por Eventos (EDA). También se le conoce como Mensaje Arquitectura Dirigida (MDA). Lo que le permite hacer un enfoque basado en EDA es construir Sistemas altamente desacoplados que pueden reaccionar a los cambios sin estar estrechamente acoplados. bibliotecas o servicios específicos. Cuando se combina con microservicios, EDA le permite agregar rápidamente nuevas funciones a su aplicación simplemente haciendo que el servicio escuche al flujo de eventos (mensajes) que emite su aplicación.

El proyecto Spring Cloud ha hecho que sea trivial crear soluciones basadas en mensajería a través del subproyecto Spring Cloud Stream. Spring Cloud Stream le permite implementar fácilmente la publicación y el consumo de mensajes, mientras protege sus servicios de los detalles de implementación asociados con la plataforma de mensajería subyacente.

## 8.1 El caso de la mensajería, EDA y los microservicios

¿Por qué es importante la mensajería en la creación de aplicaciones basadas en microservicios? Contestar esa pregunta, comenzemos con un ejemplo. Vamos a utilizar los dos servicios que tenemos. he estado utilizando a lo largo del libro: sus servicios de organización y licencias. Imaginemos que después de implementar estos servicios en producción, descubre que la licencia Las llamadas de servicio tardan muchísimo tiempo al realizar una búsqueda de organización.

información del servicio de organización. Cuando nos fijamos en los patrones de uso de la datos de la organización, encontrará que los datos de la organización rara vez cambian y que la mayoría de Las lecturas de datos del servicio de la organización se realizan mediante la clave principal del registro de la organización. Si pudiera almacenar en caché las lecturas de los datos de la organización sin tener Para incurrir en el costo de acceder a una base de datos, podría mejorar enormemente el tiempo de respuesta. de las llamadas al servicio de licencias.

Al considerar la implementación de una solución de almacenamiento en caché, se da cuenta de que tiene tres requisitos:

- 1 Los datos almacenados en caché deben ser coherentes en todas las instancias del servicio de licencias. significa que no puede almacenar en caché los datos localmente dentro del servicio de licencias porque desea garantizar que se lean los mismos datos de la organización independientemente del instancia de servicio que lo golpea.
- 2 No puede almacenar en caché los datos de la organización dentro de la memoria del contenedor que aloja el Servicio de licencias: el contenedor de tiempo de ejecución que aloja su servicio suele estar restringido. de tamaño y pueden acceder a los datos utilizando diferentes patrones de acceso. Un caché local puede introduce complejidad porque debe garantizar que su caché local esté sincronizado con todos los demás servicios del clúster.
- 3 Cuando un registro de la organización cambia mediante una actualización o eliminación, desea que el servicio de licencias reconozca que ha habido un cambio de estado en el servicio de la organización: el El servicio de licencias debería invalidar cualquier dato almacenado en caché que tenga para ese organización y expulsarla del caché.

Veamos dos enfoques para implementar estos requisitos. La primera  
El enfoque implementará los requisitos anteriores utilizando un modelo de solicitud-respuesta sincrónico. Cuando el  
estado de la organización cambia, la licencia y la organización  
Los servicios se comunican de un lado a otro a través de sus puntos finales REST . El segundo enfoque  
hará que el servicio de la organización emita un evento asíncrono (mensaje) que  
comunicar que los datos del servicio de la organización han cambiado. con el segundo  
En este enfoque, el servicio de la organización publicará un mensaje en una cola indicando que un registro de la  
organización se ha actualizado o eliminado. El servicio de licencias escuchará con el  
intermediario, ver que se ha producido un evento de organización y borrar la organización  
datos de su caché.

### 8.1.1 Uso de un enfoque de solicitud-respuesta sincrónico para comunicar cambios de estado

Para la caché de datos de su organización, utilizará Redis (<http://redis.io/>), una base de datos distribuida de almacenamiento de valores-clave. La Figura 8.1 proporciona una descripción general de alto nivel de cómo Cree una solución de almacenamiento en caché utilizando un modelo de programación de solicitud-respuesta sincrónico tradicional.

En la figura 8.1, cuando un usuario llama al servicio de licencias, el servicio de licencias necesitará para buscar también datos de la organización. El servicio de licencias primero verificará para recuperar el organización deseada por su ID del clúster de Redis. Si el servicio de licencias no puede encontrar los datos de la organización, llamará al servicio de la organización utilizando un punto final basado en REST

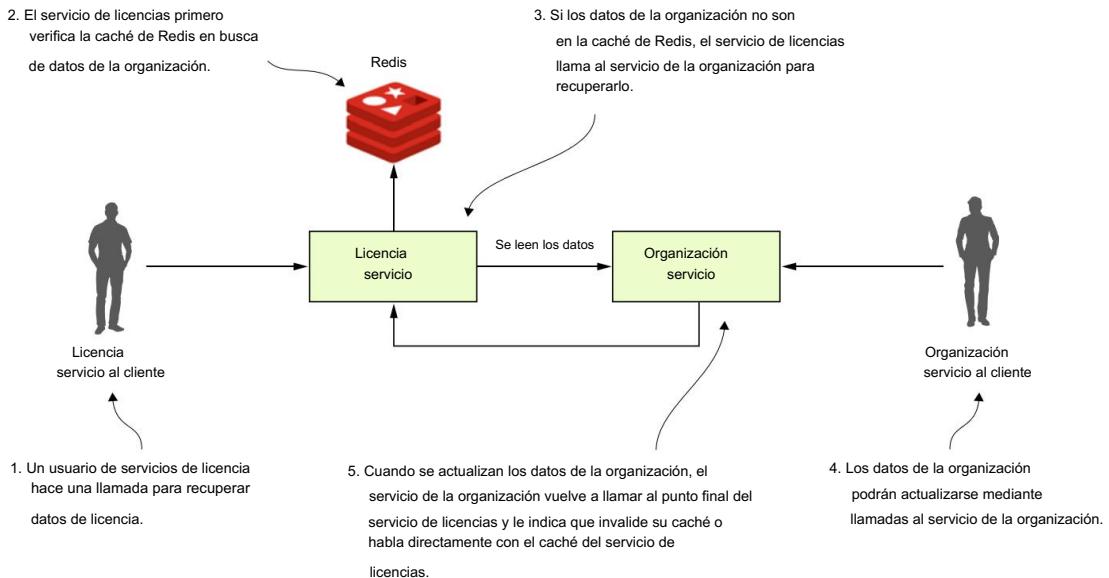


Figura 8.1 En un modelo de solicitud-respuesta sincrónico, los servicios estrechamente acoplados introducen complejidad y fragilidad.

y luego almacenar los datos devueltos en Redis, antes de devolver los datos de la organización de vuelta al usuario. Ahora, si alguien actualiza o elimina el registro de la organización usando punto final REST del servicio de organización , el servicio de organización deberá llamar a un punto final expuesto en el servicio de licencias, indicándole que invalide la organización datos en su caché. En la figura 8.1, si observa dónde devuelve la llamada el servicio de la organización en el servicio de licencias para indicarle que invalide el caché de Redis, puede ver al menos tres problemas:

- 1 Los servicios de organización y concesión de licencias están estrechamente vinculados.
- 2 El acoplamiento ha introducido fragilidad entre los servicios. Si la licencia punto final del servicio para invalidar los cambios de caché, el servicio de la organización tiene que cambiar.
- 3 El enfoque es inflexible porque no se pueden agregar nuevos consumidores de los datos de la organización incluso sin modificar el código en el servicio de la organización para separar que ha llamado al otro servicio para informarle del cambio.

#### ACOPLAMIENTO ESTRECHO ENTRE SERVICIOS

En la figura 8.1 se puede ver un estrecho vínculo entre la concesión de licencias y el servicio de la organización. El servicio de licencias siempre dependió del servicio de la organización para recuperar datos. Sin embargo, al hacer que el servicio de la organización se comunique directamente al servicio de licencias siempre que un registro de la organización haya sido actualizado o eliminado, ha introducido el acoplamiento desde el servicio de organización al servicio de licencias.

Para que los datos en el caché de Redis se invaliden, el servicio de la organización necesita un punto final en el servicio de licencias expuesto al que se pueda llamar para invalidar su caché de Redis, o el servicio de la organización tiene que hablar directamente con el servidor de Redis propiedad del servicio de licencias. para borrar los datos que contiene.

Hacer que el servicio de la organización se comunique con Redis tiene sus propios problemas porque estás hablando con un almacén de datos que pertenece directamente a otro servicio. En un entorno de microservicios, esto es un gran no-no. Si bien se puede argumentar que los datos de la organización pertenecen legítimamente al servicio de la organización, el servicio de licencias los está utilizando en un contexto específico y podría estar potencialmente transformando los datos o haber creado reglas comerciales en torno a ellos. Hacer que el servicio de la organización hable directamente con el servicio de Redis puede infringir accidentalmente las reglas que ha implementado el equipo propietario del servicio de licencias.

#### FRAGILIDAD ENTRE LOS SERVICIOS

El estrecho vínculo entre el servicio de licencias y el servicio de organización también ha introducido fragilidad entre los dos servicios. Si el servicio de licencias no funciona o funciona con lentitud, el servicio de la organización puede verse afectado porque el servicio de la organización ahora se comunica directamente con el servicio de licencias. Nuevamente, si hizo que el servicio de organización se comunicara directamente con el almacén de datos de Redis del servicio de licencias, ahora habrá creado una dependencia entre el servicio de organización y Redis. En este escenario, cualquier problema con el servidor Redis compartido ahora tiene el potencial de desactivar ambos servicios.

#### INFLEXIBLE AL SUMAR NUEVOS CONSUMIDORES A LOS CAMBIOS EN EL SERVICIO DE LA ORGANIZACIÓN

El último problema con esta arquitectura es que es inflexible. Con el modelo de la figura 8.1, si tuviera otro servicio en el que estuviera interesado cuando cambiaron los datos de la organización, necesitaría agregar otra llamada del servicio de la organización a ese otro servicio. Esto significa un cambio de código y una nueva implementación del servicio de la organización. Si utiliza el modelo sincrónico de solicitud-respuesta para comunicar el cambio de estado, comenzará a ver casi un patrón de dependencia similar a una web entre los servicios principales de su aplicación y otros servicios. Los centros de estas webs se convierten en los principales puntos de fallo dentro de su aplicación.

#### Otro tipo de acoplamiento Si

bien la mensajería agrega una capa de indirección entre sus servicios, aún puede introducir un acoplamiento estrecho entre dos servicios mediante la mensajería. Más adelante en este capítulo, enviará mensajes entre la organización y el servicio de licencias.

Estos mensajes se serializarán y deserializarán en un objeto Java utilizando JSON como protocolo de transporte para el mensaje. Los cambios en la estructura del mensaje JSON pueden causar problemas al realizar la conversión a Java si los dos servicios no manejan correctamente diferentes versiones del mismo tipo de mensaje. JSON no admite versiones de forma nativa. Sin embargo, puede utilizar Apache Avro (<https://avro.apache.org/>) si necesita versiones. Avro es un protocolo binario que tiene control de versiones integrado. Spring Cloud Stream es compatible con Apache Avro como protocolo de mensajería.

Sin embargo, el uso de Avro está fuera del alcance de este libro, pero queremos hacerle saber que ayuda si realmente necesita preocuparse por el control de versiones de los mensajes.

### 8.1.2 Uso de mensajería para comunicar cambios de estado entre servicios

Con un enfoque de mensajería, inyectará una cola entre las licencias y servicio de organización. Esta cola no se utilizará para leer datos del servicio de la organización, sino que el servicio de la organización la utilizará para publicar cuando haya algún cambios de estado dentro de los datos de la organización gestionados por el servicio de la organización ocurre. La figura 8.2 demuestra este enfoque.

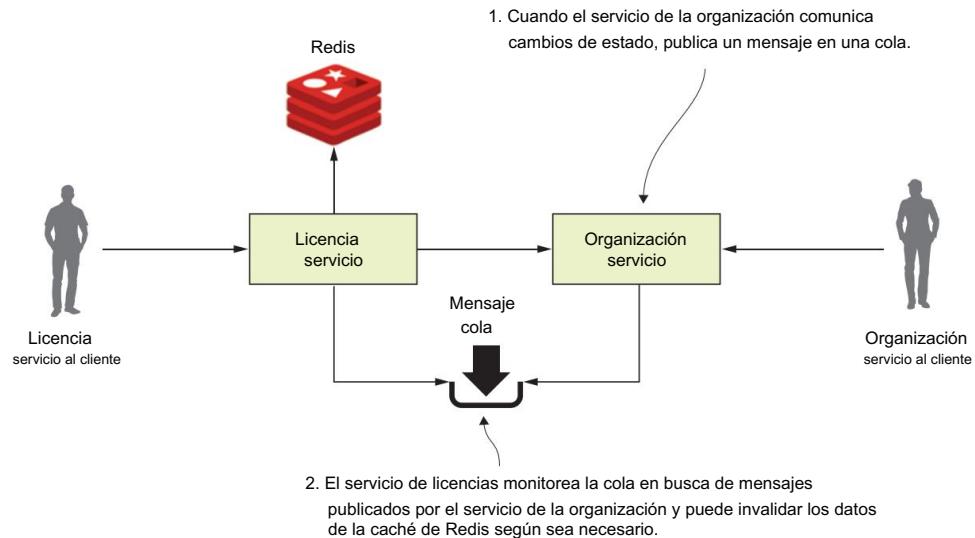


Figura 8.2 A medida que cambia el estado de la organización, los mensajes se escribirán en una cola de mensajes que se encuentra entre los dos servicios.

En el modelo de la figura 8.2, cada vez que cambian los datos de la organización, el servicio de la organización publica un mensaje en una cola. El servicio de licencias está monitoreando el cola de mensajes y, cuando llega un mensaje, borra el registro de la organización correspondiente de la caché de Redis. Cuando se trata de comunicar el estado, la cola de mensajes actúa como intermediaria entre el servicio de licencias y el de la organización.

Este enfoque ofrece cuatro beneficios:

- Acoplamiento flojo
- Durabilidad
- Escalabilidad
- Flexibilidad

#### BAJO ACOPLAMIENTO

Una aplicación de microservicios puede estar compuesta por docenas de servicios pequeños y distribuidos que tienen que interactuar entre sí y están interesados en los datos gestionados por

unos y otros. Como vio con el diseño sincrónico propuesto anteriormente, un sincrónico La respuesta HTTP crea una fuerte dependencia entre la licencia y la organización. servicio. No podemos eliminar estas dependencias por completo, pero podemos intentar minimizarlas. dependencias al exponer solo los puntos finales que administran directamente los datos propiedad del servicio. Un enfoque de mensajería le permite desacoplar los dos servicios porque cuando Cuando se trata de comunicar cambios de estado, ninguno de los servicios conoce al otro. Cuando el servicio de la organización necesita publicar un cambio de estado, escribe un mensaje en un cola. El servicio de licencias sólo sabe que recibe un mensaje; no tiene idea de quien tiene publicó el mensaje.

#### DURABILIDAD

La presencia de la cola le permite garantizar que se entregará un mensaje. incluso si el consumidor del servicio está caído. El servicio de organización puede seguir publicando mensajes incluso si el servicio de licencia no está disponible. Los mensajes serán almacenado en la cola y permanecerá allí hasta que el servicio de licencia esté disponible. Por el contrario, con la combinación de caché y cola, si el servicio de la organización no funciona, el servicio de licencias puede degradarse sin problemas porque al menos parte de los datos de la organización estarán en su caché. A veces, los datos antiguos son mejores que ningún dato.

#### ESCALABILIDAD

Dado que los mensajes se almacenan en una cola, el remitente del mensaje no tiene que esperar para obtener una respuesta del consumidor del mensaje. Pueden seguir su camino y continuar su trabajo. Del mismo modo, si un consumidor que lee un mensaje fuera de la cola no está procesar mensajes lo suficientemente rápido, es una tarea trivial atraer a más consumidores y pídale que procesen esos mensajes fuera de la cola. Este enfoque de escalabilidad encaja bien dentro de un modelo de microservicios porque una de las cosas en las que he estado enfatizando a través de este libro es que debería ser trivial crear nuevas instancias de un microservicio y hacer que ese microservicio adicional se convierta en otro servicio que pueda procesar el trabajo fuera de la cola de mensajes que contiene los mensajes. Este es un ejemplo de escalamiento horizontal. Mecanismos de escala tradicionales para leer mensajes de una cola involucrada aumentar la cantidad de subprocessos que un consumidor de mensajes podría procesar al mismo tiempo. Desafortunadamente, con este enfoque, en última instancia, estaba limitado por la cantidad de CPU disponible para el consumidor del mensaje. Un modelo de microservicio no tiene esta limitación porque está escalando al aumentar la cantidad de máquinas que alojan el servicio y consumen los mensajes.

#### FLEXIBILIDAD

El remitente de un mensaje no tiene idea de quién lo va a consumir. Esto significa que puedes agregar fácilmente nuevos consumidores de mensajes (y nuevas funcionalidades) sin afectar el Servicio de envío original. Este es un concepto extremadamente poderoso porque se pueden agregar nuevas funciones a una aplicación sin tener que tocar los servicios existentes. En cambio, el nuevo código puede escuchar los eventos que se publican y reaccionar ante ellos en consecuencia.

### 8.1.3 Desventajas de una arquitectura de mensajería

Como cualquier modelo arquitectónico, una arquitectura basada en mensajería tiene sus ventajas y desventajas. Una arquitectura basada en mensajería puede ser compleja y requiere que el equipo de desarrollo pague mucha atención a varias cosas clave, incluyendo

Semántica del manejo de mensajes.

Visibilidad del mensaje

Coreografía de mensajes

#### SEMÁNTICA DEL MANEJO DE MENSAJES

El uso de mensajes en una aplicación basada en microservicios requiere más que comprender cómo publicar y consumir mensajes. Requiere que usted entienda cómo su aplicación se comportará según el orden en que se consumen los mensajes y lo que sucede. si un mensaje se procesa fuera de orden. Por ejemplo, si tiene requisitos estrictos que todos los pedidos de un solo cliente deben procesarse en el orden en que se recibidos, tendrá que configurar y estructurar el manejo de sus mensajes de manera diferente que si cada mensaje pudiera consumirse independientemente uno del otro.

También significa que si utiliza mensajes para imponer transiciones de estado estrictas de sus datos, necesita pensar mientras diseña su aplicación en escenarios donde un mensaje genera una excepción o un error se procesa fuera de orden. Si falla un mensaje, ¿vuelve a intentar procesar el error o deja que falle? ¿Cómo te manejas?

¿Mensajes futuros relacionados con ese cliente si uno de los mensajes del cliente falla? De nuevo, Todos estos son temas para pensar.

#### VISIBILIDAD DEL MENSAJE

El uso de mensajes en sus microservicios a menudo significa una combinación de llamadas de servicio sincrónicas. y procesamiento en servicios de forma asíncrona. La naturaleza asíncrona de los mensajes. significa que es posible que no se reciban ni procesen en las proximidades del momento en que se publica o consume el mensaje. Además, tener cosas como un ID de correlación para rastrear un Las transacciones del usuario a través de invocaciones y mensajes de servicios web son fundamentales para comprender y depurar lo que sucede en su aplicación. Como recordarás del capítulo 6, un ID de correlación es un número único que se genera al inicio de una transacción del usuario y se transmite con cada llamada de servicio. También se debe pasar con cada mensaje que se publica y consume.

#### COREOGRAFÍA DEL MENSAJE

Como se mencionó en la sección sobre visibilidad de mensajes, las aplicaciones basadas en mensajería lo hacen más difícil razonar a través de la lógica empresarial de sus aplicaciones porque su código ya no se procesa de forma lineal con un modelo simple de solicitud-respuesta en bloque. En cambio, depurar aplicaciones basadas en mensajes puede implicar vadear a través de los registros de varios servicios diferentes donde las transacciones de los usuarios pueden ejecutarse desordenadamente y en diferentes momentos.

La mensajería puede ser compleja pero poderosa.

Las secciones anteriores no pretendían asustarlo para que no use la mensajería en sus aplicaciones. Más bien, mi objetivo es resaltar que el uso de la mensajería en sus servicios requiere previsión. Recientemente completé un proyecto importante en el que necesitábamos activar y desactivar un conjunto de instancias de servidor AWS con estado para cada uno de nuestros clientes. Tuvimos que integrar una combinación de llamadas y mensajes de microservicio utilizando AWS Simple Queuing Service (SQS) y Kafka. Si bien el proyecto era complejo, vi de primera mano el poder de la mensajería cuando al final del proyecto nos dimos cuenta de que tendríamos que asegurarnos de recuperar ciertos archivos del servidor antes de que se pudiera cerrar el servidor. . Este paso tuvo que realizarse aproximadamente el 75% del flujo de trabajo del usuario y el proceso general no pudo continuar hasta que se completara. Afortunadamente, teníamos un microservicio (llamado nuestro servicio de recuperación de archivos) que podía hacer gran parte del trabajo para verificar y ver si los archivos estaban fuera del servidor que se estaba desmantelando. Debido a que los servidores comunican todos sus cambios de estado (incluido que están siendo desmantelados) a través de eventos, solo tuvimos que conectar el servidor de recuperación de archivos a una secuencia de eventos proveniente del servidor que se está desmantelando y hacer que escuchen un evento de "desmantelamiento". .

Si todo este proceso hubiera sido sincrónico, agregar este paso de drenaje de archivos habría sido extremadamente doloroso. Pero al final, necesitábamos un servicio existente que ya teníamos en producción para escuchar los eventos que salían de una cola de mensajes existente y reaccionar. El trabajo se realizó en un par de días y nunca perdimos el ritmo en la entrega del proyecto. Los mensajes le permiten conectar servicios sin que los servicios estén codificados juntos en un flujo de trabajo basado en código.

## 8.2 Presentación de Spring Cloud Stream

Spring Cloud facilita la integración de mensajes en sus microservicios basados en Spring. Lo hace a través del proyecto Spring Cloud Stream (<https://cloud.spring.io/spring-cloud-stream/> ). El proyecto Spring Cloud Stream es un marco basado en anotaciones que le permite crear fácilmente publicadores y consumidores de mensajes en su aplicación Spring.

Spring Cloud Stream también le permite abstraer los detalles de implementación de la plataforma de mensajería que está utilizando. Se pueden usar múltiples plataformas de mensajes con Spring Cloud Stream (incluido el proyecto Apache Kafka y RabbitMQ) y los detalles específicos de la implementación de la plataforma se mantienen fuera del código de la aplicación. La implementación de la publicación y el consumo de mensajes en su aplicación se realiza a través de interfaces Spring neutrales para la plataforma.

**NOTA** Para este capítulo, utilizará un bus de mensajes liviano llamado Kafka (<https://kafka.apache.org/>). Kafka es un bus de mensajes liviano y de alto rendimiento que le permite enviar secuencias de mensajes de forma asíncrona desde una aplicación a una o más aplicaciones. Escrito en Java, Kafka se ha convertido en el bus de mensajes de facto para muchas aplicaciones basadas en la nube porque es altamente confiable y escalable. Spring Cloud Stream también admite el uso de RabbitMQ como bus de mensajes. Tanto Kafka como RabbitMQ son sólidas plataformas de mensajería y elegí Kafka porque es con lo que estoy más familiarizado.

Para comprender Spring Cloud Stream, comenzemos con una discusión sobre Spring Cloud. Transmite arquitectura y familiarícese con la terminología de Spring Cloud Arroyo. Si nunca antes ha trabajado con una plataforma basada en mensajería, la nueva terminología involucrada puede resultar algo abrumadora.

### 8.2.1 La arquitectura Spring Cloud Stream

Comencemos nuestra discusión mirando la arquitectura Spring Cloud Stream. a través de la lente de dos servicios que se comunican a través de mensajes. Un servicio será el El editor del mensaje y un servicio será el consumidor del mensaje. La figura 8.3 muestra cómo se utiliza Spring Cloud Stream para facilitar el paso de este mensaje.

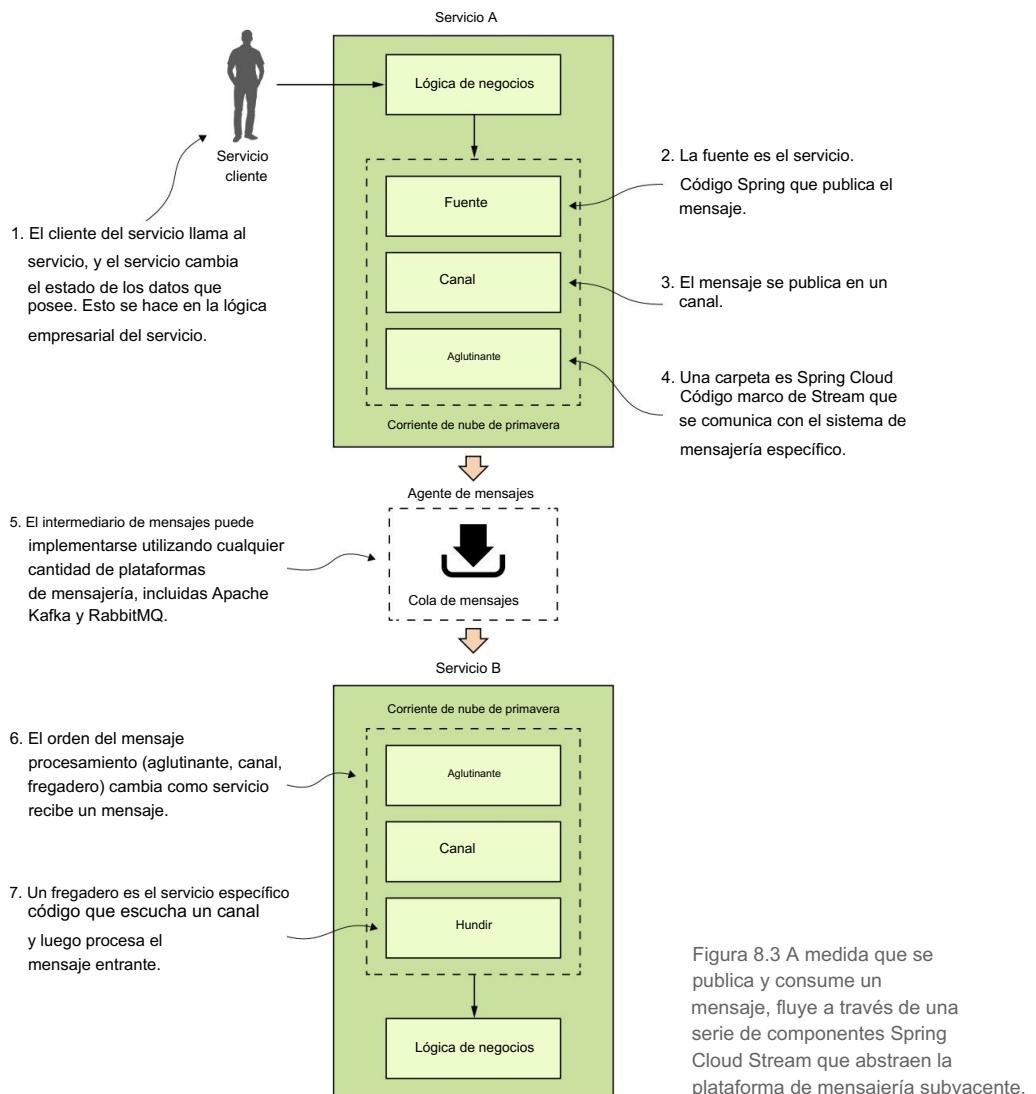


Figura 8.3 A medida que se publica y consume un mensaje, fluye a través de una serie de componentes Spring Cloud Stream que abstracta la plataforma de mensajería subyacente.

Con la publicación y el consumo de un mensaje en Spring Cloud, cuatro componentes participan en la publicación y el consumo del mensaje:

Fuente

Canal

Carpeta

Fregadero

#### FUENTE

Cuando un servicio se prepara para publicar un mensaje, lo publicará utilizando una fuente. Una fuente es una interfaz anotada por Spring que toma un objeto Java antiguo (POJO) que representa el mensaje que se va a publicar. Una fuente toma el mensaje, lo serializa (la serialización predeterminada es JSON) y publica el mensaje en un canal.

#### CANAL

Un canal es una abstracción de la cola que contendrá el mensaje después de que haya sido publicado por el productor del mensaje o consumido por un consumidor del mensaje. Un nombre de canal siempre está asociado con un nombre de cola de destino. Sin embargo, ese nombre de cola nunca se expone directamente al código. En su lugar, el nombre del canal se utiliza en el código, lo que significa que puede cambiar las colas desde las que lee o escribe el canal cambiando la configuración de la aplicación, no el código de la aplicación.

#### AGLUTINANTE

La carpeta es parte del marco Spring Cloud Stream. Es el código Spring el que se comunica con una plataforma de mensajes específica. La parte de carpeta del marco Spring Cloud Stream le permite trabajar con mensajes sin tener que estar expuesto a bibliotecas y API específicas de la plataforma para publicar y consumir mensajes.

#### HUNDIR

En Spring Cloud Stream, cuando un servicio recibe un mensaje de una cola, lo hace a través de un receptor. Un receptor escucha un canal en busca de mensajes entrantes y deserializa el mensaje nuevamente en un antiguo objeto Java. A partir de ahí, el mensaje puede ser procesado por la lógica empresarial del servicio Spring.

### 8.3 Escribir un productor y consumidor de mensajes simples

Ahora que hemos

analizado los componentes básicos de Spring Cloud Stream, veamos un ejemplo simple de Spring Cloud Stream. En el primer ejemplo, pasará un mensaje del servicio de su organización al servicio de licencias. Lo único que hará con el mensaje en el servicio de licencias es imprimir un mensaje de registro en la consola.

Además, debido a que en este ejemplo solo tendrá una fuente de Spring Cloud Stream (el productor de mensajes) y un receptor (consumidor de mensajes), comenzará el ejemplo con algunos atajos simples de Spring Cloud que facilitarán la configuración. La fuente en el servicio de organización y un sumidero en el servicio de licencias trivial.

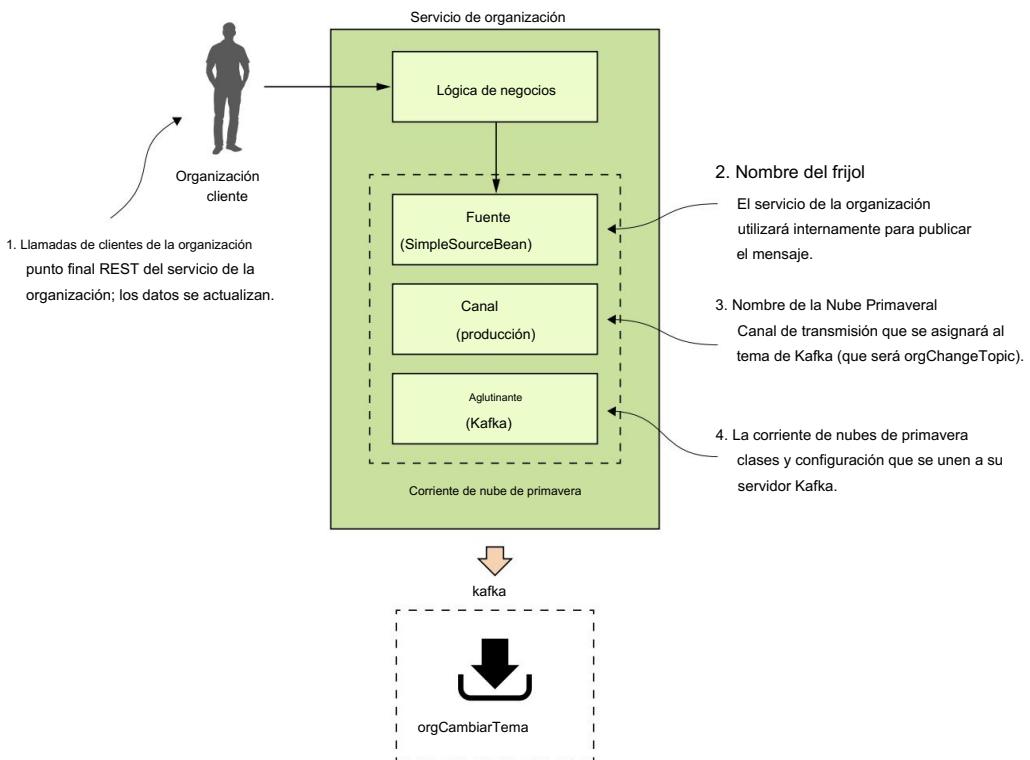


Figura 8.4 Cuando los datos del servicio de la organización cambian, publicará un mensaje en Kafka.

### 8.3.1 Escribir el productor de mensajes en el servicio de organización

Comenzará modificando el servicio de la organización para que cada vez que se agreguen, actualicen o eliminen datos de la organización, el servicio de la organización publique un mensaje en un tema de Kafka indicando que se ha producido el evento de cambio de organización.

La Figura 8.4 destaca el productor de mensajes y se basa en Spring Cloud general.

Arquitectura de flujo de la figura 8.3.

El mensaje publicado incluirá el ID de la organización asociada con el cambio. evento y también incluirá qué acción ocurrió (Agregar, Actualizar o Eliminar).

Lo primero que debe hacer es configurar sus dependencias de Maven en el archivo Maven pom.xml del servicio de la organización. El archivo pom.xml se puede encontrar en el directorio de servicios de la organización. En pom.xml, debe agregar dos dependencias: una para el bibliotecas principales de Spring Cloud Stream y la otra para incluir Spring Cloud Stream Bibliotecas Kafka:

```
<dependencia>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>flujos-nube-primavera</artifactId>
</dependencia>
```

```
<dependencia>
    <groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-
    starter-stream-kafka</artifactId>
</dependencia>
```

Una vez que se hayan definido las dependencias de Maven, debe indicarle a su aplicación que se vinculará a un agente de mensajes Spring Cloud Stream. Para ello, anote la clase de arranque del servicio de organización `organización-servicio/src/main/java/com/thinkmechanix/organization/Application.java` con una anotación `@EnableBinding`. La siguiente lista muestra el código fuente `Application.java` del servicio de la organización .

#### Listado 8.1 La clase Application.java anotada

```
paquete com.thinkmechanix.organization;

importar com.thinkmechanix.organization.utils.UserContextFilter; importar
org.springframework.boot.SpringApplication; importar
org.springframework.boot.autoconfigure.SpringBootApplication; importar
org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker; importar
org.springframework.cloud.netflix.eureka.EnableEurekaClient; importar
org.springframework.cloud.stream.annotation.EnableBinding; importar
org.springframework.cloud.stream.messaging.Source; importar
org.springframework.context.annotation.Bean; importar javax.servlet.Filter;

@SpringBootApplication
@EnableEurekaClient
@EnableCircuitBreaker
@EnableBinding(Source.class) Aplicación
de clase pública {
    @Frijol
    Filtro público userContextFilter() {
        UserContextFilter userContextFilter = nuevo UserContextFilter(); devolver usuarioContextFilter;

    }
}

} public static void main(String[] args)
{ SpringApplication.run(Application.class, args);
}
```

La anotación `@EnableBinding` le dice a Spring Cloud Stream que vincule la aplicación a un intermediario de mensajes.

En el listado 8.1, la anotación `@EnableBinding` le dice a Spring Cloud Stream que desea vincular el servicio a un agente de mensajes. El uso de `Source.class` en la anotación `@EnableBinding` le dice a Spring Cloud Stream que este servicio se comunicará con el intermediario de mensajes a través de un conjunto de canales definidos en la clase `Source` .

Recuerde, los canales se encuentran encima de una cola de mensajes. Spring Cloud Stream tiene un conjunto predeterminado de canales que se pueden configurar para hablar con un agente de mensajes.

En este punto, no le ha dicho a Spring Cloud Stream a qué agente de mensajes desea que se vincule el servicio de la organización. Llegaremos a eso en breve. Ahora puedes seguir adelante e implementar el código que publicará un mensaje.

El código de publicación del mensaje se puede encontrar en Organization-Service/src/com/thinkmechanix/organization/events/source/SimpleSource-Bean.java clase. El siguiente listado muestra el código de esta clase.

#### Listado 8.2 Publicar un mensaje en el intermediario de mensajes

```

paquete com.thinkmechanix.organization.events.source;

//Importaciones eliminadas para mayor concisión

@Component
clase pública SimpleSourceBean {
    fuente fuente privada;

    registrador final estático privado Logger =
        LoggerFactory.getLogger(SimpleSourceBean.class);

    @autocableado
    public SimpleSourceBean(Fuente fuente){ this.source = fuente;
}

public void publicarOrgChange (acción de cadena, ID de organización de cadena) {
    logger.debug("Enviando mensaje Kafka {} para ID de
        organización: {}",

        acción,
        orgId);
    Cambio de OrganizationChangeModel = nuevo OrganizationChangeModel(
        OrganizationChangeModel.class.getTypeName(),
        acción,
        orgId,
        UserContext.getCorrelationId());
    fuente
        .producción()
        .enviar(
            Generador de mensajes
            .withPayload(cambiar)
            .construir());
}
}

El mensaje
para ser publicado
es un POJO de Java.
    
```

Spring Cloud Stream injectará una implementación de interfaz de origen para uso del servicio.

```

    }
}
}

    
```

Cuando esté listo para enviar el mensaje, utilice el método send() de un canal definido en la clase Fuente.

En el listado 8.2, inyecta la clase Spring Cloud Source en su código. Recuerda todo la comunicación a un tema de mensaje específico se produce a través de Spring Cloud Stream construcción llamada canal. Un canal está representado por una clase de interfaz Java. En esto indicando que estás usando la interfaz Fuente . La interfaz de origen es una nube de primavera. interfaz definida que expone un único método llamado salida(). La interfaz Source es una interfaz conveniente para usar cuando su servicio solo necesita publicar en un único canal. El método salida() devuelve una clase de tipo MessageChannel. El MessageChannel es cómo enviará mensajes al intermediario de mensajes. Más adelante en este capítulo, le mostraré cómo exponer múltiples canales de mensajería usando un personalizado interfaz.

La publicación real del mensaje se produce en `PublishOrgChange()`.  
 método. Este método construye un POJO de Java llamado `OrganizationChangeModel`. Soy  
 No voy a poner el código para `OrganizationChangeModel` en el capítulo.  
 porque esta clase no es más que un POJO en torno a tres elementos de datos:

Acción: esta es la acción que desencadenó el evento. He incluido la acción en  
 el mensaje para darle al consumidor del mensaje más contexto sobre cómo debe transmitirse.  
 cesar un evento.

ID de organización : este es el ID de organización asociado con el evento.

ID de correlación : este es el ID de correlación de la llamada de servicio que desencadenó la  
 evento. Siempre debes incluir un ID de correlación en tus eventos, ya que ayuda  
 enormemente con el seguimiento y la depuración del flujo de mensajes a través de sus servicios.

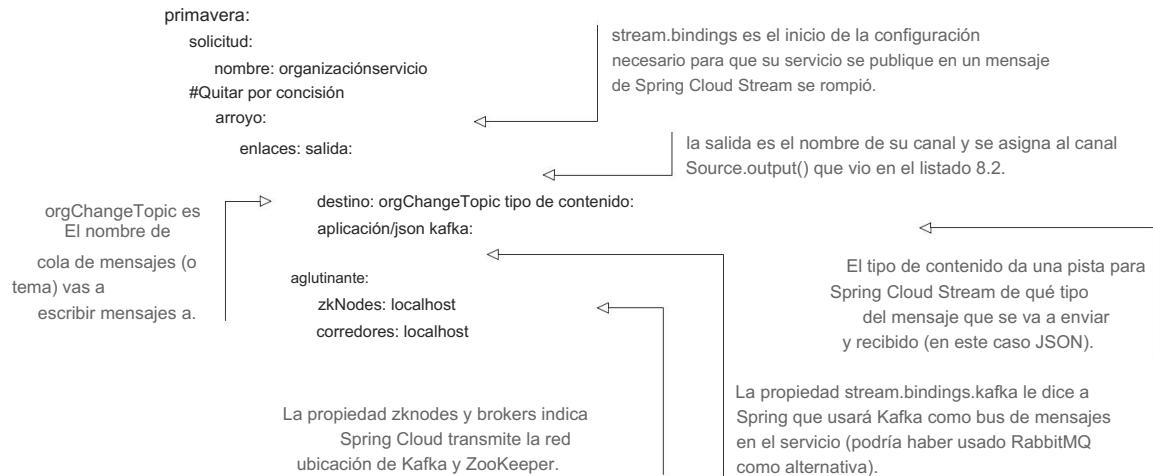
Cuando esté listo para publicar el mensaje, utilice el método `send()` en el  
 Clase `MessageChannel` devuelta por el método `source.output()`.

```
source.output().send(MessageBuilder.withPayload(change).build());
```

El método `send()` toma una clase Spring Message . Usas una clase auxiliar de Spring  
 llamado `MessageBuilder` para tomar el contenido de `OrganizationChangeModel`  
 clase y convertirla en una clase Spring Message .

Este es todo el código que necesitas para enviar un mensaje. Sin embargo, llegado a este punto, todo  
 Debería parecer un poco mágico porque no ha visto cómo vincular el servicio de su organización a una  
 cola de mensajes específica, y mucho menos al intermediario de mensajes real. Esto es todo  
 realizado a través de la configuración. El Listado 8.3 muestra la configuración que realiza el mapeo del  
 Spring Cloud Stream Source de su servicio a un agente de mensajes Kafka y un  
 Tema del mensaje en Kafka. Esta información de configuración se puede localizar en el archivo  
 application.yml de su servicio o dentro de una entrada de Spring Cloud Config para el servicio.

#### Listado 8.3 La configuración de Spring Cloud Stream para publicar un mensaje



La configuración del listado 8.3 parece densa, pero es sencilla. La propiedad de configuración `spring.stream.bindings.output` en el listado asigna el canal `source.output()` en el listado 8.2 al `orgChangeTopic` en el intermediario de mensajes con el que se va a comunicar. También le indica a Spring Cloud Stream que los mensajes que se envían a este tema deben serializarse como JSON. Spring Cloud Stream puede serializar mensajes en múltiples formatos, incluidos JSON, XML y el formato Avro de la Fundación Apache (<https://avro.apache.org/>).

La propiedad de configuración, `spring.stream.bindings.kafka` en el listado 8.3, también le indica a Spring Cloud Stream que vincule el servicio a Kafka. Las subpropiedades le indican a Spring Cloud Stream las direcciones de red de los agentes de mensajes de Kafka y los servidores Apache ZooKeeper que se ejecutan con Kafka.

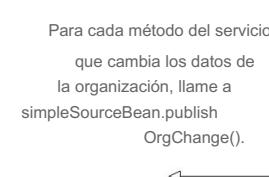
Ahora que tiene el código escrito que publicará un mensaje a través de Spring Cloud Stream y la configuración para decirle a Spring Cloud Stream que usará Kafka como intermediario de mensajes, veamos dónde se realiza la publicación del mensaje en el servicio de su organización. realmente ocurre. Este trabajo se realizará en la clase `organization-servicio/src/main/java/com/thinkmechanix/organization/services/OrganizationService.java`. El siguiente listado muestra el código de esta clase.

#### Listado 8.4 Publicar un mensaje en el servicio de su organización

```
paquete com.thinkmechanix.organization.services;

//Importaciones eliminadas por motivos de conciencia
@Service
Servicio de organización de clase pública {
    @autocableado
    Repositorio de organización privado orgRepository;
    @autocableado
    SimpleSourceBean simpleSourceBean;

    //El resto de la clase se eliminó por motivos de concisión
    public void saveOrg(Organización organización){
        org.setId( UUID.randomUUID().toString());
        orgRepository.save(org);
        simpleSourceBean.publishOrgChange("GUARDAR", org.getId());
    }
}
```



#### ¿Qué datos debo poner en el mensaje?

Una de las preguntas más comunes que recibo de los equipos cuando se embarcan por primera vez en su viaje de mensajes es exactamente cuántos datos deben incluirse en sus mensajes. Mi respuesta es que depende de su aplicación. Como podrá observar, en todos mis ejemplos solo devuelvo el ID de la organización del registro de la organización que ha cambiado. Nunca pongo una copia de los cambios en los datos del mensaje. En mis ejemplos (y en muchos

(continuado)

de los problemas que trato en el espacio de la telefonía), la lógica de negocios que se ejecuta es sensible a los cambios en los datos. Utilicé mensajes basados en eventos del sistema para informar a otros servicios que el estado de los datos ha cambiado, pero siempre fuerzo a los otros servicios a volver al maestro (el servicio propietario de los datos) para recuperar una nueva copia de los datos. Este enfoque es más costoso en términos de tiempo de ejecución, pero también garantiza que siempre tendrá la copia más reciente de los datos para trabajar. Todavía existe la posibilidad de que los datos con los que está trabajando cambien inmediatamente después de haberlos leído desde el sistema fuente, pero eso es mucho menos probable que consumir ciegamente la información directamente de la cola.

Piense detenidamente en la cantidad de datos que está transmitiendo. Tarde o temprano, te encontrarás con una situación en la que los datos que pasaste están obsoletos. Podría estar obsoleto porque un problema hizo que permaneciera demasiado tiempo en una cola de mensajes, o porque un mensaje anterior que contenía datos falló y los datos que está pasando en el mensaje ahora representan datos que están en un estado inconsistente (porque su aplicación se basó en el estado del mensaje en lugar del estado real en el almacén de datos subyacente). Si va a pasar el estado en su mensaje, asegúrese también de incluir una marca de fecha y hora o un número de versión en su mensaje para que los servicios que consumen los datos puedan inspeccionar los datos que se pasan y asegurarse de que no sean anteriores a la copia de los datos que ya tienen. (Recuerde, los datos se pueden recuperar desordenados).

### 8.3.2 Escribir el mensaje del consumidor en el servicio de licencia

En este punto, ha modificado el servicio de la organización para publicar un mensaje en Kafka cada vez que el servicio de la organización cambia los datos de la organización. Cualquier persona interesada puede reaccionar sin necesidad de ser llamado explícitamente por el servicio de organización. También significa que puede agregar fácilmente nuevas funciones que pueden reaccionar a los cambios en el servicio de la organización haciendo que escuchen los mensajes que llegan a la cola de mensajes. Ahora cambiemos de dirección y veamos cómo un servicio puede consumir un mensaje usando Spring Cloud Stream.

En este ejemplo, el servicio de licencias consumirá el mensaje publicado por el servicio de la organización. La Figura 8.5 muestra dónde encajará el servicio de licencias en la arquitectura Spring Cloud que se muestra por primera vez en la figura 8.3.

