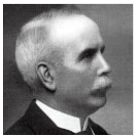
Consuming JSON Strings in SQL Server

red-gate.com/simple-talk/sql/t-sql-programming/consuming-json-strings-in-sql-server

November 15, 2010



Phil Factor
15 November 2010

142

It has always seemed strange to Phil that SQL Server has such complete support for XML, yet is completely devoid of any support for JSON. In the end, he was forced, by a website project, into doing something about it. The result is this article, an iconoclastic romp around the representation of hierarchical structures, and some code to get you started.

Last updated 1st July 2019

Articles by Phil Factor about JSON and SQL Server:

"The best thing about XML is what it shares with JSON, being human readable. That turns out to be important, not because people should be reading it, because we shouldn't, but because it avoids interoperability problems caused by fussy binary encoding issues.

Beyond that, there is not much to like. It is not very good as a data format. And it is not very good as a document format. If it were a good document format, then wikis would use it."

Doug Crockford March 2010

This article describes a TSQL JSON parser and its evil twin, a JSON outputter, and provides the source. It is also designed to illustrate a number of string manipulation techniques in TSQL. With it you can do things like this to extract the data from a JSON document:

```
Select * from parseJSON('{
"Person":
 {
   "firstName": "John",
   "lastName": "Smith",
   "age": 25,
   "Address":
   {
     "streetAddress":"21 2nd Street",
     "city":"New York",
     "state":"NY",
     "postalCode":"10021"
   },
   "PhoneNumbers":
   {
     "home":"212 555-1234",
     "fax":"646 555-4567"
   }
 }
}
')
```

And get:

	element_id	parent_ID	Object_ID	NAME	StringValue	ValueType
1	1	1	NULL	streetAddress	21 2nd Street	string
2	2	1	NULL	city	New York	string
3	3	1	NULL	state	NY	string
4	4	1	NULL	postalCode	10021	string
5	5	2	NULL	home	212 555-1234	string
6	6	2	NULL	fax	646 555-4567	string
7	7	3	NULL	firstName	John	string
8	8	3	NULL	lastName	Smith	string
9	9	3	NULL	age	25	int
10	10	3	1	Address	1	object
11	11	3	2	PhoneNumbers	2	object
12	12	4	3	Person	3	object
13	13	NULL	4	-		object

...or you can do the round trip:

```
1   DECLARE @MyHierarchy Hierarchy  INSERT INTO @myHierarchy
2   select * from parseJSON('{"menu": {
3     "id": "file",
4     "value": "File",
5     "popup": {
6       "menuitem": [
7         {"value": "New", "onclick": "CreateNewDoc()"},
8         {"value": "Open", "onclick": "OpenDoc()"},
9         {"value": "Close", "onclick": "CloseDoc()"}
10      ]
11    }
12  }}')
13  SELECT dbo.ToJSON(@MyHierarchy)
```

To get:

```
{  "menu" :  {
  "id" : "file",
  "value" : "File",
  "popup" :  {
    "menuitem" :  [
      {
      "value" : "New",
      "onclick" : "CreateNewDoc()"
      },
      {
      "value" : "Open",
      "onclick" : "OpenDoc()"
      },
      {
      "value" : "Close",
      "onclick" : "CloseDoc()"
      }
    ]
  }
 }
}
```

Background

TSQL isn't really designed for doing complex string parsing, particularly where strings represent nested data structures such as XML, JSON, YAML, or XHTML.

You can do it but it is not a pretty sight; but why would you ever want to do it anyway? Surely, if anything was meant for the 'application layer' in C# or VB.net, then this is it. 'Oh yes', will chime in the application thought police, 'this is far better done in the application or with a CLR.' Not necessarily.

Sometimes, you just need to do something inappropriate in TSQL. (note: You can now do this rather more easily using SQL Server 2016's built-in JSON support. See 'Consuming hierarchical JSON documents in SQL Server using OpenJSON' which I wrote more recently)

There are a whole lot of reasons why this might happen to you. It could be that your DBA doesn't allow a CLR, for example, or you lack the necessary skills with procedural code. Sometimes, there isn't any application, or you want to run code unobtrusively across databases or servers.

I needed to interpret or 'shred' JSON data. JSON is one of the most popular lightweight markup languages, and is probably the best choice for transfer of object data from a web page. It is, in fact, executable JavaScript that is very quick to code in the browser in order to dump the contents of a JavaScript object, and is lightning-fast to populate the browser object from the database since you are passing it executable code (you need to parse it first for security reasons – passing executable code around is potentially very risky). AJAX can use

JSON rather than XML so you have an opportunity to have a much simpler route for data between database and browser, with less opportunity for error.

