A Nerd's Guide to

DATA-INTO in RPG

Presented by

Scott Klement

http://www.profoundlogic.com

Two bytes meet. The first byte asks, "Are you ill?"
The second byte replies, "No, just feeling a bit off."

The Agenda

Agenda for this session:

1. What is DATA-INTO?
   - What is it?
   - Requirements
2. Calling DATA-INTO in RPG
   - Comparison to XML-INTO
   - Using it with Scott's YAJLINTO
3. Writing Your Own Parser
   - Why you'd want to?
   - How it works
   - Example

Background

Its often useful to read structured documents in business applications.

- Data interchange between business partners
- Data interchange between applications
- Simple way to store non-database data (such as configurations)

RPG's first foray into support for reading these was XML-INTO (which supported only XML)

The concept of XML-INTO: *think of your data as an RPG variable*

- data structure
- array
- data structure inside an array
- array inside a data structure
- ...or any combination of the above

Define an RPG variable in the same format as the XML, let XML-INTO do the rest.

XML-INTO Concept

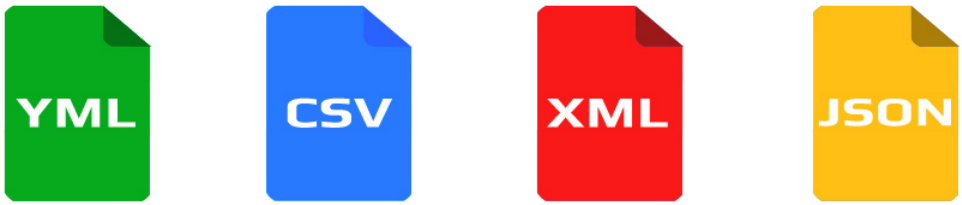
Think of XML like a data structure, it's one larger field (such as "address") that contains sub-fields (such as "street", "city", "state" and "postal")

It'd be helpful to be able to load the RPG DS from the XML.

That's what XML-INTO does!

- Maps XML fields into corresponding DS fields
- Field names must match (special characters can be mapped into underscores if needed)
- Repeating elements can be loaded into arrays

What is DATA-INTO?



Big limitation to XML-INTO

- only works with XML!
- JSON has overtaken XML as most popular
- 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!*

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 XML document, or IFS path to the XML document.

%DATA options = optional parameter containing options passed to RPG to control the reading of the XML 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.

Requirements for DATA-INTO

DATA-INTO was:

- added to RPG in March 2018 (via PTF)
- available 7.2 and 7.3 via PTFs
- releases after 7.3 (7.4+) will include DATA-INTO at GA
- RDi version 9.6.0.2 or newer to avoid syntax errors

NOTE: Like all RPG features released after March 2008, it will show up as a syntax error in SEU. SEU is no longer viable for anything but legacy work!!

PTF information can be found here:
http://ibm.biz/data-into-rpg-opcode-ptfs

Installing support for DATA-INTO will include/update the QOAR library with copybooks and sample programs from IBM

RPG Does Not Have JSON Opcodes

Currently, JSON is the most widely used format in new web services

- It has displaced XML
- Its usage is still growing!

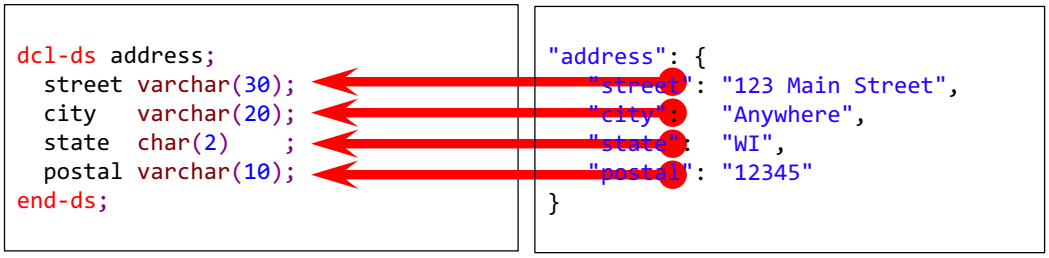
Since JSON is so popular, it is (currently) the most common use for DATA-INTO

