

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
Ejemplo de aplicación

Vamos a mejorar los ejercicios 3 y 4 de la Guía de Trabajo Nro. 4 aplicando tratamiento de errores.
Lo haremos sobre T-SQL.

Recordemos el Ejercicio 3

Trabajábamos sobre el esquema de tablas que habíamos creado en la Guía de Trabajo Nro. 2 para realizar ejercicios de manipulación de datos:

```
CREATE TABLE cliente
(
    codCli      int          NOT NULL,
    ape         varchar(30)  NOT NULL,
    nom         varchar(30)  NOT NULL,
    dir         varchar(40)  NOT NULL,
    codPost     char(9)      NULL DEFAULT 3000
)
```

cliente	
codCli	int
ape	varchar(30)
nom	varchar(30)
dir	varchar(40)
codPost	varchar(9)

```
CREATE TABLE productos
(
    codProd     int          NOT NULL,
    descr       varchar(30)  NOT NULL,
    precUnit    float        NOT NULL,
    stock       smallint     NOT NULL
)
```

productos	
codProd	int
descr	varchar(30)
precUnit	float
stock	smallint

```
CREATE TABLE proveed
(
    codProv     int          IDENTITY(1,1),
    razonSoc    varchar(30)  NOT NULL,
    dir         varchar(30)  NOT NULL
)
```

proveed	
codProv	int
razonSoc	varchar(30)
dir	varchar(30)

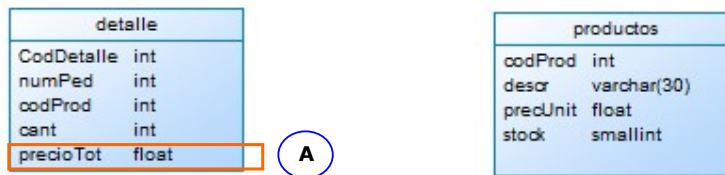
```
CREATE TABLE pedidos
(
    numPed      int          NOT NULL,
    fechPed     datetime     NOT NULL,
    codCli      int          NOT NULL
)
```

pedidos	
numPed	int
fechPed	datetime
codCli	int

```
CREATE TABLE detalle
(
    codDetalle  int          NOT NULL,
    numPed      int          NOT NULL,
    codProd     int          NOT NULL,
    cant        int          NOT NULL,
    precioTot   float        NULL
)
```

detalle	
CodDetalle	int
numPed	int
codProd	int
cant	int
precioTot	float

Específicamente trabajábamos con las tablas *Productos* y *Detalle*:



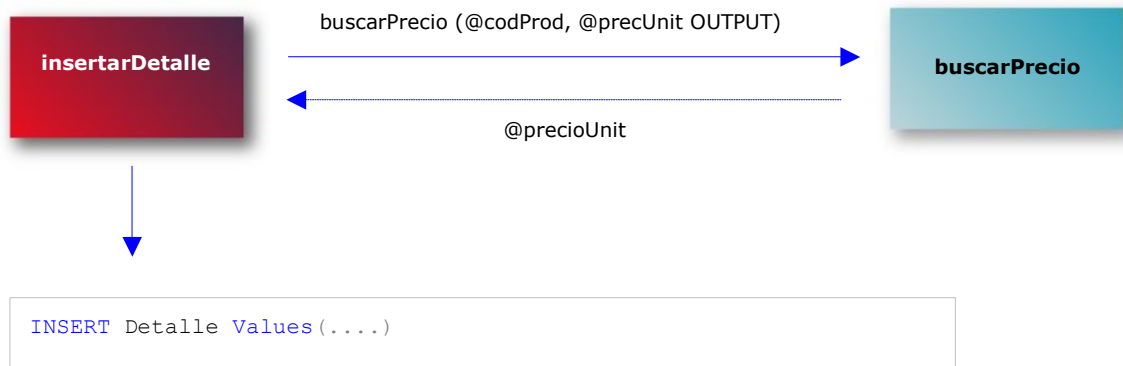
Habíamos dicho que en la columna *precioTot* de la tabla *Detalle* (A) se almacenaba el valor que se calculaba en función de la cantidad pedida de un producto (columna *cant*) y su precio unitario (columna *precUnit* en la tabla *productos*).

Se nos pedía crear un procedimiento almacenado (*insertarDetalle*). Este procedimiento recibía como parámetros código de detalle (*codDetalle*), número de Pedido (*numPed*), código de producto (*codProd*) y cantidad vendida del producto (*cant*), y debía insertar una nueva fila en la tabla *detalle*.

