Las pruebas están hechas y listas, así que regresemos y construyamos el material real. Como probablemente imaginó, esta es la implementación real del método que proporciona la lógica real. Ver listado3-14.

Listado 3-14. MultiplicationServiceImpl.java NewMethod (social-multiplication v3)

```
@Anular
público boolean checkAttempt (final MultiplicationResultAttempt
resultAttempt) {
    regreso resultAttempt.getResultAttempt () ==
            resultAttempt.getMultiplication (). getFactorA () *
            resultAttempt.getMultiplication (). getFactorB ();
}
```

En el v3 carpeta de código, también puede encontrar la RandomGeneratorService interfaz y la implementación correspondiente, como se trató en el capítulo anterior.

La capa de presentación (API REST)

Ahora que tenemos nuestras entidades de dominio y nuestra lógica comercial simple en su lugar, expondremos las interacciones admitidas a través de una API REST para que un cliente web o cualquier otra aplicación pueda interactuar con nuestra funcionalidad. REST es un estándar conocido para servicios web en la industria debido a su simplicidad: son solo interfaces básicas sobre HTTP.

Es importante tener en cuenta aquí que no necesitamos estrictamente una capa REST para nuestra aplicación, ya que podríamos usar SpringMVC[2] y luego devolver los nombres de vista y renderizar nuestros modelos directamente en HTML o cualquier otra implementación de capa de vista. Pero luego necesitaríamos diseñar vistas en nuestro código base. Eso dificultaría el cambio de la tecnología de la interfaz de usuario (por ejemplo, migrar a AngularJS o tener una aplicación móvil). Además,

[2] https://docs.spring.io/spring/docs/current/spring-frameworkreference/html/mvc.html

con REST, proporcionamos una interfaz que no requiere una interfaz de usuario, solo HTTP básico. Además, se puede acceder a esta misma API desde un servicio de backend diferente en el futuro (por ejemplo, si otra aplicación desea obtener una multiplicación aleatoria).

Spring tiene un buen soporte para construir una API REST de una manera muy rápida, pero tenga en cuenta que debe seguir algunas convenciones para las URL y los verbos HTTP que se han convertido en el estándar de facto (consulte *http://tpd.io/rest-methods*). En este libro, usaremos estas asignaciones estándar de acciones a URL y verbos HTTP.

¿Cuáles son las interfaces que queremos exponer para esta aplicación? Podemos obtenerlos de los requisitos:

- Queremos que los usuarios resuelvan una multiplicación, por lo que queremos leer una multiplicación aleatoria de complejidad media como consumidores de la API REST.
- Para resolver la multiplicación queremos enviar un resultado para una multiplicación dada, y como queremos saber quién la está resolviendo, queremos enviarlo junto con el alias del usuario.

Hasta ahora eso es lo que necesitamos: una operación de lectura y una acción de envío. Después de haber aclarado las interacciones, podemos diseñar nuestra API REST, teniendo en cuenta los estándares:

- GET / multiplicaciones / aleatorio devolverá la multiplicación aleatoria.
- POST / resultados / será nuestro punto final para enviar resultados.
- GET / results? User = [user_alias] será nuestra forma de recuperar los resultados de un usuario en particular.

Como puede ver, diseñamos la API en contextos de dos dominios para los puntos finales: multiplicaciones y resultados. Ésta es una buena práctica. No intente poner todo en el mismo contexto y controlador. Es mejor separar las interfaces según las entidades comerciales con las que se relacionan. Crearemos dos diferentesControlador clases.

El controlador de multiplicación

Sigamos TDD nuevamente y escribamos la prueba unitaria como de costumbre. Primero, necesitamos una implementación vacía de la clase del controlador para compilar el código, como se muestra en Listado3-15.

Listado 3-15. MultiplicationController.java Versión inicial (social-multiplication v3)

```
paquete microservices.book.multiplication.controller;

importar microservices.book.multiplication.service.
        MultiplicationService;
importar org.springframework.beans.factory.annotation.Autowired;
importar org.springframework.web.bind.annotation.RestController;

@RestController
clase pública MultiplicationController {

    privado final MultiplicationService multiplicationService;

    @Autowired
    público MultiplicationController (final MultiplicationService
            multiplicationService) {
        esto.multiplicationService = multiplicationService;
    }

}
```

Ahora construimos la prueba unitaria para comprobar que el Controlador de multiplicación devolverá una multiplicación aleatoria al realizar una OBTENER a la ubicación / multiplicación / aleatorio, como se muestra en el listado 3-16.

Listado 3-16. MultiplicationControllerTest.java (multiplicación social v3)

```
// Importar declaraciones ...

@RunWith (SpringRunner.class) @WebMvcTest
(MultiplicationController.class)
clase pública MultiplicationControllerTest {

    @MockBean
    privado MultiplicationService multiplicationService;

    @Autowired
    privado MockMvc mvc;

    // Este objeto se inicializará mágicamente mediante el método
    initFields a continuación.
    privado JacksonTester <Multiplication> json;

    @Antes
    público configuración vacía () {
        JacksonTester.initFields (esto, nuevo ObjectMapper ());
    }

    @Prueba
    público void getRandomMultiplicationTest () lanza Excepción{
        // dado
        dado (multiplicationService.createRandomMultiplication ())
                . willReturn (nuevo Multiplicación (70, 20));
```

```
        // Cuándo
        MockHttpServletResponse respuesta = mvc.perform (
                obtener ("/ multiplicaciones / aleatorio")
                        . aceptar (MediaType.APPLICATION_JSON))
                . andReturn (). getResponse ();

        // luego
        afirmarQue (respuesta.getStatus ()). isEqualTo (HttpStatus.
        OK.value ());
        assertThat (response.getContentAsString ())
                . isEqualTo (json.write (nuevo Multiplicación
                (70, 20)). GetJson ());
    }

}
```

Veamos los principales cambios introducidos en esta prueba:

1. Es una @WebMvcTest, por lo que inicializará el contexto de la aplicación web de Spring. Sin embargo, solo cargará la configuración relacionada con la capa MVC (controladores), a diferencia de @SpringBootTest, que carga toda la configuración. Al usar esta anotación, también obtenemos elMockMvc bean cargado.
2. ¿Te acuerdas de @MockBean del capítulo anterior? Lo usamos aquí nuevamente en lugar de @Burlarse de ya que necesitamos decirle a Spring que no inyecte el bean real (MultiplicationServiceImpl) sino un objeto simulado, que configuramos más tarde con dado() para devolver lo esperado Multiplicación. Lo que estamos haciendo aquí es aislar capas: solo queremos probar el controlador, no el servicio.

3. El JacksonTester object proporcionará métodos útiles para verificar el contenido JSON. Se puede configurar y conectar automáticamente cuando se usa @JsonTest anotación, pero como estamos escribiendo una @WebMvcTest, necesitamos configurarlo manualmente (dentro del método anotado con Antes).

BUENAS PRÁCTICAS: @WEBMVCTEST Y @SPRINGBOOTTEST

Las pruebas MVC están destinadas a cubrir solo la parte del controlador de su aplicación. Las solicitudes y respuestas http se simulan para que no se creen conexiones reales. por otro lado, cuando usas @SpringBootTest, Se carga toda la configuración para el contexto de la aplicación web y las conexiones pasan por el servidor web real. en ese caso, no usa el MockMvc frijol pero un estándar RestTemplate en su lugar (o la nueva alternativa TestRestTemplate).

Entonces, ¿cuándo deberíamos elegir uno u otro? @WebMvcTest está destinado a probar unitariamente el controlador desde el lado del servidor. @SpringBootTest, por otro lado, debe usarse para *pruebas de integración,* cuando desee interactuar con la aplicación desde el lado del cliente.

eso no significa que no puedas usar simulacros con @SpringBootTest; si está escribiendo una prueba de integración, aún podría ser necesario. en cualquier caso, es mejor no usarlo solo para una simple prueba unitaria del controlador.

