Pontificia Universidad Católica del Ecuador

Proyecto Integrador ECOCANJE

Tecnología en Desarrollo de Software

Integrantes:

- Cristian Pareja
- Franklin Toro

Fecha: 07 de Enero de 2026

1. Introducción

- **Contexto del problema**

En la actualidad, la gestión inadecuada de residuos reciclables representa un problema ambiental significativo en la ciudad de Quito. Aunque muchas personas están dispuestas a reciclar, existen barreras como la falta de incentivos, el desconocimiento de procesos adecuados y la ausencia de plataformas digitales que faciliten el intercambio responsable de materiales reciclables. Esto provoca que grandes cantidades de residuos reutilizables terminen contaminando el entorno.

Adicionalmente, las iniciativas de reciclaje suelen carecer de sistemas tecnológicos que permitan registrar, controlar y dar seguimiento a los intercambios realizados, lo que limita su alcance y sostenibilidad a largo plazo.

- **Qué es ECOCANJE**

ECOCANJE v1.0 es una aplicación orientada a la gestión de intercambio de productos reciclables, que permite registrar, administrar y controlar productos dentro de un sistema digital. El proyecto busca apoyar iniciativas de economía circular mediante una plataforma tecnológica que facilite la gestión de productos reciclables.

ECOCANJE v1.0 se implementa como una aplicación basada en una arquitectura por capas, que expone una API REST para la administración de productos y operaciones relacionadas, garantizando orden y escalabilidad.

- **A quién va dirigido**

El sistema ECOCANJE en su versión inicial 1.0 está dirigido a estudiantes de la PUCE interesados en participar en procesos de reciclaje y reutilización de productos.

- **Alcance del proyecto**

- Gestión de productos mediante operaciones CRUD (crear, listar, actualizar y eliminar).
- Exposición de una API REST para el acceso a la información de productos.
- Validación básica de datos para garantizar la integridad de la información.
- Conexión a una base de datos PostgreSQL para el almacenamiento persistente.
- Uso de una arquitectura por capas que separa responsabilidades entre rutas, servicios y repositorios.
- Pruebas de funcionamiento de la API mediante herramientas como Postman.

En esta fase del proyecto, no se contempla:

- Desarrollo de una interfaz gráfica frontend completa.
- Implementación de doble autenticación de usuarios finales.
- Integración con sistemas externos de pago o logística.
- Despliegue en un entorno de producción real.

Estas funcionalidades quedan previstas como posibles mejoras o futuras versiones del sistema.

2. Tecnologías Utilizadas

- **Lenguaje de Programación**

  JavaScript (Node.js)

  Se utiliza JavaScript como lenguaje principal del backend mediante el entorno de ejecución Node.js, lo que permite construir aplicaciones del lado del servidor de forma eficiente, asincrónica y con alto rendimiento.

- **Backend**

  Express.js (Framework)

  Express es un framework minimalista para Node.js que facilita la creación de servidores web y APIs REST. En el proyecto ECOCANJE se utiliza para:

- Definir rutas HTTP (GET, POST, PUT, DELETE).
- Manejar solicitudes y respuestas del cliente.
- Organizar la aplicación siguiendo una arquitectura por capas.

- **Base de Datos**

  PostgreSQL

  PostgreSQL es el sistema gestor de base de datos relacional utilizado para almacenar la información del sistema, destacando por su confiabilidad, soporte para integridad de datos y escalabilidad.

- **Contenedores**

  Docker y Docker Compose

  Utilizamos Docker para contener la base de datos PostgreSQL, permitiendo un entorno de desarrollo consistente y facilitar el despliegue y configuración. Docker Compose utilizamos para administrar los contenedores necesarios del sistema.

- **Acceso a data**

  Sequelize (ORM)

  Sequelize se utiliza como herramienta ORM para mapear los modelos de la base de datos a objetos JavaScript, facilitando el acceso y la manipulación de datos desde el backend.

  pg (node-postgres)

  La librería pg permite ejecutar consultas SQL directas sobre PostgreSQL, ofreciendo flexibilidad y control en determinadas operaciones.

- **Frontend**

  React (Biblioteca)

  React se utiliza para el desarrollo del frontend de ECOCANJE, permitiendo la creación de interfaces de usuario dinámicas y reutilizables mediante componentes.

Tailwind CSS (Framework)

Tailwind CSS se emplea para el diseño visual de la aplicación, proporcionando clases utilitarias que permiten construir interfaces modernas, responsivas y consistentes sin necesidad de escribir grandes hojas de estilo personalizadas.

- **Arquitectura**

  Arquitectura por Capas

  El sistema se estructura en capas (rutas, servicios y repositorios), lo que mejora la mantenibilidad y escalabilidad del proyecto.

- **Seguridad de la Información**

  Bcrypt

  Se utiliza la librería bcrypt para el hash de contraseñas de los usuarios, garantizando que las credenciales no se almacenen en texto plano. Y protegiendo la información sensible.

- **Auditoría del Sistema**

  Se implementan mecanismos de auditoría a nivel de base de datos mediante funciones y triggers, permitiendo registrar acciones como creación, actualización y eliminación de registros, junto con información del usuario y la fecha de la operación.

- **Respaldo de Información**

  Cron (tareas programadas en Linux)

  Se emplea el programador de tareas cron para automatizar respaldos periódicos de la base de datos PostgreSQL. Esta automatización garantiza la disponibilidad de copias de seguridad sin intervención manual.

- **Herramientas de pruebas**

  Postman

  Postman se utiliza para probar los endpoints de la API REST, validando el funcionamiento de las operaciones y el manejo de errores.

- **Control de versiones**

  Git

  Git se utiliza para el control de versiones del código fuente, permitiendo el seguimiento de cambios durante el desarrollo del proyecto.

3. Manual de Instalación y Ejecución del Sistema

El presente manual describe los requisitos, pasos de instalación y ejecución necesarios para el funcionamiento del sistema **ECOCANJE**. Está dirigido a usuarios técnicos, docentes y estudiantes que requieran ejecutar el sistema en un entorno local.

3.1 Requisitos del Sistema

- Sistema operativo: Windows, Linux o macOS
- **Node.js** (versión 18 o superior)
- **npm** (incluido con Node.js)
- **Docker** y **Docker Compose**
- **Git**
- Navegador web (Google Chrome, Firefox u otro)
- Herramienta para pruebas de API: **Postman**

3.2 Clonar el Proyecto

- Abrir una terminal o consola.
- Clonar el repositorio del proyecto utilizando Git:

git clone https://github.com/CristianPareja/Proyecto_final_3_Pareja_Toro.git

- Acceder al directorio del proyecto:

cd Proyecto_final_3_Pareja_Toro

3.3 Configuración y Ejecución de la Base de Datos

- Verificar que Docker esté en ejecución.
- Desde la raíz del proyecto, ejecutar:

docker compose up -d

- Confirmar que los contenedores estén activos:

docker ps

```
PS C:\Users\crist\OneDrive\Escritorio\PUCE\DesarrolloWeb\Proyecto_final_ECOCANJE> docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED       STATUS          PORTS                                            NAMES
2da24d912d9f   proyecto_final_ecocanje-backup-cron  "docker-entrypoint.s…"   21 hours ago  Up About an hour  5432/tcp                                         ecocanje_backup_cron
17db753dd7e1   postgres:15                          "docker-entrypoint.s…"   21 hours ago  Up About an hour  0.0.0.0:5435->5432/tcp, [::]:5435->5432/tcp      express_003_postgres
```

Este proceso levantará PostgreSQL con su configuración personalizada, el servicio de cron para respaldos automáticos y los scripts de inicialización de la base de datos con sequelize.

3.4 Configuración y Ejecución del Backend

- Desde la raíz del proyecto, instalar las dependencias:

  npm install

- Configurar el archivo .env con los datos de conexión a la base de datos y variables necesarias.

```
JWT_SECRET=ecocanje_super_secreto_123
DB_HOST=localhost
DB_PORT=5435
DB_NAME=ecocanje_db
DB_USER=postgres
DB_PASSWORD=postgres
PORT=4000
```

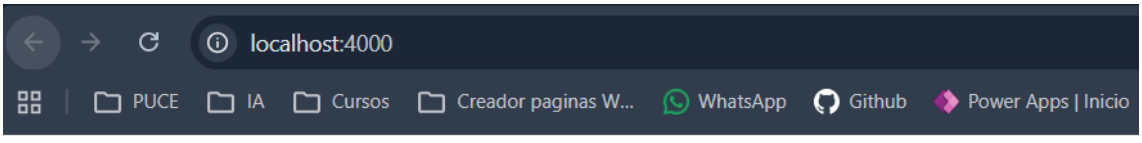
- Iniciar el servidor backend:

  node server.js

```
PS C:\Users\crist\OneDrive\Escritorio\PUCE\DesarrolloWeb\Proyecto_final_ECOCANJE> node server.js
>>
[dotenv@17.2.3] injecting env (7) from .env -- tip: override existing env vars with { override: true }
[dotenv@17.2.3] injecting env (0) from .env -- tip: encrypt with Dotenvx: https://dotenvx.com
Conexión a PostgreSQL OK
Tablas creadas / sincronizadas
Servidor corriendo en http://localhost:4000
```

- El backend quedará disponible en:

  http://localhost:4000



EcoCanje API funcionando

3.5 Configuración y Ejecución del Frontend

- Acceder al directorio del frontend:

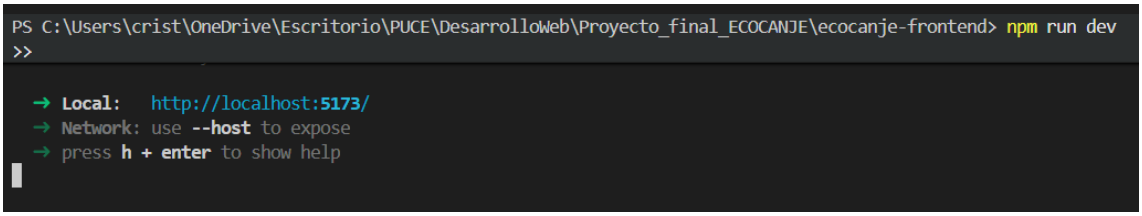
  cd ecocanje-frontend

- Instalar las dependencias:

  npm install

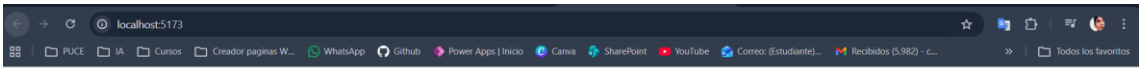
- Iniciar la aplicación frontend:

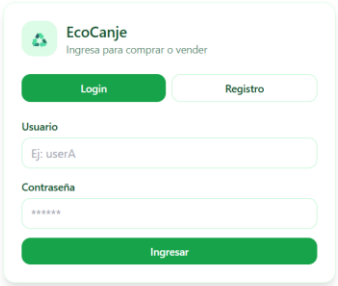
  npm run dev



- Acceder al sistema desde el navegador en una URL similar a:

  http://localhost:5173





3.6 Uso del Sistema

- El **frontend** permite al usuario interactuar con el sistema mediante una interfaz desarrollada en React y Tailwind CSS.
- El **backend** expone una API REST que gestiona la lógica del sistema.
- Las operaciones pueden ser probadas mediante herramientas como Postman o directo desde la interfaz de la app.

