



Universidad Nacional del Litoral

Facultad de Ingeniería y Ciencias Hídricas

Departamento de Informática

Bases de Datos

SQL: Guía de Trabajo Nro. 6

Tratamiento de errores

Parte 1

1. Valores SQLSTATE

El estándar SQL define una variable de estado especial, una string de cinco caracteres llamada `SQLSTATE` que define el status de la última operación realizada.

PostgreSQL categoriza sus errores de esta manera, con valores `SQLSTATE`.

Los dos primeros dígitos definen en PostgreSQL la categoría (o Class) del error.

Por ejemplo, tenemos la Class 23 (Integrity Constraint Violation):

Class 23 — Integrity Constraint Violation	
23000	<code>integrity_constraint_violation</code>
23001	<code>restrict_violation</code>
23502	<code>not_null_violation</code>
23503	<code>foreign_key_violation</code>
23505	<code>unique_violation</code>
23514	<code>check_violation</code>
23P01	<code>exclusion_violation</code>



En el documento *PostgreSQL Error Codes* podemos encontrar un listado de los valores `SQLSTATE` definidos en PostgreSQL.



[PostgreSQL Error Codes](#)

Cuando programamos con PL/pgSQL, los errores que más se evalúan y utilizan son los de la Class P0:

Class P0 — PL/pgSQL Error	
P0000	plpgsql_error
P0001	raise_exception
P0002	no_data_found
P0003	too_many_rows



T-SQL no utiliza el sistema de error handling basado en `SQLSTATE`.

2. Errores SQL Server

2.1. Tipos de errores

SQL Server define dos tipos de errores:

Fatales

Provocan que el procedimiento o batch aborte su procesamiento y finalice la conexión con la aplicación cliente.

No fatales

No abortan el procesamiento ni afectan la conexión con la aplicación cliente.

Cuando ocurre un error no fatal dentro de un procedimiento, el procesamiento continúa en la línea de código siguiente a la que provocó el error.

2.2. Componentes de un error

Cada error SQL Server posee cuatro partes:

Number

Sería algo equivalente a la variable ANSI `SQLSTATE`.

En SQL Server es simplemente un número entero.

Message

El texto del error tiene por objeto comunicar de la condición de error al usuario.

Severity

Es un entero de 0 a 25, donde un número más alto significa mayor severidad.

Severity	Significado
0 a 10	Mensajes informativos
11 a 16	Indican diferentes grados de error de usuario. Por ejemplo, referenciar una tabla que no existe o cometer un error sintáctico.
17 y 18	Fallas a nivel del sistema
19 a 25	Errores fatales. Son lo suficientemente críticos como para finalizar la conexión con el cliente.

State

Hace referencia al contexto en el cual sucedió el error. Es un valor entero entre 0 y 127. En los errores de aplicación normalmente se utiliza el valor 1.

3. PL/pgSQL: la cláusula STRICT

En la *Guía de Trabajo Nro. 4* - analizamos las *Sentencias SQL Single-Row*:



Recordemos que recuperábamos los valores de una sentencia **SELECT single-row** con una cláusula **INTO** que especificaba las variables donde deben ser ubicados los componentes de esa única tupla:

```
DECLARE
    price1 titles.price%TYPE;
    type1 titles.type%TYPE;

BEGIN
    SELECT price, type INTO price1, type1
    FROM titles
    WHERE title_id = 'TC7777';

    RAISE NOTICE 'El precio es %', price1;
    ...

END
```

A la luz del manejo de errores, debemos que agregar algo más acerca de este tipo de sentencias.

3.1. No se encuentran datos

Supongamos la siguiente función:

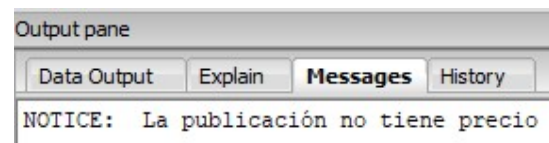
Ejemplo 1

```
CREATE OR REPLACE FUNCTION test7B()  
  RETURNS VOID  
  LANGUAGE plpgsql  
  AS  
  DECLARE  
    recTitles RECORD;  
  
  BEGIN  
    SELECT price, type INTO recTitles  
      FROM titles  
      WHERE title_id = 'VVU7777'; (A)  
  
    IF recTitles.price IS NULL THEN (B)  
      RAISE NOTICE 'La publicación no tiene precio';  
    END IF;  
  
  END
```

La publicación 'VVU7777' no existe.
Sin embargo, la función no dispara ninguna excepción.

