Erik S.C. van de Ven is werkzaam bij Microsoft Services Nederland

Eenvoudig en veilig .NET-programmeren in SQL Server 2005

DE HECHTE INTEGRATIE TUSSEN SQL SERVER 2005 EN DE .NET RUNTIME

Met Microsoft SQL Server 2005 komen de kracht en eenvoud van het programmeren in .NET beschikbaar binnen SQL server. Wie kent niet de soms omslachtige en ingewikkelde TSQL-syntax om ogenschijnlijk eenvoudige zaken op te lossen? Met SQL Server 2005 zal de .NET Common Language Runtime (CLR) binnen SQL Server gehost worden. Dit betekent dat stored procedures, triggers en functies voortaan eenvoudig in C# of VB.NET geprogrammeerd kunnen worden; of natuurlijk in een willekeurige andere .NET-taal. Naast het eenvoudig programmeren komen ook direct andere voordelen van het gebruik van .NET beschikbaar.

Een aansprekend voordeel is het gebruik van de rijke Base Class Libraries (BCL). Vanuit een stored procedure kunnen we bijvoorbeeld eenvoudig reguliere expressies (uit *System.Text.RegularExpressions*) gebruiken! Het gebruik van managed code biedt echter nog meer voordelen. Denk hierbij aan de verbeterde mogelijkheden voor debugging, het kunnen toetsen van typesafety en de extra mogelijkheden rond beveiliging. TSQL zal zeker blijven en zelfs de voorkeur genieten voor data- en set-georiënteerd werken. Maar wanneer we berekeningen, stringmanipulatie of gewoon complexe logica moeten uitvoeren, kunnen we dit veel eenvoudiger doen in een .NET-taal. In dit artikel beschrijf ik voorbeelden van de nieuwe toepassingsmogelijkheden en leg ik de interne werking van de CLR in SQL Server uit.

Functies en stored procedures in .NET

We beginnen met een eenvoudig voorbeeld van het valideren van een e-mailadres om te laten zien hoe we .NET-code in SQL Server kunnen gebruiken. Om deze functie in SQL Server te kunnen gebruiken, schrijft de ontwikkelaar een class library met daarin een class met een static methode; zie codevoorbeeld 1.
Om deze functie in SQL Server te kunnen gebruiken, moeten we deze assembly naar de database-server deployen. In Visual Studio 2005 kan dit eenvoudig door rechts te klikken op het project in de *Solution Explorer* en te kiezen voor de optie *Deploy*. Maar wat gebeurt er 'achter de schermen'? Om dat te laten zien kiezen we voor een handmatige registratie in SQL Server. Feitelijk zijn er twee stappen die we zullen moeten uitvoeren. Eerst moeten we de assembly registreren en vervolgens dienen we aan te geven welke methode we in de assembly als functie in SQL Server willen gebruiken. Codevoorbeeld 2 toont de TSQL-statements die nodig zijn om beide stappen uit te voeren.

Het laatste deel van het 'create function'-statement verwijst naar de naam van de static methode van codevoorbeeld 1. De syntax die hiervoor gebruikt wordt is:

```
<assembly naam>.<fully qualified class name>.<methode naam>
```

Nu we de functie geregistreerd hebben, kunnen we haar daadwerkelijk gebruiken. We kunnen de functie gewoon in TSQL hanteren zoals we gewend zijn:

```
select dbo.isValidEmail ('somebody@somecompany.com')
```

```
#region Using directives

using System;
using System.Text.RegularExpressions;

#endregion

namespace SQLCLR
{
  public class custom
  {
    private const string emailPattern =
      @"^([\w-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-
      9]{1,3}\.)|(([\w-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$";

    public static bool isValidEmail(string email)
    {
      return (Regex.IsMatch(email, emailPattern));
    }
  }
}
```

Codevoorbeeld 1.

```
-- registreer de assembly
create assembly CLR
from 'c:\erikven\custom.dll'

-- registreer de functie
create function isValidEmail
(
@email nvarchar(50)
)
RETURNS bit
as external NAME CLR.[SQLCLR.custom].isValidEmail
```

Codevoorbeeld 2.

