# Importing JSON Data from Web Services and Applications into SQL Server

red-gate.com/simple-talk/sql/t-sql-programming/importing-json-web-services-applications-sql-server

October 27, 2017

Phil Factor
27 October 2017



30

To support many applications, it makes sense for the database to work with JSON data, because it is the built-in way for a JavaScript or TypeScript application to represent object data. It can mean less network traffic, looser coupling, and less need for the application developer to require full access to the base tables of the database. However, it means that the database must do plenty of checks first before importing. Phil Factor explains how it can be easily done.

### Articles by Phil Factor about JSON and SQL Server:

There are a number of reasons why database developers or DBAs will regard JSON with suspicion. They are used to the protection of a higher level of constraints and checks for imported data: They want to be sure that the metadata of the data is correct before they import it: They also want to check the data itself. At the simplest level, they want to be confident that a regular feed from a data source is as robust as possible. Why so fussy? Simply because once bad data gets into a database of any size, it is tiresome and time-consuming to extract it.

As well as taking JSON data from web-based applications, especially single-page ones, many databases rely on volatile data from web services, such as a list of currencies and their current value against the dollar. We need to be sure that each time it happens, the JSON has the same metadata or data structure as usual, and that the data is valid.
Although nowadays, application data generally comes as a JSON document, we have a similar problem with other types of document-based data: XML, for example, exists in two variations: Schema-based or typed XML which is good and virtuous but has a naughty sister, schema-less or untyped XML. Typed XML is the only sort of XML you should allow in a SQL Server database. Not only is it safer to use, but it is stored much more efficiently. Untyped XML, JSON and YAML all require checks.
We'd like to do the same with JSON as we do with typed XML, and allow JSON Schema to do the donkey work; but, sadly, SQL Server don't currently support any JSON schema

binding and cannot, therefore, store JSON efficiently in a 'compiled' form. (note: you can get 25x compression of JSON data, and far better performance by using clustered columnstore indexes, but that is a different story)

The dominant JSON Schema is actually called 'JSON Schema' and is in IETF draft 6. It can describe a data format in human-readable JSON and can be used to provide a complete structural validation. PostgreSQL has **postgres-json-schema** for validating JSON, JavaScript has many add-ins for doing this. Net programmers have Json.NET Schema and several other alternatives.

PowerShell is the obvious place to validate your JSON via JSON.net schema, but there is, in fact, quite a lot you can do in SQL to check your JSON data. You can easily compare the metadata of two JSON documents to see if anything has changed, and you can apply constraints on JSON Data. This is because of the use of path expressions. To get started we'd better describe them, because they are essential to any serious use of JSON in SQL Server.

## Path expressions

When you need to read from, or write to, values from a JSON document in SQL Server, you use JSON path expressions. These can reference objects, arrays or values.

These path expressions are needed if you call OPENJSON with the WITH clause to return a relational table-source, if you call JSON_VALUE to extract a single value from JSON text, or when you call JSON_MODIFY to update this value or append to it.

The path itself starts with a dollar sign ($) that represents the context item. The path consists of elements separated by 'dots' or full-stops.

The property path is a set of path steps that consist of Key names that are optionally 'quoted'. (e.g. **$.info.address.county** or **$.info.address."post code".outward**). Each dot denotes that the Lvalue (left-side value) is the parent of the RValue (right-side value). If the key name starts with a dollar sign or contains special characters such as spaces then you need to use double-quoted delimiters for the key (e.g. **$.info.Cost."La Posche Hotel"**). If the key name refers to an array, they are indexed with a zero-based index. (e.g. **$.info.tags[0]** or **$.info.tags[1]**)

If the path references an object that exists more than once, neither **JSON_Value** nor **JSON_modify** can access the second or subsequent values. In this case, you have to use **OpenJSON** instead to get to all the values.

JSON paths in SQL Server can start with a keyword 'lax' or 'strict'. It is an unusual requirement to want to suppress errors if a path isn't found in the JSON document but this is the default, and you can specify this by using 'lax'. You'd be more likely to want 'strict' mode, but 'lax' is better if you want to test whether a path value is there because you will know something is wrong by the NULL return value

You can easily use path expressions. The first function you'd probably need is a routine to tell you what these expressions actually are for any particular JSON document.