Lo comprobamos en **(B)**, preguntando si `recTitles.price` es nulo.

Obtenemos:



Efectivamente `recTitles.price` asumió silenciosamente un valor `NULL`.

3.2. No es una sentencia SELECT Single-row

Veamos ahora la siguiente función:

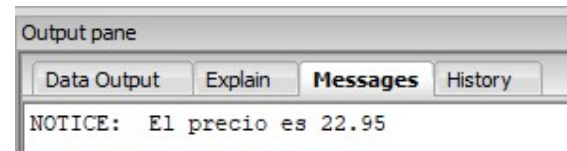
Ejemplo 2

```
CREATE OR REPLACE FUNCTION test7C()  
  RETURNS VOID  
  LANGUAGE plpgsql  
  AS  
  DECLARE  
    recTitles RECORD;  
  
  BEGIN  
    SELECT price, type INTO recTitles A  
    FROM titles;  
  
    RAISE NOTICE 'El precio es %', recTitles.price; B  
  
  END
```

La consulta no tiene cláusula `WHERE` y claramente retornará más de una tupla. Sin embargo, la función tampoco dispara ninguna excepción.

Lo comprobamos en **(B)**.

Obtenemos:



Efectivamente `recTitles.price` asumió el valor de la primer tupla de la relación obtenida:

```
SELECT price, type  
FROM titles;
```

The screenshot shows a window titled 'Output pane' with tabs for 'Data Output', 'Explain', and 'Messages'. The 'Data Output' tab is selected, displaying a table with 6 rows of data.

	price numeric(10,2)	type character(12)
1	22.95	popular comp
2	20.00	popular comp
3	19.99	business
4	7.99	psychology
5	19.99	psychology
6	11.95	business

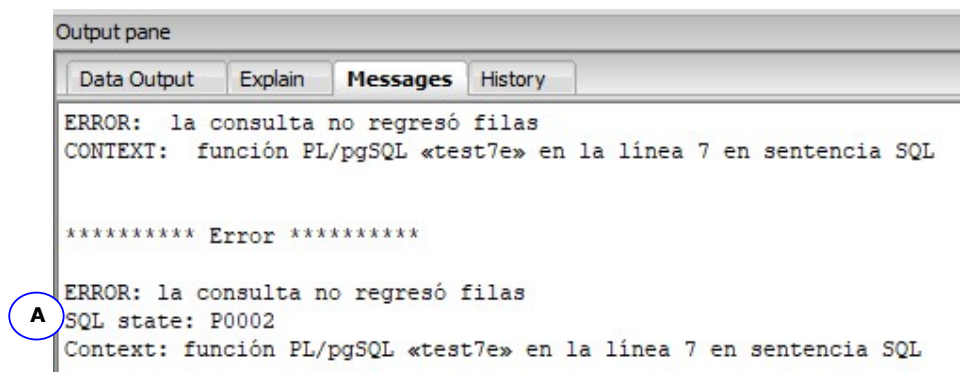
Las dos situaciones representan dos de las excepciones más comunes cuando programamos en un lenguaje de base de datos: `NO_DATA_FOUND` y `TOO_MANY_ROWS` respectivamente.

Para que PL/pgSQL las considere excepciones debemos agregar a la sentencia `SELECT... INTO` la cláusula `STRICT`.

Ejemplo 3

```
CREATE OR REPLACE FUNCTION test7E()  
  RETURNS VOID  
  LANGUAGE plpgsql  
  AS  
  DECLARE  
    recTitles RECORD;  
  
  BEGIN  
    SELECT price, type INTO STRICT recTitles  
      FROM titles  
      WHERE title_id = 'VVU7777';  
  
    IF recTitles.price IS NULL THEN  
      RAISE NOTICE 'La publicación no tiene precio';  
    END IF;  
  
  END
```

Obtenemos:



Podemos observar (A) que obtenemos el valor SQLSTATE P0002 que vimos al principio:

Class P0 — PL/pgSQL Error	
P0000	plpgsql_error
P0001	raise_exception
P0002	no_data_found
P0003	too_many_rows

