**Universidad Católica Boliviana “San Pablo”**

**“Caso de estudio 2”**



**Facultad:** Facultad de Ciencias Exactas

**Materia:** Base De Datos Il

**Docente:** Marin Salazar Carmen Rosa

**Integrantes:**

* Leandro Joe Jaldin Gutierrez
* Camila Catorceno Orellana

**Fecha: 18/06/2025**

**SPs – Gestión de Usuario**

**1. sp\_registrarNuevoCliente**

**Propósito**:  
Registra un nuevo cliente en la plataforma asegurando que no exista previamente una cuenta con el mismo correo.

**Parámetros**:

* @Correo: Correo electrónico del nuevo cliente.
* @Password: Contraseña cifrada del cliente.
* @Nombre: Nombre del cliente.
* @Apellido: Apellido del cliente.
* @Telefono: Número de teléfono del cliente.

**Lógica**:

1. Verifica si ya existe un usuario con ese correo y tipo "Cliente".
2. Si existe, lanza un error personalizado.
3. Si no existe, inserta el nuevo usuario en USERS.
4. Obtiene el UserID generado con SCOPE\_IDENTITY().
5. Inserta los datos personales en la tabla CLIENTES.

**Manejo de Errores**:  
Utiliza un bloque TRY-CATCH con transacción para garantizar la integridad. Si hay errores, se revierte la operación con ROLLBACK.

**Justificación Técnica**:

* El uso de SCOPE\_IDENTITY() permite capturar el ID del usuario recién creado.
* Se asegura la unicidad del correo y evita duplicidad de cuentas.
* La transacción garantiza atomicidad en la creación de usuario y cliente.

sql

CopiarEditar

--creacion del sp agragar clientes

create or alter procedure sp\_RegistrarNuevosClientes

@Correo NVARCHAR(100),

@Password NVARCHAR(200),

@Nombre NVARCHAR(50),

@Apellido NVARCHAR(50),

@Telefono INT

AS

BEGIN

SET NOCOUNT ON;

--1. VERIFICACION SI YA EXISTE ESE CORREO

IF EXISTS(SELECT 1 FROM USERS WHERE Correo = @Correo AND Tipo = 'Cliente')

begin

throw 50001, 'Ya tienes una cuenta registrada' , 1;

end

begin transaction;

--2.insertamos al nuevo cliente '

insert into USERS(Correo, Password, Tipo, Estado)

values(@Correo, @Password, 'Cliente', 'Activo')

commit transaction;

--confirmacion de exito

select 'Cliente registrado existosamente. ' as Mnesaje,

SCOPE\_IDENTITY () as ClienteID;

end try

begin catch

if XACT\_STATE() <> 0

rollback transaction;

declare @ErrorMessage nvarchar(4000) = Error\_Message();

declare @ErrorState int = ERROR\_STATE();

THROW 50000, @ErrorMessage, @ErrorState;

end catch

end;

go

**2. sp\_registrarNuevaTienda**

**Propósito**:  
Registra una nueva tienda en el sistema, verificando que no exista previamente con el mismo correo.

**Parámetros**:

* @Correo: Correo electrónico de la tienda.
* @Password: Contraseña de acceso.
* @Nombre: Nombre de la tienda.
* @NombreJuridico: Razón social o nombre legal.
* @NIT: Número de Identificación Tributaria.
* @Telefono: Número telefónico de contacto.
* @CategoriaID: ID de la categoría a la que pertenece la tienda.

**Lógica**:

1. Verifica si ya existe un usuario con ese correo y tipo "Tienda".
2. Si ya existe, lanza un error.
3. Inserta el nuevo usuario en USERS.
4. Usa SCOPE\_IDENTITY() para recuperar el UserID.
5. Inserta los datos de la tienda en la tabla TIENDAS.

**Manejo de Errores**:  
El bloque TRY-CATCH garantiza que la inserción de usuario y tienda sea atómica.

**Justificación Técnica**:

* Evita duplicidad de tiendas mediante validación previa.
* Se asegura que el registro sea completo o no se realice (transacción).
* Se mantiene integridad referencial entre USERS y TIENDAS.

CREATE OR ALTER PROCEDURE sp\_registrarNuevaTienda

@Correo NVARCHAR(100),

@Password NVARCHAR(200),

@Nombre NVARCHAR(50),

@NombreJuridico NVARCHAR(50),

@NIT INT,

@Telefono INT,

@CategoriaID INT

AS

BEGIN

SET NOCOUNT ON;

BEGIN TRY

-- Verificar si ya existe una tienda con ese correo

IF EXISTS (

SELECT 1 FROM USERS

WHERE Correo = @Correo AND Tipo = 'Tienda'

)

BEGIN

THROW 50002, 'Ya tienes una cuenta registrada.', 1;

END

BEGIN TRANSACTION;

-- Insertar en USERS

INSERT INTO USERS (Correo, Password, Tipo, Estado)

VALUES (@Correo, @Password, 'Tienda', 'Activo');

-- Obtener el nuevo UserID generado

DECLARE @UserID INT = SCOPE\_IDENTITY();

-- Insertar en TIENDAS

INSERT INTO TIENDAS (UserID, Nombre, NombreJuridico, NIT, Telefono, CategoriaID)

VALUES (@UserID, @Nombre, @NombreJuridico, @NIT, @Telefono, @CategoriaID);

COMMIT TRANSACTION;

SELECT 'Tienda registrada exitosamente.' AS Mensaje,

@UserID AS UserID;

END TRY

BEGIN CATCH

IF XACT\_STATE() <> 0

ROLLBACK TRANSACTION;

DECLARE @ErrorMessage NVARCHAR(4000) = ERROR\_MESSAGE();

DECLARE @ErrorState INT = ERROR\_STATE();

THROW 50000, @ErrorMessage, @ErrorState;

END CATCH

END;

GO

**3. sp\_inicioSesion**

**Propósito**:  
Verifica la existencia de un usuario con el correo proporcionado para validar el inicio de sesión.

**Parámetros**:

* @Correo: Correo electrónico del usuario.
* @Password: Contraseña del usuario (aunque en este SP no se verifica aún).

**Lógica**:

1. Verifica si existe un usuario con ese correo en USERS.
2. Si no existe, lanza un error indicando que no hay cuenta registrada.
3. Si existe, muestra un mensaje de éxito (no valida la contraseña todavía).

**Manejo de Errores**:  
Se emplea un bloque TRY-CATCH para encapsular la lógica y evitar fallos no controlados.

**Justificación Técnica**:

* Se valida la existencia previa del correo.
* Puede usarse como primer filtro antes de validar la contraseña real.

