APLICACIÓN DE GESTIÓN DE INVENTARIO

Proyecto de Fin de Curso
2023-2024

Alumno
Bin Chen

Tutor de Proyecto
Jaime Latorre

Desarrollo de Aplicaciones Multiplataforma
IES EL LAGO

Índice

1. Definición del Sistema

La *Aplicación de Gestión de Inventario* es una aplicación WEB *full stack* diseñada para ayudar a pequeñas y medianas empresas a llevar un control eficiente y preciso de sus inventarios.

1.1. Análisis de la utilidad y necesidad del sistema

La aplicación ofrece una visibilidad mejorada del inventario de la tienda a tiempo real que permite a los empleados a monitorizar de manera eficiente los niveles de *stock*, identificar rápidamente productos con baja disponibilidad y tomar acciones correctivas al respecto. Además, con las gráficas que genera la aplicación facilita la identificación de los productos en tendencia y planificar compras futuras más precisas asegurando una rotación de inventario óptima y reduciendo costes innecesarios.

1.2. Identificación del alcance del sistema

Funcionalidades incluidas:

- **Registro y Actualización de Productos.** Permite la creación de nuevos registros de productos y la actualización de la información existente de productos.
- **Gestión de Categorías.** Permite crear y gestionar categorías de productos para una mejor organización.
- **Gestión y Monitorización de Niveles de Inventario.** Permite añadir o reducir el nivel de stock y crea un informe gráfico de si misma.
- **Flexibilidad de la Información de los Productos.** Permite añadir campos de información de forma dinámica a los diferentes productos.
- **Funcionalidad de Búsqueda, Filtrado y Orden.** Permite la búsqueda de producto por nombre, código, etc, el filtrado por categorías y ordenar los productos por sus diferentes campos de información.
- **Migración de Datos.** Permite la migración de los todos los datos de la base de datos.
- **Seguridad y Control de Acceso.** Sistema de autenticación de usuarios y definición de diferentes roles con distintos niveles de acceso a las funcionalidades de la aplicación.
- **Escalabilidad y Adaptabilidad.** Implementación de un diseño modular que facilite añadir y adaptar funcionalidades según las necesidades de la tienda.

Funcionalidades excluidas:

- **Soporte Dispositivo Móvil.** El desarrollo actual de la aplicación web ha priorizado el uso de tecnologías y recursos que funcionan de manera óptima en entornos de escritorio. Esto implica que algún que otro componente de la aplicación no se

muestre correctamente en dispositivos móviles y que algunos elementos interactivos no funcionen como lo esperado.

- **Integración Directa con Sistemas de Pago.** El sistema no incluye funcionalidades de procesamiento de pagos.
- **Gestión de Clientes.** El sistema no incluye la gestión de clientes (*CRM*).

Limitaciones de la aplicación:

- **Dependencia de la Conectividad.** La aplicación requiere una conexión a Internet para su funcionamiento completo, lo que puede ser una limitación en áreas con conectividad deficiente.
- **Dependencia de *URLs* para Imágenes.** El sistema utiliza *URLs* para referenciar y cargar imágenes asociadas a los productos y esto puede presentar limitaciones para los usuarios que no están familiarizados con la estructura y gestión de *URLs*.
- **Utilización de *JSON* para Migración de Datos.** Para la migración de datos desde sistemas existentes el sistema utiliza *JSON*, que aunque es ampliamente aceptado y utilizado en el desarrollo web por su simplicidad y flexibilidad, su manipulación puede resultar complicada para usuarios sin experiencia técnica.

1.3. Identificación del entorno tecnológico y de las tecnologías a utilizar

Tecnologías utilizadas:

- **Lenguajes de Programación.** *HTML, CSS, JAVASCRIPT (VUE.js)* [1], [2] para el *frontend* y *RUST* [3] con *rocket* [4] para el *backend*.
- **Base de Datos.** *MySQL*, sistema de gestión de bases de datos relacional y *sqlx* para la conexión entre el *backend* y la base de datos.
- **Herramientas de Desarrollo.** *Visual Studio Code* como *IDE*, *Postman* para la prueba del backend y *DBeaver*, interfaz para administrar base de datos.
- **Despliegue de Aplicación.** Contenedores *Docker* [5], [6] para facilitar la implementación y la escalabilidad.
- **Otros.** *Axios*, cliente *HTTP* que se usa para realizar solicitudes *HTTP* desde el *frontend* hacia el *backend* y *JWT(Json Web Token)* [7] para la autenticación y seguridad durante la comunicación entre *back* y *front*.

1.4. Planificación de las fases del desarrollo del proyecto

La planificación del proyecto (Figura 1) se organiza en diferentes fases, cada uno con sus objetivos específicos:

1. **Análisis y Requisitos.** Elaboración de los requisitos del proyecto, realizar el análisis del sistema y definir el alcance del proyecto.
2. **Diseño del Sistema.** Diseño de la arquitectura del sistema (*frontend* y *backend*) y la interfaz de usuario y definir la base de datos.
3. **Desarrollo del *Backend*.** Implementación de la lógica del servidor y los *endpoints* necesarios.
4. **Desarrollo del *Frontend*.** Implementación de la interfaz de usuario y la lógica del cliente.
5. **Integración y Pruebas.** Integrar el *frontend* y el *backend* y realizar pruebas del sistema completo.
6. **Despliegue.** Desplegar el sistema en el entorno de producción y asegurar su correcto funcionamiento.

Desde el análisis de requisitos y el diseño del sistema hasta el desarrollo, integración, pruebas y despliegue, cada etapa está orientada a garantizar que el sistema final cumpla con las expectativas y necesidades de los usuarios.

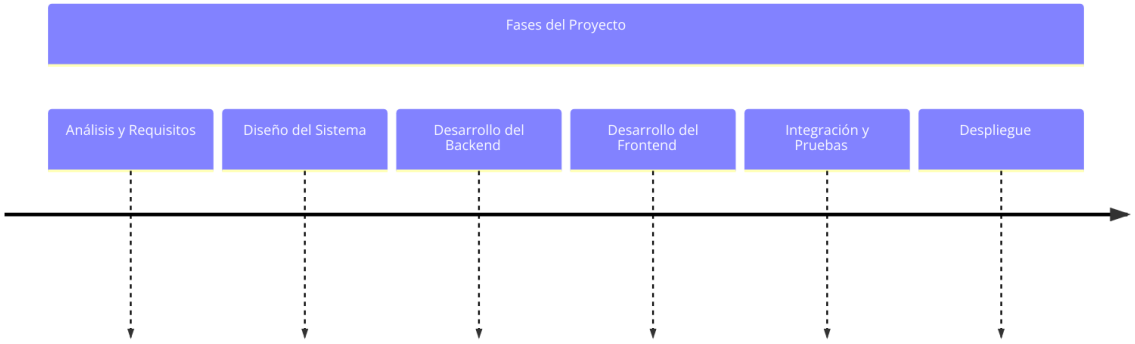

Figura 1: Diagrama de las fases de desarrollo

2. Análisis del sistema

2.1. Especificación de requisitos

Requisitos funcionales

- **Gestión de Productos**

  ▸ **Añadir producto:** El sistema debe permitir al usuario añadir nuevos productos, ingresando datos como el nombre, descripción, imagen, etc.

  ▸ **Modificar producto:** Los usuarios deben poder actualizar la información de los productos existentes.

  ▸ **Eliminar producto:** Debe ser posible eliminar productos.

  ▸ **Asignar categoría a producto:** Los usuarios deben poder relacionar productos a categorías.

- **Gestión de Categorías**

  ▸ **Crear categoría:** La aplicación debe permitir al usuario crear nuevas categorías.

  ▸ **Eliminar categoría:** Debe ser posible eliminar categorías.

  ▸ **Mostrar gráfica categorías:** La aplicación debe tener una gráfica mostrando al usuario el número de productos que pertenece a cada categoría.

- **Gestión y Monitorización de Niveles de Inventario**

  ▸ **variar *stock* de los productos:** La aplicación debe permitir al usuario añadir o reducir el número de *stock* de los productos.

  ▸ **Mostrar gráfica de *stock*:** La aplicación debe tener una gráfica mostrando al usuario la variación de *stock* en término de años de cada producto.