The conventional way of dealing with data like this is to let a separate business layer parse a JSON 'document' into some tree structure and then update the database by making a series of calls to it. This is fine, but can get more complicated if you need to ensure that the updates to the database are wrapped into one transaction so that if anything goes wrong, then the whole operation can be rolled back. This is why a CLR or TSQL approach has advantages.

"Sometimes, you just
need to do something
inappropriate in TSQL..."

I wrote the parser as a prototype because it was the quickest way to determine what was involved in the process, so I could then re-write something as a CLR in a .NET language. It takes a JSON string and produces a result in the form of an adjacency list representation of that hierarchy. In the end, the code did what I wanted with adequate performance (It reads a json file of 540 name\value pairs and creates the SQL hierarchy table in 4 seconds) so I didn't bother with the added complexity of maintaining a CLR routine. In order to test more thoroughly what I'd done, I wrote a JSON generator that used the same Adjacency list, so you can now import and export data via JSON!

These markup languages such as JSON and XML all represent object data as hierarchies. Although it looks very different to the entity-relational model, it isn't. It is rather more a different perspective on the same model. The first trick is to represent it as a Adjacency list hierarchy in a table, and then use the contents of this table to update the database. This Adjacency list is really the Database equivalent of any of the nested data structures that are used for the interchange of serialized information with the application, and can be used to create XML, OSX Property lists, Python nested structures or YAML as easily as JSON.

Adjacency list tables have the same structure whatever the data in them. This means that you can define a single Table-Valued Type and pass data structures around between stored procedures. However, they are best held at arms-length from the data, since they are not relational tables, but something more like the dreaded EAV (Entity-Attribute-Value) tables. Converting the data from its Hierarchical table form will be different for each application, but is easy with a CTE. You can, alternatively, convert the hierarchical table into XML and interrogate that with XQuery.

JSON format.

JSON is designed to be as lightweight as possible and so it has only two structures. The first, delimited by curly brackets, is a collection of name/value pairs, separated by commas. The name is followed by a colon. This structure is generally implemented in the application-level as an *object*, record, struct, dictionary, hash table, keyed list, or associative array. The other structure is an ordered list of values, separated by commas. This is usually manifested as an *array*, vector, list, or sequence.

"Using recursion in TSQL is
like Sumo Wrestlers doing Ballet.
It is possible but not pretty."

The first snag for TSQL is that the curly or square brackets are not 'escaped' within a string, so that there is no way of shredding a JSON 'document' simply. It is difficult to differentiate a bracket used as the delimiter of an array or structure, and one that is within a string. Also, interpreting a string into a SQL String isn't entirely straightforward since hex codes can be embedded anywhere to represent complex Unicode characters, and all the old C-style escaped characters are used. The second complication is that, unlike YAML, the datatypes of values can't be explicitly declared. You have to sniff them out from applying the rules from the JSON Specification.

Obviously, structures can be embedded in structures, so recursion is a natural way of making life easy. Using recursion in TSQL is like Sumo Wrestlers doing Ballet. It is possible but not pretty.

The implementation

Although the code for the JSON Parser/Shredder will run in SQL Server 2005, and even in SQL Server 2000 (with some modifications required), I couldn't resist using a TVP (Table Valued Parameter) to pass a hierarchical table to the function, **ToJSON,** that produces a JSON 'document'. Writing a SQL Server 2005 version should not be too hard.

First the function replaces all strings with tokens of the form **@Stringxx**, where **xx** is the foreign key of the table variable where the strings are held. This takes them, and their potentially difficult embedded brackets, out of the way. Names are always strings in JSON as well as string values.

Then, the routine iteratively finds the next structure that has no structure contained within it, (and is, by definition the leaf structure), and parses it, replacing it with an object token of the form '**@Objectxxx**', or '**@arrayxxx**', where **xxx** is the object id assigned to it. The values, or name/value pairs are retrieved from the string table and stored in the hierarchy table. Gradually, the JSON document is eaten until there is just a single root object left.

The JSON outputter is a great deal simpler, since one can be surer of the input, but essentially it does the reverse process, working from the root to the leaves. The only complication is working out the indent of the formatted output string.

