Aprender

Microservicios con Spring Boot

Un enfoque práctico para los servicios RESTful utilizando RabbitMQ, Eureka, Ribbon, Zuul y Pepino

Moises Macero

Apress®

Aprenda microservicios con Spring Boot

Un enfoque práctico para RESTful Servicios que utilizan RabbitMQ, Eureka, Ribbon, Zuul y Pepino

Moises Macero

Apress®

LearnMicroservices con Spring Boot: un enfoque práctico para los servicios RESTful usando RabbitMQ, Eureka, Ribbon, Zuul y Cucumber

Moises Macero

Nueva York, Estados Unidos

ISBN-13 (pbk): 978-1-4842-3164-7 ISBN-13 (electrónico): 978-1-4842-3165-4
https://doi.org/10.1007/978-1-4842-3165-4

Número de control de la Biblioteca del Congreso: 2017962334

Imagen de portada de Freepik (www.freepik.com)

Director general: Welmoed Spahr Director editorial: Todd Green Acquisitions Editor: Steve Anglin Editor de desarrollo: MatthewMoodie Revisor técnico: Manuel Jordan Elera Editor coordinador: Mark Powers Redactor de copias: Kezia Endsley

Distribuido al comercio de libros en todo el mundo por Springer Science + Business Media New York, 233 Spring Street, 6th Floor, Nueva York, NY 10013. Teléfono 1-800-SPRINGER, fax (201) 348-4505, correo electrónico orders-ny@springer-sbm.com , o visite www.springeronline.com. Apress Media, LLC es una LLC de California y el único miembro (propietario) es Springer Science + Business Media Finance Inc (SSBMFinance Inc). SSBMFinance Inc es una**Delaware** corporación.

Para obtener información sobre traducciones, envíe un correo electrónico a rights@apress.com o visite http://www.apress.com/rights-permissions.

Los títulos de Apress se pueden comprar al por mayor para uso académico, corporativo o promocional. Las versiones y licencias de libros electrónicos también están disponibles para la mayoría de los títulos. Para obtener más información, consulte nuestra página web de ventas a granel de libros electrónicos e impresos en http://www.apress.com/bulk-sales.

Cualquier código fuente u otro material complementario al que haga referencia el autor en este libro está disponible para los lectores en GitHub a través de la página del producto del libro, ubicada en www.apress.com/9781484231647. Para obtener información más detallada, visite http://www.apress.com/source-code.

Impreso en papel sin ácido

Tabla de contenido

Sobre el Autor

Moises Macero ha sido desarrollador de software desde que era niño. Ha trabajado en grandes empresas y también en startups, donde ser un desarrollador full-stack era fundamental. Durante su carrera, Moisés ha trabajado con mayor frecuencia en desarrollo, diseño y arquitectura, para proyectos pequeños y grandes, y tanto en entornos ágiles como en cascada. Le gusta trabajar en equipos donde no solo puede entrenar a otros, sino también aprender de ellos.

Moises también es el autor del blog. *thepracticaldeveloper.com,* donde comparte con otros soluciones para desafíos técnicos, guías y su visión sobre las formas de trabajar en las empresas de TI. En su tiempo libre, le gusta viajar y hacer senderismo.

Puedes seguir a Moisés en su cuenta de twitter @moises_macero.

Acerca del revisor técnico

Manuel Jordan Elera es un autodidacta Desarrollador e investigador que disfruta aprendiendo nuevas tecnologías para sus propios experimentos, que se enfocan en encontrar nuevas formas de integrarlas.

Manuel ganó el Premio Springy 2010 - Campeón de la comunidad y Campeón de primavera 2013. En su poco tiempo libre, lee la Biblia y compone música con su bajo y su guitarra.

Manuel cree que la educación constante y la formación es fundamental para todos los desarrolladores. Puede comunicarse con él principalmente a través de su cuenta de Twitter @dr_pompeii.

CAPÍTULO 1

Introducción

Preparando la escena

Los microservicios se están volviendo muy populares en estos días. No es una sorpresa; este estilo de arquitectura de software tiene muchas ventajas, como flexibilidad y facilidad de escala. Mapearlos en pequeños equipos en una organización también le brinda mucha eficiencia en el desarrollo. Sin embargo, emprender la aventura de los microservicios sabiendo solo los beneficios es una decisión incorrecta: debe saber a qué se enfrenta y estar preparado para eso con anticipación. Puede obtener mucho conocimiento de muchos libros y artículos en Internet, pero cuando obtiene el código práctico, la historia cambia.

Este libro cubre algunos de los conceptos más importantes de microservicios de una manera práctica, pero no sin explicar los conceptos. Primero, definimos un caso de uso: una aplicación para construir. Luego, comenzamos con un pequeño monolito, basado en un razonamiento sólido. Una vez que tenemos la aplicación mínima en su lugar, evaluamos si vale la pena pasar a los microservicios y cuál sería una buena forma de hacerlo. ¿Cómo debemos comunicar estas diferentes piezas? Luego, podemos describir e introducir el patrón de arquitectura impulsada por eventos para lograr un acoplamiento suelto informando a otras partes del sistema acerca de *Qué pasó* en su parte del proceso, en lugar de llamar explícitamente a otros a la acción. Una vez que tenga los microservicios en su lugar, verá en la práctica cómo funcionan las herramientas circundantes: *descubrimiento de servicios, enrutamiento, etc.* No cubrimos todos los temas una vez, sino que los incluimos uno por uno, explicando los beneficios de cada uno para la aplicación. Además, analizamos cuál sería una buena forma de probar el

sistema distribuido, de extremo a extremo.

M. Macero, *LearnMicroservices con Spring Boot,*
https://doi.org/10.1007/978-1-4842-3165-4_1

La ventaja de ir paso a paso, deteniéndose cuando es necesario para establecer los conceptos, es que *comprenderá qué problema intenta resolver cada herramienta*. Es por eso que el ejemplo en evolución es una parte esencial de este libro. También puede comprender los conceptos sin codificar una sola línea: la mayor parte del código fuente está incluido para que pueda leerlo si lo prefiere.

El código fuente incluido en este libro está disponible en el repositorio de GitHub. https://github.com/microservices-practical. Está dividido en diferentes versiones, lo que le facilita ver cómo evoluciona la aplicación a lo largo de los capítulos. El libro incluye notas con la versión que se trata en esa sección.

¿Quién eres tú?

Primero comencemos con esto: ¿qué tan interesante será este libro para usted? Este libro es práctico, así que juguemos a este juego. Si se identifica con alguna de estas afirmaciones, este libro puede resultarle útil.

- Me gustaría aprender cómo construir microservicios con Spring Boot y cómo usar las herramientas relacionadas.
- Todo el mundo habla de microservicios, pero todavía no tengo ni idea de lo que es un microservicio: o leo explicaciones teóricas o simplemente artículos exagerados. No entiendo las ventajas, aunque trabajo en TI ...
- Me gustaría aprender a diseñar y desarrollar aplicaciones Spring Boot, pero la gente recomienda libros del tamaño de Don-Quijote,[1] a veces incluso varios de ellos. ¿Existe alguna fuente única de la que pueda obtener un control rápido y práctico de los microservicios sin leer 1.000 páginas?

