Laboratorium 7 – MS SQL Server

Temat: Środowisko CLR w MS SQL Server - wprowadzenie

Opracowanie: A.Dydejczyk, A.Lemański

Ćwiczenia do opracowania w trakcie laboratorium:

1. Wprowadzenie do technologii CLR
2. Ćwiczenie A. Pierwsza funkcja UDF w CLR
3. Ćwiczenie B. Porównanie funkcji w języku T-SQL i technologii CLR
4. Ćwiczenie C. Funkcja UDF w języku C#.
5. Ćwiczenie D. Przestrzeń nazw System.Data.SqlTypes.
6. Ćwiczenie E. Procedura zwracająca czas z systemu SQL.
7. Ćwiczenie F. Przestrzeń nazw System.Data.SqlDbType.
8. Zadania.

Wprowadzenie

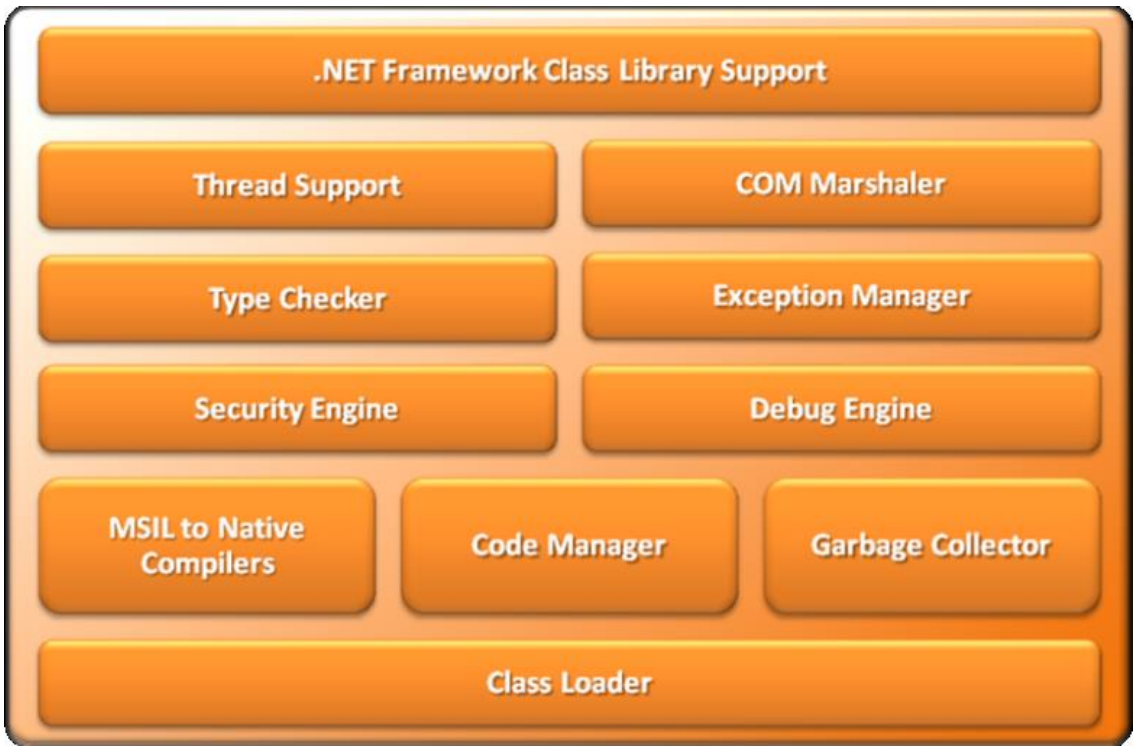
Integracja SQL Server z CLR polega na tym, że SQL Server jest hostem dla CLR. Oznacza to, że to SQL Server zarządza takimi procesami jak przyznawanie pamięci obiektom CLR, oczyszczanie pamięci z nieużywanych obiektów (garbage collection), obsługa żądań SQL zgłaszanych do procesora zapytań, zarządzanie domenami aplikacji CLR tak, by owe aplikacje były od siebie niezależne i odseparowane (żeby jedna aplikacja CLR nie miała wpływu na działanie innych aplikacji). Ma to pozytywne konsekwencje, ponieważ dzięki takiemu podejściu możliwe jest chociażby uniknięcie konfliktów w dostępie do pamięci i zasobów procesora, do jakich mogłoby dochodzić, gdyby to system operacyjny, a nie SQL Server przydzielał zasoby obiektom CLR.

