What is XML?

- XML stands for extensible Markup Language
- It does not do anything; it is just a medium to store and transport data.

Basic structure:

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

```
<ordderinfo>

<header>   ….</header>

<detal>

   <order>

         <item>Bikes</ITEM>

         <ITEM>SCOOTERS</ITEM>

     </ORDER>

 <ORDER>

        <item>Bikes</ITEM>

         <ITEM>SCOOTERS</ITEM>

   </ORDER>

</ORDERINFO>

  </detail>
```

XML-INTO (Parsing of XML)

XML-INTO DSNAME %XML(FILE OR VARIABLE : OPTIONS)

Sample:

```
clear order ;
 option  = 'case=any doc=file allowmissing=yes allowextra=yes -
```

```
                      countprefix=c_path=’path name’' ;
           FILENAME   = '/TEST/ORDER.XML' ;

           xml-into ORDER %xml(%trim(FILENAME):%trim(option)) ;
```

Option values:

Case = any (Elements can be in any case)

Doc = file (document is in IFS file)

Allowmissing = yes (even if document has missing tags ,parsing can take place)

Allowextra = yes (even if document has extra tags ,parsing can take place)

path = within xml if we want to parse specific path example xml has both header and detail and we just need to parse detail.

NS = REMOVE OR MERGE (working with namespace)

If we get run time error for XML parsing, we can refer below site to know details of parsing error codes.

https://www.ibm.com/docs/en/i/7.3?topic=documents-xml-parser-error-codes

Below example shows simple XML parsing example where XML is defined in AS400 constant.

```
     dcl-c xmlfld   '<ordinfo> <ORDER> <ITEM>Bikes</ITEM>  +
                                <QTY>5</QTY></ORDER>  +
                                <ORDER> <ITEM>SCOT</ITEM>  +
                                <QTY>5</QTY></ORDER> </ordinfo>' ;


     dcl-ds order  dim(5) qualified ;
       item  char(10) ;
       qty zoned(3:0) ;
     end-ds ;
    dcl-s option char(500) inz ;

    option =  'case=any allowmissing=yes allowextra=yes'  ;
     xml-into order %xml(%TRIM(xmlfld):%TRIM(option))  ;
   *inlr = '1' ;
```

Below example shows NS=REMOVE example

```
dcl-c xmlfld   '<O:ordinfo> <O:ORDER> <O:ITEM>Bikes</O:ITEM>  +
                                  <O:QTY>5</O:QTY></O:ORDER>  +
                                  <O:ORDER> <O:ITEM>SCOT</O:ITEM>  +
                                  <O:QTY>5</O:QTY></O:ORDER>  +
                                  </O:ordinfo>' ;
     dcl-ds order  dim(5) qualified ;
       item  char(10) ;
       qty zoned(3:0) ;
     end-ds ;
    dcl-s option char(500) inz ;

    option =  'case=any allowmissing=yes allowextra=yes ns=remove' ;
     xml-into order %xml(%TRIM(xmlfld):%TRIM(option))  ;
   *inlr = '1' ;
```

<u>Below example shows XML parsing where XML is defined in IFS file.</u>

<u>Option doc =file is required for this.</u>

```
dcl-ds PgmDs psds ;
   Count_ELEM int(20) pos(372) ;
 end-ds ;

 dcl-ds order  dim(5) qualified ;
   item  char(10) ;
   qty zoned(3:0) ;
   DCL-DS ADDRESS ;
     CITY CHAR(10) ;
     STATE CHAR(10)  ;
   END-DS  ;
 end-ds ;
dcl-s option char(500) inz ;
dcl-s filename  char(100) inz ;

option =  'case=any allowmissing=yes allowextra=yes doc=file' ;
filename = '/praful/ordxml.xml' ;
  xml-into order %xml(%TRIM(filename):%TRIM(option))  ;
```

```
*inlr = '1' ;
```

Below example shows NS=merge example

```
   dcl-c xmlfld   '<O:ordinfo> <O:ORDER> <O:ITEM>Bikes</O:ITEM>  +
                              <O:QTY>5</O:QTY></O:ORDER>  +
                              <O:ORDER> <O:ITEM>SCOT</O:ITEM>  +
                              <O:QTY>5</O:QTY></O:ORDER>  +
                              </O:ordinfo>' ;
   dcl-ds o_order  dim(5) qualified ;
     o_item  char(10) ;
     o_qty zoned(3:0) ;
   end-ds ;
  dcl-s option char(500) inz ;

  option =  'case=any allowmissing=yes allowextra=yes ns=merge' ;
   xml-into o_order %xml(%TRIM(xmlfld):%TRIM(option))  ;
 *inlr = '1' ;
```