## Finding out what paths there are in a JSON string or document

Here is a routine that takes a JSON string and returns a table-source containing the expressions, data types and values for the JSON that you specify.

```
1  IF Object_Id('dbo.JSONPathsAndValues') IS NOT NULL DROP FUNCTION
   dbo.JSONPathsAndValues;
2
     GO
3
     CREATE FUNCTION dbo.JSONPathsAndValues
4
       /**
5
       Summary: >
6
        This function takes a JSON string and returns
7
       a table containing the JSON paths to the data,
8
       and the data itself. The JSON paths are compatible with  OPENjson,
9
       JSON_Value and JSON_Modify.
10
       Author: PhilFactor
11
       Date: 06/10/2017
12
       Version: 2
13
       Database: PhilFactor
14
       Examples:
15
          - Select * from dbo.JSONPathsAndValues(N'{"person":{"info":
16 {"name":"John", "name":"Jack"}}}')
17        - Select * from MyTableWithJson cross apply
   dbo.JSONPathsAndValues(MyJSONColumn)
18
       Returns: >
19
       A table listing the paths to all the values in the JSON document
20
       with their type and their order and nesting depth in the document
21
      **/
22
       (@JSONData NVARCHAR(MAX))
23
     RETURNS @TheHierarchyMetadata TABLE
24
```

```
25      (
26      -- columns returned by the function
27      element_id INT NOT NULL,
28      Depth INT NOT NULL,
29      Thepath NVARCHAR(2000),
30      ValueType VARCHAR(10) NOT NULL,
31      TheValue NVARCHAR(MAX) NOT NULL
32      )
33    AS
34      -- body of the function
35      BEGIN
36        DECLARE @ii INT = 1, @rowcount INT = -1;
37        DECLARE @null INT = 0, @string INT = 1, @int INT = 2, --
38          @boolean INT = 3, @array INT = 4, @object INT = 5;
39        DECLARE @TheHierarchy TABLE
40          (
41          element_id INT IDENTITY(1, 1) PRIMARY KEY,
42          Depth INT NOT NULL, /* effectively, the recursion level. =the depth of
    nesting*/
43          Thepath NVARCHAR(2000) NOT NULL,
44          TheName NVARCHAR(2000) NOT NULL,
45          TheValue NVARCHAR(MAX) NOT NULL,
46          ValueType VARCHAR(10) NOT NULL
47          );
48        INSERT INTO @TheHierarchy
49          (Depth, Thepath, TheName, TheValue, ValueType)
50          SELECT @ii, '$', '$', @JSONData, 'object';
51        WHILE @rowcount <> 0
52          BEGIN
53
```

```
54          SELECT @ii = @ii + 1;
55          INSERT INTO @TheHierarchy
56            (Depth, Thepath, TheName, TheValue, ValueType)
57            SELECT @ii,
58              CASE WHEN [Key] NOT LIKE '%[^0-9]%' THEN Thepath + '[' + [Key] +
    ']' --nothing but numbers
59
                WHEN [Key] LIKE '%[$ ]%' THEN Thepath + '."' + [Key] + '"' --got a
60  space in it
61                ELSE Thepath + '.' + [Key] END, [Key], Coalesce(Value,''),
62              CASE Type WHEN @string THEN 'string'
63                WHEN @null THEN 'null'
64                WHEN @int THEN 'int'
65                WHEN @boolean THEN 'boolean'
66                WHEN @int THEN 'int'
67                WHEN @array THEN 'array' ELSE 'object' END
68            FROM @TheHierarchy AS m
69              CROSS APPLY OpenJson(TheValue) AS o
70            WHERE ValueType IN
71           ('array', 'object') AND Depth = @ii - 1;
72           SELECT @rowcount = @@RowCount;
73         END;
74       INSERT INTO @TheHierarchyMetadata
75         SELECT element_id, Depth, Thepath, ValueType, TheValue
76         FROM @TheHierarchy
77         WHERE ValueType NOT IN
       ('array', 'object');
       RETURN;
     END;
   GO
```

With this, you can see what paths lead to the keys, and you can see what is in the values of those keys.

```
DECLARE @JSONData NVARCHAR(4000) = N'
 {
    "info": {
       "type": 1,
       "address": {
          "town": "Colchester",
          "county": "Essex",
          "country": "England"
       },
       "Hotels": {
          "La Posche Hotel": "$400",
          "The Salesmans Rest": "$35",
          "The Middling Inn": "$100"}
    },
    "Sights": ["the Castle","The Barracks","the Hythe","St Bartolphs"]
 } ';

 SELECT * from dbo.JSONPathsAndValues(@JSONData)
```

Which would give you this...

