

# BitBusters



¡BitBusters, bit a bit,  
conquistamos el mundo de las  
bases de datos un 1 o 0 a la vez!

## UNIVERSIDAD AUTÓNOMA METROPOLITANA

UNIDAD AZCAPOTZALCO

Bases de datos

Proyecto final - 03

**PROFESOR:**

HUGO PABLO LEYVA

**INTEGRANTES DEL EQUIPO:**

DIEGO FLORES CHÁVEZ - 2203031129

CARLOS ADRIÁN DELGADO FRÍAS - 2192000090

**GRUPO:**

CSI81

# Interfaz de consulta

## Índice

<b>Índice</b>	<b>1</b>
<b>Introducción</b>	<b>1</b>
<b>Procedimientos almacenados</b>	<b>1</b>
Estructura general	2
<b>Gatillos</b>	<b>3</b>
Estructura general	3
<b>Resultados</b>	<b>5</b>
Prueba de los procedimientos almacenados	5
Evidencia del funcionamiento de la interfaz	5
Prueba de los gatillos	7
Evidencia del funcionamiento de los gatillos	7

## Introducción

En esta sección, se van a explorar los componentes fundamentales de la lógica de negocio y la gestión de datos en la base de datos. Los procedimientos almacenados y los gatillos, también conocidos como "triggers", son elementos clave en la implementación de la lógica de aplicación directamente en la base de datos, lo que permite la automatización de tareas, la ejecución de operaciones complejas y el mantenimiento de la integridad de los datos. Se presentarán los procedimientos almacenados y los gatillos diseñados específicamente para satisfacer las necesidades operativas y funcionales de Exmicror. A través de ejemplos de código SQL, se detallarán las funcionalidades y el propósito de cada procedimiento y gatillo, junto con referencias al repositorio de GitHub donde se encuentra disponible el código fuente. Además, se proporcionará evidencia del correcto funcionamiento de estos elementos, demostrando su contribución al éxito operativo y la gestión eficiente de datos en el contexto del proyecto "Exmicror".

<https://github.com/DieegoFC/Exmicror-DB><sup>1</sup>

## Procedimientos almacenados

Los procedimientos almacenados juegan un papel crucial en el proyecto "Exmicror" al abordar la necesidad específica de gestionar de manera eficiente y segura las operaciones de base de datos. Su utilización permite encapsular lógica empresarial compleja dentro del entorno de la base de datos, reduciendo así la exposición a errores y mejorando la

---

<sup>1</sup> Dentro del repositorio de GitHub se puede encontrar una referencia completa del código junto con el resto de la documentación. Se recomienda visitarlo para revisar el funcionamiento descrito en este documento con mayor detalle.

integridad de los datos. Además, al evitar el uso de consultas planas directamente desde la aplicación, los procedimientos almacenados contribuyen a una mayor seguridad y modularidad del sistema, al tiempo que proporcionan un mecanismo centralizado para la ejecución de tareas recurrentes y la aplicación de reglas de negocio específicas.