Zoals verwacht levert dit een resultaatset op met het getal 1 (true). Nu we het eenvoudigste voorbeeld hebben getoond, willen we de code uit codevoorbeeld 1 uitbreiden om de overige mogelijkheden van de integratie tussen SQL Server en .NET te laten zien.
In het volgende voorbeeld gaan we de code uitbreiden door data uit SQL Server te benaderen. Om aan te geven dat we data willen lezen, moeten we het attribuut *SqlFunction* (uit de *System.Data.Sql* namespace) toepassen op onze functie. In codevoorbeeld 3 is te zien hoe we de named parameter *DataAccess* de waarde *DataAccessKind.Read* toekennen. Er is een soortgelijke parameter beschikbaar met de naam *SystemDataAccess* om systeemmetadata te kunnen benaderen.

Wanneer we het *DataAccess*-attribuut gebruiken, kunnen we via de static method *GetCommand* van het *SqlContext*-type toegang krijgen tot een command-object. Het *SqlContext*-type (uit *System.Data.SqlServer*) is een belangrijke class die we vanuit server side-code toegang verschaffen tot de actuele contextinformatie zoals commando, connectie en transactie van de SQL Server-verbinding. Het *SqlCommand*-object (uit *System.Data.SqlServer*), dat door de methode *GetCommand()* wordt geretourneerd, lijkt qua signatuur sterk op het *SqlCommand*-type zoals we dat vandaag de dag kennen uit de *System.Data.SqlClient* namespace. De connectie-property van dit object is automatisch gezet en het enige dat ons nog rest is het zetten van de *CommandText* en het aanroepen van de bekende *ExecuteScalar()*. De functie kan eenvoudig worden geregistreerd en gebruikt zoals in codevoorbeeld 4 is aangegeven.

Het ontwikkelen van een stored procedure gaat op vrijwel identieke wijze. Met een stored procedure hebben we zelfs de mogelijkheid om outputparameters te gebruiken en textmessages terug te sturen naar de client. De outputparameters dienen we *by reference* mee te geven zoals weergegeven in codevoorbeeld 5. Vervolgens gebruiken we weer het *SqlContext*-type, maar deze keer om toegang te krijgen tot het *SqlPipe*-object. Het *SqlPipe*-type (uit *System.Data.SqlServer*) is een onderdeel van de in-process managed provider in SQL dat een communicatiekanaal representeert tussen de client en de server. In dit geval gebruik we een overload van de *Send()*-methode om een string terug te sturen naar de client. Vervolgens registreren we de stored procedure zoals weergegeven in het script in codevoorbeeld 6.

De aanroep van de stored procedure is weergegeven in het volgende script. Wanneer we de functie aanroepen, wordt de *moreThanOne* Boolean-waarde gezet en wordt een textmessage geretourneerd naar de caller.

```
declare @result int
declare @moreThanOne bit
set @moreThanOne = 0
exec @result = numberOfUsersP 'New York', @moreThanOne OUTPUT
select @result
select @moreThanOne
```

Naast het combineren van .NET-logica met SQL Server-functies en procedures is het ook mogelijk om datatypes in .NET te combineren met SQL Server. In de volgende paragraaf laat ik zien hoe dit werkt.