Si ejecuta la prueba ahora, obtendrá un código de estado 404. Eso no es sorprendente, porque la implementación de la lógica aún no está allí. Listado 3-17 muestra cómo construir el controlador.

Listado 3-17. MultiplicationController.java Adición de lógica (socialmultiplication v3)

```
/**
 * Esta clase implementa una API REST para nuestra multiplicación.
solicitud.
 */
@RestController
@RequestMapping ("/ multiplicaciones")
final clase MultiplicationController {

    privado final MultiplicationService multiplicationService;

    @Autowired
    público MultiplicationController (final MultiplicationService
            multiplicationService) {
        esto.multiplicationService = multiplicationService;
    }

    @GetMapping ("/ aleatorio")
    Multiplicación getRandomMultiplication () {
        regreso multiplicationService.createRandomMultiplication ();
    }

}
```

Es una clase muy simple (¡incluso más corta que la prueba!). Eso es porque con algunas anotaciones de Spring y algunas líneas de código, puede obtener todo lo que necesita:

1. La @RestController anotación especifica que la clase es un controlador y todas las @RequestMapping (o @GetMapping en nuestro caso) annotatedmethods devolverá el contenido en el cuerpo de la respuesta. Si usamos una @ simpleControlador anotación en su lugar, necesitamos

anotar nuestra clase (o cada método correspondiente) con @ ResponseBody. Por lo tanto, @RestController es un *atajo* anotación.

2. La @RequestMapping La anotación a nivel de clase establece el contexto raíz para todos los métodos (en nuestro caso, *multiplicaciones).*

3. Otra anotación de acceso directo es @GetMapping, y es equivalente a usar @RequestMapping (método = RequestMethod.GET). Entonces, el punto final resultante realizará una OBTENER operación a la URL compuesta por el contexto especificado de la clase más la asignación de solicitud del método, lo que da como resultado / multiplicaciones / aleatorio.

Así de fácil podemos construir una API REST con Spring. Si lo ejecuta ahora, la prueba pasará como se esperaba.

Algunos de ustedes pueden pensar que escribir pruebas unitarias para la capa del controlador *no es realmente importante,* debido a la simplicidad de estas clases. Ese es un buen punto, de hecho. Sin embargo, si escribe pruebas unitarias para esta capa, se asegura de una doble verificación de los cambios en el contrato de la API, lo que se vuelve realmente útil, especialmente en un entorno de microservicios donde varios equipos pueden administrar diferentes servicios. Si accidentalmente cambia / multiplicaciones / aleatorio por /multiplicación / aleatorio, su prueba fallará, al igual que si cambia los verbos HTTP o los datos que manejan los puntos finales. Los consumidores de la API REST apreciarán que debe pensarlo dos veces antes de cambiar el contrato de la API.

El controlador de resultados

Este controlador verificará los resultados publicados por nuestros usuarios y les dirá si son correctos o no. Podríamos elegir entre muchas formas diferentes de devolver la respuesta, pero un buen enfoque es crear una clase básica que

ajustará el resultado, que ahora solo consta de un campo booleano: correcto.

Tenga en cuenta que si devuelve un booleano directamente en la respuesta en lugar de incluirlo en una clase, el serializador JSON predeterminado no funcionará.[3]

La primera versión vacía del controlador se muestra en el Listado 3-18.

Listado 3-18. MultiplicationResultAttemptController.java Versión inicial (social-multiplication v3)

```
@RestController
@RequestMapping ("/ results") final clase
MultiplicationResultAttemptController {

    privado final MultiplicationService multiplicationService;

    @Autowired
    MultiplicationResultAttemptController (final
    MultiplicationService multiplicationService) {
        esto.multiplicationService = multiplicationService;
    }

    // Aquí implementaremos nuestro POST más tarde

    @RequiredArgsConstructor
    @NoArgsConstructor (fuerza = cierto)
    @Adquiridor
    privado final estática clase ResultResponse {
        privado booleano final correcto;
    }
}
```

[3] https://stackoverflow.com/questions/33185217/is-it-possible-inspring-mvc-4-return-boolean-as-json

Habiendo modelado el ResultResponse, escribamos nuestra prueba unitaria para MultiplicationResultAttemptController. Incluiremos los escenarios de envío de un intento correcto y un intento incorrecto. Dado que no hay mapeo para elCORREO solicitud que estamos realizando, las pruebas fallarán arrojando un 404 predecible (no encontrado), como se muestra en el Listado 3-19.

Listado 3-19. MultiplicationResultAttemptControllerTest.java (social-multiplication v3)

```
@RunWith (SpringRunner.class) @WebMvcTest
(MultiplicationResultAttemptController.class)
clase pública MultiplicationResultAttemptControllerTest {

    @MockBean
    privado MultiplicationService multiplicationService;

    @Autowired
    privado MockMvc mvc;

    // Este objeto se inicializará mágicamente mediante el método
    initFields a continuación.
    privado JacksonTester <MultiplicationResultAttempt> jsonResult;
    privado JacksonTester <ResultResponse> jsonResponse;

    @Antes
    público configuración vacía () {
        JacksonTester.initFields (esto, nuevo ObjectMapper ());
    }

    @Prueba
    público void postResultReturnCorrect () lanza Excepción {
        genericParameterizedTest (cierto);
    }

    @Prueba
```

```
    público void postResultReturnNotCorrect () lanza Excepción {
        genericParameterizedTest (falso);
    }

    void genericParameterizedTest (final booleano correcto) lanza
    Excepción {
        // dado (recuerde que no estamos probando aquí el servicio
        en sí)
        dado (multiplicationService
                . checkAttempt (cualquiera (clase
                MultiplicationResultAttempt)))
                . willReturn (correcto);
        Usuario usuario = nuevo Usuario ("john"); Multiplicación
        multiplicación =nuevo Multiplicación (50, 70);
        MultiplicationResultAttempt intento =nuevo
        MultiplicationResultAttempt (
                usuario, multiplicación, 3500);

        // Cuándo
        MockHttpServletResponse respuesta = mvc.perform (
                publicación ("/ resultados"). contentType (MediaType.
                APPLICATION_JSON)
                        . contenido (jsonResult.write (intento).
                        getJson ()))
                . andReturn (). getResponse ();

        // luego
        afirmarQue (respuesta.getStatus ()). isEqualTo (HttpStatus.
        OK.value ());
        assertThat (response.getContentAsString ()). isEqualTo (
                jsonResponse.write (nuevo
                ResultResponse (correcto)). GetJson ());
    }
}
```

Creamos un método de conveniencia genericParameterizedTest solo para extraer el comportamiento común para probar que, cuando el servicio considere que el resultado es correcto, recibiremos cierto como respuesta y falso de lo contrario. Como se mencionó, el propósito principal de esta prueba es verificar la API.

A continuación, podemos implementar el CORREO mapeo en el MultiplicationResultAttemptController clase para recibir nuevos intentos de los usuarios, como se muestra en el Listado 3-20.

Listado 3-20. MultiplicationResultAttemptController.java Adición del método POSTM (social-multiplication v3)

```
@PostMapping
ResponseEntity <ResultResponse> postResult (@RequestBody
MultiplicationResultAttempt multiplicationResultAttempt) {
    regreso ResponseEntity.ok (
            nuevo ResultResponse (multiplicationService
                    . checkAttempt (multiplicationResultAttempt)));
}
```

Es bastante sencillo: @PostMapping la anotación hace algo similar a @GetMapping, pero en este caso manejando un CORREO pedido. Y, como queremos recibir los datos del intento como parte del cuerpo de la solicitud, necesitamos anotar el argumento del método con @RequestBody.
En este caso, los desarrolladores de Spring decidieron no inferir este incluso cuando usaban @RestController, ya que las solicitudes también pueden venir con parámetros y luego esa suposición haría las cosas más difíciles (la magia tiene un límite).

