6. PROGRAMACIÓN CON TRANSACT-SQL

6.1. Introducción.

Transact-SQL amplía el estándar SQL con una serie de extensiones, que resultan de gran utilidad para la programación. Antes de que estas extensiones estuvieran disponibles, las solicitudes a la base de datos siempre eran instrucciones sencillas y cualquier lógica condicional tenía que realizarla la aplicación llamante.

6.2. Declaración y asignación de variables.

En Transact-SQL podemos declarar **variables**. Se trata de *variables locales,* con un ámbito y visibilidad sólo dentro del proceso por lotes o procedimiento almacenado en que se han declarado (conceptos que veremos posteriormente). Las variables deben comenzar por el carácter @ .

Transact-SQL **no** tiene variables globales. La documentación antigua hace referencia a ciertas **funciones** del sistema sin parámetros, como *variables globales,* quizá porque se designan mediante @@ , de forma similar a la designación para las variables locales.

Las variables se declaran al principio de un proceso por lotes o procedimiento almacenado, mediante la palabra reservada DECLARE, seguida del nombre de la variable y el tipo.

Es posible **asignar** valores a variables con la instrucción **SELECT** o la instrucción **SET**, utilizando el operador de asignación (=). Los valores asignados mediante SET pueden ser constantes, otras variables o expresiones. En el siguiente ejemplo declaramos una variable, le asignamos un valor mediante la instrucción SET, y después utilizamos esa variable en una cláusula WHERE:

```
DECLARE @limite money
SET @limite = 10
SELECT * FROM articulos WHERE precio <= @limite
```

Podíamos haber escrito el mismo código utilizando la instrucción SELECT en lugar de la instrucción SET. Sin embargo, habitualmente se utiliza SELECT cuando los valores que se van a asignar se encuentran en una columna de una tabla. Una instrucción SELECT utilizada para asignar valores a una o varias variables se denomina *SELECT de asignación.*

No es posible combinar la funcionalidad del *SELECT de asignación* y de un *SELECT «normal»* en la misma instrucción. Es decir, si se utiliza una instrucción SELECT para asignar valores a variables, no puede devolver también valores al cliente como un conjunto de resultados. En el siguiente ejemplo declaramos dos variables, les asignamos valores desde la tabla *precios,* y después seleccionamos sus valores como conjunto de resultados.

```
DECLARE @precio_min smallmoney, @precio_max smallmoney
SELECT  @precio_min = MIN(precio), --Variables asignadas
        @precio_max = MAX(precio)  --SELECT de asignación
           FROM precios
SELECT  @precio_min, @precio_max --Variables devueltas SELECT «normal»
```

Una única instrucción SELECT puede asignar varias variables, sin embargo, hay que utilizar una instrucción SET independiente para asignar cada variable. Por ejemplo, la siguiente instrucción SET devuelve un error:

```
SET @precio_min=0, @precio_max=100
```

Si se utiliza la expresión SELECT de asignación habrá que tener en cuenta que el valor asignado a la variable sea único. De lo contrario, se asignará uno de los valores devueltos del conjunto de valores, pero es posible que el resultado no sea el deseado.

Ejemplo: Supongamos que existen los siguientes valores de concepto en la tabla artículos:

CodArt	Concepto
0001	Ratón óptico Genius
0002	Teclado Acer
0003	Monitor Sony 17”

La siguiente asignación se ejecutará sin errores, pero probablemente su resultado no será el deseado:

```
DECLARE @Concepto varchar(30)
SELECT @Concepto=Concepto FROM articulos
SELECT @Concepto
```

El valor devuelto por la instrucción SELECT normal en este caso es: Monitor Sony 17”. Pero en general el resultado será impredecible.

Otra cosa a tener en cuenta es que si el SELECT de asignación no devuelve ninguna fila no se asignará nada a la variable, y por tanto, ésta mantendrá el valor que tuviera antes de ejecutar la instrucción SELECT de asignación.

