Los datos

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

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

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

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

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

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

importar java.util.List;

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

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

```
    List <BadgeCard> findByUserIdOrderByBadgeTimestampDesc   (ID
    de usuario largo final);

}
```

ScoreCardRepository es un poco más interesante ya que en este caso necesitamos una agregación para la puntuación total. En aras del aprendizaje, el resultado de la consulta se asignará a un nuevo objeto: unLeaderBoardRow. Este es un ejemplo de cómo *el modelo de negocio (la tabla de clasificación) no necesita compararse uno a uno con el modelo de datos (una agregación de puntajes).* Ver listado 4-13.

Listado 4-13. ScoreCardRepository.java (gamificación v5)

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

importar microservices.book.gamification.domain.LeaderBoardRow;
importar microservices.book.gamification.domain.ScoreCard;
importar org.springframework.data.jpa.repository.Query;
importar org.springframework.data.repository.CrudRepository;
importar org.springframework.data.repository.query.Param;

importar java.util.List;

/**
 * Maneja operaciones CRUD con ScoreCards
 */
interfaz pública ScoreCardRepository se extiende
CrudRepository <ScoreCard, Long> {

    /**
     * Obtiene la puntuación total de un usuario determinado, que es la suma de
     las puntuaciones de todas sus ScoreCards.
     * @param userId la identificación del usuario para el que se debe
     recuperar la puntuación total
     * @regreso la puntuación total para el usuario dado
     */
```

```
    @Query ("SELECCIONE SUM (s.score) FROM microservices.book.
    Gamification.domain.ScoreCard s WHERE s.userId =: userId
    GROUP BY s.userId")
    int getTotalScoreForUser (@Param ("userId") final Long
    userId);

    / **
      * Recupera una lista de {@Enlace LeaderBoardRow} representa la
        tabla de clasificación de usuarios y su puntuación total.
      * @regreso la tabla de clasificación, ordenada primero por la puntuación más alta.
      * /
    @Query ("SELECCIONE NUEVOS microservicios.book.gamification.domain.
    LeaderBoardRow (s.userId, SUM (s.score))" +
                "FROM microservices.book.gamification.domain.
                ScoreCard s" +
                "GROUP BY s.userId ORDER BY SUM (s.score) DESC")
    List <LeaderBoardRow> findFirst10 ();

    / **
      * Recupera todas las ScoreCards de un usuario determinado, identificado
        por su identificación de usuario.
      * @param userId la identificación del usuario
      * @regreso una lista que contiene todas las ScoreCards para el usuario
        dado, ordenadas por las más recientes.
      * /
    List <ScoreCard> findByUserIdOrderByScoreTimestampDesc   (ID de usuario
    largo final);
}
```

Esta clase se acerca a un ejemplo de la vida real, en el que el *mágico* Los métodos de consulta basados en patrones de nombres no son suficientes para lograr nuestro objetivo. Necesitamos utilizar consultas escritas en Java Persistence Query Language (JPQL). No son consultas nativas (específicas del subyacente

motor de base de datos) pero genéricos, basados en el código. Ese es un gran poder de JPQL: podemos usar consultas y aún mantener la abstracción de la implementación de la base de datos. JPQL proporciona un conjunto de funciones, operadores, expresiones, etc., que deberían ser suficientes para realizar la mayoría de las consultas. Sin embargo, tenga en cuenta que este lenguaje es una especificación y hay algunas implementaciones de bases de datos que pueden no admitirlo por completo.

Nuestra consulta creará nuevos LeaderBoardRow objetos para los resultados de la consulta, utilizando el userId y la suma de la puntuación de un usuario determinado. También cubre la clasificación de los resultados con la puntuación más alta en primer lugar.

La lógica empresarial

Habrá dos partes dominantes del código a cargo de la lógica empresarial del microservicio de gamificación:

- La GameService interfaz y su implementación
  GameServiceImpl: Se utiliza para calcular la puntuación y las insignias, en función de los intentos recibidos.
- LeaderBoardService y LeaderBoardServiceImpl:
  Se utiliza para recuperar los 10 usuarios principales con la puntuación más alta.

Primero definamos las interfaces. Ver listados4-14 y 4-15.

Listado 4-14. GameService.java (gamificación v5)

```
paquete microservices.book.gamification.service;

importar microservices.book.gamification.domain.GameStats;

/**
 * Este servicio incluye la lógica principal para gamificar el sistema.
 */
interfaz pública GameService {
```

```
    /**
     * Procesar un nuevo intento de un usuario determinado.
     *
     * @param userId la identificación única del usuario
     * @param intentId la identificación del intento, se puede utilizar para recuperar datos
    adicionales si es necesario
     * @param correcto indica si el intento fue correcto
     *
     * @regreso a {@Enlace GameStats} objeto que contiene la nueva
    puntuación y las tarjetas de insignia obtenidas
     */
    GameStats newAttemptForUser (ID de usuario largo, ID de intento largo,
    booleano correcto);

    /**
     * Obtiene las estadísticas del juego de un usuario determinado.
     * @param userId el usuario
     * @regreso las estadísticas totales de ese usuario
     */
    GameStats retrieveStatsForUser (Long userId);

}
```

Listado 4-15. LeaderBoardService.java (gamificación v5)

```
paquete microservices.book.gamification.service;

importar microservices.book.gamification.domain.LeaderBoardRow;

importar java.util.List;

/**
 * Proporciona métodos para acceder a la tabla de clasificación con usuarios y puntuaciones.
 */
interfaz pública LeaderBoardService {
```

```
    /**
     * Recupera la tabla de clasificación actual con los usuarios con la mejor
       puntuación.
     * @regreso los usuarios con la puntuación más alta
     */
    List <LeaderBoardRow> getCurrentLeaderBoard ();
}
```

EJERCICIO

¡Es hora de tDD! ahora que sabe cómo se ven las interfaces, escriba las pruebas unitarias para verificar su funcionalidad antes de escribir la implementación real. recuerda estos consejos:

- Primero, escriba clases de implementación vacías de las interfaces. Use
- la estructura when / given / then para una mejor legibilidad.
- Cubre todos los casos de uso, incluidos los escenarios de la insignia: primero ganado, bronce (100 puntos o más), plata (500 puntos o más) y oro (999 puntos o más).

Puede encontrar las pruebas completadas en el v5 repositorio de código, dentro del gamificación proyecto.

La lógica principal del juego está dentro del GameServiceImpl clase. Más específicamente, está dentro delnewAttemptForUser () método. Repasemos la implementación, que se muestra en el listado.4-16.

Listado 4-16. GameServiceImpl.java newAttemptForUser () (gamificación v5)

```
@Servicio
@ Slf4j
clase GameServiceImpl implementos GameService {
```

```
privado ScoreCardRepository scoreCardRepository;
privado BadgeCardRepository badgeCardRepository;

GameServiceImpl (ScoreCardRepository scoreCardRepository,
                 BadgeCardRepository badgeCardRepository,
    esto.scoreCardRepository = scoreCardRepository;
    esto.badgeCardRepository = badgeCardRepository;
}

@Anular
público GameStats newAttemptForUser (final Long userId,
                                     ID de intento largo final,
                                     final booleano correcto) {
    // Para la primera versión daremos puntos solo si es correcta

    Si(correcto) {
        ScoreCard scoreCard = nuevo ScoreCard (userId,
        intentId);
        scoreCardRepository.save (scoreCard); log.info ("El
        usuario con id {} obtuvo {} puntos por id de intento
        {}",
                userId, scoreCard.getScore (), intentId); Lista
        <BadgeCard> badgeCards =
        processForBadges (userId, intentId);
        volver nuevo GameStats (userId, scoreCard.getScore (),
                badgeCards.stream ().
                mapa (BadgeCard :: getBadge)
                        . recopilar (Collectors.toList ()));
    }
    regreso GameStats.emptyStats (userId);
}
```

```
/**
 * Comprueba la puntuación total y las diferentes tarjetas de puntuación
   obtenidas.
 * Dar nuevas insignias en caso de que se cumplan sus condiciones.
 */
privado Lista <BadgeCard> processForBadges (final Long userId,
                                            final largo
                                            intentId) {
    Lista <BadgeCard> badgeCards = nuevo ArrayList <> ();

    int totalScore = scoreCardRepository.
    getTotalScoreForUser (userId);
    log.info ("La nueva puntuación para el usuario {} es {}",
    userId, totalScore);

    Lista <ScoreCard> scoreCardList = scoreCardRepository
            . findByUserIdOrderByScoreTimestampDesc   (userId);
    Lista <BadgeCard> badgeCardList = badgeCardRepository
            . findByUserIdOrderByBadgeTimestampDesc   (userId);

    // Insignias en función de la puntuación
    checkAndGiveBadgeBasedOnScore (badgeCardList,
            Badge.BRONZE_MULTIPLICATOR, totalScore, 100,
            userId)
            . ifPresent (badgeCards :: add);
    checkAndGiveBadgeBasedOnScore (badgeCardList,
            Insignia.SILVER_MULTIPLICATOR, totalScore, 500,
            userId)
            . ifPresent (badgeCards :: add);
    checkAndGiveBadgeBasedOnScore (badgeCardList,
            Badge.GOLD_MULTIPLICATOR, totalScore, 999,
            userId)
            . ifPresent (badgeCards :: add);
```