--sp iniciar seseion

select \* from USERS

create or alter procedure sp\_InicioSesion

@Correo nvarchar(200),

@Password int

as

begin

set nocount on;

begin try

if not exists(select 1 from USERS where Correo = @Correo and Password = @Password)

begin

throw 50001, 'Correo o Contraseña incorrectos ', 1;

end

if exists (select 1 from USERS where Correo = @Correo and Estado = 'Inactivo')

begin

throw 50002, 'La cuenta esta inactiva. ', 1;

end

--devolvemos los datos del usuario si todo esta bien

select UserID, Correo, Tipo, Estado from USERS

where Correo = @Correo and Password = @Password

end try

begin catch

DECLARE @ErrorMessage NVARCHAR(4000) = ERROR\_MESSAGE();

DECLARE @ErrorState INT = ERROR\_STATE();

THROW 50000, @ErrorMessage, @ErrorState;

END CATCH

END;

GO

**4. sp\_perfilUsuario**

**Propósito**:  
Muestra el perfil de un usuario, ya sea cliente o tienda, según su tipo.

**Parámetros**:

* @UserID: ID del usuario cuyo perfil se desea ver.

**Lógica**:

1. Verifica el tipo de usuario (Cliente o Tienda) consultando la tabla USERS.
2. Si es cliente, selecciona sus datos desde CLIENTES.
3. Si es tienda, selecciona sus datos desde TIENDAS.

**Justificación Técnica**:

* Permite obtener información diferenciada según el rol del usuario.
* Elimina la necesidad de múltiples consultas externas.

create or alter procedure sp\_PerfilUsuario

@Correo nvarchar(200)

as

begin

set nocount on;

begin try

if not exists(select 1 from USERS where Correo = @Correo)

begin

throw 50001, 'El usuario no existe', 1;

end

select Correo, Tipo, Estado from USERS

where Correo = @Correo;

end try

begin catch

if XACT\_STATE() <> 0

rollback transaction;

declare @ERRORMESSAGE NVARCHAR(4000) = ERROR\_MESSAGE();

DECLARE @ERRORSTATE INT = ERROR\_STATE();

THROW 50000, @ERRORMESSAGE , @ERRORSTATE;

end catch

end;

go

**SP – Gestión de Inventario**

**1. sp\_aplicarDescuentoAProducto**

**Propósito**:  
Asocia un descuento existente a un producto específico del inventario.

**Parámetros**:

* @ProductoID: ID del descuento a aplicar.
* @PrecioDescuentoID: ID del producto que recibirá el descuento.

**Lógica**:

1. Actualiza el campo PrecioDescuento del producto en la tabla INVENTARIO.

**Manejo de Errores**:  
Uso de TRY-CATCH con transacción para asegurar que la operación se realice correctamente.

**Justificación Técnica**:

* Permite modificar descuentos sin alterar el resto del producto.
* Se asegura la consistencia de datos al trabajar con IDs relacionados.

CREATE OR ALTER PROCEDURE sp\_AplicarDescuentoProducto

@ProductoID INT,

@PorcentajeDescuento DECIMAL(5,2)

AS

BEGIN

SET NOCOUNT ON;

BEGIN TRY

-- Validar que el producto exista

IF NOT EXISTS (SELECT 1 FROM INVENTARIO WHERE ProductoID = @ProductoID)

BEGIN

THROW 50001, 'El producto especificado no existe.', 1;

END

-- Validar que el porcentaje esté en rango válido (ej: 0 < % < 100)

IF @PorcentajeDescuento <= 0 OR @PorcentajeDescuento >= 100

BEGIN

THROW 50002, 'El porcentaje de descuento debe estar entre 0 y 100.', 1;

END

BEGIN TRANSACTION;

-- Aplicar descuento y calcular precio con descuento

UPDATE INVENTARIO

SET PrecioDescuento = Precio - (Precio \* @PorcentajeDescuento / 100.0),

ModifiedDate = GETDATE()

WHERE ProductoID = @ProductoID;

COMMIT TRANSACTION;

SELECT 'Descuento aplicado correctamente.' AS Mensaje;

END TRY

BEGIN CATCH

IF XACT\_STATE() <> 0

ROLLBACK TRANSACTION;

DECLARE @ErrorMessage NVARCHAR(4000) = ERROR\_MESSAGE();

DECLARE @ErrorState INT = ERROR\_STATE();

THROW 50000, @ErrorMessage, @ErrorState;

END CATCH

END;

GO

**2. sp\_registrarProducto \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Propósito**:  
Registra un nuevo producto en el inventario de una tienda.

**Parámetros**:

* @TiendaID: ID de la tienda propietaria del producto.
* @SubcategoriaID: Subcategoría del producto.
* @DescuentoID: Descuento producto opcional.
* @Nombre: Nombre del producto.
* @Precio: Precio del producto.
* @Descripcion: Breve descripción del producto.
* @Stock: numero de productos.

**Lógica**:

1. Inserta un nuevo registro en INVENTARIO con los datos básicos del producto.

**Nota**:  
El parámetro @Stock está comentado, se puede añadir si se quiere controlar la cantidad desde el inicio.

**Justificación Técnica**:

* Se omite el stock para permitir registro básico y posterior actualización.
* Se puede extender para incluir estado inicial (ej. 'Disponible').

create or alter procedure sp\_RegistrarNuevoProducto

@TiendaID int,

@SubcategoriaID int ,

@DescuentoID int = null,

@Nombre nvarchar(100),

@Precio decimal(10, 2),

@Descripcion nvarchar(250),

@Stock int

as

begin

set nocount on;

begin try

--validaciones del precio

if @Precio <= 0

throw 50001, 'El precio debe ser mayor a cero ', 1;

--validar stock

if @Stock < 0

throw 50002, 'El Stock debe ser mayor a cero ', 1;

begin transaction;

--insertamos al producto

insert into INVENTARIO(TiendaID, SubcategoriaID, DescuentoID, Nombre, Precio, Descripcion, Stock, Estado, ModifiedDate)

values (@TiendaID, @SubcategoriaID, @DescuentoID, @Nombre, @Precio, @Descripcion,@Stock,

case when @Stock < 0 then 'No disponible' else 'Disponible' end, GETDATE());

commit transaction;

select 'Producto Registrado exitosamente ' as Mensaje , SCOPE\_IDENTITY() as ProductoID;

end try

begin catch

if XACT\_STATE() <> 0

rollback transaction;

Declare @ErrorMessage nvarchar(4000) = error\_message();

declare @ErrorState int = error\_state();

throw 50000, @ErrorMessage , @ErrorState;

end catch

end;

go

**3. sp\_editarProducto**

**Propósito**:  
Permite modificar los datos de un producto existente en el inventario.

**Parámetros**:

* @ProductoID: ID del producto a editar.
* @SubcategoriaID: Nueva subcategoría (si se desea cambiar).
* @Nombre: Nuevo nombre del producto.
* @Precio: Nuevo precio.
* @Descripcion: Nueva descripción.
* @DescuentoID: Nuevo descuento asociado (puede ser NULL si no se desea aplicar).

