```
        // Cuándo
        MockHttpServletResponse respuesta = mvc.perform (
                get ("/ results"). param ("alias", "john_doe"))
                . andReturn (). getResponse ();

        // luego
        assertThat (response.getStatus ()). isEqualTo (Http
        Status.OK.value ());
        assertThat (response.getContentAsString ()). isEqualTo (
                jsonResultAttemptList.write (
                        RecentAttempts
                ). getJson ());
    }

}
```

En el lado de la interfaz de usuario, debemos llamar a esta nueva API REST y presentar los resultados en la pantalla. Primero, agregamos la lógica amultiplication-client.js para llamar al servicio de backend por cada intento enviado, como se muestra en el Listado 3-45.

Listado 3-45. Intentos de adición de multiplication-client.js (socialmultiplication v4)

```
// [...]
función updateStats (alias) {
    $ .ajax ({
        url: "http: // localhost: 8080 / results? alias =" + alias,}).
    luego (función(datos) {
        $ ('# stats-body'). empty ();
        data.forEach (función(fila) {
            $ ('# stats-body'). append ('<tr> <td>' + row.id + '</
            td>' +
                '<td>' + fila.multiplicación.factorA + 'x' +
                fila.multiplicación.factorB + '</td>' +
```

```
                    '<td>' + row.resultAttempt + '</td>' + '<td>' +
                    (row.correct === cierto ? 'SÍ NO')
                    + '</td> </tr>');
            });
        });
}

$ (documento) .ready (function () {

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

    enviar (función( evento) {

        // [...]

        updateStats (userAlias);
    });
});
```

Luego agregamos una tabla bastante básica al código HTML para representar los resultados, como se muestra en Listado 3-46.

Listado 3-46. index.html Tabla de intentos de adición (socialmultiplication v4)

```
<! DOCTYPE html>
<html>
<cabeza>
    <título>Multiplicación v1 </título> <enlace rel = "hoja de estilo"
    type = "text / css" href = "styles.css">
    <guión src = "https://ajax.googleapis.com/ajax/libs/
            jquery / 3.1.1 / jquery.min.js "> </secuencia de comandos> <secuencia
de comandos src = "multiplication-client.js"> </secuencia de comandos> </head>
```

```
<cuerpo>
<div>
    <h1>Bienvenido a Social Multiplication </h1>
    <h2>Este es tu desafío de hoy: </h2> <h1>

        <intervalo class = "multiplicación-a"> </lapso> x <lapso
class = "multiplicación-b"> </intervalo> =
    </h1>
    <p>
        <formulario id = "intento-formulario">
            ¿Resultado? <aporte type = "text" name =
            "resultantettempt"> <br>
            Tu alias: <aporte type = "text" name =
            "useralias"> <br>
            <entrada tipo = "enviar" valor = "Verificar">
        </form>
    </p>
    <h2> <intervalo class = "mensaje-resultado"> </intervalo> </h2>
    <h2>Estadísticas </h2>
    <tabla id = "stats" style = "ancho: 100%">
        <tr>
            <th>ID de intento </th>
            <th>Multiplicación </th>
            <th>Ingresaste </th> <th>
            ¿Correcto? </th>
        </tr>
        <tbody id = "stats-body"> </tbody> </
        table>
</div>
</body>
</html>
```

¡LO HICIMOS DE NUEVO! LA HISTORIA DE USUARIO 2 HA TERMINADO

¡Completamos nuestros nuevos requisitos! Creamos nuestro modelo de datos, lo implementamos en nuestras entidades y creamos los repositorios para persistir y recopilar nuestros datos. También expusimos un nuevo punto final para recuperar estos últimos intentos e incluimos un nuevo componente en la interfaz de usuario. ¡Es hora de jugar con nuestra nueva funcionalidad!

Jugando con la aplicación (Parte II)

¡Finalmente! Tenemos una segunda versión de la aplicación, ahora con persistencia incluida. Ya analizó la lógica de la multiplicación antes, por lo que ahora puede concentrarse en la nueva función: persistencia y visualización de intentos.

Ejecute la aplicación usando el código o empaquetándola, como vio anteriormente. Luego navega ahttp: // localhost: 8080 / index.html con su navegador. Intenta resolver algunas multiplicaciones, asegurándote de intentarlo al menos cinco veces. Obtendrás algo como Figura3-4.

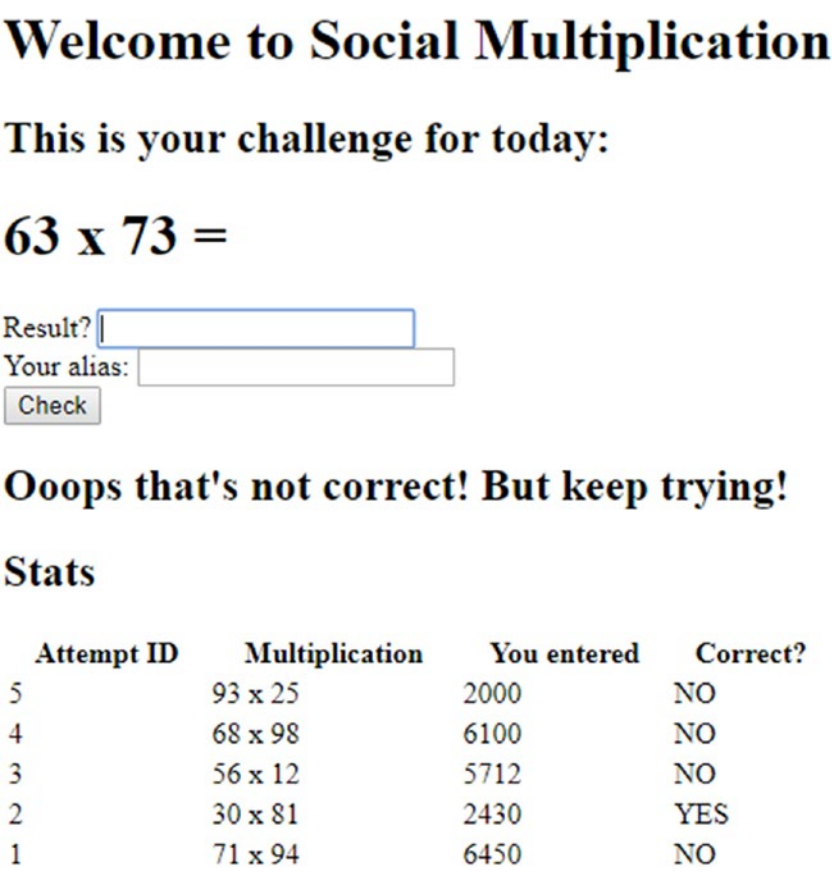

Figura 3-4. *Aplicación mejorada que enumera los intentos anteriores.*

¡Luciendo mejor! La interfaz es simple, pero lo suficientemente buena. De todos modos, lo mejoraremos en un capítulo posterior. Si desea ver los datos, puede navegar ahttp: // localhost: 8080 / h2-console /. Verá la pantalla de inicio de sesión de la Consola H2, donde puede ingresar la URL de JDBC (en caso de que aún no esté allí): jdbc: h2: file: ~ / social-multiplication. Deje los otros campos como están. Ver figura3-5.

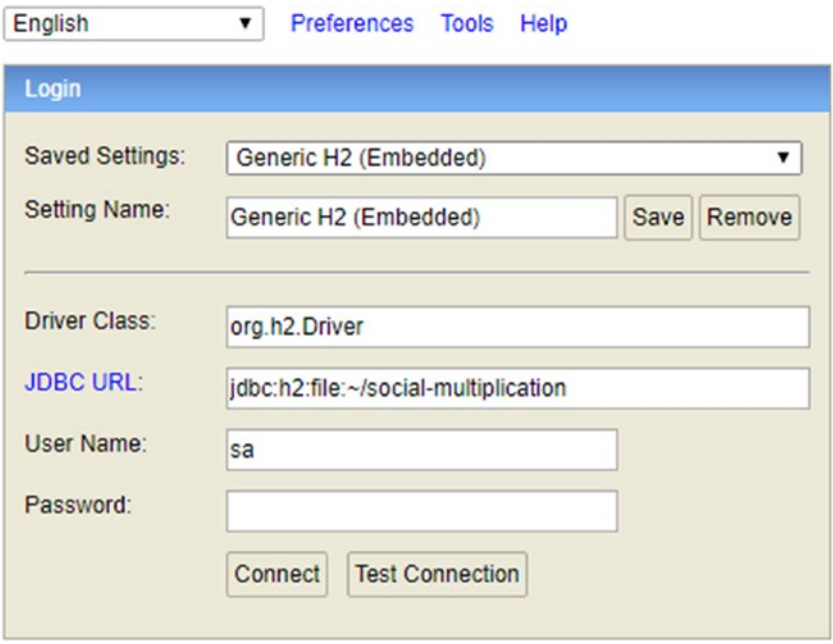

Figura 3-5. *Ingrese la URL de JDBC aquí*

Cuando hace clic en Conectar, se le presenta una interfaz rudimentaria desde la cual puede ejecutar todo tipo de comandos en la base de datos (ver Figura 3-6). Si hace clic en el nombre de una tabla, la consola generará un estándarseleccionar todo consulta para usted que luego puede ejecutar haciendo clic en el botón Ejecutar.