Countprefix : we can have subfield to return count of each tag.

```
  dcl-ds order  dim(5) qualified ;
       item  char(10) ;
       cntitem zoned(3:0) ;
       qty zoned(3:0) ;
       cntqty  zoned(3:0) ;
       DCL-DS ADDRESS ;
         CITY CHAR(10) ;
         STATE CHAR(10)  ;
       END-DS  ;
     end-ds ;
    dcl-s option char(500) inz ;
```

```
       option =  'case=any allowmissing=yes allowextra=yes ' +
                 'countprefix=cnt' ;
        xml-into order %xml(%TRIM(xmlfld):%TRIM(option))  ;
      *inlr = '1' ;
```

JSON (Java script object Notation)

It is also used to transmit data, just like XML but format is much more in human readable mode.

Simple JSON example:

```
"employee": {
   "name":       "sonoo",
   "salary":      56000,
   "married":    true
}
```

JSON array example

```
"employees":[
   {"name":"Shyam", "email":"shyamjaiswal@gmail.com"},
   {"name":"Bob", "email":"bob32@gmail.com"},
   {"name":"Jai", "email":"jai87@gmail.com"}
]}
```

DATA-INTO

Just like XML-INTO which is used just to Parse XML document into an RPG variable, we have DATA-INTO opcode, this is used to parse/import all structured document into a RPG variable/Data structure, mostly it is used to parse JSON data in day to day operations**.**

DATA-INTO result %DATA (document[:options]) %PARSER(parser[:options]);

We have an extra parser option here; this can be 3rd Party program or IBM provided JSON parser.

Mostly **YAJLINTO parser** is used ,Here options are same like XML-INTO.

Example:-

```
D Address       DS
D Street                  10
D City                    10
D State                   10
D Postal                  10


myJSON = '{ + "street": "123 Example Street", +
               "city": "Milwaukee" ,            +
               "state": "WI",                   +
               "postal": "53201-1234"           +
               }';

DATA-INTO address %DATA( myjson : 'case=any allowmissing=yes
allowextra=yes) %PARSER('YAJLINTO');
```

In this example it will parse Json present in Myjson variable into address data structure. External parse YAJLINTO is used in this example, majority of places YAJLINTO parser is used .

In DATA-INTO options are same as XML-INTO with only addition of %parser part.

DATA-GEN(Generate a document from RPG variable)

If we need to generate XML/JSON or any other structured document from RPG variable this opcode can be used.

The DATA-GEN operation generates a structured document from an RPG variable. DATA-GEN requires a generator program or procedure to generate the text for the document.
The DATA-GEN operation passes the names and values of the *source* variable to the generator, which uses callback functions to gradually pass text for the document to the DATA-GEN operation. The DATA-GEN operation places the information into the target RPG variable or the target Integrated File System file specified by the %DATA built-in function.

Syntax:

DATA-GEN   IP_DS_FOR_DOC_GENERATION
  %DATA (RPGVARIABLE or file where JSON output will be placed:Options)
%GEN(Generator_pgm)

<u>Option values:</u>

doc – controls where the document is generated string (default) or file.

- trim – remove extra blanks from strings
- countprefix – control the number of specified elements generated
- fileccsid – specifies the CCSID when creating an output file
- name – specifies the name of the top-level element (for document)
- output – should the output variable/file be cleared? Or appended?
- renameprefix – lets you specify variables containing alternate names for subfields. %DATA .

Basically Generator_pgm used commonly for **JSON generation** is **YAJLDTAGEN** from YAJL library.

1. DATA-GEN address %DATA(ADDRJson) %GEN('YAJLDTAGEN');

In this example input data structure address will be generated as JSON and result will be placed in ADDRJSON variable.

2. myStmf = '/home/praful /address.json';

data-into address %DATA(myStmf:'doc=file') %GEN('YAJLDTAGEN');

In this example JSON will be generated in IFS folder from address data structure.

<u>SQL functions in DB2 to build JSON and Parse Json</u>

JSON_Object : This can generate JSON Variable from input record or group of records.