```
        // Primera insignia ganada
        Si(scoreCardList.size () == 1 &&
                ! containsBadge (badgeCardList, Badge.
                FIRST_WON)) {
            BadgeCard firstWonBadge = giveBadgeToUser (Insignia.
            FIRST_WON, userId);
            badgeCards.add (firstWonBadge);
        }

        regreso badgeCards;
}

@Anular
público GameStats retrieveStatsForUser (final Long userId) {
        int score = scoreCardRepository.
        getTotalScoreForUser (userId);
        Lista <BadgeCard> badgeCards = badgeCardRepository
                . findByUserIdOrderByBadgeTimestampDesc   (userId);
        volver nuevo GameStats (userId, score, badgeCards.stream ()
                . map (BadgeCard :: getBadge) .collect
                (Collectors.toList ()));
}

/ **
  * Método de conveniencia para comparar la puntuación actual
  * los diferentes umbrales para ganar insignias.
  * También asigna una insignia al usuario si se cumplen las condiciones.
  * /
privado <BadgeCard> opcional checkAndGiveBadgeBasedOnScore (
            Lista final <BadgeCard> badgeCards, insignia final
            Badge,
            final int score, final int scoreThreshold, final Long
            userId) {
```

```
        Si(score> = scoreThreshold &&!
        containsBadge (badgeCards, badge)) {
            regreso Opcional. De (giveBadgeToUser (insignia, userId));
        }
        regreso Optional.empty ();
    }

    / **
     * Verifica si la lista de insignias aprobadas incluye la que se
     está verificando
     * /
    privado boolean containsBadge (lista final <BadgeCard>
    badgeCards,
                                        insignia final de la insignia) {
        regreso badgeCards.stream (). anyMatch (b -> b.getBadge (). es
        igual a (insignia));
    }

    / **
     * Asigna una nueva insignia al usuario dado
     * /
    privado BadgeCard giveBadgeToUser (insignia de insignia final, ID de
    usuario largo final) {
        BadgeCard badgeCard = nuevo BadgeCard (userId, insignia);
        badgeCardRepository.save (badgeCard);
        log.info ("El usuario con id {} ganó una nueva insignia: {}", userId,
        insignia);
        regreso badgeCard;
    }

}
```

Si nos centramos en el newAttemptForUser, entenderemos la lógica del juego: cuando recibimos un intento correcto, creamos un Tanteador objeto (con una puntuación predeterminada de 10) y persistirlo en la base de datos. Luego, invocamos el método processForBadges (), que consultará la base de datos para obtener una identificación de usuario determinada y asignará nuevas insignias cuando sea necesario. Finalmente, combinamos las puntuaciones con insignias en unGameStats objeto y devuelve este resultado. El resto de la clase está destinado a ayudar a ese método y a un sencillo retrieveStatsForUser () implementación.

Tenga en cuenta que aún no hemos cubierto la interacción de este microservicio con el microservicio de multiplicación: no hay lógica de eventos. Esta es una buena práctica para mantener las capas aisladas: no hacemos que nuestra capa de servicio dependa de la interfaz, por lo que no pasamos el evento como un argumento. El vínculo entre el bus de eventos y la lógica empresarial es elControlador de eventos. De esta manera, podemos reemplazar la interfaz de nuestro microservicio sin necesidad de cambios en otras capas (por ejemplo, si decidimos eliminar los eventos y poner algo más allí).

EJERCICIO

la implementación del servicio LeaderBoardServiceImpl es bastante sencillo: utilizará su método de repositorio existente para devolver los 10 usuarios con la puntuación más alta. intenta construirlo. si necesita ayuda, puede encontrar la solución en elv5 repositorio de código, dentro del gamificación proyecto.

La API REST (controladores)

Un evento entrante activará la lógica comercial principal en el servicio de gamificación, pero aún así, debemos exponer los resultados del juego. ¿Cuántos puntos tiene un usuario? ¿Cuál es la tabla de clasificación actual? Estas solicitudes provendrán de una interfaz de usuario o servicio de cliente que acceda a nuestro sistema, por lo que crearemos una API REST para ellas.

La LeaderBoardController expone un punto final llamado /líderes que recuperará la tabla de clasificación actual. Ver listado4-17.

Listado 4-17. LeaderBoardController.java (gamificación v5)

```
/**
 * Esta clase implementa una API REST para el servicio
   Gamification LeaderBoard.
 */
@RestController
@RequestMapping ("/ líderes")
clase LeaderBoardController {

    privado LeaderBoardService final LeaderBoardService;

    público LeaderBoardController (LeaderBoardService final
    LeaderBoardService) {
        esto.LeaderBoardService = líderBoardService;
    }

    @GetMapping
    público List <LeaderBoardRow> getLeaderBoard () {
        regreso leaderBoardService.getCurrentLeaderBoard ();
    }
}
```

Por otro lado, UserStatsController está cuidando el punto final / estadísticas, y devuelve una representación JSON del GameStats objeto: puntuación e insignias. Aquí usamos un parámetrouserId para consultar las estadísticas de un usuario determinado. En este caso, si queremos estadísticas para un usuario con ID 9, necesitamos solicitar GET / stats? UserId = 9. Ver listado 4-18.

Listado 4-18. UserStatsController.java (gamificación v5))

```
/**
 * Esta clase implementa una API REST para el servicio de estadísticas de
   usuarios de gamificación.
 */
@RestController
@RequestMapping ("/ stats")
clase UserStatsController {

    privado final GameService gameService;

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

    @GetMapping
    público GameStats getStatsForUser (@RequestParam ("userId")
    final Long userId) {
        regreso gameService.retrieveStatsForUser (userId);
    }
}
```

Además del código, vamos a cambiar el puerto predeterminado de la aplicación HTTP) a 8081, para evitar una colisión cuando los iniciemos juntos en la máquina local. Para que esto funcione, debe configurar la propiedad de Spring Boot Puerto de servicio en gamification's application.properties:

```
server.port = 8081
```

Recibir eventos con RabbitMQ

El lado del suscriptor

Al principio de este capítulo, vimos cómo conectar el microservicio de multiplicación con RabbitMQ y publicar un evento cuando un usuario envía un intento. Ahora veamos cómo luce el suscriptor (nuestro servicio de gamificación).

Configuración de RabbitMQ

Necesitaremos colocar una nueva clase ConejoMQConfiguración como hicimos para la multiplicación. Pero en este caso, será un poco más complicado. Tenemos seis métodos: cinco declare beans y el último implementa la interfazRabbitListenerConfigurer.

Para comprender esta configuración, el concepto de *vincular una cola a un intercambio* es importante aquí. Nuestro suscriptor crea una cola desde la que consumirá mensajes. Esos mensajes se publican en un intercambio con un*clave de enrutamiento*en nuestro caso multiplicación resuelta). Aquí es donde la flexibilidad de un *intercambio de temas* reside: todos los mensajes enviados a través del intercambio están "etiquetados" con una clave de enrutamiento, y los consumidores pueden seleccionar los mensajes que van a sus colas especificando una clave de enrutamiento explícita o un patrón (como en nuestro caso, multiplicación.*)
cuando unen sus colas al intercambio. Puedes mirar la página oficial de tutoriales de RabbitMQ[6] para aprender más sobre temas y ver varios ejemplos de enrutamiento. Ver listado4-19.

Listado 4-19. RabbitMQConfiguration.java (gamificación v5)

```
/**
 * Configura RabbitMQ para usar eventos en nuestra aplicación.
 */
```

[6]https://www.rabbitmq.com/tutorials/tutorial-five-spring-amqp.html

```
@Configuración
clase pública ConejoMQConfiguración implementos
RabbitListenerConfigurer {

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

    @Frijol
    público Cola gamificationMultiplicationQueue
    (@Value ("$ {multiplication.queue}") final String
    queueName) {
        volver nuevo Cola (queueName, cierto);
    }

    @Frijol
    Enlace de enlace (cola de cola final, intercambio de
    TopicExchange final,
                        @Value ("$ {multiplication.anything.routingkey}")
                        final String routingKey) {
        regreso BindingBuilder.bind (cola) .to (intercambio).
        con (routingKey);
    }

    @Frijol
    público MappingJackson2MessageConverter consumidor
    Jackson2MessageConverter () {
        volver nuevo MappingJackson2MessageConverter ();
    }
```

```
    @Frijol
    público DefaultMessageHandlerMethodFactory mensaje
    HandlerMethodFactory () {
        DefaultMessageHandlerMethodFactory factory = nuevo
        DefaultMessageHandlerMethodFactory ();
        factory.setMessageConverter (consumidorJackson2MessageConv
        erter ());
        regreso fábrica;
    }

