‘

SISTEMA GESTOR DE NEGOCIO: CASO INVENTARIO “HEIERLING GMBH”

Information Engineering

José Palacios Beortegui

Level 4 Bachelor’s in Applied Computer

29/04/2026

INDICE

1. Introducción

En este trabajo voy a explicar como he diseñado e implementado una base de datos para gestionar el inventario de Heierling GmbH, una tienda que fabrica y vende botas de esquí a medida. El sistema controla todo el ciclo de stock: que llega mercancía del proveedor hasta que se vende al cliente, pasando también por las reservas.

Lo que hace diferente a este negocio es que tiene tres zonas de almacenamiento. Primero, un almacén exterior grande donde se guarda la mayor parte del stock. Segundo, la propia tienda donde se tiene un par de casa modelo y talla para que el cliente pueda probarlo y comprarlo. Y tercero, una zorra de reservas dentro de la tienda donde se guardan botas para clientes que han pedido que se las aparten hasta que puedan venir. El sistema distingue estas ubicaciones de forma automática: si hay mas de una unidad el producto está en el almacén exterior, y si solo queda una es el ejemplar de tienda que avisa para reponer (Gehrke, (2002))

Para el diseño he aplicado el modelo relacional de (Codd, 1970) y normalización hasta la Tercera Forma Normal (3NF), lo que significa que cada dato se guarda en un solo sitio y no hay repeticiones innecesarias. Esto evita problemas cuando se hacen inserciones, modificaciones o borrados en la base de datos.

He usado PostgreSQL 16 como sistema de gestión de base de datos porque tiene buen soporte para funciones en PL/pgSQL, Python 3.11 para la aplicación y Docker para que todo funcione igual en cualquier máquina. El código está disponible en https://github.com/jjpp01x/heierling y sigue el estándar SQL (9075:2023, ISO/IEC).

2.Diagrama Entidad/Relación

2.1 Entidades y atributos

El diagrama E/R del sistema tiene cinco entidades que represento con rectángulos, tres relaciones con rombos y los atributos con óvalos, usando la notación Chen (Coronel, 2022). Las claves primarias llevan el atributo subrayado.

PROVEEDOR guarda los datos de las empresas que nos suministran las botas. Tiene cuatro atributos: NOMBRE_EMPRESA, TELEFONO, EMAIL y CIF. He usado el CIF como clave primaria porque es el identificador fiscal único de cada empresa. Lo he definido como VARCHAR(30) en lugar de VARCHAR(9) para que el sistema funcione también con proveedores de otros países, ya que cada país tiene su propio formato de identificación fiscal: CIF en España, VAT en Europa, EIN en Estados Unidos.

CATALOGO PRODUCTOS es la entidad central del sistema. Cada producto se identifica de forma única por su CODIGO_EAN, que sigue el estándar internacional EAN-13. El resto de atributos son MARCA, TIPO, NOMBRE_MODELO y TALLA. He decidido separar estos datos en campos distintos en lugar de guardar el nombre completo como un solo texto, porque así se puede filtrar el catálogo por cualquiera de estas dimensiones de forma independiente.

INVENTARIO solo tiene un atributo propio: UNIDADES. Todos los datos del producto ya están en CATALOGO PRODUCTOS, así que no tiene sentido repetirlos aquí. Esto es un principio básico de la normalización: cada dato en un único sitio (Codd, 1970). La ubicación física del stock se calcula automáticamente a partir de las unidades disponibles.

CLIENTE tiene cuatro atributos: ID_CLIENTE como clave primaria autogenerada, NOMBRE, DIRECCION y EMAIL. A diferencia del proveedor, los clientes no tienen un identificador natural único como el CIF, por eso uso una clave subrogada con SERIAL que PostgreSQL genera automáticamente.

2.2 Relaciones y cardinalidades

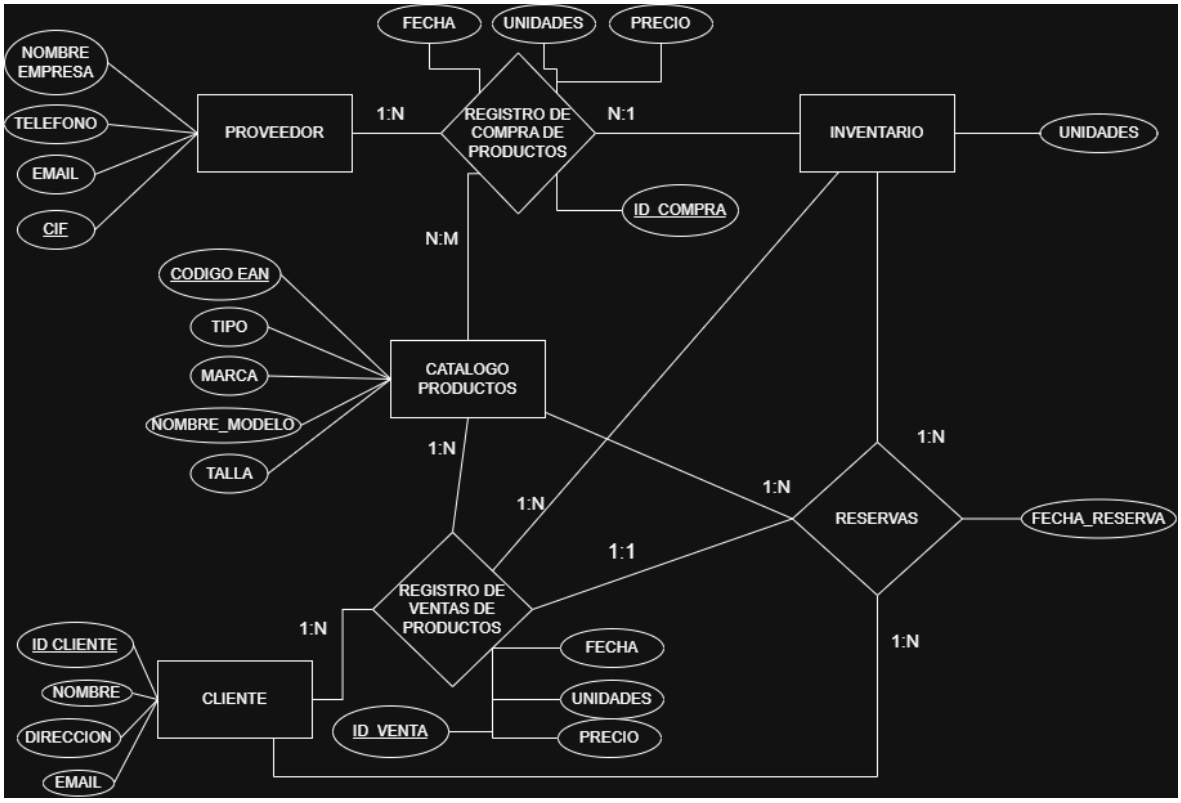
Las tres relaciones del diagrama representan las operaciones principales del negocio. Cada rombo tiene sus propios atributos y cardinalidades que indican cuantas veces pueden participar las entidades en cada relación (Coronel, 2022).

