PLJSON Library

The PLJSON library is a unit that allows you to do three things:

- Parse a JSON document into a PL/SQL object tree for easy traversal
- Create a PL/SQL object tree to make a corresponding JSON document.
- Create a JSON document in a couple of formats from a ref-cursor. This library supports Object Types, Nested Tables, nested cursors, and ANYDATA data.

This library is divided into three layers:

- A PL/SQL layer that you can see and have access to.
- The GSON library which is the underlying engine.
- The Java “glue” that translates between PL/SQL and GSON.

Part 1 – Dealing with JSON in PL/SQL

JSON defines a few particulars (refer to http://json.org for details):

- Value: This can be a primitive type (string, number, Boolean), an Object, an Array, or a NULL. A string is identified by surrounding double-quotes. A number does not have quotes and may be scientific-notated. Boolean is either the literal *true* or *false*. A NULL uses the literal *null*.
- Object: this is a collection of one or more `"name":value` pairs (herein referred to as tuple in this document). The value could be a primitive type, a NULL value, an Array, or another Object. The name must be a string type. An Object is denoted by an opening and closing brace: "{" and "}". Each tuple is separated by a comma.
- Array: this is an ordered list of zero or more values. The value could a primitive type, a NULL value, another Array, or an Object. An Array is denoted by an opening and closing bracket: "[" and "]". Each item in the array is separated by a comma. The values need not be the same type for each element in the array.
- Strings have certain escape sequences, but you need not worry about that; the GSON library takes care of translating these for you.

JavaScript is a weakly typed and fully dynamic language, and that concept is diametrically opposed to the nature of the PL/SQL language. This makes the translation to/from the two languages somewhat difficult. The attempt of this library was to make this as graceful as possible, but there are a couple of items to keep in mind.

- Do not instantiate the PLJSON objects using normal constructors. Instead, use the PLJSON package functions "createXxxx". This is partly because of the following point:
- Do not attempt to use or modify the "~" attribute of the `pljsonElement` type. It is *intended* to be private/abstract, but SQL Types must have at least one public attribute, even if they are abstract. This attribute is used internally in the Java layer of this library, so modifying it will create problems for you.

Part 2 - The SQL Types

The SQL Types are defined as follows:

PLJSONELEMENT (`pljsonElement`)

This is the root type for all JSON objects that can be instantiated. It has one attribute that should not be referenced by your code, and therefore has that unusual attribute name. This type is abstract and cannot be instantiated.

`pljsonElement` has a number of member methods:

```
  member function isObject    return boolean,
  member function isArray     return boolean,
  member function isPrimitive return boolean,
  member function isNull      return boolean,
  member function isString    return boolean,
  member function isNumber    return boolean,
  member function isBoolean   return boolean,

  member function getString   return varchar2,
  member function getNumber   return number,
  member function getBoolean  return boolean
```

Note that `getString`, `getNumber`, and `getBoolean` are considered convenience methods and will return "best-effort" for primitive types. `getString` will return a string-ified value for a number, and for Boolean, the character "*" (asterisk/star) for a true value and a " " (space) for a false value. If a type can't be readily translated, NULL is returned. For example, a call to `getNumber` for a Boolean type will return NULL.

PLJSONNULL (`pljsonNull`)

Subtype of `pljsonElement`. This is the most reasonable way to represent that a particular JSON value is actually NULL. It has no additional attributes or methods.

PLJSONOBJECT (`pljsonObject`)

Subtype of `pljsonElement`. This is the most complex of the types. It is defined as follows:

```
  tuple  pljsonObjectEntries,

  member function getIndex
    ( aName  in varchar2 )
    return binary_integer,

  member function getMember
    ( aName  in  varchar2 )
    return pljsonElement,

  member procedure addMember
    ( aName  in  varchar2,
      element in pljsonElement ),

  -- convenience methods to quickly add primitives
  member procedure addMember
    ( aName   in  varchar2,
      element in  varchar2 ),
```

```
member procedure addMember
  ( aName   in  varchar2,
    element in  number ),

member procedure addMember
  ( aName   in  varchar2,
    element in  boolean )
```