Por ejemplo: Al ejecutar el siguiente código se obtiene el nombre del autor de apellido *Cervantes* dos veces, porque el autor de apellido *Kafka* no existe en la base de datos y por tanto la sentencia de asignación correspondiente mantiene el valor anterior.

```
DECLARE @Nombre varchar(20)
SELECT @Nombre = Autor
FROM libros
WHERE Autor like '%Cervantes%'
SELECT @Nombre  -- devolver el nombre como un resultado
```

```
SELECT @Nombre = Autor
FROM libros
WHERE Autor like '%Kafka%'
SELECT @Nombre  -- devolver el nombre como un resultado
```

6.3. Control de flujo.

Como cualquier lenguaje de programación, Transact-SQL posee instrucciones de control de flujo del programa. Destacamos entre las más importantes:

BEGIN ... END	Define un bloque de instrucciones. Típicamente, BEGIN suele ir inmediatamente después de IF, ELSE o WHILE (de lo contrario sólo se ejecutaría la siguiente instrucción). Para los programadores en C BEGIN...END es similar a utilizar un bloque {..}.
IF ... ELSE	Define una instrucción condicional.
WHILE	Construcción de bucle básica para SQL Server. Repite una instrucción (o bloque) mientras una condición especifica sea cierta.
GOTO Etiqueta	Continúa el proceso en la instrucción que sigue a la etiqueta definida. La etiqueta se define: *NombreEtiqueta:*

Para más información consultar: **Libros en Pantalla** – Búsqueda - *Lenguaje de control de flujo*
En cuanto a funciones ver: **Libros en Pantalla** – Búsqueda - *Funciones (T-SQL)*

6.4. Procesos por lotes.

Un proceso por lotes (batch) es uno o varios comandos de SQL Server que se envían y se ejecutan juntos. Como cada proceso por lotes enviado desde el cliente hasta el servidor necesita ciertas comunicaciones de protocolo entre los dos, enviar un proceso por lotes en lugar de comandos independientes puede resultar más eficiente.

A continuación, se presenta un sencillo proceso por lotes emitido desde el *Analizador de consultas* (cliente) hacia SQL Server (servidor). Aunque se realizan tres operaciones no relacionadas, es posible empaquetarlas en un único proceso por lotes.

```
INSERT socios VALUES (etc.)
SELECT * FROM socios
UPDATE libros SET Autor = (etc.)
GO  -- comando que indica que todo lo anterior se envíe
    -- como un proceso por lotes.
```

Todas las instrucciones dentro de un único proceso por lotes se analizan como una unidad. Esto significa que, si existe un error de sintaxis en alguna de las instrucciones, no se ejecutará ninguna de ellas. Por ejemplo, podemos intentar ejecutar desde el *Analizador de consultas* el siguiente proceso por lotes.

```
SELECT * FROM libros
SELECT * FOM socios
GO
```

```
Al ejecutar este proceso por lotes, se recibirá el siguiente error:

   Servidor: mensaje 170, nivel 15, estado 1
   Línea 2: sintaxis incorrecta cerca de 'fom'.
```

Aunque la primera instrucción SELECT es perfectamente legal, debido al error en la segunda instrucción no se devuelven datos. Si lo separáramos en dos procesos por lotes distintos mediante dos GO, el primero devolvería un conjunto de resultados, y el segundo devolvería un error.

Supongamos ahora que en lugar de escribir mal la palabra clave FROM como FOM, hubiéramos escrito mal el nombre de la segunda tabla:

```
SELECT * FROM libros
SELECT * FROM scios
GO
```

SQL Server 2000 utiliza **resolución de nombres aplazada**, en la cual los nombres de los objetos no se resuelven hasta el tiempo de ejecución. En este ejemplo se devuelven los datos de la primera operación SELECT, seguidos por el mensaje de error siguiente:

```
Servidor:mensaje 208, nivel 16, estado 1
El nombre de objeto 'scios' no es válido
```

No obstante, si hubiéramos escrito mal el nombre del objeto en la primera instrucción, la ejecución del proceso por lotes se habría detenido al encontrar el error. Ninguna de las consultas válidas siguientes habría devuelto datos.

Normalmente, al utilizar el *Analizador de consultas*, todo lo que contiene la ventana de consulta se considera un único proceso por lotes y se envía a SQL Server para su análisis, compilación y ejecución al pulsar el botón EJECUTAR verde (o la tecla función F5).

Sin embargo, el *Analizador de consultas* proporciona dos métodos alternativos para lo mismo. En primer lugar, permite marcar simplemente una sección de código en la ventana de consulta, para que al pulsar el botón EJECUTAR (o F5) sólo se envíe el texto marcado como un proceso por lotes a SQL Server. De forma alternativa, se puede incluir la palabra clave **GO** entre sus consultas.

GO no es un comando SQL. Es la señal de final de proceso por lotes que sólo entienden ciertas herramientas cliente (por ejemplo el *Analizador de Consultas*). La interpreta el cliente para indicar que todo, desde el último **GO**, debe enviarse al servidor para su ejecución como un proceso por lotes. El servidor (SQL Server) nunca ve el comando **GO**, y no tiene ni idea de lo que significa.