Opis technologii można znaleźć na stronie Microsoft pod adresami:
https://msdn.microsoft.com/en-us/library/ms131102%28v=sql.105%29.aspx
https://msdn.microsoft.com/en-us/library/ms131102.aspx
https://technet.microsoft.com/en-us/library/ms345136%28v=sql.90%29.aspx

Common Language Runtime (CLR) to środowisko uruchomieniowe dla aplikacji stworzonych na platformie .NET. Środowisko to bardzo często nazywa się środowiskiem zarządzanym (ang. managed environment) z uwagi na duży stopień automatyzacji zarządzania zasobami. CLR posiada wiele wbudowanych mechanizmów wspomagających programowanie, m.in.:

- **Garbage Collector** – mechanizm automatycznego zarządzania czasem życia obiektów,
- **Class Loader** – mechanizm zarządzający metadanym i ładowaniem klas,
- **Exception Manager** – moduł dostarczający strukturalną obsługę wyjątków zintegrowaną ze strukturalną obsługą wyjątków w systemach Microsoft Windows,
- **Kompilatory Just-In-Time** (JIT) – kompilatory służące do kompilacji kodu będącego w postaci pośredniej (Microsoft Intermediate Language – MSIL) do kodu natywnego charakterystycznego dla platformy, na której został zainstalowany SQL Server,

- Mechanizmy zarządzania bezpieczeństwem – **Code Access Security**, czyli zarządzanie możliwością uruchamiania kodu w zależności od kontekstu użytkownika i pochodzenia kodu.



Rys. 1. Komponenty Common Language Runtime.

Praca z technologią .NET sprowadza się do tworzenia pewnej struktury klas przy użyciu języków wysokiego poziomu. Produktem kompilacji takiego kodu jest plik (lub zbiór plików) nazywany *assembly*. *Assemlby* składa się z dwóch części: kodu skompilowanego do języka pośredniego MSIL oraz tak zwanego manifestu – zbioru metadanych opisujących samo *assembly* (opis klas, metod i innych elementów kodu znajdujących się w *assembly*).

Od wersji SQL Server 2005 mamy możliwość uruchamiania zarządzanego kodu napisanego w jednym z języków oferowanych przez platformę .NET Framework z poziomu bazy danych. Programiści mają możliwość stworzenia następujących obiektów CLR w bazach danych:

- funkcji skalarnych lub tablicowych (*user-defined functions* – UDF),
- procedur składowanych (*stored procedures*),
- wyzwalaczy (*triggers*),
- agregatów (*user-defined aggregates* – UDA),
- typów (*user-defined types* – UDT).

Cel utworzenia funkcji, procedury lub wyzwalacza w kodzie zarządzanym jest łatwy do zrozumienia. Kod obiektu CLR jest uruchamiany i wykonywany tak samo, jak jego odpowiednik napisany w T-SQL. Inaczej jest w przypadku UDA i UDT. Rozszerzają one możliwości oferowane programistom baz danych w następujący sposób:

- UDA – Pozwala programistom na zbudowanie własnej funkcji agregującej, którą można używać w połączeniu z klauzulą GROUP BY w zapytaniu T-SQL. To pozwala na przeprowadzenie złożonych operacji statystycznych oraz analizę danych przy użyciu silnika baz danych. Co więcej, kodu agregatów nie można tworzyć inaczej, jak tylko używając technologii .NET. Nie ma możliwości stworzenia ich przy użyciu samego kodu języka T-SQL.
- UDT – Dostarcza programistom możliwość zdefiniowania nowych typów o określonym zachowaniu. Połączenie .NET Framework, bibliotek innych firm oraz SQL Server otwiera nowe możliwości w tworzeniu obiektów zamiast tworzenia ich relacyjnej reprezentacji. Dzięki temu można tworzyć chociażby typy złożone. Warto też dodać, że UDT stoją najwyżej w hierarchii typów w SQL Server czyli przy operacjach z udziałem UDT każdy typ istniejący w SQL Server będzie konwertowany niejawnie do UDT.

Dlaczego mówimy o tworzeniu obiektów w relacyjnej bazie danych? Na pierwszy rzut oka to rozwiązanie ma więcej wad niż zalet. Prawdą jest, że do typów UDT system ma dostęp jako całości, a więc użycie skomplikowanych obiektów biznesowych może spowodować obniżenie wydajności serwera. Zgodnie z rekomendacją firmy Microsoft należy przyjąć, że UDT w SQL Server zostały zaprojektowane dla obsługi:

- daty, czasu, waluty oraz rozszerzonych typów numerycznych,
- danych pochodzących np. z systemów GPS,
- szyfrowania / odszyfrowywania danych.

Zastosowania CLR

Do jakich zastosowań nadają się obiekty CLR? Poniżej staramy się odpowiedzieć na to pytanie.

Na początek zdefiniujmy sytuacje, w których zastosowanie CLR w SQL Server na pewno będzie błędem:

- Duża ilość danych pobierana do biblioteki CLR. Do operacji na zbiorach danych najlepiej nadają się zapytania T-SQL uruchamiane w silniku baz danych. Należy pamiętać o jeszcze jednym czynniku - operacje wykonywane na zbiorach w SQL Server zamieniane są na operacje na kursorach w aplikacjach pisanych w bibliotekach zewnętrznych.
- Długotrwałe odwołania do zewnętrznych bibliotek. Bardzo ważnym jest upewnienie się, iż działania użytkownika nie spowodują uruchomienia wywołań do zewnętrznych aplikacji lub API. Wpływ takich wywołań jest najsilniejszy w przypadku funkcji definiowanych przez użytkownika (UDF), które mogą być wywoływane dla każdego wiersza zwracanego w wyniku działania zapytania. Wywołanie zewnętrznej funkcji, które może zająć ok. 1 sekundy na wiersz będzie nieporozumieniem w przypadku konieczności powtórzenia tego kilka tysięcy razy.
- Nadużywanie w tworzeniu typów UDT oraz agregatów UDA. Podczas tworzenia własnego typu danych należy być świadomym ograniczeń wynikających ze sposobu przechowywania tego typu na stronie z danymi. Najsilniejsze ograniczenie nakładane na UDT mówi, że jego

wielkość musi być mniejsza niż 8kB. Drugim istotnym ograniczeniem, o którym należy pamiętać, jest fakt iż cały typ danych lub agregat musi zostać wczytany i przepisany w momencie jego aktualizacji (przez komendę UPDATE). Wynika z tego, iż nie można odczytywać tylko poszczególnych składników typu - jest on traktowany jako spójny obiekt.