Using *tuple* is the best way to represent a dynamic list of members, and it is a nested table `pljsonObjectEntries`, which is a table of `pljsonObjectEntry`, which is defined as:

```
name    varchar2(4000),
value   pljsonElement
```

In general, you shouldn't need to access the tuple directly, but you can if you like. You can treat it like any other nested table object. If you want to iterate through the table, be aware that it could be "sparse" and so you want to use the appropriate iteration method. If that link doesn't work, go here:
`http://docs.oracle.com/cd/E11882_01/appdev.112/e25519/composites.htm#BEIBJDBF`

The one method that you don't see is "removeMember". This is because of a particular limitation of PL/SQL reassigning the tuple back to itself inside the object. But you can delete out of the tuple as follows:

```
obj.tuple.delete(obj.getIndex('memberName'));
```

Where `obj` is your object variable. Hopefully you won't need to do this much.

The convenience methods create the correct primitive type to be used in the tuple.

PLJSONARRAY (pljsonArray)

Subtype of `pljsonElement`. This represents a JSON array, and is defined as follows:

```
elements  pljsonElements,

member procedure addElement
  ( element in pljsonElement ),

-- convenience methods to quickly add primitives
member procedure addElement
  ( element in  varchar2 ),

member procedure addElement
  ( element in  number ),

member procedure addElement
  ( element in  boolean )
```

The *elements* in this object is a `pljsonElements` which is simply a table of `pljsonElement`. Like `pljsonObject`, you can iterate through *elements* using the proper technique. As well, to delete from *elements*, use the same method described above.

The convenience methods create the correct primitive type to be used.

PLJSONPRIMITIVE (`pljsonPrimitive`)

Subtype of `pljsonElement`. This is simply the base type of the three primitive types. It is not instantiable (abstract), has no additional methods, and is typically never referenced in your code. It is here to provide structure.

PLJSONSTRING (`pljsonString`)

Subtype of `pljsonPrimitive`. It has one additional attribute:

```
value    varchar2(32000 char)
```

Note that strings stored here will be properly escaped when serialized in JSON outout. When a JSON document creates this object, the string is restored "un-escaped".

PLJSONNUMBER (`pljsonNumber`)

Subtype of `pljsonPrimitive`. It has one additional attribute:

```
value    number
```

When serialized in JSON output, the number will always be represented in non-scientific notation.

PLJSONBOOLEAN (`pljsonBoolean`)

Subtype of `pljsonPrimitive`. It has one additional attribute and one method:

```
val    varchar2(1),    -- anything other than '*' is false
member function value
  return boolean       -- but use this function to be sure
```

You should use the `value` method in your code, or use the `getBoolean` method from `pljsonElement`. When serialized in JSON output, this primitive is properly converted to use the words ***true*** and ***false*** appropriately in the document.

Part 3 - The PLJSON Package

This package is quite simple.

- First, it implements the three main tasks mentioned at the beginning of this document.
- Second, it has the "constructors" for the objects you may need to create.
- Third, it has conversion tools to allow you to convert one object type to another.

The three main tasks:

```
function parseJson
  ( json  in  CLOB )
  return pljsonElement;
```

This function takes a JSON document and returns the root element. The element could either be an object or an array. This is the slowest operation you will encounter, especially for large documents, since the document must be parsed, translated into the GSON object tree, then translated into the PL/SQL object tree. Some databases appear to work better than others, but it hasn't been determined what makes them better or worse.

```
function makeJson
  ( pljson in pljsonElement,
    pretty in boolean default false )
  return CLOB;
```

This function takes a root element (an object or an array) and returns a JSON document. Specify *pretty* to get the output in a more human-readable format.

```
function refCursorToJson
    ( input    in sys_refcursor,
      compact  in boolean  default false,
      rootName in varchar2 default 'json',
      pretty   in boolean  default false,
      dateFmt  in varchar2 default 'yyyy-MM-dd HH:mm:ss' )
    return CLOB;
```

This function processes an open ref-cursor and turns it into a JSON document. This process is done entirely in Java, and bypasses the PL/SQL object creation mechanism, so it is quite fast. This uses the GSON serializer directly to the result CLOB, so large datasets should not be an issue.