Para comenzar, nuevamente debe agregar sus dependencias de Spring Cloud Stream al archivo pom.xml de servicios de licencia. Este archivo pom.xml se puede encontrar en el directorio de servicios de licencia del código fuente del libro. De manera similar al archivo pom.xml del servicio de organización que vio anteriormente, agrega las siguientes dos entradas de dependencia :

```
<dependency>
    <groupId>org.springframework.cloud</groupId> <artifactId>spring-
    cloud-stream</artifactId> </dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId> <artifactId>spring-
    cloud-starter-stream-kafka</artifactId>
</dependency>
```

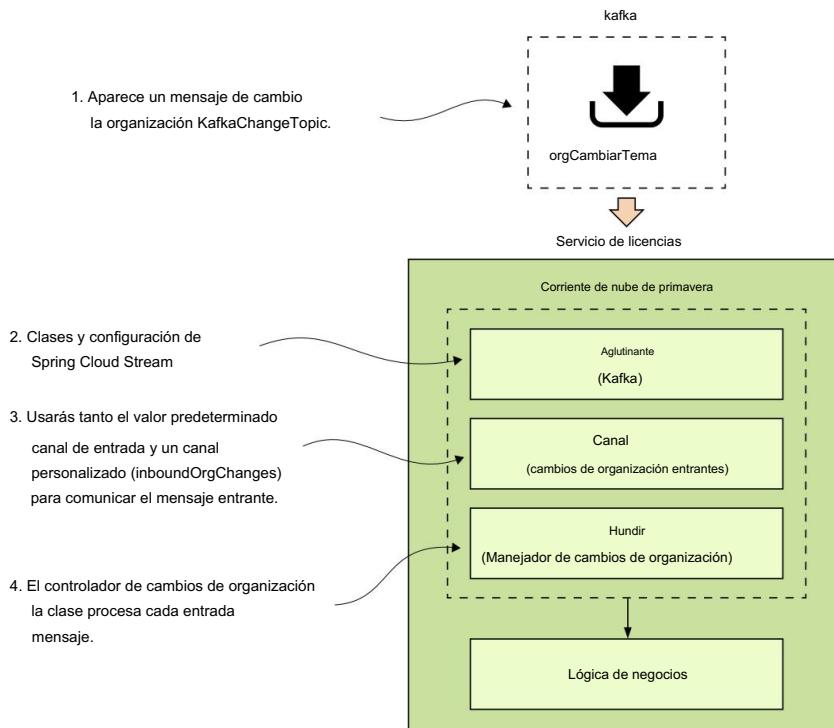


Figura 8.5 Cuando llega un mensaje a Kafka orgChangeTopic, el servicio de licencias responderá.

Luego debe informarle al servicio de licencias que necesita usar Spring Cloud Stream para vincularse a un intermediario de mensajes. Al igual que el servicio de organización, vamos a anotar el Clase de arranque de servicios de licencia (licensing-service/src/main/java/com/thinkmechanix/licenses/Application.java) con @EnableBinding anotación. La diferencia entre el servicio de licencias y el servicio de organización. es el valor que va a pasar a la anotación @EnableBinding , como se muestra en la siguiente lista.

#### Listado 8.5 Consumir un mensaje usando Spring Cloud Stream

```
paquete com.thinkmechanix.licenses;
//Importaciones eliminadas por razones de concisión
@EnableBinding(Sink.class) public
class Application { //Código eliminado
    para mayor concisión
    @StreamListener(Sink.INPUT)
    registrar de vacío públicoSink(
        OrganizationChangeModel orgChange) {
```

La anotación @EnableBinding le dice al servicio que use los canales definidos en el receptor.

Interfaz para escuchar los mensajes entrantes.

Spring Cloud Stream ejecutará este método cada vez que se reciba un mensaje desde el canal de entrada.

```

        logger.debug("Recibió un evento para ID de organización
        {},
        orgChange.getOrganizationId());
    }

}

```

Debido a que el servicio de licencias es un consumidor de un mensaje, usted pasará el `@EnableBinding` anota el valor `Sink.class`. Esto le dice a Spring Cloud Stream para vincularse a un intermediario de mensajes utilizando la interfaz Spring Sink predeterminada . Similar a Interfaz Spring Cloud Steam Source descrita en la sección 8.3.1, Spring Cloud Stream expone un canal predeterminado en la interfaz Sink . El canal en el fregadero . La interfaz se llama entrada y se utiliza para escuchar los mensajes entrantes en un canal.

Una vez que haya definido que desea escuchar mensajes a través de `@EnableBinding` anotación, puede escribir el código para procesar un mensaje que sale de la entrada del receptor canal. Para hacer esto, use la anotación Spring Cloud Stream `@StreamListener` .

La anotación `@StreamListener` le dice a Spring Cloud Stream que ejecute el método `loggerSink()` cada vez que se recibe un mensaje fuera del canal de entrada . Spring Cloud Stream deserializará automáticamente el mensaje que sale del canal a un POJO de Java llamado `OrganizationChangeModel`.

Una vez más, la asignación real del tema del intermediario de mensajes al canal de entrada se realiza en la configuración del servicio de licencia. Para el servicio de licencias, su configuración se muestra en el siguiente listado y se puede encontrar en la página del servicio de licencias. archivo `licensing-service/src/main/resources/application.yml`.

#### Listado 8.6 Asignación del servicio de licencias a un tema de mensaje en Kafka

```

primavera:
  solicitud:
    nombre: servicio de licencias
    ... #Retirar por conciencia
  nube:
    arroyo:
      fijaciones:
        aporte:
          destino: orgCambiarTema
          tipo de contenido: aplicación/json
          grupo: grupo de licencias
          aglutinante:
            zkNodes: localhost
            corredores: localhost

```

La propiedad `spring.cloud.stream.bindings.input` asigna el canal de entrada a la cola `orgChangeTopic`.

La propiedad de grupo se utiliza para garantizar la semántica de proceso único para un servicio.

La configuración en este listado se parece a la configuración del servicio de organización. Sin embargo, tiene dos diferencias clave. Primero, ahora tienes un canal llamado entrada. definido en la propiedad `spring.cloud.stream.bindings` . Este valor se asigna a el canal `Sink.INPUT` definido en el código del listado 8.5. Esta propiedad mapea la canal de entrada al `orgChangeTopic`. En segundo lugar, se ve la introducción de un nuevo

propiedad llamada `spring.cloud.stream.bindings.input.group`. El grupo

La propiedad define el nombre del grupo de consumidores que consumirá el mensaje.

El concepto de grupo de consumidores es el siguiente: es posible que tenga varios servicios con cada servicio tiene varias instancias escuchando la misma cola de mensajes. Quieres cada servicio único para procesar una copia de un mensaje, pero usted solo desea un servicio instancia dentro de un grupo de instancias de servicio para consumir y procesar un mensaje. La propiedad del grupo identifica el grupo de consumidores al que pertenece el servicio. Mientras todas las instancias de servicio tienen el mismo nombre de grupo, Spring Cloud Stream y el intermediario de mensajes subyacente garantizará que solo se enviará una copia del mensaje. El consumido por una instancia de servicio que pertenece a ese grupo. En el caso de su licencia servicio, el valor de la propiedad del grupo se llamará `licensingGroup`.

La figura 8.6 ilustra cómo se utiliza el grupo de consumidores para ayudar a hacer cumplir las normas de consumo. una vez la semántica de un mensaje que se consume en múltiples servicios.

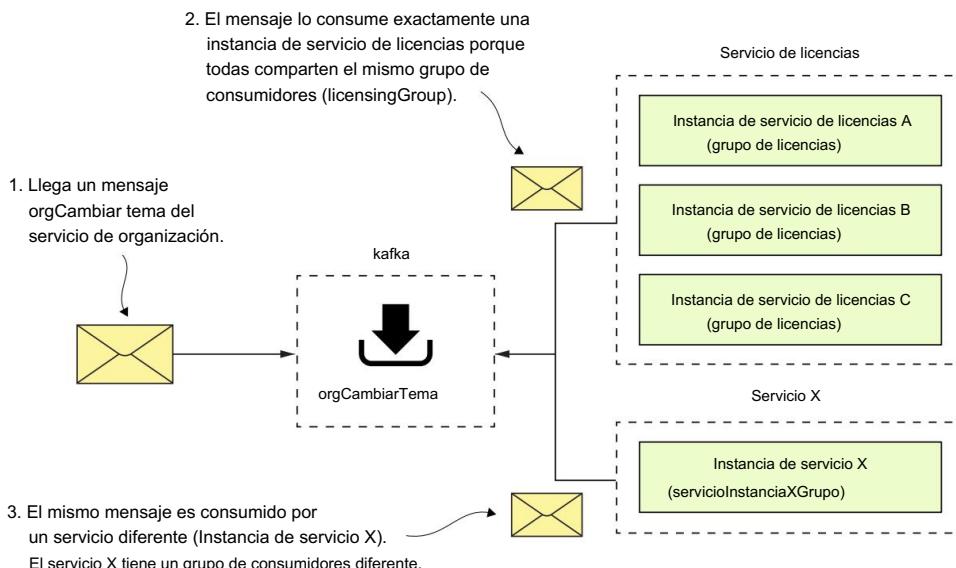


Figura 8.6 El grupo de consumidores garantiza que un mensaje solo será procesado una vez por un grupo de instancias de servicio.

### 8.3.3 Ver el servicio de mensajes en acción

En este punto, el servicio de la organización publica un mensaje en `org-ChangeTopic` cada vez que se agrega, actualiza o elimina un registro y la licencia servicio que recibe el mensaje del mismo tema. Ahora verás este código en acción al actualizar un registro de servicio de la organización y mirar la consola para ver aparecer el mensaje de registro correspondiente del servicio de licencias.

Para actualizar el registro de servicio de la organización, emitirá un PUT en el servicio de la organización para actualizar el número de teléfono de contacto de la organización. El punto final con el que vas a actualizar es `http://localhost:5555/api/organization/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a`. El cuerpo que vas a enviar al

La llamada PUT al punto final es

```
{
    "contactEmail": "mark.balster@custcrmco.com",
    "contactName": "Mark Balster",
    "contactPhone": "823-555-2222",
    "id": "e254f8c-c442-4ebe-a82a-e2fc1d1ff78a",
    "nombre": "cliente-crm-co"
}
```

La Figura 8.7 muestra el resultado devuelto por esta llamada PUT .

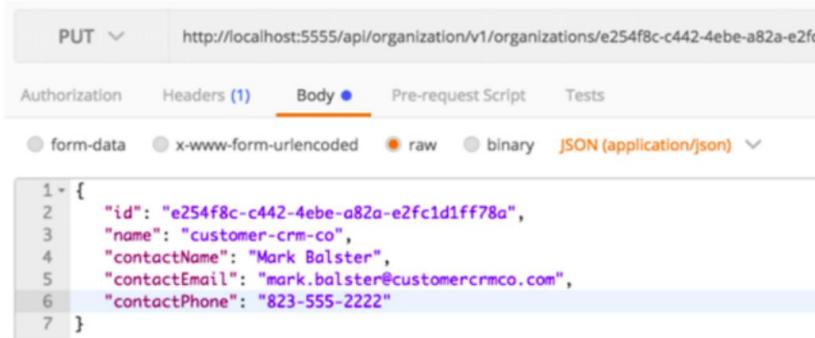


Figura 8.7 Actualización del número de teléfono de contacto mediante el servicio de la organización

Una vez que se haya realizado la llamada al servicio de la organización, debería ver el siguiente resultado en la ventana de la consola que ejecuta los servicios. La Figura 8.8 muestra este resultado.

Mensaje de registro del servicio de la organización que indica que envió un mensaje de Kafka

*↑*

Sending Kafka message UPDATE for Organization Id: e254f8c-c442-4e

Adding the correlation id to the outbound headers.

Completing outgoing request for /api/organization/v1/organization

Received a message of type com.thoughtmechanix.organization.event

Received a UPDATE event from the organization service for organiz

*↑*

Mensaje de registro del servicio de licencias que indica que Recibí un mensaje para un evento de ACTUALIZACIÓN.

Figura 8.8 La consola muestra el mensaje del servicio de la organización que se envía y luego se recibe.

Ahora tienes dos servicios que se comunican entre sí mediante mensajería. Primavera Cloud Stream actúa como intermediario para estos servicios. Desde el punto de vista de la mensajería, los servicios no saben nada entre sí. Están utilizando un corredor de mensajería para comunicarse como intermediario y Spring Cloud Stream como abstracción. capa sobre el intermediario de mensajería.

#### 8.4 Un caso de uso de Spring Cloud Stream: almacenamiento en caché distribuido

En este punto tienes dos servicios comunicándose con mensajería, pero no estás realmente haciendo algo con los mensajes. Ahora crearás el almacenamiento en caché distribuido. ejemplo que analizamos anteriormente en el capítulo. Tendrás el servicio de licencias siempre comprobar una caché de Redis distribuida para los datos de la organización asociados con un determinado licencia. Si los datos de la organización existen en el caché, devolverá los datos del cache. Si no es así, llamará al servicio de la organización y almacenará en caché los resultados de la llamada. en un hash de Redis.

Cuando los datos se actualizan en el servicio de la organización, el servicio de la organización enviar un mensaje a Kafka. El servicio de licencias recogerá el mensaje y emitirá un eliminar contra Redis para borrar el caché.

##### Almacenamiento en caché y mensajería en la nube

El uso de Redis como caché distribuido es muy relevante para el desarrollo de microservicios en la nube. Con mi empleador actual, desarrollamos nuestra solución utilizando los servicios web de Amazon (AWS) y somos un gran usuario de DynamoDB de Amazon. También utilizamos Amazon ElastiCache (Redis) para

Mejorar el rendimiento para la búsqueda de datos comunes: mediante el uso de un caché, Hemos mejorado significativamente el rendimiento de varios de nuestros servicios clave. Todas las tablas de los productos que vendemos son multiinquilino (mantienen múltiples clientes registros en una sola tabla), lo que significa que pueden ser bastante grandes. Porque el almacenamiento en caché tiende a fijar datos "muy usados"; hemos visto un rendimiento significativo Mejoras mediante el uso de Redis y el almacenamiento en caché para evitar las lecturas en Dynamo.

Reducir la carga (y el costo) de las tablas de Dynamo que contienen nuestros datos: acceder Los datos en Dynamo pueden ser una propuesta costosa. Cada lectura que realices es un evento imputable. Usar un servidor Redis es significativamente más barato para lecturas mediante una clave principal que una lectura de Dynamo.

Aumentar la resiliencia para que nuestros servicios puedan degradarse sin problemas si nuestro principal El almacén de datos (Dynamo) tiene problemas de rendimiento: si AWS Dynamo tiene problemas (sucede ocasionalmente), el uso de un caché como Redis puede ayudar a que su servicio se degrade con gracia. Dependiendo de la cantidad de datos que guardes en su caché, una solución de almacenamiento en caché puede ayudar a reducir la cantidad de errores que obtener al acceder a su almacén de datos.

Redis es mucho más que una solución de almacenamiento en caché, pero puede cumplir esa función si necesita un caché distribuido.

#### 8.4.1 Uso de Redis para almacenar en caché las búsquedas

Ahora comenzará configurando el servicio de licencias para usar Redis. Afortunadamente, Spring Data ya simplifica la introducción de Redis en su servicio de licencias. Para utilizar Redis en el servicio de licencias, debe hacer cuatro cosas:

- 1 Configure el servicio de licencias para incluir las dependencias de Spring Data Redis
- 2 Construya una conexión de base de datos a Redis
- 3 Defina los repositorios de Spring Data Redis que usará su código para interactuar con un hash de Redis.
- 4 Utilice Redis y el servicio de licencias para almacenar y leer datos de la organización

#### CONFIGURAR EL SERVICIO DE LICENCIAS CON DEPENDENCIAS DE REDIS DE SPRING DATA

Lo primero que debe hacer es incluir las dependencias `spring-data-redis`, junto con las dependencias `jedis` y `common-pools2`, en el archivo `pom.xml` del servicio de licencias. Las dependencias a incluir se muestran en el siguiente listado.

#### Listado 8.7 Agregar las dependencias de Spring Redis

```
<dependencia>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-redis</artifactId>
    <versión>1.7.4.RELEASE</versión> </
dependencia>

<dependencia>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId> <versión>2.9.0</
    versión>
</dependencia>

<dependencia>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-pool2</artifactId> <versión>2.0</
    versión> </dependencia>
```

#### CONSTRUCCIÓN DE LA CONEXIÓN DE LA BASE DE DATOS A UN SERVIDOR REDIS

Ahora que tiene las dependencias en Maven, necesita establecer una conexión con su servidor Redis. Spring utiliza el proyecto de código abierto Jedis (<https://github.com/xetorthio/jedis>) para comunicarse con un servidor Redis. Para comunicarse con una instancia de Redis específica, expondrá una `JedisConnection-Factory` en la clase `licensing-service/src/main/java/com/thinktmechanix/licenses/Application.java` como un Spring Bean. Una vez que tenga una conexión a Redis, usará esa conexión para crear un objeto `Spring RedisTemplate`. El objeto `RedisTemplate` será utilizado por las clases del repositorio de Spring Data que implementará en breve para ejecutar las consultas y guardar los datos del servicio de la organización en su servicio Redis. La siguiente lista muestra este código.

Listado 8.8 Estableciendo cómo su servicio de licencias se comunicará con Redis

```

paquete com.thinkmechanix.licenses;

// La mayoría de las importaciones se han eliminado por razones de concisión
import
    org.springframework.data.redis.connection.jedis.JedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;

@SpringBootApplication
@EnableEurekaClient
@EnableCircuitBreaker
@EnableBinding(Sink.class) aplicación
de clase pública {

    @Autowired
    serviceConfig privado serviceConfig;

    //Todos los demás métodos de la clase se han eliminado por razones de coherencia.
    @Frijol
    pública JedisConnectionFactory jedisConnectionFactory() {
        JedisConnectionFactory jedisConnFactory = nueva
        JedisConnectionFactory();
        jedisConnFactory.setHostName( serviceConfig.getRedisServer() );
        jedisConnFactory.setPort( serviceConfig.getRedisPort() ); devolver jedisConnFactory;

    }
    @Frijol
    public RedisTemplate<Cadena, Objeto> redisTemplate()
    { RedisTemplate<Cadena, Objeto> plantilla = new RedisTemplate<Cadena, Objeto>();

        plantilla.setConnectionFactory(jedisConnectionFactory()); plantilla de devolución;

    }
}

```

El trabajo fundamental para configurar el servicio de licencias para comunicarse con Redis está completo. Pasemos ahora a escribir la lógica que obtendrá, agregará, actualizará y eliminará datos de Redis.

#### DEFINICIÓN DE LOS REPOSITORIOS REDIS DE DATOS DE SPRING

Redis es un almacén de datos de valores clave que actúa como un Hash-Map grande, distribuido y en memoria. En el caso más simple, almacena datos y los busca mediante una clave. No tiene ningún tipo de lenguaje de consulta sofisticado para recuperar datos. Su simplicidad es su punto fuerte y una de las razones por las que tantos proyectos lo han adoptado para su uso en sus proyectos.

Debido a que estás utilizando Spring Data para acceder a tu tienda Redis, necesitas definir una clase de repositorio. Como recordará desde el principio del capítulo 2, Spring Data utiliza clases de repositorio definidas por el usuario para proporcionar un mecanismo simple para que una clase Java acceda a su base de datos Postgres sin tener que escribir consultas SQL de bajo nivel .

Para el servicio de licencias, definirá dos archivos para su repositorio de Redis.

El primer archivo que escribirás será una interfaz Java que se inyectará en cualquier

de las clases de servicios de licencia que necesitarán para acceder a Redis. Esta interfaz (`licensing-service/src/main/java/com/thinkmechanix/licenses/repository/OrganizationRedisRepository.java`) se muestra en el siguiente listado.

**Listado 8.9 OrganizationRedisRepository define los métodos utilizados para llamar a Redis**

```
paquete com.thinkmechanix.licenses.repository;
importar com.thinkmechanix.licenses.model.Organization;

interfaz pública OrganizationRedisRepository { void
    saveOrganization(Organización organización); void
    updateOrganization(Organización de la organización); void
    eliminarOrganización(String organizaciónId);
    Organización findOrganization(String organizaciónId);
}
```

El segundo archivo es la implementación de la interfaz `OrganizationRedisRepository`. La implementación de la interfaz, la clase `licensing-service/src/main/java/com/thinkmechanix/licenses/repository/OrganizationRedisRepositoryImpl.java`, utiliza el bean `RedisTemplate` Spring que declaró anteriormente en el listado 8.8 para interactuar con el servidor Redis y realizar acciones contra el servidor Redis. La siguiente lista muestra este código en uso.

**Listado 8.10 La implementación de OrganizationRedisRepositoryImpl**

```
paquete com.thinkmechanix.licenses.repository;

// La mayoría de las importaciones se eliminaron para una
importación concisa org.springframework.data.redis.core.HashOperations; importar
org.springframework.data.redis.core.RedisTemplate;

@Repository
clase pública OrganizationRedisRepositoryImpl implementa
    OrganizationRedisRepository {
    Cadena final estática privada HASH_NAME="organización";

    redisTemplate privado<Cadena, Organización> redisTemplate; HashOperations
    privadas hashOperations;

    Organización públicaOrganizationRedisRepositoryImpl(){ super();

    }

    @Autowired
    organización privadaOrganizationRedisRepositoryImpl(RedisTemplate redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    @PostConstruct
    inicio vacío privado() {
```

La anotación  
`@Repository` le dice a  
 Spring que esta clase es una  
 Clase de repositorio  
 utilizada con Spring Data.

El nombre del hash en su  
 servidor Redis donde se  
 almacenan los datos  
 de la organización.

```

        hashOperations = redisTemplate.opsForHash();
    }

    @Anular
    public void saveOrganization(Organización de la organización) {
        hashOperations.put(HASH_NAME, org.getId(), org);
    }

    @Anular
    organización de actualización pública vacía (organización de la organización) {
        hashOperations.put(HASH_NAME, org.getId(), org);
    }

    @Anular
    public void eliminarOrganización(String organizaciónId) { hashOperations.delete(HASH_NAME,
        organizaciónId);
    }

    @Anular
    Organización pública findOrganization (String organizaciónId) {
        return (Organización) hashOperations.get(HASH_NAME, OrganizationId);
    }
}

```

Todas las interacciones con Redis  
estar con un único objeto de  
organización almacenado por su  
clave.

OrganizationRedisRepositoryImpl contiene toda la lógica CRUD (Crear, Leer, Actualizar, Eliminar) utilizada para almacenar y recuperar datos de Redis. Hay dos cosas clave a tener en cuenta en el código del listado 8.10: Todos los datos en Redis

se almacenan y recuperan mediante una clave. Dado que está almacenando datos recuperados del servicio de la organización, el ID de la organización es la elección natural para la clave que se utiliza para almacenar un registro de la organización.

La segunda cosa a tener en cuenta es que un servidor Redis puede contener múltiples hashes y estructuras de datos en su interior. En cada operación contra el servidor Redis, debe decirle a Redis el nombre de la estructura de datos contra la que está realizando la operación. En el listado 8.10, el nombre de la estructura de datos que está utilizando se almacena en la constante HASH\_NAME y se denomina "organización".

#### USO DE REDIS Y EL SERVICIO DE LICENCIA PARA ALMACENAR Y LEER DATOS DE LA ORGANIZACIÓN

Ahora que tiene el código implementado para realizar operaciones en Redis, puede modificar su servicio de licencias para que cada vez que el servicio de licencias necesite los datos de la organización, verifique el caché de Redis antes de llamar al servicio de la organización.

La lógica para verificar Redis se producirá en la clase licensing-service/src/main/java/com/thinkmechanix/licenses/clients/OrganizationRestTemplate Client.java . El código para esta clase se muestra en el siguiente listado.

Listado 8.11 La clase OrganizationRestTemplateClient implementará la lógica de caché

```
paquete com.thinkmechanix.licenses.clients;
```

```
//Importaciones eliminadas por razones de concisión
@Component
```

```

clase pública OrganizationRestTemplateClient {
    @autocableado
    RestTemplate restTemplate;

    @autocableado
    OrganizaciónRedisRepository orgRedisRepo; ← La clase OrganizationRedisRepository se
                                                conecta automáticamente en
                                                OrganizationRestTemplateClient.

    registrador final estático privado registrador =
        LoggerFactory.getLogger(OrganizationRestTemplateClient.clase);

    verificación de organización privadaRedisCache(
        String organizaciónId) { intentar { ← Tratando de recuperar un
                                                Clase de organización con su ID
                                                de organización de Redis
            return orgRedisRepo.findOrganization(organizationId);
        }
        captura (Excepción ex){
            logger.error("Se produjo un error al intentar
                recuperar la organización {} verificar Redis Cache. Excepción
                {}", OrganizationId, ex);
            devolver nulo;
        }
    }

Private void cacheOrganizationObject(Organización de la organización) {
    intentar {
        orgRedisRepo.saveOrganization(org);
    }captura (excepción ex){
        logger.error("No se puede almacenar en caché la organización {} en Redis.
            Excepción {} org.getId(), ex);
    }
}

Organización pública getOrganization (String organizaciónId) {
    logger.debug("En el servicio de licencias
        .getOrganization: {}",
        UserContext.getCorrelationId());

    Organización de la organización = checkRedisCache(organizationId); ← Si no puede recuperar datos
                                                                de Redis, llamará al servicio de
                                                                la organización para recuperar
                                                                los datos de
                                                                la base de datos de origen.

    if (org!=null)
        { logger.debug("He recuperado exitosamente una
            organización {} del caché de Redis: {}",
            OrganizationId, org);
        organización de retorno;
    }

    logger.debug("No se puede localizar
        organización desde caché de
        redis: {}.", OrganizationId);

    ResponseEntity<Organización> restExchange =
        restoTemplate.exchange(
            "http://zuulservice/api/organización
            /v1/organizaciones/{organizationId}",
            Método Http.GET,
            nulo,

```

```

Organización.clase,
organizaciónId);

/*Guardar el registro del caché*/
org = restExchange.getBody();
if (org!=null)
    { cacheOrganizationObject(org);
}

organización de retorno;
}
}

```

Guarda el objeto recuperado en la caché

El método `getOrganization()` es donde toma la llamada al servicio de la organización. lugar. Antes de realizar la llamada REST real , intenta recuperar el objeto de organización asociado con la llamada desde Redis usando `checkRedisCache()`. método. Si el objeto de la organización en cuestión no está en Redis, el código devolverá un valor nulo . Si se devuelve un valor nulo del método `checkRedisCache()` , el El código invocará el punto final REST del servicio de la organización para recuperar el registro de la organización deseado. Si el servicio de organización devuelve una organización, el objeto de organización devuelto se almacenará en caché utilizando el método `cacheOrganizationObject()` .

**NOTA** Preste mucha atención al manejo de excepciones al interactuar con el cache. Para aumentar la resiliencia, nunca dejamos que falle toda la llamada si no podemos comunicarnos con el servidor Redis. En lugar de ello, registramos la excepción y dejamos que la llamada acudir al servicio de organización. En este caso de uso particular, el almacenamiento en caché es destinado a ayudar a mejorar el rendimiento y la ausencia del servidor de caché no debería afectar el éxito de la llamada.

Con el código de almacenamiento en caché de Redis implementado, debe acceder al servicio de licencias (sí, solo tiene dos servicios, pero tiene mucha infraestructura) y vea los mensajes de registro en el listado 8.10. Si tuviera que realizar dos solicitudes GET consecutivas en lo siguiente punto final del servicio de licencias, `http://localhost:5555/api/licensing/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a`, debería ver los dos siguientes declaraciones de salida en sus registros:

```

servicio de licencias_1 | 2016-10-26 09:10:18.455 DEBUG 28 --- [nio-8080-exec-1]
    ctlicOrganizationRestTemplateClient: No se puede ubicar la organización desde el caché de
    Redis: e254f8c-c442-4ebe-a82a-e2fc1d1ff78a.

licensingservice_1 2] | 2016-10-26 09:10:31.602 DEPURACIÓN 28 --- [nio-8080-exec-
    ctlicOrganizationRestTemplateClient: He recuperado con éxito una organización e254f8c-c442-4ebe-
    a82a-e2fc1d1ff78a del caché de Redis: com.thinkmechanix.licenses.model.Organization@6d20d301

```

La primera línea de la consola muestra la primera vez que intentó acceder al punto final del servicio de licencias para la organización `e254f8c-c442-4ebe-a82a-e2fc1d1ff78a`. El servicio de licencias verificó por primera vez el caché de Redis y no pudo encontrar la organización. registro que estaba buscando. Luego, el código llama al servicio de la organización para recuperar el

datos. La segunda línea que se imprimió desde la consola muestra que cuando presionas el Al configurar el punto final del servicio de licencias por segunda vez, el registro de la organización ahora se almacena en caché.

#### 8.4.2 Definición de canales personalizados

Anteriormente, creó su integración de mensajería entre los servicios de licencia y de organización para utilizar los canales de entrada y salida predeterminados que vienen incluidos con las interfaces Source y Sink en el proyecto Spring Cloud Stream. Sin embargo, si usted desea definir más de un canal para su aplicación o desea personalizar los nombres de tus canales, puedes definir tu propia interfaz y exponer tantos canales de entrada y salida según las necesidades de su aplicación.

Para crear un canal personalizado, llame a inboundOrgChanges en el servicio de licencias. Puede definir el canal en licensing-service/src/main/java/com/interfaz thinkmechanix/licenses/events/CustomChannels.java , como se muestra en el siguiente listado.

##### Listado 8.12 Definiendo un canal de entrada personalizado para el servicio de licencias

```
paquete com.thinkmechanix.licenses.events;

importar org.springframework.cloud.stream.annotation.Input;
importar org.springframework.messaging.SubscribableChannel;

interfaz pública CustomChannels {
    @Input("cambiosdeorganizaciónentrantes")
    Organizaciones de canales suscribibles();
}

} La anotación @Input es una anotación a nivel de método que define el nombre del canal.
```

Cada canal expuesto a través de la anotación @Input debe devolver una clase SubscribableChannel.

La conclusión clave del listado 8.12 es que para cada canal de entrada personalizado que deseé exponer, marca un método que devuelve una clase SubscribableChannel con el @ Anotación de entrada. Si desea definir canales de salida para publicar mensajes, usaría @OutputChannel encima del método que se llamará. En el caso de un canal de salida , el método definido devolverá una clase MessageChannel en su lugar de la clase SubscribableChannel utilizada con el canal de entrada :

```
@OutputChannel("organización saliente")
Canal de mensajes outboundOrg();
```

Ahora que tiene definido un canal de entrada personalizado , necesita modificar dos más. cosas para usarlo en el servicio de licencias. Primero, debe modificar el servicio de licencias para asignar el nombre de su canal de entrada personalizado a su tema de Kafka. El siguiente listado muestra esto.

##### Listado 8.13 Modificando el servicio de licencia para usar su canal de entrada personalizado

```
primavera:
...
nube:
```

```
...
arroyo:
fijaciones:
inboundOrgChanges: destino:
    orgChangeTopic
    tipo de contenido: aplicación/json
    grupo: grupo de licencias
```

Cambio el nombre del canal de entrada a inboundOrgChanges.

Para usar su canal de entrada personalizado, necesita injectar la interfaz CustomChannels que definió en una clase que la usará para procesar mensajes. Para el ejemplo de almacenamiento en caché distribuido, moví el código para manejar un mensaje entrante a la siguiente clase de servicio de licencias: licensing-service/src/main/java/es/thinkmechanix/licenses/events/handlers/OrganizationChangeControlador.java. La siguiente lista muestra el código de manejo de mensajes que Uselo con el canal inboundOrgChanges que definió.

#### Listado 8.14 Usando el nuevo canal personalizado en OrganizationChangeHandler

Mueva @EnableBindings fuera de la clase Application.java y dentro la clase OrganizationChangeHandler. Esta vez en lugar de usar Sink.class, use su clase CustomChannels como parámetro para pasar.

```
@EnableBinding(CustomChannels.class) clase pública
OrganizationChangeHandler {

    @StreamListener("inboundOrgChanges") public void
    loggerSink(OrganizationChangeModel orgChange) {
        .... //Entraremos en el resto del código en breve
    }
}
```

Con la anotación @StreamListener, pasaste el nombre de su canal, "inboundOrgChanges", en lugar de usar Sink.INPUT.

#### 8.4.3 Reuniéndolo todo: borrar el caché cuando se envía un mensaje recibió

En este punto no es necesario hacer nada con el servicio de organización. El servicio está todo configurado para publicar un mensaje cada vez que se agrega, actualiza o actualiza una organización. eliminado. Todo lo que tienes que hacer es construir la clase OrganizationChangeHandler del listado 8.14. El siguiente listado muestra la implementación completa de esta clase.

#### Listado 8.15 Procesamiento de un cambio de organización en el servicio de licencias

```
@EnableBinding(CustomChannels.clase)
clase pública OrganizationChangeHandler {
    @autocableado
    organización privadaRedisRepository
    organizaciónRedisRepository;

    registrador final estático privado registrador =
        LoggerFactory.getLogger(OrganizationChangeHandler.clase);
```

La clase OrganizationRedisRepository que que usas para interactuar con Redis se inyecta en OrganizationChangeHandler.

```

@StreamListener("inboundOrgChanges")
registrar vacío público (OrganizationChangeModel orgChange) {
    cambiar(orgChange.getAction()){
        // Eliminado por motivos de concisión

caso "ACTUALIZACIÓN":
    logger.debug("Se recibió un evento de ACTUALIZACIÓN
        del servicio de organización de
        ID de organización {}",

orgChange.getOrganizationId()); organizaciónRedisRepository

.deleteOrganization(orgChange.getOrganizationId());
romper;
caso "ELIMINAR":
    logger.debug("Se recibió un evento DELETE
        del servicio de organización para la identificación de la organización {}",

orgChange.getOrganizationId()); organizaciónRedisRepository

.deleteOrganization(orgChange.getOrganizationId());
romper;
por defecto:
logger.error("Se recibió un evento DESCONOCIDO
        del servicio de organización de tipo {}",

orgChange.getType());
romper;
}
}

```

The diagram illustrates the event processing logic. It starts with a vertical line representing the flow of events. A bracket on the right side indicates that when a message is received, it triggers an inspection of the action taken with the data, followed by a subsequent reaction. Another bracket further down the line indicates that if organization data is updated or deleted, it triggers the eviction of organization data from Redis through the OrganizationRedisRepository class.

Cuando recibes un mensaje, inspeccionar la acción que fue tomado con los datos y luego reaccionar en consecuencia.

Si los datos de la organización se actualizan o eliminan, desalojar los datos de la organización de Redis a través de Clase OrganizationRedisRepository.

## 8.5 Resumen

La comunicación asincrónica con mensajería es una parte crítica de los microservicios. arquitectura.

El uso de mensajería dentro de sus aplicaciones permite que sus servicios escala y volverse más tolerante a fallos.

Spring Cloud Stream simplifica la producción y el consumo de mensajes mediante el uso de anotaciones simples y abstrayendo detalles específicos de la plataforma de la plataforma de mensajes subyacente.

Una fuente de mensajes Spring Cloud Stream es un método Java anotado que se utiliza para publicar mensajes en la cola de un intermediario de mensajes.

Un receptor de mensajes Spring Cloud Stream es un método Java anotado que recibe mensajes fuera de la cola de un intermediario de mensajes.

Redis es un almacén de valores clave que se puede utilizar como base de datos y caché.

# Seguimiento distribuido con Detective de la nube de primavera y zipkin

## Este capítulo cubre

Uso de Spring Cloud Sleuth para inyectar información de seguimiento en llamadas de servicio

Uso de la agregación de registros para ver registros distribuidos transacción

Consulta a través de una herramienta de agregación de registros Uso de OpenZipkin para comprender visualmente la información de un usuario transacción a medida que fluye a través de múltiples llamadas de microservicio

Personalización de la información de seguimiento con Spring Cloud Sleuth y zipkin

La arquitectura de microservicios es un poderoso paradigma de diseño para desglosar sistemas de software monolíticos complejos en piezas más pequeñas y manejables. Estos las piezas manejables se pueden construir y desplegar independientemente unas de otras; Sin embargo, esta flexibilidad tiene un precio: la complejidad. Debido a que los microservicios se distribuyen por naturaleza, intentar depurar dónde se produce un problema puede resultar exasperante. La naturaleza distribuida de los servicios significa que hay que rastrear uno o más transacciones a través de múltiples servicios, máquinas físicas y diferentes almacenes de datos, y tratar de reconstruir qué está pasando exactamente.

Este capítulo presenta varias técnicas y tecnologías para hacer aplicaciones distribuidas. posible depuración. En este capítulo, analizamos lo siguiente:

Uso de ID de correlación para vincular transacciones en múltiples servicios

Agregar datos de registro de múltiples servicios en una única fuente de búsqueda Visualizar el flujo de una transacción de usuario a través de múltiples servicios y comprender las características de rendimiento de cada parte de la transacción

Para lograr las tres cosas, utilizará tres tecnologías diferentes:

Spring Cloud Sleuth (<https://cloud.spring.io/spring-cloud-sleuth/>)—Spring

Cloud Sleuth es un proyecto de Spring Cloud que instrumenta sus llamadas HTTP con ID de correlación y proporciona enlaces que alimentan los datos de seguimiento que está produciendo en AbrirZipkin. Lo hace agregando filtros e interactuando con otros Spring componentes para permitir que los ID de correlación que se generan pasen a todos los llamadas al sistema.

Papertrail (<https://papertrailapp.com>)—Papertrail es un servicio basado en la nube

(basado en freemium) que le permite agregar datos de registro de múltiples fuentes en una única base de datos con capacidad de búsqueda. Tiene opciones para la agregación de registros, incluidas soluciones locales, basadas en la nube, de código abierto y comerciales.

Exploraremos varias de estas alternativas más adelante en el capítulo.

Zipkin (<http://zipkin.io>)—Zipkin es una herramienta de visualización de datos de código abierto que puede mostrar el flujo de una transacción a través de múltiples servicios. Zipkin te permite dividir una transacción en sus componentes e identificar visualmente dónde

Puede haber puntos críticos de rendimiento.

Para comenzar este capítulo, comenzamos con la herramienta de rastreo más simple: el ID de correlación.

**NOTA** Partes de este capítulo se basan en el material cubierto en el capítulo 6 (específicamente el material sobre filtros previos, de respuesta y posteriores de Zuul). Si no has leído capítulo 6 todavía, te recomiendo que lo haga antes de leer este capítulo.

## 9.1 Spring Cloud Sleuth y el ID de correlación

Introducimos por primera vez el concepto de ID de correlación en los capítulos 5 y 6. Un ID de correlación es un número o cadena único generado aleatoriamente que se asigna a una transacción cuando se inicia una transacción. A medida que la transacción fluye a través de múltiples servicios, el ID de correlación se propaga de una llamada de servicio a otra. En el contexto del capítulo 6, utilizó un filtro Zuul para inspeccionar todas las solicitudes HTTP entrantes e injectar una ID de correlación si no había ninguna presente.

Una vez que el ID de correlación estuvo presente, utilizó un filtro Spring HTTP personalizado en cada uno de sus servicios para asignar la variable entrante a un UserContext personalizado objeto. Con el objeto UserContext en su lugar, ahora puede agregar manualmente el ID de correlación a cualquiera de sus declaraciones de registro asegurándose de agregar la correlación.

ID a la declaración de registro o, con un poco de trabajo, agregue el ID de correlación directamente a Contexto de diagnóstico mapeado de Spring (MDC). También escribiste un Spring Interceptor que

garantizaría que todas las llamadas HTTP de un servicio propagarían el ID de correlación por agregando el ID de correlación a los encabezados HTTP en cualquier llamada saliente.

Ah, y tuviste que realizar magia de Spring y Hystrix para asegurarte de que el hilo El contexto del hilo principal que contiene el ID de correlación se propagó correctamente a Hystrix. Vaya, al final se creó mucha infraestructura para algo que espera que sólo se analice cuando ocurra un problema (usando una ID de correlación para rastrear lo que sucede con una transacción).

Afortunadamente, Spring Cloud Sleuth administra toda esta infraestructura y complejidad de código por usted. Al agregar Spring Cloud Sleuth a sus Spring Microservices, puede

Cree e inyecte de forma transparente una ID de correlación en sus llamadas de servicio si una no existe.

Gestionar la propagación del ID de correlación a las llamadas de servicio salientes para que el ID de correlación para una transacción se agrega automáticamente a las llamadas salientes.

Agregue la información de correlación al registro MDC de Spring para que la información generada

El ID de correlación se registra automáticamente mediante SL4J y Logback predeterminados de Spring Boots. implementación.

Opcionalmente, publique la información de rastreo en la llamada de servicio al Zipkin-plataforma de seguimiento distribuida.

**NOTA** Con Spring Cloud Sleuth, si utiliza la implementación de registro de Spring Boot, obtendrá automáticamente ID de correlación agregados a las declaraciones de registro. pones tus microservicios.

Sigamos adelante y agreguemos Spring Cloud Sleuth a sus servicios de organización y licencias.

### 9.1.1 Agregar Spring Cloud Sleuth a las licencias y la organización

Para comenzar a utilizar Spring Cloud Sleuth en sus dos servicios (licencias y organización), debe agregar una única dependencia de Maven a los archivos pom.xml en ambos servicios:

```
<dependencia>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>detective-iniciador-de-nube-primavera</artifactId>
</dependencia>
```

Esta dependencia incorporará todas las bibliotecas principales necesarias para Spring Cloud Sleuth.

Eso es todo. Una vez que se establezca esta dependencia, su servicio ahora

**1** Inspecione cada servicio HTTP entrante y determine si Spring

La información de seguimiento de Cloud Sleuth existe en la llamada entrante. si la primavera Los datos de seguimiento de Cloud Sleuth existen, la información de seguimiento pasada a su El microservicio se capturará y estará disponible para su servicio para iniciar sesión. y procesamiento.

**2** Agregue información de seguimiento de Spring Cloud Sleuth al Spring MDC para que cada La declaración de registro creada por su microservicio se agregará a los registros.

**3** Inyecte información de seguimiento de Spring Cloud en cada llamada HTTP saliente y Mensaje del canal de mensajería de primavera que hace su servicio.

### 9.1.2 Anatomía de un rastro de Spring Cloud Sleuth

Si todo está configurado correctamente, cualquier declaración de registro escrita dentro del código de su aplicación de servicio ahora incluirá información de seguimiento de Spring Cloud Sleuth. Por ejemplo,

La figura 9.1 muestra cómo se vería la salida del servicio si hicieras un HTTP.

OBTENER <http://localhost:5555/api/organization/v1/organizations/e254f8c>

-c442-4ebe-a82a-e2fc1d1ff78a en el servicio de organización.

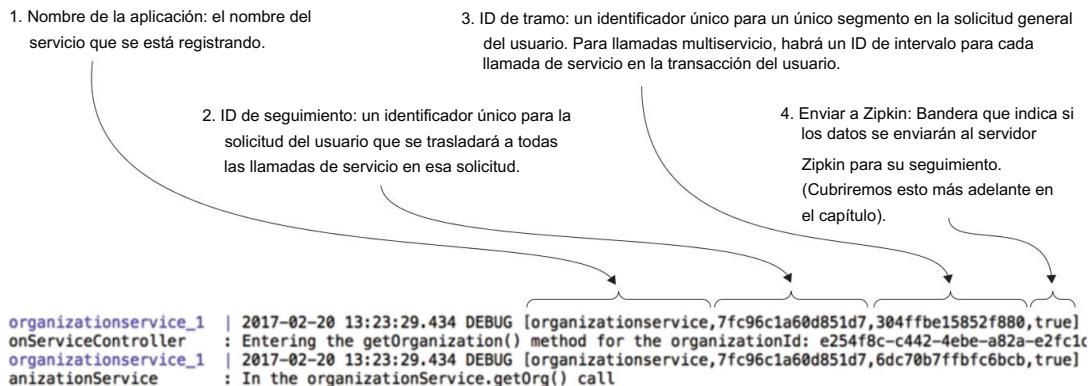


Figura 9.1 Spring Cloud Sleuth agrega cuatro piezas de información de seguimiento a cada entrada de registro escrita por su servicio.

Estos datos ayudan a vincular las llamadas de servicio para la solicitud de un usuario.

Spring Cloud Sleuth agregará cuatro datos a cada entrada del registro. Estos cuatro piezas (numeradas para corresponder con los números de la figura 9.1) son

- 1 Nombre de la aplicación del servicio: este será el nombre de la aplicación. se realiza la entrada de registro. De forma predeterminada, Spring Cloud Sleuth usa el nombre de la aplicación (`spring.application.name`) como el nombre que se escribe en el seguimiento.
- 2 ID de seguimiento : ID de seguimiento es el término equivalente para ID de correlación. Es un número único número que representa una transacción completa.
- 3 ID de intervalo : un ID de intervalo es un ID único que representa parte de la transacción general. Cada servicio que participe en la transacción tendrá su propio ID de intervalo. Los ID de tramo son particularmente relevantes cuando se integra con Zipkin para visualizar sus transacciones.
- 4 Si los datos de seguimiento se enviaron a Zipkin: en servicios de gran volumen, la cantidad de Los datos de seguimiento generados pueden ser abrumadores y no agregar una cantidad significativa de valor. Spring Cloud Sleuth le permite determinar cuándo y cómo enviar una transacción a Zipkin. El indicador verdadero/falso al final de Spring Cloud Sleuth El bloque de seguimiento le indica si la información de seguimiento se envió a Zipkin.

The diagram shows a log entry from the `licensingService_1` controller. It contains several log lines with annotations:

- Annotations:**
  - A red arrow points to the first log line, labeled "Las dos llamadas tienen el mismo ID de seguimiento." (Both calls have the same tracking ID).
  - A red arrow points to the second log line, labeled "Los ID de intervalo para las dos llamadas de servicio son diferentes." (The interval IDs for the two service calls are different).
- Log Lines:**

```

licensingService_1 | 2017-02-20 14:31:19.624 DEBUG [licensingService,a9e3e1786b74d302,a9e3e1786b74d302,true] 34 --- [nio
eController      : Entering the license-service-controller
licensingService_1 | Hibernate: select license0_.license_id as license_1_0_, license0_.comment as comment2_0_, license0_
license0_.license_max as license_4_0_, license0_.license_type as license_5_0_, license0_.organization_id as organizacion6_0_
0_ from licenses license0_ where license0_.organization_id=? and license0_.license_id=?
licensingService_1 | 2017-02-20 14:31:19.632 DEBUG [licensingService,a9e3e1786b74d302,a9e3e1786b74d302,true] 34 --- [nio
estTemplateClient : Unable to locate organization from the redis cache: e254f8c-c442-4ebe-a82a-e2fc1d1ff78a.
organizationService_1 | 2017-02-20 14:31:19.678 DEBUG [organizationService,a9e3e1786b74d302,3867263ed85ffbf4,true] 33 --- [i
onServiceController : Entering the getOrganization() method for the organizationId: e254f8c-c442-4ebe-a82a-e2fc1d1ff78a

```

Figura 9.2 Con múltiples servicios involucrados en una transacción, puede ver que comparten el mismo ID de seguimiento.

Hasta ahora, sólo hemos analizado los datos de registro producidos por una única llamada de servicio.

Veamos qué sucede cuando realiza una llamada al servicio de licencias al

**OBTENER** <http://localhost:5555/api/licensing/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-a82a-e2fc>

1d1ff78a. Recuerde, el servicio de licencias también tiene que llamar a la organización servicio. La Figura 9.2 muestra el resultado del registro de las dos llamadas de servicio.

Al observar la figura 9.2, puede ver que tanto los servicios de organización como los de licencia tienen el mismo ID de seguimiento a9e3e1786b74d302. Sin embargo, el servicio de licencias tiene un ID de intervalo de a9e3e1786b74d302 (el mismo valor que el ID de transacción). El servicio de organización tiene un ID de intervalo de 3867263ed85ffbf4.

Al agregar nada más que unas pocas dependencias POM, ha reemplazado todas las infraestructura de identificación de correlación que usted desarrolló en los capítulos 5 y 6. Personalmente, nada me hace más feliz en este mundo que reemplazar un código complejo de estilo infraestructura. con el código de otra persona.