  ▸ **Mostrar últimas variaciones de *stock*:** La aplicación debe mostrar los 15 últimos cambios en el *stock* en forma de tabla.

- **Flexibilidad de la Información de los Productos**

  ▸ **Añadir propiedad:** La aplicación debe permitir al usuario introducir nuevas propiedades.

  ▸ **Borrar propiedad:** La aplicación debe permitir al usuario borrar las propiedades de un producto.

- **Funcionalidad de Búsqueda, Filtrado y Orden**

  ▸ **Búsqueda de producto:** La aplicación debe permitir al usuario buscar un producto por sus campos, por ejemplo por nombre.

  ▸ **Filtrado de productos:** La aplicación debe permitir al usuario filtrar productos, por ejemplo mostrar únicamente los productos con una categoría especificada por el usuario.

‣ **Ordenar productos:** La aplicación debe dar la posibilidad al usuario de ordenar los productos por sus campos.

- **Migración de Datos**

  ‣ **Exportar datos:** La aplicación debe tener la opción de exportar los datos guardados en formato *JSON*.

  ‣ **Importar datos:** La aplicación debe tener la opción de importar datos desde un fichero *JSON* generada por ella misma.

- **Seguridad y Control de Acceso** ‣ **Registrar usuario:** La aplicación debe dejar únicamente al usuario con rol administrador crear nuevos usuarios.

  ‣ **Iniciar sesión:** La aplicación debe tener un sistema de inicio de sesión.

  ‣ **Cerrar sesión:** La aplicación debe dejar la opción de cerrar sesión al usuario.

  ‣ **Cambiar contraseña:** La aplicación debe obligar al usuario a cambiar su contraseña en su primer inicio de sesión.

  ‣ **Control de acceso por roles:** La aplicación debe restringir algunas funcionalidades dependiendo del rol que tenga el usuario.

REQUISITOS NO FUNCIONALES

- **Rendimiento**

  ‣ **Tiempo de respuesta:** Las operaciones comunes, como el registro de productos, actualización de *stock* y generación de informes, deben completarse en menos de 1 segundo.

  ‣ **Escalabilidad:** La aplicación debe soportar la adición de nuevas funcionalidades y módulos y debe manejar un incremento en el número de registros y transacciones con un impacto mínimo en el rendimiento.

- **Fiabilidad**

  ‣ **Disponibilidad:** La aplicación debe asegurar que esté disponible para los usuarios con un tiempo de inactividad mínimo.

  ‣ **Tolerancia a fallos:** La aplicación debe ser capaz de manejar fallos y recuperarse sin pérdida de datos ni interrupciones significativas del servicio.

- **Seguridad**

  ‣ **Protección de datos:** Los datos sensibles como las contraseñas deben ser cifradas usando algoritmos secretos.

  ‣ **Autenticación y autorización:** Se debe utilizar un sistema de control por roles y JWT para gestionar permisos y para una autenticación segura.

- **Mantenibilidad**

  ▸ **Modularidad:** El diseño debe ser modular permitiendo que los componentes individuales puedan ser desarrollados, mantenidos y actualizados de manera independiente.

2.2. Especificación de estándares y normas de aplicación en el proyecto

Estándares de codificación

▸ **Vue.js:**

– Seguir las convenciones de estilo de código recomendadas por la guía de estilo oficial de Vue.js.

– Utilizar extensiones del *Visual Studio Code* de *Vue* para formatear el código y análisis de código.

▸ ***Rust (rocket):***

– Seguir las convenciones de estilo de código recomendadas por la guía oficial de *Rust* y usando la extensión de *rust-analyzer*.

– Utilizar *rustfmt* para formatear el código y *rust_analyzer* para el análisis estático del código.

– Mantener una estructura de código modular utilizando *mod* para la claridad del código y facilitar la expansión del proyecto.

Estándares de usabilidad y accesibilidad

▸ **Diseño de interacción:** Realizar pruebas de usabilidad con usuarios finales para validar la facilidad de uso del sistema

Normas de Calidad

▸ **Pruebas de integración:** Realizar pruebas de integración para verificar el correcto funcionamiento de la aplicación WEB.

▸ **Pruebas de carga y estrés** Realizar pruebas de carga y estrés para verificar el correcto funcionamiento de la aplicación con un alto volumen de tráfico.

3. Diseño del sistema

El proyecto utiliza una arquitectura cliente-servidor (Figura 2) se constituye de los siguientes componentes:

- ***Frontend*:** Utilizando el *framework* de *Vue* y codificado en *HTML, CSS y Javascript*.
- ***Backend*:** Utilizando el *framework* de *Rocket* y codificado en *Rust*.
- **Base de datos:** Sistema de gestión de base de datos *MySQL*.

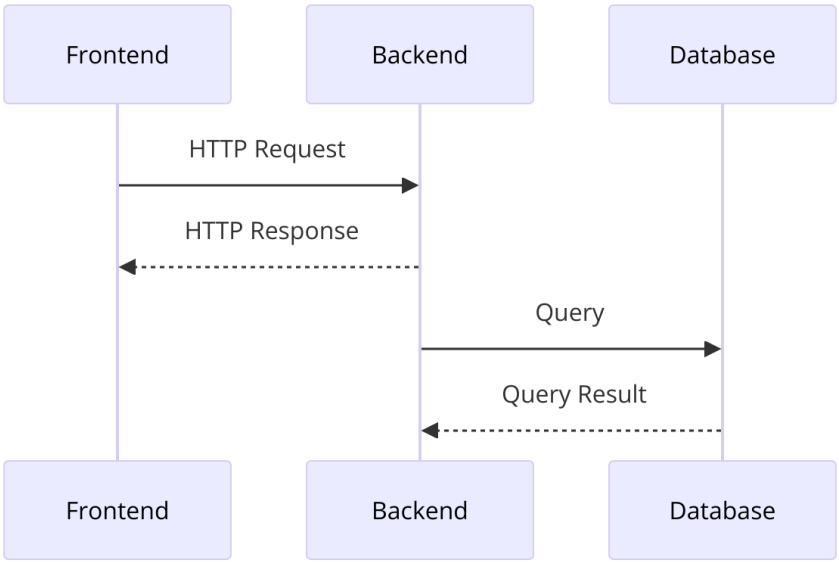

Figura 2: arquitectura cliente-servidor

3.1. Diseño de interfaces de usuario

Página de Inicio de Sesión (Figura 3)

- Elementos:
  - *Input* para introducir nombre de usuario.
  - *Input* para introducir la contraseña.
  - Botón para iniciar sesión.
- Descripción general: Página simple compuesta por un contenedor central que contiene un formulario para iniciar sesión.

Página Inicio (Figura 4)

- Elementos:
  - Buscador de productos.
  - Caja de selección para ordenar los productos.
  - Botón para ir al formulario de creación de producto.

  - Lista de productos.

- Descripción general: Página que contiene una lista de productos mostrando información básica de ellas en cajas pequeñas, en la parte superior se encuentra la barra de herramientas constituido por componentes interactivos.

Encabezamiento

- Elementos:
  - Logo de la aplicación.
  - Navegador de páginas.
  - Botón de cerrar sesión con el nombre de usuario correspondiente.
- Descripción general: Barra superior fija que está en cada página de la aplicación que sirve para navegar a cada una de las páginas y para cerrar sesión.

Página de Categorías (Figura 5)

- Elementos:
  - Buscador de categorías.
  - Lista de categorías.
  - formulario para añadir categorías.
  - Gráfica del número de productos de cada categoría.
- Descripción general: Página que contiene los elementos anteriores de arriba hacia abajo en orden: buscador, lista, formulario, gráfica.

Página de Variación de Inventario (Figura 6)

- Elementos:
  - Formulario para añadir *stock* a los productos.
  - Gráfica del inventario de los productos.
  - Tabla con los últimos cambios en inventario.
- Descripción general: La página está dividida en tres partes, la primera para añadir *stock* a los productos, la segunda con una gráfica de puntos y la tercera la tabla de las variaciones.

Registrar Usuarios (Figura 7)

- Elementos:
  - *Input* para introducir el nombre de usuario.
  - *Input* para introducir la contraseña.
  - *Input* para introducir la contraseña repetida.
  - Caja de selección para elegir el rol que va a tener el usuario creado.