The JSON document that is returned always starts with an object with one member, specified by *rootName*. That member is always an array.

{"json":[...]}
From there, you have a choice of formats: Normal, and Compact. The normal format returns an array of objects. Each object is one row of the cursor. Each object contains all of the field values of the row. If a column is NULL, the JSON object will be serialized as NULL.

Given the following query:

```
select cast(dbms_random.string('a',12) as varchar2(30)) as "varChar2",
       round(cast(dbms_random.value(1, 100000) as number), 7) as num,
       cast(sysdate+dbms_random.value(-1000, 1000) as date) as dt
  from dual connect by level < 5
```

We get the JSON document as follows:

```
{ "json": [
    { "varChar2": "xEPnPeRCmuOv",
      "NUM": 21349.6088119,
      "DT": "2010-07-21 02:58:16" },
    { "varChar2": "JQrGaYJlgvbm",
      "NUM": 49904.2727362,
      "DT": "2013-09-11 06:02:49" },
    { "varChar2": "SXijRbUUMghW",
      "NUM": 40663.8263266,
      "DT": "2015-01-10 07:05:42" },
    { "varChar2": "hXOvoqnZGAMz",
      "NUM": 97087.044715,
      "DT": "2012-08-14 13:33:33" }
  ]
}
```

A few things to note: If you specify a case-sensitive label (x AS “something”), the object attribute will reflect that. See how “varChar2” is labeled. Also note, that dates are serialized as strings based on the *dateFmt*. *dateFmt* is a Java `SimpleDateFormat` format string. Refer to the Java documentation for the elements in the format string.

Specifying compact returns the data in a slightly more compact fashion:

```
{ "json": [
    [ "varChar2", "NUM", "DT" ],
    [ "GDRXvWZmQahh", 81298.3390885, "2013-01-21 11:40:43" ],
    [ "qefRYoMzzHwu", 76503.384392, "2014-08-25 10:29:59" ],
    [ "rsEflMMgFiPe", 69963.8523149, "2012-10-29 03:03:20" ],
    [ "hmLJmYyhzhra", 36927.5054318, "2010-04-21 11:18:28" ]
  ]
}
```

This results in an array of arrays, where the first array contains the column labels, and the subsequent arrays contain the column data.

As noted at the beginning of this document, objects, nested tables, and nested cursors are supported. Given the following query:

```
select cast(dbms_random.string('a',12) as varchar2(30)) "varChar2",
       round(cast(dbms_random.value(1, 100000) as number), 7) num,
     cursor(
   select cast(dbms_random.string('a',10) as varchar2(10)) "subField1",
          round(dbms_random.value(1, 100000000000), 4) "subNumber2"
     from dual connect by level < 4) crsr
  from dual connect by level < 5
```

The normal JSON looks like:


```
{ "json": [
    { "varChar2": "UjfDosEGMNBm",
      "NUM": 93428.8585544,
      "CRSR": [
        { "subField1": "XnDcBzoHLp",
          "subNumber2": 35653935772.037 },
        { "subField1": "DQOQOlKxhv",
          "subNumber2": 91540154112.4731 },
        { "subField1": "vMMppIKxrx",
          "subNumber2": 22582327522.4681 }
      ]
    },
    { "varChar2": "GbYeoKgowuDH",
      "NUM": 48212.818676,
      "CRSR": [
        { "subField1": "kMNkiqZGiJ",
          "subNumber2": 75010070169.8976 },
        { "subField1": "yevTKNgdGv",
          "subNumber2": 74665741685.6596 },
        { "subField1": "kdslLwfFXZ",
          "subNumber2": 59021183992.4713 }
      ]
    },
    { "varChar2": "FXqxLFudaHtF",
      "NUM": 13884.848568,
      "CRSR": [
        { "subField1": "QAwGorsHUt",
          "subNumber2": 49869676584.8504 },
        { "subField1": "rfWDVlRtdi",
          "subNumber2": 79279821361.3168 },
        { "subField1": "dxZhNvxgNa",
          "subNumber2": 75694323505.735 }
      ]
    },
    { "varChar2": "NLtnDNPYaXHt",
      "NUM": 15641.151077,
      "CRSR": [
        { "subField1": "ulkLDVBWgm",
          "subNumber2": 35259647752.7328 },
        { "subField1": "HgOXdqJnMD",
          "subNumber2": 54007481353.2732 },
        { "subField1": "phKClYGiHo",
          "subNumber2": 21291801388.0795 }
      ]
    }
  ]
}
```


