Proyecto: Iteración 4. Continuos Integration / Continuous Deployment (CI/CD)

GRUPO 03:
Mariajose Zuloeta Brito
Adrián Lima García
Samuel Lorenzo Sánchez

6 de diciembre de 2024

1. Funcionalidades

Frontend:

- Vista de lista de deseos
- Previsualización de imágenes en edición de item

Backend:

- Conectar componente de subcategorías
- Posibilidad de actualizar imagen
- Mockup de prueba

Integración continua:

- Github Actions

2. Implementación de las funcionalidades

2.1. Vista de lista de deseos:

Para permitir que los compradores gestionen su lista de deseos, hemos implementado dos botones en la vista detallada del producto: uno para agregar productos y otro para eliminarlos. Estos botones se configuran para mostrarse únicamente cuando el usuario tiene el rol de comprador, garantizando que la funcionalidad sea accesible solo para ellos.

En la vista de la lista de deseos, realizamos un fetch para obtener los productos almacenados en la wishlist y utilizamos el componente GenericList, diseñado en iteraciones anteriores, para renderizar la lista de forma eficiente.

Como detalle adicional, si un comprador intenta añadir un producto que ya está en su lista de deseos, se mostrará un mensaje indicando que dicho producto ya está agregado. De forma similar, si intenta eliminar un producto que no está en la lista, recibirá una notificación señalando que el producto no se encuentra en su wishlist.

2.2. Conectar componente de subcategorías

Para conectar este componente, lo que se hizo fue desde la vista crear un atributo keywords en el data() el cual consiste en un array de strings que contienen las palabras clave para el filtrado.

Cada categoría tiene asociada una serie de palabras clave, por ejemplo, a la categoría principal Accesorios se le asocia el keyword .ªccesorio", y a la subcategoría Perros dentro de Accesorios se le asocian .ªccesorioz "perro". A su vez, a la subsubcategoría Collares contenida dentro de Perros y de la categoría Accesorios se le asocian las palabras clave .ªccesorio", "perroz çollar". Gracias a esta

asociación entre categorías y conjuntos de perros, lo que se hace es que cuando se selecciona una categoría, desde componente de Subcategorías se emite un evento con las keywords asociadas a dicha categoría y se maneja actualizando el keywords del data() de la vista y repitiendo la petición de items al backend.

Gracias a esto en el backend se le añade un primer filtrado por keywords, obteniendo los objetos que contienen todas las keywords pasadas. Gracias a esto, se consigue que al seleccionar una categoría principal, aparezcan los objetos de las subcategorías (dado que contendrán sí o sí asociada la categoría).

2.3. Posibilidad de actualizar imagen

Se modificó la forma en la que el POST y PUT trabajan con las imágenes (tanto en los productos como mascotas). En primer lugar se hizo que en lugar de trabajar con una única imagen, dar la posibilidad de trabajar con varias. Además, se ha hecho que a las imágenes que se suban, se les haga un recorte mediante parámetros que ofrece Cloudinary para dejarlas todas cuadradas y mantener una relación de aspecto concreta en todos los items. Las imágenes son almacenadas en una carpeta nombrada con el id único que ofrece MongoDB para el item y los identificadores de las imágenes son proporcionados por Cloudinary.

En el formulario que se envía tanto en la creación como en la modificación de un item, se adjuntan las imágenes y luego son trabajadas para su subida a Cloudinary. La forma de trabajar que se decidió para la modificación fue que primero se eliminen todas las imágenes y se suban de nuevo todas las que permanecieron y las nuevas tras la edición (ésto se explica mejor en el próximo apartado). Aparte de esto, el POST y el PUT funcionan de forma similar a anteriormente.

Se añadió manejo de las imágenes del DELETE. Lo que se hace es buscar recursos en Cloudinary que coincidan con el id del item y las id de las imágenes. Simplemente si los encuentra, se eliminan tanto dichos recursos como la carpeta asociada al item.

2.4. Previsualización de imágenes en edición de item

Ésto no entraba dentro de la planificación pero se implementó como mejora y para dar la posibilidad de tener una herramienta gráfica con la que poder modificar las múltiples imagenes de un determinado item. Para ello, se añadió dentro del dashboard un pequeño componente que itera las imágenes que se obtienen del item una vez se le da al botón de editar descargándolas haciendo una petición a Cloudinary y convirtiéndolas en ficheros de forma dinámica para poder cargarlas en ésta previsualización (siendo almacenados en un array que es reactivo con la lista) y tenerlas listas para volver a subirlas a Cloudinary (a excepción de las que se eliminen) junto con las nuevas subidas.

En la previsualización se muestran en una lista de tarjetas con las imágenes que tienen un botón que da la opción de eliminarlas (de este array de imágenes por subir). Las nuevas que se suben también se añaden al array y una vez se ha

terminado de editar y se envía el PUT, es enviado dicho array con las imágenes para ser manejado por le backend.

2.5. Github Actions

Se configuró un flujo de trabajo en GitHub Actions para que, cada vez que subo cambios (push) o hago una pull request al branch dev, se ejecuten pruebas automáticas y así asegurar que el código funciona correctamente.

El flujo de trabajo utiliza una máquina virtual basada en Ubuntu y define un entorno con Node.js (versión 21.x) y MongoDB (versión 7.0). Primero, se descarga el código con un checkout. Luego, se configura la versión de Node.js indicada, aprovechando una caché para optimizar la instalación de dependencias. Después, se inicia un servidor MongoDB con la versión especificada para las pruebas.

