Listado 5-20. application.properties: Modificación de la URL de JDBC (gamificación v8)

```
#. . .
# Crea la base de datos en un archivo
spring.datasource.url = jdbc: h2: file: ~ / gamification; DB_CLOSE_ON_ EXIT = FALSE; AUTO_SERVER = TRUE
```

Tenga en cuenta que, como resultado de nuestra nueva estrategia de datos, la lógica de microservicio puede escalar bien, pero no podemos decir lo mismo de la base de datos: sería solo una instancia compartida. Para resolver esa parte en un sistema de producción, tendríamos que elegir un motor de base de datos que escala y crear un clúster en nuestro nivel de base de datos. Dependiendo del motor de base de datos que elijamos, el enfoque puede ser diferente. Por ejemplo, H2 tiene un modo de agrupación simple que se basa en la replicación de datos y MariaDB usa Galera, que también proporciona equilibrio de carga. La buena noticia es que, desde el punto de vista del código, podemos manejar un clúster de base de datos como una URL JDBC simple, manteniendo toda la lógica del nivel de la base de datos a un lado de nuestro proyecto.

Arquitectura basada en eventos y equilibrio de carga

Desde el punto de vista de la comunicación REST, podemos concluir que el sistema funciona correctamente gracias al motor de base de datos compartida. Independientemente de qué instancia maneje nuestra solicitud, el resultado de publicar un intento o recuperar datos será coherente. Pero, ¿qué sucede con el proceso que abarca ambos microservicios?

Nuestro sistema maneja el proceso empresarial *intento de apuntar* basado en un evento desencadenado desde el microservicio de multiplicación y consumido desde el microservicio de gamificación. La pregunta ahora es, ¿cómo funciona eso si comenzamos más de una instancia de gamificación?

En este caso, todo funcionará. *bastante bien* sin ninguna modificación. Cada instancia de gamificación actuará como un*obrero* que se conecta a una cola compartida en RabbitMQ. Solo una instancia consume cada evento, procesos

y almacena el resultado en la base de datos compartida. De todos modos, en un sistema que pasa a producción, normalmente necesitaríamos ajustar la configuración actual para minimizar el impacto si el mismo evento se recibe dos veces, y también tratar de evitar que los eventos se pierdan debido a que un microservicio muere inesperadamente. La guía de confiabilidad de RabbitMQ (consultehttps://www.rabbitmq.com/ confiabilidad.html) es un buen punto de partida si desea obtener más información sobre lo que puede hacer para prevenir o reaccionar ante algunos problemas diferentes que pueden ocurrir en su sistema.

Por último, pero no menos importante, RabbitMQ también puede funcionar en grupos. En un entorno de producción, necesitamos configurar nuestra infraestructura de esa manera para que el sistema siga funcionando incluso si una de las instancias del servidor RabbitMQ deja de funcionar. De manera similar a la URL de base de datos única para un clúster, desde el punto de vista de nuestro código, nada cambia: nos conectaríamos a RabbitMQ como si solo hubiera una instancia. Ver figura5-15.

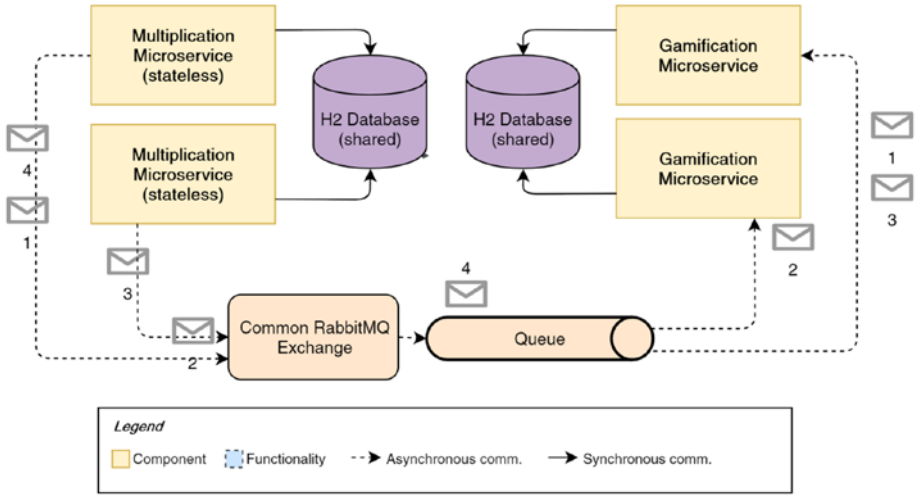

Figura 5-15. *El sistema funciona con múltiples instancias*

Equilibrio de carga con cinta

¡Es hora de algunas cosas interesantes! Ahora que sabemos que nuestros microservicios pueden escalar, nuestro objetivo es ponerlo en práctica con uno de nuestros servicios y usar el equilibrio de carga para redirigir las solicitudes a las múltiples instancias. Proporcionar alta disponibilidad (o*Resiliencia)* es una de las características más importantes que debe tener en su sistema distribuido: los servicios pueden fallar, una de las áreas geográficas en las que se implementa su servicio podría no responder bien, el servicio podría estar saturado de tráfico, etc.

Como se presentó anteriormente, Spring Cloud Netflix Ribbon es una buena opción para implementar el equilibrio de carga con Spring Boot. Viene con Eureka, por lo que combina muy bien con Zuul. Tenga en cuenta que, aunque los cubrimos por separado, el uso de Eureka sin Ribbon (o el descubrimiento de servicios sin equilibrio de carga) no es un escenario típico, ya que en ese caso solo se beneficiará de mapear una ubicación física (IP y puerto) a un alias de servicio (uno a uno).

Dicho esto, no es sorprendente descubrir que para incluir Ribbon en nuestro servicio Gateway no tenemos que hacer nada más. Ribbon viene con Eureka por defecto, y Spring Boot lo configura automáticamente. Agregaremos alguna configuración adicional más adelante en esta subsección para anular los valores predeterminados, pero, por ahora, juguemos un poco con la configuración estándar.

Para verificar rápidamente que el equilibrio de carga funciona, agregaremos algunos registros a nuestro /aleatorio punto final en el servicio de multiplicación, que imprimirá una línea en el registro de servicio que indica el número de puerto. Podemos usar Lombok's @Slf4j anotación, que ofrecerá una anotación inicializada Iniciar sesión con el cual podemos imprimir un mensaje en consola que contiene la propiedad inyectada Puerto de servicio. Tenga en cuenta que debemos agregar explícitamente Puerto de servicio para nuestro solicitud. propiedades (y configúrelo en 8080); de lo contrario, no se encontrará la propiedad. Ver listado 5-21.

Listado 5-21. MultiplicationController.java Adición de registros (multiplicación v8)

```
@ Slf4j
@RestController
@RequestMapping ("/ multiplicaciones")
final clase MultiplicationController {

    privado final MultiplicationService multiplicationService;

    privado final int serverPort;

    @Autowired
    público MultiplicationController (final MultiplicationService
    multiplicationService, @Value ("$ {server.port}") int serverPort) {

        esto.multiplicationService = multiplicationService;
        esto.serverPort = serverPort;
    }

    @GetMapping ("/ aleatorio")
    Multiplicación getRandomMultiplication () {
        log.info ("Generando una multiplicación aleatoria desde
        el servidor @ {}", serverPort);
        regreso multiplicationService.createRandom
        Multiplication ();
    }

}
```

EJERCICIO

También nos gustaría imprimir una línea en el registro cuando se llama al punto final de resultados. eso significa que necesita implementar una solución similar en el MultiplicationResultAttemptController clase. en este caso, tu invocar el registro con este comando:

log.info ("Recuperando resultado {} del servidor @ {}", resultId, serverPort);

Jugando con el equilibrio de carga

Ahora podemos volver a poner en marcha todos los servicios de nuestro sistema distribuido, como hicimos en el apartado anterior. Una vez que tengamos todo en funcionamiento, queremos iniciar una segunda instancia de nuestro microservicio de multiplicación.

Esta instancia de servicio está vinculada a un número de puerto. Para iniciar una nueva instancia de un servicio dado, necesitamos cambiar ese número de puerto (Puerto de servicio) para evitar conflictos, lo cual es muy fácil de lograr porque tenemos varias formas de anular las propiedades de Spring Boot. Para iniciar una segunda instancia de nuestro servicio de multiplicación usando Maven, podemos ejecutar cualquiera de los dos comandos que se muestran en el Listado5-22 (tenga en cuenta que para poder ejecutar el segundo comando, primero debe empaquetar su aplicación Spring Boot en un archivo JAR).

Listado 5-22. Consola: ejecución de una segunda instancia de multiplicación (multiplicación v8)

~ / book / code / v8 / social-multiplication $ ./mvnw spring-boot: ejecutar - Drun.arguments = "- server.port = 8180" ~ / book / code / v8 / social-multiplication / target $ java -jar socialmultiplication-v8-0.8.0-SNAPSHOT.jar --server.port = 8180

Como puede ver, cambiar el puerto del servidor es simple. Pasamos un argumento que anula elPuerto de servicio propiedad para usar el puerto 8180 en lugar del predeterminado 8080. Ahora, deberíamos tener dos instancias de nuestro servicio de multiplicación ejecutándose al mismo tiempo en ambos puertos y compartiendo la misma base de datos (que es inicializada por la primera instancia en ejecución).

Si vamos al Eureka Server Dashboard (http: // localhost: 8761 /), Deberíamos ver las dos instancias de nuestro microservicio de multiplicación registradas en Eureka, como se muestra en la Figura 5-16.

Figura 5-16. *Eureka Server Dashboard muestra que dos instancias de nuestro microservicio de multiplicación están registradas*

Veamos ahora que funcionan ambas instancias. Navegue hasta el cliente de IU como de costumbre (http: // localhost: 9090 / ui / index.html) y actualice la página varias veces (llamando así a /aleatorio punto final). Luego, verifique los registros de sus instancias de servicio de multiplicación en los puertos8080 y 8180. Verá las líneas de registro allí, para que pueda verificar cómo Ribbon está realizando una estrategia simple de operación por turnos y redirigiendo cada solicitud a un servicio diferente cada vez.[2] Veremos cómo cambiar esta estrategia de equilibrio de carga más adelante.