.NET abstracte datatypes in SQL Server

In deze paragraaf laten we zien hoe we de standaard SQL-types (zoals int, varchar, bit etc.) kunnen uitbreiden. Met SQL Server 2005 wordt het mogelijk om abstracte datatypen te definiëren in .NET. Een abstract datatype is een samenstelling van een nieuw type op basis van bestaande types. *1 Codevoorbeeld 7 toont een abstract datatype *Address* gedefinieerd in .NET. Ik laat nu zien welke stappen we moeten nemen om dit nieuwe type transparant in SQL Server te kunnen gebruiken.
In eerste instantie dienen we een public class te definiëren met daarop het attribuut *SqlUserDefinedType* (uit *System.Data.Sql*).

```
[SqlFunction(DataAccess = DataAccessKind.Read)]
public static int numberOfUsers(string city)
{
SqlCommand _cmd = SqlContext.GetCommand();

_cmd.CommandText = string.Format("SELECT COUNT(*)
FROM users WHERE city='{0}'", city);

return ((int)_cmd.ExecuteScalar());
}
```

Codevoorbeeld 3

```
-- registreer functie
create function numberOfUsers
(
@city nvarchar(50)
)
RETURNS int
as external NAME CLR.[SQLCLR.custom].numberOfUsers

-- roep functie aan
select dbo.numberOfUsers('New York')
```

Codevoorbeeld 4.

```
public static int numberOfUsersP(string city,
          ref bool moreThanOne)
{
   SqlCommand _cmd = SqlContext.GetCommand();

   _cmd.CommandText = string.Format("SELECT COUNT(*)
          FROM users WHERE city='{0}'", city);

   // send text messages back to client
   SqlPipe pipe = SqlContext.GetPipe();
   pipe.Send("Het aantal users is berekend!");

   // calculate the result
   int result = (int)_cmd.ExecuteScalar();

   // do some interesting business logic
   if (result > 1)
   {
   moreThanOne = true;
   }
   else
   {
   moreThanOne = false;
   }
   return (result);
}
```

Codevoorbeeld 5.

```
-- registreer stored procedure
create procedure numberOfUsersP
(
@city nvarchar(50),
@moreThanOne bit OUTPUT
)
as external NAME CLR.[SQLCLR.custom].numberOfUsersP
```

Codevoorbeeld 6.

*1 Voetnoot 1. Tot op heden was het alleen mogelijk om een nieuw type te baseren op één ander bestaand type. Hiervoor werd de stored procedure sp_addtype gebruikt. Omdat het zelfde en nog meer nu bereikt kan worden met de CREATE TYPE syntax zal het sp_addtype commando komen te vervallen.

```
[SqlUserDefinedType(Format.UserDefined, MaxByteSize = 124)]
public class Address :IBinarySerialize, INullable
{
  private string _street;
  private int _number;
  private string _country;
  private string _city;
  private bool _isNull;
  private const char cSep = ';';

  public Address()
  {
    this._isNull = true;
    this._street = "";
    this._number = 0;
    this._country = "";
    this._city = "";
  }
  #region IBinarySerialize Members

  public void Read(System.IO.BinaryReader r)
  {
    _street = r.ReadString();
    _number = r.ReadInt32();
    _country = r.ReadString();
    _city = r.ReadString();
    _isNull = BitConverter.ToBoolean(r.ReadBytes(1),0);
  }

  public void Write(System.IO.BinaryWriter w)
  {
    w.Write(_street);
    w.Write(_number);
    w.Write(_country);
    w.Write(_city);
    w.Write(false);
  }
  #endregion

  public override string ToString()
  {
    if (IsNull)
    {
      return "null";
    }
    else
    {
      return (Street + cSep + Number + cSep + Country
        + cSep + City);
    }
  }

  public static Address Parse(SqlString addressString)
  {
    if (addressString.IsNull)
    {
      return null;
    }
    else
    {
      Address newAddress = new Address();
      string[] content =
        addressString.ToString().Split(cSep);

      newAddress.Street = content[0];
      newAddress.Number = int.Parse(content[1]);
      newAddress.Country = content[2];
      newAddress.City = content[3];
      newAddress.IsNull = false;

      return (newAddress);
    }
  }
  #region INullable Members

  public bool IsNull
  {
    get
    {
      return _isNull;
    }
    set
    {
      _isNull = value;
    }
  }
  #endregion

  public static Address Null
  {
    get
    {
      Address nullAddress = new Address();
      nullAddress._isNull = true;

      return (nullAddress);
    }
  }
  #region public properties
  …
}
```