**Lógica**:

1. Actualiza los campos del producto en la tabla INVENTARIO según los datos nuevos.

**Justificación Técnica**:

* Permite mantener la información actualizada sin eliminar y volver a insertar productos.

CREATE OR ALTER PROCEDURE sp\_EditarProducto

@ProductoID INT,

@SubcategoriaID INT,

@Nombre NVARCHAR(100),

@Precio DECIMAL(10,2),

@Descripcion NVARCHAR(200),

@DescuentoID INT = NULL

AS

BEGIN

SET NOCOUNT ON;

BEGIN TRY

-- Validar existencia del producto

IF NOT EXISTS (SELECT 1 FROM INVENTARIO WHERE ProductoID = @ProductoID)

BEGIN

THROW 50001, 'El producto especificado no existe.', 1;

END

-- Validar precio positivo

IF @Precio <= 0

BEGIN

THROW 50002, 'El precio debe ser mayor a cero.', 1;

END

BEGIN TRANSACTION;

-- Actualizar los datos del producto

UPDATE INVENTARIO

SET

SubcategoriaID = @SubcategoriaID,

Nombre = @Nombre,

Precio = @Precio,

Descripcion = @Descripcion,

DescuentoID = @DescuentoID,

ModifiedDate = GETDATE()

WHERE ProductoID = @ProductoID;

COMMIT TRANSACTION;

SELECT 'Producto actualizado exitosamente.' AS Mensaje;

END TRY

BEGIN CATCH

IF XACT\_STATE() <> 0

ROLLBACK TRANSACTION;

DECLARE @ErrorMessage NVARCHAR(4000) = ERROR\_MESSAGE();

DECLARE @ErrorState INT = ERROR\_STATE();

THROW 50000, @ErrorMessage, @ErrorState;

END CATCH

END;

GO

**4. sp\_eliminarProducto**

**Propósito**:  
“Eliminar” un producto marcándolo como Descontinuado (sin borrarlo físicamente).

**Parámetros**:

* @TiendaID: ID de la tienda que posee el producto.
* @ProductoID: ID del producto a marcar como eliminado.

**Lógica**:

1. Verifica si el producto está en estado 'Disponible'.
2. Si lo está, lo actualiza a 'Descontinuado'.

**Justificación Técnica**:

* Se evita el borrado físico de registros, conservando historial de productos.
* Se facilita la trazabilidad y la auditoría.

**Nota**: Asegúrate que la tabla INVENTARIO tiene una columna Estado.

CREATE OR ALTER PROCEDURE sp\_EliminarProducto

@TiendaID INT,

@ProductoID INT

AS

BEGIN

SET NOCOUNT ON;

BEGIN TRY

-- Validar que el producto exista y pertenezca a la tienda

IF NOT EXISTS (

SELECT 1 FROM INVENTARIO

WHERE ProductoID = @ProductoID AND TiendaID = @TiendaID

)

BEGIN

THROW 50001, 'El producto no existe o no pertenece a esta tienda.', 1;

END

-- Validar que el producto esté disponible

IF NOT EXISTS (

SELECT 1 FROM INVENTARIO

WHERE ProductoID = @ProductoID AND Estado = 'Disponible'

)

BEGIN

THROW 50002, 'El producto no está disponible o ya fue descontinuado.', 1;

END

BEGIN TRANSACTION;

-- Cambiar el estado a descontinuado

UPDATE INVENTARIO

SET Estado = 'Descontinuado',

ModifiedDate = GETDATE()

WHERE ProductoID = @ProductoID;

COMMIT TRANSACTION;

SELECT 'Producto descontinuado exitosamente.' AS Mensaje;

END TRY

BEGIN CATCH

IF XACT\_STATE() <> 0

ROLLBACK TRANSACTION;

DECLARE @ErrorMessage NVARCHAR(4000) = ERROR\_MESSAGE();

DECLARE @ErrorState INT = ERROR\_STATE();

THROW 50000, @ErrorMessage, @ErrorState;

END CATCH

END;

GO

**5. sp\_verInventario**

**Propósito**:  
Lista el inventario de una tienda, mostrando productos con su información y descuento aplicado.

**Parámetros**:

* @TiendaID: ID de la tienda cuyo inventario se quiere visualizar.

**Lógica**:

1. Realiza un JOIN entre INVENTARIO, SUBCATEGORIAS, CATEGORIAS y DESCUENTOS.
2. Filtra por TiendaID.
3. Muestra campos como: nombre del producto, categoría, subcategoría, precio, nombre del descuento y porcentaje aplicado.

**Justificación Técnica**:

* Ofrece una vista detallada del inventario con descuentos aplicados.
* El LEFT JOIN permite mostrar productos sin descuentos (muestra NULL en esos campos).

CREATE OR ALTER PROCEDURE sp\_VerInventario

@TiendaID INT

AS

BEGIN

SET NOCOUNT ON;

BEGIN TRY

-- Validar que la tienda exista

IF NOT EXISTS (SELECT 1 FROM TIENDAS WHERE TiendaID = @TiendaID)

BEGIN

THROW 50001, 'La tienda especificada no existe.', 1;

END

-- Seleccionar el inventario con joins

SELECT

i.ProductoID,

i.Nombre AS Producto,

c.Nombre AS Categoria,

s.Nombre AS Subcategoria,

i.Stock,

i.Precio,

d.Nombre AS NombreDescuento,

d.Porcentaje AS PorcentajeDescuento,

i.PrecioDescuento,

i.Estado,

i.ModifiedDate

FROM INVENTARIO i

INNER JOIN SUBCATEGORIAS s ON i.SubcategoriaID = s.SubcategoriaID

INNER JOIN CATEGORIAS c ON s.CategoriaID = c.CategoriaID

LEFT JOIN DESCUENTOS d ON i.DescuentoID = d.DescuentoID

WHERE i.TiendaID = @TiendaID

ORDER BY i.Nombre ASC;

END TRY

BEGIN CATCH

DECLARE @ErrorMessage NVARCHAR(4000) = ERROR\_MESSAGE();

DECLARE @ErrorState INT = ERROR\_STATE();

THROW 50000, @ErrorMessage, @ErrorState;

END CATCH

END;

GO

**Sp – Registrar una venta o proceso de venta**

**Proposito:**

create or alter procedure sp\_RegistrarVenta

@CompraID int,

@ProductoID int,

@Cantidad int

as

begin

set nocount on;

begin try

if not exists(select 1 from INVENTARIO where ProductoID = @ProductoID) --verifica si el produc exite en la tabla inventario

begin

throw 50001, 'El Producto no existe', 1;

end

--verificacion del stock

declare @StockDisponible int;

select @StockDisponible = Stock from INVENTARIO where ProductoID = @ProductoID; --consultamos el stock disponible

if @StockDisponible < @Cantidad --verifica si tenemos sotck suficiente para vender

throw 50002, 'No tenemos stock sufuciente', 1;

