



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

1. Códigos de error en el ámbito del DBMS

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.

Los status codes `SQLSTATE` consisten de un **class code** de dos caracteres y de un **subclass code** de tres caracteres.

1.1.1. Categorías de excepciones

Un **class code** de dos caracteres seguido tres ceros (000, que significa *not applicable*) para el **subclass code** definen códigos `SQLSTATE` que representan **categorías de excepción**.

La única excepción es el código `SQLSTATE` **00000** (successful completion) que indica ausencia de excepción.

Los siguientes son ejemplos de algunas categorías de excepción:

SQLSTATE	Condición	Significado
01000	warning	
02000	no_data	Una tupla solicitada por un query no ha podido ser encontrada.
21000	cardinality_violation	Un SELECT single-row ha retornado más de una tupla.
22000	data_exception	Errores de tipos de datos, valores NULL, argumentos no válidos, etc.
23000	integrity_constraint_violation	Violación de alguna constraint unique, foreign key, check, etc.
40000	transaction_rollback	Por alguna razón el DBMS hizo un rollback (falla en el intento de ejecución serial, error de deadlock, etc.)

1.1.2. Class codes reservados y disponibles

Los siguientes **class codes** están reservados por ANSI:

- Los que comienzan con los dígitos 0, 1, 2, 3 y 4.
- Los que comienzan con una letra entre la A y la H.

Quedan disponibles entonces para **class codes** asignables a errores de aplicación:

- Los que comienzan con dígitos 5, 6, 7, 8 o 9.
- Los que comienzan con una letra entre la I y la Z.

1.1.3. Excepciones específicas

Los subclass codes diferentes de **000** definen excepciones específicas dentro de cada categoría de excepción.

Algunos valores `SQLSTATE` de excepciones específicas comunes son:

SQLSTATE	Significado
22005	<code>error_in_assignment</code>
23502	<code>not_null_violation</code>
23505	<code>unique_violation</code>
23514	<code>check_violation</code>
25P01	<code>no_active_sql_transaction</code>

1.1.4. Subclass codes reservados y disponibles

Los siguientes subclass codes están reservados por ANSI:

- Los que comienzan con los dígitos 0, 1, 2, 3 y 4.
- Los que comienzan con una letra entre la A y la H.

Quedan disponibles entonces para subclass codes definidos por el usuario:

- Los que comienzan con dígitos 5, 6, 7, 8 o 9.
- Los que comienzan con una letra entre la I y la Z.



PostgreSQL define una categoría de excepción especial, **P0000**, llamada *PL/pgSQL Error*.
Algunos valores `SQLSTATE` de esta categoría son:

P0001 `raise_exception`
P0002 `no_data_found`



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

1.2. Errores SQL Server

1.2.1. 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.

1.2.2. 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.

2. 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.



Para el catch de excepciones PL/pgSQL define un bloque `BEGIN` / `END` con una sintaxis extendida:

```
DECLARE                                A
    noUsada VARCHAR(10);

BEGIN                                  B
    EXECUTE 'DROP TABLE noexiste' ; C
EXCEPTION                             D
    WHEN SQLSTATE '42P01' THEN        F
        RAISE NOTICE 'ERROR SQLERRM: % SQLSTATE: %', G
            SQLERRM, SQLSTATE;
        J
    WHEN OTHERS THEN                  H
        RAISE NOTICE 'Others: ERROR SQLERRM: % SQLSTATE: %', I
            SQLERRM, SQLSTATE;
END;
```

Para quienes conozcan el manejo de errores de Java o JavaScript, la idea es muy similar. Lo analizaremos por partes:

A. El bloque puede tener opcionalmente su propia sección `DECLARE`.

`BEGIN` (**B**) define lo que sería el "try". O sea, vamos a "intentar" ejecutar un código "peligroso" que sabemos a priori puede disparar una excepción (**C**). A continuación del `BEGIN` definimos dichas sentencias. En este caso, un intento de `DROP TABLE`.

La cláusula `EXCEPTION` (**D**) define lo que sería el "catch". Para cada error que queramos discriminar, se define una cláusula `WHEN` (**E**) con el código de error correspondiente (**F**). En este caso el error posee el `SQLSTATE` `'42P01'` (`undefined_table`). También podríamos haber escrito la cláusula como:

```
EXCEPTION
    WHEN undefined_table THEN
```

También podemos especificar categorías de errores como las que vimos en la Sección 1.1.3 *Categorías de errores*:

```
EXCEPTION
    WHEN SQLSTATE '02000' THEN
```



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 (H)** y **SQLSTATE (I)** nos proporcionan el mensaje y el código **SQLSTATE** respectivamente.