REGISTRO DE COMPRA DE PRODUCTOS conecta PROVEEDOR, CATALOGO PRODUCTOS e INVENTARIO. Un proveedor puede hacer muchas compras, pero cada compra es de un solo proveedor, de ahí la cardinalidad 1:N entre PROVEEDOR y el rombo. Muchas compras pueden ir al mismo inventario, por eso es N:1 entre el rombo e INVENTARIO. La relación entre CATALOGO PRODUCTOS y el rombo es N:M porque en una misma compra pueden llegar varios productos distintos y un mismo producto puede aparecer en muchas compras a lo largo del tiempo (Ramakrishnan & Gehrke, 2002). Los atributos propios de esta relación son FECHA, UNIDADES y PRECIO.

REGISTRO DE VENTAS DE PRODUCTOS conecta CATALOGO PRODUCTOS, INVENTARIO y CLIENTE. Un cliente puede comprar muchas veces (1:N) y un producto

puede venderse muchas veces (1:N). La relación con SEPARA RESERVAS es 1:1 porque cuando un cliente viene a recoger su reserva se genera exactamente una venta. Sus atributos propios son también FECHA, UNIDADES y PRECIO.

SEPARA RESERVAS es la relación que representa el tercer espacio de almacenamiento. Conecta INVENTARIO, CATALOGO PRODUCTOS y CLIENTE con cardinalidad 1:N en todos los casos, ya que un cliente puede tener varias reservas, un producto puede estar reservado para distintos clientes en diferentes tallas, y puede haber varias botas apartadas al mismo tiempo. Su único atributo propio es FECHA_RESERVA.



3. Modelo de Tablas

3.1 Conversión E/R a tablas

Al convertir el diagrama a tablas hay que tomar decisiones técnicas que van más allá de simplemente traducir entidades. Aqui explico las más importantes y por qué las he tomado (Coronel & Morris, 2022).

Las restricciones de cada columna se expresan con NOT NULL para columnas obligatorias, UNIQUE para valores no repetibles, y PRIMARY KEY para la clave principal, que implica automáticamente NOT NULL y UNIQUE. Las claves ajenas se expresan con REFERENCES indicando la tabla y columna que referencian.

PostgreSQL frente a MySQL. He elegido PostgreSQL porque tiene mejor soporte para funciones procedurales en PL/pgSQL y cumple el estándar SQL de forma más estricta. MySQL habría funcionado, pero tiene menos capacidad para meter lógica de negocio directamente en la base de datos, que es lo que necesito para las funciones de este proyecto.

CIF como clave primaria en PROVEEDOR. El CIF identifica de forma única a cada empresa y no necesita una clave artificial adicional. Lo he definido como VARCHAR(30) en lugar de VARCHAR(9) porque así funciona con proveedores de cualquier país sin tener que cambiar el esquema.

CHAR(13) para CODIGO_EAN. El estándar EAN-13 garantiza que todos los códigos tienen exactamente 13 dígitos. Si uso CHAR en lugar de VARCHAR, PostgreSQL rechaza automáticamente cualquier código con una longitud diferente, lo que me ahorra tener que hacer esa comprobación en el código de la aplicación (PostgreSQL Global Development Group, 2024).

SERIAL como clave primaria en CLIENTE. Los clientes no tienen ningún identificador natural fiable: el DNI puede cambiar, el email puede repetirse en una misma familia y el teléfono tampoco es único. Por eso uso una clave subrogada que PostgreSQL genera automáticamente.

ON DELETE RESTRICT en lugar de CASCADE. Si usara CASCADE, borrar un proveedor borraría automáticamente todas sus compras, y borrar un producto borraría todas sus ventas. Con RESTRICT eso no puede pasar y el historial de operaciones queda protegido (Ramakrishnan & Gehrke, 2002).

Baja lógica en lugar de DELETE físico. Cuando un producto tiene historial de compras o ventas, la función baja producto no lo borra de la base de datos, sino que emite un aviso y lo mantiene. Así los informes anuales siguen siendo correctos, aunque el producto ya no se venda.

UPSERT en compra producto. En lugar de hacer un INSERT y un UPDATE por separado para actualizar el inventario, uso ON CONFLICT DO UPDATE que hace las dos cosas en una sola operación atómica. Si el producto ya tiene stock lo incrementa, y si no existe lo crea, sin posibilidad de que quede a medias si algo falla (Development, Group PostgreSQL Global, 2024)

SHA-256 para las contraseñas. Las contraseñas nunca se guardan en texto plano sino como hash SHA-256. MD5 habría sido más sencillo, pero lleva comprometido desde 2004 y no es seguro para credenciales. SHA-256 es el estándar recomendado actualmente (9075:2023, ISO/IEC).

3.3 Sentencias CREATE TABLE

A continuación, están las sentencias DDL del esquema. El orden es importante: las tablas referenciadas por claves ajenas tienen que crearse antes que las que las referencian.

Tabla PROVEEDOR

CIF como PK natural con VARCHAR(30) para admitir formatos internacionales. Todo NOT NULL porque un proveedor sin datos de contacto no tiene utilidad.

```
-- Tabla que almacena los proveedores que suministran productos a la tienda.
CREATE TABLE proveedor (
    cif             VARCHAR(30)   NOT NULL,
    nombre_empresa VARCHAR(150) NOT NULL,
    telefono        VARCHAR(20)  NOT NULL,
    email           VARCHAR(100) NOT NULL,
    PRIMARY KEY(cif)
);
```

Tabla CATALOGO PRODUCTOS

CHAR(13) para que PostgreSQL rechace EANs con longitud incorrecta. NUMERIC(3,1) para la talla permite valores como 42.5.

```
-- Tabla que almacena el catálogo de referencias de botas, con una fila por
cada combinación de modelo y talla.
CREATE TABLE catalogo_productos (
    codigo_ean     CHAR(13)     NOT NULL,
    nombre_modelo  VARCHAR(200) NOT NULL,
    marca          VARCHAR(100) NOT NULL,
    tipo           VARCHAR(60)  NOT NULL,
    talla          NUMERIC(3,1) NOT NULL,
    PRIMARY KEY (codigo_ean)
);
```