Un conjunto de procesos por lotes que se ejecutan con frecuencia juntos suele denominarse una secuencia de instrucciones (script). La mayor parte de las herramientas cliente proporcionan un mecanismo para cargar una secuencia de instrucciones guardadas en un archivo de texto y para ejecutarla. En el Analizador de consultas, podemos utilizar el comando Archivo/Abrir para cargar una secuencia de instrucciones.

6.5. Transacciones

Al igual que un proceso por lotes, una transacción declarada por el usuario suele constar de varios comandos SQL que leen y actualizan la base de datos. No obstante, un proceso por lotes es básicamente un concepto en el lado del cliente, que controla cuántas instrucciones se envían a SQL Server para procesar de una vez. Por el contrario, una transacción es un concepto en el lado del servidor que se encarga de determinar cuánto trabajo debe realizar SQL Server antes de considerar confirmados los cambios. Una transacción de varias instrucciones no realiza cambios permanentes en una base de datos hasta que se emite una instrucción COMMIT TRANSACTION. Además, una transacción puede deshacer sus cambios cuando se emite una instrucción ROLLBACK TRANSACTION.

Veamos ahora una transacción sencilla. Las instrucciones BEGIN TRAN y COMMIT TRAN hacen que los comandos incluidos entre ellas se ejecuten como una unidad. Es decir, o se ejecutan todos, o no se ejecuta ninguno. Incluso si se produce un fallo del hardware o un fallo general del sistema.

```
BEGIN TRAN
INSERT prestamos VALUES( , , etc.)
UPDATE ejemplares SET disponible = 0 where signatura = etc.
COMMIT TRAN
GO
```

SQL Server puede funcionar con tres **modos de transacciones**:

- **Transacciones de confirmación automática:** Cada instrucción individual es una transacción. De manera predeterminada, SQL Server trata cada instrucción, tanto si se envía de manera individual como si forma parte de un proceso por lotes, como independiente Toda instrucción de Transact-SQL se confirma o se deshace cuando termina. Si una instrucción termina correctamente, se confirma; si encuentra un error, se deshace. Una conexión de SQL Server funcionará en modo de confirmación automática hasta que la instrucción BEGIN TRAN inicie una transacción **explícita** o se active el modo de transacciones **implícitas**.
- **Transacciones explícitas:** Cada transacción se inicia explícitamente con la instrucción BEGIN TRAN y se termina explícitamente con una instrucción COMMIT TRAN o ROLLBACK TRAN.
- **Transacciones implícitas:** Se inicia implícitamente una nueva transacción cuando se ha completado la anterior, pero cada transacción se completa explícitamente con una instrucción COMMIT o ROLLBACK. Este modo se activa mediante SET IMPLICIT_TRANSACTIONS ON. Si están activadas las transacciones implícitas, todas las instrucciones, desde la última confirmación, se consideran parte de una transacción, y no se confirman hasta que se emita una instrucción COMMIT TRAN. O bien se deshacen con ROOLBACK.

Lógicamente, no tiene sentido hablar de confirmación de una instrucción SELECT. Por tanto, cuando hablamos de transacciones estamos hablando de instrucciones de modificación de datos (INSERT, UPDATE y DELETE).

Comprobación de errores en las transacciones

Uno de los errores más comunes que se cometen utilizando SQL Server es asumir que *cualquier* error dentro de una transacción provocará que la transacción se anule automáticamente. Si se encuentra un error no fatal y no se toma ninguna acción, el procesamiento pasará a la siguiente instrucción. Sólo los errores fatales provocan la cancelación automática del proceso por lotes y de la transacción que esté inmersa en este proceso por lotes.

Veamos esto a través del siguiente ejemplo:

```
create table a(a char(1) primary key)
create table b(b char(1) references a)
create table c(c char(1))
go

begin transaction
insert c values ('X')
insert b values ('X') -- falla la referencia
commit transaction
go

select * from c -- ¡Devuelve 'X'!
```

Aunque parezca que se ha violado la atomicidad de la transacción, las instrucciones se están ejecutando como era de esperar; el error está en la forma en que el usuario ha escrito la transacción. La transacción no ha comprobado los errores para cada instrucción y se confirma incondicionalmente al final. Por tanto, aunque falle una instrucción con un error de ejecución no fatal (en este caso se produce una violación de clave ajena), la ejecución continúa con la instrucción siguiente. Al final se ejecuta COMMIT, así que se confirman todas las instrucciones sin errores. Eso es exactamente lo que se ha dicho que se haga. Si queremos anular una transacción, cuando se produce cualquier error, debemos comprobar @@ERROR o utilizar SET XACT_ABORT.