Ejemplo 4

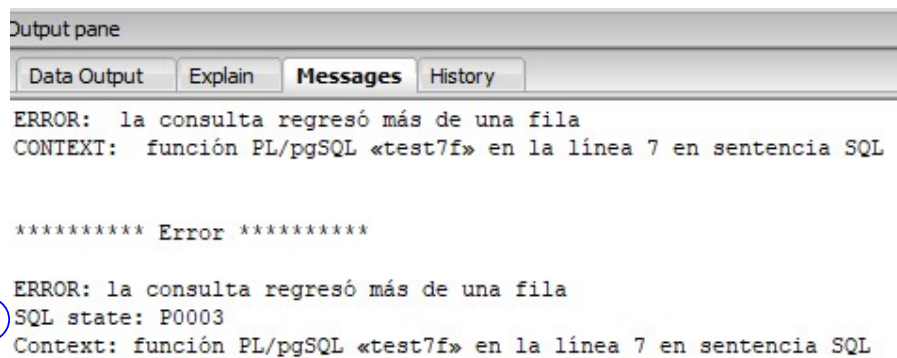
```
CREATE OR REPLACE FUNCTION test7F()
  RETURNS VOID
  LANGUAGE plpgsql
  AS
  DECLARE
    recTitles RECORD;

  BEGIN
    SELECT price, type INTO STRICT recTitles
      FROM titles;

    RAISE NOTICE 'El precio es %', recTitles.price;

  END
```

Ahora obtenemos:



Podemos observar **(B)** que obtenemos el valor SQLSTATE P0003 que vimos al principio:

Class P0 — PL/pgSQL Error	
P0000	plpgsql_error
P0001	raise_exception
P0002	no_data_found
P0003	too_many_rows

4. Captura de excepciones

Hasta ahora hemos visto como los DBMSs registran sus códigos y mensajes de error, pero todavía no vimos como realizar efectivamente el CATCH de un error o excepción.

4.1. PL/pgSQL

Vamos a tomar el mismo ejemplo que venimos analizando para demostrar la captura de excepciones.

Ejemplo 5

```
CREATE OR REPLACE FUNCTION test7G()
  RETURNS VOID
  LANGUAGE plpgsql
  AS
  DECLARE
    recTitles RECORD;

  BEGIN

    NULL;
    --- Otras operaciones---

    ----- Try/catch I -----

    A BEGIN
      SELECT price, type INTO STRICT recTitles
      FROM titles;

      RAISE NOTICE 'El precio es %', recTitles.price;

    B EXCEPTION -- Try I

    C WHEN NO_DATA_FOUND THEN
      RAISE NOTICE 'No se encontraron datos';
      RETURN;

      WHEN TOO_MANY_ROWS THEN
      RAISE NOTICE 'La sentencia SELECT retornó más de una fila';
      RETURN;

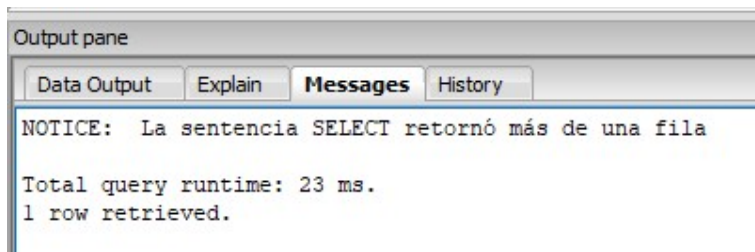
    D WHEN OTHERS THEN
      RAISE NOTICE 'ERROR Others';
      RETURN;

    END; -- Try I

    RETURN;

  END
```

En este caso obtenemos:



```
Output pane
Data Output Explain Messages History
NOTICE: La sentencia SELECT retornó más de una fila
Total query runtime: 23 ms.
1 row retrieved.
```

`BEGIN` (A) define lo que sería el “try”. O sea, vamos a “intentar” ejecutar un código “peligroso” que sabemos a priori puede disparar una excepción.

La cláusula `EXCEPTION` (B) define lo que sería el “catch”.

Para cada error que queramos discriminar, se define una cláusula `WHEN` (C)

En la cláusula `WHEN` también podemos especificar el valor `SQLSTATE`. Por ejemplo:

```
WHEN SQLSTATE '42P01' THEN
```

Además de los errores de programación SQL `NO_DATA_FOUND` y `TOO_MANY_ROWS` hemos agregado un “catch all” llamado `OTHERS` (D).

