"A computer once beat me at chess, but it was no match for me at kick boxing." — Emo Philips

The Agenda

Agenda for this session:

1. What is JSON?
   • Why use JSON?
   • Syntax Overview

2. The YAJL JSON reader/writer
   • Why YAJL?
   • Scott’s RPG interface

3. Generating JSON in RPG Code
   • Example

4. Reading JSON in RPG Code
   • Example
Ugggh, Another Thing to Learn!

This is pretty much how I felt about JSON at first!
- ugggh, I just learned XML. Do I need to learn something new?!
- But, as I learned more, I started to love it.
- Now I much prefer JSON over XML.

Much Like XML

JSON is a format for encapsulating data as it's sent over networks

Much Like XML

JSON is self-describing (field names are in the data itself) and human-readable.

Much Like XML

Very popular in Web Services and AJAX

Much Like XML

Can be used by all major programming languages

Much Like XML

So why is it better than XML.....?
**Much Different Than XML**

JSON is simpler:
- only supports UTF-8, whereas XML supports a variety of encodings.
- doesn't support schemas, transformations.
- doesn't support namespaces
- method of "escaping" data is much simpler.

JSON is faster
- more terse (less verbose). About 70% of XML's size on average
- simpler means faster to parse
- dead simple to use in JavaScript

JSON is quickly becoming important

Over 70% of all APIs in ProgrammableWeb's API directory are RESTful, increasingly at the expense of SOAP. More than 55% of those same APIs support JSON output, with 20% opting not to offer XML at all.

Source: 1 in 5 APIs Say "Bye XML", Adam DuVander, May 25, 2011
**JSON Evolved from JavaScript**

Originally JSON was the language used to describe "initializers" for JavaScript objects.

- Used to set the initial values of JavaScript Objects (data structures), and arrays. Even for arrays nested in data structures or vice-versa.
- Conceptually similar to "CTDATA" in RPG, except supports nested data as well.
- Unlike JavaScript, however, JSON does not support "methods" (executable routines in the object) so it's objects are equivalent to RPG data structures.

```javascript
var DaysOfWeek = [ "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday" ];
```

**JSON Syntax Summary**

Arrays start/end with square brackets
```
[ "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" ]
```

Objects (data structures in RPG) start/end with curly braces
```
{ x, x, x, x }
{ "first": "Scott", "last": "Klement", "sex": "male" }
```

Strings are in double-quotes. Quotes and control characters are escaped with backslashes. Numbers and true/false are not quoted.
```
{ "name": "Henry \"Hank\" Aaron", "home_runs": 755, "retired": true }
```

Names are separated from values with a colon (as above)

Successive elements (array elements or fields in an object) are separated by commas. (as above)

Data can be nested (arrays inside objects and/or objects inside arrays).
**JSON and XML to Represent a DS**

```
D list     ds  qualified
D custno   4p  0  dim(2)
D name     25a

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

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

For example, this is an array of a data structure in RPG.

This is how the same array might be represented (with data inside) in a JSON document.

And it’s approximately the same as this XML document.

---

**The YAJL Open Source Tool**

YAJL = Yet Another JSON Library
- Created by Lloyd Hilaiel (who works for Mozilla)
- completely Open Source (very permissive ISC license)
- Extremely fast. (Fastest one we benchmarked)
- Written in C.
- Bindings available for Ruby, Python, Perl, Lua, Node.js and others

Ported to IBM i (ILE C) by Scott Klement & David Russo.
- Available at [http://www.scottklement.com/yajl](http://www.scottklement.com/yajl)
- V5R4 or higher.
- Works entirely in UTF-8 Unicode

YAJLR4 = Scott’s ILE RPG language bindings
- Simplifies calling YAJL from ILE RPG
- Replaces C macros with RPG subprocedures
- Handles UTF-8/EBCDIC translation for you
**YAJL Provides**

YAJL provides sets of routines for:

- Generating JSON data
- Parsing JSON data in an event-driven (SAX-like) manner
- Parsing JSON in a tree (DOM-like) manner

I have found the tree-style routines to be easier to work with, so will use them in my examples.

**Example of Writing JSON**

For an example, an RPG program that lists invoices in a date range in JSON format, like this:

```json
{
    "success": true,
    "errmsg": "",
    "list": [
        {
            "invoice": "70689",
            "date": "03/01/2014",
            "name": "SCOTT KLEMENT",
            "amount": 14.80,
            "weight": 3.5
        },
        { another invoice },
        { another invoice },
        ...etc...
    ]
}
```
Example of Writing JSON

Or if an error occurs, it'd return an abbreviated document like this:

