CLR Triggers

Source	http://msdn2.microsoft.com/en-us/library/ms131093.aspx

CLR Triggers

Because of the Microsoft SQL Server integration with the Microsoft .NET Framework common language runtime (CLR), you can use any .NET Framework language to create CLR triggers. This section covers information specific to triggers implemented with CLR integration. For a complete discussion of triggers, see Understanding DML Triggers and Understanding DDL Triggers.

What Are Triggers?

A trigger is a special type of stored procedure that automatically runs when a language event executes. SQL Server includes two general types of triggers: data manipulation language (DML) and data definition language (DDL) triggers. DML triggers can be used when INSERT, UPDATE, or DELETE statements modify data in a specified table or view. DDL triggers fire stored procedures in response to a variety of DDL statements, which are primarily statements that begin with CREATE, ALTER, and DROP. DDL triggers can be used for administrative tasks, such as auditing and regulating database operations.

Unique Capabilities of CLR Triggers

Triggers written in Transact-SQL have the capability of determining which columns from the firing view or table have been updated by using the UPDATE(column) and COLUMNS_UPDATED() functions.

Triggers written in a CLR language differ from other CLR integration objects in several significant ways. CLR triggers can:

- Reference data in the INSERTED and DELETED tables
- Determine which columns have been modified as a result of an UPDATE operation
- Access information about database objects affected by the execution of DDL statements.

These capabilities are provided inherently in the query language, or by the SqlTriggerContext class. For information about the advantages of CLR integration and choosing between managed code and Transact-SQL, see Overview of CLR Integration.

Using the SqlTriggerContext Class

The SqlTriggerContext class cannot be publicly constructed and can only be obtained by accessing the SqlContext.TriggerContext property within the body of a CLR trigger. The SqlTriggerContext class can be obtained from the active SqlContext by calling the SqlContext.TriggerContext property:

SqlTriggerContext myTriggerContext = SqlContext.TriggerContext;

The SqlTriggerContext class provides context information about the trigger. This contextual information includes the type of action that caused the trigger to fire, which columns were modified in an UPDATE operation, and, in the case of a DDL trigger, an XML EventData structure which describes the triggering operation. For more information, see EVENTDATA (Transact-SQL).

Determining the Trigger Action

Once you have obtained a SqlTriggerContext, you can use it to determine the type of action that caused the trigger to fire. This information is available through the TriggerAction property of the SqlTriggerContext class.

For DML triggers, the TriggerAction property can be one of the following values:

- TriggerAction.Update (0x1)
- TriggerAction.Insert (0x2)
- TriggerAction.Delete(0x3)
- For DDL triggers, the list of possible TriggerAction values is considerably longer. Please see "TriggerAction Enumeration" in the .NET Framework SDK for more information.

Using the Inserted and Deleted Tables

Two special tables are used in DML trigger statements: the inserted table and the deleted table. SQL Server 2005 automatically creates and manages these tables. You can use these temporary tables to test the effects of certain data modifications and to set conditions for DML trigger actions; however, you cannot alter the data in the tables directly.

CLR triggers can access the inserted and deleted tables through the CLR in-process provider. This is done by obtaining a SqlCommand object from the SqlContext object. For example:

C#

```
SqlConnection connection = new SqlConnection ("context connection =
true");

connection.Open();

SqlCommand command = connection.CreateCommand();

command.CommandText = "SELECT * from " + "inserted";
```

Visual Basic

```
Dim connection As New SqlConnection("context connection=true")

Dim command As SqlCommand

connection.Open()

command = connection.CreateCommand()

command.CommandText = "SELECT * FROM " + "inserted"
```

Determining Updated Columns

You can determine the number of columns that were modified by an UPDATE operation by using the ColumnCount property of the SqlTriggerContext object. You can use the IsUpdatedColumn method, which takes the column ordinal as an input parameter, to determine whether the column was updated. A True value indicates that the column has been updated.