| element_id | Depth | Thepath | ValueType | TheValue |
|---|---|---|---|---|
| 4 | 3 | $.info.type | int | 1 |
| 7 | 3 | $.Sights[0] | string | the Castle |
| 8 | 3 | $.Sights[1] | string | The Barracks |
| 9 | 3 | $.Sights[2] | string | the Hythe |
| 10 | 3 | $.Sights[3] | string | St Bartolphs |
| 11 | 4 | $.info.address.town | string | Colchester |
| 12 | 4 | $.info.address.county | string | Essex |
| 13 | 4 | $.info.address.country | string | England |
| 14 | 4 | $.info.Hotels."La Posche Hotel" | string | $400 |
| 15 | 4 | $.info.Hotels."The Salesmans Rest" | string | $35 |
| 16 | 4 | $.info.Hotels."The Middling Inn" | string | $100 |

## Differences between the metadata of two JSON strings

It is very easy to check two JSON strings to make sure that they have the same metadata, and report any differences that it finds.

```
1   IF Object_Id('dbo.DifferenceBetweenJSONstrings') IS NOT NULL
2       DROP function dbo.DifferenceBetweenJSONstrings
3   GO
4   CREATE FUNCTION dbo.DifferenceBetweenJSONstrings
5   /**
6   Summary: >
7     This checks two JSON strings and  returns
8     a table listing any differences in the schema between them
9   Author: PhilFactor
10  Date: 20/10/2017
11  Database: PhilFactor
12  Examples:
13     - Select * from dbo.DifferenceBetweenJSONstrings(@Original,@new)
14     - Select  from MyTable cross apply
    dbo.DifferenceBetweenJSONstrings(FirstJ)
15
    Returns: >
16
      A table
17
          **/
18
      (
19
      @Original nvarchar (max),-- the original JSON string
20
      @New nvarchar (max)      -- the New JSON string
21
      )
22
    RETURNS TABLE
23
     --WITH ENCRYPTION|SCHEMABINDING, ..
24
    AS
25
    RETURN
26
```

```
27      (
28      SELECT Coalesce(old.thePath, new.thepath) AS JSONpath,
29        Coalesce(old.valuetype, '')
30        + CASE WHEN old.valuetype + new.valuetype IS NOT NULL THEN ' \ ' ELSE
    '' END
31
          + Coalesce(new.valuetype, '') AS ValueType,
32
        CASE WHEN old.valuetype + new.valuetype IS NOT NULL THEN 'value type
33  changed'
34          WHEN old.thePath IS NULL THEN 'added or key changed'
35          WHEN new.thePath IS NULL THEN 'missing' ELSE 'dunno' END AS
    TheDifference
36
        FROM dbo.JSONPathsAndValues(@New) AS new
37
          FULL OUTER JOIN dbo.JSONPathsAndValues(@original) AS old
38
            ON old.ThePath = new.ThePath --AND old.Valuetype=new.Valuetype
39
        WHERE old.thepath IS NULL OR new.thepath IS NULL OR old.ValueType <>
40  new.ValueType


      );

    Go
```

As part of the build, we'd run some code like this to ensure that the function still does what you expect

```
1    Select * from dbo.DifferenceBetweenJSONstrings(
2      '[12,32,35,49,15,56,37]','[1,2,3,4,5,6]')
3        Select * from dbo.DifferenceBetweenJSONstrings(
4      '[1,2,3,4,5,6]','[1,"2",3,4,5,6]')
5          Select * from dbo.DifferenceBetweenJSONstrings(
6      '{"id": "001","type": "Coupe","name": "Cougar","year":
   "2012"}',
7
       '{"id": "004","type": "Coupe","name": "Jaguar","year": "2012"}')
8
           Select * from dbo.DifferenceBetweenJSONstrings(
9
       '{"id": "001","type": "Coupe","name": "Cougar","year":
10 "2012"}',
       '{"id": "001","type": "Coupe","name": "Cougar","start":
   "2012"}')
```