    @Anular
    público void configureRabbitListeners (registrador final
    de Rabbit ListenerEndpointRegistrar) {
        registrar.setMessageHandlerMethodFactory (messageHandler
        MethodFactory ());
    }
}
```

Repasemos algunas partes importantes dentro de esta clase:

- Los primeros tres métodos son conectar una nueva cola (declarada por gamificationMultiplicationQueue ()) y un TopicExchange (declarado por multiplicationExchange ()) uniéndolos juntos (Unión(), que toma el intercambio y la cola como argumentos).
- Hacemos el Cola duradero (el segundo cierto argumento al crearlo). Introdujimos esta idea antes: al hacer esto, podemos procesar eventos pendientes incluso después de que el corredor deja de funcionar, dado que persisten.
- Tenga en cuenta que el valor de la propiedad multiplicación. intercambiodebe ser el mismo que se define en la multiplicación: intercambio_multiplicación). Usamos el patrón

multiplicación.* por el valor de la propiedad de la clave de enrutamiento (multiplicación, cualquier cosa, clave de enrutamiento). Para el nombre de la cola (cola de multiplicación propiedad), podemos usar cualquier convención que prefiramos. Todos estos valores deben definirse en elapplication.properties expediente.

- Los últimos tres métodos configuran la deserialización JSON en el suscriptor. En este caso, se hace de manera diferente si lo comparas con nuestras multiplicaciones RabbitMQConfiguration. Ahora no usamos un RabbitTemplate (ya que no estamos enviando mensajes desde este microservicio) sino métodos anotados con @ConejoListener. Por lo tanto, necesitamos configurar el RabbitListenerEndpointRegistrar de una manera que usa un MappingJackson2MessageConverter.

Listado 4-20. cambios en application.properties (gamification v5)

```
# # Otras propiedades ...
# # Configuración de RabbitMQ
multiplication.exchange = multiplication_exchange
multiplication.solved.key = multiplication.solved
multiplication.queue = gamification_multiplication_queue
multiplication.anything.routing-key = multiplication. *
```

El controlador de eventos

Recuerde que, cuando presentamos el patrón del controlador de eventos, lo hicimos junto con el despachador de eventos. El objetivo es similar: tener un lugar centralizado desde donde podamos procesar los eventos recibidos y desencadenar la lógica de negocio correspondiente.

Creamos tantos métodos anotados con @ConejoOyente como eventos para ser consumidos. Esta anotación maneja toda la complejidad de recibir un mensaje del broker a través de la cola que definimos (necesitamos pasar el nombre de la cola como parámetro a la anotación).

Dado que estamos pasando el tipo de argumento MultiplicationSolvedEvent, el conversor de mensajes (configurado en RabbitMQConfiguration) deserializará el JSON recibido en un objeto de esta clase. Para evitar interdependencias entre nuestros microservicios, copiamos nuestroMultiplicationSolvedEvent clase al proyecto de gamificación. Cubriremos esta idea con más detalle al final de este capítulo cuando hablemos de*aislamiento de dominio.* Ver listado 4-21.

Listado 4-21. EventHandler.java (gamificación v5)

```
paquete microservices.book.gamification.event;

importar lombok.extern.slf4j.Slf4j;
importar microservices.book.gamification.service.GameService;
importar org.springframework.amqp.
AmqpRejectAndDontRequeueException;
importar org.springframework.amqp.rabbit.annotation.RabbitListener;
importar org.springframework.stereotype.Component;

/**
 * Esta clase recibe los eventos y desencadena el asociado
 * lógica de negocios.
 */
@ Slf4j
@Componente
clase Controlador de eventos {

    privado GameService gameService;

    EventHandler (final GameService gameService) {
        esto.gameService = gameService;
    }
```

```
    @RabbitListener (queues = "$ {multiplication.queue}")
    void handleMultiplicationSolved (evento final
    Multiplication SolvedEvent) {
        log.info ("Evento de multiplicación resuelto recibido: {}",
        event.getMultiplicationResultAttemptId ());
        intentar {
            gameService.newAttemptForUser (event.getUserId (),
                    event.getMultiplicationResultAttemptId (),
                    event.isCorrect ());
        } captura (Excepción final e) {
            log.error ("Error al intentar procesar
            MultiplicationSolvedEvent", e);
            // Evita que el evento se vuelva a poner en cola y se
            vuelva a procesar.
            tirar nuevo AmqpRejectAndDontRequeueException (e);
        }
    }
}
```

Tenga en cuenta que también estamos envolviendo la lógica dentro de un trata de atraparlo bloquear y lanzar un AmqpRejectAndDontRequeueException en caso de que se produzca una excepción. Al hacer eso, nos aseguramos de que el evento no se ponga en cola repetidamente cuando algo está mal (que es el comportamiento predeterminado), sino que se rechaza directamente. Dado que no tenemos nada para manejar los eventos rechazados, simplemente se descartarán. Si desea profundizar en las buenas prácticas con RabbitMQ, puede ver cómo configurar un intercambio de letra muerta y colocar nuestros mensajes fallidos allí para su posterior procesamiento (como reintentar, iniciar sesión o generar alertas).[7]

[7] https://www.rabbitmq.com/dlx.html

Solicitar datos entre microservicios

Combinando patrones reactivos y REST

En la sección "Conexión de microservicios", presentamos brevemente el concepto de microservicios que se llaman entre sí para recopilar datos. En esta sección, ampliamos la idea utilizando nuestra arquitectura como referencia.

Imagina que tenemos un cambio en nuestro diseño de gamificación. Nuestros diseñadores de juegos crean una nueva insignia llamada *Número de la suerte.* Nos dicen que los usuarios solo pueden obtener esta insignia si resuelven un intento de multiplicación con el número 42 (que parece ser un número de la suerte, al menos para ellos).

Apliquemos lo que sabemos hasta ahora para adaptar el nuevo requisito a nuestro diseño. Primero, podemos concluir que el contexto de esa nueva lógica empresarial es nuestro microservicio de gamificación: es el que asigna insignias y esta es la lógica de una insignia. Sin embargo, tenemos un pequeño problema: la gamificación no sabe nada sobre los factores. Ellos no estan entrando en el MultiplicationSolvedEvent.

Además, sabemos que no suena bien incluir los factores en nuestro evento solo porque ahora es un requisito para nuestro consumidor. En este caso, puede ser simple y, sin embargo, parecer un evento genérico, pero si sigue el enfoque de adaptar los editores a los consumidores, puede terminar con *eventos gordos, demasiado inteligentes.*

Cuando desea compartir datos entre microservicios, no lo hace utilizando sus patrones reactivos (*evento-ha-sucedido)* pero use un patrón de solicitud / respuesta en su lugar. También puede hacerlo usando las mismas tecnologías subyacentes (AMQP / RabbitMQ), pero es mucho más fácil usar una de las implementaciones más comunes del patrón de solicitud / respuesta para transferir objetos: API REST.

Para solucionar este nuevo reto, el microservicio de gamificación puede contactar con el microservicio de multiplicación y solicitar los factores de multiplicación dado el identificador del intento (contenido en el evento). Luego, si encuentra el número de la suerte, le asignará la insignia. No te pongas nervioso si estás

ya visualizando esta arquitectura y pensando que vamos a acoplar nuestros microservicios; En el próximo capítulo, verá cómo este enfoque no le impide tener un sistema débilmente acoplado. Ver figura4-4.

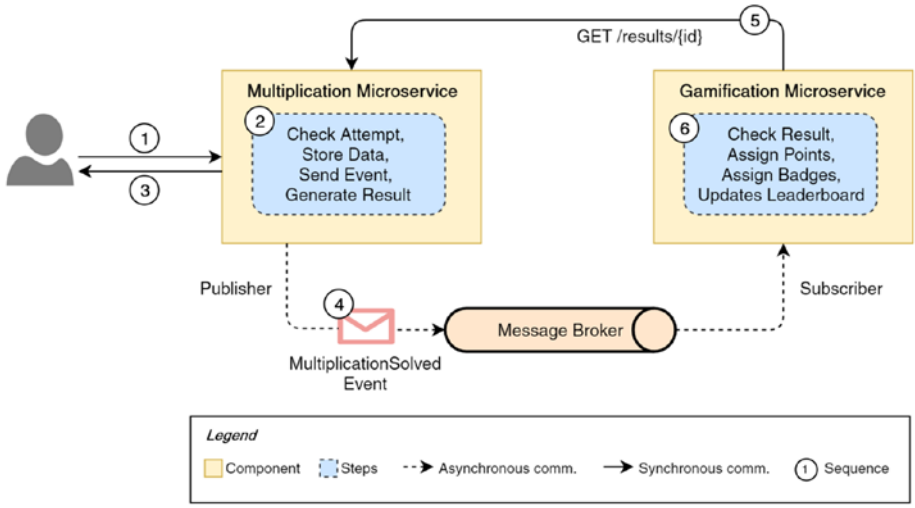

Figura 4-4. *Vista lógica actualizada*

Como puede ver en la vista lógica actualizada, necesitamos exponer un nuevo punto final en el microservicio de multiplicación para dar acceso a los intentos de multiplicación (que incluyen los factores de multiplicación) por el identificador de su intento. Luego, necesitamos crear un cliente REST en gamificación para recuperar los factores. Finalmente, los usamos en nuestra lógica para asignar la insignia si uno de ellos es el número de la suerte.

EJERCICIO

en este punto del libro, la creación de un punto final en el servicio de multiplicación para recuperar un MultiplicationResultAttempt por iD debería ser muy fácil. Asegúrese de que responda aOBTENER / resultados / {id}
y no olvides incluir las pruebas. pista: esta vez deberías usar un

@Variable anotación en el controlador ya que el iD está incrustado en el Uri y no es un parámetro. La declaración de su nuevo método en MultiplicationResultAttemptController debiera ser:

```
@GetMapping ("/ {resultId}")
ResponseEntity getResultById (@PathVariable ("resultId") Long resultId)
```

también, necesitará usar el MultiplicationRepository, lo que provocará una refactorización menor de otras clases. Si necesita ayuda, consulte la solución que se encuentra en elmultiplicación social proyecto, dentro del v5 carpeta.

Mantener dominios aislados

Hay un efecto importante de incluir esta nueva funcionalidad en nuestro sistema que merece algunas líneas en el libro: el microservicio de gamificación deberá manejar los intentos, por lo que el concepto de negocio del intento debe ser entendido por ambos microservicios. Eso significa que necesitamos un modelo MultiplicationResultAttempt dentro de la gamificación *de algun modo.*

Aunque esto no parezca un tema complejo, es uno de los factores clave del éxito o el fracaso al diseñar una arquitectura de microservicios.

Un error común es pensar en extraer el paquete de dominio de multiplicationmicroservice como una biblioteca separada que se puede compartir con la gamificación, teniendo así acceso a MultiplicationResultAttempt.
Sin embargo, es una muy mala idea: su dominio estaría fuera de control y otros microservicios podrían comenzar a depender de él para su lógica, introduciendo así múltiples interdependencias si usted (u otros) necesitan actualizaciones en el modelo. Siempre debe mantener la propiedad de las entidades de dominio en un microservicio, por lo que solo hay una fuente de verdad.

Una mejor alternativa es generar *copias simples* de su modelo y compartirlos con otros. En realidad, este es un enfoque basado en objetos de transferencia de datos (DTO). Si sigue este camino, tenga en cuenta que el mantenimiento de esos DTO lleva tiempo y también puede introducir dependencias si no sigue algunas otras buenas prácticas. Por ejemplo, si esos DTO son representaciones Java de objetos JSON que está transfiriendo a través de una API REST, debe mantener diferentes paquetes DTO por versión de API o creará muchos dolores de cabeza a sus consumidores (no podrán hacerlo). deserialice el JSON si introduce cambios en la estructura). Por otro lado, tener estos paquetes DTO puede ahorrar mucho tiempo de desarrollo si tiene muchos consumidores de API, porque no necesitarán replicar estructuras de datos.

Si no le importa lidiar con respuestas simples (digamos JSON) en sus consumidores, el enfoque ideal es mantener los microservicios lo más aislados posible: *no compartas nada.* Esto tiene la ventaja de minimizar las dependencias: si solo necesita un par de campos, los deserializa e ignora todo lo demás. De esa manera, su microservicio se verá afectado solo si esos campos específicos cambian.

Una ventaja adicional de no compartir es que puede adaptar las entidades del modelo externo como desee: en este caso, aplanaremos el *AttemptMultiplication-User* estructura a una sola clase, dado que la multiplicación no necesita tal estructura.

Seamos prácticos. En el lado de la gamificación, vamos a crear este MultiplicationResultAttempt versión en una nueva client.dto paquete. Esta clase contienefactorA y factorB como campos. Ver listado4-22.

Listado 4-22. MultiplicationResultAttempt.java (gamificación v5)

```
paquete microservices.book.gamification.client.dto;

importar com.fasterxml.jackson.databind.annotation.JsonDeserialize;
importar lombok.EqualsAndHashCode;
importar lombok.Getter;
```

```
importar lombok.RequiredArgsConstructor;
importar lombok.ToString;
importar microservices.book.gamification.client.
        MultiplicationResultAttemptDeserializer;

/**
 * Identifica el intento de un usuario de resolver una multiplicación.
 */
@RequiredArgsConstructor
@Getter
@Encadenar
@EqualsAndHashCode
@JsonDeserialize (usando =
MultiplicationResultAttemptDeserializer.class)
público final clase MultiplicationResultAttempt {

    privado final String userAlias;

    privado final int multiplicationFactorA;
    privado final int multiplicationFactorB;
    privado final int resultAttempt;

    privado booleano final correcto;

    // Constructor vacío para JSON / JPA
    MultiplicationResultAttempt () {
        userAlias   = nulo;
        multiplicationFactorA = -1;
        multiplicationFactorB = -1;
        resultAttempt = -1;
        correcto = falso;
    }

}
```

Una versión simplificada del original: aplanada y sin identificadores. Nota la @ JsonDeserialize anotación que apunta a una clase que aún no tenemos: es para instruir a nuestro @RestTemplateConversor de mensajes para usar un deserializador especial para leer los datos JSON. Necesitamos esto ya que la estructura JSON que recibiremos no coincide con nuestra clase Java (ya que coincide*el original* MultiplicationResultAttempt en el microservicio de multiplicación), por lo que el deserializador predeterminado no funcionará. Cubriremos esa implementación en la siguiente subsección.

Implementación del cliente REST

Para empezar, necesitamos decirle a la gamificación cómo encontrar la multiplicación: usaremos una nueva línea en nuestro application.properties y referenciarlo desde el código. Por ahora, lo apuntaremos directamente al host y al puerto en el que sabemos que está implementado el microservicio. Aprenderemos cómo hacer esto correctamente en el próximo capítulo cuando analicemos el descubrimiento y el enrutamiento de servicios. Ver listado4-23.

Listado 4-23. application.properties (gamificación v5)

 # Configuración del cliente REST
multiplicationHost = http: // localhost: 8080

Implementemos ahora el deserializador JSON personalizado para el MultiplicationResultAttempt. Usamos algunas clases del *Biblioteca de Jackson,* que se incluye dentro de Spring Boot. Ver listado4-24.

Listado 4-24. MultiplicationResultAttemptClient.java (gamificación v5)

paquete microservices.book.gamification.client;

importar com.fasterxml.jackson.core.JsonParser;
importar com.fasterxml.jackson.core.JsonProcessingException;
importar com.fasterxml.jackson.core.ObjectCodec;

```
importar com.fasterxml.jackson.databind.DeserializationContext;
importar com.fasterxml.jackson.databind.JsonDeserializer;
importar com.fasterxml.jackson.databind.JsonNode;
importar microservices.book.gamification.client.dto.
        MultiplicationResultAttempt;

importar java.io.IOException;

/**
 * Deserializa un intento procedente del microservicio de
   multiplicación
 * en la representación de Gamificación de un intento.
 */
clase pública MultiplicationResultAttemptDeserializer
        se extiende JsonDeserializer <MultiplicationResultAttempt>
        {@Override
    público MultiplicationResultAttempt deserialize (JsonParser
            jsonParser,
                                    DeserializationContext
                                    deserializationContext)
            lanza IOException, JsonProcessingException
        {ObjectCodec oc = jsonParser.getCodec (); Nodo
        JsonNode = oc.readTree (jsonParser);
        volver nuevo MultiplicationResultAttempt (node.
                    get ("usuario"). get ("alias"). asText (), node.get
                ("multiplicación"). get ("factorA"). asInt (), node.get
                ("multiplicación"). get (" factorB "). asInt (), node.get
                (" resultAttempt "). asInt (),
                node.get ("correcto"). asBoolean ());
    }
}
```

Como puede ver, es bastante legible. Necesitamos crear una subclase de JsonDeserializer, pasando el tipo que queremos usar como resultado. Luego, implementamos eldeserializar () método, del cual obtenemos un JsonParser que podemos usar para recorrer el árbol de nodos JSON y obtener los valores que estamos buscando.

El siguiente paso es escribir una interfaz para abstraer la lógica de comunicación. Desde el punto de vista de la lógica empresarial, solo queremos recuperar el intento, sin importar qué tipo de interfaz técnica estemos usando. Ver listado4-25.

Listado 4-25. MultiplicationResultAttemptClient.java (gamificación v5)

```
paquete microservices.book.gamification.client;

importar microservices.book.gamification.client.dto.
MultiplicationResultAttempt;

/**
 * Esta interfaz nos permite conectarnos al microservicio de
   Multiplicación.
 * Tenga en cuenta que es independiente de la forma de comunicación.
 */
interfaz pública MultiplicationResultAttemptClient {

    MultiplicationResultAttempt retrieveMultiplicationResultAtt
    emptbyId (final Long multiplicationId);

}
```

Para la implementación usamos RestTemplate, una clase proporcionada por Spring que facilita la comunicación con las API REST. Para tenerlo disponible en el contexto de nuestra aplicación Spring, necesitamos configurarlo como un bean. Crearemos una nueva clase de configuración para mantener nuestro código organizado (en elconfiguración paquete). Ver listado4-26.

Listado 4-26. RestClientConfiguration.java (gamificación v5)

```
paquete microservices.book.gamification.configuration;

importar org.springframework.boot.web.client.RestTemplateBuilder;
importar org.springframework.context.annotation.Bean;
importar org.springframework.context.annotation.Configuration;
importar org.springframework.web.client.RestTemplate;

/**
 * Configura el cliente REST en nuestra aplicación
 */
@Configuración
clase pública RestClientConfiguration {

    @Frijol
    público RestTemplate restTemplate (constructor RestTemplateBuilder) {
        regreso constructor.build ();
    }

}
```

Entonces, podemos inyectar el RestTemplate en MultiplicationResult AttemptClientImpl, junto con el multiplicationHost propiedad, y realizar una OBTENER solicitud con el identificador pasado. Ver listado4-27.

Listado 4-27. MultiplicationResultAttemptClientImpl.java (gamificación v5)