And the compact JSON looks like:


```
{ "json": [
    [ "varChar2", "NUM", "CRSR" ],
    [ "AXTRVAPeSvEb", 43237.6318831,
      [ [ "subField1", "subNumber2" ],
        [ "ElfrRjwTgh", 90999917723.145 ],
        [ "OCHDtvWXdG", 63444436634.8586 ],
        [ "FNNwfrESep", 2308724396.9671 ]
      ]
```

```
    ],
    [ "gwJZgItHYbsD", 30981.9794112,
      [ [ "subField1", "subNumber2" ],
        [ "hwzeZKMdSx", 133901885.8896 ],
        [ "gPFBAFWKTO", 18405286462.1886 ],
        [ "ShKjVCaIhO", 83805533569.5954 ]
      ]
    ],
    [ "ZnlLyWTBBABy", 84937.3157598,
      [ [ "subField1", "subNumber2" ],
        [ "OhhOGYpVdm", 93433700854.0266 ],
        [ "LOOBBBYhnI", 67318870030.6331 ],
        [ "rfVgATtrKM", 64340538077.558 ]
      ]
    ],
    [ "AHHnAvixMTaX", 46866.7575258,
      [ [ "subField1", "subNumber2" ],
        [ "NeMZpazxQZ", 45895664191.2099 ],
        [ "gjMktjORQG", 43920800115.6553 ],
        [ "jOhiAQzhKA", 51185780635.0626 ]
      ]
    ]
  ]
}
```

We will leave it as an exercise to the reader to try objects and nested tables. However, since objects and nested tables cannot easily be “compacted”, they are always presented as “normal”.

Lastly, we have constructors and converters:

Constructors:

```
  function createObject    return pljsonObject;
  function createArray     return pljsonArray;
  function createNull      return pljsonNull;
  function createString  ( val in varchar2 ) return pljsonString;
  function createNumber  ( val in number )   return pljsonNumber;
  function createBoolean ( val in boolean )  return pljsonBoolean;
```

These should be self-explanitory. In general, you should not change the value of a primitive after it has been created. Instead, simply construct a new primitive.

After creating an Object or an Array, you add members to the object, and add elements to the array using the member methods.

Converters:

```
  function getObject  ( e  in pljsonElement ) return pljsonObject;
  function getArray   ( e  in pljsonElement ) return pljsonArray;
  function getString  ( e  in pljsonElement ) return varchar2;
  function getNumber  ( e  in pljsonElement ) return number;
  function getBoolean ( e  in pljsonElement ) return boolean;
```

These also should be self-explanitory. Note that if you pass an invalid type to a converter, an exception will be thrown.