In the implementation, you'll see a fairly heavy use of PATINDEX. This uses a poor man's RegEx, a starving man's RegEx. However, it is all we have, and can be pressed into service by chopping the string it is searching (if only it had an optional third parameter like CHARINDEX that specified the index of the start position of the search!). The STUFF function is also a godsend for this sort of string-manipulation work.

```
1  Alter FUNCTION dbo.parseJSON( @JSON NVARCHAR(MAX))
2  /**
3  Summary: >
4    The code for the JSON Parser/Shredder will run in SQL Server 2005,
5    and even in SQL Server 2000 (with some modifications required).
6
7    First the function replaces all strings with tokens of the form @Stringxx,
8    where xx is the foreign key of the table variable where the strings are held.
9    This takes them, and their potentially difficult embedded brackets, out of
10   the way. Names are  always strings in JSON as well as  string values.
11
12   Then, the routine iteratively finds the next structure that has no structure
13   Contained within it, (and is, by definition the leaf structure), and parses it,
14   replacing it with an object token of the form '@Objectxxx', or '@arrayxxx',
15   where xxx is the object id assigned to it. The values, or name/value pairs
16   are retrieved from the string table and stored in the hierarchy table. G
17   radually, the JSON document is eaten until there is just a single root
18   object left.
19 Author: PhilFactor
20 Date: 01/07/2010
```

```
21  Version:
22    Number: 4.6.2
23    Date: 01/07/2019
24    Why: case-insensitive version
25  Example: >
26    Select * from parseJSON('{    "Person":
27        {
28         "firstName": "John",
29         "lastName": "Smith",
30         "age": 25,
31         "Address":
32             {
33            "streetAddress":"21 2nd Street",
34            "city":"New York",
35            "state":"NY",
36            "postalCode":"10021"
37             },
38          "PhoneNumbers":
39             {
40              "home":"212 555-1234",
41            "fax":"646 555-4567"
42             }
43          }
44       }
45    ')
46  Returns: >
47    nothing
48  **/
49  RETURNS @hierarchy TABLE
50    (
51     Element_ID INT IDENTITY(1, 1) NOT NULL, /* internal surrogate primary key gives the order of
    parsing and the list order */
52
53     SequenceNo [int] NULL, /* the place in the sequence for the element */
54     Parent_ID INT null, /* if the element has a parent then it is in this column. The document is the
    ultimate parent, so you can get the structure from recursing from the document */
55     Object_ID INT null, /* each list or object has an object id. This ties all elements to a parent. Lists are
    treated as objects here */
56
```

```
57      Name NVARCHAR(2000) NULL, /* the Name of the object */
58      StringValue NVARCHAR(MAX) NOT NULL,/*the string representation of the value of the element.
    */
59
        ValueType VARCHAR(10) NOT null /* the declared type of the value represented as a string in
60  StringValue*/
61      )
62      /*
63
64      */
65  AS
66  BEGIN
67      DECLARE
68        @FirstObject INT, --the index of the first open bracket found in the JSON string
69        @OpenDelimiter INT,--the index of the next open bracket found in the JSON string
70        @NextOpenDelimiter INT,--the index of subsequent open bracket found in the JSON string
71        @NextCloseDelimiter INT,--the index of subsequent close bracket found in the JSON string
72        @Type NVARCHAR(10),--whether it denotes an object or an array
73        @NextCloseDelimiterChar CHAR(1),--either a '}' or a ']'
74        @Contents NVARCHAR(MAX), --the unparsed contents of the bracketed expression
75        @Start INT, --index of the start of the token that you are parsing
76        @end INT,--index of the end of the token that you are parsing
77        @param INT,--the parameter at the end of the next Object/Array token
78        @EndOfName INT,--the index of the start of the parameter at end of Object/Array token
79        @token NVARCHAR(200),--either a string or object
80        @value NVARCHAR(MAX), -- the value as a string
81        @SequenceNo int, -- the sequence number within a list
82        @Name NVARCHAR(200), --the Name as a string
83        @Parent_ID INT,--the next parent ID to allocate
84        @lenJSON INT,--the current length of the JSON String
85        @characters NCHAR(36),--used to convert hex to decimal
86        @result BIGINT,--the value of the hex symbol being parsed
87        @index SMALLINT,--used for parsing the hex value
88        @Escape INT --the index of the next escape character
89
90      DECLARE @Strings TABLE /* in this temporary table we keep all strings, even the Names of the
    elements, since they are 'escaped' in a different way, and may contain, unescaped, brackets
