.UBAfiuba
FACULTAD DE INGENIERÍA

Trabajo Práctico Final

Motor de ofertas

Grupo 6

Alumno	Padrón	Mail
Carol Lugones, Ignacio	100073	icarol@fi.uba.ar
Illescas, Geronimo	102071	gillescas@fi.uba.ar
Sinisi, Fernando	99139	fsinisi@fi.uba.ar
Torresetti, Lisandro	99846	ltorresetti@fi.uba.ar

Objetivos

El objetivo del trabajo práctico es implementar una herramienta que permita validar las ofertas, dadas al inicio de ejecución de la api, sobre un carrito de compras propuesto a través de un cliente externo a la misma. Esta herramienta devuelve el listado de las ofertas aplicadas sobre los productos del carrito propuesto. Las ofertas pueden estar relacionadas con:

- Producto comprado.
- Fecha de compra.
- Método de compra.
- Combinación de las anteriores mediante negación, conjunción o con requerimiento mínimo.

También se propuso como objetivo la creación de un front mínimo para visualizar esto como un MVP, y la dockerizacion de la api que contiene esta herramienta más una dockerizacion del front para el MVP.

Como un valor agregado también se propuso hacer una herramienta de logueo y de creación de usuario manejado con JWT (JSON web token).

Composición de la API

La API está compuesta por una estructura que separa funcionalidad en folders, logrando separar la lógica de código de la api, como rest-api, de la lógica de los servicios utilizados.

Comienza usando routes como un estilo de *capa de controller* que se encarga de recibir y de devolver los mensajes que lleguen del exterior de la api en un formato entendible con el estándar *rest*.

Luego sigue por la *capa de servicio* que se encarga de la operación correspondiente para cada sector, realizando una capa del medio entre el controller y análisis de datos que sean de la api.

También tenemos una *capa de utils* donde tenemos todo lo que no involucre con un proceso core pero que influya en uno de ellos, como ejemplo tenemos clases, que implementan el patrón *Factory*, que se encargan de crear las reglas o las funciones de comparación a utilizar.

En el folder de *models* se encuentran los modelos necesarios para el funcionamiento de la api.

Persistencia de la API

La persistencia está hecha con MongoDB usando la librería de *mongoose* en typescript, cuando se utiliza de manera local, se genera una base de datos mongo de manera local donde se guardará todo lo que sea necesario. Cuando se utiliza docker compose, se crea un mongo db aparte del ambiente *dockerizado* de la api, y se utiliza ese ambiente directamente.

Esta persistencia es utilizada actualmente solo para el guardado de usuarios con su contraseña y se obtiene a la hora de hacer un login para verificar los valores propuestos a la hora de ingresar a la plataforma.

Una segunda capa de persistencia es un guardado de archivos de ofertas y productos para el cargado de los mismos cuando se inicia la API, estos son archivos en formato json que son constantes mientras la misma se ejecuta y se utilizará como base de modelos.

Lógica de la API

La lógica de la API la podemos separar en dos partes, la parte del usuario, ya sea creación o registro, y la parte del chequeo de los carritos para que se le apliquen las ofertas.

Usuario:

Se puede separar en dos grandes partes. Uno que es el signup y el signin.

Para el **signup** lo que se utiliza es un intento de guardado del usuario, pasándose datos como el nombre, apellido, mail y password, y se chequea si los datos enviados son válidos donde las condiciones son las clásicas para este tipo de proceso (password con más de 8 caracteres, no campos vacíos, etc). Si el signup es exitoso, se guardará el usuario en una base de datos de mongoDB, en caso contrario se devolverá error a la entidad tratando de handlear el sign-up.

En caso de que el usuario ya haya sido creado (ya sea por un previo sign-up o que simplemente el mail ya haya sido utilizado), se devolverá un error correspondiente indicando que no es posible tener usuarios por duplicado.

Para el **sign-in**, se valida que el email y password enviados existan en la base de datos previamente descrita, y que los mismos sean correctos (email y password coincidan). En caso de que eso sea correcto se devolverá un status ok, y un jwt token para las futuras validaciones del usuario, de lo contrario se lanza un error indicando informando que alguno de los campos es erróneo.

Este token será el encargado de ser el que indique que el usuario esté logueado y que el tiempo sea válido (el default es 2hs después de creado).

La validación será hecha a través de la librería de jwt. El mismo identificará si el token es válido y el tiempo como fue anteriormente dicho. En caso de que sea invalido se devolverá un error y no continuará con la ejecución del endpoint que se quiera usar.

