```
        <intervalo class = "multiplicación-a"> </lapso> x <lapso
        class = "multiplicación-b"> </intervalo> </
        h1>
    <p>
    <formulario id = "intento-formulario">
        <div class = "grupo-formulario">
            <etiqueta for = "resultado-intento"> Resultado? </etiqueta>
            <entrada tipo = "texto" nombre = "intento de resultado" id =
            "intento de resultado" class = "control de formulario">
        </div>
        <div class = "grupo-formulario">
            <etiqueta for = "user-alias"> Su alias: </etiqueta>
            <entrada type = "text" name = "user-alias" id = "user-
            alias" class = "form-control">
        </div>
        <entrada type = "submit" value = "Check" class = "btn
        btn-default">
    </form>
    </p>
    <div class = "mensaje-resultado"> </div>

    <div id = "stats-div" style = "display: none;">
        <h2>Tus estadísticas </h2> <tabla id =
        "estadísticas" class = "tabla">
            <tbody>
            <tr>
                <td class = "info"> ID de usuario: </
              td> <td id = "stats-user-id"> </td> </tr>

            <tr>
                <td class = "info"> Puntuación: </
                td> <td id = "stats-score"> </td>
```

```
                </tr>
                <tr>
                    <td class = "info"> Insignias: </td>
                  <td id = "stats-badges"> </td> </tr>

                </tbody>
            </table>
        </div>

</div>
<div class = "col-md-6">
    <h3>Tabla de clasificación </h3>
    <tabla id = "tabla de clasificación" class = "tabla">
        <tr>
            <th>ID de usuario </th>
            <th>Puntuación </th>
        </tr>
        <tbody id = "leaderboard-body"> </tbody> </
        table>
    <div class = "text-right">
        <botón id = "refresh-leaderboard" class = "btn btn-
      default" type = "submit"> Refresh </botón> </div>


    <div id = "results-div" style = "display: none;">
        <h2>Tus últimos intentos </h2> <tabla
        id = "resultados" class = "tabla">
            <tr>
                <th>ID de intento </th>
                <th>Multiplicación </th>
                <th>Ingresaste </th> <th>
                ¿Correcto? </th>
```

```
                        </tr>
                        <tbody id = "cuerpo de resultados"> </tbody> </
                        table>
                    </div>
                </div>
            </div>
        </div>
        <guión src = "js / bootstrap.min.js"> </script> </
        body>
        </html>
```

Como se prometió, sin esfuerzo significativo: el HTML sigue siendo compacto y legible. Quitamos elstyles.css y ahora confíe en el proporcionado por Bootstrap (que necesitamos descargar de su página y colocar dentro de nuestra css carpeta). Los principales cambios introducidos son:

- Hay nuevos cabeza y cuerpo etiquetas internas para incluir Bootstrap y nuestro nuevo gamification-client.js.
- Se han agregado etiquetas del sistema de cuadrícula para distribuir la página en dos áreas, cada una ocupando la mitad de la pantalla (6/12): es solo un conjunto de divarena filas.
- La formulario tiene algunos cambios de estilo para que se vea mejor que antes.
- La sección de la tabla de clasificación es ahora una tabla similar a la tabla de últimos intentos, con una botón para refrescarlo.
- La nueva tabla de estadísticas está ahora en el lado izquierdo y usa el info class en Bootstrap para darle algo de color a la primera columna.
- Tanto las tablas con estadísticas como los últimos intentos permanecen ocultas hasta que se envía un primer intento.

Además de eso, cambiaremos nuestro multiplication-client.js para admitir la nueva funcionalidad. Los principales cambios son:

- Cuando se carga la página, recuperamos los datos no solo para los resultados más recientes (ahora renombrados a updateResults) sino también para la información proveniente de Gamificación: Puntaje e insignias (la nueva tabla de estadísticas) y la tabla de clasificación.

- Tenga en cuenta que introdujimos un retraso (300 milisegundos) para recuperar información del servidor. Esto es para asegurarnos de que le damos algo de tiempo al evento para que se propague y obtengamos la información actualizada. Solo queremos mantener la interfaz de usuario simple, por lo que esta es una solución básica, pero si desea explorar mejores opciones, puede leer sobre cómo el servidor puede notificar al cliente cuando los datos están listos usando tecnologías como WebSockets.

- La updateStats se ha cambiado el nombre de la función a updateResults para evitar confusión. Además, esta función ahora devuelve el identificador de usuario, ya que la nuevaupdateStats para recuperar la información del servidor.

Listado 5-6. multiplication-client.js (gamificación v6)

```
función updateMultiplication () {
    $ .ajax ({
        url: "http: // localhost: 8080 / multiplications / random"}).
    luego (función(datos) {
        // Limpia el formulario
        $ ("# intento-formulario"). buscar ("entrada [nombre = 'resultado-intento']")
        . val ("");
```

```
        $ ("# intento-formulario"). buscar ("entrada [nombre = 'usuario-alias']"
        ). val ("");
        // Obtiene un desafío aleatorio de la API y carga los datos
        en HTML
        $ ('. multiplicación-a'). vacío (). append (data.factorA); $ ('.
        multiplication-b'). empty (). append (data.factorB);
    });
}

función updateResults (alias) {
    var userId = -1;
    $ .ajax ({
        asincrónico: falso,
        url: "http: // localhost: 8080 / results? alias =" + alias, éxito:
        función(datos) {
            $ ('# resultados-div'). show (); $
            ('# cuerpo-resultados'). vacío ();
            data.forEach (función(fila) {
                $ ('# resultados-cuerpo'). append ('<tr> <td>' + row.id +
                '</td>' +
                    '<td>' + fila.multiplicación.factorA + 'x'
                    + row.multiplication.factorB + '</td>' + '<td>'
                    + row.resultAttempt + '</td>' + '<td>' +
                    (row.correct === cierto ? 'SÍ ':' NO ') +' </td>
                    </tr> ');
            });
            userId = data [0] .user.id;
        }
    });
    regreso userId;
}
```

```
$ (documento) .ready (function () {

    updateMultiplication (); $ ("# intento-formulario").

    enviar (función( evento) {

        // No envíe el formulario normalmente
        event.preventDefault ();

        // Obtener algunos valores de los elementos de la página
        var a = $ ('. multiplicación-a'). texto ();
        var b = $ ('. multiplicación-b'). texto ();
        var $formulario = $ ( esto ),
            intento = $ formulario.find ("entrada [nombre =
            'resultantettempt']") .val (),
            userAlias   = $ form.find ("entrada [nombre = 'usuario-alias']")
            . val ();

        // Redacta los datos en el formato que espera la API
        var datos = {usuario: {alias: userAlias}, multiplicación:
        {factorA: a, factorB: b}, resultAttempt: intento};

        // Envía los datos usando post
        $ .ajax ({
            url: 'http: // localhost: 8080 / results',
            escriba: 'POST',
            datos: JSON.stringify (datos),
            contentType: "application / json; charset = utf-8",
            dataType: "json",
            asincrónico: falso,
            éxito: función(resultado){
                Si(result.correct) {
                    $ ('. mensaje-resultado'). vacío ()
```

```
                        . append ("<p class = 'bg-success text-
                        center'> ¡El resultado es correcto!
                        ¡Felicitaciones! </p>");
                } demás {
                    $ ('. mensaje-resultado'). vacío ()
                        . append ("<p class = 'bg-danger text-center'>
                        ¡Vaya, eso no es correcto! ¡Pero sigue
                        intentándolo! </p>");
                }
            }
        });

        updateMultiplication ();

        setTimeout (función(){
            var userId = updateResults (userAlias);
            updateStats (userId);
            updateLeaderBoard ();
        }, 300);
    });
});
```

PREPARACIÓN PARA LA PRODUCCIÓN: DISEÑO WEB RESPONSABLE

gracias a Bootstrap, agregamos una característica clave a nuestra aplicación web: ahora es receptiva. eso significa que se verá bien en pantallas más pequeñas como los teléfonos inteligentes, adaptando el contenido al tamaño de la pantalla. puede probarlo cambiando el tamaño del navegador o usando las herramientas de desarrollo web para simular diferentes dispositivos.

Después de los cambios, podemos ir a nuestra página y ver el cliente web renovado; se ve mucho mejor gracias a Bootstrap (ver Figura5-3). Recuerda que ahora tenemos que incluir un paso extra para ver nuestro sistema funcionando:

1. Inicie el corredor de RabbitMQ.
2. Ejecute el microservicio de Multiplicación (ahora sin IU) desde el IDE, o empaquételo y ejecútelo desde la línea de comandos.
3. Haga lo mismo con el microservicio de gamificación.
4. Ejecute el servidor Jetty desde la carpeta raíz de la interfaz de usuario (ejecutando java -jar [YOUR_JETTY_HOME_FOLDER] / start.jar). Entonces puedes navegar a http: // localhost: 9090 / ui / index.html.

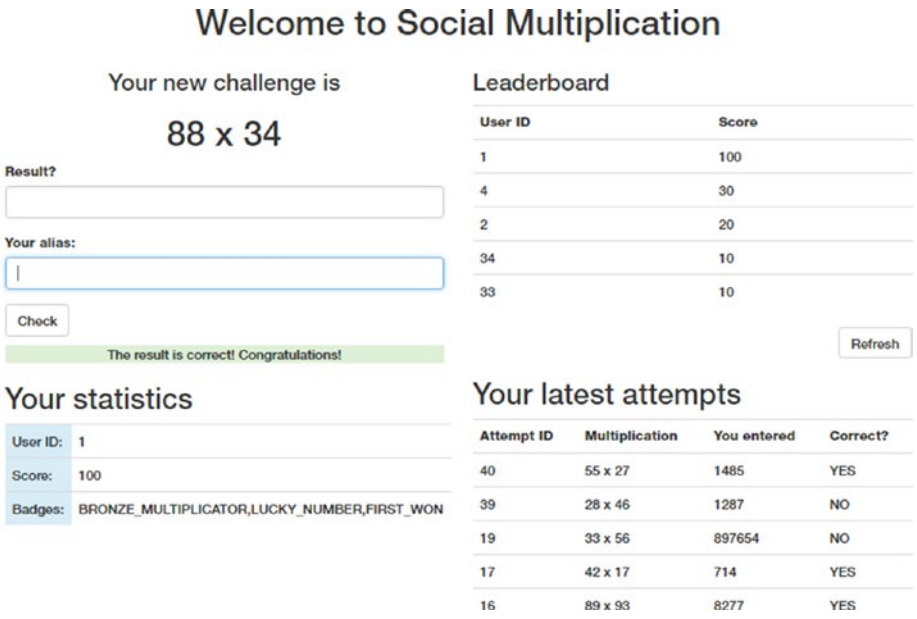

Figura 5-3. *El cliente web renovado*

La arquitectura actual

Revisemos la vista lógica de nuestro sistema, que ahora incluye el servidor de IU como un servicio separado y el navegador, que representa al cliente real que realiza solicitudes a los servicios de backend.

Nuestra arquitectura está creciendo hacia una arquitectura de microservicio real, paso a paso. Eso es genial, porque queremos beneficiarnos de ventajas como tener cambios más independientes y una escalabilidad más flexible.*Pero aún no hemos llegado.*

Como se introdujo cuando conectamos la tabla de clasificación de la interfaz de usuario de la gamificación a nuestros servicios de backend, todavía podemos ver dos problemas importantes con nuestra diseño:

- *Nuestra página de interfaz de usuario aún conoce la estructura del backend:* Necesita saber que hay un microservicio de gamificación y un microservicio de multiplicación. El problema aquí es que, si dividimos o combinamos algunos de nuestros microservicios en el futuro, impactaremos en la interfaz de usuario, lo que requerirá modificaciones para alinearse con la nueva estructura de backend.
- *La interfaz de usuario tiene URL codificadas para localizar microservicios de multiplicación y gamificación.* Lo mismo ocurre con la gamificación y su vínculo con la multiplicación. Deberíamos cambiar esos enlaces directos, de lo contrario, nuestro sistema no escalará.