  - Botón para crear el usuario.
- Descripción general: Página simple que tiene un contenedor central con el formulario constituido por los componentes.

Migración de datos (Figura 8)

- Elementos:
  - Botón para exportar los datos de la base de datos.
  - Botón para seleccionar el fichero *JSON* que contiene la información que se va a importar a la base de datos.
  - Contenedor desplazable en la que se muestra el contenido del *JSON*.
  - Botón de importar datos.
- Descripción general: Página simple en la que contiene botones y otros elementos para importar y exportar datos.

Cambiar Contraseña (Figura 9)

- Elementos:
  - *Input* para introducir contraseña.
  - *Input* para repetir contraseña.
  - Botón para cambiar contraseña.
- Descripción general: Página simple con un contenedor central en la que se encuentra el formulario para cambiar la contraseña.

Añadir Producto (Figura 10)

- Elementos:
  - *Inputs* para introducir información del producto.
  - Botón para crear producto.
- Descripción general: Contenedor modal con un formulario para crear productos.

Mostrar Detalles del Producto (Figura 11)

- Elementos:
  - Columna información básica del producto.
  - Columna de las categorías a las que pertenece el producto.
  - Columna con las propiedades personalizadas del producto.
  - Botón de borrar y modificar.
- Descripción general: Contenedor modal con tres columnas mostrando información detallada del producto.

Modificar información del producto (Figura 12)

- Elementos:
  - ▸ Columna con *inputs* para cambiar la información básica.
  - ▸ Columna con una caja de selección para añadir las categorías.
  - ▸ Columna con una tabla y con un formulario para añadir una propiedad personalizada.
  - ▸ Botón de borrar y de guardar cambios.
- Descripción general: Componente modal sobre el modal de mostrar detalles de producto con algunos componentes para modificar los datos.

