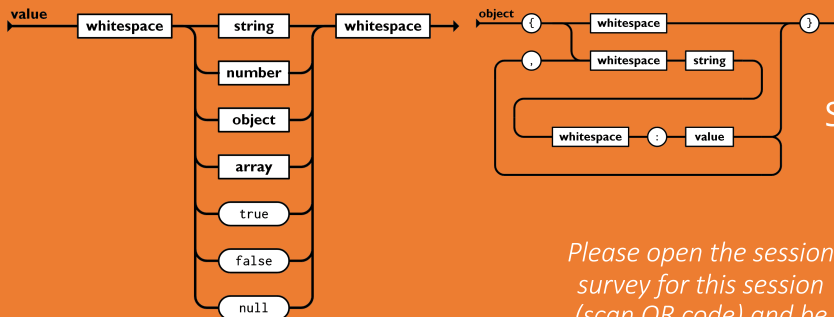
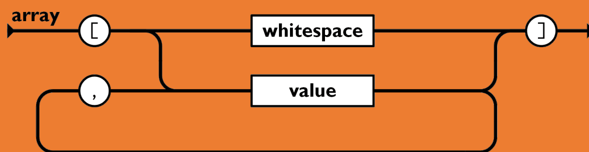
Handling JSON with DATA-INTO and DATA-GEN in ILE RPG



by
Scott Klement

Please open the session survey for this session (scan QR code) and be ready to evaluate it.



Session Concept

When writing most REST API, you use JSON format. (In most cases, it has replaced XML.)

Alternately, JSON is sometimes also sent/received to companies via other means aside from APIs. But, in any case, you typically have these tasks you need to handle:

1. Interpret/Read incoming JSON in a string or file.
2. Do your business logic (utilizing the data you got from the JSON)
3. Create an output JSON file to send back.

We won't discuss part 2 -- it's assumed that once the data is in variables in your program, you know how to write your business logic -- that part is just normal RPG business programming.

JSON and XML to Represent a DS

```
dcl-ds list qualified dim(2);
  custno packed(4: 0);
  name   char(25);
end-ds;
```

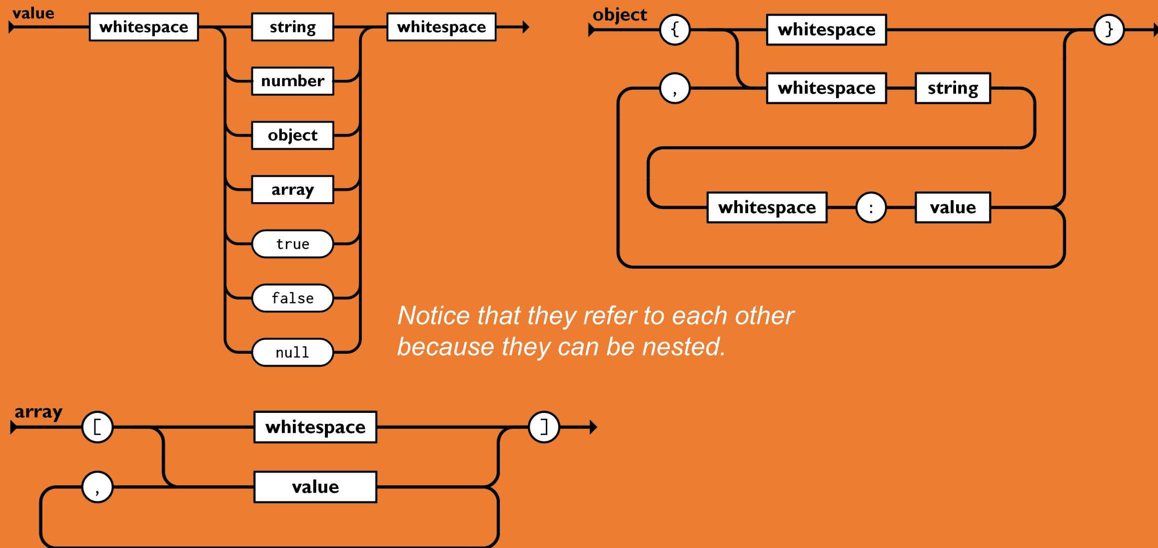
Array of data structures in RPG...

```
[
  {
    "custno": 1000,
    "name": "ACME, Inc"
  },
  {
    "custno": 2000,
    "name": "Industrial Supply Limited"
  }
]
```

Array of data structures in JSON

```
<list>
  <cust>
    <custno>1000</custno>
    <name>Acme, Inc</name>
  </cust>
  <cust>
    <custno>2000</custno>
    <name>Industrial Supply Limited</name>
  </cust>
</list>
```

Array of data structures in XML



DATA-INTO

DATA-INTO is an RPG opcode that makes it easy to process JSON in RPG.