Para obtener el valor correspondiente a la columna *precioTot*, el procedimiento *insertarDetalle* debía invocar a un **procedimiento auxiliar** (*buscarPrecio*).

insertarDetalle invocaba *buscarPrecio* y le pasaba como parámetro el código de producto. *buscarPrecio* debía retornar el precio unitario correspondiente a tal producto.

El esquema de trabajo era el siguiente:



Habíamos decidido que *buscarPrecio* retornara el precio unitario como un OUTPUT parameter.

La siguiente es una solución posible:

```
ALTER PROCEDURE buscarPrecio
(
    @CodProd int,                -- Parametro de entrada
    @PrecUnit float OUTPUT      -- Parametro de salida
)
AS
    SELECT @PrecUnit = PrecUnit
        FROM Productos
        WHERE CodProd = @Codprod
    RETURN
```

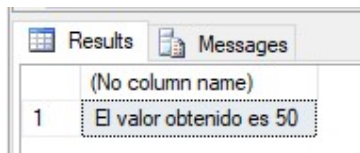
Habíamos insertado dos productos:

```
INSERT INTO PRODUCTOS VALUES (10, 'Articulo 1', $50, 20)
INSERT INTO PRODUCTOS VALUES (20, 'Articulo 2', $70, 40)
```

A todo procedimiento auxiliar debemos probarlo antes de ubicarlo en un contexto de mayor complejidad.

La siguiente es la prueba de nuestro procedimiento auxiliar:

```
DECLARE @PrecioObtenido FLOAT
EXECUTE buscarPrecio 10, @PrecioObtenido OUTPUT
SELECT 'El valor obtenido es ' + CONVERT(VARCHAR, @PrecioObtenido)
```



The screenshot shows a SQL Server Results window with two tabs: 'Results' and 'Messages'. The 'Results' tab is active, displaying a single row of data. The first column is labeled '(No column name)' and the second column contains the text 'El valor obtenido es 50'. The row is numbered '1' in the first column.

	(No column name)
1	El valor obtenido es 50

Luego de que comprobamos que el procedimiento auxiliar funciona, desarrollamos el procedimiento `insertarDetalle`. Una solución posible es la siguiente:

```
CREATE PROCEDURE insertarDetalle
(
    @CodDetalle Int,      -- Parametro de entrada
    @NumPed Int,         -- Parametro de entrada
    @CodProd int,        -- Parametro de entrada
    @Cant Int           -- Parametro de entrada
)
AS
    DECLARE @PrecioObtenido FLOAT --Parametro de salida del procedimiento auxiliar
    EXECUTE buscarprecio @CodProd, @PrecioObtenido OUTPUT

    INSERT Detalle Values(@CodDetalle, @NumPed, @CodProd, @Cant,
                          @Cant * @PrecioObtenido)

    If @@RowCount = 1
        PRINT 'Se inserto una fila'
    RETURN
```

Probamos `insertarDetalle` para 2 unidades del producto 10:

```
insertarDetalle 1540, 120, 10, 2
```

Results		Messages			
	codDetalle	numPed	codProd	cant	precioTot
1	1540	120	10	2	100

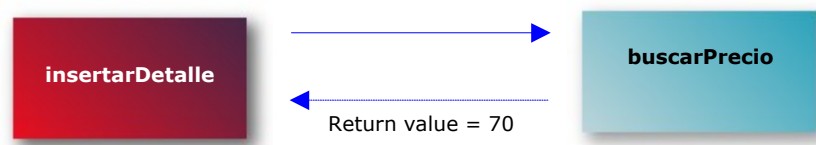
Esta solución tenía el problema de que podía darse la situación de que el producto recibido como parámetro en *insertarDetalle* no existiera. O que existiera pero no tuviese precio definido.

En el Ejercicio 4 de la Guía 4 mejorábamos el procedimiento *buscarPrecio* para que evaluara estas situaciones:

- Si sucediese que el código de producto (*codProd*) que recibía el procedure no existiese en la tabla de productos, debíamos indicar esta situación con un mensaje y evitar que se ejecute la sentencia *INSERT*.
- De manera análoga, si sucediese que el producto no tuviese definido precio, debíamos también indicar esta situación con un mensaje y evitar también que se ejecute la sentencia *INSERT*.