3.7 Backups Automatizados

El sistema incluye respaldos automáticos de la base de datos mediante:

- Script backup.sh
- Programación con cron
- Uso de la herramienta pg_dump

Las copias de seguridad se almacenan en el directorio backups/

3.8 Detención del Sistema

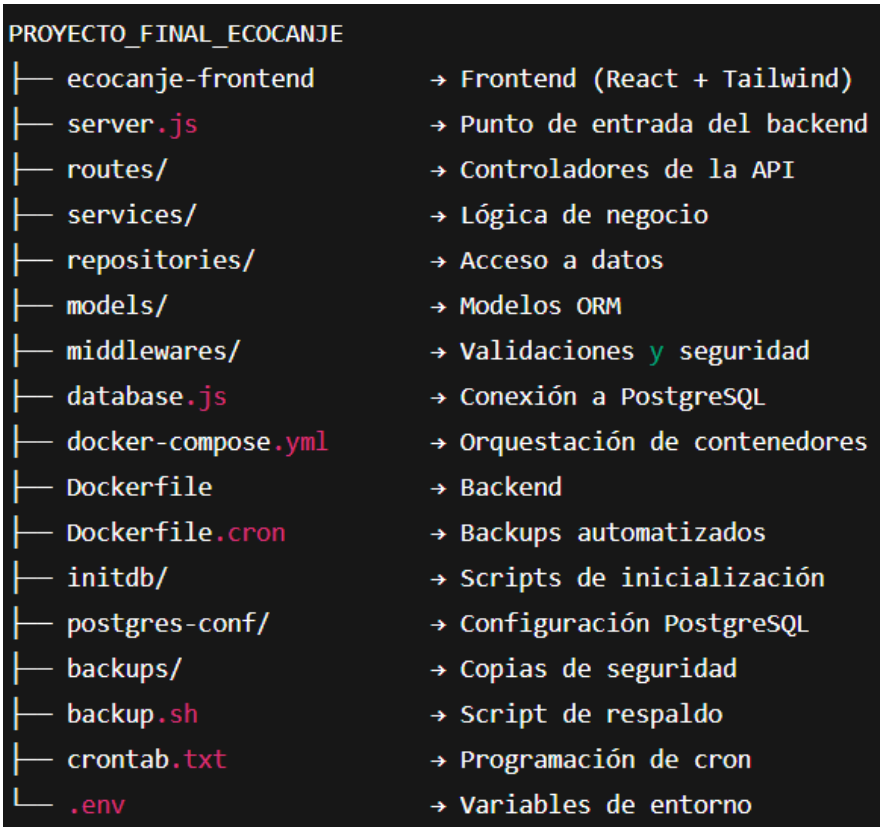
Para detener el sistema:

- Detener frontend y backend con Ctrl + C.
- Apagar los contenedores Docker:

docker compose down

4. Arquitectura del Sistema

El proyecto ECOCANJE implementa una arquitectura web de tres capas, complementada con una arquitectura por capas en el backend y soportada por una infraestructura basada en contenedores Docker.



El sistema se organiza en tres grandes capas:

1. Capa de Frontend
2. Capa de Lógica de Negocio (Backend)
3. Capa de Datos (Base de Datos)

4.1 Arquitectura del Frontend

El frontend del sistema se encuentra en el directorio ecocanje-frontend y está desarrollado utilizando React y Tailwind CSS.

- Arquitectura basada en componentes reutilizables.
- Comunicación con el backend mediante solicitudes HTTP a la API REST.
- Interfaz responsiva y moderna gracias al uso de Tailwind CSS.

4.2 Arquitectura del Backend

El backend de ECOCANJE está desarrollado en Node.js con Express y sigue una arquitectura por capas, organizada de la siguiente manera:

4.2.1 Capa de Rutas (routes)

La carpeta routes actúa como la capa de controladores del sistema.

- Recibir las solicitudes HTTP del frontend.
- Definir los endpoints de la API REST.
- Delegar la lógica de negocio a la capa de servicios.

4.2.2 Capa de Servicios (services)

La carpeta services contiene la lógica de negocio del sistema.

En esta capa se realizan:

- Validaciones de datos.
- Aplicación de reglas del sistema.

- Control de operaciones entre distintas capas.
- Lanzamiento de errores controlados.

Esta capa actúa como intermediaria entre las rutas y los repositorios.

4.2.3 Capa de Repositorios (repositories)

La carpeta repositories se encarga del acceso a los datos.

Sus funciones principales son:

- Ejecutar operaciones CRUD.
- Interactuar con la base de datos PostgreSQL.
- Abstraer el origen de los datos del resto del sistema

4.2.4 Capa de Modelos (models)

La carpeta models contiene los modelos definidos mediante Sequelize, los cuales representan las entidades de la base de datos.

- Definen la estructura de las tablas.
- Incluyen restricciones y validaciones básicas.
- Facilitan el mapeo objeto–relacional.

4.2.5 Middlewares

La carpeta middlewares agrupa funciones que se ejecutan antes o después de procesar una solicitud, tales como:

- Validaciones.
- Seguridad.

- Manejo de errores.
- Protección de rutas.

Esta capa contribuye a la robustez y seguridad del backend.

4.3 Arquitectura de Datos

La capa de datos está basada en PostgreSQL, desplegada mediante contenedores Docker.

- Base de datos relacional robusta.
- Configuración personalizada ubicada en postgres-conf.
- Persistencia de datos independiente del ciclo de vida de los contenedores.

4.4 Seguridad, Auditoría y Respaldo

- **Hash de contraseñas** para proteger credenciales.
- **Auditorías** a nivel de base de datos para registrar operaciones críticas.
- **Backups automatizados** mediante scripts y tareas programadas con cron.

  Con almacenamiento de respaldos en el directorio backups.

4.5 Flujo General del Sistema Ecocanje

1. El usuario interactúa con el frontend.
2. El frontend envía una solicitud HTTP al backend.
3. La ruta recibe la solicitud y la envía al servicio correspondiente.
4. El servicio valida y procesa la lógica de negocio.
5. El repositorio accede a la base de datos.
6. La respuesta retorna al frontend.

5. Diseño de la Base de Datos

5.1 Descripción General

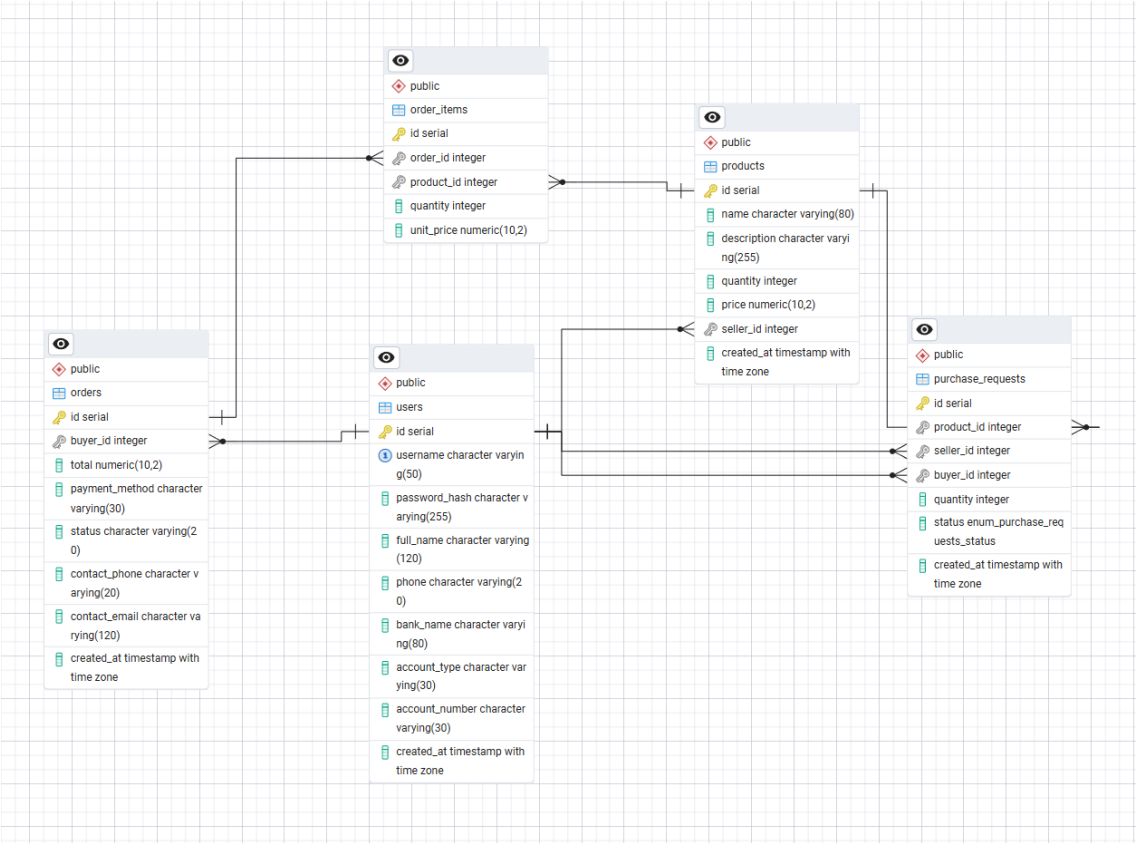
La base de datos del proyecto ECOCANJE está diseñada bajo un modelo relacional, utilizando PostgreSQL como sistema gestor. El diseño permite gestionar usuarios, productos, órdenes de compra y los detalles asociados a cada orden.

5.2 Entidades Principales del Sistema

El sistema está compuesto por las siguientes tablas principales:

- users
- products
- orders
- order_items
- purchase_requests

5.3 Modelo Entidad–Relación (ERD)



5.4 Diccionario de Datos

- **users**

Almacena la información de los usuarios del sistema (compradores y vendedores).

- **PK:** id

Campo	Tipo de dato	Descripción
id	INTEGER (PK)	Identificador único del usuario
username	VARCHAR(50)	Nombre de usuario único en el sistema
password_hash	VARCHAR(255)	Contraseña cifrada mediante bcrypt
full_name	VARCHAR(120)	Nombre completo del usuario
phone	VARCHAR(20)	Número telefónico de contacto
bank_name	VARCHAR(80)	Nombre del banco del usuario
account_type	VARCHAR(30)	Tipo de cuenta bancaria
account_number	VARCHAR(30)	Número de cuenta bancaria
created_at	TIMESTAMP	Fecha y hora de creación del usuario

- **Relaciones:**
  - users.id ← FK en products.seller_id
  - users.id ← FK en orders.buyer_id
  - users.id ← FK en purchase_requests.buyer_id
  - users.id ← FK en purchase_requests.seller_id

- **products**

Almacenar los productos ofrecidos por los usuarios vendedores.

- **PK:** id
- **FK:** seller_id → users.id

Campo	Tipo de dato	Descripción
id	INTEGER (PK)	Identificador único del producto
name	VARCHAR(80)	Nombre del producto
description	VARCHAR(255)	Descripción del producto
quantity	INTEGER	Cantidad disponible en stock
price	NUMERIC(10,2)	Precio unitario del producto
seller_id	INTEGER (FK)	Usuario que publica el producto
created_at	TIMESTAMP	Fecha y hora de creación del producto

Relaciones

- products.seller_id → users.id
- products.id → order_items.product_id
- products.id → purchase_requests.product_id

- **orders**

  Registra las órdenes de compra realizadas por los usuarios.

- **PK:** id
- **FK:** buyer_id → users.id

Campo	Tipo de dato	Descripción
id	INTEGER (PK)	Identificador único de la orden
buyer_id	INTEGER (FK)	Usuario comprador
total	NUMERIC(10,2)	Total de la compra
payment_method	VARCHAR(30)	Método de pago (didáctico)
status	VARCHAR(20)	Estado de la orden
contact_phone	VARCHAR(20)	Teléfono de contacto del comprador
contact_email	VARCHAR(120)	Correo electrónico del comprador
created_at	TIMESTAMP	Fecha y hora de creación de la orden

Relaciones

- orders.buyer_id → users.id
- orders.id → order_items.order_id

- **order_items**

  Tabla intermedia que relaciona órdenes con productos (relación muchos a muchos).

- **PK:** id
- **FK:**
  - order_id → orders.id
  - product_id → products.id

Campo	Tipo de dato	Descripción
id	INTEGER (PK)	Identificador único del detalle
order_id	INTEGER (FK)	Orden asociada
product_id	INTEGER (FK)	Producto comprado
quantity	INTEGER	Cantidad comprada
unit_price	NUMERIC(10,2)	Precio unitario al momento de la compra

Relaciones

- order_items.order_id → orders.id
- order_items.product_id → products.id

- **purchase_requests**

  Almacena las solicitudes de compra que un comprador envía a un vendedor. Funciona como un estado intermedio entre “quiero comprar” y “compra confirmada.

PK: id

- **FK:**
  - seller_id → users.id
  - buyer_id→ users.id
  - product_id → products.id

Campo	Tipo	Descripción
id	serial (PK)	Identificador único de la solicitud
product_id	integer (FK → products.id)	Producto que se desea comprar
seller_id	integer (FK → users.id)	Usuario vendedor (dueño del producto)
buyer_id	integer (FK → users.id)	Usuario comprador
quantity	integer	Cantidad solicitada
status	enum_purchase_requests_status	Estado de la solicitud
created_at	timestamp with time zone	Fecha y hora de creación

Relaciones

- order_items.order_id → orders.id
- order_items.product_id → products.id

- Todas las tablas poseen **clave primaria**.
- Las relaciones se controlan mediante **claves foráneas**.
- Se evita duplicidad de datos mediante normalización.
- Las contraseñas se almacenan únicamente como **hash**.
- Se utilizan estados (status) para control del ciclo de vida de órdenes y solicitudes.

6. API REST – Endpoints del Sistema

6.1 Módulo de Autenticación (Auth)

Ruta base: /api/auth

Este módulo permite el registro y autenticación de usuarios, asegurando que las operaciones protegidas solo puedan ser ejecutadas por usuarios autenticados.