```json
{
"success": false,
"errmsg": "Error Message Here",
"list": []
}
```

To keep it simple, we'll just have it write the result to an IFS file.

Though, you can also use this in a web service, if desired (code download from ScottKlement.com will have an example of this)

---

RPG Writing JSON -- Definitions

```plaintext
H DFTACTGRP(*NO) ACTGRP('KLEMENT') OPTION(*SRCSTMT)
H BNDDIR('YAJL') DECEDIT('0. ')
/include yajl_h
```

```
D row             ds                  qualified
D inv             5a
D date             8s 0
D name             25a
D amount           9p 2
D weight           9p 1
D cust             4s 0 inz(4997)
D sdate            8s 0 inz(20100901)
D edate            8s 0 inz(20100930)
D dateUSA          10a varying
D success          1n
D errMsg           500a varying
```

Numbers in JSON must start a digit (not the decimal point).
The BNDDIR and copy book are needed to access YAJL's routines.
To keep example simple, query criteria is hard-coded.
RPG Writing JSON -- Mainline

exec SQL declare C1 cursor for
  select aiOrdn, aiIDat, aiSNme, aiDamt, aiLbs
  from ARSHIST
  where aiCust=:cust
    and aiIDat between :sdate and :edate;
exsr JSON_Start;
dow sqlstmt='00000' or %subst(sqlstmt:1:2)='01';
exsr JSON_AddRow;
exsr JSON_Finish;
exsr JSON_Save;
*inlr = *on;

Using SQL to get list of invoices from sales history file
At the start of the list, output JSON start (subroutine)
For each invoice found, add the 'row' data structure to JSON document
At the end of the list, call subroutines to finish the JSON data & save it.

YAJL Routines Used

To generate the JSON data we'll use the following YAJL procedures:

\texttt{yajl\_genOpen()} / \texttt{yajl\_genClose()} = Open/Close JSON generator.
The genOpen routine has a parm of *ON or *OFF tells whether JSON is "pretty" or "compact"

\texttt{yajl\_beginObj()} / \texttt{yajl\_endObj()} = start or end JSON object (data struct)

\texttt{yajl\_beginArray()} / \texttt{yajl\_endArray()} = start or end JSON array

\texttt{yajl\_addBool()} = add a boolean (true/false) value to JSON

\texttt{yajl\_addChar()} = add a character string to JSON

\texttt{yajl\_addNum()} = add a numeric value to JSON

\texttt{yajl\_saveBuf()} = write JSON document to IFS
**JSON_Start Routine**

```c
begsr JSON_Start;

  yajl_genOpen(*ON); // use *ON for easier to read JSON
   // *OFF for more compact JSON

  yajl_beginObj();
  yajl_addBool('success': success );
  yajl_addChar('errmsg': errMsg );
  yajl_beginArray('list');

endsr;
```

{  
  "success": false,
  "errmsg": "Error Message Here",
  "list": [
    
    {  
      "invoice": "XYX",
      "date": "12/31/2013",
      "name": "John Doe",
      "amount": 123.45,
      "weight": 100.5
    }
  
    ]
}

**JSON_addRow Routine**

```c
begsr JSON_addRow;

  dateUsa = %char( %date(row.date:*iso) : *usa );

  yajl_beginObj();
  yajl_addChar('invoice': row.inv );
  yajl_addChar('date': dateUsa );
  yajl_addChar('name': %trim(row.name));
  yajl_addNum('amount': %char(row.amount));
  yajl_addNum('weight': %char(row.weight));
  yajl_endObj();

endsr;
```

Each time this runs, it adds a new JSON element to the end of the document.
Since we have not yet called JSON_endArray(), each object is a new element in the array
that was started in the JSON_start subroutine.
```c
begsr JSON_Finish;
  yajl_endArray();
  yajl_endObj();
endsr;

begsr JSON_Save;
  yajl_saveBuf("/tmp/example.json": errMsg);
  if errMsg <> ' '
    // handle error
  endif;

  yajl_genClose();
endsr;
```

**Finish off the array and the object that began in JSON_start.**

**Save result to IFS file.**

**Close JSON generator (frees up memory).**

---

**RPG Writing JSON – "Pretty" Output**