Example :

Create sqlrpgle program which receives employee number as input and written record of that employee in JSON format.

```
DWEMPNO           S             10  0
 DJSON_DATA         S           1000

 C      *eNTRY         PLIST
 C                     PARM                         WEMPNO

          exec sql SELECT JSON_OBJECT(
                  'Employee Name ' : trim(EMPNAME),
                  'Employee age '  : EMPAGE ,
                  'Salary'         : SALARY)
into:json_Data
        FROM EMPPF
       WHERE EMPNO  = :Wempno   ;

        if sqlcode= 100 ;
           dsply 'No data found' ;
        endif ;

        *inlr = '1' ;
```

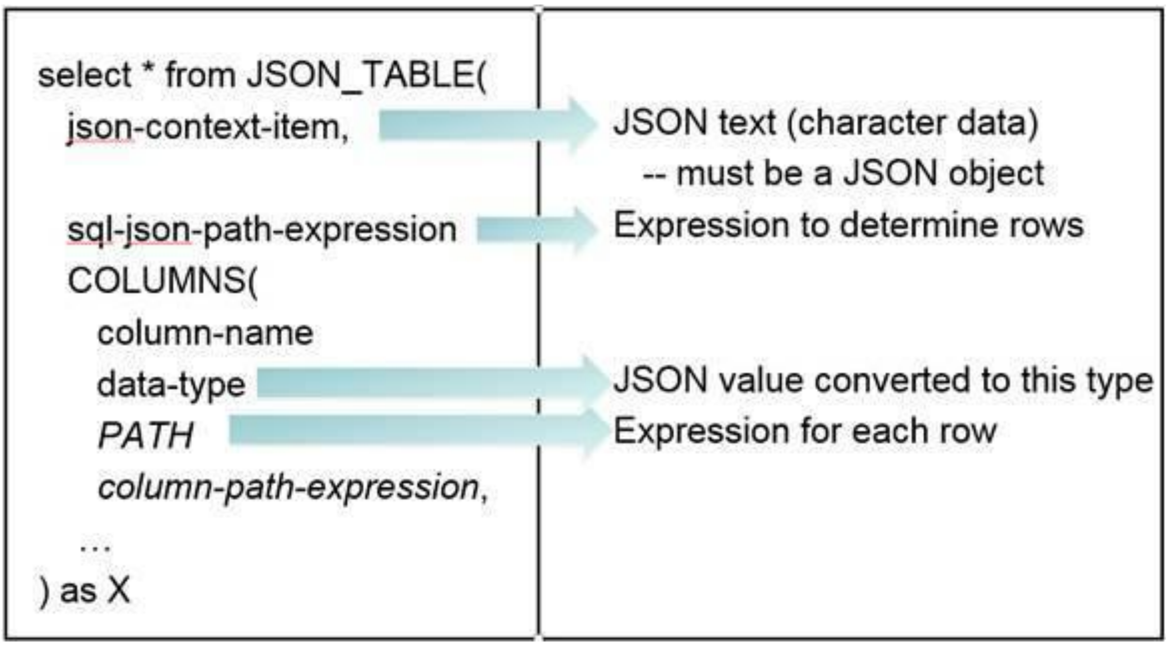
JSON_Table :

The JSON_TABLE table function converts a JSON document into a relational table.

This is basically used to

We will work with the following table, EMP, which contains four rows with one JSON object per row. By using the JSON_TABLE function we will extract the data so that it can be treated as relational data.

Syntax for this table function :



```
select * from JSON_TABLE(
  json-context-item,
  sql-json-path-expression
  COLUMNS(
    column-name
    data-type
    PATH
    column-path-expression,
   ...
) as X
```

JSON text (character data)
-- must be a JSON object

Expression to determine rows

JSON value converted to this type

Expression for each row

Example :

```
addrJSON = '{ + "street": "123 Example Street", +
               "city": "Milwaukee" ,           +
               "state": "WI",                  +
               "postal": "53201-1234"          +
               }';
```

Insert employee address information from JSON into emptable: -

```
Insert into emppf  (empstrt  , empcity, empstate , emppostal)

Values (

SELECT

 t.street, t.city, t.state , t.postal

   FROM

      JSON_TABLE(

              :addrJson,
'lax $'
COLUMNS (
        street VARCHAR(10) PATH 'lax $.street',
        city VARCHAR(10) PATH 'lax $.city',
        state VARCHAR(10) PATH 'lax $.city',
         postal varchar PATH ‘lax $.postal’
        )) AS t;
```

Lax $: Suggest name given to json \

Lax $.name