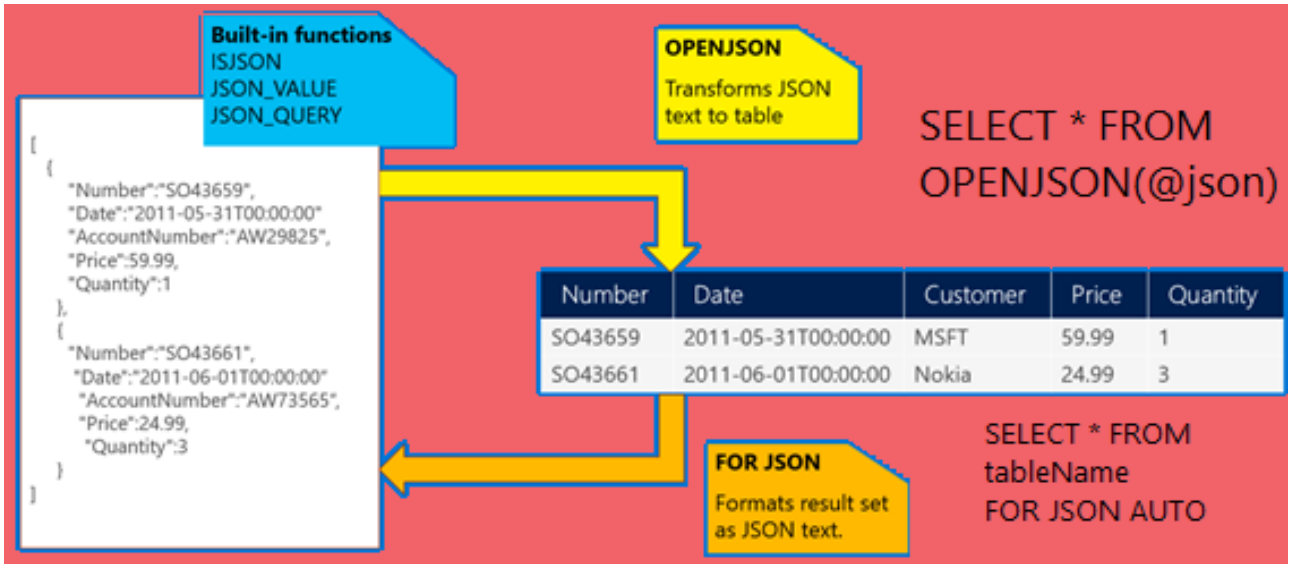
WORKING WITH JSON

IN SQL SERVER 2016

https://database.guide/sql-server-for-json-auto-examples-t-sql/

Abstract

All the important aspects of working with JSON in MS SQL Server 2016+



OPENJSON

JSON input:

```
{
  "Orders":
    [
      {
         "Order": {
               "Number":"SO43659",
               "Date":"2011-05-31T00:00:00"
            },
         "Account": "Microsoft",
         "Item": {
               "Price":59.99,
               "Quantity":1
            }
      },
      {
         "Order":{
              "Number":"SO43661",
              "Date":"2011-06-01T00:00:00"
         },
         "Account": "Nokia",
         "Item":{
              "Price":24.99,
              "Quantity":3
            }
      }
    ]
}
```

Query with OPENJSON function:

```
SELECT   *
FROM OPENJSON (@json, N'$.Orders')
WITH (
        Number   varchar(200) N'$.Order.Number',
        Date     datetime     N'$.Order.Date',
        Customer varchar(200) N'$.Account',
        Quantity int          N'$.Item.Quantity'
)
```

Output table data:

Number	Date	Customer	Quantity
SO43659	2011-05-31T00:00:00	Microsoft	1
SO43661	2011-06-01T00:00:00	Nokia	3

FOR JSON

Input table data:

Number	Date	Customer	Price	Quantity
SO43659	2011-05-31T00:00:00	MSFT	59.99	1
SO43661	2011-06-01T00:00:00	Nokia	24.99	3

Query with FOR JSON clause:

```
SELECT  Number AS [Order.Number], Date AS [Order.Date],
        Customer AS Account,
        Price AS 'Item.UnitPrice', Quantity AS 'Item.Qty'
FROM SalesOrder
FOR JSON PATH, ROOT('Orders')
```

JSON output:

```
{
  "Orders":
    [
      {
         "Order": {
               "Number":"SO43659",
               "Date":"2011-05-31T00:00:00"
            },
         "Account": "Microsoft",
         "Item": {
               "Price":59.99,
               "Quantity":1
            }
      },
      {
         "Order":{
              "Number":"SO43661",
              "Date":"2011-06-01T00:00:00"
         },
         "Account": "Nokia",
         "Item":{
              "Price":24.99,
              "Quantity":3
            }
      }
    ]
}
```

Table of Contents

SQL Server FOR JSON AUTO Examples (T-SQL)

JULY 8, 2018 / IAN

In SQL Server you can use the `FOR JSON` clause in a query to format the results as JSON. When doing this, you must choose either the `AUTO` or the `PATH` option. This article contains examples of using the `AUTO` option.

Syntax

The syntax goes like this:

```
SELECT ...
  (your query goes here)
FOR JSON AUTO;
```

So basically, all you need to do is add `FOR JSON AUTO` to the end of your query.

Example 1 – Basic Usage

Here's an example to demonstrate.

```
USE Music;
SELECT TOP 3 AlbumName, ReleaseDate
FROM Albums
FOR JSON AUTO;
```

Result:

```
[
    {
        "AlbumName": "Powerslave",
```

```
        "ReleaseDate": "1984-09-03"
    },
    {
        "AlbumName": "Powerage",
        "ReleaseDate": "1978-05-05"
    },
    {
        "AlbumName": "Singing Down the Lane",
        "ReleaseDate": "1956-01-01"
    }
]
```

So the results come out as a nicely formatted JSON document, instead of in rows and columns.

In this case I used `TOP 3` to limit the result set to just three results.

Single Line Results?

Your results may initially appear in a single row and a single column, and as one long line like this:

If this is the case, try clicking the result set. Depending on your database management software, this should launch the JSON document as it appears in the above example.

Whether this works exactly as described will depend on the software that you use to query SQL Server.

At the time of writing, this worked fine for me when using SQL Operations Studio (which has since been renamed to Azure Data Studio). It also worked fine when using the MSSQL extension in VS Code. SSMS however, only formats the results as one long line (although still in JSON format).
I also had varying degrees of success using command line tools.

You may also find that the results are initially distributed across multiple rows, depending on how large the result set is.

If you can't get it to display the results in a satisfactory way, try a different tool.