```
paquete microservices.book.gamification.client;

importar microservices.book.gamification.client.dto.
        MultiplicationResultAttempt;
importar org.springframework.beans.factory.annotation.Autowired;
importar org.springframework.beans.factory.annotation.Value;
importar org.springframework.stereotype.Component;
```

```
importar org.springframework.web.client.RestTemplate;

/**
 * Esta implementación de la interfaz
   MultiplicationResultAttemptClient se conecta a
 * el microservicio de multiplicación a través de REST.
 */
@Componente
clase MultiplicationResultAttemptClientImpl implementos
        MultiplicationResultAttemptClient {

    privado RestTemplate final restTemplate;
    privado final String multiplicationHost;

    @Autowired
    público MultiplicationResultAttemptClientImpl
            (final RestTemplate restTemplate, @Value ("$
            {multiplicationHost}") final String
            multiplicationHost) {
        esto.restTemplate = restTemplate;
        esto.multiplicationHost = multiplicationHost;
    }

    @Anular
    público MultiplicationResultAttempt retrieveMultiplication
    ResultAttemptbyId (final Long multiplicationResultAttemptId) {
        regreso restTemplate.getForObject (
                multiplicationHost + "/ results /" +
                multiplicationResultAttemptId,
                MultiplicationResultAttempt.class);
    }
}
```

No necesitamos vincular el deserializador aquí, ya que establecemos la anotación dentro MultiplicationResultAttempt. La getForObject () toma la clase como argumento, luego infiere que debe usar el deserializador personalizado.

EJERCICIO

¡Puedes intentar terminar esta tarea tú mismo! Puedes seguir a tDD nuevamente, actualizandoGameServiceImplTest para verificar que la nueva insignia se asigna a los usuarios cada vez que envían un intento con el número 42 como uno de los factores. Deberá crear la nueva insignia y adaptar el GameServiceImpl clase.

Cubriremos la implementación de la lógica empresarial pero dejaremos las pruebas fuera del libro (ya que son fáciles y no agregan nuevos conceptos). si quieres mirarlos, están ubicados en elv5 repositorio, bajo el gamificación proyecto.

Actualización de la lógica empresarial de la gamificación

Ahora que tenemos todas las piezas que necesitamos para conectar ambos servicios, podemos actualizar nuestra lógica de negocio en gamificación para comprobar si están configuradas las condiciones para asignar la nueva insignia llamada *Número de la suerte* al usuario.

Primero, necesitamos crear la nueva insignia agregando un valor a nuestra enumeración, como se muestra en Listado 4-28.

Listado 4-28. MultiplicationResultAttemptClientImpl.java (gamificación v5)

```
enumeración pública Insignia {
    // ...
    PRIMERA_GANADA,
    NÚMERO DE LA SUERTE
}
```

Ahora podemos inyectar al cliente en la lógica del juego y procesar el intento devuelto para asignar la insignia si el número de la suerte está presente. Ver listado 4-29.

Listado 4-29. MultiplicationResultAttemptClientImpl.java (gamificación v5)

```
@Servicio
@ Slf4j
clase GameServiceImpl implementos GameService {

    público estático final int LUCKY_NUMBER = 42;

    privado ScoreCardRepository scoreCardRepository;
    privado BadgeCardRepository badgeCardRepository;
    privado MultiplicationResultAttemptClient intentClient;

    GameServiceImpl (ScoreCardRepository scoreCardRepository,
                     BadgeCardRepository badgeCardRepository,
                      MultiplicationResultAttemptClient
                      intentoClient) {
        esto.scoreCardRepository = scoreCardRepository;
        esto.badgeCardRepository = badgeCardRepository;
        esto.intentoCliente = intentoCliente;
    }

    // ...

    / **
      * Comprueba la puntuación total y las diferentes tarjetas de puntuación obtenidas.
      * Dar nuevas insignias en caso de que se cumplan sus condiciones.
      * /
    privado Lista <BadgeCard> processForBadges (final Long userId,
                                                 final largo
                                                intentId) {
```

```
        Lista <BadgeCard> badgeCards = nuevo ArrayList <> ();

        int totalScore = scoreCardRepository.getTotal
        ScoreForUser (userId);
        log.info ("La nueva puntuación para el usuario {} es {}", userId, totalScore);

        Lista <ScoreCard> scoreCardList = scoreCardRepository
                . findByUserIdOrderByScoreTimestampDesc   (userId);
        Lista <BadgeCard> badgeCardList = badgeCardRepository
                . findByUserIdOrderByBadgeTimestampDesc   (userId);

        // Insignias según la puntuación ... //

        Primera insignia ganada ...

        // Insignia de número de la suerte
        MultiplicationResultAttempt intento = intentoCliente
                . retrieveMultiplicationResultAttemptbyId (intento
                tId);
        Si(!containsBadge (badgeCardList, Badge.LUCKY_NUMBER) &&
                (LUCKY_NUMBER == try.getMultiplication
                FactorA () ||
                LUCKY_NUMBER == try.getMultiplication
                FactorB ())) {
            BadgeCard luckyNumberBadge = giveBadgeToUser (
                    Badge.LUCKY_NUMBER, userId);
            badgeCards.add (luckyNumberBadge);
        }

        regreso badgeCards;
    }
    // ...
}
```

Jugando con los microservicios

A pesar de no haber terminado la historia de usuario 3, ya podemos probar si nuestros microservicios funcionan bien juntos.

Si aún no lo hizo, debe descargar e instalar RabbitMQ. Simplemente siga las instrucciones del sitio web para su sistema operativo. Una vez que lo instale, inicie el corredor (las guías de instalación también incluyen instrucciones para eso). Asegúrese de habilitar también elrabbitmq_management complemento con el comando rabbitmq-plugins habilitan rabbitmq_management (ver https://www.rabbitmq.com/management.html si necesita más detalles). Eso le dará acceso a una interfaz de usuario web para administrar RabbitMQ (enhttp: // localhost: 15672 / si lo instaló en su máquina local). Mantenga la configuración como está por defecto.

Una vez que el servidor RabbitMQ esté activo, puede iniciar ambas aplicaciones Spring Boot. Recuerde que puede hacerlo desde su IDE ejecutando la clase anotada con @SpringBootApplication o empaquetando las aplicaciones con paquete mvn y luego ejecutando los archivos JAR resultantes con java -jar your_jar_file. Si todo va bien, la multiplicación debería comenzar en el puerto 8080 y la gamificación en el puerto 8081.

Puedes navegar a http: // localhost: 8080 / index.html y empieza a intentar resolver algunas multiplicaciones, como hiciste en el capítulo anterior. La diferencia ahora es que cada vez que envías un intento, se publica un nuevo evento. Puede notar que al mirar el registro de gamificación, como se muestra en el listado4-30.

Listado 4-30. Registro de gamificación después de un intento recibido (gamificación v5)

```
2017-09-15 18: 43: 25.050 INFO 16276 --- [                principal]
m.book.GamificationApplication                : Empezado
Aplicación de gamificación en 8.225 segundos (JVM ejecutándose para
8.938) 2017-09-15 18: 43: 34.351 INFO 16276 --- [cTaskExecutor-1]
m.book.gamification.event.EventHandler: Multiplicación
Evento resuelto recibido: 65
```

2017-09-15 18: 43: 58.194 INFO 16276 --- [cTaskExecutor-1]
m.book.gamification.event.EventHandler: Multiplicación
Evento resuelto recibido: 66
Hibernar: insertar en score_card (card_id, intent_id, score, score_ts,
user_id) valores (nulo,?,?,?,?)
2017-09-15 18: 43: 58.253 INFO 16276 --- [cTaskExecutor-1]
mbgservice.GameServiceImpl: Usuario con id 1
anotó 10 puntos por intento id 66
Hibernate: seleccione sum (scorecard0_.score) como col_0_0_ from
score_card scorecard0_ donde scorecard0_.user_id =? agrupar por
cuadro de mando0_.user_id
2017-09-15 18: 43: 58.274 INFO 16276 --- [cTaskExecutor-1]
mbgservice.GameServiceImpl: Nueva puntuación para el usuario 1
es 50
Hibernate: seleccione scorecard0_.card_id como card_id1_1_,
scorecard0_.attempt_id como intento_2_1_, scorecard0_.score como
score3_1_, scorecard0_.score_ts como score_ts4_1_, scorecard0_.
user_id como user_id5_1_ de score_card scorecard0_ donde
scorecard0_.user_id =? ordenar por scorecard0_.score_ts desc
Hibernate: seleccione badgecard0_.badge_id como badge_id1_0_,
badgecard0_.badge como badge2_0_, badgecard0_.badge_timestamp
as badge_ti3_0_, badgecard0_.user_id as user_id4_0_ from badge_card
badgecard_0_ where badgecard0_ ordenar por
badgecard0_.badge_timestamp desc

Si envía intentos exitosos, también puede ver cómo se conservan los datos en la base de datos de gamificación. ¿Recuerda que tuvo acceso a la base de datos de multiplicación a través de la consola H2? Haces lo mismo con la gamificación, pero esta vez lo encontrarás enhttp: // localhost: 8081 / h2-console /. Del mismo modo, debe asegurarse de utilizar la URL adecuada para acceder a ella: jdbc: h2: file: ~ / gamification. Si consulta las tablas después de enviar intentos correctos, debería ver las diferentes insignias y puntuaciones vinculadas a los usuarios. Ver figura4-5.

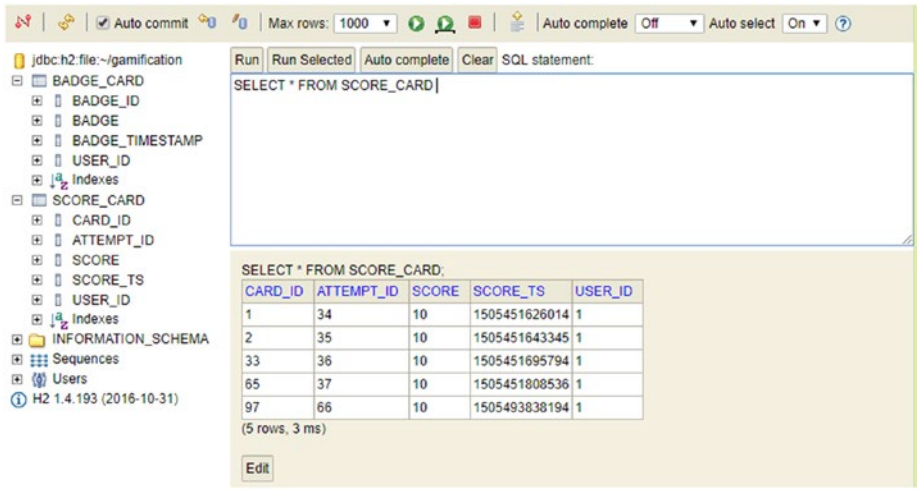

Figura 4-5. *Puedes consultar la base de datos de gamificación usando H2*

Para probar el número de la suerte, necesitamos engañar al sistema. No es tan fácil conseguir una multiplicación con un factor 42 por accidente (aunque puedes intentarlo si eres una persona muy paciente). Si prefiere no esperar, puede utilizarrizo para publicar su propia multiplicación en el sistema, como se muestra en el Listado 4-31.

Listado 4-31. Gamificación: publicar una multiplicación con un número de la suerte (gamificación v5)