91  denoting objects or lists. These are replaced in the JSON string by tokens representing the string */
92
```

```
93      (
94       String_ID INT IDENTITY(1, 1),
95       StringValue NVARCHAR(MAX)
96      )
97     SELECT--initialise the characters to convert hex to ascii
98      @characters='0123456789abcdefghijklmnopqrstuvwxyz',
99      @SequenceNo=0, --set the sequence no. to something sensible.
100    /* firstly we process all strings. This is done because [{} and ] aren't escaped in strings, which
    complicates an iterative parse. */
101
       @Parent_ID=0;
102
    WHILE 1=1 --forever until there is nothing more to do
103
      BEGIN
104
        SELECT
105
          @start=PATINDEX('%[^a-zA-Z]["]%', @json collate SQL_Latin1_General_CP850_Bin);--next
106 delimited string
107       IF @start=0 BREAK --no more so drop through the WHILE loop
108       IF SUBSTRING(@json, @start+1, 1)=""
109         BEGIN --Delimited Name
110           SET @start=@Start+1;
111           SET @end=PATINDEX('%[^\]["]%', RIGHT(@json, LEN(@json+'|')-@start) collate
    SQL_Latin1_General_CP850_Bin);
112
          END
113
       IF @end=0 --either the end or no end delimiter to last string
114
         BEGIN-- check if ending with a double slash...
115
            SET @end=PATINDEX('%[\][\]["]%', RIGHT(@json, LEN(@json+'|')-@start) collate
116 SQL_Latin1_General_CP850_Bin);
117      IF @end=0 --we really have reached the end
118  BEGIN
119  BREAK --assume all tokens found
120  END
121  END
122       SELECT @token=SUBSTRING(@json, @start+1, @end-1)
123       --now put in the escaped control characters
124       SELECT @token=REPLACE(@token, FromString, ToString)
125       FROM
126         (SELECT           '\b', CHAR(08)
127          UNION ALL SELECT '\f', CHAR(12)
128
```

```
129         UNION ALL SELECT '\n', CHAR(10)
130         UNION ALL SELECT '\r', CHAR(13)
131         UNION ALL SELECT '\t', CHAR(09)
132 UNION ALL SELECT '\"', '"'
133         UNION ALL SELECT '\/', '/'
134        ) substitutions(FromString, ToString)
135 SELECT @token=Replace(@token, '\\', '\')
136     SELECT @result=0, @escape=1
137   --Begin to take out any hex escape codes
138     WHILE @escape>0
139       BEGIN
140         SELECT @index=0,
141         --find the next hex escape sequence
142         @escape=PATINDEX('%\x[0-9a-f][0-9a-f][0-9a-f][0-9a-f]%', @token collate
    SQL_Latin1_General_CP850_Bin)
143         IF @escape>0 --if there is one
144           BEGIN
145             WHILE @index<4 --there are always four digits to a \x sequence
146               BEGIN
147                 SELECT --determine its value
148                   @result=@result+POWER(16, @index)
149                   *(CHARINDEX(SUBSTRING(@token, @escape+2+3-@index, 1),
150                           @characters)-1), @index=@index+1 ;
151
152
153               END
154               -- and replace the hex sequence by its unicode value
155             SELECT @token=STUFF(@token, @escape, 6, NCHAR(@result))
156           END
157       END
158     --now store the string away
159     INSERT INTO @Strings (StringValue) SELECT @token
160     -- and replace the string with a token
161     SELECT @JSON=STUFF(@json, @start, @end+1,
162                 '@string'+CONVERT(NCHAR(5), @@identity))
163   END
164  -- all strings are now removed. Now we find the first leaf.
```

```
165   WHILE 1=1  --forever until there is nothing more to do
166   BEGIN
167   SELECT @Parent_ID=@Parent_ID+1
168   --find the first object or list by looking for the open bracket
169   SELECT @FirstObject=PATINDEX('%[{[[]%', @json collate SQL_Latin1_General_CP850_Bin)--
    object or array
170
      IF @FirstObject = 0 BREAK
171
      IF (SUBSTRING(@json, @FirstObject, 1)='{')
172
        SELECT @NextCloseDelimiterChar='}', @type='object'
173
      ELSE
174
        SELECT @NextCloseDelimiterChar=']', @type='array'
175
      SELECT @OpenDelimiter=@firstObject
176
      WHILE 1=1 --find the innermost object or list...
177
        BEGIN
178
          SELECT
179
            @lenJSON=LEN(@JSON+'|')-1
180
      --find the matching close-delimiter proceeding after the open-delimiter
181
          SELECT
182
            @NextCloseDelimiter=CHARINDEX(@NextCloseDelimiterChar, @json,
183
                                          @OpenDelimiter+1)
184
      --is there an intervening open-delimiter of either type
185
          SELECT @NextOpenDelimiter=PATINDEX('%[{[[]%',
186
               RIGHT(@json, @lenJSON-@OpenDelimiter)collate SQL_Latin1_General_CP850_Bin)--
187   object
188         IF @NextOpenDelimiter=0
189           BREAK
190         SELECT @NextOpenDelimiter=@NextOpenDelimiter+@OpenDelimiter
191         IF @NextCloseDelimiter<@NextOpenDelimiter
192           BREAK
193         IF SUBSTRING(@json, @NextOpenDelimiter, 1)='{'
