JSON and RPGLE on IBM I.

By Henrik Rützou, April 2012.

In the resent years JSON has become a more and more used format to exchange data between servers and web 2.0 browser based clients or APP's.

What is JSON and why and where is it used.

JSON stands for Javascript Object Notation and is like XML a hierarchical data format. It is primarily used when client exchange data with the server but also used in OO javascript to pass complex parameters to functions. Since JSON is the internal data format in data objects in javascript it can be passed and processed without any transformation and thereby much faster than other formats including XML and since it also is a very lightweight format the size of JSON objects are normally much smaller than a similar XML document saving both bandwidth and memory.

Hierarchical data format.

Similar to XML, JSON has nodes that consist of parent, child and siblings. And like XML, JSON always has a root element that may either be an object or an array.

Unlike XML that only has one element type defined by <aaa> … </aaa> JSON has several types of elements:

- Objects and arrays defines the hierarchical structure in a notation like
  - **{ … object node elements …}**
  - **[… array node elements …]**
- Data elements that can be either string, number, boolean or null where strings always are imbedded in a notation like "… data …", numbers always are passed without thousands separator comma as '.' and minus sign in front.

UTF-8.

JSON has only one CCSID and that is UTF-8. UTF-8 shares x'00'-x'7F' with ASCII but any character > x'7F' must be encoded into 2-4 bytes UTF-8 codeunits. On IBM I this isn't really a problems since JSON normally only exists as IFS based files (that supports UTF-8) or is passed through the APACHE server that can be set up to automatically convert data from any EBCDIC CCSID to UTF-8 by adding the server directive DefaultNetCCSID 1208 to apaches httpd.conf file.

Special characters.

Like XML, JSON also has special, but different, characters that needs to be encode/decoded in strings. The encoding rules can be read on http://www.json.org .

Since strings (which isn't documented) may be imbedded in either single quotes or double quotes the similar characters in the string must be encoded if the if it equals the imbedding character.

In regards to RPGLE it is much easier to use the "standard" imbedding character (double quote) than the single quote character because of this character special meaning in the RPGLE syntax.

The separator and structured JSON.

JSON node elements are separated with a comma. If the JSON is compressed and has a complex hierarchical structure it is very difficult to spot errors in or even just read the structure. So it is important to have a mechanism that creates structured properly indented code that reflects the hierarchical structure.

Unstructured JSON:

```
[{NAME : "Anderson",ADDRESS : "Thestreet 123, first floor",ZIP :
1234,PHONE:"+45 1234 5678"},{NAME : "Anderson",ADDRESS : "Thestreet 123,
first floor",ZIP : 1234,PHONE : "+45 1234 5678"}]
```

The separator may be placed in front of or succeed the node element. Personally I prefer the separator in front for much faster readability but many places it in the end, it all depends of how LEAN you are.

Structured JSON:

```
[
          {
                    ID : "Separator In Front"
                    ,NAME : "Anderson"
                    ,ADDRESS : "Thestreet 123, first floor"
                    ,ZIP : 1234
                    ,PHONE : "+45 1234 5678"
          }
          ,{
                    ID : "Separator In Back",
                    NAME : "Anderson",
                    ADDRESS : "Thestreet 123, first floor",
                    ZIP : 1234,
                    PHONE : "+45 1234 5678"
          }
]
```

Note: In the structured JSON your brain can read the four commas visually in the first object without moving your eyes but it can't read the same commas in the second object without moving your eyes which means that while the first object can be processed visually by the brain, the brain has to read and thereby process the code in the second object to spot the commas.

But don't take my word for it, try to read the above JSON yourself!

The compressed JSON is hardly readable without a tool that structures it.

JSON and RPGLE.

RPGLE doesn't have %BIF'S to create neither XML nor JSON and most coding examples are based on a concatenation of the XML/JSON format with fields with data.

```
myJSON = '[{';
myJSON += 'NAME : "' + %trimr(MYNAME) + '"';
myJSON += ',ADDRESS : "' + %trimr(MYADDRESS) + '"';
. . .
```

This method does however in most cases not handle the creation of structured code or handle the encoding of special characters and it also limits the size of the JSON object.