3.2. Imágenes de las Páginas

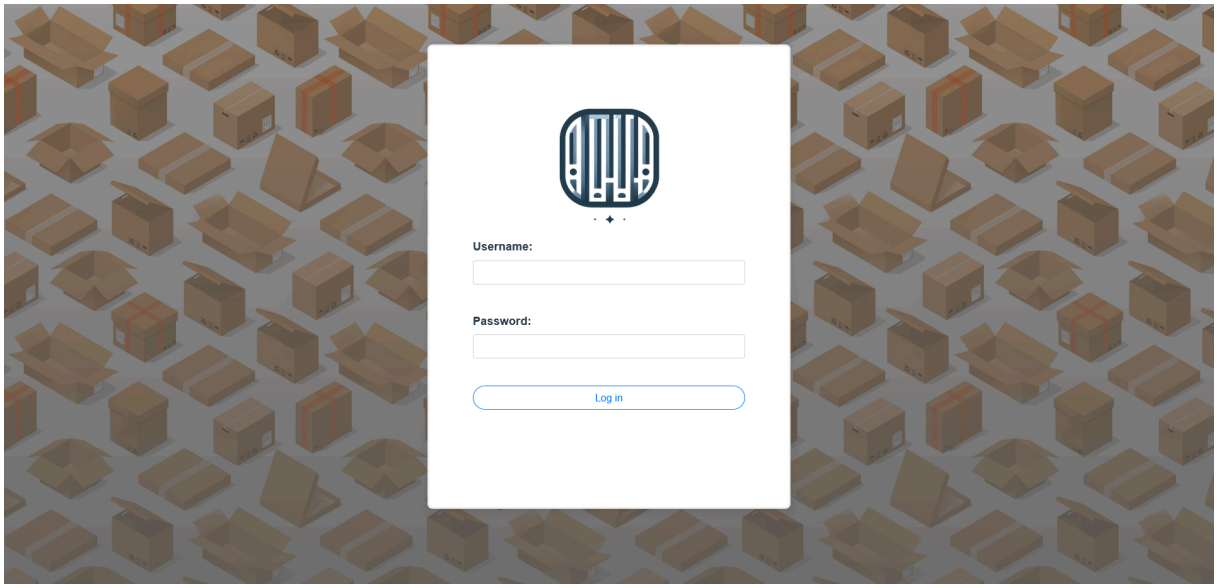

Figura 3: página de inicio de sesión

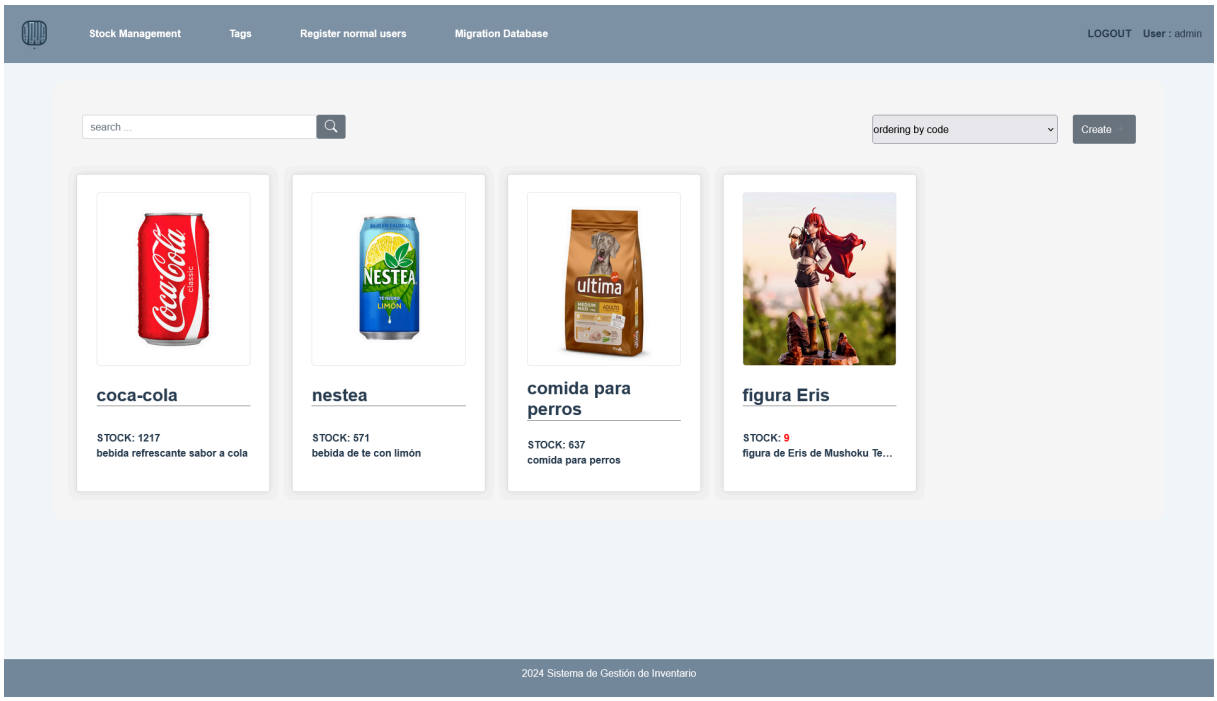

Figura 4: página principal de la aplicación

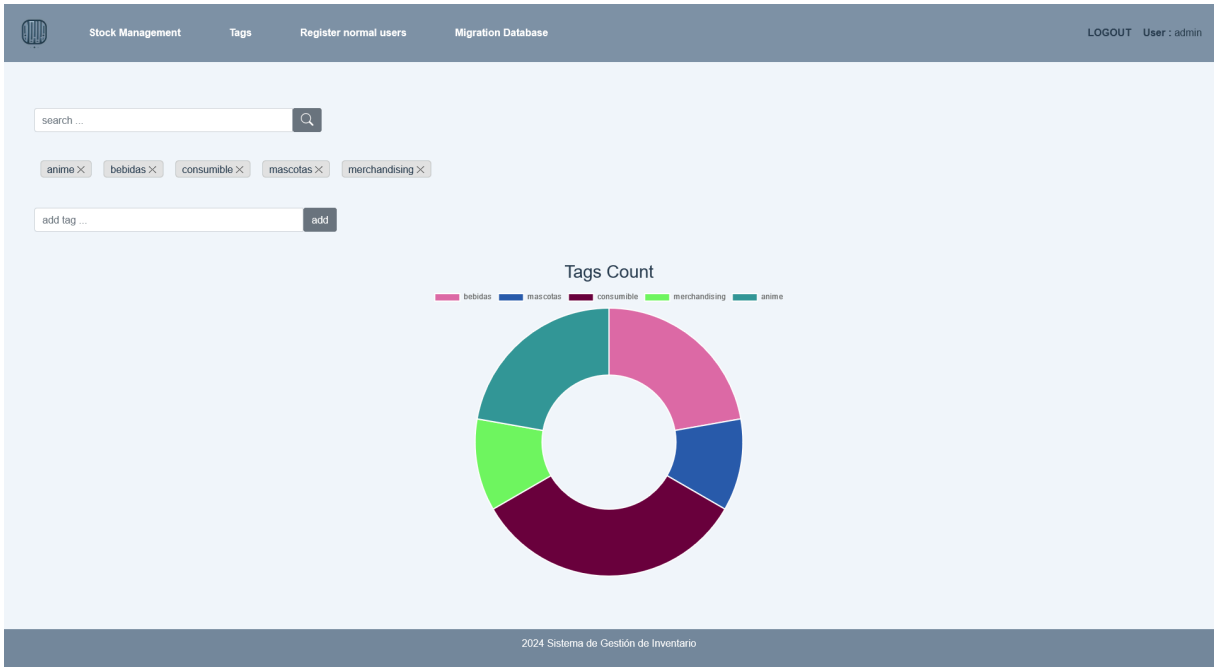

Figura 5: página de categorías

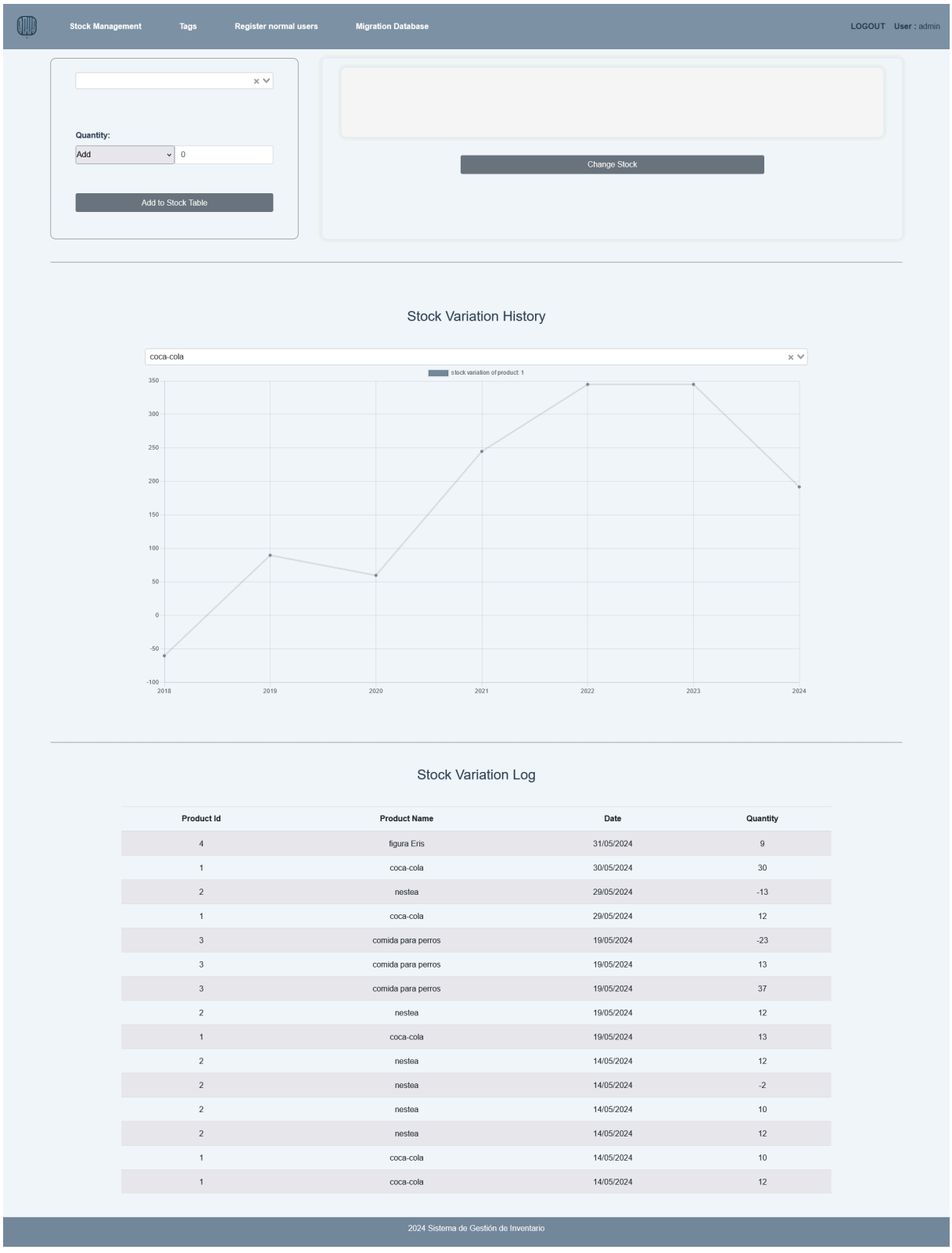

Figura 6: página de variación de inventario

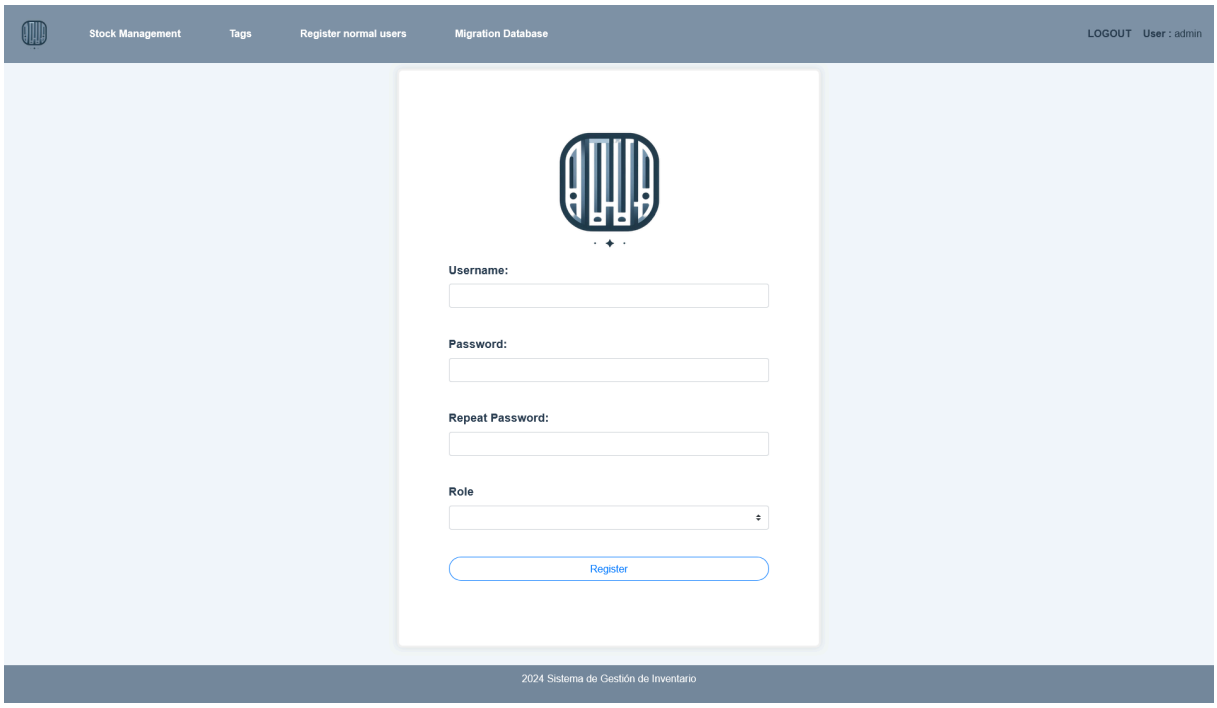

Figura 7: registrar usuario nuevo

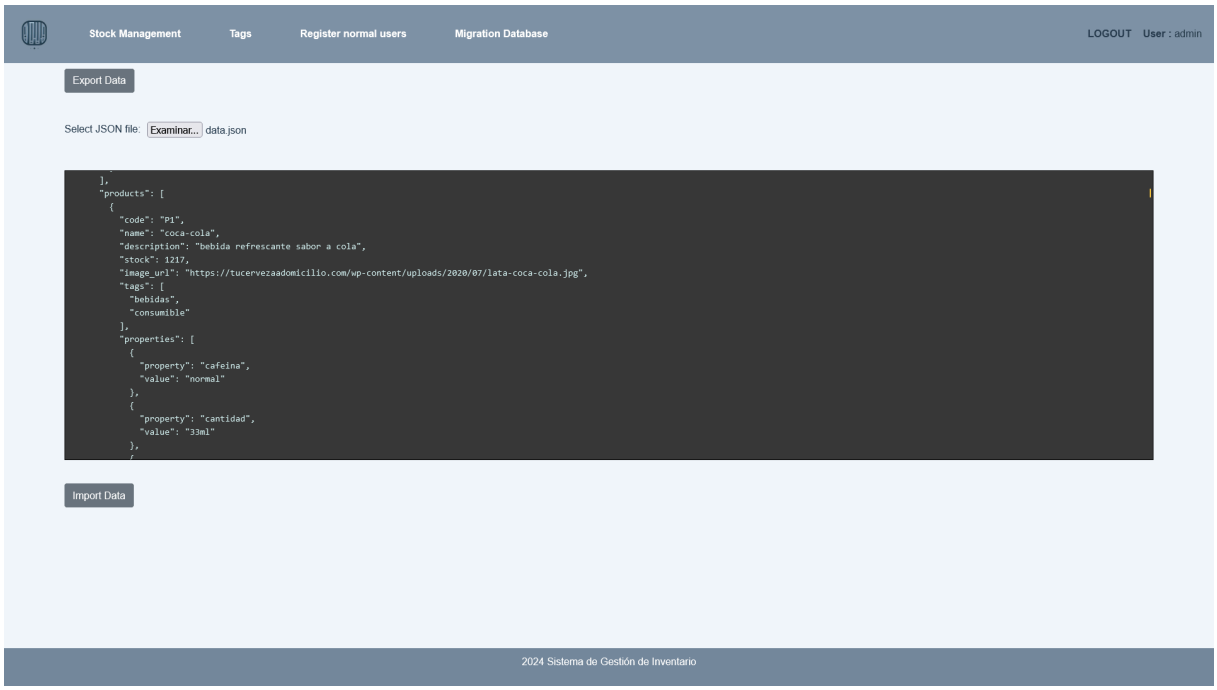

Figura 8: migración de datos

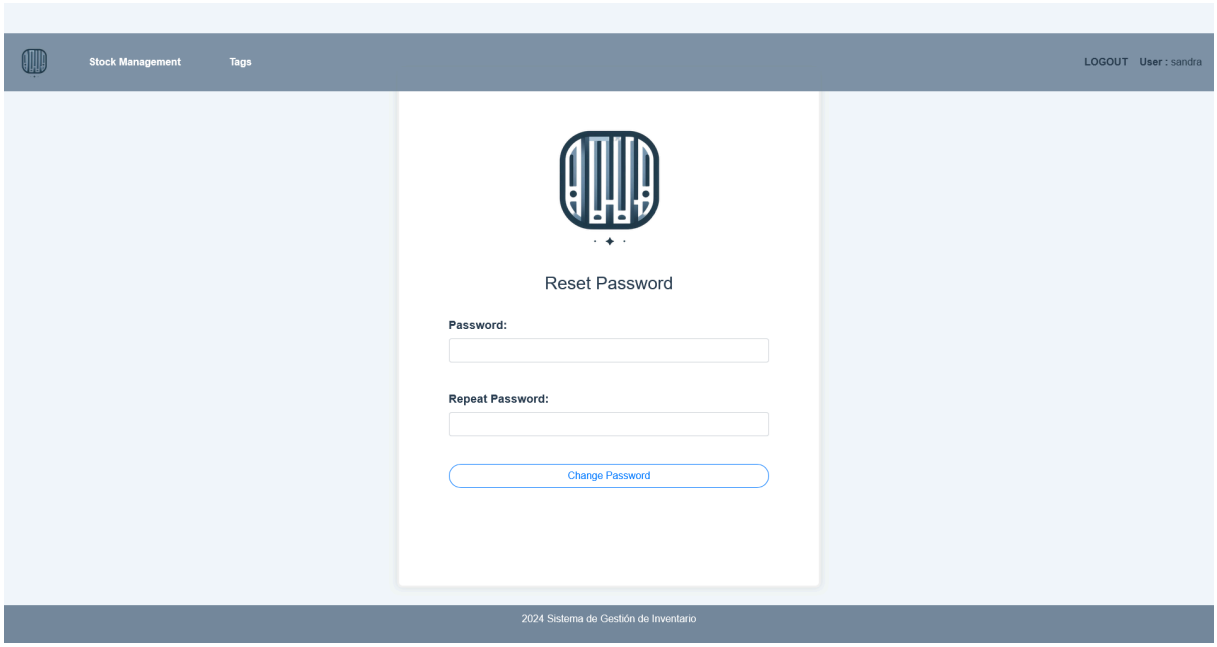

Figura 9: cambiar contraseña

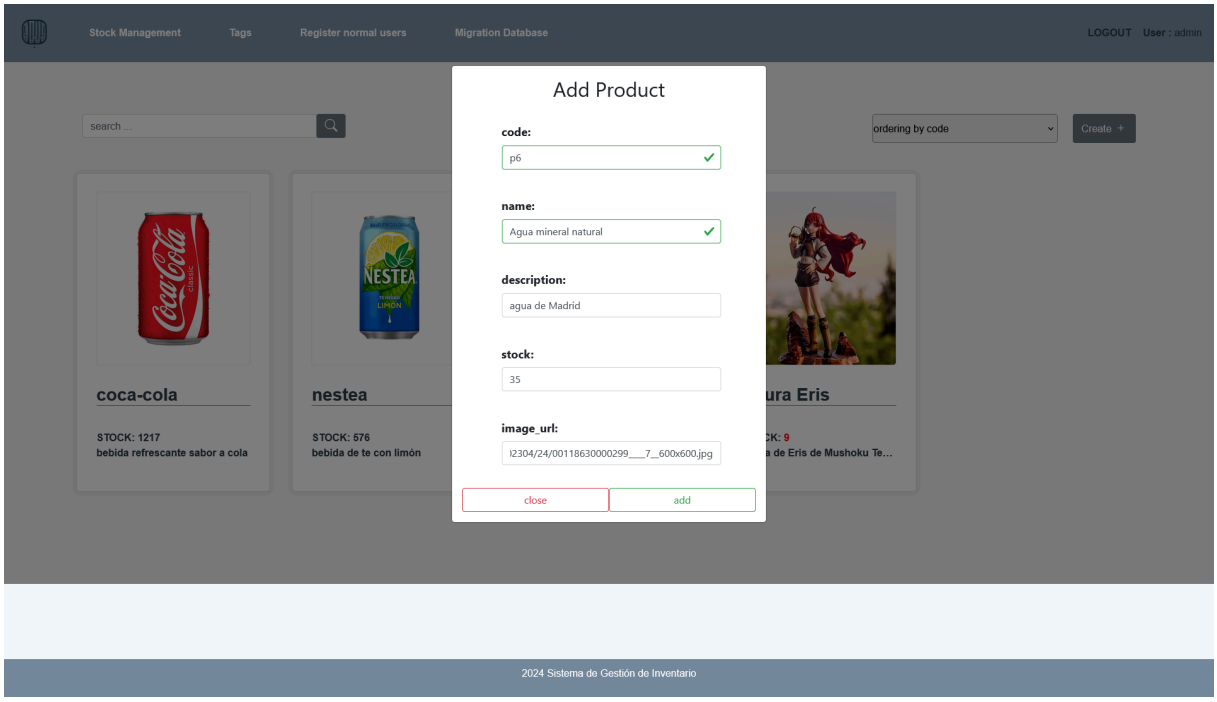

Figura 10: crear producto

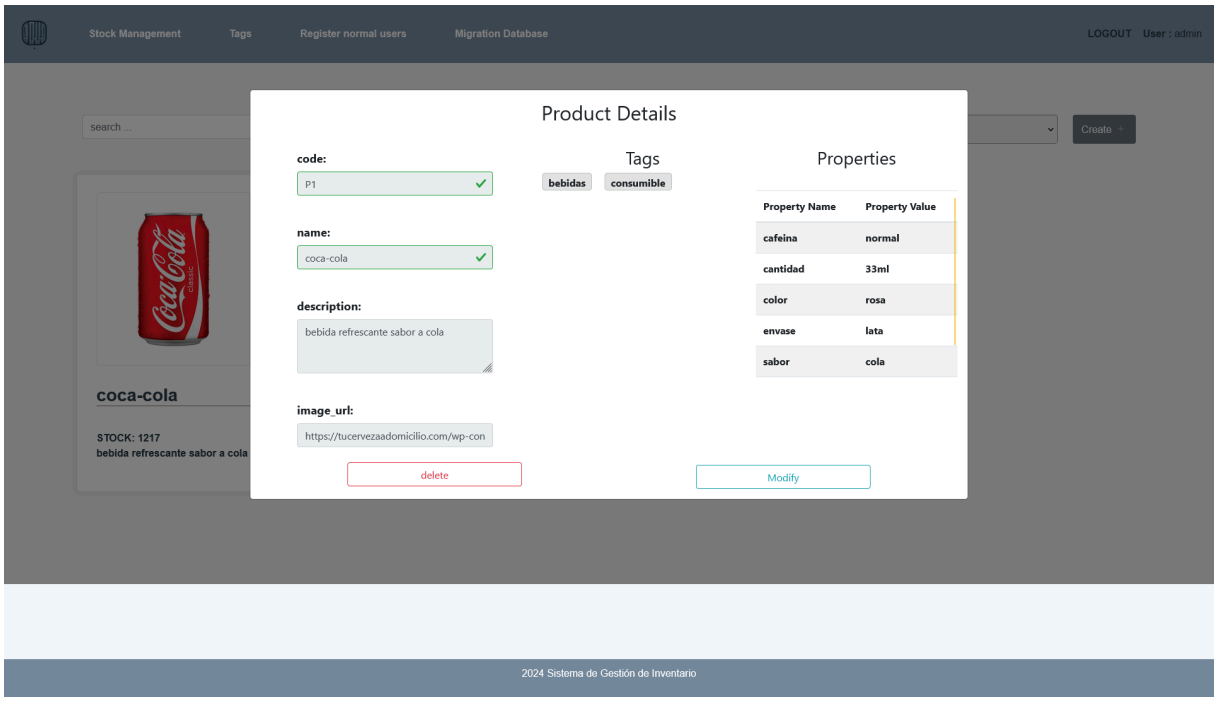

Figura 11: mostrar detalles de producto

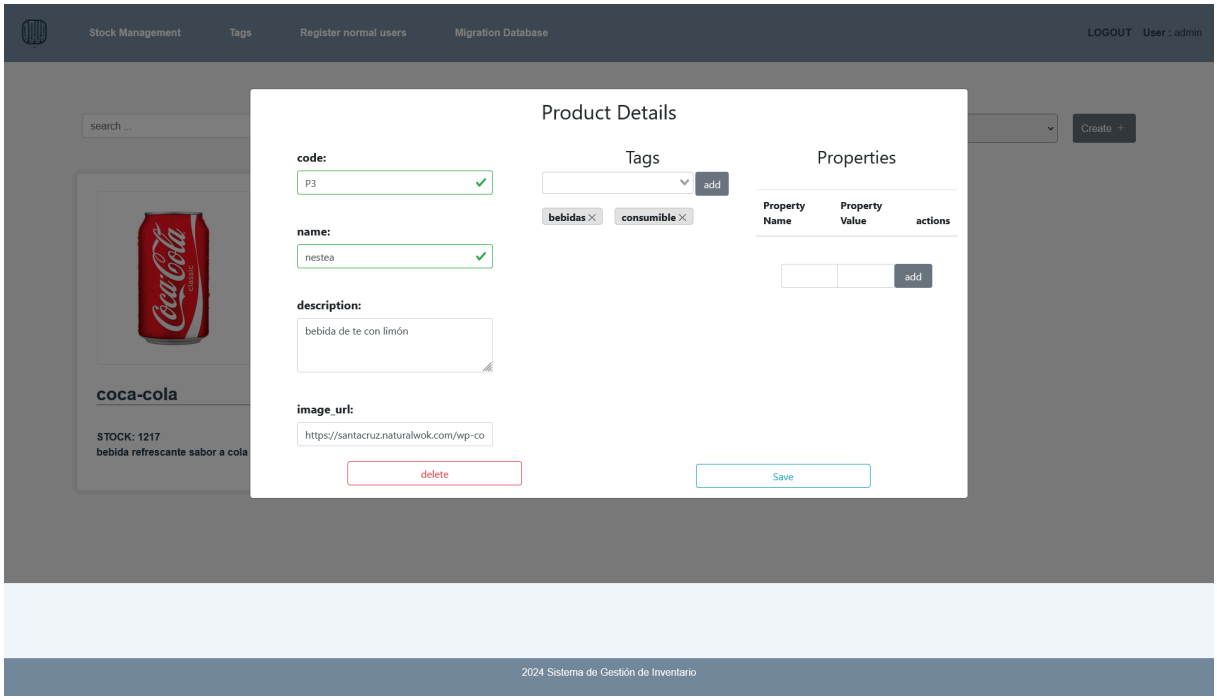

Figura 12: modificar información del producto

3.3. Diseño del modelo de datos

La base de datos (Figura 13) consta de varias tablas interrelacionadas. A continuación, se detallan las principales entidades y su propósito dentro del sistema:

- **Tabla usuarios**: Almacena la información de los usuarios del sistema, incluyendo su nombre de usuario único, contraseña cifrada, rol en el sistema y un indicador para el primer inicio de sesión.