Para hacerlo más interesante, eliminemos una de las instancias de multiplicación (por ejemplo, cerrando la ventana del terminal o presionando Ctrl + C). En teoría, esa instancia debería eliminarse inmediatamente del registro, y todo el tráfico se redirigiría a la única instancia que sigue viva. En la práctica, puede descubrir que una de cada dos veces que actualiza la página obtiene errores desagradables en la salida del registro de la puerta de enlace, como se muestra en el listado.5-23.

[2]Tenga en cuenta que pueden pasar algunos segundos antes de que ambas instancias se registren en Eureka y el equilibrador de carga inicie la estrategia de operación por turnos. ¡Se paciente!

Listado 5-23. Salida de registro de la puerta de enlace después de matar una instancia (multiplicación v8)

```
2017-09-27 18: 30: 55.798 WARN 14012 --- [nio-8000-exec-7]
oscnzfilters.post.SendErrorFilter: Error durante
filtración
com.netflix.zuul.exception.ZuulException: error de reenvío
        en org.springframework.cloud.netflix.zuul.filters.
route.RibbonRoutingFilter.handleException (RibbonRoutingFi
lter.java:183) ~ [spring-cloud-netflix-core-1.3.1.RELEASE. jar:
1.3.1.RELEASE]
    . . .
Causado por: com.netflix.hystrix.exception. HystrixRuntimeException: se
agotó el tiempo de espera de la multiplicación y no hay respaldo disponible.

        en com.netflix.hystrix.AbstractCommand $ 22. llamar
(AbstractCommand.java:819) ~ [hystrix-core-1.5.12. tarro:
1.5.12]
        . . .
Causado por: java.util.concurrent.TimeoutException: null
        en com.netflix.hystrix.AbstractCommand.handleTimeoutVi
aFallback (AbstractCommand.java:997) ~ [hystrix-core-1.5.12.
tarro: 1.5.12]
```

Después de dos o tres minutos, puede volver a intentar enviar varias solicitudes. En ese momento, debería funcionar como se esperaba: todas las solicitudes se redirigirán a la única instancia disponible. Dentro de ese período, puede verificar en el panel de Eureka cómo se tarda ese tiempo en darse cuenta de que la instancia ya no está viva. Veremos a continuación cómo podemos mejorar este comportamiento ajustando nuestras herramientas.

¡LO HICIMOS! EL GATEWAY DE API, EL DESCUBRIMIENTO DEL SERVICIO Y LA CARGA ¡EL EQUILIBRADOR ESTÁ APLICADO AHORA!

¡Desarrollamos nuestro sistema distribuido a una arquitectura de microservicios adecuada! Dividimos la interfaz de usuario y agregamos la puerta de enlace, el descubrimiento de servicios y el equilibrio de carga. Implementamos una buena solución para los requisitos de la historia de usuario 3. Todavía hay algunas partes que deberían mejorarse, ¡pero podemos celebrar nuestro éxito!

Disponibilidad de producción: ampliación del registro de servicios

Tenga en cuenta que, si implementa un sistema como este en producción, también debe proporcionar alta disponibilidad para el registro. esta función viene lista para usar con el servidor eureka, y puede verla en la documentación (en la sección de "conciencia de los pares" en http://projects.spring.io/springcloud/spring-cloud.html#_peer_awareness) lo fácil que es configurar , simplemente creando un perfil por instancia que desee que esté disponible.

Ajuste de la estrategia de equilibrio de carga

Como acabamos de ver, Spring Boot configura Ribbon de forma predeterminada con una estrategia de roundrobin para el equilibrio de carga. También establece el mecanismo de verificación de estado para*ninguno (*inyectando el NoOpPing implementación de la interfaz IPing, consulte https://tpd.io/cust-rb para mas detalles). Eso implica que el equilibrador de carga no verificará si los servicios están activos (NoOpPing simplemente devuelve verdadero para el isAlive método). Tiene sentido desde un punto de vista conceptual ya que debería ser nuestro registro de servicios, Eureka, el que registra y da de baja las instancias.

Sin embargo, Eureka tarda mucho en darse cuenta de que un servicio se interrumpió inesperadamente (en mi experiencia, lleva un promedio de tres minutos). No les está haciendo ping sino comprobando*arrendamientos:* cada instancia necesita ponerse en contacto con el registro después de un tiempo (30 segundos por defecto) para *renovar el contrato de arrendamiento*podemos imaginarlo como la instancia que dice "¡Estoy vivo!"). Después de

más tiempo (90 segundos de forma predeterminada), el registro de servicios anulará el registro de las instancias que no renovaron la concesión en ese período de tiempo. Cambiando elleaseRenewalIntervalInSeconds El parámetro puede parecer una buena idea, pero la documentación oficial lo desaconseja.[3] Como resultado, las instancias que se caigan inesperadamente no se eliminarán del registro inmediatamente, sino después de un período de minutos, y durante ese tiempo nuestra aplicación fallará.

Podemos resolver este problema utilizando una funcionalidad Ribbon para hacer ping a los servicios y aplicar el equilibrio de carga según el resultado (por lo tanto, tenemos cierta lógica en el lado del cliente para detectar si las instancias están inactivas). Para que funcione, necesitamos configurar dos Spring beans: unIPing para anular el mecanismo de verificación de estado predeterminado y una Yo mando para cambiar la estrategia de equilibrio de carga predeterminada. Además, necesitamos anotar nuestra clase principal GatewayApplication para apuntar a esta clase de configuración. Ver listados5-24 y 5-25.

Listado 5-24. RibbonConfiguration.java (puerta de enlace v8)

```
paquete microservices.book.gateway.configuration;

importar com.netflix.client.config.IClientConfig;
importar com.netflix.loadbalancer. *;
importar org.springframework.context.annotation.Bean;

clase pública RibbonConfiguration {

    @Frijol
    público IPing ribbonPing (configuración final de IClientConfig) {
        volver nuevo PingUrl (falso,"/salud");
    }
```

[3] https://tpd.io/ek-rnew

```
    @Frijol
    público IRule ribbonRule (configuración final de IClientConfig) {
        volver nuevo AvailabilityFilteringRule ();
    }

})
```

Listado 5-25. GatewayApplication.java (puerta de enlace v8)

```
@EnableZuulProxy
@EnableEurekaClient
@RibbonClients (defaultConfiguration = RibbonConfiguration.
Clase)
@SpringBootApplication
clase pública GatewayApplication {

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

- Tenga en cuenta que RibbonConfiguration no está anotado con @Configuración. Se inyecta de forma diferente. Necesitamos hacer referencia a él desde una nueva anotación agregada a la clase de aplicación principal llamada @RibbonClients. La razón es que, opcionalmente, podríamos tener varios clientes de cinta con diferentes configuraciones de equilibrio de carga.
- La PingUrl La implementación comprobará si los servicios están activos. Cambiamos la URL predeterminada y la apuntamos a / salud ya que sabemos que existe el punto final (está incluido por Spring Actuator). Lafalso La bandera es solo para indicar que el punto final no está protegido.

- La AvailabilityFilteringRule es una alternativa a la predeterminada RoundRobinRule. También recorre las instancias pero, además de eso, tiene en cuenta la disponibilidad que están verificando nuestros nuevos pings para omitir algunas instancias en caso de que no respondan.

Si ahora probamos el escenario donde se registran varias instancias y eliminamos una de ellas, notaremos que el tiempo de reacción para redirigir todo el tráfico a la única instancia viva es mucho menor. Tenga en cuenta que el estado del Registro de servicios de Eureka dentro de ese tiempo será exactamente el mismo que antes: aún se necesita tiempo para cancelar el registro de la instancia. La mejora es *en el lado del cliente (el balanceador de carga):* la puerta de enlace comprueba que la instancia no funciona realmente y elige otra.

Esta configuración es solo un ejemplo. Puede encontrar otras opciones para estrategias de equilibrio de carga en el repositorio oficial.[4] Hay implementaciones que nos permiten equilibrar la carga en función del tiempo de respuesta, afinidad geográfica, etc. La mejor idea para un entorno de producción adecuado es diseñar nuestro plan, probarlo (poniendo algo de carga en su sistema y monitoreando los resultados), y luego ajústelo en función de los resultados.

Figura 5-17 muestra nuestra vista lógica actualizada, con detección de servicios y equilibrio de carga en su lugar. Introdujimos una vista similar al explicar los conceptos, pero ahora la hicimos realidad en nuestro código fuente.

[4] https://tpd.io/lb-opts

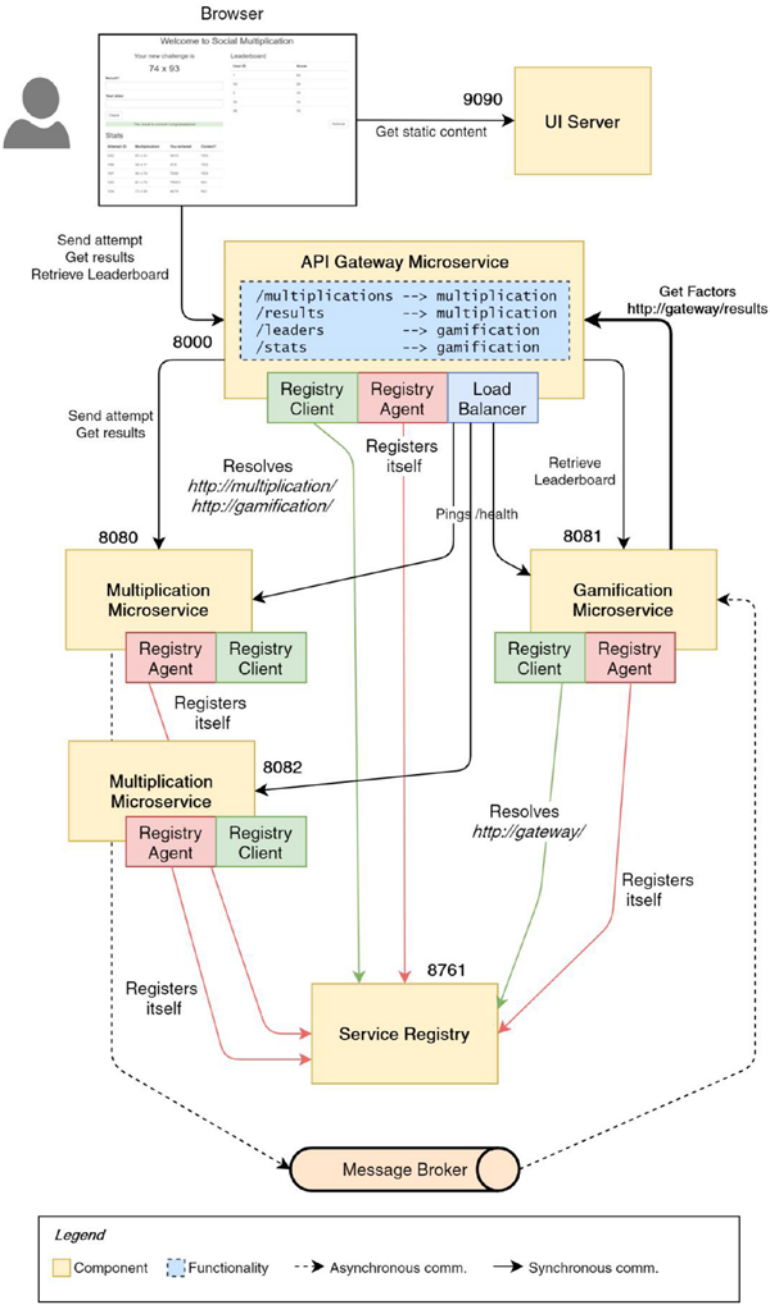

Figura 5-17. *Vista lógica actualizada, con descubrimiento de servicios y equilibrio de carga en su lugar*

Disyuntores y clientes REST

Disyuntores con Hystrix

En el mundo real, ocurren errores. Los servicios pueden ser inaccesibles o no responder a tiempo. Nuestro sistema distribuido no debería fallar en su totalidad porque una de las partes no respondió. El patrón del disyuntor es la solución para aquellos escenarios en los que es probable que todo nuestro sistema falle si una de las partes no responde.

Este patrón se basa en dos estados. Si el circuito está cerrado, eso significa que la solicitud puede llegar a su destino y se recibe la respuesta. Todo está funcionando bien. Si hay errores o expira un tiempo de espera, el punto de conexión está inactivo y el circuito se abre. Eso implica llamar a un*diferente parte del sistema,* que actuará como un reproductor de respaldo, dando una respuesta predeterminada en caso de falla. El resultado es una respuesta manejable que el sistema puede manejar sin más errores.

Spring Cloud Netflix también contiene una implementación conocida de este patrón, llamada *Hystrix.* En nuestro proyecto, uno de los lugares donde podemos conectarlo es a la puerta de enlace, por lo que puede proporcionar una respuesta predeterminada cuando un servicio no responde. En realidad, si revisamos el seguimiento de la pila que obtuvimos cuando jugamos con el equilibrio de carga y matamos una de las instancias de multiplicación, podemos ver que nos está dando una pista sobre qué hacer, como se muestra en el Listado.5-26.

Listado 5-26. Salida de la consola después de eliminar una instancia de multiplicación (puerta de enlace v8)