Ahora podemos ejecutar la prueba de forma segura con un resultado satisfactorio.

¡Oye, acabamos de terminar con la parte de backend para nuestro primer requisito comercial! Ahora es el momento de jugar con una interfaz de usuario básica.

El Frontend (cliente web)

Dado que ha terminado su primera API REST, está listo para construir una interfaz de usuario básica sobre ella. Proporcionará una interfaz fácil de usar para la aplicación (a los humanos no les gusta interactuar con las API REST en general). Hay muchas opciones, pero teniendo en cuenta la simplicidad de la aplicación que estamos construyendo, usaremos solo HTML y jQuery para la comunicación con los servicios web REST.

Podríamos separar esta parte en un nuevo proyecto ya que no tenemos ninguna dependencia gracias a nuestra API REST. Sin embargo, para empezar, lo haremos dentro del mismo proyecto, ya que entonces podemos usar el mismo servidor Tomcat incorporado que usa Spring Boot para servir nuestro contenido estático. Cubriremos esta decisión con más detalles al comienzo del próximo capítulo, explicando por qué es una buena idea seguir esta *enfoque de mini-monolito primero.*

ALERTA DE SPOILER: NO ES UN LIBRO DE FRENTE

Verá en el código JavaScript y htMl que son extremadamente básicos. la razón es que este libro se centra principalmente en los servicios de backend y en cómo conectar todo. por otro lado, no podemos simplemente omitir la interfaz de usuario, ya que este libro no sería tan práctico en ese momento, y necesitamos entregar nuestra historia de usuario con una pantalla para interactuar. la ventaja de tener nuestra api reSt en su lugar es que podemos cambiar la interfaz en cualquier momento sin afectar ninguna otra funcionalidad: ¡es bueno que tengamos un acoplamiento suelto!

Como sin duda habrá aprendido en este libro, es una buena idea comenzar de manera simple y construir a partir de ahí. En este caso, comenzaremos con un básicoindex.html archivo, una cantidad mínima de estilos en styles.css, y algunos comportamientos para la página web implementada en JavaScript usando jQuery: multiplication-client.js. Colocaremos todos estos archivos en el estático carpeta dentro principal / recursos,

creado la primera vez cuando generamos el proyecto usando Spring Initializr. Recuerda: la carpetav3 del código incluido con el libro también contiene el contenido completo de estos archivos. Veámoslos uno por uno y agreguemos algunos comentarios. Ver listado3-21.

Listado 3-21. index.html (multiplicación social v3)

```
<! 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 = "user-alias"> <br>
            <entrada tipo = "enviar" valor = "Verificar">
        </form>
    </p>
```

```
        <h2> <intervalo class = "mensaje-resultado"> </intervalo> </h2>
</div>
</body>
</html>
```

Listado 3-21 es una página de destino simple. Importamos el archivo JavaScript y los estilos (ver Listado3-22), e incluimos la referencia a la biblioteca jQuery (ver Listado 3-23). Luego tenemos el texto en el que mostramos el desafío a los usuarios, y junto a él, el formulario desde el cual los usuarios pueden completar su alias y enviar sus intentos para resolver la multiplicación. ¡Recuerde que el usuario debe realizar la operación mentalmente!

Listado 3-22. styles.css (multiplicación social v3)

```
html, cuerpo {
    altura: 100%;
}

html {
    monitor: mesa;
    margen: auto;
}

cuerpo {
    monitor: celda de tabla;
    alineación vertical: medio;
}
```

Seguramente no es el CSS más elegante del mundo: solo estamos asegurándonos de que los contenidos se muestren en el centro de la página, para evitar la sensación de vacío y centrar la atención incluso en pantallas grandes. Lo mejoraremos en los próximos capítulos.

Listado 3-23. multiplication-client.js (social-multiplication v3)

```
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);
    });
}

$ (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: '/ resultados',
            tipo: 'POST',
            datos: JSON.stringify (datos),
            contentType: "application / json; charset = utf-8",
            dataType: "json",
            éxito: función(resultado){
                Si(result.correct) {
                    $ ('. result-message'). empty (). append ("¡El
                resultado es correcto! ¡Felicitaciones!"); }
                demás {
                    $ ('. result-message'). empty (). append ("¡Vaya,
                    eso no es correcto! ¡Pero sigue intentándolo!");
                }
            }
        });

        updateMultiplication ();
    });
});
```

Lo que hacemos aquí con jQuery son dos cosas importantes:

- Cuando se carga el contenido, realizamos una llamada a la API REST para obtener una multiplicación aleatoria. Luego mostramos los factores en los marcadores de posición usando la clase para ubicarlos.

- Registramos un oyente para el enviar event en nuestro formulario para interceptarlo y evitar que realice la operación predeterminada. Luego obtenemos los datos del formulario, publicamos los datos en la API para verificar el intento resultante y luego mostramos un mensaje amigable con el resultado a los usuarios.

Como se mencionó, no es la aplicación web más agradable de la historia, pero funciona para el primer producto que queremos lograr.

¡LO HICIMOS! LA HISTORIA DE USUARIO 1 HA TERMINADO

¡Acabamos de terminar nuestra primera historia de usuario! Implementamos soluciones para todos los requerimientos. y lo hicimos usando tDD, una api reSt adecuada y con un diseño de tres niveles: ¡esta aplicación sigue buenos estándares y está lista para ser extendida!

Jugando con la aplicación (Parte I)

Ahora puede ejecutar la aplicación usando mvnw spring-boot: ejecutar. También puede empaquetarlo y enviarlo a su gente de negocios para que jueguen con él (si tienen un entorno de ejecución de Java instalado). Todo lo que necesita es Maven y Java (asegúrese de cambiar el nombre del archivo JAR si está usando su propio control de versiones). Ver listado3-24.

Listado 3-24. Consola: empaquetado y ejecución de la aplicación (social-multiplication v3)

```
paquete $ mvnw
. . .
$ cd ./target
$ java -jar social-multiplication-v3-0.3.0-SNAPSHOT.jar
. . .
INFO 12484 --- [SocialMultiplicationApplication
SocialMultiplicationApplication en 3.171 segundos (JVM
ejecutándose para 3.77)
```

Para jugar con él, puede navegar con un navegador a localhost: 8080 / index.html. Luego, puede jugar con las operaciones y ejercitar un poco su mente: verá diferentes mensajes cuando apruebe o suspenda la operación. Figura3-2 muestra cómo se ve ahora.

Figura 3-2. *La pantalla de entrada de la aplicación*

Nuevos requisitos para la persistencia de datos

Hasta ahora hemos diseñado e implementado un servicio que no mantiene ningún estado: no hay base de datos, almacenamiento de archivos, etc. Nos falta una de las capas comúnmente presentes en muchas aplicaciones de software: *la capa de datos.*

Debido a que tenemos suerte y tenemos mucho trabajo por hacer, nuestros usuarios comerciales vienen con este nuevo requisito en forma de historia de usuario.

HISTORIA DE USUARIO 2

como usuario de la aplicación, quiero que me muestre mis últimos intentos, para poder ver lo bien o mal que estoy haciendo con el tiempo.

Necesitamos algo de almacenamiento de datos para esta solicitud, ya que necesitamos realizar un seguimiento de los intentos del usuario. Daremos varios pasos para cumplir con este requisito:

- Almacene todas las instancias del MultiplicationResultAttempt clase. De esa forma, podemos extraerlos más tarde.
- Exponga un nuevo punto final REST para obtener los últimos intentos para un usuario determinado.
- Cree un nuevo servicio (lógica empresarial) para recuperar esos intentos.
- Muestre ese historial de intentos a los usuarios en la página web después de que envíen uno nuevo.