## Estructura general

*Bloque de código 1. Interfaz de consulta para la tabla PackagingDetails.*

```
-- Stored Procedure to insert packaging details
CREATE OR REPLACE FUNCTION insert_packaging_details(
    p_packaging_type VARCHAR(50),
    p_packaging_material VARCHAR(50)
)
RETURNS INTEGER AS $$
DECLARE
    new_id INTEGER;
BEGIN
    INSERT INTO PackagingDetails (packaging_type, packaging_material)
    VALUES (p_packaging_type, p_packaging_material)
    RETURNING PackagingDetails.id_packaging INTO new_id;

    RETURN new_id;
END;
$$ LANGUAGE plpgsql;

-- Stored Procedure to get packaging details by ID
CREATE OR REPLACE FUNCTION get_packaging_details_by_id(p_packaging_id INTEGER)
RETURNS TABLE (
    id_packaging INTEGER,
    packaging_type VARCHAR(50),
    packaging_material VARCHAR(50),
    packaging_date DATE
) AS $$
BEGIN
    RETURN QUERY SELECT * FROM PackagingDetails WHERE PackagingDetails.id_packaging =
p_packaging_id;
END;
$$ LANGUAGE plpgsql;

-- Stored Procedure to update packaging details
CREATE OR REPLACE FUNCTION update_packaging_details(
    p_packaging_id INTEGER,
    p_packaging_type VARCHAR(50),
    p_packaging_material VARCHAR(50)
)
RETURNS BOOLEAN AS $$
BEGIN
    UPDATE PackagingDetails
    SET
        packaging_type = p_packaging_type,
        packaging_material = p_packaging_material
    WHERE PackagingDetails.id_packaging = p_packaging_id;

    RETURN FOUND;
END;
$$ LANGUAGE plpgsql;

-- Stored Procedure to delete packaging details by ID
CREATE OR REPLACE FUNCTION delete_packaging_details(p_packaging_id INTEGER)
RETURNS BOOLEAN AS $$
BEGIN
```

```
DELETE FROM PackagingDetails WHERE PackagingDetails.id_packaging = p_packaging_id;
RETURN FOUND;
END;
$$ LANGUAGE plpgsql;
```

La estructura presentada en el *Bloque de código 1* consiste en una serie de procedimientos almacenados escritos en el lenguaje de programación plpgsql, diseñados para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en la tabla "PackagingDetails" de la base de datos PostgreSQL del proyecto "Exmicror". Cada procedimiento almacenado está dedicado a una operación específica: inserción, consulta por ID, actualización y eliminación de detalles de empaque. La eficiencia del código radica en su capacidad para encapsular la lógica de negocio relacionada con la gestión de los detalles de empaque en la base de datos, ofreciendo un mecanismo estandarizado y coherente para interactuar con los datos de esta tabla.

Es importante destacar que la estructura y funcionamiento de los procedimientos almacenados son consistentes entre sí, lo que permite su reutilización para otras tablas que siguen un patrón similar de CRUD. Por lo tanto, no es necesario incluir todo el código de cada procedimiento almacenado en el documento de la documentación, ya que la estructura y funcionalidad de los procedimientos almacenados son idénticas para cada tabla en la base de datos. Además, dado que el código está disponible en el repositorio de GitHub<sup>2</sup> del proyecto, los interesados pueden acceder fácilmente a él para obtener más detalles o realizar modificaciones según sea necesario.

## Gatillos

Los gatillos desempeñan un papel esencial en el proyecto "Exmicror" al abordar la necesidad específica de implementar lógica de negocio y reglas para garantizar la integridad de la información almacenada en la base de datos PostgreSQL. Estos elementos permiten la automatización de acciones en respuesta a eventos específicos, como inserciones, actualizaciones o eliminaciones de datos en las tablas. Su capacidad para ejecutar acciones predefinidas en tiempo real facilita la aplicación de restricciones, validaciones y procesos de negocio que aseguran la coherencia y precisión de los datos, contribuyendo así a la integridad y confiabilidad del sistema de información de Exmicror.

## Estructura general

*Bloque de código 2. Gatillos en Exmicror.*

```
-- Trigger para la regla de negocio 1: Actualización de La fecha de emisión de La factura
CREATE OR REPLACE FUNCTION check_invoice_issuance_date()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.issuance_date < (SELECT order_date FROM Orders WHERE id_order = NEW.id_order) THEN
        RAISE EXCEPTION 'The invoice issuance date cannot be earlier than the corresponding order
date.';
    END IF;
    RETURN NEW;
```

<sup>2</sup> El código se encuentra dentro del archivo de inicialización *init.sql*, mismo está ubicado en la carpeta "sql" en la raíz del repositorio, junto con la definición de las tablas y las instrucciones ETL.

```

END;
$$ LANGUAGE plpgsql;

-- Asociar el Trigger a La tabla Invoices
CREATE TRIGGER enforce_invoice_issuance_date
BEFORE UPDATE ON Invoices
FOR EACH ROW
EXECUTE FUNCTION check_invoice_issuance_date();

-- Trigger para la regla de negocio 2: Restricción de eliminación de pedidos asociados a facturas
CREATE OR REPLACE FUNCTION prevent_order_deletion_with_invoices()
RETURNS TRIGGER AS $$
BEGIN
    IF EXISTS (SELECT 1 FROM Invoices WHERE id_order = OLD.id_order) THEN
        RAISE EXCEPTION 'Cannot delete orders associated with invoices.';
    END IF;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;

-- Asociar el Trigger a La tabla Orders
CREATE TRIGGER enforce_order_deletion_with_invoices
BEFORE DELETE ON Orders
FOR EACH ROW
EXECUTE FUNCTION prevent_order_deletion_with_invoices();