Mediante el primer método la secuencia de instrucciones quedaría de la siguiente manera, (donde @@ERROR igual a 0 significa que no se ha producido ningún error):

```
begin transaction
insert c values ('X')
if (@@error <> 0) goto Hay_error
insert b values ('X') -- falla la referencia
if (@@error <> 0) goto Hay_error
commit transaction
Hay_error:
rollback transaction
go
select * from c -- ¡No devuelve 'X'!, ya que no se confirma
```

Otra forma de controlar la atomicidad de la transacción ante los errores será mediante la opción SET XACT_ABORT. Si se activa ésta, cualquier error, no sólo un error fatal (equivalente a comprobar @@ERROR <> 0 después de cada instrucción) terminará el proceso por lotes. Este es otro método de asegurar que no se confirme *nada* si se produce *cualquier* error:

```
set xact_abort on
begin transaction
insert c values ('X')
insert b values ('X') -- falla la referencia
commit transaction
go

select * from c -- ¡No devuelve 'X'
                --  puesto que ante el error el proceso por
                --  lotes se ha interrumpido,
                --  la transacción no se ha confirmado
                --  y se trata ésta de forma atómica!
```

Ten en cuenta que la opción XACT_ABORT cancela inmediatamente el proceso por lotes actual, no simplemente la transacción. Por ejemplo, sí hay dos transacciones dentro de un proceso por lotes, la segunda transacción nunca se ejecutará porque el proceso por lotes se cancelará antes de que dicha transacción tenga la oportunidad de ejecutarse.

6.6. Procedimientos Almacenados

Los procedimientos almacenados son un conjunto de sentencias Transact-SQL que se hallan compilados y almacenados en la caché del servidor para que su ejecución sea más rápida. Su estructura es similar a la de una función de cualquier lenguaje estructurado, también aceptan parámetros y devuelven valores de retorno.

Hay procedimientos almacenados que vienen definidos por el propio sistema (procedimientos almacenados de sistema) y otros que los define el usuario (procedimientos almacenados de usuario).

La sentencia Transact-SQL que permite crear procedimientos almacenados es CREATE PROCEDURE. Una sintaxis reducida es la siguiente:

```
CREATE PROCEDURE  NombreProcedimiento
[ {@parámetro tipoDatos }[= predeterminado] [OUTPUT]  ]
[WITH ENCRIPTION]
AS
 instrucciónSQL [...n]

RETURN n
```

Cuando se utiliza la opción OUTPUT en un parámetro, se estará pasando el parámetro por referencia, las modificaciones que se produzcan en el interior del procedimiento en los valores

que se han pasado como argumentos permanecerán incluso una vez finalizada la ejecución de éste.

Si no se utiliza la cláusula OUTPUT estaremos realizando el paso de parámetro por valor, lo que significa que al pasar el parámetro a un procedimiento almacenado se crea una copia de dicho argumento, que será utilizada localmente por el procedimiento. Las modificaciones que se produzcan en el interior del procedimiento almacenado no se reflejarán en los valores de los mismos una vez que el procedimiento ha finalizado su ejecución.

ENCRYPTION indica que SQL Server codifica la entrada de la tabla **syscomments** que contiene el texto de la instrucción CREATE PROCEDURE. El uso de ENCRYPTION impide que el procedimiento se publique como parte de la duplicación de SQL Server.

Una vez que se ha creado un procedimiento almacenado, éste se encontrará en disposición de ser ejecutado. Si en la primera línea de un proceso por lotes, aparece el nombre del procedimiento, SQL Server lo ejecutará. En el resto de situaciones deberemos utilizar la sentencia EXECUTE. Esta instrucción tiene la siguiente sintaxis reducida:

```
[[EXEC[UTE]]
        [@valor_de_retorno=] NombreProcedimiento
        [[@parámetro =] {valor | @variable [OUTPUT] |
[DEFAULT]] [,…n]
```

Veremos ahora algunos ejemplos de creación de procedimientos almacenados y su ejecución.

