

Construcción de una REST API con Spring Boot

Spring Academy

25 de Julio de 2025

Índice

1. Introducción	6
1.1. ¿Quieres aprender Spring Boot?	6
1.2. ¿Qué construirás?	6
1.3. ¿Qué aprenderás?	6
1.4. Laboratorios prácticos	7
1.5. Prerrequisitos	7
1.6. Basado en proyectos	7
1.7. Enfoque de pruebas primero	7
1.8. Hilo conductor	7
1.9. Continúa aprendiendo	8
2. Spring y Spring Boot	9
2.1. Spring	9
2.2. Spring Boot	9
2.3. El Contenedor de Inversión de Control de Spring	9
2.4. Spring Initializr	10
3. Configuración inicial con Spring Initializr	11
3.1. Pasos para configurar la aplicación	11
3.2. Opciones de configuración	11
3.3. Añadir dependencias	12
3.4. Generar el proyecto	12
3.5. Construir y probar la aplicación	13
3.6. Resumen	13
4. Contratos de API y JSON	14
4.1. ¿Qué son los contratos de API?	14
4.2. ¿Por qué son importantes los contratos de API?	15
4.3. ¿Qué es JSON?	15
5. ¿Qué es el Desarrollo Guiado por Pruebas?	16
5.1. La Pirámide de Pruebas	16
5.2. Pruebas Unitarias	16
5.3. Pruebas de Integración	17
5.4. Pruebas de Extremo a Extremo	17
5.5. El Ciclo Rojo, Verde, Refactorizar	17
6. laboratorio: Desarrollo guiado con pruebas y contratos de datos	18
6.1. Escribiendo una prueba que falle	18
6.2. Pruebas del contrato de datos	20
6.3. Pruebas de deserialización	23
6.4. Resumen	24
7. Implementación de GET	25
7.1. REST, CRUD y HTTP	25
7.2. El cuerpo de la solicitud	26

7.3. Ejemplo de Cash Card	26
7.4. REST en Spring Boot	27
7.5. Anotaciones de Spring y escaneo de componentes	27
7.6. Controladores de Spring Web	27
8. Laboratorio: Implementación de GET	29
8.1. Escribir una prueba de Spring Boot para el endpoint GET	29
8.2. Crear un controlador REST	30
8.3. Añadir el endpoint GET	31
8.4. Completar el endpoint GET	32
8.5. Usar @PathVariable	34
8.6. Resumen	35
9. Repositorios y Spring Data	36
9.1. Arquitectura Controlador-Repositorio	36
9.2. Eleccion de una base de datos	37
9.3. Configuración automática	38
9.4. CrudRepository de Spring Data	38
10.Laboratorio: Configuración de repositorios y bases de datos	40
10.1. Cambios desde talleres anteriores	40
10.1.1. Archivos de recursos de prueba	40
10.2. Revisar el patrón actual de gestión de datos	40
10.3. Añadir dependencias de Spring Data	41
10.4. Crear el CashCardRepository	42
10.5. Inyectar el CashCardRepository	43
10.6. Usar el CashCardRepository para la gestión de datos	45
10.7. Configurar la base de datos	46
10.8. Resumen	48
11.Implementación de POST	49
11.1. Idempotencia y HTTP	49
11.2. La solicitud POST y la respuesta	50
11.2.1. La solicitud	50
11.2.2. La respuesta	50
11.3. Métodos de conveniencia de Spring Web	51
12.Laboratorio: Implementación de POST	52
12.1. Prueba del endpoint HTTP POST	52
12.2. Añade el endpoint POST	53
12.3. Pruebas basadas en corrección semántica	53
12.4. Implementa el endpoint POST	55
12.5. Entiende CrudRepository.save	56
12.6. Entiende los otros cambios en CashCardController	56
12.7. Pruebas finales y momento de aprendizaje	56
12.8. Resumen	58
13.Devolución de una lista con GET	59
13.1. Solicitando una lista de Cash Cards	59

13.2. Paginación y ordenación	60
13.2.1. Consultas sin ordenar	61
13.2.2. API de paginación de Spring Data	61
13.3. La solicitud y la respuesta	62
13.3.1. El URI	62
13.3.2. El código Java	62
14.Laboratorio: Devolución de una lista con GET	65
14.1. Cambios desde el laboratorio anterior	65
14.2. Prueba del nuevo contrato de datos	65
14.3. Prueba para un endpoint GET adicional	69
14.4. Mejora la prueba de la lista	71
14.5. Interacción de pruebas y @DirtiesContext	73
14.6. Paginación	74
14.7. Ordenación	76
14.8. Paginación y ordenación predeterminadas	78
14.9. Resumen	79
15.Simple Spring Security	80
15.1. ¿Qué es la seguridad?	80
15.2. Autenticación	80
15.2.1. Spring Security y Autenticación	80
15.3. Autorización	81
15.4. Política de Origen Igual (Same Origin Policy)	81
15.5. Intercambio de Recursos de Origen Cruzado (CORS)	81
15.6. Explotaciones web comunes	82
15.6.1. Falsificación de Solicitudes entre Sitios (CSRF)	82
15.6.2. Guion Transversal entre Sitios (XSS)	82
16.Laboratorio: Seguridad con Spring Security	83
16.1. Entender nuestros requisitos de seguridad	83
16.2. Revisión de actualizaciones del laboratorio anterior	83
16.3. Añadir la dependencia de Spring Security	84
16.4. Satisfacer las dependencias de Spring Security	85
16.5. Configurar la autenticación básica	87
16.6. Prueba de autenticación básica	88
16.7. Soporte para autorización	89
16.8. Propiedad de Tarjeta Cash: Actualizaciones del Repositorio	91
16.9. Propiedad de Tarjeta Cash: Actualizaciones del Controlador	93
16.10 Propiedad de Tarjeta Cash: Actualizaciones de creación	95
16.11 Acerca de CSRF	97
16.12 Resumen	97
17.Implementación de PUT	98
17.1. Introducción	98
17.2. PUT y PATCH	98
17.3. PUT y POST	98
17.4. Claves surrogadas y naturales	99
17.5. Recursos y sub-recursos	99

17.6. Cuerpo de respuesta y código de estado	100
17.7. POST, PUT, PATCH y operaciones CRUD - Resumen	100
17.8. Seguridad	101
17.9. Nuestras decisiones de API	101
18.Laboratorio: Implementación de PUT	102
18.1. Visión general	102
18.2. Escribir la prueba primero	102
18.3. Implementar @PutMapping en el Controlador	104
18.4. Pruebas adicionales e influencia de Spring Security	106
18.5. Refactorizar el código del Controlador	110
18.6. Resumen	112
19.Implementación de DELETE	113
19.1. Solicitud	113
19.2. Respuesta	113
19.3. Opciones adicionales	113
19.3.1. Eliminación dura y blanda	113
19.3.2. Rastro de auditoría y archivo	114
20.Laboratorio: Implementación de DELETE	116
20.1. Visión general	116
20.2. Prueba del caso feliz	116
20.3. Implementa el endpoint DELETE	118
20.4. Caso de prueba: La Tarjeta Cash no existe	119
20.5. Hacer cumplir la propiedad	119
20.6. Refactorizar	120
20.7. Ocultar registros no autorizados	121
20.8. Resumen	123
21.Resumen rápido: implementación de la API REST CashCard	124
21.1. Modelo: CashCard	124
21.2. Repositorio: CashCardRepository	124
21.3. Controlador: CashCardController	125
21.4. Prueba: obtener tarjeta existente	126
21.5. Prueba: tarjeta no encontrada	126
21.6. Prueba: crear tarjeta	126
21.7. Prueba: actualizar tarjeta existente	127
21.8. Prueba: eliminar tarjeta	127

1 Introducción

1.1 ¿Quieres aprender Spring Boot?

¿Conoces algo de Java? ¿Quieres construir una aplicación real como si estuvieras en un proyecto real? ¿Sí, sí y sí? ¡Genial! ¡Entonces este curso es para ti!

En este curso de nivel principiante, nuestros expertos en Spring te guiarán en la construcción y despliegue de una API REST completamente funcional, segura y bien probada para una aplicación hipotética de Family Cash Card.

1.2 ¿Qué construirás?

Construirás una aplicación simple de Family Cash Card, una forma moderna para que los padres gestionen fondos de mesada para sus hijos.

Los padres a menudo carecen de una manera sencilla de gestionar (es decir, enviar, recibir, rastrear) las mesadas de sus hijos.® En lugar de entregar un fajo de billetes, nuestra aplicación Family Cash Card basada en la nube permite a los padres administrar virtualmente “tarjetas de dinero” para sus hijos.® Puedes pensar en la tarjeta de dinero como algo muy similar a una tarjeta de regalo que muchos enviamos y recibimos. El objetivo principal de la aplicación Family Cash Card es brindar a los padres facilidad y control para gestionar los fondos de sus hijos.

Usarás Spring Boot para progresar desde crear una sola tarjeta de dinero en una base de datos hasta permitir la edición, eliminación y visualización de múltiples tarjetas, y finalmente asegurar tu aplicación contra accesos no autorizados y exploits no deseados.

1.3 ¿Qué aprenderás?

Este curso está diseñado para ayudarte a construir una base sólida en Spring Boot, para que puedas aplicar lo aprendido en tus proyectos del mundo real. Al final del curso, podrás:

- Utilizar Spring Boot para construir una API REST completa mientras comprendes los beneficios y compromisos de REST.
- Aprender qué ofrece Spring Boot a los desarrolladores de aplicaciones y cómo se diferencia del framework Spring.
- Construir aplicaciones web con Spring Web.
- Usar Spring Data para conectar bases de datos y mapear datos relacionales a objetos Java.
- Usar Spring Security para desarrollar software con un enfoque de seguridad primero.

1.4 Laboratorios prácticos

Los laboratorios de este curso proporcionan un terminal y editor interactivos, por lo que no necesitas herramientas específicas instaladas en tu máquina.

1.5 Prerrequisitos

Para aprovechar al máximo este curso, debes tener conocimientos prácticos de Java. También es útil conocer un lenguaje similar, como C#, pero asumimos que ya tienes conocimiento del ecosistema de Java, sus bibliotecas, etc.

1.6 Basado en proyectos

En lugar de estructurar este curso como documentación formal o un libro de texto de referencia, lo hemos enfocado como un proyecto de desarrollo del mundo real. ¿Qué significa esto?

Las lecciones no tratan sobre temas técnicos de Spring Boot. En cambio, están centradas en una tarea específica que podrías encontrar durante un proyecto. Además, en lugar de cubrir cada tecnología de Spring Boot en profundidad, solo abordaremos la tecnología necesaria para completar cada tarea específica y para entender en general qué está sucediendo “bajo el capó” en la aplicación. También explicaremos por qué tomamos las decisiones que tomamos y qué otras opciones y compromisos existen.

Por ejemplo, al cubrir Spring Web para configurar un endpoint de API, introduciremos solo las piezas de Spring Web necesarias para que el endpoint funcione, en lugar de toda la funcionalidad de Spring Web, que es muy robusta.

1.7 Enfoque de pruebas primero

Otra característica única de este curso es nuestro enfoque de *pruebas primero* para el desarrollo. En los laboratorios, normalmente escribirás pruebas antes de la implementación, y luego añadirás la implementación para hacer que las pruebas pasen.

1.8 Hilo conductor

Abordaremos la construcción de nuestra aplicación utilizando un *hilo conductor*. La idea es crear primero un “hilo” que atravesase todos los puntos de integración de la aplicación. Completar el hilo primero tiene ventajas, incluyendo:

- **Entrega incremental:** Resulta en una aplicación ejecutable de extremo a extremo a la que podemos añadir funcionalidades.
- **Mitigación de riesgos:** Puede revelar obstáculos técnicos ocultos en todo el sistema antes que después.

Los tres puntos de integración en nuestra aplicación son la “puerta principal” de la API, la base de datos y la seguridad.

1.9 Continúa aprendiendo

Este curso es una excelente introducción a Spring Boot. Si buscas expandir aún más tus conocimientos, te recomendamos continuar con nuestro curso *Spring Essentials*. ¡Y si añades nuestro curso de Spring Boot, habrás completado todos los prerrequisitos para el examen de *Spring Certified Professional*!

2 Spring y Spring Boot

2.1 Spring

Spring es un framework integral que ofrece diversos módulos para construir diferentes tipos de aplicaciones. Es como tener una caja de herramientas gigante con todas las herramientas que podrías necesitar para construir cualquier cosa. Mientras desarrollamos la API de Family Cash Card, usaremos *Spring MVC* para la aplicación web, *Spring Data* para el acceso a datos y *Spring Security* para la autenticación y autorización.

Esta versatilidad tiene un costo. Configurar una aplicación Spring requiere mucha configuración, y los desarrolladores deben configurar manualmente varios componentes del framework para que la aplicación funcione.

2.2 Spring Boot

Afortunadamente, Spring Boot simplifica trabajar con Spring al ofrecer una versión con configuraciones preestablecidas, pensada para facilitar el desarrollo. Incluye muchas dependencias y ajustes comunes ya definidos, lo que permite comenzar rápidamente sin tener que configurar todo desde cero. Además, incorpora un servidor web embebido, lo que facilita la creación y despliegue de aplicaciones web sin necesidad de un servidor externo.

En resumen, Spring es un framework potente y flexible, pero puede resultar complejo de configurar al inicio. Spring Boot, en cambio, es una alternativa simplificada y lista para usar, que integra muchas funcionalidades de forma predeterminada para que puedas empezar a desarrollar de forma rápida y sencilla.

2.3 El Contenedor de Inversión de Control de Spring

Spring Boot aprovecha el contenedor de *Inversión de Control* (IoC) de Spring Core. Usarás esta característica de Spring extensivamente mientras desarrollas la aplicación Family Cash Card. Hay abundante documentación sobre este concepto, pero aquí lo mantendremos simple.

Spring Boot te permite configurar cómo y cuándo se proporcionan las dependencias a tu aplicación en tiempo de ejecución. Esto te da control sobre cómo opera tu aplicación en diferentes escenarios. Por ejemplo, podrías querer usar una base de datos diferente para el desarrollo local que para la aplicación pública en producción. Tu código de aplicación no debería preocuparse por esta distinción; si lo hicieras, tendrías que codificar cada escenario posible en la lógica de la aplicación. En cambio, Spring Boot te permite proporcionar una configuración externa que especifica cómo y cuándo se usan dichas dependencias.

La inversión de control a menudo se llama *inyección de dependencias* (DI), aunque esto no es estrictamente correcto. La inyección de dependencias y los frameworks asociados son solo una forma de lograr la inversión de control, y los desarrolladores de Spring a menudo dicen que las dependencias son “inyectadas” en sus aplicaciones en tiempo de

ejecución. Hacemos esta distinción porque muchos lenguajes y frameworks implementan IoC, pero no necesariamente lo llaman “inyección de dependencias”. Sin embargo, en la comunidad de Spring, a menudo escucharás estos términos usados indistintamente.

Para más información, consulta la documentación oficial de Spring sobre el contenedor de IoC en:

`https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans`.

2.4 Spring Initializr

Cuando comienzas una nueva aplicación Spring Boot, *Spring Initializr* es el primer paso recomendado. Puedes pensar en Spring Initializr como un carrito de compras para todas las dependencias que tu aplicación podría necesitar. Generará de manera rápida y sencilla una aplicación Spring Boot completa y lista para ejecutar.

En el próximo laboratorio, usaremos Spring Initializr. El flujo general de Spring Initializr consiste en completar los metadatos, añadir las dependencias relevantes y generar tu proyecto. Para la aplicación Family Cash Card, proporcionaremos todas estas instrucciones.

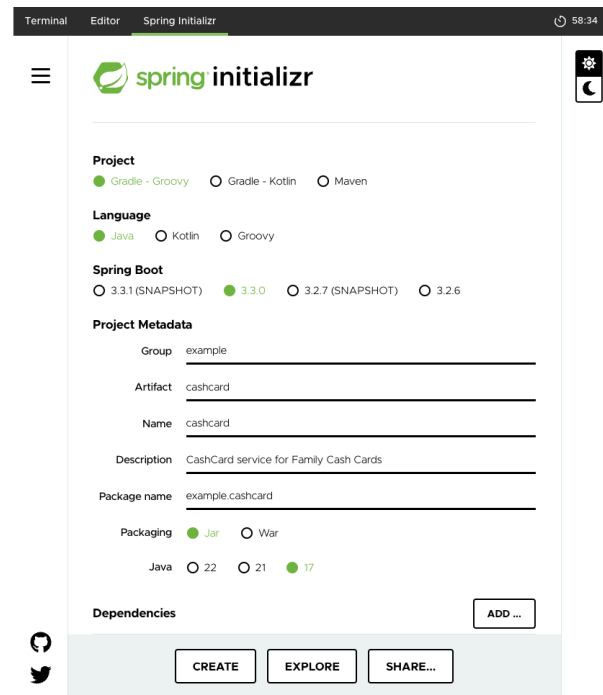
Durante el resto de este curso, añadiremos funcionalidades a la aplicación Spring Boot generada por Initializr.

3 Configuración inicial con Spring Initializr

3.1 Pasos para configurar la aplicación

Completa los siguientes pasos para usar Spring Initializr y configurar la aplicación de la API REST de Family Cash Card.

Abre la pestaña del panel etiquetada como *Spring Initializr*.



The screenshot shows the Spring Initializr web interface. At the top, there are tabs for 'Terminal', 'Editor', and 'Spring Initializr'. The 'Spring Initializr' tab is active. The interface is divided into several sections: 'Project' with radio buttons for 'Gradle - Groovy' (selected), 'Gradle - Kotlin', and 'Maven'; 'Language' with radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'; 'Spring Boot' with radio buttons for '3.3.1 (SNAPSHOT)', '3.3.0' (selected), '3.2.7 (SNAPSHOT)', and '3.2.6'; 'Project Metadata' with text input fields for 'Group' (example), 'Artifact' (cashcard), 'Name' (cashcard), 'Description' (CashCard service for Family Cash Cards), and 'Package name' (example.cashcard); 'Packaging' with radio buttons for 'Jar' (selected) and 'War'; 'Java' with radio buttons for '22', '21', and '17' (selected); and 'Dependencies' with an 'ADD ...' button. At the bottom, there are three buttons: 'CREATE', 'EXPLORE', and 'SHARE...'. The interface also includes a sidebar with a hamburger menu icon and a settings icon.

Figura 1: Panel de Spring Initializr

Nota: Es posible que notes que el panel de Initializr muestra versiones diferentes a las que presentamos aquí. El equipo de Spring actualiza continuamente Initializr con las versiones más recientes de Spring y Spring Boot disponibles.

3.2 Opciones de configuración

Selecciona las siguientes opciones:

- **Proyecto:** Gradle-Groovy
- **Idioma:** Java
- **Spring Boot:** Elige la última versión 3.x.x disponible

Introduce los siguientes valores en los campos correspondientes de *Metadatos del Proyecto*:

- **Group:** example
- **Artifact:** cashcard

- **Name:** CashCard
- **Description:** CashCard service for Family Cash Cards
- **Packaging:** Jar
- **Java:** 17

Nota: No es necesario completar el campo “Package name”; Spring Initializr lo llenará automáticamente.

3.3 Añadir dependencias

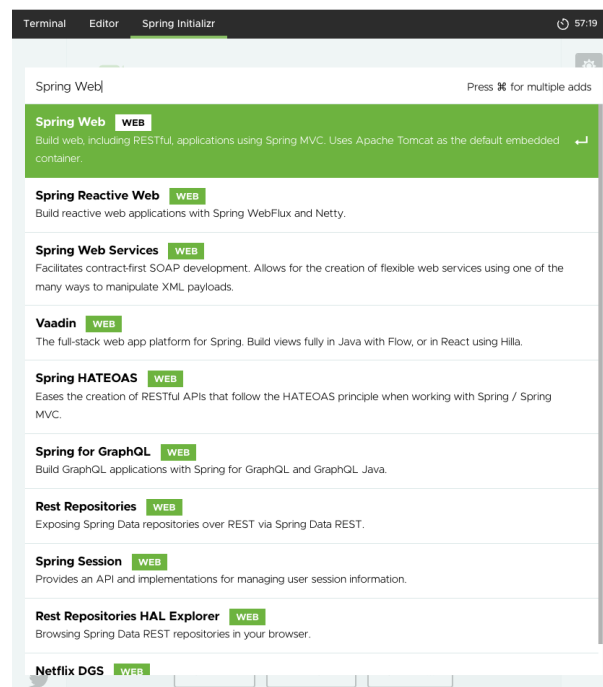


Figura 2: Añadir dependencias en Spring Initializr

Elige la siguiente opción, ya que sabemos que crearemos una aplicación web:

- **Opciones web:** Spring Web

Más adelante en el curso, añadirás dependencias adicionales sin usar Spring Initializr.

3.4 Generar el proyecto

Haz clic en el botón **CREATE**. Spring Initializr generará un archivo zip con el código y lo descomprimirá en tu directorio principal.

3.5 Construir y probar la aplicación

Desde la línea de comandos en la pestaña *Terminal*, introduce los siguientes comandos para usar el wrapper de Gradle para construir y probar la aplicación generada.

Navega al directorio `cashcard` en la pestaña del panel *Terminal*:

```
1 [~] $ cd cashcard
2 [~/cashcard] $
```

A continuación, ejecuta el comando `./gradlew build`:

```
1 [~/cashcard] $ ./gradlew build
```

La salida mostrará que la aplicación pasó las pruebas y se construyó con éxito. Un ejemplo de la salida es:

```
1 Downloading https://services.gradle.org/distributions/gradle-bin.zip
2 .....10%.....20%.....30%.....40%.....
3
4 Welcome to Gradle!
5 ...
6 Starting a Gradle Daemon (subsequent builds will be faster)
7
8 > Task :test
9 ...
10 BUILD SUCCESSFUL in 39s
11 7 actionable tasks: 7 executed
```

3.6 Resumen

¡Felicidades! Acabas de aprender cómo iniciar rápidamente un código base de Spring Boot utilizando Spring Initializr.

4 Contratos de API y JSON

4.1 ¿Qué son los contratos de API?

Estamos desarrollando una API, lo que plantea muchas preguntas sobre cómo debería comportarse:

- ¿Cómo deben interactuar los consumidores de la API con ella?
- ¿Qué datos necesitan enviar los consumidores en diferentes escenarios?
- ¿Qué datos debería devolver la API a los consumidores y en qué momentos?
- ¿Qué comunica la API cuando se usa incorrectamente o algo sale mal?

Siempre que sea posible, los proveedores y consumidores de la API deben discutir estos escenarios y llegar a acuerdos. Mejor aún, deberían documentar estos acuerdos no solo en un sistema de documentación compartido, sino también de manera que permita que las pruebas automatizadas pasen (o fallen) basándose en estas decisiones. Aquí es donde entra en juego el concepto de contratos de API.

Un *contrato de API* es un acuerdo formal entre un proveedor de software y un consumidor que comunica de manera abstracta cómo interactuar entre sí. Este contrato define cómo interactúan los proveedores y consumidores de la API, cómo son los intercambios de datos y cómo se comunican los casos de éxito y fallo.

El proveedor y los consumidores no necesitan compartir el mismo lenguaje de programación, solo los mismos contratos de API. Para el dominio de Family Cash Card, asumamos que actualmente existe un contrato entre el servicio de Cash Card y todos los servicios que lo utilizan. A continuación, se muestra un ejemplo de ese primer contrato de API. No te preocupes si no entiendes todo el contrato; abordaremos cada aspecto a lo largo de este curso.

```
1 Solicitud:
2   URI: /cashcards/{id}
3   Verbo HTTP: GET
4   Cuerpo: Ninguno
5
6 Respuesta:
7   Estado HTTP:
8     200 OK si el usuario esta autorizado y la Cash Card se recupero con exito
9     401 UNAUTHORIZED si el usuario no esta autenticado o no esta autorizado
10    404 NOT FOUND si el usuario esta autenticado y autorizado pero la Cash Card no se
    encuentra
11  Tipo de cuerpo de la respuesta: JSON
12  Ejemplo de cuerpo de la respuesta:
13    {
14      "id": 99,
15      "amount": 123.45
16    }
```

4.2 ¿Por qué son importantes los contratos de API?

Los contratos de API son importantes porque comunican el comportamiento de una API REST. Proporcionan detalles específicos sobre los datos que se serializan (o deserializan) para cada comando y parámetro intercambiado. Los contratos de API están escritos de manera que puedan traducirse fácilmente en funcionalidades para el proveedor y el consumidor de la API, así como en pruebas automatizadas correspondientes. En los laboratorios, implementaremos tanto la funcionalidad del proveedor de la API como las pruebas automatizadas.

4.3 ¿Qué es JSON?

JSON (*JavaScript Object Notation*) proporciona un formato de intercambio de datos que representa la información específica de un objeto de manera que sea fácil de leer y entender. Usaremos JSON como nuestro formato de intercambio de datos para la API de Family Cash Card.

A continuación, se muestra el ejemplo utilizado anteriormente:

```
1 {  
2   "id": 99,  
3   "amount": 123.45  
4 }
```

Otros formatos de datos populares incluyen YAML (*Yet Another Markup Language*) y XML (*Extensible Markup Language*). En comparación con XML, JSON se lee y escribe más rápido, es más fácil de usar y ocupa menos espacio. Puedes usar JSON con la mayoría de los lenguajes de programación modernos y en todas las plataformas principales. También funciona de manera fluida con aplicaciones basadas en JavaScript.

Por estas razones, JSON ha reemplazado en gran medida a XML como el formato más utilizado para APIs en aplicaciones web, incluidas las APIs REST.

5 ¿Qué es el Desarrollo Guiado por Pruebas?

Es común que los equipos de desarrollo de software creen suites de pruebas automatizadas para protegerse contra regresiones. A menudo, estas pruebas se escriben después de que se ha desarrollado el código de la aplicación o sus funcionalidades. En este curso, adoptaremos un enfoque alternativo: escribir pruebas antes de implementar el código de la aplicación. Esto se conoce como *Desarrollo Guiado por Pruebas* (TDD, por sus siglas en inglés).

¿Por qué aplicar TDD? Al definir el comportamiento esperado antes de implementar la funcionalidad deseada, diseñamos el sistema basándonos en lo que queremos que haga, en lugar de lo que el sistema ya hace.

Otro beneficio de “guiar” el código de la aplicación con pruebas es que estas pruebas nos orientan a escribir el mínimo código necesario para cumplir con la implementación. Cuando las pruebas pasan, tienes una implementación funcional (el código de la aplicación) y una protección contra errores futuros (las pruebas).

¿No estás seguro de cómo implementar TDD? No te preocupes, a lo largo de este curso practicarás el desarrollo guiado por pruebas con la aplicación Family Cash Card.

5.1 La Pirámide de Pruebas

Las pruebas pueden escribirse en diferentes niveles del sistema. En cada nivel, hay un equilibrio entre la velocidad de ejecución, el “costo” de mantener las pruebas y la confianza que aportan sobre la corrección del sistema. Esta jerarquía se representa comúnmente como una “pirámide de pruebas”.

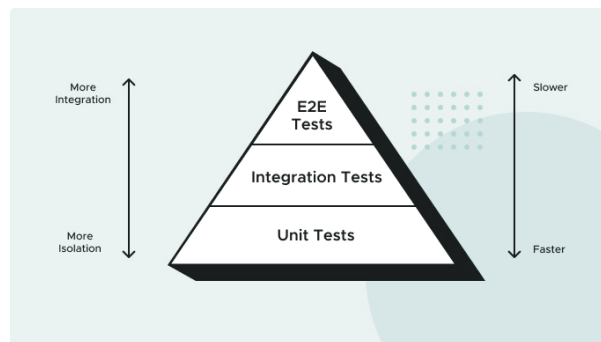


Figura 3: Pirámide de pruebas: Unitarias, Integración y Extremo a Extremo

5.2 Pruebas Unitarias

Una prueba unitaria evalúa una pequeña “unidad” del sistema, aislada del resto. Estas pruebas deben ser simples y rápidas. Querrás tener una alta proporción de pruebas unitarias en tu pirámide de pruebas, ya que son clave para diseñar software altamente cohesivo y débilmente acoplado.

5.3 Pruebas de Integración

Las pruebas de integración evalúan un subconjunto del sistema y pueden incluir grupos de unidades en una sola prueba. Son más complicadas de escribir y mantener, y se ejecutan más lentamente que las pruebas unitarias.

5.4 Pruebas de Extremo a Extremo

Una prueba de extremo a extremo evalúa el sistema utilizando la misma interfaz que usaría un usuario, como un navegador web. Aunque son extremadamente exhaustivas, las pruebas de extremo a extremo pueden ser muy lentas y frágiles porque utilizan interacciones de usuario simuladas en interfaces potencialmente complejas. Implementa la menor cantidad posible de estas pruebas.

5.5 El Ciclo Rojo, Verde, Refactorizar

Los equipos de desarrollo de software buscan moverse rápido. ¿Cómo mantener esa velocidad indefinidamente? Mejorando y simplificando continuamente el código mediante *refactorización*. Una de las únicas formas de refactorizar de manera segura es contar con una suite de pruebas confiable. Por lo tanto, el mejor momento para refactorizar el código en el que estás trabajando es durante el ciclo de TDD, conocido como el ciclo *Rojo, Verde, Refactorizar*:

1. **Rojo:** Escribe una prueba que falle para la funcionalidad deseada.
2. **Verde:** Implementa lo más simple posible para hacer que la prueba pase.
3. **Refactorizar:** Busca oportunidades para simplificar, reducir duplicaciones o mejorar el código sin cambiar su comportamiento.
4. ¡Repíte!

A lo largo de los laboratorios de este curso, practicarás el ciclo Rojo, Verde, Refactorizar para desarrollar la API REST de Family Cash Card.

6 laboratorio: Desarrollo guiado con pruebas y contratos de datos

6.1 Escribiendo una prueba que falle

Aquí cubriremos una breve introducción a la biblioteca de pruebas JUnit y a la herramienta de compilación Gradle. También utilizaremos el enfoque de *pruebas primero* para construir software.

Las clases de prueba en un proyecto Java estándar se encuentran en el directorio “src/test”, no en “src/main”. En nuestro caso, hemos decidido colocar nuestro código en el paquete “example.cashcard”, por lo que los archivos de prueba deben estar en el directorio “src/test/java/example/cashcard”.

Crea la clase de prueba “CashCardJsonTest”.

Lo primero que debemos hacer es crear nuestra nueva clase de prueba en el directorio “src/test/java/example/cashcard”.

Puedes hacerlo haciendo clic en la acción “click-action” a continuación o utilizando el Editor a la derecha de la siguiente manera:

- Crea “CashCardJsonTest.java”.
- Abre la sección *JAVA PROJECTS* en la parte inferior izquierda de la página.
- Abre el directorio “src/test/java”.
- Dentro de “src/test/java”, selecciona el paquete “example.cashcard” y haz clic en el signo “+” a la derecha.
- Aparecerá un diálogo solicitando el nombre de la nueva clase. Ingresas “CashCardJsonTest.java”.

Aquí hay una captura de pantalla que muestra los pasos anteriores: Creación de una clase de prueba en el paquete correcto usando Visual Studio Code.