194           SELECT @NextCloseDelimiterChar='}', @type='object'
195         ELSE
196           SELECT @NextCloseDelimiterChar=']', @type='array'
197         SELECT @OpenDelimiter=@NextOpenDelimiter
198       END
199   ---and parse out the list or Name/value pairs
200   SELECT
```

```
201     @contents=SUBSTRING(@json, @OpenDelimiter+1,
202                         @NextCloseDelimiter-@OpenDelimiter-1)
203    SELECT
204     @JSON=STUFF(@json, @OpenDelimiter,
205                 @NextCloseDelimiter-@OpenDelimiter+1,
206                 '@'+@type+CONVERT(NCHAR(5), @Parent_ID))
207    WHILE (PATINDEX('%[A-Za-z0-9@+.e]%', @contents collate
SQL_Latin1_General_CP850_Bin))<>0
208
       BEGIN
209
        IF @Type='object' --it will be a 0-n list containing a string followed by a string, number,boolean, or
210 null
211          BEGIN
212            SELECT
213              @SequenceNo=0,@end=CHARINDEX(':', ' '+@contents)--if there is anything, it will be a
string-based Name.
214
           SELECT  @start=PATINDEX('%[^A-Za-z@][@]%', ' '+@contents collate
215 SQL_Latin1_General_CP850_Bin)--AAAAAAAA
216              SELECT @token=RTrim(Substring(' '+@contents, @start+1, @End-@Start-1)),
217             @endofName=PATINDEX('%[0-9]%', @token collate SQL_Latin1_General_CP850_Bin),
218             @param=RIGHT(@token, LEN(@token)-@endofName+1)
219            SELECT
220             @token=LEFT(@token, @endofName-1),
221             @Contents=RIGHT(' '+@contents, LEN(' '+@contents+'|')-@end-1)
222            SELECT  @Name=StringValue FROM @strings
223             WHERE string_id=@param --fetch the Name
224          END
225        ELSE
226           SELECT @Name=null,@SequenceNo=@SequenceNo+1
227        SELECT
228          @end=CHARINDEX(',', @contents)-- a string-token, object-token, list-token, number,boolean, or
229 null
230               IF @end=0
231          --HR Engineering notation bugfix start
232           IF ISNUMERIC(@contents) = 1
233      SELECT @end = LEN(@contents) + 1
234           Else
235          --HR Engineering notation bugfix end
236
```

```
237   SELECT  @end=PATINDEX('%[A-Za-z0-9@+.e][^A-Za-z0-9@+.e]%', @contents+' ' collate
      SQL_Latin1_General_CP850_Bin) + 1
238
        SELECT
239
         @start=PATINDEX('%[^A-Za-z0-9@+.e][A-Za-z0-9@+.e]%', ' '+@contents collate
240   SQL_Latin1_General_CP850_Bin)
241       --select @start,@end, LEN(@contents+'|'), @contents
242       SELECT
243         @Value=RTRIM(SUBSTRING(@contents, @start, @End-@Start)),
244         @Contents=RIGHT(@contents+' ', LEN(@contents+'|')-@end)
245       IF SUBSTRING(@value, 1, 7)='@object'
246         INSERT INTO @hierarchy
247           (Name, SequenceNo, Parent_ID, StringValue, Object_ID, ValueType)
248           SELECT @Name, @SequenceNo, @Parent_ID, SUBSTRING(@value, 8, 5),
249             SUBSTRING(@value, 8, 5), 'object'
250       ELSE
251         IF SUBSTRING(@value, 1, 6)='@array'
252           INSERT INTO @hierarchy
253             (Name, SequenceNo, Parent_ID, StringValue, Object_ID, ValueType)
254             SELECT @Name, @SequenceNo, @Parent_ID, SUBSTRING(@value, 7, 5),
255               SUBSTRING(@value, 7, 5), 'array'
256         ELSE
257           IF SUBSTRING(@value, 1, 7)='@string'
258             INSERT INTO @hierarchy
259               (Name, SequenceNo, Parent_ID, StringValue, ValueType)
260               SELECT @Name, @SequenceNo, @Parent_ID, StringValue, 'string'
261               FROM @strings
262               WHERE string_id=SUBSTRING(@value, 8, 5)
263           ELSE
264             IF @value IN ('true', 'false')
265               INSERT INTO @hierarchy
266                 (Name, SequenceNo, Parent_ID, StringValue, ValueType)
267                 SELECT @Name, @SequenceNo, @Parent_ID, @value, 'boolean'
268             ELSE
269               IF @value='null'
270                 INSERT INTO @hierarchy
271                   (Name, SequenceNo, Parent_ID, StringValue, ValueType)
272                   SELECT @Name, @SequenceNo, @Parent_ID, @value, 'null'
```

```
273              ELSE
274                IF PATINDEX('%[^0-9]%', @value collate SQL_Latin1_General_CP850_Bin)>0
275                  INSERT INTO @hierarchy
276                    (Name, SequenceNo, Parent_ID, StringValue, ValueType)
                       SELECT @Name, @SequenceNo, @Parent_ID, @value, 'real'
                     ELSE
                      INSERT INTO @hierarchy
                       (Name, SequenceNo, Parent_ID, StringValue, ValueType)
                       SELECT @Name, @SequenceNo, @Parent_ID, @value, 'int'
          if @Contents=' ' Select @SequenceNo=0
        END
      END
    INSERT INTO @hierarchy (Name, SequenceNo, Parent_ID, StringValue, Object_ID, ValueType)
      SELECT '-',1, NULL, '', @Parent_ID-1, @type
    --
      RETURN
    END
    GO
```