```
. . .
Causado por: com.netflix.hystrix.exception. HystrixRuntimeException: se
agotó el tiempo de espera de la multiplicación y no hay respaldo disponible.

. . .
```

Hystrix y Zuul

Para conectar Zuul con Hystrix, podemos usar el Proveedor de ZuulFallback interfaz. Si inyectamos un bean implementando esta interfaz en nuestro contexto Spring Boot, podremos proporcionar *Respaldos de Hystrix (*respuestas HTTP predefinidas) cuando la puerta de enlace API no puede acceder a un servicio. Cuando Zuul no puede redirigir la solicitud, verificará si hay una reserva para ese servicio específico (usando elgetRoute () métodos de disponibilidad Proveedor de ZuulFallbacks). Si hay una, construirá y devolverá una respuesta predeterminada (usando elfallbackResponse () método).

Usémoslo para la multiplicación, como ejemplo (en la vida real deberíamos hacer eso para todas las rutas para las que podemos proporcionar una alternativa). Enviaremos un mensaje de error incrustado enfactorA Cuándo /aleatorio se llama y el servicio no está disponible, por lo que se mostrará en lugar de la multiplicación para resolver. Ver listado 5-27.

Listado 5-27. HystrixFallbackConfiguration.java (puerta de enlace v8)

```
@Configuración
clase pública HystrixFallbackConfiguration {

    @Frijol
    público ZuulFallbackProvider zuulFallbackProvider () {
        volver nuevo ZuulFallbackProvider () {

            @Anular
            público String getRoute () {
                // Puede resultar confuso: es la propiedad
                serviceId y no la ruta
                regreso "multiplicación";
            }

            @Anular
            público ClientHttpResponse fallbackResponse () {
```

```
        volver nuevo ClientHttpResponse () {

            @Anular
            público HttpStatus getStatusCode () lanza
            IOException {
                regreso HttpStatus.OK;
            }

            @Anular
            público int getRawStatusCode () lanza
            IOException {
                regreso HttpStatus.OK.value ();
            }

            @Anular
            público String getStatusText () lanza
            IOException {
                regreso HttpStatus.OK.toString ();
            }

            @Anular
            público vacío cerrado () {}

            @Anular
            público InputStream getBody () lanza
            IOException {
                volver nuevo ByteArrayInputStream ("{\"
                factorA \ ": \" ¡Lo sentimos, el servicio no
                funciona! \ ", \" FactorB \ ": \"? \ ", \" Id \ ":
                null}". GetBytes ());
            }

            @Anular
            público HttpHeaders getHeaders () {
```

```
                    Encabezados HttpHeaders = nuevo
                    HttpHeaders ();
                    headers.setContentType (MediaType.
                    APPLICATION_JSON);
                    headers.setAccessControlAllowCredential s (
                    cierto);
                    headers.setAccessControl
                    AllowOrigin ("*");
                    regreso encabezados;
                }
            };
        }
    };
  }
}
```

No es una interfaz muy amigable, pero es lo que brindan. En nuestro caso, simplemente adaptamos el cuerpo al formato esperado para /aleatorio (a Multiplicación object en JSON) pero inserte nuestro mensaje en lugar de los factores reales. Tenga en cuenta que también necesitamos agregar los encabezados CORS a la respuesta, ya que este no es procesado automáticamente por nuestroWebConfiguration.

Para probar nuestro nuevo respaldo, podemos repetir nuestro último experimento. Comenzamos cada microservicio como de costumbre y una segunda instancia de multiplicación. Después de un tiempo en el que verificamos que el equilibrio de carga funciona bien, matamos una de las instancias de multiplicación. Veremos nuestro respaldo en acción devolviendo la respuesta predefinida, que se mostrará en nuestra interfaz de usuario. Tenga en cuenta que no es una solución perfecta, pero al menos le damos algo de información al usuario sobre lo que sucedió en lugar de dejar los factores de multiplicación vacíos. Ver figura5-18.

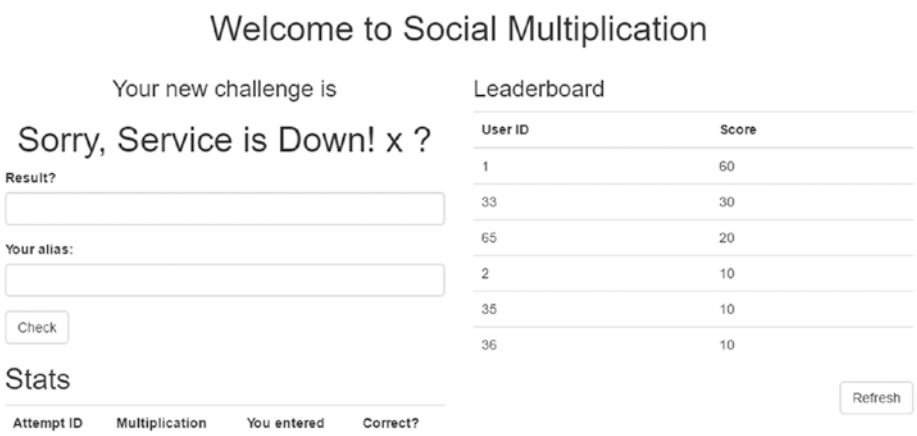

Figura 5-18. *Interfaz de usuario que muestra que el servicio no funciona*

Hystrix proporciona muchas más funciones que su integración con Zuul. Puede usarlo desde cualquier consumidor REST que desarrolle simplemente agregando anotaciones como @HystrixCommand y @EnableCircuitBreaker, y luego configurar los fallbacks. Primero cubrimos el caso particular de la integración con Zuul ya que es uno de los más complicados; ahora veamos cómo hacerlo funcionar con un cliente REST estándar.

Hystrix desde un cliente REST

Tenemos otro punto en nuestro sistema donde un interruptor encaja perfectamente: la llamada a la API REST de la gamificación a la multiplicación para comprobar si uno de los factores es el número de la suerte. Nuestro proceso comercial no debería fallar si no tiene acceso al servicio en ese momento. Tenemos dos opciones para la respuesta alternativa: incluir el número de la suerte o no. Seamos codiciosos esta vez: el usuario no recibirá esa insignia si el servicio no funciona.

Primero, agreguemos Hystrix al microservicio de gamificación. Para hacer eso, necesitamos incluir una nueva dependencia en elpom.xml archivo, como se muestra en el listado 5-28.

Listado 5-28. pom.xml Adición de Hystrix (gamificación v8)

```
<dependencia>
  <groupId>org.springframework.cloud </groupId> <artifactId>
 primavera-nube-arrancador-hystrix </artifactId> </dependency>
```

Ahora vamos a la implementación de nuestro cliente REST, MultiplicationResultAttemptClientImpl, y anotar el método usando RestTemplate con HystrixCommand. Aparte de eso, creamos nuestro método para devolver la respuesta predeterminada, defaultResult, que devuelve dos factores que no son el número de la suerte. Ver listado5-29.

Listado 5-29. MultiplicationResultAttemptClientImpl.java Adición de Hystrix (gamificación v8)