Concept:

- Define an RPG variable (usually a data structure) that matches the JSON document.
- Tell DATA-INTO where the JSON is, and where your RPG variable is.
- It will map from the document to your variable.
- After that, then you can process it in your program the same way you'd use any other RPG variable!

Mapping JSON Format

JSON format:

- The { } characters indicate an "object" (same as RPG data structure)
- The [ ] characters indicate an array
- Just as with XML, we can map them into an RPG structure

```
dcl-ds address;
  street varchar(30);
  city   varchar(20);
  state  char(2)     ;
  postal varchar(10);
end-ds;
```

```
{
 "street": "123 Main Street",
 "city":   "Anywhere",
 "state":  "WI",
 "postal": "12345"
}
```

DATA-INTO Syntax

The DATA-INTO opcode syntax is:

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

result = RPG variable (data structure) that data will be loaded into

document = the JSON document, or IFS path to the JSON document.

%DATA options = optional parameter containing options passed to RPG to control the reading of the JSON document, or how it is mapped into variables

%PARSER options = optional parameter containing options passed to the parser program. The syntax will vary depending on the parser program.

%HANDLER = like XML-INTO, the DATA-INTO opcode supports a handler. This was more widely used in IBM i 5.4 when variable sizes were more limiting. I will not cover this today.

Why Do We Need a %PARSER?

DATA-INTO overcomes a big limitation we had in XML-INTO
- only works with XML!
- many (thousands) of other document types exist
- other formats used in business today include: YAML, CSV, JSON, XDR, Property List, Pickle, OpenDDL, protobuf, OGDL, KMIP, FHIR, Feather, Arrow, EDN, CDR, Coifer, CBOR, Candle, Bond, Bencode, D-Bus, ASN.1, HOCON, MessagePack, SCaViS, Smile, Thrift, VPack

DATA-INTO
- RPG won't try to understand the document
- Calls 3rd-party tool ("parser") which interprets the document
- ...but, DATA-INTO maps result into RPG variable
- *...all you need is the right parser to read any format!*

YAJLINTO Parser

Example of DATA-INTO with YAJLINTO as the Parser:

```
DATA-INTO result %DATA( '/tmp/example.json'
                      : 'doc=file case=convert countprefix=num_')
                 %PARSER('YAJLINTO');
```

result – the name of RPG data structure that I want to load the JSON into. You can name it whatever you like on your DCL-DS.

/tmp/example.json - IFS path to the JSON document we generated

doc=file – tells RPG to read the document from a file (vs. a variable)

case=any – tells RPG that the upper/lower case of variable names does not have to match the document

countprefix=num_ – any variables in the DS that start with "num_" should receive counts of matching fields. For example, "num_list" would give the number elements in the "list" array.

Basic JSON Example

Basic DATA-INTO example using YAJLINTO

```
dcl-ds address;
  street varchar(30);
  city   varchar(20);
  state  char(2)    ;
  postal varchar(10);
end-ds;

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

data-into address %DATA(myJSON) %PARSER('YAJLINTO');
```

For simplicity, myJSON is a string built in the program. But, it could've been a parameter, read from an API call, etc.

DATA-INTO Options

Specified as the 2nd parameter to %DATA to modify DATA-INTO behavior

- doc – controls where the document is read from string (default) or file.
- case – controls whether upper/lower case field names must match.
- allowmissing – allow elements in the document to be missing
- allowextra – allow extra elements in the document
- countprefix – ask data-into count the number of specified elements
- path – specifies the subset of the document to be read
- trim – remove extra whitespace from elements
- ccsid – specifies the CCSID passed to the parser

```
%DATA(myStmf:'put options here')
```

DOC Option

The default is doc=string (read from a string)

doc=file tells DATA-INTO to read the data from the IFS. The first parameter to %DATA is now the IFS path name.

Imagine the "address" example (from the first example) was in an IFS file named /home/scott/address.json

```
myStmf = '/home/scott/address.json';

data-into address %DATA(myStmf:'doc=file') %PARSER('YAJLINTO');
```

CASE Option

The default is case=lower

- lower = the fields in the document are all lowercase
- upper = the fields in the document are all uppercase
- any = treat the fields as case-insensitive (field names in RPG and the document are converted to all uppercase before comparing)
- convert = Like 'any', except that characters with diacritics (such as accented characters) are converted to their un-accented equivalents and other characters (aside from A-Z, 0-9) are converted to underscores.

