User Defined Data Types in SQL Server 2005

What do you do, when you want a data type that should represent more than one interrelated values? Would you do this with two separate data types? Not a good idea! Is it possible that you can do this with just one single data type? Yes, you can. We have been provided a “proper” way to sort this out: The User-Defined Type in SQL Server 2005. Well, you had this in SQL Server 2000, didn’t you? Yes, but in this case it is really enhanced, or in other words, it is greatly empowered.

The integration of the .NET CLR (Common Language Runtime) into SQL Server 2005 is the man of many parts, that allows us to create our own data type by using our favorite manage code, like C#.NET. This article speaks about how to create a user-defined type and how can it be useful in our day-to-day operations. This article does not contain much detail about user-defined types, but the manipulation of them. The reason that made me write this kind of article is, being a developer, I always love to have hands-on-experience and the concept of this can be found in many forms.

What is the UDT that we are going to create?

As I really wanted to put this in a more practical manner, I went through some of my applications that were done using SQL Server 2000 in order to find a proper kind of value to make a UDT. Finally, I found one in one of the Sales-Automation applications of mine. One of the processes that is involved in this operation is to have a count on both physical and system stock on a daily basis of items. The physical stock and system stock are stored separately in two different columns. Why do we not have a single column for this purpose? This thought came up as soon as I saw my own design. So, this little scenario is going to be used for our first UDT, though a little simple.

Naming the new UDT as “ItemStock” is fine I think. The “ItemStock” will have two parts: Physical Stock and System Stock. Having a third part, something like Unit-Of-Measure will give our UDT more flexibility, but I will not go ahead with it, just to keep things simple. Remember the KISS principle?

What is the first step?

The first step of the procedure is to have an assembly that contains the implementation of the UDT. Since my favorite CLR-supported language is C#.NET, we’ll create the assembly with C#.NET. All you have to do is, open Visual Studio 2005 and create a new project of the type “Database”. Once created, add a new item which should be a “User-Defined Type”. This leads to the creation of a skeleton of the type we need. Here is the cleaned-up code:

```
[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedType(Format.Native)]
public struct ItemStock : INullable
{
   public override string ToString()
   {
      return "";
   }
```

```
    public bool IsNull
    {
        get
        {
            return m_Null;
        }
    }

    public static ItemStock  Null
    {
        get
        {
            ItemStock

             h = new ItemStock ();
            h.m_Null = true;
            return h;
        }
    }

    public static ItemStock Parse(SqlString   s)
    {
        if (s.IsNull)
            return Null;
        ItemStock
         u = new ItemStock ();
        return u;
    }
}
```

I'll give you simple description on some important parts of this code snippet.

- **[Serializable]**

This allows the UDT to be serialized.

- **[Microsoft.SqlServer.Server.SqlUserDefinedType(Format.Native)]**

This is a required attribute for UDTs. The required property "Format" determines the serialization format. With the version of Beta 2 April CTP, there are three elements under the Format enumerator: Native, Unknown and UserDefined. This attribute can be specified with some more named-parameters like IsByteOrdered, IsFixedLength, MaxByteSize, Name and ValidationMethodName.

- **Format.Native**

This is the native SQL Server binary serialization which is the fastest. This can only be applied to fixed-length, value-type data types. The named-parameter "MaxByteSize" cannot be specified with this type.

- **Format.UserDefined**

This is basically used for reference types. If this is specified, the serialization have to be implemented by ourselves. This can be done by implementing IBinarySerialize interface.

- **INullable**

Implementing INullable interface makes the UDT null aware. The public property “IsNull” of the INullable must be implemented too.

- **Static Null method**

This method must be implemented that returns the type of UDT itself.

- **Static Parse method**

This static method makes UDT convert-possible from a string value.

- **ToString()**

This Overrides the UDT’s virtual ToString() method. The convention of the UDT back to the string is done through this.

