"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 with DATA-INTO
   • Example with YAJL subprocedures
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 Has Mostly Replaced XML

Have you noticed that people are rarely discussing XML anymore?
- Google, Facebook, Twitter, IBM Watson focus on JSON
- JSON has become the most popular for REST APIs
- JSON has become the de-facto standard for Internet of Things (IoT)
- XML is still used, but mainly in pre-existing applications. Rarely in new projects.

**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 its 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
  ```javascript
  [ "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" ]
  ```

- Objects (data structures in RPG) start/end with curly braces
  ```javascript
  { "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.
  ```javascript
  { "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

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.

Without Adding Spacing for Humans

In this simple "textbook" example, that's a 35% size reduction.

50 bytes doesn't matter, but sometimes these documents can be megabytes long – so a 35% reduction can be important.

…and programs process JSON faster, too!
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
- IBM i 6.1 or higher (7.2 for DATA-INTO)
- 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.

Scott's RPG adapter additionally provides
- YAJLINTO – a DATA-INTO interface for reading JSON
- YAJLDTAGEN – a DATA-GEN generator for creating JSON

DATA-INTO requires IBM i 7.2+ w/PTFs (7.4+ without PTFs)
DATA-GEN will be released for IBM I 7.3+ in November 2019
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

```rpg
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 s 4s 0 inz(4997)
D sdate s 8s 0 inz(20100901)
D edate s 8s 0 inz(20100930)
D dateUSA s 10a varying
D success s 1n
D errMsg s 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;
exec SQL open C1;
exec SQL fetch next from C1 into :row;
exsr JSON_Start;
dow sqlstt='00000' or %subst(sqlstt:1:2)='01';
exsr JSON_AddRow;
exec SQL fetch next from C1 into :row;
enddo;
exec SQL close C1;
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:

- `yajl_genOpen() / yajl_genClose()` = Open/Close JSON generator.
  - The genOpen routine has a parm of "ON" or "OFF" tells whether JSON is "pretty" or "compact"

- `yajl_beginObj() / yajl_endObj()` = start or end JSON object (data struct)

- `yajl_beginArray() / yajl_endArray()` = start or end JSON array

- `yajl_addBool()` = add a boolean (true/false) value to JSON

- `yajl_addChar()` = add a character string to JSON

- `yajl_addNum()` = add a numeric value to JSON

- `yajl_saveBuf()` = write JSON document to IFS
JSON_Start Routine

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": [
        
        yajl_beginObj
        yajl_addBool
        yajl_addChar
        yajl_beginArray

        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();

        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 YAJL_endArray(), each object is a new element in the array that
was started in the JSON_start subroutine.

JSON_addRow Routine

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;

{  
    "invoice": "XYX",
    "date": "12/31/2013",
    "name": "John Doe",
    "amount": 123.45,
    "weight": 100.5
}


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

Save result to IFS file

Close JSON generator (frees up memory)

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.

```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}]
}
```

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 `YAJL_writeStdout()` instead of `YAJL_saveBuf()`

`YAJL_writeStdout()` writes the JSON data to standard output with HTTP headers, as would be needed if writing your own web service provider to be run through the IBM HTTP Server (powered by Apache.)

For consuming web services, you can use `YAJL_copyBuf()` or `YAJL_copyBufStr()` which returns the JSON data into a buffer (pointer) or RPG string so that you can pass it to HTTPAPI or another HTTP tool to send it.

Examples are provided in the sample code on Scott's web site, here: [http://www.scottklement.com/yajl](http://www.scottklement.com/yajl)
DATA-INTO is an RPG opcode that was added to IBM i 7.2 and newer releases.

The following link describes the PTFs needed for DATA-INTO support on 7.2 and 7.3 releases:

http://ibm.biz/data-into-rpg-opcode-ptfs

YAJL supports DATA-INTO as of the April 2018. (But, get the latest copy with the latest enhancements!)

DATA-INTO is supported with a program named YAJLINTO that works with the RPG %PARSER function.

What is DATA-INTO?

- RPG opcode that maps data into an RPG data structure
- Almost exactly the same as XML-INTO, but for other types of data
- Works with a 3rd party external parser (YAJLINTO in this case) that interprets the document.
- With the right parser, should be able to read just about any type of document. YAJLINTO is designed for JSON documents
- Fields are mapped by their name
- RPG field names must match the JSON field names to work
- Various options are provided, but I cannot cover them all here. See the ILE RPG Reference for details.
**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 is an advanced topic I will not cover today.

**Data Structure Must Match**

The trickiest part is that the DS must match the JSON document.

```rpg
  dcl-ds result qualified; // {
    success ind; // "success": true,
    errmsg varchar(500); // "errmsg": "Error message",
    num_list int(10);
  }
  dcl-ds list dim(999); // "list": [ {
    invoice char(5); // "invoice": "xxxxx",
    date char(10); // "date": "xx/xx/xxxx",
    name char(25); // "name": "xxxxxxxxxx"
    amount packed(9: 2); // "amount": "xx.xx",
    weight packed(9: 1); // "weight": "xxx.x",
  end-ds; // } ]
}
```

field names must match, objects must match a data structure, arrays must match an array.
Example of DATA-INTO with YAJLINTO as the Parser:

```rpg
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.