## 9.2 Agregación de registros y Spring Cloud Sleuth

En un entorno de microservicios a gran escala (especialmente en la nube), el registro de datos es una herramienta crítica para la depuración de problemas. Sin embargo, debido a que la funcionalidad de un La aplicación basada en microservicios se descompone en pequeños servicios granulares y usted Puede tener múltiples instancias de servicio para un solo tipo de servicio, intentando vincularlo a los datos de registro. de múltiples servicios para resolver el problema de un usuario puede ser extremadamente difícil. Los desarrolladores que intentan depurar un problema en varios servidores a menudo tienen que intentar lo siguiente:

Inicie sesión en varios servidores para inspeccionar los registros presentes en cada servidor. Esto es un tarea extremadamente laboriosa, especialmente si los servicios en cuestión tienen diferentes volúmenes de transacciones que hacen que los registros se trasladen a diferentes velocidades. Escriba scripts de consulta locales que intentarán analizar los registros e identificar las entradas de registro relevantes. Debido a que cada consulta puede ser diferente, a menudo finalizas se encuentra con una gran proliferación de scripts personalizados para consultar datos de sus registros.

Prolongar la recuperación de un proceso de servicio inactivo porque el desarrollador necesita hacer una copia de seguridad de los registros que residen en un servidor. Si un servidor que aloja un servicio falla por completo, los registros generalmente se pierden.

Cada uno de los problemas enumerados son problemas reales con los que me he encontrado. Depurar un problema en servidores distribuidos es un trabajo feo y a menudo aumenta significativamente la cantidad de tiempo que lleva identificar y resolver un problema.

Un enfoque mucho mejor es transmitir, en tiempo real, todos los registros de todos sus servicios. instancias a un punto de agregación centralizado donde los datos de registro se pueden indexar y hecho buscable. La Figura 9.3 muestra a nivel conceptual cómo este registro “unificado” la arquitectura funcionaría.

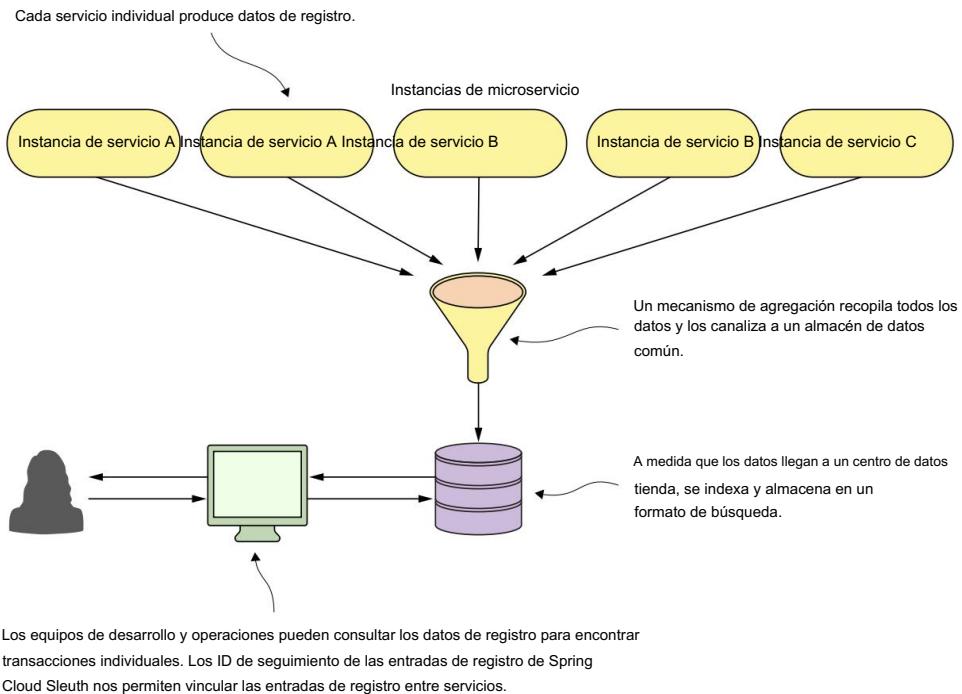


Figura 9.3 La combinación de registros agregados y una ID de transacción única en las entradas del registro de servicio hace que la depuración de transacciones distribuidas sea más manejable.

Afortunadamente, existen múltiples productos comerciales y de código abierto que pueden ayudar implementa la arquitectura de registro descrita anteriormente. Además, existen múltiples modelos de implementación que le permitirán elegir entre una solución local o local. solución gestionada o una solución basada en la nube. La tabla 9.1 resume varios de los opciones disponibles para la infraestructura maderera.

Tabla 9.1 Opciones para soluciones de agregación de registros para uso con Spring Boot

nombre del producto	Modelos de implementación	Notas
búsqueda elástica, Logstash, Kibana (ELK)	Fuente abierta Comercial  Normalmente se implementa en las instalaciones	<a href="http://elastic.co">http://elastic.co</a> Motor de búsqueda de propósito general. Puede realizar agregación de registros a través de (ELK-stack) Requiere el soporte más práctico
tronco gris	Fuente abierta Comercial  En la premisa	<a href="http://graylog.org">http://graylog.org</a> Plataforma de código abierto diseñada para instalarse localmente
Splunk	Sólo comercial  Local y basado en la nube	<a href="http://splunk.com">http://splunk.com</a> La más antigua y completa de las herramientas de agregación y gestión de registros Originalmente era una solución local, pero desde entonces ofrece una oferta en la nube.
Lógica de sumo	Freemium Comercial  Basado en la nube	<a href="http://sumologic.com">http://sumologic.com</a> Modelo de precios freemium/escalonado Se ejecuta solo como un servicio en la nube Requiere una cuenta de trabajo corporativa para registrarse (no cuentas de Gmail o Yahoo)
Rastro de papel	Freemium Comercial  Basado en la nube	<a href="http://papertrailapp.com">http://papertrailapp.com</a> Modelo de precios freemium/escalonado Se ejecuta solo como un servicio en la nube

Con todas estas opciones, puede resultar complicado elegir cuál es la mejor. Cada organización será diferente y tendrá necesidades diferentes.

Para los propósitos de este capítulo, veremos Papertrail como un ejemplo de cómo integrar registros respaldados por Spring Cloud Sleuth en una plataforma de registro unificada. I Elegí Papertrail porque

- 1 Tiene un modelo freemium que te permite registrarte para obtener una cuenta de nivel gratuito.
- 2 Es increíblemente fácil de configurar, especialmente con tiempos de ejecución de contenedores como Docker.
- 3 Está basado en la nube. Si bien creo que una buena infraestructura maderera es fundamental para un aplicación de microservicios, no creo que la mayoría de las organizaciones tengan el tiempo o Talento técnico para configurar y gestionar adecuadamente una plataforma de registro.

### 9.2.1 Una implementación de Spring Cloud Sleuth/Papertrail en acción

En la figura 9.3 vimos una arquitectura de registro unificado general. Veamos ahora cómo La misma arquitectura se puede implementar con Spring Cloud Sleuth y Papertrail.

Para configurar Papertrail para que funcione con su entorno, debemos seguir lo siguiente comportamiento:

- 1 Cree una cuenta de Papertrail y configure un conector de syslog de Papertrail.
- 2 Defina un contenedor Docker de Logspout (<https://github.com/gliderlabs/log-canalón>) para capturar el estándar de todos los contenedores Docker.

### 3 Pruebe la implementación emitiendo consultas basadas en el ID de correlación de Detective de la nube de primavera.

La Figura 9.4 muestra el estado final de su implementación y cómo Spring Cloud Sleuth y Papertrail se combinan para su solución.

1. Los contenedores individuales escriben sus datos de registro en formato estándar. Nada ha cambiado en cuanto a su configuración.

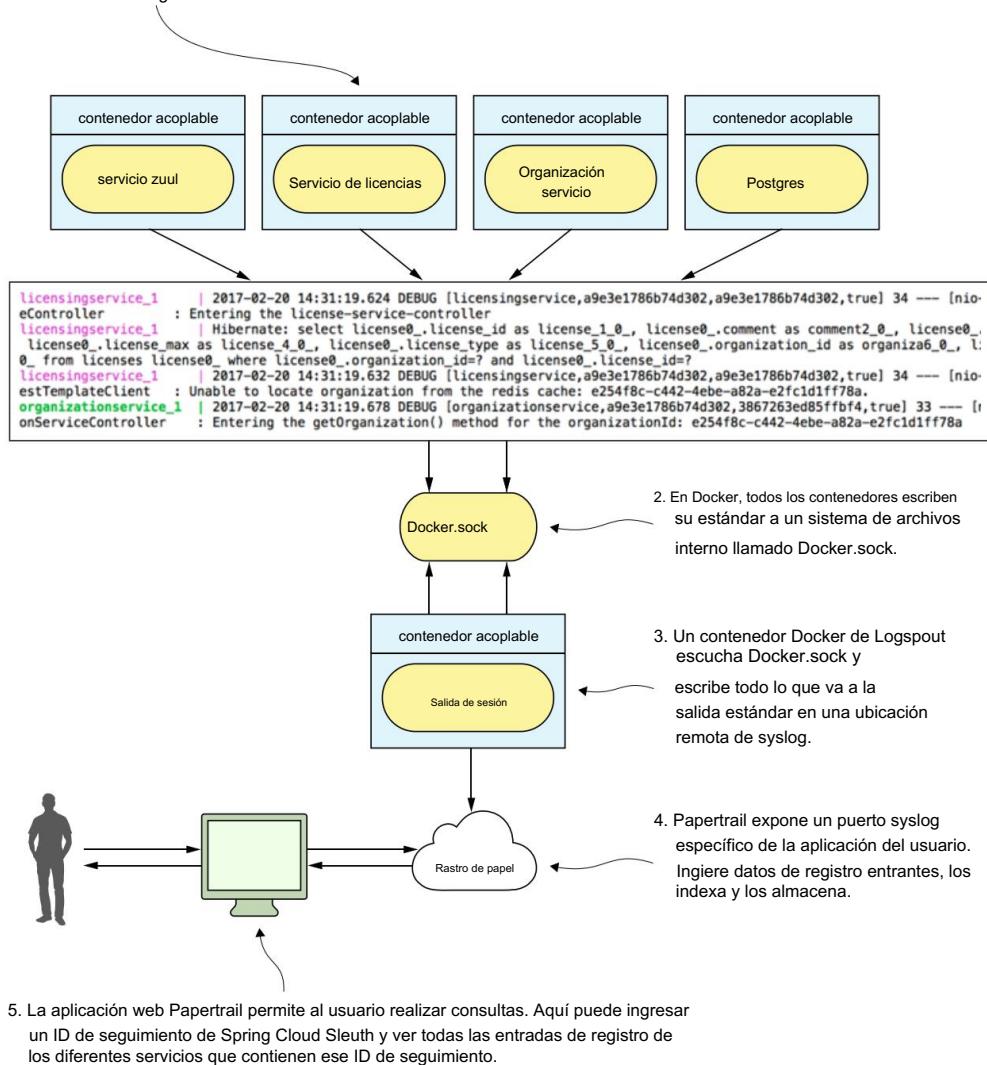
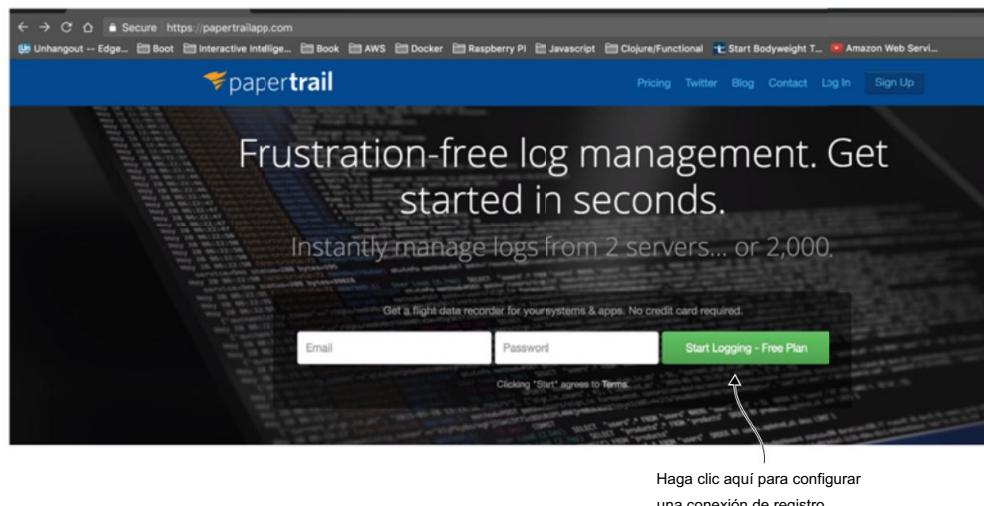


Figura 9.4 El uso de capacidades nativas de Docker, logspot y Papertrail le permite implementar rápidamente una arquitectura de registro unificada.



Haga clic aquí para configurar una conexión de registro.

Figura 9.5 Para comenzar, cree una cuenta en Papertrail.

### 9.2.2 Crear una cuenta de Papertrail y configurar un conector syslog

Comenzarás configurando un Papertrail. Para comenzar, vaya a <https://papertrailapp.com> y haga clic en el botón verde "Iniciar registro - Plan gratuito". La figura 9.5 muestra esto.

Papertrail no requiere una cantidad significativa de información para comenzar; sólo una dirección de correo electrónico válida. Una vez que haya completado la información de la cuenta, se le presentará una pantalla para configurar su primer sistema desde el cual registrar datos. La Figura 9.6 muestra esta pantalla.

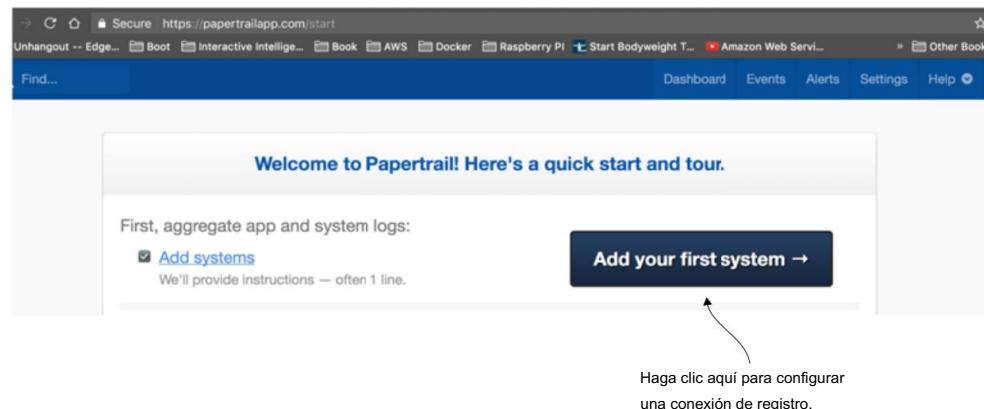


Figura 9.6 A continuación, elija cómo enviará los datos de registro a Papertrail.

De forma predeterminada, Papertrail le permite enviarle datos de registro a través de una llamada Syslog (<https://en.wikipedia.org/wiki/Syslog>). Syslog es un formato de mensajería de registro que se originó en UNIX. Syslog permite el envío de mensajes de registro a través de TCP y UDP. Papertrail definirá automáticamente un puerto Syslog que puede utilizar para escribir mensajes de registro. Para los fines de esta discusión, utilizará este puerto predeterminado. La Figura 9.7 muestra la cadena de conexión Syslog que se genera automáticamente cuando hace clic en el botón "Agregar su primer sistema" que se muestra en la figura 9.6.

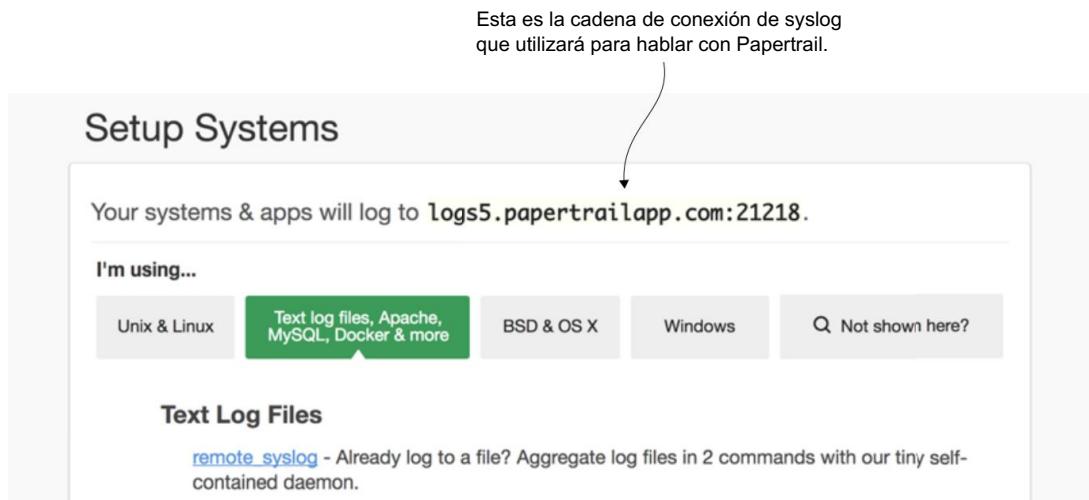


Figura 9.7 Papertrail utiliza Syslog como uno de los mecanismos para enviarle datos.

En este punto, ya está todo configurado con Papertrail. Ahora debe configurar su entorno Docker para capturar la salida de cada uno de los contenedores que ejecutan sus servicios en el punto final remoto de syslog definido en la figura 9.7.

**NOTA** La cadena de conexión de la figura 9.7 es exclusiva de mi cuenta. Deberá asegurarse de utilizar la cadena de conexión generada por Paper-trail o definir una a través de la opción de menú Configuración de Papertrail > Destinos de registro.

### 9.2.3 Redirigir la salida de Docker a Papertrail

Normalmente, si está ejecutando cada uno de sus servicios en su propia máquina virtual, tendrá que configurar la configuración de registro de cada servicio individual para enviar su información de registro a un punto final de syslog remoto (como el expuesto a través de Papertrail).

Afortunadamente, Docker hace que sea increíblemente fácil capturar todos los resultados de cualquier contenedor Docker que se ejecute en una máquina física o virtual. El demonio Docker se comunica con todos los contenedores Docker que administra a través de un socket Unix llamado docker.sock. Cualquier contenedor que se ejecute en el servidor donde se ejecuta Docker

puede conectarse a docker.sock y recibir todos los mensajes generados por todos los otros contenedores que se ejecutan en ese servidor. En los términos más simples, docker.sock es como un Tubería a la que sus contenedores pueden conectarse y capturar las actividades generales que se llevan a cabo. dentro del entorno de ejecución de Docker en el servidor virtual, el demonio Docker está que se ejecuta en.

Vas a utilizar un software "Dockerizado" llamado Logspout (<https://github.com/gliderlabs/logspout>) eso escuchará el socket docker.sock y luego capture cualquier mensaje de salida estándar generado en el tiempo de ejecución de Docker y redirija la salida a un syslog remoto (Papertrail). Para configurar su contenedor Logspout, debe agregar una sola entrada al archivo docker-compose.yml que usa para iniciar todo Docker contenedores utilizados para los ejemplos de código de este capítulo. El archivo docker/common/docker-compose.yml que necesita modificar debe tener agregada la siguiente entrada:

salida de registro:

```
imagen: gliderlabs/logspout
comando: syslog://logs5.papertrailapp.com:21218
volúmenes:
  - /var/run/docker.sock:/var/run/docker.sock
```

**NOTA** En el fragmento de código anterior, deberá reemplazar el valor en el atributo "comando" con el valor que le proporcionó Papertrail. Si usted use el fragmento de Logspout anterior, su contenedor de Logspout escribirá felizmente sus entradas de registro a mi cuenta Papertrail.

Ahora, cuando inicie su entorno Docker en este capítulo, todos los datos enviados a la salida estándar de un contenedor se enviarán a Papertrail. Puede comprobarlo usted mismo iniciando sesión en su cuenta de Papertrail después de haber iniciado los ejemplos de Docker del capítulo 9.

y haciendo clic en el botón Eventos en la parte superior derecha de su pantalla.

La Figura 9.8 muestra un ejemplo de cómo se ven los datos enviados a Papertrail.

Los eventos del registro de servicio individual se escriben en el salida estándar del contenedor. La salida estándar del contenedor es capturado por Logspout y luego enviado a Papertrail.

Haga clic aquí para ver los eventos de registro que se envían a Papertrail.

Timestamp	Container ID	Service	Log Message	Environment
Feb 20 09:30:08	8aa0b596472d	common_licensingservice_1	2017-02-20 14:30:08.192 INFO [licensingservice, main] org.apache.zookeeper.ZooKeeper : Client environment:os.name=Linux	
Feb 20 09:30:08	8aa0b596472d	common_licensingservice_1	2017-02-20 14:30:08.192 INFO [licensingservice, main] org.apache.zookeeper.ZooKeeper : Client environment:os.arch=amd64	
Feb 20 09:30:08	8aa0b596472d	common_licensingservice_1	2017-02-20 14:30:08.193 INFO [licensingservice, main] org.apache.zookeeper.ZooKeeper : Client environment:os.version=4.9.8-moby	
Feb 20 09:30:08	8aa0b596472d	common_licensingservice_1	2017-02-20 14:30:08.193 INFO [licensingservice, main] org.apache.zookeeper.ZooKeeper : Client environment:user.name=root	
Feb 20 09:30:08	8aa0b596472d	common_licensingservice_1	2017-02-20 14:30:08.193 INFO [licensingservice, main] org.apache.zookeeper.ZooKeeper : Client environment:user.home=/root	
Feb 20 09:30:08	8aa0b596472d	common_licensingservice_1	2017-02-20 14:30:08.193 INFO [licensingservice, main] org.apache.zookeeper.ZooKeeper : Client environment:user.dir=/	

Figura 9.8 Con el contenedor Docker de Logspout definido, los datos escritos en la salida estándar de cada contenedor se enviarán a Papertrail.

#### ¿Por qué no utilizar el controlador de registro de Docker?

Docker 1.6 y superiores le permiten definir controladores de registro alternativos para escribir los mensajes stdout/stderr escritos desde cada contenedor. Uno de los controladores de registro es un controlador syslog que se puede utilizar para escribir mensajes en un escucha de syslog remoto.

¿Por qué elegí Logspout en lugar de utilizar el controlador de registro estándar de Docker? La razón principal es la flexibilidad. Logspout ofrece funciones para personalizar qué datos de registro se envían a su plataforma de agregación de registros. Las características que ofrece Logspout incluyen

La capacidad de enviar datos de registro a múltiples puntos finales a la vez. Muchas empresas querrán enviar sus datos de registro a una plataforma de agregación de registros y también querrán herramientas de monitoreo de seguridad que monitorearán los registros producidos en busca de datos confidenciales.

Una ubicación centralizada para filtrar qué contenedores van a enviar sus datos de registro. Con el controlador Docker, debe configurar manualmente el controlador de registro para cada contenedor en su archivo docker-compose.yml. Logspout le permite definir filtros para contenedores específicos e incluso patrones de cadenas específicos en una configuración centralizada.

Rutas HTTP personalizadas que permiten a las aplicaciones escribir información de registro a través de puntos finales HTTP específicos. Esta característica le permite hacer cosas como escribir mensajes de registro específicos en una plataforma de agregación de registros descendente específica. Por ejemplo, es posible que los mensajes de registro generales de stdout/stderr vayan a Papertrail, donde es posible que desee enviar información de auditoría de aplicaciones específica a un servidor interno de Elasticsearch.

Integración con protocolos más allá de syslog. Logspout le permite enviar mensajes a través de los protocolos UDP y TCP. Logspout también tiene módulos de terceros que pueden integrar stdout/stderr de Docker en Elasticsearch.

#### 9.2.4 Búsqueda de ID de seguimiento de Spring Cloud Sleuth en Papertrail

Ahora que sus registros fluyen a Papertrail, realmente puede comenzar a apreciar que Spring Cloud Sleuth agrega ID de seguimiento a todas sus entradas de registro. Para consultar todas las entradas de registro relacionadas con una sola transacción, todo lo que necesita hacer es tomar un ID de seguimiento y consultarlo en el cuadro de consulta de la pantalla de eventos de Papertrail. La Figura 9.9 muestra cómo ejecutar una consulta mediante el ID de seguimiento del detective de Spring Cloud que utilizamos anteriormente en la sección 9.1.2: a9e3e1786b74d302..

#### Consolide el registro y elogie lo mundano No subestime lo importante

que es tener una arquitectura de registro consolidada y una estrategia de correlación de servicios pensada. Parece una tarea mundana, pero mientras escribía este capítulo, utilicé herramientas de agregación de registros similares a Papertrail para rastrear una condición de carrera entre tres servicios diferentes para un proyecto en el que estaba trabajando. Resultó que la condición de carrera había estado ahí por más de un año, pero el servicio con la condición de carrera había estado funcionando bien hasta que agregamos un poco más de carga y otro actor en la mezcla para causar el problema.

Encontramos el problema solo después de pasar 1,5 semanas realizando consultas de registros y caminando a través del resultado de seguimiento de docenas de escenarios únicos. No habríamos encontrado el problema sin la plataforma de registro agregado que se había implementado. Este La experiencia reafirmó varias cosas:

- 1 Asegúrese de definir e implementar sus estrategias de registro desde el principio del desarrollo de su servicio. La implementación de la infraestructura de registro es a veces tediosa. difícil y requiere mucho tiempo una vez que el proyecto está en marcha.
- 2 El registro es una pieza fundamental de la infraestructura de microservicios: piense detenidamente antes de implementar su propia solución de registro o incluso intentar implementar una solución de registro local. Las plataformas de registro basadas en la nube valen la pena dinero que se gasta en ellos.
- 3 Conozca sus herramientas de registro: casi todas las plataformas de registro tendrán un lenguaje de consulta para consultar los registros consolidados. Los registros son una fuente increíble de información y métricas. Son esencialmente otro tipo de base de datos, y el El tiempo que dedique a aprender a realizar consultas le reportará enormes dividendos.

Los registros muestran que se llamó al servicio de licencias y luego al servicio de organización como parte de esta única transacción.

The screenshot shows the Spring Cloud Sleuth interface. At the top, there's a search bar labeled 'Find...' and navigation tabs for 'Dashboard', 'Events' (which is selected), 'Alerts', and 'Sets'. A message box says: 'Need to search events before Thursday, Feb 16 at 2:13 AM? Download [archive](#) retain logs longer, increase [duration](#)'. Below this, log entries are listed:

```

Feb 20 09:31:19 8aa0b596472d common_licensingservice_1: 2017-02-20 14:31:19.624 DEBUG
[licensingService,a9e3e1786b74d302,a9e3e1786b74d302,true] 34 --- [nio-8080-exec-3] c.t.l.c.LicenseServiceController : Entering the license-service-controller
Feb 20 09:31:19 8aa0b596472d common_licensingservice_1: 2017-02-20 14:31:19.632 DEBUG
[licensingService,a9e3e1786b74d302,a9e3e1786b74d302,true] 34 --- [nio-8080-exec-3] c.t.l.c.OrganizationRepository : Unable to locate organization from the redis cache: e254f8c-c442-4ebe-a82a-e2fc1d1ff78a.
Feb 20 09:31:19 d826abba49c5 common_organizationservice_1: 2017-02-20 14:31:19.678 DEBUG
[organizationService,a9e3e1786b74d302,3867263ed85ffbf4,true] 33 --- [nio-8085-exec-2] c.t.o.c.OrganizationService : Entering the getOrganization() method for the organizationId: e254f8c-c442-4ebe-a82a-e2fc1d1ff78a
Feb 20 09:31:19 d826abba49c5 common_organizationservice_1: 2017-02-20 14:31:19.686 DEBUG
[organizationService,a9e3e1786b74d302,89ddd5de211fe516,true] 33 --- [nio-8085-exec-2] c.t.o.services.OrganizationService : In the organizationService.getOrg() call

```

At the bottom, there's a search bar with the ID 'a9e3e1786b74d302', a 'Search' button, a 'All Systems' button, and a menu icon. An annotation points to the search bar with the text: 'Aquí está el rastro de Spring Cloud Sleuth ID que vas a consultar.'

Figura 9.9 El ID de seguimiento le permite filtrar todas las entradas del registro relacionadas con esa única transacción.

### 9.2.5 Agregar el ID de correlación a la respuesta HTTP con Zuul

Si inspecciona la respuesta HTTP de cualquier llamada de servicio realizada con Spring Cloud Detective, verá que el ID de seguimiento utilizado en la llamada nunca se devuelve en el protocolo HTTP. encabezados de respuesta. Si inspecciona la documentación de Spring Cloud Sleuth, encontrará Vea que el equipo de Spring Cloud Sleuth cree que devolver cualquiera de los datos de seguimiento puede ser un posible problema de seguridad (aunque no enumeran explicitamente los motivos por los que ellos creen esto.)

Sin embargo, descubrí que la devolución de una correlación o ID de seguimiento en el protocolo HTTP La respuesta es invaluable al depurar un problema. Spring Cloud Sleuth permite usted debe "decorar" la información de respuesta HTTP con sus ID de seguimiento y extensión . Sin embargo, el proceso para hacer esto implica escribir tres clases e injectar dos clases personalizadas. Frijoles tiernos. Si desea adoptar este enfoque, puede verlo en Spring Cloud. Documentación de detective (<http://cloud.spring.io/spring-cloud-static/spring-cloud-sleuth/1.0.12.RELEASE/>). Una solución mucho más sencilla es escribir un filtro "POST" de Zuul que injectará el ID de seguimiento en la respuesta HTTP .

En el capítulo 6, cuando presentamos la puerta de enlace API de Zuul , vimos cómo construir un Zuul Filtro de respuesta "POST" para agregar el ID de correlación que generó para usar en sus servicios a la respuesta HTTP devuelta por la persona que llama. Ahora vas a modificar ese filtro para agregue el encabezado Spring Cloud Sleuth.

Para configurar su filtro de respuesta Zuul, debe agregar una única dependencia JAR a el archivo pom.xml de su servidor Zuul: spring-cloud-starter-sleuth. La dependencia spring-cloud-starter-sleuth se utilizará para decirle a Spring Cloud Sleuth que desea que Zuul participe en un seguimiento de Spring Cloud. Más adelante en el capítulo, cuando presenta Zipkin, verás que el servicio Zuul será la primera llamada en cualquier servicio invocación.

Para el capítulo 9, este archivo se puede encontrar en zuulsvr/pom.xml. El siguiente listado muestra estas dependencias.

#### Listado 9.1 Agregar Spring Cloud Sleuth a Zuul

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId> </dependency>
```

Añadiendo Spring-Cloud-Starter-Sleuth a Zuul hará que se genere un ID de seguimiento para cada servicio que se llama en Zuul

Una vez que esta nueva dependencia esté implementada, el filtro de "publicación" de Zuul real es trivial de implementar. La siguiente lista muestra el código fuente utilizado para crear el filtro Zuul. El archivo se encuentra en zuulsvr/src/main/java/com/thinktmechanix/zuulsvr/.filtros/ResponseFilter.java.

## Listado 9.2 Agregar el ID de seguimiento de Spring Cloud Sleuth mediante un filtro POST de Zuul

```

paquete com.thinkmechanix.zuulsvr.filters;

//El resto de las anotaciones se eliminaron por razones de concisión.
importar org.springframework.cloud.sleuth.Tracer;

@Componente
La clase pública ResponseFilter extiende ZuulFilter{
int final estática privada           FILTER_ORDER=1;
booleano final estático privado SHOULD_FILTER=true;
registrador final estático privado registrador =
    LoggerFactory.getLogger(ResponseFilter.clase);

@autocableado
trazador trazador;                  ←
                                         | La clase Tracer es el punto de entrada
                                         | para acceder a la información de
                                         | ID de seguimiento y tramo.

@Anular
tipo de filtro de cadena pública() {return "publicación";}

@Anular
public int filterOrder() {return FILTER_ORDER;}

@Anular
público booleano deberíaFilter() {regresar DEBE_FILTER;}

@Anular
Ejecución pública de objetos() {
    RequestContext ctx = RequestContext.getCurrentContext();
    ctx.getResponse()
        .addHeader("tmx-correlación-id",
        tracer.getCurrentSpan().traceldString());

    devolver nulo;
}
                                         | Vas a agregar un
                                         | nueva respuesta HTTP
                                         | encabezado llamado
                                         | tmx-correlation-ID para contener
                                         | la nube de primavera
                                         | ID de rastreo del detective.
                                         ←

```

Debido a que Zuul ahora está habilitado para Spring Cloud Sleuth, puede acceder a la información de seguimiento desde dentro de su ResponseFilter mediante el cableado automático en la clase Tracer al Filtro de respuesta. La clase Tracer le permite acceder a información sobre el seguimiento actual de Spring Cloud Sleuth que se está ejecutando. El rastreador.getCurrentSpan() El método .traceldString() le permite recuperar como cadena el ID de seguimiento actual para la transacción en curso.

Es trivial agregar el ID de seguimiento a la respuesta HTTP saliente que pasa de regreso. Zuul. Esto se hace llamando

```

RequestContext ctx = RequestContext.getCurrentContext();
ctx.getResponse().addHeader("tmx-correlation-id",
    tracer.getCurrentSpan().traceldString());

```

Con este código ahora implementado, si invoca un microservicio EagleEye a través de su puerta de enlace Zuul, debería recibir una respuesta HTTP llamada tmx-correlation-id.

The screenshot shows a Postman interface with a GET request to `http://localhost:5555/api/licensing/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a`. The 'Headers' tab is selected, displaying five headers: Content-Type, Date, Transfer-Encoding, X-Application-Context, and tmx-correlation-id. An annotation with an arrow points from the text 'El ID de seguimiento de Spring Cloud Sleuth.' to the 'tmx-correlation-id' value 'fbc78ad2917015de'. Below the value, the text 'Ahora puede utilizar esto para consultar Papertrail.' is displayed.

Figura 9.10 Con el ID de seguimiento de Spring Cloud Sleuth devuelto, puede consultar fácilmente los registros en Papertrail.

La Figura 9.10 muestra los resultados de una llamada a GET `http://localhost:5555/api/licensing/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a`.

### 9.3 Seguimiento distribuido con Open Zipkin

Tener una plataforma de registro unificada con ID de correlación es una poderosa herramienta de depuración. Sin embargo, durante el resto del capítulo dejaremos de rastrear las entradas del registro y, en su lugar, veremos cómo visualizar el flujo de transacciones a medida que se mueven a través de diferentes microservicios. Una imagen limpia y concisa puede funcionar con más de un millón de entradas de registro.

El seguimiento distribuido implica proporcionar una imagen visual de cómo fluye una transacción entre sus diferentes microservicios. Las herramientas de seguimiento distribuido también brindarán una aproximación aproximada de los tiempos de respuesta de los microservicios individuales. Sin embargo, las herramientas de seguimiento distribuidas no deben confundirse con los paquetes completos de gestión del rendimiento de aplicaciones (APM). Estos paquetes pueden proporcionar datos de rendimiento de bajo nivel listos para usar sobre el código real dentro de su servicio y también pueden proporcionar datos de rendimiento más allá del tiempo de respuesta, como memoria, utilización de CPU y utilización de E/S.

Aquí es donde brillan Spring Cloud Sleuth y el proyecto OpenZipkin (también conocido como Zipkin). Zipkin (<http://zipkin.io/>) es una plataforma de seguimiento distribuida que le permite rastrear transacciones en múltiples invocaciones de servicios. Zipkin le permite ver gráficamente la cantidad de tiempo que lleva una transacción y desglosa el tiempo invertido en cada microservicio involucrado en la llamada. Zipkin es una herramienta invaluable para identificar problemas de rendimiento en una arquitectura de microservicios.

La configuración de Spring Cloud Sleuth y Zipkin implica cuatro actividades:

Agregar archivos Spring Cloud Sleuth y Zipkin JAR a los servicios que capturan datos de seguimiento

Configurar una propiedad Spring en cada servicio para que apunte al servidor Zipkin que recogerá los datos de seguimiento

Instalación y configuración de un servidor Zipkin para recopilar los datos.

Definir la estrategia de muestreo que cada cliente utilizará para enviar información de seguimiento hacia Zipkin

### 9.3.1 Configuración de las dependencias de Spring Cloud Sleuth y Zipkin

Hasta ahora ha incluido dos conjuntos de dependencias de Maven en su Zuul, licencias, y servicios de organización. Estos archivos JAR eran las dependencias spring-cloud-starter-sleuth y spring-cloud-sleuth-core . Las dependencias de spring-cloud-starter-sleuth se utilizan para incluir el Spring Cloud Sleuth básico.

bibliotecas necesarias para habilitar Spring Cloud Sleuth dentro de un servicio. Las dependencias spring-cloud-sleuth-core se utilizan siempre que tenga que interactuar mediante programación con Spring Cloud Sleuth (lo que volverá a hacer más adelante en este capítulo).

Para integrarse con Zipkin, necesita agregar una segunda dependencia de Maven llamada primavera-nube-detective-zipkin . El siguiente listado muestra las entradas de Maven que debe estar presente en los servicios de organización, licencias y Zuul una vez que se agrega la dependencia spring-cloud-sleuth-zipkin .

#### Listado 9.3 Dependencias de Spring Cloud Sleuth y Zipkin del lado del cliente

```
<dependencia>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>detective-iniciador-de-nube-primavera</artifactId>
</dependencia>
<dependencia>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>primavera-nube-detective-zipkin</artifactId>
</dependencia>
```

### 9.3.2 Configurar los servicios para que apunten a Zipkin

Con los archivos JAR en su lugar, necesita configurar cada servicio que quiera comunicarse con Zipkin. Para ello, establezca una propiedad Spring que defina la URL utilizada .

para comunicarse con Zipkin. La propiedad que hay que establecer es el resorte.

Propiedad .zipkin.baseUrl . Esta propiedad se establece en el archivo application.yml de cada servicio. archivo de propiedades.

**NOTA** Spring.zipkin.baseUrl también se puede externalizar como una propiedad en la configuración de Spring Cloud.

En el archivo application.yml para cada servicio, el valor se establece en http://local-host:9411. Sin embargo, en tiempo de ejecución anula este valor usando ZIPKIN\_URI

(<http://zipkin:9411>) variable pasada en cada configuración de Docker de servicios (docker/common/docker-compose.yml).

### Zipkin, RabbitMQ y Kafka

Zipkin tiene la capacidad de enviar sus datos de rastreo a un servidor Zipkin a través de RabbitMQ o Kafka. Desde una perspectiva de funcionalidad, no hay diferencia en el comportamiento de Zipkin si utiliza HTTP, RabbitMQ o Kafka. Con el seguimiento HTTP, Zipkin utiliza un hilo asíncrono para enviar datos de rendimiento. La principal ventaja de usar RabbitMQ o Kafka para recopilar sus datos de rastreo es que si su servidor Zipkin no funciona, cualquier mensaje de rastreo enviado a Zipkin se "pondrá en cola" hasta que Zipkin pueda recoger los datos.

La configuración de Spring Cloud Sleuth para enviar datos a Zipkin a través de RabbitMQ y Kafka está cubierto en la documentación de Spring Cloud Sleuth, por lo que no lo cubriremos aquí con más detalle.

### 9.3.3 Instalación y configuración de un servidor Zipkin

Para usar Zipkin, primero debe configurar un proyecto Spring Boot como lo hizo varias veces a lo largo del libro. (En el código del capítulo, esto se llama zipkinsvr.) Luego deberá agregar dos dependencias JAR al archivo zipkinsvr/pom.xml archivo. Estas dos dependencias de jar se muestran en la siguiente lista.

Listado 9.4 Dependencias JAR necesarias para el servicio Zipkin

```
<dependencia>
    <groupId>io.zipkin.java</groupId>
    <artifactId>servidor-zipkin</artifactId>
</dependencia>
<dependencia>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-autoconfigure-ui</artifactId>
</dependencia>
```

Esta dependencia contiene las clases principales para configurar el servidor Zipkin.

Esta dependencia contiene el núcleo de las clases para ejecutar la parte de la interfaz de usuario del servidor Zipkin.

### @EnableZipkinServer frente a @EnableZipkinStreamServer: ¿qué anotación?

Una cosa a tener en cuenta sobre las dependencias JAR anteriores es que no son dependencias basadas en Spring-Cloud. Si bien Zipkin es un proyecto basado en Spring-Boot, el `@EnableZipkinServer` no es una anotación de Spring Cloud. Es una anotación que es parte del proyecto Zipkin. Esto a menudo confunde a las personas que son nuevas en Spring Cloud. Sleuth y Zipkin, porque el equipo de Spring Cloud escribió la anotación `@EnableZipkin-StreamServer` como parte de Spring Cloud Sleuth. La anotación `@EnableZipkin-StreamServer` simplifica el uso de Zipkin con RabbitMQ y Kafka.

Elegí usar `@EnableZipkinServer` debido a su simplicidad en la configuración para Este capítulo. Con el servidor `@EnableZipkinStream` necesita configurar y configurar los servicios que se están rastreando y el servidor Zipkin para publicar/escuchar RabbitMQ.

o Kafka para rastrear datos. La ventaja de la anotación `@EnableZipkinStreamServer` es que puede continuar recopilando datos de seguimiento incluso si el servidor Zipkin no está disponible. Esto se debe a que los mensajes de seguimiento acumularán los datos de seguimiento en una cola de mensajes hasta que el servidor Zipkin esté disponible para procesar los registros. Si utiliza la anotación `@EnableZipkinServer` y el servidor Zipkin no está disponible, se perderán los datos de seguimiento que los servicios habrían enviado a Zipkin.

Una vez definidas las dependencias de Jar, ahora necesita agregar la anotación `@EnableZipkin Server` a su clase de arranque de servicios Zipkin. Esta clase se encuentra en `zipkinsvr/src/main/java/com/thinkmechanix/zipkinsvr/ZipkinServer Application.java`. El siguiente listado muestra el código de la clase `bootstrap`.

#### Listado 9.5 Construyendo la clase de arranque de su servidor Zipkin

```
paquete com.thinkmechanix.zipkinsvr;

importar org.springframework.boot.SpringApplication; importar
org.springframework.boot.autoconfigure.SpringBootApplication; importar zipkin.server.EnableZipkinServer;

@SpringBootApplication
@EnableZipkinServer clase
pública ZipkinServerApplication {public static void
main(String[] args) {
    SpringApplication.run(ZipkinServerApplication.clase, argumentos); }

}
```

← @EnableZipkinServer le permite iniciar rápidamente Zipkin como un proyecto Spring Boot.

Lo clave a tener en cuenta en esta lista es el uso de la anotación `@EnableZipkinServer`. Esta anotación le permite iniciar este servicio Spring Boot como un servidor Zipkin.

En este punto, puede construir, compilar e iniciar el servidor Zipkin como uno de los contenedores Docker del capítulo.

Se necesita poca configuración para ejecutar un servidor Zipkin. Una de las únicas cosas que tendrá que configurar cuando ejecute Zipkin es el almacén de datos de back-end que Zipkin utilizará para almacenar los datos de seguimiento de sus servicios. Zipkin admite cuatro almacenes de datos de back-end diferentes. Estos almacenes de datos son

#### 1 Datos en memoria 2

MySQL: <http://mysql.com> 3

Cassandra: <http://cassandra.apache.org> 4 Búsqueda

elástica: <http://elastic.co>

De forma predeterminada, Zipkin utiliza un almacén de datos en memoria para almacenar datos de seguimiento. El equipo de Zipkin recomienda no utilizar la base de datos en memoria en un sistema de producción. La base de datos en memoria puede contener una cantidad limitada de datos y los datos se pierden cuando el servidor Zipkin se apaga o se pierde.

**NOTA** Para los fines de este libro, utilizará Zipkin con datos en memoria. almacenar. La configuración de los almacenes de datos individuales utilizados en Zipkin está fuera del alcance de este libro, pero si está interesado en el tema, puede encontrar más información en el repositorio Zipkin GitHub (<https://github.com/openzipkin/zipkin/tree/master/zipkin-server>).

#### 9.3.4 Configuración de niveles de seguimiento

En este punto, tiene los clientes configurados para hablar con un servidor Zipkin y tiene la Servidor configurado y listo para ser ejecutado. Necesitas hacer una cosa más antes de comenzar. utilizando Zipkin. Debe definir con qué frecuencia cada servicio debe escribir datos en Zipkin.

De forma predeterminada, Zipkin solo escribirá el 10% de todas las transacciones en el servidor Zipkin. El muestreo de transacciones se puede controlar estableciendo una propiedad Spring en cada uno de los servicios de envío de datos a Zipkin. Esta propiedad se llama `spring.sleuth.sampler.percentage`. La propiedad toma un valor entre 0 y 1:

Un valor de 0 significa que Spring Cloud Sleuth no enviará ninguna transacción a Zipkin.

Un valor de 0,5 significa que Spring Cloud Sleuth enviará el 50% de todas las transacciones.

Para nuestros propósitos, enviará información de seguimiento para todos los servicios. Para hacer esto, usted Puede establecer el valor de `spring.sleuth.sampler.percentage` o puede reemplazar el Clase Sampler predeterminada utilizada en Spring Cloud Sleuth con AlwaysSampler. El AlwaysSampler se puede injectar como Spring Bean en una aplicación. Por ejemplo, el El servicio de licencias tiene AlwaysSampler definido como Spring Bean en su licencia-service/src/main/java/com/thinktmechanix/licenses/Application.java. clase como

@Frijol