--obtencion del precio unitario

declare @PrecioUnitario decimal(10,2);

select @PrecioUnitario = Precio from INVENTARIO where ProductoID = @ProductoID; --consulta el precio del producto

---transation

begin transaction;

--insertamos las ventas

insert into VENTAS(CompraID, ProductoID, Cantidad, PrecioUnitario, OrderDate, ShipDate

, ModifiedDate)

values (@CompraID, @ProductoID, @Cantidad, @PrecioUnitario,GETDATE(), GETDATE(), GETDATE()); --inserta en tabla ventas

--actualizamos las tablas

update INVENTARIO --actualiza el inventaro

set Stock = Stock - @Cantidad --resta la cantidad vendida al stock del producto

where ProductoID = @ProductoID;

if exists (select 1 from INVENTARIO where ProductoID = @ProductoID and Stock = 0)

begin

update INVENTARIO --si el stock en ese producto llega a cero lo marca como no disponible

set Estado = 'No disponible'

where ProductoID = @ProductoID;

end

commit transaction;

select 'Venta registrada exitosamente' as mensaje , SCOPE\_IDENTITY() as VentaID;

END TRY

BEGIN CATCH

IF XACT\_STATE() <> 0

ROLLBACK TRANSACTION;

DECLARE @ERRORMESS NVARCHAR(4000) = ERROR\_MESSAGE();

DECLARE @ERRORST INT = ERROR\_STATE();

THROW 50000, @ERRORMESS, @ERRORST;

END CATCH

END;

GO

**Sp – Gestion de Pagos**

**1. usp\_RegistrarDireccion**

**Propósito:**  
Registra una nueva dirección activa para un cliente, si no existe una idéntica registrada.

**Parámetros:**

* @ClienteID: ID del cliente.
* @Barrio: Nombre del barrio.
* @Calle: Nombre de la calle.
* @Numero: Número del domicilio.

**Lógica:**

* Valida que el cliente exista.
* Verifica que los campos no estén vacíos y que el número sea positivo.
* Comprueba que no haya una dirección activa igual.
* Inserta una nueva dirección con estado 'Activo'.

**Justificación Técnica:**  
Evita duplicación de direcciones activas y garantiza consistencia de datos mediante validaciones y transacciones.

CREATE OR ALTER PROCEDURE usp\_RegistrarDireccion

@ClienteID INT,

@Barrio NVARCHAR(50),

@Calle NVARCHAR(50),

@Numero INT

AS

BEGIN

SET NOCOUNT ON;

BEGIN TRY

-- Validaciones...

BEGIN TRANSACTION;

INSERT INTO DIRECCIONES (ClienteID, Barrio, Calle, Numero, Estado)

VALUES (@ClienteID, @Barrio, @Calle, @Numero, 'Activo');

COMMIT TRANSACTION;

SELECT 'Dirección registrada exitosamente.' AS Mensaje,

SCOPE\_IDENTITY() AS DireccionID;

END TRY

BEGIN CATCH

IF XACT\_STATE() <> 0

ROLLBACK TRANSACTION;

THROW 50000, ERROR\_MESSAGE(), ERROR\_STATE();

END CATCH

END;

**2. usp\_EditarDireccion**

**Propósito:**  
Permite modificar una dirección activa del cliente, evitando duplicados.

**Parámetros:**

* @DireccionID: ID de la dirección a modificar.
* @Barrio: Nuevo nombre del barrio.
* @Calle: Nuevo nombre de la calle.
* @Numero: Nuevo número del domicilio.

**Lógica:**

* Verifica existencia de la dirección.
* Valida que los campos sean válidos.
* Evita duplicados con otra dirección activa del mismo cliente.
* Actualiza los datos de la dirección.

**Justificación Técnica:**  
Permite edición sin romper integridad ni generar datos duplicados. Uso de transacciones asegura atomicidad.

CREATE OR ALTER PROCEDURE usp\_EditarDireccion

@DireccionID INT,

@Barrio NVARCHAR(50),

@Calle NVARCHAR(50),

@Numero INT

AS

BEGIN

SET NOCOUNT ON;

BEGIN TRY

-- Validaciones...

BEGIN TRANSACTION;

UPDATE DIRECCIONES

SET Barrio = @Barrio,

Calle = @Calle,

Numero = @Numero,

Estado = 'Activo',

ModifiedDate = GETDATE()

WHERE DireccionID = @DireccionID;

COMMIT TRANSACTION;

SELECT 'Dirección actualizada exitosamente.' AS Mensaje,

@DireccionID AS DireccionID;

END TRY

BEGIN CATCH

IF XACT\_STATE() <> 0

ROLLBACK TRANSACTION;

THROW 50000, ERROR\_MESSAGE(), ERROR\_STATE();

END CATCH

END;

**3. usp\_EliminarDireccion**

**Propósito:**  
Desactiva una dirección activa del cliente.

**Parámetros:**

* @DireccionID: ID de la dirección a eliminar.

**Lógica:**

* Verifica que la dirección exista y esté activa.
* Marca la dirección como 'Inactivo' y actualiza la fecha.

**Justificación Técnica:**  
Se conserva el historial de direcciones al inactivar en lugar de eliminar físicamente. Mantiene integridad de referencias con compras previas.

CREATE OR ALTER PROCEDURE usp\_EliminarDireccion

@DireccionID INT

AS

BEGIN

SET NOCOUNT ON;

BEGIN TRY

-- Validaciones...

BEGIN TRANSACTION;

UPDATE DIRECCIONES

SET Estado = 'Inactivo',

ModifiedDate = GETDATE()

WHERE DireccionID = @DireccionID;

COMMIT TRANSACTION;

SELECT 'Dirección marcada como inactiva exitosamente.' AS Mensaje;

END TRY

BEGIN CATCH

IF XACT\_STATE() <> 0

ROLLBACK TRANSACTION;

THROW 50000, ERROR\_MESSAGE(), ERROR\_STATE();

END CATCH

END;

**4. usp\_VisualizarDirecciones**

**Propósito:**  
Lista las direcciones activas registradas por un cliente.

**Parámetros:**

* @ClienteID: ID del cliente.

**Lógica:**

* Verifica que el cliente exista.
* Devuelve las direcciones activas en orden de registro.

**Justificación Técnica:**  
Facilita la gestión visual de direcciones para ediciones o eliminaciones futuras. Asegura retorno solo de datos relevantes (activos).