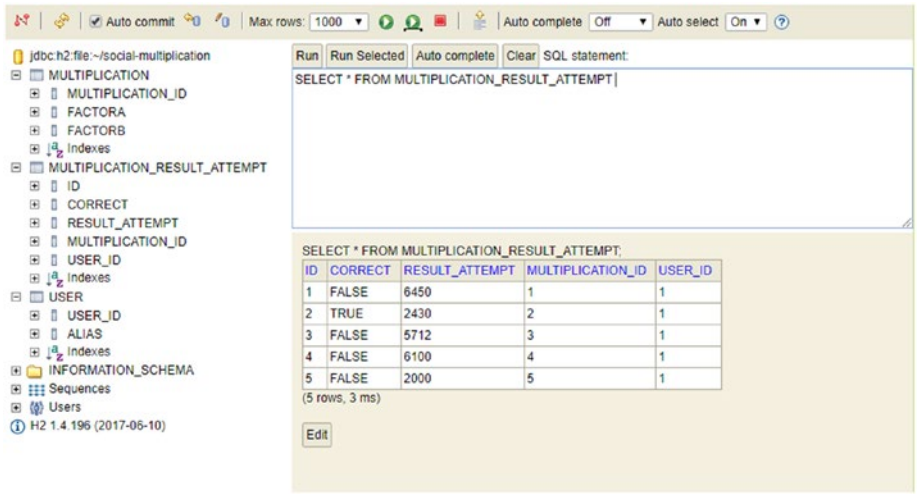

Figura 3-6. *Puede ejecutar comandos a la base de datos desde esta pantalla*

Resumen

En este capítulo, aprendió cómo construir su primera aplicación de la vida real usando Spring Boot. Lo usará como uno de sus microservicios en un capítulo posterior, cuando aprenderá acerca de la funcionalidad distribuida en múltiples aplicaciones.

Usamos un enfoque ágil simulado para ofrecer valor lo antes posible y creamos una aplicación web completa en solo dos iteraciones (representadas como dos historias de usuario). Este capítulo trató de mostrarle cómo es preferible comenzar simple y luego aplicar modificaciones cuando sea necesario: comenzamos sin una base de datos y luego desarrollamos nuestro código para incluirlo.

Otro concepto importante en este capítulo fue el desarrollo basado en pruebas: tomamos el caso de uso e implementamos la prueba unitaria, antes de codificar la implementación. Si se acostumbra a este enfoque, verá cuántas ventajas le aporta, una de las más importantes es la mejora de la definición de sus requisitos funcionales.

Sin duda, el tema más importante ha sido el diseño de software adecuado de la aplicación: utilizó un enfoque de tres niveles, colocándolo en capas en dominio, aplicación, presentación y datos. Este es un patrón bien conocido por sus beneficios relacionados con el acoplamiento flexible y la clara separación de responsabilidades. La primera parte del capítulo se centró en la lógica empresarial y las capas de presentación, incluida una interfaz sencilla para permitir la interacción del usuario. Luego, obtuvimos nuevos requisitos que nos llevaron a diseñar y desarrollar una capa de datos. Pero, antes de eso, necesitábamos detenernos por un tiempo para preparar nuestra aplicación para eso; necesitábamos una tarea de refactorización, como suele ocurrir en la vida real. Finalmente, diseñamos e implementamos nuestro modelo de datos y repositorios, y los conectamos en todas las capas hasta la interfaz de usuario para mostrar los últimos intentos enviados por los usuarios.

Puede aplicar ahora el conocimiento de este libro para escribir aplicaciones Spring Boot bien diseñadas y en capas. Pero todavía hay mucho más que aprender: una vez que obtenga varias de estas aplicaciones, ¿cómo se conectarán entre sí? ¿Cómo ven las instancias de los demás si comienzan a escalar? Estas son las preguntas que cubriremos en los próximos capítulos. Es hora de moverme a*microservicios.*

CAPÍTULO 4

Empezando con Microservicios

El enfoque del pequeño monolito

Capítulo 3 terminó con una única aplicación implementable que contenía no solo las diferentes funcionalidades de backend sino también el lado de frontend. Nuestra aplicación es una *pequeño monolito.* Como alternativa, podríamos haber comenzado a diseñar un sistema completo, identificando los diferentes contextos (o contextos acotados)[1] en él, mapeándolos a los microservicios, y luego desarrollándolos todos desde el principio, al mismo tiempo.

Puede que se sienta tentado a seguir esa estrategia. Una buena razón para hacerlo es que podría tener varios equipos trabajando en paralelo en diferentes microservicios, por lo que podría aprovechar la asignación de microservicios a equipos desde el principio; días felices, podría terminar antes. Mi experiencia dice que no hagas eso, y no soy el único que aboga por un enfoque de monolito primero (verhttps://tpd.io/monofirst).

Cuando desarrolla software de forma ágil, no puede esperar mucho tiempo para entregar su software. Tampoco puede pasar semanas diseñando su sistema completo de antemano, con detalles. Dentro del tiempo de ejecución del producto o proyecto, es mejor que entregue partes completas de su software. Si comienza a dividir su proyecto en microservicios desde el principio

[1] https://tpd.io/bounded-ctx

M. Macero, *LearnMicroservices con Spring Boot,*
https://doi.org/10.1007/978-1-4842-3165-4_4

usando una hoja de papel (o un bonito dibujo digital), ese no será el caso: llevará mucho más tiempo que construir un monolito. ¿Por qué? Porque técnicamente es más difícil implementar, orquestar y probar un sistema basado en microservicios.

Hay otra buena razón para no empezar desde cero con los microservicios: es probable que su sistema tenga un diseño de software deficiente, peor que cuando se construye un monolito. Esta predicción tiene más que ver con la forma en que trabaja la gente: cuando divide el trabajo en diferentes partes y las asigna a diferentes equipos, los equipos a menudo comienzan a preocuparse por sus piezas de trabajo y no por todo el sistema. El diseño, como se hizo al principio, se puede corromper fácilmente. Los equipos pueden comenzar a ignorar las pautas y principios comunes de la empresa a menos que usted mantenga todos los microservicios bajo control (y eso es un trabajo muy duro para los arquitectos). Las pruebas de extremo a extremo son mucho más difíciles de configurar con todas esas piezas evolucionando por sí mismas. En este escenario, Supongo que ya tenía una idea clara de las API y las formas de comunicación entre sus diferentes microservicios. Si ni siquiera tienes eso, me quedaría con el monolito para empezar, sin duda.

Eso no quiere decir que no pueda predecir con éxito los límites de sus microservicios y desarrollarlos en paralelo, pero, en ese caso, debe prestar mucha más atención a la implementación, las pruebas de integración, el cumplimiento de estándares comunes, API claras, el registro y el monitoreo, manejo de errores, canales de comunicación entre equipos, etc. Fallar solo en uno de estos temas puede poner en peligro tu proyecto si comienzas directamente con microservicios.

Considere un enfoque mejor: planifique primero una aplicación monolítica. Planifíquelo de manera que se pueda dividir más tarde con poco esfuerzo:

- *Compartimente su código en paquetes raíz que definen los contextos de su dominio.* Por ejemplo, su aplicación puede tener funcionalidades relacionadas con los clientes (persona, empresa, dirección, etc.) y otras relacionadas con pedidos (pedido

generación, despacho, manipulación, etc.). En lugar de empaquetar su estructura raíz directamente por capas, puede crear niveles superiores donde primero divide los clientes y los pedidos. Luego, replique las capas para cada uno de ellos (por ejemplo, controlador, repositorio, dominio y servicio) y asegúrese de seguir las buenas prácticas para la visibilidad de la clase (las implementaciones son privadas de paquetes). Las principales ventajas que obtiene con esta estructura son que mantiene la lógica empresarial inaccesible en todos los contextos de dominio, y que más adelante debería poder extraer un paquete raíz completo como un microservicio si lo necesita, con menos refactorización.

- *Aproveche la inyección de dependencias: base su código en interfaces y deje que Spring haga su trabajo inyectando las implementaciones.* Refactorizar usando este patrón es mucho más fácil. Por ejemplo, puede cambiar una implementación para llamar a un microservicio diferente en lugar de mantener toda la lógica empresarial en un solo lugar (si eso tiene sentido para usted cuando diseña sus límites).
- *Una vez que haya identificado los contextos (por ejemplo, clientes y pedidos), déles un nombre coherente en toda su aplicación.* Mueva su lógica de dominio aquí y allá (más fácil con un pequeño monolito) durante la fase de diseño hasta que los límites estén claros y luego respete los límites. Nunca tome atajos que enreden la lógica empresarial en distintos contextos solo porque puede hacerlo. Tenga siempre en cuenta que el monolito debe estar preparado para evolucionar.

- *Encuentre patrones comunes e identifique lo que luego se puede extraer como bibliotecas comunes, por ejemplo.*

- *Utilice revisiones de pares*[2] *para asegurarse de que los diseños de la arquitectura se entiendan y sigan.*
- *Comuníquese claramente con el gerente del proyecto para planificar el tiempo en versiones posteriores para dividir el monolito.* Explica la estrategia y crea la cultura: la refactorización va a ser necesaria y no tiene nada de malo.

Trate de mantener un pequeño monolito al menos hasta su primer lanzamiento. No le tenga miedo, un pequeño monolito le brindará muchas ventajas:

- Avanza y tiene algo con qué jugar: los usuarios comerciales pueden comenzar a verificar si eso es lo que querían y hacer ajustes.
- Puede cambiar fácilmente el modelo de dominio que creó: verifique si es lo suficientemente bueno o no y adáptelo.
- Su (s) equipo (s) se acostumbrarán a las pautas técnicas y funcionales comunes de la empresa para el proyecto.
- La funcionalidad común entre dominios se puede identificar y extraer como bibliotecas.
- Todos trabajan con la primera versión del sistema completo, por lo que es más fácil que comprendan la vista completa y no solo las partes.

Por otro lado, existen algunas desventajas que puede intentar paliar:

- Sabes que estás construyendo un monolito y eso no se siente bien. Sin embargo, esto no debería molestarle demasiado si sigue buenos patrones de diseño que le ayudarán a dividirlo más tarde (compartimentarlo, utilizar interfaces, etc.).

[2]Si no está familiarizado con las revisiones por pares, consulte https://www.atlassian.com/agile / code-reviews.

- Su primera estrategia de implementación quedará parcialmente obsoleta. Pero puede reutilizar sus herramientas existentes de la misma manera, simplemente adaptándolas a múltiples servicios.
- Demasiadas personas que trabajan al mismo tiempo en la misma base de código pueden ser ruidosas e ineficientes. Mi recomendación aquí es comenzar poco a poco con el equipo también. Trabaje con los gerentes de proyecto: si el alcance está planeado para entregar un monolito pequeño y bueno, el equipo no necesita ser grande para la primera versión. Los recursos se pueden emplear de forma diferente: sería incluso mejor empezar en paralelo dos alternativas diferentes para el monolito pequeño, con la misma funcionalidad, y luego descartar una de ellas cuando llegue a la primera versión.

