Microservicios

Un enfoque integrado

David Roldán Martínez
Pedro J. Valderas Aranda
Victoria Torres Bosch

Desde **www.ra-ma.es** podrá descargar material adicional.

Ra-Ma®

Microservicios

Un enfoque integrado

Microservicios

Un enfoque integrado

David Roldán Martínez

Pedro J. Valderas Aranda

Victoria Torres Bosch

Ra-Ma®

La ley prohíbe
fotocopiar este libro

Editado por:
RA-MA Editorial
Calle Jarama, 3A, Polígono Industrial Igarsa
28860 PARACUELLOS DE JARAMA, Madrid
Teléfono: 91 658 42 80
Fax: 91 662 81 39
Correo electrónico: *editorial@ra-ma.com*
Internet: *www.ra-ma.es* y *www.ra-ma.com*
ISBN: 978-84-9964-765-4
Depósito legal: M-28963-2018
Maquetación: Antonio García Tomé
Diseño de portada: Antonio García Tomé
Filmación e impresión: Safekat
Impreso en España en octubre de 2018

A Ana, por dejarme sin palabras
cada vez que me mira
D.R.M.

A mis padres,
por todo su apoyo y
cariño incondicional
P.V.A.

A mi sobrina Claudia,
el mejor remedio contra el estrés
que jamás se haya inventado
V.T.B.

ÍNDICE

AUTORES

David Roldán Martínez es doctor ingeniero de Telecomunicación y máster en Redes Corporativas e Integración de Sistemas por la Universidad Politécnica de Valencia (UPV). Ha trabajado en empresas de consultoría y desarrollo de proyectos y productos relacionados con las tecnologías de la información y hoy en día es analista de aplicaciones del ASIC de la UPV.

Ha contribuido activamente en comunidades open source como Sakai (software de e-learning en donde desempeñó diversos cargos de responsabilidad mundial y que le otorgó en 2011 el *Sakai Fellow Award* por su participación activa en el desarrollo y puesta en marcha de Sakai alrededor del mundo) o Jalview (herramienta para la gestión y anotación de secuencias genómicas más utilizadas a nivel mundial, en donde es desarrollador y responsable de i18n).

Además, ha sido profesor del Departamento de Comunicaciones de la Universidad Politécnica de Valencia y ha impartido formación de posgrado en distintas universidades e instituciones. Es miembro de ACTA (Asociación de Autores Científicos y Técnicos) y, siempre preocupado por la divulgación científico-tecnológica, dispone en su haber de numerosos libros y artículos relacionados con diversos aspectos de las TIC.

Pedro J. Valderas Aranda es profesor en el Departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia (UPV), donde actualmente imparte clases sobre diseño web e integración de aplicaciones. Doctor en Informática por la misma universidad, es miembro del Centro de Investigación en Métodos de Producción de Software (PROS), donde compagina sus labores docentes con su actividad investigadora. Tiene en su haber varios libros sobre bases de datos y desarrollo web, y numerosas publicaciones de I+D en el ámbito nacional e internacional. Además, es miembro de diversos comités científicos de conferencias

internacionales. Sus temas de interés incluyen el desarrollo dirigido por modelos, procesos de negocio, ingeniería web e internet de las cosas.

Victoria Torres Bosch es Doctora Ingeniera en Informática por la Universidad Politécnica de Valencia (UPV) en 2008. Actualmente compagina su labor como investigadora en el Centro de Investigación en Métodos de Producción de Software (PROS) y como docente en la Escuela Superior Politécnica de Gandía (EPSG) en la titulación de Comunicación Audiovisual, ambos centros pertenecientes a la UPV. Participa activamente en la comunidad científica en áreas como la gestión de procesos de negocio (BPM), la variabilidad en BPM, Internet de las Cosas (IoT), la ingeniería Web y el desarrollo Dirigido por Modelos. Las actividades desarrolladas en dichas áreas incluyen la participación como miembro de comité científico y organizador en diferentes eventos y la publicación en diferentes revistas (*Information and Software Technology*, *Information Systems* y *Software and System Modeling*) y conferencias (*International Conference on Business Process Management*, *International Conference on Advanced Information Systems Engineering*, *International Conference on Conceptual Modeling*) de dichas áreas.

AGRADECIMIENTOS

Paradójicamente, siempre éste es, con mucho, el apartado que más esfuerzo supone de toda una obra. Vaya por adelantado que no queremos dejarnos a nadie y pedimos disculpas por adelantado si esto sucede.

En primer lugar, a mí me gustaría agradecer a Pedro y a Vicky, coautores de este libro, el entusiasmo y la dedicación y la buena disposición a abordar proyectos nuevos. Y, quiero añadir un puntito especial para Vicky: siendo éste tu debut como escritora, espero que lo hayas aprendido y disfrutado. La primera vez, como tantas otras cosas en la vida, es la más intensa y la que con más cariño recuerdas. Espero que Pedro y yo hayamos estado a la altura.

Queremos dar las gracias a Ana, Belén y Fran por su apoyo incondicional y su paciencia con el robo de momentos que escribir un libro supone. Sin ellos, es seguro que tú, querido lector, no estarías leyendo estas letras.

Además, tampoco nos olvidamos de todos aquellos que, de una manera u otra han contribuido, a que hayamos llegado hasta aquí y la Universitat Politècnica de Valencia, donde los prácticamente nos hemos desarrollado profesionalmente.

Querido lector, ya no te entretenemos más. Te dejamos para que te deleites con el viaje que aquí vas a encontrar. Muchas gracias a ti también por confiar en nosotros. Sin ti, esto tampoco habría sido posible. Gracias.

1

INTRODUCCIÓN

1.1 A QUIÉN VA DESTINADO ESTE LIBRO

Este libro está orientado a un público con distintos niveles de experiencia en microservicios y desarrollo en la nube, aunque la audiencia principal son aquellos lectores con cierto bagaje en la Ingeniería del Software y la Informática que quieran iniciarse en el desarrollo de aplicaciones distribuidas bajo un paradigma orientado a microservicios, haciendo especial hincapié en el desarrollo mediante tecnología Java.

1.2 ESTRUCTURA DE ESTE LIBRO

Los nueve capítulos del libro están estructurados de manera similar: una introducción teórica enriquecida con multitud de cuestiones prácticas explicadas con una aplicación de ejemplo, especialmente diseñada con propósitos didácticos para esta obra.

1.3 INFORMACIÓN ADICIONAL Y GARANTÍA

- RA-MA EDITORIAL garantiza que estos contenidos han sido sometidos a un riguroso control de calidad.
- Los archivos están libres de virus, para comprobarlo se han utilizado las últimas versiones de los antivirus líderes en el mercado.

2

MICROSERVICIOS. CONCEPTOS BÁSICOS

En los últimos 10 años y gracias, entre otros factores, al impresionante impacto del Internet en prácticamente todos los ámbitos de nuestras vidas de la mano de las aplicaciones móviles, Smart TV y el Internet de las Cosas, los paradigmas de programación han sufrido, más que una evolución, una revolución que nos ha llevado a las arquitecturas basadas en microservicios.

En la actualidad, todo lo relacionado con los microservicios está despertando gran cantidad de interés y son muchos los artículos, blogs o conferencias que tratan, de una manera u otra, alguno de los aspectos de este tipo de soluciones tecnológicas.

El ciclo de sobreexpectación de Gartner (ver Figura 1) representa la madurez, adopción y aplicación de una tecnología concreta en el mercado. En la primera etapa, la de lanzamiento, se presenta el producto y el interés comienza a crecer hasta alcanzar su máximo en un pico de expectativas sobredimensionadas y poco realistas. A partir de este pico, las tecnologías dejan de estar moda y el interés decrece hasta alcanzar el abismo de desilusión. A continuación, de nuevo aumenta el interés, aunque a una tasa mucho menor, fruto de que algunas empresas todavía confían en esa tecnología y llega un momento en que dicho interés se estabiliza en la meseta de productividad.

Figura 2.1. Ciclo de sobreexpectación

Dicho esto, podríamos decir que los microservicios se encuentran entre las dos primeras fases. Sin embargo, existe una corriente escéptica que señala que es un nuevo nombre de las arquitecturas SOA (*Service Oriented Architecture*). Sin embargo, a pesar de todo, las arquitecturas de microservicios presentan algunas ventajas (y también algunos inconvenientes) que analizaremos a lo largo de este libro.

Con el fin de comprender bien los principios básicos de las aplicaciones basadas en microservicios, conviene repasar de dónde provienen, ya que surgieron como respuesta a los problemas que planteaban las aplicaciones monolíticas, sean distribuidas o no.

2.1 ARQUITECTURAS MONOLÍTICAS

Supongamos que vamos a construir una aplicación de comercio electrónico cuyo objetivo es vender consumibles de electrónica a través de Internet y del móvil. Siguiendo el tradicional paradigma monolítico, en el desarrollo de esta aplicación usaríamos una aproximación de n-capas como la de la Figura 2.2:

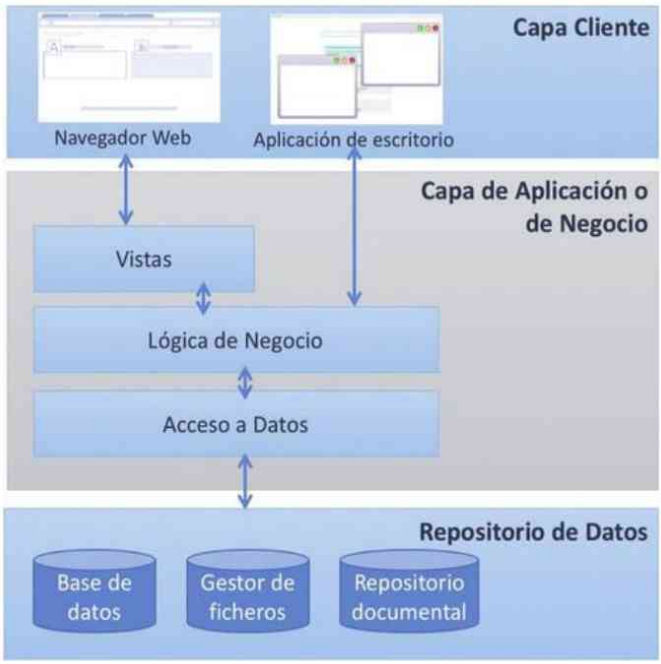

Figura 2.2. Arquitectura monolítica de n-capas

En la capa de cliente, encontramos clientes web o clientes de aplicación:

- Un cliente web está formado por un conjunto de páginas de contenido estático (HTML, XML u otro lenguaje de marcas) o bien dinámico que se generan al vuelo en el propio cliente o en el servidor. En cualquiera de los dos casos, son consumidas por un navegador web. Es común referirse a ellas como cliente ligero (*thin client*).
- Los clientes de aplicación (de escritorio) proporcionan a los usuarios una interfaz mucho más rica que la de los lenguajes de marcas y acceden directamente a la capa de Aplicación o Negocio.

La Capa de Aplicación o de Negocio se implementa en un servidor de aplicaciones. Suele dividirse en diferentes subcapas que, desde un punto de vista general, mantienen: (1) la Lógica de Negocio, que contiene las funcionalidades del negocio según las reglas lógicas adecuadas a la aplicación; (2) el Acceso a Datos, que se encarga de la recuperación y actualización de datos en un repositorio en cuestión; y (3) en el caso de aplicaciones Web, la subcapa de generación de Vistas, que envía al navegador cliente el código HTML necesario.

Bajo la Capa de Aplicación, se encuentran los Repositorios de Datos, tales como bases de datos, gestores de ficheros, repositorios documentales, etc.

Este tipo de arquitecturas, a pesar de estar divididas en capas, se denominan monolíticas porque poseen un único ejecutable lógico. Si volvemos a la figura anterior, veremos que la Capa de Aplicación está contenida en un único servidor (cuando tratemos el tema de escalado, entraremos más en detalle), es decir, que toda la aplicación se ejecuta dentro de un mismo contexto.

Entre las principales ventajas de esta aproximación, tenemos:

- Facilidad de desarrollo: al ser el tipo más común de aplicaciones, existe una cantidad enorme de entornos de desarrollo integrados (IDE, *Integrated Development Environment*) preparados para gestionar las distintas capas de una misma aplicación desde un entorno cómodo y completo.
- Simplicidad de las pruebas: puesto que la aplicación es, desde el punto de vista lógico, única, también lo es la estrategia de pruebas. Mediante pruebas sistemáticas de sus distintas capas es posible conseguir una aplicación fiable.
- Sencillez de despliegue: a menudo la aplicación consta de único fichero (por ejemplo, un *war* en el caso de aplicaciones Java EE) que se copia en un directorio específico del servidor de aplicaciones.
- Viabilidad del escalado: este tipo de arquitectura suelen ser fácilmente escalables mediante la duplicación de servidores tras un balanceador de carga. Sobre este punto, volveremos más adelante en este capítulo.

Generalmente, este tipo de aplicaciones están hechas a medida, lo que las convierte en eficientes y rápidas. Sin embargo, se trata de estructuras rígidas que carecen de flexibilidad y a medida que la complejidad de la lógica de negocio aumenta, el número de funcionalidades que la aplicación debe proporcionar, no solamente de cara a los usuarios finales sino también de comunicación entre los distintos componentes de soporte de la propia aplicación, comienzan a ponerse de manifiesto algunos problemas serios.

En primer lugar, existe un fuerte acoplamiento arquitectónico entre componentes funcionales distintos. En efecto, al ejecutarse la aplicación como un único sistema lógica (aunque esté divida en n-capas), no es posible introducir mejoras en un componente sin redesplegar la aplicación completa, lo que también afecta a las pruebas funcionales (debería verificarse el impacto en cada uno de los módulos o componentes) y que, por tanto, los ciclos de aplicación sean más largos y menos frecuentes, dado que los costes son más elevados.

En segundo lugar, hay que escalar la aplicación completa, independientemente de si las necesidades de ampliación las tenemos en un único módulo o en toda la aplicación.

Finalmente, la residencia o tolerancia a fallos es menor, ya que un fallo en un componente puede dejar sin servicio al resto, afectando a la aplicación completa.

Todos estos factores justificaron la búsqueda de un nuevo paradigma que los resolviera. Era necesario un paso más en la evolución hacia las aplicaciones distribuidas, representadas hasta entonces por SOA (*Service Oriented Architecture*) y los buses de integración, pero con un enfoque más flexible. Así nacieron los microservicios.

La arquitectura basada en microservicios tiene su fundamento en la división en módulos independientes denominados microservicios, que se construyen alrededor de funcionalidades de negocio muy concretas (autenticación, directorio, correo, CRUD de ítems, etc.) y que se comunican entre sí mediante mecanismos de interacción relativamente sencillos, como puedan ser llamadas *RESTfull* a un API o algún tipo de RPC (*Remote Procedure Communications*).

Cada microservicio se despliega de manera independiente al resto, lo que posibilidad que cada uno evolucione por separado, e incluso que estén construidos en tecnologías diferentes, siempre y cuando expongan sus funcionalidades a través de un API bien documentada y conocida.

2.2 ESCALADO DE APLICACIONES

Con el fin de representar de manera sencilla las distintas alternativas de escalado de una aplicación, Martin Abbott y Michael Fisher idearon el cubo de escalabilidad en su libro *"The Art of Scalability"* (ver Figura 2.2).

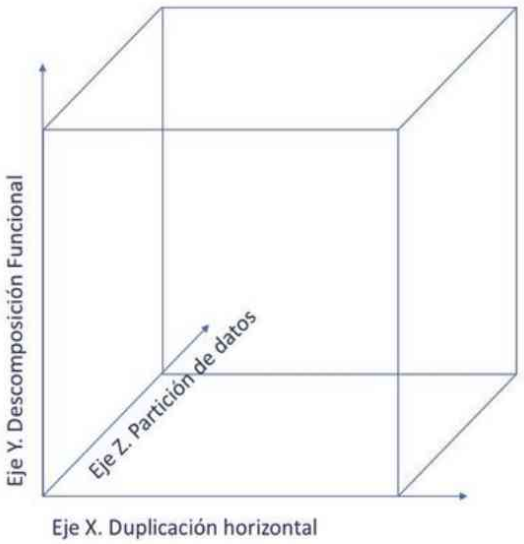

Figura 2.3. Cubo de escalabilidad

El cubo de escalabilidad considera tres dimensiones: la duplicación (eje X), la descomposición funcional (eje Y) y la partición de datos (eje Z). Variando a través de los ejes conseguimos distintas estrategias de escalado.

La opción más sencilla es el escalado horizontal o duplicación (desplazamiento a través del eje X). Tras una balanceador de carga, se ejecutan varias copias de la aplicación, de manera que dicha se reparte entre ellas. No obstante, este método presenta los mismos problemas que hemos comentado anteriormente, sólo que desplazamos el problema porque se es capaz de soportar un mayor nivel de carga.

Una estrategia similar, pero en otro dominio, es la partición de datos o balanceo en función de la información. En cada servidor, se ejecuta la aplicación completa pero el balanceo particiona los datos en rangos, de manera que a cada servidor le corresponde manejar un determinado conjunto de ellos (por ejemplo, por zonas geográficas o por tipo de cliente).

Ventajas	Inconvenientes
Cada servidor maneja menos cantidad de información debido al particionado	Incremento de la complejidad, puesto hay que implementar la estrategia de particionado
Reduce el consumo de memoria	No resuelve los problemas del carácter monolítico de la aplicación
Aumenta el rendimiento porque las transacciones son más rápidas	
Las caídas de un servidor afectan a menos usuarios	

Tabla 2.1. Ventajas e inconvenientes del escalado por partición de datos

Por último, el escalado funcional, base del concepto de microservicio, consiste en la descomposición de la aplicación en unidades funcionales independientes (ver Figura 2.4), de manera que las distintas unidades pueden evolucionar sin necesidad de afectar al resto. Además, el testeo, el despliegue y otras tareas del ciclo de la aplicación también se llevan a cabo de manera separada. Sin embargo, la división en microservicios no está exenta de dificultades, como veremos más adelante en este capítulo.

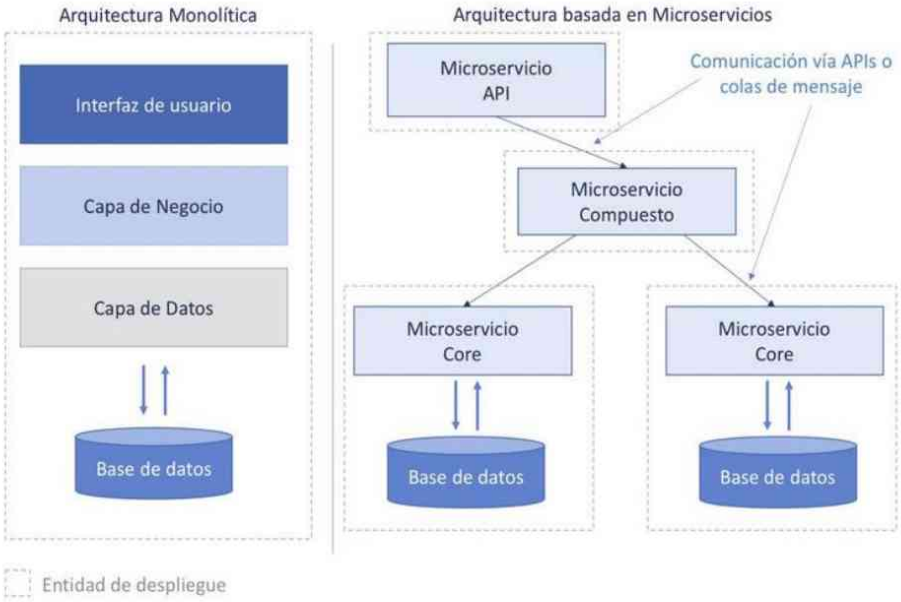

Figura 2.4. Arquitectura monolítica vs Microservicios

2.3 MICROSERVICIOS

Acabamos de afirmar que la solución más flexible de escalado es la descomposición de la aplicación en microservicios. Surge, pues, la pregunta de qué criterio seguir a la hora de dividir o agrupar funcionalidades en microcroservicios independientes. En general, existen dos enfoques de abordar esta problemática, aunque si bien es cierto que es posible una estrategia mixta: la descomposición basada en verbos y la basada en sustantivos. En cualquiera de las dos opciones, el proceso comienza con un diagrama de casos de uso.

El diagrama de casos de uso es una herramienta de modelado UML *(Unified Modeling Language*) que muestra la interacción entre los distintos roles o tipos de usuario y la aplicación que se está diseñando[1]. Por este motivo, resulta ideal como punto de partida para la división funcional de cualquier aplicación. La Figura 2.5 muestra el diagrama de casos de uso de la aplicación de ejemplo que emplearemos para aprender a dominar al arte de los microservicios y que se desarrollará a lo largo del libro.

1 En la Bibliografía pueden encontrarse referencias al respecto. Recomendamos al lector repasar los conceptos de modelado.

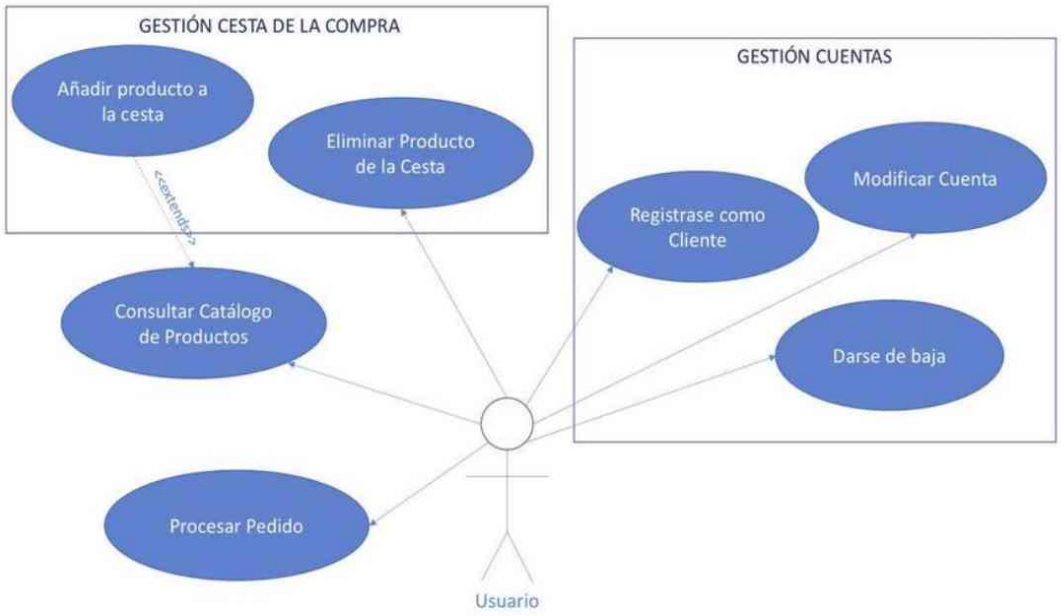

Figura 2.5. Diagrama de casos de uso

Para definir los microservicios que darán soporte a esta aplicación podemos utilizar una descomposición basada en sustantivos, mediante la cual, agrupamos en cada microservicio las funcionalidades relacionadas con una entidad en particular (sustantivo), como por ejemplo, Catálogo o Pedido. En cualquier caso, debe respetarse el Principio de Responsabilidad Individual (SRP, *Single Responsibility Prinmejorciple*) de Robert C. Martin, que establece que "debe reunirse aquello que cambia por la misma razón y separar lo que cambia por diferentes razones", de manera que los límites de responsabilidad de cada microservicio estén claramente definidos. Suele emplearse como analogía el diseño de comandos Unix, ya que existe una gran cantidad de utilidades especializadas en una tarea y que puede combinarse entre sí mediante un *script*. Volviendo a la aplicación de ejemplo, la descomposición en microservicios con la que trabajaremos será la presentada en la Figura 2.6. Queda por resolver cómo se comunicarán los microservicios entre sí, pero eso lo estudiaremos en el epígrafe siguiente.

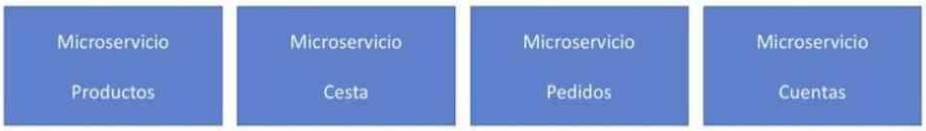

Figura 2.6. Descomposición en microservicios basada en sustantivos

Hemos partido de un diagrama de casos de uso que describe los requisitos de una aplicación relativamente fácil de desarrollar mediante una arquitectura monolítica y la hemos descompuesto en múltiples microservicios que se comunican (todavía no sabemos cómo) en un entorno distribuido, aumentando así la complejidad de la aplicación. Además, no existen hoy en día IDE potentes para el desarrollo de aplicaciones distribuidas. Surge, por tanto, la pregunta de dónde están las ventajas de los microservicios. Son las siguientes:

- El código fuente de cada microservicio es mucho menor que el mismo código en "modo monolítico", lo que mejora la comprensión y la gestión del mismo por parte del desarrollador.
- Cada microservicio se despliega de manera independiente al resto, por lo resulta más fácil desplegar nuevas versiones con la relativa frecuencia y dota a la cadena de despliegue de mayor autonomía y velocidad.
- Cada equipo de desarrollo es responsable todo el ciclo completo de un microservicio: requisitos, codificación, pruebas y puesta en producción.
- Aumenta la residencia de la aplicación conjunta, ya que los errores en un microservicio tiene una afectación mínima en el funcionamiento del resto en comparación con lo que ocurriría en una aplicación monolítica en la que, casi con toda seguridad, ésta dejaría de funcionar.
- Cada microservicio puede estar desarrollado en una tecnología diferente, con una persistencia distinta, etc.

Sin embargo, el cambio de visión que suponen las arquitectura orientadas a microservicios no afecta únicamente a la Ingeniería del Software como proceso de producción de aplicaciones, sino que también tienen una implicación organizacional. En 1967 Melvin Conway estableció la ley que hoy lleva su nombre, en la que decía que cualquier organización dedicada al diseño de software, acaba produciendo software con una estructura que tiende a ser una copia de la estructura de dicha organización. Los microservicios son servicios independientes cuya responsabilidad recae sobre equipos de desarrollos independientes pero que deben comunicarse entre sí, por lo que parece sensato afirmar que la estructura de la organización debe adaptarse a esta concepción del proceso de gestión de software. En este sentido, es importante no olvidar los canales de comunicación informales ni la distribución geográfica de los distintos equipos de desarrollo.

En resumen, las características básicas de una arquitectura de microservicios son las siguientes:

- Orientación hacia las capacidades de negocio: los microservicios flexibilizan el desarrollo de aplicaciones al proporcionar un enfoque modular.
- Independencia de los microservicios entre sí: cada microservicio contiene su propia lógica de negocio y se despliega de manera separada al resto, permitiendo así realizar actualizaciones en algunos de microservicios sin necesidad de un corte en el servicio global.
- Gestión descentralizada de datos: cada microservicio puede tener su propia base de datos (incluso de distinta tecnología).
- Tolerancia a fallos: los microservicios está débilmente acoplados por lo que los fallos no se propagan en la cadena de servicios, aumentando en un sistema global más estable y robusto.

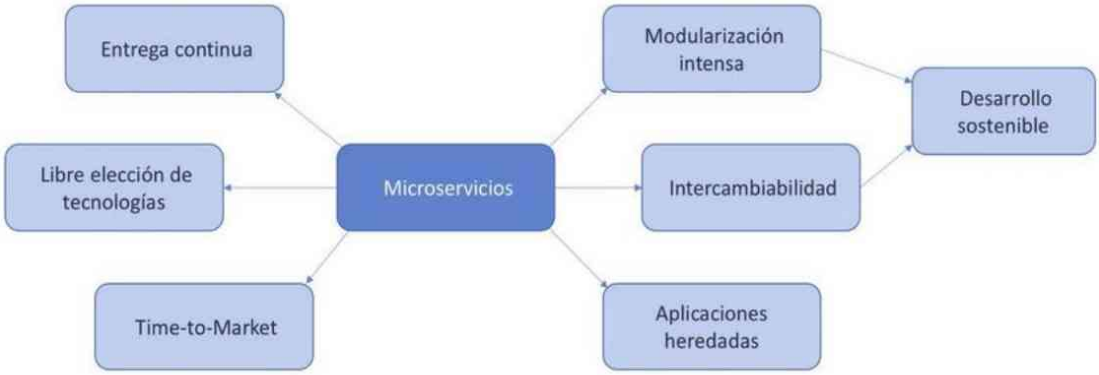

Figura 2.7. Beneficios de la arquitectura de microservicios

2.4 COMUNICACIÓN ENTRE MICROSERVICIOS

En un entorno distribuido de estas características será de vital importancia encontrar mecanismos de comunicación sencillos y fiables entre las distintas partes de la misma. En la Figura 2.8 se muestra los ámbitos de comunicación que necesitamos resolver:

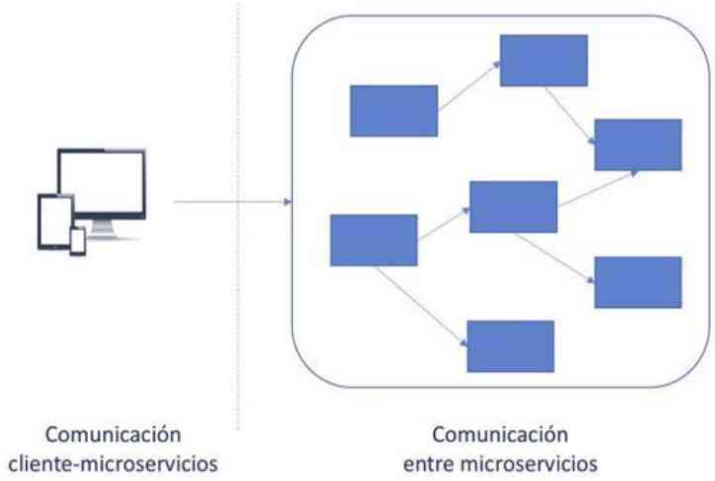

Figura 2.8. Ámbitos de comunicación en aplicación basada en microservicios

Las arquitecturas de microservicios se caracterizan por lo que se llama *dumb pipes* (tuberías tontas), es decir, que es en los puntos finales donde reside toda la inteligencia y no debe invertirse esfuerzo en complicados mecanismos de comunicación que no aportan ningún valor al sistema y sí que supone un incremento considerable de la complejidad. Por ello, es preferible optar por mecanismos de comunicación ligeros y simples (*dumb pipes*).

2.4.1 Comunicación cliente-microservicios

A la hora de construir cualquier aplicación, uno de los puntos críticos es decidir cómo accederán los clientes (usuarios) a la aplicación. Bajo un paradigma monolítico existe un único conjunto de puntos finales replicados cuya carga, quizá, se reparta a través de un balanceador. En un paradigma de microservicios, sin embargo, cada microservicio expone lo que se denomina puntos finales de grano fino y es ahí donde entra en juego el API Gateway y el impacto que tiene en la comunicación entre los clientes y la aplicación.

Para ilustrar el concepto de API Gateway, pensemos en el cliente móvil nativo de la aplicación de Amazon para Android (ver Figura 1) y detengámonos en la página de detalles de un producto cualquiera.

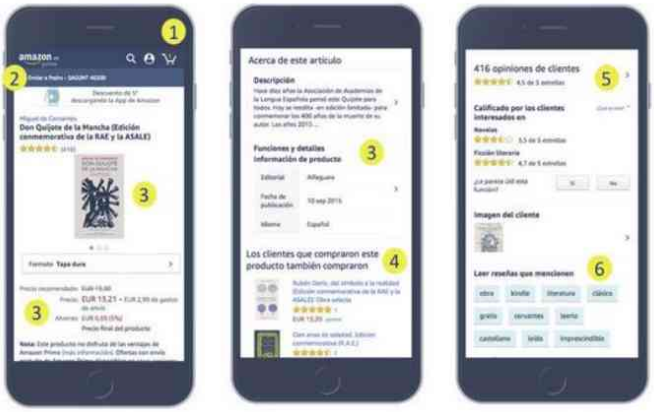

Figura 2.9. Aplicación Web Móvil de Amazon

Esta página contiene gran cantidad de información ya que ofrece al usuario no sólo información básica del producto sino también:

1. Número de elementos en la cesta de la compra
2. Datos sobre el usuario conectado
3. Información básica del producto
4. Recomendaciones
5. Opiniones de otros clientes
6. Palabras clave incluidas en reseñas

Si la aplicación fuera monolítica, el cliente móvil recuperaría todos estos datos con una única llamada REST a la aplicación que podría tener la forma:

GET api.empresa.es/detallesProducto/idProducto

Un balanceador de carga encaminaría la petición a alguna de las instancias (idénticas) de la aplicación, que consultaría varias tablas de la base de datos y devolvería la respuesta al cliente. Por el contrario, en una arquitectura de microservicios, los datos mostrados en la página de detalles de producto son responsabilidad de varios microservicios. Podríamos pensar en el siguiente reparto (ver Figura 2):

- Servicio de cesta de la compra: número de elementos en la cesta.
- Servicio de cuentas: datos sobre el usuario conectado
- Servicio de catálogo: información básica del producto.
- Servicio de recomendaciones: compras alternativas.

- Servicio de revisiones: opiniones de otros clientes y palabras clave en reseñas.

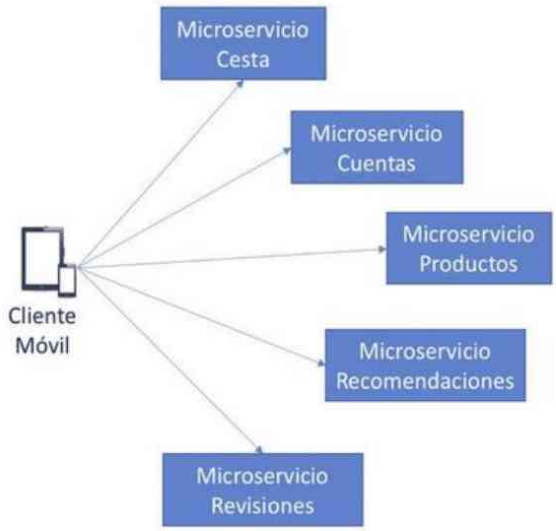

Figura 2.10. Mapeo de necesidades del cliente en microservicios

Necesitamos, por tanto, decidir cómo el cliente móvil accede a estos servicios.

2.4.1.1 COMUNICACIÓN DIRECTA ENTRE EL CLIENTE Y EL MICROSERVICIO

En una arquitectura monolítica el punto de entrada a la aplicación está bien definido. Los clientes se conectan a una página web y lanzan peticiones HTTP a la aplicación, que les contestará según la lógica de dicha aplicación y los datos del *backend*. Sin embargo, hemos distribuido las funcionalidades en microservicios independientes y, con ello, la lógica y el acceso a los datos y, por tanto, dispersado el punto de entrada a la aplicación, dado que ahora, a priori, el cliente debería conectarse con un microservicio u otro en función del objetivo de la transacción que desee ejecutar.

En teoría, un cliente puede lanzar peticiones directas a cualquiera de los microservicios, ya que cada uno de ellos expone un punto final público (ver Figura 2.11):

https://servicename.api.company.name/

Esta URL se mapearía en el balanceador de cada microservicio, encargado de distribuir las peticiones entre las diferentes instancias del microservicio. Para

recuperar la información de la página de detalles del producto, el cliente haría una petición a cada uno de los microservicios listados arriba. Sin embargo, esta alternativa de comunicación directa entre el cliente y los microservicios tiene algunos compromisos y limitaciones.

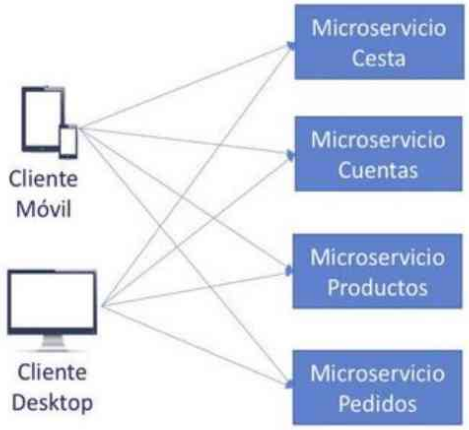

Figura 2.11. Comunicación directa con microservicios

El principal problema de esta aproximación es la variedad de clientes, ya que todos los dispositivos no tienen las mismas características en cuanto a capacidad de proceso y disponibilidad de ancho de banda. En general, una aplicación web puede exigir un importante trasiego de información que, por ejemplo, en un *smartphone* conectado a una red 4G supondría una mala experiencia de usuario.

Otro problema es que las llamadas directas desde el cliente podrían utilizar protocolos no orientados a la web como T. hrift RPC o AMQP para mensajería en el entorno interno de la aplicación y HTTP o WebSocket fuera del cortafuegos.

Finalmente, esta aproximación dificulta la refactorización de los microservicios. Si quisiéramos unir dos microservicios en uno solo, por ejemplo, habría que cambiar el código de todos clientes.

2.4.1.2 COMUNICACIÓN A TRAVÉS DE UN API GATEWAY

Para resolver éste y otros problemas relacionados se prefiere un esquema en el que un intermediario central se encargue de adaptar las llamadas a los microservicios (y sus respuestas) a distintos tipos de clientes, proporcionando API de la granularidad adecuada. Este intermediario es lo que se conoce como *API Gateway* (ver Figura 2.11).

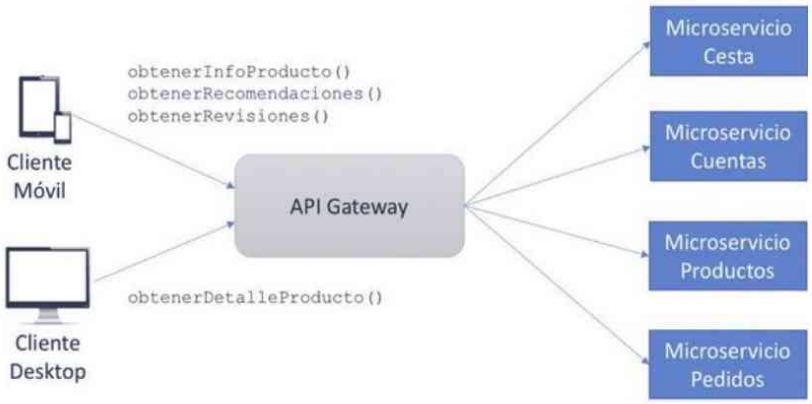

Figura 2.12. Esquema con API Gateway

El *API Gateway*, por otra parte, desacopla los clientes de los microservicios, de manera que éstos pueden evolucionar con un impacto mínimo en aquellos, ya que su uso es transparente.

La mayor ventaja de un esquema de comunicación basado en API Gateway es que encapsula la estructura interna de la aplicación. En lugar de invocar a servicios específicos, los clientes simplemente se comunican con la pasarela, que proporciona a cada tipo de cliente un API específica, reduciendo así la latencia entre la aplicación y el cliente y, además, simplificando el código del cliente.

Sin embargo, el API Gateway, por su criticidad, es un componente de alta disponibilidad más qué debería ser desarrollado, desplegado y gestionado. También existe el peligro de que el desarrollo del API Gateway se convierta en un cuello de botella y de que las actualizaciones sean pesadas. No obstante, a pesar de estos inconvenientes, se prefiere un esquema de este tipo en la comunicación entre los clientes y los microservicios.

2.4.2 Descubrimiento de microservicios

Otro punto importante en las arquitecturas basadas en microservicios es el registro de servicios. En efecto, si hemos dicho que los microservicios se ejecutan en un entorno distribuido en el que pueden existir múltiples instancias de un mismo microservicio que cambien de ubicación de manera dinámica, necesitamos algún mecanismo de registro o descubrimiento de servicios, de manera que los microservicios o el *API Gateway* sepan dónde enviar las peticiones de servicio que exija la aplicación.

La primer alternativa para el descubrimiento de servicios (ver Figura 2.13) es el basado en servidor (*server side*), también conocido como descubrimiento por

balanceo de carga. Existe un entidad central (balanceador) que consulta un servidor de registro (incluido o no en el propio balanceador) y que es el encargado de enviar a los clientes una referencia a una instancia del microservicio solicitado.

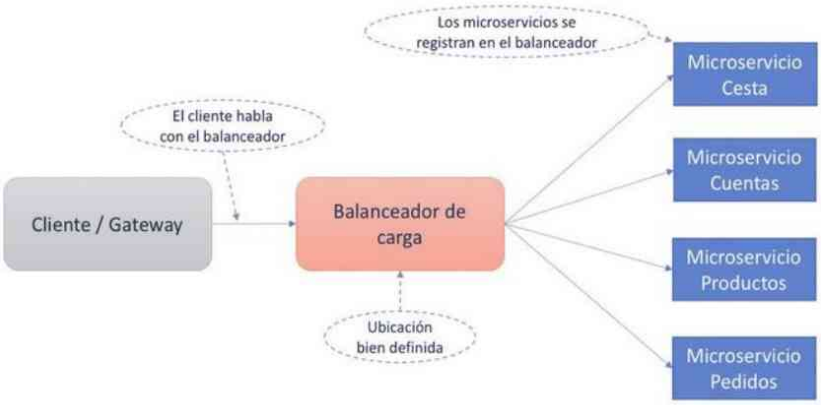

Figura 2.13. Descubrimiento de Server Side

Un ejemplo de descubrimiento *Server Side* es el ELB (*Elastic Load Balancer*) de Amazon AWS, que distribuye el tráfico de aplicación entrante a las múltiples instancias de EC2 en varias zonas de disponibilidad (ver Figura 2.14), ofreciendo, por otra parte, una mejora de la tolerancia a fallos. El balanceador de carga sirve como único punto de contacto para los clientes, siendo posible añadir o eliminar instancias del balanceador en función de las necesidades y sin suponer una interrupción del flujo total de peticiones a la aplicación. Además, el balanceador monitoriza la salud de las instancias de microservicios registradas con el fin de detectar destinos no disponibles (*unhealthy targets*), en cuyo caso, para redirigir el tráfico a dichos destinos hasta que verifique que está disponible de nuevo.

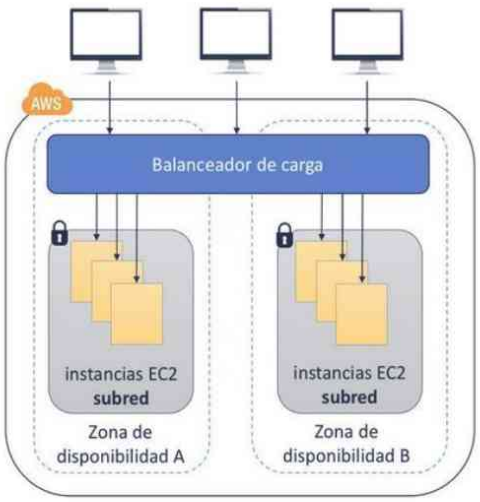

Figura 2.14. Zonas de disponibilidad con ELB

Es posible configurar el balanceador de carga para aceptar tráfico entrante especificando uno o más *listeners*, esto es, procesos que comprueban las peticiones de conexión, con un cierto protocolo y en un puerto determinado, de manera que el propio balanceador actúa también como servidor de registro de microservicios.

Otra alternativa es el descubrimiento en el cliente (*Client Side*). En este caso, el cliente se conecta directamente a un servidor de registro que es el que conoce la ubicación de todas las instancia de microservicios disponibles, en lugar de hacer a través de un balanceador. Una vez que el servidor de registro contesta al cliente, éste está en condiciones de comunicarse directamente con los microservicios que necesite.

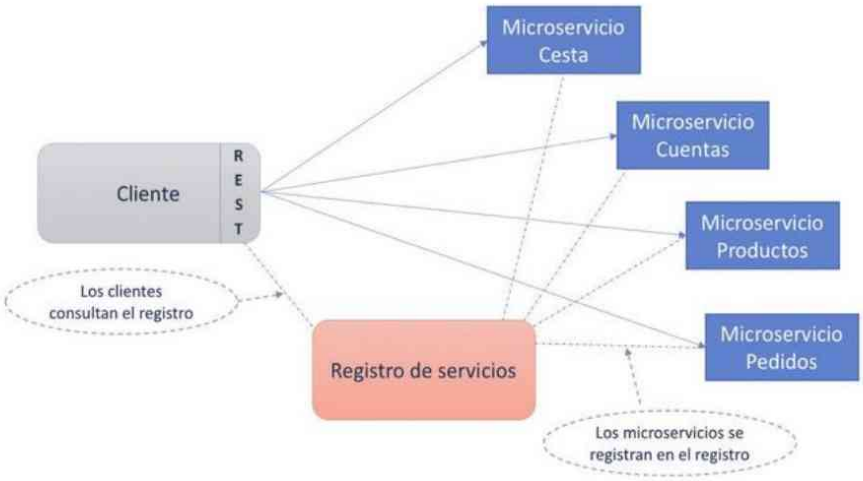

Figura 2.15. Descubrimiento Client Side

El *framework* de Netflix usa un esquema de este tipo gracias a Eureka y Ribbon, que será el que empleemos en la aplicación de ejemplo y que estudiaremos en detalle más adelante.