[1] Don Quijote. Aunque es un libro extenso, sigue siendo una obra maestra.

- Conseguí un nuevo trabajo y están usando una arquitectura de microservicios. He estado trabajando principalmente en proyectos grandes y monolíticos, por lo que me gustaría tener algunos conocimientos y orientación para aprender cómo funciona todo allí.
- Cada vez que voy a la cafetería, los desarrolladores hablan de microservicios ... No puedo socializar más con mis colegas si no entiendo lo que están diciendo. *Está bien, esto es una broma; no lea este libro por eso, especialmente si no está interesado en la programación.*

Con respecto a los conocimientos necesarios para leer este libro, los siguientes temas deberían resultarle familiares:

- Java (algunas partes del código usan Java 8 en este libro)
- Spring (no necesita una gran experiencia, pero debe saber, al menos, cómo funciona la inyección de dependencia)
- Maven (si conoces a Gradle, también estarás bien)

¿En qué se diferencia este libro de otros libros y guías?

Razonamiento detrás de las técnicas

Los desarrolladores de software y los arquitectos leen muchos libros y guías técnicos, ya sea porque estamos interesados en aprender nuevas tecnologías o simplemente porque lo necesitamos para nuestro trabajo. Tenemos que hacer eso de todos modos, ya que es un mundo en constante cambio. Podemos encontrar todo tipo de libros y guías por ahí. Los buenos suelen ser aquellos de los que aprendes rápidamente y los que te enseñan no solo cómo hacer las cosas, sino también por qué debes hacerlo de esa manera. Usar nuevas técnicas solo porque son nuevas es la forma incorrecta de hacerlo; debe comprender el razonamiento detrás de ellos para poder usarlos de la mejor manera posible.

Este libro utiliza esa filosofía: navega a través del código y los patrones de diseño, explicando las razones para seguir un camino y no otros.

Aprendizaje: un proceso incremental

Si observa las guías disponibles en Internet, notará rápidamente que no son ejemplos de la vida real. Por lo general, cuando aplica esos casos a escenarios más complejos, no encajan. Las guías son demasiado superficiales para ayudarte a construir algo real.

Los libros, por otro lado, son mucho mejores en eso. Hay muchos buenos libros que explican conceptos en torno a un ejemplo; son buenos porque aplicar conceptos teóricos al código no siempre es fácil si no ve el código. El problema con algunos de estos libros es que no son tan prácticos como guías. Primero debe leerlos para comprender los conceptos, luego codificar (o ver) el ejemplo, que con frecuencia se da como una pieza completa. Es difícil poner en práctica conceptos cuando ves la versión final directamente. Este libro se mantiene en el lado práctico y comienza con un código que evoluciona a través de la refactorización, por lo que los conceptos se entienden paso a paso. Cubrimos el problema antes de exponer las soluciones.

Debido a esta forma incremental de presentar conceptos, este libro también le permite codificar a medida que aprende y reflexionar sobre los desafíos por sí mismo.

¿Es esto una guía o un libro?

Las páginas que tienes delante no se pueden llamar guía: no te llevará 15 o 30 minutos terminarlas. Pero este tampoco es el libro típico, en el que se recorren conceptos ilustrados con algunos fragmentos de código. En cambio, comienza con una versión del código que aún no es óptima y aprende cómo evolucionarlo, después de conocer los beneficios que puede extraer de ese proceso.

Eso no significa que no pueda simplemente sentarse y leerlo, pero es mejor si codifica al mismo tiempo y juega con las opciones y alternativas presentadas. Esa es la parte del libro que lo hace similar a una guía.

En cualquier caso, para simplificar, de aquí en adelante llamaremos a esto un libro.

Contenido

De los conceptos básicos a los temas avanzados

Este libro se centra primero en algunos conceptos básicos sobre cómo diseñar e implementar una aplicación Spring Boot lista para producción utilizando patrones de arquitectura conocidos (Capítulos 2 y 3). A partir de ahí, lo lleva a través del viaje de herramientas y marcos relacionados con microservicios con la introducción de una segunda pieza de funcionalidad en una aplicación Spring Boot diferente (Capítulos4 y 5). También le muestra cómo admitir un sistema distribuido de este tipo con pruebas de integración de un extremo a otro (capítulo6).

Si ya sabe cómo diseñar aplicaciones Spring Boot, puede leer rápidamente los capítulos 2 y 3 y céntrese más en la segunda parte del libro. Allí, cubrimos temas como descubrimiento de servicios, enrutamiento, diseño impulsado por eventos, pruebas con Cucumber, etc. Sin embargo, preste atención a la estrategia que establecimos en la primera parte: desarrollo impulsado por pruebas, el enfoque en el producto mínimo viable (MVP) y monolito primero.

Esqueleto con Spring Boot, el estilo profesional

Primero, el libro lo guía a través de la creación de una aplicación usando Spring Boot. Se centra principalmente en el lado del backend, pero creará una página web simple para demostrar cómo exponer la funcionalidad como una API REST.

Es importante señalar que no creamos "código de acceso directo" solo para ver Spring Boot en ejecución: ese no es el objetivo de este libro. Usamos Spring Boot como un vehículo para enseñar conceptos, pero podríamos usar cualquier otra técnica, y las ideas de este libro seguirían siendo válidas.

Aprenderá a diseñar e implementar la aplicación siguiendo el conocido patrón de tres niveles y tres capas. Lo hace con el apoyo de un ejemplo incremental, con código práctico. El resultado será más que suficiente para que comprenda la forma profesional de escribir aplicaciones.

Desarrollo basado en pruebas

Usamos TDD para cumplir con los requisitos previos presentados a las características técnicas (como debería hacer en la vida real). TDD es una técnica que a veces no se puede utilizar en el trabajo (por muchas razones diferentes, ninguna técnica). Pero este libro intenta mostrarlo de una manera que pueda ver los beneficios desde el principio: por qué siempre es una buena idea pensar en los casos de prueba antes de escribir su código. *AssertJ* y *Mockito* nos servirá para construir pruebas útiles de manera eficiente.

El plan es el siguiente: primero aprenderá a crear las pruebas, luego las hará fallar y finalmente implementará la lógica para que funcionen.

Conexión de microservicios

Una vez que tenga lista su primera aplicación, presentamos una segunda que interactuará con la funcionalidad existente. A partir de ese momento, tendrás un *Arquitectura de microservicios.* No tiene ningún sentido tratar de comprender las ventajas de los microservicios si solo tiene uno de ellos. Los escenarios de la vida real son siempre sistemas distribuidos con funcionalidad dividida en diferentes servicios. Como de costumbre, para que sea práctico, verá cómo el traslado a microservicios se adapta a sus necesidades.

