Desafío Spring

Social Meli

Autor: Karen Martinez

Agosto 2021

Índice

1. Introducción

A continuación se hará una breve descripción sobre el trabajo, la cual será información complementaria a la documentación donde se presenta el desafío. El objetivo no será hacer una ficha técnica exhaustiva; sino que mencione los puntos más relevantes.

Se detallarán temas como la toma de decisiones en el correr del proyecto, y funcionalidades o especificaciones no mencionadas en el documento original.

2. Toma de decisiones

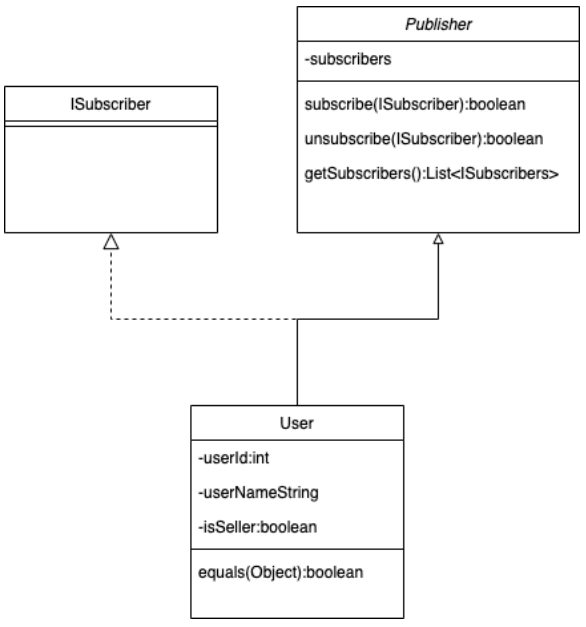
En esta sección se enumerarán los diferentes desafíos por los cuales se tuvo que optar por una decisión; sea esta de diseño y/o funcional.

1. Suscripción de un usuario a un vendedor. Relación de Usuario y Vendedor

Aquí se tuvieron que tomar dos decisiones; una en relación a cómo representaríamos a un vendedor y a un usuario en el sistema, y la otra el cómo se manipularía la suscripción.

En esta solución, tanto el usuario como el vendedor son representados por la misma entidad; diferenciándolos con el atributo *isSeller:boolean.* Bajo otras condiciones, y si hubieran diferentes tipos de usuarios se podría haber hecho una interfaz o clase abstracta (pero no fue el caso).
Lo que cabe destacar es que, ciertas operaciones del sistema sólo podrán realizarse tomando en cuenta que un usuario sea vendedor (se detallará en cada endpoint).

Por otro lado, para la suscripción se optó por aplicar el patrón Observer, o también conocido como Publish/Subscriber (de forma minimalista).



Como se puede apreciar, en esta aplicación Publisher no notifica a sus suscriptores (dado a que no se presentó la necesidad de notificar a los suscriptores), y un User puede tanto seguir como ser seguido.

Esta solución facilita las implementaciones relacionadas al seguir y dejar de seguir a un vendedor, pero, por otro lado, si quisiéramos saber los vendedores que sigue un usuario, esta solución se vuelve un poco más complicada. De todas formas, haciendo buen uso de los filtros y de la lista que dispone Publisher es algo que se puede obtener.

Una forma válida podría haber sido que cada ISubscriber mantuviera un registro de todos los vendedores a los que sigue; pero se optó por no realizarlo debido al costo de mantenerlo. Ya que cada vez que se hicieran operaciones que afectasen a la colección de suscriptores, habría que revisar y modificar eventualmente la lista de publicadores.

2. Vendedor en el sistema

Como ya mencionamos en el punto anterior, los vendedores se diferencian de los usuarios comunes dado un atributo llamado isSeller:boolean. En esta solución, hay operaciones que requieren que uno o más de los usuarios involucrados sea vendedor.