```
$ curl -X POST -H "Tipo de contenido: aplicación / json" -d '{"usuario":
{"alias": "moises"}, "multiplicación": {"factorA": "42", "factorB" : "10"},
"resultAttempt": "420"} 'http: // localhost: 8080 / results
```

Si está ejecutando la aplicación en una máquina con Windows, puede usar un emulador de terminal para ejecutar rizocomo *Git Bash*, incluido si instala el paquete Git desde https://git-scm.com). También puede utilizar Postman,[8] que le brinda una interfaz de usuario completa desde la que puede enviar todo tipo de solicitudes. Es bastante intuitivo, pero puede consultar la documentación si necesita ayuda.

[8] https://www.getpostman.com/

Resumen

En este capítulo, presentamos el concepto de microservicios. Antes de profundizar en la idea de microservicios, analizamos nuestro plan para abordarlo: comenzando con un monolito. Usamos nuestra aplicación para comparar lo costoso que hubiera sido comenzar directamente con un enfoque de microservicios y cubrimos el plan para preparar el monolito para dividirlo más tarde.

Recibimos un nuevo requisito en forma de historia de usuario y lo revisamos para especificar un segundo servicio a diseñar e implementar, basado en técnicas de gamificación. Aprendimos los conceptos básicos de estas técnicas para motivar a los jugadores:*puntos, insignias y tablas de clasificación.* Siguiendo un enfoque pragmático, comenzamos con una lógica simple que funciona.

Con respecto a las interacciones de microservicios, vimos que hay diferentes formas de conectar microservicios para cumplir con los procesos comerciales, y optamos por una arquitectura impulsada por eventos. Revisamos sus beneficios y lo aplicamos directamente a nuestra aplicación para verlo en la práctica. Lo comparamos con algunas otras técnicas relacionadas (abastecimiento de eventos, CQRS, etc.) y vimos cómo pueden funcionar juntas, muy bien en algunos casos, teniendo en cuenta que una buena evaluación de la tecnología frente a los requisitos es fundamental para evitar ser engañados. por las exageraciones tecnológicas.

La segunda parte del capítulo explicó cómo implementar la comunicación asincrónica que soporta nuestra arquitectura dirigida por eventos, usando Spring AMQP con RabbitMQ como implementaciones. Hicimos el primer cambio del lado del editor, el microservicio de multiplicación. Luego, pasamos por la implementación del nuevo microservicio, la aplicación de gamificación Spring Boot. En nuestro camino, no solo vimos cómo implementar el*del lado del suscriptor,* pero también buscamos nuevas formas de enviar consultas más avanzadas a la base de datos y el modelo de gamificación y la lógica en el código.

Terminamos este capítulo con un sistema basado en microservicios, pero no terminamos nuestra historia de usuario 3: necesitamos proporcionar acceso a la interfaz de usuario a la tabla de clasificación y, en el aspecto técnico, debemos corregir algunos vínculos físicos que hicimos entre nuestras aplicaciones (por apuntando a un host y puerto específicos).

El próximo capítulo cubre una refactorización del código para crear un sistema correctamente dividido y, mientras lo hacemos, necesitaremos comprender y aplicar dos conceptos importantes detrás de los microservicios: *enrutamiento y descubrimiento de servicios.*

CAPÍTULO 5

Los microservicios Viaje a través

Herramientas

Introducción

Completamos el capítulo anterior con una tarea pendiente: terminar la historia de usuario 3 y dejar que nuestros usuarios vean su progreso en el juego. La razón es que, para construir una buena arquitectura de microservicios, necesitamos la parte UI del sistema extraída en un nuevo servicio para que pueda interactuar como una parte independiente con nuestros servicios de multiplicación y gamificación. Cubriremos el razonamiento con más detalle en la siguiente sección.

Tan pronto como dejamos de lado la interfaz de usuario y la comunicamos a los microservicios, descubriremos que nuestro entorno se vuelve aún más complicado. Sin embargo, eso no debería ser una sorpresa, ya que sabemos que ese es el caso de cualquier arquitectura de microservicios. Muchas ventajas, a expensas de un sistema más complejo.

Si hacemos un análisis rápido de lo que sucederá en este capítulo, comenzará con la interfaz de usuario llamando a dos servicios por su nombre de host y puertos específicos, extendiendo la infraestructura de nuestros servicios con enlaces estrechamente acoplados entre ellos y creando un sistema que no es escalable y es muy difícil de mantener. Dado que ese es un enfoque incorrecto, presentaremos el

M. Macero, *LearnMicroservices con Spring Boot,*
https://doi.org/10.1007/978-1-4842-3165-4_5

concepto de *descubrimiento de servicios* y cómo eso puede ayudarnos a localizar los servicios manteniéndolos débilmente acoplados. Luego analizaremos cómo funcionaría ese escenario si apuntamos a una alta disponibilidad mediante la ampliación de nuestros servicios. Finalmente, después de una evaluación de nuestra arquitectura, llegaremos a la conclusión de que en realidad es una buena idea incluir un servicio de enrutamiento (puerta de enlace) que pueda manejar el equilibrio de carga en el lado del servidor.

La mayoría de las herramientas que cubriremos para lograr nuestra misión son parte de la familia Spring Cloud: Eureka, Ribbon, Zuul, Hystrix, etc. También explicaremos cómo funcionan Sidecar y Feign, que también forman parte de Spring Cloud. No se preocupe por la abrumadora cantidad de*herramientas con nombres elegantes,* ya que los revisaremos uno por uno y explicaremos por qué son útiles y en qué circunstancias debe confiar en ellos. Y, como de costumbre, analizaremos el problema que queremos resolver primero y solo después implementaremos la solución.

Luego llegaremos al final del capítulo conociendo todas estas fantásticas herramientas y dónde encontrarlas y nos haremos una pregunta: ¿podríamos hacerlo más fácil? ¿No podríamos simplemente centrarnos en el código comercial de los microservicios y olvidarnos de todas las herramientas? Las respuestas a estas preguntas se encuentran al final del capítulo, donde explicaremos cómo funcionan las soluciones totalmente integradas para microservicios.

Extraer la interfaz de usuario y conectarla a la gamificación

Lo primero es lo primero: les prometí buenas razones para todas las decisiones que tomamos a lo largo del libro. Es hora de explicar por qué queremos extraer nuestro contenido estático del microservicio de multiplicación. Una imagen puede ilustrar esto muy claramente, así que mire la Figura5-1.

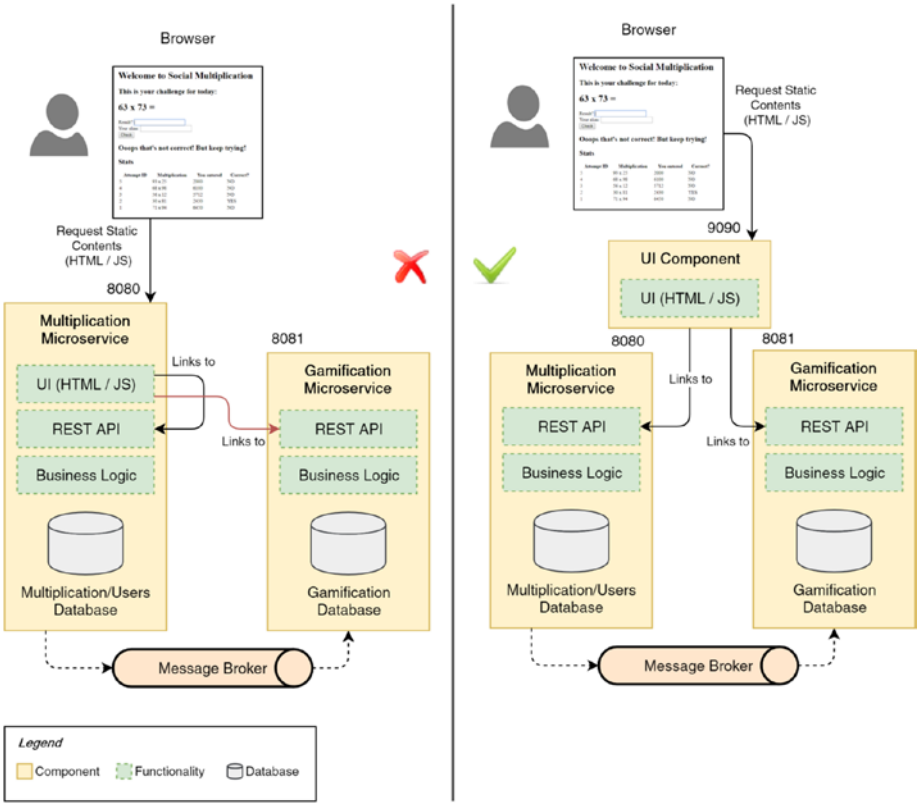

Figura 5-1. *Extraer el contenido estático del microservicio de multiplicación*

En el lado izquierdo de la figura 5-1, podemos ver cómo luciría nuestro sistema si avanzamos sin la extracción de UI. El contenido estático será servido por el microservicio de multiplicación, como antes. Si incluimos la nueva tabla de clasificación HTML / JavaScript, ese código debería tener referencias a la API REST de la gamificación. Sin embargo, ese no es un buen patrón de diseño: la interfaz de usuario ya no estaría limitada a las funciones de multiplicación, ya que incluiría funciones que abarcan ambos microservicios.

La primera desventaja de ese enfoque es la interdependencia: si cambiamos la API de gamificación, también necesitaríamos cambiar el microservicio de multiplicación y volver a implementarlo después. La segunda desventaja es que perderíamos flexibilidad para escalar: la disponibilidad del servidor de IU está vinculada a la disponibilidad del microservicio de multiplicación. Tal vez necesitemos más recursos en el futuro para servir las páginas de la interfaz de usuario, pero es posible que no queramos escalar todas las funciones que ofrece el microservicio.

El lado derecho de la figura 5-1 muestra el camino para continuar con nuestra aventura de microservicios. (No es un paso final, pero al menos nos lleva en la dirección correcta). Extraemos el contenido estático de la multiplicación y lo colocamos en un componente de servidor web diferente (que implementaremos en el puerto 9090). El código JavaScript se vinculará a las API REST ofrecidas por la multiplicación y la gamificación, que no contienen una interfaz de usuario. Con este enfoque, ganamos independencia para los cambios en nuestra interfaz de usuario y microservicios, que no dependen unos de otros. Además, ahora podríamos escalar hacia arriba o hacia abajo nuestro servidor de interfaz de usuario siguiendo una estrategia diferente a la del microservicio de multiplicación.

Ahora que conocemos nuestro plan y las razones detrás de él, ejecutémoslo.

Mover el contenido estático

El objetivo es trasladar todo el contenido estático (HTML, CSS y JavaScript) a un servicio independiente. No necesitamos Spring Boot para ello: solo necesitamos un servidor web bueno, confiable y, si es posible, liviano. Hay muchos buenos: Tomcat, Nginx, Jetty, etc. Usaremos Jetty ya que está construido sobre Java, por lo que es muy fácil de instalar y ejecutar en plataformas Linux y Windows.[1]

[1] También puede elegir Tomcat por su coherencia con los microservicios Spring Boot pero, en este caso, elegí Jetty ya que es muy fácil explicar cómo descargarlo y configurarlo en el libro.

CÓDIGO FUENTE DISPONIBLE CON EL LIBRO: V6

puede encontrar la primera actualización del código fuente de este capítulo dentro del v6 repositorio en github: https://github.com/microservices-practical.

incluye la extracción de la interfaz de usuario, por lo que encontrará tres carpetas con los diferentes servicios: Multiplicación, Gamificación, y UI.

Primero, necesitamos descargar e instalar Jetty en nuestro sistema. Usaremos una función útil de Jetty que nos permite crear un nuevo *Base del embarcadero* en una carpeta diferente, manteniendo separados el servidor web y la aplicación web (y su configuración). Este es un buen enfoque porque normalmente nos gustaría mantener nuestra capa de configuración de servidor personalizada bajo control de versiones y separada de los archivos binarios del servidor, lo que facilita mucho una solución futura para la implementación automatizada.

Creamos una nueva base de embarcadero (un ui carpeta) para colocar nuestro contenido estático siguiendo las instrucciones de la sección, "Creación de una nueva base de embarcadero" (ver *https://tpd.io/runjetty).* Obtendremos dos carpetas como resultado: inicio.d y aplicaciones web. Este último representa el contexto raíz de nuestro servidor, por lo que crearemos una nueva carpeta ui adentro aplicaciones web y colocar allí nuestro contexto estático. Listado5-1 muestra la estructura del archivo resultante.

Listado 5-1. Estructura de archivos de la interfaz de usuario (UI v6)