Ejemplo 1: Procedimiento llamado ObtenerTitulosDeAutor que toma un parámetro, parte del nombre de un autor, y devuelve los títulos de los libros de ese autor:

```
CREATE PROC ObtenerTitulosDeAutor @autor nvarchar(20)
AS
 SELECT titulo
 FROM libros
 WHERE autor like '%'+@autor+'%'
```

Este procedimiento puede ejecutarse posteriormente con una sintaxis como la siguiente:

```
ObtenerTitulosDeAutor 'Cervantes'
o
EXEC ObtenerTitulosDeAutor 'Cervantes'
o
EXEC ObtenerTitulosDeAutor @autor='Cervantes'
```

Ejemplo 2: Procedimiento que selecciona todas las filas de la tabla socios y todas las filas de la tabla libros. También define un parámetro de salida (paso de parámetro por referencia) para cada tabla basado en @@ROWCOUNT (devuelve el número de filas afectadas por la última instrucción). Este valor está disponible posteriormente para el proceso por lotes llamante comprobando las variables pasadas como parámetros de salida.

```
CREATE PROC CuentaSociosyLibros
 @CuentaSocios int OUTPUT,
 @CuentaLibros int OUTPUT
AS
 SELECT * FROM Socios
 SELECT @CuentaSocios=@@ROWCOUNT
 SELECT * FROM Libros
 SELECT @CuentaLibros=@@ROWCOUNT
```

Después, el procedimiento se ejecutaría de la siguiente forma:

```
DECLARE @CuentaLib int, @CuentaSoc int
EXEC CuentaSociosyLibros @CuentaLib OUTPUT, @CuentaSoc OUTPUT
select Total_Libros=@CuentaLib, @CuentaSoc as Total_Socios
```

Las variables declaradas en un procedimiento almacenado, siempre son locales; así que podría haber utilizado los mismos nombres, tanto para las variables del procedimiento como para las pasadas por el proceso por lotes como parámetros de salida. De hecho, esa es probablemente la forma más habitual de invocarlas. No obstante, hacemos hincapié en que las variables no tienen por qué tener el mismo nombre. Incluso con el mismo nombre son, de hecho, variables distintas porque su ámbito es diferente.

Este procedimiento devolvería todas las filas de ambas tablas. Además, las variables `@CuentaLib` y `@CuentaSoc` mantendrían los totales de las filas de las tablas *Libros* y *Socios* respectivamente. Observa los distintos modos utilizados para nombrar las columnas de los resultados `(Total_Libros y Total_Socios)`.

Ejercicio: Realiza un procedimiento almacenado en la base de datos *Biblio*, que reciba por parámetro un código de socio, una signatura y un número de ejemplar, y realice el préstamo correspondiente, siempre que el libro solicitado esté disponible, y en tal caso, lo ponga a no-disponible. Todas las acciones deberán tratarse de forma atómica. Es decir, se deberá evitar que se grabe el préstamo y no se marque como no disponible, o viceversa. Además habrá que tener en cuenta los errores de integridad referencial al insertar el préstamo (porque no exista el código de socio).

6.7. DESENCADENADORES (TRIGGERS)

Los desencadenadores o triggers son un tipo especial de procedimiento almacenado que se ejecutan automáticamente como respuesta a una determinada modificación de una tabla. Un desencadenador puede configurarse para que se active cuando cambian los datos mediante una instrucción INSERT, UPDATE o DELETE.

Una sintaxis reducida para la creación de Triggers es la siguiente:

```
CREATE TRIGGER nombreDesencadenador ON tabla
FOR {[DELETE][,][INSERT][,][UPDATE]}
AS
instrucciónSQL [...n]
```

En general, un trigger pueden incluir cualquier número y tipo de sentencias Transact-SQL. Con las siguientes excepciones:

- Sentencias de creación (CREATE).
- Sentencias de eliminación de objetos (DROP).
- Sentencias de modificación de objetos (ALTER).
- Permisos (GRANT y REVOKE).

Las tablas inserted y deleted

Son tablas temporales que proporciona SQL Server:

- La tabla ***inserted*** almacena una copia de las filas que se han añadido durante la ejecución de una sentencia INSERT o UPDATE sobre una tabla que tiene asociado un trigger.

- La tabla ***deleted*** almacena una copia de las filas eliminadas durante una sentencia DELETE o UPDATE sobre una tabla que tiene asociado un trigger.

Importante: Una transacción UPDATE es como una eliminación seguida de una inserción: primero, se copian las filas antiguas en la tabla **deleted** y, a continuación, se copian las filas nuevas en la tabla del desencadenador y en la tabla **inserted**.