```
@Componente
clase MultiplicationResultAttemptClientImpl implementos
MultiplicationResultAttemptClient {

    privado RestTemplate final restTemplate;
    privado final String multiplicationHost;

    @Autowired
    público MultiplicationResultAttemptClientImpl (final
    RestTemplate restTemplate,

@Value ("$ {multiplicationHost}") final String multiplicationHost) {
        esto.restTemplate = restTemplate;
        esto.multiplicationHost = multiplicationHost;
    }
```

```
    @HystrixCommand (fallbackMethod = "defaultResult")
    @Override
    público MultiplicationResultAttempt retrieveMultiplicationRe
    sultAttemptbyId (final Long multiplicationResultAttemptId) {
        regreso restTemplate.getForObject (
                multiplicationHost + "/ results /" +
                multiplicationResultAttemptId,
                MultiplicationResultAttempt.class);
    }

    privado MultiplicationResultAttempt defaultResult (final
    Long multiplicationResultAttemptId) {
        volver nuevo MultiplicationResultAttempt ("fakeAlias",
                10, 10, 100, cierto);
    }
}
```

El último cambio que debemos hacer en la gamificación es agregar el @EnableCircuitBreaker anotación a nuestra clase principal, como se muestra en el Listado 5-30.

Listado 5-30. GamificationApplication.java Adición de Hystrix (gamification v8)

```
paquete microservices.book;

importar org.springframework.boot.SpringApplication;
importar org.springframework.boot.autoconfigure.
SpringBootApplication;
importar org.springframework.cloud.client.circuitbreaker.
EnableCircuitBreaker;
importar org.springframework.cloud.netflix.eureka.
EnableEurekaClient;
```

```
@EnableEurekaClient
@EnableCircuitBreaker
@SpringBootApplication
clase pública GamificationApplication {

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

}
```

Probar que todo funciona como se esperaba no es fácil si nos limitamos a un escenario real. Necesitamos enviar a través de la interfaz de usuario un intento correcto; que pasará por el microservicio de multiplicación, y luego tenemos que matarlo después de que envíe elMultiplicationSolvedEvent, por lo que la gamificación no puede alcanzarlo. Afortunadamente, hay una solución más simple para probar la respuesta predefinida de Hystrix: podemos simplemente modificar la propiedad del host de multiplicación dentro de la gamificación.application.properties archivo a una URL inexistente (algo así como multiplicationHost = http: // localhost: 8001 / api).

Ahora podemos probarlo. Comenzamos de nuevo con todo nuestro conjunto de microservicios (como lo hicimos antes). Solo necesitamos una instancia de multiplicación esta vez. Con la nueva versión de gamificación apuntando a una URL de multiplicación incorrecta, Hystrix hará su trabajo y devolverá la respuesta predeterminada cada vez que publiquemos un intento correcto. Si desea volver a comprobar que Hystrix está funcionando (como un desarrollador escéptico clásico), puede depurar el microservicio de gamificación y establecer un punto de interrupción dentro deldefaultResult () método.

DESCANSAR CONSUMIDORES

No podemos terminar este capítulo sin mencionar *Fingir,* ya que es otro miembro famoso de Spring Cloud Netflix.

Feign nos permite consumir servicios REST como si fueran parte de nuestro código. Podemos generar @FeignClientsy mapee sus métodos a las solicitudes. La principal ventaja es que podemos evitar manejar solicitudes directamente con @RestTemplates en nuestro código, por lo que podemos tratar las interfaces externas como si fueran parte de nuestra base de código. Puede ver un ejemplo dentro de la documentación oficial (verhttps://tpd.io/feigndoc), la StoreClient interfaz, en el listado 5-31.

Listado 5-31. StoreClient.java Ejemplo de documentación

```
@FeignClient ("tiendas")
interfaz pública StoreClient {
    @RequestMapping (método = RequestMethod.GET, value = "/
    stores") List <Store> getStores ();

    @RequestMapping (método = RequestMethod.POST, value =
    "/ stores / {storeId}", consumes = "application / json")
    Actualización de la tienda (@PathVariable ("storeId") StoreId largo, Store store);
}
```

Feign combina bien con Eureka, Ribbon e Hystrix. El cliente de Feign utiliza Eureka y Ribbon para buscar servicios y realizar el equilibrio de carga. Además, utiliza anotaciones a nivel de interfaz para especificar qué clases contienen alternativas de Hystrix.

En nuestro sistema, podríamos usarlo para eliminar el MultiplicationResultAttemptClientImpl class y use solo la interfaz con algunas anotaciones. También necesitamos mover en ese caso el método de reserva de Hystrix a una clase separada.

No lo usaremos en el libro ya que los beneficios de usar Feign no son tan valiosos para nosotros: está orientado al caso en el que los servicios se llaman entre sí directamente, sin usar la puerta de enlace API. Se necesita tiempo para configurarlo y hacerlo funcionar con el resto de herramientas y, en nuestro caso, podemos lograr el mismo objetivo con unas pocas líneas de código usandoRestTemplate.

Patrones de microservicios y PaaS

En las secciones anteriores, cubrimos algunos patrones importantes que deberíamos aplicar a una arquitectura de microservicios: descubrimiento de servicios, equilibrio de carga, puerta de enlace API e interruptores automáticos. Vimos que existen herramientas para implementarlas, siendo Spring Cloud Netflix la solución de facto para Spring Boot. En el camino, presentamos un nuevo microservicio de registro de servicios y un microservicio de puerta de enlace API, que ahora forman parte de nuestro ecosistema.

Ahora que comprende todos estos patrones y cómo funcionan, es posible que se pregunte: ¿debo ocuparme de toda esta plomería cada vez que quiero configurar una arquitectura de microservicios? ¿No hay algún tipo de marco que incluya todo esto para mí, desde un nivel de abstracción más alto que Spring? ¿No podría concentrarme en escribir mi aplicación Spring Boot y ponerla *en algún lugar* entonces funciona directamente?

La respuesta común a estas preguntas es que puede abstraer muchos de estos patrones utilizando una solución de plataforma como servicio (PaaS). Estas plataformas contienen no solo detección de servicios, equilibrio de carga, enrutamiento (puerta de enlace API) y patrones de disyuntor, sino también registro centralizado y autenticación integrada, entre otras funcionalidades. Estas plataformas generalmente residen en la nube, y sus proveedores le ofrecen planes para suscribirse y usar no solo esos patrones, sino también su almacenamiento, CPU, red, etc.

Hay muchos PaaS diferentes de muchos proveedores diferentes: Amazon AWS, Google App Engine, CloudFoundry de Pivotal, Microsoft Azure, etc. Todos ofrecen servicios similares. Lo primero que debe hacer es "empaquetar" sus microservicios (por ejemplo, usando un *buildpack)* e implementarlos directamente en la plataforma, donde se descubren automáticamente después de alguna configuración básica. No es necesario implementar un registro de servicios o una puerta de enlace porque son parte de la plataforma: solo configura algunas reglas de enrutamiento y políticas de equilibrio de carga. Las bases de datos y los corredores de mensajes se ofrecen como servicios elásticos que escalan de forma transparente a pedido. Los crea utilizando un asistente y luego obtiene las URL para usarlos directamente en sus aplicaciones.

Si desea ver un ejemplo de lo fácil que es implementar una aplicación Spring Boot en una de estas plataformas, puede consultar la Guía de CloudFoundry para implementar una aplicación Spring (consulte *https://tpd.io/ cf-gs).* Dentro del mismo sitio de documentación, también puede verificar qué tan fácil es escalar los servicios (simplemente ejecutando un comando cf scale myApp -i 5 para obtener cinco instancias) y cómo funciona el enrutamiento asignando varias instancias al mismo nombre de host. No es sorprendente descubrir que Pivotal ofrece un disyuntor como servicio (basado en Hystrix).

La ventaja que tenemos en este punto es que sabemos lo que necesitamos cuando diseñamos nuestra arquitectura de microservicios. Podemos equilibrar todas estas opciones y decidir dónde y cómo queremos implementar los patrones. Dependiendo de nuestras necesidades, tiempo y presupuesto, podemos elegir una solución basada en implementar todo nosotros mismos o confiar en algunas plataformas o frameworks.

Resumen

En este capítulo, aprendimos algunos de los conceptos más importantes que rodean a los microservicios: descubrimiento de servicios, equilibrio de carga, puerta de enlace API e interruptores automáticos. Usamos las herramientas disponibles en Spring Cloud Netflix para implementar estos patrones (Eureka, Ribbon, Zuul e Hystrix), y logramos que nuestros servicios estén bien conectados entre ellos, lo que respalda la alta disponibilidad a través del escalado.

Pasamos por varios pasos, ilustrados con diagramas, que nos ayudaron a comprender por qué necesitamos estas herramientas. Luego, en la segunda parte del capítulo, los incluimos en nuestra base de código utilizando un enfoque incremental y experimentamos con las características de equilibrio de carga y disyuntor de nuestro sistema.

Este libro, aunque práctico, le enseña las ideas detrás de los patrones. En este momento, puede diseñar su arquitectura de microservicios utilizando las herramientas que se muestran en este libro o explorar otras alternativas para obtener el mismo resultado. Como ejemplo, vimos cómo las soluciones PaaS implementan estos patrones y pueden acelerar el tiempo de desarrollo de su proyecto si se ajusta a sus requisitos y presupuesto.

Hemos terminado nuestro sistema implementando todos los requisitos que teníamos y evolucionándolo hasta llegar a una buena arquitectura de microservicios con Spring Boot. El siguiente capítulo (el último) se centra en resolver un desafío adicional en el mundo de los microservicios: las pruebas de integración de un extremo a otro.

CAPÍTULO 6

Probando el Sistema distribuido

Introducción

En capítulos anteriores, construimos un sistema distribuido complejo, compuesto por tres *microservicios funcionales*UI, multiplicación, gamificación) y dos *soporte de microservicios*la puerta de enlace API construida con Zuul y el registro de servicios implementado con Eureka). Además, nuestro sistema utiliza un enfoque basado en eventos para cumplir con los procesos comerciales que abarcan varios microservicios (en este caso, solo*intento de puntos).*

Cuando tienes un entorno como este, compuesto por muchas piezas, es muy probable que una de ellas falle. También sucede con un sistema monolítico: es posible que tenga módulos o componentes que deben pegarse entre sí. Pero, en una arquitectura de microservicios, es aún más crítico verificar que todos los componentes (los microservicios) funcionen juntos, dado que estas partes se pueden construir e implementar de forma independiente.

Pueden evolucionar de diferentes maneras: podríamos, por ejemplo, introducir un cambio en la API REST de Multiplication para cambiar el nombre factorA y factorB a factor1 y factor2. Podríamos cambiar nuestras pruebas unitarias y hacer que la compilación pase con éxito para ese microservicio específico. Sin embargo, con ese cambio

M. Macero, *LearnMicroservices con Spring Boot,*
https://doi.org/10.1007/978-1-4842-3165-4_6

estaríamos rompiendo la funcionalidad general: la gamificación está utilizando esa API REST para realizar su lógica (al verificar los factores para el número de la suerte). Tendríamos un problema similar si cambiamos algo en el MultiplicationSolvedEvent.

Tener suites de prueba que verifiquen los microservicios de forma independiente no es suficiente: necesitamos una buena estrategia para verificar que todos los casos de uso de un extremo a otro funcionan después de los cambios.

Pruebas unitarias, pruebas de integración, pruebas de componentes, pruebas de contrato y pruebas de un extremo a otro: debe confiar en todas ellas al crear microservicios. Si no está muy familiarizado con estos diferentes tipos de pruebas, ahora puede pausar su lectura por un momento para ver esta presentación sobre las pruebas de microservicio enhttps://martinfowler.com/articles/ microservice-testing /.

Entre todo tipo de pruebas, las pruebas de extremo a extremo se encuentran en la parte superior de la pirámide: no debería haber muchas en su sistema, ya que son difíciles de mantener. Pero, por otro lado, son ellos los que protegen sus procesos comerciales con la integración de todos sus microservicios juntos, por lo que tener menos no significa que sean menos importantes.

En este capítulo, nos centraremos en las pruebas de un extremo a otro para una arquitectura de microservicios (nuestro sistema). El motivo es que suelen ser los más difíciles de implementar y mantener, por lo que merece la pena dedicar un capítulo de este libro. Cubriremos algunas buenas prácticas con el marco de Cucumber, para mantenerlas simples y enfocadas en el negocio.

Comenzamos a codificar nuestra aplicación utilizando un enfoque TDD. Ahora seguiremos una estrategia similar con estas pruebas de extremo a extremo, enfocándonos primero en los escenarios completos y luego implementando la lógica para verificar que todo funcione.

Preparando la escena

Establezcamos un objetivo razonable. Para verificar que todo está funcionando en nuestra aplicación, debemos cubrir al menos un par de funcionalidades:

1. Cuando los usuarios envíen solicitudes a la aplicación, deben recibir la respuesta correspondiente y, en función de si el intento es correcto o no, obtendrán algunos puntos.

2. La tabla de clasificación debe reflejar correctamente la clasificación de los usuarios.

Ahora, también deberíamos decidir a qué nivel queremos implementar nuestros casos de extremo a extremo. Dado que nuestro sistema expone toda la funcionalidad a través de las API REST, optaremos por*pruebas de servicio de un extremo a otro.* Son mucho más fáciles de mantener que las pruebas de interfaz de usuario de un extremo a otro (que se pueden implementar, por ejemplo, con Selenium[1], ya que dependen de una capa adicional de nuestro sistema y son muy sensibles a los cambios (como dice la documentación de Cucumber, "Validar una regla comercial a través de una interfaz de usuario es lento, y cuando hay una falla, es difícil identificar dónde está el error "[2].

Finalmente, debemos elegir el enfoque y la tecnología que usaremos para nuestros escenarios de un extremo a otro. Usamos Pepino (https://cucumber.io/), una herramienta muy poderosa que se centra en el comportamiento. Escribe las especificaciones de la prueba en lenguaje humano, y serán el guión de la ejecución de la prueba y se utilizarán para generar informes.

Tenga en cuenta que el pepino se puede utilizar para *desarrollo impulsado por el comportamiento* (BDD) también: podríamos implementar los escenarios de extremo a extremo antes que cualquier otra parte de nuestro código y comenzar desde allí, construyendo funcionalidad hasta que los hagamos pasar. No podemos ir a BDD completo en este momento porque ya tenemos nuestro código en su lugar. También habría sido una mala idea que este libro comenzara

[1] http://www.seleniumhq.org/
[2] https://cucumber.io/docs/reference

con estos escenarios de prueba de extremo a extremo, ya que obtendría muchas preguntas en su cabeza desde el principio. Sin embargo, puede intentar seguir BDD en sus proyectos: es una buena manera de asegurarse de que los requisitos sean claros y estén documentados desde una etapa temprana.

En cualquier caso, comenzaremos a codificar nuestros escenarios primero y construiremos todo el resto de la implementación de prueba después, para no perder el enfoque en el negocio.

Es mucho mejor ver Pepino en la práctica para comprender cómo funciona, así que Listado 6-1 muestra un vistazo rápido a uno de los escenarios de prueba.

Listado 6-1. característica de multiplicación (pruebas e2e v9)