For example, this code snippet (from the EmailAudit trigger later in this topic) lists all of the columns updated:

C#

```
reader = command.ExecuteReader();

reader.Read();
```

```
for (int columnNumber = 0; columnNumber < triggContext.ColumnCount;
columnNumber++)

{

   pipe.Send("Updated column "

      + reader.GetName(columnNumber) + "? "

   + triggContext.IsUpdatedColumn(columnNumber).ToString());

 }

reader.Close();
```

Visual Basic

```
reader = command.ExecuteReader()

reader.Read()

Dim columnNumber As Integer


For columnNumber=0 To triggContext.ColumnCount-1


   pipe.Send("Updated column " & reader.GetName(columnNumber) & _

   "? " & triggContext.IsUpdatedColumn(columnNumber).ToString() )

Next

reader.Close()
```

Accessing EventData for CLR DDL Triggers

DDL triggers, like regular triggers, fire stored procedures in response to an event. But unlike DML triggers, they do not fire in response to UPDATE, INSERT, or DELETE statements on a table or view. Instead, they fire in response to a variety of DDL statements, which are primarily statements that begin with CREATE, ALTER, and DROP. DDL triggers can be used for administrative tasks, such as auditing and monitoring of database operations and schema changes.

Information about an event that fires a DDL trigger is available in the EventData property of the SqlTriggerContext class. This property contains an xml value. The xml schema includes information about:

- **The time of the event.**
- **The System Process ID (SPID) of the connection during which the trigger executed.**
- **The type of event that fired the trigger.**

Then, depending on the event type, the schema includes additional information, such as the database in which the event occurred, the object against which the event occurred, and the Transact-SQL command of the event.

In the following example, the following DDL trigger returns the raw EventData property.

Note: Sending results and messages through the SqlPipe object is shown here for illustrative purposes only and is generally discouraged for production code when programming CLR triggers. Additional data returned may be unexpected and lead to application errors.

C#

```
using System;

using System.Data;

using System.Data.Sql;

using Microsoft.SqlServer.Server;

using System.Data.SqlClient;

using System.Data.SqlTypes;

using System.Xml;

using System.Text.RegularExpressions;


public class CLRTriggers

{

   public static void DropTableTrigger()
   {
       SqlTriggerContext triggContext = SqlContext.TriggerContext;


       switch(triggContext.TriggerAction)
       {
           case TriggerAction.DropTable:
           SqlContext.Pipe.Send("Table dropped! Here's the
EventData:");
           SqlContext.Pipe.Send(triggContext.EventData.Value);
           break;


           default:
           SqlContext.Pipe.Send("Something happened! Here's the
EventData:");
           SqlContext.Pipe.Send(triggContext.EventData.Value);
           break;
       }
   }
}
```

Visual Basic

```
Imports System
Imports System.Data
Imports System.Data.Sql
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server
Imports System.Data.SqlClient
```

```
'The Partial modifier is only required on one class definition per
project.
Partial Public Class CLRTriggers

    Public Shared Sub DropTableTrigger()
        Dim triggContext As SqlTriggerContext
        triggContext = SqlContext.TriggerContext

        Select Case triggContext.TriggerAction
           Case TriggerAction.DropTable
              SqlContext.Pipe.Send("Table dropped! Here's the
EventData:")
              SqlContext.Pipe.Send(triggContext.EventData.Value)

           Case Else
              SqlContext.Pipe.Send("Something else happened! Here's
the EventData:")
              SqlContext.Pipe.Send(triggContext.EventData.Value)

        End Select
    End Sub
End Class
```

The following sample output is the EventData property value after a DDL trigger fired by a CREATE TABLE event:

```
<EVENT_INSTANCE><PostTime>2004-04-
16T21:17:16.160</PostTime><SPID>58</SPID><EventType>CREATE_TABLE</EventType
><ServerName>MACHINENAME</ServerName><LoginName>MYDOMAIN\myname</Login
Name><UserName>MYDOMAIN\myname</UserName><DatabaseName>AdventureWorks<
/DatabaseName><SchemaName>dbo</SchemaName><ObjectName>UserName</ObjectN
ame><ObjectType>TABLE</ObjectType><TSQLCommand><SetOptions
ANSI_NULLS="ON" ANSI_NULL_DEFAULT="ON" ANSI_PADDING="ON"
QUOTED_IDENTIFIER="ON" ENCRYPTED="FALSE" /><CommandText>create table
```

dbo.UserName&#x0D;&#x0A;(&#x0D;&#x0A; UserName varchar(50),&#x0D;&#x0A; RealName varchar(50)&#x0D;&#x0A;)&#x0D;&#x0A;</CommandText></TSQLCommand></EVENT_INSTANCE>

In addition to the information accessible through the **SqlTriggerContext** class, queries can still refer to **COLUMNS_UPDATED** and inserted/deleted within the text of a command executed in-process.