Actually, to be honest, I wouldn't do my tests this way in order to make sure the function works. I'd run all the results into a table and make sure that the entire table was what I'd expect. After all, we all do automated unit tests as part of our daily build don't we?

```
1   DECLARE @TestData TABLE (JSONpath NVARCHAR(MAX),ValueType
    VARCHAR(20), TheDifference VARCHAR(20))
2
     INSERT INTO @testdata
3
    SELECT JSONpath,ValueType,TheDifference
4
     FROM dbo.DifferenceBetweenJSONstrings(
5
        '[12,32,35,49,15,56,37]','[1,2,3,4,5,6]')
6
     UNION ALL SELECT JSONpath,ValueType,TheDifference
7
     FROM dbo.DifferenceBetweenJSONstrings(
8
        '[1,2,3,4,5,6]','[1,"2",3,4,5,6]')
9
     UNION all Select JSONpath,ValueType,TheDifference
10
     FROM dbo.DifferenceBetweenJSONstrings(
11
    '{"id": "001","type": "Coupe","name": "Cougar","year": "2012"}',
12
    '{"id": "004","type": "Coupe","name": "Jaguar","year": "2012"}')
13
     UNION all  Select JSONpath,ValueType,TheDifference
14
     FROM dbo.DifferenceBetweenJSONstrings(
15
    '{"id": "001","type": "Coupe","name": "Cougar","year": "2012"}',
16
    '{"id": "001","type": "Coupe","name": "Cougar","start": "2012"}')
17
   IF (EXISTS(
18
    SELECT * FROM @testData g
19
      FULL OUTER JOIN
20
       (VALUES('$[6]','int','missing'),
21
         ('$[1]','int \ string','value type changed'),
22
         ('$.START','string','added or key changed'),
23
         ('$.year','string','missing'))f(JSONpath,ValueType,TheDifference)
24
      ON f.JSONpath=g.JSONpath
25
    WHERE f.JSONpath IS NULL OR g.JSONpath IS NULL))
26
   RAISERROR ('the dbo.DifferenceBetweenJSONstrings routine is
27
   giving unexpected results;',16,1)
```

OK. We can now compare the metadata of two JSON strings, but we can compare the values as well if we ever need to check that two JSON documents represent the same data.

## Checking that the JSON is what you expect

Many Web Services send information messages or warning messages in JSON format instead of the data you expect. It could contain a variety of messages such as service interruption, subscription terminations, or daily credit-limits reached. These need to be logged and you need to bypass the import routine. Probably the quickest way to check your JSON is to do a simple JSON_VALUE call on a key/value pair that needs to be there, to see if you get NULL back. To get that path in the first place, you can use the **JSONPathsAndValues** function. However, it is possible that there are a number of key/value pairs that need to be there. There are plenty of ways of doing this if you have a list or table of the paths you need. I'll use a VALUES table-source to illustrate the point.

```
1   DECLARE @json NVARCHAR(MAX) = N'[
2     {
3       "Order": {
4         "Number":"SO43659",
5         "Date":"2011-05-31T00:00:00"
6       },
7       "AccountNumber":"AW29825",
8       "Item": {
9         "Price":2024.9940,
10        "Quantity":1
11      }
12    },
13    {
14      "Order": {
15        "Number":"SO43661",
16        "Date":"2011-06-01T00:00:00"
17      },
18      "AccountNumber":"AW73565",
```

```
      "Item": {
        "Price":2024.9940,
        "Quantity":3
      }
    }
  ]'
  IF EXISTS(
  SELECT thepath
  FROM
  (VALUES
  ('$[0].AccountNumber'),
  ('$[0].Order.Number'),
  ('$[0].Order.Date'),
  ('$[0].Item.Price'),
  ('$[0].Item.Quantity'))WhatThereShouldBe(path)
  LEFT OUTER JOIN  dbo.JSONPathsAndValues( @json )
  ON WhatThereShouldBe.path=ThePath
  WHERE ThePath IS NULL)
  RAISERROR ('an essential key is missing',16,1)
```

## Comparing values as well as metadata

A very small tweak in the code for checking the keys will allow you to compare the values as well if you ever need to do that. Note that it checks the order of lists too.

```
1  SELECT FirstVersion.Thepath, SecondVersion.Thepath, FirstVersion.TheValue,
   SecondVersion.TheValue
2
     FROM dbo.JSONPathsAndValues('[1,2,3,4,5,7,8]') AS FirstVersion
3
       FULL OUTER JOIN dbo.JSONPathsAndValues('[1,2,3,4,5,6]') AS
4  SecondVersion
5        ON FirstVersion.Thepath = SecondVersion.Thepath
6     WHERE FirstVersion.TheValue <> SecondVersion.TheValue
7       OR FirstVersion.Thepath IS NULL
        OR SecondVersion.Thepath IS NULL;
```

| | Thepath | Thepath | TheValue | TheValue |
|---|---|---|---|---|
| 1 | $[5] | $[5] | 7 | 6 |
| 2 | $[6] | NULL | 8 | NULL |

As you can see, there are two differences between the JSON strings. There are different values for the same array element (line 1 of result) and there are a different number of array elements (line 2).