CREATE OR ALTER PROCEDURE usp\_VisualizarDirecciones

@ClienteID INT

AS

BEGIN

SET NOCOUNT ON;

BEGIN TRY

-- Validaciones...

SELECT DireccionID, Barrio, Calle, Numero, ModifiedDate

FROM DIRECCIONES

WHERE ClienteID = @ClienteID AND Estado = 'Activo'

ORDER BY DireccionID;

IF @@ROWCOUNT = 0

BEGIN

SELECT 'No se encontraron direcciones activas para el cliente.' AS Mensaje;

END

END TRY

BEGIN CATCH

THROW 50000, ERROR\_MESSAGE(), ERROR\_STATE();

END CATCH

END;

**Tabla Tarjetas**

**1. usp\_VisualizarTarjetas**

**Propósito:**  
Lista las tarjetas activas registradas por un cliente.

**Parámetros:**  
• @ClienteID: ID del cliente.

**Lógica:**  
• Verifica que el cliente exista.  
• Devuelve las tarjetas activas mostrando solo los últimos 4 dígitos.  
• Ordena por ID de tarjeta.

**Justificación Técnica:**  
Permite al cliente visualizar de forma segura sus tarjetas registradas, mostrando únicamente la información esencial (últimos dígitos). Facilita la gestión sin comprometer datos sensibles.

CREATE OR ALTER PROCEDURE usp\_VisualizarTarjetas

@ClienteID INT

AS

BEGIN

SET NOCOUNT ON;

BEGIN TRY

-- Validar existencia del cliente

IF NOT EXISTS (SELECT 1 FROM CLIENTES WHERE ClienteID = @ClienteID)

BEGIN

THROW 50001, 'El ClienteID especificado no existe.', 1;

END

-- Mostrar tarjetas activas

SELECT TarjetaID, Red, NombreTitular, RIGHT(CAST(Numero AS NVARCHAR(16)), 4) AS UltimosDigitos,

CVC, ExpDate, ModifiedDate

FROM TARJETAS

WHERE ClienteID = @ClienteID AND Estado = 'Activo'

ORDER BY TarjetaID;

-- Mensaje informativo si no hay resultados

IF @@ROWCOUNT = 0

BEGIN

SELECT 'No se encontraron tarjetas activas para el cliente.' AS Mensaje;

END

END TRY

BEGIN CATCH

THROW 50000, ERROR\_MESSAGE(), ERROR\_STATE();

END CATCH

END;

GO

**2. usp\_RegistrarTarjeta**

**Propósito:**  
Registra una nueva tarjeta activa para un cliente.

**Parámetros:**  
• @ClienteID: ID del cliente.  
• @Red: Red de la tarjeta (Visa o MasterCard).  
• @NombreTitular: Nombre del titular.  
• @Numero: Número de tarjeta (16 dígitos).  
• @CVC: Código de seguridad (3 dígitos).  
• @ExpDate: Fecha de expiración futura.

**Lógica:**  
• Verifica que el cliente exista.  
• Valida campos obligatorios, formato de número, CVC y fecha.  
• Verifica que la tarjeta no esté registrada como activa.  
• Inserta la nueva tarjeta.

**Justificación Técnica:**  
Evita registros duplicados de tarjetas y garantiza que los datos sean válidos y actuales antes de insertarlos. Mejora la seguridad y fiabilidad del sistema.

CREATE OR ALTER PROCEDURE usp\_RegistrarTarjeta

@ClienteID INT,

@Red NVARCHAR(20),

@NombreTitular NVARCHAR(50),

@Numero BIGINT,

@CVC INT,

@ExpDate DATE

AS

BEGIN

SET NOCOUNT ON;

BEGIN TRY

-- Validaciones...

IF NOT EXISTS (SELECT 1 FROM CLIENTES WHERE ClienteID = @ClienteID)

THROW 50001, 'El ClienteID especificado no existe.', 1;

IF @Red IS NULL OR @Red = '' OR @NombreTitular IS NULL OR @NombreTitular = ''

THROW 50002, 'Red y NombreTitular no pueden estar vacíos.', 1;

IF @Red NOT IN ('Visa', 'MasterCard')

THROW 50003, 'La red debe ser Visa o MasterCard.', 1;

IF @Numero < 1000000000000000 OR @Numero > 9999999999999999

THROW 50004, 'El número de tarjeta debe tener 16 dígitos.', 1;

IF @CVC < 100 OR @CVC > 999

THROW 50005, 'El CVC debe tener 3 dígitos.', 1;

IF @ExpDate <= CAST(GETDATE() AS DATE)

THROW 50006, 'La fecha de expiración debe ser futura.', 1;

BEGIN TRANSACTION;

IF EXISTS (

SELECT 1 FROM TARJETAS

WHERE NombreTitular = @NombreTitular

AND Numero = @Numero

AND CVC = @CVC

AND Estado = 'Activo'

)

THROW 50007, 'Esta tarjeta ya está registrada y activa.', 1;

INSERT INTO TARJETAS (ClienteID, Red, NombreTitular, Numero, CVC, ExpDate, Estado)

VALUES (@ClienteID, @Red, @NombreTitular, @Numero, @CVC, @ExpDate, 'Activo');

COMMIT TRANSACTION;

SELECT 'Tarjeta registrada exitosamente.' AS Mensaje,

SCOPE\_IDENTITY() AS TarjetaID;

END TRY

BEGIN CATCH

IF XACT\_STATE() <> 0 ROLLBACK TRANSACTION;

THROW 50000, ERROR\_MESSAGE(), ERROR\_STATE();

END CATCH

END;

GO

**3. usp\_EditarTarjeta**

**Propósito:**  
Modifica los datos de una tarjeta activa ya registrada.

**Parámetros:**  
• @TarjetaID: ID de la tarjeta.  
• @Red, @NombreTitular, @Numero, @CVC, @ExpDate: Datos a actualizar.

**Lógica:**  
• Verifica existencia de la tarjeta y que esté activa.  
• Valida todos los campos.  
• Evita duplicados con otras tarjetas activas.  
• Realiza la actualización.

**Justificación Técnica:**  
Permite modificar datos de tarjetas sin permitir inconsistencias ni duplicados. Mejora integridad de datos y flexibilidad para el usuario.

CREATE OR ALTER PROCEDURE usp\_EditarTarjeta

@TarjetaID INT,

@Red NVARCHAR(20),

@NombreTitular NVARCHAR(50),

@Numero BIGINT,

@CVC INT,

@ExpDate DATE

AS

BEGIN

SET NOCOUNT ON;

BEGIN TRY

-- Validaciones...