Analizando el monolito

Figura 4-1 describe cómo se ve la aplicación hasta ahora.

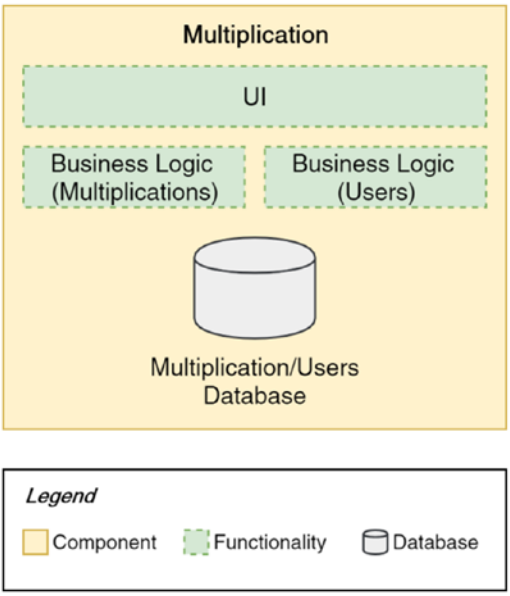

Figura 4-1. *La aplicación hasta ahora*

Hasta ahora hemos estado construyendo un pequeño monolito. Después de extraer los requisitos para la primera historia de usuario, podemos identificar al menos tres partes:

- La *dominio de multiplicación,* que se encarga de la generación y verificación de operaciones.
- La *dominio de usuario,* que maneja información relacionada con los usuarios.
- La *Componente de interfaz de usuario,* que se comunica con la API REST y contiene la página web.

Estas son las partes que podrían haberse iniciado de forma independiente e implementadas como tales, con algún rediseño de las referencias entre los objetos de dominio de la multiplicación y el usuario. Tenga en cuenta que no estamos considerando la interfaz de usuario como un dominio; es solo un componente técnico de nuestra aplicación que se puede extraer por separado. Estamos exponiendo el comportamiento de nuestra aplicación a través de una API REST, por lo que tiene más sentido tener la interfaz de usuario web en un componente diferente que se puede implementar de forma independiente. Una de las razones es que podríamos tener una aplicación móvil que consuma la misma API, que no necesita las interfaces web. Además, la interfaz de usuario podría evolucionar a una aplicación de una sola página que consuma API de diferentes servicios (por ejemplo, multiplicación y usuario). Por todas estas razones, verá que el código de la interfaz de usuario a menudo se coloca en un servicio diferente en muchos proyectos de software.

En cualquier caso, no dividimos nada desde cero al crear nuestra aplicación. Las razones, como puede imaginar, son las que se cubrieron en la subsección anterior sobre las ventajas de un enfoque de monolito primero. Tenga en cuenta que los ejemplos de libros pueden ser trampas: no puede vivir el proceso completo de diseño y desarrollo, por lo que es más difícil venderle la idea de una mejor combinación entre el pequeño monolito y la experimentación rápida. Todavía podría preguntarse por qué este libro no comenzó directamente con un conjunto predefinido de servicios que funcionan perfectamente todos juntos. Podría haberme saltado el monolito para hacer el código más agradable, pero eso iría en contra de la idea básica de este libro: ser un proyecto de software práctico y en evolución.

como lo harías en la realidad. Pero no solo eso, si hubiéramos comenzado con el ecosistema completo de servicios y las herramientas necesarias para respaldarlos, el objetivo comercial completo de nuestra aplicación de ejemplo habría sido *diluido por detalles técnicos.* Para explicarlo mejor en la práctica, si hubiéramos tomado la decisión de comenzar directamente con microservicios para nuestros primeros requisitos (historia de usuario 1), habríamos tenido que revisar casi todo el libro antes de entregar nuestra primera aplicación funcional.

Avanzando

¿Cuáles son los próximos pasos para hacer evolucionar nuestra aplicación? Mantendremos la aplicación de multiplicación tal como está por un poco más de tiempo, ya que antes de hacer la refactorización completa, introduciremos una segunda aplicación Spring Boot en nuestro sistema. Y eso, como siempre, está impulsado por una nueva funcionalidad.

HISTORIA DE USUARIO 3

Como usuario de la aplicación, quiero estar más motivado para participar cada día, así no me rindo fácilmente.

Tenga en cuenta que este también se puede escribir desde otra perspectiva: "Como administrador de la aplicación, quiero que los usuarios regresen todos los días para luego poder monetizar las visitas recurrentes".

En este escenario ficticio, podría haber algunas personas en el equipo que conozcan *técnicas de gamificación* y desea aplicarlos aquí para animar a los usuarios a volver todos los días a la página. Digamos que la idea se hunde, así que ahora el equipo de arquitectura decide cómo solucionarlo: somos nosotros.

La gamificación parece un mundo completamente diferente, nada que ver con resolver multiplicaciones. Es nuestra oportunidad perfecta para diseñar nuestro modelo y luego hacerlo realidad como una aplicación Spring Boot separada, ahora que ya tenemos la primera versión de la aplicación actual en funcionamiento.

Ahora nos estamos moviendo a una arquitectura de microservicios, en la que nuestra aplicación Spring Boot existente se convertirá en la *microservicio de multiplicación,* y la nueva aplicación se convertiría en la *gamificaciónmicroservicio.*

Pero antes de entrar en más detalles, permítanme presentarles algunos conceptos básicos sobre las técnicas de gamificación y discutir cómo las vamos a aplicar a nuestro sistema.

Conceptos básicos de gamificación

Gamificación es el proceso de diseño en el que se aplican técnicas utilizadas comúnmente en los juegos a algún otro campo, que inicialmente no era un juego. Normalmente lo haces porque quieres obtener algunos beneficios conocidos de los juegos, entre otros, motivar a los jugadores e interactuar con tu proceso, aplicación o lo que sea que estés.*gamificando.*

También es importante aclarar qué es **no** gamificación: no se trata de tratar de manipular a las personas. No es magia que puedas aplicar en todas partes para hacer que a la gente le guste algo.

La aplicación de técnicas de juego puede ser bastante compleja y entra fácilmente en grandes áreas de conocimiento como la motivación, los intereses personales y la psicología en general. Cubriremos una simplificación basada en algunas herramientas básicas disponibles para el diseñador del juego: puntos, insignias y tablas de clasificación.

Puntos, insignias y tablas de clasificación

Una idea básica sobre cómo convertir las cosas en un juego es presentar *puntos:* cada vez que realizas una acción, y lo haces bien, obtienes algunos puntos. Incluso puede obtener puntos si no lo hizo tan bien, pero debería ser un mecanismo justo: obtiene más si lo hace mejor. Ganar puntos hace que el jugador se sienta como si estuviera progresando, lo que le da*realimentación.*

Tablas de clasificación hacer que los puntos sean visibles para todos, de modo que motiven a los jugadores activando sentimientos de competencia. Queremos obtener más puntos que la persona que está por encima de nosotros y clasificarnos más alto. Esto es aún más divertido si juegas con amigos.

Por último, si bien no menos importante, *insignias* son símbolos virtuales de lograr un *estado.* Nos gustan las insignias; dicen más que puntos. Además, pueden representar diferentes cosas: puedes tener los mismos puntos que otro jugador (por ejemplo, cinco respuestas correctas), pero podrías haberlos ganado de una manera diferente (por ejemplo,*cinco en un minuto!).*

Existen diferentes aplicaciones de software que no son juegos pero que utilizan muy bien estos elementos, siendo StackOverflow mi favorito. Puedes mirar la página web stackoverflow.com si aún no la conoces: todo está diseñado con elementos del juego para animar a la gente a seguir participando.

Es importante tener en cuenta que el uso de estas herramientas no hará que tu aplicación sea un juego bien diseñado, ya que hay muchos otros aspectos que también deben tenerse en cuenta si queremos lograr un mejor resultado. Sin embargo, en aras del objetivo principal de este libro, esto es más que suficiente.

Aplicándolo al ejemplo

Lo que haremos es asignar puntos a cada respuesta correcta que envíen los usuarios. Para simplificar, solo daremos puntos si envían una respuesta correcta. En lugar de dar un punto, que no se siente tan bien, lo haremos 10 puntos por respuesta correcta.

A *tabla de clasificación* con los puntajes más altos se mostrarán en la página, para que los jugadores puedan encontrarse en el ranking y competir con otros.

Crearemos también algunas insignias básicas: *Bronce (*10 intentos correctos), *Plata (*25 intentos correctos), y *Oro (*50 intentos correctos). Las insignias no deberían ser extraordinariamente difíciles de obtener, porque eso no motivaría a nuestros usuarios. Debido a que el primer intento correcto puede ser difícil de lograr, también presentaremos la insignia llamada*¡Primero correcto!* para dar una rápida retroalimentación positiva.

Podríamos introducir más insignias en el futuro, algunas otras mecánicas de juego, etc. Pero con estos conceptos básicos, ya tenemos algo que puede motivar a nuestros usuarios a volver y seguir jugando, compitiendo con sus compañeros.

Pasar a una arquitectura de microservicios

Así que decidimos pasar a una arquitectura de microservicios: crearemos una parte diferente de nuestro sistema que se puede implementar de forma independiente y está desacoplada de la lógica empresarial anterior (la *microservicio de gamificación).* Necesitaremos conectar la aplicación Spring Boot existente (que ahora podemos llamar *microservicio de multiplicación)* con el nuevo y asegúrese de que puedan escalar de forma independiente. Y la gran pregunta es: ¿por qué deberíamos hacer eso? ¿Por qué no continuar con el enfoque de un proyecto (monolítico)? Como de costumbre, responderemos esa pregunta desde un punto de vista práctico, basándonos en algunas razones de peso.

Separación de preocupaciones y acoplamiento flojo

Si juntamos la lógica de gamificación con la lógica existente, dentro de la misma base de código y el mismo artefacto desplegable, corremos el riesgo de mezclarlos en el futuro, desechando todas las ventajas de la separación de preocupaciones. Puede pensar que esto no sucederá, pero en realidad cuanto más tiempo tenga estos dominios juntos, mayor será el riesgo de que alguien tome atajos. Especialmente si también almacena sus datos en la misma base de datos.