Sugeríamos que la nueva versión de *buscarPrecio* retornara diferentes **return values personalizados** para indicar al procedure invocante lo que estaba sucediendo.

Por ejemplo:



Si Return value <> 70 y
Return value <> 71

```
INSERT Detalle Values(....)
```

Return values definidos por el desarrollador

RETURN **70** significa: El producto no existe.
RETURN **71** significa: El producto no tiene precio definido.

La siguiente podría ser la versión mejorada del procedimiento `buscarPrecio`:

```
CREATE PROCEDURE buscarPrecioV2
(
    @CodProd int,          -- Parametro de entrada
    @PrecUnit float OUTPUT -- Parametro de salida
)
AS
    SELECT @PrecUnit = PrecUnit
    FROM Productos
    WHERE CodProd = @Codprod

    IF @@RowCount = 0
        RETURN 70          -- No se encontro el producto
    -- END IF

    IF @PrecUnit IS NULL
        RETURN 71          -- El producto existe pero su precio es NULL
    -- END IF

    RETURN 0               -- El producto existe y su precio no es NULL
```

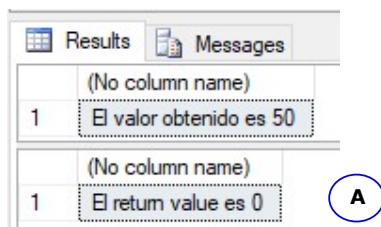
Aquí lo probamos para el producto 10:

```
DECLARE @PrecioObtenido FLOAT --Parametro de salida del inner procedure
DECLARE @StatusRetorno Int

EXECUTE @StatusRetorno = buscarPrecioV2 10, @PrecioObtenido OUTPUT

SELECT 'El valor obtenido es ' + CONVERT(VARCHAR, @PrecioObtenido)
SELECT 'El return value es ' + CONVERT(VARCHAR, @StatusRetorno)
```

Obtenemos:



	(No column name)
1	El valor obtenido es 50
1	El return value es 0

Como, vemos, el RETURN VALUE es cero (**A**), la cual implica que el producto existe y que posee precio definido.

Lo probamos ahora para un producto inexistente:

```
DECLARE @PrecioObtenido FLOAT    --Parametro de salida del inner procedure
DECLARE @StatusRetorno Int

EXECUTE @StatusRetorno = buscarPrecioV2 32, @PrecioObtenido OUTPUT

SELECT 'El valor obtenido es ' + CONVERT(VARCHAR, @PrecioObtenido)
SELECT 'El return value es ' + CONVERT(VARCHAR, @StatusRetorno)
```

Results		Messages
(No column name)		
1	NULL	
(No column name)		
1	El return value es 70	B

Como, vemos, ahora el RETURN VALUE es 70 (**B**), la cual implica que el producto no existe.

La siguiente podría ser la versión de *insertarDetalle* adaptada para este esquema de trabajo mejorado:

```

ALTER PROCEDURE insertarDetalle2
(
    @CodDetalle Int,      -- Parametro de entrada
    @NumPed Int,         -- Parametro de entrada
    @CodProd int,        -- Parametro de entrada
    @Cant Int            -- Parametro de entrada
)
    AS
    DECLARE @PrecioObtenido FLOAT  --Parametro de salida del procedimiento auxiliar

    DECLARE @StatusRetorno Int

    EXECUTE @StatusRetorno = buscarPrecioV2 @CodProd, @PrecioObtenido OUTPUT

    IF @StatusRetorno != 0
    BEGIN
        IF @StatusRetorno = 70
        BEGIN
            PRINT 'Codigo de producto inexistente'
            RETURN A
        END
    ELSE
        IF @StatusRetorno = 71
        BEGIN
            PRINT 'El producto no posee precio'
            RETURN B
        END
        -- END IF
        -- END IF
    END
    -- END IF

    INSERT Detalle Values (@CodDetalle, @NumPed, @CodProd, @Cant, C
                           @Cant * @PrecioObtenido)

    If @@RowCount = 1
        PRINT 'Se inserto una fila'
    RETURN

```

Bajo este esquema de trabajo la inserción de la fila (C) SOLO ocurre si el procedimiento auxiliar retorna un valor cero, ya que ante otros valores de RETURN VALUE tenemos sentencias RETURN (A) y (B) que finalizan la ejecución del procedimiento sin llevar a cabo el [INSERT](#).