## Checking for valid SQL Datatypes.

That is OK as far as it goes, but what about checking that the data will actually go into SQL Server. Although there are good practices for the storage of dates in JSON, for example, there is no JSON standard. If we know our constraints, it is dead easy to check. Imagine we have some JSON which is of a particular datatype. We just define the paths of the values that we want to check and perform whatever check we need on the JSON. We can demonstrate this technique.

```
SELECT g.ThePath, g.TheValue, CASE Coalesce(f.Datatype,'')
  WHEN '' THEN  '' --it hasnt had a check specified
  WHEN 'int' THEN --need to check that it is a valid int
  CASE WHEN Try_Convert(INT,g.TheValue) IS NULL THEN 'Bad int' ELSE 'good' end
  WHEN 'DateTime' THEN --need to check that it is a valid DateTime
  CASE WHEN Try_Convert(DateTime,g.TheValue) IS NULL THEN 'Bad datetime' ELSE 'good' end
  WHEN 'Money' THEN --need to check that it is a valid Money value
  CASE WHEN Try_Convert(Money,g.TheValue) IS NULL THEN 'Bad money value' ELSE 'good' END
        WHEN 'ISO8601' THEN --need to check that it is a valid ISO8601 datetime
  CASE WHEN Try_Convert(DateTime,g.TheValue) IS NULL THEN 'Bad ISO8601 datetime' ELSE 'good' end
    ELSE '' end

  FROM dbo.JSONPathsAndValues('["1.7","2","23/4/2008","1 Jun 2017","5.6d","$456,000","2017-09-12T18:26:20.000","2017/10/20T18:26:20.000"]')g
  LEFT outer JOIN
    (VALUES('$[0]','int'),
    ('$[1]','int'),
      ('$[2]','DateTime'),
    ('$[3]','DateTime'),
    ('$[4]','money'),
    ('$[5]','money'),
    ('$[6]','ISO8601'),
    ('$[7]','ISO8601'))f(Thepath,DataType)
    ON f.thepath=g.Thepath
```

Which, in this test case would give...

| | ThePath | TheValue | (No column name) |
|---|---|---|---|
| 1 | $[0] | 1.7 | Bad int |
| 2 | $[1] | 2 | good |
| 3 | $[2] | 23/4/2008 | Bad datetime |
| 4 | $[3] | 1 Jun 2017 | good |
| 5 | $[4] | 5.6d | Bad money value |
| 6 | $[5] | $456,000 | good |
| 7 | $[6] | 2017-09-12T18:26:20.000 | good |
| 8 | $[7] | 2017/10/20T18:26:20.000 | Bad ISO8601 datetime |

You will have appreciated that, as well as the JSON, you will need a separate table source to say what type of data each value should be, and I've done a custom date format to show that you can refine your constraint. I've used a VALUES table source, but you can easily swap in a JSON one. This table source needs to be kept in sync with the case statement for it to work. In a working system, you'd want to encapsulate all this in a function.

## Checking whether the values will pass constraint checks

So how would you tackle the task of checking this simple example to make sure that all these IP addresses were valid? For this example we'll just check that there are just three dots. We don't want to check the name, obviously.

```
{
   "name":[
      "Philip",
      "Mildew",
      "Factor"
   ],
   "ipAddress":[
      "80.243.543.4",
      "45.85.678.68",
      "5.8.7.9",
      "192.168.0.123",
      "34.8.8"
   ]
}
```

You wouldn't want that last IP address in your database. It isn't valid. You only want to check those IP values. Here, you can very easily run the check.

```
SELECT ThePath, TheValue
  FROM dbo.JSONPathsAndValues('{"name":
["Philip","Mildew","Factor"],"ipAddress":[
"80.243.543.4","45.85.678.68","5.8.7.9","192.168.0.123","34.8.8"]}')
  WHERE ThePath LIKE '$.ipAddress%' AND TheValue NOT LIKE  '%.%.%.%'
```

And you will see the bad IP address, but it only runs the check on the list of IP addresses, which is what you want.

If you had a number of checks to do, you'd save the table-source as a table variable or temporary table and run several queries on it.

| ThePath | TheValue |
|---|---|
| $.ipAddress[4] | 34.8.8 |

## Scaling things up

Normally you are dealing with large amounts of JSON data, so you only want to parse it once into a temporary table or table variable before running all your metadata checks. The advantage of doing it this way is that, once you've put a good primary key on the path (beware of duplicate JSON keys: they are valid JSON – RFC 4627), the process of firstly checking that the data can be successfully coerced into the appropriate column of the destination table, and then checked that it is within bounds before finally unpicking the hierarchical JSON data into all the relational tables in the right order is much easier and well-controlled.