Como vimos cuando describimos las ventajas de un enfoque de monolito primero, mantener todo junto facilita el diseño y el desarrollo durante la primera fase del proyecto. Sin embargo, si desea migrar a microservicios por sus ventajas, es fundamental encontrar el momento adecuado a lo largo del ciclo de vida del proyecto para dejar de hacer crecer su monolito. De lo contrario, se encontrará viajando de un monolito pequeño a un monolito de tamaño mediano, y de allí a un “monolito certificado”. Por supuesto, el riesgo es diez veces mayor si el proyecto de software se gestiona bajo alta presión y pautas estrictas. Este es un entorno perfecto para atajos y para que el monolito crezca sin control.

Es difícil lograr un acoplamiento suelto. Usemos el ejemplo en el que los nuevos requisitos dicen que desea mostrar en una tabla cada multiplicación

con la cantidad de puntos ganados: es muy fácil escribir un *unirse a la consulta* que muchos desarrolladores lo usarán. Incluso si tiene diferentes almacenes de datos, alguien podría escribir una clase usando los diferentes dominios juntos y agregar algo de lógica comercial adicional a eso. El problema es mucho más visible si piensa en un sistema con múltiples contextos de dominio:*servicios de espagueti* mezclando objetos de dominio y lógica empresarial aquí y allá. La jungla. Y luego necesitarías tu machete y mucha paciencia para desenredarlo todo. Tener la multiplicación y la gamificación separadas en diferentes microservicios lo obligará a pensar en una solución poco acoplada: puede replicar parte de los datos o hacer que un servicio llame al otro cuando sea necesario. Veremos un ejemplo de esto en nuestro sistema.

Cambios independientes

Tener servicios desplegables de forma independiente para los dominios de multiplicación y gamificación nos permitirá probarlos por separado, utilizando sus API. En el futuro, podríamos tener al equipo de gamificación evolucionando sus servicios sin interferir en el ciclo de desarrollo del equipo de multiplicación. Si necesitan nuevas interfaces para la comunicación, pueden simplemente crear llamadas o mensajes falsos, por lo tanto, definir los cambios de API y seguir adelante. Tenga en cuenta nuevamente la principal ventaja aquí: saben con certeza que no romperán nada en el servicio de multiplicación y que no se bloquean entre sí. Esta tranquilidad hace que los directores de proyectos duerman por la noche.

Escalabilidad

Imagina que logramos un gran éxito con el juego de multiplicar. Los usuarios comienzan a usarlo desde miles a millones, y pronto el servidor en la nube que elegimos comienza a quedarse sin recursos y no responde a tiempo a las comprobaciones de multiplicación. Así que queremos escalar nuestro sistema y aplicar algunas técnicas de equilibrio de carga.

Si tuviéramos un solo artefacto desplegable (todo el sistema), lo único que podemos hacer es crear varias instancias de ese. Pero eso podría implicar desperdiciar recursos, o al menos no ser lo más flexibles posible. Si tenemos varios servicios, podemos elegir cómo escalar, y una estrategia válida para este caso podría ser escalar el servicio de multiplicación para hacer frente a nuestras nuevas necesidades, pero no la de gamificación, ya que no es tan importante si los puntos y los marcadores son calculado con un retraso. Por supuesto, esa estrategia es más compleja de lograr que la anterior, pero no olvidemos que los servidores y la computación en la nube cuestan dinero, y ahorrar dinero es bueno para cada proyecto de software.

Conexión de microservicios

Crearemos lógica de gamificación en un microservicio separado. Debería *de algun modo* conectarse con nuestro proceso comercial existente de *resolver un intento y obtener comentarios,* extendiéndolo para ser *resolver un intento, obtener comentarios y ganar puntos.*

¿Cómo distribuimos nuestro proceso a través de esos dos microservicios? Si hace esta pregunta a personas que nunca trabajaron con un enfoque basado en eventos, generalmente obtendrá algunas de estas respuestas:

1. Podrían compartir la base de datos para que el servicio de gamificación pueda usar los datos allí directamente.
2. El servicio de gamificación podría recopilar datos periódicamente del servicio de multiplicación y procesarlos según sea necesario para asignar puntos, insignias, etc. Eso podría hacerse exponiendo algunas RESTAPS adicionales en nuestro servicio existente.
3. Cuando algo sucede en el servicio de multiplicación (es decir, se envía un intento), este llamará al servicio de gamificación y pasará los datos, para que este pueda actualizar las estadísticas del juego. Este es un tipo de enfoque de llamada a procedimiento remoto (RPC).

Analicemos estas alternativas. La opción 1 no es un buen enfoque, ya que muchas de las ventajas de los contextos de desacoplamiento se perderían en el mismo momento en que los servicios pueden acceder y mezclar los datos de los demás.

La opción 2 suena mejor, pero requiere un sondeo constante de nuevos datos y un seguimiento de los intentos que ya se han procesado (por ejemplo, solicitando los intentos enviados desde la última vez que los procesó la gamificación).

La opción 3 podría ser preferible a la opción 2 ya que, en este caso, no necesitamos el mecanismo de votación. Pero todavía hay una cosa que se puede mejorar: el servicio de multiplicación no necesita saber sobre el servicio de gamificación. Deberíamos poder diseñar un sistema en el que nuestro servicio de multiplicación pueda vivir sin gamificación y seguir comportándose como está ahora en el estado actual de la aplicación.

Entonces podemos optar por una variación mejorada de la tercera opción, diseñando una forma de comunicación en la que los servicios estén lo más desacoplados posible. La multiplicación notificará, a quien esté interesado, que se ha introducido un nuevo intento de multiplicación en el sistema, enviando un evento a un bus de mensajes. En el futuro, otras piezas de lógica podrían conectar sus procesos comerciales de forma transparente con el existente sin afectar a los demás:

- ¿Queremos enviar un correo electrónico al administrador cada vez que un usuario muestre un comportamiento sospechoso, es decir, demasiados intentos correctos consecutivos? Nos suscribimos al mismoMultiplicationSolvedEvent y realizar nuestra lógica empresarial en un microservicio diferente.
- ¿Queremos recopilar análisis y generar estadísticas como intentos correctos por usuario, por hora del día, etc.? También es posible con un nuevo microservicio, sin afectar a los demás.

- ¿Queremos agregar un complemento de red social para publicar nuevos intentos resueltos correctamente? Genial, ese es otro microservicio que consume el mismo evento y hace su parte de forma independiente.

Como puede ver, diseñar nuestra funcionalidad siguiendo estos patrones reactivos nos da mucha flexibilidad. Esta forma de modelar nuestra arquitectura se conoce como *arquitectura impulsada por eventos* o *sistemas reactivos.*

Sin embargo, la estrategia impulsada por eventos no encaja en todas las interacciones requeridas entre microservicios. Dentro de nuestros procesos comerciales, podemos tener escenarios en los que los servicios necesitan datos entre sí, no necesariamente relacionados con un evento. En esos casos, no podemos utilizar un enfoque basado en eventos, ya que coinciden con un patrón de solicitud-respuesta. Un ejemplo de eso, basado en nuestros ejemplos anteriores, sería el servicio de red social. Ese necesitaría acceder al alias de usuario (y probablemente algunos otros detalles por implementar). Para ilustrar cómo esta forma de comunicación se combina con un enfoque impulsado por eventos, cubrimos un caso práctico utilizando nuestra aplicación.

Arquitectura impulsada por eventos

En este tipo de arquitectura, los diferentes microservicios envían *eventos* siempre que ocurra una acción importante. Esos eventos se intercambian entre microservicios a través de un intermediario de mensajes (también llamado a veces *bus de eventos).* Otros pueden suscribirse a eventos en los que estén interesados y *reaccionar* a ellos.

Tenga en cuenta un concepto importante: una acción que ya sucedió. Otros no pueden cambiarlo y no pueden evitar que suceda. Es por eso que los nombres de los eventos se dan comúnmente como acciones pasadas:MultiplicationSolvedEvent.
¿Qué harán otros microservicios (si son *suscrito* a este evento) lo procesan de acuerdo con su propia lógica de negocios, lo que podría llevar a que se publiquen otros eventos (por ejemplo, ScoreUpdatedEvent). Sistemas basados en esto

El patrón de acción-reacción también se conoce como *sistemas reactivos,* un concepto que no debe confundirse con *programación reactiva (ver https://tpd.io/rprogsys),* que es un estilo de programación aplicado a un nivel diferente.

Técnicas relacionadas

La arquitectura impulsada por eventos tiene afinidad por algunas otras técnicas: abastecimiento de eventos, diseño impulsado por dominios y CQRS. Puede aplicarlos de forma independiente y siempre debe usarlos de manera razonable. Cuando diseñe su sistema, intente no dejarse seducir por las exageraciones tecnológicas, sino utilícelas como herramientas para resolver sus problemas.

Abastecimiento de eventos es un enfoque para las entidades comerciales persistentes. En lugar de modelarlos con un estado estático que puede cambiar con el tiempo, los modela como secuencias de eventos inmutables. Si usamos un ejemplo común comoCliente, no habría un Cliente tabla en sus datos, sino una secuencia de CustomerChanged eventos. Imaginemos el caso en el que creamos un cliente con un valor dadonombre: John. Lo cambiamos a nombre: Jhonas y nos damos cuenta de que cometimos un error y lo cambiamos de nuevo a
nombre: John. En un método de persistencia tradicional, si verifica los datos después de aplicar estos cambios, solo verá el nombre: John Expresar. Al utilizar el abastecimiento de eventos, su entidad es el estado final de reproducción de la secuencia de eventos. CustomerChanged -> nombre: John (creado), CustomerChanged -> nombre: Jhonas (Error), CustomerChanged -> nombre: John (corrección). Los ejemplos comunes que se utilizan normalmente para el abastecimiento de eventos se basan en aplicaciones bancarias, por lo que este patrón tiene mucho sentido. Su cuenta es una recopilación de transacciones a lo largo del tiempo.