IF NOT EXISTS (SELECT 1 FROM TARJETAS WHERE TarjetaID = @TarjetaID)

THROW 50001, 'La tarjeta especificada no existe.', 1;

IF EXISTS (SELECT 1 FROM TARJETAS WHERE TarjetaID = @TarjetaID AND Estado = 'Inactivo')

THROW 50002, 'No se puede editar una tarjeta inactiva.', 1;

IF @Red IS NULL OR @Red = '' OR @NombreTitular IS NULL OR @NombreTitular = ''

THROW 50003, 'Red y NombreTitular no pueden estar vacíos.', 1;

IF @Red NOT IN ('Visa', 'MasterCard')

THROW 50004, 'La red debe ser Visa o MasterCard.', 1;

IF @Numero < 1000000000000000 OR @Numero > 9999999999999999

THROW 50005, 'El número de tarjeta debe tener 16 dígitos.', 1;

IF @CVC < 100 OR @CVC > 999

THROW 50006, 'El CVC debe tener 3 dígitos.', 1;

IF @ExpDate <= CAST(GETDATE() AS DATE)

THROW 50007, 'La fecha de expiración debe ser futura.', 1;

BEGIN TRANSACTION;

IF EXISTS (

SELECT 1 FROM TARJETAS

WHERE NombreTitular = @NombreTitular

AND Numero = @Numero

AND CVC = @CVC

AND TarjetaID != @TarjetaID

AND Estado = 'Activo'

)

THROW 50008, 'Ya existe otra tarjeta activa idéntica.', 1;

UPDATE TARJETAS

SET Red = @Red,

NombreTitular = @NombreTitular,

Numero = @Numero,

CVC = @CVC,

ExpDate = @ExpDate,

Estado = 'Activo',

ModifiedDate = GETDATE()

WHERE TarjetaID = @TarjetaID;

COMMIT TRANSACTION;

SELECT 'Tarjeta actualizada exitosamente.' AS Mensaje,

@TarjetaID AS TarjetaID;

END TRY

BEGIN CATCH

IF XACT\_STATE() <> 0 ROLLBACK TRANSACTION;

THROW 50000, ERROR\_MESSAGE(), ERROR\_STATE();

END CATCH

END;

GO

**4. usp\_EliminarTarjeta**

**Propósito:**  
Inactiva una tarjeta registrada, sin eliminarla físicamente.

**Parámetros:**  
• @TarjetaID: ID de la tarjeta a eliminar.

**Lógica:**  
• Verifica que la tarjeta exista y esté activa.  
• Actualiza su estado a 'Inactivo'.

**Justificación Técnica:**  
Evita la pérdida de trazabilidad histórica. Desactiva la tarjeta en lugar de eliminarla, protegiendo la integridad referencial.

CREATE OR ALTER PROCEDURE usp\_EliminarTarjeta

@TarjetaID INT

AS

BEGIN

SET NOCOUNT ON;

BEGIN TRY

-- Validaciones...

IF NOT EXISTS (SELECT 1 FROM TARJETAS WHERE TarjetaID = @TarjetaID)

THROW 50001, 'La tarjeta especificada no existe.', 1;

IF EXISTS (SELECT 1 FROM TARJETAS WHERE TarjetaID = @TarjetaID AND Estado = 'Inactivo')

THROW 50002, 'La tarjeta ya está inactiva.', 1;

BEGIN TRANSACTION;

UPDATE TARJETAS

SET Estado = 'Inactivo',

ModifiedDate = GETDATE()

WHERE TarjetaID = @TarjetaID;

COMMIT TRANSACTION;

SELECT 'Tarjeta marcada como inactiva exitosamente.' AS Mensaje;

END TRY

BEGIN CATCH

IF XACT\_STATE() <> 0 ROLLBACK TRANSACTION;

THROW 50000, ERROR\_MESSAGE(), ERROR\_STATE();

END CATCH

END;

GO

**Tabla Facturas**

**1. usp\_VisualizarFacturas**

**Propósito:**  
Lista las facturas activas registradas para un cliente.

**Parámetros:**  
• @ClienteID: ID del cliente.

**Lógica:**  
• Verifica que el cliente exista.  
• Devuelve las facturas activas ordenadas por ID.

**Justificación Técnica:**  
Permite gestionar y visualizar las facturas activas asociadas a un cliente sin mostrar datos

CREATE OR ALTER PROCEDURE usp\_VisualizarFacturas

@ClienteID INT

AS

BEGIN

SET NOCOUNT ON;

BEGIN TRY

-- Validar que ClienteID existe

IF NOT EXISTS (SELECT 1 FROM CLIENTES WHERE ClienteID = @ClienteID)

BEGIN

THROW 50001, 'El ClienteID especificado no existe.', 1;

END

-- Mostrar facturas activas

SELECT FacturaID, RazonSocial, NitCi, ModifiedDate

FROM DATOS\_FACTURA

WHERE ClienteID = @ClienteID AND Estado = 'Activo'

ORDER BY FacturaID;

-- Devolver mensaje si no hay facturas activas

IF @@ROWCOUNT = 0

BEGIN

SELECT 'No se encontraron facturas activas para el cliente.' AS Mensaje;

END

END TRY

BEGIN CATCH

DECLARE @ErrorMessage NVARCHAR(4000) = ERROR\_MESSAGE();

DECLARE @ErrorSeverity INT = ERROR\_SEVERITY();

DECLARE @ErrorState INT = ERROR\_STATE();

THROW 50000, @ErrorMessage, @ErrorState;

END CATCH

END;

GO

**2. usp\_EliminarFactura**

**Propósito:**  
Inactiva una factura registrada sin eliminarla físicamente.

**Parámetros:**  
• @FacturaID: ID de la factura a eliminar.

**Lógica:**  
• Verifica que la factura exista y esté activa.  
• Cambia su estado a 'Inactivo'.

**Justificación Técnica:**  
Preserva la integridad histórica y referencial, evitando la pérdida de datos al desactivar la factura en lugar de eliminarla.

CREATE OR ALTER PROCEDURE usp\_EliminarFactura

@FacturaID INT

AS

BEGIN

SET NOCOUNT ON;

BEGIN TRY

-- Validar que FacturaID existe

IF NOT EXISTS (SELECT 1 FROM DATOS\_FACTURA WHERE FacturaID = @FacturaID)

BEGIN

THROW 50001, 'La factura especificada no existe.', 1;

END

-- Validar que la factura no esté ya inactiva

IF EXISTS (SELECT 1 FROM DATOS\_FACTURA WHERE FacturaID = @FacturaID AND Estado = 'Inactivo')

BEGIN

THROW 50002, 'La factura ya está inactiva.', 1;

END

BEGIN TRANSACTION;

-- Marcar como inactiva

UPDATE DATOS\_FACTURA

SET Estado = 'Inactivo',

ModifiedDate = GETDATE()

WHERE FacturaID = @FacturaID;

COMMIT TRANSACTION;

-- Devolver mensaje de éxito

SELECT 'Factura marcada como inactiva exitosamente.' AS Mensaje;

END TRY

BEGIN CATCH

IF XACT\_STATE() <> 0

ROLLBACK TRANSACTION;

DECLARE @ErrorMessage NVARCHAR(4000) = ERROR\_MESSAGE();

DECLARE @ErrorSeverity INT = ERROR\_SEVERITY();

DECLARE @ErrorState INT = ERROR\_STATE();

THROW 50000, @ErrorMessage, @ErrorState;

END CATCH

END;

GO

**3. usp\_RegistrarFactura**

**Propósito:**  
Registra una nueva factura para un cliente.

**Parámetros:**  
• @ClienteID: ID del cliente.  
• @RazonSocial: Razón social para la factura.  
• @NitCi: Número de identificación tributaria o cédula.