- **Tabla productos**: Contiene los detalles de los productos en el inventario, como su código único, nombre, descripción opcional, cantidad en inventario actual y ruta de la imagen asociada.
- **Tabla categorías**: Define las categorías que se pueden asignar a los productos para su clasificación y organización.
- **Tabla propiedades**: Permite almacenar propiedades adicionales personalizadas para los productos, facilitando la flexibilidad en la descripción de los mismos.
- **Tabla productos a categorías**: Establece una relación muchos a muchos entre productos y categorías, permitiendo que un producto pertenezca a múltiples categorías.
- **Tabla variación de inventario**: Registra las variaciones de inventario de los productos a lo largo del tiempo, ayudando a realizar un seguimiento histórico de los cambios de inventario.

```
create table if not exists users(
  id int auto_increment primary key,
  username varchar(30) not null UNIQUE,
  pwd varchar(255) not null,
  rol varchar(30) not null,
  first_login tinyInt not null default 1
);

create table if not exists products(
  id int auto_increment primary key,
  code varchar(30) not null UNIQUE,
  name varchar(30) not null,
  description varchar(50) default "",
  stock int default 0,
  image_url varchar(255) default ""
);

create table if not exists tags(
  id int auto_increment primary key,
  name varchar(30) not null UNIQUE
);

create table if not exists properties(
  productid int,
  property varchar(30),
  value varchar(30) default "",
  primary key (productid, property),
```

```
    foreign key (productid) references products(id)
);

create table if not exists productsTotags(
  productID int,
  tagID int,
  primary key(productID,tagID),
  foreign key (productID) references products(id) on delete cascade on update cascade,
  foreign key (tagID) references tags(id) on delete cascade on update cascade
);

create table if not exists stockVar(
  varID int auto_increment primary key,
  productID int not null,
  varDate date not null,
  quantity int not null,
  foreign key (productID) references products(id) on delete cascade on update cascade
);
```