Tenga en cuenta que esta historia de usuario impacta nuestro código de una manera diferente: estábamos verificando la exactitud del intento *sobre la marcha,* entonces nuestro MultiplicationResultAttempt la clase no incluye una bandera para indicar si es correcta o no. Eso estuvo perfectamente bien para hacer frente a los requisitos que teníamos, pero ahora, si usamos el mismo enfoque, nuestra aplicación sería muy ineficiente y tendría que calcular cada vez para extraer los resultados. Por eso necesitamos un *refactorización* tarea, que debe incluirse como parte de esta historia de usuario.

ÁGIL Y REFACTORANTE

Cuando trabajamos siguiendo la metodología ágil, debemos adoptar la refactorización como parte normal de nuestras tareas. Queremos ofrecer valor lo antes posible y luego evolucionar la aplicación en pequeños incrementos.*Eso significa que invertir demasiado tiempo al comienzo del proyecto para diseñar el estado final de nuestra aplicación sería incorrecto: los requisitos podrían cambiar.*

Encontrar un buen equilibrio es la clave. Reúnase con el propietario de su producto o los usuarios comerciales y pregúnteles su visión: qué quieren tener al final del proyecto. luego invierte tiempo en determinar el entregable mínimo que les da valor, la primera parte de todo el proyecto que podrían usar y aún así ahorrar tiempo, ganar dinero, etc.Esta es la parte más difícil cuando se sigue de manera ágil, porque normalmente las partes interesadas del negocio no no quiere sacrificar ninguna funcionalidad, y es posible que se enfrente a una*Quiero-todo-o-nada* problema. Pero esta situación se puede desbloquear con esfuerzo y una buena comunicación entre el final del negocio y la ejecución del proyecto (gerente de proyecto, propietarios de productos, arquitectos y / o desarrolladores). Después de definir la visión y el MVp (producto mínimo viable), comienza el trabajo iterativo: reuniones con las partes interesadas para definir las siguientes porciones de valor para alcanzar el objetivo. cuanto antes lo haga, mejor.

Es fundamental tener una visión perfectamente clara de lo que desea lograr desde el punto de vista comercial al final del proyecto y también un MVp perfectamente definido con casos de uso descriptivos. luego, dado que trabajaremos con Sprints, el trabajo de los próximos tres o cuatro Sprints también debería estar más o menos definido. *Improvisación total* y cambiar de dirección en cada Sprint es muy malo para un proyecto a menos que se base en la experimentación.

Dicho esto, es fundamental que las partes interesadas de su empresa comprendan qué es la refactorización y la adopten también. Mientras trabaja con ágil, habrá situaciones en las que el incremento de valor puede ser muy pequeño, pero la cantidad de esfuerzo para entregarlo es grande. No se puede ser demasiado estricto en el seguimiento del manifiesto ágil

y argumentan que la refactorización no agrega valor. Cuando vea su plan para los próximos tres o cuatro Sprints y pueda ver claramente que se necesita un cambio en el diseño o la arquitectura del software, planifíquelo lo antes posible. si lo omite, se arrepentirá más tarde cuando el*deuda técnica* consume los recursos de su proyecto.

También podríamos cuestionar este libro: en nuestra aplicación podríamos argumentar que el requisito de persistencia debería haber sido claro desde el principio y, por lo tanto, deberíamos haber diseñado los intentos de una manera diferente. es cierto, sin embargo, lo estamos usando para realizar esta explicación como un ejemplo de refactorización ágil.

Intentemos resumir lo que debería incluir nuestra tarea de refactorización:

1. El intento (MultiplicationResultAttempt) debe incluir un booleano para indicar si es correcto o no, lo almacenaremos y luego podremos consultar la base de datos.
2. El servicio (MultiplicationServiceImpl) no solo debe devolver el resultado calculado "sobre la marcha", sino también guardarlo en el intento.
3. El cliente no debería poder marcar un intento como correcto, por lo que este campo no debería leerse desde la API REST, sino calcularse internamente.
4. Las pruebas deben cambiarse para reflejar nuestras nuevas circunstancias.

Dividamos el trabajo entre la refactorización y los cambios para implementar la persistencia.

CÓDIGO FUENTE DISPONIBLE CON EL LIBRO

Puede encontrar todo el código al que se hace referencia desde aquí en el v4 repositorio en github: https://github.com/microservices-practical.

Refactorización del código

Como se explicó, debemos realizar algunos cambios en el código para evitar cálculos innecesarios; almacenaremos un booleano valor en el intento, por lo que podemos consultar la base de datos para los correctos. Ver listado3-25.

Listado 3-25. MultiplicationResultAttempt.java (social-multiplication v4)

```
// Importaciones, anotaciones ...
público final clase MultiplicationResultAttempt {

    privado usuario usuario final;
    privado multiplicación de multiplicación final;
    privado final int resultAttempt;

    privado booleano final correcto;

    // Constructor vacío para (des) serialización JSON
    MultiplicationResultAttempt () {
        usuario = nulo;
        multiplicación = nulo;
        resultAttempt = -1;
        correcto = falso;
    }

}
```

Debido a que agregamos este nuevo campo a nuestra clase, Lombok ahora generará el nuevo constructor y el captador. También se encargará de actualizar eles igual a (), hashCode (), y Encadenar() métodos, por lo que Lombok es ideal para trabajos de refactorización. Eso también significa que tenemos que cambiar elMultiplicationServiceImplTest y MultiplicationResultAttemptControllerTest clases para adaptarlas al nuevo constructor. Ver listado3-26.

Listado 3-26. MultiplicationServiceImplTest.java
(socialmultiplication v4)

```
@Prueba
público void checkCorrectAttemptTest () {
    // dado
    Multiplicación multiplicación = nuevo Multiplicación (50, 60); Usuario
    usuario =nuevo Usuario ("john_doe"); MultiplicationResultAttempt
    intento =nuevo
    MultiplicationResultAttempt (
            usuario, multiplicación, 3000, falso);

    // Cuándo
    boolean AttemptResult = multiplicationServiceImpl.check
    Intento (intento);

    // luego
    afirmar que (intentoResultado) .isVerdadero ();
}

@Prueba
público void checkWrongAttemptTest () {
    // dado
    Multiplicación multiplicación = nuevo Multiplicación (50, 60); Usuario
    usuario =nuevo Usuario ("john_doe"); MultiplicationResultAttempt
    intento =nuevo
    MultiplicationResultAttempt (
            usuario, multiplicación, 3010, falso);
    dado (userRepository.findByAlias   ("john_doe")).
    willReturn (Optional.empty ());

    // Cuándo
    boolean intentResult = multiplicationServiceImpl.
    checkAttempt (intento);
```

```
    // luego
    afirmar que (intentoResultado) .isFalse ();
}
```

¿Cómo establecemos el valor adecuado para el nuevo correcto ¿campo? Agreguemos eso a nuestra lógica comercial dentro de la implementación del servicio. Ver listado3-27.

Listado 3-27. MultiplicationServiceImpl.java (social-multiplication v4)

```
@Anular
público boolean checkAttempt (intento de MultiplicationResultAttempt
final) {
    // Comprueba si es correcto
    booleano correcto = intento.getResultAttempt () ==
                        intent.getMultiplication ().
                        getFactorA () *
                        intent.getMultiplication ().
                        getFactorB ();

    // Evita intentos de 'pirateo'
    Assert.isTrue (! Intent.isCorrect (), "¡¡No puedes enviar un
    intento marcado como correcto !!");

    // Crea una copia, ahora configurando el campo 'correcto' en
    consecuencia
    MultiplicationResultAttempt checkAttempt =
            nuevo MultiplicationResultAttempt (intento.getUser (),
                    intent.getMultiplication (),
                    intent.getResultAttempt (),
                    correcto);

    // Devuelve el resultado
    regreso correcto;
}
```

Tenga en cuenta que el argumento del método, intento,puede contener un cierto valor para el correcto campo (en caso de que estemos tratando con un usuario inteligente que quiere engañar a la aplicación) .Lo que hacemos en este caso es calcular el valor correcto para correcto y configúrelo en una nueva instancia, checkAttempt.Necesitamos crear una copia ya que queremos mantener nuestra clase inmutable. Como puede ver en nuestra lógica, también arrojamos un error a un posible tramposo gracias a la convenienteAfirmar clase incluida en Spring;[4] la aserción desencadenará una Argumento de excepción ilegal.