J es lo que se denomina un "catch all". Cuando ya no nos interesa seguir discriminando por errores puntuales, hacemos un catch de cualquier error no discriminado específicamente en una cláusula **WHEN**.

El siguiente sería el código completo de una función que hace uso de esta forma de catch:

```
CREATE FUNCTION test()
RETURNS VOID
LANGUAGE plpgsql
AS
$$
DECLARE
    res integer;

BEGIN

    -- Otras sentencias no peligrosas...

    BEGIN
        EXECUTE 'DROP TABLE noexiste' ;

    EXCEPTION
        WHEN undefined_table THEN
            RAISE NOTICE 'ERROR SQLERRM: % SQLSTATE: %',
                SQLERRM, SQLSTATE;
        WHEN OTHERS THEN
            RAISE NOTICE 'Others... ERROR SQLERRM: % SQLSTATE: %',
                SQLERRM, SQLSTATE;

    END;

    RETURN;
END
$;
```

El mensaje obtenido es:

```
NOTICE: ERROR SQLERRM: table "noexiste" does not exist SQLSTATE:
42P01
```

PostgreSQL



Información sobre la excepción

Desde la versión 9.2 PostgreSQL soporta la sentencia `GET STACKED DIAGNOSTICS` que permite brindar información más detallada sobre la excepción.

`GET STACKED DIAGNOSTICS` reemplazaría al `RAISE NOTICE` de nuestro ejemplo. La forma de catch de la excepción es idéntica a la presentada.



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 1.2.2) 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 el mismo error de tabla no existente:

```
(A) DECLARE @Error Integer

DROP TABLE noexiste (B)
SET @Error = @@Error

(C) IF @Error != 0
    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 (C)
    DROP TABLE noexiste
    -- ...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	ErrorLine
3701	Cannot drop the table 'noexiste', because it does not exist.	11	5	NULL	3

3. 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**.

3.1. 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',
      HINT = 'De de alta el producto antes de registrar el pedido';
```

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

La cláusula `USING` (B) nos permite especificar información adicional.

Lo más importante es el valor `SQLSTATE` del error que estamos disparando. Lo especificamos con la opción `ERRCODE` (C).

Como vimos en la Sección 1.1.4. *Subclass codes reservados y disponibles*, PostgreSQL define una categoría de excepción especial **P0000**, llamada *PL/pgSQL Error*. El valor `SQLSTATE` por omisión para `RAISE EXCEPTION` es justamente **P0001** (`raise_exception`)

Cuando estemos desarrollando una aplicación probablemente necesitemos definir un conjunto de errores. Como vimos en la Sección 1.1.1. *Categorías de excepciones*, tenemos disponibles como class codes asignables a errores de aplicación:

- Los que comienzan con dígitos 5, 6, 7, 8 o 9.
- Los que comienzan con una letra entre la I y la Z.

Por ejemplo, podríamos decidir que 'El producto % no existe' es el mensaje asociado al `SQLSTATE` **I0001**: class code **I0**, subclass code **001**.

Como regla general debemos evitar el subclass code `000` ya que los errores así definidos sólo pueden ser capturados capturando la categoría de excepción completa.

Como parte de la información adicional podemos añadir una sugerencia usando la opción `HINT` (D).

4. Errores y transacciones



Tratamiento de errores y transacciones.

Si no lo tratamos adecuadamente, cualquier error que ocurre en el ámbito de una función PL/pgSQL aborta la ejecución de la misma y **también la posible transacción en curso.**

El tratamiento de errores hace que nuestras funciones PL/pgSQL, ante una situación de excepción, finalicen de manera prolija y no de manera intempestiva.

Además, algunas operaciones sobre la base de datos pueden llevarse a cabo **aún cuando se haya producido un error dentro de la función.** Por ejemplo::

```
CREATE FUNCTION test ()
  RETURNS void
  AS
  BEGIN
    INSERT INTO publishers values('9988', 'Editora
                                Ingenieria Web', 'Texas', 'TA', 'USA'); (E)

    BEGIN

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

      DROP TABLE noexiste; (A)

    EXCEPTION
      WHEN SQLSTATE '42P01' THEN
        RAISE NOTICE 'ERROR SQLERRM: % SQLSTATE: %', (B)
          SQLERRM, SQLSTATE;

    END;
    RETURN; (C)
  END
```

(A) provoca una excepción, lo cual hace que el control pase al bloque `EXCEPTION`, donde es tratada (B). Luego el control pasa a (C) y la función termina.