- UDA i tworzenie raportów w czasie rzeczywistym. Agregaty tworzone przez użytkownika nie mogą być używane w połączeniu z indeksowanymi widokami w SQL Server, nie jest więc możliwe automatyczne wstępne preagregowanie danych dla poprawy wydajności dla raportów wykonywanych w czasie rzeczywistym, co znacznie poprawiłoby ich wydajność.
- Kompatybilność z poprzednią wersją SQL Server. Jeżeli aplikacja musi wspierać poprzednie wersje SQL Server to nie można stosować omawianej w tym artykule funkcjonalności.

Najistotniejsze są jednak zalety wynikające z integracji CLR z SQL Server, które powodują, iż jest to pożądane rozwiązanie i to nie tylko przy tworzeniu nowych systemów bazodanowych:

- Wykorzystanie bibliotek .NET Framework oraz środowiska programistycznego Visual Studio. W roku 2005 wydano zarówno SQL Server jak i Visual Studio .NET. Przed aplikacjami bazodanowymi otwarły się zupełnie nowe możliwości wynikające z zastosowania bibliotek .NET Framework. Dostęp do funkcjonalności w nich zawartych jest możliwy bez konieczności posiadania wysokich uprawnień na poziomie kodu.
- Zastępowanie rozszerzonych procedur składowanych (XPs). Poprzednio dostęp do zewnętrznych zasobów odbywał się przez wywołanie rozszerzonych procedur składowanych lub procedur z rodziny sp_OA*. Te metody niosą ze sobą olbrzymie ryzyko dla stabilności serwera w związku z uruchamianiem kodu niezarządzanego.

Rekomenduje się administratorom baz danych wykorzystanie integracji CLR i SQL Server głównie ze względu na aspekty bezpieczeństwa:

- Nie ma możliwości, aby uruchomienie zarządzanego kodu spowodowało zawieszenie silnika bazy danych a w efekcie jego zatrzymanie.
- Nie ma możliwości, aby uruchomienie kodu zarządzanego spowodowało wycieki pamięci a w efekcie spowolnienie i zawieszenie instancji SQL Server.
- Uzyskiwana jest większa wydajność i skalowalność ze względu na kontrolowanie pamięci, które daje SQL Server.
- Nie ma problemów związanych z bezpieczeństwem uruchomionego kodu, ponieważ modele bezpieczeństwa .NET Framework i w SQL Server są ze sobą zintegrowane.

Powyższe rozważania nie są prawdziwe, jeżeli *assembly* zostanie zarejestrowane w trybie UNSAFE, ponieważ wtedy jest możliwość wywołania zewnętrznego, niezarządzanego kodu.

Zredukowanie ruchu w sieci. Niektóre algorytmy wymagają przesłania dużej ilości danych dla wygenerowania odpowiedniego wyniku. Przesyłanie danych pomiędzy serwerami powoduje zwiększenie przepływu informacji w sieci. Umieszczenie algorytmów wewnątrz bazy danych nie tylko ogranicza ten ruch ale może również poprawić wydajność zauważalną w aplikacji klienckiej.

Przykładem takiego algorytmu może być konieczność wykonania specjalistycznych operacji statystycznych, które wymagają otrzymania całego zbioru danych przed rozpoczęciem obliczeń.

Pisanie funkcji ogólnego przeznaczenia. Funkcja ogólnego przeznaczenia jest zdefiniowana następująco:

- Dane są przekazywane do funkcji w postaci argumentów,
- Funkcja nie wymaga dodatkowego dostępu do danych,
- Złożone obliczenia wykonywane są w kodzie podobnym do działania kursora – jeden wiersz analizowany jest w jednym przebiegu pętli.

Przykładem takiej funkcji może być funkcja wyznaczająca odległość pomiędzy dwoma punktami otrzymanymi w wyniku monitorowania ruchu pojazdów. Wewnątrz takiej funkcji przeprowadzane są operacje trygonometryczne – sumy oraz iloczyny. Funkcja jako wynik mogłaby zwracać odległość w jednostce zgodnej z parametrem wejściowym – kilometrach, metrach, milach, jardach itp.