También tenemos la oportunidad de deshacernos de la clase interior. ResultadoRespuesta que usamos en nuestro controlador. Con nuestro cambio anterior para incluircorrecto, tiene mucho más sentido devolver lo mismo MultiplicationResultAttempt escriba nuestra llamada REST, con el valor booleano indicando si el intento fue correcto o no. Ver listado3-28.

Listado 3-28. MultiplicationResultAttemptController.java (social-multiplication v4)

```
@PostMapping
ResponseEntity <MultiplicationResultAttempt>
postResult (@RequestBody MultiplicationResultAttempt
multiplicationResultAttempt) {
    booleano isCorrect = multiplicationService.checkAttempt
    (multiplicationResultAttempt);
    MultiplicationResultAttempt AttemptCopy = nuevo
    MultiplicationResultAttempt (
            multiplicationResultAttempt.getUser (),
            multiplicationResultAttempt.getMultiplication (),
            multiplicationResultAttempt.getResultAttempt (),
            isCorrect
    );
    regreso ResponseEntity.ok (intentCopy);
}
```

[4] https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework / util / Assert.html

Para completar la refactorización, aplicamos los cambios correspondientes a nuestro MultiplicationResultAttemptControllerTest para usar el nuevo constructor y para verificar el devuelto MultiplicationResultAttempt objeto, en lugar del anterior booleano. Ver listado6-29.

Listado 3-29. MultiplicationResultAttemptControllerTest.java (social-multiplication v4)

```
void genericParameterizedTest (final booleano correcto) lanza
Excepción {
    // dado (recuerde que no estamos probando aquí el servicio
    en sí)
    // ...
    MultiplicationResultAttempt intento = nuevo
    MultiplicationResultAttempt (
            usuario, multiplicación, 3500, correcto);

    // Cuándo
    // ...

    // luego
    afirmarQue (respuesta.getStatus ()). isEqualTo (HttpStatus.
    OK.value ());
    assertThat (response.getContentAsString ()). isEqualTo (
            jsonResult.write (
                    nuevo MultiplicationResultAttempt (intento.
                    GetUser (),
                            intent.getMultiplication (),
                            intent.getResultAttempt (),
                            correcto)
            ). getJson ());
}
```

Normalmente, si cambia un tipo de retorno para una respuesta en su API REST, también deberá cambiar su interfaz. Sin embargo, en nuestro caso

no es necesario cambiar nada por ahora. Solíamos devolver un objeto JSON simple con un correcto booleano en el interior; ahora devolvemos un JSON más grande (de MultiplicationResultAttempt) que tambien tiene un correcto booleano. Por suerte para nosotros, nuestromultiplication-client.js seguirá trabajando. Ver listado3-30.

Listado 3-30. multiplication-client.js (social-multiplication v4)

```
// Envía los datos usando post
$ .ajax ({
    url: '/ resultados',
    tipo: 'POST',
    datos: JSON.stringify (datos),
    contentType: "application / json; charset = utf-8",
    dataType: "json",
    asincrónico: falso,
    éxito: función(resultado){
        Si(result.correct) {
            $ ('. result-message'). empty (). append ("¡El resultado
            es correcto! ¡Felicitaciones!");
        } demás {
            $ ('. result-message'). empty (). append ("¡Vaya, eso no es
            correcto! ¡Pero sigue intentándolo!");
        }
    }
});
```

La capa de datos

La tarea de refactorización ha finalizado, así que ahora continuemos colocando nuestra aplicación Spring Boot presentando nuestra capa de datos. Para este caso de uso, nos beneficiaremos de un marco ORM como Hibernate: conservaremos los datos en nuestra base de datos siguiendo un modelo que se puede asignar a los objetos Java. Si tu mantienes

el modelo no es demasiado complejo, es una solución que hace que el trabajo con la capa de persistencia sea muy sencillo. Para lograr esto, usaremos el paquete de inicio de Spring Boot para la API de persistencia de Java (JPA), que incluye Hibernate. JPA es solo la especificación estándar para la persistencia que implementan muchos proveedores diferentes (Hibernate es uno de ellos), y siempre es una buena idea usar estándares en lugar de vincularnos a una implementación específica. Si desea saber más sobre JPA, puede leer la documentación oficial o visitar el sitio de ObjectDB en http://www.objectdb.com/api/java/jpa/annotations.

El primer paso que debemos dar es incluir dos nuevas dependencias en nuestro pom.xml expediente:

- La Spring-boot-starter-data-jpa la dependencia nos dará acceso a Spring Data JPA[5] herramientas, como crear repositorios de una manera fácil y rápida. Este iniciador proporciona soporte para JPA usando Hibernate.
- La h2 artifact incluye un motor de base de datos integrado y ligero llamado H2. Podríamos haber usado MySQL, PostgreSQL o cualquier otro motor de base de datos, pero este se ajusta a nuestros requisitos y hace que nuestro servicio sea más fácil de explicar para el libro. Ver listado3-31.

Listado 3-31. pom.xml Adición de dependencias relacionadas con datos (social-multiplication v4)

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

<dependencia>
```

[5] http://projects.spring.io/spring-data-jpa/

```
  <groupId>com.h2database </groupId>
<artifactId>h2 </artifactId> <alcance>tiempo
de ejecución </alcance> </dependencia>
```

Luego llenamos la configuración en el application.properties expediente. Dado que usamos Spring Initializr para generar nuestro proyecto, este archivo debe estar ubicado ensrc / main / resources. Si lo prefiere, también puede usar el formato YAML, pero luego cambie el nombre del archivo a application.yml. Ver listado 3-32.

Listado 3-32. application.properties (social-multiplication v4)