2.4.3 Comunicación entre microservicios

En una aplicación monolítica las llamadas entre los diferentes componentes de la aplicación se basan en llamadas en memoria o a una DLL. Sin embargo, en un entorno distribuido como es el de los microservicios, los procesos se ejecutan, en general, en distintas máquinas por lo que es necesario algún mecanismo entre procesos (IPC, *Inter Proccess Communication*). Concretamente, se usa HTTP REST para las llamadas a las API de los microservicios (es decir, para ejecutar una funcionalidad determinada) y mensajería asíncrona para otras tareas.

2.4.3.1 HTTP REST

REST (*Representational State Transfer*) es protocolo basado en HTTP orientado a desarrollar aplicaciones web de manera muy sencilla y especialmente adecuado para el diseño de API. Se trata de un protocolo sin estado, es decir, que no debe guardarse información de estado en el servidor, sino que toda la información necesaria debe encontrarse en la consulta del cliente.

A la hora de diseñar API REST hay que tener en cuenta que las URI (*Universal Resource Identifier*) deben usarse correctamente, para lo cual, conviene ceñirse a las siguientes reglas:

- Debe evitarse la inclusión de verbos en las URI ya que no tienen que estar ligadas a una acción concreta. Las acciones se especifican mediante comandos de HTTP.

Comando HTTP	Utilidad
GET	Consulta de recursos
POST	Creación de recursos
PUT	Edición de recursos
DELETE	Borrado de recursos
PATCH	Edición de partes concretas de un recurso

Tabla 2.2. Comandos HTTP

- Un mismo recurso debe identificarse con una sola acción.
- Las URI no dependen del formato del recurso si no de su identificador.
- Debe mantenerse una jerarquía lógica.
- Si necesario filtrar la información, estos filtros no se incluyen en las URI, sino en los parámetros de la petición HTTP.

Otro punto a tener en cuenta son los códigos de estado de HTTP. En efecto, uno de los errores más frecuentes consiste en inventarse códigos de estado de propios de la aplicación cuando, en realidad, podría hacerse uso del amplio abanico de códigos que ofrece directamente el estándar (RFC 2616[2]). En general, los códigos de estado de HTTP se agrupan en varios tipos:

2 RFC 2616, https://tools.ietf.org/html/rfc2616. Accedido el 11/12/2017

- 1XX Respuestas informativas
- 2XX Peticiones correctas
- 3XX Redirecciones
- 4XX Errores del cliente
- 5XX Errores de servidor

2.4.3.2 MENSAJERÍA ASÍCRONA

La mensajería asíncrona se usa, fundamentalmente, para la comunicación entre los distintos microservicios a través de un intermediario de mensajes o *bróker*, de tal manera que los productores de mensajes van colocándolos en una cola distribuida de la que los consumidores van recuperando. De este modo, productor y consumidor quedan totalmente desacoplados.

2.5 ARQUITECTURA DE MICROSERVICIOS

Como hemos visto, las arquitecturas basadas en microservicios surgen para dar respuesta a los problemas de escalado de aplicaciones, en particular cuando se trata de aplicaciones distribuidas.

Sin embargo, no todos los microservicios realizan tareas análogas, salvando las distancias impuestas por su dominio actuación, si no que puede distinguirse entre microservicios de negocio, que aplican las reglas de negocio; microservicios compuestos, que coordinan un cierto número de los anteriores; y, microservicios API, que son los que exponen funcionalidades al exterior.

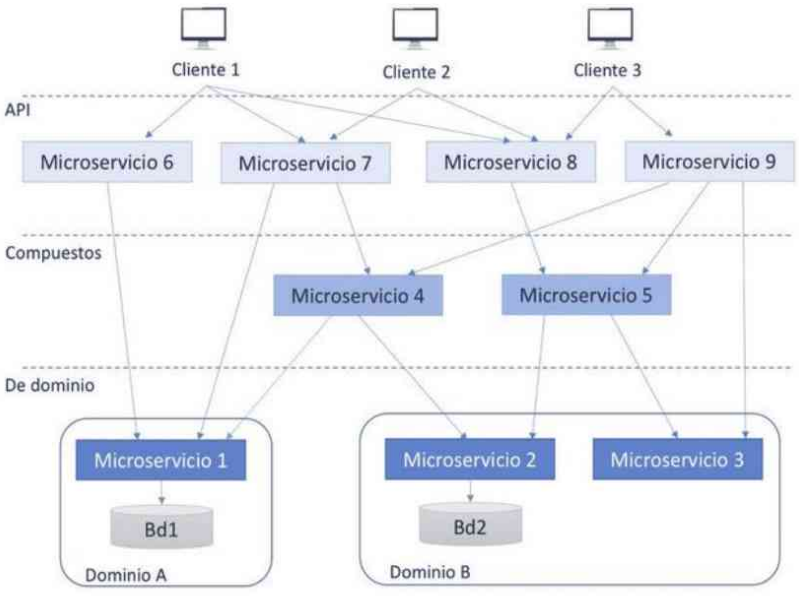

Figura 2.1. Tipos de microservicios

Para gestionar un entorno tan complejo se hacen necesarios tres modelos, a saber:

- Un modelo de referencia, en el que se establecerán las necesidades de la arquitectura de microservicios.
- Un modelo de implementación, donde se concretarán los componentes identificados en el modelo de referencia.
- Un modelo de despliegue que establezca la estrategia de puesta en marcha de los componentes de la arquitectura.

2.5.1 Modelo de referencia

La Figura 2.17 muestra los componentes funcionales del modelo de referencia de una arquitectura basada en microservicios:

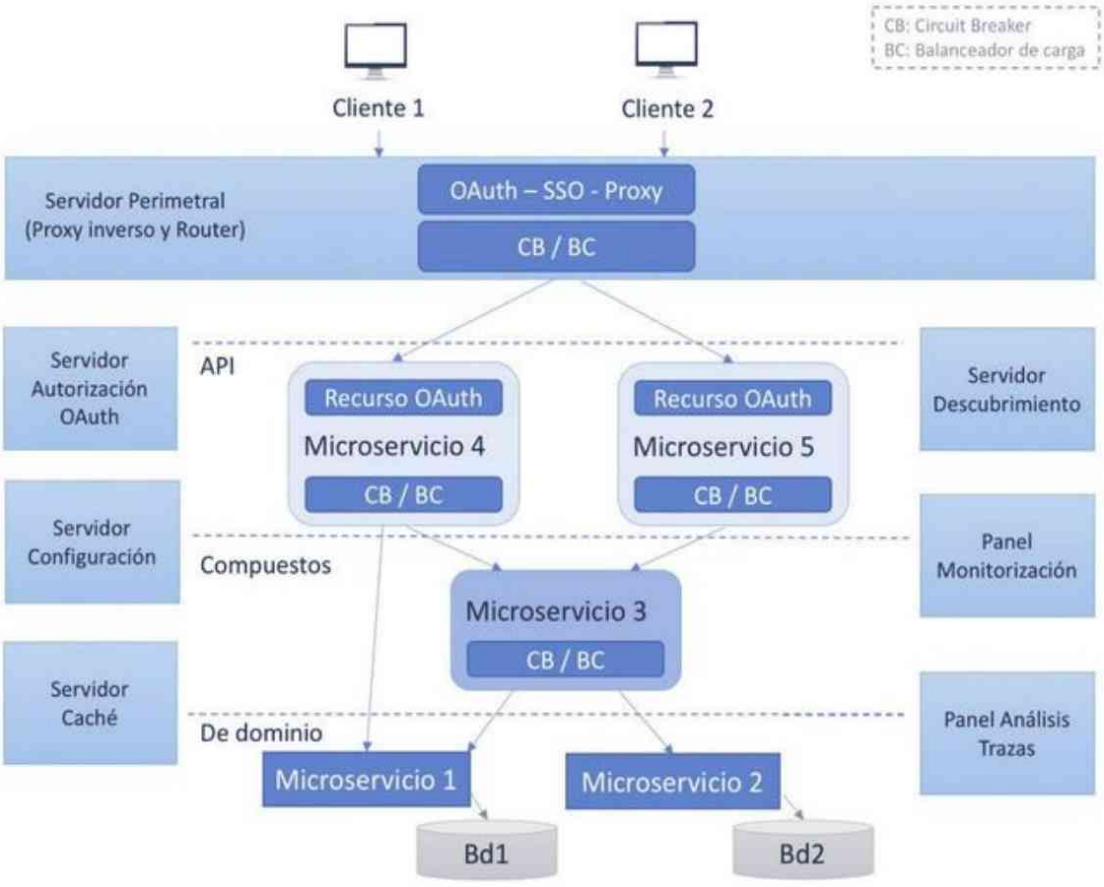

Figura 2.2. Modelo de referencia para microservicios

- Configuración central: centraliza la configuración de los microservicios y la mantiene, por defecto, en un repositorio GIT.
- Registro de microservicios: los microservicios se registran automáticamente en un servidor centralizado, lo que permite su localización y que los componentes de balanceo de carga sean capaces de elegir la instancia adecuada en cada momento.
- Balanceo de carga y enrutamiento dinámico: escoge la instancia del microservicio que se debe utilizar en base a la carga de las mismas de forma totalmente transparente a la hora de consumir el microservicio.
- Tolerancia a fallos: el objetivo es evitar los fallos en cascada evitando que se propaguen. Para ello, se encapsula la función que se desea proteger en un objeto *Circuit Breaker* (CC), que monitoriza los fallos. La mejor forma de entender el funcionamiento del patrón CC es a través de una máquina de estados (ver Figura 2.18). Inicialmente, el CC se encuentra en estado cerrado, de forma que permite que se invoque al recurso externo. Cuando se producen N errores seguidos, considera que el recurso externo no está accesible y se abre el circuito. En este estado las llamadas al sistema remoto no se realizan, sino que fallan directamente, evitando cargar el sistema remoto y evitando costes locales. Tras un tiempo prudencial, se pasa al estado Semi Abierto en el que la siguiente solicitud que se realice se enviará al recurso remoto para comprobar si ha vuelto a la vida. Si es así, el CC pasa a estado Cerrado y vuelve a funcionar normalmente. En caso contrario, vuelve a estado Abierto.

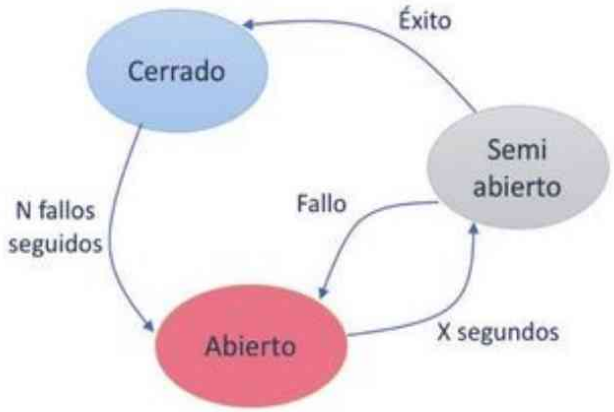

Figura 2.3. Patrón Circuit Breaker

- Exposición de servicios: se trata un *API Gateway* que expone los microservicios API que se van a consumir y previene accesos no autorizados. Se trata de una especie de *proxy* inverso activo y dinámico.
- Centralización de *logs*: las trazas de los microservicios es necesario que estén centralizadas puesto que resultaría inviable la consulta de cada una de ellas de manera individual. Además, de esta forma es más sencillo buscar y consultar información.
- Autenticación: implementa una capa de seguridad sobre los servicios API expuestos al exterior. Generalmente, se utiliza.
- Monitorización: ofrece indicadores sobre los distintos microservicios, tales como carga, salud, etc.

2.5.2 Modelo de implementación

Definidos los componentes de la arquitectura es siguiente paso en la implementación de cada uno de los mismos y para ello utilizaremos la pila tecnológica de Spring Cloud y Netflix OSS, que será la que emplearemos en la aplicación de ejemplo y que se resume en la Tabla 2.4:

Componente	Spring Cloud y OSS Netflix
Servidor de configuración central	Spring Cloud Config
Servicio de descubrimiento	Netflix Eureka
Enrutamiento dinámico y balanceador de carga	Netflix Ribbon
Circuit Breaker	Netflix Hystrix
Monitorización	Netflix Hystrix Dashboard y Turbine
Análisis centralizado de log	Elasticsearch, Logstash, Kibana (ELK)
Servidor Perimetral	Netflix Zuul
APIs de protección basadas en estándar OAuth	Spring Cloud + Spring Security OAuth
Microservicio	Spring Boot , Spring Cloud Data Flow , Spring Cloud Stream, Librerias comunicación con backends
Infraestructura de comunicación asíncrona	RabbitMQ

Tabla 2.4. Modelo de implementación

2.5.3 Modelo de despliegue

Una vez que ya están implementados los distintos componentes del modelo de referencia, hay que organizar los despliegues y la puesta en producción de todos ellos. Para ello, existen dos enfoques principales: las máquinas virtuales y los contenedores.

Un contenedor es paquete ejecutable ligero y autocontenido de un fragmento de software que incluye todo lo necesario para ejecutarse: código, entorno de ejecución, herramientas del sistema, librerías, configuración, etc. De esta manera, el software se aísla de las particularidades de la plataforma en que se ejecuta. Por el contrario, una máquina virtual (VM, Virtual Machine) sigue una filosofía distinta. Comienzan con un sistema operativo completo y, dependiendo de la aplicación, los desarrolladores pueden o no eliminar componentes no deseados. La Figura 2.19 muestra estas diferencias comparando los modelos de capas de ambas aproximaciones:

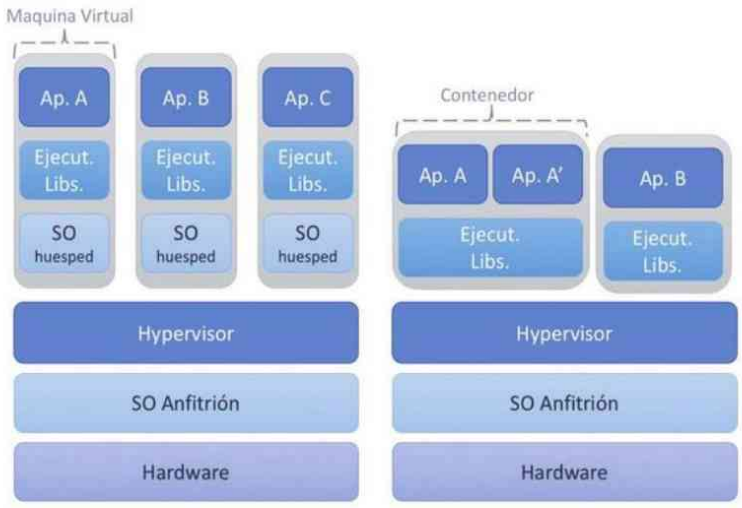

Figura 2.1. Diferencias entre contenedores y máquinas virtuales

La principal diferencia entre las máquinas virtuales y los contenedores es la forma en que se emplean los recursos virtuales de las máquinas físicas en que se ejecutan. En las máquinas virtuales se comparten los recursos físicos de la máquina siendo la asignación de recursos responsabilidad de la capa de virtualización (en este caso, Hypervisor). Los contenedores, por su parte, se ejecutan como un proceso más y de la asignación de recursos físicos se encarga el núcleo del sistema operativo, lo que cada contenedor necesita uno.

De lo anterior se deriva que los contenedores suponen una disminución de la complejidad y la sobrecarga de memoria respecto a las máquinas virtuales, además

de que los tiempos de inicio, apagado y ejecución también son menores, lo que, unido a su mejor capacidad de portabilidad, simplifica el despliegue en entornos de producción y desarrollo.

Sin embargo, los contenedores presentan problemas de disponibilidad, ya que los recursos físicos son compartidos, es decir, que un fallo en un recurso físico podría afectar a varios contenedores. Esto no ocurre en entornos de máquinas virtuales porque la capa de virtualización aísla unos recursos de otros.

Aunque más adelante en esta obra volveremos sobre este tema, basta decir aquí que se optará por contenedores Docker. Docker es una plataforma *open* source que ofrece contenedores ligeros enfocados a simplificar y acelerar la entrega de aplicaciones en entornos *cloud* distribuidos. En este libro utilizaremos una estrategia basada en la definición de un contendero Docker por cada microservicio. Así pues, el modelo de despliegue será similar al de la Figura 2.20:

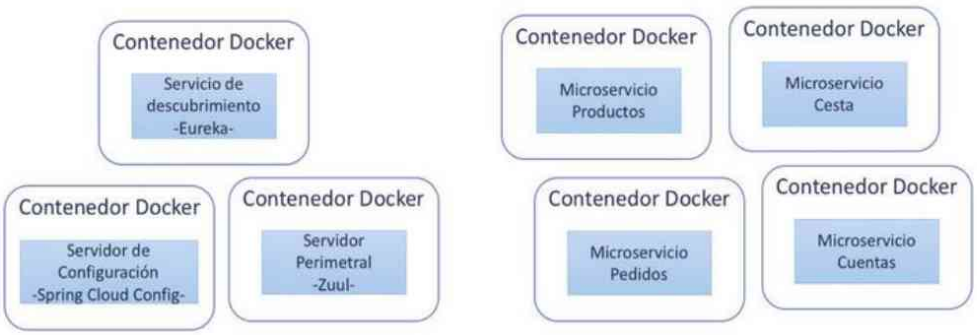

Figura 2.2. Modelo de despliegue

2.6 REFACTORIZACIÓN DE UNA APLICACIÓN MONOLÍTICA A MICROSERVICIOS

El proceso de transformación de una aplicación monolítica en un ecosistema de microservicios es una forma de modernización de dicha aplicación, algo que se ha venido haciendo por parte de los desarrolladores casi continuamente.

Antes de sugerir cómo realizar la evolución, creemos importante hacer notar lo que no hay que hacer. Hay que huir siempre de la estrategia de Big Bang. Esta estrategia consiste en focalizar todo el esfuerzo de desarrollo en construir desde cero una nueva aplicación basada en microservicios. Martin Folwer, conocido gurú, suele señalar que lo único que garantiza una estrategia de Big Bang es un verdadero Big Bang.

En lugar de esta estrategia resulta mucho más conveniente abordar la refactorización gradual de la aplicación monolítica cada vez que se añada una nueva

funcionalidad o se cree una ampliación de una existente. De esta manera, con el tiempo el uso de la parte monolítica de la aplicación será residual y, finalmente, desaparecerá.

El primer mecanismo de migración gradual es el llamado enfoque de dejar de cavar y se basa en la Ley de los Agujeros que dice que "cuando encuentras un agujero, deja de cavar". En el entorno de aplicaciones monolíticas advierte que cuando la aplicación se ha vuelto imposible de gestionar, es mejor no contribuir a hacer crecer el monolito, es decir, que a partir de ese momento todas las funcionalidades nuevas y el código que se añade debe estar concentrado en un único servicio (ver Figura 2.21).

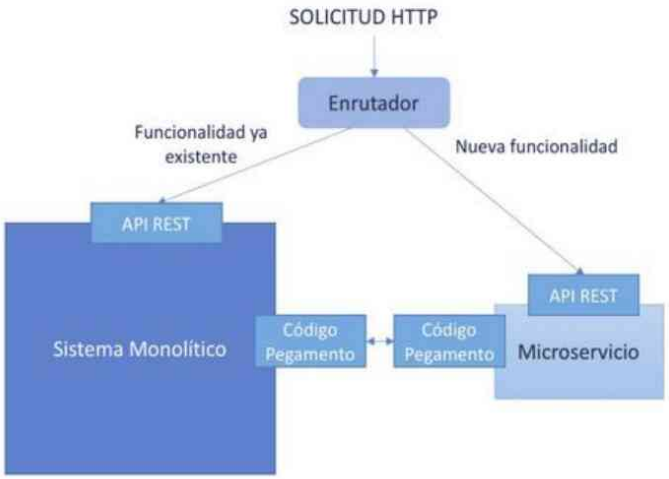

Figura 2.3. Estrategia de dejar de cavar para la evolución a microservicios

Además, del nuevo servicio y del monolito tradicional, existen otros dos componentes. El primero es un enrutador de peticiones, que recibe las peticiones HTTP (de manera parecida a un API Gateway) y las envía, bien al monolito o bien al nuevo servicio dependiendo la funcionalidad de que se trate. El otro componente es el código o lógica de pegamento, responsable de la integración de datos entre el monolito y el servicio nuevo.

El inconveniente obvio de esta aproximación es que no resuelve los problemas del monolito. Para resolverlos es necesario romper el monolito.

Un enfoque que desgaja la aplicación monolítica es distribuir cada una de las capas de su arquitectura. Las aplicaciones (monolíticas) se han desarrollado en distintas capas desacopladas entre sí: presentación, lógica de negocio y acceso a datos. Podemos llevar ese desacoplamiento un paso más allá al distribuir las llamadas entre cada una de las capas tal y como muestra la Figura 2.22.

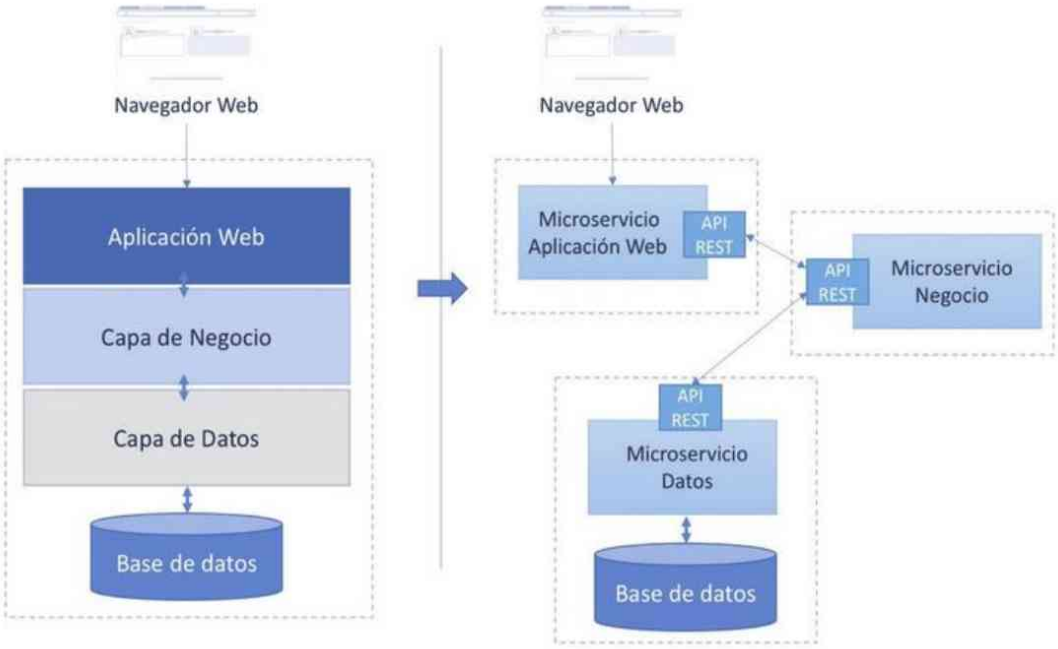

Figura 2.4. Evolución por distribución del back-end y el front-end.

Dividir el monolito de esta manera tiene dos beneficios principales: por un lado, permite desarrollar, desplegar y escalar las dos aplicaciones de manera independiente la una de la otra; y, por otro, expone un API remota que puede invocarse desde los microservicios que se desarrollen. Esta estrategia, sin embargo, constituye una solución parcial. Es muy probable que una o las dos aplicaciones se conviertan en un monolito imposible de gestionar.

Para solucionar los problemas anteriores existe una tercera estrategia: la extracción de servicios. En este caso, los módulos del monolito se convierten en microservicios independientes. Cada vez que se extrae un módulo y se convierte en microservicio, el monolito mengua. Si conseguimos convertir suficientes módulos, el monolito dejará de ser un problema. La clave está, por tanto, en cómo seleccionar los módulos que se convertirán en microservicios.

Una buena aproximación es comenzar con los módulos que sean más sencillos de extraer. De este modo, el equipo de desarrollo irá ganando experiencia con los microservicios en general y con el proceso de extracción en particular. A continuación, se extraen aquellos módulos que proporcionan mayores beneficios, puesto que la conversión de un módulo en un microservicio es un proceso que consume mucho tiempo. Suele ser útil extraer los módulos que cambian con mayor frecuencia, ya que una vez extraído, el desarrollo y el despliegue del mismo es independiente del monolito, lo que acelera el proceso de desarrollo. También resulta recomendable extraer los

módulos con necesidades de recursos significativamente distintos del resto del monolito o aquellos que implementan algoritmos computacionalmente muy pesados.

Una vez identificados los módulos, el paso siguiente es convertirlos en microservicios. En el ejemplo de la Figura 2.23, el monolito tiene tres módulos 1, 2 y 3, siendo el 3 el módulo candidato para su transformación en microservicio. En principio, los módulos 1 y 2 están acoplados al módulo 3 (y viceversa) puesto que utilizan sus componentes, objetos, etc. El primer paso de la refactorización es definir un par de interfaces gruesas entre el módulo y el monolito. La primera interfaz es de una entrada que será empleada por el módulo 1 para invocar al módulo 3 y la segunda será una interfaz de salida para que el módulo 3 pueda invocar al módulo 2. Hecho esto, ya es posible considerar al módulo 3 como un microservicio.

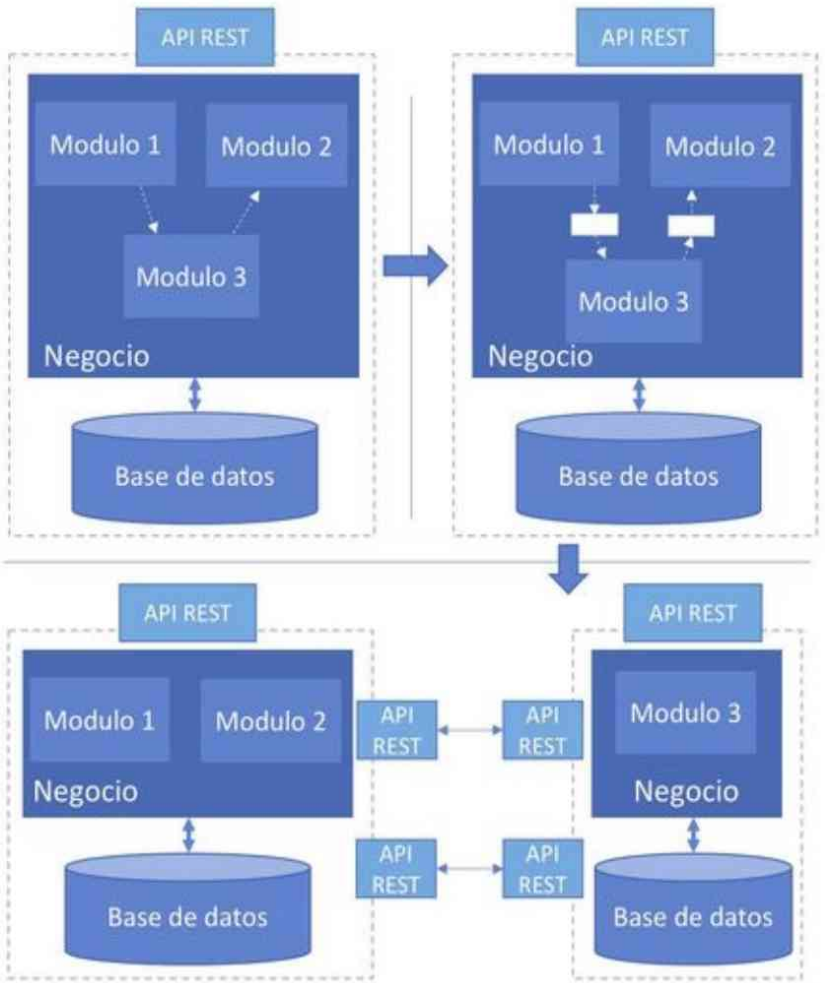

Figura 2.5. Extracción de microservicios

3

HERRAMIENTAS DE DESARROLLO

Para llevar a la práctica el desarrollo de Microservicios, vamos a hacer uso de tecnología Java. Damos por hecho que el lector está familiarizado con esta tecnología y tiene instalado el JDK 8 o superior de Java. Como entorno de desarrollo vamos a hacer uso de Eclipse. Spring Boot y Gradle serán las herramientas que utilizáramos para la construcción y despliegue de proyectos, así como para la gestión de dependencias.

3.1 ECLIPSE

Eclipse[3] es una plataforma de código abierto implementada en Java que proporciona soporte para múltiples actividades de desarrollo. Para ello, presenta una arquitectura basada en módulos (*plugins*) que permiten incorporar herramientas de desarrollo con objetivos diferentes. Aunque su distribución por defecto incorpora plugins centrados en el desarrollo JEE, desde su página de descargas[4] podemos encontrar distribuciones con plugins para el desarrollo PHP, JavaScript, modelado conceptual, etc.

Además, Eclipse proporciona un *Marketplace* en el que podemos acceder a la multitud de plugins que tiene disponible y extender nuestra distribución de Eclipse con las herramientas que necesitemos. En la siguiente subsección, veremos cómo instalar un plugin que nos ayudará en el desarrollo de aplicaciones con Gradle, una

3 https://www.eclipse.org

4 https://www.eclipse.org/downloads/eclipse-packages/

herramienta que automatiza la construcción de nuestros proyectos y que veremos en detalle en la sección 3.2.

De momento, bastará con que el lector se descargue la distribución por defecto de Eclipse, que como ya hemos dicho, se centra en el desarrollo JEE. La descarga puede realizarse siguiendo el enlace 'Download' disponible en la página principal del proyecto, o desde la página de descargas, proporcionadas ambas anteriormente. Eclipse se distribuye en forma de instalable o archivo comprimido. Desde la página de descargas podemos acceder a las versiones en archivo comprimido para sistemas Windows 32 y 64 bits, Mac OS X de 64 bits, y Linux de 32 y 64 bits. Siguiendo el enlace de la página principal se nos descargará el archivo comprimido para plataformas de 64 bits si accedemos desde Mac OS X o Linux. Si accedemos desde un sistema Windows, se descargará el instalable (archivo .exe) para 64 bits.

Una vez hayamos descargado el instalable, deberemos ejecutarlo y seguir los pasos que nos indica para poder hacer uso de Eclipse. En caso de tener un archivo comprimido, basta con descomprimirlo y tenemos listo el entorno para su uso. Al ejecutarlo, Eclipse nos solicitará que indiquemos un *workspace* (ver Figura 3.1). Este no es más que el directorio donde se guardarán los proyectos que desarrollaremos con el entorno.

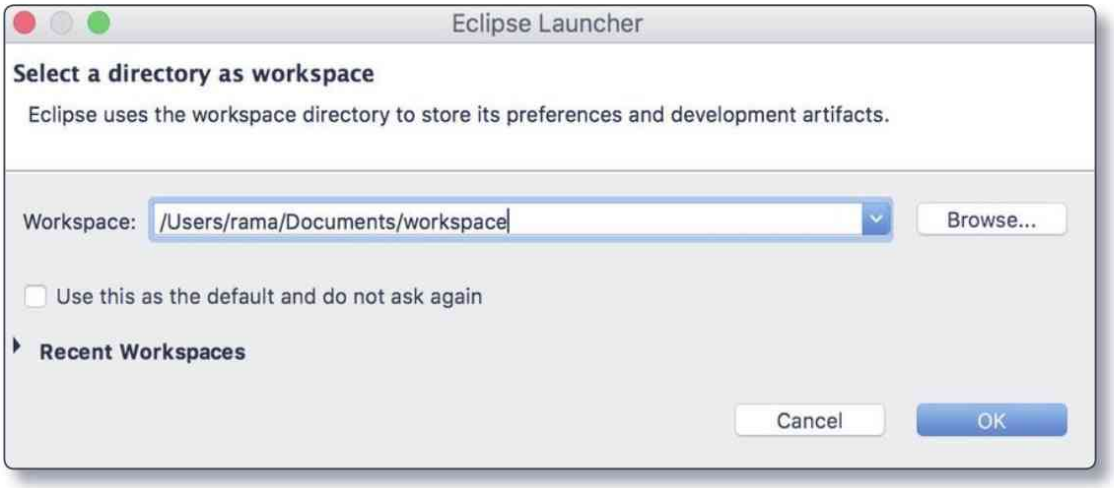

Figura 3.1. Solicitud del directorio de trabajo o *workspace*

Después de introducir el directorio de trabajo nos aparecerá la ventana de bienvenida de Eclipse (ver Figura 3.2) desde la cual podemos acceder al espacio de trabajo o *workbench* pulsando sobre la flecha que aparece en la parte superior derecha (según la versión de Eclipse, la pantalla de bienvenida puede variar, y por tanto el acceso al *workbench* puede aparecer en una ubicación diferente) .

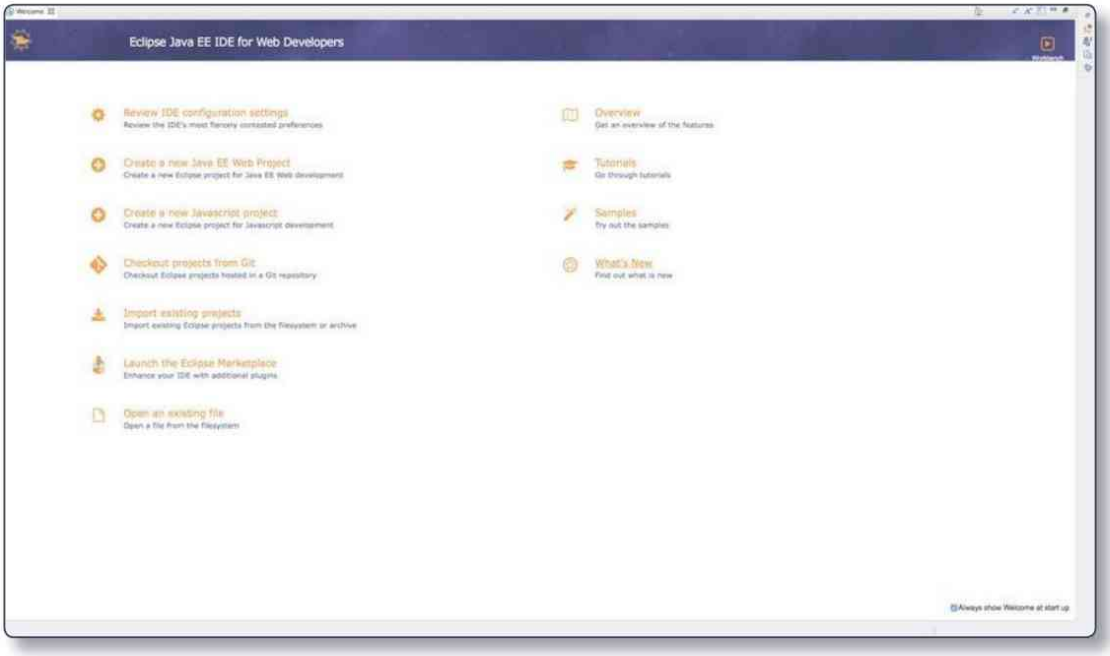

Figura 3.2. Ventana de bienvenida de Eclipse

El espacio de trabajo de Eclipse (Figura 3.3) se compone de un explorador de proyectos en la parte izquierda, donde tendremos disponible los diferentes proyectos que vayamos creando; un editor central, desde donde podremos modificar los archivos de un proyecto; y un conjunto de vistas de soporte que se sitúan en la parte inferior y derecha del entorno.

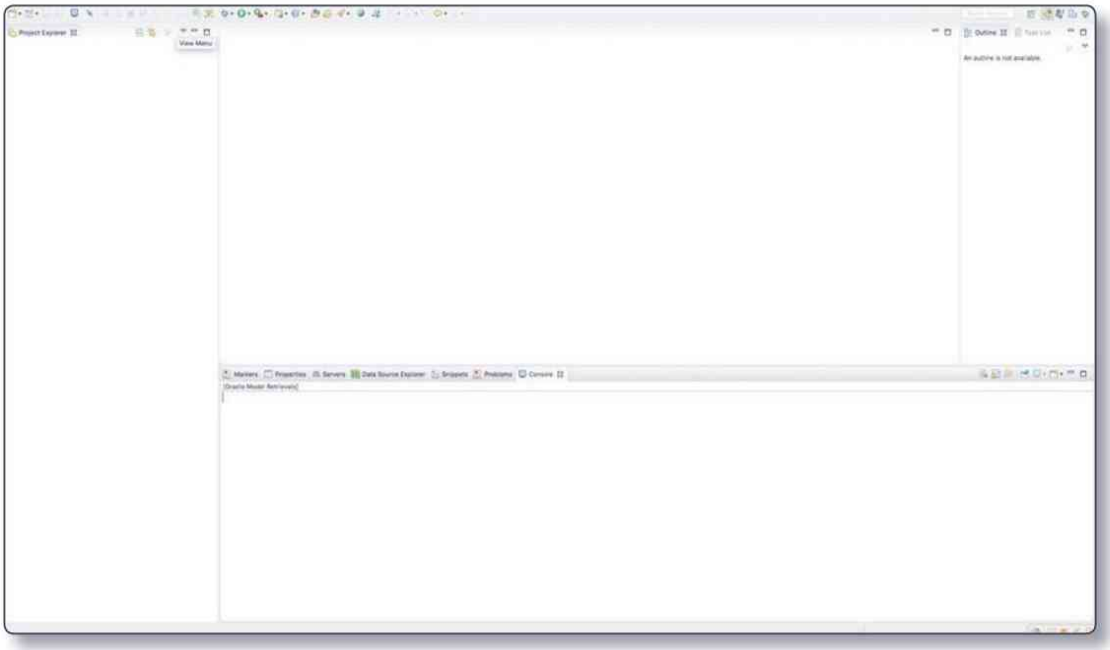

Figura 3.3. Espacio de trabajo o workbench de Eclipse

Para el tipo de desarrollo que nos atañe en este libro, los proyectos que crearemos serán proyectos Gradle. En la siguiente subsección presentaremos la herramienta Gradle. De momento, vamos a preparar Eclipse para poder integrarlo con dicha herramienta. Para ello, necesitamos instalar el plugin de Eclipse Buildship, el cual extiende el entorno de desarrollo con soporte para la construcción de proyectos con Gradle.

Para instalar Buildship haremos uso del *Marketplace* de Eclipse, al cual podemos acceder desde la opción de menú 'Help->Eclipse Marketplace'. La Figura 3.4 muestra la ventana del Marketplace. Para instalar el plugin que necesitamos basta con hacer una búsqueda por su nombre y darle al botón de *Install*.

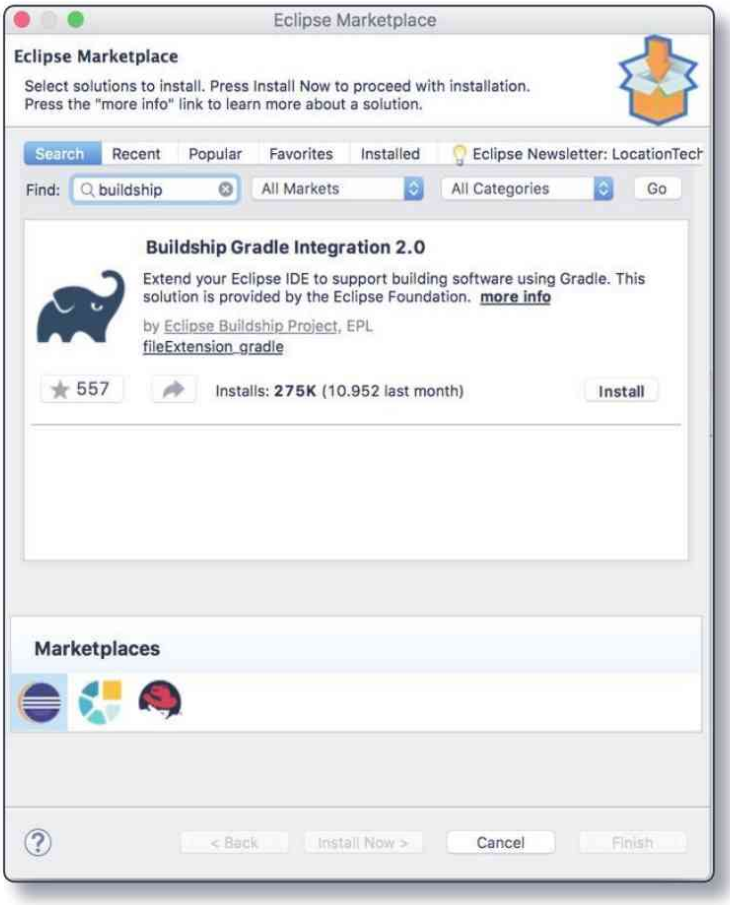

Figura 3.4. Marketplace de Eclipse. Instalación del plug-in Buildship

Una vez instalado el plugin, Eclipse nos pedirá que reiniciemos el entorno. Después de ello, estamos listos para crear nuestro primer proyecto Gradle.

3.2 GRADLE

Gradle es una herramienta para la automatización de la construcción de nuestro código. Basándose en las aportaciones de otras herramientas como Ant o Maven, Gradle va un paso más allá proporcionando un lenguaje más sencillo y claro a la hora de especificar la construcción a realizar, además de soportar múltiples lenguajes, no sólo Java.

Gradle está implementado en Groovy, que es un lenguaje de scripts que se ejecuta sobre Java. Por tanto, necesitamos tener instalado el JDK 6[5] o superior. Para verificar la versión de Java que tenemos instalada en nuestro sistema podemos lanzar el siguiente comando desde un terminal:

```
> java -version
```

En cuanto a Groovy, el propio instalador de Gradle comprueba si está instalado en nuestro sistema y en caso de no ser así, lo instala.

Si nos descargamos Gradle desde su página web[6], veremos que nos instala una herramienta para ser utilizada desde un terminal, permitiéndonos la gestión de las diferentes fases de un proyecto software, así como de sus dependencias. Sin embargo, la mayoría de entornos de desarrollo presentan soporte para su integración con Gradle. Como ya hemos visto anteriormente, en el caso de Eclipse, este soporte lo proporciona el plugin Buildship.

3.2.1 Nociones básicas de Gradle

Los conceptos básicos de Gradle son proyectos, tareas y scripts de construcción. Un proyecto es típicamente un software que se desea construir. Las tareas son acciones necesarias para construir el proyecto, como por ejemplo compilar el código fuente, generar documentación JavaDoc, comprimir las clases compiladas en un archivo JAR, etc. Un proyecto de Gradle contiene una o más tareas para construir el proyecto.

Un proyecto Gradle generalmente tiene un script de construcción en el que se definen sus tareas. Este script es utilizado por la herramienta Gradle para saber qué

5 Dado que para utilizar Spring Boot necesitamos el JDK 8 o superior, se recomienda comprobar que se dispone de esta versión

6 https://gradle.org/

tareas se definen para el proyecto. Se suele definir en un fichero llamado *build.gradle* y normalmente se encuentra en el directorio raíz del proyecto que está creando.

La filosofía de diseño de Gradle se basa en que todas las utilidades que proporciona se incluyen en un proyecto a través de la aplicación de plugins (este es el concepto utilizado, *apply plugin*).

Así pues, si queremos que nuestro proyecto tenga disponible las tareas relacionas con el desarrollo java (*assemble*, *build*, *clean*, *compile,* etc.), bastará con que apliquemos este plugin indicándolo en el fichero *build.gradle*.

```
apply plugin: 'java'
```

Es normal que los plugins que se apliquen necesiten cierta configuración para realizar determinadas tareas. Por ejemplo, si lo que queremos es realizar la tarea de empaquetado un proyecto java en un archivo jar (*assemble*), necesitamos indicar cuál es la clase principal para que se incluya en el *manifest*. También podemos querer indicar el nombre del fichero generado y su versión. Para ello, basta con añadir la siguiente configuración en el fichero *build.gradle*.

```
jar{
    manifest {
        attributes 'Main-Class': 'application.MainApp'
    }
    baseName = 'es.rama.books '
    version =  '0.1.0'
}
```

Del mismo modo, podemos configurar el plugin **java** para que busque las dependencias externas de nuestro proyecto. Sólo tenemos que indicarle el repositorio donde debe buscarlas y las propias dependencias. Una dependencia externa se describe a través de su grupo (**group**), nombre (**name**) y versión (**version**). También es posible indicar el momento en el cual son necesarias en nuestro proyecto (en compilación, **compile**; en tiempo de ejecución, **runtime**; durante el testing, **testCompile** o **testRuntime**; o en el momento de empaquetado, **archives**). Si no se indica nada, Gradle entiende que las dependencias son necesarias en tiempo de ejecución.