```json
{
  "success": true,
  "errmsg": "",
  "list": [
    {
      "invoice": "70689",
      "date": "09/01/2010",
      "name": "JIM JOHNSON",
      "amount": 14.80,
      "weight": 3.5
    },
    {
      "invoice": "70695",
      "date": "09/01/2010",
      "name": "BILL VIERS",
      "amount": 9.80,
      "weight": 3.2
    }
  ]
}
```

**Result with yajl_genOpen("ON")**

("pretty" JSON data)

Includes line breaks and indenting to make it easy as possible for humans to read.

This extra formatting isn't needed for computer programs to read it, however.
RPG Writing JSON – "Compact" output

Result with yajl_genOpen("OFF")
  ("compact" JSON data)
No line breaks or indenting. Makes file size smaller, so it transmits over the network a little bit faster.
But, is the exact same document.

{"success":true,"errmsg":"","list": [{"invoice":"70689","date":"09/01/2010","name":"JIM JOHNSON","amount":14.80,"weight":3.5},
{"invoice":"70695","date":"09/01/2010","name":"BILL VIERS","amount":9.80,"weight":3.2}]}

What if I Wanted a Web Service?

Although there isn't time to go into detail about how to code RESTful web services in this presentation, the gist would be:

• Get input parameters from the URL.
• Create the JSON document in exactly the same way.
• Use JSON_copyBuf() instead of JSON_saveBuf()

JSON_copyBuf() copies the JSON data into a memory buffer or program variable instead of writing it to the IFS. You could then return it from your web service.

An example of this is provided in the sample code on Scott's web site, here:
http://www.scottklement.com/yajl
Reading JSON Data With YAJL

YAJL provides two ways of reading JSON data:
• event-based (like SAX in XML) APIs
• tree-based (like DOM in XML) APIs

This talk will discuss the tree-based method, since I have found it much easier to use.

A tree is the JSON data loaded into memory and organized in a hierarchical fashion.

Populating the YAJL tree

To load JSON data from IFS into the tree parser, call `yajl_stmf_load_tree()`, as follows:

```c
docNode = yajl_stmf_load_tree( '/tmp/example.json' : errMsg );
```

There is also `yajl_buf_load_tree()` if you prefer to load from a buffer or RPG variable.

The return value is a YAJL 'node' that represents the outermost element of the JSON document. (the tree's “trunk”)

A 'node' represents data at one level of the document, and can be used to retrieve 'child nodes' that are within the current 'node'.

(To understand better, see the diagram on the next slide.)
Diagram of a JSON Tree

When YAJL loads the JSON data into the tree, it gives me the document node (docNode).

Given any node, I can retrieve its "children". So with docNode, I can get the nodes for the 'success', 'errmsg' and 'list' elements.

Document Node

- success (true/false)
- errmsg (char string)
- list (array)

and with the node for 'list', I can get the array elements, etc.

Retrieving A "Child Node"

yajl_object_find() will get a child node by field name.

yajl_is_true() returns whether a true/false value is true.

yajl_is_false() returns whether a true/false value is false.

```c
// { "success": true }

succNode = yajl_object_find( docNode : 'success' );

if yajl_is_true( succNode );
    // success!
else;
    // failure
endif;
```
Get a String Value From a Node

\[
yajl\_get\_string() = \text{returns a string value from a node}
\]

// { "success": false, "errmsg": "invalid start date" }

succNode = yajl\_object\_find( docNode : 'success' );

if yajl\_is\_false( succNode );
    errMsgNode = yajl\_object\_find( docNode: 'errmsg' );
    msg = yajl\_get\_string(errMsgNode);
    // msg now contains "invalid start date"
endif;

For numeric values, there's also \textit{yajl\_get\_number()}

Processing an Array

\[
yajl\_array\_loop() = \text{loops through all elements in a JSON array}
\]

// { "list": [ invoice1, invoice2, invoice 3 ] } 

list = yajl\_object\_find( docNode : 'list' );

i = 0;

dow YAJL\_ARRAY\_LOOP( list: i: node );

    // code here is repeated for each array element.
    // each time through, node and i are updated
    // to point to reflect the current array element.
enddo;

\[
yajl\_array\_elem() \text{(not demonstrated here) can be used if you prefer to get each element by it's array index number.}
\]
Processing an Object (DS)

`yajl_object_loop()` loops through all sub-fields in an object, and returns the field name ("key"), child node ("val") and index for each.

This is, equivalent to calling `yajl_object_find()` separately for each field name.

```c
// { "invoice": 123, "name": "Scott Klement", "amount": 100.00 }
i = 0;
dow YAJL_OBJECT_LOOP( docNode: i: key: val );
   // code here is repeated for each field in the object
   // each time through, key, val and i are updated
   // to point to reflect the current field
endo;
```

Freeing Up Resources (When Done)

When `yajl_stmf_load_tree()` ran, all of the JSON details were loaded into memory. To free up that memory, you must call `yajl_tree_free()`.