Método	Endpoint	Descripción	Doble autenticación
POST	/api/auth/register	Registra un nuevo usuario	No
POST	/api/auth/login	Autentica un usuario y retorna sesión/token JWT	No

6.2 Módulo de Productos

Ruta base: /api/products

Este módulo gestiona los productos ofrecidos por los usuarios vendedores.

Método	Endpoint	Descripción	Autenticación
GET	/api/products	Lista todos los productos disponibles	No
GET	/api/products/:id	Obtiene un producto por su ID	No
POST	/api/products	Crea un nuevo producto	Sí
PUT	/api/products/:id	Actualiza un producto existente	Sí
DELETE	/api/products/:id	Elimina un producto	Sí

6.3 Módulo de Solicitudes de Compra (Purchase Requests)

Ruta base: /api/Purchase Requests

Este módulo representa el núcleo del sistema de notificaciones.

Método	Endpoint	Descripción	Autenticación
POST	/api/purchase-requests	Crea una solicitud de compra (estado PENDING)	Sí
GET	/api/purchase-requests/seller/pending	Lista solicitudes pendientes del vendedor (notificaciones)	Sí
POST	/api/purchase-requests/:id/accept	Vendedor acepta solicitud (baja stock y crea orden)	Sí
POST	/api/purchase-requests/:id/reject	Vendedor rechaza solicitud	Sí
GET	/api/purchase-requests/:id/can-see-bank	Comprador consulta estado y datos del vendedor	Sí
Método	Endpoint	Descripción	Autenticación
POST	/api/orders/checkout	Crea una orden de compra a partir del carrito	Sí

GET	/api/orders/my	Lista las compras del usuario autenticado	Sí

6.3 Módulo de Órdenes

Ruta base: /api/orders

Las órdenes ya no se crean directamente por el comprador, sino solo cuando el vendedor acepta una solicitud.

Método	Endpoint	Descripción	Autenticación
GET	/api/orders/my	Lista compras del usuario autenticado	Sí

6.4 Flujo de compra implementado:

Comprador (Usuario B) selecciona un producto.

Se envía solicitud a:

POST /api/purchase-requests

El backend crea un registro en:

- purchase_requests con estado **PENDING**

Vendedor (Usuario A):

- Ve la solicitud mediante polling:
- GET /api/purchase-requests/seller/pending

El vendedor decide:

- **Aceptar**
    - Se reduce el stock del producto
    - Se crea un registro en orders
    - Se crean registros en order_items
    - La solicitud pasa a **ACCEPTED**
- **Rechazar**
    - No se modifica stock
    - La solicitud pasa a **REJECTED**

Estados de una solicitud (purchase_requests.status)

Estado	Significado
PENDING	Esperando decisión del vendedor
ACCEPTED	Aceptada, stock reducido y orden creada
REJECTED	Rechazada, no hay cambios de stock

6.5 Seguridad de los Endpoints

- Los endpoints sensibles utilizan un **middleware de autenticación**.
- El usuario autenticado se obtiene desde el token/sesión.
- Se validan parámetros y datos de entrada antes de procesar las solicitudes.

6.6 Códigos de Respuesta HTTP implementados

Código	Significado
200	Operación exitosa
201	Recurso creado correctamente
400	Error de validación
401	No autorizado
404	Recurso no encontrado
500	Error interno del servidor

7. Seguridad, Auditoría y Backups

7.1 Análisis de riesgos

Se identifican las siguientes vulnerabilidades comunes:

1. **Accesos no autorizados** debido a una mala gestión de usuarios y contraseñas.
2. **Falta de auditoría**, lo que dificulta detectar accesos indebidos o modificaciones no autorizadas.
3. **Pérdida de información** por ausencia de políticas de respaldo.
4. **Ataques de inyección SQL**, si no se controla el acceso desde la aplicación.

La superficie de ataque principal se encuentra en la conexión entre la aplicación web y la base de datos, por lo que se aplican controles de acceso estrictos y auditoría a nivel del DBMS.

7.2 Implementación de RBAC (Control de Acceso Basado en Roles)

Se implementó un modelo de **Control de Acceso Basado en Roles (RBAC)** siguiendo el principio de menor privilegio.

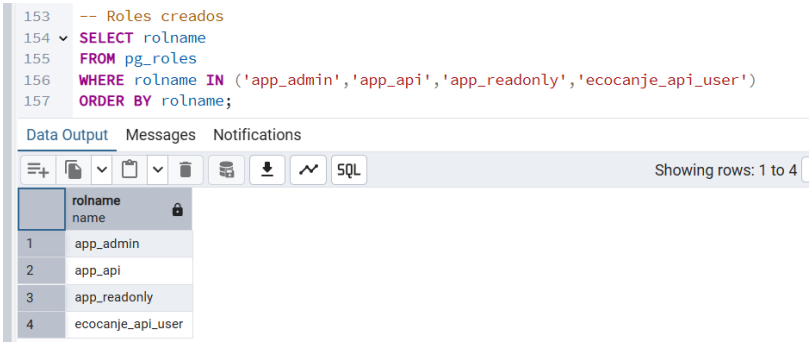
Roles definidos:

- **app_admin**: rol administrativo para mantenimiento.
- **app_api**: rol técnico utilizado por el backend de la aplicación.
- **app_readonly**: rol de solo lectura para consultas o reportes.

Usuario técnico:

- **ecocanje_api_user**, utilizado exclusivamente por el backend para acceder a la base de datos.

Roles creados en pgadmin



Cada rol cuenta únicamente con los permisos necesarios para cumplir su función, reduciendo el impacto ante un posible compromiso de credenciales.

Rol Aplicación	**Rol BD**	**Permisos**
Backend API	app_api	SELECT, INSERT, UPDATE, DELETE
Reportes	app_readonly	SELECT
Administración	app_admin	Gestión controlada

Permisos asignados a los diferentes roles

```
164 SELECT grantee, privilege_type
165 FROM information_schema.role_table_grants
166 WHERE grantee = 'app_api';
167
168
169
```

Data Output | Messages | Notifications

	grantee name	privilege_type character varying
1	app_api	INSERT
2	app_api	SELECT
3	app_api	UPDATE
4	app_api	DELETE

7.3 Configuración de cifrado

En PostgreSQL, el cifrado se maneja en dos niveles:

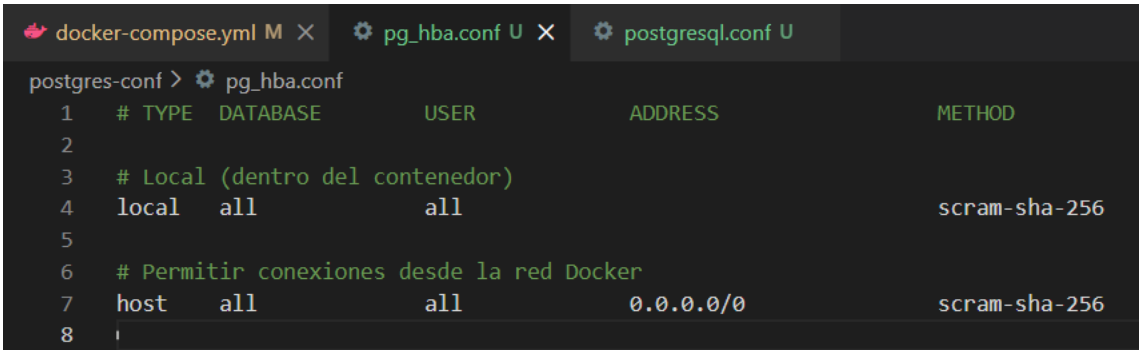
- **Cifrado en tránsito**: mediante SSL/TLS, garantizando que la comunicación entre la aplicación y la base de datos no pueda ser interceptada.
- **Cifrado en reposo**: mediante cifrado a nivel de disco y el uso de la extensión **pgcrypto** para operaciones criptográficas.

```
168  SELECT extname FROM pg_extension;
169
170
```

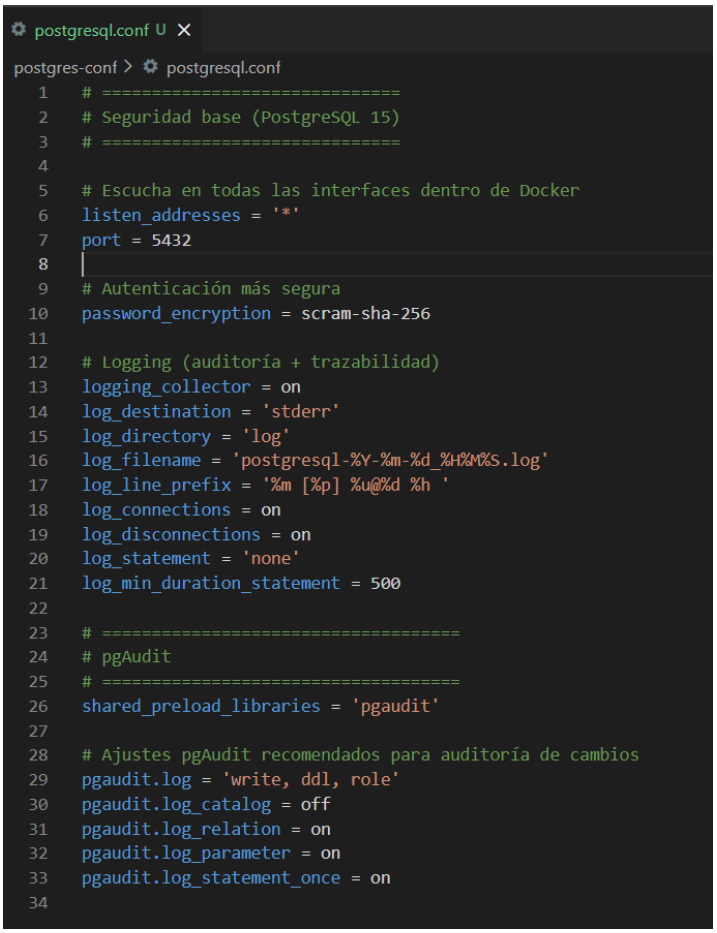
Data Output | Messages | Notifications

	extname name
1	plpgsql
2	pgcrypto

Para el proyecto Ecocanje, se habilitó la extensión pgcrypto, permitiendo el uso de funciones de hashing y cifrado cuando sea necesario para proteger información sensible.



Se configuró pg_hba.conf para aplicar autenticación SCRAM-SHA-256, evitando métodos inseguros como trust o md5, reduciendo el riesgo de accesos no autorizados.



- Se registran conexiones
- Se registran desconexiones
- Se auditan queries lentas (>500 ms)

pgAudit (Auditoría avanzada)

Esto habilita auditoría de:

- INSERT / UPDATE / DELETE
- CREATE / ALTER / DROP
- Cambios de roles
- Acceso a tablas

La gestión de claves se propone mediante variables de entorno y buenas prácticas de seguridad, evitando su exposición en el código fuente.

CONFIGURACIÓN DE AUDITORÍA Y MONITOREO

Se implementó auditoría mediante:

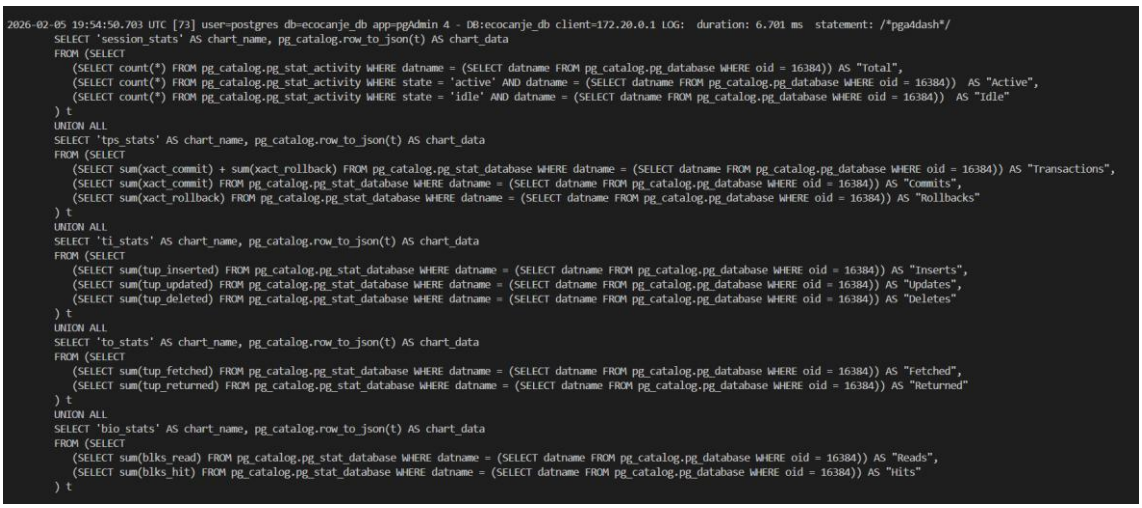
- **Logs nativos de PostgreSQL**, que registran conexiones, desconexiones y consultas.
- **Extensión pgAudit**, configurada para auditar operaciones de escritura y cambios estructurales.