Aquí es cuando ocurre una transición importante en nuestra arquitectura. Para resolver estos problemas, necesitamos introducir algunos patrones como el descubrimiento de servicios, el balanceo de carga, la puerta de enlace API (o enrutamiento), etc. En el mundo de los microservicios, generalmente vienen junto con los nombres de las herramientas o frameworks que implementan esos patrones:*Eureka, Cónsul, Ribbon, Zuul, etc.*

Como se mencionó anteriormente, es difícil encontrar el camino a través de estas herramientas: ¿cuándo las necesitamos? ¿Deberíamos implementarlos todos para tener una arquitectura de microservicios adecuada? Esas son las preguntas que responderemos en las próximas secciones. Primero, presentaremos el descubrimiento de servicios y el equilibrio de carga. Una vez que comprenda cómo funciona, cubriremos el patrón de puerta de enlace API y veremos cómo funcionan todas las piezas. Luego, como de costumbre, aplicaremos estos patrones a nuestro código y veremos los beneficios para nuestro sistema.

PREPARACIÓN PARA LA PRODUCCIÓN: DESPLIEGUE Y PRUEBAS

es posible que haya notado que el sistema se está volviendo más complejo de administrar. Nuestro estado actual de la arquitectura ya muestra la importancia de la implementación y las pruebas automatizadas si queremos lograr el éxito al implementar una arquitectura de microservicios:

- para iniciar la aplicación completa, necesitamos iniciar manualmente muchas de sus partes (como en la sección anterior). esto es molesto. Podríamos pensar en crear un archivo de secuencia de comandos para iniciar estas diferentes partes, y eso sería realmente bueno ya que crearíamos nuestra primera estrategia de implementación automatizada. No cubriremos esa parte en el libro, pero, como puede imaginar, implementar sus microservicios no es tan fácil como implementar un solo monolito.
- Para alimentar el sistema con datos y comenzar a probarlo, debemos ir a nuestra página y resolver algunos intentos para algunos usuarios diferentes, luego verificar que la gamificación está haciendo su trabajo. Hacer esto manualmente es mucho trabajo y ni siquiera cubrimos todos los diferentes casos de uso. Necesitamos pruebas de un extremo a otro, que se tratan en el próximo capítulo.

Descubrimiento de servicios y equilibrio de carga

Descubrimiento de servicios

Volviendo a la arquitectura actual como referencia, dejamos nuestro microservicio de gamificación contactando la API REST de la multiplicación para recuperar algunos datos. La gamificación sabe dónde encontrar la multiplicación ya que tiene una propiedad que apunta ahttp: // localhost: 8080 /, y póngase en contacto con él para recuperar los factores de multiplicación, como se muestra en la Figura 5-4.

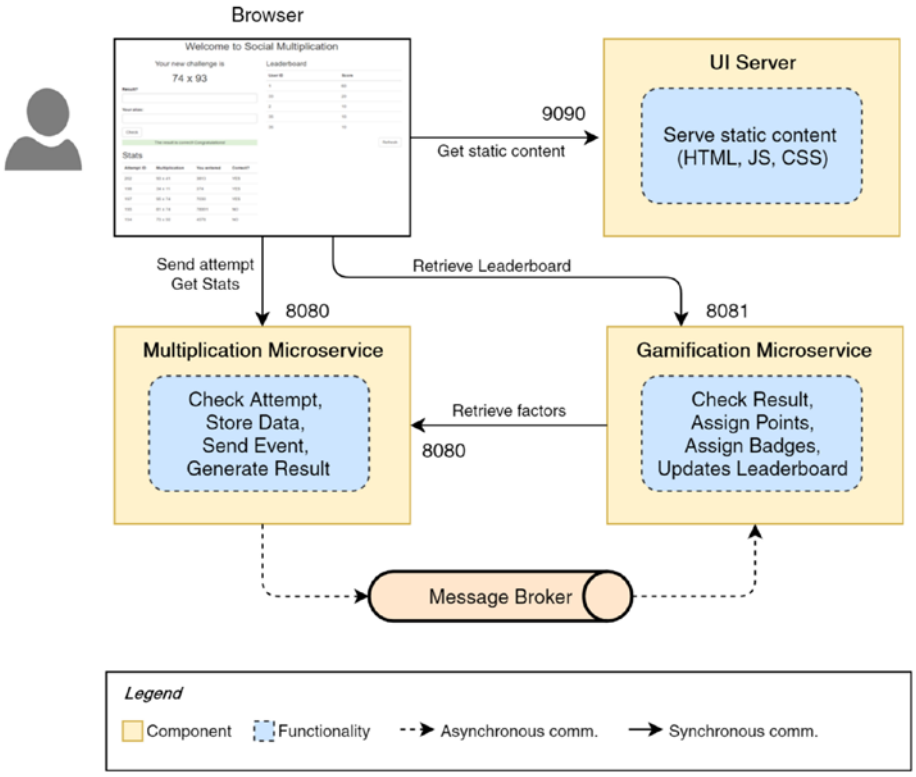

Figura 5-4. *Es casi una arquitectura de microservicio real*

Como se mencionó, ese es un diseño incorrecto. ¿Por qué la gamificación debería conocer *localizacion fisica* (IP y número de puerto) de multiplicación? En un entorno con docenas de servicios que pueden implementarse en todas partes,

ese enfoque es insostenible. No escala en absoluto: ¿qué pasa si introducimos una segunda instancia del microservicio de multiplicación? ¿A cuál debemos invocar desde la gamificación?

Tenga en cuenta que este es un problema similar al que mencionamos al explicar cómo la interfaz de usuario no debería conocer nuestra arquitectura de microservicios. La diferencia es que, en este caso, la comunicación es entre dos microservicios.

Una herramienta de Service Discovery nos dará la solución que estamos buscando. Este tipo de herramientas constan de varias piezas:

- El Registro de servicios, que realiza un seguimiento de todas las instancias de servicio y sus nombres.
- El Agente de Registro, que todo servicio debe utilizar para proporcionar su configuración para que otros puedan encontrarlo.
- El Service Discovery Client, que se pone en contacto con el registro para solicitar un servicio utilizando su alias.

Existen diferentes herramientas de descubrimiento de servicios, entre otras Consul y Eureka, que son muy compatibles con Spring. Usaremos Eureka en este libro, que es parte de la popular pila de OSS de Netflix. Spring proporciona envoltorios para esas herramientas de Netflix dentro del exitoso proyecto Spring Cloud: Spring Cloud Netflix.

Tenga en cuenta que puede seguir instrucciones similares a las de este libro para realizar el siguiente trabajo de configuración con una implementación diferente (por ejemplo, Consul). Lo que es importante entender es el concepto: necesitamos proporcionar un mecanismo a los servicios para que puedan encontrar las instancias de los demás sin vínculos rígidos entre ellos. Como también detallaremos más adelante, detrás de cada referencia entre servicios puede existir una o varias instancias, por lo que el aspecto del equilibrio de carga está estrechamente relacionado con el descubrimiento de servicios.

Ahora que presentamos el concepto y las partes involucradas en él, veamos, en una vista lógica evolucionada, cómo pueden encajar en nuestra arquitectura existente. Figura5-5 no va a ser nuestra solución final, pero es bueno

para echar un vistazo a este estado para comprender mejor cómo funcionará todo nuestro sistema, y luego comprender cuáles son las diferencias y las sinergias entre el descubrimiento de servicios y la puerta de enlace API.

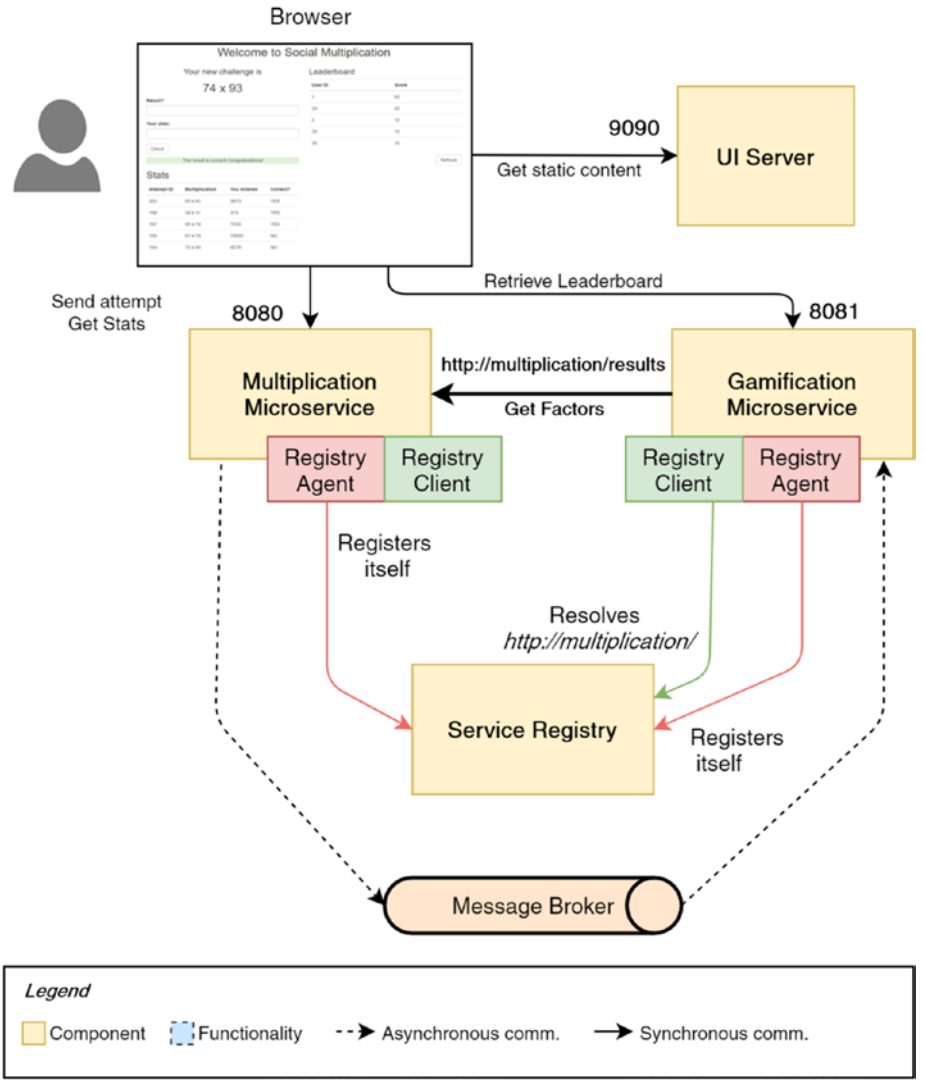

Figura 5-5. *Cómo encaja el descubrimiento de servicios en nuestro sistema*

Podemos ver las tres piezas de descubrimiento de servicios trabajando juntas en la Figura 5-5. Primero, hay un nuevo componente separado:*el Registro de Servicios.* Lo implementaremos como un nuevo microservicio. Los microservicios de multiplicación y gamificación se registrarán poniéndose en contacto con él cuando inicien, utilizando su*Agentes de registro.* En ese momento, obtendrán un alias en el registro, que es el nombre predeterminado del microservicio (lo veremos en la práctica más adelante). Ahora se pueden encontrar usando el http: // multiplicación / o http: // gamificación / direcciones, en lugar de http: // [HOST]: [PUERTO] URLs. Sin embargo, para que esas direcciones funcionen, nuestros microservicios deben usar su*Cliente de registro,* que traducirá los alias a URL específicas utilizando la asignación ubicada en el Registro de servicios. En este escenario, solo entraría en juego el cliente de registro de Gamification, traduciendohttp: // multiplicaciones / dentro http: // localhost: 8080.

Si miras este patrón con una visión nostálgica, ¿no te parece familiar? Es bastante similar a un*DNS Dinámico:* Asignamos un alias a un servicio para que pueda moverse por ubicaciones sin que tengamos que preocuparnos por el lugar (o IP) en particular en el que se implementa el servicio.