En el siguiente ejemplo, hemos configurado el plugin de Java para que busque una dependencia de Spring Boot (a través de su grupo, nombre y versión) en el repositorio de Maven, de modo que esté disponible en tiempo de compilación.

```
repositories {
   mavenCentral()
}
```

```
dependencies {
   compile group:'org.springframework.boot', name:'spring-boot-
starter-web, version: 2.0.0.RELEASE'
}
```

Para indicar las dependencias también se puede utilizar la notación **grupo:nombre:versión**, de forma que no es necesario indicar el nombre de cada propiedad. La siguiente declaración sería equivalente a la anterior.

```
dependencies {
   compile 'org.springframework.boot:spring-boot-starter-web:
2.0.0.RELEASE'
}
```

Hasta ahora hemos visto configuraciones que son utilizadas por los plugins para realizar determinadas tareas sobre el proyecto que queremos construir, como la definición de la clase principal para el jar o las dependencias externas necesarias para compilar nuestro código. Sin embargo, es posible que algunos plugins necesiten configuraciones adicionales que Gradle debe aplicar sobre ellos mismos para su correcto funcionamiento. Por ejemplo, algunos plugins pueden necesitar dependencias externas (los plugins Gradle son implementaciones Java que pueden necesitar de código externo). Este tipo de configuraciones las podemos definir a través del bloque **buildscript**, tal y como se muestra en el siguiente ejemplo, donde estamos configurando la dependencia externa que necesita el plugin de Spring Boot que veremos en la sección 3.3.

```
apply plugin: 'org.springframework.boot'
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle
plugin:1.5.3.RELEASE")
    }
}
```

Del mismo modo que Gradle permite aplicar plugins que nos facilitan determinadas tareas enfocadas a un tipo de desarrollo (por ejemplo java), también dispone de plugins que facilitan la ejecución de tareas sobre el propio entorno de desarrollo. Por ejemplo, el plugin **eclipse** (existen plugins análogos para otros entornos de desarrollo) incorpora dos tareas que nos permiten hacer un *clean* sobre un proyecto eclipse o generar los archivos del mismo.

```
apply plugin: 'eclipse'
```

3.2.2 Usando Gradle desde Eclipse

Para crear un proyecto Eclipse que pueda ser gestionado por Gradle tenemos que seleccionar la opción de menú 'File->New->Other' y en la ventana que nos aparece seleccionar la opción Gradle Project (ver Figura 3.5).

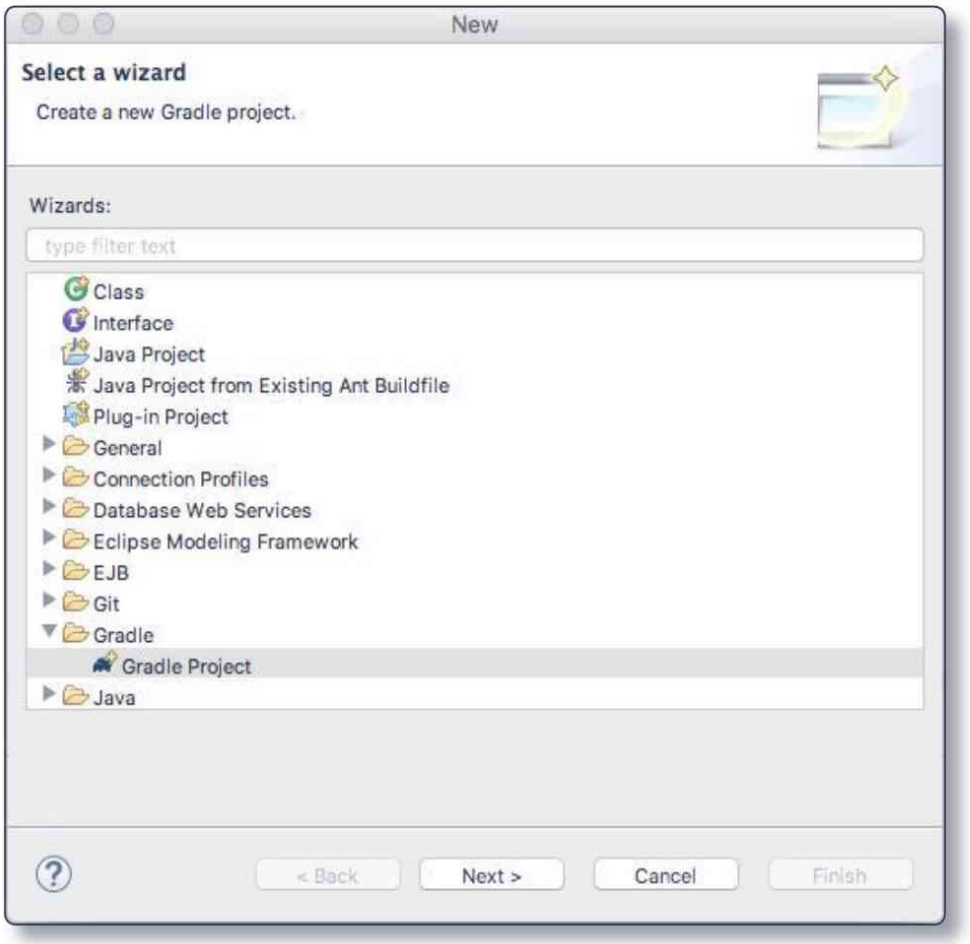

Figura 3.5. Nuevo proyecto Gradle

Para crear un proyecto Gradle basta con indicar el nombre del mismo en la ventana de la Figura 3.6 y darle al botón de finalizar (*Finish*). Si pulsamos sobre el botón siguiente (*Next* >) nos aparecerá una ventana con información resumida del proyecto antes de la creación del mismo.

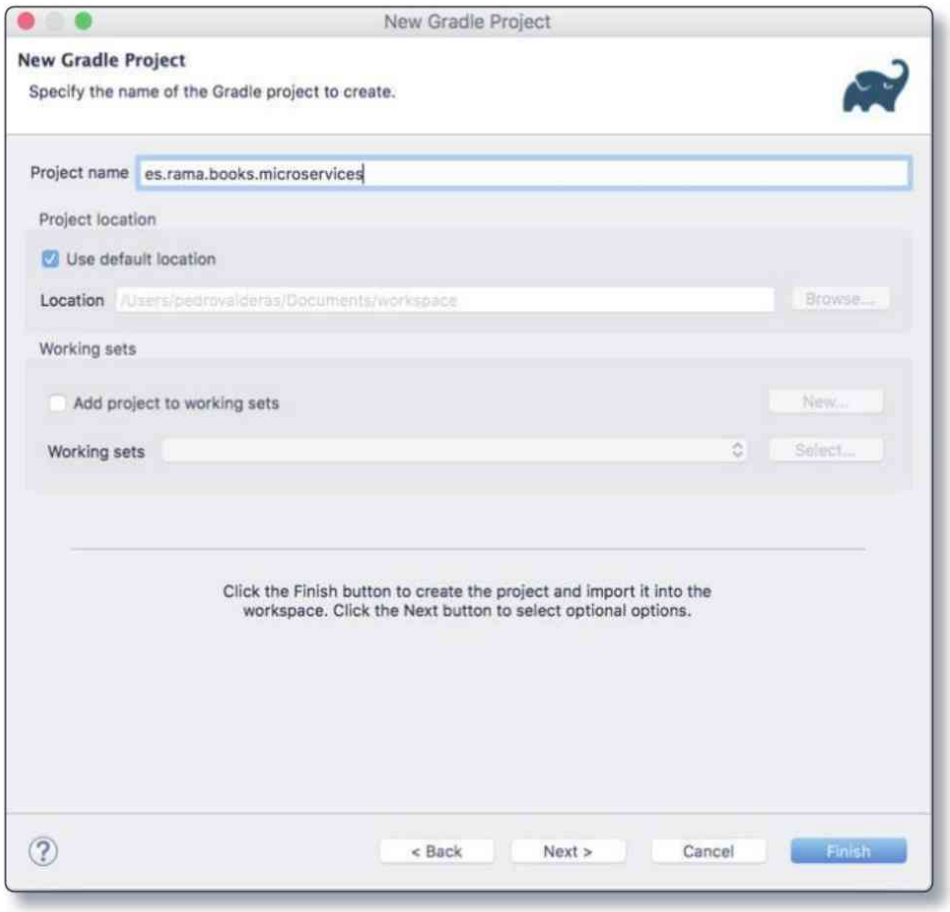

Figura 3.6. Creación de un proyecto Gradle

El proyecto creado tiene una estructura como la mostrada en la Figura 3.7. Por un lado, tenemos las carpetas de código Java, que incluyen el código fuente del proyecto y el de testing. Durante el proceso de creación del proyecto, Eclipse genera dos clases de ejemplo (**Library** y **LibraryTest**) que pueden ser eliminadas. Por otro lado, tenemos una carpeta **gradle** que incluye la implementación de la herramienta en un fichero jar. Por último, colgando del directorio raíz, tenemos unos ficheros de configuración de Gradle, entre los que se encuentra el fichero **build.grade**. Los otros ficheros son scripts que nos permiten gestionar el proyecto desde un terminal, pero dado que nosotros trabajaremos siempre desde Eclipse no los vamos a usar.

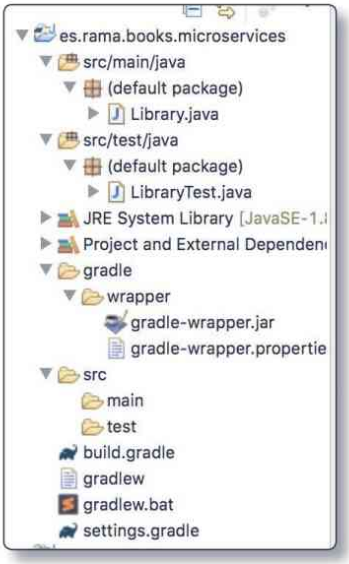

Figura 3.7. Estructura de un proyecto Gradle en Eclipse

Cabe destacar que Eclipse dispone de la vista *Gradle Tasks*[7] en la parte inferior del entorno (ver Figura 3.8). Esta vista nos permite lanzar las tareas aportadas por los diferentes plugins aplicados sobre un proyecto. En este caso, vemos todas las tareas que aportan los plugin **eclipse** (nodo ide) y **java** (el resto de nodos del árbol). Como veremos más adelante, cuando apliquemos otros plugins aparecerán otras tareas.

Figura 3.8. Vista de Eclipse para trabajar con Gradle

7 Si esta vista no está disponible puede activarse desde la opción de menú Window->Show View->Other..., seleccionando a continuación Gradle->Gradle Tasks

Para ejecutar alguna de las tareas sobre nuestro proyecto (por ejemplo, *assemble*, *build*, o *jar*) basta con desplegar el árbol que nos aparece en la vista Grade (ver figura anterior), hacer clic con el botón derecho sobre la tarea a realizar, y seleccionar la opción 'Run Gradle Tasks' del menú contextual que aparece.

Hay que tener en cuenta que Eclipse no es consciente de forma automática de los cambios realizados por Gradle. Por ejemplo, si lanzamos la tarea *build* sobre nuestro proyecto, Gradle comprobará todas las dependencias definidas en el fichero *build.gradle* y las descargará desde el repositorio indicado. Sin embargo, Eclipse no detectará estas dependencias y seguirá dando errores de compilación. Para que esto no ocurra, una vez ejecutada una tarea, debemos decirle a Eclipse que refresque el proyecto Gradle. Para ello podemos usar la opción 'Refresh Gradle Project' del menú contextual que aparece al hacer clic con el botón derecho del ratón sobre el nombre del proyecto desde el explorador de proyectos, tal y como se muestra en la Figura 3.9.

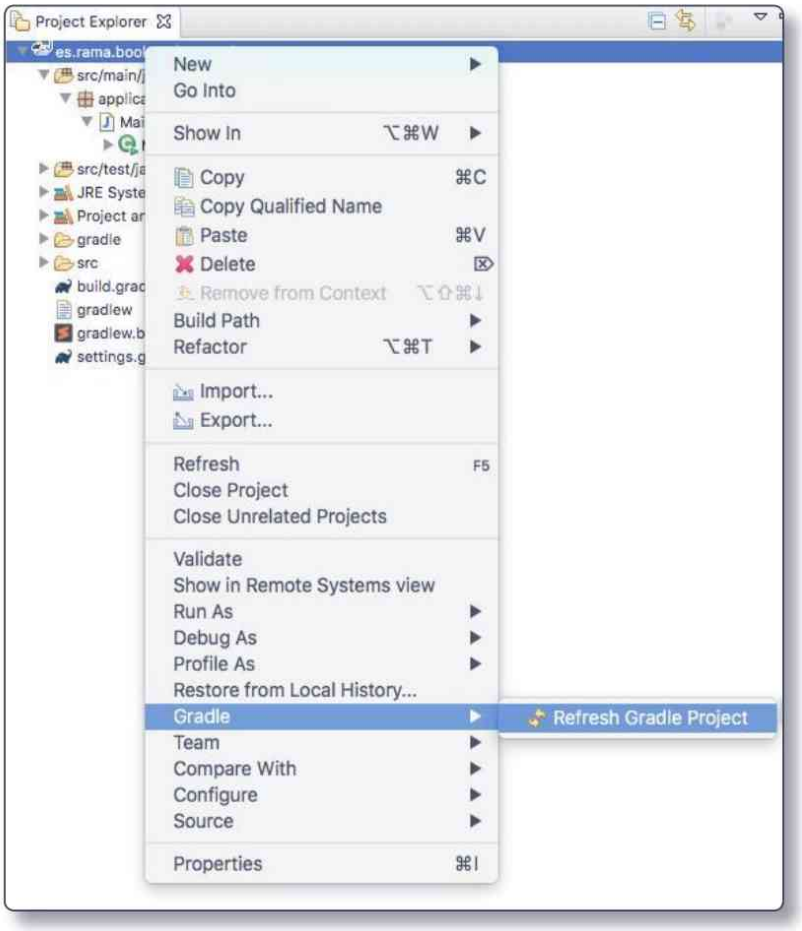

Figura 3.9. Opción para refrescar un proyecto Gradle

Por último, comentar que en el caso de que apliquemos nuevos plugin a través del fichero *build.gradle*, será necesario cerrar y volver a abrir la vista de tareas Gradle para que aparezcan las nuevas tareas incorporadas por el plugin añadido.

3.3 SPRING BOOT

Spring Boot[8] es un proyecto de Spring[9] cuyo objetivo es facilitar la configuración y desarrollo de aplicaciones web[10] Java a través de la tecnología proporcionada por el framework Spring.

En este libro no vamos a entrar en detalles sobre el framework Spring. Basta con saber que su núcleo proporciona soporte para la Inversión de Control, de forma que, en una aplicación, ya no es responsabilidad del desarrollador el crear los objetos necesarios, sino que es el propio Spring el que se encarga de ello basándose en ficheros de configuración XML o anotaciones. Esto se conoce también como inyección de dependencias.

Alrededor del núcleo de Spring se ha desarrollado todo un ecosistema de proyectos que proporcionan herramientas para facilitar el desarrollo de diferentes tipos de aplicaciones. Entre ellos, encontramos el proyecto Spring Boot, el cual tiene como principal objetivo minimizar la configuración Spring en aplicaciones web Java. Además, admite contenedores integrados para permitir que las aplicaciones web puedan ejecutarse de manera independiente, sin necesidad de un servidor web.

Spring Boot se distribuye como una serie de paquetes jar que podemos incluir en nuestro proyecto. Para hacer uso del soporte que nos proporcionan estos jars, basta con utilizar en nuestro código el conjunto de anotaciones disponibles. Para entender mejor cómo funciona esta herramienta, vamos a crear una aplicación web típica, el "Hola Mundo".

Para ello, lo primero que tenemos que hacer es crear una clase principal con la anotación **@SpringBootApplication**. Esta clase principal tendrá un método **main** como en cualquier aplicación Java. Este método debe ejecutar el método estático **run** de la clase **SpringAplication** pasando como primer parámetro una referencia al

8 https://projects.spring.io/spring-boot/

9 https://spring.io/

10 Aunque es posible crear aplicaciones no web, su principal objetivo es el desarrollo de aplicaciones web Java

objeto **Class** de nuestra clase principal. El siguiente ejemplo, muestra el código de la clase principal de nuestra aplicación Spring Boot.

```
package application;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MainApp {

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

Sin embargo, con la clase mostrada anteriormente no disponemos de una aplicación web funcional, ya que nos falta un controlador que se encargue de recibir las peticiones HTTP. Para ello, basta con incluir una clase como la siguiente:

```
package application;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class HTTPController {

   @RequestMapping(“/”)
   @ResponseBody
   String home() {
      return “Hola Mundo”;
   }
}
```

En primer lugar, vemos como la clase HTTPController ha sido marcada con la anotación **@Controller**, por lo que la aplicación Spring delegará en esta clase el procesamiento de peticiones HTTP. Además, vemos que el método **home** está marcado con las anotaciones **@RequestMapping** y **@ResponseBody**, lo que indica que este método devuelve la respuesta a las peticiones cuya URL apuntan a la raíz (“/”). El contenido que devuelve es la cadena de texto “Hola Mundo”.

Por último, comentar que Spring Boot buscará el controlador en el mismo paquete donde está la clase principal, o en sus subpaquetes. Para indicarle que busque en otro/s paquete/s podemos hacer uso de la anotación **@ComponentScan** tal y como se muestra a continuación (para usar dicha anotación es necesario añadir

un nuevo **import**). Si queremos indicar más de un paquete debemos asignar a la propiedad **basePackages** un array en formato JSON que contenga el nombre de dichos paquetes.

```
[…]
import org.springframework.context.annotation.ComponentScan;

@SpringBootApplication
@ComponentScan(basePackages="otroPaquete")
// @ComponentScan(basePackages={"paquete1","paquete2"})
public class MainApp {
[…]
```

Con estas dos clases sencillas, acabamos de crear una aplicación web. Nótese que aunque estamos creando una aplicación web, toda aplicación Spring Boot se crea como una aplicación Java clásica, y se ejecuta como tal, a través de su método main. Spring Boot incluirá un contenedor web (por defecto, Tomcat) embebido en el jar de la aplicación, que automáticamente se lanzará al ejecutarla. Por tanto, no hace falta desplegarla en un servidor web.

Para generar el archivo jar de nuestro proyecto Gradle, basta con lanzar la tarea **jar** que aparece al desplegar el nodo **build** de la vista de Gradle Tasks (ver figura 3.8). Si todo va bien, esta tarea generará un archivo jar con el nombre que hayamos especificado en la sección jar del archivo *build.gradle*. Se genera en la carpeta **build/libs** que podemos encontrar dentro de la carpeta de nuestro proyecto. Hay que tener en cuenta que desde el explorador de proyectos de Eclipse no aparece esta carpeta, por lo que tenemos que acceder a ella desde un terminal o el explorador de archivos.

Una vez hemos generado el archivo jar, basta con ejecutarlo con el comando básico **java**, incluyendo por ejemplo la etiqueta **–jar** para indicarle que se trata de una aplicación empaquetada en un archivo con dicho formato:

```
> java -jar miApp.jar
```

Una vez ejecutada la aplicación, podemos acceder a ella desde cualquier navegador a través de la url **http://localhost:8080/**.

Aunque generando el archivo jar es posible ejecutar nuestra aplicación Spring Boot, vamos a ver a continuación como podemos lanzarla directamente desde Eclipse a través del plugin de Spring Boot para Gradle.

3.3.1 Incorporando en plugin de Spring Boot a nuestro proyecto Gradle

Del mismo modo que hemos aplicado un plugin en nuestro proyecto Gradle para trabajar con aplicaciones Java, es posible aplicar un plugin para dotar del soporte Spring Boot a nuestros proyectos. Este plugin nos va a permitir ejecutar las aplicaciones desde Eclipse o un soporte adicional para la gestión de dependencias. Por ejemplo, con el plugin de Spring Boot la versión de las dependencias es opcional indicarla, de forma que, si no la indicamos el plugin se encarga de descargar la última versión estable.

Para ello, deberemos incluir en fichero build.gradle la cláusula apply con el nombre del plugin (**org.springframework.boot**) y realizar las configuraciones pertinentes. En este caso, debemos incluir las dependencias que necesita Gradle para ejecutar el plugin (mediante la cláusula buildscript) y las dependencias que necesita nuestro código para hacer uso de Spring Boot (el starter para web, nótese que sin versión). La especificación que debemos incluir en el fichero *build.gradle* es la siguiente:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle
plugin:1.5.9.RELEASE")
    }
}
apply plugin: 'org.springframework.boot'
repositories {
    mavenCentral()
}
dependencies {
    compile "org.springframework.boot:spring-boot-starter-web"
}
```

Una vez incorporado el plugin de Spring Boot a nuestro fichero *build.gradle*, y tras cerrar y volver a abrir la vista de Gradle, veremos que tenemos una nueva tarea (*application->bootRun*) que nos permite ejecutar este tipo de aplicaciones. También disponemos de la tarea *build->bootRepackage* que nos va a permitir empaquetar en un jar ejecutable el código de nuestra aplicación junto con todas sus dependencias. Esta tarea la utilizaremos en el capítulo de despliegue. En la siguiente figura se pueden apreciar las dos tareas comentadas.

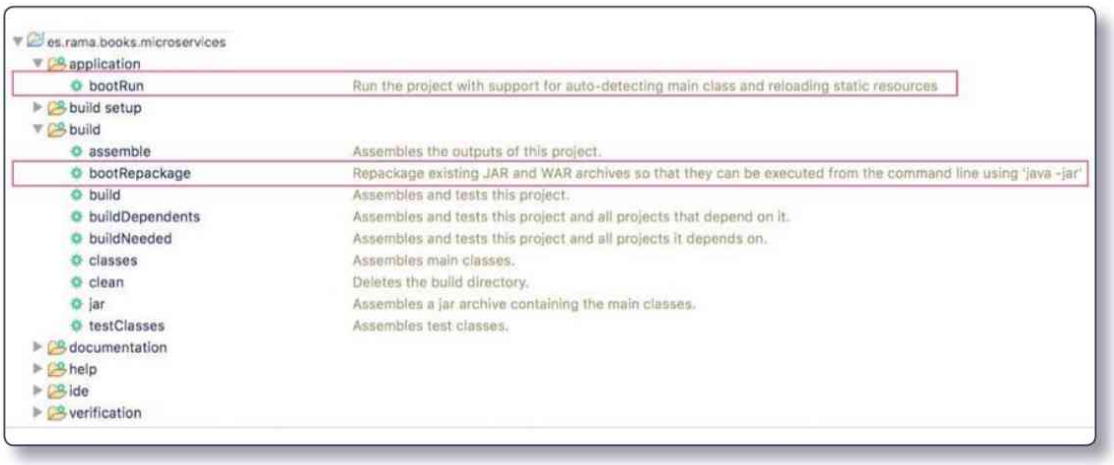

Figura 3.10. Tarea bootRun y bootRepackage para ejecutar y empaquetar aplicaciones Spring Boot

3.4 RESUMEN DE INSTALACIÓN Y USO DEL ENTORNO

En este capítulo hemos visto cómo instalar las herramientas que vamos a utilizar, así como las principales nociones de su uso. A continuación, resumimos los pasos que tenemos que realizar para preparar el entorno de desarrollo y crear una aplicación web que nos sirva como base para la creación de Microservicios.

Instalación del Entorno

1. Descarga e instalación de Eclipse
2. Instalación del plugin BuildShip

Creación del proyecto

3. Creación de un proyecto Gradle
4. Actualización del fichero *build.gradle* para que aplique los plugins **java**, **eclipse** y **org.springframework.boot**, y las configuraciones necesarias por éstos.
5. Abrir la vista Gradle Tasks (si ya estaba abierta la cerramos y volvemos a abrir)

Desarrollo y ejecución

6. Creación de la clase principal de una aplicación Spring Boot y el controlador HTTP
7. Construcción del proyecto a través de la ejecución de la tarea Java *build->build*.
8. Actualización del proyecto para que eclipse detecte las modificaciones realizadas por Gradle (botón derecho sobre el proyecto->Gradle->Refresh Gradle Project)
9. Ejecución del proyecto a través de la tarea de Spring Boot *application->bootRun*.

4

DESARROLLO DE MI PRIMER MICROSERVICIO

Los microservicios permiten crear grandes sistemas software a partir de una serie de componentes que deben interactuar entre ellos de forma colaborativa. Como se ya comentó en el capítulo 2, la interacción con microservicios se implementa a través de API REST. Además, los microservicios deben estar disponibles en algún registro de tal modo que puedan encontrarse entre ellos. Para ello, en este capítulo vamos a ver como construir el API REST de un microservicio con Spring Boot y cómo registrarlo en Eureka. También veremos como crear un servidor de configuración con Spring Cloud Config.

4.1 API RESTFUL CON SPRING BOOT

Crear el API REST de un microservicio resulta bastante sencillo si tomamos como base el ejemplo de aplicación web que construimos con Spring Boot en el capítulo anterior. Lo único que tenemos que hacer es ampliar el controlador con nuevas anotaciones que le indiquen a Spring Boot la información necesaria para construir el API.

Para ilustrar la construcción del API REST vamos a utilizar la aplicación de ejemplo de este libro. Y en concreto, vamos a crear el API REST para el microservicio que permite la gestión del catálogo de productos.

Lo primero que tenemos que hacer a la hora de diseñar una API REST es decidir lo siguiente:

1. *¿Qué recursos vamos a publicar?*

   En este caso, los recursos a publicar son el listado completo de productos y cada uno de los productos de forma individual.

2. *¿Qué operaciones vamos a permitir realizar sobre ellos?*

   Para cada uno de los productos disponibles en el catálogo queremos poder realizar operaciones de consulta y borrado, por tanto, se deberá dar soporte a las operaciones GET y DELETE (ver la sección 2.4.3.1). En cuanto al listado de productos queremos permitir la consulta y la inserción de nuevos productos, por lo que para estos recursos daremos soporte a las operaciones GET y POST.

3. *¿Qué URIS les vamos a asociar?*

   Para el listado completo de productos vamos a utilizar la siguiente URI: **/productos**

   Para cada uno de los productos vamos a utilizar la siguiente URI, donde **{id}** hace referencia al identificador de cada producto: **/productos/{id}**

4. ¿Mediante qué representación (JSON, XML, CSV, etc.) vamos a permitir trabajar con ellos?

En este ejemplo, vamos a trabajar con notación JSON, aunque podríamos utilizar cualquier otra notación o incluso hacer uso de varias.

Una vez diseñada la API, vamos a implementarla mediante Spring Boot. Para ello, vamos a tomar como base la aplicación Web que construimos en el capítulo anterior. En primer lugar, lo único que tenemos que hacer para definir un API REST es marcar el controlador con la anotación **@RestController**, la cual sustituirá a las anotaciones **@Controller** y **@ResponseBody** que habíamos definido inicialmente, ya que las incluye a las dos. A continuación, debemos implementar un método para cada URI/operación/representación que demos soporte, marcándolos con las anotaciones correspondientes. En este caso, tenemos las URI definidas para la lista de productos y los productos individuales. La lista de productos debe aceptar las operaciones GET y POST mientras que los productos las operaciones GET y DELETE. En todos los casos, como ya hemos comentado anteriormente, trabajaremos con una notación JSON. Así pues, necesitaremos cuatro métodos.

Cada uno de los métodos estará marcado con la anotación **@RequestMapping** que nos permite indicar qué URI debe soportar (propiedad **value**), qué operación (propiedad **method**) y en qué formato produce los datos (propiedad **produces**).

Esta última propiedad puede contener tener varios tipos *mime* indicando que el método puede devolver el recurso en varias representaciones. Así pues el cliente que consuma el servicio deberá indicar en la conexión HTTP el encabezado **Accept** con el tipo *mime* de la representación que desea.

En el siguiente ejemplo, tenemos la implementación del método que devuelve el listado de productos. Como se puede apreciar, el método simplemente recupera el listado de productos a partir de una clase que implementa el patrón DAO (Data Access Object) y lo devuelve como un objeto **List**. En este caso, dado que queremos devolver el recurso solicitado en JSON, el programador no debe preocuparse por ello. Spring transformará de forma automática el listado de objetos en una colección JSON. Para ello, los objetos de dominio (clase **Producto** en el ejemplo) deben ser clases POJO (Plain Old Java Objects), es decir, clases con atributos privados, métodos de acceso (setters y getters) y constructor por defecto.

```
@RequestMapping(
   value = "/productos",
   method = RequestMethod.GET,
   produces = "application/json"
)
public List<Producto> listaProductosJSON() {
   return DAO.getProductoDAO().getProductos();
}
```

A continuación, vemos la implementación del método que se encarga de dar acceso a cada uno de los productos disponibles. Como vemos, la especificación es idéntica a la anterior, con la diferencia de que la URI especificada tiene el parámetro **id** definido. Este parámetro nos permite crear un patrón de URI de forma que cualquier producto puede ser accedido a través de la API REST indicando en la URI su id justo después de la cadena "**/productos/**".

Para poder utilizar este parámetro en el cuerpo del método se ha definido el argumento **id**, de tipo **Integer**, y mediante la anotación **@PathVariable** se ha enlazado con el parámetro de la URI.

```
@RequestMapping(
value = "/productos/{id}",
method = RequestMethod.GET,
produces = "application/json"
)
public Producto getProductosJSON(
@PathVariable(value="id") Integer id) {
return DAO.getProductoDAO().getProductoById(id);
}
```

El método que da soporte a la operación DELETE se implementa de forma análoga al anterior, aunque en este caso, dado que no devuelve ningún tipo de dato, no hace falta la definir la propiedad **produces**.

```
@RequestMapping(
value = "/productos/{id}",
method = RequestMethod.DELETE
)
public void delProducto(@PathVariable(value="id") Integer id) {
DAO.getProductoDAO().delProductoById(id);
}
```

Por último, el siguiente ejemplo muestra el método que da soporte a la operación POST sobre la lista de productos. La especificación de la anotación **@RequestMapping** es similar a las anteriores, con la única diferencia de que el método definido hace referencia a la operación en cuestión. En este caso, usamos la propiedad **consumes** para indicar que el cuerpo de la conexión HTTP debe contener un producto en formato JSON. Cuando un usuario consuma esta operación deberá indicar el formato del contenido enviado a través del encabezado **Content-Type**. Hay que destacar también el uso de la anotación **@RequestBody** que nos permite enlazar el cuerpo de la conexión HTTP con el argumento de tipo Producto. En este caso, se espera un producto en formato JSON que Spring transformará de forma automática a un objeto Java **Producto**.

```
@RequestMapping(
   value = "/productos",
   method = RequestMethod.POST,
consumes = "application/json"
)
public void addProducto(@RequestBody Producto p) {
   DAO.getProductoDAO().addProducto(p);
}
```

4.2 REGISTRO DE MICROSERVICIOS CON EUREKA

Los microservicios deben trabajar juntos para alcanzar los requisitos de negocio de un sistema. Para ello, es necesario que cada microservicio sea capaz de encontrar al resto para interactuar con ellos. Aquellos que estén familiarizados con la computación distribuida mediante tecnologías como RMI (*Remote Method Invocation*), sabrán que hace falta registrar los procesos en un registro central de forma que pudieran encontrarse los unos a los otros. En el caso de los microservicios ocurre los mismo.

Para que los microservicios puedan interactuar entre ellos, es necesario publicarlos en un registro central de forma que se puedan encontrar entre ellos.

Los desarrolladores de Netflix tuvieron este problema al construir sus sistemas y crearon un servidor de registro llamado Eureka (que significa "Lo he encontrado" en griego). Afortunadamente para todos, lo publicaron como código abierto y Spring lo incorporó a Spring Cloud, proporcionando una forma fácil y sencilla de crear y ejecutar un servidor Eureka.

Eureka es un servidor de registro que ofrece funcionalidades de localización de microservicios y que, combinado con Zuul y Hystrix, respectivamente, soporta también el balanceo de carga y la tolerancia a fallos.

La Figura 4.1 muestra el funcionamiento básico de Eureka Server. Cuando arranca un microservicio, se comunica con Eureka Server y registra en él toda la información relativa a la prestación del servicio: ubicación, disponibilidad, metadatos, etc. Esta información de estado se notifica al servidor cada 30' (periodo de latido o *heartbeat*), de manera que, si después de tres latidos el servidor no recibe información de un microservicio, es eliminado del registro y se quedará fuera hasta que reciba tres nuevos latidos consecutivos. De esta manera, se minimiza el impacto de un fallo por la no disponibilidad de un microservicio. Por otra parte, los clientes de los microservicios (sean otros microservicios o una aplicación final), utilizan el cliente de Eureka para comunicarse con el servidor de registro y cachean la información sobre los microservicios registrados.

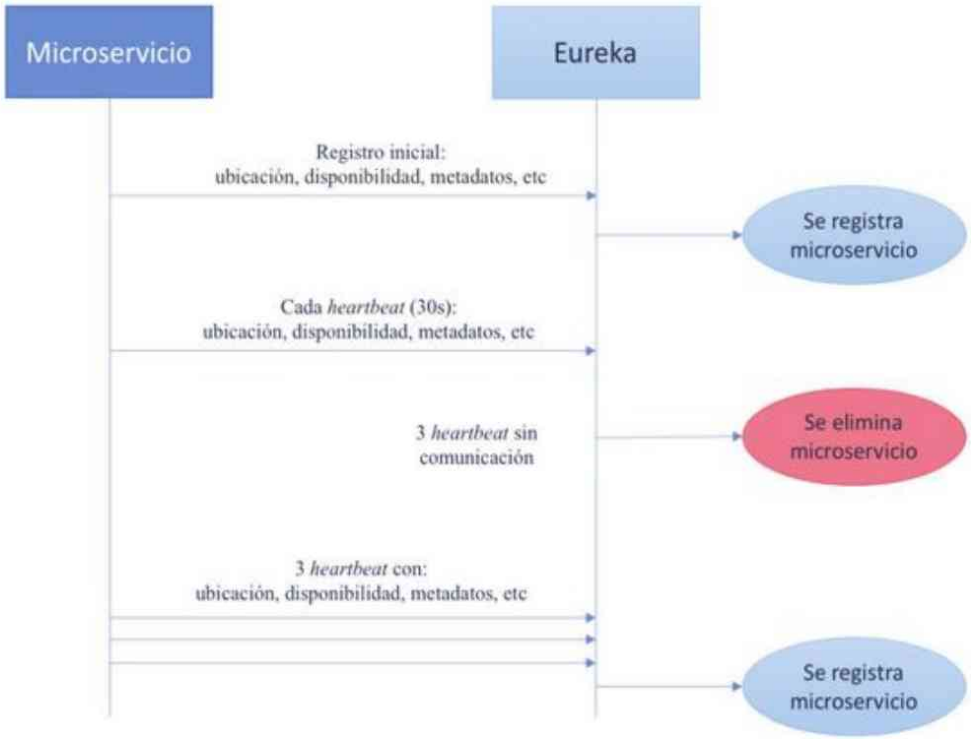

Figura 4.1. Funcionamiento básico del proceso de registro con Eureka

4.2.1 Creación del servidor Eureka

Un servidor Eureka no es más que una aplicación Spring Boot que incluye la anotación **@EnableEurekaServer** tal y como se muestra en el siguiente código. Esta aplicación la crearemos dentro de un nuevo proyecto Gradle tal y como hicimos anteriormente con los microservicios.

```
package application;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.
       eureka.server.EnableEurekaServer;
@SpringBootApplication
@EnableEurekaServer
public class ServidorEureka {
  public static void main(String[] args) {
    SpringApplication.run(ServidorEureka.class, args);
  }
}
```

En este caso, la configuración que necesitamos para el fichero *build.gradle* difiere de la utilizada en el desarrollo de microservicios (ver capítulo anterior) en cuanto a la gestión de las dependencias. En este caso, en lugar de importar el *starter* para aplicaciones web necesitamos importar el de un servidor eureka. La configuración quedará de la siguiente forma:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle
plugin:1.5.9.RELEASE")
    }
}
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'
apply plugin: 'java'
repositories {
    mavenCentral()
}
dependencyManagement {
    imports {
   mavenBom 'org.springframework.cloud:spring-cloud
```

```
        dependencies:Edgware.RELEASE'
            }
        }
        dependencies {
            compile 'org.springframework.cloud:spring-cloud-starter-eureka
        server'
        }
        jar{
            manifest {
               attributes 'Main-Class': 'application.EurekaServer'
            }
            baseName = 'es.rama.books '
            version =  '0.1.0'
        }
```

Esta nueva dependencia con el servidor eureka, a su vez, necesita de otras dependencias que se nos proporcionan a través de un archivo BOM (*Bill of Materials*). Este tipo de archivos son utilizados por Maven para el control de versiones de las dependencias de un proyecto, proporcionando un lugar para definir y actualizar estas versiones. Muchos proyectos de Spring proporcionan sus dependencias a través de ficheros BOM. Afortunadamente, Gradle permite la integración con estos ficheros. Para ello, basta con definir la cláusula **dependencyManagement** e indicar el grupo, nombre y versión del BOM, tal y como vemos en el ejemplo anterior.

Por último, necesitamos crear un archivo de configuración del servidor Eureka. Eureka puede configurarse como un único servidor (*stand-alone*) o, en entornos más complejos, como una agrupación (*cluster*) de servidores, donde varias instancias se conectan entre sí e intercambian información de registro entre ellas. Esta configuración *cluster* es la que se utiliza en grandes entornos de producción y, por tanto, queda fuera del ámbito de este libro.

La configuración *stand-alone* de Eureka consiste en un servidor autónomo que gestiona todas las tareas relacionadas con el registro de microservicios. La configuración del servidor se basa en un fichero de propiedades (*.properties*) o con formato YML, que es lo más usual donde se detallan los parámetros de configuración del servicio de registro. Por defecto, el servidor busca el fichero *application.yml* (o *.properties*) el cual debe estar ubicado en la raíz del proyecto (como el fichero *build.gradle*).

```
        server:
          port: 8761
        eureka:
          client:
```

```
    registerWithEureka: false
    fetchRegistry: false
    server:
      waitTimeInMsWhenSyncEmpty: 0
```

En el ejemplo anterior estamos indicando que el puerto en el que el servidor de registro escuchará las peticiones será el 8761 (valor por defecto). Si no indicamos nada, el servidor trata de registrase a sí mismo como un microservicio. Esto es útil cuando se utiliza una configuración por *clusters*, donde existen varios servidores que deben comunicarse entre sí. De momento, le decimos que no lo haga (**registerWithEureka: false**). Tampoco necesita cachear en local la información proporcionada por el servidor ya que se trata del propio servidor (**fetchRegistry: false**).

4.2.2 Registro de un microservicio

Una vez hemos construido el servidor Eureka tenemos que indicar a nuestros microservicios que se registren en él. Para ello, tenemos que actualizar el fichero **build.gradle** para que incluya como dependencia el *starter* de Spring Boot para Eureka en lugar del utilizado para aplicaciones web, tal y como hemos hecho con el servidor Eureka. Tras realizar esta actualización, la configuración del fichero quedará de la siguiente forma:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle
plugin:1.5.3.RELEASE")
    }
}
apply plugin: 'eclipse'
apply plugin: 'java'
apply plugin: 'org.springframework.boot'
repositories {
   mavenCentral()
}
dependencyManagement{
   imports{
      mavenBom 'org.springframework.cloud:spring-cloud
dependencies:Edgware.RELEASE'
```

```
    }
}
dependencies {
   compile 'org.springframework.cloud:spring-cloud-starter-eureka'
}
jar{
    manifest {
        attributes 'Main-Class': 'application.MainApp'
    }
    baseName = 'es.rama.books '
    version =  '0.1.0'
}
```

A continuación, debemos añadir la anotación **@EnableDiscoveryClient** a la clase principal de nuestro microservicio, convirtiéndolo así en un cliente Eureka.

```
@EnableDiscoveryClient
@SpringBootApplication
public class MainApp {
public static void main(String[] args) {
      SpringApplication.run(MainApp.class, args);
}
}
```

Por último, debemos añadir un fichero de configuración YML, tal y como hicimos con el servidor. En el ejemplo que se muestra a continuación, estamos indicando el nombre con el que se registrará el microservicio (**servicio.catalogo**), la url del servidor Eureka (aquí tenemos que poner el puerto definido en la configuración del servidor) y el puerto en el que estará escuchando el microservicio (**1111**).

```
#Spring configuration
spring:
  application:
     name: servicio.catalogo
# HTTP Server
server:
  port: 1111
# Discovery Server Access
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka
```

4.2.3 Resumen del registro de microservicios

A continuación resumimos los pasos que hemos descrito en las secciones anteriores para crear el servidor Eureka y registrar servicios en él.

Creación del servidor

1. Crear de una aplicación Spring Boot con la anotación **@EnableEurekaServer.**
2. Importar el *starter* de Eureka en las dependencias del fichero *build.gradle*.
3. Confugrar el servidor Eureka a través del fichero *application.yml.*
4. Ejecutar el servidor

Registro de un microservicio

5. En el archivo *build.gradle*, reemplazar el starter web por el starter de Eureka.
6. Marcar la aplicación principal con la anotación **@EnableDiscoveryClient.**
7. Configurar el microservicio y su conexión con Eureka en el fichero *application.yml.*
8. Ejecutar el microservicio

4.3 CONSUMO DE UN MICROSERVICIO

Una vez hemos desarrollado el servidor Eureka y los microservicios que se registran en él, podemos ejecutarlos a través de la tarea bootRun de Gradle. El único detalle a tener en cuenta es que debemos ejecutar el servidor antes que los microservicios.

Una vez tenemos en ejecución tanto el servidor como los microservicios, podemos acceder al servidor Eureka a través del navegador (en este caso, **http://localhost:8761**) y ver los microservicios registrados, tal y como se muestra en la siguiente figura.

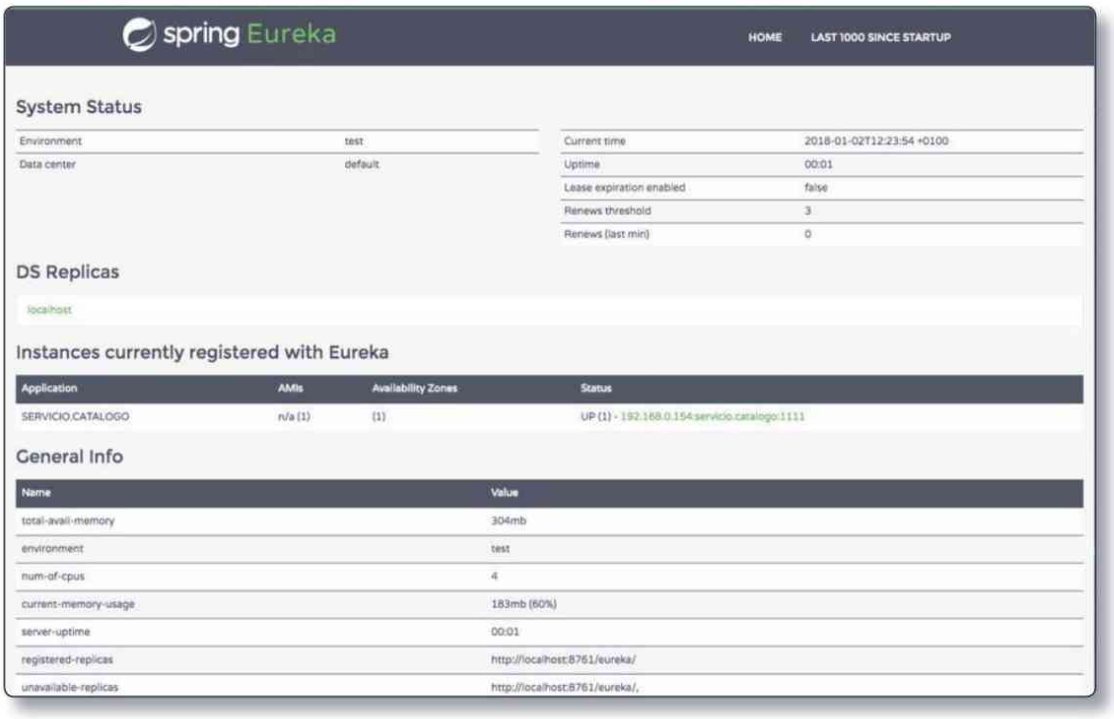

Figura 4.2. Servidor Eureka

Sin embargo, la principal utilidad del servidor Eureka es permitir ejecutar microservicios sin necesidad de conocer su datos de invocación (url, puerto, etc.). Basta con conocer su identificador. Para ello, la aplicación principal o cualquier microservicio que quiera interactuar con otros debe incluir en su archivo *build.gradle* la dependencia de Eureka, tal y como hicimos anteriormente (si se trata de un microservicio ya la tendremos incluida al ser necesario para su registro en eureka; si se trata de una aplicación web deberemos añadirla junto a las dependencias propias del desarrollo web).