Estos mecanismos permiten detectar accesos indebidos, modificaciones no autorizadas y analizar el comportamiento de los usuarios del sistema.

El monitoreo se realiza a través de logs del contenedor Docker y herramientas administrativas como pgAdmin.

Podemos verificar las auditorias con el comando:

docker logs express_003_postgres



Estrategia de backups y recuperación

Para el proyecto ECOCANJE se implementó una política de respaldos automáticos utilizando la herramienta **pg_dump**, ejecutada mediante un servicio **cron** en un contenedor Docker independiente.

Frecuencia: cada 1 hora

Retención: 7 días

Para el proyecto ECOCANJE se definieron los siguientes objetivos de recuperación:

RPO: 1 hora

RTO : 30 minutos

RPO (Recovery Point Objective):

El valor de RPO se estableció en una hora debido a que la aplicación ECOCANJE es una plataforma web académica que no maneja transacciones financieras en tiempo real.

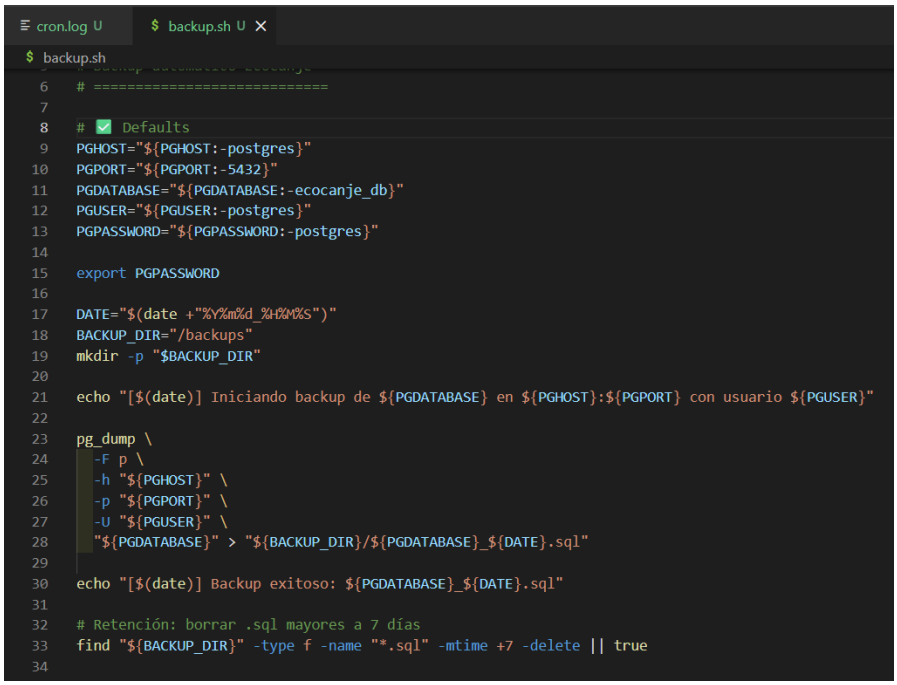
Las operaciones principales del sistema (registro de usuarios, publicación de productos y solicitudes de compra) toleran una pérdida limitada de información reciente sin comprometer la integridad general del sistema.

RTO (Recovery Time Objective):

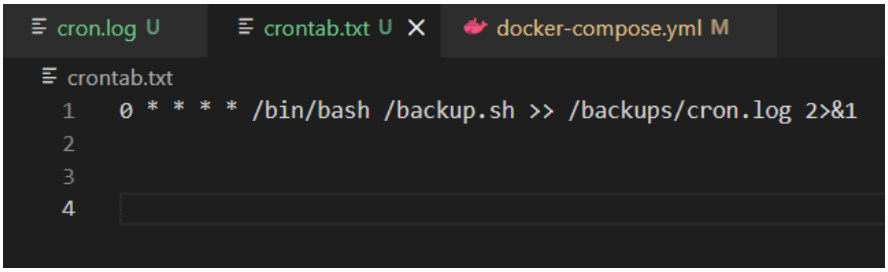
El RTO se fijó en 30 minutos considerando que los respaldos se realizan mediante archivos SQL generados con pg_dump, los cuales permiten una restauración rápida de la base de datos utilizando el comando psql

Esta combinación de RPO y RTO equilibra adecuadamente la disponibilidad del sistema, el costo operativo y el nivel de riesgo aceptable para el contexto académico del proyecto ECOCANJE.

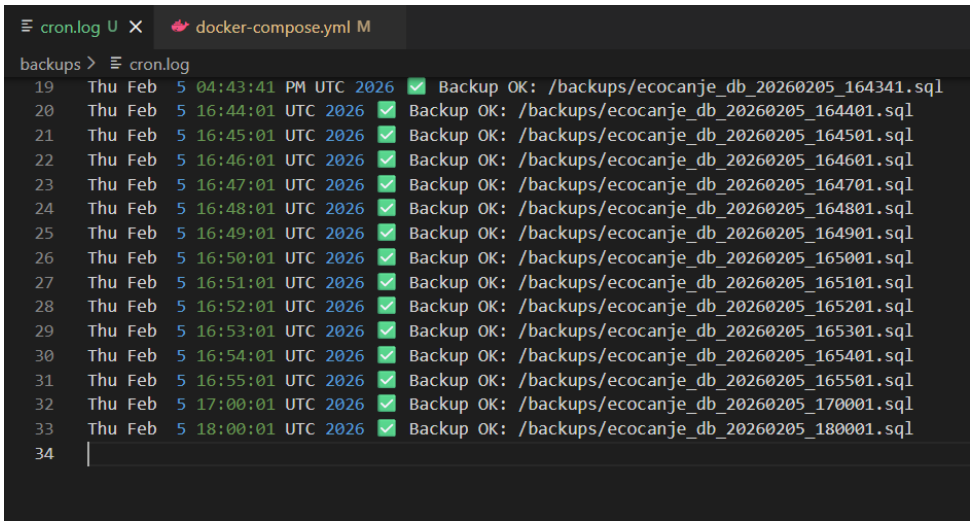
Backup.sh file: aqué vamos a realizar el backup automático



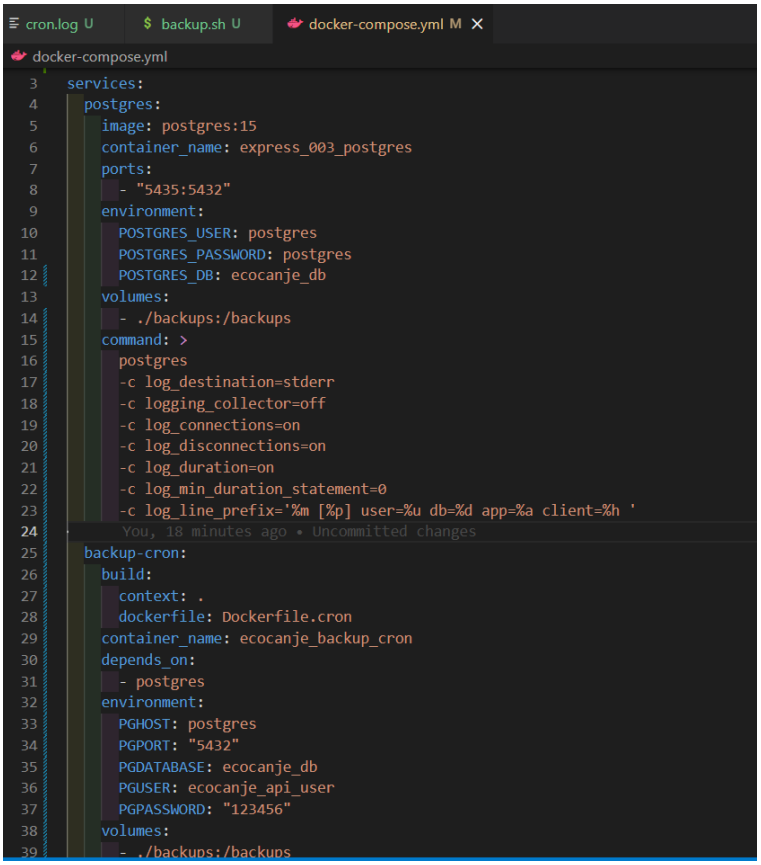
crontab: seteado para que se realice el backup cada hora



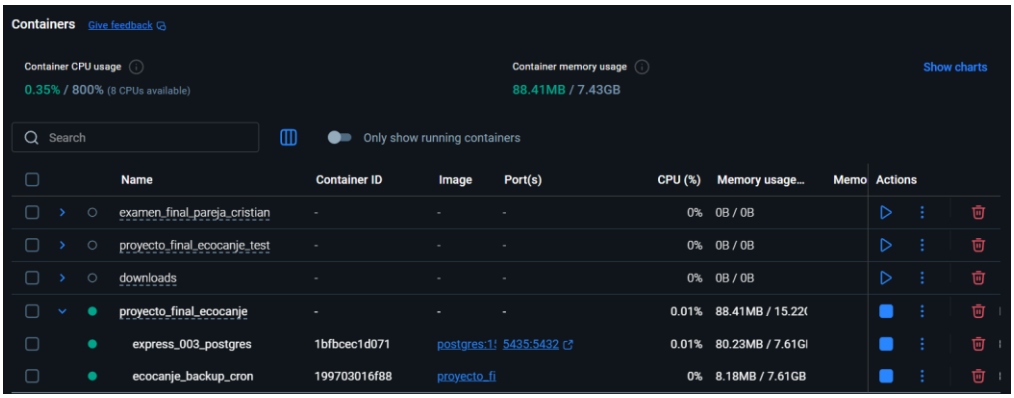
Cron.log: aquí vamos a ver todos los Backups realizados por nuestra base de datos. (al comienzo lo hicimos cada minuto para pruebas)



Docker -compose.yml: importante agregar un nuevo volumen para realizar los Backups en ese servidor de pruebas de respaldos.



ecocanje_backup_cron: container donde vamos a realizar los Backups automáticos.



8. Pruebas

8.1 POSTMAN

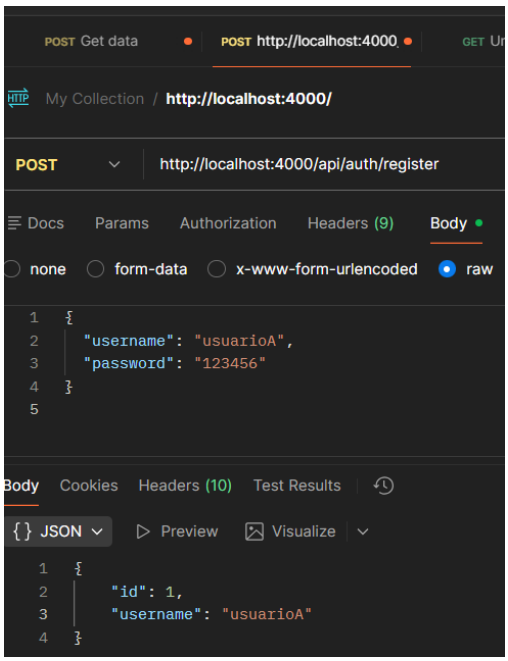
Levantamos Docker

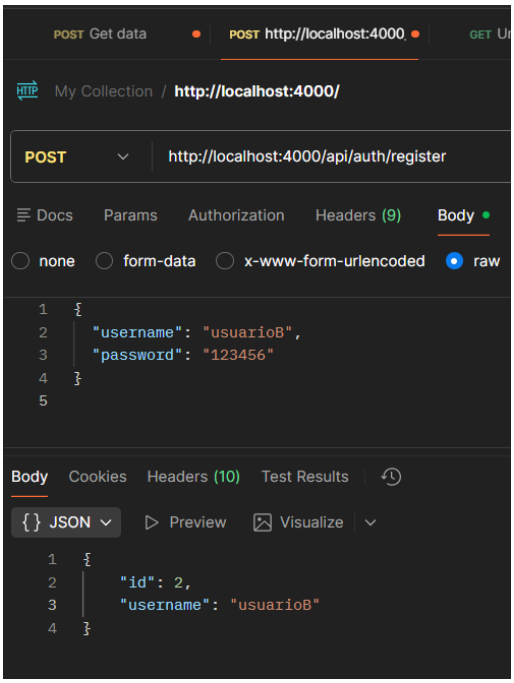
```
>> C:\Users\crist\OneDrive\Escritorio\PUCE\DesarrolloWeb\Proyecto_final_ECOCANJE>
PS C:\Users\crist\OneDrive\Escritorio\PUCE\DesarrolloWeb\Proyecto_final_ECOCANJE> docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED          STATUS          PORTS                                       NAMES
2da24d912d9f   proyecto_final_ecocanje-backup-cron  "docker-entrypoint.s…"   11 minutes ago   Up 11 minutes   5432/tcp                                    ecocanje_backup_cron
17db753dd7e1   postgres:15                          "docker-entrypoint.s…"   11 minutes ago   Up 11 minutes   0.0.0.0:5435->5432/tcp, [::]:5435->5432/tcp   express_003_postgres
PS C:\Users\crist\OneDrive\Escritorio\PUCE\DesarrolloWeb\Proyecto_final_ECOCANJE>
```