Sample CLR Trigger

In this example, consider the scenario in which you let the user choose any ID they want, but you want to know the users that specifically entered an e-mail address as an ID. The following trigger would detect that information and log it to an audit table.

C#

```
using System;
using System.Data;
using System.Data.Sql;
using Microsoft.SqlServer.Server;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using System.Xml;
using System.Text.RegularExpressions;

public class CLRTriggers
{
   [SqlTrigger(Name = @"EmailAudit", Target = "[dbo].[Users]", Event =
"FOR INSERT, UPDATE, DELETE")]
   public static void EmailAudit()
   {
      string userName;
      string realName;
      SqlCommand command;
      SqlTriggerContext triggContext = SqlContext.TriggerContext;
      SqlPipe pipe = SqlContext.Pipe;
      SqlDataReader reader;

      switch (triggContext.TriggerAction)
      {
         case TriggerAction.Insert:
         // Retrieve the connection that the trigger is using
         using (SqlConnection connection
            = new SqlConnection(@"context connection=true"))
         {
            connection.Open();
            command = new SqlCommand(@"SELECT * FROM INSERTED;",
               connection);
            reader = command.ExecuteReader();
            reader.Read();
            userName = (string)reader[0];
            realName = (string)reader[1];
            reader.Close();

            if (IsValidEMailAddress(userName))
            {
               command = new SqlCommand(
                  @"INSERT [dbo].[UserNameAudit] VALUES ('"
                  + userName + @"', '" + realName + @"');",
                  connection);
               pipe.Send(command.CommandText);
               command.ExecuteNonQuery();
               pipe.Send("You inserted: " + userName);
            }
         }

         break;
         case TriggerAction.Update:
         // Retrieve the connection that the trigger is using
         using (SqlConnection connection
            = new SqlConnection(@"context connection=true"))
         {
            connection.Open();
            command = new SqlCommand(@"SELECT * FROM INSERTED;",
               connection);
            reader = command.ExecuteReader();
            reader.Read();

            userName = (string)reader[0];
            realName = (string)reader[1];

            pipe.Send(@"You updated: '" + userName + @"' - '"
               + realName + @"'");

            for (int columnNumber = 0; columnNumber <
triggContext.ColumnCount; columnNumber++)
            {
               pipe.Send("Updated column "
                  + reader.GetName(columnNumber) + "? "
                  +
triggContext.IsUpdatedColumn(columnNumber).ToString());
            }

            reader.Close();
         }

         break;
         case TriggerAction.Delete:
            using (SqlConnection connection
               = new SqlConnection(@"context connection=true"))
               {
                  connection.Open();
                  command = new SqlCommand(@"SELECT * FROM DELETED;",
                     connection);
                  reader = command.ExecuteReader();

                  if (reader.HasRows)
                  {
                     pipe.Send(@"You deleted the following rows:");
                     while (reader.Read())
                     {
                        pipe.Send(@"'" + reader.GetString(0)
                        + @"', '" + reader.GetString(1) + @"'");
                     }

                     reader.Close();

                     //alternately, to just send a tabular resultset
back:
                     //pipe.ExecuteAndSend(command);
                  }
                  else
                  {
                     pipe.Send("No rows affected.");
                  }
               }

               break;
            }
```

```
        }

     public static bool IsValidEMailAddress(string email)
     {
         return Regex.IsMatch(email, @"^([\w-]+\.)*?[\w-]+@[\w-
]+\.([\w-]+\.)*?[\w]+$");
     }
}
```