Una vez tenemos las dependencias necesarias incluidas, podemos hacer uso de la clase **DiscoveryClient** proporcionada por Spring. A continuación vemos un ejemplo de cómo buscar en el servidor Eureka un microservicio por su nombre.

```
import java.net.URI;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.ServiceInstance;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.stereotype.Component;
@Component(“EurekaClient”)
public class EurekaClient{
    @Autowired
```

```
    private DiscoveryClient discoveryClient;
    public URI getUri(String serviceID) {
      List<ServiceInstance> list =
discoveryClient.getInstances(serviceID);
        if (list != null && list.size() > 0 ) {
            return list.get(0).getUri();
        }
        return null;
    }
}
```

En el ejemplo anterior hemos creado una clase que actuará como cliente de un servidor Eureka. Esta clase dispone de un atributo de tipo **DiscoveryClient** que es inyectado automáticamente por Spring (anotación **@Autowired**). El método **getURI** hace uso de este atributo para solicitar al servidor Eureka todas las instancias del microservicio que tenga asociado el identificador que recibe por parámetro. Dado que en el ejemplo que estamos desarrollando hasta ahora sólo tenemos una instancia de cada microservicio (en el siguiente capítulo veremos cómo usar Ribbon para balancear la carga entre múltiples instancias), nos quedamos con la primera y única que nos devuelve, y devolvemos su URI que contiene información para su invocación como el servidor o el puerto. Esta información la podemos utilizar con cualquier cliente HTTP.

En el siguiente ejemplo vemos como se usa la clase anterior en un método del microservicio que gestiona la cesta de la compra. En concreto, está solicitando el catálogo de productos al microservicio desarrollado anteriormente. La conexión HTTP se realiza a través de la clase de Spring **RestTemplate**. Spring no crea un *bean* de esta clase, por lo que no se puede inyectar de forma automática. En su lugar, dispone del *bean* **RestTemplateBuilde**r que se encarga de crear objetos **RestTemplate**. Así pues, el atributo **restTemplate** es instanciado por Spring llamando al método del mismo nombre declarado como **@Bean**. Cuando Spring llama a este método inyecta el argumento que necesita para ejecutar su código y crear el objeto **RestTemplat**e (a través de la línea de código **builder.build()**).

```
@Autowired
private EurekaClient eureka;

@Autowired
private RestTemplate restTemplate;
@Bean
public RestTemplate restTemplate(RestTemplateBuilder builder) {
 return builder.build();
}
```

```
@RequestMapping(
value = "/cesta/producto/{id}",
   method = RequestMethod.POST
)
public void addProductoCesta(@PathVariable(value="id") Integer id)
throws IOException, JSONException {

   URI catalogoURI=eureka.getUri("SERVICIO.CATALOGO");
   Producto p=restTemplate.getForObject(
catalogoURI.resolve("/productos/"+id),
Producto.class);
   DAO.getCestaDAO().addProducto(p);
}
```

4.4 CONFIGURACIÓN AVANZADA DE EUREKA

Además de la información típica sobre la IP, el puerto y el nombre del servicio que hemos configurado anteriormente, Eureka puede registrar información adicional del mismo. Por ejemplo, las siguientes dos propiedades nos permiten configurar los puntos de acceso a páginas de información sobre un microservicio:

- **statusPageUrlPath**: establece el punto final de información sobre el microservicio. Por defecto, tiene el valor **/info**.
- **healthCheckUrlPath**: indica el punto final de los indicadores de salud de la instancia de Eureka. Por defecto, el valor es **/health**. Obtendremos un objeto JSON con diferentes indicadores, entre los que destaca el estado de la instancia Eureka, que puede tener los siguientes valores: **UP**, **DOWN**, **STARTING**, **OUT_OF_SERVICE**, **UNKNOWN**.

Los enlaces anteriores aparecen en los metadatos que consumen los clientes y se utilizan en algunos escenarios para decidir si se envían solicitudes o no, por lo que resulta útil definirlos de forma precisa.

Otras propiedades son:

- **nonSecurePortEnabled, securePortEnabled**: permiten establecer si trabaja con conexión segura. Los valores que admiten son true o **false**. Si definimos la primera propiedad a **false** y la segunda a **true** Eureka publicará información de la instancia que muestre una preferencia explícita por la comunicación segura. La clase de Spring Cloud **DiscoveryClient** siempre

devolverá una URI que comience con https para un servicio configurado de esta manera, y la información de la instancia Eureka (nativa) tendrá una URL segura de verificación de estado.

- **initialStatus**: estado inicial que será enviado al servidor Eureka en la primera notificación del registro de manera que podemos registrar microservicios sin que estén inmediatamente disponibles.
- **eureka.instance.metadataMap**: en este mapa se registran metadatos de la aplicación como, por ejemplo, el propietario (**appOwner**) o su descripción (**description**).

Todas las propiedades se definen en el fichero de configuración YML, dentro del bloque **instance** contenido en el bloque **eureka**. En el siguiente ejemplo vemos la configuración definida para el microservicio de catálogo.

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:2222/eureka
  instance:
    statusPageUrlPath: /infoCatalogo
    healthCheckUrlPath: /healthCatalogo
    metadataMap:
      appOwner: valderasroldan
      description: aplicación ejemplo microservicio
```

Si en lugar de utilizar un fichero YML optamos por un fichero de propiedades, todas ellas deben estar precedías por el prefijo **eureka.instance.**

La lista completa de las propiedades que se pueden configurar se puede consultar en las clases **EurekaInstanceConfigBean** y **EurekaClientConfigBean**, incluidas ambas en el paquete **org.springframework.cloud.netflix.eureka.** En la documentación de referencia es posible encontrar una descripción de las mismas.

Podemos comprobar las propiedades definidas para los microservicios registrados en Eureka en **http://localhost:8761/eureka/apps**. Otra alternativa es añadir el nombre del microservicio a la url anterior, por ejemplo, para el microservicio del catálogo, la URL de información sería **http://localhost:8761/eureka/apps/servicio.catalogo**.

4.5 CONFIGURACIÓN EN LA NUBE

Hasta ahora, hemos creado la configuración de cada microservicio de forma local al propio servicio (archivo YML incluido en su implementación). Sin embargo, en una arquitectura de microservicios es común tener varias instancias de un mismo microservicio en diferentes máquinas (por ejemplo, podemos tener varias instancias del microservicio que gestiona el catálogo como medida de seguridad ante caídas en la red). En esta situación, si cambiamos la configuración del microservicio deberíamos hacerlo en todas y cada una de las instancias existentes de dicho microservicio.

Una mejor solución consiste en almacenar los archivos de configuración de cada servicio en la nube, y crear un servidor que permita a las diferentes instancias de los servicios, y al propio servidor Eureka, acceder a ellos de forma remota. Para ello, Spring nos proporciona Spring Cloud Config. Se trata de una solución en entornos distribuidos para la configuración externa y proporciona tanto la parte de cliente como la del servidor, de manera que se dispone de un punto central desde el que se accede a la información de configuración de los microservicios.

La información de configuración de los microservicios se almacena en un repositorio Git cuya información se lee cuando el servidor de configuración arranca. La principal ventaja de un repositorio de configuración basado en Git es que proporciona un histórico de la configuración. Cuando un microservicio arranca, se conecta al servidor de configuración para consultar las propiedades asociadas, tal y como muestra la Figura 4.3. Todo este proceso es totalmente transparente al desarrollador y se gestiona mediante el uso de anotaciones.

Figura 4.3. Utilización del servidor de configuración

Construir un servidor de configuración es muy sencillo gracias a la anotación **@EnableConfigServer**. Esto convierte a cualquier aplicación Spring Boot en un servidor de este tipo. Para hacer uso de esta anotación debemos incluir en el fichero *build.gradle* la dependencia **spring-cloud-config-server**. Para ello, el fichero *build.gradle* debe ser idéntico al que hemos creado en el servidor Eureka o los microservicios, pero con la siguiente sección de dependencias:

```
dependencies {
    compile 'org.springframework.cloud:spring-cloud-config-server'
}
```

Una vez incluida la dependencia en el fichero *build.gradle*, crear un servidor de configuración es tan sencillo como muestra el siguiente fragmento de código:

```
@EnableConfigServer
@SpringBootApplication
public class ConfigServer {
   public static void main(String[] args) {
      SpringApplication.run(ConfigServer.class, args);
   }
}
```

La URI del repositorio de configuración se especifica en un fichero *application.{yml|properties}*, tal y como se muestra a continuación:

```
spring:
  cloud:
    config:
      server:
        git:
          uri:
https://github.com/pvalderas/es.books.microservices.serverconfig
```

En el servidor Git deben estar disponibles en un repositorio los archivos de configuración de todos los clientes (archivos YML o properties). El nombre de estos archivos debe coincidir con el nombre de aplicación Spring que le demos a cada cliente (tal y como veremos en la siguiente subsección). A modo de ejemplo, la siguiente figura muestra el servidor GitHub de la aplicación ejemplo. En este caso, existen tres clientes a los que el servidor puede proporcionar los archivos de configuración. Estos clientes tendrán el nombre de aplicación Spring **eureka-server**, **servicio.catalogo** y **servicio.cesta**.

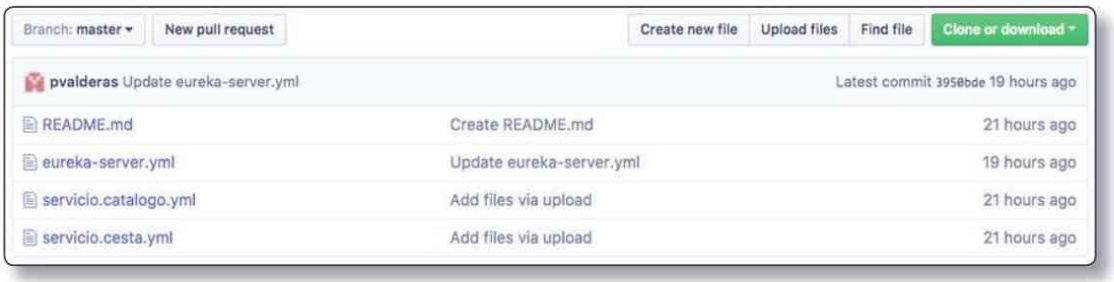

Figura 4.4. Servidor GitHub con los archivos de configuración

En ocasiones, por ejemplo, si se quiere probar en local o no se dispone de conexión a Internet por cualquier otro motivo, es posible especificar una ruta con el prefijo `file` en lugar de `uri`. También es posible trabajar en local (se logra el mismo resultado que usando la propiedad **file**) si se activa el perfil nativo de Spring:

```
spring:
  profiles:
    active: native
```

Por defecto, el servidor de configuraciones espera conexiones en el puerto 8888. Sin embargo, podemos cambiarlo tal y como ya lo hicimos anteriormente.

```
server:
  port: 4444
```

4.5.1 Configuración de los clientes del servidor de configuración

En la aplicación que estamos desarrollando como ejemplo vamos a tener tres clientes del servidor de configuración: el servidor Eureka, y los dos microservicios desarrollados (catalogo y cesta).

Lo primero que tenemos que hacer para trabajar con el servidor de configuraciones es subir los archivos YML al servidor GitHub, tal y como aparece en la figura 4.4. Estos ficheros contendrán la configuración relacionada con Eureka y los datos de conexión al servicio, que definimos anteriormente en el fichero *application.yml*. Por ejemplo, a continuación se muestra el contenido del fichero *servicio.catalogo.yml* disponible en GitHub.

```
# Conexión Eureka
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:2222/eureka
# Conexión HTTP
server:
  port: 1111
```

Al subir el fichero YML al servidor GitHub, el *application.yml* creado en los microservicios y el servidor Eureka ya no son necesario, puesto que se lo solicitan al servidor de configuraciones el cual se lo descarga de Github. Sin embargo, es usual que cada servicio mantenga en local una versión del mismo por si ocurre algún fallo en la conexión con el servidor. Si todo funciona bien, el microservicio se configurará

según el fichero YML proporcionado por el servidor. En caso de error, usará la versión que tiene en local.

Para hacer que un microservicio o el propio servidor Eureka soliciten su configuración al servidor de configuraciones tenemos que indicárselo en el archivo *bootstrap.yml*. Este archivo es similar al *application.yml*. La única diferencia reside en el momento en el que una aplicación Spring Boot accede a él. El fichero *bootstrap.yml* se lee justo antes de aplicar una configuración, de modo que nos permite indicarle de dónde debe obtenerla. En cambio, el archivo *application.yml* define la configuración en sí, de modo que se lee después, cuando se configura la aplicación.

El siguiente ejemplo muestra el contenido del archivo *bootstrap.yml* del microservicio que gestiona el catálogo. Reseñar que este archivo debe mantenerse en local para cada microservicio, colgando de la raíz del proyecto, tal y como hacíamos con el *application.yml*.

```
spring:
  application:
     name: servicio.catalogo
  cloud:
    config:
      uri: http://localhost:4444
```

En el fichero anterior vemos que solo necesitamos indicar dos datos para que un microservicio pueda interactuar con el servidor de configuraciones: el nombre de la aplicación Spring (que debe coincidir con el nombre del fichero de configuración subido al servidor Git) y la URI donde está esperando peticiones el servidor de configuraciones.

Por último, para que este fichero sea interpretado de forma adecuada necesitamos actualizar el fichero *build.grade* de los microservicios y del servidor Eureka para incluir la dependencia de Spring Cloud Config, tal y como vemos a continuación.

```
dependencies {
    compile 'org.springframework.cloud:spring-cloud-starter-config'
    compile 'org.springframework.cloud:spring-cloud-starter-eureka'
}
```

A modo de resumen, enumeramos los pasos que debemos realizar para hacer que los microservicios y el servidor Eureka que habíamos creado hasta ahora, se conviertan en clientes del servidor de configuraciones:

1. Crear el fichero de configuración YML y subirlo al servidor Git. El fichero debe tener el mismo nombre que le demos a la aplicación en el fichero *bootstrap.yml*.
2. Crear el fichero *bootstrap.yml* donde indicamos el nombre de la aplicación y la URI del servidor de configuraciones.
3. Incluir la dependencia de Spring Cloud Config en el archivo *build.gradle*.

4.5.2 Interacción con el servidor de configuraciones

Al arrancar el servidor de configuraciones, éste crea diferentes puntos finales que nos permiten interactuar con él, de forma que podemos pararlo, reiniciarlo, poner en pausa, etc.

Estos puntos finales aparecen en la consola (en nuestro caso de Eclipse) cuando lanzamos el servidor. En concreto, podemos observar que se crean los siguientes:

- **/refresh**: refresco de los cambios de configuración
- **/env**: muestra las variables de configuración disponibles en el servidor
- **/shutdown**: apaga el servidor de configuración
- **/restart**: reinicia el servidor de configuración
- **/trace**: devuelve el último par petición/respuesta
- **/pause**: invoca al método *stop* del ciclo de vida del ApplicationContext de Spring
- **/info**: proporciona información sobre el servidor de configuración
- **/health**: indica el estado de la aplicación
- **/resume**: llama al método *start* del ciclo de vida del ApplicationContext de Spring
- **/metrics**: ofrece métricas como la memoria, procesador, carga de trabajo, etc.
- **/dump**: realiza un *dump* del hilo de ejecución

- **/heapdum**: hace un volcado de la información de la pila con el fin de facilitar la detección de fallas de memoria, por ejemplo.
- **/beans**: lista todos los *beans* de Spring cargados en el sistema.
- **/mappings**: lista todos los mapeos de Spring que se hayan cargado en el sistema.

4.5.3 Seguridad

Un aspecto importante en todo tipo de aplicaciones y, quizá todavía más en las distribuidas, es la seguridad. Si un *hacker* fuera capaz de controlar el servidor de configuración nos enfrentaríamos a un verdadero desastre. Para evitar situaciones como ésta, es necesario añadir una capa de seguridad al servidor de configuración.

El primer paso será añadir en el fichero *build.gradle*. del servidor de configuración la siguiente dependencia:

```
dependencies {
   compile 'org.springframework.cloud:spring-cloud-config-server'
   compile 'org.springframework.boot:spring-boot-starter-security'
}
```

Con solo añadir esta dependencia, ya es posible utilizar el esquema de seguridad HTTP básico de Spring Boot, con un usuario por defecto llamado **user** y una contraseña generada aleatoriamente y que se imprime en consola durante el arranque, tal y como vemos en la siguiente figura.

```
2018-01-11 14:00:33.570  INFO 40799 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMappi
2018-01-11 14:00:33.571  INFO 40799 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMappi
2018-01-11 14:00:33.603  INFO 40799 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMappin
2018-01-11 14:00:33.603  INFO 40799 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMappin
2018-01-11 14:00:33.651  INFO 40799 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMappin
2018-01-11 14:00:33.924  INFO 40799 --- [           main] b.a.s.AuthenticationManagerConfiguratio

Using default security password: 346be492-a782-4f4d-9a86-b4c0250795d8

2018-01-11 14:00:33.975  INFO 40799 --- [           main] o.s.s.web.DefaultSecurityFilterChain
2018-01-11 14:00:34.544  INFO 40799 --- [           main] o.s.b.a.e.mvc.EndpointHandlerMapping
2018-01-11 14:00:34.545  INFO 40799 --- [           main] o.s.b.a.e.mvc.EndpointHandlerMapping
2018-01-11 14:00:34.546  INFO 40799 --- [           main] o.s.b.a.e.mvc.EndpointHandlerMapping
2018-01-11 14:00:34.547  INFO 40799 --- [           main] o.s.b.a.e.mvc.EndpointHandlerMapping
2018-01-11 14:00:34.548  INFO 40799 --- [           main] o.s.b.a.e.mvc.EndpointHandlerMapping
```

Figura 4.5. Contraseña generada por defecto

Para cambiar el usuario y contraseña por defecto, basta con indicarlo en el archivo *application.yml* del servidor de configuración.

```
security:
  basic:
    enabled: true
  user:
    name: admin
    password: mipassword
```

Si el servidor de configuración tiene definida la capa de seguridad, los clientes deben conectarse indicando el usuario y contraseña que se requiera. Para ello, deben indicarlo en el fichero *bootstrap.yml* tal y como se muestra a continuación.

```
spring:
  application:
    name: eureka-server
  cloud:
    config:
      uri: http://localhost:4444
      username: admin
      password: mipassword
```

En el ejemplo anterior, el cliente define la contraseña como texto planto. Del mismo modo, si es necesario indicar algún dato sensible en los ficheros disponibles en el servidor Git tenemos que hacerlo en texto plano. Para solventar este problema, Spring Cloud admite cifrado y descifrado de valores. Los valores encriptados son prefijados con la cadena **{cipher}** y se descrifrarán antes de ser enviados por HTTP. La principal ventaja de esta configuración es que los valores de las propiedades no tienen que estar en texto plano (por ejemplo, en el repositorio Git). Si el valor no se pude descifrar, se borra del fichero de configuración y se añade una propiedad adicional con la misma clave, pero prefijada con un **invalid** y un valor que significa no aplicable (generalmente, "**<n/a>**"), lo que evita que se cifren accidentalmente textos que están siendo utilizados. Los valores cifrados tienen que ir, obligatoriamente, encerrados entre comillas simples o de lo contrario no se descifrarán. En el siguiente ejemplo, se muestra cómo se indicaría el usuario y password del ejemplo anterior mediante el uso de cifrado.

```
spring:
  cloud:
    config:
      uri: http://localhost:4444
      username: admin
      password:
'{cipher}baa9a591edb3925706ccc5b2144758cc8ed7e7bc05bfc2fa6f
57210f9688c44'
```

De esta manera, evitamos definir contraseñas en texto plano, importante sobre todo en los ficheros subidos al servidor Git. Para poder utilizar esta posibilidad, necesitamos tener instalada la extensión JCE[11](*Java Cryptography Extension*). Además, debemos definir la clave de encriptación en el fichero *bootstrap.properties* del servidor de configuración o del microservicio que defina una contraseña encriptada en un archivo local, tal y como se muestra a continuación:

```
encrypt.key:1234567890
```

Por otra parte, el servidor expone dos puntos finales para cifrar **(/encrypt**) y descifrar (**/decrypt**). Ambos puntos finales esperan conexiones HTTP a través de la operación POST. En el cuerpo de la conexión debemos indicar la clave a encriptar o desencriptar. Si usamos una herramienta como **curl**, la llamada para encriptar una contraseña sería como sigue:

```
$ curl localhost:4444/encrypt -d mipassword
baa9a591edb3925706ccc5b2144758cc8ed7e7bc05bfc2fa6fa57210f9688c44
```

La operación inversa también está disponible:

```
$ curl localhost:4444/decrypt -d
baa9a591edb3925706ccc5b2144758cc8ed7e7bc05bfc2fa6fa57210f9688c44
mipassword
```

Así pues, si queremos usar contraseñas encriptadas, podemos ejecutar el servidor de configuración con las JCE instaladas y la clave de encriptación definida en el archivo *bootstrap.propeties* . Con ello en marcha, podemos usar el punto final **/encrypt** para generar la clave cifrada.

11 http://www.oracle.com/technetwork/java/javase/downloads/jce-7-download-432124.html

5

BALANCEO DE CARGA, TOLERANCIA A FALLOS, Y REDIRECCIONAMINTOS

En el capítulo anterior hemos visto cómo desarrollar microservicios con Spring Boot y cómo crear una arquitectura distribuida mediante el servidor de Eureka y un servidor de configuración. En este capítulo vamos a ver cómo integrar la arquitectura que hemos construido con tres nuevos elementos: Ribbon, el cual nos va a permitir balancear la carga de las conexiones HTTP; Hytrix, el cual nos va a permitir considerar la tolerancia a fallos en la llamada a microservicios; y por último Zuul, un Gateway para el redireccionamiento de invocaciones.

5.1 RIBBON

Ribbon es la librería incluida en la solución tecnológica de Netflix OSS, que se integra con Eureka para realizar el balanceo de carga en el lado del cliente (microservicios).

Como ya hemos visto, Eureka permite crear un registro central donde se enumeran todas las instancias de los microservicios disponibles. Ribbon se encuentra en la máquina del cliente (microservicio) y permite solicitar a Eureka las URL de las instancias de un microservicio (por ejemplo, todas las del microservicio que gestiona el catálogo de productos) y seleccionar una de ellas. Para ello, monitorea la respuesta de las URL de las instancias de un servicio proporcionadas por Eureka y se conecta al sitio que ha proporcionado una mejor experiencia en el pasado.

Básicamente, hace uso de dos elementos a la hora de decidir cuál será la instancia a que se reenviará una petición. En primer lugar, se utilizan filtros para eliminar del conjunto de instancias aquellas que estén caídas o se encuentren en una

determinada zona, por ejemplo. El siguiente paso es determinar cuál de las instancias que cumplen todos los criterios deseados, resolverá la petición.

Para ejemplificar el uso de Ribbon vamos a suponer que tenemos disponibles dos instancias del microservicio Catálogo que proporciona acceso al catálogo de productos y otra instancia del microservicio Cesta, que gestiona la cesta, y como ya hemos visto en otros ejemplos, accede al microservicio Catálogo para recuperar información sobre productos.

En esta situación utilizaremos Ribbon como librería en el microservicio Cesta para balancear la carga entre las dos instancias de Catalogo. Además de esto, lógicamente contaremos con nuestro servidor Eureka para el registro de microservicios. La siguiente Figura muestran los microservicios presentes y la comunicación entre los mismos:

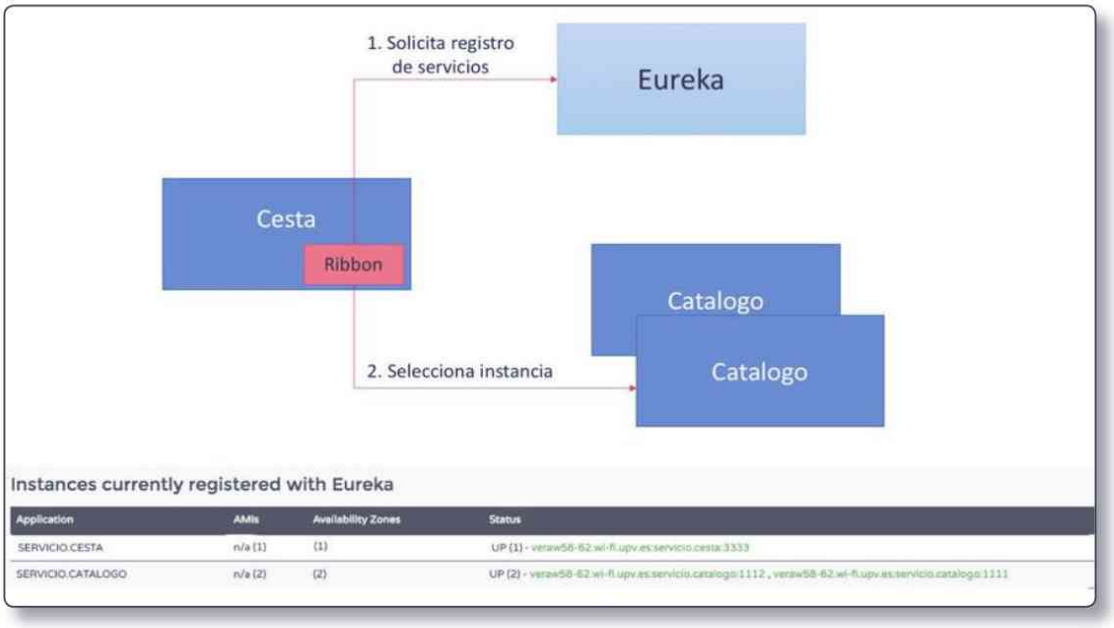

Figura 5.1. Ejemplo del libro con múltiples instancias

Cuando desde Cesta se realice una petición a Catálogo lo que ocurre es lo siguiente:

1. Ribbon solicita a Eureka el registro de microservicios.
2. Ribbon con la información del registro disponible utiliza su balanceador y reglas de balanceo para decidir a qué instancia de Catálogo enviar la petición.

Incluir Ribbon en nuestros microservicios es realmente sencillo ya que al añadir la dependencia **spring-cloud-starter-eureka** (que hemos añadido para permitir que un microservicio interactúe con Eureka), ésta incluirá las librerías de Ribbon.

Por otro lado, si recordamos el código que utilizamos para acceder al microservicio Catálogo desde Cesta (ver sección 4.3), hacíamos uso de un *bean* **RestTemplate** que era inyectado por Spring. Este bean era el que utilizábamos para solicitar el acceso al microservicio. Para que este acceso sea balanceado por Ribbon, basta con marcar el bean con la anotación **@LoadBalanced**, tal y como vemos a continuación.

```
@Bean
@LoadBalanced
public RestTemplate restTemplate(RestTemplateBuilder builder) {
 return builder.build();
}
```

Además, debemos modificar ligeramente la forma en la que invocamos al microservicio. Ahora, podemos acceder directamente a partir del nombre del mismo, tal y como se muestra a continuación.

```
@RequestMapping(
value = "/cesta/producto/{id}",
   method = RequestMethod.POST
)
public void addProductoCesta(@PathVariable(value="id") Integer id)
throws IOException, JSONException {

 Producto p=
 restTemplate.getForObject("http://SERVICIO.CATALOGO/
productos/"+id,
                          Producto.class);
 DAO.getCestaDAO().addProducto(p);
}
```

Para decidir qué instancia se encargará de una petición, Ribbon utiliza dos elementos: los balanceadores (con sus filtros asociados) y las reglas. En primer lugar, Ribbon utilizará un balanceador con sus filtros para descartar una serie de instancias del microservicio en base a diversos criterios (que las instancias estén caídas, que estén en una zona con una alta carga de peticiones, etc.). A continuación, Ribbon aplica las reglas de balanceo sobre las instancias que han pasado los filtros para seleccionar una de ellas.

5.1.1 El balanceador y la lógica de zonas

El balanceador es implementado por la clase **ZoneAwareLoadBalancer.** Este balanceador utiliza una lógica de zonas para determinar en qué zona o zonas son válidas para enviar peticiones. Una vez escogidas las zonas se aplicará las reglas de balanceo entre todas las instancias que componen dichas zonas. La selección de zonas se realiza descartando aquellas que no cumplan unas condiciones mínimas (carga, número de instancias, etc.) y en intentar fomentar la afinidad de zona, es decir, escoger la zona en la que se encuentra la instancia que va a realizar la petición. La lógica de selección de zonas permite definir una configuración en *cluster*, utilizada como ya hemos comentado anteriormente en grandes entornos de producción y que no vamos a tratar con detalle en este libro. Sin embargo, vamos a perfilar a groso modo los aspectos básicos de este tipo de configuraciones.

La figura 5.2 muestra el esquema de una arquitectura por zonas. Lo primero que necesitamos saber para desplegar una arquitectura de zonas, es que:

- Cada zona debe tener un servidor Eureka
- Cada servidor Eureka debe conocer al resto de servidores
- Cada microservicio debe estar asociado a una zona

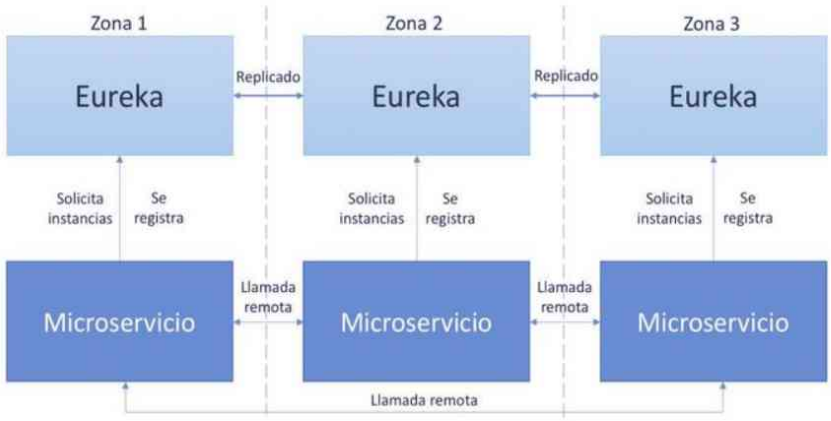

Figura 5.2. Arquitectura por zonas

La zona de un microservicio y del servidor Eureka se establece automáticamente a partir del nombre del host donde está desplegado, si definimos la propiedad **ribbon.eureka.approximateZoneFromHostname** a **true**. También puede definirse de forma explícita mediante el metadato **eureka.instance.metadataMap.zone**. Para que un servidor Eureka conozca al resto de servidores,

éste debe actuar como cliente. Si recordamos la configuración realizada del servidor Eureka en capítulos anteriores, definíamos la propiedad **registerWithEureka** a **false**. Esto lo hacíamos porque Eureka puede actuar tanto de servidor como de cliente de otro servidor Eureka, y al trabajar con una configuración *standalone* (sin *cluster*) no era necesario que fuera cliente. Al trabajar con zonas, los servidores Eureka deben conectarse entre ellos, y port tanto, además de servidores deben ser clientes, configurando la propiedad **defaultZone** tal y como hicimos con los microservicios.

Para la selección de una zona que permita satisfacer las solicitudes de un microservicio, el balanceador realiza dos tareas básicas:

1. Cada 30 segundos ejecuta el filtro **ZonePreferenceServerListFilter** que se encarga de seleccionar las zonas en las que solicitar al servidor Eureka instancias de servicios, dando preferencia a la zona en la que se encuentra el propio microservicio. Para ello, primero evalúa esta zona en base a tres criterios:
   - Cantidad de instancias disponibles en la zona (propiedad **ribbon.zoneAffinity.minAvailableServers**). Por defecto exige tener dos instancias disponibles como mínimo.
   - Carga media de las instancias (propiedad **ribbon.zoneAffinity.maxLoadPerServer**). Por defecto exige que sea inferior al 60%.
   - Porcentaje de instancias no disponibles (propiedad **ribbon.zoneAffinity.maxBlackOutServesrPercentage**). Por defecto exige que esté por debajo del 80%.

   Las propiedades anteriores pueden ser definidas de forma particular en el archivo *application.yml* (o en su defecto, el fichero *yml* subido al servidor Git) para cada microservicio que necesite invocar. Por ejemplo, dado que el microservicio Cesta requiere invocar al microservicio Catálogo, podríamos definir la siguiente configuración en su archivo *servicio.cesta.yml* ubicado en el servidor GitHub:

```
servicio.catalogo:
  ribbon:
    EnableZoneAffinity: true
    EnableZoneExclusive: false
    zoneAffinity:
      minAvailableServers: 1
      maxLoadPerServer: 0.8
      maxBlackOutServesrPercentage: 0.6
```

En caso de querer establecer una configuración común para todos los microservicios a invocar obviaremos el nombre del microservicio en la definición anterior.

Así pues, cuando se solicita la invocación de un microservicio, si la zona del microservicio solicitante no cumple los valores mínimos definidos, no se aplica la afinidad de zona y se devolverán todas zonas en las que se encuentren disponibles instancias del microservicio solicitado. En caso de que la zona del microservicio solicitante sí cumpla estos criterios mínimos, se devolverá está única zona.

2. Si el proceso anterior ha devuelto como resultado una única zona (ya sea porque es la zona afín, porque solo hay instancias del microservicio destino en esa zona o porque las demás zonas han sido descartadas), no se aplicará ningún tipo de lógica y se devuelven todas las instancias que componen esa zona a la regla de balanceo para que decida cuál invocar. Si como resultado se ha devuelto más de una zona, se aplica la lógica de selección de zonas a través de la clase **ZoneAvoidanceRule**, la cual va descartando zonas hasta quedarse con una en base al porcentaje de instancias no disponibles y al número de peticiones activas por instancia para cada zona.

Como ya hemos dicho, la configuración en cluster de Eureka, y por tanto la configuración de una arquitectura en zonas, no es objetivo de este libro. Vamos a estudiar a continuación la forma de definir reglas de balanceo para su uso dentro de una zona. Nótese que las reglas de balanceo se aplican de igual forma en una solución en *cluster* que en una solución *stand-alone*. En el primer caso, se aplicarán sobre la zona seleccionada por el balanceador, mientras que en el segundo caso se aplicarán sobre la única zona que existe.

5.1.2 Reglas de balanceo

Las reglas básicas para la selección de una instancia a partir de todas las disponibles en una zona son las siguientes:

- **RoundRobinRule**: Esta regla aplica el algoritmo *RoundRobin*, de forma que alterna las peticiones entre las diferentes instancias disponibles.
- **WeightedResponseTimeRule**: Con esta regla, a cada instancia se le asigna un peso en función de su tiempo medio de respuesta. Cuanto menor sea el tiempo, mayor peso tendrá la instancia. La instancia se elige

de forma aleatoria, de forma que, a mayor peso, mayor probabilidad de ser elegida.

- **RetryRule**: Esta regla añade lógica de reintentos a cualquiera de las dos reglas anteriores.

- Para indicar la regla de balanceo que queremos utilizar en un microservicio basta con crear una clase de configuración que reemplace el bean **ribbonRule**. En el siguiente ejemplo estamos indicado que se usa la regla **WeightedResponseTimeRule** con lógica de reintentos con un *timeout* de 500 milisegundos.

```
@Configuration
public class RibbonConfigurator {

  @Bean
  public IRule ribbonRule(){
   return new RetryRule(new WeightedResponseTimeRule(), 500);
  }
}
```

Por último, para que el microservicio tenga en cuenta la clase anterior tenemos que indicarlo en la clase principal con la anotación `@RibbonClient`.

```
@SpringBootApplication
@EnableDiscoveryClient
@RibbonClient
public class MainApp {
   public static void main(String[] args) {
      SpringApplication.run(MainApp.class, args);
   }
}
```

5.2 HYTRIX

En un entorno distribuido, como lo es una arquitectura de microservicios, los posibles puntos de fallo se multiplican, por lo que se hace indispensable prever mecanismos de protección frente a fallos que hagan el sistema resistente y tolerante a los mismos, sin interrupciones graves del servicio. Algunos de estos fallos son indisponibilidades temporales del servicio, latencia excesiva en satisfacer una petición o cortes en las comunicaciones, por ejemplo. Para minimizar sus efectos se ha diseñado un patrón llamado *circuit breaker*, que evita que los fallos se propaguen en cascada y que es el que, dentro de Netflix OSS, implementa Hystrix.

El diagrama de estados de un *circuit breaker* es relativamente sencillo (ver Figura 5.3). Si todo funciona correctamente, el circuito permanecerá cerrado (*closed*). Si, por algún motivo, se detectan inestabilidades y se supera el umbral de fallos, se abre el circuito como medida de prevención (*open*). Periódicamente, se comprueba la disponibilidad del sistema y se transiciona al estado *half open*, del que se volverá a *closed*, si se recupera el funcionamiento habitual; o, a *open*, si el fallo continúa.

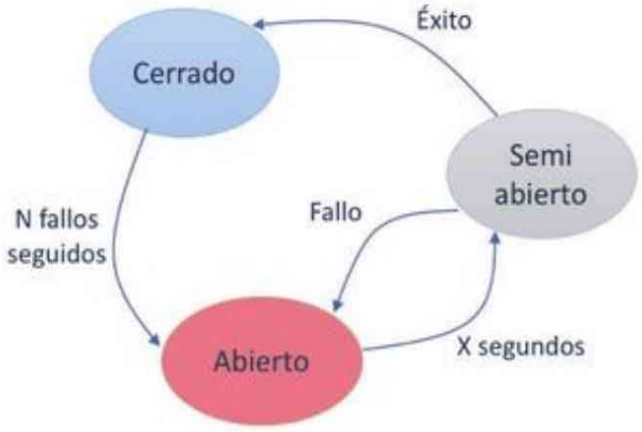

Figura 5.3. Diagrama de estados de un *circuit breaker*

Para utilizar el patrón *circuit breaker* en la invocación de microservicios, necesitamos incluir la dependencia de Hystrix el fichero *build.gradle* del microservicio que se desea proteger:

```
dependencies {
    compile 'org.springframework.cloud:spring-cloud-starter-hystrix'
}
```

A continuación, tenemos que anotar con **@HystrixCommand** los métodos que se encargan de la invocación a microservicios y, para los que se quiera, indicar un método de *fallback*, que será el que se invoque cuando se produzca un fallo. Estos métodos deben pertenecer a una clase anotada con **@Component** o **@Service** (La anotación **@RestController** incluye **@Component** por lo que, en nuestro caso, es válida). Para que Hystrix encuentre el método de *fallback* debe encontrarse en la misma clase que el método anotado como **@HystrixCommand** y debe indicarse en la propia anotación. Este método debe tener la misma signatura (parámetros y valor de retorno) que el método marcado con **@HystrixCommand.**

```
@HystrixCommand (fallbackMethod ="fallbackMethod")
@RequestMapping(
value = "/cesta/producto/{id}",
```

```
      method = RequestMethod.POST
)
public void addProductoCesta(@PathVariable(value="id") Integer id)
throws IOException, JSONException {

 Producto p=
 restTemplate.getForObject("http://SERVICIO.CATALOGO
productos/"+id,
                    Producto.class);
 DAO.getCestaDAO().addProducto(p);
}

private void fallbackMethod(Integer id){
   LOGGER.error(String.format("Error de conexión al Catalogo.
Añadir producto con id [%s]", id));
}
```

Por último, para que el microservicio procese adecuadamente los métodos marcados con **@HystrixCommand**, de forma que se implemente el patrón *circuit breaker*, debemos indicarle a la aplicación principal Spring Boot que busque en el classpath una implementación de este patrón (que en nuestro caso viene proporcionada por la librería Hystrix). Esto lo hacemos con la anotación **@EnableCircuitBreaker**. El siguiente ejemplo muestra como queda la aplicación principal del microservicio Cesta.

```
@SpringBootApplication
@EnableDiscoveryClient
@RibbonClient (name="ribbonCesta",
        configuration={RibbonConfigurator.class})
@EnableCircuitBreaker
public class MainApp {
   public static void main(String[] args) {
      SpringApplication.run(MainApp.class, args);
   }
}
```

5.2.1 Netflix Hystrix Dashboard y Turbine

Para finalizar la introducción a Hystrix, es necesario comentar que los métodos marcados con **@HystrixCommand** generan métricas en tiempo real sobre los resultados de ejecución y latencia. Estos datos son muy útiles para los operadores del sistema, ya que permiten hacerse una idea de cómo se está comportando el sistema.

Hystrix genera las métricas en bruto sobre un flujo de datos (*Hystrix stream)* y proporciona una interfaz llamada *Hystrix Dashboard* que consume estos datos y presenta una información gráfica basada en ellos. Por otro lado, también disponemos de Turbine. El *Hystrix Stream* proporciona información sobre una sola instancia, mientras que Turbine proporciona una forma de agregar la información de todas las instancias de un microservicio disponibles en un clúster.

Para hacer uso de estas herramientas, debemos anotar la clase principal de los microservicios que se quieren monitorizar con **@EnableHystrixDashboard** y **@EnableTurbine**, tal y como vemos a continuación para el microservicio Cesta.

```
@SpringBootApplication
@EnableDiscoveryClient
@RibbonClient (name="ribbonCesta",
               configuration={RibbonConfigurator.class})
@EnableCircuitBreaker
@EnableHystrixDashboard
@EnableTurbine
public class MainApp {
   public static void main(String[] args) {
      SpringApplication.run(MainApp.class, args);
   }
}
```

Antes debemos importar las dependencias en el *build.gradle*.

```
dependencies {
    compile 'org.springframework.cloud:spring-cloud-starter-
hystrix-dashboard'
    compile 'org.springframework.cloud:spring-cloud-starter-
turbine'
}
```

Finalmente, al ejecutar el microservicio tenemos disponible la página principal del dashboard con la uri **/hystrix**.

5.3 ZUUL

Tal y como ya se ha comentado, en una arquitectura de microservicios podemos tener más de una instancia de un microservicio. Por ejemplo, la Figura 5.5 muestra como disponemos de dos instancias de los servicios Catalogo y Cesta. Sin embargo, es interesante ocultar la complejidad de nuestro sistema al mundo exterior,

de forma que solo debe haber una dirección IP expuesta en un puerto disponible para clientes entrantes. Es por eso que necesitamos una *API Gateway* como Zuul.

Figura 5.4. Dos instancias de los servicios Catálogo y Cesta

Zuul es un servidor de frontera (*edge server*) que proporciona funcionalidades de enrutado dinámico, balanceo de carga, seguridad y monitorización de peticiones. Se integra fácilmente con Eureka. De hecho, Zuul se vale de Ribbon (y, por tanto, de Eureka) para localizar la instancia del microservicio a la que derivar una petición a través de un comando Hystrix.

Zuul es el punto de entrada al ecosistema de microservicios. Cada petición que recibe es procesada por una cadena de filtros (ver Figura 4.7). De esta manera, es posible rechazarla si no cumple ciertos criterios, registrarla con fines de monitorización y control o redirigirla a una instancia concreta de algún microservicio, por ejemplo. En definitiva, debe tratarse de incluir en un filtro Zuul cualquier tarea que resulte común.

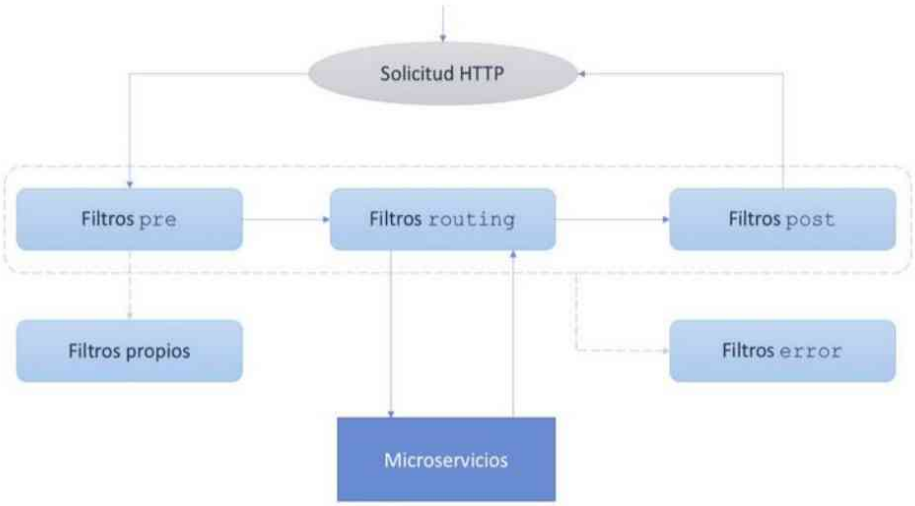

Figura 5.5. Cadena de filtros de Zuul

Para crear un servidor frontera con Zuul, basta con crear una aplicación Spring Boot y anotarla con **@EnableZuulProxy**:

```
@Configuration
@ComponentScan
@EnableAutoConfiguration(exclude = {ErrorMvcAutoConfiguration.class})
@EnableDiscoveryClient
@EnableZuulProxy
public class ZuulServerApp {
   public static void main( String[] args ){
      SpringApplication.run(ZuulServerApp.class, args);
   }
}
```

Como en anteriores ocasiones, es necesario incluir la dependencia correspondiente en el *build.gradle*.