Como puede imaginar, el abastecimiento de eventos se puede implementar más fácilmente en un sistema que se basa en eventos. Sin embargo, no es gratis: puede diseñar su arquitectura impulsada por eventos con unos pocos eventos, pero el abastecimiento completo de eventos puede incrementar significativamente la cantidad de eventos que necesita para modelar. El sistema utilizará una arquitectura impulsada por eventos, pero nuestra persistencia no se basa en el origen de eventos.

Si desea obtener más información sobre el abastecimiento de eventos, el artículo en *https: // tpd.io/evsrc* es un buen punto de partida.

El diseño controlado por dominio (a veces abreviado como DDD) es un patrón aplicado al software, descrito por primera vez por Eric Evans en su libro *Diseño impulsado por dominio: abordar la complejidad en el corazón del software.* Más que un patrón podría definirse incluso como una filosofía para el diseño de software, en el que el dominio de su negocio es el núcleo de su sistema.

Cuando sigue patrones DDD, puede identificar *contextos acotados,* que son como subdominios que pueden tratarse por separado en su sistema. Esto es muy útil al diseñar microservicios, ya que se pueden asignar fácilmente a contextos delimitados y beneficiarse del enfoque DDD. En este libro, seguimos algunos de estos principios.

Para leer más sobre DDD, puede comprar el libro de Eric Evans o descargar gratis el InfoQminibook en https://www.infoq.com/ minilibros / diseño-impulsado por dominio-rápidamente.

CQRS (Segregación de responsabilidad de consulta de comando) es un patrón en el que la *modelo de consulta*para leer) y el *modelo de comando*para escritura) están separados, lo que permite un enfoque de lectura muy rápido a expensas de tener un sistema mucho más complejo. Se puede utilizar junto con el abastecimiento de eventos, siendo el almacén de eventos el modelo de escritura.

Este artículo de Martin Fowler es un buen punto de partida si desea leer más sobre CQRS: https://martinfowler.com/bliki/CQRS.html.

Pros y contras de la arquitectura basada en eventos

Usemos nuestra aplicación para explicar las ventajas y desventajas de una arquitectura impulsada por eventos de una manera práctica. Nuestro escenario es el siguiente: si enviamos un intento de resolver un problema de multiplicación, lo procesaremos en el microservicio de multiplicación y luego enviaremos un MultiplicationSolvedEvent. El nuevo microservicio de gamificación consumirá este tipo de eventos y asignará una nueva puntuación al usuario adecuado. Mantienen sus datos y su funcionalidad separados.

Como verá, algunos de los siguientes temas no se pueden asignar inmediatamente a *ventaja* o *desventaja.* Esas son características de su sistema que, por un lado, pueden darle más trabajo por hacer, pero por otro lado, puede beneficiarse.

Bajo acoplamiento

Con la arquitectura impulsada por eventos, podemos lograr un acoplamiento flexible entre nuestros servicios, como se describe en la sección anterior al analizar las opciones para conectarlos. También puede dividir los procesos grandes en partes más pequeñas, para que múltiples servicios los completen de manera independiente. En nuestro sistema, el*intento de apuntar* El proceso se divide en dos microservicios.

Esta es una gran ventaja: tenemos nuestros procesos allí, pero son *repartido.* No hay lugar en el sistema que controle, y potencialmente enrede, todo lo demás.

Actas

Por otro lado, en una arquitectura basada en eventos, debe asumir que no tiene ACID[3] transacciones entre servicios (o comprenda que, si desea respaldarlos, debe introducir complejidad). En cambio, tiene consistencia eventual, si detiene todas las interacciones con el sistema y deja que todos los eventos se propaguen y se consuman, llegará a un estado consistente.

En nuestro escenario, si el servicio de gamificación está caído y no implementamos ningún mecanismo para prevenirlo, la puntuación no se actualizará. Esto significa que no hay atomicidad en la transacción. Resolver multiplicación - Obtener puntos. Una solución para esto es utilizar una implementación de intermediario de mensajes que garantice la entrega de los eventos al menos una vez.

[3] https://en.wikipedia.org/wiki/ACID

No tener transacciones entre servicios no es malo en sí. El gran riesgo aquí es que requiere un cambio en la forma en que diseña y traduce sus requisitos funcionales (por ejemplo, ¿qué sucede si el proceso se interrumpe en el paso N?).

Tolerancia a fallos

Como consecuencia de no tener (o minimizar) transacciones, la tolerancia a fallas se vuelve más importante en estos sistemas. Es posible que uno de los servicios no pueda completar su parte del proceso, pero eso no debería hacer que todo el sistema falle. Debe evitar que eso suceda (por ejemplo, apuntando a una alta disponibilidad con redundancia de microservicio y equilibrio de carga) y también pensar en una forma de recuperarse de posibles errores (por ejemplo, teniendo una consola de mantenimiento desde la cual pueda recrear eventos).

En cualquier caso, incluir la tolerancia a fallos es buena en cualquier tipo de sistema, no solo en aquellos que utilizan un enfoque basado en eventos. Si lo implementa correctamente, su sistema distribuido puede alcanzar una mayor disponibilidad que un monolito. Los alcances de transacciones grandes que fallan en un monolito se revertirán y los usuarios no pueden hacer nada. Los eventos se ponen en cola para su procesamiento posterior, por lo que pequeñas partes del sistema pueden morir de forma independiente y reiniciarse automáticamente. Eso es mucho más poderoso.

Orquestación y monitoreo

No tener un *capa de orquestación centralizada* puede ser problemático en sistemas donde es fundamental tener un monitoreo de procesos. En una arquitectura impulsada por eventos, usted abarca procesos a través de servicios que se activan y reaccionan a eventos. No puede seguirlos de forma centralizada: están distribuidos en sus microservicios. Para supervisar estos procesos, debe implementar mecanismos para rastrear el flujo de eventos y necesita un sistema de registro común donde pueda realizar un seguimiento de lo que sucede entre los servicios.

Imaginemos que el sistema evoluciona y hay cuatro servicios diferentes que reaccionan al primer evento (MultiplicationSolvedEvent), con algunos otros eventos posteriores que suceden después de eso (ScoreUpdatedEvent ->

LeaderboardPositionChangedEvent -> CongratulationsEmailSent).
Sería difícil hacer un seguimiento de lo que está sucediendo como un proceso comercial de un extremo a otro, a menos que mantengamos manualmente una buena documentación o introduzcamos algo más que realice un seguimiento automático en nuestro código. ¿Cómo sabemos, con solo mirar el sistema desde una perspectiva elevada, que se puede enviar un correo electrónico cuando se resuelve una multiplicación? Podemos implementar nuestro propio mecanismo integrado para correlacionar eventos (etiquetándolos a medida que cruzan los servicios), o podemos usar una herramienta existente como Zipkin.[4]

Evalúe antes de tomar una decisión

En resumen, es importante que evalúe estos factores (y sus ventajas y desventajas) cuando considere implementar un sistema con un enfoque basado en eventos. Si lo hace por las ventajas, tenga en cuenta los inconvenientes y prepare las soluciones.

Otras lecturas

Para evitar romper el enfoque práctico de este libro, no encontrará aquí una descripción extensa de los conceptos de arquitectura impulsada por eventos, pero le proporciono algunos buenos artículos en caso de que esté ansioso por profundizar en los temas:

- Este artículo fromnginx describe los conceptos básicos sobre la arquitectura impulsada por eventos en el contexto de los microservicios. La serie completa es ideal para comprender los conceptos importantes.*https://tpd.io/edd-mgm*
- Este artículo de Microsoft es un poco más técnico, pero brinda información adicional sobre la comunicación. Sin embargo, algunas partes no son fáciles de leer.*https: // tpd. io / ms-ev-arc*

[4] https://github.com/openzipkin/zipkin

- Este artículo de O'Reilly también es interesante ya que muestra dos variaciones diferentes que puede implementar dependiendo de sus necesidades de orquestación. *https: // tpd. io / edvars*

Aplicación de la arquitectura basada en eventos a la aplicación

En este punto del capítulo, está claro que usaremos una nueva aplicación Spring Boot para implementar la lógica de gamificación. Decidimos pasar a microservicios: nuestra funcionalidad se divide en la aplicación de multiplicación (ahora la *microservicio de multiplicación)* y la aplicación de gamificación (*microservicio de gamificación).*

Además, usaremos una arquitectura impulsada por eventos aplicada a nuestros microservicios, por lo que ahora necesitamos modelar las interacciones entre estos dos contextos diferentes (multiplicación y gamificación) como eventos.

Figura 4-2 muestra la vista lógica que ilustra lo que queremos lograr en el capítulo.

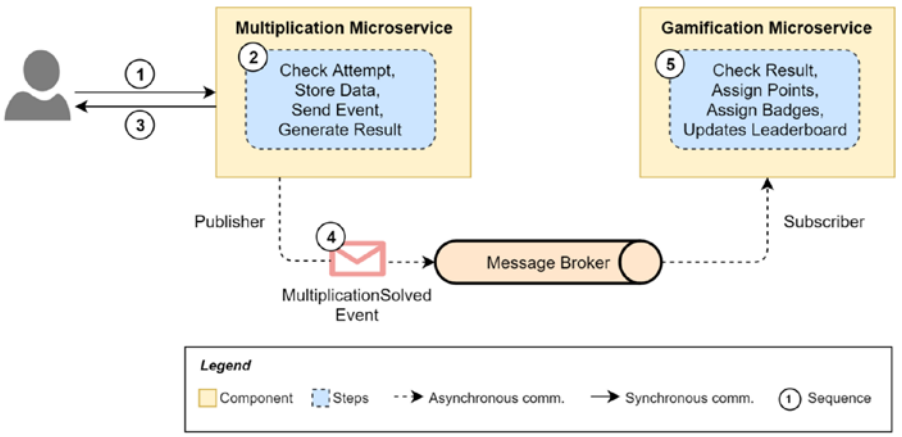

Figura 4-2. *Visión lógica de las interacciones entre los dos contextos*

Queremos que el servicio de multiplicación se comporte como está ahora, con la única diferencia de que necesitamos comunicar a las partes interesadas que una multiplicación se ha resuelto correctamente. Para hacer eso, modelamos el MultiplicationSolvedEvent, que representa esa acción comercial en el sistema.