- For that reason, Scott will use JSON as an example
- I will use the free (open source) YAJL JSON parser by Lloyd Hillael.
- YAJL is very fast, and is production quality (very robust)
- Scott provides a free (open source) DATA-INTO parser called YAJLINTO

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



DATA-INTO will do that when used with YAJLINTO (or similar)

- Aside from needing the 3rd party parser, it’s almost identical to XML-INTO
- Options like case=convert and countprefix work here as well

YAJLINTO Parser

Example of DATA-INTO with YAJLINTO as the Parser:

```
DATA-INTO result %DATA( '/tmp/example.json'
                      : 'doc=file case=any 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');
```


AllowMissing Option

```
dcl-ds address;
  street varchar(30);
  city   varchar(20);
  state  char(2)    ;
  postal_code varchar(10) inz('*NONE');
end-ds;

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

data-into address %DATA(myJSON:'case=convert allowmissing=yes')
                  %PARSER('YAJLINTO');
```

- You can specify more than one option by separating them with blanks.
- This example would fail (without allowmissing=yes) because the document is missing the "postal code"
- allowmissing may be yes or no. (default=no)
- the missing element is left unchanged (*NONE) in the RPG variable.

AllowExtra Option

```
dcl-ds address;
  street varchar(30);
  city   varchar(20);
  state  char(2)    ;
  postal_code varchar(10) inz('*NONE');
end-ds;

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

data-into address %DATA(myJSON:'case=convert allowextra=yes')
                  %PARSER('YAJLINTO');
```

- This example would fail (without allowextra=yes) because the document has an extra 'country' field that is not in the RPG code.
- allowextra may be yes or no. (default=no)

AllowMissing/AllowExtra Controversial?

- Some experts recommend that you never use these options
- If the data is missing, or extra data is found, you get no error.
- Could mean you have a coding error?
- Could coding errors go uncaught?

My take:

- Use it sparingly.
- Use countprefix instead, where it makes sense.
- However, don't avoid AllowMissing or AllowExtra when they will save you a lot of time.

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 vs. AllowMissing

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 many cases this is a better solution!

You can detect more errors this way!

However, when there are *many* optional fields, or when *new optional fields* might be added (where you won't know the names in advance) AllowMissing and AllowExtra are still useful.

PATH Option: The Problem

The PATH option lets you process only part of a document. It has no default value (the default is to process the entire document.)

However – there's a problem doing that in JSON because the document node is never assigned a name!!

The document node is the JSON element that the rest of the document is inside.

```
{
  "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" }
  ]
}
```

The document node is the { and } elements that the rest is inside.

It is "unnamed", so cannot be used in a path.

Coding path=statement/invoices won't work! There is no "statement" element!

PATH Option: Solution

YAJLINTO has a work-around. You can set the name of the document node in the %PARSER options, then use *that* name in the path.

```
dcl-ds invoices qualified dim(999);
  invoice char(5);
  amount  packed(9: 2);
  date    char(10);
end-ds;

data-into invoices  %DATA( 'statement.json'
                         : 'doc=file case=convert path=statement/invoices')
                    %PARSER('YAJLINTO'
                           :'{ "document_name": "statement" }');
```

NOTE: This problem is exclusive to JSON and similar formats. XML, for example, does assign a name to its document-level tag. (Other formats may as well, depending on the format.)

IBM may fix this in a future update?

Counting the Outermost Element

Countprefix can only be used elements within a data structure.

- What if the outermost element (document node) is an array?
- What if an element pointed to by PATH= is an array?