NOTE: In my experience it's unusual for the upper/lower case of characters to matter. Since characters not allowed in RPG (such as blanks and dashes) are often used in documents such as JSON and XML, I almost always use case=convert.

CASE Example

The following code will fail because "Postal" is not all lowercase.
Error: RNQ0356 The document for the DATA-INTO operation does not match the RPG variable.

```
dcl-ds address1;
  postal varchar(10);
end-ds;

myJSON = '{ "Postal": "53201-1234" }';
data-into address1 %DATA(myJSON) %PARSER('YAJLINTO');
```

It can be fixed by using case=any or case=convert. This works:

```
myJSON = '{ "Postal": "53201-1234" }';
data-into address1 %DATA(myJSON:'case=convert') %PARSER('YAJLINTO');
```

Likewise, case=convert works when the document has a field that isn't a valid RPG variable name:

```
dcl-ds address2;
  postal_code varchar(10);
end-ds;

myJSON = '{ "Postal Code": "53201-1234" }';
data-into address2 %DATA(myJSON:'case=convert') %PARSER('YAJLINTO');
```

CountPrefix Option (1 of 3)

CountPrefix creates a prefix used when counting document elements.

- by default, counting does not take place, so there is no default value.

To understand, imagine you receive the following "statement.json" file from a vendor. It is a statement, telling what you owe for a given month.

```
{
  "customer": 5406,
  "statement date": "2018-10-05",
  "start date": "2018-09-01",
  "end date": "2018-09-30",
  "statement total": 6600.00,
  "invoices": [
     { "invoice": "99001", "amount": 1000.00, "date": "2018-09-14" },
     { "invoice": "99309", "amount": 1500.00, "date": "2018-09-18" },
     { "invoice": "99447", "amount": 500.00, "date": "2018-09-23" },
     { "invoice": "99764", "amount": 3600.00, "date": "2018-09-14" }
  ]
}
```

Now imagine the RPG code needed to read this....

CountPrefix Option (2 of 3)

Any field in my DS beginning with the prefix is NOT mapped from the document, but instead is a count of a corresponding field.

Example: countprefix=total_, then total_XYZ is a count of the XYZ elements.

Or, for the invoice list:

```
dcl-ds statement qualified;
  customer packed(4: 0);
  statement_date char(10);
  start_date char(10);
  end_date char(10);
  statement_total packed(11: 2);
  num_invoices int(10);
  dcl-ds invoices dim(999);
    invoice char(5);
    amount  packed(9: 2);
    date    char(10);
  end-ds;
end-ds;

data-into statement %DATA('statement.json'
                         :'doc=file case=convert countprefix=num_')
                    %PARSER('YAJLINTO');
```

CountPrefix Option (3 of 3)

You can now use num_invoices to loop through the data. For example:

```
.
.
for x = 1 to statement.num_invoices;
   prinvn = statement.invoices(x).invoice;
   prdamt = statement.invoices(x).amount;
   prsdat = statement.invoices(x).date;
   write prrec;
endfor;
.
.
```

This example writes the fields to a database table (physical file).

This also illustrates the use of nested data structures/arrays. You separate each nested level with a period and place the array index (the (x) above) on the level that is an array.

CountPrefix For Optional Elements

NOTE: CountPrefix can be used to replace AllowMissing!

- When a counter defined, RPG will not issue an error if the corresponding element is missing.
- Instead of an error, RPG will set the counter field to 0.
- Your code can then check the counter to determine if the field did/didn't exist.

In most cases, this is a better option than the AllowMissing=yes option, which can make it more difficult to understand mistakes in your RPG DS.

(In fact, for that reason, I will not cover AllowMissing in this talk.)

%PARSER Options

The %PARSER function also has a space for options.

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

Options specified under %DATA are handled by DATA-INTO in the RPG compiler itself.

Options on %PARSER are handled by the 3rd-party parser program and will differ with each parser you use!

%PARSER Options:

- Can be coded as a string literal. Or can be an RPG variable.
- The parser determines the format of the parser options and what variable type(s) it will accept.