Luego, modelaremos la nueva lógica empresarial para la gamificación en un nuevo servicio, que consumirá eventos de nuestro nuevo tipo del bus de eventos. Cuando se recibe un nuevo evento, procesa los datos que contiene y asigna puntos e insignias al usuario.

CAMBIO INCREMENTADO

tenga en cuenta que no exponemos la funcionalidad de gamificación a la interfaz de usuario en este capítulo, ya que eso significaría mucha refactorización. Queremos mantener el enfoque en los patrones de arquitectura impulsada por eventos y respetar la mentalidad incremental / ágil de este libro, por lo que los cambios necesarios para exponer esa funcionalidad se cubrirán en el próximo capítulo. Spoiler: no terminaremos nuestra nueva historia de usuario al final de este capítulo. Pero no se preocupe, llegará pronto.

Pasar por eventos con RabbitMQ y Spring AMQP

RabbitMQ es un agente de mensajes de código abierto que está muy bien integrado con Spring Boot. Es la herramienta perfecta para enviar y recibir mensajes en esta aplicación. Además, implementa AMQP (*Protocolo de cola de mensajes avanzado)* para que podamos escribir nuestro código de forma genérica, evitando el acoplamiento con la propia herramienta.

Si desea obtener detalles completos sobre AMQPModel en RabbitMQ, el tutorial oficial es un buen punto de partida, donde puede aprender más sobre los conceptos de cola, intercambio y ruta, y también aprender sobre los diferentes tipos de intercambios: *Directo, Fanout, Tema y Encabezados.* Ver https://www.rabbitmq.com/tutorials/amqp-concepts.html.

Usando RabbitMQ en su sistema

Comencemos por definir lo que necesitamos de RabbitMQ para lograr lo que definimos en nuestra vista lógica. No se preocupe por los conceptos ahora; los extenderemos a lo largo de este capítulo.

- Crearemos un *intercambio,* que es un canal al que la multiplicación enviará básicos MultiplicationSolvedEvent mensajes. Elegimos JSON para serializarlos, ya que es un formato ampliamente extendido y también legible por humanos.
- Nuestro intercambio será de tipo *tema.* Esto es para ilustrar la forma más flexible de enviar mensajes.
- Enviaremos nuestro mensaje de evento con una clave de enrutamiento llamada multiplicación resuelta.
- En el lado del suscriptor (el microservicio de gamificación), crearemos una cola y la vincularemos a nuestro intercambio de temas para recibir los mensajes que nos interesan.
- Haremos que nuestras colas sean duraderas. Al hacer esto, nos aseguramos de que incluso si el corredor (RabbitMQ) deja de funcionar, podremos procesar los eventos cuando regresen (porque los mensajes persisten).

La flexibilidad de *intercambios de temas* y *claves de enrutamiento* proviene del hecho de que varios suscriptores pueden vincular colas al mismo intercambio con diferentes claves de enrutamiento, por lo que pueden recibir un subconjunto diferente de mensajes. Piense en el ejemplo de una tienda online: un microservicio encargado de cancelar pedidos se suscribiría aorden cancelada, mientras que el responsable de enviar correos electrónicos se suscribiría a pedido.* (lo que significa lo que suceda con los pedidos: cancelar, retrasar, etc.). Como puede adivinar, diseñar intercambios, colas y claves de enrutamiento no es fácil, especialmente si desea lograr un rendimiento óptimo. Te recomiendo que pases por

el tutorial de Spring AMQP[5] en el sitio web de RabbitMQ para familiarizarse con todos estos conceptos. De todos modos, cubrimos este ejemplo práctico mientras implementamos el lado del suscriptor dentro del microservicio de gamificación.

Primavera AMQP

Interactuaremos con nuestro corredor RabbitMQ usando Spring AMQP. Todo lo que necesitamos hacer desde nuestras aplicaciones Spring Boot para comenzar a usarlo es incluir el primavera-arranque-arrancador-amqp dependencia.

Podríamos configurar nuestro intercambio y la cola directamente a través de RabbitMQ (usando la línea de comando o la interfaz de usuario), pero en su lugar lo haremos desde el código Java usando Spring AMQP. La principal ventaja es que cada servicio mantiene bajo control su configuración AMQP, sin necesidad de depender de un lugar central para mantenerla.

Sin embargo, el principal inconveniente de no iniciar RabbitMQ *totalmente configurado* es que los servicios no pueden asumir, por ejemplo, que el intercambio está ahí antes de comenzar. Nuestro microservicio de gamificación debe contener la configuración para crear el intercambio de temas, en caso de que comience antes del microservicio de multiplicación. Sin embargo, eso no es un gran problema ya que Spring AMQP no creará intercambios o colas duplicados, sino que tomará los existentes si ya están allí.

Envío de eventos de multiplicación

¡Seamos prácticos! Queremos hacer que nuestro microservicio de multiplicación existente funcione en un ecosistema impulsado por eventos. Hicimos nuestras elecciones técnicas: RabbitMQ como corredor y Spring AMQP para interactuar con él desde el código Java.

[5] https://www.rabbitmq.com/tutorials/tutorial-three-spring-amqp.html

Recuerde: nuestro objetivo es que la multiplicación envíe un MultiplicationSolvedEvent cada vez que un usuario envía un nuevo intento. Más adelante implementaremos nuestro nuevo microservicio de gamificación, que se suscribirá a ese evento y reaccionará ante él.

Configuración de RabbitMQ

Como se mencionó, necesitamos editar el pom.xml archivo para incluir la nueva dependencia Spring-boot-starter-amqp. Este iniciador contiene las dependencias para usar Spring AQMP (primavera-mensajería) y RabbitMQ (primavera-conejo). Ver listado 4-1.

Listado 4-1. pom.xml (multiplicación social v5)

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

Ahora podemos empezar a modificar nuestro código. Creamos una nueva clase llamada ConejoMQConfiguración bajo un nuevo paquete llamado configuración. Agregamos la @Configuración anotación para que Spring la use cuando configure el contexto de la aplicación para generar los beans que definiremos. Esta clase se procesará automáticamente ya que está ubicada en un paquete secundario con respecto a nuestraSolicitud clase, y nuestro @SpringBootApplication la anotación incluye @ComponentScan.

Desde el lado de nuestro editor (multiplicación), la configuración mínima que necesitamos es una TopicExchange para enviar nuestro evento. Pero también agregaremos la configuración para cambiar el formato de mensaje predeterminado a JSON; luego veremos por qué. Ver listado4-2.

Listado 4-2. RabbitMQConfiguration.java (multiplicación social v5)

```
/**
 * Configura RabbitMQ para usar eventos en nuestra aplicación.
 */
@Configuración
clase pública RabbitMQConfiguration {

    @Frijol
    público TopicExchange multiplicationExchange (@
    Value ("${multiplication.exchange}") final String
    exchangeName) {
        volver nuevo TopicExchange (exchangeName);
    }

    @Frijol
    público RabbitTemplate rabbitTemplate
    (ConnectionFactory final connectionFactory) {
        final RabbitTemplate rabbitTemplate = nuevo Plantilla
        de conejo (connectionFactory);
        rabbitTemplate.setMessageConverter (productorJackson2
        MessageConverter ());
        regreso rabbitTemplate;
    }

    @Frijol
    público Jackson2JsonMessageConverter productor
    Jackson2MessageConverter () {
        volver nuevo Jackson2JsonMessageConverter ();
    }

}
```

- Creamos el TopicExchange bean usando un nombre definido por nosotros en una propiedad que necesitamos agregar a nuestro application.properties archivo (una nueva línea multiplication.exchange = multiplication_ exchange). El nombre en sí no es importante pero ten en cuenta que necesitaremos usar el mismo cuando configuremos nuestro nuevo micro-servicio de gamificación. La @Valor la anotación y la sintaxis en el interior son la forma de inyectar un valor de propiedad en Spring Boot.
- Con el segundo y tercer método, cambiamos el mecanismo de serialización predeterminado. El último inyecta unJackson2JsonMessageConverter, que toma objetos Java y los serializa a JSON. Con el rabbitTemplate () declaración de frijol, anulamos el valor predeterminado ConejoPlantilla inyectado por Spring. Tomamos como argumento elConnectionFactory (inyectado por Spring en el contexto de la aplicación) y crear un ConejoPlantilla bean que usa nuestro conversor JSONmessage. Más tarde inyectaremos esoConejoPlantilla y utilícelo para publicar nuestro mensaje de evento.

Cambiar el método de serialización a JSON en lugar de usar el mecanismo de serialización de Java predeterminado es una buena práctica en general para varios razones:

- La serialización de mensajes en Java utiliza un encabezado (__ TypeId__) para etiquetar el nombre completo de la clase. Eso significa que necesitamos que todos los suscriptores que van a deserializar el mensaje usen el mismo nombre de clase, en el mismo paquete. Esto introduce un estrecho acoplamiento entre los servicios.

- Si queremos conectarnos en el futuro con otros servicios políglotas, no podemos confiar en una serialización de Java.
- Tratar de analizar posibles errores en el canal (colas e intercambios) es una pesadilla si no utiliza un formato legible por humanos (al menos en las primeras etapas de desarrollo).

Si desea verificar en detalle las diferencias técnicas entre estos dos mecanismos de serialización (JSON y Java Serialized Object), puede leer el artículo en *https://tpd.io/rmqjson.*

Modelando el evento

Creemos la información que será intercambiada por estos dos microservicios: el evento. Tenga en cuenta los principios de la arquitectura impulsada por eventos: un evento ocurre en el pasado y debe ser genérico (sin conocimiento de los suscriptores). Indicaremos que se ha resuelto un intento de multiplicación y quien esté suscrito es irrelevante para nuestro microservicio de multiplicación.

Creamos esta nueva clase bajo un nuevo paquete llamado evento y hacer que se implemente Serializable ya que ese es un requisito del convertidor JSONmessage. Ver listado4-3.