So once we have a hierarchy, we can pass it to a stored procedure. As the output is an adjacency list, it should be easy to access the data. You might find it handy to create a table type if you are using SQL Server 2008. Here is what I use. (Note that if you drop a Table Valued Parameter type, you will have to drop any dependent functions or procedures first, and re-create them afterwards).

```
1  -- Create the data type  IF EXISTS (SELECT * FROM sys.types WHERE name LIKE 'Hierarchy')
2    DROP TYPE dbo.Hierarchy
3  go
4  CREATE TYPE dbo.Hierarchy AS TABLE
5  (
6    element_id INT NOT NULL, /* internal surrogate primary key gives the order of parsing and the list
   order */
7
     sequenceNo [int] NULL, /* the place in the sequence for the element */
8
     parent_ID INT,/* if the element has a parent then it is in this column. The document is the ultimate
9  parent, so you can get the structure from recursing from the document */
10   [Object_ID] INT,/* each list or object has an object id. This ties all elements to a parent. Lists are
   treated as objects here */
11
     NAME NVARCHAR(2000),/* the name of the object, null if it hasn't got one */
12
     StringValue NVARCHAR(MAX) NOT NULL,/*the string representation of the value of the element. */
13
     ValueType VARCHAR(10) NOT null /* the declared type of the value represented as a string in
14 StringValue*/
      PRIMARY KEY (element_id)
   )
```