CRUD (Postman)

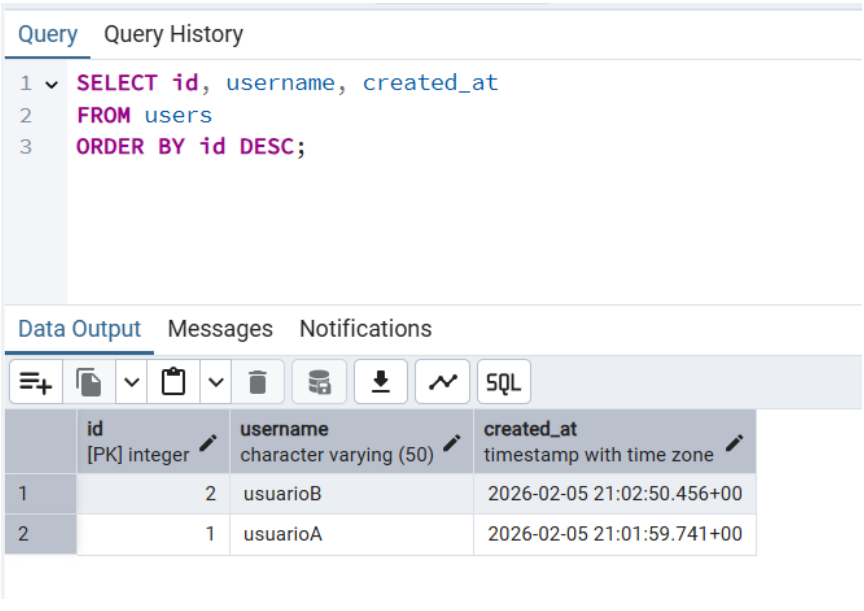
Registrar usuarioA y usuarioB

POST http://localhost:4000/api/auth/register




POST Get data
POST http://localhost:4000
My Collection / http://localhost:4000/
POST
http://localhost:4000/api/auth/register
Docs
Params
Authorization
Headers (9)
Body
none
form-data
x-www-form-urlencoded
raw
{
"username": "usuarioB",
"password": "123456"
}
Body
Cookies
Headers (10)
Test Results
{ } JSON
Preview
Visualize
{
"id": 2,
"username": "usuarioB"
}

pgadmin

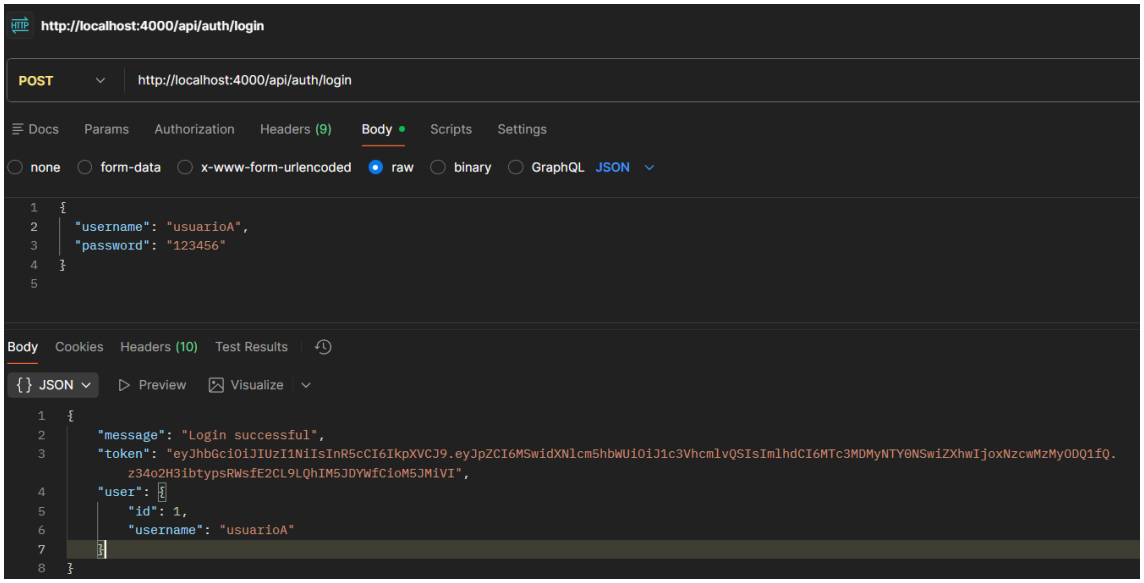

Query
Query History
SELECT id, username, created_at
FROM users
ORDER BY id DESC;
Data Output
Messages
Notifications
SQL
id
[PK] integer
username
character varying (50)
created_at
timestamp with time zone
1
2
usuarioB
2026-02-05 21:02:50.456+00
2
1
usuarioA
2026-02-05 21:01:59.741+00

Login usuarioA y usuarioB (guardar los tokens)

POST http://localhost:4000/api/auth/login

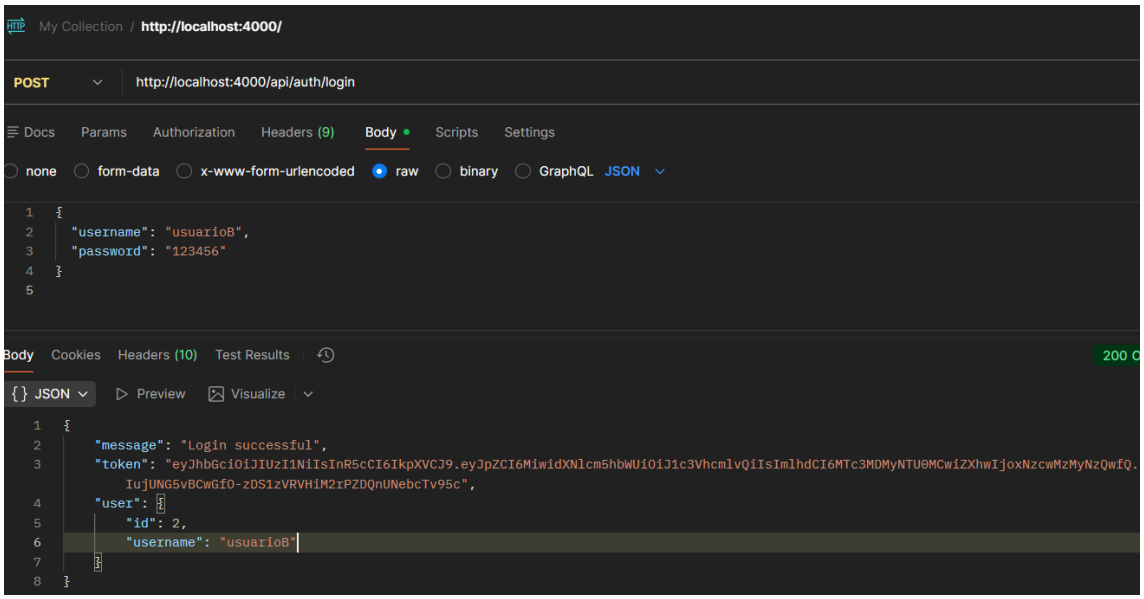
tokenA:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwidXNlcm5hbWUiOiJ1c3Vh
cmlvQSIsImlhdCI6MTc3MDMyNTY0NSwiZXhwIjoxNzcwMzMyODQ1fQ.z34o2H3i
btypsRWsfE2CL9LQhIM5JDYWfCioM5JMiVI



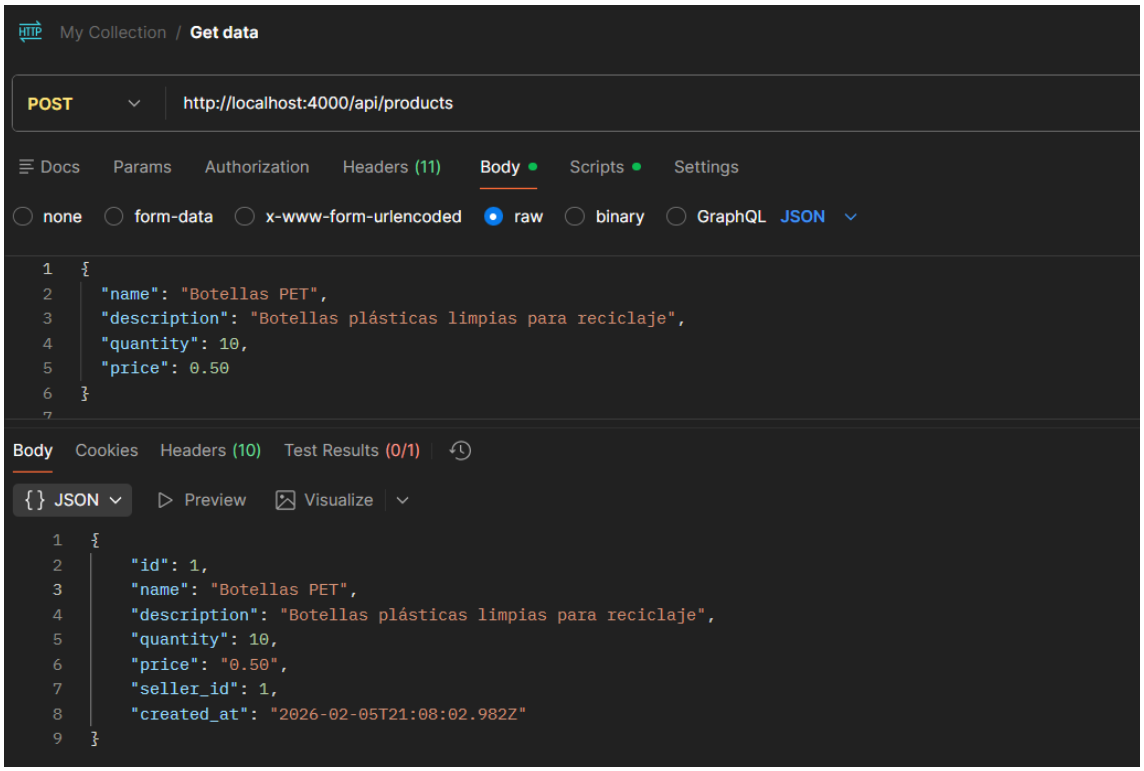
tokenB:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MiwidXNlcm5hbWUiOiJ1c3Vhc
mlvQiIsImlhdCI6MTc3MDMyNTU0MCwiZXhwIjoxNzcwMzMyNzQwfQ.IujUNG5v
BCwGfO-zDS1zVRVHiM2rPZDQnUNebcTv95c



Crear producto (A)