Listado 4-3. MultiplicationSolvedEvent.java (social-multiplication v5)

```
paquete microservices.book.multiplication.event;

importar lombok.EqualsAndHashCode;
importar lombok.Getter;
importar lombok.RequiredArgsConstructor;
importar lombok.ToString;

importar java.io.Serializable;
```

```
/**
 * Evento que modela el hecho de que una {@Enlace microservices.book.
multiplication.domain.Multiplication}
 * ha sido resuelto en el sistema. Proporciona algo de contexto
información sobre la multiplicación.
 */
@RequiredArgsConstructor
@Getter
@Encadenar
@EqualsAndHashCode
clase pública MultiplicationSolvedEvent implementos Serializable {

    privado final Long multiplicationResultAttemptId;
    privado final Long userId;
    privado booleano final correcto;

}
```

Centrémonos ahora en los contenidos. Al modelar eventos, tiene un amplio espectro de opciones con respecto a la información que coloca allí. En este caso, podríamos haber incluido todoMultiplicationResultAttempt objeto. Eso viajaría en el mensaje junto con el contenido de la referencia Usuario y Multiplicación objetos. Pero, ¿por qué harías eso? Las personas que siguen ese enfoque normalmente se basan en la idea de "por si acaso lo necesitamos".

Para ilustrar los riesgos de *eventos gordos,* Usaré otro ejemplo. Imagine que estamos recibiendo eventos cuando se actualizan los detalles del usuario y decidimos modelar el evento incluyendo los cambios realizados al usuario. Piense en el caso en el que hay varios suscriptores, uno de ellos está fallando y el corredor está enviando esos mensajes rechazados nuevamente. Ahora bien, el orden de los eventos no es la secuencia real de los cambios. No podemos estar seguros del lado del consumidor si ese cambio refleja el estado más reciente. Como una posible solución, podríamos usar una marca de tiempo en los eventos, pero luego se necesita una lógica adicional por parte del consumidor para manejar el tiempo: descartar cambios anteriores, etc.

Incluir datos en eventos que modelan cambios en un objeto mutable es arriesgado. En este caso, podría ser mejor notificar que el usuario con el identificador dado se ha actualizado y dejar que los consumidores soliciten el estado más reciente cada vez que decidan procesar su lógica.

Otro posible inconveniente de incluir demasiados datos en los eventos se puede mostrar en el siguiente ejemplo: si en el futuro incluimos un microservicio adicional (por ejemplo, un *analizador de estadísticas)* y necesita usar la marca de tiempo de los intentos, podríamos simplemente agregar la marca de tiempo a MultiplicationSolvedEvent. Pero, en ese caso, estaríamos adaptando los eventos desde el lado del editor a las necesidades de todos nuestros consumidores. No solo tendríamos un gran evento, sino también un editor inteligente que conoce demasiado sobre la lógica comercial de sus consumidores: un anti-patrón de arquitectura impulsada por eventos. En general, es más recomendable dejar que los consumidores soliciten los datos que necesitan y evitar incluirlos como parte del contenido del evento.

Volviendo a nuestro caso, tenemos la ventaja de que nuestro *intento* representa una realidad que es inmutable: no esperamos modificar los intentos una vez que son procesados por el servicio de multiplicación. Usando eso en nuestro beneficio, podemos incluir una referencia al usuario (userId) y también pasar un valor booleano que indique si el intento fue correcto o no. Esta porción de información es genérica e inmutable, y puede ahorrar algunas solicitudes REST adicionales de consumidores potenciales (que es el efecto secundario de tener *demaciado flaco* eventos).

Como puede ver, no existe un enfoque en blanco y negro, pero debe quedar claro en este punto que modelar eventos es tan importante como modelar su dominio. Piense detenidamente con cuáles necesita comenzar (no incluya muchos de ellos a la vez) y trate de mantenerlos lo más simples posible, creando contenidos pequeños y genéricos que sean lo suficientemente buenos para los suscriptores y consistentes si se reciben en una orden inesperada.

Envío del evento: patrón de despachador

El despachador de eventos (o editor de eventos) y el controlador de eventos (o suscriptor de eventos) son dos patrones comunes para la comunicación asincrónica. En lugar de que los eventos se publiquen o consuman en todas sus clases, estos puntos centralizados para la entrada / salida de eventos hacen que sus interacciones de servicio sean más fáciles de encontrar y comprender.

Por otro lado, tener todos los despachadores de eventos u oyentes en una sola clase puede terminar con una clase enorme y mucha lógica de redireccionamiento. Sin embargo, esto puede verse como una ventaja en una arquitectura de microservicios: si el EventDispatcher o el Controlador de eventos las clases se vuelven demasiado grandes, probablemente se deba a que su microservicio no lo es tanto*micro* nunca más. ¿Por qué trataría tantos eventos dentro de un solo microservicio? Debe reflexionar sobre eso e intentar identificar si es el caso que su microservicio tiene demasiadas responsabilidades. También podría ser que el microservicio realmente necesite manejar muchos eventos; entonces una buena solución es dividir los despachadores / controladores de eventos en varias clases, según la lógica empresarial.

Centrémonos primero en el microservicio de multiplicación y veamos cómo implementar el Despachador patrón allí. Más adelante en este capítulo, cubriremos la lógica del suscriptor cuando naveguemos por el código base del microservicio de gamificación. Ver listado4-4.

Listado 4-4. EventDispatcher.java (multiplicación social v5)

```
paquete microservices.book.multiplication.event;

importar org.springframework.amqp.rabbit.core.RabbitTemplate;
importar org.springframework.beans.factory.annotation.Autowired;
importar org.springframework.beans.factory.annotation.Value;
importar org.springframework.stereotype.Component;

/**
 * Maneja la comunicación con Event Bus.
 */
```

```
@Componente
clase pública EventDispatcher {

    privado RabbitTemplate rabbitTemplate;

    // El intercambio que se utilizará para enviar cualquier cosa relacionada con la
    multiplicación
    privado String multiplicationExchange;

    // La clave de enrutamiento que se utilizará para enviar este evento en particular
    privado String multiplicationSolvedRoutingKey;

    @Autowired
    EventDispatcher (final RabbitTemplate rabbitTemplate,
                     @Value ("$ {multiplication.exchange}") final
                     String multiplicationExchange,
                     @Value ("$ {multiplication.solved.key}")
                     final String multiplicationSolved
                     RoutingKey) {
        esto.RabbitTemplate = RabbitTemplate;
        esto.multiplicationExchange = multiplicationExchange;
        esto.multiplicationSolvedRoutingKey = multiplicación
        SolvedRoutingKey;
    }

    público void send (final MultiplicationSolvedEvent
    multiplicationSolvedEvent) {
        rabbitTemplate.convertAndSend (
                multiplicationExchange,
                multiplicationSolvedRoutingKey,
                multiplicationSolvedEvent);
    }
}
```

La clase recibe el ConejoPlantilla del contexto de la aplicación de Spring, junto con el nombre del intercambio y la clave de enrutamiento de las propiedades de la aplicación. Luego usamos la plantilla para convertAndSend nuestro objeto (en este caso, convertido a JSON según la configuración proporcionada). Además, nuestroMultiplicationSolvedEvent usará la clave de enrutamiento multiplicación resuelta. Recuerde que este evento será capturado por la cola del consumidor usando el patrón de enrutamiento multiplicación.*. Cubriremos eso más adelante en el capítulo. Ver listado4-5.

Listado 4-5. Application.Properties: Adición de valores RabbitMQ (social-multiplication v5)

```
# . . . (otras propiedades)
# # Configuración de RabbitMQ
multiplication.exchange = multiplication_exchange
multiplication.solved.key = multiplication.solved
```

La única parte que nos falta en nuestro código es enviar el evento desde nuestra lógica empresarial. Como se introdujo en subsecciones anteriores, lo haremos para cada intento que recibamos de los usuarios. El cambio es muy sencillo: solo inyectamos elEventDispatcher y utilícelo para enviar un nuevo MultiplicationSolvedEvent.

Es importante señalar que Spring AMQP admite transacciones. Dado que tenemos nuestro método anotado con @Transaccional el evento no se enviará en caso de una excepción incluso si colocamos nuestro eventDispatcher.send () al comienzo del método y la excepción ocurrió después. Para una mejor legibilidad, coloque los remitentes de eventos al final de la lógica, o al menos después de que ocurra la acción. Ver listado4-6.

Listado 4-6. MultiplicationServiceImpl.java: Adición de lógica de eventos (social-multiplication v5)

```
@Servicio
clase MultiplicationServiceImpl implementos
MultiplicationService {

    privado RandomGeneratorService randomGeneratorService;
    privado MultiplicationResultAttemptRepository
    intentoRepository;
    privado UserRepository userRepository;
    privado EventDispatcher eventDispatcher;

    @Autowired
    público MultiplicationServiceImpl (servicio final de
    RandomGenerator randomGeneratorService,
                                        Final MultiplicationResult
                                        AttemptRepository intento
                                        Repository,
                                        UserRepository final
                                        userRepository,
                                        EventDispatcher final
                                        eventDispatcher) {
        esto.randomGeneratorService = randomGeneratorService;
        esto.intentoRepositorio = intentoRepositorio;
        esto.userRepository = userRepository;
        esto.eventDispatcher = eventDispatcher;
    }

    @Anular
    público Multiplicación createRandomMultiplication () {
        int factorA = randomGeneratorService.generateRandom
        Factor ();
```

```
        int factorB = randomGeneratorService.
        generateRandomFactor ();
        volver nuevo Multiplicación (factorA, factorB);
    }

    @Transaccional
    @Anular
    público boolean checkAttempt (intento de
    MultiplicationResultAttempt final) {
        // Verifica si el usuario ya existe para ese alias
        <User> opcional user = userRepository.
        findByAlias   (intento.getUser (). getAlias   ());
        // Evita intentos de 'pirateo'
        Assert.isTrue (! Intent.isCorrect (), "¡¡No puedes enviar un
        intento marcado como correcto !!");

        // Comprueba si el intento es correcto
        booleano isCorrect = intento.getResultAttempt () ==
                             intent.getMultiplication ().
                             getFactorA () *
                             intent.getMultiplication ().
                             getFactorB ();

        MultiplicationResultAttempt checkAttempt = nuevo
        MultiplicationResultAttempt (
                user.orElse (intento.getUser ()),
                intento.getMultiplicación (),
                intento.getResultAttempt (),
                isCorrect
        );
```

```
        // Almacena el intento
        intentRepository.save (comprobado intento);

        // Comunica el resultado a través de un evento
        eventDispatcher.send (
                nuevo MultiplicationSolvedEvent (checkAttempt.
                GetId (),
                        checkAttempt.getUser (). getId (),
                        checkAttempt.isCorrect ())
        );

        regreso es correcto;
    }

    @Anular
    público List <MultiplicationResultAttempt> getStatsForUser
    (String userAlias) {
        regreso intentRepository.findTop5ByUserAliasOrderBy
        IdDesc (userAlias);
    }
}
```

EJERCICIO

Actualizamos la lógica para incluir el EventDispatcher y nuestras pruebas seguirán aprobadas, pero ya no están completas. queremos usar Mockito para verificar que, dentro de nuestra lógica, se envía un evento correcto. Actualice la prueba para incluir esa afirmación. Si necesita ayuda, puede consultar la solución en el MultiplicationServiceImplTest clase (multiplicación social v5).