Example 2 – Query across Multiple Tables

In this example, I query two tables that have a one-to-many relationship between them. In this case, each artist can have many albums.

```
USE Music;
SELECT
  ArtistName,
  AlbumName
FROM Artists
  INNER JOIN Albums
    ON Artists.ArtistId = Albums.ArtistId
ORDER BY ArtistName
FOR JSON AUTO;
```

Result:

```
[
    {
        "ArtistName": "AC/DC",
        "Albums": [
            {
                "AlbumName": "Powerage"
            }
        ]
    },
    {
        "ArtistName": "Allan Holdsworth",
        "Albums": [
```

```
            {
                "AlbumName": "All Night Wrong"
            },
            {
                "AlbumName": "The Sixteen Men of Tain"
            }
        ]
    },
    {
        "ArtistName": "Buddy Rich",
        "Albums": [
            {
                "AlbumName": "Big Swing Face"
            }
        ]
    },
    {
        "ArtistName": "Devin Townsend",
        "Albums": [
            {
                "AlbumName": "Ziltoid the Omniscient"
            },
            {
                "AlbumName": "Casualties of Cool"
            },
            {
                "AlbumName": "Epicloud"
            }
```

```
        ]
    },
    {
        "ArtistName": "Iron Maiden",
        "Albums": [
            {
                "AlbumName": "Powerslave"
            },
            {
                "AlbumName": "Somewhere in Time"
            },
            {
                "AlbumName": "Piece of Mind"
            },
            {
                "AlbumName": "Killers"
            },
            {
                "AlbumName": "No Prayer for the Dying"
            }
        ]
    },
    {
        "ArtistName": "Jim Reeves",
        "Albums": [
            {
                "AlbumName": "Singing Down the Lane"
            }
```

```
        ]
    },
    {
        "ArtistName": "Michael Learns to Rock",
        "Albums": [
            {
                "AlbumName": "Blue Night"
            },
            {
                "AlbumName": "Eternity"
            },
            {
                "AlbumName": "Scandinavia"
            }
        ]
    },
    {
        "ArtistName": "The Script",
        "Albums": [
            {
                "AlbumName": "No Sound Without Silence"
            }
        ]
    },
    {
        "ArtistName": "Tom Jones",
        "Albums": [
            {
```

```
                "AlbumName": "Long Lost Suitcase"
            },
            {
                "AlbumName": "Praise and Blame"
            },
            {
                "AlbumName": "Along Came Jones"
            }
        ]
    }
]
```

As you can see, each album has been nested under “Albums”. This is because the `AUTO` option determines the output based on the order of columns in the `SELECT` list and their source tables.

Example 3 – Add a Root Node

You can use the `ROOT()` option to add a root node to the output. This adds a single, top-level element to the output. To do this, simply add the `ROOT()` option to the end of the query, with the name you want root node to have.

So we can modify the previous example to this:

```
USE Music;
SELECT TOP 3
  ArtistName,
  AlbumName
FROM Artists
  INNER JOIN Albums
```

```
    ON Artists.ArtistId = Albums.ArtistId
ORDER BY ArtistName
FOR JSON AUTO, ROOT('Music');
```

Result:


```
{
    "Music": [
        {
            "ArtistName": "AC/DC",
            "Albums": [
                {
                    "AlbumName": "Powerage"
                }
            ]
        },
        {
            "ArtistName": "Allan Holdsworth",
            "Albums": [
                {
                    "AlbumName": "All Night Wrong"
                },
                {
                    "AlbumName": "The Sixteen Men of Tain"
                }
            ]
        }
    ]
}
```

I also limited the result set to just three results by adding `TOP 3` to the query.

Example 4 – Remove the Array Wrapper

You can use the `WITHOUT_ARRAY_WRAPPER` option to remove the square brackets that surround the results.

Example:

```
USE Music;
SELECT TOP 1
  AlbumName,
  ReleaseDate
FROM Albums
FOR JSON AUTO, WITHOUT_ARRAY_WRAPPER;
```

Result:

```
{
    "AlbumName": "Powerslave",
    "ReleaseDate": "1984-09-03"
}
```

Note that if you do this on a multiple row result, you'll end up with invalid JSON

SQL Server FOR JSON PATH Examples (T-SQL)

JULY 8, 2018 / IAN

When using SQL Server, you can use the `FOR JSON` clause in a query to format the results as JSON. When doing this, you must choose either the `AUTO` or the `PATH` option. This article contains examples of using the `PATH` option.

Syntax

The syntax goes like this:

```
SELECT ...
  (your query goes here)
FOR JSON PATH;
```

So basically, all you need to do is add `FOR JSON PATH` to the end of your query.

Example 1 – Basic Usage

Here's an example to demonstrate.

```
USE Music;
SELECT TOP 3
  AlbumName,
  ReleaseDate
FROM Albums
FOR JSON PATH;
```

Result:

```
[
    {
        "AlbumName": "Powerslave",
        "ReleaseDate": "1984-09-03"
    },
    {
        "AlbumName": "Powerage",
        "ReleaseDate": "1978-05-05"
    },
    {
        "AlbumName": "Singing Down the Lane",
        "ReleaseDate": "1956-01-01"
    }
]
```

So the results come out as a nicely formatted JSON document, instead of in rows and columns.

In this case I used `TOP 3` to limit the result set to just three results.

Single Line Results?

Your results may initially appear in a single row and a single column, and as one long line like this:

```
1   USE Music;
2   SELECT AlbumName, ReleaseDate
3   FROM Albums
4   FOR JSON AUTO;
5
6
7
8
9
```

RESULTS

	JSON_F52E2B61-18A1-11d1-B105-00805F49916B
1	[{"AlbumName":"Powerslave","ReleaseDate":"1984-09-03"},{"AlbumName":"Po...

If this is the case, try clicking the result set. Depending on your database management software, this should launch the JSON document as it appears in the above example.

Whether this works exactly as described will depend on the software that you use to query SQL Server.

At the time of writing, this worked fine for me when using SQL Operations Studio (which has since been renamed to Azure Data Studio). It also worked fine when using the MSSQL extension in VS Code. SSMS however, only formats the results as one long line (although still in JSON format). I also had varying degrees of success using command line tools.

You may also find that the results are initially distributed across multiple rows, depending on how large the result set is.

If you can't get it to display the results in a satisfactory way, try a different tool.

Example 2 – Query across Multiple Tables

In this example, I query two tables that have a one-to-many relationship between them. In this case, each artist can have many albums, and I use a subquery to retrieve the albums for each artist.

```
USE Music;
SELECT TOP 2 ArtistName,
    (SELECT AlbumName
        FROM Albums
        WHERE Artists.ArtistId = Albums.ArtistId
        FOR JSON PATH) AS Albums
FROM Artists
ORDER BY ArtistName
FOR JSON PATH;
```

Result:

```
[
    {
        "ArtistName": "AC/DC",
        "Albums": [
            {
                "AlbumName": "Powerage"
            }
        ]
    },
    {
        "ArtistName": "Allan Holdsworth",
        "Albums": [
```

```
            {
                "AlbumName": "All Night Wrong"
            },
            {
                "AlbumName": "The Sixteen Men of Tain"
            }
        ]
    }
]
```

In this case, each album has been nested under “Albums”. This is because the albums are returned via a subquery.