```
dependencies {
    compile('org.springframework.cloud:spring-cloud-netflix
zuul-server')
}
```

Finalmente, en el fichero *application.yml* especificamos las reglas de encaminamiento, junto con los datos de conexión a Eureka y el puerto en el que escuchará el servidor Zuul. En este caso, Zuul dirigirá las conexiones con el path / **catalogo** a una de las instancias del microservicio Catalogo. Lo mismo hará para el path **/cesta**. Por ejemplo, si lanzamos una conexión a la url **http://localhost:6666/catalogo/productos**, Zuul la redigirá a la url de una instancia microservicio servicio. catalogo, como por ejemplo **http://localhost:1111/productos.** De esta forma, todas las conexiones exteriores a los microservicios se centralizan en la url **http://localhost:6666.**

```
zuul:
  routes:
    catalogo:
      path: /catalogo/**
      serviceId: servicio.catalogo
    compra:
      path: /compra/**
      serviceId: servicio.cesta
eureka:
  client:
    registerWithEureka: false
    serviceUrl:
      defaultZone: http://localhost:2222/eureka/

server:
  port: 6666
```

Zuul utiliza de forma automática Ribbon para balancear la carga. Así pues, cada vez que le llegue una conexión a Zuul, éste hará uso del descubrimiento de servicios a través de Ribbon, de forma que solicitará a Eureka la instancias disponibles del microservicio correspondiente, y elegirá una de ellas.

De este modo, y dado que la regla por defecto de Ribbon es Round Robin, si hiciéramos múltiples llamadas al microservicio Catalogo, Zuul las redirigiría alternativamente a las dos instancias que tenemos disponibles en el sistema (una en el puerto 1111 y otra en el 1112, ver Figura 5.1).

Además de las capacidades de enrutado que hemos visto, Zuul ofrece muchas otras funcionalidades para sacarle partido a nuestro Edge Service, como por ejemplo los filtros, los cuales permiten realizar un amplio número de acciones durante el ciclo de vida de las peticiones HTTP.

Existen cuatro tipos distintos de filtros, correspondientes a cada uno de los estados por los que pasa una petición:

- **pre**: Filtro que se ejecuta antes del enrutado
- **routing**: Filtro que maneja el enrutado actual
- **post**: Filtro que se ejecuta después del enrutado
- **error**: Filtro que se ejecuta cuando ocurre un error en el manejo de la petición

Para definir un filtro en Zuul basta con crear una clase que herede de ZuulFilter, lo cual nos obliga a sobrescribir cuatro métodos:

- **filterType()**: devuelve el tipo de filtro a través de un string (pre, routing, post o error).
- **filterOrder()**: indica el orden en el que debe ejecutarse con respecto a otros filtros del mismo tipo.
- **shouldFilter()**: devuelve **true** o **false** para indicar si se debe aplicar o no. Es útil para definir condiciones que deban cumplirse para la aplicación del filtro.
- **run()**: define las acciones a realizar.

En el siguiente ejemplo se muestra la definición de un filtro de tipo **pre**.

```
public class MiPreFiltro extends ZuulFilter {

  private static Logger LOGGER =
                 LoggerFactory.getLogger(MiPreFiltro.class);
  private static final String FILTERTYPE = "pre";
```

```
    private static final Integer FILTERORDER = 1;

    public MiPreFiltro(){//Para Spring}

    @Override
    public String filterType() {
        return FILTERTYPE;
    }

    @Override
    public int filterOrder() {
        return FILTERORDER;
    }

    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
    public Object run() {
        final HttpServletRequest request =
                  RequestContext.getCurrentContext().getRequest();
        LOGGER.info("Petición {} a {}",
              request.getMethod(),
              request.getRequestURL().toString());
        return null;
    }

}
```

Por último, hay que indicarle a Zuul la existencia de cada filtro. Para ello, basta con crearlo como un Bean de Spring en la clase principal de la aplicación, tal y como se hace a continuación mediante el método **miPreFiltro()**.

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableZuulProxy
public class ZuulServer {

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

    @Bean
    public MiPreFiltro miPreFiltro() {
     return new MiPreFiltro();
    }
}
```

6

OAUTH2

Uno de los aspectos más importantes que tenemos que considerar cuando se expone una API de acceso público como la de muchos microservicios es la seguridad. En el capítulo 4 ya vimos como Spring nos proporcionaba mecanismos para activar de forma muy sencilla el protocolo básico de autenticación HTTP en el servidor de configuración o en los microservicios. En este capítulo, vamos a ver como utilizar Spring Cloud y OAuth2 para proporcionar seguridad de acceso de token.

6.1 CONCEPTOS BÁSICOS DE OAUTH2

El estándar OAuth2 es utilizado actualmente por multitud de sitios web que permiten acceder a sus recursos a través de una API pública. OAuth2 permite que las aplicaciones web (o servicios) puedan acceder de manera delimitada a los recursos de otra aplicación en nombre del usuario, sin tener que dar para ello las credenciales (usuario/password, etc.) a esta tercera aplicación.

OAuth2 se basa en el concepto de token de acceso, de forma que para que una aplicación debe conseguir este toquen para pueda acceder a un recurso sin tener que introducir las credenciales.

Los conceptos básicos de OAuth2 son los siguientes:

- **Propietario de recursos**. Es una entidad capaz de dar acceso a recursos protegidos. Cuando es una persona nos referiremos a él como usuario final.
- **Cliente**. Es la aplicación que hace peticiones a recursos protegidos en nombre de un propietario de los recursos con la autorización del mismo.

- **Proveedor**: El proveedor es el responsable de exponer los recursos protegidos. Su configuración implica procurar que los clientes de autenticación puedan acceder a dichos recursos de manera independiente o en nombre de un determinado usuario, lo que se consigue a través de la verificación de testigos (tokens) de acceso a los recursos protegidos. En ocasiones, el proveedor también ofrece una interfaz gráfica a través de la cual es posible comprobar a qué recursos un determinado cliente tiene acceso o no. El rol de proveedor se reparte entre el servidor de autorización y el servidor de recursos, aunque algunas veces se encuentran centralizados en un mismo servidor.
  - **Servidor de Recursos**. Es la entidad que tiene los recursos protegidos. Es capaz de aceptar y responder peticiones usando un token de acceso que debe venir en el cuerpo de la petición.
  - **Servidor de Autenticación y Autorización de Usuario** (UAA, de la siglas del termino en inglés, *User Authorization and Authentication*). El UAA es el responsable de generar tokens de acceso y validar usuarios y credenciales.

La descripción general del protocolo de autorización OAth2 es el que se muestra en la Figura 6.1. En concreto, se realizan los siguientes pasos:

1. La aplicación cliente solicita autorización para acceder a los recursos del propietario.
2. Si el usuario autoriza la solicitud, la aplicación recibe una concesión de autorización.
3. A continuación, la aplicación cliente solicita un token de acceso al servidor de autorización proporcionando una identificación de su identidad y la concesión de autorización otorgada.
4. Si la identidad de la aplicación cliente se autentica y la concesión de autorización es válida, el servidor de autorización emite un token de acceso para la aplicación cliente. La autorización está completa.
5. La aplicación cliente solicita el recurso al servidor de recursos presentando el token de acceso para la autenticación.
6. Si el token de acceso es válido, el servidor de recursos proporciona el recurso a la aplicación cliente.

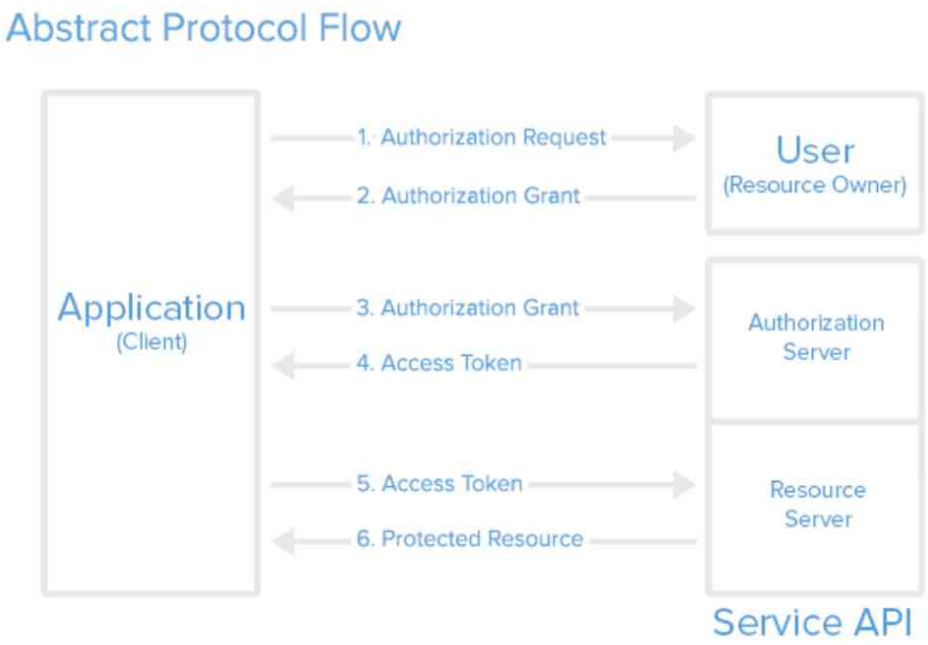

Figura 6.1. Protocolo OAth2

En la descripción anterior del protocolo OAuth2, los primeros cuatro pasos cubren la obtención de una concesión de autorización y un token de acceso. El tipo de concesión de autorización depende del método utilizado por la aplicación para solicitar la autorización y de los tipos de concesión admitidos OAth2. En concreto, OAth2 permite cuatro tipos:

- **Código de autorización** (tipo **authorization_code)**: Este tipo de autorización utiliza aplicaciones de terceros (por ejemplo, Facebook, Twitter, etc.) para identificar al usuario, de forma que se puede mantener la confidencialidad del mismo. El usuario no tendrá que compartir sus credenciales con la aplicación cliente. Este es un flujo basado en la redirección, lo que significa que la aplicación debe ser capaz de interactuar con el navegador web del usuario.
  - La aplicación cliente redirige al usuario al servidor UAA indicando los siguiente parámetros en la URL:
    - **response_type**: parámetro con el valor **code**
    - **client_id:** el ID de la aplicación cliente
    - **redirect_uri:** URI de redirección tras concederse el código de autorización. Es opcional. Si no se indica se redirige a una URI predefinida.

  - **scope:** lista de scopes, separada por espacios. Más adelante explicaremos qué es un *scope*.
  - **state:** token CSRF (*Cross-Site Request Forgery*) . Parámetro opcional, pero recomendado para evitar ataques CSRF. El token CSRF debe ser guardado en la sesión del usuario para validarlo cuando el flujo de conexiones retorne (a la URI de redirección).

- El servidor UAA valida los parámetros que le llegan y solicita al usuario que se identifique en el propio servidor y de permiso a la aplicación cliente para acceder al servicio solicitado.
- Si el usuario da permiso a la aplicación cliente, el servidor UAA lo redirige a la URI indicada en el primer paso, añadiendo a la URI los siguientes parámetros:
  - **code:** un código de autorización.
  - **state**: mismo valor que en la solicitud inicial. Se debe comparar este valor con el valor almacenado en la sesión del usuario para garantizar que el código de autorización obtenido sea en respuesta a las solicitudes realizadas por este cliente, y no por otras aplicaciones.
- La aplicación cliente envía una conexión POST al servidor UAA con el id y el secreto de la aplicación cliente en un encabezado de autenticación básica de HTTP y los siguientes parámetros en el cuerpo:
  - **grant_type**: este parámetro debe contener el valor **authorization_code**.
  - **redirect_uri:** misma URI a la que se redirigió al ususario en el paso anterior.
  - **code**: código de autorización añadido en el paso anterior a la URI de redireccionamiento.
- El UAA verifica el código de autorización y devuelve un objeto JSON con los siguientes datos:
  - **token_type**: parámetro con el valor **Bearer**
  - **expires_in**: número entero que representa el tiempo de vida (Time To Live, TTL) del *token* de acceso.
  - **access_token:** un JWT (JSON Web Token) firmado con la clave privada del servidor UAA.

  - **refresh_token:** token de refresco. Se trata de un valor encriptado que puede utilizarse para refrescar el token en caso de que expire.

- **Implícito**: La concesión implícita es similar a la concesión de código de autorización con dos diferencias básica. En primer lugar, su uso debe destinarse a clientes basados en agentes de usuario (por ejemplo, aplicaciones web de una sola página) que no pueden mantener secreto de cliente porque se puede acceder fácilmente a todo el código y al almacenamiento de la aplicación. En segundo lugar, el servidor de autorización no devuelve un código de autorización para ser intercambiado por un token de acceso. En su lugar, el servidor de autorización devuelve directamente un token de acceso. Por último, comentar que este tipo de concesión no soporta tokens de refresco.

  - La aplicación cliente redirige al usuario al servidor UAA indicando los siguientes parámetros en la URL:
    - **response_type**: parámetro con el valor **token**
    - **client_id:** el ID de la aplicación cliente
    - **redirect_uri:** URI donde debe redirigirse al usuario para su identificación. Es opcional. Si no se indica se redirige al usuario a un URI predefinida.
    - **scope:** lista de scopes, separada por espacios. Un *scope* es un permiso que se solicita para operar sobre el recurso (por ejemplo, **read** o **writ**e).
    - **state:** token CSRF (*Cross-Site Request Forgery*) . Parámetro opcional pero recomendado para evitar ataques CSRF. El token CSRF debe ser guardado en la sesión del usuario para validarlo cuando éste retorne.
  - El servidor UAA valida los parámetros que le llegan y solicita al usuario que se identifique en el propio servidor y de permiso a la aplicación cliente para acceder al servicio solicitado.
  - Si el usuario da permiso a la aplicación cliente, el servidor UAA lo redirige a la URI indicada en el primer paso, añadiendo a la URI los siguientes parámetros:
    - **token_type**: parámetro con el valor **Bearer**
    - **expires_in**: TTL del *token* de acceso.
    - **access_token:** JWT firmado con la clave privada del servidor UAA.

  - **state**: mismo valor que en la solicitud inicial. Se debe comparar este valor con el valor almacenado en la sesión del usuario para garantizar que el código de autorización obtenido sea en respuesta a las solicitudes realizadas por este cliente, y no por otras aplicaciones

- **Credenciales de contraseña** del propietario del recurso (tipo **password**): Este tipo de concesión se utiliza para aplicaciones cliente en las que el usuario tiene una relación de confianza con ellas. La aplicación se autentica en el servidor UAA en nombre del usuario y recibe el token correcto.
  - El usuario proporciona sus credenciales a la aplicación cliente, comúnmente a través de un formulario de inicio de sesión.
  - La aplicación cliente envía una solicitud POST al servidor UAA con el id y el secreto de la aplicación cliente en un encabezado de autenticación básica de HTTP y con los siguientes parámetros en el cuerpo:
    - **grant_type:** debe contener el valor **password**
    - **scope**: lista de scopes, separada por espacios.
    - **username:** nombre de usuario.
    - **password:** password de usuario.
  - El servidor UAA valida al usuario y devuelve un objeto JSON con los siguientes datos:
    - **token_type**: parámetro con el valor **Bearer**
    - **expires_in**: TTL del *token* de acceso.
    - **access_token:** JWT firmado con la clave privada del servidor UAA.
    - **refresh_token:** token de refresco.
- **Credenciales del cliente** (tipo **client_credentials)**: Esta concesión es adecuada para la autenticación de máquina a máquina, por ejemplo, para usar en un *cron job* que realiza tareas de mantenimiento sobre una API. Otro ejemplo sería un cliente que realiza solicitudes a una API que no requieren permiso del usuario.

- La aplicación cliente envía los siguiente parámetros al servidor UAA en el cuerpo de una conexión POST, y con el id y el secreto de la aplicación cliente en un encabezado de autenticación básica de HTTP:
  - **grant_type:** este parámetro debe contener el valor **client_credentials**.
  - **scope**: lista de scopes, separada por espacios.
- El UAA valida las credenciales y devuelve un objeto JSON con los siguientes datos:
  - **token_type**: parámetro con el valor **Bearer**
  - **expires_in**: TTL del *token* de acceso.
  - **access_token:** JWT firmado con la clave privada del servidor UAA.

- **Tokens de refresco**: Los tokens de acceso acaban caducando. Sin embargo, algunas concesiones (tipo **authorization_code y password***)* responden con un token de actualización que permite al cliente actualizar este token.
  - La aplicación cliente envía una solicitud POST al servidor UAA con el id y el secreto de la aplicación cliente en un encabezado de autenticación básica de HTTP y con los siguientes parámetros en el cuerpo:
    - **grant_type:** este parámetro debe contener el valor **refresh_token**.
    - **refresh_token:** token de refresco.
    - **scope**: lista de scopes, separada por espacios.
  - El UAA devuelve un objeto JSON con los siguiente datos:
    - **token_type**: parámetro con el valor **Bearer**
    - **expires_in**: TTL del *token* de acceso.
    - **access_token:** JWT firmado con la clave privada del servidor UAA.
    - **refresh_token:** token de refresco.

6.2 OAUTH2 EN UNA ARQUITECTURA DE MICROSERVICIOS

En esta sección, vamos a explicar como incluimos seguridad basada en OAuth2 en nuestra arquitectura de microservicios. Lo primero que tendremos que hacer es construir un nuevo servicio que actúe como servidor UAA y registrarlo en el servidor Eureka, tal y como se aprecia en la Figura 6.2.

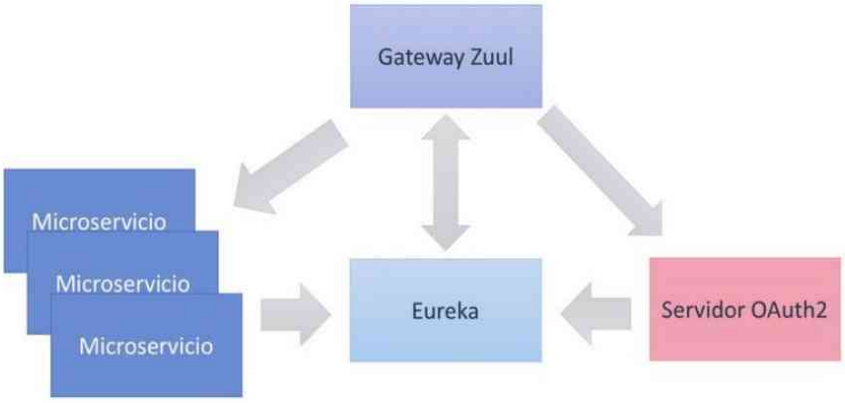

Figura 6.2. Arquitectura de microservicios con OAuth2

El servidor UAA va a ser el encargado de identificar a los usuarios en tipos de concesiones como la basada en código de autorización y la implícita, y de generar los tokens de acceso con cualquier tipo de concesión para permitir acceder a los microservicios, los cuales van a ser definidos cada uno de ellos como servidores de recursos.

Por otro lado, vamos a tener que configurar el *Gateway* Zuul para que permita el uso de OAuth2 y propague las diferentes conexiones de identificación al servidor UAA o a los microservicios.

A continuación, vamos a ver paso a paso como asegurar nuestra arquitectura de microservicios con OAuth2 a través de las herramientas que nos proporciona Spring.

6.2.1 Creación del Servidor UAA con Spring

En primer lugar, vamos a estudiar cómo construir un servidor UAA con Spring.

Spring permite definir un servidor UAA siguiendo la misma estrategia que en otras ocasiones, es decir, tenemos que crear una aplicación Spring Boot y anotarla

adecuadamente. En concreto, la anotación a utilizar es **@EnableAuthorizationServer** tal y como se muestra a continuación. Dado que lo vamos a integrar en nuestra arquitectura de microservicio, añadimos también la anotación para su registro con el servidor Eureka.

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableAuthorizationServer
public class OAuthServer {
  public static void main(String[] args) {
   SpringApplication.run(OAuthServer.class, args);
  }
}
```

Para poder utilizar la anotación que define la aplicación como un servidor UAA deberemos incluir en el *build.gradle* la dependencia de Spring Security que permite el uso de OAuth2 (además de la dependencia de Eureka que ya hemos utilizado anteriormente).

```
dependencies {
 compile 'org.springframework.security.oauth:spring-security
oauth2'
 […]
}
```

Con estos simples pasos acabamos de crear un servidor UAA que proporciona dos puntos de acceso HTTP:

/oauth/authorize	Punto de acceso destinado a que el propietario de un recurso se identifique en el servidor UAA y conceda autorización para que una aplicación cliente acceda a un recurso. Se utiliza cuando el tipo de concesión utilizando es código de autorización o implícito.
/oauth/token	Punto de acceso mediante el cual el servidor UAA proporciona un token de acceso para un recurso determinado.

Tabla 6.1. Puntos finales obligatorios

Una vez construido el servidor UAA necesitamos configurarlo adecuadamente según el tipo de concesión que vayamos a utilizar. En el caso de una arquitectura de microservicios, donde unos interactúan con otros sin una participación explicita del usuario, vamos a configurar un tipo de concesión basada en credenciales del cliente, lo cual significa que para que una aplicación cliente obtenga el token de acceso debe proporcionar el id y el secreto de dicha aplicación. Esta información debe ser reconocidas por el propio UAA. Para facilitar la implementación vamos a definir un único cliente en memoria. También es posible gestionar un conjunto de aplicaciones

cliente almacenadas, por ejemplo, en una fuente de datos JDBC, aunque este punto queda fuera del alcance de este libro. Por otro lado, en caso de que nuestros servicios tuvieran un acceso público, lo más adecuado sería implementar un tipo de concesión de tipo código de autorización.

Para definir una aplicación cliente en memoria basta con incluirla en el archivo *application.yml* de la aplicación Spring Boot que acabamos de construir. A continuación se muestra la definición de este archivo, la cual incluye también otras configuraciones ya explicadas, como el registro del servidor UAA en Eureka.

```
spring:
  application:
     name: uaa
server:
  port: 7777
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:2222/eureka/
security:
  oauth2:
    client:
      client-id: tiendaonline
      client-secret: elsecreto
      authorized-grant-types: client_credentials
      scope: all
```

Como vemos, a través del archivo *application.yml* estamos configurando el servidor UAA para que acepte concesiones de tipo **client_credential** (basada en credenciales del cliente) para el cliente con id **tiendaonline** y secreto **elsecreto**, otorgándole todos los permisos (**scope: all**).

Para comprobar que nuestro servidor UAA funciona correctamente basta con lanzar la aplicación y solicitarle el token de autorización a través de una conexión POST HTTP. Como se ha comentado anteriormente, esta conexión debe enviar en el encabezado de autenticación HTTP el id y el cliente de la aplicación cliente, y el tipo de concesión utilizado en el cuerpo de la conexión. Utilizando una herramienta como curl, sería como sigue:

```
> curl -X POST --user 'tiendaonline:elsecreto'
             -d 'grant_type=client_credentials'
             http://localhost:7777/oauth/token
```

Si todo ha funcionado correctamente, el servidor nos devolverá el token de autorización a través de un objeto JSON:

```
{“access_token”:”4b2c9c35-b031-4bd2-af34-19107395b116”,
“token_type”:”bearer”,”expires_in”:35392,”scope”:”all”}
```

Nótese que en un entorno real estás conexiones deberían hacerse bajo un protocolo seguro HTTPS. En caso de que la herramienta de conexión HTTP nos obligue a definir nosotros mismos el encabezado de autorización básica, tenemos que tener en cuenta que éste debe ir codificado en base64. Así pues, al codificar la cadena de texto **tiendaonline:elsecreto** en base64 la cabecera de autenticación básica de HTTP sería como sigue:

```
authorization: Basic dGllbmRhb25saW5lOmVsc2VjcmV0bw==
```

Una vez ya disponemos del token de acceso ya podemos conectarnos al servidor de recursos y acceder a los recursos correspondientes. En este caso, vamos a definir un servidor de recursos por cada uno de los microservicios que tenemos en nuestra arquitectura.

6.2.2 Configurando microservicios como recursos

Para definir nuestros microservicios como servidores de recursos basta con marcarlos con la anotación **@EnableResourceServer**. Con esta simple acción, Spring convertirá cada microservicio en un servidor de recursos y lo Spring segurizará de forma que sólo podamos acceder a él proporcionando un token de acceso generado por el servidor UAA.

A modo de ejemplo, se muestra la nueva aplicación principal del microservicio catálogo. Como vemos, lo único que cambia es que hemos añadido la anotación comentada.

```
@EnableDiscoveryClient
@SpringBootApplication
@EnableResourceServer
public class MainApp {
   public static void main(String[] args) {
      SpringApplication.run(MainApp.class, args);
   }
}
```

Si en este punto, lanzamos el microservicio e intentamos acceder a él mediante una conexión HTTP veremos cómo no tenemos acceso.

```
> curl http://localhost:1111/productos
{“error”:”unauthorized”,”error_description”:”Full authentication is
required to access this resource”}
```

Para poder acceder al microservicio, necesitaremos crear una conexión HTTP con un encabezado de autorización que incluya el token de acceso generado por el servidor UAA.

```
curl -H "Authorization: Bearer 4b2c9c35-b031-4bd2-af34
19107395b116" http://localhost:1111/productos
{"error":"invalid_token","error_description":"Invalid access token:
4b2c9c35-b031-4bd2-af34-19107395b116"}
```

Como vemos, en el código anterior, el microservicio no ha identificado el token como válido. Esto es debido a que, con la configuración realizada hasta ahora, el servidor de recursos (es decir, el microservicio) no tiene ninguna interacción el servidor UAA que le permita validar el token recibido. Para permitir esto vamos a añadir la siguiente configuración en el archivo YML de nuestros microservicios. Esta configuración le indica al microservicio una URL del servidor UAA en la cual puede validar el usuario correspondiente al token recibido.

```
security:
  oauth2:
    resource:
      user-info-uri: http://localhost:7777/user
```

Como vemos, la URL definida en el código anterior apunta al servidor UAA (puerto 777, definido en la configuración del mismo, ver sección anterior). Sin embargo, el punto de acceso **/user** no era uno de los puntos generados por defecto al crear un servidor UAA con la anotación **@EnableResourceServer**. Para añadir este nuevo punto de acceso, tenemos que configurar el servidor UAA como un controlador REST (anotación **@RestController**) e implementar un método que sea capaz de validar token de acceso cuando se reciban conexiones en una determinada URL. Para ello, Spring cuenta con la clase **Principal**, que nos va a permitir crear fácilmente un método que reciba un token de acceso, lo valide y devuelva los datos del usuario correspondiente. Este método debe, además, definir un punto de acceso que esté securizado mediante token (generados por el mismo UAA). Para ello, basta con definir el servidor UAA también como un servidor de recursos, de forma que securice los puntos de acceso expuesto a través de tokens. Dado que es él mismo quien genera los tokens no necesita ninguna configuración adicional para validarlos. Así pues, las extensiones que debemos realizar a nuestro servidor UAA son las siguientes:

- Convertirlo en un controlador REST añadiendo la anotación **@RestController**.
- Publicar un punto de acceso que valide tokens a través de la clase **Principa**l.

- Securizar este punto de acceso mediante una identificación con token añadiendo la anotación **@EnableResourceServer** al propio servidor UAA.

Con todo ello, el servidor UAA debe quedar como se muestra a continuación.

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableAuthorizationServer
@EnableResourceServer
@RestController
public class OAuthServer {

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

   @RequestMapping("/user")
public Principal user(Principal user) {
return user;
}

}
```

Si una vez actualizado el servidor UAA y el microservicio Catálogo tal y como hemos explicado, solicitamos nuevamente un token de acceso e intentamos acceder al microservicio, veremos como, esta vez sí, tenemos acceso al mismo.

```
> curl -X POST --user 'tiendaonline:elsecreto'
             -d 'grant_type=client_credentials'
             http://localhost:7777/oauth/token
{"access_token":"d4a74933-0f4a-455b-8b73-488b944bed63",
"token_type":"bearer","expires_in":43199,"scope":"all"}
> curl -H
       "Authorization: Bearer d4a74933-0f4a-455b-8b73-488b944bed63"
       http://localhost:1111/productos
[{"id":1,"nombre":"Apple iPhone 8
Plus","precio":571.9,"descripcion":"Smartphone con pantalla de 13,9
cm (64 GB, Gris espacial)"},{"id":2,"nombre":"Samsung Galaxy
S8","precio":669.9,"descripcion":"Smartphone libre de 5.8
QHD+"},{"id":3,"nombre":"Huawei Mediapad T3
10","precio":145.99,"descripcion":"Tablet de 9.6 pulgadas IPS HD"}]
```

6.2.3 Conexión entre microservicios

En las pruebas que hemos realizado hasta ahora hemos interactuado con el microservicio Catálogo, de forma que primero hemos solicitado al servidor UAA un token de acceso y a continuación hemos utilizado este token para acceder al microservicio.

Esta forma de proceder está bien si somos una aplicación cliente que quieres acceder a un microservicio que opera de forma aislada. Sin embargo, los microservicio interactúan entre sí. Por ejemplo, el microservicio Cesta interactúa con el microservicio Catálogo cada vez que añadimos un producto a la cesta.

Si una aplicación cliente quiere consumir el servicio Cesta, solicitará el token de acceso al servidor UAA y a continuación accederá a dicho servicio. En este caso, sin embargo, el servicio Cesta no opera de forma aislada, sino que necesita interactuar con Catálogo, el cual necesita un token de acceso. No tiene sentido que el microservicio Cesta solicite este token al servidor UAA ya que la aplicación cliente se lo acaba de proporcionar. Lo único que debe hacer es propagar el encabezado **Authentication** que recibe de la aplicación cliente en su conexión con Catálogo.

Para realizar esta tarea debemos modificar ligeramente la implementación del método **addProductoCesta** que implementaba el punto de acceso del microservicio Cesta que necesita interactuar con el microservicio Catálogo. Lo que tenemos que hacer en primer lugar es añadir un nuevo parámetro que se enlace con el encabezado que queremos propagar, para lo cual utilizamos la anotación **@RequestHeader**. A continuación, creamos el encabezado para la nueva conexión mediante las clases **HttpHeaders** y **HttpEntity**. Tras ello, realizamos la conexión HTTP con el método exchange, ya que el método **getForObject q**ue utilizamos anteriormente no permite el envío de encabezados. Por último, llamamos a la clase DAO correspondiente. Nótese también que este método estaba marcado como una comando Hystrix con su método de *fallback*. Al cambiar la signatura del método debemos cambiar también la del método de *fallback*.

```
@HystrixCommand(fallbackMethod=”fallbackMethod”)
@RequestMapping(value = “/cesta/producto/{id}”,
        method = RequestMethod.POST)
public void addProductoCesta(
        @PathVariable(value=”id”) Integer id,
        @RequestHeader(“Authorization”) String auth
    ) throws IOException, JSONException {


  // Encabezado Auhtentication
  HttpHeaders headers = new HttpHeaders();
```

```
        headers.set("Authorization", auth);
        HttpEntity<String> entity = new HttpEntity<>("parameters",
       headers);

        //Conexión HTTP enviando encabezado
        ResponseEntity<Producto> res =
           restTemplate.exchange("http://SERVICIO.CATALOGO
      productos/"+id,
                                  HttpMethod.GET, entity, Producto
      class);

        //Recuperamos el producto a partir de la respuesta
        DAO.getCestaDAO().addProducto(res.getBody());
      }

      private void fallbackMethod(Integer id, String auth){
        LOGGER.error(String.format("Error de conexión al Catalogo.
      Añadir producto con id [%s]", id));
      }
```

6.2.4 Preparando Zuul para propagar solicitudes OAuth

En la arquitectura de microservicios que hemos construido hasta ahora disponemos de un Gataway creado con Zuul que se encarga de centralizar todas las conexiones exteriores y realizar redireccionamiento y/o filtrados sobre las mismas. Sin embargo, en las pruebas que hemos realizado anteriormente hemos interactuado directamente con el servidor UAA y el microservicio, sin pasar por el Gateway.

Si intentamos hacer las mismas pruebas que hemos hecho en la sección anterior pero a través de las URL que nos proporciona Zuul, veremos como siempre obtenemos como respuesta un fallo en la autorización. Por ejemplo, si intentamos acceder al servicio Catalogo pasándole el token de acceso a través de Zuul ocurre lo que se aprecia a continuación.

```
> curl -H
        "Authorization: Bearer d4a74933-0f4a-455b-8b73
488b944bed63"
        http://localhost:6666/catalogo/productos
{"error":"unauthorized","error_description":"Full authentication is
required to access this resource"}
```

Como vemos, aunque le estamos pasando el token de acceso de igual manera que hacíamos directamente con el servicio, al hacerlo a través de Zuul falla. Esto es

debido a que Zuul, por defecto, filtra los encabezados marcado como sensibles, que por defecto son **Cookie**, **Set-Cookie**, **Authorization**, lo cual impide su propagación, es decir, no le llega al microservicio ni al servidor UAA.

Para que Zuul sea capaz de propagar estos encabezados basta con definir de forma explícita la lista de encabezados sensibles a través de la propiedad de configuración **sensitiveHeaders**. Si le damos un valor vacío como en el caso que se aprecia a continuación estamos eliminando los tres encabezados de la lista. Nótese, además, como en la configuración de Zuul que se presenta se ha añadido una nueva regla para redirigir las conexiones al servidor UAA.

```
zuul:
  routes:
    catalogo:
      path: /catalogo/**
      sensitiveHeaders:
      serviceId: servicio.catalogo
    compra:
      path: /compra/**
      sensitiveHeaders:
      serviceId: servicio.cesta
    uaa:
      path: /uaa/**
      sensitiveHeaders:
      serviceId: uaa
```

Con la confihuración anterior definida en nuestra aplicación Zuul comprobamos como realizar las pruebas anteriores a través de las URLs del Gateway (puerto 6666).

```
> curl -X POST --user 'tiendaonline:elsecreto'
       -d 'grant_type=client_credentials'
       http://localhost:6666/uaa/oauth/token
{"access_token":"6d408a4e-6b3a-4ae1-bfe7-075695c5c446",
"token_type":"bearer","expires_in":43199,"scope":"all"}

> curl -H
       "Authorization: Bearer 6d408a4e-6b3a-4ae1-bfe7-075695c5c446"
       http://localhost:6666/catalogo/productos
[{"id":1,"nombre":"Apple iPhone 8
Plus","precio":571.9,"descripcion":"Smartphone con pantalla de 13,9
cm (64 GB, Gris espacial)"},{"id":2,"nombre":"Samsung Galaxy
S8","precio":669.9,"descripcion":"Smartphone libre de 5.8
QHD+"},{"id":3,"nombre":"Huawei Mediapad T3
10","precio":145.99,"descripcion":"Tablet de 9.6 pulgadas IPS HD"}]
```

7

ACCESO A DATOS EN MICROSERVICIOS. ASPECTOS DE DISEÑO

Hasta ahora hemos desarrollado una arquitectura basado en microservicios centrándonos en cómo desarrollar los diferentes elementos que la componen y cómo conectar entre sí cada uno de ellos. Sin embargo, hemos obviado por completo el tema del acceso a datos.

En una arquitectura basada en microservicios, como en cualquier otro tipo de sistema, va a ser usual que nuestros servicios necesiten acceder a una base de datos, ya sea para almacenar datos o para recuperarlos. En el ejemplo de la tienda online que estamos desarrollando, el microservicio Catálogo proporciona información almacenada en una tabla **productos**, el servicio Cesta accede a la tabla **cesta**, el servicio Pedidos a **pedidos** y el servicio Cuenta a **cuentas**. Además, podemos tener transacciones que necesiten acceder a varias de estas tablas.

Llegados a este punto, necesitamos plantearnos de qué forma implementamos el acceso a una base de datos (o cualquier almacén de datos) en una arquitectura basada en microservicios. Para ello, consideremos, en primer lugar, una de las principales razones por las que optamos por implementar una arquitectura de microservicios: proporcionar un alto grado de desacoplamiento y baja cohesión en el código. Con ello, conseguimos que equipos de desarrollo diferentes puedan trabajar en distintas partes del sistema (distintos microservicios) a diferentes velocidades con un impacto mínimo entre cada uno de ellos. Los equipos son autónomos, capaces de tomar decisiones sobre cómo implementar y operar mejor cada uno de los servicios, y libres de realizar cambios tan rápido como sea necesario.

El hecho de que los microservicios estén poco acoplados entre ellos permite que se desarrollen e implementen de forma independiente y en paralelo. Si tenemos en cuenta los datos que manejan los servicios, éstos deben gestionarse de forma

que permita mantener este bajo nivel de acoplamiento entre microservicios, si no queremos perder los beneficios que nos aporta este tipo de arquitectura. Así pues, los datos persistentes de cada microservicio deben ser privados para ese servicio y solo se deben poder acceder a ellos a través de su API. Si dos o más microservicios comparten datos persistentes estamos creando una fuerte dependencia entre estos dos microservicios, lo cual va en contra de la filosofía de este tipo de sistemas.

Teniendo en cuenta una base de datos relacional, existen tres formas básicas de mantener la privacidad en la persistencia de los datos de un microservicio:

- Tablas privadas por servicio: cada servicio posee un conjunto de tablas a las que solo debe acceder ese servicio. Todos los servicios acceden al mismo servidor de bases de datos.
- Esquema por servicio: cada servicio tiene un esquema de base de datos privado. Todos los servicios acceden a mismo servidor de bases de datos.
- Base de datos por servicio: cada servicio tiene su propio servidor de base de datos.

Las dos primeras soluciones tienen la sobrecarga más baja en el desarrollo de los microservicios, ya que mantenemos una única servidora de datos. Sin embargo, siguiendo las buenas prácticas aceptadas en el ámbito del diseño y desarrollo de microservicios (cuyo principal estandarte es Netflix, tal y como se puede apreciar por todo el soporte que han proporcionado y hemos utilizado a lo largo de este libro), la mejor solución para el acceso a datos en una arquitectura basada en microservicios, si queremos realmente que éstos sean totalmente independientes entre sí, es que cada microservicio implemente y gestione su propia base de datos. Es decir, cada microservicio es responsable de su propia base de datos privada, al que otros servicios no pueden acceder directamente (ver Figura 9.1). Nótese que, aunque en el ejemplo de la figura cada base de datos está formada por una única tabla, esto no ocurrirá en un proyecto real, donde la base de datos de cada microservicio deberá gestionar un conjunto de tablas con sus propias relaciones y restricciones.

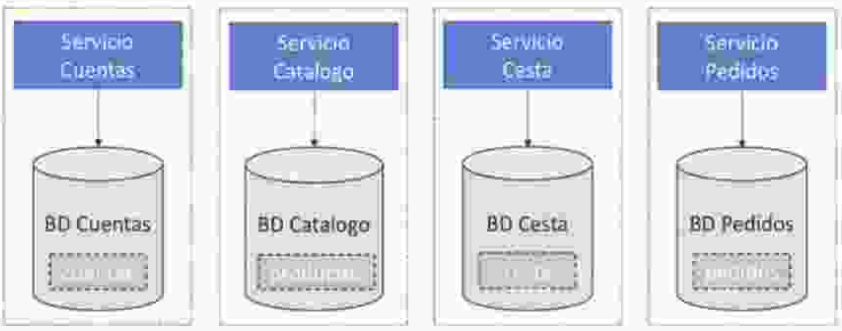

Figura 7.1. Una base de datos para cada microservicio

La solución de utilizar una base de datos por microservicio da lugar a una persistencia políglota, permitiendo el uso de varias tecnologías de almacenamiento de datos dentro de una misma aplicación. Por ejemplo, un microservicio podría hacer uso de una base de datos relacional mientras que otros podrían usar una base de datos NoSQL como MongoDB, para almacenar datos complejos no estructurados, o Neo4J, que está diseñada para almacenar y consultar datos de gráficos de manera eficiente.

Este enfoque distribuido de administración de datos plantea algunos desafíos que hay que tener en cuenta.

- **Datos distribuidos y redundantes**. Los datos son almacenados de forma distribuida a través de los diferentes servicios que componen el sistema. La actualización de los datos de un servicio puede requerir que los datos de otro sean actualizados, por lo que es necesario tener en cuenta cómo se propagan las actualizaciones a través de los servicios.

  También puede resultar necesario que las bases de datos de varios microservicios almacenen una misma información. Por ejemplo, un servicio podría almacenar los datos resultantes de una transacción y, a, además, ser necesario almacenarlos en otra ubicación para su análisis o archivado. En estos casos, los datos duplicados pueden provocar problemas de coherencia e integridad de datos. Así pues, además de la propagación de actualizaciones, se debe tener en cuenta cómo administrar la posible coherencia cuando los datos aparecen en varios lugares.

  Una solución a estos problemas es la implementación de una aproximación basada en eventos. Los servicios publican eventos cuando actualizan datos y otros servicios se suscriben a estos eventos y actualizan sus datos en respuesta.

- **Transacciones distribuidas**. Las transacciones de negocio pueden necesitar acceder a datos gestionados por diferentes microservicios. Por ejemplo, durante la creación de un nuevo pedido (microservicio Pedido) se deberá verificar que no excede el límite de crédito del cliente (microservicio Cuentas) además de actualizar el stock de los productos comprados (microservicio Catálogo).

  Una solución a este problema es implementar el patrón Saga, el cual propone dos soluciones basadas en la coreografía y orquestación de servicios a través de eventos.

- **Relaciones entre datos**. Cabe la posibilidad de que algunas relaciones entre datos abarquen diferentes servicios. Por ejemplo, la relación entre

productos y pedidos se define entre dos tablas que son gestionadas por microservicios diferentes, y, por tanto, deben ubicarse en bases de datos diferentes. Esto ocasiona que no puedan usarse técnicas de administración de datos tradicionales para hacer que se cumplan restricciones de integridad definidas por estas relaciones. Del mismo modo, se complican las consultas sobre datos que son propiedad de múltiples servicios, ya que no es posible hacer un *join* sobre estas tablas (en el caso de que ambos datos se encuentren almacenados en una base de datos relacional). Por ejemplo, encontrar clientes en una región en particular y sus pedidos recientes requiere hacer un *join* entre la tabla cuentas (microservicios Cuentas) y la tabla pedidos (microservicio Pedidos), pero ambas tablas están en bases de datos diferentes.

Para este problema existen dos soluciones que son ampliamente utilizadas: utilizar el patrón API Facade Composition o el patrón Command Query Responsibility Segregation (CQRS).

7.1 ARQUITECTURAS BASADAS EN EVENTOS

Una buena forma de resolver el desafío de administración de datos distribuidos y/o redundantes es mediante el uso de una arquitectura basada en eventos. En un sistema guiado por eventos, los servicios publican y consumen eventos. Un servicio publica un evento cada vez que cambia el estado de una entidad. Otro servicio puede suscribirse a ese evento y actualizar sus propias entidades y, posiblemente, publicar otros eventos.

En el caso de datos redundantes, esto permite que un servicio pueda mantener la consistencia de una réplica suscribiéndose a los eventos que se publican cuando se actualiza la copia maestra. Además, tal y como veremos en el patrón Saga, el uso de eventos también permite la implementación de transacciones de negocio distribuidas que actualicen varias entidades a lo largo de una serie de pasos. Cada paso de la transacción actualiza una entidad y publica un evento que desencadena el siguiente paso.

Una de las soluciones más habituales en la implementación de una arquitectura basada en eventos es utilizar un bróker de mensajería asíncrona que soporte el patrón Publicador/Suscriptor. Un servicio puede notificar los eventos que genera a través de mensajes asociados a un *topic* que son publicados en el bróker. El resto de servicios que estén suscritos a dicho *topic* serán conscientes de estos mensajes cada vez que se publiquen y podrán actuar en consecuencia.

Existen multitud de brókers de mensajería asíncrona que pueden utilizarse en una arquitectura de microservicios. Algunos ejemplos son RabbitMQ, ActiveMQ, WebSphere, Kafka, Kestrel, etc. El uso de este tipo de sistemas queda fuera del objetivo de este libro. Sin embargo, a modo de ejemplo representativo, vamos a ver cómo podríamos publicar un mensaje desde un microservicio o cómo suscribirnos desde otro mediante RabbitMQ.

Supongamos que el microservicio Pedidos ha creado un nuevo pedido y debe publicar un evento para que otros servicios actualicen sus datos de forma pertinente. Por ejemplo, el servicio Catalogo deberá actualizar el stock de los productos incluidos en el pedido. En el siguiente ejemplo, se muestra el código que debería ejecutar el servicio Pedidos para publicar un mensaje en RabbitMQ. En este mensaje se informa del evento producido (*topic* **EVENTS.PEDIDO_CREADO**) y se aporta información sobre el mismo (**productos** incluidos).

```
final String NOMBRE_EXCHANGE = "datos";
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("rabbitserver.rama.es");
Connection connection = factory.newConnection();
Channel channel = connection.createChannel();
channel.exchangeDeclare(NOMBRE_EXCHANGE,BuiltinExchangeType.TOPIC);
String productos=Utils.toJSON(DAO.getPedido(id).getProductos());
channel.basicPublish(NOMBRE_EXCHANGE, "EVENTS.PEDIDO_CREADO",
null, productos.getBytes());
channel.close();
connection.close();
```

Por otro lado, los microservicios que, como Catálogo, necesiten actualizar sus datos cada vez que se cree un pedido, deberán suscribirse al topic **EVENTS.PEDIDO_CREADO** tal y como se muestra en el siguiente ejemplo.

```
final String NOMBRE_EXCHANGE = "datos";
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("rabbitserver.rama.es");
Connection connection = factory.newConnection();
Channel channel = connection.createChannel();
channel.exchangeDeclare(NOMBRE_EXCHANGE, BuiltinExchangeType.
TOPIC);
String COLA_CONSUMER = channel.queueDeclare().getQueue();
channel.queueBind(COLA_CONSUMER, NOMBRE_EXCHANGE,
"EVENTS.PEDIDO_CREADO");
Consumer consumer = new DefaultConsumer(channel) {
```

```
    @Override
    public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
    String message = new String(body, "UTF-8");
    JSONArray productos=new JSONArray(message);
    DAO.updateStockProductos(productos);

    }
};
channel.basicConsume(COLA_CONSUMER, true, consumer);
channel.close();
connection.close();
```

7.2 PATRÓN SAGA

Como se ha comentado anteriormente, las buenas prácticas en el diseño de arquitecturas basadas en microservicios indican que cada microservicio debe gestionar su propia base de datos. Algunas transacciones de negocio, sin embargo, abarcan múltiples servicios, por lo que necesitamos un mecanismo que garantice la coherencia de los datos en todos los servicios.