**Lógica:**  
• Verifica que el cliente exista.  
• Valida que RazonSocial y NitCi no estén vacíos.  
• Inserta la nueva factura con estado 'Activo'.

**Justificación Técnica:**  
Permite agregar facturas con datos completos y válidos, manteniendo un registro activo para el cliente.

CREATE OR ALTER PROCEDURE usp\_RegistrarFactura

@ClienteID INT,

@RazonSocial NVARCHAR(100),

@NitCi NVARCHAR(20)

AS

BEGIN

SET NOCOUNT ON;

BEGIN TRY

-- Validar que ClienteID existe

IF NOT EXISTS (SELECT 1 FROM CLIENTES WHERE ClienteID = @ClienteID)

BEGIN

THROW 50001, 'El ClienteID especificado no existe.', 1;

END

-- Validar parámetros no vacíos

IF @RazonSocial IS NULL OR @RazonSocial = '' OR @NitCi IS NULL OR @NitCi = ''

BEGIN

THROW 50002, 'RazonSocial y NitCi no pueden estar vacíos.', 1;

END

BEGIN TRANSACTION;

-- Insertar la factura

INSERT INTO DATOS\_FACTURA (ClienteID, RazonSocial, NitCi, Estado)

VALUES (@ClienteID, @RazonSocial, @NitCi, 'Activo');

COMMIT TRANSACTION;

-- Devolver mensaje de éxito

SELECT 'Factura registrada exitosamente.' AS Mensaje,

SCOPE\_IDENTITY() AS FacturaID;

END TRY

BEGIN CATCH

IF XACT\_STATE() <> 0

ROLLBACK TRANSACTION;

DECLARE @ErrorMessage NVARCHAR(4000) = ERROR\_MESSAGE();

DECLARE @ErrorSeverity INT = ERROR\_SEVERITY();

DECLARE @ErrorState INT = ERROR\_STATE();

THROW 50000, @ErrorMessage, @ErrorState;

END CATCH

END;

GO

**4. usp\_EditarFactura**

**Propósito:**  
Edita los datos de una factura activa.

**Parámetros:**  
• @FacturaID: ID de la factura a editar.  
• @ClienteID: ID del cliente.  
• @RazonSocial: Nueva razón social.  
• @NitCi: Nuevo NIT o CI.

**Lógica:**  
• Verifica que la factura exista y esté activa.  
• Verifica que el cliente exista.  
• Valida que RazonSocial y NitCi no estén vacíos.  
• Actualiza los datos y la fecha de modificación.

**Justificación Técnica:**  
Permite mantener los datos actualizados y correctos sin perder la trazabilidad histórica.

CREATE OR ALTER PROCEDURE usp\_EditarFactura

@FacturaID INT,

@ClienteID INT,

@RazonSocial NVARCHAR(100),

@NitCi NVARCHAR(20)

AS

BEGIN

SET NOCOUNT ON;

BEGIN TRY

-- Validar que FacturaID existe

IF NOT EXISTS (SELECT 1 FROM DATOS\_FACTURA WHERE FacturaID = @FacturaID)

BEGIN

THROW 50001, 'La factura especificada no existe.', 1;

END

-- Validar que la factura esté activa

IF EXISTS (SELECT 1 FROM DATOS\_FACTURA WHERE FacturaID = @FacturaID AND Estado = 'Inactivo')

BEGIN

THROW 50002, 'No se puede editar una factura inactiva.', 1;

END

-- Validar que ClienteID existe

IF NOT EXISTS (SELECT 1 FROM CLIENTES WHERE ClienteID = @ClienteID)

BEGIN

THROW 50003, 'El ClienteID especificado no existe.', 1;

END

-- Validar parámetros no vacíos

IF @RazonSocial IS NULL OR @RazonSocial = '' OR @NitCi IS NULL OR @NitCi = ''

BEGIN

THROW 50004, 'RazonSocial y NitCi no pueden estar vacíos.', 1;

END

BEGIN TRANSACTION;

-- Actualizar la factura

UPDATE DATOS\_FACTURA

SET ClienteID = @ClienteID,

RazonSocial = @RazonSocial,

NitCi = @NitCi,

ModifiedDate = GETDATE(),

Estado = 'Activo'

WHERE FacturaID = @FacturaID;

COMMIT TRANSACTION;

-- Devolver mensaje de éxito

SELECT 'Factura actualizada exitosamente.' AS Mensaje,

@FacturaID AS FacturaID;

END TRY

BEGIN CATCH

IF XACT\_STATE() <> 0

ROLLBACK TRANSACTION;

DECLARE @ErrorMessage NVARCHAR(4000) = ERROR\_MESSAGE();

DECLARE @ErrorSeverity INT = ERROR\_SEVERITY();

DECLARE @ErrorState INT = ERROR\_STATE();

THROW 50000, @ErrorMessage, @ErrorState;

END CATCH

END;

GO

**INDICES**

**¿Por qué usamos índices NO CLUSTERIZADOS?**

En esta base de datos decidimos implementar índices no clusterizados (NONCLUSTERED) como complemento a las claves primarias que ya poseen índices CLUSTERED por defecto. Un índice clusterizado define el orden físico de los datos en disco, y como solo puede haber uno por tabla, está asignado automáticamente a la columna PRIMARY KEY en cada tabla.  
  
Los índices no clusterizados son estructuras separadas que no modifican el orden físico de la tabla, y son ideales para acelerar búsquedas, filtros y joins en columnas que no forman parte de la clave primaria. Esta estrategia permite mantener la integridad física de los datos mientras optimizamos el rendimiento en consultas comunes y reportes operativos sin afectar la escritura ni la estructura base.