Example 3 – Nested Output with Dot Notation

When using the `PATH` option, you can use dot-separated column names to create nested objects.

To do this, use a dot-separated alias. Here’s an example:

```
USE Music;
SELECT TOP 3
  AlbumId,
  AlbumName AS 'Details.Album Name',
  ReleaseDate AS 'Details.Release Date'
FROM Albums
FOR JSON PATH;
```

Result:

```
[
    {
        "AlbumId": 1,
```

```
        "Details": {
            "Album Name": "Powerslave",
            "Release Date": "1984-09-03"
        }
    },
    {
        "AlbumId": 2,
        "Details": {
            "Album Name": "Powerage",
            "Release Date": "1978-05-05"
        }
    },
    {
        "AlbumId": 3,
        "Details": {
            "Album Name": "Singing Down the Lane",
            "Release Date": "1956-01-01"
        }
    }
]
```

Example 4 – Add a Root Node

You can use the `ROOT()` option to add a root node to the output. This adds a single, top-level element to the output. To do this, simply add the `ROOT()` option to the end of the query, with the name you want root node to have.

So we can modify the previous example to this:

```
USE Music;
SELECT TOP 3
  AlbumId,
  AlbumName AS 'Details.Album Name',
  ReleaseDate AS 'Details.Release Date'
FROM Albums
FOR JSON PATH, ROOT('Albums');
```

Result:


```
{
    "Albums": [
        {
            "AlbumId": 1,
            "Details": {
                "Album Name": "Powerslave",
                "Release Date": "1984-09-03"
            }
        },
        {
            "AlbumId": 2,
            "Details": {
                "Album Name": "Powerage",
                "Release Date": "1978-05-05"
            }
        },
        {
            "AlbumId": 3,
            "Details": {
```

```
                "Album Name": "Singing Down the Lane",
                "Release Date": "1956-01-01"
            }
        }
    ]
}
```

Example 5 – Remove the Array Wrapper

You can use the `WITHOUT_ARRAY_WRAPPER` option to remove the square brackets that surround the results.

Example:

```
USE Music;
SELECT TOP 1
  AlbumName,
  ReleaseDate
FROM Albums
FOR JSON PATH, WITHOUT_ARRAY_WRAPPER;
```

Result:

```
{
    "AlbumName": "Powerslave",
    "ReleaseDate": "1984-09-03"
}
```

Note that if you do this on a multiple row result, you'll end up with invalid JSON.

ISJSON() Examples in SQL Server (T-SQL)

JULY 8, 2018 / IAN

When using SQL Server, you can use the `ISJSON()` function to test whether or not a string expression contains valid JSON.
If the expression contains valid JSON, `ISJSON()` returns `1`, otherwise it returns `0`.

Syntax

The syntax goes like this:

```
ISJSON ( expression )
```

Where `expression` is the string expression for which you're testing for valid JSON.

Example 1 – Valid JSON

Here's an example to demonstrate what happens when the string contains valid JSON.

```
SELECT ISJSON('{"Name": "Bob"}') AS Result;
```

Result:

```
+----------+
| Result   |
|----------|
| 1        |
+----------+
```

Example 2 – Invalid JSON

Here's an example to demonstrate what happens when the string *doesn't* contain valid JSON.

```
SELECT ISJSON('Name: Bob') AS Result;
```

Result:

```
+----------+
| Result   |
|----------|
| 0        |
+----------+
```

Example 3 – A Conditional Statement

Here's a basic conditional statement that outputs a different result, depending on whether the string contains JSON or not.

```
DECLARE @data nvarchar(255);
SET @data = '{"Name": "Bob"}';
IF (ISJSON(@data) > 0)
    SELECT 'Valid JSON' AS 'Result';
ELSE
    SELECT 'Invalid JSON' AS 'Result';
```

Result:

```
+-------------+
| Result      |
|-------------|
| Valid JSON|
```

```
+------------+
```

Example 4 – A Database Example

In this database query, the results are only returned where the `Collections.Contents` column contains valid JSON.
This particular column uses a data type of `nvarchar(4000)` to store the JSON document.

```
SELECT Contents
FROM Collections
WHERE ISJSON(Contents) > 0;
```

Result:

```
+------------+
| Contents   |
|------------|
| [
    {
        "ArtistName": "AC/DC",
        "Albums": [
            {
                "AlbumName": "Powerage"
            }
        ]
    },
    {
        "ArtistName": "Devin Townsend",
        "Albums": [
```

```
            {
                "AlbumName": "Ziltoid the Omniscient"
            },
            {
                "AlbumName": "Casualties of Cool"
            },
            {
                "AlbumName": "Epicloud"
            }
        ]
    },
    {
        "ArtistName": "Iron Maiden",
        "Albums": [
            {
                "AlbumName": "Powerslave"
            },
            {
                "AlbumName": "Somewhere in Time"
            },
            {
                "AlbumName": "Piece of Mind"
            },
            {
                "AlbumName": "Killers"
            },
            {
                "AlbumName": "No Prayer for the Dying"
```

```
                }
            ]
        }
]                |
+--------------+
```

JSON_VALUE() Examples in SQL Server (T-SQL)

JULY 9, 2018 / IAN

When using JSON with SQL Server, you can use the `JSON_VALUE()` function to return a scalar value from a JSON string.
To use this function, you provide two arguments; the JSON expression, and the property to extract.

Syntax

The syntax goes like this:

```
JSON_VALUE ( expression , path )
```

Where `expression` is the JSON string expression, and `path` is the property you want to extract from that expression.
The path argument can include an optional *path mode* component. This optional path mode can be a value of either `lax` or `strict`. This value, if any, comes before the dollar sign.

Example 1 – Basic Usage

Here's an example to demonstrate basic usage of the `JSON_VALUE()` function.

```
SELECT JSON_VALUE('{"Name": "Bruce"}', '$.Name') AS 'Result';
```

Result:

```
+----------+
| Result   |
```

```
|----------|
| Bruce    |
+----------+
```

In this example:

- The `{"Name": "Bruce"}` argument is the JSON expression (a small one, but still a valid JSON expression). JSON expressions consist of a key/value pair. In this case, `Name` is the key, `Bruce` is its value.
- The `$.Name` argument is the path. This path references the value of the `Name` key of the JSON expression. So we can extract the value by referencing the name of the pair.