```
# Nos da acceso a la consola web de la base de datos H2
spring.h2.console.enabled = true
# Genera la base de datos * solo * si aún no está allí
spring.jpa.hibernate.ddl-auto = actualizar
# Crea la base de datos en un archivo
spring.datasource.url = jdbc: h2: file: ~ / social-multiplication;
DB_CLOSE_ON_EXIT = FALSE;
# Con fines educativos mostraremos el SQL en la consola
spring.jpa.properties.hibernate.show_sql = true
```

La consola H2 es una interfaz de usuario web liviana que nos permite administrar y consultar la base de datos H2. Las instrucciones sobre cómo configurarlo con Spring Boot se pueden encontrar en *https://tpd.io/h2-spring*. Como puede ver, solo necesitamos especificar algunas propiedades básicas; Todo lo demás está siendo configurado automáticamente por Spring Boot. Una parte importante de la configuración es la URL. Allí especificamos que queremos que la base de datos se almacene en un archivo y el nombre de la base de datos. Si no lo configuramos enexpediente, tendríamos una base de datos en memoria y perderíamos nuestros datos cada vez que cerramos el servicio. Tenga en cuenta que estamos usando ~ /, por lo que el archivo se ubicará en la carpeta de inicio del usuario de su sistema operativo.

PREPARACIÓN PARA LA PRODUCCIÓN: UTILIZANDO UN MOTOR DB DIFERENTE

h2 es un motor válido para los objetivos de nuestra aplicación, también porque la intención de este libro es no entrar en detalles de la base de datos. Es posible que desee explorar algunas alternativas mejores para una base de datos de producción, y luego puede configurarla siguiendo las instrucciones de la página de documentación oficial de Spring Boot para eso (consultehttps://tpd.io/boot-prod-db). la buena noticia es que están configurados de forma muy similar a h2.

El modelo de datos

Esta es la tarea más importante que realizamos para modelar nuestra capa de persistencia: diseñar el modelo de datos. Definimos previamente cómo se ven nuestras entidades comerciales; ahora tenemos que definir cómo queremos que se relacionen desde el punto de vista de los datos.

A veces, los modelos de datos no coinciden con los modelos de dominio: por ejemplo, es posible que tenga en su dominio CustomerWithPersonalDetails y EmployeeWithPersonalDetails porque desea que sean simples, pero es posible que desee mantener el cliente, empleado, y detalles personales las tablas se separan para evitar la replicación de datos y ahorrar algo de espacio en el futuro. En este caso, haremos un mapeo directo: las entidades de dominio coinciden con las entidades de datos. Figura3-3 muestra el modelo de datos.

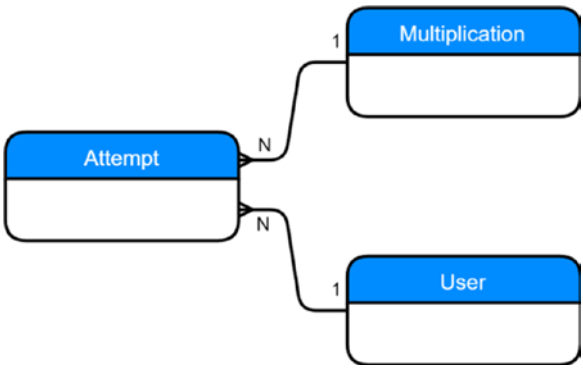

Figura 3-3. *El modelo de datos actual*

No es realmente complejo, pero es un buen ejemplo. Podemos explicarlo desde el Usuario entidad: pueden tener muchos intentos. Al mismo tiempo, los intentos provenientes de diferentes usuarios pueden tener una relación con la misma multiplicación (si tienen factores iguales).

¿Cómo podemos modelar esto con JPA? La forma más sencilla es utilizar las anotaciones proporcionadas. No entraremos en detalles sobre JPA porque eso podría abarcar un libro completo, pero cubriremos los conceptos básicos observando los cambios en nuestras entidades y explicándolos. Empecemos con elMultiplicación clase, como se muestra en el listado 3-33.

Listado 3-33. Multiplication.java (social-multiplication v4)

```
paquete microservices.book.multiplication.domain;

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

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

/**
 * Esta clase representa una multiplicación (a * b).
 */
@RequiredArgsConstructor
@Getter
@Encadenar
@EqualsAndHashCode
@Entidad
público final clase Multiplicación {
```

```
    @Identificación
    @GeneratedValue
    @Columna (nombre = "MULTIPLICATION_ID")
    privado Identificación larga;

    // Ambos factores
    privado int final factorA;
    privado int final factorB;

    // Constructor vacío para JSON / JPA
    Multiplicación () {
        esto(0, 0);
    }
}
```

- La @Entidad La anotación se utiliza para especificar que la clase debe considerarse una entidad JPA, por lo que se puede almacenar en un repositorio JPA. Tenga en cuenta que también tenemos un constructor vacío, que JPA requiere para poder instanciar objetos a través de la reflexión.
- Usaremos identificadores únicos como claves primarias, y para eso usaremos Java Largo clase. La anotación @Identificación le dice a JPA que será el identificador de clave principal, y @GeneratedValue indica que debe ser autogenerado (no lo estamos configurando).
- En algunos casos, es posible que deseemos establecer explícitamente el nombre de la columna en lugar de dejar que JPA lo haga. Para hacerlo, podemos usar @Columna anotación. A veces es útil porque queremos darle un nombre fijo a una columna y luego usarlo desde una entidad diferente para unir dos tablas (que será el caso aquí). También podríamos omitir esta anotación y hacer referencia a esteidentificación campo más tarde utilizando los nombres de campo y tabla de MultiplicationResultAttempt.

Necesitamos hacer cambios muy similares a la Usuario clase, como se muestra en el listado 3-34.

Listado 3-34. User.java (multiplicación social v4)

```
@RequiredArgsConstructor
@Getter
@Encadenar
@EqualsAndHashCode
@Entidad
público final clase Usuario {

    @Identificación
    @GeneratedValue
    @Column (nombre = "USER_ID")
    privado Identificación larga;

    privado alias de cadena final;

    // Constructor vacío para JSON / JPA
    protegido Usuario () {
        alias = nulo;
    }
}
```

Los cambios más interesantes están en MultiplicationResultAttempt, ya que es la parte del modelo que vincula a las demás entidades. Ver listado3-35.

Listado 3-35. MultiplicationResultAttempt.java (socialmultiplication v4)

```
paquete microservices.book.multiplication.domain;

importar lombok.EqualsAndHashCode;
importar lombok.Getter;
```

```
importar lombok.RequiredArgsConstructor;
importar lombok.ToString;

importar javax.persistence. *;

/ **
  * Identifica el intento de una {@Enlace Usuario} para resolver un
  * {@Enlace Multiplicación}.
  * /
@RequiredArgsConstructor
@Getter
@Encadenar
@EqualsAndHashCode
@Entidad
público final clase MultiplicationResultAttempt {

    @Identificación
    @GeneratedValue
    privado Identificación larga;

    @ManyToOne (cascade = CascadeType.PERSIST)
    @JoinColumn (nombre = "USER_ID")
    privado usuario usuario final;

    @ManyToOne (cascade = CascadeType.PERSIST)
    @JoinColumn (nombre = "MULTIPLICATION_ID")
    privado multiplicación de multiplicación final;
    privado final int resultAttempt;

    privado booleano final correcto;

    // Constructor vacío para JSON / JPA
    MultiplicationResultAttempt () {
        usuario = nulo;
```

```
            multiplicación = nulo;
            resultAttempt = -1;
            correcto = falso;
        }

}
```

- Para especificar la relación entre entidades, JPA proporciona varias anotaciones: @OneToOne, @OneToMany, @ManyToOne, y @Muchos a muchos. Con algunas de estas anotaciones, puede definir también los detalles de cómo desea vincularlas. Consulta la documentación si quieres saber más sobre esto. Con la configuración anterior, nuestra tabla de intentos tendrá dos claves externas para los identificadores respectivos deUsuario y Multiplicación (si necesita más conocimientos sobre el *Clave primaria* y *clave externa* conceptos, lea el artículo en *https: // tpd.io/ms-pk-fk).*

- Para nuestro MultiplicationResultAttempt elegimos un tipo de cascada PERSISTIR. Lo que queremos lograr con esto es que podamos almacenar desde nuestro código Java directamente los intentos, y cualquier otra entidad vinculada se mantendrá (si no está ya allí) también en sus tablas correspondientes.

- Como se muestra, usamos @JoinColumn para hacer referencia a otras entidades usando sus identificadores (usando el nombre proporcionado allí a través de @Columna).

Mesas 3-1 mediante 3-3 mostrar un ejemplo de cómo se ven estas entidades en las diferentes tablas para una iteración dada del juego (un intento de resolver una multiplicación enviada por un usuario dado).

Tabla 3-1. *Tabla de USUARIO (multiplicación social v4)*

USER_ID	ALIAS
3	juan

Tabla 3-2. *Tabla de MULTIPLICACIÓN (multiplicación social v4)*

MULTIPLICATION_ID	FACTORA	FACTORB
8	41	54

Tabla 3-3. *Tabla MULTIPLICATION_RESULT_ATTEMPT (multiplicación social v4)*

ID	CORRECTO	RESULT_ATTEMPT	MULTIPLICATION_ID	USER_ID
11	verdadero	2214	8	3

Los repositorios

Ahora que tenemos las herramientas necesarias en nuestro proyecto para implementar la persistencia y tenemos nuestro modelo diseñado, podemos crear nuestros repositorios JPA para que podamos almacenar y leer nuestros objetos Java. Siguiendo nuestra estructura de empaquetado y el patrón de aplicación en capas, crearemos repositorios en un nuevorepositorio paquete.

Comencemos con el repositorio para almacenar MultiplicationResultAttempt objetos, como se muestra en el listado 3-36.

Listado 3-36. MultiplicationResultAttemptRepository (social-multiplication v4)