**Índices creados y su explicación**

|  |  |  |
| --- | --- | --- |
| **Índice SQL** | **¿Es Clusterizado?** | **Explicación breve** |
| **IX\_Inventario\_ProductoID** | **No** | **Acelera búsquedas por ProductoID desde VENTAS y PRODUCTOS.** |
| **IX\_Inventario\_TiendaID** | **No** | **Optimiza las búsquedas de productos por tienda.** |
| **IX\_Inventario\_SubcategoriaID** | **No** | **Útil para mostrar productos por subcategoría.** |
| **IX\_Ventas\_ProductoID** | **No** | **Mejora la consulta de ventas por producto.** |
| **IX\_Ventas\_CompraID** | **No** | **Mejora los reportes de ventas por compra.** |
| **IX\_Productos\_ProductoID** | **No** | **Acelera el acceso al producto desde PRODUCTOS hacia INVENTARIO.** |
| **IX\_Compras\_ClienteID** | **No** | **Facilita consultar las compras hechas por un cliente específico.** |
| **IX\_Compras\_DireccionID** | **No** | **Ayuda a identificar compras por dirección de entrega.** |
| **IX\_Pagos\_CompraID** | **No** | **Acelera los reportes de pagos por compra.** |
| **IX\_Pagos\_FacturaID** | **No** | **Mejora consultas de pagos por número de factura.** |
| **IX\_Tiendas\_UserID** | **No** | **Relaciona la tienda con el usuario que la registró.** |
| **IX\_Tiendas\_CategoriaID** | **No** | **Mejora las búsquedas de tiendas por categoría.** |
| **IX\_Clientes\_UserID** | **No** | **Identifica clientes en base al UserID.** |
| **IX\_Users\_Tipo** | **No** | **Permite filtrar usuarios por tipo (Cliente o Tienda).** |
| **IX\_Direcciones\_ClienteID** | **No** | **Acelera la carga de direcciones de un cliente.** |
| **IX\_DatosFactura\_ClienteID** | **No** | **Mejora consultas de datos de factura por cliente.** |

**NOSQL – MONGODB**

* 1. **Guardar Imagen**

Lee un archivo JPG desde disco, lo codifica en Base64 y lo almacena en la colección de MongoDB asociada a producto\_id. Luego elimina el archivo local. **Lógica:**

1. Convierte producto\_id a entero (por si viene en otro tipo).
2. Abre la conexión a MongoDB y obtiene la colección.
3. Abre el archivo en modo binario, lo codifica a Base64 y decodifica a UTF-8.
4. Inserta un documento con campos:
   * "producto\_id"
   * "imagen" (cadena Base64)
   * "modified\_date" (timestamp UTC)
5. Intenta eliminar el archivo local .jpg; informa éxito o error por consola.

* from MongoConnection import MongoConnection
* import base64
* from datetime import datetime, timezone
* import os
* def guardar\_imagen(ruta\_img\_jpg: str, producto\_id) -> None:
* producto\_id = int(producto\_id)
* mongo = MongoConnection()
* collection = mongo.get\_collection()
* with open(ruta\_img\_jpg, 'rb') as f:
* encoded = base64.b64encode(f.read()).decode('utf-8')
* new\_doc = {
* "producto\_id": producto\_id,
* "imagen":      encoded,
* "modified\_date": datetime.now(timezone.utc)
* }
* collection.insert\_one(new\_doc)
* print("Imagen guardada en MongoDB.")
* try:
* os.remove(ruta\_img\_jpg)
* print(f"Imagen local '{ruta\_img\_jpg}' eliminada.")
* except Exception as e:
* print(f"No se pudo eliminar la imagen local: {e}")

1. **Editar Imagen**

Actualiza la imagen (o la crea si no existía) para producto\_id en MongoDB, y elimina el archivo local. **Lógica:**

1. Convierte y valida producto\_id.
2. Lee y codifica el archivo en Base64.
3. Llama a update\_one(..., upsert=True):
   * Si encuentra un documento coincidente, actualiza imagen y modified\_date.
   * Si no, inserta un nuevo documento.
4. Imprime en consola si se actualizó o insertó.
5. Elimina el archivo local y reporta éxito o error por consola.

from pymongo import MongoClient

import base64

from PIL import Image

from io import BytesIO

import os

from datetime import datetime

from MongoConnection import MongoConnection

def editar\_imagen(producto\_id: int, ruta\_img: str) -> None:

    mc = MongoConnection()

    coll = mc.get\_collection()

    with open(ruta\_img, 'rb') as f:

        encoded = base64.b64encode(f.read()).decode('utf-8')

    # Intentar actualizar documento existente

    result = coll.update\_one(

        { 'producto\_id': int(producto\_id) },

        { '$set': { 'imagen': encoded, 'modified\_date': datetime.utcnow() } },

        upsert=True

    )

    if result.matched\_count:

        print(f"Imagen de producto {producto\_id} actualizada en MongoDB.")

    else:

        print(f"No se encontró imagen previa para producto {producto\_id}. Se insertó nueva.")

    try:

        os.remove(ruta\_img)

        print(f"Archivo local '{ruta\_img}' eliminado.")

    except Exception as e:

        print(f"Error eliminando archivo local: {e}")

1. **Eliminar Imagen**

Elimina físicamente de MongoDB el documento de imagen asociado a producto\_id.

1. Convierte producto\_id a entero.
2. Conecta a MongoDB y obtiene la colección.
3. Ejecuta delete\_one({ 'producto\_id': producto\_id }).
4. Comprueba deleted\_count:
   * 0 → imprime confirmación y retorna True.
   * 0 → imprime que no existía y retorna False.

from MongoConnection import MongoConnection

def eliminar\_imagen(producto\_id: int) -> bool:

    producto\_id = int(producto\_id)

    mongo = MongoConnection()

    collection = mongo.get\_collection()

    result = collection.delete\_one({ 'producto\_id': producto\_id })

    if result.deleted\_count > 0:

        print(f"Imagen de producto {producto\_id} eliminada de MongoDB.")

        return True

    else:

        print(f"No se encontró imagen para producto {producto\_id} en MongoDB.")

        return False

1. **Recuperar Imagen**

Recupera desde MongoDB la imagen codificada en Base64 para producto\_id y la devuelve como objeto PIL.Image. Si no existe, retorna una imagen en blanco/gris de 80×80 px.

1. Conecta a MongoDB y busca con find\_one({ "producto\_id": producto\_id }).
2. Si encuentra el campo "imagen", decodifica Base64 y carga con Image.open.
3. Si no hay documento o no tiene "imagen", crea una imagen neutra (RGB(80×80;(200,200,200))).

from MongoConnection import MongoConnection

import base64

from PIL import Image

from io import BytesIO

def recuperar\_imagen(producto\_id: int) -> Image.Image:

    mongo = MongoConnection()

    coll  = mongo.get\_collection()

    doc = coll.find\_one({"producto\_id": producto\_id})

    if doc and doc.get("imagen"):

        img = Image.open(BytesIO(base64.b64decode(doc["imagen"])))

        return img

    return Image.new("RGB", (80,80), (200,200,200))