El libro no solo cubre las razones para dividir el sistema, sino también las desventajas que conlleva esa elección. Y una vez que tome la decisión, aprenderá qué herramientas debe utilizar para que el sistema funcione como un todo y no como servicios aislados: descubrimiento de servicios, puerta de enlace API, equilibrio de carga y algunas otras herramientas de apoyo.

Sistema impulsado por eventos

Un concepto adicional que no siempre acompaña a los microservicios es un *arquitectura impulsada por eventos.* Este libro lo usa porque es un patrón que encaja muy bien en una arquitectura de microservicio, y usted hará su elección basándose en buenos ejemplos.

Esta forma de pensar asincrónica introduce nuevas formas de diseñar código; lo verá mientras codifica su proyecto, utilizando RabbitMQ para respaldarlo.

Pruebas de extremo a extremo

Si desea codificar su proyecto de manera profesional, debe tener una mentalidad lista para la producción, por lo que cubriremos esta funcionalidad con pruebas. Explicamos cómo abordar los más complicados en una arquitectura de microservicios: las pruebas de un extremo a otro. Usaremos Cucumber ya que es un marco que se adapta perfectamente a muchos proyectos, llenando el vacío entre los requisitos comerciales y el desarrollo de la prueba. Aunque nadie debería necesitar razones aquí para estar convencido de por qué es una buena idea tener una base de prueba adecuada, las explicamos para mantener contentos a los escépticos de las pruebas.

Resumen

Este capítulo presentó los objetivos principales de este libro: enseñarle los aspectos principales de una arquitectura de microservicios, comenzando de manera simple y luego aumentando su conocimiento a través del desarrollo de un proyecto de muestra.

También cubrimos brevemente el contenido principal del libro: demonolithfirst a microservicios con Spring Boot, desarrollo basado en pruebas, sistemas basados en eventos y pruebas de extremo a extremo con Cucumber.

El próximo capítulo comenzará con el primer paso de nuestra ruta de aprendizaje: una aplicación Spring Boot básica.

CAPITULO 2

La primavera básica

Aplicación de arranque

Requisitos comerciales

Podríamos empezar a escribir código directamente pero eso, incluso siendo pragmático, estaría lejos de ser un caso real. El software debe tener un objetivo: en este caso, lo hacemos por el mero hecho de aprender pero, de todos modos, le damos una razón. (uno ficticio). Este enfoque orientado a los requisitos se utiliza en todo el libro para hacerlo más práctico.

Queremos escribir una aplicación para animar a los usuarios a entrenar sus habilidades matemáticas todos los días. Para empezar, tendremos multiplicaciones de dos dígitos presentados a los usuarios, una cada vez que accedan a la página. Teclearán su alias (un nombre corto) y el resultado de la operación, y para eso deben usar solo cálculo mental. Después de que envíen los datos, se presentará un resultado de éxito o fracaso.

Para motivar a los usuarios, también presentaremos algunas técnicas simples de gamificación: un ranking de usuarios en función de los puntos que obtienen cuando intentan el cálculo todos los días, y también cuando lo logran. Mostraremos esto en la página de resultados.

Esta es la idea principal de toda la aplicación que crearemos (nuestro *visión)* y este libro emulará una forma ágil de trabajo en la que los requisitos vienen en forma de historias de usuario. Agile, a pesar de ser criticado por muchos desarrolladores de software, se ha convertido en la metodología estándar aplicada en

M. Macero, *LearnMicroservices con Spring Boot,*
https://doi.org/10.1007/978-1-4842-3165-4_2

una gran mayoría de empresas de TI. La realidad es que, cuando se implementa correctamente, es una forma de trabajo que permite a los equipos entregar software que se puede usar lo antes posible y obtener una valiosa retroalimentación de él.

Con el apoyo de Agile, comenzamos de manera simple y luego construimos sobre eso. Considere la primera historia de usuario aquí.

HISTORIA DE USUARIO 1

Como usuario de la aplicación, quiero que se me presente una multiplicación aleatoria que pueda resolver en línea; no es demasiado fácil, para poder usar el cálculo mental y hacer que mi cerebro funcione todos los días.

Para realizar este trabajo, dividiremos la historia del usuario en varias subtareas:

1. Crear un servicio básico con la lógica empresarial.
2. Cree un punto final API básico para acceder a este servicio (API REST).
3. Cree una página web básica para pedir a los usuarios que resuelvan ese cálculo.

La aplicación Skeleton

Aplicaciones delgadas frente a aplicaciones de la vida real

Lo primero que encontrarás si buscas *Tutorial de Spring Boot* en Google es el *Empezando* guía de Pivotal (ver https://spring.io/guides/ gs / spring-boot /). Siguiendo la guía, puede crear un HelloWorld (o *Saludos)* aplicación, pero eso no es emocionante cuando ya tienes algunas

experiencia en desarrollo de software. Si busca algo más desafiante, se encontrará buceando en muchos otros oficiales.*Empezando* guías que, a pesar de ser realmente útiles, están totalmente desconectadas y no proporcionan ejemplos de código de la vida real. Te ayudan a construir*aplicaciones delgadas.*

No te lo tomes a mal: estas guías son muy útiles para el trabajo diario. Por ejemplo, es posible que no recuerde cómo configurar un oyente RabbitMQ y, en ese caso, puede escanear estas guías para obtener una respuesta rápida. El objetivo principal de estas guías es proporcionarle un ejemplo rápido (que normalmente puede tardar unos 15 minutos) que cubre los conceptos básicos que necesita para configurar las diferentes funcionalidades de Spring Boot. Por eso, las aplicaciones a veces se construyen con accesos directos, como tener todo el código en la misma clase o insertar datos a través de corredores de línea de comandos.

Como ya sabe, el objetivo de este libro es ayudar a ir más allá, utilizando Spring Boot para crear aplicaciones más cercanas a los casos de la vida real. Aprende cómo combinar las diferentes tecnologías y configurar un código sin atajos, siguiendo buenas prácticas e incluyendo una cobertura de prueba adecuada.

Creando el esqueleto

¡Código práctico! Lo primero que debe hacer es crear un esqueleto de la aplicación Spring Boot que servirá como referencia durante el libro. Hay varias formas de hacer esto. Navega al*Spring Initializr* sitio web en http: // inicio. spring.io/y generar un proyecto a partir de ahí (ver Figura 2-1).

