Integración del CLR en SQL Server 2005. Ejecutando código administrado desde el núcleo del motor de datos

Luis Miguel Blanco Ancos

¿Qué es la integración del CLR en SQL Server 2005?

Una de las primeras cuestiones que deberíamos preguntarnos sería "¿En qué consiste o qué debemos entender por integración del CLR en SQL Server 2005?". La definición que podemos dar acerca de este nuevo aspecto de nuestro conocido gestor de datos sería la siguiente: "se trata de aquella característica de SQL Server por la cual, el CLR (Common Language Runtime), es decir, el motor de ejecución de la plataforma .NET Framework, se encuentra albergado dentro de la maquinaría del servidor de datos como un elemento más, permitiendo que SQL Server pueda beneficiarse de toda la potencia que brinda la tecnología .NET al desarrollador".

¿Qué beneficios aporta esta integración?

El objetivo principal de incorporar el CLR a la arquitectura de SQL Server, consiste en dotar al programador de aplicaciones de bases de datos de un conjunto de herramientas que potencien su trabajo, entre las cuales se encuentra la posibilidad de emplear cualquiera de los lenguajes de la plataforma .NET, para escribir los siguientes objetos del motor de datos: procedimientos almacenados, funciones definidas por el usuario, tipos definidos por el usuario, agregados y triggers.

Los mencionados objetos, hasta este momento, sólo se podían crear utilizando Transact-SQL (T-SQL), o en el caso de procedimientos almacenados, mediante el API de programación de procedimientos almacenados extendidos. Sin embargo, a partir de ahora, podemos ampliar su potencia gracias a la rica jerarquía de clases que ofrece .NET Framework para su programación; con ella, es posible desarrollar aspectos antes impensables debido a las limitaciones de T-SQL, y obtener así las ventajas de la ejecución bajo el entorno de código administrado (managed code) de .NET.

Por otro lado, esta capacidad de integración significa que a partir de ahora, podemos escribir código administrado para SQL Server utilizando Visual Studio .NET, el mismo entorno de desarrollo que empleamos para el resto de aplicaciones habituales de esta plataforma, incrementando nuestra productividad gracias a la multitud de asistentes y ayudas para la programación, sobradamente conocidas por todos los usuarios de esta herramienta.

La seguridad es otro de los apartados favorecidos por la integración, ya que elementos como la "Seguridad de Acceso al Código" (CAS - Code Access Security), la seguridad a

nivel de tipos, etc., harán que nuestro código administrado se ejecute de modo más seguro en SQL Server.

Con respecto a las versiones de los productos utilizadas para elaborar este artículo, se han utilizado SQL Server 2005 Beta 2 y Visual Studio 2005 (Whidbey) Beta 2, los cuales aparecerán el presente año en sus correspondientes versiones definitivas

Objetos administrados de base de datos

Como ya hemos apuntado anteriormente, la integración con el CLR permite escribir los conocidos procedimientos almacenados, funciones, triggers, etc., propios de T-SQL, empleando cualquiera de los lenguajes de la plataforma .NET. Dentro de este contexto de desarrollo y ejecución, a estos elementos se les denomina “Objetos Administrados de Base de Datos” o “Managed Database Objects”; para abreviar, nosotros emplearemos a partir de ahora las siglas MDBO para referirnos a estos objetos.

Dependiendo de la naturaleza de cada uno de estos objetos su implementación varía, así pues, los procedimientos almacenados, funciones y triggers se escriben como métodos compartidos (Shared o static, según el lenguaje utilizado), mientras que los agregados y tipos definidos por el usuario se escriben como clases o estructuras.

Acceso a datos desde objetos MDBO

En un alto porcentaje de ocasiones, cuando escribamos el código de un MDBO, este objeto necesitará acceder también a la información de la base de datos en la que se encuentra alojado.

Para facilitar nuestro trabajo en este sentido, el CLR proporciona al programador un proveedor de datos (in-process data provider), que se ejecuta integrado junto al código de nuestros objetos MDBO, dentro del proceso de SQL Server.

El conjunto de tipos que conforman este proveedor integrado de datos se encuentra en el espacio de nombres System.Data.SqlServer, que se halla dentro del ensamblado Sqlaccess.dll. Entre las clases contenidas por este proveedor de datos podemos mencionar las siguientes:

- **SqlCommand**. Permite la creación y ejecución de sentencias DDL y DML en la base de datos.
- **SqlDataReader**. Proporciona capacidades de navegación de sólo lectura y avance por las filas de un conjunto de resultados, obtenido a partir de una consulta contra la base de datos.
- **SqlPipe**. Es el objeto encargado de enviar a la aplicación cliente el resultado de la ejecución de un MDBO, en forma de conjunto de resultados, cadena de caracteres, etc.
- **SqlContext**. Representa el entorno o contexto dentro del cual se ejecutan los objetos MDBO. Proporciona objetos preconstruidos del proveedor de datos integrado, tales como Connection, Command, Pipe, etc., que aportan una

mayor agilidad en la ejecución. Por ejemplo, si desde un MDBO necesitamos ejecutar un comando contra la base de datos en la que se encuentra instalado, no es necesario crear primeramente una conexión y luego un comando; llamaremos directamente al método SqlContext.GetCommand(), con el que obtendremos un objeto Command, cuya propiedad Connection ya está configurada para conectar con la base de datos en curso.

Como podrá comprobar el lector, el uso de estos objetos resultará muy familiar para aquellos programadores que ya hayan trabajado con el modelo de clases de ADO.NET, lo que facilitará la curva de aprendizaje con el proveedor de datos integrado de SQL Server.