```
muestra pública defaultSampler() { return new AlwaysSampler();}
```

Los servicios Zuul, de licencias y de organización tienen definido AlwaysSampler en ellos para que en este capítulo todas las transacciones se rastreen con Zipkin.

#### 9.3.5 Uso de Zipkin para rastrear transacciones

Comencemos esta sección con un escenario. Imagina que eres uno de los desarrolladores de Aplicación EagleEye y estarás de guardia esta semana. Recibe un ticket de soporte de un cliente que se queja de que una de las pantallas de la aplicación EagleEye funciona con lentitud. Tiene la sospecha de que el servicio de licencias que utiliza la pantalla es corriendo lento. ¿Pero por qué y dónde? El servicio de licencia depende del servicio de organización y ambos servicios realizan llamadas a bases de datos diferentes. ¿Qué servicio es el de peor desempeño? Además, sabes que estos servicios se modifican constantemente, por lo que alguien podría haber agregado una nueva llamada de servicio a la mezcla. Comprender todos los servicios que participar en la transacción del usuario y sus tiempos de rendimiento individuales es fundamental para soportar una arquitectura distribuida como una arquitectura de microservicio.

Comenzará utilizando Zipkin para observar dos transacciones del servicio de su organización a medida que el servicio Zipkin las rastrea. El servicio de organización es un servicio sencillo.

que solo realiza una llamada a una única base de datos. Lo que vas a hacer es usar POSTMAN para enviar dos llamadas al servicio de la organización (GET <http://localhost:5555/api/organización/v1/organizaciones/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a>).

Las llamadas al servicio de la organización fluirán a través de una puerta de enlace API de Zuul antes de que lleguen las llamadas. dirigido en sentido descendente a una instancia de servicio de la organización.

Después de haber realizado dos llamadas al servicio de la organización, vaya a <http://localhost:9411> y vea lo que Zipkin ha capturado para obtener resultados de seguimiento. Seleccione la "organización servicio" en el cuadro desplegable en el extremo superior izquierdo de la pantalla y luego presione el botón Buscar rastros. La Figura 9.11 muestra la pantalla de consulta Zipkin después de haber tomado estas acciones.

Ahora, si miras la captura de pantalla en la figura 9.11, verás que Zipkin capturó dos actas. Cada una de las transacciones se divide en uno o más tramos. En Zipkin, un lapso representa un servicio o llamada específica en la que se envía información de tiempo.

capturado. Cada una de las transacciones en la figura 9.11 tiene tres tramos capturados en ella: dos tramos en la puerta de enlace de Zuul y luego un tramo para el servicio de organización. Recordar, la puerta de enlace Zuul no reenvía ciegamente una llamada HTTP . Recibe el HTTP entrante. llamada, finaliza la llamada entrante y luego crea una nueva llamada al servicio de destino (en este caso, el servicio de la organización). Esta terminación de la llamada original es cómo Zuul puede agregar filtros previos, de respuesta y posteriores a cada llamada que ingresa a la puerta de enlace. Es también por qué vemos dos tramos en el servicio Zuul.

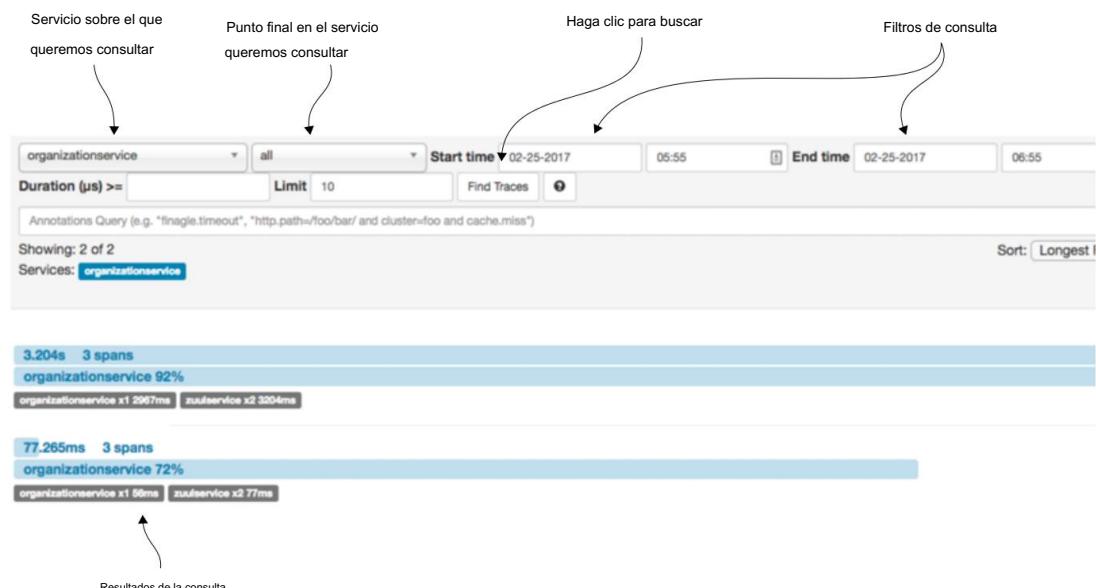
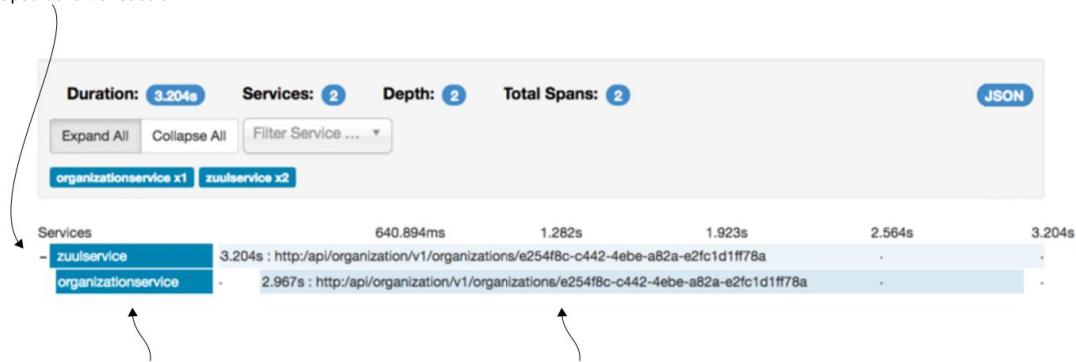


Figura 9.11 La pantalla de consulta de Zipkin le permite seleccionar el servicio que desea rastrear, junto con algunos filtros de consulta básicos.

Las dos llamadas al servicio de organización a través de Zuul tardaron 3.204 segundos y 77,2365 milisegundos respectivamente. Porque consultaste en el servicio de organización llamadas (y no las llamadas de la puerta de enlace de Zuul), puedes ver que el servicio de la organización tomó 92% y 72% del tiempo total de la transacción.

Profundicemos en los detalles de la llamada de mayor duración (3,204 segundos). Puedes ver más detalles haciendo clic en la transacción y profundizando en los detalles. Figura 9.12 muestra los detalles después de haber hecho clic para profundizar en más detalles.

Una transacción se divide en tramos individuales.  
Un lapso representa parte de la transacción que se está midiendo. Aquí se muestra el tiempo total de cada lapso de la transacción.



Al profundizar en una de las transacciones, se ven dos períodos: uno para el tiempo pasado en Zuul y otro para el tiempo pasado en el servicio de organización.

Al hacer clic en un tramo individual, aparecerán detalles adicionales sobre el tramo.

Figura 9.12 Zipkin le permite profundizar y ver la cantidad de tiempo que lleva cada lapso en una transacción.

En la figura 9.12 se puede ver que toda la transacción desde una perspectiva Zuul tomó aproximadamente 3.204 segundos. Sin embargo, la llamada de servicio de organización realizada por Zuul tardó 2.967 segundos de los 3.204 segundos que intervienen en la convocatoria general. Cada tramo presentado se puede profundizar para obtener aún más detalles. Haga clic en el intervalo de organización-servicio y vea qué detalles adicionales se pueden ver en la llamada. Figura 9.13 muestra el detalle de esta convocatoria.

Uno de los datos más valiosos de la figura 9.13 es el desglose de cuando el cliente (Zuul) llamó al servicio de la organización, cuando el servicio de la organización recibió la llamada y cuándo respondió el servicio de la organización. Este tipo de información de sincronización es invaluable para detectar e identificar problemas de latencia de la red.

organizationservice.http://api/organization/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a

2.967s

AKA: zuulservice, organizationservice

Date Time	Relative Time	Annotation	Address
2/25/2017, 6:53:57 AM	162.000ms	Client Send	172.18.0.5:5555 (zuulservice)
2/25/2017, 6:53:58 AM	1.322s	Server Receive	172.18.0.11:8085 (organizationservice)
2/25/2017, 6:53:59 AM	1.969s	Server Send	172.18.0.11:8085 (organizationservice)
2/25/2017, 6:54:00 AM	3.129s	Client Receive	172.18.0.5:5555 (zuulservice)

Al hacer clic en los detalles, puede ver cuándo Zuul llamó al servicio de la organización, cuándo

El servicio de la organización recibió la solicitud y cuándo el cliente recibió la solicitud.

Key	Value
http.method	GET
http.path	/api/organization/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a
http.status_code	200
http.url	/api/organization/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a
Local Component	zuul
mvc.controller.class	OrganizationServiceController
mvc.controller.method	getOrganization

Al hacer clic en los detalles también proporciona algunos detalles básicos sobre la llamada HTTP.

Figura 9.13 Al hacer clic en un intervalo individual se obtienen más detalles sobre el tiempo de llamada y los detalles de la llamada HTTP.

### 9.3.6 Visualización de una transacción más compleja

¿Qué sucede si desea comprender exactamente qué dependencias de servicio existen entre las llamadas de servicio? Puede llamar al servicio de licencias a través de Zuul y luego consultar a Zipkin para obtener rastros del servicio de licencias. Puede hacerlo con una llamada GET a los servicios de licencias. <http://localhost:5555/api/licensing/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a> punto final.

La Figura 9.14 muestra el seguimiento detallado de la llamada al servicio de licencias.

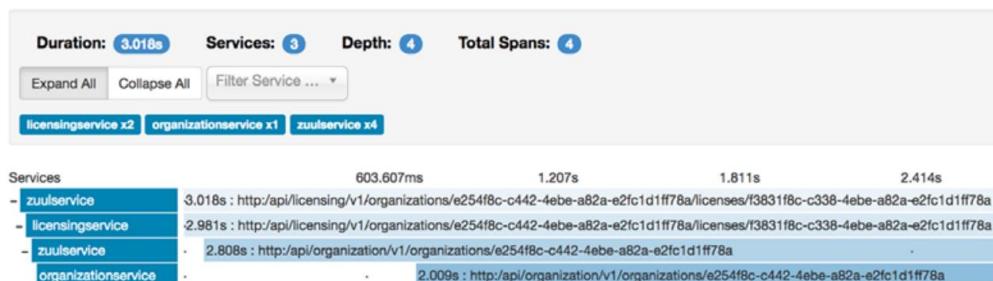


Figura 9.14 Ver los detalles de un seguimiento de cómo la llamada del servicio de licencias fluye desde Zuul al servicio de licencias y luego al servicio de la organización

En la figura 9.14, puede ver que la llamada al servicio de licencias involucra 4 discretos Llamadas HTTP . Verá la llamada a la puerta de enlace Zuul y luego desde la puerta de enlace Zuul al servicio de licencias. Luego, el servicio de licencias vuelve a llamar a través de Zuul para llamar al servicio de organización.

### 9.3.7 Captura de seguimientos de mensajes

Spring Cloud Sleuth y Zipkin no rastrean las llamadas HTTP . Spring Cloud Sleuth también envía datos de seguimiento de Zipkin en cualquier canal de mensajes entrantes o salientes registrado en el servicio.

La mensajería puede introducir sus propios problemas de rendimiento y latencia dentro de una aplicación. Es posible que un servicio no esté procesando un mensaje de una cola con la suficiente rapidez. O Podría haber un problema de latencia de la red. Me he encontrado con todos estos escenarios mientras Creación de aplicaciones basadas en microservicios.

Al utilizar Spring Cloud Sleuth y Zipkin, puede identificar cuándo se publica un mensaje desde una cola y cuándo se recibe. También puedes ver qué comportamiento tiene lugar. cuando el mensaje se recibe en una cola y se procesa.

Como recordará del capítulo 8, cada vez que se agrega un registro de organización, actualizado o eliminado, un mensaje de Kafka se produce y publica a través de Spring Cloud Arroyo. El servicio de licencias recibe el mensaje y actualiza un almacén de valores clave de Redis. está usando para almacenar datos en caché.

Ahora continuará, eliminará un registro de la organización y observará la transacción. ser rastreado por Spring Cloud Sleuth y Zipkin. Puede emitir un DELETE `http://localhost:5555/api/organization/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a` vía CARTERO al servicio de organización.

Recuerde, anteriormente en el capítulo vimos cómo agregar el ID de seguimiento como HTTP encabezado de respuesta. Agregó un nuevo encabezado de respuesta HTTP llamado tmx-correlation-id. En mi llamada, obtuve el tmx-correlation-id devuelto en mi llamada con un valor de `5e14cae0d90dc8d4`. Puede buscar en Zipkin este ID de seguimiento específico ingresando el ID de rastreo devuelto por su llamada a través del cuadro de búsqueda en la esquina superior derecha de la Pantalla de consulta Zipkin. La Figura 9.15 muestra dónde puede ingresar el ID de seguimiento.

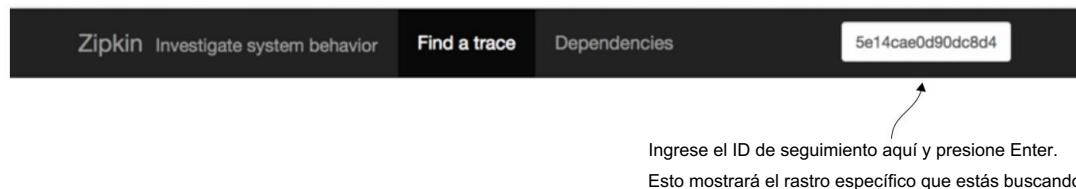


Figura 9.15 Con el ID de seguimiento devuelto en el campo tmx-correlation-id de la respuesta HTTP , puede encontrar fácilmente la transacción que está buscando.

Con el ID de seguimiento en la mano, puede consultar a Zipkin para la transacción específica y puede ver la publicación de un mensaje de eliminación para cambiar su mensaje de salida. Este mensaje El canal, salida, se utiliza para publicar en un tema de Kafka llamado orgChangeTopic. Cifra 9.16 muestra el canal del mensaje de salida y cómo aparece en el seguimiento de Zipkin.

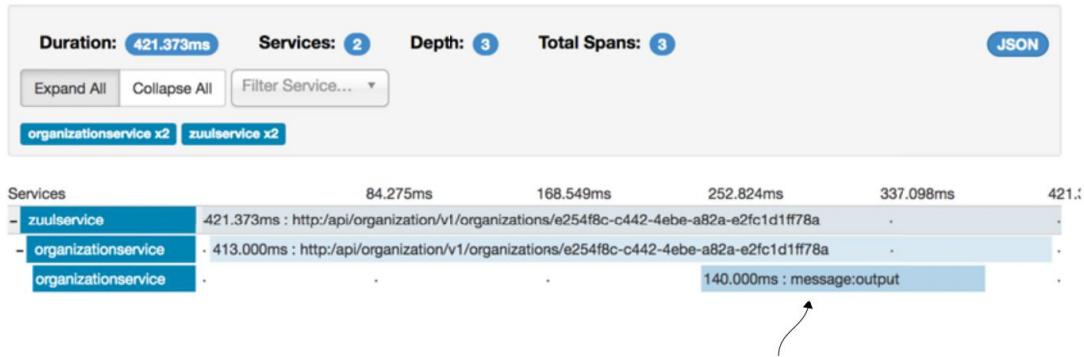


Figura 9.16 Spring Cloud Sleuth rastreará automáticamente la publicación y recepción de mensajes en los canales de mensajes de Spring.

Puede ver que el servicio de licencias recibe el mensaje consultando Zipkin y buscando para el mensaje recibido. Desafortunadamente, Spring Cloud Sleuth no propaga el ID de seguimiento de un mensaje publicado para los consumidores de ese mensaje. En lugar de ello, genera un nuevo ID de seguimiento. Sin embargo, puede consultar el servidor Zipkin sobre cualquier transacción de servicio de licencia y ordenarla por mensaje más reciente. La Figura 9.17 muestra los resultados de esta consulta.

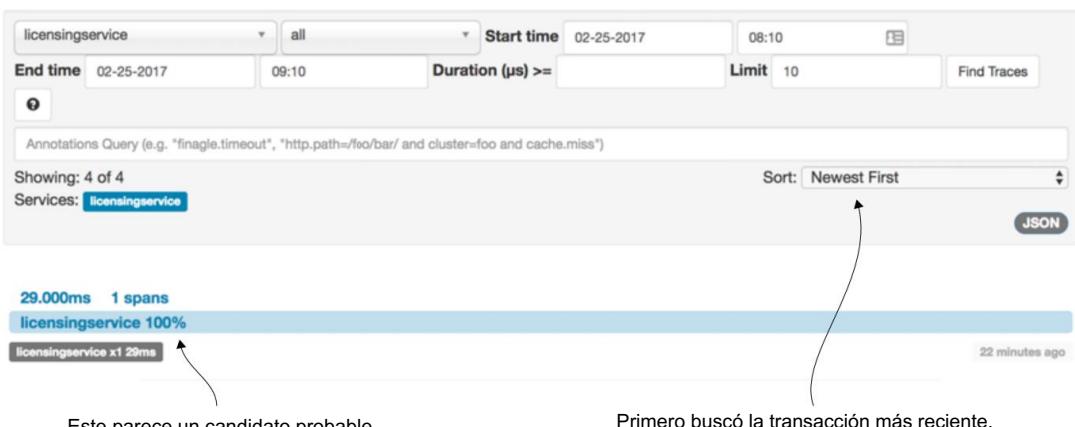


Figura 9.17 Está buscando la invocación del servicio de licencias donde se recibe un mensaje de Kafka.

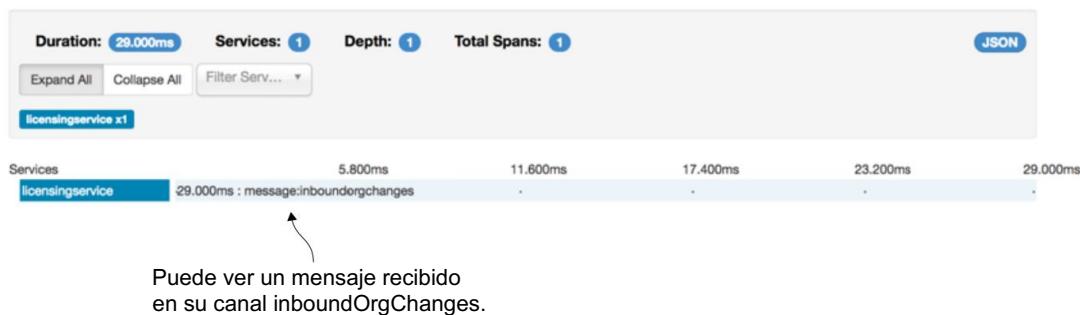


Figura 9.18 Usando Zipkin puede ver el mensaje de Kafka publicado por el servicio de la organización.

Ahora que ha encontrado su transacción de servicio de licencia objetivo, puede profundizar en la transacción. La Figura 9.18 muestra los resultados de este desglose.

Hasta ahora has utilizado Zipkin para rastrear tus llamadas HTTP y de mensajería desde dentro. sus servicios. Sin embargo, ¿qué sucede si desea realizar seguimientos a servicios de terceros? que no están instrumentados por Zipkin? Por ejemplo, ¿qué sucede si desea realizar un seguimiento y ¿información de tiempo para una llamada SQL específica de Redis o Postgres? Afortunadamente, la primavera Cloud Sleuth y Zipkin le permiten agregar intervalos personalizados a su transacción para que pueda rastrear el tiempo de ejecución asociado con estas llamadas de terceros.

### 9.3.8 Agregar tramos personalizados

Agregar un tramo personalizado es increíblemente fácil de hacer en Zipkin. Puede comenzar agregando un intervalo personalizado a su servicio de licencias para poder rastrear cuánto tiempo lleva extraer datos. fuera de Redis. Luego, agregará un intervalo personalizado al servicio de organización para vea cuánto tiempo lleva recuperar datos de la base de datos de su organización.

Para agregar un intervalo personalizado a la llamada del servicio de licencias a Redis, vas a instrumente el servicio de licencias/src/main/java/com/thinkmechanix/licencias/clientes/OrganizationRestTemplateClient.java clase. En esto clase en la que vas a instrumentar el método checkRedisCache(). La siguiente El listado muestra este código.

#### Listado 9.6 Instrumentación de la llamada para leer datos de licencia de Redis

```
import org.springframework.cloud.sleuth.Tracer;
//El resto de las importaciones se eliminaron por motivos de concisión.
@Component
clase pública OrganizationRestTemplateClient {
    @autocableado
    RestTemplate restTemplate;
    @autocableado
    trazador trazador;
    @autocableado
    OrganizaciónRedisRepository orgRedisRepo;
```

La clase Tracer se utiliza para acceder mediante programación a la información de seguimiento de Spring Cloud Sleuth.

```

    registrador final estático privado registrador =
        LoggerFactory
            .getLogger(OrganizationRestTemplateClient.clase);

    organización privada checkRedisCache (String organizaciónId) {
        Span newSpan = tracer.createSpan("readLicensingDataFromRedis"); intente { return
            orgRedisRepo.findOrganization(organizationId);

        } catch (Exception ex)
            { logger.error("Error encontrado al intentar recuperar
                la organización {} verificar Redis Cache. Excepción
                {}", OrganizationId, ex);

        Cierra el tramo
        con un
        bloque final. devolver nulo;
    } finalmente
        { newSpan.tag("peer.service", "redis"); nuevoSpan.logEvent(
            org.springframework.cloud.sleuth.Span.CLIENT_RECV); trazador.close(newSpan);
        }
    }

    //El resto de la clase se eliminó por razones de concisión.
}

Cierra el rastro. Si no llama al método close(), recibirá
mensajes de error en los registros que indican que se ha
dejado abierto un intervalo.

```

Para su intervalo personalizado, cree un nuevo intervalo llamado "readLicensingDataFromRedis".

Puede agregar información de etiqueta al intervalo. En esta clase, usted proporciona el nombre del servicio que Zipkin capturará.

Registre un evento para indicarle a Spring Cloud Sleuth que debe capturar el momento en que se completa la llamada.

El código del listado 9.6 crea un intervalo personalizado llamado readLicensingDataFromRedis. Ahora también agregará un intervalo personalizado, llamado getOrgDbCall, al servicio de la organización para monitorear cuánto tiempo lleva recuperar los datos de la organización de la base de datos de Postgres. El seguimiento de las llamadas a la base de datos del servicio de la organización se puede ver en la clase organización-servicio/src/main/java/com/thinkmechanix/organization/services/OrganizationService.java . El método que contiene el seguimiento personalizado es la llamada al método getOrg() .

La siguiente lista muestra el código fuente del método getOrg() del servicio de la organización .

#### Listado 9.7 El método getOrg() instrumentado

```

paquete com.thinkmechanix.organization.services;

//Se eliminaron las importaciones por razones de concisión.
@Service
Servicio de organización de clase pública {
    @autocableado
    Repositorio de organización privado orgRepository;

    @autocableado
    rastreador Tracer privado;

```

```

@autocableado
SimpleSourceBean simpleSourceBean;

registrar final estático privado registrar =
    LoggerFactory.getLogger(OrganizationService.clase);

Organización pública getOrg (String organizaciónId) {
Span newSpan = tracer.createSpan("getOrgDBCall");

logger.debug("En la llamada organizaciónService.getOrg()"); intente { return

orgRepository.findById(organizationId); }finalmente{ newSpan.tag("peer.service",
"postgres");
newSpan .logEvent( org.springframework.cloud.sleuth.Span.CLIENT_RECV);

trazador.close(newSpan);

}

//Se eliminó el código para mayor concisión
}

```

Con estos dos intervalos personalizados implementados, reinicie los servicios y luego presione OBTENER `http://localhost:5555/api/licensing/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-` punto final `a82a-e2fc1d1ff78a`. Si observamos la transacción en Zipkin, debería ver la suma de los dos tramos adicionales. La Figura 9.19 muestra los intervalos personalizados adicionales agregados cuando llama al punto final del servicio de licencias para recuperar información de licencias.

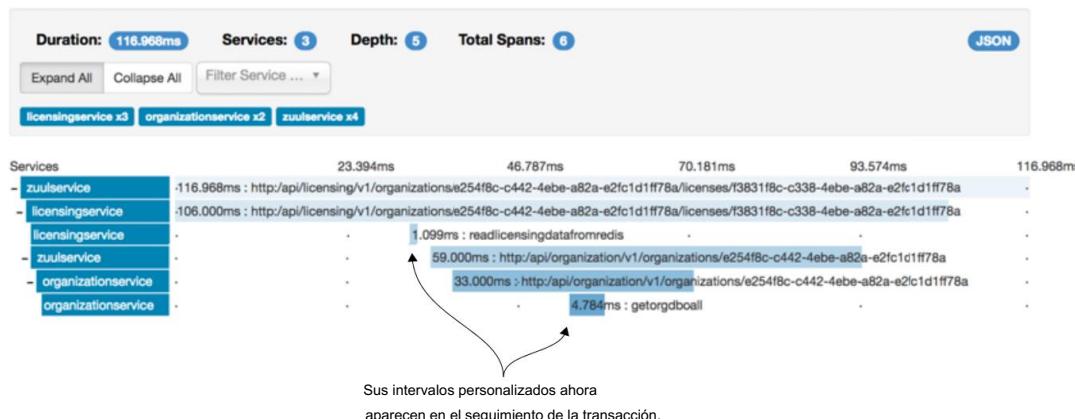


Figura 9.19 Con los intervalos personalizados definidos, ahora aparecerán en el seguimiento de la transacción.

En la figura 9.19, ahora puede ver información adicional de seguimiento y sincronización relacionada con sus búsquedas en Redis y bases de datos. Se puede decir que la llamada de lectura a Redis tomó 1,099 milisegundos. Como la llamada no encontró ningún elemento en la caché de Redis, la llamada SQL a la base de datos de Postgres tardó 4,784 milisegundos.

## 9.4 Resumen

Spring Cloud Sleuth le permite agregar sin problemas información de seguimiento (ID de correlación ) a sus llamadas de microservicio. Los ID de correlación se pueden utilizar para vincular entradas de registro entre múltiples servicios. Le permiten ver el comportamiento de una transacción en todos los servicios involucrados en una sola transacción. Si bien los ID de correlación son potentes, es necesario asociar este concepto con una plataforma de agregación de registros que le permitirá ingerir registros de múltiples fuentes y luego buscar y consultar su contenido. Si bien existen múltiples plataformas de agregación de registros locales, los servicios basados en la nube le permiten administrar sus registros sin tener que contar con una infraestructura extensa. También le permiten escalar fácilmente según el volumen de registro de su aplicación. crece. Puede integrar contenedores Docker con una plataforma de agregación de registros para capturar todos los datos de registro que se escriben en los contenedores stdout/stderr. En este capítulo, integró sus contenedores Docker con Logspout y un proveedor de registro en la nube en línea, Papertrail, para capturar y consultar sus registros. Si bien una plataforma de registro unificada es importante, la capacidad de rastrear visualmente una transacción a través de sus microservicios también es una herramienta valiosa. Zipkin le permite ver las dependencias que existen entre servicios cuando se realiza una llamada a un servicio.

Spring Cloud Sleuth se integra con Zipkin. Zipkin le permite ver gráficamente el flujo de sus transacciones y comprender las características de rendimiento de cada microservicio involucrado en la transacción de un usuario.

Spring Cloud Sleuth capturará automáticamente datos de seguimiento para una llamada HTTP y un canal de mensajes entrantes/salientes utilizados dentro de un servicio habilitado para Spring Cloud Sleuth.

Spring Cloud Sleuth asigna cada una de las llamadas de servicio al concepto de tramo. Cremallera-kin le permite ver el rendimiento de un tramo.

Spring Cloud Sleuth y Zipkin también le permiten definir sus propios intervalos personalizados para que pueda comprender el rendimiento de recursos no basados en Spring (un servidor de bases de datos como Postgres o Redis).

# 10

## Implementando su microservicios

### Este capítulo cubre

Comprender por qué el movimiento DevOps es crítico para los microservicios

Configuración de la infraestructura central de Amazon utilizada por los servicios EagleEye

Implementación manual de servicios EagleEye en Amazon

Diseñar un proceso de construcción e implementación para sus servicios.

Pasar de la integración continua al despliegue continuo

Tratar su infraestructura como código

Construyendo el servidor inmutable

Pruebas en implementación

Implementar su aplicación en la nube

Estamos al final del libro, pero no al final de nuestro viaje con los microservicios. Mientras La mayor parte de este libro se ha centrado en diseñar, construir y poner en funcionamiento microservicios basados en Spring utilizando la tecnología Spring Cloud, todavía no hemos tocado el tema. cómo construir e implementar microservicios. Crear una canalización de compilación e implementación

Puede parecer una tarea mundana, pero en realidad es una de las piezas más importantes de su arquitectura de microservicios.

¿Por qué? Recuerde, una de las ventajas clave de una arquitectura de microservicios es que los microservicios son pequeñas unidades de código que se pueden construir, modificar e implementar rápidamente en producción de forma independiente unas de otras. El pequeño tamaño del servicio significa que se pueden ofrecer nuevas funciones (y correcciones de errores críticos) con un alto grado de velocidad. Velocidad es la palabra clave aquí porque implica que existe poca o ninguna fricción entre crear una nueva característica o corregir un error e implementar su servicio. Los plazos de entrega para la implementación deben ser de minutos, no de días.

Para lograr esto, el mecanismo que utiliza para crear e implementar su código necesita ser

Automatizado: cuando crea su código, no debe haber intervención humana en el proceso de compilación e implementación, particularmente en los entornos inferiores.

El proceso de creación del software, aprovisionamiento de una imagen de máquina y luego implementación del servicio debe automatizarse y debe iniciarse mediante el acto de enviar el código al repositorio fuente.

Repetible: el proceso que utiliza para compilar e implementar su software debe ser repetible para que suceda lo mismo cada vez que se inicia una compilación e implementación. La variabilidad en su proceso es a menudo la fuente de errores sutiles que son difíciles de localizar y resolver.

Completo: el resultado del artefacto implementado debe ser una máquina virtual completa o una imagen de contenedor (Docker) que contenga el entorno de ejecución "completo" para el servicio. Este es un cambio importante en la forma de pensar sobre su infraestructura. El aprovisionamiento de las imágenes de su máquina debe automatizarse completamente mediante scripts y mantenerse bajo control de fuente con el código fuente del servicio.

En un entorno de microservicios, esta responsabilidad generalmente pasa del equipo de operaciones al equipo de desarrollo propietario del servicio. Recuerde, uno de los inquilinos principales del desarrollo de microservicios es dejar que la responsabilidad operativa total del servicio recaiga en los desarrolladores.

Inmutable: una vez creada la imagen de la máquina que contiene su servicio, la configuración de tiempo de ejecución de la imagen no debe modificarse ni cambiarse después de que se haya implementado la imagen. Si es necesario realizar cambios, la configuración debe realizarse en los scripts mantenidos bajo control de código fuente y el servicio y la infraestructura deben pasar por el proceso de compilación nuevamente.

Los cambios en la configuración del tiempo de ejecución (configuraciones de recolección de basura, uso del perfil Spring) deben pasarse como variables de entorno a la imagen, mientras que la configuración de la aplicación debe mantenerse separada del contenedor (Spring Cloud Config).

Crear un canal de implementación de compilación sólido y generalizado es una cantidad significativa de trabajo y, a menudo, está diseñado específicamente para el entorno de ejecución que ejecutarán sus servicios. A menudo implica un equipo especializado de ingenieros de DevOps (operaciones de desarrollo) cuyo único trabajo es generalizar el proceso de construcción para que cada equipo puedan construir sus microservicios sin tener que reinventar todo el proceso de construcción para ellos mismos. Desafortunadamente, Spring es un marco de desarrollo y no ofrece una cantidad significativa de capacidades para implementar un proceso de compilación e implementación.

Para este capítulo, veremos cómo implementar una compilación e implementación. canalización utilizando una serie de herramientas que no son de Spring. Vas a tomar la suite de microservicios que ha estado creando para este libro y haga lo siguiente:

- 1 Integre los scripts de compilación de Maven que ha estado utilizando en una integración continua. herramienta en la nube de creación/implementación llamada Travis CI
- 2 Cree imágenes Docker inmutables para cada servicio y envíe esas imágenes a un repositorio centralizado
- 3 Implemente todo el conjunto de microservicios en la nube de Amazon utilizando EC2 de Amazon Servicio de contenedores (ECS)
- 4 Ejecute pruebas de plataforma que probarán que el servicio está funcionando correctamente

Quiero comenzar nuestra discusión con el objetivo final en mente: un conjunto implementado de servicios para Servicio de contenedor elástico de AWS (ECS). Antes de entrar en todos los detalles de cómo estás Si vamos a implementar un proceso de construcción/implementación, veamos cómo se verán los servicios Eagle-Eye ejecutándose en la nube de Amazon. Luego discutiremos cómo implementar manualmente los servicios EagleEye en la nube de AWS . Una vez hecho esto, lo haremos automatizar todo el proceso.

## 10.1 EagleEye: configurando su infraestructura central en la nube

A lo largo de todos los ejemplos de código de este libro, ha ejecutado todas sus aplicaciones dentro de una única imagen de máquina virtual con cada servicio individual ejecutándose como Docker envase. Ahora cambiará eso separando su servidor de base de datos (PostgreSQL ) y su servidor de almacenamiento en caché (Redis) de Docker a la nube de Amazon. Todos otros servicios permanecerán ejecutándose como contenedores Docker ejecutándose dentro de un solo nodo Clúster de Amazon ECS . La Figura 10.1 muestra la implementación de los servicios EagleEye en el Nube amazónica.

Repasemos la figura 10.1 y profundicemos en más detalles:

- 1 Todos sus servicios EagleEye (menos la base de datos y el clúster de Redis) están funcionando. para implementarse como contenedores Docker que se ejecutan dentro de un clúster ECS de un solo nodo. ECS configura y configura todos los servidores necesarios para ejecutar un clúster Docker. ECS también puede monitorear el estado de los contenedores que se ejecutan en Docker y reiniciarlos. servicios si el servicio falla.
- 2 Con la implementación en la nube de Amazon, se alejará de utilizando su propia base de datos PostgreSQL y servidor Redis y en su lugar utilice los servicios Amazon RDS y Amazon ElastiCache. Podrías continuar ejecutando el Almacenes de datos de Postgres y Redis en Docker, pero quería resaltar lo fácil que es

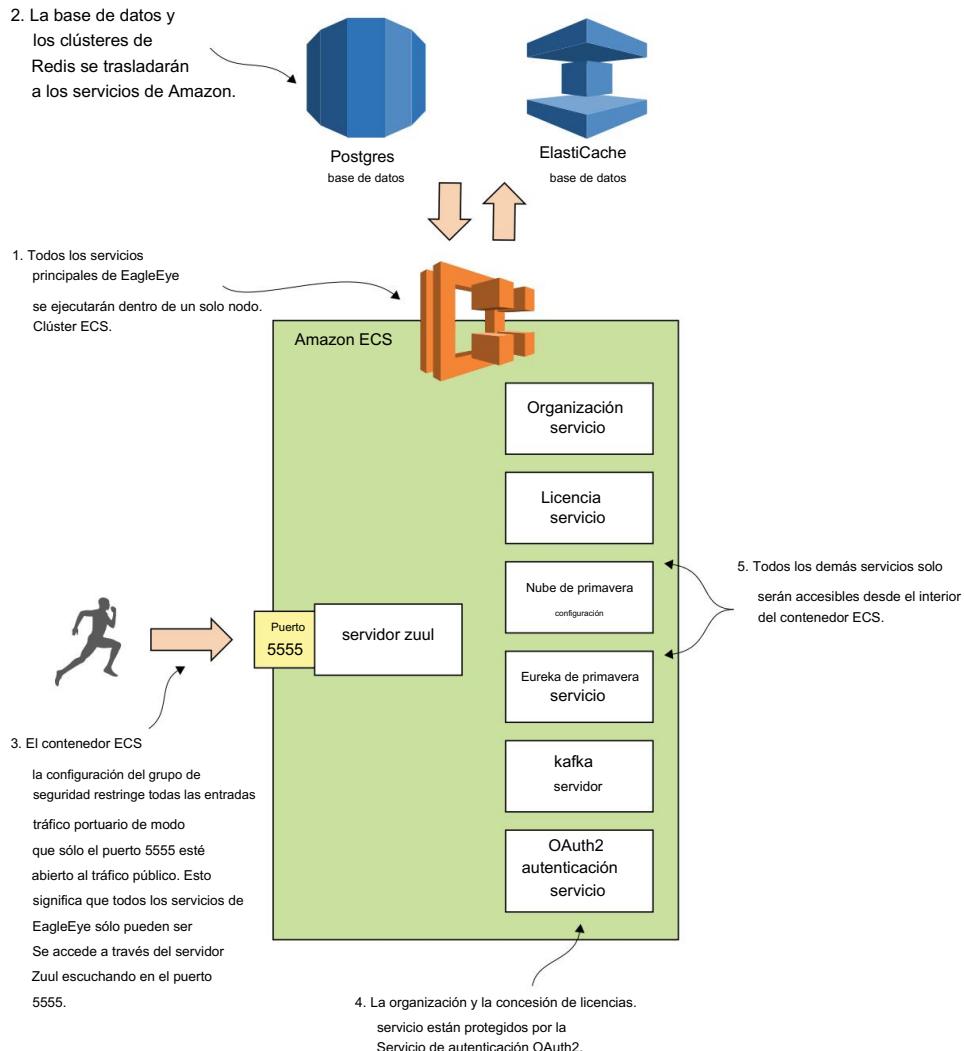


Figura 10.1 Al utilizar Docker, todos sus servicios se pueden implementar en un proveedor de nube como Amazon ECS.

para pasar de la infraestructura que usted posee y administra a la infraestructura gestionado íntegramente por el proveedor de la nube (en este caso, Amazon). En una implementación del mundo real, la mayoría de las veces implementará su base de datos, infraestructura a máquinas virtuales antes que los contenedores Docker.

- 3 A diferencia de la implementación de su escritorio, desea que todo el tráfico del servidor pase su puerta de enlace API de Zuul . Vas a utilizar un grupo de seguridad de Amazon para solo Permitir que el puerto 5555 en el clúster ECS implementado sea accesible para todo el mundo.

- 4 Seguirá utilizando el servidor OAuth2 de Spring para proteger sus servicios. Antes de poder acceder a los servicios de organización y licencias, el usuario deberá autenticarse con sus servicios de autenticación (consulte el capítulo 7 para obtener más detalles sobre esto) y presentar un token OAuth2 válido en cada llamada de servicio.
- 5 Todos sus servidores, incluido su servidor Kafka, no serán accesibles públicamente para el mundo exterior a través de sus puertos Docker expuestos.

Algunos requisitos previos para trabajar.

Para configurar su infraestructura de Amazon, necesitará lo siguiente:

- 1 Su propia cuenta de Amazon Web Services (AWS). Debe tener un conocimiento básico de la consola de AWS y los conceptos detrás del trabajo en el entorno.
- 2 Un navegador web. Para la configuración manual, configurará todo, desde la consola.
- 3 El cliente de línea de comandos de Amazon ECS (<https://github.com/aws/amazon-ecs-cli>) para hacer un despliegue.

Si no tiene experiencia en el uso de los servicios web de Amazon, configuraría una cuenta de AWS e instalaría las herramientas de la lista. También dedicaría tiempo a familiarizarme con la plataforma.

Si es completamente nuevo en AWS, le recomiendo que obtenga una copia del libro de Michael y Andreas Wittig Amazon Web Services in Action (Manning, 2015). El primer capítulo del libro (<https://www.manning.com/books/amazon-web-services-in-action#downloads>) está disponible para descargar e incluye un tutorial bien escrito al final del capítulo sobre cómo registrarse y configurar su cuenta de AWS. Amazon Web Services in Action es un libro completo y bien escrito sobre AWS. Aunque llevo años trabajando con el entorno de AWS, sigo considerándolo un recurso útil.

Finalmente, en este capítulo he intentado en la medida de lo posible utilizar los servicios gratuitos que ofrece Amazon. El único lugar donde no pude hacer esto es al configurar el clúster ECS. Utilicé un servidor t2.large cuyo funcionamiento cuesta aproximadamente 0,10 centavos por hora. Asegúrese de cerrar sus servicios una vez que haya terminado si no desea incurrir en costos significativos.

**NOTA:** No hay garantía de que los recursos de Amazon (Postgres, Redis y ECS) que estoy usando en este capítulo estén disponibles si desea ejecutar este código usted mismo. Si va a ejecutar el código de este capítulo, debe configurar su propio repositorio de GitHub (para la configuración de su aplicación), su propia cuenta de Travis CI, Docker Hub (para sus imágenes de Docker) y su cuenta de Amazon, y luego modifique la configuración de su aplicación para que apunte a su cuenta y credenciales.

### 10.1.1 Creación de la base de datos PostgreSQL mediante Amazon RDS

Antes de comenzar esta sección, debe configurar y configurar su cuenta de Amazon AWS . Una vez hecho esto, su primera tarea es crear la base de datos PostgreSQL que utilizará para sus servicios EagleEye.

Para hacer esto, iniciará sesión en la consola de Amazon AWS (<https://aws.amazon.com/console/>) y haz lo siguiente:

- 1 Cuando inicie sesión por primera vez en la consola, se le presentará una lista de servicios web de Amazon. Localice el enlace llamado RDS. Haz clic en el enlace y esto te llevará al panel de RDS .
- 2 En el panel, encontrará un botón grande que dice "Iniciar una instancia de base de datos ". Haz click en eso.
- 3 Amazon RDS admite diferentes motores de bases de datos. Debería ver una lista de bases de datos. Seleccione PostgreSQL y haga clic en el botón "Seleccionar". Esto iniciará el asistente de creación de bases de datos.

Lo primero que le preguntará el asistente de creación de bases de datos de Amazon es si se trata de una base de datos de producción o de desarrollo/prueba. Vas a crear una base de datos de desarrollo/prueba utilizando el nivel gratuito. La Figura 10.2 muestra esta pantalla.

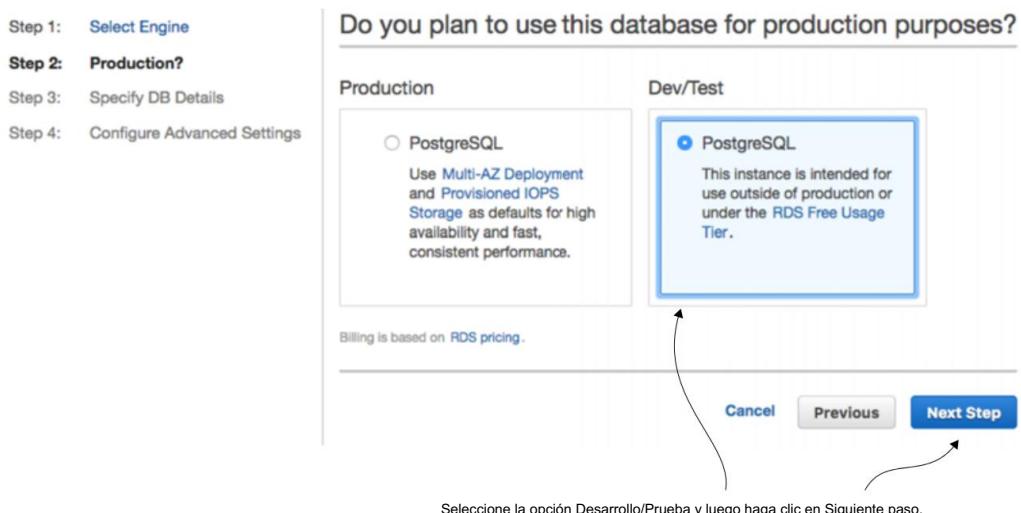


Figura 10.2 Seleccionar si la base de datos será una base de datos de producción o una base de datos de prueba

A continuación, configurará información básica sobre su base de datos PostgreSQL y También configure el ID de usuario maestro y la contraseña que utilizará para iniciar sesión en la base de datos. La Figura 10.3 muestra esta pantalla.

The Amazon RDS Free Tier provides a single db.t2.micro instance as well as up to 20 GB of storage, allowing new AWS customers to gain hands-on experience with Amazon RDS. Learn more about the RDS Free Tier and the instance restrictions [here](#).

Only show options that are eligible for RDS Free Tier

### Instance Specifications

DB Engine	postgres
License Model	postgresql-license
DB Engine Version	9.5.4
DB Instance Class	db.t2.micro — 1 vCPU, 1 GiB RAM
Multi-AZ Deployment	No
Storage Type	General Purpose (SSD)
Allocated Storage*	5 GB

**!** Provisioning less than 100 GB of General Purpose (SSD) storage for high throughput workloads could result in higher latencies upon exhaustion of the initial General Purpose (SSD) IO credit balance.  
[Click here](#) for more details.

### Settings

DB Instance Identifier*	eagle-eye-aws-dev
Master Username*	postgres_aws_dev
Master Password*	.....
Confirm Password*	.....

\* Required      Cancel      Previous      **Next Step**

Elija un db.t2.micro. Es la base de datos gratuita más pequeña y satisfará con creces sus necesidades. No necesitará una implementación multi-AZ.

Retype the value you specified for Master Password.

Tome nota de su contraseña. Para nuestros ejemplos, utilizará el master para iniciar sesión en la base de datos. En un sistema real, crearía una cuenta de usuario específica para la aplicación y nunca usaría directamente el ID de usuario/contraseña principal de la aplicación.

Figura 10.3 Configuración de la base de datos básica

El último y último paso del asistente es configurar los grupos de seguridad de la base de datos, la información del puerto y la información de respaldo de la base de datos. La Figura 10.4 muestra el contenido de esta pantalla.

### Configure Advanced Settings

**Network & Security**

VPC*	Default VPC (vpc-fa0dd89d)
Subnet Group	default
Publicly Accessible	Yes
Availability Zone	No Preference
VPC Security Group(s)	Create new Security Group default (VPC)

Por ahora, creará un nuevo grupo de seguridad y permitirá que la base de datos sea de acceso público.

Tenga en cuenta el nombre de la base de datos y el número de puerto. El número de puerto se utilizará como parte de la cadena de conexión de su servicio.

**Database Options**

Database Name	eagle_eye_aws_dev
Database Port	5432
DB Parameter Group	default.postgres9.5
Option Group	default:postgres-9-5
Copy Tags To Snapshots	<input type="checkbox"/>
Enable Encryption	No

Como este es un desarrollador base de datos, puede desactivar las copias de seguridad.

**Backup**

Backup Retention Period	0 days
-------------------------	--------

A backup retention period of zero days will disable automated backups for this DB Instance.

Backup Window

**Monitoring**

Enable Enhanced Monitoring	No
----------------------------	----

**Maintenance**

Auto Minor Version Upgrade	Yes
Maintenance Window	No Preference

Select the period in which you want pending modifications (such as changing the DB instance class) or patches applied to the DB instance by Amazon RDS. Any such maintenance should be started and completed within the selected period. If you do not select a period, Amazon RDS will assign a period randomly.  
[Learn More](#).

\* Required      Cancel      Previous      **Launch DB Instance**

Figura 10.4 Configuración del grupo de seguridad, el puerto y las opciones de respaldo para la base de datos RDS

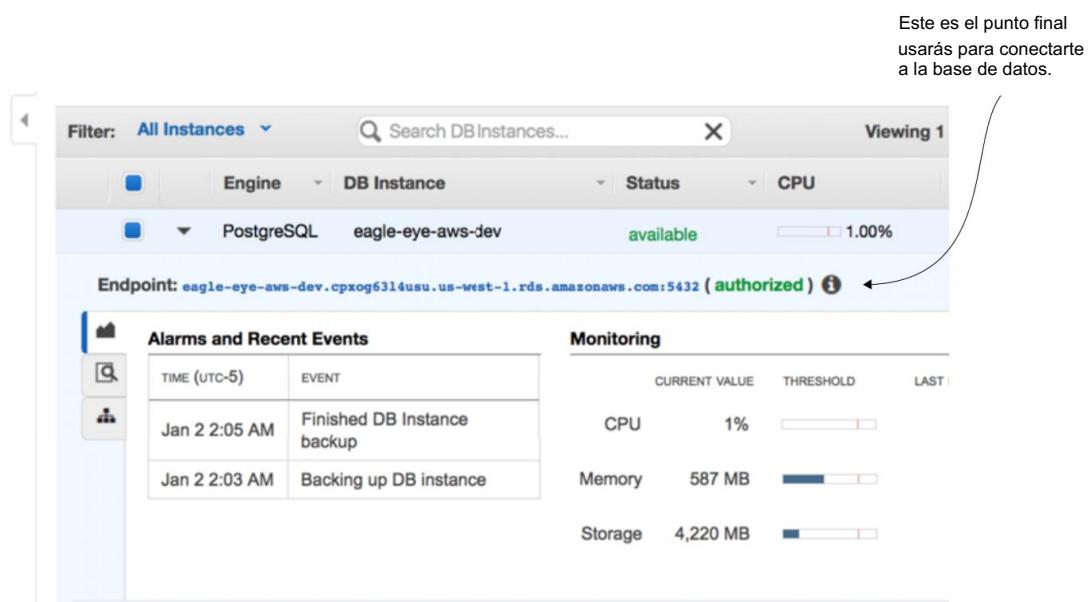


Figura 10.5 Su base de datos Amazon RDS/PostgreSQL creada

En este punto, comenzará el proceso de creación de su base de datos (puede tardar varios minutos). Una vez hecho esto, deberá configurar los servicios de Eagle Eye para usar la base de datos. Después de crear la base de datos (esto tomará varios minutos), regresará a el panel de RDS y vea su base de datos creada. La Figura 10.5 muestra esta pantalla.

Para este capítulo, creé un nuevo perfil de aplicación llamado aws-dev para cada microservicio que necesita acceder a la base de datos PostgreSQL basada en Amazon . Agregué un nuevo perfil de aplicación del servidor Spring Cloud Config en Spring Cloud Config Repositorio de GitHub (<https://github.com/carnelli/config-repo>) que contiene la información de conexión a la base de datos de Amazon. Los archivos de propiedades siguen la convención de nomenclatura (nombre-servicio)-aws-dev.yml en cada uno de los archivos de propiedades utilizando el nuevo base de datos (servicio de licencias, servicio de organización y servicio de autenticación).

En este punto, su base de datos está lista para funcionar (nada mal para configurarla en aproximadamente cinco clics). Pasemos a la siguiente pieza de infraestructura de aplicaciones y veamos cómo crear el clúster de Redis que utilizará su servicio de licencias EagleEye.

### 10.1.2 Creación del clúster de Redis en Amazon

Para configurar el clúster de Redis, utilizará el servicio Amazon ElastiCache. Amazon ElastiCache le permite crear cachés de datos en memoria utilizando Redis o Memcached (<https://memcached.org/>). Para los servicios EagleEye, usted se moverá el servidor Redis que estaba ejecutando en Docker a ElastiCache.

Para comenzar, regrese a la página principal de la consola de AWS (haga clic en el cubo naranja en la parte superior izquierda de la página) y haga clic en el enlace ElastiCache.

Desde la consola de ElastiCache, seleccione el enlace Redis (lado izquierdo de la pantalla), y luego presione el botón azul Crear en la parte superior de la pantalla. Esto hará aparecer el Asistente de creación de ElastiCache/Redis.

La Figura 10.6 muestra la pantalla de creación de Redis.

The screenshot shows the 'Create your Amazon ElastiCache cluster' wizard. In the 'Cluster engine' section, 'Redis' is selected. Below it, there's a description of Redis as an in-memory data structure store used as a database, cache, and message broker. A checkbox for 'Cluster Mode enabled (Scale Out)' is present but unchecked. The 'Memcached' option is also listed but not selected. In the 'Redis settings' section, the 'Name' field contains 'spmia-tmx-redis-dev'. To its right, a callout points to the text 'Este es el nombre de tu Servidor ElastiCache.' The 'Engine version compatibility' dropdown is set to '3.2.4'. The 'Port' is set to '6379'. The 'Parameter group' is 'default.redis3.2'. The 'Node type' is 'cache.t2.micro (0.5 GiB)', with a callout pointing to it stating 'Aquí se selecciona el tipo de instancia más pequeño.' The 'Number of replicas' dropdown is set to 'None'. At the bottom of the Redis settings section, a note says 'Como se trata de un servidor de desarrollo, no es necesario crear réplicas de los servidores Redis.' On the far right, there are 'Cancel' and 'Create' buttons.

Figura 10.6 Con unos pocos clics puede configurar un clúster de Redis cuya infraestructura es administrada por Amazon.

Continúe y presione el botón Crear una vez que haya completado todos sus datos. Amazon lo hará. Comience el proceso de creación del clúster de Redis (esto llevará varios minutos). Amazon lo hará. Cree un servidor Redis de un solo nodo que se ejecute en la instancia de servidor de Amazon más pequeña disponible. Una vez que presione el botón, verá que se crea su clúster de Redis. Una vez el. Después de crear el clúster, puede hacer clic en el nombre del clúster y lo llevará a una

Este es el punto final de Redis que utilizará en sus servicios.

Figura 10.7 El punto final de Redis es la pieza clave de información que sus servicios necesitan para conectarse a Redis.

Pantalla detallada que muestra el punto final utilizado en el clúster. La Figura 10.7 muestra los detalles del Redis agrupado después de su creación.

El servicio de licencias es el único de sus servicios que utiliza Redis, así que asegúrese de que si implementa los ejemplos de código de este capítulo en su propia instancia de Amazon, modifique los archivos Spring Cloud Config del servicio de licencias de forma adecuada.

### 10.1.3 Crear un clúster ECS

El último y último paso antes de implementar los servicios EagleEye es configurar un clúster de Amazon ECS . La configuración de un clúster de Amazon ECS aprovisiona las máquinas de Amazon que alojarán sus contenedores Docker. Para hacer esto, volverá a ir a la consola de Amazon AWS . Desde allí, hará clic en el enlace Amazon EC2 Container Service.

Esto lo llevará a la página principal del servicio EC2 Container, donde debería ver el botón "Comenzar".

Haga clic en el botón "Inicio". Esto lo llevará a "Seleccionar opciones para configurar". pantalla que se muestra en la figura 10.8.

Desmarque las dos casillas de verificación en la pantalla y haga clic en el botón cancelar. Ofertas ECS un asistente para configurar un contenedor ECS basado en un conjunto de plantillas predefinidas. No vas a utilizar este asistente. Una vez que cancele el asistente de configuración de ECS , debería ver la pestaña "Clústeres" en la página de inicio de ECS . La Figura 10.9 muestra esta pantalla.

Presione el botón "Crear clúster" para comenzar el proceso de creación de un clúster ECS .

Ahora verá una pantalla llamada "Crear clúster" que tiene tres secciones principales. La primera sección definirá la información básica del clúster. Aquí vas a entrar en

**1** Nombre de su clúster ECS .

**2** Tamaño de la máquina virtual Amazon EC2 en la que ejecutará el clúster

## Getting Started with Amazon EC2 Container Service (ECS)

### Select options to configure

Get started by running a sample app with EC2 Container Service (ECS), setting up a private image repository with EC2 Container Registry (ECR), or both.

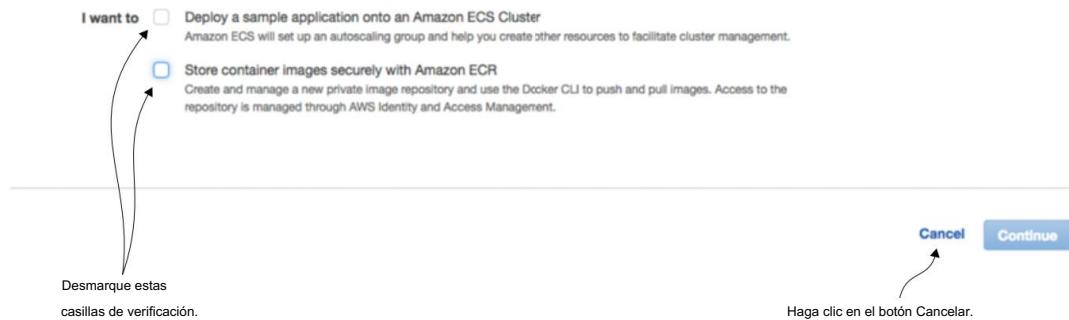


Figura 10.8 ECS ofrece un asistente para iniciar un nuevo contenedor de servicios. No lo vas a utilizar.



Figura 10.9 Inicio del proceso de creación de un clúster ECS

3 Número de instancias que ejecutará en su clúster.

4 Cantidad de espacio en disco de Elastic Block Storage (EBS) que asignará a cada nodo del clúster

## Create Cluster

When you run tasks using Amazon ECS, you place them on a cluster, which is a logical grouping of EC2 instances. This wizard will guide you through naming your cluster, and then configuring the container instances that your tasks can be placed on, the security group for your containers, and the port mappings for your container instances so that they can make calls to the AWS APIs on your behalf.

**Cluster name\*** spmia-tmx-dev

Create an empty cluster

**EC2 instance type\*** t2.large

**Number of instances\*** 1

**EC2 Ami Id\*** amzn-ami-2016.09.c-amazon-ecs-optimized [ami-1eda8d7e]

**EBS storage (GiB)\*** 22

**Key pair** spmia-tmx

You will not be able to SSH into your EC2 instances without a key pair.  
You can create a new key pair in the [EC2 console](#).

Puede elegir un servidor t2.large debido a su gran cantidad de memoria (8 GB) y bajo costo por hora (0,094 centavos por hora).

Como se trata de un entorno de desarrollo, lo ejecutará con una sola instancia.

Asegúrese de definir el par de claves SSH o no podrá utilizar SSH en la caja para diagnosticar problemas.

Figura 10.10 En el tamaño de pantalla "Crear clúster", las instancias EC2 utilizadas para alojar el clúster Docker.

La Figura 10.10 muestra la pantalla tal como la llené para los ejemplos de prueba de este libro.

**NOTA** Una de las primeras tareas que realiza cuando configura una cuenta de Amazon es definir un par de claves para SSH en cualquier servidor EC2 que inicie. No vamos a cubrir la configuración de un par de claves en este capítulo, pero si nunca lo has hecho antes te recomiendo que mires las indicaciones de Amazon al respecto (<http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-key-pairs.html>).

A continuación, configurará la red para el clúster ECS. La Figura 10.11 muestra la pantalla de red y los valores que estás configurando.

Lo primero que hay que tener en cuenta es seleccionar la nube privada virtual (VPC) de Amazon que se ejecutará el clúster ECS. De forma predeterminada, el asistente de configuración de ECS ofrecerá configurar una nueva VPC. Selecciona ejecutar el clúster ECS en mi VPC predeterminada. La VPC predeterminada alberga el servidor de base de datos y clúster Redis. En la nube de Amazon, un Redis gestionado por Amazon solo pueden acceder al servidor servidores que estén en la misma VPC que el servidor Redis.

A continuación, debe seleccionar las subredes en la VPC a las que desea dar acceso a la Clúster ECS. Como cada subred corresponde a una zona de disponibilidad de Amazon, normalmente seleccione todas las subredes en la VPC para que el clúster esté disponible.

## Networking

Configure the VPC for your container instances to use. A VPC is an isolated portion of the AWS cloud populated by AWS existing VPC, or create a new one with this wizard.



Figura 10.11 Una vez configurados los servidores, configure la red/grupos de seguridad de AWS utilizados para acceder a ellos.

Finalmente, debe seleccionar crear un nuevo grupo de seguridad o seleccionar un Amazon existente grupo de seguridad que ha creado para aplicar al nuevo clúster de ECS . Debido a que está ejecutando Zuul, desea que todo el tráfico fluya a través de un único puerto, el puerto 5555 . configurar el nuevo grupo de seguridad que está creando el asistente de ECS para permitir todas las entradas tráfico del mundo (0.0.0.0/0 es la máscara de red para todo Internet).

El último paso que se debe completar en el formulario es la creación de un Amazon IAM Rol del agente contenedor ECS que se ejecuta en el servidor. El agente ECS es responsable para comunicarse con Amazon sobre el estado de los contenedores que se ejecutan en el servidor. Permitirá que el asistente de ECS cree una función de IAM , denominada ecsIn-stanceRole, para usted. La Figura 10.12 muestra este paso de configuración.

### Container instance IAM role

The Amazon ECS container agent makes calls to the Amazon ECS API actions on your behalf, so container instances that run the agent require the service to know that the agent belongs to you. If you do not have the ecsInstanceRole already, we can create one for you.

The screenshot shows the 'Container instance IAM role' configuration screen. A dropdown menu is set to 'ecsInstanceRole'. At the bottom, there is a note about a required field, a 'Cancel' button, and a prominent blue 'Create' button.

Figura 10.12 Configuración de la función IAM del contenedor

En este punto, debería ver una pantalla que rastrea el estado de la creación del clúster. Una vez Una vez creado el clúster, debería ver un botón azul en la pantalla llamado "Ver clúster". Haga clic en el botón "Ver grupo". La Figura 10.13 muestra la pantalla que aparecerá después de presionar el botón "Ver grupo".

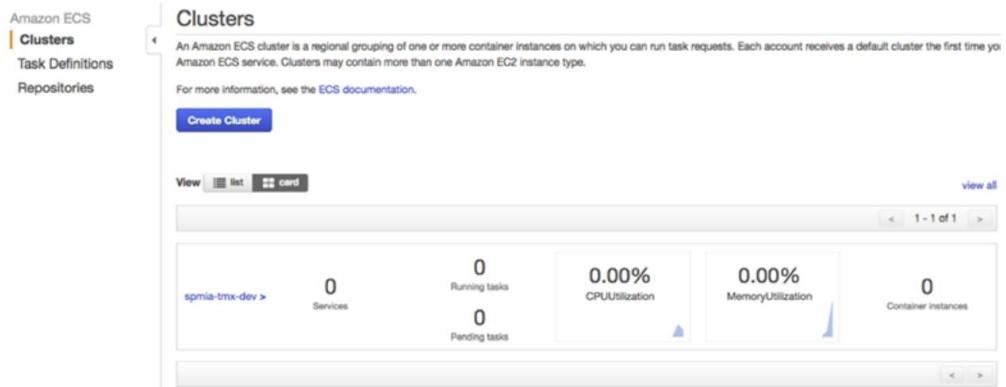


Figura 10.13 El clúster ECS en funcionamiento

En este punto, tiene toda la infraestructura que necesita para implementar con éxito los microservicios Eagle-Eye.

#### Sobre la configuración y automatización de la infraestructura

En este momento, está haciendo todo a través de la consola de Amazon AWS. En un entorno real, habría programado la creación de toda esta infraestructura utilizando DSL (lenguaje específico de dominio) de secuencias de comandos CloudFormation de Amazon o una secuencia de comandos de infraestructura en la nube. herramienta como Terraform de HashiCorp (<https://www.terraform.io/>). Sin embargo, eso es un tema entero en sí mismo y muy fuera del alcance de este libro. Si estás usando Amazon nube, probablemente ya esté familiarizado con CloudFormation. Si eres nuevo en la nube de Amazon, te recomiendo que te tomes el tiempo para aprenderla antes de llegar demasiado lejos. el camino de configuración de la infraestructura central a través de la consola de Amazon AWS.

Nuevamente, quiero remitir al lector a Amazon Web Services in Action (Manning, 2015) de Michael y Andreas Wittig. Recorren la mayor parte de la web de Amazon Servicios y demostrar cómo utilizar CloudFormation (con ejemplos) para automatizar la creación de su infraestructura.

## 10.2 Más allá de la infraestructura: implementación de EagleEye

En este punto, ya tienes la infraestructura configurada y ahora puedes pasar a la segunda mitad. del capítulo. En esta segunda parte, implementará los servicios de EagleEye en su contenedor de Amazon ECS . Vas a hacer esto en dos partes. La primera parte de tu

El trabajo es para personas con impaciencia terminal (como yo) y mostrará cómo implementar EagleEye manualmente a su instancia de Amazon. Esto le ayudará a comprender la mecánica de implementar el servicio y ver los servicios implementados ejecutándose en su contenedor. Mientras ensuciarse las manos e implementar manualmente sus servicios es divertido, no es sostenible ni recomendable.

Aquí es donde entra en juego la segunda parte de esta sección. Automatizará todo el proceso de construcción e implementación y eliminará al ser humano de la escena. Este es su estado final objetivo y realmente limita el trabajo que ha estado haciendo en el libro demostrando cómo diseñar, construir e implementar microservicios en la nube.

### 10.2.1 Implementación manual de los servicios EagleEye en ECS

Para implementar manualmente sus servicios EagleEye, cambiará de marcha y se moverá lejos de la consola de Amazon AWS . Para implementar los servicios EagleEye, deberá utilice el cliente de línea de comandos de Amazon ECS (<https://github.com/aws/amazon-ecs-cli>). Después de haber instalado el cliente de línea de comandos ECS , debe configurar ecs-cli entorno de ejecución para

- 1 Configure el cliente ECS con sus credenciales de Amazon
- 2 Seleccione la región en la que el cliente va a trabajar
- 3 Defina el clúster ECS predeterminado con el que trabajará el cliente ECS .
- 4 Este trabajo se realiza ejecutando el comando ecs-cli configure :