```
ui
├──── inicio.d
│       deploy.ini
│       http.ini
│
└──── aplicaciones web
        └──── ui
```

```
index.html
multiplication-client.js
styles.css
```

El último paso es abrir el http.ini archivo en nuestro editor de texto favorito y comentar y cambiar el valor del embarcadero.http.port a 9090 propiedad, por lo que ejecutamos la interfaz de usuario en un número de puerto diferente, evitando así un conflicto con nuestras aplicaciones Spring Boot.

Ahora podemos ejecutar nuestro servidor Jetty desde la carpeta de la interfaz de usuario (la de nivel superior) y navegar a http: // localhost: 9090 / ui / index.html para ver nuestra página web servida desde nuestro nuevo servidor web. Ver listado5-2.

Listado 5-2. Ejecución de Jetty (UI v6)

```
[/ code / v6 / ui] $ java -jar [YOUR_JETTY_HOME_FOLDER] /start.jar
```

Conectando UI con Gamificación

Ahora es el momento de crear un nuevo archivo JavaScript. gamification-client.js en el que modelará nuestras interacciones con el servicio de gamificación. Ver listado 5-3.

Listado 5-3. gamification-client.js (UI v6)

```
función updateLeaderBoard () {
    $ .ajax ({
        url: "http: // localhost: 8081 / Leaders"}).
    luego (función(datos) {
        $ ('# leaderboard-body'). empty ();
        data.forEach (función(fila) {
            $ ('# leaderboard-body'). append ('<tr> <td>' + fila.
            userId + '</td>' +
```

```
                    '<td>' + fila.totalScore + '</td>');
            });
        });
}

función updateStats (userId) {
        $ .ajax ({
            url: "http: // localhost: 8081 / stats? userId =" + userId,
            éxito: función(datos) {
                $ ('# stats-div'). show (); $ ('# stats-user-id'). empty
                (). append (userId); $ ('# stats-score'). empty ().
                append (data.score); $ ('# stats-badges'). empty ().
                append (data.badges. join ());

            },
            error: función(datos) {
                $ ('# stats-div'). show (); $ ('# stats-user-id').
                empty (). append (userId); $ ('# stats-score').
                empty (). append (0); $ ('# stats-badges'). empty
                ();
            }
        });
}

$ (documento) .ready (function () {

        updateLeaderBoard ();

        $ ("# refresh-leaderboard"). Haga clic en (función( evento) {
            updateLeaderBoard ();
        });