Ofertas y productos

Initialize Offer

En initializeOffers recibimos un objeto con reglas y ofertas donde se validan y se normalizan.

Para las reglas en el listado hacemos validaciones según su naturaleza.

- **atómicas**: se valida que tengan descripción, código, tipo y field a procesar.
- **complejas**: se valida su tipo, las subreglas internas y el código. Para las subreglas internas se modifica en caso de que tenga referencia por código a otras reglas, dejando cada rule con las subreglas listas para validar sin tener que buscar a posteriori reglas a un listado.

Para las ofertas se realizan validaciones de que tengan válido el *purchase_date*, el payment y que las reglas que tengan estén listas para ser validadas. Al igual que el listado de reglas se reemplazan en caso de que existan referencias por código de rule por el objeto como tal.

El método devuelve las reglas y las ofertas validadas en un objeto de tipo *state*.

Process Product

El método tiene por entrada al estado devuelto por initializeOffers y recibe en conjunto un objeto shoppingCart. En dicho método se computan los descuentos que deben aplicarse a los productos del carrito.

Se retorna un array de elementos compuesto por los productos y los descuentos aplicados sobre cada uno de ellos. En caso de no aplicarse descuentos no se devuelve dicho nodo.

Diagramas

Diagrama 4+1 con respecto a la obtención de las ofertas

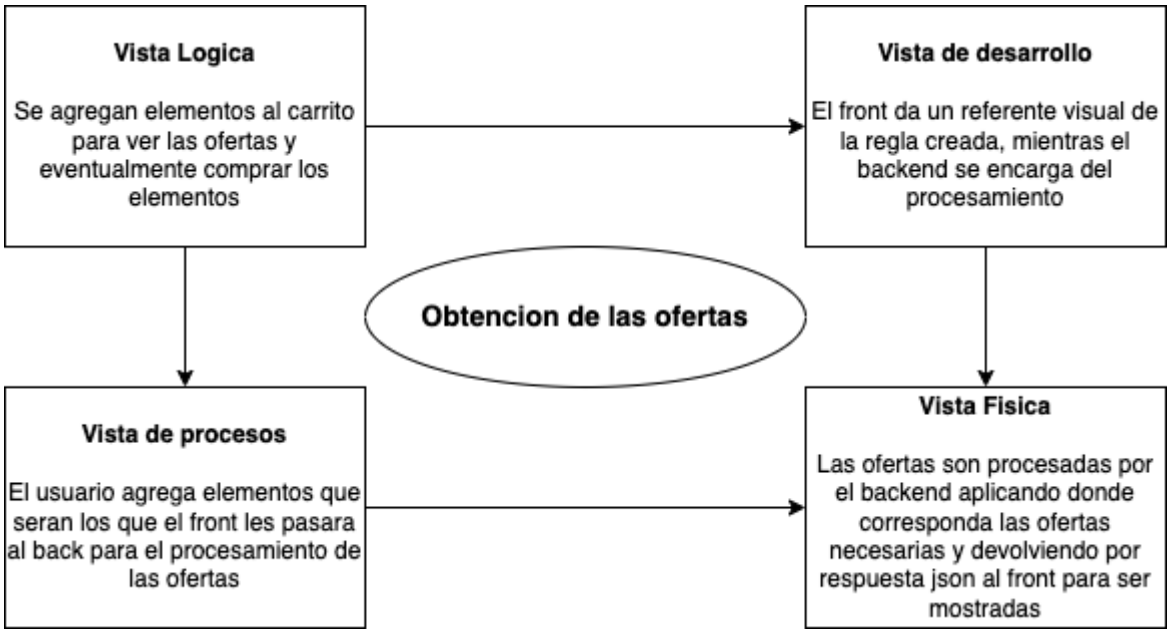


Diagrama interacción front backend para login

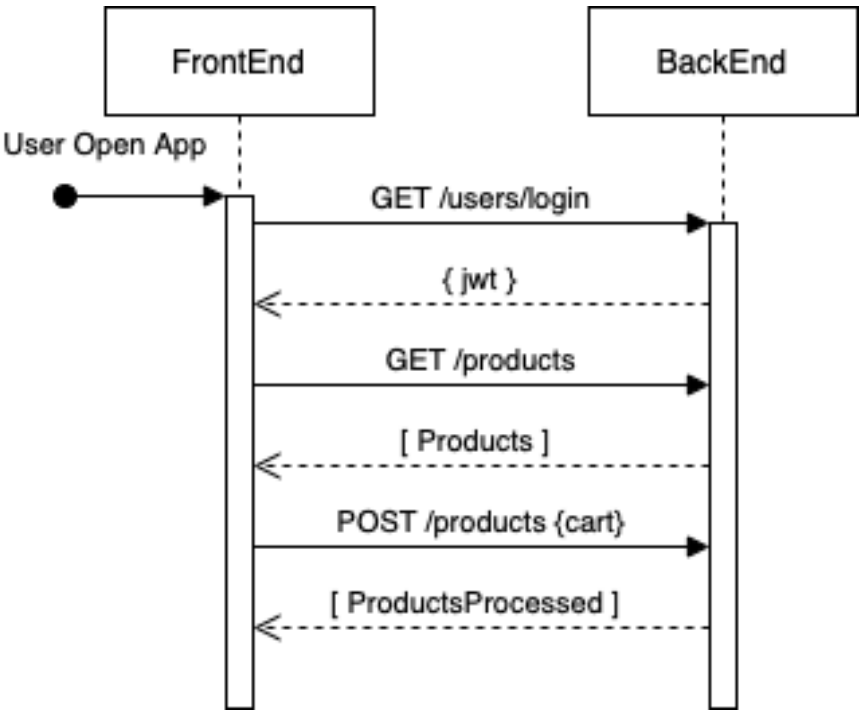


Diagrama interacción front backend para procesar productos

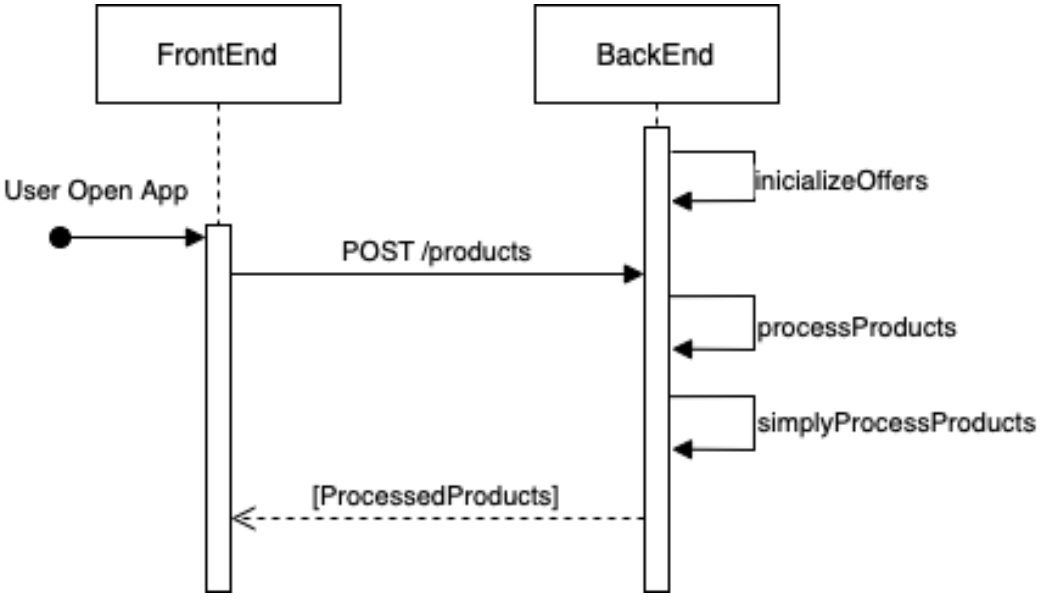
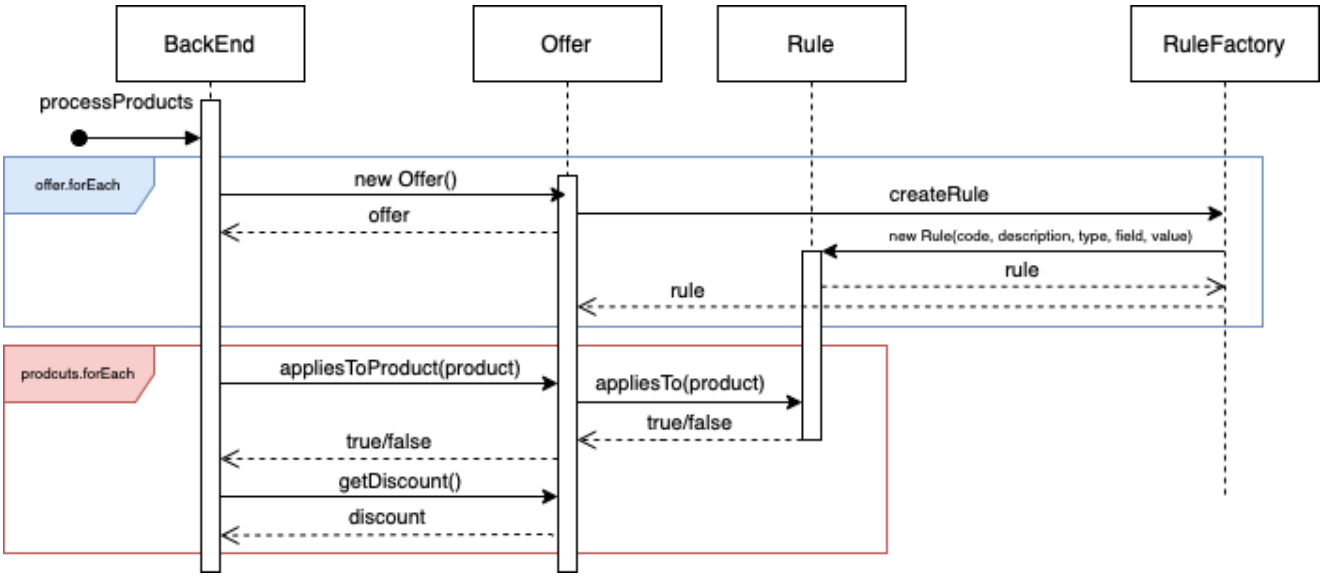