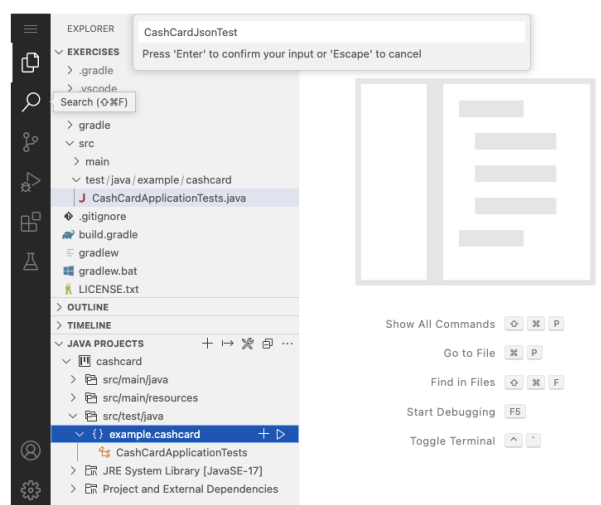


Figura 4: Creación de la clase de prueba CashCardJsonTest

Edita el archivo resultante para que contenga el siguiente contenido:

```
1 package example.cashcard;
2
3 import org.junit.jupiter.api.Test;
4 import static org.assertj.core.api.Assertions.assertThat;
5
6 class CashCardJsonTest {
7
8     @Test
9     void myFirstTest() {
10         assertThat(1).isEqualTo(42);
11     }
12 }
```

Tómate un momento para entender el código de prueba: La anotación “@Test” forma parte de la biblioteca JUnit, y el método “assertThat” pertenece a la biblioteca AssertJ. Ambas bibliotecas se importan después de la declaración del paquete.

Una convención común (aunque no obligatoria) es usar el sufijo “Test” para las clases de prueba. Lo hemos hecho aquí. El nombre completo “CashCardJsonTest” te da una pista sobre la naturaleza de la prueba que estamos a punto de escribir.

Siguiendo el enfoque de *pruebas primero*, hemos escrito primero una prueba que falla. Es importante tener una prueba que falle primero para tener alta confianza de que lo que hiciste para corregirla realmente funcionó.

No te preocupes si la prueba (que afirma que 1 es igual a 42) y el nombre del método de prueba parecen extraños. Estamos a punto de cambiarlos.

Ejecuta la prueba desde la línea de comandos en tu terminal (asegúrate de estar en el directorio de ejercicios primero):

```
1 [~/exercises] $ ./gradlew test
```

Deberías recibir una salida como esta (hemos omitido algunas salidas menos importantes):

```
1 > Task :test
2
3 CashCardApplicationTests > contextLoads() PASSED
4
5 CashCardJsonTest > myFirstTest() FAILED
6     org.opentest4j.AssertionFailedError:
7         expected: 42
8         but was: 1
9     ...
10         at
11         app//example.cashcard.CashCardJsonTest.myFirstTest(CashCardJsonTest.java:11)
12 2 tests completed, 1 failed
```

Esta es la salida esperada de la herramienta de compilación Gradle cuando tienes una prueba que falla. En este caso, tu nueva prueba falló, mientras que la prueba existente “CashCardsApplicationTest” de la lección anterior tuvo éxito.

La información relevante sobre el fallo está hacia la parte superior de la salida:

```
expected: 42
but was: 1
```

Es posible que lo hayas esperado, ya que el número 1 no es igual al número 42.

Para “corregir” la prueba, puedes afirmar una declaración que sepas que es verdadera:

```
1 assertThat(42).isEqualTo(42);
```

Ahora ejecuta la prueba nuevamente. ¡Pasa!

```
1 [~/exercises] $ ./gradlew test
2
3 > Task :test
4
5 CashCardJsonTest > myFirstTest() PASSED
6
7 CashCardApplicationTests > contextLoads() PASSED
8
9 BUILD SUCCESSFUL in 4s
```

¡Felicidades! Has completado con éxito una iteración de desarrollo basado en pruebas: Escribe una prueba que falle, luego corrige el código para que la prueba pase. Ahora estás listo para proceder con la metodología de *pruebas primero* para escribir la API REST de Cash Card.

6.2 Pruebas del contrato de datos

Ahora, escribamos una prueba que tenga sentido para tu objetivo: escribir la API REST de Cash Card.

Reemplaza la prueba muy simple del ejercicio anterior con una prueba más completa, para que el archivo de prueba quede así:

```
1 package example.cashcard;
2
3 import org.junit.jupiter.api.Test;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.boot.test.autoconfigure.json.JsonTest;
6 import org.springframework.boot.test.json.JacksonTester;
7
8 import java.io.IOException;
9
10 import static org.assertj.core.api.Assertions.assertThat;
11
12 @JsonTest
13 class CashCardJsonTest {
14
15     @Autowired
16     private JacksonTester<CashCard> json;
17
18     @Test
19     void cashCardSerializationTest() throws IOException {
20         CashCard cashCard = new CashCard(99L, 123.45);
21         assertThat(json.write(cashCard)).isEqualToJson("expected.json");
22         assertThat(json.write(cashCard)).hasJsonPathNumberValue("@.id");
```

```

23     assertThat(json.write(cashCard)).extractingJsonPathNumberValue("@.id")
24         .isEqualTo(99);
25     assertThat(json.write(cashCard)).hasJsonPathNumberValue("@.amount");
26     assertThat(json.write(cashCard)).extractingJsonPathNumberValue("@.amount")
27         .isEqualTo(123.45);
28 }
29 }

```

La anotación “@JsonTest” marca “CashCardJsonTest” como una clase de prueba que utiliza el framework Jackson (incluido como parte de Spring). Esto proporciona un soporte extenso para pruebas y análisis de JSON. También establece todo el comportamiento relacionado para probar objetos JSON.

“JacksonTester” es un contenedor de conveniencia para la biblioteca de análisis JSON Jackson. Maneja la serialización y deserialización de objetos JSON. “@Autowired” es una anotación que dirige a Spring a crear un objeto del tipo solicitado.

Ejecuta la prueba.

Ejecuta “cashCardSerializationTest()” que acabas de crear para probar la serialización de la clase ‘CashCard’.

```

1 [~/exercises] $ ./gradlew test

1 > Task :compileTestJava
2 /home/eduk8s/src/test/java/example/cashcard/CashCardJsonTest.java:16: error: cannot
   find symbol
3     private JacksonTester<CashCard> json;
4         ~
5     symbol:   class CashCard
6     location: class CashCardJsonTest
7 ...
8 > Task :compileTestJava FAILED
9
10 FAILURE: Build failed with an exception.
11
12 * What went wrong:
13 Execution failed for task ':compileTestJava'.
14 > Compilation failed; see the compiler error output for details.

```

No es sorprendente que la prueba haya fallado, ya que la clase “CashCard” aún no existe. Creemos una ahora.

Crea la clase “CashCard”.

Para crear la clase “CashCard” y el constructor utilizado en la prueba “cashCardSerializationTest ()”, crea el archivo “src/main/java/example/cashcard/CashCard.java” con el siguiente contenido (nota que este archivo está en el directorio “src/main”, no en “src/-test”):

También puedes crear “CashCard.java” haciendo clic en la “click-action” a continuación.

```

1 package example.cashcard;
2
3 record CashCard(Long id, Double amount) {
4 }

```

Vuelve a ejecutar la prueba.

Deberías recibir un fallo porque el archivo “expected.json” aún no existe:

```

1 [~/exercises] $ ./gradlew test
2 ...
3 CashCardJsonTest > cashCardSerializationTest() FAILED
4     java.lang.IllegalStateException: Unable to load JSON from class path resource
       [example/cashcard/expected.json]
5 ...
6         at example.cashcard.CashCardJsonTest.cashCardSerializationTest
7         (CashCardJsonTest.java:21)
8
9         Caused by:
10        java.io.FileNotFoundException: class path resource
       [example/cashcard/expected.json] cannot be opened because it does not exist
11 ...
12 2 tests completed, 1 failed
13
14 > Task :test FAILED

```

La forma más rápida de añadir un nuevo archivo “expected.json” es:

Crea el archivo del contrato de Cash Card.

Crea un archivo llamado “expected.json” en “src/test/resources/example/cashcard/expected.json”, con el siguiente contenido. También puedes usar la siguiente “click-action” para crear “expected.json” si lo deseas.

```

1 {}

```

Nota que necesitarás crear la estructura de directorios para contener el nuevo archivo. Una forma de hacerlo es la siguiente:

- Haz clic derecho en la parte de prueba del directorio “src/test” en el editor.
- Selecciona *New File...*
- En el diálogo, ingresa “resources/example/cashcard/expected.json” como nombre del archivo.
- Edita el archivo “expected.json” recién creado y escribe {} como único contenido.

Estamos incluyendo intencionalmente un documento JSON vacío {}, lo que hará que la prueba falle.

Ejecuta la prueba.

La prueba falla nuevamente, pero esta vez es por un fallo de comparación. Los mensajes de error indican que faltan dos campos:

```

1 CashCardJsonTest > cashCardSerializationTest() FAILED
2     java.lang.AssertionError: JSON Comparison failure:
3     Unexpected: amount
4     ;
5     Unexpected: id
6     at
       example.cashcard.CashCardJsonTest.cashCardSerializationTest(CashCardJsonTest.java:21)

```

Completa el contrato de datos.

Para añadir campos al contrato de datos en el archivo “src/test/resources/example/cash-card/expected.json”, cambia el contenido del archivo “expected.json” al siguiente:

```
1 {
2   "id": 99,
3   "amount": 123.45
4 }
```

Vuelve a ejecutar la prueba.

```
1 [~/exercises] $ ./gradlew test
2 ...
3 BUILD SUCCESSFUL in 4s
```

¡Pasa ahora que las aserciones coinciden con el archivo de contrato!

¡Felicidades! Has implementado TDD para crear un contrato de datos para tu API de CashCard.

6.3 Pruebas de deserialización

La deserialización es el proceso inverso de la serialización. @ Transforma datos desde un archivo o flujo de bytes de vuelta a un objeto para tu aplicación. @ Esto hace posible que un objeto serializado en una plataforma pueda deserializarse en una plataforma diferente. @ Por ejemplo, tu aplicación cliente puede serializar un objeto en Windows mientras el backend lo deserializa en Linux.

La serialización y la deserialización trabajan juntas para transformar/recrear objetos de datos a/from un formato portátil. @ El formato de datos más popular para serializar datos es JSON.

Escribamos una segunda prueba para deserializar datos, de modo que convierta de JSON a Java después de que la primera prueba pase. @ Esta prueba utiliza una técnica de *pruebas primero* donde escribes intencionalmente una prueba que falla. @ Específicamente: los valores para ‘id’ y ‘amount’ no son los que esperas.

En el archivo ‘src/test/java/example/cashcard/CashCardJsonTest.java’, añade una prueba:

```
1 @Test
2 void cashCardDeserializationTest() throws IOException {
3     String expected = """
4         {
5             "id":99,
6             "amount":123.45
7         }
8     """;
9     assertThat(json.parse(expected))
10        .isEqualTo(new CashCard(1000L, 67.89));
11     assertThat(json.parseObject(expected).id()).isEqualTo(1000);
12     assertThat(json.parseObject(expected).amount()).isEqualTo(67.89);
13 }
```

Ejecuta la prueba.

La prueba falla:

```
1 CashCardJsonTest > cashCardDeserializationTest() FAILED
2 org.opentest4j.AssertionFailedError:
3   expected: CashCard[id=1000, amount=67.89]
4   but was: CashCard[id=99, amount=123.45]
```

Corrige los valores erróneos de 'id' y 'amount'.

Puedes corregirlos uno a la vez. @ Asegúrate de volver a ejecutar la prueba después de cada edición.

Aquí es cómo se ven ahora las aserciones corregidas:

```
1 ...
2 assertThat(json.parse(expected))
3     .isEqualTo(new CashCard(99L, 123.45));
4 assertThat(json.parseObject(expected).id()).isEqualTo(99);
5 assertThat(json.parseObject(expected).amount()).isEqualTo(123.45);
6 ...
```

¡Eso es todo! Ahora tienes un par de pruebas de serialización/deserialización funcionales.

6.4 Resumen

En esta lección aprendiste sobre JSON y su importancia para aplicaciones modernas. @ También aprendiste cómo se utiliza JSON en la aplicación CashCard. @ Finalmente, aprendiste los beneficios del enfoque de *pruebas primero* en el desarrollo de software y ejercitaste este enfoque al probar el contrato de datos JSON para el servicio de Cash Card.

7 Implementación de GET

7.1 REST, CRUD y HTTP

En esta lección, aprenderás qué es REST y cómo usar Spring Boot para implementar un solo endpoint RESTful.

Comencemos con una definición concisa de REST: *Representational State Transfer* (Transferencia de Estado Representacional).@ En un sistema RESTful, los objetos de datos se llaman *Representaciones de Recursos*.@ El propósito de una API RESTful (Interfaz de Programación de Aplicaciones) es gestionar el estado de estos Recursos.

Dicho de otra manera, puedes pensar en “estado” como “valor” y en “Representación de Recurso” como un “objeto” o “cosa”.@ Por lo tanto, REST es solo una forma de gestionar los valores de las cosas.@ Esas cosas podrían ser accedidas a través de una API y a menudo se almacenan en un almacén de datos persistente, como una base de datos.

Un concepto frecuentemente mencionado al hablar de REST es CRUD. CRUD significa “Crear, Leer, Actualizar y Eliminar”, que son las cuatro operaciones básicas que se pueden realizar en objetos en un almacén de datos.@ Aprenderemos que REST tiene pautas específicas para implementar cada una.

Otro concepto común asociado con REST es el Protocolo de Transferencia de Hipertexto (HTTP).@ En HTTP, un llamador envía una Solicitud a un URI.@ Un servidor web recibe la solicitud y la enruta a un manejador de solicitudes.@ El manejador crea una Respuesta, que luego se envía de vuelta al llamador.

Los componentes de la Solicitud y la Respuesta son:

- **Solicitud**

- Método (también llamado Verbo)
- URI (también llamado Endpoint)
- Cuerpo

- **Respuesta**

- Código de Estado
- Cuerpo

Si deseas profundizar más en los métodos de Solicitud y Respuesta, consulta el estándar HTTP.

El poder de REST radica en la forma en que referencia un Recurso y en cómo se ven la Solicitud y la Respuesta para cada operación CRUD.@ Veamos cómo será nuestra API cuando terminemos este curso:

- Para CREAR: usa el método HTTP POST.
- Para LEER: usa el método HTTP GET.
- Para ACTUALIZAR: usa el método HTTP PUT.

- Para ELIMINAR: usa el método HTTP DELETE.

El URI del endpoint para los objetos Cash Card comienza con la palabra clave ‘/cashcards’.@ Las operaciones de LEER, ACTUALIZAR y ELIMINAR requieren que proporcionemos el identificador único del recurso objetivo.@ La aplicación necesita este identificador único para realizar la acción correcta exactamente en el recurso correcto.@ Por ejemplo, para LEER, ACTUALIZAR o ELIMINAR una Cash Card con el identificador “42”, el endpoint sería ‘/cashcards/42’.

Nota que no proporcionamos un identificador único para la operación CREAR.@ Como aprenderemos con más detalle en lecciones futuras, CREAR tendrá el efecto secundario de crear una nueva Cash Card con un nuevo identificador único.@ No se debe proporcionar un identificador al crear una nueva Cash Card porque la aplicación creará un nuevo identificador único para nosotros.

La tabla a continuación tiene más detalles sobre las operaciones CRUD RESTful.

Operación	Endpoint de API	Método HTTP	Código de Estado de Respuesta
Crear	/cashcards	POST	201 (CREATED)
Leer	/cashcards/id	GET	200 (OK)
Actualizar	/cashcards/id	PUT	204 (NO_CONTENT)
Eliminar	/cashcards/id	DELETE	204 (NO_CONTENT)

7.2 El cuerpo de la solicitud

Al seguir las convenciones REST para crear o actualizar un recurso, necesitamos enviar datos a la API.@ Esto a menudo se conoce como el cuerpo de la solicitud.@ Las operaciones CREAR y ACTUALIZAR requieren que el cuerpo de la solicitud contenga los datos necesarios para crear o actualizar adecuadamente el recurso.@ Por ejemplo, una nueva Cash Card podría tener un monto de valor inicial en efectivo, y una operación ACTUALIZAR podría cambiar ese monto.

7.3 Ejemplo de Cash Card

Usemos el ejemplo de un endpoint de Lectura.@ Para la operación de Lectura, la ruta del URI (endpoint) es ‘/cashcards/id’, donde ‘id’ se reemplaza por un identificador real de Cash Card, sin las llaves, y el método HTTP es GET.

En las solicitudes GET, el cuerpo está vacío.@ Por lo tanto, la solicitud para leer la Cash Card con un id de 123 sería:

```
1 Solicitud:
2   Metodo: GET
3   URL: http://cashcard.example.com/cashcards/123
4   Cuerpo: (vacío)
```

La respuesta a una solicitud de Lectura exitosa tiene un cuerpo que contiene la representación JSON del Recurso solicitado, con un Código de Estado de Respuesta de 200 (OK).@ Por lo tanto, la respuesta a la solicitud de Lectura anterior se vería así:

```
1 Respuesta:
2  Codigo de Estado: 200
3  Cuerpo:
4   {
5     "id": 123,
6     "amount": 25.00
7   }
```

A medida que avanzamos en este curso, aprenderás a implementar todas las operaciones CRUD restantes.

7.4 REST en Spring Boot

Ahora que hemos discutido REST en general, veamos las partes de Spring Boot que usaremos para implementarlo. Comencemos discutiendo el contenedor IoC de Spring.

7.5 Anotaciones de Spring y escaneo de componentes

Una de las principales funciones de Spring es configurar e instanciar objetos. Estos objetos se llaman *Spring Beans* y generalmente son creados por Spring (en lugar de usar la palabra clave ‘new’ de Java). Puedes dirigir a Spring a crear Beans de varias maneras.

En esta lección, anotarás una clase con una Anotación de Spring, lo que indica a Spring que cree una instancia de la clase durante la fase de Escaneo de Componentes de Spring. Esto ocurre al iniciar la aplicación. El Bean se almacena en el contenedor IoC de Spring. Desde aquí, el bean puede ser inyectado en cualquier código que lo solicite.

7.6 Controladores de Spring Web

En Spring Web, las Solicitudes son manejadas por Controladores. En esta lección, usarás el más específico ‘RestController’:

```
1 @RestController
2 class CashCardController {
3 }
```

Eso es todo lo que se necesita para decirle a Spring: “crea un Controlador REST”. El Controlador se inyecta en Spring Web, que enruta las solicitudes de API (manejadas por el Controlador) al método correcto.

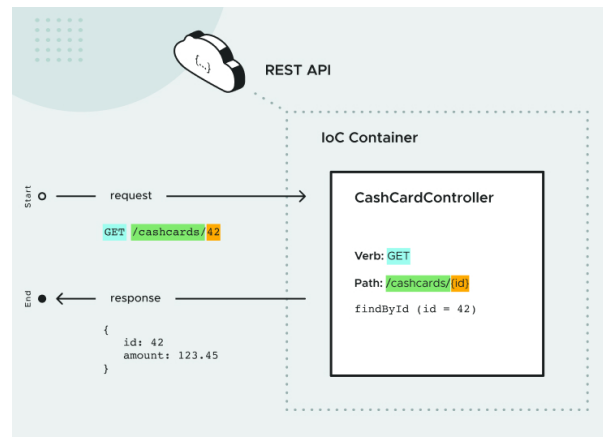


Figura 5: Controlador de Spring Web manejando solicitudes

Un método de Controlador puede ser designado como un método manejador, para ser llamado cuando se reciba una solicitud que el método sabe cómo manejar (llamada “solicitud coincidente”). ¡Escribamos un método manejador de solicitud de Lectura! Aquí hay un comienzo:

```
1 private CashCard findById(Long requestedId) {
2 }
```

Dado que REST indica que los endpoints de Lectura deben usar el método HTTP GET, necesitas decirle a Spring que enrute las solicitudes al método solo en solicitudes GET. Puedes usar la anotación ‘@GetMapping’, que necesita la ruta URI:

```
1 @GetMapping("/cashcards/{requestedId}")
2 private CashCard findById(Long requestedId) {
3 }
```

Spring necesita saber cómo obtener el valor del parámetro ‘requestedId’. Esto se hace usando la anotación ‘@PathVariable’. El hecho de que el nombre del parámetro coincida con el texto ‘requestedId’ dentro del parámetro ‘@GetMapping’ permite a Spring asignar (inyectar) el valor correcto a la variable ‘requestedId’:

```
1 @GetMapping("/cashcards/{requestedId}")
2 private CashCard findById(@PathVariable Long requestedId) {
3 }
```

REST indica que la Respuesta debe contener una Cash Card en su cuerpo y un código de Respuesta de 200 (OK). Spring Web proporciona la clase ‘ResponseEntity’ para este propósito. También ofrece varios métodos utilitarios para producir Entidades de Respuesta. Aquí, puedes usar ‘ResponseEntity’ para crear una ‘ResponseEntity’ con código 200 (OK) y un cuerpo que contiene una ‘CashCard’. La implementación final se ve así:

```
1 @RestController
2 class CashCardController {
3     @GetMapping("/cashcards/{requestedId}")
4     private ResponseEntity<CashCard> findById(@PathVariable Long requestedId) {
5         CashCard cashCard = /* Aqui estaria el codigo para recuperar la CashCard */;
6         return ResponseEntity.ok(cashCard);
7     }
8 }
```

8 Laboratorio: Implementación de GET

8.1 Escribir una prueba de Spring Boot para el endpoint GET

Al igual que si estuviéramos en un proyecto real, usemos el desarrollo dirigido por pruebas para implementar nuestro primer endpoint de API.

Escribe la prueba.

Comencemos implementando una prueba usando '@SpringBootTest' de Spring.

Actualiza 'src/test/java/example/cashcard/CashCardApplicationTests.java' con lo siguiente:

```
1 package example.cashcard;
2
3 import com.jayway.jsonpath.DocumentContext;
4 import com.jayway.jsonpath.JsonPath;
5 import org.junit.jupiter.api.Test;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.boot.test.context.SpringBootTest;
8 import org.springframework.boot.test.web.client.TestRestTemplate;
9 import org.springframework.http.HttpStatus;
10 import org.springframework.http.ResponseEntity;
11
12 import static org.assertj.core.api.Assertions.assertThat;
13
14 @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
15 class CashCardApplicationTests {
16     @Autowired
17     TestRestTemplate restTemplate;
18
19     @Test
20     void shouldReturnACashCardWhenDataIsSaved() {
21         ResponseEntity<String> response = restTemplate.getForEntity("/cashcards/99",
22             String.class);
23
24         assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
25     }
26 }
```

Entiende la prueba.

Entendamos varios elementos importantes en esta prueba.

- `@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)`: Esto iniciara nuestra aplicación Spring Boot y la hara disponible para que nuestra prueba realice solicitudes a ella.
- `@Autowired TestRestTemplate restTemplate`: Hemos pedido a Spring que inyecte una herramienta de prueba que nos permitira realizar solicitudes HTTP a la aplicación en ejecución local.
- **Nota:** Aunque `@Autowired` es una forma de inyeccion de dependencias de Spring, es mejor usarlo solo en pruebas. @ No te preocupes, lo discutiremos con mas detalle mas adelante.

- `ResponseEntity<String>response = restTemplate.getForEntity("/cashcards/99", String.class);` Aquí usamos `restTemplate` para realizar una solicitud HTTP GET a nuestro endpoint de aplicación `/cashcards/99`.
- `assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);` Podemos inspeccionar muchos aspectos de la respuesta, incluido el código de estado HTTP, que esperamos que sea 200 OK.

Ahora ejecuta la prueba.

¿Qué crees que pasará cuando ejecutemos la prueba?

Fallará, como era de esperarse. ¿Por qué? Como hemos aprendido en la práctica de pruebas primero, describimos nuestras expectativas antes de implementar el código que satisface esas expectativas.

Ahora, ejecutemos la prueba. @ Nota que ejecutaremos `./gradlew test` para cada ejecución de prueba.

```
1 [~/exercises] $ ./gradlew test
```

¡Falla! Busca en la salida lo siguiente:

```
1 CashCardApplicationTests > shouldReturnACashCardWhenDataIsSaved() FAILED
2   org.opentest4j.AssertionFailedError:
3     expected: 200 OK
4     but was: 404 NOT_FOUND
```

Pero ¿por qué estamos obteniendo este fallo específico?

Entiende el fallo de la prueba.

Como explicamos, esperábamos que nuestra prueba fallara actualmente.

¿Por que falla debido a un código de respuesta HTTP 404 NOT_FOUND inesperado?

Respuesta: Como no hemos indicado a Spring Web como manejar `GET cashcards/99`, Spring Web responde automáticamente que el endpoint no se encuentra.

¡Gracias por manejarlo por nosotros, Spring Web!

A continuación, hagamos que nuestra aplicación funcione correctamente.

8.2 Crear un controlador REST

Los controladores de Spring Web están diseñados para manejar y responder a solicitudes HTTP.

Crea el controlador.

Crea la clase de controlador en `'src/main/java/example/cashcard/CashCardController.java'`.

Crea `'CashCardController.java'`

```
1 package example.cashcard;
2
3 import org.springframework.http.ResponseEntity;
4 import org.springframework.web.bind.annotation.GetMapping;
```

```

5 import org.springframework.web.bind.annotation.PathVariable;
6 import org.springframework.web.bind.annotation.RequestMapping;
7 import org.springframework.web.bind.annotation.RestController;
8
9 class CashCardController {
10 }

```

Añade el método manejador.

Implementa un método `findById()` para manejar las solicitudes HTTP entrantes.

```

1 class CashCardController {
2     private ResponseEntity<String> findById() {
3         return ResponseEntity.ok("{}");
4     }
5 }

```

Ahora vuelve a ejecutar la prueba.

¿Qué esperamos que pase cuando reejecutemos las pruebas?

```

expected: 200 OK
but was: 404 NOT_FOUND

```

¡Mismo resultado! ¿Por qué?

A pesar del nombre, `CashCardController` no es realmente un controlador de Spring Web; es solo una clase con “Controller” en el nombre. @ Por lo tanto, no está “escuchando” nuestras solicitudes HTTP. @ Por lo tanto, necesitamos decirle a Spring que haga que el controlador esté disponible como un controlador web para manejar solicitudes a URLs `cashcards/*`.

8.3 Añadir el endpoint GET

Actualiza el controlador.

Actualicemos nuestro `CashCardController` para que esté configurado para escuchar y manejar solicitudes HTTP a `/cashcards`.

```

1 @RestController
2 @RequestMapping("/cashcards")
3 class CashCardController {
4
5     @GetMapping("/{requestedId}")
6     private ResponseEntity<String> findById() {
7         return ResponseEntity.ok("{}");
8     }
9 }

```

Entiende las anotaciones de Spring Web.

Revisemos nuestras adiciones.

- **@RestController**: Le indica a Spring que esta clase es un Componente de tipo `RestController` y capaz de manejar solicitudes HTTP.

- `@RequestMapping(/cashcards")`: Es un complemento de `@RestController` que indica qué dirección deben tener las solicitudes para acceder a este controlador.
- `@GetMapping(/requestedId") private ResponseEntity<String>findById() ...:`
`@GetMapping` marca un método como método manejador. @ Las solicitudes GET que coincidan con `cashcards/requestedId` serán manejadas por este método.

Ejecuta las pruebas.

¡Pasan!

```
1 [~/exercises] $ ./gradlew test
2 ...
3 BUILD SUCCESSFUL in 6s
```

Finalmente tenemos un controlador y un método manejador que coinciden con la solicitud realizada en nuestra prueba. ¡Genial!

8.4 Completar el endpoint GET

Hasta ahora, nuestra prueba solo afirma que la solicitud tuvo éxito al verificar un código de estado de respuesta 200 OK. A continuación, probemos que la respuesta contiene los valores correctos.

Actualiza la prueba.

```
1 assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
2
3 DocumentContext documentContext = JsonPath.parse(response.getBody());
4 Number id = documentContext.read("$.id");
5 assertThat(id).isNotNull();
```

Entiende las adiciones.

- `DocumentContext documentContext = JsonPath.parse(response.getBody());`: Esto convierte la cadena de respuesta en un objeto consciente de JSON con muchos métodos de ayuda.
- `Number id = documentContext.read("$.id"); assertThat(id).isNotNull();`: Esperamos que al solicitar una Cash Card con id de 99 se devuelva un objeto JSON con algo en el campo `id`. @ Por ahora, afirmamos que el `id` no es nulo.

Ejecuta la prueba y nota el fallo.

Dado que devolvemos un objeto JSON vacío {}, no debería sorprendernos que el campo `id` esté vacío.

```
1 CashCardApplicationTests > shouldReturnACashCardWhenDataIsSaved() FAILED
2 com.jayway.jsonpath.PathNotFoundException: No results for path: $['id']
```

Devuelve una Cash Card desde el controlador.

Hagamos que la prueba pase, pero devolvamos algo intencionalmente incorrecto, como 1000L. @ Verás por qué más tarde.

Además, utilicemos la clase de modelo de datos `CashCard` que creamos en una lección anterior. @ Revisa bajo `src/main/java/example/cashcard/CashCard.java` si es necesario.

```
1 @GetMapping("/{requestedId}")
2 private ResponseEntity<CashCard> findById() {
3     CashCard cashCard = new CashCard(1000L, 0.0);
4     return ResponseEntity.ok(cashCard);
5 }
```

Ejecuta la prueba.

¡Pasa! Sin embargo, ¿se siente correcto? No realmente. @ Que la prueba pase con datos incorrectos parece equivocado.

Te pedimos que devolvieras intencionalmente un `id` incorrecto de 1000L para ilustrar un punto: Es importante que las pruebas pasen o fallen por la razón correcta.

Actualiza la prueba.

Actualiza la prueba para afirmar que el `id` es correcto.

```
1 DocumentContext documentContext = JsonPath.parse(response.getBody());
2 Number id = documentContext.read("$.id");
3 assertThat(id).isEqualTo(99);
```

Vuelve a ejecutar las pruebas y nota el nuevo mensaje de fallo.

```
1 expected: 99
2 but was: 1000
```

Ahora, la prueba falla por la razón correcta: No devolvimos el `id` correcto.

Corrige `CashCardController`.

Actualiza `CashCardController` para devolver el `id` correcto.

```
1 @GetMapping("/{requestedId}")
2 private ResponseEntity<CashCard> findById() {
3     CashCard cashCard = new CashCard(99L, 0.0);
4     return ResponseEntity.ok(cashCard);
5 }
```

Ejecuta la prueba.

¡Hurra, pasa!

Prueba el monto.

A continuación, agreguemos una afirmación para el monto indicado por el contrato JSON.

```
1 DocumentContext documentContext = JsonPath.parse(response.getBody());
2 Number id = documentContext.read("$.id");
3 assertThat(id).isEqualTo(99);
4
5 Double amount = documentContext.read("$.amount");
6 assertThat(amount).isEqualTo(123.45);
```

Ejecuta las pruebas y observa el fallo.

Como era de esperar, no devolvemos el monto correcto en la respuesta.

```
1 expected: 123.45
2 but was: 0.0
```

Devuelve el monto correcto.

Actualicemos `CashCardController` para devolver el monto indicado por el contrato JSON.

```
1 @GetMapping("/{requestedId}")
2 private ResponseEntity<CashCard> findById() {
3     CashCard cashCard = new CashCard(99L, 123.45);
4     return ResponseEntity.ok(cashCard);
5 }
```

Vuelve a ejecutar las pruebas.

¡Pasan! Excelente.

```
1 BUILD SUCCESSFUL in 6s
```

8.5 Usar @PathVariable

Hasta ahora, hemos ignorado el `requestedId` en el método manejador del controlador. Usemos esta variable de ruta en nuestro controlador para asegurarnos de devolver la Cash Card correcta.