POST http://localhost:4000/api/products



pgadmin

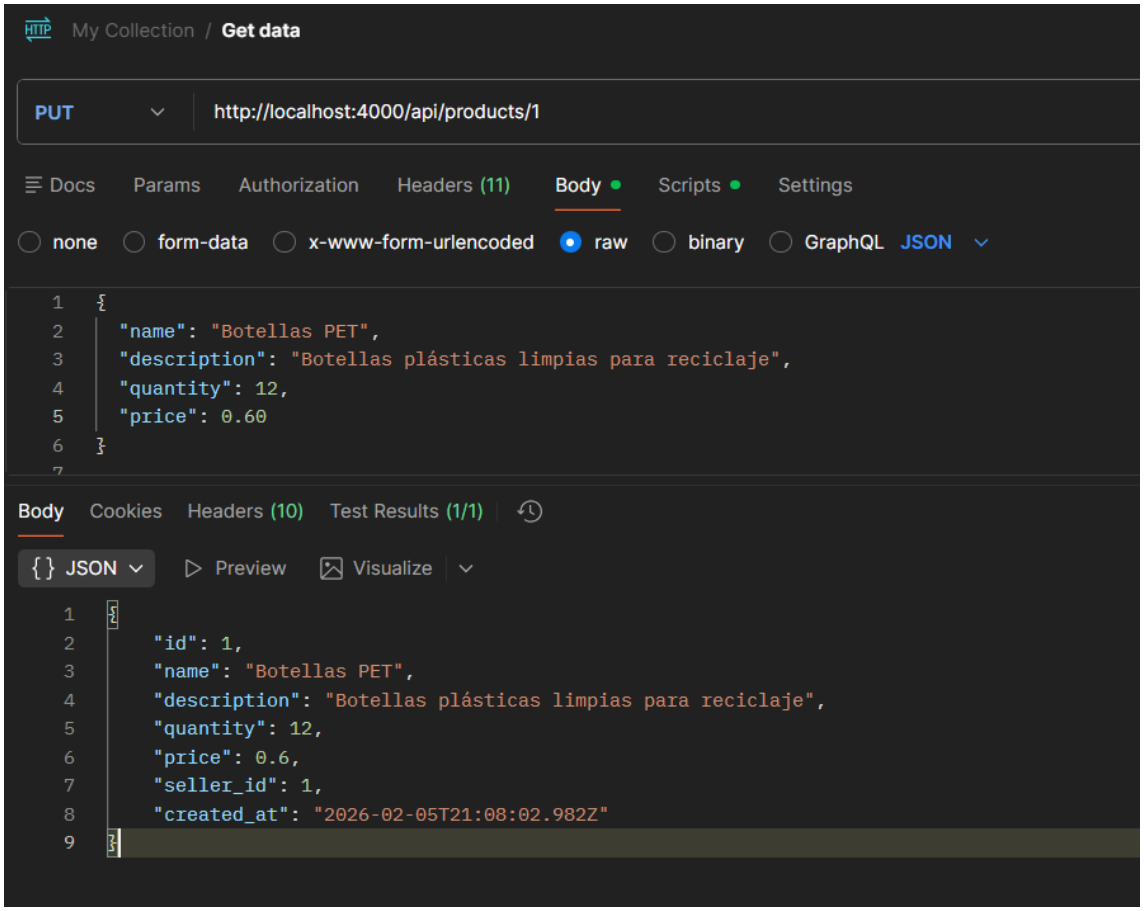
```sql
SELECT id, name, quantity, price, seller_id, created_at
FROM products
ORDER BY id DESC;
```

Data Output | Messages | Notifications

	id [PK] integer	name character varying (80)	quantity integer	price numeric (10,2)	seller_id integer	created_at timestamp with time zone
1	1	Botellas PET	10	0.50	1	2026-02-05 21:08:02.982+00

Editar producto (A)

PUT http://localhost:4000/api/products/1



pgadmin

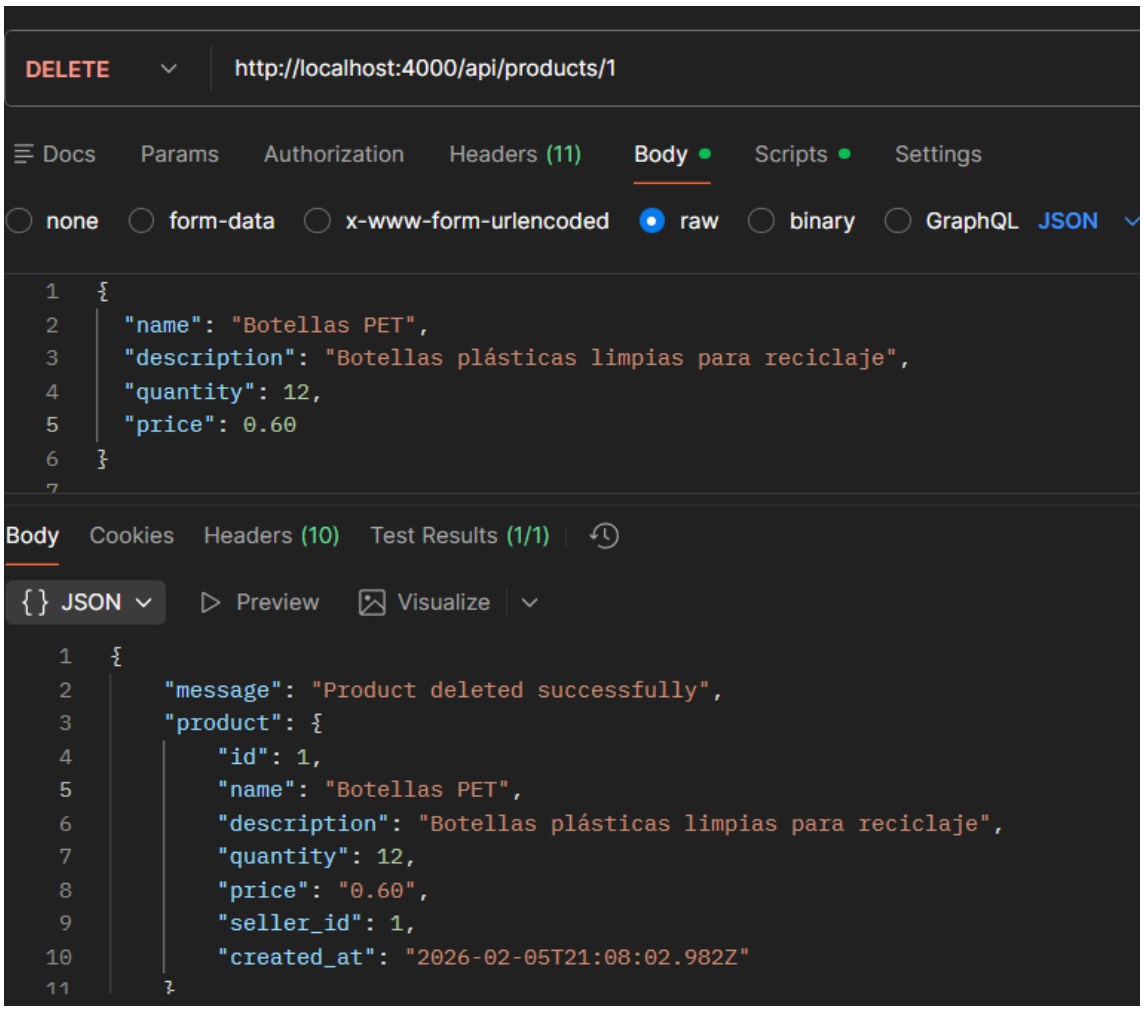
```
9  SELECT id, name, quantity, price, seller_id
10 FROM products
11 WHERE id = 1;
12
```

Data Output | Messages | Notifications

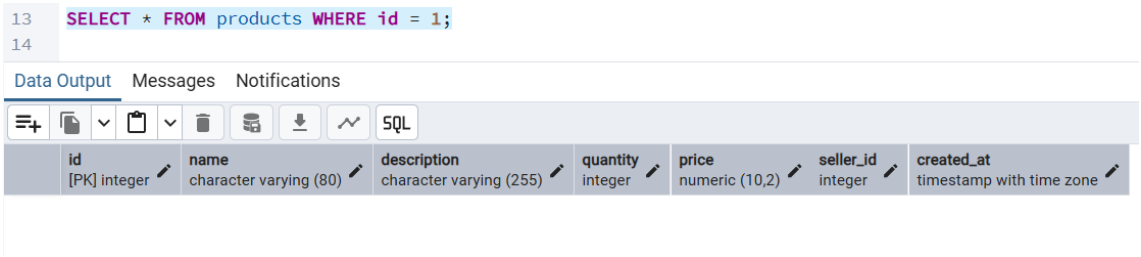
	id [PK] integer	name character varying (80)	quantity integer	price numeric (10,2)	seller_id integer
1	1	Botellas PET	12	0.60	1

Borrar producto (A)

DELETE http://localhost:4000/api/products/1



pgadmin



Pruebas de Auditoria

leyendo tablas del sistema y haciendo COPY public.users/products/... para generar el archivo .sql

Ejecutamos

docker stop ecocanje_backup_cron

docker logs -f express_003_postgres

realizamos en postman

register/login usuarioA

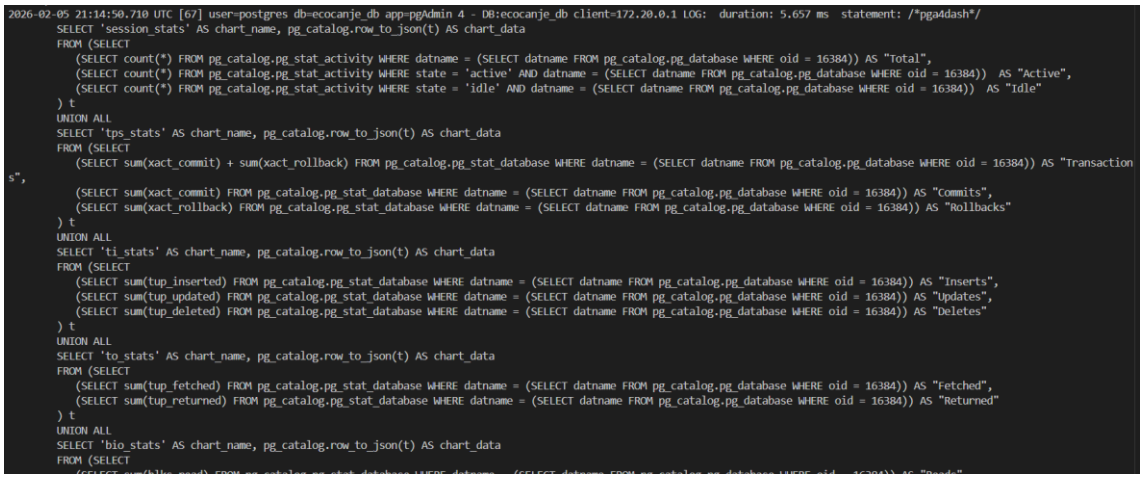
create product

update product

delete product

login usuarioB

docker start ecocanje_backup_cron



Creación de B

```
22  SELECT id, username, created_at
23  FROM users
24  WHERE username='usuarioB';
25
```

Data Output | Messages | Notifications

SQL

	id [PK] integer	username character varying (50)	created_at timestamp with time zone
1	2	usuarioB	2026-02-05 21:02:50.456+00

Creación de producto por parte de A

```
16  --Auditoria
17  SELECT p.id, p.name, u.username AS seller, p.created_at
18  FROM products p
19  JOIN users u ON u.id = p.seller_id
20  ORDER BY p.id DESC;
21
```

Data Output | Messages | Notifications

SQL

	id integer	name character varying (80)	seller character varying (50)	created_at timestamp with time zone
1	2	Botellas PET	usuarioA	2026-02-05 21:19:38.794+00

Actualización del producto por parte de A

```
26  SELECT p.id, p.name, p.quantity, p.price, u.username AS seller, p.created_at
27  FROM products p
28  JOIN users u ON u.id = p.seller_id
29  WHERE p.id = 2;
30
```

Data Output | Messages | Notifications

Showin

	id integer	name character varying (80)	quantity integer	price numeric (10,2)	seller character varying (50)	created_at timestamp with time zone
1	2	Botellas PET	120	0.80	usuarioA	2026-02-05 21:19:38.794+00

Eliminación de producto por parte de A

```
31  SELECT * FROM products WHERE id = 2;
32
```

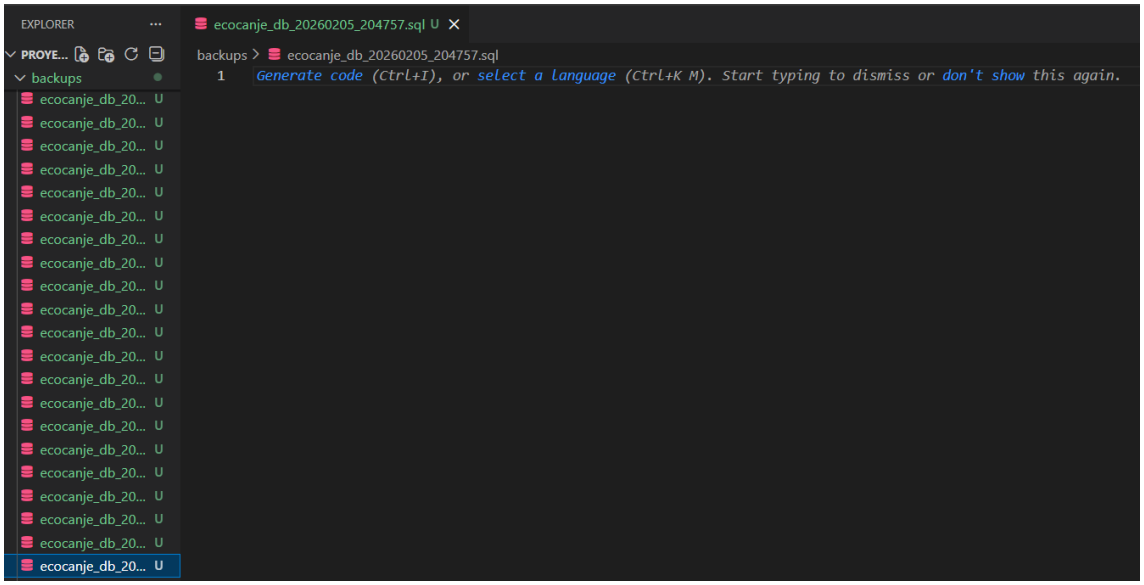
Data Output | Messages | Notifications

	id [PK] integer	name character varying (80)	description character varying (255)	quantity integer	price numeric (10,2)	seller_id integer	created_at timestamp with time zone