Tabla INVENTARIO

La PK es el propio CODIGO_EAN porque cada producto tiene exactamente un registro de inventario. DEFAULT 0 para que empiece sin unidades al crear el producto.

```
-- Tabla que almacena el número de unidades disponibles en stock para cada
producto del catálogo.
CREATE TABLE inventario (
    codigo_ean  CHAR(13) NOT NULL,
    unidades    INT      NOT NULL DEFAULT 0,
    PRIMARY KEY (codigo_ean),
    FOREIGN KEY (codigo_ean) REFERENCES catalogo_productos(codigo_ean)
);
```

Tabla CLIENTE

SERIAL genera automaticamente un ID único incremental. Los clientes no tienen identificador natural fiable.

```
-- Tabla que almacena los datos de los clientes de la tienda.
```

```
CREATE TABLE cliente (
    id_cliente   SERIAL       PRIMARY KEY,
    nombre       VARCHAR(150) NOT NULL,
    direccion    VARCHAR(250) NOT NULL,
    email        VARCHAR(100) NOT NULL
);
```

Tabla REGISTRO_COMPRA

Los CHECK constraints impiden registrar compras con unidades o precios negativos. Nota que el CIF es VARCHAR(30) para coincidir con la tabla proveedor.

```
-- Tabla que almacena el historial de compras realizadas a proveedores, con
unidades adquiridas y precio unitario.
CREATE TABLE registro_compra (
    id_compra    SERIAL         PRIMARY KEY,
    cif          VARCHAR(9)     NOT NULL,
    codigo_ean   VARCHAR(13)    NOT NULL,
    fecha        DATE           NOT NULL DEFAULT CURRENT_DATE,
    unidades     INT            NOT NULL,
    precio       NUMERIC(10,2)  NOT NULL,
    FOREIGN KEY (cif)         REFERENCES proveedor(cif),
    FOREIGN KEY (codigo_ean)  REFERENCES catalogo_productos(codigo_ean),
    CONSTRAINT chk_unidades   CHECK (unidades > 0),
    CONSTRAINT chk_precio     CHECK (precio > 0)
);
```

Tabla REGISTRO_VENTA

Misma estructura que registro_compra pero referenciando al cliente en lugar del proveedor.

```
-- Tabla que almacena el historial de ventas realizadas a clientes, con
unidades vendidas y precio unitario.
CREATE TABLE registro_venta (
    id_venta     SERIAL         PRIMARY KEY,
    id_cliente   INT            NOT NULL,
    codigo_ean   VARCHAR(13)    NOT NULL,
    fecha        DATE           NOT NULL DEFAULT CURRENT_DATE,
    unidades     INT            NOT NULL,
    precio       NUMERIC(10,2)  NOT NULL,
    FOREIGN KEY (id_cliente)  REFERENCES cliente(id_cliente),
    FOREIGN KEY (codigo_ean)  REFERENCES catalogo_productos(codigo_ean),
    CONSTRAINT chk_unidades   CHECK (unidades > 0),
    CONSTRAINT chk_precio     CHECK (precio > 0)
);
```

Tabla RESERVAS

DEFAULT CURRENT_DATE registra automaticamente el día en que se hace la reserva si no se especifica fecha.

```
-- Tabla que almacena las reservas activas de botas apartadas por clientes
pendientes de conversión en venta.
CREATE TABLE reservas (
    id_reserva     SERIAL         PRIMARY KEY,
    id_cliente     INT            NOT NULL,
    codigo_ean     VARCHAR(13)    NOT NULL,
    fecha_reserva  DATE           NOT NULL DEFAULT CURRENT_DATE,
    FOREIGN KEY (id_cliente)  REFERENCES cliente(id_cliente),
    FOREIGN KEY (codigo_ean)  REFERENCES catalogo_productos(codigo_ean)
);
```

Tabla USUARIOS

El campo password guarda el hash SHA-256, nunca el texto plano. El campo activo permite desactivar usuarios sin borrarlos. El CHECK en rol garantiza que solo pueden existir dos roles en el sistema.

```
-- Tabla que almacena los usuarios del sistema con sus credenciales hasheadas
y rol de acceso.
CREATE TABLE IF NOT EXISTS usuarios (
    id_usuario  SERIAL       PRIMARY KEY,
    usuario     VARCHAR(50)  NOT NULL UNIQUE,
    password    VARCHAR(64)  NOT NULL,
    rol         VARCHAR(20)  NOT NULL DEFAULT 'empleado',
    activo      BOOLEAN      NOT NULL DEFAULT TRUE,
    CONSTRAINT chk_rol CHECK (rol IN ('admin', 'empleado'))
);

-- Insertar usuario admin por defecto
-- La contraseña 'admin1234' se guarda como hash SHA-256
INSERT INTO usuarios (usuario, password, rol)
VALUES (
    'admin',
    encode(sha256('He1erl!ng24'::bytea), 'hex'),
    'admin'
);

-- Insertar usuario empleado por defecto
INSERT INTO usuarios (usuario, password, rol)
VALUES (
    'empleado',
    encode(sha256('sk1B00t$24'::bytea), 'hex'),
    'empleado'
);
```

3.4 Índices

Los índices son mecanismos basados en búsqueda dicotómica que aceleran enormemente la búsqueda de filas en una tabla. PostgreSQL crea automáticamente un índice para la clave primaria y para cada restricción UNIQUE. Adicionalmente he creado 12 índices manuales sobre las columnas mas consultadas para optimizar el rendimiento de las operaciones más frecuentes del sistema.

```
CREATE INDEX IF NOT EXISTS idx_compra_codigo_ean
    ON registro_compra(codigo_ean);

CREATE INDEX IF NOT EXISTS idx_compra_fecha
    ON registro_compra(fecha);

CREATE INDEX IF NOT EXISTS idx_venta_codigo_ean
    ON registro_venta(codigo_ean);

CREATE INDEX IF NOT EXISTS idx_venta_fecha
    ON registro_venta(fecha);

CREATE INDEX IF NOT EXISTS idx_venta_id_cliente
    ON registro_venta(id_cliente);

CREATE INDEX IF NOT EXISTS idx_reservas_id_cliente
    ON reservas(id_cliente);

CREATE INDEX IF NOT EXISTS idx_catalogo_marca
    ON catalogo_productos(marca);
```