Definiowanie własnych skalarnych typów (UDT). Większość typów danych może być mapowana do modelu relacyjnego, jednak jest wiele przykładów, które nadają się do rozważenia jako nowe typy:

- Typy bazujące na istniejących w SQL Server, ale rozszerzające ich możliwości – np. implementacja typu *datetime* o funkcjonalność czasu UTC
- Typy, które są wczytywane do tablic i zapisywane w całości (jako obiekty). Takie typy ukrywają swoje własności i metody przed użytkownikiem. Przykładem może być definicja punktu w przestrzeni lub dobrze wszystkim znanej liczby zespolonej.

Definiowanie własnego agregatu (UDA). W części systemów wykonanie agregacji typu SUM, AVG, MIN, MAX itd. jest niewystarczające. Wraz z możliwością integracji CLR z SQL Server użytkownik ma możliwość definiowania własnego agregatu, który może być użyty w klauzuli GROUP BY. Przykładem takiego agregatu może być obliczanie transformaty Fouriera lub średniej odległości punktu od określonego miejsca.

Mechanizm kompilacji i uruchomienia

Każdy program napisany w .NET Framework nie jest bezpośrednio kompilowany do kodu natywnego, ale do formy pośredniej zwanej MSIL (Microsoft Intermediate Language). Dopiero ona w momencie wykonywania programu jest przekształcana na kod „zrozumiały” dla systemu operacyjnego (kod natywny). Odbywa się to bezpośrednio za pośrednictwem JIT (just-in-time) compiler, jednej z usług oferowanych przez CLR (Common Language Runtime, czyli wspólne środowisko uruchomieniowe dla platformy .NET).

W środowisku, jakim jest SQL Server, gdy kod SQL jest kompilowany i pojawia się odwołanie do zarządzanych komponentów, generowany jest obiekt zastępczy, zwany namiastką (ang. stub). Zawiera ona kod pośredniczący pozwalający przekazać parametry ze „zwykłego” przebiegu kodu w SQL Server do CLR, wywołać funkcje i zwrócić wynik operacji. Ten kod jest odpowiednio

optymalizowany i gwarantuje poprawność wymuszoną przez SQL Server. Natomiast pozostała część namiastki jest kompilowana do kodu natywnego i zoptymalizowana do architektury sprzętowej, gdzie działa SQL Server. W wyniku tego otrzymujemy wskaźnik do funkcji, która może być wywoływana z kodu natywnego bazy danych. Innymi słowy za pośrednictwem namiastki baza danych przekazuje parametry CLR, ten wykonuje całą pracę, a następnie po prostu zwraca wynik.

Samo wykonanie zadań powierzonych CLR odbywa się tak, że SQL Server, a w zasadzie jego procesy, stanowią środowisko uruchomieniowe dla komponentów skompilowanych do MSIL. Współpraca przebiega w taki sposób, że SQL Server zachowuje się jak system operacyjny dla CLR, który jest wykonywany w jego „wnętrzu". CLR wywołuje niskopoziomowe funkcje udostępnione przez SQL Server, które obsługują zarządzanie pamięcią, synchronizację itd. Daje to bazie danych kontrolę nad sposobem wykonania kodu.

Przykładowo, gdy CLR wykonuje swoją wewnętrzną operację, nadzorca szeregowalności bazy danych (ang. scheduler) jest o tym informowany i może swobodnie wykonywać zadania niepowiązane z kodem zarządzanym. Oprócz tego może także wykrywać zakleszczenia jakie nastąpiły z udziałem CLR i usuwać je tradycyjnymi metodami.

Kolejną zaletą tego rozwiązania jest to, że CLR nigdy nie będzie konkurował o pamięć z SQL Server. Baza może w razie potrzeby odrzucać prośby kodu zarządzanego o przydział zasobów lub ograniczyć dostępną dla niego pamięć. Dzięki temu nie ma ryzyka przekroczenia zasobów przeznaczonych dla SQL Server'a czy problemów związanych z synchronizacją. W mechanizmie tym uniknięto także niebezpieczeństwa związanego z utratą integralności danych przechowywanych w bazie.

Warto także wspomnieć, że każdy program napisany w .NET korzysta z CAS (Code Access Security), które definiuje, jakie operacje zarządzany kod może wykonać. Baza danych ma w tym momencie możliwość nałożenia ograniczeń za sprawą host-level security.

Zatem, ładując daną kompilacje do bazy danych mamy możliwość określenia poziomu bezpieczeństwa, jaki będzie jej przysługiwał.

Schemat postępowania w przypadku tworzenia obiektów CLR.

1. Przygotowanie odpowiedniej klasy .NET, które publicznie udostępniają odpowiednie elementy.
2. Kompilacja klasy .NET do zarządzanych plików DLL z manifestem zawierających pakiet.
3. Dołączenie skompilowanej klasy do SQL Server za pomocą wyrażenia CREATE ASSEMBLY.
4. Utworzenie odpowiedniej funkcjonalności dla dołączonego kodu przy użyciu odpowiednich poleceń języka SQL: CREATE FUNCTION, CREATE PROCEDURE, CREATE TYPE, CREATE TRIGGER lub CREATE AGGREGATE.

