**Native JSON support – new feature in SQL Server 2016**

**–> JSON** (or JavaScript Object Notation) is a popular, language-independent, lightweight data-interchange format used in modern web and mobile applications, as well for **storing Unstructured data**. JSON is an alternate to XML and is **more compact** than that format, and thus has become the first choice for many applications that need to move data around on the web. One of the biggest reasons JSON is becoming more important than XML is that XML has to be parsed with an XML parser, while JSON can be parsed by a standard JavaScript function. This makes it easier and faster than working with XML.

*For more details on JSON check at* [***json.org***](http://json.org/)

**–> JSON in SQL Server 2016** is not treated as a separate Datatype by the DB Engine, like XML. For example, appending **FOR JSON AUTO** to a standard SELECT statement returns the result set in a JSON format. The JSON data is **stored as NVARCHAR type**, unlike XML datatype for XML data, thus JSON will be supported wherever NVARCHAR is supported.

**–> Here is a sample & simple query to convert a row-set into a JSON format:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | SELECT TOP 10      M.ProductModelID,      M.Name AS [ProductModel.Name],      ProductID,      P.Name AS [Product.Name],      ProductNumber,      MakeFlag  FROM Production.Product P  INNER JOIN Production.ProductModel M  ON P.ProductModelID = M.ProductModelID  FOR JSON PATH, ROOT('ProductModel') -- here, JSON syntax similar to XML |

Check the last line, the syntax is almost similar to XML datatype. So, if you familiar with XML syntax in T-SQL, working with JSON will be a seamless experience.

## Export/Convert Table or SQL Query data to JSON string format | SQL Server 2016 – Part 1

**–>** Just like XML now you can also:

1. Store JSON data in SQL Server in table columns as NVARCHAR datatype.

2. Export SQL tables rows to JSON data.

3. Query external JSON data and store back in SQL tables.

**–> Note:**

– With this **CTP2** release you can **only export** data as JSON string.

– But with the release of **CTP3** you will also be able to **read JSON data** by T-SQL query and convert it into tabular (row/column) format, and will support indexes.

**–>** Just like XML for exporting JSON data you can use **FOR JSON [AUTO | PATH]** syntax:

**1. FOR JSON AUTO:** option automatically creates a nested JSON data with sub arrays based on the table hierarchy used in the Query. The **AUTO** option must have a **FROM** clause.

**2. FOR JSON PATH:** option enables you to define the structure of output of JSON data using the column names with aliases by using a dot separator.

**–> Let’s see how export to JSON works:**

– I’ll create a sample table and insert few rows in it:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | CREATE TABLE Students (      ID INT IDENTITY(1,1) NOT NULL,      FirstName VARCHAR(255),      LastName VARCHAR(255),      Class INT,      Marks DECIMAL(3,1)  )    INSERT INTO Students (FirstName, LastName, Class, Marks)  SELECT 'Manoj',   'Pandey', 10, 80.5  UNION ALL  SELECT 'Saurabh', 'Sharma', 11, 82.7  UNION ALL  SELECT 'Kanchan', 'Pandey', 10, 90.5 |

**1. Let’s check the “FOR JSON AUTO” option:**

|  |  |
| --- | --- |
| 1  2  3 | SELECT ID, FirstName, LastName, Class, Marks  FROM Students  FOR JSON AUTO -- here |

**– Output with AUTO option:**  
[SQL Server 2016 JSON 01](https://sqlwithmanoj.files.wordpress.com/2015/05/sql-server-2016-json-01.png)

**– Output with AUTO and ROOT() option:**

|  |  |
| --- | --- |
| 1  2  3 | SELECT ID, FirstName, LastName, Class, Marks  FROM Students  FOR JSON AUTO, ROOT('StudList') -- here |

– This is how a formatted JSON looks like:

{

"StudList": [

{

"ID": 1,

"FirstName": "Manoj",

"LastName": "Pandey",

"Class": 10,

"Marks": 80.5

},

{

"ID": 2,

"FirstName": "Gaurav",

"LastName": "Pandey",

"Class": 11,

"Marks": 82.7

},

{

"ID": 3,

"FirstName": "Garvit",

"LastName": "Pandey",

"Class": 10,

"Marks": 90.5

}

]

}

This option as mentioned previously formats the JSON document automatically based upon the columns provided in the Query.