Balanceo de carga

Todavía hay una brecha en nuestra arquitectura: ¿cómo funciona Eureka con múltiples instancias del mismo servicio? Los chicos de Netflix también resolvieron ese desafío: implementaron Ribbon para proporcionar*equilibrio de carga del lado del cliente* integrado con Eureka.

Si activamos dos instancias de multiplicationmicroservice, ambas se registrarán en Eureka con el mismo alias (ya que tienen el mismo nombre de aplicación). Digamos que tenemos nuestra nueva instancia ubicada enhttp: // localhost: 8082. Cuando el microservicio de gamificación, como cliente, quiere contactar http: // multiplicación /, Eureka devolverá ambas URL y depende del consumidor decidir qué instancia debe llamarse (utilizando Ribbon, el equilibrador de carga, junto con el cliente de registro de Eureka). De forma predeterminada, Ribbon aplicaría una estrategia simple de Round-Robin, pero veremos en la práctica cómo cambiar eso más adelante. Ver figura5-6.

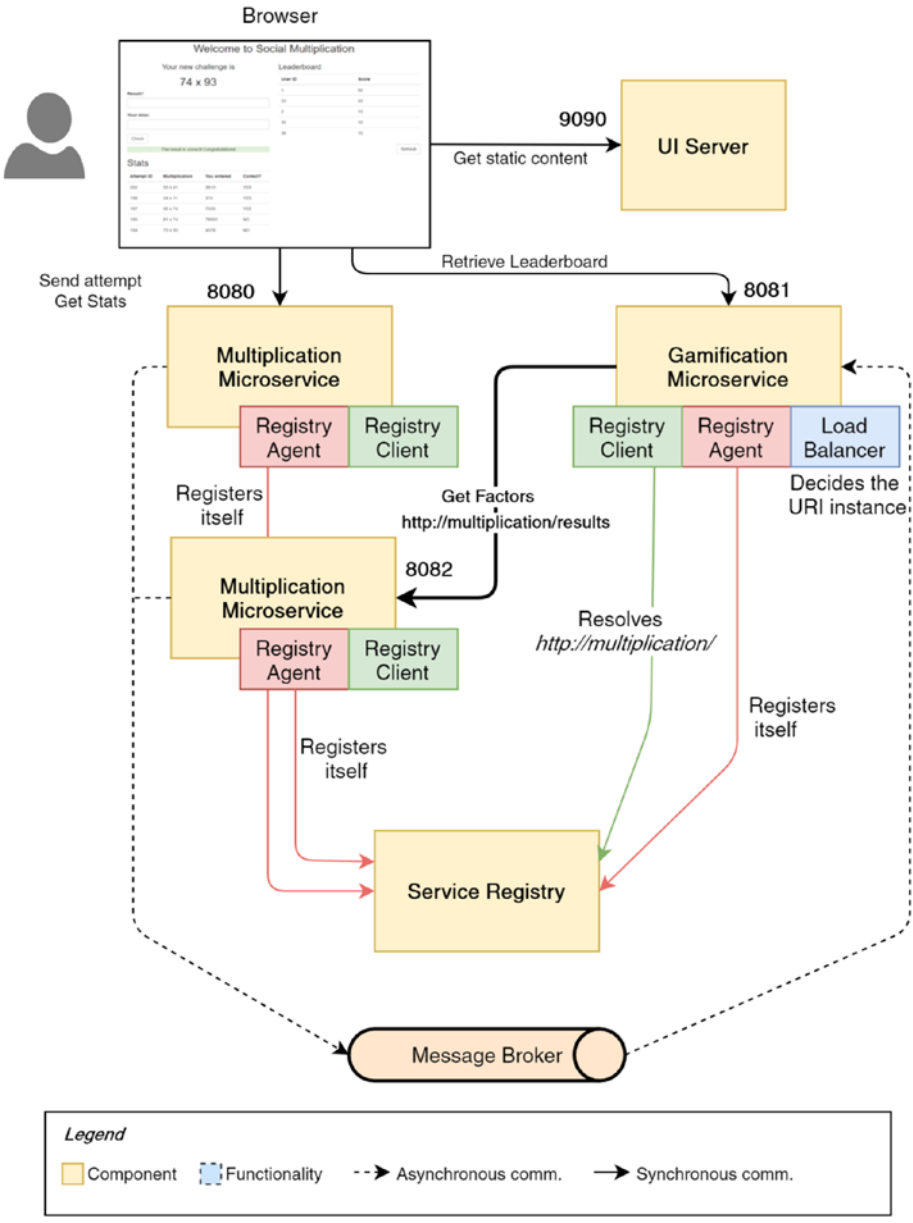

Figura 5-6. *Cómo encaja el equilibrio de carga en nuestro sistema*

Tenga en cuenta que el equilibrio de carga del lado del cliente es un concepto complicado, ya que no parece natural cuando lo lee por primera vez. Es posible que se esté preguntando: ¿por qué la persona que llama debería preocuparse por la parte de equilibrio de carga o la cantidad de instancias de otro servicio? Tienes razón, no debería. Es precisamente por eso que Eureka y Ribbon nos brindan esa funcionalidad de manera transparente, por lo que no necesitamos ocuparnos de eso dentro del código. Llamamos al servicio como si fuera solo una instancia de él. Pero recuerda:*Ribbon solo oculta el equilibrio de carga, pero aún está ahí, en su cliente.*

EL PROBLEMA DE NUESTRA VISIÓN LÓGICA

la última figura nos mostró la interfaz de usuario que se conecta directamente a una de las instancias de nuestro microservicio de multiplicación (en el puerto 8080), lo cual es una mala idea pero es lo mejor que podemos hacer por ahora. el problema es que no podemos integrar el descubrimiento de servicios directamente en la interfaz de usuario, entonces, ¿cómo podemos resolver esto? la siguiente sección sobre enrutamiento y el patrón de puerta de enlace API proporcionará una respuesta a este problema.

Polyglot Systems, Eureka y Ribbon

Hay una pregunta de un millón de dólares para los fanáticos del ambiente políglota en este punto: *¿Qué debemos hacer si uno de nuestros microservicios no está escrito con Spring?* ¿Cómo se supone que incluiremos a Eureka y Ribbon? La respuesta implica, como de costumbre, agregar un jugador extra en nuestro ecosistema de microservicios: el*Sidecar de Spring Cloud (*ver *https://tpd.io/spr-sidecar).* Sidecar es un proyecto inspirado en Netflix Prana y, como su nombre indica, requiere que inicie una instancia de este apéndice de la aplicación por instancia de aplicación que no sea Java para la que desea usar Ribbon y Eureka (que sería la parte de la motocicleta de su sidecar). , Supongo).

No usaremos Sidecar en nuestro sistema, pero puede ver la Figura 5-7 cómo se vería eso en el caso hipotético de tener un microservicio de gamificación escrito en un lenguaje diferente que no sea Java.

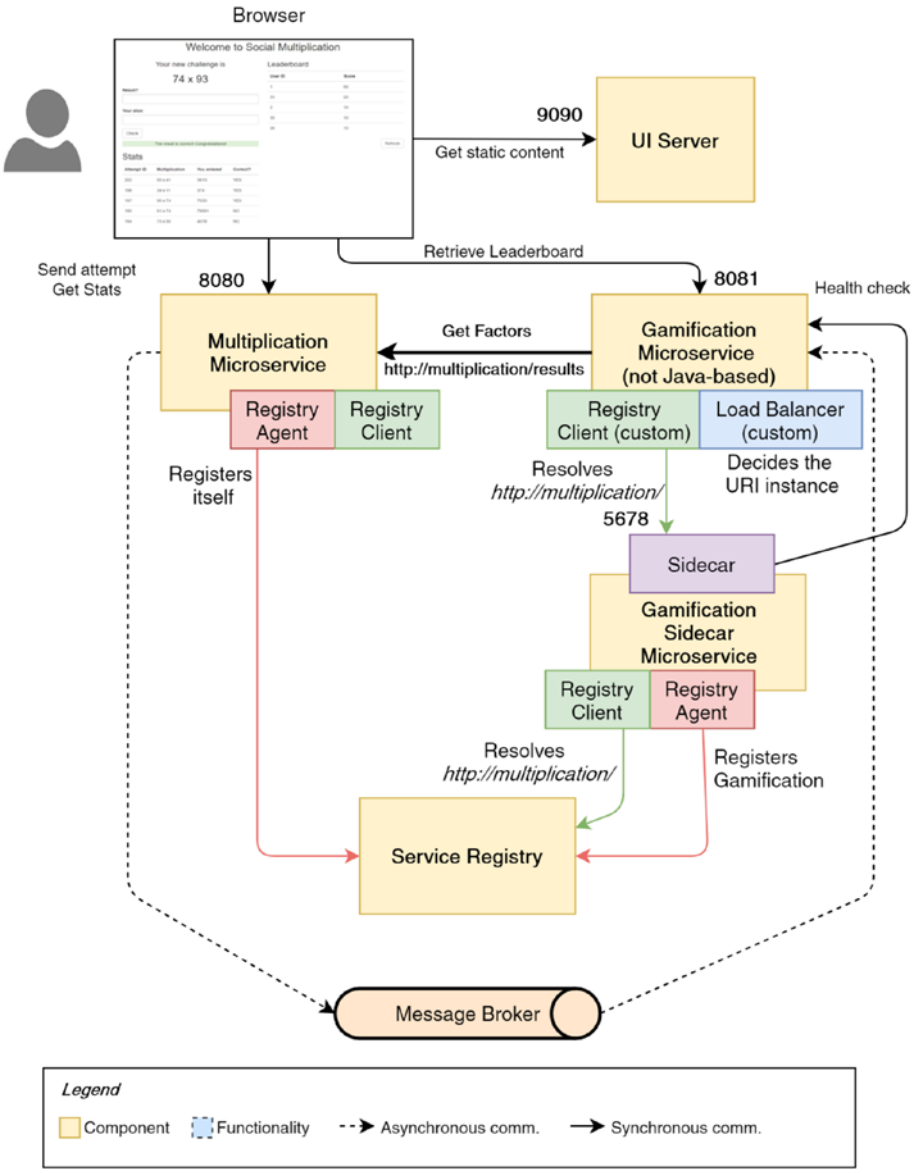

Figura 5-7. *El caso hipotético de tener el microservicio de gamificación escrito en un lenguaje diferente a Java*

Sidecarmicroservice (implementado en el puerto 5678 siguiendo el ejemplo de documentación) actúa como un proxy y requiere que su aplicación exponga un endpoint de salud para que pueda ser descubierto y monitoreado sobre su estado. Utiliza Eureka para registrar (y actualizar) las diferentes instancias de la aplicación que no es de Java (a través del Agente de registro) y para obtener las instancias disponibles de otros microservicios (a través del RegistryClient). La aplicación adjunta a Sidecar puede acceder a las API de Sidecar para encontrar instancias de otros servicios. Tenga en cuenta que depende de la aplicación que no sea Java realizar el equilibrio de carga, ya que obtendrá la lista de todas las instancias disponibles (no una específica) de Sidecar.

Para lograr una alta disponibilidad, los microservicios Sidecar también deben escalarse y monitorearse, lo que introduce una capa adicional requerida de redundancia en su sistema. Como puede imaginar, mantener este enfoque con unos pocos microservicios puede estar bien, pero tenerlo para una gran parte de su sistema podría convertirse en una pesadilla. Depende de nosotros (como miembros del equipo, líderes técnicos o arquitectos) equilibrar este tipo de inconvenientes al diseñar nuestra arquitectura y tomar una buena decisión. En este caso, dos alternativas válidas son buscar la coherencia de la programación del lenguaje en todo su sistema (codificar la mayoría de sus microservicios en Java y Spring Boot) o explorar una estrategia diferente de equilibrio de carga / alta disponibilidad para los microservicios que no son de Java.