ToJSON. A function that creates JSON Documents

Firstly, we need a simple utility function:

```
IF OBJECT_ID (N'dbo.JSONEscaped') IS NOT NULL     DROP FUNCTION dbo.JSONEscaped
GO

CREATE FUNCTION [dbo].[JSONEscaped] ( /* this is a simple utility function that takes a SQL String with all its clobber and outputs it as a sting with all the JSON escape sequences in it.*/
@Unescaped NVARCHAR(MAX) --a string with maybe characters that will break json
)
RETURNS NVARCHAR(MAX)
AS
BEGIN
  SELECT @Unescaped = REPLACE(@Unescaped, FROMString, TOString)
  FROM (SELECT '' AS FromString, '\' AS ToString
        UNION ALL SELECT '"', '"'
        UNION ALL SELECT '/', '/'
        UNION ALL SELECT CHAR(08),'b'
        UNION ALL SELECT CHAR(12),'f'
        UNION ALL SELECT CHAR(10),'n'
        UNION ALL SELECT CHAR(13),'r'
        UNION ALL SELECT CHAR(09),'t'
) substitutions
RETURN @Unescaped
END
GO
```

And now, the function that takes a JSON Hierarchy table and converts it to a JSON string.

```
CREATE FUNCTION ToJSON
(
      @Hierarchy Hierarchy READONLY
)

/*
the function that takes a Hierarchy table and converts it to a JSON string

Author: Phil Factor
Revision: 1.5
```

```
11  date: 1 May 2014
12  why: Added a fix to add a name for a list.
13  example:
14
15  Declare @XMLSample XML
16  Select @XMLSample='
17    <glossary><title>example glossary</title>
18    <GlossDiv><title>S</title>
19     <GlossList>
20      <GlossEntry id="SGML"" SortAs="SGML">
21       <GlossTerm>Standard Generalized Markup Language</GlossTerm>
22       <Acronym>SGML</Acronym>
23       <Abbrev>ISO 8879:1986</Abbrev>
24       <GlossDef>
25        <para>A meta-markup language, used to create markup languages such as DocBook.</para>
26        <GlossSeeAlso OtherTerm="GML" />
27        <GlossSeeAlso OtherTerm="XML" />
28       </GlossDef>
29       <GlossSee OtherTerm="markup" />
30      </GlossEntry>
31     </GlossList>
32    </GlossDiv>
33   </glossary>'
34
35  DECLARE @MyHierarchy Hierarchy -- to pass the hierarchy table around
36  insert into @MyHierarchy select * from dbo.ParseXML(@XMLSample)
37  SELECT dbo.ToJSON(@MyHierarchy)
38
39        */
40  RETURNS NVARCHAR(MAX)--JSON documents are always unicode.
41  AS
42  BEGIN
43   DECLARE
44     @JSON NVARCHAR(MAX),
45     @NewJSON NVARCHAR(MAX),
46     @Where INT,
```

```
47     @ANumber INT,
48     @notNumber INT,
49     @indent INT,
50     @ii int,
51     @CrLf CHAR(2)--just a simple utility to save typing!
52
53    --firstly get the root token into place
54    SELECT @CrLf=CHAR(13)+CHAR(10),--just CHAR(10) in UNIX
55          @JSON = CASE ValueType WHEN 'array' THEN
56          +COALESCE('{'+@CrLf+'  "'+NAME+'" : ','')+'['
57          ELSE '{' END
58            +@CrLf
59            + case when ValueType='array' and NAME is not null then '  ' else '' end
60            + '@Object'+CONVERT(VARCHAR(5),OBJECT_ID)
61            +@CrLf+CASE ValueType WHEN 'array' THEN
62            case when NAME is null then ']' else '  ]'+@CrLf+'}'+@CrLf end
63               ELSE '}' END
64    FROM @Hierarchy
65     WHERE parent_id IS NULL AND valueType IN ('object','document','array') --get the root element
66  /* now we simply iterat from the root token growing each branch and leaf in each iteration. This won't
67  be enormously quick, but it is simple to do. All values, or name/value pairs withing a structure can be
    created in one SQL Statement*/
68    Select @ii=1000
69    WHILE @ii>0
70     begin
71     SELECT @where= PATINDEX('%[^[a-zA-Z0-9]@Object%',@json)--find NEXT token
72     if @where=0 BREAK
73     /* this is slightly painful. we get the indent of the object we've found by looking backwards up the
    string */
74
75     SET
    @indent=CHARINDEX(char(10)+char(13),Reverse(LEFT(@json,@where))+char(10)+char(13))-1
76     SET @NotNumber= PATINDEX('%[^0-9]%', RIGHT(@json,LEN(@JSON+'|')-@Where-8)+' ')--find
    NEXT token
77
78     SET @NewJSON=NULL --this contains the structure in its JSON form
79     SELECT
80        @NewJSON=COALESCE(@NewJSON+','+@CrLf+SPACE(@indent),'')
81        +case when parent.ValueType='array' then '' else COALESCE('"'+TheRow.NAME+'" : ','') end
82        +CASE TheRow.valuetype
```

```
83          WHEN 'array' THEN '  ['+@CrLf+SPACE(@indent+2)
84            +'@Object'+CONVERT(VARCHAR(5),TheRow.[OBJECT_ID])+@CrLf+SPACE(@indent+2)+']'
85          WHEN 'object' then '  {'+@CrLf+SPACE(@indent+2)
86            +'@Object'+CONVERT(VARCHAR(5),TheRow.[OBJECT_ID])+@CrLf+SPACE(@indent+2)+'}'
87          WHEN 'string' THEN '"'+dbo.JSONEscaped(TheRow.StringValue)+'"'
88          ELSE TheRow.StringValue
89         END
90       FROM @Hierarchy TheRow
91       inner join @hierarchy Parent
92       on parent.element_ID=TheRow.parent_ID
93        WHERE TheRow.parent_id= SUBSTRING(@JSON,@where+8, @Notnumber-1)
94       /* basically, we just lookup the structure based on the ID that is appended to the @Object token.
    Simple eh? */
95
      --now we replace the token with the structure, maybe with more tokens in it.
96
      Select @JSON=STUFF (@JSON, @where+1, 8+@NotNumber-1, @NewJSON),@ii=@ii-1
97
      end

    return @JSON

  end

  go
```

ToXML. A function that creates XML

The function that converts a hierarchy table to XML gives us a JSON to XML converter. It is surprisingly similar to the previous function

```
1
2   IF OBJECT_ID (N'dbo.ToXML') IS NOT NULL
3     DROP FUNCTION dbo.ToXML
4   GO
5   CREATE FUNCTION ToXML
6   (
7   /*this function converts a Hierarchy table into an XML document. This uses the same technique as the
    toJSON function, and uses the 'entities' form of XML syntax to give a compact rendering of the structure
8   */
9         @Hierarchy Hierarchy READONLY
10  )
11  RETURNS NVARCHAR(MAX)--use unicode.
12  AS
13  BEGIN
```

```
 DECLARE
   @XMLAsString NVARCHAR(MAX),
   @NewXML NVARCHAR(MAX),
   @Entities NVARCHAR(MAX),
   @Objects NVARCHAR(MAX),
   @Name NVARCHAR(200),
   @Where INT,
   @ANumber INT,
   @notNumber INT,
   @indent INT,
   @CrLf CHAR(2)--just a simple utility to save typing!

  --firstly get the root token into place
  --firstly get the root token into place
  SELECT @CrLf=CHAR(13)+CHAR(10),--just CHAR(10) in UNIX
        @XMLasString ='<?xml version="1.0" ?>
@Object'+CONVERT(VARCHAR(5),OBJECT_ID)+'
'
   FROM @hierarchy
   WHERE parent_id IS NULL AND valueType IN ('object','array') --get the root element
/* now we simply iterate from the root token growing each branch and leaf in each iteration. This won't be enormously quick, but it is simple to do. All values, or name/value pairs within a structure can be created in one SQL Statement*/
  WHILE 1=1
   begin
   SELECT @where= PATINDEX('%[^a-zA-Z0-9]@Object%',@XMLAsString)--find NEXT token
   if @where=0 BREAK
   /* this is slightly painful. we get the indent of the object we've found by looking backwards up the string */

   SET @indent=CHARINDEX(char(10)+char(13),Reverse(LEFT(@XMLasString,@where))+char(10)+char(13))-1
   SET @NotNumber= PATINDEX('%[^0-9]%',
RIGHT(@XMLasString,LEN(@XMLAsString+'|')-@Where-8)+' ')--find NEXT token
   SET @Entities=NULL --this contains the structure in its XML form
   SELECT @Entities=COALESCE(@Entities+' ',' ')+NAME+'="'
    +REPLACE(REPLACE(REPLACE(StringValue, '<', '&lt;'), '&', '&amp;'),'>', '&gt;')
    + '"'
     FROM @hierarchy