Una vez configurado el entorno, se instalan las dependencias en la carpeta backend utilizando el comando npm ci, que garantiza una instalación limpia. Finalmente, se ejecutan las pruebas del proyecto con el comando npm run test. Este flujo me ayuda a validar automáticamente que los cambios no rompan ninguna funcionalidad existente en el branch dev.

2.6. Mockup de Pruebas

Para hacer el mockup de pruebas utilizamos la libreria sinon, librería de javascript que nos permite interceptar las peticiones que se envían al modelo de datos moongose antes de que se interactúe con lo base de datos, reduciendo como consecuencia los tiempos de ejecución de las pruebas unitarias. La librería trabaja con stubs para simular (mockear) las interacciones con la base de datos y así aislar el código que se está probando. Por ejemplo, una función que consulta una base de datos, como es el caso de los proveedors se puede reemplazar esa consulta con un stub que devuelva datos simulados o lance errores, sin realizar una operación real. Esto se logra creando un stub de la función que interactúa con la base de datos (por ejemplo, db.find o db.save) y configurándolo para que devuelva valores predefinidos (stub.resolves(data)) o simule fallos (stub.rejects(error)).

Entre las principales dificultades encontradas, están el uso de stub porque cada endpoint tenía sus propias llamadas a la base de datos y había que escribir stub especificos para cada prueba, por ejemplo, el caso de añadir un producto es el más complicado de mockear porque existe un llamada a la API de almacenamiento de las imágenes que también tuvimos que mockear. Si realizaba un save() teníamos que escribir un stub para que devolviera el producto, simulando que se ha añadido en la base de datos. Se creo un stub para el servicio cloudinary para que cada vez que hiciera una petición a la API devolvería una URL fake, simulando que se ha añadido la imagen a la nube de cloudinary, esto nos complico el borrado porque como no se ha añadido la imágen no se puede borrar algo que no existe, por lo que tuvimos que mockear también la llamada a la función de borraro.

En general, usar datos mockeados tiene ventajas como la rapidez y simplicidad: las pruebas son más rápidas porque no dependen de sistemas externos como bases de datos reales, y puedes simular fácilmente diferentes escenarios, incluidos errores. Sin embargo, tiene desventajas, como el riesgo de no reflejar completamente el comportamiento del sistema real, ya que los datos simulados pueden no capturar todos los casos reales o problemas específicos del entorno de producción. Además, puede requerir tiempo extra para crear y mantener los mocks.

3. Panel del sprint 3 y backlog del spring 4

En el sprint anterior:

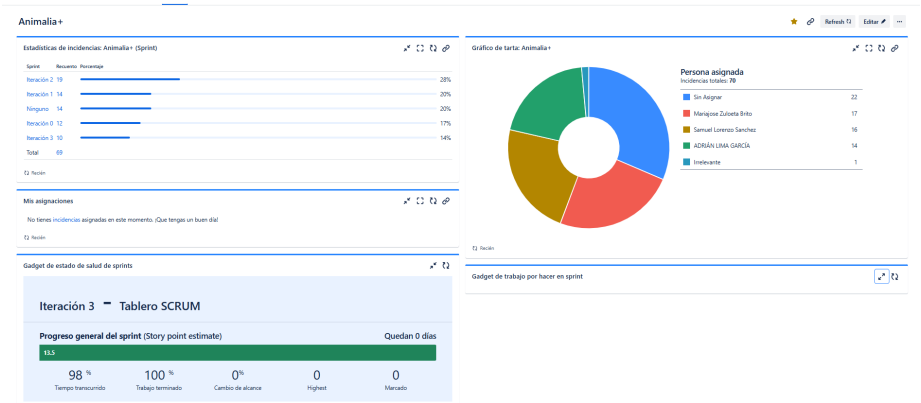

Figura 1: Imagen que muestra las cargas de trabajo de cada miembro del equipo, tareas asignadas al usuario y el estado de salud del spring

Figura 2: Imagen que muestra el spring burndown chart

Mientras que el backlog de la iteración 1 es:

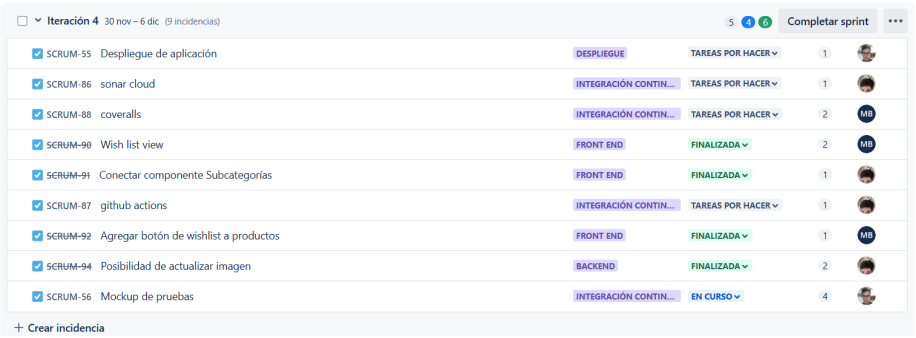

Figura 3: Imagen que muestra el Backlog del spring 1