Enrutamiento con una puerta de enlace API

El patrón API Gateway

Como acaba de ver, podemos obtener un sistema distribuido compatible con Service Discovery y Load Balancing, lo que nos permite escalar nuestros microservicios sin acoplarnos estrechamente a nuestra infraestructura. Pero hay algunos problemas que aún debemos resolver:

- *Nuestro cliente web se ejecuta en un navegador:* Está claro que no puede ejecutar ningún cliente de descubrimiento de servicios ni ocuparse del equilibrio de carga. Necesitamos una pieza extra para conectarlo

los microservicios que viven en el backend, sin perder las capacidades de equilibrio de carga.

- *Tareas como la autenticación, el control de versiones de API o cualquier filtrado de solicitudes en general, todavía no encajan en nuestro escenario distribuido.* Se requiere un punto de control centralizado para las API de nuestro sistema (puntos finales REST de multiplicación y gamificación).
- *Nuestras API REST siguen la arquitectura del sistema, lo que hace que nuestros consumidores dependan de él.* Este problema es más difícil de ver, así que lo cubriremos con un ejemplo.

Para abordar estos desafíos, implementaremos una puerta de enlace API en nuestro sistema. Siguiendo nuestra dirección actual, elegimos Zuul ya que también es parte de Spring Cloud Netflix y se integra fácilmente con el resto de herramientas incluidas en ese marco. De manera similar al registro de servicios, Zuul funcionará en nuestra arquitectura como un microservicio extra que necesitamos para conectarnos a los demás.

Veamos paso a paso cómo funciona la puerta de enlace API para que podamos entender cómo resuelve nuestros problemas.

Primero, estacionemos el *Problema de equilibrio de carga de la interfaz de usuario* asumiendo que hay uno dedicado por microservicio y analizamos nuestro tercer desafío: hacer que nuestros consumidores conozcan nuestra arquitectura de microservicios. Para comprender mejor por qué esto no es una buena idea, considere el escenario hipotético que se muestra en la Figura5-8: queremos extraer la funcionalidad Estadísticas a un nuevo microservicio, por lo que queremos mover el /estadísticas / punto final allí. Dado que el consumidor (en este caso nuestro cliente web) conoce la estructura de los microservicios, también debe actualizarse para apuntar a la nueva URL (algo así comohttp: // statsmanager / stats). Este molesto efecto secundario se agrava aún más si ofrecemos nuestras API REST a terceros, que deberían adaptar sus aplicaciones a cada trabajo de refactorización de microservicios que realizamos. En lugar de seguir ese camino, queremos crear una API REST que

no revela nuestra estructura interna a nuestros consumidores. Eso nos dará total flexibilidad para luego cambiar partes del mismo sin afectar a los demás. Lo que queremos lograr es tener URL comohttp: // aplicación / líderes y http: // aplicación / resultados (arquitectura-agnóstico), en lugar de tener http: // gamificación / líderes y http: // multiplicación / resultados.

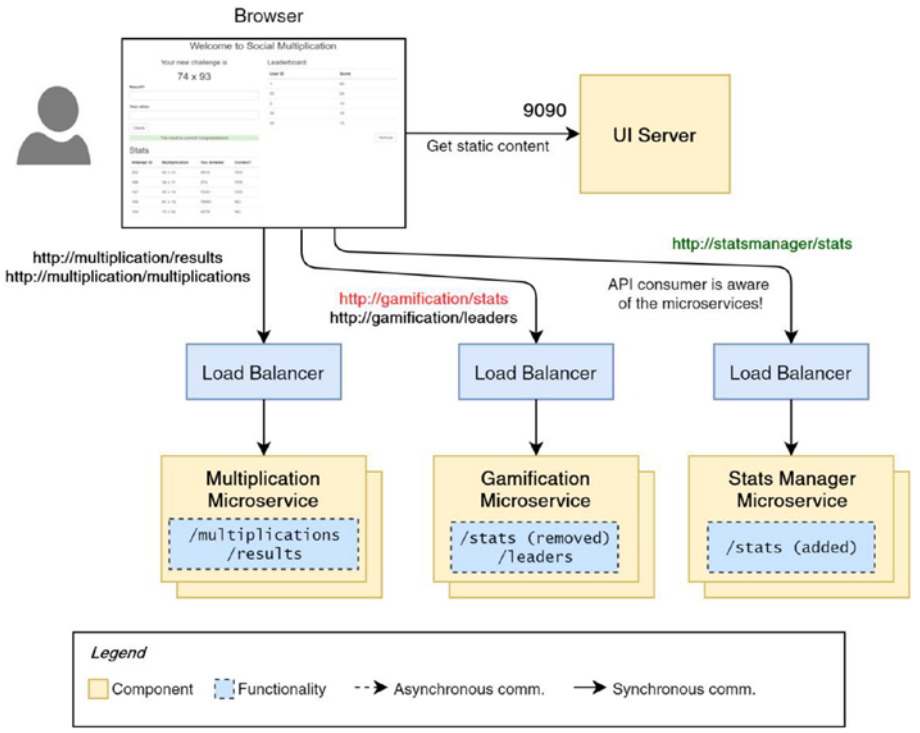

Figura 5-8. *El caso hipotético de dividir / estadísticas en un nuevo microservicio*

El patrón de puerta de enlace API nos dará la solución a este problema. Nuestra herramienta elegida, Zuul, manejará el enrutamiento de las solicitudes al servicio adecuado una vez que configuremos algunos patrones de URL, manteniendo a los consumidores totalmente inconscientes de la estructura interna. Con esa solución en su lugar, tenemos total flexibilidad para mover nuestra funcionalidad alrededor del sistema: solo necesitamos cambiar el*tabla de ruteo,* como se muestra en la Figura 5-9.

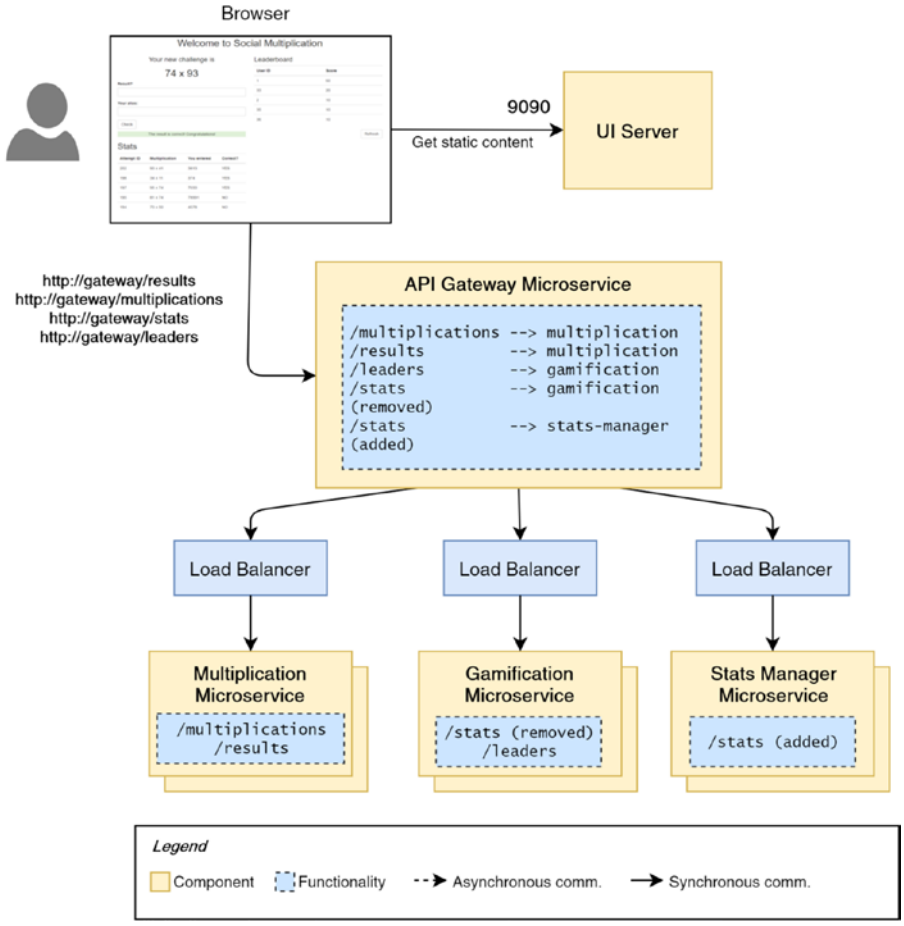

Figura 5-9. *La introducción de una puerta de enlace API y una tabla de enrutamiento*

Tenga en cuenta que la introducción de una puerta de enlace API en nuestro sistema se alinea muy bien con el enfoque de monolito primero. Recuerde que dijimos que es preferible comenzar con grandes trozos de funcionalidad en pequeños monolitos y luego comenzar a dividirlos una vez que los límites del dominio estén claros y el sistema esté evolucionando. Con una puerta de enlace API frente a nuestro sistema, nuestros consumidores de API verán un monolito todo el tiempo, mientras que podemos evolucionarlo de forma transparente para tener un

arquitectura distribuida y escalable detrás de él. Por la misma razón, incluir este patrón en un monolito existente es una forma perfecta de dividirlo y evolucionarlo a una arquitectura de microservicios paso a paso.

Con la puerta de enlace API, también podríamos integrar en un lugar central características como Autenticación, ya que todas las solicitudes pasarán por ese servicio. Podríamos usar Spring Security, por ejemplo, e integrarlo con Zuul con un filtro Zuul personalizado. Puede consultar el Tutorial de patrones de API Gateway en el sitio web de Spring si desea obtener más información sobre cómo hacer que esa solución funcione (consulte*https://tpd.io/apigwsec)*. Como alternativa, también podríamos integrar dentro de nuestro microservicio Zuul un proveedor de autenticación y autorización de terceros, como Auth0 o Okta Single Sign-On.

Tenga en cuenta que estos beneficios del patrón de puerta de enlace de API (abstraer nuestra estructura interna y centralizar el acceso de API a nuestro sistema) no requieren el descubrimiento de servicios ni el equilibrio de carga. También podríamos configurar una tabla de enrutamiento, que apunte directamente a las instancias de microservicio, según algunos patrones de URI en Zuul (así es como comenzaremos a desarrollar nuestro código en la siguiente sección). Sin embargo, ya aprendimos que necesitamos esas capacidades en nuestro sistema para proporcionar alta disponibilidad y poder mantener el sistema en funcionamiento incluso si algunos microservicios no funcionan.

Ahora estamos cerca de entender cómo se combina todo. En figura5-9, simplificamos nuestra vista lógica y colocamos un balanceador de carga frente a cada grupo de instancias de microservicio. Sin embargo, ahora que tenemos un microservicio de puerta de enlace de API que vive en nuestro backend, ¿qué pasa si lo hacemos responsable de la funcionalidad de equilibrio de carga para las solicitudes de frontend? Al hacer eso, resolvemos el problema que aún no hemos abordado: no poder proporcionar equilibrio de carga en el lado del cliente web. En la siguiente subsección, cubriremos la integración de Zuul, Eureka y Ribbon para comprender completamente cómo estas diferentes herramientas (y patrones de arquitectura) funcionan juntas en el mundo de los microservicios.

PREPARACIÓN PARA LA PRODUCCIÓN: SERVICIOS DE VELOCIDAD

Muchas personas a menudo se asustan cuando surgen conversaciones sobre tener partes centralizadas en una arquitectura de microservicios. las partes que son críticas, como el enrutamiento y el filtrado, pueden convertirse en un único punto de falla. este servicio de puerta de enlace de API en particular, como la puerta a nuestros microservicios, también se conoce como un *servicio de borde.*

La clave para hacer que este tipo de servicios funcione en una infraestructura de microservicios es aplicarles un equilibrio de carga adecuado. Para hacer eso, generalmente necesitamos ir un nivel profundo y usar soluciones como un balanceador de carga de infraestructura Dns, en el que, por ejemplo, nuestra puerta de enlace se encuentra enhttp://gateway.ourwebapp.com está respaldado por tres instancias de servidor diferentes. La mayoría de los proveedores de la nube ofrecen estos servicios listos para usar, y también podemos implementarlos nosotros mismos con herramientas como nginx. si desea obtener más información sobre los servicios de borde con un ejemplo práctico de netflix, consulte*https://tpd.io/lb-ms*.