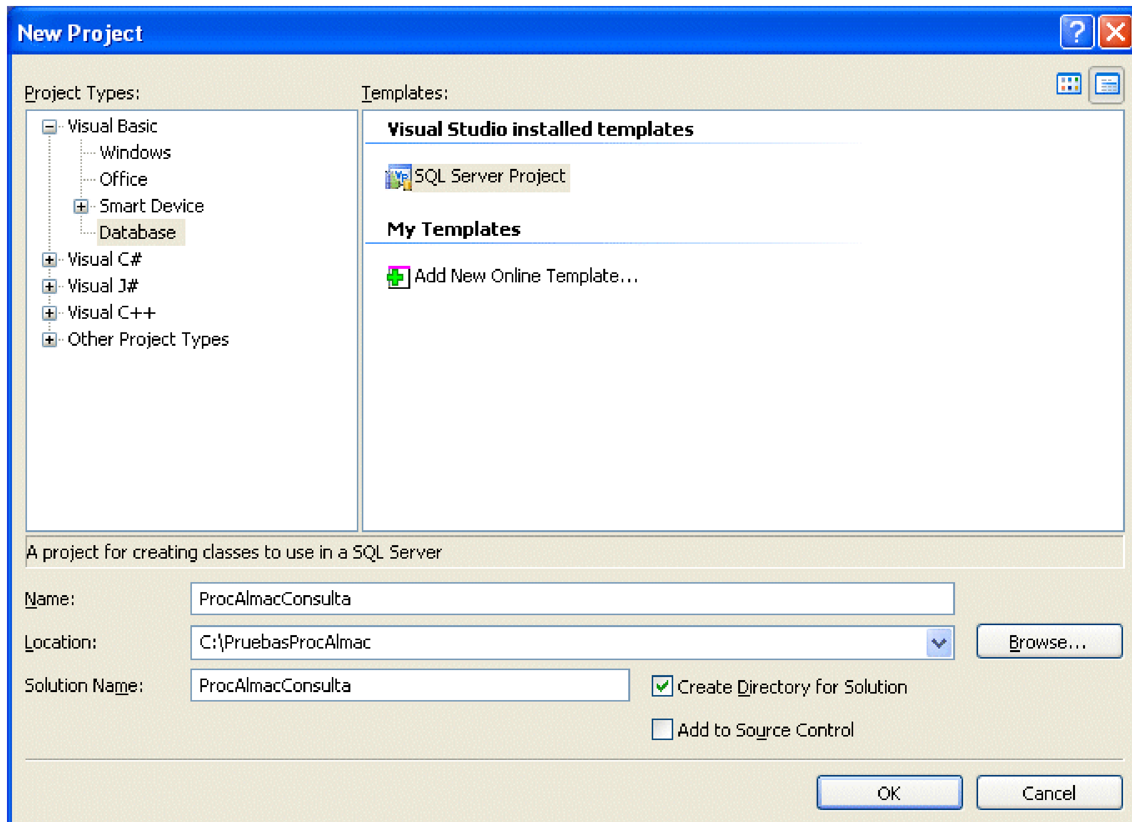
Para una descripción en profundidad de todas las clases pertenecientes al espacio de nombres System.Data.SqlServer, recomendamos al lector la consulta de la documentación de SQL Server 2005.

Creación de un procedimiento almacenado MDBO

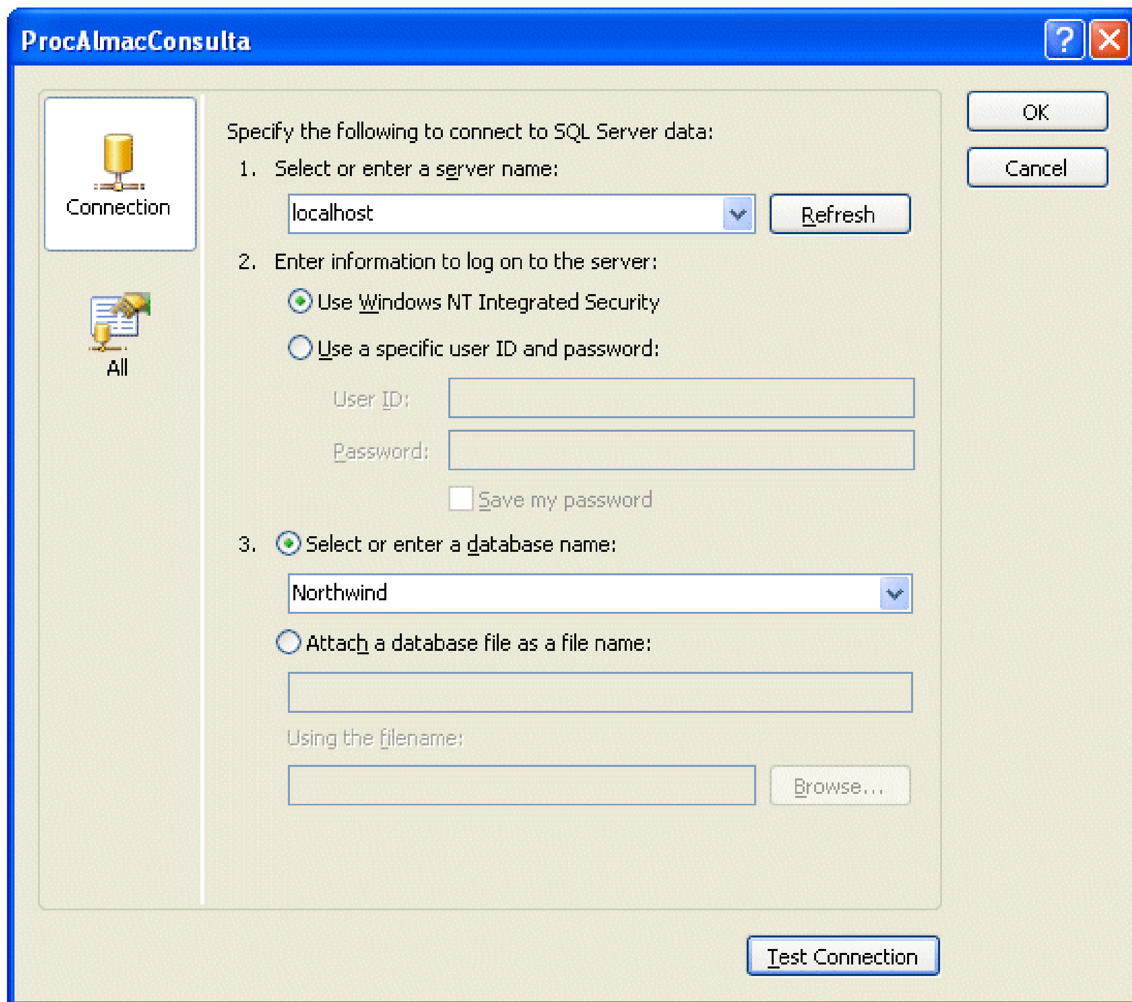
Evidentemente, no todo en este artículo va a ser teoría y conceptos sobre la integración entre el CLR y SQL Server 2005. Durante los siguientes apartados también vamos a abordar, como es natural, la vertiente práctica en la creación de objetos MDBO; y lo vamos a hacer comenzando por los procedimientos almacenados, uno de los objetos más sobradamente conocidos y utilizados habitualmente en T-SQL.