Una mirada más profunda al nuevo microservicio de gamificación

Descripción general del código

Dentro de esta sección, cubriremos la implementación de nuestro nuevo microservicio de Gamificación y veremos cómo recibir eventos del microservicio de multiplicación existente, que acabamos de modificar.

Dado que esta es la segunda aplicación Spring Boot que creamos, no revisaremos todos los detalles. En cambio, nos centraremos en las partes más interesantes. En cualquier caso, los bloques de Ejercicio lo guiarán a través del resto de los cambios necesarios.

CÓDIGO FUENTE DISPONIBLE CON EL LIBRO: V5

Puede encontrar todo el código relacionado con este capítulo (tanto microservicios de multiplicación como de gamificación) dentro del v5 repositorio en github (el social-multiplicación y gamificación proyectos) en: https: // github. com / microservices-practico.

Ejercicio

antes de comenzar a codificar el nuevo microservicio, debe crear un proyecto para él. como hiciste antes, puedes usar Spring initializr (http://start.spring.io). Llamada la nueva aplicación gamificación y usa el microservices.book. gamificación paquete. Además de la dependencia web, también debe incluir Lombok, h2 y aMQp.

después de extraer el proyecto, abra el pom.xml y alinee las versiones de dependencia con las de la aplicación de multiplicación. De esta forma, evita un comportamiento potencialmente diferente al descrito en este libro.

El dominio

Primero, tomemos un tiempo para comprender el modelo de dominio de gamificación. Consiste en lo siguiente:

- Tanteador: Modela un conjunto incremental de puntos que obtiene un usuario determinado en un momento determinado.
- Insignia: Una enumeración de todas las posibles insignias del juego.
- BadgeCard: Representa una insignia vinculada a un determinado usuario, ganada en un momento determinado.
- LeaderBoardRow: Una posición en la tabla de clasificación que es la puntuación total junto con el usuario.
- GameStats: Puntuación e insignias para un usuario determinado. Se puede usar para una iteración de juego determinada (resultado de un intento) o para una colección de intentos (puntuación e insignias agregadas).

Las tarjetas (puntuación e insignia) contienen el momento en el que se obtuvieron. El resultado de una iteración del juego puede contener uno o más ScoreCards y uno o mas BadgeCards. Ver figura 4-3.

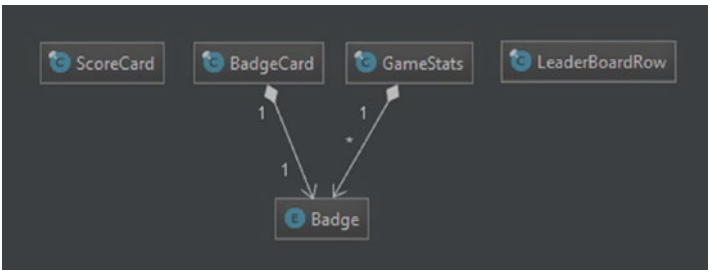

Figura 4-3. *El modelo de dominio de la gamificación*

Echemos un vistazo rápido a la base de código para estas clases de dominio. Ver listados4-7 mediante 4-11.

Listado 4-7. Badge.java (gamificación v5)

```
paquete microservices.book.gamification.domain;

/**
 * Enumeración con los diferentes tipos de Badges que puede ganar un
   usuario.
 */
enumeración pública Insignia {

    // Insignias en función de la puntuación
    BRONZE_MULTIPLICATOR,
    SILVER_MULTIPLICATOR,
    GOLD_MULTIPLICATOR,

    // Otras insignias ganadas para diferentes condiciones
    PRIMER INTENTO,
    PRIMERA_GANADA

}
```

Listado 4-8. BadgeCard.java (gamificación v5)

```
/**
 * Esta clase vincula una insignia a un usuario. Contiene también
   una marca de tiempo con el momento en que el usuario la obtuvo.
 */
@RequiredArgsConstructor
@Getter
@Encadenar
@EqualsAndHashCode
@Entidad
```

```
público final clase BadgeCard {

    @Identificación
    @GeneratedValue
    @Column (nombre = "BADGE_ID")
    privado final Long badgeId;

    privado final Long userId;
    privado placa de tiempo larga final;
    privado insignia final de la insignia;

    // Constructor vacío para JSON / JPA
    público BadgeCard () {
        esto (nulo, nulo, 0, nulo);
    }

    público BadgeCard (ID de usuario largo final, insignia de insignia final) {
        esto (nulo, userId, System.currentTimeMillis (), insignia);
    }

}
```

Listado 4-9. ScoreCard.java (gamificación v5)

```
/ **
  * Esta clase representa la puntuación vinculada a un intento en el juego,
  * con un usuario asociado y la marca de tiempo en la que la puntuación
  * Esta registrado.
  * /
@RequiredArgsConstructor
@Getter
@Encadenar
@EqualsAndHashCode
@Entidad
```

```
público final clase ScoreCard {

    // La puntuación predeterminada asignada a esta tarjeta, si no
    se especifica.
    público estático final int DEFAULT_SCORE = 10;

    @Identificación
    @GeneratedValue
    @Column (nombre = "CARD_ID")
    privado final Long cardId;

    @Column (nombre = "USER_ID")
    privado final Long userId;

    @Column (nombre = "ATTEMPT_ID")
    privado final Long intentId;

    @Column (nombre = "SCORE_TS")
    privado scoreTimestamp largo final;

    @Columna (nombre = "PUNTUACIÓN")
    privado puntuación int final;

    // Constructor vacío para JSON / JPA
    público ScoreCard () {
        esto (nulo, nulo, nulo, 0, 0);
    }

    público ScoreCard (ID de usuario largo final, ID de intento largo final) {
        esto (nulo, userId, intentId, System.current
        TimeMillis (), DEFAULT_SCORE);
    }

}
```

Listado 4-10. GameStats.java (gamificación v5)

```
/**
 * Este objeto contiene el resultado de una o varias iteraciones del juego.
 * Puede contener cualquier combinación de {@Enlace ScoreCard} objetos y {@Enlace BadgeCard} objetos.
 *
 * Puede usarse como un delta (como una iteración de un solo juego) o para representar la cantidad total de puntajes / insignias.
 */
@RequiredArgsConstructor
@Getter
@Encadenar
@EqualsAndHashCode
público final clase GameStats {

    privado final Long userId;
    privado puntuación int final;
    privado Insignias <Badge> de la lista final;

    // Constructor vacío para JSON / JPA
    público GameStats () {
        esto.userId = 0L;
        esto.puntuación = 0;
        esto.insignias = nuevo ArrayList <> ();
    }

    /**
     * Método de fábrica para construir una instancia vacía (cero puntos y sin insignias)
     * @param userId la identificación del usuario
     * @regreso a {@Enlace Objeto GameStats} con puntuación cero y sin insignias
```

```
     */
    público static GameStats emptyStats (final Long userId) {
        volver nuevo GameStats (userId, 0, Colecciones.
        EmptyList ());
    }

    /**
     * @regreso una vista no modificable de la lista de tarjetas de insignia
     */
    público List <Insignia> getBadges () {
        regreso Collections.unmodifiableList (insignias);
    }
}
```

Listado 4-11. LeaderBoardRow.java (gamificación v5)

```
/**
 * Representa una línea en nuestra tabla de clasificación: vincula a un usuario a un
puntaje total.
 */
@RequiredArgsConstructor
@Getter
@Encadenar
@EqualsAndHashCode
público final clase LeaderBoardRow {

    privado final Long userId;
    privado final Long totalScore;

    // Constructor vacío para JSON / JPA
    público LeaderBoardRow () {
        esto(0L, 0L);
    }
}
```

Los datos

Estrictamente hablando, lo que debemos conservar de nuestro modelo es la puntuación total de un usuario y las insignias vinculadas. En lugar de acumular la puntuación en un solo objeto / fila, almacenaremos las tarjetas y las agregaremos al consultar la puntuación total de un usuario. De esta forma, mantenemos la trazabilidad de la puntuación del usuario a lo largo del tiempo.

Por lo tanto, nuestros datos persistentes estarán compuestos por dos tablas, que son representaciones directas de Tanteador y BadgeCard clases.

Primero, veamos nuestro repositorio para BadgeCard objetos. Nada nuevo allí, usando el CrudRepository fromSpring Data y un método de consulta que, mediante convenciones de nomenclatura, se procesará como una consulta para obtener insignias para un usuario determinado, la más reciente primero. Ver listado4-12.

Listado 4-12. BadgeCardRepository.java (gamificación v5)

```
paquete microservices.book.gamification.repository;

importar microservices.book.gamification.domain.BadgeCard;
importar org.springframework.data.repository.CrudRepository;

importar java.util.List;

/**
 * Maneja operaciones de datos con BadgeCards
 */
interfaz pública BadgeCardRepository se extiende
CrudRepository <BadgeCard, Long> {

    /**
     * Recupera todas las BadgeCards de un usuario determinado.
     * @param userId la identificación del usuario para buscar BadgeCards
     * @regreso la lista de BadgeCards, ordenadas por las más recientes.
     */
```