```
configuración ecs-cli --región us-west-1 \
    --clave de acceso $AWS_ACCESS_KEY \
    --clave-secreta $AWS_SECRET_KEY \
    --clúster spmia-tmx-dev
```

El comando ecs-cli configure establecerá la región donde se encuentra su clúster, su clave secreta y de acceso a Amazon, y el nombre del clúster (spmia-tmx-dev) te has desplegado. Si observa el comando anterior, estoy usando variables de entorno (\$AWS\_ACCESS\_KEY y \$AWS\_SECRET\_KEY) para mantener mi acceso a Amazon y llave secreta.

**NOTA** Seleccioné la región us-west-1 con fines puramente demostrativos.

Dependiendo del país en el que te encuentres, puedes elegir un Amazon región más específica de su parte del mundo.

A continuación, veamos cómo hacer una compilación. A diferencia de otros capítulos, debes configurar la compilación. nombre porque los scripts de Maven en este capítulo se usarán en el proceso de compilación e implementación que se configurará más adelante en el capítulo. Vas a configurar una variable de entorno llamada \$BUILD\_NAME. Se utiliza la variable de entorno \$BUILD\_NAME para etiquetar la imagen de Docker creada por el script de compilación. Cambie al directorio raíz del código del capítulo 10 que descargó de GitHub y emita los dos siguientes comandos:

```
exportar BUILD_NAME=TestManualBuild
ventana acoplable del paquete mvn clean: compilación
```

Esto ejecutará una compilación de Maven utilizando un POM principal ubicado en la raíz del directorio del proyecto. El pom.xml principal está configurado para crear todos los servicios que implementará en este capítulo. Una vez que el código Maven haya terminado de ejecutarse, puede implementar las imágenes de Docker en la instancia ECS que configuró anteriormente en la sección 10.1.3. Para realizar la implementación, emita el siguiente comando:

```
ecs-cli componer --file docker/common/docker-compose.yml arriba
```

El cliente de línea de comandos de ECS le permite implementar contenedores utilizando un archivo de composición Docker. Al permitirle reutilizar su archivo Docker-compose desde su entorno de desarrollo de escritorio, Amazon ha simplificado significativamente la implementación de sus servicios en Amazon ECS. Una vez que se haya ejecutado el cliente ECS , puede validar que los servicios se estén ejecutando y descubrir la dirección IP de los servidores emitiendo el siguiente comando:

```
ecs-cli ps
```

La Figura 10.14 muestra el resultado del comando ecs-cli ps .

Name	State	Ports	
bf5d7f7-515a-4ff5-b848-f3bb60bd9096/authenticationservice	RUNNING	54.153.112.116:8901->8901/tcp	
bf5d7f7-515a-4ff5-b848-f3bb60bd9096/organizationservice	RUNNING	54.153.112.116:8085->8085/tcp	
bf5d7f7-515a-4ff5-b848-f3bb60bd9096/kafkaserver	RUNNING	54.153.112.116:2181->2181/tcp, 54.153.112.116:9092->9092/tcp	
bf5d7f7-515a-4ff5-b848-f3bb60bd9096/licensingservice	RUNNING	54.153.112.116:8080->8080/tcp	
bf5d7f7-515a-4ff5-b848-f3bb60bd9096/zuulserver	RUNNING	54.153.112.116:5555->5555/tcp	
bf5d7f7-515a-4ff5-b848-f3bb60bd9096/eurekaserver	RUNNING	54.153.112.116:8761->8761/tcp	
bf5d7f7-515a-4ff5-b848-f3bb60bd9096/configserver	RUNNING	54.153.112.116:8888->8888/tcp	

Estos son puertos que están asignados en los contenedores Docker.  
Sin embargo, sólo el puerto 5555 está abierto al mundo exterior.

Figura 10.14 Comprobación del estado de los servicios implementados

Observe tres cosas del resultado de la figura 10.14:

- 1 Puede ver que se han implementado siete contenedores Docker, cada uno de los cuales Contenedor Docker ejecutando uno de sus servicios.
- 2 Puede ver la dirección IP del clúster ECS (54.153.122.116).
- 3 Parece que tiene otros puertos abiertos además del puerto 5555. Ese no es el caso. Los identificadores de puerto en la figura 10.14 son las asignaciones de puertos para el contenedor Docker. Sin embargo, el único puerto que está abierto al mundo exterior es el puerto 5555. Recuerde que cuando configuró su clúster ECS , el asistente de configuración de ECS creó un grupo de seguridad de Amazon que solo permitía el tráfico desde el puerto 5555.

En este punto, ha implementado con éxito su primer conjunto de servicios en Amazon ECS . cliente. Ahora, aprovechemos esto y veamos cómo diseñar una compilación y una implementación. canalización que puede automatizar el proceso de compilación, empaquetado e implementación de su servicios a Amazon.

#### Depurando por qué un contenedor ECS no se inicia o no permanece activo

ECS tiene herramientas limitadas para depurar por qué no se inicia un contenedor. si tienes problemas con un servicio implementado de ECS iniciando o permaneciendo activo, necesitará SSH en el ECS cluster para ver los registros de Docker. Para hacer esto necesitas agregar el puerto 22 a la seguridad grupo con el que se ejecuta el clúster ECS y luego SSH en la caja usando Amazon par de claves que definió en el momento en que se configuró el clúster (consulte la figura 10.9) como usuario ec2. Una vez que esté en el servidor, puede obtener una lista de todos los contenedores Docker que se ejecutan en el servidor ejecutando el comando docker ps. Una vez que haya localizado el contenedor imagen que desea depurar, puede ejecutar Docker Logs -f <<ID de contenedor>> comando para seguir los registros del contenedor Docker de destino.

Este es un mecanismo primitivo para depurar una aplicación, pero a veces solo necesita iniciar sesión en un servidor y ver la salida real de la consola para determinar qué está pasando.

### 10.3 La arquitectura de un proceso de construcción/implementación

El objetivo de este capítulo es proporcionarle las piezas de trabajo de un proceso de construcción/implementación para que pueda tomar estas piezas y adaptarlas a sus necesidades específicas. ambiente.

Comencemos nuestra discusión mirando la arquitectura general de su compilación. canal de implementación y varios de los patrones y temas generales que representa. Para que los ejemplos sigan fluyendo, he hecho algunas cosas que normalmente no haría. hacer en mi propio entorno y nombraré esas piezas en consecuencia.

Nuestra discusión sobre la implementación de microservicios comenzará con una imagen que vio Volviendo al capítulo 1. La figura 10.15 es un duplicado del diagrama que vimos en el capítulo 1. y muestra las piezas y los pasos involucrados en la construcción de un proceso de construcción e implementación de microservicios.

- La figura 10.15 debería resultarle familiar, porque se basa en la perspectiva general. Patrón de compilación-implementación utilizado para implementar la integración continua (CI):
- 1 Un desarrollador envía su código al repositorio de código fuente.
  - 2 Una herramienta de compilación monitorea el repositorio de control de fuente en busca de cambios e inicia una construir cuando se detecta un cambio.
  - 3 Durante la compilación, se ejecutan las pruebas unitarias y de integración de la aplicación y, si todo pasa, se crea un artefacto de software implementable (un JAR, WAR o EAR).
  - 4 Este JAR, WAR o EAR podría luego implementarse en un servidor de aplicaciones que ejecute en un servidor (normalmente un servidor de desarrollo).

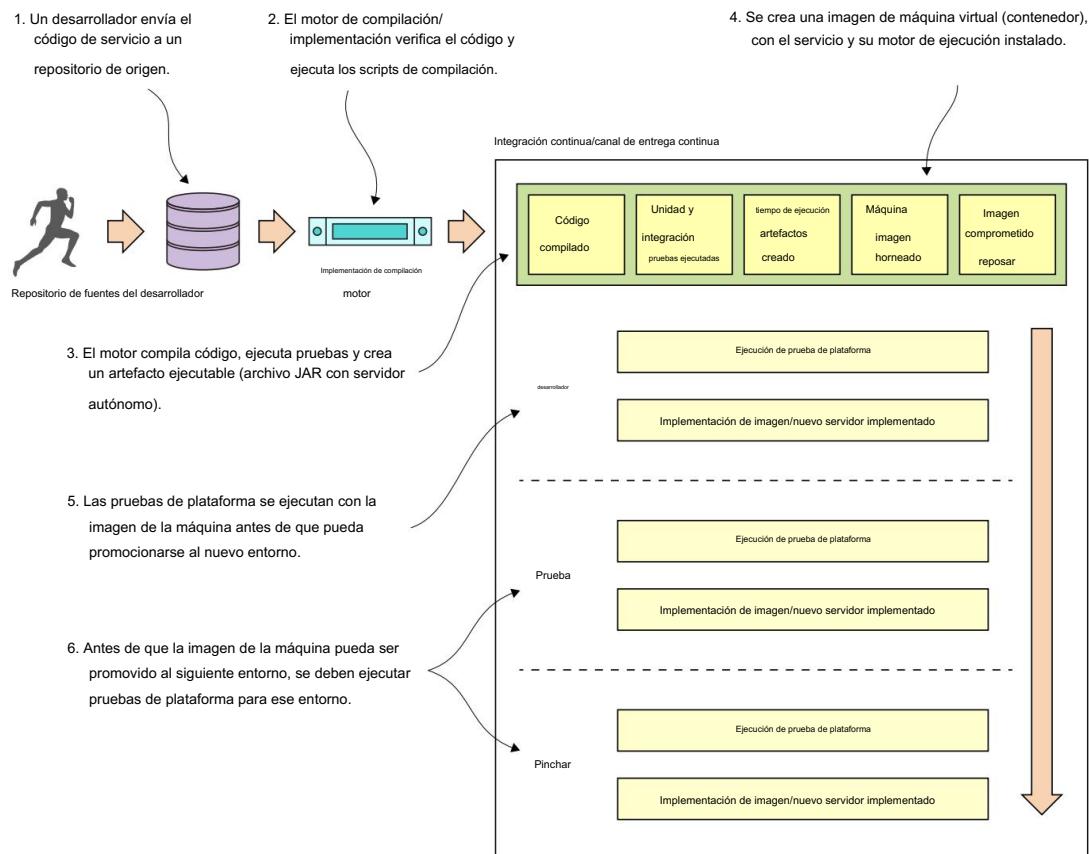


Figura 10.15 Cada componente en el proceso de construcción e implementación automatiza una tarea que se habría realizado manualmente.

Con el proceso de construcción e implementación (que se muestra en la figura 10.15), se sigue un proceso similar hasta que el código esté listo para ser implementado. En la construcción e implementación que se muestran en En la figura 10.15, agregará la entrega continua (CD) al proceso:

- 1 Un desarrollador envía su código de servicio a un repositorio fuente.
- 2 Un motor de compilación/implementación monitorea el repositorio de código fuente en busca de cambios. Si El código está confirmado, el motor de compilación/implementación verificará el código y lo ejecutará. los scripts de compilación del código.
- 3 El primer paso en el proceso de construcción/implementación es compilar el código, ejecutar su unidad y pruebas de integración, y luego compilar el servicio en un artefacto ejecutable. Debido a que sus microservicios se crean utilizando Spring Boot, su proceso de compilación cree un archivo JAR ejecutable que contenga tanto el código de servicio como el servidor Tomcat autónomo.

- 4 Aquí es donde su canal de compilación/implementación comienza a desviarse del proceso de compilación tradicional de Java CI . Después de crear su JAR ejecutable , "preparará" una imagen de máquina con su microservicio implementado en ella. Este proceso de horneado básicamente creará una imagen o contenedor de máquina virtual (Docker) e instalará su servicio en él. Cuando se inicie la imagen de la máquina virtual, su servicio se iniciará y estará listo para comenzar a aceptar solicitudes. A diferencia de un proceso de compilación de CI tradicional en el que podría (y quiero decir podría) implementar el JAR o WAR compilado en un servidor de aplicaciones que se administra de forma independiente (y a menudo con un equipo separado) desde la aplicación, con el proceso de CI/CD que está implementando. el microservicio, el motor de ejecución del servicio y la imagen de la máquina, todo como una unidad codependiente administrada por el equipo de desarrollo que escribió el software.
- 5 Antes de implementar oficialmente en un nuevo entorno, se inicia la imagen de la máquina y se ejecutan una serie de pruebas de plataforma con la imagen en ejecución para determinar si todo se está ejecutando correctamente. Si las pruebas de la plataforma pasan, la imagen de la máquina se promociona al nuevo entorno y está disponible para su uso.
- 6 Antes de promover un servicio al siguiente entorno, se deben ejecutar las pruebas de plataforma para el entorno. La promoción del servicio al nuevo entorno implica poner en marcha la imagen exacta de la máquina que se utilizó en el entorno inferior al entorno siguiente.

Esta es la salsa secreta de todo el proceso. Se implementa toda la imagen de la máquina. No se realizan cambios en ningún software instalado (incluido el sistema operativo) después de crear el servidor. Al promocionar y utilizar siempre la misma imagen de máquina, garantiza la inmutabilidad del servidor a medida que se promociona de un entorno a otro).

#### Pruebas unitarias versus pruebas de integración versus pruebas

de plataforma Verá en la figura 10.15 que realizo varios tipos de pruebas (unitarias, de integración y de plataforma) durante la construcción e implementación de un servicio. Tres tipos de pruebas son típicas en un proceso de construcción e implementación:

Pruebas unitarias: las pruebas unitarias se ejecutan inmediatamente antes de la compilación del código de servicio, pero antes de implementarlo en un entorno. Están diseñados para ejecutarse en completo aislamiento, siendo cada prueba unitaria pequeña y de enfoque limitado. Una prueba unitaria no debe depender de bases de datos de infraestructura de terceros, servicios, etc. Por lo general, el alcance de una prueba unitaria abarcará la prueba de un solo método o función.

Pruebas de integración: las pruebas de integración se ejecutan inmediatamente después de empaquetar el código de servicio. Estas pruebas están diseñadas para probar un flujo de trabajo completo y simular o simular servicios o componentes principales que deberían cancelarse de forma inmediata. Durante una prueba de integración, es posible que esté ejecutando una base de datos en memoria para almacenar datos, burlándose de llamadas de servicio de terceros, etc. Las pruebas de integración prueban un flujo de trabajo completo o una ruta de código. Para las pruebas de integración, las dependencias de terceros se burlan o eliminan para que cualquier

(continuado)

Las llamadas que invocarían un servicio remoto se burlan o bloquean para que las llamadas nunca abandone el servidor de compilación.

Pruebas de plataforma: las pruebas de plataforma se ejecutan justo antes de implementar un servicio en un entorno. Estas pruebas generalmente prueban un flujo de negocios completo y también llaman a todas las dependencias de terceros que normalmente se llamarían en un sistema de producción. Plataforma Las pruebas se ejecutan en vivo en un entorno particular y no involucran ningún simulacro. servicios. Se ejecutan pruebas de plataforma para determinar problemas de integración con terceros. servicios que normalmente no se detectarían cuando se bloquea un servicio de terceros durante una prueba de integración.

Este proceso de construcción/implementación se basa en cuatro patrones principales. Estos patrones no son creación mía, sino que surgieron de la experiencia colectiva de los equipos de desarrollo que crean microservicios y aplicaciones basadas en la nube. Estos patrones incluyen

Integración continua/entrega continua (CI/CD): con CI/CD, el código de su aplicación no solo se construye y prueba cuando se confirma; también se está implementando constantemente.

La implementación de su código debería ser algo como esto: si el código pasa las pruebas unitarias, de integración y de plataforma, debe ser inmediatamente promovido al siguiente entorno. El único punto de parada en la mayoría de las organizaciones es el impulso a la producción.

Infraestructura como código: el artefacto de software final que se impulsará hacia el desarrollo y más allá es una imagen de máquina. La imagen de la máquina y su microservicio

instalado en él se aprovisionará inmediatamente después de que su microservicio El código fuente se compila y prueba. El aprovisionamiento de la imagen de la máquina. ocurre a través de una serie de scripts que se ejecutan con cada compilación. Sin manos humanas alguna vez debería tocar el servidor después de haber sido construido. Los scripts de aprovisionamiento son mantenido bajo control de código fuente y administrado como cualquier otro fragmento de

código. Servidores inmutables: una vez creada una imagen de servidor, la configuración del servidor y el microservicio nunca se toca después del proceso de aprovisionamiento. Esto garantiza que su entorno no sufrirá una "desviación de configuración" en la que un

El desarrollador o administrador del sistema hizo "un pequeño cambio" que luego causó un apagón. Si es necesario realizar un cambio, los scripts de aprovisionamiento que aprovisionan Se cambia el servidor y se inicia una nueva compilación.

Sobre la inmutabilidad y el auge del servidor Phoenix Con el concepto de servidores inmutables, siempre debemos tener la garantía de que un La configuración del servidor coincide exactamente con la imagen de la máquina para el servidor. dice que sí. Un servidor debe tener la opción de cerrarse y reiniciarse desde el imagen de la máquina sin ningún cambio en el comportamiento del servicio o microservicios. Este Martin Fowler denominó la muerte y resurrección de un nuevo servidor como "Servidor Phoenix"

(<http://martinfowler.com/bliki/PhoenixServer.html>) porque cuando el servidor antiguo está asesinado, el nuevo servidor debería resurgir de las cenizas. El patrón del servidor Phoenix tiene dos beneficios clave.

En primer lugar, expone e impulsa la configuración fuera de su entorno. Si está constantemente derribando y configurando nuevos servidores, es más probable que exponga tempranamente cambios en la configuración. Esto es de gran ayuda para garantizar la coherencia. he gastado Demasiado tiempo y vida lejos de mi familia por llamadas de "situaciones críticas" debido a la desviación de la configuración.

En segundo lugar, el patrón del servidor Phoenix ayuda a mejorar la resiliencia al ayudar a encontrar situaciones en las que un servidor o servicio no se puede recuperar limpiamente después de haber sido eliminado y reiniciado. Recuerde, en una arquitectura de microservicio sus servicios no deben tener estado y la muerte de un servidor debe ser un problema menor. Matar y reiniciar aleatoriamente Los servidores exponen rápidamente situaciones en las que tiene estado en sus servicios o infraestructura. Es mejor encontrar estas situaciones y dependencias al principio de su proceso de implementación, en lugar de cuando está hablando por teléfono con una empresa enojada.

La organización donde trabajo utiliza Chaos Monkey de Netflix (<https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey>) para seleccionar y eliminar servidores aleatoriamente. Caos Monkey es una herramienta invaluable para probar la inmutabilidad y recuperabilidad de su entorno de microservicios. Chaos Monkey selecciona aleatoriamente instancias de servidor en tu ambiente y los mata. La idea al usar Chaos Monkey es que estés buscando para servicios que no pueden recuperarse de la pérdida de un servidor, y cuando un nuevo servidor es iniciado, se comportará de la misma manera que el servidor que fue cerrado.

## 10.4 Su canal de compilación e implementación en acción

De la arquitectura general expuesta en la sección 10.3, se puede ver que hay muchas piezas en movimiento detrás de un proceso de construcción/implementación. Porque el propósito de Este libro es para mostrarle cosas "en acción", vamos a repasar los detalles de implementar un proceso de construcción/implementación para los servicios EagleEye. Figura 10.16 describe las diferentes tecnologías que utilizará para implementar su canalización:

- 1 GitHub (<http://github.com>)—GitHub es nuestro repositorio de control de fuente. Todos El código de aplicación para este libro está en GitHub. Hay dos razones por las que GitHub fue elegido como repositorio de control de fuente. Primero, no quería administrar y mantener mi propio servidor de control de fuente Git. En segundo lugar, GitHub ofrece una amplia variedad de webhooks y sólidas API basadas en REST para integrar GitHub en su proceso de construcción.
- 2 Travis CI (<http://travis-ci.org>)—Travis CI es el motor de integración continua I utilizado para construir e implementar los microservicios y el aprovisionamiento de EagleEye la imagen de Docker que se implementará. Travis CI es un CI basado en archivos y en la nube motor que es fácil de configurar y tiene sólidas capacidades de integración con GitHub y Docker. Si bien Travis CI no tiene tantas funciones como un motor CI como Jenkins (<https://jenkins.io>), es más que adecuado para nuestros usos. lo describo usando GitHub y Travis CI en las secciones 10.5 y 10.6.

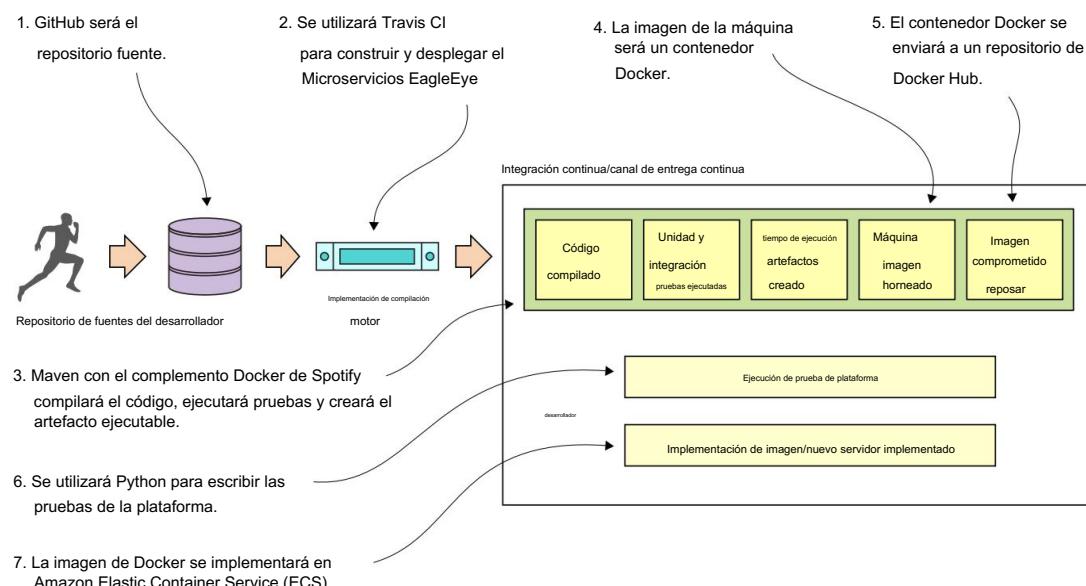


Figura 10.16 Tecnologías utilizadas en la construcción EagleEye

### 3 Complemento Docker de Maven/Spotify (<https://github.com/spotify/docker-maven-plugin>)

—Si bien utilizamos Vanilla Maven para compilar, probar y empaquetar código Java, una clave

El complemento Maven que utilizamos es el complemento Docker de Spotify. Este complemento nos permite comenzar la creación de una compilación de Docker directamente desde Maven.

### 4 Docker (<https://www.docker.com/>): yo Elegimos Docker como nuestra plataforma de contenedores.

por dos razones. Primero, Docker es portátil entre múltiples proveedores de nube. Puedo tomar el mismo contenedor Docker e implementarlo en AWS, Azure o Cloud Foundry

con una mínima cantidad de trabajo. En segundo lugar, Docker es liviano. Al final de

En este libro, ha creado e implementado aproximadamente 10 contenedores Docker.

(incluido un servidor de base de datos, una plataforma de mensajería y un motor de búsqueda).

Implementar la misma cantidad de máquinas virtuales en un escritorio local sería

difícil debido al gran tamaño y velocidad de cada imagen. La instalación y configuración de Docker, Maven y Spotify no se tratarán en este capítulo, pero sí.

en lugar de ello, se trata en el apéndice A.

### 5 Docker Hub (<https://hub.docker.com/>): después se ha construido un servicio y

Se ha creado la imagen de Docker, se etiqueta con un identificador único y se envía a un depósito central. Para el repositorio de imágenes de Docker, elegí usar Docker hub, el repositorio de imágenes públicas de Docker Corporation.

#### 6 Python (<https://python.org>): para escribir las pruebas de plataforma que se ejecutan

Antes de implementar una imagen de Docker, elegí Python como mi herramienta para escribir el pruebas de plataforma. Creo firmemente en el uso de las herramientas adecuadas para el trabajo y Francamente, creo que Python es un lenguaje de programación fantástico, especialmente para escribir casos de prueba basados en REST.

#### 7 EC2 Container Service (ECS) de Amazon : el destino final de nuestros microservicios

Se implementarán instancias de Docker en la plataforma Docker de Amazon. Elegí Amazon como mi plataforma en la nube porque es, con diferencia, el proveedor de nube más maduro y hace que sea trivial implementar servicios Docker.

### Espera... ¿dijiste Python?

Puede que le resulte un poco extraño que haya escrito las pruebas de la plataforma en Python en lugar de Java. Hice esto a propósito. Python (como Groovy) es un lenguaje de programación fantástico para escribir Casos de prueba basados en REST. Creo en utilizar la herramienta adecuada para el trabajo. Uno de los más grandes Los cambios de mentalidad que he visto en las organizaciones que adoptan microservicios es que la responsabilidad de elegir el lenguaje debe recaer en los equipos de desarrollo. En demasiadas organizaciones he visto una adopción dogmática de los estándares ("nuestro estándar empresarial es Java. . y todo el código debe estar escrito en Java"). Como resultado, he visto un desarrollo Los equipos pasan por obstáculos para escribir grandes cantidades de código Java cuando un Groovy de 10 líneas o el script Python haría el trabajo.

La segunda razón por la que elegí Python es que, a diferencia de las pruebas unitarias y de integración, la plataforma Las pruebas son verdaderamente pruebas de "caja negra" en las que usted actúa como un consumidor de API real ejecutándose en un entorno real. Las pruebas unitarias ejercitan el nivel más bajo de código y no deberían tener dependencias externas cuando se ejecutan. Las pruebas de integración suben de nivel y probar la API, pero las dependencias externas clave, como llamadas a otros servicios, bases de datos las llamadas, etc., se burlan o se eliminan. Las pruebas de plataforma deben ser pruebas verdaderamente independientes de la infraestructura subyacente.

## 10.5 Comenzando la implementación/canalización de su compilación: GitHub y Travis CI

Docenas de motores de control de código fuente y motores de implementación de compilación (tanto locales como basado en la nube) puede implementar su proceso de construcción e implementación. Para los ejemplos en este libro, elegí deliberadamente GitHub como repositorio de control de fuente y Travis CI como construir motor. El repositorio de control de código fuente de Git es un repositorio extremadamente popular y GitHub es uno de los repositorios de control de código fuente basados en la nube más grandes disponibles en la actualidad.

Travis CI es un motor de compilación que se integra estrechamente con GitHub (también es compatible con Sub-version y Mercurial). Es extremadamente fácil de usar y se acciona completamente con un solo archivo de configuración (.travis.yml) en el directorio raíz de su proyecto. Su simplicidad y su naturaleza de opinión hacen que sea fácil poner en marcha una tubería de construcción simple.

Hasta ahora, todos los ejemplos de código de este libro se pueden ejecutar únicamente desde su escritorio (con la excepción de la conectividad a GitHub). Para este capítulo, si

Si desea seguir completamente los ejemplos de código, deberá configurar sus propias cuentas de GitHub, Travis CI y Docker Hub. No vamos a explicar cómo configurar estas cuentas, pero la configuración de una cuenta personal de Travis CI y su cuenta de GitHub se puede realizar directamente desde la página web de Travis CI (<http://travis-ci.org>). .

#### Una nota rápida antes de comenzar

Para los propósitos de este libro (y mi cordura), configuré un repositorio de GitHub separado para cada capítulo del libro. Todo el código fuente del capítulo se puede crear e implementar como una sola unidad. Sin embargo, fuera de este libro, le recomiendo encarecidamente que configure cada microservicio en su entorno con su propio repositorio con sus propios procesos de compilación independientes. De esta manera, cada servicio se puede implementar de forma independiente uno del otro. Con el proceso de compilación, estoy implementando todos los servicios como una sola unidad solo porque quería llevar todo el entorno a la nube de Amazon con un único script de compilación y no administrar scripts de compilación para cada servicio individual.

#### 10.6 Cómo habilitar su servicio para construir en Travis CI En el centro

de cada servicio creado en este libro se encuentra un archivo Maven pom.xml que se utiliza para construir el servicio Spring Boot, empaquetarlo en un JAR ejecutable y luego construir una imagen de Docker que se puede utilizar para iniciar el servicio. Hasta este capítulo, la compilación y puesta en marcha de los servicios se realizaba mediante

- 1 Abrir una ventana de línea de comandos en su máquina local.
- 2 Ejecutando el script Maven para el capítulo. Esto crea todos los servicios para el capítulo y luego los empaqueta en una imagen de Docker que se enviará a un repositorio de Docker que se ejecuta localmente.
- 3 Iniciar las imágenes de Docker recién creadas desde su repositorio de Docker local, utilizando docker-compose y docker-machine para iniciar todos los servicios del capítulo.

La pregunta es, ¿cómo se repite este proceso en Travis CI? Todo comienza con un único archivo llamado .travis.yml. .travis.yml es un archivo basado en YAML que describe las acciones que desea realizar cuando Travis CI ejecute su compilación. Este archivo se almacena en el directorio raíz del repositorio GitHub de su microservicio. Para el capítulo 10, este archivo se puede encontrar en spmia-chapter10-code/.travis.yml.

Cuando se produce una confirmación en un repositorio de GitHub que Travis CI está monitoreando, buscará el archivo .travis.yml y luego iniciará el proceso de compilación. La Figura 10.17 muestra los pasos que seguirá su archivo .travis.yml cuando se realice una confirmación en el repositorio de GitHub utilizado para almacenar el código de este capítulo (<https://github.com/carnellj/spmia-chapter10>).

- 1 Un desarrollador realiza un cambio en uno de los microservicios en el repositorio de GitHub del capítulo 10.
- 2 GitHub notifica a Travis CI que se ha producido una confirmación. Esta configuración de notificación se produce sin problemas cuando se registra en Travis y proporciona su

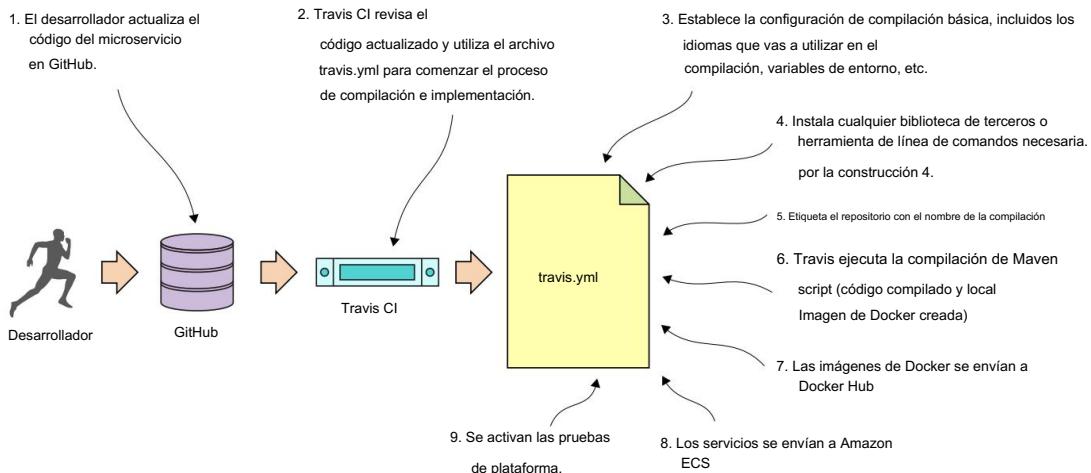


Figura 10.17 Los pasos concretos realizados por el archivo .travis.yml para construir e implementar su software

Notificación de cuenta de GitHub. Travis CI iniciará una máquina virtual que será utilizada para ejecutar la compilación. Travis CI luego verificará el código fuente de GitHub y luego use el archivo .travis.yml para comenzar la compilación e implementación general. proceso.

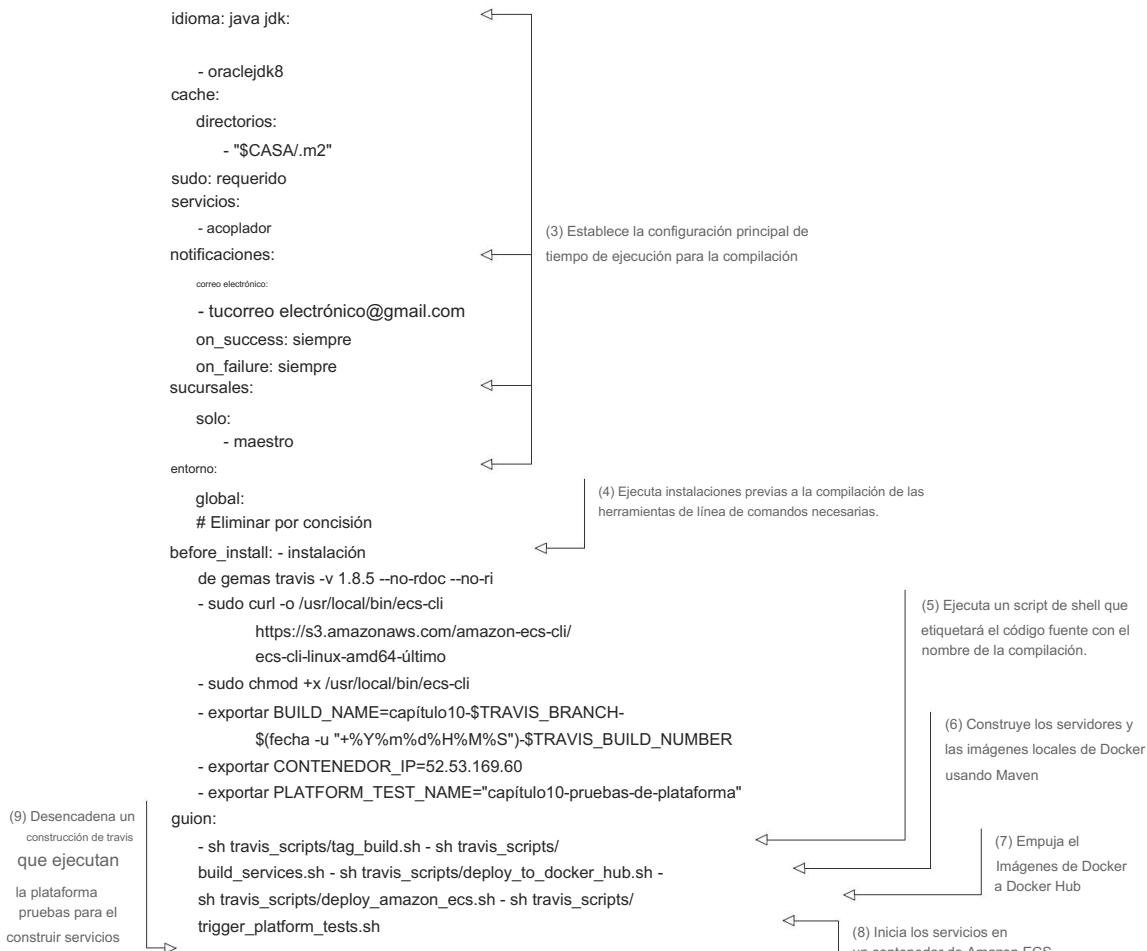
- 3 Travis CI establece la configuración básica en la compilación e instala las dependencias. La configuración básica incluye qué idioma vas a utilizar en el build (Java), si vas a necesitar Sudo para realizar instalaciones de software y acceso a Docker (para crear y etiquetar contenedores Docker), configurando cualquier variables de entorno seguras necesarias en la compilación y definir cómo debe ser notificado sobre el éxito o el fracaso de la construcción.
- 4 Antes de ejecutar la compilación real, se puede indicar a Travis CI que instale cualquier bibliotecas de terceros o herramientas de línea de comandos que podrían ser necesarias como parte del proceso de construcción. Se utilizan dos de estas herramientas, travis y Amazon ecs-cli (EC2 Cliente de Container Service) herramientas de línea de comandos.
- 5 Para su proceso de compilación, comience siempre etiquetando el código en el repositorio fuente para que en cualquier momento en el futuro pueda extraer la versión completa de el código fuente basado en la etiqueta de la compilación.
- 6 Su proceso de compilación luego ejecutará los scripts de Maven para los servicios. El Los scripts de Maven compilarán su microservicio Spring, ejecutarán las pruebas unitarias y de integración y luego crearán una imagen de Docker basada en la compilación.
- 7 Una vez que se complete la imagen de Docker para la compilación, el proceso de compilación impulsará la imagen al Docker Hub con el mismo nombre de etiqueta que usó para etiquetar su depósito de código fuente.

- 8 Luego, su proceso de compilación utilizará el archivo docker-compose del proyecto y ecs-cli de Amazon para implementar todos los servicios que ha creado en el servicio Docker de Amazon, Amazon ECS.
- 9 Una vez que se complete la implementación de los servicios, su proceso de compilación iniciará una Proyecto Travis CI completamente separado que ejecutará las pruebas de la plataforma contra el entorno de desarrollo.

Ahora que hemos recorrido los pasos generales involucrados en el archivo .travis.yml, veamos Mire los detalles de su archivo .travis.yml. El listado 10.1 muestra las diferentes piezas de el archivo .travis.yml.

**NOTA** Las anotaciones de código en el listado 10.1 están alineadas con los números en figura 10.17.

Listado 10.1 Anatomía de la compilación .travis.yml



Ahora vamos a repasar cada uno de los pasos involucrados en el proceso de construcción en mas detalle.

#### 10.6.1 Configuración del tiempo de ejecución de la compilación principal

La primera parte del archivo travis.yml trata sobre la configuración del tiempo de ejecución principal de su compilación de Travis. Normalmente, esta sección del archivo .travis.yml contendrá funciones específicas de Travis que harán cosas como

- 1 Dile a Travis en qué lenguaje de programación vas a trabajar
- 2 Defina si necesita acceso a Sudo para su proceso de compilación
- 3 Defina si desea utilizar Docker en su proceso de compilación
- 4 Declare las variables de entorno seguras que va a utilizar

La siguiente lista muestra esta sección específica del archivo de compilación.

Listado 10.2 Configurando el tiempo de ejecución central para su compilación

