FINNEGANS

Construcción backend del crud de usuarios

Sprint deadline	18/10/2023
Tarjeta	SCRUM-12
Responsable	Martín Paez

Tabla 1 Detalle de la tarjeta correspondiente en Jira

1. Objetivos. Contenido de la tarjeta

- Controller: Definir API.
- Model:
  - Definir constraints.
  - Definir DTO para request y response.
- Service:
  - Validación de información recibida.
  - Validaciones previas a interactuar con la BD.
  - Lógica de negocio.
- IRepository.

2. Dependencias

En primer lugar, la tarjeta “SCRUM-13: Construcción frontend del crud de usuarios” precisa la definición de una API. En segundo lugar, la tarjeta “SCRUM-10: Validación de un usuario en el backend: login”, y, por consiguiente “SCRUM-11: Construcción del login (frontend)”, dependen de la definición de las estructuras DTO y Entity para User.

3. Entregable

- API bajo la especificación OpenAPI.
- Json para importación de la API en Postman.
- Migración de la base de datos completa para la tabla User.
- Migración básica necesaria de la base de datos para la tabla Order.
- Método de trabajo reutilizable en los próximos endpoints que el proyecto requiera.
- Mecanismo centralizado para los valores de las restricciones de la tabla User.
- Definición de los DTOs para Request y Response de la entidad User.
- Fábrica para la creación de los DTO y entidades User.
- Mecanismo que permita informar un conjunto de errores encontrados al procesar una solicitud (y que minimice el acceso a la base de datos).
- Mecanismo centralizado y reutilizable para validaciones comunes.
- Set granular de excepciones con su correspondiente mapeo a los códigos de respuesta HTTP.
- Sistema diseñado con un mecanismo de bajas lógicas para conservar información de los usuarios eliminados que tomaron al menos un pedido.
- Endpoins funcionales con todos los puntos detallados anteriormente para: “Registrar usuario”, “Actualizar datos de un usuario”, “Recuperar un usuario”, “Listar todos los usuarios”, “Eliminar un usuario”.

Se destaca que si bien se realizó la entrega con la especificación OpenAPI, posteriormente fue modificada a pedido del equipo, por otra más amigable para aquellos que no conocen el estándar.

FINNEGANS

1. Procedimientos

1.1. Respuesta a las dependencias

Por un lado, se charló con los responsables de las tarjetas con dependencia directa para confirmar los atributos de la entidad y el formato en que se iban a enviar los “json”. Se sentó un compromiso implícito de comunicar cualquier modificación a raíz de alguna necesidad originada en cualquiera de las partes.

Habiendo ganado tiempo con la decisión anterior, se dispuso a implementar detalladamente “UserController”. La idea era definir los “caminos felices” con códigos de respuesta superiores a 199 y menores a 300, así como los métodos HTTP, URL, parámetros variables eh información del cuerpo del mensaje de cada solicitud. Además, se implementaron respuestas “falsas” simulando el acceso a la base de datos, para brindar un comportamiento provisorio que permitiera testear la definición API y las implementaciones del “Frontend”.

1.1.1. Entrega provisoria: API

Para hacer disponible la API al resto del equipo, primero se “pusheo” la rama a Github (pero sin realizar el “merge” a dev, porque la “feature SRUM-12” no estaba completa). Luego se comunicó al equipo de frontend que podía clonar el repositorio en una carpeta separada y levantar el servidor de modo tal que respondiese sus peticiones HTML.

Además, se creo una colección en Postman con un ejemplo de consulta por cada “endpoint”, luego se exportó la colección y se guardo en la carpeta “test” del proyecto. Este sería el mecanismo de testeo durante el proceso de construcción y el método para dar a conocer la API al resto del equipo.

1.1. Trabajo de investigación y migración de la base de datos

Se está trabajando con la versión 11 de Java, con lo cual había que tomar los recaudos necesarios a la hora de buscar información.

Respecto a “javax.presistence”, a partir de la versión 8 Java EE se renombró como Jakarta EE, con lo cual desde el IDE IntelliJ se obtuvo información sobre la documentación a consultar: “jakarta.persistence:jakarta.persistence-api:2.2.3”. De allí, se investigó sobre la tecnología y sus decoradores.

Con la información recopilada se realizó la migración completa de la entidad User, y, de modo parcial, de la entidad Order. En esta última, solamente se contempló lo necesario para modelar la funcionalidad que vincula al usuario con las ordenes de compra que tomó.