```
Característica: los usuarios pueden enviar sus intentos de multiplicación,
    que pueden ser correctos o no. Cuando los usuarios envían un intento
    correcto, obtienen una respuesta que indica que el resultado es el
    correcto. Además, obtienen puntos y potencialmente algunas insignias
    cuando tienen razón, por lo que obtienen la motivación para volver y
    seguir jugando. Las insignias se ganan por el primer intento correcto y
    cuando el usuario obtiene 100, 500 y 999 puntos respectivamente. Si los
    usuarios envían un intento incorrecto, no obtienen ningún punto o
    insignia.

  Escenario: el usuario envía un primer intento correcto y obtiene una insignia
      Cuando el usuario john_snow envía 1 intentos correctos
      Luego, el usuario recibe una respuesta que indica que el intento es
      correcto y el usuario obtiene 10 puntos por el intento.
      Y el usuario obtiene la insignia FIRST_WON

  Escenario: el usuario envía un segundo intento correcto y solo
  obtiene puntos
      Dado el usuario, john_snow envía 1 intentos correctos y el
      usuario obtiene la insignia FIRST_WON
      Cuando el usuario john_snow envía 1 intentos correctos
      Luego, el usuario recibe una respuesta que indica que el intento es correcto.
```

> Y el usuario obtiene 10 puntos por el intento Y el usuario no obtiene ninguna insignia

Perfectamente legible. Es amigable para los humanos y especifica lo que queremos hacer. La mejor parte: esta es la definición de nuestra prueba que será utilizada por código y puede ser ejecutada por Cucumber. Este idioma es pepinillo, y puede encontrar la especificación completa en la página oficial enhttps://tpd.io/ gherkin-doc. También cubriremos los conceptos básicos en pocas palabras en la siguiente sección.

El pepino tiene muchas ventajas, entre otras, el hecho de que los usuarios comerciales pueden leer y modificar los escenarios de prueba directamente y determinar si funcionan. Esa es la gran superpotencia de Pepino, ya que elimina las brechas entre el desarrollo y los requisitos comerciales. Si están escritos en pepinillo, no pueden malinterpretarse: son los ejecutables. Además, estos archivos Gherkin pueden servir como documentación de casos de uso. Sabemos que se mantendrán con seguridad cuando cambien las funcionalidades porque, de lo contrario, esos cambios romperían las pruebas.

Cómo funciona el pepino

Las implementaciones de pepino están disponibles para múltiples lenguajes y marcos. Todos comparten las mismas funcionalidades, que se describen en la página oficial de documentación de referencia enhttps://cucumber.io/ docs / reference.

La idea es que organicemos nuestras funciones en múltiples.característica archivos. En cada uno de ellos incluimos la descripción de la característica en la parte superior. Esa descripción será ignorada por el motor. Cada característica consta de múltiples escenarios, que técnicamente son sus diferentes definiciones de casos de prueba. Finalmente, cada escenario se define mediante varios pasos, utilizando las palabras clave BDD:*Dado, cuando,* y *Luego (*más *Y* y *Pero).*

Cada escenario de una característica (o definición de caso de prueba) se ejecutará dentro de los mismos objetos almacenados en caché. Es muy importante entender este concepto para implementar nuestras pruebas correctamente: podemos compartir el estado entre los pasos del mismo escenario, pero no entre los escenarios. Eso significa que podemos guardar algunos datos en la memoria (es decir, usar algunos campos de clase) para ejecutar varios pasos, incluso si pertenecen a varias clases. A veces, esto puede resultar confuso para los desarrolladores de Java, ya que, en una prueba JUnit, las clases se instancian por método de prueba (a menos que usemos campos estáticos y
@Antes de clase, que normalmente no es una buena idea).

Los pasos se pueden parametrizar utilizando *argumentos,* para que podamos reutilizar la misma definición de paso en varios escenarios con diferentes valores. También podemos pasar una tabla de datos al mismo escenario (cuyas filas se llaman *Ejemplos).* Luego, el escenario se ejecutará una vez por fila de datos.

Usemos este paso para comprender cómo funcionan los argumentos. Ver listado6-2.

Listado 6-2. característica de multiplicación (pruebas e2e v9)

```
Cuando el usuario john_snow envía 1 intentos correctos
```

En principio, el propio Gherkin no sabe cuáles de nuestras palabras son argumentos. Definimos eso a nivel de código. En este caso, nos gustaría pasar a nuestro paso el*alias de usuario,* la *Número de intentos* y si son *derecho* o *equivocado.* Listado 6-3 muestra cómo este paso debe implementarse en Java para admitirlo.

Listado 6-3. MultiplicationFeatureSteps.java (e2e-tests v9)

```
@Given ("^ el usuario ([^ \\ s] +) envía (\\ d +) ([^ \\ s] +) intentos")
público void the_user_sends_attempts (cadena final userAlias,
intentos int finales, cadena final rightOrWrong) lanza Throwable {