```c
    yajl_tree_free( docNode );
```

You must pass the document node into `yajl_tree_free()`, so be sure to save it when you call `yajl_xxxx_load_tree()`.

`yajl_tree_free()` will free up all of the child nodes as well as the document node. So be sure that you do not refer to any of the nodes after calling it.
To put together all of the YAJL tree concepts shown in the preceding slides, I have provided an RPG example.

- Reads the same JSON file (from IFS) that we created earlier
- Loads the JSON data into an RPG data structure.
- After all is loaded, loops through and prints the data (just to demonstrate reading)

```rpg
H DFTACTGRP(*NO) ACTGRP('KLEMENT') OPTION(*SRCSTMT)
H BNDDIR('YAJL')

#include yajl_h

D list_t ds qualified 
D                      template
D     inv 5a
D     date 8s 0
D     name 25a
D     amount 9p 2
D     weight 9p 1
D result ds qualified 
D     success 1n
D     errmsg 500a varying
D     list likeds(list_t) dim(999)
D i s 10i 0
D j s 10i 0
D dateUSA s 10a
D errMsg s 500a varying inz('')
```

The 'result' data structure will be populated from the JSON data.
Variables that represent JSON nodes are defined as 'yajl_val'

Technically, under the covers, these are pointers to the data structures that YAJL uses internally.

However, there's no need for the RPG program to be concerned with how it works, and it's not necessary to do any sort of pointer logic on these fields. They are just placeholders for the JSON nodes.

```rpg
// load the example.js document into a tree.
docNode = yajl_stmf_load_tree( '/tmp/example.json' : errMsg );
if errMsg <> '';
   // handle error
endif;

// get the 'success' field into 'result' DS
// result.success is an RPG named indicator, and will be *ON if success=true, *OFF if success=false
node = YAJL_object_find(docNode: 'success');
result.success = YAJL_is_true(node);

// get the 'errmsg' field into 'result' DS
node = YAJL_object_find(docNode: 'errmsg');
result.errmsg = YAJL_get_string(node);
```
list = YAJL_object_find(docNode: 'list');
i = 0;
dow YAJL_ARRAY_LOOP( list: i: node );
  j = 0;
dow YAJL_OBJECT_LOOP( node: j: key: val);

  // when 'load_subfield' is run, "key" will contain
  // the JSON field name, and "val" will contain
  // a YAJL node from which the value can be extracted

  exsr load_subfield;
  enddo;
enddo;

begsr load_subfield;

select;
when key = 'invoice';
  result.list(i).inv = yajl_get_string(val);
when key = 'date';
  dateUSA = yajl_get_string(val);
  result.list(i).date = %dec(%date(dateUSA:*usa):*iso);
when key = 'name';
  result.list(i).name = yajl_get_string(val);
when key = 'amount';
  result.list(i).amount = yajl_get_number(val);
when key = 'weight';
  result.list(i).weight = yajl_get_number(val);
endsr;
RPG Reading JSON (6 of 6)

Just for the sake of having some output, here's a quick & dirty routine to print the invoice list (with O-specs)

```
D prt    ds    likeds(list_t)

  .
  .
  for i = 1 to YAJL.ARRAY_SIZE(list);
  prt = result.list(i);
  except print;
  endfor;
  .

QSYSRPT E PRINT
  0   PRT.INV  5
  0   PRT.DATE 17    - - -
  0   PRT.NAME 44
  0   PRT.AMOUNT L 56
  0   PRT.WEIGHT L 67
```

RPG Reading JSON -- Output

The output of the program would look as follows:

```
70689 2010-09-01 JIM JOHNSON 14.80 3.5
70695 2010-09-01 BILL VIERS  9.80 3.2
70700 2010-09-01 JOSE MENDOZA  6.00 3.0
70703 2010-09-01 RICHARD KERBEL 10.00 5.0
70715 2010-09-01 JACKIE OLSON 23.80 10.0
70736 2010-09-01 LISA XIONG  24.00 7.0
70748 2010-09-01 JOHN HANSON  11.80 5.0
70806 2010-09-01 JOHN ESSLINGER  7.50 5.0
70809 2010-09-01 LORI SKUZENSKI  20.00 1.0
70826 2010-09-02 KURT KADOW  11.25 7.0
70926 2010-09-02 PENNY STRAW  25.00 5.0
70979 2010-09-02 WOLSKI STEVE  12.75 .0
71021 2010-09-02 KENNETH HALE  21.25 5.9
71062 2010-09-02 ALEX AGULIERA  10.00 2.0
71081 2010-09-03 JIM JOHNSON  41.50 13.5
71270 2010-09-03 DAVE DRESEN  11.90 3.5
```
This Presentation

You can download YAJL and the sample code presented in this session from:
http://www.scottklement.com/yajl

You can download a PDF copy of this presentation from:
http://www.scottklement.com/presentations/

Thank you!