4. Modelo Funcional

He implementado todas las operaciones de negocio como funciones PL/pgSQL de PostgreSQL. Esto tiene una ventaja importante: la lógica queda dentro de la base de datos y funciona igual independientemente de si se llama desde Python, desde psql o desde cualquier otra aplicación (Development, Group PostgreSQL Global, 2024). Todas las funciones comprueban los datos de entrada antes de ejecutar la operación y lanzan excepciones descriptivas si algo no está bien. He elegido funciones en todos los casos por que necesitan retornar información al llamador o por que usan dentro de disparadores. Este enfoque tiene una ventaja importante: la lógica queda dentro del sistema gestor de base de datos y funcional igual independientemente de di se llama desde Python, desde psql o desde cualquier otra aplicación. Todas las funciones comprueban los datos de entrada antes de ejecutar la operación y lanzan excepciones descriptivas si algo no esta bien.

4.1 Alta de un producto

Añade un nuevo producto al catálogo. Antes de insertar comprueba que el EAN tiene 13 caracteres, que la talla esta entre 35 y 50, que el producto no existe ya y que ninguno de los campos de texto está vacío.

```
CREATE OR REPLACE FUNCTION alta_producto(
    p_codigo_ean    CHAR(13),
    p_nombre_modelo VARCHAR(200),
    p_marca         VARCHAR(100),
    p_tipo          VARCHAR(60),
    p_talla         NUMERIC(3,1)
)
RETURNS VOID AS $$
BEGIN
    -- Validar longitud EAN
    IF LENGTH(TRIM(p_codigo_ean)) <> 13 THEN
        RAISE EXCEPTION 'El codigo EAN debe tener exactamente 13 caracteres.';
    END IF;

    -- Validar talla en rango válido
    IF p_talla < 35 OR p_talla > 50 THEN
        RAISE EXCEPTION 'La talla % no es válida. Debe estar entre 35 y 50.',
p_talla;
    END IF;

    -- Validar que el producto no existe ya
    IF EXISTS (
        SELECT 1 FROM catalogo_productos
        WHERE codigo_ean = p_codigo_ean
    ) THEN
        RAISE EXCEPTION 'Ya existe un producto con el EAN %.', p_codigo_ean;
    END IF;

    -- Validar que nombre, marca y tipo no están vacíos
    IF TRIM(p_nombre_modelo) = '' THEN
        RAISE EXCEPTION 'El nombre del modelo no puede estar vacío.';
    END IF;

    IF TRIM(p_marca) = '' THEN
        RAISE EXCEPTION 'La marca no puede estar vacía.';
    END IF;

    IF TRIM(p_tipo) = '' THEN
```

```
        RAISE EXCEPTION 'El tipo no puede estar vacío.';
    END IF;

    INSERT INTO catalogo_productos (
        codigo_ean,
        nombre_modelo,
        marca,
        tipo,
        talla
    )
    VALUES (
        p_codigo_ean,
        p_nombre_modelo,
        p_marca,
        p_tipo,
        p_talla
    );

    RAISE NOTICE 'Producto % añadido correctamente.', p_nombre_modelo;
END;
$$ LANGUAGE plpgsql;
```

4.2 Baja de un producto

Da de baja un producto del catálogo con tres niveles de comprobación. Primero verifica que el producto existe. Después comprueba que no tiene reservas activas porque eso significaría que hay un cliente esperando. Luego comprueba el stock: si quedan unidades no se puede dar de baja. Y si el producto tiene historial de compras o ventas no se borra físicamente, sino que se mantiene el registro para que los informes históricos sigan siendo correctos.

```
CREATE OR REPLACE FUNCTION baja_producto(p_codigo_ean CHAR(13))
RETURNS VOID AS $$
DECLARE
    v_stock INT;
BEGIN
    IF NOT EXISTS (SELECT 1 FROM catalogo_productos
        WHERE codigo_ean = p_codigo_ean) THEN
        RAISE EXCEPTION 'Producto % no encontrado.', p_codigo_ean;
    END IF;
    IF EXISTS (SELECT 1 FROM reservas
        WHERE codigo_ean = p_codigo_ean) THEN
        RAISE EXCEPTION 'El producto % tiene reservas activas.', p_codigo_ean;
    END IF;
    SELECT unidades INTO v_stock FROM inventario
    WHERE codigo_ean = p_codigo_ean;
    IF v_stock > 0 THEN
        RAISE EXCEPTION 'El producto % tiene % unidades en inventario.',
            p_codigo_ean, v_stock;
    END IF;
    IF EXISTS (SELECT 1 FROM registro_compra WHERE codigo_ean = p_codigo_ean)
    OR EXISTS (SELECT 1 FROM registro_venta WHERE codigo_ean = p_codigo_ean)
THEN
        RAISE NOTICE 'Producto % con historial. Se mantiene como inactivo.',
            p_codigo_ean;
        RETURN;
    END IF;
    DELETE FROM inventario WHERE codigo_ean = p_codigo_ean;
    DELETE FROM catalogo_productos WHERE codigo_ean = p_codigo_ean;
    RAISE NOTICE 'Producto % eliminado correctamente.', p_codigo_ean;
END;
$$ LANGUAGE plpgsql;
```

4.3 Renombrado de un producto

Cambia el nombre de un producto. Comprueba que existe, que el nuevo nombre no está vacio y que es diferente al nombre actual para evitar actualizaciones sin sentido.

```
CREATE OR REPLACE FUNCTION renombrar_producto(
    p_codigo_ean    CHAR(13),
    p_nuevo_nombre  VARCHAR(200)
)
RETURNS VOID AS $$
DECLARE
    v_nombre_anterior VARCHAR(200);
BEGIN
    -- Comprobar que el producto existe
    IF NOT EXISTS (
        SELECT 1 FROM catalogo_productos
        WHERE codigo_ean = p_codigo_ean
    ) THEN
        RAISE EXCEPTION 'Producto % no encontrado.', p_codigo_ean;
    END IF;

    -- Comprobar que el nuevo nombre no está vacío
    IF TRIM(p_nuevo_nombre) = '' THEN
        RAISE EXCEPTION 'El nuevo nombre no puede estar vacío.';
    END IF;

    -- Guardar el nombre anterior para el mensaje de confirmación
    SELECT nombre_modelo INTO v_nombre_anterior
    FROM catalogo_productos
    WHERE codigo_ean = p_codigo_ean;

    -- Comprobar que el nuevo nombre es distinto al actual
    IF v_nombre_anterior = p_nuevo_nombre THEN
        RAISE EXCEPTION 'El nuevo nombre es igual al actual.';
    END IF;

    -- Actualizar el nombre
    UPDATE catalogo_productos
    SET nombre_modelo = p_nuevo_nombre
    WHERE codigo_ean = p_codigo_ean;

    RAISE NOTICE 'Producto renombrado de "%" a "%".',
        v_nombre_anterior, p_nuevo_nombre;
END;
$$ LANGUAGE plpgsql;
```