Konfiguracja serwera SQL Server do realizacji funkcjonalności CLR

```
sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO
sp_configure 'clr enabled', 1;
GO
RECONFIGURE;
GO
```

Ćwiczenie A. Pierwsza funkcja UDF w CLR

1. Ćwiczenie zostanie wykonane z wykorzystaniem poleceń wykonanych z wiersza poleceń. W przypadku kompilacji klasy wykorzystamy okno komend dostępne z menu aplikacji Visual Studio 2008, Visual Studio Tools: „Visual Studio 2008 Command Prompt" (okno zawiera poprawnie ustawione zmienne środowiskowe, m.in. dostęp do kompilatora). Polecenia w języku T-SQL wykonamy z wykorzystaniem narzędzia SQL Server Management Studio (SSMS).
2. Na początek tworzymy klasę publiczną w języku .NET z przynajmniej jedną statyczną metodą (tzw. entry point) w dowolnym edytorze tekstowym.
3. W kolejnym kroku wykonamy kompilację klasy do pliku DLL w technologii .NET.
4. Plik zostanie dołączony do serwera SQL poprzez załadowanie pliku DLL do bazy przy pomocy polecenia CREATE ASSEMBLY T-SQL.
5. Na koniec przydzielimy określoną funkcjonalność (funkcja, procedura składowana, wyzwalacz) w bazie danych dla przygotowanej klasy w technologii CLR. Realizacja tego zadania realizowane jest poprzez odpowiednie polecenie DDL języku SQL, które połączy utworzony obiekt SQL referencją do „*assembly*".

6. Przykładowe utworzenie funkcji z użyciem CLR.

   a. Na potrzeby realizacji skryptów w ramach laboratorium tworzymy bazę danych: „testCLR".

   b. Tworzymy plik zawierający klasę i kompilujemy.

```
// plik kompilujemy w Visual Studio 2008 z linii poleceń
// csc /t:library GetMVer.cs
// (nazwa pliku – GetMVer.cs)

public partial class LabCLR
{
    public static int GetCLRFrameworkMajorVersion()
    {
        return System.Environment.Version.Major;
    }
};
```

   c. Tworzymy „*assembly*" w bazie „testCLR".

```
USE testCLR
GO
---- tworzymy assembly z dll
CREATE ASSEMBLY [Lab7.GetMVerCLR] FROM
                        ' --katalog z biblioteka -- \getmver.dll'
```

d. Tworzymy funkcję na podstawie zarejestrowanego „*assembly*”.

```
-- tworzymy funkcje odnoszaca sie do funkcji GetCLRFrameworkMajorVersion
-- w klasie LabCLR w assembly Lab7.GetMVerCLR

CREATE FUNCTION [dbo].[fnGetMVerCLR]()
RETURNS [int]
AS
EXTERNAL NAME
[Lab7.GetMVerCLR].[LabCLR].[GetCLRFrameworkMajorVersion]
```

e. Uruchomienie utworzonej funkcji.

```
-- sposob uzycia
SELECT SqlCLRVersion=[dbo].[fnGetMVerCLR]()
```

Ćwiczenie B. Porównanie funkcji w języku T-SQL i technologii CLR

1. Tworzymy funkcję w języku T-SQL w bazie „testCLR” .

```
--- Skrypt Lab07.01
CREATE FUNCTION dbo.Factorial1 (@Number FLOAT)
RETURNS FLOAT
AS
BEGIN
DECLARE @returnValue FLOAT
IF @Number <= 1
        SELECT @returnValue = 1
ELSE
        SELECT @returnValue = @Number * dbo.Factorial1(@Number - 1)
RETURN @returnValue
END
GO
```

Poprawność funkcji sprawdzamy wykonując poniższe polecenie:

```
SELECT dbo.Factorial1(4)
```

2. Tworzymy funkcję w technologii CLR realizującą funkcjonalność funkcji opracowanej w języku T-SQL w skrypcie Lab07.01. Funkcja zostanie napisana w języku Visual Basic. Skorzystamy graficznego interfejsu Visual Studio i narzędzi do umieszczenia gotowej aplikacji na serwerze bazodanowym w określonej bazie danych.

Kolejność etapów tworzenia odpowiedniego obiektu w bazie danych:
- otwieramy aplikację Visual Studio
- wybieramy kolejno: File | New | Project

- wybieramy kolejno:
  język programowania : Visual Basic, Database
  w szablonach: SQL Server Project
  nazwa projektu: „CLR01”
- w oknie „Add Database Reference” wybieramy:
  serwer - SQLSERVER i bazę danych - „testCLR”
  Uwaga, jeżeli nie będzie uruchomiony serwis „SQL Server Browser” w oknie wyboru serwera nie wyświetli się żaden serwer.
- w oknie „Solution Explorer”:
  podświetlamy CLR01, prawy przycisk myszy
  i wybieramy Add | New Item
  a następnie User-Defined Function i wpisujemy nazwę „Factorial2”
- wpisujemy poniższy tekst w obszarze przeznaczonym na naszą funkcję

```
// Skrypt Lab07.02
<SqlFunction()> _
Public Shared Function Factorial2(ByVal Number As Integer) _
As Double
If Number <= 1 Then
        Return 1
Else
        Return Number * Factorial2(Number - 1)
End If
End Function
```

- w oknie „Solution Explorer” podświetlamy CLR01 i wybieramy „Properties”:
  zmieniamy nazwę w „Assembly name”:
  z „SqlClassLibrary” na „SqlClassLibrary1”.
- następnie w oknie „Solution Explorer”, podświetlamy CLR01:
  wybieramy Build i jeżeli nie ma błędów kompilacji Deploy
- sprawdzamy w SQL Server Management Studio w bazie danych „testCLR”
  w zakładce „Programmability” – Function – Scalar-Valued Functions, czy pojawiła się funkcja dbo.Factorial2.