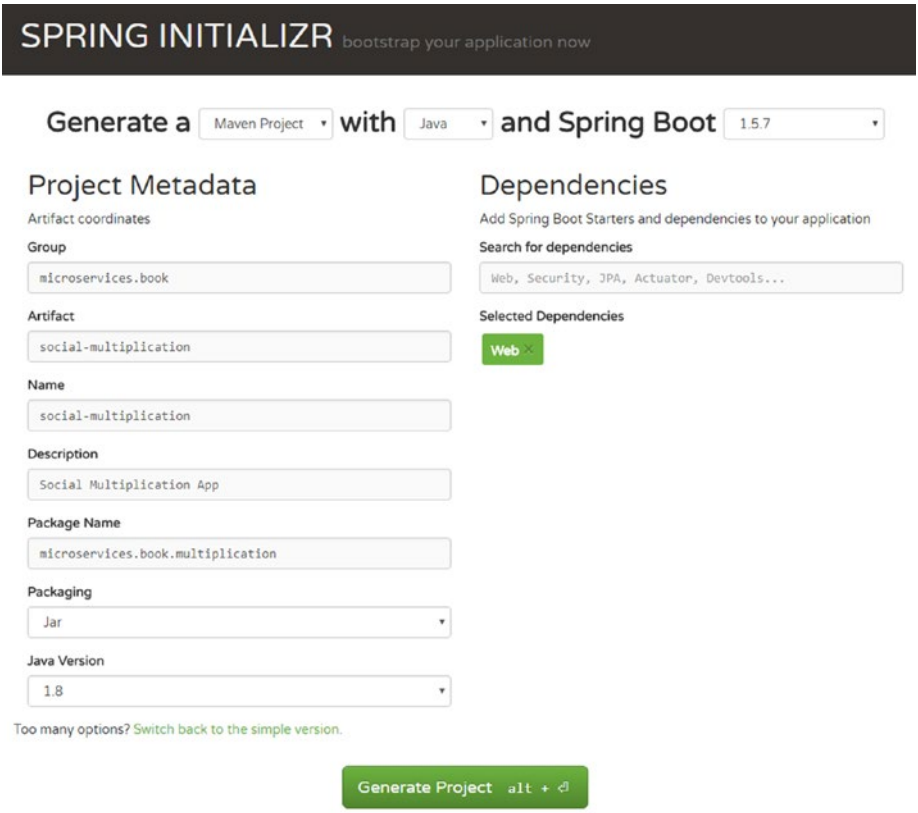

Figura 2-1. *El sitio web de Spring Initializr le ayuda a crear una aplicación básica*

Demos algunos valores al Grupo (microservices.book) y al artefacto (multiplicación social). Ahora haga clic en Cambiar a la versión completa y cambie el nombre del paquete a microservices.book.multiplication. Ingrese una descripción personalizada si lo desea. Luego, en dependencias, seleccione Web. El último paso es seleccionar la versión Spring Boot, 1.5.7 en este caso. Eso es todo lo que necesitas por ahora. Deje las otras configuraciones como están, ya que trabajará con Maven y Java.

Genere el proyecto y extraiga el contenido ZIP. Lasocialmultiplication-v1 carpeta contiene todo lo que necesita para ejecutar su aplicación,

incluyendo una envoltura de Maven (mvnw) que puede ejecutar desde la carpeta de origen. Si lo prefiere, puede utilizar su propia instalación de Maven.

Ahora puede usar su shell favorito para ejecutar la aplicación con este comando:

```
$ mvnw spring-boot: ejecutar
```

Se iniciará su aplicación. La última línea que debería ver es algo como esto:

```
m.book.SocialMultiplicationApplication : Empezado
SocialMultiplicationApplication en 2.385 segundos (JVM
ejecutándose para 6.07)
```

Esta aplicación, como habrás adivinado, aún no es práctica. No hay ninguna funcionalidad allí, aunque es *ocupando* un puerto en 8080. Pero es útil generar el proyecto esqueleto de esta manera, teniendo la configuración de Maven en su lugar y los paquetes raíz.

EJECUTANDO LA APLICACIÓN SPRING BOOT

De aquí en adelante, se asumirá que sabe cómo ejecutar la aplicación Spring Boot. También se recomienda que utilice su IDe preferido para trabajar con el código o importar los proyectos de Maven (eclipse, IntelliJ, conjunto de herramientas Spring, etc.). los ID más populares tienen una buena integración con Spring Boot y Maven y le permiten ejecutarlo directamente sin escribir nada en la línea de comandos. Si necesita más ayuda con esto, simplemente visite las guías oficiales para estos entornos de desarrollo integrados.

Calentamiento: algunos TDD en acción

Desarrollo impulsado por pruebas se basa en escribir las pruebas de la aplicación antes de la lógica del código principal, hacer que estas pruebas fallen primero y luego escribir el código para que pasen.

¿POR QUÉ TDD ES BUENO PARA LOS DESARROLLADORES?

Hay muchas razones por las que, pero la más importante es que tDD lo obliga a usted y a la persona de negocios a pensar en los requisitos previos en un *camino más profundo.* esto incluye pensar en lo que debería hacer el código en determinadas situaciones o casos de uso. Le ayudará a aclarar los requisitos previos vagos y rechazar los inválidos.

sin embargo, hay una idea generalmente asociada con tDD que a veces se lleva al extremo: la refactorización continua del código en varias iteraciones. Debería encontrar un equilibrio: no es una buena idea escribir código de mala calidad que no se pueda mantener solo para que las pruebas pasen y luego refactorizarlas.

Empecemos a pensar en lo que necesitamos. Empezaremos con MultiplicationServiceTest, en el que queremos comprobar que un Multiplicación ha sido generado. LaMultiplicación la clase se muestra en el listado 2-1.

CÓDIGO FUENTE DISPONIBLE CON EL LIBRO

Puede encontrar el código en este capítulo en el v1 repositorio en github en https://github.com/microservices-practical.

Listado 2-1. Multiplication.java (multiplicación social v1)

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

/**
 * Esta clase representa una multiplicación en nuestra aplicación.
 */
clase pública Multiplicación {
```

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

    // El resultado de la operación A * B
    privado resultado int;

    público Multiplicación (int factorA, int factorB) {
        esto.factorA = factorA;
        esto.factorB = factorB;
        esto.resultado = factorA * factorB;
    }

    público int getFactorA () {
        regreso factorA;
    }

    público int getFactorB () {
        regreso factorB;
    }

    público int getResult () {
        regreso resultado;
    }

    @Anular
    público String toString () {
        regreso "Multiplicación {"+
                "factorA =" + factorA + ",
                factorB =" + factorB + ",
                resultado (A * B) =" + resultado
                + '}';
    }

}
```

Sencillo. Es una clase básica y también contiene el resultado. No es necesario calcularlo todo el tiempo en toda la aplicación.

Definimos también la interfaz de servicio, como se muestra en el Listado 2-2.

Listado 2-2. MultiplicationService.java (multiplicación social v1)

```
paquete microservices.book.multiplication.service;

importar microservices.book.multiplication.domain.Multiplication;

interfaz pública MultiplicationService {

    /**
     * Crea un objeto de multiplicación con dos factores generados
       aleatoriamente
     * Entre 11 y 99.
     *
     * @regreso un objeto de multiplicación con factores aleatorios
     */
    Multiplicación createRandomMultiplication ();

}
```

Además, porque queremos *generar multiplicaciones aleatorias,* Creamos un servicio para proporcionar factores aleatorios (ver Listado 2-3). Eso nos ayudará a escribir pruebas adecuadas; sería mucho más difícil si usamosAleatorio dentro de la implementación del servicio.