El patrón Saga propone diseñar una transacción de negocio que abarque múltiples servicios como una saga, siendo una saga es una secuencia de transacciones locales. Cada transacción local actualiza sus datos y publica un mensaje o evento para activar la próxima transacción local en la saga. Si una transacción local falla porque infringe una regla de negocio, la saga ejecuta una serie de acciones de compensación que deshacen los cambios que se realizaron en las transacciones locales anteriores.

Existen dos formas básicas de implementar una transacción en saga:

- Coreografía: no hay una coordinación central. Cada servicio produce y escucha los eventos de otros servicios y decide si se debe tomar o no una acción.
- Orquestación: existe un servicio coordinador que es el responsable de centralizar la lógica de negocio y la toma de decisiones en la secuencia de la saga.

7.2.1 Transacción Saga con coreografía

En esta solución, el primer servicio ejecuta una transacción y luego publica un evento. Este evento es escuchado por uno o más servicios que, en respuesta, ejecutan transacciones locales y publican (o no) eventos nuevos.

La transacción distribuida finaliza cuando el último servicio ejecuta su transacción local y no publica ningún evento o el evento publicado no es escuchado por ninguno de los participantes de la saga.

En la Figura 9.2 se muestra un ejemplo. Las acciones que se realizan son las siguientes:

1. El servicio Pedidos crea un pedido con estado 'en proceso', y genera el evento 'Pedido creado'.
2. El servicio Cuentas está a la escucha de este evento y cuando se publica pasa a realizar el cobro del pedido. Una vez realizado el cobro publica el evento 'Pago realizado'.
3. Al escuchar este último evento, el servicio Catalogo actualiza el stock y publica el evento 'Stock actualizado'.
4. Este evento es escuchado por el servicio Pedidos que finaliza la transacción pasando el pedido a estado 'confirmado'.

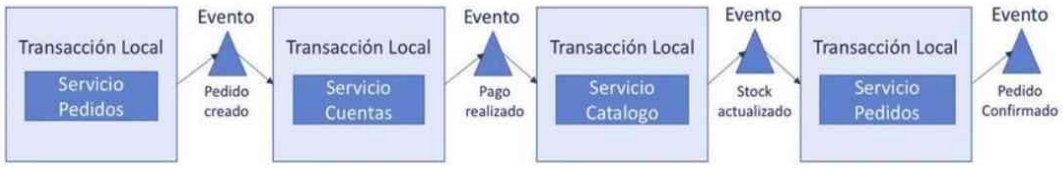

Figura 7.2. Transacción en saga con coreografía

El ejemplo anterior supone que toda la transacción se ejecuta de forma correcta, sin errores o excepciones. Pero, ¿qué sucede en caso de error? En una transacción tradicional, haríamos un *rollback* en la base de datos (en lugar de un *commit*). Sin embargo, esto no es posible o en una transacción Saga. En este caso, se usan eventos de compensación para reaccionar a las excepciones. Si algo sale mal, se produce un evento compensatorio que "cancela" la Saga e informa a los servicios que ya han realizado alguna acción dentro de la transacción para que reviertan estas acciones.

En el siguiente ejemplo vemos lo que ocurre cuando la transacción anterior falla al actualizar el stock. En este caso, el servicio Catalogo comprueba que no existen suficientes unidades en stock para procesar el pedido y lanza un evento compensatorio con el que avisa a los servicios Pedidos y Cuentas. El servicio Pedido actualiza el estado del pedido a 'Cancelado' mientras que el servicio Cuentas cancela el cobro realizado.

Nótese la importancia de realizar un buen diseño de las transacciones distribuidas ya que, si la comprobación de stock se hubiera realizado antes, hubiéramos evitado tener que procesar el pago y luego revertirlo.

Figura 7.3. Rollback en una transacción en saga con coreografía

La coreografía de microservicios a través de la publicación de eventos define una forma natural de implementar el patrón Saga. Es una solución simple, fácil de entender, no requiere mucho esfuerzo para construir, y todos los participantes están ligeramente vinculados ya que no tienen conocimiento directo el uno del otro. Si la transacción que queremos implementar no tiene muchos pasos (de 2 a 4 pasos) constituye una buena opción. Sin embargo, este enfoque puede volverse confuso rápidamente si la transacción que queremos definir tiene un número mayor de pasos, ya que resulta difícil gestionar qué servicios escuchan qué eventos. Además, también corremos el riesgo de crear dependencias cíclicas entre los servicios, ya que tienen que suscribirse a los eventos de los demás.

A la hora de implementar el patrón saga con coreografía se suele hacer uso de un bróker de mensajería. Tal y como se muestra en la Figura 9.4, este bróker se utiliza como medio para publicar los eventos que cada servicio genera a lo largo de una transacción, de forma que el resto de servicios suscritos pueden escucharlos.

Se utiliza el patrón Publicador/Suscriptor de modo que los servicios publican los eventos asociados a un *topic* (la transacción) y los servicios que estén suscritos a este *topic* (aquellos que participan en la transacción) los reciben y actúan en consecuencia. Es recomendable que el mensaje que cada servicio publique en el bróker incluya, además del evento generado, el identificador de la transacción en cuestión, de modo que los servicios suscritos puedan identificarla fácilmente.

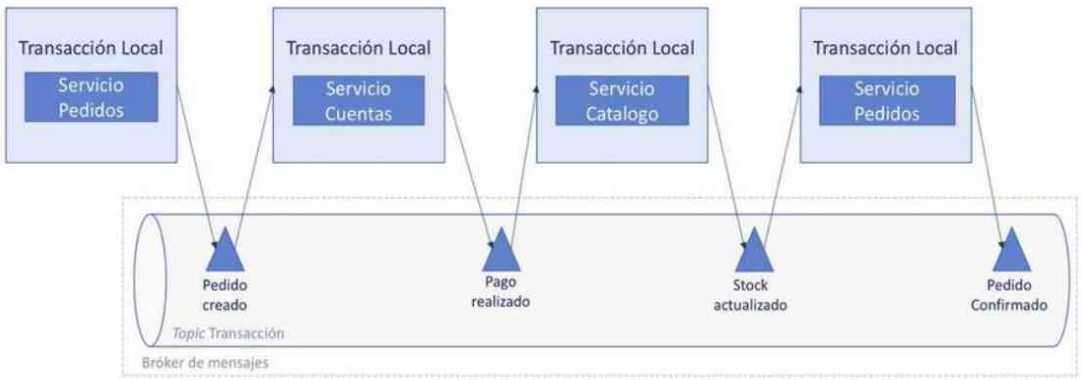

Figura 7.4. Transacción en saga con coreografía y bróker de mensajería

7.2.2 Transacción Saga con orquestación

En la solución basada en orquestación, debemos crear un nuevo servicio cuya única responsabilidad es decirle a cada participante de una transacción qué debe hacer y cuándo hacerlo. La transacción es iniciada por un microservicio a través del orquestador, al cual le cede inmediatamente toda la responsabilidad. El orquestador de la saga se comunica con cada servicio mediante un estilo de comando/respuesta, indicando a cada uno de ellos lo que debe hacer, según la secuencia que define la transacción. Cuando un servicio termina, responde al orquestador informándole de ello, de modo que el orquestador puede ponerse en contacto con el servicio que debe realizar la siguiente acción de la transacción.

En la Figura 9.5 se muestra un ejemplo. Las acciones que se realizan son las siguientes:

1. El servicio Pedidos crea un pedido con estado 'en proceso' e inicia la transacción ProcesaPedidoTX a través del Orquestador.

2. El Orquestador indica al servicio Cuentas de debe realizar el cobro del pedido. Una vez realizado el cobro el servicio Cuentas responde al Orquestador informando sobre ello.

3. A continuación, el Orquestador indica a Catálogo que actualice el *stock*. Cuando este servicio ha realizado esta acción informa al Orquestador sobre ello.

4. Por último, el Orquestador indica al servicio Pedidos que actualice el estado del pedido a 'confirmado'. Una vez hecho esto, el servicio informa al orquestador y éste finaliza la transacción.

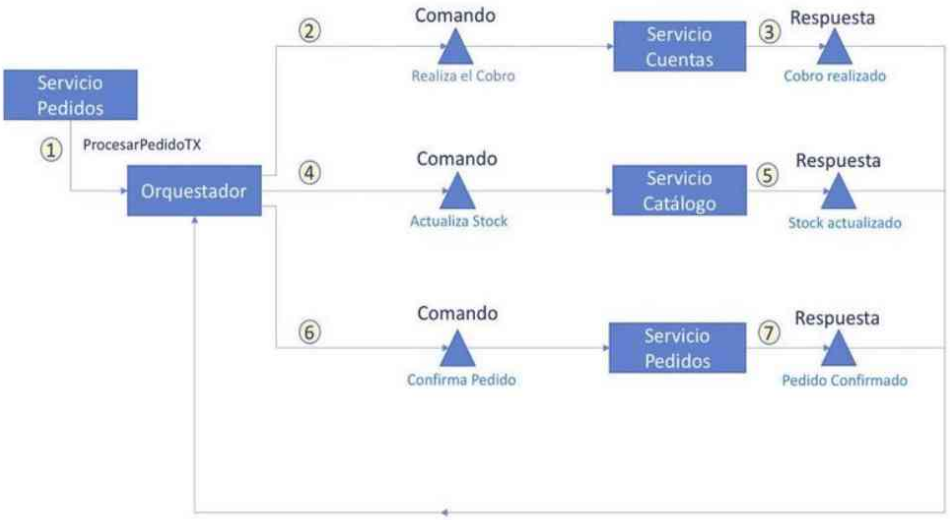

Figura 7.5. Transacción en saga con orquestador

Como puede apreciarse en el ejemplo anterior, el Orquestador es el único que sabe cuál es el flujo necesario para ejecutar una transacción. Si algo falla, también es responsable de coordinar la reversión, enviando comandos a cada participante para deshacer la operación anterior. Tal y como se muestra en la Figura 9.6, cuando el servicio Catálogo falla al actualizar el stock, informa de ello al orquestador para que revierta las acciones de la transacción. En este caso, el orquestador indicará al servicio Cuentas que devuelva el cobro y al servicio Pedidos que actualice el estado del pedido a 'Cancelado'.

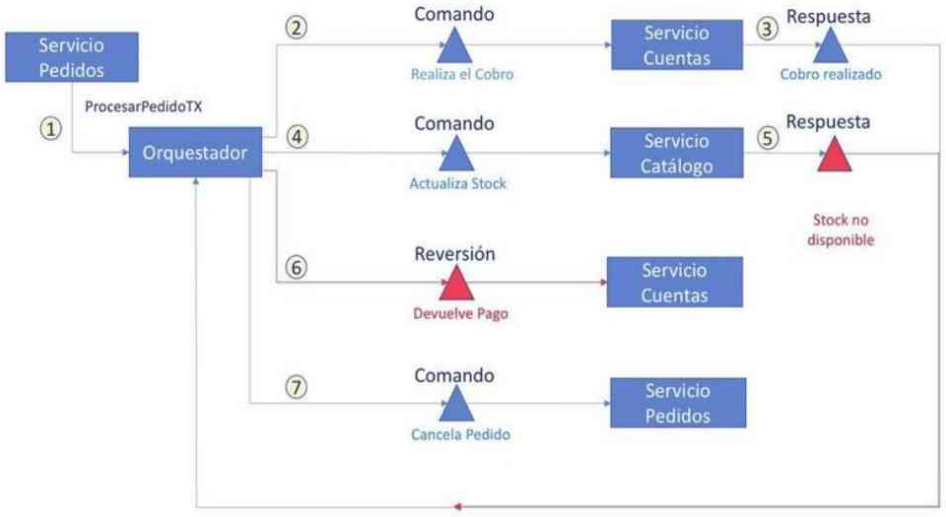

Figura 7.6. Rollback en una transacción en saga con orquestador

El uso de un orquestador para el diseño de transacciones Saga tienen diferentes de beneficios:

- Evita las dependencias cíclicas entre servicios, ya que el orquestador de sagas invoca a los participantes de la saga, pero los participantes no invocan al orquestador.
- Centraliza la orquestación de la transacción distribuida.
- Reduce la complejidad de los participantes ya que solo necesitan ejecutar / responder comandos.
- Más fácil de implementar y probar que mediante el uso de coreografías.
- La complejidad de la transacción permanece lineal cuando se agregan nuevos pasos.
- Los rollbacks son más fáciles de administrar que mediante el uso de coreografías.
- Facilita la puesta en espera de transacciones cuando intentan acceder a un mismo recurso.

Sin embargo, este enfoque todavía tiene algunos inconvenientes. El más importante es el riesgo de concentrar demasiada lógica en el orquestador y terminar con una arquitectura donde un orquestador inteligente le dice a unos servicios tontos qué hacer. Otra desventaja es que aumenta ligeramente la complejidad de la infraestructura que soporta a los microservicios, ya que necesitará administrar un servicio adicional.

Igual que ocurría en el caso anterior, en la implementación del patrón Saga con orquestador suele hacerse uso de algún bróker de mensajería. En este caso, y tal y como se muestra en la Figura 9.7, este bróker mantiene una cola Punto a Punto (los mensajes tienen un único destinatario a diferencia del modelo Publicador/Suscriptor donde un mensaje es enviado a todos los suscriptores) para cada servicio, de forma que el orquestador puede utilizarlas para enviar a cada servicio los comandos a realizar de forma asíncrona. Del mismo modo, se define una cola en la que los microservicios envían sus respuestas al orquestador.

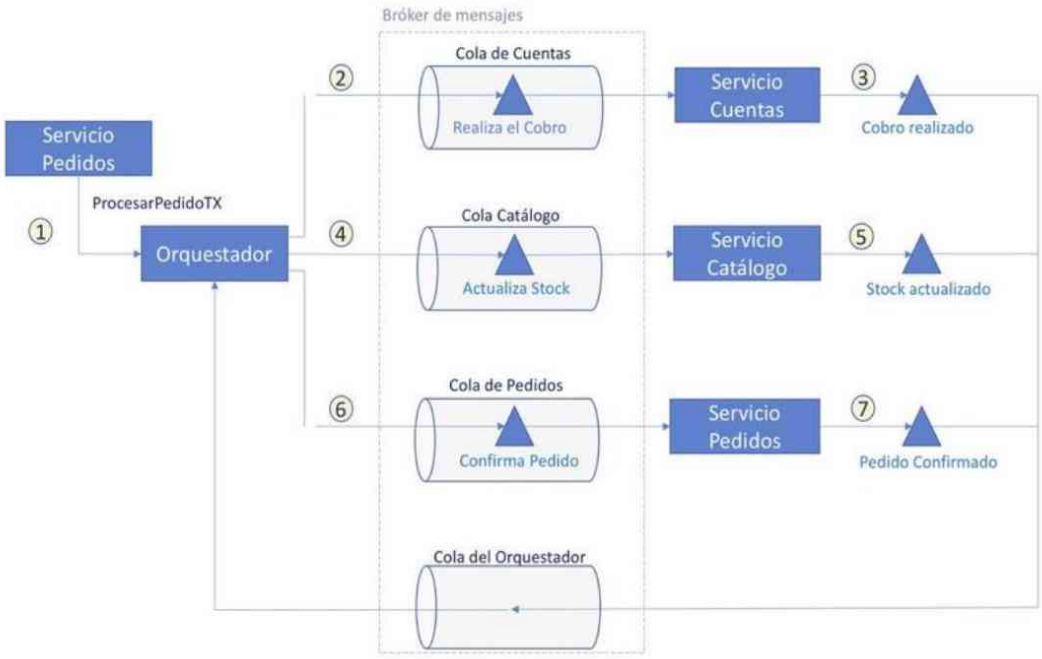

Figura 7.7. Transacción en saga con orquestador y bróker de mensajería

7.3 CONSULTAS SOBRE DATOS RELACIONADOS

Al aplicar las buenas prácticas en el diseño de una arquitectura de microservicios, hemos visto que cada uno de ellos debe gestionar su propia base de datos de forma independiente. Esto hace que las consultas sobre datos gestionados por diversos microservicios se compliquen, ya que no es posible realizar un *join* de tablas como haríamos en sistemas con una única base de datos.

Por ejemplo, supongamos que hemos desarrollado una aplicación Web para dar soporte a nuestra tienda online basada en microservicios. El objetivo de esta aplicación es proporcionar a los usuarios todos los productos en catálogo y permitirles la compra de los mismo. Para ello, es probable que necesitemos tener páginas en las que se muestren datos que estén gestionados por diversos microservicios y por lo tanto tengamos que hacer una consulta sobre tablas de diferentes bases de datos. Por ejemplo, la página de un producto puede mostrar la información que está disponible sobre el mismo en la tabla **productos** (servicio Catalogo) pero también puede mostrar información sobre productos similares que hayan comprados otros usuarios, para lo cual tendremos que acceder a las tabla **pedidos** (microservicio Pedidos).

Dado que no es posible realizar una consulta conjunta sobre todas las tablas (hacer un *join*) deberemos buscar una solución alternativa. Las dos soluciones más utilizadas para afrontar este problema son aplicar el patrón API Facade Composition o el patrón Command Query Responsibility Segregation (CQRS).

7.3.1 API Facade Composition

Partimos del hecho de que no podemos recuperar en una sola consulta todos los datos que nos solicita la aplicación cliente (nuestra Web) ya que, como ya se ha comentado, los datos están en tablas gestionadas por diferentes microservicios. Además, ya hemos explicado que la única forma de acceder a los datos de un microservicio debe ser a través de la API que éste pública.

Así pues, si queremos satisfacer la solicitud de la aplicación cliente deberemos acceder a la información necesaria accediendo a través de la API proporcionada por cada microservicio y a continuación componer los datos en memoria. Esta tarea podemos dejarla bajo la responsabilidad de la aplicación cliente o proporcionar algún mecanismo para que el acceso del cliente a los datos que necesita sea posible a través de una única conexión.

El patrón API Facade Composition propone componer las consultas necesarias por los clientes accediendo a través de las API de los microservicios y publicar estas consultas en una nueva API. La Figura 7.8 ilustra esta solución. En este caso la nueva API permite a las aplicaciones clientes acceder a la vista de un producto a través de una única conexión. El elemento que implementa la API (el *Composer)* se encarga de crear las diferentes conexiones sobre los microservicios Catálogo y Pedidos, y componer la información del producto solicitado y de los productos relacionados que otros usuarios han comprado.

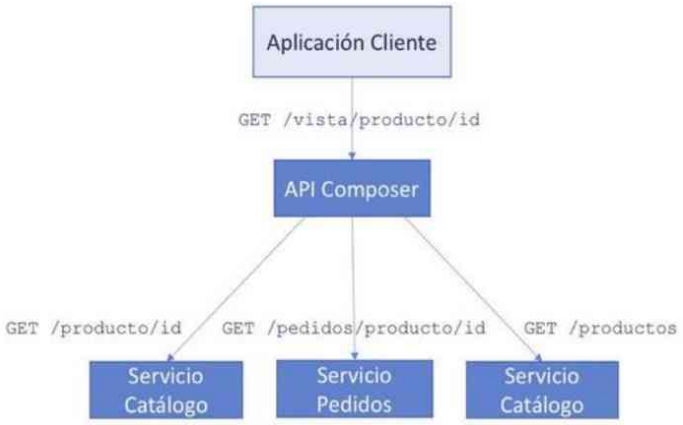

Figura 7.8. Patrón Facade Composition

Al aplicar este patrón tenemos que decir qué elemento actúa como API *Composer*. Considerando que en la arquitectura que hemos montado hasta ahora disponemos de un Gateway implementado con Zuul, una opción podría ser delegar en Zuul las tareas de composición de consultas. Sin embargo, Zuul no nos proporciona una solución sencilla para realizar esta tarea, ni siquiera a través de sus filtros, ya que el principal objetivo de Zuul es el de redireccionar peticiones.

Así pues, la solución más utilizada en estas situaciones es la de implementar un nuevo servicio por cada aplicación cliente a la que demos soporte. Cada uno de estos servicios se encargarán de crear una API que proporciona los datos que cada aplicación necesita. Zuul se encargará de centralizar el acceso a estos servicios y redireccionar la solicitudes de cada aplicación. Este escenario se muestra en la Figura 7.9.

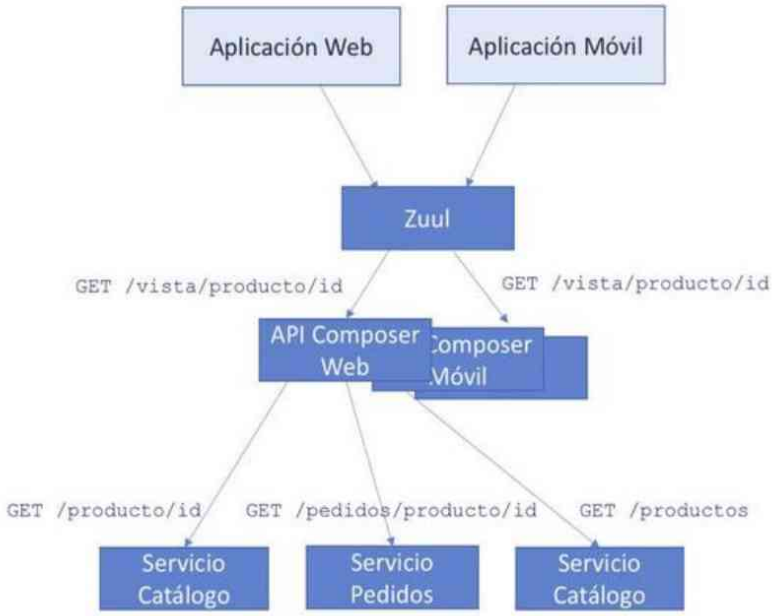

Figura 7.9. Nuevos servicios para la composición de consultas

En el desarrollo de los servicios encargados de la composición de consultas, éstos deben invocar al resto de servicios en paralelo siempre que sea posible, con el fin de minimizar el tiempo de respuesta para una operación de consulta. Si no hay dependencias en las llamadas a servicios (para invocar a un servicio no necesitamos la información proporcionada por otro) podremos acceder a todos ellos de forma paralela. En otras ocasiones, sin embargo, necesitaremos el resultado de un servicio para poder invocar otro servicio. En este caso, tendremos que invocar secuencialmente algunos de los servicios necesario para componer la consulta, aunque no todos.

El patrón API Facade Composition constituye una solución simple e intuitiva de implementar operaciones de consulta en una arquitectura de microservicio. Como principales ventajas podemos destacar:

- Las aplicaciones cliente sólo necesitan realizar una conexión para recuperar los datos a visualizar en una página web o pantalla.
- Podemos cambiar el elemento que proporciona los datos a cada aplicación cliente (servicio Composer) sin que éstas tengan que ser modificadas.
- Posible control y validación adicional antes de que una solicitud cliente acceda a los microservicios de nuestro sistema.
- Facilidad para generar datos con el objetivo de su análisis y/o obtención de estadísticas.

Sin embargo, también presenta algunos inconvenientes:

- Costosas e ineficientes uniones en memoria de grandes conjuntos de datos. En una aplicación monolítica, un cliente puede recuperar todos los datos que necesita con una única consulta a la base de datos. En comparación, con el patrón API Facade Composition debemos realizar múltiples consultas y componerlas en memoria, lo cual puede resultar excesivamente costoso si se manejan grandes cantidades de datos.
- Mayor sobrecarga al invocar múltiples servicios. El uso de este patrón implica múltiples solicitudes a diferentes microservicios, para lo cual se requieren más recursos informáticos y de red. Igual que ocurría en el caso anterior, esto puede aumentar el coste de ejecución de la aplicación.
- Riesgo de disponibilidad reducida. La disponibilidad de una operación disminuye con la cantidad de servicios involucrados. Cuantas más invocaciones a servicios necesite hacerse en una operación de consulta, mayor riesgo de que alguno de ellos esté caído y que, por lo tanto, no pueda satisfacerse la consulta. En este sentido, la disponibilidad de las operaciones del API *Composer* será significativamente menor que las de un solo servicio.
- Falta de coherencia en datos transaccionales. Una aplicación monolítica normalmente ejecuta una operación de consulta utilizando una única transacción de base de datos. Estas transacciones se rigen por los principios ACID (acrónimo anglosajón para Atomicidad, Consistencia, Aislamiento y Durabilidad) de forma que garantizan que una aplicación

tenga una vista coherente de los datos al finalizar la transacción, incluso si se ejecutan múltiples consultas de bases de datos. Por el contrario, con el patrón API Facade Composition no siempre resulta fácil garantizar estos principios, por lo que existe el riesgo de que una operación de consulta devuelva datos incoherentes si existe otra transacción en curso que los está modificando.

7.3.2 Command Query Responsibility Segregation (CQRS)

El patrón CQRS propone la separación de las operaciones de lectura de datos y las de actualización a través de APIs independientes. En concreto, la idea básica es dividir las operaciones en dos categorías distintas:

- Consultas: operaciones que devuelven un resultado y no cambian el estado del sistema.
- Comandos: operaciones que cambian el estado del sistema, pero no devuelven valores.

Las consultas y los comandos están desacoplados, de modo que el enlace entre ellos se realiza a través de eventos generados por el procesamiento de comandos, que alimentan los repositorios para datos de consulta.

Al aplicar estas ideas en una arquitectura de microservicios, pasamos de tener microservicios encargados de las operaciones de lectura y escritura sobre un conjunto de datos, a tener microservicios especializados en operaciones de escritura o de consulta. Más concretamente, se propone crear uno o más microservicios que dupliquen datos de múltiples servicios con el objetivo de facilitar su consulta (por ejemplo, permitiendo consultas con *joins*). Estos servicios de consulta están suscritos a los eventos que el resto de microservicios (cuyo objetivo es procesar comandos, operaciones de escritura) generan cuando actualizan sus datos con el fin de mantener los datos duplicados consistentes.

La Figura 7.10 ilustra este diseño de microservicios implementado mediante un bróker de mensajería, tal y como ya hemos hecho en casos anteriores. En este caso, el servicio Consultas mantiene en su propia base de datos una copia de los datos manejados por los servicios Catálogo, Pedidos y Cuentas. Esto le permite mantener las relaciones y restricciones entre estas tablas en una única base de datos, facilitando así las consultas que requieren hacer un *join* sobre ellas. Este nuevo servicio únicamente se centra en satisfacer las consultas que las aplicaciones cliente puedan realizar al sistema a través del Gateway Zuul. Las operaciones de

actualización de datos son procesadas por los servicios propietarios de los mismos (es decir, los servicios Catálogo, Pedidos y Cuentas). Cuando uno de estos servicios realiza una operación de actualización genera el evento correspondiente para que el servicio de Consultas actualice sus datos.

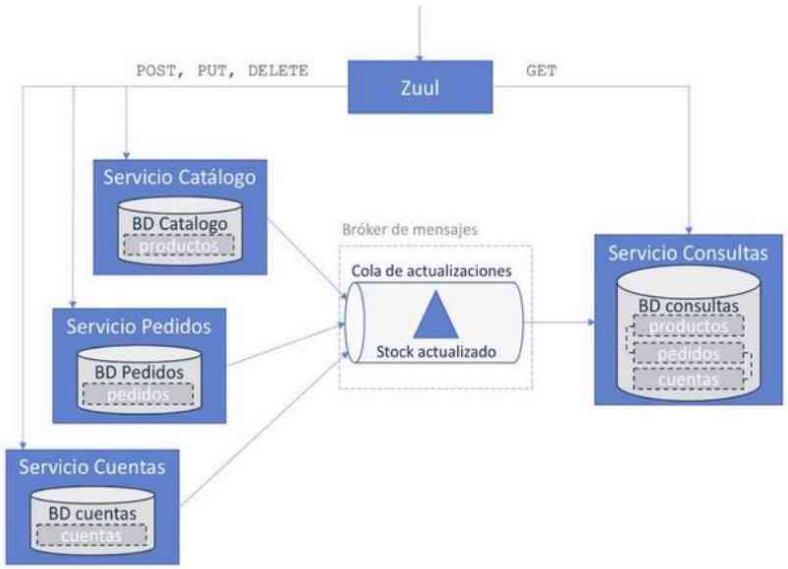

Figura 7.10. Patrón CQRS para el procesamiento de consultas complejas

Tal y como podemos apreciar, una de las ventajas más importantes de aplicar esta solución es que permite la implementación eficiente de consultas que recuperan datos que son propiedad de múltiples servicios. Ya se comentó en la sección anterior que el patrón API Facade Composition puede necesitar costosas e ineficientes uniones de datos en memoria cuando se implementan consultas sobre grandes cantidades de datos de diferentes servicios. Para estas consultas, es más eficiente utilizar CQRS, ya que se puede consultar fácilmente todos los datos desde una única consulta sobre una base de datos.

Otra ventaja de este patrón reside en el hecho de que mejora la separación de aspectos al definir módulos de código separados (diferentes servicios) para los comandos (operaciones de escritura) y las consultas (operaciones de lectura). La separación de aspectos proporciona una manera más simple y fácil de mantener los servicios de comandos y de consultas.

En cuanto a sus inconvenientes, podemos destacar dos, principalmente. Por un lado, añade mayor complejidad a la arquitectura, ya que se debe crear un repositorio adicional para duplicar los datos a consultar, y programar todo el código necesario para la generación de eventos y de procesamiento de los mismos con el fin

de mantener este repositorio actualizado. Por otro lado, aparece el problema de tratar con el retraso entre que un servicio comando actualiza sus datos y estos cambios se propagan al servicio consulta. Cada vez que un servicio comando modifica sus datos debe generar el correspondiente evento y éste debe ser procesado por los servicios consulta. Así pues, si una aplicación actualiza unos determinados datos e inmediatamente realiza una consulta sobre los mismos, es posible que consulte los datos anteriores, sin actualizar. Una solución a esto es que los servicios comando y de consulta proporcionen al cliente información sobre la versión que le permita detectar que los datos de una consulta están desactualizados.

8

TESTING DE MICROSERVICIOS

La arquitectura de microservicios es un concepto estrechamente ligado a la entrega continua. Se trata de un nuevo enfoque según el cual los ciclos de desarrollo se acortan con el objetivo de que el software pueda entrar en producción de manera fiable en cualquier momento. La entrega continua es un proceso basado en la automatización de tres etapas (ver Figura 10.1):

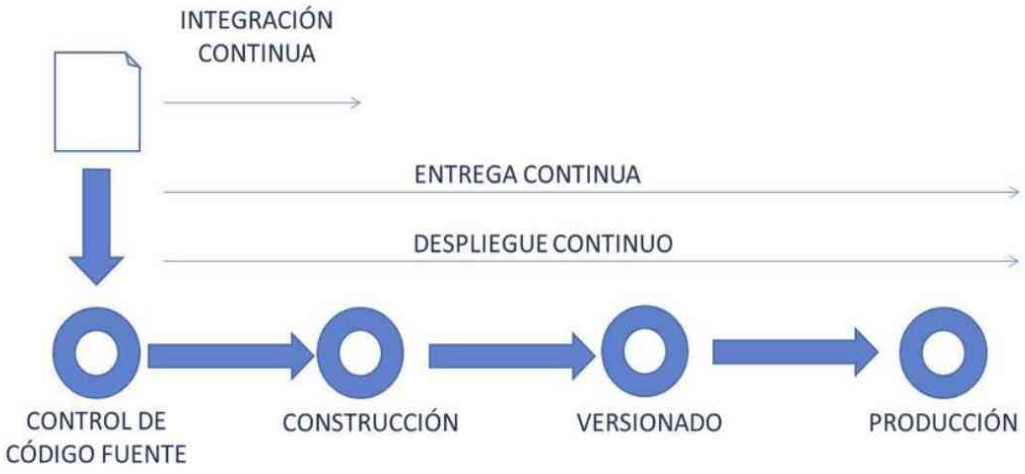

Figura 8.1. Entrega continua

- Automatización de la compilación e integración continua: a medida que se va generando el código fuente se va compilando e integrando en el código central de la aplicación.

- Automatización de las pruebas: consiste en probar, de manera rigurosa, la nueva versión de la aplicación con el fin de garantizar que se cumplen los requisitos de calidad establecidos. Hay que destacar que no se trata únicamente de las pruebas unitarias, sino también del resto de niveles de pruebas.

- Automatización de la implementación: las revisiones se implementan en un entorno de producción sin la aprobación explícita del desarrollador automatizando así la publicación del software de manera que las nuevas funcionalidades o la solución a los fallos están disponibles en un tiempo reducido.

Como puede verse, un concepto clave es el de automatización. En efecto, en un entorno con cientos de microservicios el desarrollo, prueba y despliegue de los mismos de manera manual es un proceso inabordable desde el punto de vista práctico.

En cualquier proyecto de desarrollo de software las pruebas automáticas son cruciales, ya que mejoran la calidad y optimizan el tiempo de puesta en producción, a la vez que proporcionan una mayor cobertura de casos de prueba y agilizan los tiempos de *test*, con la ventaja añadida de que en algunos casos es posible reutilizarlos. Si, además, consideramos un entorno tan complejo como es el de los microservicios, las pruebas manuales quedan descartadas por ser tediosas y tendentes a errores.

Resulta esencial que una aplicación basada en microservicios se construya siendo consciente de cómo probará. Una buena cobertura de pruebas supone una mayor fiabilidad del código y garantiza mejores resultados en los ciclos de entrega continua.

8.1 LA PIRÁMIDE DE COHN

Uno de los problemas clásicos de la automatización de pruebas de software en general y de las arquitecturas basadas en microservicios en particular es llegar a un compromiso entre la exhaustividad de las mismas y su viabilidad. La llamada pirámide de Cohn (ver Figura 2) ofrece una respuesta a este dilema, representar

gráficamente el número de pruebas y el esfuerzo en automatización de manera relativa a cada uno de los cinco niveles de pruebas estudiados.

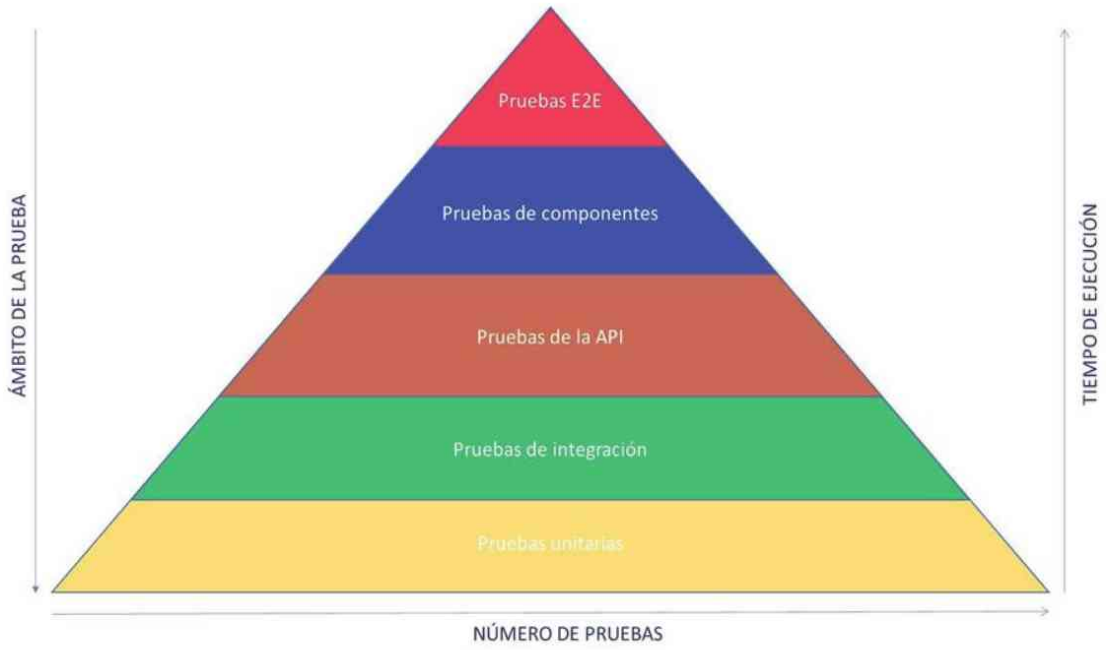

Figura 8.2. Pirámide de Cohn

Según Cohn, cuanto más abajo en la pila de niveles nos movemos, más pruebas son necesarias y menos tiempo de ejecución es necesario. De la misma manera, ascender por los peldaños de la pirámide implica disminuir el número de pruebas a costa de aumentar la complejidad de las mismas.

Llegados a este punto, hay que llamar la atención sobre uno de los peligros más devastadores de las pruebas automáticas: la pérdida de foco. Efectivamente, si no se planifica la estrategia de pruebas adecuadamente, resulta muy sencillo invertir tiempo y recursos en la automatización en un nivel y un grado no adecuados. Es lo que se conoce como patrón del cono de helado (ver Figura 3) y consiste, básicamente, en todo el contrario a lo que establece la pirámide de Cohn.

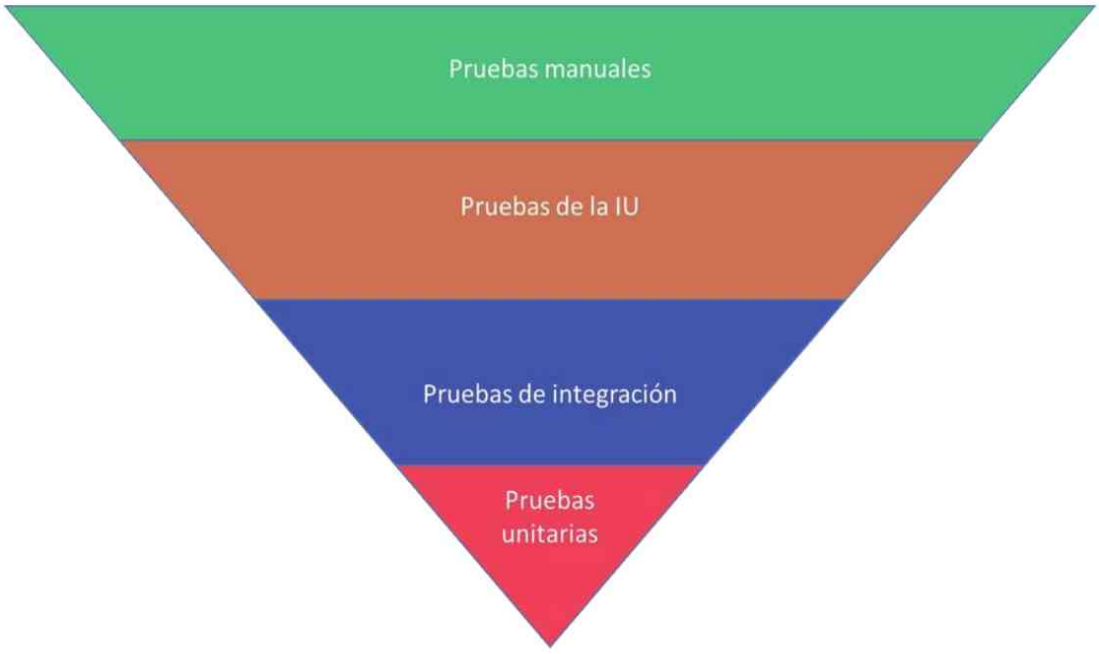

Figura 8.3. Estrategia del cono de helado

Finalmente, no hay que olvidar que una buena estrategia de automatización implica, no sólo escribir las pruebas y documentarlas, si no también crear un *framework* de pruebas sólido, parametrizarlo, etc. Este esfuerzo, y el coste que conlleva, no son nada despreciables.

8.2 NIVELES DE PRUEBAS

En una aplicación diseñada según el paradigma de los microservicios se distinguen varios niveles de granularidad. En primer lugar, los usuarios finales interaccionarán con el sistema como un todo. En el siguiente nivel, dicho sistema es un conjunto de servicios que se comunican entre sí. Finalmente, los propios servicios están formados por módulos de código o componentes. Por tanto, cualquier estrategia de pruebas de ser capaz de cubrir todos estos niveles, así como las comunicaciones entre los mismos. La Figura 4 muestra los cinco niveles de pruebas y en qué parte deben aplicarse en una arquitectura de microservicios.

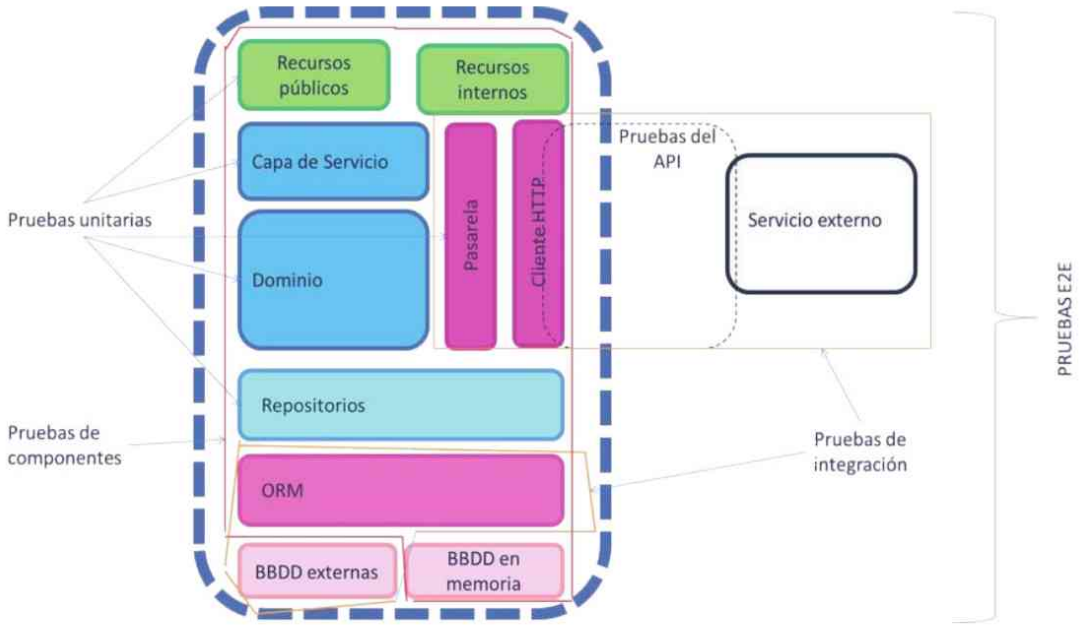

Figura 8.4. Niveles de prueba en una arquitectura de microservicios

8.2.1 Pruebas unitarias

La base de cualquier estrategia de pruebas son las pruebas unitarias. Se centran en verificar una única clase o un conjunto de clases fuertemente acopladas y pueden ejecutarse utilizando objetos reales o bien objetos diseñados específicamente para la prueba (*mock objects*).

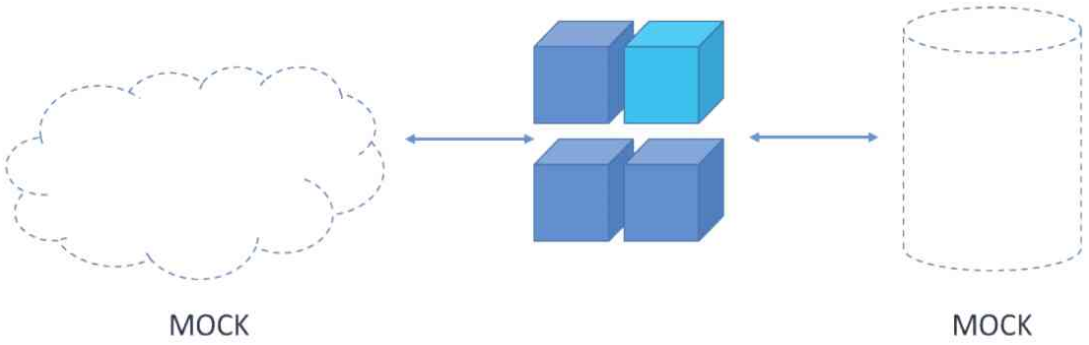

Figura 8.5. Esquema de una prueba unitaria

En una arquitectura de microservicios, el ámbito de las pruebas unitarias se limita al funcionamiento interno del propio microservicio. En términos de volumen de pruebas, son las más numerosas y las más fácilmente autorizables, dependiendo del lenguaje de programación y del *framework* del microservicio.