Antes de comenzar, es necesario puntualizar que el código de los ejemplos de este artículo está desarrollado en VB.NET, estando disponible para su descarga por parte del lector en la dirección www.dotnetmania.com.

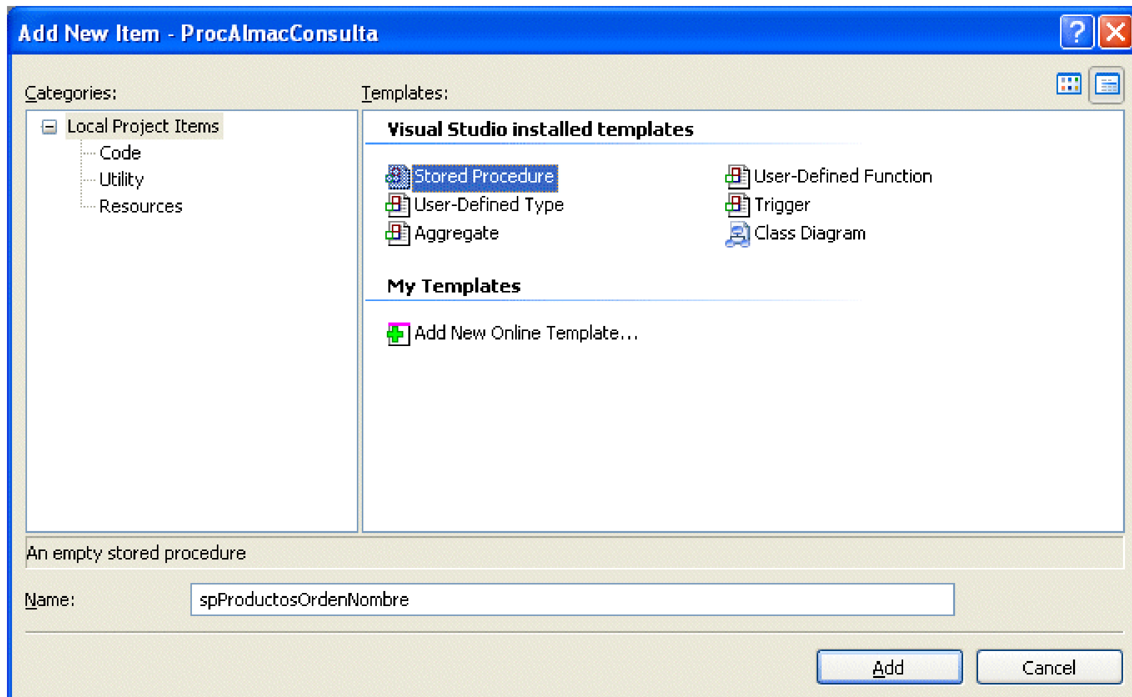
En primer lugar abriremos Visual Studio 2005 y seleccionaremos la opción de menú "File + New + Project". En el cuadro de diálogo "New Project", dentro del panel de tipos de proyecto, haremos clic sobre el nuevo tipo Database; como plantilla elegiremos "SQL Server Project", la cual dispone por defecto de todos los ajustes necesarios para crear objetos MDBO. El nombre del proyecto será ProcAlmacConsulta como vemos en la siguiente figura.



Una vez aceptado el cuadro de diálogo de creación de proyecto, la primera acción a realizar será establecer una conexión con un origen de datos, para lo cual, y de modo automático, Visual Studio mostrará el cuadro de diálogo de conexión a una base de datos, en el que introduciremos los valores necesarios para conectar con nuestro servidor SQL Server local, y utilizar la base de datos Northwind, empleando seguridad integrada de Windows, como vemos en la siguiente figura.



A continuación, mediante la opción de menú “Add + New Item…”, abriremos el cuadro de diálogo para añadir un objeto MDBO de tipo Stored Procedure, al que llamaremos spProductosOrdenNombre, como vemos en la siguiente figura.