**2. Now let’s check the “FOR JSON PATH” option:** with this option you can use the **dot syntax** as used in below Query to form a **nested output**.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | SELECT      ID,      FirstName AS "StudentName.FirstName",      LastName AS "StudentName.LastName",      Marks  FROM Students  FOR JSON PATH -- here |

**– Output with PATH option:**

[

{

"ID":1,

"StudentName":{

"FirstName":"Manoj",

"LastName":"Pandey"

},

"Marks":80.5

},

{

"ID":2,

"StudentName":{

"FirstName":"Saurabh",

"LastName":"Sharma"

},

"Marks":82.7

},

{

"ID":3,

"StudentName":{

"FirstName":"Kanchan",

"LastName":"Pandey"

},

"Marks":90.5

}

]

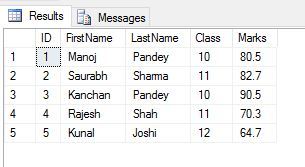
As you can see with **PATH option** you can **create wrapper objects** (here “StudentName”) and **nest properties** (here “FirstName” & “LastName”).

## Import/Read a JSON string and convert it in tabular (row/column) form | SQL Server 2016 – Part 2

But now with the **CTP 3** release you can do reverse of it also, means now you can read back JSON data and convert it to tabular or row & column format.

Let’s check this by taking same sample data from our previous [**JSON-Export post**](https://sqlwithmanoj.com/2015/06/01/exportconvert-table-or-sql-query-data-to-json-string-format-sql-server-2016-part-1/).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | DECLARE @json NVARCHAR(1000)  SELECT @json = N'{"StudList":  [     {        "ID":1,        "FirstName":"Manoj",        "LastName":"Pandey",        "Class":10,        "Marks":80.5     },     {        "ID":2,        "FirstName":"Saurabh",        "LastName":"Sharma",        "Class":11,        "Marks":82.7     },     {        "ID":3,        "FirstName":"Kanchan",        "LastName":"Pandey",        "Class":10,        "Marks":90.5     },     {        "ID":4,        "FirstName":"Rajesh",        "LastName":"Shah",        "Class":11,        "Marks":70.3     },     {        "ID":5,        "FirstName":"Kunal",        "LastName":"Joshi",        "Class":12,        "Marks":64.7     }  ]'    SELECT ID, FirstName, LastName, Class, Marks  FROM OPENJSON (@json, '$.StudList')  WITH (      ID INT,      FirstName VARCHAR(255),      LastName VARCHAR(255),      Class INT,      Marks DECIMAL(3,1)  ) AS StudList |

**– Output:**  
[](https://sqlwithmanoj.files.wordpress.com/2015/10/sql-server-2016-json-export-01.jpg)

Well, that was simple, isn’t it!!!

## Reading JSON string with Nested array of elements | SQL Server 2016 – Part 3

**Native JSON** support in SQL Server 2016 provides you few functions to read and parse your JSON string into relational format and these are:

**– OPENJSON()** Table valued function: parses JSON text and returns rowset view of JSON.

**– JSON\_Value()** Scalar function: returns a value from JSON on the specified path.

We will see usage of both the functions in our example below:

Here, we have just **one nested element**, and the OPENJSON() function will get you the child elements values.

**–> Method #1.a. Using OPENJSON() function:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | DECLARE @json NVARCHAR(1000)  SELECT @json =  N'{    "OrderHeader": [      {        "OrderID": 100,        "CustomerID": 2000,          "OrderDetail":          {            "ProductID": 2000,            "UnitPrice": 350          }      }    ]  }'    SELECT      OrderID,      CustomerID,        [OrderDetail.ProductID] AS ProductID,      [OrderDetail.UnitPrice] AS UnitPrice    FROM OPENJSON (@json, '$.OrderHeader')  WITH (      OrderID INT,      CustomerID INT,        [OrderDetail.ProductID] INT,      [OrderDetail.UnitPrice] INT  ) AS Orders |

**OrderID CustomerID ProductID UnitPrice**

100 2000 2000 350

But, if you have **more than one nested elements** the same query will give just 1 row with NULL values under the child columns, like this.