Codevoorbeeld 7

Met dit attribuut geven we aan dat het een user-defined type is en ook welke vorm van serialisatie we willen gebruiken. Wanneer het abstracte datatype alleen value-types bevat, kunnen we de native serialisatie (*Format.Native*) gebruiken. In ons geval bevat het *Address*-type ook reference-types (string) en zullen we dus zelf de serialisatie moeten verzorgen (*Format.UserDefined*). Dat doen we door *IBinarySerialize* (uit *System.Data.Sql*) te implementeren. Ook geven we aan wat de maximale omvang is van een instance met de *MaxByteSize*-parameter. De *IBinarySerialize* interface bevat een *Write()*- en *Read()*-methode die respectievelijk verantwoordelijk zijn voor het serialiseren en deserialiseren van het object. Deze methodes worden gebruikt voor de interne opslag van het type in SQL Server. Vervolgens implementeren we een static *Parse()*-functie en een *ToString()*-functie om serialisatie en deserialisatie van en naar een string te verzorgen. De *Parse()*-functie wordt gebruikt wanneer we vanuit bijvoorbeeld TSQL een object willen creëren aan de hand van een string. De *ToString()*-functie daarentegen wordt gebruikt wanneer we de inhoud van het object willen weergeven. Als laatste dienen we ervoor te zorgen dat we kunnen omgaan met null-values. Hiervoor implementeren we INullable (uit *System.Data.SqlTypes*) en definiëren we een static property Null die gebruikt kan worden om een null value te representeren. Nu hebben we de implementatie van het abstrace type gereed en kunnen we het type registreren en gebruiken. In codevoorbeeld 8 zien we de TSQL-statements die hiervoor nodig zijn en het resultaat.

Het *address*-type kunnen we nu verder in SQL Server gebruiken zoals we gewend zijn met andere typen. We kunnen bijvoorbeeld een tabel definiëren met een kolom van het *Address*-type. De TSQL-statements in codevoorbeeld 9 laten het gebruik en resultaat van het .NET-type zien.

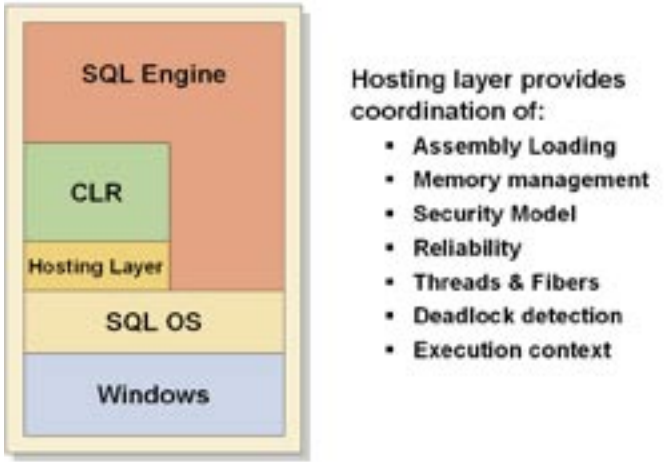
Er zijn nog meer krachtige mogelijkheden met CLR user-defined types, maar dat valt buiten de scope van dit artikel. Meer informatie hierover kan je vinden in de bijgeleverde helpdocumentatie van SQL Server 2005 (SQL Server 2005 Books Online, onderwerp *CLR User-Defined Types*).