Listado 2-3. RandomGeneratorService.java (multiplicación social v1)

```
paquete microservices.book.multiplication.service;

interfaz pública RandomGeneratorService {
```

```
    /**
     * @regreso un factor generado aleatoriamente. Siempre es
     un número entre 11 y 99.
     */
    int generateRandomFactor ();

}
```

Una vez que tenga las interfaces que necesita, puede escribir la primera versión de prueba, como se muestra en el Listado 2-4.

Listado 2-4. MultiplicationServiceTest.java (multiplicación social v1)

```
paquete microservices.book.multiplication.service;

importar microservices.book.multiplication.domain.Multiplication;
importar org.junit.Test;
importar org.junit.runner.RunWith;
importar org.springframework.beans.factory.annotation.Autowired;
importar org.springframework.boot.test.context.SpringBootTest;
importar org.springframework.boot.test.mock.mockito.MockBean;
importar org.springframework.test.context.junit4.SpringRunner;

importar estática org.assertj.core.api.Assertions.assertThat;
importar estática org.mockito.BDDMockito.given;

@RunWith (SpringRunner.class)
@SpringBootTest
clase pública MultiplicationServiceTest {

    @MockBean
    privado RandomGeneratorService randomGeneratorService;

    @Autowired
    privado MultiplicationService multiplicationService;
```

```
    @Prueba
    público void createRandomMultiplicationTest () {
        // dado (nuestro servicio de generador aleatorio simulado
        devolverá primero 50, luego 30)
        dado (randomGeneratorService.generateRandomFactor ()).
        willReturn (50, 30);

        // Cuándo
        Multiplicación multiplicación = multiplicationService.
        createRandomMultiplication ();

        // luego
        afirmar que (multiplicación.getFactorA ()). esEqualTo (50);
        afirmar que (multiplicación.getFactorB ()). esEqualTo (30);
        afirmar que (multiplicación.getResult ()). isEqualTo (1500);
    }
}
```

La @MockBean La anotación es importante en esta prueba: le dice a Spring que inyecte una RandomGeneratorService bean, en lugar de dejar que busque una implementación adecuada de la interfaz (que aún no existe).

Estamos utilizando algunos de los beneficios de Mockito y Spring Boot para realizar una prueba unitaria simple y concisa. También estamos utilizando el desarrollo impulsado por el comportamiento (BDD, compatible con MockitoBDD) para definir qué debería suceder cuando RandomGeneratorService se llama. Eso hace que la prueba sea aún más fácil de leer, lo cual es excelente para el objetivo que tenemos: obtener ayuda de la persona que define nuestros requisitos para crear los casos de uso.

Si solo escribimos estas tres clases y ejecutamos la prueba, obviamente fallará, ya que no hay implementación de MultiplicationService Probar. Una vez más, ese es exactamente el punto de TDD: escribimos las especificaciones primero, luego las validamos con un analista de negocios (como un Product Owner en Scrum; consulte*https://tpd.io/prd-own),* y luego enumere qué otros casos deben cubrirse. Todo esto sin implementación de la solución.

Una vez que la prueba (requisito) está clara, escribimos la solución, como se muestra en el Listado 2-5.

Listado 2-5. MultiplicationServiceImpl.java (multiplicación social v1)

```
paquete microservices.book.multiplication.service;

importar microservices.book.multiplication.domain.Multiplication;
importar org.springframework.beans.factory.annotation.Autowired;
importar org.springframework.stereotype.Service;

@Servicio
clase MultiplicationServiceImpl implementos
MultiplicationService {

    privado RandomGeneratorService randomGeneratorService;

    @Autowired
    público MultiplicationServiceImpl (RandomGeneratorService
    randomGeneratorService) {
        esto.randomGeneratorService = randomGeneratorService;
    }

    @Anular
    público Multiplicación createRandomMultiplication () {
        int factorA = randomGeneratorService.
        generateRandomFactor ();
        int factorB = randomGeneratorService.
        generateRandomFactor ();
        volver nuevo Multiplicación (factorA, factorB);
    }
}
```

Aquí tampoco hay sorpresas; es sencillo. Ahora puede ejecutar la prueba con éxito con la siguiente línea de comando (o también puede usar su IDE preferido):

$ mvnw -Dtest = Prueba MultiplicationServiceTest

USTED VAQUERO! LA APLICACIÓN FALLA ...

Si intenta ejecutar todas las pruebas en lugar de solo MultiplicationServiceTest, obtendrás un errorNo hay bean calificado de tipo 'microservicios. book.multiplication.service.RandomGeneratorService ' disponible). la razón es que, de forma predeterminada, cuando crea la aplicación desde Spring Initializr, el paquete incluye un vacío SocialMultiplicationApplicationTests que intenta cargar el lleno contexto de la aplicación. lo mismo sucede si intenta ejecutar la aplicación. Al cargar el contexto, Spring intentará encontrar una implementación de RandomGeneratorService inyectar, pero no hay ninguno. esto no significa que esté haciendo un desarrollo cowboy, solo está utilizando una ventaja de tDD: prueba a medida que desarrolla, incluso si aún no tiene la aplicación completa en funcionamiento.

Repasemos las ventajas del enfoque TDD:

- Traducimos los requisitos al código (creando nuestra prueba), y eso nos obliga a pensar en lo que necesitamos y lo que no necesitamos. Hasta ahora solo necesitamos generar una multiplicación aleatoria; es nuestro primer requisito comercial.
- Creamos código comprobable. Imagínese que hubiéramos comenzado a codificar sin tener la prueba. Habría sido más fácil incluir la lógica de generación aleatoria directamente dentro delMultiplicationService implementación, lo que hace que sea realmente difícil probar

después, ya que la prueba usaría números aleatorios, por lo que sería impredecible. Al tener que escribir la prueba con anticipación, nos obligamos a pensar en una buena manera de verificar la funcionalidad, generando la lógica separada en RandomGeneratorService.

- Tenga en cuenta que no necesitamos escribir la implementación de RandomGeneratorService. Podemos centrarnos primero en lo más importante y luego implementar los servicios de ayuda. Dejamos elRandomGeneratorService implementación para el próximo capítulo.

Resumen

El objetivo principal de este capítulo fue presentar los requisitos y el enfoque de desarrollo basado en pruebas que seguirá en este libro. Creó una aplicación Spring Boot desarrollando algunas funciones básicas usando TDD.

El captulo tambin prepar el escenario para que pienses en un *Ágil* forma, que se utiliza cada vez más en las empresas de software por sus beneficios. Se tomó un tiempo para refinar su primer requisito comercial, lo dividió en subtareas y pensó en una primera*prueba de unidad.*

El siguiente capítulo se centra mucho más en el trabajo práctico: creará la primera versión completa de esta aplicación, incluida una interfaz de usuario simple. Lo hará utilizando buenas prácticas de diseño desde el principio: un software de tres niveles y capas que le dará la flexibilidad para hacer evolucionar su aplicación.