    // Implementa la lógica
}
```

Veremos los detalles de la codificación en la siguiente sección, pero como puede ver, es solo una cuestión de configurar el paso con algunas expresiones regulares, que coincidirán con las palabras de la oración. Dos expresiones de palabras y una numérica funcionarán en este caso particular.

Tenga en cuenta que, si cambiamos palabras en la oración (las que no son argumentos), también debemos actualizar la expresión del método. La buena noticia es que la mayoría de los IDE pueden integrarse con Cucumber a través de complementos, que nos advertirán si las oraciones no tienen un patrón de coincidencia válido en el código.

Los resultados de la ejecución de nuestras funciones, con detalles sobre escenarios y pasos, se pueden generar en múltiples formatos para informes. Entre ellos se incluyen los específicos de Cucumber (pepinillo de color), pero también los informes JUnit estándar, que pueden ser utilizados por marcos de integración continua.

Eso es todo en pocas palabras. Vayamos ahora a la práctica y escribamos un código para que funcione en nuestra aplicación.

Código práctico

Crearemos dos características, como se definen al principio de este capítulo: la primera se enfoca en probar las interacciones a través de intentos y la segunda verifica las funcionalidades de la tabla de clasificación.

CÓDIGO FUENTE DISPONIBLE CON EL LIBRO: V9

puede encontrar la nueva versión del código con el nuevo proyecto de pruebas de un extremo a otro (el tests_e2e carpeta) y las modificaciones para hacer que el sistema *comprobable* en el v9 repositorio en github: https://github.com/microservicespractical.

Crear un proyecto vacío y elegir las herramientas

Como queremos interactuar con el sistema como un agente externo, crearemos un nuevo proyecto con el código necesario para hacerlo. En este caso, no estamos creando un nuevo microservicio para que podamos cambiar el conjunto de herramientas a utilizar.

Para mantenerlo simple, comenzaremos con código simple de Java 8 y algunas bibliotecas y marcos para lograr una estrategia de prueba sólida de extremo a extremo:

- *Pepino:* Más específicamente *pepino-jvm,* la implementación de Java de esta herramienta.
- *JUnit de pepino:* Nos dará la integración con JUnit.
- *Picocontenedor de pepino:* Lo usaremos para el tabla de clasificación función, para utilizar la inyección de dependencia en nuestras pruebas.
- *JUnit:* Agregamos esta dependencia para obtener el soporte principal para las pruebas en Java.
- *AssertJ:* Proporciona una forma natural de hacer afirmaciones.
- *Apache Fluent HttpClient:* Lo usaremos para conectarnos a la API REST de la aplicación.
- *Jackson 2:* Nos permite deserializar JSON sin demasiado esfuerzo.

Como puede ver, no necesitamos Spring Boot para esto. Una forma sencilla de crear un proyecto de Maven vacío es utilizar uno de los arquetipos. Podemos usar una versión instalada de Maven o también podemos copiar el contenedor de uno de nuestros proyectos anteriores (.mvn carpeta y ejecutable) en una carpeta vacía que nombraremos tests_e2e. Luego ejecute lo siguiente:

```
. / mvnw arquetipo: generar -DgroupId = microservices.book
- DartifactId = e2e-tests-v9 -DarchetypeArtifactId =
mavenarchetype-quickstart -DinteractiveMode = false
```

El siguiente paso es abrir nuestro casi vacío pom.xml e incluir las dependencias enumeradas. Necesitamos tres artefactos Maven diferentes para usar Jackson 2. En el ejemplo de Listing6-4, extrajimos algunas versiones como propiedades para manejar mejor las actualizaciones desde un lugar común.

Listado 6-4. pom.xml (e2e-tests v9)

```
<proyecto xmlns = "http://maven.apache.org/POM/4.0.0" xmlns:
xsi = "http://www.w3.org/2001/XMLSchema-instance"
          xsi: schemaLocation = "http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0 </modelVersion>
    <groupId>microservices.book </groupId>
    <artifactId>pruebas-e2e-v9 </artifactId>
    <paquete>jar </embalaje> <versión>0.9.0-
    INSTANTÁNEA </versión> <build>

        <plugins>
            <enchufe>
                <groupId>org.apache.maven.plugins </groupId>
                <artifactId>complemento-compilador-maven </
                artifactId> <configuración>
                    <fuente>1,8 </fuente>
                    <objetivo>1,8 </objetivo>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <nombre>pruebas-e2e-v9 </nombre> <descripción>Pruebas de
    extremo a extremo - Microservicios - La forma práctica (libro) </
    descripción>
```

```
<propiedades>
    <project.build.sourceEncoding>UTF-8 </proyecto.build.
    sourceEncoding>
    <project.reporting.outputEncoding>UTF-8 </proyecto.
    reporting.outputEncoding>
    <versión java>1,8 </java.version> <jackson-2-
  version>2.8.9 </jackson-2-version> <versión de
  pepino>1.2.5 </pepino-version> </properties>


<url>http://maven.apache.org </url>
<dependencias>
    <dependencia>
        <groupId>info.cukes </groupId> <artifactId>
        pepino-java </artifactId> <versión> $ {pepino-
        version} </versión> <alcance>prueba </
        alcance>
    </dependencia>
    <dependencia>
        <groupId>info.cukes </groupId> <artifactId>
        pepino-junit </artifactId> <versión> $ {pepino-
        version} </versión> <alcance>prueba </
        alcance>
    </dependencia>
    <dependencia>
        <groupId>info.cukes </groupId> <artifactId>pepino-
        picocontenedor </artifactId> <versión> $ {pepino-
        version} </versión> <alcance>prueba </alcance>

    </dependencia>
    <dependencia>
        <groupId>org.apache.httpcomponents </groupId>
```

```
        <artifactId>fluida-hc </artifactId>
        <versión>4.5.3 </versión> <alcance>
        prueba </alcance>
</dependencia>
<dependencia>
        <groupId>junit </groupId>
        <artifactId>junit </artifactId>
        <versión>4.12 </versión> <alcance>
        prueba </alcance>
</dependencia>
<dependencia>
        <groupId>org.assertj </groupId>
        <artifactId>asertj-core </artifactId>
        <versión>3.8.0 </versión> <alcance>prueba
        </alcance>
</dependencia>
<! - el núcleo, que incluye Streaming API, compartido de
bajo nivel
  abstracciones (pero NO enlace de datos) ->
<dependencia>
        <groupId>com.fasterxml.jackson.core </groupId>
        <artifactId>jackson-core </artifactId> <versión> $ {
        jackson-2-versión} </versión> <alcance>prueba </
        alcance>
</dependencia>
<dependencia>
        <groupId>com.fasterxml.jackson.core </groupId>
        <artifactId>Jackson-anotaciones </artifactId>
        <versión> $ {jackson-2-versión} </versión> <alcance>
        prueba </alcance>
</dependencia>
```

```
            <dependencia>
                <groupId>com.fasterxml.jackson.core </groupId>
                <artifactId>jackson-databind </artifactId> <versión>
                ${jackson-2-versión} </versión> <alcance>prueba </
                alcance>
            </dependencia>
        </dependencias>
</proyecto>
```

Hacer que el sistema sea comprobable

Uno de los principales problemas de algunos sistemas es que no son suficientemente comprobables. La raíz del problema puede ser diferente en cada caso, pero uno común está relacionado con un concepto erróneo de la*prueba de caja negra* enfoque, que establece que simplemente interactuamos con el sistema desde el exterior y usamos *lo que tenemos disponible* para comprobar que el comportamiento es el esperado. La implementación del sistema sigue siendo desconocida para el evaluador. La principal ventaja de aplicar pruebas de caja negra para escenarios de extremo a extremo es que podemos verificar que todo el sistema funciona desde el punto de vista del usuario. Sin embargo, no deberíamos llevar esa idea al límite. Si nos encontramos analizando múltiples registros o haciendo scripts complicados para verificar algunas salidas generadas (como archivos, o incluso la base de datos), deberíamos detenernos y pensar si no sería mejor ofrecer más herramientas para verificar el comportamiento del sistema. o incluso proporcionar interfaces específicas de prueba en nuestro sistema. Debemos considerar si lo que tenemos disponible es lo suficientemente bueno.

Edificio *como un hacker* Los scripts de aserción (como los de análisis de registros) son aún más comunes en empresas donde existe una marcada separación entre desarrolladores y probadores. En cualquier caso, no ayuda en absoluto: esos scripts se convertirán en una pesadilla de mantener, especialmente si se basan en salidas que no son consecuencia de requisitos funcionales (como registros).

Pueden cambiar sin previo aviso y romper todo el conjunto de pruebas con un *falso negativo (*ya que la prueba fallaría, pero la lógica aún funciona).

Sumergiéndonos en nuestro sistema como ejemplo, resulta que no tenemos ninguna API para obtener la puntuación resultante para un identificador de intento. Lo que tenemos es una salida de línea en el registro de gamificación,El usuario con id {} obtuvo {} puntos por intento de id {}. Pero, como se explicó, si usamos esa línea en nuestras afirmaciones de prueba, podríamos terminar con fallas de prueba si esa línea cambia o se elimina, lo que puede suceder fácilmente ya que no es un requisito funcional.

Como alternativa, podemos considerar incluir la API REST para obtener la puntuación de un intento determinado. Podríamos argumentar que no existe un requisito funcional para incluir eso en nuestra API y, por lo tanto, nunca debería estar allí. Sin embargo, ciertamente tiene sentido tenerlo ahí: no está exponiendo ninguna lógica, podría ser útil para la funcionalidad en el futuro, y es simplemente un requisito de prueba ahora. Este es el enfoque que seguiremos: para evitar complicar demasiado las cosas, crearemos algo de soporte en nuestra aplicación para brindar un mejor soporte a las pruebas.

En los siguientes párrafos, explicaremos cómo debemos adaptar nuestra aplicación para que sea comprobable: incluir nuevas interfaces API, usar perfiles de prueba y cuidar bien los datos de prueba por separado.

Nuevas interfaces API

Como se mencionó, necesitamos obtener los datos de puntuación por su identificador de intento. Dado que esta es una función que podría ser útil en el futuro, no la incluiremos solo para pruebas, sino como un nuevo punto final disponible. Lo necesitamos para verificar que un intento publicado haya generado puntos para el usuario, por lo que si la identificación del intento es1, podemos llamar OBTENER / puntuaciones / 1 y obtenga la puntuación asociada con ese intento. Conseguir

eso, crearemos un nuevo controlador en el servicio de gamificación y su correspondiente prueba unitaria. Ver listado6-5.

Listado 6-5. ScoreController.java (Nuevo) (Gamification v9)