Añade un nuevo método de prueba.

Escribamos una nueva prueba que espere ignorar Cash Cards que no tengan un id de 99. Usa 1000, como lo hemos hecho en pruebas anteriores.

```
1 @Test
2 void shouldNotReturnACashCardWithAnUnknownId() {
3     ResponseEntity<String> response = restTemplate.getForEntity("/cashcards/1000",
4         String.class);
5     assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
6     assertThat(response.getBody()).isBlank();
7 }
```

Nota que estamos esperando un código de estado de respuesta HTTP semántico de 404 NOT_FOUND. Si solicitamos una Cash Card que no existe, entonces esa Cash Card efectivamente “no se encuentra”.

Ejecuta la prueba y nota el resultado.

```
expected: 404 NOT_FOUND
but was: 200 OK
```

Añade `@PathVariable`.

Hagamos que la prueba pase haciendo que el controlador devuelva la Cash Card específica solo si enviamos el identificador correcto.

Para hacerlo, primero haz que el controlador esté consciente de la variable de ruta que estamos enviando, añadiendo la anotación `@PathVariable` al argumento del método manejador.

```
1 @GetMapping("/{requestedId}")
2 private ResponseEntity<CashCard> findById(@PathVariable Long requestedId) {
3     CashCard cashCard = new CashCard(99L, 123.45);
4     return ResponseEntity.ok(cashCard);
5 }
```

`@PathVariable` hace que Spring Web esté consciente del `requestedId` proporcionado en la solicitud HTTP. Ahora está disponible para que lo usemos en nuestro método manejador.

Utiliza `@PathVariable`.

Actualiza el método manejador para devolver una respuesta vacía con estado `NOT_FOUND` a menos que el `requestedId` sea 99.

```
1 @GetMapping("/{requestedId}")
2 private ResponseEntity<CashCard> findById(@PathVariable Long requestedId) {
3     if (requestedId.equals(99L)) {
4         CashCard cashCard = new CashCard(99L, 123.45);
5         return ResponseEntity.ok(cashCard);
6     } else {
7         return ResponseEntity.notFound().build();
8     }
9 }
```

Vuelve a ejecutar las pruebas.

¡Genial! ¡Pasan!

```
1 [~/exercises] $ ./gradlew test
2 ...
3 BUILD SUCCESSFUL in 6s
```

8.6 Resumen

¡Felicidades! En esta lección aprendiste a usar el desarrollo dirigido por pruebas para crear tu primer endpoint REST de Family Cash Card: un GET que devuelve una `CashCard` con un cierto ID.

9 Repositorios y Spring Data

En este punto de nuestro viaje de desarrollo, tenemos un sistema que devuelve un registro de Cash Card codificado de forma estática desde nuestro controlador.® Sin embargo, lo que realmente queremos es devolver datos reales desde una base de datos. ¡Así que continuemos nuestro *Steel Thread* cambiando nuestra atención a la base de datos!

Spring Data funciona con Spring Boot para hacer que la integración con bases de datos sea sencilla.® Antes de sumergirnos, hablemos brevemente sobre la arquitectura de Spring Data.

9.1 Arquitectura Controlador-Repositorio

El principio de Separación de Preocupaciones establece que un software bien diseñado debe ser modular, con cada módulo teniendo preocupaciones distintas y separadas de cualquier otro módulo.

Hasta ahora, nuestro código base solo devuelve una respuesta codificada de forma estática desde el controlador.® Esta configuración viola el principio de Separación de Preocupaciones al mezclar las preocupaciones de un controlador, que es una abstracción de una interfaz web, con las preocupaciones de leer y escribir datos en un almacén de datos, como una base de datos.® Para solucionar esto, usaremos un patrón de arquitectura de software común para imponer la separación de la gestión de datos a través del patrón de Repositorio.

Un marco arquitectónico común que divide estas capas, típicamente por función o valor, como las capas de negocio, datos y presentación, se llama Arquitectura en Capas.® En este sentido, podemos pensar en nuestro Repositorio y Controlador como dos capas en una Arquitectura en Capas.® El Controlador está en una capa cerca del Cliente (ya que recibe y responde a solicitudes web) mientras que el Repositorio está en una capa cerca del almacén de datos (ya que lee de y escribe en el almacén de datos).® Pueden haber capas intermedias también, según las necesidades del negocio. ¡No necesitamos capas adicionales, al menos no todavía!

El Repositorio es la interfaz entre la aplicación y la base de datos, y proporciona una abstracción común para cualquier base de datos, lo que facilita cambiar a una base de datos diferente cuando sea necesario.

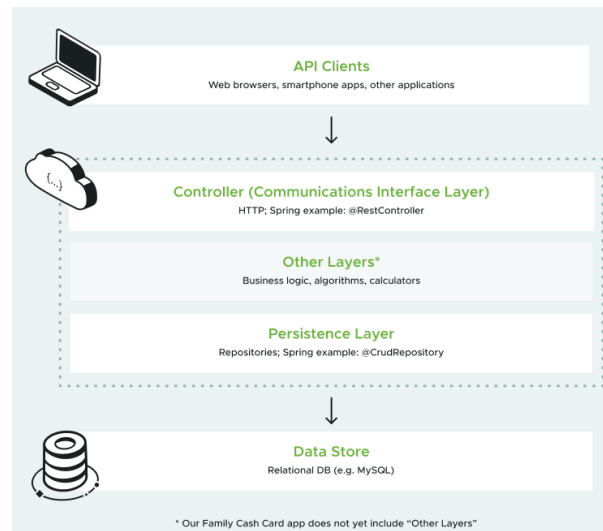


Figura 6: Arquitectura de capas: Controlador y Repositorio

Buenas noticias, Spring Data proporciona una colección de herramientas robustas de gestión de datos, incluyendo implementaciones del patrón de Repositorio.

9.2 Eleccion de una base de datos

Para la selección de nuestra base de datos, usaremos una base de datos incrustada en memoria. “Incrustada” simplemente significa que es una biblioteca Java, por lo que puede añadirse al proyecto como cualquier otra dependencia. “En memoria” significa que almacena datos solo en memoria, en oposición a persistir datos en almacenamiento permanente y duradero.® Al mismo tiempo, nuestra base de datos en memoria es en gran medida compatible con sistemas de gestión de bases de datos relacionales (RDBMS) de producción como MySQL, SQL Server y muchos otros.® Específicamente, usa JDBC (la biblioteca estándar de Java para conectividad de bases de datos) y SQL (el lenguaje de consulta de bases de datos estándar).

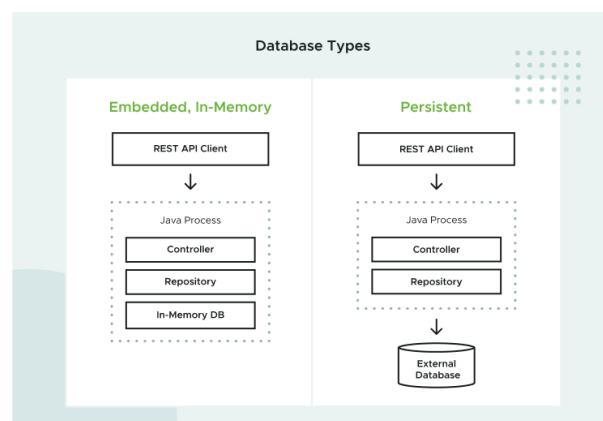


Figura 7: Base de datos incrustada en memoria versus externa

Hay compensaciones al usar una base de datos en memoria en lugar de una base de

datos persistente. @ Por un lado, en memoria permite desarrollar sin instalar un RDBMS separado y asegura que la base de datos esté en el mismo estado (es decir, vacía) en cada ejecución de prueba. @ Sin embargo, necesitas una base de datos persistente para la aplicación “en producción”. @ Esto lleva a una discrepancia de ****Paridad Dev-Prod****: Tu aplicación podría comportarse de manera diferente al ejecutar la base de datos en memoria que al ejecutarla en producción.

La base de datos en memoria específica que usaremos es H2. @ Afortunadamente, H2 es altamente compatible con otras bases de datos relacionales, por lo que la paridad dev-prod no será un gran problema. @ Usaremos H2 por conveniencia para el desarrollo local, pero queremos reconocer las compensaciones.

9.3 Configuración automática

En el laboratorio, todo lo que necesitamos para la funcionalidad completa de la base de datos es añadir dos dependencias. @ Esto muestra maravillosamente una de las características más poderosas de Spring Boot: Configuración Automática. @ Sin Spring Boot, tendríamos que configurar Spring Data para comunicarse con H2. @ Sin embargo, porque hemos incluido la dependencia de Spring Data (y un proveedor de datos específico, H2), Spring Boot configurará automáticamente tu aplicación para comunicarse con H2.

9.4 CrudRepository de Spring Data

Para nuestra selección de Repositorio, usaremos un tipo específico de Repositorio: **CrudRepository** de Spring Data. A primera vista, parece un poco mágico, pero desglosemos esa magia.

Lo siguiente es una implementación completa de todas las operaciones CRUD al extender **CrudRepository**:

```
1 interface CashCardRepository extends CrudRepository<CashCard, Long> {  
2 }
```

Con solo el código anterior, un llamador puede llamar a cualquier número de métodos predefinidos de **CrudRepository**, como **findById**:

```
1 cashCard = cashCardRepository.findById(99);
```

Podrías preguntarte de inmediato:

¿Dónde está la implementación del método **CashCardRepository.findById()**?

CrudRepository y todo lo que hereda de él es una Interfaz sin código real. @ Bueno, basado en el marco específico de Spring Data usado (que para nosotros será Spring Data JDBC), Spring Data se encarga de esta implementación durante el tiempo de inicio del contenedor IoC. @ El tiempo de ejecución de Spring luego expone el repositorio como otro bean que puedes referenciar donde sea necesario en tu aplicación.

Como hemos aprendido, normalmente hay compensaciones. @ Por ejemplo, **CrudRepository** genera sentencias SQL para leer y escribir tus datos, lo cual es útil para muchos casos, pero a veces necesitas escribir tus propias sentencias SQL personalizadas para casos de

uso específicos. @ Por ahora, estamos felices de aprovechar sus métodos convenientes fuera de la caja, así que sigamos adelante con el laboratorio.

10 Laboratorio: Configuración de repositorios y bases de datos

10.1 Cambios desde talleres anteriores

Si has tomado los laboratorios anteriores de este curso, notarás los siguientes cambios, que hemos realizado en tu nombre para hacer este laboratorio más fácil de entender y completar.

10.1.1. Archivos de recursos de prueba

Hemos proporcionado los siguientes archivos que utilizarás en este laboratorio.

- `src/main/resources/schema.sql`
- `src/test/resources/data.sql`

```
1 /* CREATE TABLE cash_card
2 (
3     ID      BIGINT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
4     AMOUNT  NUMBER NOT NULL DEFAULT 0
5 );
6 */

1 /* INSERT INTO CASH_CARD(ID, AMOUNT) VALUES (99, 123.45); */
```

10.2 Revisar el patrón actual de gestión de datos

Nuestra API REST de Family Cash Card actualmente depende de datos de CashCard codificados directamente en nuestro `CashCardController`.

Nuestras pruebas en `CashCardApplicationTests` afirman que estos datos están presentes.

Sabemos que un controlador web no debería gestionar datos.® Esto es una violación del principio de Separación de Preocupaciones.® El tráfico web es tráfico web, los datos son datos, y un software saludable tiene arquitecturas dedicadas a cada área.

Revisa `CashCardController`.

Nota líneas como las siguientes:

```
1 ...
2 if (requestedId.equals(99L)) {
3     CashCard cashCard = new CashCard(99L, 123.45);
4     return ResponseEntity.ok(cashCard);
5 }
6 ...
```

Esto es gestión de datos.® Nuestro controlador no debería preocuparse por verificar IDs o crear datos.

Revisa `CashCardApplicationTests`.

Curiosamente, aunque nuestras pruebas hacen afirmaciones sobre los datos, no dependen ni especifican cómo se crean o gestionan esos datos.

Este desacoplamiento es importante, ya que nos ayuda a realizar los cambios que necesitamos.

Prepara la refactorización para usar un Repositorio y una base de datos.

La refactorización es el acto de alterar la implementación de un sistema de software sin alterar sus entradas, salidas o comportamiento.

Nuestras pruebas nos permitirán cambiar la implementación de la gestión de datos de nuestra API de Cash Card de datos codificados dentro de nuestro controlador a utilizar un Repositorio y una base de datos.

Este laboratorio es un ejemplo continuo del ciclo de desarrollo Rojo, Verde, Refactor que aprendimos en una lección anterior.

A medida que refactorizamos, nuestras pruebas fallarán periódicamente cuando las ejecutemos. Sabremos que hemos eliminado con éxito todos los datos codificados de nuestro controlador y "migrado" esos datos (y la gestión de datos) a un Repositorio respaldado por base de datos cuando nuestras pruebas pasen nuevamente.

10.3 Añadir dependencias de Spring Data

Este proyecto fue creado originalmente usando Spring Initializr, lo que nos permitió añadir dependencias automáticamente a nuestro proyecto. Sin embargo, ahora debemos añadir dependencias manualmente a nuestro proyecto.

Añade dependencias para Spring Data y una base de datos.

En `build.gradle`:

```
1 dependencies {
2     implementation 'org.springframework.boot:spring-boot-starter-web'
3     testImplementation 'org.springframework.boot:spring-boot-starter-test'
4
5     // Añade las dos dependencias a continuación
6     implementation 'org.springframework.data:spring-data-jdbc'
7     implementation 'com.h2database:h2'
8 }
```

Entiende las dependencias.

Las dos dependencias que añadimos están relacionadas, pero son diferentes.

- `implementation 'org.springframework.data:spring-data-jdbc'`: Spring Data tiene muchas implementaciones para una variedad de tecnologías de bases de datos relacionales y no relacionales. Spring Data también tiene varias abstracciones sobre esas tecnologías. Estas se conocen comúnmente como un marco de Mapeo Objeto-Relacional, o ORM.

- Aquí elegiremos usar Spring Data JDBC. @ De la documentación de Spring Data JDBC: “Spring Data JDBC busca ser conceptualmente sencillo... @ Esto hace de Spring Data JDBC un ORM simple, limitado y con opiniones definidas.”
- `implementation 'com.h2database:h2'`: Los marcos de gestión de bases de datos solo funcionan si tienen una base de datos vinculada. H2 es una base de datos SQL “muy rápida, de código abierto, API JDBC” implementada en Java. @ Funciona sin problemas con Spring Data JDBC.

Ejecuta las pruebas.

Esto instalará las dependencias y verificará que su adición no haya roto nada.

Siempre usaremos `./gradlew test` para ejecutar nuestras pruebas.

```
1 [~/exercises] $ ./gradlew test
2 ...
3 BUILD SUCCESSFUL in 4s
```

¡Las dependencias están instaladas! Podrías notar salida adicional en comparación con laboratorios anteriores, como "Shutting down embedded database".

La Configuración Automática de Spring ahora está iniciando y configurando una base de datos H2 para que la usemos con las pruebas. ¡Genial!

10.4 Crear el CashCardRepository

Crea el `CashCardRepository`.

Crea `src/main/java/example/cashcard/CashCardRepository.java` y haz que extienda `CrudRepository`.

```
1 package example.cashcard;
2
3 import org.springframework.data.repository.CrudRepository;
4
5 interface CashCardRepository extends CrudRepository {
6 }
```

Entiende `extends CrudRepository`.

Aquí es donde aprovechamos la magia de Spring Data y su patrón de repositorio de datos.

`CrudRepository` es una interfaz proporcionada por Spring Data. @ Cuando la extendemos (u otras subinterfases de `Repository` de Spring Data), Spring Boot y Spring Data trabajan juntos para generar automáticamente los métodos CRUD que necesitamos para interactuar con una base de datos.

Usaremos uno de estos métodos CRUD, `findById`, más adelante en el laboratorio.

Ejecuta las pruebas.

Podemos ver que todo se compila, sin embargo, nuestra aplicación se estrella gravemente al iniciar. @ Investigando los mensajes de fallo, encontramos esto:

```

1 [~/exercises] $ ./gradlew test
2 ...
3 CashCardApplicationTests > shouldNotReturnACashCardWithAnUnknownId() FAILED
4 java.lang.IllegalStateException: Failed to load ApplicationContext for ...
5
6 Caused by:
7 java.lang.IllegalArgumentException: Could not resolve domain type of interface
   example.cashcard.CashCardRepository
8 ...

```

Este error críptico significa que no hemos indicado qué tipo de datos debe gestionar el `CashCardRepository`.@ Para nuestra aplicación, el "tipo de dominio" de este repositorio será el `CashCard`.

Configura el `CashCardRepository`.

Edita el `CashCardRepository` para especificar que gestiona los datos de `CashCard`, y que el tipo de datos del ID de Cash Card es `Long`.

```

1 interface CashCardRepository extends CrudRepository<CashCard, Long> {
2 }

```

Configura el `CashCard`.

Cuando configuramos el repositorio como `CrudRepository<CashCard, Long>` indicamos que el ID de `CashCard` es `Long`.@ Sin embargo, aún necesitamos decirle a Spring Data qué campo es el ID.

Edita la clase `CashCard` para configurar el id como el `@Id` para el `CashCardRepository`.

No olvides añadir la nueva importación.

```

1 package example.cashcard;
2
3 // Anade esta importacion
4 import org.springframework.data.annotation.Id;
5
6 record CashCard(@Id Long id, Double amount) {
7 }

```

Ejecuta las pruebas.

```

1 [~/exercises] $ ./gradlew test
2 ...
3 BUILD SUCCESSFUL in 4s

```

Las pruebas pasan, pero no hemos hecho cambios significativos en el código... ¡todavía!

10.5 Inyectar el CashCardRepository

Aunque hemos configurado nuestras clases `CashCard` y `CashCardRepository`, no hemos utilizado el nuevo `CashCardRepository` para gestionar nuestros datos de `CashCard`. ¡Hagámoslo ahora!

Inyecta el `CashCardRepository` en `CashCardController`.

Edita `CashCardController` para aceptar un `CashCardRepository`.

```

1 @RestController
2 @RequestMapping("/cashcards")
3 class CashCardController {
4     private final CashCardRepository cashCardRepository;
5
6     private CashCardController(CashCardRepository cashCardRepository) {
7         this.cashCardRepository = cashCardRepository;
8     }
9     ...
10 }

```

Ejecuta las pruebas.

Si ejecutas las pruebas ahora, todas pasarán, a pesar de que no hay otros cambios en el código base que utilicen el nuevo constructor requerido `CashCardController(CashCardRepository cashCardRepository)`.

```
1 BUILD SUCCESSFUL in 7s
```

Entonces, ¿cómo es esto posible?

¡Contempla la Configuración Automática y la Inyección por Constructor!

La Configuración Automática de Spring está utilizando su marco de inyección de dependencias (DI), específicamente la inyección por constructor, para suministrar `CashCardController` con la implementación correcta de `CashCardRepository` en tiempo de ejecución.

¡Cosas mágicas!

Momento de aprendizaje: Elimina la DI.

Acabamos de presenciar la gloria de la configuración automática y la inyección por constructor.

Pero, ¿qué pasa cuando desactivamos esta maravilla?

Cambia temporalmente el `CashCardRepository` para eliminar la implementación de `CrudRepository`.

```

1 interface CashCardRepository {
2 }

```

Compila el proyecto y nota el fallo.

```
1 [~/exercises] $ ./gradlew build
```

Buscando en la salida, encontramos esta línea:

```
org.springframework.beans.factory.NoSuchBeanDefinitionException: No qualifying bean of type [org.springframework.data.repository.CrudRepository] for dependency: [org.springframework.data.repository.CrudRepository]
```

Pistas como `NoSuchBeanDefinitionException`, `No qualifying bean`, y `expected at least 1 bean which qualifies as autowire candidate` nos dicen que Spring está intentando encontrar una clase configurada correctamente para proporcionar durante la fase de inyección de dependencias de la Configuración Automática, pero ninguna califica.

Podemos satisfacer este requisito de DI extendiendo el `CrudRepository`.

Deshace todos esos cambios.

Asegúrate de deshacer los cambios temporales en `CashCardRepository` antes de continuar.

```
1 interface CashCardRepository extends CrudRepository<CashCard, Long> {  
2 }
```

10.6 Usar el CashCardRepository para la gestión de datos

¡Finalmente estás listo para usar el `CashCardRepository`!

Encuentra la `CashCard` usando `findById`.

La interfaz `CrudRepository` proporciona muchos métodos útiles, incluyendo `findById(ID id)`.

Actualiza el `CashCardController` para utilizar este método en el `CashCardRepository` y actualiza la lógica; asegúrate de importar `java.util.Optional`;

```
1 import java.util.Optional;  
2 ...  
3 @GetMapping("/{requestedId}")  
4 private ResponseEntity<CashCard> findById(@PathVariable Long requestedId) {  
5     Optional<CashCard> cashCardOptional = cashCardRepository.findById(requestedId);  
6     if (cashCardOptional.isPresent()) {  
7         return ResponseEntity.ok(cashCardOptional.get());  
8     } else {  
9         return ResponseEntity.notFound().build();  
10    }  
11 }
```

Entiende los cambios.

Acabamos de alterar `CashCardController.findById` de varias maneras importantes.

- `Optional<CashCard>cashCardOptional = cashCardRepository.findById(requestedId);`: Estamos llamando a `CrudRepository.findById`, que devuelve un `Optional`.@ Este objeto inteligente puede o no contener la `CashCard` que estamos buscando.@ Aprende más sobre `Optional` aquí.
- `cashCardOptional.isPresent()` y `cashCardOptional.get()`: Esto es cómo determines si `findById` encontró o no la `CashCard` con el ID proporcionado.
- Si `cashCardOptional.isPresent()` es verdadero, entonces el repositorio encontró con éxito la `CashCard` y podemos recuperarla con `cashCardOptional.get()`.
- Si no, el repositorio no encontró la `CashCard`.

Ejecuta las pruebas.

Podemos ver que las pruebas fallan con un error `500 INTERNAL_SERVER_ERROR`.

```
1 CashCardApplicationTests > shouldReturnACashCardWhenDataIsSaved() FAILED  
2 org.opentest4j.AssertionFailedError:  
3 expected: 200 OK  
4 but was: 500 INTERNAL\_SERVER\_ERROR
```

Esto significa que la API de Cash Card "se estrelló".

Necesitamos un poco más de información...

Actualicemos temporalmente la sección de salida de pruebas de `build.gradle` con `showStandardStreams = true`, para que nuestras ejecuciones de prueba produzcan mucha más salida.

```

1 test {
2   testLogging {
3     events "passed", "skipped", "failed" //, "standardOut", "standardError"
4
5     showExceptions true
6     exceptionFormat "full"
7     showCauses true
8     showStackTraces true
9
10    // Cambia de false a true
11    showStandardStreams = true
12  }
13 }
```

Vuelve a ejecutar las pruebas.

Nota que la salida de la prueba es mucho más verbose.

Buscando en la salida encontramos estos fallos:

```
org.h2.jdbc.JdbcSQLException: Table "CASH\_CARD" not found (this database
  SELECT "CASH\_CARD"."ID" AS "ID", "CASH\_CARD"."AMOUNT" AS "AMOUNT" FROM "CASH\_CARD"
```

La causa de los fallos de nuestras pruebas es clara: “Table CASH_CARD not found” significa que no tenemos una base de datos ni datos.

10.7 Configurar la base de datos

Nuestras pruebas esperan que la API encuentre y devuelva una `CashCard` con `id` de 99. Sin embargo, acabamos de eliminar los datos codificados de `CashCard` y los reemplazamos con una llamada a `cashCardRepository.findById`.

Ahora nuestra aplicación se está estrellando, quejándose de una tabla de base de datos faltante llamada `CASH_CARD`:

```
org.h2.jdbc.JdbcSQLException: Table "CASH\_CARD" not found (this database
```

Necesitamos ayudar a Spring Data a configurar la base de datos y cargar algunos datos de muestra, como nuestro amigo, `CashCard` 99.

Spring Data y H2 pueden crear y poblar automáticamente la base de datos en memoria que necesitamos para nuestras pruebas. Hemos proporcionado estos archivos para ti, pero necesitarás amendarlos: `schema.sql` y `data.sql`.

Nota: Proporcionar `schema.sql` y `data.sql` es una de las muchas formas en que Spring proporciona para inicializar fácilmente una base de datos. Para aprender más, lee la documentación del marco Spring.

Edita `schema.sql`.

Como se menciono anteriormente, Spring Data configurará automáticamente una base de datos para pruebas si proporcionamos el archivo correcto en la ubicación correcta.

¡Y lo tenemos! Es `src/main/resources/schema.sql`.

Pero, actualmente está deshabilitado.

Edita `src/main/resources/schema.sql` y elimina el comentario de bloque `/* ... */`.

```
1 CREATE TABLE cash_card
2 (
3     ID          BIGINT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
4     AMOUNT      NUMBER NOT NULL DEFAULT 0
5 );
```

Entiende `schema.sql`.

Un esquema de base de datos es un "plano" de cómo se almacenan los datos en una base de datos. @ No cubriremos los esquemas de base de datos en profundidad aquí.

Nuestro esquema de base de datos refleja el objeto `CashCard` que entendemos, que contiene un `id` y un `amount`.

Vuelve a ejecutar las pruebas.

Nota: Si la salida de la prueba es demasiado verbose, revierta el cambio en `build.gradle` realizado previamente.

Nuestras pruebas ya no se estrellan con un error `500 INTERNAL_SERVER_ERROR`. @ Sin embargo, ahora obtenemos un `404 NOT_FOUND`.

```
1 CashCardApplicationTests > shouldReturnACashCardWhenDataIsSaved() FAILED
2 org.opentest4j.AssertionFailedError:
3   expected: 200 OK
4   but was: 404 NOT_FOUND
```

Traducción: Nuestro repositorio no puede encontrar `CashCard` con `id` de 99. @ Entonces, ¿por qué no?

Aunque hemos ayudado a Spring Data a crear una base de datos de prueba al descomentar `schema.sql`, sigue siendo una base de datos vacía.

¡Vamos a cargar algunos datos!

Carga datos de prueba desde `data.sql`.

No solo puede Spring Data crear nuestra base de datos de prueba, también puede cargar datos en ella, que podemos usar en nuestras pruebas.

Similar a `schema.sql`, hemos proporcionado `src/test/resources/data.sql`, pero su contenido también está comentado.

Eliminemos los comentarios de bloque en `src/test/resources/data.sql`.

```
1 INSERT INTO CASH_CARD(ID, AMOUNT) VALUES (99, 123.45);
```

Esta declaración SQL inserta una fila en la tabla `CASH_CARD` con `ID=99` y `AMOUNT=123.45`, que coincide con los valores que esperamos en nuestras pruebas.

Vuelve a ejecutar las pruebas.

¡Pasan! ¡Hurra!

```
1 [~/exercises] $ ./gradlew test
2 ...
3 BUILD SUCCESSFUL in 7s
```

¡Éxito! Ahora estamos usando datos reales en nuestra API.

Momento de aprendizaje: recursos principales vs de prueba.

¿Has notado que `src/main/resources/schema.sql` y `src/test/resources/data.sql` están en diferentes ubicaciones de recursos?

¿Puedes adivinar por qué esto es así?

Recuerda que nuestra Cash Card con ID 99 y Amount 123.45 es una Cash Card falsa e inventada que solo queremos usar en nuestras pruebas. @ No queremos que nuestro sistema “real” o de producción cargue Cash Card 99 en el sistema... ¿que le pasaría a la Cash Card 99 real?

Spring nos ha proporcionado una característica poderosa: nos permite separar nuestros recursos solo de prueba de nuestros recursos principales cuando sea necesario.

Nuestro escenario aquí es un ejemplo común de esto: nuestro esquema de base de datos siempre es el mismo, ¡pero nuestros datos no lo son!

¡Gracias de nuevo, Spring!

10.8 Resumen

Ahora has refactorizado con éxito la forma en que la API de Family Cash Card gestiona sus datos. @ Spring Data ahora está creando una base de datos en memoria H2 y cargándola con datos de prueba, que nuestras pruebas utilizan para ejercitar nuestra API.

Además, ¡no cambiamos ninguna de nuestras pruebas! De hecho, nos guiaron hacia una implementación correcta. ¿Qué tan increíble es eso?!

11 Implementación de POST

Nuestra API REST ahora puede recuperar Cash Cards con un ID específico. En esta lección, añadirás el endpoint de Creación a la API.

Cuatro preguntas que necesitaremos responder mientras lo hacemos son:

- ¿Quién especifica el ID - el cliente o el servidor?
- En la solicitud de la API, ¿cómo representamos el objeto a crear?
- ¿Qué método HTTP deberíamos usar en la solicitud?
- ¿Qué envía la API como respuesta?

Comencemos respondiendo la primera pregunta: "¿Quién especifica el ID?". En realidad, esto depende del creador de la API. REST no es exactamente un estándar; es simplemente una forma de usar HTTP para realizar operaciones de datos. REST contiene una serie de pautas, muchas de las cuales estamos siguiendo en este curso.

Aquí elegiremos dejar que el servidor cree el ID. ¿Por qué? Porque es la solución más simple y las bases de datos son eficientes en la gestión de IDs únicos. Sin embargo, por completitud, discutamos nuestras alternativas:

- Podríamos requerir que el cliente proporcione el ID. Esto podría tener sentido si hubiera un ID único preexistente, pero ese no es el caso.
- Podríamos permitir que el cliente proporcione el ID opcionalmente (y crearlo en el servidor si el cliente no lo suministra). Sin embargo, no tenemos un requisito para hacer esto, y complicaría nuestra aplicación. Si piensas que podrías querer hacerlo "por si acaso", el artículo Yagni (enlace en la sección de Referencias) podría disuadirte.

Antes de responder la tercera pregunta, "¿Qué método HTTP deberíamos usar en la solicitud?", hablemos sobre el concepto relevante de idempotencia.

11.1 Idempotencia y HTTP

Una operación idempotente se define como una que, si se realiza más de una vez, resulta en el mismo resultado. En una API REST, una operación idempotente es una que, incluso si se realizara varias veces, los datos resultantes en el servidor serían los mismos que si se hubiera realizado solo una vez.

Para cada método, el estándar HTTP especifica si es idempotente o no. GET, PUT y DELETE son idempotentes, mientras que POST y PATCH no lo son.

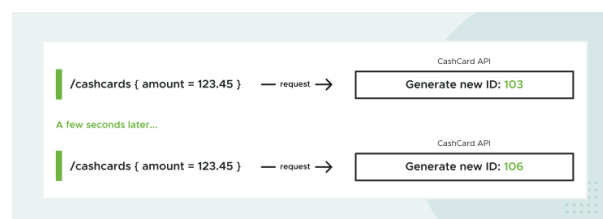


Figura 8: Idempotencia en la API

Esto nos deja con las opciones POST y PATCH.® Resulta que REST permite POST como uno de los métodos adecuados para usar en operaciones de Creación, así que lo usaremos.® Revisaremos PATCH en una lección posterior.

11.2 La solicitud POST y la respuesta

Ahora hablemos sobre el contenido de la solicitud POST y la respuesta.

11.2.1. La solicitud

El método POST permite un Cuerpo, así que usaremos el Cuerpo para enviar una representación JSON del objeto:

```
1 Metodo: POST
2 URI: /cashcards/
3 Cuerpo:
4 {
5     "amount": 123.45
6 }
```

En contraste, si recuerdas de una lección anterior, la operación GET incluye el ID de la Cash Card en el URI, pero no en el Cuerpo de la solicitud.