Visual Basic

```
Imports System
Imports System.Data
Imports System.Data.Sql
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server
Imports System.Data.SqlClient
Imports System.Text.RegularExpressions

'The Partial modifier is only required on one class definition per
project.
Partial Public Class CLRTriggers

    <SqlTrigger(Name:="EmailAudit", Target:="[dbo].[Users]",
Event:="FOR INSERT, UPDATE, DELETE")> _
    Public Shared Sub EmailAudit()
        Dim userName As String
        Dim realName As String
        Dim command As SqlCommand
        Dim triggContext As SqlTriggerContext
        Dim pipe As SqlPipe
        Dim reader As SqlDataReader
        triggContext = SqlContext.TriggerContext
        pipe = SqlContext.Pipe

        Select Case triggContext.TriggerAction
           Case TriggerAction.Insert
              Using connection As New SqlConnection("context
connection=true")
                 connection.Open()
                 command = new SqlCommand("SELECT * FROM INSERTED;",
connection)

                 reader = command.ExecuteReader()
                 reader.Read()

                 userName = CType(reader(0), String)
                 realName = CType(reader(1), String)

                 reader.Close()

                 If IsValidEmailAddress(userName) Then
                     command = New SqlCommand("INSERT
[dbo].[UserNameAudit] VALUES ('" & _
                       userName & "', '" & realName & "');",
connection)

                    pipe.Send(command.CommandText)
                    command.ExecuteNonQuery()
                    pipe.Send("You inserted: " & userName)

                 End If
              End Using
           Case TriggerAction.Update
              Using connection As New SqlConnection("context
connection=true")
                 connection.Open()
                 command = new SqlCommand("SELECT * FROM INSERTED;",
connection)

                 reader = command.ExecuteReader()
                 reader.Read()

                 userName = CType(reader(0), String)
                 realName = CType(reader(1), String)

                 pipe.Send("You updated: " & userName & " - " &
realName)

                 Dim columnNumber As Integer

                 For columnNumber=0 To triggContext.ColumnCount-1

                    pipe.Send("Updated column " &
reader.GetName(columnNumber) & _
                      "? " &
triggContext.IsUpdatedColumn(columnNumber).ToString() )

                 Next

                 reader.Close()
              End Using

           Case TriggerAction.Delete
```

```
              Using connection As New SqlConnection("context
connection=true")
                 connection.Open()
                 command = new SqlCommand("SELECT * FROM DELETED;",
connection)

                 reader = command.ExecuteReader()

                 If reader.HasRows Then
                    pipe.Send("You deleted the following rows:")

                    While reader.Read()

                       pipe.Send( reader.GetString(0) & ", " &
reader.GetString(1) )

                    End While

                    reader.Close()

                    ' Alternately, just send a tabular resultset back:
                    ' pipe.ExecuteAndSend(command)

                 Else
                   pipe.Send("No rows affected.")
                 End If

              End Using
        End Select
    End Sub
    Public Shared Function IsValidEMailAddress(emailAddress As String)
As Boolean

       return Regex.IsMatch(emailAddress, "^([\w-]+\.)*?[\w-]+@[\w-
]+\.([\w-]+\.)*?[\w]+$")
    End Function
End Class
```

Assuming two tables exist with the following definitions:

```
CREATE TABLE Users
(
    UserName nvarchar(200) NOT NULL,
    RealName nvarchar(200) NOT NULL
);
GO CREATE TABLE UserNameAudit
(
    UserName nvarchar(200) NOT NULL,
    RealName nvarchar(200) NOT NULL
)
```

The Transact-SQL statement that creates the trigger in SQL Server is as follows, and assumes assembly SQLCLRTest is already registered in the current SQL Server database.

```
CREATE TRIGGER EmailAudit

ON Users

FOR INSERT, UPDATE, DELETE

AS

EXTERNAL NAME SQLCLRTest.CLRTriggers.EmailAudit
```

On a SQL Server 2005 database with a compatibility level of "80," you cannot create managed user-defined types, stored procedures, functions, aggregates, or triggers. To take advantage of these CLR integration features of SQL Server 2005, you must use the sp_dbcmptlevel (Transact-SQL) stored procedure to set the database compatibility level to "90.".

Validating and Cancelling Invalid Transactions

Using triggers to validate and cancel invalid INSERT, UPDATE, or DELETE transactions or to prevent changes to your database schema is common. This can be accomplished by incorporating validation logic into your trigger and then rolling back the current transaction if the action does not meet the validation criteria.

When called within a trigger, the Transaction.Rollback method or a SqlCommand with the command text "TRANSACTION ROLLBACK" throws an exception with an ambiguous error message and must be wrapped in a try/catch block. The error message you see is similar to the following:

Msg 6549, Level 16, State 1, Procedure trig_InsertValidator, Line 0

A .NET Framework error occurred during execution of user defined routine or aggregate 'trig_InsertValidator':

System.Data.SqlClient.SqlException: Transaction is not allowed to roll back inside a user defined routine, trigger or aggregate because the transaction is not started in that CLR level. Change application logic to enforce strict transaction nesting... User transaction, if any, will be rolled back.