Since you cannot insert a prefixed field into a DS in that case, RPG will place the count in position 372 of the Program Status Data Structure.

```
dcl-ds pgmStat psds;
  numElements int(20) pos(372);
end-ds;
 .
 .       (data-into and other code from last slide goes here)
 .
for x = 1 to numElements;
   prinvn = invoices(x).invoice;
   prdamt = invoices(x).amount;
   prsdat = invoices(x).date;
   write prrec;
endfor;
```


%PARSER Options

The previous slide illustrated using the "document_name" option to YAJLINTO. This was a %PARSER option – an option specified on the %PARSER BIF.

```
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 can differ with each parser you use!

%PARSER Options:

- Can be coded as a string literal (as in the document_name example). In this case, they are passed to the parser as a pointer to null-terminated (C-style) string.
- Or can be an RPG variable. In this case, the parser gets a pointer to that variable.
- It is up to the parser to determine 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 (see PATH option on prior slides)
- 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*

But…. DATA-INTO Isn't Just for JSON!!

- ✓ It is much easier to explain DATA-INTO if I can show you examples.
- ✓ To show you examples, I need an example PARSER
- ✓ Since JSON is the most common document to use with DATA-INTO, and YAJLINTO is the best JSON parser available, I used it as an example.

But DATA-INTO can be used for just about anything!

In addition to these examples, you can find:

- IBM provides JSONPARSE for JSON *(not as good as YAJLINTO)*
- IBM provides PROPPARSE for property file format
- Jon Paris has written a CSV example
- Profound Logic has one (for a fee) that does both XML and JSON
- I will also show you a CSV of my own
- *Plus whatever else you can dream up!*

Debugging the Parser

IBM provides a special environment variable to assist you with using DATA-INTO. It traces all of the information passed into DATA-INTO from the parser. (Parsers can add additional information as well.)

To enable it for your job:

```
ADDENVVAR ENVVAR(QIBM_RPG_DATA_INTO_TRACE_PARSER) VALUE(*STDOUT)
```

Example output:

```
---------------- Start ---------------
Data length 886 bytes
Data CCSID 13488
Document name "statement" has been added to path
ReportName:  'statement'
Converting data to UTF-8
Allocating YAJL stream parser
Parsing JSON data (yajl_parse)
StartStruct
 ReportName:  'customer'
 ReportValue:  '5406'
 ReportName:  'statement date'
 ReportValue:  '2018-10-05'
 ReportName:  'start date'
 ReportValue:  '2018-09-01'
```

Writing Your Own Parser

This is for the real nerds out there! (ahem, like Scott)

Imagine what you could do if you wrote your own parser!!

Why?
- Write your own JSON one because it's fun.
- Write one for a different document type – where no other option exists
- Add cool features that don't already exist!

Ideas:
- Maybe autodetect XML or JSON, and handle either one.
- Parser that fetches data from https:// URLs before parsing
- Just about anything, really…. E-mail? Spreadsheet?
- Less limiting than Open Access because not limited to a 32k flat record.

DATA-INTO Flow Overview

DATA-INTO calls the parser program.

- DATA-INTO reads the document from a variable (doc=string) or a stream file (doc=file) into memory
- DATA-INTO calls the parser with parameters
  - Including the document and options
- Parser interprets the document and calls subprocedures that tell DATA-INTO what data to load into RPG variables.
  - Start of document (QrnDiStart)
  - End of document (QrnDiFinish)
  - Variable name (QrnDiReportName or QrnDiReportNameCCSID)
  - Variable value (QrnDiReportValue or QrnDiReportValueCCSID)
  - Start/End of DS (QrnDiStartStruct and QrnDiEndStruct)
  - Start/End of Array (QrnDiStartArray and QrnDiEndArray)
- When parser ends, "calling RPG" can continue

DATA-INTO Exceptions

NOTE:

Any of the events on the preceding slide can cause DATA-INTO to end the parser program if an error is detected.

- i.e. control won't return to the parser after calling a subprocedure.
- You must ensure that anything you allocated or opened has been cleaned up.
- Or provide a clean-up routine using something like ON-EXIT or an ILE cancel handler.

IMPORTANT:

Not cleaning up resources properly when RPG stops the parser is the most common mistake when writing a DATA-INTO parser.

Parser Generated Errors/Diagnostics

There are also two other subprocedures the parser can call:

- Report an error (QrnDiReportError)

  Parser processing ends immediately, the error is sent to the calling RPG.

- Write a message to the "trace" log (QrnDiTrace)

  This is typically used for diagnostic messages or other messages that might help with troubleshooting.

  The QIBM_RPG_DATA_INTO_TRACE_PARSER environment variable controls whether trace data is or isn't visible.

Writing a Parser: CSV Example

This example:

- Name is CSVINTO
- Reads a CSV file using my CSVR4 utility
- Routines in CSVR4 utility start with *CSV_*
- Loads the result into an array of data structures using DATA-INTO
- Routines that communicate with data-into begin with *QrnDi*

Note:

- CSV files don't have externally-defined names
- I used names field1, field2, field3, etc.

Writing a parser is a more "systems" style of programming using pointers, APIs, etc. This requires you to channel your inner nerd.

CSVINTO: The Caller

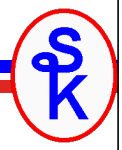
The goal of this parser will be to read a CSV file like this:

```
dcl-ds pgmStat psds;
  numRows int(20) pos(372);
end-ds;

dcl-ds CSV qualified dim(5000);
   field1 like(acct);    // acct
   field2 like(name);    // name
   field8 like(credLmt); // credit limit
end-ds;

data-into CSV %DATA( '/home/sklement/addrtest.csv'
                   : 'doc=file allowextra=yes')
              %PARSER('CSVINTO');

for x = 1 to numRows;
   acct = CSV(x).field1;
   name = CSV(x).field2;
   credLmt = CSV(x).field8;
   except rec;              // print data
endfor;
```

Only need fields 1, 2 and 8.

I can skip fields I don't need because I used AllowExtra

CSV Parser (1 of 5)



The goal of this parser will be to read a CSV file like this:

```
ctl-opt OPTION(*SRCSTMT:*NODEBUGIO: *NOSHOWCPY)
        BNDDIR('CSV') main(CSVINTO);

/copy QOAR/QRPGLESRC,QRNDTAINTO
/copy CSV_H

dcl-proc CSVINTO;

   dcl-pi *n;
     parm likeds(QrnDiParm_t);
   end-pi;

  ... other definitions here ...

   pQrnDiEnv = parm.env;

   QrnDiStart(parm.handle);
```

RPG linear main. Eliminates cycle, and allows the use of 'on-exit' in the mainline.

DATA-INTO always passes just one parameter.

It is a DS, defined in the (IBM supplied) QRNDTAINTO copybook

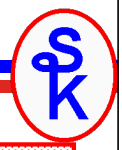
This sets up access to call the various *QrnDiXXXXX* procs.

Prototypes are also defined in QRNDTAINTO

Anytime we call the data-into procs, we must pass a "handle". RPG needs this in case you run multiple copies of DATA-INTO simultaneously within the same job.

QrnDiStart tells DATA-INTO that I started processing the document.

CSV Parser (2 of 5)



```
QrnDiTrace( parm.handle
          : 'Now opening CSV data from buffer'
          : *OFF );
monitor;
   h = CSV_openBuf( parm.data: parm.dataLen: parm.dataCCSID
                  : *omit: *omit: *omit );
on-error;
   h = *null;
endmon;

if h = *null;
   QrnDiTrace( parm.handle
             : 'Error opening CSV from buffer! (See job log)'
             : *OFF );
   errorNo = 1001;
   bytes   = 0;
   QrnDiReportError( parm.handle: errorNo: bytes);
   // DATA-INTO does not return control from QdiReportError!
endif;
```

QrnDiTrace can be called any time we want to log diagnostic info to the debug/trace data.

CSV_openBuf is a part of the CSVR4 tool to load a CSV file from memory (rather than IFS).

DATA-INTO read the doc for us, and passed it in the parm.data (pointer) and parm.dataLen (number) fields.

It also supplies the CCSID of the data as parm.dataCCSID. By default, RPG tries to normalize the data to UCS-2 (CCSID 13488)

QrnDiReportError will tell DATA-INTO that the parser encountered an error, and must stop.

We provide our own error number and bytes processed – it will include these in the diagnostics.

This procedure never returns control back to your program.

CSV Parser (3 of 5)