```

El código proporcionado en el *Bloque de código 2* implementa dos Triggers en el contexto de las reglas de negocio establecidas para la empresa Exmicror. Estos Triggers garantizan el cumplimiento de las reglas de negocio y la integridad referencial de la base de datos.

El primer Trigger se refiere a la regla de negocio 1, que especifica que la fecha de emisión de una factura no puede ser anterior a la fecha de la orden correspondiente. El Trigger `check_invoice_issuance_date` se dispara antes de actualizar una factura y verifica si la nueva fecha de emisión (`NEW.issuance_date`) es menor que la fecha de la orden correspondiente obtenida de la tabla `Orders`. Si la condición no se cumple, se genera una excepción que indica que la fecha de emisión de la factura no puede ser anterior a la fecha de la orden. Esto asegura que las fechas de las facturas estén sincronizadas con las fechas de las órdenes, lo que es crucial para el seguimiento preciso de las transacciones y la contabilidad.

El segundo Trigger está relacionado con una nueva regla de negocio que impide la eliminación de pedidos asociados a facturas. Este Trigger, llamado `enforce_order_deletion_with_invoices`, se activa antes de que se elimine un pedido de la tabla `Orders`. Si el pedido que se intenta eliminar está asociado a una factura en la tabla `Invoices`, el Trigger lanzará una excepción y evitará la eliminación del pedido. Esto asegura que no se eliminen pedidos que tengan facturas asociadas, lo que ayuda a mantener la integridad de los datos y evita posibles inconsistencias en la base de datos.

La importancia de estos Triggers radica en su capacidad para automatizar la aplicación de las reglas de negocio y mantener la integridad de los datos de la base de datos. Al garantizar que las reglas de negocio se cumplan automáticamente, se reduce la posibilidad de errores humanos y se mejora la confiabilidad de los datos. Además, estos Triggers

mejoran la eficiencia operativa al eliminar la necesidad de intervención manual para aplicar las reglas de negocio.

# Resultados

## Prueba de los procedimientos almacenados

Aquí se presentan ejemplos representativos del funcionamiento de los procedimientos almacenados diseñados para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en la base de datos. Estas evidencias se seleccionan con el propósito de demostrar la coherencia y efectividad de la estructura y funcionamiento de los procedimientos almacenados, los cuales siguen un patrón común a lo largo de la base de datos.