This exception is expected and the try/catch block is necessary for code execution to continue. When the trigger code finishes execution, another exception is raised

Msg 3991, Level 16, State 1, Procedure trig_InsertValidator, Line 1

The context transaction which was active before entering user defined routine, trigger or aggregate "trig_InsertValidator" has been ended inside of it, which is not allowed. Change application logic to enforce strict transaction nesting.

The statement has been terminated.

This exception is also expected, and a try/catch block around the Transact-SQL statement that performs the action that fires the trigger is necessary so that execution can continue. Despite the two exceptions thrown, the transaction is rolled back and the changes are not committed to the table. A major difference between CLR triggers and Transact-SQL triggers is that Transact-SQL triggers can continue to perform more work after the transaction is rolled back.

Example

The following trigger performs simple validation of INSERT statements on a table. If the inserted integer value is equal to one, the transaction is rolled back and the value is not

inserted into the table. All other integer values are inserted into the table. Note the try/catch block around the `Transaction.Rollback` method. The Transact-SQL script creates a test table, assembly, and managed stored procedure. Note that the two INSERT statements are wrapped in a try/catch block so that the exception thrown when the trigger finishes execution is caught.

C#

```
using System;

using System.Data.SqlClient;

using Microsoft.SqlServer.Server;

using System.Transactions;


public partial class Triggers

{

    // Enter existing table or view for the target and uncomment the
attribute line

    // [Microsoft.SqlServer.Server.SqlTrigger
(Name="trig_InsertValidator", Target="Table1", Event="FOR INSERT")]

    public static void trig_InsertValidator()

    {

        using (SqlConnection connection = new SqlConnection(@"context
connection=true"))

        {

            SqlCommand command;

            SqlDataReader reader;

            int value;


            // Open the connection.

            connection.Open();
            // Get the inserted value.
            command = new SqlCommand(@"SELECT * FROM INSERTED",
connection);
            reader = command.ExecuteReader();
            reader.Read();
            value = (int)reader[0];
            reader.Close();

            // Rollback the transaction if a value of 1 was inserted.
            if (1 == value)
            {
                try
                {
                    // Get the current transaction and roll it back.
                    Transaction trans = Transaction.Current;
                    trans.Rollback();
                }
                catch (SqlException ex)
                {
                    // Catch the expected exception.
                }
            }
            else
            {
                // Perform other actions here.
            }

            // Close the connection.
            connection.Close();
        }
    }
}
```

Visual Basic

```
Imports System
Imports System.Data.SqlClient
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server
Imports System.Transactions

Partial Public Class Triggers
' Enter existing table or view for the target and uncomment the
attribute line
' <Microsoft.SqlServer.Server.SqlTrigger(Name:="trig_InsertValidator",
Target:="Table1", Event:="FOR INSERT")> _
Public Shared Sub  trig_InsertValidator ()
    Using connection As New SqlConnection("context connection=true")

        Dim command As SqlCommand
        Dim reader As SqlDataReader
        Dim value As Integer
```

```
' Open the connection.
connection.Open()

' Get the inserted value.
command = New SqlCommand("SELECT * FROM INSERTED", connection)
reader = command.ExecuteReader()
reader.Read()
value = CType(reader(0), Integer)
reader.Close()

' Rollback the transaction if a value of 1 was inserted.
If value = 1 Then

    Try
        ' Get the current transaction and roll it back.
        Dim trans As Transaction
        trans = Transaction.Current
        trans.Rollback()

    Catch ex As SqlException

        ' Catch the exception.
    End Try
Else
```

```
            ' Perform other actions here.
        End If

        ' Close the connection.
        connection.Close()
    End Using
End Sub
End Class
```

Transact-SQL

```
-- Create the test table, assembly, and trigger.
create table Table1(c1 int);
go

CREATE ASSEMBLY ValidationTriggers from 'E:\programming\
ValidationTriggers.dll';
go

CREATE TRIGGER trig_InsertValidator
ON Table1
FOR INSERT
AS EXTERNAL NAME ValidationTriggers.Triggers.trig_InsertValidator;
go

-- Use a Try/Catch block to catch the expected exception
BEGIN TRY
```

```
   insert into Table1 values(42)

   insert into Table1 values(1)

END TRY

BEGIN CATCH

  SELECT ERROR_NUMBER() AS ErrorNum, ERROR_MESSAGE() AS ErrorMessage

END CATCH;


-- Clean up.

DROP TRIGGER trig_InsertValidator;

DROP ASSEMBLY ValidationTriggers;

drop table Table1;
```

***~~~ End of Article ~~~***