Example 2 – Arrays

To extract a value from an array, reference its index within square brackets, followed by the relevant key. Here's an example:

```
/*
CREATE THE ARRAY (and put into a variable called @data)
*/
DECLARE @data NVARCHAR(4000)
SET @data=N'{
    "Cities": [
        {
            "Name": "Kabul",
            "CountryCode": "AFG",
            "District": "Kabol",
            "Population": 1780000
        },
```

```
        {
            "Name": "Qandahar",
            "CountryCode": "AFG",
            "District": "Qandahar",
            "Population": 237500
        }
    ]
}'

/* QUERY THE ARRAY */
SELECT
  JSON_VALUE(@data,'$.Cities[0].Name') AS 'Name',
  JSON_VALUE(@data,'$.Cities[0].CountryCode') AS 'Country Code',
  JSON_VALUE(@data,'$.Cities[0].District') AS 'District',
  JSON_VALUE(@data,'$.Cities[0].Population') AS 'Population'
UNION ALL
SELECT
  JSON_VALUE(@data,'$.Cities[1].Name') AS 'Name',
  JSON_VALUE(@data,'$.Cities[1].CountryCode') AS 'Country Code',
  JSON_VALUE(@data,'$.Cities[1].District') AS 'District',
  JSON_VALUE(@data,'$.Cities[1].Population') AS 'Population';
```

Result:

```
+----------+----------------+------------+--------------+
| Name     | Country Code   | District   | Population   |
|----------+----------------+------------+--------------|
| Kabul    | AFG            | Kabol      | 1780000      |
| Qandahar | AFG            | Qandahar   | 237500       |
+----------+----------------+------------+--------------+
```

So in this example, we create a JSON array and put it into a variable called `@data`. We then run a query, using `@data` as the first argument of the `JSON_VALUE()` function (this is because `@data` contains the JSON expression).

Arrays use zero-based numbering, so to extract the first item we need to use `Cities[0]`, the second one `Cities[1]`, and so on.

Example 3 – A Database Example

If we were to put the data from the previous example into a database, we could rewrite the query as follows:

```
SELECT
  JSON_VALUE(Document,'$.Cities[0].Name') AS 'Name',
  JSON_VALUE(Document,'$.Cities[0].CountryCode') AS 'Country
Code',
  JSON_VALUE(Document,'$.Cities[0].District') AS 'District',
  JSON_VALUE(Document,'$.Cities[0].Population') AS 'Population'
FROM Json_Documents

UNION ALL

SELECT
  JSON_VALUE(Document,'$.Cities[1].Name') AS 'Name',
  JSON_VALUE(Document,'$.Cities[1].CountryCode') AS 'Country
Code',
  JSON_VALUE(Document,'$.Cities[1].District') AS 'District',
  JSON_VALUE(Document,'$.Cities[1].Population') AS 'Population'
FROM Json_Documents
```

Result:

```
+----------+----------------+------------+--------------+
| Name     | Country Code   | District   | Population   |
|----------+----------------+------------+--------------|
| Kabul    | AFG            | Kabol      | 1780000      |
| Qandahar | AFG            | Qandahar   | 237500       |
+----------+----------------+------------+--------------+
```

This assumes that the JSON document is stored in a column called `Document`, which is in a table called `Json_Documents`.

Example 4 – Path Mode

As mentioned, you also have the option of specifying the path mode. This can be either `lax` or `strict`.

The value of the path mode determines what happens when the path expression contains an error. Specifically:

- In **lax** mode, the function returns empty values if the path expression contains an error. For example, if you request the value **$.name**, and the JSON text doesn't contain a **name** key, the function returns null, but does not raise an error.
- In **strict** mode, the function raises an error if the path expression contains an error.

The default value is `lax`.

Here's an example to demonstrate the difference between these two modes.

Error in lax mode

Here's what happens when the path expression contains an error while in lax mode.

```
SELECT JSON_VALUE('{"Name": "Bruce"}', 'lax $.Hobbies') AS
'Result';
```

Result:

```
+----------+
| Result   |
|----------|
| NULL     |
+----------+
```

In this example we're trying to reference `Hobbies`, but that key doesn't exist in the JSON document. In this case we get a null value (because we're using lax mode).

Error in strict mode

Here's what happens when we run the same code in strict mode.

```
SELECT JSON_VALUE('{"Name": "Bruce"}', 'strict $.Hobbies') AS
'Result';
```

Result:

```
Msg 13608, Level 16, State 1, Line 1
Property cannot be found on the specified JSON path.
```

As expected, strict mode results in an error message being displayed.

Example 5 – Returning Objects and Arrays

The `JSON_VALUE()` function doesn't return objects and arrays. If you want to return an object or an array, use the `JSON_QUERY()` function instead. Here's an example where I use both functions within a query.

```
DECLARE @data NVARCHAR(4000)
SET @data=N'{
    "Suspect": {
       "Name": "Homer Simpson",
       "Address": {
         "City": "Mae Sai",
         "Province": "Chiang Rai",
         "Country": "Thailand"
       },
       "Hobbies": ["Eating", "Sleeping", "Base Jumping"]
    }
 }'
 SELECT
   JSON_VALUE(@data,'$.Suspect.Name') AS 'Name',
   JSON_VALUE(@data,'$.Suspect.Address.Country') AS 'Country',
   JSON_QUERY(@data,'$.Suspect.Hobbies') AS 'Hobbies',
   JSON_VALUE(@data,'$.Suspect.Hobbies[2]') AS 'Last Hobby';
```

Result:

```
+---------------+-----------+-----------------------------------
----+--------------+
| Name          | Country   | Hobbies
| Last Hobby    |
```

```
|---------------+-----------+-------------------------------------
----+--------------|
| Homer Simpson | Thailand  | ["Eating", "Sleeping", "Base
Jumping"] | Base Jumping |
+---------------+-----------+-------------------------------------
----+--------------+
```

In this case, I use `JSON_VALUE()` to return various scalar values, and `JSON_QUERY()` to return an array.

So if you need to return an object or an array (including the whole JSON document), see `JSON_QUERY()` Examples in SQL Server.

JSON_QUERY() Examples in SQL Server (T-SQL)

JULY 9, 2018 / IAN

When using JSON with SQL Server, you can use the `JSON_QUERY()` function to extract an object or an array from a JSON string.
To use this function, you provide the JSON expression as an argument. You can also provide a second (optional) argument to specify the object or array to extract.

Syntax

The syntax goes like this:

```
JSON_QUERY ( expression [ , path ] )
```

Where `expression` is the JSON string expression, and `path` is the object or array that you want to extract from that expression. The `path` argument is optional (if you don't provide it, the whole JSON document is returned).
The path argument (if supplied) can include an optional *path mode* component. This optional path mode can be a value of either `lax` or `strict`. This value determines what happens in the event the supplied path is invalid. The path mode (if supplied) comes before the dollar sign.