CAPÍTULO 3

Un verdadero de tres niveles

Bota de primavera

Solicitud

Introducción

La arquitectura de Amultitier proporcionará a nuestra aplicación un aspecto más listo para la producción. La mayoría de las aplicaciones del mundo real siguen este patrón de arquitectura y, para ser más específicos, el diseño de tres niveles es el más popular y ampliamente extendido entre las aplicaciones web. El tres

los niveles son:

- *Nivel de cliente:* Responsable de la interfaz de usuario. Normalmente, lo que llamamos frontend.
- *Nivel de aplicación:* Contiene toda la lógica empresarial junto con las interfaces para interactuar con ella y las interfaces de datos para la persistencia. Esto se asigna con lo que llamamos backend.
- *Nivel de almacenamiento de datos:* Es la base de datos, el sistema de archivos, etc., lo que conserva los datos de la aplicación.

M. Macero, *LearnMicroservices con Spring Boot,*
https://doi.org/10.1007/978-1-4842-3165-4_3

En este libro nos centramos principalmente en el nivel de aplicación, aunque también usaremos los otros dos. Si ahora nos acercamos, ese nivel de aplicación se diseña comúnmente usando tres capas:

- *Capa empresarial:* Las clases que modelan nuestro dominio y las especificidades comerciales. Es donde reside la inteligencia de la aplicación. Normalmente, estará compuesto por entidades (nuestroMultiplicación) y Servicios que brindan lógica comercial (como nuestro MultiplicationService). A veces, esta capa se divide en dos partes: dominios (entidades) y aplicaciones (servicios).
- *Capa de presentación:* En nuestro caso, estará representado por el Controlador clases, que proporcionarán funcionalidad al cliente Web. Nuestra implementación de API REST residirá aquí.
- *Capa de datos:* Será responsable de mantener nuestras entidades en un almacenamiento de datos, generalmente una base de datos. Por lo general, puede incluir un objeto de acceso a datos (DAO) clases, que trabajan con representación directa del modelo de base de datos, o Repositorio clases, que están centradas en el dominio y se traducen de los dominios a la capa de la base de datos (para que puedan usar DAO cuando no coincidan).

El patrón de arquitectura que se muestra en la Figura 3-1 se utiliza en nuestra aplicación mientras desarrollamos las funcionalidades requeridas.

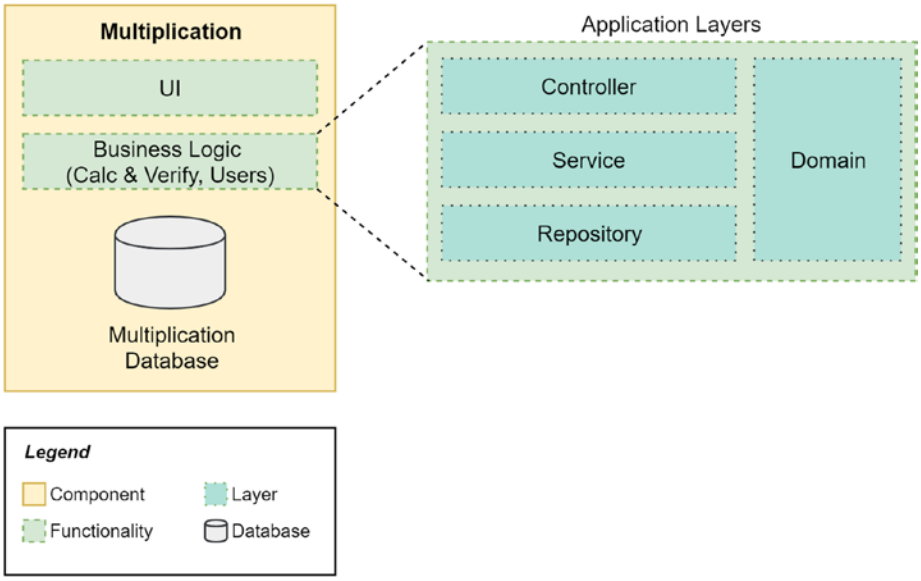

Figura 3-1. *El patrón de arquitectura de la aplicación*

Las ventajas de utilizar esta arquitectura de software están intrínsecamente relacionadas con el hecho de desacoplar capas. Resumamos tres ventajas importantes:

- La parte del dominio está aislada e independiente de la solución, en lugar de mezclarse con la interfaz o las especificaciones de la base de datos.
- Las capas no comerciales son intercambiables (como, por ejemplo, cambiar la base de datos para una solución de almacenamiento de archivos o cambiar de REST a cualquier otra interfaz).
- Hay una clara separación de responsabilidades: una clase para manejar el almacenamiento de la base de datos de los objetos, una clase separada para la implementación de la API REST y otra clase para la lógica empresarial.

Spring es una excelente opción para construir este tipo de arquitectura, con muchas características listas para usar que nos ayudarán a crear fácilmente una aplicación de tres niveles lista para producción.

Completando los conceptos básicos

Antes de continuar con las capas ... ¿nos perdimos algo? Dejamos atrás la implementación de nuestroRandomGeneratorService. Creemos la prueba para ello, ya que sabemos qué esperar. Ver listado3-1.

CÓDIGO FUENTE DISPONIBLE CON EL LIBRO: V2

Puede encontrar el código del capítulo en el v2 repositorio en github: https://github.com/microservices-practical.

Listado 3-1. RandomGeneratorServiceTest.java (multiplicación social v2)

```
paquete microservices.book.multiplication.service;

importar org.junit.Test;
importar org.junit.runner.RunWith;
importar org.springframework.beans.factory.annotation.Autowired;
importar org.springframework.boot.test.context.SpringBootTest;
importar org.springframework.test.context.junit4.SpringRunner;

importar java.util.List;
importar java.util.stream.Collectors;
importar java.util.stream.IntStream;
```

```
importar estática org.assertj.core.api.Assertions.assertThat;

@RunWith (SpringRunner.class)
@SpringBootTest
clase pública RandomGeneratorServiceTest {

    @Autowired
    privado RandomGeneratorService randomGeneratorService;

    @Prueba
    público vacío generateRandomFactorIsBetweenExpectedLimits ()
    lanza Excepción {
        // cuando se genera una buena muestra de factores generados
        aleatoriamente
        List <Integer> randomFactors = IntStream.range (0, 1000)
                . map (i -> randomGeneratorService.
                generateRandomFactor ())
                . boxed (). collect (Collectors.toList ());

        // entonces todos deberían estar entre 11 y 100 //
        porque queremos un cálculo de complejidad media
        assertThat (randomFactors) .containsOnlyElementsOf
        (IntStream.range (11, 100)
                . boxed (). collect (Collectors.toList ()));
    }

}
```

Usamos un Java 8 Arroyo de los primeros 1000 nmeros tomimic un por círculo. Luego, transformamos cada número conmapa a un azar En t factor, encajonamos cada uno a un Entero objeto, y finalmente nosotros recoger en una lista. La prueba verifica que todos estén dentro del rango esperado que definimos usando un enfoque similar.