En la charla de a pares del día uno surgió la idea de convertir un usuario a vendedor si el mismo realizaba una publicación. Esto afectaría al menos al requerimiento US0005 (ya que se podría decir que si no es un vendedor efectivo no puede hacer publicaciones en promoción).

Conste que un usuario vendedor no se define por si tiene posts o no, sino por haber sido definido en el sistema como vendedor, o al haber realizado al menos una publicación. En lo personal se hubiera preferido considerar a cualquier usuario como candidato a vendedor, pero se mencionó poner un criterio para diferenciar un usuario a otro que sea vendedor. Así que se manejó esta posibilidad; la cual puede ser debatible.

A continuación detallaremos los requerimientos que se vieron afectados.

US0001

Method	**Sign**
POST	/users/{userId}/follow/{userIdToFollow}

{userId} no podrá seguir a {userIdToFollow} si {userIdToFollow} no es vendedor. Por lo cual devolverá mensaje con un status 400 (BAD_REQUEST)

US0002

Method	**Sign**

GET	/users/{userId}/followers/count

No se podrá obtener el listado de seguidores si {userId} no es vendedor. Por lo cual devolverá un mensaje con un status 400 (BAD_REQUEST)

US0003

Method	**Sign**
GET	/users/{UserID}/followers/list

No se podrá obtener el listado de seguidores si {userId} no es vendedor. Por lo cual devolverá un mensaje con un status 400 (BAD_REQUEST)

US0005

Method	**Sign**
POST	/products/newpost

Si el usuario con la identificación del post no es vendedor, luego de publicar pasará a serlo.

US0007

Method	**Sign**
POST	/users/{userId}/unfollow/{userIdToUnfollow}

No se podrá dejar de seguir a {userIdToUnfollow} si el mismo no es vendedor. Ya que de por si nunca se podría haber seguido a {userIdToUnfollow} por US0001.

US0010

Method	**Sign**
POST	/posts/newpromopost

No se podrá realizar una publicación si el usuario con la identificación del post no es vendedor.

US0011

Method	**Sign**
GET	/products/{userId}/countPromo

No se podrá obtener la cantidad de productos en promoción de {userId} si el mismo no es un vendedor.

US0012

Method	**Sign**
GET	/products/{userId}/list

No se podrá obtener la lista de productos en promoción de {userId} si el mismo no es un vendedor.

3. División de Controladores

Cabe destacar que en esta solución se hicieron dos controladores; uno llamado ProductController y UserController aunque ambos utilizan el mismo servicio. Esto se hizo por un tema de orden y relación con las uris. En este caso, todas las requests que comienzan con “users” están en UserController, y con “products” en ProductsController.

4. División de Repositorios

En el proyecto se optó por manejar dos repositorios; uno de posts(PostRepository) y otro de usuarios(UserRepository). Esto se debió mayormente por la necesidad de mantener publicaciones y usuarios con una única identificación (es decir, no pueden haber dos usuarios con la misma identificación, ni dos publicaciones con la misma identificación).

La identificación de los usuarios como la de los posts son generados de forma automática utilizando la clase Calculator; el cual asigna números consecutivos en cada repositorio. Se consideró que esta forma de manejar las identificaciones es mejor a pedirle al que use la api ingresarlos; ya que se desliga al mismo de manejar dicha información.

Para el manejo de posts de los usuarios vendedores se optó inicialmente por que Post tuviera la identificación del usuario creador. Pero, al estar programando en un lenguaje orientado a objetos esto no es una buena práctica (se debería pasar User y no mantener un identificador). Por ende, finalmente se modificó para que Post tuviera un User y no su identificador.

Para los productos, los mismos son creados y guardados dentro de Post, en PostRepository.

Se consideró mostrar los atributos hasPromo:boolean y discount en los DTO que involucren Post debido a los requerimientos bonus (US0010 al US0012).

La solución actualmente no maneja persistencia, sino que los datos son creados al momento de iniciar la aplicación (mediante los constructores de los respectivos repositorios).