## Using a JSON Webservice from SQL Server

Normally, you'd use SSIS or Powershell for a regular production data feed, but it is certainly possible to do it in SQL. The downside is that your SQL Server needs to have internet access, which is a security risk, and also you have to open up your security surface-area by allowing OLE automation. That said, this is how you do it.

```
IF NOT EXISTS (SELECT * FROM sys.configurations WHERE name ='Ole
Automation Procedures' AND value=1)
  BEGIN
    EXECUTE sp_configure 'Ole Automation Procedures', 1;
    RECONFIGURE;
    end
 SET ANSI_NULLS ON;
```

```
7       SET QUOTED_IDENTIFIER ON;
8       GO
9       IF Object_Id('dbo.GetWebService','P') IS NOT NULL
10       DROP procedure dbo.GetWebService
11      GO
12      CREATE PROCEDURE dbo.GetWebService
13        @TheURL VARCHAR(255),-- the url of the web service
14        @TheResponse NVARCHAR(4000) OUTPUT --the resulting JSON
15      AS
16        BEGIN
17          DECLARE @obj INT, @hr INT, @status INT, @message VARCHAR(255);
18          /**
19      Summary: >
20        This is intended for using web services that
21        utilize JavaScript Object Notation (JSON). You pass it the link to
22        a webservice and it returns the JSON string
23      Note: >
24        OLE Automation objects can be used within a Transact-SQL batch, but
25        SQL Server blocks access to OLE Automation stored procedures because
26        this component is turned off as part of the security configuration.
27
28      Author: PhilFactor
29      Date: 26/10/2017
30      Database: PhilFactor
31      Examples:
32          - >
33          DECLARE @response NVARCHAR(MAX)
34          EXECUTE dbo.GetWebService 'http://headers.jsontest.com/', @response
    OUTPUT
35
```

```
36      SELECT  @response
37    Returns: >
38      nothing
39    **/
40        EXEC @hr = sp_OACreate 'MSXML2.ServerXMLHttp', @obj OUT;
41        SET @message = 'sp_OAMethod Open failed';
42        IF @hr = 0 EXEC @hr = sp_OAMethod @obj, 'open', NULL, 'GET',
  @TheURL, false;
43
          SET @message = 'sp_OAMethod setRequestHeader failed';
44
          IF @hr = 0
45
            EXEC @hr = sp_OAMethod @obj, 'setRequestHeader', NULL, 'Content-
46  Type',
47            'application/x-www-form-urlencoded';
48        SET @message = 'sp_OAMethod Send failed';
49        IF @hr = 0 EXEC @hr = sp_OAMethod @obj, send, NULL, '';
50        SET @message = 'sp_OAMethod read status failed';
51        IF @hr = 0 EXEC @hr = sp_OAGetProperty @obj, 'status', @status OUT;
52        IF @status <> 200 BEGIN
53                        SELECT @message = 'sp_OAMethod http status ' +
  Str(@status), @hr = -1;
54
            END;
55
          SET @message = 'sp_OAMethod read response failed';
56
          IF @hr = 0
57
            BEGIN
58
              EXEC @hr = sp_OAGetProperty @obj, 'responseText', @Theresponse
59  OUT;
60            END;
61        EXEC sp_OADestroy @obj;
          IF @hr <> 0 RAISERROR(@message, 16, 1);
          END;
        GO
```

To use this is simple. You can use this on any of the JSONTest samples

```
DECLARE @response NVARCHAR(MAX)
 EXECUTE dbo.GetWebService 'http://headers.jsontest.com/', @response OUTPUT
 SELECT  @response
```

Which will give you ...

```
{
    "X-Cloud-Trace-Context": "54e570f5620dc6ef3b087ac6042dca03/10421626717945848480",
    "Host": "headers.jsontest.com",
    "User-Agent": "Mozilla/4.0 (compatible; Win32; WinHttp.WinHttpRequest.5)",
    "Accept": "*/*",
    "Content-Type": "application/x-www-form-urlencoded"
 }
```

Now that we have the means to get a real service we can try it out. To try out the next example, you need to register with GeoNames.org, but they are good people running a free and useful service. (If you decide to use it in production, make sure you buy support).