Example 1 – Basic Usage

Here's an example to demonstrate basic usage of the `JSON_QUERY()` function.

```
DECLARE @data NVARCHAR(4000)
SET @data=N'{
    "Cities": [
        {
            "Name": "Kabul",
            "CountryCode": "AFG",
            "District": "Kabol",
            "Population": 1780000
        },
        {
            "Name": "Qandahar",
            "CountryCode": "AFG",
            "District": "Qandahar",
            "Population": 237500
        }
    ]
}'
SELECT JSON_QUERY(@data, '$.Cities[0]') AS 'Result';
```

Result:

```
      {
            "Name": "Kabul",
            "CountryCode": "AFG",
            "District": "Kabol",
            "Population": 1780000
      }
```

In this example, I first declare and set a variable called `@data`. I then assign an array to this variable. Once I've done this, I run a query against that array.

In this case I use `Cities[0]` to reference the first item in the array (JSON arrays use zero-based numbering).

I could access the second item by using `Cities[1]`. Like this:

```
DECLARE @data NVARCHAR(4000)
SET @data=N'{
    "Cities": [
        {
            "Name": "Kabul",
            "CountryCode": "AFG",
            "District": "Kabol",
            "Population": 1780000
        },
        {
            "Name": "Qandahar",
            "CountryCode": "AFG",
            "District": "Qandahar",
            "Population": 237500
        }
    ]
}'
SELECT JSON_QUERY(@data, '$.Cities[1]') AS 'Result';
```

Result:

```
        {
            "Name": "Qandahar",
            "CountryCode": "AFG",
            "District": "Qandahar",
            "Population": 237500
        }
```

Example 2 – Return the Whole JSON Expression

The second argument is optional, so if you omit it, the whole JSON document is returned. Here's what happens when we do that using the same data from the previous examples:

```
DECLARE @data NVARCHAR(4000)
SET @data=N'{
    "Cities": [
        {
            "Name": "Kabul",
            "CountryCode": "AFG",
            "District": "Kabol",
            "Population": 1780000
        },
        {
            "Name": "Qandahar",
            "CountryCode": "AFG",
            "District": "Qandahar",
            "Population": 237500
        }
```

```
    ]
}'
SELECT JSON_QUERY(@data) AS 'Result';
```

Result:

```
{
    "Cities": [
        {
            "Name": "Kabul",
            "CountryCode": "AFG",
            "District": "Kabol",
            "Population": 1780000
        },
        {
            "Name": "Qandahar",
            "CountryCode": "AFG",
            "District": "Qandahar",
            "Population": 237500
        }
    ]
}
```

Example 3 – A Database Example

If we were to put the data from the previous example into a database, we could rewrite the query as follows:

```
SELECT
  JSON_QUERY(Document,'$.Cities[0]') AS 'City 1'
FROM Json_Documents
```

Result:

```
      {
            "ID": 1,
            "Name": "Kabul",
            "CountryCode": "AFG",
            "District": "Kabol",
            "Population": 1780000
        }
```

This assumes that the JSON document is stored in a column called `Document`, which is in a table called `Json_Documents`.

Example 4 – Scalar Values

The `JSON_QUERY()` function is not designed to return scalar values. If you want to return a scalar value, use the `JSON_VALUE()` function instead. However, there's nothing to stop you combining both functions within a query to return data at various levels of granularity.

Here's an example:

```
DECLARE @data NVARCHAR(4000)
SET @data=N'{
    "Suspect": {
       "Name": "Homer Simpson",
       "Hobbies": ["Eating", "Sleeping", "Base Jumping"]
    }
 }'
 SELECT
   JSON_VALUE(@data,'$.Suspect.Name') AS 'Name',
```

```
   JSON_QUERY(@data,'$.Suspect.Hobbies') AS 'Hobbies',
   JSON_VALUE(@data,'$.Suspect.Hobbies[2]') AS 'Last Hobby';
```

Result:

```
+---------------+-----------------------------------------+-------
-------+
| Name          | Hobbies                                 | Last
Hobby   |
|---------------+-----------------------------------------+-------
-------|
| Homer Simpson | ["Eating", "Sleeping", "Base Jumping"] | Base
Jumping |
+---------------+-----------------------------------------+-------
-------+
```

In this example I used `JSON_VALUE()` to extract various scalar values, but I also used `JSON_QUERY()` to return a whole array (which `JSON_VALUE()` can't do).

Example 5 – Path Mode

As mentioned, you also have the option of specifying the path mode. This can be either `lax` or `strict`.

The value of the path mode determines what happens when the path expression contains an error. Specifically:

- In **lax** mode, the function returns empty values if the path expression contains an error. For example, if you request the value **$.name**, and the JSON text doesn't contain a **name** key, the function returns null, but does not raise an error.

- In **strict** mode, the function raises an error if the path expression contains an error.

The default value is `lax`.
Here's an example to demonstrate the difference between these two modes.

Error in lax mode

Here's what happens when the path expression contains an error while in lax mode.

```
SELECT JSON_QUERY('{"Name": "Bruce"}', 'lax $.Name') AS 'Result';
```

Result:

```
+----------+
| Result   |
|----------|
| NULL     |
+----------+
```

In this example we're trying to return a scalar value,
but `JSON_QUERY()` doesn't do scalar values. As mentioned, it only returns objects and arrays. In this case we get a null value (because we're using lax mode).

Error in strict mode

Here's what happens when we run the same code in strict mode.

```
SELECT JSON_QUERY('{"Name": "Bruce"}', 'strict $.Name') AS
'Result';
```

Result:

```
Msg 13624, Level 16, State 2, Line 1
Object or array cannot be found in the specified JSON path.
```

As expected, strict mode results in an error message explaining the error.

JSON_QUERY() vs JSON_VALUE() in SQL Server: What's the Difference?

JULY 9, 2018 / IAN

Two of the many T-SQL functions available in SQL Server are `JSON_QUERY()` and `JSON_VALUE()`. These functions can be used to extract data from JSON documents.
Their general syntax is similar, and at first glance, you might think they do exactly the same thing, but they don't. There's definitely a place for both functions when working with JSON and SQL Server.

This article looks at the difference between `JSON_QUERY()` and `JSON_VALUE()`.

The Difference

These two functions have slightly different definitions, a slightly different syntax, and their return values are slightly different.

Definitions

Here's how the two functions are defined:

`JSON_QUERY()`

> Extracts an object or an array from a JSON string.

`JSON_VALUE()`

> Extracts a scalar value from a JSON string.

So the difference between these two functions is what they extract. One extracts an object or an array, the other extracts a scalar value.

Syntax Differences

Another difference is in the syntax:

```
JSON_QUERY ( expression [ , path ] )
JSON_VALUE ( expression , path )
```

Look at the `JSON_QUERY()` syntax. Those square brackets around the `path` argument mean that it's an optional argument. That's because this function can return a whole JSON document if required.
However, the path argument is a required argument when using the `JSON_VALUE()` function. So you must provide both arguments when using this function.

Return Values

And one more difference is in their return values.