Cuando ya no nos interesa -o tal vez no tuvimos tiempo- de seguir discriminando por errores específicos, usamos `WHEN OTHERS` para hacer un catch de cualquier error no discriminado específicamente en el resto de las cláusulas `WHEN`.

Pensemos que nuestro código puede ser tan complejo como para abrir archivos, leer JSON, escribir BLOBs o enviar emails, y cada prestación del motor de base de datos traerá un conjunto de posibles excepciones asociadas que deberemos estudiar y tratar.

Por supuesto que cada bloque de manejo de excepción debe tener las cláusulas `WHEN` concordantes con la operación que estamos realizando.

En el ejemplo 5 la cláusula `WHEN NO_DATA_FOUND` es inapropiada ya que estamos seguros de que la sentencia `SELECT` retornará resultados.

(a menos que la tabla haya sido borrada, por ejemplo. en tal caso obtendremos **otra** excepción, no `NO_DATA_FOUND`).

4.1.1. Ejemplo de uso de bloques Exception en una función compleja

----- Try/catch I -----

BEGIN

Operación peligrosa I
-- Hacer fileopen de un archivo
--Si tuvo éxito, continúo.

----- Try/catch II -----

BEGIN

Operacion peligrosa II
-- Enviar un email al administrador
--Si tuvo éxito, continúo.

----- Try/catch III -----

BEGIN

Operacion peligrosa III
--Realizar un INSERT
--Si tuvo éxito, continúo.

EXCEPTION -- Try III
WHEN OTHERS THEN
 --Mostrar información del error;
END;

EXCEPTION -- Try II
WHEN OTHERS THEN
 --Mostrar información del error;
END;

EXCEPTION -- Try I
WHEN OTHERS THEN
 ----Mostrar información del error;
END;

4.1.2. Mostrar información del error

Debajo del `WHEN` podemos hacer lo que creamos conveniente. Usualmente mostraremos una descripción del error. Una forma de hacerlo es a través de `RAISE NOTICE`. Dos variables auxiliares, `SQLERRM` y `SQLSTATE` nos proporcionan el mensaje y el código `SQLSTATE` respectivamente.

A continuación la función mejorada mostrando información del error:

```
CREATE OR REPLACE FUNCTION test7H()
  RETURNS VOID
  LANGUAGE plpgsql
  AS
  DECLARE
    recTitles RECORD;

  BEGIN

    NULL;
    --- Otras operaciones---

    ----- Try/catch I -----
    BEGIN
      SELECT price, type INTO STRICT recTitles
        FROM titles;

      RAISE NOTICE 'El precio es %', recTitles.price;

    EXCEPTION -- Try I

      WHEN TOO_MANY_ROWS THEN
        RAISE NOTICE 'ERROR SQLERRM: % SQLSTATE: %', SQLERRM, SQLSTATE;
        RETURN;

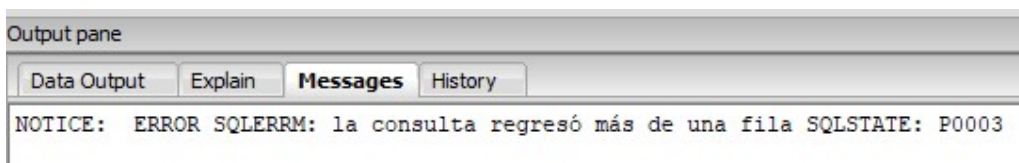
      WHEN OTHERS THEN
        RAISE NOTICE 'ERROR SQLERRM: % SQLSTATE: %', SQLERRM, SQLSTATE;
        RETURN;

    END; -- Try I

    RETURN;

  END
```

Obtenemos:



4.2. T-SQL



Catch de errores usando @@ERROR

T-SQL proporciona una variable del sistema llamada `@@error` que es específica por cada conexión al DBMS.

`@@error` contiene el número del error (El **Number** de la Sección 2.2) producido por la última sentencia SQL ejecutada por esa conexión cliente.

Un valor 0 indica ausencia de error (la sentencia se ejecutó de manera exitosa).

Un valor distinto de cero indica una condición de error.