Imagine that you need to find the exact geographical coordinated for any postal code in the world. You can now do this, or a whole range of geographical services. For this example, though, we'll just list the capitals of all the countries in a defined area, together with their populations and longitude/latitude coordinates.

This web service is liable to send you messages instead of your data so be sure to check and log these.

```
status": {
   "message": "the hourly limit of 2000 credits for demo has been exceeded. Please use an application specific account. Do not use the demo account for your application.",
   "value": 19
  }}
```

In the following batch, we are merely showing this sort of message as an error.

```
Msg 50000, Level 16, State 1, Line 30
 The import failed ({"status": {
   "message": "the hourly limit of 2000 credits for demo has been exceeded. Please use an application specific account. Do not use the demo account for your application.",
   "value": 19
  }})
```

In this case, I should have used my own account rather than demo mode. Although the 'demo' name in the URL will work a few times per hour, you will need to change this for your own registered name if you're getting stuck in.

```
DECLARE @response NVARCHAR(4000);
  --get the data from the provider as JSON
  EXECUTE dbo.GetWebService 'http://api.geonames.org/citiesJSON?formatted=false&north=44.1&south=-9.9&east=-22.4&west=55.2&username=demo&style=full',
    @response OUTPUT;
  --now check to see if it is all there.
  IF EXISTS
    (
    SELECT * FROM dbo.JSONPathsAndValues(@response)
      WHERE Thepath IN
  ('$.geonames[0].lng', '$.geonames[0].geonameId', '$.geonames[0].countrycode', '$.geonames[0].name',

```

```
13      '$.geonames[0].fclName', '$.geonames[0].toponymName',
'$.geonames[0].fcodeName',
14
      '$.geonames[0].wikipedia', '$.geonames[0].lat', '$.geonames[0].fcl',
15  '$.geonames[0].population',
16      '$.geonames[0].fcode'
17    )
18      )
19      BEGIN
20        SELECT CountryCode, name, population, latitude, longitude, id,
WikipediaURL
21
          FROM OpenJson(@response) --we have to walk to the level of the array
22  that
23       --we are interested in. OPENjson doesn't support accessing an array at
24       --a higher level when using the WITH clause.
25            OUTER APPLY
26          OpenJson(Value)
27          WITH
28            (CountryCode CHAR(2) '$.countrycode', Latitude NUMERIC(38, 15) '$.lat',
29            Longitude NUMERIC(38, 15) '$.lng', Name VARCHAR(200) '$.name',
30            Population BIGINT '$.population', wikipediaURL VARCHAR(200)
'$.wikipedia',
            id INT '$.geonameId'
            );
      END;
    ELSE RAISERROR('The import failed (%s)', 16, 1, @response);
```

If all is well, this will give you a result something like this...

| | CountryCode | name | population | latitude | longitude | id | WikipediaURL |
|---|---|---|---|---|---|---|---|
| 1 | MX | Mexico City | 12294193 | 19.428472427036000 | -99.127664566040000 | 3530597 | en.wikipedia.org/wiki/Mexico_City |
| 2 | CN | Beijing | 11716620 | 39.907497741440500 | 116.397228240967000 | 1816670 | en.wikipedia.org/wiki/Beijing |
| 3 | BD | Dhaka | 10356500 | 23.710395616597037 | 90.407438278198240 | 1185241 | en.wikipedia.org/wiki/Dhaka |
| 4 | KR | Seoul | 10349312 | 37.566000000000000 | 126.978400000000000 | 1835848 | en.wikipedia.org/wiki/Seoul |
| 5 | ID | Jakarta | 8540121 | -6.214623197035775 | 106.845130920410160 | 1642911 | en.wikipedia.org/wiki/Jakarta |
| 6 | JP | Tokyo | 8336599 | 35.689500000000000 | 139.691710000000000 | 1850147 | en.wikipedia.org/wiki/Tokyo |
| 7 | TW | Taipei | 7871900 | 25.047763000000000 | 121.531846000000000 | 1668341 | en.wikipedia.org/wiki/Taipei |
| 8 | CO | Bogotá | 7674366 | 4.609705849789108 | -74.081754684448240 | 3688689 | en.wikipedia.org/wiki/Bogot%C3%A1 |
| 9 | HK | Hong Kong | 7012738 | 22.278320000000000 | 114.174690000000000 | 1819729 | en.wikipedia.org/wiki/Hong_Kong |
| 10 | TH | Bangkok | 5104476 | 13.753979000000000 | 100.501444000000000 | 1609350 | en.wikipedia.org/wiki/Bangkok |

Of course, there is a lot more you can do. You can, for example, check that the data fits the datatypes in the table, and do whatever other constraint checks you need.