Entonces, ¿por qué no hay ID en la solicitud? Porque decidimos permitir que el servidor cree el ID.® Así, el contrato de datos para la operación de Lectura es diferente del de la operación de Creación.

11.2.2. La respuesta

Pasemos a la respuesta.® En caso de creación exitosa, ¿qué código de estado HTTP de respuesta deberíamos enviar? Podríamos usar 200 OK (la respuesta que devuelve Lectura), pero hay un código más específico y preciso para APIs REST: 201 CREATED.

El hecho de que CREATED sea el nombre del código lo hace parecer intuitivamente apropiado, pero hay otra razón más técnica para usarlo: Un código de respuesta de 200 OK no responde a la pregunta "¿Hubo algún cambio en los datos del servidor?".® Al devolver el estado 201 CREATED, la API está comunicando específicamente que se añadieron datos al almacén de datos en el servidor.

En una lección anterior aprendiste que una respuesta HTTP contiene dos cosas: un Código de Estado y un Cuerpo. ¡Pero eso no es todo! Una respuesta también contiene Cabeceras.® Las cabeceras tienen un nombre y un valor.® El estándar HTTP especifica que la cabecera Location en una respuesta 201 CREATED debe contener el URI del recurso creado.® Esto es útil porque permite al llamador recuperar fácilmente el nuevo recurso usando el endpoint GET (el que implementamos anteriormente).

Aquí está la respuesta completa:

Respuesta:

```
1 Código de Estado: 201 CREATED
2 Cabecera: Location=/cashcards/42
```

11.3 Métodos de conveniencia de Spring Web

En el laboratorio que acompaña, verás que Spring Web proporciona métodos diseñados para el uso recomendado de HTTP y REST.

Por ejemplo, usaremos el método `ResponseEntity.created(uriOfCashCard)` para crear la respuesta anterior. Este método requiere que especifiques la ubicación, asegura que el URI de Location esté bien formado (usando la clase `URI`), añade la cabecera Location y establece el Código de Estado por ti. Y al hacerlo, esto nos ahorra usar métodos más verbose. Por ejemplo, los siguientes dos fragmentos de código son equivalentes (siempre que `uriOfCashCard` no sea nulo):

```
1 return ResponseEntity
2     .created(uriOfCashCard)
3     .build();
```

Versus:

```
1 return ResponseEntity
2     .status(HttpStatus.CREATED)
3     .header(HttpHeaders.LOCATION, uriOfCashCard.toASCIIString())
4     .build();
```

¿No estás contento de que Spring Web proporcione el método de conveniencia `.created()`?

12 Laboratorio: Implementación de POST

12.1 Prueba del endpoint HTTP POST

Como hemos hecho en laboratorios anteriores, comenzaremos escribiendo una prueba de lo que esperamos que el éxito se vea.

Añade una prueba para el endpoint POST.

El ejemplo más simple de éxito es una solicitud HTTP POST no fallida a nuestra API de Family Cash Card.@ Probaremos con una respuesta 200 OK en lugar de un 201 CREATED por ahora.@ No te preocupes, lo cambiaremos pronto.

Edita `src/test/java/example/cashcard/CashCardApplicationTests.java` y añade el siguiente método de prueba.

```
1 @Test
2 void shouldCreateANewCashCard() {
3     CashCard newCashCard = new CashCard(null, 250.00);
4     ResponseEntity<Void> createResponse = restTemplate.postForEntity("/cashcards",
5         newCashCard, Void.class);
6     assertThat(createResponse.getStatusCode()).isEqualTo(HttpStatus.OK);
7 }
```

Entiende la prueba.

- `CashCard newCashCard = new CashCard(null, 250.00);`: La base de datos creará y gestionará todos los valores únicos de `CashCard.id` para nosotros.@ No deberíamos proporcionar uno.
- `restTemplate.postForEntity("/cashcards", newCashCard, Void.class);`: Esto es muy similar a `restTemplate.getForEntity`, pero también debemos proporcionar los datos `newCashCard` para la nueva Cash Card.
- Además, y a diferencia de `restTemplate.getForEntity`, no esperamos que se nos devuelva una `CashCard`, por lo que esperamos un cuerpo de respuesta `Void`.

Ejecuta las pruebas.

Siempre usaremos `./gradlew test` para ejecutar nuestras pruebas.

```
1 [~/exercises] $ ./gradlew test
```

¿Qué esperas que pase?

```
1 CashCardApplicationTests > shouldCreateANewCashCard() FAILED
2 org.opentest4j.AssertionFailedError:
3 expected: 200 OK
4 but was: 404 NOT_FOUND
```

No deberíamos sorprendernos por el error 404 NOT_FOUND. ¡Aún no hemos añadido el endpoint POST!

Hagámoslo a continuación.

12.2 Añade el endpoint POST

El endpoint POST es similar al endpoint GET en nuestro `CashCardController`, pero usa la anotación `@PostMapping` de Spring Web.

El endpoint POST debe aceptar los datos que estamos enviando para nuestra nueva `CashCard`, específicamente el `amount`.

Pero, ¿qué pasa si no aceptamos la `CashCard`?

Añade el endpoint POST sin aceptar datos de `CashCard`.

Edita `src/main/java/example/cashcard/CashCardController.java` y añade el siguiente método.

No olvides añadir la importación para `PostMapping`.

```
1 import org.springframework.web.bind.annotation.PostMapping;
2 ...
3
4 @PostMapping
5 private ResponseEntity<Void> createCashCard() {
6     return null;
7 }
```

Nota que al no devolver nada en absoluto, Spring Web generará automáticamente un código de estado de respuesta HTTP de 200 OK.

Ejecuta las pruebas.

Al volver a ejecutar las pruebas, estas pasan.

```
1 BUILD SUCCESSFUL in 7s
```

Pero, ¡esto no es muy satisfactorio! Nuestro endpoint POST no hace nada.

Así que hagamos nuestras pruebas mejores.

12.3 Pruebas basadas en corrección semántica

Queremos que nuestra API de Cash Card se comporte lo más semánticamente correcto posible. Esto significa que los usuarios de nuestra API no deberían sorprenderse por cómo se comporta.

Hagamos referencia a la solicitud oficial de comentarios para Semántica y Contenido de HTTP (RFC 9110) para obtener orientación sobre cómo debería comportarse nuestra API.

Para nuestro endpoint POST, revisa esta sección sobre HTTP POST; nota que hemos añadido énfasis:

*If one or more resources has been created on the origin server as a result of successfully processing a POST request, the origin server **SHOULD** send a 201 (Created) response containing a Location header field that provides an identifier for the primary resource created ...*

Explicaremos más sobre esta especificación mientras escribimos nuestra prueba.

Comencemos actualizando la prueba POST.

Actualiza la prueba `shouldCreateANewCashCard`.

Aquí está cómo codificaremos la especificación HTTP como expectativas en nuestra prueba. @ Asegúrate de añadir la importación adicional.

```

1 import java.net.URI;
2 ...
3
4 @Test
5 void shouldCreateANewCashCard() {
6     CashCard newCashCard = new CashCard(null, 250.00);
7     ResponseEntity<Void> createResponse = restTemplate.postForEntity("/cashcards",
8         newCashCard, Void.class);
9     assertThat(createResponse.getStatusCode()).isEqualTo(HttpStatus.CREATED);
10
11     URI locationOfNewCashCard = createResponse.getHeaders().getLocation();
12     ResponseEntity<String> getResponse =
13         restTemplate.getForEntity(locationOfNewCashCard, String.class);
14     assertThat(getResponse.getStatusCode()).isEqualTo(HttpStatus.OK);
15 }

```

Entiende las actualizaciones de la prueba.

Hemos hecho bastantes cambios. @ Revisémoslos.

- `assertThat(createResponse.getStatusCode()).isEqualTo(HttpStatus.CREATED);` : Según la especificación oficial: *the origin server SHOULD send a 201 (Created) response ...* Ahora esperamos que el código de estado HTTP de respuesta sea 201 CREATED, lo cual es semánticamente correcto si nuestra API crea una nueva `CashCard` a partir de nuestra solicitud.
- `URI locationOfNewCashCard = createResponse.getHeaders().getLocation();` : La especificación oficial continúa diciendo: *send a 201 (Created) response containing a Location header field that provides an identifier for the primary resource created ...* En otras palabras, cuando una solicitud POST resulta en la creación exitosa de un recurso, como una nueva `CashCard`, la respuesta debería incluir información sobre cómo recuperar ese recurso. @ Lo haremos suministrando un URI en una cabecera de respuesta llamada “Location”.
- Nota que URI es de hecho la entidad correcta aquí y no una URL; una URL es un tipo de URI, mientras que un URI es más genérico.
- `ResponseEntity<String>getResponse = restTemplate.getForEntity(locationOfNewCashCard, String.class);` `assertThat(getResponse.getStatusCode()).isEqualTo(HttpStatus.OK);` : Finalmente, usaremos la información de la cabecera Location para recuperar la `CashCard` recién creada.

Ejecuta las pruebas.

Sin sorpresa, fallan en la primera afirmación cambiada.

```

expected: 201 CREATED
but was: 200 OK

```

¡Comencemos a arreglar cosas!

12.4 Implementa el endpoint POST

Nuestro endpoint POST en `CashCardController` está actualmente vacío. @ Implementemos la lógica correcta.

Devuelve un estado 201 CREATED.

A medida que hagamos que nuestra prueba pase incrementalmente, podemos comenzar devolviendo 201 CREATED.

Como aprendimos anteriormente, debemos proporcionar una cabecera Location con el URI para donde encontrar la `CashCard` recién creada. @ Aún no estamos ahí, así que usaremos un URI placeholder por ahora.

Asegúrate de añadir las dos nuevas declaraciones de importación.

```
1 import java.net.URI;
2 import org.springframework.web.bind.annotation.RequestBody;
3 ...
4
5 @PostMapping
6 private ResponseEntity<Void> createCashCard(@RequestBody CashCard newCashCardRequest)
7 {
8     return ResponseEntity.created(URI.create("/what/should/go/here?")).build();
9 }
```

Ejecuta las pruebas.

```
expected: 200 OK
but was: 404 NOT_FOUND
```

Notablemente, nuestra nueva prueba pasa hasta la última línea de la nueva prueba.

```
...
assertThat(getResponse.getStatusCode()).isEqualTo(HttpStatus.OK);
```

Aquí esperamos haber recuperado nuestra `CashCard` recién creada, que no hemos creado ni devuelto desde nuestro `CashCardController`. @ Así, nuestra expectativa falla con un resultado de NOT_FOUND.

Guarda la nueva `CashCard` y devuelve su ubicación.

Añadamos el resto de la implementación POST, que describiremos en detalle.

Asegúrate de añadir la nueva importación.

```
1 import org.springframework.web.util.UriComponentsBuilder;
2 ...
3
4 @PostMapping
5 private ResponseEntity<Void> createCashCard(@RequestBody CashCard newCashCardRequest,
6     UriComponentsBuilder ucb) {
7     CashCard savedCashCard = cashCardRepository.save(newCashCardRequest);
8     URI locationOfNewCashCard = ucb
9         .path("cashcards/{id}")
```

```

9         .buildAndExpand(savedCashCard.id())
10        .toUri();
11    return ResponseEntity.created(locationOfNewCashCard).build();
12 }

```

A continuacion, revisaremos estos cambios en detalle.

12.5 Entiende CrudRepository.save

Esta línea en `CashCardController.createCashCard` es engañosamente simple:

```

1 CashCard savedCashCard = cashCardRepository.save(newCashCardRequest);

```

Como aprendimos en lecciones y laboratorios anteriores, el `CrudRepository` de Spring Data proporciona métodos que soportan crear, leer, actualizar y eliminar datos de un almacén de datos. `cashCardRepository.save(newCashCardRequest)` hace exactamente lo que dice: guarda una nueva `CashCard` para nosotros y devuelve el objeto guardado con un ID único proporcionado por la base de datos. ¡Asombroso!

12.6 Entiende los otros cambios en CashCardController

Nuestro `CashCardController` ahora implementa las entradas y resultados esperados de un HTTP POST.

- `createCashCard(@RequestBody CashCard newCashCardRequest, ...)`: A diferencia del GET que añadimos anteriormente, el POST espera un “cuerpo” de solicitud. @ Esto contiene los datos enviados a la API. @ Spring Web deserializará los datos en una `CashCard` para nosotros.
- `URI locationOfNewCashCard = ucb.path(“cashcards/{id}”).buildAndExpand(savedCashCard.id)` Esto está construyendo un URI a la `CashCard` recién creada. @ Este es el URI que el llamador puede luego usar para GET la `CashCard` recién creada.
- Nota que `savedCashCard.id` se usa como el identificador, lo que coincide con la especificación del endpoint GET de `cashcards/{CashCard.id}`.
- ¿De dónde vino `UriComponentsBuilder`? Pudimos añadir `UriComponentsBuilder ucb` como argumento del método en este manejador POST y se pasó automáticamente. ¿Cómo? Fue inyectado desde nuestro ahora familiar amigo, el Contenedor IoC de Spring. ¡Gracias, Spring Web!
- `return ResponseEntity.created(locationOfNewCashCard).build();`: Finalmente, devolvemos 201 CREATED con la cabecera Location correcta.

12.7 Pruebas finales y momento de aprendizaje

Ejecuta las pruebas.

¡Pasan!

```

1 BUILD SUCCESSFUL in 7s

```


La nueva `CashCard` fue creada, y usamos el URI suministrado en la cabecera de respuesta `Location` para recuperar el recurso recién creado.

Añade más afirmaciones de prueba.

Si lo deseas, añade más afirmaciones de prueba para el nuevo `id` y `amount` para solidificar tu aprendizaje.

```

1 ...
2 assertThat(getResponse.getStatusCode()).isEqualTo(HttpStatus.OK);
3
4 // Añade afirmaciones como estas
5 DocumentContext documentContext = JsonPath.parse(getResponse.getBody());
6 Number id = documentContext.read("$.id");
7 Double amount = documentContext.read("$.amount");
8
9 assertThat(id).isNotNull();
10 assertThat(amount).isEqualTo(250.00);

```

Las adiciones verifican que el nuevo `CashCard.id` no sea nulo y que el `CashCard.amount` recién creado sea 250.00, tal como especificamos en el momento de la creación.

Momento de aprendizaje.

Anteriormente declaramos que la base de datos (vía el Repositorio) gestionaría la creación de todos los valores de ID de la base de datos para nosotros.

¿Qué pasaría si proporcionáramos un `id` para nuestra nueva `CashCard` no guardada?

Descubrámoslo.

Actualiza la prueba para enviar un `CashCard.id`.

Cambia el `id` enviado de `null` a uno que no existe, como 44L.

```

1 @Test
2 void shouldCreateANewCashCard() {
3     CashCard newCashCard = new CashCard(44L, 250.00);
4     ...

```

Además, edita `build.gradle` para habilitar una salida de prueba más verbose, lo que nos ayudará a identificar el próximo fallo de prueba.

```

1 test {
2     testLogging {
3         ...
4         // Establece en true para un registro mas detallado.
5         showStandardStreams = true
6     }
7 }

```

Ejecuta las pruebas.

Al ejecutar la prueba, vemos que la API se estrella con un código de estado 500.

```

1 [~/exercises] $ ./gradlew test
2 ...
3 expected: 201 CREATED
4 but was: 500 INTERNAL\_SERVER\_ERROR

```

Descubramos por qué falla la prueba.

Encuentra y entiende el fallo de la base de datos.

Busca en la salida de la prueba el siguiente mensaje:

```
1 Failed to update entity [CashCard[id=44, amount=250.0]].@ Id [44] not found in
   database.
```

El Repositorio está intentando encontrar una **CashCard** con **id** de 44 y lanza un error cuando no la encuentra. ¡Interesante! ¿Puedes adivinar por qué?

Suministrar un **id** a **cashCardRepository.save** se soporta cuando se realiza una actualización en un recurso existente.

Cubriremos este escenario en un laboratorio posterior enfocado en actualizar una **CashCard** existente.

En este Momento de Aprendizaje aprendiste que la API requiere que no suministres un **CashCard.id** al crear una nueva **CashCard**.

¿Deberíamos validar ese requisito en la API? ¡Claro que sí! De nuevo, quédate atento para cómo hacerlo en una lección futura.

12.8 Resumen

En este laboratorio aprendiste lo simple que es añadir otro endpoint a nuestra API: el endpoint **POST**.@ También aprendiste cómo usar ese endpoint para crear y guardar una nueva **CashCard** en nuestra base de datos usando **Spring Data**.@ No solo eso, sino que el endpoint implementa con precisión la especificación **HTTP POST**, lo que verificamos usando desarrollo dirigido por pruebas. ¡La API está comenzando a ser útil!

13 Devolución de una lista con GET

Ahora que nuestra API puede crear Cash Cards, es razonable aprender cómo recuperar todas (¡o algunas!) de las Cash Cards. En esta lección, implementaremos el endpoint “Leer Múltiples” y entenderemos cómo esta operación difiere sustancialmente del endpoint de Lectura que creamos previamente.

13.1 Solicitando una lista de Cash Cards

Podemos esperar que cada uno de nuestros usuarios de Family Cash Card tenga algunas tarjetas: imagina una para cada miembro de su familia y quizás algunas que dieron como regalos. La API debería poder devolver múltiples Cash Cards en respuesta a una sola solicitud REST.

Cuando hagas una solicitud de API para varias Cash Cards, idealmente harías una sola solicitud, que devuelva una lista de Cash Cards. Así que, necesitaremos un nuevo contrato de datos. En lugar de una sola Cash Card, el nuevo contrato debería especificar que la respuesta es un Array JSON de objetos Cash Card:

```
1 [
2   {
3     "id": 1,
4     "amount": 123.45
5   },
6   {
7     "id": 2,
8     "amount": 50.0
9   }
10 ]
```

Resulta que nuestro viejo amigo, `CrudRepository`, tiene un método `findAll` que podemos usar para recuperar fácilmente todas las Cash Cards en la base de datos. ¡Adelante, usemos ese método! A primera vista, parece bastante simple:

```
1 @GetMapping()
2 private ResponseEntity<Iterable<CashCard>> findAll() {
3     return ResponseEntity.ok(cashCardRepository.findAll());
4 }
```

Sin embargo, resulta que hay mucho más en esta operación que simplemente devolver todas las Cash Cards en la base de datos. Algunas preguntas vienen a la mente:

- ¿Cómo devuelvo solo las Cash Cards que posee el usuario? (¡Gran pregunta! Lo discutiremos en la próxima lección de Spring Security).
- ¿Qué pasa si hay cientos (¡o miles!?) de Cash Cards? ¿Debería la API devolver un número ilimitado de resultados o devolverlos en "fragmentos"? Esto se conoce como Paginación.
- ¿Deberían las Cash Cards devolverse en un orden particular (es decir, ¿deberían estar ordenadas?)?

Dejaremos la primera pregunta para más tarde, pero responderemos las preguntas de paginación y ordenación en esta lección. ¡Sigamos adelante!

13.2 Paginación y ordenación

Para comenzar nuestro trabajo de paginación y ordenación, usaremos una versión especializada de `CrudRepository`, llamada `PagingAndSortingRepository`.@ Como podrías adivinar, esto hace exactamente lo que su nombre sugiere.@ Pero primero, hablemos sobre la funcionalidad de “Paginación”.

Aunque es poco probable que tengamos usuarios con miles de Cash Cards, nunca sabemos cómo podrían usar el producto los usuarios.@ Idealmente, una API no debería poder producir una respuesta de tamaño ilimitado, porque esto podría abrumar la memoria del cliente o del servidor, sin mencionar que tomaría bastante tiempo.

Para asegurar que una respuesta de API no incluya un número astronómicamente grande de Cash Cards, usemos la funcionalidad de paginación de Spring Data.@ La paginación en Spring (y muchos otros marcos) consiste en especificar la longitud de la página (por ejemplo, 10 ítems) y el índice de la página (comenzando con 0).@ Por ejemplo, si un usuario tiene 25 Cash Cards y eliges solicitar la segunda página donde cada página tiene 10 ítems, solicitarías una página de tamaño 10 e índice de página 1.

¡Bingo! ¿Verdad? Pero espera, esto plantea otro obstáculo.@ Para que la paginación produzca el contenido correcto de la página, los ítems deben estar ordenados en algún orden específico. ¿Por qué? Bueno, digamos que tenemos un montón de Cash Cards con las siguientes cantidades:

- \$0.19 (¡esta casi se ha ido, oh bueno!)
- \$1,000.00 (esta es para compras de emergencia para un estudiante universitario)
- \$50.00
- \$20.00
- \$10.00 (alguien regaló esta a tu sobrina por su cumpleaños)

Ahora pasemos por un ejemplo usando un tamaño de página de 3.@ Dado que hay 5 Cash Cards, haríamos dos solicitudes para devolverlas todas.@ La página 1 (índice 0) contiene tres ítems, y la página 2 (índice 1, la última página) contiene 2 ítems.@ Pero, ¿qué ítems van dónde? Si especificas que los ítems deben ordenarse por cantidad en orden descendente, entonces así se paginan los datos:

- Página 1:
 - \$1,000.00
 - \$50.00
 - \$20.00
- Página 2:
 - \$10.00

- \$0.19

13.2.1. Consultas sin ordenar

Aunque Spring proporciona una estrategia de ordenación “sin ordenar”, seamos explícitos cuando seleccionemos qué campos para ordenar. ¿Por qué hacer esto? Bueno, imagina que eliges usar paginación “sin ordenar”.@ En realidad, el orden no es aleatorio, sino predecible; nunca cambia en solicitudes subsiguientes.@ Digamos que haces una solicitud y Spring devuelve los siguientes resultados “sin ordenar”:

- Página 1:

- \$0.19
- \$1,000.00
- \$50.00

- Página 2:

- \$20.00
- \$10.00

Aunque parecen aleatorios, cada vez que hagas la solicitud, las tarjetas volverán en exactamente este orden, para que cada ítem se devuelva en exactamente una página.

Ahora para el remate: Imagina que ahora creas una nueva Cash Card con una cantidad de \$42.00. ¿En qué página crees que estará? Como podrías adivinar, no hay manera de saberlo más que haciendo la solicitud y viendo dónde cae la nueva Cash Card.

Entonces, ¿cómo podemos hacer esto un poco más útil? Optemos por ordenar por un campo específico.@ Hay algunas buenas razones para hacerlo, incluyendo:

- Minimizar la sobrecarga cognitiva: Otros desarrolladores (sin mencionar a los usuarios) probablemente apreciarán un ordenamiento bien pensado al desarrollarlo.
- Minimizar errores futuros: ¿Qué pasa cuando una nueva versión de Spring, o Java, o la base de datos, de repente causa que el orden “aleatorio” cambie overnight?

13.2.2. API de paginación de Spring Data

Afortunadamente, Spring Data proporciona las clases `PageRequest` y `Sort` para paginación.@ Veamos una consulta para obtener la página 2 con un tamaño de página 10, ordenando por cantidad en orden descendente (cantidades más grandes primero):

```
1 Page<CashCard> page2 = cashCardRepository.findAll(  
2     PageRequest.of(  
3         1, // indice de pagina para la segunda pagina - la indexacion comienza en 0  
4         10, // tamaño de pagina (la ultima pagina podria tener menos items)  
5         Sort.by(new Sort.Order(Sort.Direction.DESC, "amount"))));
```

13.3 La solicitud y la respuesta

Ahora usemos Spring Web para extraer los datos para alimentar la funcionalidad de paginación:

- **Paginación:** Spring puede analizar los parámetros de página y tamaño si pasas un objeto `Pageable` a un método `find...`() de `PagingAndSortingRepository`.
- **Ordenación:** Spring puede analizar el parámetro `sort`, que consta del nombre del campo y la dirección separados por una coma – ¡pero cuidado, no se permite espacio antes o después de la coma! De nuevo, estos datos forman parte del objeto `Pageable`.

13.3.1. El URI

Ahora aprendamos cómo podemos componer un URI para el nuevo endpoint, paso a paso (hemos omitido el prefijo `https://domain` en lo siguiente):

Obtener la segunda página:

```
1 /cashcards?page=1
```

...donde una página tiene una longitud de 3:

```
1 /cashcards?page=1&size=3
```

...ordenado por el saldo actual de la Cash Card:

```
1 /cashcards?page=1&size=3&sort=amount
```

...en orden descendente (saldo más alto primero):

```
1 /cashcards?page=1&size=3&sort=amount,desc
```

13.3.2. El código Java

Revisemos la implementación completa del método del Controlador para nuestro nuevo endpoint “obtener una página de Cash Cards”:

```
1 @GetMapping
2 private ResponseEntity<List<CashCard>> findAll(Pageable pageable) {
3     Page<CashCard> page = cashCardRepository.findAll(
4         PageRequest.of(
5             pageable.getPageNumber(),
6             pageable.getPageSize(),
7             pageable.getSortOr(Sort.by(Sort.Direction.DESC, "amount"))));
8     return ResponseEntity.ok(page.getContent());
9 }
```

Busquemos un poco mas de detalle:

- Primero, analicemos los valores necesarios de la cadena de consulta:
 - Usamos `Pageable`, que permite a Spring analizar los parametros de la cadena de consulta de número de página y tamaño.

- Nota: Si el llamador no proporciona los parametros, Spring proporciona valores predeterminados: `page=0`, `size=20`.
- Usamos `getSortOr()` para que, incluso si el llamador no suministra el parametro `sort`, haya un valor predeterminado. A diferencia de los parametros `page` y `size`, para los cuales tiene sentido que Spring suministre un valor predeterminado, no tendria sentido que Spring eligiera arbitrariamente un campo y direccion de ordenacion.
- Usamos el método `page.getContent()` para devolver las Cash Cards contenidas en el objeto `Page` al llamador.
- Entonces, que contiene el objeto `Page` ademas de las Cash Cards? Aqui esta el objeto `Page` en formato JSON. @ Las Cash Cards estan contenidas en el `content`. @ Los demas campos contienen información sobre como esta `Page` se relaciona con otras `Pages` en la consulta.

```
1 {
2   "content": [
3     {
4       "id": 1,
5       "amount": 10.0
6     },
7     {
8       "id": 2,
9       "amount": 0.19
10    }
11  ],
12  "pageable": {
13    "sort": {
14      "empty": false,
15      "sorted": true,
16      "unsorted": false
17    },
18    "offset": 3,
19    "pageNumber": 1,
20    "pageSize": 3,
21    "paged": true,
22    "unpaged": false
23  },
24  "last": true,
25  "totalElements": 5,
26  "totalPages": 2,
27  "first": false,
28  "size": 3,
29  "number": 1,
30  "sort": {
31    "empty": false,
32    "sorted": true,
33    "unsorted": false
34  },
35  "numberOfElements": 2,
36  "empty": false
37 }
```

Aunque podriamos devolver el objeto `Page` completo al cliente, no necesitamos toda esa información. @ Definiremos nuestro contrato de datos para devolver solo las Cash Cards,

no el resto de los datos de `Page`.

14 Laboratorio: Devolución de una lista con GET

14.1 Cambios desde el laboratorio anterior

Hemos realizado los siguientes cambios desde el laboratorio anterior.

- Añadimos algunos datos ficticios más de Cash Card a `src/test/resources/data.sql`.
- Refactorizamos `CashCardJsonTest.java` para incorporar los nuevos datos ficticios.
- Renombramos `expected.json` a `single.json` y añadimos otro archivo JSON de contrato de datos: `list.json`.
- Añadimos algunas importaciones a las clases de Prueba, ¡para que no tengas que hacerlo!
- Añadimos la anotación `@DirtyContext` a la clase `CashCardApplicationTests`.

Esta lista es solo un resumen. @ Ampliaremos cada punto a lo largo de las instrucciones del laboratorio.

Los nuevos datos ficticios añadidos a `src/test/resources/data.sql` son:

```
1 INSERT INTO CASH_CARD(ID, AMOUNT) VALUES (99, 123.45);
2 INSERT INTO CASH_CARD(ID, AMOUNT) VALUES (100, 1.00);
3 INSERT INTO CASH_CARD(ID, AMOUNT) VALUES (101, 150.00);
```

14.2 Prueba del nuevo contrato de datos

Como hemos hecho en talleres anteriores, comenzaremos escribiendo una prueba de lo que esperamos que el éxito se vea.

Dado que estamos introduciendo un nuevo contrato de datos, ¡comencemos probándolo!

Revisa los nuevos datos ficticios.

Mira el archivo `src/test/resources/example/cashcard/list.json`. @ Contiene el siguiente array JSON:

```
1 [
2   {"id": 99, "amount": 123.45 },
3   {"id": 100, "amount": 1.00 },
4   {"id": 101, "amount": 150.00 }
5 ]
```

Este es nuestro nuevo contrato de datos que contiene una lista de Cash Cards, que coincide con los datos en el nuevo archivo `data.sql`; adelante, mira el archivo `src/test/resources/data.sql` para verificar que los valores del archivo JSON coincidan.

Ahora abre el archivo `CashCardJsonTest.java`. @ El código completo es:

```
1 package example.cashcard;
2
3 import org.assertj.core.util.Arrays;
4 import org.junit.jupiter.api.BeforeEach;
```

```

5 import org.junit.jupiter.api.Test;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.boot.test.autoconfigure.json.JsonTest;
8 import org.springframework.boot.test.json.JacksonTester;
9
10 import java.io.IOException;
11
12 import static org.assertj.core.api.Assertions.assertThat;
13
14 @JsonTest
15 class CashCardJsonTest {
16
17     @Autowired
18     private JacksonTester<CashCard> json;
19
20     @Autowired
21     private JacksonTester<CashCard[]> jsonList;
22
23     private CashCard[] cashCards;
24
25     @BeforeEach
26     void setUp() {
27         cashCards = Arrays.array(
28             new CashCard(99L, 123.45),
29             new CashCard(100L, 100.00),
30             new CashCard(101L, 150.00));
31     }
32
33     @Test
34     void cashCardSerializationTest() throws IOException {
35         CashCard cashCard = cashCards[0];
36         assertThat(json.write(cashCard)).isEqualToJson("single.json");
37         assertThat(json.write(cashCard)).hasJsonPathNumberValue("@.id");
38         assertThat(json.write(cashCard).extractingJsonPathNumberValue("@.id")
39             .isEqualTo(99);
40         assertThat(json.write(cashCard)).hasJsonPathNumberValue("@.amount");
41         assertThat(json.write(cashCard).extractingJsonPathNumberValue("@.amount")
42             .isEqualTo(123.45);
43     }
44
45     @Test
46     void cashCardDeserializationTest() throws IOException {
47         String expected = """
48             {
49                 "id": 99,
50                 "amount": 123.45
51             }
52             """;
53         assertThat(json.parse(expected))
54             .isEqualTo(new CashCard(99L, 123.45));
55         assertThat(json.parseObject(expected).id()).isEqualTo(99);
56         assertThat(json.parseObject(expected).amount()).isEqualTo(123.45);
57     }
58 }

```

Nota que la variable de nivel de clase `cashCards` está configurada para contener el siguiente array Java:

```
1 cashCards = Arrays.array(  
2     new CashCard(99L, 123.45),  
3     new CashCard(100L, 100.00),  
4     new CashCard(101L, 150.00));
```

Si miras de cerca, verás que uno de los objetos `CashCard` en nuestra prueba no coincide con los datos de prueba en `data.sql`. ¡Esto es para prepararnos para escribir una prueba que falle!

Añade una prueba de serialización para la lista de Cash Card.