```
idioma: java jdk:  
  - oraclejdk8  
cache:  
  directorios:  
    - "$CASAS/.m2"  
sudo: servicios  
requeridos:  
  - acoplador  
notificaciones:  
  correo electrónico:  
    - tucorreo electrónico@gmail.com  
  on_success: siempre  
  on_failure: siempre  
sucursales:  
  solo:  
    - maestro  
entorno:  
  global:  
    -seguro: IAs5WrQIYjH0rpO6W37wbLAixjMB7kr7DBAeWhjeZFwOkUMJbfuHNC=z...  
    #d Quitar por razones de concisión
```

A: Le dice a Travis que use Java y JDK 8 para su entorno de ejecución principal

B: Le dice a Travis que almacene en caché y reutilice su Directorio Maven entre compilaciones

C: Permite que la compilación use el acceso Sudo en d la máquina virtual en la que se está ejecutando

D: Configura la dirección de correo electrónico utilizada para notificar el éxito o el fracaso de la compilación.

E: Le indica a Travis que solo debe basarse en un compromiso con la rama maestra.

F: Configura variables de entorno seguras para usar en sus scripts

Lo primero que hace el script de compilación de Travis es decirle a Travis qué idioma principal se utilizará para realizar la construcción. Al especificar el idioma como java y atributos jdk como java y oraclejdk8, B Travis se asegurará de que el JDK esté instalado y configurado para su proyecto.

La siguiente parte de su archivo .travis.yml, el atributo c de cache.directories , le indica Travis almacenará en caché los resultados de este directorio cuando se ejecute una compilación y los reutilizará en múltiples construcciones. Esto es extremadamente útil cuando se trata de administradores de paquetes como Maven, donde puede llevar una cantidad significativa de tiempo descargar copias nuevas de jar dependencias cada vez que se inicia una compilación. Sin los directorios cache.

conjunto de atributos, la compilación de este capítulo puede tardar hasta 10 minutos en descargar todo los frascos dependientes.

Los siguientes dos atributos en el listado 10.2 son el atributo sudo y el servicio. atributo. El atributo sudo se usa para decirle a Travis que su proceso de compilación necesitará usar sudo como parte de la compilación. El comando sudo de UNIX se utiliza para elevar temporalmente a un usuario a privilegios de root. Generalmente, usas sudo cuando necesitas instalar herramientas de terceros. Haz exactamente esto más adelante en la compilación cuando necesites instalar Amazon.

Herramientas ECS .

El atributo de servicios se utiliza para decirle a Travis si vas a utilizar ciertos servicios clave mientras ejecuta su compilación. Por ejemplo, si sus pruebas de integración necesitan una base de datos local disponible para su ejecución, Travis le permite iniciar MySQL o PostgreSQL base de datos directamente en su cuadro de compilación. En este caso, necesita que Docker se ejecute para construir su imágenes de Docker para cada uno de sus servicios EagleEye y envíe sus imágenes a Docker centro. Ha configurado el atributo de servicios para iniciar Docker cuando se inicia la compilación.

El siguiente atributo, notificaciones e define el canal de comunicación a Úselo cada vez que una compilación tenga éxito o falle. En este momento, siempre comunica la compilación. resultados configurando el canal de notificación para la compilación en correo electrónico. Travis te avisará por correo electrónico sobre el éxito y el fracaso de la construcción. Travis CI puede notificar a través de múltiples canales además del correo electrónico, incluidos Slack, IRC, HipChat o un enlace web personalizado.

El atributo Branches.only f le dice a Travis qué ramas debe construir Travis. contra. Para los ejemplos de este capítulo, solo realizará una compilación a partir del rama maestra de Git. Esto le impide iniciar una compilación cada vez que etiqueta un repositorio o comprometerse con una rama dentro de GitHub. Esto es importante porque GitHub hace un devolución de llamada a Travis cada vez que etiqueta un repositorio o crea una versión. La presencia del El atributo Branches.only establecido en master evita que Travis entre en una compilación interminable.

La última parte de la configuración de compilación es la configuración de variables de entorno sensibles g. En su proceso de construcción, puede comunicarse con proveedores externos como como Docker, GitHub y Amazon. A veces te comunica a través de sus herramientas de línea de comandos y otras veces utilizas las API. De todos modos, a menudo tienes que presentar credenciales confidenciales. Travis CI le brinda la posibilidad de agregar variables de entorno cifradas para proteger estas credenciales.

Para agregar una variable de entorno cifrada, debe cifrar el entorno variable usando la herramienta de línea de comando de travis en su escritorio en el directorio del proyecto donde tienes tu código fuente. Para instalar la herramienta de línea de comandos de Travis localmente, revisa la documentación de la herramienta en <https://github.com/travis-ci/travis.rb>. Para En el .travis.yml utilizado en este capítulo, creé y cifré las siguientes variables de entorno:

DOCKER\_USERNAME: nombre de usuario de Docker Hub.

DOCKER\_PASSWORD: contraseña de Docker Hub.

AWS\_ACCESS\_KEY: clave de acceso de AWS utilizada por el comando Amazon ecs-cli cliente de línea.

AWS\_SECRET\_KEY: clave secreta de AWS utilizada por el comando Amazon ecs-cli.  
cliente de línea.

GITHUB\_TOKEN: token generado por GitHub que se utiliza para indicar el acceso  
nivel que la aplicación que realiza la llamada puede realizar contra el servidor. Este  
El token debe generarse primero con la aplicación GitHub.

Una vez instalada la herramienta Travis , el siguiente comando agregará la variable de entorno cifrada  
DOCKER\_USERNAME a la sección env.global de su archivo .travis.yml:

```
travis cifrar DOCKER_USERNAME=algún nombre aleatorio --añadir env.global
```

Una vez que se ejecuta este comando, ahora debería ver en la sección env.global de su  
.travis.yml presenta una etiqueta de atributo segura seguida de una larga cadena de texto. Figura 10.18  
muestra cómo se ve una variable de entorno cifrada.

Las herramientas de cifrado de Travis no colocan el  
nombre de la variable de entorno cifrada en el archivo.

```
env:  
  global:  
    - secure: IAs5WrQIYjH0rp06W37wbLAixjMB7kr7DBAeWhjeZFw0k  
    - secure: HRSq780tWtfKKXZSq10ue/wV07TzIU+0mYPn1DctCnovs  
    - secure: m4IkvlGXq6LBzSEHJbabS/0cfCD1IRCmjfgp8BaN+wFY+
```

Cada variable de entorno cifrada tendrá  
una etiqueta de atributo seguro.

Figura 10.18 Las variables de entorno cifradas de Travis se colocan directamente en el archivo .travis.yml.

Desafortunadamente, Travis no etiqueta los nombres de las variables de entorno cifradas en su  
archivo .travis.yml.

**NOTA** Las variables cifradas solo son válidas para el único repositorio de GitHub.  
están encriptados y Travis está construyendo en contra. No puedes cortar y pegar un  
variable de entorno cifrada en varios archivos .travis.yml. Tus construcciones  
no se podrá ejecutar porque las variables de entorno cifradas no se descifrarán  
adecuadamente.

#### Independientemente de la herramienta de compilación, cifre siempre sus credenciales

Aunque todos nuestros ejemplos utilizan Travis CI como herramienta de compilación, todos los motores de compilación modernos  
le permite cifrar sus credenciales y tokens. Por favor, por favor, por favor asegúrese  
cifra sus credenciales. Las credenciales integradas en un repositorio de origen son una vulnerabilidad de  
seguridad común. No confie en la creencia de que su repositorio de control de código fuente  
es seguro y, por lo tanto, las credenciales que contiene son seguras.

### 10.6.2 Instalaciones de herramientas previas a la compilación

Vaya, la configuración previa a la compilación era enorme, pero la siguiente sección es pequeña. Construir Los motores son a menudo una fuente de una cantidad significativa de secuencias de comandos de "código adhesivo" para unir Reúne diferentes herramientas utilizadas en el proceso de construcción. Con tu script de Travis, necesitas Instale dos herramientas de línea de comandos:

Travis: esta herramienta de línea de comandos se utiliza para interactuar con la compilación de Travis. usted

Úselo más adelante en el capítulo para recuperar un token de GitHub y activar mediante programación otra compilación de Travis.

ecs-cli: esta es la herramienta de línea de comandos para interactuar con Amazon Elastic Servicio de contenedores.

Cada elemento enumerado en la sección before\_install del archivo .travis.yml es un comando UNIX que se ejecutará antes de que comience la compilación. El siguiente listado muestra la atributo before\_install junto con los comandos que deben ejecutarse.

#### Listado 10.3 Pasos de instalación previos a la compilación

```
Instala el
Amazonas
Cliente ECS
    ↗
antes_instalar:
    - instalación de gemas travis -v 1.8.5 --no-rdoc --no-ri - sudo curl -o /
      usr/local/bin/ecs-cli https://s3.amazonaws.com/
        amazon-ecs-cli / ecs-cli-linux-amd64-latest - sudo chmod +x /
          usr/local/bin/ecs-cli - exportar
            BUILD_NAME=chapter10-$TRAVIS_BRANCH-
                ↗
                ↗
                ↗
$fecha -u "%Y%m%d%H%M%S")-$TRAVIS_BUILD_NUMBER
- exportar CONTAINER_IP=52.53.169.60 - exportar
  PLATFORM_TEST_NAME="capítulo10-pruebas-de-plataforma"
    ↗
    ↗
    ↗
Instala la herramienta de
línea de comandos de Travis.
Cambia el permiso en el
El cliente de Amazon ECS será ejecutable
Establece las variables de
entorno utilizadas a través
de su proceso.
```

Lo primero que debe hacer en el proceso de compilación es instalar la herramienta de línea de comandos de travis en el servidor de compilación remoto:

instalación de gemas travis -v 1.8.5 --no-rdoc --no-ri

Más adelante en la compilación, iniciará otro trabajo de Travis a través de la API REST de Travis.

Necesita la herramienta de línea de comando de Travis para obtener un token para invocar esta llamada REST .

Después de haber instalado la herramienta travis , instalará Amazon ecs-cli

herramienta. Esta es una herramienta de línea de comandos que se utiliza para implementar, iniciar y detener Docker.

contenedores que se ejecutan dentro de Amazon. Para instalar ecs-cli , primero descargue el binario y luego cambiar el permiso en el binario descargado para que sea ejecutable:

```
- sudo curl -o /usr/local/bin/ecs-cli https://s3.amazonaws.com/amazon-ecs-
  cli/ecs-cli-linux-amd64-latest - sudo chmod
  +x /usr/local/bin/ecs-cli
```

Lo último que debe hacer en la sección before\_install de .travis.yml es configurar tres variables de entorno en su compilación. Estas tres variables de entorno ayudarán impulsa el comportamiento de tus compilaciones. Estas variables de entorno son

CONSTRUCCIÓN\_NOMBRE

CONTENEDOR\_IP

PLATAFORMA\_PRUEBA\_NOMBRE

Los valores reales establecidos en estas variables de entorno son

- exportar BUILD\_NAME=capítulo10-\$TRAVIS\_BRANCH-\$(fecha -u "+%Y%m%d%H%M%S")-\$TRAVIS\_BUILD\_NUMBER
- exportar CONTAINER\_IP=52.53.169.60 - exportar PLATFORM\_TEST\_NAME="capítulo10-pruebas-de-plataforma"

La primera variable de entorno, BUILD\_NAME, genera un nombre de compilación único que contiene el nombre de la compilación, seguido de la fecha y la hora (hasta los segundos campo) y luego el número de compilación en Travis. Este BUILD\_NAME se utilizará para etiquetar tu código fuente en GitHub y su imagen de Docker cuando se envía al centro de Docker repositorio.

La segunda variable de entorno, CONTAINER\_IP, contiene la dirección IP del Máquina virtual de Amazon ECS en la que se ejecutarán sus contenedores Docker. Este CONTAINER\_IP se pasará más tarde a otro trabajo de Travis CI que ejecutará su plataforma. pruebas de forma.

**NOTA** No estoy asignando una dirección IP estática al servidor de Amazon ECS que está hilado. Si destruyo el contenedor por completo, recibiré una nueva IP. en un verdadero entorno de producción, los servidores en su clúster ECS probablemente tendrán IP estáticas (que no cambian) asignadas a ellos, y el clúster tendrá un Amazon Enterprise Load Balancer (ELB) y un nombre DNS de Amazon Route 53 para que que la dirección IP real del servidor ECS sea transparente para los servicios. Sin embargo, establecer tanta infraestructura está fuera del alcance del ejemplo que estoy tratando de demostrar en este capítulo.

La tercera variable de entorno, PLATFORM\_TEST\_NAME, contiene el nombre del trabajo de construcción que se está ejecutando. Exploraremos su uso más adelante en el capítulo.

## Sobre auditoría y trazabilidad

Un requisito común en muchas empresas de servicios financieros y de atención médica es que Tienen que demostrar la trazabilidad del software implementado en producción, desde el principio hasta el final. a través de todos los entornos inferiores, de regreso al trabajo de compilación que creó el software, y luego, de regreso al momento en que el código se registró en el repositorio de código fuente. El patrón de servidor inmutable realmente destaca al ayudar a las organizaciones a cumplir con este requisito. Como Como vio en nuestro ejemplo de compilación, etiquetó el repositorio de control de fuente y la imagen del contenedor que se implementará con el mismo nombre de compilación. Ese nombre de construcción es único y está vinculado a un número de compilación de Travis. Porque solo promocionas el contenedor. imagen a través de cada entorno y cada imagen de contenedor está etiquetada con la compilación nombre, ha establecido la trazabilidad de esa imagen de contenedor hasta el origen código asociado a él. Porque los contenedores nunca se cambian una vez que están etiquetado, tiene una posición de auditoría sólida para demostrar que el código implementado coincide con el

(continuado)

repositorio de código fuente subyacente. Ahora, si quisieras ir más seguro, en el Una vez que etiquetó el código fuente del proyecto, también puede etiquetar la configuración de la aplicación que reside en el repositorio de Spring Cloud Config con la misma etiqueta generada. para la construcción.

### 10.6.3 Ejecutando la construcción

En este punto, toda la configuración previa a la compilación y la instalación de dependencias están completas. Para ejecutar su compilación, utilizará el atributo de secuencia de comandos de `Travis`. Como el atributo `before_install`, el atributo `script` toma una lista de comandos que serán ejecutado. Debido a que estos comandos son largos, elegí encapsular cada uno de los principales realice la compilación en su propio script de shell y haga que `Travis` ejecute el script de shell. El La siguiente lista muestra los principales pasos que se llevarán a cabo en la construcción.

#### Listado 10.4 Ejecutando la compilación

guion:

```
- sh travis_scripts/tag_build.sh - sh travis_scripts/
build_services.sh - sh travis_scripts/
deploy_to_docker_hub.sh - sh travis_scripts/deploy_amazon_ecs.sh
- sh travis_scripts/trigger_platform_tests.sh
```

Repasemos cada uno de los pasos principales que se ejecutan en el paso del guión.

### 10.6.4 Etiquetar el código de control fuente

El script `travis_scripts/tag_build.sh` se encarga de etiquetar el código en el repositorio con un nombre de compilación. Para el ejemplo aquí, estoy creando una versión de GitHub a través de GitHub. API DESCANSO. Una versión de GitHub no sólo etiquetará el repositorio de control de código fuente, sino que también le permite publicar cosas como notas de la versión en la página web de GitHub junto con si El código fuente es una versión preliminar del código.

Debido a que la API de lanzamiento de GitHub es una llamada basada en REST, usarás `curl` en tu shell script para realizar la invocación real. El siguiente listado muestra el código de la secuencia de comandos `travis_scripts/tag_build.sh`.

#### Listado 10.5 Etiquetado del repositorio de código del capítulo 10 con la API de lanzamiento de GitHub

```
echo "Etiquetar compilación con $BUILD_NAME"
exportar TARGET_URL="https://api.github.com/repos/
carnellj/spmia-chapter10/
lanzamientos?access_token=$GITHUB_TOKEN"

body="{ \"tag_name\": \"$BUILD_NAME\",
\"target_commitish\": \"maestro\",
\"nombre\": \"$BUILD_NAME\",
```

Punto final objetivo para el API de lanzamiento de GitHub

cuerpo de la llamada DESCANSO

```
\\"body\\": \"Lanzamiento de la versión $BUILD_NAME\", \\\"draft\\\":  
true, \\\"prerelease\\\":  
true }"
```

```
rizo -k -X POST \  
-H "Tipo de contenido: aplicación/json" \  
-d "$cuerpo" \  
$TARGET_URL
```

Utiliza curl para invocar el servicio  
utilizado para iniciar una compilación

Este guión es simple. Lo primero que debe hacer es crear la URL de destino para la API de lanzamiento de GitHub:

```
exportar TARGET_URL="https://api.github.com/repos/  
carnellijs/spmia-chapter10/  
lanzamientos?access_token=$GITHUB_TOKEN"
```

En TARGET\_URL estás pasando un parámetro de consulta HTTP llamado access\_token.

Este parámetro contiene un token de acceso personal de GitHub configurado para permitir específicamente que su secuencia de comandos actúe a través de la API REST. Su token de acceso personal de GitHub se almacena en una variable de entorno cifrada llamada GITHUB\_TOKEN. Para generar un token de acceso personal, inicie sesión en su cuenta de GitHub y navegue hasta <https://github.com/settings/tokens>. Cuando genere un token, asegúrese de cortarlo y pegarlo de inmediato. Cuando salgas de la pantalla de GitHub, desaparecerá y tendrás que regenerarla.

El segundo paso de tu script es configurar el cuerpo JSON para la llamada REST :

```
body="{ \"tag_name\": \"$BUILD_NAME\",  
\"target_commitish\": \"master\", \"name\": \"$BUILD_NAME\", \"body\": \"  
Lanzamiento de la versión $BUILD_NAME\", \"draft\": true,  
\"prerelease\": true }"
```

En el fragmento de código anterior, proporciona \$BUILD\_NAME para un valor de nombre de etiqueta y la configuración de notas de versión básicas utilizando el campo del cuerpo.

Una vez creado el cuerpo JSON de la llamada, ejecutar la llamada mediante el comando curl es trivial:

```
rizo -k -X POST \  
-H "Tipo de contenido: aplicación/json" -d "$cuerpo" \  
$TARGET_URL
```

## 10.6.5 Construyendo los microservicios y creando las imágenes de Docker

El siguiente paso en el atributo del script de Travis es crear los servicios individuales y luego crear imágenes de contenedor Docker para cada servicio. Esto se hace a través de un pequeño script llamado travis\_scripts/build\_services.sh. Este script ejecutará el siguiente comando:

ventana acopiable del paquete mvn clean: compilación

Este comando de Maven ejecuta el archivo principal Maven spmia-chapter10-code/pom.xml para todos los servicios en el repositorio de código del capítulo 10. El pom.xml principal ejecuta el pom.xml de Maven individual para cada servicio. Cada servicio individual crea el código fuente del servicio, ejecuta las pruebas unitarias y de integración y luego empaqueta el servicio en un archivo jar ejecutable.

Lo último que sucede en la compilación de Maven es la creación de una imagen de contenedor de Docker que se envía al repositorio local de Docker que se ejecuta en su máquina de compilación de Travis. La creación de la imagen de Docker se realiza mediante el complemento Spotify Docker (<https://github.com/spotify/docker-maven-plugin>). Si está interesado en cómo funciona el complemento Spotify Docker dentro del proceso de compilación, consulte el apéndice A, "Configuración de su entorno de escritorio". Allí se explican el proceso de compilación de Maven y la configuración de Docker.

#### 10.6.6 Enviar las imágenes a Docker Hub

En este punto de la compilación, los servicios se compilaron y empaquetaron y se creó una imagen de contenedor Docker en la máquina de compilación de Travis. Ahora vas a enviar la imagen del contenedor Docker a un repositorio central de Docker a través de tu script travis\_scripts/deploy\_to\_docker\_hub.sh. Un repositorio Docker es como un repositorio Maven para las imágenes Docker creadas. Las imágenes de Docker se pueden etiquetar y cargar en él, y otros proyectos pueden descargar y usar las imágenes.

Para este ejemplo de código, utilizará Docker Hub (<https://hub.docker.com/>). La siguiente lista muestra los comandos utilizados en el script travis\_scripts/deploy\_to\_docker\_hub.sh.

##### Listado 10.6 Enviar imágenes Docker creadas a Docker Hub

```
echo "Envío imágenes de la ventana acoplable del servicio a Docker Hub..."  
inicio de sesión de la ventana acoplable -u $DOCKER_USERNAME -p  
$DOCKER_PASSWORD docker push johncarnell/tmx-authentication-service:  
$BUILD_NAME docker push johncarnell/tmx-licensing-service:$BUILD_NAME  
docker push johncarnell/tmx-organization-service:$BUILD_NAME docker push  
johncarnell/tmx-confsrv:$BUILD_NAME docker push johncarnell/tmx-  
eurekasvr:$BUILD_NAME docker push johncarnell/tmx-zuulsvr:  
$BUILD_NAME
```

El flujo de este script de shell es sencillo. Lo primero que debe hacer es iniciar sesión en Docker Hub utilizando las herramientas de línea de comandos de Docker y las credenciales de usuario de la cuenta de Docker Hub a la que se enviarán las imágenes. Recuerde, sus credenciales para Docker Hub se almacenan como variables de entorno cifradas:

```
inicio de sesión en la ventana acoplable -u $DOCKER_USERNAME -p $DOCKER_PASSWORD
```

Una vez que el script haya iniciado sesión, el código impulsará el acceso de cada microservicio individual. Imagen de Docker que reside en el repositorio local de Docker que se ejecuta en el servidor de compilación de Travis, al repositorio de Docker Hub:

```
ventana acoplable push johncarnell/tmx-confsrv:$BUILD_NAME
```

En el comando anterior, le indica a la herramienta de línea de comandos de Docker que envíe al Docker Hub (que es el centro predeterminado que utilizan las herramientas de línea de comandos de Docker) al cuenta `johncarnell` . La imagen que se enviará será la imagen `tmx-confsrv` . con el nombre de etiqueta del valor de la variable de entorno `$BUILD_NAME` .

### 10.6.7 Inicio de los servicios en Amazon ECS

En este punto, todo el código se ha creado y etiquetado y se ha creado una imagen de Docker. Ahora está listo para implementar sus servicios en el contenedor de Amazon ECS que creado nuevamente en la sección 10.1.3. El trabajo para realizar esta implementación se encuentra en `travis_scripts/implementar_to_amazon_ecs.sh`. La siguiente lista muestra el código de este script.

#### Listado 10.7 Implementación de imágenes de Docker en EC2

```
echo "Lanzando $BUILD_NAME EN AMAZON ECS"
configuración ecs-cli --región us-west-1 \
    --clave de acceso $AWS_ACCESS_KEY
    --clave secreta $AWS_SECRET_KEY --
cluster spmia-tmx-dev
ecs-cli componer --file docker/common/docker-compose.yml arriba
rm -rf ~/.ecs
```

**NOTA** En la consola de Amazon, Amazon solo muestra el nombre del estado/ciudad/país en el que se encuentra la región y no el nombre real de la región (us-west-1, us-east-1, etc.). Por ejemplo, si buscara en la consola de Amazon y quisiera ver la región del norte de California, no habría ninguna indicación de que el nombre de la región sea us-west-1. Para obtener una lista de todas las regiones del Amazonas (y puntos finales para cada servicio), consulte <http://docs.aws.amazon.com/general/latest/gr/rande.html> .

Debido a que Travis inicia una nueva máquina virtual de compilación con cada compilación, es necesario para configurar el cliente `ecs-cli` de su entorno de compilación con su acceso a AWS y llave secreta. Una vez que esté completo, puede iniciar una implementación en su clúster ECS . usando el comando `ecs-cli compose` y un archivo `docker-compose.yml`. Su `docker-compose.yml` está parametrizado para usar el nombre de compilación (contenido en el entorno variable `$BUILD_NAME`).

### 10.6.8 Inicio de las pruebas de la plataforma

Le queda un último paso en el proceso de construcción: iniciar una prueba de plataforma. Después de cada implementación en un nuevo entorno, inicia una serie de pruebas de plataforma que verifican asegúrese de que todos sus servicios funcionen correctamente. El objetivo de las pruebas de la plataforma es para llamar a los microservicios en la compilación implementada y garantizar que los servicios funcionen correctamente.

He separado el trabajo de prueba de la plataforma de la compilación principal para poder invocarlo. independientemente de la construcción principal. Para hacer esto, utilizo la API REST de Travis CI para invocar mediante programación las pruebas de la plataforma. El script `travis_scripts/trigger_platform_tests.sh` Esto funciona. La siguiente lista muestra el código de este script.

## Listado 10.8 Inicio de las pruebas de la plataforma utilizando la API REST de Travis CI

```

echo "Comenzando pruebas de plataforma para la compilación $BUILD_NAME"
travis iniciar sesión --org --no-interactivo \
    --github-token $GITHUB_TOKEN export
RESULTS=`token travis --org` ←
exportar TARGET_URL="https://api.travis-ci.org/repo/
    carnellj%2F$PLATFORM_TEST_NAME/solicitudes"
echo "Iniciando el trabajo usando la URL de destino: $TARGET_URL"

cuerpo="{ \"solicitud\": {
    \"message\": \"Iniciando pruebas de plataforma para la compilación $BUILD_NAME\",
    \"cursors\": \"maestro\",
    \"config\": {
        \"entorno\": {
            \"global\": [\"$BUILD_NAME=$BUILD_NAME\",
                \"$CONTENEDOR_IP=$CONTENEDOR_IP\"] ←
        }
    }
}}"
```

Inicie sesión con Travis CI usando su Ficha de GitHub. Almacene el token devuelto en la variable RESULTADOS.

```
rizo -s -X POST \
    -H "Tipo de contenido: aplicación/json" \
    -H "Aceptar: aplicación/json" \
    -H "Travis-API-Versión: 3" \
    -H "Autorización: token $RESULTADOS" \
    -d "$cuerpo" \
    $TARGET_URL
```

Cree el cuerpo JSON para la llamada y pase dos valores al trabajo posterior.

Usando Curl para invocar la API REST de Travis CI

Lo primero que debe hacer en el listado 10.8 es usar la herramienta de línea de comandos de Travis CI para iniciar sesión en Travis CI y obtenga un token OAuth2 que puede usar para llamar a otras API REST de Travis. tu almacenas este token OAUTH2 en la variable de entorno \$RESULTS .

A continuación, crea el cuerpo JSON para la llamada a la API REST . Su trabajo posterior de Travis CI inicia una serie de scripts de Python que prueban su API. Este trabajo posterior espera dos variables de entorno que se van a establecer. En el cuerpo JSON que se está creando en el listado 10.8, estás pasando dos variables de entorno, \$BUILD\_NAME y \$CONTAINER\_IP, que serán pasado a su trabajo de prueba:

```

    \"entorno\": {
        \"global\": [\"$BUILD_NAME=$BUILD_NAME\",
            \"$CONTENEDOR_IP=$CONTENEDOR_IP\"] ←
    }
```

La última acción en su secuencia de comandos es invocar el trabajo de compilación de Travis CI que ejecuta las secuencias de comandos de prueba de su plataforma. Esto se hace usando el comando curl para llamar a Travis CI REST punto final para su trabajo de prueba:

```
rizo -s -X POST \
    -H "Tipo de contenido: aplicación/json" \
    -H "Aceptar: aplicación/json" \
    -H "Travis-API-Versión: 3" \
```

```
-H "Autorización: token $RESULTADOS" \ -d  
"$cuerpo" \  
$TARGET_URL
```

Los scripts de prueba de la plataforma se almacenan en un repositorio de GitHub separado llamado capítulo10-platform-tests (<https://github.com/carnelliJ/chapter10-platform-tests>). Este repositorio tiene tres scripts de Python que prueban el servidor Spring Cloud Config, el servidor Eureka y el servidor Zuul. Las pruebas de la plataforma del servidor Zuul también prueban los servicios de organización y licencias. Estas pruebas no son exhaustivas en el sentido de que analizan todos los aspectos de los servicios, pero sí analizan lo suficiente del servicio para garantizar su funcionamiento.

**NOTA** No vamos a repasar las pruebas de la plataforma. Las pruebas son sencillas y un recorrido por ellas no agregaría mucho valor a este capítulo.

#### 10.7 Pensamientos finales sobre el proceso de construcción/implementación

Al cerrar este capítulo (y el libro), espero que haya apreciado la cantidad de trabajo que implica construir un proceso de construcción/implementación. Un proceso de construcción e implementación que funcione bien es fundamental para la implementación de servicios. El éxito de su arquitectura de microservicio depende de algo más que el código involucrado en el servicio:

Comprenda que el código en este proceso de compilación/implementación está simplificado para los propósitos de este libro. Un buen proceso de construcción/implementación será mucho más generalizado. Será respaldado por el equipo de DevOps y se dividirá en una serie de pasos independientes (compilar > empaquetar > implementar > probar) que los equipos de desarrollo pueden usar para "conectar" sus scripts de compilación de microservicios.

El proceso de creación de imágenes de máquinas virtuales utilizado en este capítulo es simplista: cada microservicio se crea utilizando un archivo Docker para definir el software que se instalará en el contenedor Docker. Muchas tiendas utilizarán herramientas de aprovisionamiento como Ansible (<https://github.com/ansible/ansible>), Marioneta (<https://github.com/puppetlabs/puppet>), o Chef (<https://github.com/chef/chef>) para instalar y configurar los sistemas operativos en la máquina virtual o en las imágenes del contenedor que se están creando. La topología de implementación en la nube para su aplicación se ha consolidado en un único servidor. En el

proceso real de compilación/implementación, cada microservicio tendría sus propios scripts de compilación y se implementaría de forma independiente entre sí en un contenedor ECS de clúster .

## 10.8 Resumen

El proceso de construcción e implementación es una parte fundamental de la prestación de microservicios. Un proceso de compilación e implementación que funcione bien debería permitir la implementación de nuevas funciones y correcciones de errores en minutos.

El proceso de construcción e implementación debe automatizarse sin interacción humana directa para brindar un servicio. Cualquier parte manual del proceso representa una oportunidad de variabilidad y fracaso. La

automatización del proceso de construcción e implementación requiere una gran cantidad de secuencias de comandos y configuración para funcionar correctamente. No se debe subestimar la cantidad de trabajo necesario para construirlo.

La canalización de compilación e implementación debe entregar una máquina virtual o una imagen de contenedor inmutable. Una vez creada una imagen de servidor, nunca se debe modificar.

La configuración del servidor específica del entorno se debe pasar como parámetros en la hora en que se configura el servidor.

# Apéndice A

## Ejecutando una nube en su escritorio

---

### Este apéndice cubre

Listar el software necesario para ejecutar el código en este libro

Descargar el código fuente de GitHub para cada capítulo

Compilación y empaquetado del código fuente usando Maven

Creación y aprovisionamiento de las imágenes de Docker utilizadas en cada capítulo

Lanzar las imágenes de Docker compiladas por la compilación usando Docker Compose

Tenía dos objetivos al presentar los ejemplos de código de este libro y elegir las tecnologías de tiempo de ejecución necesarias para implementar el código. El primer objetivo era asegurarse de que los ejemplos de código fueran consumibles y fáciles de configurar. Recuerde, una aplicación de microservicios tiene múltiples partes móviles, y configurar estas partes para que se ejecuten limpiamente con un mínimo esfuerzo para el lector puede ser difícil si no hay algo de previsión.

El segundo objetivo era que cada capítulo fuera completamente independiente para que pudiera elegir cualquier capítulo del libro y tener disponible un entorno de ejecución completo que encapsule todos los servicios y software necesarios para ejecutar los ejemplos de código del capítulo sin dependencias de otros capítulos.

Con este fin, verá la siguiente tecnología y patrones utilizados en todo cada capítulo de este libro:

- 1 Todos los proyectos utilizan Apache Maven (<http://maven.apache.org>) como herramienta de construcción para los capítulos. Cada servicio se construye utilizando una estructura de proyecto Maven y cada estructura de servicio se establece de manera consistente capítulo a capítulo.
- 2 Todos los servicios desarrollados en el capítulo se compilan en Docker (<http://docker.io>) imagen del contenedor. Docker es un increíble motor de virtualización en tiempo de ejecución que ejecuta en Windows, OS X y Linux. Usando Docker, puedo construir un tiempo de ejecución completo entorno en el escritorio que incluye los servicios de la aplicación y todos la infraestructura necesaria para soportar los servicios. Además, Docker, a diferencia de más tecnologías de virtualización patentadas, es fácilmente portátil a través de múltiples nubes proveedores.

Estoy usando el complemento Docker Maven de Spotify (<https://github.com/spotify/complemento-maven-docker>) integrar la construcción del contenedor Docker con el Proceso de construcción de Maven.

- 3 Para iniciar los servicios después de que se hayan compilado en imágenes de Docker, uso Docker Redactar para iniciar los servicios como grupo. He evitado deliberadamente herramientas de orquestación Docker más sofisticadas como Kubernetes (<https://github.com/kubernetes/kubernetes>) o Mesos (<http://mesos.apache.org/>) para mantener el Ejemplos de capítulos sencillos y portátiles.

Todo el aprovisionamiento de las imágenes de Docker se realiza con scripts de shell simples.

## A.1 Software necesario

Para crear el software para todos los capítulos, necesitará tener el siguiente software instalado en su escritorio. Es importante tener en cuenta que estas son las versiones de software. Trabajé con para el libro. El software puede funcionar con otras versiones, pero esto es lo que Construí el código con:

- 1 Apache Maven (<http://apache.maven.org>)— Usé la versión 3.3.9 de Maven. I Eligió Maven porque, si bien otras herramientas de compilación como Gradle son extremadamente populares, Maven sigue siendo la herramienta de compilación predominante en uso en el ecosistema Java. Todo Los ejemplos de código de este libro se compilaron con la versión 1.8 de Java.
- 2 Docker (<http://docker.com>): yo construyó los ejemplos de código en este libro usando Ventana acoplable V1.12. Los ejemplos de código de este libro funcionarán con versiones anteriores de Docker, pero es posible que deba cambiar al formato de enlaces de composición de Docker versión 1 si desea utilizar este código con versiones anteriores de Docker.
- 3 Cliente Git (<http://git-scm.com>): todos El código fuente de este libro está almacenado en un Repositorio de GitHub. Para el libro, utilicé la versión 2.8.4 del cliente Git.

No voy a explicar cómo instalar cada uno de estos componentes. Cada una de las Los paquetes de software enumerados en la lista con viñetas tienen instrucciones de instalación simples y Debe poder instalarse con el mínimo esfuerzo. Docker tiene un cliente GUI para la instalación.

## A.2 Descarga de los proyectos desde GitHub

Todo el código fuente del libro está en mi repositorio de GitHub (<http://github.com/carnellj>). Cada capítulo del libro tiene su propio repositorio de código fuente. Aquí hay una lista de todos los repositorios de GitHub utilizados en el libro:

Capítulo 1 (Bienvenido a la nube, Spring): <http://github.com/carnellj/spmia-capítulo1>

Capítulo 2 (Introducción a los microservicios): <http://github.com/carnellj/spmia-Capítulo 2>

Capítulo 3 (Configuración de Spring Cloud): <http://github.com/carnellj/spmia-Capítulo 3> y <http://github.com/carnellj/config-repo>

Capítulo 4 (Spring Cloud/Eureka): <http://github.com/carnellj/spmia-Capítulo 4>

Capítulo 5 (Spring Cloud/Hystrix): <http://github.com/carnellj/spmia-Capítulo 5>

Capítulo 6 (Spring Cloud/Zuul): <http://github.com/carnellj/spmia-chapter6>

Capítulo 7 (Spring Cloud/Oauth2): <http://github.com/carnellj/spmia-Capítulo 7>

Capítulo 8 (Spring Cloud Stream): <http://github.com/carnellj/spmia-chapter8>

Capítulo 9 (Spring Cloud Sleuth): <http://github.com/carnellj/spmia-chapter9>

Capítulo 10 (Implementación): <http://github.com/carnellj/spmia-chapter10> y <http://github.com/carnellj/chapter-10-platform-tests>

Con GitHub, puede descargar los archivos como un archivo zip mediante la interfaz de usuario web. Cada GitHub El repositorio tendrá un botón de descarga. La Figura A.1 muestra dónde se realiza la descarga. El botón está en el repositorio de GitHub para el capítulo 1.

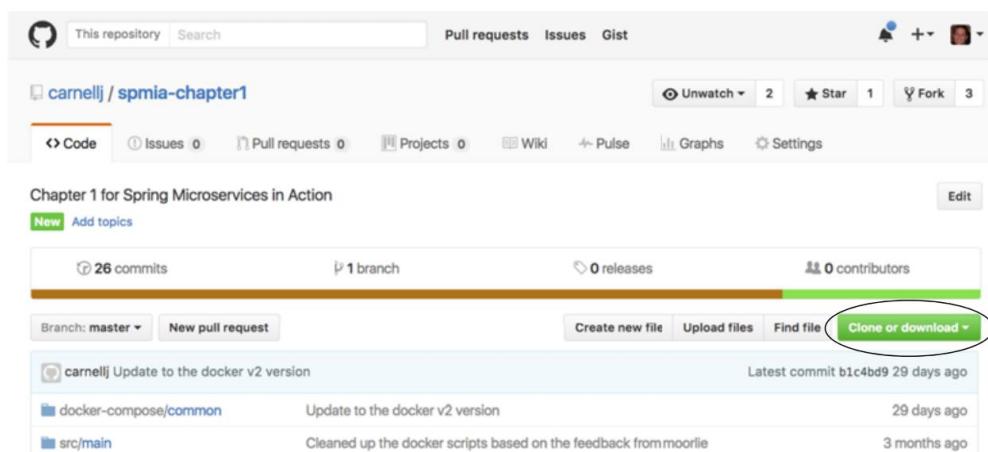


Figura A.1 La interfaz de usuario de GitHub le permite descargar un proyecto como un archivo zip.

Si es un usuario de línea de comandos, puede instalar el cliente git y clonar el proyecto.

Por ejemplo, si desea descargar el capítulo 1 de GitHub usando el cliente git , puede abrir una línea de comando y ejecutar el siguiente comando:

```
clon de git https://github.com/carnellj/spmia-chapter1.git
```

Esto descargará todos los archivos del proyecto del capítulo 1 en un directorio llamado spmia-chapter1 en el directorio desde donde ejecutó el comando git .

#### A.3 Anatomía de cada capítulo Cada capítulo del libro

tiene uno o más servicios asociados. Cada servicio en un capítulo tiene su propio directorio de proyectos. Por ejemplo, si observa el capítulo 6 (<http://github.com/carnellj/spmia-chapter6> ), Verás que hay siete servicios en él.

Estos servicios son

1 confsvr: servidor Spring Cloud Config 2 eurekasvr:

Spring Cloud/con Eureka 3 servicio de licencia: servicio

de licencia Eagle Eye 4 servicio de organización: servicio Eagle

Organization 5 orgservice-new: nueva versión de prueba del servicio

EagleEye 6 servicio de rutas especiales: A /B servicio de enrutamiento 7 zuulsvr

—servicio EagleEye Zuul

Cada directorio de servicios de un capítulo está estructurado como un proyecto de compilación basado en Maven.

Dentro de cada proyecto hay un directorio src/main con los siguientes subdirectorios:

1 java: este directorio contiene el código fuente de Java utilizado para crear el servicio. 2 docker: este directorio contiene dos archivos necesarios para crear una imagen de Docker para cada servicio. El primer archivo siempre se llamará Dockerfile y contiene las instrucciones paso a paso utilizadas por Docker para crear la imagen de Docker. El segundo archivo, run.sh, es un script Bash personalizado que se ejecuta dentro del contenedor Docker. Este script garantiza que el servicio no se inicie hasta que ciertas dependencias clave (la base de datos esté en funcionamiento) estén disponibles. 3 recursos: el directorio de recursos contiene todos los archivos

application.yml de los servicios. Si bien la configuración de la aplicación se almacena en Spring Cloud Config, todos los servicios tienen una configuración que se almacena localmente en application.yml. Además, el directorio de recursos contendrá un archivo esquema.sql que contiene todos los comandos SQL utilizados para crear las tablas y precargar datos para los servicios en la base de datos Post-gres.

#### A.4 Construcción y compilación de los proyectos

Debido a que todos los capítulos del libro siguen la misma estructura y utilizan Maven como herramienta de compilación, resulta extremadamente sencillo compilar el código fuente. Cada capítulo tiene en la raíz del directorio un pom.xml que actúa como pom principal para todos los subcapítulos. Si

desea compilar el código fuente y crear las imágenes de Docker para todos los proyectos dentro de un solo capítulo, debe ejecutar lo siguiente en la raíz del capítulo:

ventana acoplable del paquete mvn clean: compilación

Esto ejecutará el archivo Maven pom.xml en cada uno de los directorios de servicios. También lo hará Construya las imágenes de Docker localmente.

Si desea crear un único servicio dentro del capítulo, puede cambiar a ese directorio de servicios específico y ejecutar el comando mvn clean package docker:build .

## A.5 Construyendo la imagen de Docker

Durante el proceso de compilación, todos los servicios del libro se empaquetan como imágenes de Docker. Este proceso lo lleva a cabo el complemento Spotify Maven. Para ver un ejemplo de este complemento en acción, puede consultar el archivo pom.xml del servicio de licencias del capítulo 3 (capítulo 3/ servicio de licencias). El siguiente listado muestra el fragmento XML que configura este complemento en el archivo pom.xml de cada servicio.

Listado A.1 Complemento Spotify Docker Maven utilizado para crear Dockerimage

```
<complemento>
    <groupId>com.spotify</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <versión>0.4.10</versión>
    <configuración>
        <nombre de imagen>
            ${docker.image.name}:
            [ca]${docker.image.tag} <!-- Cada imagen de Docker creada tendrá una etiqueta asociada. El complemento de Spotify nombrará la imagen creada con lo que esté definido en la etiqueta ${docker.image.tag}.-->
        <imageName>
        <dockerDirectory>
            ${basedir}/target/dockerfile <!-- Todas las imágenes de Docker en este libro se crean utilizando un Dockerfile. Se utiliza un Dockerfile para brindar instrucciones paso a paso sobre cómo se debe aprovisionar la imagen de Docker.-->
        <recursos>
            <recurso>
                <rutadestino>/</rutadestino>
                <directorio>${project.build.directory}</directorio> <include>${project.build.finalName}.jar</include>
            </recurso>
        </recursos>
    </configuración>
</complemento>
```

← Cuando el complemento de Spotify está ejecutado, copiará el servicio jar ejecutable a la imagen de Docker.

El fragmento XML hace tres cosas:

- 1 Copia el jar ejecutable del servicio, junto con el contenido del archivo src/directorio principal/docker, hasta destino/docker.
- 2 Ejecuta el Dockerfile definido en el directorio target/docker. El archivo Docker es una lista de comandos que se ejecutan cada vez que se crea una nueva imagen de Docker para se presta ese servicio.

- 3** Envía la imagen de Docker al repositorio de imágenes de Docker local que está instalado. cuando instalas Docker.

La siguiente lista muestra el contenido del Dockerfile de su servicio de licencia.

Listado A.2 Dockerfile prepara la imagen de Docker

```
Esta es la imagen de Linux Docker que vas a ver  
Úsalo en su tiempo de ejecución de Docker. Esta instalación es  
optimizado para aplicaciones Java.  
  