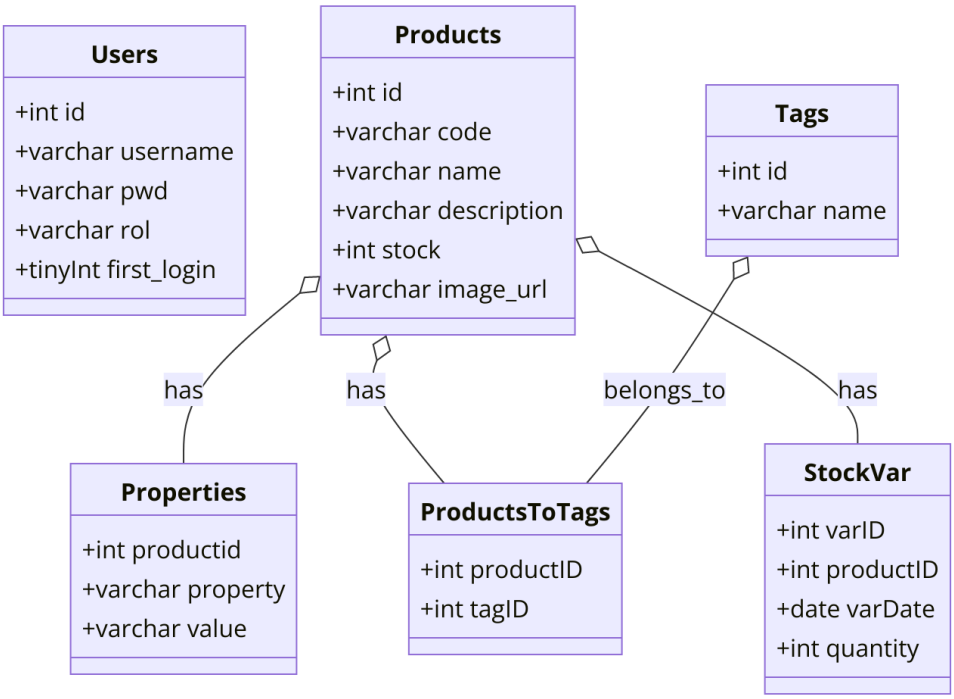

Figura 13: diagrama de clases de la base de datos

3.4. Diseño *Backend*

El *backend* de nuestra aplicación de gestión de inventario se basa en una *API RESTful* implementada en *Rust* con el *framework Rocket*. Esta *API* proporciona *endpoints* claves que permiten la interacción fluida y segura con los datos del sistema:

- ***POST*** **/api/register** : registro de usuario.
- ***POST*** **/api/login** : inicio de sesión de usuario.
- ***POST*** **/api/get_products** : extraer datos de productos.
- ***POST*** **/api/add_product** : añadir producto.
- ***POST*** **/api/delete_product** : borrar producto.
- ***POST*** **/api/modify_product** : modificar producto.
- ***POST*** **/api/add_property** : añadir propiedad personalizada.
- ***POST*** **/api/delete_property** : borrar propiedad.
- ***POST*** **/api/get_tags** : extraer datos de las categorías.
- ***POST*** **/api/add_tag** : añadir categoría.
- ***POST*** **/api/delete_tag** : borrar categoría.
- ***POST*** **/api/bind_tag** : enlazar categoría con producto.
- ***POST*** **/api/unbind_tag** : desenlazar categoría con producto.
- ***POST*** **/api/var_stock** : variar inventario de producto.
- ***POST*** **/api/get_stocks** : extraer variación de inventario de un producto por año.
- ***POST*** **/api/get_stock_history** : extraer las últimas 15 variaciones.
- ***POST*** **/api/get_popular_tags** : extraer la cantidad de productos enlazados con cada categoría.
- ***POST*** **/api/import_data** : importar datos.
- ***POST*** **/api/export_data** : exportar datos a *JSON*.
- ***POST*** **/api/reset_pwd** : cambiar contraseña.

La estructura de la implementación está organizado en módulos y cada *endpoint* utiliza diferentes métodos y estructura de datos personalizados de cada módulo.

4. Aspectos relevantes de la implementación

La implementación abarca varios aspectos técnicos y funcionales claves que aseguran la eficiencia, robustez y escalabilidad de la aplicación. A continuación, se destacan los aspectos más relevantes:

- **Arquitectura de la Aplicación**

  ▸ **Cliente-Servidor.** La aplicación utiliza una arquitectura cliente-servidor, donde el *frontend* y el *backend* son completamente independientes entre sí, permitiendo una mayor flexibilidad.

  ▸**API RESTful.** El *backend* presenta una *API RESTful* que facilita la interacción con la base de datos.

  ▸ **Base de datos.** Se utiliza *sqlx* [8], una biblioteca para la integración con *MySQL*, facilitando la interacción con la base de datos a través de consultas seguras y eficientes.

  ▸ **Parseo de datos provenientes del *front*.** Se utiliza la biblioteca de *serde* [9] para la conversión de los datos de entrada a una estructura personalizada de *Rust*.

- **Desarrollo del *Backend***

  ▸ **Lenguaje y *framework*.** El *backend* está implementado en *Rust* utilizando el *framework Rocket*, conocido por su alto rendimiento y seguridad.

  ▸ **Autenticación y autorización con *JWT*.** El uso de *JSON Web Token* garantiza un acceso seguro y controlado a la base de datos.

  ▸ **Tokio.** El *backend* utiliza *tokio* [10] para hacer asíncrona las peticiones.

- **Desarrollo del *Frontend***

  ▸ ***Framework* de *frontend*.** *Vue* ofrece una interfaz de usuario dinámica y reactiva.

  ▸ **Componentes reutilizables.** La organización por componentes de *Vue* mejora la mantenibilidad del código.

  ▸ ***Axios*.** *axios* permite la comunicación mediante llamadas *HTTP* del *frontend* con *APIs*.

  ▸ **Chart.js.** [11] uso de vue-chart.js para crear gráficas.

- **Seguridad**

  ▸ **Cifrado de contraseñas.** Con dependencias de *Rust* como argon2 [12] se almacenan las contraseñas de forma cifrada añadiendo una capa adicional de seguridad.

  ▸ **Control de Acceso.** Se realiza tanto como el *front* como el *back* utilizando el *payload* del *JWT* del usuario en la que se almacena el rol y se verifica.

5. Pruebas de Accesibilidad y Usabilidad

A continuación se detallan estrategias y pruebas realizadas para asegurar que la aplicación sea accesible y fácil de usar y mantener un código limpio para futuras expansiones:

- **Pruebas funcionales manuales:** Se realiza principalmente pruebas de integración, de sistema y de aceptación para garantizar el funcionamiento correcto de la aplicación y identificar errores tempranos existentes.
- **Pruebas de rendimiento:** Se inserta una gran cantidad de datos a la base de datos y se comprueba el comportamiento del sistema verificando su correcto funcionamiento y rendimiento, asegurando este último que tarde menos de 200 milisegundos.
- **Contraste de colores:** Se utiliza herramientas para identificar si la combinación de colores permite al usuario ver de forma clara el contenido.
- **Pruebas de penetración (*Pentesting*):** Se simula ataques contra la aplicación dando al evaluador diferente cantidad de información de la aplicación evitando explotaciones de vulnerabilidad posibles:

  ▸ Pruebas de caja negra: Simula un ataque de un intruso sin conocimientos internos de la aplicación.

  ▸ Pruebas de caja gris: Simula un ataque de alguien con acceso limitado al sistema, como un usuario con privilegios básicos.

  ▸ Pruebas de caja blanca: Simula un ataque de alguien con conocimientos detallados del sistema.
- **Encuestas y feeedback:** Se recopila encuestas y comentarios de usuarios con diferentes roles para mejorar la experiencia general, facilidad de uso y cualquier dificultad encontrada.