4.4 Compra de un producto

Registra una entrada de stock. Comprueba que el proveedor y el producto existen, que las unidades y el precio son positivos y que la fecha no es futura. Luego registra la compra y actualiza el inventario con UPSERT: si el producto ya tiene stock lo incrementa, si no lo crea directamente.

```
CREATE OR REPLACE FUNCTION compra_producto(
    p_cif        VARCHAR(30),
    p_codigo_ean CHAR(13),
    p_unidades   INT,
    p_precio     NUMERIC(10,2),
    p_fecha      DATE DEFAULT CURRENT_DATE
)
RETURNS VOID AS $$
BEGIN
    -- Comprobar que el proveedor existe
    IF NOT EXISTS (
        SELECT 1 FROM proveedor
```

```
        WHERE cif = p_cif
    ) THEN
        RAISE EXCEPTION 'Proveedor con CIF % no encontrado.', p_cif;
    END IF;

    -- Comprobar que el producto existe
    IF NOT EXISTS (
        SELECT 1 FROM catalogo_productos
        WHERE codigo_ean = p_codigo_ean
    ) THEN
        RAISE EXCEPTION 'Producto % no encontrado.', p_codigo_ean;
    END IF;

    -- Comprobar que las unidades son positivas
    IF p_unidades <= 0 THEN
        RAISE EXCEPTION 'Las unidades deben ser un número positivo.';
    END IF;

    -- Comprobar que el precio es positivo
    IF p_precio <= 0 THEN
        RAISE EXCEPTION 'El precio debe ser un número positivo.';
    END IF;

    -- Comprobar que la fecha no es futura
    IF p_fecha > CURRENT_DATE THEN
        RAISE EXCEPTION 'La fecha de compra no puede ser futura.';
    END IF;

    -- Registrar la compra
    INSERT INTO registro_compra (
        cif,
        codigo_ean,
        fecha,
        unidades,
        precio
    )
    VALUES (
        p_cif,
        p_codigo_ean,
        p_fecha,
        p_unidades,
        p_precio
    );

    -- Actualizar inventario con UPSERT
    INSERT INTO inventario (codigo_ean, unidades)
    VALUES (p_codigo_ean, p_unidades)
    ON CONFLICT (codigo_ean)
    DO UPDATE SET unidades = inventario.unidades + p_unidades;

    RAISE NOTICE 'Compra registrada: % unidades de % a %.€/ud.',
        p_unidades, p_codigo_ean, p_precio;
END;
$$ LANGUAGE plpgsql;
```

4.5 Venta de un producto

Registra una salida de stock. Además de las comprobaciones habituales de existencia y datos positivos, verifica que hay suficiente stock antes de proceder. Si después de la venta queda solo 1 unidad, emite un aviso porque eso significa que solo queda el ejemplar de tienda y hay que reponer desde el almacén exterior.

```
CREATE OR REPLACE FUNCTION venta_producto(
    p_id_cliente INT,
    p_codigo_ean CHAR(13),
    p_unidades   INT,
    p_precio     NUMERIC(10,2),
    p_fecha      DATE DEFAULT CURRENT_DATE
```

```
)
RETURNS VOID AS $$
DECLARE
    v_stock INT;
BEGIN
    -- Comprobar que el cliente existe
    IF NOT EXISTS (
        SELECT 1 FROM cliente
        WHERE id_cliente = p_id_cliente
    ) THEN
        RAISE EXCEPTION 'Cliente con ID % no encontrado.', p_id_cliente;
    END IF;

    -- Comprobar que el producto existe
    IF NOT EXISTS (
        SELECT 1 FROM catalogo_productos
        WHERE codigo_ean = p_codigo_ean
    ) THEN
        RAISE EXCEPTION 'Producto % no encontrado.', p_codigo_ean;
    END IF;

    -- Comprobar que las unidades son positivas
    IF p_unidades <= 0 THEN
        RAISE EXCEPTION 'Las unidades deben ser un número positivo.';
    END IF;

    -- Comprobar que el precio es positivo
    IF p_precio <= 0 THEN
        RAISE EXCEPTION 'El precio debe ser un número positivo.';
    END IF;

    -- Comprobar que la fecha no es futura
    IF p_fecha > CURRENT_DATE THEN
        RAISE EXCEPTION 'La fecha de venta no puede ser futura.';
    END IF;

    -- Comprobar stock suficiente
    SELECT unidades INTO v_stock
    FROM inventario
    WHERE codigo_ean = p_codigo_ean;

    IF v_stock IS NULL OR v_stock < p_unidades THEN
        RAISE EXCEPTION 'Stock insuficiente: disponible=%, solicitado=%.',
            COALESCE(v_stock, 0), p_unidades;
    END IF;

    -- Registrar la venta
    INSERT INTO registro_venta (
        id_cliente,
        codigo_ean,
        fecha,
        unidades,
        precio
    )
    VALUES (
        p_id_cliente,
        p_codigo_ean,
        p_fecha,
        p_unidades,
        p_precio
    );

    -- Descontar stock del inventario
    UPDATE inventario
    SET unidades = unidades - p_unidades
    WHERE codigo_ean = p_codigo_ean;

    -- Avisar si el stock queda en 0
    IF v_stock - p_unidades = 0 THEN
```

```
        RAISE NOTICE 'AVISO: El producto % ha agotado su stock.',
p_codigo_ean;
    END IF;

    RAISE NOTICE 'Venta registrada: % unidades de % al cliente %.',
        p_unidades, p_codigo_ean, p_id_cliente;
END;
$$ LANGUAGE plpgsql;
```

Para evitar errores cuando un productos no tiene ningún registro en la tabla inventario, se usa COALESCE(V_STOCK, 0), que retorna el primer argumento que no sea nulo. Si v_stock es nulo por que el producto no existe en el inventario, COALESCE devuelve 0 en su lugar, lo que permite que el mensaje de error sea coherente indicando que hay 0 unidades disponibles.