Al aceptar esta ventana, se creará una nueva clase con el nombre StoredProcedures, que tendrá un método con el nombre spProductosOrdenNombre. Llegados a este punto, debemos comentar algunas características del lenguaje referentes a esta clase:

- La clase está definida con la palabra clave Partial. Esta es una novedad incluida en los lenguajes de la plataforma .NET, que nos permite repartir los miembros de una misma clase en varios archivos de código. Hasta la presente versión de .NET, el código de una clase tenía que escribirse en un único archivo.
- El método se define como Shared o compartido, para que SQL Server pueda ejecutarlo directamente sin tener que instanciar un objeto.
- A nivel de método se aplica el atributo SqlProcedure, que proporciona información a SQL Server acerca del tipo de objeto de base de datos que vamos a crear. De esta manera, en función del tipo de objeto a escribir, existen diferentes atributos que deberemos aplicar: SqlFunction, SqlUserDefinedType, etc.

Seguidamente escribiremos el método de la forma mostrada en el siguiente código fuente.

```
Imports System
Imports System.Data
Imports System.Data.Sql
Imports System.Data.SqlServer
Imports System.Data.SqlTypes

Partial Public Class StoredProcedures
    <SqlProcedure()> _
    Public Shared Sub spProductosOrdenNombre()
```

```
        Dim oCommand As SqlCommand
        Dim oDataReader As SqlDataReader
        Dim oPipe As SqlPipe

        oCommand = SqlContext.GetCommand()
        oCommand.CommandType = CommandType.Text
        oCommand.CommandText = "SELECT ProductID, ProductName, QuantityPerUnit "
& _
            "FROM Products ORDER BY ProductName"

        oDataReader = oCommand.ExecuteReader()

        oPipe = SqlContext.GetPipe()
        oPipe.Send(oDataReader)
        oPipe.Send("Resultados enviados al cliente")
    End Sub
End Class
```

En el código que acabamos de ver, utilizamos el objeto SqlContext para obtener una instancia de un objeto SqlCommand, que emplearemos para ejecutar una consulta contra la base de datos. Observemos que, gracias al diseño de las clases del proveedor de datos integrado, no es necesario crear un objeto Connection, ya que por defecto se asume que vamos a trabajar con la base de datos que hemos especificado al crear el proyecto; por lo tanto, dicha conexión ya se encuentra incluida en aquellos objetos que lo necesiten, como es el caso del comando.

Asignaremos al comando el texto de la consulta y la ejecutaremos, obteniendo el conjunto de resultados como un objeto SqlDataReader; hasta ahora nada nuevo para todo aquel que haya programado con ADO.NET. A partir de aquí es donde entramos en la parte más interesante, ya que necesitamos devolver el objeto SqlDataReader de alguna manera al cliente que ejecuta el procedimiento almacenado; esto lo conseguiremos utilizando un objeto SqlPipe, que obtenemos también de SqlContext.

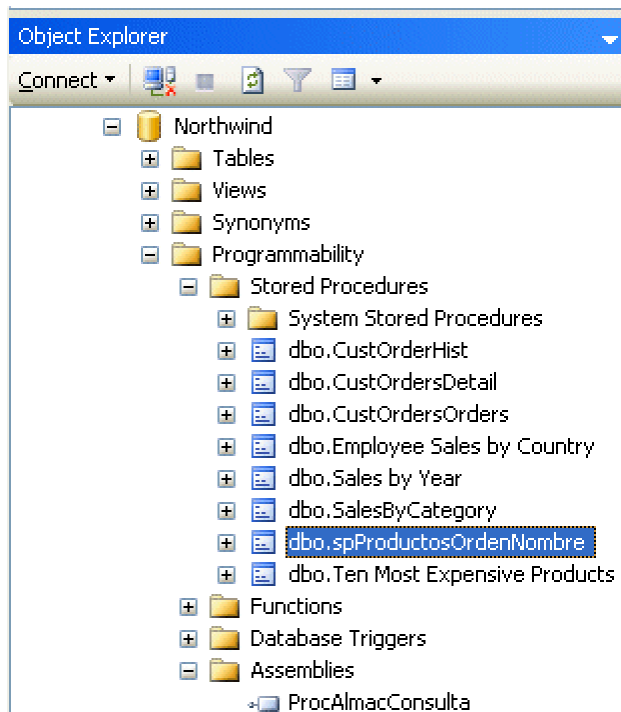
La clase SqlPipe dispone del método Send(), que será el empleado para enviar resultados al cliente. Gracias a las distintas sobrecargas de este método, es posible devolver información de diferente tipo; en este caso lo usamos para retornar un objeto SqlDataReader, y a continuación una cadena de caracteres.

Compilación, despliegue y ejecución de un objeto MDBO

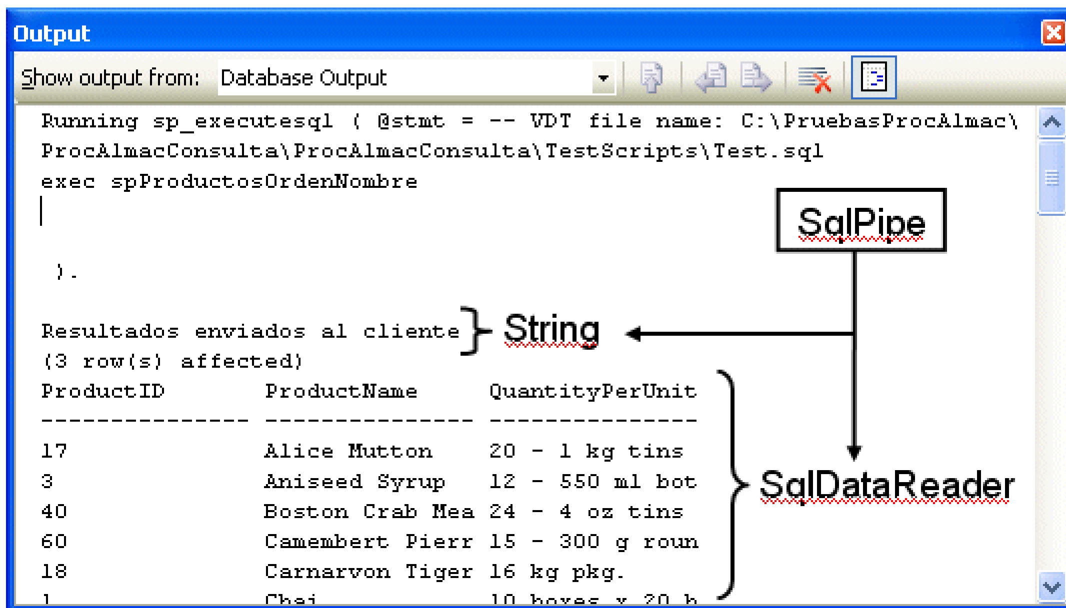
Terminada la escritura del objeto vamos a proceder a ejecutarlo para probar su funcionamiento.