DESDE openjdk:8-jdk-alpine RUN  
actualización de apk && actualización de apk && apk agrega netcat-openbsd RUN  
mkdir -p /usr/local/licensingservice  
AGREGAR licencia-servicio-0.0.1-SNAPSHOT.jar /usr/local/licensingservice/ AGREGAR run.sh run.sh  
  
EJECUTAR chmod +x ejecutar.sh  
CMD ./run.sh  
Agrega un shell BASH personalizado  
script que monitoreará  
dependencias del servicio y luego  
lanzar el servicio real.  
  
Instala nc (netcat), una utilidad  
que usarás para hacer ping dependiente  
servicios para ver si están activos.  
  
El comando AGREGAR de Docker  
copia el JAR ejecutable  
desde el sistema de archivos local a  
la imagen de Docker.
```

En el Dockerfile de este listado, estás aprovisionando tu instancia usando Alpine Linux (<https://alpinelinux.org/>). Alpine Linux es una pequeña distribución de Linux que A menudo se utiliza para crear imágenes de Docker. La imagen de Alpine Linux que estás usando ya tiene Java JDK instalado en él.

Cuando esté aprovisionando su imagen de Docker, instalará una utilidad de línea de comandos llamada nc. El comando nc se utiliza para hacer ping a un servidor y ver si un El puerto está en línea. Lo usará en su script de comando run.sh para asegurarse de que antes de iniciar su servicio, todos sus servicios dependientes (por ejemplo, la base de datos y el servicio Spring Cloud Config) se han iniciado. El comando nc hace esto mediante observando los puertos en los que escuchan los servicios dependientes. Esta instalación de nc se realiza mediante la actualización RUN apk && actualización apk && apk agrega netcat-openbsd, ejecutando los servicios utilizando Docker Compose.

A continuación, su Dockerfile creará un directorio para el ejecutable del servicio de licencias. jar y luego copie el archivo jar del sistema de archivos local a un directorio que se creó en la imagen de Docker. Todo esto se hace a través de ADD licensing-service-0.0.1-SNAPSHOT.jar /usr/local/licensingservice/.

El siguiente paso en el proceso de aprovisionamiento es instalar el script run.sh mediante ADD dominio. El script run.sh es un script personalizado que escribí y que inicia el servicio de destino cuando se inicia la imagen de Docker. Utiliza el comando nc para escuchar los puertos. de cualquier dependencia de servicio clave que el servicio de licencias necesite y luego bloquea hasta esas dependencias se inician.

La siguiente lista muestra cómo se utiliza run.sh para iniciar el servicio de licencias.

### Listado A.3 script run.sh utilizado para iniciar el servicio de licencias

```
#!/bin/sh

echo "***** Esperando que el servidor de configuración se inicie en el puerto $CONFIGSERVER_PORT"
echo "*****"
mientras ! `nc -z servidor de configuración $CONFIGSERVER_PORT`;
[ca]dormir 3; done echo
">>>>>>>>> El servidor de configuración se ha iniciado"

echo "***** Esperando que el servidor de base de datos se inicie en el puerto $DATABASESERVER_PORT"
echo "*****"
mientras ! `nc -z base de datos $DATABASESERVER_PORT`; duerme 3; hecho
echo ">>>>>>>>> El servidor de base de datos se ha iniciado"

echo "***** Iniciando el servidor de licencias con el servicio de configuración:
$CONFIGSERVER_URI";
echo "*****"

java -Dspring.cloud.config.uri=$CONFIGSERVER_URI \
-Dspring.profiles.active=$PROFILE \ -jar /usr/local/
licensingservice/licensing-service-0.0.1-SNAPSHOT.jar
```

Los scripts run.sh esperan a que se abra el puerto del servicio dependiente antes de continuar intentando iniciar el servicio.

Inicie el servicio de licencias utilizando Java para llamar al jar ejecutable el script Dockerfile instalado. \$<<VARIABLE\_NAME>> representa una variable de entorno que se pasa a la imagen de Docker.

Una vez que el comando run.sh se copia en la imagen de Docker del servicio de licencia, el comando CMD ./run.sh Docker se utiliza para indicarle a Docker que ejecute el script de inicio run.sh cuando se inicie la imagen real.

**NOTA:** Le estoy brindando una descripción general de alto nivel de cómo Docker aprovisiona una imagen. Si desea obtener más información sobre Docker en profundidad, le sugiero que consulte Docker in Action de Jeff Nickoloff (Manning, 2016) o Usando Docker de Adrian Mouat (O'Reilly, 2016). Ambos libros son excelentes recursos de Docker.

## A.6 Lanzar los servicios con Docker Compose

Una vez ejecutada la compilación de Maven, ahora puede iniciar todos los servicios del capítulo utilizando Docker Compose. Docker Compose se instala como parte del proceso de instalación de Docker. Es una herramienta de orquestación de servicios que le permite definir servicios como un grupo y luego lanzarlos juntos como una sola unidad. Docker Compose incluye capacidades para definir también variables de entorno con cada servicio.

Docker Compose utiliza un archivo YAML para definir los servicios que se lanzarán. Cada capítulo de este libro tiene un archivo llamado "<<chapter>>/docker/common/

`docker-compose.yml`". Este archivo contiene las definiciones de servicio utilizadas para iniciar los servicios del capítulo. Veamos el archivo `docker-compose.yml` utilizado en el capítulo 3.

La siguiente lista muestra el contenido de este archivo.

**Listado A.4 El archivo docker-compose.yml define los servicios que se lanzarán**

```
versión 2'
servidor de configuración:
  imagen: johncarnell/tmx-confsvr:chapter3
  puertos:
    - "8888:8888"
  ambiente:
    ENCRYPT_KEY: "IMSIMÉTRICO"
  base de datos:
    imagen: postgres
    puertos:
      - "5432:5432"
    ambiente:
      POSTGRES_USER: "postgres"
      POSTGRES_PASSWORD: "p0stgr@s"
      POSTGRES_DB: "eagle_eye_local"
  servicio de licencia:
    imagen: johncarnell/tmx-licensing-service:capítulo3
    puertos:
      - "8080:8080"
    ambiente:
      PERFIL: "predeterminado"
      CONFIGSERVER_URI: "http://configserver:8888"
      CONFIGSERVER_PORT: "8888"
      SERVIDOR_BASE DE DATOS: "5432"
      ENCRYPT_KEY: "IMSIMÉTRICO"
```

Docker Compose primero intentará encontrar la imagen de destino que se iniciará en el repositorio local de Docker. Si no puede Encuéntralo, comprobará la central. Centro Docker (<http://hub.docker.com>).

Esta entrada define los números de puerto. en el contenedor Docker iniciado que Estará expuesto al mundo exterior.

La etiqueta de entorno se utiliza para pasar variables de entorno a la imagen inicial de Docker. En este caso, la variable de entorno ENCRYPT\_KEY se establecerá en la imagen inicial de Docker.

Este es un ejemplo de cómo un servicio definido en una parte del Docker Se utiliza el archivo de redacción (configserver) como nombre DNS en otro servicio.

En `docker-compose.yml` del listado A.4, vemos que se definen tres servicios (servidor de configuración, base de datos y servicio de licencias). Cada servicio tiene una imagen de Docker definido con él usando la etiqueta de imagen . A medida que se inicia cada servicio, expondrá los puertos a través de la etiqueta del puerto y luego pasar las variables de entorno al contenedor Docker inicial a través de la etiqueta de entorno .

Continúe e inicie sus contenedores Docker ejecutando el siguiente comando desde la raíz del directorio del capítulo extraído de GitHub:

```
docker-compose -f docker/common/docker-compose.yml arriba
```

Cuando se emite este comando, `docker-compose` inicia todos los servicios definidos en el archivo `docker-compose.yml` . Cada servicio imprimirá su estándar en la consola. La figura A.2 muestra el resultado del archivo `docker-compose.yml` en el capítulo 3.

Los tres servicios escriben resultados en la consola.