Poprawność działania funkcji sprawdzamy tak jak w funkcji Factorial1(). Następnie należy wykonać sprawdzenie poprawności działania obu funkcji dla wartości argumentu 33. Znaleźć przyczynę otrzymania różnych wyników otrzymanych dla podanego argumentu funkcji.

3. Kolejna metoda wyznaczenia funkcji Factorial3() w ramach języka T-SQL. Jaka jest maksymalna liczba wywołań funkcji, jak można tę wartość ustawić ?

```
--- Skrypt Lab07.03
CREATE FUNCTION dbo.Factorial3 (@n int = 1) RETURNS float
WITH RETURNS NULL ON NULL INPUT
AS
BEGIN
DECLARE @wynik float;
```

```
SET @wynik = NULL;
IF @n > 0
BEGIN
  SET @wynik = 1.0;
  WITH cte (val)
  AS (
    SELECT 1
    UNION ALL
    SELECT val + 1 FROM cte WHERE val < @n
  )
  SELECT @wynik = @wynik * val  FROM cte;
END;
RETURN @wynik;
END;
```

Ćwiczenie C. Funkcje UDF w języku C#.

1. Tworzymy funkcję w języku C#. Kolejność wykonanych operacji w interfejsie Visual Studio przedstawiono w poniższych punktach.
   a) Otwieramy program Visual Studio.
   b) Tworzymy nowy projekt w języku C#, Database, nadajemy nazwę CLR02.
   c) W oknie „Add Database Reference" wybieramy serwer SQLSERVER i bazę danych „testCLR".
   d) W oknie „Solution Explorer" podświetlamy CLR02 i dodajemy z menu „User-defined Function" i nadajemy nazwę plikowi z kodem funkcji: Fun1.cs.
   e) Zmieniamy w własnościach projektu CLR02 nazwę "Assembly name" dodając na końcu nazwy 2 – „SqlClassLibrary2".
   f) Po wprowadzeniu tekstu skryptu wykorzystując okno „Solution Explorer" wybieramy akcje – Build i Deploy (jeżeli nie ma błędów kompilacji)
   g) Sprawdzamy poprawność działania skryptu w SQL Management Studio.
   h) Sprawdzamy skrypty w języku T-SQL tworzące obiekty „assemblies" i „functions" z wykorzystaniem pozycji „Script As" w menu kontekstowym.

Poniżej skrypt do wykonania ćwiczenia:

```
// Skrypt Lab07.04

public partial class UserDefinedFunctions
{
   [Microsoft.SqlServer.Server.SqlFunction]
   public static int GetVerCLR()
   {
      // Put your code here
      return System.Environment.Version.Major;
   }
};
```

Poprawność opracowanej funkcji sprawdzamy poleceniem SELECT w SSMS.

```
SELECT dbo.GetVerCLR();
```

Kolejną funkcją zrealizowaną w ramach technologii CLR będzie funkcja sprawdzająca poprawność kodu pocztowego. Do realizacji tego zadania wykorzystamy wyrażenia regularne. Ćwiczenie zrealizujemy z wykorzystaniem interfejsu Visual Studio w wcześniej otwartym projekcie CLR02.

1. W projekcie CLR02 w oknie „Solution Explorer” w menu kontekstowym wybieramy - Add a następnie - User-Defined Function.
2. Wpisujemy nazwę pliku z funkcją „Fun2.cs”
3. Wybieramy wzorzec „Template user-Defined Function”
4. Edytujemy kod funkcji wstawiając przedstawiony poniżej kod.

```
// Skrypt Lab07.05
public partial class UserDefinedFunctions
{
   [Microsoft.SqlServer.Server.SqlFunction]
   public static bool IsValidZipCode(string ZipCode)
   {
      // Put your code here
      return System.Text.RegularExpressions.Regex.IsMatch(
         ZipCode, @"^\s*(\d{2}-\d{3})\s*$");
   }
};
```

Poprawność opracowanej funkcji sprawdzamy poleceniem SELECT w SSMS.

```
SELECT dbo.IsValidZipCode('11-111');
```

Ćwiczenie D. Przestrzenie nazw w ramach CLR

Poza standardowymi przestrzeniami nazw i klasami .NET, przy integracji CLR dołączane specyficzne dla SQL Server przestrzenie nazw i klasy umożliwiające komunikację tworzonego kodu z SQL Server. Poniżej przedstawiono najczęściej wykorzystywane przestrzenie nazw.