});
```

Consta de un par de funciones que realizarán OBTENER solicitudes al microservicio Gamification (en este caso ejecutándose en el puerto 8081) para recuperar los datos y completar las tablas. Además, proporcionaremos un botón para actualizar la tabla de clasificación (no es sorprendente que se llameactualizar la tabla de clasificación), así que adjuntamos un hacer clic oyente.

Tenga en cuenta que estamos usando las URL http: // localhost: 8081 / ... adentro cliente de gamificación, y http: // localhost: 8080 / ... en multiplication-client.js. No solo estamos codificando las URL, sino que también apuntamos a servicios específicos por sus direcciones de host y puertos, que pueden cambiar con el tiempo. Nunca deberíamos usar este enfoque, porque si movemos nuestros microservicios, necesitaremos cambiar el host, el puerto, el contexto de URI (por ejemplo, /resultados), etc. Otro problema que tenemos al apuntar a puertos específicos es que nuestro sistema no escala de forma transparente. Si queremos incluir una instancia extra del servicio de multiplicación, debemos implementar la lógica para detectarlo y hacer el balanceo de carga desde nuestro cliente web.

Afortunadamente, existen soluciones para resolver este peligroso enfoque que acabamos de tomar. Cubriremos una alternativa mejor en la siguiente sección de este capítulo, utilizando el descubrimiento de servicios y un servicio de puerta de enlace. Pero antes de eso, adaptemos nuestros microservicios a esta nueva arquitectura y demos una nueva mirada a nuestro cliente web.

Cambios en los servicios existentes

Aunque dividir el servicio de interfaz de usuario en un nuevo proyecto podría parecer inofensivo para la multiplicación y la gamificación, ese no es el caso. Serviremos ahora el contenido estático de un*origen*localhost: 9090) que es diferente de donde residen los servicios de backend. En este caso, los números de puerto no son los mismos (8080 y 8081). Tendremos algunos problemas con esto si no aplicamos cambios en el backend, porque Spring Security usa la Política del mismo origen de forma predeterminada. Tenga en cuenta que, en nuestro caso, se trata de diferentes puertos, pero este problema también ocurriría si trabaja con diferentes nombres de host.

Puede ver el problema usted mismo si omite esta parte, continúa con los cambios de la interfaz de usuario y ejecuta todos los servicios como se explica al final de esta sección. Luego, debe abrir las Herramientas de desarrollo dentro de su navegador (por ejemplo, presionando Ctrl + Shift + I en Chrome) y asegúrese de que la consola esté visible (en Chrome es una de las pestañas predeterminadas en la parte inferior). Ahora si navegas ahttp: // localhost: 9090 / ui / index.html, Verá algunos mensajes rojos, como se muestra en la Figura 5-2.

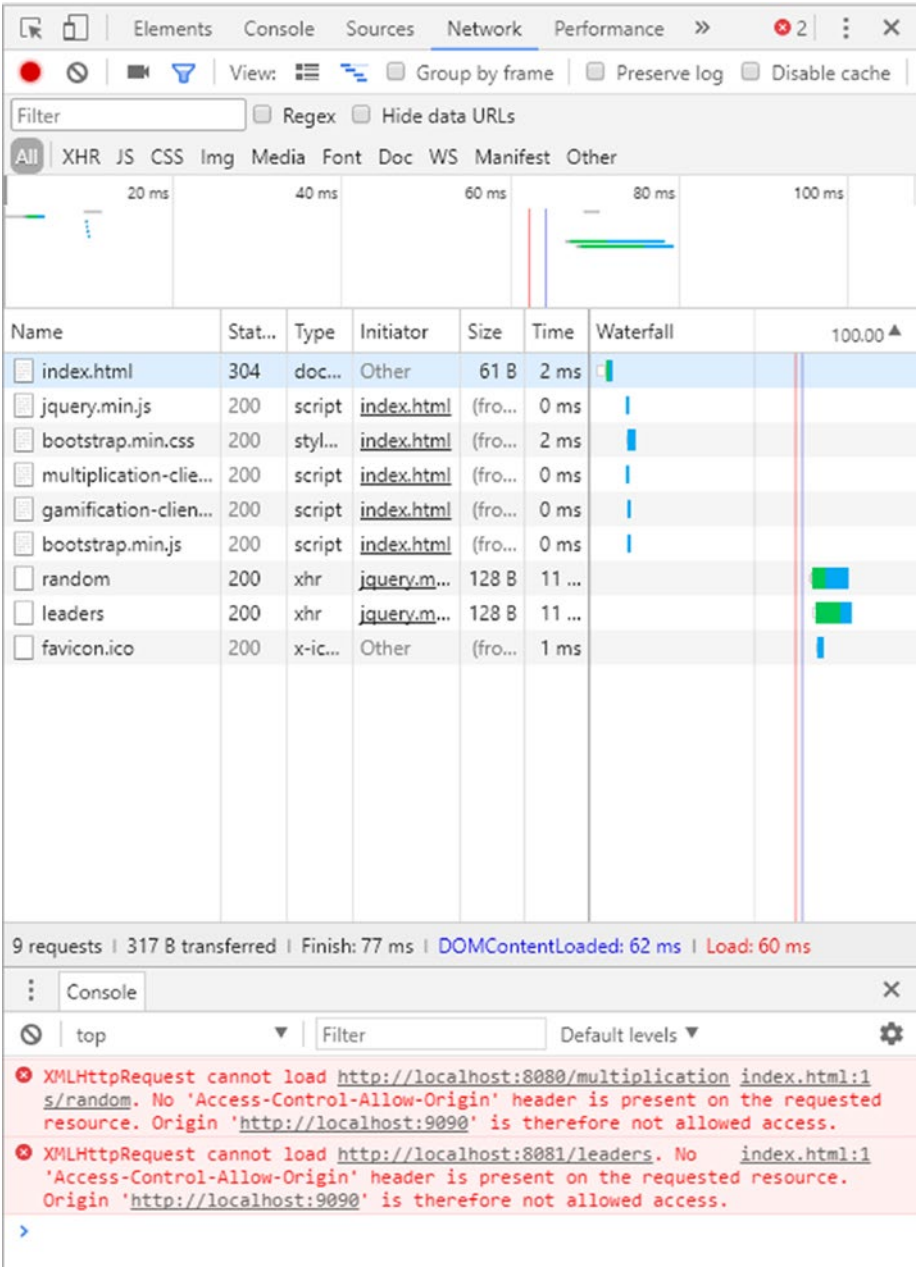

Figura 5-2. *Mensajes de error de CORS*

Para solucionar este problema, debemos habilitar CORS (uso compartido de recursos de origen cruzado) para nuestros servicios de backend para permitir solicitudes que provengan de un origen diferente. Para lograr eso, necesitamos agregar alguna configuración de Spring a nuestros dos servicios. Listado5-4 muestra la clase agregada a la gamificación; también necesitamos crear una clase idéntica dentro de la multiplicación para habilitar CORS allí también.

Listado 5-4. WebConfiguration.java (gamificación v6)

```
paquete microservices.book.gamification.configuration;

importar org.springframework.context.annotation.Configuration;
importar org.springframework.web.servlet.config.annotation.
CorsRegistry;
importar org.springframework.web.servlet.config.annotation.
EnableWebMvc;
importar org.springframework.web.servlet.config.annotation.
WebMvcConfigurerAdapter;

/ **
  * @autor moises.macero
  */
@Configuración
@EnableWebMvc
clase pública WebConfiguration se extiende WebMvcConfigurerAdapter {

    / **
     * Habilita el intercambio de recursos entre orígenes (CORS)
     * Más información: http://docs.spring.io/ spring / docs /
     current / spring-framework-reference / html
     /cors.html
     * @param registro
     */
```

```
    @Anular
    público void addCorsMappings (registro final de CorsRegistry) {
        registro.addMapping ("/ **");
    }
}
```

Tenga en cuenta que, para simplificar, estamos habilitando CORS para cada origen (no especificamos ninguna restricción) y para cada mapeo (con el patrón / **). Cuando su sistema esté maduro y su infraestructura esté configurada, es posible que desee ser más estricto aquí pasando algunos valores de propiedad a sus aplicaciones para permitir solo algunos dominios específicos como orígenes. Para obtener más detalles sobre cómo ajustar su configuración, lea la documentación oficial de Spring en *https://tpd.io/spr-cors.*

Una interfaz de usuario nueva y mejor con (casi) ningún esfuerzo

Hemos entregado hasta ahora un diseño de interfaz de usuario muy minimalista para mantenerlo simple y lanzar nuestra primera versión de la aplicación lo antes posible para que nuestros usuarios puedan comenzar a jugar. Ahora, aprovechando la extracción y dado que necesitamos algo de espacio para nuestra tabla de clasificación, actualizaremos el diseño de nuestra página usando Bootstrap.

Con respecto al diseño de la página, lo dividiremos en dos áreas diferentes:

- En el lado izquierdo, colocaremos el formulario de intento de multiplicación y mostraremos las estadísticas al usuario: puntaje total e insignias (provenientes de la gamificación).
- En el lado derecho, mostraremos la tabla de clasificación con los mejores usuarios (provenientes de Gamificación) y, debajo, la tabla existente con los últimos intentos enviados por el usuario que acaba de jugar (proporcionada por el microservicio de Multiplicación).

No entraremos en detalles profundos sobre Bootstrap, ya que tienen una documentación bastante buena que es muy fácil de seguir (incluso para los desarrolladores de Java, consulte http://getbootstrap.com/css/). La *Sistema de cuadrícula, formularios,* y *Botones* serán las principales características utilizadas en esta aplicación. Ver listado5-5.

Listado 5-5. index.html Adición de Bootstrap (gamificación v6)

```
<! DOCTYPE html>
<html>
<cabeza>
    <título>Multiplicación v1 </título> <enlace href = "css / bootstrap.min.css" rel = "hoja de estilo">
    <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> <secuencia de comandos src = "gamification-client.js"> </secuencia de comandos> <meta name = "viewport" content = "width = device-width, initial-scale = 1">

</head>

<cuerpo>
<div class = "contenedor">
    <div class = "fila">
        <div class = "col-md-12">
            <h1 class = "text-center"> Bienvenido a Social Multiplication </h1>
        </div>
    </div>
    <div class = "fila">
        <div class = "col-md-6">
            <h3 class = "text-center"> Tu nuevo desafío es </h3> <h1 class = "centro de texto">
```

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

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

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