**–> Method #1.b. In case of multiple child elements:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | DECLARE @json NVARCHAR(1000)  SELECT @json =  N'{    "OrderHeader": [      {        "OrderID": 100,        "CustomerID": 2000,          "OrderDetail": [          {            "ProductID": 2000,            "UnitPrice": 350          },          {            "ProductID": 3000,            "UnitPrice": 450          }        ]      }    ]  }'    SELECT      OrderID,      CustomerID,        [OrderDetail.ProductID] AS ProductID,      [OrderDetail.UnitPrice] AS UnitPrice    FROM OPENJSON (@json, '$.OrderHeader')  WITH (      OrderID INT,      CustomerID INT,        [OrderDetail.ProductID] INT,      [OrderDetail.UnitPrice] INT  ) AS Orders |

**OrderID CustomerID ProductID UnitPrice**

100 2000 NULL NULL

You might be expecting 2 rows with same OrderID & CustomerID, with different ProductID & UnitPrice, right?

Instead you get ProductID & UnitPrice column values as NULL. Because, here you are having array of child elements with OrderDetail node (notice the square-bracket after **“OrderDetail”:** node), thus the Query is not able to find the key on the path.

In this case what you have to do is, use the array positions with square brackets (“[” and “]”) in your query and call out separate columns for each child element, like below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | DECLARE @json NVARCHAR(1000)  SELECT @json =  N'{    "OrderHeader": [      {        "OrderID": 100,        "CustomerID": 2000,          "OrderDetail": [          {            "ProductID": 2000,            "UnitPrice": 350          },          {            "ProductID": 3000,            "UnitPrice": 450          }        ]      }    ]  }'    SELECT      OrderID,      CustomerID,        [OrderDetail[0]].ProductID] AS ProductID1,      [OrderDetail[0]].UnitPrice] AS UnitPrice1,        [OrderDetail[1]].ProductID] AS ProductID2,      [OrderDetail[1]].UnitPrice] AS UnitPrice2    FROM OPENJSON (@json, '$.OrderHeader')  WITH (      OrderID INT,      CustomerID INT,        [OrderDetail[0]].ProductID] INT,      [OrderDetail[0]].UnitPrice] INT,        [OrderDetail[1]].ProductID] INT,      [OrderDetail[1]].UnitPrice] INT  ) AS Orders |

**OrderID CustomerID ProductID1 UnitPrice1 ProductID2 UnitPrice2**

100 2000 2000 350 3000 450

You can also specify the child elements with full path by using the dollar sign “$” inside the WITH() clause (instead at column level above), like below:

**–> Method #2. Using OPENJSON() function:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | DECLARE @json NVARCHAR(1000)  SELECT @json =  N'{    "OrderHeader": [      {        "OrderID": 100,        "CustomerID": 2000,          "OrderDetail": [          {            "ProductID": 2000,            "UnitPrice": 350          },          {            "ProductID": 3000,            "UnitPrice": 450          }        ]      }    ]  }'    SELECT      OrderID,      CustomerID,        ProductID1,      UnitPrice1,        ProductID2,      UnitPrice2    FROM OPENJSON (@json, '$.OrderHeader')  WITH (      OrderID    INT '$.OrderID',      CustomerID INT '$.CustomerID',        ProductID1 INT '$.OrderDetail[0].ProductID',      UnitPrice1 INT '$.OrderDetail[0].UnitPrice',        ProductID2 INT '$.OrderDetail[1].ProductID',      UnitPrice2 INT '$.OrderDetail[1].UnitPrice'  ) AS Orders |

**OrderID CustomerID ProductID1 UnitPrice1 ProductID2 UnitPrice2**

100 2000 2000 350 3000 450

Ok, so by using the key path and the array position we can get the child elements value in our Query result-set by using above 2 methods.

But instead of having them in separate columns how about pulling them in separate rows, this will also make your query dynamic as you would not know the number of child-elements before hand, right?