4.6 Listado de existencias de un producto

Devuelve las existencias de un producto concreto con todos sus datos haciendo un JOIN entre inventario y catalogo_productos. Además avisa cuando queda solo 1 unidad, que es cuando el stock ha bajado al nivel del ejemplar de tienda y hay que reponer.

```
CREATE OR REPLACE FUNCTION existencias_producto(p_codigo_ean CHAR(13))
RETURNS TABLE(codigo_ean CHAR(13), nombre_modelo VARCHAR(200),
    marca VARCHAR(100), tipo VARCHAR(60),
    talla NUMERIC(3,1), unidades INT) AS $$
BEGIN
    IF NOT EXISTS (SELECT 1 FROM catalogo_productos
        WHERE catalogo_productos.codigo_ean = p_codigo_ean) THEN
        RAISE EXCEPTION 'Producto % no encontrado.', p_codigo_ean;
    END IF;
    IF NOT EXISTS (SELECT 1 FROM inventario
        WHERE inventario.codigo_ean = p_codigo_ean) THEN
        RAISE EXCEPTION 'El producto % no tiene unidades en inventario.',
            p_codigo_ean;
    END IF;
    RETURN QUERY
    SELECT cp.codigo_ean, cp.nombre_modelo, cp.marca,
           cp.tipo, cp.talla, i.unidades
    FROM catalogo_productos cp
    JOIN inventario i ON i.codigo_ean = cp.codigo_ean
    WHERE cp.codigo_ean = p_codigo_ean;
    -- Avisar si solo queda 1 unidad (hay que reponer)
    IF (SELECT i.unidades FROM inventario i
        WHERE i.codigo_ean = p_codigo_ean) < 1 THEN
        RAISE NOTICE 'AVISO: El producto % tiene 1 unidad en stock.',
            p_codigo_ean;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

4.7 Compras anuales

Devuelve un resumen de todas las compras de un año agrupado por producto y ordenado por importe total de mayor a menor. Comprueba que el año es válido y que hay registros para ese año antes de ejecutar la consulta.

```
CREATE OR REPLACE FUNCTION compras_anuales(p_año INT)
RETURNS TABLE(codigo_ean CHAR(13), nombre_modelo VARCHAR(200),
    marca VARCHAR(100), total_unidades BIGINT,
    importe_total NUMERIC) AS $$
BEGIN
    IF p_año < 2000 OR p_año > EXTRACT(YEAR FROM CURRENT_DATE) THEN
        RAISE EXCEPTION 'El año % no es válido.', p_año;
    END IF;
    IF NOT EXISTS (SELECT 1 FROM registro_compra
        WHERE EXTRACT(YEAR FROM fecha) = p_año) THEN
```

```
        RAISE EXCEPTION 'No hay compras registradas en el año %.', p_año;
    END IF;
    RETURN QUERY
    SELECT cp.codigo_ean, cp.nombre_modelo, cp.marca,
           SUM(rc.unidades)::BIGINT AS total_unidades,
           SUM(rc.unidades * rc.precio) AS importe_total
    FROM registro_compra rc
    JOIN catalogo_productos cp ON cp.codigo_ean = rc.codigo_ean
    WHERE EXTRACT(YEAR FROM rc.fecha) = p_año
    GROUP BY cp.codigo_ean, cp.nombre_modelo, cp.marca
    ORDER BY importe_total DESC;
END;
$$ LANGUAGE plpgsql;
```

4.8 Ventas anuales

Igual que compras_anuales pero para ventas. Permite ver el importe facturado por producto en un año concreto.

```
CREATE OR REPLACE FUNCTION ventas_anuales(p_año INT)
RETURNS TABLE(codigo_ean CHAR(13), nombre_modelo VARCHAR(200),
    marca VARCHAR(100), total_unidades BIGINT,
    importe_total NUMERIC) AS $$
BEGIN
    IF p_año < 2000 OR p_año > EXTRACT(YEAR FROM CURRENT_DATE) THEN
        RAISE EXCEPTION 'El año % no es válido.', p_año;
    END IF;
    IF NOT EXISTS (SELECT 1 FROM registro_venta
        WHERE EXTRACT(YEAR FROM fecha) = p_año) THEN
        RAISE EXCEPTION 'No hay ventas registradas en el año %.', p_año;
    END IF;
    RETURN QUERY
    SELECT cp.codigo_ean, cp.nombre_modelo, cp.marca,
           SUM(rv.unidades)::BIGINT AS total_unidades,
           SUM(rv.unidades * rv.precio) AS importe_total
    FROM registro_venta rv
    JOIN catalogo_productos cp ON cp.codigo_ean = rv.codigo_ean
    WHERE EXTRACT(YEAR FROM rv.fecha) = p_año
    GROUP BY cp.codigo_ean, cp.nombre_modelo, cp.marca
    ORDER BY importe_total DESC;
END;
$$ LANGUAGE plpgsql;
```

4.9 Cancelar reserva

Cancela una reserva y devuelve las unidades al inventario automaticamente. Esto es importante para que el stock no quede inconsistente si un cliente cambia de opinión.

```
CREATE OR REPLACE FUNCTION cancelar_reserva(p_id_reserva INT)
RETURNS VOID AS $$
DECLARE
    v_codigo_ean CHAR(13);
    v_id_cliente INT;
BEGIN
    IF NOT EXISTS (SELECT 1 FROM reservas
        WHERE id_reserva = p_id_reserva) THEN
        RAISE EXCEPTION 'Reserva con ID % no encontrada.', p_id_reserva;
    END IF;
    SELECT codigo_ean, id_cliente INTO v_codigo_ean, v_id_cliente
    FROM reservas WHERE id_reserva = p_id_reserva;
    DELETE FROM reservas WHERE id_reserva = p_id_reserva;
    UPDATE inventario SET unidades = unidades + 1
    WHERE codigo_ean = v_codigo_ean;
    RAISE NOTICE 'Reserva % cancelada. Unidades de % devueltas al
inventario.',
        p_id_reserva, v_codigo_ean;
```

```
END;
$$ LANGUAGE plpgsql;
```

4.10 Convertir reserva en venta

Esta es la función más compleja del sistema. Cuando un cliente viene a recoger su reserva, hace tres cosas en una sola operación atómica: registra la venta, descuenta el stock y elimina la reserva. Si cualquier paso falla, PostgreSQL revierte todo automaticamente y no queda nada a medias.