It is important to notice that JSON that just has one wrong separator or one non encoded character just stops working – there are no warnings or errors because most JSON requests are based on AJAX handlers.

JSON and RPGLE with Open Source.

There are a couple of Open Source projects that supply JSON support for RPGLE. One is RPGNEXTGEN that can be found on http://www.rpgnextgen.com/ another is powerEXT Core that can be found on http://www.powerEXT.com .

Since I'm the author of powerEXT Core I will in the following show some examples of how easy it is to make JSON in RPGLE with powerEXT JSON Node and/or Template Support:

Inline JSON created with powerEXT Node support.

powerEXT Core supports a variety of formats such as XML, HTML, JSON, CSV in simple to use sub procedures called Node support.

There are two sub procedures that controls generation and encoding of JSON: jsonNode(); and jsonEndNode();

jsonNode(); has three parameters, nodetype (object, array, string etc.), element name and data.

JsonEndNode(); is a structural subprocedure, with on optional parameter, that either ends an object or an array or defaults to the current type in the hierarchical tree.

Behind the curtains jsonNode(); and jsonEndNode(); keeps track of the hierarchical tree and provides the insertions of relevant CRLF and TABS in the code unless told otherwise – and most important – they also keep track of separators. The node support also automatically handles the encoding of special characters of strings.

So if the above example should be "hardcoded" in powerEXT/RPGLE it would look like

```
jsonNode('*array');
  jsonNode('*object');
    jsonNode('*string':'ID':'Separator In Front');
    jsonNode('*string':'NAME':'Anderson');
    jsonNode('*string':'ADDRESS':'Thestreet 123, first floor');
    jsonNode('*number':'ZIP':'1234');
    jsonNode('*string':'PHONE':'+45 1234 5678');
  jsonEndNode();
  jsonNode('*object');
    jsonNode('*string':'ID':'Separator In Back');
    jsonNode('*string':'NAME':'Anderson');
    jsonNode('*string':'ADDRESS':'Thestreet 123, first floor');
    jsonNode('*number':'ZIP':'1234');
    jsonNode('*string':'PHONE':'+45 1234 5678');
  jsonEndNode();
jsonEndNode();
```

What these sub procedures does is that they builds the JSON OBJECT in a terabyte capable memory storage and afterwards this storage can either be written to an IFS file or send to a HTTP client by other build in sub procedures:

```
echoToStmf('/myJSON.json':1208); // save JSON to disk
echoToClient();                  // send JSON to a HTTP requester
myAddr = bufAddr();              // returns the %addr of the buffer
mySize = bufSize();              // returns the %size of the buffer
```

You may also notice that the above programming, even though it is procedural, has the same hierarchical structure as the tree structure that it reflects, which makes it very easy and fast to "mind debug".

Templated JSON with powerEXT templates and encode support.

Since powerEXT is built on the top of CGIDEV2 it is of course also possible to use templates to do the same job and even mix node support and templates and formats:

IFS Template File: /myCustomertmp.txt (<powerEXT> or whatever as cutomerized section tag)

```
<powerEXT>customerJSON
          ID : "/%custid%/"
          ,NAME : "/%custname%/"
          ,ADDRESS : "/%custaddr%/"
          ,ZIP : /%custzip%/
          ,PHONE : "/%custPhone%"
<powerEXT>customerXML
          <ID>/%custid%/</ID>
          <NAME>/%custname%/</NAME>
          <ADDRESS>/%custaddr%/</ADDRESS>
          <ZIP>/%custzip%/</ZIP>
          <PHONE>/%custPhone%/</PHONE>
```

powerEXT/RPGLE to create JSON:

```
jsonNode('*array');
  jsonNode('*object');
    setExtVar('custid':encodeJSON('Separator In Front'));
    setExtVar('custname':encodeJSON('Anderson'));
    setExtVar('custaddr':encodeJSON('Thestreet 123, first floor'));
    setExtVar('custzip':'1234');
    setExtVar('custPhone':'+45 1234 5678');
    echoCgi('/myCustomertmp.txt':'customerJSON');
  jsonEndNode();
  jsonNode('*object');
    setExtVar('custid':encodeJSON('Separator In Back'));
    setExtVar('custname':encodeJSON('Anderson'));
    setExtVar('custaddr':encodeJSON('Thestreet 123, first floor'));
    setExtVar('custzip':'1234');
    setExtVar('custPhone':'+45 1234 5678');
    echoCgi('/myCustomertmp.txt':'customerJSON');
  jsonEndNode();
jsonEndNode();
```

Reading JSON in RPGLE with powerEXT.