You can of xcourse dispense with the gymnastics of that last bit of openJSON by using what is in the output of **dbo.JSONPathsAndValues**

```
DECLARE @response NVARCHAR(4000);
  --get the data from the provider as JSON
  EXECUTE dbo.GetWebService 'http://api.geonames.org/citiesJSON?formatted=false&north=44.1&south=-9.9&east=-22.4&west=55.2&username=demo&style=full',
    @response OUTPUT;
  DECLARE @TheData  table (
    -- columns returned by the function
    element_id INT NOT NULL,
    Depth INT NOT NULL,
    Thepath NVARCHAR(2000),
    ValueType VARCHAR(10) NOT NULL,
    TheValue NVARCHAR(MAX) NOT NULL
    )
  INSERT INTO @TheData SELECT * FROM dbo.JSONPathsAndValues(@response)
  SELECT
  Max(Convert(CHAR(2),CASE WHEN Thepath LIKE '%.countrycode' THEN Thevalue ELSE '' END)) AS countycode,
  Max(Convert(NVARCHAR(200),CASE WHEN Thepath LIKE '%.name' THEN Thevalue ELSE '' END)) AS name,
  Max(Convert(NUMERIC(38, 15),CASE WHEN Thepath LIKE '%.lat' THEN Thevalue ELSE '-90' END)) AS latitude,
  Max(Convert(NUMERIC(38, 15),CASE WHEN Thepath LIKE '%.lng' THEN Thevalue ELSE '-180' END)) AS longitude,
  Max(Convert(BigInt,CASE WHEN Thepath LIKE '%.population' THEN Thevalue ELSE '-1' END)) AS population,
  Max(Convert(VARCHAR(200),CASE WHEN Thepath LIKE '%.wikipedia' THEN Thevalue ELSE '' END)) AS wikipediaURL,
  Max(Convert (INT,CASE WHEN Thepath LIKE '%.geonameId' THEN Thevalue ELSE '0' END)) AS ID
  FROM @TheData GROUP BY Left(ThePath,CharIndex(']',ThePath+']'))
```

This gives the same result but saving some parsing.

| countycode | name | latitude | longitude | population | wikipediaURL | ID |
|---|---|---|---|---|---|---|
| MX | Mexico City | 19.428472427036000 | -99.127664566040000 | 12294193 | en.wikipedia.org/wiki/Mexico_City | 3530597 |
| CN | Beijing | 39.907497741440500 | 116.397228240967000 | 11716620 | en.wikipedia.org/wiki/Beijing | 1816670 |
| BD | Dhaka | 23.710395616597037 | 90.407438278198240 | 10356500 | en.wikipedia.org/wiki/Dhaka | 1185241 |
| KR | Seoul | 37.566000000000000 | 126.978400000000000 | 10349312 | en.wikipedia.org/wiki/Seoul | 1835848 |
| ID | Jakarta | -6.214623197035775 | 106.845130920410160 | 8540121 | en.wikipedia.org/wiki/Jakarta | 1642911 |
| JP | Tokyo | 35.689500000000000 | 139.691710000000000 | 8336599 | en.wikipedia.org/wiki/Tokyo | 1850147 |
| TW | Taipei | 25.047763000000000 | 121.531846000000000 | 7871900 | en.wikipedia.org/wiki/Taipei | 1668341 |
| CO | Bogotá | 4.609705849789108 | -74.081754684448240 | 7674366 | en.wikipedia.org/wiki/Bogot%C3%A1 | 3688689 |
| HK | Hong Kong | 22.278320000000000 | 114.174690000000000 | 7012738 | en.wikipedia.org/wiki/Hong_Kong | 1819729 |
| TH | Bangkok | 13.753979000000000 | 100.501444000000000 | 5104476 | en.wikipedia.org/wiki/Bangkok | 1609350 |

## Summary

I've set out to show how you can run all manner of checks before importing some JSON into SQL Server, such as ensuring that the JSON metadata is what you expect, finding out the full paths of the information you want, or checking that the values are valid for your datatypes and within range. It is easy to work out what is in a JSON document without having to inspect it. By showing you a simple feed from a web service, I hope I've shown how you can make it as robust as you require. There is a lot more you can do, of course, but I hope I've shown you enough to enable you to feel confident about accepting data in JSON format.