PREPARACIÓN PARA LA PRODUCCIÓN: NO USE EN EXCESO LAS PRUEBAS DE SPRINGBOOT

Hemos creado ambas pruebas como @SpringBoot pruebas, corriendo con SpringRunner, lo que hace que se inicialice el contexto de la aplicación, por lo que se inyectan los beans. afortunadamente, Spring almacena y reutiliza el contexto, por lo que solo se carga una vez por suite.[1]

este es un ejemplo de cómo las guías de Internet pueden ser confusas a veces. No necesitamos la inyección de dependencias ni el contexto de la aplicación para probar las funcionalidades de estas clases; en estas situaciones, es mejor no usar @SpringBoot y simplemente pruebe las clases de implementación: build *verdadero* pruebas unitarias que verifican solo una clase. Usamos pruebas con un contexto Spring para un tipo diferente de prueba: pruebas de integración, cuyo objetivo es verificar las interacciones entre más de una clase. incluso con la reutilización del contexto, si usamos @SpringBootTest, estamos perdiendo tiempo cargando recursos y tenemos que asegurarnos de revertir las transacciones y limpiar el contexto de la aplicación Spring para evitar efectos secundarios.

Manteniendo eso en mente, creemos una clase adicional para una prueba unitaria real que no necesita un contexto Spring para ejecutarse (tenga en cuenta el cambio de nombre ya que estamos probando directamente la implementación). Podemos eliminar de forma segura RandomGeneratorServiceTest ya que cubre la misma prueba (aunque puede verlos a ambos en la v2 carpeta con fines educativos). Ver listado3-2.

Listado 3-2. RandomGeneratorServiceImplTest.java (socialmultiplication v2)

```
paquete microservices.book.multiplication.service;

importar org.junit.Before;
importar org.junit.Test;
```

[1] https://docs.spring.io/spring/docs/current/spring-frameworkreference/testing.html#testing-ctx-management

```
importar java.util.List;
importar java.util.stream.Collectors;
importar java.util.stream.IntStream;

importar estática org.assertj.core.api.Assertions.assertThat;

clase pública RandomGeneratorServiceImplTest {

    privado RandomGeneratorServiceImpl
    randomGeneratorServiceImpl;

    @Antes
    público configuración vacía () {
        randomGeneratorServiceImpl = nuevo
        RandomGeneratorServiceImpl ();
    }

    @Prueba
    público vacío generateRandomFactorIsBetweenExpectedLimits ()
    lanza Excepción {
        // cuando se genera una buena muestra de factores generados
        aleatoriamente
        List <Integer> randomFactors = IntStream.range (0, 1000)
                . mapa (i -> randomGeneratorServiceImpl.
                generateRandomFactor ())
                . boxed (). collect (Collectors.toList ());

        // entonces todos deberían estar entre 11 y 100 //
        porque queremos un cálculo de complejidad media
        assertThat (randomFactors) .containsOnlyElementsOf (IntStr
        eam.range (11, 100)
                . boxed (). collect (Collectors.toList ()));
    }

}
```

¿Ves alguna desventaja? Bueno, hay una pequeña: ahora no podemos evitar generar laRandomGeneratorServiceImpl clase, pero la implementación de nuestro método puede volver 0 para empezar. Ver listado3-3.

Listado 3-3. RandomGeneratorServiceImpl.java: Solución temporal (multiplicación social v2)

```
paquete microservices.book.multiplication.service;

importar org.springframework.stereotype.Service;
importar java.util.Random;

@Servicio
clase RandomGeneratorServiceImpl implementos
RandomGeneratorService {
    @Anular
    público int generateRandomFactor () {
        regreso 0;
    }
}
```

Luego tenemos nuestra prueba fallida compilando y esperando que escribamos una implementación adecuada, como se muestra en Listado 3-4.

Listado 3-4. Consola: Ejecución de RandomGeneratorServiceImplTest (social-multiplication v2)

```
$ mvnw prueba -Dtest = RandomGeneratorServiceImplTest
[...]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 PRUEBAS
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Ejecutando microservices.book.multiplication.service.
RandomGeneratorServiceImplTest
```

```
Pruebas ejecutadas: 1, Fallos: 1, Errores: 0, Omitidos: 0, Tiempo transcurrido:
0.109 seg <<< ¡FALLO! - en microservices.book.multiplication.
service.RandomGeneratorServiceImplTest
generateRandomFactorIsBetweenExpectedLimits (microservices.book.
multiplication.service.RandomGeneratorServiceImplTest) Tiempo
transcurrido: 0.108 seg <<< ¡FALLO!
java.lang.AssertionError: [...]
```

¡Hagámoslo pasar! Necesitamos darle un poco más de lógica que simplemente devolver cero, como se muestra en el Listado3-5.

Listado 3-5. RandomGeneratorServiceImpl.java (multiplicación social v2)

```
paquete microservices.book.multiplication.service;

importar org.springframework.stereotype.Service;
importar java.util.Random;

@Servicio
clase RandomGeneratorServiceImpl implementos
RandomGeneratorService {

    final static int MINIMUM_FACTOR = 11;
    final static int MAXIMUM_FACTOR = 99;

    @Anular
    público int generateRandomFactor () {
        volver nuevo Aleatorio (). NextInt ((MAXIMUM_FACTOR - MINIMUM_
        FACTOR) + 1) + MINIMUM_FACTOR;
    }
}
```

Ahora, si ejecutamos prueba mvnw -Dtest = Prueba RandomGeneratorServiceImpl de nuevo, la prueba pasará. ¡Estupendo! Y, dado que tenemos un mejor enfoque para realizar pruebas unitarias, debemos hacer lo mismo conMultiplicationService.
Vamos a crear MultiplicationServiceImplTest y aplique este conocimiento allí también, como se muestra en el Listado 3-6.