6. Despliegue del proyecto

El despliegue del proyecto se ha realizado con *Docker* organizando el código con la siguiente estructura:

```
TFG_DAM/
├── back-end/
│   ├── Dockerfile
│   ├── src/
│   │   ├── main.rs
│   │   └── ...
│   ├── Cargo.toml
│   └── ...
├── frontend/
│   ├── Dockerfile
│   ├── src/
│   │   ├── main.js
│   │   ├── App.vue
│   │   └── ...
│   └── ...
├── mysql/
│   ├── init.sql
│   └── ...
└── docker-compose.yml
```

Dockerfile del *frontend*

```
FROM node:lts-alpine
# make the 'app' folder the current working directory
WORKDIR /app
# copy 'package.json' to install dependencies
COPY package*.json ./
# install dependencies
RUN npm install
# copy files and folders to the current working directory
COPY . .
EXPOSE 8080
CMD [ "npm", "run", "serve"]
```

Dockerfile del *backend*

```
FROM rust:1.78.0-bullseye as builder
# make the 'tfg-backend' folder the current working directory
WORKDIR /usr/src/tfg-backend
# copy the code to the container
COPY . .
# install dependencies
RUN rustup default nightly && cargo install --path .
```

```
FROM debian:bullseye-slim
# compile and remove unnesesary dependencies files
RUN apt-get update && rm -rf /var/lib/apt/lists/*
# copy the rocket app executable
COPY --from=builder /usr/src/tfg-backend/target/release/rocket /rocket
EXPOSE 8000
CMD ["/rocket"]
```

Docker compose de la aplicación

```
services:

  mysql:
    container_name: my_database
    image: mariadb
    restart: always
    environment:
      - MYSQL_ROOT_PASSWORD=XXXXXXXXX
      - MYSQL_DATABASE=my_database
    volumes:
      - $PWD/my_database:/var/lib/mysql
      - type: bind
        source: ./mysql/
        target: /docker-entrypoint-initdb.d

    ports:
      - 3306:3306

  rocket:
    container_name: rocket
    build: ./tfg-backend
    links:
      - mysql
    depends_on:
      - mysql
    environment:
      - DATABASE_URL=mysql://root:@my_database:3306/my_database
      - SECRET=XXXXXXXXXX
      - SQLX_OFFLINE=true
    ports:
      - 8000:8000

  vue-app:
    container_name: vue_app
    build: ./tfg-frontend
    depends_on:
      - rocket
    ports:
      - 8080:8080
```

Este archivo define tres servicios: *rocket, vue-app* y *mysql*.

- ***rocket:*** Expone el puerto 8000 y depende del servicio *mysql*.
- ***vue-app***: Expone el puerto 8080 y depende del servicio *backend*.
- ***mysql***: Utiliza una imagen oficial de mariadb y configura una base de datos para la aplicación, expone el puerto 3306.

Para el despliegue del proyecto es necesario tener instalado el comando *docker* y realizar una llamada al comando *docker compose up -d* en una terminal, se crean 3 contenedores que contienen cada servicio. Si es necesario el *feedback* de cada servicio es recomendable quitar la opción de -d para ver más detalles del despliegue y de cada petición que se realiza mientras la aplicación está en funcionamiento.

7. Problemas encontrados

Problemas de versiones con *Bootstrap* y *Vue*

En la integración de *Bootstrap* [13] con Vue.js, surgió un problema debido a la incompatibilidad de versiones entre *Bootstrap* y ciertos componentes de Vue.js. La versión utilizada de *Bootstrap* ha provocado conflictos con los componentes de *Vue*, resultando en comportamientos inesperados y estilos que no se aplicaban correctamente.

- **Solución**: actualización de *Bootstrap* a una versión compatible con Vue.js y arreglando el código de algunos componentes que perdieron su funcionalidad después de la actualización.

Problemas de tipado, especialmente con fechas y nulos en *Rust*

Durante el desarrollo del *backend* en *Rust*, los problemas más frecuentes estuvieron relacionados con el manejo de tipos, especialmente con fechas y valores nulos. *Rust* es estricto en cuanto al manejo de nulos y tipos, lo que llevó a errores al manipular fechas, específicamente con la conversión automática de *JSON* a una estructura personalizada con *serde*, o al intentar manejar valores nulos en estructuras de datos complejas.

- **Solución**: Añadir implementaciones de conversión a las estructuras para su automatización y usar *Optionals* para el manejo de nulos.

Problemas con *docker* y el *frontend*: redireccionamiento del *proxy*

Al configurar *Docker* para el despliegue del *frontend*, se enfrentó un problema crítico con el redireccionamiento del *proxy*. La configuración del *proxy* dentro del entorno Docker no se integró correctamente con la configuración de rutas y *endpoints* del *frontend*. Como resultado, las solicitudes *HTTP* desde el *frontend* hacia el *backend* a través del *proxy* no se dirigieron correctamente, lo que provocó errores de conexión y funcionalidad interrumpida en la interfaz de usuario.

- **Solución**: Actualización de la configuración de *proxy* en el *frontend* para alinear correctamente las solicitudes.

Problemas con *CORS* (*Cross-Origin Resource Sharing*)

CORS es un mecanismo de seguridad implementado por los navegadores web que restringe las solicitudes *HTTP* entre diferentes dominios. Cuando el *frontend* y el *backend* se ejecutan en diferentes dominios o puertos, *CORS* puede bloquear las solicitudes *HTTP* debido a restricciones de seguridad.

- **Solución**: Añadir configuración de encabezados CORS en el *backend* para permitir solicitudes desde cualquier origen.

8. Conclusiones

Durante este proyecto de gestión de inventario, he aprendido mucho al utilizar tecnologías nuevas como Vue.js para el *frontend*, *Rust* con *Rocket* para el *backend*, y *Docker* para la gestión de contenedores.

En el *frontend* adquirí habilidades importantes en la creación de interfaces de usuario interactivas y responsivas utilizando Vue.js, aprovechando sus componentes y la forma eficiente de gestionar el estado de la aplicación con *Vuex*.

Por otro lado, elegir *Rust* con *Rocket* para el desarrollo del *backend* fue una decisión acertada, ya que proporcionó una base sólida y segura para la lógica de negocio y la gestión de datos. Aprendí a usar el sistema de tipos seguro de *Rust* para evitar errores comunes y optimizar el rendimiento del código. Además, la configuración y optimización de *Docker* fueron fundamentales para asegurar un entorno de desarrollo y despliegue consistente, facilitando la *escalabilidad* y la gestión eficiente de dependencias en diferentes situaciones.

Incluso he tenido la oportunidad de descubrir herramientas útiles para crear documentación como *Typst* [14], una mezcla de *markdown* y *latex* añadiendo sus propias funcionalidades, completamente web.

En resumen, este proyecto no solo me permitió aplicar conocimientos teóricos previos como *HTML, CSS, JAVASCRIPT y SQL*, sino que también exploré y dominé nuevas tecnologías esenciales en el campo del desarrollo de software moderno. A través de los desafíos enfrentados y las soluciones implementadas, he mejorado significativamente mi capacidad para diseñar, desarrollar y desplegar aplicaciones complejas de manera efectiva.

Bibliografía

[1] «Vue Javascript Framework». [En línea]. Disponible en: https://vuejs.org/

[2] «Vue Development Tool». [En línea]. Disponible en: https://cli.vuejs.org/

[3] «Rust Programming Language». [En línea]. Disponible en: https://www.rust-lang.org/

[4] «Rocket». [En línea]. Disponible en: https://rocket.rs/

[5] «Docker Documentation and Guides». [En línea]. Disponible en: https://docs.docker.com/

[6] «Docker Container Registry». [En línea]. Disponible en: https://hub.docker.com/

[7] «JWT Rust Documentation». [En línea]. Disponible en: https://docs.rs/jwt/latest/jwt/

[8] «SQLX for Rust». [En línea]. Disponible en: https://github.com/launchbadge/sqlx

[9] «Serde Rust». [En línea]. Disponible en: https://serde.rs/

[10] «Tokio Rust». [En línea]. Disponible en: https://tokio.rs/

[11] «Vue Charts». [En línea]. Disponible en: https://vue-chartjs.org/

[12] «Argon2 Rust Documentation». [En línea]. Disponible en: https://docs.rs/argon2/latest/argon2/

[13] «Vue Bootstrap». [En línea]. Disponible en: https://bootstrap-vue.org/

[14] «Typst». [En línea]. Disponible en: https://typst.app/