Zuul, Eureka y Ribbon trabajando juntos

Centrémonos ahora en cómo podemos aprovechar nuestro sistema, incluido el descubrimiento de servicios y el equilibrio de carga en nuestra puerta de enlace API. En otras palabras, aprendamos cómo Zuul, Eureka y Ribbon trabajan juntos para brindarnos la solución completa que estamos buscando.

Figura 5-10 representa nuestro sistema con la introducción del microservicio de puerta de enlace API, que se basa en el descubrimiento de servicios y el equilibrio de carga para encontrar otros y permitir que otros lo encuentren.

Mire primero en la parte superior de la figura: el cliente web ya no está conectado directamente a los microservicios sino a la puerta de enlace, enviando todas las solicitudes a través de él (lo haremos disponible en el puerto 8000 dentro de nuestro sistema).

El microservicio de puerta de enlace, que se implementará con Zuul y Spring Boot, contiene una tabla de enrutamiento que apunta a los alias de microservicio registrados en Eureka, en lugar de direcciones físicas. Aquí es donde la integración entre Zuul, Eureka y Ribbon se combina perfectamente para brindarnos una solución completa basada en una puerta de enlace API, descubrimiento de servicios y equilibrio de carga. Cuando Zuul recibe una solicitud, descompone la URL y ubica el patrón en la tabla de enrutamiento. Cada patrón se asigna a un alias de microservicio, por lo que Zuul usa Eureka para ir al registro y encontrar las instancias disponibles. Luego, Ribbon entra en juego y elige una de las instancias según la estrategia de equilibrio de carga (Round-Robin de forma predeterminada). Finalmente, Zuul redirige la solicitud original a la instancia de microservicio correspondiente.

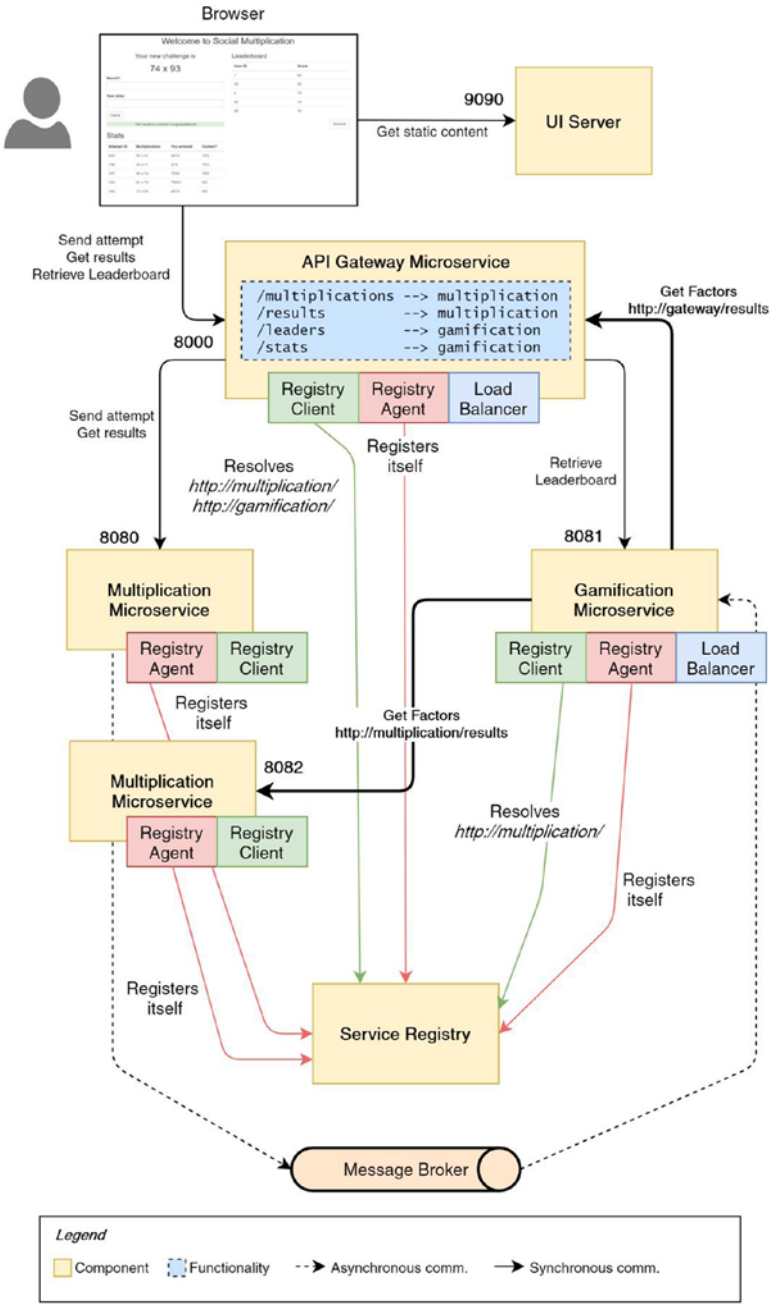

Figura 5-10. *El sistema con la introducción del microservicio de puerta de enlace API*

Ahora, si nos enfocamos en el lado de la figura debajo de la puerta de enlace, podemos ver cómo nuestro microservicio de gamificación tiene dos opciones para recuperar los resultados del microservicio de multiplicación. Puede funcionar como antes y usar su propia conexión al registro de servicios, pero también puede usar el nuevo servicio de puerta de enlace API para lograr su objetivo. Aquí estamos ante uno de los temas más controvertidos en microservicios:*¿Deberían los servicios utilizar la puerta de enlace para llamarse entre sí o deberían seguir utilizando Eureka / Ribbon para la comunicación interna?* No hay una respuesta correcta o incorrecta aquí: depende de su escenario y, más específicamente, de cómo esté configurada su infraestructura. Sin embargo, mantener centralizadas las capacidades de enrutamiento y equilibrio de carga puede traer algunas ventajas:

- Podemos mantener a los microservicios inconscientes de la *ubicación de una funcionalidad dada.* Independientemente de si estamos dividiendo un monolito o decidimos trasladar alguna funcionalidad a un microservicio diferente, podemos hacer que todos los demás microservicios funcionen sin ningún impacto si siempre pasan a través de la puerta de enlace API. Conseguimos un acoplamiento aún más flexible entre nuestros microservicios.
- El equilibrio de carga es con frecuencia un tema crítico en nuestra infraestructura. Configurarlo correctamente es complicado: puede depender de áreas geográficas, latencia de red, carga de microservicios, etc. Normalmente, esas políticas se mantienen mejor centralizadas y eso significa que no debemos depender de la implementación de cada servicio. El equilibrio de carga del lado del cliente no es un buen enfoque para los servicios de backend en estas situaciones, ya que cada servicio tiene un *vista local* de los posibles problemas de infraestructura, pero en su lugar puede ser necesaria una perspectiva global.

Por otro lado, si tomas la decisión de utilizar la puerta de enlace API para procesar todas tus solicitudes dentro del sistema, toma en cuenta que se vuelve aún más crítico. *servicio de borde* y debe respaldarlo con una buena estrategia de redundancia para mantenerlo altamente disponible, tanto interna como externamente.

Para esta aplicación, usaremos el microservicio de puerta de enlace API como el único responsable de enrutar las solicitudes, por lo que mantenemos los microservicios de gamificación y multiplicación inconscientes entre sí. Si traducimos eso a nuestra última cifra, significa que la gamificación y la multiplicación usarán Eureka y Ribbon solo para ubicar la puerta de enlace, que podría replicarse en varias instancias. Dado que esto se está volviendo demasiado teórico, pasemos a la siguiente sección en la que evolucionaremos nuestro sistema paso a paso para hacer realidad esta última visión lógica.

Código práctico

¡Finalmente! Ahora que entendimos los conceptos, podemos aplicar estos patrones a nuestra arquitectura de microservicios e incluir Zuul, Eureka y Ribbon en nuestras aplicaciones Spring Boot. En lugar de hacerlo en un solo paso, presentaremos estas herramientas en dos fases. El primero será Zuul, nuestro servicio de puerta de enlace API.

Implementando API Gateway con Zuul

CÓDIGO FUENTE DISPONIBLE CON EL LIBRO: V7

puede encontrar la nueva versión del código con la puerta de enlace que enruta las solicitudes dentro del v7 repositorio en github: https://github.com/microservicios-practicos.

Para agregar un servicio de puerta de enlace con Zuul, necesitamos crear una nueva aplicación Spring Boot. Podemos navegar de nuevo a Spring Initialzr (http: // inicio. spring.io) y completar nuestros datos, seleccionando Zuul como dependencia que queremos usar. Nuestro nombre de proyecto serápuerta y el nombre del paquete será microservices.book.gateway, como se muestra en la Figura 5-11.

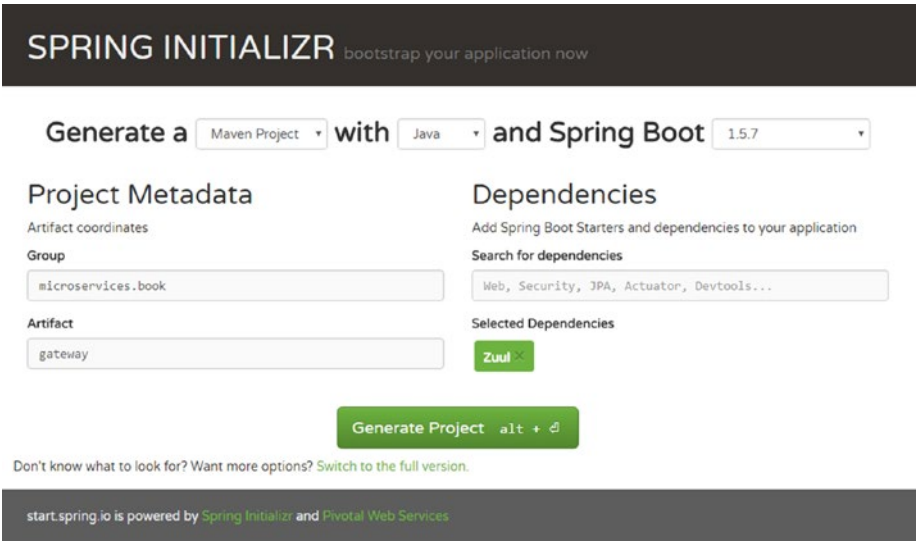

Figura 5-11. *Use Spring Initialzr para crear el microservicio de puerta de enlace*

Luego lo extraemos, lo importamos a nuestro IDE preferido y navegamos directamente a nuestro application.properties archivo (ubicado debajo src / main / resources). Le cambiaremos el nombre a application.yml ya que, en este caso, el formato YAML hará que nuestra configuración sea más legible.

Para que nuestra aplicación Spring Boot se comporte como una puerta de enlace Zuul, solo necesitamos agregar una anotación a nuestra clase principal: @EnableZuulProxy. Ver listado 5-7.