## Evidencia del funcionamiento de la interfaz

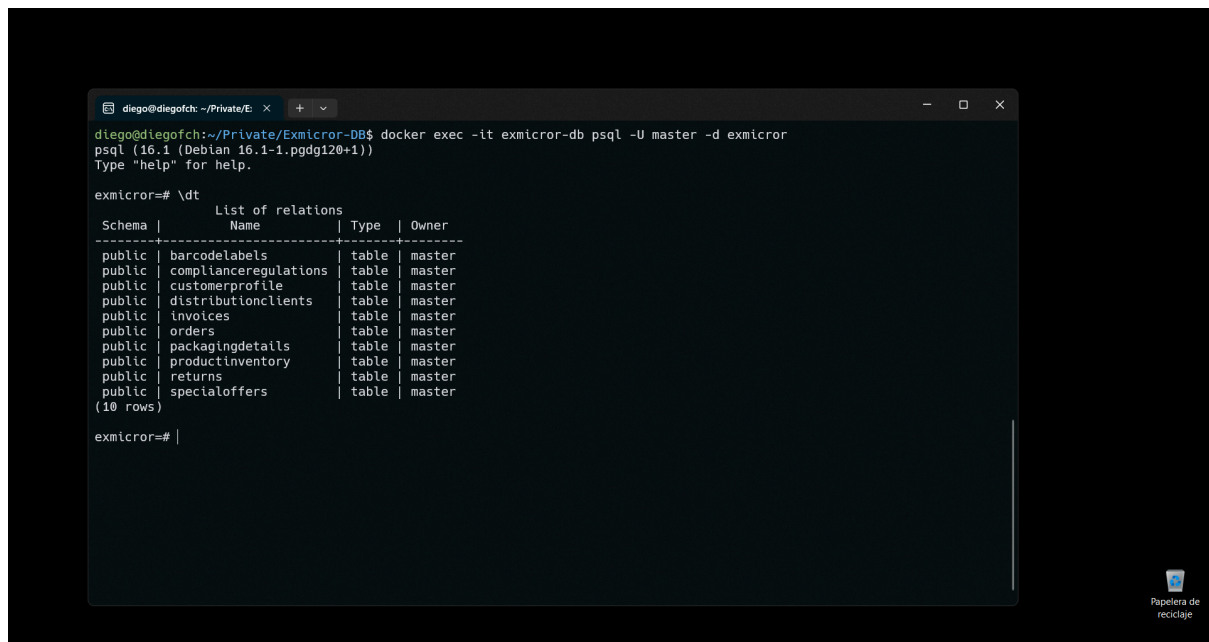


Figura 1. Levantamiento exitoso del contenedor de la base de datos.

```

diego@diegofch: ~/Private/E
exmicror=# SELECT * FROM barcodeLabels;
id_label | label_information | barcode
-----|-----|-----
1 | Exmicror Desinfectante 500ml - Frescura de Limón | ABC123
2 | Exmicror Desinfectante 750ml - Aroma a Lavanda | DEF456
3 | Exmicror Desinfectante 1L - Fragancia de Eucalipto | GHI789
4 | Exmicror Desinfectante 250ml - Esencia de Pino | JKL012
5 | Exmicror Desinfectante 2L - Frescor de Naranja | MNO345
6 | Exmicror Desinfectante 5L - Olor a Pera | PQR678
7 | Exmicror Desinfectante 100ml - Aroma a Manzana | STU901
8 | Exmicror Desinfectante 150ml - Fragancia a Coco | VWX234
9 | Exmicror Desinfectante 3L - Esencia de Melocotón | YZAB567
10 | Exmicror Desinfectante 10L - Frescura de Sandía | CDE890
(10 rows)

exmicror=#