YAJLINTO %PARSER Options

YAJLINTO expects:

- %parser options are passed as a small JSON document
- Must be a literal or an RPG character string variable
- If using a variable, it must be in job's CCSID (EBCDIC)
- No options are required – *only specify the ones you need to use.*

YAJLINTO's options are:

- document_name = a string representing the name of the document node (used with the PATH option)
- value_true = value to place in RPG variable for a JSON boolean that is true. (Default='1' – this is ideal if mapping to an RPG indicator.)
- value_false = value to place in RPG variable for a JSON boolean that is false. (Default='0' – same reason.)
- value_null = value to place in RPG variable if the special value null is provided for a field in the JSON document. (default: '*NULL')

```
data-into invoices  %DATA( 'statement.json'
                         : 'doc=file case=convert path=statement/invoices')
                    %PARSER( 'YAJLINTO'
                           : '{ +
                              "value_true": "true", +          (default is '1')
                              "value_false": "false", +        (default is '0')
                              "value_null": "**NONE**", +      (default is *NULL)
                              "document_name": "statement" +   (default is no name)
                            }');
```

YAJLINTO with a web service

YAJLINTO has a special feature for writing web services:

- use this when RPG is called from Apache via ScriptAlias
- primarily for "do it yourself" style web services
- *not* for use with tools like Integrated Web Services or WebSphere

```
data-into result %DATA( '*STDIN'
                      : 'case=convert countprefix=num_')
                 %PARSER('YAJLINTO');
```

Since September 2018, YAJLINTO supports direct reading from standard input by passing the special value *STDIN.

See Scott's other presentations for more information:

- *Providing Web Services on IBM i* (Do It Yourself section)
- *Working with JSON in RPG*

DATA-GEN

DATA-GEN is an RPG opcode that makes it easy to create a JSON document in RPG.

Concept:

- Define an RPG variable (usually a data structure) that matches the JSON document.
- Tell DATA-GEN where the RPG variable is, and where you want it to put the JSON document.
- It will make the document from your variable.

(It's like DATA-INTO, but in reverse!)

What?

For example:

```
{
  "name": "Scott Klement",
  "street": "8825 S Howell Avenue Ste 301",
  "city": "Oak Creek",
  "state": "WI",
  "postal": "53154"
}
```

How?

```
dcl-s Json varchar(1000);

dcl-ds address qualified;
  name   varchar(30)  inz('Scott Klement');
  street varchar(30)  inz('8825 S Howell Avenue');
  city   varchar(20)  inz('Oak Creek');
  state  char(2)      inz('WI');
  postal varchar(10)  inz('53154');
end-ds;

DATA-GEN address %DATA(Json) %GEN('YAJLDTAGEN');
```

Yeah. It's easy. DATA-GEN put the document in the Json variable.

Why?

Each JSON thing has an RPG equivalent.

DATA-GEN makes the JSON thing from the RPG thing.

```
{
  "sub field 1": 123.45,
  "sub field 2": "string goes here",
  "accepted": true,
  "days open": [ "Monday", "Wednesday", "Friday" ]
}
```

Characters	Json Meaning	RPG Equivalent
"string goes here"	Character string	CHAR or VARCHAR
123.45	Number	Packed, Zoned, Int, Float, etc
true	Boolean (true or false)	Indicator (*ON or *OFF)
{ "field": "value" }	Object	Data Structure
[1, 2, 3]	Array	DIM

Look Again

Makes JSON things from RPG things.

```
DATA-GEN address %DATA(Json) %GEN('YAJLDTAGEN');
```

```
dcl-ds address qualified;
  name    varchar(30);
  street  varchar(30);
  city    varchar(20);
  state   char(2);
  postal  varchar(10);
end-ds;
```

```
{
  "name": "string",
  "street": "string",
  "city": "string",
  "state": "string",
  "postal": "string"
}
```


Characters	Json Meaning	RPG Equivalent
"string goes here"	Character string	CHAR or VARCHAR
123.45	Number	Packed, Zoned, Int, Float, etc
true	Boolean (true or false)	Indicator (*ON or *OFF)
{ "field": "value" }	Object	Data Structure
[1, 2, 3]	Array	DIM

What the parts of DATA-GEN mean and do.

DATA-GEN Syntax

```
DATA-GEN source-variable %DATA(result {: options}) %GEN(generator {: options});
```

- source-variable: RPG variable (usually a data structure) to generate the structured document from.
- result: Specifies the result variable, either as a character variable (default) or as an IFS pathname to write to.
- result options: Space-separated list of options that control how RPG transfers data from your source variable into the result (more to come!)
- generator: Third-party program or service program that will generate the document. The generator is what determines the format of the document you're generating.
- generator options: Character literal or RPG variable that contains options used by the generator. The format of this variable is defined by the generator program and will be different for each generator you use.

Many options exist for %DATA.

Here are the most commonly used ones.

%DATA Options

- doc – controls where the document is generated string (default) or file.
- countprefix – control the number of specified elements generated
- renameprefix – lets you specify variables containing alternate names for subfields.

```
%DATA(myStmf:'put options here')
```

Notice that doc and countprefix are (more or less) the same in DATA-GEN as they were in DATA-INTO.