La sentencia (D) es deshecha (`ROLLBACK`). Sin embargo, la sentencia (E) es hecha efectiva.



Si nuestra función no tuviese tratamiento de errores en absoluto. Por ejemplo:

```
CREATE FUNCTION test()
  RETURNS void
  AS
  DECLARE
  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;

    RETURN;
  END
```

...todos los cambios serían dejados sin efecto (ROLLBACK)

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 FUNCTION test()
  RETURNS void
  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;

    END;
    RETURN;
  END

BEGIN TRANSACTION;
  SELECT test();
COMMIT TRANSACTION;

NOTICE:      ERROR SQLERRM: table "noexiste" does not exist
SQLSTATE: 42P01
Query result with 1 row discarded.
```



Tratamiento de errores y transacciones

En T-SQL, un error no fatal (de Severity menor a 19) **no provoca un ROLLBACK automático**.

Por ejemplo:

```
BEGIN TRANSACTION
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; (A)

COMMIT TRANSACTION
```

(A) provoca un error de severidad 11:

```
Msg 3701, Level 11, State 5, Line 1
Cannot drop the table 'noexiste', because it does not exist or you do not
have permission.
```

...por lo tanto, **no se produce un ROLLBACK automático**, y las dos primeras sentencias son ejecutadas de manera exitosa **aún cuando la transacción completa no se llevó a cabo**.



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

El siguiente es el mismo ejemplo resuelto correctamente:

```
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

Como es de esperar, ninguna de las sentencias es hecha permanente en la base de datos.

Ejercicio 1.

Implemente una función PL/pgSQL que intente recuperar el precio de una publicación con una sentencia como la siguiente:

```
SELECT pepe INTO vPrice
FROM titles
WHERE title_id = 'BU1111';
```

Haga un catch de la excepción usando las variables `SQLSTATE` y `SQLERRM` para mostrar información del error.

Ejercicio 2

En la Guía de Trabajo Nro. 4, en la Sección 9.3 *Sentencias SELECT Single-Row*, dijimos que, en una sentencia PL/pgSQL como la siguiente:

```
SELECT price
INTO vPrice1
FROM titles
WHERE title_id = vTitle_id;
```

...si la sentencia `SELECT` recuperaba más de una tupla, las variables asumían el valor de la primer tupla recuperada, y si la sentencia `SELECT` no recuperaba ninguna tupla, las variables asumían el valor `NULL`.

PL/pgSQL proporciona una cláusula `STRICT` que –de ser utilizada– provoca que se disparen excepciones `SQLERROR P0003` y `P0002` respectivamente para estos casos:

En el siguiente ejemplo ejecutamos la misma *sentencia SELECT Single-Row* especificando `STRICT`:

```
SELECT price
INTO STRICT vPrice
FROM titles
WHERE title_id = vTitle_id;
```

Escriba una función PL/pgSQL que reciba como parámetro un valor de `type`, dispare una sentencia como la siguiente filtrando por el `type`:

```
SELECT price INTO vPrice FROM titles WHERE type = vType;
```

y provoque:

A. Una excepción `P0003` para el valor `type` `'business'`:

```
SELECT test ('business');
NOTICE: ERROR SQLERRM: query returned more than one row SQLSTATE: P0003
```

B. Una excepción **P0002** para el valor type 'noexiste':

```
SELECT test('noexiste');  
NOTICE: ERROR SQLERRM: query returned no rows SQLSTATE: P0002
```

Ejercicio 3.

Realice, en T-SQL, un ejercicio según las siguientes consignas:

A. Cree en pubs el stored procedure usp_GetErrorInfo:

```
CREATE PROCEDURE usp_GetErrorInfo  
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
```

B. Implemente un stored procedure llamado sp_BuscarPrecio3 que recupere el precio de una publicación cuyo title_id recibe como parámetro. La "firma" del SP debe ser la siguiente:

```
CREATE PROCEDURE sp_BuscarPrecio3  
(  
    @title_id VARCHAR(6),  
    @price FLOAT OUTPUT  
)
```

EL SP no debe disparar errores.