Añade una nueva prueba a `CashCardJsonTest.java`:

```
1 @Test  
2 void cashCardListSerializationTest() throws IOException {  
3     assertThat(jsonList.write(cashCards)).isStrictlyEqualToJson("list.json");  
4 }
```

El código de la prueba es autoexplicativo: Serializa la variable `cashCards` en JSON y luego afirma que `list.json` debería contener los mismos datos que la variable `cashCards` serializada.

Ejecuta las pruebas.

¿Puedes predecir si la prueba fallará y, si lo hace, cuál será la causa del fallo? ¡Adelante, haz la predicción! ¿Qué crees que pasará?

Verifica tu predicción ejecutando las pruebas.

Nota que siempre ejecutaremos `./gradlew test` para ejecutar las pruebas.

```
1 [~/exercises] $ ./gradlew test  
2 ...  
3 > Task :test FAILED  
4 ...  
5 java.lang.AssertionError: JSON Comparison failure: [1].amount  
6 Expected: 1.0  
7     got: 100.0
```

¡Tu predicción fue correcta (¡esperemos!)? La prueba falló. @ Felizmente, el mensaje de error señala el lugar exacto donde ocurre el fallo: el campo `amount` de la segunda `CashCard` en el array (índice [1]) no es lo que se esperaba.

Corrige y vuelve a ejecutar las pruebas.

Cambia el `cashCards[1].amount` para esperar la cantidad correcta de la Cash Card.

Para verificar qué cantidad debería esperar la prueba, mira nuevamente el archivo `list.json`. @ Verás que la cantidad esperada debería ser 1.00 en lugar de 100.0. @ Así que cambiemos el código de la prueba:

```
1 ...  
2 new CashCard(100L, 1.00),  
3 ...
```

Después de corregir el código de la prueba para esperar 1.0 en lugar de 100.0 para el valor del campo `amount`, vuelve a ejecutar las pruebas. @ Pasarán:

```

1 [~/exercises] $ ./gradlew test
2 ...
3 BUILD SUCCESSFUL in 7s

```

Añade una prueba de deserialización.

Ahora probemos la deserialización. Añade la siguiente prueba:

```

1 @Test
2 void cashCardListDeserializationTest() throws IOException {
3     String expected=""
4     [
5         { "id": 99, "amount": 123.45 },
6         { "id": 100, "amount": 100.00 },
7         { "id": 101, "amount": 150.00 }
8     ]
9     """;
10    assertThat(jsonList.parse(expected)).isEqualTo(cashCards);
11 }

```

De nuevo, hemos afirmado intencionalmente un valor incorrecto para que sea obvio qué está probando la prueba.

Ejecuta las pruebas.

Cuando ejecutes las pruebas, verás que se detectó el valor incorrecto.

```

1 [~/exercises] $ ./gradlew test
2 ...
3 > Task :test FAILED
4 ...
5 expected:
6     [CashCard[id=99, amount=123.45],
7       CashCard[id=100, amount=1.0],
8       CashCard[id=101, amount=150.0]]
9 but was:
10    [CashCard[id=99, amount=123.45],
11     CashCard[id=100, amount=100.0],
12     CashCard[id=101, amount=150.0]]

```

Esta vez, la prueba falló porque deserializamos la cadena JSON esperada y la comparamos con la variable `cashCards`.@ De nuevo, esa molesta Cash Card de \$100.00 no coincide con la expectativa.

Cambia la expectativa:

```

1 String expected=""
2 [
3     { "id": 99, "amount": 123.45 },
4     { "id": 100, "amount": 1.00 },
5     { "id": 101, "amount": 150.00 }
6 ]
7 """;

```

A continuación, vuelve a ejecutar las pruebas y observa que la prueba pasa:

```

1 [~/exercises] $ ./gradlew test
2 ...
3 CashCardJsonTest > cashCardListDeserializationTest() PASSED

```

Ahora que hemos probado el contrato de datos, pasemos al endpoint del Controlador.

14.3 Prueba para un endpoint GET adicional

Escribe una prueba que falle para un nuevo endpoint GET.

Añadamos un nuevo método de prueba que espere un endpoint GET que devuelva múltiples objetos CashCard.

En `CashCardApplicationTests.java`, añade una nueva prueba. El código completo de la clase es:

```
1 package example.cashcard;
2
3 import com.jayway.jsonpath.DocumentContext;
4 import com.jayway.jsonpath.JsonPath;
5 import net.minidev.json.JSONArray;
6 import org.junit.jupiter.api.Test;
7 import org.springframework.beans.factory.annotation.Autowired;
8 import org.springframework.boot.test.context.SpringBootTest;
9 import org.springframework.boot.test.web.client.TestRestTemplate;
10 import org.springframework.http.HttpStatus;
11 import org.springframework.http.ResponseEntity;
12 import org.springframework.test.annotation.DirtiesContext;
13
14 import java.net.URI;
15
16 import static org.assertj.core.api.Assertions.assertThat;
17 import static org.springframework.test.annotation.DirtiesContext.*;
18
19 @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
20 @DirtiesContext(classMode = ClassMode.AFTER_EACH_TEST_METHOD)
21 class CashCardApplicationTests {
22     @Autowired
23     TestRestTemplate restTemplate;
24
25     @Test
26     void shouldReturnACashCardWhenDataIsSaved() {
27         ResponseEntity<String> response = restTemplate.getForEntity("/cashcards/99",
28             String.class);
29         assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
30
31         DocumentContext documentContext = JsonPath.parse(response.getBody());
32         Number id = documentContext.read("$.id");
33         assertThat(id).isEqualTo(99);
34
35         Double amount = documentContext.read("$.amount");
36         assertThat(amount).isEqualTo(123.45);
37     }
38
39     @Test
40     void shouldNotReturnACashCardWithAnUnknownId() {
41         ResponseEntity<String> response =
42             restTemplate.getForEntity("/cashcards/1000", String.class);
43
44         assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
45         assertThat(response.getBody()).isBlank();
46     }
47 }
```

```

44     }
45
46     @Test
47     // @ DirtiesContext
48     void shouldCreateANewCashCard() {
49         CashCard newCashCard = new CashCard(null, 250.00);
50         ResponseEntity<Void> createResponse =
51         restTemplate.postForEntity("/cashcards", newCashCard, Void.class);
52         assertEquals(createResponse.getStatusCode(), HttpStatus.CREATED);
53
54         URI locationOfNewCashCard = createResponse.getHeaders().getLocation();
55         ResponseEntity<String> getResponse =
56         restTemplate.getForEntity(locationOfNewCashCard, String.class);
57         assertEquals(getResponse.getStatusCode(), HttpStatus.OK);
58
59         DocumentContext documentContext = JsonPath.parse(getResponse.getBody());
60         Number id = documentContext.read("$.id");
61         Double amount = documentContext.read("$.amount");
62
63         assertEquals(id, 1);
64         assertEquals(amount, 250.00);
65     }
66
67     @Test
68     void shouldReturnAllCashCardsWhenListIsRequested() {
69         ResponseEntity<String> response = restTemplate.getForEntity("/cashcards",
70         String.class);
71         assertEquals(response.getStatusCode(), HttpStatus.OK);
72     }
73 }

```

Aquí estamos haciendo una solicitud al endpoint `/cashcards`.@ Dado que estamos obteniendo la lista completa de tarjetas, no necesitamos especificar información adicional en la solicitud.

Ejecuta las pruebas y observa el fallo.

La prueba debería fallar porque no hemos implementado un endpoint de Controlador para manejar esta solicitud GET.

¿Cómo crees que fallará? ¿Quizás un 404 NOT FOUND?

En una lección anterior escribimos una prueba que falló porque no existía aún un endpoint para coincidir con la ruta solicitada.@ El resultado fue un error 404 NOT FOUND.@ Podríamos esperar que ocurra lo mismo cuando ejecutemos la nueva prueba, ya que no hemos añadido ningún código al Controlador.

Veamos qué pasa.@ Ejecuta la prueba y busca el siguiente fallo:

```

expected: 200 OK
but was: 405 METHOD\_NOT\_ALLOWED

```

Los mensajes de error no hacen claro por qué recibimos un error 405 METHOD_NOT_ALLOWED.@ La razón es un poco difícil de descubrir, así que lo resumiremos rápidamente: ¡Ya hemos implementado un endpoint `/cashcards`, pero no para un verbo GET!

Este es el proceso de Spring:

- Spring recibe una solicitud al endpoint `/cashcards`.
- No hay un mapeo para el verbo HTTP GET en ese endpoint.
- Sin embargo, sí hay un mapeo a ese endpoint para el verbo HTTP POST. ¡Es el endpoint para la operación de Creación que implementamos en una lección anterior!
- Por lo tanto, Spring reporta un error 405 METHOD_NOT_ALLOWED en lugar de 404 NOT_FOUND: la ruta se encontró, pero no soporta el verbo GET.

Implementa el endpoint GET en el Controlador.

Para superar el error 405, necesitamos implementar el endpoint `/cashcards` en el Controlador usando una anotación `@GetMapping`:

```
1 @GetMapping()
2 private ResponseEntity<Iterable<CashCard>> findAll() {
3     return ResponseEntity.ok(cashCardRepository.findAll());
4 }
```

Entiende el método manejador.

Una vez más, estamos usando una de las implementaciones integradas de Spring Data: `CrudRepository.findAll().@` Nuestro Repositorio implementado, `CashCardRepository`, devolverá automáticamente todos los registros `CashCard` de la base de datos cuando se invoque `findAll()`.

Vuelve a ejecutar las pruebas.

Cuando volvamos a ejecutar las pruebas, veremos que todas pasan, incluyendo la prueba para el endpoint GET de una lista de `CashCard`.

```
1 [~/exercises] $ ./gradlew test
2 ...
3 BUILD SUCCESSFUL in 7s
```

14.4 Mejora la prueba de la lista

Como hemos hecho en lecciones anteriores, hemos probado que nuestro Controlador de API de Cash Card "escucha" nuestras llamadas HTTP y no se estrella al ser invocado, esta vez para un GET sin parámetros adicionales.

Mejoremos nuestras pruebas y asegurémonos de que se devuelvan los datos correctos desde nuestra solicitud HTTP.

Mejora la prueba.

Primero, completemos la prueba para afirmar los valores de datos esperados:

```
1 @Test
2 void shouldReturnAllCashCardsWhenListIsRequested() {
3     ResponseEntity<String> response = restTemplate.getForEntity("/cashcards",
4         String.class);
5     assertEquals(HttpStatus.OK, response.getStatusCode());
6     DocumentContext documentContext = JsonPath.parse(response.getBody());
7     int cashCardCount = documentContext.read("$.length()");
```

```

8      assertThat(cashCardCount).isEqualTo(3);
9
10     JSONArray ids = documentContext.read("$.id");
11     assertThat(ids).containsExactlyInAnyOrder(99, 100, 101);
12
13     JSONArray amounts = documentContext.read("$.amount");
14     assertThat(amounts).containsExactlyInAnyOrder(123.45, 100.0, 150.00);
15 }

```

Entiende la prueba.

- `documentContext.read($.length())`; ...`documentContext.read($.id)`; ...`documentContext`
¡Revisa estas nuevas expresiones JsonPath!
 - `documentContext.read($.length())` calcula la longitud del array.
 - `.read($.id)` recupera la lista de todos los valores `id` devueltos, mientras que `.read($.amount)` recopila todos los montos devueltos.
 - Para aprender más sobre JsonPath, un buen lugar para empezar está en la documentación de JsonPath.
- `assertThat(...).containsExactlyInAnyOrder(...)`: No hemos garantizado el orden de la lista de `CashCard` — salen en el orden que la base de datos elija devolverlos. @ Dado que no especificamos el orden, `containsExactlyInAnyOrder(...)` afirma que, aunque la lista debe contener todo lo que afirmamos, el orden no importa.

Ejecuta las pruebas.

¿Qué crees que será el resultado de la prueba?

```

Expecting actual:
  [123.45, 1.0, 150.0]
to contain exactly in any order:
  [123.45, 100.0, 150.0]
elements not found:
  [100.0]
and elements not expected:
  [1.0]

```

El mensaje de fallo señala exactamente la causa del fallo. @ Hemos escrito sigilosamente una prueba que falla, que espera que la segunda `Cash Card` tenga un monto de \$100.00, mientras que en `src/test/resources/data.sql` el valor real es \$1.00.

Corrige las pruebas y vuelve a ejecutarlas.

Cambia la expectativa para la `Cash Card` de \$1:

```

1 assertThat(amounts).containsExactlyInAnyOrder(123.45, 1.00, 150.00);

```

¡Y observa que la prueba pasa!

```

1 [~/exercises] $ ./gradlew test
2 ...
3 CashCardApplicationTests > shouldReturnAllCashCardsWhenListIsRequested() PASSED
4 ...
5 BUILD SUCCESSFUL in 6s

```


14.5 Interacción de pruebas y @DirtyiesContext

Tomemos un momento ahora para hablar sobre la anotación @DirtyiesContext.

Verás dos usos de esta anotación en la clase `CashCardApplicationTests`: Uno en la definición de la clase, y otro en la definición del método de prueba `shouldCreateANewCashCard`, que está comentado por ahora.

Expliquemos.

Primero, comenta la anotación a nivel de clase:

```
1 @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
2 //@DirtyiesContext(classMode = ClassMode.AFTER_EACH_TEST_METHOD)
3 class CashCardApplicationTests {
```

Ejecuta todas las pruebas:

```
1 [~/exercises] $ ./gradlew test
2 ...
3 org.opentest4j.AssertionFailedError:
4 expected: 3
5 but was: 4
6 ...
7 at
  app//example.cashcard.CashCardApplicationTests.shouldReturnAllCashCardsWhenListIsRequested(CashC
```

¡Nuestra nueva prueba `shouldReturnAllCashCardsWhenListIsRequested` no pasó esta vez! ¿Por qué?

La razón es que una de las otras pruebas está interfiriendo con nuestra nueva prueba al crear una nueva `Cash Card`. @DirtyiesContext soluciona este problema al hacer que Spring comience con un estado limpio, como si esas otras pruebas no se hubieran ejecutado. @ Quitarlo (comentándolo) de la clase causó que nuestra nueva prueba fallara.

Momento de aprendizaje.

Aunque puedes usar @DirtyiesContext para trabajar alrededor de la interacción entre pruebas, no deberías usarlo indiscriminadamente; deberías tener una buena razón. @ Nuestra razón aquí es limpiar después de crear una nueva `Cash Card`.

Deja DirtyiesContext comentado a nivel de clase y descoméntalo en el método que crea una nueva `Cash Card`:

```
1 //@DirtyiesContext(classMode = ClassMode.AFTER_EACH_TEST_METHOD)
2 class CashCardApplicationTests {
3     ...
4
5     @Test
6     @DirtyiesContext
7     void shouldCreateANewCashCard() {
8         ...
```

Ejecuta las pruebas y pasarán!

14.6 Paginación

¡Implementemos la paginación ahora, comenzando con una prueba!

Tenemos 3 `CashCards` en nuestra base de datos.® Configuremos una prueba para recuperarlas una a la vez (tamaño de página de 1), luego ordenemos sus montos de mayor a menor (descendente).

Escribe la prueba de paginación.

Añade la siguiente prueba a `CashCardApplicationTests`:

```
1 @Test
2 void shouldReturnAPageOfCashCards() {
3     ResponseEntity<String> response =
4         restTemplate.getForEntity("/cashcards?page=0&size=1", String.class);
5         assertEquals(response.getStatusCode(), HttpStatus.OK);
6
7         DocumentContext documentContext = JsonPath.parse(response.getBody());
8         JSONArray page = documentContext.read("$.[*]");
9         assertEquals(page.size(), 1);
10 }
```

Ejecuta las pruebas.

Cuando ejecutemos las pruebas, no deberíamos sorprendernos de que se devuelvan todas las `CashCards`.

expected: 1
but was: 3

Implementa la paginación en `CashCardController`.

¡Así que, añadamos nuestro nuevo endpoint al Controlador! Añade el siguiente método a `CashCardController` (no elimines el método `findAll()` existente):

```
1 @GetMapping
2 private ResponseEntity<List<CashCard>> findAll(Pageable pageable) {
3     Page<CashCard> page = cashCardRepository.findAll(
4         PageRequest.of(
5             pageable.getPageNumber(),
6             pageable.getPageSize()
7         ));
8     return ResponseEntity.ok(page.getContent());
9 }
```

Entiende el código de paginación.

- `findAll(Pageable pageable)`: `Pageable` es otro objeto que Spring Web nos proporciona.® Dado que especificamos los parámetros URI de `page=0&size=1`, `pageable` contendrá los valores que necesitamos.
- `PageRequest.of(pageable.getPageNumber(), pageable.getPageSize())`: `PageRequest` es una implementación básica de Java Bean de `Pageable`.® Cosas que quieren implementación de paginación y ordenación a menudo soportan esto, como algunos tipos de Repositorios de Spring Data.

- ¿Nuestro `CashCardRepository` soporta Paginación y Ordenación aún? ¡Descubrámoslo!

Intenta compilar.

Cuando ejecutemos las pruebas, descubriremos que nuestro código ni siquiera compila!

```
1 [~/exercises] $ ./gradlew test
2 ...
3 > Task :compileJava FAILED
4 exercises/src/main/java/example/cashcard/CashCardController.java:50: error: method
   findAll in interface CrudRepository<T,ID> cannot be applied to given types;
5     Page<CashCard> page = cashCardRepository.findAll(
6                                     ~
7     required: no arguments
8     found:    PageRequest
```

¡Pero claro! No hemos cambiado el Repositorio para extender la interfaz adicional. @ Así que hagámoslo. @ En `CashCardRepository.java`, también extiende `PagingAndSortingRepository`:

Extiende `PagingAndSortingRepository` y vuelve a ejecutar las pruebas.

Actualiza `CashCardRepository` para que también extienda `PagingAndSortingRepository`.

¡No olvides añadir la nueva importación!

```
1 import org.springframework.data.repository.PagingAndSortingRepository;
2 ...
3
4 interface CashCardRepository extends CrudRepository<CashCard, Long>,
   PagingAndSortingRepository<CashCard, Long> { ... }
```

Ahora nuestro repositorio sí soporta Paginación y Ordenación.

¡Pero nuestras pruebas aún fallan! Busca el siguiente fallo:

```
1 [~/exercises] $ ./gradlew test
2 ...
3 Failed to load ApplicationContext
4 java.lang.IllegalStateException: Failed to load ApplicationContext
5 ...
6 Caused by: java.lang.IllegalStateException: Ambiguous mapping. @ Cannot map
   'cashCardController' method
7 example.cashcard.CashCardController#findAll(Pageable)
8 to {GET [/cashcards]}: There is already 'cashCardController' bean method
9 example.cashcard.CashCardController#findAll() mapped.
```

(La salida real es inmensamente larga. @ Hemos incluido el mensaje de error más útil en la salida anterior.)

Entiende y resuelve el fallo.

Entonces, ¿qué pasó?

¡No eliminamos el método `findAll()` existente del Controlador!

¿Por qué es esto un problema? ¿No tenemos nombres de métodos únicos y todo compila?

El problema es que tenemos dos métodos mapeados al mismo endpoint. @ Spring detecta este error en tiempo de ejecución, durante el proceso de inicio de Spring.

Así que eliminemos el antiguo método `findAll()` problemático:

```
1 // Elimina este:
2 @GetMapping()
3 private ResponseEntity<Iterable<CashCard>> findAll() {
4     return ResponseEntity.ok(cashCardRepository.findAll());
5 }
```

Ejecuta las pruebas y asegúrate de que pasen.

```
1 BUILD SUCCESSFUL in 7s
```

A continuación, implementemos la Ordenación.

14.7 Ordenación

Nos gustaría que las Cash Cards se devuelvan en un orden que tenga sentido para los humanos. Así que ordenémoslas por monto en orden descendente con los montos más altos primero.

Escribe una prueba (que esperamos que falle).

Añade la siguiente prueba a `CashCardApplicationTests`:

```
1 @Test
2 void shouldReturnASortedPageOfCashCards() {
3     ResponseEntity<String> response =
4         restTemplate.getForEntity("/cashcards?page=0&size=1&sort=amount,desc",
5             String.class);
6     assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
7
8     DocumentContext documentContext = JsonPath.parse(response.getBody());
9     JSONArray read = documentContext.read("$[*]");
10    assertThat(read.size()).isEqualTo(1);
11
12    double amount = documentContext.read("$[0].amount");
13    assertThat(amount).isEqualTo(150.00);
14 }
```

Entiende la prueba.

El URI que estamos solicitando contiene información de paginación y ordenación: `/cashcards?page=0&size=1&sort=amount,desc`.

- **page=0**: Obtén la primera página. Los índices de página comienzan en 0.
- **size=1**: Cada página tiene tamaño 1.
- **sort=amount,desc**: La extracción de datos (usando más `JsonPath`!) y las afirmaciones acompañantes esperan que la Cash Card devuelta sea la de \$150.00.

¿Crees que la prueba pasará? Antes de ejecutarla, intenta descubrir si lo hará o no. Si crees que no pasará, ¿dónde crees que estará el fallo?

Ejecuta la prueba.

```
1 [~/exercises] $ ./gradlew test
2 ...
3 org.opentest4j.AssertionFailedError:
```

```
4 expected: 150.0
5 but was: 123.45
```

La prueba esperaba obtener la **Cash Card** de \$150.00, pero obtuvo la de \$123.45. ¿Por qué?

La razón es que, como no especificamos un orden de clasificación, las tarjetas se devuelven en el orden en que la base de datos las devuelve. Y esto resulta ser el mismo orden en que fueron insertadas.

Observación importante: No todas las bases de datos actuarán de la misma manera. @ Ahora, debería tener aún más sentido por qué especificamos un orden de clasificación (en lugar de depender del orden predeterminado de la base de datos).

Implementa la ordenación en el Controlador.

Añadir ordenación al código del Controlador es una adición de una sola línea súper simple. @ En la clase **CashCardController**, añade un parámetro adicional a la llamada **PageRequest.of()**:

```
1 PageRequest.of(
2     pageable.getPageNumber(),
3     pageable.getPageSize(),
4     pageable.getSort()
5 );
```

El método **getSort()** extrae el parámetro de consulta de ordenación del URI de la solicitud.

Vuelve a ejecutar las pruebas. ¡Pasan!

CashCardApplicationTests > shouldReturnAllCashCardsWhenListIsRequested() PASSED

Aprende rompiendo cosas.

Para ganar un poco más de confianza en la prueba, hagamos un experimento.

En la prueba, cambia el orden de clasificación de descendente a ascendente:

```
1 ResponseEntity<String> response =
    restTemplate.getForEntity("/cashcards?page=0&size=1&sort=amount,asc",
        String.class);
```

Esto debería causar que la prueba falle porque la primera **Cash Card** en orden ascendente debería ser la de \$1.00. @ Ejecuta las pruebas y observa el fallo:

```
CashCardApplicationTests > shouldReturnASortedPageOfCashCards() FAILED
org.opentest4j.AssertionFailedError:
    expected: 150.0
    but was: 1.0
```

¡Correcto! Este resultado refuerza nuestra confianza en la prueba. @ En lugar de escribir una prueba completamente nueva, usamos una existente para realizar un pequeño experimento.

Ahora cambiemos la prueba de vuelta para solicitar orden descendente para que vuelva a pasar.

14.8 Paginación y ordenación predeterminadas

Ahora tenemos un endpoint que requiere que el cliente envíe cuatro piezas de información: el índice y tamaño de la página, el orden de clasificación y la dirección. Esto es mucho pedir, así que hagámoslo más fácil para ellos.

Escribe una nueva prueba que no envíe parámetros de paginación ni ordenación.

Añadiremos una nueva prueba que espere valores predeterminados razonables para los parámetros.

Los valores predeterminados serán:

- Ordenar por monto ascendente.
- Un tamaño de página mayor que 3, para que se devuelvan todos nuestros datos ficticios.

```

1 @Test
2 void shouldReturnASortedPageOfCashCardsWithNoParametersAndUseDefaultValues() {
3     ResponseEntity<String> response = restTemplate.getForEntity("/cashcards",
4         String.class);
5     assertEquals(response.getStatusCode(), HttpStatus.OK);
6
7     DocumentContext documentContext = JsonPath.parse(response.getBody());
8     JSONArray page = documentContext.read("$.page");
9     assertEquals(page.size(), 3);
10
11     JSONArray amounts = documentContext.read("$.amounts");
12     assertEquals(amounts, new JSONArray(1.00, 123.45, 150.00));
13 }

```

Ejecuta las pruebas.

```

1 [~/exercises] $ ./gradlew test
2 ...
3 Actual and expected have the same elements but not in the same order, at index 0
4   actual element was:
5     123.45
6   whereas expected element was:
7     1.0
8   \end{verbatim}
9 El fallo de la prueba muestra:
10
11 \begin{itemize}
12   \item Se estan devolviendo todas las \texttt{Cash Cards}, ya que la afirmacion
13     \texttt{(page.size()).isEqualTo(3)} tuvo exito.
14   \item PERO: No estan ordenadas ya que la afirmacion
15     \texttt{(amounts).containsExactly(1.00, 123.45, 150.00)} falla.
16 \end{itemize}
17
18 Haz que la prueba pase.
19
20 Cambia la implementacion anadiendo una sola linea al metodo del Controlador:
21
22 \begin{lstlisting}[language=Java]
23 ...
24 PageRequest.of(
25     pageable.getPageNumber(),

```

```
24     pageable.getPageSize(),
25     pageable.getSortOr(Sort.by(Sort.Direction.ASC, "amount"))
26 ));
27 ...
```

Entiende la implementación.

Entonces, ¿qué acaba de pasar?

La respuesta es que el método `getSortOr()` proporciona valores predeterminados para los parámetros de página, tamaño y ordenación. @ Los valores predeterminados provienen de dos fuentes diferentes:

- Spring proporciona los valores predeterminados de página y tamaño (son 0 y 20, respectivamente). @ Un valor predeterminado de 20 para el tamaño de página explica por qué se devolvieron las tres **Cash Cards**. @ De nuevo: no necesitábamos definir estos valores predeterminados explícitamente. @ Spring los proporciona “fuera de la caja”.
- Definimos el parámetro de ordenación predeterminado en nuestro propio código, pasando un objeto `Sort` a `getSortOr()`:
 - `Sort.by(Sort.Direction.ASC, "amount")`

El resultado neto es que si alguno de los tres parámetros requeridos no se pasa a la aplicación, se proporcionarán valores predeterminados razonables.

Ejecuta las pruebas... ¡de nuevo!

¡Felicidades!

Todo pasa ahora.

```
1 [~/exercises] $ ./gradlew test
2 ...
3 BUILD SUCCESSFUL in 7s
```

14.9 Resumen

En este laboratorio, implementamos un endpoint “GET múltiples” y añadimos ordenación y paginación. @ Esto logró dos cosas:

- Aseguró que los datos recibidos del servidor estén en un orden predecible y comprensible.
- Protegió al cliente y al servidor de ser abrumados por una gran cantidad de datos (el tamaño de la página pone un límite a la cantidad de datos que pueden devolverse en una sola respuesta).

15 Simple Spring Security

15.1 ¿Qué es la seguridad?

Volvamos nuestra atención a nuestro "Steel Thread", concentrándonos en otro componente de la arquitectura: la Seguridad.

La seguridad en el software puede significar muchas cosas.® Este campo es un tema enorme que merece su propio curso.® En esta lección, hablaremos sobre la Seguridad Web.® Más específicamente, cubriremos cómo funcionan la Autenticación y la Autorización HTTP, las formas comunes en las que el ecosistema web es vulnerable a ataques, y cómo podemos usar Spring Security para prevenir accesos no autorizados a nuestro servicio de Tarjetas Familiares Cash Card.

15.2 Autenticación

Un usuario de una API puede ser una persona u otro programa, por lo que a menudo usaremos el término *Principal* como sinónimo de "usuario".® La autenticación es el acto de un *Principal* que prueba su identidad al sistema.® Una forma de hacerlo es proporcionar credenciales (por ejemplo, un nombre de usuario y contraseña usando Autenticación Básica).® Decimos que una vez que se han presentado las credenciales adecuadas, el *Principal* está autenticado, o en otras palabras, el usuario ha iniciado sesión con éxito.

HTTP es un protocolo sin estado, por lo que cada solicitud debe contener datos que prueben que proviene de un *Principal* autenticado.® Aunque es posible presentar las credenciales en cada solicitud, hacerlo es ineficiente porque requiere más procesamiento en el servidor.® En su lugar, se crea una Sesión de Autenticación (o Sesión de Autenticación, o simplemente Sesión) cuando un usuario se autentica.® Las sesiones pueden implementarse de muchas maneras.® Usaremos un mecanismo común: un Token de Sesión (una cadena de caracteres aleatorios) que se genera y se coloca en una Cookie.® Una Cookie es un conjunto de datos almacenado en un cliente web (como un navegador) y asociado a un URI específico.

Algunas ventajas de las Cookies:

- Las Cookies se envían automáticamente al servidor con cada solicitud (no es necesario escribir código adicional para que esto ocurra).® Siempre que el servidor verifique que el Token en la Cookie es válido, las solicitudes no autenticadas pueden ser rechazadas.
- Las Cookies pueden persistir durante un cierto tiempo incluso si se cierra la página web y se visita nuevamente más tarde.® Esta capacidad suele mejorar la experiencia del usuario en el sitio web.

15.2.1. Spring Security y Autenticación

Spring Security implementa la autenticación en la Cadena de Filtros.® La Cadena de Filtros es un componente de la arquitectura web de Java que permite a los programadores definir una secuencia de métodos que se ejecutan antes del Controlador.® Cada filtro en

la cadena decide si permite que el procesamiento de la solicitud continúe o no. @ Spring Security inserta un filtro que verifica la autenticación del usuario y devuelve una respuesta 401 UNAUTHORIZED si la solicitud no está autenticada.

15.3 Autorización

Hasta ahora hemos discutido la autenticación. @ Pero en realidad, la autenticación es solo el primer paso. @ La autorización ocurre después de la autenticación y permite que diferentes usuarios del mismo sistema tengan diferentes permisos.

Spring Security proporciona Autorización a través del Control de Acceso Basado en Roles (RBAC, por sus siglas en inglés). @ Esto significa que un *Principal* tiene una serie de Roles. @ Cada recurso (o operación) especifica qué Roles debe tener un *Principal* para realizar acciones con la autorización adecuada. @ Por ejemplo, un usuario con un Rol de Administrador probablemente esté autorizado a realizar más acciones que un usuario con un Rol de Propietario de Tarjeta. @ Puedes configurar la autorización basada en roles tanto a nivel global como por método.

15.4 Política de Origen Igual (Same Origin Policy)

La web es un lugar peligroso, donde los actores maliciosos están constantemente intentando explotar vulnerabilidades de seguridad. @ El mecanismo más básico de protección depende de que los clientes y servidores HTTP implementen la Política de Origen Igual (SOP, por sus siglas en inglés). @ Esta política establece que solo los scripts contenidos en una página web pueden enviar solicitudes al origen (URI) de la página web.

La SOP es crítica para la seguridad de los sitios web porque sin esta política, cualquiera podría escribir una página web que contenga un script que envíe solicitudes a cualquier otro sitio. @ Por ejemplo, consideremos un sitio web bancario típico. @ Si un usuario está conectado a su cuenta bancaria y visita una página web maliciosa (en una pestaña o ventana diferente del navegador), las solicitudes maliciosas podrían enviarse (con las Cookies de Autenticación) al sitio bancario. ¡Esto podría resultar en acciones no deseadas, como un retiro de la cuenta del usuario!