SQL Server 2005 heeft de touwtjes in handen

Tot nu toe hebben we verschillende toepassingen laten zien van de integratie tussen SQL Server en de CLR. In deze paragraaf komt de interne werking van de server aan de orde. Om .NET-functionaliteit te kunnen uitvoeren heeft SQL Server natuurlijk een CLR nodig. Op het moment dat voor het eerst .NET-functionaliteit uitgevoerd moet worden, zal SQL Server de runtime laden. Het SQL Server-proces (*sqlservr.exe*) treedt dan ook op als host van de runtime. De hosting API's van de CLR 2.0 zijn uitgebreid, zodat een host meer invloed kan uitoefenen op de lopende managed code. Hiermee is het mogelijk voor SQL Server om efficiënt en veilig .NET-code uit te voeren. Zie ook afbeelding 1.

Ten gunste van de efficiency is het belangrijk dat SQL Server niet vecht met de .NET-code om dezelfde resources zoals CPU of memory. Om dit te voorkomen kan SQL Server bijvoorbeeld bepaalde .NET threads, die te veel resources alloceren, een lagere prioriteit geven. Daarnaast kan SQL Server detecteren wanneer er een garbage collection plaatsvindt in de CLR. SQL Server kan op dat moment besluiten ook allerlei managementtaken uit te voeren die geen CLR vereisen. Door deze hechte integratie blijft SQL Server bepalen wat er gebeurt en kan het zelf een optimum bepalen van de beschikbare resources.

Voor de veiligheid is het belangrijk dat SQL Server controle heeft over wat de .NET-code doet. Vanuit .NET is het natuurlijk eenvoudig om naar het lokale filesysteem te schrijven of wijzigingen te maken in de registry. In een gecontroleerde database-omgeving zijn dit acties die we juist willen inperken. Om de mogelijkheden van de .NET-code in te perken zijn er twee mechanismen beschikbaar: *code access security* en *host-protection attributes*. Code Access Security (CAS) is een mechanisme dat we natuurlijk al kennen uit de begindagen van .NET. Met CAS wordt afgedwongen dat een assembly alleen bepaalde acties mag uitvoeren - bijvoorbeeld toegang tot de registry of het filesysteem - wanneer de assembly beschikt over de juiste privileges. In codevoorbeeld 10 zien we een functie die naar het filesysteem probeert te schrijven. De bijbehorende registratie in SQL Server is in codevoorbeeld 11 te vinden. Wanneer we deze TSQL-statements uitvoeren krijgen we de volgende foutmelding:



Afbeelding 1. CLR integratie in SQL Server

```
-- registreer het type
create type Address
external NAME CLR.[SQLCLR.Address]

-- construeer een object
declare @ad Address
set @ad.Street = 'North Ave'
set @ad.Number = 2
set @ad.Country = 'United States'
set @ad.City = 'New York'

-- print het object
print CAST(@ad as nvarchar(100))
```

Resultaat

```
North Ave;2;United States;New York
```

Codevoorbeeld 8

```
-- creeer tabel op basis van address type
create table users (
  id int,
  name nvarchar(50),
  address Address
)

-- insert data
insert
into users (id, name, address)
values (1, 'Ann Davidson', CONVERT(Address,
        'North Ave;1;New York;United States'))

-- selecteer data
select id, name, adres = convert(nvarchar(100), address)
from users
```

Resultaat

```
id          name                 adres
--------    -------------------- -------------------------------------
1           Ann Davidson         North Ave;1;New York;United States
1 row(s) affected)
```

Codevoorbeeld 9.

```
public static bool WriteToFile(string text)
{
StreamWriter writer =
File.CreateText(@"c:\temp\test.txt");

writer.WriteLine(text);

writer.Close();

return (true);
}
```

Codevoorbeeld 10.

```
create function WriteToFile(
  @text nvarchar(100)
)
returns bit
as external NAME CLR.[SQLCLR.custom].WriteToFile

Select dbo.WriteToFile('hello world')
```

Codevoorbeeld 11.

```
create assembly clr
from 'c:\erikven\sqlclr.dll'
with permission_set = external_access
```

Codevoorbeeld 12.