```
CREATE OR REPLACE FUNCTION convertir_reserva_en_venta(
    p_id_reserva INT, p_precio NUMERIC(10,2))
RETURNS VOID AS $$
DECLARE
    v_codigo_ean CHAR(13);
    v_id_cliente INT;
    v_stock INT;
BEGIN
    IF NOT EXISTS (SELECT 1 FROM reservas
        WHERE id_reserva = p_id_reserva) THEN
        RAISE EXCEPTION 'Reserva con ID % no encontrada.', p_id_reserva;
    END IF;
    IF p_precio <= 0 THEN
        RAISE EXCEPTION 'El precio debe ser un numero positivo.';
    END IF;
    SELECT codigo_ean, id_cliente INTO v_codigo_ean, v_id_cliente
    FROM reservas WHERE id_reserva = p_id_reserva;
    SELECT unidades INTO v_stock FROM inventario
    WHERE codigo_ean = v_codigo_ean;
    IF v_stock IS NULL OR v_stock <= 0 THEN
        RAISE EXCEPTION 'Sin stock disponible para el producto %.',
            v_codigo_ean;
    END IF;
    -- Registrar la venta
    INSERT INTO registro_venta (id_cliente, codigo_ean, fecha, unidades,
precio)
    VALUES (v_id_cliente, v_codigo_ean, CURRENT_DATE, 1, p_precio);
    -- Descontar stock
    UPDATE inventario SET unidades = unidades - 1
    WHERE codigo_ean = v_codigo_ean;
    -- Eliminar la reserva
    DELETE FROM reservas WHERE id_reserva = p_id_reserva;
    IF v_stock - 1 = 0 THEN
        RAISE NOTICE 'AVISO: El producto % ha agotado su stock.',
v_codigo_ean;
    END IF;
    RAISE NOTICE 'Reserva % convertida en venta para el cliente %.',
        p_id_reserva, v_id_cliente;
END;
$$ LANGUAGE plpgsql;
```

4.11 Inventario completo

Devuelve todo el inventario con la ubicación calculada automaticamente según las unidades: 1 unidad significa que está en la tienda, más de 1 en el almacén exterior, y 0 sin stock.

```
CREATE OR REPLACE FUNCTION inventario_completo()
RETURNS TABLE(codigo_ean CHAR(13), nombre_modelo VARCHAR(200),
    marca VARCHAR(100), tipo VARCHAR(60), talla NUMERIC(3,1),
    unidades INT, ubicación VARCHAR(20)) AS $$
BEGIN
    IF NOT EXISTS (SELECT 1 FROM inventario) THEN
        RAISE EXCEPTION 'El inventario esta vacio.';
    END IF;
```

```
    RETURN QUERY
    SELECT cp.codigo_ean, cp.nombre_modelo, cp.marca,
           cp.tipo, cp.talla, i.unidades,
        CASE
            WHEN i.unidades = 1 THEN 'TIENDA'::VARCHAR(20)
            WHEN i.unidades = 0 THEN 'SIN STOCK'::VARCHAR(20)
            ELSE 'ALMACEN EXTERIOR'::VARCHAR(20)
        END AS ubicación
    FROM inventario i
    JOIN catalogo_productos cp ON cp.codigo_ean = i.codigo_ean
    ORDER BY cp.marca, cp.nombre_modelo, cp.talla;
END;
$$ LANGUAGE plpgsql;
```

4.12 Vista de inventario

He creado una vista, donde se combina la tabla inventario con catalogo_productos mediante JOIN y calcula automaticamente la ubicación de cada producto según sus unidades. Esto permite consultar el inventario completo con un SELECT sin necesidad de escribir el JOIN cada vez.

```
CREATE OR REPLACE VIEW vista_inventario AS
SELECT
    cp.codigo_ean,
    cp.nombre_modelo,
    cp.marca,
    cp.tipo,
    cp.talla,
    i.unidades,
    CASE
        WHEN i.unidades = 1 THEN 'TIENDA'
        WHEN i.unidades = 0 THEN 'SIN STOCK'
        ELSE 'ALMACEN EXTERIOR'
    END AS ubicacion
FROM inventario i
JOIN catalogo_productos cp ON cp.codigo_ean = i.codigo_ean
ORDER BY cp.marca, cp.nombre_modelo, cp.talla;
```

5. Pruebas de validación

He probado todas las funciones en el sistema real para verificar que funcionan correctamente tanto en el caso normal como en los casos de error. La siguiente tabla recoge las pruebas realizadas con sus resultados (PostgreSQL Global Development Group, 2024).

Prueba	Entrada	Resultado esperado	Estado
Alta producto EAN invalido	EAN de 12 digitos	EXCEPTION: EAN debe tener 13 caracteres	OK
Alta producto talla fuera de rango	Talla 60.0	EXCEPTION: talla no valida	OK
Alta producto duplicado	EAN ya existente	EXCEPTION: ya existe un producto con ese EAN	OK
Alta producto correcto	Datos validos	Producto insertado en catálogo	OK
Compra proveedor inexistente	CIF no registrado	EXCEPTION: proveedor no encontrado	OK
Compra fecha futura	Fecha del dia siguiente	EXCEPTION: fecha no puede ser futura	OK
Compra correcta + UPSERT	Datos válidos, producto ya en inventario	Compra registrada y stock incrementado	OK
Venta sin stock suficiente	999 unidades	EXCEPTION: stock insuficiente	OK
Venta correcta	Datos validos	Venta registrada y stock decrementado	OK
Venta con 1 unidad restante	Ultimas unidades del almacén	NOTICE: solo queda 1 unidad en tienda	OK
Baja con reservas activas	EAN con reservas	EXCEPTION: tiene reservas activas	OK
Baja con stock	EAN con unidades	EXCEPTION: tiene unidades en inventario	OK
Baja con historial	EAN con compras o ventas	NOTICE: se mantiene como inactivo	OK
Convertir reserva en venta	ID reserva valido y precio	Venta registrada y reserva eliminada	OK
Cancelar reserva	ID reserva válido	Reserva eliminada y stock devuelto	OK

Login incorrecto	Contraseña erronea	Acceso denegado	OK
Login usuario inactivo	Usuario desactivado	Acceso denegado	OK
Empleado accede a función de admin	Opción restringida	Sin permisos para esta acción	OK
Contraseña débil al crear usuario	Sin mayusculas ni simbolos	EXCEPTION: no cumple requisitos de seguridad	OK

6. Seguridad

He añadido seguridad al sistema en cuatro capas distintas. La idea es que si una falla, las demás siguen protegiendo el sistema. Este concepto se conoce como defensa en profundidad (ISO/IEC 9075:2023).

6.1 Autenticación y control de acceso

Todos los módulos Python piden usuario y contraseña antes de hacer nada. Las credenciales se verifican contra la tabla usuarios de PostgreSQL. Si son incorrectas o el usuario esta inactivo, el programa termina sin mostrar ninguna opción.