---

**YAJLINTO Example (1 of 2)**

```rpg
**free
c1-opt DFTACTGRP(*NO) OPTION(*SRCSTMT) BNDDIR('YAJL');
dcl-f QSYSPRT printer(132);
/include yajl_h
dcl-ds result qualified;
  success ind;
  errmsg varchar(500);
  num_list int(10);

dcl-ds list dim(999);
  invoice char(5);  
  date char(10);   
  name char(25);   
  amount packed(9:2); 
  weight packed(9:1);
end-ds;
end-ds;
```
**YAQLINTO Example (2 of 2)**

```cl
DCL-DS PRINTME LEN(132) END-DS;

DCL-S I INT(10);
DCL-S DATEISO DATE(*ISO);

DATA-INTO RESULT %DATA('/tmp/example.json'
    : 'doc=file case=any countprefix=num_'
    %PARSER('YAQLINTO');

FOR I = 1 TO RESULT.NUM_LIST;
    DATEISO = %DATE(RESULT.LIST(I).DATE:*USA);
    PRINTME = RESULT.LIST(I).INVOICE + ' ' + %CHAR(DATEISO:*ISO) + ' ' + RESULT.LIST(I).NAME + ' ' + %EDITC(RESULT.LIST(I).AMOUNT:'L') + ' ' + %EDITC(RESULT.LIST(I).WEIGHT:'L');
    WRITE QSYSPRNT PRINTME;
ENDFOR;

*INLR = *ON;
```

**YAQLINTO Output**

The output of the program would look as follows (goes to the spool, I didn't take the time to add headings, etc)

```
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
70748 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
70892 2010-09-02  PENNY STRAW          25.00   5.0
70979 2010-09-02  WOLSKI STEVE         12.75   0.0
71021 2010-09-02  KENNETH HALE          21.25   5.9
71062 2010-09-02  ALEX AGUILERA        10.00   2.0
71081 2010-09-03  JIM JOHNSON           41.50  13.5
71270 2010-09-03  DAVE DRESEN          11.90   3.5
```
Data-Into from a Web Service

If you need to read JSON from a web service, the JSON may be provided to you in two ways:
• some tools provide JSON as a string (usually parameter) to your program
• some tools (such as the IBM HTTP server (powered by Apache)) send the data via “standard input”

To read data sent in a character string, use doc=string (just as you would with XML-INTO)

```plaintext
data-into result %DATA(
  myJsonString :
    'doc=string case=convert countprefix=num_')
%PARSER('YAJLINTO');
```

Since September 2018, YAJLINTO supports direct reading from standard input by passing the special value *STDIN. This makes it easy to get input via Apache.

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

Using YAJL’s Tree Method

As mentioned earlier, 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.

Advantages over DATA-INTO:
• Works on older releases (6.1+)
• has more capabilities (pointers, generate, generate from tree nodes)
• the RPG document doesn't have to match the JSON document

Disadvantages:
• trickier to learn/code
• uses more memory
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()` and `yajl_string_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 it's "children". So with `docNode`, I can get the nodes for the 'success', 'errmsg' and 'list' elements.

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

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

```
array index 1 (object)
  invno (char string)
  date (char string)
  name (char string)
  amount (number)
  weight (number)
```

```
array index 2 (object)
  invno (char string)
  date (char string)
  name (char string)
  amount (number)
  weight (number)
```
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.

```plaintext
// { "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()` = returns a string value from a node

```plaintext
// { "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 `yajl_get_number()`
Processing an Array

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

```c
// { "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.
endo;
```

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

Processing an Object (DS)

\[\text{yajl\_object\_loop()} = \text{loops through all sub-fields in an object, and returns the field name ("key"), child node ("val") and index for each.}\]

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.

---

Reading JSON – RPG Example

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)
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.
// load the example.json 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);
endsl;
endsr;

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

<table>
<thead>
<tr>
<th>D</th>
<th>prt</th>
<th>ds</th>
<th>likeds(list_t)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>for i = 1 to YAJL_ARRAY_SIZE(list);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>prt = result.list(i);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>except print;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>endfor;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>OQS</td>
<td>E</td>
<td>PRINT</td>
</tr>
<tr>
<td></td>
<td>SYSPRT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>PRT.INV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>PRT.DATE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>PRT.NAME</td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>PRT.AMOUNT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>PRT.WEIGHT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The output of the program would look as follows:

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

What About Web Service Input?

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.
- Load the input document with YAJL_stdin_load_tree()

YAJL_stdin_load_tree() reads JSON data from standard input, and returns the document node. If you are writing a web service provider called from Apache, you can use it in place of YAJL_stmf_load_tree() to get the data from Apache instead of from a file.

There is also a routine called YAJL_buf_load_tree() for loading JSON data from a buffer or variable instead of a file.

Examples are provided in the sample code on Scott's web site, here: [http://www.scottklement.com/yajl](http://www.scottklement.com/yajl)
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!