```

Figura 2. Tabla BarcodeLabels con el contenido original de la carga ETL.

```

diego@diegofch: ~/Private/E
exmicror=# SELECT * FROM barcodeLabels;
id_label | label_information | barcode
-----|-----|-----
1 | Exmicror Desinfectante 500ml - Frescura de Limón | ABC123
2 | Exmicror Desinfectante 750ml - Aroma a Lavanda | DEF456
3 | Exmicror Desinfectante 1L - Fragancia de Eucalipto | GHI789
4 | Exmicror Desinfectante 250ml - Esencia de Pino | JKL012
5 | Exmicror Desinfectante 2L - Frescor de Naranja | MNO345
6 | Exmicror Desinfectante 5L - Olor a Pera | PQR678
7 | Exmicror Desinfectante 100ml - Aroma a Manzana | STU901
8 | Exmicror Desinfectante 150ml - Fragancia a Coco | VWX234
9 | Exmicror Desinfectante 3L - Esencia de Melocotón | YZAB567
10 | Exmicror Desinfectante 10L - Frescura de Sandía | CDE890
(10 rows)

exmicror=# SELECT insert_barcode_label('Prueba', 'X');
insert_barcode_label
-----
11
(1 row)

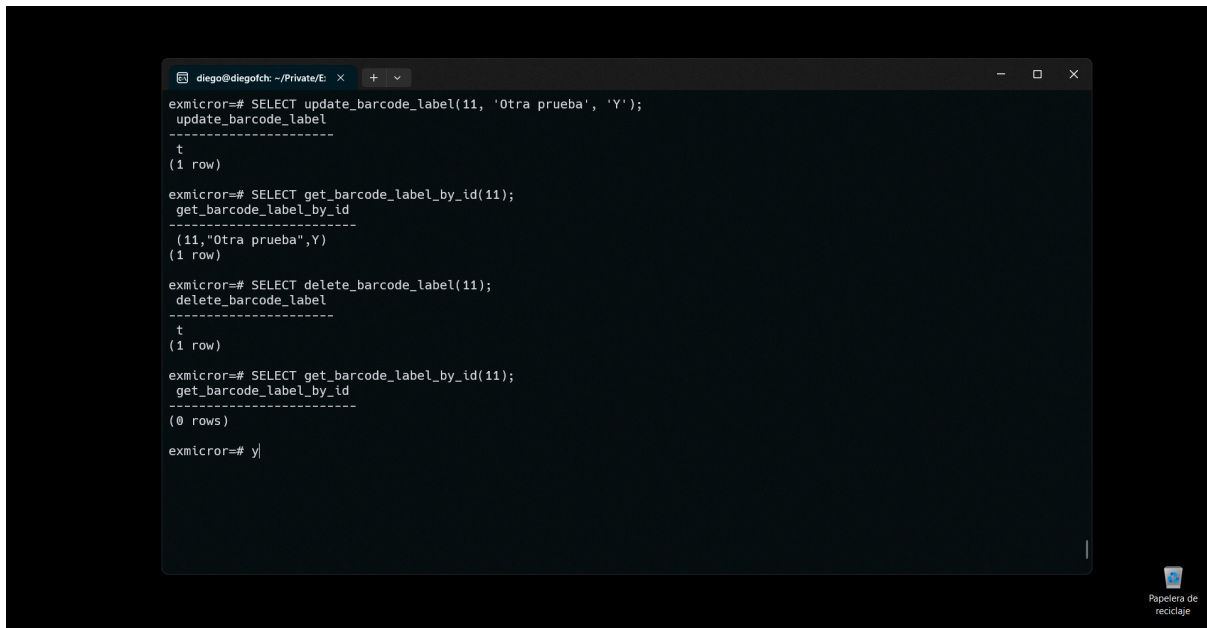
exmicror=# SELECT get_barcode_label_by_id(11);
get_barcode_label_by_id
-----
(11,Prueba,X)
(1 row)

exmicror=#

```

Figura 3. Creación y lectura exitosa en BarcodeLabels.





```

diego@diegofch: ~/Private/E: x + v
exmicror=# SELECT update_barcode_label(11, 'Otra prueba', 'Y');
update_barcode_label
-----
t
(1 row)

exmicror=# SELECT get_barcode_label_by_id(11);
get_barcode_label_by_id
-----
(11,"Otra prueba",Y)
(1 row)

exmicror=# SELECT delete_barcode_label(11);
delete_barcode_label
-----
t
(1 row)

exmicror=# SELECT get_barcode_label_by_id(11);
get_barcode_label_by_id
-----
(0 rows)

exmicror=# y|

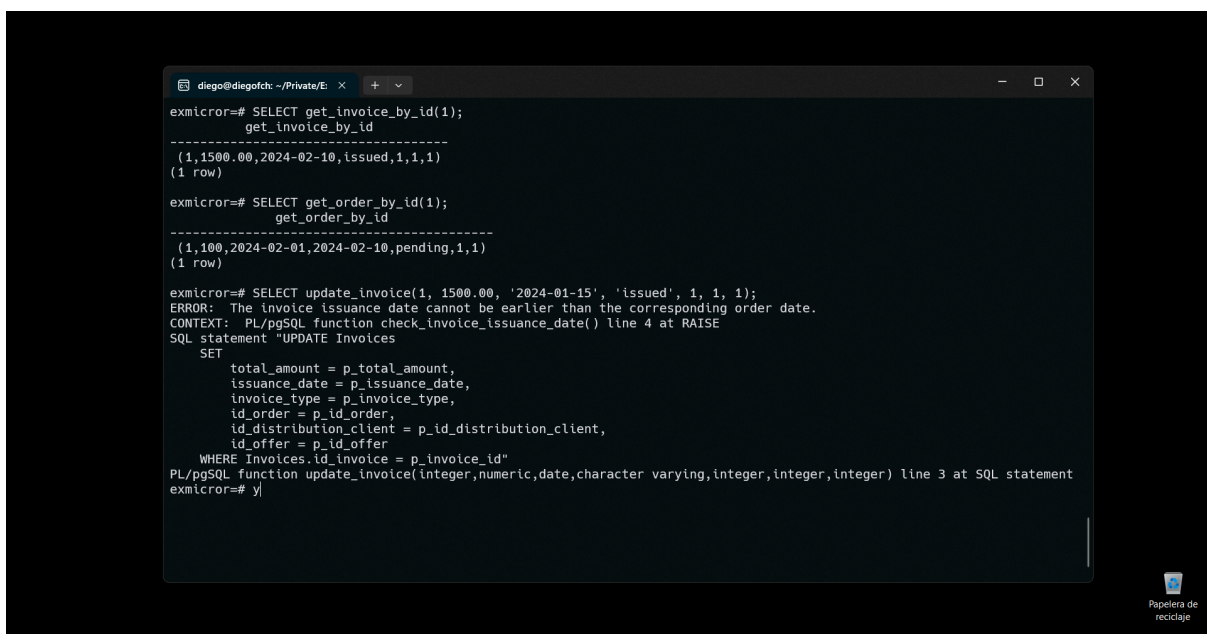
```