```
def login():
    # Pedir credenciales al usuario
    usuario = input('Usuario: ')
    password = input('Contraseña: ')
    password_hash = hash_password(password)
    # Verificar credenciales en la base de datos
    conn = conectar()
    cursor = conn.cursor()
    cursor.execute('''
        SELECT id_usuario, usuario, rol FROM usuarios
        WHERE usuario = %s AND password = %s AND activo = TRUE
    ''', (usuario, password_hash))
    resultado = cursor.fetchone()
    if resultado is None:
        print('Usuario o contrasena incorrectos.')
        return None
    print(f'Bienvenido {resultado[1]} - Rol: {resultado[2]}')
    return {'id': resultado[0], 'usuario': resultado[1], 'rol': resultado[2]}
```

6.2 Hash de contraseñas

Las contrasenas nunca se guardan en texto plano. Se transforman con SHA-256, que es una función matemática que convierte cualquier texto en una cadena de 64 caracteres hexadecimales y no se puede revertir. Aunque alguien accediera a la base de datos no podría recuperar las contraseñas originales (Coronel & Morris, 2022). He elegido SHA-256 en lugar de MD5 porque MD5 lleva comprometido criptográficamente desde 2004.

```
def hash_password(password):
    # Convertir la contrasena a hash SHA-256
    return hashlib.sha256(password.encode()).hexdigest()
```

Además las contraseñas nuevas se validan antes de guardarse: mínimo 8 caracteres, al menos una mayúscula, una minúscula, un número y un símbolo.

6.3 Roles y permisos

He implementado control de acceso basado en roles (RBAC) con dos niveles: admin y empleado. El admin tiene acceso a todo. El empleado solo puede hacer las operaciones del día a día como ventas, reservas y consultas.

Funcion	Admin	Empleado
Alta, baja y renombrado de productos	Si	No

Registrar compra y compras anuales	Si	No
Registrar venta e historial de ventas	Si	Si
Ventas anuales	Si	No
Consultas de inventario	Si	Si
Crear reserva y convertir en venta	Si	Si
Cancelar reserva	Si	No
Gestion de usuarios y roles	Si	No

6.4 Auditoria

He creado un disparador en PostgreSQL que registra automaticamente cada cambio en la tabla inventario. Cada inserción, actualización o borrado genera un registro en la tabla log_operaciones con el tipo de operación, el producto afectado, las unidades antes y después, la fecha y el usuario de PostgreSQL que hizo el cambio. Esto permite reconstruir el historial completo del inventario en cualquier momento.

```
CREATE OR REPLACE TRIGGER tr_auditoria_inventario
AFTER INSERT OR UPDATE OR DELETE ON inventario
FOR EACH ROW
EXECUTE FUNCTION fn_auditoria_inventario();
```

7. Docker y Despliegue

7.1 Arquitectura de contenedores

He desplegado el sistema con Docker Compose, que coordina dos contenedores que trabajan juntos. La ventaja principal es que el entorno es idéntico en cualquier maquina: no importa el sistema operativo ni las versiones instaladas (PostgreSQL Global Development Group, 2024).

El contenedor heierling_db ejecuta PostgreSQL 16 y al arrancar ejecuta automaticamente todos los archivos SQL en orden numérico, creando las tablas, funciones, disparador, vista, índices y usuarios sin que haya que hacer nada manualmente. El contenedor heierling_app ejecuta Python 3.11 con psycopg2 instalado y se conecta al contenedor de base de datos a través de la red interna de Docker. Las credenciales se gestionan con un archivo .env que no se sube al repositorio.

7.2 Instrucciones de despliegue

Para ejecutar el sistema en cualquier maquina con Docker solo hacen falta cuatro comandos:

```
git clone https://github.com/jjpp01x/heierling.git
cd heierling
cp .env.example .env
docker compose up
```

Cada modulo Python se ejecuta desde una terminal separada con docker exec -it heierling_app python3 nombre_modulo.py.

7.3 Repositorio GitHub

Todo el codigo está en github.com/jjpp01x/heierling. El repositorio tiene el historial completo de commits con mensajes que explican cada cambio, un README.md con las instrucciones de uso, un .gitignore que excluye el archivo .env con las credenciales y un .env.example con los campos vacíos para que cualquiera que clone el proyecto sepa que valores necesita configurar.

8. Conclusiones

El sistema que he diseñado cubre todos los requisitos del trabajo y los amplia con funcionalidades que reflejan cómo funciona un negocio real. El modelo relacional normalizado hasta 3NF evita redundancias y garantiza la integridad de los datos siguiendo los principios de Codd (1970). Las diez funciones PL/pgSQL implementadas encapsulan la lógica dentro de la base de datos, lo que significa que las reglas se aplican siempre independientemente de cómo se llame al sistema.

Mas allá de lo que pedía la rúbrica he añadido cosas que considero importantes en cualquier sistema real: autenticación con contraseñas hasheadas y control de acceso por roles para proteger los datos, un disparador de auditoria para tener trazabilidad completa de los movimientos de inventario, doce índices para optimizar las consultas más frecuentes y despliegue con Docker para que el sistema funcione igual en cualquier entorno. Todo el codigo está disponible en GitHub con historial de cambios.

Como mejoras que se podrían hacer en el futuro veo tres principales: una interfaz gráfica web para que sea más fácil de usar, alertas automáticas por email cuando el stock baja de un mínimo, y exportación de los informes anuales a Excel o PDF para poder usarlos en herramientas de análisis.

9. Bibliografia

9075:2023, ISO/IEC. (n.d.). Information Technology - Database Languages - SQL. Internacional Organization for Standarization.

Codd, E. F. (1 June, 1970). *ACM Digital Library.* Retrieved from https://doi.org/10.1145/362384.362685

Coronel, C. &. (2022). *Database systems: Design, Implementation, and Management (14th ed.) .* Retrieved from Cengage Learning.

Development, Group PostgreSQL Global. (2024). *PostgreSQL 16 Documentation.*

Gehrke, J. R. ((2002)). Database Management Systems.

CESTE
Escuela Internacional
de Negocios