En primer lugar, mediante la opción de menú “Build + Build Solution” compilaremos el código, obteniendo el ensamblado ProcAlmacConsulta.dll en el directorio bin del proyecto. A continuación, la opción “Build + Deploy Solution” desplegará-instalará el ensamblado generado dentro de SQL Server.

Si queremos verificar que la instalación del objeto ha tenido éxito, abriremos SQL Server Management Studio y realizaremos las siguientes acciones: en la ventana "Object Explorer" nos situaremos sobre la base de datos Northwind, expandiremos el nodo Programmability, y dentro de este, los nodos Assemblies y Stored Procedures, que mostrarán respectivamente, el ensamblado y procedimiento almacenado que hemos creado desde Visual Studio, como vemos en la siguiente figura.



Para ejecutar nuestro procedimiento almacenado desde Visual Studio pulsaremos F5, o el botón Start de la barra de herramientas, obteniendo el resultado en la ventana Output, como vemos en la siguiente figura, en la que también señalamos cada uno de los elementos devueltos por el objeto SqlPipe. En el caso de que esta ventana no muestre la información deseada, debemos seleccionar, de su lista desplegable "Show output from", el valor "Database Output".



Lo que realmente hace Visual Studio cuando ponemos en funcionamiento el proyecto es ejecutar el archivo de script Test.sql, que contiene la instrucción "exec spProductosOrdenNombre". Este archivo está situado en la carpeta TestScripts del proyecto, y es creado automáticamente cuando desarrollamos proyectos de tipo SQL Server. Según el elemento de nuestro proyecto que deseemos probar al ejecutar el proyecto, deberemos escribir la/s instrucción/es adecuadas en este archivo.

También podemos ejecutar este procedimiento almacenado desde SQL Server Management Studio, escribiendo la instrucción antes mencionada en una nueva ventana de consultas, en cuya parte inferior, las pestañas Results y Messages, mostrarán respectivamente los datos devueltos por el procedimiento y la cadena enviada con el objeto SqlPipe.

Agregar y registrar manualmente un ensamblado y su contenido en SQL Server

Aunque como acabamos de ver, el medio más sencillo y directo de instalar un ensamblado .NET en SQL Server 2005 es desde el IDE de Visual Studio 2005, quizá nos encontremos en algún momento con la necesidad de realizar esta operación, pero no dispongamos del mencionado entorno de desarrollo. ¿Qué hacemos entonces, nos quedamos sin instalar el ensamblado?.

No se preocupe el lector, ya que es perfectamente posible agregar manualmente el ensamblado y registrar su contenido en el servidor de datos, puesto que la nueva versión de SQL Server incorpora los comandos apropiados a tal efecto.

Vamos a comprobar esta característica de la integración con el CLR, eliminando primeramente desde SQL Server Mgmt. Studio los elementos creados en el apartado anterior. Para ello, en el nodo Stored Procedures haremos clic derecho sobre el procedimiento almacenado creado anteriormente, y seleccionando la opción de menú

Delete lo eliminaremos de la base de datos. A continuación, en el nodo Assemblies, procederemos exactamente igual con el ensamblado.

Posteriormente, para añadir un ensamblado manualmente a la base de datos, deberemos utilizar la instrucción CREATE ASSEMBLY, proporcionando la ruta en la que reside el archivo del ensamblado, y especificando el conjunto de permisos de seguridad que otorgamos al mismo, como vemos en el siguiente fuente.

```
CREATE ASSEMBLY ProcAlmacConsulta
FROM 'C:\EnsambladosCLR\ProcAlmacConsulta.dll'
WITH PERMISSION_SET = SAFE
```

La cláusula WITH PERMISSION_SET será la que empleemos para establecer el grupo de permisos asignados al ensamblado; dispone de tres valores que describimos seguidamente:

- SAFE. Representa el nivel con mayor grado de seguridad, ya que restringe el acceso del ensamblado a recursos tales como el registro, sistema de archivos, redes, etc. En el caso de no especificar ningún valor de seguridad para el ensamblado, se tomará por defecto este nivel.
- EXTERNAL_ACCESS. Cuando establecemos este tipo de permisos, el ensamblado puede acceder a recursos como archivos, el registro, variables de entorno, etc.
- UNSAFE. El ensamblado tiene pleno acceso a todos los recursos. Al ser la opción menos restrictiva ha de emplearse con sumo cuidado.

Depuración de objetos MDBO desde Visual Studio

Al igual que ocurre durante el desarrollo de otros tipos de aplicaciones, la creación de objetos administrados para SQL Server es una labor que no está exenta de errores. Es por ello que Visual Studio nos proporciona también la capacidad de depurar este tipo de código, aunque se trata de una característica que no está activada por defecto.