Listado 3-6. MultiplicationServiceImplTest.java (multiplicación social v2)

```
paquete microservices.book.multiplication.service;

importar microservices.book.multiplication.domain.Multiplication;
importar org.junit.Before;
importar org.junit.Test;
importar org.mockito.Mock;
importar org.mockito.MockitoAnnotations;

importar estática org.assertj.core.api.Assertions.assertThat;
importar estática org.mockito.BDDMockito.given;

clase pública MultiplicationServiceImplTest {

    privado MultiplicationServiceImpl
    multiplicationServiceImpl;

    @Burlarse de
    privado RandomGeneratorService randomGeneratorService;

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

```
    @Prueba
    público void createRandomMultiplicationTest () {
        // dado (nuestro servicio de generador aleatorio simulado
        devolverá primero 50, luego 30)
        dado (randomGeneratorService.generateRandomFactor ()).
        willReturn (50, 30);

        // Cuándo
        Multiplicación multiplicación =
        multiplicationServiceImpl.createRandomMultiplication ();

        // afirmar
        afirmar que (multiplicación.getFactorA ()). esEqualTo (50);
        afirmar que (multiplicación.getFactorB ()). esEqualTo (30);
        afirmar que (multiplicación.getResult ()). isEqualTo (1500);
    }
}
```

Tenga en cuenta que no inyectamos amock bean con @MockBean pero solo usa el simple @Burlarse de anotación para crear un servicio falso, que luego usamos programáticamente para construir el MultiplicationServiceImpl objeto.

En este punto, también podemos ejecutar el conjunto completo de pruebas en la aplicación con prueba mvnw, y ver cómo pasan todos.

Diseñando el dominio

Antes de iniciar el proceso de desarrollo, es importante tener una imagen clara del dominio empresarial, incluidos los diferentes *objetos (*en el sentido más genérico de la palabra) puede identificar en su sistema y cómo se relacionan. Este ejercicio debe realizarse lo antes posible cuando diseñe software. Será el corazón de su sistema y, por lo tanto, la parte más difícil de cambiar.

Dados nuestros requisitos, podemos identificar los siguientes objetos comerciales:

- Multiplicación: Contiene los factores de la operación.
- Usuario: Identifica al usuario que intentará solucionar un Multiplicación.
- MultiplicationResultAttempt: Contiene una referencia a la Multiplicación y una referencia a la Usuario, junto con el valor enviado (el intento de resolver la operación) y el resultado real.

CÓDIGO FUENTE DISPONIBLE CON EL LIBRO: V3

Puede encontrar estas entidades de dominio modeladas en el v3 repositorio en github: https://github.com/microservices-practical.

Inmutabilidad y Lombok

Veremos en un momento que el Multiplicación la clase es final, y también lo son sus campos, a los que solo se puede acceder con *captadores.* eso hace que nuestra clase sea inmutable. La inmutabilidad tiene muchos beneficios, el más importante es que le ahorra muchos problemas potenciales al trabajar con un sistema de subprocesos múltiples. si quieres saber más sobre las ventajas de inmutabilidad, visita https://en.wikipedia.org/wiki/Immutable_object.

También incluimos lombok en nuestro código, agregando una dependencia adicional en el pom.xml archivo (ver https://projectlombok.org/). es una anotación procesador que generará código antes de que se ejecute el compilador. ¿Cuáles son las ventajas de eso? Puede mantener sus clases pequeñas, eliminando todas las partes estándar: captadores, constructores,toString, hashCode, equals, etc., todos se reemplazan con anotaciones. incluso hay algunos atajos para agrupar varios de ellos (como @Datos). la principal desventaja es que si su iDe no tiene un complemento que admita lombok, el asistente de código no funcionará y el compilador integrado de iDe se quejará. sin embargo, hay complementos

para los más importantes. (Desde la página de inicio, use el menú para navegar e instalar➤ iDes ➤ su ID preferido.)

Usaremos lombok de aquí en adelante, ya que también es más conveniente cuando mira el código dentro de estas páginas, pero depende de usted mantener sus poJos con todo el resto del código si así lo desea.

Agregar Lombok al proyecto es tan fácil como incluir una nueva dependencia en el pom.xml archivo, como se muestra en el listado 3-7.

Listado 3-7. pom.xml: Añadiendo Lombok (social-multiplication v3)

```
<dependencia>
  <groupId>org.projectlombok </groupId>
 <artifactId>lombok </artifactId> <versión>
 1.16.12 </versión> </dependency>
```

Listados 3-8 mediante 3-10 muestre cómo implementar las entidades de dominio en Java usando Lombok.

Listado 3-8. Multiplication.java (social-multiplication v3)

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

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

/**
 * Esto representa una multiplicación (a * b).
 */
@RequiredArgsConstructor
@Getter
@Encadenar
```

```
@EqualsAndHashCode
público final clase Multiplicación {

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

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

Listado 3-9. User.java (multiplicación social v3)

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

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

/**
 * Almacena información para identificar al usuario.
 */
@RequiredArgsConstructor
@Getter
@Encadenar
@EqualsAndHashCode
público final clase Usuario {

    privado alias de cadena final;

    // Constructor vacío para (des) serialización JSON
    protegido Usuario () {
```

```
        alias = nulo;
    }
}
```

Listado3-10. MultiplicationResultAttempt.java (social-multiplicationv3)

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

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

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

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

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

}
```

- @RequiredArgsConstructor genera un constructor tomando todos los final campos.
- @Adquiridor genera todos los captadores para nuestros campos.
- @Encadenar incluye un amigable para los humanos Encadenar() método en nuestra clase.
- @EqualsAndHashCode crea el es igual a () y código hash() métodos.

La capa de lógica empresarial

Una vez que haya definido el modelo de dominio, es hora de pensar en la otra parte de la lógica empresarial: *servicios de aplicación.* Echando un vistazo a nuestros requisitos, necesitamos:

- Alguna funcionalidad para comprobar si un intento es correcto o no
- Una forma de generar una multiplicación de complejidad media

Incluiremos esta nueva lógica en el Servicio capa. Ya hicimos la parte de generar multiplicaciones en el capítulo anterior (ubicado en elServicio paquete de capas), por lo que Listado 3-11 muestra cómo implementar la lógica para verificar los intentos (el checkAttempt método).

Listado 3-11. MultiplicationService.java (social-multiplication v3)

```
paquete microservices.book.multiplication.service;

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

```
interfaz pública MultiplicationService {

    / **
     * Genera un {@Enlace Objeto de multiplicación}.
     *
     * @regreso una multiplicación de números generados aleatoriamente
     * /
    Multiplicación createRandomMultiplication ();

    / **
     * @regreso verdadero si el intento coincide con el resultado de la
     *              multiplicación, falso en caso contrario.
     * /
    boolean checkAttempt (final MultiplicationResultAttempt
    resultAttempt);
}
```

Como estamos haciendo TDD, crearemos una implementación ficticia que siempre resultará en un intento incorrecto, como se muestra en el listado. 3-12.

Listado 3-12. Solución temporal MultiplicationServiceImpl.java (social-multiplication v3)

```
@Anular
público boolean checkAttempt (final MultiplicationResultAttempt
resultAttempt) {
    falso retorno;
}
```

Luego codificamos nuestros nuevos métodos de prueba, sabiendo que uno de ellos (verificar un resultado exitoso) fallará, por lo que debemos volver a la implementación y hacer que pase de acuerdo con los casos de uso (ver Listado 3-13). ¡De nuevo TDD en la práctica!

Listado 3-13. MultiplicationServiceImplTest.java
(multiplicación social v3)

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

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

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

@Prueba
público void checkWrongAttemptTest () {
    // dado
    Multiplicación multiplicación = nuevo Multiplicación (50, 60); Usuario
    usuario =nuevo Usuario ("john_doe"); MultiplicationResultAttempt
    intento =nuevo ResultAttempt de multiplicación (usuario,
    multiplicación, 3010);

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

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

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