```
paquete microservices.book.multiplication.repository;

importar microservices.book.multiplication.domain.
MultiplicationResultAttempt;
```

```
importar org.springframework.data.repository.CrudRepository;

importar java.util.List;

/**
 * Esta interfaz nos permite almacenar y recuperar intentos
 */
interfaz pública MultiplicationResultAttemptRepository
        se extiende CrudRepository <MultiplicationResultAttempt,
Largo> {

    /**
     * @regreso lo último 5 intentos para un usuario
     determinado, identificado por su alias.
     */
    List <MultiplicationResultAttempt> findTop5ByUserAliasOrder
    ByIdDesc (String userAlias);
}
```

Una vez más, una solución simple: con solo crear una interfaz que amplíe una de las interfaces provistas en Spring Data JPA, tendremos toda la funcionalidad que necesitamos en nuestra aplicación. En este caso, CrudRepository es una solución conveniente (una especie de interfaz mágica) proporcionada por Spring para implementar las operaciones para crear, leer, actualizar y eliminar entidades (CRUD). Utiliza genéricos de Java, por lo que solo necesitamos pasar como parámetros la clase anotada con @Entidad para el cual queremos un repositorio (el primero), y el tipo de identificador (en nuestro caso, los declaramos como Largo). PagingAndSortingRepositoryTambién es útil que, además de las operaciones CRUD, proporciona capacidades de paginación y clasificación.

¿Has notado el findTop5ByUserAliasOrderByIdDesc ¿método? Está usando otra característica interesante de Spring Data JPA: métodos de consulta. Simplemente siguiendo algunos patrones de nomenclatura dados, puede crear fácilmente consultas personalizadas definiendo el método en la interfaz. Puedes conseguir mas

información sobre esto en la página de documentación oficial.[6] Si no te gusta esa magia, también puedes crear tu método (llámalo como quieras) y usa @Consulta anotación con su JPQL personalizado (cubriremos algunos casos en el próximo capítulo). Puede encontrar más información sobre esta alternativa en la misma página.

Podemos aplicar lo que sabemos ahora para crear nuestro UserRepository y MultiplicationRepository clases. El último no necesita ningún método de consulta personalizada, por lo que es solo la declaración de la interfaz (y, sin embargo, tenemos el poder de todos los métodos predefinidos enCrudRepository). Ver listados 3-37 y 3-38.

Listado 3-37. UserRepository (social-multiplication v4)

```
paquete microservices.book.multiplication.repository;

importar microservices.book.multiplication.domain.User;
importar org.springframework.data.repository.CrudRepository;

importar java.util.Optional;


/**
 * Esta interfaz nos permite guardar y recuperar usuarios
 */
interfaz pública UserRepository se extiende CrudRepository <Usuario,
largo> {

    <User> opcional findByAlias   (alias de cadena final);

}
```

[6] https://docs.spring.io/spring-data/jpa/docs/current/reference/ html / # repositories.query-methods

Listado 3-38. MultiplicationRepository (social-multiplication v4)

```
paquete microservices.book.multiplication.repository;

importar microservices.book.multiplication.domain.Multiplication;
importar org.springframework.data.repository.CrudRepository;

/**
 * Esta interfaz nos permite guardar y recuperar
Multiplicaciones
 */
interfaz pública MultiplicationRepository se extiende
CrudRepository <Multiplication, Long> {}
```

Es posible que se pregunte en este punto por qué no seguimos TDD para los repositorios. La respuesta es simple: no hay código nuevo y confiamos en la implementación proporcionada por Spring, por lo que no necesitamos incluir pruebas unitarias para estos*muy delgado* repositorios.

ELEGIR QUÉ MÉTODOS EXPONER

la Repositorio La interfaz tiene un truco oculto: si crea métodos de interfaz que coincidan con la firma de CrudRepository, el resultado es un parcial CrudRepository solución con solo los métodos que desea exponer. Puede encontrar este truco documentado en el Javadoc.

Una vez que tenemos nuestros repositorios, queremos usarlos. Nuestra lógica empresarial debe encargarse de llamarlos a persistir en nuestras entidades. Volviendo a TDD: incluyémoslo en nuestra prueba unitaria verificando la lógica del intento (MultiplicationServiceImplTest), tanto por una correcta como por una incorrecta. Ver listado3-39.

Listado 3-39. MultiplicationServiceImplTest (social-multiplication v4)

```
// paquete, importaciones ...
clase pública MultiplicationServiceImplTest {

    privado MultiplicationServiceImpl
    multiplicationServiceImpl;

    @Burlarse de
    privado RandomGeneratorService randomGeneratorService;

    @Burlarse de
    privado MultiplicationResultAttemptRepository
    intentoRepository;

    @Burlarse de
    privado UserRepository userRepository;

    @Antes
    público configuración vacía () {
        // Con esta llamada a initMocks le decimos a Mockito que
        procese las anotaciones
        MockitoAnnotations.initMocks (esto);
        multiplicationServiceImpl = nuevo MultiplicationService
        Impl (randomGeneratorService, intentRepository,
        userRepository);
    }

    @Prueba
    público void createRandomMultiplicationTest () {
        // [...] aquí no hay cambios, mantenlo como estaba antes
    }
```

```
@Prueba
público void checkCorrectAttemptTest () {
    // dado
    Multiplicación multiplicación = nuevo Multiplicación
    (50, 60);
    Usuario usuario = nuevo Usuario ("john_doe");
    MultiplicationResultAttempt intento =nuevo
    MultiplicationResultAttempt (
            usuario, multiplicación, 3000, falso);
    MultiplicationResultAttempt verificadoAttempt = nuevo
    MultiplicationResultAttempt (
            usuario, multiplicación, 3000, cierto);
    dado (userRepository.findByAlias ("john_doe")).
    willReturn (Optional.empty ());

    // Cuándo
    boolean intentResult = multiplicationServiceImpl.
    checkAttempt (intento);

    // luego
    afirmar que (intentoResultado) .isVerdadero (); verificar
    (tryRepository) .save (VerifiedAttempt);
}

@Prueba
público void checkWrongAttemptTest () {
    // dado
    Multiplicación multiplicación = nuevo Multiplicación
    (50, 60);
    Usuario usuario = nuevo Usuario ("john_doe");
    MultiplicationResultAttempt intento =nuevo
    MultiplicationResultAttempt (
            usuario, multiplicación, 3010, falso);
    dado (userRepository.findByAlias ("john_doe")).
    willReturn (Optional.empty ());
```

```
        // Cuándo
        boolean intentResult = multiplicationServiceImpl.
        checkAttempt (intento);