1.2. Data transfer objects

Se optó por separar la lógica de creación de entidades y DTO de los constructores de las clases, para pasar a ser responsabilidad de una fábrica. Así como también, se centralizaron los valores máximo y mínimo de cada atributo del tipo “varchar”. Esto último fue útil para evitar redundancia de información tanto a la hora de realizar la migración como llevar a cabo las validaciones.

Para finalizar, se crearon los DTO UserRequest y UserResponse, los cuales son suficientes para el alcance de las funcionalidades del primer Sprint y probablemente de todo el proyecto.

1.3. Servicio

El equipo determinó que se implementará un único servicio con toda la lógica relativa al ABM de usuario y métodos auxiliares necesarios para el servicio de login. La otra opción era implementar un servicio por caso de uso, aunque se desestimó porque podría llegar a implicar mayor complejidad.

Al finalizar se obtuvo una clase de aproximadamente 200 líneas, cuyos métodos permiten realizar Alta, Baja, Modificación, Lectura y Listado de usuarios. Estos métodos invocan otros auxiliares para validaciones, que en algunos casos requieren de otros para cumplir con su funcionalidad y aspectos relacionados a “Clean Code”, en la mayoría de los casos “Single Responsability”.

Hay que recordar que la implementación de este primer servicio tenia como uno de sus objetivos establecer un plan de trabajo general, que sirva de guía para las futuras implementaciones de los demas “endpoint”. Por ello se describe este planteo, previendo que otros servicios, como el que tiene la lógica de las Ordenes de compra, pueda llegar a requerir muchas más líneas de código.

1.4. Validación y manejo de errores

Se implementó la interfaz Validator para realizar validaciones generales de parámetros, logrando centralizar el “feedback” de error en dichos casos y mejorando la seguridad en términos de evitar o simplificar la corrección de “bugs”. Se validan aspectos tales como longitud mínima o máxima de “Strings”, o, correctitud de un id contemplando el sistema de baja lógica, entre otros.

En las validaciones se hace uso del pasaje de StringBuilders para armar un error compuesto que luego se transformará en una determinada excepción (según sea el caso), y finalmente en una respuesta HTTP con el statusCode determinado por el manejador de excepciones.

Se consideró la idea de centralizar el texto de cada error en un archivo json o xml, para luego implementar una fábrica que los construya. Pero se concluyó que era una tarea que estaba por fuera del alcance del proyecto, ya que la meta es establecer un mecanismo de trabajo que todos los integrantes del equipo puedan llevar a delante sin complejidad. Hay miembros del equipo que recién comienzan a ayornarse en estas tecnologías.

El criterio general es evaluar todos los errores posibles en la información de la solicitud recibida antes de hacer cualquier consulta a la base de datos, aún si estas consultas fueran necesarias para completar el proceso de validación. O sea, si, por ejemplo, se encontraron tres errores diferentes antes de necesitar acceder a la base de datos, se le informarán al usuario esos mismos tres errores, finalizando así la petición. Luego, el usuario realizará una segunda petición con las correcciones pertinentes, para la cual, de ser necesario, se accederá

a la base de datos para completar el resto de los chequeos. Respecto a las validaciones de lógica de negocio, el criterio es el mismo, se solucionará primero todo aquello que no requiera solicitudes a la base de datos.

1.5. Mecanismo de baja lógica

La necesidad de esta funcionalidad radica en el hecho de que, durante la etapa de diseño, elegimos asociar las órdenes al usuario que las tomó, o sea, al empleado de la empresa que habló con el cliente y concretó la venta. Siguiendo este planteo, si en algún momento se quiciera eliminar dicho usuario, por ejemplo, tendríamos una “foreign key” en la tabla “Order” presentando una restricción. Por consiguiente, la idea es conservar la información, y, en lugar de eliminarlo asignarle una baja lógica, de modo tal que para el resto del sistema sería indistinto a si el usuario hubiera sido eliminado.

Implementar una baja lógica trae una complejidad inherente a las validaciones. Como no es natural contemplarla, es fácil que se le pase por alto al desarrollador. Estas validaciones inciden en multiplicidad de métodos, incluso tuvieron participación en la implementación del servicio de Login (que usa un método perteneciente al servicio que estamos describiendo).