This can be done by CROSS APPLYing the JSON child node with the parent node and using the JSON\_Value() function, like shown below:

**–> Method #3. Using JSON\_Value() with OPENJSON() function:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | DECLARE @json NVARCHAR(1000)  SELECT @json =  N'{    "OrderHeader": [      {        "OrderID": 100,        "CustomerID": 2000,          "OrderDetail": [          {            "ProductID": 2000,            "UnitPrice": 350          },          {            "ProductID": 3000,            "UnitPrice": 450          },          {            "ProductID": 4000,            "UnitPrice": 550          }        ]      }    ]  }'    SELECT      JSON\_Value (c.value, '$.OrderID') as OrderID,      JSON\_Value (c.value, '$.CustomerID') as CustomerID,      JSON\_Value (p.value, '$.ProductID') as ProductID,      JSON\_Value (p.value, '$.UnitPrice') as UnitPrice    FROM OPENJSON (@json, '$.OrderHeader') as c  CROSS APPLY OPENJSON (c.value, '$.OrderDetail') as p |

**OrderID CustomerID ProductID UnitPrice**

100 2000 2000 350

100 2000 3000 450

100 2000 4000 550

Ok, that’s it for today.

## Store JSON data in a table, OPENJSON and JSON\_Value functions | SQL Server 2016 – Part 4

Here, in this post I’ll show how we can store JSON data in a normal table column, just like you store XML data.

XML data is stored in a column of XML datatype which also check the validity of the XML data to be stored. But to store JSON data there is no new datatype introduced, JSON can be stored in an **NVARCHAR** datatype column just like a plain text, and to validate it you can add a CHECK constraint on it.

**IsJSON() function:** can be used as a **CHECK constraint** on the columns that contain JSON string which will **validate** if the JSON string is in proper format or not.

As we will need **AdvantureWorks2014** Sample Database in our example below, we need to upgrade its Compatibility from SQL 2014 to SQL 2016, i.e. from level 120 to 130, like:

|  |  |
| --- | --- |
| 1  2  3  4  5 | USE [master]  GO    ALTER DATABASE [AdventureWorks2014] SET COMPATIBILITY\_LEVEL = 130  GO |

You can download AdvantureWorks2014 sample Database from Microsoft **[[CodePlex site](https://sqlwithmanoj.com/2014/08/05/adventureworks-2014-sample-database-released-for-sql-server-2014-hekaton/" \t "_blank)]**.

**–>** Ok, now let’s create a new Table with **OrderDetailsJSON** column for storing JSON string with a **CHECK constraint** on it:

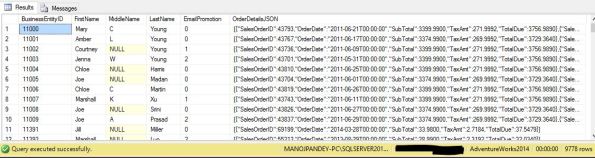
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | USE [AdventureWorks2014]  GO    CREATE TABLE CustomerOrder (      BusinessEntityID INT,      FirstName NVARCHAR(50),      MiddleName NVARCHAR(50),      LastName NVARCHAR(50),      EmailPromotion INT,        OrderDetailsJSON NVARCHAR(MAX) -- normal column with NVARCHAR datatype          CHECK ( IsJSON ( OrderDetailsJSON ) = 1 ) -- CHECK Constraint to validate JSON string  ) |

**–>** Let’s create a sample record-set with JSON data in **OrderDetailsJSON** column. We will use **FOR JSON AUTO** option to convert relational data to JSON string for our example, as shown below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | ;WITH CTE\_PersonContact AS (      SELECT          BusinessEntityID, FirstName, MiddleName, LastName, EmailPromotion,          OrderDetailsJSON =          (   SELECT SalesOrderID, OrderDate, SubTotal, TaxAmt, TotalDue              FROM [AdventureWorks2014].[Sales].[SalesOrderHeader] S              WHERE S.CustomerID = P.BusinessEntityID              FOR JSON AUTO -- here          ) -- our JSON column      FROM [Person].[Person] P  )  INSERT INTO CustomerOrder  SELECT      BusinessEntityID, FirstName, MiddleName, LastName, EmailPromotion,      OrderDetailsJSON  FROM CTE\_PersonContact  WHERE OrderDetailsJSON IS NOT NULL    -- (9778 row(s) affected) |