5. División de Servicios

Actualmente solo existe una interfaz(e implementación) de servicio que es manipulada por los controladores; IPostUserService, la cual antes se llamaba IUserService. Se ha querido separar dicha capa en dos; para por un lado manejar las operaciones relacionadas a los posts, y otra para los usuarios. Pero, debido a que los posts requieren de validaciones de usuarios, se optó por ahora dejar ambas responsabilidades en la capa IPostUserService.

6. Manejo de Excepciones

A la hora de realizar el manejo de las excepciones, se pensó en primer lugar en desligar a los repositorios de manejarlos; siendo la capa de servicios la única responsable de instanciarlos. Salvo en algunos casos particulares, como la excepción *HttpMessageNotReadableException* (generada a partir de la anotación *@jsonformat* en PostDTO), y excepciones del sistema, todas las excepciones previstas en la solución poseen su clase particular y mensaje (derivando todas de *GenericException*, que extiende de *RunTimeException*).
El responsable de atrapar dichas excepciones es la clase *ApiExceptionMiddleware*, bajo la anotación de *@ControllerAdvice*, para que la misma atrape todas las excepciones que puedan surgir en nuestra API. Y, mediante la anotación *@ExceptionHandler* definimos los métodos correspondientes para cada tipo de excepción. Todas las excepciones particulares instanciadas por la capa de servicio irán al método *handleOwnExceptions*.

Se reconoce que varias validaciones se podrían haber encapsulado. Por ejemplo, si un Post llegase a tener varias validaciones, y hoy o mañana se permite modificar los Post, tal vez en vez de tenerlo en addPost sería mejor separarlo en otro método que realice todas las validaciones en común en un alta y modificación. En todo caso habría que evaluar los usos a futuro y el cómo podría a llegar a complejizarse el método con el incremento de validaciones.

3. Especificación de endpoints

A continuación se detallarán todos los endpoints. Indicando las consideraciones pertinentes a tomar en cuenta en cada uno. Cabe destacar que algunas de las condiciones ya han sido mencionadas en la sección anterior; pero se opta por mencionarlas de todas formas por practicidad para el lector. Los requerimientos que terminen con * son aquellos que no fueron solicitados.

US0001

Method	**Sign**
POST	/users/{userId}/follow/{userIdToFollow}

- Si {userIdToFollow} no es vendedor, devolverá mensaje con un status 400 (BAD_REQUEST)
- Si el {userId} no existe, devolverá status 400 (BAD_REQUEST) con un mensaje.

- Si el {userIdToFollow} no existe, devolverá status 400 (BAD_REQUEST) con un mensaje.
- Si {userId} == {userIdToFollow}, devolverá status 400 (BAD_REQUEST) con un mensaje.

US0002

Method	**Sign**
GET	/users/{userId}/followers/count

- Si {userId} no es vendedor, devolverá un mensaje con un status 400 (BAD_REQUEST)
- Si el {userId} no existe, devolverá status 400 (BAD_REQUEST) con un mensaje.

US0003

Method	**Sign**
GET	/users/{UserID}/followers/list

- Si {userId} no es vendedor, devolverá un mensaje con un status 400 (BAD_REQUEST)
- Si el {UserID} no existe, devolverá status 400 (BAD_REQUEST) con un mensaje.

US0004

Method	**Sign**
GET	/users/{UserID}/followed/list

- Si el {UserID} no existe, devolverá status 400 (BAD_REQUEST) con un mensaje.
- La request devolverá los elementos de *followed* con la ordenación *order=name_asc* por defecto (requerimiento US0008).

US0005

Method	**Sign**
POST	/products/newpost

- Si el usuario con la identificación del post no es vendedor, el mismo pasará a convertirse en vendedor luego de haberse creado la publicación.
- Si el usuario con la identificación del post no existe, devolverá status 400 (BAD_REQUEST) con un mensaje.
- Si la date:Date es inválida, devolverá status 400 (BAD_REQUEST) con un mensaje.
- Los atributos hasPromo:boolean y discount:discount estarán pre-fijados. hasPromo=false y discount=0.0.
- Si price:status<0, devolverá status 400 (BAD_REQUEST) con un mensaje.
- Devolverá el Post creado con la id generada.