15.5 Intercambio de Recursos de Origen Cruzado (CORS)

A veces, un sistema consiste en servicios que se ejecutan en varias máquinas con diferentes URIs (es decir, Microservicios). @ El Intercambio de Recursos de Origen Cruzado (CORS, por sus siglas en inglés) es una manera en que los navegadores y servidores pueden cooperar para relajar la SOP. @ Un servidor puede permitir explícitamente una lista de “orígenes permitidos” de solicitudes que provienen de un origen fuera del servidor.

Spring Security proporciona la anotación `@CrossOrigin`, que te permite especificar una lista de sitios permitidos. ¡Ten cuidado! Si usas la anotación sin argumentos, permitirá todos los orígenes, ¡así que tenlo en mente!

15.6 Explotaciones web comunes

Junto con explotar vulnerabilidades de seguridad conocidas, los actores maliciosos en la web también están constantemente descubriendo nuevas vulnerabilidades. @ Afortunadamente, Spring Security proporciona un conjunto de herramientas potente para protegerse contra explotaciones de seguridad comunes. @ Discutamos dos explotaciones comunes, cómo funcionan y cómo Spring Security ayuda a mitigarlas.

15.6.1. Falsificación de Solicitudes entre Sitios (CSRF)

Un tipo de vulnerabilidad es la Falsificación de Solicitudes entre Sitios (CSRF, por sus siglas en inglés), que a menudo se pronuncia “Sea-Surf” y también se conoce como Session Riding. @ Session Riding está habilitado por Cookies. @ Los ataques CSRF ocurren cuando un código malicioso envía una solicitud a un servidor donde un usuario está autenticado. @ Cuando el servidor recibe la Cookie de Autenticación, no tiene forma de saber si la víctima envió la solicitud dañina de manera no intencional.

Para protegerse contra ataques CSRF, puedes usar un Token CSRF. Un Token CSRF es diferente de un Token de Autenticación porque se genera un token único en cada solicitud. @ Esto hace que sea más difícil para un actor externo insertarse en la “conversación” entre el cliente y el servidor.

Afortunadamente, Spring Security tiene soporte integrado para tokens CSRF que está habilitado por defecto. @ Aprenderás más sobre esto en el próximo laboratorio.

15.6.2. Guion Transversal entre Sitios (XSS)

Quizás aún más peligroso que la vulnerabilidad CSRF es el Guion Transversal entre Sitios (XSS, por sus siglas en inglés). @ Esto ocurre cuando un atacante logra de alguna manera “engañar” a la aplicación víctima para que ejecute código arbitrario. @ Hay muchas maneras de hacerlo. @ Un ejemplo simple es guardar una cadena en una base de datos que contenga una etiqueta `<script>`, y luego esperar hasta que la cadena se renderice en una página web, lo que resulta en la ejecución del script.

XSS es potencialmente más peligroso que CSRF. En CSRF, solo se pueden ejecutar acciones para las que un usuario está autorizado. @ Sin embargo, en XSS, se ejecuta código malicioso arbitrario en el cliente o en el servidor. @ Además, los ataques XSS no dependen de la Autenticación. @ Más bien, los ataques XSS dependen de “agujeros” de seguridad causados por malas prácticas de programación.

La principal manera de protegerse contra ataques XSS es procesar adecuadamente todos los datos de fuentes externas (como formularios web y cadenas de consulta de URI). @ En el caso del ejemplo de la etiqueta `<script>`, los ataques pueden mitigarse escapando correctamente los caracteres HTML especiales cuando se renderiza la cadena.

¡Uf! Eso es todo para nuestra breve introducción a la Seguridad Web. @ La Seguridad Web es un tema amplio y diverso, ¡pero ahora tienes una visión general de cómo puedes usar Spring Security para proteger a tus usuarios y aplicaciones!

16 Laboratorio: Seguridad con Spring Security

16.1 Entender nuestros requisitos de seguridad

¿Quién debería poder gestionar cualquier Tarjeta Cash dada?

En nuestro dominio simple, establezcamos que el usuario que creó la Tarjeta Cash "posee" la Tarjeta Cash. @ Así, ellos son el "propietario de la tarjeta". @ Solo el propietario de la tarjeta puede ver o actualizar una Tarjeta Cash.

La lógica será algo como esto:

- SI el usuario está autenticado
- ... Y está autorizado como "propietario de la tarjeta"
- Y posee la Tarjeta Cash solicitada
- ENTONCES completar la solicitud del usuario
- PERO no permitir que los usuarios accedan a Tarjetas Cash que no poseen.

16.2 Revisión de actualizaciones del laboratorio anterior

En este laboratorio aseguraremos nuestra API de Tarjetas Familiares Cash Card y restringiremos el acceso a cualquier Tarjeta Cash al "propietario" de la tarjeta.

Para prepararnos para esto, introdujimos el concepto de propietario en la aplicación.

El propietario es la identidad única de la persona que creó y puede gestionar una Tarjeta Cash dada.

Revisemos los siguientes cambios que realizamos por ti:

- Propietario añadido como campo al registro Java CashCard.

```
1 package example.cashcard;
2
3 import org.springframework.data.annotation.Id;
4
5 record CashCard(@Id Long id, Double amount, String owner) {
6 }
```

- Propietario añadido a todos los archivos .sql en src/main/resources/ y src/test/resources/.

```
1 CREATE TABLE cash_card
2 (
3     ID          BIGINT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
4     AMOUNT      NUMBER NOT NULL DEFAULT 0,
5     OWNER       VARCHAR(256) NOT NULL
6 );
7
8 INSERT INTO CASH_CARD(ID, AMOUNT, OWNER) VALUES (99, 123.45, 'sarah1');
9 INSERT INTO CASH_CARD(ID, AMOUNT, OWNER) VALUES (100, 1.00, 'sarah1');
10 INSERT INTO CASH_CARD(ID, AMOUNT, OWNER) VALUES (101, 150.00, 'sarah1');
```

- Propietario añadido a todos los archivos `.json` en `src/test/resources/example/cashcard`.

```

1 [
2   {"id": 99, "amount": 123.45, "owner": "sarah1"},
3   {"id": 100, "amount": 1.00, "owner": "sarah1"},
4   {"id": 101, "amount": 150.00, "owner": "sarah1"}
5 ]
6 {
7   "id": 99,
8   "amount": 123.45,
9   "owner": "sarah1"
10 }
```

Todo el código de la aplicación y las pruebas se actualizaron para soportar el nuevo campo propietario. @ No ha cambiado ninguna funcionalidad como resultado de estas actualizaciones.

16.3 Añadir la dependencia de Spring Security

Podemos añadir soporte para Spring Security añadiendo la dependencia adecuada.

Añade la dependencia.

Añade lo siguiente al archivo `build.gradle` en la sección `dependencies`:

```

1 dependencies {
2     implementation 'org.springframework.boot:spring-boot-starter-web'
3
4     // Añade la siguiente dependencia
5     implementation 'org.springframework.boot:spring-boot-starter-security'
6     ...
}
```

Ejecuta las pruebas.

Hemos añadido capacidades de Spring Security a nuestra aplicación, pero no hemos cambiado ningún código.

Entonces, ¿qué esperamos que pase cuando ejecutemos las pruebas?

Nota que siempre ejecutaremos `./gradlew test` para ejecutar las pruebas.

```

1 [~/exercises] $ ./gradlew test
2 ...
3 CashCardApplicationTests > shouldReturnASortedPageOfCashCards() FAILED
4 ...
5 CashCardApplicationTests > shouldReturnACashCardWhenDataIsSaved() FAILED
6 ...
7 CashCardApplicationTests > shouldCreateANewCashCard() FAILED
8 ...
9 CashCardApplicationTests > shouldReturnAPageOfCashCards() FAILED
10 ...
11 CashCardApplicationTests > shouldReturnAllCashCardsWhenListIsRequested() FAILED
12 ...
13 CashCardApplicationTests >
14     shouldReturnASortedPageOfCashCardsWithNoParametersAndUseDefaultValues() FAILED
15 ...
16 CashCardApplicationTests > shouldNotReturnACashCardWithAnUnknownId() FAILED
```

```
16 11 tests completed, 7 failed
17 > Task :test FAILED
```

¡Las cosas están realmente rotas!

Todos los métodos de prueba dentro de `CashCardApplicationTests` fallaron.

Muchas fallas son similares a la siguiente:

```
expected: <SOME NUMBER>
but was: 0
```

En la mayoría de los casos, nuestras pruebas esperan que los datos de `CashCard` sean devueltos por nuestra API, pero no se devolvió nada.

¿Por qué crees que todas las pruebas de nuestra API de Tarjeta Cash fallan después de añadir la dependencia de Spring Security?

Entiende por qué todo está roto.

¿Qué pasó aquí?

Cuando añadimos la dependencia de Spring Security a nuestra aplicación, la seguridad se habilitó por defecto.

Dado que no hemos especificado cómo se realizan la autenticación y la autorización dentro de nuestra API de Tarjeta Cash, Spring Security ha bloqueado completamente nuestra API.

Bueno, ¡mejor prevenir que lamentar, ¿verdad?

A continuacion, configuremos Spring Security para nuestra aplicación.

16.4 Satisfacer las dependencias de Spring Security

A continuacion, nos enfocaremos en hacer que nuestras pruebas pasen nuevamente proporcionando la configuración mínima necesaria para Spring Security.

Hemos proporcionado otro archivo por ti: `example/cashcard/SecurityConfig.java`.@ Este será el Bean de Java donde configuraremos Spring Security para nuestra aplicación.

Descomenta `SecurityConfig.java` y revisa.

Abre `SecurityConfig`.

Nota que la mayor parte del archivo está comentada.

Descomenta todas las líneas comentadas dentro de `SecurityConfig`, incluyendo todos los métodos e instrucciones de importación.

```
1 package example.cashcard;
2 ...
3
4 class SecurityConfig {
5
6     SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
7         return http.build();
8     }
9 }
```

```

8     }
9
10    @Bean
11    PasswordEncoder passwordEncoder() {
12        return new BCryptPasswordEncoder();
13    }
14    ...

```

`filterChain` devuelve `http.build()`, que es lo mínimo necesario por ahora.

Nota: Por favor, ignora el método `passwordEncoder()` por ahora.

Habilita Spring Security.

En este momento, `SecurityConfig` es solo una clase Java sin referencia, ya que nada la está usando.

Convirtamos `SecurityConfig` en nuestro Bean de configuración para Spring Security.

```

1 // Aníade esta anotación
2 @Configuration
3 class SecurityConfig {
4
5     // Aníade esta anotación
6     @Bean
7     SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
8         return http.build();
9     }
10    ...

```

Entiende las anotaciones.

- `@Configuration class SecurityConfig {...}`: La anotación `@Configuration` le indica a Spring que use esta clase para configurar Spring y Spring Boot en sí. @ Cualquier Bean especificado en esta clase estará ahora disponible para el motor de Auto Configuración de Spring.
- `@Bean SecurityFilterChain filterChain`: Spring Security espera un Bean para configurar su Cadena de Filtros, que aprendiste en la lección de Simple Spring Security. @ Anotar un método que devuelva un `SecurityFilterChain` con `@Bean` satisface esta expectativa.

Ejecuta las pruebas.

Cuando ejecutes las pruebas, verás que una vez más todas las pruebas pasan, excepto la prueba para crear una nueva `CashCard` mediante un POST.

```

1 [~/exercises] $ ./gradlew test
2 ...
3 CashCardApplicationTests > shouldCreateANewCashCard() FAILED
4     org.opentest4j.AssertionFailedError:
5         expected: 201 CREATED
6         but was: 403 FORBIDDEN
7     ...
8 11 tests completed, 1 failed

```

Esto es esperado. @ Lo cubriremos en profundidad un poco más adelante.

16.5 Configurar la autenticación básica

Hasta ahora hemos inicializado Spring Security, pero no hemos asegurado realmente nuestra aplicación.

Ahora aseguraremos nuestra aplicación configurando la autenticación básica.

Configura la autenticación básica.

Actualiza `SecurityConfig.filterChain` con lo siguiente para habilitar la autenticación básica:

```
1 @Bean
2 SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
3     http
4         .authorizeHttpRequests(request -> request
5             .requestMatchers("/cashcards/**")
6             .authenticated())
7         .httpBasic(Customizer.withDefaults())
8         .csrf(csrf -> csrf.disable());
9     return http.build();
10 }
```

Entiende la configuración de Spring Security.

¡Eso son muchos llamados a métodos!

Aquí, si explicamos el patrón de construcción de Spring Security en un lenguaje más comprensible, vemos:

- Todas las solicitudes HTTP a los endpoints `cashcards/` deben estar autenticadas usando seguridad de Autenticación Básica HTTP (nombre de usuario y contraseña).
- Además, no se requiere seguridad CSRF.

Nota: Hablaremos sobre la seguridad CSRF más adelante en este laboratorio.

Ejecuta las pruebas.

¿Qué pasará cuando ejecutemos nuestras pruebas?

Cuando ejecutes las pruebas, notarás que la mayoría de las pruebas fallan con un código de estado HTTP 401 UNAUTHORIZED, como el siguiente:

```
1 [~/exercises] $ ./gradlew test
2 ...
3 expected: 200 OK
4 but was: 401 UNAUTHORIZED
```

¡Aunque no lo parezca, esto es progreso!

Hemos habilitado la autenticación básica, requiriendo que las solicitudes suministren un nombre de usuario y contraseña.

Nuestras pruebas no proporcionan un nombre de usuario y contraseña con nuestras solicitudes HTTP. Así que hagámoslo a continuación.

16.6 Prueba de autenticación básica

Como aprendimos en la lección acompañante, hay muchas maneras de proporcionar información de autenticación y autorización de usuario para una aplicación Spring Boot usando Spring Security.

Para nuestras pruebas, configuraremos un servicio solo para pruebas que Spring Security usará para este propósito: un `InMemoryUserDetailsManager`.

Similar a cómo configuramos una base de datos en memoria usando H2 para probar Spring Data, configuraremos un servicio en memoria con usuarios de prueba para probar Spring Security.

Configura un `UserDetailsService` solo para pruebas.

¿Cuál nombre de usuario y contraseña deberíamos enviar en nuestras solicitudes HTTP de prueba?

Cuando revisaste los cambios en `src/test/resources/data.sql`, deberías haber visto que establecimos un valor OWNER para cada `CashCard` en la base de datos al nombre de usuario `sarah1`.@ Por ejemplo:

```
1 INSERT INTO CASH_CARD(ID, AMOUNT, OWNER) VALUES (100, 1.00, 'sarah1');
```

Proporcionemos un `UserDetailsService` solo para pruebas con el usuario `sarah1`.

Añade el siguiente Bean a `SecurityConfig`.

```
1 @Bean
2 UserDetailsService testOnlyUsers(PasswordEncoder passwordEncoder) {
3     User.UserBuilder users = User.builder();
4     UserDetails sarah = users
5         .username("sarah1")
6         .password(passwordEncoder.encode("abc123"))
7         .roles() // Sin roles por ahora
8         .build();
9     return new InMemoryUserDetailsManager(sarah);
10 }
```

Esta configuración de `UserDetailsService` debería ser comprensible: configura un usuario llamado `sarah1` con contraseña `abc123`.

El contenedor IoC de Spring encontrará el Bean `UserDetailsService` y Spring Data lo usará cuando sea necesario.

Configura la autenticación básica en las pruebas HTTP.

Selecciona un método de prueba que use `restTemplate.getForEntity` y actualízalo con autenticación básica para `sarah1`.

```
1 void shouldReturnACashCardWhenDataIsSaved() {
2     ResponseEntity<String> response = restTemplate
3         .withBasicAuth("sarah1", "abc123") // Aníade esto
4         .getForEntity("/cashcards/99", String.class);
5     assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
6     ...
}
```


Ejecuta las pruebas.

¡La prueba actualizada que proporciona las credenciales básicas ahora debería pasar!

```
1 [~/exercises] $ ./gradlew test
2 ...
3 CashCardApplicationTests > shouldReturnACashCardWhenDataIsSaved() PASSED
4 ...
```

Actualiza todas las pruebas restantes de `CashCardApplicationTests` y vuelve a ejecutar las pruebas.

Ahora, un poco de tedio: Actualiza todas las pruebas basadas en `restTemplate` restantes para suministrar `.withBasicAuth("sarah1", "abc123")` con cada solicitud HTTP.

Cuando termines, vuelve a ejecutar la prueba.

```
1 [~/exercises] $ ./gradlew test
2 ...
3 BUILD SUCCESSFUL in 9s
```

¡Todo pasa!

¡Felicidades, has implementado y probado la Autenticación Básica!

Verifica la Autenticación Básica con pruebas adicionales.

Ahora añadamos pruebas que esperen una respuesta 401 UNAUTHORIZED cuando se envíen credenciales incorrectas usando autenticación básica.

```
1 @Test
2 void shouldNotReturnACashCardWhenUsingBadCredentials() {
3     ResponseEntity<String> response = restTemplate
4         .withBasicAuth("BAD-USER", "abc123")
5         .getForEntity("/cashcards/99", String.class);
6     assertThat(response.getStatusCode()).isEqualTo(HttpStatus.UNAUTHORIZED);
7
8     response = restTemplate
9         .withBasicAuth("sarah1", "BAD-PASSWORD")
10        .getForEntity("/cashcards/99", String.class);
11    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.UNAUTHORIZED);
12 }
```

Esta prueba debería pasar...

```
1 [~/exercises] $ ./gradlew test
2 ...
3 CashCardApplicationTests > shouldNotReturnACashCardWhenUsingBadCredentials() PASSED
```

¡Éxito! Ahora que hemos implementado la autenticación, pasemos a implementar la autorización a continuación.

16.7 Soporte para autorización

Como aprendimos en la lección acompañante, Spring Security soporta muchas formas de autorización.

Aquí implementaremos el Control de Acceso Basado en Roles (RBAC).

Es probable que un servicio de usuarios proporcione acceso a muchos usuarios autenticados, pero solo los "propietarios de tarjetas" deberían poder acceder a las Tarjetas Familiares Cash gestionadas por nuestra aplicación. Hagamos esas actualizaciones ahora.

Añade usuarios y roles al Bean `UserDetailsService`.

Para probar la autorización, necesitamos múltiples usuarios de prueba con una variedad de roles.

Actualiza `SecurityConfig.testOnlyUsers` y añade el rol `CARD-OWNER` a `sarah1`.

Además, añadamos un nuevo usuario llamado `hank-owns-no-cards` con un rol de `NON-OWNER`.

```

1 ...
2 @Bean
3 UserDetailsService testOnlyUsers(PasswordEncoder passwordEncoder) {
4     User.UserBuilder users = User.builder();
5     UserDetails sarah = users
6         .username("sarah1")
7         .password(passwordEncoder.encode("abc123"))
8         .roles("CARD-OWNER") // nuevo rol
9         .build();
10    UserDetails hankOwnsNoCards = users
11        .username("hank-owns-no-cards")
12        .password(passwordEncoder.encode("qrs456"))
13        .roles("NON-OWNER") // nuevo rol
14        .build();
15    return new InMemoryUserDetailsManager(sarah, hankOwnsNoCards);
16 }
```

Prueba la verificación de roles.

Añadamos una prueba que fallará al principio, pero pasará cuando implementemos completamente la autorización.

Aquí, afirmaremos que el usuario `hank-owns-no-cards` no debería tener acceso a una `CashCard` ya que ese usuario no es un `CARD-OWNER`.

```

1 @Test
2 void shouldRejectUsersWhoAreNotCardOwners() {
3     ResponseEntity<String> response = restTemplate
4         .withBasicAuth("hank-owns-no-cards", "qrs456")
5         .getForEntity("/cashcards/99", String.class);
6     assertThat(response.getStatusCode()).isEqualTo(HttpStatus.FORBIDDEN);
7 }
```

Espera, La `CashCard` con ID 99 pertenece a `sarah1`, ¿verdad? ¿No debería solo `sarah1` tener acceso a esos datos independientemente del rol?

¡Tienes razón! Ten eso en mente para más adelante en este laboratorio.

Ejecuta las pruebas.

Vemos que nuestra nueva prueba falla cuando la ejecutamos.

```

1 [~/exercises] $ ./gradlew test
2 ...
3 CashCardApplicationTests > shouldRejectUsersWhoAreNotCardOwners() FAILED
```

```

4 org.opentest4j.AssertionFailedError:
5 expected: 403 FORBIDDEN
6 but was: 200 OK

```

¿Por qué `hank-owns-no-cards` pudo acceder a una `CashCard` como lo indica la respuesta 200 OK?

Aunque hemos dado roles a los usuarios de prueba, no estamos haciendo cumplir la seguridad basada en roles.

Habilita la seguridad basada en roles.

Edita `SecurityConfig.filterChain` para restringir el acceso solo a usuarios con el rol `CARD-OWNER`.

```

1 @Bean
2 SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
3     http
4         .authorizeHttpRequests(request -> request
5             .requestMatchers("/cashcards/**")
6                 .hasRole("CARD-OWNER")) // habilita RBAC: Reemplaza la llamada
7         .authenticated() con la llamada hasRole(...)
8         .httpBasic(Customizer.withDefaults())
9         .csrf(csrf -> csrf.disable());
10    return http.build();

```

Ejecuta las pruebas.

Vemos que nuestras pruebas pasan!

```

1 [~/exercises] $ ./gradlew test
2 ...
3 CashCardApplicationTests > shouldRejectUsersWhoAreNotCardOwners() PASSED

```

Ahora hemos habilitado con éxito la autorización basada en RBAC!

16.8 Propiedad de Tarjeta Cash: Actualizaciones del Repositorio

Como se menciona en el ejercicio anterior, tenemos un agujero de seguridad evidente en nuestra aplicación.

Cualquier usuario autenticado con el rol `CARD-OWNER` puede ver las Tarjetas Familiares Cash de cualquier otra persona!

Para corregir esto, actualizaremos nuestras pruebas, `CashCardRepository` y `CashCardController`:

- Anadiremos funcionalidad a `CashCardRepository` para restringir o .especificar"las consultas al propietario correcto.
- A continuacion, actualizaremos nuestro `CashCardController` para garantizar que solo se use el propietario correcto.

Momento de aprendizaje: Mejores prácticas.

Espera, ¿No es este laboratorio todo sobre el proyecto Spring Security y su tecnología increíble? ¿No puede Spring Security hacer toda esta validación de propiedad para que no tengamos que modificar nuestros Repositorios y Controladores?

Respuesta: Este laboratorio y la lección asociada tratan sobre asegurar una API HTTP. @ Sí, las tecnologías de seguridad como Spring Security son increíbles. @ Sí, hay características como la Seguridad de Métodos de Spring Security que podrían ayudar en esta situación, pero sigue siendo tu responsabilidad como desarrollador escribir código seguro y seguir las mejores prácticas de seguridad. @ Por ejemplo, ¡no escribas código que permita a los usuarios acceder a los datos de otros usuarios!

Ahora, actualicemos nuestras pruebas y `CashCardRepository`.

Añade una nueva `CashCard` para un usuario llamado `kumar2`.

Actualiza `src/test/resources/data.sql` con un registro de `CashCard` propiedad de un usuario diferente:

```
1 ...
2 INSERT INTO CASH_CARD(ID, AMOUNT, OWNER) VALUES (102, 200.00, 'kumar2');
```

Prueba que los usuarios no puedan acceder a los datos de los demás.

Añadamos una prueba que afirme explícitamente que nuestra API devuelve un 404 NOT FOUND cuando un usuario intenta acceder a una `CashCard` que no posee.

Nota: Podrías preguntarte por qué queremos devolver una respuesta 404 NOT FOUND en lugar de algo más, como 401 UNAUTHORIZED. @ Un argumento a favor de elegir devolver NOT FOUND es que es la misma respuesta que daríamos si la `CashCard` solicitada no existiera. @ Es más seguro errar del lado de no revelar información sobre datos que no están autorizados para el usuario.

Ahora haremos que `sarah1` intente acceder a los datos de `kumar2`.

```
1 @Test
2 void shouldNotAllowAccessToCashCardsTheyDoNotOwn() {
3     ResponseEntity<String> response = restTemplate
4         .withBasicAuth("sarah1", "abc123")
5         .getForEntity("/cashcards/102", String.class); // datos de kumar2
6     assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
7 }
```

Ejecuta las pruebas.

Que crees que pasara cuando ejecutemos las pruebas? Averiguemoslo.

Como era de esperar, nuestra nueva prueba falla, junto con muchas otras.

```
1 CashCardApplicationTests > shouldNotAllowAccessToCashCardsTheyDoNotOwn() FAILED
2 org.opentest4j.AssertionFailedError:
3   expected: 404 NOT_FOUND
4   but was: 200 OK
```

Que esta pasando aqui?

Respuesta: Actualmente, el usuario `sarah1` puede ver los datos de `kumar2` porque:

- `sarah1` está autenticado.

- `sarah1` es un `CARD-OWNER` autorizado.

Además, nuestra prueba para obtener una lista de `CashCards` también está fallando:

```
1 CashCardApplicationTests > shouldReturnAllCashCardsWhenListIsRequested() FAILED
2 org.opentest4j.AssertionFailedError:
3   expected: 3
4   but was: 4
```

¿Por qué estamos devolviendo demasiadas `Cash Cards`? Por la misma razón que arriba: `sarah1` tiene acceso a los datos de `kumar2`.@ La `Cash Card` de `kumar2` está siendo devuelta en la lista de `Cash Cards` de `sarah1`.

Impidamos que los usuarios accedan a los datos de los demás.

Actualiza `CashCardRepository` con nuevos métodos `findBy...`

Lo más simple que podemos hacer es siempre filtrar nuestro acceso a datos por el propietario de la `CashCard`.

Hagámoslo en nuestro Repositorio.

Necesitaremos filtrar por propietario al encontrar tanto una sola `CashCard` como una lista de `CashCards`.

Edita `CashCardRepository` para añadir dos nuevos métodos de búsqueda.

Asegúrate de incluir las nuevas importaciones para `Page` y `PageRequest`.

```
1 ...
2 import org.springframework.data.domain.Page;
3 import org.springframework.data.domain.PageRequest;
4 ...
5
6 interface CashCardRepository extends CrudRepository<CashCard, Long>,
7     PagingAndSortingRepository<CashCard, Long> {
8     CashCard findByIdAndOwner(Long id, String owner);
9     Page<CashCard> findByOwner(String owner, PageRequest pageRequest);
10 }
```

Como se menciona en los laboratorios y lecciones de Spring Data, Spring Data se encargará de las implementaciones reales (escribir las consultas SQL) por nosotros.

Nota: Podrías preguntarte si Spring Data te permite escribir tu propio SQL.@ Después de todo, Spring Data no puede anticipar cada necesidad, ¿verdad? ¡La respuesta es Sí! Es fácil para ti escribir tu propio código SQL.@ La documentación de Métodos de Consulta de Spring Data describe cómo hacerlo usando la anotación `@Query`.

Sin embargo, Spring Data es perfectamente capaz de generar el SQL para las consultas que necesitamos. ¡Gracias, Spring Data!

A continuación, actualicemos el Controlador.

16.9 Propiedad de Tarjeta Cash: Actualizaciones del Controlador

El `CashCardRepository` ahora soporta filtrar los datos de `CashCard` por propietario.

Pero no estamos usando esta nueva funcionalidad. @ Hagamos esas actualizaciones a `CashCardController` ahora introduciendo un concepto que explicamos en la lección escrita: el *Principal*.

Al igual que con otros objetos útiles, el *Principal* está disponible para que lo usemos en nuestro Controlador. @ El *Principal* contiene la información autenticada y autorizada de nuestro usuario.

Actualiza el endpoint GET por ID del Controlador.

Actualiza `CashCardController` para pasar la información del *Principal* al nuevo método `findByIdAndOwner` de nuestro Repositorio.

Asegúrate de añadir la nueva instrucción de importación.

```
1 import java.security.Principal;
2 ...
3
4 @GetMapping("/{requestedId}")
5 private ResponseEntity<CashCard> findById(@PathVariable Long requestedId, Principal
   principal) {
6     Optional<CashCard> cashCardOptional =
7     Optional.ofNullable(cashCardRepository.findByIdAndOwner(requestedId,
8     principal.getName()));
9     if (cashCardOptional.isPresent()) {
10         ...
11     }
```

Nota que `principal.getName()` devolverá el nombre de usuario proporcionado desde la Autenticación Básica.

Ejecuta las pruebas.

El GET está pasando, pero nuestras pruebas para listas de `Cash Card` están fallando.

```
1 CashCardApplicationTests > shouldReturnASortedPageOfCashCards() FAILED
2 ...
3 CashCardApplicationTests > shouldReturnACashCardWhenDataIsSaved() PASSED
4 ...
5 CashCardApplicationTests > shouldReturnAllCashCardsWhenListIsRequested() FAILED
6 ...
7 CashCardApplicationTests >
   shouldReturnASortedPageOfCashCardsWithNoParametersAndUseDefaultValues() FAILED
8 ...
```

Actualiza el endpoint GET para listas del Controlador.

Edita `CashCardController` para filtrar las listas por propietario.

```
1 @GetMapping
2 private ResponseEntity<List<CashCard>> findAll(Pageable pageable, Principal
   principal) {
3     Page<CashCard> page = cashCardRepository.findByOwner(principal.getName(),
4     PageRequest.of(
5     pageable.getPageNumber(),
6     ...
```

Una vez más, obtenemos el nombre de usuario autenticado del método `principal.getName()`.

Ejecuta las pruebas.

¡Todas pasan!

```
1 BUILD SUCCESSFUL in 8s
```

¡Se ve bien!

16.10 Propiedad de Tarjeta Cash: Actualizaciones de creación

Tenemos un agujero de seguridad más: la creación de `CashCards`.

El *Principal* autenticado y autorizado debería usarse como el propietario al crear una nueva `CashCard`.

Pregunta: ¿Qué pasaría si usáramos automáticamente el valor de propietario enviado?

Respuesta: ¡Corremos el riesgo de permitir que los usuarios creen `CashCards` para alguien más!

Aseguremos que solo el *Principal* autenticado y autorizado posea las `CashCards` que están creando.

Actualiza la prueba POST.

Para probar que no necesitamos enviar un propietario, usemos `null` como el propietario para la `CashCard`.

```
1 void shouldCreateANewCashCard() {
2     CashCard newCashCard = new CashCard(null, 250.00, null);
3     ...
}
```

Ejecuta las pruebas.

¿Qué crees que pasará cuando ejecutemos las pruebas? Probablemente fallarán, pero ¿puedes adivinar por qué?

Para ayudarnos, pasa la bandera `-info` al ejecutar las pruebas para obtener un poco más de insight sobre lo que está pasando:

```
1 [~/exercises] $ ./gradlew test --info
2 ...
3 CashCardApplicationTests > shouldCreateANewCashCard() FAILED
4 org.opentest4j.AssertionFailedError:
5   expected: 201 CREATED
6   but was: 403 FORBIDDEN
```

Verás que las pruebas fallan con un error 403 FORBIDDEN.

¿Pero por qué? Buscando a través del stacktrace encontramos lo siguiente:

```
1 DbActionExecutionException: Failed to execute InsertRoot[entity=CashCard[id=null,
   amount=250.0, owner=null], idValueSource=GENERATED]] with root cause
2
3 org.h2.jdbc.JdbcSQLIntegrityConstraintViolationException: NULL not allowed for column
   "OWNER"; SQL statement:
4 INSERT INTO "CASH_CARD" ("AMOUNT", "OWNER") VALUES (?, ?) [23502-214]
```

Interesante! Parece que intentamos crear una nueva `CASH_CARD` en la base de datos, pero `NULL` no está permitido como `OWNER`, que es exactamente lo que pasamos para el `OWNER` en nuestra prueba.