	EXECUTE	FileIOPermission	RegistryPermission	SocketPermission	EnvironmentPermission	UIPermission	IsolatedStorageFilePermission	EventLogPermission	FileDialogPermission	ReflectionPermission	DnsPermission	...
SAFE	✓											
EXTERNAL_ACCESS	✓	✓	✓ (read)	✓	✓ (get)							
UNSAFE	✓	✓	✓ (write)	✓	✓ (set)	✓	✓	✓	✓	✓	✓	✓

Tabel 1.

System.Security.SecurityException: Request for the permission of type 'System.Security.Permissions.FileIOPermission, mscorlib, Version=2.0.3600.0, Culture=neutral, PublicKeyToken=b77a5c561934e089' failed.

Hoewel lokale assemblies standaard over alle privileges beschikken (fulltrust), is het toch niet mogelijk om vanuit SQL Server het filesysteem te benaderen. Dit komt doordat SQL Server de bestaande levels van security-policy (enterprise, machine en user) uitbreidt met een SQL Server-specifieke hostlevel-policy. De uiteindelijke set van permissies wordt bepaald door de doorsnede te nemen van deze vier levels. Het *FileIOPermission*-privilege is ons dus ontnomen door SQL Server. We moeten dus ook op SQL Server hostpolicy-niveau dit recht geven aan de assembly. Hiervoor kent SQL Server drie voorgedefinieerde permissiesets, namelijk *Safe*, *External_Access* en *Unsafe*. De safe permissieset is de default en het meest restrictief. De tabel toont deze restricties.

Om de privileges van een assembly uit te breiden, dienen we tijdens de registratie een permissieset mee te geven. In codevoorbeeld 12 kunnen we zien hoe we de permissieset *external_access* meegeven aan de assembly.
Hiermee beschikt de assembly wel over het *FileIOPermission*-privilege zoals te lezen valt uit de tabel. Wanneer we opnieuw de functie aanroepen functioneert onze code wel correct.

Een tweede uitbreiding ten gunste van de veiligheid is een nieuw concept met de naam *host-protection-attributes* (HPA's). Zoals de naam al doet vermoeden zijn dit attributen die een host kan gebruiken om zichzelf te beschermen tegen potentieel gevaarlijke code. Zo kan SQL Server beslissen geen code te laden waarmee thread-synchronisatie kan worden uitgevoerd. Als voorbeeld hiervan kunnen we zien dat het *System.Threading.Mutex* type in .NET 2.0 is voorzien van een host protection-attribuut. SQL Server kan dit attribuut gebruiken om code van tevoren te inspecteren en vervolgens besluiten deze code te weigeren.

De assemblies worden bewaard in SQL Server 2005

Ten slotte kijken we nog naar de interne opslag van de assemblies in SQL Server. Via registratie van een assembly (create assembly) wordt de volledige assembly bewaard in de database. Na registratie kunnen we dus de assembly van het filesysteem verwijderen. Alle informatie die SQL Server nodig heeft, is te vinden in de database zelf. We kunnen de catalog view *sys.assemblies* raadplegen om algemene informatie te vinden over de assemblies. De feitelijke inhoud van de assembly is te vinden met de view *sys. assembly_files*.

De krachten gebundeld

Door de hechte integratie tussen SQL Server 2005 en de .NET runtime wordt het mogelijk database-objecten als functies, stored procedures en triggers te ontwikkelen in een willekeurige .NET-taal. De kracht, eenvoud en veiligheid van .NET komt hiermee ook beschikbaar binnen Microsoft SQL Server.

Erik S.C. van de Ven is senior consultant bij Microsoft Services Nederland. Zijn e-mailadres is erikven@microsoft.com

Nuttige internetadressen

SQL Server 2005 algemeen - www.microsoft.com/sql/2005
SQL Server Development center – http://msdn.microsoft.com/sql
SQL 2005 programming basics - http://msdn.microsoft.com/msdnmag/issues/04/02/yukonbasics
Security permissions - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemSecurityPermissions.asp
Professional Association for SQL Server - www.sqlpass.org