Listado 5-7. GatewayApplication.java (puerta de enlace v7)

```
paquete microservices.book.gateway;

importar org.springframework.boot.SpringApplication;
importar org.springframework.boot.autoconfigure.
        SpringBootApplication;
importar org.springframework.cloud.netflix.zuul.EnableZuulProxy;

@EnableZuulProxy
@SpringBootApplication
clase pública GatewayApplication {

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

Zuul le permite configurar el enrutamiento directamente en el archivo de propiedades, lo que lo hace muy conveniente y sencillo. En este caso, podemos comenzar con esta configuración, como se muestra en Listado5-8.

Listado 5-8. application.yml (puerta de enlace v7)

```
servidor:
  puerto: 8000

zuul:
  prefijo: / api
  rutas:
    multiplicaciones:
      ruta: / multiplicaciones / **
      url: http: // localhost: 8080 / resultados de
      multiplicaciones:
```

```
        ruta: / resultados / **
        url: http: // localhost: 8080 / líderes de
        resultados:
        ruta: / líderes / **
        url: http: // localhost: 8081 / Leaders
        stats:
        ruta: / stats / **
        url: http: // localhost: 8081 / stats

puntos finales:
  rastro:
    sensible: falso

cinta:
  eureka:
    habilitado: falso
```

Veamos lo que estamos configurando dentro de este archivo:

- El puerto del servidor de la puerta de enlace se cambia a 8000. Recuerde: ese será nuestro punto de entrada para todos nuestros consumidores de API REST.
- La última parte es establecer el ribbon.eureka.enabled propiedad a falso ya que decidimos no presentar Eureka y Ribbon todavía para poder desarrollar nuestra aplicación en pequeños incrementos.
- También configuramos Zuul's /rastro punto final como no sensible (no requiere autenticación). Lo usaremos más tarde para ver a Zuul en acción.
- El resto de la configuración (bajo zuul) está ahí para configurar el enrutamiento.

- Establecemos un prefijo para todas nuestras solicitudes. Todas las solicitudes que lleguen deben tener esa parte en la URL, que será eliminada por Zuul al redirigir la solicitud. En nuestro ejemplo, la URL esperada es http: // localhost: 8000 / api / multiplications y será redirigido a http: // localhost: 8080 / multiplicaciones (la /api se elimina el prefijo). Es una forma práctica de agrupar rutas y aplicar diferentes políticas. También podríamos tener dos configuraciones de puerta de enlace para manejar /interno y /público prefijos, por ejemplo.
- Para cada patrón de URL diferente, configuramos el enrutamiento al servicio adecuado (ahora codificamos direcciones físicas). Tenga en cuenta que tenemos dos patrones que apuntan al mismo servicio ya que, por ejemplo, tanto /multiplicaciones y /resultados las entidades están siendo administradas por el microservicio de multiplicación.

El mapeo resultante será el siguiente:

Patrón de solicitud	Objetivo
http: // localhost: 8000 / api / multiplications / **	http: // localhost: 8080 / multiplicaciones / **
http: // localhost: 8000 / api / results / **	http: // localhost: 8080 / resultados / **
http: // localhost: 8000 / api / Leaders / **	http: // localhost: 8081 / líderes / **
http: // localhost: 8000 / api / stats / **	http: // localhost: 8081 / estadísticas / **

Si miramos los patrones de solicitud, notamos cómo logramos lograr nuestro primer objetivo: ahora es posible realizar solicitudes a nuestra aplicación en un lugar central. Los consumidores de API no saben nada sobre nuestros microservicios: todos pasanhttp: // localhost: 8000.

El siguiente paso es incluir un WebConfiguration clase que habilita CORS para el proyecto de puerta de enlace también (como hicimos para los microservicios de multiplicación y gamificación). Esto también es necesario aquí por la misma razón: la interfaz, la puerta de enlace y los microservicios están ubicados en diferentes orígenes (números de puerto en nuestro caso).

Ahora tenemos que vincular todo. Para aplicar los cambios a nuestra interfaz, simplemente modificamos nuestros clientes JavaScript de gamificación y multiplicación para que apunten al servicio de puerta de enlace enhttp: // localhost: 8000 /. Tenga en cuenta que también necesitamos agregar el prefijo api / a todas nuestras peticiones. Asegúrese de utilizar esta nueva variable en todas las llamadas a la API distribuidas en ambos archivos. Ver listado5-9.

Listado 5-9. gamification-client.js / multiplication-client.js Cambio de la URL del servidor (ui v7)

```
var SERVER_URL = "http: // localhost: 8000 / api";
```

En el lado del backend, necesitamos actualizar la gamificación para llamar a la multiplicación usando la puerta de enlace API en lugar de hacerlo directamente. Para hacer eso, necesitamos actualizar la propiedad que incluimos en susolicitud. propiedadesarchivo, como se muestra en el listado 5-10.

Listado 5-10. application.properties Cambio de la URL del servidor (gamification v7)

```
# Configuración del cliente REST
multiplicationHost = http: // localhost: 8000 / api
```

Parece que tenemos todo configurado. Dado que la puerta de enlace es un microservicio en sí mismo, ahora solo necesitamos iniciarlo junto con el resto de nuestros servicios existentes. Resumamos los pasos para hacer que nuestro sistema funcione una vez más:

1. Ejecute el servidor RabbitMQ (si aún no se está ejecutando en segundo plano).
2. Ejecute el microservicio de puerta de enlace.
3. Ejecute el microservicio de multiplicación.
4. Ejecute el microservicio de gamificación.
5. Ejecute el servidor web Jetty desde el ui carpeta raíz.

Recuerde que puede ejecutar microservicios directamente desde su IDE, usando mvnw spring-boot: ejecutar de la carpeta raíz del proyecto o empaquetándolos (paquete mvnw) y usando el java -jar [nombre-artefacto resultante.jar]) comando en la consola. También tenga en cuenta que el orden de esos pasos no es importante aparte del primero, que es necesario para que la multiplicación y la gamificación funcionen correctamente. Puede ejecutar los pasos 2 a 5 en el orden que prefiera.

Si navegamos a http: // localhost: 9090 / ui / index.html, deberíamos ver nuestra aplicación en funcionamiento. Esta vez, cuando publicamos nuevas multiplicaciones y recuperamos la tabla de clasificación, nos contactamos con la puerta de enlace de API en lugar de con las instancias de microservicio.

Podemos verificar cómo funciona Zuul si, después de algunas solicitudes de la interfaz de usuario, navegamos a http: // localhost: 8000 / trace. Allí veremos todas las solicitudes que gestiona Zuul y sus correspondientes respuestas, incluido el tiempo necesario para procesarlas. Si lo hace desde su navegador, la respuesta JSON puede ser difícil de leer, pero puede copiarla / pegarla en un formateador en línea como *https://jsonformatter.org.*

Figura 5-12 representa el estado actual de nuestro sistema, con la puerta de enlace enrutando las solicitudes a los servicios correspondientes. Tenga en cuenta que aún no se ha implementado ningún descubrimiento de servicios, que es lo que haremos a continuación.

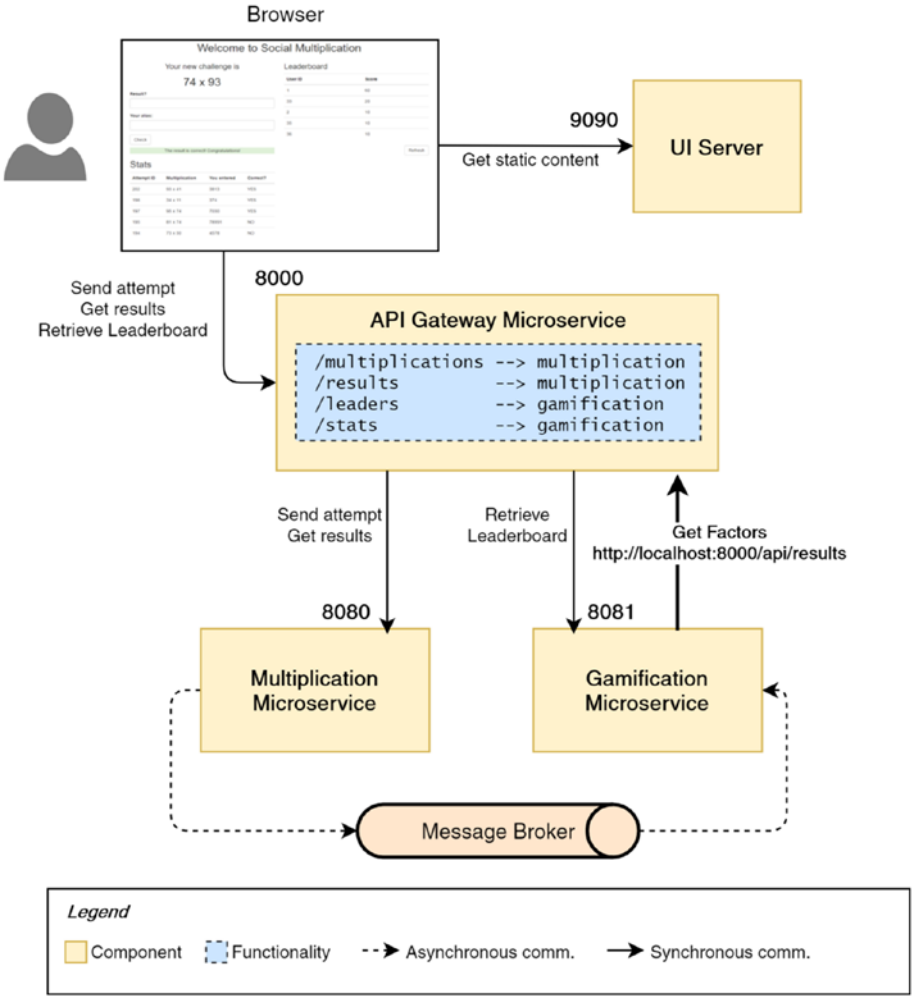

Figura 5-12. *Estado actual del sistema, con la pasarela enrutando las solicitudes a los servicios correspondientes*

Implementar el descubrimiento de servicios

Siguiendo nuestro plan definido durante la primera parte del capítulo, agregaremos descubrimiento de servicios y balanceo de carga a todos nuestros microservicios, incluido nuestro nuevo servicio de puerta de enlace. Haciendo eso, podremos escalar y distribuir de forma segura nuestros servicios, ya que no nos limitaremos a configuraciones específicas de puerto / puerto.

CÓDIGO FUENTE DISPONIBLE CON EL LIBRO: V8

puede encontrar la nueva versión del código con descubrimiento de servicios (eureka) y equilibrio de carga (cinta), junto con el servicio de puerta de enlace de API dentro del v8 repositorio en github: https://github.com/microservices-practical.

Primero, crearemos el registro de servicios. Podríamos hacer eso manualmente, pero para simplificar, usemos Spring Initializr nuevamente, enhttp://start.spring.io.
Esta vez solo necesitamos seleccionar el servidor Eureka como dependencia. Nombra el proyectoservicio-registro y mantenga el nombre del paquete como predeterminado de microservices.book.serviceregistry. Ver figura 5-13.

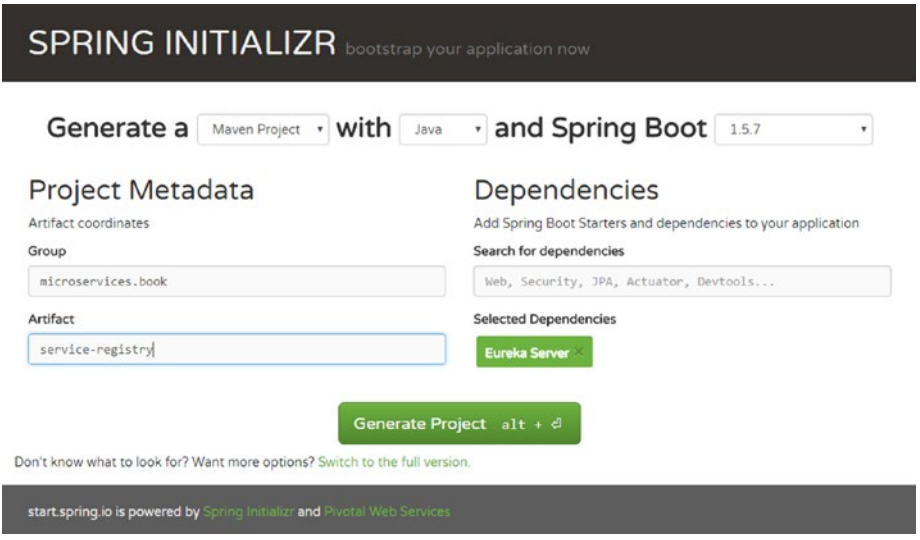

Figura 5-13. *Utilice Spring Initializr para crear el registro de servicios*

Como de costumbre, tomamos el contenido del archivo ZIP descargado y lo colocamos junto con los otros servicios en la estructura de nuestro proyecto y lo importamos a nuestro IDE. El asistente de Spring solo lo está ayudando a crear un proyecto vacío con las dependencias correctas; no está configurando su registro de servicios. Vamos a hacer eso.

Para convertir nuestro servicio en un servidor de registro Eureka, también usamos una anotación (como en el Gateway) en la clase de aplicación: @EnableEurekaServer. Ver listado 5-11.

Listado 5-11. ServiceRegistryApplication.java (registro de servicio v8)