Well, I think that I have covered some basic, important areas of the UDT. Now, have a look on our UDT: The ItemStock.

```
[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedType(Format.Native)]
public struct ItemStock : INullable
{
        //these three variables hold the value of physical stock, system stock,
        //and the indicator whether the value of UDT is null respectively.
        private int _physicalStock;
        private int _systemStock;
        private bool _isNull;

        //The INullable readonly property.
        public bool IsNull { get { return _isNull; } }

        //This method returns null-valued UDT.
        public static ItemStock Null
        {
                get
                {
                        ItemStock itemStock = new ItemStock();
                        itemStock._isNull = true;
                        return itemStock;
                }
        }

        //Coversion from string to UDT.
        //If the value is null, return null-valued UDT.
        //If not, validate and create a new ItemStock UDT.
        public static ItemStock Parse(SqlString value)
        {
                if (value.IsNull)
                        return ItemStock.Null;

                ItemStock itemStock = new ItemStock();
                try
                {
                        string[] values = value.ToString().Split('|');
```

```
                        itemStock._physicalStock = Convert.ToInt32(values[0]);
                        itemStock._systemStock = Convert.ToInt32(values[1]);
                        itemStock._isNull = false;

                        return itemStock;
                }
                catch (Exception exception)
                {
                        throw exception;
                }
        }

        //String representation of the UDT.
        public override string ToString()
        {
                if (this.IsNull)
                        return "NULL";
                else
                        return this._physicalStock + " | " + this._systemStock;
        }

        //Returns the physical stock. This is a read-only property
        public int PhysicalStock { get { return this._physicalStock; } }

        //Returns the system stock. This is a read-only property
        public int SystemStock { get { return this._systemStock; } }

        //Returns the difference between physical and system stock. This is a read-o
        public int Difference { get { return _physicalStock - _physicalStock;}}
}
```

Well, the code is very simple. You can clearly see, we have implemented all what we discussed so far. In addition to that, three new properties have been implemented. You will see the usability of it soon.

The Second Step

Now it is time to register the assembly we created in the SQL Server 2005. Open the MS SQL Server Management Studio and Click on “New Query”. Set the database you want to add the UDT and place the code below on it.

```
CREATE ASSEMBLY SQLServerUDTs
FROM '{FilePath}\SQLServerUDTs.dll'
```

Here is the Last step

Once you run the second step, the last step of the procedure is, create a data type base on the UDT we created. See the code below.

```
CREATE TYPE ItemStock EXTERNAL NAME SQLServerUDTs.ItemStock
```

The above code creates a UDT named “ItemStock” that points to the assembly registered. The “SQLServerUDTs.ItemStock” represents the namespace and the UDT name in the assembly.

Time to use it

We have done! To see the usability of the newly created UDT, let's create a simple table that contains the UDT and insert some records.

```
CREATE TABLE Items
(Id int IDENTITY(1,1) PRIMARY KEY,
[Name] varchar(100) NOT NULL,
InStock ItemStock NULL)

INSERT INTO Items ([Name], InStock) VALUES ('Dining Table 1', '0|0')
INSERT INTO Items ([Name], InStock) VALUES ('Dining Table 2', NULL)
INSERT INTO Items ([Name], InStock) VALUES ('Dining Table 3', NULL)
INSERT INTO Items ([Name], InStock) VALUES ('Dining Table 4', NULL)
--this will cause to an exception
INSERT INTO Items ([Name], InStock) VALUES ('Dining Table 4', 'p|s')
```

Ok, now table is ready with data. Let's see how we it help us in day-to-day operations.

```
-- Select all columns from the tables.
-- See the way we have to access UDT properties
SELECT Id, [Name], InStock.ToString(),
        InStock.PhysicalStock, InStock.SystemStock
FROM Items

-- Updating records
UPDATE Items
        SET InStock = '1860|1860'
WHERE Id = 3
UPDATE Items
        SET InStock = '138|156'
WHERE Id = 4

-- applying for conditions
SELECT Id, InStock.Difference
FROM Items
WHERE InStock.PhysicalStock < InStock.SystemStock
```

Well, you can see now how the new stuff that comes with SQL Server 2005, make our life easier. Especially the MS .Net Framework integration, which is the most intriguing part for me. You can see some of the experiments I have done with this and more are being done.

As I have stated above, this article started you off on some basic UDT stuff. Well, in my next article, I will be covering more on this and I will highly appreciate if you can give some comments on this. I know definitely, it will help me out with next part on UDT.