En el siguiente batch T-SQL capturamos un error de tabla no existente:

```
DECLARE @Error Integer
(A) DROP TABLE noexiste
SET @Error = @@Error (B)

IF @Error != 0
(C) PRINT 'Ocurrió el error ' + Convert(varchar,@Error);
-- END IF
```

En (A) declaramos una variable local para retener el valor de la variable del sistema `@@Error`.

Luego de cada operación que sabemos puede provocar una excepción, debemos recuperar el valor de `@@Error` (B).

Si el valor obtenido es diferente de cero (C) significa que ocurrió una excepción. Lo que hagamos aquí es análogo a lo que hacemos en PL/pgSQL debajo de la cláusula `WHEN`. Usualmente mostraremos una descripción lo más detallada posible del error.



Usando @@ERROR con GoTo

Cuando usamos `@@Error`, la sentencia `GOTO` se puede usar para reducir la cantidad de código necesaria para implementar manejo de errores. Supongamos que necesitamos ejecutar n operaciones "peligrosas" (**A** y **B**) en el ejemplo:

```
DECLARE @Error Int
BEGIN TRANSACTION

A INSERT Prueba Values (@Col2)
SET @Error = @@Error
IF @Error <> 0 GOTO lblError

B INSERT Prueba Values (@Col2)
SET @Error = @@Error
IF @Error <> 0 GOTO lblError

...

COMMIT TRANSACTION

C lblError:
ROLLBACK TRANSACTION
RETURN @Error
```

Por cada operación recuperamos el valor de `@@Error`, pero, si existe error, en vez de tratarlo en el lugar n veces, redireccionamos el control a una sección especial definida por una label (**C**). En esa sección tratamos una única vez la condición de error.



Catch de errores usando try/catch

T-SQL también nos permite capturar errores utilizando bloques try/catch:

```
CREATE PROCEDURE usp_GetErrorInfo (F)
AS
    SELECT ERROR_NUMBER() AS ErrorNumber,
           ERROR_MESSAGE() AS ErrorMessage,
           ERROR_SEVERITY() AS ErrorSeverity,
           ERROR_STATE() AS ErrorState,
           ERROR_PROCEDURE() AS ErrorProcedure,
           ERROR_LINE() AS ErrorLine (E)

(A) BEGIN TRY
    DROP TABLE noexiste (C)
    -- ...otras sentencias
    COMMIT TRANSACTION
END TRY

(B) BEGIN CATCH (D)
    EXECUTE usp_GetErrorInfo
    ROLLBACK TRANSACTION
END CATCH
```

T-SQL define explícitamente un bloque "try" y un bloque "catch" **(A)** y **(B)** como los usuales en muchos lenguajes de programación. Los bloques se definen a través de las sentencias `BEGIN TRY/END TRY` y `BEGIN CATCH/END CATCH` respectivamente.

Dentro del bloque "try" ejecutamos las sentencias peligrosas **(C)**.

Dentro del bloque "catch" tratamos las excepciones **(D)**.

T-SQL proporciona cuatro funciones que nos permiten averiguar las características de un error en un bloque try/catch. Estas son `ERROR_NUMBER()`, `ERROR_MESSAGE()`, `ERROR_SEVERITY()` y `ERROR_STATE()` **(E)**, que retornan el número de error, su message, su severity y su state respectivamente. Si bien estas funciones solo están disponibles dentro de un bloque `CATCH`, podemos incluirlas en un stored procedure **(F)** a fin de no duplicar código.

En el caso del ejemplo obtenemos:

ErrorNumber	ErrorMessage	ErrorSeverity	ErrorState	ErrorProcedure	ErrorLi
3701	Cannot drop the table 'noexiste', because it does not exis...	11	5	NULL	3

5. Errores de aplicación

Los errores que hemos visto hasta ahora son errores que dispara automáticamente el DBMS ante una situación inesperada, provocada por nosotros o por el mismo sistema.

Sin embargo, existen otro tipo de “errores” que no ponen en peligro el DBMS pero sí son significativos para el dominio de nuestra aplicación. Un ejemplo sería la violación de una “regla de negocio”. A estos errores le podemos llamar **errores de aplicación**.

5.1. Disparar errores de aplicación



En T-SQL disparamos un error de aplicación usando la función `RAISERROR`. Por ejemplo:

```
RAISERROR ('Codigo de producto inexistente', 16, 1)
```

(A)

(B)

(C)

(A) es el mensaje de error.