Writing a File

doc=file changes the first parameter to %DATA to be an IFS path name, then writes there.

```
dcl-s MyFile varchar(100);

MyFile = '/home/scott/address.json';
data-gen address %data(MyFile: 'doc=file') %gen('YAJLDTAGEN');
```

Variable Length Arrays

How to deal with variable-length arrays? Example: An invoice has a variable number of items on it.

```
{
  "items": [
    { "itemNo": 1001, "desc": "Some Description",   "qty":  12, "price":  51.99 },
    { "itemNo": 1002, "desc": "Second Description", "qty":   6, "price":  94.10 },
    { "itemNo": 1003, "desc": "Third Description",  "qty":  20, "price":  12.00 },
    { "itemNo": 1004, "desc": "Silly Things",       "qty": 104, "price":   3.75 },
    { "itemNo": 1005, "desc": "Some other things",  "qty":   3, "price": 101.06 }
  ]
}
```

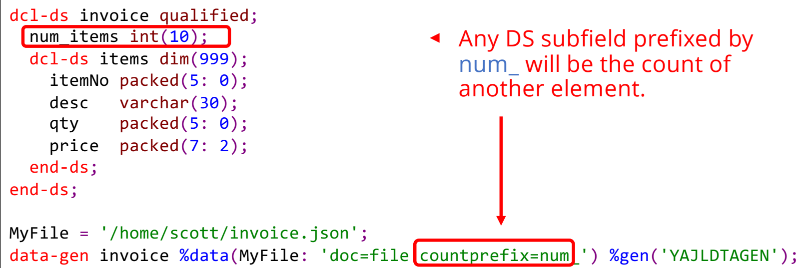
```
dcl-ds invoice qualified;
  dcl-ds items dim(999);
    itemNo packed(5: 0);
    desc   varchar(30);
    qty    packed(5: 0);
    price  packed(7: 2);
  end-ds;
end-ds;
```

A DS like this would be a problem. It would output 999 elements.

CountPrefix

How to deal with variable-length arrays? Example: An invoice has a variable number of items on it.

```
{
  "items": [
    { "itemNo": 1001, "desc": "Some Description",   "qty":  12, "price":  51.99 },
    { "itemNo": 1002, "desc": "Second Description", "qty":   6, "price":  94.10 },
    { "itemNo": 1003, "desc": "Third Description",  "qty":  20, "price":  12.00 },
    { "itemNo": 1004, "desc": "Silly Things",       "qty": 104, "price":   3.75 },
    { "itemNo": 1005, "desc": "Some other things",  "qty":   3, "price": 101.06 }
  ]
}
```



```
dcl-ds invoice qualified;
  num_items int(10);
  dcl-ds items dim(999);
    itemNo packed(5: 0);
    desc   varchar(30);
    qty    packed(5: 0);
    price  packed(7: 2);
  end-ds;
end-ds;

MyFile = '/home/scott/invoice.json';
data-gen invoice %data(MyFile: 'doc=file countprefix=num_') %gen('YAJLDTAGEN');
```


Special Names

When a name in a JSON document isn't a possible RPG field name...

```
{
  "customer name": "string",
  "street address": "string",
  "city": "string",
  "state": "string",
  "postal": "string"
}
```

This will NOT work; RPG doesn't allow spaces or quotes in a variable name:

```
dcl-ds address qualified;
  "customer name"   varchar(30);
  "street address"  varchar(30);
   city             varchar(20);
   state            char(2);
   postal           varchar(10);
end-ds;
```

RenamePrefix

When a name in a JSON document isn't a possible RPG field name...

```
dcl-ds address qualified;
  name          varchar(13);
  name_name     varchar(30) inz('customer name');
  street        varchar(30);
  name_street   varchar(14) inz('street address');
  city          varchar(20);
  state         char(2);
  postal    varchar(10);
end-ds;

data-gen invoice %data(MyFile: 'doc=file renameprefix=name_')
                 %gen('YAJLDTAGEN');
```

```
{
  "customer name": "string",
  "street address": "string",
  "city": "string",
  "state": "string",
  "postal": "string"
}
```

Anything prefixed by name_ controls the output name of the variable.

Questions?

- It's simple.
- But can be less simple (with lots of options) when needed.
- Assuming you need to generate a JSON document -- DATA-GEN lets you do that.
- Thank you.

Scott Klement - Handling JSON with DATA-INTO and DATA-GEN in ILE RPG

Please take the last minute of this session to complete the evaluation. A direct link to the evaluation can be found using the QR code below.