Veamos un ejemplo: El siguiente trigger avisa al usuario que se ha eliminado un registro de la tabla precios.

```
CREATE TRIGGER BorraPrecio
ON precios FOR DELETE
AS
PRINT '¡Acaba de eliminar una fila! '
```

La siguiente instrucción de borrado activará el desencadenador:

```
DELETE FROM precios WHERE NifPro='1111';
```

El desencadenador se activa una sola vez, independientemente del número de filas que estén afectadas, **incluso si el número de filas afectado es 0**. Prueba el ejemplo anterior borrando los precios del NifPro='1234', que no existe. Para tener una respuesta correcta podríamos cambiar el desencadenador por el siguiente:

```
ALTER TRIGGER BorraPrecio
ON precios FOR DELETE
AS
declare @contador int
select @contador=count(*) from deleted
PRINT '¡Ha eliminado '+convert(char(3),@contador)+'fila(s)!'
```

6.8. Controlar la integridad referencial mediante triggers

Si quisiéramos implementar la acción ON DELETE CASCADE mediante triggers. Antes debemos desactivar los controles de integridad referencial, ya que las restricciones se comprueban antes de activar un desencadenador. (Al intentar borrar una fila de la tabla referenciada por otra, aparece un error de violación de la regla de integridad y no deja borrar la tupla, y por tanto, el desencadenador no puede actuar). Así, el desencadenador tendrá que realizar tanto la comprobación de la integridad referencial (un desencadenador INSERT sobre la tabla que hace la referencia) como la acción de borrado en cascada (un desencadenador DELETE por cada una de las tablas a las que se hace referencia)

Veamos esto a través de un ejemplo. Sobre la base de datos precios vamos a sacar una copia de las tablas Proveedores, Artículos y Precios (a las que llamaremos Prov2, Arti2 y Prec2 respectivamente). Sobre estas tablas no definiremos ninguna regla de integridad referencial mediante las restricciones FOREIGN KEY. La integridad referencial la estableceremos mediante el desencadenador *InsertarPrecios*. Las acciones de borrado las implementaremos mediante los desencadenadores *BorraProveed* y *BorrarArti*.

El script SQL es el siguiente:

-- Instrucciones para generar las tablas Prov2, Arti2 y Prec2. (*sin integridad referencial*)

```
Use precios
go

SELECT * INTO Prov2  FROM Proveedores
SELECT * INTO Arti2 FROM Articulos
SELECT * INTO Prec2  FROM Precios

go
```

*-- Con la sintaxis utilizada **no** es necesario realizar CREATE TABLE con antelación.*
*-- Si utilizamos la sintaxis INSERT INTO Prov2 SELECT * FROM Proveedores, sí es*
-- necesario crear las tablas Prov2, Arti2 y Prec2 con antelación.
--creación de los triggers

```
CREATE TRIGGER InsertarPrecios
ON Prec2 FOR insert AS
declare @integ_pro int, @integ_art int

select @integ_pro=count(*) from inserted join Prov2 on inserted.NifPro=Prov2.nif
select @integ_art=count(*) from inserted join Arti2 on inserted.CodArt=Arti2.codigo
if @integ_pro=0 or @integ_art=0
begin
  if @integ_pro=0
    print 'Violación de regla de integridad referencial con la tabla Prov2'
  if @integ_art=0
    print 'Violación de regla de integridad referencial con la tabla Arti2'
  rollback tran
end
go

CREATE TRIGGER BorrarProveedor
ON Prov2 FOR delete AS
Delete from Prec2 where Prec2.NifPro in (select Nif from deleted)
Print 'Se ha activado el trigger BorrarProveedor'
Print 'Se ha borrado en cascada en la tabla Prec2'
go

CREATE TRIGGER BorraArticulo
ON Arti2 FOR delete AS
Delete from Prec2 where Prec2.CodArt in (select codigo from deleted)
Print 'Se ha activado el trigger BorrarArticulo'
Print 'Se ha borrado en cascada en la tabla Prec2'
go
```

-- instrucciones para comprobar el borrado en cascada

```
delete Prov2 where Nif='1111'
delete Arti2 where Codigo='11'
```

-- instrucción para comprobar la integridad referencial

```
INSERT INTO Prec2 VALUES ('9999','55',12.50)
```

Ejercicio: Realiza trigger en la base de datos *Biblio*, que permita realizar un préstamo, siempre que el libro solicitado esté disponible, y en tal caso, lo ponga a no-disponible.