(B) es la **severity**. Los valores válidos son enteros de 0 a 25, pero las severidades 19 a 25 están reservadas para usuarios especiales, como el administrador del sistema. Un error de aplicación debería tener una severidad de 0 a 18.

(C) es un entero llamado state. Normalmente se especifica el valor 1.

Los tres parámetros son obligatorios.

De manera similar a como sucede con `RAISE NOTICE` en PL/pgSQL, la string puede especificar placeholders cuyos valores son especificados como argumentos adicionales a la función. Por ejemplo:

```
RAISERROR ('El producto %s no existe', 16, 1, @cod_prod)
```

`%d` es utilizado para placeholders numéricos, mientras que `%s` es utilizado para placeholders de tipo string.

Por supuesto, `RAISERROR` modifica el valor de `@@Error`.



Disparamos un error en PL/pgSQL usando la sentencia `RAISE EXCEPTION`. Por ejemplo:

```
RAISE EXCEPTION 'El producto % no existe', vCod_prod
    USING ERRCODE = 'P0001';
```

`RAISE EXCEPTION` hace uso de placeholders (**A**) de la misma manera que `RAISE NOTICE`.

La cláusula `USING ERRCODE` (**B**) nos permite especificar el valor `SQLSTATE` del error que estamos disparando..

El valor `SQLSTATE` por omisión para `RAISE EXCEPTION` es `P0001` (`raise_exception`)

En PostgreSQL tenemos disponibles class codes asignables a errores de aplicación. Por ejemplo, letras entre la I y la Z.

Por ejemplo, podríamos decidir que `'El producto % no existe'` posea un `SQLSTATE` `I0001`:

6. Errores y transacciones

La ocurrencia de un error no siempre deshace una transacción.

Algunas operaciones sobre la base de datos pueden llevarse a cabo **aún cuando se haya producido un error**.

De ahí que es fundamental tratar las transacciones en función de los errores.

PL/pgSQL

Podemos asegurarnos que todas las operaciones se lleven a cabo o –en caso de excepción– ninguna de ellas se lleve a cabo, incluyendo todas las sentencias “peligrosas” en el bloque “catch” y encerrando la ejecución de la función en una transacción:

```
CREATE OR REPLACE FUNCTION testE()  
  RETURNS void  
  LANGUAGE plpgsql  
  AS  
  BEGIN  
  
    BEGIN  
      INSERT INTO publishers values('9988', 'Editora Ingenieria Web',  
                                   'Texas', 'TA', 'USA');  
      UPDATE publishers  
        SET pub_name = 'Editora Ingenieria Web 2000'  
        WHERE pub_id = '9988';  
      DROP TABLE noexiste;  
    EXCEPTION  
      WHEN SQLSTATE '42P01' THEN  
        RAISE NOTICE 'ERROR SQLERRM: % SQLSTATE: %',  
                      SQLERRM, SQLSTATE;  
        RETURN;  
    END;  
  
    RETURN;  
  END
```

```
BEGIN TRANSACTION;  
  SELECT test();  
COMMIT TRANSACTION;
```

Output pane

Data Output Explain Messages History

NOTICE: ERROR SQLERRM: no existe la tabla «noexiste» SQLSTATE: 42P01
Query result with 1 row discarded.



El approach usual en caso de error dentro de una transacción es realizar el "catch" de la excepción y hacer un ROLLBACK manual.

Por ejemplo:

```
BEGIN TRANSACTION

BEGIN TRY
    INSERT
        INTO publishers
        values('9988', 'Editora Ingenieria Web', 'Texas', 'TA',
            'USA');

    UPDATE publishers
        SET pub_name = 'Editora Ingenieria Web 2000'
        WHERE pub_id = '9988';

    DROP TABLE noexiste;

    COMMIT TRANSACTION
END TRY

BEGIN CATCH
    EXECUTE usp_GetErrorInfo
    ROLLBACK TRANSACTION
END CATCH
```

Obtenemos:

ErrorNumber	ErrorMessage	ErrorSeverity	ErrorState	ErrorProcedure	ErrorLine
3701	Cannot drop the table 'noexiste', because it do...	11	5	NULL	11

...y ninguna de las sentencias es hecha permanente en la base de datos.