Para habilitar la depuración en un proyecto de tipo SQL Server debemos seguir estos pasos: abrir la ventana "Server explorer", expandir el nodo “Data Connections”, hacer clic derecho sobre la conexión de datos que hayamos establecido para el proyecto, y finalmente, seleccionar la opción de menú “Allow SQL/CLR Debugging”. Aparecerá una ventana con el siguiente mensaje: “SQL/CLR debugging will cause all manager threads on the Server to be stopped. Do you want to continue?”, a la que deberemos responder afirmativamente, con lo que ya tendremos disponible la depuración del código en nuestro proyecto.

Personalizando el conjunto de resultados a devolver

En el anterior ejemplo hemos podido ver que la clase SqlPipe es la encargada, dentro de un procedimiento almacenado, de devolver la información resultante al cliente en forma de objeto SqlDataReader. Aunque este modo de trabajo será el adecuado en

muchas ocasiones, es posible que nos encontremos con requerimientos adicionales a la hora de enviar la información.

Supongamos que vamos a crear un procedimiento almacenado MDBO con una consulta sobre la tabla Employees, y por un lado debemos traducir los valores del campo TitleOfCourtesy, mientras que por otra parte tenemos que devolver FirstName y LastName como un único campo. Daremos el nombre spEmpleadosFormato a este nuevo procedimiento almacenado.

En este caso no podemos retornar directamente el objeto SqlDataReader producto de ejecutar el comando, lo que debemos hacer es construir nuestra propia estructura de almacenamiento de datos, recorrer el SqlDataReader para manipular los campos necesarios, y devolver estos datos transformados utilizando el objeto SqlPipe.

Todo lo anterior lo conseguiremos definiendo un modelo de registro y campos personalizados. Para el registro utilizaremos la clase SqlDataRecord, mientras que para los campos emplearemos la clase SqlMetaData; en estos últimos crearemos un objeto por campo, y los añadiremos todos a un array, que posteriormente pasaremos al constructor de SqlDataRecord.

Antes de iniciar la manipulación de datos, indicaremos el comienzo del envío de resultados llamando al método SqlPipe.SendResultsStart(), cada vez que enviemos un registro personalizado llamaremos a SqlPipe.SendResultsRow(), y cuando hayamos terminado de procesar todos los registros ejecutaremos SqlPipe.SendResultsEnd(). Veamos el siguiente fuente, que contiene el código necesario.

```
Partial Public Class StoredProcedures
  <SqlProcedure()> _
  Public Shared Sub spEmpleadosFormato ()
    Dim oCommand As SqlCommand
    Dim oDataReader As SqlDataReader
    Dim oPipe As SqlPipe
    Dim aMetadatos(1) As SqlMetaData
    Dim oRegistro As SqlDataRecord
    Dim sTitCortesia As String
    Dim sNombre As String

    ' crear comando, ejecutarlo y obtener DataReader
    oCommand = SqlContext.GetCommand()
    oCommand.CommandType = CommandType.Text
    oCommand.CommandText = "SELECT TitleOfCourtesy, FirstName, LastName " & _
      "FROM Employees"
    oDataReader = oCommand.ExecuteReader()

    ' definir metadatos y registro para devolver datos resultantes
    aMetadatos(0) = New SqlMetaData("TituloCortesia", SqlDbType.NVarChar, 15)
    aMetadatos(1) = New SqlMetaData("NombreEmpleado", SqlDbType.NVarChar, 40)
```

```
        oRegistro = New SqlDataRecord(aMetadatos)

        oPipe = SqlContext.GetPipe()

        ' indicamos que va a comenzar el envío de resultados
        oPipe.SendResultsStart(oRegistro, False)
        While oDataReader.Read()
            ' obtener valores de campos y modificarlos
            sTitCortesia = oDataReader.GetString(0)
            Select Case sTitCortesia
                Case "Ms."
                    sTitCortesia = "Señorita"
                Case "Dr."
                    sTitCortesia = "Doctor"
                Case "Mrs."
                    sTitCortesia = "Señora"
                Case "Mr."
                    sTitCortesia = "Señor"
            End Select

            sNombre = oDataReader.GetString(1) & " " & oDataReader.GetString(2)

            ' pasar valores modificados a registro
            oRegistro.SetString(0, sTitCortesia)
            oRegistro.SetString(1, sNombre)

            ' enviar registro al cliente
            oPipe.SendResultsRow(oRegistro)
        End While
        ' aquí termina el envío de resultados
        oPipe.SendResultsEnd()
    End Sub
End Class
```

Los datos obtenidos al ejecutar el anterior procedimiento los podemos ver en la siguiente figura.

```
TituloCortesia  NombreEmpleado
--------------- ---------------
Señorita        Nancy Davolio
Doctor          Andrew Fuller
Señorita        Janet Leverling
Señora          Margaret Peacoc
Señor           Steven Buchanan
Señor           Michael Suyama
Señor           Robert King
Señorita        Laura Callahan
Señorita        Anne Dodsworth
(9 row(s) returned)
```

Desarrollo de un tipo definido por el usuario MDBO

A continuación vamos a abordar una característica, que si bien ya se encuentra presente en T-SQL, se ve potenciada gracias a la integración del CLR; nos referimos a la creación de tipos de datos propios (User Defined Type) para SQL Server, usando código administrado.