Si no se encuentran filas (utilice @@RowCount para verificarlo), el valor de retorno debe ser 70, cuyo significado para la aplicación es "No se encontró la publicación".

Si se encuentra una fila (utilice @@RowCount para verificarlo) pero el valor obtenido para el precio es NULL, el valor de retorno debe ser 71, cuyo significado para la aplicación es "La publicación existe pero su precio es NULL".

Si se encuentra una fila y el valor obtenido para precio es diferente de NULL, el valor de retorno debe ser 0, cuyo significado para la aplicación es "La publicación existe y posee precio".

Por ejemplo, tomemos tres publicaciones de titles:

La publicación con title_id 'MK5656' no existe (el SP debería retornar un valor 70):

```
SET NOCOUNT ON
DECLARE @price FLOAT, @retorno INTEGER
EXECUTE @retorno = sp_BuscarPrecio3 'MK5656', @price
PRINT @retorno; --70
```

La publicación con title_id 'MC3026' no posee precio (el SP debería retornar un valor 71):

```
SET NOCOUNT ON
DECLARE @price FLOAT, @retorno INTEGER
EXECUTE @retorno = sp_BuscarPrecio3 'MC3026', @price
PRINT @retorno; --71
```

La publicación con title_id 'BU1032' existe y tiene precio (el SP debería retornar un valor 0):

```
SET NOCOUNT ON
DECLARE @price FLOAT, @retorno INTEGER
EXECUTE @retorno = sp_BuscarPrecio3 'BU1032', @price
PRINT @retorno; --0
```

C. Implemente ahora un stored procedure para insertar una venta en la tabla Sales

La venta debe ser para el almacén (stor_id) '6380'

El número de factura (ord_num) debe ser '4545'

La fecha de la factura es la fecha actual (CURRENT_TIMESTAMP)

La cantidad pedida (qty) debe ser 2.

La "firma" del SP debe ser la siguiente:

```
CREATE PROCEDURE sp_InsertaSales
(
    @stor_id CHAR(4),
    @ord_num VARCHAR(20),
    @qty Smallint,
    @title_id VARCHAR(6)
)
```

El procedimiento debe:

- Buscar el precio de la publicación cuya columna `title_id` vale `@title_id` utilizando `sp_BuscarPrecio3`.

- Según el valor de retorno obtenido, debe seguir los siguientes cursos de acción:

- Si se obtuvo 70 (la publicación no existe), debe disparar el error de aplicación `'Publicación inexistente'` con una severidad 12 y contexto 1.

- Si se obtuvo 71 (la publicación no posee precio), debe disparar el error de aplicación `'La publicación no posee precio'` con una severidad 12 y contexto 1.

- **Solamente** si se obtuvo 0 como valor de retorno de `sp_BuscarPrecio3` (la publicación existe y tiene precio), se debe utilizar un bloque `BEGIN TRY/END TRY` para intentar hacer un `INSERT` en `Sales` para la venta:

```
INSERT INTO Sales
VALUES (@stor_id, @ord_num, CURRENT_TIMESTAMP, @qty, 'Net 40',
@title_id)1
```

y terminar retornando el valor 0 a la aplicación invocante.

En el bloque `BEGIN CATCH/END CATCH` utilice el SP `usp_GetErrorInfo` para obtener la condición del error y termine retornando el valor 72, que para la aplicación invocante significa `'Error de inserción en Sales'`.

¹ El valor de la columna `payterms` se especifica "hard coded".

Las siguientes son pruebas de ejecución para los tres casos planteados

```
----- Caso 1 -----  
-----No debe insertar -----  
  
SET NOCOUNT ON  
DECLARE @retorno INTEGER  
EXECUTE @retorno = sp_InsertaSales '6380', '4545', 2, 'MK5656'  
PRINT @retorno; --Publicacion inexistente  
  
----- Caso 2 -----  
-----No debe insertar -----  
  
SET NOCOUNT ON  
DECLARE @retorno INTEGER  
EXECUTE @retorno = sp_InsertaSales '6380', '4545', 2, 'MC3026'  
PRINT @retorno; --La publicacion no posee precio  
  
----- Caso 3 -----  
-----Debe insertar -----  
  
SET NOCOUNT ON  
DECLARE @retorno INTEGER  
EXECUTE @retorno = sp_InsertaSales '6380', '4545', 2, 'BU1032'  
PRINT @retorno; --0  
SELECT * FROM sales WHERE YEAR(ord_date) = '2015'
```