Seguro, más pistas se pueden encontrar en el `main/resources/schema.sql`, que configura nuestras tablas de base de datos:

```
1 CREATE TABLE cash_card
2 (
3     ...
4     OWNER    VARCHAR(256) NOT NULL
5 );
```

¿Qué significa todo esto? Significa que aunque las pruebas (y los usuarios) ven un error 403 FORBIDDEN, la aplicación en realidad se está estrellando!

Entonces, ¿por qué no estamos viendo un 500 `INTERNAL_SERVER_ERROR`, que sería más apropiado para un servidor que se estrella?

Momento de aprendizaje: Spring Security y manejo de errores.

Nuestro Controlador está devolviendo 403 FORBIDDEN en lugar de un 500 `INTERNAL_SERVER_ERROR` porque Spring Security está implementando automáticamente una mejor práctica sobre cómo se manejan los errores por Spring Web.

Es importante entender que cualquier información devuelta por nuestra aplicación podría ser útil para un actor malicioso que intente violar la seguridad de nuestra aplicación. Por ejemplo: conocimiento sobre acciones que causan que nuestra aplicación se estrelle, un 500 `INTERNAL_SERVER_ERROR`.

Para evitar "filtrar" información sobre nuestra aplicación, Spring Security ha configurado Spring Web para devolver un genérico 403 FORBIDDEN en la mayoría de las condiciones de error. Si casi todo resulta en una respuesta 403 FORBIDDEN, un atacante realmente no sabe qué está pasando.

Ahora que entendemos qué está sucediendo, hagamos que el Controlador maneje correctamente esta situación.

Actualiza el endpoint POST en el Controlador.

Una vez más usaremos el *Principal* proporcionado para asegurar que se guarde el propietario correcto con la nueva `CashCard`.

```
1 @PostMapping
2 private ResponseEntity<Void> createCashCard(@RequestBody CashCard newCashCardRequest,
3     UriComponentsBuilder ucb, Principal principal) {
4     CashCard cashCardWithOwner = new CashCard(null, newCashCardRequest.amount(),
5         principal.getName());
6     CashCard savedCashCard = cashCardRepository.save(cashCardWithOwner);
7     ...
8 }
```

Ejecuta las pruebas.

¡Afortunadamente, todo pasa otra vez!

```
1 CashCardApplicationTests > shouldCreateANewCashCard() PASSED
2 ...
3 BUILD SUCCESSFUL in 7s
```


Ahora solo el *Principal* autenticado y autorizado se usa para crear una **CashCard**.

¡Seguridad para ganar!

16.11 Acerca de CSRF

Como aprendimos en la lección acompañante, la protección contra Falsificación de Solicitudes entre Sitios (CSRF, o "sea-surf") es un aspecto importante de las APIs basadas en HTTP utilizadas por aplicaciones web.

Sin embargo, hemos desactivado CSRF mediante el código:

```
1 csrf.disable()
```

en

```
1 SecurityConfig.filterChain
```

¿Por qué hemos desactivado CSRF?

Con el propósito de nuestra API de Tarjetas Familiares Cash, seguiremos la guía del equipo de Spring Security respecto a clientes no basados en navegadores:

¿Cuándo deberías usar la protección CSRF? Nuestra recomendación es usar protección CSRF para cualquier solicitud que podría ser procesada por un navegador por usuarios normales. Si solo estás creando un servicio que es usado por clientes no basados en navegadores, probablemente querrás desactivar la protección CSRF.

Si te gustaría añadir seguridad CSRF a nuestra aplicación, por favor revisa las opciones de soporte de pruebas a continuación.

- Ejemplos de pruebas de CSRF con **MockMvc**.
- Ejemplos de pruebas de CSRF con **WebTestClient**.
- Una descripción del Patrón de Cookie de Doble Envío.

16.12 Resumen

En este laboratorio aprendiste cómo usar Spring Security para asegurar que solo los usuarios autenticados y autorizados tengan acceso a la API de Tarjetas Familiares Cash. Además, seguiste las mejores prácticas en las capas de Controlador y Repositorio de nuestra aplicación para asegurar que solo los usuarios correctos tengan acceso a sus (y solo sus) datos de Tarjeta Cash. También aprendiste que Spring Security no solo protege nuestras solicitudes, sino que también modifica el manejo de errores de nuestra API para evitar "filtrar" información sobre fallos y otras operaciones internas que ocurren en nuestra aplicación.

17 Implementación de PUT

17.1 Introducción

Hasta ahora podemos crear y recuperar Tarjetas Cash. ¡El siguiente paso lógico es la capacidad de ajustar el saldo de una tarjeta! Cómo implementar la operación de Actualización en una API RESTful es un tema candente, y eso es lo que abordaremos en esta lección.

Cuando decimos "ajustar el saldo" de una Tarjeta Cash, lo que realmente queremos decir es actualizar la cantidad en un registro existente de la base de datos. Esto implicará:

- Crear un nuevo endpoint para recibir una solicitud HTTP con un verbo, URI y cuerpo.
- Devolver una respuesta adecuada desde el endpoint para condiciones de éxito y error.

Ya estamos familiarizados con el verbo HTTP POST, que usamos para el endpoint de Creación. Ahora hablemos sobre HTTP PUT y PATCH, y cómo los tres están relacionados.

17.2 PUT y PATCH

Tanto PUT como PATCH se pueden usar para actualizar, pero funcionan de diferentes maneras. Esencialmente, PUT significa "crear o reemplazar el registro completo", mientras que PATCH significa "actualizar solo algunos campos del registro existente"—en otras palabras, una actualización parcial.

¿Por qué querías hacer una actualización parcial? Las actualizaciones parciales liberan al cliente de tener que cargar el registro completo y luego transmitirlo de vuelta al servidor. Si el registro es lo suficientemente grande, esto puede tener un impacto no trivial en el rendimiento.

Para nuestra aplicación, decidiremos no implementar una actualización parcial.

17.3 PUT y POST

Hay algo que no te contamos cuando escribimos el endpoint de Creación: ¡El estándar HTTP no especifica si el verbo POST o PUT es preferido para una operación de Creación! Esto es relevante porque usaremos el verbo PUT para nuestro endpoint de Actualización, por lo que necesitamos decidir si nuestra API soportará usar PUT para crear o actualizar un recurso.

Existen diferentes maneras de ver la relación entre las operaciones de Creación y Actualización y cómo se implementan en REST usando verbos HTTP. En las siguientes secciones, lo discutimos desde tres puntos de vista diferentes, luego resumimos lo que hemos cubierto hasta ahora, en una tabla.

El punto importante a tomar de las siguientes secciones no es memorizar todos los detalles, sino simplemente darse cuenta de que hay muchas opciones diferentes, y que nosotros,

como autores de la API de Tarjetas Cash, estamos tomando decisiones conscientes sobre cómo implementar REST.

17.4 Claves surrogadas y naturales

¿Por qué querríamos usar una operación PUT para crear un recurso? Esto tiene que ver con la definición HTTP de los dos verbos. @ La diferencia es sutil. @ Expliquémoslo comparando dos sistemas diferentes: nuestra API de Tarjetas Cash y otra API que introduciremos con fines explicativos, llamada la API de Facturas. @ La API de Facturas acepta el Número de Factura como el identificador único. @ Esto es un ejemplo de usar una Clave Natural (suministrada por el cliente a la API) en lugar de una Clave Surrogada (generalmente generada por el servidor, que es lo que estamos haciendo en nuestra API de Tarjetas Cash).

La diferencia importante es si la URI (que incluye el ID del recurso) necesita ser generada por el servidor o no. @ Aquí está cómo PUT y POST lo consideran:

- Si necesitas que el servidor devuelva la URI del recurso creado (o los datos que usas para construir la URI), entonces debes usar POST. Esto es el caso de nuestra API de Tarjetas Cash: Para crear una Tarjeta Cash, proporcionamos el endpoint POST /cashcards. @ La URI real de la Tarjeta Cash creada depende del ID generado y es proporcionada por el servidor, por ejemplo, /cashcards/101 si el ID de la tarjeta creada es 101.
- Alternativamente, cuando la URI del recurso es conocida en el momento de la creación (como es el caso en nuestro ejemplo de la API de Facturas), puedes usar PUT. Para la API de Facturas, podríamos escribir un endpoint de Creación que acepte solicitudes como PUT /invoice/1234-567. @ La llamada de Lectura correspondiente usaría la misma URI exacta: GET /invoice/1234-567.

17.5 Recursos y sub-recursos

Otra forma de ver la diferencia es en términos de URIs y colecciones de sub-recursos. @ Este es el lenguaje usado por la documentación de HTTP, por lo que es bueno estar familiarizado con él. @ Siguiendo los ejemplos anteriores, encontraríamos:

- POST crea un sub-recurso (recurso hijo) bajo (después de), o dentro de la URI de la solicitud. @ Esto es lo que hace la API de Tarjetas Cash: El cliente llama al endpoint de Creación en POST /cashcards, pero la URI real del recurso creado contiene un ID generado al final: /cashcards/101.
- PUT crea o reemplaza (actualiza) un recurso en una URI de solicitud específica. @ Para el ejemplo de /invoice anterior, el endpoint de Creación sería PUT /invoice/1234-567, y la URI del recurso creado sería la misma que la URI enviada en la solicitud PUT.

17.6 Cuerpo de respuesta y código de estado

Relacionado con decidir si permitir que PUT cree objetos, necesitas decidir qué código de estado y cuerpo de respuesta deberían ser. @ Dos opciones diferentes son:

- Devolver 201 CREATED (si creaste el objeto), o 200 OK (si reemplazaste un objeto existente). @ En este caso, se recomienda devolver el objeto en el cuerpo de la respuesta. @ Esto es útil si el servidor añadió datos al objeto (por ejemplo, si el servidor registra la fecha de creación).
- O devolver 204 NO_CONTENT, y un cuerpo de respuesta vacío. @ La justificación en este caso es que, dado que un PUT simplemente coloca un objeto en la URI en la solicitud, el cliente no necesita ninguna información de vuelta: sabe que el objeto en la solicitud ha sido guardado, verbatim, en el servidor.

17.7 POST, PUT, PATCH y operaciones CRUD - Resumen

Las secciones anteriores se pueden resumir usando la siguiente tabla:

Método HTTP	Operación	Definición de URI del recurso	¿Qué hace?	Código de estado	Cuerpo de respuesta
POST	Crear	El servidor genera y devuelve la URI	Crea un sub-recurso ("bajo." "dentro" de la URI pasada)	201 CREA-TED	El recurso creado
PUT	Crear	El cliente suministra la URI	Crea un recurso (en la URI de la solicitud)	201 CREA-TED	El recurso creado
PUT	Actualizar	El cliente suministra la URI	Reemplaza el recurso: El registro completo es reemplazado por el objeto en la solicitud	204 NO_CONTENT	(vacío)
PATCH	Actualizar	El cliente suministra la URI	Actualización parcial: modifica solo los campos incluidos en la solicitud en el registro existente	200 OK	El recurso actualizado

En la API de Tarjetas Cash, no necesitamos permitir que PUT cree recursos. @ Tampoco necesitamos añadir datos del lado del servidor para una operación de Actualización, ni permitir actualizaciones parciales. @ Así, nuestro endpoint PUT se limita a la fila 3 de la tabla anterior.

Las filas en negrita en la tabla anterior son implementadas por la API de Tarjetas Cash. @ Las que no están en negrita no lo están.

17.8 Seguridad

Otra decisión que tomaremos es cómo aplicar la lógica de seguridad a la nueva operación de Actualización.

Recuerda de la lección de Seguridad Simple que decidimos devolver 404 NOT FOUND para solicitudes GET en dos casos: IDs inexistentes y intentos de acceso a tarjetas para las que el usuario no está autorizado. @ Usaremos la misma estrategia para el endpoint de Actualización, por razones similares (que detallaremos más en el laboratorio).

17.9 Nuestras decisiones de API

¡Vaya, eso fue un montón de decisiones! Para resumir, decidimos:

- PUT no soportará crear una Tarjeta Cash.
- Nuestro nuevo endpoint de Actualización (que construiremos en el próximo laboratorio):
 - usará el verbo PUT.
 - acepta una Tarjeta Cash y reemplaza la Tarjeta Cash existente con ella.
 - en caso de éxito, devolverá 204 NO_CONTENT con un cuerpo vacío.
 - devolverá un 404 NOT FOUND para una actualización no autorizada, así como intentos de actualizar IDs inexistentes.

18 Laboratorio: Implementación de PUT

18.1 Visión general

Como se discutió en la lección asociada, permitiremos a los usuarios de la API REST de Tarjetas Familiares Cash actualizar una Tarjeta Cash usando un HTTP PUT.

¡Implementemos esta funcionalidad ahora!

18.2 Escribir la prueba primero

Como hemos hecho en casi todos los laboratorios, comencemos con una prueba.

¿Cuál es la funcionalidad que queremos que tenga nuestra aplicación? ¿Cómo queremos que se comporte nuestra aplicación?

Definamos esto ahora y trabajemos hacia la satisfacción de nuestras aspiraciones.

Escribe la prueba de actualización.

Usaremos PUT para actualizar Tarjetas Cash.

Ten en cuenta que esperaremos una respuesta 204 NO_CONTENT en lugar de un 200 OK. El 204 indica que la acción se realizó con éxito y no se necesita más acción por parte del llamador.

Edita `src/test/java/example/cashcard/CashCardApplicationTests.java` y agrega la siguiente prueba, que actualiza la Tarjeta Cash 99 y establece su cantidad en 19.99.

No olvides agregar las dos nuevas importaciones.

```
1 import org.springframework.http.HttpEntity;
2 import org.springframework.http.HttpMethod;
3 ...
4 @Test
5 @DirtiesContext
6 void shouldUpdateAnExistingCashCard() {
7     CashCard cashCardUpdate = new CashCard(null, 19.99, null);
8     HttpEntity<CashCard> request = new HttpEntity<>(cashCardUpdate);
9     ResponseEntity<Void> response = restTemplate
10         .withBasicAuth("sarah1", "abc123")
11         .exchange("/cashcards/99", HttpMethod.PUT, request, Void.class);
12     assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NO_CONTENT);
13 }
```

Momento de aprendizaje: ¿Qué pasa con `restTemplate.exchange()`?

¿Notaste que no estamos usando `RestTemplate` de la misma manera que en nuestras pruebas anteriores?

Todas las demás pruebas usan métodos `RestTemplate.xyzForEntity` como `getForEntity()` y `postForEntity()`.

Entonces, ¿por qué no seguimos el mismo patrón de utilizar `putForEntity()`?

Respuesta: `putForEntity()` no existe! Lee más sobre esto aquí en el issue de GitHub sobre el tema.

Afortunadamente, `RestTemplate` soporta múltiples maneras de interactuar con APIs REST, como `RestTemplate.exchange()`.

Aprendamos sobre `RestTemplate.exchange()` ahora.

Entiende `RestTemplate.exchange()`.

Entendamos más sobre lo que está sucediendo.

El método `exchange()` es una versión más general de los métodos `xyzForEntity()` que hemos usado en otras pruebas: `exchange()` requiere que el verbo y la entidad de solicitud (el cuerpo de la solicitud) se suministren como parámetros.

Usando `getForEntity()` como ejemplo, puedes imaginar que las siguientes dos líneas de código logran el mismo objetivo:

```
1 .exchange("/cashcards/99", HttpMethod.GET, new HttpEntity(null), String.class);
```

La línea anterior es funcionalmente equivalente a la siguiente línea:

```
1 .getForEntity("/cashcards/99", String.class);
```

Ahora expliquemos el código de la prueba.

Primero creamos la `HttpEntity` que el método `exchange()` necesita:

```
1 HttpEntity<CashCard> request = new HttpEntity<>(cashCardUpdate);
```

Luego llamamos a `exchange()`, que envía una solicitud PUT para el ID objetivo de 99 y los datos de Tarjeta Cash actualizados:

```
1 .exchange("/cashcards/99", HttpMethod.PUT, request, Void.class);
```

Ejecuta las pruebas.

Es hora de ver cómo fallan nuestras pruebas por las razones correctas.

¿Por qué fallará nuestra nueva prueba?

Ten en cuenta que siempre usaremos `./gradlew test` para ejecutar nuestras pruebas.

```
1 [~/exercises] $ ./gradlew test
2 ...
3 CashCardApplicationTests > shouldUpdateAnExistingCashCard() FAILED
4 org.opentest4j.AssertionFailedError:
5   expected: 204 NO_CONTENT
6   but was: 403 FORBIDDEN
7 ...
8 BUILD FAILED in 6s
```

Respuesta: ¡Aún no hemos implementado un manejador de método de solicitud PUT!

Como puedes ver, la prueba falló con un código de respuesta 403 FORBIDDEN.

¡Sin un endpoint de Controlador, esta llamada PUT está prohibida! Spring Security manejó automáticamente este escenario para nosotros. ¡Genial!

A continuación, implementemos el endpoint del Controlador.

18.3 Implementar @PutMapping en el Controlador

Siguiendo el patrón que hemos usado hasta ahora en nuestro Controlador, implementemos el endpoint PUT en nuestro `CashCardController`.

Agrega un `@PutMapping` mínimo.

Edita `src/main/java/example/cashcard/CashCardController.java` y agrega el endpoint PUT.

```
1 @PutMapping("/{requestedId}")
2 private ResponseEntity<Void> putCashCard(@PathVariable Long requestedId, @RequestBody
   CashCard cashCardUpdate) {
3     // solo devuelve 204 NO_CONTENT por ahora.
4     return ResponseEntity.noContent().build();
5 }
```

Este endpoint del Controlador es bastante autoexplicativo:

- El `@PutMapping` soporta el verbo PUT y suministra el `requestedId` objetivo.
- El `@RequestBody` contiene los datos de Tarjeta Cash actualizados.
- Devuelve un código de respuesta HTTP 204 NO_CONTENT por ahora, solo para comenzar.

Ejecuta las pruebas.

¿Qué crees que pasará?

Ejecútalas ahora.

```
1 ...
2 CashCardApplicationTests > shouldUpdateAnExistingCashCard() PASSED
3 ...
4 BUILD SUCCESSFUL in 6s
```

¡Pasan!

Pero, ¿no es eso muy satisfactorio, verdad?

¡No hemos actualizado la Tarjeta Cash!

Hagámoslo a continuación.

Mejora la prueba para verificar una actualización exitosa.

Similar a otras verificaciones que hemos realizado en nuestra suite de pruebas, afirmemos que la actualización fue exitosa.

```
1 void shouldUpdateAnExistingCashCard() {
2     ...
3     assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NO_CONTENT);
4
5     ResponseEntity<String> getResponse = restTemplate
6         .withBasicAuth("sarah1", "abc123")
7         .getForEntity("/cashcards/99", String.class);
8     assertThat(getResponse.getStatusCode()).isEqualTo(HttpStatus.OK);
9     DocumentContext documentContext = JsonPath.parse(getResponse.getBody());
10    Number id = documentContext.read("$.id");
```



```

11 Double amount = documentContext.read("$.amount");
12 assertThat(id).isEqualTo(99);
13 assertThat(amount).isEqualTo(19.99);
14 }

```

Entiende las actualizaciones de la prueba.

```

1 ResponseEntity<String> getResponse = restTemplate
2     .withBasicAuth("sarah1", "abc123")
3     .getForEntity("/cashcards/99", String.class);

```

Aquí, volvemos a obtener la Tarjeta Cash 99, para verificar que fue actualizada.

```

1 ...
2 assertThat(id).isEqualTo(99);
3 assertThat(amount).isEqualTo(19.99);

```

A continuación, afirmamos que hemos recuperado la Tarjeta Cash correcta – 99 – y que su cantidad se actualizó con éxito a 19.99.

Ejecuta las pruebas.

Nuestras expectativas son legítimas, pero no hemos actualizado el `CashCardController` para que coincida.

La prueba debería fallar con mensajes de error valiosos, ¿verdad?

Ejecuta las pruebas ahora.

```

1 CashCardApplicationTests > shouldUpdateAnExistingCashCard() FAILED
2 org.opentest4j.AssertionFailedError:
3   expected: 19.99
4   but was: 123.45

```

¡Excelente! Esperamos una cantidad de 19.99, pero sin cambios seguimos devolviendo 123.45.

Actualicemos el Controlador.

Actualiza `CashCardController` para realizar la actualización de datos.

Al igual que con nuestros otros métodos manejadores, asegurémonos de garantizar que solo el propietario de la Tarjeta Cash pueda realizar las actualizaciones – el *Principal* conectado debe ser el mismo que el propietario de la Tarjeta Cash:

```

1 @PutMapping("/{requestedId}")
2 private ResponseEntity<Void> putCashCard(@PathVariable Long requestedId, @RequestBody
3     CashCard cashCardUpdate, Principal principal) {
4     CashCard cashCard = cashCardRepository.findByIdAndOwner(requestedId,
5         principal.getName());
6     CashCard updatedCashCard = new CashCard(cashCard.id(), cashCardUpdate.amount(),
7         principal.getName());
8     cashCardRepository.save(updatedCashCard);
9     return ResponseEntity.noContent().build();
10 }

```

Entiende las actualizaciones de `CashCardController`.

Estamos siguiendo un patrón similar al del endpoint `@PostMapping createCashCard()`.

```

1 @PutMapping("/{requestedId}")
2 private ResponseEntity<Void> putCashCard(@PathVariable Long requestedId, @RequestBody
    CashCard cashCardUpdate, Principal principal) {

```

Hemos agregado el *Principal* como argumento del método, proporcionado automáticamente por Spring Security.

¡Gracias una vez más, Spring Security!

```

1 cashCardRepository.findByIdAndOwner(requestedId, principal.getName());

```

Aquí, limitamos la recuperación de la Tarjeta Cash al `requestedId` enviado y al *Principal* (proporcionado por Spring Security) para asegurar que solo el propietario autenticado y autorizado pueda actualizar esta Tarjeta Cash.

```

1 CashCard updatedCashCard = new CashCard(cashCard.id(), cashCardUpdate.amount(),
    principal.getName());
2 cashCardRepository.save(updatedCashCard);

```

Finalmente, construimos una Tarjeta Cash con valores actualizados y la guardamos.

¡Eso fue mucho! Ejecutemos las pruebas y evaluemos dónde estamos.

Ejecuta las pruebas.

```

1 ...
2 CashCardApplicationTests > shouldUpdateAnExistingCashCard() PASSED
3 ...
4 BUILD SUCCESSFUL in 6s

```

¡Pasan!

Hemos implementado con éxito la actualización de una Tarjeta Cash.

Ahora dirijamos nuestra atención a la seguridad. @ Específicamente, ¿qué pasa si intentamos actualizar una Tarjeta Cash que no poseemos o una que no existe?

Probemos esos escenarios a continuación.

18.4 Pruebas adicionales e influencia de Spring Security

Tenemos dos escenarios restantes que probar: el caso en que un usuario intenta actualizar una Tarjeta Cash que no existe; y el caso en que un usuario intenta actualizar una Tarjeta Cash que no posee.

Caso 1: Intento de actualizar una Tarjeta Cash que no existe.

Comencemos con el primero de esos dos casos. @ Comienza agregando una prueba.

Intenta actualizar una Tarjeta Cash que no existe.

Agrega la siguiente prueba a `CashCardApplicationTests`.

```

1 @Test
2 void shouldNotUpdateACashCardThatDoesNotExist() {
3     CashCard unknownCard = new CashCard(null, 19.99, null);
4     HttpEntity<CashCard> request = new HttpEntity<>(unknownCard);
5     ResponseEntity<Void> response = restTemplate

```

```

6         .withBasicAuth("sarah1", "abc123")
7         .exchange("/cashcards/99999", HttpMethod.PUT, request, Void.class);
8         assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
9     }

```

Aquí intentaremos actualizar una Tarjeta Cash con ID 99999, que no existe.

La prueba debería esperar un error genérico 404 NOT_FOUND.

Ejecuta las pruebas.

Antes de ejecutar la prueba, adivina cuál será el resultado.

¿OK, hiciste tu suposición? Ahora, ejecuta la prueba para verificar tu hipótesis.

```

1 ...
2 CashCardApplicationTests > shouldNotUpdateACashCardThatDoesNotExist() FAILED
3 org.opentest4j.AssertionFailedError:
4     expected: 404 NOT_FOUND
5     but was: 403 FORBIDDEN

```

Bueno, eso es... interesante. ¿Por qué obtuvimos un 403 FORBIDDEN?

Antes de ejecutarlas de nuevo, editemos `build.gradle` para habilitar salida adicional de pruebas.

```

1 test {
2     testLogging {
3         ...
4         // Cambia a true para una salida de prueba mas detallada
5         showStandardStreams = true
6     }
7 }

```

Después de volver a ejecutar las pruebas, busca en la salida lo siguiente:

```

1 CashCardApplicationTests > shouldNotUpdateACashCardThatDoesNotExist() STANDARD_OUT
2 ...
3 java.lang.NullPointerException: Cannot invoke "example.cashcard.CashCard.id()"
   because "cashCard" is null

```

¡Una `NullPointerException`! ¿Por qué una `NullPointerException`?

Mirando `CashCardController.putCashCard`, podemos ver que si no encontramos la `cashCard`, las llamadas al método de `cashCard` resultarán en una `NullPointerException`.@
Eso tiene sentido.

Pero, ¿por qué una `NullPointerException` lanzada en nuestro Controlador resulta en un 403 FORBIDDEN en lugar de un 500 INTERNAL_SERVER_ERROR, dado que el servidor “se estrelló”?

Recuerda: Aprendimos en el módulo de Spring Security que para evitar “filtrar” información sobre nuestra aplicación, Spring Security ha configurado Spring Web para devolver un genérico 403 FORBIDDEN en la mayoría de las condiciones de error. ¡Gracias otra vez, Spring Security!

Ahora que entendemos qué está pasando, arreglemoslo.

No te estrelles.

Aunque estamos agradecidos con Spring Security, nuestra aplicación no debería estrellarse: no deberíamos permitir que nuestro código lance una `NullPointerException`.@ En lugar de eso, deberíamos manejar la condición cuando `cashCard == null`, y devolver una respuesta HTTP genérica 404 NOT_FOUND.

Actualiza `CashCardController.putCashCard` para devolver 404 NOT_FOUND si no se encuentra una Tarjeta Cash existente.

```

1 @PostMapping("/{requestedId}")
2 private ResponseEntity<Void> putCashCard(@PathVariable Long requestedId, @RequestBody
   CashCard cashCardUpdate, Principal principal) {
3     CashCard cashCard = cashCardRepository.findByIdAndOwner(requestedId,
   principal.getName());
4     if (cashCard != null) {
5         CashCard updatedCashCard = new CashCard(cashCard.id(),
   cashCardUpdate.amount(), principal.getName());
6         cashCardRepository.save(updatedCashCard);
7         return ResponseEntity.noContent().build();
8     }
9     return ResponseEntity.notFound().build();
10 }

```

Ejecuta las pruebas.

Nuestras pruebas pasan ahora que devolvemos un 404 cuando no se encuentra una Tarjeta Cash.

```

1 [~/exercises] $ ./gradlew test
2 ...
3 BUILD SUCCESSFUL in 6s

```

Caso 2: Intento de actualizar una Tarjeta Cash propiedad de alguien más.

Escribe una prueba para el caso restante.

Escribamos una prueba que afirme que `sarah1` no está permitido actualizar una de las Tarjetas Cash de `kumar2`.

```

1 @Test
2 void shouldNotUpdateACashCardThatIsOwnedBySomeoneElse() {
3     CashCard kumarsCard = new CashCard(null, 333.33, null);
4     HttpEntity<CashCard> request = new HttpEntity<>(kumarsCard);
5     ResponseEntity<Void> response = restTemplate
6         .withBasicAuth("sarah1", "abc123")
7         .exchange("/cashcards/102", HttpMethod.PUT, request, Void.class);
8     assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
9 }

```

Haz una hipótesis sobre qué será el resultado de la prueba, luego verifica la hipótesis ejecutando la prueba.

```

1 [~/exercises] $ ./gradlew test
2 ...
3 CashCardApplicationTests > shouldNotUpdateACashCardThatIsOwnedBySomeoneElse() PASSED

```

¿Fue correcta tu hipótesis?

La prueba pasó porque la modificación al Controlador que hicimos en el paso anterior —específicamente, verificar un resultado nulo al recuperar la Tarjeta Cash existente— también hace que esta prueba pase. Pero para estar seguros de este diagnóstico, hagamos un experimento.

Verifica la razón del éxito de la prueba.

¡Adelante! Comenta los cambios en el Controlador y vuelve a ejecutar las pruebas:

```

1 @PutMapping("/{requestedId}")
2 private ResponseEntity<Void> putCashCard(@PathVariable Long requestedId, @RequestBody
   CashCard cashCardUpdate, Principal principal) {
3     CashCard cashCard = cashCardRepository.findByIdAndOwner(requestedId,
   principal.getName());
4     // if (null != cashCard) {
5         CashCard updatedCashCard = new CashCard(cashCard.id(),
   cashCardUpdate.amount(), principal.getName());
6         cashCardRepository.save(updatedCashCard);
7         return ResponseEntity.noContent().build();
8     // }
9     // return ResponseEntity.notFound().build();
10 }

```

```

1 [~/exercises] $ ./gradlew test
2 ...
3 CashCardApplicationTests > shouldNotUpdateACashCardThatIsOwnedBySomeoneElse() FAILED
4 org.opentest4j.AssertionFailedError:
5 expected: 404 NOT_FOUND
6 but was: 403 FORBIDDEN
7 ...
8 CashCardApplicationTests > shouldNotUpdateACashCardThatDoesNotExist() FAILED
9 org.opentest4j.AssertionFailedError:
10 expected: 404 NOT_FOUND
11 but was: 403 FORBIDDEN

```

Estamos de vuelta al estado de colapso: Ambas pruebas resultan en 403 FORBIDDEN debido a la `NullPointerException` subyacente.

Deshace tus cambios experimentales.

Recuerda descomentar las tres líneas en `CashCardController.putCashCard`

```

1 @PutMapping("/{requestedId}")
2 private ResponseEntity<Void> putCashCard(@PathVariable Long requestedId, @RequestBody
   CashCard cashCardUpdate, Principal principal) {
3     CashCard cashCard = cashCardRepository.findByIdAndOwner(requestedId,
   principal.getName());
4     if (null != cashCard) {
5         CashCard updatedCashCard = new CashCard(cashCard.id(),
   cashCardUpdate.amount(), principal.getName());
6         cashCardRepository.save(updatedCashCard);
7         return ResponseEntity.noContent().build();
8     }
9     return ResponseEntity.notFound().build();
10 }

```

Momento de aprendizaje: Existe controversia.

En este punto podrías preguntarte: ¿Debería escribir dos pruebas diferentes que actúen exactamente igual con respecto a un solo cambio en el código de la aplicación? ¿Y si debería, por qué?

Una razón por la que podrías desear mantener las dos pruebas es que en algún momento futuro, alguien podría cambiar la implementación del controlador de alguna manera que cause que las dos pruebas actúen de manera diferente.

Después de todo, estamos probando dos escenarios diferentes:

- Un registro de Tarjeta Cash con `id=99999` existe en la base de datos.
- Un registro de Tarjeta Cash con `id=99999` no existe en la base de datos.

La verdad es que si planteas esto como tema de conversación, podrías obtener diferentes respuestas de diferentes personas bajo diferentes circunstancias. @ Aunque eso podría ser interesante de discutir, como desarrolladores de la API de Tarjetas Familiares Cash, necesitamos seguir adelante.