- `JSON_QUERY()` returns a JSON fragment of type `nvarchar(max)`
- `JSON_VALUE()` returns a single text value of type `nvarchar(4000)`

Example 1 – Extract a Scalar Value

Here's an example to demonstrate the difference between these functions when trying to extract a scalar value.

```
SELECT
  JSON_VALUE('{"Name": "Homer"}', '$.Name') AS 'JSON_VALUE',
  JSON_QUERY('{"Name": "Homer"}', '$.Name') AS 'JSON_QUERY';
```

Result:

```
+--------------+--------------+
| JSON_VALUE   | JSON_QUERY   |
|--------------+--------------|
| Homer        | NULL         |
+--------------+--------------+
```

So both functions are trying to extract the same value from the JSON document, but only one succeeds: `JSON_VALUE()`. This is because the value they're trying to extract is a scalar value. Basically, a *scalar value* is one unit of data. It could be a string of text or a number. But it can't be an object or an array.

Example 2 – Extract an Array

In this example, both functions try to extract a whole array.

```
DECLARE @data NVARCHAR(4000)
SET @data=N'{
    "Suspect": {
       "Name": "Homer Simpson",
       "Hobbies": ["Eating", "Sleeping", "Base Jumping"]
    }
```

```
 }'
 SELECT
   JSON_VALUE(@data,'$.Suspect.Hobbies') AS 'JSON_VALUE',
   JSON_QUERY(@data,'$.Suspect.Hobbies') AS 'JSON_QUERY';
```

Result:

```
+--------------+----------------------------------------+
| JSON_VALUE   | JSON_QUERY                             |
|--------------+----------------------------------------|
| NULL         | ["Eating", "Sleeping", "Base Jumping"] |
+--------------+----------------------------------------+
```

In this case, only the `JSON_QUERY()` function succeeds.

Example 3 – Extract an Array Item

This example is similar to the previous one, except that instead of trying to extract the whole array, we only want a single item from the array.

```
DECLARE @data NVARCHAR(4000)
SET @data=N'{
    "Suspect": {
       "Name": "Homer Simpson",
       "Hobbies": ["Eating", "Sleeping", "Base Jumping"]
    }
 }'
 SELECT
   JSON_VALUE(@data,'$.Suspect.Hobbies[2]') AS 'JSON_VALUE',
   JSON_QUERY(@data,'$.Suspect.Hobbies[2]') AS 'JSON_QUERY';
```

Result:

```
+--------------+--------------+
| JSON_VALUE   | JSON_QUERY   |
|--------------+--------------|
| Base Jumping | NULL         |
+--------------+--------------+
```

So this time `JSON_VALUE()` is the winner.

Example 4 – Extract an Object

Let's try for a whole object.

```
DECLARE @data NVARCHAR(4000)
SET @data=N'{
    "Suspect": {
       "Name": "Homer Simpson",
       "Hobbies": ["Eating", "Sleeping", "Base Jumping"]
    }
 }'
 SELECT
   JSON_VALUE(@data,'$.Suspect') AS 'JSON_VALUE',
   JSON_QUERY(@data,'$.Suspect') AS 'JSON_QUERY';
```

Result:

```
+--------------+--------------+
| JSON_VALUE   | JSON_QUERY   |
|--------------+--------------|
| NULL         | {
                  "Name": "Homer Simpson",
                  "Hobbies": ["Eating", "Sleeping", "Base Jumping"]
                  }            |
+--------------+--------------+
```

And `JSON_QUERY()` wins.

(Excuse the formatting, this is how my MSSQL command line tool returns the results).

Example 5 – Extract the Whole JSON Document

Let's try for the whole JSON document.

```
DECLARE @data NVARCHAR(4000)
SET @data=N'{
    "Cities": [
        {
            "Name": "Kabul",
            "CountryCode": "AFG",
            "District": "Kabol",
            "Population": 1780000
        },
        {
            "Name": "Qandahar",
            "CountryCode": "AFG",
            "District": "Qandahar",
            "Population": 237500
        }
    ]
}'
SELECT
  JSON_VALUE(@data, '$') AS 'JSON_VALUE',
  JSON_QUERY(@data, '$') AS 'JSON_QUERY';
```

Result:

```
+--------------+--------------+
| JSON_VALUE   | JSON_QUERY   |
|--------------+--------------|
| NULL         | {
                        "Cities": [
                        {
                                "Name": "Kabul",
                                "CountryCode": "AFG",
                                "District": "Kabol",
                                "Population": 1780000
                        },
                        {
                                "Name": "Qandahar",
                                "CountryCode": "AFG",
                                "District": "Qandahar",
                                "Population": 237500
                        }
                        ]
}               |
+--------------+--------------+
```

So `JSON_QUERY()` is the only one that can return the whole document.

Example 6 – Omit the Path

Another difference between these two functions is that, the path argument is optional when using `JSON_QUERY()`. If you omit this, the whole JSON document is returned.

You can't omit this argument when using `JSON_VALUE()`, as it is a required argument. This is probably due to the fact that the function can only return a scalar value. If the first argument only consisted of a scalar value, it wouldn't be valid JSON.

Anyway, here's an example of omitting the path argument from `JSON_QUERY()`:

```
SELECT JSON_QUERY('{"Name": "Homer"}') AS 'Result';
```

Result:

```
+--------------------+
| Result             |
|--------------------|
| {"Name": "Homer"}  |
+--------------------+
```

And here's what happens if we try that trick with `JSON_VALUE()`:

```
SELECT JSON_VALUE('{"Name": "Homer"}') AS 'Result';
```

Result:

```
Msg 174, Level 15, State 1, Line 1
The json_value function requires 2 argument(s).
```

Example 7 – Path Mode

In the earlier examples, when a function couldn't handle the supplied path, it returned `NULL`. This is because all of those examples were run in lax mode (the default mode).

If we'd run them in strict mode, we would've received an error instead. To explicitly specify the path mode, simply add it before the dollar sign (and leave a space between them).

Here's an example of what happens when you provide an invalid path while in strict mode:

```
SELECT
  JSON_VALUE('{"Name": "Homer"}', 'strict $.Name') AS
'JSON_VALUE',
  JSON_QUERY('{"Name": "Homer"}', 'strict $.Name') AS
'JSON_QUERY';
```

Result:

```
Msg 13624, Level 16, State 2, Line 1
Object or array cannot be found in the specified JSON path.
```

JSON_MODIFY() Examples in SQL Server

JULY 10, 2018 / IAN

In SQL Server, you can use the T-SQL `JSON_MODIFY()` function to modify the value of a property in a JSON string. The function returns the updated JSON string.

Syntax

The syntax goes like this:

```
JSON_MODIFY ( expression , path , newValue )
```

Where `expression` is the JSON string expression, `path` is the path to the property you want to update, and `newValue` is the new value to apply to that property.