**–>** Check the above inserted records with the **OrderDetailsJSON** column containing data in JSON format:

|  |  |
| --- | --- |
| 1  2  3 | SELECT \* FROM CustomerOrder    -- (9778 row(s) affected) |

[](https://sqlwithmanoj.files.wordpress.com/2015/11/sql-server-2016-json-table.jpg)

**–>** Let’s Query back the JSON data from the **OrderDetailsJSON** column with other columns in **relational** form, by using **OPENJSON()** function. As for each Customer it can contain multiple orders we will get multiple rows for each Customer and multiple columns as per defined in the JSON string:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | SELECT      C.BusinessEntityID, C.FirstName, C.MiddleName, C.LastName, C.EmailPromotion,      J.SalesOrderID, J.OrderDate, J.SubTotal, J.TaxAmt, J.TotalDue  FROM CustomerOrder C  CROSS APPLY OPENJSON (OrderDetailsJSON)  WITH (      SalesOrderID INT,      OrderDate DATETIME,      SubTotal MONEY,      TaxAmt MONEY,      TotalDue MONEY  ) AS J    -- (17463 row(s) affected) |

[](https://sqlwithmanoj.files.wordpress.com/2015/11/sql-server-2016-json-table-2.jpg)

**–>** And if you want to get just one Order per Customer then you can use following Query, by using **JSON\_Value()** function and by specifying the array key pointer/position to get the first value fro the array:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | SELECT      C.BusinessEntityID, C.FirstName, C.MiddleName, C.LastName, C.EmailPromotion,      SalesOrderID        = JSON\_Value (OrderDetailsJSON, '$[0].SalesOrderID'),      OrderDate           = JSON\_Value (OrderDetailsJSON, '$[0].OrderDate'),      SubTotal            = JSON\_Value (OrderDetailsJSON, '$[0].SubTotal'),      TaxAmt              = JSON\_Value (OrderDetailsJSON, '$[0].TaxAmt'),      TotalDue            = JSON\_Value (OrderDetailsJSON, '$[0].TotalDue')  FROM CustomerOrder C    -- (9778 row(s) affected) |

## Passing multiple/dynamic values to Stored Procedures & Functions | Part 5 – by passing JSON string

Adding the **fifth part** to this series we will use **JSON string** that will contain the set of values and pass as an JSON param variable to the SP. Then inside the SP we will parse this JSON and use those values in our SQL Queries, just like we did in previous posts with CSV/XML strings:

USE [AdventureWorks2014]

GO

-- Create an SP with NVARCHAR type parameter for JSON string:

CREATE PROCEDURE uspGetPersonDetailsJSON (

    @persons NVARCHAR(MAX)

)

AS

BEGIN

    --DECLARE @persons NVARCHAR(MAX)

    --SET @persons = '{"root":[{"Name":"Charles"},{"Name":"Jade"},{"Name":"Jim"},{"Name":"Luke"},{"Name":"Ken"}]}'

    SELECT Name

    INTO #tblPersons

    FROM OPENJSON (@persons, '$.root')

    WITH (

        Name NVARCHAR(100)

    )

    SELECT

        BusinessEntityID,

        Title,

        FirstName,

        MiddleName,

        LastName,

        ModifiedDate

    FROM [Person].[Person] PER

    WHERE EXISTS (

        SELECT \*

        FROM #tblPersons tmp

        WHERE tmp.Name  = PER.FirstName

    )

    ORDER BY FirstName, LastName

    DROP TABLE #tblPersons

END

GO

-- Create JSON string:

DECLARE @json NVARCHAR(1000)

SET @json = N'{

  "root": [

    { "Name": "Charles" },

    { "Name": "Jade" },

    { "Name": "Jim" },

    { "Name": "Luke" },

    { "Name": "Ken" }

  ]

}'

-- Use the JSON string as parameter which calling the SP:

EXEC uspGetPersonDetailsJSON @json

GO

-- Check the output, objective achieved

-- Final Cleanup

DROP PROCEDURE uspGetPersonDetailsXML

GO

Thus you can also use JSON string similar to the way you used XML string, to pass multiple and dynamic number of parameters to your Stored Procedures.

As JSON feature is new to SQL Server 2016, so this method will only work with SQL Server 2016 and above versions.