Reading JSON in RPGLE isn't simple; JSON has however the same hierarchical data structures as XML and therefore JSON can be transformed into XML.

powerEXT has an advanced build in procedural XML reader and a subprocedure that can transform JSON to XML:

jsonToXML(myJsonAddr:myJsonSize); that transforms the above JSON to XML in the following format:

```
<array depth="1">
 <object depth="2">
  <ID>Separator In Front</ID>
  <NAME>Anderson</NAME>
  <ADDRESS>Thestreet 123, first floor</ADDRESS>
  <ZIP>1234</ZIP>
  <PHONE>+45 1234 5678</PHONE>
 </object>
 <object depth="2">
  <ID>Separator In Back</ID>
  <NAME>Anderson</NAME>
  <ADDRESS>Thestreet 123, first floor</ADDRESS>
  <ZIP>1234</ZIP>
  <PHONE>+45 1234 5678</PHONE>
 </object>
</array>
```

The transformed JSON can then be read quite easily with the XML build in reader:

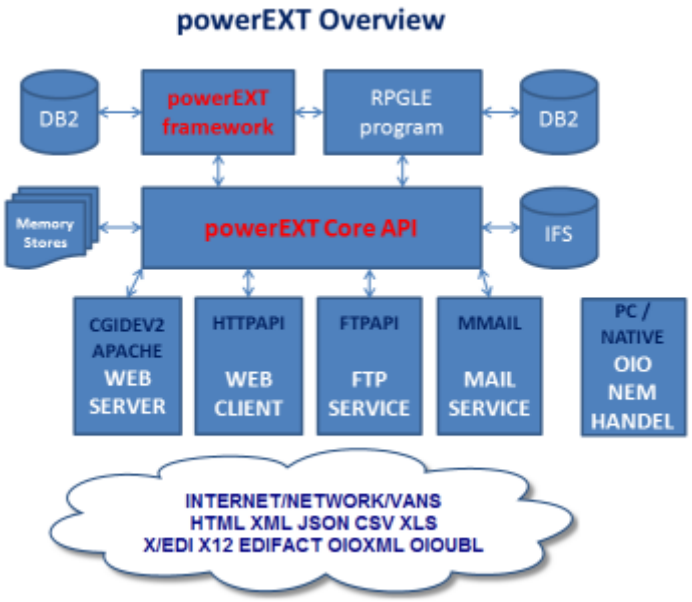
```
dow xmlReader = 0;
  select;
    when xmlGetNode = 'ID' and xmlGetAttr = '';
      myid  = xmlGetData;
    when xmlGetNode = 'NAME' and xmlGetAttr = '';
      myname =  xmlGetData;
    when xmlGetNode = 'ADDRESS' and xmlGetAttr = '';
      myaddress  =  xmlGetData;
   when xmlGetNode = 'ZIP' and xmlGetAttr = '';
      myzip = xmlGetData;
   when xmlGetNode = 'PHONE' and xmlGetAttr = '';
      myphone = xmlGetData;
  endsl;
enddo;
```

About powerEXT Core

powerEXT Core is an FREE Open Source service program with 100+ sub procedures that are designed to support creating and reading popular formats such as HTML, XML, JSON CSV.

For a complete list of sub procedures see http://powerEXT.com/PXAPICGI.htm

It works with several other Open Source projects on IBM I and it is also the basic service program for powerEXT Framework made for 5250 free modernization and 5250 free developing on IBM I:



powerEXT Core can be downloaded from the download area on http://powerEXT.com