También cabe mencionar que este no es un caso aislado, y para cumplir el objetivo hay que contemplarlo a la hora de establecer la metodología de trabajo. Puede resultar útil en otras funcionalidades como eliminar un Cliente que ah concretado órdenes de compra en el pasado.

1.5.1. Implementación del mecanismo de baja lógica

Para llevar a cabo la baja lógica, la entidad, en este caso User, debe poseer un atributo “enabled” eh implementar la interfaz SoftDeletable. Luego, la lógica de validación emplea los métodos de dicha interfaz. Incluso la antes mencionada interfaz Validator contempla un método de tipos genéricos para elementos SoftDeletable.

La idea básica es que, a la hora de eliminar el elemento, si este se encuentra asociado a algún otro registro de la base de datos, en vez de borrarlo se le de una baja lógica. Luego, al de recuperarlo, por ejemplo, en base a un id o una búsqueda con algún otro criterio, habrá que contemplar un filtro para trabajar únicamente con aquellos registros que tienen el atributo “enabled” habilitado.

4. Resultados

4.1. API y Postman

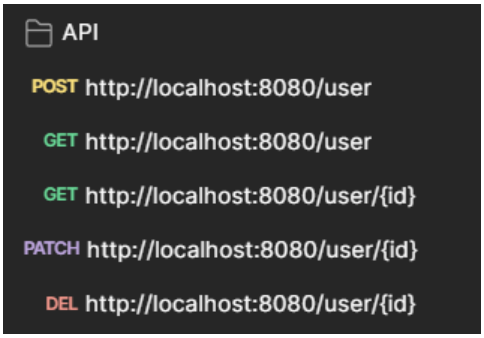

Ilustración 1Captura de pantalla de Postman de la API para el ABM de usuario bajo la especificación openAPI.

4.2. Gestión de múltiples errores en simultáneo

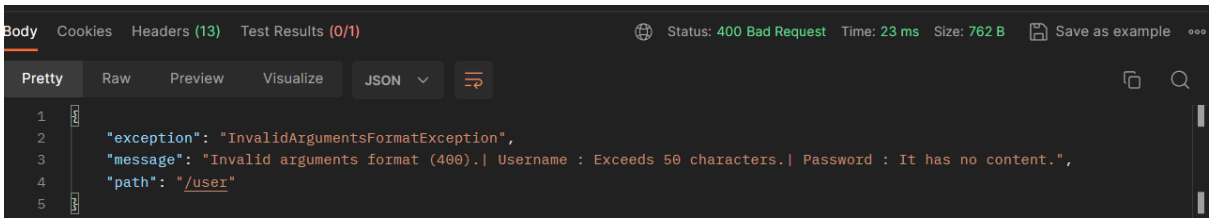

Ilustración 2 Captura de pantalla de Postman para un ejemplo de respuesta con información de múltiples errores.

4.3. Mecanismo de baja lógica.

	id	enabled	password	username
1	7	0	{bcrypt}$2a$10$K2YInadA/hbiXkNV4Sa3vuoIkWvNTOo9zoE52vqqOw…	admin
2	8	1	{bcrypt}$2a$10$K2YInadA/hbiXkNV4Sa3vuoIkWvNTOo9zoE52vqqOw…	antoniosena
3	10	1	{bcrypt}$2a$10$K2YInadA/hbiXkNV4Sa3vuoIkWvNTOo9zoE52vqqOw…	martinpaez

	order_id	user_id
1	3	10

Ilustración 3 Captura de pantalla de Azure. Se observan que, además del usuario admin, hay otros dos usuarios registrados y activos, uno de ellos tomó una orden.