        // luego
        afirmar que (intentoResultado) .isFalse ();
        verificar (intentoRepositorio). guardar (intento);
    }
}
```

- Necesitamos tomar MultiplicationResultAttempt Repository y UserRepository para mantener la prueba unitaria centrada en la capa de servicio. Tenga en cuenta en el código que también se pasan en elMultiplicationServiceImpl constructor. Es posible que desee actualizar esa línea más adelante, cuando modifique esa clase, o también puede crear el nuevo constructor en este punto.
- En nuestro checkCorrectAttemptTest () estamos incluyendo una copia del intento (VerifiedAttempt) para acercarlo a la realidad: el enviado por el usuario debe tener un falso valor para el correcto campo. Luego, en la última línea, verificarusingMockito) que llamamos al repositorio para almacenar el que tiene correcto ajustado a cierto.
- La checkWrongAttemptTest () necesita una actualización para comprobar que el repositorio también se llama por intentos incorrectos. Recuerde que la llamada real no se ejecuta: solo estamos verificando que los objetos simulados sean llamados con esos argumentos.

Así que ahora volvemos a tener una prueba fallida (nunca se llama a los repositorios). Necesitamos agregar los repositorios a nuestroMultiplicationServiceImpl y salvar los intentos en ambos casos. Listado3-40 muestra los cambios.

Listado 3-40. MultiplicationServiceImpl (multiplicación social v4)

```
// [...]
@Servicio
clase MultiplicationServiceImpl implementos
        MultiplicationService {

    privado RandomGeneratorService randomGeneratorService;
    privado MultiplicationResultAttemptRepository
                intentRepository;
    privado UserRepository userRepository;

    @Autowired
    público MultiplicationServiceImpl (final
            RandomGeneratorService randomGeneratorService,
                                        multiplicación final
                                        ResultAttemptRepository
                                        intentoRepository,
                                        UserRepository final
                                        userRepository) {
        esto.randomGeneratorService = randomGeneratorService;
        esto.intentoRepositorio = intentoRepositorio;
        esto.userRepository = userRepository;
    }

    // [...]

    @Transaccional
    @Anular
    público boolean checkAttempt (intento de intento de
    resultado de multiplicación final) {
        // Verifica si el usuario ya existe para ese alias
        <User> opcional user = userRepository.findBy
        Alias   (intent.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);

        regreso es correcto;
    }
}
```

- Un concepto importante para entender aquí es que realmente no necesitamos usar todos los nuevos repositorios. Desde que presentamos antes de nuestroCascadeType.PERSIST en el MultiplicationResultAttempt entidad, cada vez que guardamos un intento, el enlace Multiplicación y Usuario los objetos también se conservarán.

- Sin embargo, todavía necesitamos UserRepository para obtener el identificador de usuario dado el alias. Cada vez que recibimos un intento, es uno nuevo desde el punto de vista de la API REST, por lo que está vinculado a unMultiplicación y Usuario con ID nulos. Si el usuario es uno existente, ya estará en nuestra base de datos. Queremos vincular este intento al usuario, por lo que necesitamos recuperar el identificador existente de la base de datos dado el alias del usuario. Usando Java Opcional, luego podemos resolver bien si es un nuevo (ID nulo) o un usuario existente con user.orElse (intento. getUser ()).
- Cambiamos el constructor, así que ahora Spring también inyecta las implementaciones del repositorio (no las desarrollamos; son *automágicamente* generado).
- Almacenamos el intento utilizando el repositorio de intentos. Como se mencionó anteriormente, JPA también almacenará las entidades vinculadas.

EJERCICIO (OPCIONAL)

¡Es hora de que aceptes un desafío! Con la implementación actual, tenemos un pequeño problema con las multiplicaciones. Persistiremos cada uno de ellos como si fueran nuevos, incluso si los hay con la misma combinación de factorB y factorB. ¿Estás listo para resolverlo? Acabamos de cubrir la idea principal, así que ... ¡adelante!

Completando la historia de usuario 2: pasando por las capas

Casi hemos completado la historia de usuario 2. Recuerda: queríamos mostrar a los usuarios sus últimos intentos. Ahora los estamos guardando, por lo que podemos crear un punto final de API REST para recuperar los más recientes para un usuario determinado. Cubrimos esto parcialmente cuando presentamos el método de consultafindTop5ByUserAliasOrderByIdDesc () en MultiplicationResultAttemptRepository. Vamos a vincularlo de nuevo a través de todas las capas a la interfaz de usuario.

La siguiente capa es el servicio. Dado que este no contiene mucha lógica empresarial, primero escribamos nuestra implementación y luego agreguemos nuestro nuevo caso de prueba. Ver listado3-41.

Listado 3-41. MultiplicationServiceImpl Adición de un nuevo método (social-multiplication v4)

```
@Servicio
clase MultiplicationServiceImpl implementos
MultiplicationService {

    // [...]

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

Listado 3-42. MultiplicationServiceImplTest Agregar una prueba (socialmultiplication v4)

```
clase pública MultiplicationServiceImplTest {

    // [...]

    @Prueba
    público void retrieveStatsTest () {
        // dado
        Multiplicación multiplicación = nuevo Multiplicación
        (50, 60);
        Usuario usuario = nuevo Usuario ("john_doe");
        MultiplicationResultAttempt intento1 =nuevo
        MultiplicationResultAttempt (
                usuario, multiplicación, 3010, falso);
        MultiplicationResultAttempt Attempt2 = nuevo
        MultiplicationResultAttempt (
                usuario, multiplicación, 3051, falso);
        List <MultiplicationResultAttempt> latestAttempts =
        Lists.newArrayList (intento1, intento2);
        dado (userRepository.findByAlias   ("john_doe")).
        willReturn (Optional.empty ());
        dado (intentRepository.findTop5ByUserAliasOrderById
        Desc ("john_doe"))
                . willReturn (latestAttempts);

        // Cuándo
        Lista <MultiplicationResultAttempt> latestAttemptsResult =
                multiplicationServiceImpl.
                getStatsForUser ("john_doe");
```

```
        // luego
        assertThat (latestAttemptsResult) .isEqualTo
        (latestAttempts);
    }
}
```

Es lo mismo para la capa Controlador: toda la lógica proviene de la consulta, por lo que solo necesitamos pasar el resultado. En este caso, necesitamos hacer una ligera modificación a la clase de prueba para incluir una nuevaJacksonTester para afirmar los resultados mediante una lista de intentos. Ver listados3-43 y 3-44.

Listado 3-43. MultiplicationResultAttemptController Adición de un nuevo método (social-multiplication v4)

```
@RestController
@RequestMapping ("/ results") final clase
MultiplicationResultAttemptController {

    // [...]

    @GetMapping
    ResponseEntity <List <MultiplicationResultAttempt>>
    getStatistics (@RequestParam ("alias") String alias) {
        regreso ResponseEntity.ok (
                multiplicationService.getStatsForUser (alias)
        );
    }
}
```

Listado 3-44. MultiplicationResultAttemptControllerTest - agregar una prueba (social-multiplication v4)

```
@RunWith (SpringRunner.class) @WebMvcTest
(MultiplicationResultAttemptController.class)
```

```
clase pública MultiplicationResultAttemptControllerTest {

    // [...]

    // Estos objetos se inicializarán mágicamente mediante el
    método initFields a continuación.
    privado JacksonTester <MultiplicationResultAttempt>
    jsonResultAttempt;
    privado JacksonTester <List <MultiplicationResultAttempt>>
    jsonResultAttemptList;

    @Antes
    público configuración vacía () {
        JacksonTester.initFields (esto, nuevo ObjectMapper ());
    }

    // [...]

    @Prueba
    público void getUserStats () lanza Excepción {
        // dado
        Usuario usuario = nuevo Usuario ("john_doe"); Multiplicación
        multiplicación =nuevo Multiplicación (50, 70);

        MultiplicationResultAttempt intento = nuevo
        MultiplicationResultAttempt (
                usuario, multiplicación, 3500, cierto);
        List <MultiplicationResultAttempt> RecentAttempts =
        Lists.newArrayList (intento, intento);
        dado (multiplicationService
                . getStatsForUser ("john_doe"))
                . willReturn (RecentAttempts);
```

```
        // 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>' +
```