Figura 4. Actualización y eliminación exitosa en BarcodeLabels.

## Prueba de los gatillos

Se presentará un ejemplo representativo que ilustre la efectividad de los gatillos en mantener la integridad de la base de datos. Este ejemplo ha sido seleccionado por su consistencia estructural y su capacidad para ejecutar acciones automáticas en respuesta a eventos específicos. A través de evidencias concretas, se demostrará cómo estos gatillos contribuyen a asegurar la coherencia y confiabilidad de los datos en el sistema de información de Exmicror, fortaleciendo así la integridad del sistema en su totalidad.

## Evidencia del funcionamiento de los gatillos



```

diego@diegofch: ~/Private/E: x + v
exmicror=# SELECT get_invoice_by_id(1);
get_invoice_by_id
-----
(1,1500.00,2024-02-10,issued,1,1,1)
(1 row)

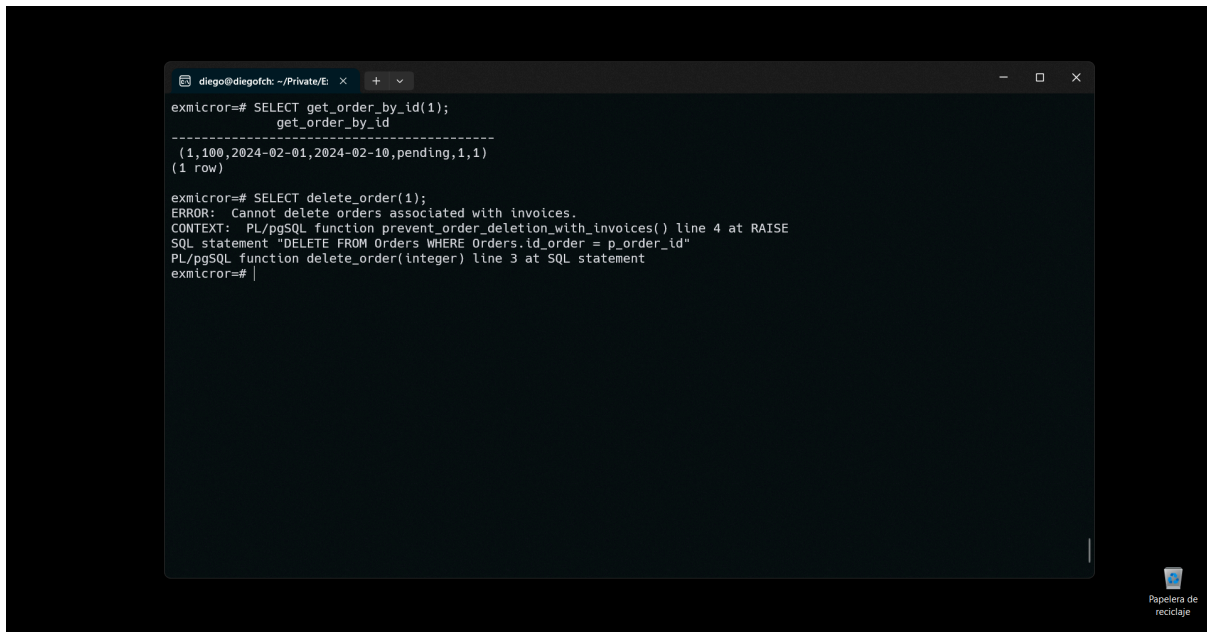
exmicror=# SELECT get_order_by_id(1);
get_order_by_id
-----
(1,100,2024-02-01,2024-02-10,pending,1,1)
(1 row)

exmicror=# SELECT update_invoice(1, 1500.00, '2024-01-15', 'issued', 1, 1, 1);
ERROR:  The invoice issuance date cannot be earlier than the corresponding order date.
CONTEXT:  PL/pgSQL function check_invoice_issuance_date() line 4 at RAISE
SQL statement "UPDATE Invoices
SET
    total_amount = p_total_amount,
    issuance_date = p_issuance_date,
    invoice_type = p_invoice_type,
    id_order = p_id_order,
    id_distribution_client = p_id_distribution_client,
    id_offer = p_id_offer
WHERE Invoices.id_invoice = p_invoice_id"
PL/pgSQL function update_invoice(integer,numeric,date,character varying,integer,integer,integer) line 3 at SQL statement
exmicror=# y|