```

```
50        WHERE parent_id= SUBSTRING(@XMLasString,@where+8, @Notnumber-1)
51          AND ValueType NOT IN ('array', 'object')
52     SELECT @Entities=COALESCE(@entities,''),@Objects='',@name=CASE WHEN Name='-' THEN 'root'
   ELSE NAME end
53       FROM @hierarchy
54       WHERE [Object_id]= SUBSTRING(@XMLasString,@where+8, @Notnumber-1)
55
56
     SELECT  @Objects=@Objects+@CrLf+SPACE(@indent+2)
57           +'@Object'+CONVERT(VARCHAR(5),OBJECT_ID)
58           --+@CrLf+SPACE(@indent+2)+''
59       FROM @hierarchy
60       WHERE parent_id= SUBSTRING(@XMLasString,@where+8, @Notnumber-1)
61       AND ValueType IN ('array', 'object')
62     IF @Objects='' --if it is a lef, we can do a more compact rendering
63          SELECT @NewXML='<'+COALESCE(@name,'item')+@entities+' />'
64     ELSE
65        SELECT @NewXML='<'+COALESCE(@name,'item')+@entities+'>'
66            +@Objects+@CrLf++SPACE(@indent)+'</'+COALESCE(@name,'item')+'>'
67      /* basically, we just lookup the structure based on the ID that is appended to the @Object token.
68  Simple eh? */
     --now we replace the token with the structure, maybe with more tokens in it.
     Select @XMLasString=STUFF (@XMLasString, @where+1, 8+@NotNumber-1, @NewXML)
     end
   return @XMLasString
   end
```

This provides you the means of converting a JSON string into XML

```
DECLARE @MyHierarchy Hierarchy,@xml XML
INSERT INTO @myHierarchy
select * from parseJSON('{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}')
SELECT dbo.ToXML(@MyHierarchy)
SELECT @XML=dbo.ToXML(@MyHierarchy)
SELECT @XML
```

This gives the result...

```
1
2   <?xml version="1.0" ?>
3   <root>
4    <menu id="file""" value="File">
5      <popup>
6        <menuitem>
7          <item value="New" onclick="CreateNewDoc()" />
8          <item value="Open" onclick="OpenDoc()" />
9          <item value="Close" onclick="CloseDoc()" />
10       </menuitem>
11     </popup>
12   </menu>
13  </root>
14
15
16  (1 row(s) affected)
17
18
19  <root><menu id="file""" value="File"><popup><menuitem><item value="New"
    onclick="CreateNewDoc()" /><item value="Open" onclick="OpenDoc()" /><item value="Close"
20  onclick="CloseDoc()" /></menuitem></popup></menu></root>
21

    (1 row(s) affected)
```

Wrap-up

The so-called 'impedence-mismatch' between applications and databases is, I reckon, an illusion. The object-oriented nested data-structures that we receive from applications are, if the developer has understood the data correctly, merely a perspective from a particular entity of the relationships it is involved with. Whereas it is easy to shred XML documents to get the data from it to update the database, it has been trickier with other formats such as JSON. By using techniques like this, it should be possible to liberate the application, or website, programmer from having to do the mapping from the object model to the relational, and spraying the database with ad-hoc TSQL that uses the base tables or updateable views. If the database can be provided with the JSON, or the Table-Valued parameter, then there is a better chance of maintaining full transactional integrity for the more complex updates.

The database developer already has the tools to do the work with XML, but why not the simpler, and more practical JSON? I hope these two routines get you started with experimenting with this.

Interesting JSON-related articles and sites

Since writing this article, Phil has also developed a CSV parser and output and an XML parser (Producing JSON Documents from SQL Server queries via TSQL)