US0006

Method	**Sign**
GET	/products/followed/{userId}/list

- Si el {userId} no existe, devolverá status 400 (BAD_REQUEST) con un mensaje.
- La request devolverá los elementos de *followed* con la ordenación *order=date_desc* por defecto (requerimiento US0008).

US0007

Method	**Sign**
POST	/users/{userId}/unfollow/{userIdToUnfollow}

- Si {userIdToUnfollow} no es vendedor, devolverá status 400 (BAD_REQUEST) con un mensaje.
- Si {userId} no seguía a {userIdToUnfollow}, devolverá status 400 (BAD_REQUEST) con un mensaje.
- Si el {userId} no existe, devolverá status 400 (BAD_REQUEST) con un mensaje.
- Si el {userIdToFollow} no existe, devolverá status 400 (BAD_REQUEST) con un mensaje.
- Si {userId} == {userIdToFollow}, devolverá status 400 (BAD_REQUEST) con un mensaje.

US0008

Method	**Sign**
GET	/users/{UserID}/followers/list?order=name_asc /users/{UserID}/followers/list?order=name_desc /users/{UserID}/followed/list?order=name_asc /users/{UserID}/followed/list?order=name_desc

- Aplican las mismas características que las respectivas especificaciones de los requerimientos US0003, US0004.
- Si order=name_asc, ordenará *followers*/*followed* de forma ascendente por nombre de usuario.
- Si order=name_desc, ordenará *followers*/*followed* de forma descendente por nombre de usuario.

US0009

Method	**Sign**
GET	/products/followed/{userId}/list?order=date_asc /products/followed/{userId}/list?order=date_desc

- Aplican las mismas características que las respectivas especificaciones del requerimiento US0006.
- Si order=date_asc, ordenará *posts* de forma ascendente por fecha de publicación.
- Si order=date_desc, ordenará *posts* de forma descendente por fecha de publicación.

US0010

Method	**Sign**
POST	/posts/newpromopost

- Aplican las mismas características que las respectivas especificaciones del requerimiento US0005 exceptuando a que si el usuario no es vendedor, devolverá status 400 (BAD_REQUEST) con un mensaje. A continuación se detallarán más diferencias.
- El atributo hasPromo:boolean estará pre-fijado, siendo hasPromo=true.
- discount:double deberá estar entre los valores mayores a 0 y menores a 1, sino devolverá status 400 (BAD_REQUEST) con un mensaje.
- Si price:double<0, devolverá status 400 (BAD_REQUEST) con un mensaje.
- Devolverá el Post creado con la id generada.

US0011

Method	**Sign**
GET	/products/{userId}/countPromo

- Si {userId} no es un vendedor, devolverá status 400 (BAD_REQUEST) con un mensaje.

US0012

Method	**Sign**
GET	/products/{userId}/list /products/{userId}/list?order=name_desc /products/{userId}/list?order=name_asc

- Si {userId} no es un vendedor, devolverá status 400 (BAD_REQUEST) con un mensaje.
- Opcionalmente, los elementos en posts podrán ordenarse por nombre de producto de forma ascendente y descendente. De forma predeterminada no posee una ordenación definida.

US0013*

Method	**Sign**
GET	/users/list

- Este requerimiento devuelve todos los usuarios del sistema. Se agrega para saber los usuarios que hay en el sistema.

4. Datos de prueba

Los datos de prueba se pueden encontrar de forma hardcodeada en cada implementación de repositorio (UserRepositoryImp y ProductRepositoryImp). Para la corrección se dejaron los datos de prueba que cumplen con las siguientes condiciones anunciadas en Slack:

- 2 compradores con ids 1 y 2.
- 2 vendedores con ids 3 y 4.
- Sin post en el sistema.
- Sin follows.