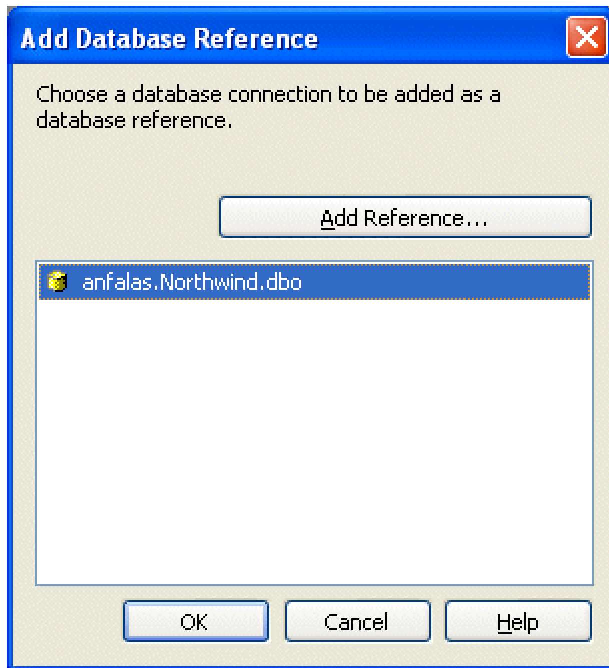
Los tipos de usuario MDBO nos permiten extender los tipos de datos nativos de SQL Server, y entre sus características podemos destacar las siguientes:

- Se deben codificar como una clase o estructura.
- Permiten la creación de tipos de datos complejos sin la necesidad de estar basados en un tipo nativo de SQL Server; en este sentido aportan mayor flexibilidad que los tipos de usuario tradicionales de T-SQL.
- En función de lo expresado en el anterior punto, estos tipos se ejecutan, manipulan y almacenan como objetos reales, lo cual permite al código cliente que los utiliza, llamar a sus métodos y manipular sus propiedades.

Pedido, un nuevo tipo de datos para SQL Server

Partiendo de las anteriores premisas, nuestro siguiente ejemplo va a consistir en realizar el desarrollo de Pedido, un nuevo tipo de dato MDBO. Este tipo estará basado en el supuesto de una empresa de transportes, que distribuye pedidos de material a diversos puntos del país. Por cada pedido necesita registrar la descripción del material, y la distancia recorrida en kilómetros hasta el punto de destino, para poder calcular, en base a una tarifa de desplazamientos, el importe a facturar al cliente por el envío del pedido.

En primer lugar, crearemos en Visual Studio un nuevo proyecto de tipo SQL Server Project, con el nombre TiposUsr. Acto seguido tendremos que establecer la referencia con la base de datos asociada al proyecto; como ya creamos una conexión hacia Northwind en el anterior proyecto, Visual Studio nos ofrece una caja de diálogo con las bases de datos disponibles. En nuestro caso volveremos a elegir la misma, aunque siempre podemos pulsar el botón "Add Reference", y abrir la ventana de selección para establecer referencia con base de datos diferente. Ver la siguiente figura.



A continuación, agregaremos al proyecto un elemento de tipo "User-Defined Type" al que llamaremos Pedido. El entorno de desarrollo creará entonces una clase con ese mismo nombre y el esqueleto básico de la misma, a la que deberemos añadir nuestro propio código para completar su funcionalidad. El código al completo podemos verlo en el siguiente fuente.

```
<Serializable()> _
<SqlUserDefinedType(Format.SerializedDataWithMetadata, MaxByteSize:=512)> _
Public Class Pedido
    Implements INullable

    Private bIsNull As Boolean
    Private sDescripcion As String
    Private nDistancia As Integer
    Private nTarifa As Integer

    Public Property Descripcion() As String
        Get
            Return sDescripcion
        End Get
        Set(ByVal value As String)
            sDescripcion = value
        End Set
    End Property

    Public Property Distancia() As Integer
        Get
            Return nDistancia
        End Get
```

```
  Set(ByVal value As Integer)
    nDistancia = value
  End Set
End Property

Public Property Tarifa() As Integer
  Get
    Return nTarifa
  End Get
  Set(ByVal value As Integer)
    nTarifa = value
  End Set
End Property

Public ReadOnly Property IsNull() As Boolean Implements INullable.IsNull
  Get
    Return bIsNull
  End Get
End Property

Public Shared ReadOnly Property Null() As Pedido
  Get
    Dim oPedido As Pedido = New Pedido()
    Return oPedido
  End Get
End Property

Public Overrides Function ToString() As String
  Dim sResultado As String

  If Me.IsNull Then
    sResultado = "NULL"
  Else
    sResultado = sDescripcion & "-" & CType(nDistancia, String) & "-" & _
      CType(nTarifa, String)
  End If

  Return sResultado
End Function

Public Shared Function Parse(ByVal s As SqlString) As Pedido
  Dim oPedido As Pedido
  Dim aElementos() As String

  Try
    If s.IsNull Then
      Return Pedido.Null
```

```
            Else
                oPedido = New Pedido()
                aElementos = s.ToString().Split("-"c)

                oPedido.sDescripcion = aElementos(0)
                oPedido.nDistancia = CType(aElementos(1), Integer)
                oPedido.nTarifa = CType(aElementos(2), Integer)

                Return oPedido
            End If

        Catch ex As Exception
            Throw New Exception("Error al convertir")
        End Try
    End Function

    Public Function ImporteFacturar() As Integer
        Dim nResultado As Integer
        nResultado = nDistancia * nTarifa
        Return nResultado
    End Function
End Class
```

De todo el código necesario para crear el tipo Pedido destacaremos los siguientes aspectos:

- Aplicamos a nivel de clase los atributos Serializable y SqlUserDefinedType, para controlar el modo de serialización del tipo.
- Implementamos el interfaz INullable, y por consiguiente, las propiedades IsNull y Null, para que la clase disponga de la capacidad de manejar valores nulos.
- Al igual que las clases nativas de la plataforma .NET soportan la conversión a/desde cadenas, en nuestra clase creamos los métodos ToString() y Parse(), en los que desarrollamos la lógica necesaria para disponer también de esta característica.