Part 4: Example Code

The following complete script can be run in SQL*Plus. Copy it into a text file and run it in a database that has PLJSON installed. Notice on the Object and Array demos that you can select from the items (tuple or elements). Or in the last example, you can programmatically look for a particular item.

```
column name format a40
column val format a60
var rc refcursor
var clb  clob
set head off

-- demonstrate an object with a bunch of string members
declare
  object    pljsonObject;
begin

  object := pljson.createObject();
  object.addMember('aaa', 'this is a test');
  object.addMember('bbb', 'of the emergency broadcast system');
  object.addMember('ccc', 'this is only a test');
  object.addMember('ggg', 'if this were an actual emergency');
  object.addMember('fff', 'you would have been instructed');
  object.addMember('zzz', 'this shouldn''t be here');
  object.addMember('eee', 'to put your head between your knees');
  object.addMember('ddd', 'and kiss your a** goodbye');

  -- deletes have to go like this
  object.tuple.delete(object.getIndex('zzz'));

  -- select member data from the object
  open :rc for select p.value.getString() val
               from table(object.tuple) p
               where p.name in ('ggg','aaa');

  -- make the JSON
  :clb := pljson.makeJson(object, true);

end;
/

print rc

select :clb from dual;


-- demonstrate an array of string elements
declare
  vArray pljsonArray;
begin
  vArray := pljson.createArray();
  vArray.addElement('this is a test');
  vArray.addElement('of the emergency broadcast system');
  vArray.addElement('this is only a test');
  vArray.addElement('had this been an actual emergency');
  vArray.addElement('you would have been instructed');
  vArray.addElement('to put your head between your knees');
  vArray.addElement('and kiss your a** goodbye');
```

```
  :clb := pljson.makeJson(vArray, true);

  open :rc for select value(p).getString() val from table(vArray.elements) p;

end;
/

print rc

select :clb from dual;

var json varchar2(4000)
begin
  :json := q'!
{"web-app": {"servlet": [{"servlet-name": "cofaxCDS",
      "servlet-class": "org.cofax.cds.CDSServlet","init-param": {
        "configGlossary:installationAt": "Philadelphia, PA",
        "configGlossary:adminEmail": "ksm@pobox.com",
        "configGlossary:poweredBy": "Cofax",
        "configGlossary:poweredByIcon": "/images/cofax.gif",
        "configGlossary:staticPath": "/content/static",
        "templateProcessorClass": "org.cofax.WysiwygTemplate",
        "templateLoaderClass": "org.cofax.FilesTemplateLoader",
        "templatePath": "templates",
        "templateOverridePath": "",
        "defaultListTemplate": "listTemplate.htm",
        "defaultFileTemplate": "articleTemplate.htm",
        "useJSP": false,
        "jspListTemplate": "listTemplate.jsp",
        "jspFileTemplate": "articleTemplate.jsp",
        "cachePackageTagsTrack": 200,
        "cachePackageTagsStore": 200,
        "cachePackageTagsRefresh": 60,
        "cacheTemplatesTrack": 100,
        "cacheTemplatesStore": 50,
        "cacheTemplatesRefresh": 15,
        "cachePagesTrack": 200,
        "cachePagesStore": 100,
        "cachePagesRefresh": 10,
        "cachePagesDirtyRead": 10,
        "searchEngineListTemplate": "forSearchEnginesList.htm",
        "searchEngineFileTemplate": "forSearchEngines.htm",
        "searchEngineRobotsDb": "WEB-INF/robots.db",
        "useDataStore": true,
        "dataStoreClass": "org.cofax.SqlDataStore",
        "redirectionClass": "org.cofax.SqlRedirection",
        "dataStoreName": "cofax",
        "dataStoreDriver": "com.microsoft.jdbc.sqlserver.SQLServerDriver",
        "dataStoreUrl":
"jdbc:microsoft:sqlserver://LOCALHOST:1433;DatabaseName=goon",
        "dataStoreUser": "sa",
        "dataStorePassword": "dataStoreTestQuery",
        "dataStoreTestQuery": "SET NOCOUNT ON;select test='test';",
        "dataStoreLogFile": "/usr/local/tomcat/logs/datastore.log",
        "dataStoreInitConns": 10,
        "dataStoreMaxConns": 100,
        "dataStoreConnUsageLimit": 100,
        "dataStoreLogLevel": "debug",