- ✓ **System** - zawiera podstawowe typy danych .NET oraz klasę bazową Object, z której dziedziczą wszystkie klasy .NET.
- ✓ **System.Data** - zawiera klasę DataSet i inne klasy do zarządzania danymi w ADO.NET.
- ✓ **System.Data.SqlClient** - dostarcza specyficzne dla SQL Server mechanizmy pobierania danych poprzez ADO.NET.
- ✓ **System.Data.SqlTypes** - zawiera typy danych SQL Server. Udostępnia ona klasy, które w sposób precyzyjny umożliwiają konwersję typów pomiędzy bazą danych SQL a aplikacją uruchomioną w środowisku .NET. Typy danych z tej przestrzeni posiadają właściwość IsNull. Nie można jednakże przypisywać wartości NULL zmiennym z przestrzeni SqlTypes.
  Opis przestrzeni nazw *System.Data.SqlTypes* i porównanie tych typów z typami danych

dostępnych na serwerze SQL i typami danych dostępnych w środowisku .NET dostępne jest pod poniższym adresem:
https://msdn.microsoft.com/pl-pl/library/system.data.sqltypes%28v=vs.110%29.aspx

- ✓ **Microsoft.SqlServer.Server** - zawiera klasy SqlContext i SqlPipe, umożliwiające komunikowanie kodu zarządzanego z SQL Server'em.
- ✓ **System.Data. SqlDbTypes** – umożliwia przesyłanie danych do serwera SQL z wykorzystaniem typów SqlParameter.

W ramach procedur CLR korzystamy z obiektów SqlContext i SqlPipe (dostępne w przestrzeni nazw *Microsoft.SqlServer.Server***)**. Obiekt SqlContext reprezentuje połączenie zwrotne do sesji, która utworzyła instancję CLR. Obiekt jest dostępny w zasięgu procedur CLR i nie należy tworzyć jego instancji (błąd) lecz od razu używać. Kolejny obiekt SqlPipe z właściwą metodą umożliwia przesłanie wyników do sesji wywołującej. Obiekt SqlPipe tworzymy wykorzystując polecenie SqlContext.Pipe.

W ramach obiektu *SqlPipe* udostępnione są metody *Send()* i *ExecuteAndSend()*. Metodę *Send()* przeciążać. Poniżej przedstawiono funkcjonalność metod.

- ✓ ***Send(String komunikat)*** – metoda wysyła ciąg znaków, który zostanie zinterpretowany jako komunikat. Komunikat odpowiada obiektowi InfoMessage ADO.NET albo informacjom wyświetlanym w panelu komunikatów SSMS. Działanie jest podobne do instrukcji PRINT w T-SQL.

```
SqlContext.Pipe.Send("SQL CLR - komunikat");
```

- ✓ ***Send(SqlRecord rekord)*** – wysyła do metody wywołującej pojedynczy rekord. Metodę te wykorzystuje się do przesyłania pojedynczych rekordów z tabeli w połączeniu z metodami *SendResultStart()* i *SendResultEnd()*. Prawidłowe wykorzystanie tej metody nie należy do prostych implementacji.

- ✓ ***Send(SqlDataReader reader)*** – ta wersja metody *Send* wysyła do metody wywołującej od razu cała tabelę. Procedura wykorzystuje obiekt SqlDataReader, która zawiera wszystkie dane wymagane do przesłania.

```
command.CommandText ="SELECT * FROM table";
SqlDataReader reader = command.ExecuteReader();
SqlContext.Pipe.Send(reader);
```

- ✓ ***ExecuteAndSend(SqlCommand command)*** – metoda ta otrzymuje informacje o poleceniu, które ma zostać wykonane na serwerze SQL i po wykonaniu wyniki zostaną przesłane bez konieczności pośrednictwa procedury CLR jak w przypadku ostatniej wersji dla metody Send().

Realizacja ćwiczenia z wykorzystaniem interfejsu Visual Studio w projekcie CLR02.

1. W projekcie CLR02 w oknie „Solution Explorer” w menu kontekstowym wybieramy - Add a następnie - Stored Procedure.
2. Wpisujemy nazwę pliku z procedurą „SProc01.cs”
3. Wybieramy wzorzec „Template Stored Procedure”
4. Edytujemy kod procedury wstawiając przedstawiony poniżej kod.

```
// Skrypt Lab07.06
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
public partial class StoredProcedures
{
   [Microsoft.SqlServer.Server.SqlProcedure]
   public static void SqlTypesNull(SqlInt32 oInt)
   {
      SqlContext.Pipe.Send(Convert.ToString((oInt.Value == 1)));
      // "odkomentowac" ponizszy kod i "zakomentować" powyzszy
      //if (oInt.IsNull == false)
      //{
      //   SqlContext.Pipe.Send(Convert.ToString((oInt.Value == 1)));
      //}
      //else
      //{
      //   SqlContext.Pipe.Send("Parametr ma wartosc NULL");
      //}
   }
};
```

Funkcjonalność procedury sprawdzamy poniższym poleceniem w SSMS.

```
EXECUTE  SqlTypesNull  NULL;
```

Ćwiczenie E. Procedura zwracająca czas z systemu SQL

W ramach kolejnego ćwiczenia połączymy się z procedury CLR z lokalną bazą danych. Wykonamy zapytanie SQL oraz zwrócimy wynik zapytania do procedury.