Reconociendo que las opiniones pueden diferir, tomaremos la decisión de dejar ambas pruebas en su lugar.

18.5 Refactorizar el código del Controlador

Reforcemos tu uso del ciclo de desarrollo Red, Green, Refactor.

Acabamos de completar varias pruebas enfocadas en actualizar una Tarjeta Cash existente.

¿Tenemos alguna oportunidad de simplificar, reducir duplicación o de otra manera refactorizar nuestro código sin cambiar el comportamiento?

Continúa para abordar varias oportunidades de refactorización.

Simplifica el código.

Elimina el `Optional`.

Esto podría ser controvertido: Dado que no estamos aprovechando las características de `Optional` en `CashCardController.findById`, y ningún otro método del Controlador usa un `Optional`, simplifiquemos nuestro código eliminándolo.

Edita `CashCardController.findById` para eliminar el uso de `Optional`:

```
1 // elimina la importacion no utilizada de Optional si esta presente
2 // import java.util.Optional;
3
4 @GetMapping("/{requestedId}")
5 private ResponseEntity<CashCard> findById(@PathVariable Long requestedId, Principal
    principal) {
6     CashCard cashCard = cashCardRepository.findByIdAndOwner(requestedId,
        principal.getName());
7     if (cashCard != null) {
8         return ResponseEntity.ok(cashCard);
9     } else {
10        return ResponseEntity.notFound().build();
    }
```

```

11     }
12 }

```

Ejecuta las pruebas.

Como no hemos cambiado ninguna funcionalidad, las pruebas seguirán pasando.

```

1 ./gradlew test
2 ...
3 BUILD SUCCESSFUL in 6s

```

Reduce la duplicación de código.

Nota que tanto `CashCardController.findById` como `CashCardController.putCashCard` tienen código casi idéntico que recupera una Tarjeta Cash objetivo desde el `CashCardRepository` usando información de una `CashCard` y *Principal*.

Reduzcamos la duplicación de código extrayendo un método auxiliar llamado `findCashCard`, y utilizándolo en ambos `.findById` y `.putCashCard`.

Esto nos permitirá actualizar cómo recuperamos una Tarjeta Cash en un solo lugar a medida que el Controlador cambia con el tiempo.

Crea un método compartido `findCashCard`.

Crea un nuevo método en `CashCardController` llamado `findCashCard`, usando funcionalidad que hemos escrito antes:

```

1 private CashCard findCashCard(Long requestedId, Principal principal) {
2     return cashCardRepository.findByIdAndOwner(requestedId, principal.getName());
3 }

```

Actualiza `CashCardController.findById` y vuelve a ejecutar las pruebas.

A continuación, utiliza el nuevo método `findCashCard` en `CashCardController.findById`.

```

1 @GetMapping("/{requestedId}")
2 private ResponseEntity<CashCard> findById(@PathVariable Long requestedId, Principal
    principal) {
3     CashCard cashCard = findCashCard(requestedId, principal);
4     ...

```

No ha cambiado ninguna funcionalidad, por lo que no deberían fallar pruebas cuando volvamos a ejecutarlas.

```

1 ./gradlew test
2 ...
3 BUILD SUCCESSFUL in 6s
4 \end{lstlisting}
5
6 Actualiza \texttt{CashCardController.putCashCard} y vuelve a ejecutar las pruebas.
7
8 Similar al paso anterior, utiliza el nuevo método \texttt{findCashCard} en
    \texttt{CashCardController.putCashCard}.
9
10 \begin{lstlisting}[language=Java]
11 @PostMapping("/{requestedId}")
12 private ResponseEntity<Void> putCashCard(@PathVariable Long requestedId, @RequestBody
    CashCard cashCardUpdate, Principal principal) {

```

```
13 CashCard cashCard = findCashCard(requestedId, principal);  
14 ...
```

Como con la otra refactorización que hemos realizado, no ha cambiado ninguna funcionalidad, por lo que no deberían fallar pruebas cuando volvamos a ejecutarlas.

```
1 ./gradlew test  
2 ...  
3 BUILD SUCCESSFUL in 6s
```

¡Mira eso! Hemos refactorizado nuestro código simplificándolo y también reduciendo la duplicación de código. ¡Excelente!

18.6 Resumen

En este laboratorio, aprendiste cómo implementar un endpoint HTTP PUT que permite a un propietario autenticado y autorizado actualizar su Tarjeta Cash.

También aprendiste cómo Spring Security maneja automáticamente el manejo de errores por Spring Web para asegurar que no se revele accidentalmente información sensible a la seguridad cuando un Controlador encuentra un error.

Además, reforzaste tu comprensión y utilización del ciclo de desarrollo Red, Green, Refactor refactorizando nuestro código del Controlador sin cambiar su funcionalidad.

19 Implementación de DELETE

En esta lección, implementaremos la última de las cuatro operaciones CRUD: ¡Eliminar! Para ahora, deberías estar familiarizado con la pregunta que debemos hacer primero: ¿Cuál es la especificación de datos de la API para el endpoint de Eliminar? La especificación incluye los detalles de la Solicitud y la Respuesta.

Con el riesgo de spoilear el resultado, esta es la especificación que definiremos:

19.1 Solicitud

- Verbo: DELETE
- URI: /cashcards/id
- Cuerpo: (vacío)

19.2 Respuesta

- Código de estado: 204 NO_CONTENT
- Cuerpo: (vacío)

Devolveremos el código de estado 204 NO_CONTENT para una eliminación exitosa, pero hay casos adicionales:

Código de respuesta	Caso de uso
204 NO_CONTENT	El registro existe, y el Principal está autorizado, y el registro fue eliminado correctamente.
404 NOT_FOUND	El registro no existe (se envió un ID inexistente).
404 NOT_FOUND	El registro existe pero el Principal no es el propietario.

¿Por qué devolvemos 404 para los casos “ID no existe” y “no autorizado para acceder a este ID”? Para no “filtrar” información: Si la API devolviera resultados diferentes para los dos casos, un usuario no autorizado podría descubrir IDs específicos a los que no está autorizado a acceder.

19.3 Opciones adicionales

Profundicemos en algunas opciones más que consideraremos al implementar una operación de Eliminación.

19.3.1. Eliminación dura y blanda

Entonces, ¿qué significa eliminar una Tarjeta Cash desde el punto de vista de una base de datos? Similar a cómo decidimos que nuestra operación de Actualización significa “reemplazar el registro existente completo” (en lugar de soportar actualizaciones parciales), necesitamos decidir qué pasa con los recursos cuando se eliminan.

Una opción simple, llamada eliminación dura, es eliminar el registro de la base de datos. Con una eliminación dura, se pierde para siempre. Entonces, ¿qué podemos hacer si necesitamos datos que existieron antes de su eliminación?

Una alternativa es la eliminación blanda, que funciona marcando los registros como “eliminados” en la base de datos (de modo que se retienen, pero se marcan como eliminados). Por ejemplo, podemos introducir una columna booleana `IS_DELETED` o un timestamp `DELETED_DATE` y luego establecer ese valor en lugar de eliminar completamente el registro al borrar la(s) fila(s) de la base de datos. Con una eliminación blanda, también necesitamos cambiar cómo los Repositorios interactúan con la base de datos. Por ejemplo, un repositorio necesita respetar la columna “eliminado” y excluir registros marcados como eliminados de las solicitudes de Lectura.

19.3.2. Rastro de auditoría y archivo

Al trabajar con bases de datos, encontrarás que a menudo hay un requisito para mantener un registro de modificaciones a los registros de datos. Por ejemplo:

- Un representante de servicio al cliente podría necesitar saber cuándo un cliente eliminó su Tarjeta Cash.
- Pueden existir regulaciones de cumplimiento de retención de datos que requieran que los datos eliminados se conserven por un cierto período de tiempo.

Si la Tarjeta Cash se elimina de forma dura, entonces necesitaríamos almacenar datos adicionales para poder responder a esta pregunta. Discutamos algunas maneras de registrar información histórica:

- Archivar (mover) los datos eliminados a una ubicación diferente.
- Agregar campos de auditoría al registro en sí. Por ejemplo, la columna `DELETED_DATE` que mencionamos anteriormente. Se pueden agregar campos de auditoría adicionales, por ejemplo `DELETED_BY_USER`. Nuevamente, esto no se limita a operaciones de Eliminación, sino también a Crear y Actualizar.

Las APIs que implementan eliminación blanda y campos de auditoría pueden devolver el estado del objeto en la respuesta, y el código de estado 200 OK. Entonces, ¿por qué elegimos usar 204 en lugar de 200? Porque el estado 204 `NO_CONTENT` implica que no hay cuerpo en la respuesta.

- Mantener un rastro de auditoría. El rastro de auditoría es un registro de todas las operaciones importantes realizadas en un registro. Puede contener no solo operaciones de Eliminación, sino también Crear y Actualizar.

La ventaja de un rastro de auditoría sobre los campos de auditoría es que un rastro registra todos los eventos, mientras que los campos de auditoría en el registro capturan solo la operación más reciente. Un rastro de auditoría puede almacenarse en una ubicación de base de datos diferente, o incluso en archivos de registro.

Vale la pena mencionar que una combinación de varias de las estrategias anteriores es posible. Aquí hay algunos ejemplos:

- Podríamos implementar eliminación blanda, luego tener un proceso separado que elimine de forma dura o archive los registros eliminados de forma blanda después de un cierto período de tiempo, como una vez por año.
- Podríamos implementar eliminación dura y archivar los registros eliminados.

En cualquiera de los casos anteriores, podríamos mantener un registro de auditoría de qué operaciones ocurrieron cuándo.

Finalmente, observa que incluso la especificación simple que hemos elegido no determina si implementamos eliminación dura o blanda. @ Tampoco determina si agregamos campos de auditoría o mantenemos un rastro de auditoría. @ Sin embargo, el hecho de que hayamos elegido devolver 204 NO_CONTENT implica que no está ocurriendo una eliminación blanda, ya que si lo estuviera, probablemente elegiríamos devolver 200 OK con el registro en el cuerpo.

20 Laboratorio: Implementación de DELETE

20.1 Visión general

En la lección asociada, aprendiste sobre la implementación de la operación de Eliminación en una API REST. En este laboratorio, implementaremos una eliminación dura en nuestra API de Tarjetas Cash, utilizando las especificaciones de API que definimos en la lección asociada.

Credenciales para el usuario de prueba Kumar Si has tomado laboratorios anteriores en este curso, notarás los siguientes cambios en el código base, que hemos realizado por ti, para hacer este laboratorio más fácil de entender y completar.

Hemos agregado credenciales para el usuario `kumar2` al bean `testOnlyUsers` en `src/main/java/example`.
¡Implementemos el endpoint de Eliminación!

20.2 Prueba del caso feliz

Comencemos con la ruta feliz más simple: eliminar con éxito una Tarjeta Cash que existe.

Necesitamos que el endpoint de Eliminación devuelva el código de estado 204 NO CONTENT.

Escribe la prueba.

Agrega el siguiente método de prueba a:

`src/test/java/example/cashcard/CashCardApplicationTests.java`:

```
1 @Test
2 @ DirtiesContext
3 void shouldDeleteAnExistingCashCard() {
4     ResponseEntity<Void> response = restTemplate
5         .withBasicAuth("sarah1", "abc123")
6         .exchange("/cashcards/99", HttpMethod.DELETE, null, Void.class);
7     assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NO_CONTENT);
8 }
```

Nota que hemos agregado la anotación `@DirtiesContext`. Añadiremos esta anotación a todas las pruebas que cambien los datos. Si no lo hacemos, estas pruebas podrían afectar el resultado de otras pruebas en el archivo.

¿Por qué no usar `RestTemplate.delete()`?

Nota que estamos usando `RestTemplate.exchange()` aunque `RestTemplate` suministra un método que parece que podríamos usar: `RestTemplate.delete()`. Sin embargo, veamos la firma:

```
1 public class RestTemplate ... {
2     public void delete(String url, Object... uriVariables)
```

Los otros métodos que hemos estado usando (como `getForEntity()` y `exchange()`) devuelven un `ResponseEntity`, pero `delete()` no. @ En cambio, es un método `void`. ¿Por qué es esto?

El framework Spring Web suministra el método `delete()` como una conveniencia, pero viene con algunas suposiciones:

- Una respuesta a una solicitud DELETE no tendrá cuerpo.
- Al cliente no debería importarle qué código de respuesta es a menos que sea un error, en cuyo caso lanzará una excepción.

Dadas esas suposiciones, no se necesita ningún valor de retorno de `delete()`.

Pero, la segunda suposición hace que `delete()` no sea adecuado para nosotros: ¡Necesitamos el `ResponseEntity` para afirmar el código de estado! Así que, no usaremos el método de conveniencia, sino que usemos el método más general: `exchange()`.

Ejecuta las pruebas.

Como siempre, usaremos `./gradlew test` para ejecutar las pruebas.

```
1 [~/exercises] $ ./gradlew test
2 ...
3 CashCardApplicationTests > shouldDeleteAnExistingCashCard() FAILED
4     org.opentest4j.AssertionFailedError:
5         expected: 204 NO_CONTENT
6         but was: 403 FORBIDDEN
```

La prueba falló porque la solicitud DELETE `/cashcards/99` devolvió un 403 FORBIDDEN.

En este punto probablemente esperabas este resultado: Spring Security devuelve una respuesta 403 para cualquier endpoint que no esté mapeado.

Necesitamos implementar el método del Controlador. ¡Hagámoslo!

Implementa el endpoint de Eliminación en el Controlador.

Agrega el siguiente método a la clase `CashCardController`:

```
1 @DeleteMapping("/{id}")
2 private ResponseEntity<Void> deleteCashCard(@PathVariable Long id) {
3     return ResponseEntity.noContent().build();
4 }
```

¿Qué pasará si ejecutamos las pruebas?

Ejecuta las pruebas.

```
1 [~/exercises] $ ./gradlew test
2 ...
3 BUILD SUCCESSFUL in 8s
```

¡Pasan!

¿Significa eso que hemos terminado? ¡Aún no!

No hemos escrito el código para eliminar realmente el elemento. @ Hagámoslo a continuación.

Escribiremos la prueba primero, por supuesto.

Prueba que realmente estamos eliminando la Tarjeta Cash.

Agrega las siguientes afirmaciones al método `shouldDeleteAnExistingCashCard()`:

```

1 @Test
2 @ DirtiesContext
3 void shouldDeleteAnExistingCashCard() {
4     ResponseEntity<Void> response = restTemplate
5         .withBasicAuth("sarah1", "abc123")
6         .exchange("/cashcards/99", HttpMethod.DELETE, null, Void.class);
7     assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NO_CONTENT);
8
9     // Agrega el siguiente código:
10    ResponseEntity<String> getResponse = restTemplate
11        .withBasicAuth("sarah1", "abc123")
12        .getForEntity("/cashcards/99", String.class);
13    assertThat(getResponse.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
14 }

```

Entiende el código de la prueba.

Queremos probar que la Tarjeta Cash eliminada realmente se eliminó, así que intentamos obtenerla con GET y afirmamos que el código de resultado es 404 NOT FOUND.

Ejecuta la prueba. ¿Pasa? ¡Por supuesto que no!

```

1 CashCardApplicationTests > shouldDeleteAnExistingCashCard() FAILED
2     org.opentest4j.AssertionFailedError:
3         expected: 404 NOT_FOUND
4         but was: 200 OK

```

¿Qué necesitamos hacer para que la prueba pase? ¡Escribir algo de código para eliminar el registro!

Vamos.

20.3 Implementa el endpoint DELETE

Ahora necesitamos escribir un método de Controlador que se llamará cuando enviemos una solicitud DELETE con la URI adecuada.

Agrega código al Controlador para eliminar el registro.

Cambia el método `CashCardController.deleteCashCard()`:

```

1 @DeleteMapping("/{id}")
2 private ResponseEntity<Void> deleteCashCard(@PathVariable Long id) {
3     cashCardRepository.deleteById(id); // Agrega esta línea
4     return ResponseEntity.noContent().build();
5 }

```

El cambio es directo:

- Usamos el `@DeleteMapping` con el parámetro "id", que Spring Web coincide con el parámetro del método id.

- `CashCardRepository` ya tiene el método que necesitamos: `deleteById()` (es heredado de `CrudRepository`).

Ejecuta las pruebas y observa cómo pasan!

```
1 [~/exercises] $ ./gradlew test
2 ...
3 BUILD SUCCESSFUL in 8s
```

¡Genial, qué hacer a continuación?

20.4 Caso de prueba: La Tarjeta Cash no existe

Nuestro contrato establece que deberíamos devolver 404 NOT FOUND si intentamos eliminar una tarjeta que no existe.

Escribe la prueba.

Agrega el siguiente método de prueba a `CashCardApplicationTests`:

```
1 @Test
2 void shouldNotDeleteACashCardThatDoesNotExist() {
3     ResponseEntity<Void> deleteResponse = restTemplate
4         .withBasicAuth("sarah1", "abc123")
5         .exchange("/cashcards/99999", HttpMethod.DELETE, null, Void.class);
6     assertThat(deleteResponse.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
7 }
```

Ejecuta las pruebas.

```
1 CashCardApplicationTests > shouldNotDeleteACashCardThatDoesNotExist() FAILED
2 org.opentest4j.AssertionFailedError:
3     expected: 404 NOT_FOUND
4     but was: 204 NO_CONTENT
```

¡No hay sorpresa aquí!

Necesitamos hacer cumplir la seguridad de nuestra aplicación verificando que el usuario que intenta eliminar una tarjeta es el propietario.

Spring Security hace muchas cosas por nosotros, pero esto no es una de ellas.

20.5 Hacer cumplir la propiedad

Necesitamos verificar si el registro existe. Si no, no deberíamos eliminar la Tarjeta Cash y deberíamos devolver 404 NOT FOUND.

Realiza los siguientes cambios en `CashCardController.deleteCashCard`:

```
1 private ResponseEntity<Void> deleteCashCard(
2     @PathVariable Long id,
3     Principal principal // Agrega Principal a la lista de parametros
4 ) {
5     // Agrega las siguientes 3 lineas:
6     if (!cashCardRepository.existsByIdAndOwner(id, principal.getName())) {
7         return ResponseEntity.notFound().build();
8     }
```

```

8     }
9     ...
10 }

```

¡Asegúrate de agregar el parámetro *Principal*!

Estamos usando el *Principal* para verificar si la Tarjeta Cash existe y, al mismo tiempo, hacer cumplir la propiedad.

Agrega el método `existsByIdAndOwner()` al Repositorio.

Además, agreguemos el nuevo método `existsByIdAndOwner()` a `CashCardRepository`:

```

1 interface CashCardRepository extends CrudRepository<CashCard, Long>,
    PagingAndSortingRepository<CashCard, Long> {
2     ...
3     boolean existsByIdAndOwner(Long id, String owner);
4     ...
5 }

```

Entiende el código del Repositorio.

Agregamos lógica al método del Controlador para verificar si el ID de Tarjeta Cash en la solicitud realmente existe en la base de datos. @ El método que usaremos es `CashCardRepository.existsByIdAndOwner()`.

Este es otro caso donde Spring Data generará la implementación de este método siempre que lo agreguemos al Repositorio.

Entonces, ¿por qué no usar simplemente el método `findByIdAndOwner()` y verificar si devuelve `null`? ¡Podríamos hacerlo absolutamente! Pero, tal llamada devolvería información extra (el contenido de la Tarjeta Cash recuperada), por lo que nos gustaría evitarla para no introducir complejidad adicional.

Si prefieres no usar el método `existsByIdAndOwner()`, ¡está bien! Puedes elegir usar `findByIdAndOwner()`. ¡El resultado de la prueba será el mismo!

Observa cómo pasa la prueba.

Ejecutemos la prueba y, sin gran sorpresa, ¡la prueba pasa!

```

1 [~/exercises] $ ./gradlew test
2 ...
3 BUILD SUCCESSFUL in 8s

```

20.6 Refactorizar

En este punto, tenemos una oportunidad de practicar el proceso Red, Green, Refactor. @ Ya hemos hecho Red (la prueba fallida) y Green (la prueba aprobada). @ Ahora podemos preguntarnos, ¿deberíamos refactorizar algo?

Aquí está el cuerpo de nuestro método `CashCardController.deleteCashCard`:

```

1 ...
2 if (!cashCardRepository.existsByIdAndOwner(id, principal.getName())) {
3     return ResponseEntity.notFound().build();

```



```

4 }
5 cashCardRepository.deleteById(id);
6 return ResponseEntity.noContent().build();

```

Podrías encontrar la siguiente versión, que es lógicamente equivalente pero ligeramente más simple, más fácil de leer:

```

1 if (cashCardRepository.existsByIdAndOwner(id, principal.getName())) {
2     cashCardRepository.deleteById(id);
3     return ResponseEntity.noContent().build();
4 }
5 return ResponseEntity.notFound().build();

```

Las diferencias son sutiles, pero eliminar un operador `!` de una declaración `if` a menudo hace que el código sea más fácil de leer, y la legibilidad es importante!

Si encuentras que la segunda versión es más fácil de leer y entender, entonces reemplaza el código existente con la nueva versión.

Ejecuta las pruebas de nuevo. ¡Siguen pasando!

```

1 [~/exercises] $ ./gradlew test
2 ...
3 BUILD SUCCESSFUL in 8s

```

20.7 Ocultar registros no autorizados

En este punto, podrías preguntarte, "¿Hemos terminado?" ¡Tú eres la mejor persona para responder esa pregunta! Si quieres, tómate un par de minutos para refrescarte con la lección asociada para ver si hemos probado e implementado cada aspecto del contrato de API para DELETE.

OK, ese fue tiempo bien empleado, ¿verdad? Así es - Hay un caso más que aún no hemos probado: ¿Qué pasa si el usuario intenta eliminar una Tarjeta Cash propiedad de alguien más? Decidimos en la lección asociada que la respuesta debería ser 404 NOT FOUND en este caso. @ Esa es información suficiente para nosotros para escribir una prueba para ese caso:

En `CashCardApplicationTests.java`, agrega el siguiente método de prueba al final de la clase:

```

1 @Test
2 void shouldNotAllowDeletionOfCashCardsTheyDoNotOwn() {
3     ResponseEntity<Void> deleteResponse = restTemplate
4         .withBasicAuth("sarah1", "abc123")
5         .exchange("/cashcards/102", HttpMethod.DELETE, null, Void.class);
6     assertThat(deleteResponse.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
7 }

```

Ejecuta la prueba.

¿Crees que la prueba pasará? Tómate un momento para predecir el resultado, luego ejecuta la prueba.

```

1 [~/exercises] $ ./gradlew test
2 ...
3 BUILD SUCCESSFUL in 8s

```

¡Todas pasaron! Esa es una gran noticia.

Hemos escrito una prueba para un caso específico en nuestra API. La prueba pasó sin cambios de código. Ahora, consideremos una pregunta que pudo haberte ocurrido: ¿Por qué necesito esa prueba, ya que pasa sin tener que hacer cambios de código? ¿No es el propósito de TDD usar pruebas para guiar la implementación de la aplicación? Si ese es el caso, ¿por qué me molesté en escribir esa prueba?

Respuestas:

- Sí, ese es uno de los muchos beneficios que TDD proporciona: Un proceso para guiar la creación de código para llegar a un resultado deseado.
- Sin embargo, las pruebas mismas tienen otro propósito, separado de TDD: Las pruebas son una red de seguridad poderosa para hacer cumplir la corrección. @ Dado que la prueba que acabas de escribir prueba un caso diferente a los ya escritos, proporciona valor. @ Si alguien hiciera un cambio de código que causara que esta nueva prueba fallara, ¡habrás atrapado el error antes de que se convirtiera en un problema! ¡Viva por las pruebas!

Una prueba más.

¡Pero espera, dices!

¿No deberíamos probar que el registro que intentamos eliminar aún existe en la base de datos - que no se eliminó?

Sí, esa es una prueba válida. ¡Gracias por mencionarlo!

Agrega el siguiente código al método de prueba para verificar que el registro que intentaste eliminar sin éxito aún está ahí:

```

1 void shouldNotAllowDeletionOfCashCardsTheyDoNotOwn() {
2     ...
3     // Agrega este código al final del método de prueba:
4     ResponseEntity<String> getResponse = restTemplate
5         .withBasicAuth("kumar2", "xyz789")
6         .getForEntity("/cashcards/102", String.class);
7     assertThat(getResponse.getStatusCode()).isEqualTo(HttpStatus.OK);
8 }

```

¿Crees que la prueba pasará? ¡Por supuesto que sí! (¿verdad?)

Ejecuta la prueba.

¡Una última ejecución de prueba!

Por favor, ejecuta todas las pruebas y asegúrate de que pasen.

```

1 [~/exercises] $ ./gradlew test
2 ...
3 BUILD SUCCESSFUL in 6s

```

¡Somos los mejores!

20.8 Resumen

En este laboratorio, implementaste un endpoint RESTful de Eliminación que no “filtra” información de seguridad a posibles atacantes. También tuviste la oportunidad de realizar una pequeña pero útil refactorización para practicar el proceso Red, Green, Refactor.

Esta es la última de las operaciones CRUD para implementar en la API, lo que nos lleva a una conclusión exitosa de nuestro trabajo técnico. ¡Felicidades!

21 Resumen rápido: implementación de la API REST CashCard

Inicio del proyecto

- Accede a `start.spring.io`
- Tipo de proyecto: `Gradle - Groovy`
- Lenguaje: `Java`
- Versión de Spring Boot: `3.x.x`
- Grupo: `example`
- Artefacto: `cashcard`
- Nombre: `cashcard`
- Paquete base: `example.cashcard`
- Dependencias:
 - Spring Web
 - Spring Data JPA
 - H2 Database
 - Spring Boot DevTools
 - Spring Security

Estructura general del proyecto

- Código fuente principal: `src/main/java/example/cashcard/`
- Pruebas: `src/test/java/example/cashcard/`

21.1 Modelo: CashCard

`src/main/java/example/cashcard/CashCard.java`

```
1 package example.cashcard;  
2  
3 public record CashCard(Long id, Double amount) {}
```

21.2 Repositorio: CashCardRepository

`src/main/java/example/cashcard/CashCardRepository.java`

```
1 package example.cashcard;
2
3 import org.springframework.data.domain.Pageable;
4 import org.springframework.data.jpa.repository.JpaRepository;
5
6 public interface CashCardRepository extends JpaRepository<CashCard, Long> {}
```

21.3 Controlador: CashCardController

src/main/java/example/cashcard/CashCardController.java

```
1 package example.cashcard;
2
3 import org.springframework.http.ResponseEntity;
4 import org.springframework.web.bind.annotation.*;
5 import java.net.URI;
6 import java.util.List;
7 import org.springframework.data.domain.Pageable;
8
9 @RestController
10 @RequestMapping("/cashcards")
11 class CashCardController {
12
13     private final CashCardRepository repository;
14
15     public CashCardController(CashCardRepository repository) {
16         this.repository = repository;
17     }
18
19     @GetMapping("/{id}")
20     public ResponseEntity<CashCard> findById(@PathVariable Long id) {
21         return repository.findById(id)
22             .map(ResponseEntity::ok)
23             .orElse(ResponseEntity.notFound().build());
24     }
25
26     @GetMapping
27     public ResponseEntity<List<CashCard>> findAll(Pageable pageable) {
28         return ResponseEntity.ok(repository.findAll(pageable).getContent());
29     }
30
31     @PostMapping
32     public ResponseEntity<Void> create(@RequestBody CashCard card) {
33         CashCard saved = repository.save(new CashCard(null, card.amount()));
34         return ResponseEntity.created(URI.create("/cashcards/" + saved.id())).build();
35     }
36
37     @PutMapping("/{id}")
38     public ResponseEntity<Void> update(@PathVariable Long id, @RequestBody CashCard
39         card) {
40         return repository.findById(id).map(existing -> {
41             repository.save(new CashCard(id, card.amount()));
42             return ResponseEntity.noContent().build();
43         }).orElse(ResponseEntity.notFound().build());
44     }
```

```
45 @DeleteMapping("/{id}")
46 public ResponseEntity<Void> delete(@PathVariable Long id) {
47     return repository.findById(id).map(existing -> {
48         repository.deleteById(id);
49         return ResponseEntity.noContent().build();
50     }).orElse(ResponseEntity.notFound().build());
51 }
52 }
```

21.4 Prueba: obtener tarjeta existente

src/test/java/example/cashcard/CashCardApplicationTests.java

```
1 @Test
2 void shouldReturnACashCardWhenDataIsSaved() {
3     HttpHeaders headers = new HttpHeaders();
4     headers.setBasicAuth("leonardo", "1234");
5     HttpEntity<Void> request = new HttpEntity<>(headers);
6
7     ResponseEntity<String> response = restTemplate.exchange(
8         "/cashcards/99", HttpMethod.GET, request, String.class);
9
10    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
11    Number amount = JsonPath.parse(response.getBody()).read("$.amount");
12    assertThat(amount).isEqualTo(123.45);
13 }
```

21.5 Prueba: tarjeta no encontrada

src/test/java/example/cashcard/CashCardApplicationTests.java

```
1 @Test
2 void shouldNotReturnACashCardWithUnknownId() {
3     HttpHeaders headers = new HttpHeaders();
4     headers.setBasicAuth("leonardo", "1234");
5     HttpEntity<Void> request = new HttpEntity<>(headers);
6
7     ResponseEntity<String> response = restTemplate.exchange(
8         "/cashcards/999", HttpMethod.GET, request, String.class);
9
10    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
11 }
```

21.6 Prueba: crear tarjeta

src/test/java/example/cashcard/CashCardApplicationTests.java

```
1 @Test
2 void shouldCreateANewCashCard() {
3     HttpHeaders headers = new HttpHeaders();
4     headers.setBasicAuth("leonardo", "1234");
5     headers.setContentType(MediaType.APPLICATION_JSON);
```

```
6
7 String newCardJson = "{ \"amount\": 250.00 }";
8
9 HttpEntity<String> request = new HttpEntity<>(newCardJson, headers);
10
11 ResponseEntity<Void> response = restTemplate.postForEntity(
12     "/cashcards", request, Void.class);
13
14 assertThat(response.getStatusCode()).isEqualTo(HttpStatus.CREATED);
15 }
```

21.7 Prueba: actualizar tarjeta existente

src/test/java/example/cashcard/CashCardApplicationTests.java

```
1 @Test
2 void shouldUpdateAnExistingCashCard() {
3     HttpHeaders headers = new HttpHeaders();
4     headers.setBasicAuth("leonardo", "1234");
5     headers.setContentType(MediaType.APPLICATION_JSON);
6
7     String updatedCardJson = "{ \"amount\": 999.99 }";
8
9     HttpEntity<String> request = new HttpEntity<>(updatedCardJson, headers);
10
11     ResponseEntity<Void> response = restTemplate.exchange(
12         "/cashcards/99", HttpMethod.PUT, request, Void.class);
13
14     assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NO_CONTENT);
15 }
```

21.8 Prueba: eliminar tarjeta

src/test/java/example/cashcard/CashCardApplicationTests.java

```
1 @Test
2 void shouldDeleteAnExistingCashCard() {
3     HttpHeaders headers = new HttpHeaders();
4     headers.setBasicAuth("leonardo", "1234");
5     HttpEntity<Void> request = new HttpEntity<>(headers);
6
7     ResponseEntity<Void> response = restTemplate.exchange(
8         "/cashcards/99", HttpMethod.DELETE, request, Void.class);
9
10     assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NO_CONTENT);
11 }
```