```

Figura 5. Prueba exitosa para el gatillo de la primera regla de negocio.





```

diego@diegofch: ~/Private/E
exmicror=# SELECT get_order_by_id(1);
      get_order_by_id
-----
(1,100,2024-02-01,2024-02-10,pending,1,1)
(1 row)

exmicror=# SELECT delete_order(1);
ERROR:  Cannot delete orders associated with invoices.
CONTEXT:  PL/pgSQL function prevent_order_deletion_with_invoices() line 4 at RAISE
SQL statement "DELETE FROM Orders WHERE Orders.id_order = p.order_id"
PL/pgSQL function delete_order(integer) line 3 at SQL statement
exmicror=#

```

Figura 6. Prueba exitosa para el gatillo de la segunda regla de negocio.

## Conclusión

Los procedimientos almacenados y los gatillos (triggers) son herramientas fundamentales en la gestión de bases de datos, particularmente en el caso de Exmicror. Si bien su desarrollo puede presentar ciertas dificultades, como la complejidad en su sintaxis y la necesidad de comprender profundamente la estructura y los requerimientos del sistema, su implementación ofrece una serie de ventajas significativas.

En primer lugar, los procedimientos almacenados permiten encapsular lógica de negocio compleja dentro de la base de datos, lo que promueve la reutilización del código, mejora la seguridad y la integridad de los datos, y reduce la redundancia de operaciones. Esto se traduce en un código más eficiente y mantenible, así como en un rendimiento mejorado del sistema.

Por otro lado, los gatillos proporcionan un mecanismo automatizado para aplicar acciones o validar datos en respuesta a eventos específicos en la base de datos, como inserciones, actualizaciones o eliminaciones de registros. Esto garantiza que se cumplan las reglas de negocio establecidas y se mantenga la consistencia de los datos en todo momento.

A pesar de las ventajas que ofrecen, es importante tener en cuenta que tanto los procedimientos almacenados como los gatillos deben ser diseñados y probados cuidadosamente para evitar efectos no deseados o impactos negativos en el rendimiento del sistema. Además, su complejidad puede dificultar la depuración y el mantenimiento, por lo que es fundamental contar con un proceso de desarrollo riguroso y una documentación clara y detallada.