A menudo, las dificultades que se encuentran al escribir una prueba unitaria hacen emerger la necesidad de dividir un módulo en fragmentos independientes que se puedan probar individualmente. Por tanto, además de una estrategia de pruebas útil, las pruebas unitarias también son una herramienta de desarrollo muy potente, especialmente en desarrollos dirigidos por pruebas o TDD (*Test Driven Development*).

En general, las pruebas unitarias tienen por objetivo determinar si el software que se está probando funciona como se espera o no. Incluso a este nivel tan bajo de pruebas, el código se basa en la colaboración con otros módulos, por lo que cabe distinguir entre pruebas en solitario y pruebas sociales (ver Figura 6). En el primer caso, las pruebas unitarias se denominan en solitario y se centran en la interacción y colaboraciones entre un objeto y sus dependencias. Por otra parte, las pruebas unitarias sociales hacen hincapié en el comportamiento de los módulos observando sus cambios de estado y consideran al módulo bajo pruebas como una caja negra de que la que se verifica su interfaz. Ambas aproximaciones no son excluyentes.

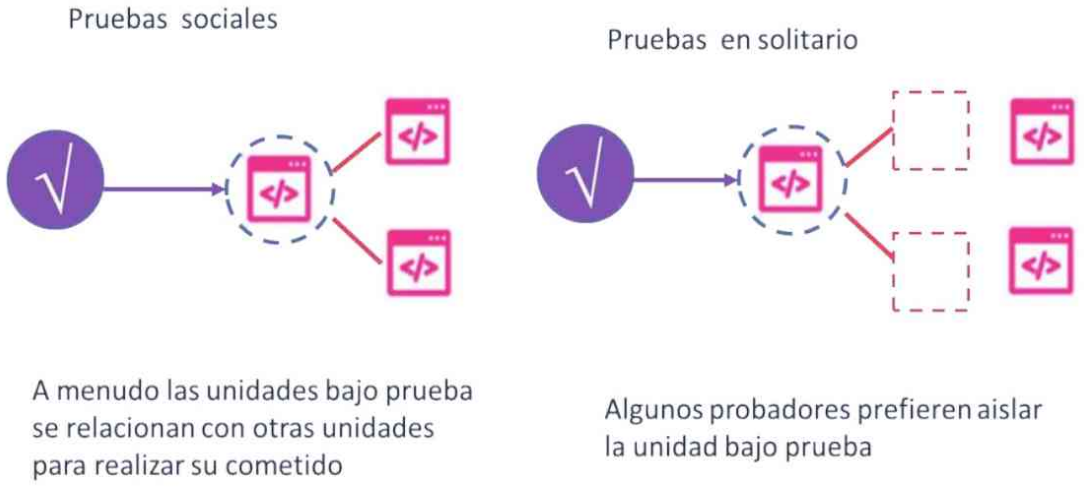

Figura 8.6. Pruebas sociales vs pruebas en solitario.

La Figura 7 muestra los puntos de una arquitectura de microservicios en que resultan más adecuadas las pruebas unitarias, distinguiendo entre pruebas en solitario y pruebas sociales:

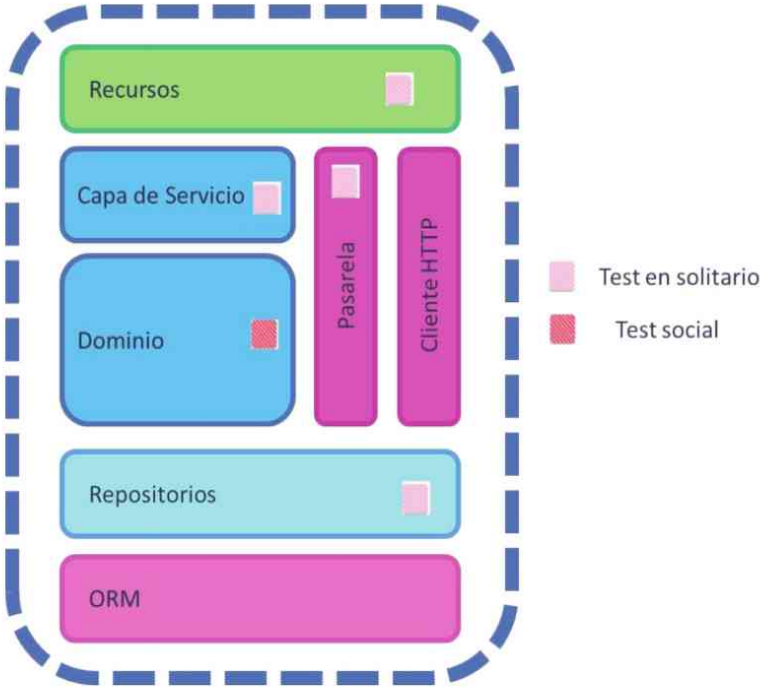

Figura 8.7. Pruebas unitarias en una arquitectura de microservicios.

- Servicios: los microservicios suelen contener gran cantidad de código de coordinación y de trabajo "de fontanería".
- Dominio: la lógica del dominio a menudo se caracteriza por cálculos complejos y una colección de transiciones entre estados. Puesto que estos tipos de lógica se basan en dichos estados tiene muy poco sentido aislar unidades, es decir, que los objetos del dominio real deben ser usados por los colaboradores de la unidad bajo prueba.
- Repositorios y pasarelas: con el código de fontanería es muy difícil tanto aislar la unidad bajo prueba de los módulos externos como verificar el comportamiento frente a cambios de estado. Por tanto, las pruebas unitarias son doblemente efectivas.

  El propósito de las pruebas unitarias a este nivel es verificar cualquier lógica empleada para producir peticiones o mapear recursos de dependencias externas en lugar de verificar la comunicación de una manera integrada. De este modo, las pruebas proporcionan una manera de controlar el ciclo petición-respuesta de una manera fiable y repetible.

  Las pruebas unitarias a este nivel proporcional una realimentación más rápida que las pruebas de integración y pueden forzar condiciones de

error mostrando el comportamiento de las dependencias externas en estas circunstancias excepcionales.

- Recursos y capa de servicio: la lógica de coordinación se encarga más de los mensajes pasados entre los módulos que de cualquier lógica compleja incluida en ellos. Las pruebas permiten verificar los detalles de los mensajes intercambiados y el flujo de comunicaciones que desencadenan en el módulo que se está probando.

  El que una pieza de lógica de coordinación requiera excesivos mensajes suele ser un buen indicador de que algún concepto debe extraerse y probarse de manera aislada.

A medida que el tamaño del servicio disminuye, la relación entre el código de fontería y coordinación y el código de lógica de dominio compleja aumenta. De la misma manera, algunos servicios contendrán únicamente código de fontanería y coordinación, como los adaptadores a distintas tecnologías o los agregadores de servicios. En estos casos, las pruebas unitarias no aportan mucho valor añadido, siendo preferible invertir tiempo en las pruebas de componentes, por ejemplo. Por ello, es muy importante cuestionarse continuamente el valor que las pruebas unitarias proporcionan frente al coste que tiene el mantenerlas o las restricciones de implementación que imponen.

Las pruebas unitarias, por sí mismas, no garantizan el comportamiento correcto del sistema, incluso aunque tengamos una buena cobertura para cada uno de los módulos del sistema aislados. Sin embargo, no hay cobertura para aquellos módulos que cuando trabajan juntos forman un servicio completo o que únicamente interaccionan con dependencias remotas.

Por tanto, para verificar que cada módulo interactúa correctamente con sus colaboradores es necesario un nivel más fino de pruebas.

8.2.2 Pruebas de integración

Cualquier aplicación que no sea trivial no estará aislada del resto del mundo y, de una manera u otra, interactuará con bases de datos, sistemas de ficheros u otros servicios, por ejemplo. Estas interacciones o integraciones tendrán que ser verificadas también y aquí es donde entran en juego las pruebas de integración. En general, se deben diseñar pruebas de integración para cualquier parte del código en el que se serialice o deserialice información, algo que en un entorno de microservicios ocurre más de lo que a primera vista puede parecer. Algunos ejemplos son llamadas a un API REST, lectura/escritura en base de datos o un sistema de ficheros o invocaciones a otros microservicios.

En comparación con las pruebas de unitarias, las pruebas de integración son más finas gradualmente, puesto que su propósito es verificar que cada módulo interactúa correctamente con sus colaboradores, detectando cualquier defecto en cómo los módulos se comunican entre sí con el fin de garantizar que cualquier nueva funcionalidad u otro cambio en el código puede desplegarse de manera segura en un entorno de producción.

Este punto es crítico en una arquitectura de microservicios y se debe realizar una verificación individual de cada servicio, ya que el éxito depende del buen funcionamiento de las comunicaciones entre servicios. Las llamadas a los servicios deben realizarse integradas con servicios externos, incluyendo tanto casos de éxito como de error. Las pruebas de integración, por tanto, validan que el sistema esté trabajando perfectamente y que las dependencias entre servicios son las esperadas y proporcionan una realimentación rápida a la hora de refactorizar o extender la lógica contenida en los módulos de integración. Sin embargo, también tienen más de una razón para fallar, si la lógica del módulo de integración sufre una regresión o si los componentes externos dejan de estar disponibles o rompen su contrato.

Tras las pruebas unitarias y de integración podemos estar seguros de la corrección de la lógica contenida en los módulos individuales que forman parte del microservicio. Sin embargo, de nuevo es necesario un mayor nivel de granularidad que satisfaga los requerimientos del negocio.

8.2.3 Pruebas de la API

Cuando se implementa y se mantiene un servicio, es esencial que la API expuesta por el servicio (también llamada contrato de servicio) cumpla con los requisitos y las expectativas de todos sus consumidores. Un contrato de servicio consiste en entradas y salidas de datos esperadas, los efectos colaterales y otros puntos no funcionales como las prestaciones y la concurrencia. Si un microservicio puede cambiar a lo largo del tiempo, es importante que los contratos de servicio establecidos con cada uno de sus consumidores se respeten. En este sentido, las pruebas de la API aseguran que las fronteras entre servicios se comportan como se supone. Estas interacciones entre microservicios son las más heterogéneas de todas ya que tienen lugar a través del más variopinto conjunto de tecnologías y tanto podemos encontrarnos REST y JSON sobre HTTPS, como llamadas a procedimiento remotos o sistemas de mensajería asíncrona.

Las pruebas de la API, por su propia concepción, se dividen dos partes (ver Figura 8). Por un lado, las pruebas de las aplicaciones clientes que consumen el API y que abarcan aquellos aspectos del servicio (productor) empleados por el cliente. Y, por el otro, las pruebas del servicio que ejecutan las pruebas de la API para determinar el impacto de los cambios en los consumidores.

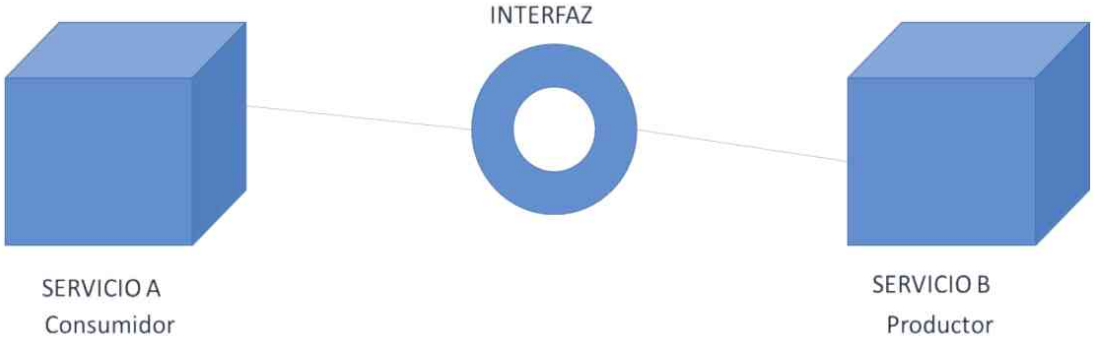

Figura 8.8. Actores implicados en una prueba de la API

La naturaleza de las pruebas de la API las convierte en valiosas cuando se define un nuevo servicio puesto los consumidores de un servicio pueden ayudar al diseño de la API durante las primeras etapas del proceso de diseño (desarrollo CDC, *Consumer-Driven Contracts*). El equipo encargado del consumidor escribe pruebas automáticas que reflejen las expectativas del consumidor y las publica. Hecho esto, el equipo encargado del productor ejecuta continuamente esas pruebas y, en caso de que no se superen, entre ambos equipos se analiza la causa, se propone una solución y comienza una nueva iteración.

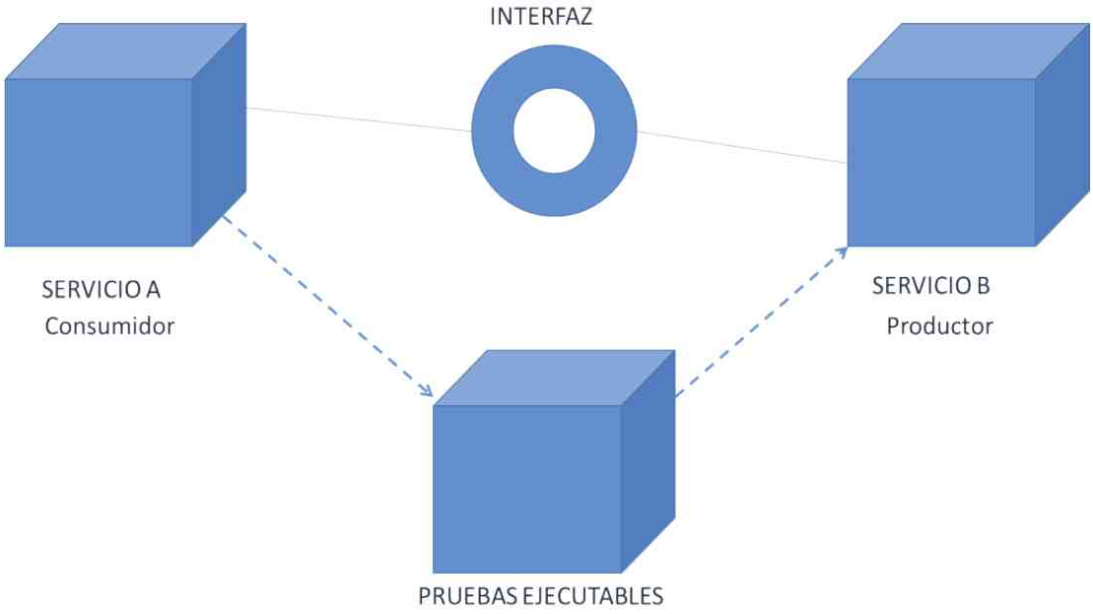

Figura 8.9. Desarrollo CDC

Esta aproximación permite que el equipo de desarrollo del productor se centre sus esfuerzos en aquello que es estrictamente necesario.

8.2.4 Pruebas de componentes

En un sistema compuesto por un conjunto de microservicios, a este nivel de pruebas, la unidad de test es cada uno de los servicios. Las pruebas de componentes proporcionan un entorno de pruebas controlado para el servicio con el objetivo de disparar cualquier posible error de manera contenida.

Por todo lo anterior, el ámbito de estas pruebas se limita, intencionadamente, a un único servicio, lo que significa que dicho servicio se configura aislado del resto de componentes externos, configurados con objetos que simulen las dependencias o *test doubles* inyectados a través de una configuración especial del servicio bajo prueba. Existen cinco tipos de *test doubles*:

- *Dummy*: se trata de objetos que se envían pero que nunca se utilizan durante la prueba.
- *Stub*: es un objeto configurado para devolver un valor determinado cuando se invoca el componente.
- *Spy*: guardan información sobre los métodos que se han invocado y con qué parámetros.
- *Mock*: se trata de una ampliación del anterior en la que es necesario configurar qué comportamiento de espera cuando se invoca a alguno de sus métodos.
- *Fake*: es un objeto completo cuyo único fin es simplificar las pruebas.

Como consecuencia de este aislamiento, las pruebas de componentes no garantizan que los microservicios, en conjunto, satisfacen los requerimientos del negocio, sino que son necesarias las pruebas de aceptación por parte del usuario final. Una ventaja de las pruebas de componentes comparadas con las pruebas de aceptación es proporcionan una realimentación más precisa sobre el propio servicio. Por ejemplo, las API del servicio se verifican desde la perspectiva del consumidor, en lugar de la ejecución de una interacción de alto nivel entre el usuario final y el sistema completo. También suelen ser más rápidas.

A la hora de implementar las pruebas de componentes existen dos opciones:

- Pruebas IPCT (In-Process Component Tests): se ejecutan en la misma CPU que el servicio que se está probando. Los almacenes de datos que necesite el servicio para su persistencia se mantienen en memoria para que se acceda a ellos sin necesidad de cruzar las fronteras del propio proceso.

  Las pruebas se suelen comunicar con el microservicio a través de una interfaz interna especial que permite servir peticiones y recuperar respuestas sin ningún tipo de llamada HTTP a los servicios de API subyacente. De esta manera, las pruebas IPCT son prácticamente iguales que las O2PCT, pero sin incurrir en la sobrecarga adicional de las verdaderas interacciones a través de la red. Sin embargo, su principal inconveniente que el microservicio debe alterarse para las pruebas, ya que hay que arrancarlo en un modo "test", diferente del entorno de producción habitual.

- Pruebas O2PCT (Out-Of-Process Component Tests): tratan el servicio como una caja negra y verifican su ejecución en un proceso distinto al que se accede a través de la red utilizando las API públicas expuestas.

  Este tipo de pruebas permite verificar más capas y puntos de integración que las pruebas IPCT. Todas las interacciones tienen lugar a través de las llamadas de red, por lo que no es necesario cambiar la configuración del despliegue. Si el microservicio tiene una integración, persistencia o arranque complejos, esta aproximación es la única manera de comprobarlo a nivel de componente.

8.2.5 Pruebas E2E (End-To-End)

Las pruebas de usuario final tienen como objetivo verificar el correcto funcionamiento de la aplicación completa y de garantizar que se han cumplido los requisitos. La diferencia con las pruebas de componentes explicadas en el epígrafe anterior, es que no abarcan un solo microservicio sino es el conjunto completo, razón por las que también se llaman pruebas FTS (*Full Stack Tests*) o de pila completa.

Este nivel de pruebas permite a las arquitecturas basadas en microservicios evolucionar con el tiempo, ya que garantizan que el sistema permanece intacto durante las refactorizaciones y escalado de la arquitectura.

Uno de los problemas más frecuentes es cómo gestionar las dependencias potenciales de la aplicación bajo prueba respecto de servicios verdaderamente externos proporcionados por terceras partes.

Es posible que no se puedan escribir pruebas E2E repetibles y libres de efectos colaterales o que puedan fallar por razones que están fuera del control del equipo de pruebas. En estas circunstancias, puede ser beneficioso sacrificar cierto nivel de fiabilidad en las pruebas introduciendo *stubs* para dichos servicios externos, de manera que lo que se pierde en fidelidad se gana en una mayor estabilidad de las pruebas.

Como regla general, los test E2E son mucho más costosos que sus homólogos a niveles más bajos. Dada su naturaleza asíncrona, sus tiempos de ejecución más largos y el hecho de que implican varias partes del sistema, son más laboriosos de implementar y mantener. Para resolver este problema, resulta prudente limitar el presupuesto para las pruebas de usuario final y descartar algunos escenarios de pruebas una vez que se haya superado el presupuesto. Con el fin de determinar qué pruebas mantener, resulta útil diseñar los escenarios de test centrados en varios roles de usuario y formular historia que plasmen cómo estos usuarios interactúan con el sistema y centrar los esfuerzos en las partes de la aplicación que proporcionan un mayor valor. La cobertura para el resto se delega en pruebas de más bajo nivel. A este nivel, herramientas como Cucumber o Gauge permiten diseñar las pruebas en lenguajes específicos del dominio (DSL, Domain Specific Language), como veremos más adelante en este capítulo.

8.3 IMPLEMENTACIÓN DE LAS PRUEBAS

Llegados a este punto, describiremos cómo automatizar las pruebas en una arquitectura de microservicios haciendo uso de la aplicación de ejemplo que estamos desarrollando a lo largo del libro. Para ello, haremos uso de las herramientas y tecnologías recogidas en la Tabla 1. Por supuesto, es posible emplear otras.

Herramienta	Descripción
JUnit	Motor de ejecución de pruebas
Mockito	Objetos de prueba (*mocks*)
Wiremock	Objetos de prueba (*stubs*)
MockMVC	Pruebas de integración HTTP
Selemium	Pruebas E2E

Tabla 8.1. Herramientas y tecnologías empleadas en las pruebas

8.3.1 Pruebas unitarias

Las pruebas unitarias verifican funcionalidades aisladas, por lo que las interacciones con otras clases deben eliminarse siempre que sea posible. Puesto que estamos trabajando en JAVA (un lenguaje de programación orientado a objetos) las pruebas unitarias consistirán en verificar el perfecto funcionamiento de los métodos de las clases JAVA. Una buena regla práctica es disponer de una clase de prueba por cada clase del código de producción, independientemente de su funcionalidad y de en qué capa de la arquitectura de aplicación se ubique.

En general, la estructura de cualquier prueba, no sólo de las unitarias, es siempre la misma: se establecen los datos para la prueba, se invoca al método bajo prueba y se verifica que el resultado de esta llamada es el esperado.

Resulta muy habitual en las pruebas unitarias (y también en otro tipo de pruebas) hacer uso de objetos *mock* para conseguir el aislamiento de clases, ya sean creados de manera manual o a través de algún *framework* específico que permita no sólo crearlos sino también definir su comportamiento.

El ejemplo clásico de un *mock* es un proveedor de datos. En producción existe una implementación para conectarse a la base de datos real, pero, para pruebas, un *mock* simula dicha fuente de datos y garantiza que las condiciones de la prueba sean siempre las mismas. Estos objetos *mock* se proporcionan a la clase que se quiere probar de manera que no es necesario que dependa de datos externos.

En nuestro caso utilizaremos Mockito como *framework* de gestión de *mocks*, junto con JUnit. De este modo, la estructura de cualquier prueba con objetos *mock* será la siguiente (ver Figura 4):

Figura 8.1. Estructura de una prueba con Mockito

El primer paso para implementar pruebas con Mockito es importar las dependencias en gradle:

```
repositories { jcenter() }
dependencies { testCompile 'org.mockito:mockito-core:2.7.22' }
```

La anotación @Mock es la más utilizada en Mockito y sirve para crear e inyectar instancias de objetos *mock* sin necesidad de hacerlo manualmente.

```
package application.test;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;
import static org.junit.Assert.assertEquals;
import static org.mockito.MockitoAnnotations.initMocks;
import java.util.List;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import application.CatalogoHTTPController;
import application.dao.DAO;
import application.domain.Producto;

@RunWith(MockitoJUnitRunner.class)
public class CatalogoHTTPControllerUnitTest {
   private CatalogoHTTPController catalogoController;
   @Mock
   private DAO repository;

   @Before
   public void setUp() throws Exception {
      initMocks(this);
      catalogoController = new CatalogoHTTPController();
   }
   @Test
   public void test_getProductoPorId(){
      //Recuperamos el producto #1 del objeto mock
      Producto pMock =
repository.getProductoDAO().getProductoById(1);
      //Recuperamos el producto #1 del servicio
      Producto pService = catalogoController.getProductosJSON(1);
      assertEquals(pMock, pService);
   }
}
```

Resulta bastante habitual que en varios de los casos de prueba se empleen los mismos datos de entrada o de salida. Para evitar código repetido, JUnit proporciona las llamadas *fixtures*, que son unas anotaciones cuya ejecución está bien delimitada dentro de cada prueba. La Tabla 2 recoge las principales:

Anotación	Comportamiento
@Before	El método anotado se ejecutará antes de cada prueba. Se emplea para inicialización.
@After	El método anotado se ejecutará después de cada prueba. Se emplea para liberar recursos.
@BeforeClass	Se ejecuta una sola vez antes de todas las pruebas de una clase. Se emplea para crear estructuras de datos y componentes para todas las pruebas, por lo que los métodos anotados deben ser *static*.
@AfterClass	Se ejecuta una sola vez después de todas las pruebas de una clase. Se emplea para liberar los recursos inicializados por el método @BeforeClass y sólo se aplica a métodos estáticos.

Tabla 8.2. Fixtures de JUnit

8.3.2 Pruebas de integración

Las pruebas de integración son el siguiente nivel de la pirámide de Cohn. Se trata de verificar que la aplicación interacciona correctamente con su entorno externo, es decir, que es necesaria la ejecución no sólo de la aplicación sino también de los componentes con los que se integra.

Una buena manera de pensar qué debe contener una prueba de integración es pensar dónde se serializa o deserializa información. Buenos candidatos son la lectura de peticiones HTTP y el envío de respuestas HTTP a través de la API REST, la lectura/escritura de/en base de datos o un sistema de ficheros o el envío de peticiones a otros servicios con el consiguiente procesado de la respuesta obtenida.

En el ejemplo que presentamos a continuación comprobaremos la integración de la API REST del microservicio de catálogo. Por supuesto, podemos escribir pruebas unitarias sencillas para cada controlador de microservicio e invocar directamente a los métodos del controlador, tal y como hemos hecho en la prueba unitaria del ejemplo. Sin embargo, puesto que los controladores son muy simples y se evita incluir en ellos lógica de negocio, pruebas unitarias de este tipo carecen de utilidad práctica puesto que resultan o muy sencillas o triviales.

Por otra parte, los controladores hacen un uso intensivo de anotaciones Spring MVC para definir puntos finales, parámetros de consulta, etc., por lo que si queremos comprobar si la API funciona correctamente, debemos ir más allá de las pruebas unitarias.

Una manera de comprobar el API es arrancar todo el contexto de SpringBoot y lanzar peticiones HTTP reales. Esto es posible gracias a la anotación @MockMVC. El siguiente fragmento de código muestra cómo hacerlo.

```
@RunWith(SpringRunner.class)
@WebMvcTest(controllers=CatalogoHTTPController.class)
public class CatalogoHTTPControllerIntegrationTest {
   @Autowired
   private MockMvc mockMvc;
   @MockBean
   private DAO repository;
   @Test
   public void test_getProductoPorId() throws Exception {
      MockHttpServletRequestBuilder builder =
                MockMvcRequestBuilders.get("/productos/1")
                                .accept(MediaType.APPLICATION
JSON);
      mockMvc.perform(builder)
         .andExpect(MockMvcResultMatchers.status().isOk());
   }
   @Test
   public void test_getProductos() throws Exception{
      MockHttpServletRequestBuilder builder =
                MockMvcRequestBuilders.get("/productos")
                                .accept(MediaType.APPLICATION
JSON);
      mockMvc.perform(builder)
         .andExpect(MockMvcResultMatchers.status().isOk());
   }
}
```

La clase que implementa la prueba se anota con @WebMvcTest y se le indica a Spring de qué controlador se va verificar el funcionamiento. Por otra parte, en lugar de hacer uso de un repositorio de base de datos real, se emplea un *mock* anotándolo con @MockBean. En el ejemplo, se han implementado las mismas pruebas que en las unitarias pero lanzando peticiones HTTP reales al controlador y esperando recibir códigos de estado HTTP.

8.3.3 Pruebas E2E

En este punto ya hemos llegado a la parte superior de la pirámide de Cohn. Las pruebas E2E invocan a los microservicios desde la interfaz de usuario.

Para este tipo de pruebas utilizaremos Selenium y el protocolo WebDriver. Con Selenium podemos escoger un navegador e indicarle que invoque directamente a un sitio web, haga clic en determinados sitios, introduzca datos y compruebe los cambios que se producen en la interfaz de usuario.

Selenium necesita un navegador que pueda arrancar para ejecutar las pruebas. Una vez decidido cuál se utilizará se debe incluir la dependencia correspondiente en el build.gradle:

```
testCompile 'org.seleniumhq.selenium:selenium-firefox-driver:3.9.1'
testCompile 'org.seleniumhq.selenium:selenium-api:3.9.1'
testCompile 'org.seleniumhq.selenium:selenium-remote-driver:3.9.1'
```

El código siguiente muestra una prueba simple que arranca Firefox, navega por el servicio y comprueba el contenido del sitio web:

```
@RunWith(SpringRunner.class)
@WebMvcTest(CatalogoHTTPController.class)
public class CatalogoE2ESeleniumTest {
   private WebDriver driver;
   @Before
   public void setUp(){
      System.setProperty("webdriver.gecko.driver",
"C:\\geckodriver\\geckodriver.exe");
      driver = new FirefoxDriver();
   }
   @After
   public void tearDown() {
      driver.close();
   }
   @Test
   public void test_loadHomePage() {
      driver.get("/productos");
   }
}
```

El código es muy simple. Se arranca la aplicación SpringBott en un puerto aleatoria y se le indica al webdriver de Fireforx que abra la página principal del sitio web. Si encuentra la etiqueta <body>, se da la prueba como correcta. Obviamente, es posible escribir pruebas mucho más complejas.

La primera línea de la configuración de WebDriver es la especificación de la ruta a GeckoDriver.Se trata de una implementación de la API HTTP del protocolo WebDriver usada para la comunicación con los navegadores compatibles con Gecko,

como Firefox. Las distintas versiones del driver están disponibles en https://github.com/mozilla/geckodriver/releases (ver Figura 1).

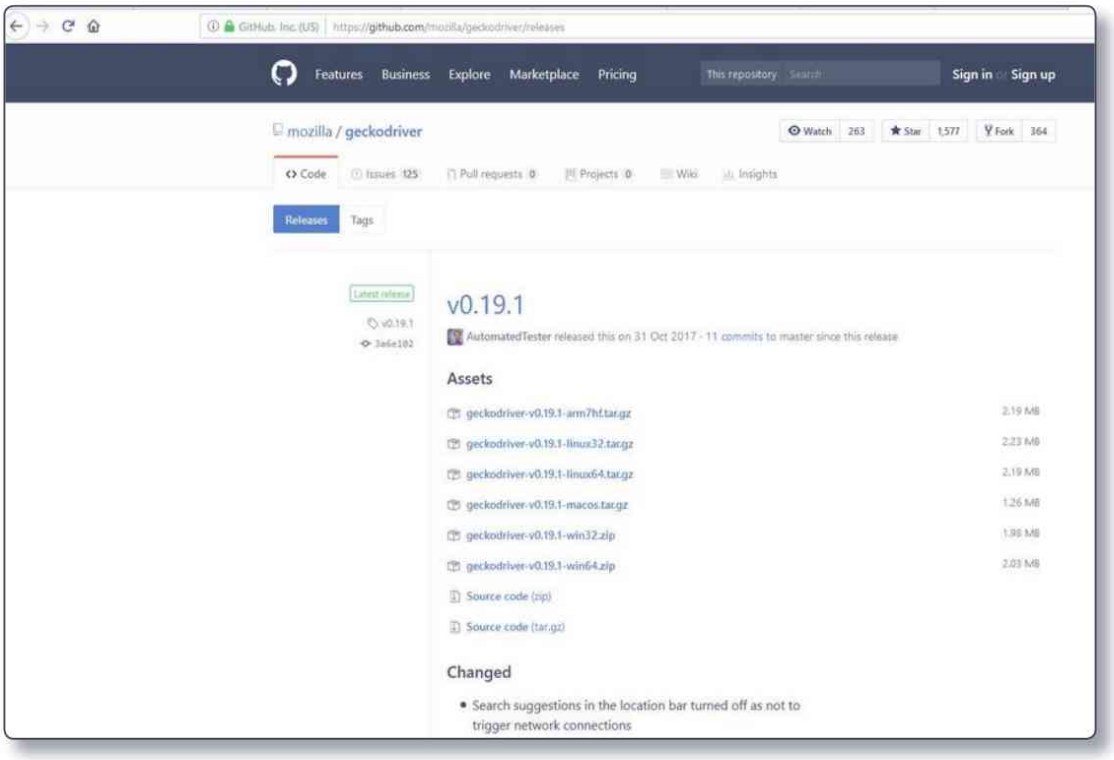

Figura 8.2.

Conviene en una carpeta en la que lo identifiquemos convenientemente. Puesto que este *driver* es útil también con otros navegadores, es mejor no ligarlo al Firefox, a pesar de que sea con este navegador con el que haremos las pruebas. Por tanto, lo guardaremos en una carpeta con su nombre (ver Figura 1).

Figura 8.3. Instalación de Geckodriver.

Otra alternativa para escribir pruebas E2E es Gherkin. Se trata de un lenguaje de dominio que permite diseñar las pruebas en lenguaje casi natural. La sintaxis de un fichero Gherkin resulta sencilla de comprender. Por ejemplo, para probar la

creación de un nuevo producto el fichero de diseño de la prueba en Gherkin sería el siguiente:

```
Feature: Crear un producto
Como usuario del microservicio Catalogo se debe ser capaz de
crear un producto
Scenario: Crear un productos datos un id, un nombre, una
descripcion y un precio
    Given Uso id <id>, nombre <name>, descripcion
<description> y precio <price>
    When Solicito la creación de un producto
    Then Debo obtener el código de estado HTTP 201
```

El fichero de diseño de la prueba es un fichero de texto plano con extensión *.features* debe guardarse dentro de la ruta de recursos de las pruebas. Estos ficheros se van leyendo al lanzar la prueba y se van ejecutando las clases de pruebas correspondientes que tienen una estructura similar.

En Java se utiliza Cucumber (*https://github.com/cucumber/cucumber*) como herramienta de ejecución de pruebas E2E diseñadas en Gherkin y que está disponible para su integración en Eclipse sin más que importar sus dependencias en el build.gradle:

```
   //Pruebas con Gerkhin y Cucumber
    testCompile('io.cucumber:cucumber-java:2.0.1')
    testCompile('io.cucumber:cucumber-junit:2.0.1')
```

Hecho esto, la clase de prueba es un reflejo fiel del fichero de Gherkin:

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment
RANDOM_PORT)
@ContextConfiguration
public class CatalogoGerkhinSteps {
     @Autowired
     private TestRestTemplate restTemplate;

     //Inputs
     private Integer id;
     private String nombre;
     private String descripcion;
     private Double precio;
   // output
     private ResponseEntity<String> response;
     @Given("I use a id (.*), nombre (.*), descripcion
(.*) and a precio (.*)")
```

```
     public void getProducto(Integer id, String nombre,
String descripcion, Double precio) {
       this.id = id;
       this.nombre = nombre;
       this.descripcion = descripcion;
       this.precio = precio;
     }
     @When("I request an expense creation")
     public void requestProductoCreation() {
        HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);
        MultiValueMap<String, Object> map= new
LinkedMultiValueMap<String, Object>();
        map.add("id", this.id);
        map.add("nombre", this.nombre);
        map.add("descripcion", this.descripcion);
        map.add("precio", this.precio);
        HttpEntity<MultiValueMap<String, Object>>
request = new HttpEntity<MultiValueMap<String, Object>>(map,
headers);
        response = restTemplate.postForEntity(
"/productos", request , String.class );
        }
     @Then("I should get a response with HTTP status code (.*)")
     public void shouldGetResponseWithHttpStatusCode(int
statusCode) {
       assertThat(response.getStatusCodeValue())
isEqualTo(statusCode);
     }
}
```

9

DESPLIEGUE DE MICROSERVICIOS

Una vez hemos desarrollado y testeado nuestros microservicios ha llegado el momento de realizar su despliegue, es decir, trasladarlos de las manos de los desarrolladores a las de los usuarios de forma que éstos puedan hacer uso de los microservicios. Hay varios factores que pueden ralentizar y entorpecer dicha fase de despliegue. Entre estos factores encontramos el entorno de explotación donde se van a desplegar los microservicios. Y es que solo en el caso que despleguemos los microservicios en el mismo entorno que el de producción, podremos no solo reducir el tiempo invertido en la realización de pruebas, sino también la aparición de bugs producidos como consecuencia de dicho cambio. Para dar solución a este problema podemos realizar el despliegue sobre máquinas virtuales que nos permitan aislar los microservicios en procesos de ejecución y despliegue independientes. Sin embargo, las virtualizaciones tradicionales suponen un consumo muy elevado de recursos de las máquinas sobre las que corren, ya que introducen una capa extra llamada *hypervisor* encargada de gestionar cada máquina virtual creada (ver figura 9.1). Es aquí donde los *Contenedores de Linux (LXC)*[12] surgen como alternativa más ligera a estas soluciones de virtualización ya que éstos no requieren del uso de esa capa extra. Los contenedores ejecutan procesos ligeros directamente sobre el sistema operativo anfitrión (ver figura 9.1), aislando los procesos de las aplicaciones del resto del sistema.

12 https://linuxcontainers.org/

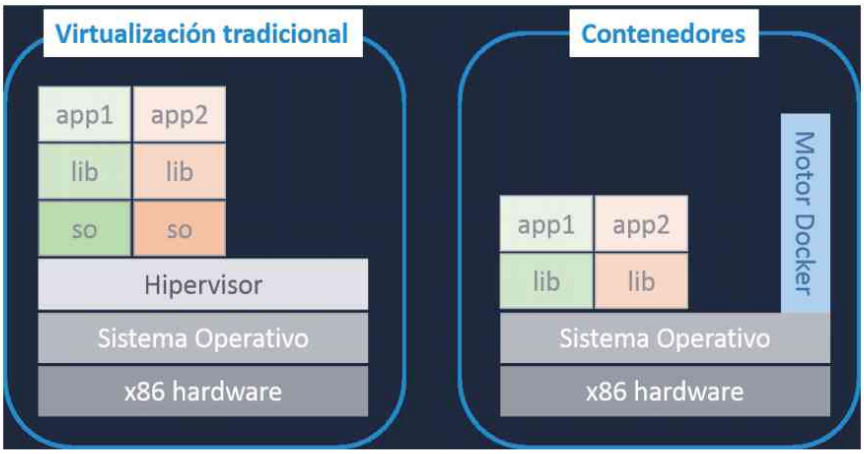

Figura 9.1.

Actualmente, la forma más extendida para trabajar con *LXC* es *Docker*[13], una tecnología de código abierto que permite no solo ejecutar contenedores sino que también facilita su gestión (proceso de creación y desarrollo, envió y versionado de imágenes, etc.). La tecnología *Docker*, es suficiente en sí si lo que queremos es gestionar contenedores únicos. Sin embargo, cuando pasamos a gestionar múltiples contenedores y aplicaciones en contenedores, deberemos combinar dicha tecnología con *Kubernetes*[14] , lo cual nos permitirá llevar a cabo una gestión y orquestación más sencilla de nuestros contenedores y de las aplicaciones en contenedores. Además, como proyecto software de código abierto, Docker cuenta con una comunidad de desarrolladores, la *comunidad Docker de open source*[15], que trabaja para mejorar dicha tecnología.

9.1 CONCEPTOS BÁSICOS DE DOCKER

A lo largo de esta sección introduciremos los diferentes conceptos con los que trabaja *Docker* ya que haremos referencia a éstos cuando nos dispongamos a *dockerizar* los microservicios de nuestro ejemplo en las siguientes secciones.

13 https://www.docker.com/

14 https://kubernetes.io/

15 https://forums.docker.com/

9.1.1 Repositorios Docker

Docker cuenta con una serie de repositorios desde los cuales podemos descargarnos plantillas de imágenes que se encuentran ya preparadas para su uso y/o modificación. En estos repositorios nos vamos a encontrar dos tipos de imágenes, oficiales y de usuario. Las imágenes oficiales son imágenes incluidas en los repositorios oficiales de *Docker* (*Docker Hub*[16]) los cuales son mantenidas por un equipo de desarrollo financiado por la propia compañía Docker Inc. que se encarga de revisar y publicar todo su contenido (ver figura 9.2). Estos repositorios proporcionan imágenes básicas que incluyen **ubuntu**, **alpine** o **python,** por ejemplo, y que sirven como punto de partida para la mayoría de usuarios. Una forma de diferenciar estas imágenes de las de usuario es que éstas no incluyen en su nombre ningún prefijo que referencie ni a una organización ni a un usuario particular. Siempre que se pueda, es recomendable utilizar estos repositorios ya que proporcionan documentación clara que sirve como referencia para la construcción de ficheros *Dockerfile* (ficheros de texto que determinan, mediante la especificación de una serie de comandos, la forma en la que se debe construir una imagen a partir de otra ya existente).

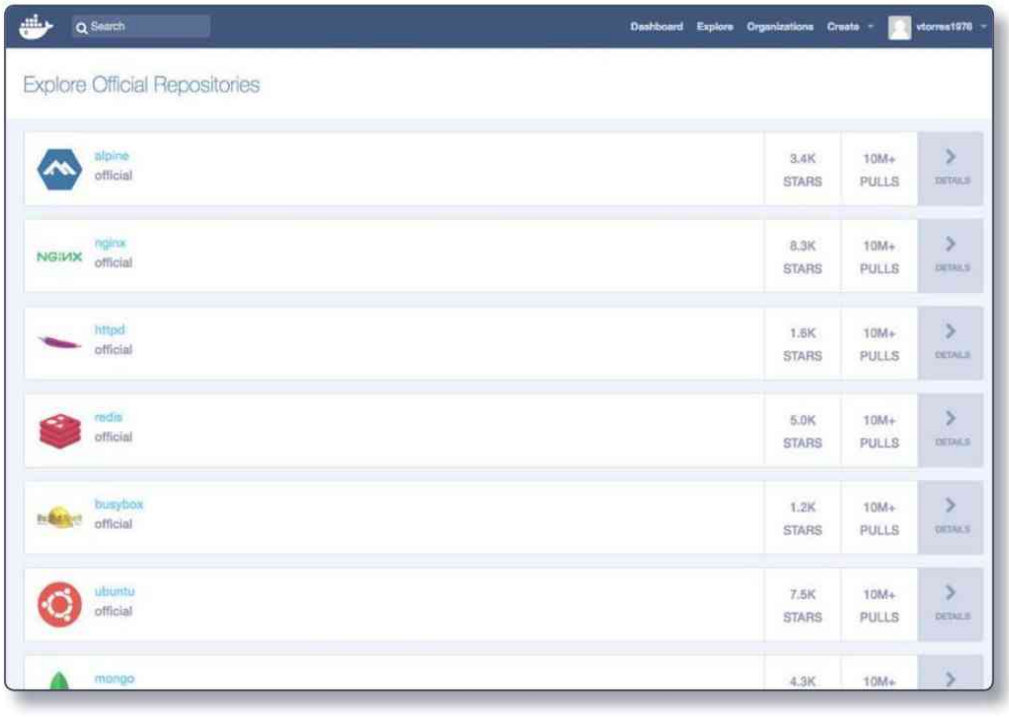

Figura 9.2. Lista de repositorios oficiales disponibles desde la web de Docker

16 https://hub.docker.com/

Por otro lado, los repositorios de usuario son repositorios creados y compartidos por organizaciones o usuarios que ofrecen imágenes que añaden funcionalidad adicional sobre las imágenes oficiales. Estos repositorios, a diferencia de los oficiales, incluyen un prefijo en su nombre que hace referencia al autor de éste, el cual se formatea normalmente como el par **usuario/nombre-repositorio**, siendo el **usuario** el id registrado en Docker y el **nombre-repositorio** el nombre que identifica la funcionalidad incluida en la imagen. En la figura 9.3 podemos ver parte de la lista de repositorios del Docker Hub que ofrecen openjdk, donde podemos ver en primer lugar el repositorio openjdk oficial de *Docker* y a continuación repositorios de organizaciones o usuarios.

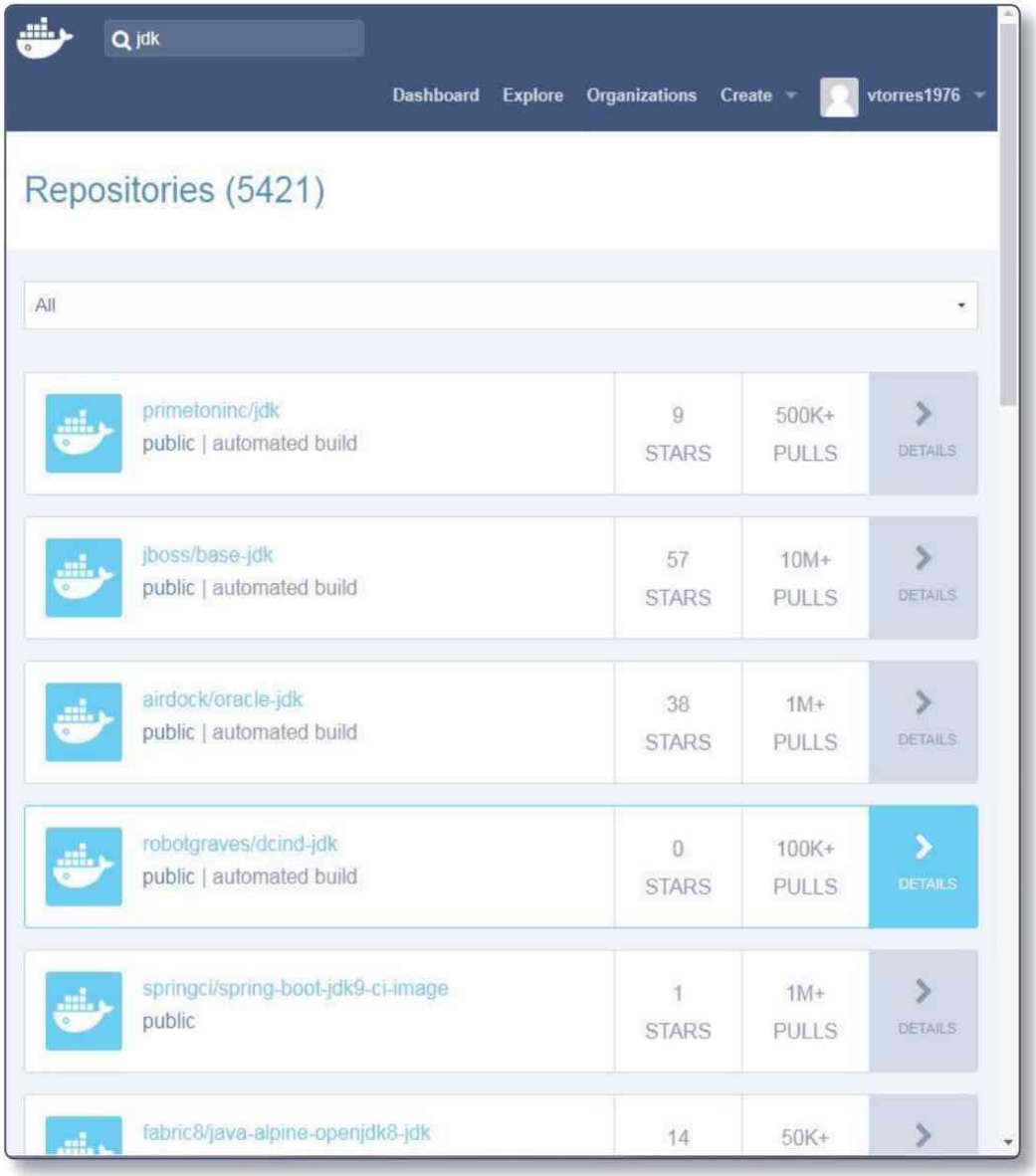

Figura 9.3. Lista de repositorios (oficiales y de usuario) que ofrecen imágenes con openjdk

9.1.2 Imágenes

Una imagen Docker es como una instantánea de una máquina virtual pero mucho más ligera. Sobre esta instantánea deberemos añadir todo lo que necesitemos (el código, el entorno de ejecución, librerías, variables de entorno y ficheros de configuración) para ejecutar una aplicación en cualquier otro ordenador. Para llevar a cabo la creación de imágenes podemos proceder de dos formas, bien desde cero o bien desde una imagen ya existente que podemos descargar desde el Docker Hub y la cual modificaremos mediante un fichero *Dockerfile*. Estos ficheros son el mecanismo que permite automatizar de forma sencilla el proceso de creación de imágenes. Contienen toda la información que *Docker* necesita para ejecutar la aplicación, es decir, la imagen *Docker* base desde dónde ejecutarla, la localización del código del proyecto, cualquier dependencia que éste tenga y los comandos para ejecutar el arranque de la aplicación. Toda esta información está incluida en estos ficheros como una serie de comandos que serán ejecutados por el demonio *Docker* para llevar a cabo la creación de la imagen. La figura 9.4 ilustra cómo las imágenes pueden ser creadas bien desde un fichero Dockerfile (Imagen 1.1a, Imagen 2.1a e imagen 2.3a) o bien desde una imagen directamente descargada desde Docker Hub (Imagen 1.2).

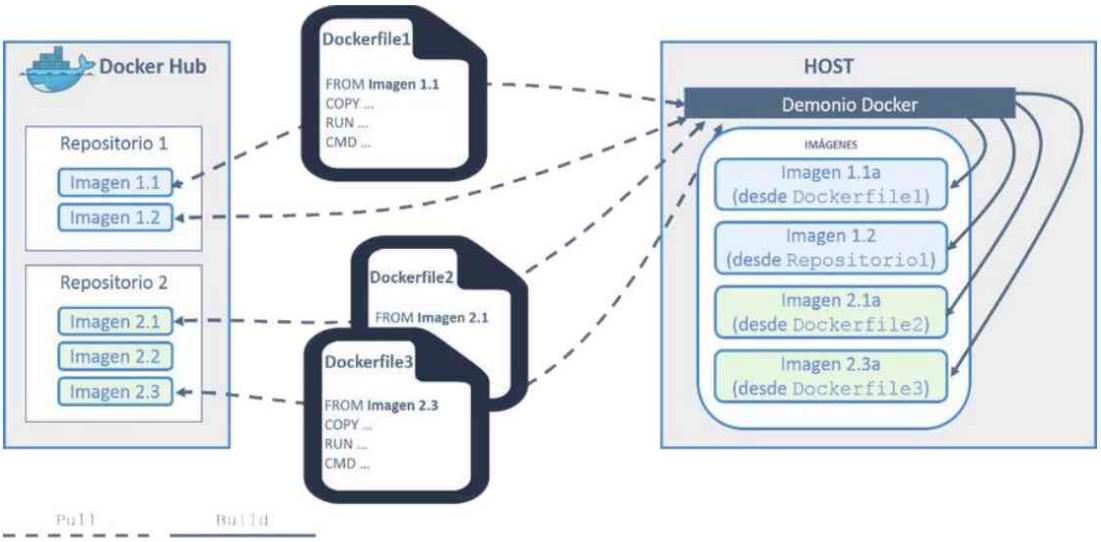

Figura 9.4. Creación de imágenes a partir de imágenes existentes en el Docker Hub

Las imágenes tienen un único *id* el cual se compone del par "nombre:etiqueta", donde el "nombre" hace referencia al repositorio al que pertenece dicha imagen y la "etiqueta" a la versión de ésta. Ejemplos de ids de imágenes son "Ubuntu:latest", "tomcat:9.0-slim", "netflixoss/tomcat:7.0.64" o "netflixoss/eureka:1.3.1".

Docker ofrece una serie de comandos que nos permiten conocer el estado de las imágenes creadas en nuestro sistema host. Entre estos comandos encontramos los siguientes:

- Comando para listar las imágenes descargadas en el host. Este comando acepta argumentos opcionales como son el nombre del repositorio e incluso la etiqueta de la imagen de forma que se mostraría solo las imágenes que coincidieran con dichos argumentos.