```
QrnDiStartArray( parm.handle );
row = 0;

// (this space intentionally left blank)


dow CSV_loadRec( h ) = *ON;

   row += 1;
   exsr load_fields;

enddo;

CSV_close(h);
h = *null;

QrnDiEndArray(parm.handle);
QrnDiFinish(parm.handle);
```

Each row should be reported to data into as an array. QrnDiStartArray tells DATA-INTO to start a new array.

CSV_loadRec loads one record from the CSV document into memory.

I am calling a subroutine to get the fields from that record and report them to data-into.

Once all records were processed, CSV_close cleans up CSVR4's dynamic memory allocations.

QrnDiEndArray ends the array started above.

QrnDiFinish tells DATA-INTO that the parser has finished processing the document.

CSV Parser (4 of 5)

```
begsr load_fields;

     QrnDiStartStruct(parm.handle);
     row += 1;

     QrnDiTrace( parm.handle
               : 'Processing row ' + %char(row)
               : *ON );


     fldno = 0;
     dow CSV_getfld(h: val: %size(val)) = *on;
        fldno += 1;
        name = 'field' + %char(fldno);
        QrnDiReportNameCCSID( parm.handle: %addr(name: *data): %len(name): 0);
        QrnDiReportValueCCSID( parm.handle: %addr(val: *data): %len(val): 0);
     enddo;

     QrnDiEndStruct(parm.handle);

endsr;
```

Each row is a data structure (within the array.) QrnDiStartStruct starts the data structure.

CSV_getFld gets one field from the CSV record.

QrnDiReportNameCCSID tells data-into the RPG field name to load into.

QrnDiReportValueCCSID tells it the value to load.

The data can be passed to DATA-INTO in any CCSID. CCSID=0 means "the job CCSID" (your systems' normal EBCDIC)

QrnDiEndStruct tells DATA-INTO we've finished the data structure.

CSV Parser (5 of 5)

```
on-exit;

   if h <> *null;
      CSV_close(h);
      h = *null;
   endif;

end-proc;
```

CSV_close (mentioned before) frees up dynamic memory used inside CSVR4

IMPORTANT:

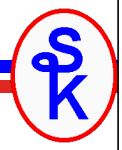
Any of the QrnDi functions can stop the processing.

- For example: if a field name isn't found in a DS
- ...or a value isn't valid (invalid number, etc)
- ...or various other reasons.

ON-EXIT code will still be called!

- This way we can make sure any temporary objects, memory, etc gets cleaned up.

More Information



PTF information for DATA-INTO support on IBM i 7.2 and 7.3
http://ibm.biz/data-into-rpg-opcode-ptfs

IBM's *Writing a Parser for the RPG DATA-INTO Operation Code:*
https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_73/rzasm/roaDataInto.htm

DATA-INTO operation code in the *ILE RPG Reference Manual*.
https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_73/rzasd/zzdatainto.htm

Articles in IBMSystems Magazine by Jon Paris & Susan Gantner:

XML-INTO Meets Open Access with RPG's New DATA-INTO
http://ibmsystemsmag.com/ibmi/developer/rpg/rpg-data-into/
A Closer Look at RPG's DATA-INTO
http://ibmsystemsmag.com/ibmi/developer/rpg/closer-look-data-into/
YAJL Meets DATA-INTO
http://ibmsystemsmag.com/ibmi/developer/rpg/yajl-meets-data-into/

More Information

From Scott Klement:

Scott's IBM i Port of YAJL (includes YAJLINTO)
https://www.scottklement.com/yajl/

Scott's CSVutil (includes CSVINTO, example report):
https://www.scottklement.com/csv/

Consuming Web Services on IBM i with HTTPAPI
http://www.scottklement.com/presentations/#HTTPAPI

Providing Web Services on IBM i
http://www.scottklement.com/presentations/#PROVIDING

Working with JSON in RPG with YAJL
http://www.scottklement.com/presentations/#YAJL

This Presentation

You can download a PDF copy of this presentation

http://www.scottklement.com/presentations/

Thank you!