Como vimos en "4.1. Return value" en la "Guía de trabajo Nro. 4 - Parte 2", una finalización normal (exitosa) de un stored procedure retorna un status de 0. En *buscarPrecioV2* estamos retornando un valor cero de manera explícita.

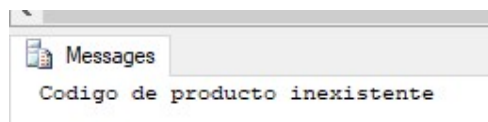
En (A) y (B) estamos retornando cero, lo cual no sería del todo correcto. Podríamos retornar un valor diferente. Sin embargo, el objetivo se cumple, que es que no se lleve a cabo el [INSERT](#).

Finalmente probamos `insertarDetalle2` para dos unidades de un código de producto inexistente:

```
CodDetalle  1540
NumPed      120
CodProd     99
Cant        2
```

```
insertarDetalle2 1540, 120, 99, 2
```

...obtenemos:



... y la inserción no debe llevarse a cabo:

```
SELECT * FROM detalle
```

Results		Messages			
	codDetalle	numPed	codProd	cant	precioTot
1	1540	120	10	2	100

Ahora volvemos a mejorar nuevamente al procedimiento *insertarDetalle* agregándole tratamiento de errores:

```

ALTER PROCEDURE insertarDetalle3
(
    @CodDetalle Int,      -- Parametro de entrada
    @NumPed Int,          -- Parametro de entrada
    @CodProd int,         -- Parametro de entrada
    @Cant Int            -- Parametro de entrada
)
    AS
    DECLARE @PrecioObtenido FLOAT  --Parametro de salida del inner procedure

    DECLARE @StatusRetorno Int, @error INTEGER
    -- SET NOCOUNT ON

    EXECUTE @StatusRetorno = buscarPrecioV2 @CodProd, @PrecioObtenido OUTPUT

    -- @StatusRetorno 0 significa que la publicacion existe y su precio no es NULL
A SET @error = 0;

    IF @StatusRetorno != 0
    BEGIN
        IF @StatusRetorno = 70
        BEGIN
            RAISERROR ('Publicación inexistente', 12, 1);
            SET @error = @@error
C
        END
    ELSE
        IF @StatusRetorno = 71
        BEGIN
            RAISERROR ('La publicación no posee precio', 12, 1);
            SET @error = @@error
D
        END
        -- END IF
    -- END IF
    END
    -- END IF

```

```
--PRINT ' @Error vale ' + CONVERT(VARCHAR(10), @error)
```

```
E IF (@error = 0)
```

```
    BEGIN
```

```
        BEGIN TRY
```

```
            INSERT Detalle Values(@CodDetalle, @NumPed, @CodProd, @Cant,  
                                   @Cant * @PrecioObtenido)
```

```
            PRINT 'Se insertó una fila'
```

```
            RETURN 0
```

```
        END TRY
```

```
I BEGIN CATCH
```

```
    EXECUTE usp_GetErrorInfo
```

```
    RETURN 72
```

```
END CATCH
```

```
END
```

```
-- END IF
```

```
If @@RowCount = 1
```

```
    PRINT 'Se inserto una fila'
```

```
RETURN
```

Lo primero que hacemos (**A**) es preparar una variable local para guardar el valor de la variable global `@@error`.



Ver "Catch de errores usando `@@error`" en la "Guía de trabajo Nro. 6 - Parte 1".

¿Por qué usamos una variable local?. Porque el valor de la variable global `@@error` cambia todo el tiempo y necesitamos sacar una "fotografía" del mismo en un momento determinado.

Seguimos evaluando, como antes, el RETURN VALUE del procedimiento `buscarPrecio`. Eso no ha cambiado en absoluto. Solo que ahora, realizamos acciones diferentes.

En el caso de que la publicación no exista (**B**) disparamos **nosotros mismos** un error usando `RAISERROR`.



Ver "5.1. Disparar errores de aplicación " en la "Guía de trabajo Nro. 6 - Parte 1".

Nosotros también elegimos la severidad del error. En este caso le damos una severidad de 12.



Ver Severity en "2.2. Componentes de un error" en la "Guía de trabajo Nro. 6 - Parte 1".

Nuestro propio RAISE provoca que cambie el valor de `@@error`. En (**C**) capturamos este valor antes de perderlo.

Un proceso idéntico hacemos en (**D**), solo que con otro mensaje de error.