Pruebas de Backup

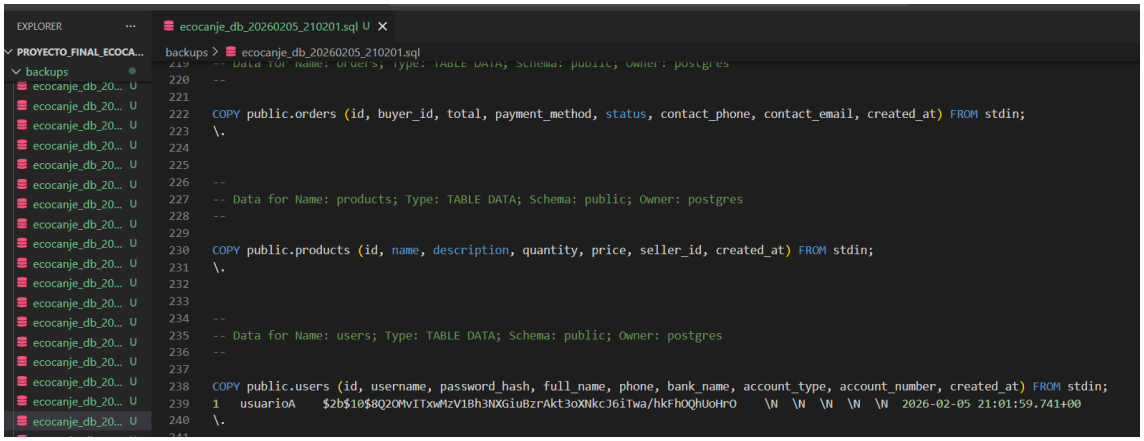
Antes:

ecocanje_db_20260205_204757.sql se realizó antes de que se creen los usuarios A y B por eso está vacio

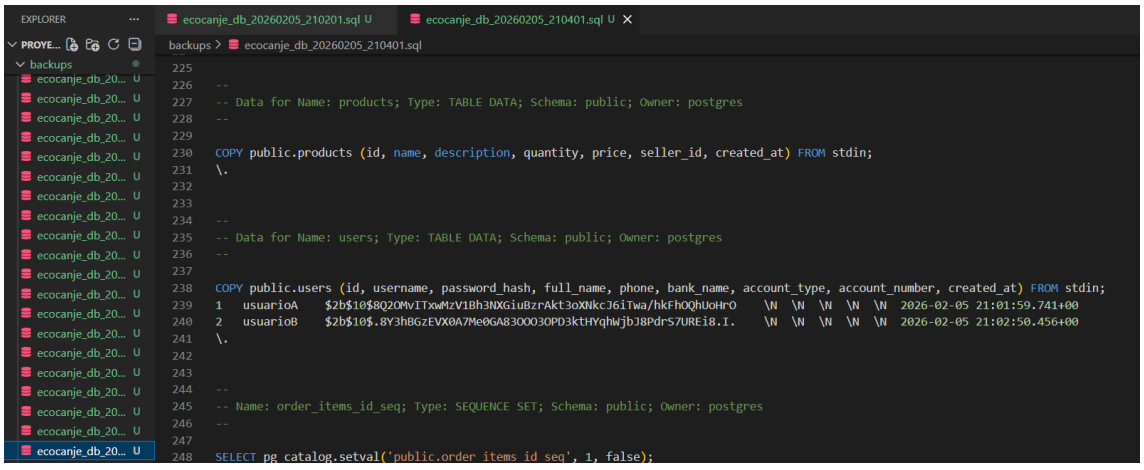


Después:

ecocanje_db_20260205_210201.sql se realizó después de que se creen los usuarios A por lo que ya contiene información



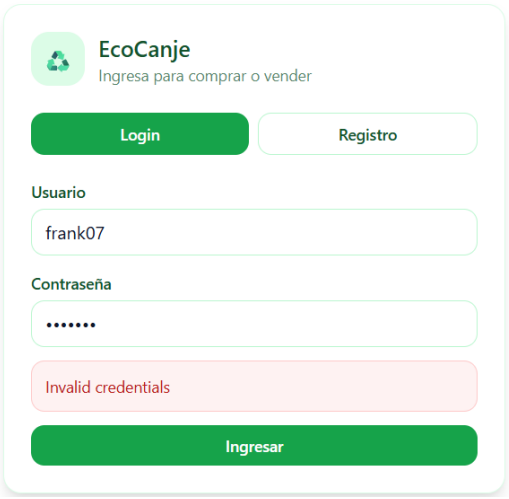
ecocanje_db_20260205_210401.sql se realizó después de que se creen los usuarios B por lo que ya contiene información



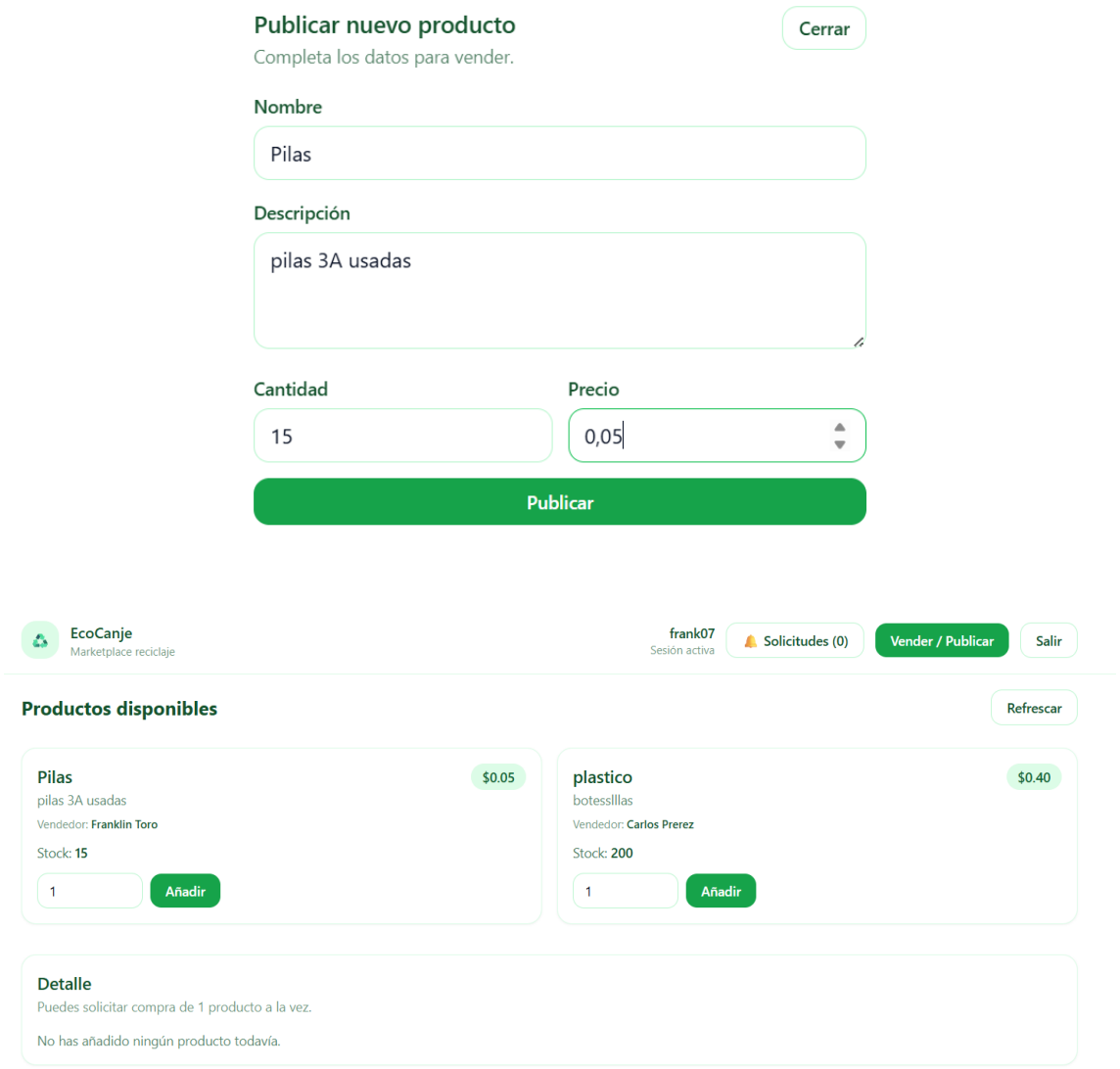
8.2 API (FRONTEND)

Registro nuevo usuario frank07

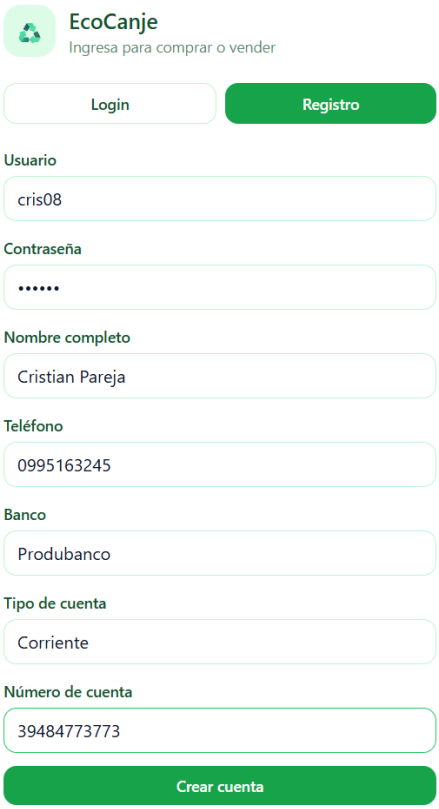
Login usuario frank07



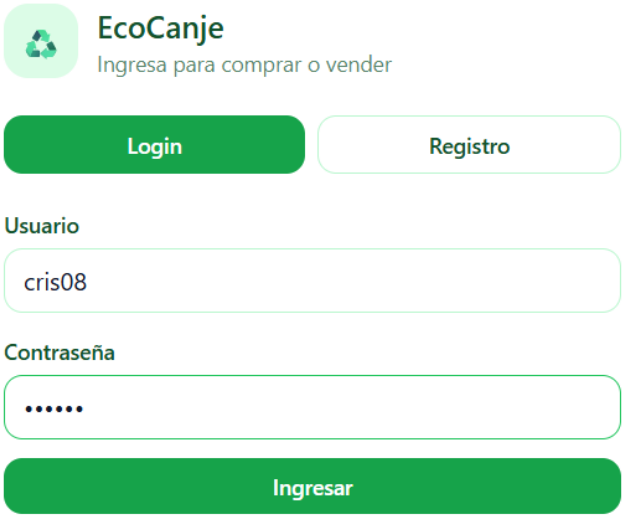
Crear nuevo producto frank07 (seller)