```
configserver_1 | 2017-02-07 11:28:48.586 INFO 7 --- [           main] c.t.confsrv.ConfigServerApplication      : Sta
3 seconds (JVM running for 7.113)
licensingservice_1 | >>>>>>> Configuration Server has started
licensingservice_1 | ****
licensingservice_1 | Waiting for the database server to start on port 5432
licensingservice_1 | ****
licensingservice_1 | >>>>>>> Database Server has started
licensingservice_1 | ****
licensingservice_1 | Starting License Server with Configuration Service : http://configserver:8888
licensingservice_1 | ****
database_1 | LOG: incomplete startup packet
licensingservice_1 | 2017-02-07 11:28:52.715 INFO 17 --- [           main] s.c.a.AnnotationConfigApplicationContext : Re
.annotation.AnnotationConfigApplicationContext@6477463f: startup date [Tue Feb 07 11:28:52 GMT 2017]; root of context hier
licensingservice_1 | 2017-02-07 11:28:53.099 INFO 17 --- [           main] trationDelegate$BeanPostProcessorChecker : Be
utoConfiguration' of type [class org.springframework.cloud.autoconfigure.ConfigurationPropertiesRebinderAutoConfiguration$ not eligible for getting processed by all BeanPostProcessors (for example: not eligible for auto-proxying)
```

Figura A.2 Toda la salida de los contenedores Docker iniciados se escribe en la salida estándar.

**SUGERENCIA** Cada línea escrita en formato estándar por un servicio iniciado con Docker Compose tendrá el nombre del servicio impreso en formato estándar. Cuando inicia una orquestación de Docker Compose, encontrar errores que se imprimen puede resultar doloroso. Si desea ver el resultado de un servicio basado en Docker, inicie el comando docker-compose en modo independiente con la opción -d (docker-compose -f docker/common/docker-compose.yml up -d). Luego puede ver los registros específicos para ese contenedor emitiendo el comando docker-compose con la opción de registros (docker-compose -f docker/common/docker-compose.yml logs -f licensingservice).

Todos los contenedores Docker utilizados en este libro son efímeros: no conservarán su estado cuando se inicien y se detengan. Tenga esto en cuenta si comienza a jugar con el código y ve que sus datos desaparecen después de reiniciar sus contenedores. Si desea que su base de datos de Postgres sea persistente entre el inicio y la detención de los contenedores, le recomiendo que consulte las notas de Postgres Docker ([https://hub.docker.com/\\_/postgres/](https://hub.docker.com/_/postgres/) ).

## apéndice B

### Tipos de concesión de OAuth2

---

#### Este apéndice cubre

- Otorgamiento de contraseña de OAuth2
- OAuth2    Otorgamiento de credenciales de cliente de OAuth2
- de OAuth2    Otorgamiento de código de autorización de OAuth2
- de OAuth2    Otorgamiento de credenciales implícitas de OAuth2
- de OAuth2    Actualización de token de OAuth2

Al leer el capítulo 7, quizás estés pensando que OAuth2 no parece demasiado complicado. Después de todo, usted tiene un servicio de autenticación que verifica las credenciales de un usuario y le devuelve un token. El token, a su vez, puede presentarse cada vez que el usuario quiera llamar a un servicio protegido por el servidor OAuth2 .

Lamentablemente, el mundo real nunca es sencillo. Con la naturaleza interconectada de la web y las aplicaciones basadas en la nube, los usuarios esperan poder compartir de forma segura sus datos e integrar funcionalidades entre diferentes aplicaciones propiedad de diferentes servicios. Esto presenta un desafío único desde una perspectiva de seguridad porque desea integrar diferentes aplicaciones sin obligar a los usuarios a compartir sus credenciales con cada aplicación con la que desean integrarse.

Afortunadamente, OAuth2 es un marco de autorización flexible que proporciona múltiples Mecanismos para que las aplicaciones autentiquen y autoricen a los usuarios sin forzarlos. compartir credenciales. Desafortunadamente, también es una de las razones por las que OAuth2 es considerado complicado. Estos mecanismos de autenticación se denominan autenticación. subsidios. OAuth2 tiene cuatro formas de concesiones de autenticación que las aplicaciones cliente pueden usar para autenticar usuarios, recibir un token de acceso y luego validarlos. Estos

las subvenciones son

- Contraseña
- Credencial de cliente
- Código de autorización
- Implícito

En las siguientes secciones, analizo las actividades que tienen lugar durante la ejecución de cada uno de estos flujos de subvención de OAuth2 . También hablo sobre cuándo usar un tipo de subvención. Sobre otra.

## B.1 Concesión de contraseñas

Una concesión de contraseña OAuth2 es probablemente el tipo de concesión más sencillo de entender. Este tipo de concesión se utiliza cuando tanto la aplicación como los servicios explícitamente confiar unos en otros. Por ejemplo, la aplicación web EagleEye y la aplicación web EagleEye Los servicios (la licencia y la organización) son propiedad de ThoughtMechanix, por lo que existe una relación de confianza natural entre ellos.

**NOTA** Para ser explícito, cuando me refiero a una "relación de confianza natural" me refiero que la aplicación y los servicios son propiedad exclusiva de la misma organización. Se gestionan bajo las mismas políticas y procedimientos.

Cuando existe una relación de confianza natural, hay poca preocupación por exponer una Token de acceso OAuth2 a la aplicación que llama. Por ejemplo, la aplicación web EagleEye puede utilizar la concesión de contraseña OAuth2 para capturar las credenciales del usuario y autenticarse directamente contra el servicio EagleEye OAuth2 . La Figura B.1 muestra la concesión de contraseña en acción entre EagleEye y los servicios descendentes.

En la figura B.1 se están llevando a cabo las siguientes acciones:

- 1 Antes de que la aplicación EagleEye pueda utilizar un recurso protegido, es necesario identificado de forma única dentro del servicio OAuth2 . Normalmente, el propietario de la aplicación se registra en el servicio de aplicación OAuth2 y proporciona un nombre para su aplicación. El servicio OAuth2 luego proporciona una clave secreta. para registrar la aplicación.

El nombre de la aplicación y la clave secreta proporcionada por OAuth2.

El servicio identifica de forma única la aplicación que intenta acceder a cualquier área protegida. recursos.

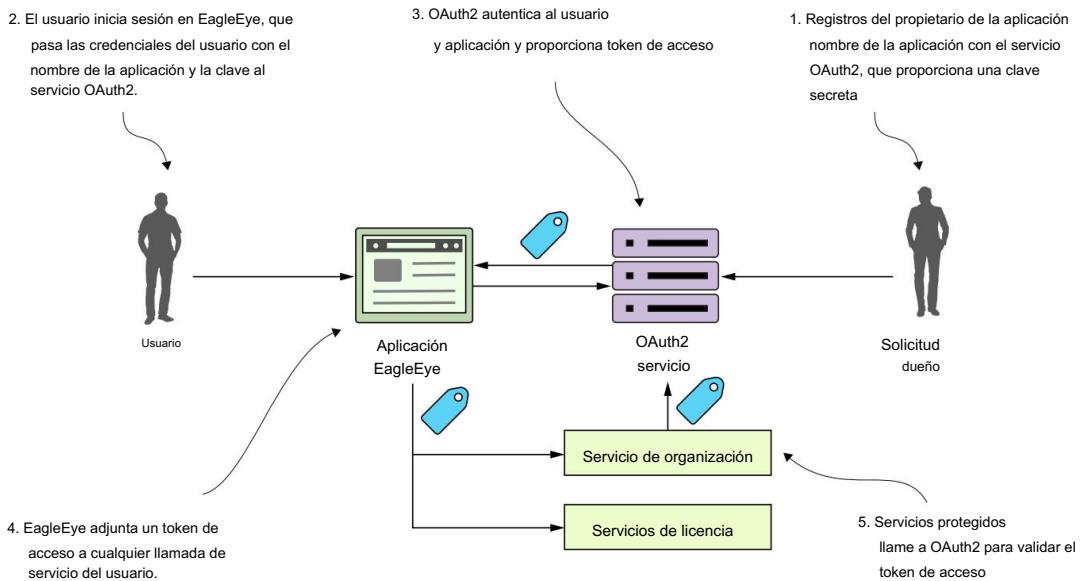


Figura B.1 El servicio OAuth2 determina si el usuario que accede al servicio es un usuario autenticado.

- 2 El usuario inicia sesión en EagleEye y proporciona sus credenciales de inicio de sesión a la aplicación Eagle-Eye. EagleEye pasa las credenciales del usuario, junto con el nombre de la aplicación/clave secreta de la aplicación, directamente al servicio EagleEye OAuth2 .
- 3 El servicio EagleEye OAuth2 autentica la aplicación y el usuario y luego proporciona un token de acceso OAuth2 al usuario.
- 4 Cada vez que la aplicación EagleEye llama a un servicio en nombre del usuario, pasa el token de acceso proporcionado por el servidor OAuth2 .
- 5 Cuando se llama a un servicio protegido (en este caso, el servicio de organización y licencia), el servicio vuelve a llamar al servicio EagleEye OAuth2 para validar el token. Si el token es bueno, el servicio que se invoca le permite al usuario para proceder. Si el token no es válido, el servicio OAuth2 devuelve un HTTP código de estado de 403, que indica que el token no es válido.

## B.2 Concesión de credenciales de cliente

La concesión de credenciales de cliente se utiliza normalmente cuando una aplicación necesita acceder a un Recurso protegido por OAuth2 , pero ningún ser humano participa en la transacción. Con Según el tipo de concesión de credenciales del cliente, el servidor OAuth2 solo se autentica según el nombre de la aplicación y la clave secreta proporcionada por el propietario del recurso. Nuevamente, la tarea de credenciales de cliente generalmente se usa cuando ambas aplicaciones pertenecen al mismo compañía. La diferencia entre la concesión de contraseña y la concesión de credenciales de cliente es que la concesión de una credencial de cliente se autentica utilizando únicamente la aplicación registrada nombre y la clave secreta.

Por ejemplo, digamos que una vez cada hora la aplicación EagleEye tiene un trabajo de análisis de datos que se ejecuta. Como parte de su trabajo, realiza llamadas a los servicios de EagleEye. Sin embargo, Los desarrolladores de EagleEye todavía quieren que esa aplicación se autentique y autorice. antes de que pueda acceder a los datos en esos servicios. Aquí es donde se otorga la credencial de cliente. puede ser usado. La figura B.2 muestra este flujo.

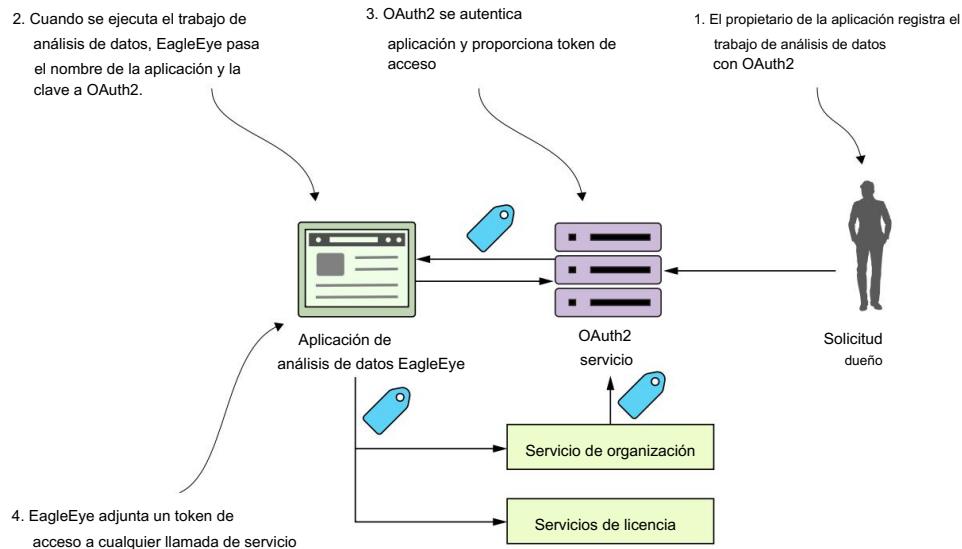


Figura B.2 La concesión de credenciales de cliente es para autenticación y autorización de aplicaciones "sin participación del usuario".

- 1 El propietario del recurso registra la aplicación de análisis de datos EagleEye con el Servicio OAuth2. El propietario del recurso proporcionará el nombre de la aplicación y recibir de vuelta una clave secreta.
- 2 Cuando se ejecute el trabajo de análisis de datos de EagleEye, presentará el nombre de su aplicación y clave secreta proporcionada por el propietario del recurso.
- 3 El servicio EagleEye OAuth2 autenticará la aplicación utilizando el nombre de la aplicación y la clave secreta proporcionada y luego devolverá un OAuth2. ficha de acceso.
- 4 Cada vez que la aplicación llame a uno de los servicios EagleEye, presentará el Token de acceso OAuth2 que recibió con la llamada de servicio.

### B.3 Concesión de códigos de autorización

La concesión del código de autorización es, con diferencia, la más complicada de las concesiones de OAuth2 . pero también es el flujo más común utilizado porque permite diferentes aplicaciones de diferentes proveedores para compartir datos y servicios sin tener que exponer la información de un usuario

credenciales en múltiples aplicaciones. También impone una capa adicional de verificación mediante no permitir que una aplicación que llama obtenga inmediatamente un token de acceso OAuth2 , sino más bien un código de autorización de “pre-acceso”.

La forma más sencilla de entender la concesión de autorización es mediante un ejemplo. vamos Supongamos que tiene un usuario de EagleEye que también utiliza Salesforce.com. El cliente de EagleEye El departamento de TI ha creado una aplicación Salesforce que necesita datos de un servicio EagleEye (el servicio de la organización). Repasemos la figura B.3 y veamos cómo funciona el flujo de concesión del código de autorización para permitir que Salesforce acceda a los datos desde EagleEye. servicio de organización, sin que el cliente de EagleEye tenga que exponer su Credenciales de EagleEye para Salesforce.

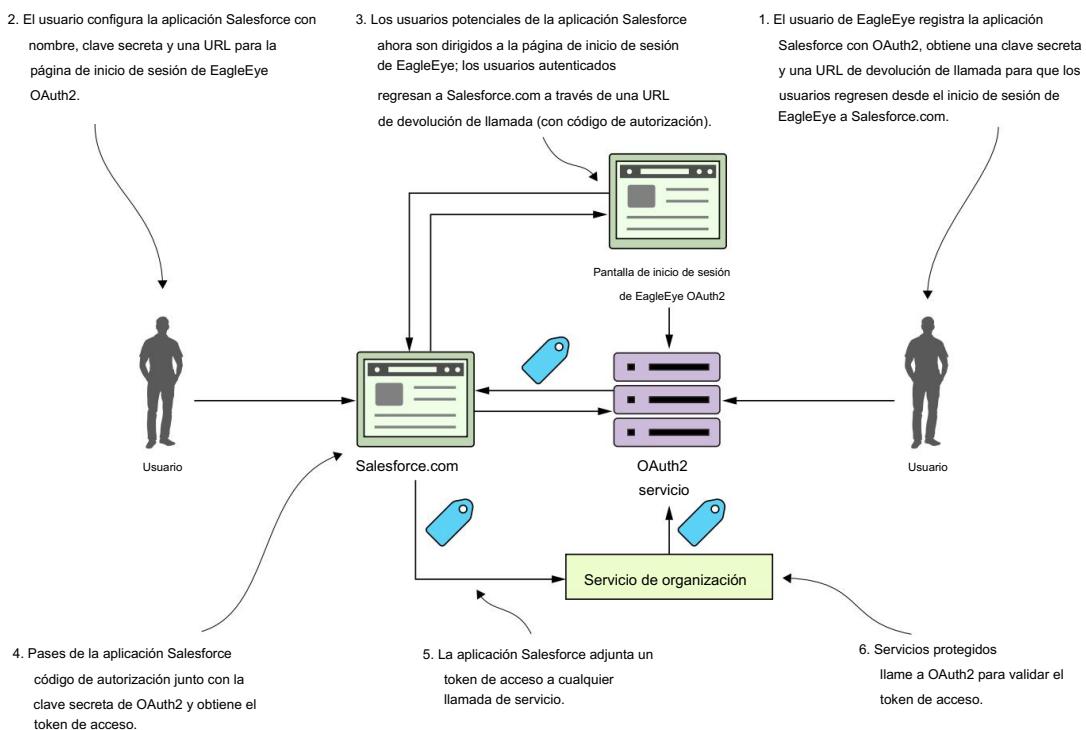


Figura B.3 La concesión del código de autenticación permite que las aplicaciones comparten datos sin exponer las credenciales del usuario.

- 1 El usuario de EagleEye inicia sesión en EagleEye y genera un nombre de aplicación y clave secreta de la aplicación para su aplicación Salesforce. Como parte del proceso de registro, también proporcionarán una URL de devolución de llamada a su proveedor basado en Salesforce. solicitud. Esta URL de devolución de llamada es una URL de Salesforce a la que se llamará después de la El servidor EagleEye OAuth2 ha autenticado las credenciales EagleEye del usuario.

**2** El usuario configura su aplicación Salesforce con la siguiente información:

- El nombre de la aplicación que crearon para Salesforce.
- La clave secreta que generaron para Salesforce.
- Una URL que apunta a la página de inicio de sesión de EagleEye OAuth2
- Ahora, cuando el usuario intenta utilizar su aplicación Salesforce y acceder a su datos de EagleEye a través del servicio de organización, serán redirigidos al Página de inicio de sesión de EagleEye a través de la URL descrita en el punto anterior. El usuario proporcionará sus credenciales de EagleEye. Si han proporcionado EagleEye válido credenciales, el servidor EagleEye OAuth2 generará un código de autorización y redirigir al usuario nuevamente a SalesForce a través de la URL proporcionada en el número 1. El código de autorización se enviará como parámetro de consulta en la URL de devolución de llamada.

**3** La aplicación Salesforce personalizada mantendrá este código de autorización. Nota: este código de autorización no es un token de acceso OAuth2 .

**4** Una vez que se ha almacenado el código de autorización, la aplicación Salesforce personalizada puede presentarle la clave secreta que generaron durante el proceso de registro y el código de autorización de regreso a EagleEye OAuth2 servidor. El servidor EagleEye OAuth2 validará que el código de autorización sea válido y luego devolver un token OAuth2 a la aplicación personalizada de Salesforce. Este código de autorización se utiliza cada vez que el Salesforce personalizado necesita autenticar al usuario y obtener un token de acceso OAuth2 .

**5** La aplicación Salesforce llamará al servicio de organización EagleEye, pasando un token OAuth2 en el encabezado.

**6** El servicio de la organización validará el token de acceso OAuth2 pasado al Llamada de servicio EagleEye con el servicio EagleEye OAuth2 . Si el token es válido, el servicio de la organización procesará la solicitud del usuario.

¡Guau! Necesito salir a tomar aire. La integración de aplicación a aplicación es complicada. La clave a tener en cuenta de todo este proceso es que aunque el usuario haya iniciado sesión Salesforce y están accediendo a los datos de EagleEye, en ningún momento el EagleEye del usuario credenciales expuestas directamente a Salesforce. Después de que el servicio EagleEye OAuth2 generó y proporcionó el código de autorización inicial , el usuario nunca tuvo que proporcionar sus credenciales al servicio EagleEye.

## B.4 Subvención implícita

La concesión de autorización se utiliza cuando ejecuta una aplicación web a través de un entorno de programación web tradicional del lado del servidor como Java o .NET. Qué pasa si su aplicación cliente es una aplicación JavaScript pura o una aplicación móvil que ¿Se ejecuta completamente en un navegador web y no depende de llamadas del lado del servidor para invocar servicios de terceros?

Aquí es donde entra en juego el último tipo de subvención, la subvención implícita. Figura B.4 muestra el flujo general de lo que ocurre en la concesión implícita.

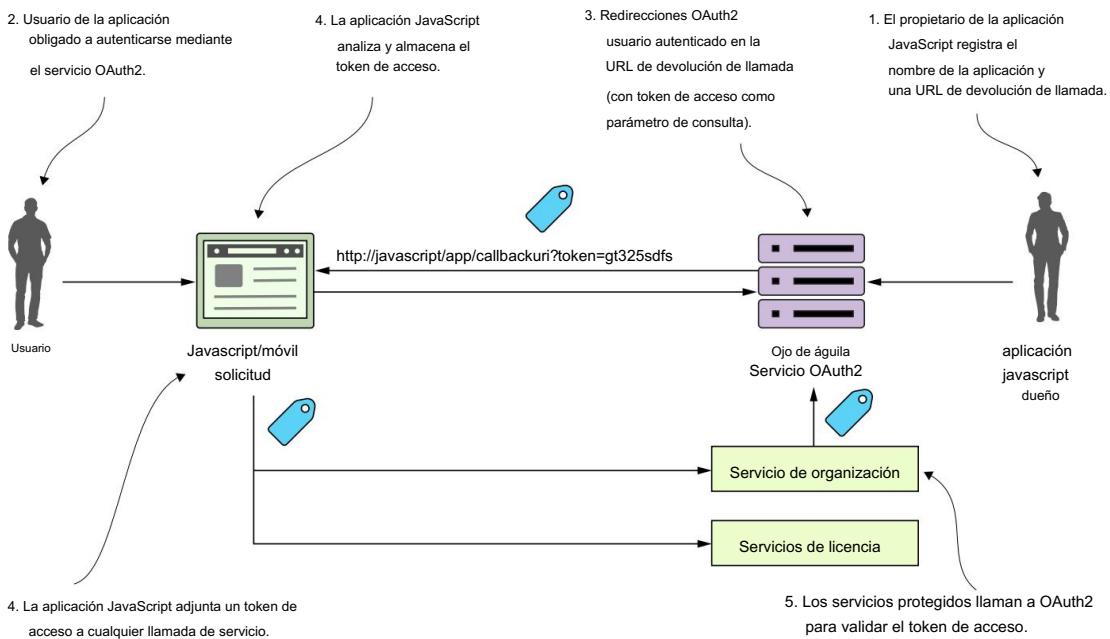


Figura B.4 La concesión implícita se utiliza en una aplicación JavaScript de aplicación de página única (SPA) basada en navegador.

Con una concesión implícita, normalmente estás trabajando con una aplicación JavaScript pura que se ejecuta completamente dentro del navegador. En los otros flujos, el cliente se está comunicando con un servidor de aplicaciones que lleva a cabo las solicitudes del usuario y la aplicación. El servidor interactúa con cualquier servicio descendente. Con un tipo de concesión implícita, todos los interacciones del servicio ocurren directamente desde el cliente del usuario (generalmente un navegador web). En el gráfico B.4 se están llevando a cabo las siguientes actividades:

- 1 El propietario de la aplicación JavaScript ha registrado la aplicación con el Servidor EagleEye OAuth2. Proporcionaron un nombre de aplicación y también una URL de devolución de llamada que será redirigida con el token de acceso OAuth2 para el usuario.
- 2 La aplicación JavaScript llamará al servicio OAuth2. La aplicación JavaScript debe presentar un nombre de aplicación registrado previamente. El servidor OAuth2 obligará al usuario a autenticarse.
- 3 Si el usuario se autentica correctamente, el servicio EagleEye OAuth2 no volverá a emitir un token, pero en su lugar redirige al usuario a una página en la que el propietario del JavaScript solicita la URL registrada en el paso uno. En la URL a la que se redirige, el servicio de autenticación de OAuth2 pasará el token de acceso de OAuth2 como parámetro de consulta.
- 4 La aplicación tomará la solicitud entrante y ejecutará un script JavaScript que analizará el token de acceso OAuth2 y lo almacenará (normalmente como una cookie).

- 5 Cada vez que se llama a un recurso protegido, se presenta el token de acceso OAuth2 al servicio de llamadas.
- 6 El servicio de llamadas validará el token OAuth2 y comprobará que el usuario autorizado para realizar la actividad que están intentando realizar.

Tenga en cuenta varias cosas con respecto a la concesión implícita de OAuth2 :

La concesión implícita es el único tipo de concesión en el que el token de acceso de OAuth2 se expone directamente a un cliente público (navegador web). En la concesión de autorización, la aplicación cliente obtiene un código de autorización devuelto al servidor de aplicaciones que aloja la aplicación. Con la concesión de un código de autorización, al usuario se le concede acceso OAuth2 presentando el código de autorización. El token OAuth2 devuelto nunca se expone directamente al navegador del usuario.

En la concesión de credenciales de cliente, la concesión se produce entre dos aplicaciones basadas en servidor. En la concesión de contraseña, tanto la aplicación que solicita un servicio como los servicios son confiables y pertenecen a la misma organización. Los tokens OAuth2 generados por la concesión implícita son más vulnerables a ataques y uso indebido porque los tokens están disponibles para el navegador. Cualquier JavaScript malicioso que se ejecute en el navegador puede obtener acceso al token de acceso de OAuth2 y llamar a los servicios para los que recuperó el token de OAuth2 en su nombre y, esencialmente, hacerse pasar por usted.

Los tokens OAuth2 de tipo de concesión implícita deben ser de corta duración (1-2 horas).

Debido a que el token de acceso de OAuth2 se almacena en el navegador, la especificación de OAuth2 (y la seguridad de Spring Cloud) no admite el concepto de un token de actualización en el que un token se pueda renovar automáticamente.

## B.5 Cómo se actualizan los tokens

Cuando se emite un token de acceso OAuth2, tiene un período de validez limitado y eventualmente caducará. Cuando el token caduque, la aplicación (y el usuario) que realiza la llamada deberán volver a autenticarse con el servicio OAuth2 . Sin embargo, en la mayoría de los flujos de concesión de Oauth2, el servidor OAuth2 emitirá un token de acceso y un token de actualización. Un cliente puede presentar el token de actualización al servicio de autenticación OAuth2 y el servicio validará el token de actualización y luego emitirá un nuevo token de acceso OAuth2 . Miremos la figura B.5 y analicemos el flujo del token de actualización:

- 1 El usuario inició sesión en EagleEye y ya está autenticado con el servicio Eagle-Eye OAuth2 . El usuario está trabajando felizmente, pero desafortunadamente su token ha caducado.
- 2 La próxima vez que el usuario intente llamar a un servicio (por ejemplo, el servicio de la organización), la aplicación EagleEye pasará el token caducado al servicio de la organización.
- 3 El servicio de la organización intentará validar el token con el servicio OAuth2 , que devuelve un código de estado HTTP 401 (no autorizado) y una carga útil JSON .

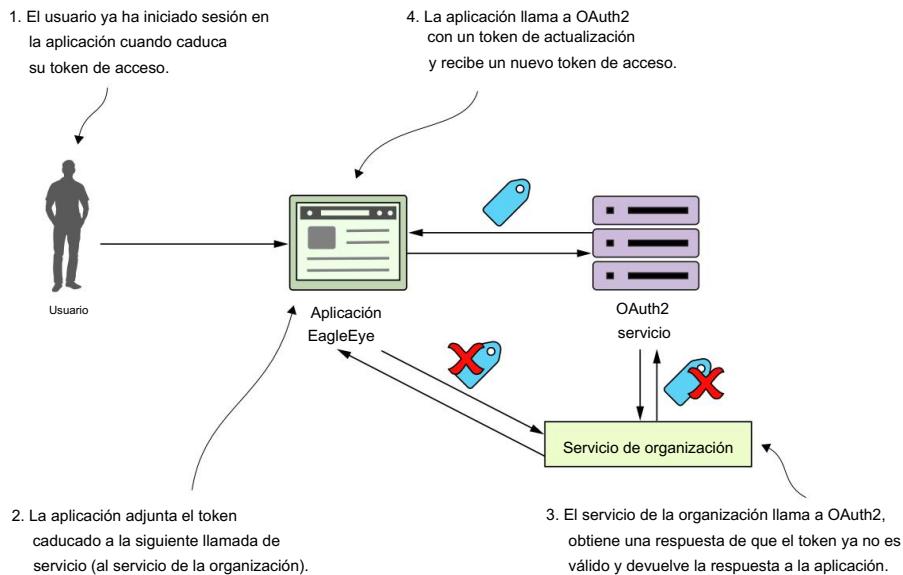


Figura B.5 El flujo del token de actualización permite que una aplicación obtenga un nuevo token de acceso sin obligar al usuario a volver a autenticarse.

indicando que el token ya no es válido. El servicio de organización volverá un código de estado HTTP 401 al servicio de llamada.

- 4 La aplicación EagleEye obtiene el código de estado HTTP 401 y la carga útil JSON indicando el motivo por el cual la llamada falló desde el servicio de la organización. El La aplicación EagleEye luego llamará al servicio de autenticación OAuth2 con el token de actualización. El servicio de autenticación OAuth2 validará el token de actualización y luego devolver un nuevo token de acceso.

# índice

---

## A

---

acceso abstracto 65  
tokens de acceso, OAuth2 210–212  
atributo access\_token 203, 321 cuentas,  
creación en Papertrail 267–268  
Amazon ECS (Elastic Container Service) crea  
clústeres 298–302 implementa  
manualmente servicios Eagle Eye en 303–305 inicia  
servicios  
en 323  
Servicio Amazon ElastiCache 296–298  
Amazon RDS (servicio de base de datos relacional)  
293–296  
Servicios web de Amazon. Ver AWS  
anotación, Spring Cloud 157–158 método  
antMatcher() 209  
Protocolo Apache Avro 50  
Apache Maven 45, 328, 330–333  
Apache de ahorro 50  
API (interfaces de programas de aplicaciones) 225–226  
Clase de aplicación 48  
aplicaciones 194  
arquitectura  
del proceso de construcción/implementación 305–  
308 de la gestión de la configuración 67–69 de  
los microservicios 38–44  
de descubrimiento de servicios 100–  
102 de Spring Cloud Stream 237–238  
carpeta 238  
canal 238  
sumidero  
238 fuente  
238 superficie de ataque, de microservicios 226–  
227 auditoría 319

descripción general de  
usuarios autenticados  
202–205 servicio de protección por  
207–209 servicio de autenticación  
modificando para emitir JWT 214–217  
configuración para EagleEye OAuth2 194–197  
método authenticManagerBean() 201  
AuthenticationServerConfigurer clase 198 autorización  
156, 197 código de  
autorización otorga 339–341 método  
autorizadoGrantTypes() 199  
@Anotación Autowired 114, 182  
AWS (servicios web de Amazon) 292  
Variable AWS\_ACCESS\_KEY 316  
Variable AWS\_SECRET\_KEY 317

## B

---

@Bean etiqueta  
48 atributo before\_install 318, 320  
carpetas 238  
Bitbucket 74  
Clases de arranque  
configuradas en Spring Cloud Config 74–75 escritura  
47–48 servicios  
de arranque, administración de 58–59 ramas.solo  
atributo 316 fragilidad, entre  
servicios 232 \$BUILD\_NAME variable 303  
patrones de construcción/  
implementación, microservicios 25–26 arquitectura de  
canalización de construcción/  
implementación de 305–308  
Implementación con GitHub 311–312  
Implementación con Travis CI 311–312 en acción  
309–311

método buildFallbackLicenseList() 134 implementación del patrón de mamparo 136–138 descripción general 122–123

**C**

CaaS (Contenedor como servicio) 15, 17 caché, borrado cuando se recibe el mensaje 257–258 atributo cache.directories 315 método cacheOrganizationObject() 255 búsquedas en caché con Redis 250–256 configuración del servicio de licencias con Spring Data Dependencias de Redis 250 construir una conexión de base de datos con el servidor de Redis 250–251 definir los repositorios de Spring Data Redis 251–253 usar Redis y el servicio de licencias para almacenar y leer datos de la organización 253–256 método call() 150 Clase invocable 149–150 que captura rastros de mensajes 282–284 CD (entrega continua) 306 canales personalizados, definiendo 256–257 descripción general 238 método checkRedisCache() 255, 284 coreografía de mensajes 235 Patrón de disyuntor CI (integración continua) 305, que implementa 128–133 personalización del tiempo de espera 132–133 temporización de la llamada a los microservicios de la organización 131–132 propiedad circuitoBreaker.errorThresholdPercentage 141, 143 propiedad circuitoBreaker.requestVolumeThreshold 141, 143 propiedad circuitoBreaker.sleepWindowInMillisegundos 141, 143 clases arranque 74 HystrixConcurrencyStrategy, define costumbre 147–149 Java invocable 149–150 aplicaciones cliente, registro en el servicio OAuth2 197–200 concesiones de credenciales de cliente 338–339 mamparo de patrones de resiliencia del cliente 122–123 disyuntor 122 equilibrio de carga del lado del cliente 121–122 respaldo 122

microservicios 21–23 descripción general 120–123 con Netflix Hystrix 119, 126–152 procesamiento de respaldo 133–135 ajuste fino 138–143 implementación del patrón de mamparo 136–138 implementación del disyuntor 128–133 configuración de servidores de licencias para usar 127–128 contexto de subprocessos y 144–152 con Nube de primavera 119–152 procesamiento de respaldo 133–135 implementación del patrón de tabique 136–138 configuración de servidores de licencias para su uso 127–128 resiliencia del cliente, por qué es importante 123–126 equilibrio de carga del lado del cliente 22, 96, 102–104, 121–122 ClientDetailsServiceConfigurer clase 199 ClientHttpRequestInterceptor 219 parámetro clientType 111 nube definida 13–14 microservicios y 15–17 ejecutándose en escritorio 327–335 crear imágenes de Docker 331–333 crear proyectos 330–331 compilar proyectos 330–331 descargar proyectos de GitHub 329–330 iniciar servicios con Docker Redactar 333–335 software requerido 328 descubrimiento de servicios en 100–104 arquitectura de 100–102 usando Netflix Eureka 103–104 usando Spring 103–104 almacenamiento en caché en la nube 249 aplicaciones basadas en la nube 1–34 construcción 12–13 microservicios 17–26 construcción/implementación patrones 25–26 creación con Spring Boot 8–12 patrones de resiliencia del cliente 21–23 patrones de desarrollo básicos 19–20 patrones de registro y seguimiento 24 descripción general 2–5 patrones de enrutamiento 20–21 patrones de seguridad 23 Microservicios de Spring y descripción general de 5–6 5–6 Spring Boot, creación de microservicios con 8 a 12 microservicios basados en la nube 15 a 17 Spring Cloud, construyendo con 26–30 Bibliotecas Netflix Hystrix 29 Bibliotecas de cintas de Netflix 29

- Gateway de servicio Netflix Zuul 29
- implementaciones de aprovisionamiento 30
- Bota de primavera 28
- Configuración de nube de primavera 28
- Seguridad en la nube de primavera 30
- Descubrimiento del servicio Spring Cloud 28
- Detective de la nube de primavera 29–30
- Corriente de nube de primavera 29
- Clústeres de CloudFormation
- 302 en
  - Amazon ECS 298–302
  - Redis 296–298
- microservicios de grano grueso 42
- @Column atributo 84
- commandPoolProperties atributo 140, 143 commandProperties atributo 132, 142, 144 comandos, Netflix Hystrix 149–150
- protocolos de comunicación 20 complejidad, gestión 65–70 configuración creación de servidores de configuración Spring Cloud 70–77 configuración
  - de la clase de arranque Spring Cloud Config 74–75 usando el servidor
  - de configuración Spring Cloud con sistema de archivos 75–77 controlando
  - con el servidor de configuración Spring Cloud 64–95
- integrando Spring Cloud Config con Spring Cliente de arranque 77–
- 89 gestión de la complejidad 65–70
- gestión de la configuración 65–70 servicios de licencia para usar Spring Cloud Config 79 servicios de licencia con Spring Data Redis 250 arquitectura de administración 67–69 implementación 69–70
- microservicios 93–94
- Netflix Zuul 158–159 de tiempo de ejecución central, en Travis CI 315–317 de microservicios, administrando 58–59 de Netflix Hystrix 142–143 protegiendo información confidencial 89–94
  - configurar microservicios para usar cifrado en el lado del cliente 93–94
  - descifrar propiedad 91–93 descargar archivos jar de Oracle JCE para cifrado 90 propiedades de cifrado 91–93 instalar archivos jar de Oracle JCE para cifrado 90 configurar claves de cifrado 91
- rutas
  - recarga dinámica 168 en Netflix Zuul 159–169
- servicio para apuntar al servicio de autenticación OAuth2 206–207 servicios, para apuntar a Zipkin 275–276 Spring Cloud 150–152 conector syslog 267–268 servidor Zipkin 276–277 servidores de configuración Edificio Spring Cloud 70–77 configuración de control con 64–95 propiedades de actualización usando 88 usando con Git 87–88 cableado en fuente de datos usando 83–86 etiqueta @Configuration 48 método configure() 198–199, 201, 207, 209, 222 consistencia 45 Cónsul 28, 69 Contenedor como servicio. Ver CaaS
- Entrega continua. Consulte Integración continua de CD. Ver Controlador CI clase 48
- Parámetro de encabezado de cookie 90 configuración de tiempo de ejecución principal, en Travis CI 315–317 atributo coreSize 137–138 ID de correlación agregando a la respuesta HTTP con Netflix Zuul 272–274
  - postfiltro del edificio que recibe 182–184 prefiltro del edificio que genera 173–182
  - Spring Cloud Sleuth y 260–263 utilizan llamadas de servicio 176–182 RestTemplate personalizado para garantizar que el ID de correlación se propague hacia adelante 181–182
  - HttpContext para hacer que los encabezados HTTP sean fácilmente accesibles 179–180
  - HttpContextFilter para interceptar entrantes Solicitudes HTTP 178–179
  - HttpContextInterceptor para garantizar la correlación La identificación se propaga hacia adelante 181–182 acoplamiento flojo 233–234 apretado entre servicios 231–232 administración de credenciales 23 CRM (gestión de relaciones con el cliente) 2, 36 inquietudes transversales 170 CRUD (Crear, Leer, Actualizar, Eliminar) 43, 253 Campos personalizados de clase 85 de CrudRepository, analizando los intervalos personalizados de JWT 222–224, agregando 284–287, personalizando el tiempo de espera en el disyuntor 132–133
- D
  - Parámetros D 82 fuente de datos, cableado usando Spring Cloud 83–86

- transformaciones de datos 45
- base de datos, construcción de conexión al servidor Redis 250–251 depuración 250–251
- 305 descomposición
- de problemas comerciales 38–40 /decrypt endpoint 92
- propiedad de descifrado 91–93
- Anotación @DefaultProperties 142
- DelegatingUserContextInvocable 149–150
- DelegatingUserContextCallable servidor de dependencias clase 150,
- configuración 79
  - Detective de la nube de primavera 275
  - Implementación de Spring Data
- Redis 250
- Microservicios EagleEye 303–305 288
    - arquitectura del proceso de construcción/ implementación 305–
    - 308 proceso de construcción/implementación en acción
    - 309–311 habilitación del servicio para construir en Travis CI
    - 312–325 implementación del proceso de construcción/
      - implementación 311–312 con EagleEye 290–305
      - ensamblaje de servicios 56–58
    - escritorios, ejecución en la nube 327–335 creación de imágenes de Docker 331–333 creación de proyectos 330–331 compilación de proyectos 330–331 descarga de proyectos de GitHub 329–330 lanzamiento de servicios con Docker Compose 333–335
    - software requerido 328
    - patrones de desarrollo, microservicios 19–20
    - DevOps (operaciones de desarrollador) 290
    - Ingeniero de DevOps 38, 63
    - descubrimiento de servicios 59–60, 96–118
      - arquitectura de 100–102 mapeo automatizado de rutas usando 159–160 construcción del servicio Spring Eureka 105–107 en la nube 100–104 servicios de localización 97–99 mapeo manual de rutas utilizando 161–165 servicios de registro con el servidor Eureka basado en Spring 107–110
      - usando Netflix Eureka 103–104 usando Spring 103–104 usando para buscar servicios 111–118
    - DiscoveryClient, buscando instancias de servicio con 112– 114 almacenamiento
    - en caché distribuido 249–258 borrando el caché cuando se recibe un mensaje 257–258
  - definir canales personalizados 256–257 usar Redis para almacenar en caché las búsquedas 250–256 configurar el servicio de licencias con Spring Data Dependencias de Redis 250
  - construcción de una conexión de base de datos al servidor de Redis 250–251
  - definición de repositorios de Spring Data Redis 251– 253 uso de Redis y servicio de licencias para almacenar y leer datos de la organización 253–256 sistemas distribuidos, complejidad de la construcción 44 seguimiento distribuido con Spring Cloud Sleuth 259– 287 ID de correlación y 260–263
  - agregación de registros y 263–274 con Zipkin 259, 274–287 agregando intervalos personalizados 284–287 capturando rastros de mensajes 282–284 configurando el servidor 276–277 configurando servicios para que apunten a 275–276 instalando el servidor 276–277 integrando las dependencias de Spring Cloud Sleuth 275 configuración niveles de seguimiento 278 seguimiento de transacciones 278–280 visualización de transacciones complejas 281–282
  - DNS (Servicio de nombres de dominio) 97
  - Docker
    - crea imágenes, en imágenes Travis CI 321–322, genera resultados 331–333 y redirige a Papertrail 268–269
    - Docker Compose, iniciar servicios con 333–335 Docker Hub, enviar imágenes a 322–323 Complemento Docker Maven 328, 331 comando docker ps 305 variable DOCKER\_PASSWORD 316 variable DOCKER\_USERNAME 316 comando docker-compose 334 docker-compose.yml 334 docker.sock 268 descarga proyectos de GitHub 329–330 durabilidad 234 recarga dinámica, configuración de ruta 168 filtros de ruta dinámica, construcción 184–191 ruta de reenvío 188–189 implementación del método run() 187–188 esqueleto de 186 enrutamiento dinámico 156
  - mi

---

  - EagleEye 290–302
    - configuración de usuarios 200–202
    - creación de clústeres de Amazon ECS 298–302

- creando una base de datos PostgreSQL usando Amazon RDS 293–296
- creando un clúster de Redis en Amazon ElastiCache Servicio 296–298
- implementación 302–
- 305 configuración del servicio de autenticación OAuth2 196–197
- EBS (almacenamiento en bloque elástico) 299
- ECS (Servicio de contenedor elástico) 15
- Comando ecs-cli 303, 316–317
- Comando configure ecs-cli 303 Comando ps ecs-cli 304 ecsInstanceRole 301
- EDA (Arquitectura impulsada por eventos) 229
- ELB (equilibrador de carga empresarial) 319
- @EnableAuthorizationServer anotación 197
- @EnableBinding anotación 240, 245–246
- @EnableCircuitBreaker anotación 31, 128
- Anotación @EnableDiscoveryClient 112–114, 117
- @EnableEurekaClient anotación 31
- @EnableFeignClients anotación 112, 116
- @EnableResourceServer anotación 206
- Anotación @EnableZipkinServer 276–277
- Anotación @EnableZipkinStreamServer 276–277
- @EnableZuulProxy anotación 158
- @EnableZuulServer anotación 158
- /encriptar punto final 91
- ENCRYPT\_KEY variable de entorno 91 variables cifradas 317 cifrado configuración de microservicios para usar en el lado del cliente 93–94
- descargar archivos jar de Oracle JCE 90
- instalar archivos jar de Oracle JCE 90 propiedad 91–
- 93 configurar claves
- 91 claves de cifrado, configurar 91
- puntos finales, proteger 195–205
- autenticar usuarios 202–205 configurar usuarios de EagleEye 200–202 registrar aplicaciones cliente con el servicio OAuth2 197–200 configurar el servicio de autenticación EagleEye OAuth2 196–197
- Anotación @Entity 84
- etiqueta de entorno 334
- variables de entorno 81 Etcd 69
- Eureka 28, 69 atributo
- eureka.client.fetchRegistry 106, 108 atributo eureka.client.registerWithEureka 106, 108
- Propiedad eureka.instance.preferIpAddress 108
- Atributo eureka.serviceUrl.defaultZone 109 procesamiento de eventos 20 arquitectura basada en eventos, con Spring Cloud Stream 228–258 arquitectura 237–238 almacenamiento en caché distribuido 249–258 desventajas de la arquitectura de mensajería 235 uso de mensajería para comunicar cambios de estado entre servicios 233–234 uso de un enfoque sincrónico de solicitud-respuesta para comunicar el cambio de estado 230–232 escribir consumidor de mensajes 238–249 escribir productor de mensajes 238–249 ejecución.isolation.thread.timeoutInMilliseconds propiedad 132 expires\_in atributo 204 que extiende JWT (tokens web JavaScript) 220–222
- 
- F
- 
- equilibrador de carga F5 97
- FaaS (funciones como servicio) 15 patrón de respaldo 122 estrategia de respaldo, procesamiento 133–135 método de respaldo 134, 143
- Campos de anotación 117 de @FeignClient, análisis del sistema de archivos JWT 222–224, uso con servidores de configuración Spring Cloud 75–77
- filterOrder() método 175 filtros
- edificio 157
- generando ID de correlación 173–182 en Netflix Zuul 169–173 recibiendo ID de correlación 182–184
- método filterType() 174–175, 186 Clase FilterUtils 174–175
- microservicios detallados 42 flexibilidad 234 método forwardToSpecialRoute() 187–189
- 
- DISTRIBUO
- 
- distribución geográfica 16
- OBTENER punto final HTTP 10
- método getAbRoutingInfo() 187 método getCorrelationId() 175–176 método getInstances() 114 método getLicense() 86, 112 método getLicenses() 51 método getLicensesByOrg() 130, 137, 146 método getOrg() 285 getOrganization() método 117, 255

función getOrganizationId() 223 getOrgDbCall 285 Git, uso con servidores de configuración Spring Cloud 87–88 GitHub descargar proyectos del 329 al 330 implementar el proceso de construcción/implementación con 311–312 Variables GITHUB\_TOKEN 317, 321 Gradle 10 concede, en OAuth2 336–344 concede código de autorización 339–341 concede credencial de cliente 338–339 concede implícitamente 341–343 concede contraseña 337–338 maravilloso 10

***h***

HAProxy 97 salud, de microservicios 60–62 método helloRemoteServiceCall 31–32 escalabilidad horizontal 16, 99 Encabezados HTTP 179–180 Solicitudes HTTP 178–179 Respuesta HTTP 272–274 Códigos de estado HTTP 43 HTTP, uso para la comunicación de servicios 224 HttpStatus clase 207 dependencias hystrix-javanica 127 @HystrixCommand anotación 32, 129–135, 137–138, 140, 142–144, 149–150 Estrategia de concurrencia de Hystrix 147–152 configurar Spring Cloud para usar 150–152 definir clases personalizadas 147–149 definir clases invocables de Java para injectar contexto de usuario en los comandos de Hystrix 149–150 HystrixRuntimeException 135

***I***

IaaS (infraestructura como servicio) 13–14, 17 @Id anotación 84 IETF (Grupo de trabajo de ingeniería de Internet) 213 atributo de servicios ignorados 163–164 creación de imágenes en Docker 331–333 creación en Travis CI 321–322 envío a Docker Hub 322–323 servidores inmutables 25, 308 concesiones implícitas 341–343 acceso a puertos de entrada 99 inboundOrgChanges 256 servicios individuales 205

protocolo de estilo de infección 102 infraestructura 25 método init() 151 @Input anotación 256 instalación del servidor Zipkin 276–277 integración Spring Cloud Config con el cliente Spring Boot 77–89 configurando el servicio de licencias para usar Spring Cloud Config 79–82 leer propiedades directamente usando la anotación @Value 86–87 actualizar propiedades usando el servidor de configuración Spring Cloud 88 configurar el servicio de licencias Spring Cloud Configurar las dependencias del servidor 79 usando el servidor de configuración Spring Cloud con Git 87–88 cableado en la fuente de datos utilizando el servidor de configuración Spring Cloud 83–86 Dependencias de Spring Cloud Sleuth con Zipkin 275 pruebas de integración 307 interceptación de solicitudes HTTP entrantes 178–179 diseño de interfaz 20 invocación de servicios con el cliente Netflix Feign 116–118 con Spring RestTemplate compatible con Ribbon 114–116

***J***

Pila J2EE 6 Java que crea microservicios con 45–53 proyectos esqueleto 46–47 Controlador Spring Boot 48–53 escritura de la clase Bootstrap 47–48 define una clase invocable para injectar contexto de usuario en los comandos Hystrix 149–150 java.lang.ThreadLocal 179 JCE (Extensión de criptografía) Java ) 90 JedisConnectionFactory 250 JPA (Anotaciones de persistencia de Java) 84 JSON (Notación de objetos de JavaScript) 5, 20 JWT (Tokens web de JavaScript) que consumen microservicios 218–220 Ampliación de 220–222 Modificación del servicio de autenticación para emitir 214–217 OAuth2 y 213–224 Análisis campo personalizado de 222–224 método jwtAccessTokenConverter() 215 clase JWTOAuth2Config 216, 222 clase JWTTokenStoreConfig 215, 218, 221

- 
- |
- Clase de licencia 51, 84  
método licencia.withComment() 86 Clase  
LicenseRepository 84–85 Clase  
LicenseService 84–86 Método  
LicenseService.getLicensesByOrder() 147  
    Clase  
LicenseServiceController 111 punto final de  
licencia estática 166 configuración  
de servicios de  
    licencia para usar Spring Cloud Config  
        79–82  
    configurar con Spring Data Redis 250 dependencias  
    del servidor 79 configurar para  
    usar Netflix Hystrix 127–128 configurar para usar  
    Spring Cloud 127–128 usar con Redis para  
    almacenar y leer datos de la organización 253–256 escribir  
        mensajes al  
    consumidor en 244–247 licencias, agregar  
Spring Cloud Sleuth al grupo de licencias 261–263 247
- LinkedBlockingQueue 138 servicios  
de localización 97–99 bloqueo  
de puertos de red innecesarios 226–227 agregación  
de registros, Spring Cloud Sleuth y 263–274  
    agregando  
    ID de correlación a la respuesta HTTP con  
        Netflix Zuul 272–274  
    configuración del conector syslog 267–268  
    creación de una cuenta Papertrail 267–268  
    implementación de Papertrail 265–266  
    implementación de Spring Cloud Sleuth 265–266  
    redirección de la salida de Docker a Papertrail  
        268–269  
    buscando ID de seguimiento de Spring Cloud Sleuth en  
        Papertrail 270  
correlación de registros  
24 método loggerSink() 246  
registro 156  
patrones de registro y rastreo, microservicios 24 controlador  
de registro, Docker 270 búsquedas,  
almacenamiento en caché con Redis 250–256  
    configuración del servicio de licencias con Spring Data  
        Dependencias de Redis 250  
        construir una conexión de base de datos con el  
            servidor de Redis  
    250–251 definir los repositorios de Spring Data  
        Redis  
    251–253 usar Redis y el servicio de licencias para  
        almacenar y leer datos de la organización  
253–256 acoplamiento flexible 233–234
- METRO
- método main() 48, 74 mapeo  
de rutas  
    automatizadas usando el descubrimiento de servicios  
        159–160 manual usando el descubrimiento de  
            servicios 161–165 manual usando URL estáticas 165–167  
    Maven BOM (lista de materiales) 71 atributo  
maxQueueSize 138 propiedad  
maxQueueSize 138, 143  
MDA (Arquitectura basada en mensajes) 229  
Mercurio 311  
manejo de mensajes, semántica de 235  
servicios de mensajes 247–249  
Clase MessageChannel 241  
coreografía  
    de mensajes de 235  
    borrado de caché cuando se reciben 257–258  
    visibilidad de 235  
    consumidor de escritura 238–249  
        en el servicio de licencias 244–247  
        servicios de mensajes en acción 247–249  
    productor de escritura 238–249  
        en el servicio de organización 239–243  
        servicios de mensajes en acción 247–249  
arquitectura de mensajería, desventajas de 235  
seguimientos de mensajería, captura de 282–  
284 mensajes, comunicación de cambios de estado entre  
servicios con 233–234  
durabilidad 234  
flexibilidad 234  
acoplamiento flexible 233–234  
    escalabilidad 234  
colección métrica 156  
Propiedad metrics.rollingStats.numBuckets 141 Propiedad  
metrics.rollingStats.timeInMilisegundos 141  
    Propiedad  
metricsRollingStats.numBuckets 143 Propiedad  
metricsRollingStats.timeInMilisegundos 143  
microservicios  
17–26 acceso con puerta  
    de enlace de servicios 225 patrones de  
        compilación/implementación 25–26 creación  
        en Travis CI 321–322 creación con  
        Spring Arranque 8–12 patrones de  
        resiliencia del cliente 21–23 nube y 15–  
        17  
    estado de comunicación de 60 a 62  
    configuración para utilizar cifrado en el lado del cliente  
        93–94  
    consumiendo JWT en 218–220  
    patrones de desarrollo básicos 19–20  
    implementando  
        288 arquitectura de canal de construcción/  
            implementación 305–308

canal de construcción/  
 implementación de microservicios (continuación)  
 en acción 309–311  
 permitir que el servicio se incorpore en Travis CI 312–325  
 implementar el proceso de construcción/  
 implementación 311–  
 312 con EagleEye 290–305  
**Java**, construyendo con 45–53  
 proyectos esqueleto 46–47  
 Controlador Spring Boot 48–53 que  
 escribe la clase Bootstrap 47–48 que  
 limita la superficie de ataque bloqueando los  
 puertos de red innecesarios 226–227  
 patrones de registro y rastreo 24  
 administración de la configuración de 58–  
 59 descripción general 2–5  
 proteger un único punto final 195–205 patrones  
 de enrutamiento 20–21  
 asegurar  
 Servicio de organización JWT y  
 OAuth2 213–224 que utiliza OAuth2 205–212 con  
 patrones de seguridad  
**OAuth2** 193–195 23  
**Spring Boot**, construyendo con 35, 45–63  
 diseñar arquitectura de microservicio 38–44 para  
 tiempo de ejecución 53–62  
 proyectos de esqueleto 46–47  
 Controlador Spring Boot 48–53  
 escribiendo la clase Bootstrap 47–48  
 Llame al 131–132 cuando no lo  
 use 44–45  
 complejidad de la construcción de sistemas distribuidos 44  
 consistencia 45  
 transformaciones de datos 45  
 expansión del  
 servidor 44 tipos de aplicaciones  
 44 mvn spring-boot:ejecutar comando 76

notas

NAS (almacenamiento de área de red) 123  
 comando nc 332

**NetflixEureka**

construir servicio usando Spring Boot 105–107 configurar  
**Netflix Zuul** para comunicarse  
 con 158–159  
 registrar servicios con un servidor basado en  
 Spring 107–110  
 descubrimiento de servicios usando 103–104

**Netflix fingir** 116–118

**Netflix Hystrix** y

Spring Cloud 119–152 patrones de  
 resiliencia del cliente con 126–127 procesamiento  
 de respaldo 133–135

ajuste fino 138–143  
 implementación del patrón de mamparo 136–138  
 implementación del disyuntor 128–133  
 configuración de servidores de licencias para usar 127–  
 128 contexto de subprocessos y 144–  
 152 comandos 149–150  
 configuración de 142–143  
 contexto de subprocessos y 144–152  
 Estrategia de concurrencia de Hystrix 147–152  
 Hilo local 144–147  
**Cinta de Netflix** 29, 114–116  
**Netflix Zuul**  
 agregar ID de correlación a la respuesta HTTP con  
 272–274  
 construir prefiltro generando correlación  
 ID 173–182  
 configuración de rutas en 159–169  
 mapeo automatizado de rutas a través del  
 descubrimiento de  
 servicios 159–160 recarga dinámica de la configuración  
 de rutas 168 mapeo manual de rutas usando estática  
 URL 165–167  
 mapear rutas manualmente usando el  
 descubrimiento de  
 servicios 161–165 tiempos de  
 espera de servicio y 169 configuración para comunicarse con Netflix  
 Eureka 158–159 filtra  
 la ruta de servicio  
 169–173 con 153–157, 159–191  
 construir un filtro de ruta dinámico 184–191  
 construir una correlación de recepción del post-filtro  
 ID 182–184  
 puertas de enlace de servicios  
 154–156 configuración del proyecto Spring  
 Boot 157 usando la anotación Spring Cloud para  
 servicios 157–158  
 puertos de red, bloqueo para limitar la superficie de ataque de  
 microservicios 226–  
 227 atributo de notificaciones 316

oh

**OAuth2**

agregando frascos a servicios individuales 205  
 tipos de concesión 336–  
 344 código de autorización 339–341  
 credencial de cliente 338–339  
 implícita 341–343  
 contraseña 337–338  
**JWT** y 213–224  
 consumen JWT en microservicios 218–220 extienden  
 JWT 220–222 modifican el  
 servicio de autenticación para emitir  
 JWT 214–217

- analizando el campo personalizado de JWT 222–224
- propagar tokens de acceso 210–212 proteger el servicio de la organización con 205–212
  - agregar archivos jar de OAuth2 a servicios individuales 205
  - agregar Spring Security a servicios individuales 205 configurar el servicio para que apunte al servicio de autenticación OAuth2 206–207 definir qué puede acceder a los servicios 207–210 definir quién puede acceder a los servicios 207–210 propagar los tokens de acceso de OAuth2 210–212 proteger un punto final único con 195–205 autenticar usuarios 202–205 configurar usuarios EagleEye 200–202 registrar aplicaciones cliente con el servicio OAuth2 197–200
  - configurar la autenticación EagleEye servicio 196–197
  - actualizar tokens 343–344 registrar aplicaciones cliente con el servicio 197–200 proteger microservicios con 193–195 configurar el servicio de autenticación EagleEye 196–197
- Clase OAuth2Config 198–199, 201, 215
- ODS (almacén de datos operativos) 134
- Oracle JCE (Java Cryptography Extension) descarga de archivos jar para cifrado 90 instalación de archivos jar para cifrado 90 datos de la organización 253–256 ID de la organización 242 servicios de la organización inflexibles
  - al agregar nuevos consumidores a los cambios en 232
  - proteger 205–212 escribir el productor de mensajes en 239–243
- OrganizationChangeHandler clase 257
- OrganizationChangeModel clase 242
- OrganizationId parámetro 117
- OrganizaciónRedisRepository interfaz 252
- OrganizationRedisRepositoryImpl 253
- Organizationservice 159–160
- orgChangeTopic 245, 283 acceso al puerto de salida 99 salida, redireccionamiento a Papertrail 268–269 método de salida() 241
- PaaS (Plataforma como servicio) 13–14, 17
- Conjunto de servicios de empaquetado 56–58
- Papertrail 29
  - creando cuenta 267–268
  - implementando 265–266
- redirigir la salida de Docker a 268–269 en busca de ID de seguimiento de Spring Cloud Sleuth en 270
- analizando campos personalizados de JWT 222–224
- concesiones de contraseña 337–338
- @PathVariable anotación 51, 117 mamparo de patrones 122–123, 136–138 disyuntor 122, 128, 131–133 mamparo de resiliencia del cliente 122–123
  - disyuntor 122 equilibrio de carga del lado del cliente 121–122 respaldo 122 descripción general 120–123
- con Netflix Hystrix 119, 126–152 con Spring Cloud 119, 127–128, 133–152 equilibrio de carga del lado del cliente 121–122 respaldo 122
- PCI (Industria de tarjetas de pago) 99, 224 modelo peer-to-peer 101
- PEP (punto de aplicación de políticas) 154–155, 225
- Servidores Phoenix 25, 308
- servidores físicos, 15
- canalizaciones, arquitectura de construcción/implementación de 305 a 308
  - Implementación con GitHub 311–312
  - Implementación con Travis CI 311–312 en acción 309–311 Plataforma
- como servicio. Consulte las pruebas de la plataforma PaaS 308, 323–325
- Variable PLATFORM\_TEST\_NAME 319 etiqueta de puerto 334 filtro posterior, creación para recibir ID de correlación 182–184 Base de datos Postgres 72, 77, 80, 82–84, 91 Base de datos PostgreSQL, creación con Amazon RDS 293–296 CARTERO 12
- herramientas preconstruidas, en Travis CI 318–319 prefiltro, para generar ID de correlación 173–182 atributo preferIpAddress 108 API privadas, servicios de zonificación en 225–226 propagación de tokens de acceso OAuth2 210–212 propagación 23 descifrado de propiedades
  - 91–93 lectura directa usando la anotación @Value 86–87
  - cifrado 91–93 actualización utilizando servidores de configuración Spring Cloud 88
- propagar ID de correlación
  - con RestTemplate 181–182
  - conUserContextInterceptor 181–182

- recurso protegido 193 proteger  
puntos finales  
  195–205 autenticar  
    usuarios 202–205 configurar usuarios  
    EagleEye 200–202 registrar aplicaciones cliente  
    con el servicio OAuth2 197–200 configurar el servicio de autenticación EagleEye  
  OAuth2 196–197 servicios por usuarios autenticados 207–209 a través de una función  
    especifica 209–210 protección del servicio de la organización, con  
  OAuth2 205–212  
aprovisionamiento de  
30 API públicas, servicios de zonificación en el método  
225–226 PublishOrgChange() 242
- 
- q**
- atributo queueSizeRejectionThreshold 138
- 
- R**
- RabbitMQ 89  
propiedades de lectura usando la anotación @Value 86–87
- readLicensingDataFromRedis 285 redirigiendo  
la salida de Docker a Papertrail 268–269  
Redis  
  construir una conexión de base de datos con el servidor 250–251  
  crear clústeres en Amazon ElastiCache Servicio 296–298 para  
  almacenar en caché las búsquedas 250–256 configurando el servicio de licencias con Spring Data Dependencias de Redis 250  
  construcción de una conexión de base de datos al servidor de Redis 250–251  
  definir repositorios de Spring Data Redis 251–253 usar Redis y el servicio de licencias para almacenar y leer datos de la organización 253–256 usar el servicio de licencias para almacenar y leer datos de la organización 253–256  
RedisTemplate 250 /refresh  
endpoint 89 atributo de refresco\_token 204, 344 propiedades de actualización  
  utilizando servidores de configuración de Spring Cloud 88 tokens 343–344
- Anotación @RefreshScope 88–89 registro de aplicaciones cliente 197, 200
- servicios, con servidor Eureka basado en Spring 107–110  
registro, de servicios 59–60 configuración de ruta de recarga 168 repositorios, Spring Data Redis 251–253  
@RequestMapping anotación 51, 117 propietario del recurso 194  
RespuestaClase corporal 50 Filtro de respuesta 173 Punto final REST 77 Filosofía DESCANSO 43 Orientado a REST (Estado Representacional Transferencia) 6  
@RestController anotación 50 Plantilla de descanso 114–116 compatible con cinta para garantizar que se propague el ID de correlación adelante 181–182  
Método restTemplate.exchange() 116 Variable \$RESULTS 324 Método retrieveOrgInfo() 112 Proyecto Ribbon 29 filtros de ruta, construcción dinámica 184–191 ruta de reenvío 188–189 implementación del método run() 187–188 esqueleto de 186
- rutas  
  configuración en Netflix Zuul 159–169  
  mapeo automatizado de rutas a través del descubrimiento de servicios 159–160 recargar dinámicamente la configuración de ruta 168 mapeo manual de rutas usando estática URL 165–167  
  mapeo de rutas manualmente usando el descubrimiento de servicios 161–165 tiempos de espera de servicio y 169 reenvío 188–189 mapeo automatizado  
    usando el descubrimiento de servicios 159–160 manual usando el descubrimiento de servicios 161–165 manual usando URL estáticas 165–167  
  patrones de enrutamiento, microservicios 20–21  
  configuración de tiempo de ejecución, CI 315–317  
  método run() 174–176, 187 tiempo de ejecución, creación de microservicios para 53–62  
    comunicación del estado del microservicio 60–62 ensamblaje de servicios 56–58 arranque de servicios 58–59 descubrimiento de servicios 59–60 registro de servicios 59–60
- 
- S**
- SaaS (software como servicio) 13–14 Clase de muestreo 278

- escalabilidad 234
- Atributo de alcance 204
- método de alcances() 199
- búsqueda de ID de seguimiento de Spring Cloud Sleuth 270 atributo searchLocations 75–76 método secreto() 199
- Capa de sockets seguros.
- Consulte SSL protegiendo microservicios 192, 224–227.
  - acceder a microservicios con puerta de enlace de servicios 225
  - JWT y OAuth2 213–224 limitar la superficie de ataque de los microservicios 226–227 proteger el servicio de la organización mediante OAuth2 205–212
  - protegiendo un único punto final con OAuth2 195–205
  - proteger un punto final único con Spring 195–205 usar HTTP para comunicación de servicios 224 usar SSL para comunicación de servicios 224 con OAuth2 193–195 zonificar servicios en API pública y privada
  - API 225–226
  - patrones de seguridad, microservicios 23
  - propiedad security.oauth2.resource.userInfoUri 206 semántica del manejo de mensajes 235 Aislamiento basado en SEMAPHORE 144 método send() 241–242
  - información confidencial, protección 89–94
    - configurar microservicios para usar cifrado en el lado del cliente 93–94
    - descifrar propiedad 91–93 descargar archivos jar de Oracle JCE para cifrado 90 propiedades de cifrado 91–93 instalar archivos jar de Oracle JCE para cifrado 90 configurar claves de cifrado 91
  - servidor
    - dependencias 79
  - Configuración de Zipkin 276–277
    - instalación 276–277
  - expansión del servidor
  - 44 atributo server.port 106
  - implementación del ensamblaje de servicios 56–58 empaquetado 56–58
  - 58 arranque del servicio, gestión de la configuración de microservicios 58–59
  - llamadas de servicio, utilizando ID de correlación en 176–182
    - RestTemplate personalizado para garantizar que el ID de correlación se propague hacia adelante 181–182
  - UserContext para hacer que los encabezados HTTP sean fácilmente accesibles 179–180
- UserContextFilter para interceptar solicitudes HTTP entrantes 178–179
- UserContextInterceptor para garantizar que el ID de correlación se propague hacia adelante 181–182
- descubrimiento de servicios 96–118
  - arquitectura de 100–102 mapeo automatizado de rutas utilizando 159–160 construcción del servicio Spring Eureka 105–107 en la nube 100–104
- arquitectura de 100–102
  - usando Netflix Eureka 103–104 usando Spring 103–104 servicios de localización 97–99 mapeo manual de rutas usando 161–165 registro de servicios con el servidor Eureka basado en Spring 107–110
- usando Netflix Eureka 103–104 usando Spring 103–104 usando para buscar servicios 111–118
  - invocar servicios con el cliente Netflix Feign 116–118
  - invocar servicios con Spring compatible con Ribbon RestTemplate 114–116 buscando instancias de servicio con Spring DiscoveryClient 112–114 granularidad del servicio 20, 41–43 interfaces de servicio 43–44
- monitoreo de servicios 54
- registro de servicios 59–60
- enrutamiento de servicios con Netflix Zuul 153–157, 159–191 construir un filtro de ruta dinámico 184–191 construir una correlación de recepción del post-filtro ID 182–184
  - creando correlación de generación de prefiltro ID 173–182
- configurando rutas 159–169 configurando para comunicarse con Netflix Eureka 158–159 filtros 169–173 puertas de enlace de servicios 154–156
  - configuración del proyecto Spring Boot 157 usando la anotación Spring Cloud para los servicios 157–158 con Spring Cloud 153–157, 159–191
    - construir un filtro de ruta dinámico 184–191 construir una correlación de recepción del post-filtro ID 182–184
- puertas de enlace de servicios 154–156 usando anotaciones para los servicios Netflix Zuul 157–158
- inicios de servicios 110
- tiempos de espera de servicios 133, 169
- Clase de instancia de servicio 114

- propiedad servicename.ribbon.ReadTimeout 169 fragilidad de los servicios
  - entre 232 cambios de estado de comunicación entre con mensajería 233–234 durabilidad 234
    - flexibilidad 234 acoplamiento
    - flexible 233–234
    - escalabilidad 234
    - configuración para apuntar a Zipkin 275–276 definición de qué tiene acceso para proteger el servicio por parte de usuarios autenticados 207–209 proteger el servicio a través de un rol específico 209–210 definir quién tiene acceso a 207, 210 proteger el servicio por parte de usuarios autenticados 207–209 proteger el servicio a través de un rol específico 209–210 invocar con el cliente Netflix Feign 116–118 con Spring RestTemplate compatible con Ribbon 114–116
- lanzamiento con Docker Compose 333–335 localización 97–99 búsqueda usando descubrimiento de servicios 111–118 invocando servicios con el cliente Netflix Feign 116–118 invocando servicios con Ribbon-aware Spring
  - RestTemplate 114–116 buscando instancias de servicio con Spring DiscoveryClient 112–114 protección mediante
    - usuarios autenticados 207–209 a través de una función específica 209–210 registro con un servidor Eureka basado en Spring 107–110 comenzando en Amazon ECS 323 acoplamiento estrecho entre 231–232 zonificación en API pública y API privada 225–226 puertas de enlace de servicios que acceden a microservicios con 225 descripción general 154–156 método setContext() 150 método setCorrelationId() 176 método shouldFilter() 175, 186 Servicio de cola simple. Consulte SQS SimpleHostRoutingFilter clase 189 Inicio de sesión único. Consulte SSO sumideros 238 proyectos básicos 46–47 SOAP (Protocolo simple de acceso a objetos) 50 Software como servicio. Ver SaaS ingeniero de software 63 Clase de fuente 241
- código de control de fuente, etiquetado en Travis CI 320–321 fuentes 238 ID de tramo 262 tramos, personalizado 284–287 Rutas Especiales 185 Rutas EspecialesFiltro 173, 184, 186 SPIA (Aplicaciones de Internet de una sola página) 50 Microservicios de Spring y descripción general de 5–6 5–6 protección de punto final único con 195–205 autenticación de usuarios 202–205 configuración de usuarios de EagleEye 200–202 registro de aplicaciones cliente con el servicio OAuth2 197–200 configuración del servicio de autenticación EagleEye OAuth2 196–197 registro de servicios con Netflix Servidor Eureka 107–110 descubrimiento de servicios usando 103–104 Módulo actuador de resorte 61 Spring Boot crea microservicios con 8–12, 35, 45–63 diseña arquitectura 38–44 para tiempo de ejecución 53–62 proyectos esqueleto 46–47 Controlador Spring Boot 48–53 escritura de la clase Bootstrap 47–48 creación del servicio Netflix Eureka con el cliente 105–107 77–89 controlador 48–53 configurar el proyecto Netflix Zuul 157 cuándo no usar microservicios 44–45 complejidad de la construcción de sistemas distribuidos 44 consistencia 45 transformaciones de datos 45 expansión de servidores 44 tipos de aplicaciones 44 Spring Cloud y Netflix Hystrix 119–152 crean microservicios con 26–30 Bibliotecas Netflix Hystrix 29 Bibliotecas de cintas de Netflix 29 Gateway de servicio Netflix Zuul 29 implementaciones de aprovisionamiento 30 Bota de primavera 28 Configuración de nube de primavera 28 Seguridad en la nube de primavera 30 Descubrimiento del servicio Spring Cloud 28 Detective de la nube de primavera 29–30 Patrones de resiliencia del cliente Spring Cloud Stream 29 con procesamiento de respaldo 133–135 implementación del patrón de mamparo 136–138

- configurar servidores de licencias para usar 127–128
- servidores de configuración
  - construyendo 70–
  - 77 controlando la configuración con 64–95
  - actualizando propiedades usando 88
  - usando con el sistema de archivos
  - 75–77 usando con Git 87–
    - 88 cableado en la fuente de datos usando
    - 83–86 configurando para usar personalizado *HystrixConcurrencyStrategy* 150–152
  - descubrimiento de
  - servicios 28 enrutamiento de servicios con 153–157, 159–191
    - construir un filtro de ruta dinámico 184–191
    - construir una correlación de recepción del post-filtro ID 182–184
    - puertas de enlace de servicios
  - 154–156 usando anotaciones para los servicios Netflix Zuul 157–158
- Spring Cloud Config 28**
  - configuración del servicio de licencias para usar 79–82
  - integración con el cliente Spring Boot 77–89 lectura directa de propiedades usando la anotación `@Value` 86–87
  - propiedades refrescantes usando el servidor de configuración de Spring
  - Cloud 88 usando el servidor de configuración de Spring Cloud Git 87–88
  - cableado en la fuente de datos usando el servidor de configuración Spring Cloud 83–
  - 86 configuración de clases de arranque 74–
  - 75 configuración de dependencias del servidor del servicio de licencias 79
- Seguridad en la nube de primavera 30
- Spring Cloud Sleuth**
  - agregando licencias 261–263
  - agregando a la organización 261–263
  - anatomía del seguimiento 262–263 ID de correlación y 260–263
  - dependencias 275
  - seguimiento distribuido con 259–287
  - implementando 265–266
  - agregación de registros y 263–274
    - agregando ID de correlación a HTTP respuesta con Zuul 272–274
    - configurar el conector syslog 267–268 crear una cuenta de Papertrail 267–268 implementar Papertrail 265–266 implementar Spring Cloud Sleuth 265–266 redirigir la salida de Docker a Papertrail 268–269 ID de seguimiento 270
- Arquitectura Spring Cloud Stream de carpeta 237–238 238**
- canal 238
- sumidero
- 238 fuente
- 238 arquitectura basada en eventos con 228–258
  - almacenamiento en caché distribuido
  - 249–258 desventajas de la arquitectura de mensajería
  - 235 usar mensajería para comunicar cambios de estado entre servicios 233–234 usar un enfoque de solicitud-respuesta sincrónico para comunicar cambios de estado 230–232
  - escribir consumidor de mensajes 238–249
  - escribir productor de mensajes 238–249
- Spring Data Redis**
  - define repositorios 251–253
  - dependencias 250
- Spring Security, sumándose a los servicios individuales 205**
  - dependencia de `spring-cloud-security` 196
  - dependencia de `spring-cloud-sleuth-zipkin` 275 biblioteca `spring-cloud-starter-eureka` 107 dependencia de `spring-cloud-starter-sleuth` 272 dependencia de `spring-security-jwt` 218 dependencia de `spring-security-oauth2` 196 primavera. propiedad `application.name` 80, 108 propiedad `spring.cloud.config.uri` 80 propiedad `spring.cloud.stream.bindings` 2416 propiedad `spring.cloud.stream.bindings.input.group` 247 propiedad `spring.datasource.password` 92 propiedad
  - `spring.profiles.active` 80 propiedad
  - `spring.stream.bindings.kafka` 243 propiedad
  - `spring.stream.bindings.output` 243 propiedad
  - `spring.zipkin.baseUrl` 275 anotación `@SpringBootApplication` 48 SQS (servicio de cola simple) 236 SSL (capa de sockets seguros) 39, 224 SSO (señal única) Activado) 212 cambios de estado comunicando entre servicios con
- mensajería 233–234
- durabilidad 234
- flexibilidad 234
- acoplamiento flojo 233–234
- escalabilidad 234
- comunicación con un enfoque sincrónico de solicitud–respuesta 230–232 fragilidad entre servicios 232 inflexibilidad al agregar
- nuevos consumidores a cambios en el servicio de organización 232
- estrecho acoplamiento entre servicios 231–232
- enrutamiento estático
- 155 URL estáticas, mapeo manual de rutas usando 165–167
- `@StreamListener` anotación 246, 257

Clase de canal suscribible 256  
 Subversión 311  
 atributo sudo 316  
 enfoque sincrónico de solicitud-respuesta 230–232  
     fragilidad entre servicios 232 inflexibilidad  
     al agregar nuevos consumidores a los cambios en la  
         organización servicio 232 estrecho  
         acoplamiento entre servicios 231–232  
 SynchronousQueue 138 syslog,  
 configuración del conector 267–268

**t**

Anotación @Table 84 valor  
 de nombre de etiqueta  
 321 código de control de fuente de etiquetado  
 320–321 protocolo tecnológicamente neutral 5  
 Contexto del hilo  
 ThoughtMechanix 337, Netflix Hystrix y 144–152  
     Estrategia de concurrencia de Hystrix 147–152  
     Hilo local 144–147  
 Aislamiento de hilo 144  
 ThreadLocal, Netflix Hystrix y 144–147  
 ThreadLocalAwareStrategy.java 148  
 ThreadLocalConfiguration 150 propiedad  
 threadPoolKey 143 atributo  
 threadPoolProperties 137–138, 140 Thrift 20

tiempo de espera, personalización en el disyuntor  
 132–133  
 encabezado tmx-correlation-id 173, 282  
 atributo token\_type 203 tokens,  
 actualización 343–344 método  
 tokenServices() 215 herramientas,  
 instalación en Travis CI 318–319 ID de seguimiento  
 262  
 trazabilidad 319  
 Clase de seguimiento 273,  
 284  
     niveles de configuración  
     de seguimiento 278 transacciones con Zipkin 278–280  
 TrackingFilter clase 173, 176, 222  
 transacciones  
     complejo, visualizando 281–282  
     rastreo con Zipkin 278–280  
 Travis CI 30  
     permitir la integración del servicio 312–325  
         crear microservicios 321–322 configuración  
         del tiempo de ejecución de la compilación central 315–  
         317 crear imágenes de Docker 321–322  
         ejecutar la compilación  
         320 invocar pruebas de plataforma 323–  
         325 instalaciones de herramientas previas a la  
             compilación 318–319 enviar imágenes a Docker Hub 322–323

iniciar servicios en Amazon ECS 323 etiquetar  
 el código de control fuente 320–321 implementar  
 el proceso de construcción/implementación  
 con 311–312  
 archivo travis.yml 313  
 Manifiesto de aplicación de doce factores 55

**Ud.**

UDDI (Descripción Universal, Descubrimiento e Integración)  
     gración) repositorio 96  
 pruebas unitarias 307  
 URL (localizador uniforme de recursos) 52  
 UserContext 145, 179–180  
 usercontext 149–150  
 UserContext.AUTH\_TOKEN 220  
 UserContextFilter 178–179, 220  
 UserContextFilter clase 178  
 UserContextHolder clase 146  
 UserContextIntceptor 181–182  
 UserContextIntceptor clase 178, 181, 219 método  
 userDetailsServiceBean() 201  
 usuarios  
     autenticado, protegiendo el servicio mediante 207–209  
     autenticando 202–205 de  
     EagleEye, configurando 200–202 método  
 useSpecialRoute() 187–188 paquete de  
 utilidades 178

**V**

@Anotación de valor 86–87  
 esquema de versiones 52  
 contenedores virtuales 16  
 imágenes de máquinas virtuales  
 15 visibilidad de mensajes 235  
 visualización de transacciones complejas 281–282  
 VPC (Nube Privada Virtual) 300

**W.**

WebSecurityConfigurerAdapter 201  
 WebSecurityConfigurerAdapter clase 201  
 WebSphere 66  
     fuente de datos de cableado 83–  
     86 método withClient() 199  
     método wrapCallable() 149 escritura  
     de clases Bootstrap 47–48

**X**

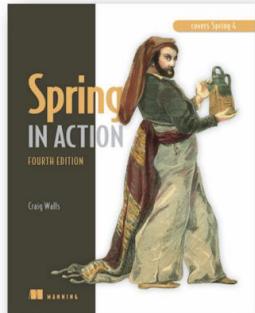
XML (lenguaje de marcado extensible) 20

**Z****Zipkin**

configura servicios para que apunten a 275–276  
seguimiento distribuido con 259, 274–287  
agrega intervalos personalizados 284–287  
captura de seguimientos de mensajes 282–284  
configura el servidor 276–277  
configura los servicios para que apunten a 275–276  
instala el servidor 276–277  
integra Spring Cloud Dependencias de detectives 275

establecer niveles de seguimiento 278  
rastrear transacciones 278–280  
visualizar transacciones complejas 281–282  
servidor  
configurando 276–277  
instalando 276–277  
rastreando transacciones con 278–280  
Guardián del zoológico 69  
Clase de filtro Zuul 174–175, 186  
ZuulRequestHeaders 176  
zuulservice 168

## TÍTULOS DE MANNING RELACIONADOS



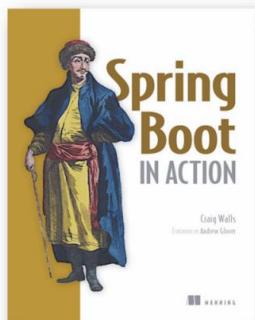
Primavera en acción, cuarta edición

Portadas Primavera

4 de Craig Walls

ISBN: 9781617291203 624

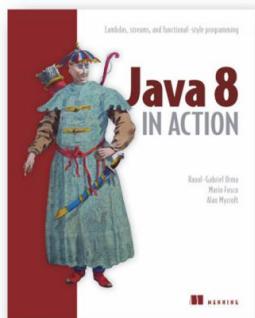
páginas, \$49,99  
noviembre de 2014



Arranque de primavera en  
acción por Craig Walls

ISBN: 9781617292545 264

páginas, \$44,99  
diciembre de 2015

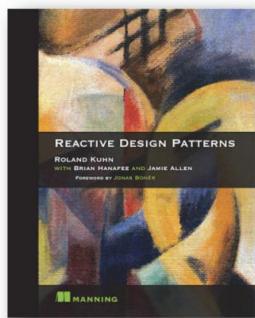


Java 8 en acción

Lambdas, streams y programación de estilo funcional de Raoul-Gabriel Urma, Mario Fusco, y Alan Mycroft

ISBN: 9781617291999 424

páginas, 49,99 dólares  
agosto de 2014



Patrones de diseño reactivo por

Roland Kuhn con Brian Hanafee y Jamie Allen

ISBN: 9781617291807 392

páginas, \$49,99  
febrero de 2017

Para obtener información sobre pedidos, visite [www.manning.com](http://www.manning.com)

# Spring Microservices EN ACCIÓN

Juan Carnell



VER INSERTO

**Los** microservicios dividen su código en pequeños y distribuidos.

servicios adaptados el diseño de los sistemas que requieren una cuidadosa previsión y diseño. Afortunadamente, Spring Boot y Spring Cloud simplifican sus aplicaciones de microservicios, al igual que Spring Framework simplifica el desarrollo empresarial de Java.

Spring Boot elimina el código repetitivo involucrado en la escritura de un servicio basado en REST. Spring Cloud proporciona un conjunto de herramientas para el descubrimiento, enrutamiento e implementación de microservicios para la empresa y la nube.

**Spring Microservices in Action** le enseña cómo crear aplicaciones basadas en microservicios utilizando Java y la plataforma Spring. Aprenderá a diseñar microservicios a medida que crea e implementa su primera aplicación Spring Cloud. A lo largo del libro, ejemplos de la vida real cuidadosamente seleccionados exponen patrones basados en microservicios para configurar, enrutar, escalar e implementar sus servicios. Verá cómo las herramientas intuitivas de Spring pueden ayudar a aumentar y refactorizar aplicaciones existentes con microservicios.

## Qué hay dentro

- Principios básicos de diseño de microservicios
- Administrar la configuración con Spring Cloud Config
- Resiliencia del lado del cliente con Spring, Hystrix y Ribbon
- Enrutamiento inteligente usando Netflix Zuul
- Implementación de aplicaciones Spring Cloud

Este libro está escrito para desarrolladores con experiencia en Java y Spring.

**John Carnell** es un ingeniero senior de nube con veinte años de experiencia en Java.

Para descargar su libro electrónico gratuito en formatos PDF, ePUB y Kindle, los propietarios de este libro deben visitar

[www.manning.com/books/spring-microservices-in-action](http://www.manning.com/books/spring-microservices-in-action)

"La primavera se está convirtiendo rápidamente en el marco para los microservicios: este libro te muestra por qué y cómo.

—John Guthrie, Dell/EMC

"Una biblia completa del mundo real para cualquier proyecto de microservicios en Spring."

—Mirko Bernardoni, Ixxus

"Completo y práctico... con todas las capacidades especiales de Spring tirado en."

—Vipul Gupta, SAVIA

"Aprenda a dominar el diseño de sistemas complejos y distribuidos."

Muy recomendable."

—Ashwin Raj, Inoceptos

ISBN-13: 978-1-61729-398-6  
ISBN-10: 1-61729-398-9

5 4 9 9 9



9 7 8 1 6 1 7 2 9 3 9 8 6



MANTENIMIENTO

\$49.99 / Can \$65.99 [INCLUYE EL LIBRO electrónico]