No nos importa mucho cuanto vale `@@error`, solo nos interesa que un valor distinto de cero **indica una condición de error**.

Eso es lo que evaluamos en (**E**): solo vamos a intentar el `INSERT` SI NO EXISTEN ERRORES PREVIOS.

Y justamente estamos diciendo intentar, porque el `INSERT` también puede fallar, por otras circunstancias diferentes.

Así que aquí encerramos esta sentencia "peligrosa" en un bloque `TRY` (**G**)

Si el `INSERT` tiene éxito, mostramos un mensaje y salimos definitivamente del procedure retornando un cero (**H**)

Si el `INSERT` tiene algún problema, el control va al bloque `CATCH (I)`.

Allí `(J)` mostramos la información del error y retornamos del procedure indicando que algo salió mal. Lo hacemos especificando un valor diferente de cero, que pueda orientar sobre lo que está sucediendo a la aplicación invocante. (una aplicación Java o .NET, por ejemplo).

Ahora bien, un detalle más que importante.

¿Qué sucedería si no hiciera la comprobación de los errores previos en `(E)`?

Hagamos un ensayo:


```

CREATE PROCEDURE insertarDetalle3B
(
    @CodDetalle Int,      -- Parametro de entrada
    @NumPed Int,         -- Parametro de entrada
    @CodProd int,        -- Parametro de entrada
    @Cant Int            -- Parametro de entrada
)
    AS
    DECLARE @PrecioObtenido FLOAT  --Parametro de salida del inner procedure

    DECLARE @StatusRetorno Int, @error INTEGER
    -- SET NOCOUNT ON

    EXECUTE @StatusRetorno = buscarPrecioV2 @CodProd, @PrecioObtenido OUTPUT

    -- @StatusRetorno 0 significa que la publicacion existe y su precio no es NULL
    SET @error = 0;

    IF @StatusRetorno != 0
    BEGIN
        IF @StatusRetorno = 70
        BEGIN
            RAISERROR ('Publicación inexistente', 12, 1);
            SET @error = @@error
        END
    ELSE
        IF @StatusRetorno = 71
        BEGIN
            RAISERROR ('La publicación no posee precio', 12, 1);
            SET @error = @@error
        END
        -- END IF
    -- END IF
    END
    -- END IF

```

B

```
PRINT ' @Error vale ' + CONVERT(VARCHAR(10), @error)
```

C

A

```
--IF (@error = 0)
BEGIN
    BEGIN TRY
        INSERT Detalle Values (@CodDetalle, @NumPed, @CodProd, @Cant,
                                @Cant * @PrecioObtenido)
        PRINT 'Se insertó una fila'
        RETURN 0
    END TRY

    BEGIN CATCH
        EXECUTE usp_GetErrorInfo
        RETURN 72
    END CATCH
END

-- END IF
--PRINT 'Código posterior a evaluación del error'

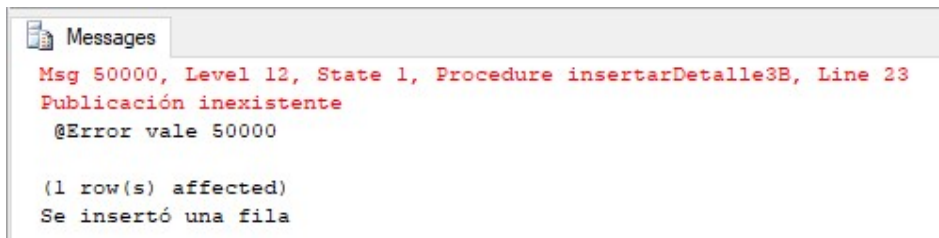
If @@RowCount = 1
    PRINT 'Se inserto una fila'
RETURN
```

Esta vez no realizamos la comprobación de errores previos **(A)**.

Ejecutamos el procedure para dos unidades de un artículo inexistente:

```
insertarDetalle3B 1540, 120, 99, 2
```

Obtenemos:



Si nos fijamos, la fila se ha insertado con un valor `NULL` en la columna `precioTot`:

Results		Messages			
	codDetalle	numPed	codProd	cant	precioTot
1	1540	120	99	2	NULL

Analicemos lo que ha sucedido:

`RAISERROR` dispara el error en **(B)**.

Mostramos `@@ERROR` en **(C)**. Vale 50000

...pero aún ante una condición de error, el programa continúa su ejecución, y el `INSERT` se lleva a cabo.

Esto es lo que debemos evitar que suceda.