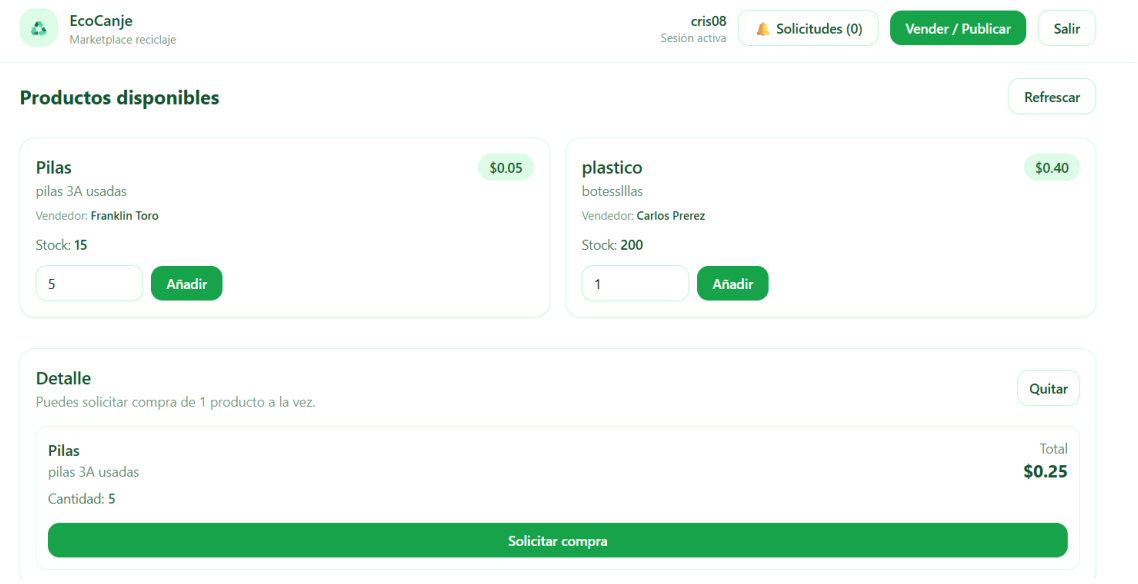
Registro nuevo usuario cris08

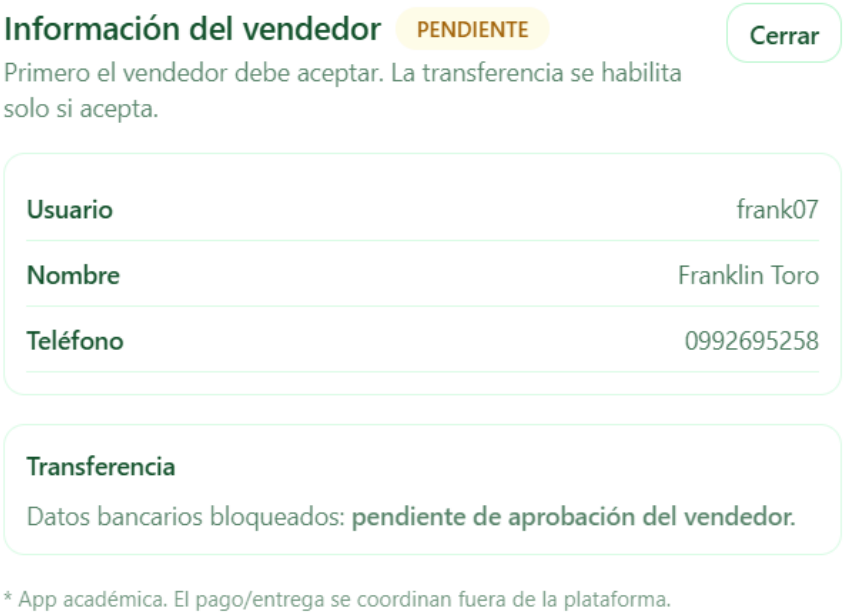


Login usuario cris08

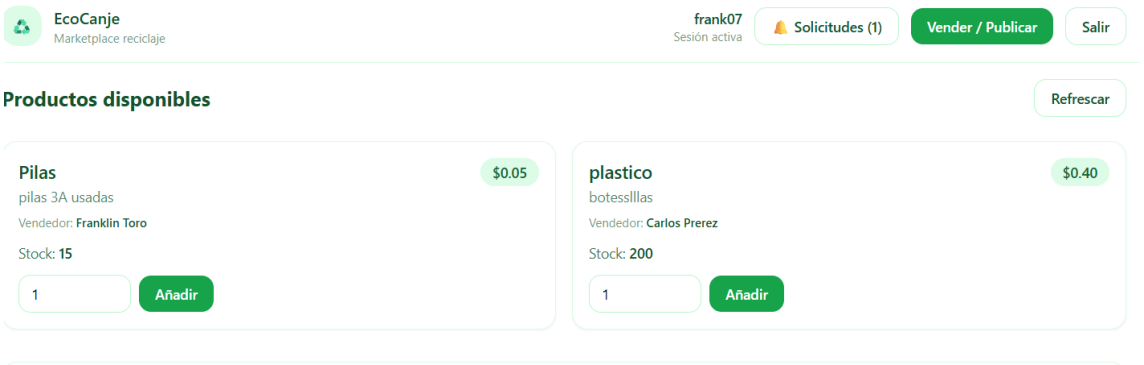


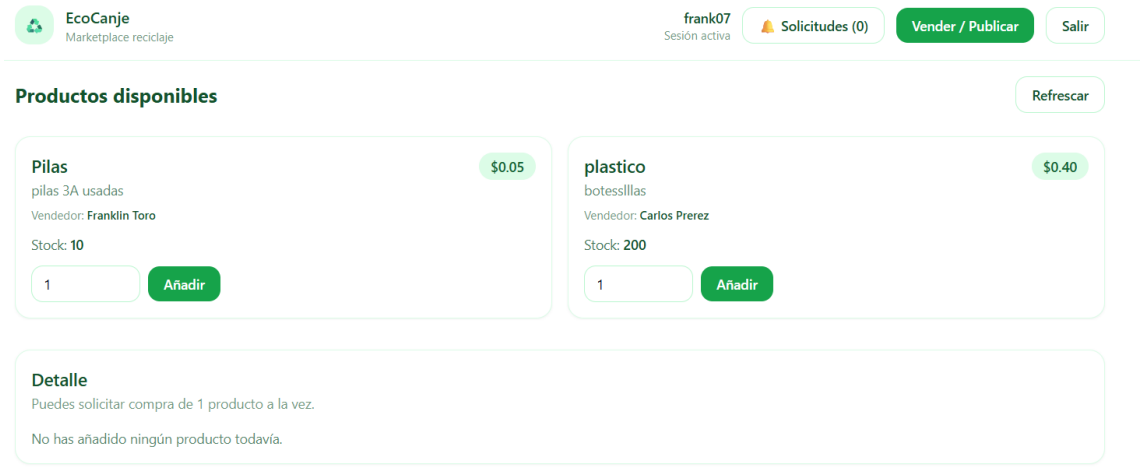
Cris08 compra 5 unidades del producto publicado por frank07 (buyer)



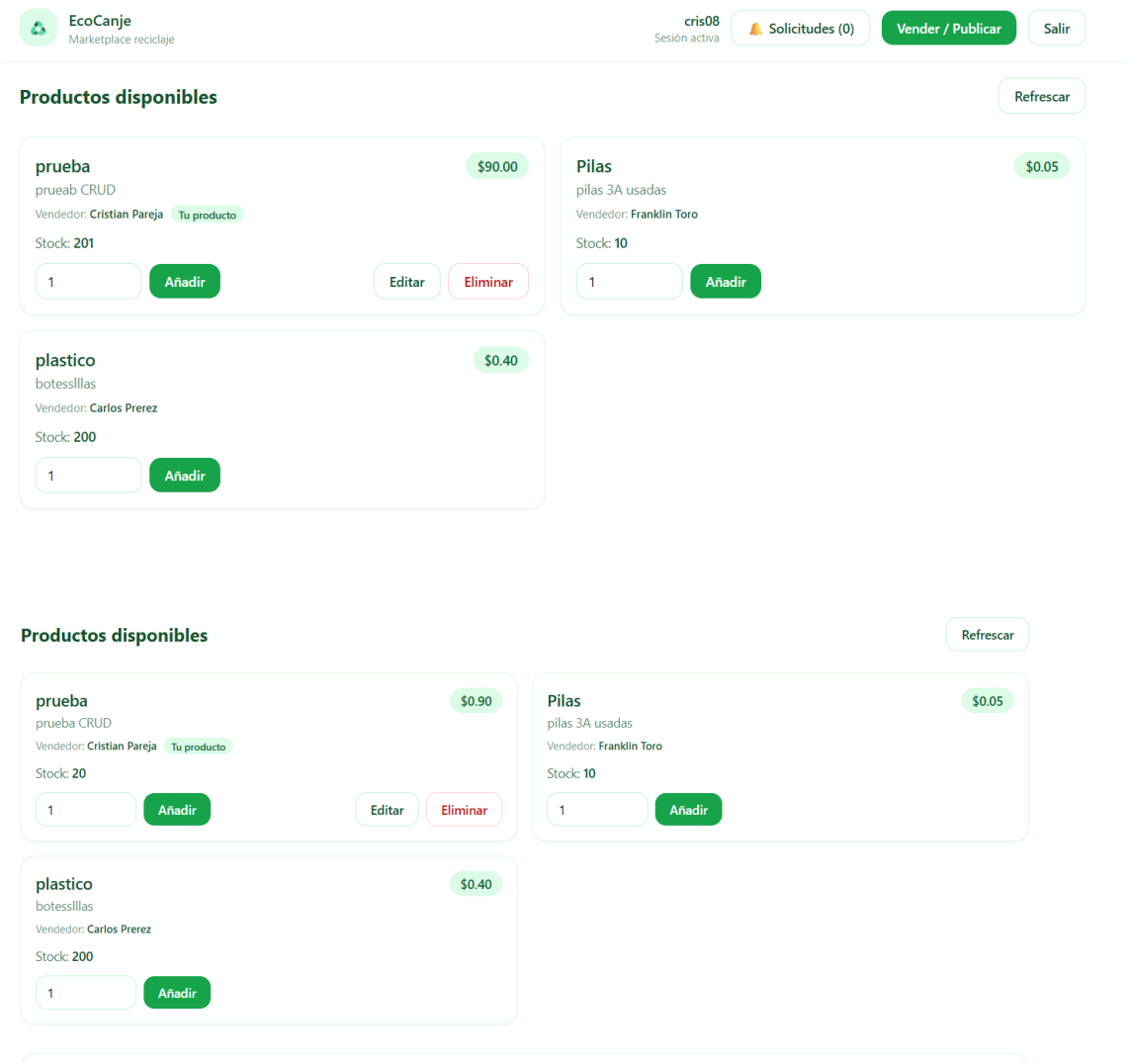


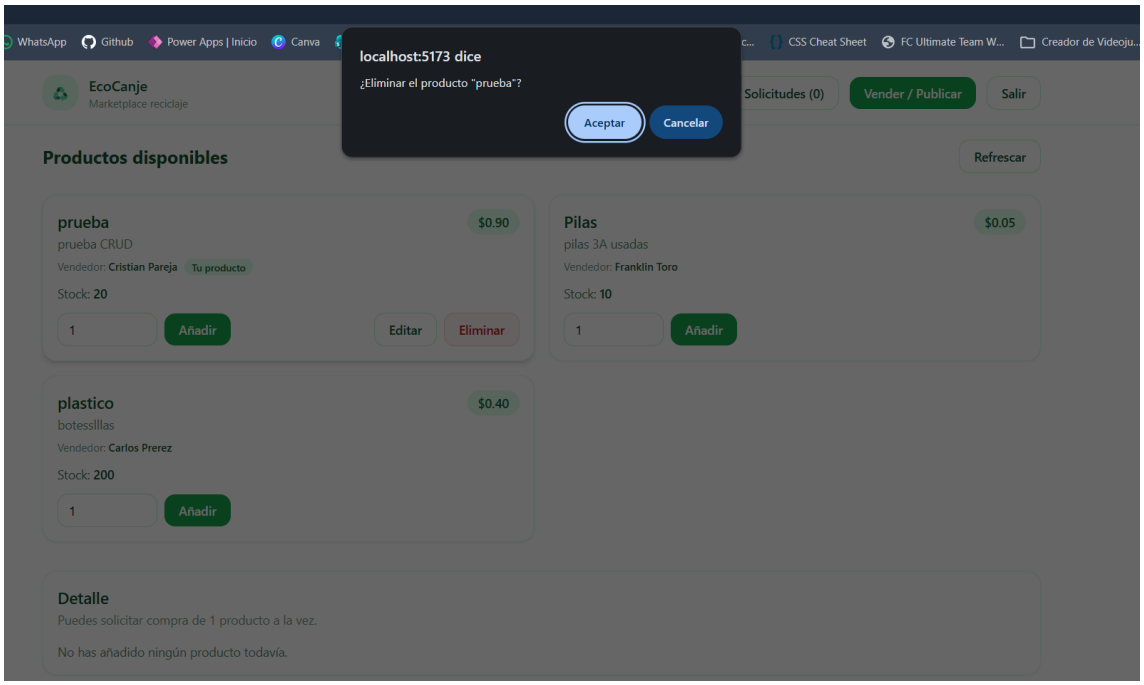
frank07 (buyer) deberá aceptar o rechazar la venta del producto





Usuario cris08 va a realizar un update y un delete en producto prueba





9. Conclusiones

- **La arquitectura por capas (routes, services, repositories)** permitió aislar errores sin afectar la base de datos ni la lógica de negocio.
- **El sistema de notificaciones basado en polling** es simple, efectivo y adecuado para un proyecto académico.
- **La auditoría nativa de PostgreSQL** permitió rastrear todas las operaciones críticas sin agregar complejidad al código.
- **La política de backups automatizados con cron** garantiza recuperación ante fallos y respalda los objetivos RPO y RTO definidos.
- **El control de flujo de compra (PENDING, ACCEPTED or REJECTED)** asegura integridad de stock y coherencia transaccional.