Example 1 – Basic Usage

Here's an example to demonstrate.

```
SELECT JSON_MODIFY('{"Name": "Homer"}', '$.Name', 'Bart') AS
'Result';
```

Result:

```
+------------------+
| Result           |
|------------------|
| {"Name": "Bart"} |
+------------------+
```

In this example:

- `{"Name": "Homer"}` is the original JSON string
- `$.Name` is the path (this begins with `$.` followed by the path to the property we want to update).
- `Bart` is the new value we want to assign to `Name` (i.e. to replace the current value)

Example 2 – Return the Original and Modified JSON

Note that `JSON_MODIFY()` doesn't modify the original JSON. It takes a copy, then modifies and returns the copy.

Here's an example to demonstrate this:

```
DECLARE @suspect NVARCHAR(4000)
SET @suspect= '{"Name": "Homer"}'
SELECT
  @suspect AS 'Original String',
  JSON_MODIFY(@suspect, '$.Name', 'Bart') AS 'Modified String',
  @suspect AS 'Original String';
```

Result:

```
+-------------------+-------------------+-------------------+
| Original String   | Modified String   | Original String   |
|-------------------+-------------------+-------------------|
| {"Name": "Homer"} | {"Name": "Bart"}  | {"Name": "Homer"} |
+-------------------+-------------------+-------------------+
```

Example 3 – Nested Properties

The path can use dot-notation to reference nested properties. Here's an example.

```
DECLARE @data NVARCHAR(4000)
SET @data=N'{
    "Suspect": {
       "Name": "Homer Simpson",
       "Address": {
         "City": "Dunedin",
         "Region": "Otago",
         "Country": "New Zealand"
       },
       "Hobbies": ["Eating", "Sleeping", "Base Jumping"]
    }
 }'
 SELECT
   JSON_MODIFY(@data,'$.Suspect.Address.City', 'Timaru') AS
'Modified Array';
```

Result:

```
+------------------+
| Modified Array   |
|------------------|
{
    "Suspect": {
       "Name": "Homer Simpson",
       "Address": {
         "City": "Timaru",
         "Region": "Otago",
```

```
          "Country": "New Zealand"
       },
       "Hobbies": ["Eating", "Sleeping", "Base Jumping"]
    }
}
```

So we can see that the city has been changed from `Dunedin` to `Timaru`.

Example 4 – Update Values in an Array

You can also update values within an array. In this example, we update a value in the `Hobbies` array.

```
DECLARE @data NVARCHAR(4000)
SET @data=N'{
    "Suspect": {
       "Name": "Homer Simpson",
       "Address": {
         "City": "Dunedin",
         "Region": "Otago",
         "Country": "New Zealand"
       },
       "Hobbies": ["Eating", "Sleeping", "Base Jumping"]
    }
 }'
 SELECT
   JSON_MODIFY(@data,'$.Suspect.Hobbies[2]', 'Brain Surgery') AS
'Updated Hobbies';
```

Result:

```
+--------------------+
| Updated Hobbies    |
|--------------------|
| {
    "Suspect": {
       "Name": "Homer Simpson",
       "Address": {
         "City": "Dunedin",
         "Region": "Otago",
         "Country": "New Zealand"
       },
       "Hobbies": ["Eating", "Sleeping", "Brain Surgery"]
    }
 }                    |
+--------------------+
```

Seeing as arrays use zero-based numbering, we update the third item by referencing `Hobbies[2]`.

Example 5 – Append a Value to an Array

In this example, we append a value to the `Hobbies` array. We do this by adding `append` at the start of the path argument.

```
DECLARE @data NVARCHAR(4000)
SET @data=N'{
    "Suspect": {
       "Name": "Homer Simpson",
       "Address": {
```

```
          "City": "Dunedin",
          "Region": "Otago",
          "Country": "New Zealand"
       },
       "Hobbies": ["Eating", "Sleeping", "Base Jumping"]
    }
 }'
 SELECT
   JSON_MODIFY(@data,'append $.Suspect.Hobbies', 'Brain Surgery')
AS 'Updated Hobbies';
```

Result:

```
+--------------------+
| Updated Hobbies    |
|--------------------|
| {
    "Suspect": {
       "Name": "Homer Simpson",
       "Address": {
          "City": "Dunedin",
          "Region": "Otago",
          "Country": "New Zealand"
       },
       "Hobbies": ["Eating", "Sleeping", "Base Jumping","Brain
Surgery"]
    }
 }                   |
+--------------------+
```

Example 6 – Update a Whole Array

In this example, I update the whole array.

```
DECLARE @data NVARCHAR(4000)
SET @data=N'{
    "Suspect": {
       "Name": "Homer Simpson",
       "Address": {
         "City": "Dunedin",
         "Region": "Otago",
         "Country": "New Zealand"
       },
       "Hobbies": ["Eating", "Sleeping", "Base Jumping"]
    }
 }'
 SELECT
   JSON_MODIFY(@data,'$.Suspect.Hobbies', JSON_QUERY('["Chess",
"Brain Surgery"]')) AS 'Updated Hobbies';
```

Result:

```
+--------------------+
| Updated Hobbies    |
|--------------------|
| {
    "Suspect": {
       "Name": "Homer Simpson",
       "Address": {
         "City": "Dunedin",
         "Region": "Otago",
```

```
          "Country": "New Zealand"
       },
       "Hobbies": ["Chess", "Brain Surgery"]
    }
 }                    |
+--------------------+
```

Note that in this example, the third argument is passed to the `JSON_QUERY()` function. If I hadn't done this, SQL Server would have escaped the double quotes and square brackets using the backslash (`\`) character (and therefore messing up the array). It would've done this because it wouldn't have known whether the updated value was an actual array, or a string literal.

So to get around this, we can use `JSON_QUERY()`. This function returns valid JSON, and SQL Server will then assume that the new value is an array.

Here's what would've happened if we *hadn't* used `JSON_QUERY()`:

```
DECLARE @data NVARCHAR(4000)
SET @data=N'{
    "Suspect": {
       "Name": "Homer Simpson",
       "Address": {
          "City": "Dunedin",
          "Region": "Otago",
          "Country": "New Zealand"
       },
       "Hobbies": ["Eating", "Sleeping", "Base Jumping"]
    }
```

```
 }'
 SELECT
   JSON_MODIFY(@data,'$.Suspect.Hobbies', '["Chess", "Brain
Surgery"]') AS 'Updated Hobbies';
```

Result:

```
+--------------------+
| Updated Hobbies    |
|--------------------|
| {
    "Suspect": {
       "Name": "Homer Simpson",
       "Address": {
         "City": "Dunedin",
         "Region": "Otago",
         "Country": "New Zealand"
       },
       "Hobbies": "[\"Chess\", \"Brain Surgery\"]"
    }
 }                    |
+--------------------+
```

So SQL Server has escaped the square brackets and double quotes.