```
paquete microservices.book.serviceregistry;

importar org.springframework.boot.SpringApplication;
importar org.springframework.boot.autoconfigure.
SpringBootApplication;
importar org.springframework.cloud.netflix.eureka.server.
EnableEurekaServer;

@EnableEurekaServer
@SpringBootApplication
clase pública ServiceRegistryApplication {

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

    }
})
```

También necesitamos cambiar el puerto del servidor predeterminado de Spring Boot, 8080, al puerto predeterminado esperado por los clientes de Eureka, que es 8761. Podríamos configurarlo en cualquier otro número de puerto pero, en ese caso, necesitaríamos una configuración adicional en todos nuestros microservicios para anular el puerto predeterminado. Un dato extra curioso es que el registro de Eureka intentará registrarse mediante

también el puerto 8761, así que si no cambia este puerto o deshabilita la función Eureka para registrar la propia instancia (eureka.client.register-witheureka = false), la aplicación fallará cuando la iniciemos porque no se puede encontrar a sí misma. Simplemente cambiaremos el puerto al predeterminado, ya que es una buena práctica que el servidor se registre a sí mismo para que luego se pueda escalar. Ver listado5-12.

Listado 5-12. application.properties (registro de servicio v8)

```
server.port = 8761
```

Eso es todo lo que necesitamos para tener un registro de servicios con Eureka. Ahora podemos ejecutar esto*muy delgado* microservicio como cualquier otra aplicación Spring Boot, pero aún no está conectado a nada más en nuestra arquitectura.

Es el momento de configurar el resto de servicios (social-multiplication, gamification y gateway) para que puedan incluir el cliente Eureka y enviar su información a nuestro nuevo Eureka Server. Para lograr esto, primero debemos agregar la dependencia adecuada a cada serviciopom.xml expediente. Los cambios que introduciremos son:

1. Un bloque de administración de dependencias para resolver las dependencias de Spring Cloud.
2. Una nueva propiedad para hacer referencia a la versión de Spring Cloud.
3. Nuestra nueva dependencia para utilizar el descubrimiento de servicios, primavera-nube-iniciador-eureka.
4. Una dependencia adicional para exponer el estado de nuestros microservicios, resorte-arranque-arrancador-actuador.

Por lo general, solo necesitaríamos incluir las nuevas dependencias (Pasos 3 y 4) pero, en el caso de microservicios de gamificación y multiplicación, la primera y la segunda tarea son necesarias porque es la primera vez que usamos las dependencias de Spring Cloud allí. Echemos un vistazo alpom.xml solicitud de gamificación,

donde puede encontrar las principales diferencias (ver Listado 5-13). Asegúrate de aplicar los mismos cambios a la multiplicación y a la puerta de enlace API, que en este caso ya incluía la configuración de Spring Cloud desde que la creamos con Zuul.

Listado 5-13. pom.xml (gamificación v8)

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

    <groupId>microservices.book </groupId>
    <artifactId>gamificación-v8 </artifactId>
    <versión>0.8.0-INSTANTÁNEA </versión>
    <embalaje>jar </embalaje>

    <nombre>gamificación-v8 </nombre> <descripción>Aplicación
    Social Multiplication - Gamification (Microservices - The Practical
    Way book) </descripción>

    <padre>
        <groupId>org.springframework.boot </groupId> <artifactId>
        Spring-boot-starter-parent </artifactId> <versión>1.5.7
        LIBERACIÓN </versión> <relativaPath /> <! - buscar padre
        desde el repositorio ->
    </padre>

    <propiedades>
        <project.build.sourceEncoding>UTF-8 </proyecto.build.
        sourceEncoding>
```

```
        <project.reporting.outputEncoding>UTF-8 </proyecto.
        reporting.outputEncoding>
        <versión java>1,8 </java.version> <spring-cloud.version>
        Dalston.SR1 </primavera-nube. versión>

    </properties>

    <gestión de dependencias>
        <dependencias>
            <dependencia>)

                <groupId>org.springframework.cloud </groupId>
                <artifactId>Spring-cloud-dependencies </
                artifactId>
                <versión> $ {spring-cloud.version} </versión>
                <tipo>pom </tipo>
                <alcance>importar </
            alcance> </dependencia>
        </dependencias>
    </dependencyManagement>

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

        <dependencia>
      <groupId>org.springframework.boot </groupId>
     <artifactId>Spring-boot-starter-amqp </artifactId> </
     dependency>

        <dependencia>
            <groupId>org.springframework.cloud </groupId>
```

```
            <artifactId>primavera-nube-iniciador-eureka
            </artifactId>
        </dependencia>)

        <dependencia>
            <groupId>org.springframework.boot </groupId>
            <artifactId>actuador de arranque de resorte
            </artifactId>
        </dependencia>

        <! - ... resto de dependencias ->
    </dependencias>

    <build>
        <plugins>
            <enchufe>
                <groupId>org.springframework.boot </groupId>
                <artifactId>Spring-boot-maven-plugin
                </artifactId>
            </plugin>
        </plugins>
    </build>
</proyecto>)
```

La dependencia adicional que agregamos a nuestros servicios es la *Actuador de arranque de resorte.* Una vez que agreguemos esto a una aplicación Spring Boot, automáticamente pondrá a disposición algunos puntos finales que son muy útiles para monitorear: métricas, mapeos, estado, registradores, etc. Nuestro descubrimiento de servicios y configuración del balanceador de carga usará el /salud endpoint para comprobar si el servicio está activo o no. Tenga en cuenta que no enrutaremos estos puntos finales a través de nuestra puerta de enlace API, ya que no queremos que los consumidores externos accedan a ellos (y, obviamente, nuestra infraestructura debería denegar el acceso público directo a nuestros microservicios en producción).

Después de configurar los POM de nuestros clientes de registro, hay algunos cambios más que debemos realizar para cada aplicación Spring Boot que desee utilizar el registro de servicios (consulte Listados 5-14) mediante 5-16:

- Agregamos la @EnableEurekaClient anotación a la clase de aplicación principal, que activará el agente de descubrimiento de servicios.
- Agregamos alguna configuración a nuestro solicitud. propiedades archivo, indicando a Eureka dónde encontrar el registro de servicios.
- Para hacer que nuestro nombre de aplicación sea configurable y no creado automáticamente, incluimos un oreja. propiedades archivo (en la misma carpeta que solicitud. propiedades)en el que configuramos el nombre del servicio. De esa manera nos aseguramos de que los cambios futuros del nombre del proyecto no afecten nuestra infraestructura. Tenga en cuenta que debemos incluirlo en ese nuevo archivo y no en application.properties dado que el registro del servicio ocurre durante el arranque de la aplicación y, durante esa fase, las propiedades de la aplicación aún no se cargan.

Listado 5-14. GamificationApplication.java (gamification v8)

```
paquete microservices.book;

importar org.springframework.boot.SpringApplication;
importar org.springframework.boot.autoconfigure.
SpringBootApplication;
importar org.springframework.cloud.netflix.eureka.
EnableEurekaClient;

@EnableEurekaClient
@SpringBootApplication
```

```
clase pública GamificationApplication {

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

}
```

Listado 5-15. application.properties (gamification v8)

```
# Configuración de Service Discovery eureka.client.service-url.default-
zone = http: //localhost: 8761 / eureka /
```

Listado 5-16. bootstrap.properties (agregado) (gamificación v8)

```
spring.application.name = gamificación
```

Recuerde agregar la anotación y esas dos nuevas propiedades para multiplicar (nombre de la aplicación multiplicación) y puerta de enlace (nombre de la aplicación puerta) también. También agregamos unbootstrap.properties archivo a nuestro nuevo microservicio de registro de servicios para asegurarse de que el nombre de la aplicación sea coherente (ver Listado 5-17).

Listado 5-17. bootstrap.properties (agregado) (registro de servicio v8)

```
spring.application.name = registro de servicio
```

Tres sencillos pasos, pequeños cambios de código e inclusión de algunas dependencias nuevas. Eso es todo lo que necesitamos para que nuestros servicios existentes funcionen con una herramienta de descubrimiento de servicios.

La última tarea que debemos hacer para finalizar nuestra configuración de descubrimiento de servicios para todo el sistema es cambiar la configuración de enrutamiento en nuestro servicio de puerta de enlace API (Zuul). Recuerde que dejamos enlaces directos allí, pero ahora

puede conectar Zuul con el servidor Eureka, ya que la puerta de enlace también utiliza el cliente de descubrimiento para encontrar el registro y asignar nombres de servicio a direcciones específicas. La configuración actual se muestra en el listado5-18.

Listado 5-18. application.yml (puerta de enlace v8)

```
servidor:
  puerto: 8000

zuul:
  ignoredServices: '*'
  prefijo: / api
  rutas:
    multiplicaciones:
      ruta: / multiplications / **
      serviceId: multiplication
      strip-prefix: false
    resultados:
      ruta: / resultados / **
      serviceId: prefijo de la tira de
      multiplicación: falso
    líderes:
      ruta: / líderes / **
      serviceId: gamification
      strip-prefix: false

eureka:
  cliente:
    URL de servicio:
      zona predeterminada: http: // localhost: 8761 / eureka /

puntos finales:
  rutas:
    sensible: falso
```

- La principal diferencia es que cada ruta tiene la serviceId propiedad en lugar de url. Este es el cambio más importante y el que nos da la flexibilidad de tener servicios que pueden cambiar sus ubicaciones de forma dinámica y pueden escalar con múltiples instancias. Ese valor de propiedad debe ser igual al nombre del servicio, que configuramos en cadabootstrap.properties expediente.
- Establecimos prefijo de tira a falso ya que estamos utilizando rutas explícitas, por lo que no es necesario eliminar nada de la especificada camino. Esta propiedad se establece en cierto de forma predeterminada para que las rutas dinámicas eliminen el nombre del servicio de la URL.
- Con el ignoredServices propiedad le decimos a Zuul que no registre dinámicamente rutas para servicios registrados con Eureka. Todavía queremos decidir nuestra ruta por nosotros mismos.
- El cliente Eureka también está configurado en la puerta de enlace, por lo que puede encontrar el registro.
- Eliminamos la parte de la configuración que deshabilita Ribbon, el equilibrador de carga. Lo veremos en acción más adelante en este capítulo.

ENGAÑARNOS A NOSOTROS MISMOS CON EL DESCUBRIMIENTO DEL SERVICIO Y EL GATEWAY

El comportamiento predeterminado en Zuul para agregar automáticamente rutas para servicios en el registro es arriesgado si no se usa correctamente. como se mencionó, si no agregamos elzuul. ignoredServicespropiedad, obtendremos la siguiente configuración de enrutamiento *gratis,* sin necesidad de agregar nada a nuestro archivo de propiedades:

- / multiplicación / multiplicaciones / ** -> microservicio de multiplicación + /multiplicaciones / **
- / multiplicación / resultados / ** -> multiplicación microservicio + /resultados / **

- / gamificación / líderes / ** -> microservicio de gamificación + / líderes / **
- / gamification / stats / ** -> microservicio de gamificación + / stats / **

sin embargo, si trabajamos con esas URL, ya no podemos decir que nuestros consumidores son agnósticos a los microservicios: estarían apuntando nuevamente a nuestra estructura interna. Perdemos uno de los principales beneficios de la pasarela api.