Realizacja tego projektu wymaga połączenia z serwerem SQL. Podobnie jak w aplikacjach ADO.NET wykorzystujemy tutaj obiekt *SqlConnection* z odpowiednim ciągiem połączenia przesłanym we właściwościach. Dla połączenia naszego obiektu z bazą danych w którym została utworzona procedura CLR wyślemy następujący ciąg połączenia ***"context connection=true"***. Jest to informacja dla serwera SQL, że dostęp do danych następuje w ramach tego samego połączenia, które utworzyło metodę CLR.

Zadanie można zrealizować wysyłając zapytanie do bazy danych SQL Server poniżym skryptem.

```
SqlConnection cn = new SqlConnection(connectionString);
```

```
SqlCommand cm = new SqlCommand(commandString, cn);
cn.Open();
cn.ExecuteAndSend(SqlCommand cm);
cn.Close();
```

Realizacja ćwiczenia z wykorzystaniem interfejsu graficznego Visual Studio w otwartym projekcie CLR02.

1. W projekcie CLR02 w oknie „Solution Explorer” w menu kontekstowym wybieramy - Add a następnie - Stored Procedure.
2. Wpisujemy nazwę pliku z procedurą „SProc02.cs”
3. Wybieramy wzorzec „Template Stored Procedure”
4. Edytujemy kod procedury wstawiając przedstawiony poniżej kod.

```
// Skrypt Lab07.07
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

public partial class StoredProcedures
{
   [Microsoft.SqlServer.Server.SqlProcedure]
   public static void GetCurrentTime()
   {
      SqlConnection conn = new SqlConnection("context connection=true");
      conn.Open();
      SqlCommand cmd = new SqlCommand("SELECT CURRENT_TIMESTAMP AS czas",
conn);
      SqlDataReader rdr = cmd.ExecuteReader();
      SqlContext.Pipe.Send(rdr);
   }
};
```

Sprawdzamy w SQL Management Studio poprawność działania procedury.

```
EXECUTE GetCurrentTime ;
```

Ćwiczenie F. Przestrzeń nazw System.Data.SqlDbType

W przypadku tworzenia zapytań do bazy danych wykorzystujących SqlParametr należy korzystać z enumeracji dostępnej w przestrzeni nazw **System.Data.SqlDbTypes**.

Lista typów zdefiniowanych w ramach tej przestrzeni dostępna jest pod poniższym adresem: https://msdn.microsoft.com/pl-pl/library/system.data.sqldbtype%28v=vs.110%29.aspx

Poniżej skrypt realizyujący zapytanie do bazy danych wykorzystające parametr SqlParametr, ćwiczenie realizujemy z wykorzystaniem interfejsu graficznego Visual Studio w wcześniej otwartym projekcie CLR02.

1. W projekcie CLR02 w oknie „Solution Explorer” w menu kontekstowym wybieramy - Add a następnie - Stored Procedure.
2. Wpisujemy nazwę pliku z procedurą „SProc03.cs”
3. Wybieramy wzorzec „Template Stored Procedure”
4. Edytujemy kod procedury wstawiając przedstawiony poniżej kod.

```
// Skrypt Lab04.06
// procedura ma parametr typu SqlInt32
// EXECUTE SqlTypesVsSqlDbTypes 2 w SSMS
[Microsoft.SqlServer.Server.SqlProcedure]
public static void SqlTypesVsSqlDbTypes(SqlInt32 iID)
{
    using (SqlConnection oConn =
    new SqlConnection("context connection=true"))
    {
        SqlCommand oCmd = new SqlCommand("SELECT * FROM " +
            "AdventureWorks.Production.ProductCategory " +
            "WHERE ProductCategoryID = @ID", oConn);
        oCmd.Parameters.Add("@ID", SqlDbType.Int).Value = iID;
        oConn.Open();
        SqlContext.Pipe.ExecuteAndSend(oCmd);
        oConn.Close();
    }
}
```

Poprawność opracowanej procedury sprawdzamy w SSMS wykonując poniżesz polecenie.

```
EXECUTE  SqlTypesVsSqlDbTypes 2
```

Zadania

Zadanie Z1

Napisać prostą funkcję w technologii CLR zwracająca czas systemowy. Funkcję umieścić w bazie testCLR.

Zadanie Z2

Napisać funkcję sparametryzowaną w technologii CLR w bazie „testCLR", która wyświetli powitanie w rodzaju:

Witaj: <login>, dzisiaj jest: <data>, pracujesz na serwerze <system> w systemie <system>.

Pomocne funkcje i parametry:
DateTime.Now.ToString, SUSER_SNAME(), @@SERVERNAME, CURRENT_USER, @@VERSION, SYSTEM_USER.

Zadanie Z3

Napisać procedurę składowaną mającą za zadanie wyszukanie wszystkich rekordów w tabeli AdventureWorks.HumanResources.Employee, dla których podany ciąg znaków (parametr procedury) będzie zawarty w polu EmailAddress tabeli Person.Contact. Procedurę umieścić w bazie testCLR.

Uwaga:

W ramach rozwiązania należy przekazać odpowiednie kody procedur w języku C# oraz skrypty T-SQL tworzące odpowiednie obiekty w bazie danych „testCLR".