Diagrama interacción componentes del backend para procesar productos



Estructura de la interfaz gráfica

La interfaz gráfica es una simple aplicación frontend realizada utilizando node.js y javascript, con el agregado de un dockerfile. La misma no cuenta con reglas de negocio, tan solo es una manera de representar la información obtenida de la aplicación backend y a su vez permitir interactuar con la misma, como por ejemplo mostrando los productos disponibles y agregandolos al carrito para luego enviar el mismo a la aplicación backend y que esta aplique, si corresponden, las ofertas.

Esta interfaz cuenta con 3 pantallas, ubicadas en el directorio “src.screens”, de las cuales 2 corresponden a cuando un usuario no está logueado (pantalla de login y pantalla de signup) y la restante es la pantalla principal cuando se logueo correctamente. Las pantallas de login y signup cuentan con unas mínimas validaciones respecto del llenado de campos obligatorios, con el fin de que un request, que se sabe de antemano va a ser fallido, no llegue al backend.

Algo importante de mencionar es que estas pantallas comparten el contexto de la aplicación, en el directorio “src.contexts”, con el fin de mantener en el storage local información de inicio de sesión como el jwt o mail del usuario logueado.

En el folder “components” se guardan los componentes que podemos reutilizar en distintas pantallas si las hubiera, en nuestro caso solo tenemos una barra de navegación que en la pantalla principal muestra el botón de logout y en que pestaña estamos ubicados.

Consideraciones

- Podemos mejorar el uso de jwt con un refresh token para cuando expira el mismo.
- Escala de manera buena el agregado de productos y reglas para las ofertas de esos productos, ya que es sólo crear la clase que represente a la nueva regla y luego agregarla al correspondiente Factory.
- Para el caso de las funciones de comparación, agregar nuevas también resulta sencillo ya que implica realizar cambios en una sola clase (*ComparisonFunctionFactory*).

- Para correr el docker-compose lo que se necesita es que el formato de los repos sea los dos en el mismo root (que se pueda acceder retrocediendo un folder a cualquiera de los dos).

Dificultades

- La creación de docker compose fue complejo ya que tuvimos que ver cómo integrar los networks para vincular el front con el back y el modelo.
- Aprender el uso de typescript como lenguaje nuevo para todos los integrantes del equipo. Por ejemplo el manejo de diferentes tipos de typescript unificado (union type)
- El uso de jwt como token de control fue un desafío interesante.
- Las inconsistencias halladas entre el enunciado y los test de aceptación brindados.

Conclusiones

El trabajo realizado resultó un desafío desde varios puntos de vista, del diseño de la estructura del motor de reglas, ya que se debió encontrar una manera adecuada y que resulte escalable de diseñar la aplicación de reglas respetando el contrato de los métodos y clases, desde el uso de typescript que no todos lo habíamos utilizado con anterioridad y desde la coordinación entre los miembros del grupo para el uso de las buenas prácticas de programación.

Por otra parte, el aprendizaje fue mucho ya que pudimos implementar una aplicación full-stack, tanto back como front, lo cuál es sumamente útil porque tuvimos que tomar decisiones tales como qué se espera de cada componente, qué arquitectura funciona mejor, como se le debería mostrar la información al usuario, si se debería utilizar un patrón de diseño en particular y porque.