Pero algunas personas saben cómo jugar ese juego y hacerlo funcionar. si modelamos nuestros microservicios para proporcionar funcionalidad en torno a una sola entidad comercial, podemos utilizar el enrutamiento dinámico. Piense en esta situación por un momento: dividimos la funcionalidad dentro del microservicio de multiplicación y extraemos el / resultados punto final a un nuevo microservicio, gestor de resultados. después de eso, refactorizamos el código en nuestros controladores para mover la funcionalidad a la raíz (p.ej, @RequestMapping ("/ multiplicaciones") se convertiría claramente @RequestMapping ("/")). luego, para que todo funcione de manera transparente, cambiamos el nombre de la aplicación de nuestro microservicio de multiplicación a multiplicacionesdesagradable), y el nombre de ese gerente de resultados ficticio para resultados. desde el prefijo de tira La propiedad es verdadera de forma predeterminada, el nombre del servicio se elimina de la URL cuando pasa por la puerta de enlace. ya que su cabeza podría estar a punto de explotar, detallemos el resultado en tres sencillos pasos:

1. el cliente solicita OBTENGA http: // localhost: 8000 / api / multiplications / 1.
2. Zuul tiene una ruta agregada dinámicamente desde el registro, mapeando / multiplicaciones (nombre del servicio) al servicio de multiplicaciones. desdeprefijo de tira es cierto, lo asigna a OBTENER http: // localhost: 8080/1.
3. Refactorizamos nuestro controlador para aceptar solicitudes de root, por lo que la multiplicación con iD 1 es regresado. esto funciona porque ahora el microservicio solo maneja entidades de multiplicación.

lo mismo se aplicaría a la /resultados punto final. yendo aún más lejos, también podríamos dividir el microservicio de gamificación, y luego terminaríamos con un sistema en el que las rutas se establecen mágicamente. Esta forma de trabajar con la puerta de enlace y el descubrimiento de servicios se difunde lamentablemente en muchas guías rápidas en Internet, lo que confunde a las personas sobre el propósito real del descubrimiento de servicios. en el mundo real, los microservicios no necesariamente se asignan con una sola entidad comercial. por el contrario, el patrón de puerta de enlace de API está ahí para que tengamos el control total de dónde y cómo redirigimos las solicitudes a los microservicios; y el descubrimiento de servicios no se trata de descubrir automáticamente la funcionalidad, sino de proporcionar una forma eficiente de encontrar y equilibrar la carga entre una o varias instancias del mismo microservicio.

si no le gusta especificar rutas explícitas (lo cual es justo cuando su sistema tiene docenas de microservicios), hay formas de cargarlos dinámicamente creando su propia implementación del RouteLocator interfaz o utilizar cualquiera de los existentes.

Jugando con Service Discovery

Con el microservicio adicional agregado (servicio-registro) y los cambios que hicimos a los demás para usarlo, ahora podemos jugar con nuestro sistema y ver cómo funciona el descubrimiento de servicios en combinación con la puerta de enlace API. Cuando llega una nueva solicitud, Zuul usa el cliente de descubrimiento para encontrar una URL adecuada para elserviceId y luego redirige la solicitud allí. No se especifican direcciones físicas, por lo que obtenemos flexibilidad para mover o refactorizar servicios en nuestro sistema, implementar varias instancias, etc. Con la introducción de la puerta de enlace API y los microservicios de descubrimiento de servicios, finalmente obtuvimos los beneficios que deseamos.

Para iniciar el sistema, debemos seguir los pasos que ya conocemos, agregando uno adicional para ejecutar el registro del servicio.

1. Ejecute el servidor RabbitMQ (si aún no se está ejecutando).
2. Ejecute el microservicio de registro de servicios.

3. Ejecute el microservicio de puerta de enlace.

4. Ejecute el microservicio de multiplicación.

5. Ejecute el microservicio de gamificación.

6. Ejecute el servidor Jetty desde el ui carpeta raíz.

Igual que antes, el único para el que el orden es importante es el primero. Tenga en cuenta que solo es conveniente, pero no obligatorio, que iniciemos el registro de servicios antes que los microservicios. Si no lo hacemos, nuestros microservicios funcionarán de todos modos, ya que volverán a intentar el proceso de registro hasta que el registro esté disponible.

Puede experimentar en algunas ocasiones que la interacción entre la puerta de enlace y el registro del servicio toma algún tiempo para ser efectiva, por lo que puede obtener algunos errores del servidor (código de estado HTTP 500) de la puerta de enlace si juega con la aplicación dentro del primer minuto después de arrancar todos los microservicios. solo necesito darle más tiempo al registro de servicios para que haga su trabajo. Más adelante en este capítulo, cubriremos cómo lidiar con estos errores de una mejor manera usando el patrón de Disyuntor.

La función de descubrimiento de servicios se nota de diferentes formas. Primero, cuando iniciamos los microservicios, vemos un conjunto de mensajes en la salida de la consola, como se muestra en Listado5-19.

Listado 5-19. Multiplicación de salida de consola (multiplicación v8)

```
2017-09-27 15: 41: 45.238 INFO 656 ---                    principal] com.
[netflix.discovery.DiscoveryClient: Obteniendo todas las instancias
información de registro del servidor eureka 2017-09-27 15: 41:
45.471 INFO 656 --- [netflix.discovery.DiscoveryClient: El estado de  principal] com.
respuesta es
200
2017-09-27 15: 41: 45.472 INFO 656 ---                    principal] com.
[netflix.discovery.DiscoveryClient: latido inicial
ejecutor: intervalo de renovación es: 30
```

2017-09-27 15: 41: 45.474 INFO 656 - [principal] cndiscovery.InstanceInfoReplicator : La tasa de actualización permitida de InstanceInfoReplicator onDemand por minuto es 4 2017-09-27 15: 41: 45.477 INFO 656 --- [main] com. netflix.discovery.DiscoveryClient: Discovery Client inicializado en la marca de tiempo 1506519705477 con el recuento de instancias iniciales: 3
2017-09-27 15: 41: 45.503 INFO 656 --- [principal] osc . nesEurekaServiceRegistry: Registro de la aplicación multiplicación con eureka con estado UP 2017-09-27 15: 41: 45.504 INFO 656 --- [netflix.discovery.DiscoveryClient: Vio el cambio de principal] com. estado local evento StatusChangeEvent [timestamp = 1506519705504, current = UP, previous = STARTING]
2017-09-27 15: 41: 45.505 INFO 656 --- [nfoReplicator-0] com. netflix.discovery.DiscoveryClient_ MULTIPLICACIÓN / localhost: multiplicación: 8080: servicio de registro ...
2017-09-27 15: 41: 45.540 INFO 656 --- [nfoReplicator-0] com. netflix.discovery.DiscoveryClient_ MULTIPLICACIÓN / localhost: multiplicación: 8080 - estado de registro: 204)

Como podemos ver, Eureka está registrando el servicio con éxito después de verificar que el registro está activo. También podemos verificar el estado de los diferentes servicios desde nuestro navegador. La interfaz web de nuestro servidor Eureka (el Panel de control del servidor Eureka) se encuentra enhttp: // localhost: 8761 /.
Allí podemos encontrar una página web que muestra información sobre qué servicios están registrados y su estado, junto con algunos detalles sobre el propio registro, como se muestra en la Figura 5-14.

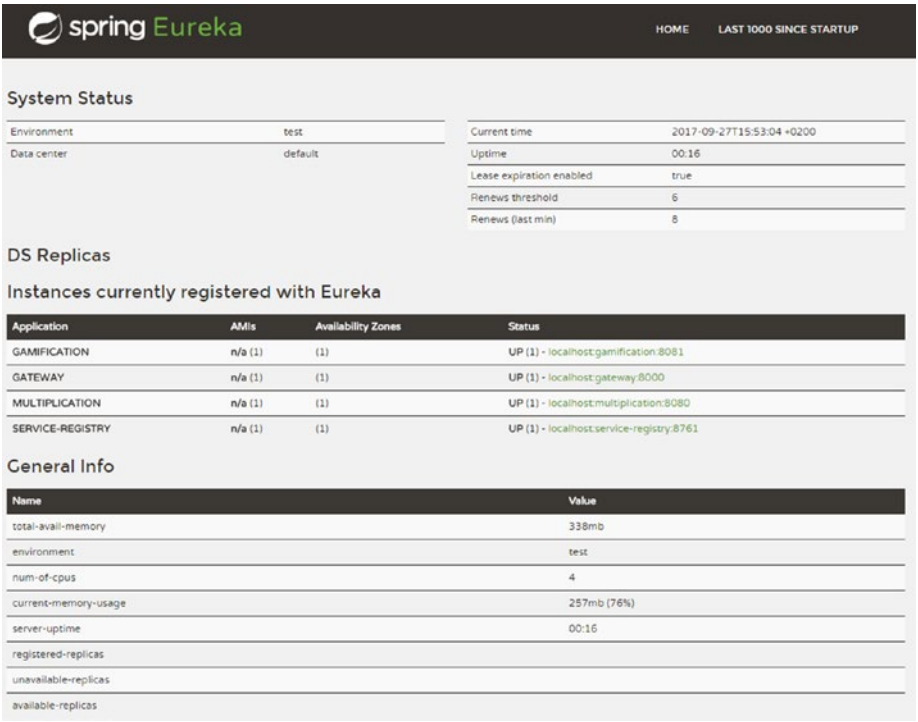

Figura 5-14. *Panel de Eureka que muestra información sobre los servicios*

Desde el punto de vista del usuario, no hay muchos cambios. Todavía podemos enviar intentos y actualizar la tabla de clasificación. Los grandes cambios están detrás de escena: las solicitudes pasan por la puerta de enlace API, que usa el registro para encontrar las instancias y redirigirlas al microservicio adecuado. Nuestro sistema está evolucionando muy bien hacia una arquitectura de microservicios adecuada.

Tenga en cuenta que dejamos atrás la parte más interesante sobre el descubrimiento de servicios: escalar nuestro sistema iniciando múltiples instancias del mismo microservicio. Veremos cómo Ribbon resuelve eso por nosotros pero, antes de eso, necesitamos verificar si nuestros microservicios están preparados para ello.

¿Están nuestros microservicios listos para escalar?

Antes de continuar nuestro camino a través de todas estas herramientas, detengámonos un momento para analizar lo que vamos a hacer. Queremos iniciar varias instancias de nuestros servicios y dejar que Zuul, la puerta de enlace API, decida a qué instancia redirigir cada solicitud, respaldada por Eureka y el registro de servicios. Pero, ¿podemos realmente hacer eso? ¿Va a funcionar con nuestras implementaciones de microservicios actuales?

Una de las cosas más críticas cuando comenzamos a trabajar con sistemas escalables es que debemos ser conscientes de algunos conceptos básicos importantes al diseñar nuestros microservicios. No podemos simplemente agregar herramientas para proporcionar descubrimiento y enrutamiento de servicios y esperar que todo funcione de inmediato. Necesitamos ser conscientes de hacia dónde nos dirigimos y prepararnos para ello. Analicemos si nuestra estrategia de datos y nuestras interfaces de comunicación están alineadas con nuestros objetivos.

Bases de datos y servicios sin estado

Primero, nuestros servicios deben ser apátridas, lo que significa que no deben guardar ningún dato o estado en la memoria. De lo contrario, necesitamos tener*afinidad de sesión:* Todas las solicitudes del mismo usuario deben terminar en la misma instancia de microservicio, porque mantiene cierta información de contexto. Para evitar complicar demasiado nuestras aplicaciones, es mejor si siempre diseñamos microservicios sin estado.

En nuestro sistema, las bases de datos están integradas en los servicios, lo que nos impide escalar correctamente. Cada instancia de nuestro servicio no debería tener su propia base de datos, ya que eso provocaría la recuperación de datos diferentes por solicitud. Todas las instancias deben mantener sus datos en el mismo lugar, en el mismo servidor de base de datos compartido.

La base de datos H2 también puede funcionar en modo servidor. Solo necesitamos habilitar una opción para que esto funcione para nuestras bases de datos:*modo mixto automático.* Solo necesitamos agregar el sufijo AUTO_SERVER = TRUE a todas nuestras URL de JDBC. Recuerde agregarlo a las URL de JDBC de multiplicación y gamificación. Ver listado5-20.

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