Finalizada la escritura de esta clase, la compilaremos e instalaremos en SQL Server. Para trabajar con este nuevo tipo de dato, iniciaremos SQL Server Management Studio y nos situaremos sobre la base de datos Northwind; seguidamente abriremos una nueva ventana de consultas, en la que ejecutaremos las instrucciones mostradas en el siguiente código fuente.

```
/* crear objeto de tipo Pedido, asignar valores
y llamar a sus métodos */
DECLARE @oP Pedido
SET @oP.Descripcion = 'Tarjetas Sintonizadoras TV'
SET @oP.Distancia = 214
SET @oP.Tarifa = 87
```

```
SELECT @oP.ImporteFacturar()
SELECT @oP.ToString()

/* crear objeto a partir de una cadena mediante
el método compartido Parse */
DECLARE @oP Pedido
SET @oP = Pedido::Parse('Grabadoras DVD-384-65')
SELECT @oP.Descripcion AS DESCRIPCION,
@oP.Distancia AS DISTANCIA,
@oP.Tarifa AS TARIFA,
@oP.ImporteFacturar() AS IMPORTE

/* crear tabla con un campo de tipo Pedido y añadir filas,
hemos de utilizar una variable del mismo tipo para
almacenar los valores a insertar en dicho campo */
CREATE TABLE MisPedidos (
IDPedido int NOT NULL,
DatoPedido Pedido NULL)

DECLARE @oPedido Pedido

SET @oPedido.Descripcion = 'Monitores TFT 17'
SET @oPedido.Distancia = 125
SET @oPedido.Tarifa = 6
INSERT INTO MisPedidos VALUES (1,@oPedido)

SET @oPedido.Descripcion = 'Micros P.III'
SET @oPedido.Distancia = 95
SET @oPedido.Tarifa = 4
INSERT INTO MisPedidos VALUES (2,@oPedido)

/* consulta simple contra la tabla */
SELECT IDPedido,
DatoPedido.Descripcion AS DESCRIPCION,
DatoPedido.Distancia AS DISTANCIA,
DatoPedido.Tarifa AS TARIFA,
DatoPedido.ImporteFacturar() AS IMPORTE
FROM MisPedidos

/* obtener el campo Pedido de una de las filas de la tabla
y llamar a un método del tipo */
DECLARE @oPed Pedido

SELECT @oPed = DatoPedido
FROM MisPedidos
WHERE IDPedido = 1
```

SELECT @oPed.ImporteFacturar() AS RESULTADO

Observe el lector que en los ejemplos del anterior código, al declarar una variable de tipo Pedido, el acceso a sus miembros se realiza usando la sintaxis habitual del operador punto (.), con excepción del método Parse(), para el que utilizamos el operador dos puntos (::). Esto se debe a que Parse() es un método calificado como Shared, y dadas las características de este tipo de métodos, tendremos que emplear el mencionado operador cuando los invoquemos desde SQL Server.

Siguiendo con el método Parse(), dado que es posible que la cadena que le pasamos como parámetro no tenga el formato esperado, por ejemplo: "Pedido::Parse('Grabadoras DVD-abc-65')", creamos una excepción con un mensaje personalizado, de forma que cuando se produzca un error por este motivo, podamos identificarlo adecuadamente. Ver la siguiente figura.

```
Msg 6522, Level 16, State 2, Line 2
A .NET Framework error occurred during execution of user defined routine or aggregate 'Pedido':
System.Exception: Error al convertir
    at TiposUsr.Pedido.Parse(SqlString s)
    at SQLCLR_Eval(IntPtr , IntPtr ).
```

Por último, cuando utilizamos Pedido como campo de una tabla, dado que se trata de un tipo complejo, una consulta del estilo "SELECT * FROM MisPedidos" daría error, por lo que hemos de ser más precisos, indicando a qué propiedad o método del objeto contenido en el campo queremos acceder, mediante la sintaxis NombreCampo.Miembro.

Algunas reflexiones sobre la integración del CLR como conclusión

A lo largo del presente artículo hemos repasado algunos de los principales aspectos de la integración del CLR en el nuevo SQL Server 2005. Ahora bien, el conjunto de características aquí expuestas, no se traducen en que esta nueva funcionalidad de SQL Server sea el remedio que vaya a solucionar todos nuestros problemas de diseño y rendimiento a la hora de desarrollar aplicaciones de gestión de datos.

La integración del CLR es una excelente característica que mejora enormemente el gestor de datos, pero Transact-SQL sigue existiendo, y ambos elementos ocupan un sitio necesario en SQL Server; en ningún momento se pretende que la posibilidad de escribir código administrado en una base de datos sea su sustituto.

A la hora de plantear el uso de una u otra característica, debemos sopesar las ventajas e inconvenientes de cada una dentro del escenario de ejecución bajo el cual van a trabajar, y elegir la que proporcione un rendimiento más eficaz.

Dentro del modelo de desarrollo basado en n-capas, la integración del CLR difumina el límite existente entre la capa de datos y la capa de lógica de negocio, ya que ahora

podemos integrar, si así lo queremos, una gran parte de la lógica de proceso en el interior del almacén de datos.

Cuanto más código de negocio traslademos a la capa de datos, dicho código tendrá un acceso más veloz a los datos, lo cual quiere decir que se mejorará la velocidad de proceso de nuestro código. No obstante, esto puede ser un arma de doble filo, ya que situar demasiado código en la capa de datos, puede penalizar seriamente el rendimiento y las posibilidades de extensibilidad de nuestro diseño.

Por todo ello, debemos hacer un uso racional de la integración, realizando un análisis de nuestros requerimientos, y aplicándola en aquellas situaciones que resulte conveniente.