```
$ docker images [REPOSITORIO:[ETIQUETA]]
```

- Comando para eliminar imágenes. Este comando permite el borrado de una o varias imágenes de acuerdo al número de identificadores de imágenes incluidos como argumentos.

```
$ docker image rm <ID-imagen> [<ID-imagen>...]
```

9.1.3 Contenedores

Un contenedor es una instanciación de una imagen, la cual se ejecutará en *Docker*. A partir de una misma imagen podemos instanciar uno o más contenedores, los cuales estarán aislados los unos de los otros, asegurando la independencia entre aplicaciones instaladas sobre el sistema operativo anfitrión.

Al igual que ocurría con las imágenes, los contenedores tienen un único id, aunque en este caso éste se compone de un único elemento, el nombre. Ejemplos de ids de contenedores son "app_comercio1", "app_escolar" o "app_gestion". Además, los contenedores exponen los servicios que contienen mapeando puertos accesibles desde el exterior con puertos internos que utilizan las aplicaciones tal y como se muestra en la figura 9.5.

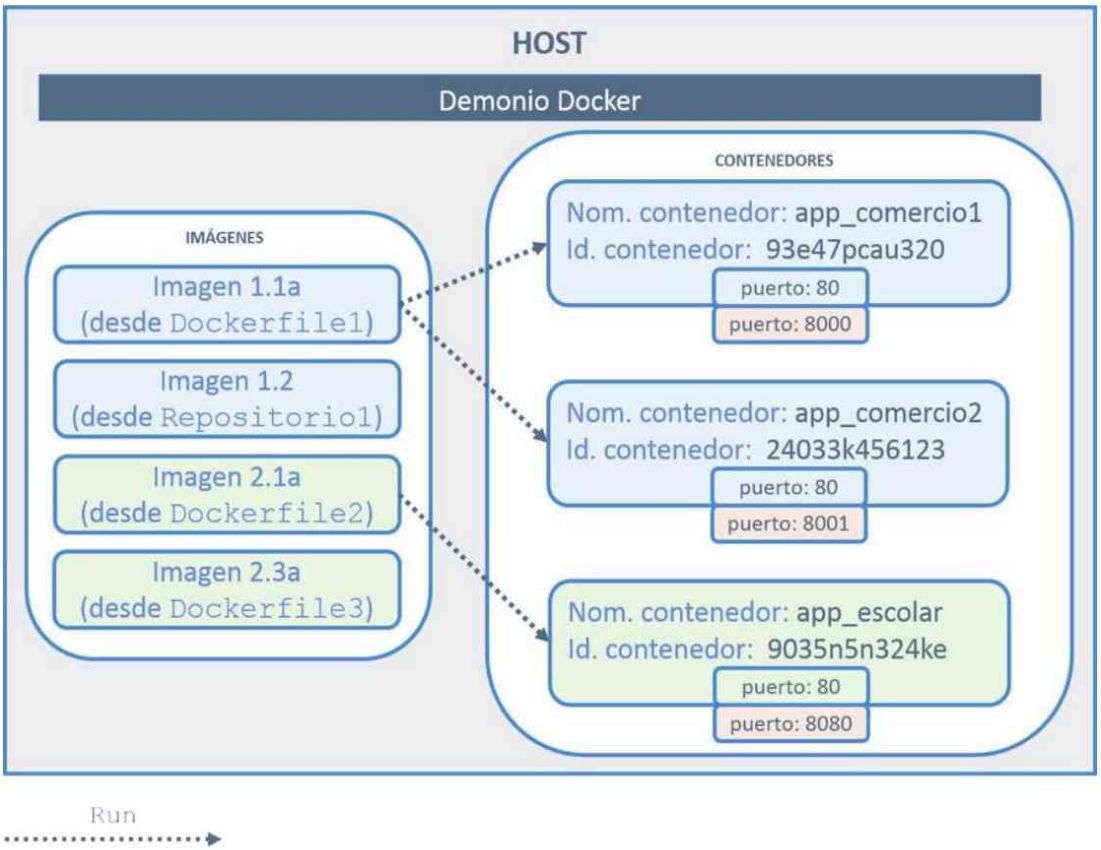

Figura 9.5. Contenedores exponiendo sus servicios al exterior en diferentes puertos

Entre el conjunto de comandos que ofrece *Docker* para gestionar contenedores destacamos los siguientes:

- Comando para listar los contenedores creados y activos en el host:

```
$ docker ps -a
```

- Comando para arrancar un contenedor:

```
$ docker start <nombre-del-contenedor>
```

- Comando para parar un contenedor:

```
$ docker stop <nombre-del-contenedor>
```

- Comando para eliminar contenedores:

```
$ docker rm <nombre-del-contenedor>
```

- Comando para obtener la IP asignada a un contenedor:

```
$ docker inspect <nombre-del-contenedor>
```

9.1.4 Volúmenes

Los contenedores son elementos efímeros que una vez son borrados desaparecen totalmente de nuestros equipos. Sin embargo, es posible que queramos hacer persistente los datos que las aplicaciones instaladas en ellos generan, incluso una vez hayamos eliminado cualquier rastro de ellos. Para llevar a cabo esta tarea de persistencia surgen los *volúmenes*, los cuales permiten separar el ciclo de vida de los contenedores de los datos. Gracias a esta separación, la eliminación de los contenedores no implica la eliminación de volúmenes utilizados por éstos. Esta separación es posible ya que *Docker* aloja dichos volúmenes fuera del propio contenedor en el sistema de ficheros del host donde está corriendo *Docker*. Estos volúmenes se almacenan como carpetas en un área reservada explícitamente para *Docker* (ver carpetas azules de la figura 11.6), en particular en la ruta /var/lib/docker/volumes/[17].

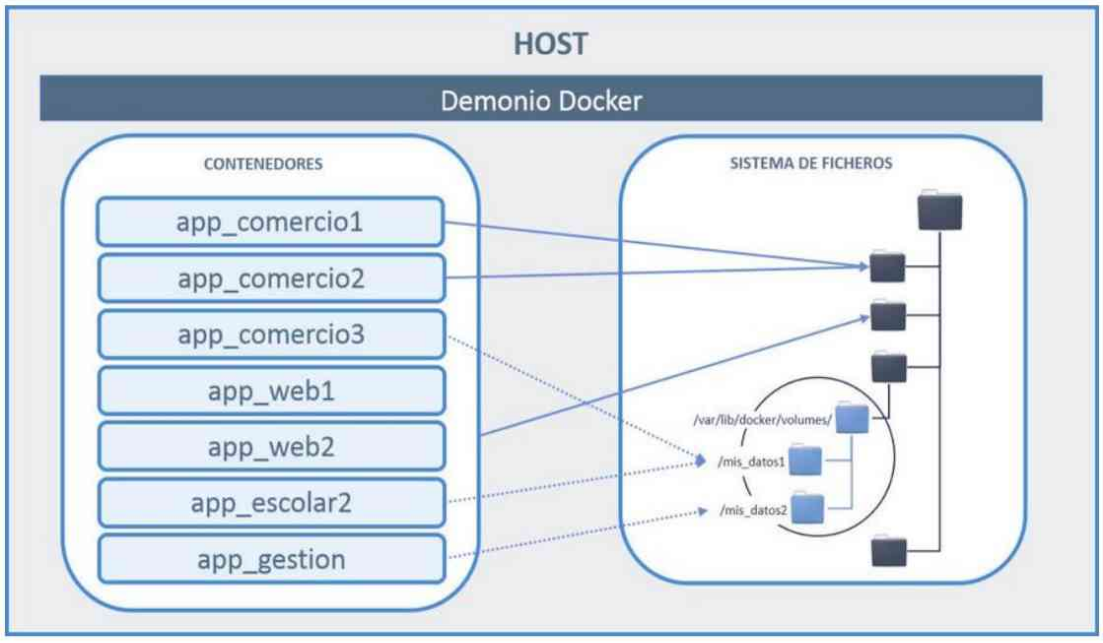

Figura 9.6. Separación entre contenedores y persistencia de datos

17 Ruta relativa a la máquina virtual sobre la que corre Docker

Aunque la manera predefinida para almacenar datos de forma persistente en *Docker* es mediante la utilización de *volúmenes* (ver flechas punteadas en la Figura 6), también se puede realizar mediante *volúmenes bind conectados* (ver flechas sólidas en la Figura 11.6). Mediante este último mecanismo conseguimos mapear carpetas de nuestro equipo con carpetas visibles y accesibles dentro de los contenedores. Ambas soluciones permiten la compartición de datos entre diferentes contenedores.

9.1.5 Docker Compose

Docker Compose es una herramienta que permite definir y ejecutar aplicaciones complejas con *Docker* cuando éstas se encuentran ubicadas en un único host. Con esta herramienta podemos definir una aplicación multi-contenedor en un fichero YAML (**docker-compose.yml**), el cual tras ser ejecutado mediante un único comando (**docker-compose up -d**) pondrá en marcha todo lo necesario para arrancar dicha aplicación. Trabajar con *Docker Compose* implica la realización de estos tres pasos:

1. Definir el entorno de nuestra aplicación con un fichero *Dockerfile* de forma que ésta pueda ser reproducida en cualquier lugar.
2. Definir en el fichero **docker-compose.yml** los servicios que definen nuestra aplicación.
3. Ejecutar el fichero YAML creado en el punto 2 para iniciar la herramienta Docker Compose y ejecutar la aplicación.

9.2 PREPARACIÓN DEL ENTORNO

En primer lugar vamos a preparar el entorno que nos permitirá desplegar nuestros microservicios en contenedores Docker. Para ello deberemos instalar *Docker Engine* y *Docker Compose* en nuestra máquina. Para sistemas Mac y Windows, la instalación de *Docker* incluye ya *Compose*. Sin embargo, para sistemas Linux la instalación debe realizarse en dos pasos, primero instalando *Docker* y después *Compose*. Para instalar *Docker* deberemos descargar la versión de *Docker Community Edition (CE)*[18] para la plataforma correspondiente. Para instalar Compose en sistemas Linux deberemos seguir las instrucciones de la web oficial[19]. Deberemos tener una cuenta en Docker (https://www.docker.com/) ya que desde ahí podremos crear y compartir repositorios tanto de forma privada como pública.

18 https://www.docker.com/community-edition

19 https://docs.docker.com/compose/install/

Una vez tengamos instalado *Docker Community Edition* deberemos probar la instalación para ver que funciona correctamente. Para ello, arrancamos *Docker* y a continuación, desde un terminal, ejecutamos el siguiente comando:

```
$ docker run hello-world
```

Mediante este comando estamos indicándole a *Docker* que cree una instancia (Contenedor) de la imagen llamada "hello-world". Docker busca inicialmente esta imagen en nuestro repositorio local y si no la encuentra la busca en el repositorio público de imágenes de Docker para descargársela y lanzar una instancia de ésta.

9.3 DOCKERIZACIÓN DE MICROSERVICIOS

A lo largo de esta sección vamos a ver qué pasos debemos realizar para desplegar nuestros microservicios en *Docker*. De acuerdo al diagrama de arquitectura de microservicios mostrado en la tabla 2.4, deberíamos crear un contenedor individual para cada microservicio, ya se refiera este a un servidor o a los propios microservicios del dominio. Sin embargo, en este capítulo nos vamos a limitar a describir el proceso de *dockerización* de los servidores de configuración (config server) y de descubrimiento (Eureka), del servicio de gateway (servidor Zuul) y de dos de los microservicios del dominio, el del catálogo y el de la cesta. De esta forma procederemos a crear contenedores para los siguientes cinco microservicios:

- Servidor de configuración
- Servidor Eureka
- Servicio Gateway
- Microservicio Catálogo
- Microservicio Cesta

A groso modo la secuencia de pasos que vamos a realizar y que se explica en las siguientes secciones es la siguiente:

1. Crearemos una estructura de directorios en local donde ubicaremos todos los ficheros necesarios para el proyecto (.jar, .sh y Dockerfile) (ver sección 9.3.1).

2. Crearemos dos scripts en la carpeta raíz para asegurarnos de que los contenedores de los microservicios del dominio no arrancan antes de los del servidor de configuración y del servidor Eureka (ver sección 9.3.2).

3. Crearemos un fichero *Dockerfile* que definirá la imagen base a partir de la cual crearemos las imágenes para nuestros microservicios (ver sección 9.3.3).

4. Crearemos el fichero *Dockerfile* a partir de la imagen base creada en el punto 3 para crear la imagen y el contenedor del servidor de configuración (ver sección 9.3.4).
5. Empaquetaremos el resto de microservicios como ficheros jar (ver sección 9.3.5)
6. Crearemos un fichero *Dockerfile* a partir de la imagen base creada en el punto 3 para crear la imagen del servidor Eureka (ver sección 9.3.6).
7. Actualizaremos los ficheros de configuración del github del resto de microservicios (ver sección 9.3.7).
8. Crearemos tres ficheros *Dockerfile* a partir de la imagen base creada en el punto 3 para crear las imágenes del resto de microservicios (ver sección 9.3.8).
9. Crearemos el fichero YAML **Docker-compose.yml** donde configuraremos los servicios de nuestra aplicación. La ejecución de dicho fichero nos permitirá crear y arrancar los servicios definidos en la configuración de este fichero (ver sección 9.3.9).

Tras la ejecución del fichero YAML el host incluirá una serie de imágenes, contenedores y volúmenes tal y como se muestra en la figura 9.7.

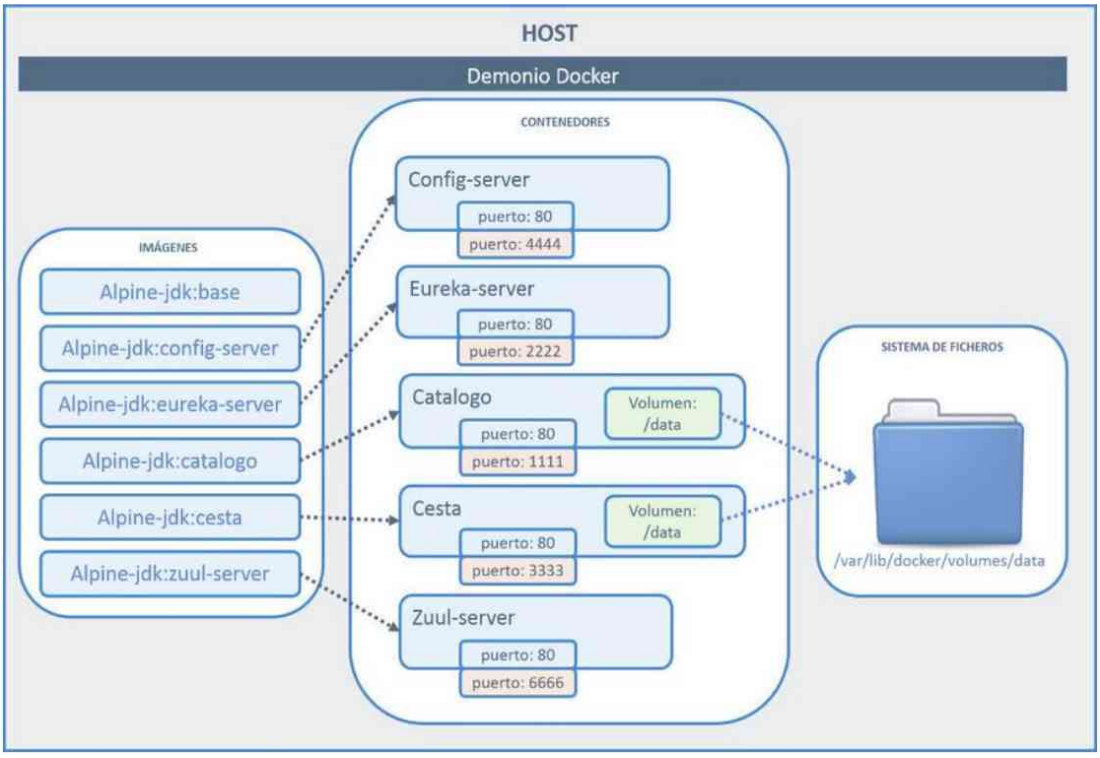

Figura 9.7. Dockerización de los microservicios del ejemplo

9.3.1 Preparación de la estructura de directorios y ficheros

Antes de ponernos manos a la obra con las imágenes y los contenedores, vamos a crear, en local, la estructura directorios donde organizaremos los ficheros jar de nuestros microservicios. En primer lugar, vamos a crear un directorio llamado "cestaCompraApp" el cual será el directorio raíz del proyecto. Este directorio incluirá los siguientes ficheros y carpetas que se detallan a continuación y que se muestra en la figura 9.8.

- Los ficheros *Docker* (*Dockerfile*) que utilizaremos para crear las imágenes y los contenedores de los microservicios.
- El fichero YAML (docker-compose.yml) donde definiremos los servicios que conforman nuestra aplicación.
- Dos shell scripts que se encargarán de controlar que los microservicios del dominio no arranquen antes que los servidores de configuración y de descubrimiento.
- La carpeta "ficheros" que contendrá los ficheros jar de los microservicios. Puesto que los microservicios necesitan conocer la IP donde se expone el servidor de configuración y el servidor eureka, la generación de los ficheros jar debe hacerse de forma progresiva. En primer lugar, deberemos generar los jars de estos dos servidores y tras comprobar su IP, configurar los microservicios para que se conecten a ellos y generar los correspondientes jars (ver sección 9.3.5).

Figura 9.8. Estructura de directorios y ficheros del proyecto

9.3.2 Dependencias entre contenedores

Para que nuestros microservicios funcionen correctamente deberemos asegurarnos de que éstos se ponen en marcha una vez tengamos el servidor de configuración y Eureka ejecutándose. Es por ello que vamos a incluir un par de shell scripts los cuales asegurarán que los microservicios del dominio (catálogo y cesta) no se arranquen hasta que los servidores no se hayan puesto en marcha. Estos scripts los ubicaremos en el directorio raíz de nuestro proyecto y los llamaremos **catalogo-entrypoint.sh** y **cesta-entrypoint.sh** respectivamente. El código de ambos scripts es el mismo a excepción del fichero jar a ejecutar en la instrucción java tal y como se muestra en los siguientes fragmentos de código.

```
#!/bin/sh
while ! nc -z config-server 4444 ; do
    echo “Esperando al Servidor de Configuracion”
    sleep 3
done
while ! nc -z eureka-server 5555 ; do
    echo “Esperando al Servidor Eureka”
    sleep 3
done
java -jar /opt/lib/ es.rama.books.catalogo-0.1.0.jar
```

```
#!/bin/sh
while ! nc -z config-server 4444 ; do
    echo “Esperando al Servidor de Configuracion”
    sleep 3
done
while ! nc -z eureka-server 5555 ; do
    echo “Esperando al Servidor Eureka”
    sleep 3
done
java -jar /opt/lib/ es.rama.books.cesta-0.1.0.jar
```

9.3.3 Creación de la imagen base

En nuestro caso, como ya vimos en los capítulos 3 y 4, para desarrollar y ejecutar nuestros microservicios necesitamos crear un entorno que nos permita ejecutar aplicaciones Spring Boot y el servidor de configuración y el servidor Eureka. Para ejecutar todos estos microservicios nos bastará con una imagen que incluya Java. Aunque podemos crear los contenedores desde cero, lo más normal

será que primero busquemos alguna imagen en el repositorio *Docker* que contenga la tecnología necesaria y nos la descarguemos para lanzar contenedores basados en ella. De esta forma, la imagen base la crearemos a partir de la imagen "openjdk:8-jre-alpine", imagen basada en la distribución ligera de Linux Alpine[20] que está disponible desde el repositorio oficial openjdk desde el *Docker Hub*. Esta imagen es más ligera que el resto de imágenes "no Alpine" ya que está creada a partir de la imagen base Alpine, que es más ligera que las basadas en la imagen Debian (debian:jessie).

Dentro del directorio "cestaCompraApp" creamos un fichero *Dockerfile* a partir del cual crearemos la imagen base que utilizaremos para crear los 5 contenedores donde *dockerizaremos* nuestros microservicios. Dicho fichero deberá incluir las siguientes líneas:

```
1. FROM openjdk:8-jre-alpine
2. LABEL maintainer="xxx@gmail.com"
3. RUN apk add --no-cache openjdk8
```

Todo fichero *Dockerfile* válido debe comenzar con la instrucción "FROM" la cual indica a *Docker* cuál es la imagen base sobre las que se ejecutarán las siguientes instrucciones. En este caso la imagen base es "**openjdk:8-jre-alpine**" y las siguientes instrucciones son "LABEL" con la meta-etiqueta "maintainer" la cual se utiliza para indicar el correo de contacto del autor de la imagen generada, y "RUN", el cual ejecuta el applet **add** de la herramienta **apk** para instalar el paquete **openjdk** en una capa nueva sobre la imagen actual. La opción **–-no-cache** la incluimos para mantener el contenedor pequeño, ya que de esta forma no almacenamos el índice localmente. A partir de este momento, la imagen resultante de dicha instrucción será utilizada por las siguientes instrucciones que hubiera en el fichero *Dockerfile*.

A continuación ejecutamos el fichero *Dockerfile* para crear la imagen base que utilizaremos para crear los contenedores. Para ejecutar dicho fichero ejecutaremos la siguiente instrucción desde la línea de comando de un terminal:

```
$ docker build --tag=alpine-jdk:base --rm=true .
```

Mediante esta instrucción estamos indicándole al demonio *Docker* (servicio en segundo plano que se ejecuta en el host encargado de gestionar la construcción, ejecución y distribución de contenedores *Docker*) que ejecute el fichero *Dockerfile*, donde con la opción **--tag** estamos indicamos el nombre y la etiqueta de la imagen en el formato "nombre:etiqueta" (alpine-jdk:base en nuestro caso) y con la opción **--rm**

20 http://gliderlabs.viewdocs.io/docker-alpine/

igual a **true** estamos especificando que se eliminen los contenedores intermediarios una vez se ha creado la imagen.

Podemos comprobar si la imagen base se ha creado correctamente ejecutando el comando introducido en la sección 9.1.2 que nos permite listar las imágenes descargadas y creadas en el host. Si todo ha ido correctamente deberíamos obtener desde la consola algo similar a lo mostrado en la figura 9.9.

```
veraw58-220:cestaCompraApp victoria$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
alpine-jdk          base                453d8203decd        About a minute ago  102MB
openjdk             8-jre-alpine        c529fb7782f9        7 days ago          82MB
veraw58-220:cestaCompraApp victoria$
```

Figura 9.9. Listado de imágenes del host

9.3.4 Creación de la imagen y contenedor para el servidor de configuración

A partir de la imagen "alpine-jdk:base" recién creada vamos a crear una segunda imagen *Docker*. Esta segunda imagen será la que utilicemos para crear el contenedor que contendrá el Servidor de configuración (Config Server).

Sin embargo, antes de comenzar deberemos empaquetar el servidor de configuración en un fichero jar y ubicar dicho fichero en la carpeta "ficheros" que habíamos creado dentro de la carpeta raíz "cestaCompraApp". Podemos generar dicho fichero lanzando la tarea Gradle *build->bootRepackage* tal y como se muestra en la figura 9.10. Es fichero jar se genera dentro de la carpeta **build/lib** del correspondiente proyecto y podemos acceder a él mediante cualquier explorador de archivos.

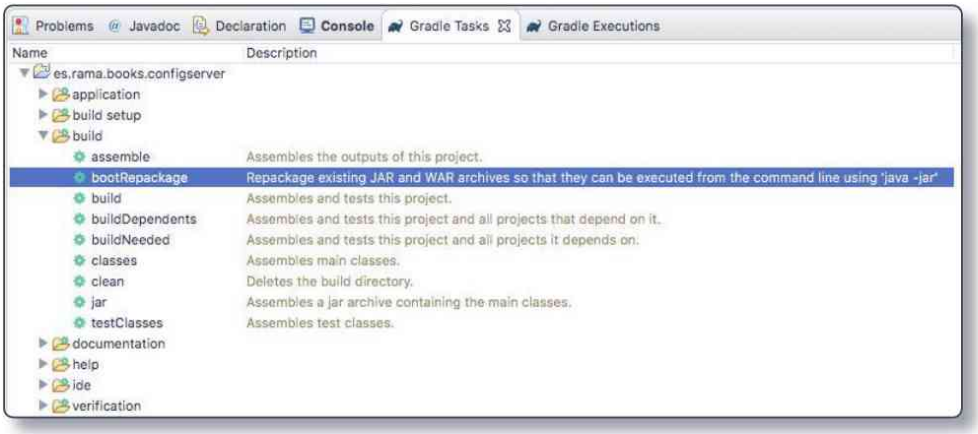

Figura 9.10. Tarea Gradle para generar el fichero jar

Una vez empaquetado el servidor, en el directorio raíz "cestaCompraApp" creamos un nuevo fichero *Dockerfile* que llamaremos **Dockerfile-configserver** el cual deberá incluir las siguientes instrucciones:

```
1. FROM alpine-jdk:base
2. LABEL maintainer="xxx@gmail.com"
3. COPY ficheros/es.rama.books.configserver-0.1.0.jar /opt/lib/
4. ENTRYPOINT ["/usr/bin/java"]
5. CMD ["-jar", "/opt/lib/es.rama.books.configserver-0.1.0.jar"]
6. EXPOSE 4444
```

Mediante estas seis instrucciones estamos indicándole al demonio *Docker* que cree una imagen a partir de la imagen base "alpine-jdk:base" recientemente creada (instrucción "FROM", línea 1), que copie el fichero .jar del servidor de configuración en el directorio /opt/lib (línea 2). A partir de este punto, indicamos mediante las instrucciones "ENTRYPOINT" y "CMD" que cuando el contenedor arranque queremos que el servidor de configuración comience a correr (líneas 4 y 5 respectivamente). Al incluir estas dos instrucciones evitamos incluir argumentos adicionales cuando ejecutemos el fichero *Docker* desde el comando **docker run**. Por último, mediante la instrucción "EXPOSE" especificamos que el servidor de configuración estará accesible desde fuera a través del puerto 4444 (línea 6).

Una vez hemos definido el fichero *Dockerfile* ya podemos crear la imagen y etiquetarla como **config-server**. Esto lo conseguiremos ejecutando la siguiente instrucción desde un terminal:

```
$ docker build --file=Dockerfile-configserver
  --tag=config-server:latest --rm=true .
```

Ahora, mediante el comando `run` de *Docker* ya podemos crear un contenedor *Docker* para el servidor de configuración. La instrucción completa a ejecutar desde la línea de comando del terminal es la siguiente:

```
$ docker run --name=config-server --publish=4444:4444
  config-server:latest
```

Una vez hemos dockerizado el servidor de configuración deberemos averiguar cuál es la IP que *Docker* le ha asignado. Necesitamos realizar este paso ya que deberemos actualizar los ficheros **bootstrap.yml** del resto de microservicios y del servidor eureka antes de generar los ficheros jar de cada uno de ellos. Esta actualización es necesaria ya que todos los microservicios deben conectarse al servidor de configuración para obtener información sobre su configuración. Para averiguar la IP asignada a dicho contenedor deberemos ejecutar desde la línea de comando de un terminal la siguiente instrucción:

```
$ docker inspect config-server | grep IPAddress
```

Como resultado de dicha ejecución el demonio de Docker muestra en el terminal la IP asignada a dicho contenedor:

```
"SecondaryIPAddresses": null,
            "IPAddress": "172.17.0.2",
                    "IPAddress": "172.17.0.2",
```

9.3.5 Generación de ficheros jar de microservicios y Eureka

Como ya hemos comentado anteriormente en la sección 11.3, antes de empaquetar los microservicios y el servidor Eureka debemos actualizar los ficheros **bootstrap.yml** de éstos con la IP del servidor de configuración. En particular, tenemos que actualizar el valor del campo **cloud.config.uri** asignándole la IP correspondiente. Una vez hayamos actualizado este campo en cada uno de los ficheros *bootstrap.yml* de los microservicios y de Eureka ya estamos en condiciones de generar los ficheros jar correspondientes. Para ello, procederemos de la misma forma que cuando empaquetamos el servicio de configuración (ver sección 9.3.4).

Deberemos ubicar los ficheros jar generados en el directorio **ficheros** que habíamos creado dentro del directorio raíz "cestaCompraApp".

9.3.6 Creación de la imagen y contenedor para el servidor Eureka

Siguiendo con el mecanismo de creación de imágenes a partir de ficheros *Dockerfile*, crearemos un nuevo fichero para el servidor Eureka que llamaremos **Dockerfile-eurekaserver**. Las instrucciones contenidas en este fichero son:

```
1. FROM alpine-jdk:base
2. LABEL maintainer="xxx@gmail.com"
3. COPY ficheros/es.rama.books.eurekaserver-0.1.0.jar /opt/lib/
4. ENTRYPOINT ["/usr/bin/java"]
5. CMD ["-jar", "/opt/lib/es.rama.books.eurekaserver-0.1.0.jar"]
6. EXPOSE 5555
```

En este caso el servidor estará disponible desde el puerto 5555 tal y como hemos especificado en la instrucción "EXPOSE" (línea 6).

Una vez completada la definición del fichero *Dockerfile* ya podemos crear la imagen y crear un contenedor mediante las instrucciones **build** y **run** respectivamente tal y como se indica en las siguientes instrucciones:

```
$ docker build --file=Dockerfile-eurekaserver
  --tag=eureka-server:latest --rm=true .
$ docker run --name=eureka-server --publish=5555:5555
  eureka-server:latest
```

Consultamos la IP asignada al contenedor recién creado desde la línea de comando de un terminal mediante la siguiente instrucción:

```
$ docker inspect eureka-server | grep IPAddress
```

Como resultado de dicha ejecución el demonio de *Docker* muestra en el terminal la IP asignada a dicho contenedor (ver figura 11.12)

```
  "SecondaryIPAddresses": null,
           "IPAddress": "172.17.0.3",
                  "IPAddress": "172.17.0.3",
```

9.3.7 Ficheros de configuración del GitHub

Tras dockerizar y consultar la IP asignada al servidor Eureka deberemos actualizar los archivos de configuración del resto de microservicios que tenemos alojados en el servidor Git. En particular, deberemos indicar la IP asignada al servidor Eureka en la propiedad **eureka.client.serviceUrl.defaultZone** de cada archivo de configuración.

Esta actualización también la deberemos realizar con la propiedad **security.oauth2.resource.user-info-uri** del servidor oAuth una vez lo hayamos dockerizado y consultado la IP asignada.

9.3.8 Creación de la imagen y contenedores para los microservicios

A continuación, vamos a desplegar los propios microservicios. Procederemos de forma similar a como lo hemos realizado con los servidores de configuración y Eureka, mediante la especificación de un fichero *Dockerfile*. Comenzaremos con el fichero para el microservicio del Catálogo. A este fichero lo llamaremos **Dockerfile-catalogo** y contendrá las siguientes siete instrucciones:

```
FROM alpine-jdk:base
LABEL maintainer="xxx@gmail.com"
COPY ficheros/es.rama.books.catalogo-0.1.0.jar /opt/lib/
COPY catalogo-entrypoint.sh /opt/bin/catalogo-entrypoint.sh
RUN chmod 755 /opt/bin/catalogo-entrypoint.sh
EXPOSE 1111
```

Procederemos de la misma forma con el microservicio de Cesta. A este fichero lo llamaremos **Dockerfile-cesta** y contendrá las siguientes siete instrucciones:

```
FROM alpine-jdk:base
LABEL maintainer="xxx@gmail.com"
COPY ficheros/ es.rama.books.cesta-0.1.0.jar /opt/lib/
COPY cesta-entrypoint.sh /opt/bin/cesta-entrypoint.sh
RUN chmod 755 /opt/bin/cesta-entrypoint.sh
EXPOSE 3333
```

Por último, creamos el fichero *Docker* para el servidor Zuul. A este fichero lo llamaremos **Dockerfile-zuulserver** y contendrá las siguientes siete instrucciones:

```
FROM alpine-jdk:base
LABEL maintainer="xxx@gmail.com"
COPY ficheros/es.rama.books.zuulserver-0.1.0.jar /opt/lib/
ENTRYPOINT ["/usr/bin/java"]
CMD ["-jar", "/opt/lib/es.rama.books.zuulserver-0.1.0.jar"]
EXPOSE 6666
```

9.3.9 Definición del Docker Compose

Por último, nos queda especificar el fichero YAML docker-compose.yml. En este fichero definimos los servicios que confoman nuestra aplicación, de forma que estos pueden lanzarse de forma conjunta en un entorno aislado.

```
version: '3.6'
services:
    config-server:
        container_name: config-server
        build:
            context: .
            dockerfile: Dockerfile-configserver
        image: config-server:latest
        expose:
            - 4444
        ports:
            - 4444:4444
    eureka-server:
        container_name: eureka-server
        build:
            context: .
            dockerfile: Dockerfile-eurekaserver
        image: eureka-server:latest
        expose:
```

```
                - 5555
            ports:
                - 5555:5555
        catalogo:
            container_name: catalogo
            build:
                context: .
                dockerfile: Dockerfile-catalogo
            image: catalogo:latest
            environment:
                SPRING_APPLICATION_JSON: '{"spring": {"cloud":
    {"config": {"uri": "http://config-server:4444"}}}}'
            entrypoint: /opt/bin/Catalogo-entrypoint.sh
            expose:
                - 1111
            ports:
                - 80:1111
            links:
                - config-server:config-server
                - eureka-server:eureka-server
            depends_on:
                - config-server
                - eureka-server
            logging:
                driver: json-file
        cesta:
            container_name: cesta
            build:
                context: .
                dockerfile: Dockerfile-cesta
            image: cesta:latest
            environment:
                SPRING_APPLICATION_JSON: '{"spring": {"cloud":
    {"config": {"uri": "http://config-server:4444"}}}}'
            entrypoint: /opt/bin/Cesta-entrypoint.sh
            expose:
                - 9091
            ports:
                - 9091: 9091
    links:
                - config-server:config-server
                - eureka-server:eureka-server
            depends_on:
                - config-server
                - eureka-server
            logging:
```

```
            driver: json-file
    zuul-server:
        container_name: zuul-server
        build:
            context: .
            dockerfile: Dockerfile-zuulserver
        image: zuul-server:latest
        expose:
            - 6666
        ports:
            - 6666:6666
        links:
            - eureka-server:eureka-server
        depends_on:
            - eureka-server
        logging:
            driver: json-file
```

Veamos con un poco de detalle qué es lo que hemos especificado en el fichero docker-compose.yml. En primer lugar debemos especificar de forma obligatoria el formato del fichero Compose mediante la propiedad "version. Existen varias versiones de dicho formato, la 1, 2, 2.x y 3.x. En nuestro caso hemos definido la versión 3.6 ya que es el formato soportado por la versión del motor de Docker que tenemos instalada en nuestra máquina de acuerdo a la tabla mostrada en la documentación oficial de Docker (ver tabla 9.1).

Formato fichero compose	Versión del motor Docker
3.6	18.02.0+
3.5	17.12.0+
3.4	17.09.0+
3.3	17.06.0+
3.2	17.04.0+
3.1	1.13.1+
3.0	1.13.0+
2.4	17.12.0+
2.3	17.06.0+
2.2	1.13.0+
2.1	1.12.0+
2.0	1.10.0+
1.0	1.9.1+

Tabla 9.1. Formatos compose

Podemos consultar la versión bien desde el interfaz gráfico de la instalación (ver figura 9.13) o desde un terminal ejecutando la siguiente instrucción:

```
$ docker --version
```

Figura 9.1. Versión del motor Docker

A continuación, pasamos a detallar la sección "services" donde definimos una entrada por cada contenedor a crear. Es por ello que dentro de esta sección aparecen 6 secciones, una que hace referencia al servidor de configuración (config-server), una segunda que hace referencia al servidor Eureka (eureka-server), una tercera que hace referencia al servicio del Catálogo (catalogo), una cuarta al servicio Cesta (cesta), una quinta al servidor Zuul (zuul-server) y por último una sexta que hace referencia a Hystrix (hystrix-dashboard). Para cada uno de estos servicios especificamos:

- **build** para indicar que Docker-compose debe crear una imagen a partir del fichero Dockerfile especificado
- **image** para indicar el nombre de la imagen que será creada
- **network** para indicar el nombre de la red que se va a utilizar
- **links** para crear un enlace interno entre el servicio que se está especificando y el indicado en esta propiedad. En nuestro caso, el servicio Catalogo tiene que acceder al servidor de configuración y al servidor Eureka.

- **depends** permite especificar el orden entre los contenedores creados. Por ejemplo, el contenedor Catalogo depende de los contenedores del servidor de configuración y de Eureka. Por lo tanto, Docker asegura que los contenedores de los servidores de configuración y Eureka son creados antes de que el contenedor del Catálogo sea creado.

Una vez tengamos definido el fichero docker-compose.yml bastará con que lo ejecutemos para poner en marcha la aplicación al completo. La instrucción a ejecutar desde un terminal es la siguiente:

```
$ docker-compose up --build
```

La instrucción que deberíamos ejecutar para detener la ejecución del entorno completo sería la siguiente:

```
$ docker-compose down
```

MATERIAL ADICIONAL

El material adicional de este libro puede descargarlo en nuestro portal web: *http://www.ra-ma.es*.

Debe dirigirse a la ficha correspondiente a esta obra, dentro de la ficha encontrará el enlace para poder realizar la descarga. Dicha descarga consiste en un fichero ZIP con una contraseña de este tipo: XXX-XX-XXXX-XXX-X la cual se corresponde con el ISBN de este libro.

Podrá localizar el número de ISBN en la página IV (página de créditos). Para su correcta descompresión deberá introducir los dígitos y los guiones.

Cuando descomprima el fichero obtendrá los archivos que complementan al libro para que pueda continuar con su aprendizaje.

INFORMACIÓN ADICIONAL Y GARANTÍA

- RA-MA EDITORIAL garantiza que estos contenidos han sido sometidos a un riguroso control de calidad.
- Los archivos están libres de virus, para comprobarlo se han utilizado las últimas versiones de los antivirus líderes en el mercado.
- RA-MA EDITORIAL no se hace responsable de cualquier pérdida, daño o costes provocados por el uso incorrecto del contenido descargable.
- Este material es gratuito y se distribuye como contenido complementario al libro que ha adquirido, por lo que queda terminantemente prohibida su venta o distribución.

ÍNDICE ALFABÉTICO

Microservicios

Un enfoque integrado

David Roldán Martínez
Pedro J. Valderas Aranda
Victoria Torres Bosch

Desde **www.ra-ma.es** podrá descargar material adicional.

Ra-Ma®