        "maxUrlLength": 500}},{"servlet-name": "cofaxEmail",
      "servlet-class": "org.cofax.cds.EmailServlet",
```

```
      "init-param": {"mailHost": "mail1",
      "mailHostOverride": "mail2"}},{"servlet-name": "cofaxAdmin",
      "servlet-class": "org.cofax.cds.AdminServlet"},{"servlet-name":
"fileServlet",
      "servlet-class": "org.cofax.cds.FileServlet"},
    { "servlet-name": "cofaxTools",
      "servlet-class": "org.cofax.cms.CofaxToolsServlet",
      "init-param": {
        "templatePath": "toolstemplates/",
        "log": 1,
        "logLocation": "/usr/local/tomcat/logs/CofaxTools.log",
        "logMaxSize": "",
        "dataLog": 1,
        "dataLogLocation": "/usr/local/tomcat/logs/dataLog.log",
        "dataLogMaxSize": "",
        "removePageCache": "/content/admin/remove?cache=pages&id=",
        "removeTemplateCache": "/content/admin/remove?cache=templates&id=",
        "fileTransferFolder":
"/usr/local/tomcat/webapps/content/fileTransferFolder",
        "lookInContext": 1,
        "adminGroupID": 4,
        "betaServer": true}}],
  "servlet-mapping": {
    "cofaxCDS": "/",
    "cofaxEmail": "/cofaxutil/aemail/*",
    "cofaxAdmin": "/admin/*",
    "fileServlet": "/static/*",
    "cofaxTools": "/tools/*"},  "taglib": {
    "taglib-uri": "cofax.tld",
    "taglib-location": "/WEB-INF/tlds/cofax.tld"}}}!';
end;
/

-- demonstrate parsing and traversing a JSON tree
declare
  root   pljsonElement;

  t1 timestamp;
  t2 timestamp;

  -- forward declarations
  procedure processElement(e pljsonElement, lvl binary_integer);
  procedure parseObject(o pljsonObject, lvl binary_integer);
  procedure parseArray(a pljsonArray, lvl binary_integer);

  -- helper
  procedure doOutput (o varchar2, lvl binary_integer)
  is
  begin
    dbms_output.put_line(lpad('  '||o, length(o)+lvl+2, '-'));
  end doOutput;

  -- an object contains a "tuple", which is a pljsonObjectEntries nested
table
  -- each element in the tuple is a pljsonObjectEntry object, which is a
name/value pair
  -- the name is a string, the value is a pljsonElement object
  procedure parseObject(o pljsonObject, lvl binary_integer)
  is
    i   binary_integer;
    oe  pljsonObjectEntry;
```

```
  begin
    doOutput('OBJECT START', lvl);

    -- demonstrate looking for a particular member of an object
    declare
      v1  pljsonElement;
    begin
      v1 := o.getMember('jspFileTemplate');
      if v1 is not null then
        dbms_output.put_line('**** FOUND IT **** '||v1.getString());
      end if;
    end;

    -- demonstrate traversing the members of an object
    i := o.tuple.first;
    while i is not null
    loop
      oe := o.tuple(i);
      doOutput('object entry: '||oe.name, lvl);
      processElement(oe.value, lvl+1);
      i := o.tuple.next(i);
    end loop;

    doOutput('OBJECT END', lvl);
  end parseObject;

  -- an array contains elements which is a pljsonElements nested table
  -- each element in the array is a pljsonElement object
  procedure parseArray(a pljsonArray, lvl binary_integer)
  is
    i binary_integer;
  begin
    -- demonstrate traversing the elements of an array
    doOutput('ARRAY START', lvl);
    i := a.elements.first;
    while i is not null
    loop
      processElement(a.elements(i), lvl+1);
      i := a.elements.next(i);
    end loop;
    doOutput('ARRAY END', lvl);
  end parseArray;

  procedure processElement(e pljsonElement, lvl binary_integer)
  is
  begin
    case when e.isNull      then doOutput('NULL', lvl);
         when e.isPrimitive then doOutput('primitive: '||e.getString(), lvl);
         when e.isObject    then parseObject(pljson.getObject(e), lvl);
         when e.isArray     then parseArray(pljson.getArray(e), lvl);
    end case;
 end;

begin
  t1 := systimestamp;
  root := pljson.parseJson(:json);
  t2 := systimestamp;

  doOutput('time to parse: '||to_char(t2-t1), 0);

  t1 := systimestamp;
```

```
  processElement(root, 0);
  t2 := systimestamp;

  doOutput('time to process: '||to_char(t2-t1), 0);

  t1 := systimestamp;
  :clb := pljson.makeJson(root, false);
  t2 := systimestamp;

  doOutput('time to stream: '||to_char(t2-t1), 0);
end;
/

select :clb from dual;

exit
```