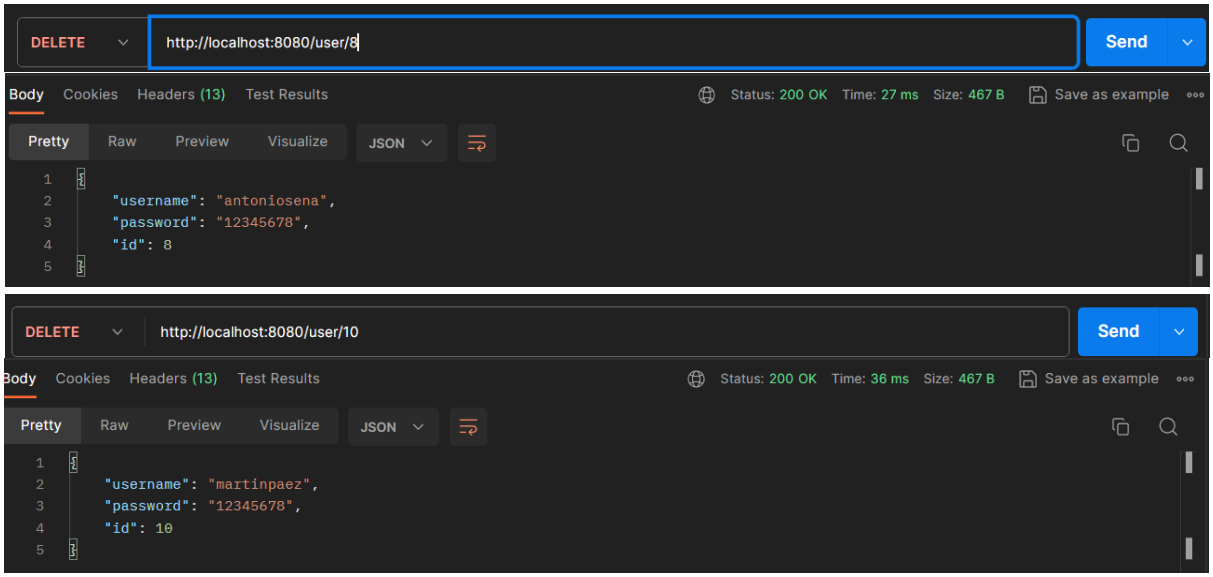

Ilustración 4 Dos capturas de pantalla del Postman, en las que se elimina a los usuarios.

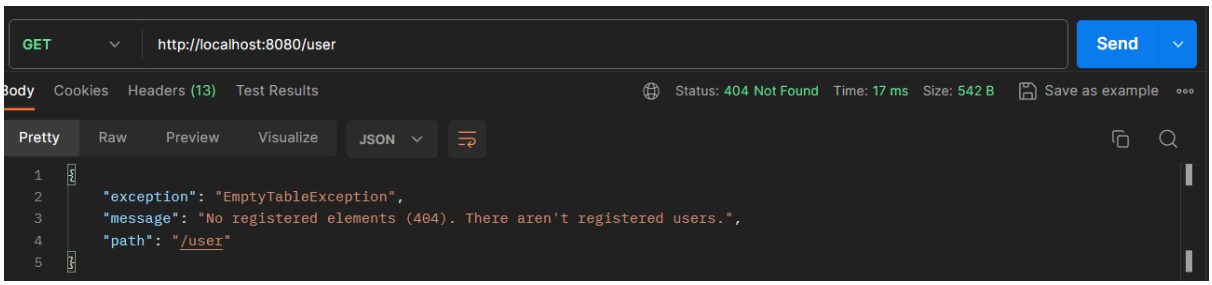

Ilustración 5 Luego de consultar la API, se confirma que no hay usuarios registrados, ya que el usuario admin tenia baja lógica y los otros únicos dos usuarios que había fueron eliminados antes de esta consulta.

	id	enabled	password	username
1	7	0	{bcrypt}$2a$10$K2YInadA/hbiXkNV4Sa3vuoIkWvNTOo9zoE52vqqOw…	admin
2	10	0	{bcrypt}$2a$10$K2YInadA/hbiXkNV4Sa3vuoIkWvNTOo9zoE52vqqOw…	martinpaez

	order_id	user_id
1	3	10

Ilustración 6 Las respuestas a las consultas de la base de datos desde Azure revelan que el usuario que no tenia ninguna orden asociada efectvamente fué eliminado, mientras que el otro tiene baja lógica.

2. Conclusión

Se implementó una API para alta, baja, modificación, obtención de un usuario y listado de usuarios activos. La misma servirá para guiar el desarrollo de los futuros “endpoints” del proyecto.

Respecto a las validaciones se realizó un trabajo minucioso, que de ser necesario envía información de múltiples errores encontrados en una misma petición. Además, queda a disposición del equipo la implementación de una interfaz que permitirá reutilizar código en validaciones de otros servicios.

Por último, se conservará la información de aquellos usuarios que fueron eliminados, pero tomaron al menos una orden de compra, mediante un mecanismo de baja lógica.