```
paquete microservices.book.gamification.controller;

importar microservices.book.gamification.domain.ScoreCard;
importar microservices.book.gamification.service.GameService;
importar org.springframework.web.bind.annotation.GetMapping;
importar org.springframework.web.bind.annotation.PathVariable;
importar org.springframework.web.bind.annotation.RequestMapping;
importar org.springframework.web.bind.annotation.RestController;

/**
 * Esta clase implementa una API REST para el usuario de gamificación.
Servicio de estadísticas.
 */
@RestController
@RequestMapping ("/ puntuaciones")
clase ScoreController {

    privado final GameService gameService;

    público ScoreController (final GameService gameService) {
        esto.gameService = gameService;
    }

    @GetMapping ("/ {intentId}")
    público ScoreCard getScoreForAttempt (
            @PathVariable ("intentId") final Long AttempId) {
        regreso gameService.getScoreForAttempt (intentId);
    }
}
```

No podemos olvidar incluir la nueva configuración de enrutamiento en la puerta de enlace, como se muestra en el Listado 6-6.

Listado 6-6. application.yml (Gateway v9)

```
zuul:
  ignoredServices: '*'
  prefijo: / api
  rutas:
    #. . . otras rutas existentes ...
    puntuaciones:
      ruta: / puntuaciones / **
      serviceId: gamification
      strip-prefix: false
```

EJERCICIO

Necesitamos exponer un punto final adicional, esta vez dentro del microservicio de multiplicación: */usuarios / {userId}*. la razón es que, dentro de nuestra segunda característica para probar, la funcionalidad de la tabla de clasificación, queremos mapear los alias de usuario (especificados en el archivo pepinillo) a identificadores de usuario internos.

esto debería ser una tarea fácil para usted en este momento. de todos modos, si necesita ayuda, encontrará elUserController y el UserRepository clases en el v9 repositorio en github.

Perfiles de prueba

Spring Boot tiene perfiles, que son diferentes configuraciones que podemos cargar para nuestro sistema (mediante la invalidación o extensión de propiedades). Crearemos un prueba perfil para que nuestros microservicios se utilicen en el entorno de prueba. Eso no cambiará ninguna lógica dentro del sistema, pero nos ayudará a verificar las características.

Tenemos dos objetivos de dominio con la introducción de los perfiles de prueba que cubriremos en las siguientes páginas:

- Queremos manejar datos ficticios que descartaremos después de las pruebas.
- Queremos poder limpiar el sistema y devolverlo a un estado nuevo.

Tenga en cuenta que para nuestra estrategia de prueba de un extremo a otro, no podemos beneficiarnos de algunas funciones de prueba de Spring, como la reversión de transacciones al final de nuestras pruebas. Esas características son realmente poderosas para todo tipo de pruebas que se ejecutan dentro del alcance de una sola aplicación Spring (incluidas las pruebas de integración dentro de un microservicio). Pero no son útiles aquí por varias razones:

- Las pruebas se ejecutan desde un proyecto diferente (debido al enfoque de prueba de caja negra) que interactúa con otras aplicaciones a través de llamadas RESEST. Nuestros microservicios ignorarán las anotaciones de prueba y, en general, cualquier uso que hagamos en el proyecto de prueba de las capacidades de Spring Test.
- Al seguir un enfoque de extremo a extremo, normalmente también nos gusta probar cómo funciona la infraestructura. Eso incluye transacciones reales a la base de datos y mensajes que fluyen a través de RabbitMQ.
- Incluso si creáramos un perfil en nuestros microservicios que imita el comportamiento de reversión de transacciones, tenga en cuenta que nuestro proceso no es una sola transacción. La *intento de apuntar* El proceso es el resultado de la multiplicación almacenando el intento en la base de datos y enviando un evento (una transacción), más la gamificación consumiendo el evento y calculando la nueva puntuación (otra transacción). Si revertimos el primero después de que termine, el segundo fallará.

Por lo tanto, para nuestro enfoque de extremo a extremo, debemos ocuparnos de los datos ficticios y limpiarlos nosotros mismos. Veamos cómo.

Manejo de datos de prueba

Pondremos algunos datos de prueba en nuestro sistema para demostrar que todo funciona, pero no queremos esos datos en nuestro entorno de producción. Para evitarlo, crearemos una base de datos diferente con fines de prueba. Necesitamos anular la propiedad de la URL de la base de datos para darle un nombre diferente, tanto para los servicios de gamificación como para los de multiplicación. Como se explicó, lo haremos en un perfil de Spring diferente (prueba), que para un archivo de propiedades se puede definir simplemente mediante la convención de nomenclatura (nuestro -prueba sufijo). Ver listado6-7.

Listado 6-7. application-test.properties (Gateway v9)

```
spring.datasource.url = jdbc: h2: file: ~ / gamification-test; DB_
CLOSE_ON_EXIT = FALSE; AUTO_SERVER = TRUE
```

Tenga en cuenta que no necesitamos incluir ninguna otra propiedad en ese perfil, ya que se cargarán desde el plano application.properties expediente. Spring maneja estos archivos con herencia. Si iniciamos nuestra aplicación con el prueba perfil activo, cargamos en el contexto de nuestra aplicación todas las propiedades del archivo de propiedades principal (el que no tiene sufijo) más las del perfil específico (que anulará los valores principales si tienen la misma clave).

Además de separar los datos, también queremos comenzar cada caso de prueba con una base de datos limpia, por lo que nos aseguramos de que los datos existentes (provenientes de pruebas ejecutadas previamente) no afecten el resultado de la ejecución de nuestra prueba. Hay muchas formas de lograr esto, pero una simple desde el punto de vista del mantenimiento (y alineada con una visión de DevOps) es integrar esa funcionalidad de limpieza en nuestra aplicación. Dado que el servicio posee la lógica para crear entidades, no hay nadie mejor que un desarrollador para escribir y mantener un punto final REST para eliminar los datos y realizar una nueva inicialización de un servicio. No se preocupe por el riesgo potencial de hacer esto: expondremos esta funcionalidad solo mientras se ejecuta en modo de prueba.

Creamos un AdminController tanto en los microservicios de gamificación como de multiplicación. Para restringirlo correctamente, incluimos el @Perfil anotación, que le dirá a Spring que cargue ese bean solo cuando el perfil esté prueba. Ver listado 6-8.

Listado 6-8. AdminController.java (Gamificación v9)

```
@Profile ("prueba")
@RestController
@RequestMapping ("/ gamification / admin")
clase AdminController {

    privado final AdminService adminService;

    público AdminController (AdminService final adminService) {
        esto.adminService = adminService;
    }

    @PostMapping ("/ delete-db")
    público ResponseEntity deleteDatabase () {
        adminService.deleteDatabaseContents ();
        regreso ResponseEntity.ok (). Build ();
    }
}
```

Ahora, cada vez que hacemos un POST / [servicio] / admin / delete-db, limpiará la base de datos. Dado que esta es una funcionalidad que queremos asegurarnos de que esté correctamente oculta, también verificaremos con pruebas unitarias que sea accesible solo cuando el perfilprueba Está establecido. Lo logramos usando en nuestras pruebas la anotación @ActiveProfiles y asegurándose de que, en el caso de que no se establezca ningún perfil, el punto final devuelva un EXTRAVIADO estado. Implementemos un par de pruebas dentro de la multiplicación y la gamificación para verificar esto: AdminControllerEnabledTest y AdminControllerDisabledTest—Ver listados 6-9 y 6-10.

Listado 6-9. AdminControllerEnabledTest.java (Gamification v9)

```
@RunWith (SpringRunner.class)
@ActiveProfiles (profiles = "prueba")
@WebMvcTest (AdminController.class)
clase pública AdminControllerEnabledTest {

    @MockBean
    privado AdminService adminService;

    @Autowired
    privado MockMvc mvc;

    / **
     * Esta prueba verifica que el controlador esté funcionando como se
    esperaba cuando
     * el perfil está configurado para probar (ver anotación en la
    declaración de clase)
     * @throws Exception si ocurre algún error
     * /
    @Prueba
    público vacío deleteDatabaseTest () lanza Excepción {
        // Cuándo
        MockHttpServletResponse respuesta = mvc.perform (
                publicación ("/ gamification / admin / delete-db")
                        . aceptar (MediaType.APPLICATION_JSON))
                . andReturn (). getResponse ();

        // luego
        afirmarQue (respuesta.getStatus ()). isEqualTo (HttpStatus.
        OK.value ());
        verificar (adminService) .deleteDatabaseContents ();
    }
}
```

Listado 6-10. AdminControllerDisabledTest.java (Gamificación v9)

```
@RunWith (SpringRunner.class)
@WebMvcTest (AdminController.class)
clase pública AdminControllerDisabledTest {

    @MockBean
    privado AdminService adminService;

    @Autowired
    privado MockMvc mvc;

    / **
     * Esta prueba verifica que el controlador NO ES ACCESIBLE
     * cuando el perfil no está configurado para probar
     *
     * @throws Exception si ocurre algún error
     * /
    @Prueba
    público vacío deleteDatabaseTest () lanza Excepción {
        // Cuándo
        MockHttpServletResponse respuesta = mvc.perform (
                publicación ("/ gamification / admin / delete-db")
                        . aceptar (MediaType.APPLICATION_JSON))
                . andReturn (). getResponse ();

        // luego
        afirmarQue (respuesta.getStatus ()). isEqualTo (HttpStatus.
        NOT_FOUND.value ());
        verifyZeroInteractions (adminService);
    }
}
```

Por último, para alinear eso con nuestra estrategia de enrutamiento en Zuul, creamos un prueba profile también en el servicio de puerta de enlace y cargue la configuración de enrutamiento adecuada solo para el modo de prueba. Tenga en cuenta que para un archivo de propiedades YAML, tenemos una forma adicional de crear un perfil, agregando tres guiones y el nombre del perfil (aunque también puede crear unapplication-test.yml si tu prefieres). Ver listado6-11.

Listado 6-11. application.ym: Agregar un perfil (Gateway v9)