Example 7 – Update a Whole Object

Here's an example of updating a whole object.

```
DECLARE @data NVARCHAR(4000)
SET @data=N'{
    "Suspect": {
```

```
       "Name": "Homer Simpson",
       "Hobbies": ["Eating", "Sleeping", "Base Jumping"]
    }
 }'
 SELECT
   JSON_MODIFY(@data,'$.Suspect', JSON_QUERY('{"Name": "Peter
Griffin", "Hobbies": "None"}')) AS 'Updated Object';
```

Result:

```
+------------------+
| Updated Object   |
|------------------|
 {
    "Suspect": {"Name": "Peter Griffin", "Hobbies": "None"}
 }
+------------------+
```

Again, if we hadn't used `JSON_QUERY()`, we would've received an escaped string:

```
DECLARE @data NVARCHAR(4000)
SET @data=N'{
    "Suspect": {
       "Name": "Homer Simpson",
       "Hobbies": ["Eating", "Sleeping", "Base Jumping"]
    }
 }'
 SELECT
   JSON_MODIFY(@data,'$.Suspect', '{"Name": "Peter Griffin",
"Hobbies": "None"}') AS 'Updated Object';
```

Result:

```
+------------------+
| Updated Object   |
|------------------|
 {
    "Suspect": "{\"Name\": \"Peter Griffin\", \"Hobbies\":
\"None\"}"
 }
+------------------+
```

Example 8 – Rename a Key

You're not just limited to updating a property's value, you can also rename its key. Here's an example.

```
DECLARE @data NVARCHAR(50)='{"Name":"Homer"}'
PRINT @data
-- Rename the key
SET @data=
 JSON_MODIFY(
  JSON_MODIFY(@data,'$.Handle', JSON_VALUE(@data,'$.Name')),
  '$.Name',
  NULL
 )
PRINT @data
```

Result:

```
{"Name":"Homer"}
{"Handle":"Homer"}
```

Here, we take the value from the existing property and assign it to a new key/value pair. We then set the value of the original key to `NULL` (which automatically deletes it).

For more examples of renaming a key, see How to Rename a JSON Key in SQL Server.

How to Rename a JSON Key in SQL Server (T-SQL)

JULY 10, 2018 / IAN

If you've been using the `JSON_MODIFY()` function to modify JSON documents in SQL Server, you might be used to modifying the *value* part of a *key/value* property. But did you know that you can also modify the *key* part?

The trick to doing this is to copy the value to a new key, then delete the old key.

Examples below.

Basic Example

Here's a basic example to show what I mean.

```
-- Declare a variable and assign some JSON to it
DECLARE @data NVARCHAR(50)='{"Name":"Homer"}'

-- Print the current JSON
PRINT @data

-- Rename the key (by copying the value to a new key, then
deleting the old one)
SET @data=
 JSON_MODIFY(
  JSON_MODIFY(@data,'$.Handle', JSON_VALUE(@data,'$.Name')),
  '$.Name',
```

```
  NULL
 )
-- Print the new JSON
PRINT @data
```

Result:

```
{"Name":"Homer"}
{"Handle":"Homer"}
```

This prints out the original key/value pair, followed by the new key/value pair.

Although we can say that we "renamed" the key, we actually just created a new key, copied the existing value to that new key, then deleted the old key by setting it to `NULL`.
In this case, we used the `JSON_VALUE()` function to extract the value.

Numeric Values

You need to be careful when copying the data to the new key. By default, SQL Server will enclose it in double quotes. This may or may not be what you want.

However, if you're copying a numeric value, chances are you want it to remain a numeric value (i.e. without double quotes). In this case you'll need to use the `CAST()` function to cast it as a numeric data type. Here's an example:

```
-- Declare a variable and assign some JSON to it
DECLARE @data NVARCHAR(50)='{"Residents":768}'
```

```
-- Print the current JSON
PRINT @data

-- Rename the key (by copying the value to a new key, then
deleting the old one)
SET @data=
 JSON_MODIFY(
  JSON_MODIFY(@data,'$.Population',
CAST(JSON_VALUE(@data,'$.Residents') AS int)),
  '$.Residents',
  NULL
 )
-- Print the new JSON
PRINT @data
```

Result:

```
{"Residents":768}
{"Population":768}
```

So the resulting value is a number.

If we remove the `CAST()` function from that example, we end up with this:

```
-- Declare a variable and assign some JSON to it
DECLARE @data NVARCHAR(50)='{"Residents": 768}'

-- Print the current JSON
PRINT @data
```

```
-- Rename the key (by copying the value to a new key, then deleting the old one)

SET @data=

 JSON_MODIFY(

  JSON_MODIFY(@data,'$.Population', JSON_VALUE(@data,'$.Residents')),

  '$.Residents',

  NULL

 )

-- Print the new JSON

PRINT @data
```

Result:

```
{"Residents": 768}

{"Population":"768"}
```

So in this case, we didn't just rename the key, we also changed the (JSON) data type from a number to a string.

Note that JSON doesn't distinguish between different numeric types. It has only one numeric type: number.

Keys with Spaces

In this example, I rename an existing key to a new key that contains a space (it consists of two words, separated by a space).

Because the new key contains a space, I need to surround the key with double quotes. If I don't do this an error will occur.

```
-- Declare a variable and assign some JSON to it

DECLARE @data NVARCHAR(50)='{"Population":68}'



-- Print the current JSON

PRINT @data



-- Rename the key (by copying the value to a new key, then deleting the
old one)

SET @data=

 JSON_MODIFY(

  JSON_MODIFY(@data,'$."Average IQ"',
CAST(JSON_VALUE(@data,'$.Population') AS int)),

  '$.Population',

  NULL

 )

-- Print the new JSON

PRINT @data
```

Result:

```
{"Population":68}

{"Average IQ":68}
```

Nested Properties

If the property is nested, no problem. Simply use dot-notation to reference it.

```
DECLARE @data NVARCHAR(4000)

SET @data=N'{

    "Suspect": {

       "Name": "Homer Simpson",

       "Hobbies": ["Eating", "Sleeping", "Base Jumping"]

    }

 }'

PRINT @data

SET @data=

  JSON_MODIFY(

    JSON_MODIFY(@data,'$.Suspect.Qualifications',
JSON_QUERY(@data,'$.Suspect.Hobbies')),

   '$.Suspect.Hobbies',

   NULL

  )

PRINT @data
```

Result:

```
       {

"Suspect": {
```

```
"Name": "Homer Simpson",

"Hobbies": ["Eating", "Sleeping", "Base Jumping"]

}

}

{

"Suspect": {

"Name": "Homer Simpson"

,"Qualifications":["Eating", "Sleeping", "Base Jumping"]}

}
```

You might also have noticed that this example uses the `JSON_QUERY()` function to extract the value, instead of `JSON_VALUE()` like in the previous examples.
This is because in this case we're extracting an array and `JSON_VALUE()` can't extract a whole array (it can only extract a scalar value from the array). The `JSON_QUERY()` function, on the other hand, extracts objects and arrays, but not scalar values.

To read more about this, see `JSON_QUERY()` vs `JSON_VALUE()`: What's the Difference?