```
#. . . (nuestro contenido application.yml anterior)
- - -
# Agrega rutas de administración con fines de prueba
primavera:
  perfiles: prueba
zuul:
  rutas:
    gamificación-admin:
      ruta: / gamification / admin /
      ** serviceId: gamification strip-
      prefix: false
    multiplicación-admin:
      ruta: / multiplication / admin / **
      serviceId: multiplication strip-
      prefix: false
```

Escribiendo la primera prueba del pepino

Una vez que hayamos preparado nuestro sistema para las pruebas, volvamos a centrarnos en nuestro proyecto de principio a fin, que dejamos vacío. El primer paso es crear una característica en un archivo Gherkin dentrosrc / test / resources, con algunos escenarios. Lo nombraremos característica de multiplicación. Si tiene un IDE que tiene un complemento disponible para Cucumber (como IntelliJ), es hora de instalarlo para que pueda beneficiarse

desde la coloración de la sintaxis, las advertencias del compilador cuando los pasos en las características no coinciden con las expresiones en el código, etc.

La primera característica que queremos crear es la funcionalidad principal proporcionada por nuestra aplicación: los usuarios están enviando sus intentos y recibiendo respuestas. Si el intento es correcto, reciben puntos y, en algunos casos, insignias. Escribamos la descripción completa en nuestra función y definamos nuestros diferentes escenarios y pasos. Ver listado6-12.

Listado 6-12. Función de multiplicación (tests-e2e v9)

```
Característica: los usuarios pueden enviar sus intentos de multiplicación,
    que pueden ser correctos o no. Cuando los usuarios envían un intento
    correcto, obtienen una respuesta que indica que el resultado es el
    correcto. Además, obtienen puntos y potencialmente algunas insignias
    cuando tienen razón, por lo que obtienen la motivación para volver y
    seguir jugando. Las insignias se ganan por el primer intento correcto y
    cuando el usuario obtiene 100, 500 y 999 puntos respectivamente. Si los
    usuarios envían un intento incorrecto, no obtienen ningún punto o
    insignia.

  Escenario: el usuario envía un primer intento correcto y obtiene una insignia
      Cuando el usuario john_snow envía 1 intentos correctos
      Luego, el usuario recibe una respuesta que indica que el intento es
      correcto y el usuario obtiene 10 puntos por el intento.
      Y el usuario obtiene la insignia FIRST_WON

  Escenario: El usuario envía un segundo intento correcto y solo obtiene puntos
      Dado que el usuario, john_snow envía 1 intentos correctos
      Y el usuario obtiene la insignia FIRST_WON
      Cuando el usuario john_snow envía 1 intentos correctos
      Luego, el usuario recibe una respuesta que indica que el intento es
      correcto y el usuario obtiene 10 puntos por el intento.
      Y el usuario no recibe ninguna insignia.
```

```
Escenario: el usuario envía un intento incorrecto y no obtiene nada
  Cuando el usuario john_snow envía 1 intentos incorrectos
  Luego, el usuario recibe una respuesta que indica que el intento es
  incorrecto y el usuario obtiene 0 puntos por el intento.
  Y el usuario no recibe ninguna insignia.

# Comprueba las insignias de bronce, plata y oro.
Esquema del escenario: el usuario envía un intento correcto después de
<previous_attempts> intentos correctos y luego obtiene una insignia
<badge_name>
  Dado el usuario, john_snow envía <previous_attempts> intentos
  correctos
  Cuando el usuario john_snow envía 1 intentos correctos
  Luego, el usuario recibe una respuesta que indica que el intento es
  correcto.
  Y el usuario obtiene 10 puntos por el intento Y el
  usuario obtiene la insignia <badge_name>

  Ejemplos:
    | previous_attempts | badge_name | 9 |          |
    BRONZE_MULTIPLICATOR |
    | 49 | SILVER_MULTIPLICATOR |
    | 99 | GOLD_MULTIPLICATOR |
```

Como ya vimos, lo bueno de Gherkin es que no necesitamos describir lo que queremos probar con ese archivo; es perfectamente comprensible. Verificamos los casos en los que el usuario envía un intento correcto e incorrecto, y luego los diferentes escenarios de la insignia. Para las insignias basadas en el número de buenos intentos, podemos usar una tabla con ejemplos ya que el esquema del escenario es exactamente el mismo.

Tenga en cuenta que en nuestro archivo Gherkin estamos reutilizando algunos de los pasos y también escribiéndolos de una manera que se pueda analizar fácilmente más adelante. La reformulación de frases es una habilidad que aprendes después de la primera vez que escribes escenarios. No se preocupe por la redacción mientras escribe; solo concéntrese en el contenido, escriba oraciones de la manera más amigable para los humanos que pueda imaginar, y luego haga una segunda ronda identificando frases que se pueden parametrizar y combinar usando argumentos. Por ejemplo, mientras escribía esta función, me encontré escribiendo las oraciones en Listing6-13.

Listado 6-13. Multiplication.feature: Fragmento que se puede mejorar (tests-e2e v9)

```
# Primer ejemplo de una oración que se puede combinar con otra

Y el usuario no obtiene puntos por el intento
# Otro ejemplo
Y el usuario obtiene la insignia vinculada al primer intento.
```

Cuando prestas atención a la definición completa de la característica, observas que la primera se puede generalizar a la común para verificar la puntuación (y establecerla en 0), y la segunda se puede reformular para que sea genérica y podamos usarla. para cualquier insignia. Ver listado6-14.

Listado 6-14. Multiplication.feature: Fragmento mejorado (tests-e2e v9)

```
# Primer ejemplo traducido a una oración existente
Y el usuario obtiene 0 puntos por el intento
# Segundo ejemplo hecho genérico
Y el usuario obtiene la insignia FIRST_WON
```

En un caso de la vida real, esto es parte del proceso de desarrollo de software: el usuario empresarial escribirá un archivo de pepinillo sin prestar atención a la reutilización de los pasos, pero luego el desarrollador puede sugerir modificaciones ligeras para hacerlo más eficiente desde un punto de vista técnico. . Puede argumentar que eso nunca debería suceder y que deberíamos mantener el archivo Gherkin tal como viene, y luego implementar algunas referencias a métodos en segundo plano. Eso también es factible, pero el hecho es que podemos encontrar algunas modificaciones que mantienen nuestras definiciones de características bastante legibles pero al mismo tiempo bien parametrizadas, por lo que reducimos el mantenimiento y ayudamos a escribir oraciones más flexibles (en lugar de un uso puntual y ad hoc). Mantenga las oraciones funcionales y amigables con los humanos, pero lo suficientemente genéricas como para reutilizarlas si es necesario. Ese debería ser tu objetivo.

Vincular una función al código Java

Una vez que tengamos un.característica archivo que describe los casos de uso, escribimos nuestro código Java para procesarlos y realizar las acciones y afirmaciones deseadas. Primero, necesitamos vincular nuestra característica a una clase de prueba en Java. Vamos a crear
Prueba de multiplicación dentro de microservices.book paquete. El pepino proporciona un corredor de prueba (Pepino, que debemos pasar a JUnit's @Corre con anotación) y una @PepinoOpciones anotación que podemos usar para configurar algunos complementos. En este caso, nos gustaría agregar algunos informes adicionales, por lo que lo configuramos allí. Ver listado6-15.

Listado 6-15. MultiplicationFeatureTest.java (tests-e2e v9)

```
paquete microservices.book;

importar cucumber.api.CucumberOptions;
importar pepino.api.junit.Cucumber;
importar org.junit.runner.RunWith;
```

```
/ **
  * @autor moises.macero
  * /
@RunWith (Pepino.class)
@CucumberOptions (plugin = {"pretty", "html: target / cucumber",
"junit: target / junit-report.xml"},
            features = "src / test / resources / multiplication.feature")
clase pública MultiplicationFeatureTest {}
```

Para definir los pasos dentro de esa clase, podríamos empezar desde cero, pero también podemos beneficiarnos de una característica interesante en Cucumber: la autogeneración de código para pasos indefinidos. Desde la carpeta raíz del proyecto, ejecutamos ./prueba mvnw. Eso debería ejecutar nuestras pruebas pero, dado que no hay pasos definidos vinculados a nuestro.característica archivo, el corredor Cucumber generará contenido muy útil al final de la salida de nuestra consola, como se muestra en el Listado 6-16.

Listado 6-16. Salida de la consola: Pasos generados automáticamente (tests-e2e v9)

```
. . .
Puede implementar los pasos que faltan con los siguientes fragmentos:

@When ("^ el usuario john_snow envía (\\ d +) intentos correctos $")
public void the_user_john_snow_sends_right_attempts (int arg1) throws
Throwable {
    // Escriba aquí el código que convierta la frase anterior en
    acciones concretas
    lanzar nueva PendingException ();
}

@Entonces ("^ el usuario obtiene una respuesta que indica que el intento es
correcto $")
```

```
public void the_user_gets_a_response_indicating_the_attempt_is_
right () throws Throwable {
    // Escriba aquí el código que convierta la frase anterior en
    acciones concretas
    lanzar nueva PendingException ();
}

@Entonces ("^ el usuario obtiene (\\ d +) puntos por el intento $")
public void the_user_gets_points_for_the_attempt (int arg1) throws
Throwable {
    // Escriba aquí el código que convierta la frase anterior en
    acciones concretas
    lanzar nueva PendingException ();
}

@Entonces ("^ el usuario obtiene la insignia FIRST_WON $")
public void the_user_gets_the_FIRST_WON_badge () throws
Throwable {
    // Escriba aquí el código que convierta la frase anterior en
    acciones concretas
    lanzar nueva PendingException ();
}

@Given ("^ el usuario john_snow envía (\\ d +) intentos correctos $")
public void the_user_john_snow_sends_right_attempts (int arg1) throws
Throwable {
    // Escriba aquí el código que convierta la frase anterior en
    acciones concretas
    lanzar nueva PendingException ();
}
. . .
```